****

Technion

*Electrical Engineering Department*

High Speed Digital System Lab

Project Documentation

**Internal Logic Analyzer Core**

Part B

Date of last update: 26.1.2014Students: Moran Katz, Zvika Pery

Mentor: Moshe Porian

TABLE OF CONTENT

[1 INTRODUCTION 7](#_Toc378518862)

[1.1 Abstract 7](#_Toc378518863)

[1.2 Project Goal 7](#_Toc378518864)

[1.3 Project Requirements 7](#_Toc378518865)

[1.4 Engineering Tools 8](#_Toc378518866)

[1.5 Applicable Documents 8](#_Toc378518867)

[2 GENERAL DESCRIPTION 9](#_Toc378518868)

[2.1 Data Flow 10](#_Toc378518869)

[3 MICRO ARCHITECTURE 16](#_Toc378518871)

[3.1 Registers Unit 16](#_Toc378518872)

[3.1.1 Simulation 18](#_Toc378518873)

[3.2 Write Controller 19](#_Toc378518874)

[3.2.1 Simulation 24](#_Toc378518875)

[3.3 Read Controller 25](#_Toc378518876)

[3.3.1 Simulation 28](#_Toc378518877)

[3.4 In Out Coordinator: 29](#_Toc378518878)

[3.4.1 Simulation 30](#_Toc378518879)

[3.4.3 Input>Output Coordinator 31](#_Toc378518880)

[Input> Output Coordinator FSM 32](#_Toc378518881)

[3.4.4 Input<Output Coordinator 33](#_Toc378518882)

[3.4.5 Input=Output 35](#_Toc378518883)

[3.5 Enable FSM 37](#_Toc378518884)

[3.6 Memory Unit 39](#_Toc378518885)

[3.7 Internal Logic Analyzer Core Top 42](#_Toc378518886)

[3.8 Signal Generator 45](#_Toc378518887)

[3.9 Reused Blocks Architecture 49](#_Toc378518888)

[3.9.1 UART Protocol 49](#_Toc378518889)

[3.9.2 Wishbone Protocol 61](#_Toc378518890)

[3.9.3 Wishbone Intercon 62](#_Toc378518891)

[3.9.4 Wishbone Master 63](#_Toc378518892)

[3.9.5 Wishbone Slave 66](#_Toc378518893)

[3.9.6 OUTPUT BLOCK (compressed file memory) 68](#_Toc378518894)

[4 DATA TRANSFER 74](#_Toc378518895)

[4.1 *Write transaction* 74](#_Toc378518896)

[Registers Write Message 74](#_Toc378518897)

[4.2 Read transaction 75](#_Toc378518898)

[Registers Read Request 75](#_Toc378518899)

[Registers Read Reply 75](#_Toc378518900)

[5 SIMULATIONS 77](#_Toc378518901)

[5.1 part a simulations 77](#_Toc378518902)

[5.1.1 Example: 79](#_Toc378518903)

[5.2 Part b simulations 81](#_Toc378518904)

[6 Synthesis 85](#_Toc378518905)

[7 PROBLEMS AND SOLUTIONS 87](#_Toc378518906)

[7.1 Examples 88](#_Toc378518907)

[First example: After first trigger rise, the system identify another trigger rise although the data was still recorded . 88](#_Toc378518908)

[8 WORKING METHODS 92](#_Toc378518911)

[8.1 SVN 92](#_Toc378518913)

[8.2 Coding Guidelines 92](#_Toc378518914)

[8.3 Code Review 94](#_Toc378518915)

[8.4 Documentation 94](#_Toc378518916)

[9 summary 94](#_Toc378518917)

[9.1 Project Usage 94](#_Toc378518918)

[9.2 Conclusions 94](#_Toc378518919)

[10 ABBREVIATIONS 94](#_Toc378518920)

[11 Appendix 95](#_Toc378518921)

TABLE OF FIGURES

[Figure 1- The final system 9](#_Toc378518922)

[Figure 2-Top data flow(1) 10](#_Toc378518923)

[Figure 3-Top data flow(2) 11](#_Toc378518924)

[Figure 4-Top data flow(3) 11](#_Toc378518925)

[Figure 5- Top data flow(4) 12](#_Toc378518926)

[Figure 6-Core data flow (5) 12](#_Toc378518927)

[Figure 7- Core data flow (6) 13](#_Toc378518928)

[Figure 8- Core data flow (7) 13](#_Toc378518929)

[Figure 9- Core data flow (8) 14](#_Toc378518930)

[Figure 10- Core data flow (9) 14](#_Toc378518931)

[Figure 11- Core data flow (10) 15](#_Toc378518932)

[Figure 12- Core data flow (11) 15](#_Toc378518933)

[Figure 13- Registers unit 16](#_Toc378518934)

[Figure 14- Registers unit simulation 19](#_Toc378518935)

[Figure 15- Write controller 20](#_Toc378518936)

[Figure 16- Write controller FSM 22](#_Toc378518937)

[Figure 17- Write controller simulation 24](#_Toc378518938)

[Figure 18- Read controller 25](#_Toc378518939)

[Figure 19- Read controller FSM 27](#_Toc378518940)

[Figure 20- Read controller simulation 28](#_Toc378518941)

[Figure 21- In out coordinator 29](#_Toc378518942)

[Figure 22- In out coordinator simulation 31](#_Toc378518943)

[Figure 23- In >out coordinator FSM 32](#_Toc378518944)

[Figure 24- In <out coordinator FSM 34](#_Toc378518945)

[Figure 25- In =out coordinator FSM 36](#_Toc378518946)

[Figure 26- Enable FSM 39](#_Toc378518947)

[Figure 27- memory unit 40](#_Toc378518948)

[Figure 28- Generic RAM example 41](#_Toc378518949)

[Figure 29- The core 42](#_Toc378518950)

[Figure 30- Signal generator 46](#_Toc378518951)

[Figure 31- Signal generator FSM 48](#_Toc378518952)

[Figure 32 - RX path 49](#_Toc378518953)

[Figure 33 - TX path 54](#_Toc378518954)

[Figure 34 - BUS to encoder FSM 57](#_Toc378518955)

[Figure 35 - Wishbone cycle waveform example 61](#_Toc378518956)

[Figure 36 - Wishbone Intercon 63](#_Toc378518957)

[Figure 37 - Connecting a Wishbone Master 64](#_Toc378518958)

[Figure 38 - Wishbone Master FSM 65](#_Toc378518959)

[Figure 39 - Connecting a Wishbone Slave 67](#_Toc378518960)

[Figure 40- Output block 68](#_Toc378518961)

[*Figure 41 - Output block illustration* 70](#_Toc378518962)

[Figure 42- Part a simulations (1) 79](#_Toc378518963)

[Figure 43- Part a simulations (2) 80](#_Toc378518964)

[Figure 44- Part a simulations (3) 80](#_Toc378518965)

[Figure 45- Part a simulations (4) 81](#_Toc378518966)

[Figure 46- Quartus Synthesis results 85](#_Toc378518967)

[Figure 47- Maximal frequency 86](#_Toc378518968)

[Figure 48- RTL scheme of the system 86](#_Toc378518969)

[Figure 49- Code solution 89](#_Toc378518970)

[Figure 50- Second problem 89](#_Toc378518971)

[Figure 51- Second solution 90](#_Toc378518972)

[Figure 52- Third problem 90](#_Toc378518973)

[Figure 53-Third problem solution 91](#_Toc378518974)

[Figure 54- SVN 92](#_Toc378518975)

[Figure 55- SVN snapshot 92](#_Toc378518976)

[Figure 56- Coding guidelines 93](#_Toc378518977)

[Figure 57- Entity template 93](#_Toc378518978)

TABLE OF TABLES

[Table 1 - Registers unit generics 17](#_Toc378518985)

[Table 2- Registers unit signals 18](#_Toc378518986)

[Table 3- Write controller generics 20](#_Toc378518987)

[Table 4- Write controller signals 21](#_Toc378518988)

[Table 5- Write controller output 23](#_Toc378518989)

[Table 6- Read controller generics 25](#_Toc378518990)

[Table 7- Read controller signals 26](#_Toc378518991)

[Table 8- Read controller output 28](#_Toc378518992)

[Table 9- In out coordinator generics 30](#_Toc378518993)

[Table 10- In out coordinator signals 30](#_Toc378518994)

[Table 11- In> out coordinator generics 32](#_Toc378518995)

[Table 12- In> out coordinator signals 32](#_Toc378518996)

[Table 13- In> out coordinator output 33](#_Toc378518997)

[Table 14- In< out coordinator generics 33](#_Toc378518998)

[Table 15- In< out coordinator signals 34](#_Toc378518999)

[Table 16- In< out coordinator output 35](#_Toc378519000)

[Table 17- In= out coordinator generics 35](#_Toc378519001)

[Table 18- In= out coordinator signals 36](#_Toc378519002)

[Table 19- In= out coordinator output 37](#_Toc378519003)

[Table 20- Enable generics 37](#_Toc378519004)

[Table 21- Enable signals 38](#_Toc378519005)

[Table 22- Enable output 39](#_Toc378519006)

[Table 23- memory unit generics 40](#_Toc378519007)

[Table 24- memory unit signals 41](#_Toc378519008)

[Table 25- Core generics table 43](#_Toc378519009)

[Table 26- Core signals table 45](#_Toc378519010)

[Table 27- Signal generator generic table 46](#_Toc378519011)

[Table 28- Signal generator signals table 47](#_Toc378519012)

[Table 29- Signal generator outputs 48](#_Toc378519013)

[Table 30 - RX path interface 50](#_Toc378519014)

[Table 31 - Uart rx interface 50](#_Toc378519015)

[Table 32 - Uart rx generics 51](#_Toc378519016)

[Table 33 - mp\_dec interface 51](#_Toc378519017)

[Table 34 - mp\_dec generics 52](#_Toc378519018)

[Table 35 - RAM interface 52](#_Toc378519019)

[Table 36- RAM generics 52](#_Toc378519020)

[Table 37 - CRC interface 53](#_Toc378519021)

[Table 38 - CRC generics 53](#_Toc378519022)

[Table 39 - Error Register interface 54](#_Toc378519023)

[Table 40 - Error register generics 54](#_Toc378519024)

[Table 41 - TX Path interface 55](#_Toc378519025)

[Table 42 - TX Path generics 56](#_Toc378519026)

[Table 43 - BUS to encoder FSM interface 57](#_Toc378519027)

[Table 44 - BUS to encoder generic 57](#_Toc378519028)

[Table 45 - message encoder signals 58](#_Toc378519029)

[Table 46 - message encoder generics 59](#_Toc378519030)

[Table 47 - FIFO signals 59](#_Toc378519031)

[Table 48 - FIFO Generics 60](#_Toc378519032)

[Table 49 - UART\_TX signals 60](#_Toc378519033)

[Table 50 - Wishbone Master interface 66](#_Toc378519034)

[Table 51 - Wishbone Master generics 66](#_Toc378519035)

[Table 52 - whishbone slave interface 67](#_Toc378519036)

[Table 53 - Wishbone Slave generics 68](#_Toc378519037)

[Table 54- Inputs & outputs output block 74](#_Toc378519038)

[Table 55 - Register Write Message 74](#_Toc378519039)

[Table 56- Register read request 75](#_Toc378519040)

[Table 57- Register read reply 75](#_Toc378519041)

[Table 58-Part a simulations 77](#_Toc378519042)

[Table 59- Part b simulations 82](#_Toc378519043)

[Table 60- Part b simulations 84](#_Toc378519044)

[Table 61- Problems & solutions 87](#_Toc378519045)

[Table 62- First problem simulation 88](#_Toc378519046)

# 

# 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 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 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.

## 1.3 Project Requirements

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.
* Units with wishbone master interfaces: RX path, TX path, core
* Units with wishbone slave interfaces: TX path, signal generator,core
* The wishbone work with a pipeline mode.
* The wishbone contains watchdog timers that avoid a system hang.
* The transactions used are:
* Read single
* Write single
* Read burst
* Write burst

1. The communication between the user and the FPGA is through Uart protocol.
2. UART protocol:
   1. Line not active = '1'
   2. 8 bits will be wrapped by *start bit*, represented by '0', and *stop bit*, represented by '1'.
   3. Parity bit will be used in order to verify the bit physical transmission :
      1. Odd - a bit will be added so the total '1' bits will be odd.
      2. Even - a bit will be added so the total '1' bits will be even.
      3. Inhibited.
3. Message Pack Structure transferred on UART lines:
   1. SOF - Start of Frame: “00111100” (0x3C)– one byte.
   2. Type – Indicates which client is being accessed - one byte.
   3. Address – Address of the register in a client – tree byte.
   4. Length - Number of Bytes of Data - two bytes.
   5. Data – Data written or read from registers in clients or from the FLASH - [length] bytes.
   6. CRC - A check if a successful data transfer was made. CRC will be calculated on the TYPE, ADDRESS, LENGTH and DATA blocks, in that order – one byte.
   7. EOF - End of Frame: “10100101” (0xA5)– one byte.
4. All the configurations will be saved in the core's registers at the initial stage.
5. Input data and trigger signal will be injected from signal generator every clock cycle.
6. the user will be able to write to each register in the system and to read from it.

## **1.4 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

## 1.5 Applicable Documents

Following is the list of the applicable documents for additional reading:

1. Output block document:

<http://compressor-decompressor.googlecode.com/svn/DOC/Project_Doc/project_document_lzrw3_compression_core.doc>

1. Uart & Whishbon protocols:

<http://fpga-setting-using-flash.googlecode.com/svn/DOC/Project%20document/part> B/[Flash\_Setting\_Project\_Document.docx](http://fpga-setting-using-flash.googlecode.com/svn/DOC/Project%20document/part%20B/Flash_Setting_Project_Document.docx)

# 2 GENERAL DESCRIPTION

Our project is reusing blocks from other projects, such as: Rx Path, Tx Path,Wishbone Intercon, basic RAM, output block. See "Reused Blocks Architecture" chapter for more information.



Figure - The final system

## 2.1 Data Flow

**Initialization:**

User chooses the desired configurations (trigger type, trigger position and enable). Host transmits the information through Uart to the RX path. Then through the wishbone intercon to the WBS of the core. (1)

Moreover, the scene's type of the signal generator is established.(2)

Data is being sent from the signal generator to the core.(3)

The data will be sent to the output block, then to the TX path. There it will be wrapped in data package and sent through the Uart back to the user.(4)

**Data flow inside the core:**

The configurations are being saved in the registers.(5)

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.(6)

The WC sends the start address out to the RC. The WC also sends the data and validity signal to the RAM and send the WC\_finish signal to the registers.(7)

The RC sends an address of data to extract from the RAM.(8)

The RAM sends the data back to the RC.(9)

The RC sends the data to the In Out Coordinator and the RC\_finish signal to the registers.(10)

Then the In Out Coordinator will coordinate between the recorded data's width to the width of the bus in the Wishbone protocol.(11)

Flow diagram:

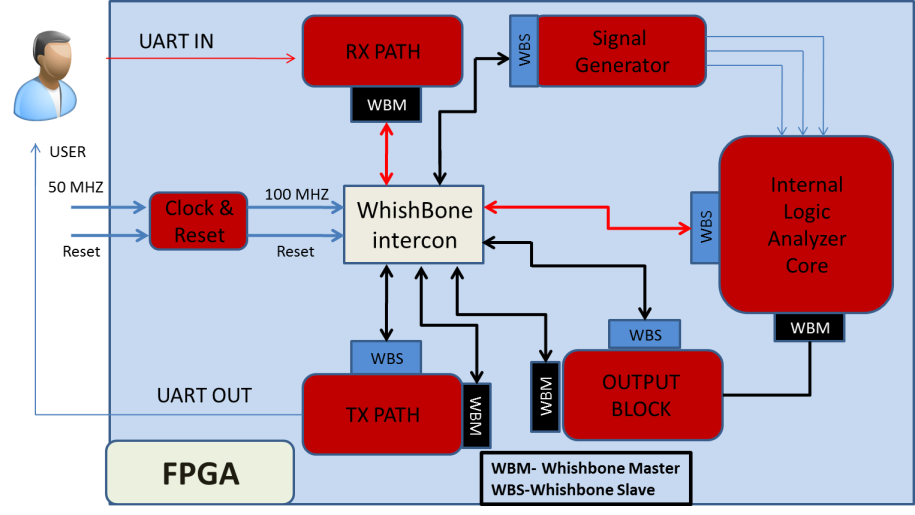


Figure -Top data flow(1)



Figure -Top data flow(2)

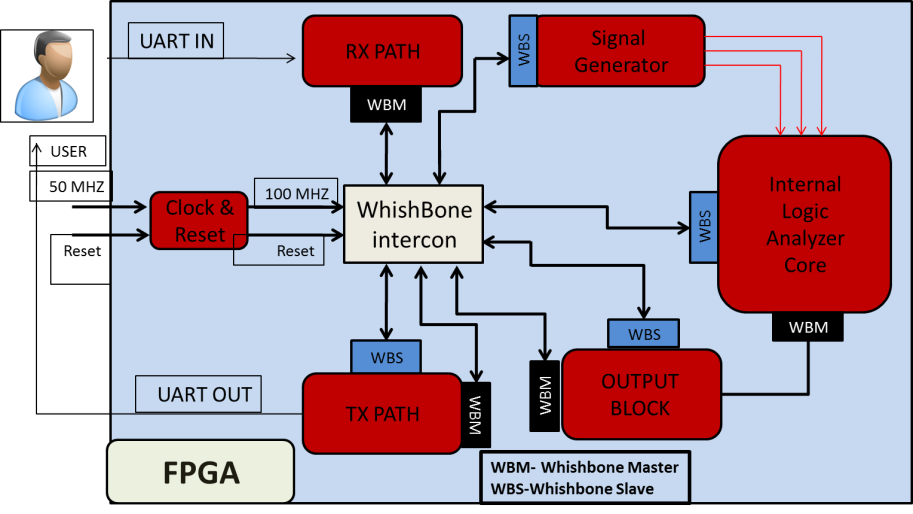


Figure -Top data flow(3)

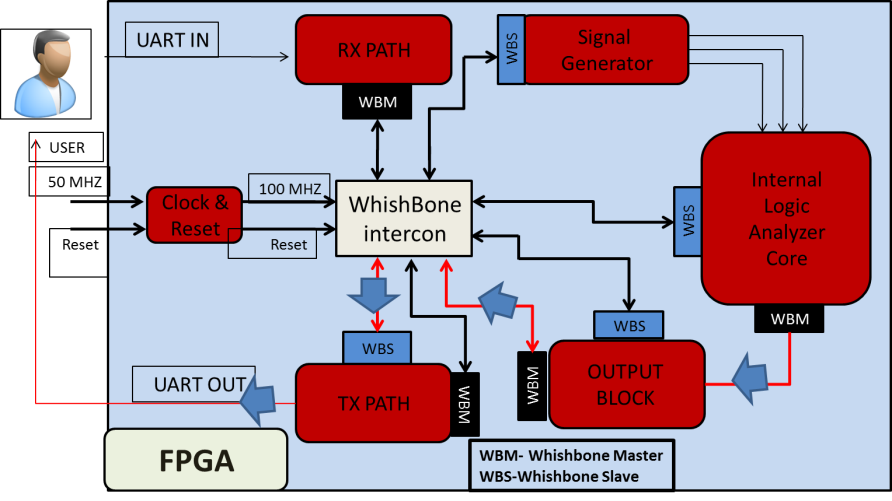


Figure - Top data flow(4)

