Insper

Desenvolvimento de conformance tester para TAGs com comunicação baseada no protocolo EPC-GEN2 RFID

Alexandre Almeida Edington (alexandreae@al.insper.edu.br)
Bruno Signorelli Domingues (brunosd1@al.insper.edu.br)
Lucas Leal Vale (lucaslv1@al.insper.edu.br)
Rafael dos Santos (rafaels6@al.insper.edu.br)

Trabalho de Conclusão de Curso

Relatório
Versão Final
do Projeto Final de Engenharia

São Paulo-SP
Novembro 2021

Alexandre Almeida Edington
Bruno Signorelli Domingues
Lucas Leal Vale
Rafael dos Santos

Desenvolvimento de conformance tester para TAGs com comunicação baseada no protocolo EPC-GEN2 UHF RFID

Relatório Final do Projeto Final de Engenharia

Relatório apresentado ao curso de Engenharia, como requisito para o Trabalho de Conclusão de Curso.

Professor Orientador: Prof. Rafael Corsi Ferrão

Mentor na Empresa: W. Shepherd Pitts, PhD

Coordenador TCC/PFE: Prof. Dr. Luciano Pereira Soares

São Paulo - SP
Novembro 2021

Insper: Open-source Conformance Tester for Tags EPC-GEN2 UHF RFID

None

None

Table of contents

1. Conformance Tester for Tags EPC-GEN2 UHF RFID

Connection between two DE-10 Standard boards

- **Students:**
  - Alexandre Almeida Edington (alexandreedington@gmail.com)
  - Bruno Signorelli Domingues (brunao.signorelli@hotmail.com.br)
  - Lucas Leal Vale (lucaslealvale01@gmail.com)
  - Rafael Dos Santos (rafael_1999_@hotmail.com.br)
- **Course:** Computer Engineering
- **Semester:** 8th
- **Professor Advisor:** Rafael Corsi Ferrão (rafael.corsi@insper.edu.br)
- **Mentor:** Wallace Shepherd Pitts (wspitts2@ncsu.edu)

- **Year:** 2021
- **Repository URL:** https://github.com/pfeinsper/21b-indago-rfid-conformance-tester

1.1 About

This project's objective is to develop an equipment capable of performing a series of tests on RFID tags, based on the communication protocol "EPC-GEN2 UHF RFID" [1]. Such equipment will be of value in simplifying the development of new tags that need to conform to the established protocol, being able to assert whether the tag satisfies the requirements of the aforementioned protocol, and also whether the tag itself is working as intended.

1.1.1 Insper

This project was developed by four computer engineering students at "Insper Instituto de Ensino e Pesquisa" [2], who worked together with "Indago Devices Inc." [3]. As part of the engineering completion of course work, the students must choose between a variety of projects proposed by different companies, choosing one that attracts their interest. Once the groups and projects are settled, the students work together with a representative of the company who proposed the project in order to find and implement a solution. For each group of students there is also a professor advisor which aids the group with matters of communication, organization, meetings and project/report feedbacks.

1.1.2 Indago Devices Inc.

Indago Devices Inc. is a startup that has its headquarters in the city of Birmingham, Alabama, US, and works in the field of development and study of electronics. Despite having few employees, it seeks to innovate in the electronics market, specifically in the development of systems that communicate through RFID.

They had already been conversing with Insper in the first half of 2021, with another group of students who planned to do a similar project. On the second half of 2021, they decided to follow up with a conformance tester to assist the development of RFID tags. One of the driving points of the project is that the currently existing solutions are proprietary, and there is no open-source alternative available for the RFID community. Hence, they opted to make one, so it could positively impact not only their company, but also the worldwide RFID development community.

1.2 Project Overview

The main objective of this project is to develop and assemble a conformance tester for RFID tags where a microcontroller will be implemented together with an IP core for communication with the DUT (device under testing). This device then shall be able to run a series of tests as a reader interacting with a tag through the EPC-GEN2 protocol, analyzing if the tag works as intended and complies with the requirements of the protocol.

The tests will be implemented using the C programming language, allowing for a variety of tests to be created, each test targeting different aspects of the tag's processing, independently evaluating most of them. Also, the tests are customizable, being possible to edit or develop new ones, should the user need it.

It is important to highlight that this project does not make use of RFID communication, nor does it intend to test whether the tag is able to communicate through it. Given the complexity of communicating through radio waves, the group and the professor agreed to not cover those points in this project. Therefore, the device, tag and computer shall be connected by cables.

For more information on the architecture, the RTL diagram generated by Quartus can be found here.

1.2.1 Protocol EPC-GEN2 UHF RFID

Documentation avaliable on: https://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf

The main purpose of the protocol is to allow two independently obtained pieces of hardware (that adhere to the protocol) to communicate flawlessly. To achieve this, it specifies how physical and logical interactions should take place, as well as the possible commands between reader and tag.

To claim compliance with the protocol, a reader (also called interrogator) must meet all required specifications, having implemented all mandatory commands, be able to encode, send, receive, and decode data so that it can communicate with a tag, as well as comply with all local government radio regulations. Optionally, it is allowed to implement any number of optional commands defined in the protocol and any other private commands that do not conflict with any of the mandatory ones. Finally, a reader must not require a tag to be able to process any command that is not specified as mandatory in the protocol.

To claim compliance with the protocol, a tag must meet all required specifications, having implemented all mandatory commands, be able to modulate a response signal after receiving a command from a reader, and comply with all local government radio regulations. Optionally, it is allowed to implement any number of optional commands defined in the protocol and any other private commands that do not conflict with any of the mandatory ones. Finally, the tag must not require a reader to be able to process any optional command from the protocol and is not allowed to modulate a response signal unless it has been commanded to do so by a reader using the commands present in the protocol.

The EPC-GEN2 UHF RFID allow four types of commands in its documentation:

1. mandatory;
2. optional;
3. proprietary;
4. custom.

All commands defined in the protocol are either mandatory or optional. Proprietary and custom commands are manufacturer-defined. Mandatory commands should be be supported by all tags and readers that claim compliance to the protocol.

Optional commands may or may not be supported by tags or readers. If any implements optional commands, then it shall do so in the manner specified in the protocol.

Proprietary commands may be enabled in conformance with the protocol but are not specified in it. All proprietary commands shall be capable of being permanently disabled. Proprietary commands are intended for manufacturing purposes and shall not be used in field-deployed RFID systems.

Custom commands may be enabled in conformance with the protocol but are not specified in it. A reader shall issue a custom command only after singulating a tag and reading (or having prior knowledge of) the tag manufacturer's identification in the tag's TID memory. A reader shall use a custom command only in accordance with the specifications of the tag manufacturer identified in the TID. A custom command shall not solely duplicate the functionality of any mandatory or optional command defined in the protocol by a different method.

Mandatory Commands

- `Select` selects the population of tags that will be communicated with. The set can be defined by intersection, union or negation of tags;
- `Query / Query Adjust / Query Rep` starts a communication round between the tags and reader, deciding which tag will participate in the round and sending the Q value for such. `Query Adjust` can adjust que Q value for the tag. `Query Rep` decreases the value of Q stored within the tag's memory by 1;
- `ACK / NAK` is sent to the tag with the same value sent by the tag when returning to the `Query` command. It signifies the reader recognized the tag's response. `NAK` changes the state of the tags involved in the round to `arbitrate` , in which they remain as stand-by;
- `Req_RN` requests a new random number (RN16), sending the previous one as authentication;
- `Read / Write` requests the reading of information within a specified address in the tag's memory bank. `Write` sends information to be written in that address instead;
- `Kill / Lock` sets the tag as unusable. It is a way to end the communication so that the tag no longer responds. `Lock` can lock or unlock portions of the tag's memory bank for `Write` access.

Handshake

The diagram below can be found in annex E of the EPC-GEN2 documentation and represents the handshake between reader and tag.

Handshake diagram *EPC UHF Gen2 Air Interface Protocol, p 138*

The reader sends a `Query` (1), to start an inventory round with the tag. Upon recognizing the inventory round, the tag checks whether to respond, and responds with a 16-bit random number `RN16` (2). To establish the communication as successful, the reader sends the `ACK` (3) containing the same RN16. Having received and validated the confirmation, the tag responds with `PC/XPC, EPC` (4). The reader then send a `Req_RN` (5), again with the old RN16, requesting a new RN16 to continue the communication. If the tag again validates the RN16, it responds with the `handle` (6), a new RN16. Once the reader receives the `handle`, the handshake is effectively over and the `handle` will be used as authentication for all communication from that point forwards. Every `command` (7) will be sent together with the `handle` and tag will always verify the `handle` before responding (8).

Tari

The reference time interval for a package in the reader to be sent to the tag. The acronym TARI derives from Type A Reference Interval.

According to the EPC-GEN2 protocol, section 6.3.1.2.4, p 27:

> *Interrogators shall communicate using Tari values in the range of 6.25µs to 25µs. Interrogator compliance shall be evaluated using at least one Tari value between 6.25µs and 25µs with at least one value of the parameter x. The tolerance on all parameters specified in units of Tari shall be +/– 1%. The choice of Tari value and x shall be in accordance with local radio regulations.*

This time reference dictates the whole communication and guarantees its conformance.

1.2.2 State-of-the-Art Review

The market currently has a variety solutions regarding RFID technology. Among the options currently available, proprietary equipment and products dominate the market, as they are developed by well-established companies. A example of these companies is CISC semiconductor [4], whcih specializes in RFID and NFC services, and working both in the production of laboratory equipment and product testers for the market. Another company that is worth mentioning is HID global [5], which has several solutions for RFID tags end operates worldwide under sales and distribution of these products.

There are, however, other solutions present in the market, such as open-source solutions. As proprietary products are expensive and not easily customizable/reproduceable, some users choose to develop their own version of those products, leaving them open for others to use and improve. The use of open-source helps to develop a highly customizable and reproduceable product, as any user can download the project's files and make their own changes to better suit their need. Another benefit of open-source is the collaboration aspect, where users around the world can suggest changes or improvements, as well as implement them to improve the overall product quality.

An example of an open-source product is the WISP5 [6] tag, initially developed at the University of Washington [7]. The WISP is a battery-free platform with a software-defined implementation of a passive RFID tag, that can communicate with commercial-off-the-shelf RFID readers and is powered by the carrier signal emitted by the reader. It is also built from low-cost components commonly found in hardware stores, allowing WISP users to fabricate their own tags if desired.

Another open-source product is the S.U.R.F.E.R. (Software-defined UHF RFID Flexible Economical Reader) [8], an RFID reader. It operates with the same technology as the WISP5 tag, enabling readings up to 60 feet (20 meters) away. Due to it being software-defined, the reader is highly versatile, as the user can input the specifications of the desired tag into the software. It also has a relatively simple structure to find, which result in a low-cost product.

Open-source products bring a series of benefits to the users, such as:

- reduced hardware and software costs, due to the products being intentionally built to be easily accessible;
- simple licensing management because they often are free to use and impose no restrictions at all;
- abundant support, as there are many companies that develop open-source products and offer both free and varied levels of paid support.

Given these advantages, Indago Devices opted for a completely open-source product as well. In a meeting with our mentor Wallace Shepherd Pitts, he mentioned he had previously researched and studied some of the options currently available, but none of them had the features he had in mind, mostly because they offered little room for customization regarding the tag testing.

The direct competitors of our project would be the previously mentioned products. However, as the team is aiming for an open-source solution, the project may attract users interested in a more accessible/customizable product.

Another point mentioned by the mentor is that he also intends to use the project as study material for students at the University of North Carolina [9], which consequently opens up possibilities for further expansion of the product.

1.2.3 Methodology

During the first weeks of the project, the group settled on definitions and agreements on what would be the methodology used throughout the semester, as well as the different tools and softwares that would be used.

The platform GitHub[10] was chosen as the method for sharing the code between the group members and the professor, as it can store many important files other than code files, such as diagrams and images the group would produce for the project. Another feature often used by the group is the creation of issues, which can help define and order the group's next tasks and assign members to complete them.

The day-to-day communication between the members were done through Discord[11], and meetings with Indago's representative/mentor or Insper's coordinators though Microsoft Teams[12]. As meetings with the mentor were infrequent, taking place every fortnight, the group usually kept a list of questions and issues about the EPC-GEN2 protocol and the project in general so that the representative could provide some support.

Documents and reports were produced and stored in Google Drive[13], so that multiple members could work on them simultaneously, and also be accessed by the professor to provide insights and feedback. It also served as another backup storage to the Github repository, in case any problems occurred.

The programming languages VHDL and C were used throughout the project, and the Intel® Quartus® Prime FPGA Design Software[14] was used in conjunction with the Nios® II Software Build Tools for Eclipse[15] plugin, which supports simulations and tests that assisted in the development process, as well as the ModelSim*-Intel® FPGA Edition Software[16], widely used for testing VHDL component codes. As the client specified that he wanted the project to be open-source, all code, reports and images relevant to this will also be available on the project's public GitHub repository.

Considering the project consists of the creation of a conformance tester for the EPC-GEN2 UHF RFID protocol, its documentation was widely used, researched, and discussed by all members of the group during the project, focusing mainly on the communication sections between the reader and tag, as well as encoding data, and mandatory commands for protocol standards.

Given the project's open-source nature and its public availability on GitHub, it was decided that the group would also provide a documentation to the whole project, which was later decided would be done using GitHub Pages[17]. Inside, the group would give an in-depth description of all components, a tutorial on how to clone, run, utilize and modify this project, and explanations for the hardware, the IP core[18] interface and the firmware.

To further help with code documentation, the group used the Doxygen[19] application, which was incorporated into the already existing GitHub Pages documentation. This application generates a page of every VHDL file, giving a brief explanation of the purpose of that component, as well as explain every aspect of the entity, including generics, ports in and outs.

1.2.4 Environment Tools

These are the tools used by the team in order to meet up, develop and design the project.

- **Development:**
  - Intel® Quartus® Prime FPGA Design Software - v18.1.0.625
  - Nios® II Software Build Tools for Eclipse
  - Intel® FPGA Simulation - ModelSim
  - Logic 2 Software[20] - v2.3.39-master
- **Design:**
  - Google Drive
  - Discord
  - Microsoft Teams
  - Excalidraw[21]

1. EPC UHF Gen2 Air Interface Protocol. https://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf Accessed on: 16/08/2021. ↩
2. Insper Instituto de Ensino e Pesquisa. https://www.insper.edu.br/ Accessed on: 16/08/2021. ↩
3. Indago Devices Inc. https://indagodevices.com Accessed on: 16/08/2021. ↩
4. Cisc Semiconductors. https://www.cisc.at/ Accessed on: 20/09/2021. ↩
5. HID Global. https://www.hidglobal.com/products/rfid-tags Accessed on: 20/09/2021. ↩
6. WISP5 Wiki. https://sites.google.com/uw.edu/WISP-wiki/home Accessed on: 20/09/2021. ↩
7. University of Washington. https://www.washington.edu/ Accessed on: 20/09/2021. ↩
8. S.U.R.F.E.R. reader. https://openrfidreader.net/ Accessed on: 20/09/2021. ↩
9. University of North Carolina. https://www.uncg.edu/ Accessed on: 20/09/2021. ↩
10. GitHub website. https://github.com/ Accessed on: 16/08/2021. ↩
11. Discord. https://discord.com/ Accessed on: 16/08/2021. ↩
12. Microsoft Teams. https://www.microsoft.com/pt-br/microsoft-teams/group-chat-software/ Accessed on: 16/08/2021. ↩
13. Google Drive. https://www.google.com/intl/pt-BR/drive/ Accessed on: 16/08/2021. ↩
14. Intel® Quartus® Prime Software Suite. https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/overview.html Accessed on: 16/08/2021. ↩
15. Nios® II Software Build Tools for Eclipse. https://www.intel.com/content/www/us/en/support/programmable/support-resources/intellectual-property/ips-nios2-ide.html Accessed on: 16/08/2021. ↩
16. ModelSim*-Intel® FPGA Edition Software. https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/model-sim.html Accessed on: 16/08/2021. ↩
17. GitHub Pages. https://pages.github.com/ Accessed on: 16/08/2021. ↩
18. IP Core. https://www.intel.com/content/www/us/en/products/programmable/intellectual-property.html Accessed on: 23/08/2021. ↩
19. Doxygen documentation. https://www.doxygen.nl/index.html Accessed on: 01/10/2021. ↩
20. Logic 2 Software. https://www.saleae.com/downloads/ Accessed on: 01/10/2021. ↩
21. Excalidraw. https://www.excalidraw.com/ Accessed on: 01/10/2021. ↩

2. Getting started

2.1 Tools

To be able to use the Conformance Tester for Tags EPC-GEN2 UHF RFID, you'll need the following tools:

- **Hardware:**
  - DE10-Standard + accessories
  - jumpers
- **Software:**
  - Quartus Prime Lite Edition 18.1
  - Cyclone V device support
  - ModelSim-Intel FPGA Edition
  - git

You can find the Quartus and ModelSim software downloads here. The user manual for the DE-10 Standard board can be found here.

2.2 Cloning

To clone the project, run the following command on your preferred terminal.

```
git clone <https://github.com/pfeinsper/21b-indago-rfid-conformance-tester.git>
```

The repository is now cloned, and you can start testing/running the project.

