****

**Technion**

*Electrical Engineering Department*

High Speed Digital System Lab

**FPGA Setting**

**Using FLASH**

**Project Documentation**

**Students: Dor Obstbaum**

**Kami Elbaz**

**Advisor: Moshe Porian**

**Table of contents**

[**Scope** 4](#_Toc308900957)

[**Abbreviations** 4](#_Toc308900958)

[**General Description** 4](#_Toc308900959)

[**Block Details** 6](#_Toc308900960)

[***rx path*** 6](#_Toc308900961)

[***Config Control Block*** 17](#_Toc308900962)

[***Flash Control*** 18](#_Toc308900963)

[***Clk & Reset*** 19](#_Toc308900964)

[***Filter*** 20](#_Toc308900965)

[***Wait - client*** 21](#_Toc308900966)

[***LED – client*** 23](#_Toc308900967)

[***Display – client*** 28](#_Toc308900968)

[***Wishbone Blocks and protocol*** 28](#_Toc308900969)

[**Wishbone Protocol** 28](#_Toc308900970)

[**Wishbone Intercon** 30](#_Toc308900971)

[**Wishbone Master** 30](#_Toc308900972)

[**Wishbone Slave** 32](#_Toc308900973)

[***Data transfer*** 34](#_Toc308900974)

[***Write transaction*** 34](#_Toc308900975)

[Write transaction example 34](#_Toc308900976)

[***Read transaction*** 35](#_Toc308900977)

[***FLASH Memory*** 36](#_Toc308900978)

[***Testability*** 37](#_Toc308900979)

[Top level testing was done using the configuration seen in the figure: 37](#_Toc308900980)

[Messages to the FPGA 38](#_Toc308900981)

[Registers Write Message 38](#_Toc308900982)

**Table of figures**

[Figure 1 - Top Architecture 4](#_Toc308632023)

[Figure 2 - rx path 6](#_Toc308632024)

[Figure 3 - MDWM FSM 10](#_Toc308632025)

[Figure 4 - TX path 12](#_Toc308632026)

[Figure 5 - WTME FSM 14](#_Toc308632027)

[Figure 7 - clk& reset 18](#_Toc308632028)

[Figure 6 - message pack structure 18](file:///C:\Users\dobstbau\Dropbox\Project\DOC\Project%20document\Flash_Setting_Project_Document.docx#_Toc308632029)

[Figure 8 - Filter 19](#_Toc308632030)

[Figure 9 – LED client 23](#_Toc308632031)

[Figure 10 - Led waveform 25](file:///C:\Users\dobstbau\Dropbox\Project\DOC\Project%20document\Flash_Setting_Project_Document.docx#_Toc308632032)

[Figure 11 - Led unit 27](#_Toc308632033)

[Figure 12 - Wishbone cycle waveform example 29](#_Toc308632034)

[Figure 13 - Wishbone Intercon 30](#_Toc308632035)

[Figure 14 - Wishbone Master FSM 31](#_Toc308632036)

[Figure 15 - Wishbone Slave FSM 32](#_Toc308632037)

[Figure 16 - GUI 33](#_Toc308632038)

[Figure 17 -DATA PROCCESSED IN THE RX PATH 35](#_Toc308632039)

[Figure 18 - Data transfer from rx\_path to led client 35](#_Toc308632040)

[Figure 19 - Flash schematic symbol and schematics 36](file:///C:\Users\dobstbau\Dropbox\Project\DOC\Project%20document\Flash_Setting_Project_Document.docx#_Toc308632041)

**Table of Tables**

[Table 1 - RX path interface 7](#_Toc308546004)

[Table 2 - Uart rx interface 7](#_Toc308546005)

[Table 3 - Uart rx generics 8](#_Toc308546006)

[Table 4 - mp\_dec interface 8](#_Toc308546007)

[Table 5 - mp\_dec generics 9](#_Toc308546008)

[Table 6 - RAM interface 9](#_Toc308546009)

[Table 7 - RAM generics 9](#_Toc308546010)

[Table 8 - CRC interface 9](#_Toc308546011)

[Table 9 - CRC generics 9](#_Toc308546012)

[Table 10 - MDWM interface 10](#_Toc308546013)

[Table 11 - MDWM generics 11](#_Toc308546014)

[Table 12 - Error Register interface 11](#_Toc308546015)

[Table 13 - Error register generics 11](#_Toc308546016)

[Table 14 - Led client interface 17](#_Toc308546017)

[Table 15 - Led client generics 18](#_Toc308546018)

[Table 16 - Led Registers interface 19](#_Toc308546019)

[Table 17 - Led Registers generics 19](#_Toc308546020)

[Table 19 - Address Advancer generics 20](#_Toc308546021)

[Table 20 - Led interface 20](#_Toc308546022)

[Table 21 - Led generics 21](#_Toc308546023)

[Table 22 - Timer interface 21](#_Toc308546024)

[Table 23 - Timer generics 21](#_Toc308546025)

[Table 24 - Wishbone Master interface 25](#_Toc308546026)

[Table 25 - Wishbone Master generics 26](#_Toc308546027)

# **Scope**

This document aims to describe the requirements from the FPGA setting using FLASH project, explain each block's and each signal's function, and to explain how to use the projects blocks.

# **Abbreviations**

CCB – Config Control Block

RAM - Random Access Memory

TX – Transmission

RX – Receive

SOF – Start of Frame

EOF – End of Frame

CRC – Cyclic Redundancy Check

FCB - Flash Control Block

IWB - Intercon Wishbone Block

EOD – End of Data

CFI - Common Flash Interface

FSM – Final State Machine

MDWM – Message Decoder to Wishbone Master

WTME – Wishbone To Message Encoder

WM – Wishbone Master

WS – Wishbone Slave

VESA - Video Electronics Standards Association

ROI - Region Of Interest

# **General Description**

The system would set an FPGA component by data stored in FLASH when power is turned on. This would allow the same FPGA with the same burning file to be handy in a variety of applications. Data stored in FLASH memory would configure registers that set the mode of operation of units. Data reading and writing to FLASH would be done with a MATLAB based GUI. The GUI would also have debugging capabilities.

**Top Architecture**

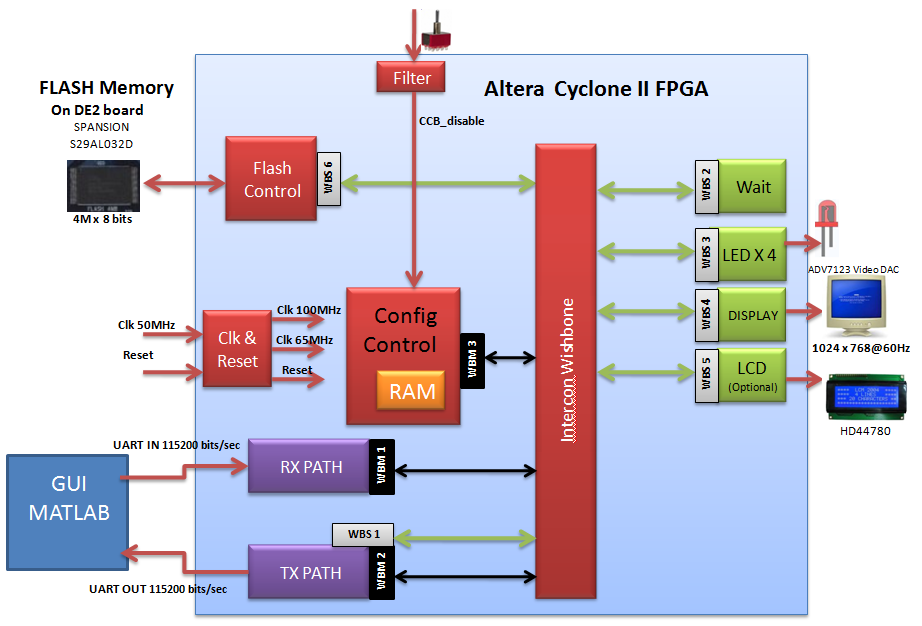


Figure 1 - Top Architecture

**Project Requirements**

1. Configuration of clients in the FPGA would be determined by data stored and read from FLASH.
2. 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).
3. System clock will be 100MHz. will be generated by a PLL from a 50MHZ oscillator onboard the DE2 card.
4. The System would enable a Host to read and write data to FLASH and clients through FPGA.
5. The FPGA will communicate with its PC host via UART protocol with baud rate of 115,200 bits/sec.
6. 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 can be added after the data bits frame, include 3 options:
      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.
7. 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 – one 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.
8. Internal Communication using Wishbone protocol.

* Bus width: 8 bits
* Units with wishbone master interfaces: RX path, TX path, CCB
* Units with wishbone slave interfaces: TX path, Wait, LED, VIDEO, LCD, Flash control
* The transactions used are:
* Read single
* Write single
* Read burst
* Write burst

1. The FLASH memory being used is 4MB Spansion.
2. Four client can be recruited to active (as slaves):

* Wait: this client roll is to occupy the bus for given time.
* Led: this client will be use to operate 4 leds at different frequencies.
* LCD: this client will control an external LCD screen.
* Display: this client will control an external screen according to configuration stored in the registers.

