千线程并行模拟器

用户使用手册

中国科学院计算技术研究所

目 录

[千线程并行模拟器 I](#_Toc458592555)

[用户使用手册 I](#_Toc458592556)

[中国科学院计算技术研究所 I](#_Toc458592557)

[目 录 I](#_Toc458592558)

[1 运行环境要求 3](#_Toc458592559)

[2 项目文件结构 3](#_Toc458592560)

[3 建立用户目录 3](#_Toc458592561)

[4 组件开发接口 4](#_Toc458592562)

[5 模拟器组件配置 5](#_Toc458592563)

[5.1 Core配置 5](#_Toc458592564)

[5.1.1 流水线部件配置 5](#_Toc458592565)

[5.1.2 分支预测策略配置 6](#_Toc458592566)

[5.1.3 存储器映射配置 6](#_Toc458592567)

[5.2 Mesh模块的配置 7](#_Toc458592568)

[5.3 双环模块配置 8](#_Toc458592569)

[5.4 XBAR模块配置 9](#_Toc458592570)

[5.5 Cache模块配置 9](#_Toc458592571)

[5.6 Memory配置 10](#_Toc458592572)

[6 拓扑搭建方法 10](#_Toc458592573)

[6.1 创建原子模块 10](#_Toc458592574)

[6.2 搭建拓扑 11](#_Toc458592575)

[6.2.1 mesh4x4拓扑介绍 11](#_Toc458592576)

[6.2.2 创建复合模块 12](#_Toc458592577)

[6.2.3 .so、.so.c、.so.objects文件的生成 14](#_Toc458592578)

[6.2.4 与拓扑有关的组件参数配置 14](#_Toc458592579)

[6.3 拓扑搭建常见错误汇总 23](#_Toc458592580)

[6.3.1 Error 1 23](#_Toc458592581)

[6.3.2 Error 2 23](#_Toc458592582)

[6.3.3 Error 3 24](#_Toc458592583)

[6.3.4 Error 4 24](#_Toc458592584)

[6.3.5 Error 5 24](#_Toc458592585)

[7 模拟器编译方法 24](#_Toc458592586)

[7.1 用户开发组件编译方法 24](#_Toc458592587)

[7.2 测试用例编译方法 24](#_Toc458592588)

[8 运行 24](#_Toc458592589)

[8.1 编译 24](#_Toc458592590)

[8.2 运行 24](#_Toc458592591)

[8.3 多配置批量执行功能 24](#_Toc458592592)

[8.4 调试功能 25](#_Toc458592593)

[8.5 性能统计 26](#_Toc458592594)

[8.5.1 core性能统计 26](#_Toc458592595)

[8.5.2 Cache性能统计 27](#_Toc458592596)

[8.5.3 NoC的性能参数 28](#_Toc458592597)

[8.5.4 SPM性能统计 28](#_Toc458592598)

[8.5.5 memory的统计信息 29](#_Toc458592599)

# 运行环境要求

模拟器开发及测试环境如下：

|  |  |  |
| --- | --- | --- |
| **项目** | **要求** | **备注** |
| 硬件平台 | Intel Xeon CPU | 16核服务器 |
| 操作系统 | Linux 2.6.18-128.e15  Linux 3.10.0-123.e17 | 其它操作系统暂未测试 |
| GCC | 4.6.1版本 |  |
| 交叉编译GCC | arm-linux-gcc-4.3.3版本 |  |

# 项目文件结构

整个工程文件的顶层文件夹名为SimICT，其包含六个子文件夹，如下所示：



1. benchmarks文件夹为选取的几个典型的大数据应用测试程序，包括K-Means、KMP、PageRank、RNC、TeraSort、WordCount；
2. compile 文件夹为给用户提供的应用程序编译环境，包括Makefile、pthread库文件等等；
3. components文件夹为已开发的各功能组件，包括core、cache、memory、mesh、ring、xbar、mact等等；
4. experiments文件夹为已搭建好的各实验环境，包括四个字符加速结构的实验环境ACC\_\*\*\*、高密度网络实验环境HDNoC、访存请求批量收集实验环境MACT、带锥形访存通路的两级环实验环境：ring\_cone\_mesh，以及对千核处理器结构进行性能功耗评估的工具McPAT[[1]](#footnote-1)；
5. topos文件夹为已搭建好的各拓扑环境，包括mesh结构的16、64、1024核，两级环结构的1024核，xbar结构的1024核拓扑等；
6. workspace文件夹为用户开发/执行目录，里面包括用户搭建的拓扑结构和组件，用户运行自己搭建的目标系统需要在此文件夹下的xtec-simict/sim\_project/users里面（里面已有一些可运行的配置）。users目录下的每个文件夹即代表一个可以执行的模拟目标系统。在每个模拟目标系统中可以编辑和运行整个模拟器。

# 建立用户目录

用户搭建自己的目标模拟系统时，需要在workspace/xtec-simict/sim\_project/users建立自己的文件夹。例如，如果想搭建16核的mesh结构，可以命名文件夹名为mesh\_4x4。然后将需要的已开发好的各个功能组件（在components中）复制到此文件夹下，逐个编译，组件的编译需要使用已有示例中的Makefile进行改写。

# 组件开发接口

由于SimICT是基于PDES（并行离散事件模拟）实现的，因此组件的开发也必须采用基于事件的功能实现方式。例如对于Memory的开发，组件需要能够处理“读操作”，“写操作”等，对于读操作事件的处理，Memory需要按照地址进行读操作并返回“读数据”事件，即读操作的数据返回。对于写操作事件，Memory写完成之后需要返回“写成功”事件。事件的定义可以更加组件模拟的精确来实现，即事件可大可小。

功能组件在模拟器中即为一个一个的原子模块，原子模块的内部逻辑需要定义三个回调函数来实现。除**simict\_create\_object**外，回调函数内部可以调用**simict\_write\_port**函数向本地端口发送消息。

void\* simict\_create\_object(u64 object\_id);

**simict\_create\_object** 系统在启动时会调用这个函数来创建这个模块的对象，如果一个模块在运行时有多个对象，这个函数会被调用多次。每次调用时，参数object\_id是当前对象的唯一标识。函数的返回值会成为当前对象的私有数据，此后，系统回调其它函数时，这个返回值会作为参数（usr\_data）被传入。注意：这个函数内部不能调用simict\_write\_port函数。

void simict\_start\_object(u64 object\_id, void \*usr\_data);

**simict\_start\_object** 系统回调所有对象的simict\_create\_object之后调用该函数。在该函数中可以调用simict\_write\_port函数。参数object\_id是当前对象的唯一标识。参数usr\_data是当前对象的私有数据。

void simict\_write\_port(u64 port\_id,

u64 delay, const char \*data, u64 data\_bytes);