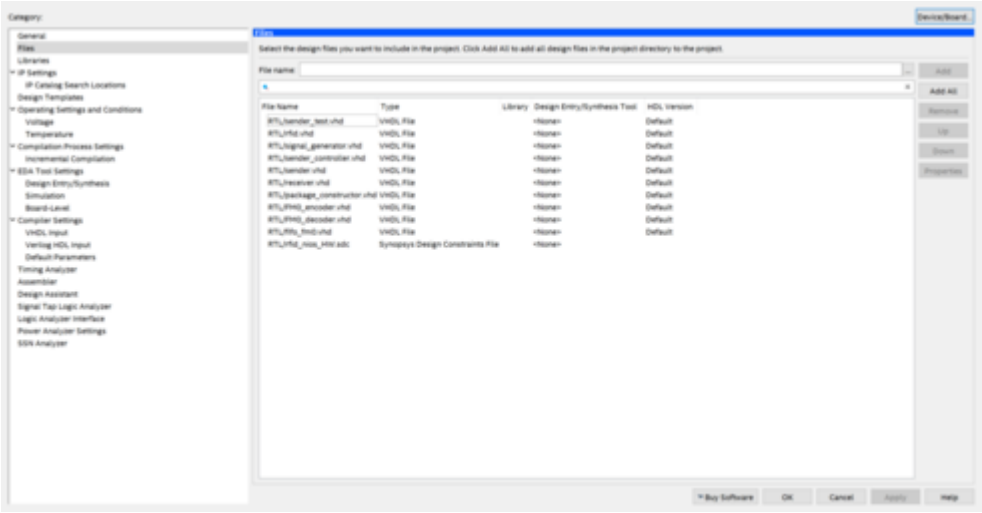
2.3 Testing / Running

First, launch Quartus Prime. After launching, click on `File` `Open Project`. A window will pop up, and you need to choose the `rfid-conformance-tester.qpf` file, located in the `fpga` folder of the repository. The video below shows how to do so.

Once the project has been opened, you can work on it; however, the steps to run the project depend on whether you want to simulate using ModelSim or whether you want to launch on the DE-10 Standard board.

2.3.1 ModelSim guide

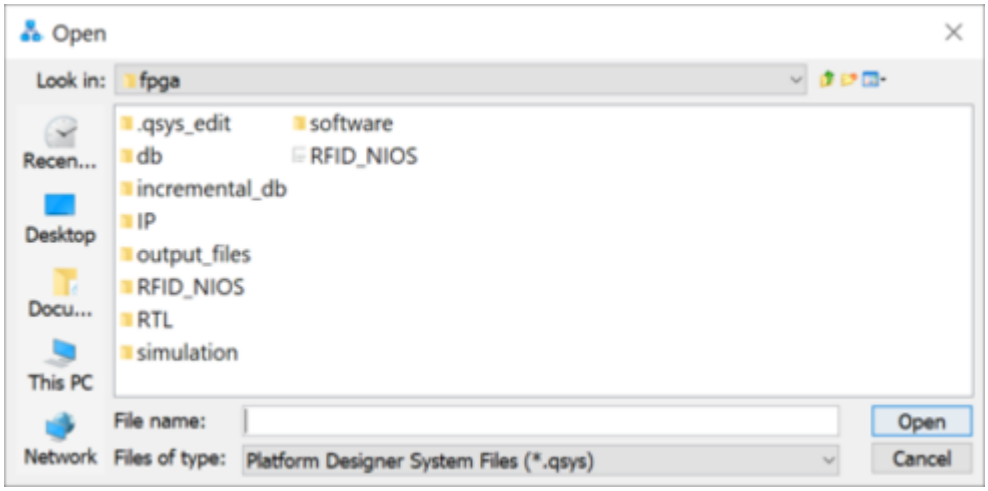
To test and simulate the project on ModelSim, the first step is to check the files present in the project. Click on `Project` `Add/Remove Files in Project`. It should look like the picture below.



ModelSim project files

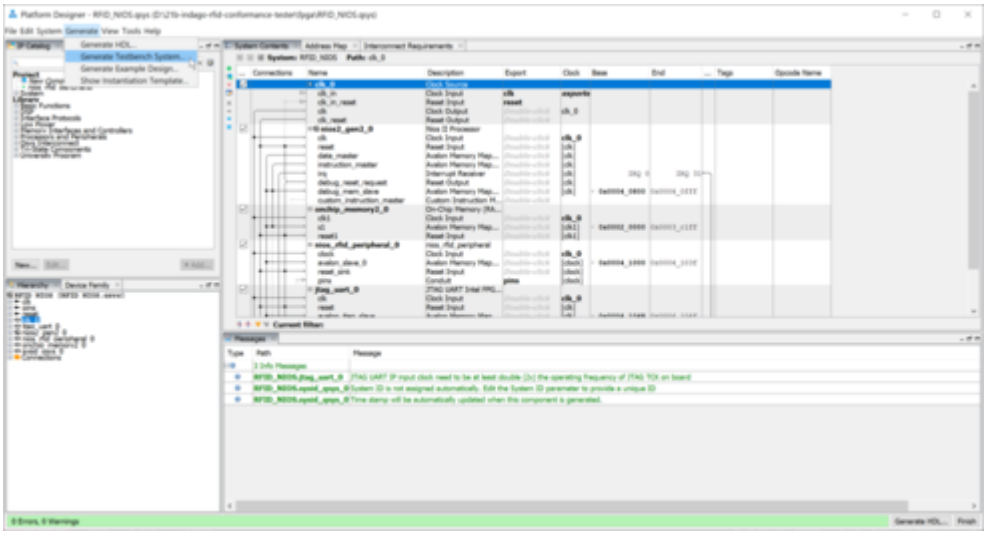
After checking the files, go to `Project Navigator` on the left panel, click on the dropdown menu that says `Hierarchy`, then click on `Files`. After that, right-click on the `RTL/rfid.vhd` file, and click on the option that says `Set as Top-Level Entity`. Now you can click on the blue play button next to the stop button to start the compilation.

Once the project is compiled, click on `Tools` `Platform Designer`. A new window should open. Find the `RFID_NIOS.qsys` file, and open it.



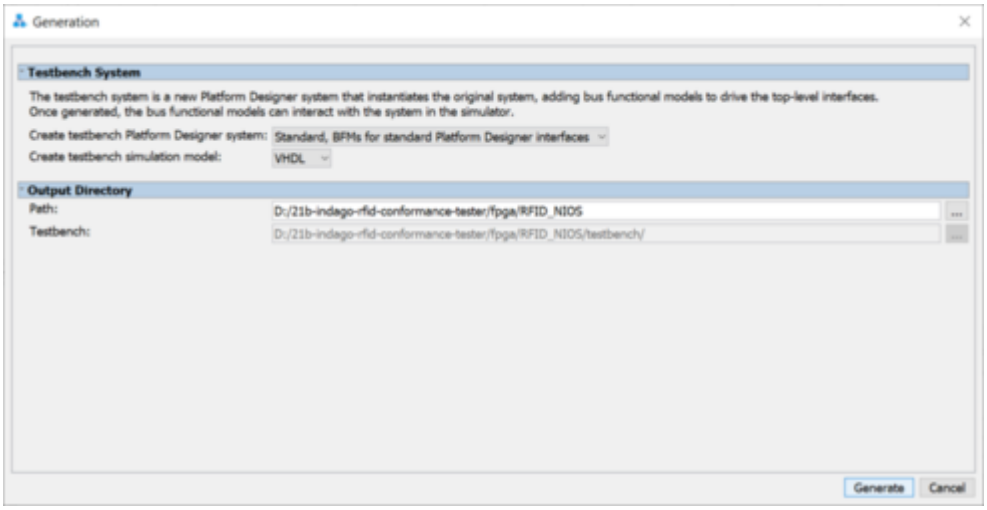
Platform Designer qsys file window prompt

Once you've opened the file, it should show the project's design. On this window, click on `Generate` `Generate Testbench System...`.



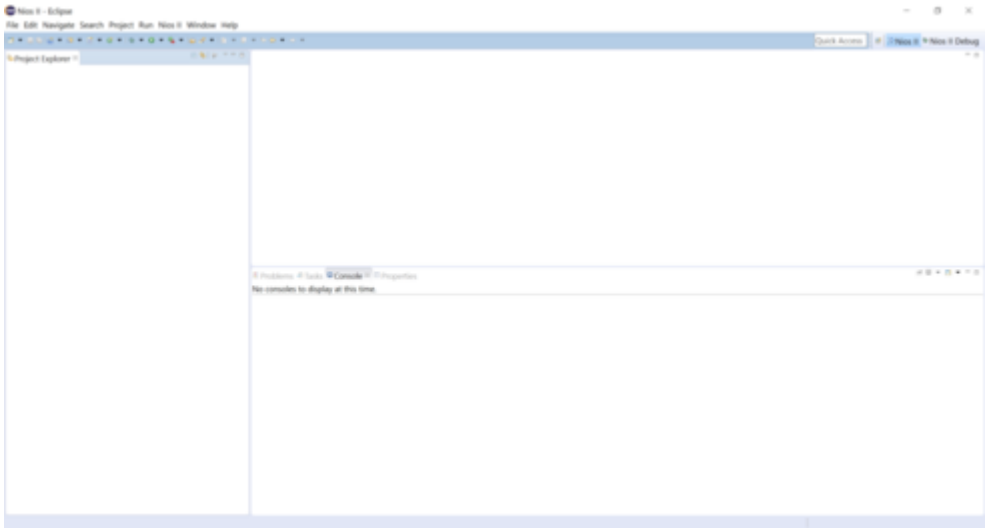
Platform Designer schematic with peripherals

Make sure the settings are as follows, and click on `Generate` on the bottom.



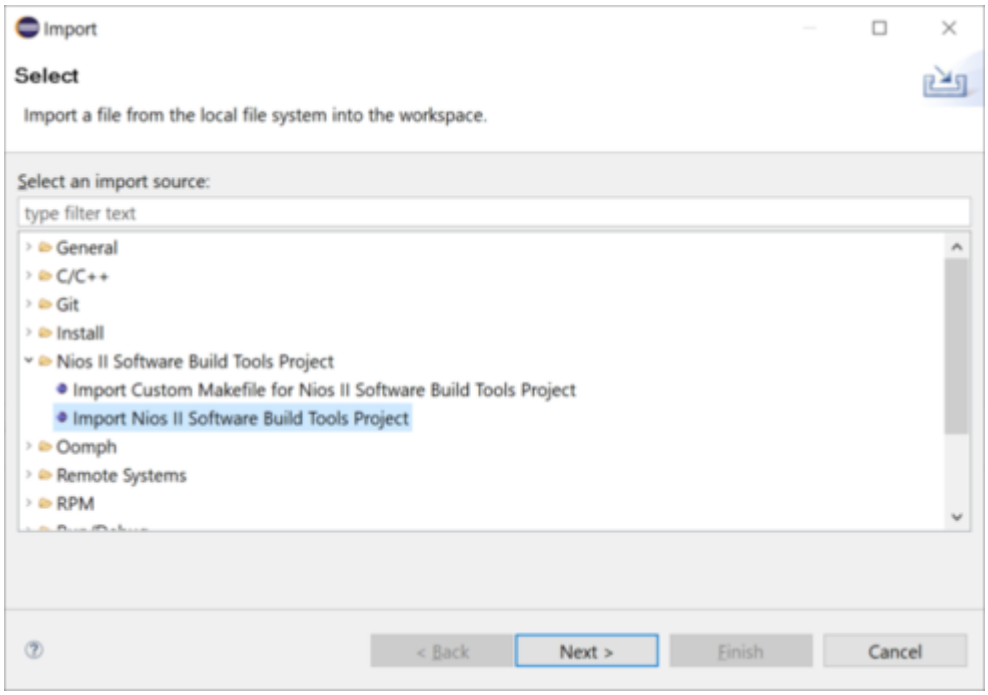
Generate testbench window prompt with settings

Next, you wanna open up the `Nios® II Software Build Tools for Eclipse`. If it's your first time opening this tool, the Eclipse window will ask you to choose a workspace. Now the Eclipse window should be open with a blank workspace like the one below.



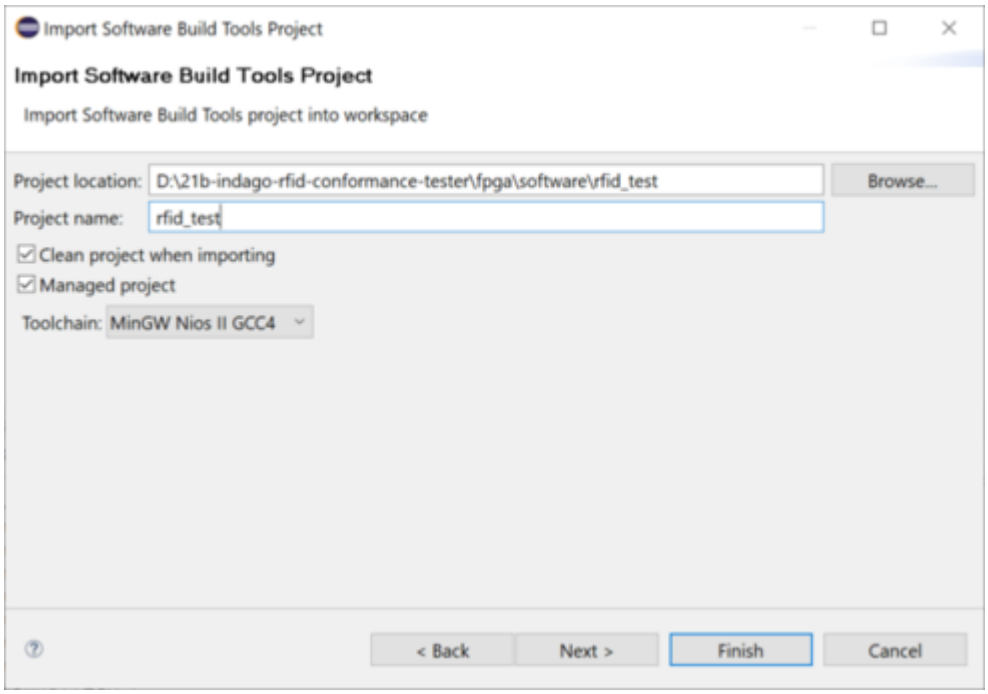
Eclipse workspace

The next step is to import our projects into Eclipse. Go to `File` `Import`, and it should show a window like the one below. Click on `Nios II Software Build Tools Project` `Import Nios II Software Build Tools Project`, then click `Next`.



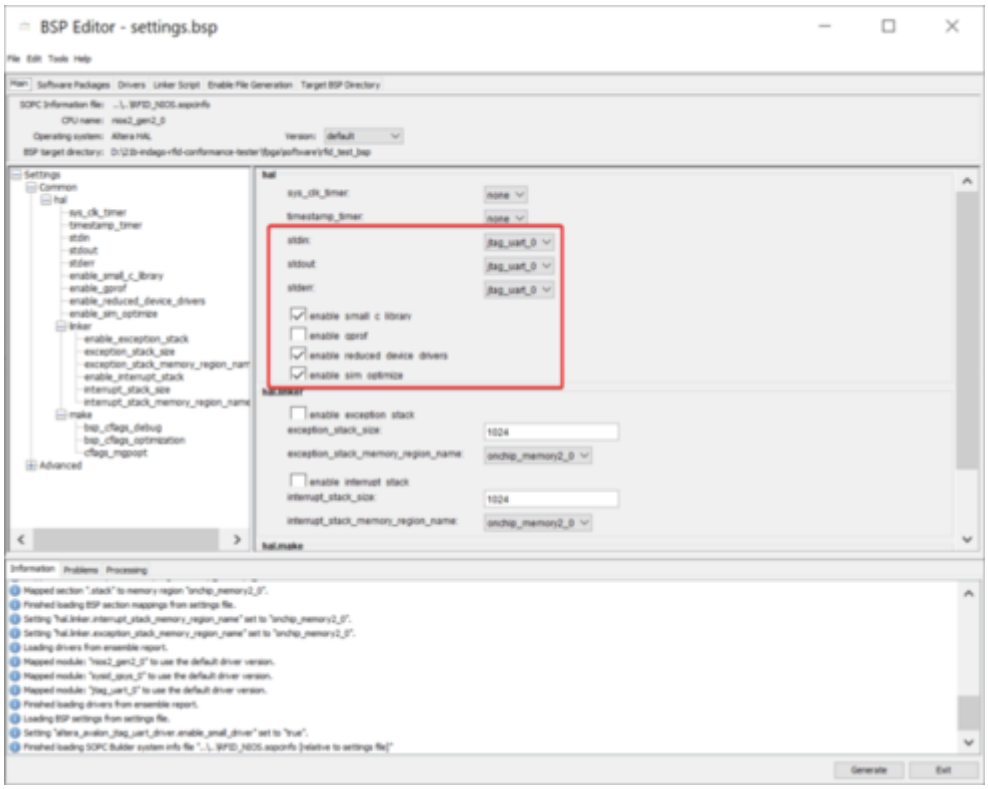
Import project window prompt

Now you need to click on the `Browse` button and look for the project. The projects are located at `21b-indago-rfid-conformance-tester/fpga/software`. First, import the `rfid_test` project, then click `Finish`, and make sure that the `Clean project when importing` box is checked. Do the same steps for the `rfid_test_bsp` project.



Import project second window prompt

After both projects have been imported, right-click on `rfid_test_bsp` `BSP Editor`. It will open a window like the one below. Make sure that the options are all the same.



