SiPU Memory系统 之 地址管理

Version History

Version	Date	Owner	Description	Comment
0.01	2024.9.13	Yunsheng Sun	Initial version	
0.09	2024.9.27	Yunsheng Sun	根据内部review的结果进行调整： 1）所有需要地址转换的模块都有自己的配置寄存器空间，分别进行配置； 2）在PE内部，地址转换只需要一份，在MIF出口调用，不需要LSU/TLSU/DTE分别进行地址转换； 3）按照VA-->LA-->PA的方式描述地址转换过程； 4）最大并行进程数为8；	
0.1	2024.10.8	Yunsheng Sun	根据评论意见，做如下描述调整： 1）地址格式表格中VA/LA中，L2B/DRAM都是看作一个整体，不需要bank ID, port ID的描述； 2）Sub-DRAM的名字容易引起误解，改成one-port-DRAM-space;	
0.11	2024.10.23	Yunsheng Sun	更新3.3.2 可配置地址格式	
	2024.10.29	Yunsheng Sun	根据2024.10.23sync会议结论做成修改： 1）PA的地址格式也是连续的，不再是分离的；	

review反馈表： SiPU Memory系统地址管理--反馈表

议题讨论结论和录屏记录： SiPU Memory系统地址管理--会议录屏记录

遗留问题追踪： SiPU Memory系统地址管理--遗留问题追踪

Abbreviation

Abbreviation	Description
Host	
Device	The SiPU chip
GCS	Global Control System，under the SiPU
CP	Command Processor
GTE	Global Transfer Engine
PEC	Process Engine Cluster
PE	Process Engine
DTE	Data Transfer Engine
RV	Risc-V core
RVV	Risc-V core's vector unit
Tile-core	Tensor function core under PE
LSU	Load and Store Unit
L1DC	L1 Level Data Cache
L1IC	L1 Level Instruction Cache
L2C	L2 Level Cache
L2B	L2 Buffer
TBC	Thread Block Cluster
TB	Thread Block
PKT	Packet
PMU	Performance Monitor Unit

1. Overview

本文档主要介绍SiPU芯片项目中和Memory系统的地址管理相关的信息，将从整个系统的角度介绍整个地址空间的管理，地址信息的格式，地址信息的转换和配置等内容。本文档是以 SiOrigin SIPU总体架构概要 中的内容为基础，进行了内容扩展和细化。

”Memory Hierarchy“: 简要介绍下SiPU芯片中的Memory层次结构；

”Address Space“: 主要介绍整体地址空间，”逻辑“地址空间，物理地址空间，地址表现格式等内容；

”Address Translation“: 主要介绍地址信息在整个系统传递过程中，不同执行部件对其进行解析，分流的方法；

”Address Interleaving“: 主要介绍在存储单元，不同data type在存储单元具体组织结构中的映射关系等；

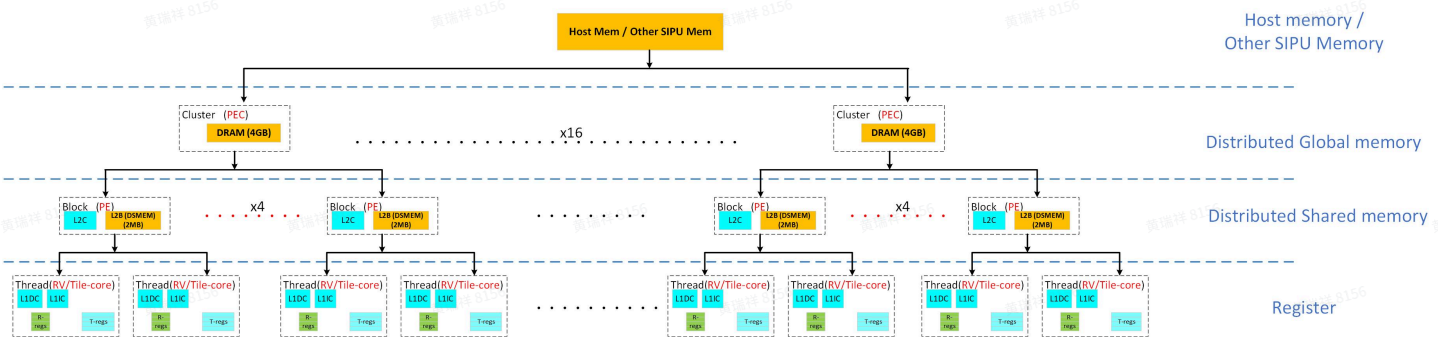
”Configuration Register“: 主要介绍为实现整体的地址信息转换，各执行部件所需的配置信息；

”Compiler/Driver Information“:

2. Memory Hierarchy

SiPU芯片项目的存储体系采用的是两层分布式共享存储体系：**Distributed Shared Memory** (对应硬件实现中的L2B存储单元) 和 **Distributed Global Memory** (对应硬件实现中的DRAM存储单元)。此部分的详细介绍请参看 SiOrigin SIPU总体架构概要 的”Memory层次“章节，此处不再赘述。

为了方便后文整体流程中地址转换的描述，在上述存储体系层次的基础上，本文对存储层级结构进行了扩展，向下把指令操作中的寄存器层级也加入进来，向上把Host Memory层级也加入进来，层级分布如下：



从地址信息管理角度，各级功能如下：

”Register~~/cache~~“ level : 指令级地址信息产生的源头，程序人员可以通过配置寄存器和基本的运算指令计算出该地址信息；（Cache逻辑是硬件的实现，对程序人员不可见）

”~~Level 1 memory/~~Distributed Shared memory“ level: 地址信息指定的目的地之一，距离执行单元最近，并且编译器和驱动可直接进行分配，操作和释放的存储单元；此处的地址信息可来自指令级，也来自系统级命令；

”~~Level 2 memory/~~Distributed Global memory“ level : 地址信息指定的目的地之一，比Shared memory更高一级，编译器和驱动可直接进行分配，操作和释放的存储单元；此处的地址信息可来自指令级，也来自系统级命令；

”Host memory/External memory“ level : 地址信息指定的目的地之一，Host级存储空间；在SiPU中采用统一编址方式，所以此处的地址信息可来自指令级，也来自系统级命令；

3. Address Space 地址空间

在SiPU芯片项目中，地址位宽是48位，可表示范围 0~0xFFFF FFFF FFFF，即256TB空间大小。

整个地址空间采用统一编址方式，即Host和Devices共享这256TB地址空间。根据不同的应用场景，可以把地址空间分成虚拟地址空间（Virtual Address Space），逻辑地址空间（Logical Address Space），物理地址空间（Physical Address Space）。不同层面的操作看到的是不同的地址空间。

虚拟地址空间，是软件层面看到的地址空间，在该空间内，每个Chip的16个Clusters的L2B地址是连续的，16个Clusters的DRAM地址也是连续的，而且此空间下，每个Chip下16个Clusters的 ID信息是虚拟的ID信息。

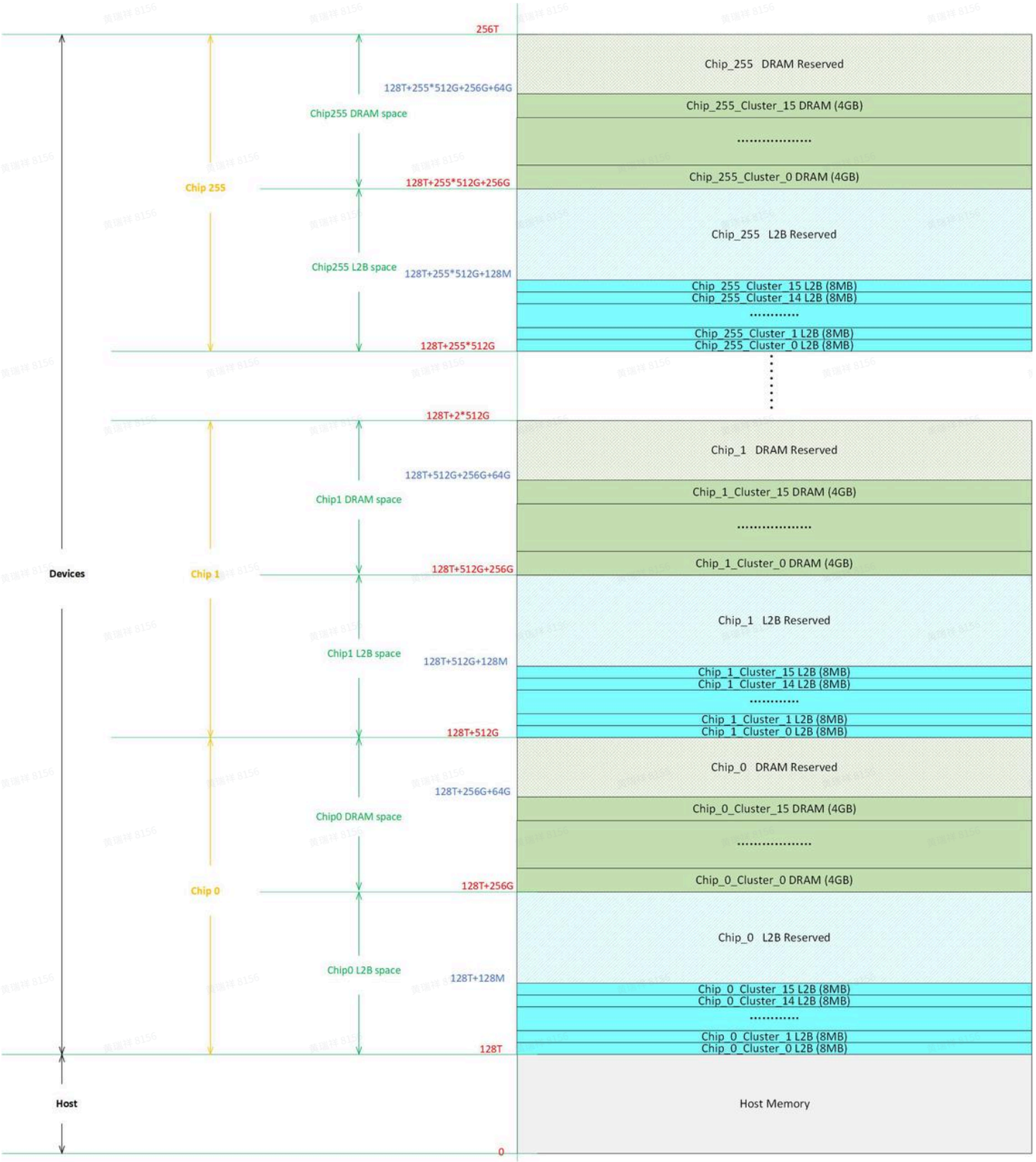
逻辑地址空间，~~该空间的地址组织方式和虚拟地址空间的地址组织方式是相同的，即每个Chip的16个Clusters的L2B地址是连续的，16个Clusters的DRAM地址也是连续的；和虚拟地址空间不同的时，此空间下每个Chip下16个Clusters的ID信息是物理的ID信息~~。历史遗留的名称，软件层面和硬件设计层面可以不用关心这个地址空间，地址转换单元会看到这个空间。该空间和虚拟地址空间的不同是：1）Cluster ID已经从虚拟ID值转换成物理ID值；2）对于可配置地址格式模式，相对于虚拟地址空间，该空间下，分散的Cluster ID信息已经被合并在一起。

物理地址空间，是硬件层面看到的地址空间，该空间内，~~每个Chip的16个Clusters的L2B地址是离散的，即每个Cluster都有自己的L2B地址空间，并且相邻两个Clusters的L2B地址空间不连续，而是间隔开的；同样16个Clusters的DRAM地址也是离散的，相邻两个Clusters的DRAM地址空间不连续，而是间隔开的~~。每个Chip的16个Clusters的L2B地址是连续的，16个Clusters的DRAM地址也是连续的。该空间和逻辑地址空间的不同是，对DRAM地址，internal address完成了多进程地址偏移的映射。

3.1 Virtual/Logical Address Space (VA/LA Space)

虚拟地址空间和逻辑地址空间的地址组织方式是相同的，放在一起介绍。这个地址空间的地址位宽是48位。

如下是虚拟/逻辑地址空间的示意：



虚拟/逻辑地址空间的256TB空间按照低128TB和高128TB分配给Host和Devices 。在devices内256个chip均分128TB空间，每个chip有512GB空间。在chip内，高低256GB空间分别分配给DRAM和L2B类型存储单元。

~~和物理地址空间不一样~~，高低256GB空间并不是均分给16个Cluster，而是按照每个Cluster占用的大小连续排列在一起。对L2B类型存储单元的256GB空间，每个Cluster占用8MB，16个Cluster共占用128MB， 所以[0,128MB)空间（相对于每个chip的每个低256GB空间起始地址而言）被分配给L2B，而

[128MB, 256GB)空间是reserved。同样对于DRAM类型存储单元的256GB空间而言，也是如此排布，[0,64GB)空间（相对于每个chip的每个高256GB空间起始地址而言）被分配给DRAM，每个Cluster占用4GB，而[64GB,256GB)空间reserved。

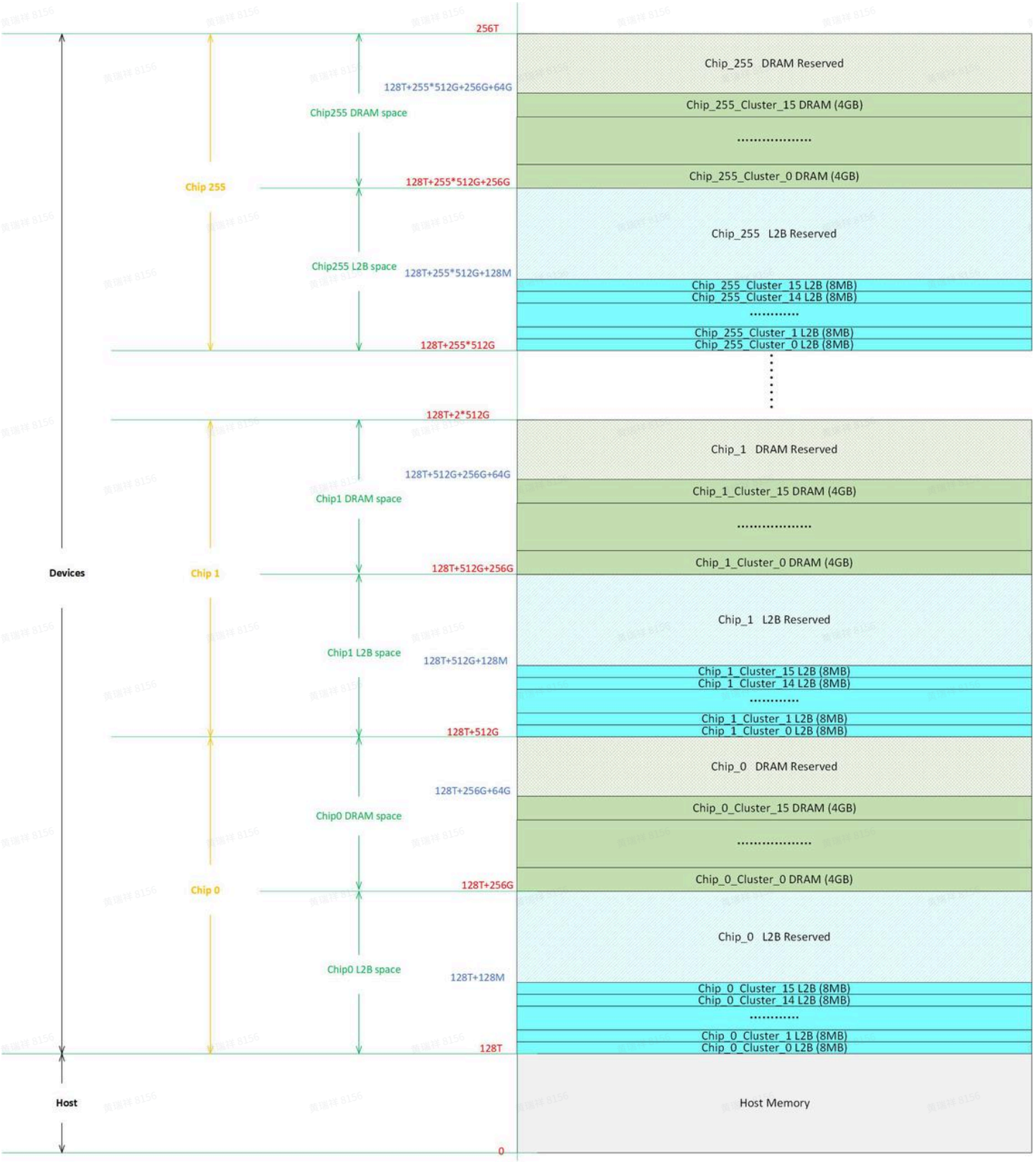
虚拟地址空间和逻辑地址空间的组织形式相同，两者的不同点是Cluster ID是当作虚拟ID来看待，还是当作物理ID来看待。如果是虚拟ID，则是虚拟地址空间，如果是物理ID，则是逻辑地址空间。在虚拟ID和物理ID之间有映射关系，两者之间的转换可查看4.1章节。

3.2 Physical Address Space (PA Space)

物理地址空间，是芯片中物理存在的存储单元的地址空间，在SiPU芯片项目中，具体指的就是Host Memory，Device上的L2B存储单元，DRAM存储单元。因为采用统一编址方式，devices上各部件的配置寄存器所对应的地址也应统一在该地址空间内（**TBD:待确认？？**）。

硬件实现时，采用的是物理地址空间。

如下，描述了SiPU中各devices存储单元的物理地址分布（当前主要描述L2B/DRAM存储单元的分布情况，其他存储单元的分布情况，待明确后再添加）：



如图所示，整个256TB空间被分成高低128TB两部分，低128TB是Host memory空间。 高128TB属于devices的memory空间。

整个devices的128TB大小的memory空间，被均分成256份，对应256个SiPU chip，每个chip空间大小是512GB 。在每个chip内部，512GB空间也分成高低256GB两部分，低256GB空间属于L2B类型存储单元，高256GB空间则属于DRAM类型存储单元。

~~无论是高256GB空间，还是低256GB空间，这个256GB的空间又被均分成16份，对应16个PE Cluster (PEC)，每个PEC都拥有16GB的空间。~~

~~对于每一个PEC，它都拥有两个16GB的地址空间，一个对应L2B类型存储单元，一个对应DRAM类型存储单元。在L2B类型存储单元地址空间内，其占用了连续的8MB地址空间，并对应于PEC内部的4个PE执行单元（即4个L2B存储单元，每个PE包含一个L2B），其余16GB-8MB空间，目前是reserved的。DRAM类型存储单元地址空间内，其占用了连续的4GB地址空间，其余12GB空间，目前是reserved~~的。

和VA地址空间格式一样，L2B/DRAM的地址，对每个Chip内的16个Cluster而言，都是连续的。

3.3 Address Information Format

在SiPU芯片项目中，地址位宽是48位，而且采用统一编址，即Host Memory 和 Devices Memory空间共用这48位地址来表示。

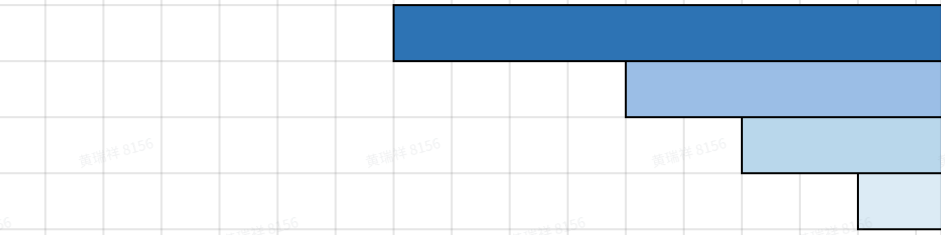
地址的编址方式支持两种模式：固定模式 和 动态可配置模式。

地址编址模式的选择可以通过配置寄存器来控制。

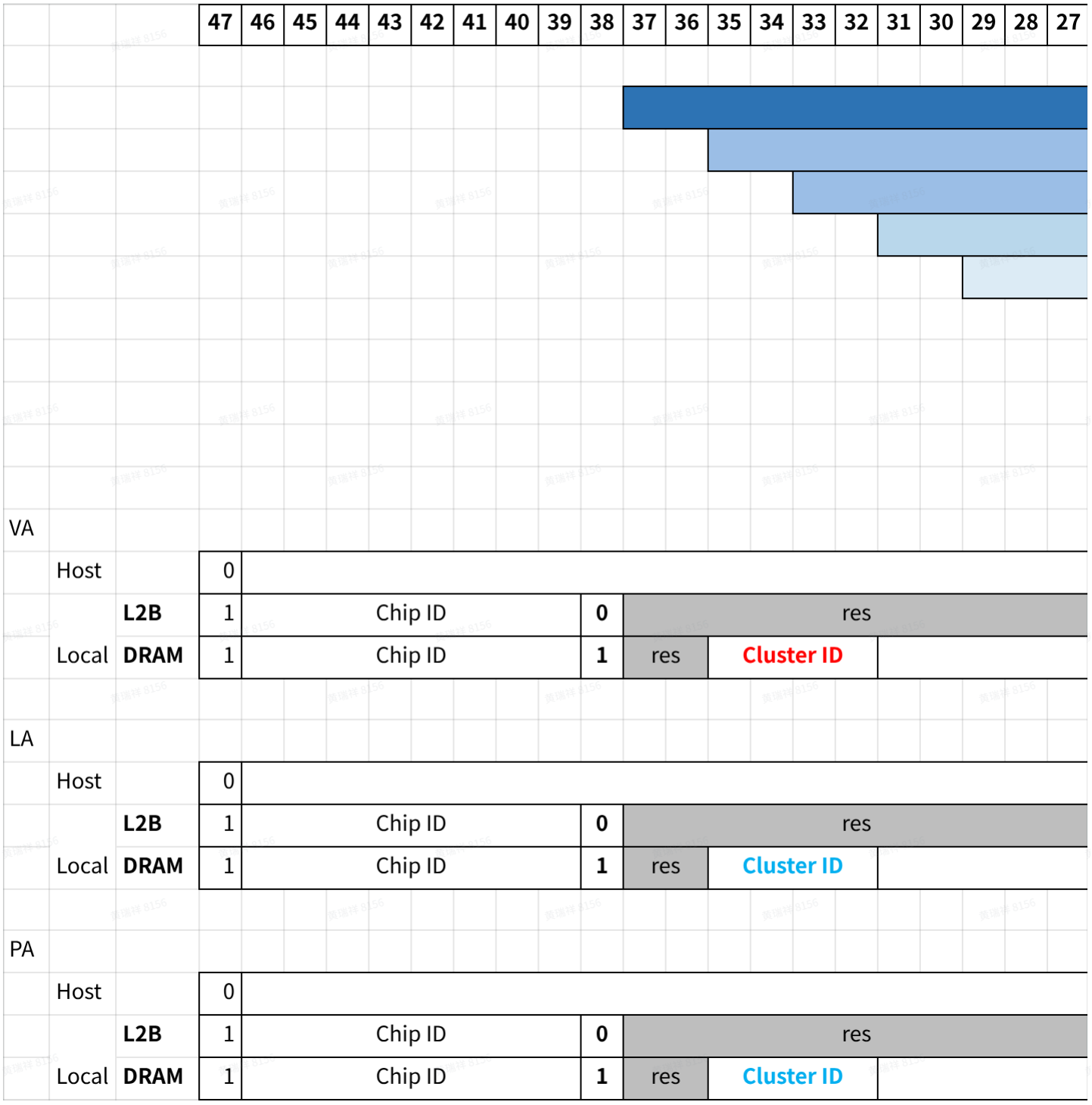
3.3.1 Fixed Format

在固定模式下，地址空间可覆盖256个SiPU chips， 每个chip内部包含16个PE Clusters (PEC), 每个Cluster内包含4个PE。在每个PE内，包含一个2MB的L2B存储单元。在每个PEC内，有一个4GB大小的DRAM存储单元。

