THE TECHNICAL UNIVERSITY OF ŁÓDŹ
Faculty of Electrical and Electronic Engineering

Master of Engineering Thesis

DESIGN OF RADIATION TOLERANT TRANSMISSION CHANNEL CIRCUIT

Jakub Mielczarek

Student's number: 106119

Supervisor:
Grzegorz Jabłoński, PhD

Auxiliary supervisor:
Dariusz Makowski, MSc

Łódź, 2005

Streszczenie

Niniejsza praca magisterska została wykonana dzięki współpracy Politechniki Łódzkiej z ośrodkiem Deutsches Elektronen-Synchrotron (DESY) w ramach programu Coordinated Accelerator Research in Europe (CARE). Ośrodek ten mieści się w Hamburgu, w Niemczech. DESY specjalizuje się w badaniach dotyczących fizyki wysokich energii i cząstek elementarnych. Mieszczą się tam również laboratoria biologii molekularnej, chemii i materiałoznawstwa, wykorzystujące w badaniach promieniowanie synchrotronowe. Ośrodek DESY został wybrany jako lokalizacja nowej generacji akceleratora cząstek. International Linear Collider (ILC) jest konstruowany i budowany w ramach międzynarodowej współpracy ośrodków badawczych i naukowych. Akcelerator ten umożliwi przyspieszanie elektronów i pozytonów do energii rzędu 1 TeV. Będzie to możliwe dzięki zastosowaniu niedawno opracowanej i jeszcze testowanej technologii Teraelectronvolt Superconducting Linear Accelerator (TESLA). Technologia ta opiera się na nadprzewodzących wnękach rezonansowych wykonanych z czystego niobu. Fala elektromagnetyczna o częstotliwości 1,3 GHz przyspiesza elektrony i pozytony. W końcowym etapie przyspieszania cząstki są poddawane kolizji. Produkty kolizji być może pozwolą na zweryfikowanie istnienia bozonu Higssa. Strumień wysokoenergetycznych elektronów posłuży również jako źródło promieniowania w laserze rentgenowskim X-Ray Free Electron Laser (X-FEL). To źródło koherentnego promieniowania rentgenowskiego będzie w stanie wytwarzać impulsy o czasie trwania rzędu 80 femtosekund, pozwalając na obserwację m.in. drgań struktur krystalicznych czy kolejnych etapów podczas zachodzenia reakcji chemicznych.

Akcelerator ILC będzie kontrolowany przez zaawansowane, głównie cyfrowe, systemy sterowania. Systemy te oparte zostaną o urządzenia programowalne FPGA i procesory sygnałowe DSP. Akcelerator został zaprojektowany z wykorzystaniem tylko jednego tunelu. Będzie on mieścił zarówno tor przyspieszający wykonany ze wspomnianych wnęk rezonansowych, jak i wymienione systemy sterowania. Takie rozwiązanie sprawi, iż urządzenia elektroniczne będą narażone na wpływ szkodliwego promieniowania gamma i neutronów. Wynikiem tego mogą być awarie urządzeń elektronicznych, polegające m.in. na nieprzewidywalnych zatrzymaniach pracy modułów, ich funkcjonowaniu w sposób niezgodny z zaprogramowanym lub trwałe uszkodzenia tychże urządzeń. Dlatego też, ważnym aspektem projektu ILC jest ilościowe zbadanie

i opisanie potencjalnych, negatywnych skutków wpływu promieniowania gamma i neutronów na urządzenia elektroniczne. Co więcej, konieczne jest opracowanie sposobów łagodzenia tychże skutków lub zapobiegania im.

Część praktyczna wykonana w ramach niniejszej pracy magisterskiej koncentruje się na zaprojektowaniu i wykonaniu niezawodnego, niewrażliwego na pewne skutki promieniowania neutronowego, układu cyfrowego. Głównym zadaniem powyższego układu jest umożliwienie i obsługa niezawodnej komunikacji pomiędzy różnymi urządzeniami elektronicznymi, znajdującymi się w tunelu akceleratora, a centralnym systemem, znajdującym się poza wpływem promieniowania. Potrzeba opisanej komunikacji rodzi się z dwóch powodów: konieczności poddania urządzeń elektronicznych testom w środowisku promieniowania gamma i neutronowego, oraz monitorowania poziomu promieniowania w tunelu akceleratora. Układ został zaprojektowany, wykonany, uruchomiony i poddany testom w tunelu akceleratora LINAC II w ośrodku DESY. Testy odbyły się na przełomie sierpnia i września 2005 roku. Na projekt układu złożyło się wiele etapów. Praca zawiera opis każdego z nich. Przed przystąpieniem do fazy projektowania, przestudiowano wpływ promieniowania gamma i neutronów na układy elektroniczne. Następnie przeprowadzono analizę dostępnych układów programowalnych pod kątem wrażliwości na promieniowanie, kosztów i możliwości wielokrotnego programowania. Do implementacji układu niewrażliwego na promieniowanie wybrano układ programowalny FPGA oparty na wbudowanej pamięci Flash. W układzie FPGA został zaimplementowany mikrokontroler oparty o rdzeń procesora PIC16C57, który jest szeroko wspieranym standardem. Do opisu układu cyfrowego użyto języka VHDL. Na część praktyczną pracy złożyło się kilka etapów. Pierwszym było zaprojektowanie i zmontowanie płytek drukowanych. Kolejnym, wspomniany opis układu w języku VHDL. Ostatnim etapem było napisanie oprogramowania na zaimplementowany mikrokontroler. Dzięki temu oprogramowaniu, wykonane urządzenie może funkcjonować jako detektor zjawisk Single Event Upset (SEU) w pamięci SRAM. Detektor taki może zostać wykorzystany w zautomatyzowanym systemie pomiaru strumienia neutronów w tunelu akceleratora. Dodatkowo powstało oprogramowanie na komputer PC, pozwalające na przetestowanie wykonanego urządzenia, podczas gdy pełniło ono wspomnianą funkcję detektora. Podczas 72-godzinnego testu urządzenie niezawodnie pracowało w obecności promieniowania gamma i neutronowego. W tym czasie bezbłędnie realizowało zaprogramowaną funkcję detektora SEU w pamięci SRAM. Komunikacja z komputerem PC, umieszczonym poza tunelem akceleratora przebiegała również bezbłędnie.

Mikrokontroler zaimplementowany w układzie FPGA został częściowo uodporniony na promieniowanie wyłącznie na poziomie języka VHDL. Do tego celu zostały użyte kody Hamminga i potrójna redundancja. Metoda uodparniania oparta na potrójnej redundancji została sformalizowana w postaci algorytmu. Pozwala ona na rozdzielenie procesu opisu układu cyfrowego od procesu uodpornienia na wpływ promieniowania neutronowego. Dzięki temu nie jest wymagana ingerencja w kod źródłowy VHDL. Ograniczeniem opracowanej metody jest wsparcie wyłącznie dla układów FPGA ProAsic Plus firmy Actel. Dalsze badania są niezbędne, aby zweryfikować jej realizowalność dla innych rodzajów układów FPGA.

1 Introduction

The present master's thesis was carried out in cooperation with Deutsches Elektronen-Synchrotron (DESY). DESY is a high-energy particle physics research centre, located in Hamburg, Germany. It also incorporates molecular biology and chemistry laboratories, taking advantage of synchrotron radiation. The centre has been chosen as a home site for a new generation particle collider, which will be designed and built in an international collaboration. The International Linear Collider (ILC) will accelerate electrons and positrons to energies of up to 1 TeV. The capability will be enabled by the newest, currently being tested in TESLA Test Facility 2 (TTF2) site in DESY, Tera Electronvolt Superconducting Linear Accelerator (TESLA) technology. In this technology, pure niobium, superconducting cavities fed with 1.3 GHz electromagnetic wave accelerate the electrons and positrons, bringing them into collisions. The products of collisions might verify the existence of the Higgs' particle, being considered the origin of matter's mass. The highly energetic streams of electrons will be also the basis for X-Ray Free Electron Laser (X-FEL). This coherent source of ca. 80-femtosecond X-ray pulses of high brilliance will facilitate the observation of crystalline structure oscillations and give empirical insight into subsequent steps of chemical reactions.

The ILC will be controlled by sophisticated, mostly digital, electronic control systems. The systems will be based on Field Programmable Gate Array (FPGA) devices, and Digital Signal Processors (DSPs). Since the main accelerator tunnel will confine both the accelerating cavities and the electronic control systems, the latter will be exposed to influence of mixed neutron and *Bremsstrahlung* gamma radiation. Both constituents of the mixed radiation have detrimental effects on electronic devices, ranging from inducing halts or infinite loops, functional interrupts to severe, permanent damage. The effects may be short or long term ones. Therefore, it is vital for the ILC project to quantify the damage effects and take measures against them.

This project concentrates on designing and implementing a reliable, radiation tolerant transmission channel circuit, enabling flexible and reliable communication between electronic devices located in the accelerator tunnel and central system located in radiation-free environment. The need of such communication channel arises from two necessities: testing electronic devices in the accelerator's radiation environment, monitoring neutron and/or gamma radiation level in the accelerator tunnel. The detailed requirements for the radiation tolerant device and its exemplary application are described in Chapter 2.

Before the design phase commenced, the influence of neutron and gamma radiation on electronic devices was thoroughly studied. The summary of the study can be found in Chapter 3.

The assumption was made, that the transmission channel circuit should be built based on Commercial Off-The-Shelf (COTS) components only. Therefore, as documented in Chapter 4, various families of suitable electronic devices were considered for the implementation and the optimal one has been chosen.

Investigation of already developed and well-tested techniques improving radiation tolerance of digital systems was carried out. The most valuable techniques are described in Chapter 5.

The transmission channel circuit was described in Very High Speed Integrated Circuit Hardware Description Language (VHDL) and implemented in a Flash-based FPGA device. The circuit is based on industry standard PIC16C57 CPU. Some of the mitigation techniques described in Chapter 5 were implemented to improve the radiation tolerance of the transmission channel circuit. During this process a semi-automatic technique for mitigating sequential designs on the VHDL level was developed and used in the project. A printed circuit board was designed and assembled to facilitate prototyping of the circuit. The board, together with designed and assembled auxiliary hardware was used during in-field tests. Finally, software for the CPU was developed, to demonstrate device's capabilities and test it in field. Chapter 6 gives the full description of the project.

The physically implemented transmission channel circuit, together with auxiliary hardware was subject to tests in DESY, in August and September 2005. The radiation tolerant device was installed in LINAC II accelerator, whose radiation environment is harsher than that of TTF2 and the planned ILC. The device was placed in close vicinity of the *electron-to-positron* converter, guaranteeing high doses of the mixed radiation reaching the device. Description of experiments, which were carried out and the obtained results are presented in Chapter 7.

2 Objective of the Thesis

The aim of this master's thesis is to design a radiation tolerant device, which will enable reliable exchange of information between accelerator tunnel and a monitoring station (e.g. personal computer) located outside the tunnel. Operation of the device will be demonstrated in a useful application.

This chapter gives details on the origins of the necessity of such design, and briefly describes the main idea behind the solution.

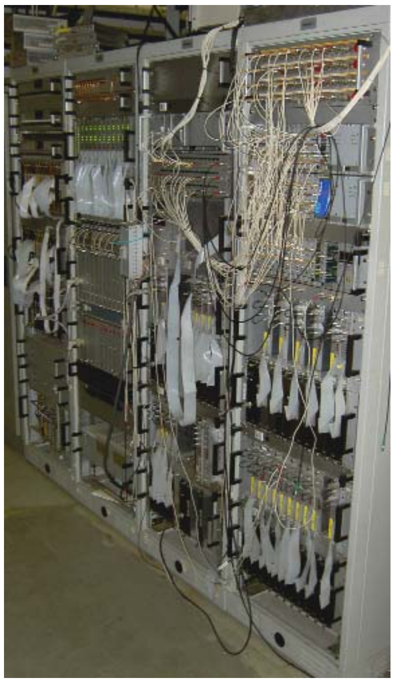
2.1 Problem Description

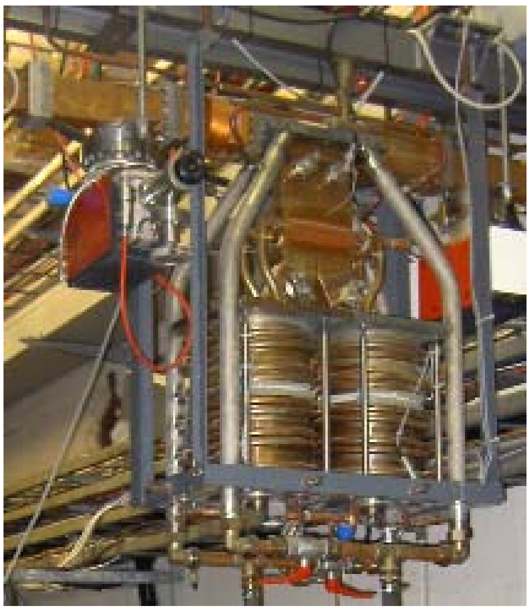
The International Linear Collider (ILC) tunnel will house both array of superconducting accelerating cavities and sophisticated electronic control systems (mainly digital), responsible for controlling the collider. Such an arrangement exposes the electronic systems to mixed gamma radiation and neutron influence, of doses and intensities greater by orders of magnitude than that occurring in regular conditions. Therefore measures need to be taken to harden electronic devices against radiation, neutralize or mitigate its negative effects. Moreover, the points of failure, due to radiation, of the systems must be identified and evaluated. For that purpose a reliable communication channel must exist between devices under test, located in the accelerator tunnel, and a monitoring station, installed outside the tunnel, e.g. in the accelerator hall.

2.2 Proposed Solution

Communication channel in the accelerator environment will be subject to the mixed radiation and electromagnetic interference from high-power klystrons, electric vacuum pumps and nearly omnipresent control electronics (see Figure 2.1).

a)


b)


c)

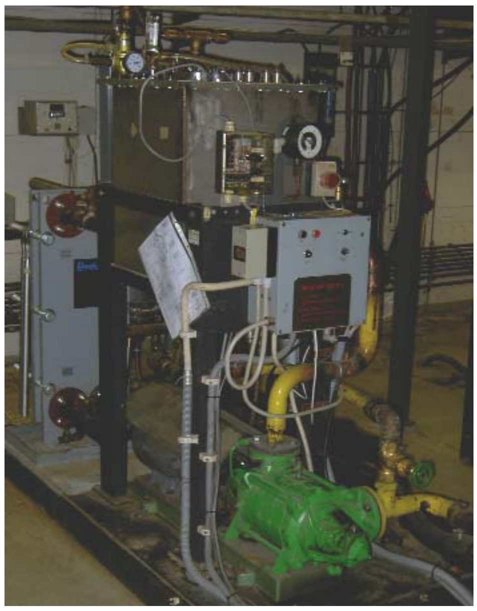

d)

Figure 2.1. Service hall of LINAC II accelerator.
a) Array of high-power klystrons; b) Control electronics;
c) High-power waveguide with SLED (SLAC Energy Doubler);
d) Vacuum pump

Thus, a reliable channel must meet following requirements:

- Transmission channel circuit hardened against radiation or radiation tolerant,
- Transmission medium immune to electromagnetic interference (EMI).

The transmission medium does not need to be radiation protected, as neither gamma rays nor neutrons generate Single Event Effects (SEE – see section 3.1) in known media. The only observed degradation of medium under gamma radiation is described in section 3.2.

Apart from the above requirements, the channel must enable the following:

- Full-duplex serial communication between monitoring station and transmission channel circuit - serial communication reduces the number of necessary signal lines,
- Set of basic control signals (reset, power-off) for the transmission circuit and/or the device under test. The channel circuit may provide those signals for the Device Under Test (DUT), or the lines may be shared, thus reducing the number of control lines,
- Deliver power to the communication circuit and the DUT from power supply located outside the accelerator tunnel.

The most valuable solutions are universal ones, allowing easy accommodation to new conditions. To make this requirement and the ones mentioned above feasible, a transceiver, in a broad sense, is needed on the side of the monitoring station. The transceiver should interface common communication media and standards supporting serial communication, pass control signals from monitoring station and deliver power to the devices in tunnel. The general block diagram of proposed communication channel set-up is presented in Figure 2.2.

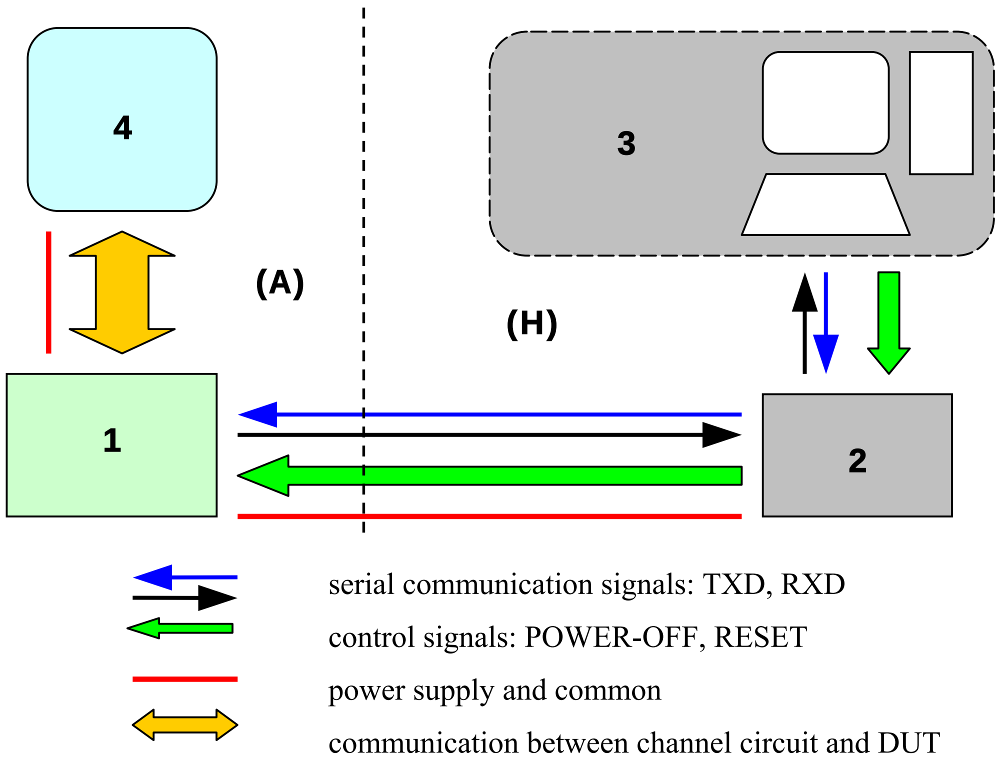

4
3
(A)
(H)
1
2
serial communication signals: TXD, RXD
control signals: POWER-OFF, RESET
power supply and common
communication between channel circuit and DUT

Figure 2.2. Block diagram of proposed communication channel. 1 – radiation hardened or tolerant communication circuit, 2 – communication, control signal and supply transceiver, 3 – monitoring station, 4 –DUT, (A) - accelerator tunnel, (H) - accelerator hall. The power supply unit is not included.

2.3 Possible Implementations

The proposed block diagram in Figure 2.2 is a high-level abstraction of the design. Choice has to be made on the components used for implementing it. To make the devices easy to build, commercial off-the-shelf (COTS) components will be used. This lowers the cost, and enables to avoid problems arising from limited components availability.

The transmission channel circuit should be flexible enough, to interface it to various devices, with only minor or no changes to the circuit's configuration. It is not acceptable, that for every type of device to be interfaced, a different circuit must be built, starting from the schematic and printed circuit board level. This would increase the cost of a single device significantly, and extend design time a number of times. The circuit should be also able to accommodate different signalling standards and protocols. The design, which will meet the requirements, calls for a programmable or re-programmable transmission channel circuit. Programmability or re-programmability is available in many COTS component families, just to mention re-programmable integrated circuits like: GAL, PLD, and FPGA devices, microprocessors, and microcontrollers. In this particular design, however, FPGA devices and microcontrollers are the field of interest. Both provide abundant resources, thus greatly lowering the number of required components and decreasing the complexity of design on the PCB level.

The channel circuit must be a command driven system, with a degree of autonomy. It should support a well-established serial communication protocol. The EIA-232 protocol is sufficient as a carrier, for a more reliable, frame-based protocol. The possibility of employing signalling standard other than EIA-232 should be also available. There must exist a means of guaranteeing reliability of communication, e.g. through Cyclic Redundancy Check (CRC). The types of devices, which can be interfaced to the transmission channel circuit, may not depend on circuit's architecture, should be handled in a general Input/Output manner. The main task for the circuit will be testing of electronic devices in the radiation environment. The core functions for this, such as proper communication with a DUT through control signals and buses should be implemented in the channel circuit, not the monitoring station. Simple, but extremely useful, application for the circuit could be detector of Single Event Upsets (SEUs – see section 3.1.2) in a Static Random Access Memory (SRAM). In this case the DUT would be the SRAM. The memory would be tested for occurrence of SEUs. The task for the channel circuit would be writing and reading the memory, accomplished through proper use of chip's address and data buses, accompanied by assertion of control signals. The block diagram of such application is shown in Figure 2.3. The number of available I/Os must be sufficient for a wide range of SRAM modules, i.e. 8 I/Os for data bus, at least 20 I/Os for address (1 MB address space), at least one I/O for Chip Select (CS), one for Write Enable (WE) and one for Output Enable (OE). This gives a total of at least 31 I/O lines.

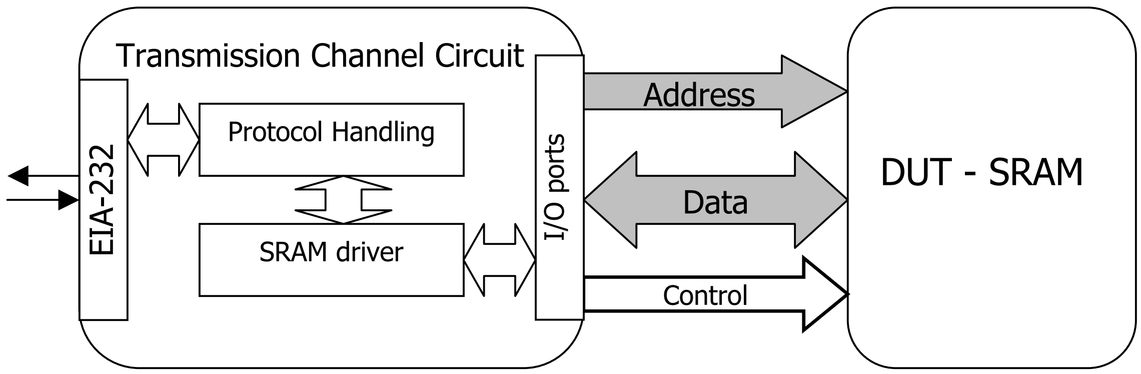


Figure 2.3. Block diagram of an exemplary application of the transmission channel circuit

3 Radiation Effects on Electronic Devices

This chapter briefly describes the adverse effects of neutron and gamma radiation on electronic devices and optical components. The emphasis is put on soft errors, particularly Single Event Upsets (SEUs), since the main goal of this project is to minimize their effects.

3.1 Neutron Radiation

The detrimental neutron radiation is the dominating component of accelerator's radiation environment. Electronic devices operating under neutron irradiation suffer damage. The damage can be done through different mechanisms. They are the displacement damage and Single Event Effects (SEEs).

Since neutrons have no electronic charge, they cannot interact with semiconductor material atoms electrically. However, being 1840 times heavier than an electron, they interact with semiconductor lattice atoms through collisions. The collisions result in dislodging or displacing the lattice atoms from their lattice sites. The atoms are forced to take up interstitial positions within the crystal, resulting in distortion of lattice structure. This effect, as being of mechanical nature, is significant due to high-energy neutrons, such as cosmic ones, thus being a serious problem in space applications. Among results of the damage are resistivity changes and bipolar transistor gain degradation [1]. The displacement damage may have reversible nature. The semiconductor lattice structure may be restored. This self-healing of a device is referred to as *annealing*. It relies on thermal motion of the defects. The displaced atoms migrate from interstitial to vacancy positions. On the other hand, the migrating atoms may form stable associations with impurity atoms in the semiconductor structure. This will enhance the degradation, as such associations are defects in the lattice.

The second type of neutron effect on electronic systems is an indirect one. As mentioned earlier, there are no direct electrical interactions of neutrons with semiconductor atoms. Nevertheless, neutrons can induce ionisation through secondary processes [1]. The most important in the context of this project is the generation of alpha particles within semiconductor lattice. Low-energy neutrons, in particular *thermal neutrons*, dominate the neutron-energy-spectrum of the electron accelerator housed in a concrete containment. Under such conditions the neutron capture reaction in Boron

(^{10}B) atoms, present in silicon chips as dopant and used in passivation layers, takes place. The product of the reaction can be either ^{7}Li ion or ^{4}He ion. The ^{4}He ion (an alpha particle) is probably the agent for triggering SEE in silicon devices [2]. Alpha particles also emanate from heavy elements, contaminants of chip package material, being potential sources of SEEs in commercial electronics. Such problems are avoided, by proper passivation of semiconductor structures. Unfortunately, this protection is ineffective in case of neutron-induced alpha particles, since they are generated in the very bulk of silicon.

The Single Event Effects are further subdivided into *firm* and *soft* errors, briefly characterized in sections 3.1.1 and 3.1.2.

3.1.1 Firm Errors

The firm errors may result in permanent damage to the electronic device, rendering a device or its component unusable, posing potential threat to other system's components. The most popular firm error is Single Event Latch-up.

Single Event Latch-up takes place when parasitic thyristor in CMOS device (Figure 3.1) is switched into low-impedance ("on") state. The transition into "on" state is initiated by a charged particle (e.g. alpha particle) depositing charge in the structure, thus creating a current pulse. The current pulse switches the thyristor into conducting state and causes excessive current flow between power terminals through the device. If the power is not immediately switched off, the excessive heat can permanently damage the silicon device.

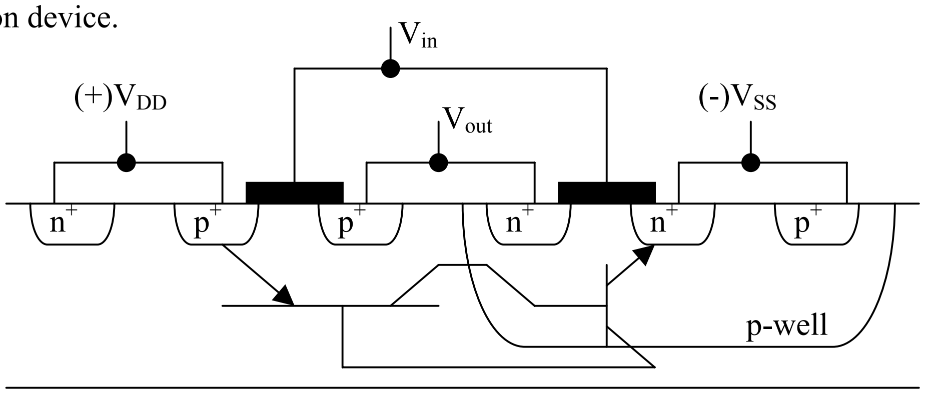


Figure 3.1. Parasitic thyristor in CMOS inverter structure (p-well technology) [3]

The other two known firm errors due to SEE consider power devices. The SEGR (Single Event Gate Rupture) and SEB (Single Event Burnout) are effects, which lead to the permanent damage of power MOSFET transistors. Since susceptibility to firm errors can be improved on technological level only, these phenomena are not described in detail.

3.1.2 Soft Errors

The soft errors are much less severe than the firm ones. There is no direct permanent damage done to silicon devices due to soft SEE. The only impairment caused, is on the functional and performance level of the subject system. Digital systems are exposed to two types of soft errors in the radiation environment. These are Single Event Upset (SEU) and Single Event Transient (SET).

Single Event Upset (SEU) concerns data storage elements. Memory cells such as Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM) and Master-Slave edge-triggered Flip-Flops are subject to this detrimental phenomenon. The effect of SEU is distortion of data stored in the cell, simply called a bit-flip. The bit-flip takes place if either logic '1' was written into the cell, but a logic '0' is read, or '0' was written and '1' is read. The mechanism of SEU is briefly explained for the 6-transistor SRAM cell shown in Figure 3.2 [3]. The cell consists of two cross-coupled CMOS inverters (T1-T3 and T2-T4) with access transistors (T5, T6) added.

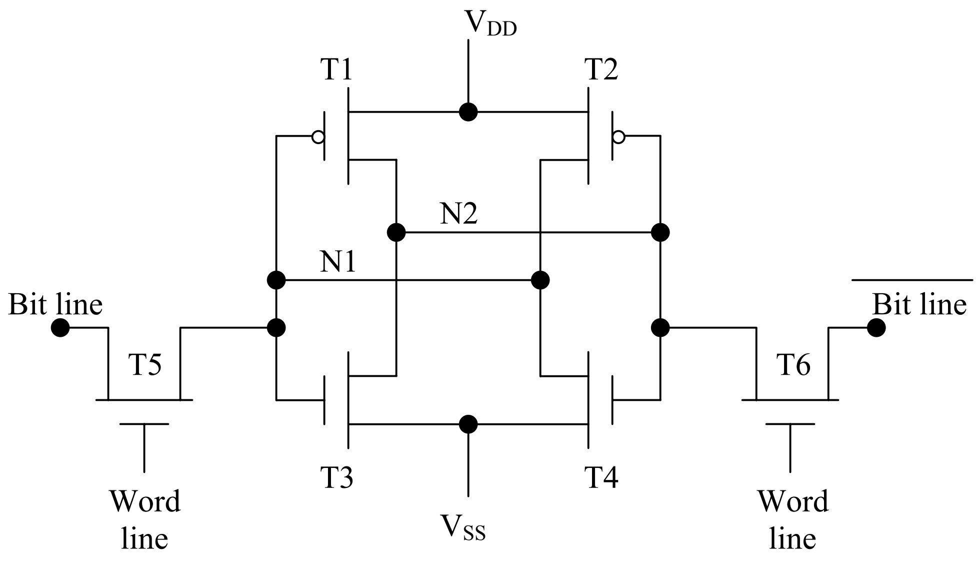


Figure 3.2. The 6-transistor SRAM memory cell. T5 and T6 are n-type access transistors, T1 and T2 are p-type load transistors, T3 and T4 are n-type drive transistors [3].

The bit-flip in the cell of Figure 3.2 may proceed in two different scenarios, depending on the data, which is stored in the cell. If a '1' is stored, the transistors T2 and T3 are conducting ("on") whereas T1 and T4 are non-conducting "off". If an ionising particle, such as previously mentioned alpha particle, crosses the transistor's T4 or T1 channel region, it may ionise the area. This creates electron-hole pairs and may switch the affected transistor into the “on” state, temporarily. If this happens, a low-voltage spike at node N2 is created. The ‘0’ spike is propagated to the input of T1-T3 inverter and effectively latched into the cell. The scenario for the ‘0’-to-‘1’ bit-flip is similar. The difference is that in this case the vulnerable transistors are the T2 and T3. None of the “on” transistors can be switched “off” by ionisation.

Apart from impairing the data stored by a system or system’s state, the SEU can temporarily transform system’s functionality. This is a severe problem of SRAM based Field Programmable Gate Array (FPGA) devices (briefly characterized in section 4.1). Such an event is referred to as Single Event Functional Interrupt (SEFI). The rate at which SEU occur depends strongly on the packing density and fabrication technology of the affected integrated circuit. The higher the packing density is, the greater is the SEU rate. The feature size of circuit’s fabrication technology is also an important factor. As scaling down progresses, the vulnerable area decreases. However, the supply voltage is usually also decreased and the frequency of operation rises. These effects, when cumulated, yield increased SEU sensitivity of sequential systems.

Single Event Transient affects combinatorial circuits only. It manifests itself by transient changes of voltage levels on signal lines, either being input to combinatorial blocks, or the outputs from them. An SET occurs if e.g. a block is driving an output in the logic ‘1’ state, and a transient ‘0’ is observed, which does not correspond to input vector. The SETs are mainly caused by ionising particles, particularly heavy ions. If an SET is latched in a storage element, it may be perceived as SEU.

3.2 Gamma Radiation

Gamma radiation is the high-energy electromagnetic radiation. The wavelength of high-energy gamma rays is of the order of fractions of an angstrom, i.e. of the order 10^{-8} cm. Their interaction with matter is through three types of phenomena: the photoelectric effect, the Compton effect and pair creation. The damage caused by gamma radiation is ionisation damage [1]. Its level is determined by the Total Ionising Dose (TID) – the dose of ionising energy absorbed by the subject material – and the material type. Among the effects of gamma on electronic devices and structures, is the erasure of Flash memories. Flash memory is based on floating-gate MOSFET transistors, shown in Figure 3.3 [3]. The transistors act as non-volatile storage cells.

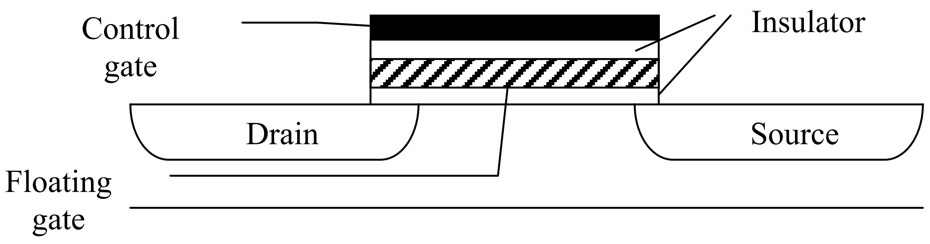


Figure 3.3. Floating-gate MOSFET transistor [3]

The memory cell is in programmed state (stores logic ‘1’), when charge is accumulated in the floating gate. When the programmed structure is exposed to gamma irradiation of high TID, the accumulated charge is gradually liberated from the floating gate, finally resulting in erasure of stored data, as confirmed experimentally [4]. Since gamma radiation is electromagnetic wave, it can be relatively easily shielded, by means of iron or lead slabs of proper thickness [1]. Other materials composed of elements of high “Z” (atomic) number may also be used for the purpose.

Gamma radiation causes also detrimental effects in optical devices. Current experimental results show limited tolerance of COTS (Commercial-Off-The-Shelf) optical components to gamma irradiation [5], [6]. The effects comprise reduction of LED transmitter efficiency and decreased SNR (Signal-To-Noise Ratio) at the P-I-N receiver [6]. Vulnerable element of optical component assemblies is the optical lens, which purpose is to increase the coupling efficiency of light into the fibre. The lens tends to lose clarity at high radiation doses. While the latter effect is a permanent one, the decrease of SNR is reversible if radiation is removed. Apart from the transducers, the optical fibre is also affected by the gamma radiation [7]. The optical fibre suffers increase of attenuation per length unit, proportional to the TID. This effect is less significant if the fibre is “live”, i.e. if it is intensively used. A fibre, which was subject to gamma radiation, will gradually self-heal provided it is used intensively while radiation is removed. This effect is referred to as *photobleaching*.

4 Families of FPGA Devices, Microcontrollers and Their Radiation Tolerance

This chapter gives a brief insight into families of FPGA devices available on the market. Microcontrollers are also briefly described. Every device family is characterized in respect of its re-programmability and radiation vulnerability. Finally radiation influence on Finite State Machines and microcontrollers is explained, and the choice of the device for the main component of the transmission channel circuit is justified.

Considering FPGA devices as candidates for the main component has its additional explanation in the fact, that these circuits will be broadly used in the control systems of the ILC. Engineers have recently completed and tested a prototype of cavity controller, implemented in an FPGA. Hence, it is vital to investigate the influence of radiation on the FPGA devices. Building the transmission channel circuit based on such a device, would allow conducting this investigation simultaneously with the main task of the circuit.

4.1 SRAM Based FPGA

The SRAM (Static Random Access Memory) based FPGA device is the most widely used one. Numerous medical, automotive, as well as entertainment and multimedia applications are built based on the devices. The greatest market for them is, however, telecommunications [8]. SRAM-based FPGA device is re-programmable. Most of FPGA devices are constructed following a common architecture, presented in Figure 4.1.
Configuration of SRAM based FPGA is stored in SRAM, during the device's operation period. Since SRAM is volatile, the configuration is permanently stored in a non-volatile memory (ROM, EEPROM or Flash) and uploaded to the FPGA upon its start-up. Some most advanced devices are capable of being partially re-configured during full operation. The necessity of the external memory increases complexity of the PCB design. On the other hand, the use of SRAM enables the FPGA to operate at high frequencies. As described in section 3.1.2, an SRAM memory cell is highly susceptible to SEU. This poses high risk on proper device operation under neutron irradiation, as the functional configuration of the device will unpredictably change in result to such an event.

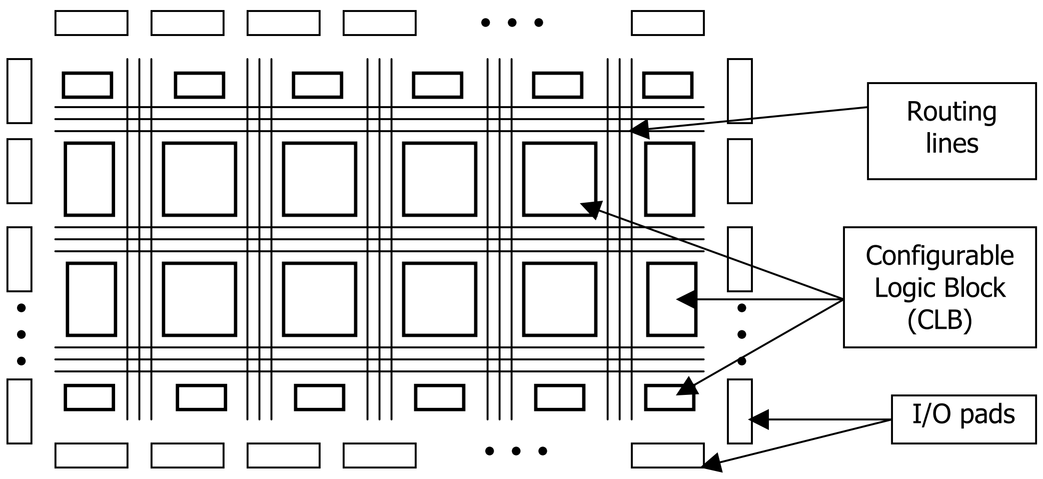


Figure 4.1. Architecture diagram of a typical FPGA device

This problem has been confirmed by an independent study [9]. The consequence of that will be circuit malfunction, a SEFI, or even device damage. To remedy the malfunction, the FPGA must be restarted, thus loading a valid configuration from non-volatile memory. This greatly degrades performance, due to delays caused by configuration uploads to FPGA or even renders the device unusable, if the Mean Time Between Functional Interrupt (MTBF) is too short for a required task to complete. The problem of detecting such changes poses additional difficulty [10], [11]. Storing the configuration in a non-volatile memory does not make it fully immune to radiation. Experiments [4] have shown, that e.g. Flash memory is erased under high TID. The device may also be damaged by a SEL. The advantage of SRAM based FPGA devices is low cost per device and unlimited number of programming cycles.

4.2 Flash-based FPGA

The Flash-based FPGA device is a relatively new technology, the first devices were introduced to the market in 2002. The only company offering such devices is Actel [12]. The device is, similarly to SRAM based one, re-programmable. However, it is different from the SRAM FPGA in two respects. First, it does not require external non-volatile memory to store the configuration. This enables simpler PCB design, reduces required component count. Second, during operation of the device, the configuration is not transferred to SRAM. The Flash-based FPGA stores its configuration in (as the name implies) Flash memory. Flash is non-volatile, thus configuration is retained after power-off. What is the greatest advantage of the device is that the configuration is used directly from Flash during device's operation. This is achieved by implementing configurable interconnections by means of flash-switches (details can be found in Actel's Flash devices datasheets [13]). The benefit of such architecture is the "live at power-up" capability of the devices – there is no time delay for configuration upload, since no upload takes place. Flash memory is also immune to soft neutron-induced effects [14], rendering the FPGA's configuration resistant to unpredictable change due to SEU. However, as mentioned in section 3.2, contents of Flash may be erased by gamma radiation, if exposed to it for extended time. The drawback of using Flash-based FPGA is lower maximum frequency, at which complicated designs can run, as compared to that of SRAM FPGA. Also the available resources are less numerous, when the most powerful devices of both families are compared. The price per available number of equivalent system gates and a limit on programming cycles the device can sustain also favours SRAM based devices.

Other FPGA devices, referred to as Flash-based, are also available on the market [15]. The Flash is embedded into the FPGA, eliminating the need of external non-volatile memory for storing configuration. However, as in SRAM based devices, the configuration is copied to that memory for the period of device's operation. This imposes the same limitations on device reliability as in case of SRAM FPGA.

4.3 Antifuse FPGA

Antifuse devices use metal-to-metal connections created during programming, to define device's configuration. Hence, they are programmable only once (are not re-programmable). Because they use permanent metal-to-metal connections, they are the least volatile and least susceptible to radiation of any programmable logic technology in terms of configuration. This applies to both neutron and gamma radiation. They also are capable of operating at the highest frequencies, because the all-metal routing path is faster than one where interconnections pass through transistors. Their main drawback in the context of this project is the lack of re-programmability. When developing prototype of a device, multiple iterations of the design are unavoidable. The device, before reaching its final stage, must be verified in the field. Choosing a one-time programmable (OTP) device would require larger expenditures. Also the flexibility of the device would be compromised, as adding new components (e.g. signalling encoders or decoders) would require using a new one.

4.4 Radiation Hardened, Radiation Tolerant FPGA

For markets demanding level of design's reliability greater than regular (space, aerospace, defence and military), dedicated FPGA device families were developed. The effort put into the development of Radiation Hardened devices is focused on preventing SEEs from occurring. Some of these hardening methods will be briefly described in section 5.1. Radiation Tolerant devices offer lower reliability than Radiation Hardened ones. The SEEs are not prevented from occurring, but their negative effects are mitigated. This improves the reliability to a level acceptable in respective applications. Both types of devices have been in the market for less than few years. Since the technology is the newest in the FPGA sector, the most sophisticated and used in very specialized applications only, the prices of the Radiation Hardened and Radiation Tolerant devices are beyond financial range of this project. An exemplary price of a Radiation Hardened FPGA can range from €5 000 to €20 000 per one item. What is even more discouraging, available Radiation Hardened devices are one-time programmable, being based on antifuse technology. The Radiation Tolerant devices are available from different vendors in two types: SRAM based [16], and antifuse based [17].

4.5 Microcontrollers

Microcontrollers can be briefly described as microprocessors fabricated with embedded program memory for storing executable code or hard-coded data (ROM, EPROM, EEPROM or Flash), Random Access Memory for volatile program data (SRAM), communication interfaces (e.g. UART, SPI, I^2C) and general-purpose I/O ports (GPIO), just to mention the most common. These devices are present in virtually every electrical appliance. Their fixed hardware configuration can be, on one hand, an advantage, providing a design error-free platform for software development. On the other hand, it is a severe limitation. Supporting communication standards other than those available in embedded hardware requires either adding sophisticated external components (specialized integrated circuits) or developing software routines for that purpose. The latter solution, however, compromises overall system performance, particularly speed, and is not always feasible. The former – increases complexity of design, and introduces additional potential points of failure. The main drawback of fixed hardware architecture, in addition to those described, is inability to implement SEE protection mechanisms other than system-level redundancy, which is firstly complicated to manage, and secondly best for manned equipment, not for autonomous [1]. The protection against SEU is particularly vital, since CPU registers in the microcontroller are implemented in SRAM. The same is true for stack, FIFO (First Input-First Output) instances of communication devices, and embedded RAM. The advantages of microcontrollers are: low price and well established development support at reasonable prices. Their generality, attained through software, is a strong pro, as well.

