Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione
"Guglielmo Marconi"

Corso di Laurea magistrale in
Ingegneria Elettronica e Telecomunicazioni per l'Energia

titolo della tesi:

DIGITAL DESIGN OF AN EPC GEN2 CONTROLLER FOR ENHANCED RFID TAGS

Tesi di laurea magistrale in
Elettronica dei sistemi digitali

Relatore:
ALDO ROMANI

Presentata da:
NICHOLAS BATTISTINI

Correlatori:
DAVIDE FABBRI
MATTEO PIZZOTTI

II Appello - II Sessione
Anno accademico 2018/2019

Contents

Chapter 1

Introduction

This thesis presents an improvement of the long range battery-less UHF RFID platform for sensor applications[1] which is based on the open source Wireless Identification and Sensing Platform (WISP) project[2]. The purpose of this work is to design a digital logic that performs the RFID EPC gen2 protocol[3] communication, is able to acquire information by sensors and provide an accurate estimation of tag location ensuring low energy consumption. This thesis will describe the hardware architecture on which the digital logic was inserted, the verilog code developed, the methods by which the digital logic was tested and an explorative study of chip synthesis on Cadence.
Design has been based on a modular and easily extensible verilog open source code [4] described in [5][6] which implements a part of RFID EPC gen2 protocol. This code has been improved by interfacing an ultra wide band (UWB) backscattering system, an external memory and a sensor through the serial protocol interface (SPI) and adding the possibility of read and write data located in the external non volatile memory.
UWB localization system has been chosen to obtain tag position with high accuracy. UHF communication can ensure an estimate of tag location through the measurement of round trip time or power strength received by the reader involved in the communication. However, this technique provides only a one-dimension positioning with an accuracy of some meters. Since the operative range of these tags is about from 2 to 10 meters, the accuracy ensured by UHF communication isn't sufficient for a reliable localization. UWB systems offer a cm level accuracy and through the use of at least three UWB transmitter, they guarantee a three-dimensional positioning. However this implies the presence of a UWB on board antenna which increases tag's size.

Chapter 2

RFID Technology

Radio frequency identification (RFID) systems are based on the information exchange between one or more interrogators (reader) and one ore more labels (tag). In simplest application, information communicated by the tags are only identifiers, whereas in advanced application tags are able to communicate memory data. Memory data may be information obtained by sensors or data written by readers.
RFID tags differentiate through:

- kind of power supply:
  - **Active**: tags are equipped with a battery that supplies them entirely. An active transmitter is used to amplify and send a response to reader's queries.
  - **Passive**: tags have no battery and are supplied through the conversion from RF to DC of reader's transmitted signals. They use a passive transmitter, that is a system which replies to reader's queries by backscattering the same signal sent by reader.
  - **Semi passive**: Tags are equipped with battery that supplies the receiver and the control logic. The transmitter is the same of passive tags.
- Operating principle:
  - **Electromagnetic**: The communication occurs in far field region through the transmission of electromagnetic waves.
  - **Inductive coupling**: The communication occurs in near field region through inductive coupling.

	Frequency	standard	Operating principle	Power Supply	range	Data transfer	Application	features	cost
LF	125kHz 134.2kHz	ISO 18000-2	Inductive coupling	Mainly passive tags	Max 0.5m	Until 1kbit/s	·Animal and vehicle identification ·Access control ·Car immobilizer	Able to communicate through liquid Sensitivity to antenna orientation	Generally expensive, variable with application
HF	13.56MHz	ISO 18000-3, ISO15693 (*Vicinity*), ISO14443 (*Proximity*), EPC 13.56class1	Inductive coupling	Mainly passive tags	Max 1.5m	About 25 kbit/s	·Smart card ·Ticket ·Baggage handling	Able to communicate through liquid Limited tag number	40/60 €cent
UHF	865.6÷867.6; 868÷869 MHz in EU 902÷928MHz in USA, 950MHz in Japan	18000-6 type C, EPC Class0/1 Gen1, EPC Class1 Gen2	Electromagnetic	Both passive and active	Until 12m (passive) about 100m (active)	From 28kbit/s to 128kbit/s	·Logistics ·Baggage control	Low performance through metal and liquid Suitable for large group of tags	20/40€cent
Microwave	2.4 2.483MHz in EU 2.4 2.5GHz in US	ISO 18000-4 mode 1/2	Electromagnetic	Both passive and active	About 1m(passive) about 30m(active)	30/40kbit/s	·Logistics ·Tool collection	Unable to communicate through liquid Fast read operation Crowded working band	Expensive

Figure 2.1: RFID technology summary features

- Operating frequency:
  - LF $125/134.2kHz$
  - HF $13,56MHz$
  - UHF $860 \div 960MHz$
  - Microwaves $2.4 \div 2.5GHz$

The operating frequency is defined as reader's transmission frequency, uplink and downlink bit-rate may be different. On table 2.1 are summarized the information about RFID technology.
Many industrial applications are conformed with the UHF EPC gen2 standard because of the good trade off between cost, size and operative distance of tags. Generally tag's cost increases with the operating frequency, sizes depend mostly on the antenna which scales with the working frequency. Operative distance depends on supply, operating principle and frequency since the ability of electromagnetic waves to cross obstacle decrease with increasing frequency.

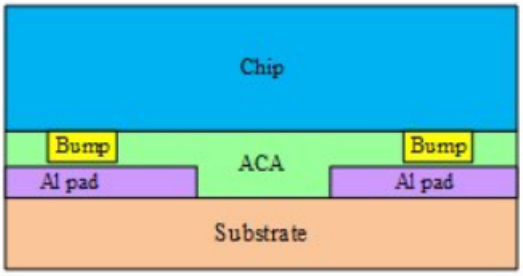

Figure 2.2: Layout of a generic RFID tag [7]

A generic layout of an RFID tag is shown in figure 2.2. The silicon chip is placed on a conductive, anisotropic, adhesive layer (ACA) which rests on a flexible substrate. the chip is flip-chip assembled on the substrate, through the ACA, applying high pressure at high temperature.

Chapter 3

UWB positioning

Ultra wide band (UWB) is a technology for short-range radiocommunication which uses signals that spread over a large frequency range. UWB operates with a low transmit power (0.5 mW) and signals involved ensure a low power spectral density (-41.3dBm/MHz). The frequency range of UWB is between 3.1 and 10.6 GHz with a bandwidth of at least 500MHz. An example of UWB antenna, signal waveform and spectrum are shows in figures 5.3, 3.1.
Thanks to its robustness against interference and its high channel capacity, UWB is suitable for audio, video, data transmission and indoor positioning.
UWB localization can be performed through active and passive approach. Active approach implies that the object to be located generates and transmits UWB signals. In passive approach the object merely backscatters, with an appropriate modulation, UWB signals received by an external transmitter. Thanks to the large bandwidth of spectrum, UWB indoor positioning systems are able to pinpoint location in real time with a cm level accuracy.
However these systems have several drawbacks, the most relevant are costs and system synchronization. UWB hardware is about 10 times more expensive compared to a Bluetooth low energy (BLE) positioning systems. To perform a correct triangulation at least three receivers are necessary, time synchronization between them is tricky because they must be precisely synchronized down to the picosecond to calculate location accurately. Furthermore, even in UWB, systems have the potential to interfere to each other (for more information see [8]).

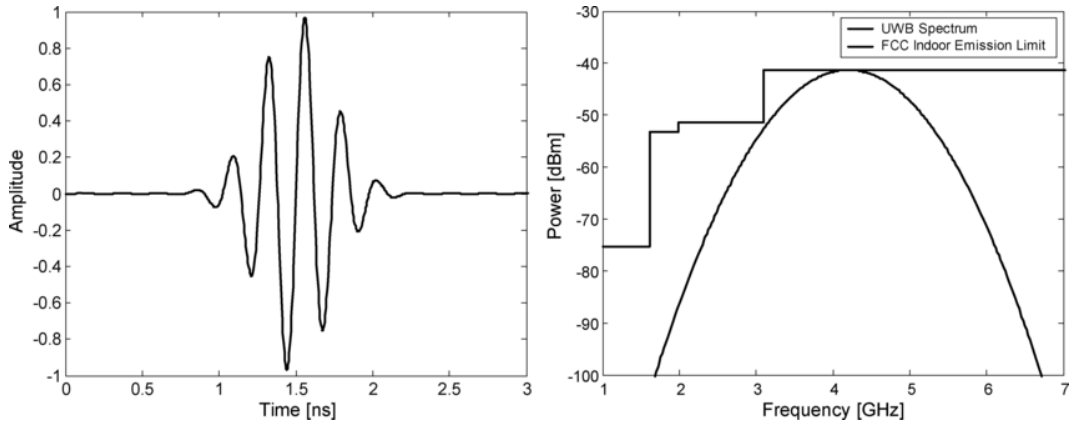

Figure 3.1: Example of UWB pulse waveform and related spectrum [8]

Chapter 4

EPC gen2 protocol

This section summarizes the main features of the protocol useful for the project. Being a prototype, this project does not implement the entire protocol. For more information about the protocol see [3].
The protocol is referred to all passive tags that work in the frequency range of 860÷960 MHz. It defines the physical layer and the logic interface. The physical layer sets the characteristic of signals exchanged by tag and reader. The logic interface sets the behavior of tag and reader.

4.1 General features

- **Reader driven communication**: The communication is half-duplex and it's always started by readers, tags interact only when they are interrogated. This is inefficient because it needs a constant polling by the reader in order to obtained information from tags.
- **No direct communication between tags**: Because of tag's low sensitivity (max -20dBm) and low retransmitted power (generally lower than -20dBm), the communication between tags can't be demodulated. Obviously, tags can exchange information between them by passing through the reader. However this kind of communication is characterized by a low capacity because it introduces high delays and an high overhead.
- **Interferences between readers**: Readers have active interfaces, so they are characterized by high sensitivity (-80dBm) and they can transmit high power signals (up to 2W for the european standard and 4W for USA). Therefore if readers aren't sufficiently spaced they require channel multiple access technique to avoid interference between them.

- **Random multiple access to the channel**: The policy of multiple access to the channel for tags, provided by the protocol, is random without carrier sensing (no channel state analysis). Readers broadcast a parameter Q which initialize, with a number within the range $(0,\ 2^Q - 1)$, a counter on each tags. Tags will not reply to queries until their counters won't reach the value 0. The counter is decremented at each reception of command QueryRep (see 4.3.3). The Q value is chosen according to the expected tag population and it's modified, while performing the communication, in order to minimize the replay time with a low collision probability.

- **Identification of message recipient**: The protocol provides the insertion of a 16 bit sequence inside almost each message in order to identify the recipient tag. This sequence is called RN16 or in some cases handle. Every message received by tags, containing an incorrect identification sequence, is rejected. The identification sequence is a random number that is communicated by tags after several protocol phases.

- **Error detection**: The protocol makes use of a cyclic code belonging to the binary Galois filed GF(2) for error detection. This code may be at 16 bit (CRC-16) or 5 bit (CRC-5), it is used to verify the integrity of many messages transmitted by readers or tags. Generator polynomials of cyclic code are used to compute and verify the CRC. An example of CRC-5 encoder/decoder which make use of parameters defined in figure 4.2, is shown in figure 4.3. The CRC computation is performed by loading the preset value in registers and applying sequentially data in DATA input. The output Q, which is the result of CRC computation, represent the CRC code inserted in many protocol messages .
  The CRC verification is performed as the computation, however the output is different. If data are not affected by error then the output Q will assume the residual value.
  CRC codes allow to quickly and easily check the integrity of a message, however they can't detect an intentional alteration. A malicious user may modify the message, compute the new CRC and include it into the message making the error detection ineffective.

- **Link timing**: The protocol defines time slots where tags or readers shall reply. These time slots are depicted in figure 4.1

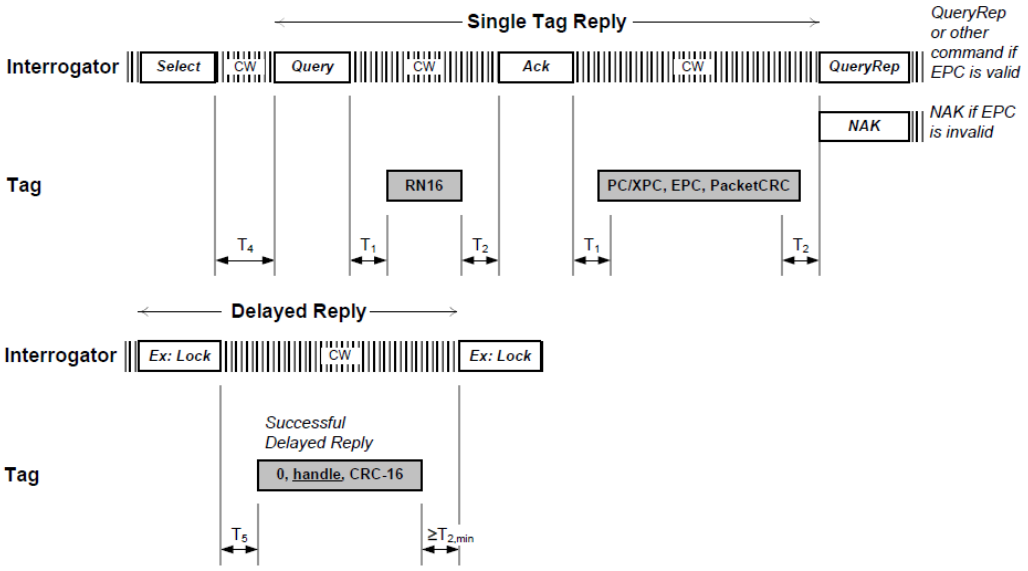

Figure 4.1: Link timing representation[3]

CRC-16 precursor				
CRC Type	**Length**	**Polynomial**	**Preset**	**Residue**
ISO/IEC 13239	16 bits	$x^{16} + x^{12} + x^5 + 1$	$FFFF_h$	$1D0F_h$

CRC-5 Definition				
CRC Type	**Length**	**Polynomial**	**Preset**	**Residue**
—	5 bits	$x^5 + x^3 + 1$	01001_2	00000_2

Figure 4.2: CRC Computation and verification parameters[3]

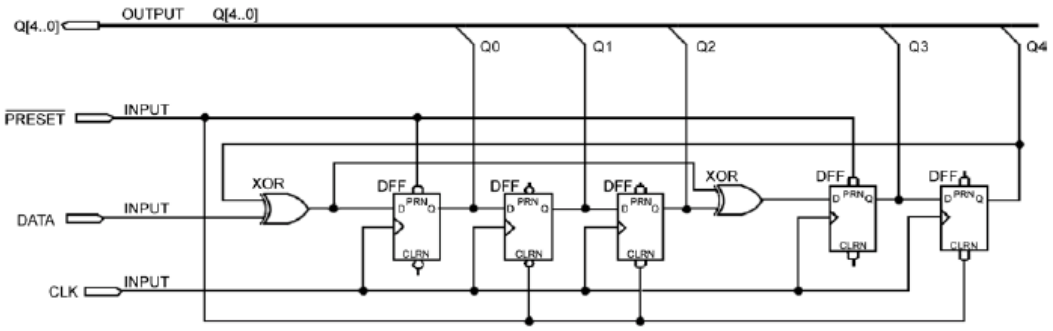

Figure F.1: Example CRC-5 circuit

Figure 4.3: Network example for computation and verification of CRC-5[3]

4.2 Physical Layer

Tags receive power while communicating with reader within the frequency range from 860 to 960 MHz. The reader operational frequency depends on radio regulations and on the local radio-frequency environment.
A reader communicates with one or more tags by modulating an RF carrier using DSB-ASK, SSB-ASK or PR-ASK. Tags respond to an interrogator by backscattering the modulating amplitude and or phase of the RF carrier. The data rate is set through the preamble parameter RTcal and it depends on data encoding.

4.2.1 Interrogator to tag communications - Data encoding

Pulse interval encoding (PIE) describes how binary symbols are represented into the interrogator to tag communications. PIE symbols are formed by TARI and PW values. TARI is the time length of Data-0 symbol, while PW is the time length of low logic level of both symbols. Data-0 and Data-1 symbols are shown in figure 4.4.

Preamble or frame-synch are concatenated at each command transmitted by reader. A preamble is comprised by a start delimiter, a Data-0 symbol, an RTcal symbol and a TRcal symbol as well as frame-sync but it doesn't include the TRcal symbol.
RTcal and TRcal are measured by tags. RTcal is used by tags to compute $pivot = RTcal/2$. Every subsequences shorter than pivot are interpreted as Data-0 whereas subsequences longer than pivot are translated as Data-1. Trcal, with divide ratio (DR), is used to compute the backscatter link frequency (BLF).

$$BLF = \frac{DR}{TRcal} \tag{4.1}$$

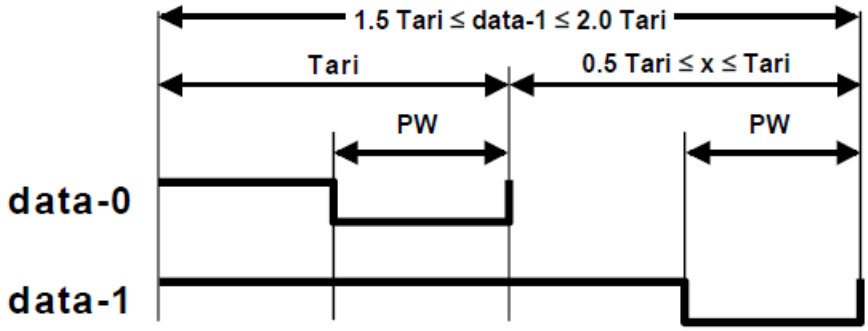

Figure 4.4: PIE symbols[3]

DR is defined by reader as parameter of query command.
Preamble and framesync are represented in figure 4.5

4.2.2 Tag to interrogator communications - Data encoding

Tags use one of 4 data encoding types: FM0, Miller 2, Miller 4, or Miller 8. The data encoding type is specified in the query command by the reader.
FM0 data encoding defines Data-0 through a middle-symbol phase inversion and provides a phase inversion at every symbol boundary.
Miller data encoding provides phase inversion between two Data-0s in sequence and phase inversion in the middle of Data-1 symbol. The transmitted waveform is the baseband waveform multiplied by a square wave at M times the symbol rate for M = 2,4,8. M value is specified in the Query command by the reader. Miller encoding implies more transitions per bit compared to FM0 encoding. The number of transition per bit increases with M but data rate consequently decreases. Therefore miller encoding works better in presence of noise, however data rate is lower than FM0 encoding. Both FM0 and Miller data encoding use preamble and dummy symbol to synchronize the data transmission. In both cases preamble can be normal or extended and is decided by the reader through TRext field in the query command (see figure 4.6 and 4.7). Dummy symbol is introduced at the end of the sequences and represents the stop symbol. In figure 4.8 and 4.9 are shown data and dummy symbol for respectively FM0 and Miller encoding.

