VME64x to WB core
User Guide

Group: CO-HT

Author:
Davide Pedretti

Date: 17/09/2012

Table of Contents

1 Introduction

This document provides an overview of the VME64x to WB core features.
This core implements a VME64 slave on one side and a WishBone master on the other without FIFOs in-between.

The vme64x core conform to the standards defined by ANSI/VITA VME64x Standard [1] [2][3][6].
In particular this core has been provided of the *"plug and play"* capability; it means that in the vme64x core you can find a CR/CSR space whose base address is setted automatically with the geographical address lines and has not to be set by jumpers or switches on the board. Indeed this operation is error prone.
Software must map the module memory in the 64-bit address space by writing the CSR space as explained later.

The core supports SINGLE, BLT (D32), MBLT (D64) transfers in A16, A24, A32 and A64 address modes and D08 (OE), D16, D32, D64 data transfers. The core can be configured via the CR/CSR configuration space. A ROACK type IRQ controller with one interrupt input and a programmable interrupt level and Status/ID register is also provided.

The document also provides an overview of the default power-up configuration and configuration procedure.

Since the vme64x core acts as a WB master in the WB side, the WB **pipelined single** read/write transfer has been provided to the core. This functionality conform the Wishbone B4 standard [5].

The implementation follows the design rules set by Design rules for custom VME modules in CMS [4].

The project can be found at the following link:

http://www.ohwr.org/projects/vme64x-core

2 VME64x features

This chapter lists and explains features the VME64x slave implements.

2.1 CR/CSR space

To provide a “plug and play” capability CR/CSR space is implemented as defined by ANSI/VITA Standards for VME64 Extensions [2].

A dedicated “Configuration ROM / Control & Status Register” (CR/CSR) address space has been introduced. It consists of ROM and RAM regions with a set of well defined registers. It is addressed with the address modifier 0x2F in the A24 address space.

Every VME module occupies a 512 kB page in this address space. The location of this page in the A24 space is defined by geographical address lines on the backplane: each slot is provided with a unique geographical five bit address at the J1 connector (row d). From these bits A23...A19 of the CR/CSR page are derived.
If the geographical address is not correct (GA parity bit does not match), the base address is set to 0x00, which indicates a faulty condition. An **odd** parity is used.
If the board is plugged into an old crate that doesn't provide the GA lines, all the GA lines are asserted to '1' by the pull-up resistors on the boards and the BAR register is set to 0x00; it is not set by hand with some switches on the board so in this condition the CR/CSR space can't be accessed.

The CR/CSR space can be accessed with the data width D08(EO), D16 byte(2-3) and D32. Please note that in compliance with the CR/CSR definition, only every fourth location in the CR/CSR space is used. If the master tries to write another location the write will not take effect and if the master reads the byte(0) or byte(1) or byte(2) locations the value returned is 0.

As you can see in the figure 1, not all this space of memory is defined yet, and the digital designer can add additional CR and CSR spaces called User Csr and User CR which are not implemented in the vme64x core.
The location of the User CR and User CSR regions as well as the CRAM region are programmable. For each of these, six bytes defining the start and the end address (with respect to the start of the configuration space) are reserved in the CR region. Designers are free to use these regions for module specific purposes.

In the vme64x core only the CRAM space is implemented from 0x001000 to 0x0013ff (1 KB).
If a freely programmable region (see fig. 1) is not implemented the start and end address must be set to 0x000000.

All the registers in the CSR space have been implemented as defined by the VME64 Extensions [2] and the registers showed in the table 1 have been added.

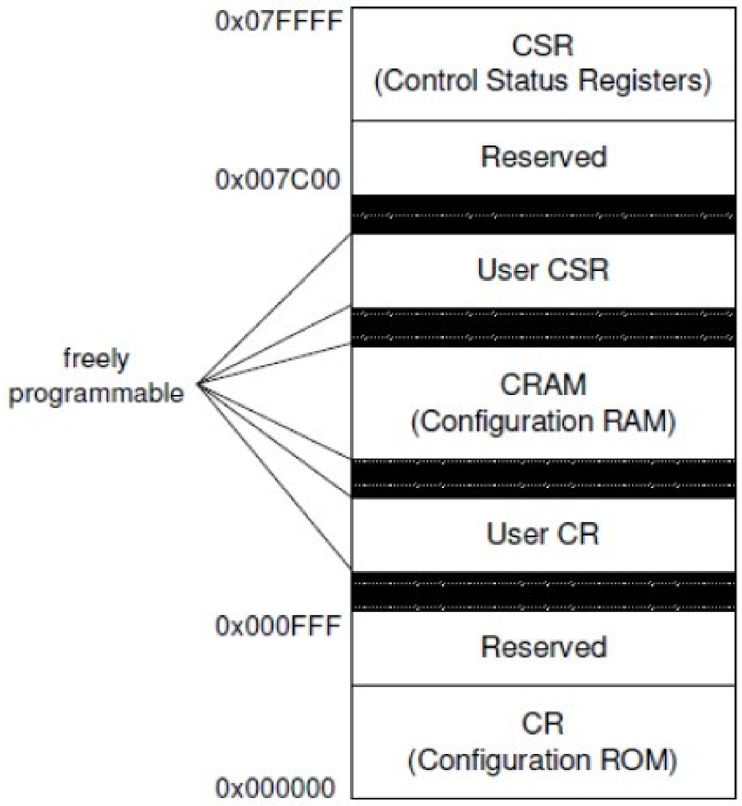


Figure 1: Configuration space organization in VME64x

Table 1 : *Additional Registers in the CSR space*

Name of the register	**location**
IRQ_Vector	0x07FF5F
IRQ_level	0x07FF5B
MBLT_Endian	0x07FF53
TIME0	0X07FF4F
TIME1	0X07FF4b
TIME2	0x07FF47
TIME3	0x07FF43
TIME4	0x07FF3F
BYTES0	0x07FF3b
BYTES1	0x07FF37
WB32bits	0x07FF33

The VME Master can set the IRQ level and the Status/ID (IRQ Vector) of each slave board. These two registers are used by the Interrupt Controller, see section 3.6.

With the MBLT_Endian register the Master can select the data swap type:

Table 2 : MBLT Endian register-Swap Type

MBLT_Endian value	**Swap type**	**8 bytes data**	**Swapped data**
0x00	No Swap	01234567	01234567
0x01	Byte Swap	01234567	10325476
0x02	Word Swap	01234567	23016745
0x03	Word + Byte Swap	01234567	32107654
0x04	Dword + Word + Byte Swap	01234567	76543210

All the TIMEx and BYTEx registers can be used to calculate the data transfer rate; these registers stored the time in ns and the number of bytes transferred in the last WB access.
The Time is misured as showed in the following image:

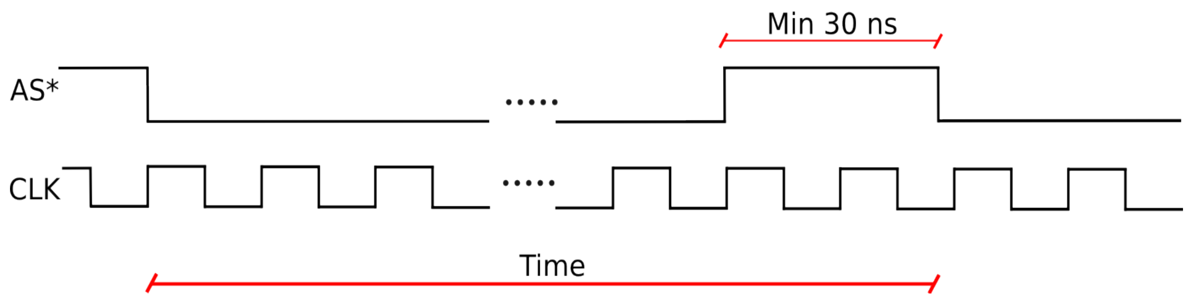


Figure 2: Time

We can misure the time between the AS falling and rising edge, but we do not know how much time the VME Master takes before the next access. Indeed the VME bus can be busy if another master or an interrupt handler is using it. Thus to calculate the transfer rate we add the minimum time (30 ns) between two consecutive transfers.
It means that the real transfer rate can be a little less than the value calculated with this time, because between two accesses the VME Master should wait more than 30 ns.
Any way the transfer rate thus calculated is a good test in order to see immediately if the cycle is terminated correctly or with a time out error.
The division between the number of bytes and the time shall be done by the softwere, indeed a divisor implemented in hardware will take a lot of resource in the FPGA.

The WB32bits tells the Master if the Slave WB data bus is 32 or 64 bits wide:
0x00 : the WB data bus is 64 bits.
0x01 : the WB data bus is 32 bits.
This is a read only register.

