Python guidelines for vme64x core:
Go to
/user/dcobas/cage/pyvme/pylib

$ export LD_LIBRARY_PATH=/user/dcobas/cage/pyvme/pylib

Now you can access python:

$ python

You enter the python interprete
Before starting the test you need to import the pyvmelib module:

>>> import pyvmelib

## First test: CR/CSR space access:

| | | |
|---|---|---|
| Begin CR | : | 0x000000 |
| End CR | : | 0x000FFF |
| Begin CRAM | : | 0x001000 |
| End CRAM | : | 0x07FBFF |
| Begin CSR | : | 0x07FC00 |
| End CSR | : | 0x07FFFF |

How to access the CR/CSR space:

- AM = 0x2f
- Address width = 24 bit
- bits 23 downto 19 = not(VME_GA) = Slot number
- eg. Slot 8:

>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=512*1024)
The parameter data_width can be also 32; In the CR/CSR space only the byte 3 locations are implemented so the offsett should be like the following examples:

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)

If you access to byte0, byte1 or byte2 (these locations are not implemented) the output is 0!

Test CRAM:
We will use the byte access so the offset parameter match the register address.

- The Master reads the CRAM_OWNER register:
  >>> m.read(offset=0x7FFF3, num=1, width=8)
- If 0 the Master can write his ID in this register

>>> m.write(offset=0x7FFF3, values= ID, width=8)

- If the Master can read his ID it means that it has the ownership of the CRAM
  >>> m.read(offset=0x7FFF3, num=1, width=8)
- If others masters try to write their ID when the CRAM_OWNER register contain a non zero values the write will not take place.
  >>> m.write(offset=0x7FFF3, values= ID2, width=8)
- If you read now the CRAM OWNER reg the value is still ID, not ID2!!
  >>> m.read(offset=0x7FFF3, num=1, width=8)
- If the Master reads the BIT_SET_REG the CRAM owned flag is asserted:
  >>> m.read(offset=0x7FFFB, num=1, width=8)
  you should read 0x14
- The Master writes one location between the Begin CRAM and End CRAM:
  >>> m.write(offset=0x03007, values= 5, width=8)
- you should read back 5:
  >>> m.read(offset=0x03007, num= 1, width=8)
- The Master releases the CRAM by asserting the corresponding bit in the BIT_CLR_REG:
  >>> m.write(offset=0x7FFF7, values= 4, width=8)
- The CRAM_OWNER reg is automatically resetted to 0:
  >>> m.read(offset=0x7FFF3, num= 1, width=8)
  you should read 0.
- Also the flag in the BIT_SET_REG is resetted:
  >>> m.read(offset=0x7FFFB, num= 1, width=8)
  You should read 0x10 (→ module enable)

All the other registers in the CSR space can be read/write as explained in the ANSI/VITA 1.1-1997 VME64 Extensions.

Other important flags implemented in the vme64x core:
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 had 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.

## Second Test: WB access

FUNC0 : For address width 32-bit:
A32_S : AM = 0x09
A32_S sup : AM = 0x0d
A32_BLT : AM = 0x0b
A32_BLT sup : AM = 0x0f
A32_MBLT : AM = 0x08

A32_MBLT sup : AM = 0x0c

1) eg. A32_S, base_address = 0xc0 = 192

- The Master writes the FUNC0_ADER register 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) ←FUNC0_ADER3
  >>> m.write(offset=0x7FF67, values= 0, width=8) ←FUNC0_ADER2
  >>> m.write(offset=0x7FF6B, values= 0, width=8) ←FUNC0_ADER1
  >>> m.write(offset=0x7FF6F, values= 36, width=8) ←FUNC0_ADER0 = AM & "00"
  (the last write operation is optional indeed the DFS bit is 0)
- **After power-up or reset the module is disabled**. The Master writes to the Bit Set Register with bit 4 set to enable the module and to access the WB side.
  >>> m.write(offset=0x7FFFB, values= 16, width=8).
- The master access the WB bus with data width = 32-bit
  >>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size= "size of your memory")
  >>> m.write(offset=multiple of 4, values=.., width=32)
  >>> m.read(offset= multiple of 4, num= 1, width=32)

- The master access the WB bus with data_width = byte:
  >>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=16, size= "size of your memory")
  >>> m.write(offset=0x00, values=.., width=8) --byte0
  …..
  >>> m.write(offset=0x07, values=.., width=8) --byte7
  >>> m.read(offset=0x00, num= 1, width=8) --byte0
  …..
  >>> m.read(offset=0x07, num= 1, width=8) --byte7