1. The reset of the system is filtered, and then sampled in the clock rate.
2. Signals connected to DE2 switches would be filtered in order to avoid an unstable signal in the system. The signal CCB\_disable would be filtered in the Filter block. The signal Reset\_in would be filtered in the CLK&Reset block.
3. The FLASH size is 4M. In the initial of the system the FLASH will transmit 256 bytes to the CCB, where the data will be saved.
4. The data base in the flash will begin with type and then address, length, data. The fields meaning are as mentioned on section 7. The last pack would contain EOD (not EOF!) in the Type field. When the CCB state machine reads the EOD value, client configuration will end. Data transfer is explained in detail in the “Data Transfer” section in this document.

EOD: “11110000”

Example for data pack structure:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Len | address | data | data |
| Led | 2 | 0 | 0 | 1 |

The following pack configures one led turned off and one turned on in the led block. The values in this example are listed as decimal.

# **Block Details**

## rx path

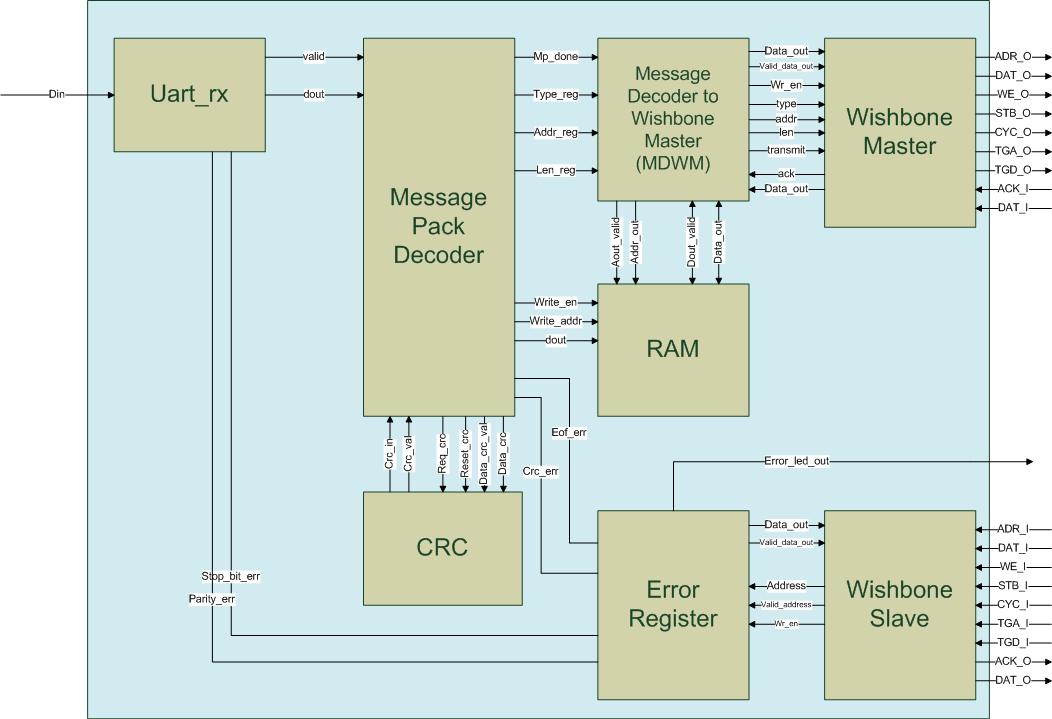


Figure 2 - rx path

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 |
| 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 |
| 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 1 - RX path interface

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

The RX path contains the following units:

**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 then ‘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 2 - Uart rx interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 3 - Uart rx generics

**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 mdwm (message decoder to 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 recieved |
| 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 caluclated 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 4 - mp\_dec interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 - Recieved length is decreased by 1 ,to save 1 bit --  FALSE - Recieved 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 |

Table 5 - mp\_dec generics

**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 6 - RAM interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 7 - RAM generics

**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 x^8 polynomial calculation.

|  |  |  |  |
| --- | --- | --- | --- |
| **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 8 - CRC interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 9 - CRC generics

**Message Decoder to Wishbone Master (mdwm) -** The MDWM operates the rx\_path’s wishbone master in order to send the data received to the intended client. It is the wishbone’s masters director, telling it when to start a transition, and when to end one and what data would be transferred. The mdwm is awakened when the mp\_done signal is asserted by the mp\_dec unit. Then the mdwm reads the TYPE, ADDRESS, LENGTH values and directs it to the wishbone master. After this the unit reads data from the RAM according to the LEN value it has. Every wishbone transaction is ended when ACK\_I signal is asserted. The units operation is according to the following FSM:

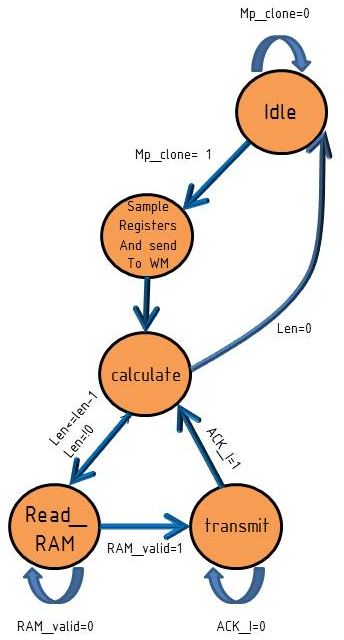


Figure 3 - MDWM FSM

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| clk | input | 1 | clock |
| rst | input | 1 | reset |
| mp\_done | input | 1 | Message Pack Decoder is done when '1' is recieved |
| type\_in | input | type\_d\_g \* data\_width\_g | type is the client which the data is directed to |
| addr\_in |  | addr\_d\_g \* data\_width\_g | the address in the client that the information will be written to |
| len\_in | input | len\_d\_g \* data\_width\_g | length of the data (in words) |
| valid\_RAM\_data | input | 1 | '1' is recieved when the data from RAM is valid |
| RAM\_data | input | data\_width\_g | data recieved from RAM |
| valid\_RAM\_address | input | 1 | '1' is when mdwm requests to read from RAM |
| RAM\_address | input | data\_width\_g | address in RAM to be read |
| data\_out | output | data\_width\_g | data word that will be written to Wishbone client |
| valid\_data\_out | output | 1 | data word is valid |
| wr\_en | output | 1 | '1' for write request, '0' for read request |
| transmit | output | 1 | '1' for wishbone bus transmition request |
| type\_out | output | type\_d\_g \* data\_width\_g | type is the client which the data is directed to |
| addr\_out | output | addr\_d\_g \* data\_width\_g | the address in the client that the information will be written to |
| len\_out | output | len\_d\_g \* data\_width\_g | length of the data (in words) |
| ack | input | 1 | ack is recieved from wishbone master when a successfull transaction ends |

Table 10 - MDWM interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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] |
| type\_d\_g | positive | 1 | Type Depth |
| addr\_d\_g | positive | 3 | Address Depth |
| len\_d\_g | positive | 1 | Length Depth |

Table 11 - MDWM generics

**Wishbone Master -** see wishbone units.

**Wishbone Slave -** see wishbone units.

**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. 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..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 12 - Error Register interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 |

Table 13 - Error register generics

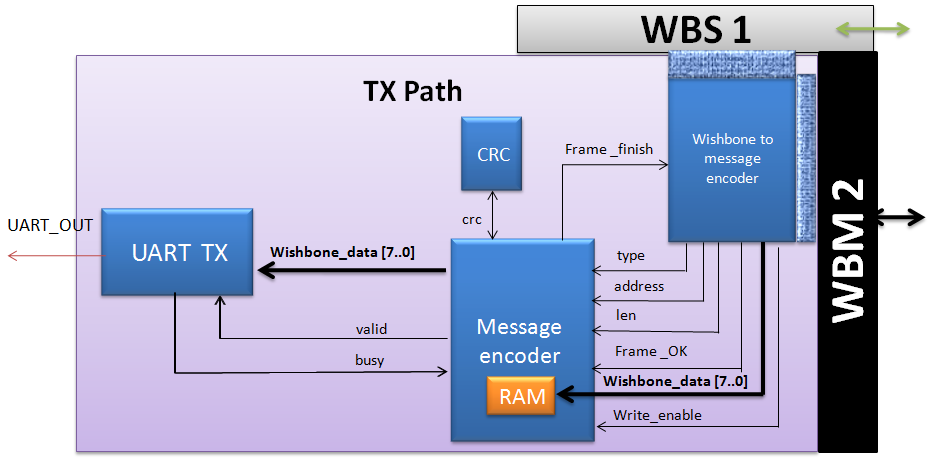
TX path 

Figure 4 - 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 WTME receive and save the data request and the slave 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** |
| Sys\_clk | In | 1 | clock |
| Sys\_reset | In | 1 | Reset. Reset polarity will be set according to the generic parameter 'reset\_polarity\_g' |
| DAT\_I\_S | In | data\_width\_g | The **data** input receiving to the TX slave from the RX master |
| ADR\_I\_S\_TX | In | (addr\_d\_g)\*(data\_width\_g) | The **address**  input receiving to the TX slave from the RX master |
| TGA\_I\_S\_TX | In | (type\_d\_g)\*(data\_width\_g) | The **type** input receiving to the TX slave from the RX master |
| TGD\_I\_S\_TX | In | (len\_d\_g)\*(data\_width\_g) | The **length** input receiving to the TX slave from the RX master |
| WE\_I\_S\_TX | In | write\_en\_polarity\_g | The wishbone Write enable signal transmitted to the TX slave, this signal will be here at ‘0’ (the RX make a write transaction to the TX ) |
| STB\_I\_S\_TX | In | 1 | The wishbone **STB** signal transmitted to the TX slave |
| CYC\_I\_TX | In | 1 | The wishbone **CYC** signal transmitted to the TX slave |
| ACK\_O\_TO\_M | out | 1 | The wishbone **acknowledge**  signal transmitted to the RX master |
| DAT\_O\_TO\_M | Out | data\_width\_g | The data output transferring to the RX master - NOT USED |
| DAT\_I\_CLIENT | in | data\_width\_g | The data input that was read from the CLIENT / FLASH |
| ACK\_I\_CLIENT | in | 1 | The wishbone acknowledge signal from the CLIENT / FLASH |
| ADR\_O\_CLIENT | Out | (addr\_d\_g)\*(data\_width\_g) | The address to begin read in the CLIENT / FLASH |

