VciEthernet

Functional Description

This component is an network controller which enables connecting embedded software running in the simulator to real ethernet networks outside the simulation.

Ethernet frames are relayed using an ethernet tap device on the host operating system. Such virtual ethernet device can be bridged to a physical interface so that the VciEthernet component is able to receive and transmit real world packets and access the internet.

The controller has a built-in DMA engine and separate RX and TX packet FIFOs.

The TX FIFO must be filled by the software with size and address of packets to send. When a packet has been processed, it must be popped from the FIFO. The RX FIFO is filled by the software with size and address of buffers ready to store incoming packets. When a packet has been received, it can be popped from the FIFO. The maximum size of the FIFOs must not be exceeded by pushing too many entries without first popping the processing results.

A DMA transfer starts as soon as some packets are pushed in the TX FIFO or an incoming packets is available and some buffers are free in the RX FIFO. An interrupt can be triggered when a TX or RX operation is completed.

The device is controlled by accessing a few memory mapped registers:

- `ETHERNET_TX_SIZE` (write): Used to set the size of the packet to send, must be set before pushing the address in the TX FIFO.
- `ETHERNET_TX_FIFO` (write): Push address and size of packet to send on TX FIFO.
- `ETHERNET_TX_FIFO` (read): Pop the status of sent packet from the TX FIFO. Contains 0 if no completed TX packet is available in the FIFO. A value of 1 indicates a successfully transmitted packet and other values are error codes.
- `ETHERNET_RX_SIZE` (write): Used to set the size of RX buffer, must be set before pushing the address in the FIFO.
- `ETHERNET_RX_SIZE` (read): Contains the actual size of the received packet, this register can be read before popping the associated status from the FIFO.
- `ETHERNET_RX_FIFO` (write): Used to push address and size of a new buffer on the RX FIFO.
- `ETHERNET_RX_FIFO` (read): Pop the status of received packet from the RX FIFO. Contains 0 if no completed RX packet is available in the FIFO. A value of 1 indicates a successfully received packet and other values are error codes.
- `ETHERNET_STATUS` (read): Contains device status flags. Bit 0 indicate if the link is up. Bit 1 is set if a completed TX operation can be popped from the TX FIFO and Bit 2 is set if a completed RX packet can be popped from the RX FIFO.
- `ETHERNET_CTRL` (write): Perform device control operations. Setting bit 0 resets the device and disable interrupts. Setting bit 1 enables the TX done interrupt, setting bit 2 enables the RX done interrupt and

setting bit 3 enables the link status changed interrupt.

- `ETHERNET_FIFO_SIZE` (read): Contain maximum size of the TX & RX FIFOs.
- `ETHERNET_MAC_LOW` and ETHERNET_MAC_HIGH (read): Contain the device MAC address.

Using the virtal tap device

To be able to create a virtual ethernet device on the host operating system, the SoCLib simulator which contains the VciEthernet component must be granted some privileges. There is no need to run the simulation as root, instead you can setup some special privileges for the simulator. Under GNU/Linux this can be done by running:

```
sudo setcap cap_net_admin=eip ./simulator
```

This command must be executed again if you generate a new executable.

Once the simulation as started, the tap device associated with the VciEthernet component is down and so is the link status reported by the VciEthernet component inside the simulation. This can be changed by setting the interface up:

```
sudo ifconfig soclib0 up
```

It's likely that the host operating system will start sending probing packets on this interface and these packets will be received inside the simulation provided that the embedded operating system has properly configured the VciEthernet device.

You can then bridge the `soclib0` interface with a real interface like `eht0` for instance:

```
sudo brctl addbr br0

sudo brctl addif br0 soclib0
sudo brctl addif br0 eth0

sudo ifconfig br0 up
```

Component definition & usage

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

See SoclibCc/VciParameters

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

CABA Implementation

CABA sources

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

CABA Constructor parameters

```
VciEthernet(
     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 &if_name = "soclib0"); // host os tap interface name
```

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