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)

- 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 only 32 bit you can execute the same steps but remember to set 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=1, 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 only 32 bit remember to set the corresponding bit if not yet done.

3) eg A32_MBLT, base_address = 0xc0 = 192
   This is a data 64 bit mode so don't use 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(“3333333355555555777777771111111”)
  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 only 32 bit you can execute the same steps but remember to set 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=1, 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 only 32 bit you can execute the same steps but remember to set 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=1, width=8)
```

Swap test:

(This test will be provided in short-term)

Interrupter test

(This test will be provided in short-term)