# 

Figure -Core data flow (5)

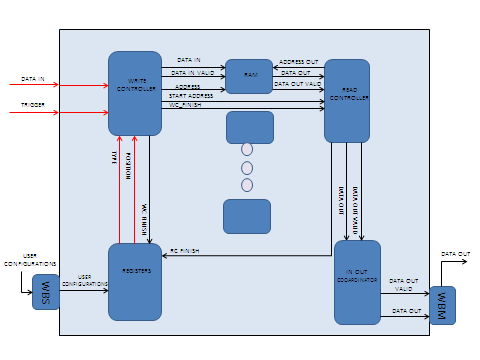


Figure - Core data flow (6)

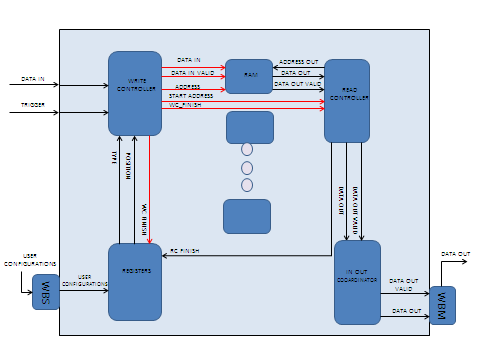


Figure - Core data flow (7)

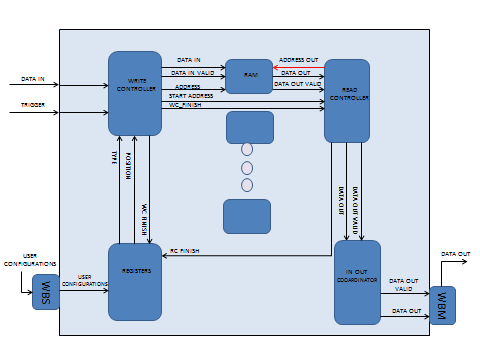


Figure - Core data flow (8)

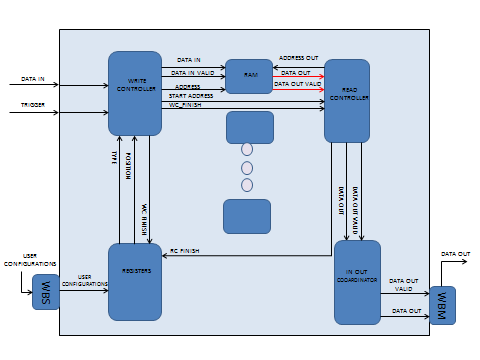


Figure - Core data flow (9)

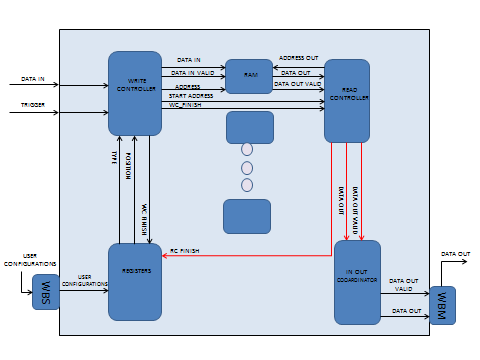


Figure - Core data flow (10)

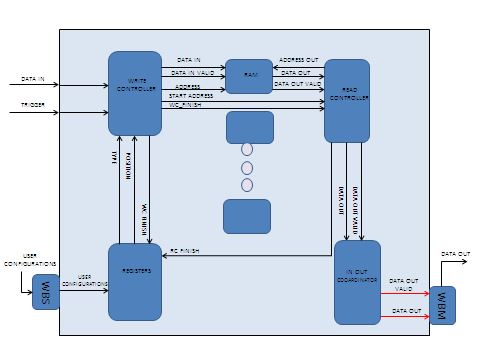


Figure - Core data flow (11)

# **3 MICRO ARCHITECTURE**

## 3.1 Registers Unit

General 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.

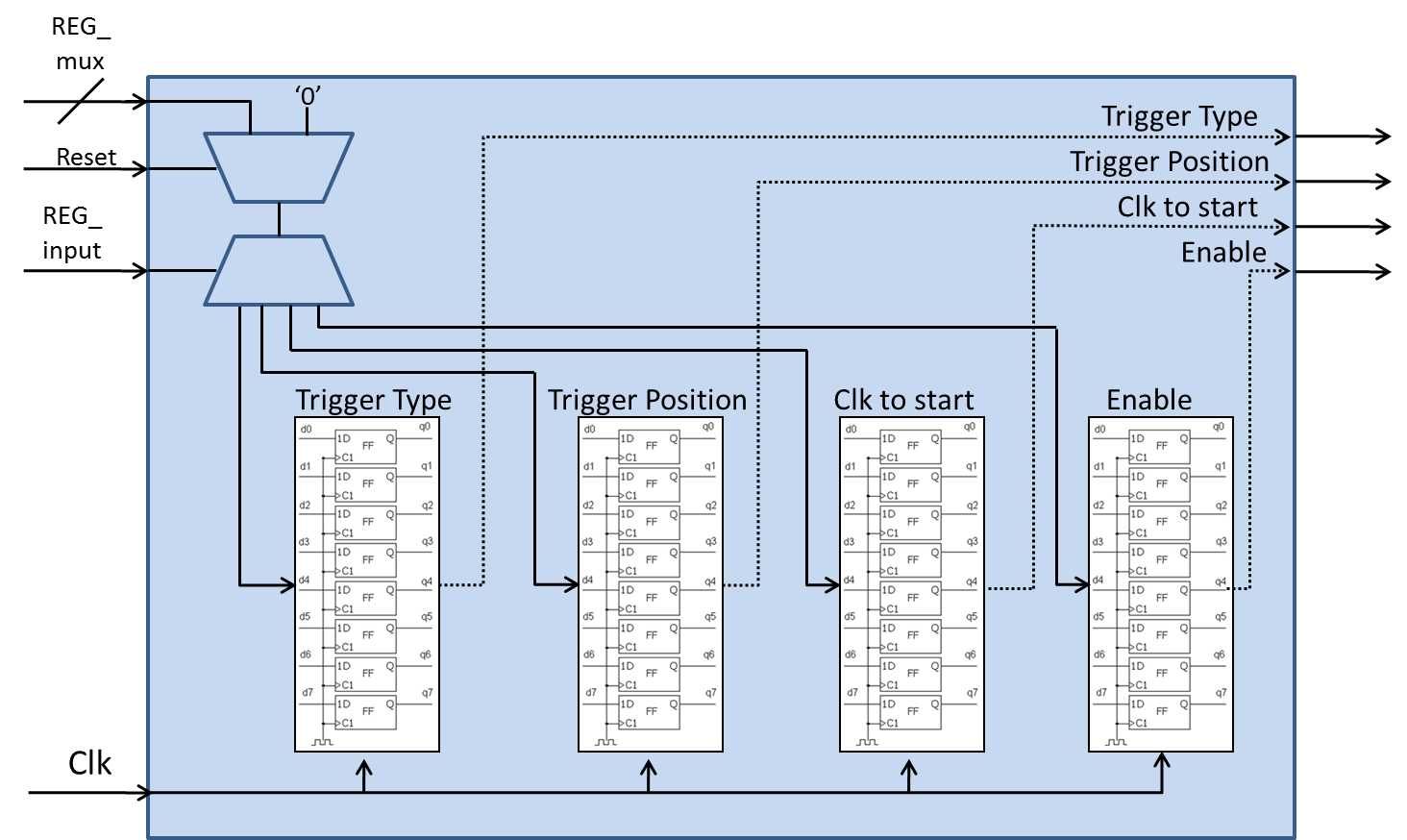


Figure - Registers unit

The unit contains four registers which are configured as follow:

trigger\_type\_reg\_1 = trigger\_type

the change in the trigger signal, that the system will define as trigger found.

Receives the values:

0(default value)- the trigger type defines as a rise of the trigger signal from low to high.

1-the trigger type defines as downfall from high to low.

2-the trigger type defines as high, i.e. three clock cycles in which high trigger signal is being sampled.

3-the trigger type defines as low, i.e. three clock cycles in which low trigger signal is being sampled.

trigger\_position\_reg\_2 = trigger\_position

a number in the range 0-100, which is the percentage of the data from the entire recorded data(which is recorded\_depth\_g^2) , that will be recorded before trigger rise.

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

will not be used eventually in our project.

enable\_reg\_4 = enable

excepts the values 0/1. Enable signal, defined according to 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 |

Table - 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 |

Table - Registers unit signals

### 

### 3.1.1 Simulation

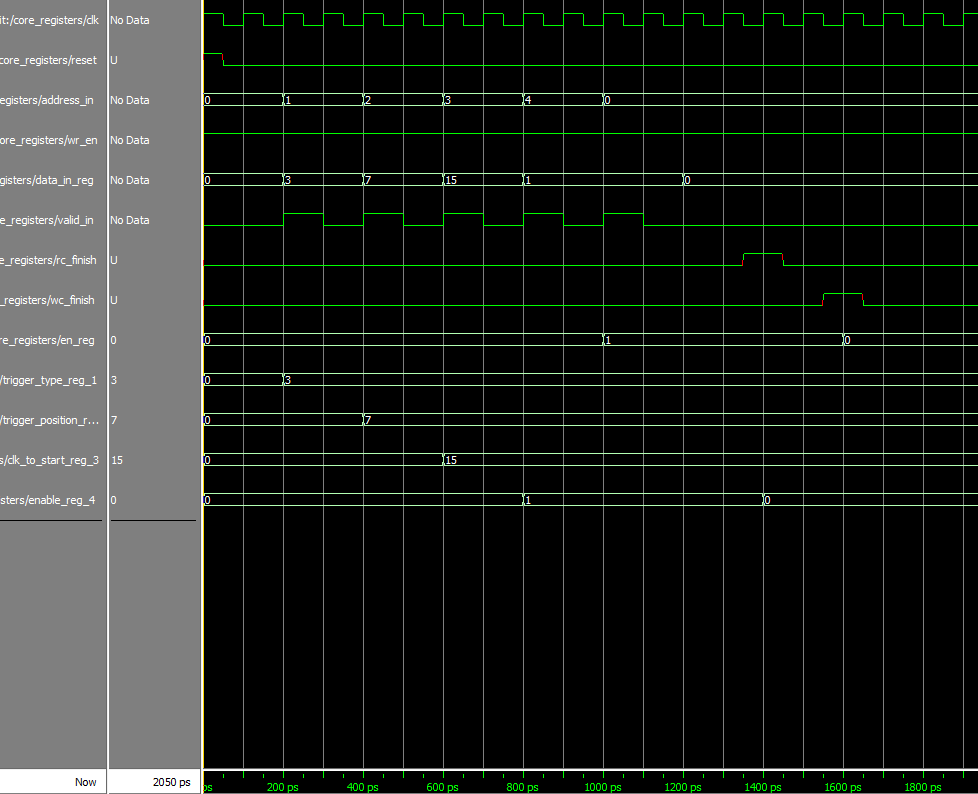


Figure - Registers unit simulation

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

General 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.

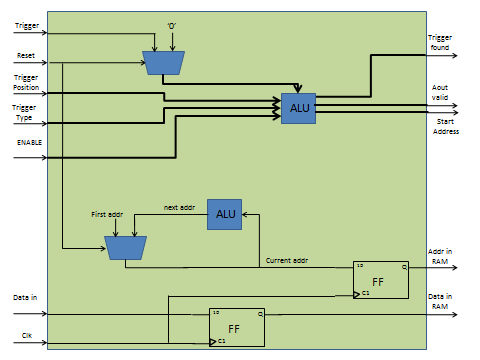


Figure - Write controller

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 |

Table - 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 |

Table - Write controller signals

Write controller FSM

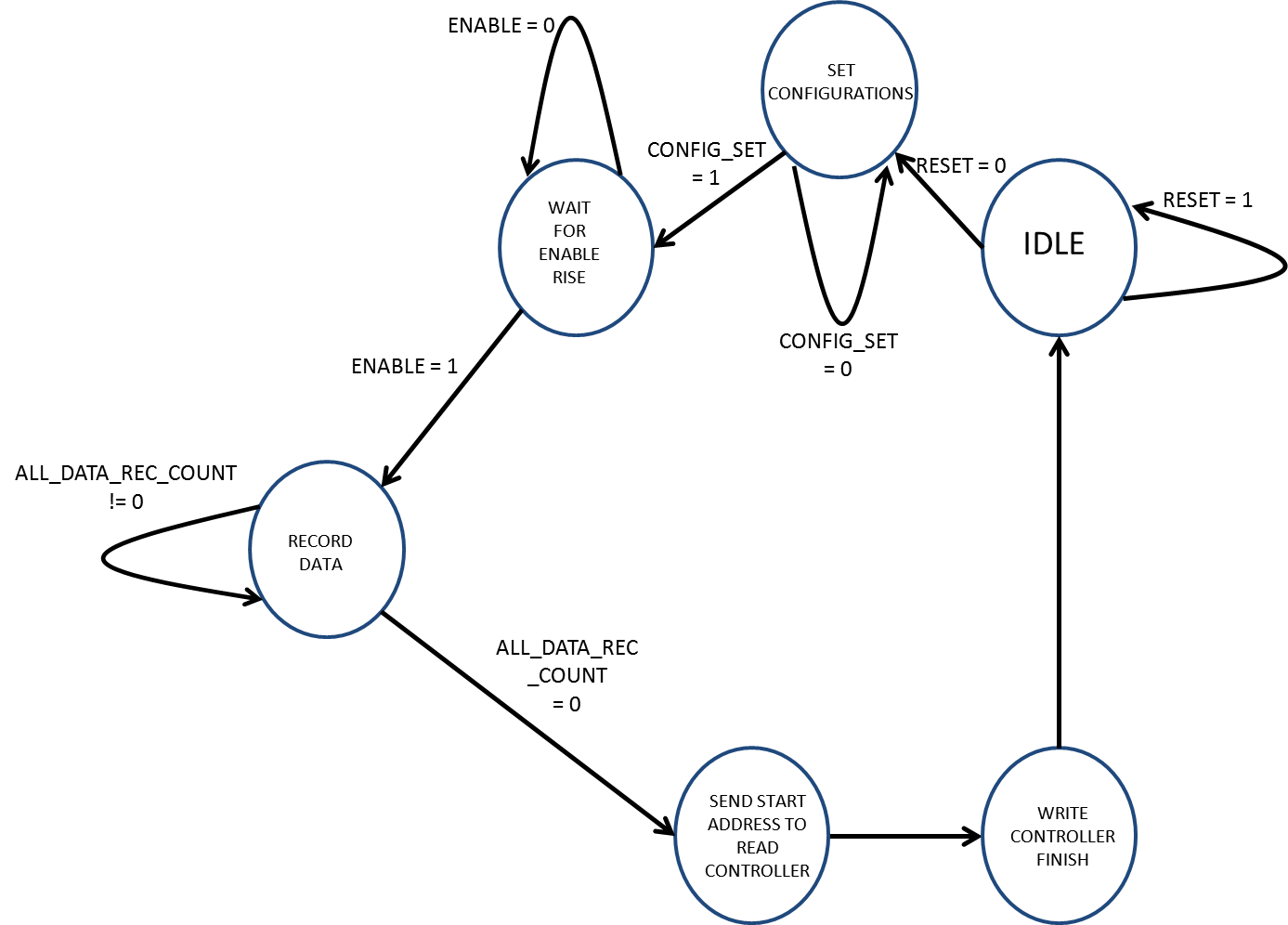


Figure - Write controller FSM

Output table

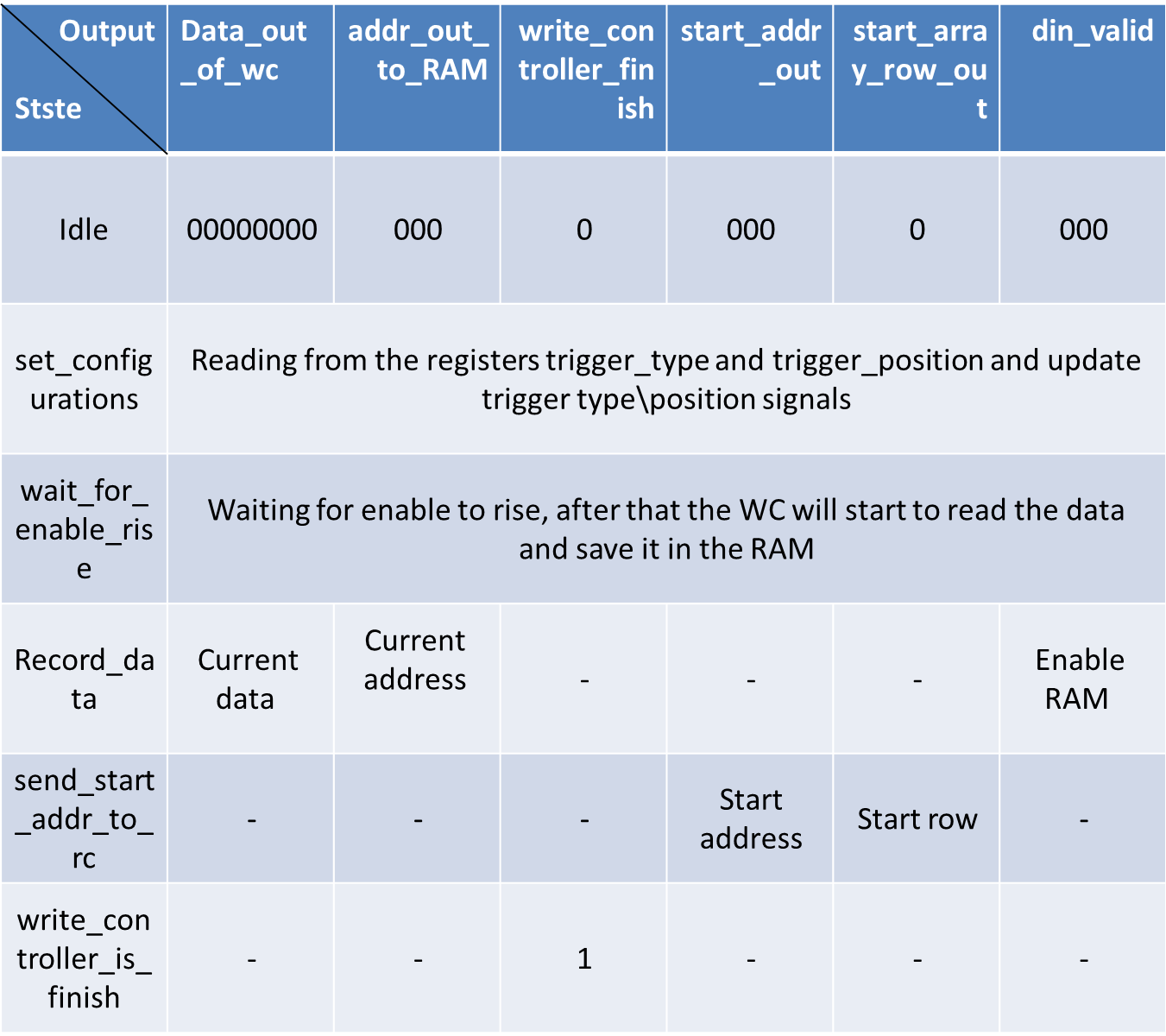


Table - Write controller output

### 3.2.1 Simulation

### 

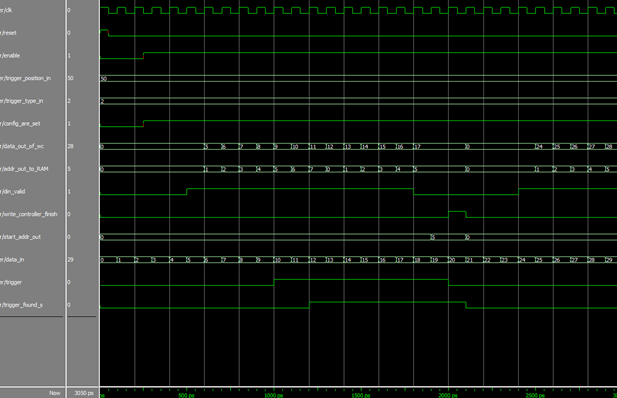


Figure - Write controller simulation

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

General 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.