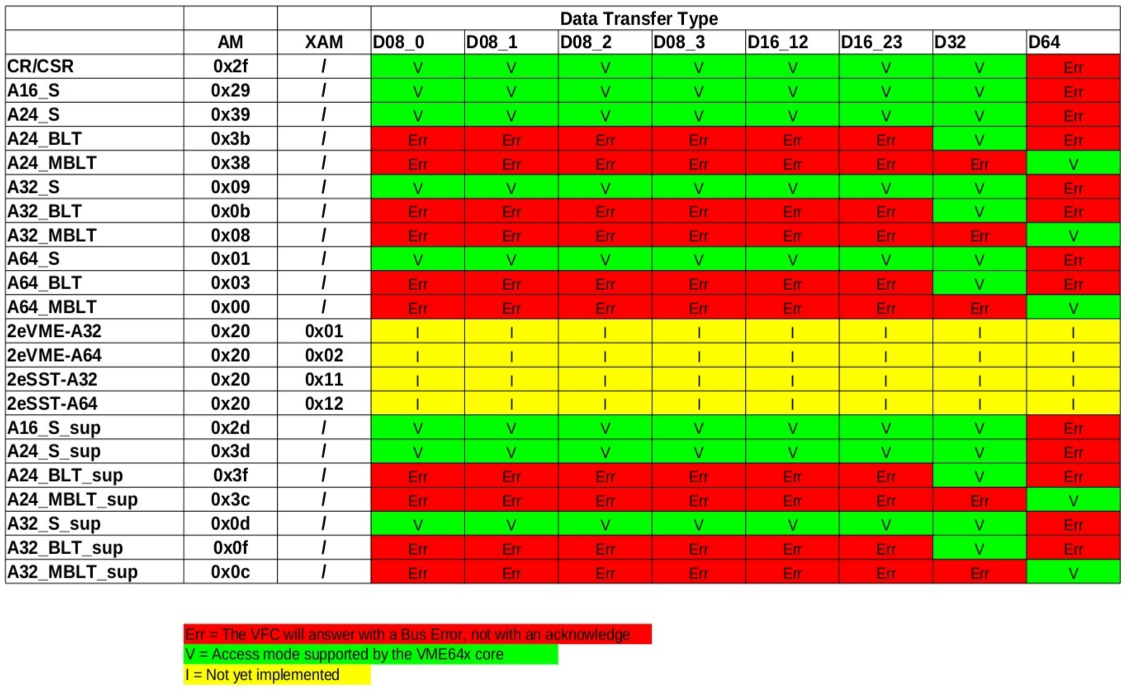
In the CR/CSR space a BIG ENDIAN order is used.

2.2 Data types

This VME64x core supports D08(EO), D16, D32 and D64 data transfers. The implementation assumes that the target data memory is 8-bit wide.
Each byte in the memory has one unique address.
The vme64x core supports the byte access so the WB memory should have an 8 bit granularity.
Upon D16 access, only every other byte is addressed and the D16 byte(1-2) access is not supported, upon D32 every fourth and upon D64 data access only every eighth byte is addressed.

2.3 Addressing types

The address width can be 16 bit, 24 bit, 32 bit and 64 bit to address more than 1 TB of memory.
The figure 2 shows all the access modes implemented and the relative Address Modifier (AM).



			Data Transfer Type							
	AM	XAM	D08_0	D08_1	D08_2	D08_3	D16_12	D16_23	D32	D64
CR/CSR	0x2f	/	V	V	V	V	V	V	V	Err
A16_S	0x29	/	V	V	V	V	V	V	V	Err
A24_S	0x39	/	V	V	V	V	V	V	V	Err
A24_BLT	0x3b	/	Err	Err	Err	Err	Err	Err	V	Err
A24_MBLT	0x38	/	Err	Err	Err	Err	Err	Err	Err	V
A32_S	0x09	/	V	V	V	V	V	V	V	Err
A32_BLT	0x0b	/	Err	Err	Err	Err	Err	Err	V	Err
A32_MBLT	0x08	/	Err	Err	Err	Err	Err	Err	Err	V
A64_S	0x01	/	V	V	V	V	V	V	V	Err
A64_BLT	0x03	/	Err	Err	Err	Err	Err	Err	V	Err
A64_MBLT	0x00	/	Err	Err	Err	Err	Err	Err	Err	V
2eVME-A32	0x20	0x01	I	I	I	I	I	I	I	I
2eVME-A64	0x20	0x02	I	I	I	I	I	I	I	I
2eSST-A32	0x20	0x11	I	I	I	I	I	I	I	I
2eSST-A64	0x20	0x12	I	I	I	I	I	I	I	I
A16_S_sup	0x2d	/	V	V	V	V	V	V	V	Err
A24_S_sup	0x3d	/	V	V	V	V	V	V	V	Err
A24_BLT_sup	0x3f	/	Err	Err	Err	Err	Err	Err	V	Err
A24_MBLT_sup	0x3c	/	Err	Err	Err	Err	Err	Err	Err	V
A32_S_sup	0x0d	/	V	V	V	V	V	V	V	Err
A32_BLT_sup	0x0f	/	Err	Err	Err	Err	Err	Err	V	Err
A32_MBLT_sup	0x0c	/	Err	Err	Err	Err	Err	Err	Err	V

Err = The VFC will answer with a Bus Error, not with an acknowledge
V = Access mode supported by the VME64x core
I = Not yet implemented

Figure 2: Addressing types

The list of the addressing type supported with their address modifier codes can be found also in the ohwr:

http://www.ohwr.org/projects/vme64x-core/repository/raw/trunk/documentation/user_guides/VME_access_modes.pdf

2.4 Signals

This section focuses on functionality of certain VME bus signals.

2.4.1 Reset

The reset signal resets the entire core to the default configuration.
The reset signals are: VME_RST_n_i and the software reset (BIT_SET_REG[7]).
The BIT_SET_REG is located in the CSR space (address: 0x7fffb).
The software can reset the vme64x core by writing '1' the BIT_SET_REG's bit 7.
Since after a reset assertion the CSR space is overwritten with the defoult values, the reset is automatically released.
Please wait about 8800 ns after the reset assertion; the vme64x core needs this time to be initialized again.

2.4.2 Berr

The Berr signal is used to signal a bus error. A transfer cycle is terminated with assertion of this signal in the following cases:

- the VME64x slave does not recognize the data or addressing type used in the transfer cycle.
- if a master attempts to write to a read-only memory (CR).
- if an error is received from the module which is addressed
- if master attempts to access in BLT mode with D08 or D16
- if the master attempts to access in MBLT mode and the WB data bus is only 32 bits.

The vme64x asserts this signal in the DTACK_LOW state, and not as soon as it detects an error condition, to avoid that a temporary error condition (during the decode access phase more combinatorial process are working at the same time and it is possible to have temporary error conditions) causes the Berr assertion.

2.4.3 Retry

The Retry signal terminates the transfer cycle if the VME64x slave receives a retry request from the addressed module (via the WishBone bus), signaling that the read/write request cannot be completed at this time.

2.5 Interrupts

The interrupt controller implemented is a ROAK (Release On Acknowledge) type controller. It means that the Interrupter releases the interrupt request lines when it acknowledges the interrupt cycle.
Upon synchronously detecting a rising edge on the interrupt request signal input on the WB bus, the VME64x core drives the IRQ request line on the VME bus low thus issuing an interrupt request. The VME master acknowledges the interrupt in a form of an IACK cycle.
During the IACK cycle the vme64x core sends the IRQ_Vector to the master. After the interrupt is acknowledged, the VME IRQ line is released.

A new rising edge on the input interrupt request line before the interrupt acknowledge cycle is terminated will be lost.

There are seven VME IRQ lines but only one interrupt request input. For the purpose of configuring which of the seven IRQ lines the VME64x core will drive (in response to a rising edge on the IRQ input), an IRQ Level register has been implemented in the user CSR space. The value of this register corresponds to the number of the IRQ line on the VME bus that is to be used (note that on the VME master side priorities are taken into account, IRQ7 having the highest priority and IRQ1 the lowest). If the IRQ level register is set to 0x00 or values above 0x07, interrupts are disabled.
In the default power-up and reset configuration the interrupts are disabled.
The interrupts can be enabled/disenabled by writing a register in the WB slave application.

Table 3 shows the Interrupt related registers in the CSR space:

Table 3: Interrupter registers

Register	Where	Address	Default value
IRQ Vector	CR/CSR Space	0x07ff5f	0x00
IRQ Level	CR/CSR Space	0x07ff5b	0x02

To generate the Interrupt request pulse, an Interrupt Request Generator must be inserted in the Application (WB Slave) . An example is showed in the Interconnection Diagram, chapter 9.

3 Configuration

Upon power-up or reset, the module is disabled and only its CR/CSR space can be accessed. Software must then first map the module memory in the 64-bit address space by setting the Address Decoder Compare (ADER) registers in CSR, which, together with Address Decoder Mask (ADEM) registers in the CR relocate the module memory to the desired address range.
ADER for each function also contains an AM or XAM code to which it responds.
After the module has been placed in the desired address space, it can be enabled by writing to Bit Set Register in the CSR and thus setting the correct enable bit:
BIT_SET_REG = 0x10
The Master can check the WB data bus wide reading the following register in the CSR space:
loc 0x7ff33 WB32bits = 0x00 ⟵ WB 64 bit.
loc 0x7ff33 WB32bits = 0x01 ⟵ WB 32 bit.

To access the WB bus we have 7 functions; only one at time can be selected.
If one of these functions is selected, it means that this is the responding Slave.
Each function has one ADER, one ADEM, one AMCAP and one XAMCAP register has shown in the VME64 Extensions ANSI/VITA 1.1 1997 [2] chapter 10.

With the CR default configuration used, each function decodes some access modes as shown in the following table.

Table 4: Functions used during the decode phase

Function	Access Modes	In use
Function 0	A32, A32 BLT, A32 MBLT (user/supervisor)	Yes
Function 1	A24, A24 BLT, A24 MBLT (user/supervisor)	Yes
Function 2	A16 (user/supervisor)	Yes
Function 3 and Function 4	A64, A64 BLT A64 MBLT	No
Function 5 and Function 6	2 edge modes	No
Function 7	/	No

We can not guarantee the correctness of the ADEM, AMCAP and XAMCAP registers corresponding to the functions 3, 4, 5, 6, and 7, in the CR space, since these functions are not used.

4 Guidelines to access a board

The aim of this chapter is give you some guidelines to access a board (slave module) that is carrying the vme64x core.
Of course you need a properly front-end software; this topic deserves a separate discussion and beyond the scope of this manual.
The first step is enter the python interprete and import the libraries thus you can call the following instructions.

1) CR/CSR space access.
This is the vme64x CR/CSR space:

```
 Begin CR      :  0x000000
 End CR        :  0x000fff
 Begin CRAM : 0x001000
 End CRAM    : 0x0013ff
 Begin CSR    : 0x07fc00
 End CSR       : 0x07ffff
```

How to access the CR/CSR space:

- AM = 0x2f;
- Address width = 24 bit
- Addr[23:19] = not(VME_GA) = Slot number.