固定模式的地址信息如下所示：



			47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	2
VA																						
	Host		0																			
		L2B	1	256 SiPU 选一								0	reserved									
	Local	DRAM	1	256 SiPU 选一								1	reserved		16 PEC选一							
LA																						
	Host		0																			
		L2B	1	256 SiPU 选一								0	reserved									
	Local	DRAM	1	256 SiPU 选一								1	reserved		16 PEC选一							
PA																						
	Host		0																			
		L2B	1	256 SiPU 选一								0	16 PEC选一					rese				
	Local	DRAM	1	256 SiPU 选一								1	16 PEC选一				reserved		4 DRAM选一		1GB 空	



Note：

1 从上表可以看到，从VA到LA，Cluster ID 信息从红色变成蓝色，表示已经完成从虚拟Cluster ID值到物理ID值的映射；

2 从LA到PA，DRAM internal address 信息从红色变成蓝色，表示已经完成DRAM地址多进程偏移的映射；

3 对PE ID 和 L2B internal address信息，没有进行任何映射功能；

无论是VA/LA还是PA，以下域段的含义是一样的

Address Field	Description
[47]	**Host/Devices memory selection** : 0: Host memory, low 128TB space; 1: Devices memory, high 128TB space;
[46:39]	**Chip ID** : 8-bit, it can select one chip from 256 chips;
[38]	**L2B/DRAM selection** : 0: L2B memory, low 256GB space; 1: DRAM memory, high 256GB space;

其他域段在~~不同地址空间模式，~~不同类型存储单元时，含义不同，分别进行说明。

VA/LA/PA L2B :

Address Field	Description
[37:27]	**Reserved** ;
[26:23]	**PE Cluster (PEC) ID** : 4-bit, it can select one PEC from 16;
[22:21]	**PE ID** : 2-bit, it can select one PE from 4;
~~[20:10]~~	~~**L2B bank entry :**~~ ~~11-bit, there are 2048 entries for the L2B banks, every entry can get 1KB data;~~
~~[9:6]~~	~~**L2B Bank ID :**~~ ~~4-bit, it selects one bank from 16;~~
~~[5:0]~~	~~**Element Selection :**~~ ~~It is 64B data space;~~
[20:0]	**L2B size** :

	It is 2MB space;

VA/LA/PA DRAM :

Address Field	Description
[37:36]	**Reserved** ;
[35:32]	**PE Cluster (PEC) ID** : 4-bit, it can select one PEC from 16;
~~[31:30]~~	~~**Sub-DRAM one-port-DRAM-space ID :**~~ ~~2-bit, it can select one Sub-DRAM one-port-DRAM-space from 4;~~
~~[29:0]~~	~~**Sub-DRAM one-port-DRAM-space size :**~~ ~~It is 1GB data space;~~
[31:0]	**DRAM size** : It is 4GB space;

~~PA L2B~~ :

~~Address Field~~	~~Description~~
~~[37:34]~~	~~**PE Cluster (PEC) ID :**~~ ~~4-bit, it can select one PEC from 16;~~
~~[33:23]~~	~~**Reserved :**~~ ;
~~[22:21]~~	~~**PE ID :**~~ ~~2-bit, it can select one PE from 4;~~
~~[20:10]~~	~~**L2B bank entry :**~~ ~~11-bit, there are 2048 entries for the L2B banks, every entry can get 1KB data;~~
~~[9:6]~~	~~**L2B Bank ID :**~~ ~~4-bit, it selects one bank from 16;~~

[5:0]	**Element Selection :** It is 64B data space;

PA DRAM :

Address Field	Description
[37:34]	**PE Cluster (PEC) ID :** 4-bit, it can select one PEC from 16;
[33:32]	**Reserved :**;
[31:30]	**Sub-DRAM one-port-DRAM-space ID :** 2-bit, it can select one Sub-DRAM one-port-DRAM-space from 4;
[29:0]	**Sub-DRAM one-port-DRAM-space size :** It is 1GB data space;

3.3.2 Configurable Format

地址编址方式动态可配置模式时，地址位宽还是48位，和固定模式不同的是，各个域段的起始地址是寄存器可配置的，方便根据实际需要来调整L2B/DRAM存储单元的大小和个数。

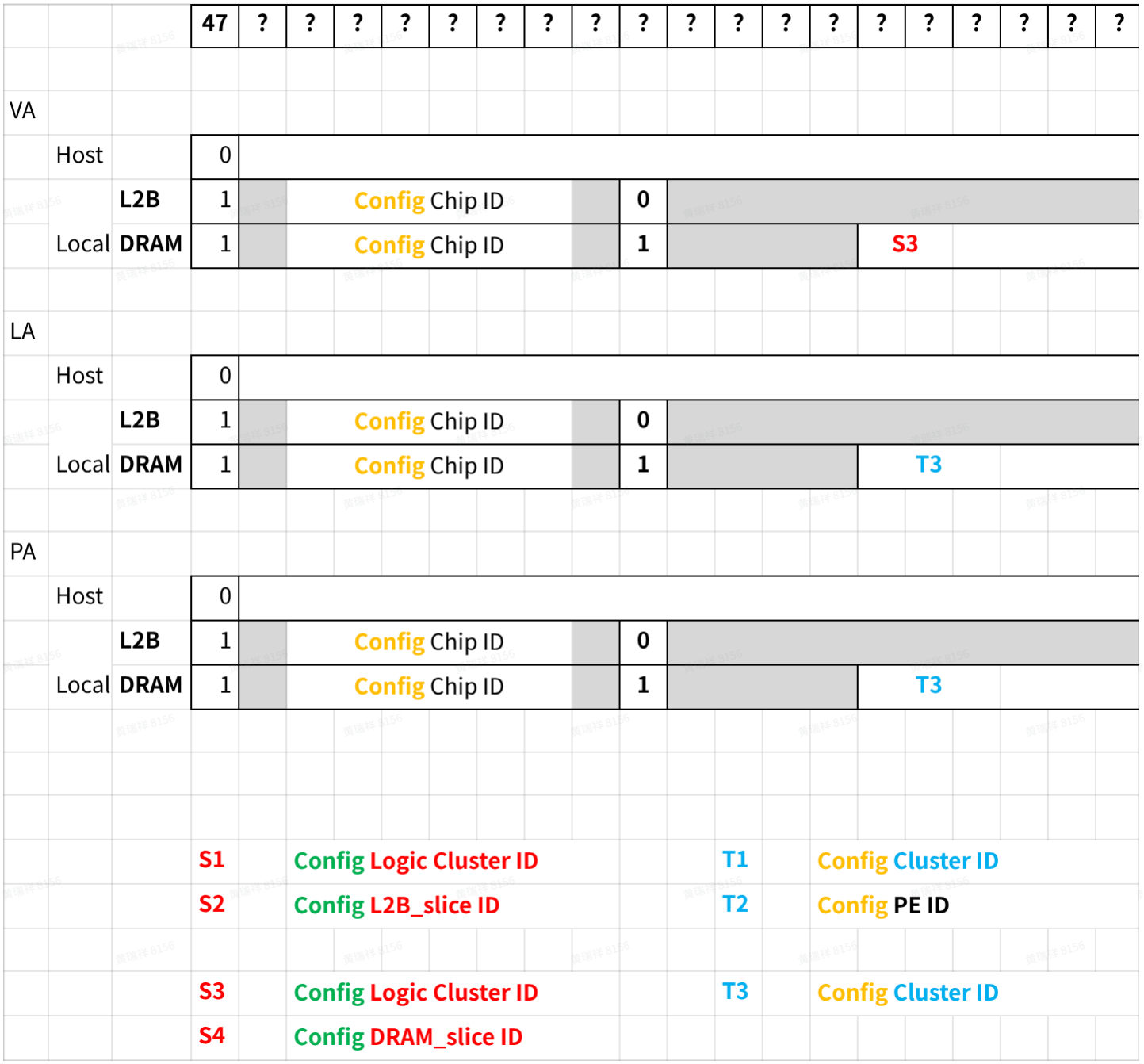
为支持地址编址动态可配置，需要硬件在设计代码时，支持规格的参数化，根据配置的参数，生成对应规格的代码。

支持这种可配置的地址编码方式的好处是，一方面是在代码开发过程中，可以快速完成芯片规格的更换，不需要重新开发硬件代码，只需要修改规格参数，就可完成新一代芯片的代码开发；另一方面，考虑到芯片回片后的良率问题，芯片内部的硬件资源可能无法全部使用，可根据芯片内部硬件资源的具体可用情况来实时更改规格，当然这种使用方式是有局限性的，在配置过程中，不能配置成超过实际物理单元的大小和个数。

除了上述的便利外，可配置的地址编码方式还可以实现支持灵活的存储单元的交织组合。

基于SIPU芯片架构的特点，~~我们定义只在VA层面支持地址格式的动态可配置，而在PA层面还是如上一小节描述的固定地址格式~~。我们定义在VA层面支持交织可配置的地址格式，而在PA层面，支持规格可配置的地址格式。当然VA层面配置的规格一定是和PA层面的定义是一致的。这就需要在硬件实现时有个专用模块进行可配置地址的解析，以及从配置的地址信息转换成固定的地址格式的功能。

地址格式如下所示：



Note :

1 在VA层面，把存储单元分成两级结构，可以配置指定有多少个“逻辑Cluster”以及每个“逻辑Cluster”的大小（S1），同时可配置可以进行交织的单位粒度以及参与交织的单位的个数（S2）；

2 在PA层面，主要体现的是规格的配置，可以指定物理Cluster的个数，物理PE的个数，以及单个基本存储单元L2B/DRAM的大小；

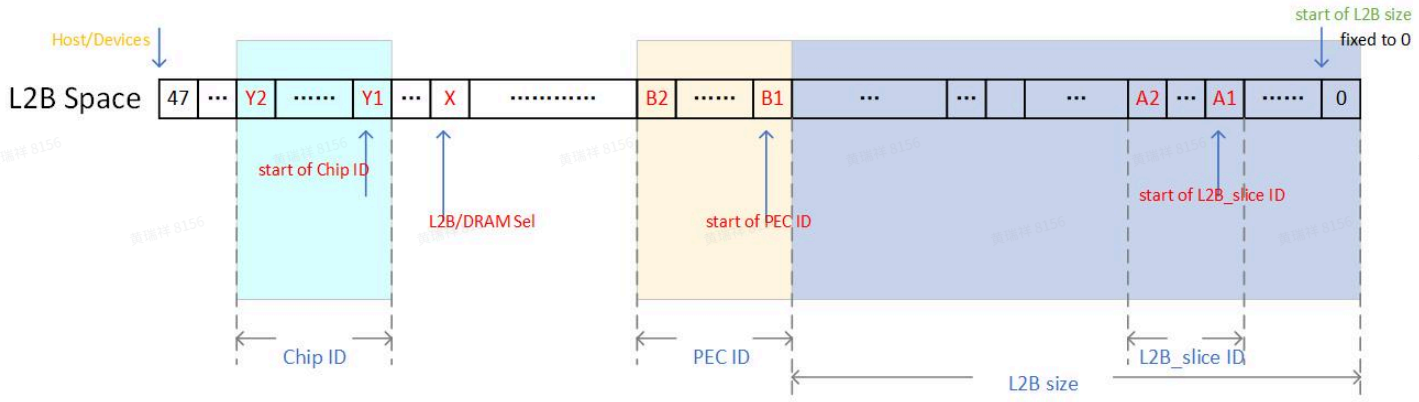
3 无论VA层面怎样配置交织方式，需要保证VA层面配置的存储单元的规格和PA层面的规格是一致的。

下面章节详细说明如何进行配置。

3.3.2.1 Configurable Address Format

下面就L2B space和DRAM space分别展开具体说明。

（1）L2B space：



上图中的[B2:B1]对应地址格式中的S1, [A2:A1]对应地址格式中的S2。

如上的地址编码格式表示：

>> 有多个Chip， Chip的个数是[Y2:Y1]能表示的最大值加1，而每个Chip 的ID 由 [Y2:Y1]表示；

>> 有多个“Logic Cluster”，Logic Cluster的个数是[B2:B1]能表示的最大值加1，每个Logic Cluster的ID由[B2:B1]表示；

>> 对于每一个Logic Cluster， 它的L2B 空间大小是 (2^B1)B大小，注意这是个Virtual L2B的大小；

>> 允许对L2B的交织使用，有多个L2B_slice，L2B_size的个数是[A2:A1]能表示的最大值加1，每个L2B_slice的ID由[A2:A1]表示；

>> 对于每一个L2B_slice，它的大小是(2^A1)B；

>> ~~支持对[A2:A1]域段表示的物理存储单元的可配置映射关系；~~

>> 允许[A2:A1]域段的取消，当配置成A2=A1=0时，表示没有[A2:A1]域段；

>> 允许[B2:B1]域段的取消，当配置成B2=B1=0时，表示没有[B2:B1]域段；

需要注意的是，即使地址格式可配，但配置的结果不能超出实际物理资源的范围，比如实际物理资源中只有64个2MB大小的L2B，如果配置成128个2MB大小的L2B，这种配置就是不合理的。

所以对配置方式是有如下限制的：

1）[B2:B1]和[A2:A1]合起来总的bit数等于6，即

(B2 - B1) + 1 + (A2 - A1) + 1 <= 6（非A2=A1=0情况，非B2=B1=0情况）；或

(B2 - B1) + 1 <= 6 （A2=A1=0）；或

(A2 - A1) + 1 <= 6 （B2=B1=0）；

2）假定[B2:B1]能表示的最大值是V1（比如[B2:B1]=[26:25]，则V1=4； [B2:B1]=[26:24]，则V1=8；），则：

V1 * (2^B1) <= 128MB;

(**Note** : 此处允许采用“<”的值，是支持低规格应用的配置)

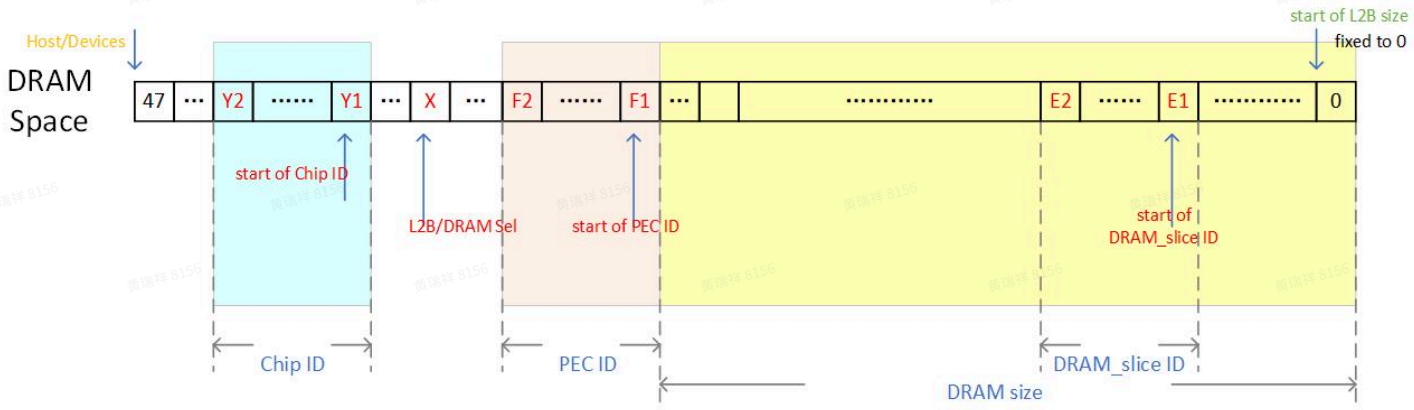
~~该地址格式是软件开发人员看到的格式，到物理设计层面，地址格式还是需要转换成固定的地址格式，转换方式如下：（以转换成固定的VA格式为例来说明）~~

~~（1）经过查询[A2:A1]映射关系配置寄存器，将[A2:A1]表示的值转换成[A2:A1]';~~

~~（2）将[B2:B1]和[A2:A1]'拼接起来，并放到地址的[26:21]的位置；~~

~~（3）将B1和A2中间的域段向下移位（A2 - A1）+ 1位；（如果A2=A1=0，则忽略该步骤）~~

（2）DRAM space :



上图中的[F2:F1]对应地址格式中的S3, [E2:E1]对应地址格式中的S4。

如上的地址编码格式表示：

>> 有多个Chip， Chip的个数是[Y2:Y1]能表示的最大值加1，而每个Chip 的ID 由 [Y2:Y1]表示；

>> 有多个“Logic Cluster”，Logic Cluster的个数是[F2:F1]能表示的最大值加1，每个Logic Cluster的ID由[F2:F1]表示；

>> 对于每一个Logic Cluster， 它的DRAM 空间大小是 (2^F1)B大小，注意这是个Virtual DRAM的大小；

>> 允许对DRAM的交织使用，有多个DRAM_slice，DRAM_size的个数是[E2:E1]能表示的最大值加1，每个DRAM_slice的ID由[E2:E1]表示；

>> 对于每一个DRAM_slice，它的大小是(2^E1)B；

>> ~~支持对[E2:E1]域段表示的物理存储单元的可配置映射关系~~；

>> 允许[E2:E1]域段的取消，当配置成E2=E1=0时，表示没有[E2:E1]域段；

>> 允许[F2:F1]域段的取消，当配置成F2=F1=0时，表示没有[F2:F1]域段；

需要注意的是，即使地址格式可配，但配置的结果不能超出实际物理资源的范围，比如实际物理资源中只有16个4GB大小的DRAM，如果配置成64个4GB大小的DRAM，这种配置就是不合理的。

所以对配置方式是有如下限制的：

1）[F2:F1]和[E2:E1]合起来总的bit数等于4，即

(F2 - F1) + 1 + (E2 - E1) + 1 <= 4 (非E2=E1=0情况，非F2=F1=0情况）；或

(F2 - F1) + 1 <= 4 （E2=E1=0）；或

(E2 - E1) + 1 <= 4 （F2=F1=0）；

2）假定[F2:F1]能表示的最大值是V2（比如[F2:F1]=[35:34]，则V2=4； [F2:F1]=[35:33]，则V2=8；），则：

V2 * (2^F1) <= 64GB;

(**Note**：此处允许采用“<”的值，是支持低规格应用的配置)

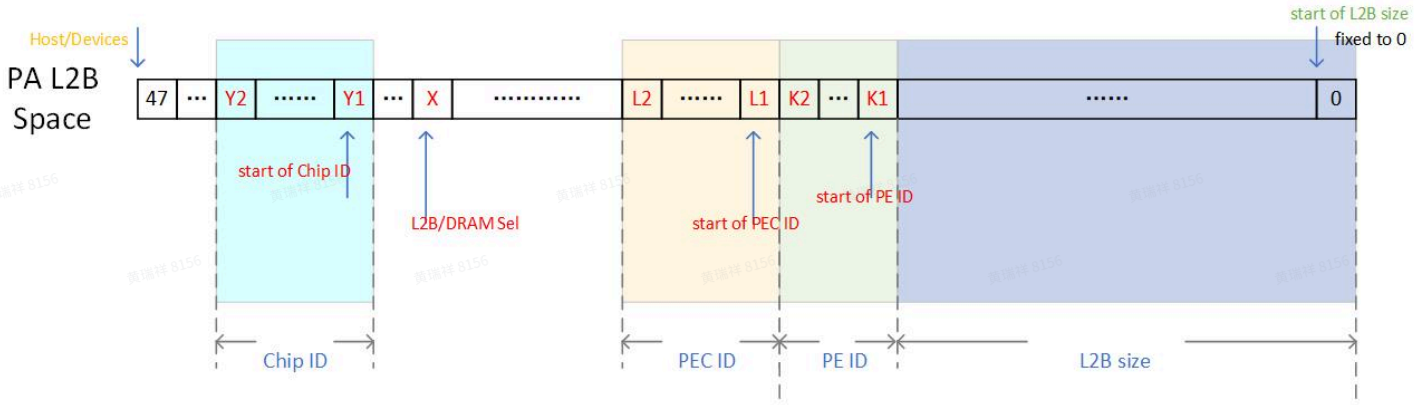
~~该地址格式是软件开发人员看到的格式，到物理设计层面，地址格式还是需要转换成固定的地址格式，转换方式如下：（以转换成固定的VA格式为例来说明）~~

~~（1）经过查询[E2:E1]映射关系配置寄存器，将[E2:E1]表示的值转换成[E2:E1]';~~

~~（2）将[F2:F1]和[E2:E1]'拼接起来，并放到地址的[35:32]的位置；~~

~~（3）将F1和E2中间的域段向下移位（E2 - E1）+ 1位；（如果E2=E1=0，则忽略该步骤）~~

（3） PA L2B space：



上图中的[L2:L1]对应地址格式中的T1，[K2:K1]对应地址格式中的T2。

如上的地址编码格式表示：

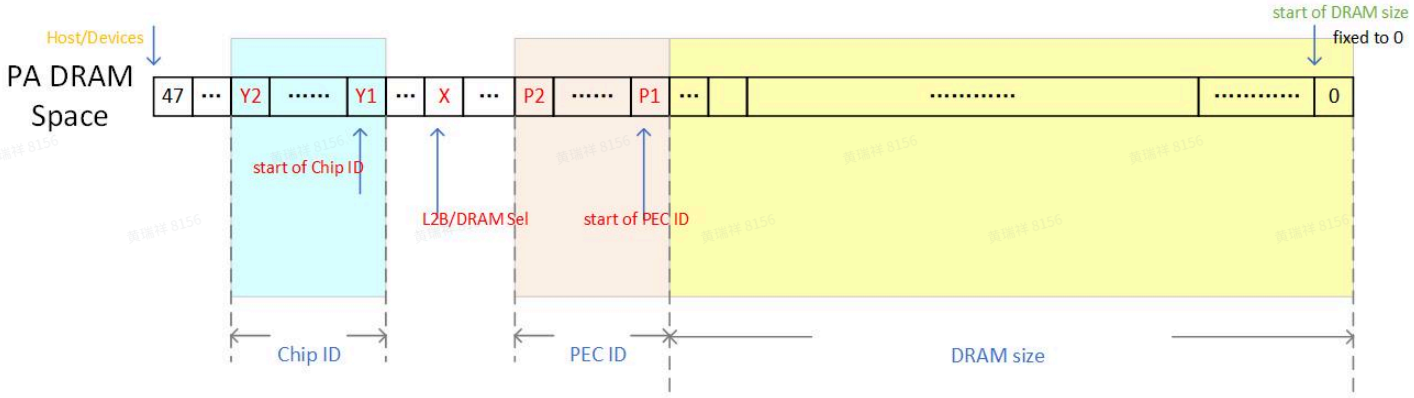
>> 有多个Cluster，Cluster的个数是[L2:L1]能表示的最大值加1，每个Cluster的ID由[L2:L1]表示；

>> 每个Cluster中有多个PE，PE的个数是[K2:K1]能表示的最大值加1，每个PE的ID由[K2:K1]表示；

>> 对于每一个PE内的L2B，它的大小是(2^K1)B；

>> 如上表示Cluster的个数，每个Cluster内PE的个数，每个PE下L2B的大小都是可配置的；

（4） PA DRAM space：



上图中的[P2:P1]对应地址格式中的T3。

>> 有多个Cluster，Cluster的个数是[P2:P1]能表示的最大值加1，每个Cluster的ID由[P2:P1]表示；

>> 对于每一个Cluster内的DRAM，它的大小是(2^P1)B；

>> 如上表示Cluster的个数，每个Cluster下DRAM的大小都是可配置的；

（5） VA和PA的匹配

对于可配置的地址格式，从VA向PA转换时，实际就是把VA中分散的ID信息拼接起来和PA中对应的位置进行匹配。

对L2B格式，把S1（[B2:B1]）/S2（[A2:A1]）拼接起来和T1（[L2:L1]）/T2（[K2:K1]）对齐；

对DRAM格式，把S3（[F2:F1]）/S4（[E2:E1]）拼接起来和T3（[P2:P1])对齐；

以下举例说明可配置地址格式的应用。

前提：

在固定地址格式模式下，软件看到的信息是一个Chip下有16个Clusters，每个Cluster内有一个4GB大小的DRAM存储单元，和4个PE，在每个PE内有一个2MB大小的L2B存储单元。

