Tables types

From the segments defined in the Mapping table, it is possible to generate 3 types of tables indexed by the VCI address:

- routing table, yielding a target port number;
- locality table, yielding a boolean indicating wether the address is local;
- Cacheability table, yielding a boolean indicating wether the address is cacheable;

When the mapping table is created, 4 informations must be defined:

- Address size (in bits)
- Address routing table fields sizes (in bits, from the VCI ADDRES MSB bits)
- Index routing table fields sizes (in bits, from the VCI SRCID)
- Cacheability mask

Segments are registered with the `.add()` method. Nothing is verified until actual tables are created.

Routing tables

In the general case, we can define hierarchical interconnects, where both initiators and targets are grouped in subsystems, called clusters. Therefore, each initiator (and each target) is identified by two indexes: a cluster_index, and a local_index.

In such a case, we must use, local routing tables, global routing tables and locality tables.

We'll suppose we create a Mapping Table with the following code:

```
MappingTable maptab(32, IntTab(8, 4), IntTab(4, 3), 0x00300000 );
```

For a command packet, the first 8 MSB ADDRES bits must be decoded by the global interconnect using the global routing table to get the target cluster_index, and the next 4 ADDRESS bits must be decoded by the local interconnect using a local routing table to get the target local_index.

The locality table is used by the local interconnect to decide wether a command packet is local or not.

For a response packet, the 4 SRCID MSB bits define directly the initiator cluster_index, and the next 3 SRCID bits define directly the initiator local_index.

The interconnect hierarchy can be seen as a tree. Each interconnect in tree has an unique index, which is an IntTab. The root interconnect has the empty `IntTab()` ID, if there are local interconnects, they are numbered `IntTab(n)` where `n` is the local cluster_index. This ID **must** be the same as the targets and initiator ports it is connected to on the global interconnect.

In the example above, `vgmn` is the global interconnect and uses the 8 address MSB bits. `lc0` and `lc1` use the 4 next address bits (but the tables content is generally different for lc0 and lc1).

widths	8	4
bits	31 ? 24	23 ? 20
locality decision	lc0, lc1	
routing decision	vgmn	lc0, lc1

When code calls `getRoutingTable( index )` on a `MappingTable`, `MappingTable` scans the list of registered segments and filters all the segments corresponding to index value.

Let's say we have the following segments:

Name	Address	Size	Target	Cacheable
seg0	0x12000000	0x00100000	(0, 0)	False
seg1	0x12100000	0x00100000	(0, 1)	True
seg2	0x14000000	0x00100000	(1, 0)	False
seg3	0x14100000	0x00100000	(1, 1)	True
seg4	0x14200000	0x00080000	(1, 2)	True

When calling `getRoutingTable( IntTab(1) )`, the resulting local routing table will only contain information about segments located in cluster 1: `seg2`, `seg3` and `seg4`.

As the 8 first bits of address are assumed already decoded to select cluster 1, the local routing table only decodes the next 4 address bits:

Input (bits 23-20)	Target ID
0000	0 (seg2)
0001	1 (seg3)
0010	2 (seg4)
0011	Don't Care
0100	Don't Care
...	Don't Care
1111	Don't Care

If the routing table creator encounters an impossible configuration in the mapping table, it raises an exception. Let's suppose we add the following segment:

Name	Address	Size	Target	Cacheable
seg5	0x12300000	0x00010000	(1, 3)	False

The global routing table should decode the 8 address MSB bits to define the cluster_index, segment seg0 and segment seg5 have the same MSB bits (0x12**), but, they are mapped to different clusters, which is illegal.**

Locality tables

Locality tables just tell whether an address is local to a subtree of the network or not.

In the above example, locality table creation for local interconnect 0 (`getLocalityTable( IntTab(0) )`) would involve:

Name	Address	Address[31:24]	locality
seg0	0x12000000	00010010	0 (local)

seg1	0x12100000	00010010	0 (local)
seg2	0x14000000	00010100	1 (foreign)
seg3	0x14100000	00010100	1 (foreign)
seg4	0x14200000	00010100	1 (foreign)

So the locality table will be:

Address[31:24]	Is Local
00010010	True
00010100	False
else	Don't Care

Cacheability Table

Cacheability tables are a built the same way, but bits used for decoding are selected through the cacheability mask:

- take all segments
- extract masked value
- set the cacheability attribute for the value

We use a cacheability mask of 0x00300000 (bits Address[21:22]

Name	Address	Masked value	Address[21:20]	Cacheablility
seg0	0x12000000	0x00000000	00	False
seg1	0x12100000	0x00100000	01	True
seg2	0x14000000	0x00000000	00	False
seg3	0x14100000	0x00100000	01	True
seg4	0x14200000	0x00200000	10	True

We obtain the following cacheability table:

Address[21:20]	Cacheability
00	False
01	True
10	True
11	Don't Care

Cacheability Tables take an address, select appropriate bits and yield the Cacheability boolean.

Here again an exception is raised if we encounter an incoherent mapping table.

Assume we add a new segment seg5:

Name	Address	Size	Target	Cacheable
seg5	0x20280000	0x00080000	(1, 2)	False

Its cacheability entry should be:

Name	Address	Masked value	Address[21:20]	Cacheablility
seg5	0x20280000	0x00200000	10	False

The cacheability should be True for segment 4, and False for segment 5, which is not possible.