Eg: we access the CR/CSR space of the board plugged in the Slot 8:

```
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=512*1024)
```

The parameter data_width can be also 32 ; the offset will be selected accordingly.

Now we read the BAR register located to the address 0x07FFFF:
Byte access:
```
>>> m.read(offset=0x7FFFF, num=1, width=8)
```
Word access:
```
>>> m.read(offset=0x7FFFE, num=1, width=16)
```
Dword access (if data_width=32):
```
>>> m.read(offset=0x7FFFC, num=1, width=32)
```

In the CR/CSR space only the byte 3 locations are implemented .

In a similar way you the reader can access all the other registers mapped in the CR/CSR space.

Other important flags implemented in the CSR space:
BIT_SET_REG[3] → error flag
The Master can check if the slave asserted the BERR* line:
```
>>> m.read(offset=0x7FFFB, num= 1, width=8)
```
if you read 0x18 it means that the BERR* line has been asserted.
The Master clears the error flag:
```
>>> m.write(offset=0x7FFF7, values= 8, width=8)
```

BIT_SET_REG[7] → software reset:
```
>>> m.write(offset=0x7FFFB, values= 128, width=8)
```
with this command is possible reset the core.

The field "value" does not accept hexadecimal values.

2) WB access:

This example shows a A32 single access.
First the VME Master maps the memory in the correct address space by writing the base address (eg 0xc0 or 192 in integer) in the ADER register.
As already said the function 0 decodes the access mode A32, thus the Master writes the FUNC0_ADER register which is 4 bytes in the CSR space:

```
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size=
512*1024)
>>> m.write(offset=0x7FF63, values= 192, width=8)
>>> m.write(offset=0x7FF67, values= 0, width=8)
>>> m.write(offset=0x7FF6B, values= 0, width=8)
>>> m.write(offset=0x7FF6F, values= 36, width=8)      36 = integer(AM & "00")  [*]
```

After power-up or reset the module is disabled. If this is the first access to the WB bus the Master shall enable the module:

```
>>> m.write(offset=0x7FFFB, values= 16, width=8).
```

Now the Master accesses the WB bus with data width 32 bits:

```
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size= “size
of your memory or less”)
>>> m.write(offset= shall be a multiple of 4, values=...., width=32)
>>> m.read(offset= shall be a multiple of 4, num= 1, width=32)
```

Likewise the Master can access with data_width = byte or word, but remember that the offset shall be setted accordingly or the software does not select the data_width specified.
To access with a different AM the Master shall write the properly ADER accordingly with the Table 4.

Note:
[*] Since the ADEM's DFS bit (Dinamic Function Sizing) in the CR space is '0', the Master can not specify the AM it is going to use, when it writes the less significant byte of the ADER.
More detail about DFS can be found in the VME64 Extensions ANSI/VITA 1.1 1997 [2] chapter 10.

More detailed guidelines to access a VME slave board can be found here:

http://www.ohwr.org/projects/vme64x-core/repository/raw/trunk/documentation/user_guides/Python_test.pdf

5 Transfer Rates

We report in the Table 5 the transfer rates calculated in two way:
a) with the software with the BYTES and TIME registers (see Table 1)
b) the transfer rate observed with the VMETRO Vanguard.
We already discussed about how the time is calculated (see figure 2). A similar approach is used to misure the transfer rate with the VMETRO; we read the time between two consecutive accesses on the waveform output from the VMETRO, and manually we calculate the transfer rate since we know the number of bytes transferred.
Give an accurate estimation of the transfer rate is very difficult, indeed it is influenced by a lot factors. Thus is important define well the test system used and the condition in which the test has been done.
The figure 3 shows the test system used.

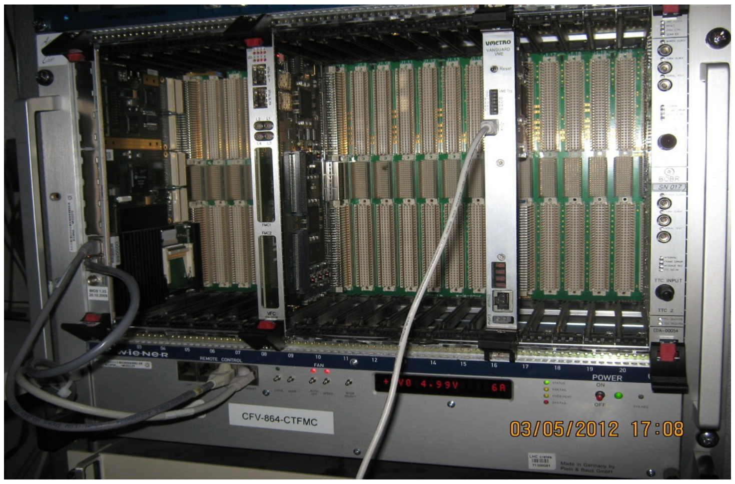


Figure 3: VME crate

You can see in the Slot 16 the Vanguard VMETRO connected by a usb cable to a monitor, in the Slot 8 the slave module and in the Slot 1 the VME front-end computer which is the A20 MAN Tundra board in which is located the TSI 148 PCIX-VME bridge. This board acts as VME Master converting the high level commands to an action on the VME bus.
Now let's see what we have in the WB side. The VME and the Wb are two handshake protocols and each the master and the slave in either the protocols can slow down the transfer rate taking how long they want before to acknowledge a cycle (slave) or starting a new cycle (master).

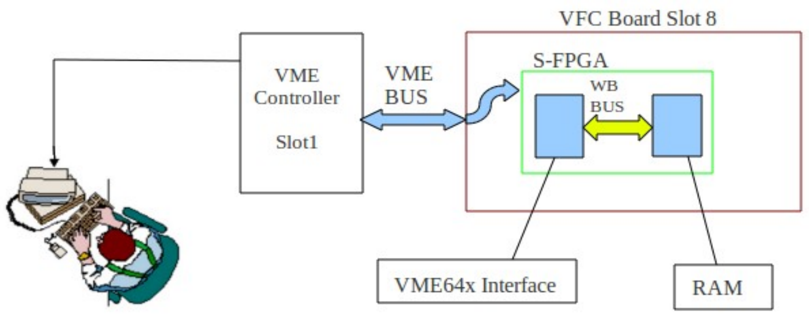


Figure 4: System test layout

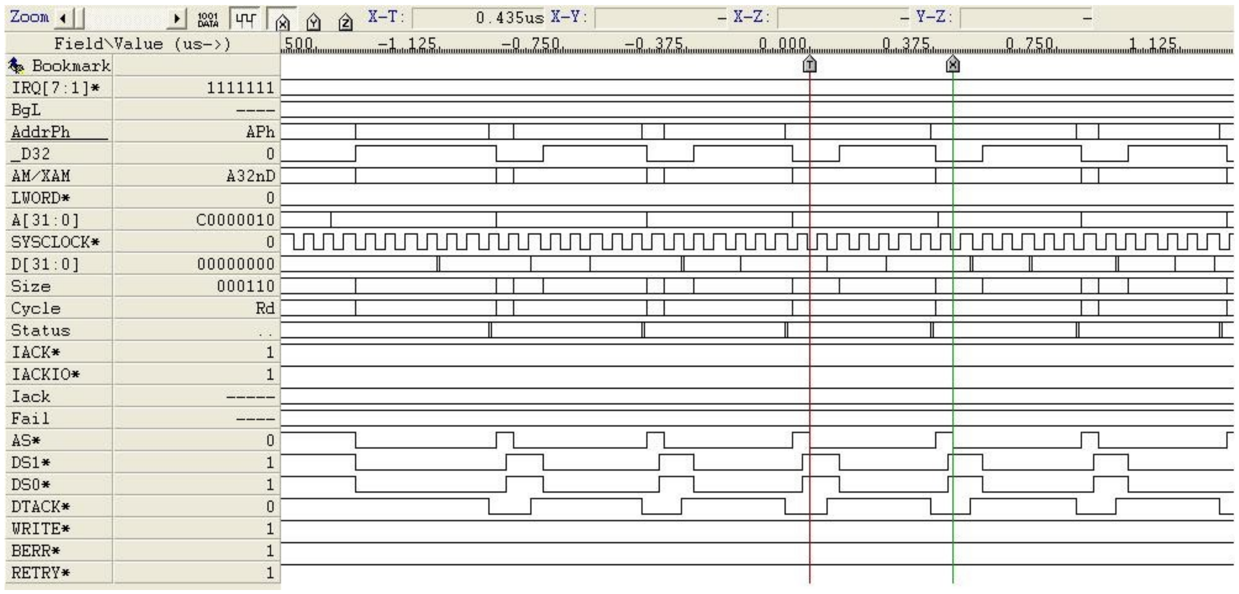
The WB slave used during the test is a simple RAM memory. Thus the vme64x core is connected to the WB salve by a simple poit to poit connection, and the WB slave is always ready; it gives back the acknowledge to the WB master (vme64x core) as soon as it detects the cyc and stb signals [5] asserted; the stall line is never asserted.