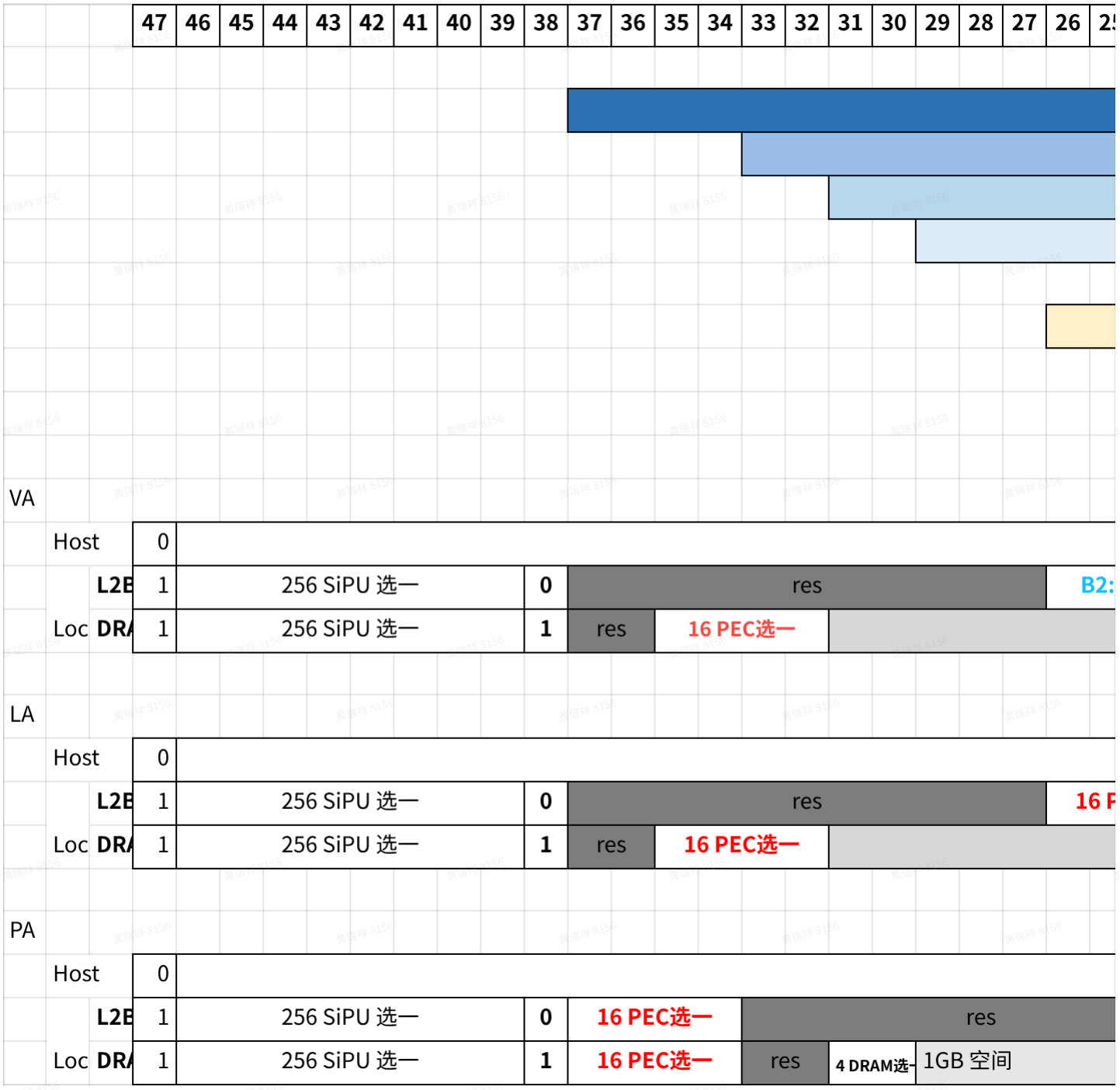
例一：软件认为一个Chip内有8个“Logic Cluster”，每个Logic Cluster的L2B size是16MB；对每个L2B，支持以1KB的L2B_slice粒度交织，共8个子单元之间交织；

配置方式如下：

X=38; Y1=39; Y2=46;

A1=10; A2=12; B1=24; B2=26;

则地址格式如下：



~~支持对[A2:A1]域段索引的可配置映射，映射关系如下：~~

原始值	映射值
~~0 00~~	~~1 01~~
~~0 01~~	~~1 00~~
~~0 10~~	~~1 11~~
~~0 11~~	~~1 10~~
~~1 00~~	~~0 11~~

~~1 01~~	~~0 10~~
~~1 10~~	~~0 01~~
~~1 11~~	~~0 00~~

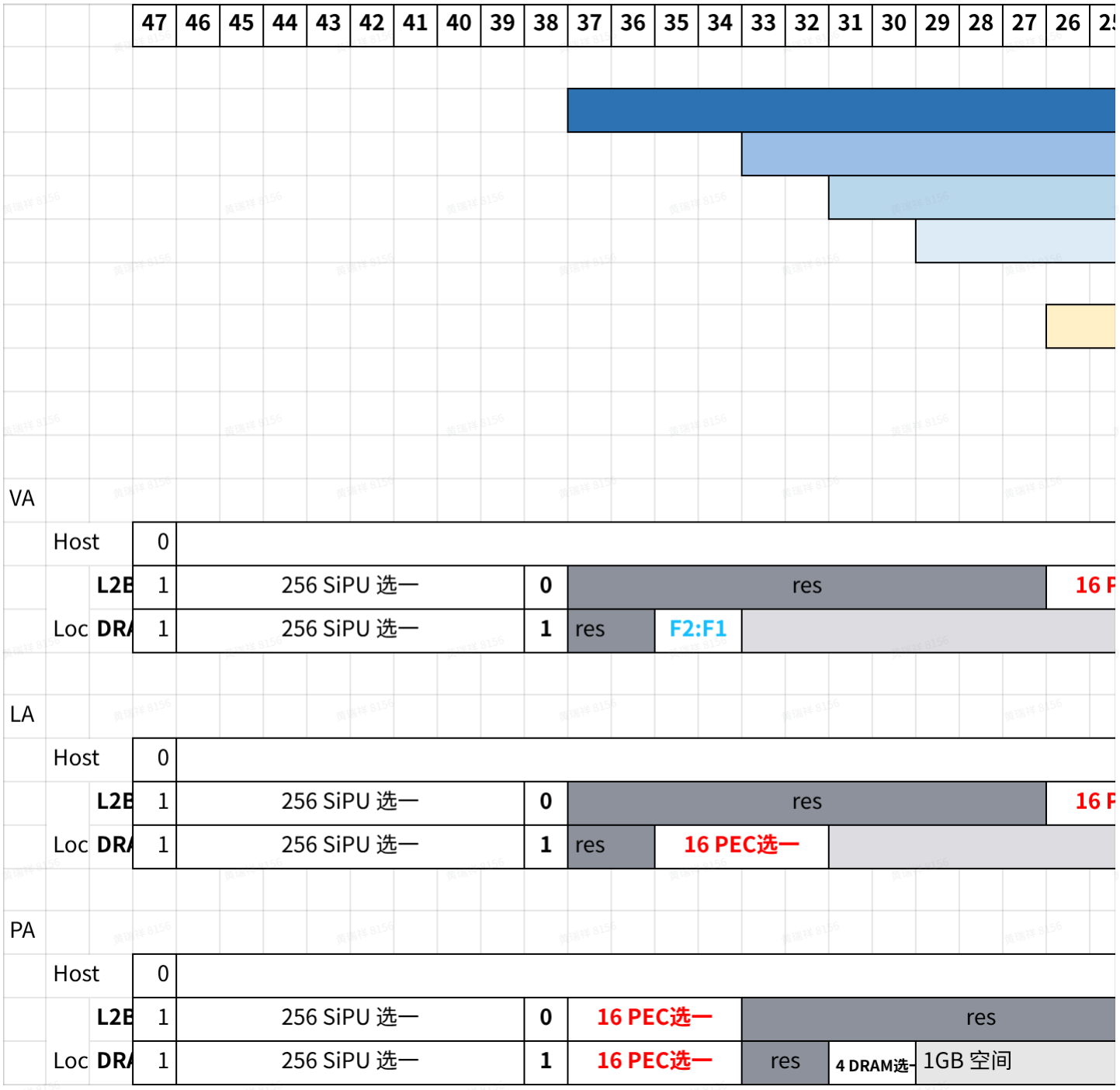
例二：软件认为一个Chip内有4个“Logic Cluster”，每个Logic Cluster的DRAM size是16GB；对每个DRAM，支持以128KB的DRAM_slice粒度交织，共4个子单元之间交织；

配置方式如下：

X=38; Y1=39; Y2=46;

E1=17; E2=18; F1=34; F2=35;

则地址格式如下：



~~支持对[E2:E1]域段索引的可配置映射，映射关系如下~~：

原始值	映射值
~~00~~	~~11~~
~~01~~	~~10~~
~~10~~	~~01~~
~~11~~	~~00~~

例三：当前降规格使用芯片，软件可认为一个Chip下有8个"Logic Cluster"， 每个“Logic Cluster”内的DRAM存储单元的大小是4GB；每个"Logic Cluster"内有4个PE，每个PE内的L2B存储单元的大小是2MB；

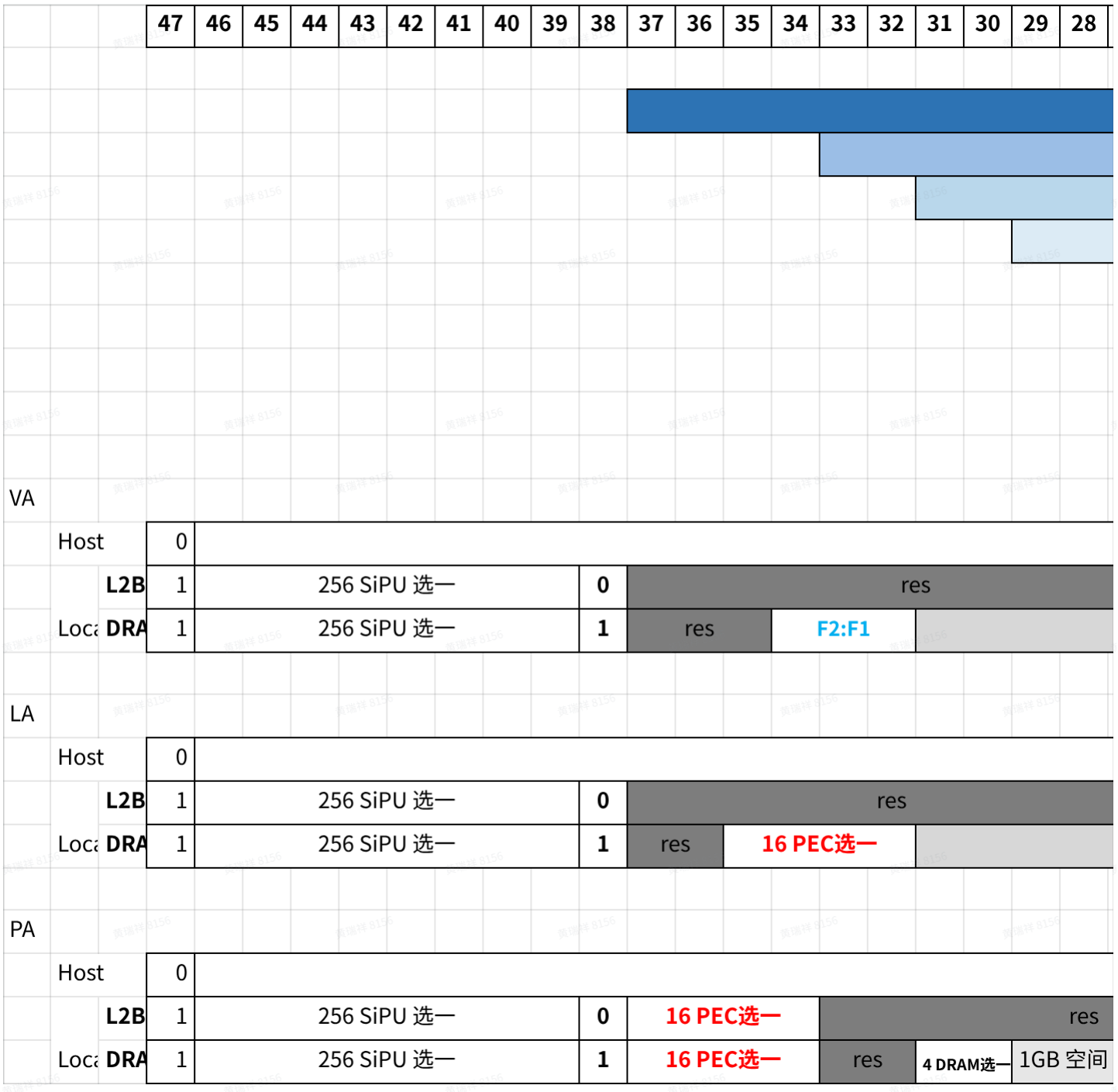
配置寄存器时:

X=38; Y1=39; Y2=46;

A1=21; A2=22; B1=23; B2=25;

E1=0; E2=0; F1=32; F2=34;

则地址格式如下：



3.3.2.2 Configuration Register

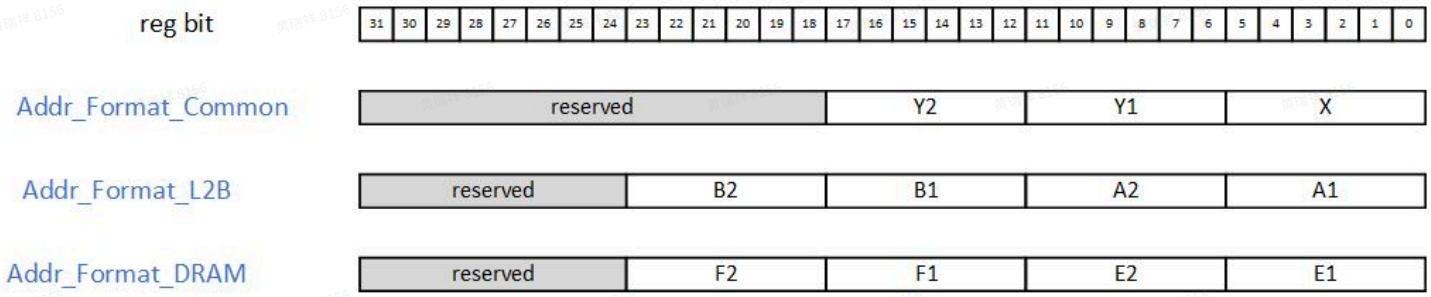
此处涉及到的配置寄存器有两类：一是指定地址中各域段的起始结束位置的配置信息；一是对交织域段映射关系的配置信息；

各域段起始结束位置配置寄存器

从上述的格式定义中可以看到，需要定义的域段有

域段含义	域段起点终点	含义描述
L2B/DRAM define	X	此位定义当前是L2B类型空间，还是DRAM类型空间 6bit表示
Chip ID域段	Y1	Chip ID域段的起始位置， 6bit;
	Y2	Chip ID域段的结束位置，6bit; 整个域段共 （Y2 - Y1 + 1）bit;
L2B空间中 L2B_slice ID域段	A1	L2B_slice ID域段的起始位置，6bit;
	A2	L2B_slice ID域段的结束位置，6bit; 整个域段共 （Y2 - Y1 + 1）bit;
L2B空间中 Logic Cluster ID域段	B1	L2B Logic Cluster ID域段的起始位置，6bit;
	B2	L2B Logic Cluster ID域段的结束位置，6bit; 整个域段共 （Y2 - Y1 + 1）bit;
DRAM空间中 DRAM_slice ID域段	E1	DRAM_slice ID域段的起始位置，6bit;
	E2	DRAM_slice ID域段的结束位置，6bit; 整个域段共 （Y2 - Y1 + 1）bit;
DRAM空间中 Logic Cluster ID域段	F1	DRAM Logic Cluster ID域段的起始位置，6bit;
	F2	DRAM Logic Cluster ID域段的结束位置，6bit; 整个域段共 （Y2 - Y1 + 1）bit;

每个位置需要6bit表示，则寄存器定义如下：



交织域段映射关系配置寄存器

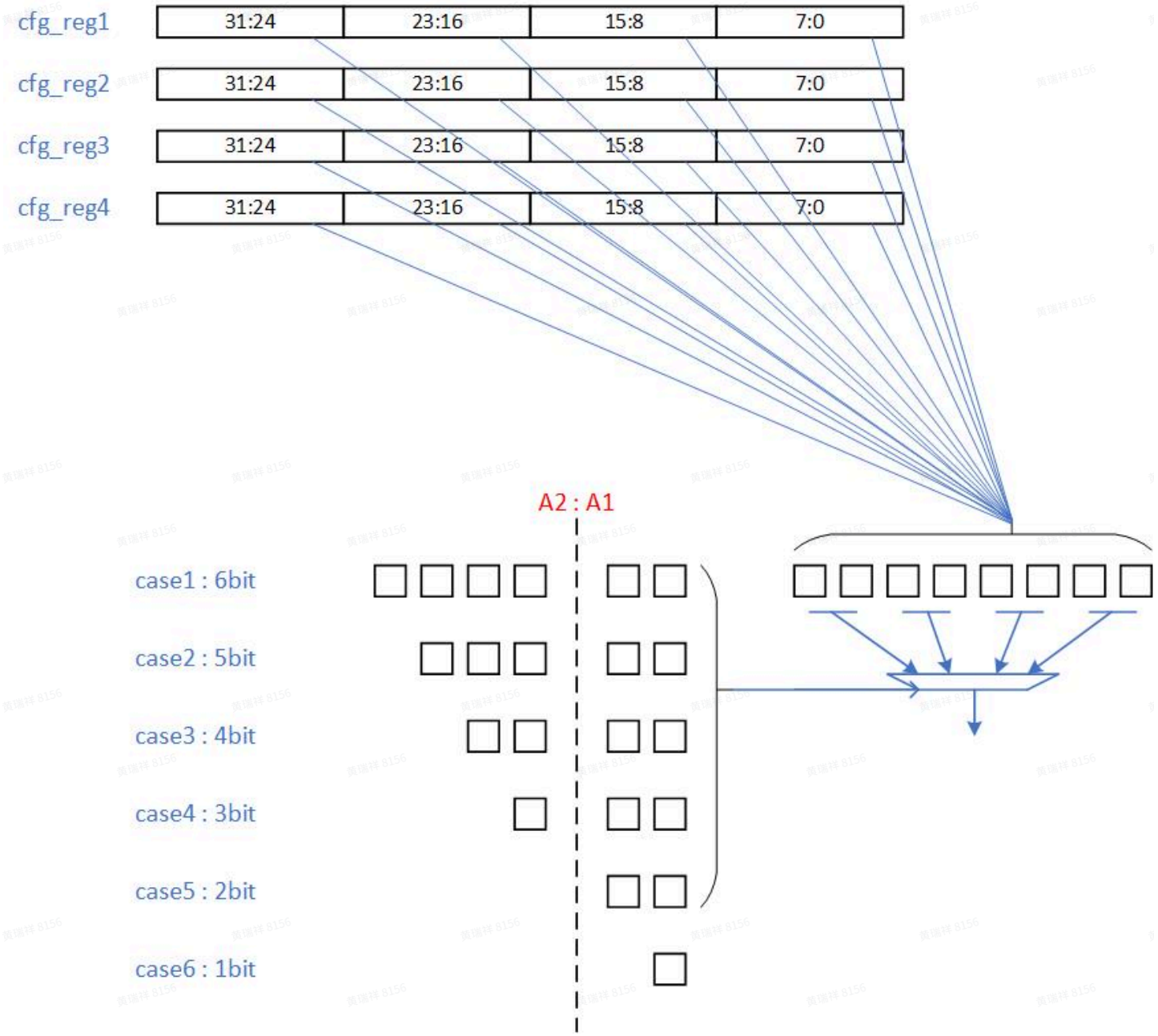
~~交织域段的映射可以采用复杂的算法来实现具体的映射关系，也可以采用简单的对应映射，这部分可以由软件灵活管理。~~

~~下面是个简单的对应映射的方式。~~

~~对L2B地址空间格式，交织域段ID[A2:A1]的位宽可以取值0~6，但因物理上每个硬件Cluster内有4个PE，所以[A2:A1]域段位宽取值0和1的场景不再考虑映射的配置，而只对取值2~6的场景进行映射配置。而且为了匹配物理上每个硬件Cluster内有4个PE，将[A2:A1]分成两段分别映射：最低2bit 和 高位bit。~~

~~最低2bit对应Cluster内4个PE的映射配置，2bit可表示四种选择，每种选择用2bit表示，共8bit 。~~

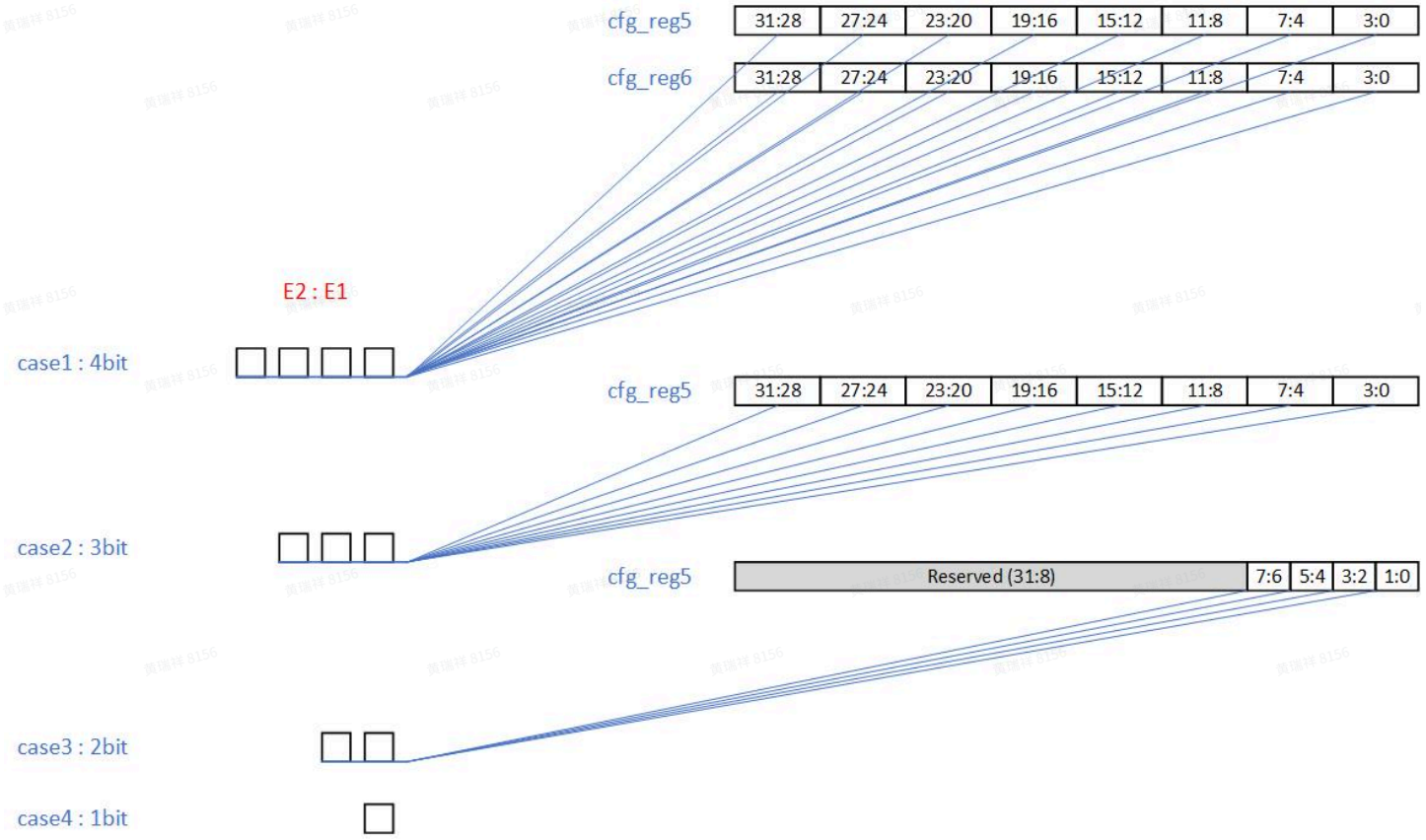
~~高位则选择寄存器中的位置。~~



~~如上图所示，最多需要4个32位宽的配置寄存器。~~

~~对DRAM地址空间格式，交织域段ID[E2:E1]的位宽可以取值0~4，对取值为0和1的场景不再考虑映射的配置，而只对取值2~4的场景进行映射配置。~~

~~位宽为4时，对应16种选择，每种需要用4bit表示；位宽为3时，对应8种选择，每种也是用4bit表示（只有低3bit有效）；位宽为2时，对应4种选择，每种用2bit表示。~~