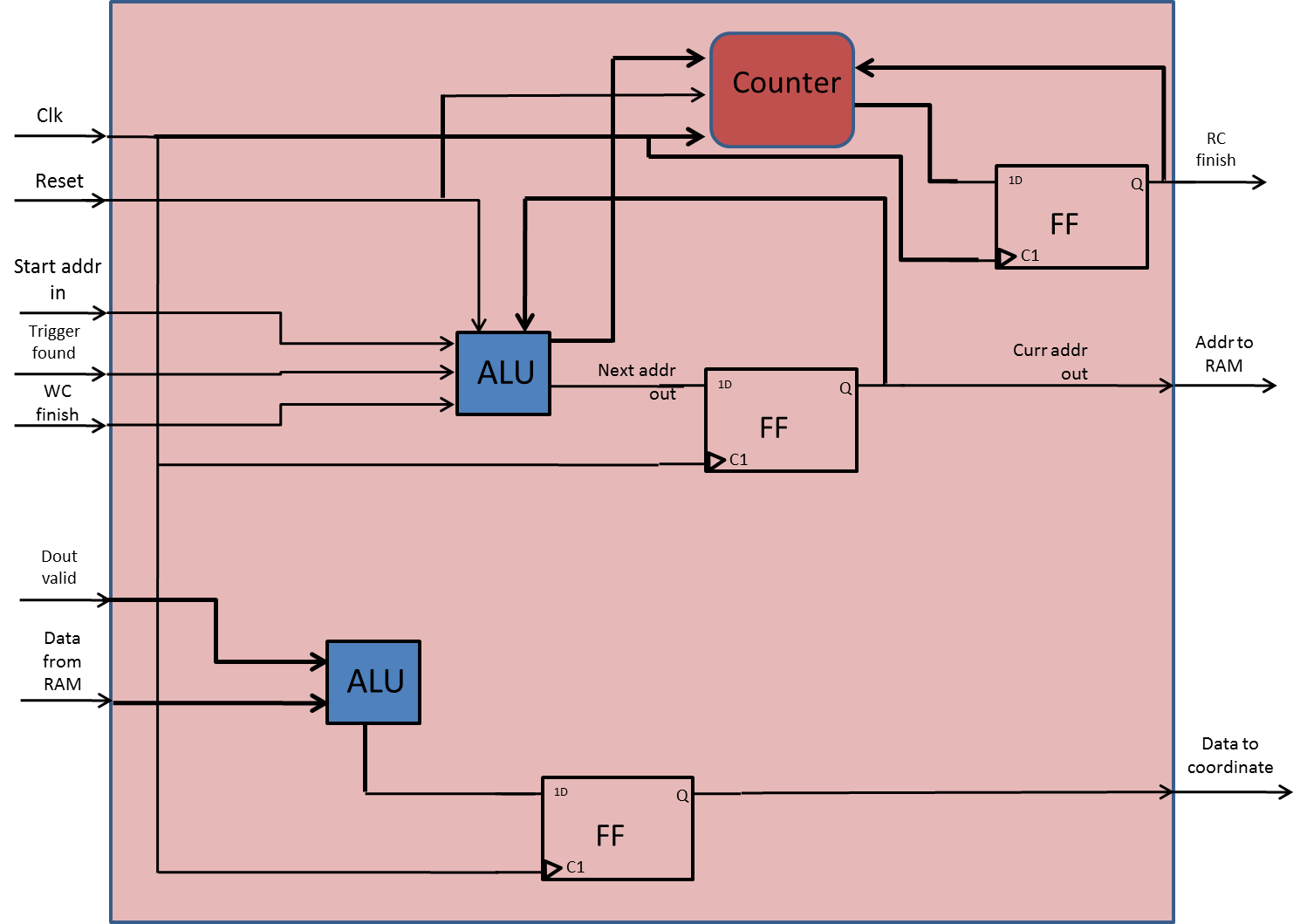


Figure - Read controller

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 |

Table - 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 |

Table - Read controller signals

Read controller FSM

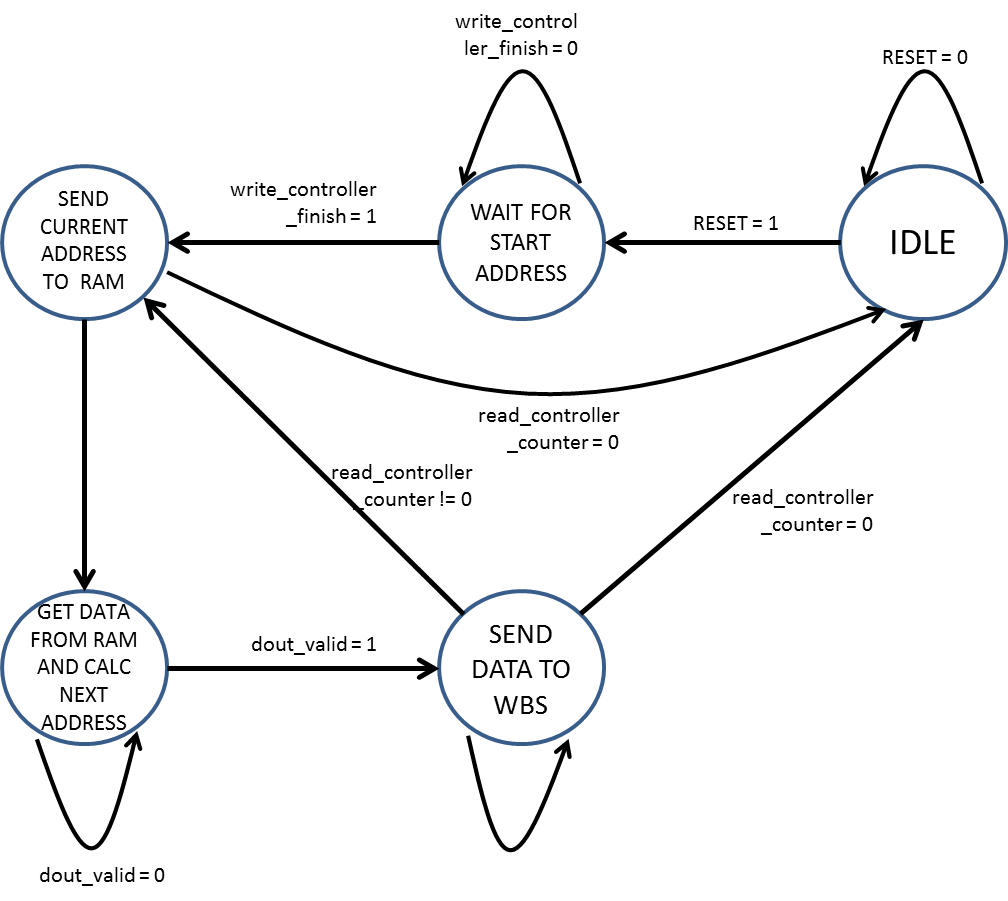


Figure - Read controller FSM

Output table

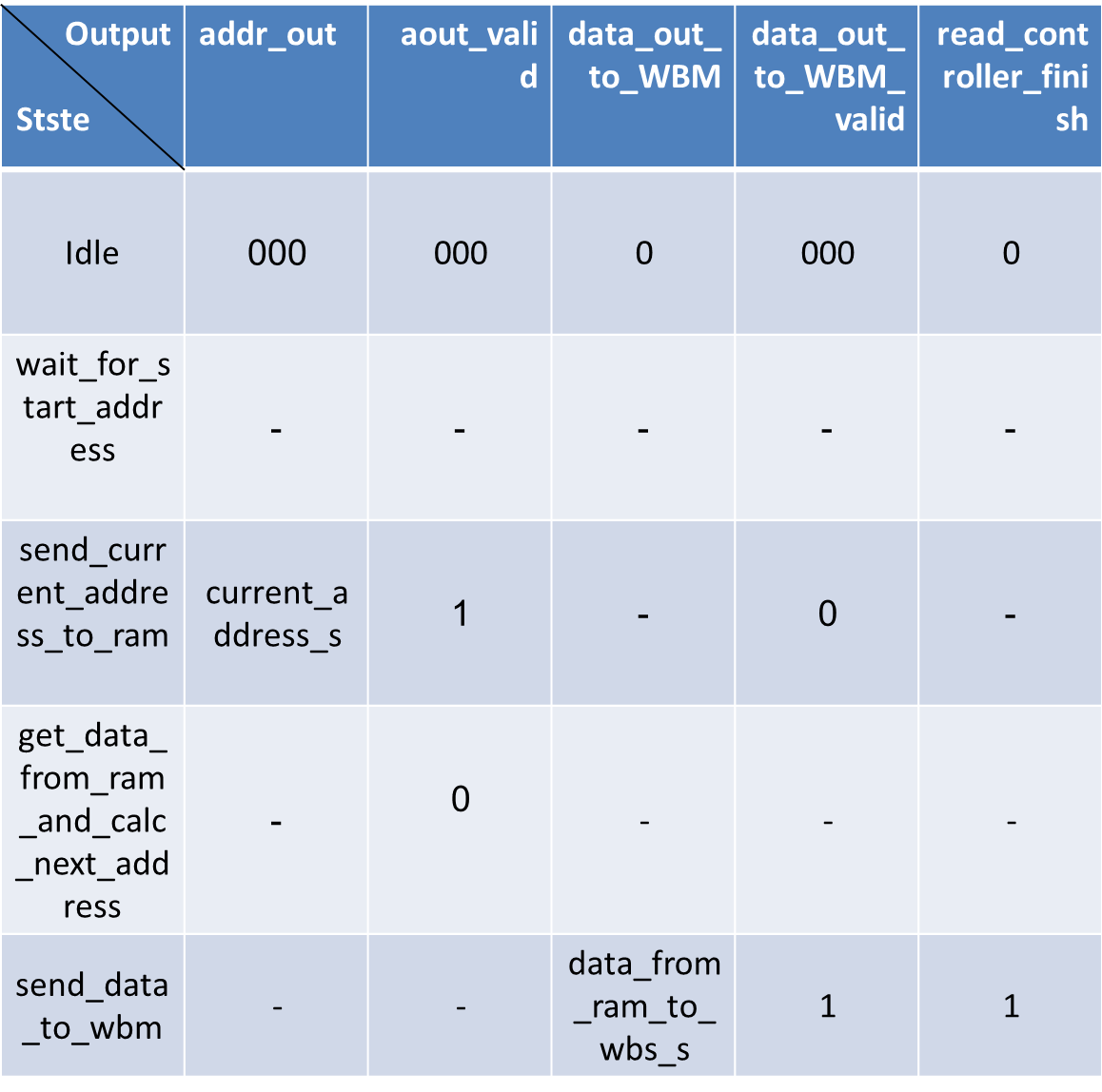


Table - Read controller output

### 3.3.1 Simulation

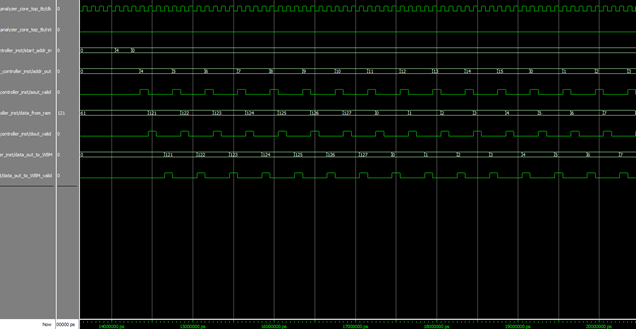


Figure - Read controller simulation

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:

General 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).

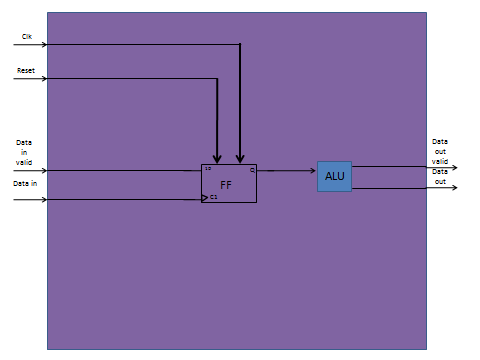


Figure - In out coordinator

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 |

Table - 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 |

Table - In out coordinator signals

### 3.4.1 Simulation

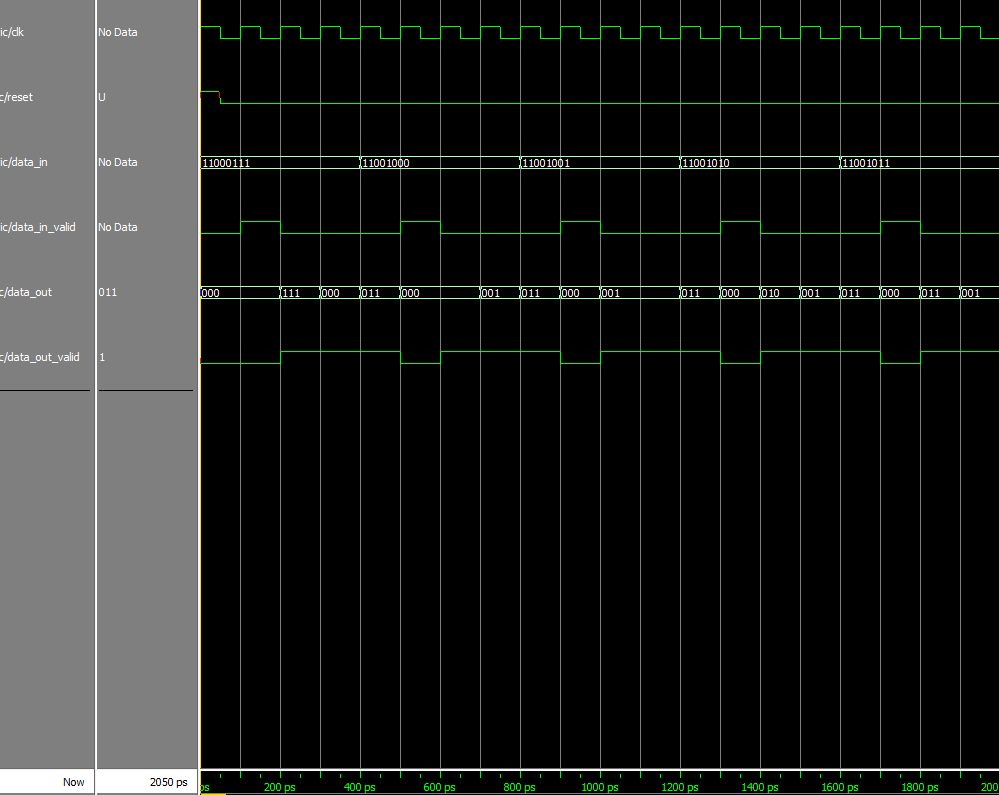


Figure - In out coordinator simulation

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.