Summarizing in our test system we have only one VME Master (Tundra TSI 148), one VME slave (the board plugged in the slot 8), one WB master and one WB slave.
The Interrupt is disabled during the tests.
Of course the transfer rate depends on the clock frequency.
The resolution in the calculation of the transfer time depends on the clock frequency used in the vme64x core for the metod a, and on the sampling frequency of the VMETRO (133 MHz) for the metod b.
The next table shows the transfer rates detected during the tests:

Table 5: Transfer rates

		Clock frequency 50 MHz			
		Transfer type			
		SINGLE dma off	SINGLE dma on	BLT	MBLT
a) software	READ	9.75 MB/s	9.75 MB/s	9.09 MB/s	18.10 MB/s
	WRITE	10.25 MB/s	10.25 MB/s	10.51 MB/s	20.98 MB/s
b) VMETRO	READ	1.57 MB/s	9.19 MB/s	7.87 MB/s	16.29 MB/s
	WRITE	8.33 MB/s	9.52 MB/s	8.82 MB/s	20.03 MB/s
		Clock frequency 80 MHz			
		Transfer type			
		SINGLE dma off	SINGLE dma on	BLT	MBLT
a) software	READ	13.79 MB/s	13.79 MB/s	11.95 MB/s	23.81 MB/s
	WRITE	14.44 MB/s	14.44 MB/s	14.02 MB/s	28.32 MB/s
b) VMETRO	READ	1.68 MB/s	13.33 MB/s	10.25 MB/s	21.51 MB/s
	WRITE	11.33 MB/s	13.38 MB/s	10.68 MB/s	27.77 MB/s
		Clock frequency 100 MHz			
		Transfer type			
		SINGLE dma off	SINGLE dma on	BLT	MBLT
a) software	READ	16.00 MB/s	16.00 MB/s	14.05 MB/s	27.72 MB/s
	WRITE	16.67 MB/s	16.67 MB/s	16.66 MB/s	33.22 MB/s
b) VMETRO	READ	1.68 MB/s	14.81 MB/s	11.33 MB/s	23.79 MB/s
	WRITE	12.70 MB/s	16.13 MB/s	12.86 MB/s	30.93 MB/s

To better understand how we calculate the transfer rate with the VMETRO we report here a picture about an A32 single accesses with the dma on:



Field\Value (us->)	
Bookmark	
IRQ[7:1]*	1111111
BgL	----
AddrPh	APh
_D32	0
AM/XAM	A32nD
LWORD*	0
A[31:0]	C0000010
SYSCLOCK*	0
D[31:0]	00000000
Size	000110
Cycle	Rd
Status	. .
IACK*	1
IACKIO*	1
Iack	-----
Fail	----
AS*	0
DS1*	1
DS0*	1
DTACK*	0
WRITE*	1
BERR*	1
RETRY*	1

Figure 5: VMETRO output waveform

In this case the time between two consecutive accesses we are looking for is 435 ns as indicated by the two markers. We are accessing with a data transfer type D32 (all the single accesses in the above table are referred to D32 accesses); it means that the trasfer rate is $(4/435)x\,10^9 = 9.19\,MB/s$ as you can see in the table 5.
Now let's call the time during which AS* is asserted (low) *Cycle Time* and the time during which the AS* is not asserted (high) *Inter Cycle Time*. The transfer rate depends on both these time.
The hardware (vme64x core) acts only on the Cycle Time. The Inter Cycle Time depends only on the VME Master (A20 MAN board and software).
The transfer rates calculated by the software are optimistic, indeed the Inter Cycle Time used is 30 ns, the minimum.
We wish report here another table about the Inetr Cycle times observed with the VMETRO. This Table should help you to understand the values in the Table 5.

Table 6: Inter Cycle times

Transfer mode	**Inter Cycle time**
Single access dma off , Read	2200 ns
Single access dma on , Write	100 ns
Single access dma on, read/write	45 ns
BLT access read/write	4500 ns
MBLT access Read	12500 ns
MBLT access Write	4600 ns

Of course these Inter Cycle times are independend from the clock frequency used in the vme64x core. The resolution whereby these times are calculated is 7.5 ns indeed the sampling frequency of the VMETRO is 133 MHz.
Note that the BLT and MBLT transfer modes are possible only with dma on.

The first time reported in the Table 6 explain why with the VMETRO we observe a very low transfer rate during single read operations without dma.

The frequency increases does not increases a lot the transfer rates on the VME bus indeed only the Cycle time will be lower at higher frequencies.
Anyway make the vme64x core work at 100 MHz is important indeed it is an interface between the VME and a Wishbone bus without buffer in between and it means that the vme64x core must work at the same frequency as the WB application.
The WB is a synchronous protocol thus can be important work at 100 MHz in the WB side.

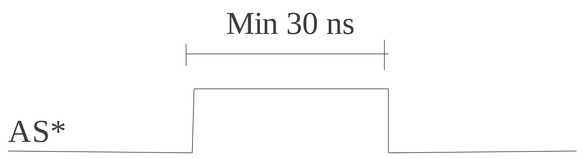
The last observation is that in our test system the BLT transfers are not faster than single transfers with dma on, indeed the VME Master takes long time (more than 100 ns) before asserting again the DS* lines in order to indicate that it is ready to write/read a new data. The waveform observed during BLT accesses is more or less the same as the waveform shown in figure 5, with the AS* asserted all the Cycle time.

6 Additional low level information-for a digital designer

6.1 System clock

The clock in the vme64x core must be the same as the clock in the WB Slave application, indeed the vme64x core does not provide buffer between the VME bus and the WB bus.
We suggest to use a clock frequency between 50 MHz and 100 MHz.

Fig. 25 , pag. 107 “VME bus specification” ANSI/IEEE STD1014-1987



As shown in the figure, to be sure that the slave detects the rising edge and the following falling edge on the AS* signal the clk_i's period must be maximum 30 ns.
The minimum time between two consecutive cycles observed with the VMETRO is 45 ns, but a 50 MHz clock or more is suggested. This time of 45 ns has been observed with a MEN A20 CPU board.
The VME Bus is asynchronous so each board plugged into the crate can work with a different frequency.

6.2 CR CSR space

The CR space is implemented in the file VME_CR_pack.vhd.
The CSR space is implemented in the file VME_CSR_pack.vhd.

6.3 IRQ Controller FSM

This is the finite state machine implemented to obtain the Interrupter features mentioned in the chapter 3.6

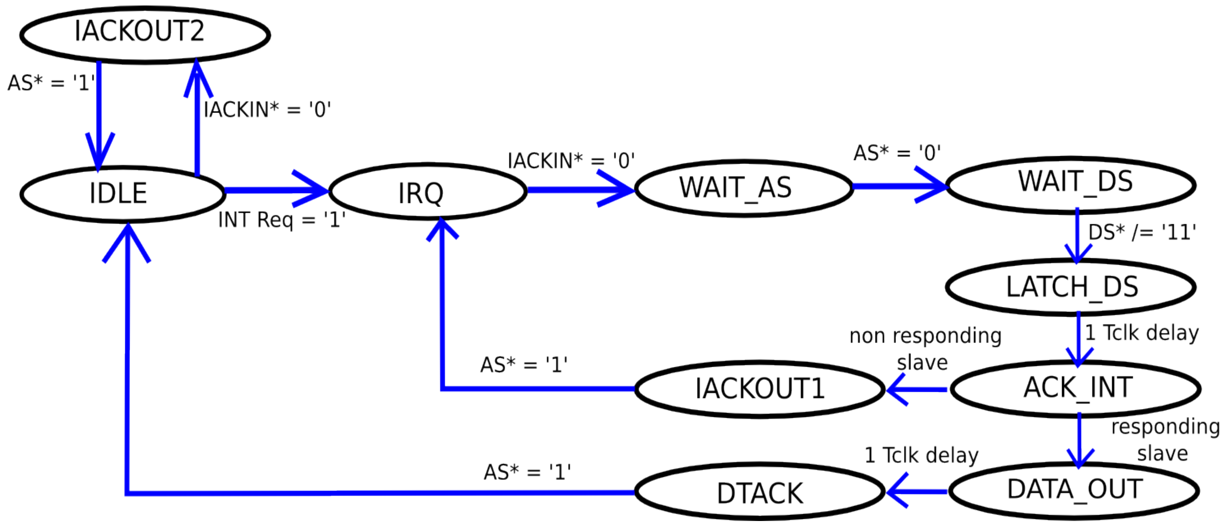


Figure 6: IRQ Controller FSM

6.4 Main FSM

The Main finite state machine is located in the VME_bus.vhd component.
This FSM does not support the 2e transfers.
If the board is not addressed the FSM is in the DECODE_ACCESS state until the rising edge on AS signal. Indeed the FSM is reset by a rising edge on the AS signal.
If the board is addressed, it will acknowledge the cycle in the DTACK_LOW state. If an error is detected, in this state the BERR* line is asserted.

