****

Technion

*Electrical Engineering Department*

High Speed Digital System Lab

Project Documentation

**Internal Logic Analyzer Core**

Part B

Date of last update: 19.12.2013Students: Moran Katz, Zvika Pery

Mentor: Moshe Porian

TABLE OF CONTENT

[1 INTRODUCTION..... 6](#_Toc370066502)

[1.1 ABSTRACT 6](#_Toc370066503)

[1.2 PROJECT GOAL 6](#_Toc370066504)

[1.3 PROJECT REQUIERMENTS 6](#_Toc370066505)

[1.4 TOP ARCHITECTURE 7](#_Toc370066506)

[1.5 ENGINEERING TOOLS 7](#_Toc370066507)

[2 GENERAL DESCRIPTION 8](#_Toc370066508)

[2.1 DATA FLOW 8](#_Toc370066509)

[9](#_Toc370066510)

[3 MICRO ARCHITECTURE 11](#_Toc370066511)

[3.1 REGISTERS UNIT 11](#_Toc370066512)

[3.1.1 Description: 11](#_Toc370066513)

[3.1.2 Simulation 13](#_Toc370066514)

[3.2 WRITE CONTROLLER 14](#_Toc370066515)

[3.2.1 Description: 14](#_Toc370066516)

[3.2.2 SIMULATON 19](#_Toc370066517)

[3.3 READ CONTROLLER 19](#_Toc370066518)

[3.3.1 Description: 19](#_Toc370066519)

[3.3.2SIMULATION 23](#_Toc370066520)

[3.4 IN OUT COORDINATOR: 24](#_Toc370066521)

[3.4.1 Description: 24](#_Toc370066522)

[3.4.2 Simulation 26](#_Toc370066523)

[3.5 INPUT>OUTPUT COORDINATOR 27](#_Toc370066524)

[3.5.1 Description: 27](#_Toc370066525)

[INPUT> OUTPUT CORDINATOR FSM 28](#_Toc370066526)

[3.6 INPUT<OUTPUT COORDINATOR 29](#_Toc370066527)

[3.6.1 Description: 29](#_Toc370066528)

[3.7 INPUT=OUTPUT 31](#_Toc370066529)

[3.7.1 Description: 31](#_Toc370066530)

[3.8 ENABLE FSM 33](#_Toc370066531)

[3.8.1 Description: 33](#_Toc370066532)

[3.9 MEMORY UNIT 35](#_Toc370066533)

[3.9.1 Description: 35](#_Toc370066534)

[4 SIMULATIONS 36](#_Toc370066535)

[4.1 Description: 37](#_Toc370066536)

[4.2 EXAMPLE: 39](#_Toc370066537)

[5 PROBLEMS AND SOLUTIONS 42](#_Toc370066538)

[5.1 EXMPLES 43](#_Toc370066539)

[First example: After first trigger rise, the system identify another trigger rise although the data was still recorded 43](#_Toc370066540)

[43](#_Toc370066541)

[45](#_Toc370066542)

[6 WORKING SYSTEMS 45](#_Toc370066543)

[45](#_Toc370066544)

[6.1 SVN 45](#_Toc370066545)

[6.2 CODING GUIDELINES 45](#_Toc370066546)

[6.3 CODE REVIEW 47](#_Toc370066547)

[6.4 DOCUMENTATION 47](#_Toc370066548)

[7 ABBREVIATIONS 47](#_Toc370066549)

TABLE OF FIGURES

[1Figure 1 – top architecture 7](#_Toc370066368)

[2Figure 2- the final system 8](#_Toc370066369)

[3Figure 3 – flow diagram – (1) 9](#_Toc370066370)

[4Figure 4 – flow diagram – (2) 9](#_Toc370066371)

[5Figure 5 – flow diagram – (3) 10](#_Toc370066372)

[6Figure 6 – flow diagram – (4) 10](#_Toc370066373)

[7Figure 7 – flow diagram – (5) 11](#_Toc370066374)

[8Figure 8 – registers unit 12](#_Toc370066375)

[9Figure 9-registers unit simulation 14](#_Toc370066376)

[10Figure 10-write controller 15](#_Toc370066377)

[11Figure 11- write controller FSM 17](#_Toc370066378)

[12Figure 12-write controller simulation 19](#_Toc370066379)

[13Figure 13-read controller 20](#_Toc370066380)

[14Figure 14-read controller FSM 22](#_Toc370066381)

[15Figure 15-read controller simulation 23](#_Toc370066382)

[16Figure 16-in out coordinator 24](#_Toc370066383)

[17Figure 17-in out coordinator simulation 26](#_Toc370066384)

[18Figure 18-in >out coordinator FSM 28](#_Toc370066385)

[19Figure 19-in <out coordinator FSM 30](#_Toc370066386)

[21Figure 21-enable FSM 34](#_Toc370066387)

[22figure 22-memory unit 35](#_Toc370066388)

[23Figure 23-simulation(1) 39](#_Toc370066389)

[24Figure 24-simulation (2) 40](#_Toc370066390)

[25Figure 25-simulation (3) 40](#_Toc370066391)

[26Figure 26-simulation (4) 41](#_Toc370066392)

[27Figure 27-first problem simulation 43](#_Toc370066393)

[28Figure 28-code solution 44](#_Toc370066394)

[29Figure 29-second problem 44](#_Toc370066395)

[30Figure 30-second solution 45](#_Toc370066396)

[31Figure 31-SVN 45](#_Toc370066397)

[32Figure 32-SVN snapshot 45](#_Toc370066398)

[33Figure 33-coding guidelines 46](#_Toc370066399)

[34Figure 34-entity template 46](#_Toc370066400)

TABLE OF TABLES

[1Table 1- registers unit generics 12](#_Toc370066445)

[2Table 2- registers unit signals 13](#_Toc370066446)

[3Table 3- write controller generics 15](#_Toc370066447)

[4Table 4- write controller signals 16](#_Toc370066448)

[5Table 5- write controller output 18](#_Toc370066449)

[6Table 6- read controller generics 20](#_Toc370066450)

[7Table 7- read controller signals 21](#_Toc370066451)

[8Table 8- read controller output 23](#_Toc370066452)

[9Table 9- in out coordinator generics 25](#_Toc370066453)

[10Table 10- in out coordinator signals 25](#_Toc370066454)

[**11**Table 11- in> out coordinator generics 27](#_Toc370066455)

[12Table 12- in> out coordinator signals 27](#_Toc370066456)

[13Table 13- in> out coordinator output 28](#_Toc370066457)

[14Table 14- in< out coordinator generics 29](#_Toc370066458)

[15Table 15- in< out coordinator signals 29](#_Toc370066459)

[16Table 16- in< out coordinator output 30](#_Toc370066460)

[17Table 17- in= out coordinator generics 31](#_Toc370066461)

[18Table 18- in= out coordinator signals 31](#_Toc370066462)

[19Table 19- in= out coordinator output 32](#_Toc370066463)

[20Table 20 – enable generics 33](#_Toc370066464)

[21Table 21 – enable signals 33](#_Toc370066465)

[22Table 22– enable output 34](#_Toc370066466)

[23Table 23 – memory unit generics 35](#_Toc370066467)

[24Table 23 – memory unit signals 36](#_Toc370066468)

[25Table 25- simulations 37](#_Toc370066469)

# 

# 1 INTRODUCTION

## 1.1 ABSTRACT

FPGA's manufacturers supply debugging tool for labs, named Logic Analyzer. It allows recording of internal data in the FPGA and displaying it to the user. The tool is composed of both hardware and software.

The hardware part enters the FPGA's code and includes memories to store the recorded data, logic aimed to configuration changes (e.g. trigger type), also logic that identifies trigger lock, and logic for sending the recorded data to the software.

The software part includes GUI, which allows choosing the type of trigger for the recording, and also the position of the trigger regarding the recorded information. The GUI also allows a convenient display of the recorded signals names and a display of the recording results, which arrive from the hardware, to the user.

The tool of the FPGA manufactures, Altera, is named SignalTap. The Xilinx tool is named ChipScope.

## 1.2 PROJECT GOAL

The goal is to design an Internal Logic Analyzer for the FPGA, which will be independent in the FPGA manufacturer.

The hardware part includes building a system in VHDL, that allows recording of the chosen signals according to configuration and sending the recorded data back to the user.

The software part includes building a GUI, that allows changing configuration and displaying the recorded data to the user.

## 1.3 PROJECT REQUIERMENTS

1. Designated board is an [Altera DE2 board](http://university.altera.com/materials/boards/de2/) that features an [Altera Cyclone® II 2C35 FPGA](http://www.altera.com/products/devkits/altera/kit-nios-2c35.html).
2. The Logic Analyzer will enable:

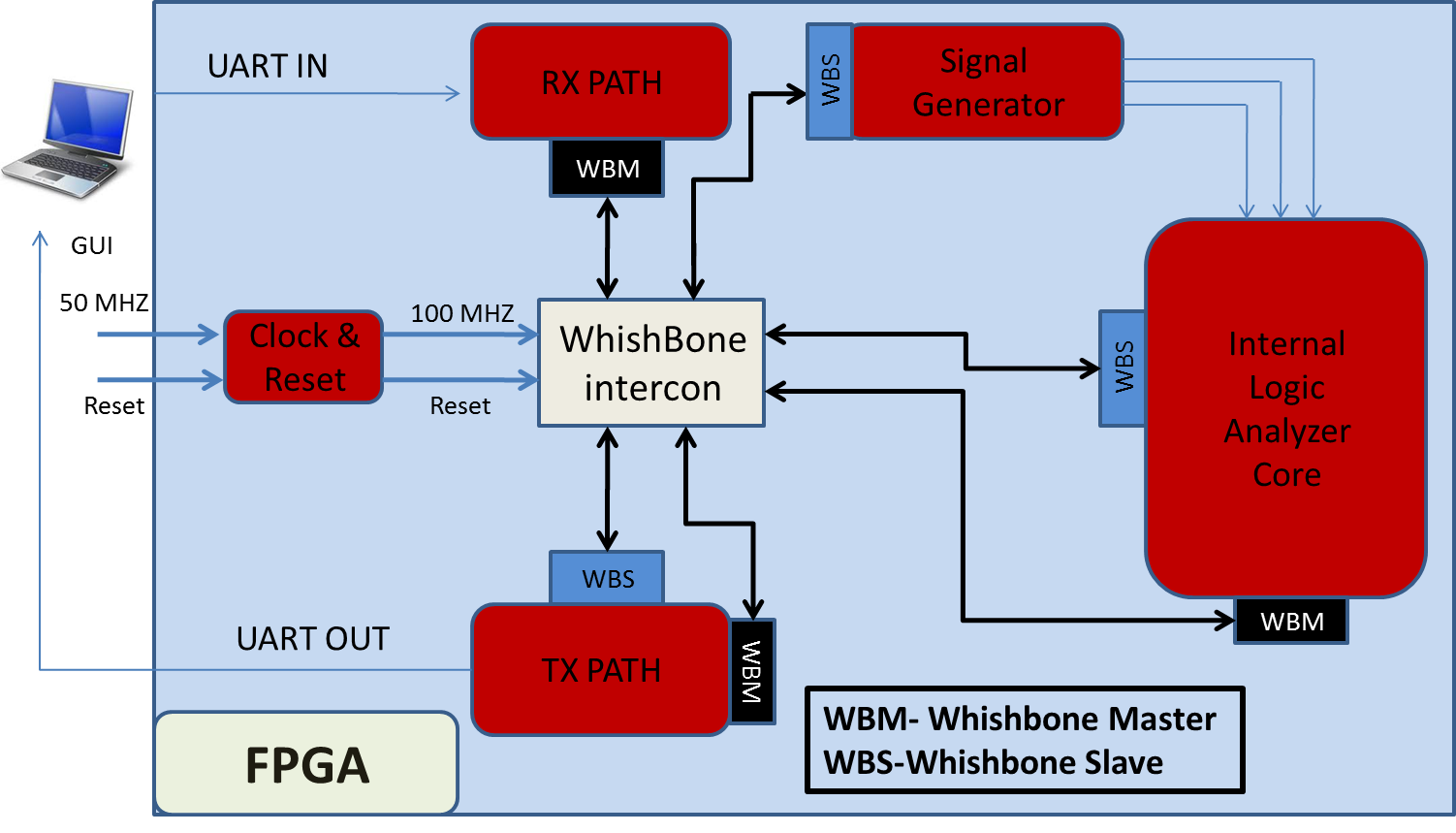
* Choosing the trigger type: rise, fall, high for 3 cycles, low for 3 cycles.
* Choosing the trigger position regarding the recorded data. Trigger position will be a number between 0 to 100.
* Choosing the number of signals for recording.
* Choosing the recording depth (recording time).
* Changing the names of the displayed signals.
* Using memories that are independent of the type of FPGA.

1. The internal communication between the blocks is through Wishbone Protocol.

* Bus width is a generic.

1. The communication between the GUI and the FPGA is through Uart protocol.
2. All the configurations will be saved in the core's registers at the initial stage.
3. Input data and trigger signal will be injected from signal generator every clock cycle.