Table 14 - TX Path interface

|  |  |  |  |
| --- | --- | --- | --- |
| **Generic Parameter** | **type** | **Default 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 |
| valid\_in\_polarity\_g | Std\_logic | ‘1’ | ‘1’  Value is valid, ‘0’ value isn’t valid |
| write\_en\_polarity\_g | Std\_logic | ‘1’ | defines the polarity which the write\_en is active on: '1' is active high |
| 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 (2^10 = 1024 addresses) |

Table 15 - TX Path generics

**The TX Path contains the following blocks:**

**Wishbone To Message Encoder (WTME):**

The WTME is activated through the RX master in request to read data from the FLASH or from one CLIENT. The RX master transfer the type, address and length of the request the WTME save them in register, upraise acknowledge and activate his own master.

The TX master turn to the FLASH / CLIENT slave and request to read the data. The data is transmitted to the WTME byte by byte (total of LENGTH byte). After all the data was transmitted ‘frame\_ok’ is raise to inform the message encoder to begin to read the data (from the RAM).

WTME FSM:

Figure 5 - WTME FSM

**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 ‘frame finish’ that signal to the WTME while the present transaction is still running. While frame finish is ‘0’ the WTME 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 | Clock |
| Rst | In | Reset. Reset polarity will be set according to the generic parameter 'reset\_polarity\_g' |
| Fifo\_full | In | FIFO is full, and cannot receive more data from MP Encoder |
| Reg\_ready | In | Input Type, Address and Data Length registers values are ready |
| Type\_reg [width\_g \* type\_d\_g-1..0] | In | Input Type value. Will be valid together with the *reg\_ready* signal |
| Addr\_reg [width\_g \* addr\_d\_g-1..0] | In | Input Address value. Will be valid together with the *reg\_ready* signal |
| Len\_reg [width\_g \* len\_d\_g-1..0] | In | Input Data Length value. Will be valid together with the *reg\_ready* signal |
| Crc\_in [width\_g \* crc\_d\_g-1..0] | In | Calculated CRC value from Checksum block |
| Crc\_in\_val | In | Calculated CRC value from Checksum block is valid |
| Din [width\_g-1..0] | In | Input data (payload), from RAM |
| Din\_valid | In | Input data (payload), from RAM is valid |
| Mp\_done | Out | Message Pack has been successfully transmitted. This flag will be raised together with the EOF output data |
| Dout [width\_g-1..0] | Out | Output data, to the FIFO |
| Dout\_valid | Out | Output data, to the FIFO, is valid |
| Data\_crc\_val | Out | Data to the CRC block is valid |
| Data\_crc [width\_g-1..0] | Out | Data to the CRC, for CRC calculation |
| Reset\_crc | Out | Reset the CRC value to its default value |
| Req\_crc | Out | Request for calculated CRC value |
| Read\_addr\_en | Out | Address to RAM is valid |
| Read\_addr [width\_g \* len\_d\_g-1..0] | Out | Address to RAM |

Table 16 - message encoder signals

| Generic Parameter | type | Default 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 17 - message encoder generics

**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.

**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 18 - FIFO signals

| Generic Parameter | type | Default 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 19 - FIFO Generics

**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 |  | FIFO Ready to transmitte 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 20 - UART\_TX signals

See generics list at the RX Path.

**Wishbone Master -** see wishbone units.

**Wishbone Slave -** see wishbone units.

## Config Control Block

General Description

1. The Config Control Block (CCB) has a wishbone master interface (WBM 3).
2. When power is turned on the CCB would initiate a transaction to Flash control (WBS 6), causing the data stored in the Flash to be transferred by the wishbone Bus to the CCB.
3. The data from the Flash would be stored in an internal RAM in the CCB, before it is transferred to the clients. RAM size is 256 rows x 8 bits.
4. When all the data from Flash is stored in the RAM, it will be sent to the clients. The data transfer would include the following fields:

* Type – which client is being accessed – 1byte
* Len – length of data in bytes – 2 bytes
* Add – address of registers in clients – 1 byte
* Data – the data being transferred – [len] x bytes
* EOD – end of data “11110000” - this data counts as a Type. When the CCB reads this Type data transfer stops– 1 byte

1. An option to disable CCB data transfer by user would be implemented by CCB\_disable signal connected to the **Filter** unit.
2. Data pack structure stored in the CCB RAM has the following format:

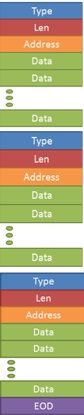


Figure 6 – Pack stored in CCB RAM

## Flash Control

General Description

1. The Flash Control Block (FCB) has a wishbone slave interface (WBS 6).
2. The FCB would consist of two main blocks:

* Flash write/read FSM
* Flash Interface

1. The FCB would initiate communication with Flash by request from CCB. Data from Flash would be transferred threw the bus to the CCB. Data transfer would be done also for debug purposes.
2. **Flash details:**
3. The Flash memory contains the configuration that the system uses.
4. The Flash communicates with the system via the FCB.
5. System configuration is saved to the Flash when system is in debug mode.
6. The configuration data is transferred to the system when power is turned on.
7. Flash memory size: 4MB

## Clk & Reset

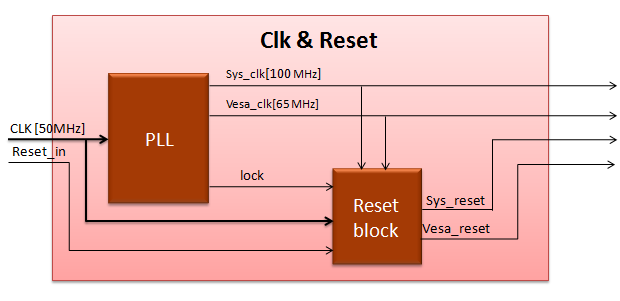


Figure 7 - clk& reset

General Description

1. The Clk & Reset unit would generate the clock and reset signals for the system.
2. Inputs:

* 50MHz clock from DE2.
* Reset signal activated by user.

1. Outputs:

* *Sys\_clk* – 100MHz clock. This is the clock the system would use.
* *Vesa\_clk* – 65MHz clock. This clock would be directed to the Display unit.
* *sys\_reset* - The signal would reset the blocks in the system.
* *Vesa\_reset – the signal would reset the Display unit.*

1. The Clk & Reset block would consist of two sub – blocks:

* **PLL** – This block would receive a 50MHz clock and will generate three signals:

1 – *sys\_clk* (100MHz)

2 *– vesa\_clk* (65MHz)

3 – *lock* – a signal that would be asserted when the clock signals are ready and stable.

* **reset\_blk** – This block would receive the internal *lock* signal and an external *reset\_in*  signal. When both signals are asserted the unit would assert the *sys\_rst and vesa\_rst* signals that would reset the system blocks.