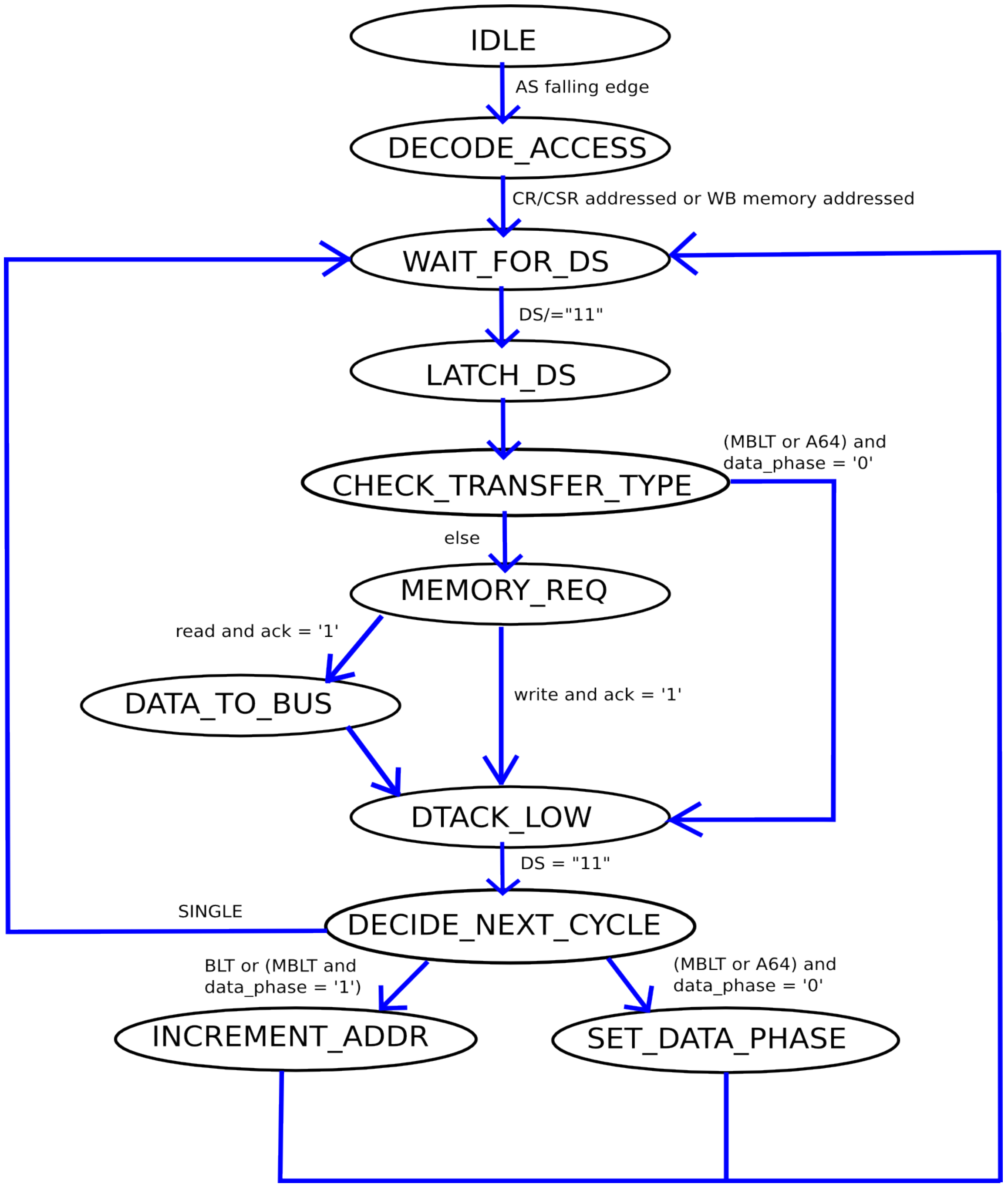


Figure 7: Main Finite State Machine

6.5 VME bus transceivers

The VME64x slave core also includes output signals that drive external hardware transceivers. These signals are DTACK OE, DATA DIR, DATA OE, ADDR DIR and ADDR OE, RETRY_OE.
Direction (DIR) signals specify the direction of data and address. These signals should be used as select line in the multiplexer on bidirectional signals.
Output enable (OE) signals are used to disable the transceivers so that the buses are effectively isolated.

Table 6: VME Bus Transceivers

OEAB	OEBYn	OUTPUT
L	H	Z
H	H	A to B
L	L	B to A
H	L	A to B and B to Y

OEn	DIR	OUTPUT
H	X	Z
L	H	A to B
L	L	B to A

6.6 WishBone master

This section describes the functional operation of the WishBone Master component VME_Wb_master.vhd.
When the core is addressed with SINGLE, BLT, MBLT, 2eVME (not yet supported) transfers the WB master operates in accordance with the WB specification document [5].
In particular it works in pipelined single write/read mode:

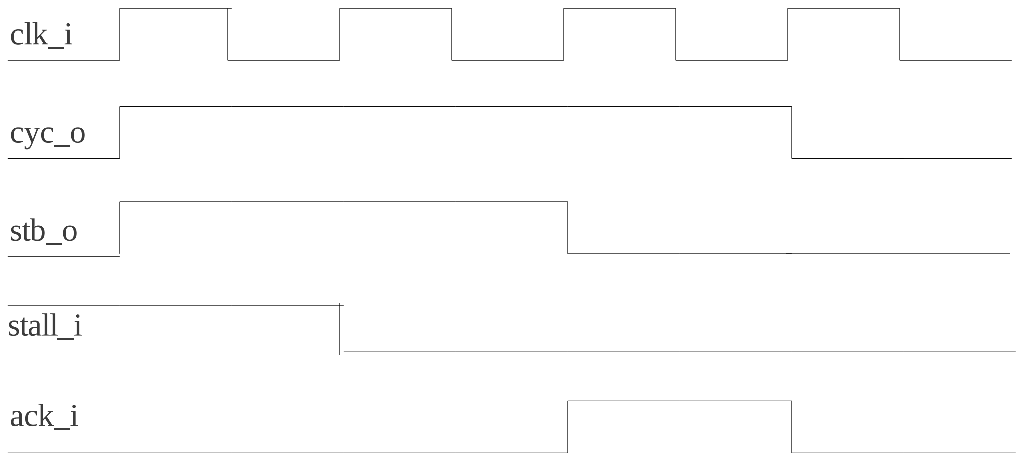


Figure 8: WB Master

If the WB slave doesn't drive the stall signal, it can set the stall signal to 0.
Err and rty signals are supported by the vme64x core.
The Wb bus can be 32 bit or 64 bit ; see the section 7.7

6.7 WB data/address bus width

The vme64x is a generic core which can be interfaced with a 32 bits or 64 bits WB slave and also the address width can be 64 bits or less.
The vme64x core is not an independent core, indeed in the WB side we need to interface the vme64x core with a WB bus. In the WB side we can find more than one core, and all these cores/IP are interfaced on the WB bus which makes them communicate. The vme64x and all the WB cores are assembled in an unique TOP_LEVEL. Since the cores are assembled before synthesizing the project, it is sufficient setting the data bus width and address bus width by setting a constant in a package (vme64x_pack.vhd). The synthesizer shall generate the busses as you specified in these constants. Thus it is the task of the digital designer who assembles all the cores to set the properly data/address width.
Some constants you can find in the vme64x_pack.vhd file:
c_width = 32 or 64
c_addr_width = 64 or less
c_CLK_PERIOD = set the clock period in order to read the correct transfer rate.

6.8 Data organization in the VME and WB bus

In this section you can find some examples about the data organization in the VME and WB bus.

Example 1:
WB bus:
- data bus : 64 bits
- address bus : 10 bits
- 8 select lines

Access mode selected: A32 MBLT
Base address : 0xc0000000
Data transfer type : Byte(0-7)
Offset : 0x00000008 (should be a multiple of 8 with the MBLT transfer).

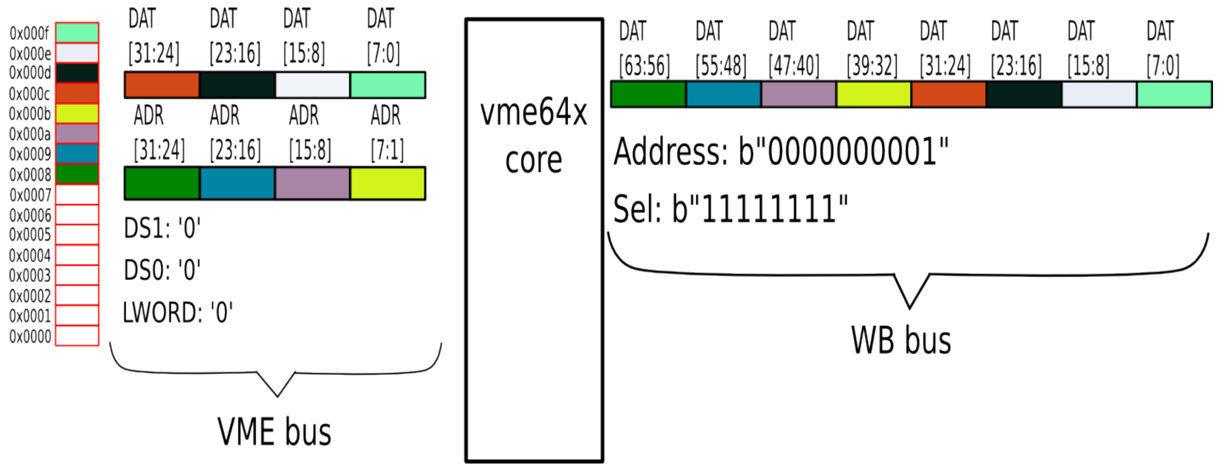


Figure 9: Data organization 1

On the VME bus the data transfer type 64 bits is possible only by multiplexing the data and address lines; the picture shows the data phase.

Example 2:

WB bus:

– data bus : 64 bits
– address bus : 10 bits
– 8 select lines

Access mode selected: A32 S
Base address : 0xc0000000
Data transfer type : Byte(0-3)
Offset : 0x00000008 (should be a multiple of 4 with the D32 transfer).

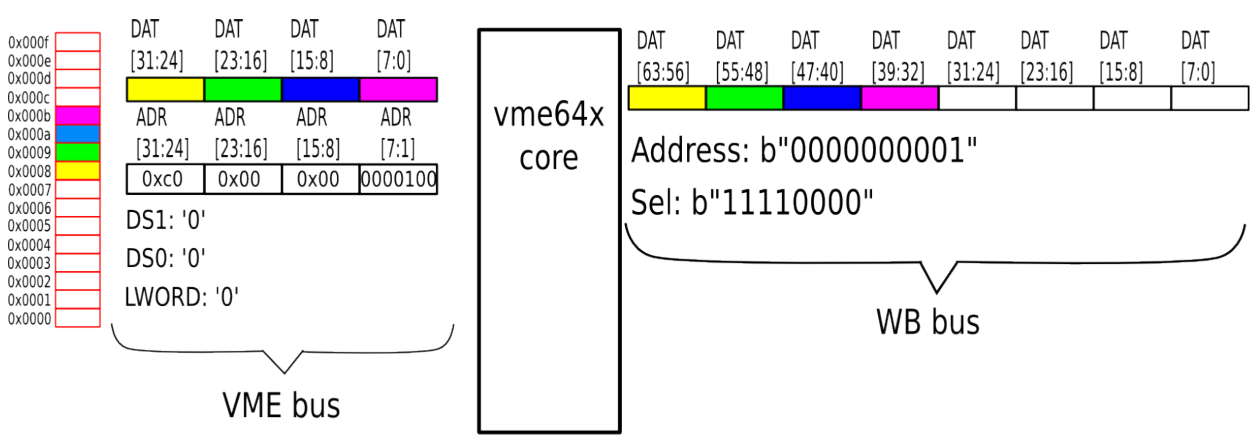


Figure 10: Data organization 2

Example 3:

WB bus:

– data bus : 64 bits
– address bus : 10 bits
– 8 select lines

Access mode selected: A32 S
Base address : 0xc0000000
Data transfer type : Byte(2)
Offset : 0x0000000a.