## 1.4 TOP ARCHITECTURE



## 

1Figure 1 – top architecture

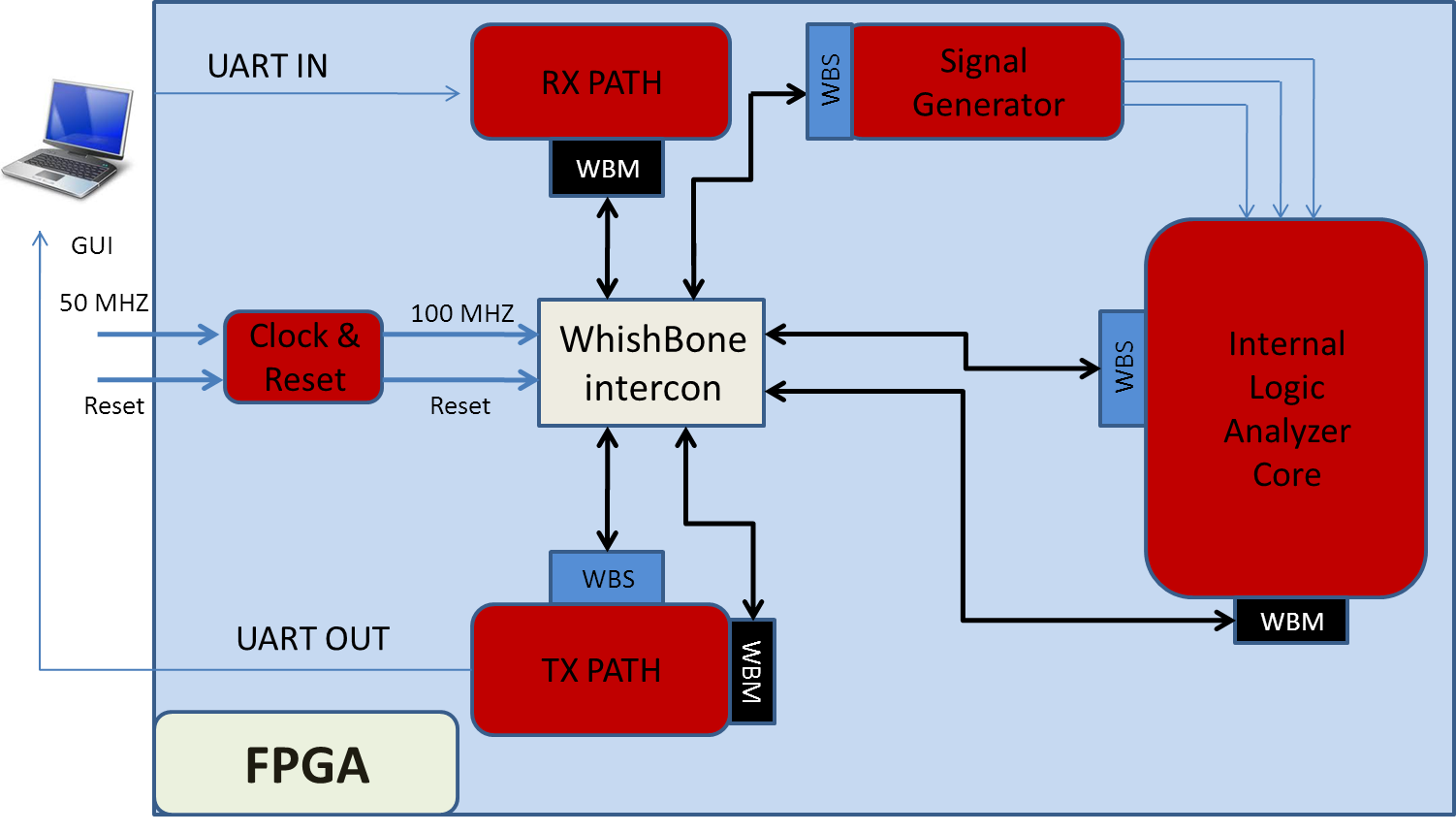
## **1.5 ENGINEERING TOOLS**

The tools which were used in this project are:

1. Altera DE2 Development and Education board with Altera Cyclone II 2C35 FPGA
2. Modelsim10.1d
3. Quartus II 12.1 for Place and Route.
4. Notepad ++
5. SVN

# 2 GENERAL DESCRIPTION

Our project is reusing blocks from other projects, such as: Rx Path, Tx Path,Wishbone Intercon, basic RAM.



2Figure 2- the final system

## 2.1 DATA FLOW

Initial state: User chooses the desired configurations (trigger type, trigger position and enable) using the GUI. Host transmits the information through Uart to the RX path. Then through the wishbone intercon to the WBS of the core.

The configurations are being saved the registers.(1)

Afterwards data and trigger from the signal generator are entering the WC. The WC uses the registers to determine the start address of the data and send it out to the RC. The WC also sends the data and validity signal to the RAM.(2)(3)

The RC uses this address to extract data from the RAM after the WC has finished.(4)

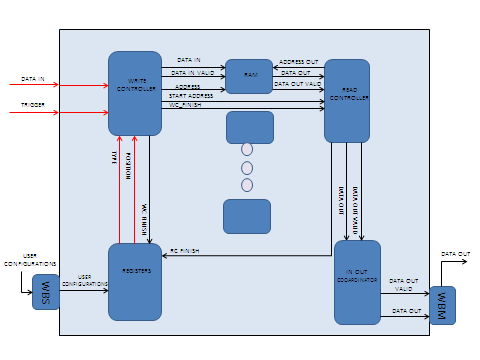
Then the In Out Coordinator will coordinate between the recorded data's width to the width of the bus in the Wishbone protocol.

The data will be sent back to host through WBM.(5)

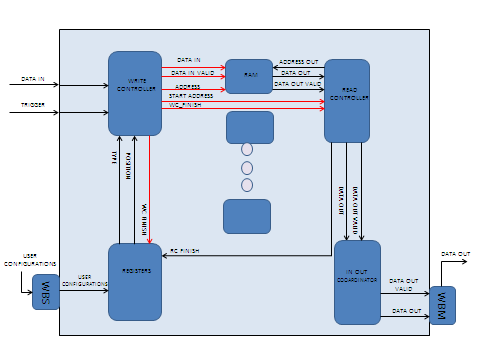
Flow diagram:

# 

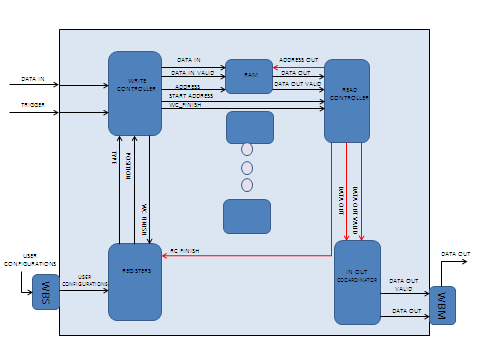
3Figure 3 – flow diagram – (1)



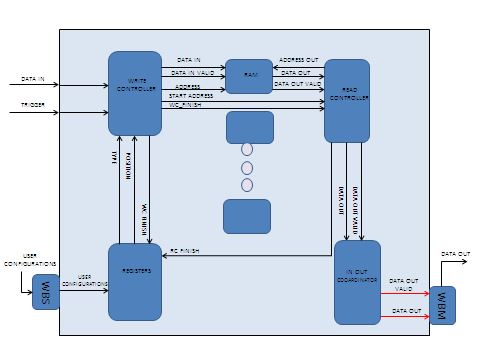
4Figure 4 – flow diagram – (2)



5Figure 5 – flow diagram – (3)



6Figure 6 – flow diagram – (4)



7Figure 7 – flow diagram – (5)

## Reused Blocks Architecture

The Symbol Generator project reused important IPs from a previous "Modular Decompression System" project.

The following description of the reused block includes the original implementation details and also the changes that were made in our project.

### UART Protocol

#### UART Receiver and Transmitter

This component is used for asynchronous serial data channel.

The receiver (UART RX) converts serial start bit, data, parity and stop bit to parallel data.

The transmitter (UART TX) converts parallel data into serial form and automatically adds start bit, parity and stop bit.

Both of the components are based on the same characteristics:

1. Start bit
2. Data length: 8 data bits (between Start and stop bit).
3. Parity bit
4. Stop bit
5. Reset polarity is active low
6. Baud Rate (Transmission rate)
7. System Clock: 100MHz

#### UART Message Pack Description

SOF

Type

Address

Data Length

Data (Payload)

CRC

EOF

Figure - UART Message Pack Structure

The UART message has the following structure:

1. **SOF** – Start of Frame
2. **Type** – Message type
3. **Address** – Address for the data
4. **Length** – Data length. Data length CANNOT be less than 1.
5. **Data** **(Payload)** – The wrapped data
6. **CRC** – CRC of Type, Address, Length and Data blocks
7. **EOF** – End of Frame

For examples of reading/writing data from registers/memory, refer to Appendix B in this document.

### Wishbone Intercon

The Wishbone Intercon consists of a router and an arbiter. The router directs the wishbone signals from the operating master to the chosen slave through a series of muxes. The arbiter is a FSM which enables only one master to use the wishbone bus. Every master that wants to make a transaction asserts CYC. Only the master that is enabled by the arbiter FSM gets its signals passed to the intended slave and can get back an ACK sig.

At the wishbone intercon a watchdog timer was implemented: If a transaction does not end within a TimeOut, determine by generics (clk\_freq\_g/watchdog\_timer\_freq\_g), the master that hold the bus will be reseted by assertion of his ERR\_I signal, and the transaction will end.

The arbiter FSM operates as follow. If no master is using the bus than the master that asserts CYC firsts can use the bus. If more than one master requests the bus at the same time, the priority is:

WM 1 -> WM 2 -> WM3

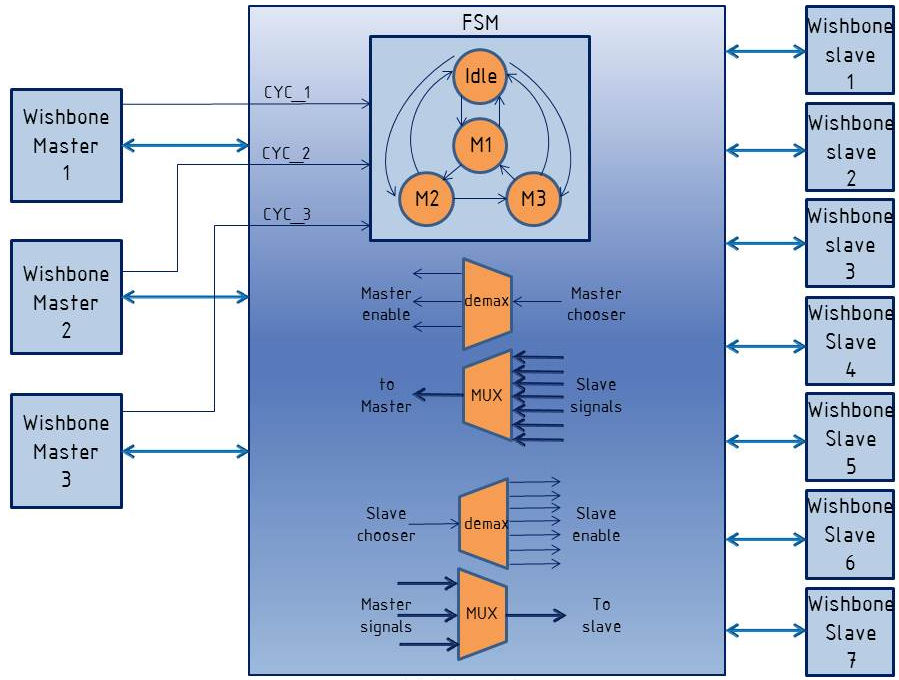


Figure - Wishbone Intercon

### Wishbone Master

The Wishbone Master is the unit that initiates and manages a transaction on the wishbone bus. The wishbone mode supported is pipeline mode defined in the Wishbone B4 spec. Therefore, a respond to Wishbone request can be replied as soon as one cycle after it is broadcast on a bus.

**Connecting a Wishbone Master:**

To make a transactions using a wishbone master a requesting unit should be connected and a RAM. The requesting unit should supply the following data to make a transaction:

1. Assert wm\_start for 1 cycle.
2. When asserting wm\_start - provide valid values on:

- wr – ‘1’ for write transaction. ‘0’ for read transaction.

- len\_in – length of the read/write.

- type\_in – client being accessed in the transaction

- addr\_in – address being accessed in the transaction

- ram\_start\_addr – the RAM address WM will write values received in a read transaction or read from in a write transaction.

1. When transaction ends wm\_end output would be asserted for 1 cycle to notify that the WM has ended the current transaction and is ready for another one.

An example of how to connect a wishbone master is shown in the following figure.

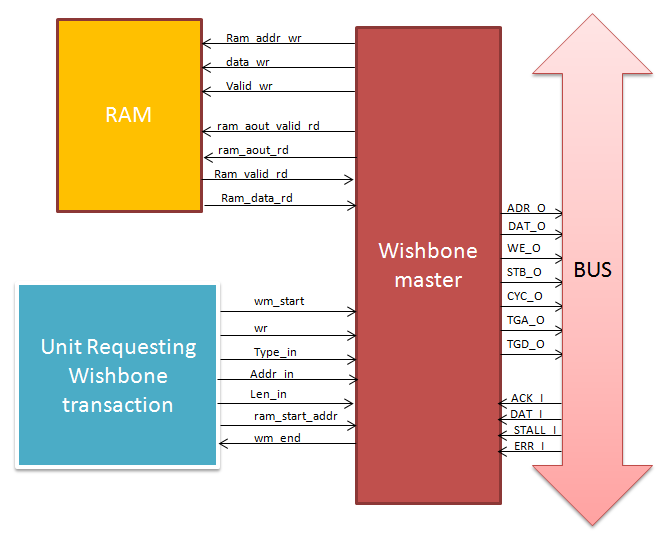


Figure - Connecting a Wishbone Master

**Important signals:**

**STALL\_I -**  A signal received from a client indicating the client is not ready for a transaction. When stall is asserted WM holds the transaction and waits for STALL\_I to be de-asserted.

**ERR\_I –** A signal received from Wishbone Intercon and indicates a bus timeout. When ERR\_I is asserted the WM is reseted and the transaction ends. The wm\_end output is asserted by the WM to notify the requesting unit that the transaction ended.

The Wishbone Master FSM consists of 2 main branches:

Read – for a reading transaction

Write – for a writing transaction

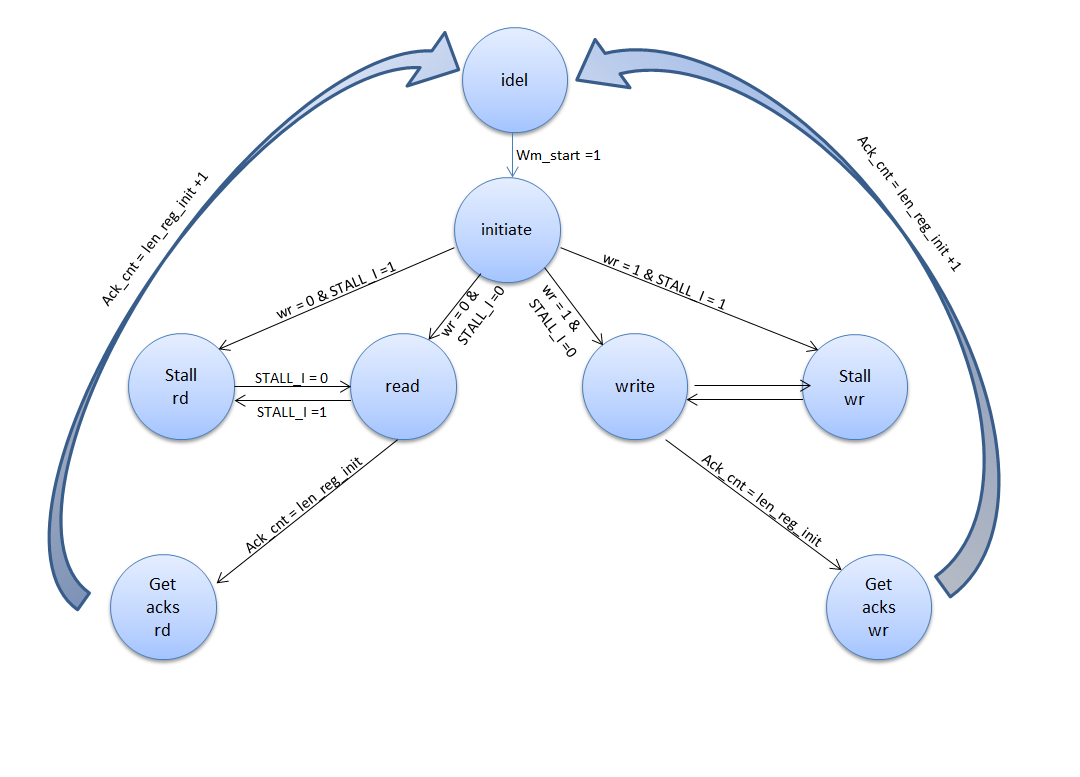


Figure - Wishbone Master FSM

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| ADR\_O | output | addr\_d\_g \* data\_width\_g | contains the address word |
| DAT\_O | output | data\_width\_g | contains the data\_in word |
| WE\_O | output | 1 | '1' for write, '0' for read |
| STB\_O | output | 1 | 1' for active bus operation, '0' for no bus operation |
| CYC\_O | output | 1 | 1' for bus transmition request, '0' for no bus transmition request |
| TGA\_O | output | type\_d\_g \* data\_width\_g | contains the type word |
| TGD\_O | output | len\_d\_g \* data\_width\_g | contains the length word |
| ACK\_I | input | 1 | 1' when valid data is received from WS or for successful write operation in WS |
| DAT\_I | input | data\_width\_g | data received from WS |
| STALL\_I | input | 1 | STALL - WS is not available for transaction |
| ERR\_I | input | 1 | Watchdog interrupts, resets wishbone master |
| ram\_addr | output | addr\_bits\_g | RAM Input address |
| ram\_dout | output | data\_width\_g | RAM Input data |
| ram\_dout\_valid | output | 1 | RAM Input data valid |
| ram\_aout | output | addr\_bits\_g | RAM Output address |
| ram\_aout\_valid | output | 1 | RAM Output address is valid |
| ram\_din | input | data\_width\_g | RAM Output data |
| ram\_din\_valid | input | 1 | RAM Output data valid |
| wm\_start | input | 1 | when '1' WM starts a transaction |
| wr | input | 1 | determines if the WM will make a read('0') or write('1') transaction |
| type\_in | input | type\_d\_g \* data\_width\_g | type is the client which the data is directed to |
| len\_in | input | len\_d\_g \* data\_width\_g | length of the data (in words) |
| addr\_in | input | addr\_d\_g \* data\_width\_g | the address in the client that the information will be written to |
| ram\_start\_addr | input | addr\_bits\_g | start address for WM to read from RAM |
| wm\_end | output | data\_width\_g | when '1' WM ended a transaction or reseted by watchdog ERR\_I signal |

Table - Wishbone Master interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Actual value** | **description** |
| reset\_polarity\_g | std\_logic | 1 | '0' = Active Low, '1' = Active High |
| data\_width\_g | natural | 8 | defines the width of the data lines of the system |
| type\_d\_g | positive | 1 | Type Depth |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |
| addr\_bits\_g | positive | 8 | Log2 of the size of the RAM connected to master |

Table - Wishbone Master generics

### 

### Wishbone Slave

Wishbone Slave is the client’s interface to the system. The unit is based on the same principles as the wishbone master. It has the simplest hardware that can handle the wishbone communication. Therefore it only responds to wishbone master signals and passes any information received to its host’s units without any processing.

Important: The Wishbone mode of work implemented is **pipeline** mode. Therefore , the slave automatically responds to a write transaction with an ACK after one cycle.

A read request could be answered by the client at any time.

**Connecting a Wishbone Slave:**

A Wishbone Slave could be connected to a register unit, RAM or any other unit that could save the data once it is broadcasted on the bus.

The WS signals should be connected as followed:

Always connected: Reg\_data, reg\_data\_valid, wr\_en, ws\_data, ws\_data\_valid, address.

Optional: typ, len, active\_cycle

Stall signal: The stall signal should be driven by the client or alternatively connected constantly to ‘0’.

An example for a connection of a Wishbone Slave to a registers unit is shown in the following figure.

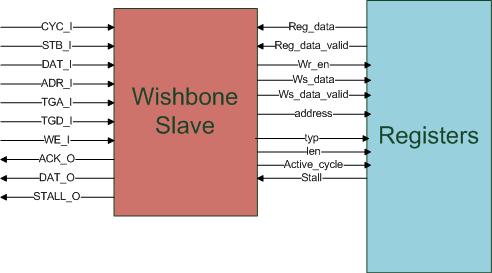


Figure - Connecting a Wishbone Slave

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| ADR\_I | input | (data\_width\_g)\*(addr\_d\_g) | contains the addr word |
| DAT\_I | input | data\_width\_g | contains the data\_in word |
| WE\_I | input | 1 | '1' for write, '0' for read |
| STB\_I | input | 1 | '1' for active bus operation, '0' for no bus operation |
| CYC\_I | input | 1 | '1' for bus transmition request, '0' for no bus transmition request |
| TGA\_I | input | (data\_width\_g)\*(type\_d\_g) | contains the type word |
| TGD\_I | input | (data\_width\_g)\*(len\_d\_g) | contains the len word |
| ACK\_O | output | 1 | '1' when valid data is transmitted to MW or for successful write operation |
| DAT\_O | output | data\_width\_g | data transmit to MW |
| STALL\_O | output | 1 | STALL - WS is not available for transaction |
| typ | output | (data\_width\_g)\*(type\_d\_g) | Type |
| addr | output | (data\_width\_g)\*(addr\_d\_g) | the beginning address in the client that the information will be written to |
| len | output | (data\_width\_g)\*(len\_d\_g) | Length |
| wr\_en | output | 1 | data out to registers |
| ws\_data | output | data\_width\_g | write data |
| ws\_data\_valid | output | 1 | data valid to registers |
| reg\_data | input | data\_width\_g | data to be transmitted to the WM |
| reg\_data\_valid | input | 1 | data to be transmitted to the WM validity |
| active\_cycle | output | 1 | CYC\_I outputed to user side |
| stall | input | 1 | stall - suspend wishbone transaction |

Table - whishbone slave interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Actual value** | **description** |
| reset\_polarity\_g | std\_logic | 1 | '0' = Active Low, '1' = Active High |
| data\_width\_g | natural | 8 | defines the width of the data lines of the system |
| type\_d\_g | positive | 1 | Type Depth |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |

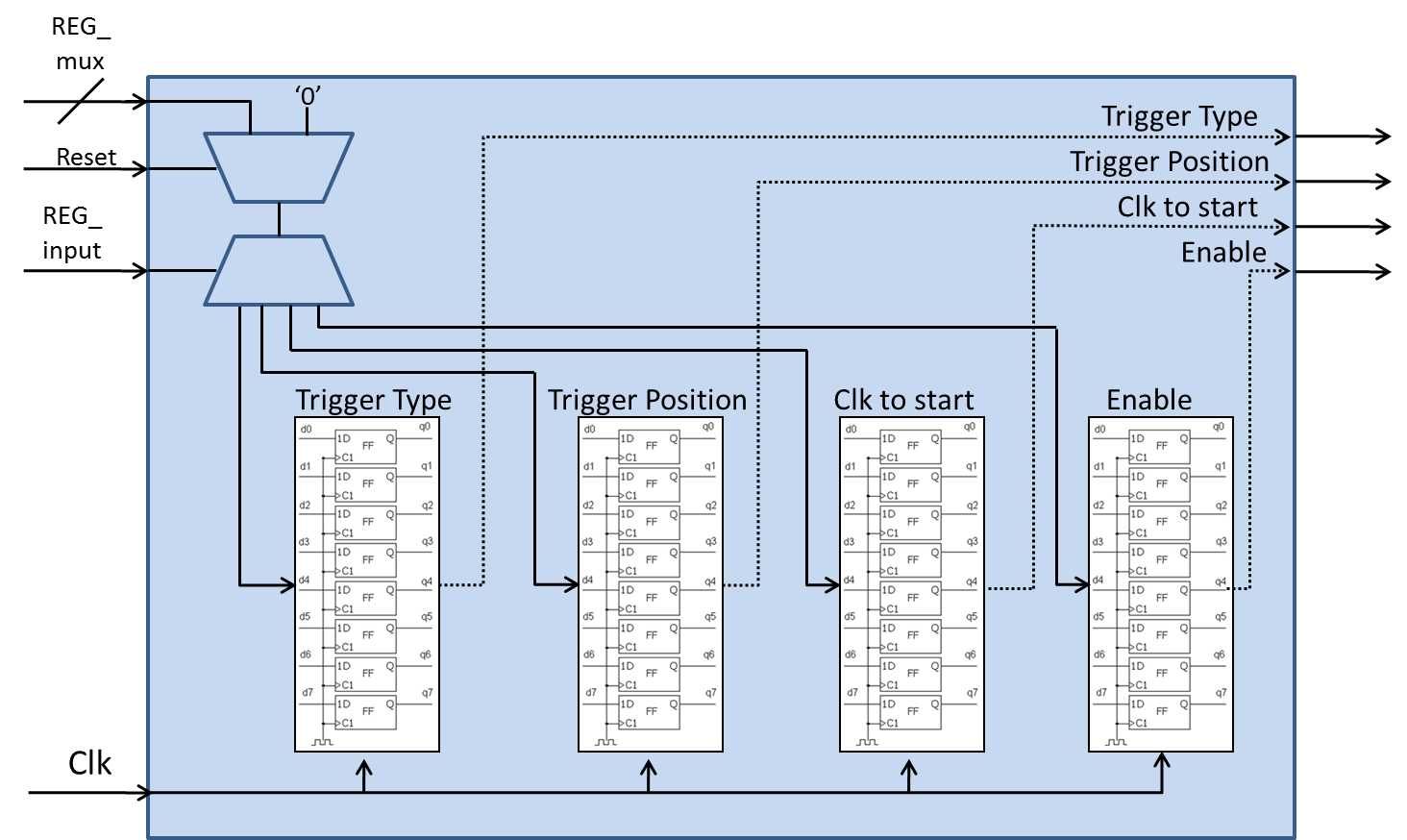
Table - Wishbone Slave generics

# **3 MICRO ARCHITECTURE**

## 3.1 REGISTERS UNIT

### 3.1.1 Description:

The Core Registers unit receives data from the wishbone slaves, samples it, and transmits it to the core blocks. When reset is activated no register should be enabled. The register's addresses are defined by generics.



8Figure 8 – registers unit

The unit contains four registers which are configured as follow:

trigger\_type\_reg\_1 = trigger\_type

השינוי באות הטריגר שהמערכת תגדיר כמציאת טריגר.

מקבל את הערכים הבאים:

0 (ערך ברירת המחדל)- סוג הטריגר מוגדר כעליה של אות הטריגר מנמוך לגבוה.

1. סוג הטריגר מוגדר כירידה מגבוה לנמוך.
2. סוג הטריגר מוגדר כגבוה, כלומר שלושה מחזורים בהם נדגם אות טריגר גבוה.
3. סוג הטריגר מוגדר כנמוך, כלומר שלושה מחזורים בהם נדגם אות טריגר נמוך.

trigger\_position\_reg\_2 = trigger\_position

מספר בין 0-100, מהווה את האחוזים של המידע מסך כל המידע המוקלט (שהוא 2^recorded\_depth\_g) שיוקלטו לפני עליית הטריגר.

clk\_to\_start\_reg\_3 = clk\_to\_start

לא בשימוש בסופו של דבר בפרויקט

enable\_reg\_4 = enable

מקבל את הערכים 0/1. אות ENABLE, מוגדר בהתאם ל enable\_polarity\_g.

Generic table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Width | | Description |
| reset\_polarity\_g | 1 | | '1' reset active high, '0' active low |
| enable\_polarity\_g | 1 | | '1' the entity is active high, '0' entity is active low |
| data\_width\_g | 8 | | defines the width of the data lines of the system |
| Add\_width\_g | 8 | | width of address word in the system |
|  | Type | Default Value |  |
| en\_reg\_address\_g | STD\_LOGIC | 0 | Enable the registers for reading |
| trigger\_type\_reg\_1\_address\_g | STD\_LOGIC\_VECTOR | 1 | Address of the type register |
| trigger\_position\_reg\_2\_address\_g | STD\_LOGIC\_VECTOR | 2 | Address of position register |
| clk\_to\_start\_reg\_3\_address\_g | STD\_LOGIC\_VECTOR | 3 | Address of counter register |
| enable\_reg\_address\_4\_g | STD\_LOGIC | 4 | Address of enable register |

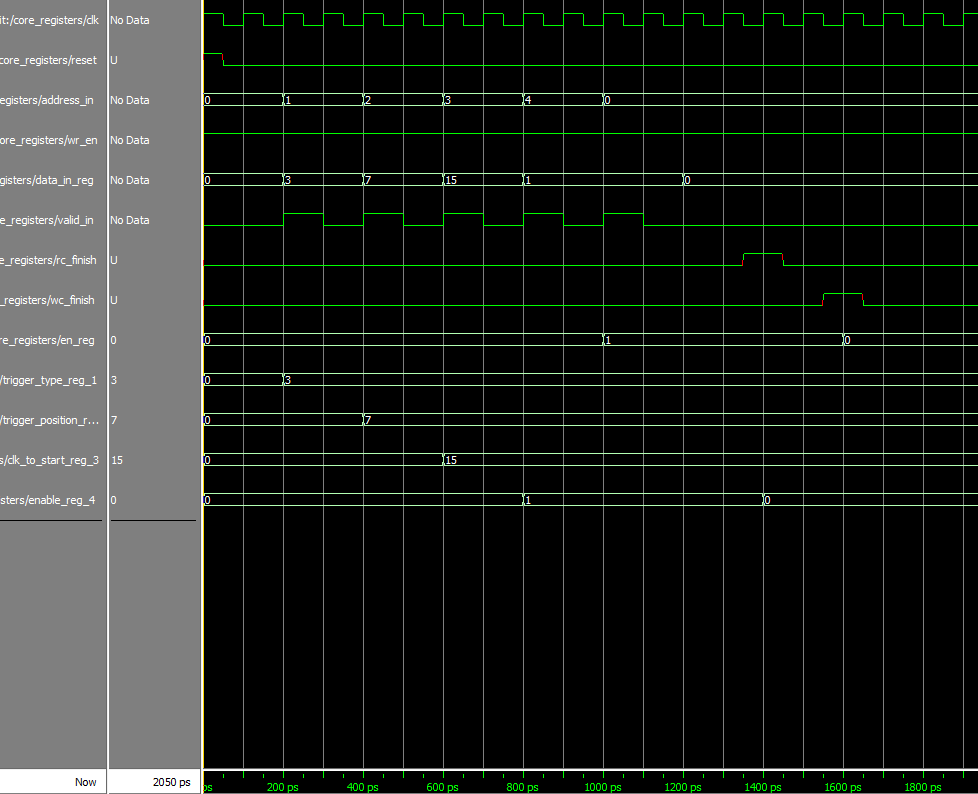
1Table 1- registers unit generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| address\_in | In | Add\_width\_g | Address of the register that the data is writing to. (determine by the generics) |
| wr\_en | In | 1 | write enable: '1' for write, '0' for read |
| data\_in\_reg | In | data\_width\_g | data sent from WS to store in the registers |
| valid\_in | In | 1 | Data in valid |
| rc\_finish | In | 1 | When signal rise we reset enable register |
| wc\_finish | In | 1 | After write controller finish, we need to input a new configurations |
| en\_out | Out | 1 | Enable reading from registers |
| trigger\_type\_out\_1 | Out | 7 | Trigger type |
| trigger\_positionout\_2 | Out | 7 | Trigger position |
| clk\_to\_start\_out\_3 | Out | 7 | Trigger counter |
| enable\_out\_4 | Out | 1 | System enable |

2Table 2- registers unit signals

### 3.1.2 Simulation



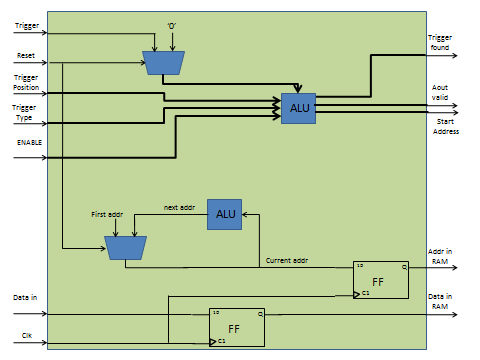
9Figure 9-registers unit simulation

We can see a data that is written to each one of the registers. In addition we can see that after wc\_finish signal rise, the registers are not enabled- register number 0 is resetting into 0 (the configuration need to be enabled again), and when rc\_finish signal rise (system output all the data back to the user), the enable register is resetting back into 0, (register number 4).