4.6 Radiation Induced Errors in Finite State Machines and Microcontrollers

The transmission channel circuit, as described in section 2.3, must be implemented as a Finite State Machine, other speaking, as a sequential system. Therefore, before commencing the design phase, possible failures arising from influence of radiation on FSMs must be identified.

Sequential systems, or FSMs, base their operation on states. States are information on the history of system's activity and its current status. In a particular instant of time, an FSM can find itself in only one particular state. Transitions between states can be either conditional or unconditional. In hardware, states are represented as data, stored in registers. In FPGA devices, registers are built using either Master-Slave D-type Flip-Flops or SRAM cells. Since both are memory elements, then, as described in section 3.1.2, are susceptible to SEU, in principle [18]. For sequential system, such an event may have disastrous consequences. Randomly negating one bit in the state register drives the FSM into unintended state. Hence, the required flow of state transitions is discontinued. The FSM may be driven either into a valid or invalid state. Being driven into the former, the system would still operate, however incorrectly. If driven into the latter, the system could enter an infinite loop or destroy volatile, but valuable data. In either case, system will encounter a malfunction.

Microcontrollers face similar problems. As mentioned in section 4.5, their working and status registers are implemented in SRAM cells. The same may apply to embedded RAM and other memory components. A SEU in any of them will falsify the data causing software to produce erroneous results, errors in communication or even collapse of the whole system. Apart from that, the hardware may behave unpredictably. Currently executed instruction, latched in Instruction Register can be changed by SEU, making the MCU operate in a random manner. Moreover, the FSM, which manages the MCU's datapath, will fail due to SEU, as well.

To summarize, both FSMs and microcontrollers are built of components vulnerable to neutron radiation. Their behaviour will be unpredictable due to SEUs.

4.7 Selection of the Optimal Device

Several factors need to be taken into account, when deciding which of the, briefly presented, device families to choose for the main component of the transmission channel circuit. Since a prototype will be developed, re-programmability is very important. The specific environment, in which the final device must operate, requires the devices functional configuration to remain intact under irradiation, at least between possible servicing interventions. To meet the requirements, either a microcontroller or a Flash-based FPGA should be used. However, microcontrollers, having fixed hardware platform, cannot be protected by any SEE hardware mitigation techniques, moreover, may force the designer to use third-party components, also sensitive to radiation.

The Flash-based FPGA was chosen to house the main circuit of the transmission channel. It offers re-programmability and functional configuration unchanged under radiation (except TID effects and SEL), during devices operation, for extended periods of time. It gives flexibility on hardware level of the design, without the need to redesign from schematic and PCB. Hardware mitigation methods can be implemented and tested for effectiveness. However, the great flexibility offered by software, being the attribute of microcontrollers, is also highly demandable, as changes to e.g. communication protocol can be introduced without FPGA re-configuration. The ideal solution would be to merge the reliability and hardware flexibility of Flash FPGA with software flexibility of a microcontroller. Therefore, the decision to implement a microcontroller in the Flash-based FPGA was made. The Actel ProAsic Plus APA600 FPGA was chosen, as offering 600k equivalent system gates, yet being affordable. This, and other devices from this line, has another advantage, important in novel designs, and generally in the embedded systems market. Their FlashLock feature makes it virtually impossible to reengineer the design, disabling the possibility to copy it by competitive companies or individuals [19].

5 Radiation Hardening and Mitigation Techniques

This chapter is a brief summary of well known and used mitigation and hardening techniques. Section 5.1 highlights hardware hardening and mitigation against different types of SEEs. The remaining sections focus on mitigating SEUs. Most of the techniques were developed especially for high-reliability applications, such as space exploration, medical, military or satellite communications. Others come directly from telecommunications, where are used for improving transmission reliability, but can be adopted for mitigation, as well. The techniques can be characterised in two categories. Technological, realisable in hardware only, usually through modified fabrication processes of integrated circuits. Software, which can yield a system or module, which is radiation tolerant, but not hardened. This means, that the circuits operate properly up to specified levels of radiation, possibly with some loss of performance. The software techniques can be realised in either hardware or software, depending on the platform.

5.1 Technological Hardening and Mitigation

The most effective, robust techniques allowing to harden electronic devices against radiation are those applied at the lowest, technology, level. They involve modified cells, being the building blocks of integrated circuits, altered structure of transistors and properly selected compounds for passivation of integrated circuits. The manufacturing technology can also decrease circuit’s radiation sensitivity. The Complementary Metal Oxide Semiconductor (CMOS) technology is inherently more radiation resistant than e.g. N-type Metal Oxide Semiconductor (NMOS) one. Even greater radiation resistance is achieved in Silicon On Insulator (SOI) technology. It enables complete elimination of latch-up, due to lack of parasitic thyristor (see section 3.1.1). Additional advantages of SOI are: greater attainable frequency of operation, decreased power consumption. The drawback is the cost, as compared to standard CMOS.

Protection on higher levels of hardware has also been developed. Particular attention was paid to hardening SRAM cells, being the most frequent point of failure. One of approaches toward reducing SRAM vulnerability to ionising particles is to add series resistors on the cell’s feedback paths (see Figure 3.2 for schematic of SRAM cell). The modified cell is shown in Figure 5.1.

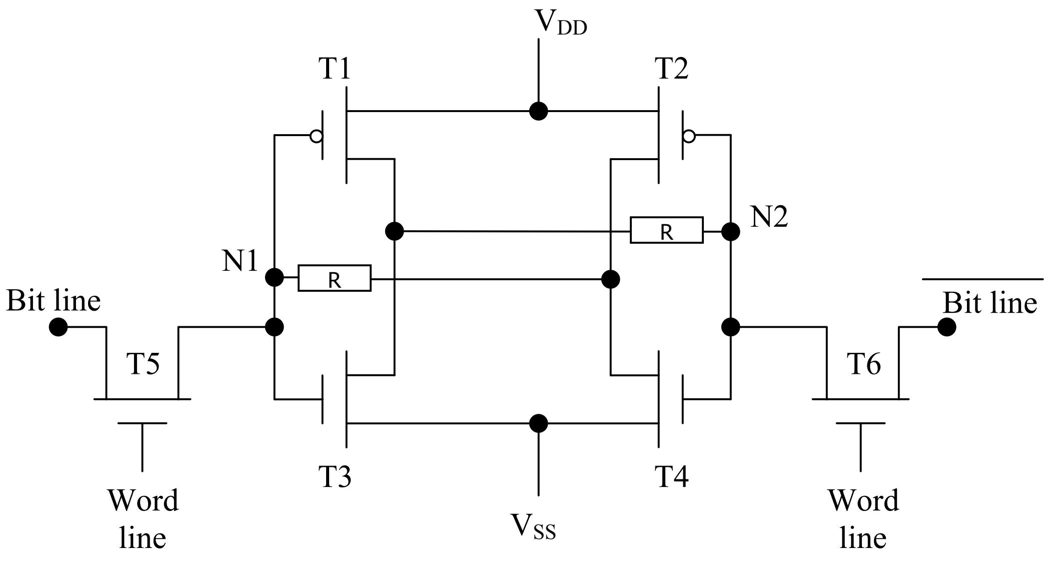


Figure 5.1. SRAM cell with resistors in feedback paths

The benefit coming from this solution is an effective RC low-pass filter. The filter is composed of an added resistor and input capacitance of an inverter. Short transient spikes on N1 or N2 nodes can now be filtered-out by the RC network. This reduces the probability of such a spike being latched in the cell.

At this point it is important to mention, that the Actel ProAsic Plus devices are intrinsically hardened in a similar way. Figure 5.2 shows an elementary configurable logic cell of a ProAsic Plus FPGA [20].

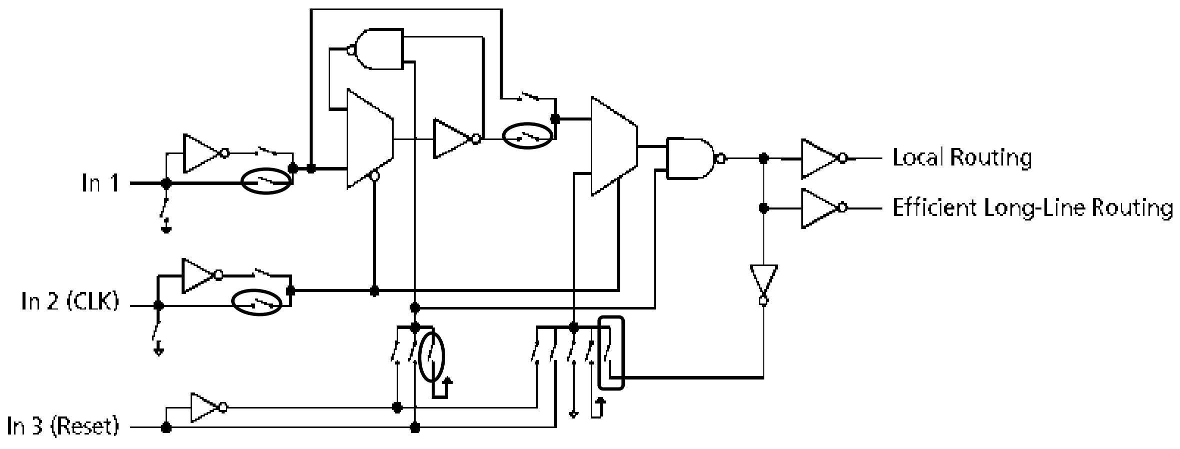


Figure 5.2. Elementary configurable logic cell of Actel ProAsic Plus FPGA [20]

When the tile is configured as a storage element, a master-slave D-type flip-flop is created. Equivalent circuit is shown in Figure 5.3.

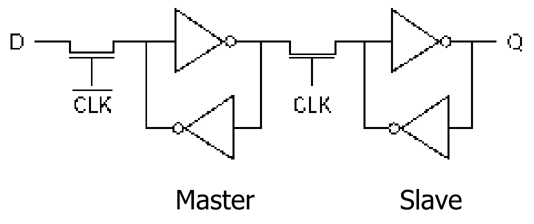


Figure 5.3. A Master-Slave D-type Flip-Flop

Such configuration is accomplished by closing the flash switches encircled in Figure 5.2. A flash switch is depicted in Figure 5.4

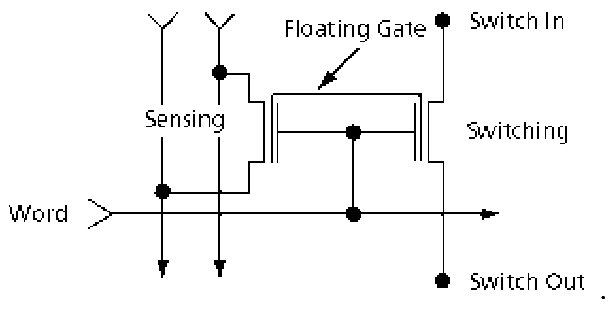


Figure 5.4. Flash switch of Actel ProAsic Plus FPGA [20]

One switch in Figure 5.2 is enclosed in a rounded rectangle. This is the only switch, which finds itself in a feedback path. This feedback constitutes to the slave latch of the flip-flop. If a voltage spike is generated by ionised particle in any of the components along the feedback path, it can potentially generate an SEU. The slave latch will be more resistant to such effects, as an RC filter is present in its feedback path. The RC network is composed of resistance of the flash switch and the input capacitance of multiplexer. The flip-flop is radiation hardened during part of the clock cycle. It is the part, during which the transistor gating the Master flip-flop is conducting. At the same time the transistor which gates the Slave flip-flop is non-conducting, i.e. the feedback path of Slave flip-flop is closed, sustaining the stored binary value.

Other hardware mitigation techniques usually comprise Triple Modular Redundancy (TMR, describer in section 5.2). The TMR is applied to flip-flops or latches, with voting circuits being tripled, as well. An example of flip-flop tripled on the latch and voter level is shown in Figure 5.5.

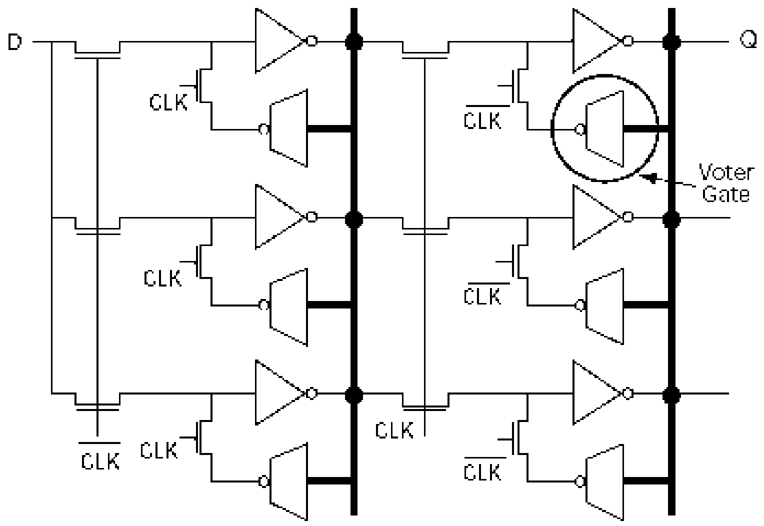


Figure 5.5. D-type Flip-Flop hardened with TMR [21]

5.2 Double Modular Redundancy, Triple Modular Redundancy

The technique of modular redundancy is most widely used in high-reliability applications. There are various kinds of this scheme, depending on the number of single module replications. Thus, there is Double (DMR) and Triple (TMR) Modular Redundancy – the most common, there may also be Quintuple Modular Redundancy (QMR) or other of higher order (xMR). This scheme is used from car or aircraft control and safety systems, power plant systems to military and space ones. Either the whole system or subsystem can be replicated or only its most vulnerable modules. There may be two types of modular redundancy. First when only one system or module is operating at given time. The module or system is diagnosed. Once it undergoes malfunction it is disconnected and the back-up system or module takes over its function. The additional entity is referred to as *cold reserve*. The second is based on voting. In DMR two entities function in parallel, performing the critical tasks in the same instants of time. The scheme is also referred to as *hot reserve*. The outcomes of the task, coming from the two entities are fed to a voter, or simply a comparator (see Figure 5.6). Whenever the results are different an error is signalled. The redundant system has entered an erroneous state.

It cannot recover from it, as there is sufficient information only to detect the error, no correction is possible. The two modules or systems produced different results, but it is unknown which one is correct. Therefore, the systems must be restarted, in order to initialise them with known and correct values.

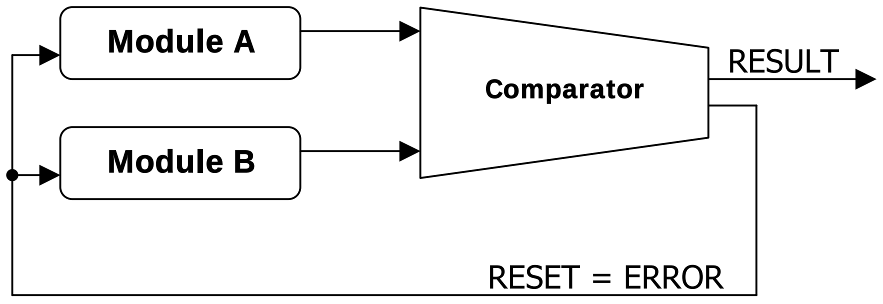


Figure 5.6. The idea of DMR

TMR is more capable. In this scheme every critical entity is tripled. A voter, more sophisticated than the comparator in DMR scheme, is fed with outputs from every entity. The voter decides on the result, by performing majority voting (see Figure 5.7). The result is as indicated by at least two entities. If the representation of outputs from the entities is binary, the voter is always able to decide, it is never confused. Hence, the TMR scheme is not only capable of detecting an error, but also correcting it. For the reason, it is able to sustain system's operation if an error is encountered. This capability can pose a potential threat. If two or all three modules are corrupted by error, the TMR will not notice it, and proceed as if the error was not present or was corrected. The limitation of TMR is that it can correct single errors only.

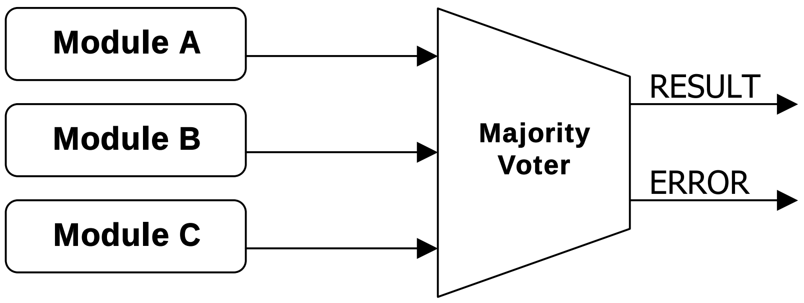


Figure 5.7. The idea of TMR

Many times a designer faces a dilemma, which xMR technique to choose. Both DMR and TMR are capable of detecting system's malfunction. The DMR requires a restart in such a situation, thus destabilising the system and compromising performance. The TMR scheme allows for continuous operation, at higher cost, however. TMR requires more resources than DMR and a more sophisticated voter. The tripled entities must be well synchronised not to generate some transient states at the voter output. For applications, which do not require continuous operation the DMR scheme is an optimal choice. The TMR should be applied in critical high-reliability designs.

In every xMR scheme the voting circuit becomes the vulnerable element, as not being replicated. Measures must be taken to minimise its sensitivity to radiation.

5.3 Hamming Codes

Hamming codes belong to the family of Forward Error Correcting (FEC) codes. They stem from digital communications, where are used for increasing communication reliability. They are also used in ECC computer memories. Hamming Codes rely on adding redundant bits to the transmitted word of information. The redundancy is used at the receiver to detect and correct errors. Hamming Codes are capable of detecting and correcting all single-bit errors within a word. Detection of double errors is also possible, but the amount of redundant information is insufficient to correct such errors. The theory behind Hamming Codes is based on matrix multiplication in finite field (Galois Field) arithmetic [22]. The Hamming rule ensures that all single-bit errors are correctable - the distance between two codewords is equal to 3, i.e. any two codewords of a particular Hamming Code differ in at least three positions, in binary. Hamming Code can be constructed for any $m \geq 2$, where m is the number of check bits.

The Hamming rule is given by:

$$n = 2^m - 1$$
$$k = 2^m - m - 1$$

The k is the length of the data vector, n designates the length of codeword and m is the number of check bits. Code with such parameters is a *perfect code*. For the code to have the double error detecting capability, additional check bit is required, thus the double error detecting, single error correcting Hamming Code has m+1 check bits. Its length n is 2^m. Generally, such code is designated *(n, k) Hamming Code.* From the engineer's point of view the conclusions following from the matrix operations are most valuable, as they provide him with a straightforward method to implement Hamming

Codes without the need of matrix operations. This is explained on an example of (8, 4) double error detecting, single error correcting Hamming Code.

The parity check bits are located at positions *p*, which fulfil the condition:

$$2^i = p,$$

for non-negative integer *i*, such that $p < n$. Therefore, for the exemplary code the check bits are placed at positions: 1, 2 and 4. The additional check bit for double error detection is placed at position 0. The check bits are even-parity bits for groups of data bits. The data bits, starting from the least significant, are assigned the remaining positions within the codeword, starting from the lowest position available. The resultant codeword is shown in Figure 5.8.

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
D3	D2	D1	*C3*	D0	*C2*	*C1*	*C0*

Figure 5.8. Structure of (8,4) Hamming Code codeword. 'D' – data bit, 'C' – check bit

The binary representation of check bit's position (*cb*) determines the bits for which it is computed. The binary value is used as a mask. Every position (*bp*), except position 0, is masked with the value. If the result of masking is the check bit position (*cb*), then the bit at position (*bp*) contributes to the computation of check bit. For the (8, 4) code one obtains groups of bits shown in Table 5.1. The additional check bit at position 0 is a parity check over all codeword's bits.

Table 5.1. Contribution of codeword's bits to parity checks

Check bit position	**Bits in the group**
1	3, 5, 6
2	3, 5, 7
4	5, 6, 7
0	1, 2, 3, 4, 5, 6, 7

When the codeword is received, the data bits along with check bits are used to compute a *syndrome*. The additional check bit is not employed in the computation of syndrome. It is used along with all the other bits to make additional parity check. The syndrome computation is parity check for all the bit groups, except 0^{th} bit group. The syndrome is composed of computed parity values.

The position of a value in the syndrome is given by:

$$S_{position} = \log_2 (group's\ check\ bit\ position\ in\ the\ codeword)$$

If the value of syndrome is zero and parity check for the additional bit passes, the codeword was received correctly. If the syndrome is zero, but additional parity check fails, the additional parity bit is corrupted by error. It is a single error, correctable by the code. In another case, when syndrome is non-zero and additional parity check fails, the additional bit is correct, but one of the others is corrupted. Again, a correctable single error has been encountered. In the last case, when syndrome is non-zero and additional check passes, the received codeword has a double error. The value of syndrome in case of a single error indicates the erroneous position. In this way not only the data bits can be recovered, but also the check bits.

The described interpretation was used throughout this project to implement purely combinatorial Hamming Code encoders and decoders. Examples are shown in Figure 5.9 and Figure 5.10.

Hamming Codes are suitable for protecting both, memory arrays and distributed registers.

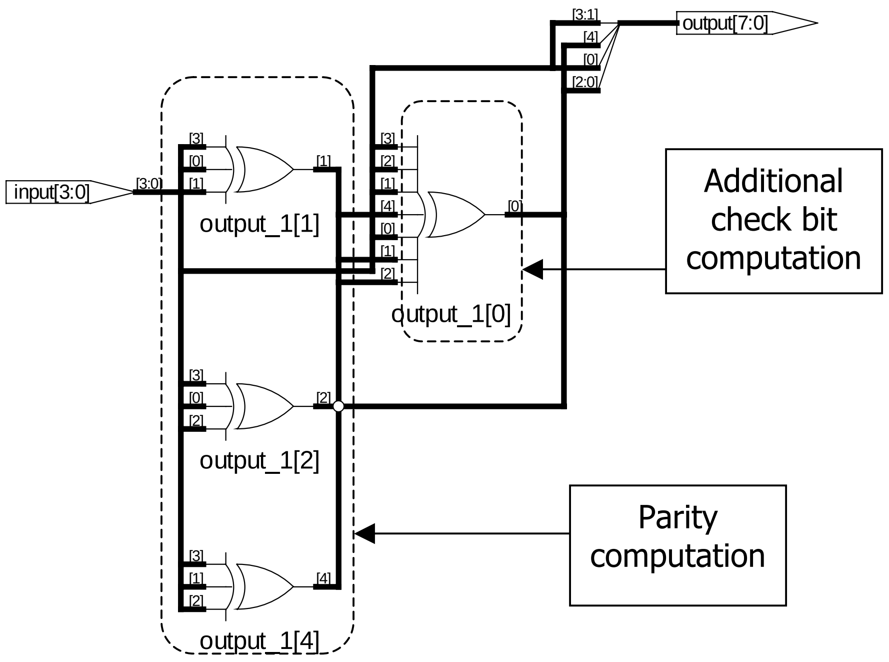


Figure 5.9. Purely combinatorial encoder for (8,4) Hamming Code

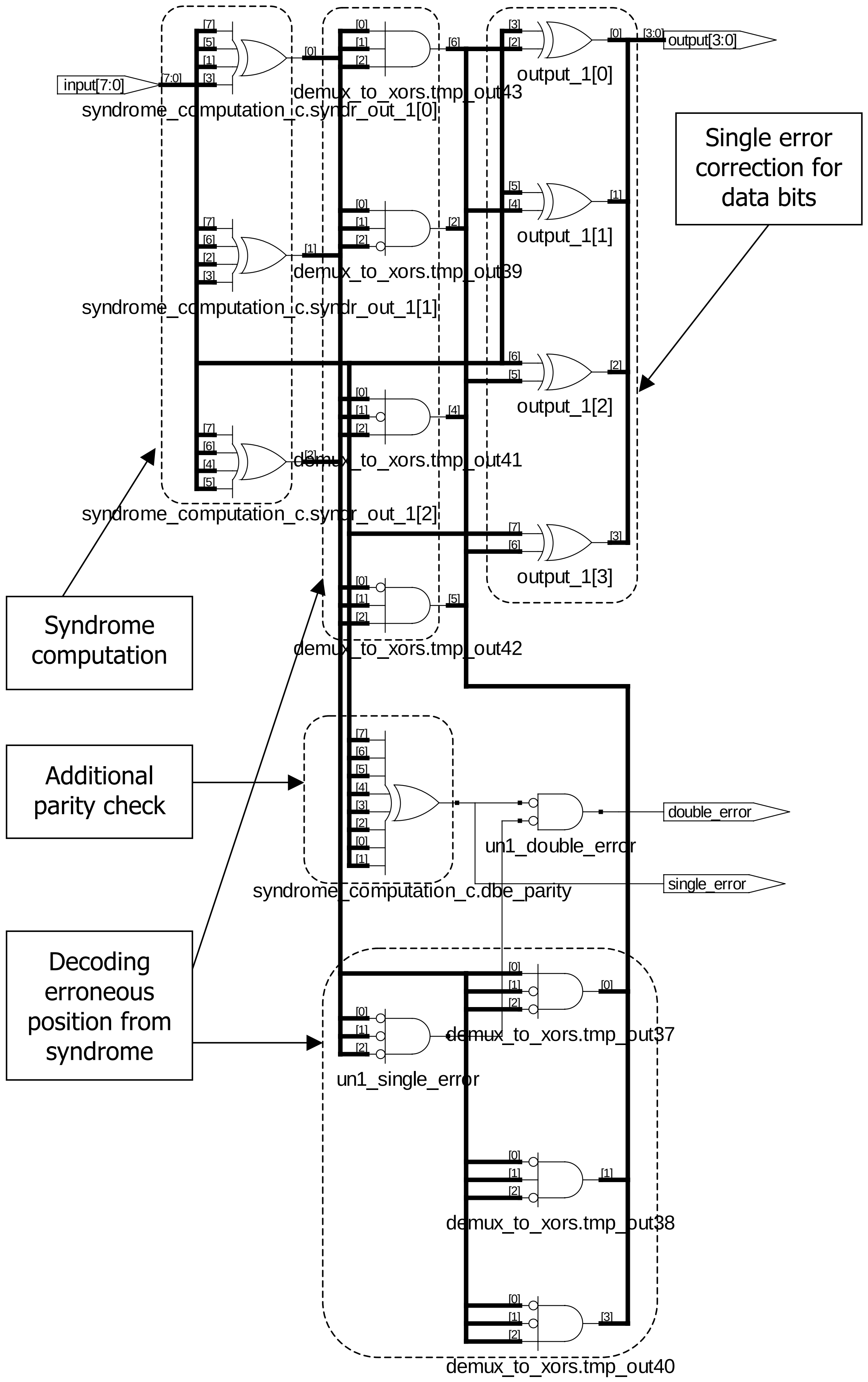


Figure 5.10. Purely combinatorial decoder for (8,4) Hamming Code

5.4 2-D parity Checking

The 2-D (two-dimensional) parity is a technique used for protecting memories or their blocks. In contrast to Hamming Codes, this method is not suitable for mitigating single words of data. The idea of 2-D parity is explained in Figure 5.11. The memory is first subdivided into blocks of known size. The size should be adjusted according to expected rate of SEUs and memory read attempts. Every block contains an even number of words. Such a cluster may be treated as a binary matrix, with words being its rows, while vectors of bits of the same significance, the columns of the matrix. For every row and column a parity bit is assigned. The even-parity scheme is used, i.e. the number of '1s' of a row or column along with the parity bit is always even. The value of the parity bit is chosen to satisfy the condition, parity generation. The even-parity can detect any odd number of errors, but is unable to detect a double error. The simple parity is also unable to correct any error, since there is no information available on its location. This is not a limitation of the 2-dimensional parity. It is capable of correcting all single-bit errors within the protected block. The method is also capable of detecting a double-bit error within either a row or a column, but its correction is not possible. The drawback of 2-D parity, as compared to Hamming Codes, is the inability to read the word of data and correct it on the fly, since the parity bits must also be read. Since at least three read cycles are necessary, read data has to be temporarily stored, resulting in jeopardizing it due to SEU.

Bit / Address	3	2	1	0	Row parity
0x00	1	0	0	1	*0*
0x01	1	1	1	0	*1*
0x10	0	0	1	0	*1*
0x11	0	1	0	1	*0*
Column parity	*0*	*0*	*0*	*0*	

Bit / Address	3	2	1	0	Row parity
0x00	1	0	0	1	*0*
0x01	1	1	1	0	*1*
0x10	0	0	**<u>0</u>**	0	***1***
0x11	0	1	0	1	*0*
Column parity	*0*	*0*	***0***	*0*	

Incorrect parity values due to error

Single error within 4x4 block

Figure 5.11. 2-D parity and single-error correction example

5.5 Scrubbing

Scrubbing cannot be used as a standalone technique of mitigation. It has no inherent error detection or correction capabilities. This technique requires redundancy of data capable of correcting errors e.g. TMR, Hamming Codes or 2-D parity. Other techniques, which enable error detection or correction, are well suited, too. The general idea behind scrubbing is refreshing. It is executed periodically. The memory, its block or other data storage entity is sequentially read, word by word. The employed mitigation technique is used to assess whether data within read word is correct or not. In the latter case, the error correction follows, according to the mitigation technique. The corrected data is written back and the process continues, until whole storage space has been swept. Scrubbing is an important technique, as it prevents errors from accumulating. By carefully choosing and adjusting mitigation technique and the rate at which scrubbing is executed, one can minimise the risk of encountering a non-correctable error. In this way system stability is sustained for extended periods of time. There are drawbacks of scrubbing, too. The resource, which is being refreshed, cannot be accessed by the system. This requires the system to be halted if it claims the refreshed resource. Another solution could be executing scrubbing partially, for resources not being currently used. This, however, is not possible e.g. for memory storing program code. Therefore, the system will suffer some loss of performance if scrubbing is employed. It is important to state, that scrubbing itself is vulnerable to radiation, since an FSM handles it and should be mitigated.

6 Project Description

This chapter gives a description of the implemented transmission channel circuit. The overall structure of the circuit is presented, followed by a more detailed description. The techniques, which were employed to improve circuit's radiation tolerance are further described and assisted with relevant examples of code. Next, the designed hardware platform is presented. Finally, the software for an exemplary application, namely SRAM SEU detector, is described.

6.1 Overview

The transmission channel circuit was implemented in FPGA. As it was mentioned in section 4.7, to achieve high flexibility, the circuit relies on a microcontroller core. The microcontroller chosen is Microchip's [23] PIC16C57 [24]. The Microcontroller Unit (MCU) was significantly augmented, the details are described in section 6.2. In order to demonstrate the functionality of the circuit, a complete communication channel was built, with the MCU at one end, and a PC at the other. The PC functions as a monitoring and command station. The medium chosen for transmission of signals is a full-duplex optical fibre, however other media are also supported. In order to interface the optical signals to the PC and meet requirements described in section 2.2, the transceiver was built, as well. The diagram shown on Figure 6.1 depicts the structure of the complete communication channel. The PC communicates with the MCU via serial port, in conformance to EIA-232 protocol. The serial communication signals can be further carried, apart from optical link, over differential EIA-485 or again EIA-232. On the MCU side an UART is responsible for serial transmission. The EIA-232 and EIA-485 signals are carried over 20-wire ribbon cable, together with RESET and POWER-OFF signals and power supply. To support MCU's re-programmability, external Flash and SRAM modules are installed. The MCU communicates with external devices, such as DUT (Device Under Test) by means of GPIO (General-Purpose Input/Output) ports.

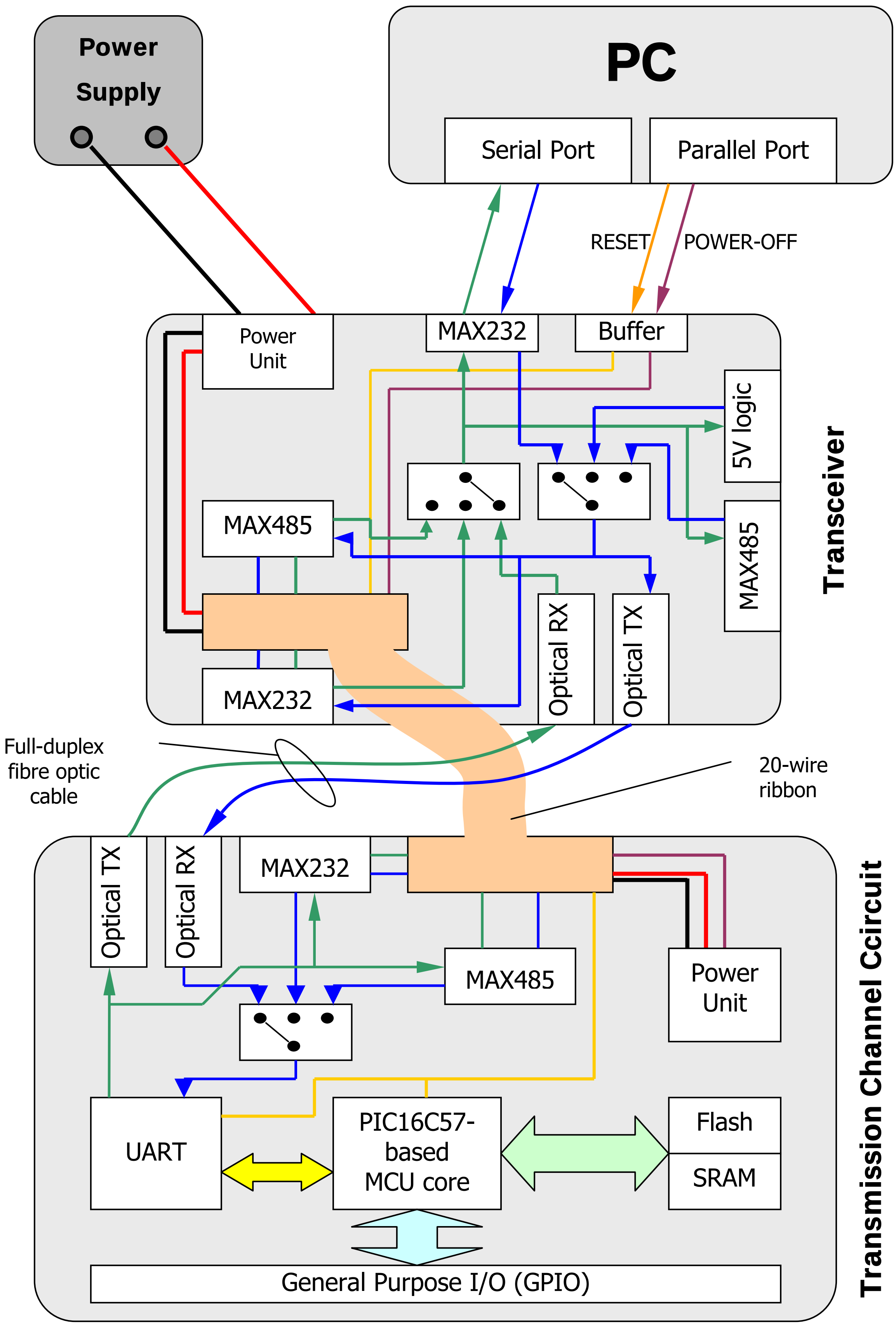


Figure 6.1. Detailed block diagram of the communication channel set-up

6.2 Radiation Tolerant MCU Based on PIC16C57

The transmission channel circuit is based on the PIC16C57 microcontroller [24]. The MCU is designed in the Harvard architecture, what speeds-up program execution and enables pipelining. The MCU core is of RISC (Reduced Instruction Set Computer) type. There are only 33 instructions [24], most of them need one machine cycle to execute. The CALL and GOTO instructions take two cycles to execute. The MCU has 2048-word program memory and 128-byte register file, arranged in 4 banks (0 - 3). Due to mapping of some registers into bank 0 and the fact that some Special Function Registers and peripheral devices are mapped into the register file address space, the effective size of register file is 72 bytes. This MCU has been implemented in the Actel ProAsic Plus APA600 FPGA. The functional block diagram is presented in Figure 6.2. The great advantage of having a flexible hardware platform, compliant with a widely supported standard, is the ability to develop software for it, and what is more, be capable of augmenting or changing peripheral hardware without the need to redesign the PCB. The tools for software development, such as C or Basic compilers for the MCU, are already present and constantly improved. The basis for developing the MCU was description of PIC16C5 core written in VHDL. The core is published under GPL (General Public License) on [26]. The basis core is significantly limited according to PIC16C57 specification. It is also impaired by several errors. Therefore, corrections were necessary, before the core could be used for the transmission channel circuit. The original code was written in VHDL (Very High Speed Integrated Circuit Hardware Description Language), so this HDL (Hardware Description Language) was used for the description of all required hardware, implemented in the FPGA. The following functional features of the PIC16C57 were either not present, and were added, or needed to be corrected:

- generator of MCU phases and properly operating 2-stage pipeline,
- four banks of registers in the register file,
- mapping registers with addresses 0x0 – 0xF from each bank to registers 0x0 – 0xF in bank 0,
- support for 2048-word program memory,
- assignment of status bits to their proper functions,
- addition and subtraction in ALU (Arithmetic Logic Unit),
- Timer, Prescaler and Option register.

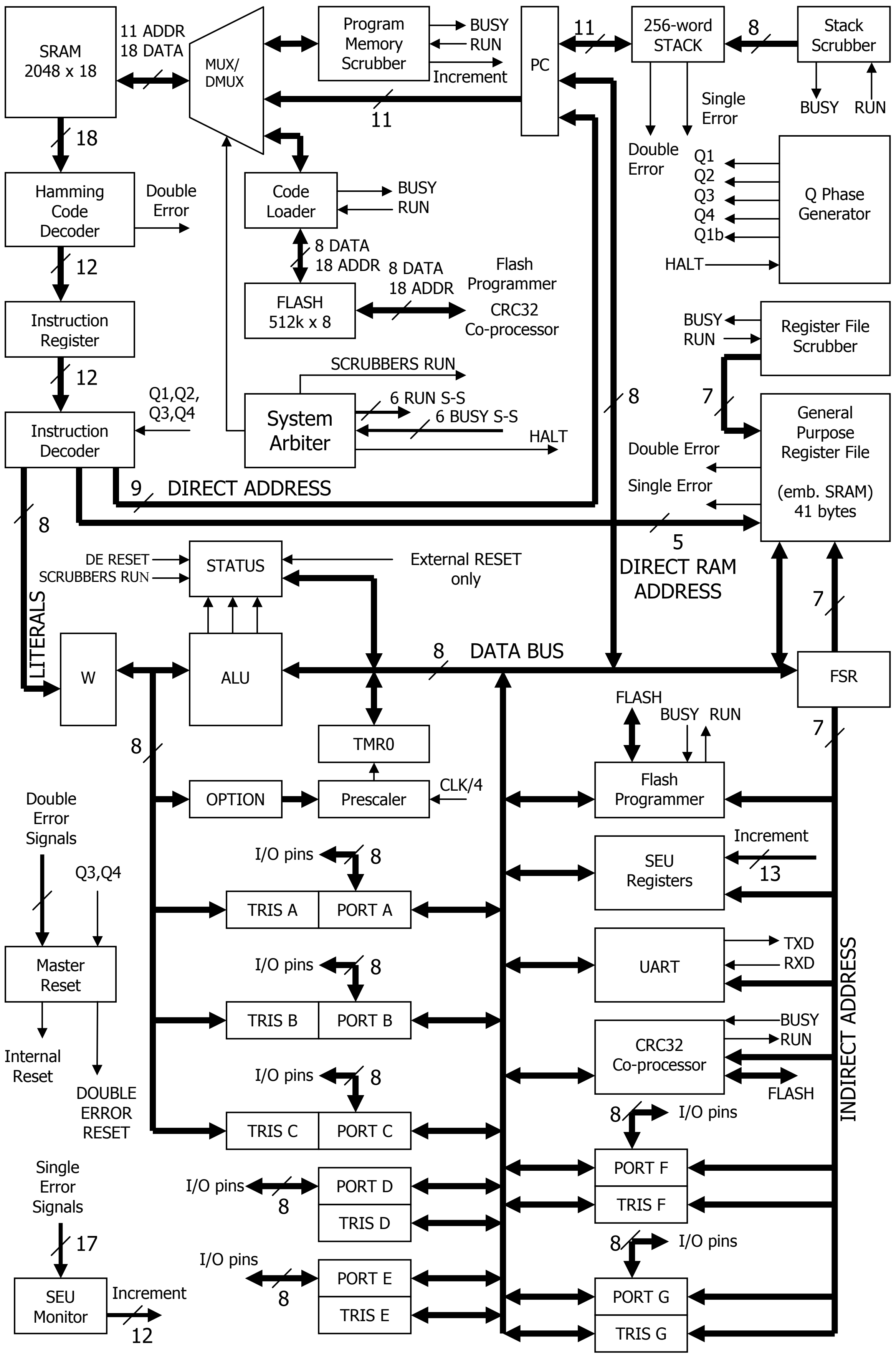


Figure 6.2. Functional block diagram of Radiation Tolerant MCU based on PIC16C57

The internal watchdog was not embedded due to existence of monitoring station, which also performs the function of an external watchdog.
To further extend the functionality of the circuit, several additional components were added. The main features of the modified MCU are summarised in Table 6.1.

Table 6.1. Summary of components of the modified MCU based on PIC16C57

Component	**Purpose**
7 GPIO ports, 8 independent I/Os each	Enables convenient communication with DUT
256-word deep stack, extended from 2 words	Improves reuse of code, effectively decreasing its size
CRC32 co-processor	Computes and verifies CRC32 for blocks of data, significantly decreasing size of code required for accomplishing this task; speeds-up program execution by taking-off computational load from the MCU core
UART	Handles serial communication compliant with EIA-232 protocol; takes-off communication handling from the MCU core, speeding-up program execution and reducing code size
Code Loader	After Reset or Power-On copies code from Flash to SRAM for faster code execution
Flash programmer	Handles Flash programming; extends programmability of PIC16C57 (the Microchip's device is OTP) to re-programmability
System Arbiter	Manages the MCU core and all embedded components, which require access to external memory

The most important requirement for the transmission channel circuit is improved radiation tolerance. For that reason the MCU is backed by specialised components. Moreover, to enable observability of SEUs in the MCU a monitor was also embedded. The components are summarised in Table 6.2.

Table 6.2. Summary of components necessary for improved radiation tolerance and diagnosis of the MCU

Component	Purpose
Program Memory Scrubber	Scrubs the Program Memory. If during scrubbing a single error in the memory is encountered, the appropriate SEU register is incremented.
Register File Scrubber	Scrubs register file
Stack Scrubber	Scrubs stack
SEU Monitor	Monitors groups of components for occurrence of SEU. Once a SEU is spotted, the appropriate SEU register is incremented.
SEU registers	Store the number of SEUs detected in the MCU. Content of such register is automatically cleared by hardware after reading.
Master Reset	Monitors for occurrence of double error in any of the SRAM components, resets the MCU if such error is spotted

Most relevant components are further described. Details on scrubbers and Master Reset can be found in section 6.2.2.1. The map of register file is collected in Table 6.4.
The Status register bits have in some cases different function than in the Microchip's PIC16C5. These differences have no impact on compatibility with existing tools for the MCU. The Status register bits, which have different function than original are described in Table 6.3.

