VciMultiAhci

1) Functional Description

This component emulates a multi-channels disk controller with VCI interface. Each channel[k] can access a different physical disk, modeled as a different file[k] belonging to the host system, and containing a complete disk image. Each channel[k] can perform data transfers between file[k] and a buffer in the physical memory of the virtual system. he number of supported channels, the file name(s), the VCI burst size, and the block size are hardware parameters,defined as constructors parameters. The number of channels cannot be larger than 8. The burst size must be a power of 2 between 8 and 64 bytes. The block size must be a power of 2 between 128 and 4096 bytes.

According to the AHCI specification, each channel[k] controller uses a private *Command List* that is handled as a software FIFO. For each channel[k], the *Command List* can register up to 32 *read* or *write* commands, that are handled in pseudo-parallelism by the channel controller. This VCI component has a DMA capability, and use it to access the *Command List* and to transfer the data to or from memory.

On the VCI side, it supports both 32 bits and 64 bits data words, and up to 64 bits address width.

For each channel, a single IRQ[k] can be (optionally) asserted as soon as at list one command in the Command List is completed. WARNING: the IRQ[k] is associated to a specific channel, but not to a specific command.

This hardware component checks for segmentation violation, and can be used as a default target.

2) Addressable registers

Each channel[k] contains seven 32 bits registers:

- **HBA** (read/write)

Physical address of the source (or destination) buffer in SoC memory.

- **BLOCK_DEVICE_COUNT** (read/write)

Number of blocks to be transfered.

- **BLOCK_DEVICE_LBA** (read/write)

Logical Base Address (index of the first block in the block device)

- **BLOCK_DEVICE_OP** (write only)

Type of operation, writing here initiates the operation.This register goes back to BLOCK_DEVICE_NOOP when operation is finished. The following operations codes are defined:

`BLOCK_DEVICE_NOOP`	No operation
`BLOCK_DEVICE_READ`	Transfer from block device to memory
`BLOCK_DEVICE_WRITE`	Transfer from memory to block device

- **BLOCK_DEVICE_STATUS** (read only)

State of the transfer. Reading this register while not busy resets its value to IDLE, and acknowledge the IRQ. Value may be one of :

```
BLOCK_DEVICE_IDLE
BLOCK_DEVICE_BUSY
BLOCK_DEVICE_READ_SUCCESS
BLOCK_DEVICE_WRITE_SUCCESS
BLOCK_DEVICE_READ_ERROR
BLOCK_DEVICE_WRITE_ERROR
```

- **BLOCK_DEVICE_IRQ_ENABLE** (read/write)

Boolean enabling the IRQ line

- **BLOCK_DEVICE_SIZE** (read only)

Number of blocks addressable in the block device

- **BLOCK_DEVICE_BLOCK_SIZE** (read only)

Block size (in bytes)

For extensibility issues, you should access this component using globally-defined offsets. You should include file `soclib/block_device.h` from your software, it defines `BLOCK_DEVICE_COUNT`, `BLOCK_DEVICE_READ`, ...

Sample code: Please see reference implementation in source:trunk/soclib/soclib/platform/topcells/caba-vgmn-block_device-mips32el

(add -I/path/to/soclib/include to your compilation command-line)

2) Component definition & usage

source:trunk/soclib/soclib/module/connectivity_component/vci_block_device/caba/metadata/vci_block_device.sd?

See SoclibCc/VciParameters

```
Uses( 'vci_block_device', **vci_parameters )
```

3) CABA Implementation

CABA sources

- interface : source:trunk/soclib/soclib/module/connectivity_component/vci_block_device/caba/source/include/vci_block_device.h
- implementation : source:trunk/soclib/soclib/module/connectivity_component/vci_block_device/caba/source/src/vci_block_device.cpp?

CABA Constructor parameters

```
VciBlockDevice(
     sc_module_name name,   //  Component Name
     const soclib::common::MappingTable &mt, // MappingTable
     const soclib::common::IntTab &srcid,    // Initiator index
     const soclib::common::IntTab &tgtid,    // Target index
     const std::string &filename, // mapped file, may be a host block device
     const uint32_t block_size = 512, // block size in bytes
     const uint32_t latency = 0);  // initial access time (number of cycles)
```

CABA Ports

- **p_resetn** : Global system reset
- **p_clk** : Global system clock
- **p_vci_target** : The VCI target port
- **p_vci_initiator** : The VCI initiator port
- **p_irq** : Interrupt port

4) TLM-DT Implementation

TLM-DT sources

- interface : source:trunk/soclib/soclib/module/connectivity_component/vci_block_device/tlmdt/source/include/vci_block_device
- implementation : source:trunk/soclib/soclib/module/connectivity_component/vci_block_device/tlmdt/source/src/vci_block_device.cpp

TLM-DT Constructor parameters

```
VciBlockDevice(
     sc_module_name name,   //  Component Name
     const soclib::common::MappingTable &mt, // MappingTable
     const soclib::common::IntTab &srcid,    // Initiator index
     const soclib::common::IntTab &tgtid,    // Target index
     const std::string &filename, // mapped file, may be a host block device
     const uint32_t block_size = 512, // block size in bytes
     const uint32_t latency = 0);  // initial access time (number of cycles)
```

TLM-DT Ports

- **p_vci_target** : The VCI target port
- **p_vci_initiator** : The VCI initiator port
- **p_irq** : Interrupt port