

VciMultiAhci

1) Functional Description

This component emulates a multi-channels disk controller respecting the AHCI standard. Each channel[k] can access a different physical disk, modeled as a file (one file per channel) 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 prototype. 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* implemented in the kernel memory. For each channel[k], the associated *Command List* can register up to 32 *read* or *write* concurrent commands, that are handled 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[k], a single IRQ[k] can be (optionally) asserted when one (or several) command(s) in the Command List completes. 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 specific Command List to register a command: The Command Descriptor array (32 entries) define the Command List. Each Command Descriptor occupies 16 bytes, and must be aligned on a 16 bytes boundary. It 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, followed by a variable size array of fixed size buffer descriptors. The Command Table describing one command is defined by the two following C structures:

```

typedef struct hba_cmd_header_s // size = 16 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      res3;          // 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
-- R -- CMD_ID BUFFER_ID ----- D

```

Bit[0] : set by hardware when at least one command has been completed. Bit[30] : set by hardware when an error has been detected in a command. Bit[28:24] : index of the faulty command in command list (set by the hardware). Bit[23:8] : index of the faulty buffer in the faulty command (set by the hardware). When an error is detected for a command, the R bit is set, the channel FSM stops immediately, without handling the remaining commands in the command list, and keep blocked, waiting for a software reset on this PXIS register.

Any write access to this register reset all bits to 0, whatever the VCI WDATA value.

- **HBA_PXIE**

This register enables and disables the IRQ reporting the completion (success or error) of the commands for a given channel. Only 2 bits are used:

```

31 30 29 ..... 1 0

```

3) Command Table

-- R ----- D

Bit 0 : when set, an IRQ is generated when bit0 of AHCI_PXIS is set. Bit 30 : when set, an IRQ is generated when bit30 of AHCI_PXIS is set.

- **HBA_PXCMD**

Boolean : Writing a non zero value activates the polling of the Command List. Writing a zero value makes a soft reset on PXCI, PXIS, PXIE, and PXCMD registers.

- **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 has been posted in Command List / reset by hardware when the command is completed. A write command on this register makes a OR between the VCI WDATA field and the current value of the register.

For extensibility issues, the software drivers must use the mnemonics defined [here?](#) to access this component .

Even if there is only six registers per channel, each channel sub-segment occupies 4K bytes, and the HBA segment must be aligned on a 32 Kbytes boundary.

5) Component definition & usage

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

6) CABA Implementation

CABA sources

- interface :
source:trunk/soclib/soclib/module/connectivity_component/vci_multi_ahci/caba/source/include/vci_multi_ahci.h?
- implementation :
source:trunk/soclib/soclib/module/connectivity_component/vci_multi_ahci/caba/source/src/vci_multi_ahci.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::vector<std::string> &filenames, // vector of filenames (one per channel)  
    const uint32_t block_size = 512, // block size (bytes)  
    const uint32_t burst_size = 64,  // burst size (bytes)  
    const uint32_t latency = 0); // initial access time (number of cycles)
```

CABA Ports

- sc_in<bool> **p_resetn** : Global system reset
- sc_in<bool> **p_clk** : Global system clock
- VciTarget<vci_param> **p_vci_target** : VCI target port
- VciInitiator<vci_param> **p_vci_initiator** : VCI initiator port
- sc_out<bool> **p_irq** : Interrupt port

- sc_out<bool> **p_sdc_clk** : SD Card clock
- sc_out<bool> **p_sdc_cmd_value_out** : CMD bus value to SD Card
- sc_out<bool> **p_sdc_cmd_enable_out** : CMD bus enable to SD Card
- sc_in<bool> **p_sdc_cmd_value_in** : CMD bus value from SD Card
- sc_in<bool> **p_sdc_cmd_enable_in** : CMD bus enable from SD Card
- sc_out<bool> **p_sdc_dat_value_out[4]** : DAT bus value to SD Card
- sc_out<bool> **p_sdc_dat_enable_out** : DAT bus enable to SD Card
- sc_in<bool> **p_sdc_dat_value_in[4]** : DAT bus value from SD Card
- sc_in<bool> **p_sdc_dat_enable_in** : DAT bus enable from SD Card

7) TLM-DT Implementation

Not available yet.