General rules for the SoCLib hardware components

Naming conventions

Namespaces

SystemC being build upon C++, we use the C++ namespace constructs, to create unambiguous names. SoCLib defines the following namespaces:

- `soclib`
- `soclib::common`
- `soclib::caba`
- `soclib::tlmt`

Variables

The following conventions have been defined :

- All component port names should be prefixed with `p_`
- All component register names should be prefixed with `r_`
- All component member variable names should be prefixed with `m_`

VCI initiators and targets indexation

In a VCI-based architecture, all initiators and targets must be indexed. Initiators and targets have different address spaces.

- The target index is used by interconnect components to route the VCI command packets : the target index is decoded from VCI ADDRESS MSBs.
- The initiator index is used by the interconnect components to route the VCI response packets : the initiator index is the VCI RSRCID.

Indexes can be :

- a simple scalar index, in case of a *flat* interconnect.
- a composite index, in case of a *clusterised* architecture, using a two-level (or more) hierarchical interconnect.

mapping_table/include/int_tab.h? defines an utility class storing a list of indexes.

All indexes must be declared as IntTab?s.

Endianness

All SoCLib targets components are little-endian. In case of write, the bytes positions are fully controlled by the VCI BE bits :

- LSBs of the VCI ADDRESS are ignored, and the VCI ADDRESS is only used to select a VCI cell (a word in memory).
- Bytes are selected by the VCI BE field, and the BE[0] bit is always associated to the Byte 0 of the VCI WDATA field (ie WDATA[7:0]).

Source Code Style and Indentation (to be completed)

In so far as possible, people writing code are encouraged to follow some rules regarding indentation and coding style. These rules are described below:

- The coding style used is a variant of the BSD KNF style.
  - ♦ There must be one instruction per line : a line can be terminated by:
    - ◊ a semi-colon `;`
    - ◊ an opening bracket `{`
    - ◊ a closing bracket `}`
    - ◊ a colon after a `case` statement
  - ♦ A line following an opening bracket must be right-shifted
  - ♦ The opening bracket must always be the last character of its line
  - ♦ A left shift is done on a line containing a closing bracket
  - ♦ The opening bracket can be either on the same line as the instruction before or be the single instruction of its line
  - ♦ A line following a `case` statement must be right-shifted if no new block is created
  - ♦ A space is inserted at the following places:
    - ◊ after the statements `if`, `while`, `for` and `switch`
    - ◊ before and after a =
    - ◊ before and after all binary operators: ==, `!=`, `&&`, `||`, `&` (bitwise and), `|`, <<, >>, `%`, `/`, `*` (multiplication), +, −, <, >
    - ◊ before and after a `*` in a pointer declaration
    - ◊ after a comma `,`
    - ◊ before the question mark `?` and the semi-colon `:` in a unary if construct `(x ? y : z)`
    - ◊ after the closing parenthesis in a cast statement
    - ◊ after the `//` sequence (commentary)
  - ♦ A space is **not** inserted at the following places :
    - ◊ before a semi-colon `;`
    - ◊ before and after the operators −> and `.` (field selection)
    - ◊ before the opening bracket of a function declaration
    - ◊ before the opening bracket of a function call
    - ◊ before a comma `,`
    - ◊ before the operators ++ and −−
    - ◊ after a `*` in a pointer dereferencement
    - ◊ before a colon `:` of a case statement
    - ◊ before an indexation operator `[]`
  - ♦ It is tolerated to add spaces to align some elements such as: affectations, indexation, conditions on several lines, parameters in function calls, variables declarations.
- C-99 types must be used when possible: `int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t`
- The indentation must use spaces and not tabs. The number of spaces used is 4.

- Blocks (i.e. brackets) must be used even for a single instruction.
- The keywords `public` and `private` can be either indented regularly or the same way as their parents (the class in which they are).

Here is a code exemple :

```
template <int32_t size> class MyClass : public Parent {

public:
    sc_signal<uint32_t> my_signal;
};

typedef struct _mystruct {
    uint32_t val;
} mystruct;


int32_t f(int32_t x, mystruct * y, uint32_t c) {
    int16_t a, b = 0;
    int32_t * ptr;
    mystruct st;

    MyClass<1>  t1;
    MyClass<10> t10;

    while (st.val == b && *ptr != 0) {
        if ((a & 0x10) != 0 || y->val) {
            *y--;
#ifdef SOME_FLAG
            my_counter++;
#endif
            return 1;
        }

        switch (c) {
            case 0:
                printf("c is %d\n", c);
                break;
            case 1:
            {
                int local = 2;
                c = local + 1;
            }
            default:
                break;
        }
    }
    // f is a recursive function
    f(x, y, c + 1);
    int32_t exp = (a + (b * st.val) % d) >> 2;
    uint32_t b = (a * (b + c) == 1) ? c : c + 1;
    uint32_t var = *(uint32_t *) 0x10000000;

    return exp;
}
```

- Every file must contain at its end an epilogue specifying some indentation parameters for vim and emacs. This epilogue must be:

```
/*
# Local Variables:
# tab-width: 4;
# c-basic-offset: 4;
# c-file-offsets:((innamespace . 0)(inline-open . 0));
```

```
# indent-tabs-mode: nil;
# End:
#
# vim: filetype=cpp:expandtab:shiftwidth=4:tabstop=4:softtabstop=4
*/
```

Please note: some files do not respect yet all these conventions.