- The master access the WB bus with data_width = word:
  >>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=16, size= "size of your memory")
  >>> m.write(offset=0x00, values=.., width=16) --byte0 and byte1
  >>> m.write(offset=0x02, values=.., width=16) --byte2 and byte3
  >>> m.write(offset=0x04, values=.., width=16) --byte4 and byte5
  >>> m.write(offset=0x06, values=.., width=16) --byte6 and byte7

- If the WB Data bus is 64 bit you can execute the same steps but remember to set to 0 the following flag in the CSR space:
  >>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size= 512*1024)
  >>> m.write(offset=0x7FF33, values=0, width=8)

2) eg A32_BLT, base_address = 0xc0 = 192

- The Master writes the FUNC0_ADER register 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) ←FUNC0_ADER3
>>> m.write(offset=0x7FF67, values= 0, width=8) ←FUNC0_ADER2
>>> m.write(offset=0x7FF6B, values= 0, width=8) ←FUNC0_ADER1
>>> m.write(offset=0x7FF6F, values= 44, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)

- You can prepare a buffer:
  >>> from ctypes import*
  >>>hello=create_string_buffer(“012345671234567”)
  this buffer is 16 bytes; each characters will be coded according to the ascii code; the last characters is '\0'
  >>>len(hello)
  should be 16 in this case
- The Master writes the buffer in the WB memory:
  >>>pyvmelib.dma_write(am=0x0b, address= 0xc0000100,data_width=32,buffer=hello)
- The Master reads the buffer in the WB memory:
  >>>pyvmelib.dma_read(am=0x0b, address= 0xc0000100,data_width=32,size=4*4)

With the A32_BLT access, only data transfer 32 bit is supported.
The address and the size must be a multiple of 4
If the WB Data bus is 64 bit remember to set to 0 the corresponding bit if not yet done.

3) eg A32_MBLT, base_address = 0xc0 = 192
   This is a data 64 bit mode so you **don't have to use it** with the WB Data Bus 32 bit flag asserted.

   The Master writes the FUNC0_ADER register 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) ←FUNC0_ADER3
   >>> m.write(offset=0x7FF67, values= 0, width=8) ←FUNC0_ADER2
   >>> m.write(offset=0x7FF6B, values= 0, width=8) ←FUNC0_ADER1
   >>> m.write(offset=0x7FF6F, values= 32, width=8) ←FUNC0_ADER0 = AM & “00”
   (the last write operation is optional because the DFS bit is 0)

- You can prepare a buffer:
  >>> from ctypes import*
  >>>hello=create_string_buffer(“33333333555555557777777711111111”)
  this buffer is 32 bytes; each characters will be coded according to the ascii code; the last characters is '\0'
  >>>len(hello)
  should be 32 in this case
- The Master writes the buffer in the WB memory:
  >>>pyvmelib.dma_write(am=0x08, address= 0xc0000100,data_width=32,buffer=hello)
- The Master reads the buffer in the WB memory:
  >>>pyvmelib.dma_read(am=0x08, address= 0xc0000100,data_width=32,size=8*4)

With the A32_MBLT access, only data transfer 64 bit is supported.

The address and the size must be a multiple of 8.

FUNC1 : For address width 24-bit:
A24_S : AM = 0x39
A24_S sup : AM = 0x3d
A24_BLT : AM = 0x3b
A24_BLT sup : AM = 0x3f
A24_MBLT : AM = 0x38
A24_MBLT sup : AM = 0x3c

1) eg. A24_S, base_address = 0xc0 = 192

- The Master writes the FUNC1_ADER register in the CSR space:
  >>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size= 512*1024)
  >>> m.write(offset=0x7FF73, values= 0, width=8) ←FUNC1_ADER3
  >>> m.write(offset=0x7FF77, values= 192, width=8) ←FUNC1_ADER2
  >>> m.write(offset=0x7FF7B, values= 0, width=8) ←FUNC1_ADER1
  >>> m.write(offset=0x7FF7F, values= 228, width=8) ←FUNC1_ADER0 = AM & "00"
  (the last write operation is optional indeed the DFS bit is 0)

- The master access the WB bus with data width = 32-bit
  >>> m = pyvmelib.Mapping(am=0x39, base_address=0xc00000, data_width=32, size= "size of your memory")
  >>> m.write(offset=multiple of 4, values=.., width=32)
  >>> m.read(offset= multiple of 4, num= 1, width=32)