4.3 Logical interface

In this section are presented the main operations performed by tag and reader. An example of communication flow is depicted in figure 4.10.

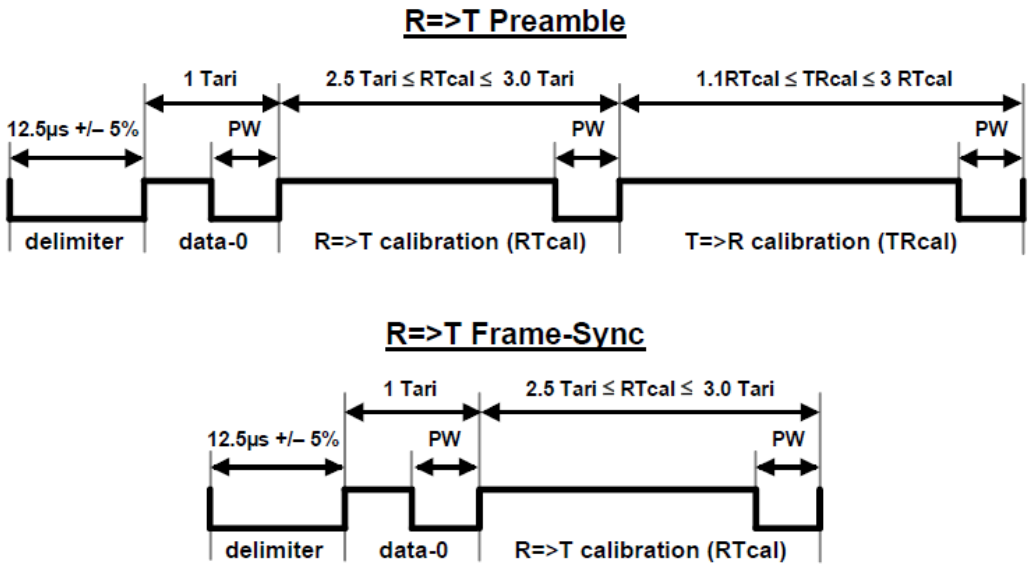

Figure 4.5: Preamble and framesync of reader to tag communication[3]

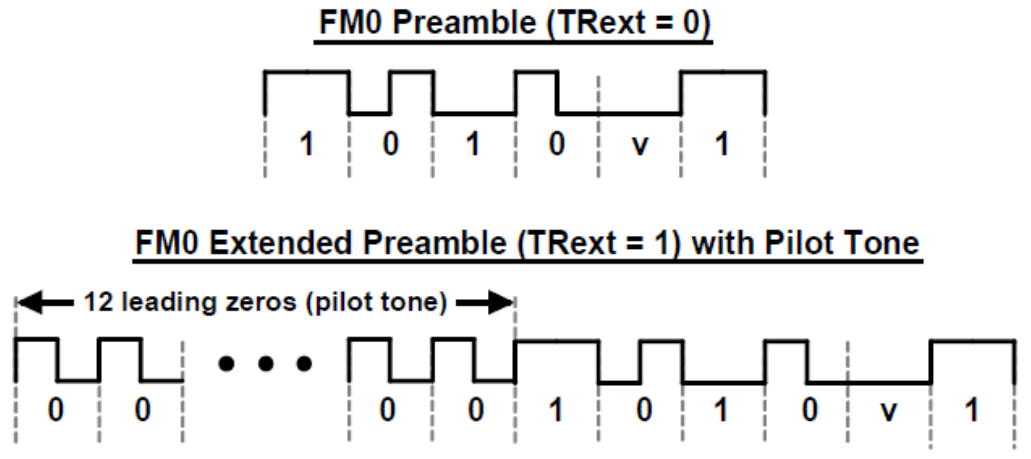

Figure 4.6: FM0 preamble sequences[3]

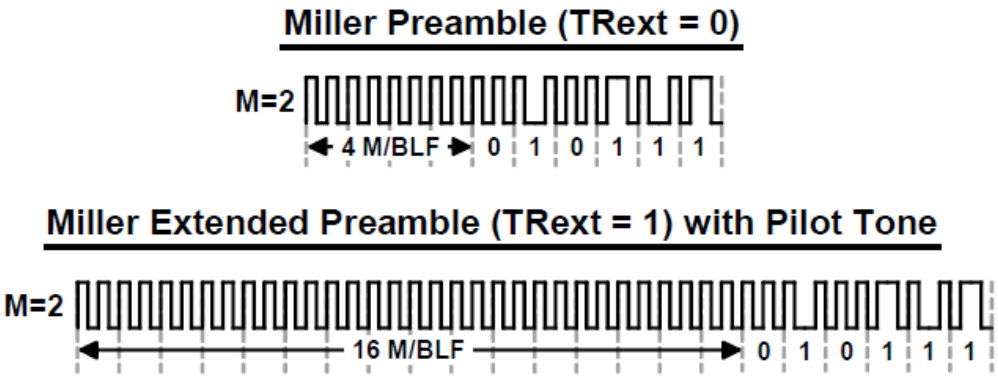

Figure 4.7: Miller preamble sequences with M=2[3]

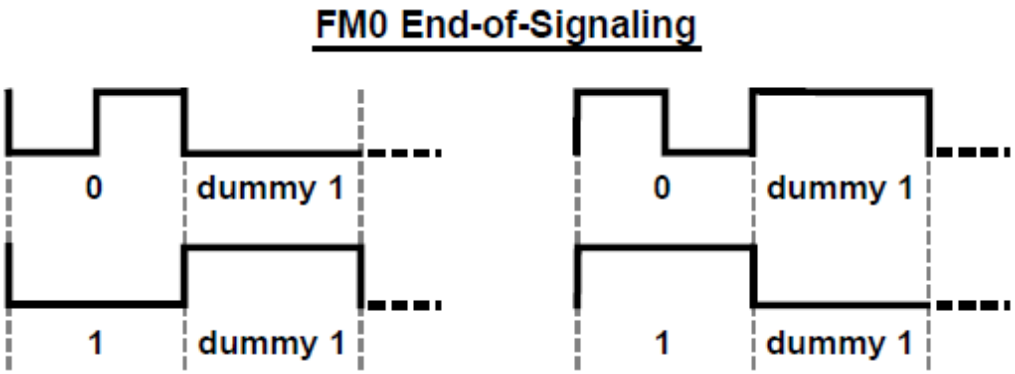

Figure 4.8: Data and dummy symbols with FM0 encoding[3]

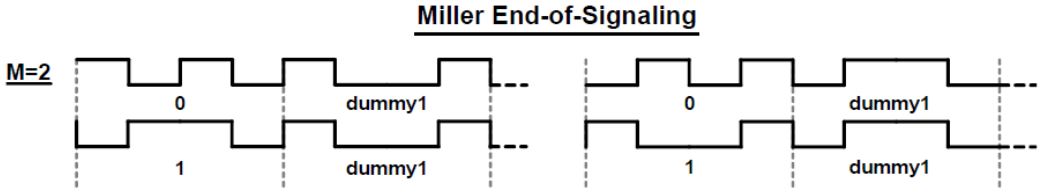

Figure 4.9: Data and dummy symbols with M=2 miller encoding[3]

4.3.1 Basic operations

An Interrogator manages Tag populations using three basic operations:

- **Select**: Choosing a Tag population. This command allow to include or exclude a subset of tag(s). An Interrogator may subsequently inventory and access the chosen tag(s).
- **Inventory**: Identifying individual Tags. An Interrogator begins an inventory round by transmitting a Query command. One or more Tags may reply on the basis of parameters set with select command. The Interrogator detects a single Tag reply and requests the tag's Electronic Product Code (EPC).
- **Access**: Communicating with an identified Tag. The Interrogator may perform a core operation such as reading, writing, locking, or killing the tag; a security-related operation such as authenticating the tag; or a file-related operation such as opening a particular file in the Tag's User memory. Access comprises multiple commands. An Interrogator may only access a uniquely identified Tag.

4.3.2 Tag states

Tag response depends on command transmitted by interrogator and on tag state. Some Tag states are listed herein:

- **Ready**: It can be viewed as a "holding-state" for energized tags that aren't participating in an inventory round. They remain until the reception of a query command.
- **Arbitrate**: It can be viewed as a "holding-state" for tags that are participating in the current inventory round whose counter holds nonzero value. Tag decrements its slot counter at each queryrep command, when it reaches the value 0000h tag transitions to reply state and backscatters an RN16
- **Reply**: In this state the tag is waiting for an ACK command. If tag receives a correct ACK command then it transits to acknowledged state backscattering a reply carrying the EPC. If this doesn't happen, the tag returns to arbitrate state. An ACK command is correct if it is received within time $T_{2,max}$(see figure 4.1) and with the same RN16 backscattered in the arbitrate state
- **Acknowledged**: It's a transition state. If a tag, whose access password is zero, received a Req_RN command then it backscatters a new random number

denoted handle and transitions to secured state. If a tag receives a correct ACK command then it reply as in the previous state. If a tag fails to receive a valid command within time $T_{2,max}$ then it returns to arbitrate state.

- **Secured** In this state tag execute access command. If a tag receives a correct ACK command (containing the correct handle) then it response as in the reply state.

4.3.3 Reader commands

Some reader commands are:

- **Query** starts an inventory round. It defines:
  - *Divide Ratio (DR)* is used to compute the BLF (see 4.1)
  - *M* sets the type of tag data encoding
  - *TRext* enables the transmission of extended preamble
  - *Sel, Target* choose which tags respond to query command and participate in the inventory round
  - *Session* chooses a session for the inventory round. Sessions determines when a tag will respond to a query from the reader and allows tags to maintain independent states when communicating with multiple readers at the same time
  - *Q* sets the parameter that initializes tag counter
- **QueryAdjust** adjusts the parameter Q.
- **QueryRep** decrements Tags slot counter
- **ACK** used to acknowledge a single tag. If a tag is in the reply or acknowledgment state then ACK echoes the RN16. Otherwise if a tag is in the secured state then ACK echoes the handle.
- **NAK** is transmitted when somethings went wrong. It forces a tag to return to arbitrate state unless the tag is in ready state.
- **Req_RN** instructs a tag to backscatter a new RN16. When issuing a Req_RN to a Tag in the acknowledged state an interrogator shall include the tag's last backscattered RN16 as a parameter in the Req_RN. If a Tag receives a Req_RN with a correct RN16 and a correct CRC-16 then it generate, store and backscatter a handle, and transition to the secured state. Otherwise if a

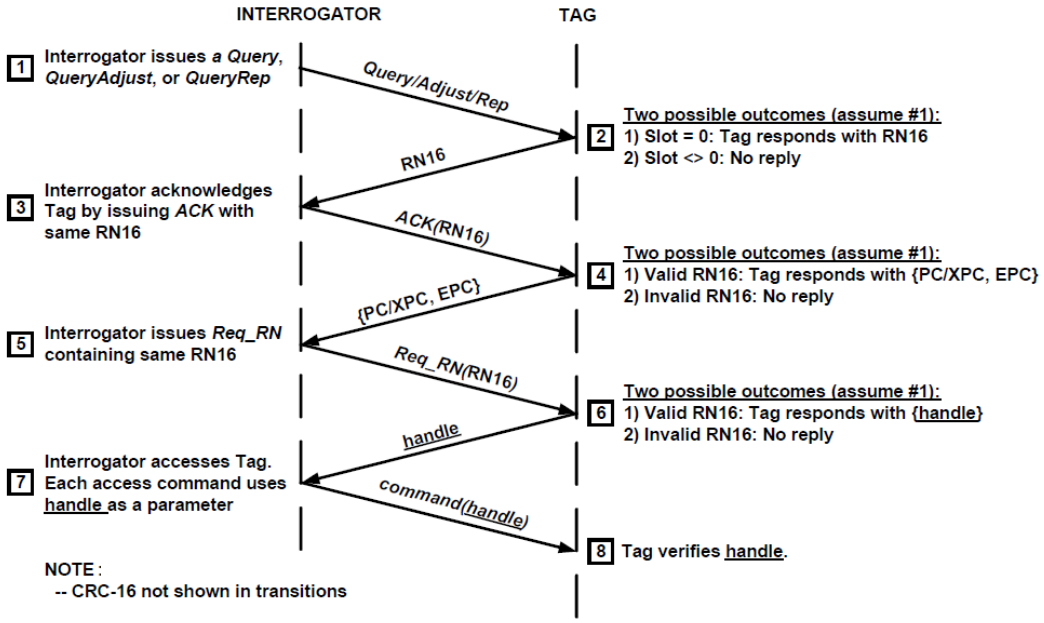

Figure 4.10: Example of inventory and access operations[3]

tag is in the secured state and receives the Req_RN with a correct handle and a correct CRC-16 then it generate and backscatter a new RN16, remaining in its current state

- **Read** allows an interrogator to read one or more words in tag's memory. Read command specify bank, address and number of words to be read in tag's memory. Tag's reply comprise a 0-bit of header, memory words, handle and CRC-16.

- **Write** allows an interrogator to write a word in tag's memory. A tag execute write command only in the secured state. If a tag receives a write command and the immediately preceding command wasn't a Req_RN then it treat the command as invalid. If the command was successfully executed then tag reply with a 0-bit of header, handle and CRC-16.

4.3.4 Tag memory organization

Tag memory is divided into four distinct memory banks as depicted in figure 4.11:

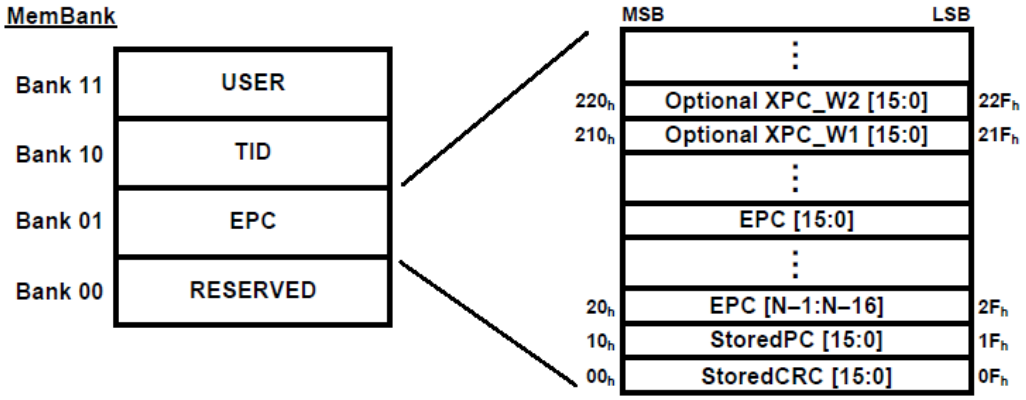

Figure 4.11: Tag memory and EPC bank organization[3]

- **Reserved memory** contains the kill and access passwords, if passwords are implemented on the tag
- **EPC memory** contain a storedCRC, a storedPC, the EPC and if tag implements Extended protocol control then either one or two XPC word(s).
- **TID memory** contains the manufacturer class identifier, sufficient information for an Interrogator to uniquely identify the custom commands and/or optional features that a Tag supports.
- **User memory** is used to storage user data.

EPC memory contains a StoredCRC at addresses 00h to 0Fh, a StoredPC at 10h to 1Fh, an EPC beginning at 20h. The electronic product code (EPC) identifies the object to which the tag is affixed, storedPC is a protocol control word and StoredCRC is a CRC code stored every times a tag power up or when an interrogator writes in the EPC and storedPC memory allocation.

