Vcilcu

1) Functional Description

This VCI target is a memory mapped peripheral implementing a vectorized interrupt controller. It can concentrate up to 32 independent interrupt lines **p_irq_in[i]** to a single **p_irq** interrupt line.

The active state is high, and the output interrupt is the logical OR of all input interrupts. Each input interrupt can be individually masked through a programmable register.

This component can be addressed to return the index of the highest priority active interrupt **p_irq_[i]**. The priority scheme is fixed : The lower indexes have the highest priority.

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

This component contains 5 memory mapped registers:

- `ICU_INT` Each bit in this register reflects the state of the corresponding interrupt line. This is read-only.
- `ICU_MASK` Each bit in this register reflects the state of the enable for the corresponding interrupt line. This is read-only.
- `ICU_MASK_SET` Each bit set in the written word will be set in the ICU MASK. (ICU_MASK = ICU_MASK | written_data). This is write-only.
- `ICU_MASK_CLEAR` Each bit set in the written word will be reset in the ICU MASK. (ICU_MASK = ICU_MASK & ~written_data). This is write-only.
- `ICU_IT_VECTOR` This register gives the number of the highest-priority active interrupt. If no interrupt is active, (-1) is returned. This is read-only.

For extensibility issues, you should access your ICU using globally-defined offsets.

You should include file `soclib/icu.h` from your software, it defines `ICU_INT`, `ICU_MASK`, `ICU_MASK_SET`, `ICU_MASK_CLEAR`, `ICU_IT_VECTOR`.

Sample code:

```
#include "soclib/icu.h"

static const volatile void* icu_address = 0xc0000000;

static icu_test()
{
    // Getting / setting interrupt mask
    uint32_t current_interrupt_mask = soclib_io_get( icu_address, ICU_MASK );

    // Enabling IRQ #5
    soclib_io_set( icu_address, ICU_MASK_SET, 1<<5 );
    // Disabling IRQ #0
    soclib_io_set( icu_address, ICU_MASK_CLEAR, 1<<0 );

    // When interrupt is raised, you may do:
```

```
    int irq_to_serve = soclib_io_get( icu_address, ICU_IT_VECTOR );
    // This should be equivalent to (see man 3 ffs)
    int irq_to_serve = ffs( soclib_io_get( icu_address, ICU_IT_VECTOR )
                          & soclib_io_get( icu_address, ICU_MASK ) );
}
```

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

2) Component definition & usage

source:trunk/soclib/module/infrastructure_component/interrupt_infrastructure/vci_icu/caba/metadata/vci_icu.sd

See SoclibCc/VciParameters

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

3) CABA Implementation

CABA sources

- interface : source:trunk/soclib/soclib/module/infrastructure_component/interrupt_infrastructure/vci_icu/caba/source/include/vci_
- implementation : source:trunk/soclib/soclib/module/infrastructure_component/interrupt_infrastructure/vci_icu/caba/source/src/vci_icu.

CABA Constructor parameters

```
VciIcu(
     sc_module_name name,  // Component Name
     const soclib::common::InTab &index, // Target index
     const soclib::common::MappingTable &mt,  // Mapping Table
     size_t nirq);  // Number of input interrupts
```

CABA Ports

- sc_in<bool> **p_resetn** : Global system reset
- sc_in<bool> **p_clk** : Global system clock
- soclib::caba::VciTarget<vci_param> **p_vci** : VCI port
- sc_out<bool> **p_irq** : Output interrupt port
- sc_in<bool> **p_irq_in[]** : Input interrupts ports array