1. The PLL block would be generated by Quartus Megawizard.

Signals

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **Type** | **width (bits)** | **description** |
| clk[50MHz] | Input | 1 | clock from DE2 |
| reset\_in | Input | 1 | reset from switch on DE2 |
| sys\_clk | Output | 1 | clock for system blocks |
| vesa\_clk | Output | 1 | clock for VESA block |
| sys\_reset | Output | 1 | reset for system blocks |
| vesa\_reset | Output | 1 | reset for VESA block |

## Filter

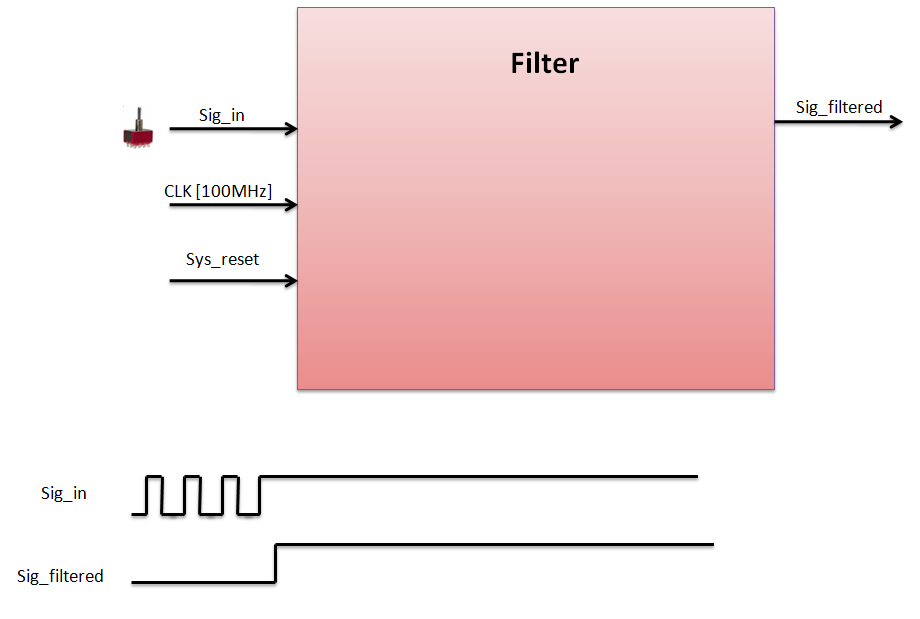


Figure 8 - Filter

General Description

1. The Filter block receives the CCB\_disable signal from user (a switch on DE2 board) in order to disable the CCB for debug purposes. This makes it possible to read and write data to Flash by the GUI without accessing the Flash on initialization.
2. The CCB\_disable is generated using a DE2 switch, therefore it could be unstable. The Filter block samples the signal a given amount of times defined by its generics.
3. If CCB\_disable = ‘1’ the system would function normally – and data would be read from FLASH to CCB RAM in order to start client configuration.
4. If CCB\_disable = ‘0’ the CCB is disabled and data would not be read from FLASH, meaning clients would not be configured by that data.
5. Sig\_in could be sampled once or twice (defined by a generic). Sampling the signals twice helps to avoid metastability.
6. Generics:

* sample\_depth\_g – the number of samples needed before the filtered signal (Sig\_filtered) can change from one state to another.
* sig\_reset\_value\_g – the default value of sig\_filtered (0 or 1) before the arrival of enough samples (defined by sample\_depth\_g).
* reset\_activity\_polarity\_g – the value of which resets the Filter block (reset is done by the Sys\_reset signal)
* sample\_twice\_en – defines if sig\_in would be sampled once or twice.

Signals

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **Type** | **width (bits)** | **description** |
| sig\_in | Input | 1 | the signal which needs filtering |
| clk (100MHz) | Input | 1 | system clock |
| sys\_reset | Input | 1 | block reset |
| sig\_filtered | Output | 1 | signal after filtering |

## Wait - client

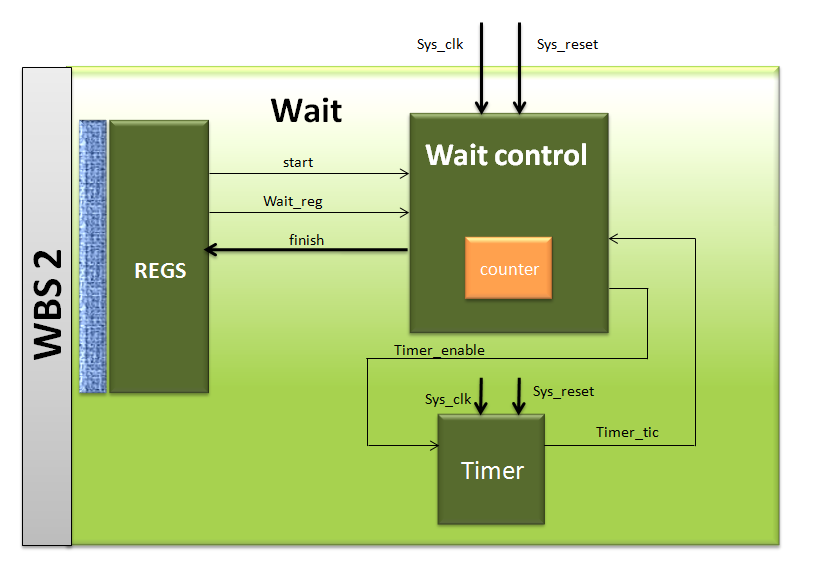


Figure 9 - wait calient

General Description

1. The Wait Block would occupy the bus on demand from master (CCB) for a given amount of time
2. The Wait client has a wishbone slave interface.
3. The counter in the wait control and the timer will calculate the needed time to occupy the bus: The timer will raise the ‘timer\_tic’ signal every time unit (generic) for one cycle time, the counter will count the number of time\_unit tics, when the counting will arrive to ‘wait\_reg’ – the value saved in the register- the wait control will raise ‘finish’ to signify that this transaction is finish.

|  |  |  |
| --- | --- | --- |
| Pin Name | Direction | Description |
| Sys\_clk | in | clock |
| Sys\_reset | In | Reset. Reset polarity will be set according to the generic parameter 'reset\_polarity\_g' |
| Dat\_in | In | Data output from the wishbone slave – NOT USED |
| We\_in | In | The wishbone write enable signal. When ‘1’ write to the unit |
| Stb\_in | In | The wishbone STB signal |
| Cyc\_in | In | The wishbone CYC signal |
| Ack\_output | out | Indicate that the unit finish to wait, and free the BUS |

Table 21 - wait client signals

|  |  |  |  |
| --- | --- | --- | --- |
| Generic Parameter | type | Default Value | Description |
| reset\_activity\_polarity\_g | Std\_logic | ‘1’ | defines reset active polarity: '0' active low, '1' active high |
| block\_enable\_polarity\_g | std\_logic | ‘1’ | defines the active block enable polarity : '0' active low, '1' active high |
| finish\_activity\_polarity\_g | std\_logic | ‘1’ | defines finish active polarity: '0' active low, '1' active high |
| wait\_value\_width\_g | natural | 8 | width of the wait\_reg vector |
| timer\_freq\_g | positive | 100 | timer\_tick will raise for 1 sys\_clk period every timer\_freq\_g. units: [Hz] |
| clk\_freq\_g | positive | 10000 | the clock input to the block. this is the clock used in the system containing the timer unit. units: [Hz] |
| timer\_en\_polarity\_g | std\_logic | ‘1’ | defines the polarity which the timer enable (timer\_en) is active on: '0' active low, '1' active high |

Table 22 - wait client generics

Wait\_client\_ins: This unit will count number of wait\_reg ‘ticks’ from the timer and then will upraise the finish signal.

|  |  |  |
| --- | --- | --- |
| Pin Name | Direction | Description |
| Sys\_clk | input | clock |
| Sys\_reset | input | Reset. Reset polarity will be set according to the generic parameter 'reset\_polarity\_g' |
| enable | input | determines if the wait client is enables |
| wait\_reg | input | The waiting time value recieved from registers |
| finish | output | this signal signify the end of the wait transaction |

Table 23 - wait\_client\_int interface

|  |  |  |  |
| --- | --- | --- | --- |
| Generic Parameter | type | Default Value | Description |
| reset\_activity\_polarity\_g | Std\_logic | 1' | defines reset active polarity: '0' active low, '1' active high |
| block\_enable\_polarity\_g | Std\_logic | 1' | defines the active block enable polarity : '0' active low, '1' active high |
| finish\_activity\_polarity\_g | Std\_logic | 1' | defines finish active polarity: '0' active low, '1' active high |
| wait\_value\_width\_g | natural | 8 | width of the wait\_reg vector |

Table 24 - wait\_client\_int generics

**Timer -** A timer unit is placed inside every Led unit and function as detailed in the Led description.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| timer\_en | input | 1 | determines if the timer is enabled |
| timer\_tick | output | 1 | asserted for 1 cycle every 100,000 cycles |

Table 25 - Timer interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default value** | **description** |
| reset\_polarity\_g | std\_logic | 1 | '0' = Active Low, '1' = Active High |
| timer\_freq\_g | positive | 10 | timer\_tick will raise for 1 sys\_clk period every timer\_freq\_g. units: [Hz] |
| clk\_freq\_g | positive | 100000000 | the clock input to the block. this is the clock used in the system containing the timer unit. units: [Hz] |
| timer\_en\_polarity\_g | std\_logic | 1 | defines the active state of the timer: '0' active low, '1' active high |