## 3.2 WRITE CONTROLLER

### 3.2.1 Description:

1. The entity gets the incoming data and trigger signals from the signal generator, calculating the address in the RAM that the data will be saved in, and sending the data and address to the RAM.
2. According the configurations that are saved in the core registers, detecting trigger rise and calculating the start address of the outputting data, and send it to the read controller.



10Figure 10-write controller

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| enable\_polarity\_g | 1 | '1' the entity is active high, '0' entity is active low |
| signal\_ram\_depth\_g | 3 | depth of single RAM is 2^signal\_ram\_depth\_g |
| signal\_ram\_width\_g | 8 | width of single RAM |
| record\_depth\_g | 4 | number of bits that are recorded from each signal is 2^record\_depth\_g |
| data\_width\_g | 8 | defines the width of the data lines of the system |
| Add\_width\_g | 8 | width of addr word in the RAM |
| num\_of\_signals\_g | 8 | number of signals that will be recorded simultaneously |

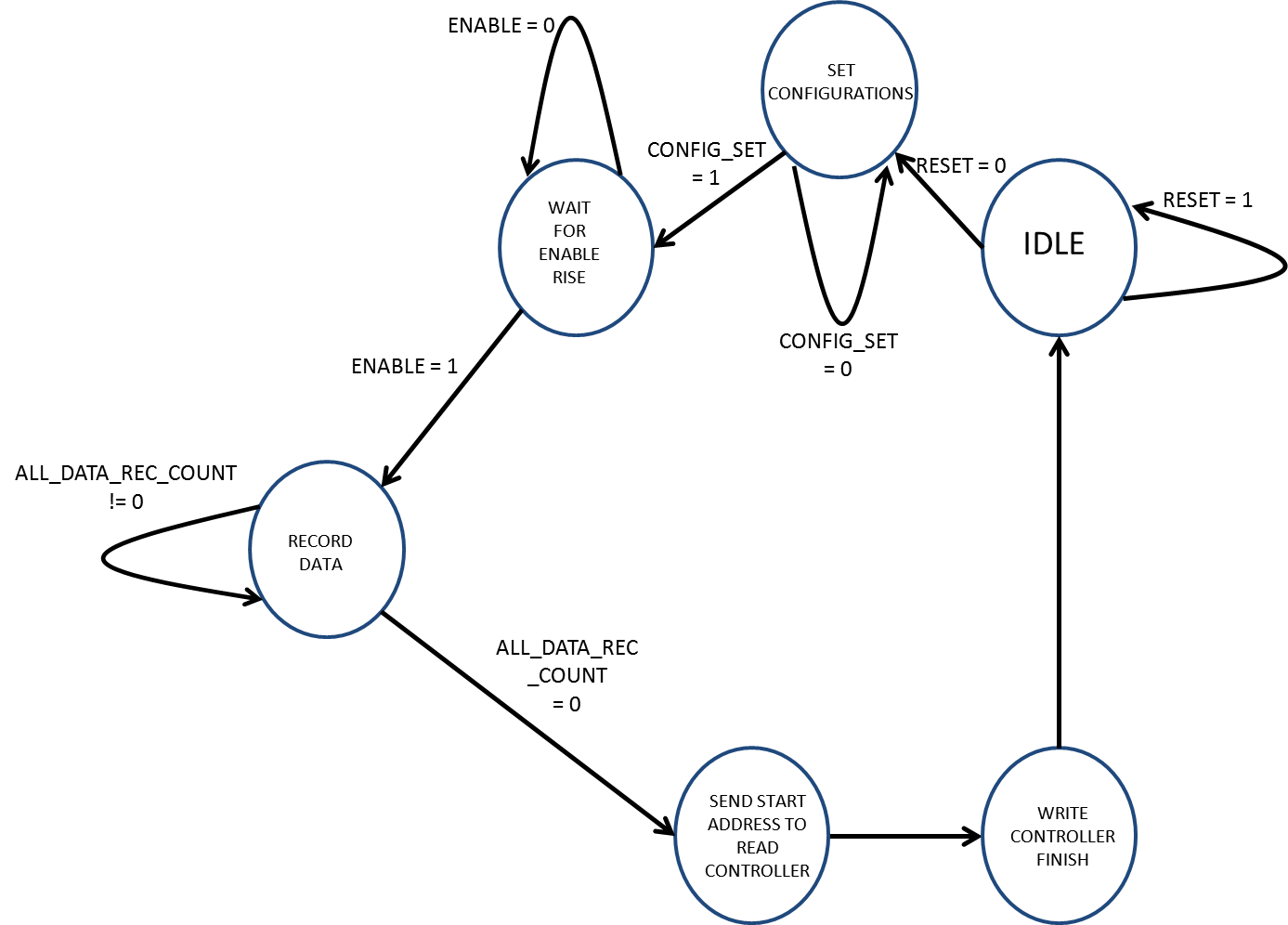
3Table 3- write controller generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| Enable | In | 1 | Enabling the entity. if (enable = enable\_polarity\_g) -> start working, else-> do nothing |
| trigger\_position\_in | In | 7 | The percentage of the data that is recorded before trigger rise |
| trigger\_type\_in | In | 7 | Type of trigger |
| config\_are\_set | In | 1 | '1'-> configurations from registers are ready to be read (trigger position + type). '0'->config are not ready |
| data\_in | In | num\_of\_signals\_g | Data in from signal generator |
| Trigger | In | 1 | Trigger signal from signal generator |
| data\_out\_of\_wc | Out | num\_of\_signals\_g | sending the data to be saved in the RAM |
| addr\_out\_to\_RAM | Out | Add\_width\_g | the address in the RAM to save the data |
| write\_controller\_finish | Out | 1 | '1' ->WC has finish working and saving all the relevant data (RC will start work), '0' ->WC is still working |
| start\_addr\_out | Out | Add\_width\_g | The start address of the data that we need to send out to the user |
| din\_valid | Out | 1 | Data to RAM valid |

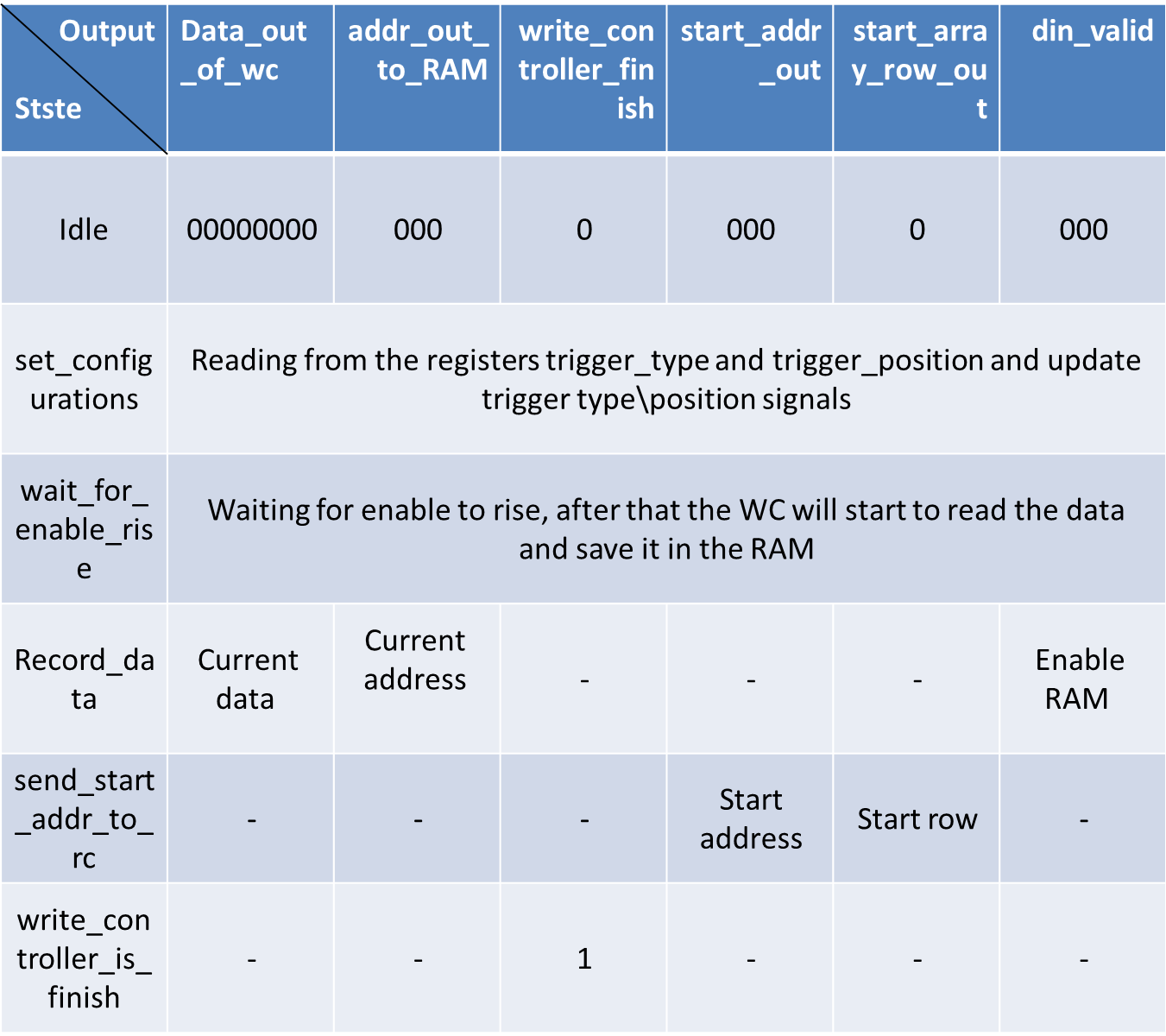
4Table 4- write controller signals

Write controller FSM



11Figure 11- write controller FSM

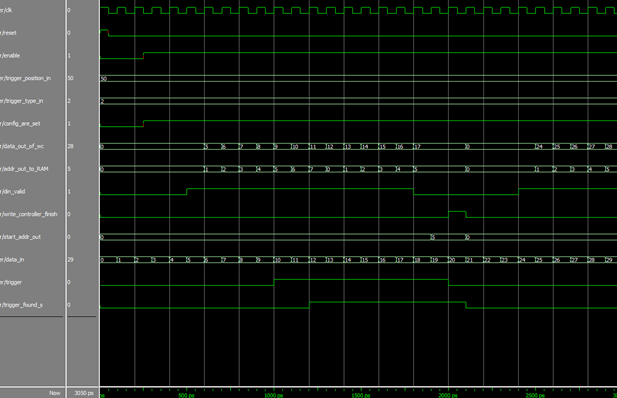
Output table



5Table 5- write controller output

### 3.2.2 SIMULATON

### 



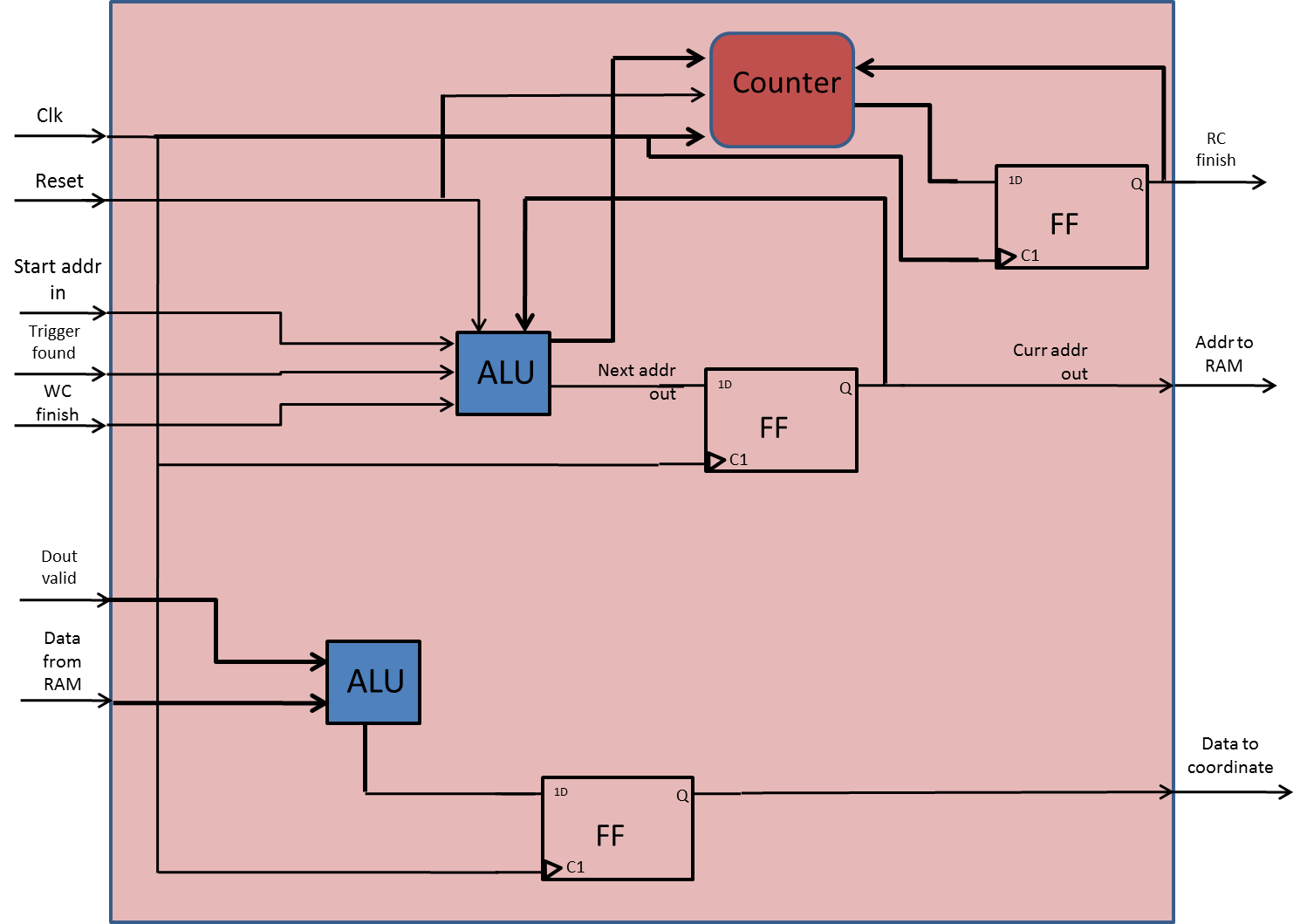
12Figure 12-write controller simulation

Trigger and data are entering each cycle. Data address and validity are being calculated and are being sent to the RAM. Trigger is compared to the configuration to identify trigger rise. If necessary start address is calculated according to the position and is being sent out

## 3.3 READ CONTROLLER

### 3.3.1 Description:

The read controller gets the start address of the valid data that was calculated in the write controller and extract the correct data from the RAM and send it out.