~~如上所示，最多需要2个32位宽的配置寄存器~~。

4. Address Translation 地址转换

在SiPU芯片项目中，从硬件角度，它有两处地址产生源头：GCS模块中packet命令携带的地址信息 和PE模块（RV core）中程序指令携带的地址信息。

无论是packet命令携带的地址信息，还是程序指令携带的地址信息，它们都是VA的地址信息，在硬件的执行单元中传递时，需要将其转换成PA的地址信息。

地址信息转换涉及以下几个方面：

1） VA --> LA： Cluster ID Mapping : 将虚拟Cluster ID转换成物理Cluster ID;

~~2） LA --> PA(IPA) : Discreted Address Mapping : 将地址进行离散转换；~~

3） LA --> PA(IPA)： Multi-Process DRAM Address Mapping : 多进程并行模式时地址的页表映射；

此处提到一个IPA (Intermediate Physical Address)的概念，这个主要是针对如下场景：在存储单元内部可能会有自己独有的更底层的一些地址处理，存储单元会认为经过这些处理后生成的地址才是真正的物理地址。

为支持如上的场景，我们作如下定义：

如果下层存储单元内部没有其他地址处理的需求，认为经过上述地址转换后的地址就是物理地址，则该地址转换的输出是PA；

如果下层存储单元内部还有其他地址处理的需求，不认为经过上述地址转换后的地址是物理地址，则该地址转换的输出是IPA；

Note : 为方便描述，下文的描述中都以地址转换后的输出是PA来描述。

下面章节将详细介绍这些地址信息转换的方式。

4.1 VA --> LA : Cluster ID Mapping （Cluster ID映射）

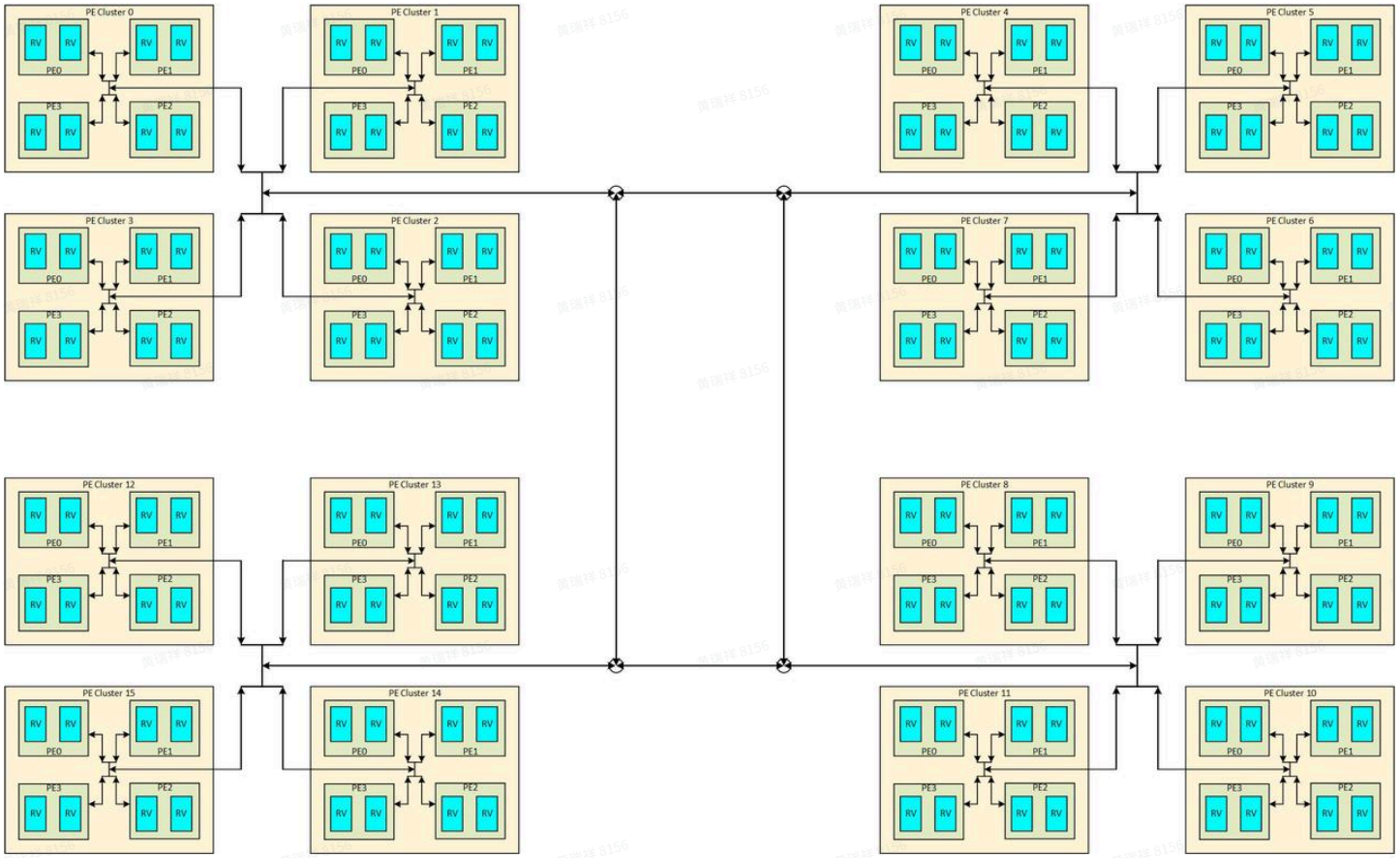
在SiPU芯片项目中，它支持虚拟Cluster ID到物理Cluster ID的映射转换功能。

该功能主要是为了支持SiPU的多进程并行执行模式。从”Address Information Format“章节我们可以了解到，在SiPU中，地址信息可大致分为两部分：存储单元定位和单个存储单元内寻址。存储单元的定位是通过Chip ID, PE Cluster ID, PE ID这些信息来实现的；单个存储单元内寻址则是通过低位的internal address来实现的。本章节介绍的Cluster ID Mapping，针对的就是地址信息中用于存储单元定位的ID信息。

在SiPU芯片项目中，~~单个芯片内包含~~最大的scale-up系统规模是256个Chips，单Chip内包含16个PE Clusters，单个PE Cluster内实际可用PE个数为4，而单个PE内包含2个RV core。~~这两个RV core可以执行相同的进程，也可以分别执行不同的进程~~。单个PE内的2个RV core是执行相同的进程。

下图示意了一个SiPU chip的内部结构。每个PE Cluster和其下的每个PE都有自己的编号。所有16个PE Cluster统一进行编号，而每个PE Cluster内部的4个PE分别独立编号。

在实际的物理芯片上，每个PE Cluster内部的4个PE的物理ID支持寄存器可配置，PE Cluster的物理ID也支持寄存器可配置。在此，有一个”Super Cluster“的概念，即将16个PE Clusters按照4个一组，分成4组。每组内的4个PE Cluster在物理上更靠近。因此，在对PE Cluster进行物理ID配置时，建议同一个Super Cluster内的Cluster是连续的物理ID。

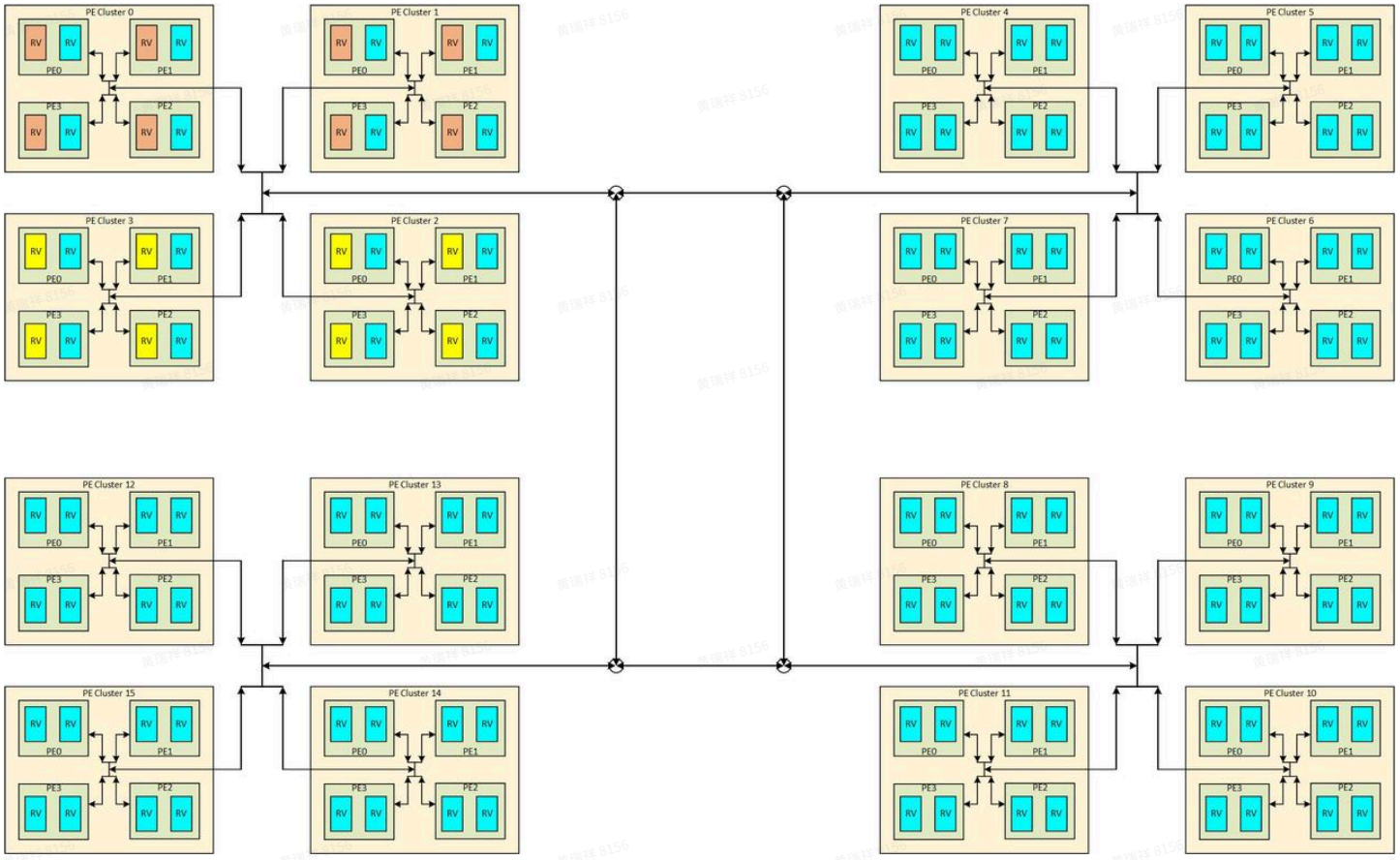


在多进程模式执行时，每个进程可占用多个PE Clusters，但软件编程人员不需要清楚每个进程具体对应哪个物理Cluster。例如，当前有两个进程：进程A（图中桔色块）和进程B（图中黄色块）。进程A和进程B都指定需要两个Clusters来执行任务。进程A指定使用Cluster ID0和Cluster ID1，进程B也同样指定使用Cluster ID0和Cluster ID1。此时的Cluster ID都是虚拟的ID，在芯片硬件上，需要将其映射到具体的物理Cluster上。

Cluster ID的映射可以有多种方式，（1）可以映射到同一个Super Cluster内不同的Cluster上，（2）也可以映射到不同的Super Cluster上，（3）或者根据Cluster的空闲情况，两个进程分配到同一个物理Cluster上。

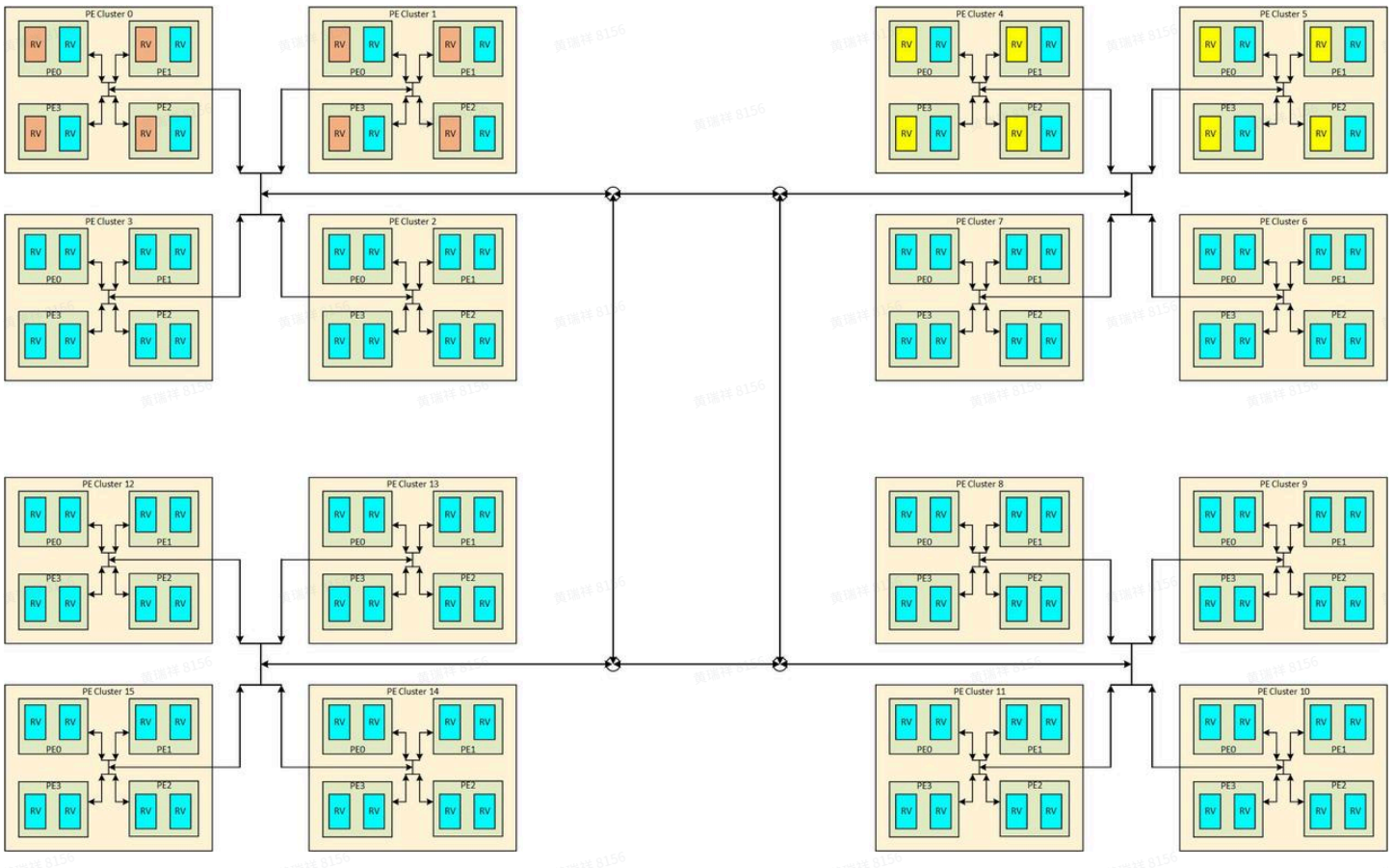
（1）两个进程分别占用同一个Super Cluster的不同物理Cluster

进程A使用物理Cluster 0 和 Cluster 1；进程B使用物理Cluster 2 和 Cluster 3；



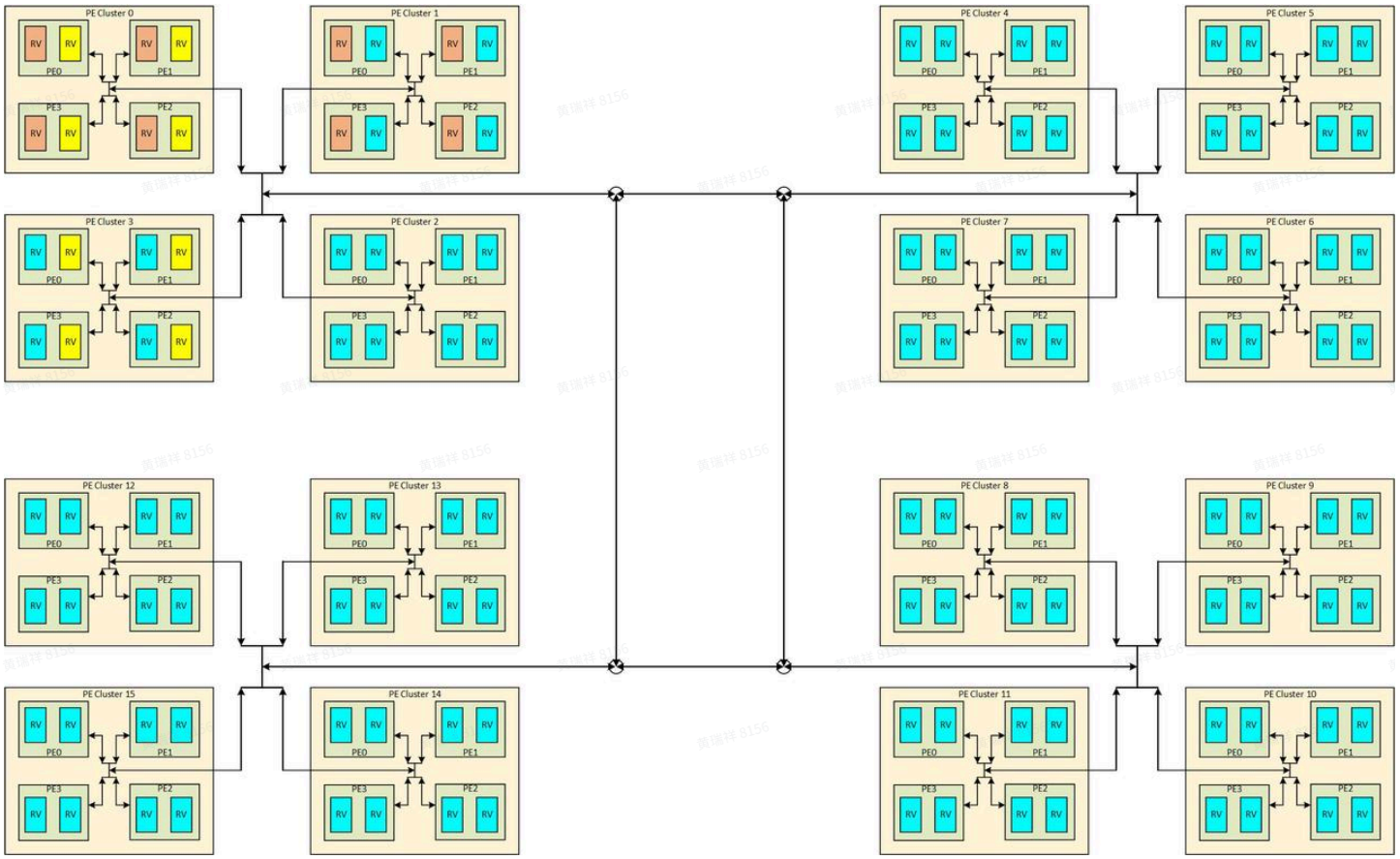
（2）两个进程占用不同的Super Cluster的物理Cluster

进程A使用物理Cluster 0 和 Cluster 1； 进程B使用物理Cluster 4 和 Cluster 5;

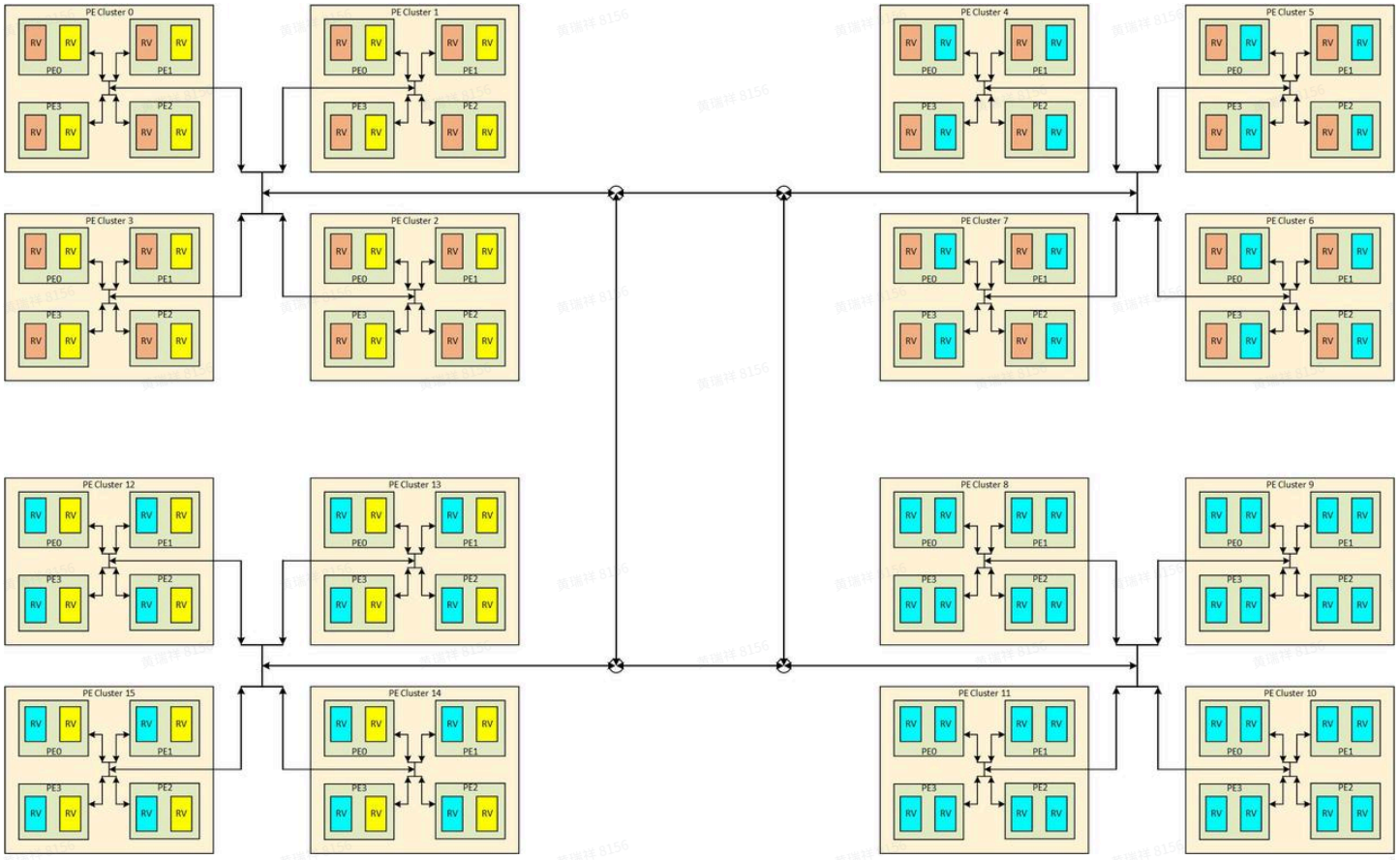


（3）两个进程部分使用相同的物理Cluster

进程A使用物理Cluster 0 和 Cluster 1；进程B使用物理Cluster 0 和 Cluster 3；在Cluster 0，进程A和进程B分别使用不同的RV-core。



进程对Cluster的占用也可以以Super Cluster为基本单位，如下示意以Super Cluster的粒度进行进程的Cluster分配情况。



无论是上述哪种分配情况，都需要系统在启动一个新的进程前，对硬件进行虚拟Cluster ID到物理Cluster ID的映射配置，并在进程生存周期内保持不变。

是否需要这个映射关系，以及单个进程可使用的Cluster个数是和同时可并行的进程个数相匹配的，这个匹配关系如下表所示。

模式	并行进程个数	单个进程可使用Cluster个数	是否需要映射表	是否支持	模式说明 （64个PE，2 RV-core/PE）
Model1	128	1	N	N	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model2	64	1	N	N	同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model3	64	2	Y	N	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model4	32	2	Y	N	同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model5	32	4	Y	N	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；

Model6	32	1	N	N	同PEC的4个PE相同进程，同PE的2个RV-core不同进程；
Model7	16	4	Y		同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model8	16	8	Y	?	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model9	16	1	N		同PEC的4个PE相同进程，同PE的2个RV-core相同进程；
Model10	16	2	Y	?	同PEC的4个PE相同进程，同PE的2个RV-core不同进程；
Model11	8	8	Y		同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model12	8	16	Y	?	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model13	8	2	Y		同PEC的4个PE相同进程，同PE的2个RV-core相同进程；
Model14	8	4	Y	?	同PEC的4个PE相同进程，同PE的2个RV-core不同进程；
Model15	4	16	Y		同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model16	4	16	Y	?	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model17	4	4	Y		同PEC的4个PE相同进程，同PE的2个RV-core相同进程；
Model18	4	8	Y	?	同PEC的4个PE相同进程，同PE的2个RV-core不同进程；
Model19	2	16	Y		同PEC的4个PE不同进程，同PE的2个RV-core相同进程；
Model20	2	16	Y	?	同PEC的4个PE不同进程，同PE的2个RV-core不同进程；
Model21	2	8	Y		同PEC的4个PE相同进程，同PE的2个RV-core相同进程；
Model22	2	16	Y	?	