Table 26 - Timer generics

## LED – client

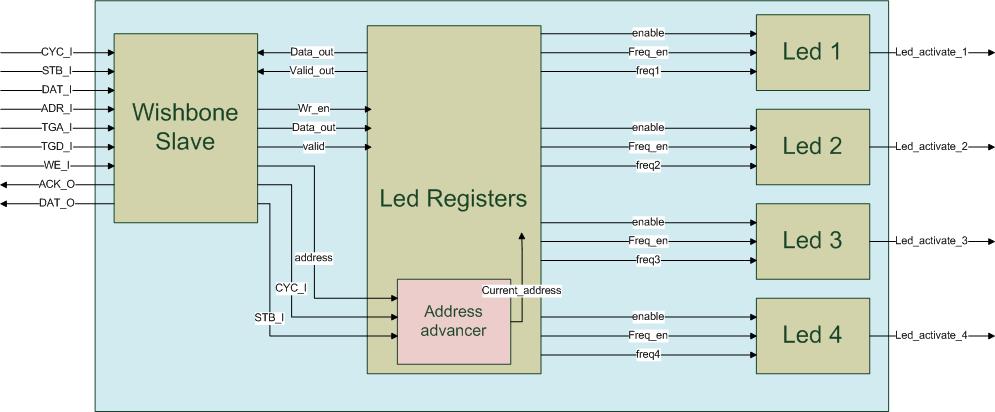


Figure 10 – LED client

General Description

1. The LED block would be used to operate a led at different frequencies or at steady on/off state.
2. The LED client has a wishbone slave interface.
3. The client will contains 4 LEDs, each one of them will have led control and timer.
4. The Led client contains a register block that will hold the data of the LEDs requirement:
5. The Timer in the LED block generates a timer\_tick every 100,000 cycles which are 1 msec.
6. All the information regarding the leds is loaded to the led\_registers block.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| DAT\_I | input | data\_width\_g | data recieved from WS |
| 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 transmited to MW or for successfull write operation |
| DAT\_O | output | data\_width\_g | data transmit to MW |
| led\_activate\_1 | output | 1 | this signals activates the led number 1 |
| led\_activate\_2 | output | 1 | this signals activates the led number 2 |
| led\_activate\_3 | output | 1 | this signals activates the led number 3 |
| led\_activate\_4 | output | 1 | this signals activates the led number 4 |

27

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 |
| en\_reg\_address\_g | natural | 0 | enable register address |
| freq\_reg\_1\_address\_g | natural | 1 | frequency register 1 address |
| freq\_reg\_2\_address\_g | natural | 2 | frequency register 2 address |
| freq\_reg\_3\_address\_g | natural | 3 | frequency register 3 address |
| freq\_reg\_4\_address\_g | natural | 4 | frequency register 4 address |
| active\_state\_polarity\_g | std\_logic | 1 | defines the active state of the led: '0' active low, '1' active high |
| block\_enable\_polarity\_g | std\_logic | 1 | defines the active block enable polarity: '0' active low, '1' active high |
| freq\_enable\_polarity\_g | std\_logic | 1 | defines freq\_en input active state: '0' active low, '1' active high |
| freq\_width\_g | natural | 8 | width of the frequency vector |
| timer\_freq\_g | positive | 1000 | timer\_tick will raise for 1 sys\_clk period every timer\_freq\_g. units: [Hz] |
| clk\_freq\_g | positive | 100000000 | the clock input to the block. this is the clock used in the system containing the timer unit. units: [Hz] |
| timer\_en\_polarity\_g | std\_logic | 1 | defines the active state of the timer: '0' active low, '1' active high |
| advanced\_amount\_g | natural | 1 | determines the amount the address output is advanced every time |

28

The led clients generics are all its sub-units generics.

The Led client contatins the following units:

**Led Registers -** Contains 5 registers which control the functionality of the leds. It receives this data from the wishbone slave of the led\_client. The registers:

* Enable register: enables led operation and flickering in the following way. Each led has the following bits:

Enable bit: ‘0’ to disable the led and ‘1’ to enable it.

Frequency enable bit -‘0’ signify that the led is turned on without flashing (if Enable equal to '1') and ‘1’ signify that the led will flash.

The bits are mapped as follow:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Led 4 enable | Led 4 frequency enable | Led 3 enable | Led 3 frequency enable | Led 2 enable | Led 2 frequency enable | Led 1 enable | Led 1 frequency enable |

Table 29 - Led register mapping

* Frequency register (4 instances) – These registers include the frequency that the led will flash when the frequency enable is ‘1’.

The frequency of the flickering will be (time\_unit\_g) \*(Frequency value) .

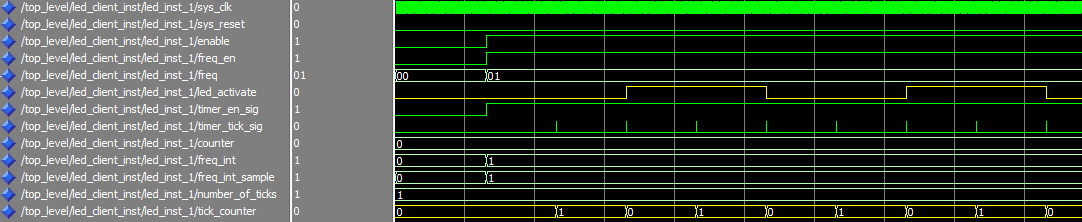
Example: A led that will receive 01\_0x on its frequency input would count: 2 timer ticks before it inverts the active signal. The timer tick will be asserted for one cycle every 100,000 system clock cycles – thus 1msec.

Figure 11 - Led waveform

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| data\_out | output | data\_width\_g | data sent to WS |
| valid\_data\_out | output | 1 | validity of data directed to WS |
| address\_in | input | addr\_d\_g \* data\_width\_g | address line |
| data\_in | input | data\_width\_g | data sent from WS |
| valid\_in | input | 1 | validity of the address directed from WS |
| wr\_en | input | 1 | write enable: '1' for write, '0' for read |
| CYC\_I | input | 1 | active wishbone cycle |
| STB\_I | input | 1 | active wishbone operation within a cycle |
| en\_out | input | data\_width\_g | enable data sent to led\_1, led\_2, led\_3, led\_4 |
| freq\_out\_1 | output | data\_width\_g | frequency data sent to led\_1 |
| freq\_out\_2 | output | data\_width\_g | frequency data sent to led\_2 |
| freq\_out\_3 | output | data\_width\_g | frequency data sent to led\_3 |
| freq\_out\_4 | output | data\_width\_g | frequency data sent to led\_4 |

Table 30 - Led Registers interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 |
| en\_reg\_address\_g | natural | 0 | enable register address |
| freq\_reg\_1\_address\_g | natural | 1 | frequency register 1 address |
| freq\_reg\_2\_address\_g | natural | 2 | frequency register 2 address |
| freq\_reg\_3\_address\_g | natural | 3 | frequency register 3 address |
| freq\_reg\_4\_address\_g | natural | 4 | frequency register 4 address |
| advanced\_amount\_g | natural | 1 | determines the amount the address output is advanced every time |

Table 31 - Led Registers generics

**Address advancer -** Recieves an initial address from the wishbone slave and advances it every new valid cycle, hence when there is a rising edge of the STB\_I signal and CYC\_I =’1’. It is a generic unit that is placed inside the led\_registers unit.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| address\_in | input | addr\_d\_g \* data\_width\_g | initial address given from wishbone interface |
| address\_out | input | addr\_d\_g \* data\_width\_g | current address |
| CYC\_I | input | 1 | active wishbone cycle |
| STB\_I | input | 1 | active wishbone operation within a cycle |

Table 32 - Address Advancer interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 |
| advanced\_amount\_g | natural | 1 | determines the amount the address output is advanced every time |

Table 33 - Address Advancer generics

**LED (4 instances) -**  Every register value would be transferred to a led block that will function accordingly. The block’s output will be connected to a LED on the DE2 board. Inside every led unit there is a timer unit which raises a timer tick every given amount of clock cycles (set by the timer\_freq\_g generic).

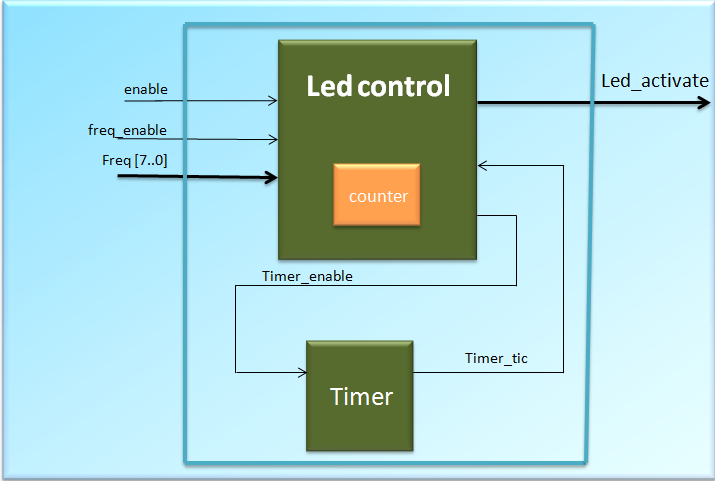


Figure 12 - Led unit

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | input | 1 | system clock |
| sys\_reset | input | 1 | system reset |
| enable | input | 1 | determines if the led is enables |
| freq\_en | input | 1 | enables led flickering |
| freq | input | freq\_width\_g | led flickering frequency recieved from registers |
| led\_activate | output | 1 | this signals activates the led: '1' led is turned on, '0' led is turned off |