Table 6.3. Modified Status Register bits

Bit	Function	Original function
3	Scrubbers Run flag. Set by hardware every time the scrubbers were run. Cleared by software. Enables software to check if scrubbers were run.	Power down. Used in conjunction with SLEEP instruction, which is not implemented.
7	Double Error Reset flag. Set by hardware prior to performing soft reset caused by double error in SRAM components. Cleared by software. Enables software to determine the cause of last restart: '1' – double error caused '0' – power-on or normal reset	Not used

Table 6.4. MCU's register file map

Address	Register name	Component	Description
0x00	Indirect	None	[24]
0x01	TMR0		
0x02	PCL		
0x03	STATUS		
0x04	FSR		
0x05	PORTA		
0x06	PORTB		
0x07	PORTC		
0x51	SEU_RS_SS	SEU register	SEU detected in Program Memory Scrubber or Stack Scrubber
0x52	SEU_A_RS_CL		SEU detected in System Arbiter, Register File Scrubber or Code Loader
0x53	SEU_CRC32		SEU detected in CRC32 co-processor
0x54	CRC32_GEN	CRC32 co-processor	Input/Output register
0x55	FP_CMD_STAT	Flash programmer	Table 6.10
0x56	FP_BLS_L		
0x57	FP_BLS_M		
0x58	FP_BLS_H		
0x59	FP_BA_LN_L		
0x5A	FP_BA_LN_H		
0x5B	FP_BUFF		
0x5C	PORTD	I/O port	Additional I/O ports
0x5D	PORTE		
0x5E	PORTF		
0x5F	PORTG		
0x70	TRISD	Tri-state control	Control the mode of each I/O pin (either input or output), depending on the stored value: '1' – input; '0' – output (compliant with PIC16C5)
0x71	TRISE		
0x72	TRISF		
0x73	TRISG		
0x74	SEU_IO_TRIS	SEU register	SEU detected in I/O ports' registers or their TRIS registers
0x75	SEU_PC_IR_STACK		SEU detected in Program Counter, Instruction Register or Stack
0x76	SEU_W_FSR_STAT		SEU detected in W register, FSR register or Status register
0x77	SEU_FILE_REG		SEU detected in Register File
0x78	SEU_PROG_MEM		SEU detected in Program Memory
0x79	SEU_UART_RX		SEU detected in UART receiver
0x7A	SEU_UART_TX		SEU detected in UART transmitter
0x7B	SEU_UART_TX_FIFO		SEU detected in UART transmitter FIFO
0x7C	SEU_UART_RX_FIFO		SEU detected in UART receiver FIFO
0x7D	SEU_OTHER		SEU detected in other components
0x7E	UART_DATA	UART	Data register
0x7F	UART_STAT		Status/configuration register

The modified MCU has two modes of operation:

- Software Upgrade Mode,
- Normal Mode.

In the Software Upgrade Mode no special protection mechanisms, apart from TMR and Hamming Codes, are active, but programming external Flash memory is enabled. On the contrary, in the Normal Mode, all the protection mechanisms are active, but Flash programming is disabled for safety reasons. In the implemented MCU, the mode is selected by a dedicated jumper (section 6.3.1). In each mode a different program is executed. The System Arbiter recognizes the selected mode and appropriately runs Code Loader. Both programs are stored in Flash memory. Table 6.5 shows the organisation of data in the Flash.

Table 6.5. Organisation of data in the external Flash memory

Sector	Start Address	Content	Size [words / bytes]
0	0x00000	"Firmware" – program for downloading new software and programming it into the Flash. Protected by Hamming Codes	2048 / 6144
1	0x10000	The "normal" program. Any code placed at this location cannot program the Flash. When this code is executed, it is fully protected with all available mechanisms. Protected by Hamming Codes	2048 / 6144
7	0x70000	Pre-computed table used by CRC32 co-processor.	256 / 1024

CRC32 co-processor

The function of CRC32 co-processor is to calculate the 32-bit Cyclic Redundancy Check (CRC32) value for a block of data. CRC32 is widely used in data storage and transmission, as it enables to verify whether a block of data is error-free. It does not enable to correct errors, merely detect them. In this project, the CRC32 was used for reliable data transmission. The method for calculating the CRC32 is table-based. The table with pre-computed values is stored in external Flash memory, sector 7. It occupies 1024 bytes, starting from the first address – each table entry is 4-bytes long, there are 256 entries. The implemented method of calculating CRC32 is essentially the same, as one used in RadMon [4]. The CRC32 for a block of data is computed byte-wise, i.e. for each byte of data the CRC32 value is updated and taken as initial for the next byte, until all bytes were processed. The final result is the last CRC32 value. Listing 6.1 is a piece of code, which illustrates this method.

Listing 6.1. Method for calculating CRC32 – code in C++

```
crc_accum = 0xdeadbeef;
for (j = 0; j < data_blk_size; j++){
      i = (static_cast<int>(crc_accum >> 24) ^ *data_blk_ptr++ ) & 0xff;
      crc_accum = (crc_accum << 8) ^ crc_table[i];
}
return crc_accum;
```

There is one register in the address space of register file, dedicated for the co-processor (address 0x54, i.e. bank 2, register 0x14). Upon reset, the CRC32 value is initialised with value 0xDEADBEEF. To calculate CRC32 for a block of data, consecutive bytes must be written to the register. A rule must be followed, when writing to CRC32 co-processor. In assembly, every two write attempts must be separated by at least one another instruction. Properly writing the bytes is the responsibility of software. After whole data block has been written, the computed CRC32 value can be read by software from the CRC32 register. To properly read computed CRC32, the read instruction may not directly follow the last write one, moreover, two read instructions may not follow each other directly. Since the register is 8-bits wide, the four bytes of CRC32 are available in following sequence: byte3, byte2, byte1, byte0. Thus, the resultant CRC32 is 0xbyte3byte2byte1byte0. The co-processor provides the CRC32 bytes in this order, for convenience. When transmitting a frame, the frame is sent byte by byte, at the same time the CRC32 for it is updated. Once frame data has been transmitted, the CRC32 follows. The bytes should be transmitted in byte3, byte2, byte1, byte0 sequence. This simplifies CRC32 verification on the receiver side. The whole received frame with the CRC32 bytes is subject to CRC32 calculation. If the result is 0x00000000, the frame was received correctly [27].

UART

The Universal Asynchronous Receiver Transmitter (UART) handles transmission and reception of serial data in compliance with the EIA-232 protocol. It enables full-duplex communication. The component supports only 8-N-1 signalling, which means: every frame contains 8 bits of data, there is no parity checking, there is one stop bit. The UART supports eight selectable baud rates, namely: 115200 bps, 57600 bps, 38400 bps, 28800 bps, 19200 bps, 9600 bps (default, selected upon reset), 1200 bps and 600 bps. The input RXD signal is sampled sixteen times per bit. The receiver has one-byte buffer. It manifests its status by means of following flags (active high): DREADY (a byte has been received), FRM_ERR (a byte has been received, but framing error occurred – there was no stop bit). The transmitter is non-buffered. Its status is manifested by the flag BUSY (active high), active when a byte is being transmitted. Due to lack of support for interrupts in the MCU, the receiver is further buffered by a 256-byte FIFO (First Input First Output), implemented in SRAM embedded in the FPGA. The same FIFO holds the value of framing error flag, corresponding to each byte stored in the FIFO. Dedicated FSM handles reading data and the flag from receiver and storing it in the FIFO. The MCU reads directly from the receiver FIFO. The transmitter is buffered by 256-byte FIFO at the input. The MCU writes data directly to the FIFO, not the transmitter. Dedicated FSM handles reading data from FIFO and commanding transmitter to send it, once the component is idle. Having a FIFO at the input to the transmitter speeds-up the process of sending data blocks, since the CPU does need to wait while transmitter sends a byte. This is true if the FIFO is not filled-up with data, which may occur for the slowest baud rates. The data register of the UART is mapped to register file address space. The value of the address is 0x7E, i.e. bank 3, register 0x1E. When MCU reads from the address, the head of the receiver FIFO is read, i.e. the oldest byte in the FIFO. When MCU writes to the address, the data is written to the transmitter FIFO. The status of UART is available for reading at address 0x7F, i.e. bank 3, register 0x1F. Reading the status returns current baud rate settings and following flags: receiver FIFO full, data in receiver FIFO available, transmitter FIFO full, framing error for last read byte. Writing the status register, which becomes configuration register for that time, is only necessary to change the baud rate. Table 6.6 gives a detailed description of the UART's status register, while Table 6.7 lists supported baud rate values of configuration register.

Table 6.6. Detailed description of UART Status Register

Bit Number	Name	Meaning	Reset value
0	F_ERROR	'1' when last read byte was received with framing error, '0' otherwise (updated every new byte is read by MCU)	0
1	TX_BUSY	'1' when transmitter FIFO is full – no data should be written, '0' otherwise	0
2	RX_READY	'1' when receiver FIFO is not empty – byte(s) is/are available for reading, '0' otherwise	0
3	RX_FULL	'1' when receiver FIFO is filled up with data in at least 75%	0
4	BR0	baud rate select bit 0	0
5	BR1	baud rate select bit 1	1
6	BR2	baud rate select bit 2	0
7	-	not used, read as '0'	0

Table 6.7. Baud rates supported by UART

Baud rate [bps]	Baud rate select bits (BR2, BR1, BR0)	Configuration register value
600	(0,0,0)	0x00
1200	(0,0,1)	0x10
9600	(0,1,0)	0x20
19200	(0,1,1)	0x30
28800	(1,0,0)	0x40
38400	(1,0,1)	0x50
57600	(1,1,0)	0x60
115200	(1,1,1)	0x70

Code Loader

The component is responsible for copying executable code from external non-volatile Flash memory to external volatile SRAM memory. Depending on the mode of MCU operation, the code is copied either from the 0th sector or the 1st sector. After a word is assembled and written in SRAM, the data is read back for verification. If verification succeeds, next address is processed, if not, another attempt is made, until verification passes successfully. Code Loader is run automatically by System Arbiter after a Power-On or Reset. While it is copying code, it sets BUSY flag.

Flash Programmer

The Flash Programmer is used for programming the external Flash memory. The component has been designed to extend the MCU with capability to be re-programmed. The original PIC16C57 is OTP, hence there are no dedicated instructions for accessing program memory, particularly writing it. The external Flash is not available in the CPU address space. The Flash Programmer is in fact an embedded device, which could be used in other designs, as well. The programmer comprises two sub-components: 7 kB data buffer and programmer FSM. The data buffer can be accessed by both MCU and the FSM, with restrictions. The MCU is only allowed to write to the buffer, the FSM can only read it. The Flash Programmer supports three operations:

- Erasing the whole Flash - Chip Erase,
- Erasing a particular sector of the Flash – Sector Erase,
- Programming the Flash from a given starting address, with block of data of given length (the length of the block may not be larger than 7 kB - the size of the data buffer).

The binary representation of commands is summarised in Table 6.8.

Table 6.8. Summary of commands supported by Flash Programmer

Command	Binary representation	Parameters
Chip Erase	0001 0000 (0x10)	None
Sector Erase	0010 0sss (0x2S)	Sector number “sss” (0 – 7) to be erased
Program	0011 0000 (0x30)	None

Once programming is complete, the status of the operation can be read from the Flash Programmer’s status register – Table 6.9.

Table 6.9. Possible values for status of Program command

Status value	Meaning
0x00	Programming completed successfully
0x01	The Flash failed to be programmed. This may be caused by an attempt to program a non-erased location or a malfunction of Flash’s embedded programming circuitry
0x02	The Flash was programmed, but verification of programmed data failed, i.e. the data read from Flash did not match the one stored in the buffer

The MCU communicates with the Flash Programmer through registers, mapped in the address space of the register file. The registers are collected in Table 6.10. To issue a command, the MCU has to write proper value to the Command/Status register. Writing data to the buffer is more complicated. The first step is to set proper address in the buffer, by setting Buffer Address Low/High appropriately. Next, the MCU must write the data to the Buffer Data Input register. Before the Program command is issued, software must ensure that:

- proper start address is written in the Start Address Low, Start Address Middle and Start Address High – the addresses determine the location in Flash, from which programming will commence,
- proper length of the data block stored in the buffer is written in the Buffer Address Low/Length Low and Buffer Address High/Length High registers.

No wait routines need to be implemented in software while Flash Programmer operates. The System Arbiter halts the MCU during this period, i.e. the software execution is paused.

Table 6.10. Flash Programmer registers

Address	Name	Function
0x55 (bank 2, register 0x15)	Command/Status	Writing command or reading status
0x56 (bank 2, register 0x16)	Start Address Low	Low byte of the starting address in Flash for programming
0x57 (bank 2, register 0x17)	Start Address Middle	Middle byte of the starting address in Flash for programming
0x58 (bank 2, register 0x18)	Start Address High	High byte of the starting address in Flash for programming
0x59 (bank 2, register 0x19)	Buffer Address Low / Length Low	Low byte of buffer address, when MCU writes data to the buffer; low byte of data block length, when Program command is issued
0x5A (bank 2, register 0x1A)	Buffer Address High / Length High	High byte of buffer address, when MCU writes data to the buffer; high byte of data block length, when Program command is issued
0x5B (bank 2, register 0x1B)	Buffer Data Input	Writing data to the buffer at address specified in above buffer address registers

System Arbiter

The System Arbiter manages the MCU and other components comprising the transmission channel circuit. It grants or denies access to external memory bus to Code Loader, CRC32 Co-processor, Flash Programmer and MCU. When the MCU is in Software Upgrade mode, System Arbiter switches off the scrubbers. When MCU is in Normal mode, System Arbiter never grants access to the external memory bus to Flash Programmer, but periodically runs scrubbers. The frequency of running scrubbers is hard-coded in VHDL and set to 1 250 000 instruction cycles. Since the MCU is clocked at 5 MHz, scrubbers are started once every 1 second. All three scrubbers are run in parallel, as they scrub different resources.

6.2.2 Techniques Employed for SEU Mitigation

This section gives insight into the techniques, which were used to mitigate the effects of SEU. Techniques for components based on SRAM, both embedded in the FPGA and external are described separately from those applied to sequential components, which were implemented in the FPGA. The only non-mitigated component is the Flash Programmer, since its usage is disabled in the radiation environment for safety reasons.

6.2.2.1 SRAM Components

There are a few components in the transmission channel circuit, which rely on SRAM modules. The following rely on SRAM embedded in the FPGA: stack, UART receiver FIFO, UART transmitter FIFO, register file and Flash Programmer buffer. The program memory is located in external SRAM. The Flash Programmer is not used in the Normal mode of MCU, which is dedicated for radiation environment. Hence, the buffer is not mitigated.

In chapter 5 various techniques for protecting memory were described. In the project, Hamming Codes were chosen to protect SRAM components. They require greater redundancy of data than 2-D Parity, when it comes to mitigating blocks of data, but are more suitable for mitigating single words. A short summary of Hamming Codes employed for protection of the earlier mentioned components is given in Table 6.11. Every code has the double error detection capability. This capability is made use of by the Master Reset component. This component monitors the double error indication output of every Hamming Code decoder. If a double error is spotted, a soft restart of the system is performed. Any double error is not correctable by Hamming Code. Therefore, the system enters an erroneous state, from which it cannot recover by means other restart. The program memory is monitored on falling edge of phase Q4, when an instruction is latched into the Instruction Register. The register file is monitored for double error in the middle of phase Q4, on when result of an ALU operation is stored. The stack is monitored for double errors if address is popped from it. The error will cause restart if it is spotted in the middle of phase Q4, when popped address is latched into the Program Counter. Every restart performed by the Master Reset is preceded with setting Double Error Reset flag in the Status Register. The UART FIFOs are not monitored for double errors, since existence of one does not result in malfunction of the system, merely communication error, which is easily detectable due to CRC32 and recoverable by retransmission.

Table 6.11. Hamming Codes employed for protecting SRAM components

Component	Hamming Code
Program Memory	(18, 12)
Register File	(13, 8)
Stack	(16, 11)
UART transmitter FIFO	(13, 8)
UART receiver FIFO	(14, 9)

The redundancy in the form of Hamming Codes has to be embedded in the description of the design. There is no algorithm for automatic or semi-automatic generation of such redundancy. The main reason for this is the way embedded SRAM components are instantiated in the design description for the ProAsic Plus FPGA family. It is best to let the instantiation be handled by ACTGen tool, a part of Libero IDE. In the initial phase of the process the required width of the SRAM block must be entered. The width corresponds to the length *n* of a (n, k) Hamming Code. Having such SRAM block, the only components left are the Hamming Code encoder at the input to the block and Hamming Code decoder at the output, suitable for the required code. These can be automatically generated by a piece of software, written for this occasion.

The Program Memory is, in this design, external to FPGA. It stores program code protected by (18, 12) Hamming Code. Every protected instruction is one 18-bit word. The process of program code protection takes place before the code is programmed into Flash. After a program is written, compiled and linked, an Intel HEX File or Binary file is generated. If only the former is available, it should be converted into Binary file. The Binary file is simply a raw image of Program Memory with the program code placed in it. The next step is to read every 12-bit instruction from the Binary file and protect it with Hamming code. This task is performed by a piece of software running on PC under MS-DOS or Windows. The result is a file with image of memory with protected program code. The file must be downloaded and programmed into Flash memory in Software Upgrade mode. Both the software upgrading code and the normal code must be protected by Hamming Code, otherwise the program code will not be understood by the MCU. During the "firmware" upgrade process, there exists a risk, that the operation will be interrupted while programming of flash memory is in progress. The vulnerable time is very short – less than 10 seconds. However, a power failure during this period will render the MCU incapable of being programmed in system. In such a case the flash module needs to be programmed externally. After correct software upgrading code is written, the MCU will re-gain its in-system programmability.

For data, which may be stored for extended period of time in SRAM, before being read by the MCU, such as data stored in register file or stack, simple protection by Hamming Codes may appear to be insufficient. The errors originating from SEUs will accumulate with time. It is probable, that a word will be affected by SEU more than once, rendering the data non-correctable. The result would be system reset or failure. To counteract this effect scrubbing needs to be employed. As it was described in section 5.5, scrubbing means periodically reading data from memory, correcting errors if any occurred, and writing back the data. Scrubbing is employed for protection of register file, stack and program memory. The described in section 6.2 System Arbiter is responsible for starting scrubbers periodically. Scrubbing of program memory is essential, as it cannot be written by the MCU, merely read. The UART's FIFOs are not scrubbed. Data is not stored in the components for extended periods of time, therefore the threat of multiple errors by accumulation is not significant. Moreover, if multiple errors occurred within a byte, they would not impair system stability as it was mentioned earlier in this section.

There is a drawback coming from scrubbing. Since it operates on memory components, which are used by MCU, the latter has to be halted during operation of scrubbers. This results in decreased MIPS value of the MCU, slower program execution. The time required by scrubbers to complete one pass is the time needed by Program Memory

Scrubber to scrub the program memory. This results from the fact that all scrubbers are run in parallel, and the mentioned scrubber refreshes the largest memory in the system, hence the others complete their tasks earlier. Every scrubber is clocked at 5 MHz. The Program Memory Scrubber must read, correct and write back 2048 words. The scrubber FSM has six states, and it will enter all six in the worst case, which is when every word is in error. The time required to scrub all 2048 words under such conditions is:

$$t_S = 6 \cdot 2048 \cdot \frac{1}{5} \cdot 10^{-6} = 0.0024576 = 2.46\,\mathrm{ms}$$

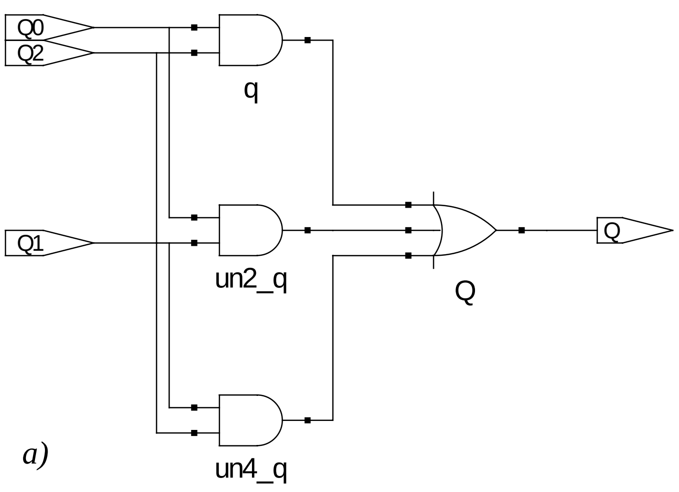
Now it is possible to compute the effective MIPS value, taking into account that scrubbers are run every second, and knowing that it takes 800 ns to execute one instruction.

$$MIPS_{eff} = \frac{1 - t_s}{t_{inst}} = \frac{0.99754}{800 \cdot 10^{-9}} = 1246925 = 1.247\,\mathrm{MIPS}$$

For comparison, the nominal value of MIPS for this MCU is 1.25 (5 MHz clock, every instruction takes four clock cycles to execute). Therefore, scrubbing causes decrease in the speed of computations by 0.25% - unnoticeably.

6.2.2.2 Sequential Components

The technique employed for protecting sequential components, such as FSMs or others, which rely on storing information in distributed memory elements – i.e. flip-flops – is TMR (Triple Modular Redundancy). As described in section 5.2 the TMR scheme enables to correct single errors on the level of tripled module. Since the transmission channel circuit is meant to facilitate communication, possibly with device being subject to tests, it should operate without human intervention. For that reason it is wise to apply the TMR scheme on the lowest possible level, according to [1]. Since it is impossible to alter the structure of logic tiles in the FPGA, the lowest possible level attainable from HDL is a single logic module. In the design, every D-type flip-flop is tripled. The three outputs are fed to the inputs of majority voting circuit (see Figure 6.3). The majority voting circuit outputs the value indicated by at least two flip-flops. Since there are only two logic levels, either '0' or '1', possible at the output of each flip-flop, it is always feasible to decide, i.e. choose the majority. Obviously, it may happen that an error will be double, then TMR makes a wrong decision, imposed by double error. Probability of such situation is, however, safely small.



a)

Q0	*Q1*	*Q2*	*Q*
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

b)

Figure 6.3. a) Glitch-free majority voting circuit for TMR scheme applied to single flip-flops; b) The truth table for voter function

The voting circuit is glitch-free, which means that there are no transient spikes at the output whenever one of the inputs changes. This is an important issue, since such glitches could be potentially latched into following flip-flop stages, resulting in arbitrary errors. The radiation tolerance or immunity of the majority voter is an important issue. The circuit is not protected against the influence of radiation. The SEU effect poses no threat on proper operation of the circuit, since the voter is purely combinatorial. The threat for such circuit is the SET effect, as described in section 3.1.2. This effect is mainly caused by heavy energetic ions. In the environment of the ILC or LINAC II accelerator, such ions are not expected at all. Moreover, the area occupied by a single voter is relatively small. It occupies only two cells in the FPGA. For these reasons the intrinsic radiation tolerance of the voter is accepted as sufficient. On the other hand, it is impossible to design a 100% radiation immune device on the level of HDL.

Complete tripled D-type flip-flop is depicted in Figure 6.4.

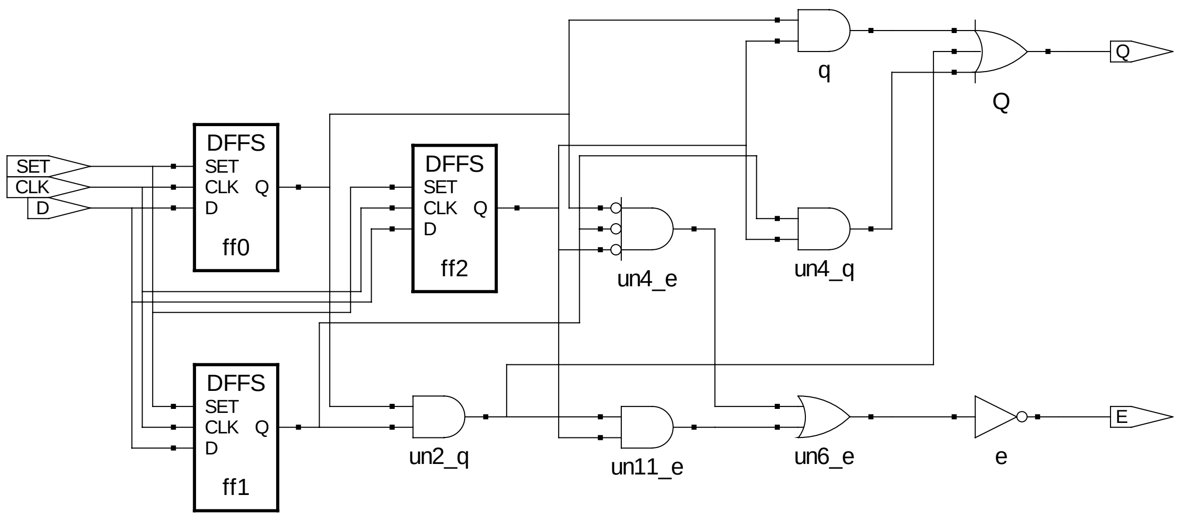


Figure 6.4. D-type flip-flop mitigated with TMR scheme; Q output driven by majority voting circuit; single error detection circuit added – output E.

The complete TMR D-type flip-flop is extended with error indication output. Whenever one of the sub flip-flops is altered by SEU, the single error detection output is set. This may provide information on the level of vulnerability of distributed memory elements of FPGA to SEU. Due to the fact that the error detecting circuit is sensitive to changes on every input separately, it is prone to produce glitches at the output. The problem is particularly visible for large designs, where delays of output signals from the sub flip-flops are unequal due to differences in routing lengths. In such a case the error detecting circuit produces "false alarms". The glitches settle as soon as all inputs are stable. Such problems were encountered as described in chapter 7.

The algorithm for applying the TMR scheme for every flip-flop is described in section 6.2.3.

The purpose of employing TMR scheme to sequential components is improving their reliability in the presence of radiation causing bit flips. As it was described in section 4.6, SEU induced errors may put an FSM into undefined state, an infinite loop or a halt. Every such situation can be escaped from without redundancy. It will not allow an FSM to operate with higher reliability for extended time in the presence of radiation, but will enable automatic soft restarting of the FSM, thus reducing or eliminating the need of supervisory. It is necessary to define all possible state transitions, and design the FSM in such a manner, that from every unused state a path will lead to the initial one. There also exist special encoding styles of the states. An encoding of a state is its binary representation. The so-called "one-hot" encoding is most reliable. In this scheme every state is represented as vector of '0s' with only single '1', e.g. "010", "100". Every state of an FSM is encoded with vector of the same length, and the single ‘1’ is placed in different location for each state. Therefore every single-bit error and some multiple-bit ones are spotted, as an undefined state and the FSM is brought to the initial state. This happens provided that the result of the errors does not yield a valid state description.

Once a register is mitigated with TMR, it should also be refreshed periodically, not to let errors accumulate. This, however, is done automatically. The clock is always supplied to the register’s flip-flops, it is never gated or disconnected (see Figure 6.5). If the condition for writing a new value to the register is not met, the value from output is written back on every active clock edge. This provides sufficient rate of refreshing register’s contents.

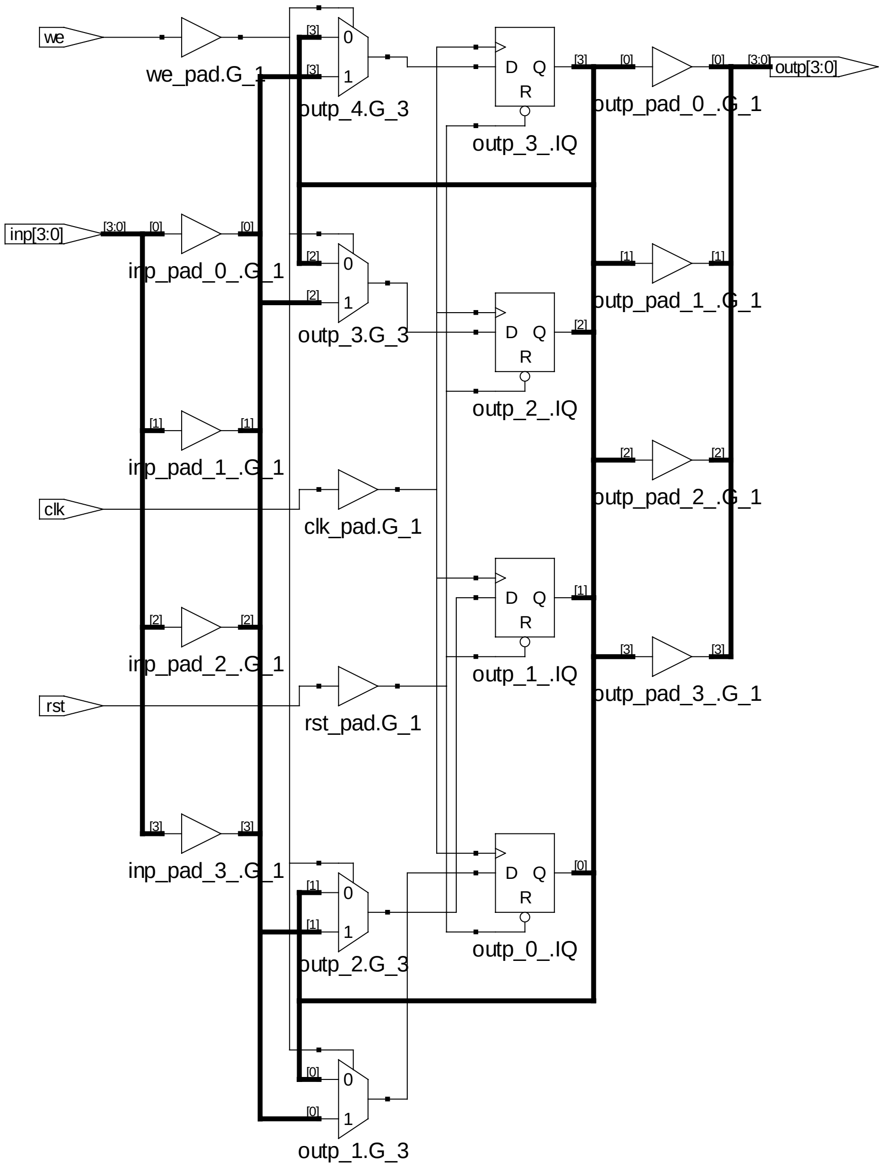


Figure 6.5. 4-bit register rising-edge active with asynchronous reset and synchronous write enable

6.2.3 Modified FPGA Design Flow for Improved Radiation Tolerance with TMR

This section gives a detailed description of the algorithm developed for applying TMR scheme for D-type flip-flops in the Actel's ProAsic Plus devices. The method can be applied to any device from this family. Modifications would be necessary to accommodate the technique for other device families, but this was not verified. For employing this scheme and better understanding its description, it is advised to read [28].

The starting point in applying the TMR scheme is description of a component in VHDL, such as one presented in Listing 6.2, describing a simple 3-bit up counter with asynchronous reset.

Listing 6.2. Exemplary 3-bit up counter described in VHDL

```
-- counter_3bit.vhd
-- A simple 3 bit up counter with asynchronous reset active low.
-- Counts on positive edge of the clock.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity Counter_3bit is
port(
    RST, CLK    :   in std_logic; -- asynchronous reset and clock inputs
    Q   :   out std_logic_vector(2 downto 0) -- the count value in binary
);
end entity Counter_3bit;

architecture Behavioural of Counter_3bit is
signal count : integer range 0 to 7; -- signal storing the count value
begin
    FSM : process (RST, CLK) is -- the counter FSM
    begin
        if RST = '0' then -- asynchronous reset active low
            count <= 0;
        elsif rising_edge(CLK) then -- count up on rising edge of CLK
            if count = 7 then -- check range for behavioural simulation
                count <= 0; -- roll-over
            else
                count <= count + 1; -- count up
            end if;
        end if;
    end process FSM;
    Q <= conv_std_logic_vector(count, 3); -- convert integer into binary

end architecture Behavioural;
```

The 3-bit counter is interpreted on RTL (Register Transfer Logic) level as shown in Figure 6.6.

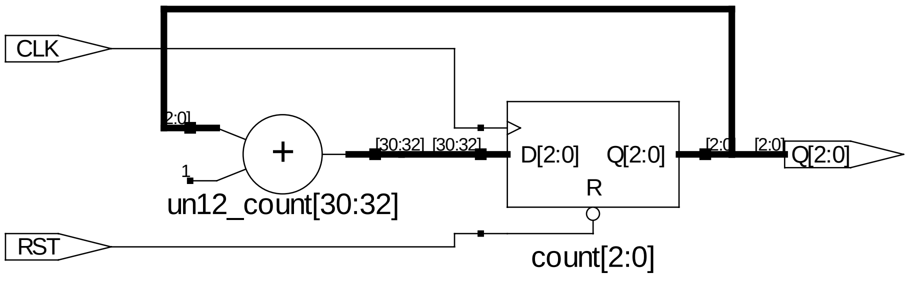


Figure 6.6. RTL schematic of the 3-bit counter

It is clearly visible, that the *count* value is stored in a register (3-bit register), which will be implemented in the distributed D-type flip-flops.

The next step, after assuring that counter works properly in simulation, is to synthesize the component. The process of synthesis is handled by dedicated tools. It follows the process of compilation, which transforms every description into RTL description. The RTL description is then used as a prerequisite for the synthesis process. During synthesis the RTL is *mapped* to the resources available in a particular FPGA device [28]. Therefore, the synthesis results in a *netlist*, which can be either an industry standard *EDIF* netlist or a structural VHDL description. The VHDL description is much more human-readable and can be easily modified. Listing 6.3 shows a VHDL netlist after synthesis of the 3-bit counter. Further actions are less tedious if the synthesis tool does not insert I/O pads into the netlist automatically. Such feature can be switched-off, e.g. in Synplify or Synplify Pro, which comes with Libero IDE following has to be done to switch the I/O insertion off: open *Project -> Implementation Options –> Device* and make sure that the *Disable I/O Insertion* checkbox is checked.

Listing 6.3. VHDL netlist of the 3-bit counter

```
library ieee;
use ieee.std_logic_1164.all;
library APA; -- library with components (various configurations of a
             -- logic tile, aka. CLB) specific for ProAsic Plus

entity Counter_3bit is
    port(Q : out std_logic_vector(2 downto 0); RST, CLK : in
         std_logic); -- I/O definition
end Counter_3bit;

architecture DEF_ARCH of Counter_3bit is
```

Listing 6.3 (cont.)

```
-- Declaration of components specific for the ProAsic Plus architecture
component XOR2FT
    port(A, B : in std_logic; Y : out std_logic);
  end component;

  component DFFC
    port(CLK, D, CLR : in std_logic; Q : out std_logic);
  end component;
  component PWR
    port(Y : out std_logic);
  end component;
  component INV
    port(A : in std_logic; Y : out std_logic);
  end component;
  component NAND2
    port(A, B : in std_logic; Y : out std_logic);
  end component;
  component XOR2
    port(A, B : in std_logic; Y : out std_logic);
  end component;
  component GND
    port(Y : out std_logic);
  end component;
-- declaration of signals
    signal \count[0]_net_1\, \count[1]_net_1\, \count[2]_net_1\,
        \un12_count_1.N_9_i_i_0\, CO2_0_o2_n, RST_i_0,
        \count_i_0[0]\, SUM1_0_x2_n, \VCC\, \GND\ : std_logic;

begin

    Q(2) <= \count[2]_net_1\;
    Q(1) <= \count[1]_net_1\;
    Q(0) <= \count[0]_net_1\;

    un12_count_1_SUM2_0_x2 : XOR2FT -- component instatiation

      port map(A => \count[2]_net_1\, B => CO2_0_o2_n, Y =>
        \un12_count_1.N_9_i_i_0\);

    \count[1]\ : DFFC -- a D-type flip-flop
      port map(CLK => CLK, D => SUM1_0_x2_n, CLR => RST_i_0, Q
         => \count[1]_net_1\);

    PWR_i : PWR
      port map(Y => \VCC\);

    \count_i[2]\ : INV
      port map(A => RST, Y => RST_i_0);
    un12_count_1_CO2_0_o2 : NAND2
      port map(A => \count[0]_net_1\, B => \count[1]_net_1\, Y
         => CO2_0_o2_n);

    \count_i[0]\ : INV
      port map(A => \count[0]_net_1\, Y => \count_i_0[0]\);

    \count[0]\ : DFFC -- a D-type flip-flop
      port map(CLK => CLK, D => \count_i_0[0]\, CLR => RST_i_0, Q
         => \count[0]_net_1\);
```

Listing 6.3 (cont.)

```
    un12_count_1_SUM1_0_x2 : XOR2
      port map(A => \count[0]_net_1\, B => \count[1]_net_1\, Y
         => SUM1_0_x2_n);

    GND_i : GND
      port map(Y => \GND\);
    \count[2]\ : DFFC -- a D-type flip-flop
      port map(CLK => CLK, D => \un12_count_1.N_9_i_i_0\, CLR =>
        RST_i_0, Q => \count[2]_net_1\);

end DEF_ARCH;
```

To mitigate the three flip-flops, which store the count value, with the TMR scheme, processing of the netlist is required and further described.

The *GND* and *PWR* components should be removed. They are a means of introducing constant ‘0’ and ‘1’ respectively, thus used for tying signals or outputs permanently to ‘0’ or ‘1’. If in a design a signal or output is permanently tied to ‘0’ or ‘1’, it will be connected to */GND/* or */VCC/* signal, respectively. Those signals are outputs from the two above components. Therefore, after removing the components, the mentioned signal or output must be explicitly assigned either ‘0’ or ‘1’.

After removal of *GND* and *PWR* components, netlist is ready for the proper TMR processing. At first, every D-type flip-flop component must be substituted by its TMR equivalent. The set of TMR flip-flops has been prepared during implementation of the project. In ProAsic Plus, there are following types of D-type flip-flops [28]:

- DFF
- DFFC
- DFFS
- DFFB
- DFFL
- DFFLC
- DFFLS
- DFFLB
- DFFI
- DFFCI
- DFFSI
- DFFBI
- DFFLI
- DFFLCI
- DFFLSI
- DFFLB

The “DFF” is the simplest D-type flip-flop rising edge active. If “DFF” is followed by “L”, it means the flip-flop is falling edge active. The letter “C” means that the flip-flop possesses an asynchronous CLEAR input, active high; “S” stands for asynchronous SET active high, and “B” designates that the flip-flop has both CLEAR and SET inputs, both asynchronous and active high. Finally, if an “I” is appended at the tail, the output from the flip-flop is inverted. Latches are neither described nor considered in the TMR scheme, as well designed synchronous system should not involve them. However, if needed, should be mitigated in exactly the same manner as flip-flops. For every flip-flop of the ProAsic Plus architecture a TMR counterpart has been designed, which differs in two respects from the basic version. First, the name of every TMR flip-flop starts with "TMR_" and next, one of the enumerated character sets follows. For example, a TMR counterpart of a rising edge flip-flop with asynchronous CLEAR is TMR_DFFC. The second difference between the two flip-flop families is the interface. The TMR flip-flops have an additional diagnostic output, which indicates whether an error has been spotted in any of the sub flip-flops. This output may or may not be used, it is the matter of design requirements. If information on SEU in FPGA flip-flops is needed, the output will be used, on the cost of more resources consumed. If only mitigation is required, no information on SEU is necessary, the output may be left unused. In the latter case fewer resources should be allocated to the design, as the synthesis tool should automatically remove some unused gates.

To mitigate a flip-flop the following steps must be taken. First, every DFF* component declaration must be substituted by TMR_DFF* declaration, as shown in Listing 6.4.

Listing 6.4. Substitution of DFF component declarations with their TMR counterparts*

```
  component DFFC
    port(CLK, D, CLR : in std_logic; Q : out std_logic);
  end component;
```

```
  component TMR_DFFC
    port(CLK, D, CLR : in std_logic; Q, E : out std_logic);
  end component;
```

The next step is to change every instantiation of DFF* component into instantiation of TMR_DFF* component, as shown in Listing 6.5.

Listing 6.5. Changing instantiation of DFF into instantiation of TMR_DFF**

```
    \count[0]\ : DFFC
      port map(CLK => CLK, D => \count_i_0[0]\, CLR => RST_i_0, Q
         => \count[0]_net_1\);
```

```
    \count[0]\ : TMR_DFFC
      port map(CLK => CLK, D => \count_i_0[0]\, CLR => RST_i_0, Q
         => \count[0]_net_1\, E => seu_count_0);
```

In the example of Listing 6.5 the single error detection output has been used, and connected to signal "seu_count_0". If the single error detection outputs are used, the respective signals must be declared in the signal declaration section – see Listing 6.3. The single error detection signals from all flip-flops of a component may be propagated to its additional output(s) in any desired manner. One must only remember to add the needed outputs to the entity declaration – see Listing 6.3. In this project, for every component the indications were "or-ed" and propagated as single error signal for the whole component. Having completed the described steps, one should arrive to a netlist, describing 3-bit counter mitigated by TMR, as shown in Listing 6.6.

Listing 6.6. The netlist of a 3-bit counter mitigated with TMR

```
-- tmr_counter_3bit.vhd
-- The 3-bit counter mitigated with TMR
-- Additional output (SE) was added to propagate the single error
-- indications
library ieee;
use ieee.std_logic_1164.all;
library APA;

entity TMR_Counter_3bit is
    port(Q, SE : out std_logic_vector(2 downto 0); RST, CLK : in
         std_logic);
end TMR_Counter_3bit;

architecture TMR_DEF_ARCH of TMR_Counter_3bit is
  component XOR2FT
    port(A, B : in std_logic; Y : out std_logic);
  end component;

  component TMR_DFFC
    port(CLK, D, CLR : in std_logic; Q, E : out std_logic);
  end component;

  component INV
    port(A : in std_logic; Y : out std_logic);
  end component;

component NAND2
    port(A, B : in std_logic; Y : out std_logic);
  end component;
  component XOR2
    port(A, B : in std_logic; Y : out std_logic);
  end component;

    signal \count[0]_net_1\, \count[1]_net_1\, \count[2]_net_1\,
        \un12_count_1.N_9_i_i_0\, CO2_0_o2_n, RST_i_0,
        \count_i_0[0]\, SUM1_0_x2_n: std_logic;

-- Single error detection signals
    signal seu_count_0. seu_count_1, seu_count_2 : std_logic;
```

Listing 6.6 (cont.)

```
begin
    Q(2) <= \count[2]_net_1\;
    Q(1) <= \count[1]_net_1\;
    Q(0) <= \count[0]_net_1\;

    un12_count_1_SUM2_0_x2 : XOR2FT
      port map(A => \count[2]_net_1\, B => CO2_0_o2_n, Y =>
        \un12_count_1.N_9_i_i_0\);

    \count[1]\ : TMR_DFFC
      port map(CLK => CLK, D => SUM1_0_x2_n, CLR => RST_i_0, Q
         => \count[1]_net_1\, E => seu_count_1);

    \count_i[2]\ : INV
      port map(A => RST, Y => RST_i_0);

    un12_count_1_CO2_0_o2 : NAND2
      port map(A => \count[0]_net_1\, B => \count[1]_net_1\, Y
         => CO2_0_o2_n);

    \count_i[0]\ : INV
      port map(A => \count[0]_net_1\, Y => \count_i_0[0]\);

    \count[0]\ : TMR_DFFC
      port map(CLK => CLK, D => \count_i_0[0]\, CLR => RST_i_0, Q
         => \count[0]_net_1\, E => seu_count_0);

    un12_count_1_SUM1_0_x2 : XOR2
      port map(A => \count[0]_net_1\, B => \count[1]_net_1\, Y
         => SUM1_0_x2_n);

    \count[2]\ : TMR_DFFC
      port map(CLK => CLK, D => \un12_count_1.N_9_i_i_0\, CLR =>
        RST_i_0, Q => \count[2]_net_1\, E => seu_count_2);

-- The single error detection signals collected into one signal by OR
-- operation
    SE  <=  seu_count_0 or seu_count_1 or seu_count_2;

end TMR_DEF_ARCH;
```

The name of the entity has been prefixed with “TMR_” to distinguish it from the original one. To stay consequent, the architecture name has also been prefixed with “TMR_”.