Chapter 5

Tag architecture

The hardware platform on which the digital logic was built is a battery-less UHF-RFID sensor tag for environmental monitoring [1]. This sensor is an improvement of the last version of the open source WISP project [2].
In this project, the hardware architecture of the platform is modified by adding a UWB system composed by an antenna and a switch, an external memory, replacing the MCU with a custom digital logic and the TMP112 temperature sensor with HTS221 temperature and humidity sensor as depicted in figure 5.1. The node is based on a single UHF monopole used for both RF energy harvesting and Wake-up radio capabilities. The RF path is split in two different parts through a capacitive divider. The rectified voltage on the main power conversion output port is boosted by the onboard DC/DC converter which performs a maximum power point tracking (MPPT) of the RF source and supplies the load circuitry when sufficient energy is available on the storage capacitor C_s. A voltage monitor embedded in the DC/DC converter allows to perform the load power gating (SW1) between high and low threshold voltages according to load requirements. On the RF side, a very limited part of the incoming power is rectified to provide wake-up signals to a digital logic that implement the EPC Gen 2 protocol. Data are stored in a low power consumption Ferroelectric RAM (FRAM) which communicates with the digital logic through the SPI protocol.
Tag communicates and spreads its position by backscattering the UHF and UWB signals. The backscattering process is controlled by transistors connected to the antennas. Switching transistors change antenna load impedance, in this way backscattered signals can be modulated controlling the gate signals of transistors. Tag localization could be in principle performed through the UHF backscattered signals, however UWB signals ensure greater accuracy.
The memory choice is based on voltage supply, power consumption, type of

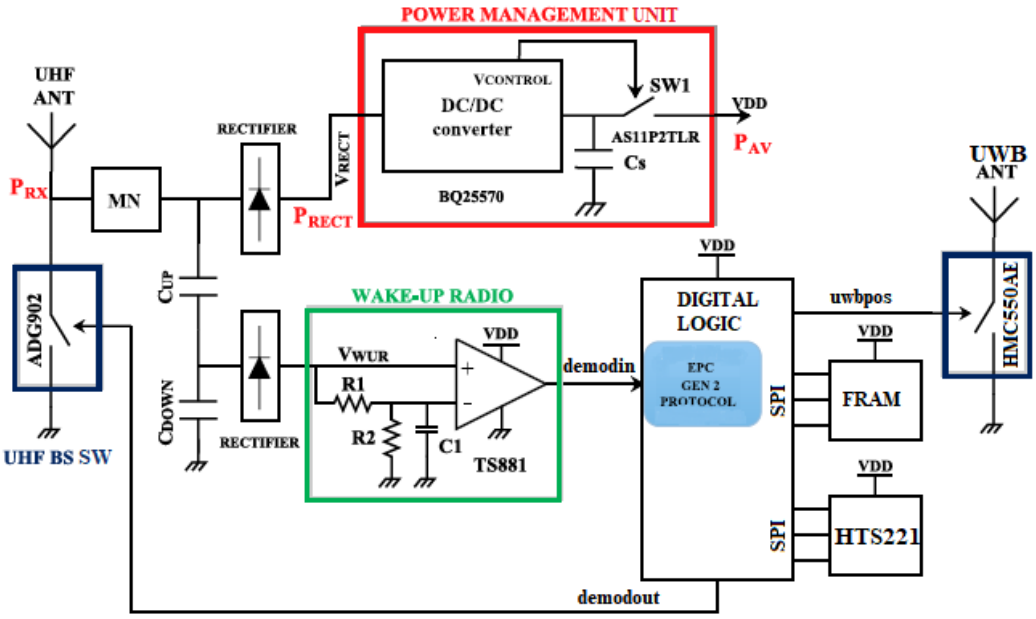

Figure 5.1: Hardware platform architecture

communication, read and write times. Since a non volatile memory is required and an high storage capacity is not necessary, FRAM turns out to be the best choice. FRAM ensure faster write operation compared to EEPROM and flash memory, providing low power consumption, low voltage supply and fast read operation. Fast read and write operation are required to satisfy reply timing defined by the protocol (see 4.1). Low power consumption is required since tags are battery-less, therefore little power is generally available. Voltage supply is kept at 2.5V by the DC/DC converter however, due to power fluctuations, voltage supply could fall below 2V therefore low voltage supply is still significant to ensure tag operation. Serial communication was chosen to reduce the number of interconnections between the memory and the digital logic. SPI protocol was chosen because it is faster, does not require pull-up resistor and is less power consuming compared to I^2C protocol.

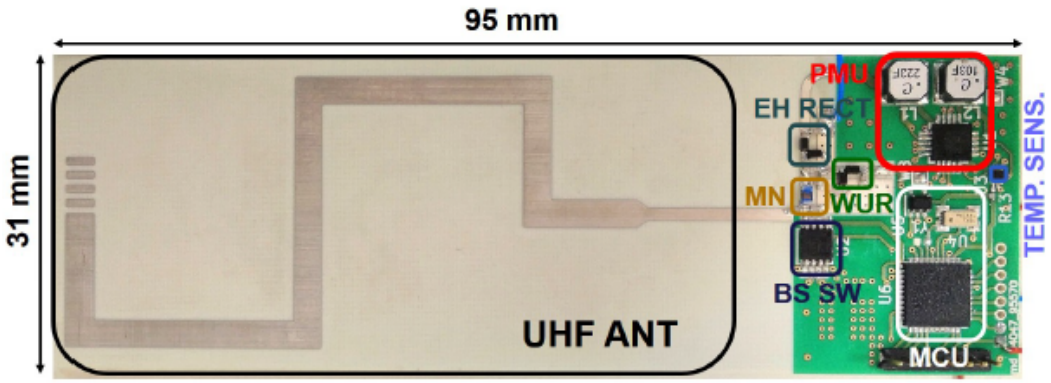

Figure 5.2: PCB Board of UHF-RFID tag. The MCU and temperature sensor have been disabled to connect the digital logic[1]

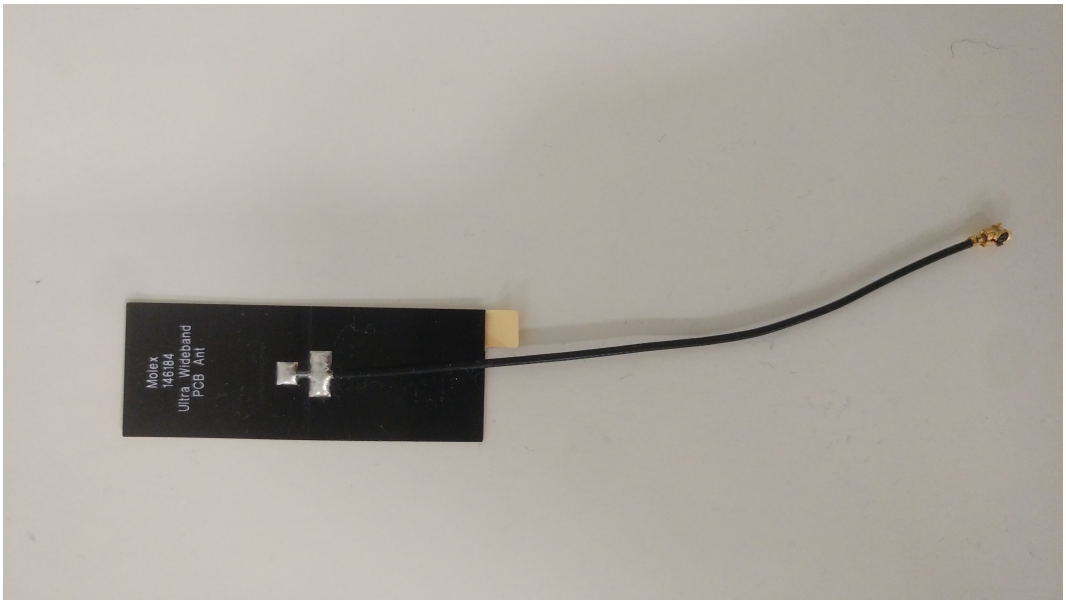

Figure 5.3: Example of UWB antenna integrable with tag architecture

Chapter 6

Digital logic implementation

This chapter will present two implementations of the digital logic which implement the EPC gen2 protocol. The first makes use of a Verilog synthesized system memory, the second uses an external FRAM as system memory with localization and sensing additional features. The first implementation is easier, however verilog memory is volatile and inefficient because it's synthesized with a RTL approach.
Digital logic has been implemented starting from the open source verilog code of WISP project[4],[5],[6].

6.1 Open source code explanation

The open source code available from the WISP project, used as base for the project, is able to manage: Query, QueryAdjust, QueryRep, ACK, NAK, Req_RN protocol commands. It doesn't provide the use of a system memory, therefore the write command has not been deployed and read command was managed in a different way to that defined by the protocol. In particular, it uses read command to acquire and transmit data collected by an ADC. Moreover, EPC identification code isn't stored in a non-volatile memory but is rather fixed with a Verilog code line as a constant, therefore the EPC can't be modified by readers.
It was developed in a modular way. It measures RTcal and TRcal parameters, decodes data received, recognizes the command type, verifies the matching of RN16 or handle, decodes command parameters, computes the BLF and CRC code, sets the data encoding type (FM0, Miller) and provides the correct reply sequence to the passive transmitter.
The main Verilog blocks that compose the digital logic are (see 6.1):

- **top**: The top level entity that connect all the functional blocks
- **rx**: decodes PIE symbols received by readers and measures how long is TRcal symbol
- **cmdparser**: decodes the command bit sequence and provide, to txsettings block, M, DR, TRext values transmitted inside the query command
- **txsettings**: stores M, DR, TRext values
- **packetparser**: decodes the other command's field and verify if RN16 or handle included in the command received match with what tag expects.
- **controller**: decides if and what to reply after a command reception. It also manages the update of Q parameter, initialize and update the slot counter and enables tag's replay when the slot counter reaches 0 as value.
- **txclkdivide**: sets the BLF through the computation of 4.1
- **sequencer**: manages the data flow that will be provided to the transmission block (tx). Data flow includes preamble, information, CRC16 and dummy 1 symbol.
- **preamble**: checks the TRext value and generates the preamble or extended preamble sequence for FM0 or miller encoding.
- **crc16**: takes the sequence to be transmitted and compute the CRC-16.
- **tx**: converts the serial data stream, provided by the sequencer block, into a miller or FM0 encoding.
- **counter10**: a simple counter used by rx block to perform time measurements.
- **rng**: generates random number used as RN16 or handle. It takes the LSB of counter10 block and generates random numbers using the CRC algorithm.
- **epc**: used to generate response to an ACK command. It's not a memory; it assigns (through verilog line code) the storedPC and EPC values.

6.2 Digital logic with Verilog synthesized memory

It's an improvement of the open source code. With this update the digital logic is able to manage also Read and Write commands as described by the protocol.

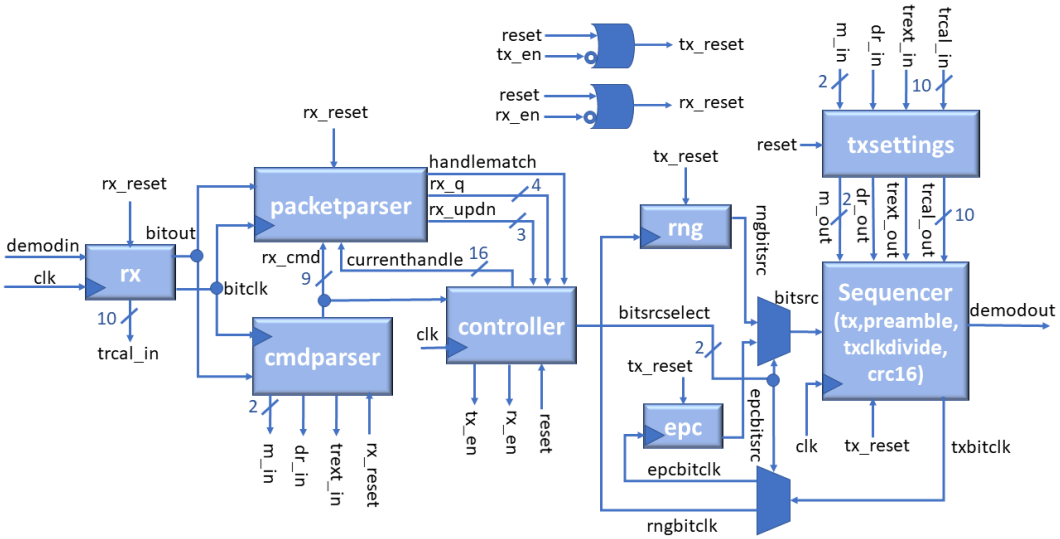

Figure 6.1: Simplified representation of main blocks and signals that compose the open source digital logic

This new implementation also allows to change the EPC value by the reader since it's stored in the system memory as suggested by the protocol.

This improvement adds the following Verilog blocks:

- **memory**: a simple 128x16 synchronous parallel memory. It can be written when write enable signal is high, otherwise it can be read. Whenever it received the reset signal, it preloads the storedPC and EPC values.
- **writemem**: converts the serial data stream into a parallel one. It also manages the memory addressing by adding an offset that select the correct memory area. The offset value depend on the memory bank chooses by the reader.
- **readmem**: provides the address (added with the offset) to the memory and converts the parallel data received into a serial one. It also builds the replays for read or ACK commands and provide the serial data stream to the sequencer block.

The following open source blocks are modified:

- **epc**: has been removed because storedPC and EPC are stored in memory block and they are provided, as response of ACK command, by readmem block.

- **packeparser**: introduced the write enable signal for memory and the selection management of RN16 or handle as verification sequence.
- **controller**: introduced the storage of address and number of words received during a read command. This feature was introduced because the packetparse block is reset during a transmission operation while controller block is reset only through system reset. Added some control flags used to control the other blocks:
  - *ack_flag*: select which kind of replay the readmem block should transmit.
  - *tagisopen*: select the handle or RN16 used by packetparse to verify tags identification
  - *selectRN*: select the handle or RN16 to be transmitted by rng block.
  - *delay_tag_replay*: flag that indicates if sequencer block should transmit a delay tag replay.

  Updated management of the following command:
  - *ACK*: Modified the data source of command reply and added the raising of ack_flag. EPC is now stored in the system memory therefore tag will response to ACK command taking data from memory.
  - *REQ_RN*: Introduced a control that allows tags to store a new handle only if tag isn't in the secured state and raises tagisopen flag.
  - *READ*: Introduced the selection of system memory as data source for command reply, the storage of address, memory bank and number of words to be read and the raising of selectRN flag.
  - *WRITE*: Introduced the selection of rng block as data source for command reply and the raising of selectRN and delay_tag_replay flags.
- **sequencer**: introduced a new state (STATE_HEADER) in the state machine that allows the transmission of an header bit when delay_tag_reply flag is high.
- **preamble**: introduced a control that allow the transmission of extended preamble when delay_tag_reply flag is high.
- **rng**: introduced the selection of RN16 or handle as output data flow through the selectRN flag.

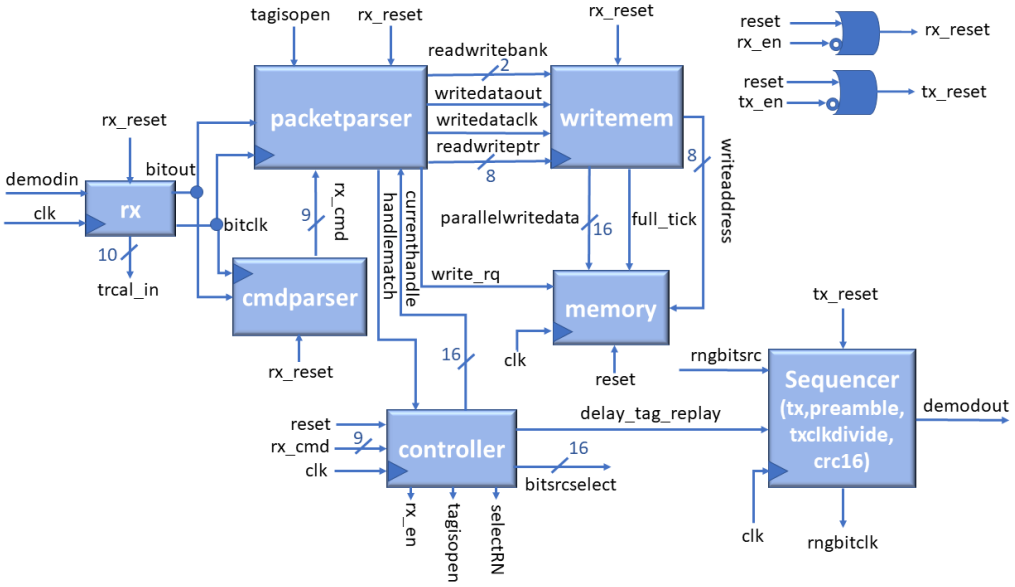

Figure 6.2: Simplified representation of main blocks and signals involved in write command with verilog synthesized memory

Write and read operations are set out in figures 6.2, 6.3.
When a tag receives a write command then the packetparse block communicates to the writemem block the address, the memory bank and the data to be written. The latter computes the correct memory address, parallelizes data received and writes them when parallelization is complete. At the same time the sequencer block replies to the command by transmitting a delay tag reply. The delay tag reply is built by chaining an header bit, the handle and the CRC-16.
When a tag receives a read command then the controller block communicates to the readmem block the address and the number of words to be read. The latter computes the correct memory address, reads and serializes words and builds the reply through the concatenation of an header bit, words read, the handle and the CRC-16.

6.3 Digital logic with external FRAM, UWB localization, temperature and humidity sensing

This version is an improvement of the digital logic with Verilog synthesized memory. With this update the system memory used by the digital logic is external and non-volatile. The non-volatile memory allows to keep stored the EPC and user data when power supply goes down.