- The master access the WB bus with data_width = byte:
  >>> m = pyvmelib.Mapping(am=0x39, base_address=0xc00000, data_width=16, size= "size of your memory")
  >>> m.write(offset=0x00, values=.., width=8) --byte0
  …..
  >>> m.write(offset=0x07, values=.., width=8) --byte7
  >>> m.read(offset=0x00, num= 1, width=8) --byte0
  …..
  >>> m.read(offset=0x07, num= 1, width=8) --byte7

- The master access the WB bus with data_width = word:
  >>> m = pyvmelib.Mapping(am=0x39, base_address=0xc00000, data_width=16, size= "size of your memory")
  >>> m.write(offset=0x00, values=.., width=16) --byte0 and byte1
  >>> m.write(offset=0x02, values=.., width=16) --byte2 and byte3
  >>> m.write(offset=0x04, values=.., width=16) --byte4 and byte5
  >>> m.write(offset=0x06, values=.., width=16) --byte6 and byte7

- If the WB Data bus is 64 bit you can execute the same steps but remember to set to 0 the following flag in the CSR space:

>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size= 512*1024)
>>> m.write(offset=0x7FF33, values=0, width=8)

To access with A24_BLT and A24_MBLT modes you can do the same steps as shown before for the A32_BLT and A32_MBLT, remember only to change the AM and to write the base address in the FUNC1_ADER2 instead of in the FUNC1_ADER3 because now the address width is 24-bit.

FUNC2 : For address width 16-bit:
A16_S : AM = 0x29
A16_S sup : AM = 0x2d

1) eg. A16_S, base_address = 0xc0 = 192

- The Master writes the FUNC2_ADER register in the CSR space:
  >>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size= 512*1024)
  >>> m.write(offset=0x7FF83, values= 0, width=8) ←FUNC2_ADER3
  >>> m.write(offset=0x7FF87, values= 0, width=8) ←FUNC2_ADER2
  >>> m.write(offset=0x7FF8B, values= 192, width=8) ←FUNC2_ADER1
  >>> m.write(offset=0x7FF8F, values= 164, width=8) ←FUNC2_ADER0 = AM & “00”
  (the last write operation is optional indeed the DFS bit is 0)

- The master access the WB bus with data width = 32-bit
  >>> m = pyvmelib.Mapping(am=0x29, base_address=0xc000, data_width=32, size= “size of your memory”)
  >>> m.write(offset=multiple of 4, values=.., width=32)
  >>> m.read(offset= multiple of 4, num= 1, width=32)
  Of course the size of the memory should be compatible with the address width.

- The master access the WB bus with data_width = byte:
  >>> m = pyvmelib.Mapping(am=0x29, base_address=0xc000, data_width=16, size= “size of your memory”)
  >>> m.write(offset=0x00, values=.., width=8) --byte0
  …..
  >>> m.write(offset=0x07, values=.., width=8) --byte7
  >>> m.read(offset=0x00, num= 1, width=8) --byte0
  …..
  >>> m.read(offset=0x07, num= 1, width=8) --byte7

- The master access the WB bus with data_width = word:
  >>> m = pyvmelib.Mapping(am=0x29, base_address=0xc000, data_width=16, size= “size of your memory”)
  >>> m.write(offset=0x00, values=.., width=16) --byte0 and byte1
  >>> m.write(offset=0x02, values=.., width=16) --byte2 and byte3
  >>> m.write(offset=0x04, values=.., width=16) --byte4 and byte5
  >>> m.write(offset=0x06, values=.., width=16) --byte6 and byte7

- If the WB Data bus is 64 bit you can execute the same steps but remember to set to 0 the

following flag in the CSR space:
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size=
512*1024)
>>> m.write(offset=0x7FF33, values=0, width=8)

## Swap test:

The vme64x core has been provided of a swapper component so is possible write/read the memory with dma ON and read/write the memory with dma OFF.