					同PEC的4个PE相同进程，同PE的2个RV-core不同进程；

Note : 上表中单个进程可使用的Cluster个数是按照每个并行的进程都拥有相同的Cluster个数来均分的，如果并行的进程各自所需的Cluster个数不同，也是支持的，系统配置时需要保证所有并行进程所占用的Cluster个数总和不超过实际物理Cluster个数。

如下是一些支持的场景的说明：

>> 同一个进程可以分布到不同的Chip上，每个Chip都可以有自己的Cluster ID映射关系；

>> 目前定义最大可并行的进程个数是8；

>> 支持”同PE的2个RV-core同一时间执行不同进程“；

4.1.1 Cluster ID映射实现步骤

Cluster ID映射的实现步骤如下：

步骤一： 芯片上电后，驱动会通过配置寄存器配置好各物理Cluster的物理ID信息；

步骤二： 启动一个进程之前，系统需要首先完成Cluster ID映射表的配置；（见4.1.2章节）

步骤三： 系统启动进程；

步骤四： 进程的Kernel进入到GCS模块，GCS会通过查询GCS中该进程所对应的Cluster ID映射表获取真实的物理Cluster信息，然后将该Kernel信息发送到正确的物理Cluster上去；

步骤五： Kernel在PE中执行时，程序指令中的访存指令会给出存储单元的访问地址，该地址中的Cluster ID信息也是虚拟ID，也是需要通过查询PE中事先配置好的Cluster ID映射表来获取真实的物理Cluster信息，然后发送到正确的物理Cluster内的存储单元；

步骤六： 在整个进程生命周期内，配置的Cluster ID映射表应保持不变，直到该进程全部完成。如果要启动新的进程，将重复步骤二到步骤六；

4.1.2 Cluster ID映射表的配置

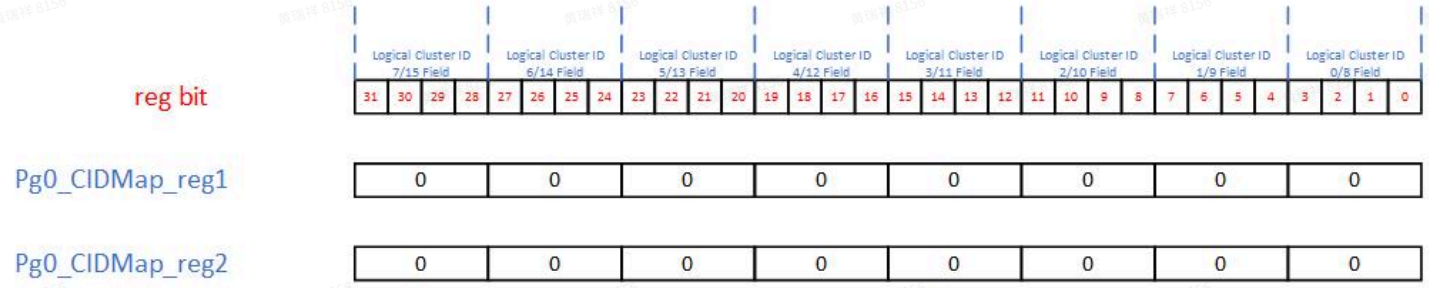
Cluster ID映射表需要在进程启动前，就由系统配置好，并且在进程生命周期内保持不变。另外，系统需要保证后续该进程的Kernel中指定的Cluster的使用个数和配置的个数保持一致，不能超出配置的映射表中的Cluster的个数。

Cluster ID的映射表配置~~可以有多种方式，此处先按照使用配置寄存器的方式进行描述。~~ 采用配置寄存器的方式。

在一个Chip内，需要进行地址转换的模块都需要Cluster ID映射表，这些模块包括： GCS模块， PE模块，C2C模块和Media模块。这些模块都需要分别进行配置。

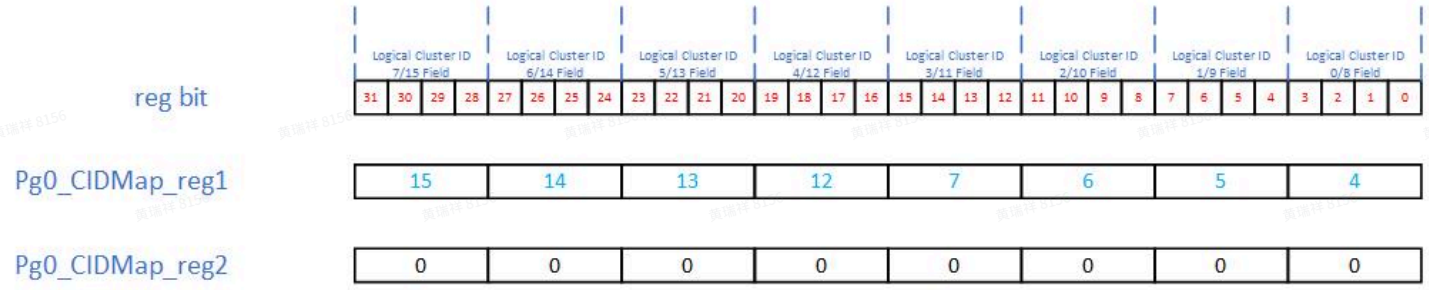
4.1.2.1 Cluster ID映射关系配置寄存器格式

一个进程最多可使用16个Clusters，每个映射的Cluster ID用4bit表示，则每个进程使用2个32-bit的配置寄存器就可以实现。如图：



这些寄存器各个域段的初始值都为全0值，如果要使能某个进程，则将该进程对应的寄存器配置成需要的映射关系即可。

举例，如下是使能进程0的例子。该进程会占用8个Clusters。从软件角度，该进程的Cluster ID 是0~7，即VCID0~VCID7 (VCID : Virtual Cluster ID)；该进程实际占用的物理Cluster是编号4~7和12~15的8个Clusters，即PCID4~PCID7和PCID12~PCID15 (PCID : Physical Cluster ID)。两者对应关系如下：VCID0-->PCID4，VCID1-->PCID5，VCID2-->PCID6，VCID3-->PCID7，VCID4-->PCID12，VCID5-->PCID13，VCID6-->PCID14，VCID7-->PCID15。



在SiPU芯片项目中，最大可并行进程数是8，则需要8组上述的寄存器。

reg bit	Logical Cluster ID 7/15 Field				Logical Cluster ID 6/14 Field				Logical Cluster ID 5/13 Field				Logical Cluster ID 4/12 Field				Logical Cluster ID 3/11 Field				Logical Cluster ID 2/10 Field				Logical Cluster ID 1/9 Field				Logical Cluster ID 0/8 Field			
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Pg0_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg0_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg1_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg1_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg2_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg2_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg3_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg3_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg4_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg4_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg5_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg5_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg6_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg6_CIDMap_reg2	0				0				0				0				0				0				0				0			
Pg7_CIDMap_reg1	0				0				0				0				0				0				0				0			
Pg7_CIDMap_reg2	0				0				0				0				0				0				0				0			

4.1.2.2 *GCS模块*

在GCS模块中，无论是进行TBC调度，还是执行DMA类型packet命令，都需要执行从虚拟Cluster ID到物理Cluster ID的映射转换，所以GCS模块需要这个Cluster ID映射信息。

GCS模块有自己的Cluster ID映射关系配置寄存器空间（8x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

GCS模块的Cluster ID映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.1.2.1章节。

4.1.2.3 *PE模块*

在PE模块中，无论是RV-core的LSU操作，还是Tile-core的LSU操作或者数据搬移操作，都需要执行从虚拟Cluster ID到物理Cluster ID的映射转换，所以PE模块需要Cluster ID映射信息。

对PE模块中的Cluster ID映射信息的配置有以下方案：

方案A）采用寄存器配置方式，同Cluster中的PE模块统一配置；

方案B）跟随GCS拆分出的TBC命令随路传送；

方案C）采用寄存器配置方式，每个PE模块都独立配置；

方案A： 采用寄存器配置方式，同Cluster中的PE模块统一配置

该方案中，每个PE Cluster共用一组配置寄存器，只需要16组寄存器（16x8x2个32位配置寄存器）即可。如果约定同PEC内的4个PE只能执行相同进程，不存在问题；但如果同PEC的4个PE可以执行不同的进程，可能会引入不同进程抢占了相同的PE执行单元的情况。例如：进程A被映射到物理Cluster0，因为同PEC的不同PE可以执行不同进程，进程B也允许被映射到物理Cluster0。如果分配不当，有可能进程A和进程B都占用了PE0这个执行单元，从而导致进程冲突。

为了解决这个问题，软件需要在给进程分配PE时，显式的避开冲突，但因为进程的Cluster ID是逻辑的ID，而不是物理ID，所以还是有可能分配冲突。为此，需要引入PE ID的映射表配置，这会引起更多的配置寄存器使用。如果再考虑到同PE内2个RV-core可以并行执行不同的进程，那情况会更为复杂。

方案B： 跟随GCS拆分出的TBC命令随路传送

该方案中，如上所说的寄存器不再是配置寄存器，而是硬件内部寄存器，软件编程不可见。

无论是什么进程，其是以Kernel的方式通知硬件去执行操作的。在硬件结构中，每个Kernel的packet命令会被拆分成TBC命令，并被下发给具体的PE Cluster执行。在下发时，硬件会根据进程ID信息查询Cluster ID映射关系，并获知真正的物理Cluster ID。在下发这个TBC命令时，可以把该进程配置的Cluster ID映射关系随命令一起传递下去。该命令会进入到某个PE Cluster，然后进入到某个PE，最后到达某个RV-core。映射关系也一路传递，最终被传送到最底层的RV-core，并被RV-core保存下来，在该kernel执行期间保持有效。

采用这种方式，即使同PEC下不同PE执行不同的进程，硬件也是可以准确获知该进程所对应的Cluster ID映射关系，而不需要软件层面显式的管理分配。

每个进程的Cluster ID映射关系用64位就可以表达。当前的TBC命令带宽完全可以容纳这些信息。

方案C： 采用寄存器配置方式，每个PE模块都独立配置

该方案中，上述所说寄存器是配置寄存器。

每个PE模块都需要有自己的Cluster ID映射配置寄存器空间（8x2个32位配置寄存器），在进程启动前，系统需要把每个PE模块的这组配置寄存器配置好。

单个Chip内有64个PE模块，共64x8x2个32位配置寄存器。寄存器格式见4.1.2.1章节。

单从Cluster ID映射关系的配置来看，方案B和方案C都是可行的，但考虑到后面章节的DRAM地址映射关系的配置，还是建议采用方案C。

4.1.2.4 C2C模块

在C2C模块中，无论是Host发起的packet命令，还是程序指令流产生的跨Chip数据搬移指令，执行数据搬移操作时，对本Chip内的存储单元进行访问时，都需要执行从虚拟Cluster ID到物理Cluster ID的映射转换，所以C2C模块需要这个Cluster ID映射信息。

C2C模块有自己的Cluster ID映射关系配置寄存器空间（8x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

C2C模块的Cluster ID映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.1.2.1章节。

4.1.2.5 Media模块

在Media模块中，当执行数据搬移操作时，需要执行从虚拟Cluster ID到物理Cluster ID的映射转换，所以Media模块需要这个Cluster ID映射信息。

Media模块有自己的Cluster ID映射关系配置寄存器空间（8x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

Media模块的Cluster ID映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.1.2.1章节。

4.1.3 进程启动和Kernel Launch

在系统配置好映射表寄存器后，系统就可以启动进程了。

每个进程可能会包含多个Kernel，每个Kernel可以只使用一个Cluster，也可以使用多个Cluster（使用的Cluster个数是在该进程占用的Cluster总个数之内），这个信息由Kernel自身携带，并且Kernel信息中会指定自己需要使用的Cluster编号，当然，这个Cluster编号是虚拟ID，需要通过查询Cluster ID映射关系配置寄存器来获知物理Cluster ID信息。

~~除了携带Cluster虚拟ID信息外，Kernel还会携带可用RV-core范围的信息，该信息是一个8位信息，分别对应4个PE和每个PE下的两个RV-core，格式如下：~~

~~[1:0]对应PE0的RV-core1 和 RV-cove0, [3:2]对应PE1的RV-core1 和 RV-core0，依次类推，[7:6]对应PE3的RV-core1 和 RV-core0 。对应的位置如果被置为1，则表示当前Kernel可以在该PE的该RV-core上执行，如果该位置的值是0，表示不能把当前Kernel发送到该PE的该RV-core上执行。硬件会根据这个信息，选择一个可用的RV-core来下发这个Kernel执行。~~

~~在实际硬件实现时，一个Kernel对应的是一个packet命令。硬件会将这个packet拆分成多个TBC，在Kernel信息中会给每个TBC指定其虚拟Cluster ID，但整个packet会有一个公共的可用RV-core范围的信息。硬件会逐个TBC的进行调度，先查询Cluster ID映射表获取该TBC的物理Cluster ID，然后根据可用RV-core范围信息，获取其可使用的PE和RV-core信息，然后根据PE/RV-core的空闲/忙碌状态，从中挑选一个可用的RV-core下发该TBC。~~

~~在PE/RV-core这个层面的调度是由硬件来决定的，软件不会显式参与，只需要把Kernel的可用RV-core范围信息配置好（建议是8位全部置1）即可。如果软件需要显式的参与控制，需要保证好如下两方面的管理：1）如果两个进程映射到同一个物理Cluster，需要记录各自占用的PE/RV-core，避免不同进程分配给同一个PE/RV-core；2）软件实时获知各个PE/RV-core的空闲状态，避免PE/RV-core分配不均衡。~~

4.1.4 Cluster ID映射表的查询

4.1.4.1 *GCS模块*

在GCS模块中，在进行Kernel Launch时，它需要先经过查询Cluster ID映射表来获取物理Cluster ID信息。查询步骤如下：

步骤一：GCS获取到新的Kernel信息（即packet命令），对其开始进行调度。该Kernel所属的进程ID事先已经定义好；

步骤二：GCS解析该Kernel信息，获知其包含的TBC个数，，每个TBC的虚拟Cluster ID，以及该Kernel的可用RV-core范围信息；

步骤三：对于每一个TBC，GCS先根据进程ID信息从8组寄存器中选中当前进程所对应的寄存器组，然后根据该TBC指定的虚拟Cluster ID，选择该组寄存器对应的域段（比如虚拟Cluster ID是3，则选择第一个寄存器的[15:12]），从而获知其对应的物理Cluster ID；

步骤四：根据该Kernel的可用RV-core范围信息，可获知TBC可以发送给选中的物理Cluster的哪些PE/RV-core。GCS根据记录的各个PE/RV-core的空闲状态，选择一个最空闲的PE/RV-core（如果一个RV-core可以接收多个TBC，则选择正在执行TBC个数最少的）。如果多个RV-core都空闲，则可按照PE0/RV-core0 -> PE0/RV-core1 -> PE1/RV-core0 -> PE1/RV-core1 -> PE2/RV-core0 -> PE2/RV-core1 -> PE3/RV-core0 -> PE3/RV-core1的顺序依次选择；

步骤五：通过步骤三和步骤四已经选中了某个物理Cluster的某个PE的某个RV-core，GCS会把TBC信息下发到选中的RV-core；

步骤六：重复步骤三到步骤五，直到该Kernel所有的TBC都下发完成；

步骤七：后续如果还有新的Kernel，重复执行步骤一到步骤六；

Note : *上述描述的执行过程是对一个进程已经配置好Cluster ID映射关系后的步骤，如果一个进程执行完，在同进程Queue中要切换新的进程，需要等前面进程的所有Kernel已经全部执行完（不只是下发*

完TBC，而是真正的被执行单元执行完），之后重新配置新进程的Cluster ID映射寄存器，然后再开始Kernel的执行。

4.1.4.2 *PE模块*

见4.4.2章节

4.1.4.3 C2C模块

见4.4.3章节

4.1.4.4 Media模块

见4.4.4章节

4.2 ~~LA --> PA : Discreted Address Mapping (离散地址映射)~~

~~从“Address Space”章节的描述我们知道，在SiPU芯片项目中，LA/VA的地址空间分布方式和PA的地址空间分布方式是不同的。软件编程人员和用户看到的地址空间，对单个Chip而言，它的L2B地址空间和DRAM地址空间分别都是连续的，但在实际的物理存储单元中，16个PE Clusters分别均分了高低256GB的L2B地址空间和DRAM地址空间，每个PE Cluster都独占了16GB的地址空间，实际的L2B和DRAM空间是不连续的。~~

~~以Chip0为例，LA上，0x8000_0000_0000 ~ 0x8000_07FF_FFFF表示的是L2B地址空间，0x8040_0000_0000 ~ 0x804F_FFFF_FFFF表示的是DRAM地址空间。但在PA上，L2B地址空间对应的是 0x80000_0000_0000 ~ 0x8000_007F_FFFF，加上 0x8004_0000_0000 ~ 0x8004_007F_FFFF，加上0x8008_0000_0000 ~ 0x8008_007F_FFFF .....，直到加上 0x803C_0000_0000 ~ 0x803C_007F_FFFF 。~~

~~L2B 地址空间：~~

(**~~Note~~**: ~~LA的[37:27]，以及PA的[33:23]都是reserved的，值固定为全0~~)

~~Chip ID~~	~~LA L2B~~	~~PA L2B~~
~~Chip0~~	~~0x8000_0000_0000~0x8000_007F_FFFF~~	~~0x8000_0000_0000~0x8000_007F_FFFF~~
	~~0x8000_0080_0000~0x8000_00FF_FFFF~~	~~0x8004_0000_0000~0x8004_007F_FFFF~~
	~~0x8000_0100_0000~0x8000_017F_FFFF~~	~~0x8008_0000_0000~0x8008_007F_FFFF~~
	~~0x8000_0180_0000~0x8000_01FF_FFFF~~	~~0x800C_0000_0000~0x800C_007F_FFFF~~

	0x8000_0200_0000~0x8000_027F_FFFF	0x8010_0000_0000~0x8010_007F_FFFF
	0x8000_0280_0000~0x8000_02FF_FFFF	0x8014_0000_0000~0x8014_007F_FFFF
	0x8000_0300_0000~0x8000_037F_FFFF	0x8018_0000_0000~0x8018_007F_FFFF
	0x8000_0380_0000~0x8000_03FF_FFFF	0x801C_0000_0000~0x801C_007F_FFFF
	0x8000_0400_0000~0x8000_047F_FFFF	0x8020_0000_0000~0x8020_007F_FFFF
	0x8000_0480_0000~0x8000_04FF_FFFF	0x8024_0000_0000~0x8024_007F_FFFF
	0x8000_0500_0000~0x8000_057F_FFFF	0x8028_0000_0000~0x8028_007F_FFFF
	0x8000_0580_0000~0x8000_05FF_FFFF	0x802C_0000_0000~0x802C_007F_FFFF
	0x8000_0600_0000~0x8000_067F_FFFF	0x8030_0000_0000~0x8030_007F_FFFF
	0x8000_0680_0000~0x8000_06FF_FFFF	0x8034_0000_0000~0x8034_007F_FFFF
	0x8000_0700_0000~0x8000_077F_FFFF	0x8038_0000_0000~0x8038_007F_FFFF
	0x8000_0780_0000~0x8000_07FF_FFFF	0x803C_0000_0000~0x803C_007F_FFFF
Chip1	0x8080_0000_0000~0x8080_007F_FFFF	0x8080_0000_0000~0x8080_007F_FFFF
	0x8080_0080_0000~0x8080_00FF_FFFF	0x8084_0000_0000~0x8084_007F_FFFF
	0x8080_0100_0000~0x8080_017F_FFFF	0x8088_0000_0000~0x8088_007F_FFFF
	0x8080_0180_0000~0x8080_01FF_FFFF	0x808C_0000_0000~0x808C_007F_FFFF
	0x8080_0200_0000~0x8080_027F_FFFF	0x8090_0000_0000~0x8090_007F_FFFF
	0x8080_0280_0000~0x8080_02FF_FFFF	0x8094_0000_0000~0x8094_007F_FFFF
	0x8080_0300_0000~0x8080_037F_FFFF	0x8098_0000_0000~0x8098_007F_FFFF
	0x8080_0380_0000~0x8080_03FF_FFFF	0x809C_0000_0000~0x809C_007F_FFFF
	0x8080_0400_0000~0x8080_047F_FFFF	0x80A0_0000_0000~0x80A0_007F_FFFF
	0x8080_0480_0000~0x8080_04FF_FFFF	0x80A4_0000_0000~0x80A4_007F_FFFF
	0x8080_0500_0000~0x8080_057F_FFFF	0x80A8_0000_0000~0x80A8_007F_FFFF
	0x8080_0580_0000~0x8080_05FF_FFFF	0x80AC_0000_0000~0x80AC_007F_FFFF
	0x8080_0600_0000~0x8080_067F_FFFF	0x80B0_0000_0000~0x80B0_007F_FFFF
	0x8080_0680_0000~0x8080_06FF_FFFF	0x80B4_0000_0000~0x80B4_007F_FFFF