Once the netlist has been processed, the component is saved under the new name. In this manner a new component is created, which can be used as any other one, from the phase of design description. This means that the created TMR counter can be further used in a structural description of a larger system, and instantiated as a usual component. Exactly this approach was used during the design of the radiation tolerant MCU. The described process of mitigation is not limited to components only, whole unpartitioned designs can be protected in this manner as well. However, bottom-up approach is encouraged, as it enables easier testing, which should be done for every mitigated component.

Handling Hi-Z

In the ProAsic Plus devices there is no direct support for the high-impedance ‘Z’ logic state through tri-state buffers in the bulk of the FPGA. The Hi-Z can only be directly implemented at the I/O pads. If the TMR mitigation is to be applied on per component basis and some components are based on three-valued logic, the components need special treatment in the process of mitigation. To illustrate this, the 3-bit counter will be extended by output enable pin, which, when driven low puts the Q output into high-impedance state - Listing 6.7. The Hi-Z requires no particular treatment if mitigation is applied on the top-level entity.

For components involving the high-impedance, a *wrapper* is required. The TMR cannot be directly applied. The *wrapper entity* has the same interface as the original one. Modification is required for the architecture of the original entity. The high-impedance description must be moved to the wrapper. Since the OE input will not be used in the modified entity, it can be removed. Afterwards, the source component is mitigated according to the described algorithm. After completion a TMR version of the source component is obtained. This resultant mitigated component must be encapsulated in the wrapper. Listing 6.8 shows the VHDL code for the wrapper of the counter. The modified counter is exactly the same as one shown in Listing 6.2, while the mitigated one is as that of Listing 6.6. The wrapper is a purely combinatorial circuit. Hence, it is not necessary to mitigate it against SEU.

Thus obtained component, mitigated with TMR and having the capability of putting its outputs in the high-impedance state can be further used in structural descriptions of more complex systems or treated as the top-level entity and implemented.

Listing 6.7. The 3-bit counter with OE and Hi-Z

```
-- counter_3bit_HiZ.vhd
-- A simple 3 bit up counter with asynchronous reset active low.
-- Counts on positive edge of the clock.
-- The OE pin is output enable, actibe high. When low, puts the Q output
-- into high-impedance state

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity Counter_3bit_HiZ is
port(
    RST, CLK    :   in std_logic; -- asynchronous reset and clock inputs
    OE  :   in std_logic; -- output enable pin
    Q   :   out std_logic_vector(2 downto 0) -- the count value in binary
);
end entity Counter_3bit_HiZ;

architecture Behavioural of Counter_3bit_HiZ is
signal count : integer range 0 to 7; -- signal storing the count value
begin
    FSM : process (RST, CLK) is -- the counter FSM
    begin
        if RST = '0' then -- asynchronous reset active low
            count <= 0;
        elsif rising_edge(CLK) then -- count up on rising edge of CLK
            if count = 7 then -- check range for behavioural simulation
                count <= 0; -- roll-over
            else
                count <= count + 1; -- count up
            end if;
        end if;
    end process FSM;
    Q <= conv_std_logic_vector(count, 3) when OE = '1' else (others =>
'Z"); -- output the count value if OE is active, otherwise put Q into HiZ

end architecture Behavioural;
```

Listing 6.8. The wrapper for the 3-bit counter

```vhdl
-- wrap_counter_3bit.vhd
-- Wrapper for the 3-bit counter
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- The interface is identical to that of the original HiZ counter
entity Wrap_Counter_3bit is
port(
    RST, CLK    :   in std_logic;
    OE  :   in std_logic;
    Q   :   out std_logic_vector(2 downto 0)
);
end entity Wrap_Counter_3bit;

architecture Structural of Wrap_Counter_3bit is
component TMR_Counter_3bit is
port(
    RST, CLK    :   in std_logic;
    Q   :   out std_logic_vector(2 downto 0)
);
end component TMR_Counter_3bit;

signal encap_Q : std_logic_vector(2 downto 0);
begin
    encap_counter : TMR_Counter_3bit -- encapsulate mitigated counter
    port map
    (
        RST =>  RST,
        CLK =>  CLK,
        Q   =>  encap_Q
    );

    Q <= encap_Q when OE = '1' else (others => 'Z"); -- realise HiZ

end architecture Structural;
```

Summary

The algorithm for mitigating sequential components has been described. Below is a short summary of the steps, which need to be undertaken in the mitigation process.

1. Describe the component using VHDL – any style of description is possible, particularly behavioural.
2. If the component employs high-impedance, transform the component into one without Hi-Z, and describe wrapper for the component.
3. Synthesize the sequential component.
4. Process the obtained VHDL netlist:
    a. Replace every occurrence of “DFF” with “TMR_DFF” – in this way TMR flip-flop components are declared and instantiated instead of the normal ones.
    b. Add the “E” – single error detection – port to the declaration of every TMR_DFF* component.
    c. Connect single error detection outputs of the TMR_DFF* components if necessary.
    d. Propagate the single error signals, if used, to the output of the component – remember to add required I/O to the **port** clause.
    e. Prefix the original entity and architecture name with “TMR_”.
5. Save the processed netlist under original name prefixed with “TMR_” in the place where HDL files of the project are stored.
6. Test the mitigated component or design.

6.3 Hardware Platform

This section describes the hardware platform, on which the MCU was implemented and tested. The platform was designed according to the requirements described in section 2.2. Every PCB was designed and assembled exclusively for this project. Only COTS (Commercial Off-The-Shelf) components were used. Apart from external memory chips there is no component, which relies on data storage or is sequential. Therefore, no mitigation techniques were employed for the components. It was proved during tests that such techniques would not be necessary.

6.3.1 FPGA Development Board

The MCU was implemented in the Actel ProAsic Plus APA600 FPGA. Since there are no development boards for this FPGA available, a PCB was designed and assembled. The design is based on Actel ProAsic Plus Development Board, which comes either with APA75, APA150 or APA300 device. The necessity to design the board, gave the ability to include on it many additional components, not present on the stock development board. The board's overview is shown in Figure 6.7. Full schematic and PCB layout are attached in Appendix A. Following resources are available on the PCB:

- APA600 FPGA in PQ208 package (the board can house any other ProAsic Plus device in PQ208 package) [20],
- AMD Am29LV040B 512 kB Flash memory – sub-divided into 8 sectors of 64 kB size [29]. The memory chip is installed in a PLCC32 carrier. This enables off-the-board programming or simply replacing the module with another one,
- Renesas EDAC R1LV1616H-I 16 Mbit SRAM capable to operate in x16-bit or x8-bit word mode [30]. The memory has built-in Error Detection and Correction circuitry,
- Full-duplex 820 nm optical link based on Agilent COTS HFBR-1414 transmitter and HFBR-2412 receiver compatible with ST connectors, capable of transmission speeds from DC to 5 Mbits [31], [32], [33], [34], [35],
- differential point-to-point EIA-485 full-duplex link [36],

- full-duplex EIA-232 link [37],
- two clock generators with 3.3 V compatible outputs: 40 MHz and 3.6864 MHz – easily divisible for standard EIA-232 baud rates [38],
- regulated 12 V, 5 V, 3.3 V, 2.5 V [39], [40],
- 8 LEDs, 8 mono-stable push buttons,
- over 140 available I/O pins connected to the FPGA.

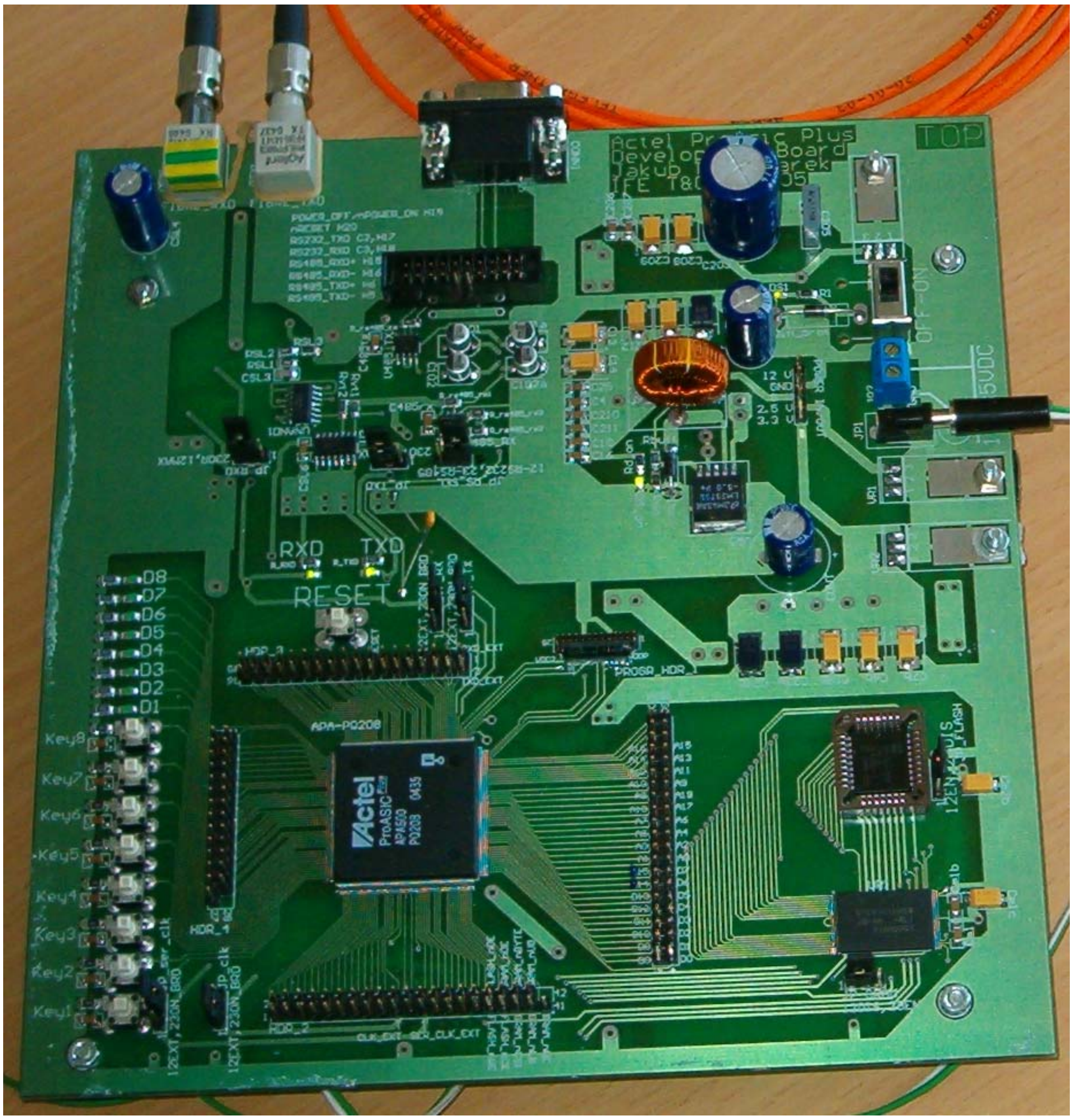
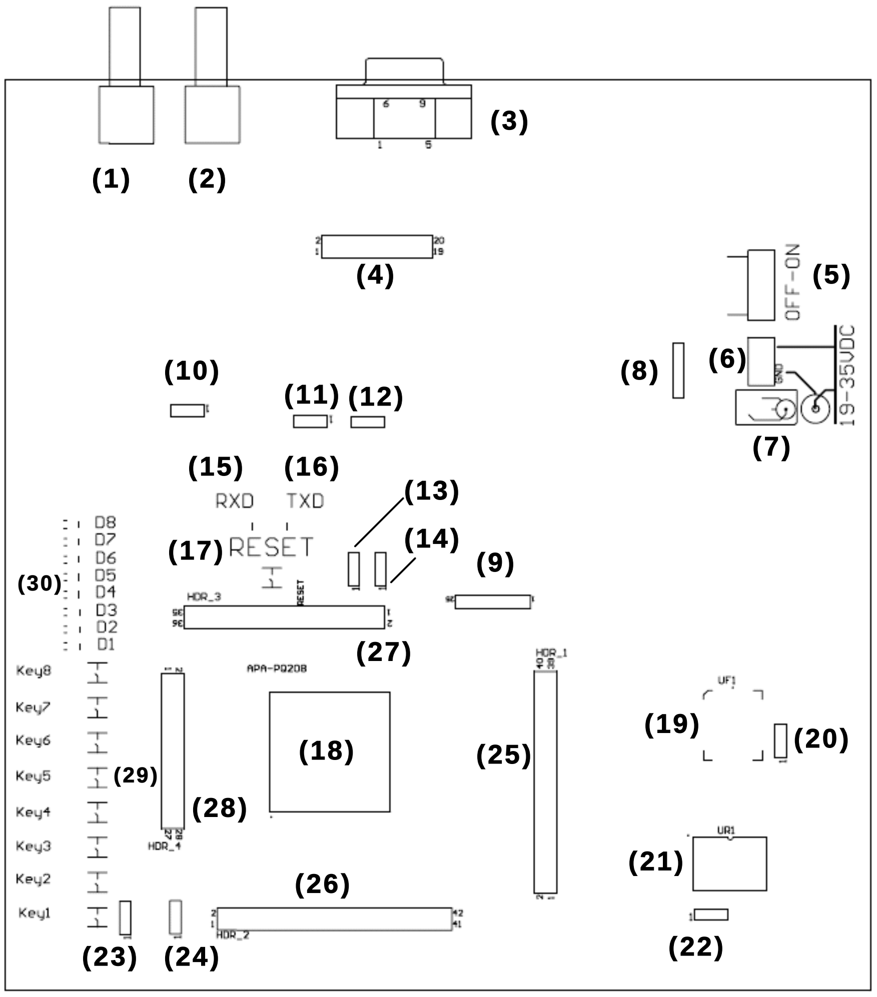


Figure 6.7. The designed FPGA development board

Some resources are connected to the FPGA through jumpers, which allow deciding if and which resource is used. The LEDs and buttons are permanently connected.

The symbolic view of the PCB is shown in Figure 6.8, together with explanation and designation of all jumpers.



1. Optical TX
2. Optical RX
3. EIA-232 female connector
4. 20-pin male header
5. Master On-Off switch
6. Power Connector
7. Power Connector
8. Power Out/In male header
9. FlashPro Lite programmer connector for APA programming
10. RXD select
11. TXD select
12. MAX RXD select
13. RXD master select
14. TXD master select
15. RXD LED
16. TXD LED
17. Reset button
18. APA600
19. Flash
20. Flash enable
21. SRAM
22. SRAM enable
23. Clock select 1
24. Clock select 2
25. Header 1
26. Header 2
27. Header 3
28. Header 4
29. Buttons
30. LEDs

Figure 6.8. Symbolic view of the FPGA development board with main components explained

To configure the serial communication channel following jumpers must be properly set:

1) (13) - RXD Master select: allows choosing the source of signal for receiver. When set in position 1-2, external source connected to Header 3 pin 1 is connected to FPGA; when set in position 2-3, one of the on-board sources is connected. The external source must produce either 3.3 V logic levels. The voltage may not swing above 3.3 V, as the I/O pads of the FPGA are not tolerant to higher voltages.
2) (14) – TXD Master select: allows to direct signal from transmitter to either an external transducer (Header 3 pin 2) or other component (position 1-2) or to one of the on-board components (position 2-3). This jumper was installed to decrease overall current consumption. The external device, which is fed with the signal, must be input-compatible with 3.3 V logic levels.

3) (10) – RXD select: selects the type of on-board source for receiver. When set in position 1-2 the source signal will be collected from copper lines (either EIA-232 or EIA-485), when in position 2-3, the signal from optical receiver (2) will be passed on to jumper (13).
4) (11) – TXD select: if the jumper (14) is in position 2-3, this jumper further couples signal from transmitter either to copper lines (position 1-2) or the optical transmitter (position 2-3).
5) (12) – MAX RXD select: this jumper allows choosing which copper serial link will feed signal to the receiver. When set in position 1-2 the EIA-232 is selected, while when configured in 2-3 the EIA-485 signal is coupled.

Table 6.12 gives a summary of possible configurations of both TXD and RXD lines for on-board circuitry.

Table 6.12. Possible serial line configurations

TXD signal transmitted by	**Jumper settings (14) always in 2-3**	**RXD signal received from**	**Jumper settings (13) always in 2-3**	
	(11)		**(10)**	**(12)**
EIA-232	1-2	EIA-232	1-2	1-2
EIA-485	1-2	EIA-485	1-2	2-3
Optical TX HFBR-1414	2-3	Optical RX HFBR-2412	2-3	Any

The jumpers (23) and (24) allow to choose the clock sources fed to the FPGA.

Jumper (23) couples clock signal to Global Pad 24 of the FPGA from Header 2 pin 19 – external source if installed in position 1-2. The clock from on-board 40 MHz generator is coupled when the jumper is installed in position 2-3.

Jumper (24) couples clock signal to Global Pad 30 of the FPGA from Header 2 pin 23 – external source if installed in position 1-2. The clock from on-board 3.6864 MHz generator is coupled when the jumper is installed in position 2-3.

The on-board Flash and SRAM memories can be disconnected from FPGA pins by means of their enable jumpers. The Flash memory is disconnected by placing jumper (20) in the 1-2 position. The memory is connected to the FPGA, when the jumper is placed in position 2-3. Disconnecting the SRAM memory is accomplished by installing jumper (22) in position 1-2, while connecting it – by installing the jumper in position 2-3. The connection or disconnection is realised through tri-state buffers embedded in the memory modules. The enable jumpers simply pass proper logic levels to the CE (Chip Enable) inputs of the memory modules.

The Reset signal can be fed from various sources, too. The on-board source is the Reset button (17). The communication header (4) has a dedicated pin for Reset, as well (Table 6.13). Yet another source of Reset might be connected to the Header 3 pin 15, which further connects to Global I/O pad 128 of the FPGA, which is permanently connected to the Reset signal, hence signals from all sources meet at this pin. There are two conditions, which must be fulfilled by each Reset signal. The signal is active low, i.e. will reset the circuits implemented in FPGA when its value is logic '0'. The inactive voltage level is '1', which must be represented voltage compatible with 3.3 V levels. Voltages higher than 3.3 V are not allowed.

The communication header (4) is described in detail in Table 6.13.

The board must be powered from a regulated DC power supply. The input voltage must not exceed 35 V. If the on-board regulated 12 V is required, the minimum value of the DC input voltage is 19 V, otherwise the value may not fall below 6 V.

The VDDP voltage, which supplies I/O pads of the FPGA, is permanently set to 3.3 V.

The PCB was designed and assembled before it was decided which MCU will be implemented in the FPGA. After the choice was made, it appeared that the 16-bit wide data bus offered by the on-board SRAM is not wide enough. All PIC16C5 cores have 12-bit instruction word. Employing Hamming Code to protect every instruction word further extends the instruction word to 18 bits (see Table 6.11). To stay directly compatible with PIC16C5, i.e. let the MCU fetch every instruction in a single cycle, additional SRAM was required. An expansion board was designed and assembled. It houses the additional SRAM. The board was designed with an exemplary application of the transmission channel circuit in mind – SRAM SEU detector. A snapshot of the expansion board is shown in Figure 6.9. The schematic of the expansion board and PCB layout are attached in Appendix A.

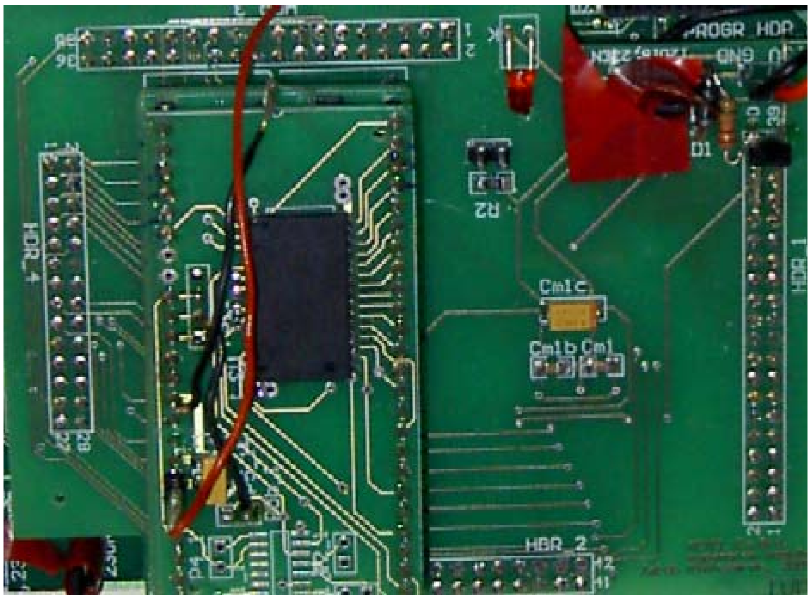


Figure 6.9. Photograph of the designed expansion board

The additional SRAM memory is not visible on the photograph, since it is soldered on the bottom side of the PCB. The vertically oriented rectangular PCB with clearly visible memory chip is a module of SRAM memory, being subject to test. More information on this is given in section 6.4. The expansion board is equipped with a proprietary RadMon socket for standardised interfacing to tested memories. The symbolic view of the expansion board is presented in Figure 6.10.

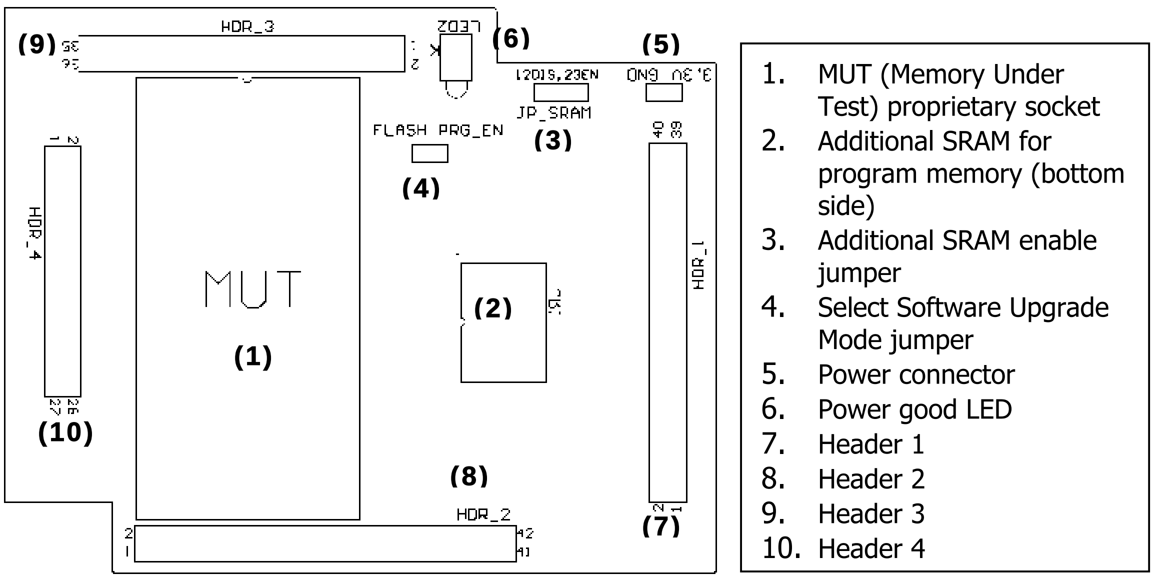

(9)
HDR_3
(6)
(5)
JP_SRAM
FLASH PRG_EN
(3)
(4)
HDR_4
MUT
(1)
(2)
HDR_1
(10)
(8)
HDR_2
(7)
1. MUT (Memory Under Test) proprietary socket
2. Additional SRAM for program memory (bottom side)
3. Additional SRAM enable jumper
4. Select Software Upgrade Mode jumper
5. Power connector
6. Power good LED
7. Header 1
8. Header 2
9. Header 3
10. Header 4

Figure 6.10. Symbolic view of the designed expansion board

The socket for Memory Under Test (MUT) (1) is used in the exemplary application of the transmission channel circuit, detector of SEUs in MUT. The additional SRAM module for program memory is capable of being connected to or disconnected from the address and data buses. It is realised in exactly the same manner as in the development board. The SRAM is connected to the bus, hence the FPGA, when jumper (3) is installed in position 1-2, disconnected when the jumper occupies position 2-3. There is an error on the Top Overlay of the PCB. The enable and disable positions of the jumper are swapped. In section 6.2 the two modes of MCU's operation were described. The mode is selected by means of jumper (4). When the jumper is installed, the MCU operates in the Software Upgrade Mode. The Normal mode is selected by removing the jumper, leaving the two pins unconnected. This jumper is named "FLASH PRG_EN" on the PCB, since its implicit function is disabling or enabling the MCU to program the Flash. After selection of the mode has changed (jumper was installed or de-installed), the system must be restarted, by means of the Reset signal or button. The expansion board connects to the development board by means of the general-purpose I/O pins collected in headers.

The power is delivered to the board via connector (5). Both ground and 3.3 V must be supplied from the development board through wires. The LED (6) indicates that power supply is connected correctly, by emanating red light.

6.3.2 Transceiver

In the radiation-free environment a transceiver is necessary, as stated in earlier sections. The transceiver PCB has also been designed and assembled. Its photograph is presented in Figure 6.11, while symbolic view is depicted in Figure 6.12. The complete schematic and PCB layout can be found in Appendix A. The function of the transceiver is to provide a centralised connection between various communication interfaces and media. From the point of view of a monitoring station (e.g. a PC) the following communication interfaces are available:

- EIA-232 based on MAX232 [37],
- EIA-485 based on MAX485 [36],
- 5 V logic.

The interfaces available from the FPGA point of view are as follows:

- EIA-232,
- EIA-485,
- Optical link identical to the one described in section 6.3.1.

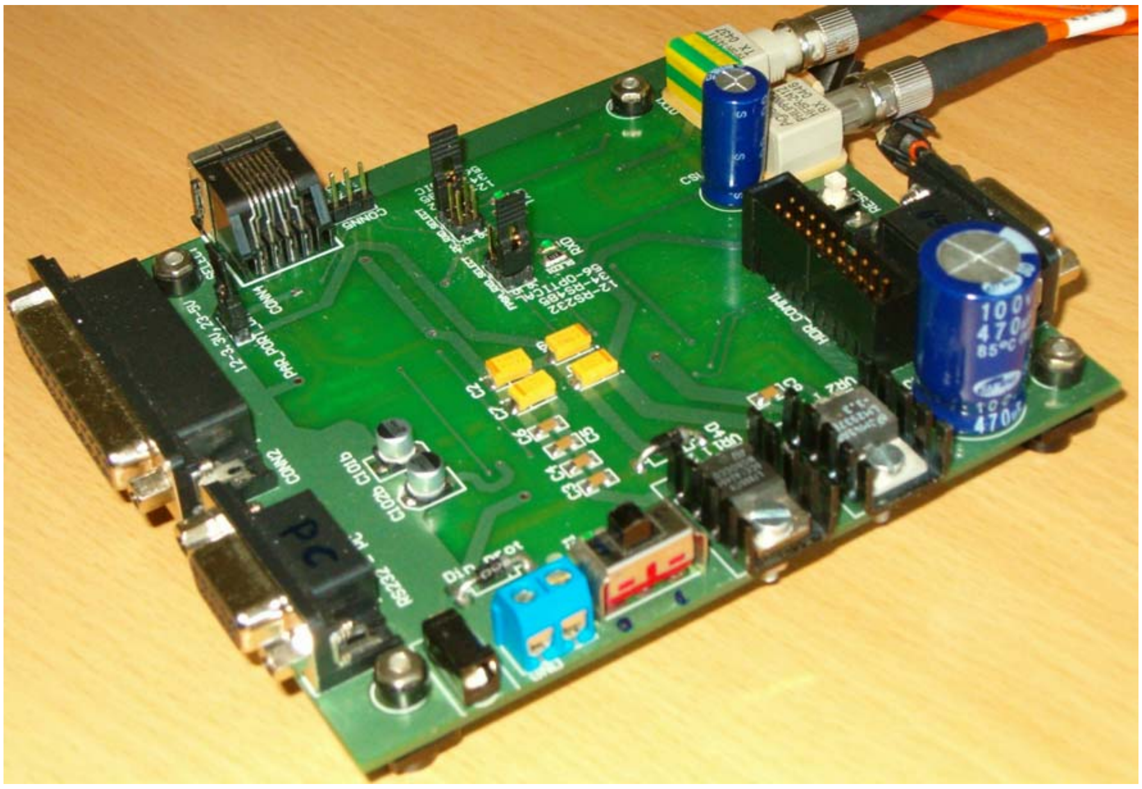
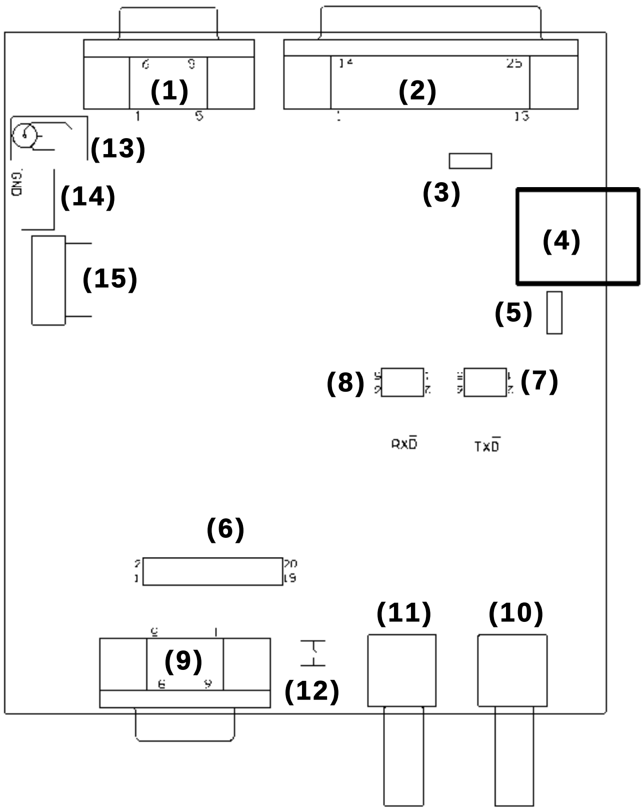


Figure 6.11. The transceiver PCB



1. EIA-232 socket for PC (DB9)
2. Parallel port socket for PC (DB25)
3. Parallel port VCC select
4. RJ-45 socket for EIA-485
5. Header for 5V logic
6. 20-pin header
7. PC RXD select
8. FPGA RXD select
9. EIA-232 socket for FPGA (DB9)
10. Optical transmitter
11. Optical receiver
12. Reset button
13. Power connector
14. Power connector
15. Master On-Off switch

Figure 6.12. Symbolic view of transceiver PCB

Selecting proper communication link set-up is fairly easy. The only choice that has to be made applies to the source of received signal. The jumper (7) is used to select the source of communication received from the monitoring station. Installing the jumper in position 1-2 causes that signal coming from the EIA-232 socket (1) is transmitted to the FPGA. When this jumper is installed in position 3-4, FPGA will receive signal coming from EIA-485 socket (4). Finally the logic signals will be transmitted to FPGA if the jumper is set in position 5-6. Regardless of the selected source, the signal from monitoring station is transmitted via all three available interfaces to the FPGA.

In a similar manner the source of signal coming from the FPGA is selected by jumper (8). Again, regardless of the selected source, the received signal is transmitted via all three available interfaces to the monitoring station. If jumper (8) is installed in position 1-2, the signal coming from FPGA over EIA-232 will be transmitted to the monitoring station. To transmit signal, which was received from FPGA's EIA-485, the jumper must be placed in location 3-4. Finally, to use the optical link for reception of data from FPGA, jumper (8) must be set in position 5-6.

The parallel port socket (2) enables passing the Reset and Power-Off signals from the monitoring station to the FPGA. Both 3.3 V and 5 V logic levels are supported, and selected by means of jumper (3) (position 1-2 selects 3.3 V, position 2-3 selects 5 V).

The transceiver board should be powered from a regulated DC voltage supply. The input DC voltage may range from 6 V to 35 V. The supplied power can be delivered to the FPGA board if the two are connected with 20-wire ribbon cable, described in Table 6.13. In that case the input voltage must follow the guidelines given in section 6.3.1. The master on-off switch (15) will toggle power on or off on both FPGA and transceiver boards, if the supply to FPGA is carried over the 20-wire ribbon cable.

It is important to provide proper voltage levels on the Reset and Power-Off inputs of the parallel port socket, when FPGA board and transceiver are interconnected with the 20-wire ribbon cable. It this requirement is not fulfilled, the 5 V regulator on FPGA board will be switched-off. Hence no power will be delivered to the FPGA and communication circuitry.

Table 6.13. Description of the 20-pin header

Pin number	Signal name	Function
1	GND	Common / Ground
2		
3		
4		
5	EIA-485 RX Inverting	EIA-485 received from FPGA. NOTE: In the FPGA connector, corresponding TX signals are located on these pins
6	EIA-485 RX Non-Inverting	
7	GND	Common / Ground
8		
9	Supply	The DC power supply delivered to FPGA
10		
11		
12		
13		
14		
15	EIA-485 TX Non-Inverting	EIA-485 transmitted to FPGA. NOTE: In the FPGA connector, corresponding TX signals are located on these pins
16	EIA-485 TX Inverting	
17	EIA-232 RXD	EIA-232 received from FPGA
18	EIA-232 TXD	EIA-232 transmitted to FPGA
19	nPower-ON / Power-OFF	Active high Power-Off signal for 5 V regulator on the FPGA board (connected to pin 3, i.e. D1, of the parallel port)
20	Reset	Active low reset (connected to pin 2, i.e. D0, of the parallel port)

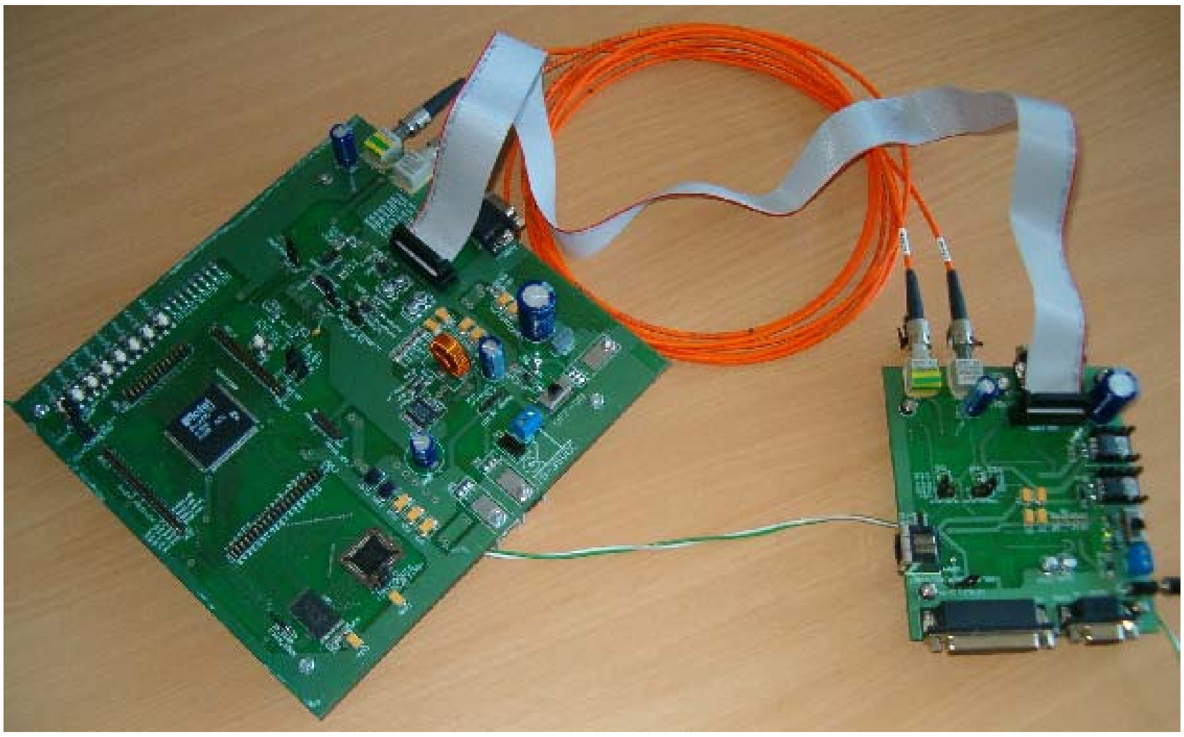


Figure 6.13. The FPGA and transceiver boards interconnected with full-duplex fibre optic link and 20-wire ribbon cable

6.4 Exemplary Application – SRAM SEU Detector

The SRAM SEU detector demonstrates the capabilities of the designed radiation tolerant microcontroller and the communication channel components. Such a detector can be used as a module of a radiation tolerant system for on-line measurement of neutron fluence in the ILC, LINAC or other accelerator facility. Such measurements are vital for assessing radiation dose absorbed by electronic devices. This is important for both, accelerator's control systems operating in radiation environment, to estimate remaining lifetime, and various devices under test, while their performance in radiation environment is examined. The SRAM memory, as explained in section 3.1.2 is particularly vulnerable to neutron induced SEUs. Therefore, it can be used as a neutron sensor [4]. Installing memory chips with various packing densities and sizes can alter the sensitivity of the detector. The communication between the MCU and Memory Under Test (MUT) is realised through MCU's GPIO ports. The MUT is installed in a proprietary RadMon socket. This facilitates replacement of the memory modules and establishes a common interface, regardless of the memory used. The arrangement is schematically depicted in Figure 6.14.

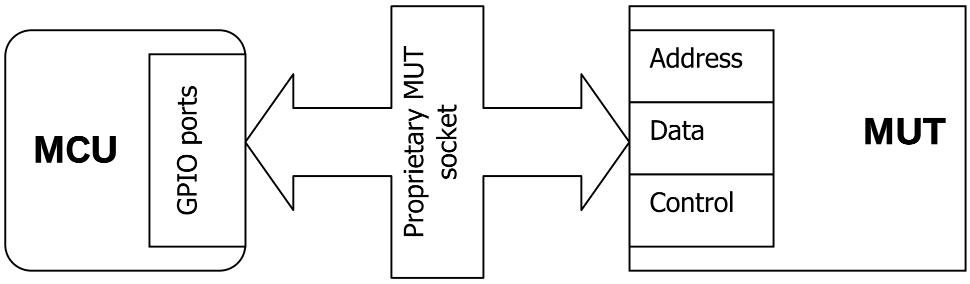


Figure 6.14. Schematic representation of SRAM SEU detector

The detection of SEUs in the MUT is based on the bit-flip effect described in section 3.1.2. The major task of the detector is to spot the distortion of data and immediately report it to the monitoring station or another central system. The detector must listen to the commands coming from the central system and react in an appropriate way. Reliable exchange of information is accomplished according to proprietary frame-based RadMon protocol. Every frame is appended with CRC32 checksum.

6.4.2 Software for the MCU

The MCU is responsible for detecting SEUs. This task is accomplished in the following way. The MUT module address space is subdivided into two sectors of equal size. The sectors are designated Low and High. Before the MCU commences SEU detection, it initialises the memory by writing it with known contents, upon reception of a proper command. The pattern for initialisation is a parameter to the command. The received pattern is written into the MUT in a pair-wise manner. The same data is written at location *n* in the Low sector and location *n* in the High sector. The memory cells of address *n* form the pair of cells. Once all locations have been written, the detection process starts. The detection is accomplished by continually reading data from a pair of cells and comparing it. If data read from cell of Low sector does not match the data read from cell of High sector, SEU has been spotted and is reported by sending appropriate information. Finally the contents of the cells are “fixed”, by writing data read from Low sector cell to the High one. Afterwards the process continues for next pair of cells. If whole MUT has been swept, the process starts again from the first pair of cells. The central system can issue a command at any time during detector’s operation. From the moment of being switched-on, the detector checks each SEU register once every ten seconds and reports SEUs, which occurred in the MCU. All these tasks are realised in software, executed by the designed radiation tolerant microcontroller. The software is written in C language, and was compiled in CCSC compiler version 3.203 [41]. Full source code is available on the attached CD-ROM.

The MCU, similarly as original PIC16C57, has no support for interrupts. Therefore, the necessary software routines are executed in a *round-robin* manner. The support for interrupts would be particularly helpful for receiving data, but is not necessary.

The main loop governing detector's operation is shown in Listing 6.9. This loop is run infinitely, after initialisation of the detector.

Listing 6.9. The main loop of SEU detector software

```
while (1)
{
   PORTG = radmon_status; // signal detector's status by means of LEDs
   run_receiver();   // run the receiver FSM
   start_new_command();  // run new command if any received
   continue_command();   // continue running the command
   check_scrubbing_status(); // see if scrubbers were run
                             // and react appropriately
}
```

The **run_receiver()** function runs a software receiver FSM, which is responsible for reception of data according to the RadMon protocol. The FSM is realised in the **run_rx_fsm()** function. Code for the FSM is presented in Listing 6.11. This fragment of code is very important mostly for one reason. Before the **switch** clause, some code in assembly is added. The purpose of this code is to fix a compiler error. The error was discovered during development of the software. Other compilers were not tested in this respect. The error was encountered with CCSC compiler and seems to recur in all **switch** clauses and can be analysed in assembly, only. The error and correction are indicated in Listing 6.10.

Listing 6.10. Switch clause error in CCSC compiler and assembly of a fixed code

```
; Erroneous code
00C0:  MOVF   08,F ;0x08 arbitrary
00C1:  BTFSC  03.2
00C2:  GOTO   0CC
00C3:  MOVLW  08
```

```
; Fixed code
00C0:  MOVF   10,W ;copy variable
00C1:  MOVWF  08   ;to 0x08 reg.
00C2:  MOVF   08,F ;proceed as before
00C3:  BTFSC  03.2
00C4:  GOTO   0CE
00C5:  MOVLW  08
```

The compiler assumes that the variable being the subject of **switch** clause has been stored in register 0x08 (line 00C0 of erroneous code), but this did not take place. The register stores an arbitrary value, hence the flow of the clause is unpredictable. After investigating each **switch** clause in the software, it appeared that register 0x08 is always assumed to store the variable's value. This was also confirmed by compiling the code several times – always register 0x08 was used. The remedy was, thus, to manually copy the required variable's value into the 0x08 register, to make the **switch** clause behave properly. As it was later discovered, the 0x08 register is used, since it is the first register of *scratchpad*, used by compiler for some intermediate operations. If the *scratchpad* is located at another address, this should be taken into account, when correcting the error.

Another problem encountered with the used CCSC compiler refers to indirect addressing. If this addressing mode is used by compiler to implement usage of pointers or access elements of an array in a loop, the program starts to behave unpredictably. The FSR register is used in indirect addressing mode. The full address of the register being accessed must be written to the FSR. Reading or writing the indirectly addressed register is accomplished by performing this operation on the INDF register. Problem lies in the usage of the FSR. Its second function is to select the register bank for direct addressing modes. The compiler seems to be ignoring this fact and does not restore proper selection of bank after completing the indirect access. When following code refers directly to e.g. loop control variable, there is high chance that incorrect bank is selected for this operation. As a result the right variable is not accessed, other data is corrupted and loop execution is not as indicated by source code. Correcting this error is done in assembly, but is not as straightforward as the former one. Every loop or other case of using indirect addressing mode should be separately analysed. During this process full addresses of involved variables are necessary. Attention must be paid to select proper register bank after the indirect access.

No other compiler errors were encountered. The whole software is not described here in detail, as it would unnecessarily inflate the thesis' volume. The code is well documented and smartly partitioned into functions, those are collected into header files. Therefore, it is easy to read and understand the code, available on the attached CD-ROM

The operation of the detector follows the flow diagram depicted in Figure 6.15.