The Master writes the FUNC0_ADER register 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) ←FUNC0_ADER3
>>> m.write(offset=0x7FF67, values= 0, width=8) ←FUNC0_ADER2
>>> m.write(offset=0x7FF6B, values= 0, width=8) ←FUNC0_ADER1
>>> m.write(offset=0x7FF6F, values= 44, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)

Generate one buffer:

>>>from ctypes import*
>>>hello=create_string_buffer("012345671234567")

The VME Master write the memory with dma ON; A32_BLT access

>>>pyvmelib.dma_write(am=0x0b,address=0xc0000008,data_width=32,buffer=hello)

Now in the memory we have:
loc1: 0x3031323334353637
loc2: 0x31323334353637'/0'

The Master start to read back the memory with A32_S (dma OFF):

>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=8, size=
512*1024)
>>> m.write(offset=0x7FF6F, values= 36, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)
A32_S D32:
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size=24)
>>>m.read(offset=0x08, num=1, width=32)
you can read:
hex(858927408) = 0x33323130

We are not reading the correct value! Indeed with the single access the dma is off and the data are swapped. To read the correct value we should set properly the MBLT_Endian register:
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=
512*1024)
The Master writes the MBLT_Endian reg → **Swap word + Swap byte select**
>>> m.write(offset=0x7FF53, values=3, width=32)

Now we can access again the memory:
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size=24)
>>> m.read(offset=0x08, num=1, width=32)
We can read:
hex(808530483) = 0x30313233 → CORRECT! The Swap is working.

If the access is D16, to read the correct values we should set the MBLT_Endian reg. Like this:
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size= 512*1024)
The Master writes the MBLT_Endian reg → **Swap word select**
>>> m.write(offset=0x7FF53, values=2, width=32)

Now we can access again the memory:
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=16, size=24)
>>> m.read(offset=0x08, num=1, width=16)
We should read:
0x3031 .

If the access is D08 of course we don't need to swap data.

The vme64x core allows to swap the data also during write operation and read with dma OFF eg:
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=512*1024)
>>> m.write(offset=0x7FF53, values=3, width=32) → **Swap word + Swap byte select**
>>> m.write(offset=0x7FF6F, values= 44, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)
>>>pyvmelib.dma_write(am=0x0b,address=0xc0000008,data_width=32,buffer=hello)
Now in the memory we have:
loc1: 0x3332313037363534
loc2: 0x34333231'/0'373635
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=512*1024)
>>> m.write(offset=0x7FF53, values=0, width=32) → **Swap data off**
>>> m.write(offset=0x7FF6F, values= 36, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size=24)
>>>m.read(offset=0x08, num=1, width=32)
you can read:
hex(808530483) = 0x30313233 CORRECT!

Eg. Use the Byte swap:
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16,size=512*1024)
>>> m.write(offset=0x7FF53, values=1, width=32) → **Byte Swap**
>>> m.write(offset=0x7FF6F, values= 36, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)
>>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=16, size=24)
….The Master write the memory with data transfer width = D16 (dma OFF)
>>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16,size=512*1024)
>>> m.write(offset=0x7FF53, values=0, width=32) → **Swap data off**
>>> m.write(offset=0x7FF6F, values= 44, width=8) ←FUNC0_ADER0 = AM & “00”
(the last write operation is optional indeed the DFS bit is 0)
The Master read the memory in BLT mode (dma ON)
>>>pyvmelib.dma_read(am=0x0b,address=0xc0000008,data_width=32,size=4*4)

# Interrupter test

To generate interrupts you must insert the IRQ Generator in your WB Application. The address 0x00 and 0x04 should be reserved to implement the INTERRUPT COUNTER and INTERRUPT RATE.

1) Access the CSR space and set the IRQ_Vector register. Each Slot must have its unique IRQ Vector.
   >>> m = pyvmelib.Mapping(am=0x2f, base_address=0x400000, data_width=16, size=512*1024)
   >>> m.write(offset=0x7FF5F, values=137, width=32) (137 = 0x87)
2) Access the WB Memory and set the Interrupt rate:
   >>> m = pyvmelib.Mapping(am=0x09, base_address=0xc0000000, data_width=32, size=24)
   >>> m.write(offset=0x04, values=4194304, width=32) (4194304 = 0x400000)
   Interrupt every 200 ms.
3) The board will generate a new interrupt request only after the Master read the Interrupt Counter register in the location 0x00.
   >>> m.read(offset=0x00, num=1, width=32)
4) As soon as the Master receives an interrupt request and the Slave acknowledge the interrupt cycle by sending the IRQ Vector and asserting the DTACK* line, it should accesses the Interrupt Counter register and read the count; if the Master read:
   1, 4, 7... it means that it is missing interrupts and the Interrupt rate should be slow down.
   If the Master read 1,2,3,4 …. it means that it isn't missing interrupts and it can increase the Interrupt rate if necessary.
5) Of course in the crate we can have more than 1 board generating Interrupt requests ----> the Interrupt rate limit (to not miss interrupts) of each board will be slow down if the number of boards that are generating interrupts increase!
6) To turn off the Interrupter the Master can write 0 in the Interrupt rate register loc 0x04.