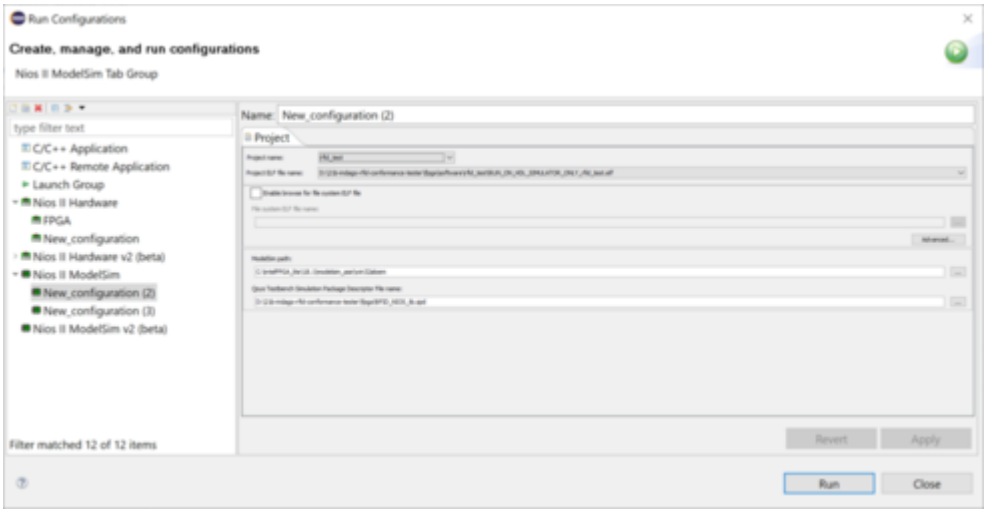
BSP Editor settings for ModelSim

You also need to check the `enable_small_driver` box on the `Drivers` tab.

After checking the settings, click on the `Generate` button. Once you generated the BSP, open up the `config.h` file in the `helpers` folder of the `rfid_test` project, and make sure that the value of the `MASK_LOOPBACK` is set to `1`. After that, save the `config.h` file, right-click the `rfid_test` project and click on `Build Project`.

If you had any errors, try cleaning both projects, generating the BSP and rebuilding the project.

Once the build is complete, go to `Run` `Run configurations`. It should open up a new window. Right click on `Nios II Modelsim`, and click on `New`. A new ModelSim run configuration has been created, and you can click on `Run` to simulate. The image below shows the configuration settings.



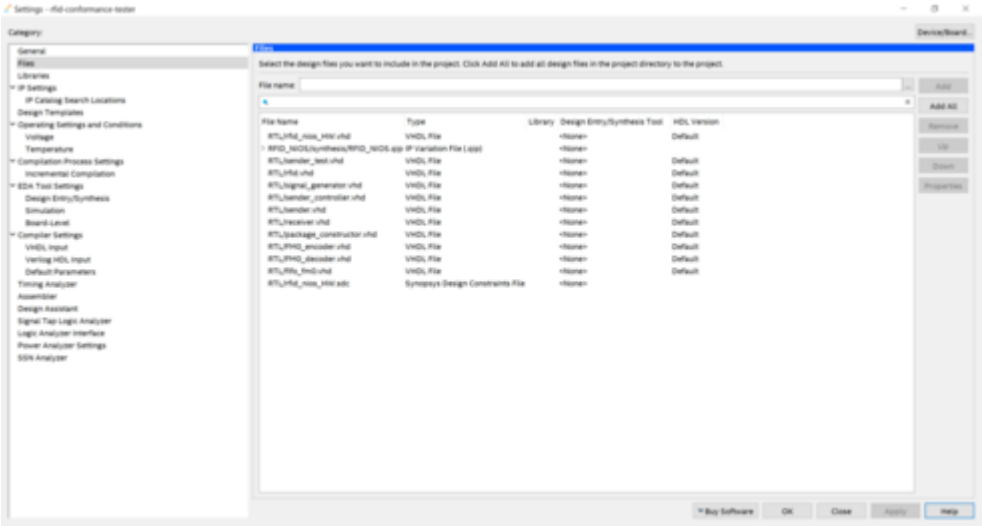
ModelSim run configuration window prompt

You can also watch the video below, which shows the step-by-step process.

2.3.2 Hardware guide

Important: If you want to launch the project on the DE-10 Standard board, a Quartus License is required.

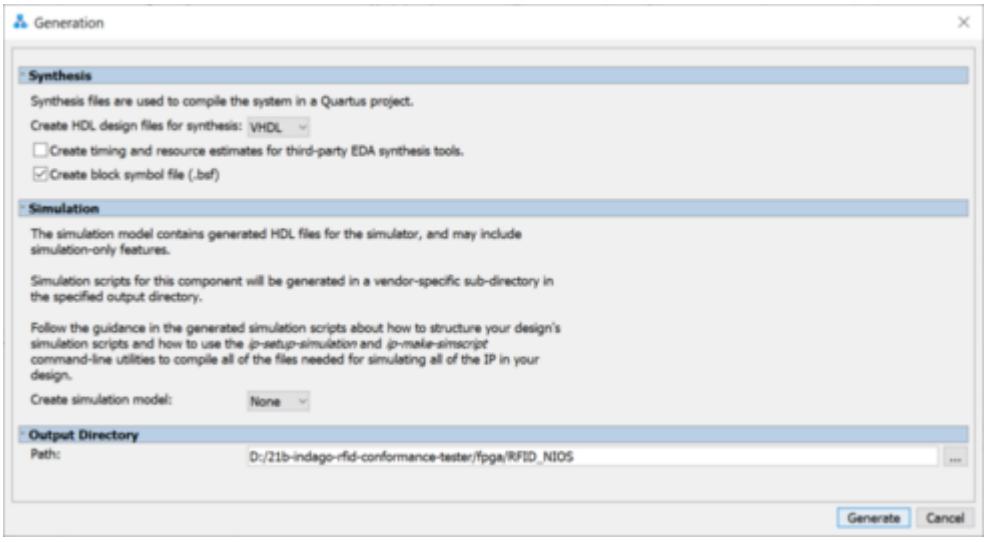
To launch the project on the DE-10 Standard board, the first step is to check the files present in the project. Click on `Project` `Add/Remove Files in Project`. It should look like the picture below.



Hardware implementation project files

If you are missing the `RFID_NIOS.qip` file, you can generate it using the `Platform Designer`. You can follow the same instructions used on the ModelSim guide to open the `Platform Designer` and open the project's design.

Once the design is open, click on `Generate` `Generate HDL...`. The settings should match the ones present in the image below. After that, just click on the `Generate` button, and it should generate the IP variation file.



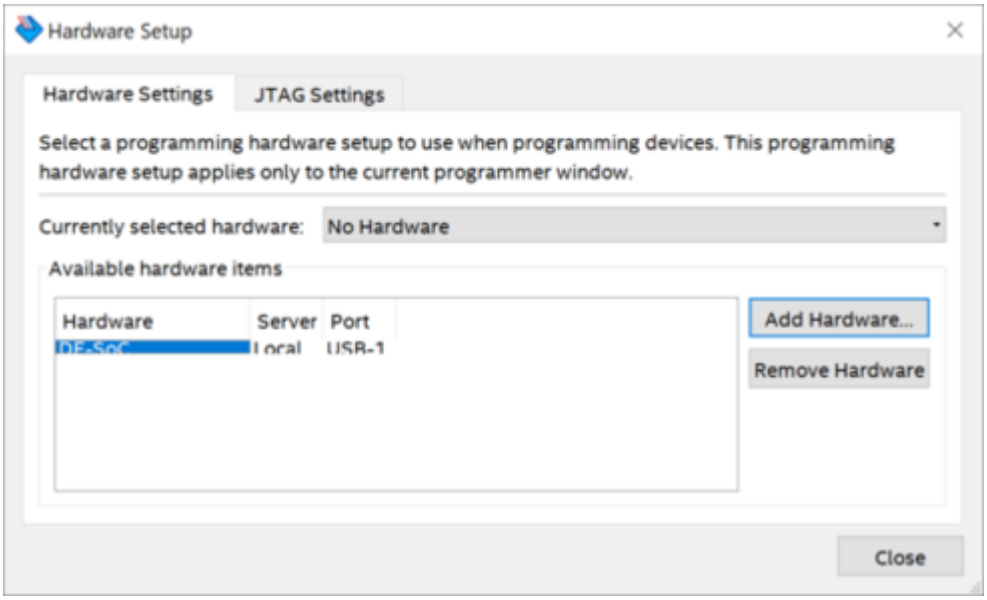
Generate HDL window prompt with settings

After checking the files, go to `Project Navigator` on the left panel, click on the dropdown menu that says `Hierarchy`, then click on `Files`. After that, right-click on the `RTL/rfid_nios_HW.vhd` file, and click on the option that says `Set as Top-Level Entity`. Now you can click on the blue play button next to the stop button to start the compilation.

After compiling, go to `Tools` `Programmmer` to program the board. Be sure that the board is plugged in before opening.

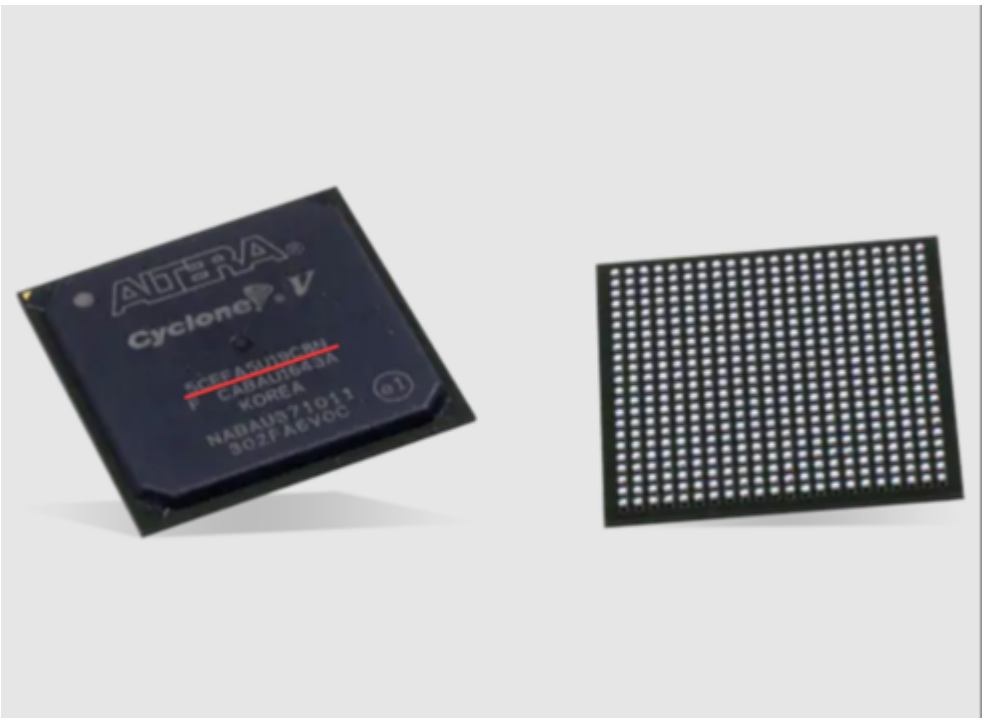
Quartus Programmer window

If your board is not showing up on the `Hardware` menu on top, click on `Hardware Setup`, then double click on your board in the menu. After selecting your board, close this window.



Hardware Setup window prompt

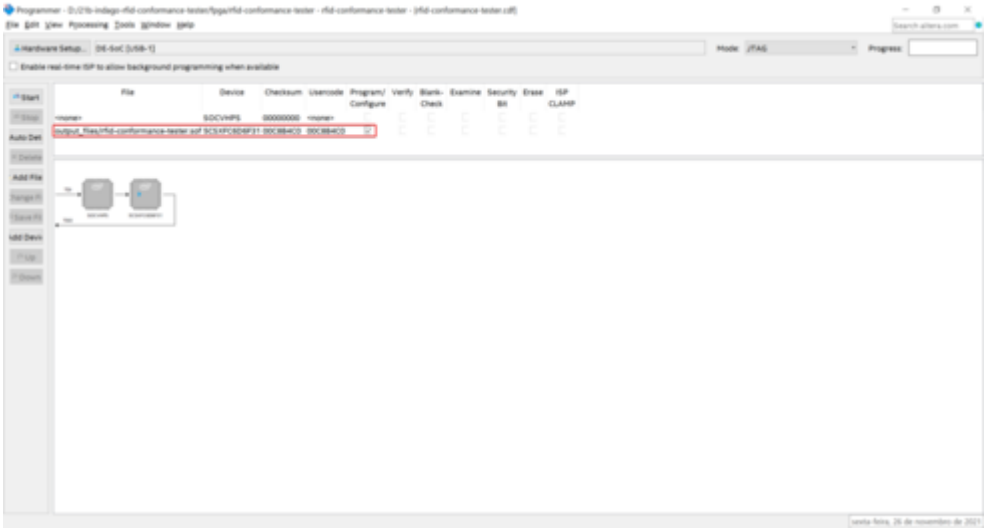
Once your board is shown in the `Hardware` menu, click on the `Auto Detect` button on the left menu, and select the corresponding device name for your board (it is engraved on the chip, shown in the image below).



Cyclone V device chip (image obtained here)

After selecting the corresponding device name, the program may show two chips: one named `SOCVHPS` and the other with your device. Double-click on the `File` tab next to the name of your device, and look for the `rfid-conformance-tester.sof` file for the compiled project, located on the `21b-indago-rfid-conformance-tester/fpga/output_files` folder.

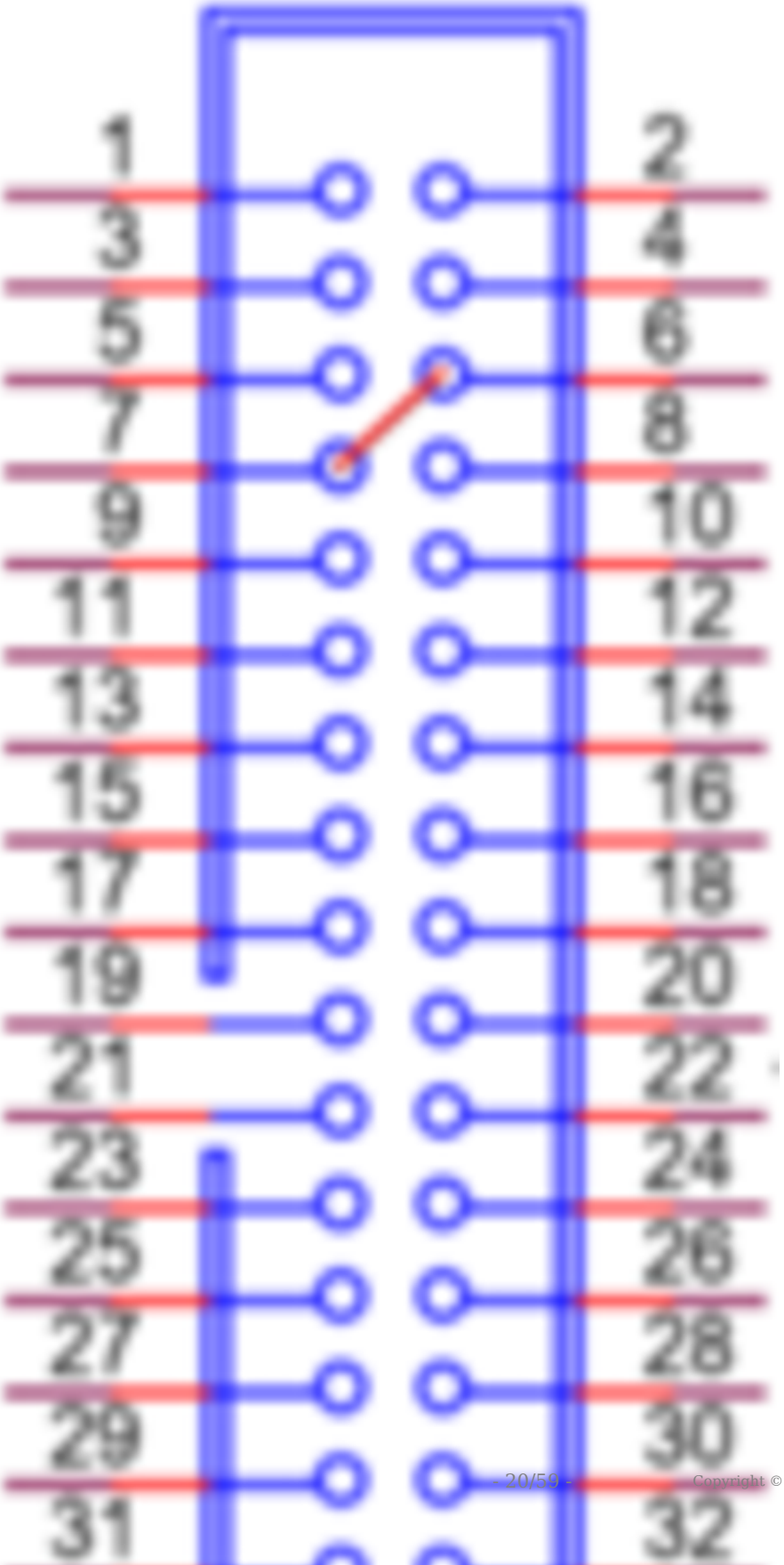
Once you selected the file, check the `Program/Configure` box next to your device as is shown in the image below, then click on `Start` to program the board. Once it's finished, you can close the `Programmer` window.



Quartus Programmer window with board detected

After programming the board, get a female/female jumper, plug one end on pin 6 of the GPIO header and the other end on pin 7 of the GPIO header, as shown in the image below.

