Metadata

For SoCLib components, metadata is in `.sd` files, parsed by `soclib-cc`.

In these files, there is an optional `extensions =` statement. This statement is a list of string formatted as `"namespace:keyword=value"`. An example for the dsx namespace is:

```
extensions = [
    'dsx:cpu=wrapper:iss_t',
    'dsx:addressable=target_index',
],
```

Here we'll make a list of these extensions and their effects

Pure C++ module constraints

Module without a name as first parameter

Some pure-C++ (i.e. not SystemC) modules do not need a name as first parameter. There is an extension telling the netlister to generate a correct constructor invocation for the module:

```
"dsx:noname",
```

Memory mapping & organization

VCI srcid / tgtid, ports and mapping tables

VCI uses `(r)srcid` to route response packets back to the originating initiator. As it is decoded, the srcid field attributed to an initiator is directly equivalent to the port it is connected to on peer interconnect. A module then has to be able to retrieve:

- srcid field value (or initiator index as an `IntTab` parameter)
- mapping table associated to the interconnect
- correspondance between ports and indexes

Likewise for targets.

Target components

For a simple target component, we'll probably have:

- a `VciTarget` port
- a mapping table as parameter
- a target index as parameter

```
ports = [
    Port('caba:vci_target', 'p_vci_port_name'),
    ?
    ],
instance_parameters = [
    ?
    parameter.IntTab('target_index_parameter_name'),
    parameter.Module('mapping_table_parameter_name', typename = 'common:mapping_table'),
    ?
],
```

Needed metadata are:

```
"dsx:get_ident=target_index_parameter_name:p_vci_port_name:mapping_table_port_name",
"dsx:addressable=target_index_parameter_name",
```

`dsx:get_ident` associates a given index to a port and a mapping table. Actual index value and mapping table to use will be implicitly retrieved through the module connected to the port.

`dsx:addressable` tells the target is accessible through this target index.

Maximum target segments

Sometimes targets can only have a limited number of addressable segments. You can enforce this limit telling DSX to do so.

For instance, a simple controller may only need one segment:

```
extensions = [
    'dsx:max_segments=1',
],
```

Addressable segments associated to a srcid

For coherent platforms, we sometimes need to associate a source id to a segment. Thus when creating the segment, we have to add an id. This is done through "`dsx:on_segment=` extension. Example in the `vci_cc_xcache_wrapper_v1`:

Syntax: `target_segment_mapping_table:add_index:initiator_index_name`

```
instance_parameters = [
```

```
        parameter.Int('proc_id'),
        parameter.Module('mtp', 'common:mapping_table'),
        parameter.Module('mtc', 'common:mapping_table'),
        parameter.IntTab('initiator_rw_index'),
        parameter.IntTab('initiator_c_index'),
        parameter.IntTab('target_index'),
        ?
        ],
    extensions = [
        'dsx:get_ident='
           'initiator_rw_index:p_vci_ini_rw:mt,'
           'initiator_c_index:p_vci_ini_c:mc,'
           'target_index:p_vci_tgt:mc',
        'dsx:on_segment=mc:add_index:initiator_rw_index',
        'dsx:addressable=target_index',
        ?
        ],
```

Initiator

For a simple initiator component, we'll probably have:

- a `VciInitiator` port
- a mapping table as parameter
- a source index as parameter

```
    ports = [
        Port('caba:vci_initiator', 'p_vci'),
        ?
        ],
    instance_parameters = [
        ?
        parameter.Module('mt', 'common:mapping_table'),
        parameter.IntTab('index'),
        ?
        ],
```

This works the same way as for targets. Let's have an example with real names:

```
    extensions = [
        'dsx:get_ident=index:p_vci:mt',
    ],
```

`dsx:get_ident` associates a given index to a port and a mapping table. Actual index value and mapping table to use will be implicitly retrieved through the module connected to the port.