**simict\_write\_port** 对象通过调用该函数来发送消息。参数port\_id是接收消息对象的端口号。参数data是发送的数据。参数data\_bytes是数据长度。参数delay是消息的延迟时间，必须大于等于1。

void simict\_port\_msg\_proc(void \*usr\_data,

u64 port\_id, char \*msg, u64 msg\_bytes);

**simict\_port\_msg\_proc**对象通过调用该函数来接收消息。参数msg是接收的数据，对应simict\_write\_port函数的参数data，由消息的发送者传入。参数port\_id表示当前对象接收消息的端口号。参数usr\_data是当前对象的私有数据。参数msg\_bytes是收到的消息长度。

这几个接口函数是组件与框架的桥梁，需要在组件的主文件中引用公用头文件“simict.h”。 

# 模拟器组件配置

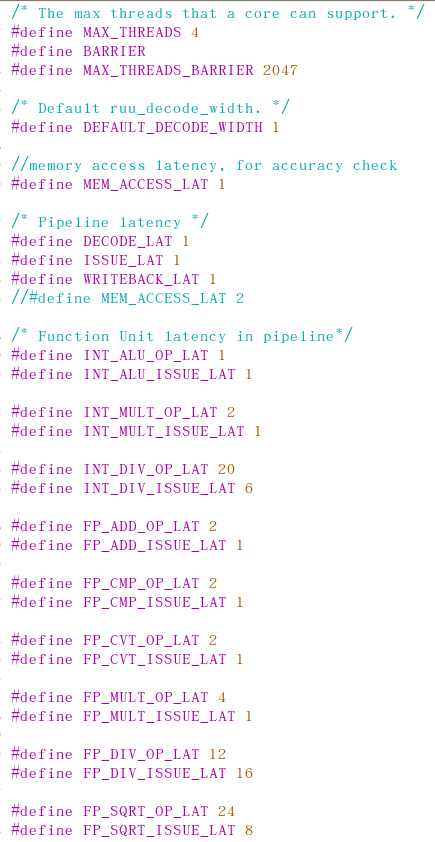
## Core配置

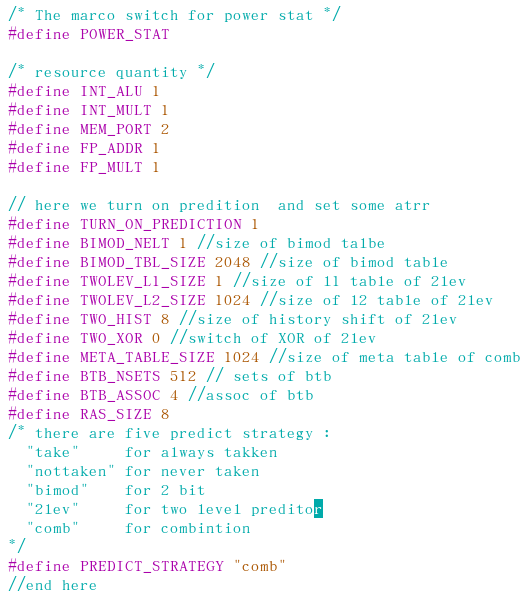
Core模块支持ARMv6指令级、SMT、分支预测。ARM V6指令集定义、译码，实现相关通用及系统寄存器定义，实现中断、异常处理功能，单个ARM Core组件支持核内8线程SMT ARM1176单发射8级流水，实现分支预测功能，实现并行流水线等功能。

Core组件下的conf.h文件为用户提供配置选项。

### 流水线部件配置

，主要配置包括SMT支持的线程数、各流水部件数目等等，如下所示：



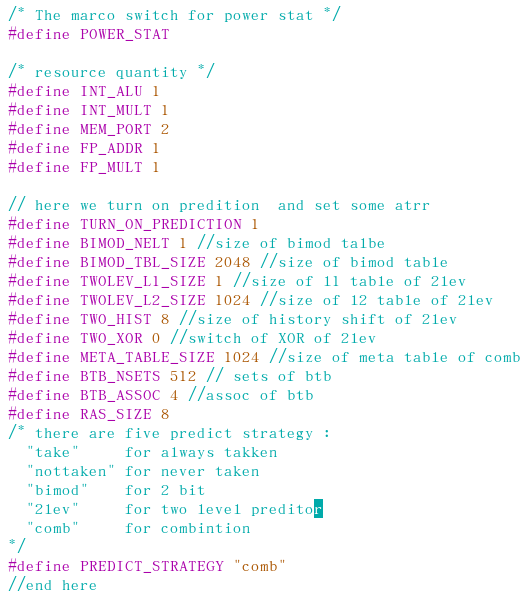


### 分支预测策略配置

在conf.c的设置定义了 TURN\_ON\_PREDICTION的话，就可以打开分支预测功能。

在conf.c中通过设置 PREDICT\_STRATEGY 的值来选择不同的分支预测法，其值包括taken,nottaken,bimod,2lev,comb，分别对应着五种不同的分支预测算法。

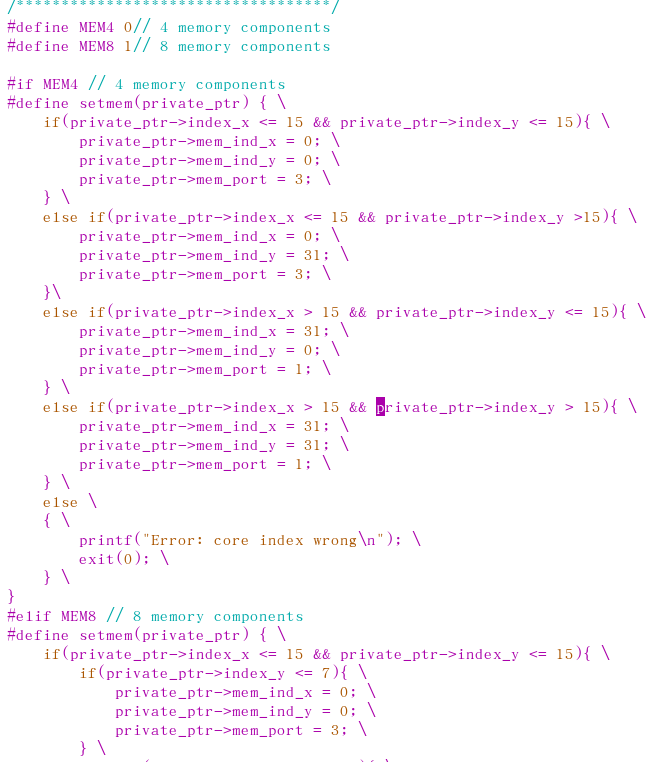
分支预测所有相关的参数在在core.c中配置。分支预测算法参数：



|  |  |  |
| --- | --- | --- |
| 可配参数 | 参数意义 | 推荐配置范围 |
| bimod\_config[0] | Bimode分支预测表长度 | 128-65536 |
| twolev\_config[0] | 一级表（分支历史表）长度 | 1-1024 |
| twolev\_config[1] | 二级表（分支模式表）长度 | 128-65536 |
| twolev\_config[2] | 历史移位器宽度 | 1-8 |
| twolev\_config[3] | 两级分支预测器异或开关 | 0-1 |
| comb\_config[0] | Meta表长度 | 128-65536 |
| btb\_config[0] | 分支目标缓冲区大小 | 128-65536 |
| btb\_config[1] | 分支目标缓冲区路数 | 1-1024 |