Listing 6.11. Software implementation of receiver FSM for SEU detector

```
// Fix CCSC switch bug
#ifdef CCSC_COMPILER
#asm
   MOVF   rx_state, W
   MOVWF  0x08
#endasm
#endif //CCSC_COMPILER
switch (rx_state) {
   case RX_WAIT   :
      if (c == BOF) { // beginning of frame received
         set_rx_state_normal();
         // set FS bit - new frame started
         #asm
            BSF radmon_status, RS_FS
         #endasm
         rx_buffer_pt = 0; // reset buffer write pointer
         return; // exit function, nothing to process
      }
      else { // otherwise wait for BOF
         return;
      }
   case RX_NORMAL :
      if (c == SES) { // escape sequence initiated
         set_rx_state_escape();
         return; // exit function without processing character
      }
      else if (c == BOF) { // resync. RX
         #asm
            BSF radmon_status, RS_FS
         #endasm
         rx_buffer_pt = 0; // reset buffer write pointer
         return; // exit function, nothing to process
      }
      break;
   case RX_ESCAPE :
      if (c == BOF_SUBS) {
         c = BOF;
      }
      else if (c == SES_SUBS) {
         c = SES;
      }
      else if (c == BOF) { // resynchronise RX
         #asm
            BSF radmon_status, RS_FS
         #endasm
         rx_buffer_pt = 0; // reset buffer write pointer
         set_rx_state_normal();
         return;
      }
      else { // invalid escape sequence
         set_rx_state_wait();
         return; // exit function, nothing to process
      }
      set_rx_state_normal();
      break;
    default :
      return;
}
```

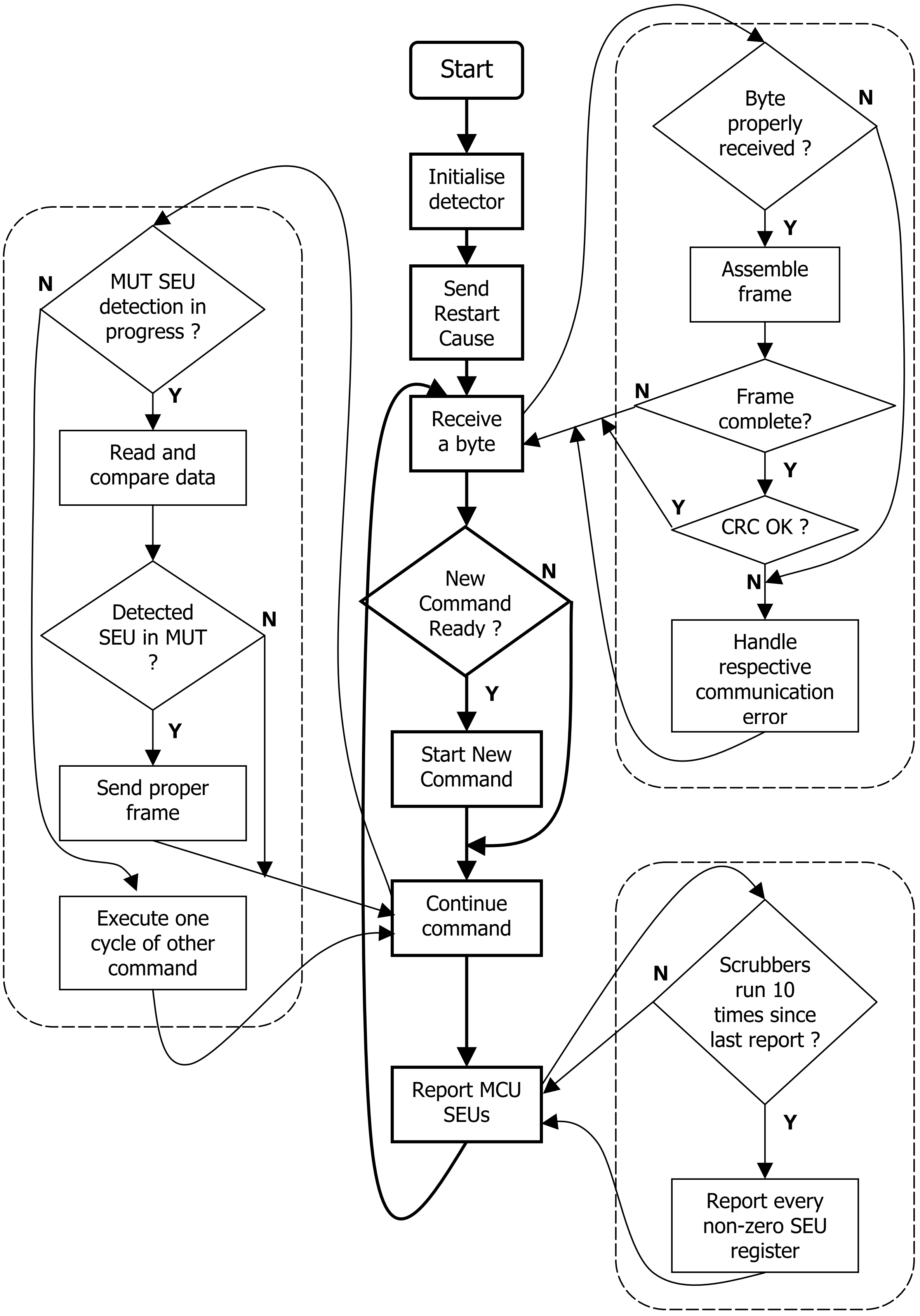


Figure 6.15. Flow diagram of SEU detector software. 'Y' stands for condition satisfied; 'N' stands for condition not satisfied.

6.4.3 Software for PC

The PC functions as a monitoring and command station. This functionality is implemented in software, which was written in C++, compiled using GCC and run on Slackware Linux. The software conducts an automated experiment. The experiment is aimed at registering SEUs occurring in the MUT. The information on such events is delivered by the SEU detector. The flow of the experiment is described in section 7 and depicted in Figure 7.5.

Every frame sent from the PC to SEU detector and every frame received from the SEU detector are archived in a common log file. Total number of SEUs detected in MUT and communicated to the PC is constantly updated. All SEUs, which occurred in the MCU and were reported, are handled in the same way. The software enables PC to communicate with detector in compliance with the RadMon protocol. The RadMon protocol is a frame-based one. The structure of a frame is depicted in Figure 6.16.

Beginning of Frame (BOF)	Frame length	Frame number	Frame type	Data block	CRC32 checksum
0x55	1 byte	1 byte	1 byte	1 – 253 bytes	4 bytes

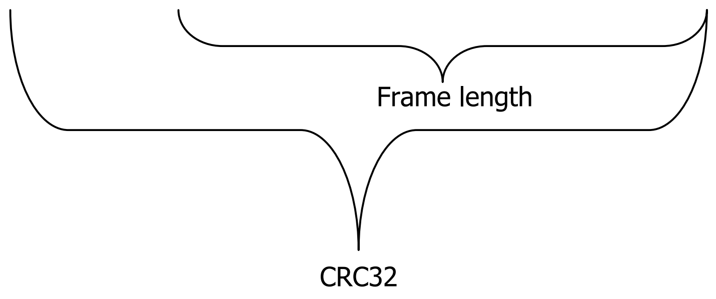


Figure 6.16. Frame of the RadMon protocol

There are two special characters defined in the protocol: BOF = 0x55, SES = 0xAA. If any byte within a frame, except the first BOF, is equal to 0x55 or 0xAA, then is it substituted by an escape sequence. Therefore, there are two escape sequences. Substitution of BOF yields 'SES 0x01', while substitution of SES gives 'SES 0x02'.

The frame types supported by PC software listed in Table 6.14.

Table 6.14. Frame types originating from PC

Frame type value	Frame type name	Description
0x01	PC_REG_DUMP	Request detector to send registers dump
0x02	PC_ECHO_REQ	Request detector to echo a message
0x07	PC_GEN_MUT_SEU	Request detector to generate a SEU in MUT at given address
0x0B	PC_MUT_DUMP	Request detector to send contents of MUT
0x0D	PC_BREAK_MUT_RW	Request detector to give up MUT reading or writing
0x0E	PC_MUT_WRITE_PATT	Request detector to write MUT with given pattern
0x0F	PC_DIAGN_REQ	Request detector to reply to diagnostic data
0x12	PC_SEND_VERSION	Request detector to send version information
0x1E	PC_RESTART_CAUSE	Request detector to report cause of last restart
0x1F	PC_SEU_REGS_REQ	Request detector to dump SEU registers

The PC does not acknowledge any message, only the detector is obliged to acknowledge a message, which does not yield an immediate result in the form of a frame with response. The Table 6.15 summarises frames, which are sent by the detector.

Table 6.15. Frames originating from the detector

Frame type value	Frame type name	Description
0x01	PIC_REG_DUMP	Dump of detector's registers
0x02	PIC_ECHO_REPLY	Reply to echo request
0x04	PIC_MUT_DUMP	Contents of MUT
0x05	PIC_DIAGN_DATA	Reply to diagnostic request
0x06	PIC_VERSION_REPLY	Version information
0x07	PIC_ACK_7	Acknowledgment to PC frame 0x07
0x0B	PIC_MUT_SEU	Report on SEU in MUT
0x0C	PIC_CORE_SEU	Report on SEU in MCU
0x13	PIC_ACK_13	Acknowledgment to PC frame 0x13
0x14	PIC_ACK_14	Acknowledgment to PC frame 0x14
0x1B	PIC_MUT_WRITTEN	Notification on MUT writing completion
0x1C	PIC_CMD_DENY	Command denied, not executable in current detector state
0x1D	PIC_CMD_NIMP	Command denied, since not implemented
0x1E	PIC_RST_CAUSE	Cause of last restart
0x1F	PIC_SEU_REGS_DUMP	Contents of SEU registers
0x20	PIC_FRERR	Notification on reception of a byte with framing error
0x21	PIC_RX_FULL	Notification on receiver FIFO filled up in 75%
0x22	PIC_CRC_BAD	Notification on reception of a frame with bad CRC32

Full source code of the software for PC is available on the attached CD-ROM. Every crucial block is documented with comments. The code has been partitioned into several files, for easier handling and clear indication of logical interdependencies.

The program runs in daemon mode, being unnoticeable for user. It takes control over one serial port and parallel port. There is no possibility to communicate with the software from outside, e.g. via TCP. Such functionality could be added in the future.

7 Experimental Results from Deutsches Elektronen-Synchrotron

The SRAM SEU detector was tested in DESY (Deutsches Elektronen-Synchrotron). The test was run from the 30th of August 2005 till the 2nd of September 2005, a total of 72 hours. The purpose of the test was to verify the designed MCU's behaviour under the mixed gamma and neutron irradiation. Once results were available, conclusions were drawn on the effectiveness of techniques improving radiation tolerance, which were implemented. The device was installed in the LINAC II linear lepton accelerator. The LINAC II was chosen as the test site, due to much higher neutron doses present there, than in the TTF or TTF2 sites. Testing the device in harsher conditions than those in which it is intended to operate, gives additional confidence margin if tests are successful. The LINAC II accelerator is used for filling electron (PETRA) and positron (DORIS) storage rings through an accumulating ring (PIA) at the DESY facility. There are two modes of operation of the accelerator, depending on the storage ring being filled. These are the electron mode and positron mode. In the electron mode, the electrons, generated in the electron gun, are accelerated and passed via PIA to the PETRA ring. In this mode the level of neutron radiation is low compared to that present in the other mode, as previous experiments with SRAM memories have shown. The mode desired for testing radiation tolerance of electronic devices is the positron mode, when DORIS is filled. The production of positrons starts from generating electrons, by the abovementioned electron gun. After being accelerated, the electrons are directed to hit tungsten target, in the form of a filament. The target is referred to as *electron-to-positron* converter, since as the result of the collision electrons are absorbed and positrons are produced. This process is accompanied by generation of high number of neutrons. The neutrons are moderated by surrounding obstacles and reach electronic devices as *thermal neutrons* [42]. These neutrons, as described in section 3.1 have the detrimental effect on the devices under test. The neutron energy spectrum in LINAC II tunnel is shown in Figure 7.1.

The SEU detector was placed in about five meters distance from the converter. About two meters farther, area is restricted due to high radiation doses, despite the shield. The experimental set-up in the accelerator is shown in Figure 7.3. In the place the device was installed radiation is so intensive that after the test, the whole device was too radioactive to be transported back to Poland. It emitted gamma radiation at the rate of 156 µSv/h, while other, non-activated objects emit gamma at about 0.1 µSv/h or less. The activation is, short-term. Mainly copper is activated. The Cu isotope half-life is very short, ca. 20 minutes. After a cool down period of ca. one week, for that level of activation, transporting is possible.

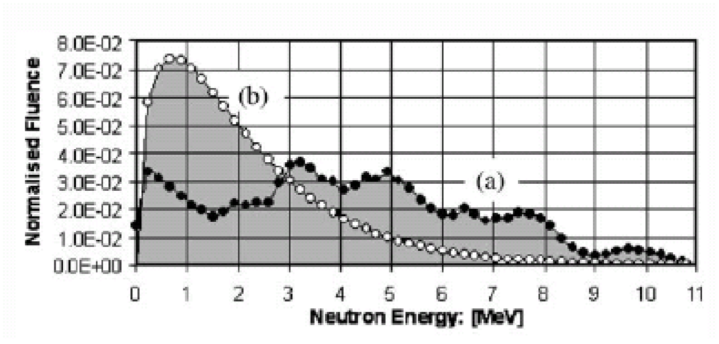


Figure 7.1. Energy spectrum of neutrons in LINAC II tunnel (b)

The monitoring station (PC computer running Linux operating system) together with transceiver was placed on the test bench. The test bench was located in radiation-free environment, namely the accelerator's service hall, see Figure 7.4. The RESET and POWER-OFF signals from PC were fed through parallel-port cable to the transceiver, and next from the transceiver to the SEU detector over 20-wire ribbon cable. In the same manner power and ground were delivered from the power supply to the tested device. The communication between PC and detector was carried over 50 meters long full-duplex 62.5/125 micron multimode optical fibre. The bitrate was set to 9600 bps. The optical fibre was installed in the accelerator three months before the test (in May 2005). During the three months time the fibre was not used and subject to mixed gamma and neutron irradiation. This allowed for a qualitative evaluation of radiation influence on transmission reliability over optical fibre. As stated further, no communication errors were observed.

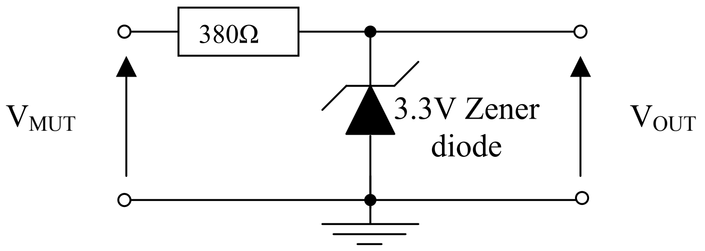


Figure 7.2. Interfacing 5 V signals to the detector

The tested memory (MUT – Memory Under Test) was a standard RadMon memory module with 2 x 512 kB SRAM. According to the datasheet [43] the K6T4008C1B-GB70 SRAM is powered from 5 V. The detector is configured to be compatible with 3.3 V logic levels, and its inputs are not tolerant to 5 V signals. Therefore a voltage limiting circuit was added on every output pin of the memory, to limit the 5 V levels to 3.3 V levels. The circuit, shown on Figure 7.2, follows Actel's recommendation on interfacing ProAsic Plus devices to 5 V logic [44]. No modifications were necessary for the signal levels input to the memory, as it is input-compatible with 3.3 V logic.

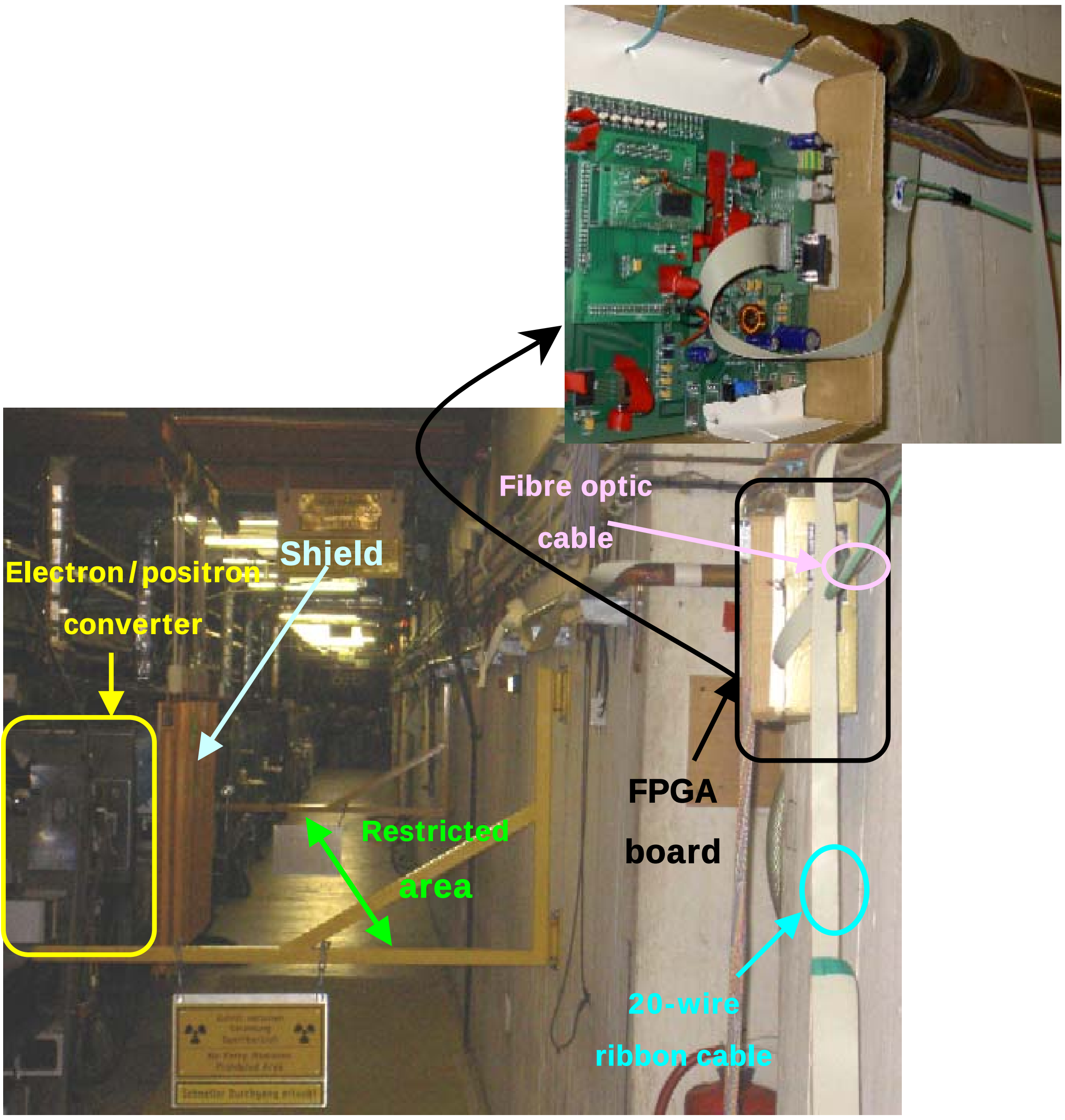


Figure 7.3. The experimental set-up inside the LINAC II accelerator

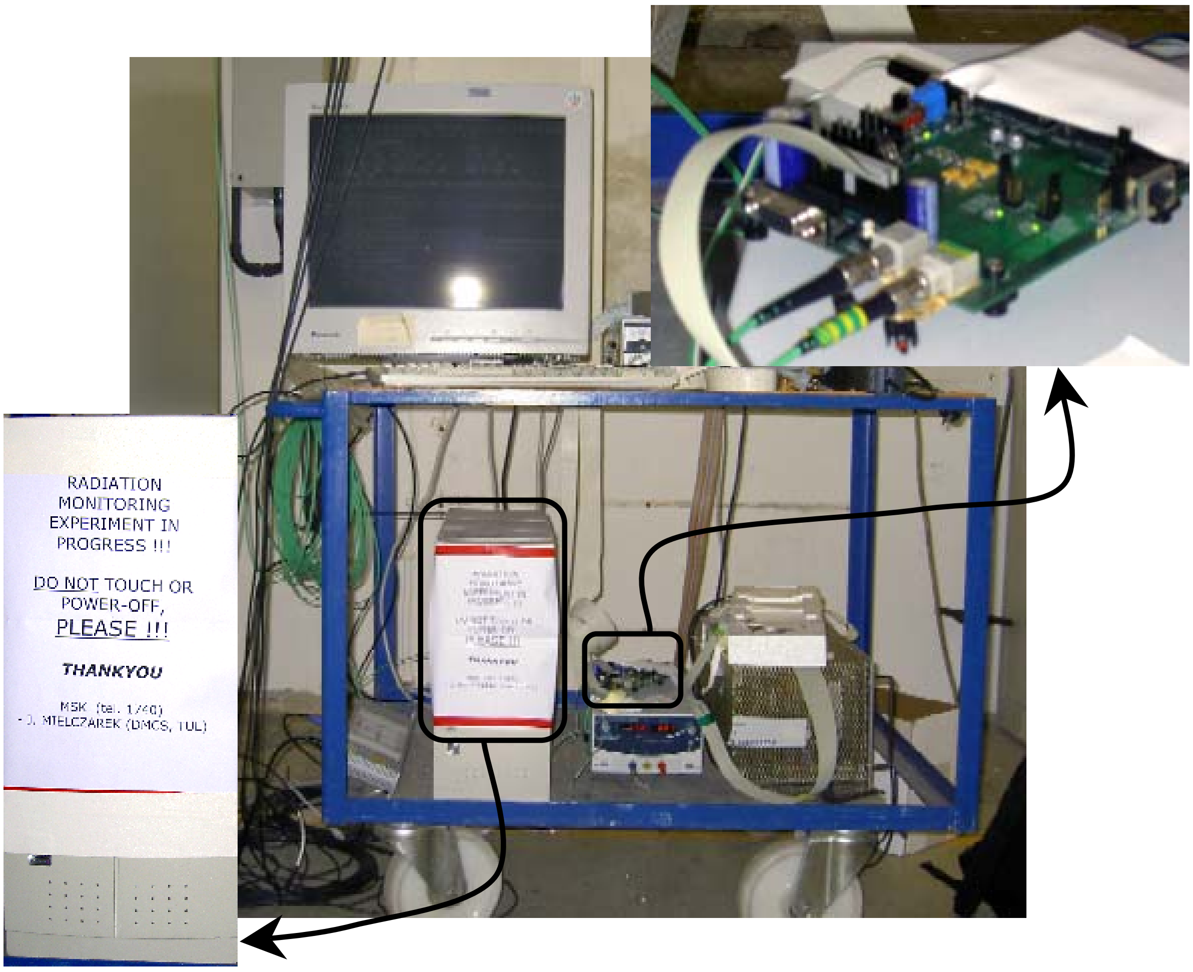

RADIATION MONITORING EXPERIMENT IN PROGRESS !!!
DO NOT TOUCH OR POWER-OFF, PLEASE !!!
THANKYOU
MSK (tel. 1740)
- J. MIELCZAREK (DMCS, TUL)

Figure 7.4. The monitoring station (a PC running Linux OS) with transceiver connected, in the accelerator hall

Two types of tests were run, designated Experiment I and Experiment II. During both, the detector was performing its main function – detecting SEUs in the MUT. Experiment I followed the scenario depicted on Figure 7.5. After resetting the detector, the PC listened for a frame carrying information on the cause of detector's restart. If this was not properly received, the detector was restarted. Due to an unidentified problem with restart, the procedure repeated several times, before synchronisation was reached. After it succeeded, the PC requested dump of detector's registers and MCU's SEU registers. Every request and following requests were attempted three times, before decision on a failure was made. No additional attempts were taken once answer from the detector was correct. Every failure followed by detector restart. After reporting its status, the detector was commanded to write the MUT with 0xFF pattern. Once acknowledgment for the command was received, a timeout period of 120 seconds started. If it expired and no information on MUT writing completion was received, the detector was restarted.

After the detector has successfully written the MUT (reported by sending a proper frame), the monitoring phase started. In this phase the PC recorded all information coming from the detector and reacted appropriately to SEU reports. Both MUT and MCU SEUs were tracked and their numbers recorded. Every five minutes periodic diagnostics were executed. The diagnostics comprised dumping detector's registers, dumping SEU registers and generating one SEU in MUT at address calculated from current time. The number of generated SEUs was recorded. If the diagnostics did not pass, the detector was restarted. If during any phase of the experiment the restart cause frame was received again, the cause was recorded (either due to a glitch on RESET or POWER-OFF line, or due to a double error in any of MCU's SRAM components) and the detector restarted. This is not shown in Figure 7.5. The summary of Experiment I is presented in Table 7.1.

Table 7.1. The summary of main events during Experiment I

Time	Event	Comment
2005-08-30 12:11:41	Experiment Started	The beginning of experiment
2005-08-30 14:03:55	Detector Restart	Caused by "bug", which was fixed later on.
2005-08-30 14:20:48	Detector Restart	As above
2005-08-30 14:47:55	Detector Restart	As above
2005-08-30 15:39:53	Detector Restart	As above
2005-08-30 18:07:42	Experiment Finished	Monitoring daemon killed. Debugging code added.
2005-08-30 18:08:05	Experiment Started	Monitoring daemon restarted.
2005-08-30 19:45:08	Detector Restart	Caused by "bug" – confirmed by added debugging code.
2005-08-30 21:31:57	Detector Restart	As above
2005-08-30 21:58:40	Detector Restart	As above
2005-08-30 21:58:54	Detector Restart	Problem with previous restart.
2005-08-31 00:49:32	Experiment Finished	Monitoring daemon killed. Analysis of collected debugging data. "Bug" fixed.
2005-08-31 00:50:50	Experiment Started	Monitoring daemon restarted after fixing "bug".
2005-08-31 20:35:31	Experiment Finished	Experiment I finished. Total detector restarts: 8 Detected MUT SEUs due to neutrons: 335 Detected MCU SEUs due to neutrons: 0

The flow of Experiment II was modified. To increase the level at which detector's registers were utilised, put more load on stack and UART, a diagnostic frame was sent every second, during the monitoring phase. If the answer was incorrect, the detector was restarted. Putting more load on the detector slowed down the rate at which MUT was swept, however without noticeable effects to the experiment. The increased load allowed for better evaluation of communication reliability and channel BER (Bit Error Rate). It also increased the chance to observe SEUs in the detector itself as more of its resources were used. Summary of events for Experiment II is presented in Table 7.2.

Table 7.2. Summary of main events during Experiment II

Time	**Event**	**Comment**
2005-08-31 20:42:22	Experiment Started	The beginning of Experiment II
2005-08-31 20:47:30	Experiment Finished	Monitoring daemon killed. Spotted "bug" in the added diagnostic routine.
2005-08-31 20:47:49	Experiment Started	Monitoring daemon restarted. Fixing "bug".
2005-08-31 20:12:44	Experiment Finished	Monitoring daemon killed. Test complete. Bug fixed.
2005-08-31 21:13:09	Experiment Started	Monitoring daemon restarted. Normal operation mode.
2005-09-02 12:17:31	Experiment Finished	Experiment II finished. Total detector restarts: 1 Detected MUT SEUs due to neutrons: 208 Detected MCU SEUs due to neutrons: 5

Appendix C contains listing of SEUs registered in the MUT during both experiments. Full track of the experiment is available on attached CD-ROM. The PC used for the experiments had incorrect time settings. To obtain the real time from log file, one must add 40 minutes to the logged time. Table 7.3 summarizes the results of both experiments.

Table 7.3. Summary of Experiment I and Experiment II

Day	**MUT SEUs**	**MCU SEUs**	**Restarts**	
			PC performed	**Double Error**
2005-08-30	101	0	8	0
2005-08-31	238	2	3	0
2005-09-01	120	2	0	0
2005-09-02	56	1	0	0

Only single errors were detected in the pairs of cells. There was no such situation that both cells making up the pair were in error. There is a non-zero probability that the two cells had exactly the same error, and such a situation would not be considered an SEU by the detector. However, for the sake of extremely low value of that probability, it can be assumed, that such situation did not take place. There were no multiple errors, in the sense, that only one cell was affected, but more than one bit was distorted. The distribution of SEU was approximately equal for both modules making up the MUT. The module located on the bottom side of PCB exhibited 226 SEUs in total, whereas the one on the top side - 289. Other speaking, 56% of the total number of detected SEUs was located in the top module, 43% in the bottom one.

The four restarts on 30th of August, during Experiment I, were not caused by detector instability. Their origin was a previously unrecognised logic error in PC software, particularly the receiver state machine. Whenever the detector sent a frame, which contained an escape sequence, the PC would treat the sequence as erroneous and discard the frame. In this manner acknowledgments containing any escape sequence, coming from the detector were not accepted. After third unsuccessful attempt to send a command again and receive acknowledgment, the detector was restarted. The problem was solved by correcting the code for the receiver state machine between 23:50 the 30th and 00:10 the 31st. From that time no restarts were observed during Experiment I.

The Experiment II was started on the 31st, at about 21:00. In total three restarts of the detector were necessary, due to an error in the newly run diagnostics on the PC side. The error was immediately corrected. From that moment the detector operated perfectly without interruptions.

The exchange of information between the detector and the PC proceeded without errors. There were no framing errors in detector's UART, no frames were received incorrectly on either side, for all of them the CRC check always passed. The amount of exchanged data allows estimating the BER (Bit Error Rate) of the communication channel. This estimation is based on the total number of transmitted bits (start and stop bits of the used EIA-232 transport protocol are included, as vital for proper communication).

The BER achieved during the experiment is zero. The estimated BER of the channel is calculated as the reciprocal of the total number of the bits, since none of them was transmitted in error – no communication errors occurred.

$$BER < 6.12 * 10^{-8}$$

The obtained value of BER is rather large as based on relatively small amount of exchanged raw data - only ca. 1.5 MB. A longer lasting experiment would allow evaluating the parameter more precisely.

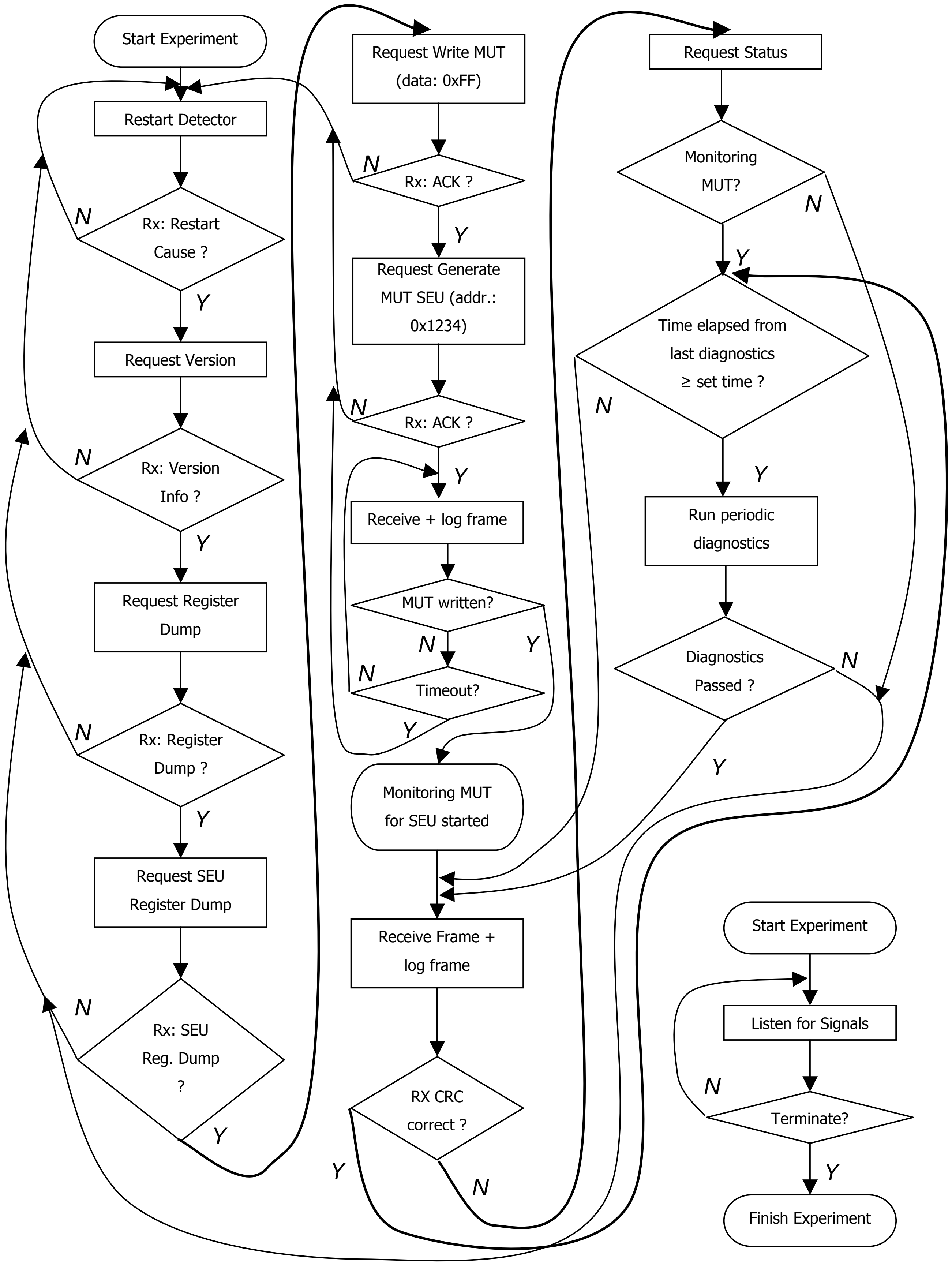


Figure 7.5. The flow of experiment conducted in LINAC II. 'N' stands for condition not satisfied; 'Y' stands for condition satisfied.

The SEU, which were spotted in the MCU, require a separate comment. If one browses the log file from the experiments, will notice, that a significantly greater amount of SEU in the MCU was reported. These SEU were noticed in the OTHER and UART_RX SEU registers. These registers hold the number of SEUs detected in a group of non-critical components of the MCU and the UART's receiver, respectively. As mentioned in section 6.2.2.2 the SEU indicating circuit is not glitch-free. Therefore, the glitches may cause "false alarms", which was confirmed during tests in the laboratory. These two groups of components were the only ones burdened with the influence of glitches. No other component exhibited the burden. Therefore, all the SEU reports regarding the OTHER and UART_RX registers were ignored, as not being confident. However, five SEU were reported to have occurred in the UART transmitter component, as detailed in Table 7.4.

Table 7.4. SEU in MCU's UART transmitter

Date	Time	Component Group
2005-08-31	22:54:19	UART TX
2005-08-31	23:16:53	UART TX
2005-09-01	01:50:47	UART TX
2005-09-01	12:53:54	UART TX
2005-09-02	03:38:08	UART TX

The following facts support the position that these SEU were caused by neutrons:

- No "false alarms" were reported for UART transmitter component in the laboratory,
- Strong correlation of time instants at which the SEU were reported with reports concerning SEU in MUT, moreover the MCU SEUs occurred at instants of higher neutron number, as indicated by denser number SEUs in MUT,
- No time periodicity of the SEU (there was some in case of the earlier mentioned "problematic" components"),
- Only five occurred – the "problematic" components reported a total of over 4000 SEUs.

If neutrons caused the SEU, then the Triple Modular Redundancy coped with them, sustaining correct operation of the transmitter. This is further proved by the fact that no communication errors occurred during the experiments, particularly no such errors occurred in the time vicinity of the discussed SEU reports. The fact that SEU occurred in user flip-flops of the FPGA is also of great importance, as it justifies the need to protect designs implemented in FPGA devices against these effects. There were no reports on SEUs in the embedded SRAM components of the FPGA. This can be explained by several facts. The size of used embedded SRAM was only 297 bytes (256 for stack, 41 for register file). Taking into account that packing density of the SRAM is low (total of 14 kB for the whole FPGA) and that it is fabricated in 0.22 μm CMOS technology further justifies lack of registered SEUs. A longer lasting experiment would definitely yield SEUs in the embedded SRAM. There were no SEUs in the ECC program memory as well. The EDAC circuitry of the chip has corrected all single-bit errors, if any occurred. There are no means to verify that, as EDAC is transparent to the device, which uses the memory.

The best way to show correlation between SEUs detected in the MUT and the activity of LINAC II is to use cumulative plot. Obtained experimental data has been post-processed to yield such a plot, presented in Figure 7.6. Additionally, the SEUs detected in the MUT and MCU during the experiments are presented against LINAC II activity. In types of plots the activity is determined by the value of PIA current. During highest activity levels of LINAC II, the PIA current attains, or slightly overshoots 25 mA. The plots in Figure 7.7, Figure 7.8, Figure 7.9 and Figure 7.10 show SEUs for each day of experiments.

During the first experiment an interesting coincidence took place. On the 30th an SEU was detected in the same pair of memory cells twice, with ca. 5 hours time separation. The relevant events are extracted in Table 7.5.

Table 7.5. SEU in the same pair of MUT cells

Date	**Time**	**Data Low**	**Data High**	**Address Low**	**Address High**
2005-08-30	16:09:06	DF	FF	F72F4C	DF2F4C
2005-08-30	21:51:41	FE	FF	F72F4C	DF2F4C

The probability of such an event is ca. 1E-6.

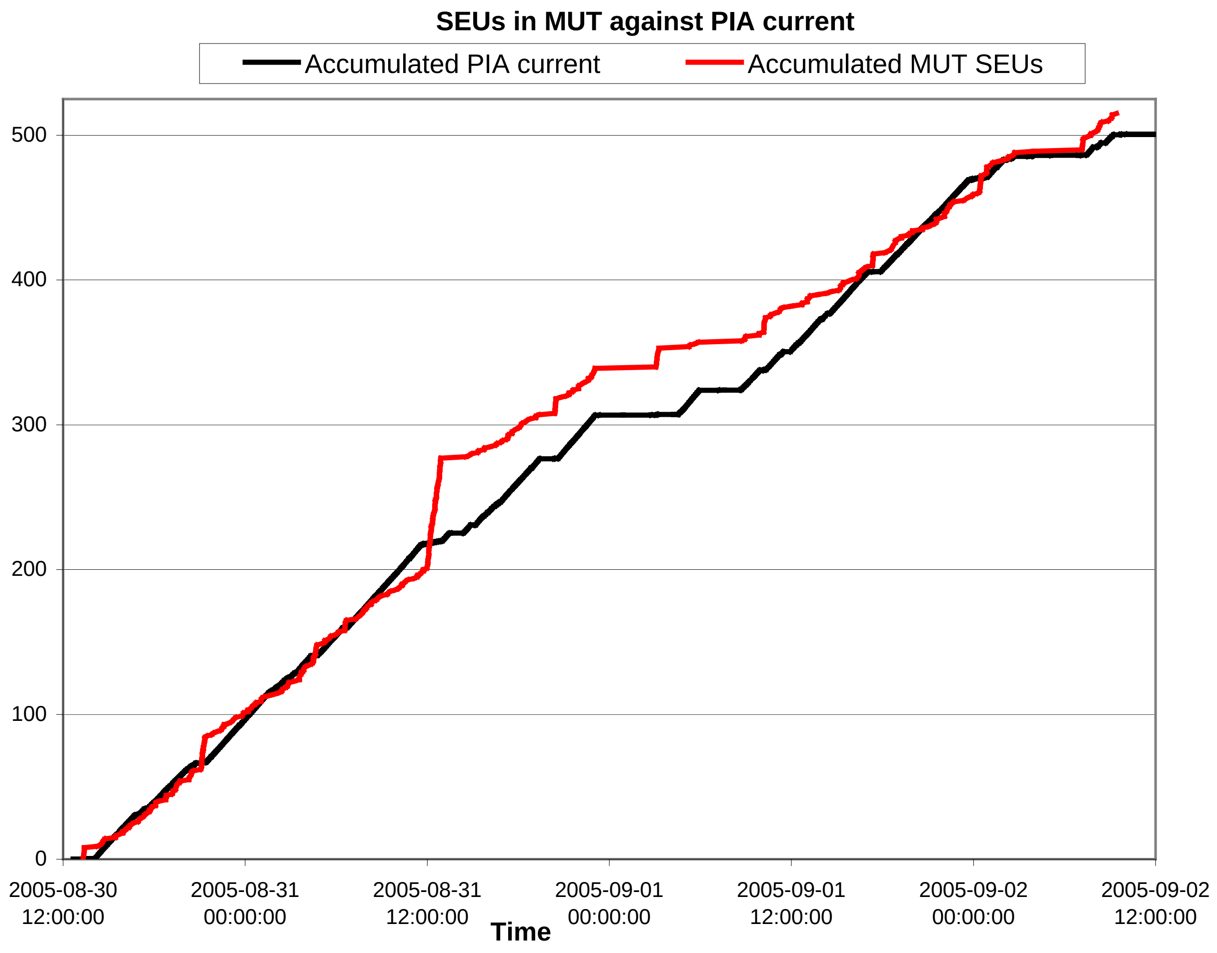


Figure 7.6. Accumulated Single Event Upsets in Memory Under Test against accumulated activity of LINAC II qualified in terms of accumulated PIA current. The value of PIA current in [mA] has been divided by 125 to scale it down for better observability of PIA – SEU correlation.

The analysis of cumulative plot of Figure 7.6 yields interesting observations. First of all, it proves that the detector functioned correctly. During the periods while the LINAC II was not operating, the number of MUT SEUs does not rise, remains flat. The second observation is the existence of two strong slopes of the accumulated MUT SEUs. The same can be stated about the accumulated PIA current. The greater slope of MUT SEUs correlates with the smaller one of PIA current and vice versa. The greater slope of PIA current clearly indicates, that during the time PIA current was high. However, during that time SEUs in MUT were less frequent than for lower values of PIA, corresponding to the smaller PIA slope. The lower values of PIA current correspond to the positron, i.e. DORIS mode of the accelerator. This mode is characterised by much higher level of neutron radiation than the electron, i.e. PETRA mode, and PIA current smaller than 10 mA. This is clearly confirmed by the frequently occurring SEUs in the tested SRAM.

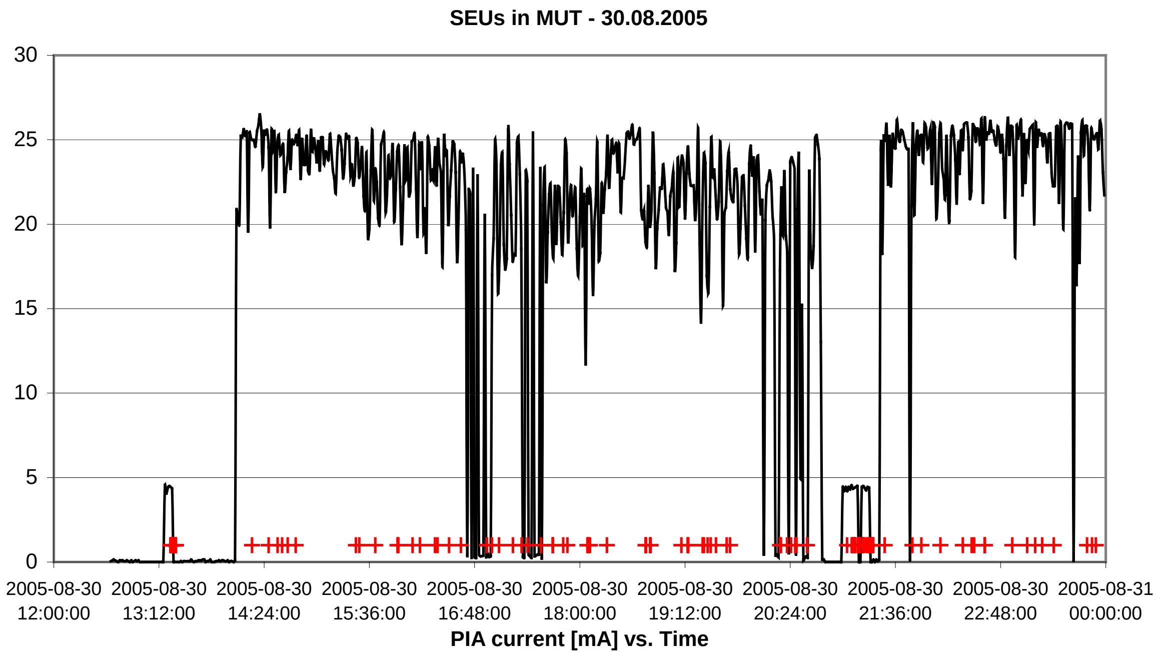


Figure 7.7. Single Event Upsets detected in the Memory Under Test against activity of LINAC II, qualified in terms of PIA current on the 30th of August.
The time instants, at which the detector reported SEUs in MUT are marked as red crosses.