	0x8080_0700_0000~0x8080_077F_FFFF	0x80B8_0000_0000~0x80B8_007F_FFFF
	0x8080_0780_0000~0x8080_07FF_FFFF	0x80BC_0000_0000~0x80BC_007F_FFFF
……	……	……
……	……	……
Chip255	0xFF80_0000_0000~0xFF80_007F_FFFF	0xFF80_0000_0000~0xFF80_007F_FFFF
	0xFF80_0080_0000~0xFF80_00FF_FFFF	0xFF84_0000_0000~0xFF84_007F_FFFF
	0xFF80_0100_0000~0xFF80_017F_FFFF	0xFF88_0000_0000~0xFF88_007F_FFFF
	0xFF80_0180_0000~0xFF80_01FF_FFFF	0xFF8C_0000_0000~0xFF8C_007F_FFFF
	0xFF80_0200_0000~0xFF80_027F_FFFF	0xFF90_0000_0000~0xFF90_007F_FFFF
	0xFF80_0280_0000~0xFF80_02FF_FFFF	0xFF94_0000_0000~0xFF94_007F_FFFF
	0xFF80_0300_0000~0xFF80_037F_FFFF	0xFF98_0000_0000~0xFF98_007F_FFFF
	0xFF80_0380_0000~0xFF80_03FF_FFFF	0xFF9C_0000_0000~0xFF9C_007F_FFFF
	0xFF80_0400_0000~0xFF80_047F_FFFF	0xFFA0_0000_0000~0xFFA0_007F_FFFF
	0xFF80_0480_0000~0xFF80_04FF_FFFF	0xFFA4_0000_0000~0xFFA4_007F_FFFF
	0xFF80_0500_0000~0xFF80_057F_FFFF	0xFFA8_0000_0000~0xFFA8_007F_FFFF
	0xFF80_0580_0000~0xFF80_05FF_FFFF	0xFFAC_0000_0000~0xFFAC_007F_FFFF
	0xFF80_0600_0000~0xFF80_067F_FFFF	0xFFB0_0000_0000~0xFFB0_007F_FFFF
	0xFF80_0680_0000~0xFF80_06FF_FFFF	0xFFB4_0000_0000~0xFFB4_007F_FFFF
	0xFF80_0700_0000~0xFF80_077F_FFFF	0xFFB8_0000_0000~0xFFB8_007F_FFFF
	0xFF80_0780_0000~0xFF80_07FF_FFFF	0xFFBC_0000_0000~0xFFBC_007F_FFFF

从上表可以看出，对L2B地址空间，只要把LA[26:23]的值移动到[37:34]，然后将[26:23]填充为0就可以得到PA的信息。

DRAM地址空间：

（**Note**: LA的[37:36]，以及PA的[33:32]都是reserved的，值固定为全0）

Chip ID	LA DRAM	PA DRAN

	0x8040_0000_0000~0x8040_FFFF_FFFF	0x8040_0000_0000~0x8040_FFFF_FFFF
	0x8041_0000_0000~0x8041_FFFF_FFFF	0x8044_0000_0000~0x8044_FFFF_FFFF
	0x8042_0000_0000~0x8042_FFFF_FFFF	0x8048_0000_0000~0x8048_FFFF_FFFF
	0x8043_0000_0000~0x8043_FFFF_FFFF	0x804C_0000_0000~0x804C_FFFF_FFFF
	0x8044_0000_0000~0x8044_FFFF_FFFF	0x8050_0000_0000~0x8050_FFFF_FFFF
	0x8045_0000_0000~0x8045_FFFF_FFFF	0x8054_0000_0000~0x8054_FFFF_FFFF
	0x8046_0000_0000~0x8046_FFFF_FFFF	0x8058_0000_0000~0x8058_FFFF_FFFF
Chip0	0x8047_0000_0000~0x8047_FFFF_FFFF	0x805C_0000_0000~0x805C_FFFF_FFFF
	0x8048_0000_0000~0x8048_FFFF_FFFF	0x8060_0000_0000~0x8060_FFFF_FFFF
	0x8049_0000_0000~0x8049_FFFF_FFFF	0x8064_0000_0000~0x8064_FFFF_FFFF
	0x804A_0000_0000~0x804A_FFFF_FFFF	0x8068_0000_0000~0x8068_FFFF_FFFF
	0x804B_0000_0000~0x804B_FFFF_FFFF	0x806C_0000_0000~0x806C_FFFF_FFFF
	0x804C_0000_0000~0x804C_FFFF_FFFF	0x8070_0000_0000~0x8070_FFFF_FFFF
	0x804D_0000_0000~0x804D_FFFF_FFFF	0x8074_0000_0000~0x8074_FFFF_FFFF
	0x804E_0000_0000~0x804E_FFFF_FFFF	0x8078_0000_0000~0x8078_FFFF_FFFF
	0x804F_0000_0000~0x804F_FFFF_FFFF	0x807C_0000_0000~0x807C_FFFF_FFFF
Chip1	0x80C0_0000_0000~0x80C0_FFFF_FFFF	0x80C0_0000_0000~0x80C0_FFFF_FFFF
	0x80C1_0000_0000~0x80C1_FFFF_FFFF	0x80C4_0000_0000~0x80C4_FFFF_FFFF
	0x80C2_0000_0000~0x80C2_FFFF_FFFF	0x80C8_0000_0000~0x80C8_FFFF_FFFF
	0x80C3_0000_0000~0x80C3_FFFF_FFFF	0x80CC_0000_0000~0x80CC_FFFF_FFFF
	0x80C4_0000_0000~0x80C4_FFFF_FFFF	0x80D0_0000_0000~0x80D0_FFFF_FFFF
	0x80C5_0000_0000~0x80C5_FFFF_FFFF	0x80D4_0000_0000~0x80D4_FFFF_FFFF
	C0x80C6_0000_0000~0x80C6_FFFF_FFFF	0x80D8_0000_0000~0x80D8_FFFF_FFFF
	0x80C7_0000_0000~0x80C7_FFFF_FFFF	0x80DC_0000_0000~0x80DC_FFFF_FFFF
	0x80C8_0000_0000~0x80C8_FFFF_FFFF	0x80E0_0000_0000~0x80E0_FFFF_FFFF
	0x80C9_0000_0000~0x80C9_FFFF_FFFF	0x80E4_0000_0000~0x80E4_FFFF_FFFF

	0x80CA_0000_0000~0x80CA_FFFF_FFFF	0x80E8_0000_0000~0x80E8_FFFF_FFFF
	0x80CB_0000_0000~0x80CB_FFFF_FFFF	0x80EC_0000_0000~0x80EC_FFFF_FFFF
	0x80CC_0000_0000~0x80CC_FFFF_FFFF	0x80F0_0000_0000~0x80F0_FFFF_FFFF
	0x80CD_0000_0000~0x80CD_FFFF_FFFF	0x80F4_0000_0000~0x80F4_FFFF_FFFF
	0x80CE_0000_0000~0x80CE_FFFF_FFFF	0x80F8_0000_0000~0x80F8_FFFF_FFFF
	0x80CF_0000_0000~0x80CF_FFFF_FFFF	0x80FC_0000_0000~0x80FC_FFFF_FFFF
……	……	……
……	……	……
Chip255	0xFFC0_0000_0000~0xFFC0_FFFF_FFFF	0xFFC0_0000_0000~0xFFC0_FFFF_FFFF
	0xFFC1_0000_0000~0xFFC1_FFFF_FFFF	0xFFC4_0000_0000~0xFFC4_FFFF_FFFF
	0xFFC2_0000_0000~0xFFC2_FFFF_FFFF	0xFFC8_0000_0000~0xFFC8_FFFF_FFFF
	0xFFC3_0000_0000~0xFFC3_FFFF_FFFF	0xFFCC_0000_0000~0xFFCC_FFFF_FFFF
	0xFFC4_0000_0000~0xFFC4_FFFF_FFFF	0xFFD0_0000_0000~0xFFD0_FFFF_FFFF
	0xFFC5_0000_0000~0xFFC5_FFFF_FFFF	0xFFD4_0000_0000~0xFFD4_FFFF_FFFF
	0xFFC6_0000_0000~0xFFC6_FFFF_FFFF	0xFFD8_0000_0000~0xFFD8_FFFF_FFFF
	0xFFC7_0000_0000~0xFFC7_FFFF_FFFF	0xFFDC_0000_0000~0xFFDC_FFFF_FFFF
	0xFFC8_0000_0000~0xFFC8_FFFF_FFFF	0xFFE0_0000_0000~0xFFE0_FFFF_FFFF
	0xFFC9_0000_0000~0xFFC9_FFFF_FFFF	0xFFE4_0000_0000~0xFFE4_FFFF_FFFF
	0xFFCA_0000_0000~0xFFCA_FFFF_FFFF	0xFFE8_0000_0000~0xFFE8_FFFF_FFFF
	0xFFCB_0000_0000~0xFFCB_FFFF_FFFF	0xFFEC_0000_0000~0xFFEC_FFFF_FFFF
	0xFFCC_0000_0000~0xFFCC_FFFF_FFFF	0xFFF0_0000_0000~0xFFF0_FFFF_FFFF
	0xFFCD_0000_0000~0xFFCD_FFFF_FFFF	0xFFF4_0000_0000~0xFFF4_FFFF_FFFF
	0xFFCE_0000_0000~0xFFCE_FFFF_FFFF	0xFFF8_0000_0000~0xFFF8_FFFF_FFFF
	0xFFCF_0000_0000~0xFFCF_FFFF_FFFF	0xFFFC_0000_0000~0xFFFC_FFFF_FFFF

从上表可以看出，对DRAM L2B 地址空间，只要把LA[35:32]的值移动到[37:34]，然后将[33:32]填充为0就可以得到PA的信息。

这部分的地址信息转换由硬件通过地址域段的移动就可以实现，不需要软件或者编译器的参与。

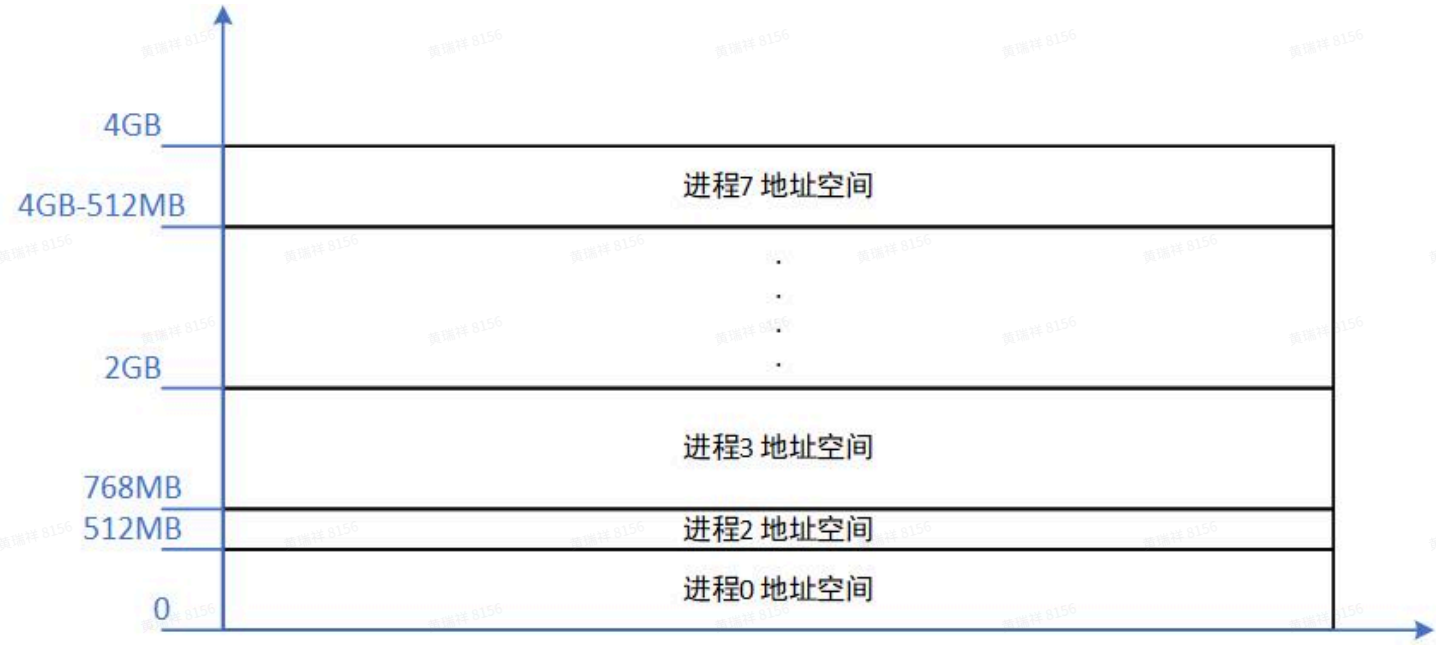
4.3 LA --> PA : Multi-Process DRAM Address Mapping (多进程DRAM地址映射)

在SiPU芯片项目中，它支持多进程模式下DRAM地址的映射转换功能。

该功能主要是为了支持多进程并行执行模式时，不同进程之间数据的隔离。从”Address Information Format“章节我们可以了解到，在SiPU中，地址信息可大致分为两部分：存储单元定位和单个存储单元内寻址。存储单元的定位是通过Chip ID, PE Cluster ID, PE ID这些信息来实现的；单个存储单元内寻址则是通过低位的internal address来实现的。本章节介绍的Multi-Process DRAM Address Mapping，针对的就是地址信息中单个存储单元内的寻址。

在SiPU芯片项目中，不同进程的数据可以并存在DRAM存储单元中，对更低一级的L2B存储单元，其只会存储同一进程的数据，同一时间不会有多个进程的数据都有效。

每个PE Cluster下都有一个4GB大小的DRAM，这个DRAM可以被多个进程占用，如图所示：



如上，DRAM 4GB空间被分给8个不同的进程，每个进程都占用一段连续的地址空间，如进程0占用 [0, 512MB) 空间，进程2占用 [512MB, 768MB) 空间，进程3占用 [768MB， 2GB）空间等。每个进程各自占用一段独立连续的地址空间，相互之间隔离开来。

在SiPU中，我们的目的主要是实现不同进程间数据的相互隔离，不需要其他复杂的功能，所以不需要复杂的页表，TLB等功能。但不同的实现方式还是有所不同。下面就不同的实现方案进行一些说明。

4.3.1 DRAM Address映射的配置方案选择

对DRAM Address映射的配置，主要是给每个进程指定一段可用的DRAM地址空间，并且该地址空间在进程生命周期内只给该进程使用，其他进程不能使用该空间。

下表是按照最多支持8个进程并行，每个进程都可以看到全部16个Cluster的方式列出的配置关系，每个进程对每个Cluster都有两个参数：Start_Addr和End_Addr，表示该进程在该Cluster的DRAM内可用地址空间的起始地址和结束地址。每个参数都是32位，表示的最小单位是1KB。

进程ID	Cluster ID	Start Addr (32bit，单位1KB)	End Addr (32bit，单位1KB)
进程 0	Cluster 0	Pg0_Clus0_Start_Addr	Pg0_Clus0_End_Addr
	Cluster 1	Pg0_Clus1_Start_Addr	Pg0_Clus1_End_Addr
	Cluster 2	Pg0_Clus2_Start_Addr	Pg0_Clus2_End_Addr
	Cluster 3	Pg0_Clus3_Start_Addr	Pg0_Clus3_End_Addr
	Cluster 4	Pg0_Clus4_Start_Addr	Pg0_Clus4_End_Addr
	Cluster 5	Pg0_Clus5_Start_Addr	Pg0_Clus5_End_Addr
	Cluster 6	Pg0_Clus6_Start_Addr	Pg0_Clus6_End_Addr
	Cluster 7	Pg0_Clus7_Start_Addr	Pg0_Clus7_End_Addr
	Cluster 8	Pg0_Clus8_Start_Addr	Pg0_Clus8_End_Addr
	Cluster 9	Pg0_Clus9_Start_Addr	Pg0_Clus9_End_Addr
	Cluster 10	Pg0_Clus10_Start_Addr	Pg0_Clus10_End_Addr
	Cluster 11	Pg0_Clus11_Start_Addr	Pg0_Clus11_End_Addr
	Cluster 12	Pg0_Clus12_Start_Addr	Pg0_Clus12_End_Addr
	Cluster 13	Pg0_Clus13_Start_Addr	Pg0_Clus13_End_Addr
	Cluster 14	Pg0_Clus14_Start_Addr	Pg0_Clus14_End_Addr
	Cluster 15	Pg0_Clus15_Start_Addr	Pg0_Clus15_End_Addr
进程 1	Cluster 0	Pg1_Clus0_Start_Addr	Pg1_Clus0_End_Addr
	Cluster 1	Pg1_Clus1_Start_Addr	Pg1_Clus1_End_Addr
	Cluster 2	Pg1_Clus2_Start_Addr	Pg1_Clus2_End_Addr
	Cluster 3	Pg1_Clus3_Start_Addr	Pg1_Clus3_End_Addr
	Cluster 4	Pg1_Clus4_Start_Addr	Pg1_Clus4_End_Addr

	Cluster 5	Pg1_Clus5_Start_Addr	Pg1_Clus5_End_Addr
	Cluster 6	Pg1_Clus6_Start_Addr	Pg1_Clus6_End_Addr
	Cluster 7	Pg1_Clus7_Start_Addr	Pg1_Clus7_End_Addr
	Cluster 8	Pg1_Clus8_Start_Addr	Pg1_Clus8_End_Addr
	Cluster 9	Pg1_Clus9_Start_Addr	Pg1_Clus9_End_Addr
	Cluster 10	Pg1_Clus10_Start_Addr	Pg1_Clus10_End_Addr
	Cluster 11	Pg1_Clus11_Start_Addr	Pg1_Clus11_End_Addr
	Cluster 12	Pg1_Clus12_Start_Addr	Pg1_Clus12_End_Addr
	Cluster 13	Pg1_Clus13_Start_Addr	Pg1_Clus13_End_Addr
	Cluster 14	Pg1_Clus14_Start_Addr	Pg1_Clus14_End_Addr
	Cluster 15	Pg1_Clus15_Start_Addr	Pg1_Clus15_End_Addr
.....	.....	.....	.....
.....	.....	.....	.....
进程 7	Cluster 0	Pg7_Clus0_Start_Addr	Pg7_Clus0_End_Addr
	Cluster 1	Pg7_Clus1_Start_Addr	Pg7_Clus1_End_Addr
	Cluster 2	Pg7_Clus2_Start_Addr	Pg7_Clus2_End_Addr
	Cluster 3	Pg7_Clus3_Start_Addr	Pg7_Clus3_End_Addr
	Cluster 4	Pg7_Clus4_Start_Addr	Pg7_Clus4_End_Addr
	Cluster 5	Pg7_Clus5_Start_Addr	Pg7_Clus5_End_Addr
	Cluster 6	Pg7_Clus6_Start_Addr	Pg7_Clus6_End_Addr
	Cluster 7	Pg7_Clus7_Start_Addr	Pg7_Clus7_End_Addr
	Cluster 8	Pg7_Clus8_Start_Addr	Pg7_Clus8_End_Addr
	Cluster 9	Pg7_Clus9_Start_Addr	Pg7_Clus9_End_Addr
	Cluster 10	Pg7_Clus10_Start_Addr	Pg7_Clus10_End_Addr
	Cluster 11	Pg7_Clus11_Start_Addr	Pg7_Clus11_End_Addr
	Cluster 12	Pg7_Clus12_Start_Addr	Pg7_Clus12_End_Addr

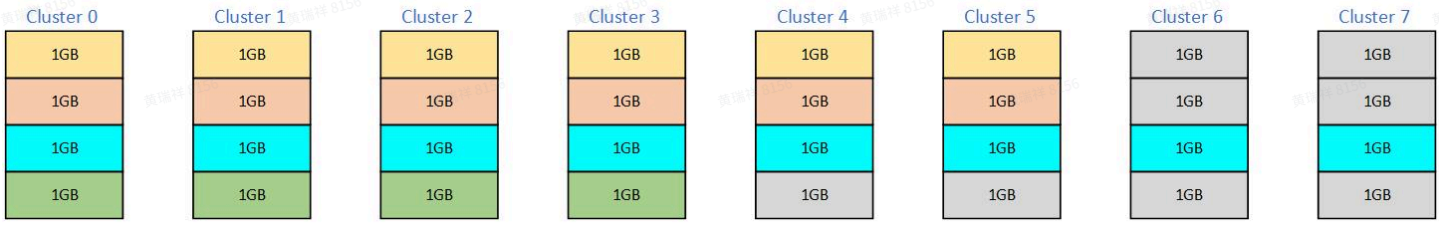
	Cluster 13	Pg7_Clus13_Start_Addr	Pg7_Clus13_End_Addr
	Cluster 14	Pg7_Clus14_Start_Addr	Pg7_Clus14_End_Addr
	Cluster 15	Pg7_Clus15_Start_Addr	Pg7_Clus15_End_Addr

单个进程，各个Cluster的地址空间单独配置（在每个Cluster的DRAM中的地址空间可以不同），还是所有Cluster统一配置（在每个Cluster的DRAM中地址空间相同）？

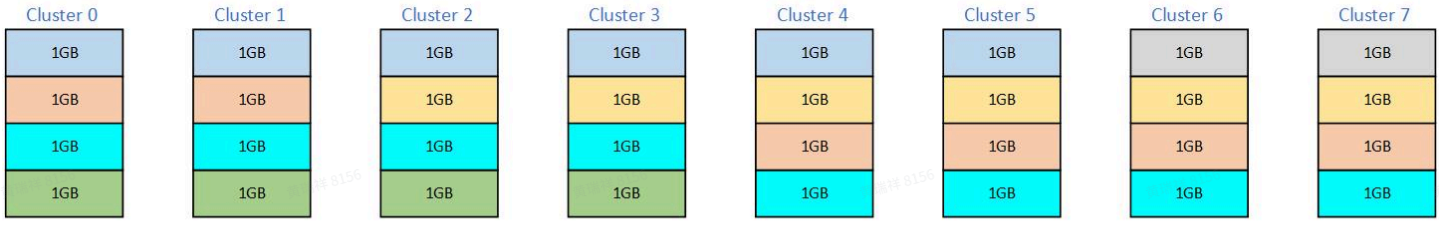
如果单个进程所有的Cluster统一配置，可能会造成DRAM地址空间的浪费。举例：以最多有8个Clusters为例，进程0（绿色块）需要4个Clusters，每个Cluster的DRAM中占用1GB空间；进程1（天蓝色块）需要8个Clusters，每个Cluster的DRAM中占用1GB空间；进程2（橙色块）需要6个Clusters，每个Cluster的DRAM中占用1GB空间；进程3（金色块）需要6个Clusters，每个Cluster的DRAM中占用1GB空间；进程4（蓝色块）需要6个Clusters，每个Cluster的DRAM中占用1GB空间。

统一配置方式时，进程0分配了Cluster0/1/2/3，进程1分配了Cluster0/1/2/3/4/5/6/7，进程2分配了Cluster0/1/2/3/4/5，进程3分配了Cluster0/1/2/3/4/5。因为是统一配置方式，到进程4时，已经没有足够的空间可以分配给进程4了。

独立配置方式时，进程0分配了Cluster0/1/2/3；进程1分配了Cluster0/1/2/3/4/5/6/7，但在Cluster4/5/6/7中占用的是最低的1GB空间；进程2分配了Cluster0/1/4/5/6/7，各自占用的地址空间并不相同；进程3分配了Cluster2/3/4/5/6/7；此时，所有Cluster的最高1GB空间还是空闲的，进程4可以被分配，并且使用了Cluster0/1/2/3/4/5。