### 存储器映射配置

其中的MEMX宏用来确定每个从core发出的消息最终要发给哪个memory，为内存映射策略。

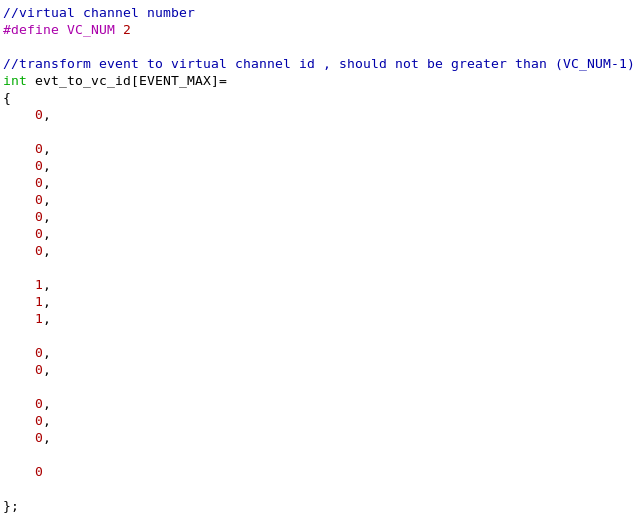


## Mesh模块的配置

支持大规模片上互联网络的组件，实现互联结构可配置，路由、仲裁等功能。通过修改宏，来修改这些参数的设置，修改后，需要重新编译一次mesh模块。在user.h中，有如下宏DEBUG\_SWITCH，是一个调试开关，用来控制是否开启调试模式。开启时，会输出一些调试信息。其它功能可配置的参数都置于头文件conf.h中， STATISTICS通过宏定义控制是否进行性能统计。



VC\_NUM虚通道的数量。evt\_to\_vc\_id 记录着msg.h中每种事件对应的虚通道id。数组大小应与事件数量一致。虚通道的修改方式：a. 修改虚通道的数量（记为n)；b. 修改evt\_to\_vc\_id中事件对应的优先级，0的优先级最低，n-1的优先级最高。



PORT\_WIDTH 物理链路的宽度

ROUTING\_INFO\_SIZE 每个packet路由信息的长度

SELF\_DELAY router给自己发送消息的延迟

VC\_DELAY 虚通道发送消息的延迟

ACK\_DELAY 发送ack或nack的延迟

OUTPORT\_PACKET\_BUFF\_SIZE router输出方向虚通道上packet缓存队列的大小，即可缓存多少个packet，大小至少为1

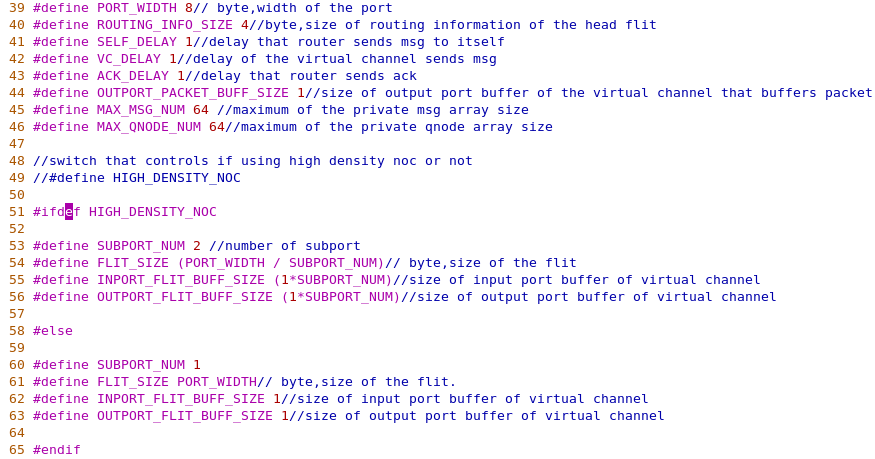
HIGH\_DENSITY\_NOC 宏定义，控制是否启用高密度网络

SUBPORT\_NUM 子端口的数量，即将每个端口划分为多少个子端口。未开启高密度网络时，应将其设置为1。

FLIT\_SIZE 微片的宽度。开启高密度网络时，PORT\_SIZE为FLIT\_SIZE的SUBPORT\_NUM倍；未开启高密度网络时，FLIT\_SIZE等于PORT\_SIZE

INPORT\_FLIT\_BUFF\_SIZE 每个端口输入方向的虚拟通道缓存队列的长度。开启高密度网络时，大小至少为SUBPORT\_NUM，建议为SUBPORT\_NUM的整数倍；未开启高密度网络时，大小至少为1

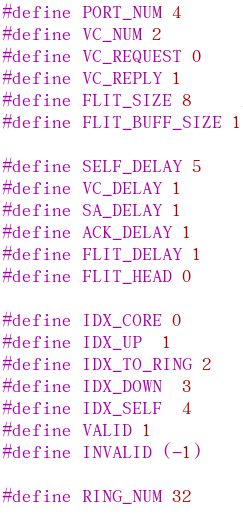
OUTPORT\_FLIT\_BUFF\_SIZE 每个端口输出方向的虚拟通道缓存队列的长度。开启高密度网络时，大小至少为SUBPORT\_NUM，建议为SUBPORT\_NUM的整数倍；未开启高密度网络时，大小至少为1。