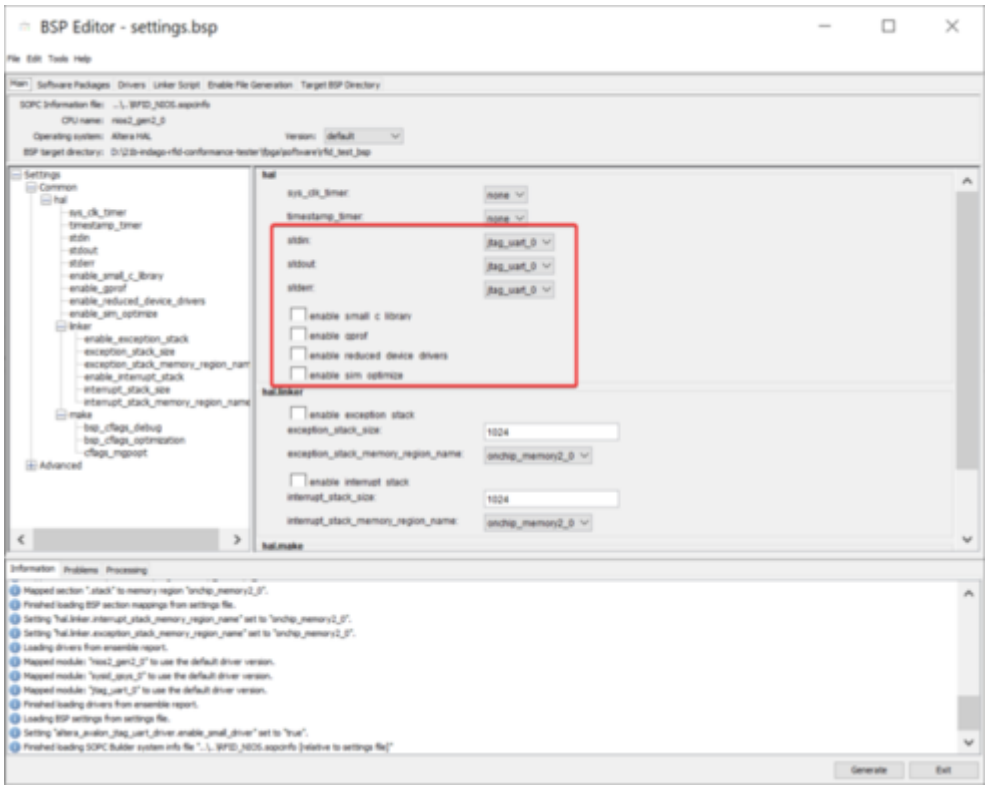
GPIO



connections (image obtained here)

Once the jumpers are connected, open up the `Nios® II Software Build Tools for Eclipse`. You can follow the same steps shown in the ModelSim guide on how to do so.

After both projects have been imported, right-click on `rfid_test_bsp` `BSP Editor`.Make sure that the options are the same as the ones present in the image below (they are **not** the same settings from the ModelSim guide).



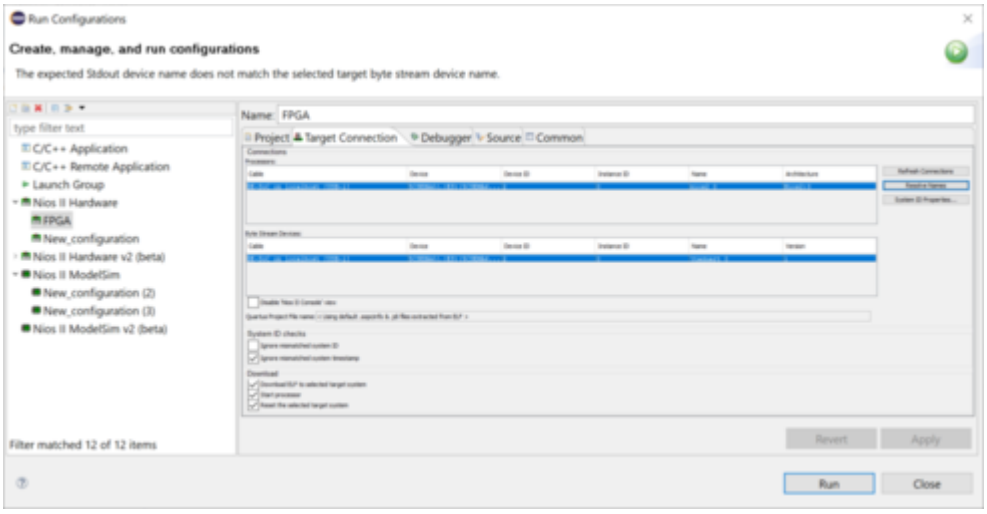
BSP Editor settings for hardware

The `enable_small_driver` box on the `Drivers` tab needs to be kept checked.

After checking the settings, click on the `Generate` button on the editor. Once you generated the BSP, open up the `config.h` file in the `helpers` folder of the `rfid_test` project. This time, the value of the `MASK_LOOPBACK` needs to be set to `0`. After that, save the `config.h` file, right-click the `rfid_test` project and click on `Build Project`.

If you had any errors, try cleaning both projects, generating the BSP and rebuilding the project.

Once the build is complete, go to `Run` `Run configurations`. It should open up a new window. Right click on `Nios II Hardware`, and click on `New`. Once the configuration has been created, go to the `Target Connection` tab, and check if the settings are the same as the one shown in the image below. Once you checked the settings, you can click on `Run` to launch.



Hardware run configuration window prompt

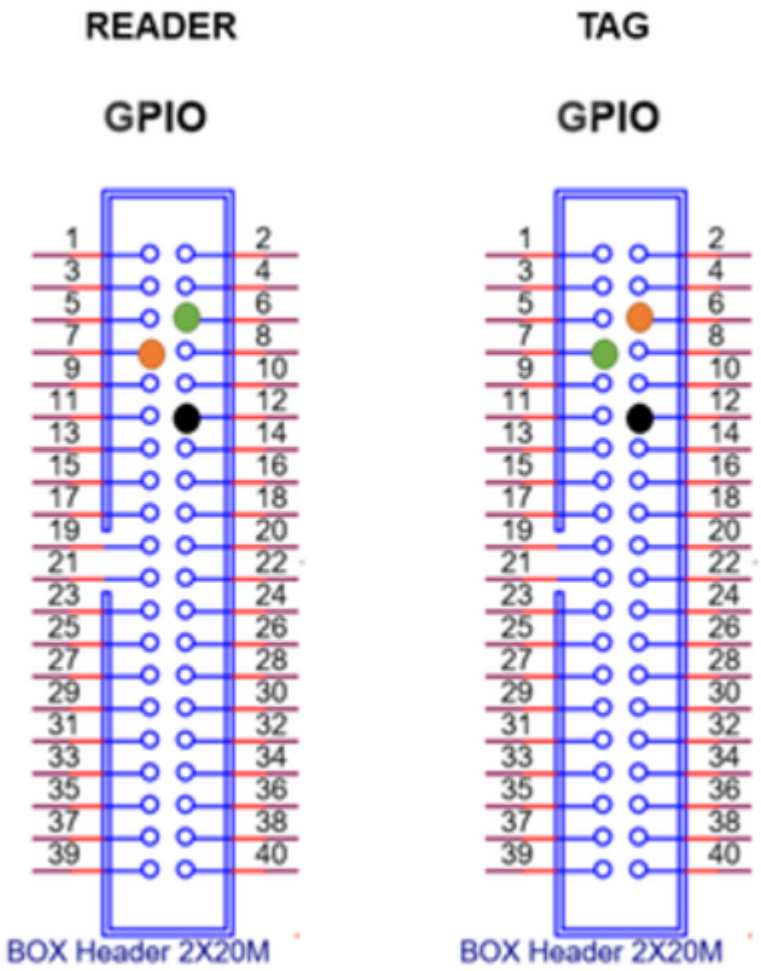
You can also watch the video below, which shows the step-by-step process.

2.3.3 Hardware guide (handshake)

You can also run a handshake version of the project using two DE-10 Standard boards. To do so, follow the same steps shown on the hardware guide until you reach the `rfid_test` project build. Before building the project, you need to find the `tag.c` and `reader.c` files, located in the `fpga` folder.

Choose one board to act as the tag, and one board to act as the reader; after that, replace the code present in the `main.c` with the corresponding code (one board will have the `tag.c` code and the other will have the `reader.c` code).

After overwriting the code, build the project on each Nios workspace (you can open up two workspaces on the same PC, but it is a little trickier to do so; hence, it is recommended that you do this on two PCs). Once the projects are built, use female/female jumpers to connect the board pins as shown in the image below.



GPIO schematic with jumper connections (image obtained here)

After properly connecting the pins, you can use the same run configurations created on the hardware guide to run the project.

Important: First, launch the tag configuration, and wait for the `Nios II Console` to print the `waiting for query` message. After that, you can launch the reader configuration.

You can also watch the video below, which shows the step-by-step process.

2.4 How to collaborate

Firstly, you should create a fork of the original repository to work on. You can learn how to do so here. After creating your own fork, you can open a pull request to this repository. The instructions on how to do it can be found here. After making your pull request, it will be reviewed by one of the team members, and if everything's ok, they will approve it. If there are problems with your pull request, the reviewer will inform you of them, and you can make the adjustments necessary.

3. Hardware

For this project, the chosen solution for implementing the conformance tester was to develop a dedicated hardware in FPGA [1]. The chosen product was the DE10-Standard[2] produced by Terasic[3] as well as a FPGA Cyclone® V[4] from Intel®. The professor had ample experience working with this specific model, and it also perfectly fits the requirements need to implement the chosen solution. This is because, through a tool called "platform designer"[5], it can edit its configuration on demand, allowing great flexibility when needed.

The proposed solution makes use of Intel®'s solution development ecosystem, providing flexibility in the use of a soft processor, enabling the integration of peripherals called IP cores to its architecture, as well as allowing the creation of new instructions implemented in the hardware. For more information on these modifications, see the document "Nios II Custom Instruction User Guide" [6].

The image below shows the Analysis & Synthesis Status Report generated by Quartus, which indicates the ROM and RAM memory usage for this project on the FPGA.

Analysis & Synthesis Status	Successful - Mon Nov 29 16:43:20 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	rfid-conformance-tester
Top-level Entity Name	rfid_nios_hw
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	2456
Total pins	3
Total virtual pins	0
Total block memory bits	1,001,728
Total DSP Blocks	3
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

Analysis & Synthesis Status Report

3.1 File Hierarchy

All necessary VHDL hardware description files are located in the project's `fpga/RTL` folder. The top-level entity of the entire processor that includes all the required configuration generics is the `RTL/rfid_nios_HW.vhd` file.

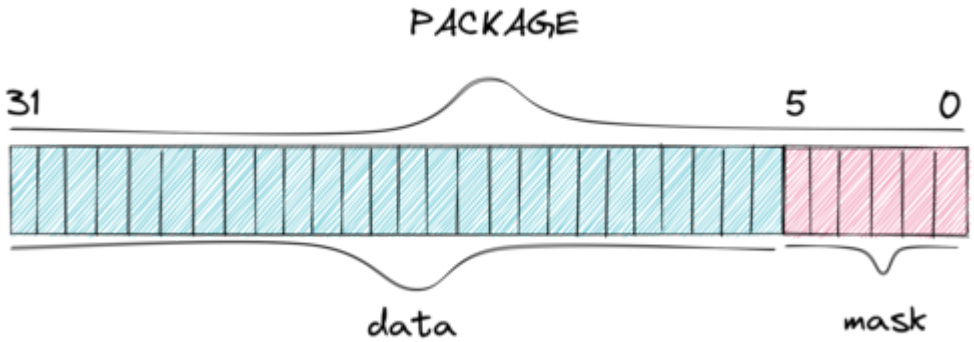
```
rfid_nios_HW.vhd            - Conformance tester top-level entity
|
├rfid.vhd                      - RFID component which integrates the sender and the receiver
```

```
|   ├sender.vhd                   - Sender component top entity
|   |    ├FIFO_fm0.vhd               - Encoder-specific FIFO
|   |    |   ├FM0_encoder.vhd          - Encoder-FM0-specific FIFO
|   |    |   └fifo_32_32.vhd           - General use FIFO
|   |    ├sender_controller.vhd      - Controls the flow of packages to the tag
|   |    └signal_generator.vhd       - Generates preamble or frame-sync signal
|   |
|   └receiver.vhd                 - Receiver component top entity
|        ├FM0_decoder.vhd            - Decoder-FM0-specific FIFO
|        ├fifo_32_32.vhd             - General use FIFO
|        └package_constructor.vhd    - Stores bits into packages before storing in the FIFO
|
├rfid_nios_HW.sdc            - Synopsis Design Constraint file, used for setting up the board's clock
```

The top-level entity implements the RFID component via an IP variation file (in this case the `fpga/RFID_NIOS/synthesis/RFID_NIOS.qip` file), which creates a new component that can be ported.

3.2 Packages and commands

This project uses the mandatory commands specified in the EPC-GEN2 documentation. However, those commands have varying sizes and even the same command could vary its size based on the data it sends. To work with this fluctuating command bit size, the group decided to separate the commands into 32-bit packages, where the 26 more significant bits are the actual data of the packet, and the 6 less significant are the mask, indicating how many of the 26 are in use.



Package visual example

This way, there are three possible situations given the command sizes:

- The command is larger than one package: if the command has more than 26 bits, it is split into multiple packages, communicated in order through the components (more significant -> less significant);
- The command is the same size as the package: the easiest case, where the package is treated as the full command;
- The command is shorter than one package: in this case the package will be filled up to the number of bits the command has, and then use the mask to communicate how many of the data bits in the package are useful, ignoring the ones not needed to the command;

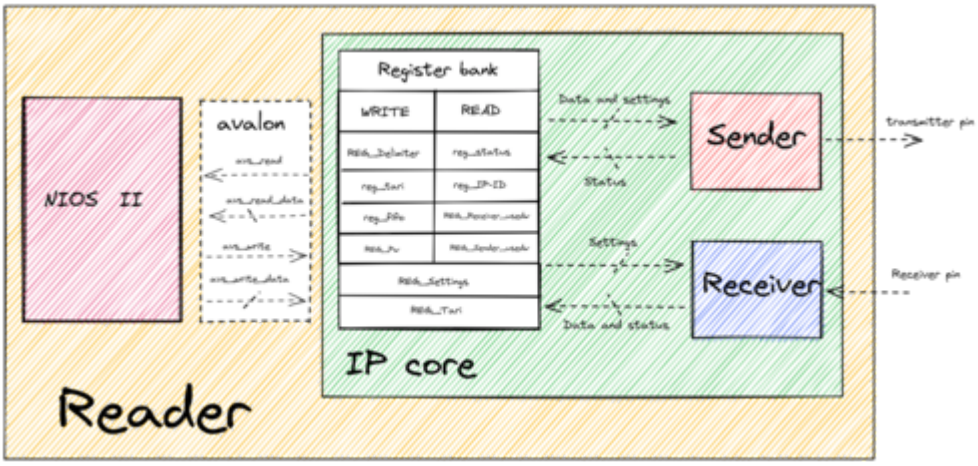
For example, if a command has 40 bits, it will be separated into two packets. The first one uses the 26 data bits, and the mask `011010` (26) to indicate all the data bits are in use. Then, the second package would only use 14 of the 26 data bits available to reach the 40 bits the command has, and so the mask would be `001110` (14) to indicate that only 14 bits should be analyzed.

To communicate between the components that the command is over, a `void package 0x00000000` is sent after the last package of the command. This occurs in two times in the product: first, when sending the command to the tag,

the Nios II sends a void package to the Sender to indicate the command is over. Second, when receiving the response from the tag, the receiver sends a void package to the Nios II to indicate that the command received is over.

3.3 Solution: Reader

The reader, as shown in the diagram below, is the solution implemented in this project, which contains the two main components. An in depth analysis is present in the sections below.



Reader component visual diagram

The first one is the Nios II soft processor, where the group programmed the tests that will be run on the tag. Therefore, its responsible to generate the commands for communicating with the tag, as well as interpreting the responses it receives, to assert whether the tag passes or fails each test.

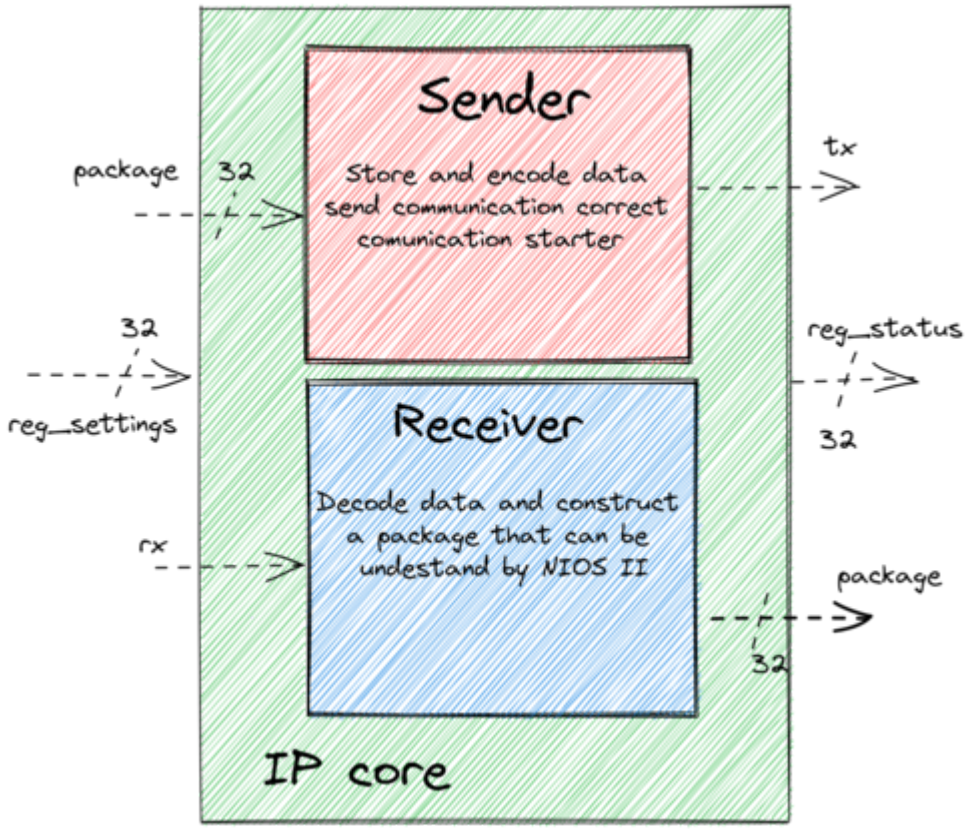
The second component is the IP core, developed in VHDL, and is responsible for encoding and sending messages to the tag, as well as decoding any responses and passing it through to the processor. Those two tasks have been separated into the sender and the receiver respectively.