Table 34 - Led interface

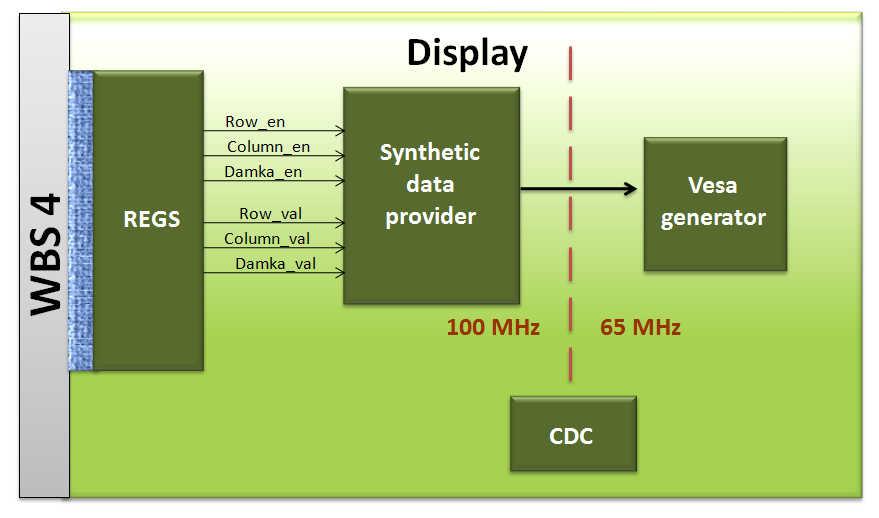
|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default value** | **description** |
| reset\_polarity\_g | std\_logic | 1 | '0' = Active Low, '1' = Active High |
| data\_width\_g | natural | 8 | defines the width of the data lines of the system |
| block\_enable\_polarity\_g | std\_logic | 1 | defines the active block enable polarity: '0' active low, '1' active high |
| freq\_enable\_polarity\_g | std\_logic | 1 | defines freq\_en input active state: '0' active low, '1' active high |
| freq\_width\_g | natural | 8 | width of the frequency vector |
| timer\_freq\_g | positive | 1000 | timer\_tick will raise for 1 sys\_clk period every timer\_freq\_g. units: [Hz] |
| clk\_freq\_g | positive | 100000000 | the clock input to the block. this is the clock used in the system containing the timer unit. units: [Hz] |
| timer\_en\_polarity\_g | std\_logic | 1 | defines the active state of the timer: '0' active low, '1' active high |

Table 35 - Led generics

**Timer**: see at wait client.

**Wishbone Slave -** see wishbone units.

## Display – client



General Description

1. The display client would control an external screen in 1024X768 resolution. The ROI (region of interest) is 800X600.

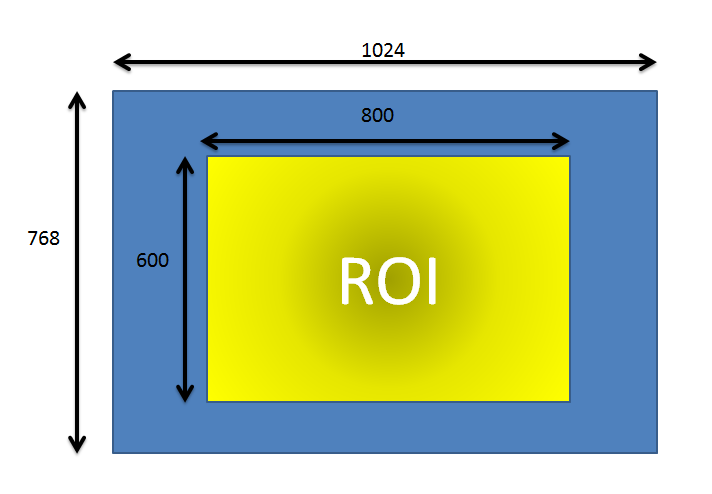


Figure 13 - Display resolution and ROI

1. The display client has a wishbone slave interface. WS number 5.
2. An image can be display on the screen in one form a line, column or squares (Damka) format according to register values. The Image would be displayed with a graduate change of colors from line to line (or column to column, or square to square)
3. The display interface to the display will be with VESA protocol which operates with a 65MHz clock.
4. In order to fit the system clock of 100 MHz to the VESA clock of 65 MHz we will use Clock Domain Crossing (CDC) hardware.
5. The system includes the following registers:

Line\_Location\_reg – The location of the top left corner of the ROI using line mode

Column\_Location\_reg – The location of the top left corner of the ROI using Column mode

Damka\_Location\_reg – The location of the top left corner of the ROI using Damka mode

Line\_width\_reg – the width of a single line

Column\_width\_reg – the width of a single column

Damka\_width\_reg – the width of the side of a single square

r\_Line\_jump\_value\_reg – the difference of colors between each line – red color

g\_Line\_jump\_value\_reg – the difference of colors between each line – green color

b\_Line\_jump\_value\_reg – the difference of colors between each line – blue color

r\_Column\_jump\_value\_reg - the difference of colors between each column – red color

g\_Column\_jump\_value\_reg - the difference of colors between each column – green color

b\_Column\_jump\_value\_reg - the difference of colors between each column – blue color

r\_damka\_jump\_value\_reg - the difference of colors between each square – red color

g\_damka\_jump\_value\_reg - the difference of colors between each square – green color

b\_damka\_jump\_value\_reg - the difference of colors between each square – blue color

Enable\_reg – enabling parameters:

* VESA enable – enables the operation of the VESA block
* Picture enable – enables picture displaying. If the VESA enable is enabled and the picture enable is not, then a default constant display will be displayed.
* Line enable – enables line framing
* Column enable – enable column framing
* Damka enable – enables square framing

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| Not used | Not used | Not used | Damka  Enable | Column enable | Line enable | Picture enable | VESA enable |

Table 36 - VESA enable\_reg mapping

1. The display unit does not cope with an out of boundary values of the ROI. The location\_reg will only receive values that for them all the ROI will be inside the screen. The host’s GUI would be responsible to check the legality of these values.

## Wishbone Blocks and protocol

### **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.

Wishbone cycle example: In the following waveform a master is writing 5 bytes (TGD=04\_0x) to the Led client which is client number 4 (TGA=04\_0x) . The initial address is ADR=000000\_0x. Notice the assertion of CYC in the beginning of the process and the assertion of STB every new data. The slave asserts ACK=’1’ for one cycle for each data byte.

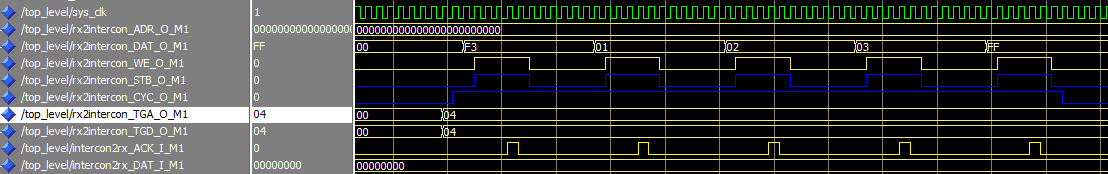


Figure 14 - Wishbone cycle waveform example

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

Wishbone slave 7 – LCD client (optional)

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.

### **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 threw 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.

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

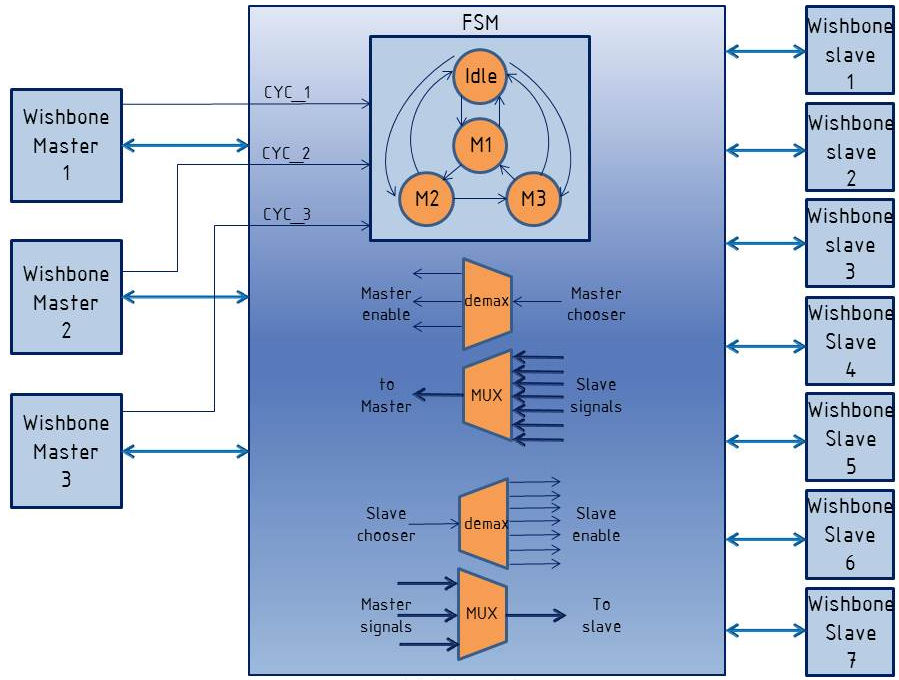


Figure 15 - Wishbone Intercon

### **Wishbone Master**

The Wishbone Master is the unit that initiates and manages a transaction on the wishbone bus. Although it seems like an active unit, it actually needs to be operated by its host block. For example , the MDWM in the RX path operates WM1. The MDWM reads the data from the RAM and transfers the data to WM1 in order to be transmitted to a chosen slave. The principle is the Wishbone Master is a simple unit that does only its role – transfer data on the wishbone bus. No reading data from RAM or any other operations except the transition.