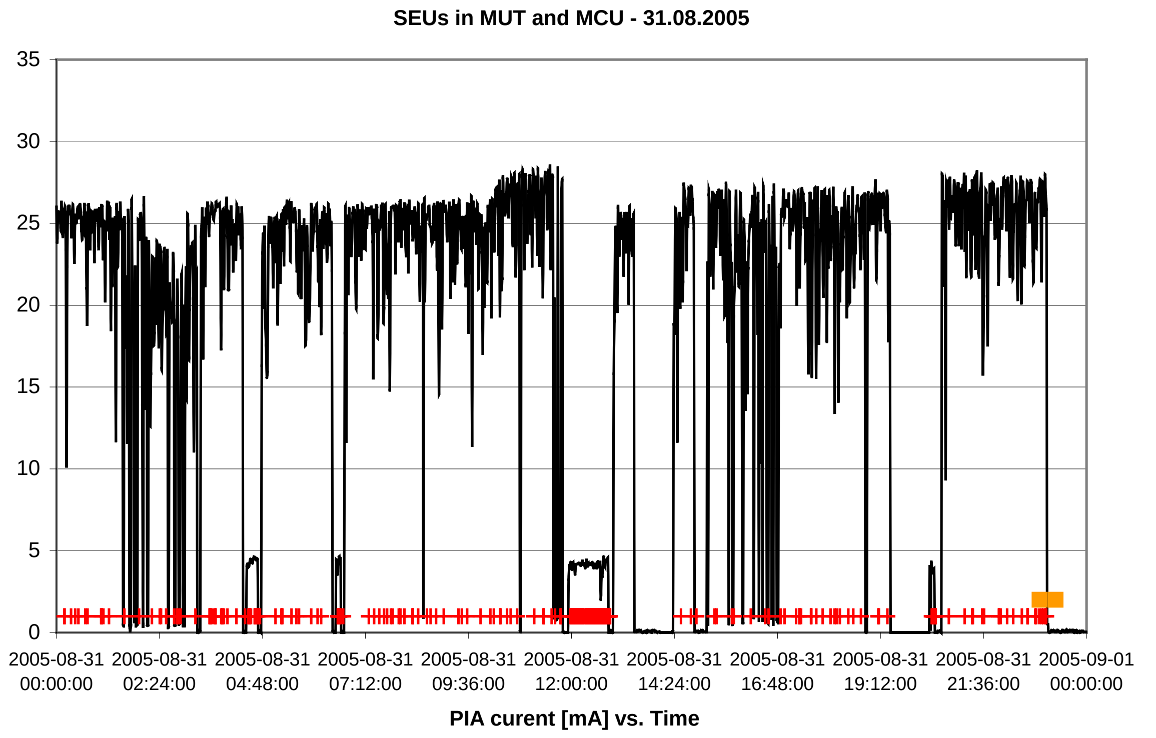


Figure 7.8. Single Event Upsets detected in the Memory Under Test against activity of LINAC II, qualified in terms of PIA current on the 31st of August.
The time instants, at which the detector reported SEUs in MUT are marked as red crosses.
Two SEUs in the detector's MCU were detected – marked as orange squares.

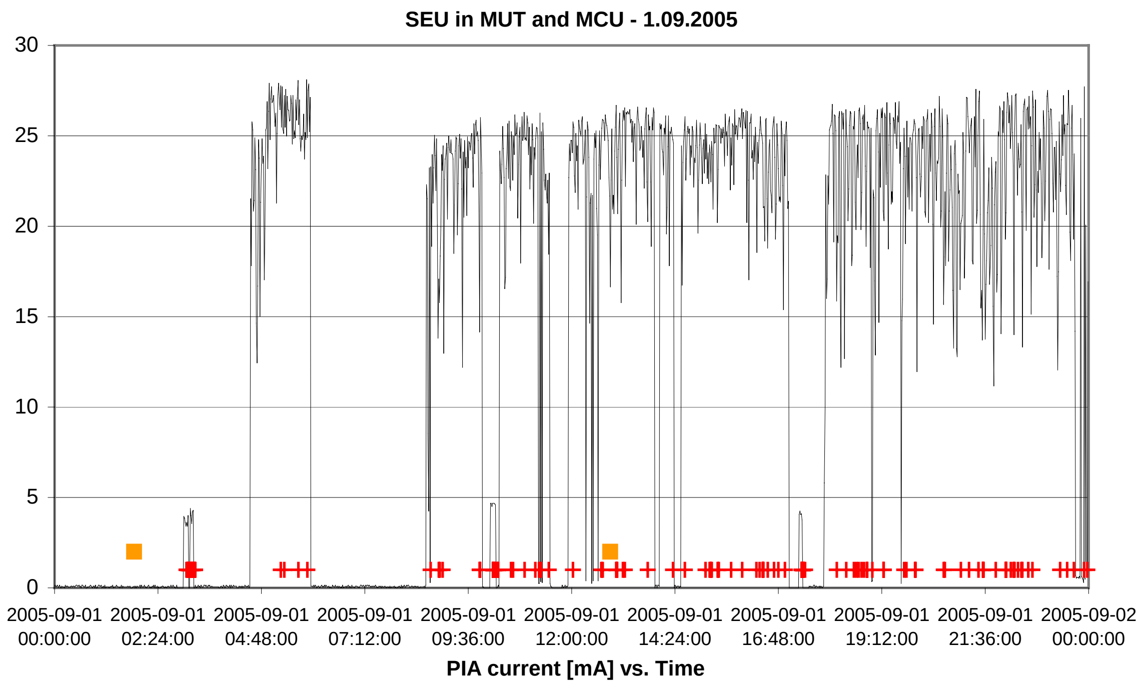


Figure 7.9. Single Event Upsets detected in the Memory Under Test against activity of LINAC II, qualified in terms of PIA current on the 1st of September.
The time instants, at which the detector reported SEUs in MUT, are marked as red stars.
Two SEUs in the detector's MCU were detected – marked as orange squares.

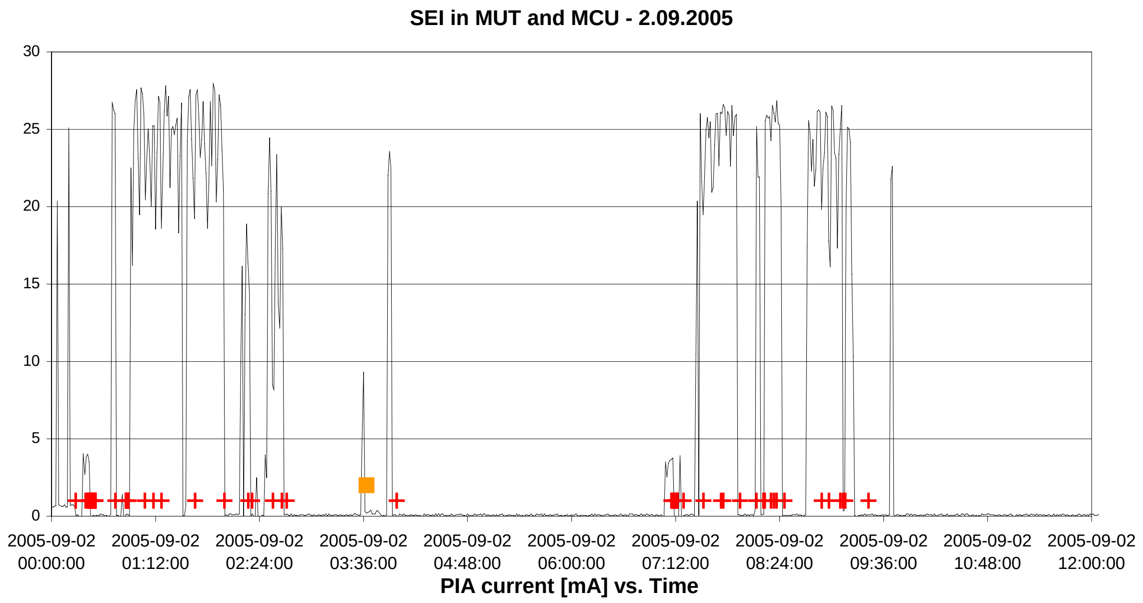


Figure 7.10. Single Event Upsets detected in the Memory Under Test against activity of LINAC II, qualified in terms of PIA current on the 1st of September.
The time instants, at which the detector reported SEUs in MUT, are marked as red stars.
One SEU in the detector's MCU was detected – marked as orange square.

8 Conclusions

The objective of this project was to design and physically implement a radiation tolerant transmission channel circuit. The device was required to enable reliable communication between a device under test and a central system. The radiation tolerant circuit ought to have a degree of autonomy, flexible hardware configuration and capability of being conveniently interfaced to various types of devices under test. The objective has been accomplished. All the requirements for the radiation tolerant transmission channel circuit have been met, particularly tolerance to radiation. The circuit operated without failures for 72 hours under mixed gamma and neutron irradiation. The only breaks in operation were either due to software problems on PC or change in experiments. Various techniques were employed for mitigating the effects of SEUs, namely Hamming Codes backed with Scrubbing and the well-established Triple Modular Redundancy. The TMR has presumably sustained system's operation five times, during the tests in LINAC II, DESY. There was no situation, when Hamming Codes had the opportunity to mitigate an effect of SEU. Tests of longer duration are necessary to thoroughly verify the system and the mitigation techniques.

Communication between the radiation tolerant device and PC proceeded without errors, was reliable. It was proved, that it is possible to apply well-established SEU mitigation techniques on the level of HDL. The developed semi-automatic mitigation technique employing the well-known TMR scheme was successfully used throughout the project. Its main advantage is the independence of VHDL source code. No modifications need to be introduced to the source code of hardware description, except for one special case of high-impedance. This enables to design radiation tolerant digital systems in exactly the same manner as common digital systems. The tested design is post-processed on the VHDL netlist level and its radiation tolerance improved. This technique also enables to mitigate already existing designs, which were not originally meant to be radiation tolerant. The current limitation of the technique is its support for Actel ProAsic Plus FPGAs only. It might be desirable to add support for other FPGA device families.

The hardware platform was based on COTS components only, none of the used electronic components were hardened against radiation. This renders the involved costs a fraction of a price of available radiation tolerant or hardened programmable devices. However, the designed system has been mitigated to SEUs only, which might prove to be insufficient during long-term tests. On the other hand, VHDL or other HDL relies on a fixed hardware technology, which is the only level, on which the full suite of SEE mitigation and hardening mechanisms can be adopted.

The mitigating capabilities of adopted techniques should be in future quantified in terms of specific radiation doses, which the mitigated circuit is able to withstand.

The drawback of employed techniques is compromised performance of the system. Particularly the maximum attainable frequency is decreased when the TMR scheme or Hamming Codes are applied. The effect of Scrubbing is periodical pauses in system's operation, hence decreased computational capability. The effects caused by TMR and Hamming Codes could be potentially reduced if dedicated floorplanning of the FPGA resources was involved, but generally the speed penalty is unavoidable if these techniques are employed.

References

[1] G. C. Messenger, M. S. Ash, "The Effects of Radiation on Electronic Systems", Van Nostrand Reinhold Company, 1986

[2] B. Mukherjee, D. Makowski, D. Rybka. M. Grecki, S. Simrock, "Interpretation of the Single Event Upset in Static Random Access Memory chips induced by low energy neutrons", *12th International MIXDES Conference*, 22-25 June 2005

[3] S. M. Sze, "Semiconductor Devices", John Wiley & Sons, 2002

[4] D. Makowski, M. Grecki, B. Mukherjee, B. Świercz, S. Simrock, "Radiation Tolerant System for Neutrons Fluence Measurement", April 2005

[5] M. Gastal, P. Moreira, "Radiation Qualification of Commercial Off-The-Shelf P-I-N Receivers for the TTC System", *9th Workshop on Electronics for LHC Experiments*, September/October 2003

[6] F. Berghmans, M. Van Uffelen, A. Nowodzinski, A. Fernandez Fernandez, B. Brichard, A. Gusarov, M. Decréton, "Radiation Effects in Optical Communication Devices", *Proceedings Symposium IEEE/LEOS Benelux Chapter*, 2000

[7] M. Ott, S. Macmurphy, M. Dodson, "Radiation Testing of Commercial off the Shelf 62.5/125/250 Micron Optical Fibre for Space Flight Environments", Sigma Research and Engineering, NASA Goddard Space Flight Centre

[8] "All is Not SRAM - A survey of flash, antifuse, and EE programmable logic", http://www.fpgajournal.com/articles/sram.htm

[9] "Radiation Results of the SER Test of Actel, Xilinx and Altera FPGA instances", iRoC, October 2004

[10] F. G. de Lima Kastensmidt, G. Neuberger, R. F. Hentschke, L. Carro, R. Reis, "Designing Fault-Tolerant Techniques for SRAM-Based FPGAs," *IEEE Design and Test of Computers*, vol. 21, no. 6, pp. 552-562, November/December 2004

[11] Single Event Upset, Altera, http://www.altera.com/products/devices/stratix/features/stx-seu.html

[12] Actel website, http://www.actel.com

[13] Actel's devices datasheets, http://www.actel.com/techdocs/ds/default.aspx

[14] "APA750 and A54SX32A LANSCE Neutron Test Report", ACTEL, December 2003

[15] Instant-On FPGA Solutions: ispXPGA, Lattice Semiconductor Corporation, http://www.latticesemi.com/products/fpga/xpga/index.cfm

[16] Aerospace and Defence Applications, Xilinx, http://www.xilinx.com/products/silicon_solutions/market_specific_devices/aero_def/capabilities/aero_def_app.htm

[17] Aerospace & HiRel devices datasheets, Actel, http://www.actel.com/products/aero/ds.aspx

[18] J. J. Wang, "Radiation Effects in Field Programmable Gate Arrays", *9th Workshop on Electronics for LHC Experiments*, September/October 2003

[19] FlashLock: Security in Actel Flash FPGAs, Actel, http://www.actel.com/products/rescenter/security/solutions/flash.aspx

[20] ProAsic Plus Datasheet, Actel, http://www.actel.com/documents/ProASICPlus_DS.pdf

[21] RTSX-S RadTolerant FPGAs Datasheet, Actel, http://www.actel.com/products/aero/ds.aspx

[22] S. Moschoyiannis, "Group Theory & Error Detecting/Correcting Codes Technical Report", University of Surrey, December 2001

[23] Microchip website, http://www.microchip.com

[24] PIC16C5x Datasheet, Microchip

[25] C. Peacock, "Interfacing the Serial/RS-232 Port", January 1998, http://www.senet.com.au/~cpeacock

[26] OpenCores, http://www.opencores.org/

[27] Weidong Lu, "Designing TCP/IP Functions In FPGA", TU Delft, 2003, http://ce.et.tudelft.nl

[28] ProAsic and ProAsic Plus Macro Library Guide, Actel, 2004

[29] Am29LV040B Datasheet, AMD, http://www.amd.com

[30] R1LV1616H-I 16M SRAM Datasheet, Renesas, http://www.renesas.com

[31] Agilent HFBR-0400, HFBR-14xx and HFBR-24xx Series Low Cost, Miniature Fiber Optic Components with ST®, SMA, SC and FC Ports Datasheet, Agilent, http://www.agilent.com

[32] "Inexpensive dc to 32 MBd Fiber-Optic Solutions for Industrial, Medical, Telecom, and Proprietary Data Communication Applications", Agilent's Application Note 1121, Agilent, http://www.agilent.com

[33] "Generic Printed Circuit Layout Rules for Agilent's Low-Cost Fiber-Optic Components", Agilent's Application Note 1137, Agilent, http://www.agilent.com

[34] SN54F00, SN74F00 Quadruple 2-Input Positive-NAND Gates Datasheet, Texas Instruments, http://www.ti.com

[35] SMD beads EMI suppression products Produktinformation, ELFA, http://www.elfa.se

[36] Low Power Slew-Rate-Limited RS-485/RS-422 Transceivers Datasheet, Maxim, http://www.maxim-ic.com

[37] MAX232 Dual EIA-232 Driver/Receiver, Texas Instruments, http://www.ti.com

[38] CFPS-72, -73 Produktinformation, ELFA, http://www.elfa.se

[39] LM1575/LM2575/LM2575HV Simple Switcher® 1A Step-Down Voltage Regulator Datasheet, National Semiconductor, http://www.national.com

[40] LM78XX Series Voltage Regulators Datasheet, National Semiconductor, http://www.national.com

[41] CCSC Compiler home website, http://www.ccsc.com

[42] B. Mukherjee, D. Makowski, S. Simrock, "Dosimetry of high-energy electron linac produced photoneutrons and the bremsstrahlung gamma-rays using TLD-500 and TLD-700 dosimeter pairs", *NIMA Conference*, 2005

[43] "512Kx8 bit Low Power CMOS Static RAM" - K6T4008C1B-GB70 SRAM Datasheet, Samsung

[44] "Interfacing ProAsic Plus FPGAs with 5V Input Signals", Application Note, Actel, March 2004

Appendix A Schematics and PCB Layouts

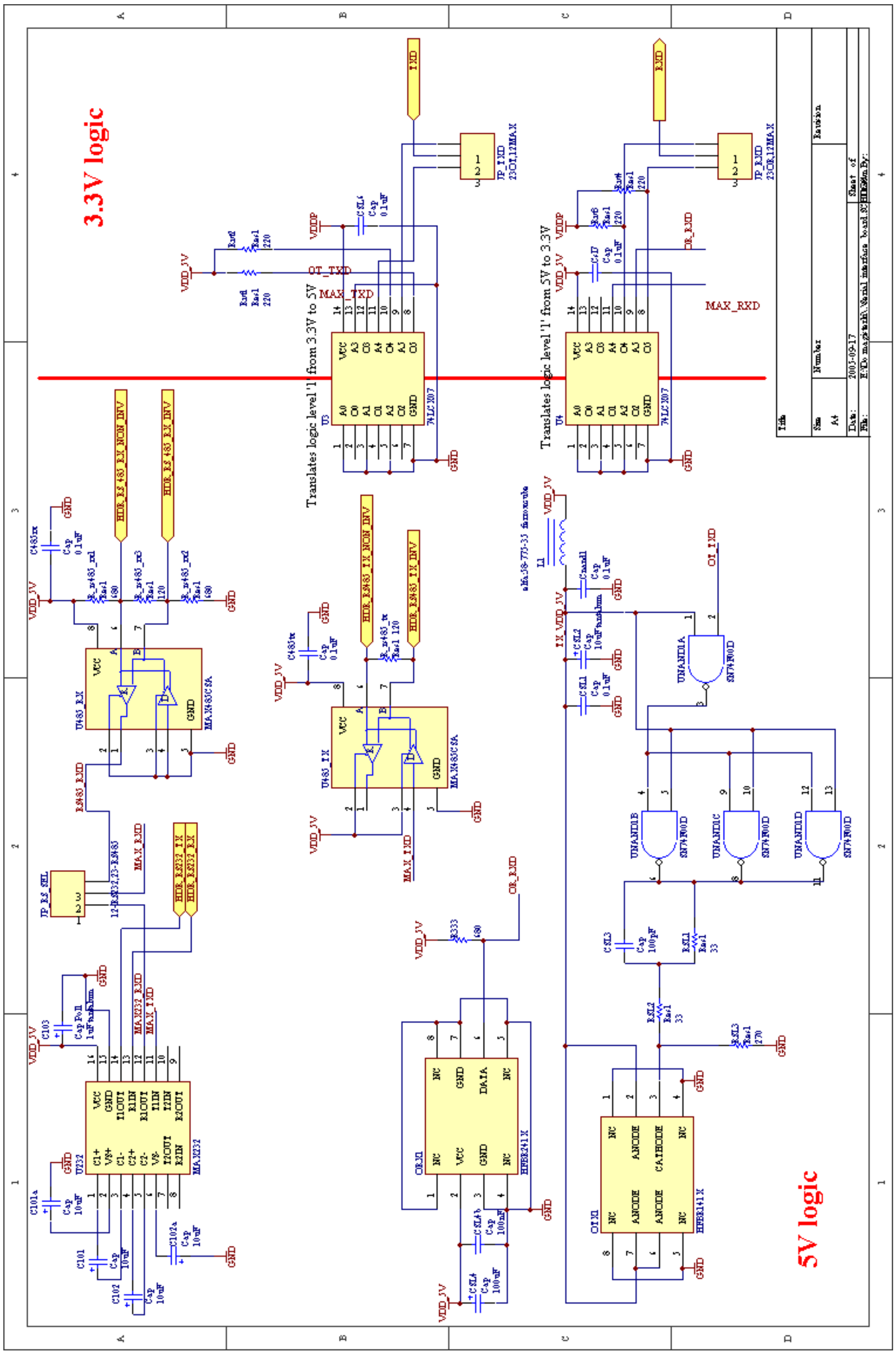

3.3V logic
5V logic
Translates logic level '1' from 3.3V to 5V
Translates logic level '1' from 5V to 3.3V
TXD
RXD
74LC3207
MAX483CSA
MAX3232
HFBR2412
HFBR1412
SN74F00D

Figure A.1. FPGA board – serial interfaces

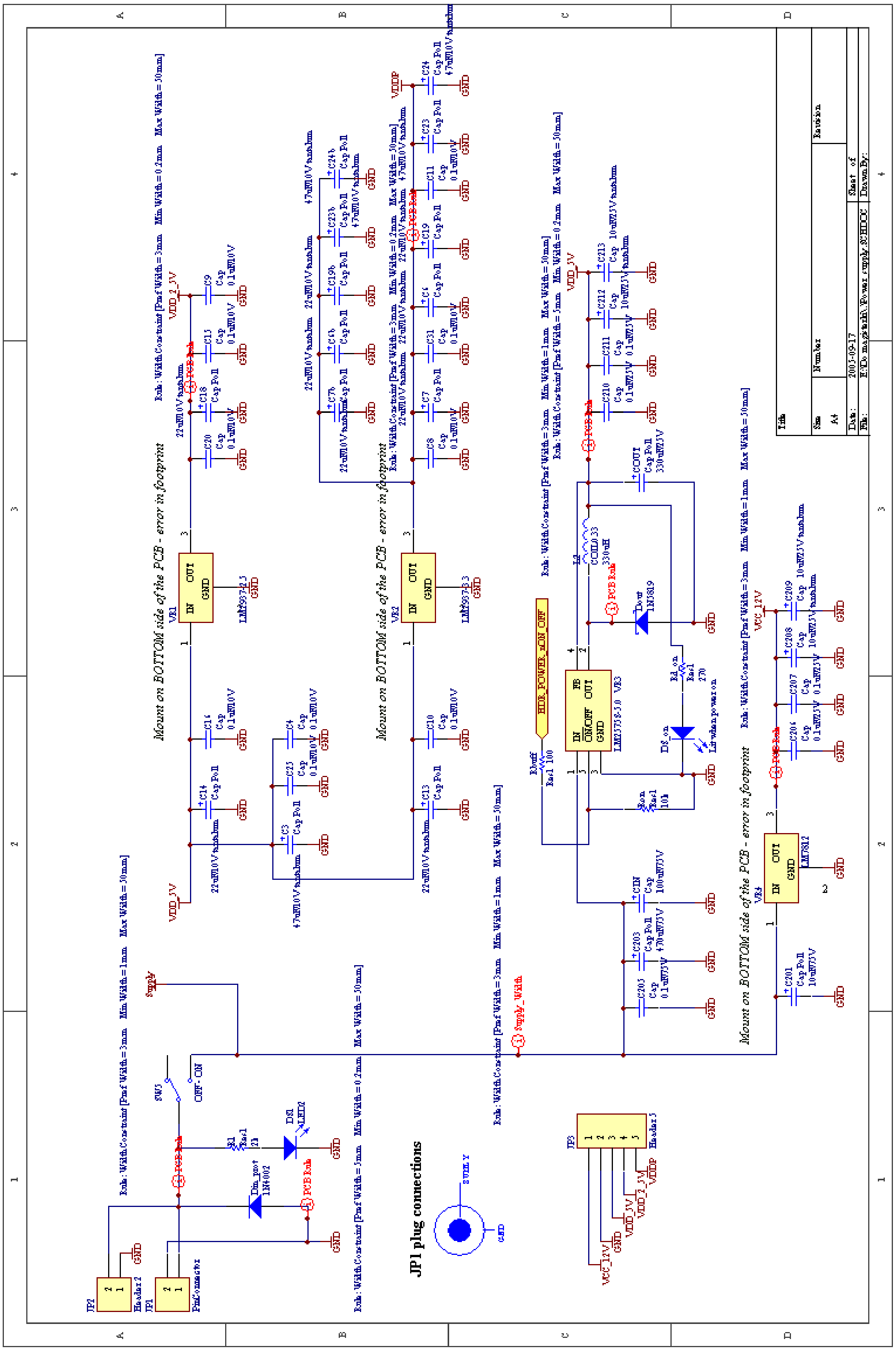


Figure A.2. FPGA board – power supply

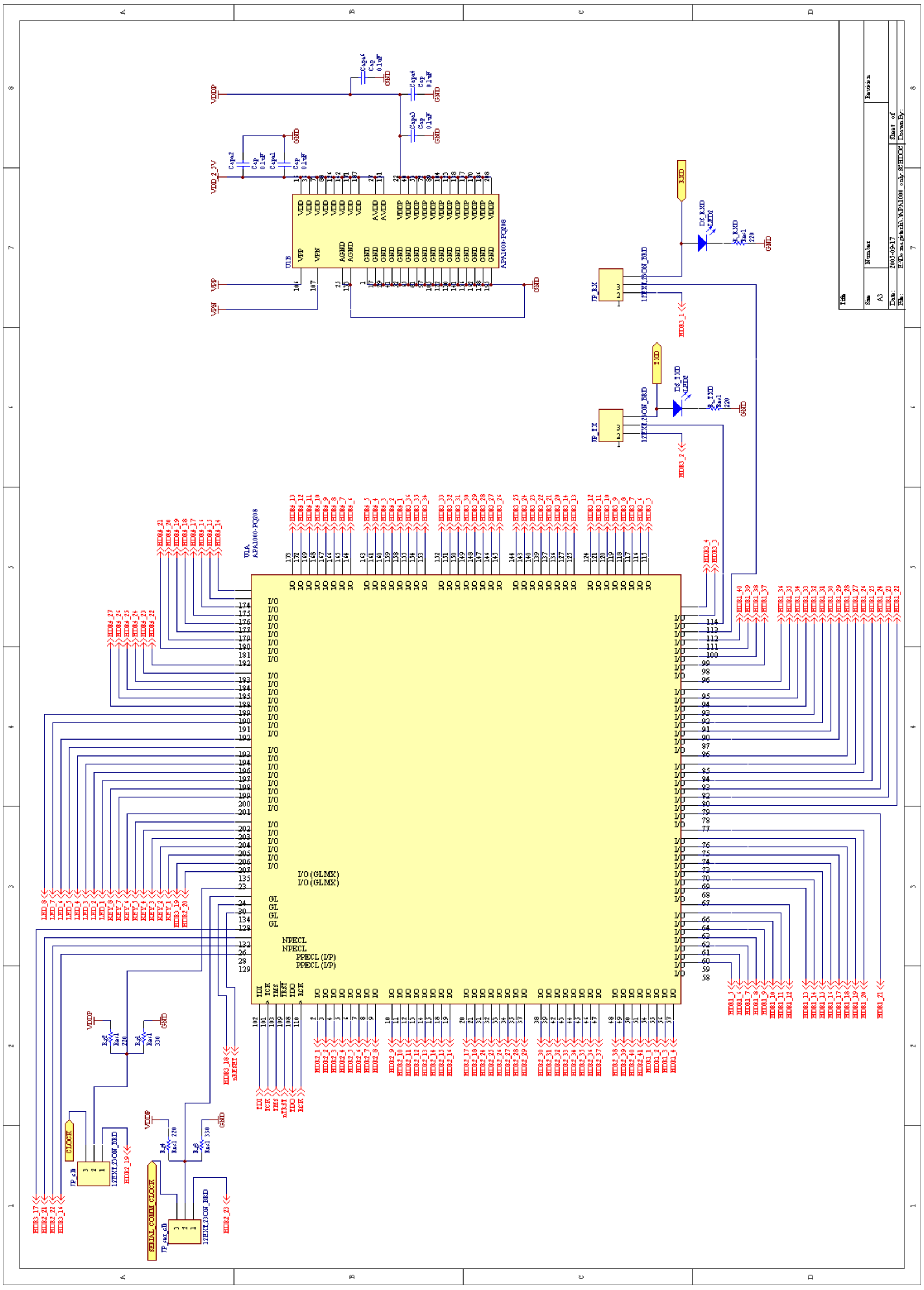


Figure A.3. FPGA board – the FPGA

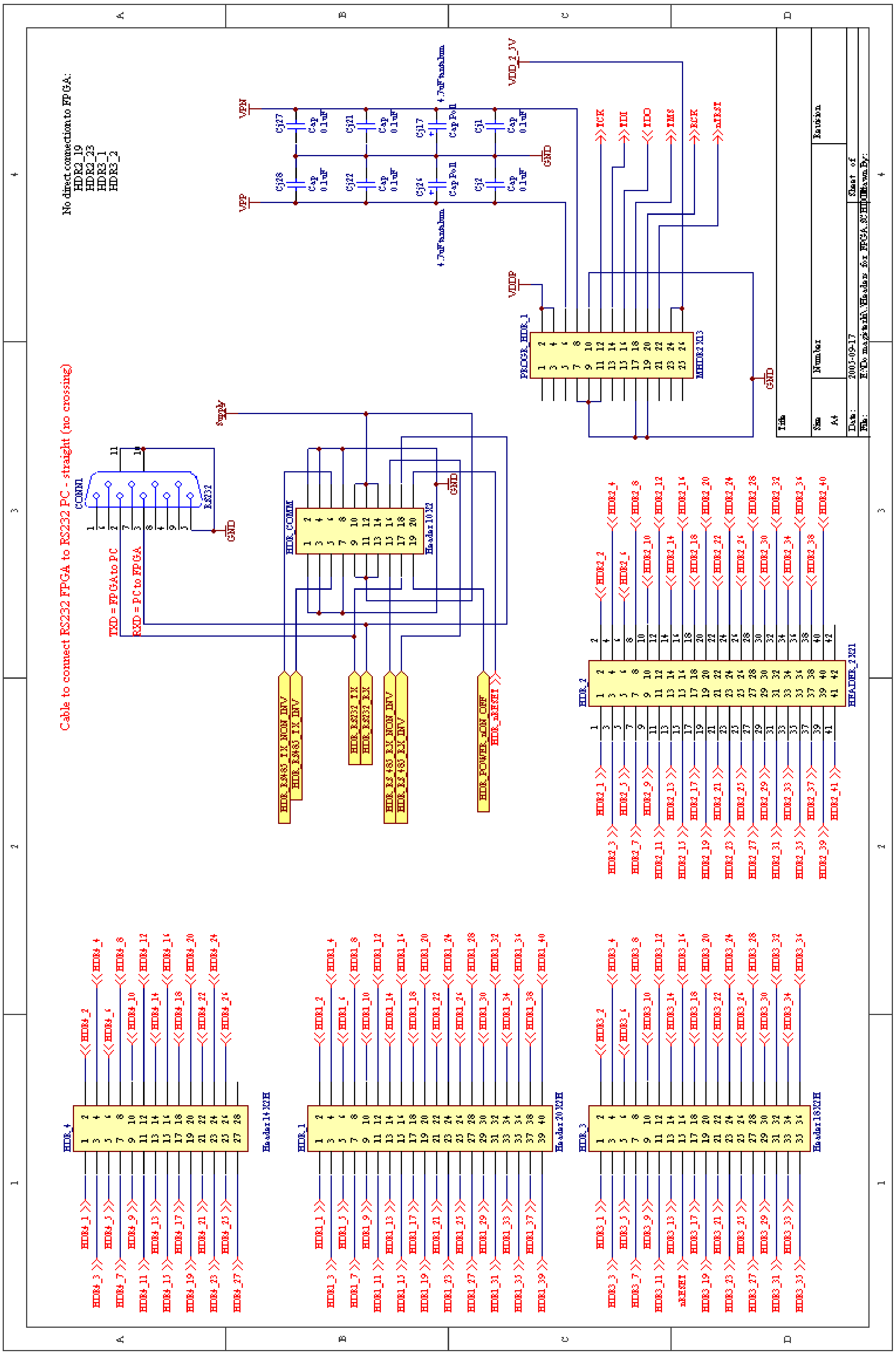


Figure A.4. FPGA board – headers and sockets

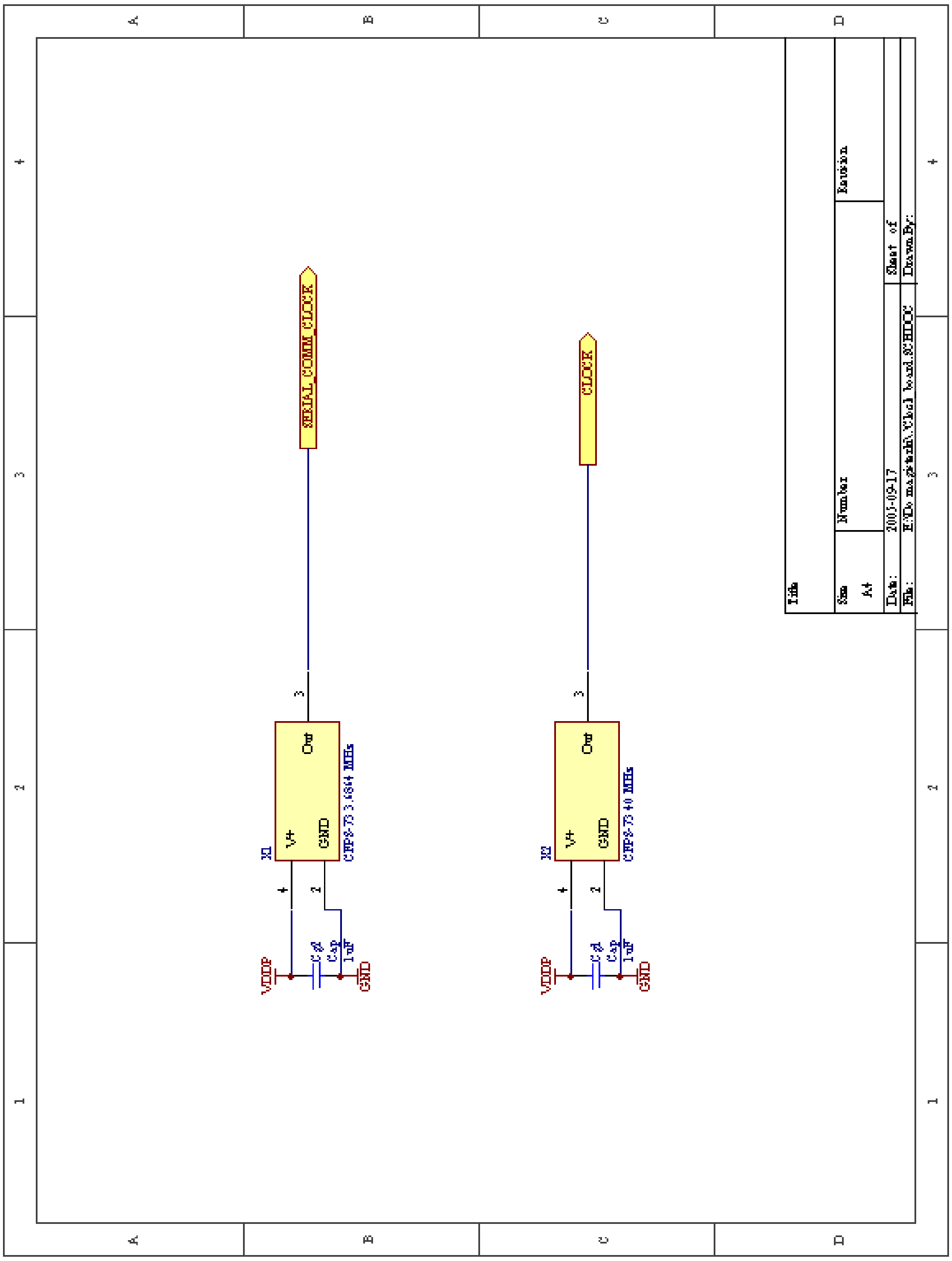


Figure A.5. FPGA board – clock generators

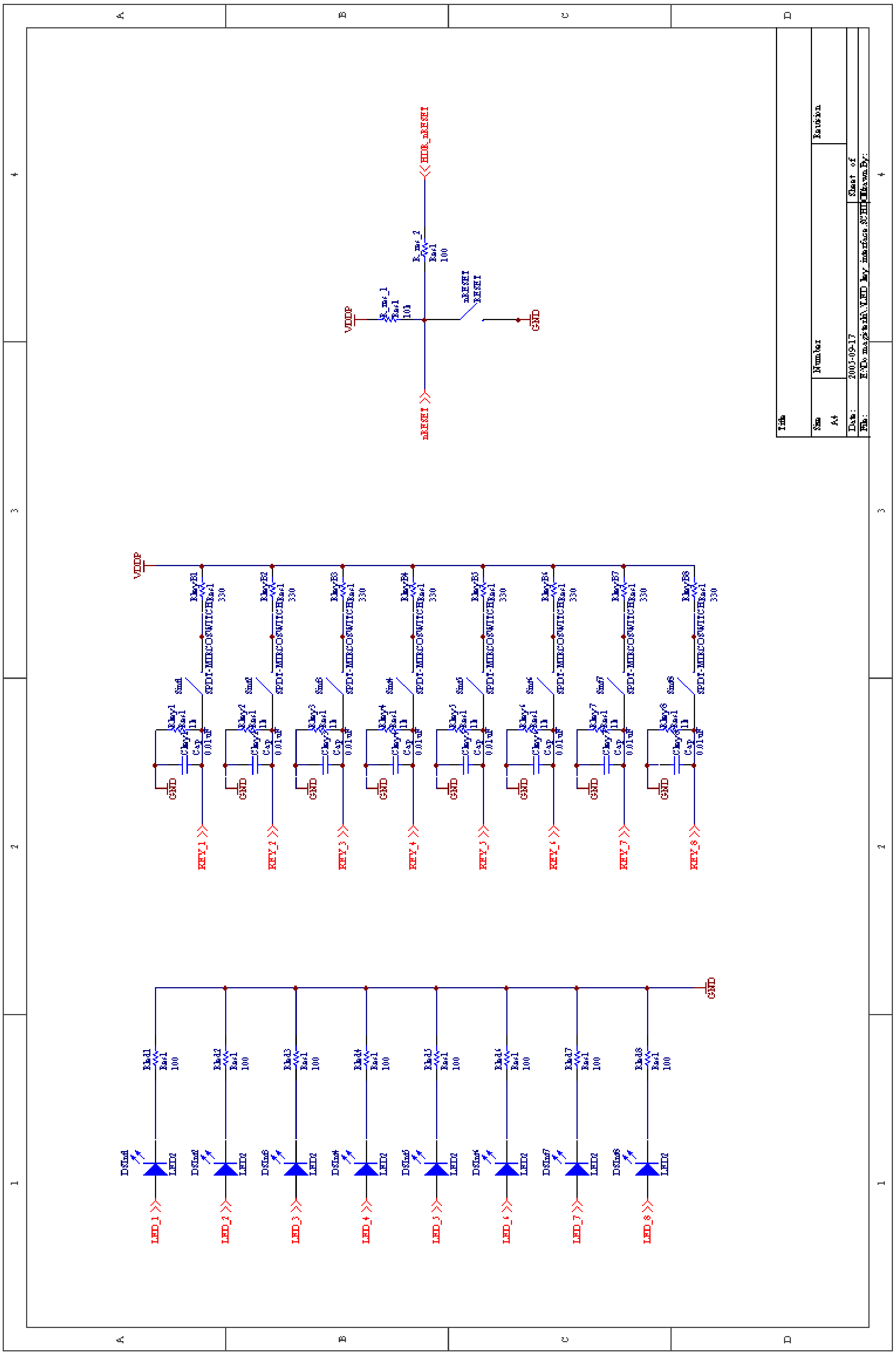


Figure A.6. FPGA board – LEDs and push buttons

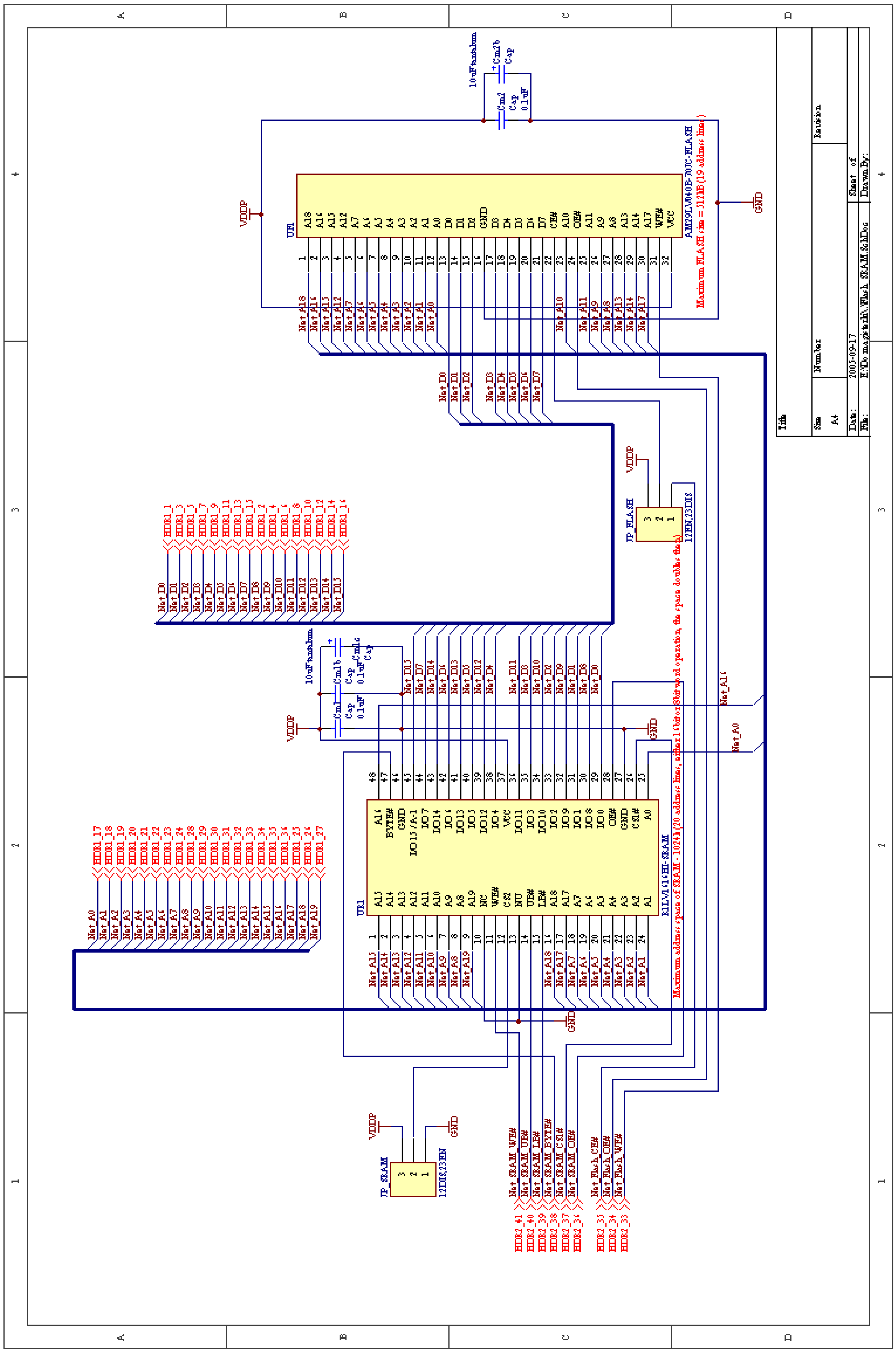


Figure A.7. FPGA board – SRAM and Flash

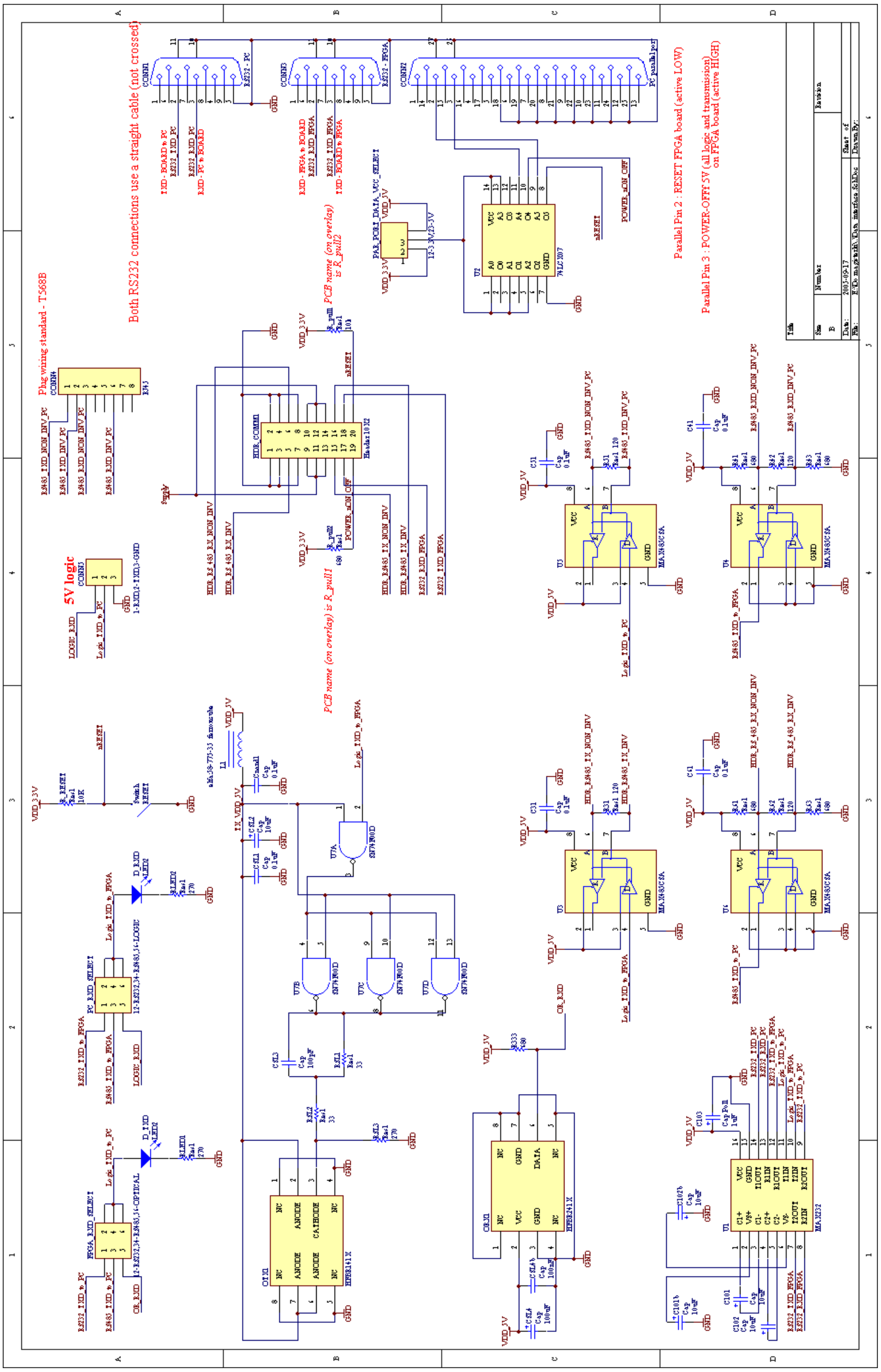

Both RS232 connections use a straight cable (not crossed)
Plug wiring standard - T568B
5V logic
PCB name (on overlay) is R_pull1
PCB name (on overlay) is R_pull2
Parallel Pin 2 : RESET FPGA board (active LOW)
Parallel Pin 3 : POWER-OFF 5V (all logic and transmission) on FPGA board (active HIGH)
RS232 - PC
RS232 - FPGA
PC parallel port
74LC1G07
MAX483CSA
MAX232
HFBR2412
HFBR1412
SN74F00D
Header 10X2
RJ45
nRESET
POWER_nON_OFF
VDD_3.3V
VDD_5V
GND