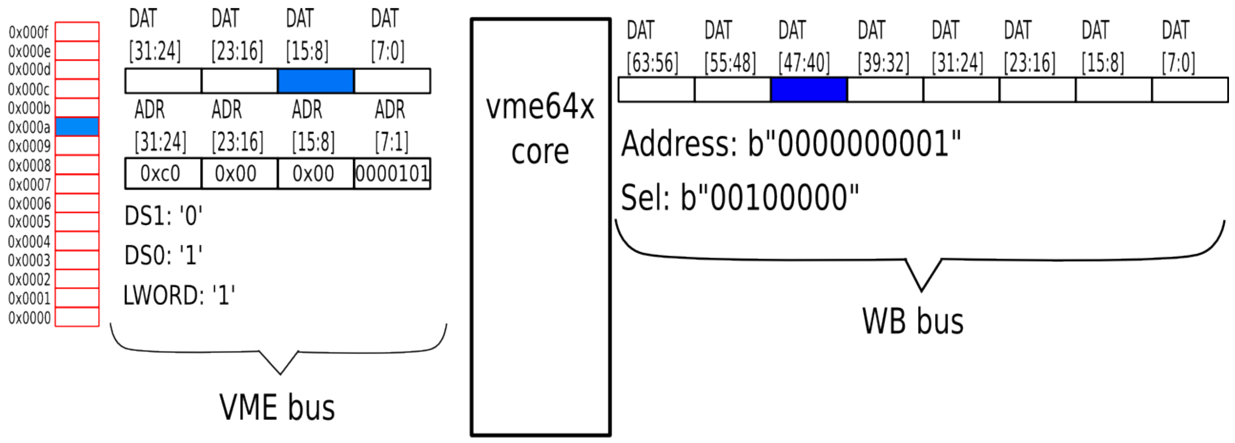


Figure 11: Data organization 3

Example 4:
WB bus:

- data bus : 32 bits
- address bus : 10 bits
- 4 select lines

Access mode selected: A32 S
Base address : 0xc0000000
Data transfer type : Byte(0-3)
Offset : 0x00000008 (should be a multiple of 4 with the D32 transfer).
The WB address is right shifted only of two position if the Wb data bus is 32 bit.

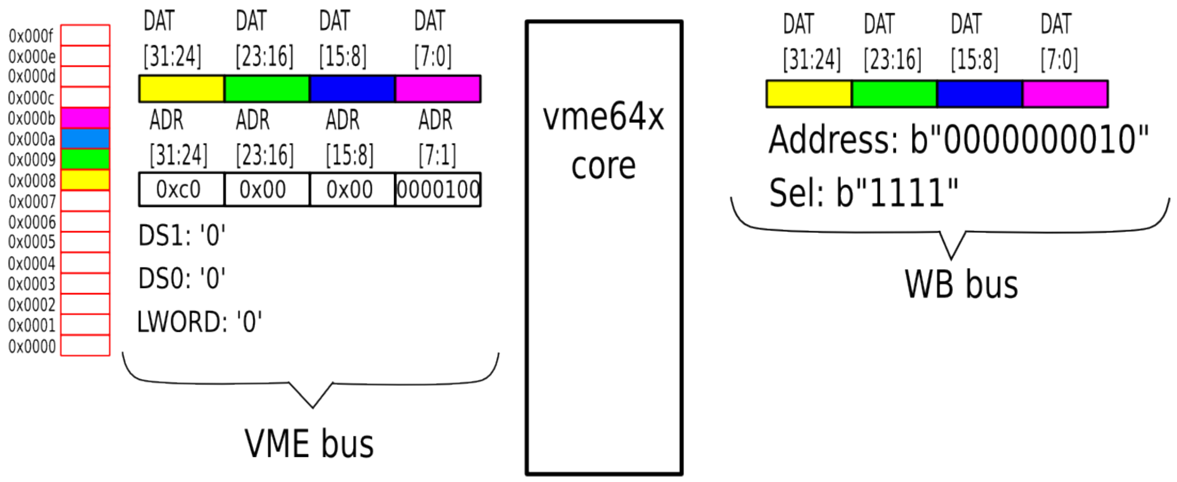


Figure 12: Data organization 4

6.9 vme64x core I/O ports

Clock:

clk_i	: in	std_logic;

VME signals:

VME_AS_n_i	: in	std_logic;
VME_RST_n_i	: in	std_logic;
VME_WRITE_n_i	: in	std_logic;
VME_AM_i	: in	std_logic_vector(5 downto 0);
VME_DS_n_i	: in	std_logic_vector(1 downto 0);
VME_GA_i	: in	std_logic_vector(5 downto 0);
VME_BERR_o	: out	std_logic;
VME_DTACK_n_o	: out	std_logic;
VME_RETRY_n_o	: out	std_logic;
VME_LWORD_n_i	: in	std_logic;
VME_LWORD_n_o	: out	std_logic;
VME_ADDR_i	: in	std_logic_vector(31 downto 1);
VME_ADDR_o	: out	std_logic_vector(31 downto 1);
VME_DATA_i	: in	std_logic_vector(31 downto 0);

```
VME_DATA_o          :  out    std_logic_vector(31 downto 0);
VME_IRQ_n_o         :  out   std_logic_vector(6 downto 0);
VME_IACKIN_n_i      :  in     std_logic;
VME_IACK_n_i        :  in     std_logic;
VME_IACKOUT_n_o  :  out   std_logic;
VME bus transceivers :
VME_DTACK_OE_o    :  out   std_logic;
VME_DATA_DIR_o     :  out   std_logic;
VME_DATA_OE_N_o   :  out   std_logic;
VME_ADDR_DIR_o     :  out   std_logic;
VME_ADDR_OE_N_o  :  out   std_logic;
VME_RETRY_OE_o    :  out     std_logic;
WishBone signals:
DAT_i          :  in      std_logic_vector(63 downto 0);
DAT_o          :  out    std_logic_vector(63 downto 0);
ADR_o          :  out    std_logic_vector(63 downto 0);
CYC_o          :  out    std_logic;
ERR_i          :  in      std_logic;
RTY_i          :  in      std_logic;
SEL_o          :  out    std_logic_vector(7 downto 0);
STB_o          :  out    std_logic;
ACK_i          :  in      std_logic;
WE_o           :  out    std_logic;
STALL_i        :  in      std_logic;
IRQ Generator signals:
INT_ack        : out     std_logic;
IRQ_i          : in      std_logic;
reset_o        : out     std_logic;       – asserted when '1'
Debug (8 leds on the board):
debug          : out   std_logic_vector(7 downto 0);
```

6.10 Interconnection Diagram

This diagram is an example showing you how you can connect the vme64x core to your WB Slave application without interrupt generator:

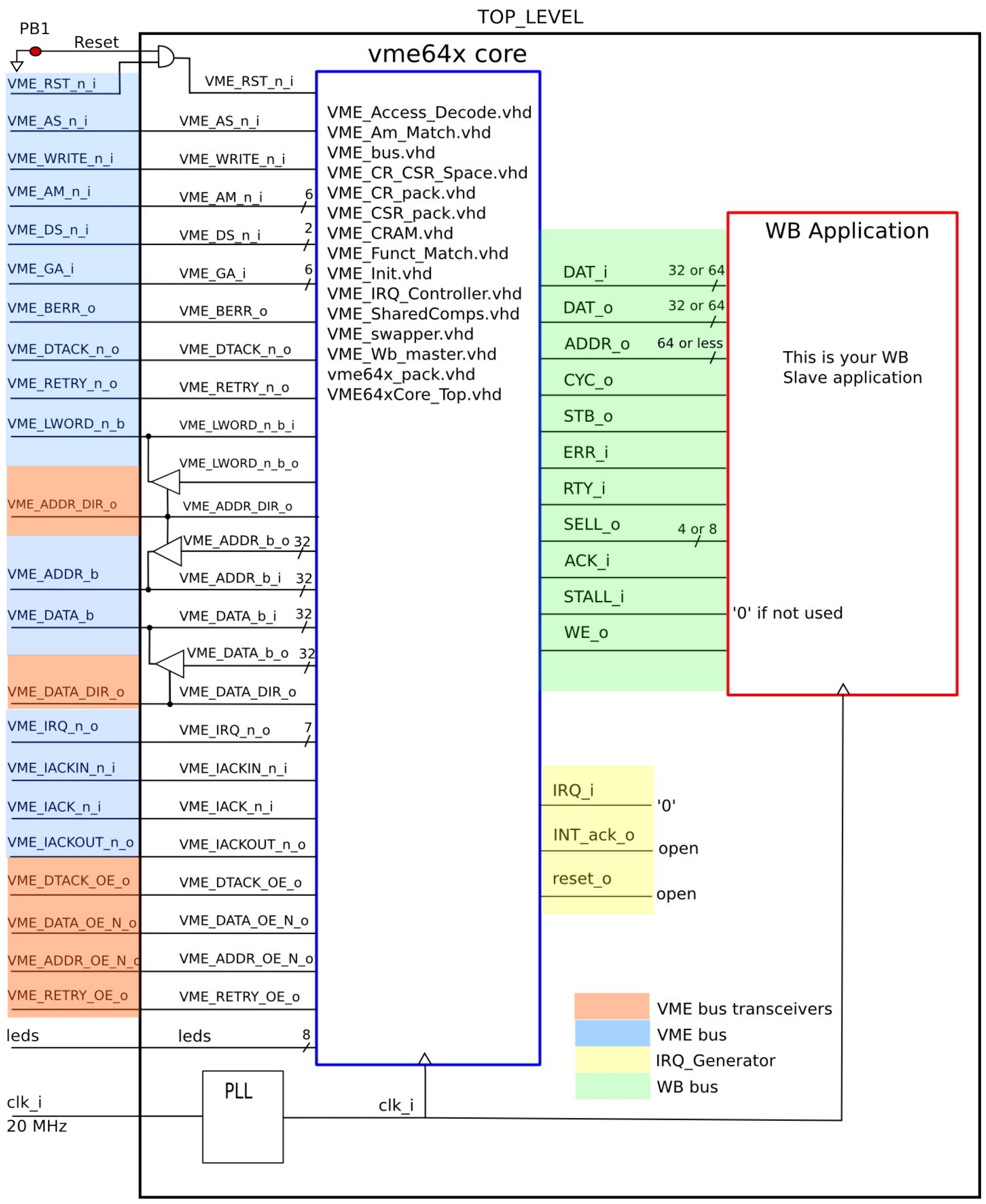


Figure 13: Interconnection Diagram 1

The WB slave Data Bus can be 32 bit or 64 bit; before synthesizing the project set the following constants in the vme64x_pack.vhd file:
c_width: this is the WB Data Bus width (must be 32 or 64)
c_addr_width: this is the WB Address Bus width (must be 64 or less).