The Wishbone master operates according to its FSM. There are 2 main branches on this FSM:

Read – for a reading transaction

Write – for a writing transaction

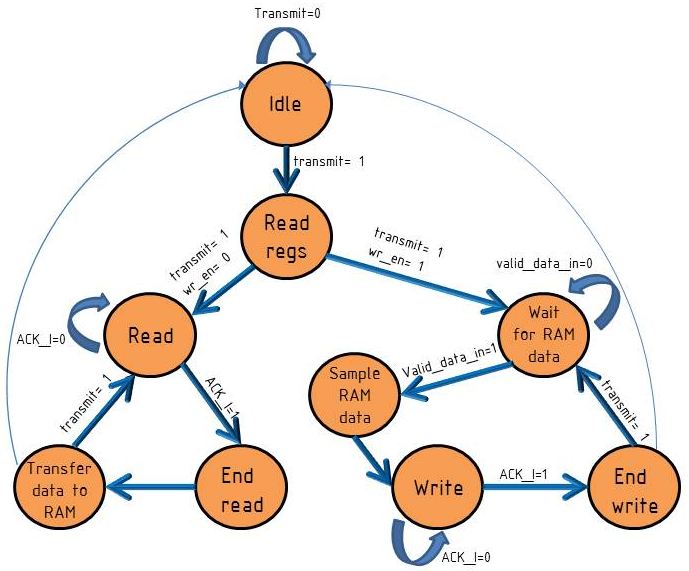


Figure 16 - Wishbone Master FSM

|  |  |  |  |
| --- | --- | --- | --- |
| **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 recieved from WS or for successfull write operation in WS |
| DAT\_I | input | data\_width\_g | data recieved from WS |
| data\_in | input | data\_width\_g | data to be transmited via bus |
| valid\_data\_in | input | 1 | data validity |
| wr\_en | input | 1 | determines if the WM will make a read('0') or write('1') request |
| transmit | input | 1 | when '1' the units request permission to use the bus by asserting CYC\_O |
| 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 |
| ack | output | 1 | informs the user side that ACK\_I has been asserted |
| data\_out | output | data\_width\_g | data recieved from WS , valid when ack='1' |

Table 36 - Wishbone Master interface

|  |  |  |  |
| --- | --- | --- | --- |
| **generic name** | **type** | **Default 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 37 - Wishbone Master generics

### **Wishbone Slave**

Wishbone Slave is the client’s interface to the system. The unit is based on the same principles as the wishbone master. It has the simplest hardware that can handle the wishbone communication. Therefore it only responds to wishbone master signals and passes any information received to its host’s units without any processing. It operates according to the following FSM that has two branches: one for reading cycles and the other for writing cycles.

|  |  |  |  |
| --- | --- | --- | --- |
| **signal name** | **type** | **width (bits)** | **description** |
| sys\_clk | Input | 1 | system clock |
| sys\_reset | Input | 1 | system reset |
| ADR\_I | Input | addr\_d\_g \* data\_width\_g | contains the address 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 | 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 recieved from WS or for successfull write operation in WS |
| DAT\_O | Output | data\_width\_g | data recieved from WS |
| data\_in | Input | data\_width\_g | data that was transmited frome the user side |
| valid\_in | Input | 1 | data validity |
| we\_out | Output | 1 | determines if the WM did a make a read('0') or write('1') requeste |
| type\_out | Output | type\_d\_g \* data\_width\_g | type is the client which the data is directed to |
| len\_out | Output | len\_d\_g \* data\_width\_g | length of the data (in words) |
| addr\_out | Output | addr\_d\_g \* data\_width\_g | the address in the client that the information will be written to |
| user\_ack\_i | Input | 1 | the user uprise this signal in order to force ACK\_O to be '1' |
| cyc\_out | Output | 1 | the CYC\_I signal transfering to the user side |
| stb\_out | Output | 1 | the STB\_I signal transfering to the user side |
| valid\_out | Output | 1 | data output validity |
| data\_out | Output | data\_width\_g | data transmited to the user side |

Table 38 - whishbone slave interface

Wishbone slave generics : same as whishbone master, see there.

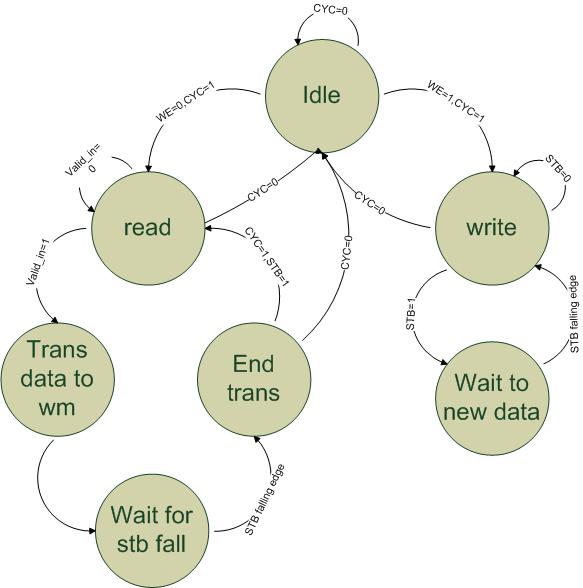


Figure 17 - Wishbone Slave FSM

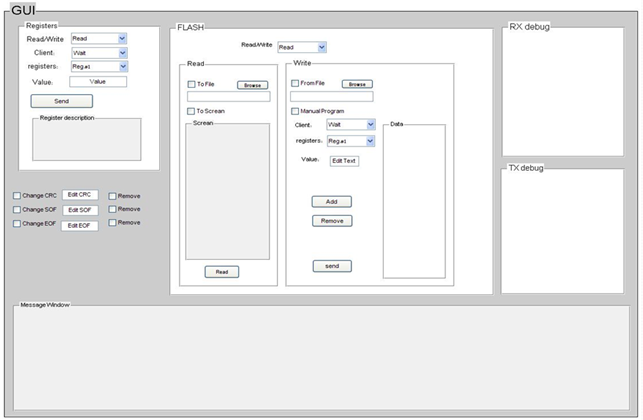
GUI MATLAB  


Figure 18 - GUI

General Description

1. The GUI will give the user the following capability to Read and write data to Flash or directly to a client. In addition, it would have features for debugging.
2. Communication to system via UART protocol with baud rate of 115,200 kbits/sec.
3. One of The GUI aims is to simplify user operations as much as possible. Therefore the user won’t need to insert information like SOF, EOF, CRC but only the minimum relevant data.
4. **GUI capabilities:**

* Data transfer directly to registers in clients. The user would type the following information to use this feature:

|  |  |  |
| --- | --- | --- |
| Client | Register | Value |

The GUI would show information about a register once it is chosen.

* Reading data from registers and displaying it on the GUI.
* Transmitting data to Flash by one of two methods:

1. Direct typing
2. Loading data from a text file.

* Dumping data received from Flash or from clients to a text file.
* Correction check of data inputted by user and notification if data was inputted incorrectly.
* Data wrapping in order to meet system message protocol, including CRC calculation.
* An option to cancel or change CRC, EOF, SOF for debug purposes.
* Two display fields, that would give extra debugging ability:

1. Transmitted data display (TX debug)
2. Received data display (RX debug)

* Manual writing of data that will be transmitted to Flash or to clients.

# Data transfer

# Write transaction

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

1 - User enters in the **Matlab GUI** the data to be sent.

2 – The GUI checks correctness of the data input. If the check passes data is sent on the UART line to the RX path else an error message is displayed to the user.

3 – 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)

4 – 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).

5 – 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**.

6 – If no errors are received (CRC error for example) the type, address and length data is transferred to registers on the **mdwm** (Message Decoder to Wishbone Master) in order to start data transfer.

7 – The mdwm reads data from RAM one byte at a time and commands the **wishbone\_master (WM)** to start a wishbone bus transaction.

8 – 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.

9 – When successful writing of a byte is done in a **client** it responds with ACK signal assertion to the wishbone\_master. When this signal is received it is passed to the mdwm. When the number of ACK assertion reaches the <Length + 1> in the mdwm register it requests the wishbone\_master to end the transaction.

### Write transaction example

In this example 5 data bytes are going to be written to the Led Client.

* Data is entered in GUI
* The following data is sent on the UART line:

3C - SOF

04 – Led client type

00 00 00 – Initial address

04 – Length of data minus one

F3 11 12 13 FF – five data bytes

1C – CRC value

A5 – EOF

* Data is received by uart\_rx and transferred to mp\_dec. Data bytes are written to RAM. CRC check is done.