Figure A.8. Transceiver board – communication interfaces

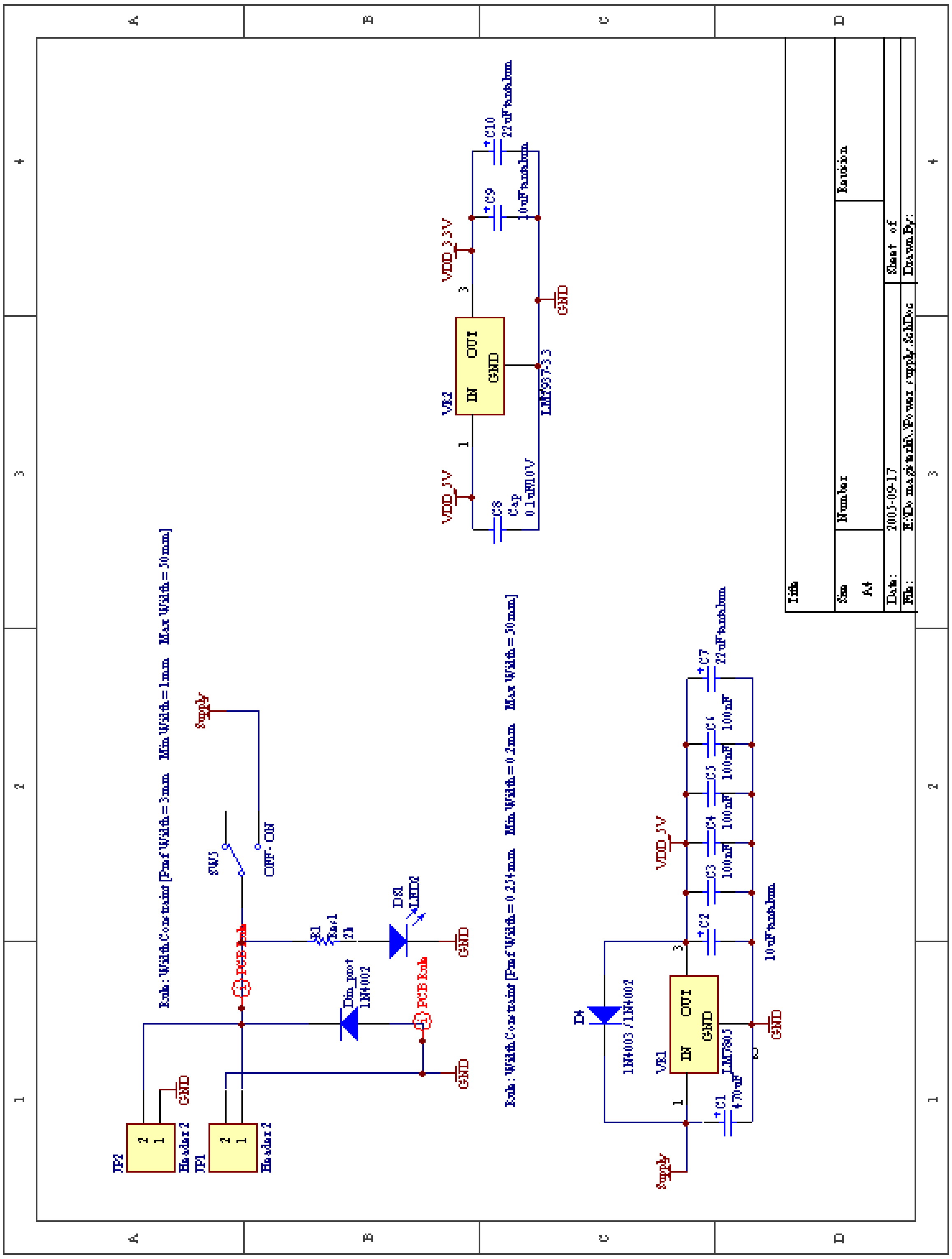


Figure A.9. Transceiver board – power supply

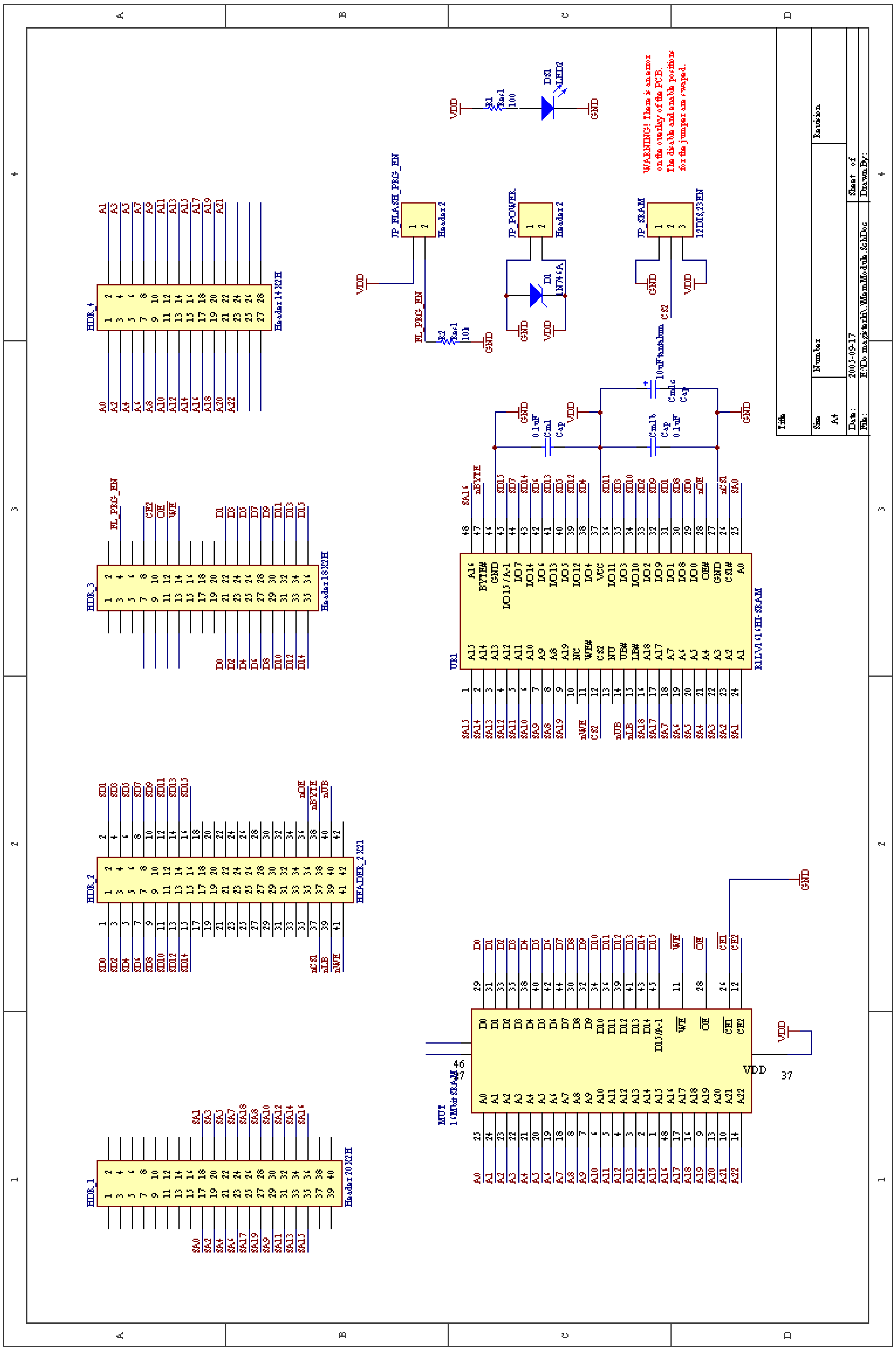


Figure A.10. Expansion board for FPGA

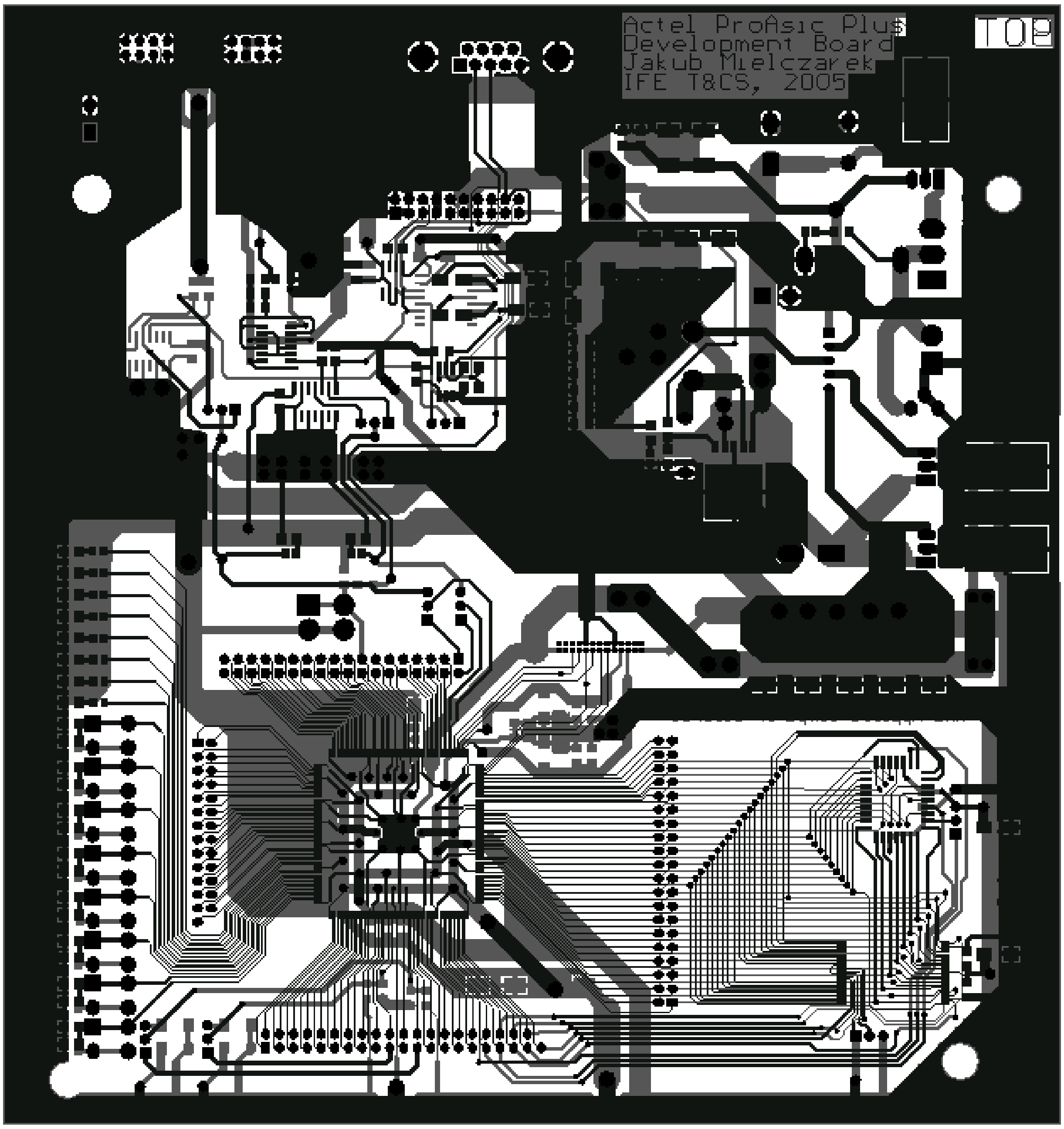


Figure A.11. FPGA PCB

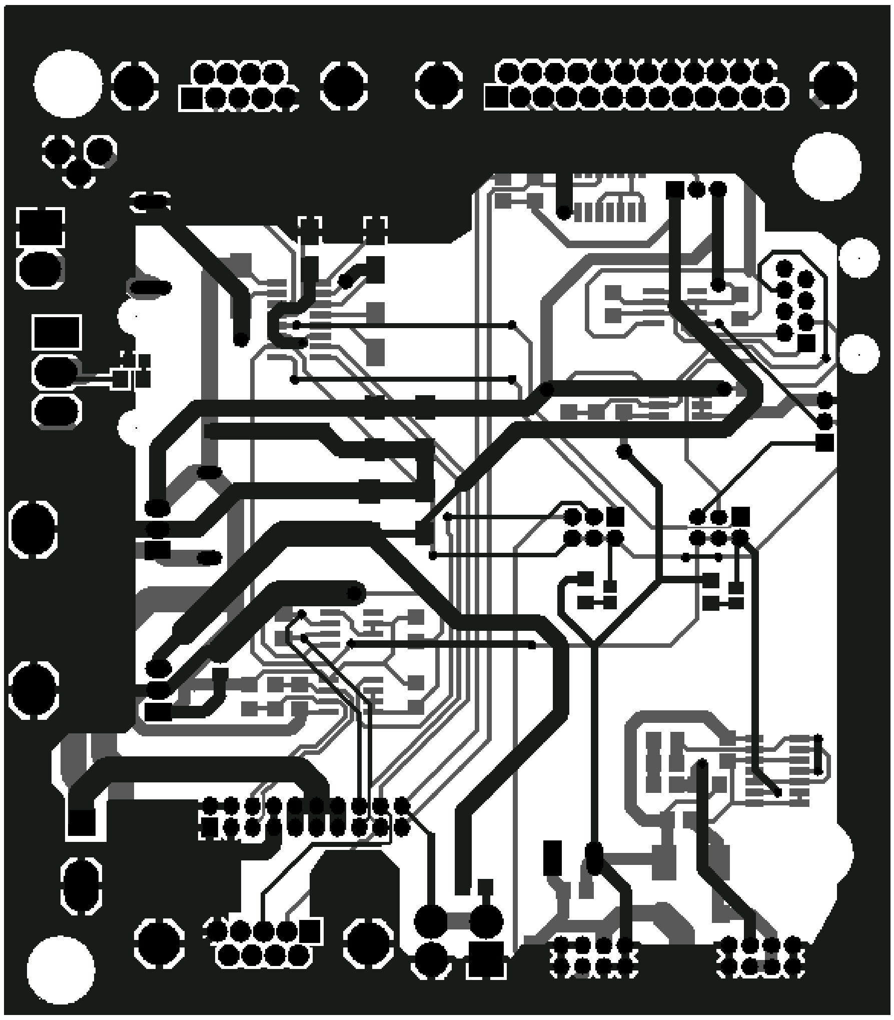


Figure A.12. Transceiver PCB

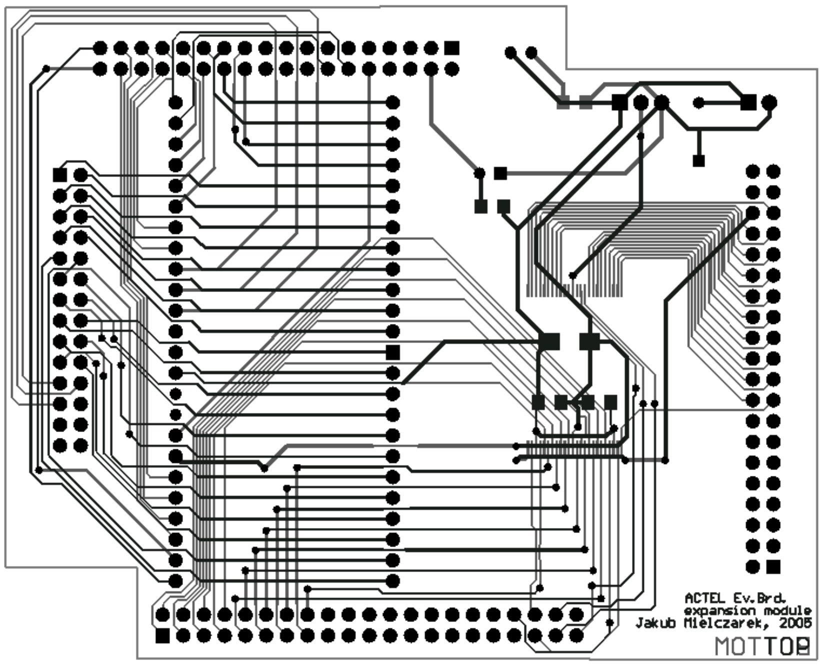


Figure A.13. FPGA expansion PCB

Appendix B Examples of VHDL code

Listing B.1. UART receiver

```
-- UART_RX.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.all;

entity UART_RX is
port(
        RXD : in std_logic; -- serial input line
        RST : in std_logic; -- RESET active high
        CLK : in std_logic; -- clock dependent on baud rate (16x oversampling)
        DOUT : out std_logic_vector (7 downto 0); -- output bus, 8 bits wide
        DREADY : out std_logic; -- flag indicating that received byte is ready, active high
        FRM_ERR : out std_logic; -- flag indicating that framing error occured
        SE  : out std_logic -- single error indication for RadTol version
);
end entity UART_RX;

architecture ARCH_UART_RX of UART_RX is

    -- state constants
    constant rx_t_idle      : std_logic_vector(1 downto 0) := "00";
    constant rx_t_rb_start  : std_logic_vector(1 downto 0) := "01";
    constant rx_t_rx_byte   : std_logic_vector(1 downto 0) := "10";
    constant rx_t_rb_stop   : std_logic_vector(1 downto 0) := "11";


    signal sampler_count : integer range 0 to 15;
    signal bit_count : integer range 0 to 7;
    signal rx_reg : std_logic_vector (7 downto 0);
    signal current_state : std_logic_vector(1 downto 0);
    signal prev_sin, prev_prev_sin : std_logic;

begin

        fsm : process (RST, CLK) is
        begin
                if RST = '0' then -- RST SEQUENCE
                    rx_reg <= (others => '0');
                    FRM_ERR <= '0';
                    current_state <= rx_t_idle;
                    sampler_count <= 0;
                    prev_sin <= '0';
                    prev_prev_sin <= '0';
                    DREADY <= '0';
                    DOUT <= (others => '0');
                    bit_count <= 0;
                elsif rising_edge (CLK) then -- OPERATION
                    prev_prev_sin <= prev_sin;   -- store last sample of RXD
                    prev_sin <= RXD;             -- sample and hold RXD
                    if current_state = rx_t_idle then
                       if prev_sin = '0' and prev_prev_sin = '1' then
                       current_state <= rx_t_rb_start;
                       sampler_count <= 7;
                    else
                          if sampler_count = 15 then -- sampling data on serial input
                          sampler_count <= 0;
                          case current_state is
                          when rx_t_rb_start =>
                                if prev_sin = '0' then
                                   current_state <= rx_t_rx_byte;
                                   DREADY <= '0';
                                   FRM_ERR <= '0';
                                else
                                   current_state <= rx_t_idle;
                                end if;
                          when rx_t_rx_byte =>
                                rx_reg <= prev_sin & rx_reg (7 downto 1);
                                if bit_count = 7 then
```

Listing B.1 (cont.)

```
                                    current_state <= rx_t_rb_stop;
                                 else
                                    bit_count <= bit_count + 1;
                                 end if;
                           when rx_t_rb_stop =>
                                 bit_count <= 0;
                                 if prev_sin = '0' then
                                    FRM_ERR <= '1';
                                 end if;
                                 DOUT <= rx_reg;
                                 current_state <= rx_t_idle;
                                 DREADY <= '1';
                           when others =>
                                 FRM_ERR <= '0';
                                 DREADY <= '0';
                                       current_state <= rx_t_idle;
                           end case;
                        else
                            sampler_count <= sampler_count + 1;
                        end if;
                     end if;
                end if;
        end process fsm;

end architecture ARCH_UART_RX;
```

Listing B.2. UART transmitter

```
-- UART_TX.vhd
library ieee;
use ieee.std_logic_1164.all;

entity UART_TX is
port(
        DIN  : in std_logic_vector (7 downto 0); -- input byte to be transmitted
        LOAD : in std_logic; -- active high signal indicating that new data for transmission
                            -- is available can be asserted continously, is ignored during
                            -- transmission
        RST  : in std_logic; -- asynchronous RST
        CLK  : in std_logic; -- transmitter CLK, dependent on the set baud rate
        TXD  : out std_logic; -- serial output line
        BUSY : out std_logic; -- active high signal indicating that transmission of a byte
                              -- is in progress transmitter ignores LOAD and input data
                              -- until current transmission completes
    SE   : out std_logic -- single error indication for RadTol version
);
end entity UART_TX;

architecture ARCH_UART_TX of UART_TX is

    type TX_type is (idle, tx_byte);

    signal transmit_reg : std_logic_vector (9 downto 0);
    signal state : TX_type;
    signal bit_cnt : integer range 0 to 9;

begin
        tx_process : process (CLK, RST)
        begin
                if RST = '0' then -- RESET SEQUENCE
                        transmit_reg <= (others => '1'); -- clear internal tx register
                        BUSY <= '0'; -- clear BUSY flag
                        state <= idle; -- RST state
                        bit_cnt <= 0;
                elsif rising_edge(CLK) then -- OPERATION
                        case state is
                        when idle =>
                                if LOAD = '1' then -- LOADING DATA
                                        transmit_reg(8 downto 1) <= DIN; -- load data
                                        transmit_reg(9) <= '1'; -- set stop bit
```

Listing B.2 (cont.)

```
                                      transmit_reg(0) <= '0'; -- clear start bit
                                      state <= tx_byte;
                                      BUSY <= '1';
                                      bit_cnt <= 0;
                              end if;
                      when tx_byte => -- TRANSMITTING DATA
                               if bit_cnt = 9 then -- when all transmitted
                                      state <= idle;  -- return to idle
                                      BUSY <= '0';
                               else -- if not all bits transmitted
                                      -- perform shift right
                                     transmit_reg(8 downto 0) <= transmit_reg(9 downto 1);
                                     bit_cnt <= bit_cnt + 1;
                                     state <= tx_byte;
                              end if;
                      end case;
              end if;
        end process tx_process;

        TXD <= transmit_reg(0);

end architecture ARCH_UART_TX;
```

Listing B.3. UART transmitter FIFO controller

```
-- UART_TX_FIFO_controller.vhd

library ieee;
use ieee.std_logic_1164.all;

entity UART_TX_FIFO_CONTROLLER is
port(
        RST : in std_logic;
        CLK : in std_logic;
        TX_BUSY : in std_logic;
        F_EMPTY : in std_logic;
        FIFO_RD : out std_logic;
        LOAD : out std_logic;
    SE   : out std_logic
);
end entity UART_TX_FIFO_CONTROLLER;

architecture ARCH_UART_TX_FIFO_CONTROLLER of UART_TX_FIFO_CONTROLLER is
        type state_type is (idle, s0, s1, s2);
        signal state, next_state : state_type;

    signal prev_tx_busy, prev_f_empty : std_logic;
begin
        state_logic : process (prev_TX_BUSY, prev_F_EMPTY, state) is
        begin
                next_state <= state;
         case state is
            when idle =>
                if prev_F_EMPTY = '0' then
                    next_state <= s0;
                end if;
            when s0 =>
                if prev_TX_BUSY = '1' then
                    next_state <= s1;
                elsif prev_TX_BUSY = '0' then
                    next_state <= s2;
                end if;
            when s1 =>
                if prev_TX_BUSY = '0' then
                    next_state <= s2;
                end if;
            when s2 =>
                if prev_TX_BUSY = '1' then
                    next_state <= idle;
                end if;
```

Listing B.3 (cont.)

```
         when others =>
                next_state <= idle;
        end case;
       end process state_logic;

    output_logic : process (state) is
    begin
        case state is
            when idle =>
                FIFO_RD <= '1';
                LOAD <= '0';
            when s0 =>
                FIFO_RD <= '0';
                LOAD <= '0';
            when s1 =>
                FIFO_RD <= '1';
                LOAD <= '0';
            when s2 =>
                FIFO_RD <= '1';
                LOAD <= '1';
            when others =>
                FIFO_RD <= '1';
                LOAD <= '0';
        end case;
    end process output_logic;

    state_update : process (RST, CLK) is
    begin
        if RST = '0' then
            state <= idle;
            prev_tx_busy <= '1';
            prev_f_empty <= '1';
        elsif falling_edge(CLK) then
            state <= next_state;
            prev_tx_busy <= TX_BUSY;
            prev_f_empty <= F_EMPTY;
        end if;
    end process state_update;

end architecture ARCH_UART_TX_FIFO_CONTROLLER;
```

Listing B.4. System Arbiter

```
-- Arbiter.vhd
library ieee;
use ieee.std_logic_1164.all;

entity ARBITER is
port(
    CLK, RST        :   in std_logic;
    PRG_LDR_BUSY    :   in std_logic;
    FL_PRG_EN       :   in std_logic;
    FL_PRG_CLAIM_BUS : in std_logic;
    FL_PRG_GRANT_BUS : out std_logic;
    CRC_CLAIM_BUS   :   in std_logic;
    CRC_GRANT_BUS   :   out std_logic;
    SRAM_SCRB_BUSY  :   in std_logic;
    REGF_SCRB_BUSY  :   in std_logic;
    STACK_SCRB_BUSY :   in std_logic;

    BUS_FOR_PRG_LDR    :   out std_logic;
    BUS_FOR_SCRB       :   out std_logic;
    BUS_FOR_CPU        :   out std_logic;
    BUS_FOR_FL_PRG     :   out std_logic;
    BUS_FOR_CRC        :   out std_logic;

    REGF_FOR_SCRB       :   out std_logic;
    STACK_FOR_SCRB      :   out std_logic;

    HALT_CPU    :   out std_logic;

    PRG_LDR_START_SECTOR  :   out std_logic_vector(2 downto 0);
```

Listing B.4 (cont.)

```
    RUN_PRG_LOADER      :   out std_logic;
    RUN_STACK_SCRB      :   out std_logic;
    RUN_REGF_SCRB       :   out std_logic;
    RUN_SRAM_SCRB       :   out std_logic;

    SCRB_DONE   :   out std_logic; -- sets scrubbing flag in status register

    SE  :   out std_logic
);
end entity ARBITER;

architecture AR of ARBITER is

constant MAX_SCRUB_COUNTER : integer := 1250000;

type state_type is (   start, start_prg_loader, wait_1, check_if_loader_busy,
                       loader_complete, run_cpu, run_flash_programmer, wait_for_flash_prg,
                       run_crc, wait_for_crc, rerun_cpu, check_if_run_scrubbers,
                       run_scrubbers, wait_2, check_if_scrubbing_done, scrubbing_done
                   );

type cnt_state_type is (count, notify, verify, wait_for_scrubbers);

signal state : state_type;
signal cnt_state : cnt_state_type;
signal run_scrubbers_now, scrubbers_running : std_logic;
signal scrb_counter : integer range 0 to MAX_SCRUB_COUNTER;

begin
    main_fsm : process (RST, CLK) is
    begin
        if RST = '0' then
            state <= start;
            SCRB_DONE <= '0';
            BUS_FOR_PRG_LDR <= '0';
            BUS_FOR_SCRB <= '0';
            BUS_FOR_CPU <= '0';
            BUS_FOR_FL_PRG <= '0';
            BUS_FOR_CRC <= '0';
            REGF_FOR_SCRB <= '0';
            STACK_FOR_SCRB <= '0';
            HALT_CPU <= '1';
                   RUN_PRG_LOADER <= '0';
                   RUN_STACK_SCRB <= '0';
                   RUN_REGF_SCRB <= '0';
                   RUN_SRAM_SCRB <= '0';
            FL_PRG_GRANT_BUS <= '0';
            CRC_GRANT_BUS <= '0';
            scrubbers_running <= '0';
            PRG_LDR_START_SECTOR <= (others => '0');
        elsif rising_edge(CLK) then
            case state is
            when start =>
                -- grant access to flash and sram bus to loader
                BUS_FOR_PRG_LDR <= '1';
                SCRB_DONE <= '0';
                if FL_PRG_EN = '1' then -- if flash programming is enabled
                    PRG_LDR_START_SECTOR <= (others => '0'); -- firmware upgrade software
                                                             -- starts at 0th address in the
                                                             -- flash
                else -- otherwise
                    PRG_LDR_START_SECTOR <= "001";  -- the start address of the first
                                                    -- 'normal' program
                                                    -- is the beginning of the 1st sector
                end if;
                state <= start_prg_loader;
            when start_prg_loader =>
                RUN_PRG_LOADER <= '1';
                state <= wait_1;
            when wait_1 =>
                state <= check_if_loader_busy;
```

Listing B.4 (cont.)

```
            when check_if_loader_busy =>
                RUN_PRG_LOADER <= '0';
                if PRG_LDR_BUSY = '1' then
                    state <= wait_1;
                else
                    state <= loader_complete;
                end if;
            when loader_complete =>
                BUS_FOR_PRG_LDR <= '0';
                BUS_FOR_CPU <= '1';
                state <= run_cpu;
            when run_cpu =>
                SCRB_DONE <= '0';
                HALT_CPU <= '0';
                if FL_PRG_EN = '0' then
                    state <= check_if_run_scrubbers;
                else
                    state <= run_cpu;
                end if;
                if FL_PRG_CLAIM_BUS = '1' then
                    HALT_CPU <= '1';
                    state <= run_flash_programmer;
                end if;
                if CRC_CLAIM_BUS = '1' then
                    HALT_CPU <= '1';
                    state <= run_crc;
                end if;
            when run_crc =>
                BUS_FOR_CRC <= '1';
                BUS_FOR_CPU <= '0';
                CRC_GRANT_BUS <= '1';
                state <= wait_for_crc;
            when wait_for_crc =>
                CRC_GRANT_BUS <= '0';
                if CRC_CLAIM_BUS = '1' then
                    state <= wait_for_crc;
                else
                    state <= rerun_cpu;
                end if;
            when run_flash_programmer =>
                BUS_FOR_FL_PRG <= '1';
                BUS_FOR_CPU <= '0';
                FL_PRG_GRANT_BUS <= '1';
                state <= wait_for_flash_prg;
            when wait_for_flash_prg =>
                FL_PRG_GRANT_BUS <= '0';
                if FL_PRG_CLAIM_BUS = '1' then
                    state <= wait_for_flash_prg;
                else
                    state <= rerun_cpu;
                end if;
            when rerun_cpu =>
                BUS_FOR_FL_PRG <= '0';
                BUS_FOR_CRC <= '0';
                BUS_FOR_CPU <= '1';
                state <= run_cpu;
            when check_if_run_scrubbers =>
                if run_scrubbers_now = '1' then
                    BUS_FOR_CPU <= '0';
                    BUS_FOR_SCRB <= '1';
                    REGF_FOR_SCRB <= '1';
                    STACK_FOR_SCRB <= '1';
                    HALT_CPU <= '1';
                    scrubbers_running <= '1';
                    state <= run_scrubbers;
                else
                    state <= run_cpu;
                end if;
            when run_scrubbers =>
                RUN_SRAM_SCRB <= '1';
                RUN_STACK_SCRB <= '1';
```

Listing B.4 (cont.)

```
                RUN_REGF_SCRB <= '1';
                state <= wait_2;
            when wait_2 =>
                state <= check_if_scrubbing_done;
            when check_if_scrubbing_done =>
                RUN_SRAM_SCRB <= '0';
                RUN_REGF_SCRB <= '0';
                RUN_STACK_SCRB <= '0';

                if (SRAM_SCRB_BUSY = '0' and REGF_SCRB_BUSY = '0'
                     and STACK_SCRB_BUSY = '0') then
                    state <= scrubbing_done;
                else
                    state <= wait_2;
                end if;
            when scrubbing_done =>
                SCRB_DONE <= '1';
                BUS_FOR_CPU <= '1';
                BUS_FOR_SCRB <= '0';
                REGF_FOR_SCRB <= '0';
                STACK_FOR_SCRB <= '0';
                scrubbers_running <= '0';
                state <= run_cpu;
            end case;
        end if;
    end process main_fsm;

    scrub_counter : process (RST, CLK) is
    begin
        if RST = '0' then
            scrb_counter <= 0;
               run_scrubbers_now <= '0';
            cnt_state <= count;
        elsif falling_edge(CLK) then
            case cnt_state is
            when count =>
                if scrb_counter = MAX_SCRUB_COUNTER then
                    scrb_counter <= 0;
                    cnt_state <= notify;
                else
                    scrb_counter <= scrb_counter + 1;
                end if;
            when notify =>
                run_scrubbers_now <= '1';
                cnt_state <= verify;
            when verify =>
                if scrubbers_running = '0' then
                    cnt_state <= verify;
                else
                    cnt_state <= wait_for_scrubbers;
                end if;
            when wait_for_scrubbers =>
                run_scrubbers_now <= '0';
                if scrubbers_running = '1' then
                    cnt_state <= wait_for_scrubbers;
                else
                    cnt_state <= count;
                end if;
            end case;
        end if;
    end process scrub_counter;

end architecture AR;
```

Listing B.5. (13, 8) Hamming Code encoder

```
library ieee;
use ieee.std_logic_1164.all;
entity HAMMING_ENC_8D_5P is
        port (
                input : in std_logic_vector(7 downto 0);
                output : out std_logic_vector(12 downto 0)
                );
end entity HAMMING_ENC_8D_5P;

architecture H_ENCODER_FUNC of HAMMING_ENC_8D_5P is
begin
ENCODING_PROCESS : process (input)
        variable temp_data_out : std_logic_vector(12 downto 0);
        variable parity : std_logic_vector(4 downto 0);
        begin
                temp_data_out(3) := input(0);
                temp_data_out(5) := input(1);
                temp_data_out(6) := input(2);
                temp_data_out(7) := input(3);
                temp_data_out(9) := input(4);
                temp_data_out(10) := input(5);
                temp_data_out(11) := input(6);
                temp_data_out(12) := input(7);
                parity(1) := temp_data_out(3) xor temp_data_out(5) xor temp_data_out(7)
                             xor temp_data_out(9)
                             xor temp_data_out(11);
                parity(2) := temp_data_out(3) xor temp_data_out(6) xor temp_data_out(7)
                             xor temp_data_out(10) xor temp_data_out(11);
                parity(3) := temp_data_out(5) xor temp_data_out(6) xor temp_data_out(7)
                             xor temp_data_out(12);
                parity(4) := temp_data_out(9) xor temp_data_out(10) xor temp_data_out(11)
                             xor temp_data_out(12);
                temp_data_out(1) := parity(1);
                temp_data_out(2) := parity(2);
                temp_data_out(4) := parity(3);
                temp_data_out(8) := parity(4);
                parity(0) := temp_data_out(1) xor temp_data_out(2) xor temp_data_out(3)
                             xor temp_data_out(4) xor temp_data_out(5) xor temp_data_out(6)
                             xor temp_data_out(7) xor temp_data_out(8) xor temp_data_out(9)
                             xor temp_data_out(10) xor temp_data_out(11)
                             xor temp_data_out(12);
                temp_data_out(0) := parity(0);
                output <= temp_data_out;
end process;

end H_ENCODER_FUNC;
```

Listing B.6. (13, 8) Hamming Code decoder

```
library ieee;
use ieee.std_logic_1164.all;
entity HAMMING_DEC_8D_5P is
port (
        input : in std_logic_vector(12 downto 0);
        output : out std_logic_vector(7 downto 0);
        single_error : out std_logic;
        double_error : out std_logic
        );
end entity HAMMING_DEC_8D_5P;

library ieee;
use ieee.std_logic_1164.all;
entity DEMUX is
port(
        slct : in std_logic_vector(3 downto 0);
        output : out std_logic_vector(11 downto 0)
);
end DEMUX;

architecture DEMUX_ARCH of DEMUX is
begin
process (slct)
variable tmp_out : std_logic_vector(11 downto 0);
begin
        tmp_out := "000000000000";
        case slct is
        when "0001" =>
                tmp_out(0) := '1';
        when "0010" =>
                tmp_out(1) := '1';
        when "0011" =>
                tmp_out(2) := '1';
        when "0100" =>
                tmp_out(3) := '1';
        when "0101" =>
                tmp_out(4) := '1';
        when "0110" =>
                tmp_out(5) := '1';
        when "0111" =>
                tmp_out(6) := '1';
        when "1000" =>
                tmp_out(7) := '1';
        when "1001" =>
                tmp_out(8) := '1';
        when "1010" =>
                tmp_out(9) := '1';
        when "1011" =>
                tmp_out(10) := '1';
        when "1100" =>
                tmp_out(11) := '1';
        when others    =>
                tmp_out := "000000000000";
        end case;
                output <= tmp_out;
end process;
end DEMUX_ARCH;

library ieee;
use ieee.std_logic_1164.all;
entity Syndrome_computation is
        port (
                input : in std_logic_vector(12 downto 0);
                syndr_out : out std_logic_vector(3 downto 0);
                dbe_parity : out std_logic
        );
end Syndrome_computation;

architecture Syndrome_computation_arch of Syndrome_computation is
begin
        syndr_out(0) <= input(1) xor input(3) xor input(5) xor input(7) xor input(9)
                        xor input(11);
```

Listing B.6 (cont.)

```
       syndr_out(1) <= input(2) xor input(3) xor input(6) xor input(7) xor input(10)
                       xor input(11);
       syndr_out(2) <= input(4) xor input(5) xor input(6) xor input(7) xor input(12);
       syndr_out(3) <= input(8) xor input(9) xor input(10) xor input(11) xor input(12);
       dbe_parity <= input(0) xor input(1) xor input(2) xor input(3) xor input(4)
                      xor input(5) xor input(6) xor input(7) xor input(8) xor input(9)
                      xor input(10) xor input(11) xor input(12);
end Syndrome_computation_arch;

architecture H_DECODER_FUNC of HAMMING_DEC_8D_5P is
component Syndrome_computation
       port (
              input : in std_logic_vector(12 downto 0);
              syndr_out : out std_logic_vector(3 downto 0);
              dbe_parity : out std_logic
       );
end component;
component DEMUX
       port (
              slct : in std_logic_vector(3 downto 0);
              output : out std_logic_vector(11 downto 0)
       );
end component;
signal syndrome_sig : std_logic_vector(3 downto 0);
signal dbe_parity_sig : std_logic;
signal xorin_signal : std_logic_vector(11 downto 0);
begin

syndrome_computation_c : Syndrome_computation
port map (input => input, syndr_out => syndrome_sig, dbe_parity => dbe_parity_sig);

demux_to_xors : DEMUX
port map (slct => syndrome_sig, output => xorin_signal);
       output(0) <= input(3) xor xorin_signal(2);
       output(1) <= input(5) xor xorin_signal(4);
       output(2) <= input(6) xor xorin_signal(5);
       output(3) <= input(7) xor xorin_signal(6);
       output(4) <= input(9) xor xorin_signal(8);
       output(5) <= input(10) xor xorin_signal(9);
       output(6) <= input(11) xor xorin_signal(10);
       output(7) <= input(12) xor xorin_signal(11);
       single_error <= '1' when ((syndrome_sig = "0000" and dbe_parity_sig = '1')
                      or (syndrome_sig /= "0000" and dbe_parity_sig = '1')) else '0';
       double_error <= '1' when (syndrome_sig /= "0000" and dbe_parity_sig = '0') else '0';

end H_DECODER_FUNC;
```

Listing B.7. Program Memory Scrubber

```
-- rom_scrubber.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ROM_SCRUBBER is
port(
    RST     : in  std_logic;
    CLK     : in  std_logic;
    START   : in  std_logic;
    CE      : in  std_logic;
    DIN     : in  std_logic_vector(11 downto 0);
    DOUT    : out std_logic_vector(17 downto 0);
    ADDR    : out std_logic_vector(19 downto 0);
    SR_NOE  : out std_logic;
    SR_NWE  : out std_logic;
    SR_NCE  : out std_logic;
    BUSY    : out std_logic;
    INC_ROM_SEU_REG : out std_logic;
    SRAM_SE :   in std_logic;

    SE      : out std_logic
);
end entity ROM_SCRUBBER;

architecture ARCH of ROM_SCRUBBER is

component HAMMING_DEC_12D_6P is
port (
       input : in std_logic_vector(17 downto 0);
       output : out std_logic_vector(11 downto 0);
       single_error : out std_logic;
       double_error : out std_logic
       );
end component HAMMING_DEC_12D_6P;

component HAMMING_ENC_12D_6P is
       port (
              input : in std_logic_vector(11 downto 0);
              output : out std_logic_vector(17 downto 0)
              );
end component HAMMING_ENC_12D_6P;

constant MAX_ROM_ADDR : integer := 2047;

signal s_data : std_logic_vector(17 downto 0);
signal s_address : integer range 0 to (MAX_ROM_ADDR + 1); -- the address counter value
signal prev_SRAM_SE, s_nce , s_noe, s_nwe : std_logic; -- single and double error flag
respectively,
                                   -- for the decoded instruction

type state_type is (IDLE, NEXT_CHECK, READ, CHECK_ERROR, INC_SEU_REG, WRITE, INC_ADDR);
signal state, next_state : state_type; -- current and next state of the FSM

begin

    SR_NOE <= s_noe when CE = '1' else '1';
    SR_NWE <= s_nwe when CE = '1' else '1';
    SR_NCE <= s_nce when CE = '1' else '1';

    DOUT <= s_data;

    state_updt : process (RST, CLK) is -- standard state updating process
    begin
        if RST = '0' then
            state <= IDLE;
            prev_SRAM_SE <= '0';
        elsif rising_edge(CLK) then
            state <= next_state;
            if state = READ then
                prev_SRAM_SE <= SRAM_SE;
            end if;
```

Listing B.7 (cont.)

```
        end if;
    end process state_updt;

    addr_counter : process (RST, CLK) is -- responsible for incrementing address counter
    begin
        if RST = '0' then
            s_address <= 0;
        elsif falling_edge(CLK) then
            if state = INC_ADDR then
                if s_address > MAX_ROM_ADDR then
                    s_address <= 0;
                else
                    s_address <= s_address + 1;
                end if;
            elsif START = '1' then
                s_address <= 0;
            end if;
        end if;
    end process addr_counter;

    output_logic : process (state) is
    begin
        BUSY <= '0';
        INC_ROM_SEU_REG <= '0';
        S_NOE <= '1';
        S_NWE <= '1';
        S_NCE <= '0';
        case state is
            when IDLE =>
                null;
            when NEXT_CHECK =>
                BUSY <= '1';
            when READ =>
                BUSY <= '1';
                S_NOE <= '0';
            when CHECK_ERROR =>
                BUSY <= '1';
                S_NOE <= '0';
            when INC_SEU_REG =>
                BUSY <= '1';
                INC_ROM_SEU_REG <= '1';
            when WRITE =>
                BUSY <= '1';
                S_NWE <= '0';
            when INC_ADDR =>
                BUSY <= '1';
            when others =>
                null;
               end case;
    end process output_logic;

    next_state_prcs : process (state, prev_SRAM_SE, s_address, START) is
    begin
        next_state <= state;
        case state is
            when IDLE =>
                if START = '1' then
                    next_state <= NEXT_CHECK;
                end if;
            when NEXT_CHECK =>
                if s_address > MAX_ROM_ADDR then
                    next_state <= IDLE;
                else
                    next_state <= READ;
                end if;
            when READ =>
                next_state <= CHECK_ERROR;
            when CHECK_ERROR =>
                if prev_SRAM_SE = '1' then
                    next_state <= INC_SEU_REG;
                else
                    next_state <= INC_ADDR;
                end if;
```

Listing B.7 (cont.)

```
         when INC_SEU_REG =>
                 next_state <= WRITE;
             when WRITE =>
                 next_state <= INC_ADDR;
             when INC_ADDR =>
                 next_state <= NEXT_CHECK;
         end case;
     end process next_state_prcs;

     h_enc : HAMMING_ENC_12D_6P
        port map
     (
                input => DIN,
                output => s_data
        );

     ADDR <= conv_std_logic_vector(s_address, 20);

end architecture ARCH;
```

Appendix C Summary of detected SEUs

Table C.1 shows full listing of Single Event Upsets (SEUs) detected in the Memory Under Test (MUT) during the experiments conducted at DESY from 30th of August till 2nd of September, 2005. All SEUs, which matched the pattern of SEU generated for diagnosis (Data Low = 0xFF, Data High = 0xFE) were removed. Hence, 515 SEUs are listed instead of 543 detected, caused by neutrons. The address in a memory chip is obtained by following the operation:

CHIP_ADDR = ADDRESS & 0x07FFFF

where '&' stands for bit-wise "and" operation of the two hexadecimal numbers.