13Figure 13-read controller

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| record\_depth\_g | 4 | number of bits that are recorded from each signal is 2^record\_depth\_g |
| data\_width\_g | 8 | defines the width of the data lines of the system |
| num\_of\_signals\_g | 8 | number of signals that will be recorded simultaneously |

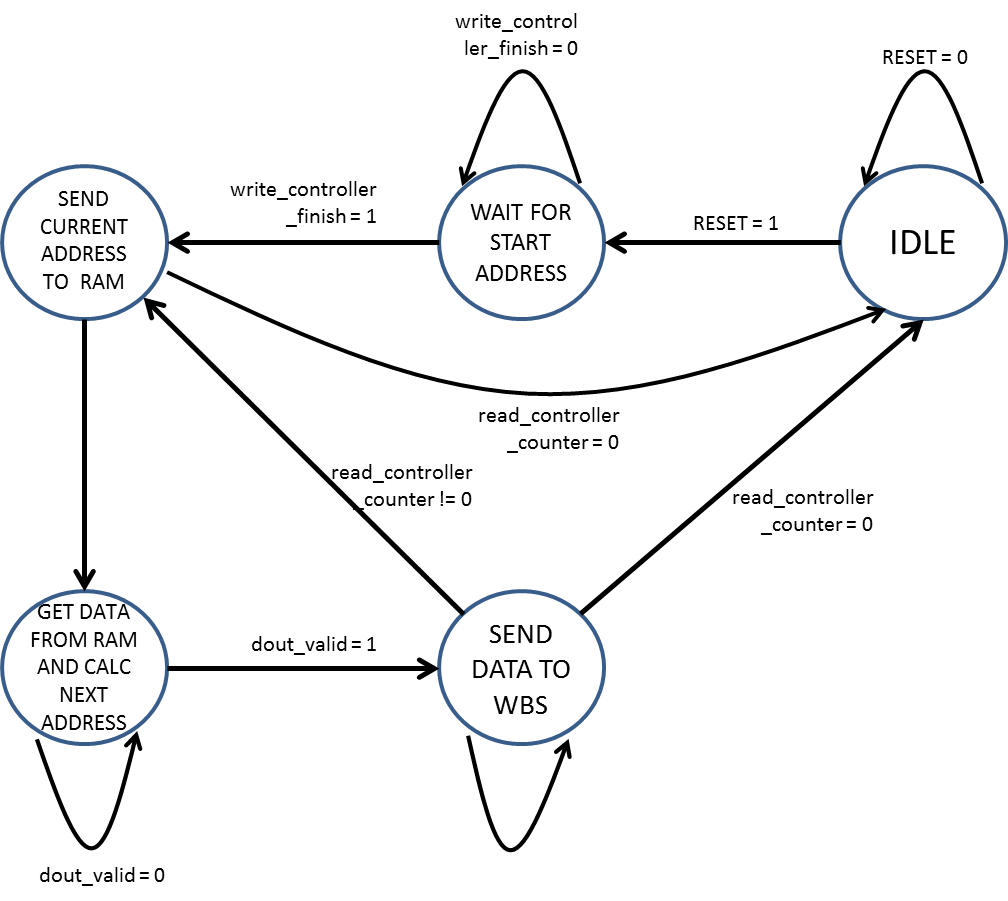
6Table 6- read controller generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| start\_addr\_in | In | record\_depth\_g | The start address of the data that we need to send out to the user |
| write\_controller\_finish | In | 1 | '1' ->WC has finish working and saving all the relevant data |
| dout\_valid | In | 1 | Data coming from RAM validity |
| data\_from\_ram | In | num\_of\_signals\_g | Data coming from RAM |
| data\_in | In | num\_of\_signals\_g | Data in from signal generator |
| read\_controller\_finish | Out | 1 | 1 -> RC finish sending all the data out. 0 -> other |
| addr\_out | Out | record\_depth\_g | Address sent to the RAM in order to extract it out back to the user |
| aout\_valid | Out | 1 | Validity of address that sent to the RAM |
| data\_out\_to\_WBM | Out | num\_of\_signals\_g | Data sent out back to the user |
| data\_out\_to\_WBM\_valid | Out | Add\_width\_g | Validity of the outputting data |

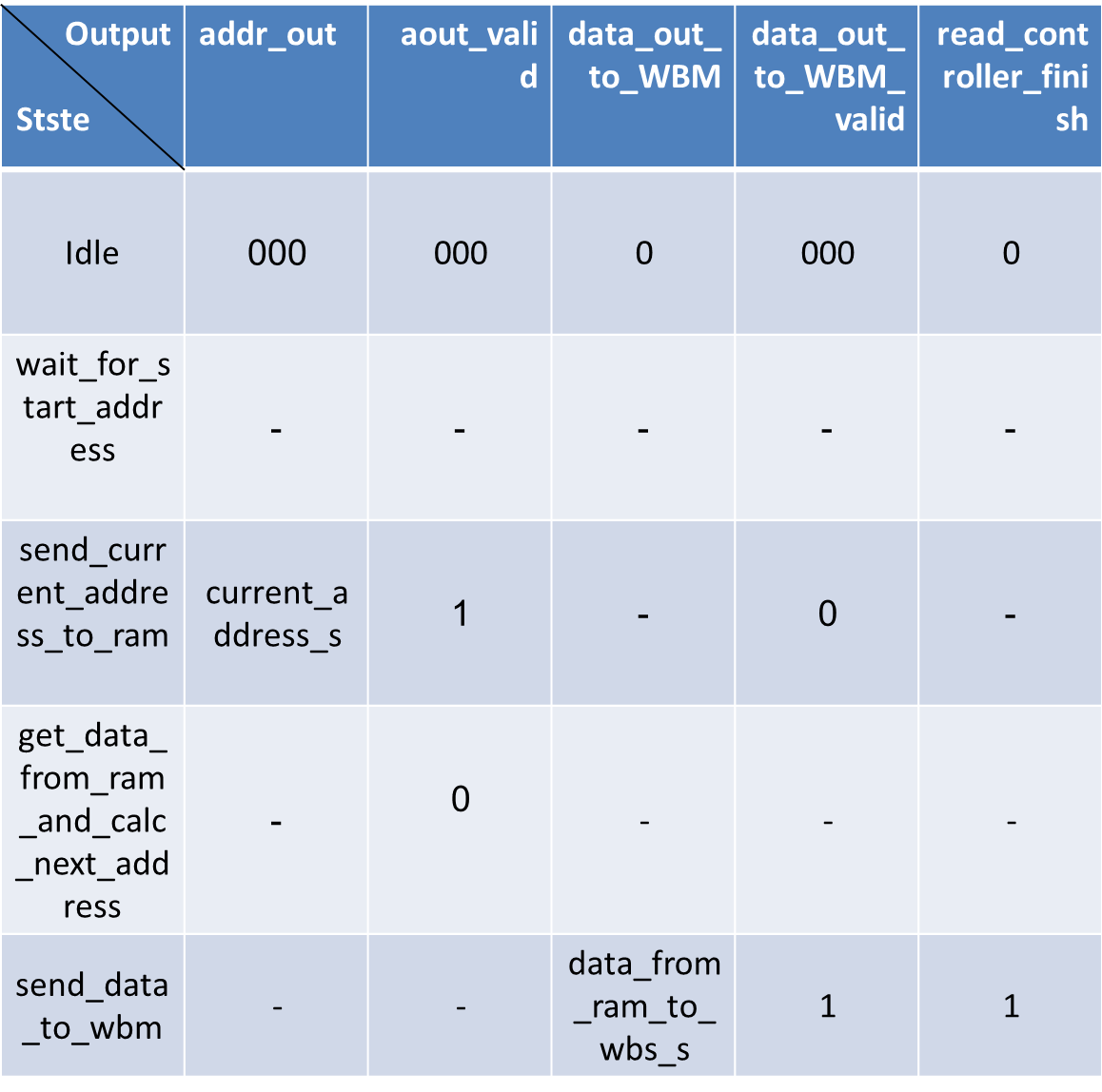
7Table 7- read controller signals

Read controller FSM



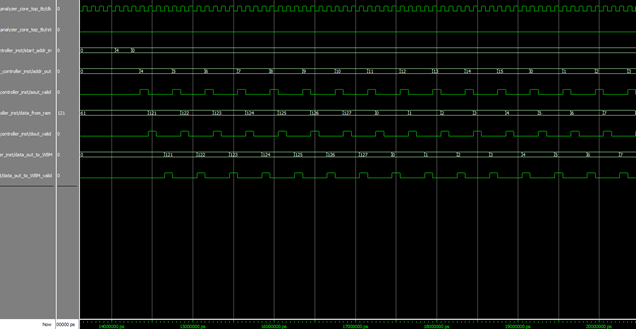
14Figure 14-read controller FSM

Output table



8Table 8- read controller output

### 3.3.2SIMULATION



15Figure 15-read controller simulation

Start address is received. The next address is calculated and sent to the RAM. Data and validity is received from RAM. Output data is being sent to the coordinator.

## 3.4 IN OUT COORDINATOR:

### 3.4.1 Description:

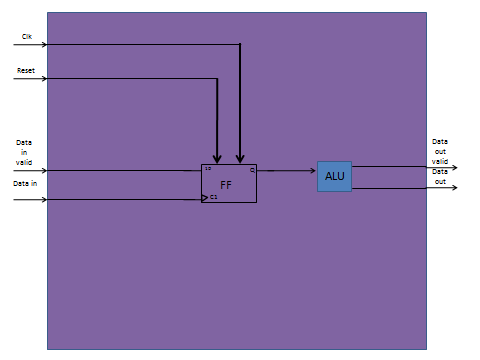
In most cases the width of the recorded signals, which is in fact the number of the recorded signals, and the width of the bus in the system (data\_width\_g( are different.

This entity coordinates between the input data (num\_of\_signals\_g) and the output data (data\_width\_g)

If input < output use in\_small\_out\_cordinator, (adding zeroes in the MSB)

if input = output use in\_equal\_out\_cordinator, (stay as is)

else (input > output) use in\_big\_out\_cordinator. (break every input to a few output cycles)



16Figure 16-in out coordinator

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| out\_width\_g | 3 | Width of outputting data |
| in\_width\_g | 8 | Width of incoming data |

9Table 9- in out coordinator generics

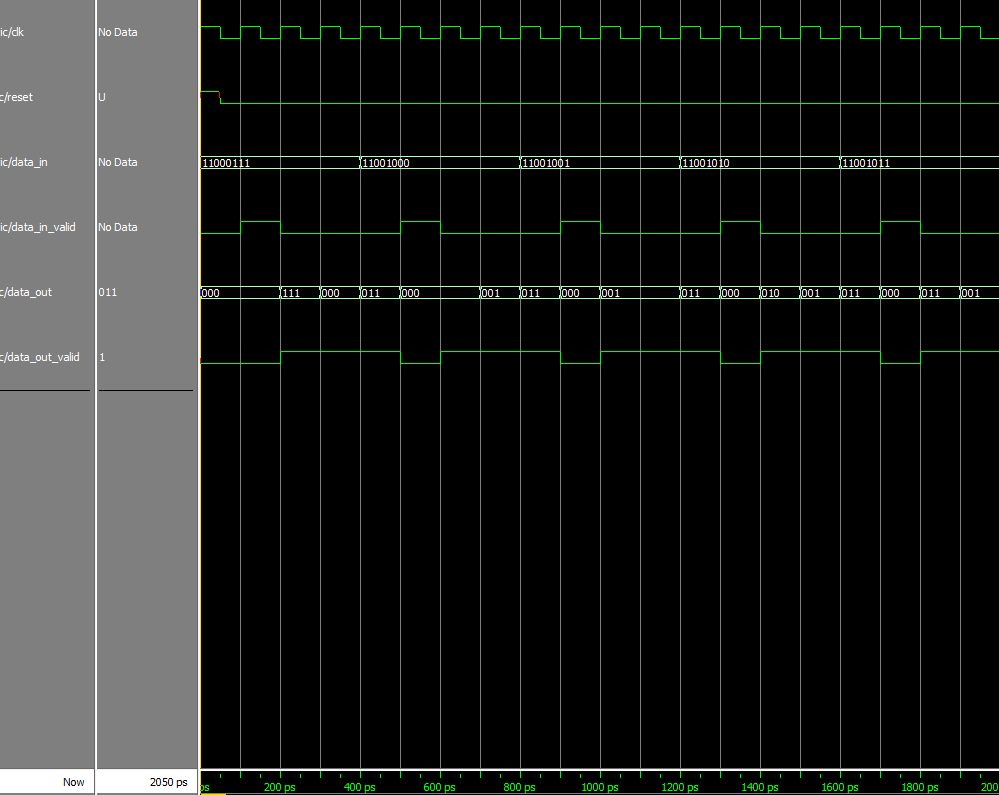
Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| data\_in | In | in\_width\_g | Incoming data |
| data\_in\_valid | In | 1 | Incoming data valid |
| data\_out | In | out\_width\_g | Outputting data |
| data\_out\_valid | In | 1 | Outputting data valid |

10Table 10- in out coordinator signals

### 

### 3.4.2 Simulation



17Figure 17-in out coordinator simulation

In that example, the input width is 8 and the output width is 3.

Each time that data\_in\_valid signal rise, the entity sample the input and start to "break" it to output signals in the width of 3. Of course that one "output cycle" take us 3 clock cycle and for that reason we can see that data\_out\_valid signal is high 3 clock cycles each time. The sampled input data is divided in order to keep the value of the original input, meaning that we start outputting the number from the less significant bit (LSB) to the most significant bit (MSB), and if needed (like in that case) we add 0 to the MSB of the last output data.