Interconnects

In order to get the two preceding points working, interconnects need to provide a way to get a source / target identifier. When DSX reaches an interconnect, it looks for the following extension:

```
    extensions = [
        'dsx:interconnect=root',
        'dsx:obtain_ident_method=port',
        ],
```

`dsx:interconnect` may either have no value, or "root". "root" means the interconnect is the root of the routing tree, for instance a VGMN. Local interconnects (like the `vci_local_crossbar`) only need `"dsx:interconnect"`.

`dsx:obtain_ident_method` may value either `port` or `param`.

`port`
: The identifier is directly related to the port number in a port array, this is the usual value for all port-array based interconnects like VGMN or crossbars.

`param=some_parameter_name`
: Use a given instance parameter as identifier for associated component. This is the usual value for one-wrapper-per-component interconnects like ring or dspin.

CPUs / ISS wrappers

Processor type

For DSX software deployment system, we need to highlight CPUs and their types. This is the `"dsx:cpu="` extension.

As ISS are wrapped in caches, and wrapping may be nested (GDB, memchecker, ?) a recursive way, a recursive lookup is achieved.

`"dsx:cpu="` may be:

`wrapper:some_template_parameter_type`
: used in caches and iss wrappers, to tell the iss is unknown, but given as a parameter

`some_arch`
: used for actual CPU implementations, tells what the cpu architure is. Current known architectures are `mips32el`, `mips32eb`, `powerpc405`, `arm`.

Processor identifier

CPU are associated to an unique identifier for the platform, usually a sequential name. DSX needs to know what parameter corresponds to this identifier. This is done with the `"dsx:mapping_type=processor:parameter_name"`:

```
instance_parameters = [
    parameter.Int('ident'),   # processor identifier
    parameter.Module('mt', 'common:mapping_table'),
    parameter.IntTab('index'),
    ?
    ],
extensions = [
    'dsx:mapping_type=processor:ident',
    ],
```

Devices

Devices may be declared, in order to be able to map specific options to them. For now, only ttys are supported.

```
extensions = [
    'dsx:device=tty',
    ],
```

C++ hacks

Sometimes, you have to call a C++ static function once per class, or a method once per object instance. These two things will help, even if this obscure feature is mostly undocumented. Use source code for reference.

Static configuration lines

`'dsx:static_config_lines=%(class)s::init(%(env:mapping_table)s,%(env:loader)s,"%(met`

This is an excerpt from `iss_memchecker` metadata, telling the netlister must generate a line with a given template once for each IssMemchecker?<?> type, taking references from envorinment and all peripherals.

Instance configuration lines

Likewise, there is

`'dsx:config_lines=?'`, emitted once for each object. See dsx soclib's component handling code for usage.

MutekH configuration

Some modules may want to influence MutekH's own configuration. This is only available when using DSX software-generation feature. Using a given module may then change some configuration tokens in MutekH's build system.

For instance, when putting an `iss_memchecker` in the netlist, we want MutekH to be built with memchecker support code activated, thus we can add in iss_memchecker's metadata the following extension:

```
extensions = [
    'dsx:set_config=CONFIG_SOCLIB_MEMCHECK',
],
```

Examples

DMA

Let's describe a DMA. It has

- one initiator port
- one target port
- an initiator index
- a target index
- a mapping table associated to this all

It only supports one addressable segment

```
ports = [
    Port('caba:vci_target', 'p_vci_target'),
    Port('caba:vci_initiator', 'p_vci_initiator'),
    ?
],
instance_parameters = [
    parameter.Module('mt', typename = 'common:mapping_table'),
    parameter.IntTab('srcid'),
```

```
    parameter.IntTab('tgtid'),
    parameter.Int('burst_size'),
],
extensions = [
    'dsx:addressable=tgtid',
    'dsx:max_segments=1',
    'dsx:get_ident=srcid:p_vci_target:mt,tgtid:p_vci_initiator:mt',
],
```