### 3.4.3 Input>Output Coordinator

General 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 |

Table - 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 |

Table - In> out coordinator signals

## Input> Output Coordinator FSM

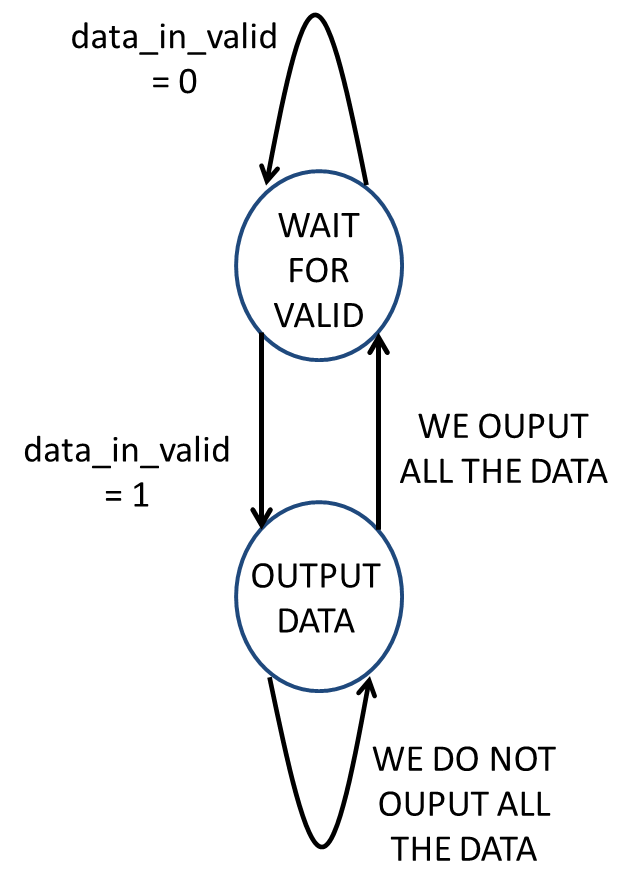


Figure - In >out coordinator FSM

Output table



Table - In> out coordinator output

### 

### 3.4.4 Input<Output Coordinator

General 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 |

Table - 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 |

Table - In< out coordinator signals

Input< Output Coordinator FSM

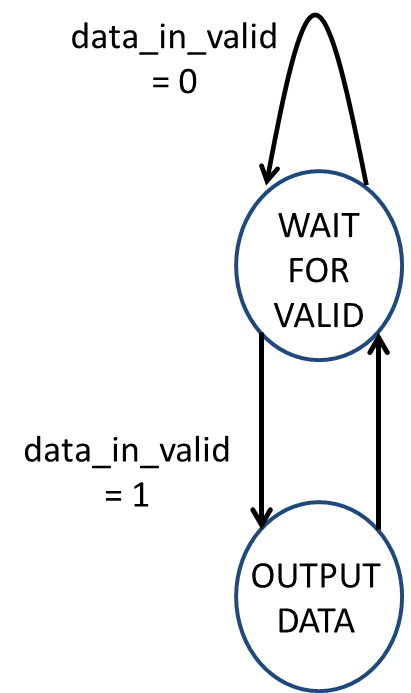


Figure - In <out coordinator FSM

Output table



Table - In< out coordinator output

### 3.4.5 Input=Output

General 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 |

Table - 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 |

Table - In= out coordinator signals

Input=Output Coordinator FSM

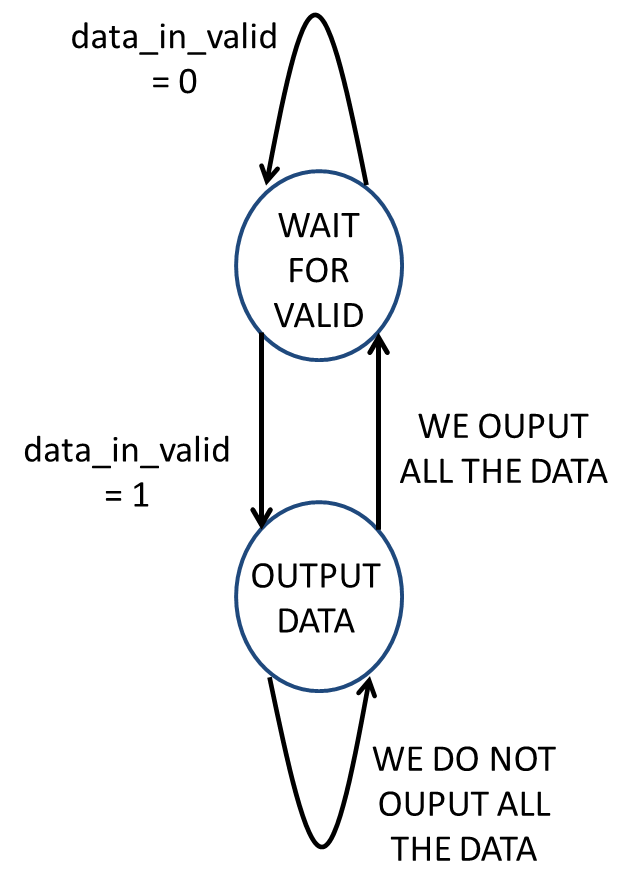


Figure - In =out coordinator FSM

Output table



Table - In= out coordinator output

## 

## 3.5 Enable FSM

General 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 |

Table - 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 |

Table - Enable signals

Enable FSM

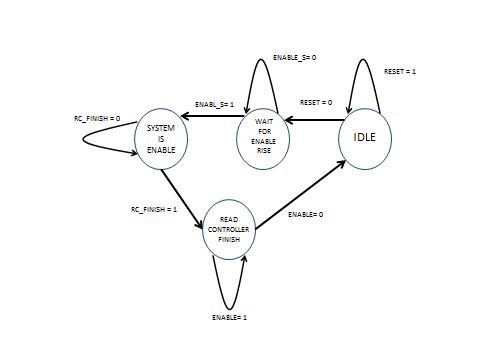


Figure - Enable FSM

Output table

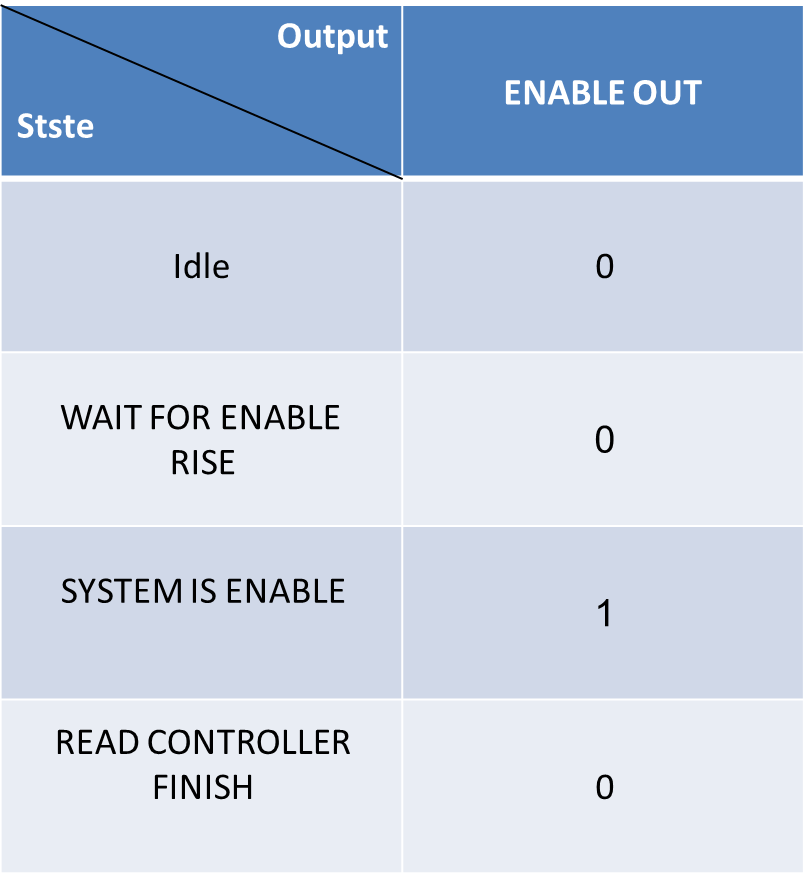


Table - Enable output

## 3.6 Memory Unit

General 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

Figure - 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 |

Table - 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 |

Table - 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

Figure - Generic RAM example

# 3.7 Internal Logic Analyzer Core Top

General Description

The core is built from 7 entities, all detailed in this chapter:

* WBS
* Registers
* Write Controller
* RAM
* Read Controller
* Data Coordinator
* WBM

The core tasks are getting and saving user configurations, getting new data each clock cycle and saving it, getting new trigger signal each clock cycle and check for trigger rise according user configurations and finally, extracting relevant data back to user.



Figure - The core

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) |

Table - Core generics table

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 |

Table - Core signals table

## 3.8 Signal Generator

General Description

The entity generates a trigger and data signals according the chosen scene, the user chose one of different scenes that are defined in the entity (we have 5 scenes for now).

The internal output data is a cyclic counter that change from 0 to (2^ num\_of\_signals\_g) – 1, but we can also get the data and trigger signals from an external source.

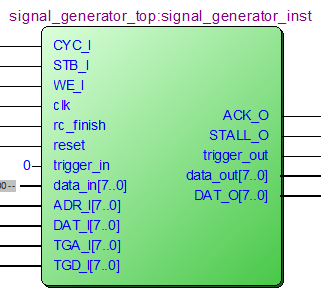


Figure - Signal generator

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 |
| num\_of\_signals\_g | 4 | number of signals that will be recorded simultaneously |
| external\_en\_g | 1 | 1 -> getting the data from an external source . 0 -> dout is a counter |
| Add\_width\_g | 8 | width of address word in the WB |
| len\_d\_g | 1 | Length Depth |
| type\_d\_g | 1 | Type Depth |

Table - Signal generator generic table

Signals table

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Direction | Width | Description |
| clk | In | 1 | System clock |
| Reset | In | 1 | System reset |
| data\_in | In | 1 | External data in |
| trigger\_in | In | num\_of\_signals\_g | External trigger in |
| 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 len word |
| reg\_data | In | data\_width\_g | data to be transmitted to the WM |
| reg\_data\_valid | In | 1 | data to be transmitted to the WM validity |
| stall | In | 1 | stall - suspend wishbone transaction |
| data\_out | Out | num\_of\_signals\_g | Output data |
| trigger\_out | Out | 1 | Output trigger |
| ACK\_O | Out | 1 | '1' when valid data is transmitted to MW or for successful write operation |
| DAT\_O | Out | data\_width\_g | data transmit to MW |
| STALL\_O | Out | 1 | STALL - WS is not available for transaction |
| typ | Out | (data\_width\_g)\*(type\_d\_g) | Type |
| len | Out | (data\_width\_g)\*(len\_d\_g) | Length |
| active\_cycle | Out | 1 | CYC\_I outputted to user side |

Table - Signal generator signals table

Signal generator FSM

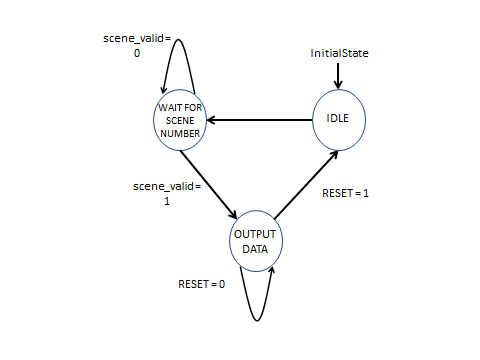


Figure 31- Signal generator FSM

Output table

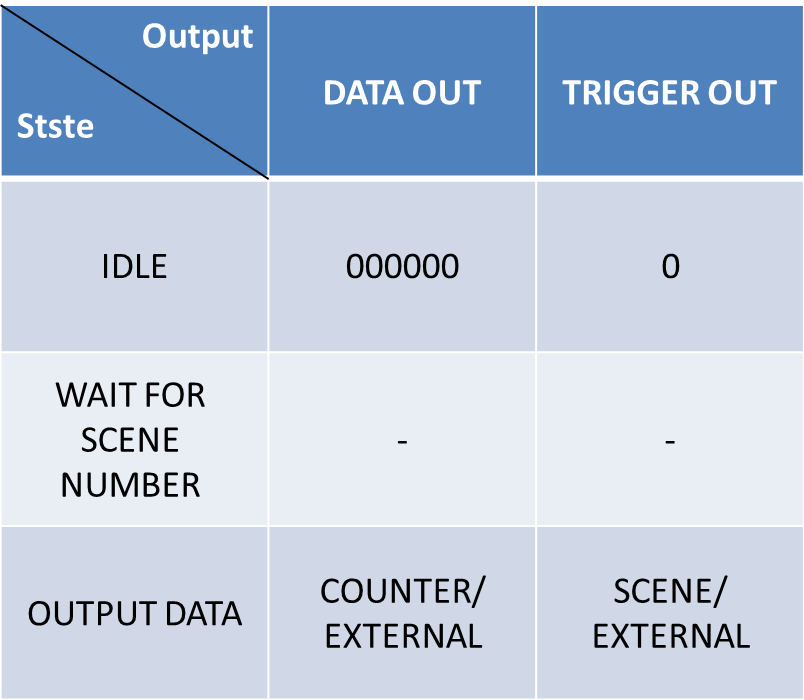


Table 29- Signal generator outputs

## Reused Blocks Architecture

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

### UART Protocol

#### 3.9.1.1 RX path

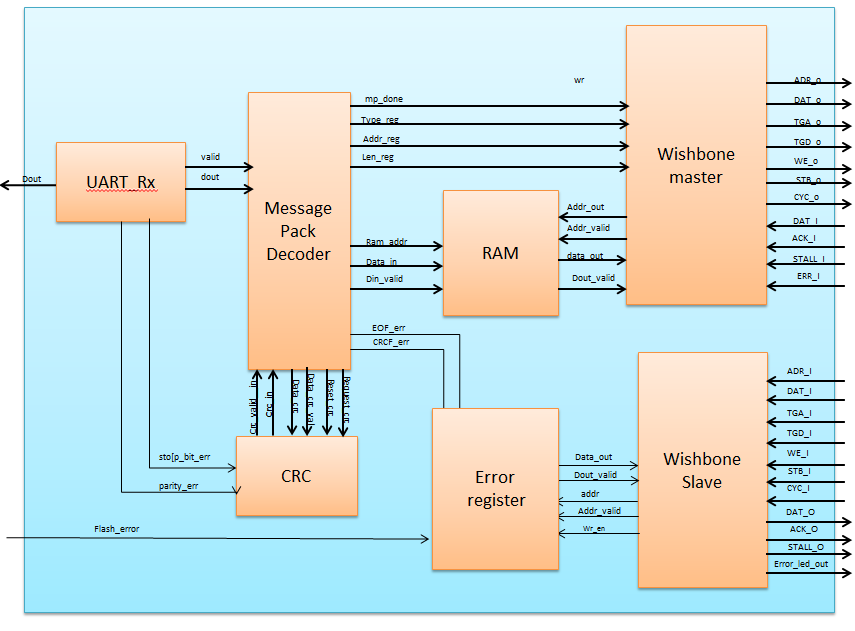


Figure - RX path

##### General Description

The RX Path processes data received from the host. It unwraps the data before it is transferred to the CCB or other clients of the bus.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| rx\_din | input | 1 | input of UART data |
| error\_led\_out | output | 1 | ‘1' when one of the error bits in the register is high |
| flash\_error | input | 1 | error signal from flash client - directed to error register |
| 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 |
| ADR\_I | input | addr\_d\_g \* data\_width\_g | contains the address 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 | type\_d\_g \* data\_width\_g | contains the type word |
| TGD\_I | input | len\_d\_g \* data\_width\_g | contains the length 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 |

Table - RX path interface

The generics of the rx\_path are all its sub-units generics.

The RX path contains the following units:

##### **3.9.1.1.1 UART RX**

This unit receives data via UART protocol. It converts the data received on the UART serial line to an 8 bit vector [dout] and sends it to the mp\_dec (message pack decoder unit). When the data is sent a valid signal is asserted. The uart\_rx also detects two types of errors:

* + - 1. Stop\_bit\_error – if the stop bit is different than ‘1’.
      2. Parity\_bit\_error – if the parity bit is different from the parity result calculated in the unit.

The errors are sent to the error register in the rx\_path and could be read later. An error assertion won’t interrupt the continuation of the data transfer.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| clk | input | 1 | system clock |
| Reset | input | 1 | block reset |
| Din | input | 1 | UART serial input |
| Dout | output | 8 | Parallel data out |
| valid | output | 1 | Parallel data valid |
| parity\_err | output | 1 | parity error |
| stop\_bit\_err | output | 1 | Stop bit error |

Table - Uart rx interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Actual value** | **description** |
| parity\_en\_g | natural | 0 | 1 to Enable parity bit, 0 to disable parity bit |
| parity\_odd\_g | Boolean | False | TRUE = odd, FALSE = even |
| uart\_idle\_g | std\_logic | ‘1’ | IDLE\_ST line value |
| baudrate\_g | positive | 115200 | UART baudrate [Hz] |
| clkrate\_g | positive | 100000000 | Sys. clock [Hz] |
| databits\_g | natural | 8 | Number of databits |
| reset\_polarity\_g | std\_logic | 1 | '0' = Active Low, '1' = Active High |

Table - Uart rx generics

##### 

##### **3.9.1.1.2 Message Pack Decoder (mp\_dec)**

The mp\_dec receives the [dout] vector transferred from the uart\_rx unit and prepares the data before transfer. It detects the start of the transmition (3C\_0x) then it checks in the TYPE, ADDRESS, LENGTH data to registers. The data that will be transferred is saved on the RAM. Finally it requests a CRC check. After receiving the EOF byte (A5\_0x) it asserts the mp\_done signal that informs the wishbone master to start the data transfer.

The mp\_dec detects two types of errors:

1. Eof\_error – if the EOF byte is different from A5\_0x
2. Crc\_error – if the CRC error received is different from the one calculated by the CRC block.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| Clk | input | 1 | clock |
| Rst | input | 1 | reset |
| Din | input | width\_g | Input data |
| Valid | input | 1 | Data valid |
| mp\_done | output | 1 | Message Pack has been received |
| eof\_err | output | 1 | EOF has not found |
| crc\_err | output | 1 | CRC error |
| type\_reg | output | width\_g \* type\_d\_g | type register |
| addr\_reg | output | width\_g \* addr\_d\_g | address register |
| len\_reg | output | width\_g \* len\_d\_g | length register |
| data\_crc\_val: | output | 1 | '1' when new data for CRC is valid, '0' otherwise |
| data\_crc | output | width\_g | Data to be calculated by CRC |
| reset\_crc | output | 1 | '1' to reset CRC value |
| req\_crc | output | 1 | '1' to request for current calculated CRC |
| crc\_in | input | width\_g \* crc\_d\_g | CRC value |
| crc\_in\_val | input | 1 | '1' when CRC is valid |
| write\_en | output | 1 | 1' = Data is available (width\_g length) |
| write\_addr | output | width\_g \* len\_d\_g | RAM Address |
| Dout | output | width\_g | Data to RAM |

Table - mp\_dec 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 |
| clkrate\_g | positive | 100000000 | Sys. clock [Hz] |
| len\_dec1\_g | Boolean | True | TRUE - Received length is decreased by 1 ,to save 1 bit --  FALSE - Received length is the actual length |
| sof\_d\_g | positive | 1 | SOF Depth |
| type\_d\_g | positive | 1 | Type Depth |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |
| crc\_d\_g | positive | 1 | CRC Depth |
| eof\_d\_g | positive | 1 | EOF Depth |
| sof\_val\_g | natural | 60 | (3Ch) SOF block value. Upper block is MSB |
| eof\_val\_g | natural | 165 | (A5h) EOF block value. Upper block is MSB |
| width\_g | positive | 8 | Data Width (UART = 8 bits) |
|  | positive | 8 | RAM size in bytes(2^8 = 256bytes) |

Table - mp\_dec generics

### 

##### **3.9.1.1.3 RAM**

##### 

A 256 byte RAM.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| Clk | input | 1 | clock |
| Rst | input | 1 | reset |
| addr\_in | input | addr\_bits\_g | Input address |
| addr\_out | input | addr\_bits\_g | Output address |
| aout\_valid | input | 1 | Output address is valid |
| data\_in | input | width\_in\_g | Input data |
| din\_valid | input | 1 | Input data valid |
| data\_out | output | width\_in\_g | Output data |
| dout\_valid | output | 1 | Output data valid |

Table - RAM 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 |
| clkrate\_g | positive | 100000000 | Sys. clock [Hz] |
| addr\_bits\_g | positive | 8 | Depth of data (2^10 = 1024 addresses) |

Table - RAM generics

##### **3.9.1.1.4 CRC**

A block that calculates the CRC value of the data transferred to it. The mp\_dec uses this block for comparing the CRC value received with the one calculated. The system uses an 8 degree polynomial calculation.

The CRC polynomial is **0xEA 🡪** 

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| Clk | input | 1 | clock |
| Rst | input | 1 | reset |
| Soc | input | 1 | start of calculation |
| Data | input | 8 | data in |
| data\_valid | input | 1 | data in valid |
| Eoc | input | 1 | end of calculation |
| Crc | output | 8 | crc value |
| crc\_valid | output | 1 | crc value validity |

Table - CRC 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 |
| clkrate\_g | positive | 100000000 | Sys. clock [Hz] |

Table - CRC generics

**Wishbone Master**

See *Wishbone Master* at wishbone units.