The Avalon[7] Interface is the connection between the Nios II and the IP core, where the commands, generated in the programming language C, will be passed on to the VHDL sender, and responses will take the opposite path, going from the receiver to the processor.

3.3.1 IP CORE

rfid.vhd entity

The developed peripheral can be split into two components, as mentioned in the previous subsection, it can be visualized in the diagram below. Those are the sender, in red, responsible for receiving the data from the Nios II, encoding and forwarding them to the tag; and the receiver, in blue, responsible for receiving the data from the tag, decoding and forwarding them to the Nios II.



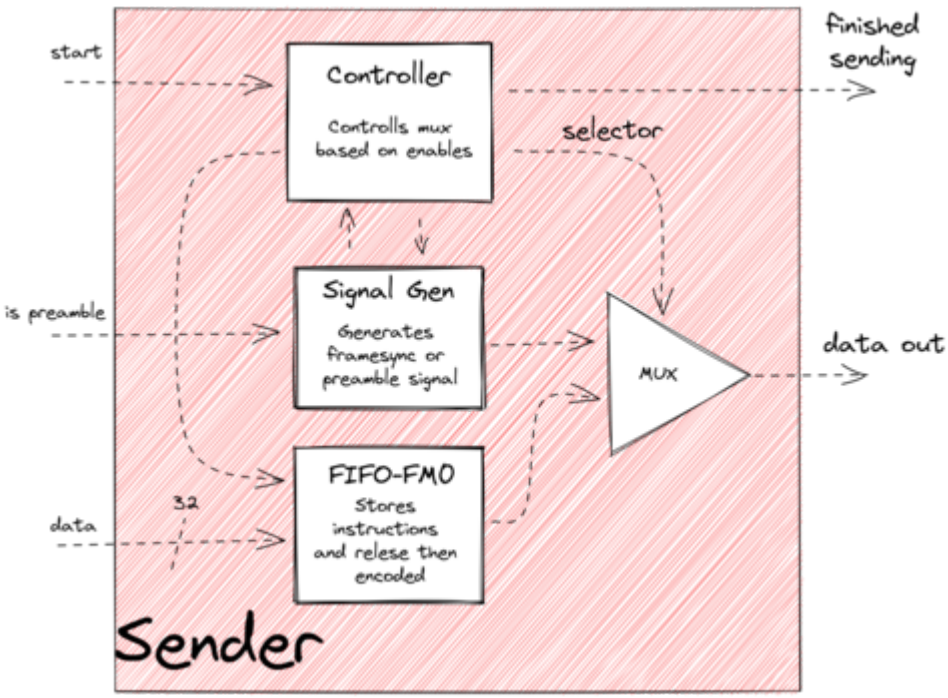
IP core visual diagram

That implementation permits a feature in which the developer can link the transmitter pin with the receiver pin, which the team named by loopback mode. This mode helps with debugging of both the IP core and the Nios II software side, because it permits a full communication in which the same fpga makes the role of the reader and the tag at the same time. This feature will eventually show up in the firmware page.

3.3.2 Sender

sender.vhd entity

This component is responsible for encoding the commands generated by the processor, and send them to the tag, respecting the rules of the EPC-GEN2 protocol, including the Tari, signal generation and EOP. Its components are detailed below:



Sender component visual diagram

3.3.3 FIFO

/main/fpga/fifo_32_32.vhd

The first component of the sender is the FIFO, a storage system that receives the assembled commands from the Nios II and waits for the encoder to send the read request flag, signaling for the FIFO to send the oldest package. It is possible that the command to be sent to the tag is composed of more than one package, so the FIFO serves as a storage system for packages already received from the processor until it signals that the entire command is ready for encoding.

For this project, the group opted to use the FIFO produced by Intel, which was obtained through the Quartus automatic generator, the main software used by the group for programming in VHDL language. It is possible to include several customizations before generating the code, and thus, the group defined that the FIFO would have the settings below:

3.4 FIFO characteristics

- IP variation file type VHDL;
- Synchronous reading and writing to clock;
- 32 bits per word;
- 256 word depth;
- `full`, `empty`, `usedw` (number of words in fifo), `synchronous clear` (flush the FIFO);
- Memory block type `auto`;
- Not registered output;
- Circuitry protection enabled;

3.4.1 FIFO_FM0

fifo_fm0.vhd entity

FIFO_FM0 is a component created to separate the data encoding component from the signal-generator and the sender-controller, mapping the inputs: `data` to be encoded, `write request` for the FIFO, `enable` and `tari` to the other components, allowing for individual control of each component.

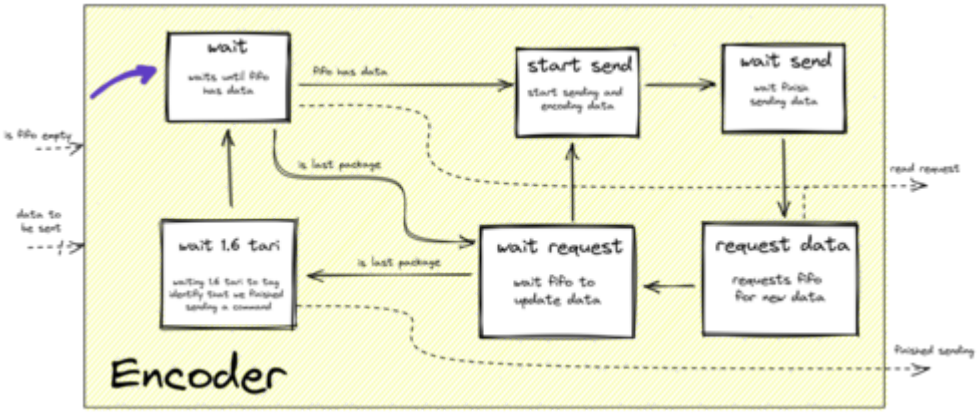
3.4.2 Encoder

FM0_encoder.vhd entity

The encoder is the main component of the sender, being responsible for encoding the packages received from the FIFO with FM0 band, as specified in the EPC-GEN2, as well as sending the encoded data to the tag. for this purpose, a refined control of the time intervals is necessary to obey the Tari, also defined in the protocol as being between 6.25µs and 25µs.

This component has two state machines that work simultaneously, one responsible for communicating with the FIFO and sending it to the tag, while the other encodes the received data. The diagram below demonstrates the first state machine:

- `Wait`: the encoder's default state, it remains in this state while it doesn't receive any new packages to encode;
- `Start Send`: this state starts the FM0 Generator state machine, which is responsible for encoding the packages. Furthermore, it also expects to receive the correct Tari to send data to the tag;
- `Wait Send` is a temporary state for when the data has not been fully encoded, and therefore needs to wait until the other state machine finishes the encoding;
- `Request Data` happens after the data has been sent, and signals the FIFO to send more data. This state is very short, as its only duty is to send a flag to the FIFO, and immediately change to the `Wait Request` state;
- `Wait Request` can happen in two situations. First, if the encoder is waiting for the next package from the FIFO, which case it goes back to the `Start Send` state once the package is received. Second, it can happen once the FIFO sends the `FIFO_empty` signal to the encoder, in which case the void package is removed, waiting for the next command;
- `Wait 1.6 tari` is the formal completion of the command sent to the tag, where a `dummy 1` bit is sent, which will remain active for 1.6 Tari and then stop the communication.

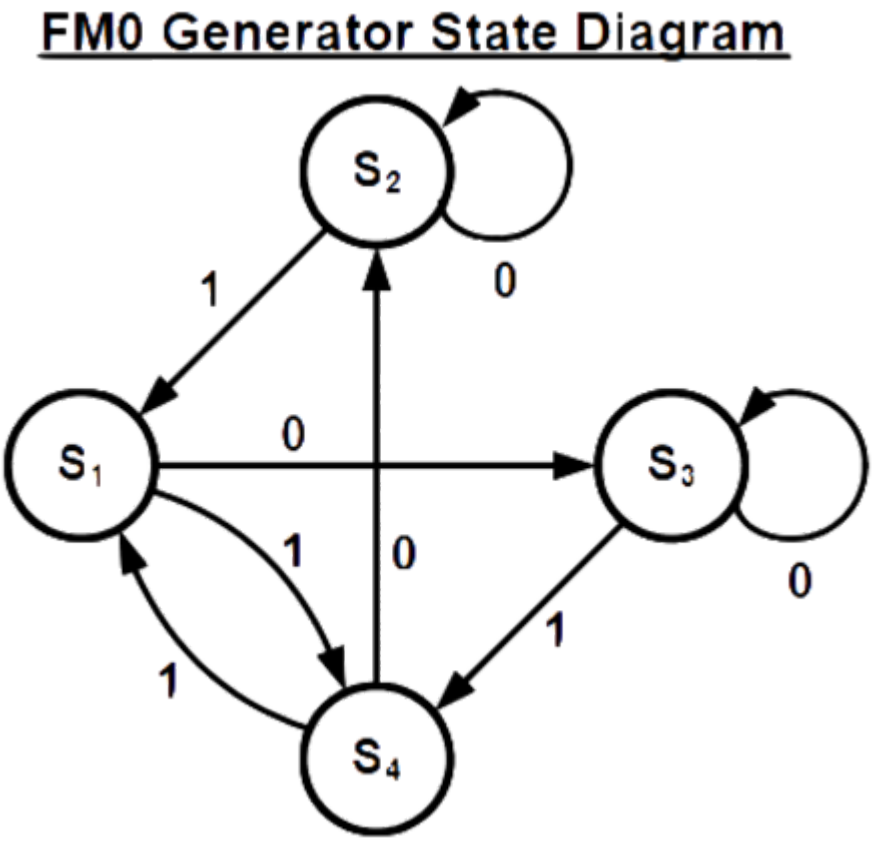


Encoder state-machine visual diagram

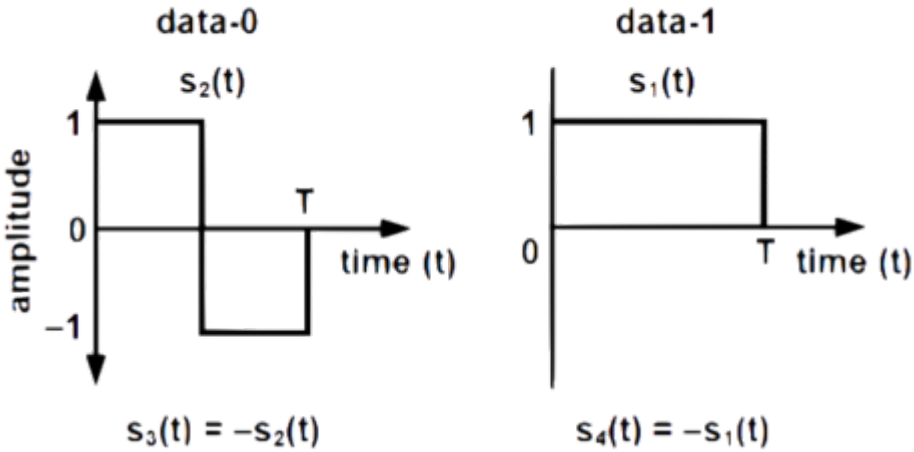
The next image demonstrates the other state machine present in the component, responsible for the encoding of the data. It was defined that it would always start in state `S3`. FM0 encoding transforms each bit of information into two bits, in such a way so that a `1` becomes two bits of the same value (either `1 1` or `0 0`), and a `0` becomes

two bits of different values (either `1 0` or `0 1`), where the signal always alternates when encoding a new bit. The change of state occurs after each bit has been sent and is defined by the value of the next bit.

- `S1` encodes `1` into `1 1`;
- `S2` encodes `0` into `1 0`;
- `S3` encodes `0` into `0 1`;
- `S4` encodes `1` into `0 0`.

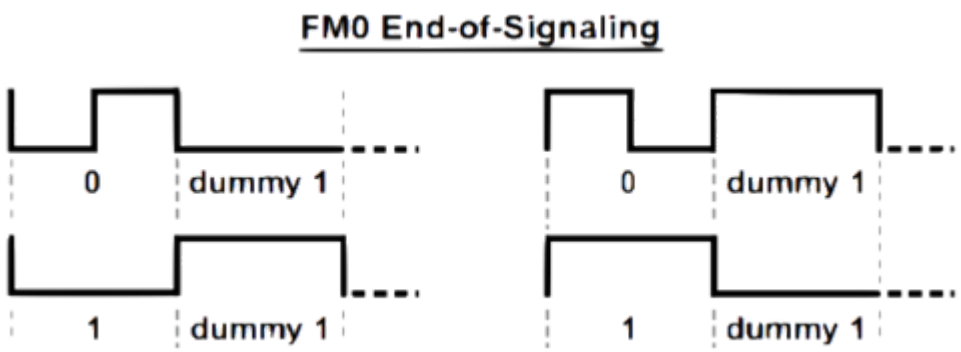


EPC UHF Gen2 Air Interface Protocol, p 32



EPC UHF Gen2 Air Interface Protocol, p 32

The previously defined `dummy 1` acts as the `EOP` of a command passed to the tag, however it also needs to be encoded, and is always followed by a `0` bit. This is shown in the image below.



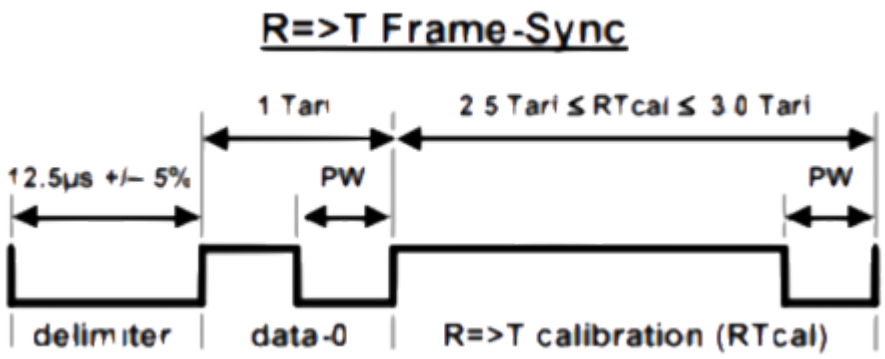
EPC UHF Gen2 Air Interface Protocol, p 33

Signal Generator

signal_generator.vhd entity

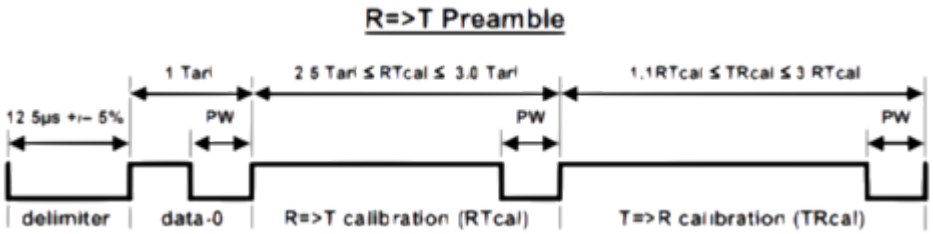
This component encompasses both the Preamble and Frame-sync functions, and receives flags to determine which one, if any, will be activated.

The Frame-sync is responsible for defining and regulating the interval at which information is sent to the tag, and sharing this interval to all other sender components, so that they can communicate within the correct time intervals. This period, named Tari, must be within the range defined in the protocol, and have a variation of less than 1% between each pulse.



EPC UHF Gen2 Air Interface Protocol, p 29

The Preamble is responsible for the first wave of information sent to the tag for each new command, and it defines which Tari will be used throughout the next command. This component needs to be activated for every command that is sent to the tag, except when more than one command is sent in sequence, without a response in between. In this case, the preamble informed will be valid for all subsequent commands, until a response is requested.



EPC UHF Gen2 Air Interface Protocol, p 29

Sender Controller

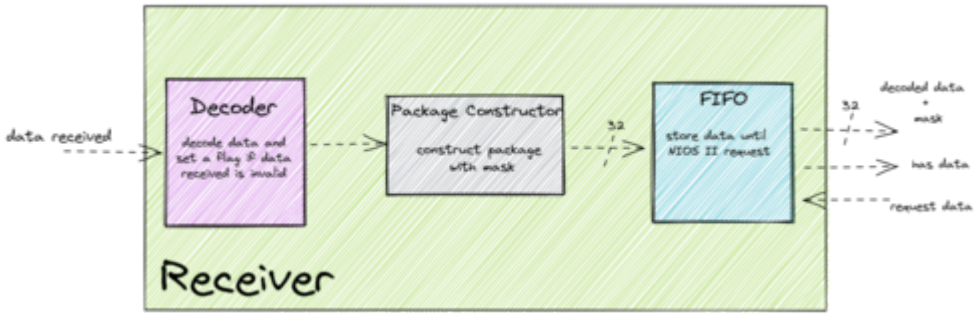
sender_controller.vhd entity

The Sender controller is responsible for managing the components of the sender, such as controlling the encoder, or generating the flags for the signal generator. This component is essential to the sender, since each step must be done in order, so each package can be encoded and sent, and the preamble or frame-sync data can be generated and sent at the start of each new communication.