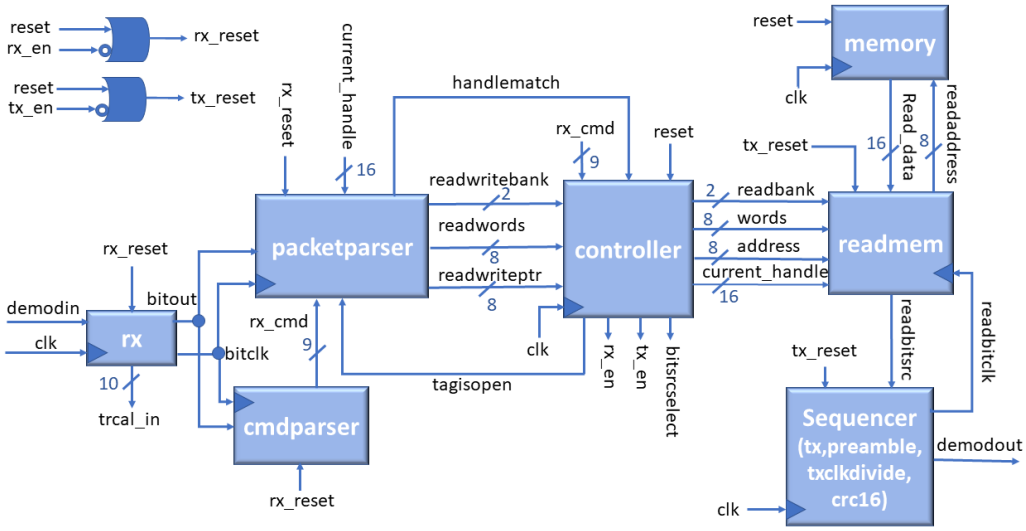

Figure 6.3: Simplified representation of main blocks and signals involved in read command with verilog synthesized memory

6.3.1 Sensing and UWB localization features:

The UWB localization feature has been introduced with a passive approach. The UWB localization is performed by backscattering an UWB signal transmitted by an external UWB source. The digital logic modulates the UWB backscattered signal through a clock signal given in output.
Moreover, temperature and humidity sensing have been introduced through HTS221. This sensor requires a conversion of output data by means of calibration parameters. Data conversion requires tricky operations for digital logic like division or multiplications for decimal numbers. Since Tag should be less power consuming as possible, data conversion is supposed to be executed by the reader. Calibration parameters are set by the manufacturer and are different for each devices. To perform data conversion the reader must read calibration parameters, temperature or humidity acquired and then compute the converted value as depicted in figure 6.4 and described in equation 6.1.

$$T_{DegC} = T_{OUT} \cdot m + q \tag{6.1}$$

Where m and q are obtained with straigh line passing through two points equation:

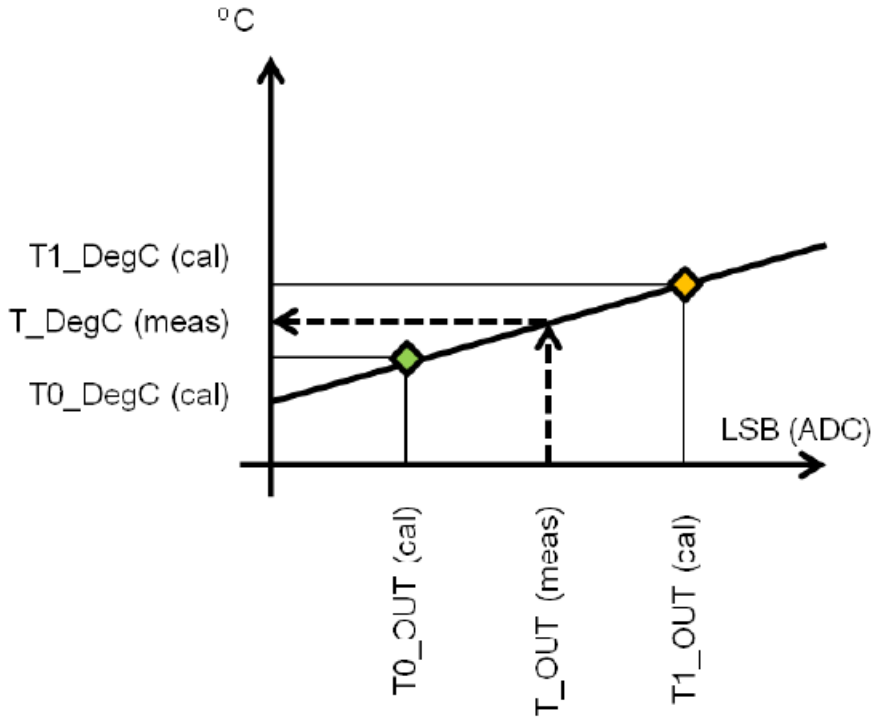

Figure 6.4: Graphic description of temperature data conversion[9]. T_{OUT} is the temperature acquired, T_{DegC} is the temperature converted and the others are calibration parameters.

$$m = \frac{T0_{DegC} - T1_{DegC}}{T0_{OUT} - T1_{OUT}} \tag{6.2}$$

$$q = \frac{T0_{OUT} \cdot T1_{DegC} - T1_{OUT} \cdot T0_{DegC}}{T0_{OUT} - T1_{OUT}} \tag{6.3}$$

Calibration parameters don't change in time so reader must acquire them only once. Obviously if the sensor is replaced they must be updated. Calibration parameters are stored in reserved memory bank from address 40_h to 130_h in the following order: H0_rH_x2, H1_rH_x2, T0_degC_x8, T1_degC_x8, T1/T0 msb, H0_T0_OUT, H1_T0_OUT, T0_OUT, T1_OUT. Humidity and temperature are read and stored in memory every time the reader transmits a query command and are respectively available at 00_h and 10_h address of user memory bank.

Every time the sensor is powered on, some configuration parameters must be set as described by the datasheet [9]. These parameters are set by writing its configuration registers in order to control its behaviour.

6.3.2 Digital logic explanation

Since the FRAM slots contain 8 bit whereas words to be written are 16 bit length, the address received by readers does not correspond to physical memory address. Physical memory address is computed through equation 6.4:

$$ADDR_{fram} = ADDR_{reader} \cdot 2 + \#BANK \cdot 64 \tag{6.4}$$

Where $ADDR_{reader}$ is doubling because every words need two memory slots, 64 is twice the number of words per bank which reader's software (Impinj Item Test) admits, $\#BANK$ is respectively 0,1,2,3 for Reserved, EPC, TID, User memory banks.
The features described above have been implemented with the following verilog blocks:

- **FRAMcontroller**: interfaces the digital logic with the external SPI FRAM. Through the spi_master block it manages the FRAM control signals and issues commands to perform read and write operations. Specifically it provides to spi_master block the data to be transmitted and starts or quits the SPI communication. Commands expected by FRAM always start with an operation code which identify the type of operation to be performed (write enable, write, read ...). A write enable operation shall always precede a write operation. The finite state machine implemented is described in figure 6.6. Cypress FM25L04B FRAM automatically disable write enable latch after a write operation. However write disable command is execute after a write operation, in case of others FRAM models don't implement this function. It also manages the filling of fifo buffer block during read operation.

- **spi_master**: drives chip select, serial clock, data in and data out signals. It also converts serial data received into a parallel one. It was developed through an open source code [10]. This basic code has been modified to transmit a serial stream of more than 8 bit until the reception of stop signal. Moreover the maximum serial clock frequency has been increase to clock/2 instead of clock/4.

- **fifo**: implements a first in firs out (fifo) buffer used to temporarily stores data read from memory. it stores up to 64 words of 8 bit. This block was implemented in the open source verilog code [4].

- **localization**: puts on output the system clock when readers transmit the

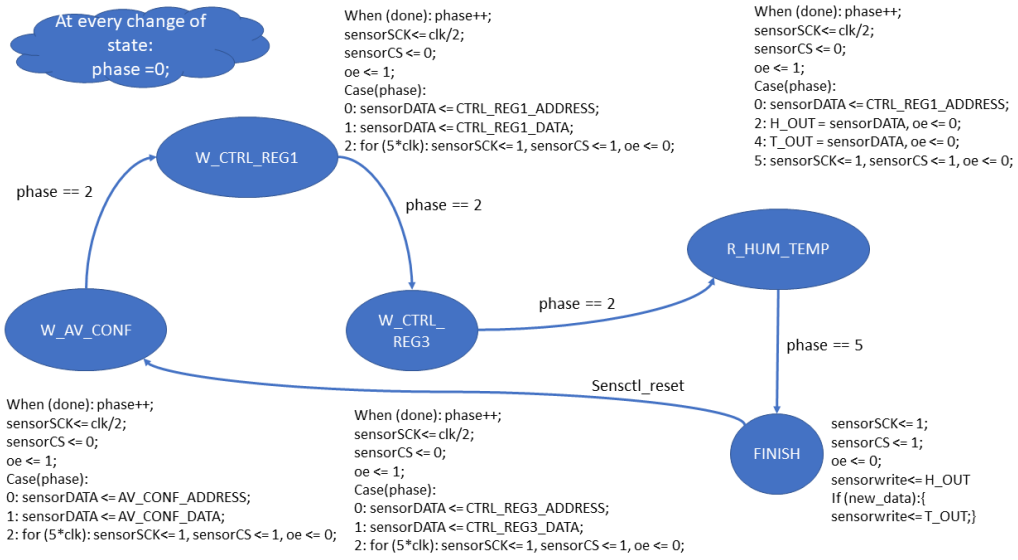

Figure 6.5: Logical explanation of sensor controller's finite state machine

ACK command. The length of localization window depends on system clock frequency; with a system clock of 10 MHz it is given in output for $2 \div 3ms$.

- **SENSORcontroller**: interfaces the digital logic with HTS221 sensor through SPI communication protocol. By the spi_master block it issues commands to write configuration registers and to perform temperature and humidity readings. Specifically it provides to spi_master block the data to be transmitted and starts or quits the SPI communication. Since tag is supplied by energy harvesting, sensor may turn off in an unpredictable way. For this reason configuration registers are written whenever the sensor is read, therefore every time a query command is performed. Command expected by sensor are composed by one address byte and one or more data bytes. The two MSB of address byte are respectively used to select read or write operation and to enable multiple readings or writings. During a multiple operation, the communication is performed until the raising of chip select pin and address in automatically increased by sensor's internal logic. The finite state machine implemented is described in figure 6.5.

The introduction of the external memory needs an update of the following main blocks:

- **memory**: has been removed because external FRAM has become the system

memory

- **controller**: introduced the read_rq flag which notify a read memory operation to FRAMcontroller block. It also computes the physical memory address as previously specified
- **writemem**: updated the physical address computation and removed the increase of address since it is execute by FRAM internal logic.
- **readmem**: introduced the shift management of fifo buffer and removed the increase of address since it is execute by FRAM internal logic.
- **packetparse**: added the announcement, during a Query command, of address, memory bank and number of words to be read exploited by SENSORcontroller block.
- **top**: added input and output signals to interface FRAM and sensor. Added two combinational logic blocks to control the reset of FRAMcontroller and SENSORcontroller blocks. The FRAMcontroller is reset whenever the tag does not transmit and SENSORcontroller does not communicate with it or reset is raised. SENSORcontroller is reset every time a query command isn't performed or reset is raised. Three multiplexer are added, one provides clock signal to FRAMcontroller block, the others select data source for the previous block.

A simplified representation of how FRAMcontroller and SENSORcontroller blocks are connected in the digital logic is showed in figure 6.7.

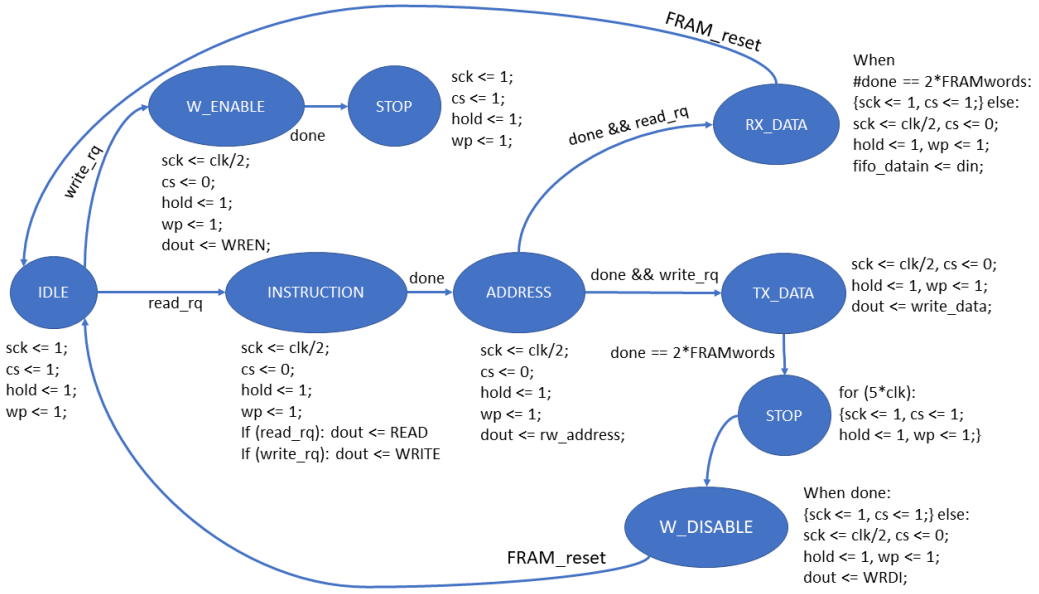

Figure 6.6: Logical explanation of FRAM controller's finite state machine

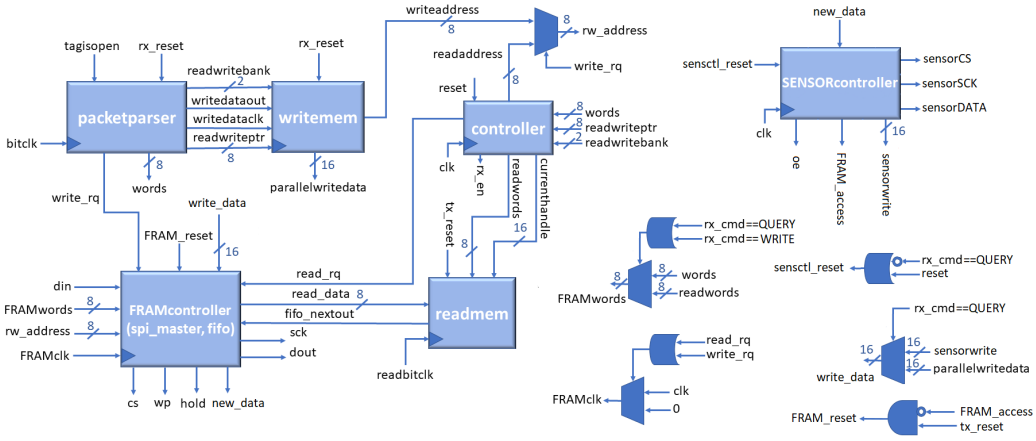

Figure 6.7: Simplified representation of main blocks and signals that interact with FRAMcontroller and SENSORcontroller blocks

Chapter 7

Simulation and test

This chapter will discuss the simulation and test approaches adopted and the results obtained. Simulations were performed through the implementation of Verilog testbenches while physical tests were executed on a FPGA, connected with the RFID tag described in figure 5.1, controlled by a commercial reader.
Simulations were performed with Altera Modelsim 10.1d to verify if tag's reply sequences are correct, if tag's response time meets the constraint defined by the protocol and if tag implements correctly the SPI communication protocol.
FPGA test verifies if the entire physical system is working. Specifically it checks if the tag interacts with the commercial reader, unrolls all the command imposed in a real environment and communicates correctly with FRAM memory, temperature and humidity sensor. Tests are conducted with an Altera Cyclone II FPGA programmed via Quartus II, a Speedway R420 reader by Impinj connected to a Far Field antenna (8.5dBi of gain) of the same company jointly with the Impinj Item Test software in indoor environment. To be ETSI[11] compliant, the maximum transmitted power of 2 W ERP has been used. The reader is initialized to perform Inventory rounds with Single Tag Inventory mode in Session 0, which represents the maximum reading throughput. However, the reader could change its configuration in run-time to ensure reliable communication with the tag.

7.1 Simulation with sampled reader signals

The purpose of this simulation is to verify if tag works correctly with a real sampled reader's data stream received by tag platform.
The data stream is obtained enabling reader transmission and acquiring, by means the oscilloscope Teledyne Lecroy WavePro 804HD, the output signal of the TS881 comparator placed on tag platform (see figure 5.1). A window of $3851\mu s$ has been

Figure 7.1: Simulation with reader signal acquired

acquired with a sampling frequency of $100MSample/s$, and the samples have been saved in a text file. The testbench reads this test file, extrapolates the samples, digitizes them by imposing a threshold and streams out the data flow to "demodin" input. The results of simulation are shown in figure 7.1.
The sequence sampled is a series of query commands since during the acquisition there weren't any responding tags. Simulation highlights that tag is able to interpret real reader's data and reply within the T_1 interval(see 4.1), in fact reply time results to be about $47\mu s$.

7.2 Simulation with generated reader signals

In this simulation the testbench generates a stream of PIE data which is put on "demodin" input of digital logic. The stream of data was generated in a static way, switching "demodin" input value after a defined time interval. This stream simulate the interaction of reader during a communication, the data stream generated (shown in 7.2) includes an inventory round and a write and read access operation.
Patterns of Data-0, Data-1 and preamble depend on Tari value as depicted in figures 4.4, 4.5. Ranges of parameters are reasumed in Table 7.3, Table 7.4 shows an example of parameter setting to generates the input data stream.