**Wishbone Slave**

See *Wishbone Slave* at wishbone units.

##### **3.9.1.1.5 Error Register**

Samples an 8 bit vector of error bits. The error vector is sampled every cycle and saved in a register. When the rx\_path’s requests the value of the register it is transferred to it and being set to zero on the following cycle. The unit also asserts the error\_led\_out signal – it is an OR operation on the error register saved bits. If this signal is ‘1’ a led would be lightened, meaning that at least one error has occurred.

The Error Register actually contains two register. Register #1 contains the code version.

Register #0 contains the error bits which have occurred:

On the current configuration:

Error\_in[0] – stop\_bit\_error

Error\_in[1] - parity\_error

Error\_in[2] – eof\_error

Error\_in[3] – crc\_error

Error\_in[4] – FLASH timeout

Error\_in[5..7] – not in use, set to ‘0’

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| Clk | input | 1 | clock |
| Rst | input | 1 | reset |
| error\_in | input | data\_width\_g | error vector |
| error\_led\_out | output | 1 | '1' when one of the error bits in the register is high |
| data\_out | output | data\_width\_g | data sent to WS |
| valid\_data\_out | output | 1 | validity of data directed to WS |
| address\_in | input | address\_width\_g | address line |
| valid\_in | input | 1 | validity of the address directed from WS |
| wr\_en | input | 1 | enables reading the error register |

Table - Error Register 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 |
| address\_width\_g | natural | 8 | defines the width of the address lines of the system |
| led\_active\_polarity\_g | std\_logic | 1 | defines the active state of the error signal input: '0' active low, '1' |
| error\_register\_address\_g | natural | 0 | defines the address that should be sent on access to the unit |
| error\_active\_polarity\_g | std\_logic | 1 | defines the polarity which the error signal is active in |
| code\_version\_g | natural | 0 | Hardware code version |

Table - Error register generics

#### 3.9.1.2 TX path

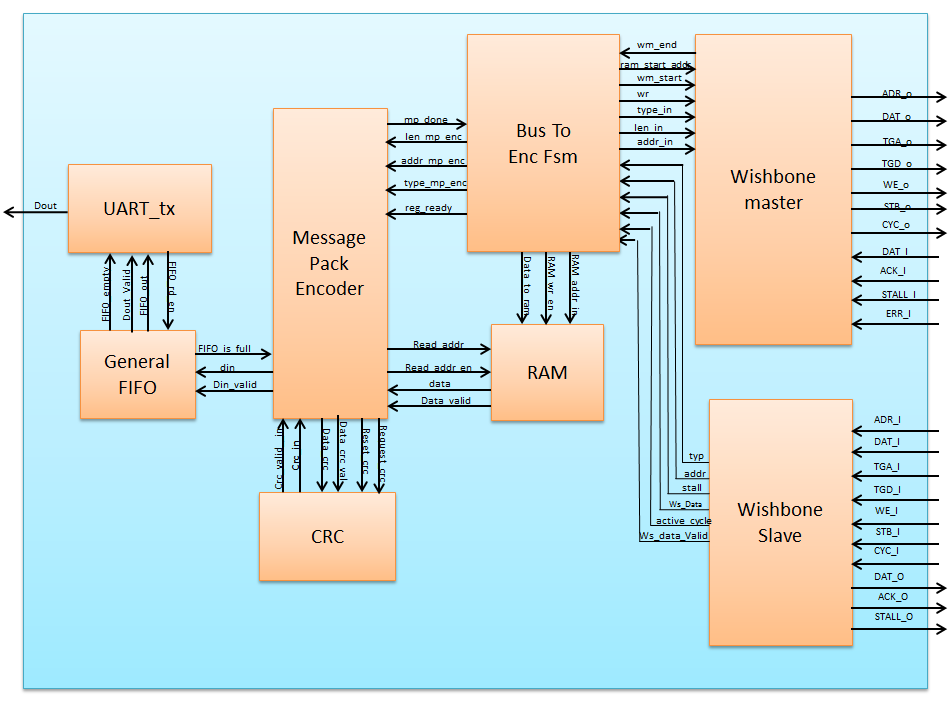


Figure - TX path

General Description

The TX is activated when a read transaction is occurring.

The TX (via wishbone slave) is informed that there is a read request, the Bus\_To\_Enc\_Fsm receive the data request properties, transmit them to the Message Pack Encoder, and store the data in the RAM. Afterwards, the slave will upraise acknowledge . Then the master of the TX is activated and asks to read the required data from the FLASH or from one CLIENT. When the TX master receives the data he wraps it in the form of the message pack structure according to the type, address and length than Transferred over to him.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **Description** |
| clk | input | 1 | system clock |
| reset | input | 1 | system reset |
| DAT\_I\_S | input | (data\_width\_g)\*(addr\_d\_g) | data inpute from the RX via the WB slave to the RAM |
| ADR\_I\_S\_TX | input | (data\_width\_g)\*(type\_d\_g) | address where the data will be write |
| TGA\_I\_S\_TX | input | (data\_width\_g)\*(len\_d\_g) | the type of the client that the data come from |
| TGD\_I\_S\_TX | input | 1 | the length of the data |
| WE\_I\_S\_TX | input | 1 | write enable to the RAM |
| STB\_I\_S\_TX | input | 1 | 1' for active bus operation, '0' for no bus operation |
| CYC\_I\_S\_TX | input | 1 | 1' for bus transmition request, '0' for no bus transmition request |
| ACK\_O\_TO\_M | output | 1 | 1' when the data were successfully write |
| DAT\_O\_TO\_M | output | data\_width\_g | data output via WM to client |
| STALL\_O\_TO\_M | output | 1 | stall - suspend wishbone transaction |
| DAT\_I\_CLIENT | input | data\_width\_g | data inpute from the client |
| ACK\_I\_CLIENT | input | 1 | ack from the client's slave |
| STALL\_I\_CLIENT | input | 1 | stall - suspend wishbone transaction from client WS |
| ERR\_I\_CLIENT | input | 1 | Watchdog interrupts, resets wishbone master |
| ADR\_O\_CLIENT | output | data\_width\_g)\*(addr\_d\_g) | address to the WS client |
| DAT\_O\_CLIENT | output | data\_width\_g | data required by the client |
| WE\_O\_CLIENT | output | 1 | write enable to the client |
| STB\_O\_CLIENT | output | 1 | 1' for active bus operation, '0' for no bus operation |
| CYC\_O\_CLIENT | output | 1 | 1' for bus transmition request, '0' for no bus transmition request |
| TGA\_O\_CLIENT | output | (data\_width\_g)\*(type\_d\_g) | the type of the client to be send the data to |
| TGD\_O\_CLIENT | output | (data\_width\_g)\*(len\_d\_g | the length of the send data |
| uart\_out | output | 1 | the UART signal output from the system, to the host |

Table - TX Path interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Generic Parameter** | **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 |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |
| type\_d\_g | positive | 1 | Type Depth |
| fifo\_d\_g | positive | 9 | Maximum elements in FIFO |
| addr\_bits\_g | positive | 8 | Depth of data |

Table - TX Path generics

### 

##### 3.9.1.2.1 BUS to encoder FSM

The BUS to encoder FSM unit is an interface between a Wishbone Bus and the message pack encoder. Once a read request has arrived from WS, the unit asks the WM to read data from the requested client on the bus. WM writes the data to RAM. When data reading is finished the unit asserts the reg\_ready signal for the message pack encoder to start reading data from RAM.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| Clk | input | 1 | system clock |
| reset | input | 1 | system reset |
| typ | input | (data\_width\_g)\*(type\_d\_g) | Type |
| addr | input | (data\_width\_g)\*(addr\_d\_g) | the beginning address in the client that the information will be written to |
| ws\_data | input | data\_width\_g | data out to registers |
| ws\_data\_valid | input | 1 | data valid to registers |
| active\_cycle | input | 1 | CYC\_I outputed to user side |
| stall | output | 1 | stall - suspend wishbone transaction |
| wm\_start | output | 1 | when '1' WM starts a transaction |
| wr | output | 1 | determines if the WM will make a read('0') or write('1') transaction |
| type\_in | output | type\_d\_g \* data\_width\_g | type is the client which the data is directed to |
| len\_in | output | len\_d\_g \* data\_width\_g | length of the data (in words) |
| addr\_in | output | addr\_d\_g \* data\_width\_g | the address in the client that the information will be written to |
| ram\_start\_addr | output | addr\_bits\_g | start address for WM to read from RAM |
| wm\_end | input | 1 | when '1' WM ended a transaction or reseted by watchdog ERR\_I signal |
| reg\_ready | output | 1 | Registers are ready for reading. MP Encoder can start transmitting |
| type\_mp\_enc | output | data\_width\_g \* type\_d\_g | Type register |
| addr\_mp\_enc | output | data\_width\_g \* addr\_d\_g | Address register |
| len\_mp\_enc | output | data\_width\_g \* len\_d\_g | Length Register |
| mp\_done | input | 1 | Message Pack has been transmitted |

Table - BUS to encoder FSM interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Generic Parameter** | **type** | **Actual value** | **Description** |
| reset\_polarity\_g | std\_logic | 0 | reset active polarity |
| data\_width\_g | natural | 8 | defines the width of the data lines of the system |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |
| type\_d\_g | positive | 1 | Type Depth |
| addr\_bits\_g | positive | 8 | Depth of data in RAM |
| reset\_polarity\_g | std\_logic | 0 | reset active polarity |

Table - BUS to encoder generic

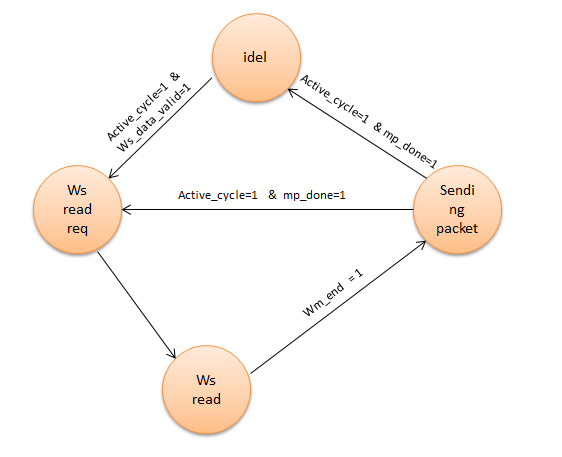


Figure - BUS to encoder FSM

##### 3.9.1.2.2 Message encoder

Message Pack Decoder Encoder transmits data from the Type and Address registers, and from the RAM, in a Message Pack format, wraps it and transfer the date to the UART.

This block also produces ‘mp\_done’ that signal the Bus\_To\_Enc\_Fsm while the present transaction is still running. When the mp\_done will drop to ‘0’ the Bus\_To\_Enc\_Fsm won’t try to write new message to the Message Encoder, but only after this signal will turn to ‘1’.

This unit also receives write enable, to allow writing the data to the RAM.

| Pin Name | Direction |  | Description |
| --- | --- | --- | --- |
| Clk | In | 1 | Clock |
| Rst | In | 1 | Reset. Reset polarity will be set according to the generic parameter 'reset\_polarity\_g' |
| Fifo\_full | In | 1 | FIFO is full, and cannot receive more data from MP Encoder |
| Reg\_ready | In | 1 | Input Type, Address and Data Length registers values are ready |
| Type\_reg | In | width\_g \* type\_d\_g | Input Type value. Will be valid together with the *reg\_ready* signal |
| Addr\_reg | In | width\_g \* addr\_d\_g | Input Address value. Will be valid together with the *reg\_ready* signal |
| Len\_reg | In | width\_g \* len\_d\_g | Input Data Length value. Will be valid together with the *reg\_ready* signal |
| Crc\_in | In | width\_g \* crc\_d\_g | Calculated CRC value from Checksum block |
| Crc\_in\_val | In | 1 | Calculated CRC value from Checksum block is valid |
| Din | In | Width\_g | Input data (payload), from RAM |
| Din\_valid | In | 1 | Input data (payload), from RAM is valid |
| Mp\_done | Out | 1 | Message Pack has been successfully transmitted. This flag will be raised together with the EOF output data |
| Dout | Out | Width\_g | Output data, to the FIFO |
| Dout\_valid | Out | 1 | Output data, to the FIFO, is valid |
| Data\_crc\_val | Out | 1 | Data to the CRC block is valid |
| Data\_crc | Out | width\_g | Data to the CRC, for CRC calculation |
| Reset\_crc | Out | 1 | Reset the CRC value to its default value |
| Req\_crc | Out | 1 | Request for calculated CRC value |
| Read\_addr\_en | Out | 1 | Address to RAM is valid |
| Read\_addr | Out | width\_g \* len\_d\_g | Address to RAM |

Table 45 - message encoder signals

| **Generic Parameter** | **type** | **Actual value** | **Description** |
| --- | --- | --- | --- |
| Reset\_polartiy\_g | Std\_logic | '0' | Reset active in this polarity |
| Len\_dec1\_g | Boolean | true | TRUE to receive decreased length by 1. For example: in case actual length is 6, 5 will be received. |
| Sof\_d\_g | positive | 1 | SOF block depth |
| Type\_d\_g | positive | 1 | Type block depth |
| Addr\_d\_g | positive | 3 | Address block depth |
| Len\_d\_g | positive | 2 | Length block depth |
| Crc\_d\_g | positive | 1 | CRC block depth |
| Eof\_d\_g | positive | 1 | EOF block depth |
| Sof\_val\_g | natural | 100 | Initial SOF value (decimal = 64hex) |
| Eof\_val\_g | natural | 200 | Initial EOF value (decimal = C8hex) |
| Width\_g | positive | 8 | Data width (number of bits) |

Table - message encoder generics

### 

##### 3.9.1.2.3 RAM

A 256 byte RAM. See signal and generics list at the RX path.

##### CRC

The CRC receive the data from the Message encoder, calculate it’s CRC value, return it to the Message Encoder which insert the CRC value into the UART package. See signal and generics list at the TX Path.

##### 3.9.1.2.5 FIFO

This block is a general FIFO, it receiving from the message encoder the data and arrange it in a queue before it arrive to the UART.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| clk | input | 1 | system clock |
| Reset | input | 1 | block reset |
| Din | input | width\_g | Input Data |
| rd\_en | input | 1 | Read Enable (request for data) |
| Flush | input | 1 | Flush data |
| Dout | output | width\_g | Output Data |
| Dout\_valid | output | 1 | Output data is valid |
| Afull | output | 1 | FIFO is almost full |
| Full | output | 1 | FIFO is full |
| Aempty | output | 1 | FIFO is almost empty |
| Empty | output | 1 | FIFO is empty |
| Used | output | log\_depth\_g | Current number of elements is FIFO. Note the range. In case depth\_g is 2^x, then the extra bit will be used |

Table - FIFO signals

| **Generic Parameter** | **type** | **Actual value** | **Description** |
| --- | --- | --- | --- |
| Reset\_polartiy\_g | Std\_logic | '0' | Reset active in this polarity |
| width\_g | positive | 8 | Width of data |
| depth\_g | positive | 9 | Maximum elements in FIFO |
| log\_depth\_g | natural | 4 | Logarithm of depth\_g (Number of bits to represent depth\_g. 2^4=16 > 9) |
| almost\_full\_g | positive | 8 | Rise almost full flag at this number of elements in FIFO |
| almost\_empty\_g | positive | 1 | Rise almost empty flag at this number of elements in FIFO |

Table - FIFO Generics

##### 3.9.1.2.6 UART TX

Receiving the data wrapped from the MESSAGE ENCODER and transmits it out to the HOST.

The UART require rate of 115,200 Kbit/sec.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| clk | input | 1 | system clock |
| Reset | input | 1 | block reset |
| Din | input | databits\_g | Parallel data in |
| fifo\_din\_valid | input | 1 | FIFO Ready to transmitted new data to TX |
| fifo\_empty | input | 1 | FIFO is not empty |
| fifo\_rd\_en | output | 1 | Controls FIFO rd\_en |
| Dout | output | 1 | Serial data out |

Table - UART\_TX signals

See generics list at the RX Path.

**Wishbone Master**

See *Wishbone Master* at wishbone units.

**Wishbone Slave**

see *Wishbone Slave* at wishbone units.

### 3.9.2 Wishbone Protocol

The Internal communication system uses the international Wishbone protocol. The Wishbone Bus is an open source hardware computer bus intended to let the parts of an integrated circuit communicate with each other. The aim is to allow the connection of differing cores to each other inside of a chip.

There are two types of clients on the wishbone bus: master and slave.

1 – wishbone master (WM) – active unit. Initiates bus cycles and ends them.

2 – wishbone slave (WS) - passive unit. Writes or reads data according to master request.

In our system we use the following signals in order to activate the wishbone bus.

CYC – '1' for bus transmition request, '0' for no bus transmition request

STB – '1' for active bus operation, '0' for no bus operation

DAT(sent from WM to WS) – data sent on the bus

ADR – initial address for transaction

TGA – type of client being accessed

TGD – length of the data ( length[data] -1)

WE – write enable

DAT – data sent on the bus

ACK(sent from WS to WM) - successful read or write operation ended by WS.

STALL – indicate that the slave is not available for transaction.

ERR - Watchdog interrupts, resets the wishbone master.

Wishbone cycle example: In the following waveform a master performs a write transaction.

The assertion of the CYC signal indicates the beginning of the new transaction, while the STB signal is high it indicates the data package transmission.

STB would de-asserted when master finishes sending or requesting data.

The maser will count the numbers of received ack’s (equal to the numbers of cycle the acknowledge was high) and after all the acks will are received, the CYC signal will drop and the transaction would end.

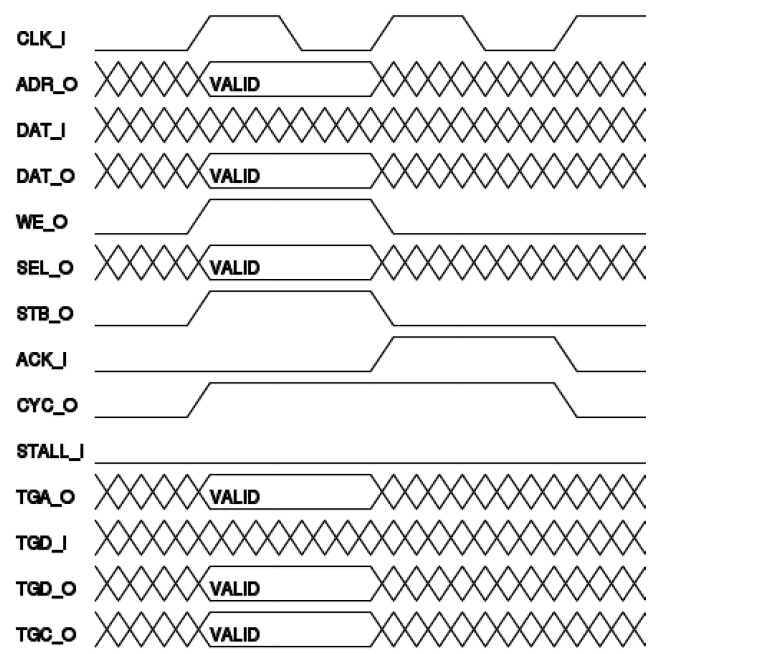


Figure - Wishbone cycle waveform example

This mode of work is a pipeline mode, in that way all the bytes are transmit one after the other, without waiting for an acknowledge after every byte.