其它配置见第6章拓扑搭建方法章节。

## 双环模块配置

双环路由器的配置方法与Mesh结构的Router配置方法类似，具体选项如下：



其它配置见第6章拓扑搭建方法章节。

## XBAR模块配置

XBAR路由器的配置方法与Mesh结构的Router配置方法类似，具体选项如下：

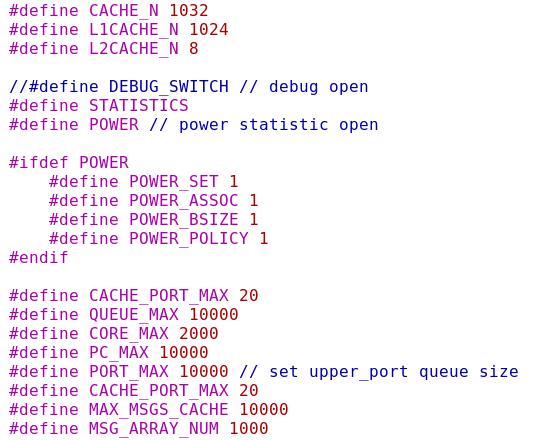


其它配置见第6章拓扑搭建方法章节。

## Cache模块配置

Cache能够配置为多级属性的Cache功能组件，支持LRU等替换策略，支持容量、宽度、端口数可配，支持时序精确级模拟。SPM支持大小、映射范围可配置。

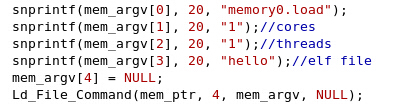
cache的conf.h文件在cache/src/目录下，如下所示：



由于统计需要，用户必须在conf.h中指明cache/module-info中生成的cache的总个数（CACHE\_N）、L1级cache的个数（L1CACHE\_N）、L2级cache的个数（L2CACHE\_N）。DEBUG\_SWITCH为debug开关，打开时会输出一些debug信息。STATISTICS为统计开关，打开时cache将对性能进行统计并生成统计文件。POWER为功耗统计开关，打开时会进行一些功耗相关信息的统计（统计输出该cache的set、assoc、bsize、policy）。CACHE\_PORT\_MAX表示多端口cache最多可以定义的端口个数。QUEUE\_MAX、CORE\_MAX、PC\_MAX、PORT\_MAX代表各个cache队列数组的长度，如果运行期间发生溢出将会提示用户将这些队列改大。MAX\_MSGS\_CACHE、MSG\_ARRAY\_NUM是与消息池相关的参数。

## Memory配置

Memory 组件不需要额外配置，只需要在加载文件时指定为多少核心加载ELF文件，每个核心运行多少线程。通过在代码中显式指定（memory.c文件）。如下图：



要注意，每个核心最多支持8个线程，每个核心的线程数应在1-8之间。

# 拓扑搭建方法

框架将模块分为原子模块和复合模块。如果将搭建框架比作建造房屋，原子模块就相当于砖瓦，它负责完善自己模块内的逻辑功能（砖瓦要烧制得结实），并给外界提供合适的端口（砖瓦要烧制得整齐）；复合模块就相当于设计图纸，它规定了需要用哪些原子模块（需要用哪些种类的砖瓦，用多少），以及原子模块之间如何互联（砖瓦之间如何组装）。

## 创建原子模块

步骤如下：

1) 在工程目录（sim\_project/users）下建立一个文件，随意起名，如叫tmp，tmp文件中写入要创建的原子模块的名字和端口，格式如下。

((name "module\_name") (ports "portA" "portB" ...))

2) 保存tmp文件。在终端输入命令：

cat tmp | ../../core/bin/init-module-dir module\_name ./module\_name atomic

即可生成一个名叫module\_name的原子模块。

以cache模块为例具体说明。

1) 在工程目录下建立tmp文件，内容为：



表示将要创建的原子模块名叫cache，cache有3个端口。

2) 保存tmp文件。在终端输入命令：



即在当前目录下生成了一个cache目录。



cache模块创建完成。

## 搭建拓扑

搭建拓扑就是将我们已经创建好的原子模块按照需求组装起来。组装方式可以多种多样，比如使用core、mesh、cache、memory四个原子模块我们已经组装了mesh4x4、mesh8x8、mesh16x16、mesh32x32等多种拓扑，基本无需改动原子模块代码。以下举例说明如何搭建拓扑。

### mesh4x4拓扑介绍

图1是mesh4x4的拓扑，用到四类原子模块：core、mesh、cache、memory。其中core、mesh、cache的实例各16个，memory的实例8个。图2给出了各个原子模块的端口示意图。



图1 mesh4x4拓扑



图2 各原子模块端口示意图

core：两个端口。端口0连接自己，端口1与cache的端口1互连。

cache：四个端口。端口0连接自己，端口1与core的端口1互连，端口4与mesh的端口4互连，端口2、3在图1拓扑中未用。

memory：仅有一个端口0，与mesh的0/1/2/3端口互连。mesh的哪个端口与之相连视memory位置而定，如(0, 0)位置mesh的3端口跟memory互连。

mesh：六个端口。端口5连接自己，端口4与cache的端口4互连，其余4个端口要么和其他mesh实例互连，要么和memory互连。

### 创建复合模块

创建复合模块的过程与创建原子模块类似。

步骤如下：

1) 在工程目录下建立一个文件，随意起名，如叫tmp，tmp文件中写入要创建的复合模块的名字、端口、原子模块的实例化、实例间的互连，格式如下。

((name "module\_name")(ports "portA" "portB" ...)(objects ("module\_A" "objA1" "objA2" ... )("module\_B"... ))(connections ("src\_obj\_name" "src\_port\_name" "dest\_obj\_name" "dest\_port\_name")(...)...))

name：复合模块的名字。

ports：复合模块的端口。

objects：原子模块实例化。

connections：实例间的互联。

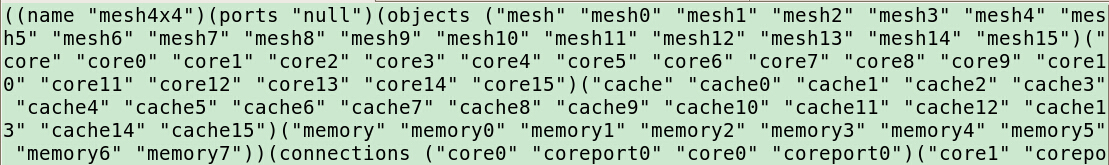
2) 保存tmp文件。在终端输入命令：

cat tmp | ../../core/bin/init-module-dir module\_name ./module\_name compound

即可生成一个名叫module\_name的复合模块。

以mesh4x4模块为例具体说明。

**1) 在工程目录下建立tmp文件**。内容为：

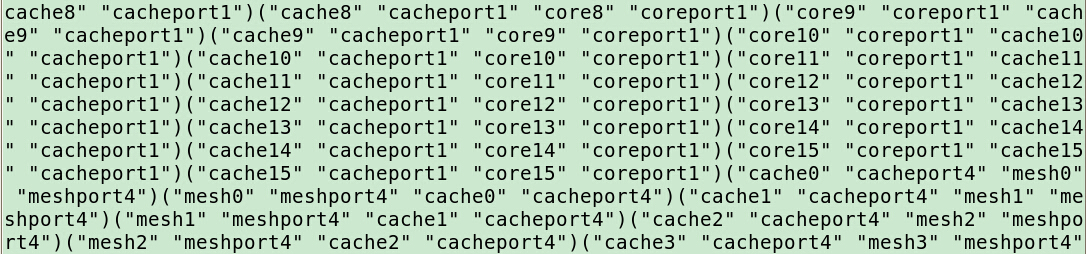


表示要创建的复合模块名字为mesh4x4，它属于顶层模块，所以没有定义端口，ports为null。objects给出了所有原子模块的实例化，例如mesh有16个实例，编号为0~15。connections给出了所有实例间的互连关系。