Tag backscatter link frequency (BLF) and intervals T_1, T_2, T_5 are computed as defined by the protocol[3]:

$$BLF = \frac{DR}{TRcal} \tag{7.1}$$

$$T_1^{max} = MAX(RTcal, 10T_{pri}) \cdot (1 - \mid FrT \mid) + 2\mu s \tag{7.2}$$

$$T_1^{min} = MAX(RTcal, 10T_{pri}) \cdot (1 - \mid FrT \mid) - 2\mu s \tag{7.3}$$

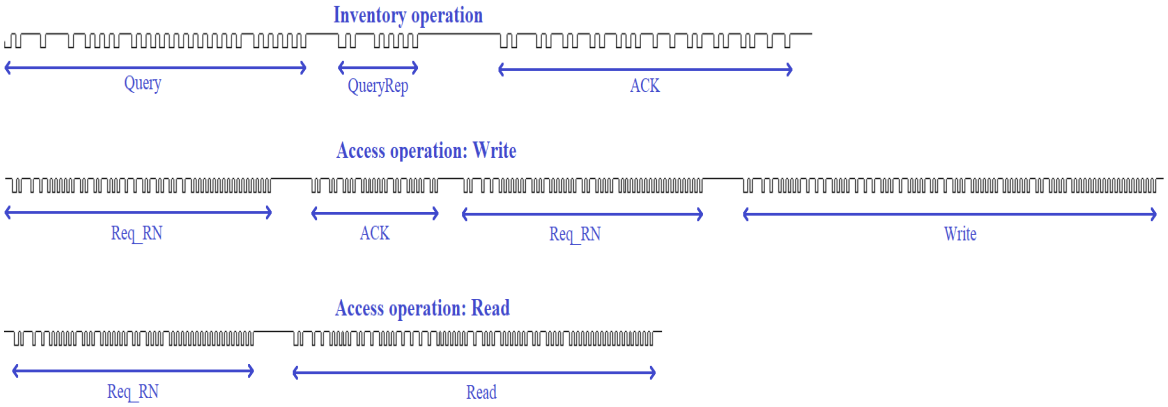

Figure 7.2: Generated signals by Testbench. To simplify the representation, different zoom levels have been used for inventory and access waveforms. The input data stream is obtained concatenating the three waveforms depicted.

	Min[µs]	Max[µs]
Tari	6.25us	25us
PW	MAX(0.265Tari, 2)	0.525Tari
Data-0	Tari	Tari
Data-1	1.5Tari	2Tari
RTcal	2.5Tari	3Tari
TRcal	2.75Tari	9Tari
Delimiter	11.875	13.125

Figure 7.3: Variability ranges of PIE data parameters

Name	Value	Explanation
Delimiter	12.5 μs	Start of frame
Tari	16.67 μs	Modulation depth
PW	8.335 μs	"0" duration
RT_{CAL}	41.7 μs - 50 μs	Pulsewidth
TR_{CAL}	45.8μs - 150μs	"0" + "1" duration
BLF	436 $Kbps$	Backscat. link freq.
FLF	128 $Kbps$	Forward link freq.
DR	64/3	Divide Ratio
Rev. Mod.	FM0	Reverse signal coding
T1 RT_{CAL}: 41.6μs RT_{CAL}: 50μs	 30.5μs - 52.8μs 37 μs - 63 μs	Immediate reply time from Interrogator transmission to Tag
T2	7.5 μs - 50 μs	Time from immediate Tag reply to Interrogator transmission
T5 RT_{CAL}: 41.6μs RT_{CAL}: 50μs	 30.5μs - 20ms 37 μs - 20ms	Delayed reply time from Interrogator transmission to Tag

Figure 7.4: PIE data parameters values set

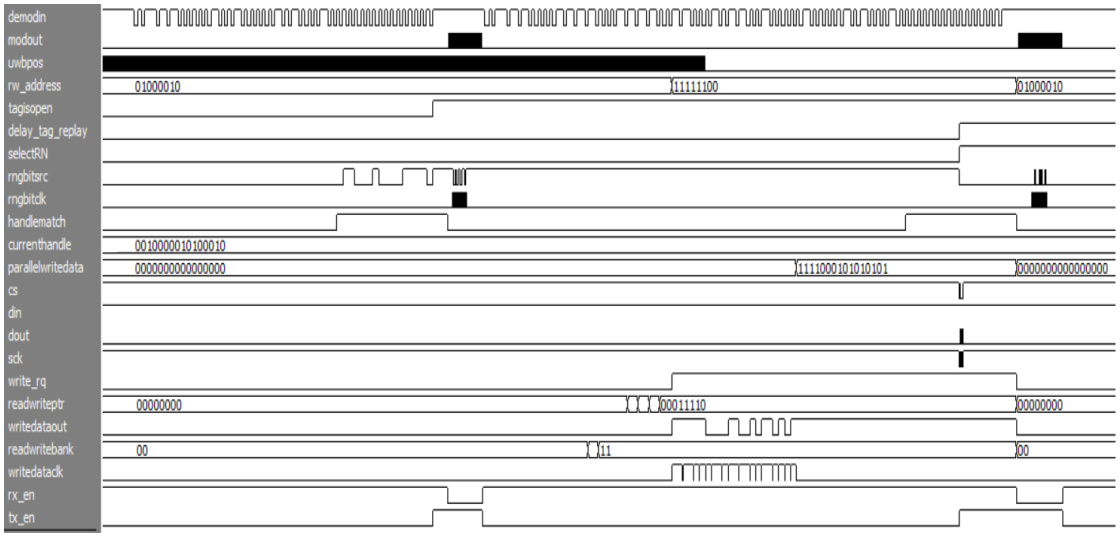

Figure 7.5: Waveforms of main signals involved during a write command simulation

$$T_2^{max} = 20Tpri \tag{7.4}$$

$$T_2^{min} = 3Tpri \tag{7.5}$$

$$T_5^{max} = MAX(RTcal, 10T_{pri}) \cdot (1- \mid FrT \mid) - 2\mu s \tag{7.6}$$

$$T_5^{min} = 20ms \tag{7.7}$$

where FrT is the frequency tolerance that depends on DR and BLF (see [3]) equal to 22% in this case study and T_{pri}:

$$T_{pri} = \frac{1}{BLF} = \frac{TRcal}{DR} \tag{7.8}$$

The presence of FRAM memory has been simulated by generating (through the testbench) a data stream on input "din". This stream simulate the data provided by FRAM memory during a read operation. In this way the simulation shows if tag interprets correctly data that memory will provide it. Similarly the presence of temperature and humidity sensor has been simulated by generating a data stream on "sensorDATA" interface during a read data operation. Since sensorDATA is an inout pin, a tristate buffer is added in the testbench to control the data flow. Figures 6.7 and 6.2 help understanding the results of a write command simulation shown in figure 7.5.

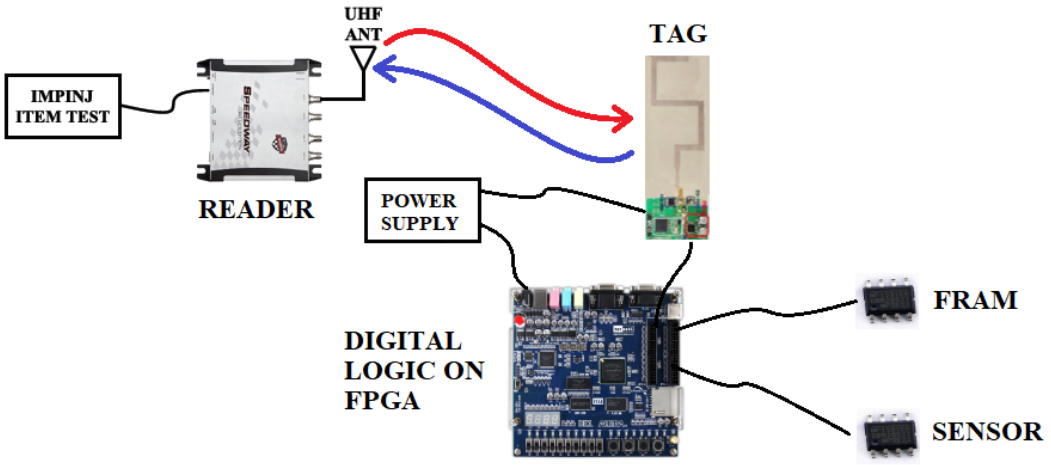

Figure 7.6: Laboratory setup during hardware test

7.3 Test on FPGA

The purpose of this test is to verify if the entire physical system is working correctly. The system tested is a prototype externally supplied in which the digital logic is synthesized on a FPGA. It involves Cypress FM25L04B as external FRAM memory, an STMicroelectronics HTS221 temperature and humidity sensor and doesn't include UWB antenna and switch. In future versions, the system will be evolved by integrating the digital logic as a chip mounted on tag platform, by using an FRAM with lower minimum voltage supply (e.g. like Lapis MR45V064B), with all blocks entirely supplied by reader's UHF signals through an on-chip RF power harvesting circuit. The test setup is depicted in figure 7.6.

A feasibility assessment has been performed to add the external FRAM. Since EPC gen2 protocol defines tag response time (see figure 4.1), the latency introduced by FRAM serial communication should ensure that all replies meet link timing. ACK and READ commands are more sensitive to link timing since their replies are obtained by reading the external memory.
The number of bit to transmit within link timing is:

$$\#bit = opcode + address + word = 8 + 8 + 16 = 32 \tag{7.9}$$

In the worst case the system clock is 3MHz:

$$SYS_{clk} = 3MHz \tag{7.10}$$

$$SPI_{clk} = \frac{SYS_{clk}}{2} = 1.5MHz \tag{7.11}$$

$$T_{SPI} = \frac{1}{SPI_{clock}} = 0.66\mu s \tag{7.12}$$

Time required to ensure tag reply:

$$t_r = \#bit \cdot T_{SPI} = 32 \cdot 0.66 \cdot 10^{-6} = 21.12\mu s \tag{7.13}$$

t_r turns out to be much shorter than link timing, therefore the uses of external serial memory is feasible.
The number of words to be read doesn't affect the response time. In fact FRAM communication is faster compared to tag reply data rate and replies are made by getting data from a fifo buffer which is in parallel filled up by FRAM read data.

The Impinj Item Test software was used to test the inventory process, the EPC change, read and write commands. The test conditions adopted:

- Tag voltage supply: 1.9V
- Reader transmitted power: 1W ERP
- Reader in max throughput mode and session 0
- Frequency of system clock: 10MHz
- indoor environment with line of sight propagation

Some example of data exchange captured by oscilloscope are showed in figures 7.8, 7.9, 7.10, 7.11, 7.12, 7.13.
The number of Inventory, read and write rates per minutes are difficult to evaluate since the system is only a prototype. The long flying wires used to connect the FPGA to tag, FRAM memory and sensor, behave as antennas which interfere with UHF signals transmitted by the reader, degrading performance. The interferences between "demodin" and "demodout" lines are highlighted in figure 7.12. Moreover, since the digital logic is implemented on FPGA which is externally supplies, even the tag is externally supplied, therefore the tag's working distance isn't valuable. However indicative measures have been performed to prove that the tag works correctly, the results are showed in figure 7.7.

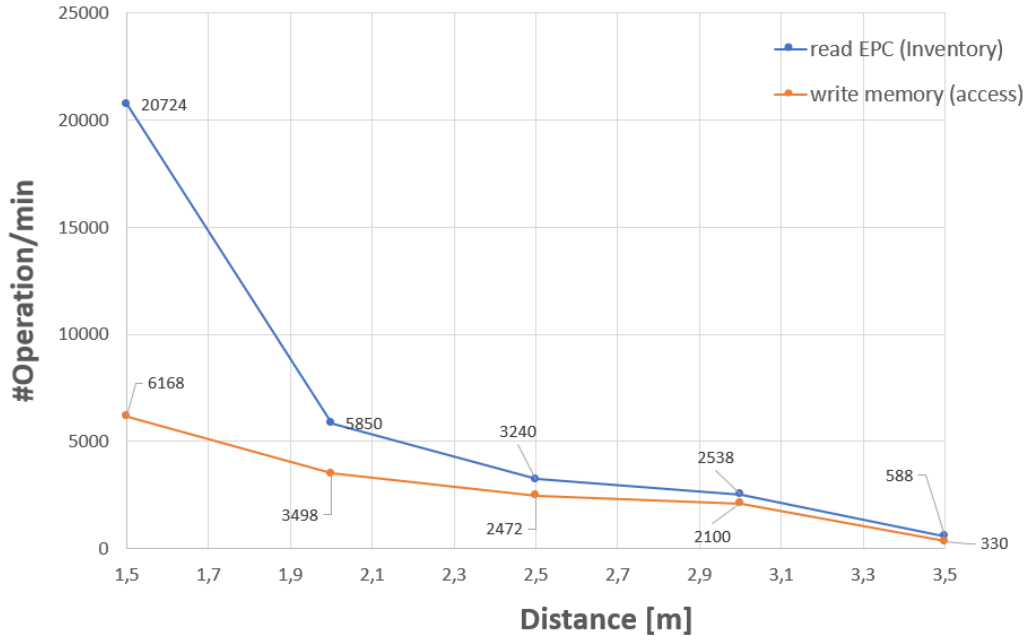

Figure 7.7: Indicative number of operation per minute measured under the test conditions previously defined

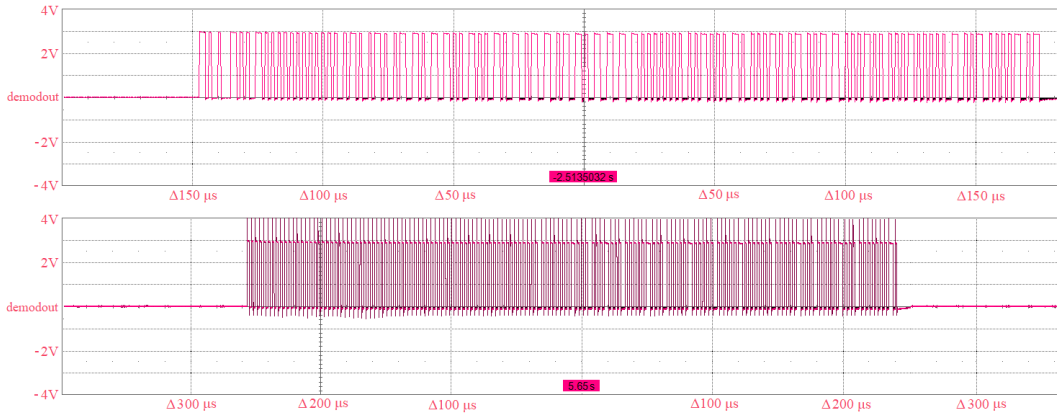

Figure 7.8: EPC transmitted by the tag with FM0 (top figure) and Miller (bottom figure) encoding

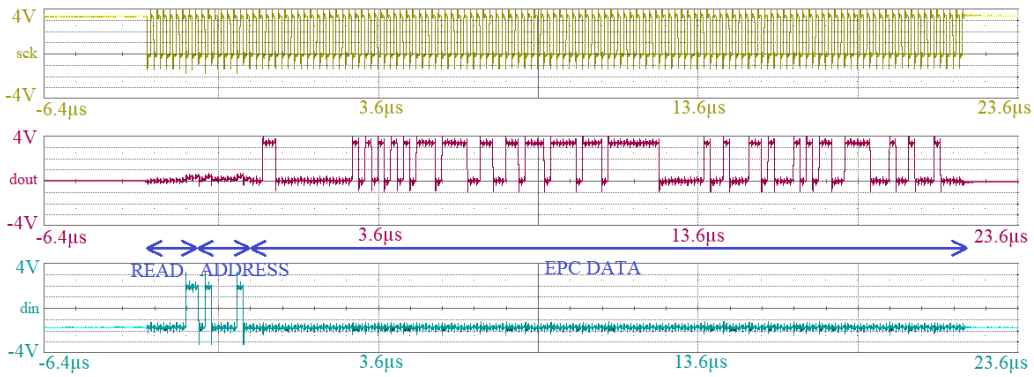

Figure 7.9: SPI signals during a read of external FM25L04B FRAM

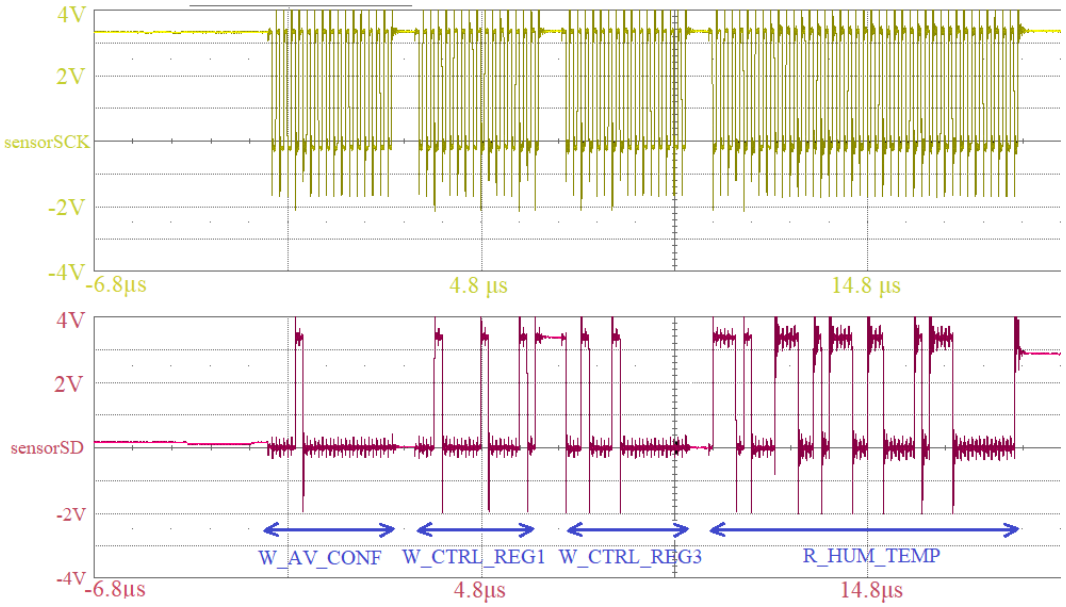

Figure 7.10: SPI signals during humidity and temperature acquisition from HTS221 sensor