\*More about the advantage of a pipeline mode compared to the standard mode in the system analysis chapter.

Numbering: Each master and slave has a unique number that characterizes it. For the wishbone slaves this number is the TGA signal.

Wishbone master 1 – RX path

Wishbone master 2 – TX path

Wishbone master 3 – Config Control Block (CCB)

Wishbone slave 1 – RX path

Wishbone slave 2 – TX path

Wishbone slave 3 – Wait client

Wishbone slave 4 – Led client

Wishbone slave 5 – Display client

Wishbone slave 6 – Flash control

As seen above there are three masters in the systems. Only one master can make a transaction on the bus. Therefore there is a routing policy implemented in the **wishbone\_intercon** unit.

### 3.9.3 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

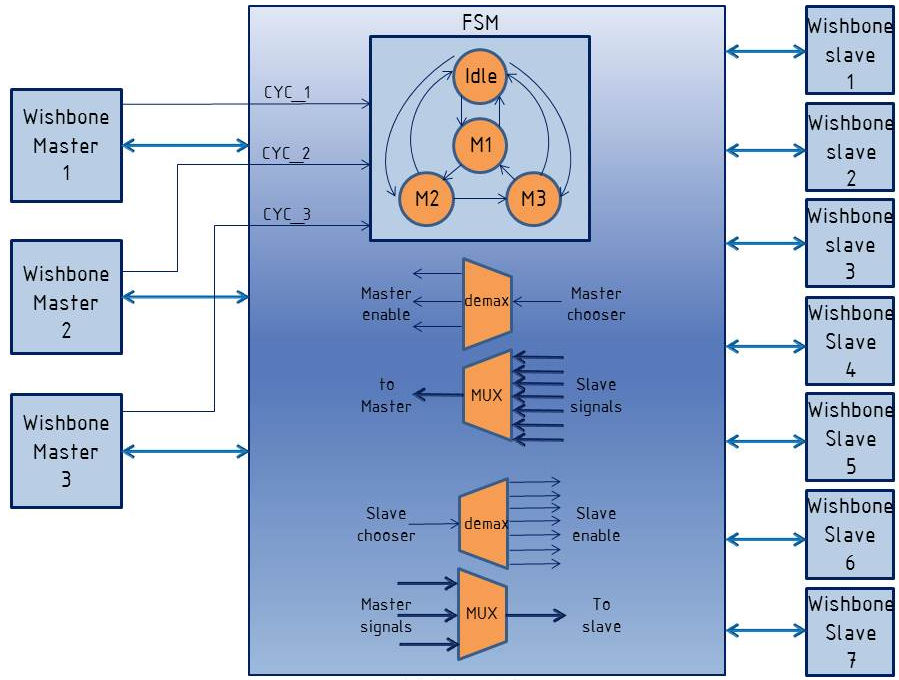


Figure - Wishbone Intercon

### 3.9.4 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.

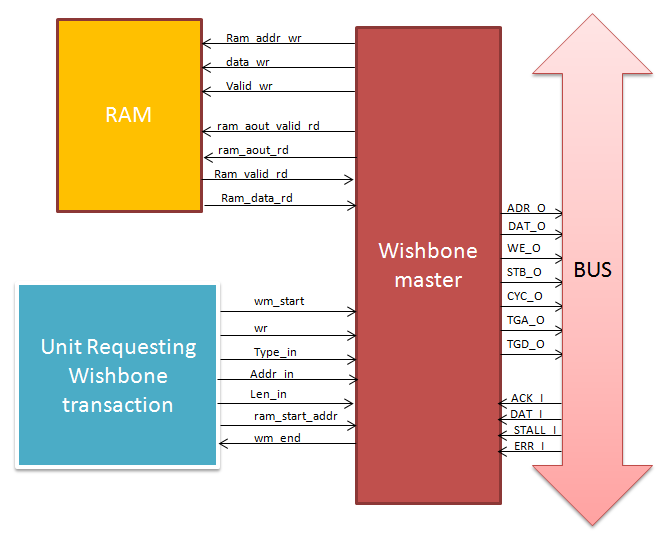


Figure - Connecting a Wishbone Master

**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

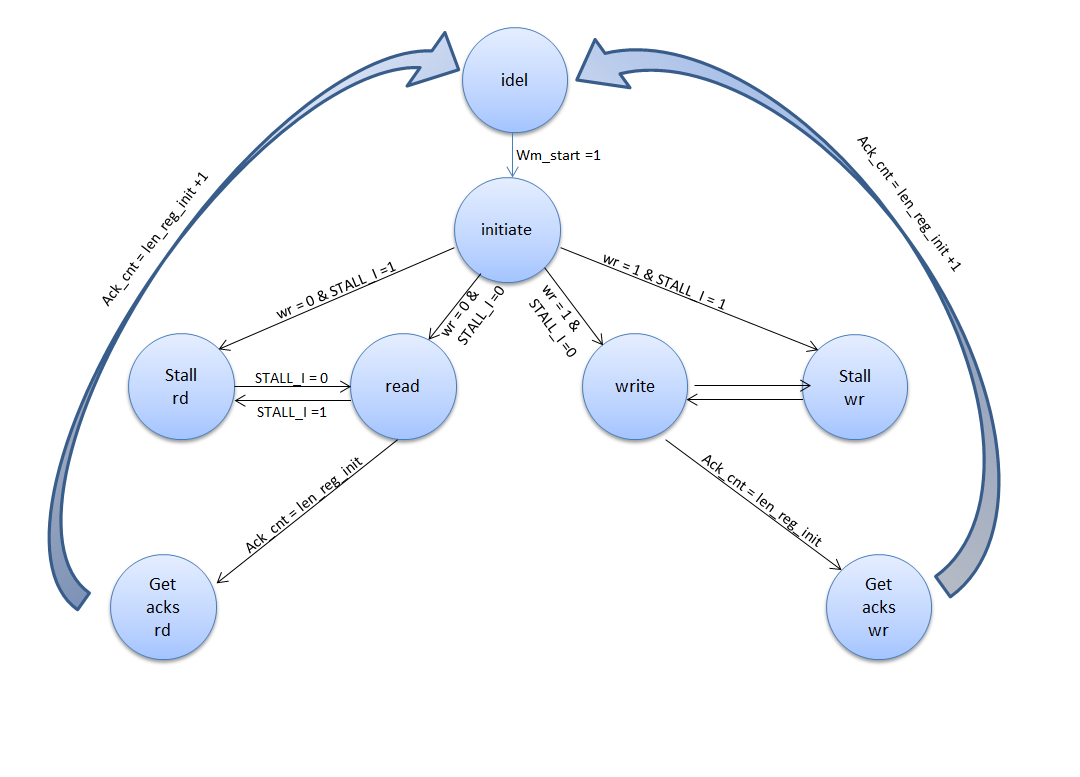


Figure - Wishbone Master FSM

|  |  |  |  |
| --- | --- | --- | --- |
| **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

### 

### 3.9.5 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.

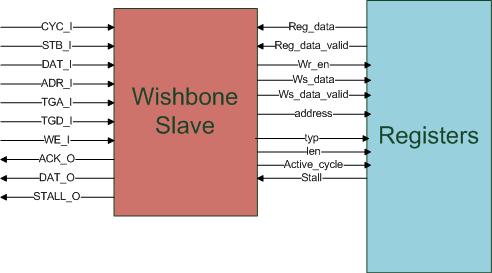


Figure - Connecting a Wishbone Slave

|  |  |  |  |
| --- | --- | --- | --- |
| **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

### 

### 3.9.6 OUTPUT BLOCK (compressed file memory)

General Description

The unit's goal is to be used as a data container between the core's fast data output to the slow TX PATH interface through wishbone protocol.

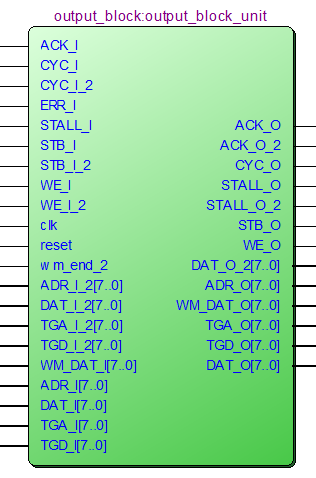


Figure - Output block

Functionality

The unit reads the whole file from the core and afterwards initiates its transmission to the TX PATH unit (via Wishbone intercon) which will encapsulate it with the header bytes and transmit it according to UART protocol.

Reused components:

* Wishbone master
* Wishbone slave
* 2X General FIFO

In order to initiate a transmission the unit contains a Wishbone master which transfers a message with "type = tx\_type\_g" field. This message will be routed by the Wishbone intercon according to this field to the Wishbone slave in the TX path unit. The message will contain 1 byte of data that'll contain the bytes amount that the unit intends to transmit to the TX path unit in this transaction.

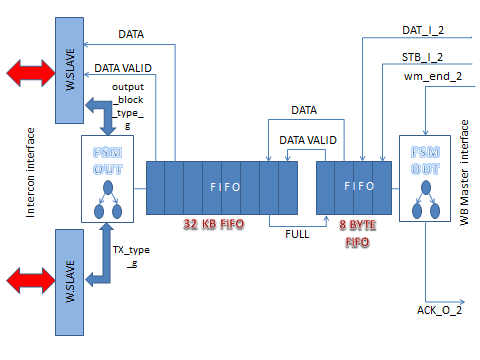
The TX path slave unit will transfer the data to the master of the unit and it will send requests that contains "type = output\_block\_type\_g" in order to receive data from the output block FIFO through the unit's slave.

* The TX path unit is reuse of an existing block which was built as unit that is being informed from where to read and how much. It could have been written as a unit that will directly receive data writing.

The unit contains 2 RAM memories that are used as a FIFO. The first memory is 8 byte long and the second is 32(KB) in order to be able to cope with full size input block. (the big memory is based on generic values in order to support block max size changing, the second memory is 8 bytes long in order to contain the added header in case of a 32KB uncompressed file).

The small memory is called short FIFO, reads data from the compressor and is managed by "FSM IN", it transfers the data to the large memory FIFO that is managed by "FSM OUT" which sends the data when it's called to the Wishbone Intercon that'll route them to the TX PATH.

The connection between the 2 memories is serial and is managed independently .

****

*Figure 41 - Output block illustration*

**Compressor data receiving- FSM IN :**

This FSM contains only 2 states, its goal is to control the output "ACK\_O\_2" that signals the core that the unit is available/unavailable to receive any more data.

As long as the core didn't finish transferring the whole file (the unit signals when its data transfer is finished by raising the port "wm\_end\_2") the unit is in a state where its ready to receive/ receiving more data – "buffering state".

When the core signals that it's done transferring data the unit moves to the next state where it sends data to the next unit and is not able to receive any more data (another compressed file) until all its data is transferred to the transmitting unit - TX PATH.

Then it'll move back to its first state when the FIFO is empty again.

**TX PATH data transfer – FSM OUT:**

This FSM needs to interact with the Wishbone unit and through it to initiate data transfer from the FIFO in case there is any.

The unit sends a transmit request to the TX path, in which it contains the data length it intends to transmit on the following transaction by using 1 byte in the data field (1 – 256) according to the max frame size.

After the Wishbone master receives an ACK for the request it sent (with the ACK comes a flag informing that the destination unit is free – stall=0), the machine will move to the following state in which FIFO data reading is performed to the BUS by demand from the master of the TX path unit. The data reading will be performed through the Wishbone slave unit of the sending unit (output block).

After sending the whole data in the current transaction, the unit moves to a state in which it waits until the destination unit will finish sending all the data that was sent to it on the last transaction on UART.

**Timing calculation between send requests:**

In order to calculate the transfer time of each byte through the UART the unit receives the line properties (baud rate, parity bit, stop bit etc.) as generics.

Each byte arriving to the TX path, is transferred by UART protocol, this frame size can change according to the given properties that reflects on the added headers to the data being transferred.

Additionally the UART baud rate reflects on the whole frame transmission. Since it's very slow comparing to the core clock many 50 MHz clock cycles will pass while the frame is transmitted.

Before sending the next data frame the system needs to wait a while according to the following equation:

requested bytes – the number of bytes transferred to the TX path unit.

Additionally you need to consider the data overhead addition:

* Addr\_depth\_g – depth of address word in the WB (configured to 8).
* Len\_d\_g – number of bytes assigned to represent the number of data bytes in the current frame (configured to 1).
* Type\_d\_g – number of bytes assigned to represent the type field (configured to 1).

Additionally attached the constant 4 that represents the remaining bytes (SOF, CRC, EOF, saved byte).

In addition the TX path's FIFO's depth needs to be considered so another transaction can start without remnants from the former transaction. That'll supply an additional safety period so all the FSMs will return to idle state.

All the last connections yield the number of bytes that the TX path unit needs to transmit on the current frame.

Each byte will be transmitted in UART protocol according to the following calculation:



* Data\_bits\_g – number of data bits sent in each frame.
* Parity\_en\_g – considers if in each frame a prity bit is added.
* 2 for start + stop bits.

Transfer time of each bit is the clocks speed ratio, for the calculation we'll consider the constant:



* The addition 1 is to make up for the remainder of the division.
* Division will not actually be performed but replaced.

**Interface signals:**

|  |  |  |
| --- | --- | --- |
| ***INPUTS*** | | |
| **Signal name** | **Width (bits)** | **Description** |
| Clk | 1 | System clock |
| Reset | 1 | System reset |
| ADR\_I\_2 | 24 | NOT IN USE |
| DAT\_I\_2 | 8 | Data from core |
| WE\_I\_2 | 1 | NOT IN USE |
| STB\_I\_2 | 1 | Validity of data from core |
| CYC\_I\_2 | 1 | NOT IN USE |
| TGA\_I\_2 | 8 | NOT IN USE |
| TGD\_I\_2 | 8 | NOT IN USE |
| Wm\_end\_2 | 1 | End Of Compressed file indication |
| ACK\_I | 1 | when data had received and recognize as valid. |
| WM\_DAT\_I | 8 | Input data (master) |
| STALL\_I | 1 | '1' when client (Tx path) sending data and not ready to receive another frame |
| ERR\_I | 1 | '1' when error (timeout) |
| ADR\_I | 24 | Address - NOT IN USE |
| DAT\_I | 8 | Data byte |
| WE\_I | 1 | '1' for write, '0' for read. Only write transaction should occur |
| STB\_I | 1 | indicates a valid data transfer cycle |
| CYC\_I | 1 | indicates that a valid bus cycle is in progress |
| TGA\_I | 8 | Type |
| TGD\_I | 8 | Length |
| ***OUTPUTS*** | | |
| **Signal name** | **Width (bits)** | **Description** |
| ACK\_O\_2 | 1 | '1' when ready to receive file  '0' when not ready (sending current file) |
| DAT\_O\_2 | 1 | NOT IN USE |
| STALL\_O\_2 | 1 | NOT IN USE |
| ADR\_O | 24 | Address (not in use) |
| WM\_DAT\_O | 8 | Output data (master) |
| WE\_O | 1 | '1' for write, '0' for read. Only write transaction should occur |
| STB\_O | 1 | indicates a valid data transfer cycle |
| CYC\_O | 1 | indicates that a valid bus cycle is in progress |
| TGA\_O | 8 | Type |
| TGD\_O | 8 | Length |
| ACK\_O | 1 | Output to BUS when data had received and recognize as valid |
| DAT\_O | 1 | Output data (slave) |
| STALL\_O | 1 | 1 when the block can't receive data |

Table - Inputs & outputs output block

# 4 DATA TRANSFER

## 4.1 Write transaction

### Registers Write Message

|  |  |  |
| --- | --- | --- |
| remarks | Content | Value |
|  | SOF | 0x3C |
| 0x04 – Signal Generator, 0x05 – Core | Write Register Type ID |  |
|  | Address [7:0] |  |
|  | Length [7:0] |  |
|  | Data #1[7:0] |  |
|  | Data #2[7:0] |  |
|  | .  .  . |  |
|  | Data #N[7:0] |  |
| Polynom is 0xEA | CRC [7:0] |  |
|  | EOF | 0xA5 |

Table - Register Write Message

* Min burst is 1
* Max burst is 256 limited by the size of the RAM in the RX path

When data is written to a client on the system it goes through the following stages:

1 - User enter in data to be sent as a packet.

2 – Data on the UART line enters the RX path in the **uart\_rx** block, one byte at a time. The data received is in the following format (the length in bytes is written inside the brackets [\*\*\*]):

SOF (Start Of Frame)[1] => Type [1] => Address [3] => Length [1] => Data [#data\_bytes] => CRC[1] => EOF (End Of Frame)

3 – The uart\_rx converts the data received on the UART line to an 8 bit vector and transfers it to the **mp\_dec** (Message Pack Decoder).

4 – The type, address and length fields are saved to registers in the mp\_dec. The data bytes are saved on the rx\_path’s **RAM**. The CRC byte received is compared to the CRC value calculated by the **CRC block**.

5 – If no errors are received (CRC error for example) the type, address and length data is transferred to wishbone master in order to start data transfer.

6 – The wishbone master reads data from RAM and starts a wishbone bus transaction.

7 – If the bus is not occupied by another master, the WM transfers the data using wishbone protocol to the client via **wishbone\_intercon** unit which does the correct routing using the type field.

8 – When successful writing of a burst is done, the master will receive the ACK signal from his slave, When the number of ACK assertion reaches the <Length + 1> the wishbone master will end the transaction.

## **4.2 Read transaction**

### Registers Read Request

|  |  |  |
| --- | --- | --- |
| remarks | Content | Value |
|  | SOF | 0x3C |
| 0x82 – Signal Generator, 0xA2 - Core | Read Register Type ID |  |
| Address of first register to read | Address [7:0] |  |
| Always 0 on read transaction | Length [7:0] | 0x00 |
| Length of the data to be read | Data [7:0] |  |
| Polynom is 0xEA | CRC [7:0] |  |
|  | EOF | 0xA5 |

Table - Register read request

\*\*Packet is delivered first to TX\_path, therefore its length is 1.

### Registers Read Reply

|  |  |  |
| --- | --- | --- |
| remarks | Content | Value |
|  | SOF | 0x3C |
| 0x04 – Signal Generator, 0x05 – Core | Read Register Type ID |  |
|  | Address [7:0] |  |
|  | Length [7:0] |  |
|  | Data #1[7:0] |  |
|  | Data #2[7:0] |  |
|  | .  .  . |  |
|  | Data #N[7:0] |  |
| Polynom is 0xEA | **CRC** [7:0] |  |
|  | EOF | 0xA5 |

Table 57- Register read reply

* Min burst is 1
* Max burst is 256 limited by the size of the RAM in the TX path

The read transaction is different from the write transaction and has more units participating in it. A read request is sent to the TX path which reads the data from the client and sends the data back to host via UART line. The following stages are made:

1 – Stages 1 to 9 of the write transaction are made with the following exceptions:

* WM1 (rx\_path) writes to WS2 (tx\_path).
* The length field is always 00\_0x. The reason is because only one word is written to WS2.
* The data field contains the length of the data to be read.
* The Type field includes both the client that should be read and the tx\_path’s wishbone slave in the following pattern:

Type[3 to 0] = “0010” which is WS2 on the tx\_path

Type[7 to 5] = <type of client that will be read>

2 – WS2 sends the data received to the message encoder in the tx\_path.