Table C.1. Listing of SEUs detected in the MUT

Time	Data Low	Data High	Address Low	Address High
2005-08-30 13:19:49	FF	FD	F56C48	DD6C48
2005-08-30 13:20:26	FB	FF	F02A2D	D82A2D
2005-08-30 13:21:25	FF	FB	F485D2	DC85D2
2005-08-30 13:22:02	EF	FF	F72E4C	DF2E4C
2005-08-30 13:22:43	FF	7F	F23681	DA3681
2005-08-30 13:22:52	FF	F7	F2ED9D	DAED9D
2005-08-30 13:23:30	EF	FF	F5B333	DDB333
2005-08-30 13:23:39	DF	FF	F66187	DE6187
2005-08-30 14:15:41	FD	FF	F56780	DD6780
2005-08-30 14:27:00	FF	FD	F4ED80	DCED80
2005-08-30 14:33:11	EF	FF	F028F9	D828F9
2005-08-30 14:36:12	7F	FF	F57430	DD7430
2005-08-30 14:40:07	BF	FF	F6B6CC	DEB6CC
2005-08-30 14:45:37	FF	FD	F70693	DF0693
2005-08-30 15:26:46	FF	BF	F4E66E	DCE66E
2005-08-30 15:29:06	**DF**	**FF**	**F72F4C**	**DF2F4C**
2005-08-30 15:40:03	FF	EF	F65DEA	DE5DEA
2005-08-30 15:55:17	FF	FB	F189E8	D989E8
2005-08-30 15:55:45	DF	FF	F392B1	DB92B1
2005-08-30 16:05:35	FE	FF	F6F3C8	DEF3C8
2005-08-30 16:10:34	7F	FF	F4F766	DCF766
2005-08-30 16:20:53	FD	FF	F27BC1	DA7BC1
2005-08-30 16:22:36	DF	FF	F202D3	DA02D3
2005-08-30 16:30:31	FF	FB	F4F5C0	DCF5C0
2005-08-30 16:38:38	FE	FF	F0BA01	D8BA01
2005-08-30 16:56:31	DF	FF	F79354	DF9354
2005-08-30 16:59:51	F7	FF	F64DF2	DE4DF2
2005-08-30 17:04:40	EF	FF	F39492	DB9492
2005-08-30 17:14:15	FF	EF	F5DCC9	DDDCC9
2005-08-30 17:20:09	FF	7F	F7EFF2	DFEFF2
2005-08-30 17:24:16	FD	FF	F20FA1	DA0FA1
2005-08-30 17:33:34	EF	FF	F30AF2	DB0AF2
2005-08-30 17:41:28	FD	FF	F5D242	DDD242
2005-08-30 17:41:41	FF	F7	F6E210	DEE210
2005-08-30 17:48:37	F7	FF	F563FA	DD63FA
2005-08-30 17:51:31	FF	BF	F203C0	DA03C0
2005-08-30 18:05:14	FF	FD	F6AD62	DEAD62
2005-08-30 18:06:00	FF	EF	F22712	DA2712
2005-08-30 18:06:55	FE	FF	F61F92	DE1F92
2005-08-30 18:18:37	FB	FF	F7922D	DF922D
2005-08-30 18:44:57	FF	DF	F3A7E6	DBA7E6
2005-08-30 18:45:19	FD	FF	F54960	DD4960
2005-08-30 18:48:04	7F	FF	F179B0	D979B0
2005-08-30 18:48:28	BF	FF	F33C30	DB3C30
2005-08-30 19:09:34	FF	F7	F0416F	D8416F
2005-08-30 19:13:51	FF	7F	F32475	DB2475
2005-08-30 19:14:13	FF	FD	F4CB0F	DCCB0F
2005-08-30 19:24:04	FF	DF	F039E6	D839E6
2005-08-30 19:24:59	EF	FF	F44B26	DC4B26
2005-08-30 19:27:32	DF	FF	F78DC2	DF8DC2
2005-08-30 19:29:32	FF	EF	F0606C	D8606C
2005-08-30 19:33:09	FF	BF	F05118	D85118
2005-08-30 19:40:35	FE	FF	F115BB	D915BB
2005-08-30 19:42:53	DF	FF	F34C1D	DB4C1D
2005-08-30 20:17:02	FF	DF	F61748	DE1748
2005-08-30 20:17:41	7F	FF	F0F5E1	D8F5E1
2005-08-30 20:21:49	FD	FF	F338CE	DB38CE
2005-08-30 20:24:34	7F	FF	F74D17	DF4D17

Time	Data Low	Data High	Address Low	Address High
2005-08-30 20:27:27	FF	FB	F407FA	DC07FA
2005-08-30 20:28:19	DF	FF	F7E2F8	DFE2F8
2005-08-30 20:35:32	BF	FF	F7B83F	DFB83F
2005-08-30 21:02:56	FE	FF	F09210	D89210
2005-08-30 21:06:06	F7	FF	F68A87	DE8A87
2005-08-30 21:06:52	EF	FF	F1F41E	D9F41E
2005-08-30 21:07:41	FE	FF	F589A7	DD89A7
2005-08-30 21:08:00	7F	FF	F6EA47	DEEA47
2005-08-30 21:08:08	FF	DF	F78152	DF8152
2005-08-30 21:08:10	FF	EF	F78526	DF8526
2005-08-30 21:08:21	FF	DF	F06D37	D86D37
2005-08-30 21:10:19	EF	FF	F11C0E	D91C0E
2005-08-30 21:10:36	F7	FF	F257C0	DA57C0
2005-08-30 21:10:51	FF	EF	F3731E	DB731E
2005-08-30 21:11:35	7F	FF	F6B584	DEB584
2005-08-30 21:11:41	**FE**	**FF**	**F72F4C**	**DF2F4C**
2005-08-30 21:13:44	FF	DF	F0366F	D8366F
2005-08-30 21:13:59	FF	7F	F147E7	D947E7
2005-08-30 21:14:04	F7	FF	F1AFE2	D9AFE2
2005-08-30 21:16:01	FF	FB	F25451	DA5451
2005-08-30 21:17:09	FF	BF	F74C97	DF4C97
2005-08-30 21:17:57	EF	FF	F2C73E	DAC73E
2005-08-30 21:18:34	FD	FF	F58B6E	DD8B6E
2005-08-30 21:20:18	FF	FD	F52517	DD2517
2005-08-30 21:20:20	FF	FD	F52617	DD2617
2005-08-30 21:20:52	FF	BF	F7A4A4	DFA4A4
2005-08-30 21:28:38	FF	BF	F1EE49	D9EE49
2005-08-30 21:47:41	BF	FF	F6CB8F	DECB8F
2005-08-30 21:53:46	FF	FB	F199CD	D999CD
2005-08-30 22:06:50	FF	FB	F45D22	DC5D22
2005-08-30 22:22:11	FB	FF	F017FB	D817FB
2005-08-30 22:28:11	FD	FF	F29E92	DA9E92
2005-08-30 22:29:59	FF	F7	F28E20	DA8E20
2005-08-30 22:37:00	FF	BF	F17D2B	D97D2B
2005-08-30 22:37:17	FF	FB	F2B4B7	DAB4B7
2005-08-30 22:56:00	BF	FF	F5401A	DD401A
2005-08-30 23:06:09	FF	7F	F1F706	D9F706
2005-08-30 23:11:40	FE	FF	F2601D	DA601D
2005-08-30 23:16:29	FF	DF	F79F98	DF9F98
2005-08-30 23:24:21	FF	DF	F25808	DA5808
2005-08-30 23:47:11	FF	FB	F705D3	DF05D3
2005-08-30 23:50:35	FD	FF	F5DD92	DDDD92
2005-08-30 23:53:12	FF	FD	F187D0	D987D0
2005-08-31 00:11:07	FF	FD	F09B62	D89B62
2005-08-31 00:11:09	FE	FF	F0AC1E	D8AC1E
2005-08-31 00:20:25	EF	FF	F18D64	D98D64
2005-08-31 00:26:10	BF	FF	F2F810	DAF810

Time	Data Low	Data High	Address Low	Address High
2005-08-31 00:30:40	EF	FF	F6CE1E	DECE1E
2005-08-31 00:40:11	FE	FF	F0CC37	D8CC37
2005-08-31 00:43:20	FE	FF	F6B525	DEB525
2005-08-31 01:02:20	FF	FD	F0942E	D8942E
2005-08-31 01:03:02	7F	FF	F3BBCB	DBBBCB
2005-08-31 01:05:33	FF	EF	F6DA6E	DEDA6E
2005-08-31 01:13:23	FF	F7	F15FCA	D95FCA
2005-08-31 01:34:48	FE	FF	F7E4EA	DFE4EA
2005-08-31 01:55:30	FF	EF	F31D25	DB1D25
2005-08-31 02:13:23	FF	7F	F20B69	DA0B69
2005-08-31 02:24:26	FF	FD	F2C920	DAC920
2005-08-31 02:26:03	FD	FF	F1DE50	D9DE50
2005-08-31 02:33:23	FF	7F	F24792	DA4792
2005-08-31 02:44:26	FF	FB	F2FE4B	DAFE4B
2005-08-31 02:48:16	FF	BF	F3C9A1	DBC9A1
2005-08-31 02:50:17	FF	7F	F4BFFC	DCBFFC
2005-08-31 02:53:28	FF	FB	F2D37D	DAD37D
2005-08-31 03:14:13	FF	DF	F658EB	DE58EB
2005-08-31 03:34:01	BF	FF	F5B97F	DDB97F
2005-08-31 03:34:32	EF	FF	F7E911	DFE911
2005-08-31 03:35:15	FF	EF	F3194A	DB194A
2005-08-31 03:38:19	DF	FF	F09941	D89941
2005-08-31 03:41:42	FF	7F	F79EE0	DF9EE0
2005-08-31 03:42:30	FF	BF	F314B4	DB14B4
2005-08-31 03:50:17	FF	DF	F57075	DD7075
2005-08-31 03:51:52	BF	FF	F469C9	DC69C9
2005-08-31 03:54:12	FF	7F	F6AF20	DEAF20
2005-08-31 03:59:04	FB	FF	F424DC	DC24DC
2005-08-31 04:11:31	DF	FF	F317AA	DB17AA
2005-08-31 04:23:33	F7	FF	F02930	D82930
2005-08-31 04:28:19	7F	FF	F500B2	DD00B2
2005-08-31 04:30:02	FD	FF	F4B593	DCB593
2005-08-31 04:30:09	BF	FF	F54385	DD4385
2005-08-31 04:30:17	FF	FB	F5DD04	DDDD04
2005-08-31 04:31:55	FF	FB	F50791	DD0791
2005-08-31 04:37:03	FF	FD	F3B946	DBB946
2005-08-31 04:37:36	FF	F7	F630C7	DE30C7
2005-08-31 04:38:18	F7	FF	F11818	D91818
2005-08-31 04:39:09	FE	FF	F4F66B	DCF66B
2005-08-31 04:40:14	FF	EF	F1C1CB	D9C1CB
2005-08-31 04:41:11	FF	F7	F5F606	DDF606
2005-08-31 04:42:22	FF	EF	F32B64	DB2B64
2005-08-31 04:44:27	FF	EF	F45A11	DC5A11
2005-08-31 05:06:05	EF	FF	F3C393	DBC393
2005-08-31 05:14:25	FB	FF	F08C11	D88C11
2005-08-31 05:15:52	FB	FF	F6F0D1	DEF0D1
2005-08-31 05:28:34	FF	EF	F6E1F2	DEE1F2

Time	Data Low	Data High	Address Low	Address High
2005-08-31 05:35:22	FF	F7	F4EBB8	DCEBB8
2005-08-31 05:39:04	BF	FF	F51D07	DD1D07
2005-08-31 05:56:08	FB	FF	F08360	D88360
2005-08-31 06:05:01	FF	DF	F7B93E	DFB93E
2005-08-31 06:09:57	FD	FF	F574D6	DD74D6
2005-08-31 06:33:11	FF	EF	F3EAD7	DBEAD7
2005-08-31 06:34:17	EF	FF	F0DB0F	D8DB0F
2005-08-31 06:34:51	FF	F7	F34A81	DB4A81
2005-08-31 06:35:05	F7	FF	F44739	DC4739
2005-08-31 06:35:45	BF	FF	F749C2	DF49C2
2005-08-31 06:35:54	F7	FF	F7F1C4	DFF1C4
2005-08-31 06:37:28	FF	FD	F6C8EA	DEC8EA
2005-08-31 06:40:56	FF	BF	F61990	DE1990
2005-08-31 07:16:43	FF	DF	F3F3CD	DBF3CD
2005-08-31 07:23:59	FF	7F	F40423	DC0423
2005-08-31 07:31:15	EF	FF	F4175C	DC175C
2005-08-31 07:37:52	FF	7F	F1408A	D9408A
2005-08-31 07:42:16	FF	7F	F4B931	DCB931
2005-08-31 07:47:31	FF	FD	F3D45E	DBD45E
2005-08-31 07:49:56	FD	FF	F68830	DE8830
2005-08-31 07:58:32	EF	FF	F465AB	DC65AB
2005-08-31 08:00:45	FF	FB	F63910	DE3910
2005-08-31 08:06:14	BF	FF	F667D6	DE67D6
2005-08-31 08:17:27	FD	FF	F7DF31	DFDF31
2005-08-31 08:18:08	FF	FD	F2D30C	DAD30C
2005-08-31 08:25:29	EF	FF	F34217	DB4217
2005-08-31 08:37:58	EF	FF	F257C5	DA57C5
2005-08-31 08:43:09	FD	FF	F12C5E	D92C5E
2005-08-31 08:50:34	FF	FD	F1F99F	D9F99F
2005-08-31 09:01:20	BF	FF	F15ECF	D95ECF
2005-08-31 09:22:04	DF	FF	F4E4F6	DCE4F6
2005-08-31 09:26:38	FF	F7	F0FB6C	D8FB6C
2005-08-31 09:34:00	FF	F7	F17A1A	D97A1A
2005-08-31 09:52:58	BF	FF	F5229D	DD229D
2005-08-31 10:06:28	FF	FB	F0C5C9	D8C5C9
2005-08-31 10:11:31	7F	FF	F70539	DF0539
2005-08-31 10:20:10	FF	F7	F52A1E	DD2A1E
2005-08-31 10:20:55	EF	FF	F07EB1	D87EB1
2005-08-31 10:29:51	7F	FF	F7DADF	DFDADF
2005-08-31 10:35:01	FF	EF	F69F45	DE9F45
2005-08-31 10:43:50	FF	FB	F57070	DD7070
2005-08-31 11:07:57	F7	FF	F7DF10	DFDF10
2005-08-31 11:20:33	FF	FB	F77853	DF7853
2005-08-31 11:21:19	FF	7F	F2E484	DAE484
2005-08-31 11:32:10	DF	FF	F2C12A	DAC12A
2005-08-31 11:37:01	FF	BF	F0296F	D8296F
2005-08-31 11:44:53	F7	FF	F2E18F	DAE18F
2005-08-31 11:45:00	FB	FF	F359BE	DB59BE
2005-08-31 11:58:21	FF	7F	F63032	DE3032
2005-08-31 11:59:49	7F	FF	F4B41D	DCB41D
2005-08-31 12:01:05	FF	EF	F24C47	DA4C47
2005-08-31 12:02:09	FD	FF	F6E7A1	DEE7A1
2005-08-31 12:02:12	FB	FF	F737DF	DF37DF
2005-08-31 12:02:17	BF	FF	F7870D	DF870D
2005-08-31 12:03:21	FE	FF	F4445C	DC445C
2005-08-31 12:03:53	7F	FF	F6A514	DEA514
2005-08-31 12:05:33	DF	FF	F60526	DE0526
2005-08-31 12:05:46	FF	7F	F6EEDF	DEEEDF
2005-08-31 12:05:59	FB	FF	F7E455	DFE455
2005-08-31 12:06:01	FF	7F	F7E93B	DFE93B
2005-08-31 12:06:08	7F	FF	F083E7	D883E7
2005-08-31 12:06:50	F7	FF	F3A2DE	DBA2DE
2005-08-31 12:07:04	FF	DF	F4A64C	DCA64C
2005-08-31 12:08:22	F7	FF	F26CBF	DA6CBF
2005-08-31 12:09:07	FF	FB	F59E90	DD9E90
2005-08-31 12:09:52	7F	FF	F1052E	D9052E
2005-08-31 12:10:46	FF	FD	F4F98B	DCF98B
2005-08-31 12:11:31	FF	7F	F04AEB	D84AEB
2005-08-31 12:11:32	FD	FF	F05150	D85150
2005-08-31 12:11:34	FF	7F	F085EF	D885EF
2005-08-31 12:12:12	FF	EF	F35154	DB5154
2005-08-31 12:13:03	FF	F7	F71069	DF1069
2005-08-31 12:13:05	7F	FF	F72223	DF2223
2005-08-31 12:14:33	FB	FF	F5B2C6	DDB2C6
2005-08-31 12:15:12	FF	7F	F08A8E	D88A8E
2005-08-31 12:15:19	FE	FF	F10729	D90729
2005-08-31 12:15:51	FE	FF	F36370	DB6370
2005-08-31 12:15:53	FB	FF	F389D6	DB89D6
2005-08-31 12:19:16	FD	FF	F2804E	DA804E
2005-08-31 12:20:00	F7	FF	F5B1BB	DDB1BB
2005-08-31 12:20:37	FD	FF	F06C0F	D86C0F
2005-08-31 12:21:48	DF	FF	F5AD63	DDAD63
2005-08-31 12:21:50	FE	FF	F5C32C	DDC32C
2005-08-31 12:22:01	F7	FF	F6A654	DEA654
2005-08-31 12:23:32	F7	FF	F5430F	DD430F
2005-08-31 12:24:32	FE	FF	F1B287	D9B287
2005-08-31 12:26:25	FE	FF	F1FCC9	D9FCC9
2005-08-31 12:27:21	FF	EF	F610F1	DE10F1
2005-08-31 12:29:41	FF	FB	F065DC	D865DC
2005-08-31 12:30:38	FF	DF	F4A0BC	DCA0BC
2005-08-31 12:30:46	BF	FF	F53A9E	DD3A9E
2005-08-31 12:30:48	FF	BF	F5548F	DD548F
2005-08-31 12:31:05	FF	FB	F693D3	DE93D3
2005-08-31 12:31:32	DF	FF	F090C4	D890C4

Time	Data Low	Data High	Address Low	Address High
2005-08-31 12:31:54	FF	BF	F22C52	DA2C52
2005-08-31 12:31:57	F7	FF	F26CA6	DA6CA6
2005-08-31 12:35:06	FF	BF	F04A46	D84A46
2005-08-31 12:35:44	FF	FD	F3165D	DB165D
2005-08-31 12:35:58	DF	FF	F424C8	DC24C8
2005-08-31 12:36:06	FF	FB	F4ADF4	DCADF4
2005-08-31 12:36:42	7F	FF	F756B6	DF56B6
2005-08-31 12:37:21	FE	FF	F244C6	DA44C6
2005-08-31 12:38:52	FE	FF	F0EFB7	D8EFB7
2005-08-31 12:38:58	FF	DF	F1600E	D9600E
2005-08-31 12:39:35	FE	FF	F411B9	DC11B9
2005-08-31 12:41:53	FF	DF	F648BA	DE48BA
2005-08-31 12:42:47	FF	FB	F23981	DA3981
2005-08-31 12:43:14	FB	FF	F4274F	DC274F
2005-08-31 12:44:38	EF	FF	F26771	DA6771
2005-08-31 12:46:30	7F	FF	F28B87	DA8B87
2005-08-31 12:46:47	FB	FF	F3D6C1	DBD6C1
2005-08-31 12:47:23	EF	FF	F67F03	DE7F03
2005-08-31 12:47:31	F7	FF	F7183F	DF183F
2005-08-31 12:48:33	FF	FD	F39342	DB9342
2005-08-31 12:48:47	FF	7F	F4A88A	DCA88A
2005-08-31 12:48:50	FF	F7	F4F385	DCF385
2005-08-31 12:49:39	FE	FF	F075A7	D875A7
2005-08-31 12:49:53	F7	FF	F184BD	D984BD
2005-08-31 12:50:24	FB	FF	F3C67E	DBC67E
2005-08-31 12:50:59	FF	DF	F660D7	DE60D7
2005-08-31 12:52:39	BF	FF	F5ADC2	DDADC2
2005-08-31 12:52:44	FD	FF	F610D2	DE10D2
2005-08-31 12:52:56	DF	FF	F703AE	DF03AE
2005-08-31 12:52:58	DF	FF	F71341	DF1341
2005-08-31 12:53:54	FB	FF	F3443F	DB443F
2005-08-31 14:33:00	DF	FF	F06071	D86071
2005-08-31 14:46:59	FF	F7	F5F802	DDF802
2005-08-31 14:54:40	FF	FB	F7DE26	DFDE26
2005-08-31 15:19:55	FE	FF	F7468D	DF468D
2005-08-31 15:22:40	BF	FF	F3695F	DB695F
2005-08-31 15:44:15	F7	FF	F2A5D9	DAA5D9
2005-08-31 15:46:56	FF	F7	F67755	DE7755
2005-08-31 16:10:36	FE	FF	F6DEB1	DEDEB1
2005-08-31 16:30:55	BF	FF	F081FD	D881FD
2005-08-31 16:35:28	F7	FF	F47C40	DC7C40
2005-08-31 16:52:13	FE	FF	F658F6	DE58F6
2005-08-31 16:57:53	EF	FF	F764E8	DF64E8
2005-08-31 17:14:07	FB	FF	F705E6	DF05E6
2005-08-31 17:18:24	FB	FF	F1DEDC	D9DEDC
2005-08-31 17:18:50	FF	FD	F3DD8B	DBDD8B
2005-08-31 17:20:55	FB	FF	F50C2C	DD0C2C
2005-08-31 17:34:45	DF	FF	F1F69E	D9F69E
2005-08-31 17:35:16	FF	BF	F45625	DC5625
2005-08-31 17:42:14	FF	BF	F30583	DB0583
2005-08-31 17:51:06	FD	FF	F21883	DA1883
2005-08-31 18:02:07	FF	F7	F2BBE0	DABBE0
2005-08-31 18:07:45	FF	F7	F38DB7	DB8DB7
2005-08-31 18:10:32	FF	FD	F7DB77	DFDB77
2005-08-31 18:15:02	FF	F7	F3BA94	DBBA94
2005-08-31 18:26:56	7F	FF	F0359E	D8359E
2005-08-31 18:33:51	FF	EF	F6B2BD	DEB2BD
2005-08-31 18:44:37	FF	FD	F62A2A	DE2A2A
2005-08-31 19:08:43	FF	7F	F072D7	D872D7
2005-08-31 19:09:46	FD	FF	F5202C	DD202C
2005-08-31 19:21:33	FF	FB	F12868	D92868
2005-08-31 20:23:32	FD	FF	F27BBD	DA7BBD
2005-08-31 20:25:17	FE	FF	F237BC	DA37BC
2005-08-31 20:25:24	FF	EF	F2BE4B	DABE4B
2005-08-31 20:26:01	FD	FF	F57304	DD7304
2005-08-31 20:26:10	EF	FF	F614D4	DE14D4
2005-08-31 20:26:31	FF	FB	F7A81B	DFA81B
2005-08-31 20:26:49	FB	FF	F102DF	D902DF
2005-08-31 20:27:22	FF	F7	F365DE	DB65DE
2005-08-31 20:28:28	FF	DF	F036E2	D836E2
2005-08-31 20:28:29	FF	FB	F05D2E	D85D2E
2005-08-31 20:29:12	FF	FD	F36CE8	DB6CE8
2005-08-31 20:47:27	FF	F7	F4E821	DCE821
2005-08-31 21:09:38	FF	F7	F0F287	D8F287
2005-08-31 21:20:15	FE	FF	F03F49	D83F49
2005-08-31 21:21:10	7F	FF	F43F55	DC3F55
2005-08-31 21:34:10	FB	FF	F58C5E	DD8C5E
2005-08-31 21:36:45	FF	FB	F107A3	D907A3
2005-08-31 21:57:34	FB	FF	F4A276	DCA276
2005-08-31 21:57:46	FE	FF	F598F6	DD98F6
2005-08-31 22:00:01	7F	FF	F77B29	DF7B29
2005-08-31 22:09:02	F7	FF	F740D7	DF40D7
2005-08-31 22:17:26	F7	FF	F44888	DC4888
2005-08-31 22:29:37	F7	FF	F1E44E	D9E44E
2005-08-31 22:37:27	EF	FF	F48A3D	DC8A3D
2005-08-31 22:37:49	FF	FD	F6180D	DE180D
2005-08-31 22:48:27	FF	BF	F4F001	DCF001
2005-08-31 22:49:41	EF	FF	F2669E	DA669E
2005-08-31 22:54:04	FF	DF	F5BDE9	DDBDE9
2005-08-31 22:57:08	EF	FF	F32930	DB2930
2005-08-31 22:59:28	FE	FF	F56CD4	DD6CD4
2005-08-31 23:02:14	FF	7F	F18862	D98862
2005-08-31 23:03:38	FF	FB	F7D252	DFD252
2005-09-01 03:03:49	EF	FF	F26889	DA6889

Time	Data Low	Data High	Address Low	Address High
2005-09-01 03:05:16	FF	FD	F0BB4F	D8BB4F
2005-09-01 03:06:02	FF	DF	F41E19	DC1E19
2005-09-01 03:07:03	DF	FF	F08CFA	D88CFA
2005-09-01 03:07:31	7F	FF	F296F1	DA96F1
2005-09-01 03:08:09	FE	FF	F56F89	DD6F89
2005-09-01 03:08:51	BF	FF	F08EC0	D88EC0
2005-09-01 03:09:10	FF	F7	F1E0CF	D9E0CF
2005-09-01 03:10:02	FF	BF	F5C1E0	DDC1E0
2005-09-01 03:11:12	FF	BF	F2D1F2	DAD1F2
2005-09-01 03:12:32	7F	FF	F0C810	D8C810
2005-09-01 03:13:27	FD	FF	F4CFCA	DCCFCA
2005-09-01 03:15:25	BF	FF	F57205	DD7205
2005-09-01 03:15:44	FF	F7	F6F138	DEF138
2005-09-01 05:14:59	FF	BF	F46703	DC6703
2005-09-01 05:19:52	FB	FF	F1EFDC	D9EFDC
2005-09-01 05:39:33	FD	FF	F0A580	D8A580
2005-09-01 05:52:04	FF	F7	F7C47E	DFC47E
2005-09-01 08:43:44	BF	FF	F46BAF	DC6BAF
2005-09-01 08:55:07	FF	FD	F67EDE	DE7EDE
2005-09-01 08:56:20	FF	DF	F3F906	DBF906
2005-09-01 09:00:29	7F	FF	F64398	DE4398
2005-09-01 09:51:58	F7	FF	F121C5	D921C5
2005-09-01 09:52:32	EF	FF	F3889E	DB889E
2005-09-01 10:10:25	FF	FB	F265EC	DA65EC
2005-09-01 10:10:42	FF	DF	F3AC24	DBAC24
2005-09-01 10:11:13	BF	FF	F5E17D	DDE17D
2005-09-01 10:11:19	FF	7F	F64F26	DE4F26
2005-09-01 10:11:24	FF	EF	F6A861	DEA861
2005-09-01 10:12:23	FF	BF	F324B2	DB24B2
2005-09-01 10:12:27	FD	FF	F36789	DB6789
2005-09-01 10:13:16	F7	FF	F6FC12	DEFC12
2005-09-01 10:14:50	BF	FF	F5EC15	DDEC15
2005-09-01 10:17:18	FB	FF	F0B0E8	D8B0E8
2005-09-01 10:17:41	DF	FF	F264A2	DA64A2
2005-09-01 10:35:41	FF	EF	F1CAA9	D9CAA9
2005-09-01 10:38:35	FF	EF	F68299	DE8299
2005-09-01 10:54:33	FF	FD	F4E183	DCE183
2005-09-01 11:09:28	FF	F7	F6A547	DEA547
2005-09-01 11:14:42	BF	FF	F5BF88	DDBF88
2005-09-01 11:17:30	FB	FF	F217C7	DA17C7
2005-09-01 11:28:08	FB	FF	F0FCF1	D8FCF1
2005-09-01 12:01:58	EF	FF	F60484	DE0484
2005-09-01 12:41:23	FF	EF	F3B097	DBB097
2005-09-01 12:43:21	FB	FF	F45D26	DC5D26
2005-09-01 13:02:10	F7	FF	F74A18	DF4A18
2005-09-01 13:02:52	FF	BF	F27098	DA7098
2005-09-01 13:03:32	FF	EF	F5655B	DD655B

Time	Data Low	Data High	Address Low	Address High
2005-09-01 13:11:22	FF	FD	F7CCB3	DFCCB3
2005-09-01 13:14:13	DF	FF	F4791D	DC791D
2005-09-01 13:45:58	FF	7F	F06604	D86604
2005-09-01 14:21:14	BF	FF	F3DEC7	DBDEC7
2005-09-01 14:37:41	7F	FF	F454B7	DC54B7
2005-09-01 15:06:27	FF	F7	F31CBF	DB1CBF
2005-09-01 15:12:16	FF	FB	F4A9E9	DCA9E9
2005-09-01 15:13:51	DF	FF	F3A066	DBA066
2005-09-01 15:15:08	FB	FF	F142D5	D942D5
2005-09-01 15:23:28	FE	FF	F5FDAA	DDFDAA
2005-09-01 15:24:59	FF	DF	F4D092	DCD092
2005-09-01 15:42:15	FB	FF	F0E7E4	D8E7E4
2005-09-01 15:57:29	DF	FF	F3F1F1	DBF1F1
2005-09-01 16:17:22	FE	FF	F388D8	DB88D8
2005-09-01 16:21:53	FF	FD	F7743E	DF743E
2005-09-01 16:26:11	FF	F7	F260F0	DA60F0
2005-09-01 16:26:51	BF	FF	F561C6	DD61C6
2005-09-01 16:26:52	FF	FB	F56667	DD6667
2005-09-01 16:33:16	FF	DF	F1A4C8	D9A4C8
2005-09-01 16:41:54	BF	FF	F7B0C0	DFB0C0
2005-09-01 16:47:50	DF	FF	F1D3AB	D9D3AB
2005-09-01 16:57:11	7F	FF	F31226	DB1226
2005-09-01 17:20:18	FF	7F	F0F2B2	D8F2B2
2005-09-01 17:20:29	7F	FF	F1ADBF	D9ADBF
2005-09-01 17:21:18	FE	FF	F54CCA	DD4CCA
2005-09-01 17:21:48	FF	EF	F78615	DF8615
2005-09-01 17:22:19	FE	FF	F1DBA8	D9DBA8
2005-09-01 17:22:30	FF	FB	F28B85	DA8B85
2005-09-01 17:22:31	EF	FF	F2ADB5	DAADB5
2005-09-01 17:22:43	FB	FF	F39358	DB9358
2005-09-01 17:25:10	FF	F7	F64C41	DE4C41
2005-09-01 18:09:15	FF	BF	F0B656	D8B656
2005-09-01 18:22:24	FF	EF	F297D5	DA97D5
2005-09-01 18:33:08	FF	EF	F1E722	D9E722
2005-09-01 18:36:25	FF	DF	F05D51	D85D51
2005-09-01 18:39:41	FD	FF	F6D7E8	DED7E8
2005-09-01 18:43:45	FD	FF	F0BD68	D8BD68
2005-09-01 18:47:16	FF	FD	F04588	D84588
2005-09-01 18:51:10	FE	FF	F1718A	D9718A
2005-09-01 18:51:34	FB	FF	F325E6	DB25E6
2005-09-01 18:58:52	DF	FF	F341EF	DB41EF
2005-09-01 19:14:28	FF	EF	F7F461	DFF461
2005-09-01 19:14:34	FF	7F	F08C46	D88C46
2005-09-01 19:43:15	FD	FF	F70606	DF0606
2005-09-01 19:46:17	FF	FB	F443C5	DC43C5
2005-09-01 19:58:22	FF	F7	F17C73	D97C73
2005-09-01 19:59:02	DF	FF	F47AD6	DC7AD6

Time	Data Low	Data High	Address Low	Address High
2005-09-01 20:38:10	FF	F7	F0E87F	D8E87F
2005-09-01 20:39:53	F7	FF	F08751	D88751
2005-09-01 21:02:09	FF	FD	F29AC4	DA9AC4
2005-09-01 21:13:27	EF	FF	F4727C	DC727C
2005-09-01 21:26:43	7F	FF	F6FB77	DEFB77
2005-09-01 21:32:54	FB	FF	F23E4E	DA3E4E
2005-09-01 21:33:24	7F	FF	F46836	DC6836
2005-09-01 21:33:35	FF	EF	F52B48	DD2B48
2005-09-01 21:50:37	FB	FF	F04D39	D84D39
2005-09-01 22:04:45	FF	FB	F6A5E1	DEA5E1
2005-09-01 22:04:58	DF	FF	F790A7	DF90A7
2005-09-01 22:05:00	FF	BF	F79B12	DF9B12
2005-09-01 22:11:45	EF	FF	F57D84	DD7D84
2005-09-01 22:15:26	DF	FF	F5B0F5	DDB0F5
2005-09-01 22:16:53	FF	F7	F418AB	DC18AB
2005-09-01 22:21:38	FE	FF	F0F372	D8F372
2005-09-01 22:26:23	FD	FF	F5DF87	DDDF87
2005-09-01 22:27:36	FF	7F	F32134	DB2134
2005-09-01 22:35:55	F7	FF	F7E124	DFE124
2005-09-01 22:41:39	FF	EF	F1396B	D9396B
2005-09-01 23:20:21	FF	7F	F3A834	DBA834
2005-09-01 23:30:04	EF	FF	F68B2E	DE8B2E
2005-09-01 23:39:27	F7	FF	F7E441	DFE441
2005-09-01 23:53:46	FD	FF	F6F693	DEF693
2005-09-01 23:58:15	FE	FF	F2CAD6	DACAD6
2005-09-02 00:16:42	FE	FF	F40F2D	DC0F2D
2005-09-02 00:23:38	7F	FF	F28C61	DA8C61
2005-09-02 00:25:02	FD	FF	F0C857	D8C857
2005-09-02 00:25:07	FF	EF	F11AE1	D91AE1
2005-09-02 00:25:52	FB	FF	F46A7F	DC6A7F
2005-09-02 00:26:33	FF	7F	F76898	DF6898
2005-09-02 00:26:50	FF	FB	F0B996	D8B996
2005-09-02 00:28:08	EF	FF	F66EB8	DE6EB8
2005-09-02 00:28:24	BF	FF	F790A9	DF90A9
2005-09-02 00:28:30	BF	FF	F014B3	D814B3
2005-09-02 00:30:06	FF	DF	F70141	DF0141
2005-09-02 00:30:08	FF	BF	F73BA3	DF3BA3
2005-09-02 00:30:31	FD	FF	F0F02B	D8F02B
2005-09-02 00:44:12	FF	BF	F551A9	DD51A9
2005-09-02 00:51:33	7F	FF	F5AAF0	DDAAF0
2005-09-02 00:51:41	FF	FD	F625A7	DE25A7
2005-09-02 00:52:52	FB	FF	F362DA	DB62DA
2005-09-02 00:53:03	F7	FF	F44214	DC4214
2005-09-02 00:53:11	FF	FD	F4E39A	DCE39A
2005-09-02 01:04:35	BF	FF	F706E3	DF06E3
2005-09-02 01:10:35	7F	FF	F18255	D98255
2005-09-02 01:16:09	BF	FF	F1F646	D9F646
2005-09-02 01:39:26	FF	F7	F09DEB	D89DEB
2005-09-02 01:59:42	FF	DF	F1FB8A	D9FB8A
2005-09-02 02:16:11	BF	FF	F2B74C	DAB74C
2005-09-02 02:18:54	FB	FF	F6A82D	DEA82D
2005-09-02 02:33:18	FF	FB	F6124A	DE124A
2005-09-02 02:39:28	BF	FF	F14550	D94550
2005-09-02 02:42:50	EF	FF	F006BD	D806BD
2005-09-02 03:59:00	DF	FF	F7BCF5	DFBCF5
2005-09-02 07:09:06	F7	FF	F57501	DD7501
2005-09-02 07:09:56	FD	FF	F130F4	D930F4
2005-09-02 07:10:04	EF	FF	F1E1D4	D9E1D4
2005-09-02 07:11:03	FF	FD	F61F2D	DE1F2D
2005-09-02 07:12:03	FF	FB	F281D4	DA81D4
2005-09-02 07:12:45	FD	FF	F59C23	DD9C23
2005-09-02 07:12:59	DF	FF	F6A982	DEA982
2005-09-02 07:13:18	FD	FF	F0022E	D8022E
2005-09-02 07:17:28	FF	BF	F27873	DA7873
2005-09-02 07:31:13	FE	FF	F6F239	DEF239
2005-09-02 07:43:24	BF	FF	F4BFEA	DCBFEA
2005-09-02 07:44:55	DF	FF	F36929	DB6929
2005-09-02 07:56:36	FE	FF	F6CE46	DECE46
2005-09-02 08:07:52	EF	FF	F087D4	D887D4
2005-09-02 08:12:45	FF	BF	F60A2A	DE0A2A
2005-09-02 08:13:43	BF	FF	F266F5	DA66F5
2005-09-02 08:17:59	FF	F7	F5278C	DD278C
2005-09-02 08:20:01	FF	DF	F61909	DE1909
2005-09-02 08:21:53	F7	FF	F6597C	DE597C
2005-09-02 08:27:20	EF	FF	F65E5D	DE5E5D
2005-09-02 08:53:08	EF	FF	F7FC3B	DFFC3B
2005-09-02 08:58:10	FF	EF	F6268A	DE268A
2005-09-02 09:06:00	FF	DF	F0CE51	D8CE51
2005-09-02 09:07:54	F7	FF	F119A1	D919A1
2005-09-02 09:09:23	F7	FF	F79D4D	DF9D4D
2005-09-02 09:25:27	FF	FB	F6639A	DE639A