Tables types

Mapping table creates 5 types of tables:

- Commands routing table, indexed by addresses, yielding target port number;
- Commands locality table, indexed by addresses, yielding boolean whether an address is local or not;
- Response routing table, indexed by source ID, yielding initiator port number;
- Response routing table, indexed by source ID, yielding boolean whether an index is local or not;
- Cacheability table, indexed by address, yielding whether allowed to cache or not.

When the mapping table is created, it gets 4 informations:

- Address size (in bits)
- Address routing table fields sizes (in bits, from the MSBs)
- Index routing table fields sizes (in bits, from MSB of indexes)
- Cacheability mask

When the mapping table is created, segments are registered with the `.add()` method. This does nothing except registering segments. Nothing is verified until actual tables are created.

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

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

Variable tables

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

In this figure, the command routing table is different is `lc0`, `lc1` and `vgmn`.

Command tables

Routing tables can only use a part of the address to do their job. In the example above, `vgmn` is the global interconnect and uses Most-significant-bits of the addresses; `lc0` and `lc1` use the same bits (but on different tables), just after the MSBs used by `vgmn`:

An address and its decoding fields, if we suppose we created the Mapping Table as before, we have a 32-bit address:

width:	8	4	(the rest)
bits:	31 ? 24	23 ? 20	19 ? 0
field:	vgmn	lc0 & lc1	rest of address

Creating the routing tables

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

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, 1)	True

When calling `getRoutingTable( IntTab(1) )`, the resulting routing table will only contain information about `seg2`, `seg3` and `seg4`, which targets `(1, ?)`. As the 8 first bits of address are assumed already decoded, the table only decodes the next 4 bits:

Input (bits 23-20)	Target ID
0	0 (seg2)
1	1 (seg3)
2	1 (seg4)
3 .. 0xf	unknown

Incoherences

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

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

Routing table should now be (even if bits 31?24 are 0x20):

Address (bits 23-20)	Target value
0	0 (seg2)
1	1 (seg3)
2	1 or 2 (seg4 & seg5)
3 .. 0xf	unknown

Creating the 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 be:

Name	Address	Address[31:24]	Target cluster
seg0	0x12000000	0x12	0 (local)
seg1	0x12100000	0x12	0 (local)
seg2	0x14000000	0x14	1 (foreign)
seg3	0x14100000	0x14	1 (foreign)
seg4	0x14200000	0x14	1 (foreign)

So the locality table would be:

Address[31:24]	Is Local
0x00 .. 0x11	Unknown
0x12	True
0x12	True
0x13	Unknown
0x14	False
0x14	False
0x14	False
0x15 .. 0xff	Unknown

Response tables

Response Routing table

The response tables are quite the same as the command ones, except bits used in decoding the source ID field are equal to the result.

`getIdRoutingTable( IntTab(1) )` yields:

Srcid (bits 7-4)	Target value
0	0
1	1
2	2
...	...
0xf	0xf

Cacheability Table

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

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

We use a cacheability mask of 0x00300000.

Name	Address	Masked value	Address[21:20]	Cacheablility
seg0	0x12000000	0x00000000	0	False
seg1	0x12100000	0x00100000	1	True
seg2	0x14000000	0x00000000	0	False
seg3	0x14100000	0x00100000	1	True
seg4	0x14200000	0x00200000	2	True

We can deduct the following table:

Address[21:20]	Cacheability
0	False
1	True
2	True
3	unknown

In components' code, Cacheability Tables directly take an address, select appropriate bits and yield the Cacheability boolean.

Incoherences

Again, if we encounter an incoherent value, exception will be raised; let's suppose we add the following segment:

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

Its entry is

Name	Address	Masked value	Address[21:20]	Cacheablility
seg5	0x20280000	0x00200000	2	False

Now the table becomes:

Shortened value	Cacheability
0	False
1	True
2	True & False
3	unknown

This must not happen