In our example we divide the input 11000111 to the outputs of: first- 111, second- 000 third- 011. We can see that one 0 was added to the last output in order to fit him to the output width.

### 3.5 INPUT>OUTPUT COORDINATOR

### 3.5.1 Description:

Coordinate between input data (num\_of\_signals\_g) to output data (data\_width\_g) when input > output. We save at first the incoming data (when it is valid) and start to output it in the width of out\_width\_g every clock cycle. In the case that the last output is smaller than that width, we add 0 to the MSB until it's fit, in order to not change the number value.

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| out\_width\_g | 3 | Width of outputting data |
| in\_width\_g | 8 | Width of incoming data |

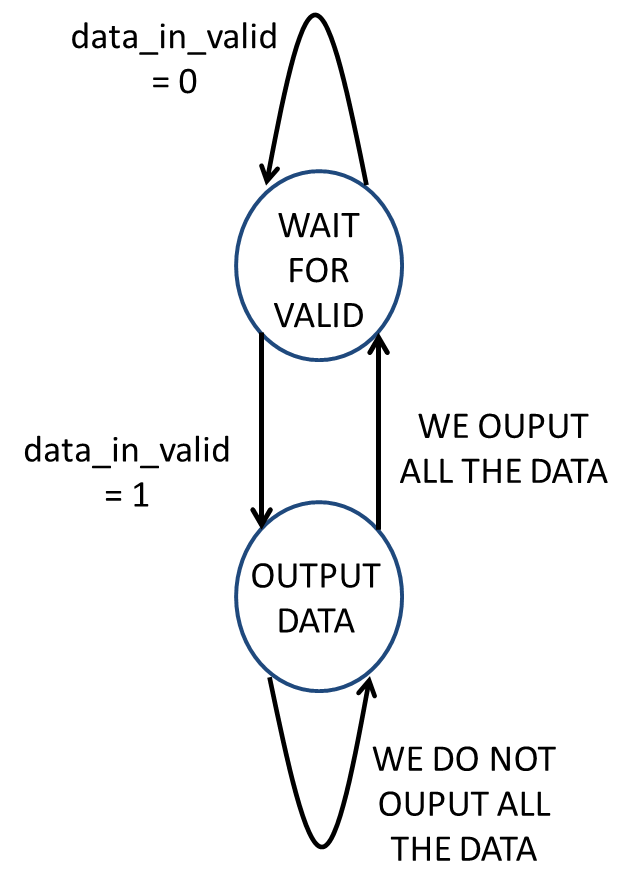
**11**Table 11- in> out coordinator generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| data\_in | In | in\_width\_g | Incoming data |
| data\_in\_valid | In | 1 | Incoming data valid |
| data\_out | In | out\_width\_g | Outputting data |
| data\_out\_valid | In | 1 | Outputting data valid |

12Table 12- in> out coordinator signals

## INPUT> OUTPUT CORDINATOR FSM



18Figure 18-in >out coordinator FSM

Output table



13Table 13- in> out coordinator output

### 3.6 INPUT<OUTPUT COORDINATOR

### 3.6.1 Description:

Coordinate between input data (num\_of\_signals\_g) to output data (data\_width\_g) when input < output. We save at first the incoming data (when it is valid) and in the next clock cycle we output it. We add 0 to the MSB until it's fit to out\_width\_g , in order to not change the number's value.

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| out\_width\_g | 3 | Width of outputting data |
| in\_width\_g | 8 | Width of incoming data |

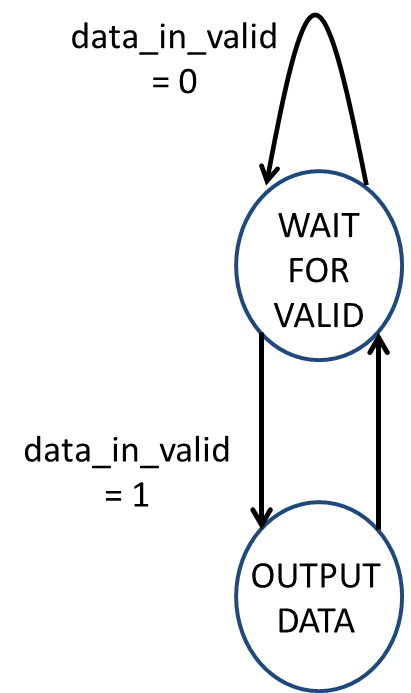
14Table 14- in< out coordinator generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| data\_in | In | in\_width\_g | Incoming data |
| data\_in\_valid | In | 1 | Incoming data valid |
| data\_out | In | out\_width\_g | Outputting data |
| data\_out\_valid | In | 1 | Outputting data valid |

15Table 15- in< out coordinator signals

INPUT< OUTPUT CORDINATOR FSM



19Figure 19-in <out coordinator FSM

Output table



16Table 16- in< out coordinator output

### 3.7 INPUT=OUTPUT

### 3.7.1 Description:

Coordinate between input data (num\_of\_signals\_g) to output data (data\_width\_g) when input > output. We save at first the incoming data (when it is valid) and start to output it in the width of out\_width\_g every clock cycle. In the case that the last output is smaller than that width, we add 0 to the MSB until it's fit, in order to not change the number value.

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| out\_width\_g | 3 | Width of outputting data |
| in\_width\_g | 8 | Width of incoming data |

17Table 17- in= out coordinator generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| data\_in | In | in\_width\_g | Incoming data |
| data\_in\_valid | In | 1 | Incoming data valid |
| data\_out | In | out\_width\_g | Outputting data |
| data\_out\_valid | In | 1 | Outputting data valid |

18Table 18- in= out coordinator signals

INPUT=OUTPUT CORDINATOR FSM

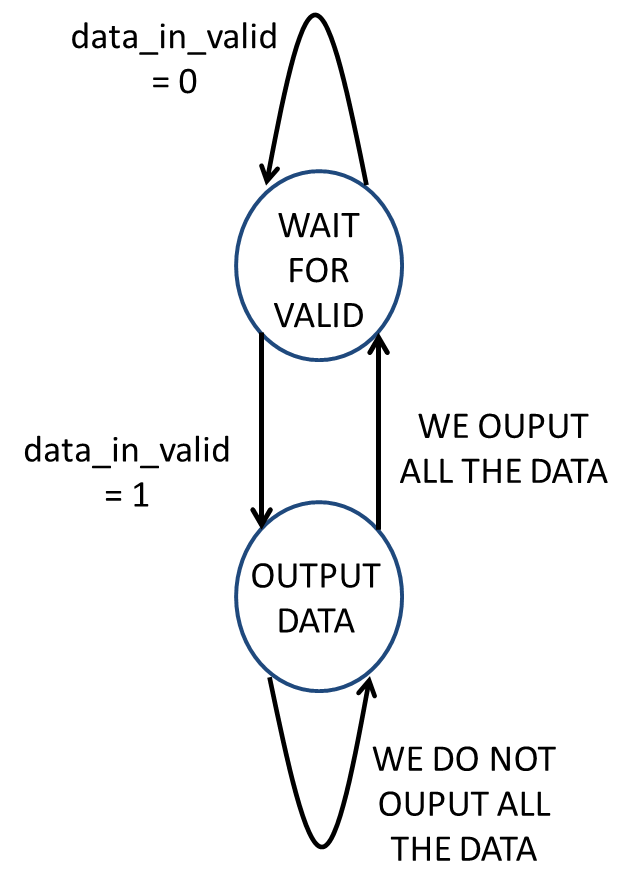


Figure 20-in =out coordinator FSM

Output table



19Table 19- in= out coordinator output

## 3.8 ENABLE FSM

### 3.8.1 Description:

State Machine for enabling write controller, determine when to enable the WC.

RC enable is through WC\_finish signal

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| enable\_polarity\_g | 1 | '1' the entity is active high, '0' entity isactive low |

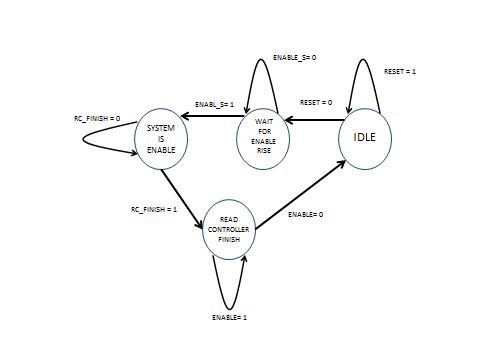
20Table 20 – enable generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| Enable | In | 1 | Enabling the entity. if (enable = enable\_polarity\_g) -> start working, else-> do nothing. Come from registers |
| wc\_finish | in | 1 | '1' ->WC has finish working and saving all the relevant data (RC will start work), '0' ->WC is still working |
| rc\_finish | in | 1 | '1' -> read controller finish working, '0' -> system still working |
| enable\_out | Out | 1 | Enable send to the write controller to start saving the data and searching trigger rise |

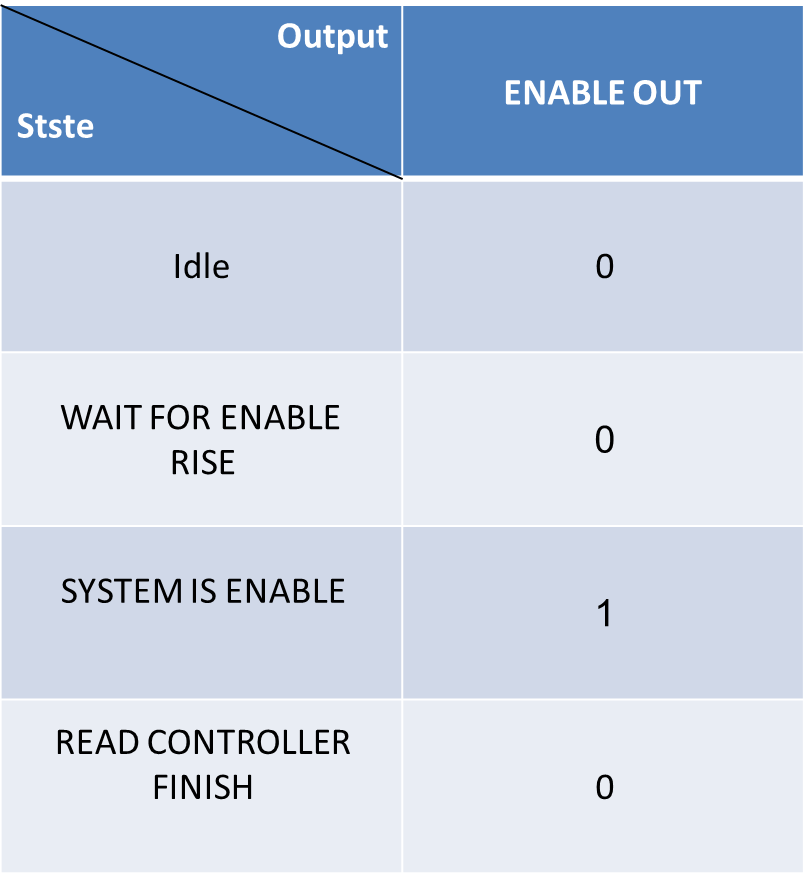
21Table 21 – enable signals

Enable FSM



21Figure 21-enable FSM

Output table



22Table 22– enable output

## 3.9 MEMORY UNIT

### 3.9.1 Description:

The memory unit is implemented as an array of RAMs, when the depth and width of each one of them is known. We used them to implement the memory unit by the width and depth we need.

In this project, we took a simple RAM with fix width and length, and replicate it in the form of an array, to create a generic RAM, which holds the generics width and length that is defined in the system.

clk

Reset

RAM

RAM

addr\_in data\_out

addr\_out

aout\_valid

data\_in

din\_valid

RAM

dout\_valid

22figure 22-memory unit

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| width\_in\_g | 8 | Width of data |
| addr\_bits\_g | 4 | Depth of data (2^4 = 16 addresses) |
| power2\_out\_g | 0 | Output width is multiplied by this power factor (2^1). In case of 2: output will be (2^2\*8=) 32 bits wide |
| power\_sign\_g | 1 | '-1' => output width > input width ; '1' => input width > output width |

23Table 23 – memory unit generics

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| addr\_in | In | addr\_bits\_g | Input address |
| addr\_out | In | addr\_bits\_g | Output address |
| aout\_valid | In | 1 | Output address is valid |
| data\_in | In | width\_in\_g | Input data |
| din\_valid | In | 1 | Input data valid |
| data\_out | Out | width\_in\_g | Output data |
| dout\_valid | Out | 1 | Output data valid |

24Table 23 – memory unit signals

Number of RAM's determine be the formula:



For example: signal\_ram\_depth\_g = 4, signal\_ram\_width\_g = 2, recorded\_depth\_g = 12, num\_of\_signals = 6.

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

RAM SIMPLE

GENERIC RAM

Internal logic analyzer core top

Generic table

|  |  |  |
| --- | --- | --- |
| Name | Width | Description |
| reset\_polarity\_g | 1 | '1' reset active high, '0' active low |
| enable\_polarity\_g | 1 | '1' the entity is active, '0' entity not active |
| signal\_ram\_depth\_g | 3 | depth of basic RAM |
| signal\_ram\_width\_g | 8 | width of basic RAM |
| record\_depth\_g | 3 | number of bits that are recorded from each signal is 2^ record\_depth\_g |
| data\_width\_g | 8 | defines the width of the data lines of the system |
| Add\_width\_g | 8 | width of address word in the system |
| num\_of\_signals\_g | 8 | number of signals that will be recorded simultaneously (Width of data) |
| en\_reg\_address\_g | 0 | Address of the register that enable reading from other registers |
| trigger\_type\_reg\_1\_address\_g | 1 | Address of the position register |
| trigger\_position\_reg\_2\_address\_g | 2 | Address of the type register |
| clk\_to\_start\_reg\_3\_address\_g | 3 | Address of the clock to start register (not in use) |
| enable\_reg\_address\_4\_g | 4 | Address of the enable register |
| power2\_out\_g | 0 | Output width is multiplied by this power factor (2^1). In case of 2: output will be (2^2\*8=) 32 bits wide -> our output and input are at the same width |
| power\_sign\_g | 1 | '-1' => output width > input width ; '1' => input width > output width |
| type\_d\_g | 1 | Type Depth |
| len\_d\_g | 1 | Length Depth |
| output\_type\_id\_g | 2 | The TYPE of the component that the data is directed to. (our case- TX PATH) |

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| rst | In | 1 | System reset |
| data\_in | In | num\_of\_signals\_g | Input data from Signal Generator |
| trigger | In | 1 | trigger signal from Signal Generator |
| ADR\_I | In | Add\_width\_g | contains the address word |
| DAT\_I | In | data\_width\_g | contains the data\_in word |
| WE\_I | In | 1 | '1' for write, '0' for read |
| STB\_I | In | 1 | '1' for active bus operation, '0' for no bus operation |
| CYC\_I | In | 1 | '1' for bus transition request, '0' for no bus transition request |
| TGA\_I | In | data\_width\_g\*type\_d\_g | contains the type word |
| TGD\_I | In | data\_width\_g\*len\_d\_g | contains the length word |
| ACK\_O | Out | 1 | '1' when valid data is transmitted to MW or for successful write operation |
| WS\_DAT\_O | Out | data\_width\_g | data transmit to MW |
| STALL\_O | Out | 1 | STALL - WS is not available for transaction |
| wm\_end\_out | Out | 1 | when '1' WM ended a transaction or reset by watchdog ERR\_I signal |
| ADR\_O | Out | Add\_width\_g | contains the address word |
| WM\_DAT\_O | Out | data\_width\_g | contains the data\_in word |
| WE\_O | Out | 1 | '1' for write, '0' for read |
| STB\_O | Out | 1 | '1' for active bus operation, '0' for no bus operation |
| CYC\_O | Out | 1 | '1' for bus transition request, '0' for no bus transition request |
| TGA\_O | Out | data\_width\_g\*type\_d\_g | contains the type word |
| TGD\_O | Out | len\_d\_g \* data\_width\_g | contains the length word |
| ACK\_I | In | 1 | '1' when valid data is received from WS or for successful write operation in WS |
| DAT\_I\_WM | In | data\_width\_g | data received from WS |
| STALL\_I | In | 1 | STALL - WS is not available for transaction |
| ERR\_I | In | 1 | Watchdog interrupts, resets wishbone master |

# 4 SIMULATIONS

At first we made a manual simulation to each entity to check the functionality, afterwards, we built a core test bunch in order to check the entire core. Each diagram was checked and confirmed for the correct result and if necessary, code changes were made and the simulation were made again.