If you need the Interrupter you can insert the IRQ Generator as shown in the following Block Diagram:

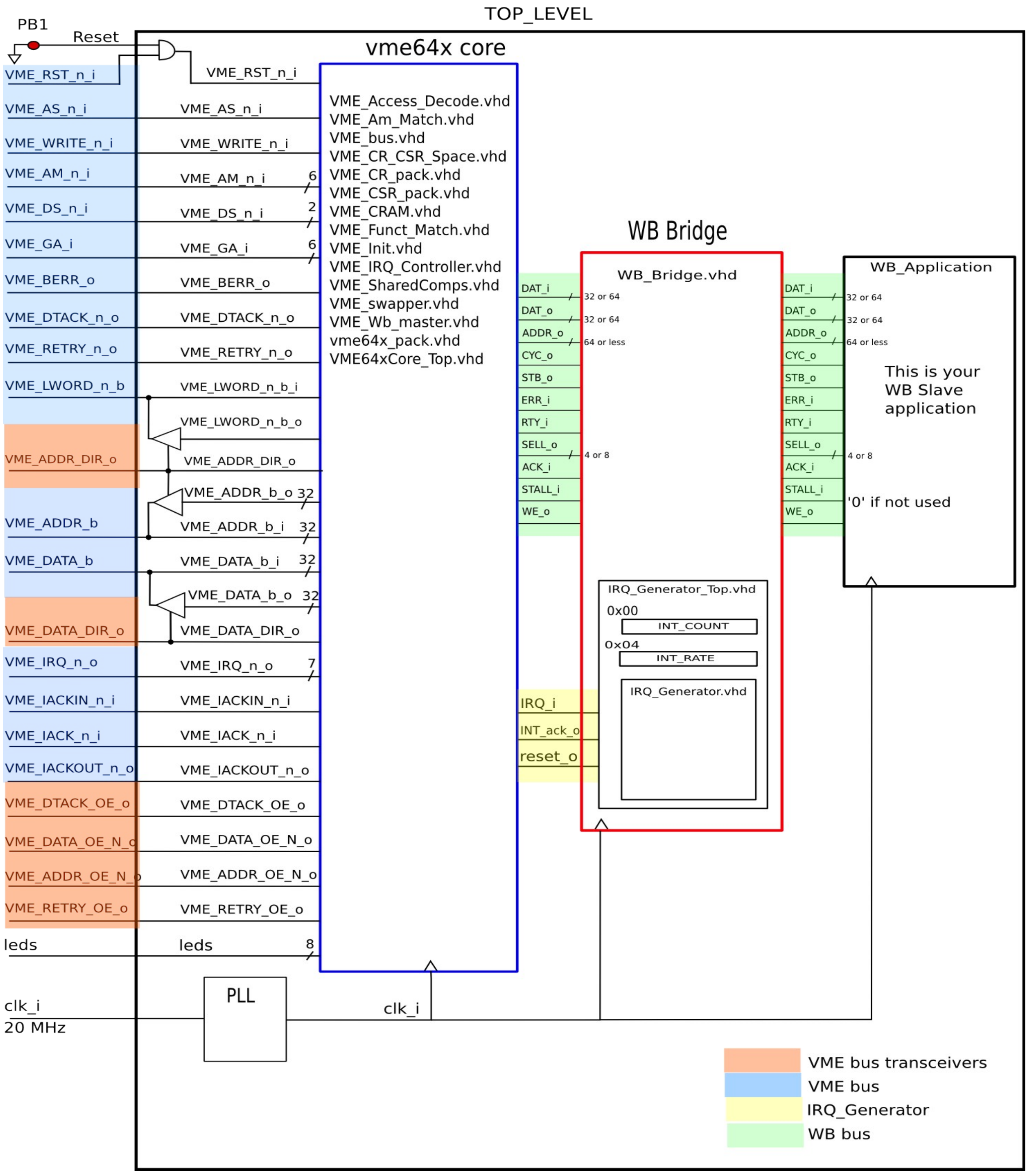


Figure 14: Interconnection Diagram 2

The next Interconnection Diagram shows the WB Slave that is used to debug the vme64x core and to develop the Interrupter.

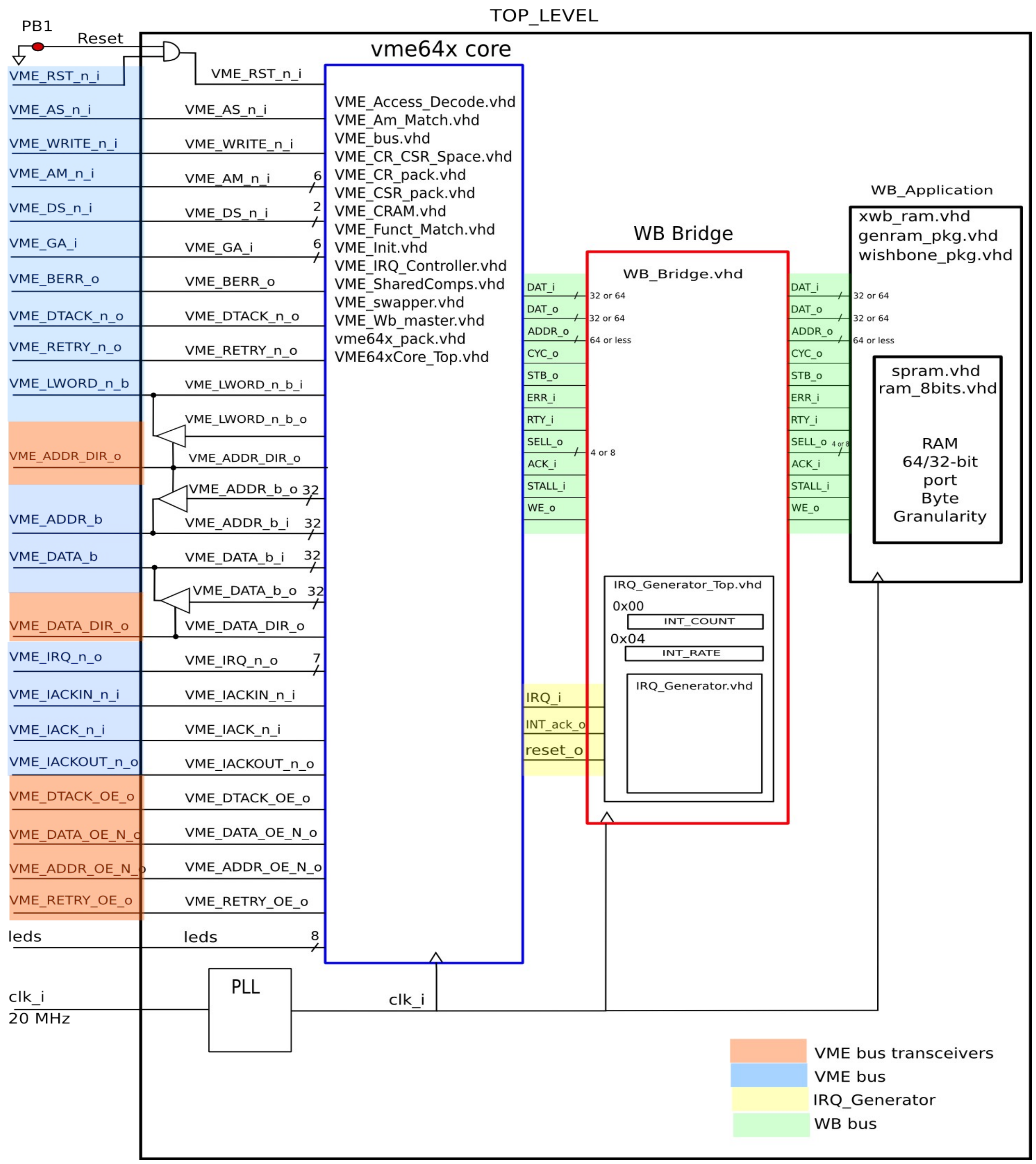


Figure 15: Interconnection Diagram 3

6.11 Test with the wbgen2

An interesting test executed on the vme64x core is to check if it is compatible with a WB slave automatically generated with the wbgen2 tool.

