Page Outline?

MappingTable

1) Functional Description

This object is NOT an hardware component. It can be used by the system designer to describe the memory mapping and address decoding scheme of any shared memory architecture build with the SoCLib hardware components.

This object is basically an associative table, where each entry define a segment descriptor.

The mapping table is a centralized description of both the address space segmentation, and the mapping of the segments on the VCI physical targets.

From this centralized description, it is possible to derive the **Routing Tables** used by the hardware interconnect components to decode the VCI address, and route the VCI packets to the proper VCI target. It is also possible to derive the **Cacheability Table** used by the cache controllers to determine cached & uncached addresses.

2) Usage

The mapping table and the segments must be defined in the SystemC Top Cell, before instanciating the hardware components, as the mapping table is used by the hardware components constructors.

mapping table instanciation

The mapping table must define :

- The VCI ADDRESS width (number of bits)
- The number of interconnection levels, defining the number of address subfields to be decoded.
- The widths of the VCI ADDRESS subfields used for command packets routing
- The widths of the VCI RSRCID subfields used for response packets routing
- The mask used by the cache controller for determining if an address is cacheable or not

```
MappingTable maptab( addr_width, addr_bits, srcid_bits, cacheability_mask);
```

For instance:

```
MappingTable maptab( 32, IntTab(8, 4), IntTab(8, 2), 0x000c0000);
```

In this example, we created a 32-bit Mapping Table, with 2 level interconnection, 8 address bits for global command routing, 4 address bits for local command routing, 8 RSRCID bits for global response routing, 2 RSRCID bits for local response routing. This makes RSRCID field 10-bit wide. The cacheability table will be indexed by the two bits ADDRESS[19:18].

Segments definition

Segments holds information about a portion of the address space, with some attributes:

- a name

- a base address
- a size (number of bytes)
- a target_index (an IntTab)
- a cacheability flag

The target_index field identifies the VCI target that contains the corresponding segment, and is used by the interconnect to route a command packet to the proper target. The cacheability field is used by the VciXcache component to define if the corresponding segment is cacheable. All segments defined by the system designer for a given architecture must be non-overlaping.

A segment can be added in the mapping table with the method add():

```
maptab.add(Segment( "seg0", 0x50000, 0x1000, IntTab(3,2), true ))
```

In this example, the segment is associated to target no 2 in cluster no 3, and is cacheable.

3) Routing Tables

One level interconnect

This is the simplest case, where all VCI targets and VCI initiators are connected to a "flat" interconnect.

- each VCI component is identified by a simple index.
- all VCI targets must have different indexes.
- all VCI initiators must have different indexes.
- The initiator index must be equal to the VCI SRCID value.
- The VCI ADDRESS field is structured in two fields: `| MSB | OFFSET |`
    - The MSB field is decoded by the flat interconnect to route the command packet to the proper VCI target.
    - The OFFSET field is decoded by the VCI target.
- The VCI SRCID field is used by the interconnect to route the response packet to the proper VCI initiator.
- Most hardware interconnects (such as the PibusBcu or the VciVgmn components) make the assumption that the target indexes are between 0 and T-1 (where T is the total number of VCI targets), and the initiator indexes are between 0 and M-1 (where M is the total number of VCI initiators).

The hardware interconnect contains a ROM implementing a **Routing Table** indexed by the VCI address MSBs and containing the corresponding target index. The content of this table is automatically computed by a method associated to the mapping table.

Two-level interconnect

With a two-level interconnect, the hardware architecture is supposed to be split into several subsystems (or clusters). There is a global interconnect (such as the VciVgmn for inter-cluster communications, and one local interconnect (such as the VciLocalCrossbar in each cluster for intra-cluster communications.

- each VCI component is identified by a structured index containing two indexes:
    - a global index that identifies the subsystem (or cluster) index.
    - a local index, that identifies the VCI component in the cluster.
- all VCI components in the same cluster have the same global index.
- all VCI components in the same cluster must have different local indexes.
- The VCI ADDRESS field is structured in three fields : `| MSB | LSB | OFFSET |`
    - The MSB field is decoded by the global interconnect to route the command packet to the proper cluster.

    - ♦ The LSB field is decoded by the local interconnect to route the command packet to the proper target.
    - ♦ The OFFSET field is decoded by the VCI target.
- The VCI SRCID field is structured in two fields : `| MSB | LSB |`
    - ♦ The SRCID MSB field must be equal to the initiator global index.
    - ♦ The SRCID LSB field must be equal to the initiator local index.

The services provided by the Mapping Table for a two level interconnect are the following :

- It generates the **Global Routing Table**, implemented as a ROM by the global interconnect. This table is indexed by the VCI ADDRESS MSB bits, and contains the corresponding global index (cluster index).
- It generates - for each cluster - the **Local Routing Table**, implemented as a ROM by the local interconnect. This table is indexed by the VCI ADDRESS LSB bits and contains the target local index. Depending on the mapping, each cluster can have a different Local Routing Table.

4) Cacheability Table

Cacheability is a by-segment attribute. Cacheability is implemented by the VciXcache component. As this component does not implement a MMU, it contains a dedicated address decoder to determine the cacheability. This decoder is implemented as a **Cacheability Table**. This table is accessed by the cache controller for each processor request. This table is indexed by the address bits defined by the cacheability_mask, and contains the Boolean defining the cacheability.

The content of the **Cacheability Table** is automatically constructed by a method of the mapping table. Of course, the mapping of the cacheable segments must be consistent with the cacheability mask defined in the mapping table. This is checked by the mapping table, and you'll get an `"Incoherent MappingTable" exception` if the mapping is inconsistent.