# 5 SIMULATIONS

## 5.1 part a 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:



Table -Part a simulations

General 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)

## 5.1.1 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.

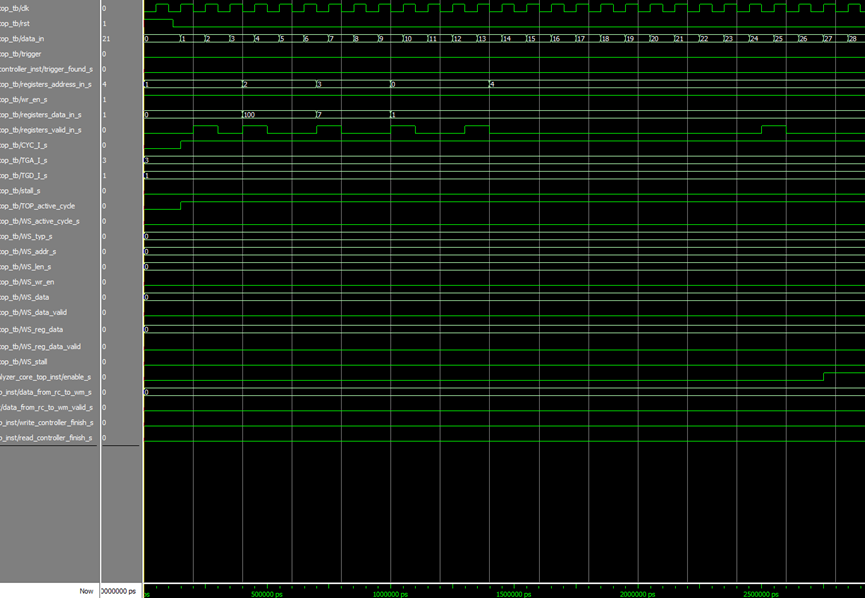


Figure - Part a simulations (1)

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

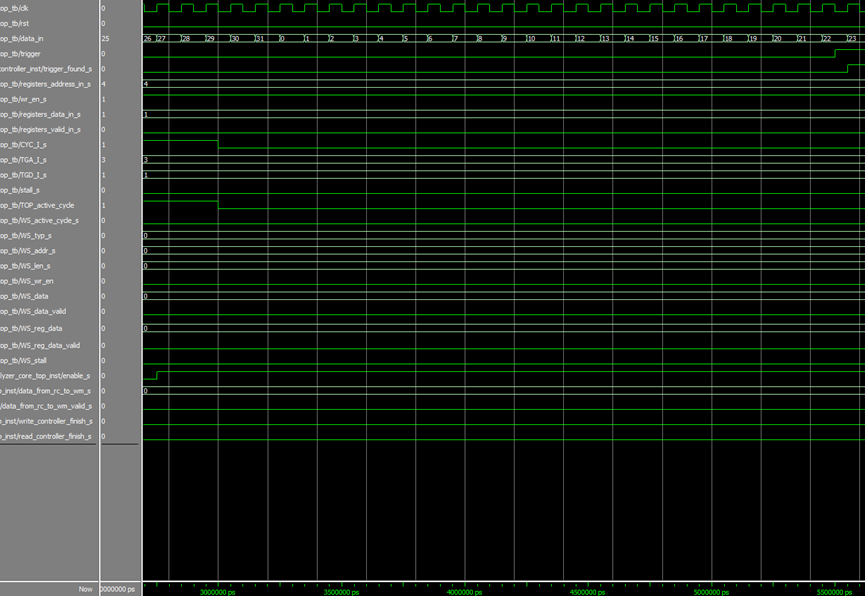


Figure - Part a simulations (2)

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

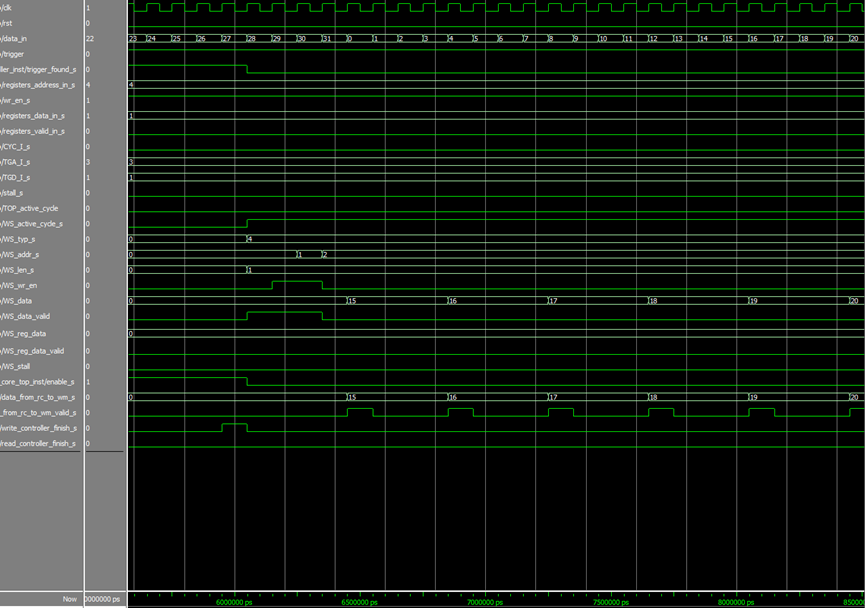


Figure - Part a simulations (3)

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

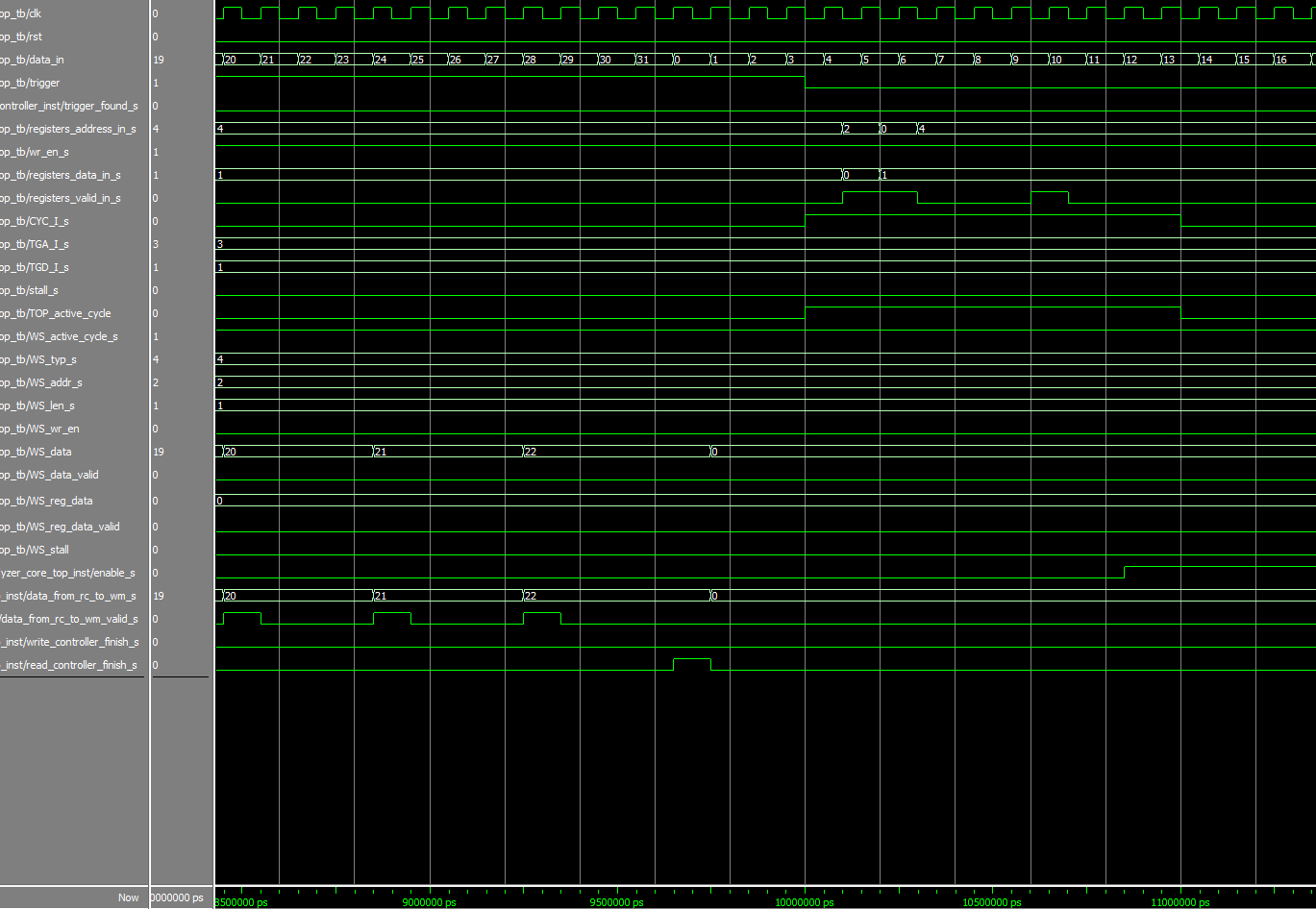


Figure - Part a simulations (4)

## 5.2 Part b simulations

We made two different sets of simulations:

* In the first set we created a Test Bunch to the whole system, and simulated different scenes and cases that the system could get.
* The second set was simulated a read and write requests from all of the registers in the system.

First set:

* At first we made a manual simulations to the core in order to check functionality.
* Afterwards, we built a top test bunch in order to check the entire system.

Test plan first set:

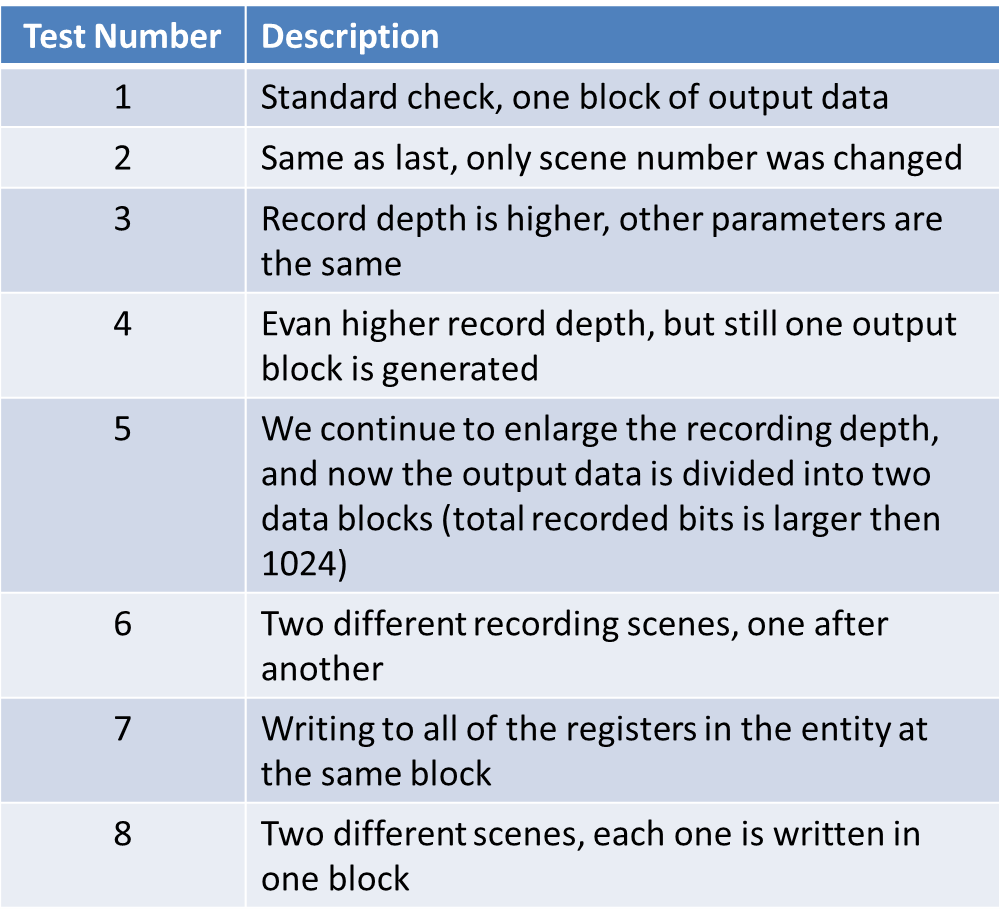


Table 59- Part b simulations

General Description

1. Recording width is 8,recording depth is 16. Chosen scene type is 4, trigger type is downfall, trigger position is 50%.
2. Recording width is 8,recording depth is 16. Chosen scene type is 3, trigger type is downfall, trigger position is 50%.
3. Recording width is 8,recording depth is 32. Chosen scene type is 3, trigger type is downfall, trigger position is 50%.
4. Recording width is 8,recording depth is 64. Chosen scene type is 3, trigger type is downfall, trigger position is 50%.
5. We enlarge the depth of the data recorded, so it will surpass the maximal depth for one packet of the data package . The system will divide the data into couple of data packets. Scene type is 5, trigger type is downfall, trigger position is 0.
6. In this test we conduct two separate data records- at first we record with scene type 5, trigger type is downfall, trigger position is 0. Afterwards we enter another check with scene type 3,trigger type is high, trigger position is 50%.
7. We record one scene, and use one data package to write to all the registers of single entity, i.e. for the signal generator, which has two registers, we perform in the first data package writing for both of them. For the core, which has five registers, we write them all in the second data package. In this test scene type is 5, trigger type is downfall, trigger position is 0.
8. We record two scenes, each written similar to rhe previous simulation. We use one data package to write to all the registers of a single entity. , i.e. for the signal generator, which has two registers, we perform in the first data package writing for both of them. For the core, which has five registers, we write them all in the second data package. In the first test scene type is 5, trigger type is downfall, trigger position is 0. In the second test scene type is 3, trigger type is downfall, trigger position is 10.

* For more information: <http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/TOP/SIMULATIONS/>

Second set:

* The same Test Bunch was used in both sets.
* A read and write scenes were made for all of the registers in order to check their functionality.

Test plan second set:

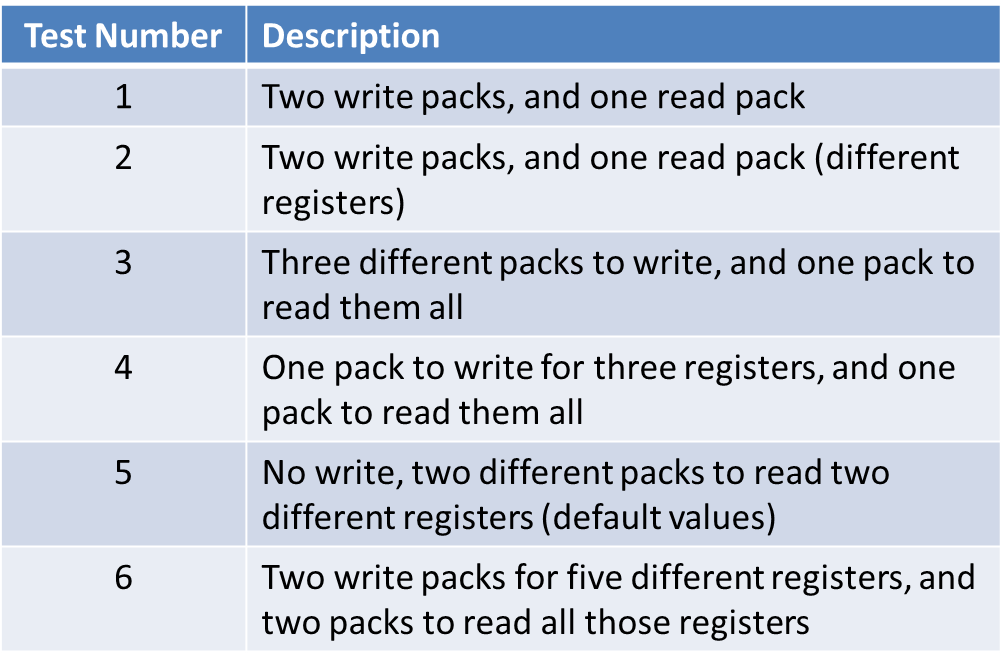


Table 60- Part b simulations

General Description

1. In this simulation we use two data packets in order to write to two registers in the core: to register 1 the value 6, and to register 4 the value 7. Afterwards we use another data packet in order to read from register 4.
2. We use two data packets in order to write the value 6 to register 0 in the core, and to register 2 in the signal generator the value 7. Afterwards we use data package to read the value of register 2.
3. In this simulation we send data in three different packages to registers 2, 3, 4 in the core. Afterwards we read using one data package the data from the three registers.

In order to read the data we send the address of register 2 with the value 3, so the system starts reading from the second register through three sequential words, i.e. three sequential registers are being read.

1. This simulation is similar to the previous one, only we use one data packet to write to the same three registers in the core the values in the order: register 2-value 6, register 3-value 7, register 4-value 8. Afterwards we use one data packet in order to read the values of the registers, similar to the previous simulation.
2. We use two data packets in order to read data. The first read data from the core, and we read the default value of register 3. The second packet reads the default values of the two registers in the signal generator entity. We also changed the value of enable\_polarity\_g, so that register 2 had the value 1.
3. We write data to registers 2, 3, 4 in the core in the first data packet. In the second data packet we write values to registers 1, 2 in the signal generator.

Afterwards in the third data packet we read the values from registers 1,2,3,4 in the core(register 1 has default value). Finally in the fourth data packet we read the values of registers 1,2 in the signal generator.

* For more information: <http://moran-zvika-project.googlecode.com/svn/trunk/Documentation/TOP/Registers%20Simulations/>

# Synthesis

We made initial synthesis.

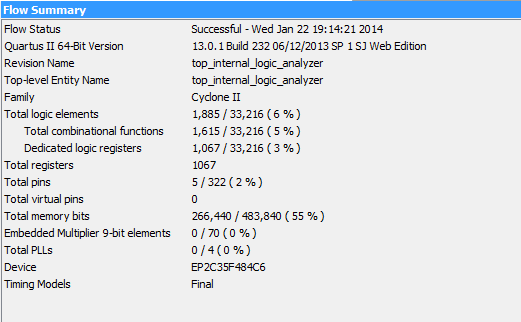


Figure - Quartus Synthesis results

The maximum frequency is 78.03MHZ.