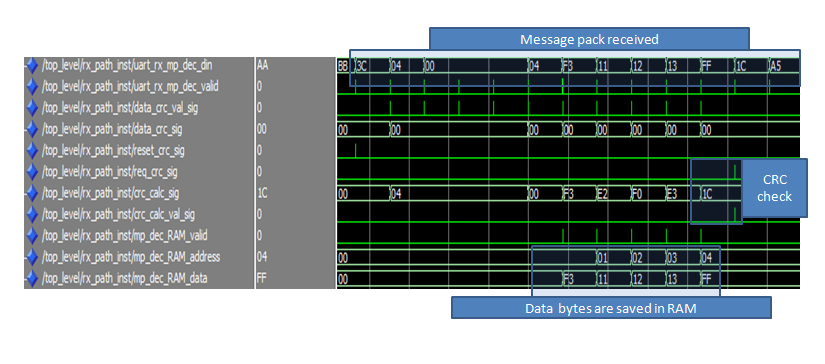


Figure -DATA PROCCESSED IN THE RX PATH

* Mdwm operation starts. Reads data from RAM and transfers it on the wishbone bus. The data is received by clients and saved in registers.

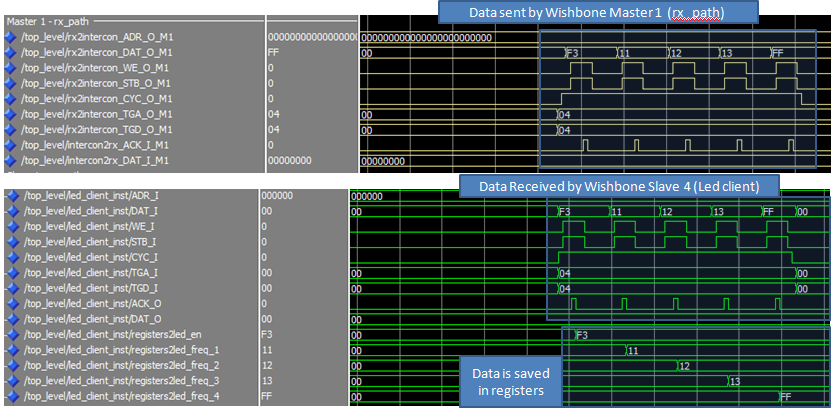


Figure - Data transfer from rx\_path to led client

## Read transaction

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.

Host to Flash

Data from host to FLASH makes the following path:

1. From Host to RX Path via UART\_IN line.
2. The RX Path unwraps the data and transfers it to through the Wishbone bus to the FCB and then to the Flash.

Flash to Clients

Data from FLASH to clients makes the following path:

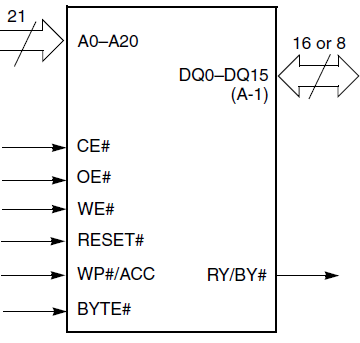
1. The FCB initiates a reading transaction and reads data from Flash.
2. All data for a specific configuration is transferred to the RAM on CCB using the FCB.
3. The CCB sends all the data to the clients via Wishbone bus.

Data sent back to host from Flash - Debug

Data sent back to host from FLASH makes the following path.

1. WBM1 (RX Path) initiates the transaction and orders WBS1 (TX Path) to begin a read transaction.
2. WBM2 (TX Path) orders the FCB to transfer data from Flash.
3. Data is transferred to RAM on TX Path via FCB and Wishbone bus.
4. Data is wrapped and sent to host via UART\_OUT line.

# FLASH Memory



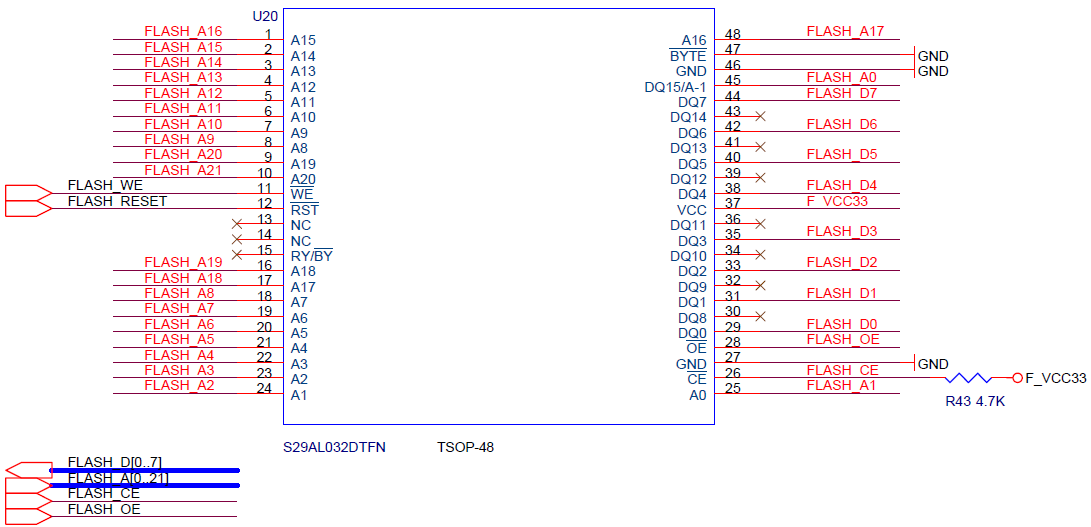


Figure 21 - Flash schematic symbol and schematics

1. The flash memory used in the system is the spansion S29AL032D included in the Altera DE2 development board. The flash uses Common Flash Interface (CFI) in its interface with the Cyclone II FPGA. CFI has been created to allow a system designer the flexibility to design products now that can use both current and future flash memory devices, as well as the security of knowing that second source products may be used without system software modifications.
2. Pin Configuration:

A0-A20 address inputs

DQ0-DQ14 15 data inputs/outputs

DQ15/A-1 DQ15 (data input/output, word mode), A-1 (LSB address input, byte mode)

BYTE# Selects 8-bit or 16-bit mode

CE# Chip enable

OE# Output enable

WE# Write enable

RESET# Hardware reset pin

WP#/ACC Hardware Write Protect input/Programming Acceleration input.

ACC Hardware Write Protect input

Testability

## Top level testing was done using the configuration seen in the figure:

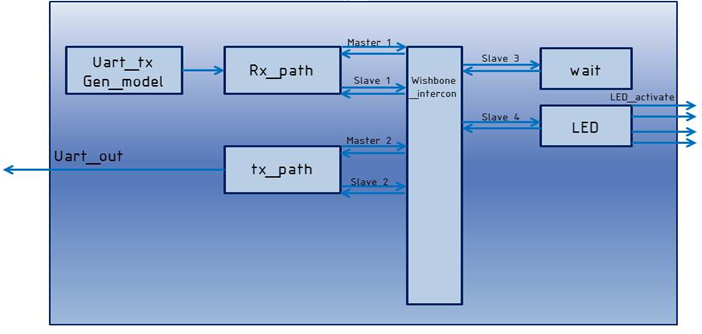


Figure 22 - Top level simulation

The **uart\_tx\_gen\_model** is a unit that reads data from a text file and sends it to the RX path via UART protocol. By watching the expected waveforms the test was defined successful or not.

The following tests were run on the system:

1. 5 word write transaction to Led client.
2. 1 word write transaction to Led client.
3. Two sequent write transactions to Led Client.
4. 3 word read transaction with Led client.
5. 1 word read transaction with Led client.
6. Two sequent read transactions to Led Client.
7. 3 sequent transactions with led client: write => read => write.
8. Write to a Led register the value 00\_0x.
9. Write transaction to Wait client.
10. 2 sequent write transactions to Wait client.
11. Write 00\_0x to Wait client.
12. Sequent transactions to different clients: Write to Led => Write to wait => Read from Led
13. Reading the error register value (WS1 – inside the rx\_path).
14. System reset while a wishbone active cycle occurs.

## 

## Messages to the FPGA

### Registers Write Message

|  |  |  |
| --- | --- | --- |
| SOF | | 0x3C |
| Write Register Type ID | | 0x75 |
| NA | Address [31:24] |  |
| NA | Address [23:16] |  |
| [15:12] – NA  [11:8] – Space of registers address [4:1] | Address [15:8] |  |
| [7] – Space of registers address [0]  [6:0] – Local start address of burst registers [6:0] | Address [7:0] |  |
| [15:10] – NA  [9:8] - Data Length[9:8] | Length [15:8] |  |
| [7:0] - Data Length[7:0] | Length [7:0] |  |
| [7:0] - Registers Data of Start Address[7:0] | Data #1[7:0] |  |
| [7:0] - Registers Data of Start Address+1[7:0] | Data #2[7:0] |  |
| .  .  . | |  |
| [7:2] - Registers Data of Start Address+N[7:0] | Data #N[7:0] |  |
| [7:0] | |  |
| EOF | | 0xA5 |

* Min burst is 1
* Max burst is limited by number of registers in current address space (or 1023)