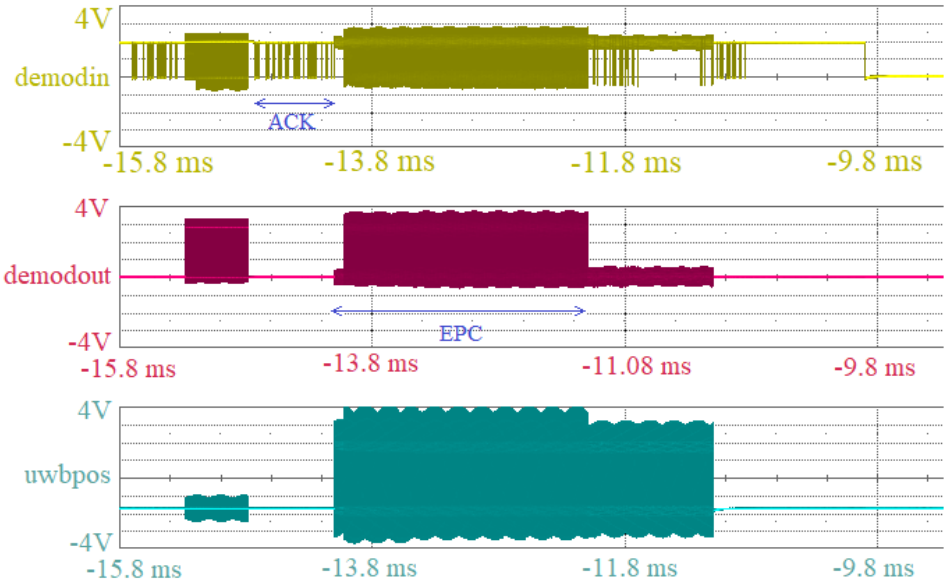

Figure 7.11: uwbpos localization signal given on output during inventory operation

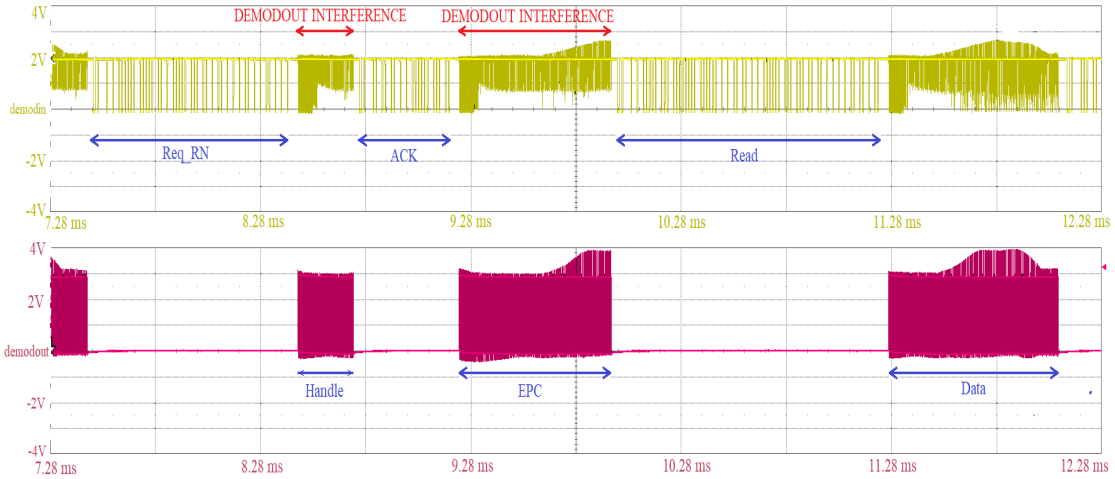

Figure 7.12: Data exchanged during a read command. "Demodout" interference on "demodin" are pointed out

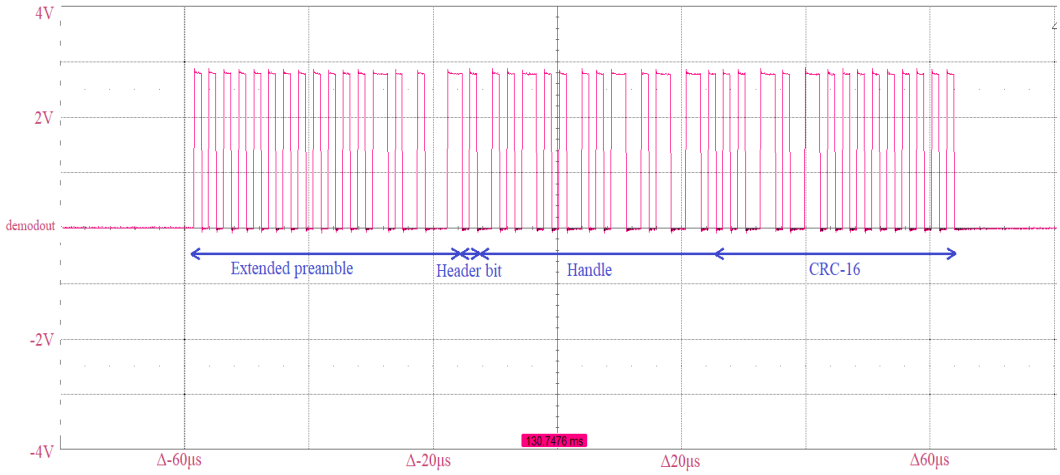

Figure 7.13: Tag reply of a write command

Chapter 8

Chip design

This chapter will present some steps of the microelectronic design based of the implemented digital logic on a 0.18 μm CMOS process. Cadence tools used for the design are Genus for synthesis process and Innovus for placement, clock tree synthesis and routing. Timing checks were performed after placement, clock tree synthesis and routing to verify that time violating paths didn't exist. This design does not aim to be a final implementation but rather an explorative study, then a more accurate setup should be performed to obtained the necessary performance for a final silicon implementation.

8.1 Synthesis and simulation

In this operation the Verilog code has been imported into Cadence environment and has been synthesized at register transfer level (RTL), applying some rules and constrains, through the Genus tool.
In the first step the verilog code is imported and elaborated. After that, power constraints are imposed on the design to ensure leakage and dynamic power respectively below $20nW$ and $15\mu W$. Dynamic power constraint has been chosen through an evaluation of the available power on tag's antenna.
Assuming the condition of free space propagation and unitary antennas gains:

$$P_{av} = P_t \cdot \left(\frac{\lambda_{max}}{4\pi \cdot d}\right)^2 \tag{8.1}$$

Since $P_t = 2W$, $\lambda_{max} = 0.345m$ and assuming an operating range of $2 \div 10m$:

$$P_{av}^{max} = 377\mu W \tag{8.2}$$

$$P_{av}^{min} = 15\mu W \tag{8.3}$$

It can be deduced that the maximum allowed dynamic power was imposed as the minimum available power. Obviously, the available power for digital logic is less than P_{av} since some power is lost on the rectifier and DC/DC stages. However, $15\mu W$ was chosen as initial parameter to avoid a too restrictive condition, which afterwards will be modified to ensure better performance. The next step consisted in compiling the constraints file. In this file generally are included:

- All signals on clock input of all registers of entire network
- The system clock skew admissible on setup and hold time
- The transition time range admissible of system clock
- All networks which won't be modified or replaced during optimization.
- All paths on which timing constrain are disabled because are asynchronous with system clock and don't affect system operations
- The input delay range admissible of all logic's input pins
- The transition time range admissible of all logic's input pins
- The output delay range admissible of all logic's output pins
- The expected driver resistance of all logic's input pins
- The expected output capacitance of all logic's output pins

These information are used to drive the tool during the synthesis process and to check that the network working correctly during place and route process.
After that the synthesis was performed through the $180nm$ CMOS technology, the result of this process is showed in figure 8.1.

A more accurate simulation was performed thanks to the mathematical model included in the physical libraries. These model describe the physical behaviour of each elementary block which composes the digital logic. During this phase each elementary blocks is connected with ideal wires, therefore a more accurate simulation will be executed after placement and routing operations. The simulation was

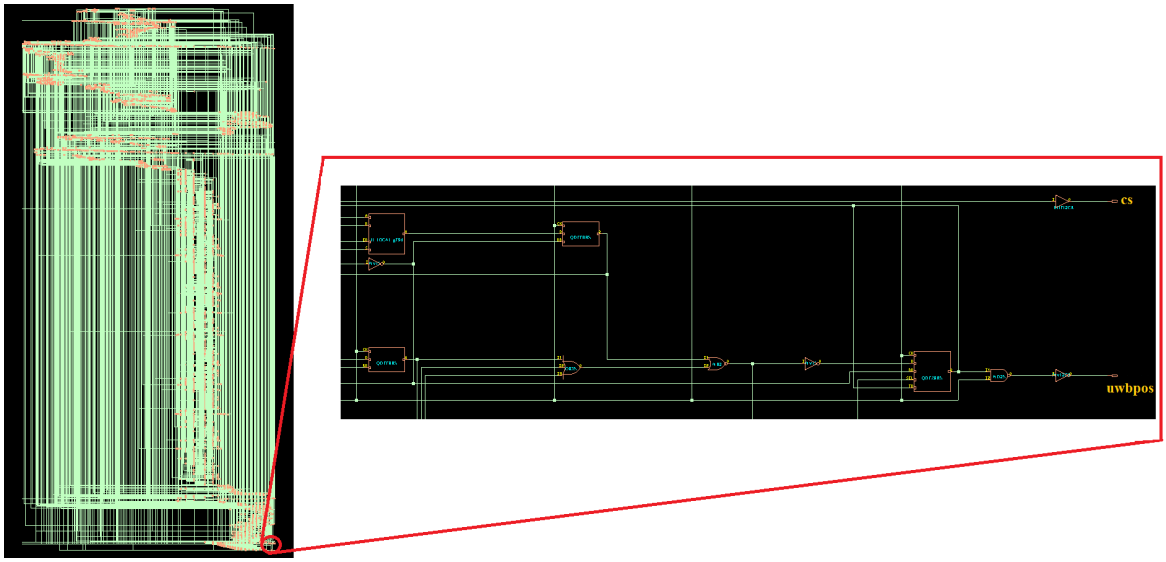

Figure 8.1: Result of synthesis process and zoom on a small area

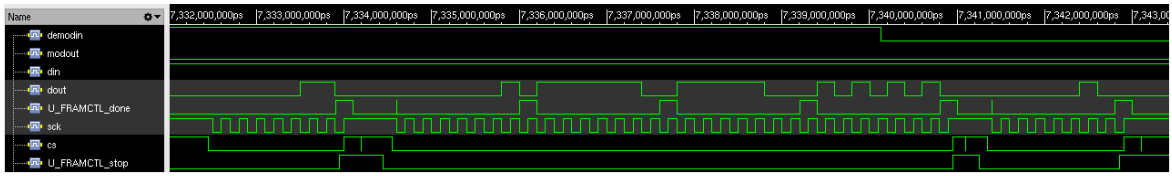

Figure 8.2: Post synthesis simulation. The simulation shows the FRAM signals exchanged during a write command and the glitches that affect cs and done signals.

performed through the same testbench used previously. The result of post-synthesis simulation showed the presence of some glitches on signals which did not emerge during simulation on Altera Modelsim 10.1d. These glitches are shown in figures 8.2. An estimate of power consumption and chip size has been performed after the synthesis process. The estimate is based on the digital library made available with the $180nm$ CMOS process design kit over a $12ms$ of simulation performed with the usual testbench. The chip size obtained is $0.097mm^2$, the leakage and dynamic power obtained are respectively $0.381\mu W$ and $141.7\mu W$. Since this power values depend on simulation time length, is appropriate to convert them in energy values. Therefore leakage energy amounts to $4.6pJ$ and dynamic energy result to be $1.7\mu J$.

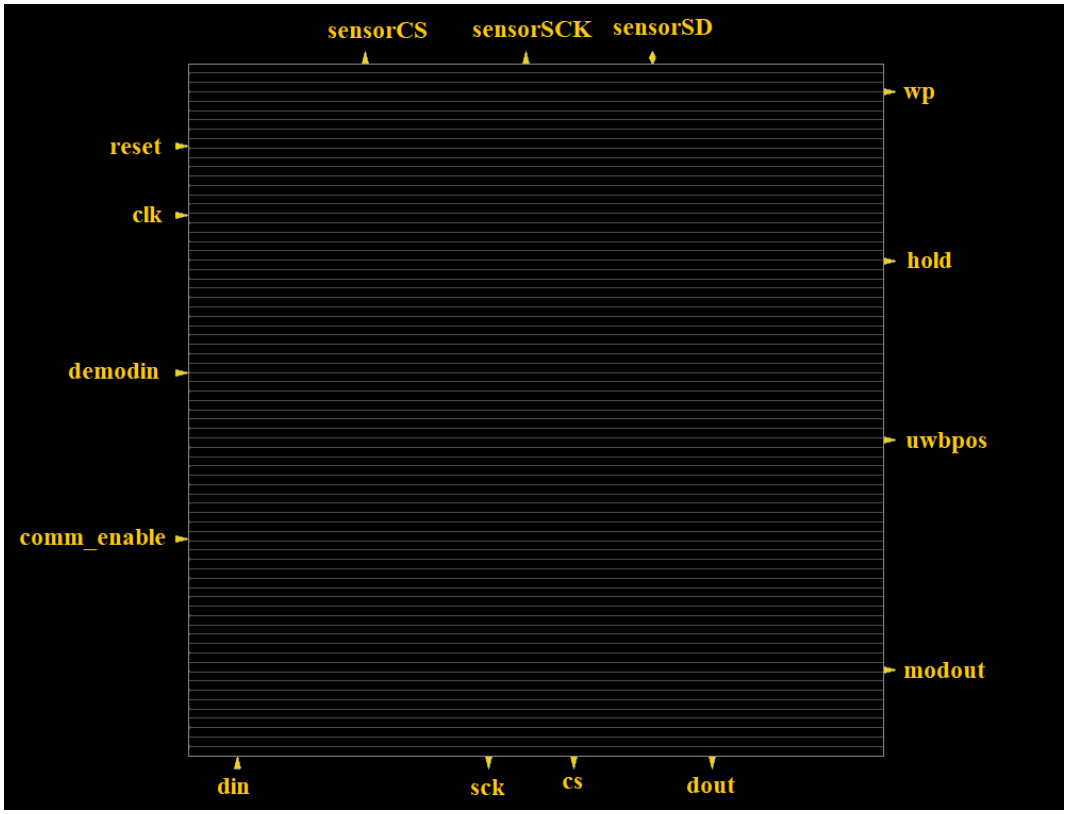

Figure 8.3: Initial floorplan in which are defined only chip edges and I/O pins position.

8.2 Placement

The purpose of this step is to build the floorplan of synthesized chip. The floor-planning is a process in which the standard cells are placed in order to meet time constraints and to occupy less area. Firstly, the tool estimates the chip dimension and traces its borders. I/O pins were distributed on chip edges and after the placement operation they were manually placed in order to simplify the I/O interconnection and reduce their length. The initial floorplan is showed in figure 8.3.
Placement is a step of placing the standard cell in a standard cell rows in order to:

- minimize the total wirelength
- ensure that critical paths don't exceed the the maximum specified delay
- avoid congested region in which excessive routing detours or make it impossible to complete all routes

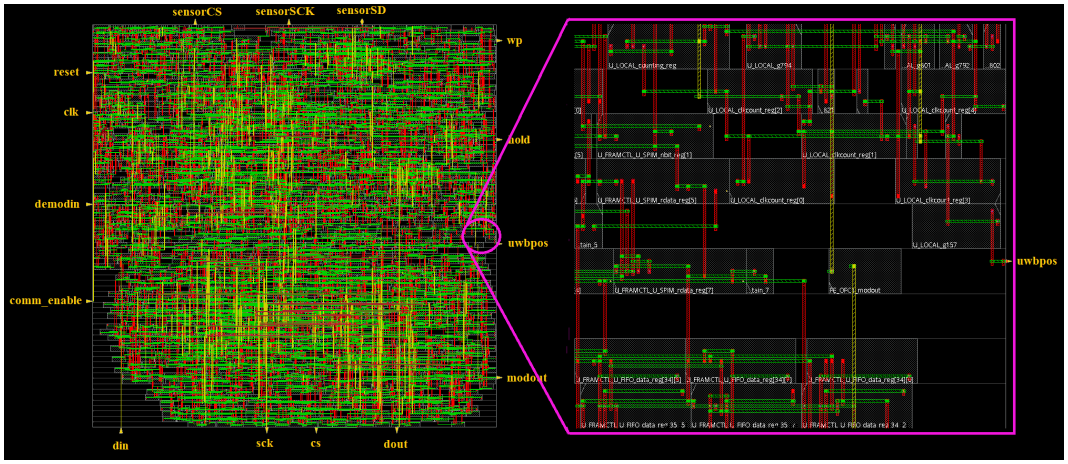

Figure 8.4: Result of placement process and zoom on a small area

- minimize power consumption

The result of placement step is showed in figure 8.4.

8.3 Clock tree synthesis and Routing

After placing the cells, the clock tree has been synthesized. Clock Tree Synthesis (CTS) is a process which makes sure that the clock gets distributed evenly to all sequential elements in a design. The goal of CTS is to minimize the skew and latency. Clock tree can be built including clock tree inverters so as to ensure clock transition constrain. Clock tree balancing is done by clock tree buffers which tune clock skew and latency to meet the requirements. For this reason neighboring cells may have clock sources that have passed through a different number of buffers.
The result of this steps is showed in figure 8.5.

After that routing has been performed which is the process of establishing physical connections between various components in the design. To correctly connect the nets, routers should ensure that the design meets timing, has no crosstalk problems, meets any metal density requirements, does not suffer from antenna effects, and so on. Routing process is divided into three steps:

- **Global Routing**:It is done in the placement stage. In Global routing, the core area is divided into Global Routing Cells (GRC). In this stage only routing resources are allocated. Global routing assigns nets to specific GRC but it does not define the specific tracks for each of them.