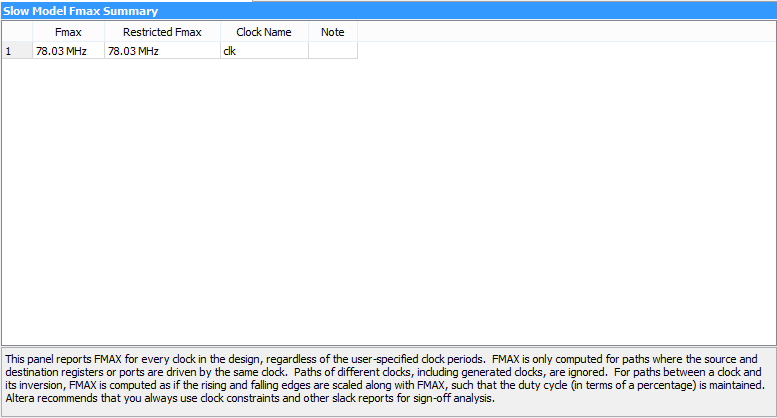


Figure 47- Maximal frequency

We made initial debugging, during which problems occurred and was fixed.

RTL scheme of the system:

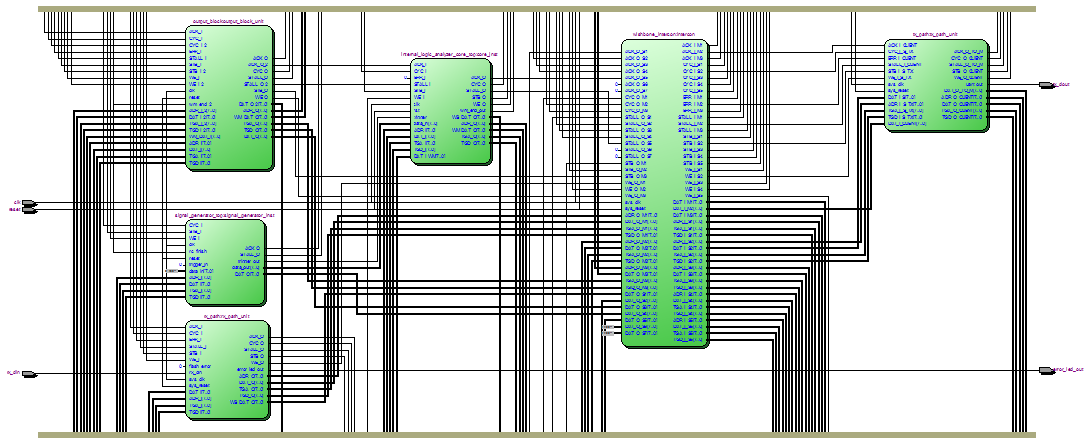


Figure - RTL scheme of the system

# PROBLEMS AND SOLUTIONS

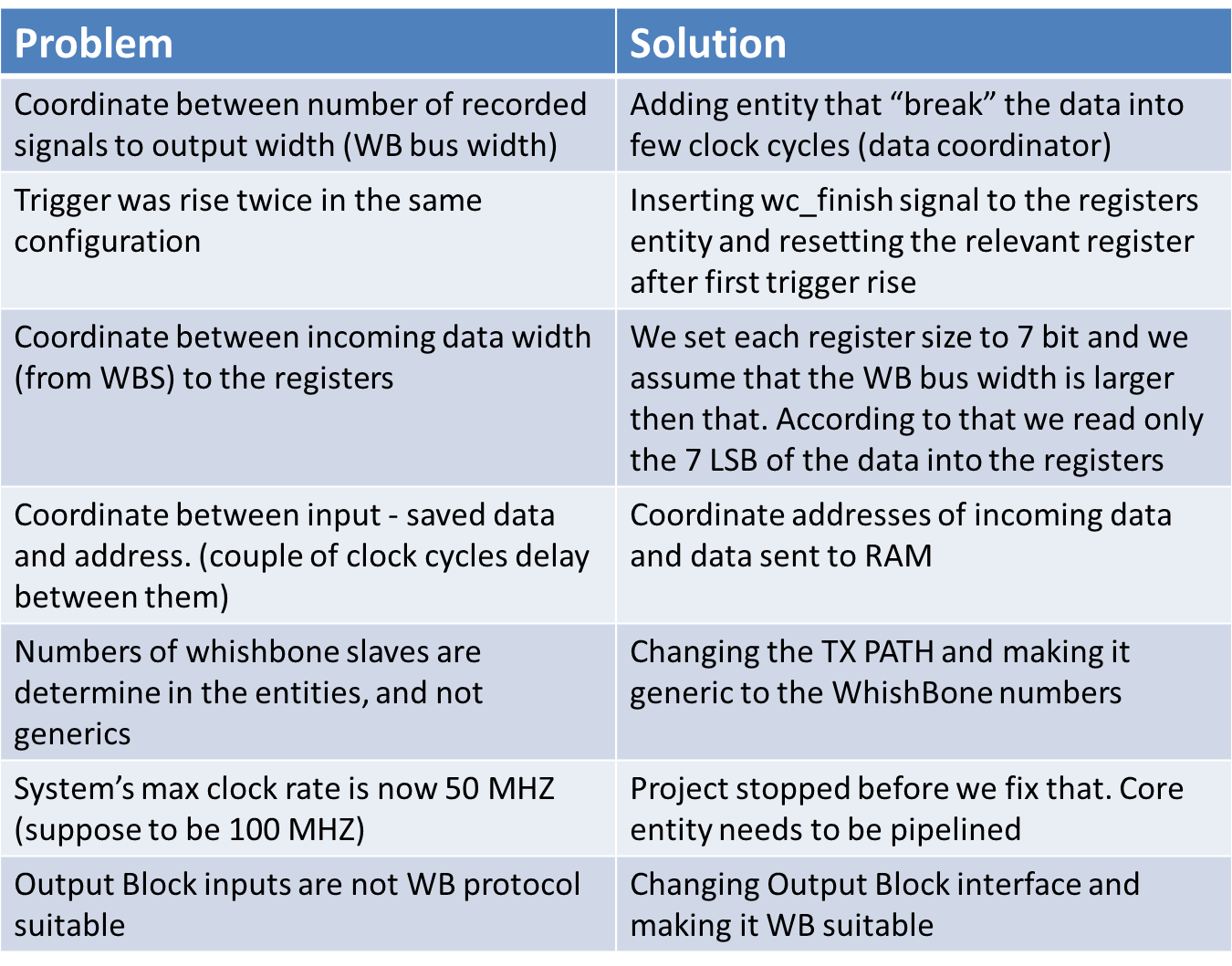


Table - Problems & solutions

## 7.1 Examples

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

# 

Table - 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.

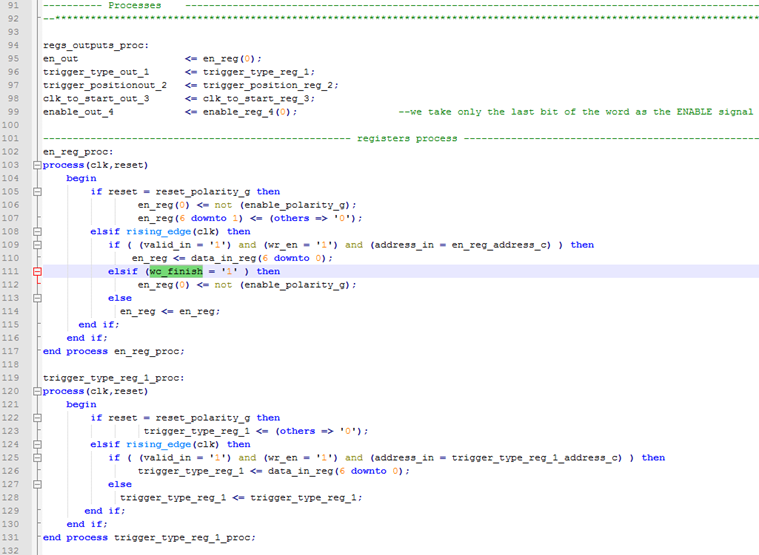


Figure - Code solution

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

Problem- the two widths don’t match.

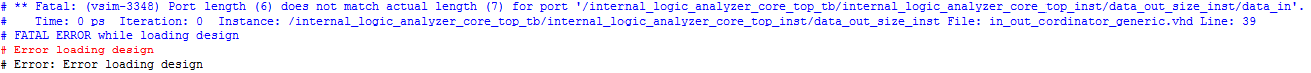


Figure - Second problem

Our solution- adding an entity that coordinates between them.

# 

Figure - Second solution

Third example: Output Block and Core interfaces are not match.



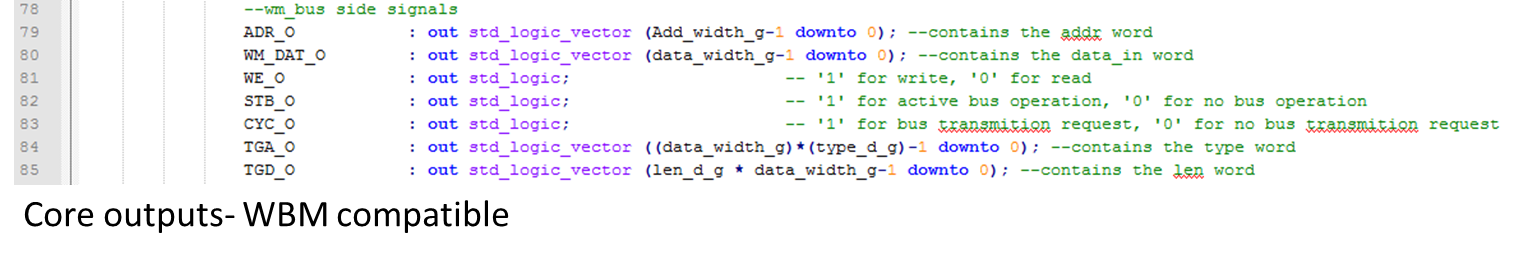


Figure 52- Third problem

Our solution- Making changes in the output Block inputs and making it compatible to WB protocol .

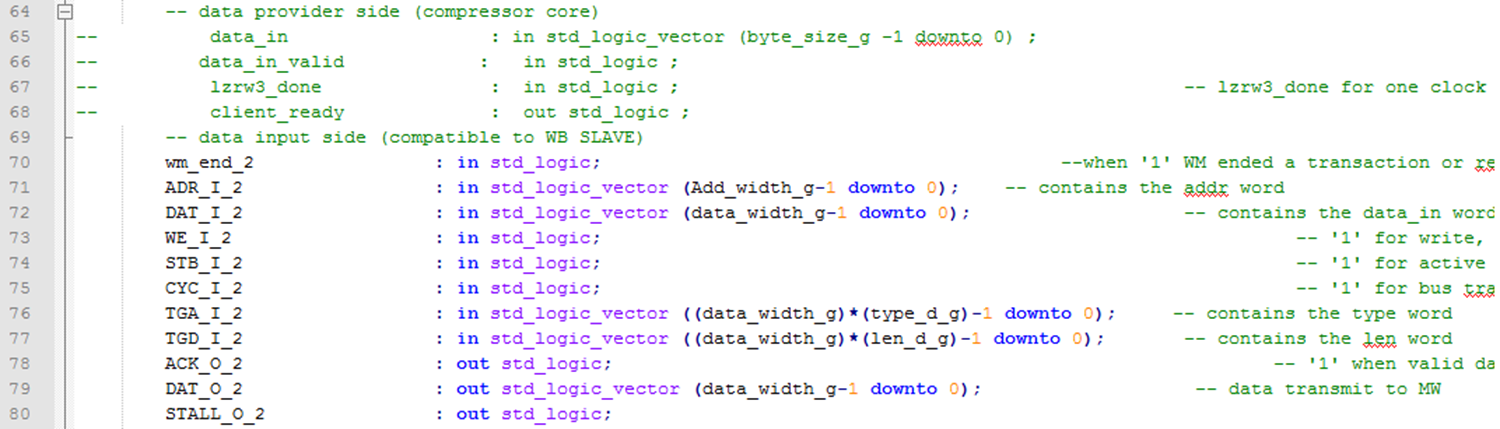


Figure -Third problem solution

# 8 WORKING METHODS

## 

Figure - SVN

## 8.1 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:

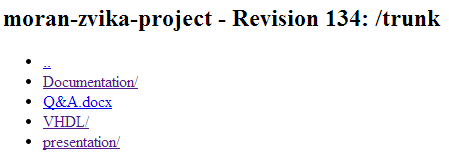


Figure - SVN snapshot

## 8.2 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/>

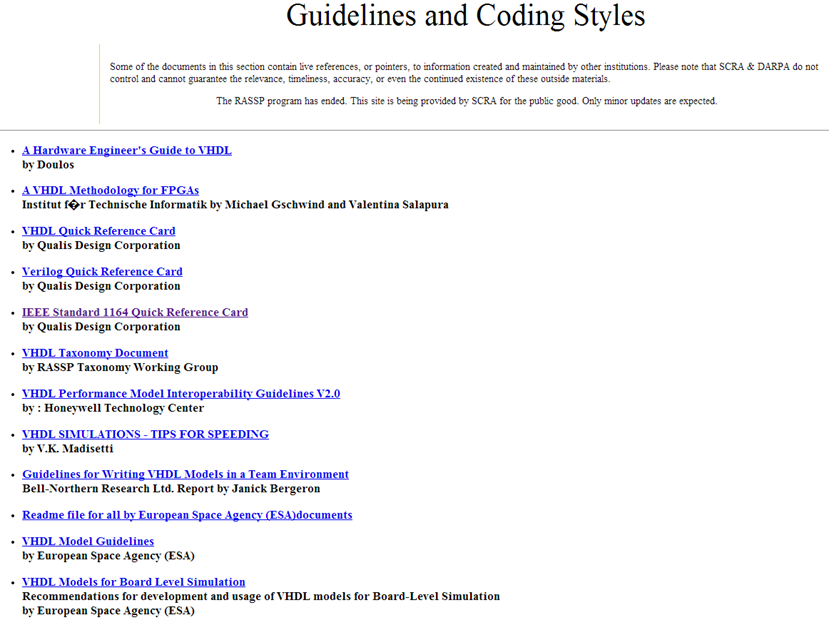


Figure - Coding guidelines

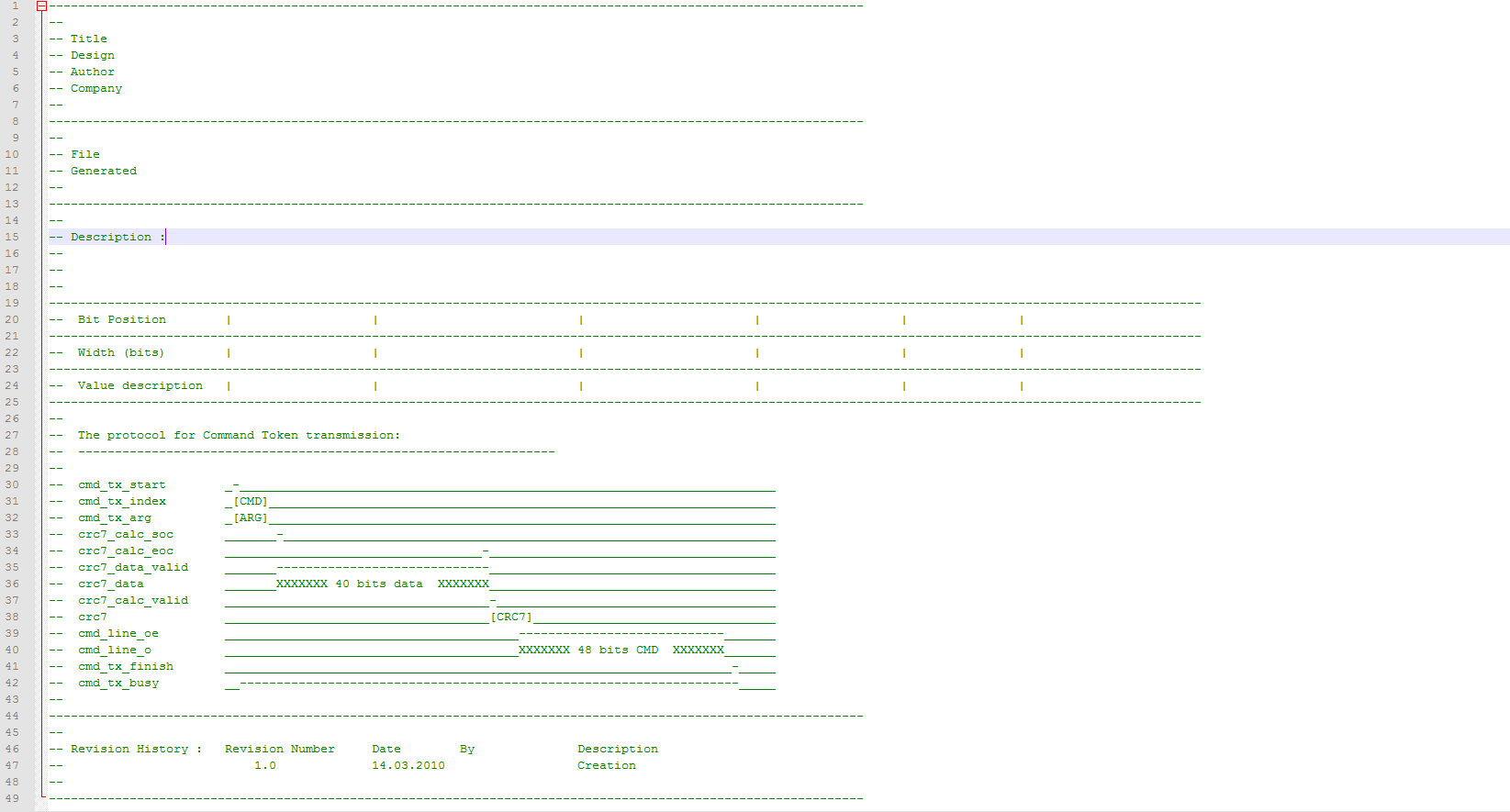


Figure - Entity template

## 8.3 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.

## 8.4 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/>

# 9 summary

## 9.1 Project Usage

The project is aimed to be an easy, comfortable and efficient debugging tool for the FPGA, which is independent in the manufacturer. Therefor the project can be used by anyone uses the FPGA, for any purpose.

These cards are being used, among all, by electrical circuit's planner, chips designers, and in teaching labs. And so each of them is a potential user of this project.

## 9.2 Conclusions

The project included integrating many different entities using several communication protocols. That fact made the working process complex. What made it easier to solve the problems we have come across was a very organized working methods including a well detailed documentation.

# 10 ABBREVIATIONS

WC – Write Controller

RC – Read Controller

CLK – Clock

RST – Reset

UART – Universal Asynchronous Receiver Transmitter

RAM – Random Access Memory

FSM - final state machine

MHz – Mega Hz

WBS – Wishbone slave

WBM – Wishbone master

# 11 Appendix

* Project in SVN:

<http://moran-zvika-project.googlecode.com/svn/>

* Final B video:

<http://youtu.be/QlNaVDeh94A>