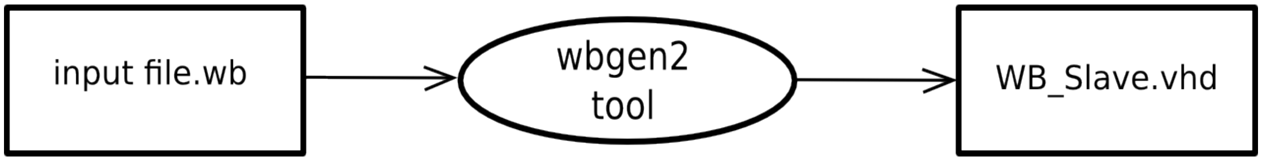


Figure 16: WBgen2

This is an example of an input file:

```
peripheral {
name = "Wishbone Slave port";
description = "RAM and Interrupt Generator";
hdl_entity = "wishbone_port_slave";
c_prefix = "WBslave";
hdl_prefix = "WBslave";

 ram {
   name="Memory";
   prefix = "RAM";
   size = 256;
   width = 32;
   access_bus = READ_WRITE;
   access_dev = READ_WRITE;
  };

  reg {
   name = "INT_COUNT";
   description = "interrupt counter";
   prefix = "reg";
   field{
      name = "INT_COUNT";
      description = "interrupt counter";
      prefix = "Int_count";
      type = SLV;
      size = 32;
      align = 0;
      access_bus = READ_ONLY;
      access_dev = WRITE_ONLY;
   };
};

 reg {
   name = "INT_RATE";
   description = "interrupt rate";
   prefix = "reg";
   field{
      name = "INT_RATE";
      description = "interrupt rate";
      prefix = "Int_rate";
      type = SLV;
      size = 32;
      align = 32;
      access_bus = READ_WRITE;
      access_dev = READ_ONLY;
   };
  };
irq{
  name = "Rising edge IRQ";
  description = "Rising edge triggered IRQ line";
  prefix = "irq0";
  trigger = EDGE_RISING;
  };

irq{
  name = "Rising edge IRQ";
  description = "Falling edge triggered IRQ line";
  prefix = "irq1";
  trigger = EDGE_FALLING;
  };

};
```

With this input file we can generate a WB slave similar to the WB slave showed in the interconnection diagram in the section 7.10.

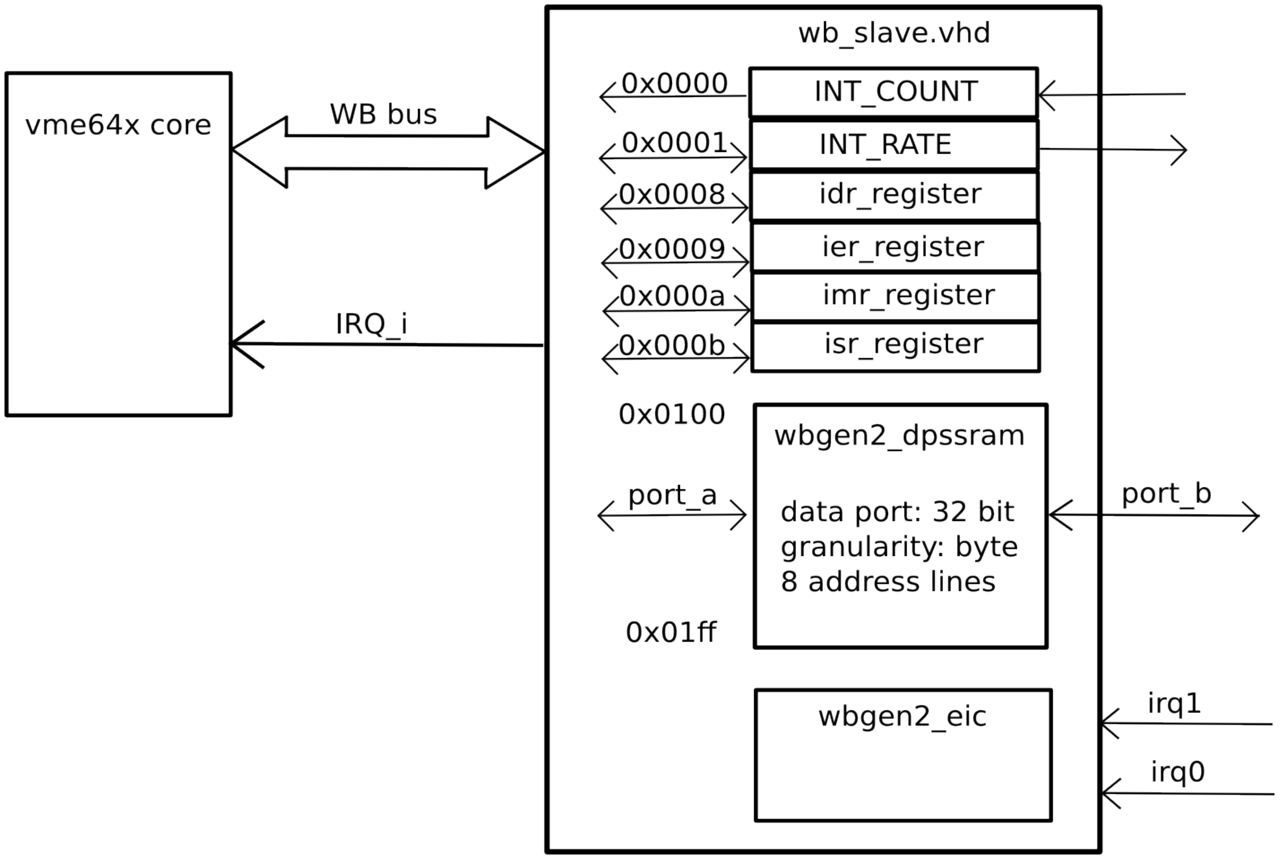


Figure 17: Wb_slave generated with wbgen2

Please note that these are the address in the WB side; in the VME side all these addresses shall be left shifted two position; eg all the correct addresses for accesing the ram memory are the addresses between 0x400 and 0x3ff.
If the input file contains some interrupt requests (irq 0 and irq1 in this case) the wbgen2 tool will generate the idr, ier, imr, isr registers and the wbgen2_eic component.

We can have up to 32 irq lines; each irq line correspond to a bit in the idr, ier, imr and isr registers. Let's see what these registers are.
A write operation on the ier_register (interrupt enable register) shall enable the irq input lines corresponding to the bit set to '1' in the ier register. The VME Master can check which irq lines are enabled by reading the imr register (mask register). In a similar way, with a write operation on the idr register the VME Master can disable the irq lines.
The irq lines can generate an interrupt request on the rising/falling edge, or on the level. If active on the level you should be careful because if the irq line is asserted for long time you will produce a train of interrupt request.
This structure has no queues, thus we suggest to implement at least an interrupt counter which will be incremented all times the wb_slave receives an interrupt request, and the VME Master can check if it is missing interrupt.
The status register called isr register in the block diagram above, informs about the presence of pending interrupt.
The wb_slave has more than one irq line and only one output called IRQ_i connected to the vme64x core. It means that all the irq have the same INT_LEVEL and INT_VECTOR (or STATUS/ID) which are stored in the CSR space. Thus when the VME Master executes the service routine, it shall:

- Read the status register in order to know which peripheral needs service. Note that if more peripherals are requesting service at the same time, in the status register more bits are asserted. A software arbiter determines which interrupt pending can obtain service. Eg the Master can read the status register from the lsb to the msb and it means that the irq0 corresponding to the bit 0 in the status register has the priority.
- After the software selects the peripheral to be served, it executes the properly routine.
- Before exit the software should read the interrupt counter to check if it is missing interrupts and clear the just served interrupt pending by writing '1' the corresponding bit in the isr register.

An example is showed in the following image where the irq0 is active on the rising edge and the irq1 on the falling edge like in the input file in the page before.

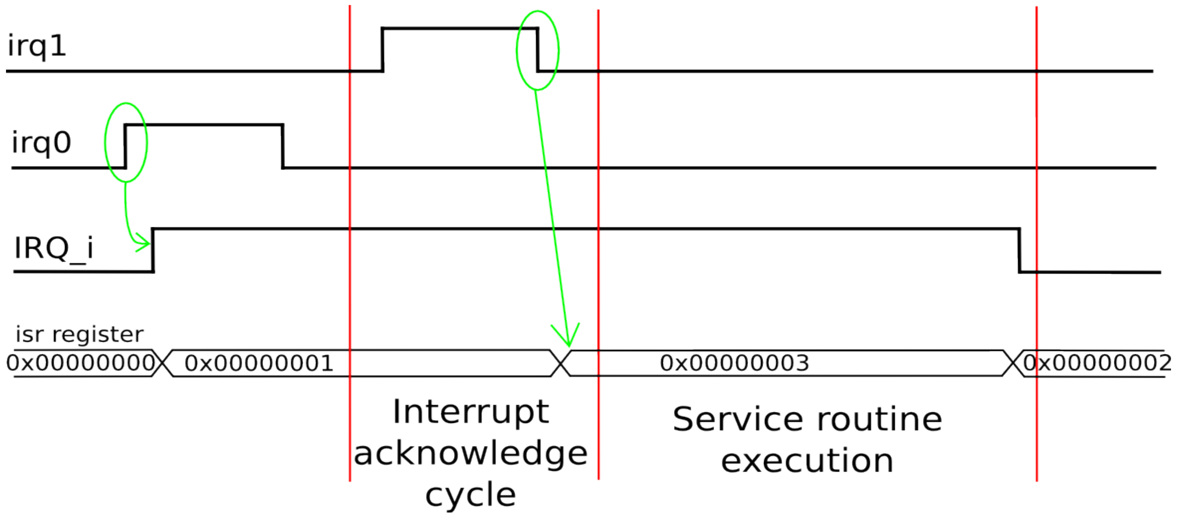


Figure 18: Example irq lines

Note that the IRQ_i line is not released until the VME master clears the interrupt pending just served; indeed is important not to separate the Interrupt acknowledge cycle and the execution of the corresponding interrupt service routine.

The Interrupt controller in the vme64x core works on the IRQ_i rising edge.
More details about the wbgen2 can be found here:

http://www.ohwr.org/projects/wishbone-gen

7 References

[1] The VMEbus Specification ANSI/VITA October 1987
[2] VME64 Extensions ANSI/VITA 1.1 1997
[3] VME64 ANSI/VITA 1 1994
[4] CMS Internal Note
Design rules for custom VME modules in CMS, CMS IN 2004/005
[5] Wishbone System-on-chip (SoC) Interconnection Architecture for
Portable IP Cores, Revision B4
[6] ANSI/VITA 1.5-2003 2eSST