下图展示了**各实例自己和自己互连**的情况。只需一根线描述消息传递的单向关系。



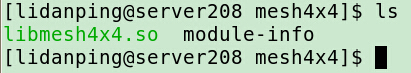
下图展示了不同实例间的互连情况。需要两根线描述消息传递的双向关系。例如core10和cache10的互连，core10的消息要能发给cache10 ("core10" "coreport1" "cache10" "cacheport1")，cache10的消息也要能发给core10("cache10" "cacheport1" "core10" "coreport1")。



**2) 保存tmp文件**。在终端输入命令：



即在当前目录下生成了一个mesh4x4目录。mesh4x4复合模块创建完成。

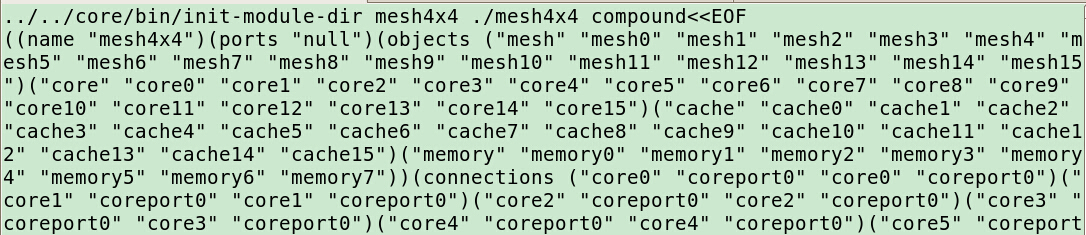


拓扑规模若是比较大，那么手动输入这些拓扑信息就很麻烦，此时需要借助脚本完成。我们在此前搭建好的mesh系列拓扑中已经用python语言编写了一个脚本gen\_conf.py。



图3 gen\_conf.py

运行gen\_conf.py后，生成一个文件mesh16。mesh16内容如下：



mesh16事实上又相当于一个新的脚本，运行mesh16后，即可完成mesh4x4复合模块的创建。

### .so、.so.c、.so.objects文件的生成

运行如下命令：

../../core/bin/gen-runtime-info mesh4x4.so 1 mesh4x4 ./

mesh4x4是欲生成复合模块.so的名字，mesh4x4表示需要用到我们此前生成的复合模块mesh4x4。其中参数1表示模拟器使用一个1线程运行，当采用多线程加速时，参数可配置为2到8，目前最多支持8个线程加速。

我们将上述命令写入了gen\_conf.sh脚本，运行之前要根据已经生成的复合模块相应地修改该脚本。运行“gen\_conf.sh X”后（X为加速的线程数），目录下将多出三个文件：mesh4x4\_X.so、mesh4x4\_X.so.c、mesh4x4.so\_X.objects。

### 与拓扑有关的组件参数配置

在obj\_conf目录下需要修改index.c、mesh\_conf.c、core\_conf.c、cache\_conf.c、mcu\_conf.c几个文件。

#### index.c文件修改

index.c文件只需修改一处，即objs[]数组。其中的元素要替换成与mesh4x4.so.c中的objects[]数组对应。对应关系如下图。

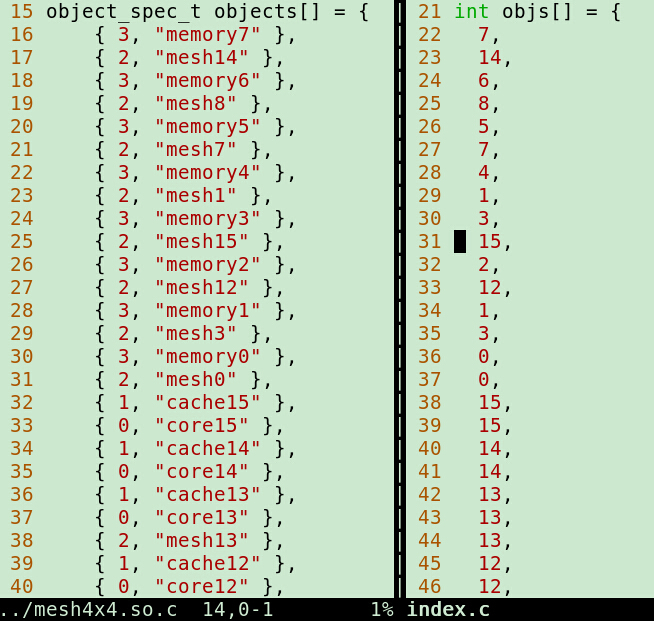


图4 mesh4x4.so.c与index.c

左边是mesh4x4.so.c文件，右边是与之对应的index.c文件。如左边objects中第一个元素memory7，与之对应的objs[]元素即是memory7在所有8个memory实例中的局部编号，为7。事实上，objs[]的元素也可以通过mesh4x4.so.objects文件获得，如下图。提取objs[]元素的过程一般通过脚本完成。

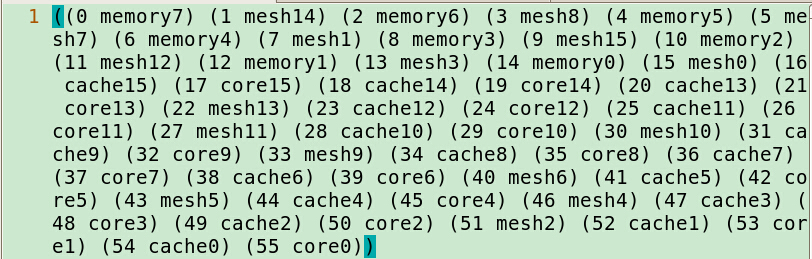
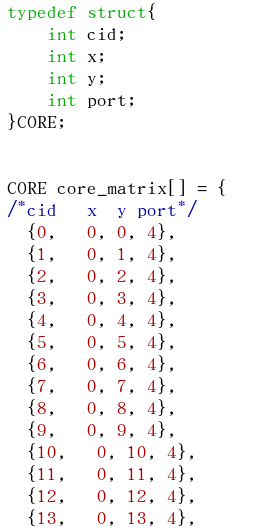