3.4.3 Receiver

receiver.vhd entity

The receiver is responsible for receiving the responses from the tag, decode them, and notify the Nios II processor that there was a response, as well as store each package of the response until the processor sends the `read request` flags to analyze them. In order for the received data to be interpreted, it is necessary that the information is decoded and grouped into packages, because it is possible the response is too large for the processor to receive all at once. The group decided to split the receiver into three smaller components, shown and described below:



Receiver component visual diagram

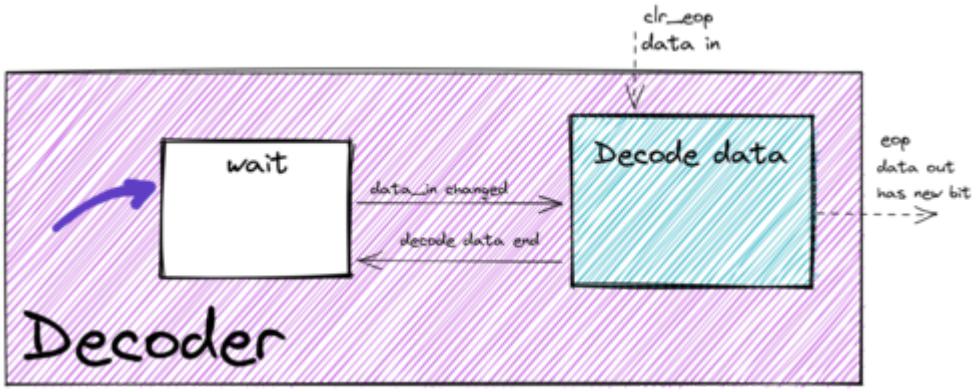
Decoder

FM0_decoder.vhd entity

Since the tag also communicates back to the reader using FM0 encoding, a decoder component is needed to decode the received data, allowing it to be interpreted by the processor. This component was built in a similar

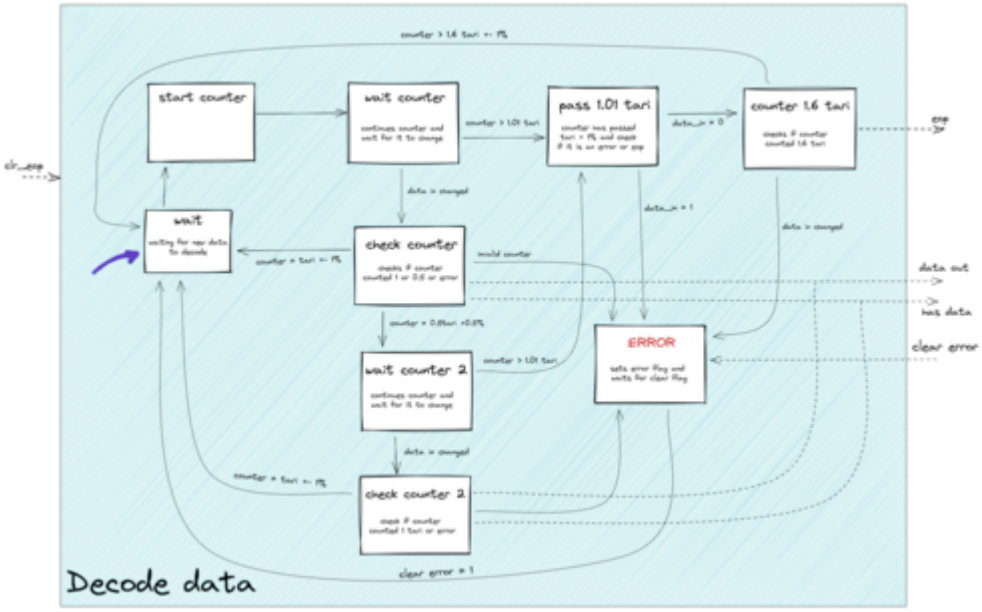
way to the sender, through the use of two state machines, one of which operates inside the other. The diagram below demonstrates the first state machine programmed for this purpose:

- `Wait` is the decoder's default, the state it remains in while it doesn't receive any new data to decode;
- `Decode Data` runs a stand-by mode while for this state machine and activates the other one, responsible for decoding the data received and sending it to the FIFO;



Below is the diagram for the other state machine, responsible for checking if all the time intervals for the communication are correct, as well as decoding the data, and sending it to the FIFO afterwards:

- `Wait` is the decoder's default, the state it remains in while it doesn't receive any new data to decode;
- `Start Counter` starts a time counter as soon as the decoder receives new data, in order to determine if the bit will change after 0.5 or 1.0 Tari, then passing to the next state. It is also possible for the bit to remain unchanged for more than 1.0 Tari, in which case it will go to the `Pass 1.01 tari` state;
- `Stop Counter` sends `1` to the package constructor if the input signal has not changed, and `0` otherwise;
- `Continue Counter` is necessary because the stop counter always stop at 0.5 Tari, so it is activated if no bit change occurs;
- `Pass 1.01 tari` is activated when the tag sends the `dummy 1`, which has a duration of 1.6 Tari, and checks if the times are in accordance with the protocol;
- `Counter CS` stops the counter and resets the decoder to its default state;
- `ERROR` is a state that can be activated by almost any other state, as they all check certain characteristics of that tag that must comply with the protocol. If something is irregular, this status will be activated and will send an error message explaining what caused this to happen;



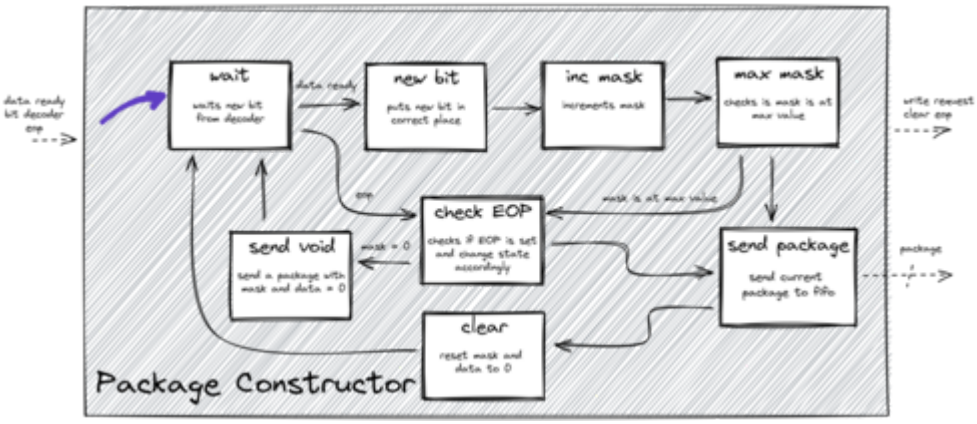
Decoder state-machine visual diagram

Package Constructor

[package_constructor.vhd entity]

This component is responsible for assembling the decoded bits into packages and storing them in the FIFO. It gathers the received bits until reaching the limit defined in the code, and then sending to the FIFO. If, however, the package constructor receives the `EOP` signal before completing the package, it will concatenate a mask with the current package to inform how many bits were filled. Lastly, it will the `void package` to the FIFO and the processor, which indicates the receiver has finished capturing and decoding the whole response command, and that it has been fully passed to the FIFO.

- `Wait` is the package constructor's default, the state it remains in while it doesn't receive any new data to store;
- `New Bit` happens when the decoder sends a decoded bit to the package constructor, which stores it in the current package being constructed;
- `Inc Mask` increases the package mask by 1 representing the new bit received;
- `Max Mask` checks whether the mask, and therefore the package, is full, preparing to send the package if it is;
- `Send Package` sends the current package to the FIFO, an intermediary step before going to the Nios II processor;
- `Check EOP` checks if the EOP flag is high, and changes state based on the current packet. If it is empty, goes to the `Send void` state, if not goes to the `Send Package`;
- `Clear` clears the current package before starting a new one;
- `Send Void` send to the FIFO an empty package - `0x000000`



Package constructor state-machine visual diagram

FIFO

/main/fpga/fifo_32_32.vhd

The FIFO in the receiver is the same as the one in the sender and serves similar purpose; however, its direction is inverted. This FIFO stores packages leaving the package constructor until it receives the `EOP` flag, which also signals the Nios II that the command is ready to be requested. After this, the FIFO sends one package at a time to the processor whenever it receives a `read request` flag.

---

1. FPGA Intel®. https://www.intel.com.br/content/www/br/pt/products/programmable.html Accessed on: 23/08/2021 ↩

2. Terasic DE10-Standard Development Kit. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=1081 Accessed on: 23/08/2021 ↩

3. Terasic Inc.. https://www.terasic.com.tw/en/ Accessed on: 23/08/2021 ↩

4. Cyclone® V GT FPGA. https://www.intel.com.br/content/www/br/pt/products/details/fpga/cyclone/v/gt.html Accessed on: 23/08/2021 ↩

5. Platform Designer - Intel®'s System Integration Tool. https://www.intel.com/content/www/us/en/programmable/products/design-software/fpga-design/quartus-prime/features/qts-platform-designer.html Accessed on: 23/08/2021 ↩

6. Nios II Custom Instruction User Guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf Acessed on: 23/08/2021. ↩

7. Avalon® Interface. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf Acessed on: 23/08/2021. ↩

4. IP core interface

Below is an in depth explanation of the mechanism responsible for the communication between the Nios II processor and the IP core, which is composed by the Avalon Interface and a register bank that connect the other components.

4.1 Solution Diagram

In this diagram, generated by the Platform Designer, it is possible to see the overview of the solution implemented by the group. Inside, the clock interface, Nios II processor, memory, and IP core are connected by the Avalon Interface, represented by the blue dotted lines.

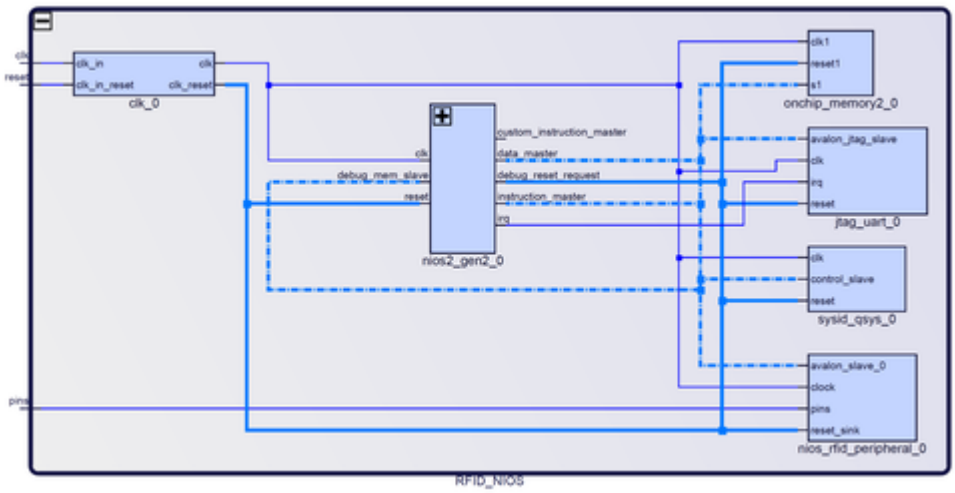


Diagram generated by the Quartus Platform Designer

The Cyclone V FPGA clock interface works on a 50MHz clock speed, which allows for multiple clocks between each Tari interval, ensuring that the encoding and decoding processes run smoothly. All the time intervals necessary for communication, as present in the EPC-GEN2 documentation, are also calculated based on the original 50MHz clock.

The Nios II soft processor used in this project was implemented by the team and is based on RISC architecture. It is responsible for generating all the commands sent to the tag, as well as interpreting the responses our device receives. More information on the Nios II can be found here.

The on-chip memory is the RAM memory for the system. It is accessed through the Avalon bus which writes and reads data on the register bank. It has 115.199 addressable bytes divided between the different interfaces.

The IP core contains all the VHDL components of our projects, encompassing both the sender and receiver, as well as all their inner components. These are also accessed by the Avalon bus to receive settings and commands, as well as to send the responses back to the processor. More information on the IP core can be found here.

4.2 Avalon Interface

The Avalon Interface, according to its developer Intel®: "Avalon® interfaces simplify system design by allowing you to easily connect components in Intel® FPGA. The Avalon interface family defines interfaces appropriate for

streaming high-speed data, reading, and writing registers and memory, and controlling off-chip devices. Components available in Platform Designer incorporate these standard interfaces. Additionally, you can incorporate Avalon interfaces in custom components, enhancing the interoperability of designs.".

In other words, it is possible to conclude from this excerpt that, in addition to enabling the connection between Intel® FPGA components, that once the interfaces are added to Platform Designer, they can connect custom components, which the group uses for the IP core in our product. All documentation for other possible interfaces and connection is present in the document “Avalon® Interface Specifications”.

The Avalon Bus can be implemented with multiple features and modules. In this project, Avalon Memory-Mapped will be used, but other alternatives were considered, such as Avalon Interrupt Interfaces and Avalon Streaming Interfaces.

The Avalon Memory-Mapped (Avalon-MM) interface implements both the Instruction and the Data Master Ports. Both of them share a single memory and also a single bus to outside of the processor. According to the same document the data and instructions master ports never cause a gridlock in which one port starves the other.

The one accessed directly by the software on this project is the Data Master Port, which is a 32-bit bus capable of reading and writing in the memory or in any other peripheral of the project, just like it does in our IP core. This bus is one of the dotted line in the platform designer image above.

4.3 Register Bank

The register bank intermediates the communication between the Nios II processor and the IP core. Each access to the bus is done through a `read` or `write` command in which they carry data up to 32 bits, a value that represents the size of an integer in the programming language C, as it is the one implemented in the processor.

As seen in the table, this communication path is responsible for the IP peripheral settings, informing essential values for communication such as `Tari`, `Delimiter` and `Status`, as well as others shown below.

offset	Name	Access	Size (bits)
0	REG_Settings	Read/Write	32
1	REG_Tari	Read/Write	16
2	REG_FIFO	Write	32
3	REG_Tari_101	Write	16
4	REG_Tari_099	Write	16
5	REG_Tari_1616	Write	16
6	REG_Tari_1584	Write	16
7	REG_Pw	Write	16
8	REG_Delimiter	Write	16
9	REG_RTcal	Write	16
10	REG_TRcal	Write	16
3	REG_Status	Read	32
4	REG_Receiver_data_out	Read	32
5	REG_Sender_usedw	Read	8
6	REG_Receiver_usedw	Read	8
7	REG_IP-ID	Read	32

While some registers indicate a single variable, such as those that indicate Tari values, others carry multiple peripheral control variables with them, those being `REG_SETTINGS` and `REG_STATUS`.

4.3.1 Offset 0 - REG_Settings (R/W)

The Register Settings is responsible for the control of the IP core, as a Read and Write register it is capable of setting flags, making pulses in specific bits and also activating components of the IP core

31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
x	x	x	x	x	x	x	x
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
x	x	x	x	x	x	x	x
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
x	x	x	Receiver read request	Receiver reset	Sender clr finished send	x	Loopback
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Sender is preamble	Sender start controller	Sender has_gen	Receiver enable	x	Sender clear FIFO	Sender enable	Sender reset

- Bits 31 through 13 are unused, instead being reserved for future implementations.
- Bit 12 - `Receiver read request` stores the read request flag for the Receiver FIFO;
- Bit 11 - `Receiver reset` stores the reset flag for the Receiver;
- Bit 10 - `Sender clr finished send` stores the Sender clear finished send flag;
- Bit 9 - unused;
- Bit 8 - `Loopback` stores the loopback flag used in testing the reader;
- Bit 7 - `Sender is preamble` stores the preamble flag for the Sender generator;
- Bit 6 - `Sender start controller` stores the start controller flag for the Sender;
- Bit 5 - `Sender has_gen` stores the generator flag for the Sender;
- Bit 4 - `Receiver enable` stores the enable flag for the Receiver;
- Bit 3 - unused;
- Bit 2 - `Sender clear FIFO` stores the clear FIFO flag for the Sender FIFO;
- Bit 1 - `Sender enable` stores the enable flag for the Sender;
- Bit 0 - `Sender reset` stores the reset flag for the Sender;

4.3.2 Offset 1 - REG_Tari (R/W)

The Register Tari carries the time parameter to the IP Core, in the software it is declared as an int and can vary between 313 to 1250 values. These range is calculated by multiplying frequency of the µProcessor times the Tari range. The Tari range is 6.25µs to 25µs, which is also stablished in EPC-GEN2 Documentation.

Some values can be used to manually set the Tari time parameter. Values from the range of 300 - 600 are allowed by the EPC-GEN2 documentation.

The Tari is written in the Register Tari from least significant (0) to most significant bit (15) of the register. Depending only on the binary value for the Integer mentioned previously.

15	**14**	**...**	**2**	**1**	**0**
x	x	...	x	x	x

4.3.3 Offset 2 - REG_FIFO (W)

The register FIFO carries the package, from the C Software to the IP core through Avalon bus.

The FIFO register is also written from least significant (0) to most significant bit (31) of the register. In which the six least significant bits are the mask which carries the size of the data and the remaining 26 are reserved for data itself.

31	30	...	2	1	0
x	x	...	x	x	x

4.3.4 TARI limit Registers

The Registers 3 to 6 are called Registers Tari Boundaries, they stablish the 1%-time variation in which the READER can communicate with the tag. If the tag does not reply in time or if the Sender or Receiver peripherals delay, an error flag will be raised as if the communication had ended.

Each Tari Boundary is written its respective Register from least significant (0) to most significant bit (15) of the register. Depending only on the binary value for the Integer that it represents.

Offset 3 - REG_Tari_101 (W)

15	14	...	2	1	0
x	x	...	x	x	x

Stores the Tari upper limit.

Offset 4 - REG_Tari_099 (W)

15	14	...	2	1	0
x	x	...	x	x	x

Stores the Tari lower limit.

Offset 5 - REG_Tari_1616 (W)

15	14	...	2	1	0
x	x	...	x	x	x

stores the 1.6 Tari upper limit.

Offset 6 - REG_Tari_1584 (W)

15	14	...	2	1	0
x	x	...	x	x	x

stores the 1.6 Tari lower limit.