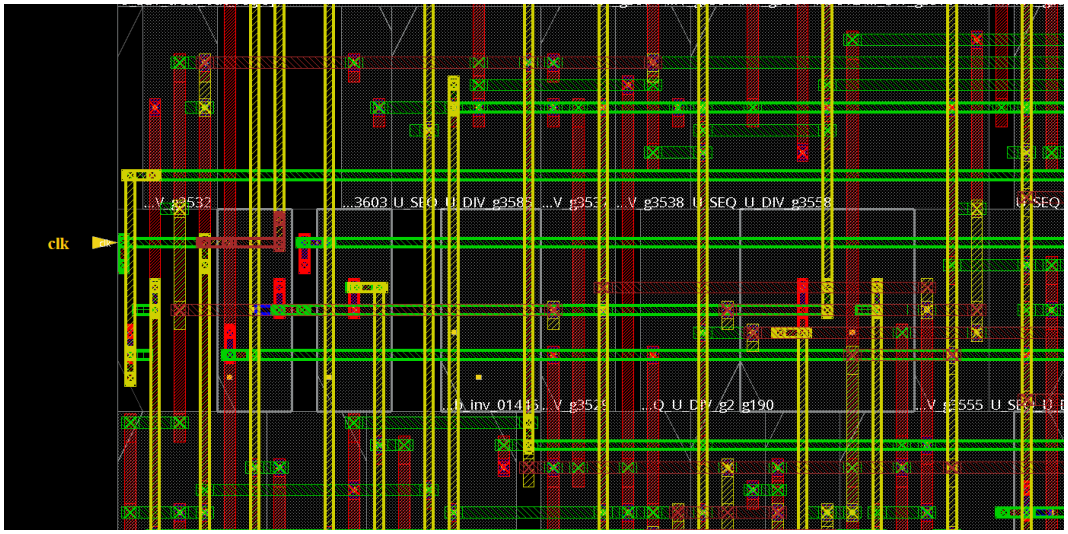

Figure 8.5: Zoom on clock input port after CTS step. Yellow and green track added represent the clock tree branches

- **Track Assignment**: Tracks will be assigning to all metal layers through each GRC. Routing will be done but pin to pin connection isn't yet executed.
- **Detail routing**: The actual routing takes place, the physical connections are made i. e. it will create actual metal and via connections.

The chip configuration is showed in figure 8.6.

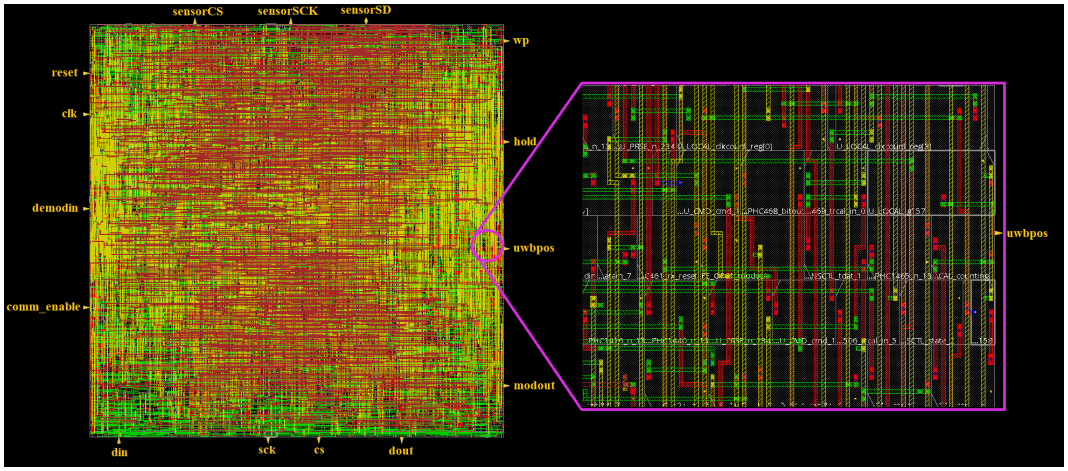

Figure 8.6: Result of routing process and zoom on a small area

Chapter 9

Conclusion

The RFID system, initially developed only for identification, are becoming more appealing over the years with the addiction of sensing and positioning capabilities which makes this technology ready for a large number of application. A future vision of RFID technology is provided by RAIN RFID global alliance[12] which promoting the universal adoption of UHF RFID technology in a way similiar to other wireless technology organizations including NFC Forum, WiFi Alliance and Bluethoot SIG. The purpose of RAIN alliance is to connect items through RFID tag to enable businesses and consumers to identify, locate, authenticate and engage it in our everyday world.
The work presented in this thesis aims to develop the RFID technology in order to reach the set future goals. In this work an existing firmware has been improved to store data received by readers, data acquired by sensor in a non volatile memory and to add the UWB localization. Many simulation and test on FPGA have been performed to validate the work and finally a CMOS synthesis of HDL firmware has been introduced. Many future developments are available for this projects:

- Optimize the verilog code of digital logic
- add CRC check by the tag
- add error replies by the tag in case of problem
- add the compatibility of all EPC gen2 protocol commands
- Optimize the chip synthesis on Cadence
- Integrate the entire tag platform on chip except the antennas

Bibliography

[1] D. Fabbri, E. Berthet-Bondet, D. Masotti, A. Costanzo, D. Dardari, and A. Romani, *Long Range Battery-Less UHF-RFID Platform for Sensor Applications.*

[2] "Wisp project." Avaiable at http://www.wispsensor.net/home.

[3] *EPC™ Radio-Frequency Identity Protocols Generation-2 UHF RFID.* Available at https://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf.

[4] "Open source rfid sensor tag verilog code." Avaiable at http://www.wispsensor.net/rfid_verilog_code.

[5] D. Yeager, F. Zhang, A. Zarrasvand, N. T. George, T. Daniel, and B. P. Otis, *A 9uA, Addressable Gen2 Sensor Tag for Biosignal Acquisition.* Vol. 45, IEEE Journal of Solid-State Circuits (JSSC), No. 10, 2010.

[6] D. Yeager, *Development and Application of Wirelessly Powered Sensor Nodes.* University of Washington, 2009.

[7] "Radio-frequency identification." Avaiable at https://en.wikipedia.org/wiki/Radio-frequency_identification.

[8] K. Ohno and T. Ikegami, *Interference Mitigation Study for UWB RadioUsing Template Waveform Processing.* IEEE TRANSACTIONS ON MICROWAVE THEORY AND TECHNIQUES, VOL. 54, NO. 4, APRIL 2006.

[9] STMicroelectronics, *Capacitive digital sensor for relative humidity and temperature.* HTS221 Datasheet.

[10] G. Santhosh, "Master spi verilog." Avaiable at https://opencores.org/projects/spi_verilog_master_slave.

[11] "Etsi en 302 208 v3.1.0 (2016-02)." Avaiable at https://www.etsi.org.

[12] "Rain rfid technology." Avaiable at https://rainrfid.org.

[13] *Characteristics of ultra-wideband technology.* RECOMMENDATION ITU-R SM.1755-0, avaiable at https://www.itu.int/dms_pubrec/itu-r/rec/sm/R-REC-SM.1755-0-200605-I!!PDF-E.pdf.

[14] "Vlsi basic." Avaiable at https://vlsibasic.blogspot.com.

[15] "Placement (electronic design automation)." Avaiable at https://en.wikipedia.org/wiki/Placement_(electronic_design_automation).

[16] "Routing (electronic design automation)." Avaiable at https://en.wikipedia.org/wiki/Routing_(electronic_design_automation).

[17] "Clock tree synthesis." Avaiable at https://www.techdesignforums.com/practice/guides/clock-tree-synthesis-distribution-strategies/.

List of Figures

Appendix A

Verilog code of main blocks implemented

A.1 spi_master.v

```
/*************************************************
* Author: Nicholas Battistini
* Data: 25/10/2019
*************************************************/

// Developed on the base of:

//////////////////////////////////////////////////////////////////////
////                                                              ////
//// Project Name: SPI (Verilog)                                  ////
////                                                              ////
//// Module Name: spi_master                                      ////
////                                                              ////
////                                                              ////
////  This file is part of the Ethernet IP core project           ////
////  http://opencores.com/project,spi_verilog_master_slave       ////
////                                                              ////
////  Author(s):                                                  ////
////      Santhosh G (santhg@opencores.org)                       ////
////                                                              ////
////  Refer to Readme.txt for more information                    ////
////                                                              ////
////////////////////////////////////////////////////////////////////// ////
//// Copyright (C) 2014, 2015 Authors                             ////
////                                                              ////
```

```
//// This source file may be used and distributed without         ////
//// restriction provided that this copyright statement is not  ////
//// removed from the file and that any derivative work contains
////
//// the original copyright notice and the associated disclaimer. ////
////                                                              ////
//// This source file is free software; you can redistribute it ////
//// and/or modify it under the terms of the GNU Lesser General ////
//// Public License as published by the Free Software Foundation; ////
//// either version 2.1 of the License, or (at your option) any ////
//// later version.                                               ////
////                                                              ////
//// This source is distributed in the hope that it will be     ////
//// useful, but WITHOUT ANY WARRANTY; without even the implied ////
//// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR     ////
//// PURPOSE.  See the GNU Lesser General Public License for more ////
//// details.                                                     ////
////                                                              ////
//// You should have received a copy of the GNU Lesser General  ////
//// Public License along with this source; if not, download it ////
//// from http://www.opencores.org/lgpl.shtml                     ////
////
//////////////////////////////////////////////////////////////////////
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  SPI MODE 3
                CHANGE DATA @ NEGEDGE
                read data @posedge

 RSTB-active low asyn reset, CLK-clock, T_RB=0-rx  1-TX,
 mlb=0-LSB 1st 1-msb 1st
 START=1- starts data transmission cdiv 0=clk/4 1=/8   2=/16  3=/32
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

// Differences:  1) introduced feature to transmit treg until stop = 1.
// Thank that it can transmit a serial stream of more than 8 bit
//              2) Modified settings of done, ss
//3)            SCK can be also clk/2

module spi_master(rstb,clk,mlb,start,tdat,din,ss,sck,dout,done,rdata,
                  stop);
    input rstb,clk,mlb;
        input start; // start data transmission or reception
        input stop; // stop data transmission or reception
    input [7:0] tdat;  //transmit data
        input din;
        output reg ss;
```

```
        output reg sck;
        output reg dout;
    output reg done;
        output reg [7:0] rdata; //received data

        // FSM
        parameter idle=2'b00;
        parameter send=2'b10;
        parameter finish=2'b11;
        reg [1:0] cur,nxt;

        reg [7:0] treg,rreg; // temp register to store data
        reg [3:0] nbit;
        reg [4:0] cnt;
        reg shift; // shift = 1 -> start sck
        reg data_valid;

// SPI clock generator
//setup falling edge (shift dout) sample rising edge (read din)
always@(negedge clk  or posedge rstb) begin
  if (rstb) begin
        cnt=0; sck=1;
  end else begin
        if(shift==1) begin
                cnt=cnt+5'd1;
          if(cnt==1) begin
                sck=~sck;
                cnt=0;
                end //mid
        end //shift
 end //rst
end //always



//FSM i/o
always @(start or cur or stop or data_valid or nbit) begin

        case(cur)
            idle:begin
                        if(start==1) begin
                                nxt=send;
                                ss=0;
                                done = 0;
                                shift=1;
                        end else begin
```

```
                                ss=1;
                                done = 0;
                                nxt=cur;
                                shift=0;
                        end
                    end //idle
                send:begin
                                done = (nbit == 4'd8) ? 1'b1 : 1'b0;
                                if (stop && data_valid) begin
                                        nxt=finish;
                                        shift=0;
                                        ss=1;
                                end else begin
                                        nxt=cur;
                                        shift=1;
                                        ss=0;
                                end
                        end//send
                finish:begin
                                shift=0;
                                ss=1;
                                done = 0;
                                nxt=idle;
                         end
                        default: begin
                                nxt=idle;
                                shift=0;
                                ss=1;
                                done = 0;
                        end
      endcase
end//always


//state transition (system clock)
always@(negedge clk or posedge rstb) begin
        if(rstb) begin
                        cur     <=idle;
                        rdata <= 0;
        end else if (data_valid) begin
                        cur     <=nxt;
                        rdata <= rreg;
        end else begin
                        cur     <= nxt;
        end
 end
```

```
//sample @ rising edge (read din)
always@(posedge sck or posedge rstb) begin // or negedge rstb
        if(rstb) begin
                rreg = 0;
                data_valid = 0;
        end else begin
                data_valid = (nbit == 4'd8) ? 1'b1 : 1'b0;
                rreg = (mlb == 0) ? {din,rreg[7:1]} : {rreg[6:0],din};
   end //rst
end //always

//sample @ falling edge (write dout)
always@(negedge sck or posedge rstb) begin

    if(rstb) begin
        dout = 0;
        treg = 0;
        nbit = 0;

    end else if(nbit==0) begin // execute only at first start
        treg=tdat;
        dout=mlb?treg[7]:treg[0];
        nbit=nbit+4'd1;
    end else if (nbit==8) begin
        treg=tdat;
        dout=mlb?treg[7]:treg[0];
        nbit=4'd1;
    end else begin
        nbit=nbit+4'd1;
        treg = (mlb==0) ? {1'b1,treg[7:1]} : {treg[6:0],1'b1};
        dout = (mlb==0) ? treg[0] : treg[7];
        end
    end //always

endmodule
```

A.2 FRAMcontroller.v

```
// Author: Nicholas Battistini
// Data: 23/10/2019
```

```
module FRAMcontroller(reset, clk, din, dout, sck, cs, wp, hold,
readwords, rw_address, wdata, fifo_dataout, write_rq, read_rq,
next_out, new_data);

  input  reset, clk;
  input din; // serial data received from FRAM
  output dout; // serial data transmitted to FRAM
  output sck; // SPI clock
  output cs; // FRAM chip select
  output wp; // FRAM write protect
  output hold; // FRAM hold
  input [7:0] rw_address; // read or write address
  input [15:0] wdata; // data to write in FRAM
  output [7:0] fifo_dataout; // data reads from FRAM
  input [7:0] readwords;
  input write_rq; // write mode
  input read_rq; //  read mode
  input next_out; // shift of fifo buffer
  output new_data;

  // instruction bits
  parameter WREN           = 8'b00000110;
  parameter WRDI       = 8'b00000100;
  parameter READ       = 8'b00000011; //  the A8 bit is always 0
  parameter WRITE      = 8'b00000010; //  the A8 bit is always 0

  // state
  parameter IDLE                        = 3'd0;
  parameter INSTRUCTION         = 3'd1;
  parameter     ADDRESS                 = 3'd2;
  parameter     RX_DATA                 = 3'd3;
  parameter     TX_DATA                 = 3'd4;
  parameter     W_ENABLE                = 3'd5;
  parameter     W_DISABLE               = 3'd6;
  parameter     STOP                    = 3'd7;

  reg next_in;
  wire [7:0] data_in;
  reg start;
  reg stop;
  reg new_data;
  wire done;
  wire [1:0] cdiv;
  reg [2:0] state;
  reg [4:0] count;
```

```
 reg [4:0] fifo_count;
 reg [7:0] tdat;
 wire [7:0] fifo_datain;
 wire [8:0] readbytes;
 reg flag_w_en;
 reg flag_newstate;
 reg count_en;
 wire mlb;
 // Fifo unused pins:
 wire firstbyte, restart, empty, full;


 spi_master U_SPIM(reset, clk, mlb, start, tdat, din, cs, sck, dout,
 done, fifo_datain, stop); // sck = clk/2

 fifo U_FIFO(reset, fifo_datain, fifo_dataout, next_in, next_out,
 empty, full, firstbyte, restart); // stores data read from FRAM

assign firstbyte = 0;
assign restart = 0;
assign hold = 1;
assign wp = 1;
assign mlb = 1; // set MSB first
assign readbytes = (readwords == 0) ? 9'd10 : (readwords << 1);

// Counter to mange next_in command
 always @ (posedge clk or posedge reset) begin
       if (reset)
               fifo_count <= 5'd0;
       else if (count_en)
               fifo_count <= fifo_count + 5'd1;
       else
               fifo_count <= 5'd0;
 end

 always @ (posedge clk or posedge reset) begin
if (reset) begin
       state   <= 0;
       stop    <= 0;
       tdat    <= 0;
       count   <= 0;
       start   <= 0;
       next_in <= 0;
       flag_w_en <= 0;
       flag_newstate <= 0;
       count_en <= 0;
```

```
        new_data <= 0;

end else begin

case(state)
        IDLE: begin
                if (read_rq) begin
                        state <= INSTRUCTION;
                        flag_newstate <= 1;
                end else if (write_rq) begin
                        state <= W_ENABLE;
                        flag_newstate <= 1;
                end else
                        state <= IDLE;
                        flag_newstate <= 0;
        end

        W_ENABLE: begin
                if (done && !flag_newstate) begin
                        state <= STOP;
                        flag_newstate <= 1;
                        flag_w_en <= 1;
                        tdat <= 8'd0; //new data to transmit
                        count <= 0;
                        stop    <= 1;
                end else begin
                        if (count == 5'd4) begin
                                start <= 0;
                                tdat <= WREN;
                                stop <= 0;
                                flag_newstate <= 0;
                        end else begin
                                start <= 1;
                                tdat <= WREN;
                                stop <= 0;
                                count <= count + 5'd1;
                                flag_newstate <= 0;
                                end
                end

        end

        STOP: begin
                if (count == 5'd4) begin
                        stop <= 0;
                        count <= 0;
```

```
                        if (flag_w_en) begin
                                state <= INSTRUCTION;
                                flag_newstate <= 1;
                        end else begin
                                state <= W_DISABLE;
                                flag_newstate <= 1;
                        end
                end else begin
                        count <= count +5'd1;
                        stop <= 1;
                end
        end

        INSTRUCTION: begin
                if (done && !flag_newstate) begin
                        state <= ADDRESS;
                        flag_newstate <= 1;
                        tdat <= rw_address; //new data to transmit
                        count <= 0;
                end else begin
                        if (count == 5'd4) begin
                                start <= 0;
                                stop  <= 0;
                                flag_newstate <= 0;
                                if (read_rq)
                                        tdat <= READ;
                                else if (write_rq)
                                        tdat <= WRITE;
                                else
                                        tdat <= 8'd0;

                        end else begin
                                start <= 1;
                                stop  <= 0;
                                flag_newstate <= 0;
                                count <= count + 5'd1;
                                if (read_rq)
                                        tdat <= READ;
                                else if (write_rq)
                                        tdat <= WRITE;
                                else
                                        tdat <= 8'd0;
                                end
                end

        end
```

```
ADDRESS: begin
        if (done && read_rq && !flag_newstate) begin
                tdat  <= 8'd0; // nothing to transmit
                state <= RX_DATA;
                flag_newstate <= 1;
        end else if (done && write_rq && !flag_newstate) begin
                tdat <= wdata[15:8];
                state <= TX_DATA;
                flag_newstate <= 1;
        end else begin
                tdat <= rw_address;
                stop <= 0;
                flag_newstate <= 0;
        end
end


RX_DATA: begin
        if (done && !flag_newstate) begin
                flag_newstate <= 1;
                count_en <= 1;
                if (count == readbytes - 1'd1)  begin
                        stop    <= 1;
                end else begin
                        count <= count + 5'd1;
                        end
        end else begin
                flag_newstate <= 0;
                if (fifo_count == 2) begin
                        next_in <= 1;
                        count_en <= 0;
                end else
                        next_in <= 0;
        end
end

TX_DATA: begin
        if (done && !flag_newstate) begin
                flag_newstate <= 1;

                if (count[0] == 0)   begin
                        tdat <= wdata[7:0];
                        count_en <= 1;
                end else tdat <= wdata[15:8];
```

```
                    if (count == readbytes - 1'd1) begin
                            tdat  <= 8'd0; // nothing to transmit
                            flag_w_en <= 0;
                            count_en <= 0;
                            state <= STOP;
                    end else begin
                            count <= count + 5'd1;
                            end
            end else begin
                    flag_newstate <= 0;
                    if (fifo_count == 2 && count[0] == 1) begin
                            new_data <= 1;
                            count_en <= 0;
                    end else
                            new_data <= 0;
            end
    end

    W_DISABLE: begin
            if (done && !flag_newstate) begin
                    tdat <= 8'd0; //no data to transmit
                    count <= 5'd31;
                    stop    <= 1;
                    flag_newstate <= 1;
            end else begin
                    if (count == 5'd4) begin
                            start <= 0;
                            tdat <= WRDI;
                            stop <= 0;
                            flag_newstate <= 0;
                    end else if (count == 5'd31) begin
                            tdat <= 8'd0; //no data to transmit
                            stop    <= 1;
                            flag_newstate <= 1;
                    end else begin
                            start <= 1;
                            tdat <= WRDI;
                            stop <= 0;
                            flag_newstate <= 0;
                            count <= count + 5'd1;
                            end
            end

    end

    default begin
```

```
                stop <= 0;
                state <= IDLE;
                flag_newstate <= 0;
        end
endcase
end

end

endmodule
```