Simulation table:

We conducted about 17 simulations:

-



25Table 25- simulations

## 4.1 Description:

1. We enter the user's configurations through the WBS and save it in the registers, after that we wait for enable rise and the system start to save the data since that. When we detect trigger rise we continue to save the relevant data and after that send it out through the WBM.
2. We change the recording depth to 4 (2^4 = 16), position to 75% (meaning that 75% -> 12 bits, will recorded before the trigger and the other after that, we also change the trigger type to zeroes (number 3 ) so trigger will rise after three low sampling.
3. Number of signals is changed to 5, meaning our input data is between 0-32 in decimal (2^5), at first the trigger position is 100 and all the data is recorded before the trigger, and second time the position is 0 and all the data is recorded after the trigger.
4. In this simulation we check two trigger positions that are not complete, 15% and 85%.

We also change the trigger type, from 'zeroes' (3 low in a row) in the first one to rise in the second.

In the end we rise the RESET signal to show that the system responds to that signal.

1. We checked the RESET signal. At first trigger position is 15%, trigger type is low. While data is exiting we rise RESET signal and change trigger position to 85% and trigger type to rise. We now make sure that the system recognize trigger rise again.
2. This simulation checks for different types of trigger type and position. First check: type is rise, position is 0. Second check: type is fall, position is 10%. Third check: type is high, position is 30%.
3. Continue checking different types of trigger type and position. First check: type is low, position is 50%. Second check: type is rise, position is 70%. Third check: type is fall, position is 90%.
4. In this simulation we focus on changing configuration after trigger rise. We also check that trigger rise after we have already found trigger rise or after the data has already been extracted but there wasn’t another configure, will not affect the system.

At first we commit certain configuration, and after trigger rise and extracting the data, we can see that a second trigger rise doesn't cause data extraction until new configure is being committed.

In advance, we record 32 samples and make sure that the system react well to this value( record\_depth\_g=5).

1. Check of configuration position. In this simulation we can see that system configuration can occur even before the data was entirely extracted back to the user. But the second stream and trigger identification can only happen after the first stream has end( after READ CONTROLLER FINISH signal rise).

In this simulation we configure the system to first state and while the data is still extracting we configure to a different state. After all the data was extracted we only need to enable the register ENABLE and the system starts to work with the second configuration.

1. Another check of configuration's position. We change trigger type to all possible types, but configure the system only afterwards and check whether trigger rises when the system wasn’t configure.
2. We check the change in ENABLE signal that is now active low. At first we change the trigger signal to see that it doesn't rise before the system is configured. After enabling the system( ENABLE register is low) the system starts to save the data and look for trigger rise.
3. We check the change in RESET signal that is now active low. At first we configure the system, rise trigger and after few samples we enable RESET signal and configure the system to different values. We check the system reaction to RESET signal changes(active low instead of high), and how the system recovers after lowering RESET signal.
4. From this simulation on we replace the generic of address width Add\_width\_g by the generic record\_depth\_g.

We check the situation when single RAM width is wider than the width of the saved signals, i.e. signal\_ram\_width\_g> num\_of\_signals\_g.

1. In this simulation we compare the number of signals entering the BUS width' i.e.

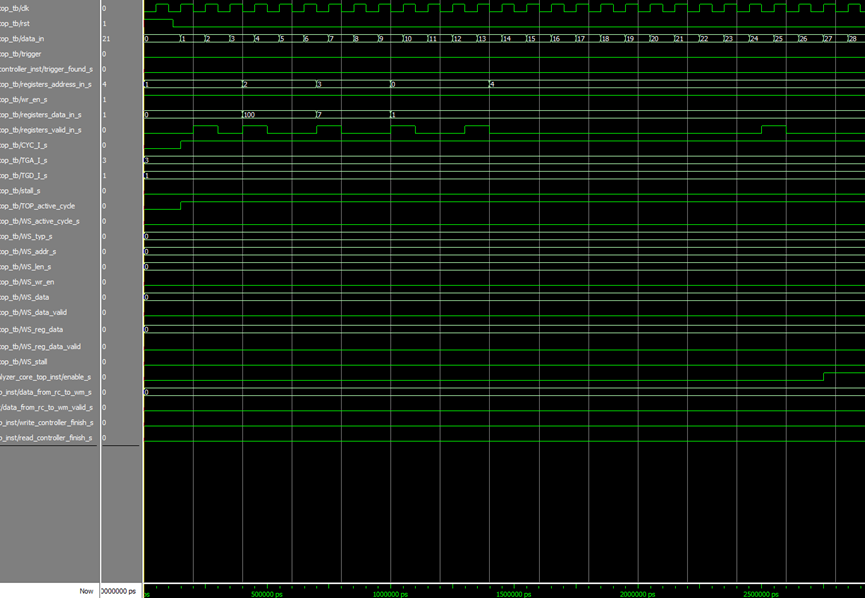
num\_of\_signals\_g = data\_width\_g.

1. In this simulation the BUS width is larger than the number of sampled signals, i.e num\_of\_signals\_g > data\_width\_g.
2. In this simulation BUS width is smaller than the number of sampled signals, i.e. num\_of\_signals\_g < data\_width\_g. As a result we can't extract the data in a single clock cycle (data width is 8), therefor we need two clock cycles.
3. In this simulation BUS width is still smaller than the number of sampled signals, but now we have enlarged the saved data width.

* For more information: <http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/> [סימולציות internal\_logic\_ananlyzer\_core\_top.docx](http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/core/%d7%a1%d7%99%d7%9e%d7%95%d7%9c%d7%a6%d7%99%d7%95%d7%aa%20internal_logic_ananlyzer_core_top.docx)

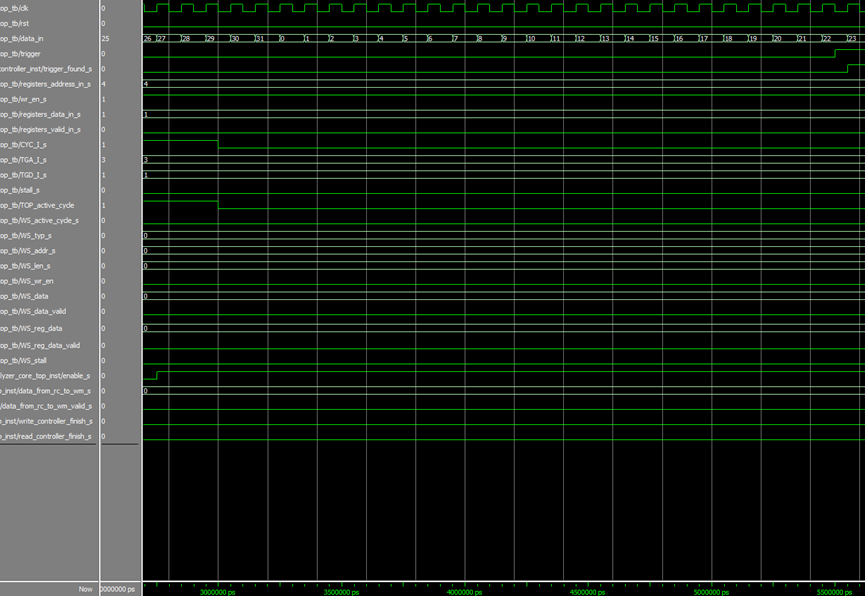
## 4.2 EXAMPLE:

Data is insert to the registers, in order to configure the user trigger position and type, enable signal is written to the register to enable the system.



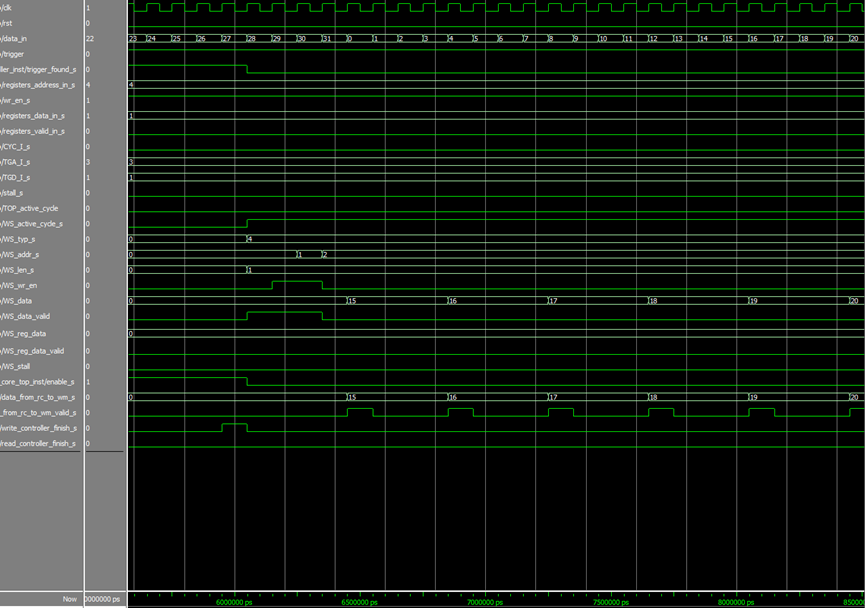
23Figure 23-simulation(1)

Data is being saved in the RAM until trigger rise, since position is 100, we do not save data after trigger rise.



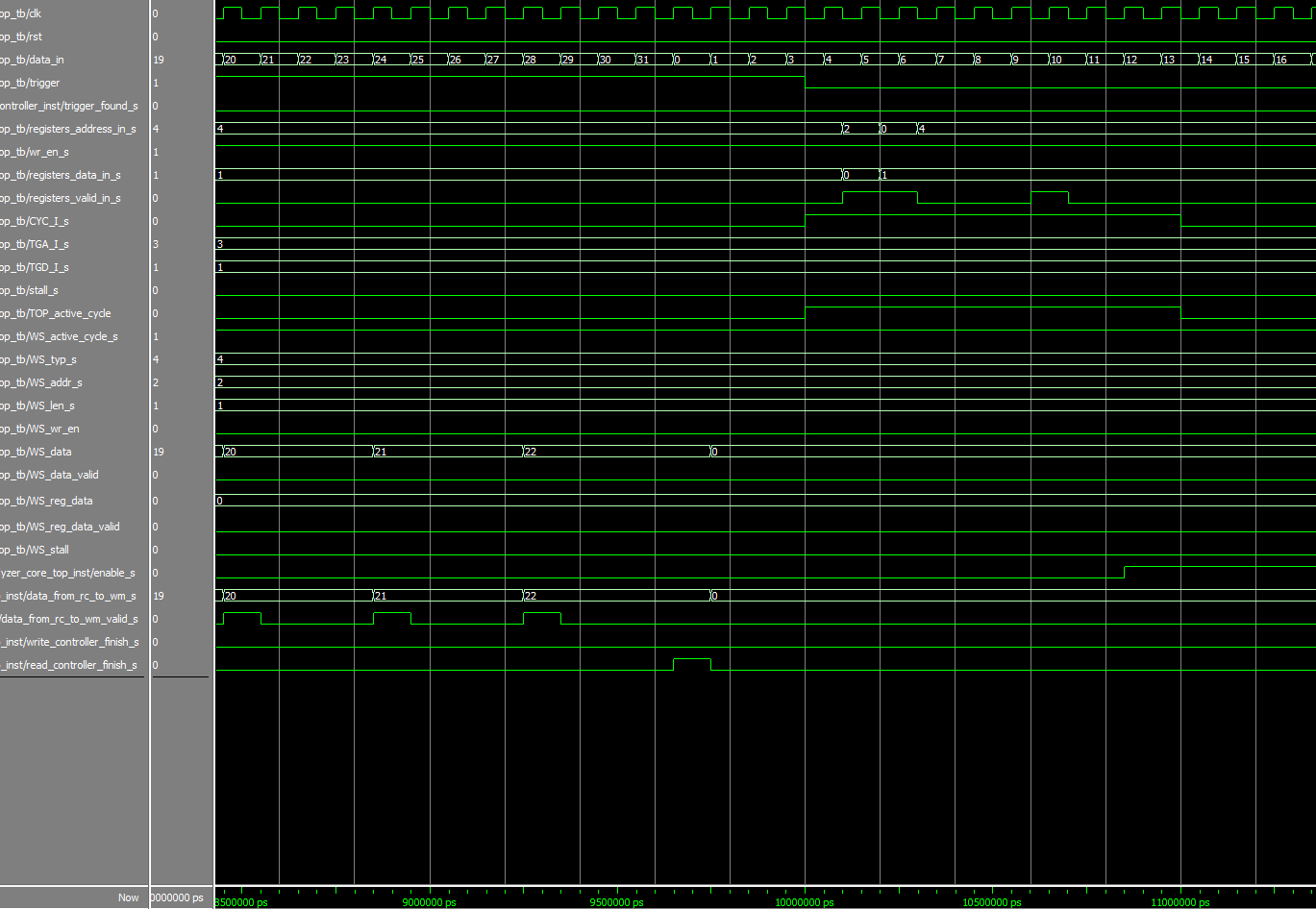
24Figure 24-simulation (2)

Write controller is finish, Read controller starting to send the relevant data out.