4.3.5 Preamble and Framesync time parameter registers

The following registers are also time parameters which can be seen in the Preamble and Frame-sync images, they are part of the RFID modulation and a future implementation of Radio frequency depends on these values to be set.

Each value written its respective Register from least significant (0) to most significant bit (15) of the register. Depending only on the binary value for the Integer that it represents.

Offset 7 - REG_Pw (W)

Stores the value of a half-tari, which is used in the signal generator component to generate the preamble or frame-sync.

15	14	...	2	1	0
x	x	...	x	x	x

Offset 8 - REG_Delimiter (W)

Stores a value between 0.5 - 1.0 Tari responsible for starting the communication between READER and tag.

15	14	...	2	1	0
x	x	...	x	x	x

Offset 9 - REG_RTcal (W)

Stores the RTcal variable, which is based on the data lenght of the communication.

15	14	...	2	1	0
x	x	...	x	x	x

Offset 10 - REG_TRcal (W)

Stores the TRcal variable, which is used by the READER to specify the tag's backscatter link frequency.

15	14	...	2	1	0
x	x	...	x	x	x

4.3.6 Offset 3 - REG_Status (R)

In contrast to the `REG_Settings`, the `REG_Status` is a read only register from the processor. It carries errors and others status flags explained bellow.

31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
x	x	x	x	x	x	x	x
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
x	x	x	x	x	x	x	x
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
x	Receiver full	Receiver empty	Receiver sclear	x	Receiver clear error decoder	Receiver error decoder	x
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
x	x	x	x	Sender finished sending	x	x	Sender is fifo full

- Bits 31 through 15 are not in use, instead being reserved for future implementations.
- Bit 14 - `Receiver full` stores the full flag for the Receiver FIFO;
- Bit 13 - `Receiver empty` stores the empty flag for the Recevier FIFO;
- Bit 12 - `Receiver sclear`stores the clear flag for the Receiver FIFO;
- Bit 10 - `Receiver clear error decoder` stores the clear error flag for the Receiver decoder;
- Bit 9 - `Receiver error decoder` stores the error flag for the Receiver decoder;
- Bit 3 - `Sender finished sending` stores the finished sending flag for the Sender FIFO;
- Bit 0 - `Sender is fifo full` stores the full flag for the Sender FIFO;

4.3.7 Offset 4 - REG_Receiver_data_out (R)

The Register `Receiver data out` is the output of the Receiver peripheral from the IP core. This Register carries a package built from the signal received, this package comes from the FIFO in which it was stored. A combination of one or more packages read from this register makes a command.

This register is written from least significant (0) to most significant bit(31) of the register. In which the least six significant bits are the mask that carries the size of the data and the remaining 26 are reserved for data itself.

31	**30**	**...**	**2**	**1**	**0**
x	x	...	x	x	x

4.3.8 FIFO Package count Registers

The next two registers store the number of packages in their respective FIFOs. The former being the Sender FIFO and the latter being the Receiver FIFO.

Both registers are automatically filled by the Intel FIFO and are just read by the C software through the Avalon bus.

Offset 5 - REG_Sender_usedw (R)

Stores the usedw variable for the Sender FIFO, which specifies the amount of words currently stored in their memory.

7	**6**	**...**	**2**	**1**	**0**
x	x	...	x	x	x

Offset 6 - REG_Receiver_usedw (R)

Stores the usedw variable for the Receiver FIFO, which specifies the amount of words currently stored in their memory.

7	**6**	**...**	**2**	**1**	**0**
x	x	...	x	x	x

4.3.9 Offset 7 - REG_IP-ID (R)

This register stores the ID of the IP core so that is possible to verify if the software is communicating properly with the IP core and if it is correctly addressing the Avalon bus. The binary value above can be read in Hex as `0XFF0055FF`, and it is written to the Register IP-ID from least significant (0) to most significant bit (31).

31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
1	1	1	1	1	1	1	1
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
0	0	0	0	0	0	0	0
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
0	1	0	1	0	1	0	1
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	1	1	1	1	1	1

5. Firmware

This section contains an explanation of the firmware/code of this project, which is centered around the Nios II soft processor.

5.1 Nios II

The Nios II is a soft processor, which means that, unlike discrete processors, such as those in conventional computers, its peripherals and addressing can be reconfigured on demand. This enables the development of a specialized and efficient processor, reducing the costs and time of producing a prototype since it is dynamically generated inside the FPGA without the need to produce a new processor.

Communication between the Nios II and the peripheral IP core is done via the Avalon data bus, which is a memory-mapped peripheral. The addressing works as in a common memory, having write, read, and address signals, as well as the input and output vectors of this bus.

The Nios II function is to write the commands and tests in the register banks present in the IP peripheral, so that it can communicate with the tag. This processor can be viewed as the conductor and all other components as the orchestra, as it is responsible for enabling, configuring, reading, and writing data from the Avalon memory to the IP core.

Throughout this project, commands are separated into packages for ease of use. Details on how this is done can be found here.

5.1.1 File Hierarchy

All necessary C and header files are located in the project's `fpga/software/rfid_test` folder. The top entity of the entire processor (including all the required configuration generics) is the `main.c` file, with all other relevant files present inside the helpers folder.

```
main.c                  - Nios II soft processor top entity
|
└/helpers               - Holds all complimentary C files
    |
    ├/functions           - Holds all functions that dictate how the components act
    |   ├sender.c           - Holds all functions that dictate how the sender acts
    |   ├receiver.c         - Holds all functions that dictate how the receiver acts
    |   └rfid.c             - Holds all functions about Tari, commands and packages
    |
    ├config.h              - Stores all the needed defines
    |
    ├crc.c                 - Cyclic Redundance Check file
    |
    └/commands            - Holds all the EPC-GEN2 mandatory commands
        ├ack.c              - Mandatory command ack
        ├kill.c             - Mandatory command kill
        ├lock.c             - Mandatory command lock
        ├nak.c              - Mandatory command nak
        ├query.c            - Mandatory command query
        ├query_adjust.c     - Mandatory command query_adjust
        ├query_rep.c        - Mandatory command query_rep
        ├read.c             - Mandatory command read
```

```
├req_rn.c              - Mandatory command req_rn
├rn16.c                - Mandatory command rn16
├rn_crc.c              - Mandatory command rn_crc
├select.c              - Mandatory command select
├write.c               - Mandatory command write
```

Additional information on the EPC-GEN2 protocol and mandatory commands (as well as the other command types) can be found here.

5.2 Main code

/main/fpga/software/rfid_test/main.c

The software was developed in a way that it would be easy to understand it. It is responsible for calling functions, defines and commands structures from the `/helpers` folder, advantages that are present due to the C programming language being considered high-level. These features are all called in the `main.c` code, which is responsible for controlling and monitoring the IP core.

In the main code, the user can choose which commands they want to send to the tag, to see if it responds properly according to the EPC-GEN2 protocol. It is also possible to make timing based tests by varying the Tari values, to check if the tag will respond accordingly. This also allows the user to implement new commands, such as proprietary or custom ones.

The group has also prepared a set of examples in which this code will be used by the final user. For example, if the user wants to test their version of the reader or even make a simulation in ModelSim, they must copy the `loopback_handshake.c` (located here) file from the `fpga/examples_of_main` folder to the main code located in the `main.c` file. Once this code is run, the user will be able to see and test different parameters, which will be shown later in the "Example of main code" section. This folder can be found here.

Besides the loopback_handshake.c, the other examples of codes that can be used in the `main.c` file are the `reader.c`, the `tag.c` and the `test_individual_commands_loopback.c`, all of which are variations that are capable, respectively, of acting as a reader, emulating a tag and even testing the send/receive functionalities of a single command in a single DE-10 Standard board.

5.3 `config.h` - Starting Variables

Inside the `/helpers` folder, the first group to be explained are the defines that declare the addresses in which the IP core interface (previously mentioned here) acts. In this file, there are also defines that store the default values for package, command and mask sizes. The following section explains them in more detail.

5.3.1 Register Status

```
//READ ONLY
define BASE_IS_FIFO_FULL    (1 << 0)
define MASK_EMPTY_RECEIVER  (1 << 13)
```

- `BASE_IS_FIFO_FULL` - Indicates the necessary shift for the `is_FIFO_full` flag
- `MASK_EMPTY_RECEIVER` - Indicates the necessary shift for the `is_receiver_empty` flag

5.3.2 Register Settings

```
//READ/WRITE
define BASE_REG_SET         (0)
define MASK_RST             (1 << 0)
define MASK_EN              (1 << 1)
define MASK_RST_RECEIVER    (1 << 10)
define MASK_EN_RECEIVER     (1 << 4)
define MASK_CLR_FIFO        (1 << 2)
define MASK_LOOPBACK        (1 << 8)
define MASK_CLR_FINISHED    (1 << 1)
define SENDER_HAS_GEN       (0 << 5)
define SENDER_ENABLE_CTRL   (1 << 6)
define SENDER_IS_PREAMBLE   (0 << 7)
define MASK_READ_REQ        (1 << 12)
define MASK_FINISH_SEND     (1 << 3)
```

- `BASE_REG_SET` - Memory address for the `REGISTER_SETTINGS`
- `MASK_RST` - Indicates the necessary shift for the `sender_reset` flag
- `MASK_EN` - Indicates the necessary shift for the `sender_enable` flag
- `MASK_RST_RECEIVER` - Indicates the necessary shift for the `receiver_reset` flag
- `MASK_EN_RECEIVER` - Indicates the necessary shift for the `receiver_enable` flag
- `MASK_CLR_FIFO` - Indicates the necessary shift for the `sender_clear_FIFO` flag
- `MASK_LOOPBACK` - Indicates the necessary shift for the `RFID_loopback` flag
- `MASK_CLR_FINISHED` - Indicates the necessary shift for the `sender_clear_finished` flag
- `SENDER_HAS_GEN` - Indicates the necessary shift for the `sender_has_generator` flag
- `SENDER_ENABLE_CTRL` - Indicates the necessary shift for the `sender_enable_controller` flag
- `SENDER_IS_PREAMBLE` - Indicates the necessary shift for the `sender_is_preamble` flag
- `MASK_READ_REQ` - Indicates the necessary shift for the `receiver_read_request` flag
- `MASK_FINISH_SEND` - Indicates the necessary shift for the `mask_finish_send` flag

5.3.3 RFID - Addresses

```
define BASE_REG_TARI       (1)
define BASE_REG_FIFO       (2)
define BASE_REG_TARI_101   (3)
define BASE_REG_TARI_099   (4)
define BASE_REG_TARI_1616  (5)
define BASE_REG_TARI_1584  (6)
define BASE_REG_PW         (7)
define BASE_REG_DELIMITER  (8)
define BASE_REG_RTCAL      (9)
define BASE_REG_TRCAL      (10)
define BASE_REG_STATUS     (3)
define BASE_RECEIVER_DATA  (4)
define BASE_SENDER_USEDW   (5)
```

```
define BASE_RECEIVER_USEDW (6)
define BASE_ID             (7)
```

- `BASE_REG_TARI` - R/W - address of the tari
- `BASE_REG_FIFO` - R/W - address of FIFO R/W
- `BASE_REG_TARI_101` - W - address of tari_101
- `BASE_REG_TARI_099` - W - address of tari_099
- `BASE_REG_TARI_1616` - W - address of tari_1616
- `BASE_REG_TARI_1584` - W - address of tari_1584
- `BASE_REG_PW` - W - address of pw
- `BASE_REG_DELIMITER` - W - address of delimiter
- `BASE_REG_RTCAL` - W - address of receiver transmitter calibration
- `BASE_REG_TRCAL` - W - address of transmitter receiver calibration
- `BASE_REG_STATUS` - R - address of `REGISTER_STATUS`
- `BASE_RECEIVER_DATA` - R - address of receiver data
- `BASE_SENDER_USEDW` - R - address of `sender_FIFO_actual_size`
- `BASE_RECEIVER_USEDW` - R - address of `receiver_FIFO_actual_size`
- `BASE_ID` - R - address of IP core

5.3.4 RFID - Command specifications

```
// package defines
define data_mask_size      (6)
define data_package_size   (26)
define eop                 (0x000000000)
define bits6               (0x3F)
define bits26              (0x3FFFFFF)
define bits32              (0xFFFFFFFF)
```

- `data_mask_size` - defines the number of bits reserved for the mask
- `data_package_size` - defines the number of bits reserved for the data
- `eop` - defines the `END_OF_PACKAGE` format
- `bits6` - mask for full package mask
- `bits26` - mask for full package data
- `bits32` - mask for full package

5.4 Commands

/main/fpga/software/rfid_test/helpers/commands

The group has implemented all the mandatory commands (described in the Mandatory Commands subsection page). A couple of them still need to be validated, but the ones that are necessary for a full handshake have been implemented and thoroughly tested.

5.4.1 Command Struct and CRC

Both the Command Struct and the CRC files are separate from the rest of the commands, because they are not commands, but rather act in building each command, which depends on the protocol's requirements of the command.

Command Struct

/main/fpga/software/rfid_test/helpers/commands/command_struct.h

The first is the Command Struct, which establishes the base struct that every command will have in common. It is composed by the command's size and data, as shown in the code block below.

```
typedef struct
{
    int size;
    unsigned long long result_data;
} command;
```

Cyclic Redundancy Check - CRC-5/CRC-16

/main/fpga/software/rfid_test/helpers/commands/crc.c

Secondly, the CRC code was designed by the previous group. It was designed by BarrGroup[1], a verified hardware site, which is cited and documented properly inside the file.

The CRC implemented by them is the same that the EPC-GEN2 documentation requires, and as it is an open-source code, just like this project, the group maintained and made use of it.

The following code block shows one of the functions used in the CRC code.

```
void crc_16_ccitt_init(void)
{
    unsigned short polynomial = POLYNOMIAL_16;
    crc16 remainder;
    int dividend;
    unsigned char bit;

    // Compute the remainder of each possible dividend.
    for (dividend = 0; dividend < 256; ++dividend)
    {
        // Start with the dividend followed by zeros.
        remainder = dividend << 8;

        // Perform modulo-2 division, a bit at a time.
        for (bit = 8; bit > 0; --bit)
        {
            // Try to divide the current data bit.
            if (remainder & 0x8000)
            {
                remainder = (remainder << 1) ^ polynomial;
            }
            else
            {
                remainder = (remainder << 1);
            }
        }

        // Store the result into the table.
        crc_table[dividend] = remainder;
    }
}
```

5.4.2 Mandatory commands

/main/fpga/software/rfid_test/helpers/commands/

Table of commands

This section contains a table of the mandatory commands (described in the Mandatory Commands subsection page) with their respective statuses. In it, there are four columns:

- tested: these are the commands that were sent and received properly;
- validated: in this column are present the commands that were built solely around the EPC-GEN2 protocol specifications;
- functional: these are the commands that are already being correctly interpreted once sent or received by the tag or by the reader;
- ToDo: these are commands that have some issue concerning validation/functionality. These commands are present as issues in the GitHub repository.

commands	tested	validated	functional	ToDo
Ack	x	x	x	
Kill	x			x
Lock	x			x
Nak	x	x	x	
Query	x	x	x	
Query_adjust	x	x	x	
Query_rep	x	x	x	
Read	x	x		
Req_rn	x	x	x	
Rn16	x	x	x	x
Rn_crc	x	x	x	
Select	x			x
Write	x	x		

Example of command build: the `ack` command

/main/fpga/software/rfid_test/helpers/commands/ack.c

The code block below shows an example of a command (in this case, the `ack` command):

```
#include "ack.h"

void ack_build(command *ack, int rn)
{
    ack->result_data = ((ACK_COMMAND & 0b11) << 16) | (rn & 0xFFFF);
    ack->size = ACK_SIZE;
}

int ack_validate(int packages[], int command_size)
{
    if (command_size != ACK_SIZE && command_size != ACK_SIZE + 1)
```

```
        return 0;

    // |       packages[0]      |
    // |  command  |  rn/randle  |
    // |    X*2    |     X*16    |

    int command = (packages[0] >> 16) & 0b11;
    if (command != ACK_COMMAND)
        return 0;

    return 1;
}
```

5.5 Functions

5.5.1 `main.c` - First declaration of the time parameters

/main/fpga/examples_of_main/

In each variation of the main code present inside the `fpga/examples_of_main`, a sample of code is common between them all: these are all of the time parameters mentioned in the Signal Generator section.

```
int tari_100  = rfid_tari_2_clock(10e-6, FREQUENCY);
int pw        = rfid_tari_2_clock(5e-6, FREQUENCY);
int delimiter = rfid_tari_2_clock(62.5e-6, FREQUENCY);
int RTcal     = rfid_tari_2_clock(135e-6, FREQUENCY);
int TRcal     = rfid_tari_2_clock(135e-6, FREQUENCY);
```

- `tari_100` - tari time parameter
- `pw` - pw parameter
- `delimiter` - Delimiter parameter
- `RTcal` - Receiver transmitter calibration parameter
- `TRcal` - Transmitter receiver calibration parameter

5.5.2 `rfid.c`

/main/fpga/software/rfid_test/helpers/functions/rfid.c

The RFID code's first appearance is in the main code, because it stores the functions that set all the needed parameters so that the test can be launched. Some of these functions set the time parameters, others are designed to interpreting the mask of the package, and the last few ones check if the answer received is a valid command. They are all described in the code block below:

RFID functions

```
void rfid_set_loopback()
void rfid_set_tari(int tari_value)
void rfid_set_tari_boundaries(int tari_101, int tari_099, int tari_1616, int tari_1584, int pw, int delimiter, int RTcal, int TR
int rfid_create_mask_from_value(int value)
int rfid_check_command(int *packages, int quant_packages, int command_size)
```

```
int rfid_get_ip_id()
int rfid_tari_2_clock(double tari, double clock)
```

- `void rfid_set_loopback` - Connects Tx on Rx, creating a loop. Used for testing the reader.
- `void rfid_set_tari` - Sets the Tari value on the IP core.
- `void rfid_set_tari_boundaries` - Sets the Tari boundaries on the IP core.
- `int rfid_create_mask_from_value` - Generates the package's mask based on the package received.
- `int rfid_check_command` - Checks if the received command is valid and present on the EPC-GEN2 protocol.
- `int rfid_get_ip_id` - Checks the id of the IP core.
- `rfid_tari_2_clock` - Calculates the tari parameter based on the microprocessor clock.