A.3 SENSORcontroller.v

```
// Author: Nicholas Battistini
// Data: 5/12/2019

module SENSORcontroller(clk, reset, sensorCS, sensorSCK, sensorDATA,
FRAM_access, oe, write_mem, next_out);

input clk, reset;
input next_out;    //
output sensorCS;   // chip select per il sensore SPI
output sensorSCK;  // chip select per il sensore SPI
output FRAM_access;
output oe;
output [15:0] write_mem; // data reads from SENSOR
inout sensorDATA;

wire  sensorCS;
reg FRAM_access;
reg [2:0] state;
reg [4:0] count;
wire sensorSCK;
reg start;
reg flag_newstate;
reg [2:0] phase;
reg oe;
reg [15:0] H_OUT;
reg [15:0] T_OUT;
reg [15:0] write_mem;

// SPI master connections:
reg [7:0] tdat;
```

```
reg stop;
wire [7:0] rdat;
wire mlb, done, din, cs, sck, dout;



  // state
  parameter W_AV_CONF           = 3'd0;
  parameter W_CTRL_REG1         = 3'd1;
  parameter     W_CTRL_REG2             = 3'd2;
  parameter     W_CTRL_REG3             = 3'd3;
  parameter     R_STATUS_REG    = 3'd4;
  parameter     R_HUM_TEMP              = 3'd5;
  parameter     FINISH                  = 3'd6;

  // Dati da scrivere
  parameter AV_CONF_ADDRESS    = 8'h10;
  parameter AV_CONF_DATA       = 8'h00;
  parameter CTRL_REG1_ADDRESS  = 8'h20;
  parameter CTRL_REG1_DATA     = 8'b10000101;
  parameter CTRL_REG2_ADDRESS  = 8'h21;
  parameter CTRL_REG2_DATA       = 8'b00000001;  e
  parameter CTRL_REG3_ADDRESS  = 8'h22;
  parameter CTRL_REG3_DATA     = 8'b00000000;
  parameter STATUS_REG_ADDRESS = 8'h27;
  parameter HUM_TEMP_ADDRESS   = 8'hE8;

spi_master U_SPIM(reset, clk, mlb, start, tdat, din, cs, sck, dout,
done, rdat, stop);


assign mlb = 1;
// serial clock for SPI sensorCS
assign sensorSCK = sck;
assign sensorCS = cs;

assign sensorDATA = oe ? dout : 1'bz;
assign din = !oe ? sensorDATA : 1'bz;


always @ (negedge clk or posedge reset) begin
if (reset) begin
        FRAM_access <= 0;
        count <= 0;
        start <= 0;
        flag_newstate <= 1;
```

```
        phase <= 0;
        state <= W_AV_CONF;
        oe <= 0;
        stop <= 0;
        tdat <= 0;
        write_mem <= 0;
        H_OUT <= 0;
        T_OUT <= 0;

end else begin
case(state)

    W_AV_CONF: begin
        if (done && !flag_newstate) begin

                        if (phase == 1) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= AV_CONF_DATA;
                                count <= 0;
                                stop <= 1;
                        end else begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= AV_CONF_DATA;
                                count <= 0;
                                stop <= 0;
                        end

        end else begin
                case(phase)
                        0: begin // address
                                if (count == 5'd4) begin
                                        start <= 0;
                                        tdat <= AV_CONF_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                end else begin
                                        start <= 1;
                                        tdat <= AV_CONF_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                        count <= count + 5'd1;
                                        flag_newstate <= 0;
                                        end
                        end
```

```
                        1: begin // data
                                start <= 0;
                                tdat <= AV_CONF_DATA;
                                stop <= 0;
                                oe <= 1;
                                flag_newstate <= 0;
                           end
                        2: begin //stop communication
                                        if (count == 5'd4) begin
                                                stop <= 0;
                                                count <= 0;
                                                phase <= 0;
                                                state <= W_CTRL_REG1;
                                                tdat <= CTRL_REG1_ADDRESS;
                                                flag_newstate <= 1;
                                                oe <= 0;
                                        end else begin
                                                count <= count + 5'd1;
                                                stop <= 1;
                                                oe <= 0;
                                        end
                            end

                endcase
        end
    end


    W_CTRL_REG1: begin
        if (done && !flag_newstate) begin

                        if (phase == 1) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= CTRL_REG1_DATA;
                                count <= 0;
                                stop <= 1;
                        end else begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= CTRL_REG1_DATA;
                                count <= 0;
                                stop <= 0;
                        end

        end else begin
```

```
                case(phase)
                        0: begin // address
                                if (count == 5'd4) begin
                                        start <= 0;
                                        tdat <= CTRL_REG1_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                end else begin
                                        start <= 1;
                                        tdat <= CTRL_REG1_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                        count <= count + 5'd1;
                                        flag_newstate <= 0;
                                        end
                        end
                        1: begin // data

                                start <= 0;
                                tdat <= CTRL_REG1_DATA;
                                stop <= 0;
                                oe <= 1;
                                flag_newstate <= 0;

                                end
                        2: begin //stop communication
                                if (count == 5'd4) begin
                                        stop <= 0;
                                        count <= 0;
                                        phase <= 0;
                                        state <= W_CTRL_REG3;
                                        tdat <= CTRL_REG3_ADDRESS;
                                        flag_newstate <= 1;
                                        oe <= 0;
                                end else begin
                                        count <= count +5'd1;
                                        stop <= 1;
                                        oe <= 0;
                                end
                            end

                endcase
        end
    end

    W_CTRL_REG3: begin
```

```
        if (done && !flag_newstate) begin

                        if (phase == 1) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= CTRL_REG3_DATA;
                                count <= 0;
                                stop <= 1;
                        end else begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                tdat <= CTRL_REG3_DATA;
                                count <= 0;
                                stop <= 0;
                        end

        end else begin
                case(phase)
                        0: begin // address
                                if (count == 5'd4) begin
                                        start <= 0;
                                        tdat <= CTRL_REG3_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                end else begin
                                        start <= 1;
                                        tdat <= CTRL_REG3_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                        count <= count + 5'd1;
                                        flag_newstate <= 0;
                                        end
                        end
                        1: begin // data

                                        start <= 0;
                                        tdat <= CTRL_REG3_DATA;
                                        stop <= 0;
                                        oe <= 1;
                                        flag_newstate <= 0;

                                end
                        2: begin  //stop communication
                                if (count == 5'd4) begin
                                        stop <= 0;
                                        count <= 0;
```

```
                                        phase <= 0;
                                        state <= R_HUM_TEMP;
                                        tdat <= HUM_TEMP_ADDRESS;
                                        flag_newstate <= 1;
                                        oe <= 0;
                                    end else begin
                                        count <= count +5'd1;
                                        stop <= 1;
                                        oe <= 0;
                                    end
                                end

            endcase
        end
    end

    // READ HUMIDITY AND TEMPERATURE
    R_HUM_TEMP: begin
        if (done && !flag_newstate) begin
                        if (phase == 0) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                count <= 0;
                                stop <= 0;
                        end else if (phase == 2) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                count <= 0;
                                stop <= 0;
                            H_OUT[15:8] <= rdat;
                        end else if (phase == 3) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                count <= 0;
                                stop <= 0;
                            H_OUT[7:0] <= rdat;
                        end else if (phase == 4) begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                count <= 0;
                                stop <= 1;
                               T_OUT[15:8] <= rdat;
                        end else begin
                                phase <= phase +3'd1;
                                flag_newstate <= 1;
                                count <= 0;
```

```
                        stop <= 0;
                end

        end else begin
                case(phase)
                        0: begin // address
                                if (count == 5'd4) begin
                                        start <= 0;
                                        tdat <= HUM_TEMP_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                end else begin
                                        start <= 1;
                                        tdat <= HUM_TEMP_ADDRESS;
                                        stop <= 0;
                                        oe <= 1;
                                        count <= count + 5'd1;
                                        flag_newstate <= 0;
                                        end
                        end
                        1: begin // receiving first byte of humidity

                                        start <= 0;
                                        stop <= 0;
                                        oe <= 0;
                                        FRAM_access <= 1;
                                        flag_newstate <= 0;

                                end

                        2: begin // receiving second byte of humidity

                                        start <= 0;
                                        stop <= 0;
                                        oe <= 0;
                                        FRAM_access <= 1;
                                        flag_newstate <= 0;

                                end

                        3: begin // receiving second byte of Temperature

                                        start <= 0;
                                        stop <= 0;
                                        oe <= 0;
                                        FRAM_access <= 1;
```

```
                                        flag_newstate <= 0;

                                end

                        4: begin // receiving second byte of Temperature

                                        start <= 0;
                                        stop <= 0;
                                        oe <= 0;
                                        FRAM_access <= 1;
                                        flag_newstate <= 0;

                                end

                        5: begin //stop communication
                                        if (count == 1) begin
                                                T_OUT[7:0] <= rdat;
                                                state <= FINISH;
                                                write_mem <= H_OUT;
                                                count <= 0;
                                                oe <= 0;
                                                stop <= 1;
                                        end else
                                                FRAM_access <= 1;
                                                count <= count +5'd1;
                                                stop <= 1;
                                                oe <= 0;

                            end
                                endcase
                        end
                end
    FINISH: begin
    if (next_out) begin
                write_mem <= T_OUT;
                stop    <= 1;
                oe <= 0;
    end else
                stop    <= 1;
                oe <= 0;
    end
endcase
end
end
endmodule
```

A.4 localization.v

```
// Author: Nicholas Battistini
// Data: 1/12/2019

module localization(clk, reset, enable, uwbpos);

input clk, reset, enable;
output uwbpos;

wire uwbpos;
reg  [13:0] clkcount;
reg counting;

assign uwbpos = (counting) ? clk : 1'b0;


always @ (posedge clk or posedge reset) begin
  if (reset) begin
               counting  <= 0;
               clkcount  <= 0;
  end else if (clkcount == 14'd10000) begin
               counting <= 0;
               clkcount <= 14'd0;
  end else if (enable) begin
               counting <= 1;
               clkcount <= clkcount + 14'd1;
  end else if (counting) begin
               clkcount <= clkcount + 14'd1;
  end // ~reset
end // always @ clk

endmodule
```

A.5 readmem.v

```
/***********************************************
* File: readmem.v
* Author: Nicholas Battistini
* Modules: readmem
* Description: Block that reads data into a memory at a particular
address and turns them to the transmitter
```

```
* Data: 19/11/2019
* Version: 1.0
**********************************************/

module readmem(reset, readbitclk, datain, readbitout, readbitdone,
                handle, readwords, ack_flag, fifo_nextout);

input  reset, readbitclk;
output readbitout, readbitdone;
output fifo_nextout;
input [7:0]  datain;
input [15:0] handle;
input [7:0]  readwords; // number of words to read
input ack_flag;

wire [8:0] readbytes;
assign     readbytes = (readwords == 0) ? 9'd10 : (readwords << 1);

reg [8:0] bytecounter;
reg [3:0] bitoutcounter;
reg header_flag;

wire readbitout, readbitdone, bytecounterdone;
reg fifo_nextout;

reg send_handle;

assign readbitout      = (bytecounter == 0 && ack_flag == 0) ? 1'b0 :
((send_handle && ack_flag == 0) ? handle[bitoutcounter] :
datain[bitoutcounter[2:0]]);

assign bytecounterdone = (bytecounter >= readbytes &&
(bitoutcounter == 0 || bitoutcounter == 1 && ack_flag == 1)) ||
send_handle;

assign readbitdone     = (send_handle && bitoutcounter == 0);

always @ (posedge readbitclk or posedge reset) begin
  if (reset) begin
    bitoutcounter   <= 0;
    bytecounter     <= 0;
    send_handle     <= 0;
        header_flag     <= 0;
        fifo_nextout    <= 0;

  end else if (header_flag == 0 && ack_flag == 0) begin
```

```
    header_flag     <= 1;

  end else if (bytecounter == 0 && (header_flag == 1 ||
  ack_flag == 1)) begin
    bytecounter     <= 1;
    bitoutcounter   <= 4'd7;
  end else if (!send_handle) begin

    if (bytecounterdone) begin
      send_handle   <= 1;
          fifo_nextout          <= 1;

          if (ack_flag == 1) bitoutcounter <= 4'd0;
      else bitoutcounter <= 4'd15;

    end else if (bitoutcounter == 0) begin
      bitoutcounter <= 4'd7;
      bytecounter   <= bytecounter + 9'b1;
          fifo_nextout          <= 1;

    end else begin
      bitoutcounter <= bitoutcounter - 4'd1;
          fifo_nextout          <= 0;

    end
  end else if (!readbitdone) begin
    bitoutcounter <= bitoutcounter - 4'd1;
        fifo_nextout  <= 0;

  end else begin

  end // ~reset
end

endmodule
```

A.6 writemem.v

```
/***********************************************
* File: writemem.v
* Author: Nicholas Battistini
* Modules: writemem
* Description: Block that writes data into a memory
```

```
* to a particular address
* Data: 02/12/2019
* Version: 2.0
**************************************************/

module writemem(clk, reset, data_in, full_reg, data_out,
readwriteptr, writeaddress, readwritebank);

parameter N = 15;
parameter BANK_WORDS = 6'd32;

input clk, reset;
input data_in; // serial data
input [7:0] readwriteptr;
input [1:0] readwritebank;
output [7:0] writeaddress;
output full_reg;  // for external control
output [15:0] data_out;  // parallel data

wire [7:0] readwriteptr;
//reg [7:0] writeaddress;
wire [7:0] writeaddress;
reg [15:0] data_reg;
wire [15:0] data_out;
reg [4:0]  count_reg;
reg full_reg;

assign writeaddress = (readwriteptr * 8'd2) +
(readwritebank * (BANK_WORDS * 6'd2));

assign data_out = (full_reg == 1) ? data_reg[15:0] : 16'd0;

// save initial and next value in register
always @(negedge clk, posedge reset) begin

        if(reset) begin
                count_reg  <= 0;
                full_reg   <= 0;
                data_reg   <= 0;

        end else begin

                data_reg[N-count_reg] <= data_in;

                if(count_reg == 15) begin
                                count_reg  <= 0;
```

```
                                full_reg   <= 1;

                end else begin  // else continue count
                                count_reg <= count_reg + 5'd1;
                                full_reg  <= 0;
                end
        end
end

endmodule
```