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. The number of supported channels, the file name(s), the VCI burst size, and the block size are hardware parameters, defined as constructor 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 both 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) Command List

For each channel, the VciMultiAhci driver must use a software FIFO to register a command: The Command Descriptor array (32 entries) define the Command List. Each Command Descriptor occupies 16 bytes, and contains mainly the physical address of the associated Command Table. A command Descriptor is defined by the following C structure:

```
typedef struct hba_cmd_desc_s  // size = 16 bytes
{
    unsigned char       flag[2];        // WRITE when bit 6 of flag[0] is set
    unsigned char       prdtl[2];       // Number of buffers
    unsigned int        prdbc;          // Number of bytes actually transfered
    unsigned int        ctba;           // Command Table base address 32 LSB bits
    unsigned int        ctbau;          // Command Table base address 32 MSB bits
} hba_cmd_desc_t;
```

3) Command Table

There is one Command Table for each Command descriptor. For a given command, there is one single LBA (Logic Bloc Address) on the block device, coded on 48 bits, but the source (or destination) memory buffer can be split in a variable number of contiguous buffers. Therefore, the Command Table contains two parts: a fixed size Header, defining the LBA, and an array of buffer descriptors containing up to 248 buffer descriptors. A Command Table occupies 4 Kbytes, and is defined by the following C structures:

```
typedef struct hba_cmd_table_s  // size = 4 Kbytes
{
```

```
    hba_cmd_header_t   header;      // contains LBA
    hba_cmd_buffer_t   buffer[248]; // 248 buffers max

} hba_cmd_table_t;

typedef struct hba_cmd_header_s // size = 128 bytes
{
    unsigned int       res0;        // reserved
    unsigned char      lba0;        // LBA 7:0
    unsigned char      lba1;        // LBA 15:8
    unsigned char      lba2;        // LBA 23:16
    unsigned char      res1;        // reserved
    unsigned char      lba3;        // LBA 31:24
    unsigned char      lba4;        // LBA 39:32
    unsigned char      lba5;        // LBA 47:40
    unsigned char      res2;        // reserved
    unsigned int       res[29];     // reserved
} hba_cmd_header_t;

typedef struct hba_cmd_buffer_s // size = 16 bytes
{
    unsigned int       dba;         // Buffer base address 32 LSB bits
    unsigned int       dbau;        // Buffer base address 32 MSB bits
    unsigned int       res0;        // reserved
    unsigned int       dbc;         // Buffer byte count

} hba_cmd_buffer_t;
```

4) Addressable registers

Each channel[k] contains six 32 bits read/write registers:

- **HBA_PXCLB**

32 LSB bits of the Command List physical base address. This address must be aligned on a 16 bytes boundary.

- **HBA_PXCLBU**

32 MSB bits of the Command List array physical address.

- **HBA_PXIS**

Channel status, used for error reporting.

31 30 29 28 24 23 ... 8 7 ... 1 0

- **HBA_PXIE**

Intcommanderrupt enable.

- **HBA_PXCMD**

Boolean : running when non zero

- **HBA_PXCI**

Bit-vector, one bit per command in the Command List. These bits are handled as 32 set/reset flip-flops: set by software when a command ha been posted in Command List / reset by hardware when the command is completed.

- **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.
- 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