5.5.3 `sender.c`

/main/fpga/software/rfid_test/helpers/functions/sender.c

This file is responsible for controlling the sender peripheral. It has functions that can enable the peripheral, properly format the commands in order to be sent to the tag and send the completed command. The code block below gives a little more insight on them.

Sender functions

```
int  sender_check_usedw()
int  sender_check_fifo_full()
void sender_enable()
void sender_send_package(int package)
void sender_send_end_of_package()
void sender_start_ctrl()
void sender_write_clr_finished_sending()
int sender_read_finished_send()
int sender_get_command_ints_size(int size_of_command)
void sender_add_mask(int n, int command_vector_masked[n], unsigned long long result_data, unsigned int result_data_size)
void sender_has_gen(int usesPreorFrameSync)
void sender_is_preamble()
void sender_send_command(command *command_ptr)
```

- `sender_check_usedw` - Accesses the address that indicates how many packages are in the sender's FIFO.
- `sender_check_fifo_full` - Accesses `REG_STATUS` to verify whether the FIFO is full or not.
- `sender_enable` - Accesses `REG_SET` to activate the sender peripheral on the IP core.
- `sender_send_package` - Writes the package on the FIFO address.
- `sender_send_end_of_package` - Writes the EOP on the FIFO address.
- `sender_start_ctrl` - Accesses `REG_SET` to activate the sender controller with a pulse.
- `sender_write_clr_finished_sending` - Accesses `REG_SET` to clear the `finished_sending` flag with a pulse.
- `sender_read_finished_send` - Accesses `RES_STATUS` to check whether the package has been sent or not.
- `sender_get_command_ints_size` - Checks the size of the command and calculates how many packages will be sent.
- `sender_add_mask` - Divides the command into smaller packages if needed and generates a mask based on the current package data size.
- `sender_has_gen` - Accesses `REG_SET` to define whether the signal generator should be activated.
- `sender_is_preamble` - If the generator is activated, defines if the signal generator is a preamble or a frame sync.
- `sender_send_command` - Runs the all the previous functions related to the command, going through all the steps necessary to split it into packages, add the masks, send and clear the flag registers in the end.

5.5.4 `receiver.c`

/main/fpga/software/rfid_test/helpers/functions/receiver.c

This file stores all the functions required to retrieve data from the IP core, which in turn consist of the functions that enable the receiver peripheral, check the FIFO for data and also request a new package. The code block gives a little more information about these functions.

Receiver functions

```
void receiver_enable()
int receiver_check_usedw()
int receiver_request_package()
int receiver_empty()
void receiver_rdreq()
void receiver_get_package(int *command_vector, int quant_packages, int *command_size, int *quant_packages_received)
```

- `receiver_enable` - Accesses `REG_SET` to activate the receiver peripheral on the IP core.
- `receiver_check_usedw` - Accesses the address that indicates how many packages are in the receiver's FIFO.
- `receiver_request_package` - Accesses `BASE_RECEIVER_DATA` to read the received package.
- `receiver_empty` - Accesses `REG_SET` to check whether the receiver FIFO is empty or not.
- `receiver_rdreq` - Accesses `REG_SET` to set the `read_request` flag with a pulse.
- `receiver_get_package` - Separates the package from `receiver_request_package` into data and mask.

5.6 Example of main code

As mentioned in the Main code subsection, there are a set of examples already implemented in which the user can work on tests and communication between reader and tag. All those files are located in the `fpga/examples_of_main` folder and in this section the code will be thoroughly described so that the functionality of the project can be clarified.

For this example, the test_individual_commands_loopback.c file was chosen, because it is succinct and it suffices in showing a simple communication in loopback mode.

Firstly is the header of the code, which brings all the necessary includes to this test. The first one present is the `io.h`: this is the Nios II include that establishes the communication with the IP core interface; after that is `system.h`, which helps bring the functionalities of the FPGA. The following includes are the proprietary codes of the functions and the commands used. The code block below shows these includes.

```
#include "io.h"
#include "system.h"
#include "stdint.h"
#include "helpers/commands/commands.h"
#include "helpers/functions/functions.h"
#include "stdio.h"
#include "helpers/config.h"
```

Following the includes is the main function, and inside it are the first declaration of the time parameters, in which the time parameters are calculated, followed by the definition of the loopback function that indicates a

test that is implemented using a single DE-10 Standard board and the setting of the signal generator. Also present are the enables of the sender and receiver peripherals, as both of them will be needed for this test.

```
// Time parameters
int tari_100  = rfid_tari_2_clock(10e-6, FREQUENCY);
int pw        = rfid_tari_2_clock(5e-6, FREQUENCY);
int delimiter = rfid_tari_2_clock(62.5e-6, FREQUENCY);
int RTcal     = rfid_tari_2_clock(135e-6, FREQUENCY);
int TRcal     = rfid_tari_2_clock(135e-6, FREQUENCY);
//configurations-------------------------------------------------------------------------------
rfid_set_loopback();
rfid_set_tari(tari_100);
sender_enable();

receiver_enable();
rfid_set_tari_boundaries(tari_100 * 1.01, tari_100 * 0.99, tari_100 * 1.616, tari_100 * 1.584, pw, delimiter, RTcal, TRcal);
sender_has_gen(0);
//sender_is_preamble(); // NOTE: enable this function if implementing RFID tech
```

Disclaimer: the `sender_has_gen` function is set to disable the signal generator, and the `sender_is_preamble` (which is responsible for defining if the signal generated will be a preamble or a frame sync, previously mentioned here) is commented out in the code due to the team's decision of not implementing the radio frequency part of the project (explained in the Project Overview section), but it is coded in case the radio frequency communication is implemented so that the user can know where to set these functions.

Now it is necessary to send the desired command to be tested and validated. So, in the following code, an ack command will be built and sent.

```
    printf("==============================\n");
    printf("==       TEST COMMAND       ==\n");
    printf("==============================\n");

    command rn16;
    rn16_build(&rn16);

    command ack;

    ack_build(&ack, rn16);

    printf("sending ack\n");

    sender_send_command(&ack);

    printf("sent ack\n\n");
```

The ack command also needs a random number in its build, hence why it was also instantiated.

Last, it is necessary to read the IP core for the command that has been sent in order to check if it has been properly received, as is shown the following code block.

```
    int quant_packages = 3;
    int command_size = 0;
    int packages[quant_packages];

    printf("waiting for packages\n");

    if (receiver_get_package(packages,quant_packages, &command_size) == -1)
    {
```

```
        printf("exiting on receiver\n");
        return 1;
    }

    printf("command_size: %d\n", command_size);
    int label = rfid_check_command(packages, command_size);
    printf("label is: %d\n\n", label);
```

In sum, this code shows the whole process: the command to be sent, it being received and also validated.

For more information on how to run this (and the other examples of main code) and get the outputs for it, see the Testing / Running section.

---

1. BarrGroup. https://barrgroup.com Accessed on: 05/10/2021. ↩

6. Testing the components

Due to the complexity of our `.vhd` files, they had to be extensively tested individually. This was done through the ModelSim software, which enabled us to simulate values and compare the results with the expected ones.

6.1 Overview and testing order

The first step was to test each component individually to see if they fit to the specifications of the protocol. Once each component was tested and validated, the next step was to implement the interaction between the components using ModelSim to visualize if the components were working together properly.

After running the necessary simulations, a hardware test was implemented, in which the sender was tested firstly and separately from the receiver component in order to validate the signal transmitted. To do so, a Saleae Logic 8 Analyzer[1] was used in conjunction with the Logic 2 Software to visualize the signal transmitted by the selected DE-10 Standard board pin. Once the transmitted signal had been validated, the next step was to use the board to send the signal via one pin and receive it on another pin.

Lastly, after validating that the hardware implementation worked on a single board, the team decided to test the same communication, but using two boards: one acting as a tag and another acting as a reader. To do so, the transmitter of the tag was plugged into the reader's receiver, and the reader's transmitter was plugged into the tag's receiver. Both boards' ground pin were also connected so that both could share the same ground signal, otherwise the communication would not work.

6.2 Individual testing

This section contains more information on how each component was individually tested.

6.2.1 Sender

/main/fpga/RTL/tb/sender_tb.vhd

The `sender_tb.vhd` file tests the sender component by simulating a series of commands, each divided in packages, to be encoded and sent. Each component was manually inserted into the testbench, as the group knew what the sender output should be once the commands had been encoded, and compared the results obtained.

FMO Encoder

/main/fpga/RTL/tb/FM0_encoder_tb.vhd

The `FM0_encoder_tb.vhd` file tests the sender encoder component in regards to the encoding process and FIFO communication.

The encoding process tests happen in the same way as the sender tests, in which the program simulates a series of commands, each divided in packages. Each component was manually inserted into the testbench, as the group knew what the encoder output should be once the commands had been encoded, and compared the results obtained.

The FIFO communication test by simulating a FIFO component, and the testbench compares the values of the communication cable to their intended results.

Sender Controller

/main/fpga/RTL/tb/sender_controller_tb.vhd

The sender_controller_tb.vhd file tests the sender controller component by simulating the flags that dictate its functionalities, those being the `has gen`, `start encoder`, `start generator`, `clr_finished sending`.

Signal Generator

/main/fpga/RTL/tb/signal_generator_tb.vhd

The signal_generator_tb.vhd file tests the sender signal generator component by simulating the flags that dictate its functionalities, those being the `has gen`, `start generator`, `is_preamble`.

6.2.2 Receiver

/main/fpga/RTL/tb/receiver_tb.vhd

The receiver_tb.vhd file tests the receiver component by simulating a series of commands, each divided in packages, as a response being sent from the tag. Each component was manually inserted into the testbench, as the group knew what the sender output should be once the commands had been decoded, and compared the results obtained.

FM0 Decoder

/main/fpga/RTL/tb/FM0_decoder_tb.vhd

The FM0_decoder_tb.vhd file tests the receiver decoder component by simulating a series of commands, each divided in packages. Each component was manually inserted into the testbench, as the group knew what the decoder output should be once the commands had been decoded, and compared the results obtained.

Package Constructor

/main/fpga/RTL/tb/FM0_decoder_tb.vhd

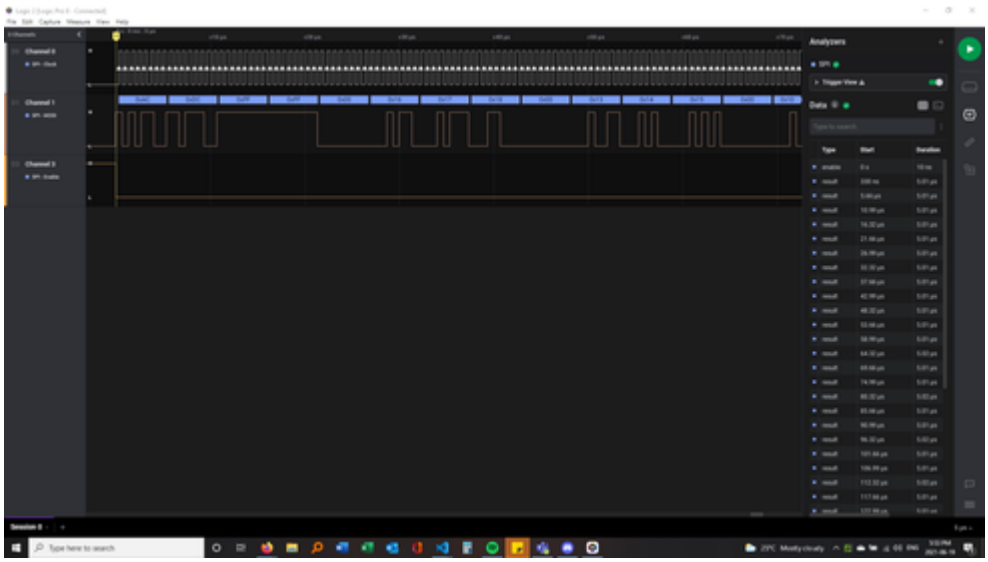
The package_constructor_tb.vhd file tests the receiver package constructor component by simulating a decoder component that sends decoded data to the package constructor, which once correctly packaged should result in full commands that are compared to the expected command outputs.

6.3 Saleae Logic 8 Analyzer

The Saleae Logic 8 Analyzer is a small analyzer that can be plugged into the user's computer. It can be set up to run on loop, run during a set period of time or set to wait for a trigger of a specific signal. The user can visualize the signal and choose their preferred method for visualizing the signal using the Logic 2 Software.

The images below show the Saleae Logic 8 Analyzer and the Logic 2 Software respectively.

Logic 8 Analyzer (image obtained here)

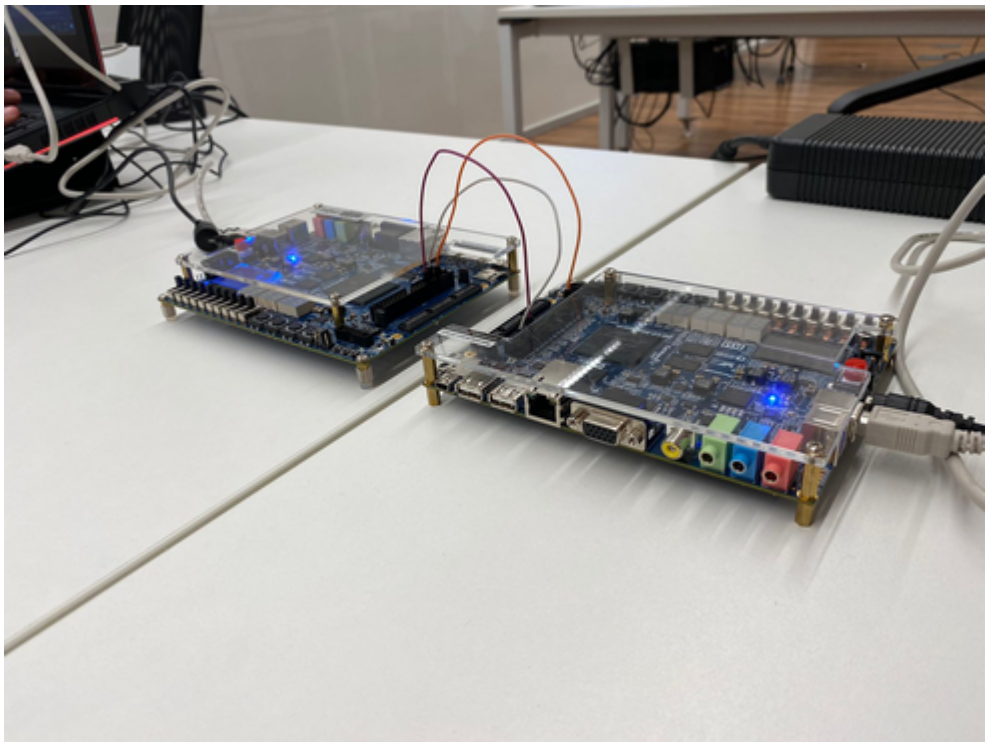


Logic 2 Software (image obtained here)

6.4 Handshake between two boards

The steps on how to do this implementation can be read here.

The purpose of this implementation is to validate if two boards can succesfully communicate using the implemented mandatory commands to do so. A more detailed explanation of how this works can be found in the example of main code section. The image below shows the two boards and their connections.



Handshake implementation setup with the two DE-10 Standard boards connected

6.5 How to run any `.vhd` testbench

The video below shows a visual example of how the `.vhd` testbenches were validated using ModelSim (in this case, the `sender_tb.vhd`).

---

1. Saleae Logic 8 Analyzer. https://www.saleae.com/ Accessed on: 01/10/2021. ↩

7. Conclusion

The main purpose of the project was to develop a conformance tester for the EPC-GEN2 protocol TAGs. Doing so required thorough understanding of most of the EPC-GEN2 UHF RFID protocol throughout each member of the group; development of complex VHDL components that operated through state machines; development of a NIOS II soft processor in C programming language that included all of the protocol's mandatory commands, as well as other customizations for testing purposes; as well as building the Avalon Interface that intermediates the processor and IP core communication.

The group as a whole had to do an extensive research on all these topics to learn how to implement the code, as well as understand the protocol to have a clear image of what the conformance tester should be evaluating in a TAG.

8. The team

This page is to give a brief description of the team responsible for the development of this project.

8.1 Alexandre Almeida Edington

GitHub profile: https://github.com/Alexandreae

Bio: Computer engineering student in the 9th period, passionate about front-end development and user experience. Mainly responsible for this project's GitHub-pages and Doxygen documentation.

8.2 Bruno Signorelli Domingues

GitHub profile: https://github.com/BrunoSDomingues

Bio: Computer engineering student in the 8th semester, interested in backend and data science. Listens to music and games in the spare time. Mainly responsible for the hardware implementation of the project in the DE-10 Standard Board.

8.3 Lucas Leal Vale

GitHub profile: https://github.com/lucaslealvale

Bio: Computer engineering student in the 8th period, Hardware enthusiast and passionate to it, interested in backend development. Mainly responsible for the Nios soft processor and software implementations of this project.

8.4 Rafael dos Santos

GitHub profile: https://github.com/4rfel

Bio: Computer engineering student in the 8th period, interested in game development and hardware advances. Mainly responsible for the VHDL development in this project.