每个进程的Cluster统一配置



每个进程的Cluster独立配置

从性能角度考虑，建议采用各个Cluster地址空间单独配置的方式。

根据DRAM的地址映射关系配置信息所在的位置，可以有两种实现方式：方案A）地址映射关系存放在DRAM中；方案B）地址映射关系存放在配置寄存器中；

4.3.1.1 方案A： DRAM地址映射关系存放在DRAM中

该方案中，DRAM地址映射关系表存放在DRAM中。

单个Chip中，DRAM是分布式的存储结构，在16个Clusters中各有一份DRAM存储单元，大小是4GB空间。需要在每个Cluster的DRAM中都存放一份这个地址映射关系表。

单个进程的一个Cluster的配置参数是2个32位的地址信息，即Start_Addr和End_Addr，需要占用8B空间，全部16个Cluster的配置参数共128B大小。如果最多支持8个进程并行，则需要1KB空间~~，如果最多支持16个进程并行，则需要2KB空间~~。

可以约定在每个Cluster的DRAM中，最低1KB空间预留给这个DRAM地址映射关系表。

这个DRAM地址映射关系表需要由系统来维护，上电后，系统需要为各个进程分配好其在每个Cluster中DRAM的地址空间，并把这个映射关系表写入到每个DRAM的最低1KB空间中去。如果不是上电后一次性配置好，也可以在每次启动新的进程前来配置这个映射关系表，但需要注意在配置新的进程时，不能破坏掉之前已经配置好的，但还没有执行完的进程的映射关系。

这个配置过程，在硬件上，对应的就是GCS中的DMA功能模块进行从Host到各个Cluster中DRAM的数据搬移操作。

在进程执行过程中，各个PE模块执行各自的程序流，当出现需要访存DRAM的指令时，PE中的执行单元需要先去本Cluster的DRAM的最低1KB空间读取这个地址映射关系表，然后根据目的Cluster ID选择对应的entry，获取其映射关系，然后计算出需要访问的真实地址。这个计算过程如下：

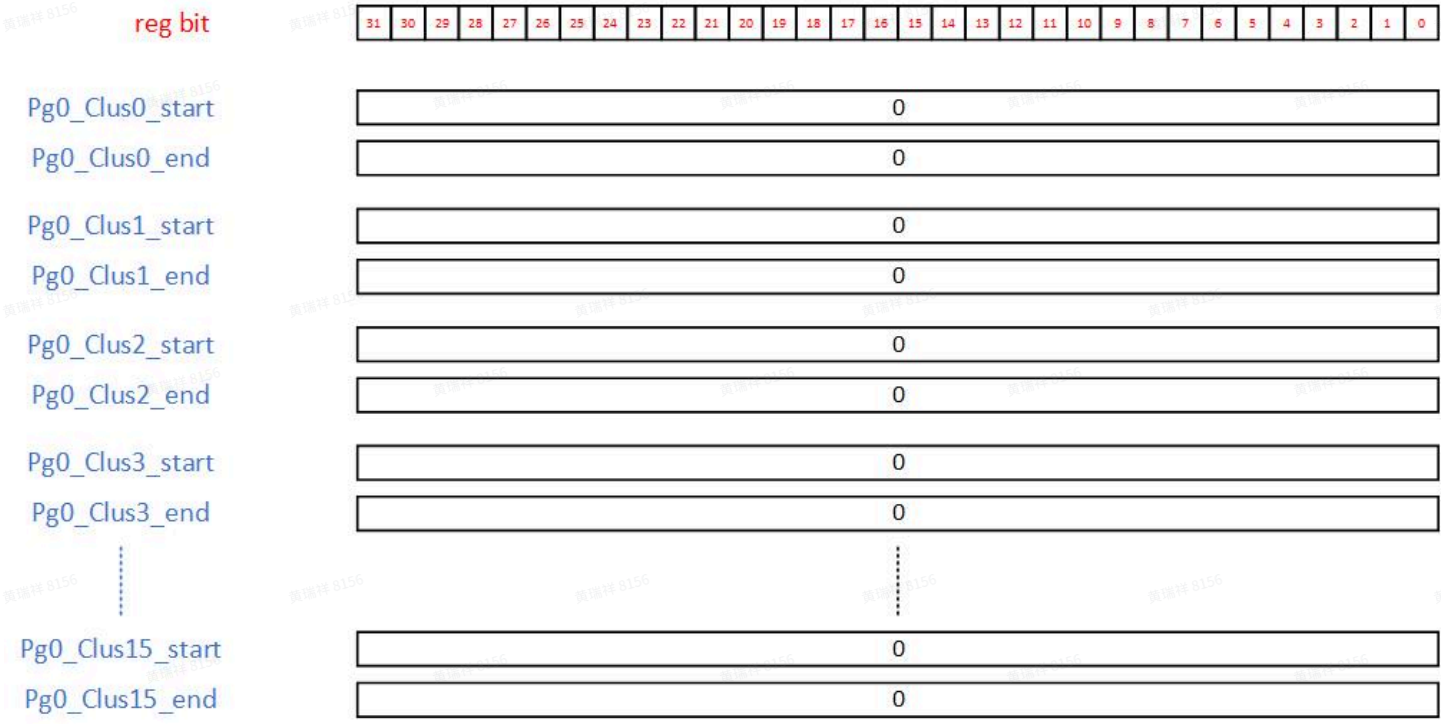
目的真实地址 = 程序指令中的地址 + 地址映射关系表中选中的entry中的Start_Addr参数；

Note：访存的操作在GCS的DMA中也会存在，类似于PE模块中的执行单元，GCS的DMA模块在进行数据搬移时，也需要先去选中的Cluster的DRAM的最低1KB空间中读取地址映射关系，然后根据目的Cluster选择对应的entry，获取其映射关系，然后计算出修访问的真实地址，计算方式同PE模块的执行单元。

4.3.1.2 方案B： DRAM地址映射关系存在于配置寄存器中

该方案中，DRAM地址映射关系表存放于配置寄存器中。

单个进程的一个Cluster的配置参数需要2个32位配置寄存器，全部16个Clusters 共需要32个配置寄存器。如果最多支持8个进程并行，则需要256个配置寄存器~~，如果最多支持16个进程并行，则需要512个配置寄存器~~。



这些配置寄存器需要由系统进行配置。在每次启动新的进程前，系统需要把对应进程的配置寄存器配置好。

配置寄存器信息如何传递给各个PE模块？

目前有以下几种方案。

方案B-1：硬连线广播方式

该方案中，配置寄存器存在于GCS模块中，使用硬连线的方式广播给所有的PE模块。

如果最多8进程并行，这组寄存器信息是256x32=8192根线~~，如果最多是16进程并行，这组寄存器信息是16384根线~~。

这种方式对后端的布局布线很不友好。

方案B-2：随路TBC命令下发方式

该方案中，配置寄存器存在于GCS模块中，当GCS下发TBC命令时，随路携带这些寄存器信息发送给对应的PE模块，PE模块保存并使用该信息。

GCS模块在下发TBC命令时，需要计算其需要下发到哪个Cluster，此时可以根据进程ID信息将该进程的DRAM地址映射关系选择出来，跟随TBC命令一起下发下去。

如果支持一个进程的所有Cluster单独配置，那么一个进程就有16x2个32寄存器，也就是1024位，128B信息，这些信息需要多个cycle才能够发送完成。增加了TBC命令下发的延迟。

方案B-3： 配置命令广播方式

该方案中，配置寄存器存在于GCS模块中，当DRAM地址映射关系配置寄存器配置完成后，由GCS主动发起配置信息下发命令，将这些配置信息发送给对应的PE模块，由PE模块中的RV-core负责锁存这些配置信息。

GCS模块可以精准控制向哪些PE模块下发配置信息命令，即它知道每个进程占用了哪几个物理Cluster，只向该进程占用的Cluster下发配置信息命令；也可以简化设计，向所有的PE模块下发所有进程的配置信息。

相比于方案B-2，该方案不需要对每个TBC命令都增加配置信息的发送，只需要在每个进程确定后，下发一次配置信息即可，减少了单次TBC命令下发的延时。

方案B-4：每个PE模块有自己的配置寄存器空间，每个PE模块都单独配置；

该方案中，不只GCS有DRAM地址映射关系配置寄存器，各PE模块都有自己的DRAM地址映射关系配置寄存器。系统对每个需要进行地址转换的模块都进行配置寄存器配置。

4.3.1.3 方案选择

几种方案比较如下：

方案A： DRAM 地址映射关系存放在DRAM中

映射关系表存放到DRAM中的固定地址，每个Cluster中的DRAM存一份，由系统完成这个关系表的搬移。

优点：不占用配置寄存器空间；

缺点：1）每个Cluster需要拿出部分空间来存放这个映射关系；

2）PE中的执行单元在处理访存指令时，需要首先去DRAM中读取这个映射关系，增加指令执行的延时；

方案B： DRAM 地址映射关系存放在配置寄存器中

映射关系存放在配置寄存器中。

优点：执行单元可以快速完成访存指令地址的映射；

缺点：占用配置寄存器空间；

方案B-1）广播的方式：因为一个chip内所有64个PE（还有C2C模块，Media模块）都需要这个配置寄存器信息，如果采用广播硬连线的方式，对后端设计很不友好；

方案B-2）跟随TBC命令随路发送方式：如果一个进程所有的Cluster都使用相同的配置方式，随路发送的方式很方便，但所有的Cluster都使用相同的映射关系，可能会造成DRAM地址空间的浪费；如果一个进程所有的Cluster都有自己的配置方式，那么每个进程都需要16组寄存器配置，会增加TBD命令下

发的时间；另外这种方式只能对PE模块进行配置，C2C模块和Media模块还是没有办法实现配置功能；

方案B-3）配置命令广播方式：这种方式需要GCS模块额外进行配置命令的发送，而且是硬件自定义的一种命令格式；另外这种方式只能对PE模块进行配置，C2C模块和Media模块还是没有办法实现配置功能；

方案B-4) 每个模块单独可配置方式：这种方式中，每个需要进行地址转换的模块都需要有自己的配置寄存器空间，但可以实现对所有需要进行地址转换的模块的单独配置；

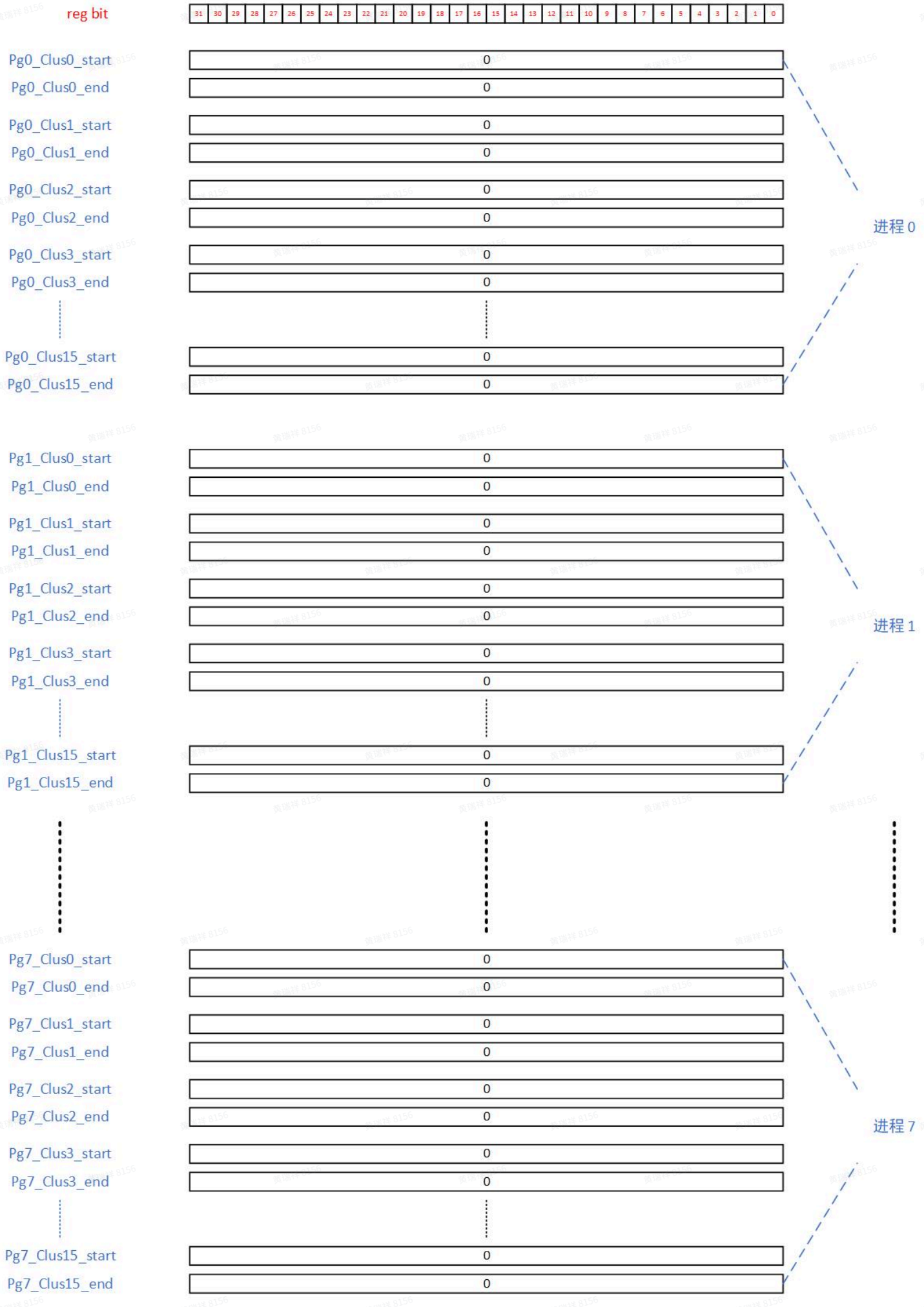
综合上述，建议选择 **方案B-4** 方式。

4.3.2 DRAM Address映射的配置

4.3.2.1 DRAM地址映射关系配置寄存器格式

每个进程每个Cluster都需要2个配置寄存器：start_addr和end_addr。所有16个Cluster共需16x2=32个配置寄存器，最多8个进程，共需8x16x2=256个配置寄存器。

寄存器格式如下：


reg bit
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Pg0_Clus0_start
Pg0_Clus0_end
Pg0_Clus1_start
Pg0_Clus1_end
Pg0_Clus2_start
Pg0_Clus2_end
Pg0_Clus3_start
Pg0_Clus3_end
Pg0_Clus15_start
Pg0_Clus15_end
0
进程 0
Pg1_Clus0_start
Pg1_Clus0_end
Pg1_Clus1_start
Pg1_Clus1_end
Pg1_Clus2_start
Pg1_Clus2_end
Pg1_Clus3_start
Pg1_Clus3_end
Pg1_Clus15_start
Pg1_Clus15_end
进程 1
Pg7_Clus0_start
Pg7_Clus0_end
Pg7_Clus1_start
Pg7_Clus1_end
Pg7_Clus2_start
Pg7_Clus2_end
Pg7_Clus3_start
Pg7_Clus3_end
Pg7_Clus15_start
Pg7_Clus15_end
进程 7

4.3.2.2 GCS模块

在GCS模块中，在执行DMA类型packet命令时，如果需要访问DRAM，需要进行DRAM地址映射关系转换，所以GCS模块需要这个DRAM地址映射信息。

GCS模块有自己的DRAM地址映射关系配置寄存器空间（8x16x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

GCS模块的DRAM地址映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.3.2.1章节。

4.3.2.3 PE模块

在PE模块中，无论是RV-core的LSU操作，还是Tile-core的LSU操作或者数据搬移操作，在访问DRAM时，都需要进行DRAM地址映射关系转换，所以PE模块需要这个DRAM地址映射信息。

PE模块有自己的DRAM地址映射关系配置寄存器空间（8x16x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

PE模块的DRAM地址映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.3.2.1章节。

单个Chip内有64个PE模块，共64x8x16x2个32位配置寄存器。

4.3.2.4 C2C模块

在C2C模块中，无论是Host发起的packet命令，还是程序指令流产生的跨Chip数据搬移指令，执行数据搬移操作时，对本Chip内的DRAM存储单元进行访问时，都需要进行DRAM地址映射关系转换，所以C2C模块需要这个DRAM地址映射信息。

C2C模块有自己的DRAM地址映射关系配置寄存器空间（8x16x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

C2C模块的DRAM地址映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.3.2.1章节。

4.3.2.5 Media模块

在Media模块中，当执行数据搬移操作时，如果对DRAM进行访问，需要进行DRAM地址映射关系转换，所以Media模块需要这个DRAM地址映射信息。

Media模块有自己的DRAM地址映射关系配置寄存器空间（8x16x2个32位配置寄存器），在进程启动前，系统需要把相应的配置寄存器配置好。

Media模块的DRAM地址映射关系配置寄存器有8组，一个进程对应一组，寄存器格式见4.3.2.1章节。

4.4 Address Translation Implementation 地址转换实现

4.4.1 GCS module

在GCS模块中，和Memory系统的地址转换相关的工作有三个方面：

1）各种映射关系的配置工作；

2）GCS中TBC命令调度时的地址映射工作；

3）GCS中DMA执行数据搬移功能时的地址映射工作；

下面详细介绍各工作的具体操作。

4.4.1.1 *GCS模块中各种映射关系的配置工作*

为了支持地址转换功能，需要配置如下信息： CLuster ID映射关系，DRAM地址映射关系。

对每一个进程，通过Cluster ID映射关系可以知道其占用的物理Cluster信息，而通过DRAM地址映射关系可以知道其在每个Cluster的DRAM中占用的地址空间信息。

Cluster ID映射关系是通过配置寄存器来实现的，系统会产生寄存器配置命令，GCS模块需要接收APB总线传送来的配置信息，并把Cluster ID映射关系锁存下来。

DRAM地址映射关系的配置有两种方案：

~~**方案A**：DRAM地址映射关系配置到DRAM中。~~

~~系统会安排好各个进程的各个Cluster的DRAM地址映射关系，之后系统可以通过一个DMA类型的packet命令通知GCS，让其将这个映射关系数据从Host搬移到各个Cluster的DRAM的最低1KB空间去。~~

~~GCS模块执行该DMA packet命令，首先从Host中读取到这些地址映射关系的数据，然后依次传送到每个Cluster的DRAM中去。在这个过程中，GCS不需要去查询Cluster ID映射表去进行逻辑Cluster ID到物理Cluster ID的转换，它只需要依次对每个物理Cluster都进行数据搬移即可~~。（不采用的方案）

方案B：DRAM地址映射关系配置到配置寄存器中。

系统会安排好各个进程的各个Cluster的DRAM地址映射关系，之后系统会产生寄存器配置命令，GCS模块需要接收APB总线传送来的配置信息，并把DRAM地址映射关系锁存下来。

在4.3.1.2章节提到了PE模块的不同配置方式，在不同配置方式时，GCS模块的行为各不相同：

~~***“方案B-1”***：硬连线广播 - GCS模块已经锁存了DRAM地址映射关系信息，其以硬连线的方式把这些信息输出给不同模块，不需要其他的操作；~~

~~***“方案B-2”***：随路TBC命令下发 - GCS模块在为TBC命令选择好PE模块后，需要把Cluster ID映射关系（64bit=8B）和DRAM地址映射关系（单个进程 16x64bit=128B），以及进程ID信息打包好，附加在~~

~~TBC命令信息上，跟随TBC命令随路下发到PE模块的RV-core，由RV-core接收这条TBC命令时，同时接收子配置信息，并把解析出的映射关系锁存到对应进程ID的寄存器中；~~

~~***“方案B-3”***：以配置信息命令方式广播 - GCS锁存好进程的DRAM地址映射关系配置信息后，在启动Kernel packet命令拆分TBC并下发之前，先把这些配置信息以配置信息命令的方式广播给对应的PE模块。~~

~~GCS首先查询Cluster ID映射关系配置寄存器，根据进程ID选择出该进程的Cluster ID映射关系，从而获知该进程占用几个Cluster，以及每个CLuster的物理ID信息，然后GCS需要把Cluster ID映射关系（64bit=8B）和DRAM地址映射关系（单个进程 16x64bit=128B），以及进程ID信息打包好，并发送给该进程所有占用的物理Cluster中所有的PE模块的RV-core，由RV-core接收这条配置信息命令，并把解析出的映射关系锁存到对应进程ID的寄存器中~~。（不采用的方案）

“方案B-4”：各模块有自己的配置寄存器空间 - 在此方案下，GCS模块只需要锁存好自己的DRAM地址映射关系配置寄存器信息就可以，其他模块的配置不需要它负责；

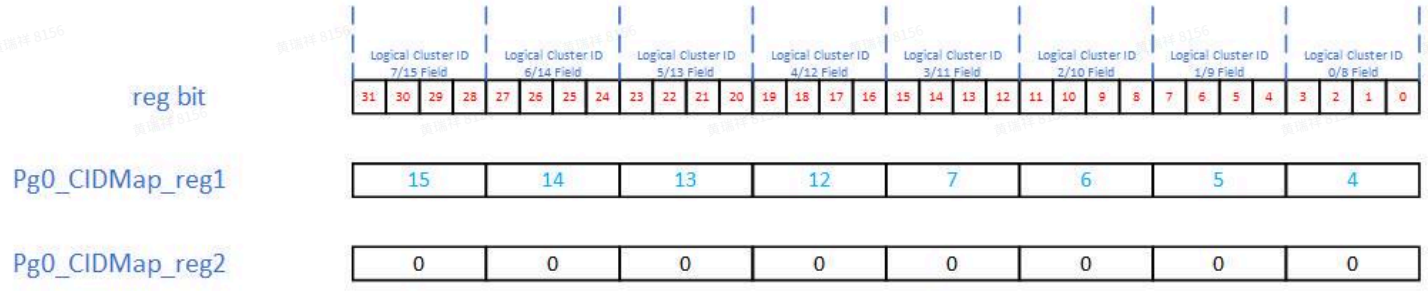
由4.3.1.3章节可知，我们选择方案B-4，GCS只需要维护自己的地址转换配置信息相关的配置寄存器就可以了。

4.4.1.2 GCS中TBC命令调度时的地址映射工作

对于Kernel类型的packet命令，GCS需要将其拆分成TBC命令，然后下发到PE模块去执行。

packet命令中会携带信息，指定TBC使用哪个Cluster，这个Cluster ID信息是虚拟的ID信息，GCS需要去查询Cluster ID映射配置寄存器，根据进程ID获取该进程的从虚拟ID到物理ID的映射关系，然后再根据虚拟ID选择寄存器对应的域段，从而获取物理Cluster ID信息，之后才将该TBC命令下发。

借用之前的一个例子，举例说明。假定进程0占用8个Clusters。该进程的Cluster ID映射关系如下所示：



该进程中有个Kernel packet命令，它可以拆分出4个TBC命令，每个TBC依次指定使用的Cluster ID信息是0/1/2/3 。这个Cluster ID信息是虚拟信息。GCS在将这个Kernel的TBC下发时，需要通过查询进程0的Cluster ID映射关系寄存器来获知物理的Cluster ID信息。

经查询，第一个TBC命令，虽然指定Cluster ID是0，但实际它是被下发到物理Cluster4，而第二个TBC，指定的Cluster ID是1，但实际下发到物理Cluster5，依次，第三个TBC下发到物理Cluster6，第四个TBC下发到物理Cluster7 。

4.4.1.3 GCS中DMA数据搬移时的地址映射工作

对于DMA类型的packet命令，GCS的DMA执行单元需要完成数据的搬移功能。

对于数据搬移操作，命令会指定从哪个存储单元的什么地址，搬移多少数据到哪个存储单元的什么地址。为充分说明地址映射的工作，下面的描述是以从哪个Cluster的DRAM的某地址，搬移数据到另外一个Cluster的DRAM的某地址的搬移操作来说明。