图5 mesh4x4.so.objects

#### Core组件配置

**1） Core与Mesh组件连接：**

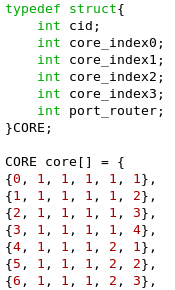


CORE结构体定义了需要配置的各个参数，core数组是具体的配置情况。与Mesh相关的参数含义解释如下：

x/y：与core相连的mesh路由器的坐标值。

port：与core相连的路由器的端口值。注意：是路由器的端口。

**2） Core 与Xbar组件连接：**

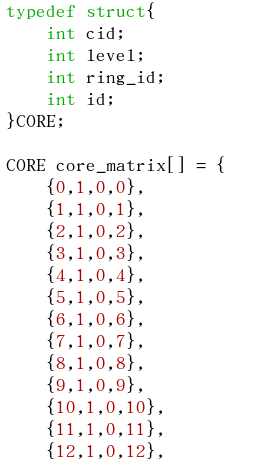


CORE结构体定义了需要配置的各个参数，core数组是具体的配置情况。与Xbar相关的参数含义解释如下：

core\_index：与core相连的路由器的坐标值。

port\_router：与core相连的路由器的端口值，注意：是路由器的端口。

**3） Core 与Ring组件连接：**



CORE结构体定义了需要配置的各个参数，core数组是具体的配置情况。与Ring相关的参数含义解释如下：

level：核出于环的哪一级。

ring\_id：所处子环的编号。

#### 片上网络配置

**1） Mesh片上网络配置：**

MESH\_PARA结构体定义了需要配置的各个参数。各个参数的含义解释如下：

mesh\_id：路由器的绝对id。

x/y：分别代表路由器在片上网络中的坐标。

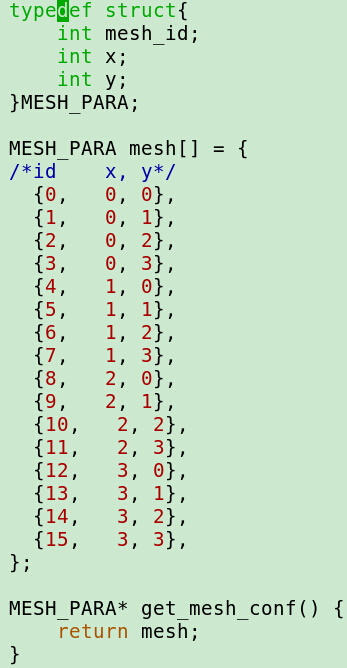


图6 mesh\_conf.c

mesh\_id：mesh的局部编号。

x, y：mesh在拓扑中的位置坐标。参考图1。

**2） Xbar片上网络配置：**

XBAR\_PARA结构体定义了需要配置的各个参数，xbar数组是具体的配置情况。各个参数的含义解释如下：

router\_id：路由器的绝对id。

level：路由器的级别。位于拓扑中心的0号路由器级别最高，为0。直接与0号路由器相连的其他路由器级别为1，依次类推。

xbar\_index：这一系列值标定了该路由器在整个拓扑图中的位置。 0号路由器处于拓扑中心，都为0。1号路由器与0号路由器的1号端口相连，故xbar\_index0==1，其他坐标都未用到，为0。5号路由器与1号路由器的1号端口相连，故xbar\_index0==1、xbar\_index1==1。所以某个路由器的坐标值可以看作以中心路由器为起点到达该路由器的一系列端口值。

portid\_up：该路由器与它上一级路由器相连的端口。注意：是该路由器的端口。在6.1节的说明中我们都规定了为5。

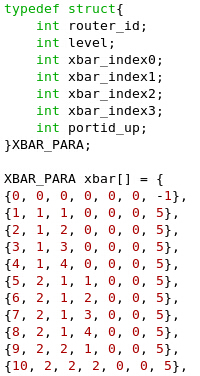


图7 xbar\_conf.c

**1） 双环片上网络配置：**

双环拓扑由主环和子环两类环构成。为描述方便，以下用主路由代表主环上的路由，用子路由代表子环上的路由。以ring4x4的拓扑为例，如下图。该拓扑的主环由4个主路由构成，各个主环上又连接有一个子环，每个子环由4个子路由构成。core连接在各个子环上，而memory连接在各个主环上。ring4x4拓扑连接图：



Ring的端口配置在设计的时候已经固定，用户只需使用默认的配置即可。下图是框架生成ring模块时的端口定义。这些端口有确定的含义，并且与特定的组件相连。主路由和子路由的端口含义有所不同，下面予以说明。



主路由：self端口连接自己、core端口连接memory、up端口连接主环上编号在其之前的主路由（首除外）、down端口连接编号在其之后的主路由（尾除外）、another\_level端口连接该主路由的第一个子路由。

子路由：self端口连接自己、core端口连接core、up端口连接当前子环上编号在其之前的子路由（首除外）、down端口连接当前子环上编号在其之后的子路由（尾除外）、another\_level端口连接与该子环相连的主路由（只有第一个子路由和主路由相连，该端口被使用，其他子路由该端口空闲）。

router\_id：路由器的绝对id。

level：路由器的级别。分别为主环和子环两个结构。

ring\_id：为子环的编号。

Id：为子环内router的编号。

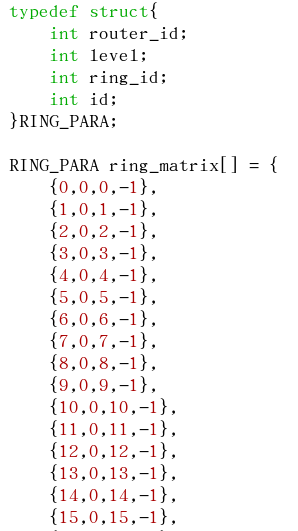


图8 ring\_conf.c

#### Cache配置

以Mesh中的Cache为例。需配置的参数有：

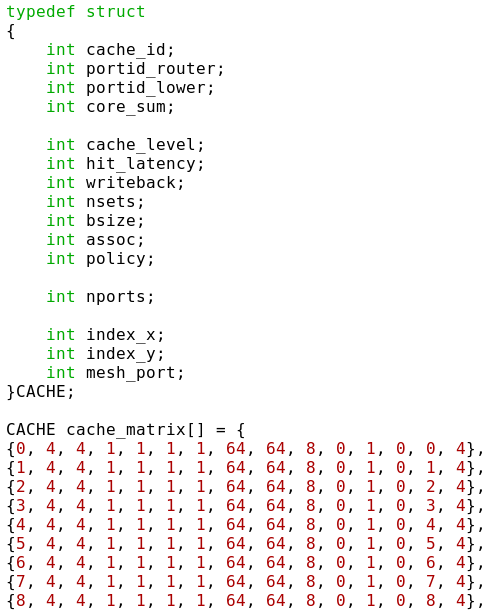
1．框架所需的参数：cache\_id、portid\_router、portid\_lower、core\_sum。