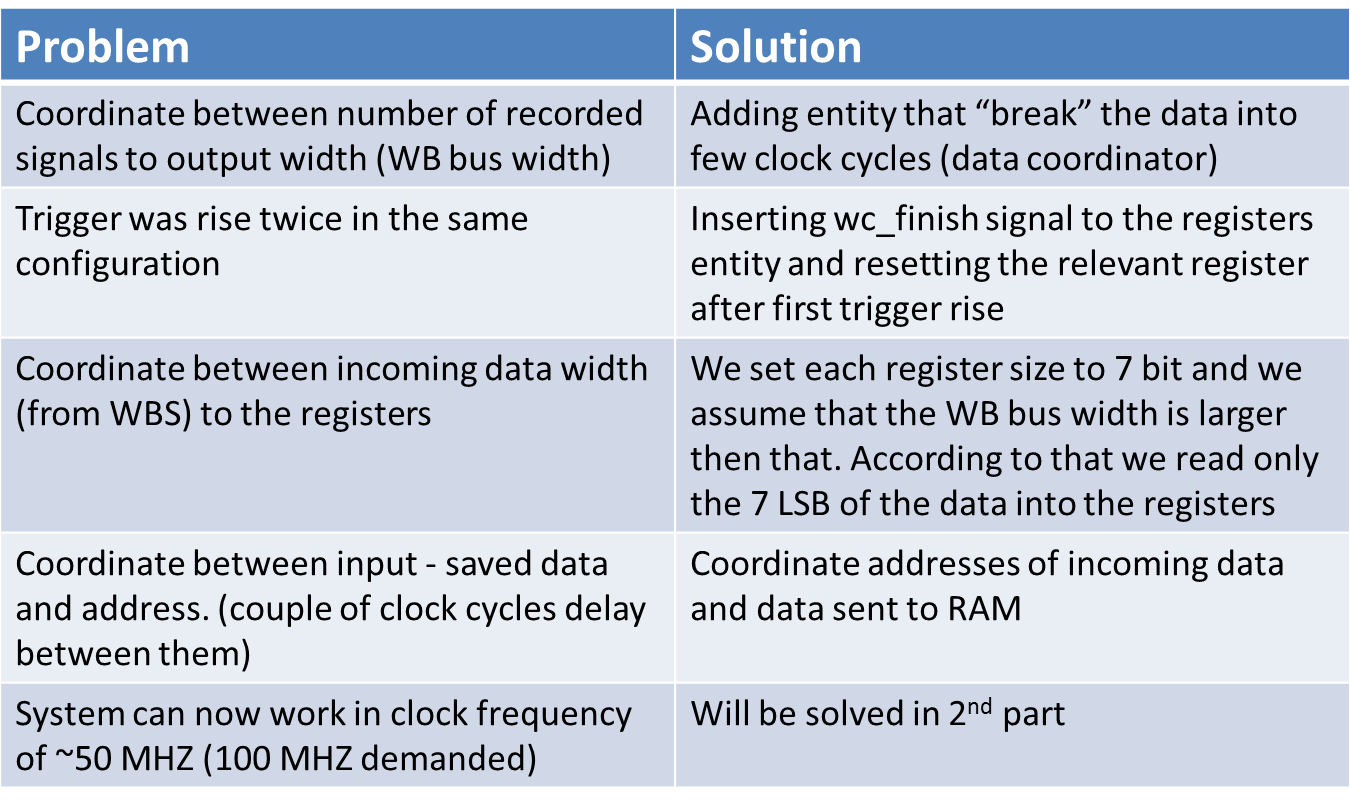
25Figure 25-simulation (3)

After that all the relevant data has being sent out, read controller finish working. We can now configure a new and different simulation



26Figure 26-simulation (4)

# 5 PROBLEMS AND SOLUTIONS



## EXMPLES

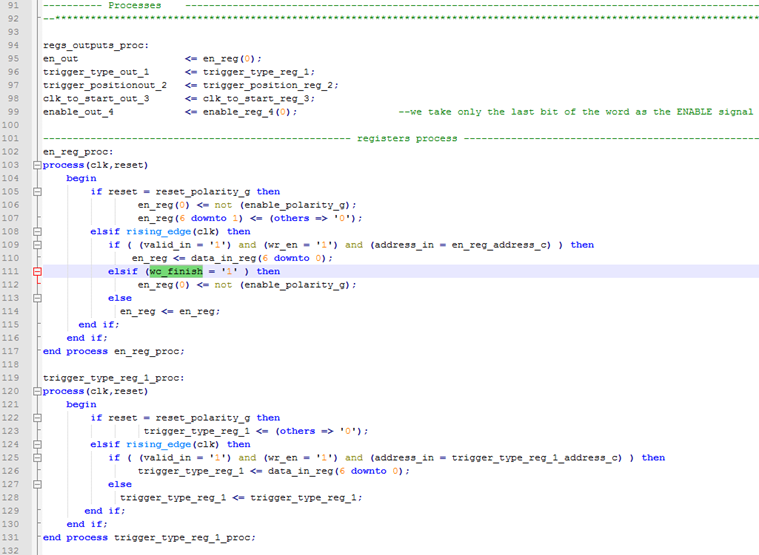
# First example: After first trigger rise, the system identify another trigger rise although the data was still recorded

# 

27Figure 27-first problem simulation

Problem- there was no dependency between two trigger rises

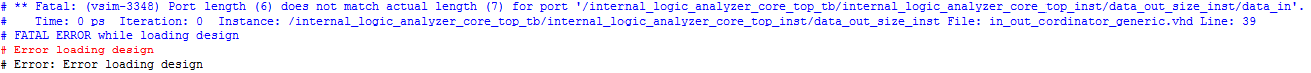
Our solution- adding wc\_finish signal to the registers and resetting the enable register



28Figure 28-code solution

Second example: Input width is num\_of\_signals\_g, output width is data\_width\_g.

Problem- the two widths don’t match.



29Figure 29-second problem

Our solution- adding an entity that coordinates between them.

# 

30Figure 30-second solution

# 6 WORKING Methods

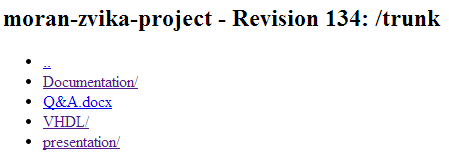
## 

## 

31Figure 31-SVN

## 6.1 SVN

The SVN proved to be a useful tool in the documentation process. The tool helps synchronizing the project designers and supervisor, document the changes with the project progress and backup each version. An SVN snapshot example can be shown below:



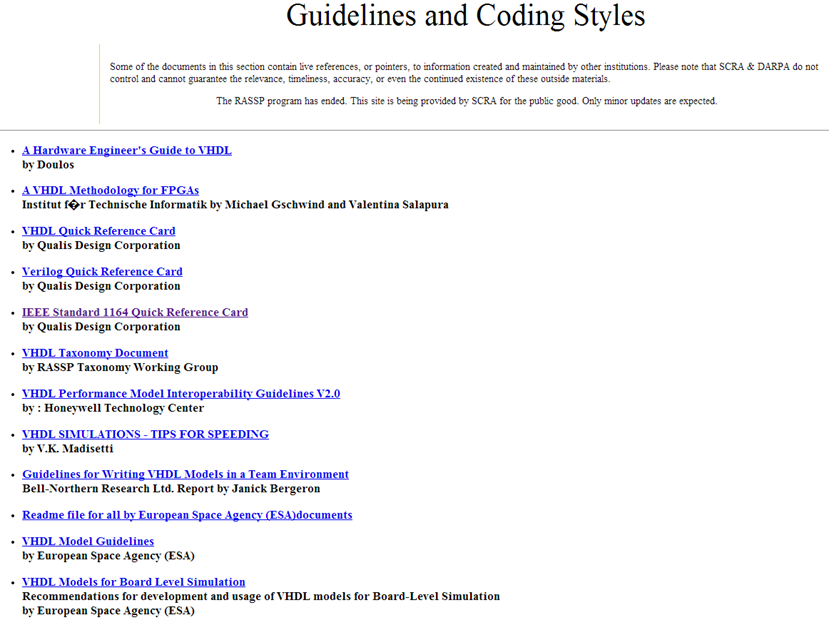
## 

32Figure 32-SVN snapshot

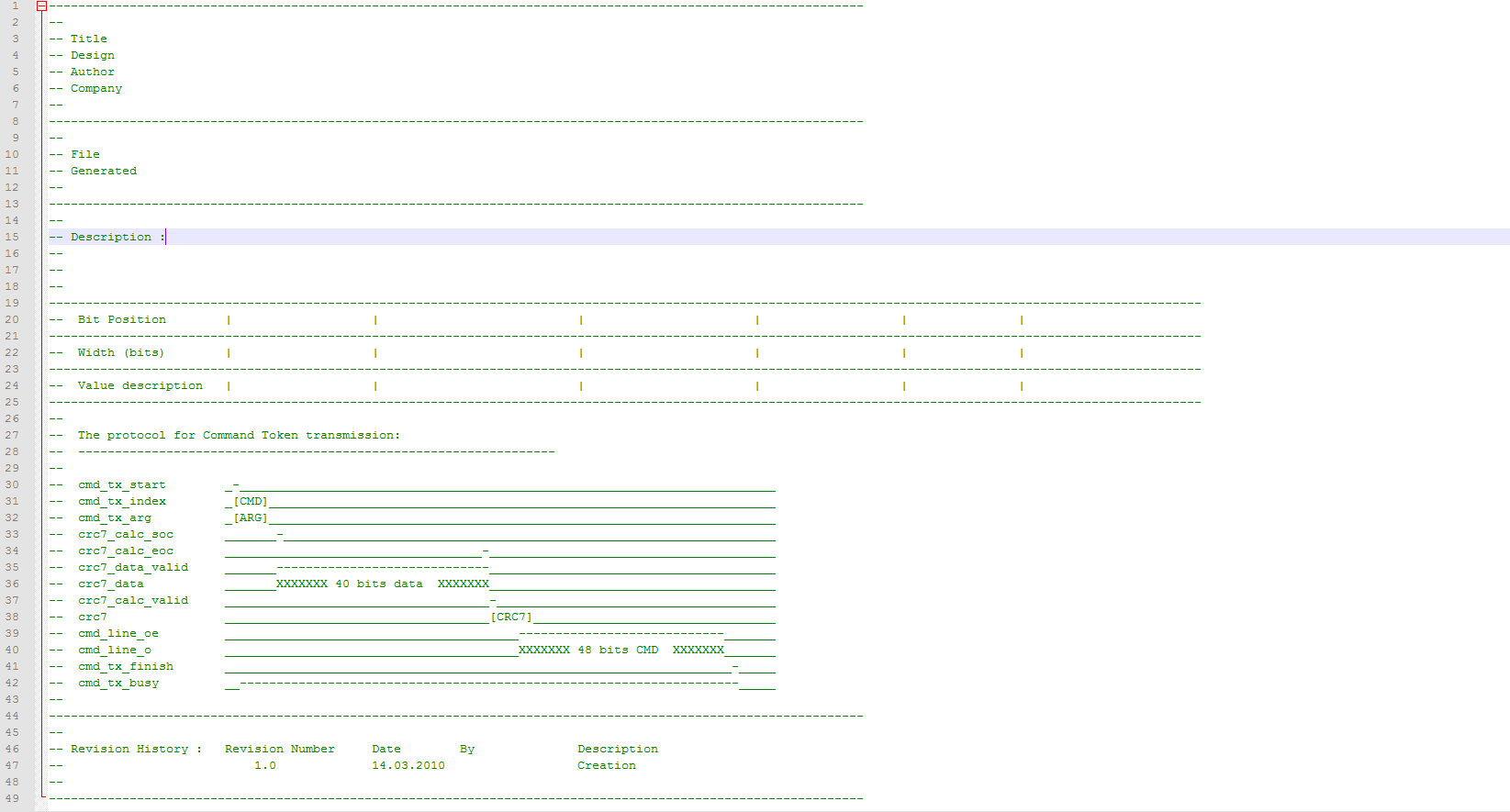
## 6.2 CODING GUIDELINES

The project was written according to strict coding guidelines which include code design according known conventions, usage of entity template.

For more information: <http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/vhdl&modelsim%20guid/>



33Figure 33-coding guidelines



34Figure 34-entity template

## 6.3 CODE REVIEW

1. Visual/ Compiler

2.Local simulation to the entity- each block was tested separately, manually first then using test bunch.

3. Top simulation- after all blocks were combined, many simulations were made to check each situation.

## 6.4 DOCUMENTATION

Detailed documents of each block, and of the entire core, including simulations can be found in the link: <http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/core/>

Summary

Project Usage

הפרויקט נועד להוות כלי לדיבוג יעיל, נוח וקל לכרטיס FPGA, ללא תלות ביצרן הכרטיס ובסוגו. מכאן שהפרויקט יכול להיות בשימוש כל אדם המשתמש בכרטיס FPGA, לכל מטרה.

כרטיסים אלו נמצאים בין השאר בשימוש מתכנני מעגלים חשמליים, מתכנני צ'יפים למיניהם ואף במעבדה בפקולטה להנדסת חשמל בטכניון, ולכן כל אחד מאלו הוא משתמש פוטנציאלי לפרויקט זה.

Problems during the project

Project status

הפרויקט כעת נמצא בשלב הסינתזה הראשונה, לאחר האינטגרציה בין כל חלקי המערכת ושלב הסימולציות.

השלבים הבאים הם צריבה על כרטיס ה FPGA, יצירת הGUI, תיקון באגים וביצוע סימולציות במעבדה.

Conclusions

בפרויקט זה יצרנו מערכת מורכבת המכילה רכיבים רבים ושונים, ונתקלנו בבעיות אשר הצריכו לעיתים חשיבה יצירתית והבנה עמוקה. כמו כן העמקנו את שליטתנו בשפת התכנות VHDL המצויה בחלקים נרחבים בתעשיית תכנון הרכיבים החשמליים.

הפרויקט תרם לנו רבות כיוון שהוא כולל אינטגרציה בין רכיבים רבים ושונים, ובזכותו נחשפנו לפרוטוקול הWB, לתהליך התכנון והדיבוג של מערכת מורכבת, לעבודה בסביבת MODELSIM, לנושא חבילות המידע והמעבר בין הרכיבים האנלוגים והדיגיטלים שמבוצע על ידי רכיבי הUART ועוד מגוון נושאים.

עם זאת, הפרויקט לקח זמן רב ולבסוף לא סיימנו את כל שלבי התהליך כפי שציפינו שיקרה, כך שמספר נושאים חשובים הושמטו, כמו צריבת כרטיס הFPGA, בניית הGUI ועוד.

לסיכום אנו מרגישים שהפרויקט תרם לנו רבות במספר די רחב של נושאים אשר משמשים את התעשיה, אך עם זאת יש לנו תחושת פיספוס קלה על כמה נושאים שלא הספקנו לגעת בהם.

# 7 ABBREVIATIONS

WC – Write Controller

RC – Read Controller

CLK – Clock

RST – Reset

WM – Wishbone Master

WS – Wishbone Slave

UART – Universal Asynchronous Receiver Transmitter

RAM – Random Access Memory

FSM - final state machine