例如，当前DMA操作是进程0的操作，需要把Cluster0 (虚拟ID)的DRAM的0~255MB地址空间的数据，搬移到Cluster1（虚拟ID）的DRAM的0~255MB地址空间去。即从命令角度看，源数据存在于Cluster0的DRAM的0~255MB空间内，目的地址是Cluster1的DRAM的0~255MB空间。

GCS首先需要查询Cluster ID映射表，从进程0的配置寄存器中获知，虚拟Cluster0对应物理Cluster0，而虚拟Cluster1对应物理Cluster3，这个搬移操作实际是从物理Cluster0搬移数据到物理Cluster3中去。

之后，GCS需要查询DRAM地址映射关系，来获知数据在DRAM中真正的地址，有两种方案：

~~**方案A**：DRAM地址映射关系信息在DRAM中~~

~~DRAM地址映射关系信息存在于每个Cluster的DRAM的最低1KB空间，逐进程逐Cluster存放。~~

~~当前进程号是0，涉及到的物理Cluster ID是0和3，所以地址映射关系就在DRAM的最低32B数据内（见下表）。GCS需要发起对物理Cluster0的DRAM低32B数据的读请求。等数据读取回来后，经解析，我们获知，进程0在物理Cluster0的DRAM中占用的地址空间是1GB~2GB，而在物理Cluster3的DRAM中占用的地址空间是3GB~4GB。~~

~~上述地址映射信息已经明确，即源数据存放在物理Cluster0的DRAM的 1GB ~ 1GB+255MB中，目的地址是物理Cluster3的DRAM的 3GB ~ 3GB+255MB。DMA开始执行数据搬移动作。它先发起对物理Cluster0的DRAM的读请求，读数的起始地址是1GB，读数大小是256MB；待收到数据后，再发起对物理Cluster3的DRAM的写请求，写请求的起始地址是3GB，全部写数据大小是256MB。~~

~~进程号~~	~~Cluster ID~~	~~地址应该关系在DRAM最低2KB中的位置偏移~~
~~进程 0~~	~~Clus 0~~	~~0~~
	~~Clus 1~~	~~8B~~
	~~Clus 2~~	~~16B~~
	~~Clus 3~~	~~24B~~
	~~Clus 4~~	~~32B~~
	~~……~~	~~……~~

	~~Clus 15~~	~~120B~~
~~进程 1~~	~~Clus 0~~	~~128B~~
	~~……~~	~~……~~
~~……~~	~~……~~	~~……~~

（不采用的方案）

方案B： DRAM地址映射关系信息在配置寄存器中

DRAM地址映射关系信息存在于GCS的配置寄存器中。

当前进程号是0，涉及到的物理Cluster ID是0和3，所以需要的配置寄存器是gcs_pg0_clus0_start_addr/gcs_pg0_clus0_end_addr 和gcs_pg0_clus3_start_addr/gcs_pg0_clus3_end_addr 。GCS从这两组寄存器中获知，进程0在物理Cluster0的DRAM中占用的地址空间是1GB~2GB，而在物理Cluster3的DRAM中占用的地址空间是3GB~4GB 。

后续的操作和方案A就相同了，源数据存放在物理Cluster0的DRAM的 1GB ~ 1GB+255MB中，目的地址是物理Cluster3的DRAM的 3GB ~ 3GB+255MB 。DMA开始执行数据搬移动作。它先发起对物理Cluster0的DRAM的读请求，读数的起始地址是1GB，读数大小是256MB； 待收到数据后，再发起对物理Cluster3的DRAM的写请求，写请求的起始地址是3GB，全部写数据大小是256MB 。

该章节的地址转换功能和其他模块中的地址转换功能很相似，可以做成通用IP--ATU，具体可见“SIPU 1.0 ATU架构设计方案”文档 SIPU 1.0 ATU架构设计方案v0.5。

4.4.2 PE module

在PE模块中，和Memory系统的地址转换相关的工作有两个方面：

1）各种映射关系的配置工作；

2）访存指令需要访问存储单元时的地址映射工作；

下面详细介绍各工作的具体操作。

4.4.2.1 PE模块中各种映射关系的配置工作

为了支持地址转换功能，需要配置如下信息： CLuster ID映射关系，DRAM地址映射关系。

Cluster ID映射关系的配置之前没有提过存放在DRAM中的方案，此处为了方便，把Cluster ID映射的配置和DRAM地址映射的配置一起说明。

对地址转换所需的映射关系的配置有两种方案：

~~**方案A**：地址转换映射关系配置到DRAM中。~~

~~系统为每个进程定义了地址转换映射关系后，通过一个DMA类型的packet命令，将这些映射关系数据从Host中搬移到各个Cluster的DRAM的指定地址空间去。该过程不需要PE模块的参与。~~（不采用的方案）

方案B：地址转换映射关系配置到配置寄存器中。

系统为每个进程定义了地址转换映射关系后，通过寄存器配置命令去配置相应的配置寄存器。

在4.3.1.2章节提到了PE模块的不同配置方式，在不同配置方式时，PE模块的行为各不相同：

~~***“方案B-1”***：硬连线广播 - 地址转换映射关系以硬连线的方式输送到PE模块的接口上，PE模块可直接使用；~~

~~***“方案B-2”***：随路TBC命令下发 - 地址转换映射关系跟随TBC命令随路下发到PE模块的RV-core，RV-core接收这条TBC命令时，同时接收了配置信息，它需要解析这些映射关系，并锁存到对应进程ID的寄存器中；~~

~~***“方案B-3”***：以配置信息命令方式广播 - 地址转换映射关系以专门的配置信息命令的方式发送给PE模块的RV-core，RV-core接收这个配置信息命令，它需要解析这些映射关系，并锁存到对应进程ID的寄存器中；~~（不采用的方案）

“方案B-4”：各模块有自己的配置寄存器空间 - 在此方案下，PE模块有自己的配置寄存器空间，PE模块需要接收寄存器配置命令，并把相应的配置寄存器信息锁存下来；

由4.3.1.3章节可知，我们选择方案B-4。

4.4.2.2 PE模块中访存时的地址映射工作

PE模块中，可能会执行访存指令的单元包括RV-core的LSU执行单元，Tile-core的TLSU执行单元和DTE执行单元。这些单元在执行访存指令时，指令中给出的Cluster ID是虚拟的ID信息，给出的存储单元的地址也是软件层面看到的地址，并不是实际的物理地址。这些指令的操作都需要进行地址转换。

我们知道，在PE模块现在的架构中，L2B存储单元就位于PE模块内部，访存指令要访问本PE模块的L2B存储单元和访问其他地方的存储单元走的是不同路线：本PE内部L2B的访问是PE-xBar总线，对外部其他存储单元的访问是经过MIF模块上到ICI总线。而地址转换时也是有两部分信息的转换：Cluster ID的映射和DRAM地址的映射。

无论是访问内部L2B还是访问外部的存储单元，Cluster ID映射这个地址转换都是需要执行的；而只有访问外部的DRAM的请求才需要进行DRAM地址映射转换。

PE模块的地址转换配置信息是整个PE模块一份的，这个地址转换功能也会是统一的由一个功能IP来实现，这是个通用IP--ATU，具体可见“SIPU 1.0 ATU架构设计方案”文档 SIPU 1.0 ATU架构设计方案v0.5。这个功能IP将放在MIF模块中，即经过MUX仲裁后选出的一路访存请求，在上传到ICI总线之前，先进行地址转换功能。

至于在PE内部，其判断是分支到PE-xBar，还是分支到MIF时需要进行的Cluster ID映射关系查询，可以有两种方案：

方案一：各执行单元发送Cluster ID映射关系查询请求给地址转换功能块ATU，由ATU完成查询，并把映射后的物理Cluster ID信息反馈给执行单元；

方案二：ATU把Cluster ID映射关系配置寄存器的配置信息广播给各个执行单元，由各执行单元自己查询；

如果采用方案一，在PE模块中最多有4路源头（2个RV-core，TLSU, DTE）可以发起Cluster ID映射关系查询请求，ATU至少需要留出4组这样的通信接口，ATU是通用IP，在其他模块中如果不需要这么多通信接口，多余的接口空置即可。

（**TBD:需确定采用方案一，还是方案二，和硬件设计Owner讨论**）

根据地址转换映射关系存放的地方不同，有两种实现方案，下面以具体的例子来进行说明：

例子一： RV-core的LSU执行单元执行Load指令。

当前是进程2在工作，一条load指令指明要从Cluster7(虚拟ID)的DRAM中的512MB偏移地址读取数据。

LSU执行单元首先查询Cluster ID映射寄存器，在这个TBC被下发到该RV-core之前，该寄存器已经被配置好，通过查询寄存器中虚拟Cluster7对应的位置，即寄存器的[31:28]域段，获知其物理Cluster是Cluster4 。

之后，LSU还需要去查询DRAM地址映射关系，这个查询有两种方案：

~~**例一方案A**： DRAM地址映射关系信息在DRAM中~~

~~DRAM地址映射关系信息存在于每个Cluster的DRAM的最低1KB空间，逐进程逐Cluster存放。~~

~~当前进程号是2，涉及到的物理Cluster ID是4，所以其地址映射关系存在于DRAM的最低1KB空间的288B位置。LSU执行单元需要首先读取该DRAM的地址，获知其DRAM地址映射关系。假定该关系指定进程2在Cluster4中的地址空间是1GB~2GB，则该指令真正的访问地址是 1GB+512MB 。~~

~~LSU执行单元发起对物理Cluster4的 1GB+512MB地址的读请求，然后把读回的数据传送给目的寄存器，该load指令执行完成。~~

~~进程号~~	~~Cluster ID~~	~~地址应该关系在DRAM最低2KB中的位置偏移~~
~~进程 0~~	~~Clus 0~~	~~0~~

	Clus 1	8B
	……	……
	Clus 15	120B
进程 1	Clus 0	128B
	Clus 1	136B
	……	……
	Clus 15	248B
进程 2	Clus 0	256B
	Clus 1	264B
	Clus 2	272B
	Clus 3	280B
	Clus 4	288B
	Clus 5	296B
	Clus 6	304B
	……	……
……	……	……

（不采用的方案）

例一方案B： DRAM地址映射关系信息在寄存器中

DRAM地址映射关系信息存在于PE模块的配置寄存器中。

当前进程号是2，涉及到的物理Cluster ID是4，所以需要的寄存器是 pg2_clus4_start_addr/pg2_clus4_end_addr 。从这组寄存器中获知，进程2在Cluster4中的地址空间是1GB~2GB，则该指令真正的访问地址是 1GB+512MB 。

LSU执行单元发起对物理Cluster4的 1GB+512MB地址的读请求，然后把读回的数据传送给目的寄存器，该load指令执行完成。

例子二： Tile-core的DTE执行单元进行不同Cluster之间的数据搬移。

当前DTE执行进程1的数据搬移指令，需要把Cluster2(虚拟ID)的DRAM的256MB~511MB地址空间的数据搬移到Cluster3(虚拟ID)的DRAM的256MB~511MB地址空间去。即从指令角度看，源数据存在于Cluster2的DRAM的256MB~511MB空间，目的地址是Cluster3的DRAM的256MB~511MB空间。

DTE执行单元首先需要查询Cluster ID映射表。从Cluster ID映射表中获知，虚拟Cluster2对应物理Cluster14，而虚拟Cluster3对应物理Cluster15，即这个数据搬移指令实际是从物理Cluster14搬移数据到物理Cluster15中去。

之后，DTE执行单元需要查询DRAM地址映射关系，来获知数据在DRAM中真正的地址，有两种方案：

~~**例二方案A**：DRAM地址映射关系信息在DRAM中~~

~~DRAM地址映射关系信息存在于每个Cluster的DRAM的最低1KB空间，逐进程逐Cluster存放。~~

~~当前进程号是1，涉及到的物理Cluster是14和15，所以地址映射关系存在于DRAM最低2KB中的240B和248B的偏移地址内（见下表）。DTE执行单元需要发起对物理Cluster14的DRAM的240B/248B地址偏移数据的读请求。等数据读取回来后，经解析，可以获知，进程1在物理Cluster14的DRAM中占用的地址空间是0~1GB，而在物理Cluster15的DRAM中占用的地址空间是1GB~2GB。~~

~~现在具体的地址信息已经明确，即源数据存放在物理Cluster14的DRAM的256MB~511MB中，而目的地址是物理Cluster15的DRAM的1GB+256MB ~ 1GB+511MB。~~

~~DTE执行单元开始执行数据搬移动作，它先发起对物理Cluster14的DRAM的读请求，读数的起始地址是256MB，读数大小是256MB；待收到所需的数据后，再发起对物理Cluster15的DRAM的写请求，写请求的起始地址是1GB+256MB，全部写数据大小是256MB。~~

~~进程号~~	~~Cluster ID~~	~~地址应该关系在DRAM最低2KB中的位置偏移~~
~~进程 0~~	~~Clus 0~~	~~0~~
	~~Clus 1~~	~~8B~~
	~~……~~	~~……~~
	~~Clus 15~~	~~120B~~
~~进程 1~~	~~Clus 0~~	~~128B~~
	~~Clus 1~~	~~136B~~
	~~……~~	~~……~~
	~~Clus 14~~	~~240B~~
	~~Clus 15~~	~~248B~~
~~进程 2~~	~~Clus 0~~	~~256B~~
	~~……~~	~~……~~

~~……~~	~~……~~	~~……~~

~~（不采用的方案）~~

例二方案B： DRAM地址映射关系信息在配置寄存器中

DRAM地址映射关系信息存在于PE的配置寄存器中。

当前进程号是1，涉及到的物理Cluster ID是14和15，所以需要的寄存器是 pg1_clus14_start_addr/pg1_clus14_end_addr 和 pg1_clus15_start_addr/pg1_clus15_end_addr 。DTE从这两组寄存器中获知，进程1在物理Cluster14的DRAM中占用的地址空间是0~1GB，而在物理Cluster15的DRAM中占用的地址空间是1GB~2GB 。

后续操作和方案A相同~~，不再赘述~~。现在具体的地址信息已经明确，即源数据存放在物理Cluster14的DRAM的256MB~511MB中，而目的地址是物理Cluster15的DRAM的1GB+256MB ~ 1GB+511MB 。

DTE执行单元开始执行数据搬移动作，它先发起对物理Cluster14的DRAM的读请求，读数的起始地址是256MB，读数大小是256MB；待收到所需的数据后，再发起对物理Cluster15的DRAM的写请求，写请求的起始地址是1GB+256MB，全部写数据大小是256MB 。

4.4.3 C2C module

在C2C模块中，和Memory系统的地址转换相关的工作有两个方面：

1）各种映射关系的配置工作；

2）访存指令需要访问存储单元时的地址映射工作；

下面详细介绍各工作的具体操作。

4.4.3.1 C2C模块中各种映射关系的配置工作

为了支持地址转换功能，需要配置如下信息： CLuster ID映射关系，DRAM地址映射关系。

Cluster ID映射关系的配置之前没有提过存放在DRAM中的方案，此处为了方便，把Cluster ID映射的配置和DRAM地址映射的配置一起说明。

对地址转换所需的映射关系的配置有两种方案：

~~**方案A：地址转换映射关系配置到DRAM中。**~~

~~系统为每个进程定义了地址转换映射关系后，通过一个DMA类型的packet命令，将这些映射关系数据从Host中搬移到各个Cluster的DRAM的指定地址空间去。该过程不需要C2C模块的参与~~。（不采用的方案）

方案B：地址转换映射关系配置到配置寄存器中。

系统为每个进程定义了地址转换映射关系后，通过寄存器配置命令去配置相应的配置寄存器。

在4.3.1.2章节提到了不同配置方式，在不同配置方式时，C2C模块的行为各不相同：

~~***“方案B-1”***：硬连线广播 - 地址转换映射关系以硬连线的方式输送到C2C模块的接口上，C2C模块可直接使用；~~

~~***“方案B-2”***：随路TBC命令下发 - 地址转换映射关系跟随TBC命令随路下发。该方案对C2C模块不适用；~~

~~***“方案B-3”***：以配置信息命令方式广播 - 地址转换映射关系以专门的配置信息命令的方式发送。该方案对C2C模块不适用；~~（不采用的方案）

“方案B-4”：各模块有自己的配置寄存器空间 - 在此方案下，C2C模块有自己的配置寄存器空间，C2C模块需要接收寄存器配置命令，并把相应的配置寄存器信息锁存下来；

由4.3.1.3章节可知，我们选择方案B-4。

4.4.3.2 C2C模块中访存时的地址映射工作

在C2C模块中，无论是Host发起的packet命令，还是程序指令流产生的跨Chip数据搬移指令，执行数据搬移操作时，都有可能需要进行地址转换。

对于C2C模块中的地址转换，需要分两个角度来说明：

角度一：如果当前的访存请求，无论是读请求还是写请求，无论该请求的源头是来自本Chip还是其他Chip，无论是Host的packet命令，还是程序指令流的指令类型，只要目的存储单元是在本chip内的，都需要对地址的信息进行Cluster ID映射和DRAM地址映射的转换；

角度二：如果当前的访存请求，目的存储单元是位于其他chip内的，此时C2C模块不需要进行Cluster ID映射和DRAM地址映射的地址转换，因为对应的地址转换映射信息是存在于其他chip内的，本chip的C2C模块并不知道这些信息，C2C模块只需要把原始的Cluster ID, PE ID, internal address等地址信息，即48位地址信息的最低[38:0]发送出去即可，由目的chip的C2C模块进行地址转换；

无论是角度一还是角度二，执行C2C类型的数据搬移操作时，涉及到Chip ID的映射问题。暂不讨论由谁负责执行Chip ID的映射转换，下面提出两种可能的方案：

方案A：Chip ID的映射关系由配置寄存器指定

该方案中，每个chip都有自己的一套Chip ID映射关系的配置寄存器，由配置寄存器指定一个虚拟Chip ID对应的物理Chip ID是多少。这个方案类似于Cluster ID映射关系的配置。不过因为最大scale-up规模是256个chip，所以每一个映射关系需要8位信息来表示，所以一个32位的配置寄存器中最多可表示4个虚拟Chip ID的映射关系，要把256个chip全部表达完，需要64个配置寄存器。这样的一组配置寄存器属于一个进程，如果有8个进程，则需要有8组这样的配置寄存器。

为了减少配置寄存器的个数，可以考虑使用多级链表的方式对256个chip进行分组后，再实现配置寄存器的管理。暂时不对此展开详细讨论。

方案B： Chip ID的映射关系采用固定的轮转方式

该方案中，无论是分配到哪个物理Chip上的Kernel，在软件层面上看，都认为当前这个Chip ID是0（虚拟ID），认为在物理ID上和本chip的物理ID是连续的chip的Chip ID是1（虚拟ID），依次类推，直到全部256个chip排布完。在实际应用时，对每个chip都会使用配置寄存器配置其物理Cluster ID号，执行程序时，则是把命令或者指令中指定的虚拟ID加上该物理Cluster ID号，来计算需要访问的目的chip的物理Chip ID信息。这时，计算的结果可能会超过256（但不会超过512），只需用256取模，即截取结果的最低8位即可。

举例：

例一：当前有Chip A， Chip A上执行的命令认为Chip A 就是Cluster 0，然后它需要在Cluster 0 和 Cluster 4之间进行C2C操作。在物理层面看，对Chip A 配置的物理Cluster ID号是10，则它实际上是在物理Cluster 10 和 物理Cluster 14之间进行数据搬移；

例二：当前有Chip B， Chip B上执行的命令认为Chip B 就是Cluster 0，然后它需要在Cluster 0 和 Cluster 4之间进行C2C操作。在物理层面看，对Chip B 配置的物理Cluster ID号是255，则它实际上是在物理Cluster 255 和 物理Cluster 3之间进行数据搬移；

例子三：当前有Chip C 和 Chip D，Chip C上执行的命令认为Chip C 就是Cluster 0，然后它需要在Cluster 0 和 Cluster 235之间进行C2C操作。Chip D上执行的命令认为Chip D 就是Cluster 0，然后它需要在Cluster 0 和 Cluster 21之间进行C2C操作。在物理层面看，对Chip C 配置的物理Cluster ID号是10，对Chip D 配置的物理Cluster ID号是245，则它实际上是在物理Cluster 10 和 物理Cluster 245之间相互搬移数据；

Chip ID的映射转换由谁来执行，以及采用什么方案，后续再决定。（**TBD: 待确定，硬件的什么地方来实现该功能，具体实现方式**）

话题收回，从角度一看，对当前chip内部进行访存时，需要进行地址转换。这类地址转换的执行过程和GCS模块进行数据搬移时的地址转换过程很相似，可参见4.4.1.3章节的描述，此处不再赘述。

C2C模块内部的地址转换功能也会是统一的由一个功能IP来实现，这是个通用IP--ATU，具体可见“SIPU 1.0 ATU架构设计方案”文档 SIPU 1.0 ATU架构设计方案v0.5。无论C2C模块内部有几组C2C执行单元，在请求上dNoC总线之前使用ATU实现地址转换就可以了。

4.4.4 Media-SS module

在Media-SS模块中，和Memory系统的地址转换相关的工作有两个方面：

1）各种映射关系的配置工作；

2）访存指令需要访问存储单元时的地址映射工作；

下面详细介绍各工作的具体操作。

4.4.4.1 Media-SS模块中各种映射关系的配置工作

为了支持地址转换功能，需要配置如下信息： CLuster ID映射关系，DRAM地址映射关系。

Cluster ID映射关系的配置之前没有提过存放在DRAM中的方案，此处为了方便，把Cluster ID映射的配置和DRAM地址映射的配置一起说明。

对地址转换所需的映射关系的配置有两种方案：

~~**方案A**：地址转换映射关系配置到DRAM中。~~

~~系统为每个进程定义了地址转换映射关系后，通过一个DMA类型的packet命令，将这些映射关系数据从Host中搬移到各个Cluster的DRAM的指定地址空间去。该过程不需要Media-SS模块的参与。~~（不采用的方案）

方案B：地址转换映射关系配置到配置寄存器中。

系统为每个进程定义了地址转换映射关系后，通过寄存器配置命令去配置相应的配置寄存器。

在4.3.1.2章节提到了不同配置方式，在不同配置方式时，Media-SS模块的行为各不相同：

~~***“方案B-1”***：硬连线广播 - 地址转换映射关系以硬连线的方式输送到Media-SS模块的接口上，Media-SS模块可直接使用；~~

~~***“方案B-2”***：随路TBC命令下发 - 地址转换映射关系跟随TBC命令随路下发。该方案对Media-SS模块不适用；~~

~~***“方案B-3”***：以配置信息命令方式广播 - 地址转换映射关系以专门的配置信息命令的方式发送。该方案对Media-SS模块不适用；~~（不采用的方案）

“方案B-4”：各模块有自己的配置寄存器空间 - 在此方案下，Media-SS模块有自己的配置寄存器空间，Media-SS模块需要接收寄存器配置命令，并把相应的配置寄存器信息锁存下来；

由4.3.1.3章节可知，我们选择方案B-4。

4.4.4.2 Media-SS模块中访存时的地址应该工作

在Media-SS模块中，在执行数据搬移操作时，需要对地址信息进行地址转换。

Media-SS模块中和访存相关的具体的架构目前还没有详细的对接，可能存在的场景有可能和PE模块的类似，其实现方式可参见4.4.2.2章节。后续如有其他特殊的需求，再详细展开说明，此处不再赘述。

4.4.5 ICI module

地址转换的工作在GCS(包括DMA模块)模块，PE（包括RV-core的的LSU执行单元，Tile-core的TLSU/DTE执行单元）模块，C2C模块中已经完成，在ICI总线上看到的地址信息已经是物理的地址信息，即Cluster ID是物理Cluster ID，存储单元的地址也是数据在存储单元中真实的地址。

4.4.6 NoC module

在Noc总线上看到的地址信息也已经是物理的地址信息。

5. Address Interleaving

5.1 L2B (Shared Memory)

5.2 DRAM (Global Memory)

6. Configuration Register

7. Compiler/Driver Information