2．Cache基本参数：cache\_level（第几级Cache）、hit\_latency（读写操作延时）、writeback（写回还是写直达：写回：1；写直达：0）、policy（替换策略：LRU：0；Random：1；FIFO：2）、nports（cache的端口数）。

3．Cache大小：nsets（组数目）、bsize（块大小）、assoc（路数目）。Cache大小等于这三个参数的乘积：nsets\*assoc\*bsize，以byte为单位。

4．Cache在mesh网络中的坐标：index\_x、index\_y、mesh\_port。

下图给出了topos/mesh32x32\_l2cache中的一个配置例子，其他片上网络的配置方法可参见相应代码。



1．Cache的端口数可以随意设置，但至少要有3个端口：一个连Cache自己，一个连接上级，一个连接下级。连接上级的端口可以有多个（Cache可能被多个核共享），连接下级的端口只有一个（默认最末一个端口号连接给下级）。所以端口号的配置遵循下面的规则：

假设Cache一共有N个端口

* 0号端口总是连接Cache自己；
* 当Cache被m个核(m<n)共享时，1~m号端口号连接这m个core；
* m+1号端口连接下级。

2．portid\_router，指的是Cache连接router的端口号。注意：是Cache的端口号，不是router的端口号。

portid\_lower，指的是Cache连接下级的端口号，即Cache最末一个端口号。

core\_sum，指的是与Cache相连的core的数目。

* 如果是L1私有Cache，Cache只连一个core，core\_sum配置成1。
* 如果Cache是共享Cache，且共享cache直接和Cache相连或直接和core相连，core\_sum配置成共享该Cache的core的数目。如下图所示的Share Cache其core\_sum都配置成2。



* 如果cache是共享cache，且共享cache连接在memory和mesh之间，则core\_sum配置成0。如图1中L2 Cache所示、如下图Share Cache所示。



#### SPM模块配置

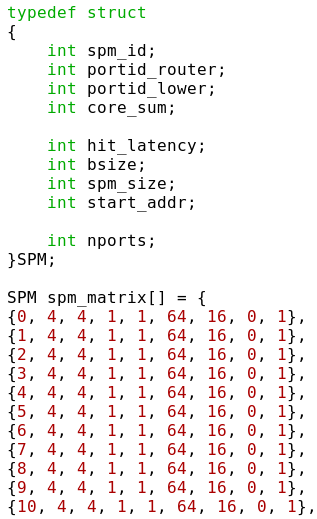
这部分的配置在框架代码的obj\_conf/spm\_conf.c文件，配置之后make，可在obj\_conf目录下生成相应的spm\_conf.o文件。

需配置的参数有：

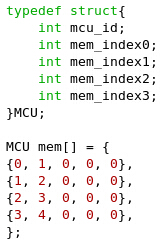
1．框架所需的参数：spm\_id、portid\_router、portid\_lower、core\_sum。

2. SPM基本参数：hit\_latency（读写操作延时）、bsize（块大小）、spm\_size（SPM空间大小）、start\_addr（SPM空间起始地址）、nports（端口数目）。

下图给出了topos/mesh32x32\_spm中的一个配置例子。



#### Memory配置



MCU结构体定义了需要配置的各个参数，mem数组是具体的配置情况。与Xbar相关的参数含义解释如下：

mem\_index：与memory相连的路由器的坐标值。

## 拓扑搭建常见错误汇总

以下解决方法只是经验总结，实际操作过程中还是要具体问题具体分析。

### Error 1

服务器上框架编译复合模块，报错error while loading shared libraries: libtinfo.so.5。

解决方法：在~下建立软链接：ln -s /usr/lib64/libncurses.so.5 libtinfo.so.5，再在.bashrc下添加路径：export LD\_LIBRARY\_PATH=~:$LD\_LIBRARY\_PATH。

### Error 2

运行复合模块，报错：Exception occurred with no-condition value (only support one link per port ...)。

解决方法：检查gen\_conf.py文件，看看是否一个端口上连接了多根线。

### Error 3

运行复合模块，报错：Exception: #f is not of type #。

解决方法：这个错误并不绝对，一般是gen\_conf.py写错了。也可能是创建的原子模块有问题。

### Error 4

欲生成复合模块，结果却生成了原子模块。

解决方法：一般是gen\_conf.py错误。例如，少写右括号。

### Error 5

运行的时候，程序执行错误，出现读入的参数错误或其他问题。

解决方法：一般是生成的obj\_conf/obj\_index.so错误，检查几个\_conf.c文件是否编写正确，是否重新编译等等。

# 模拟器编译方法

## 用户开发组件编译方法

直接在各组件的/src目录下make即可编译，在CFLAGS中增加宏定义选项 -DNEW\_FS即可，最好使用已开发组件里面的makefile进行改写后编译用户新开发的组件。

## 测试用例编译方法

测试用例的编译采用交叉编译器arm-linux-gcc，在编译时使用选项 -static -march=armv6。比如：

Selection_019

# 运行

## 编译

原子模块和复合模块的编译方法见第6章。

各功能模块，在模块的src目录下直接make clean和make即可。

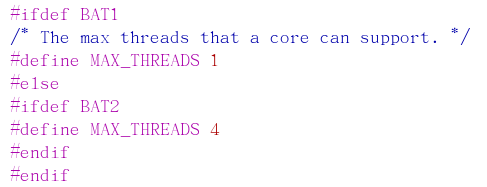
## 运行

例如执行mesh拓扑下的：mesh32x32\_l2cache，则运行以下脚本。可以直接使用命令：time ../../../core/runtime/runtime ./ ./mesh32x32\_X.so obj\_conf/obj\_index.so，或者直接使用提供的脚步run文件，执行“run X”。 (其中X表示模拟器加速的线程数)。

## 多配置批量执行功能

在模拟器运行之前，可以预先实现多种运行配置结构，模拟器可以按照预先设置的多种配置逐个运行，生成多个配置对应的输出结果，方便用户使用。

在各功能组件的配置文中，通过宏定义分别实现不同的配置参数，例如core组件配置如下：



对同一配置选项，实现BAT1和BAT2两种配置参数。

在运行脚本中，实现两种配置的自动编译，自动运行以及不同配置下的输出结果的整理。



例如，对于power的模拟，输出的不同结果为：

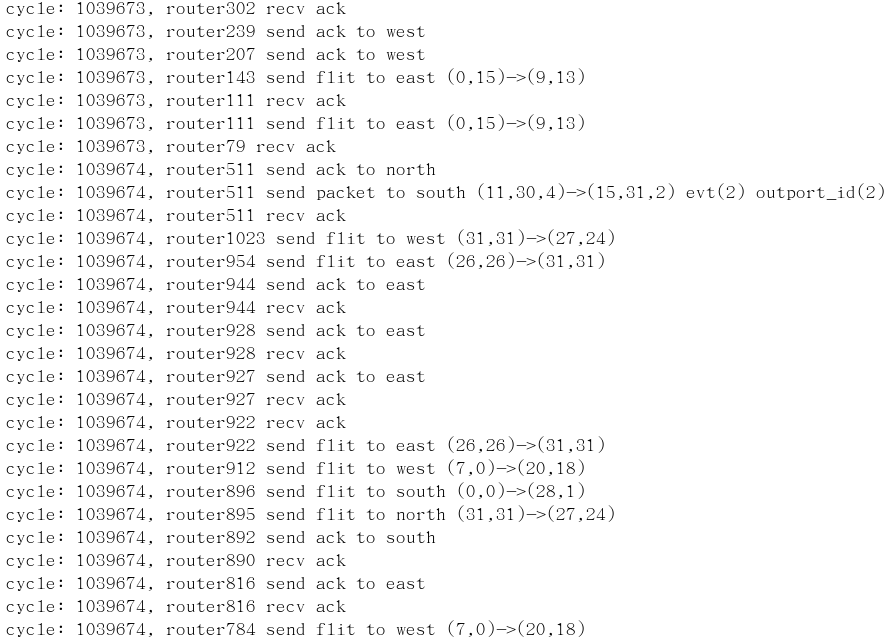


## 调试功能

为了方便应用程序的调试，模拟器开发的组件支持Debug功能，可以打印输出Cycle级的模拟行为。例如Core组件可以打印通用寄存器的内容、记录指令Trace，Mesh组件记录数据包的流动信息（经过的router，端口等），Cache组件记录访问的地址、数据等等。

用户需要在组件文件夹内部的conf.h或其定义的其它配置文件中打开#define XXX\_DEBUG选项，在执行目录建立trace文件夹，各个组件的Debug信息便会输出到trace文件夹下。

例如，下图是Mesh组件的调试输出内容部分截图：

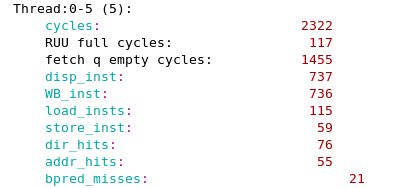


## 性能统计

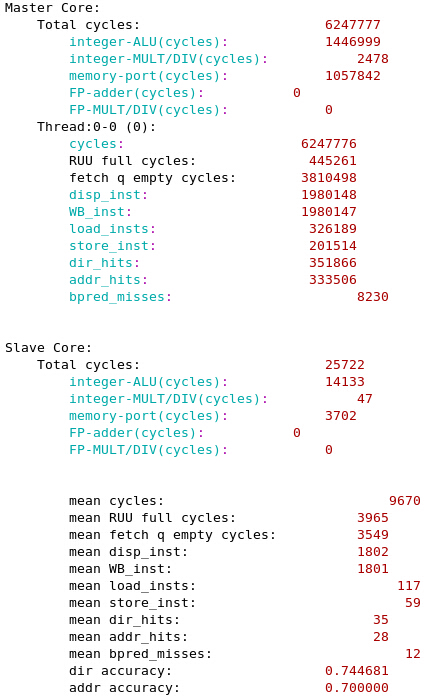
### core性能统计

所有核心及线程的模拟运行完成后程序会自动退出，core和memory分别生成统计信息文件stat/stat\_core.yaml和stat/stat\_memory.yaml，如下图。

单个线程的统计信息：



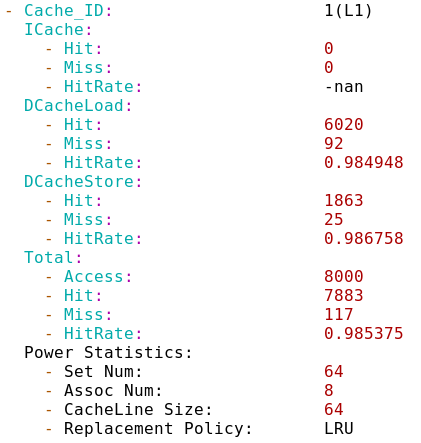
整体的统计信息：



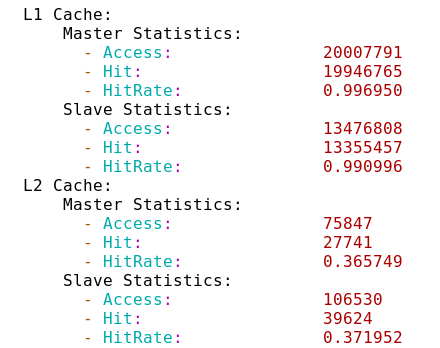
### Cache性能统计

Cache模块性能统计在cache\_statistics函数中实现。统计文件见stat/stat\_cache.yaml。

如下所示为某个Cache的性能统计情况：

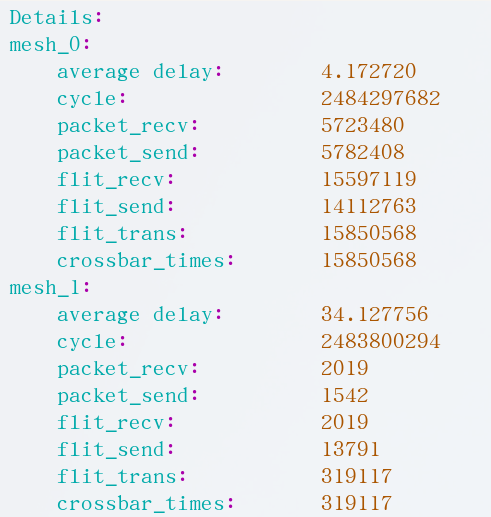


性能统计的总体情况：



### NoC的性能参数

片上网络的性能统计值在stat\_mesh.yaml中，以mesh网络为例：

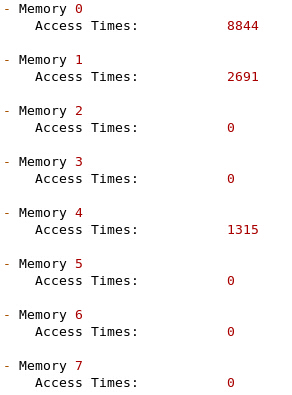


### SPM性能统计

SPM模块性能统计统计在spm\_access函数中实现。统计文件见spm/stat\_spm。统计了每次SPM与内存数据交互的情况，包括发生地址交换的SPM编号，消息请求的地址。如下图。



### memory的统计信息



1. http://www.hpl.hp.com/research/mcpat/ [↑](#footnote-ref-1)