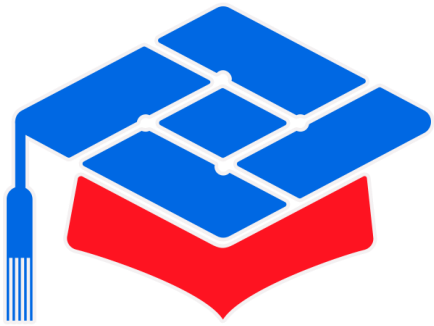


第八届

全国大学生集成电路创新创业大赛

报告类型*：设计报告

参赛杯赛*：中科芯杯

作品名称*：高速多端口共享缓存管理模块

队伍编号*：CICC3263

团队名称*：随便起了个名字

1 项目规划

1.1 赛题分析

随着近代网络技术的高速发展，互联网流量呈爆炸式增长，企业和家庭对网络带宽的要求日益提高。在网络设备对数据包的处理中，存储管理、调度占去了大部分时间，因此大多数存储器的低速缓存能力成为了限制网络处理器进一步提高的瓶颈。支持高速数据存储的 SRAM 管理模块，避免数据包的长度和存储器数据通道的数量对存储器资源的影响可以通过内存回收动态调整空间节约达到存储资源的目的，提高数据存储效率，提升数据校验纠错的能力。

赛题要求设计一款可对 SRAM 进行有效管理的 SRAM 控制器 IP，模块框图如下图 1.1.1 所示。

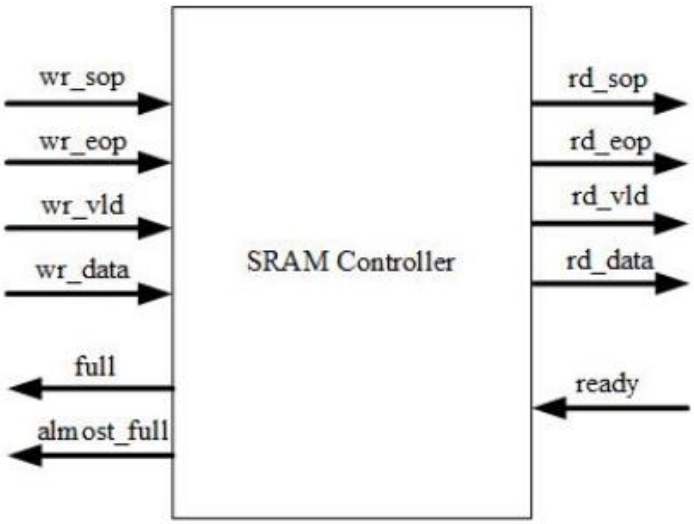


图 1.1.1 SRAM 控制器 IP 模块框图

该 SRAM 控制器 IP 的具体要求如下所示：

1) 支持管理不少于 32 块 256K bit 的 SRAM 单元，满足 8M bit 容量，时钟>250Mhz，支持高速缓存。
2) 支持 16 个端口同时读写缓存，每端口数据传输带宽可达到 1G bps。
3) 每端口支持 8 个优先级队列，按队列缓存数据。
4) 支持按包缓存、调度数据，数据包长 64-1024 字节，每包帧格式含控制帧部分（包括但不限于目的地址、优先级）。

5) 支持内存回收，对已读出的内存空间进行重分配。

6) 支持数据校验。

1.2 开发计划

1) 阶段一：将 32 块 SRAM 进行分割，划分为 16 组的 SRAM，并且端口 0 使用第 0 个组 SRAM，端口 1 使用第 1 个组 SRAM，依次类推。每个端口都不能使用其他组的 SRAM。

2) 阶段二：在阶段一的基础上实现每个端口在对应端口写满的情况下，可以使用使用其他组的 SRAM。

3) 阶段三：进一步优化。

2 系统设计

2.1 系统框图

本作品支持管理不少于 32 块 256K bit 的 SRAM 单元，支持 16 个端口同时读写缓存，每个端口支持 8 个优先级队列，支持内存回收。本作品的整体框图如下图 2.1.1 所示。

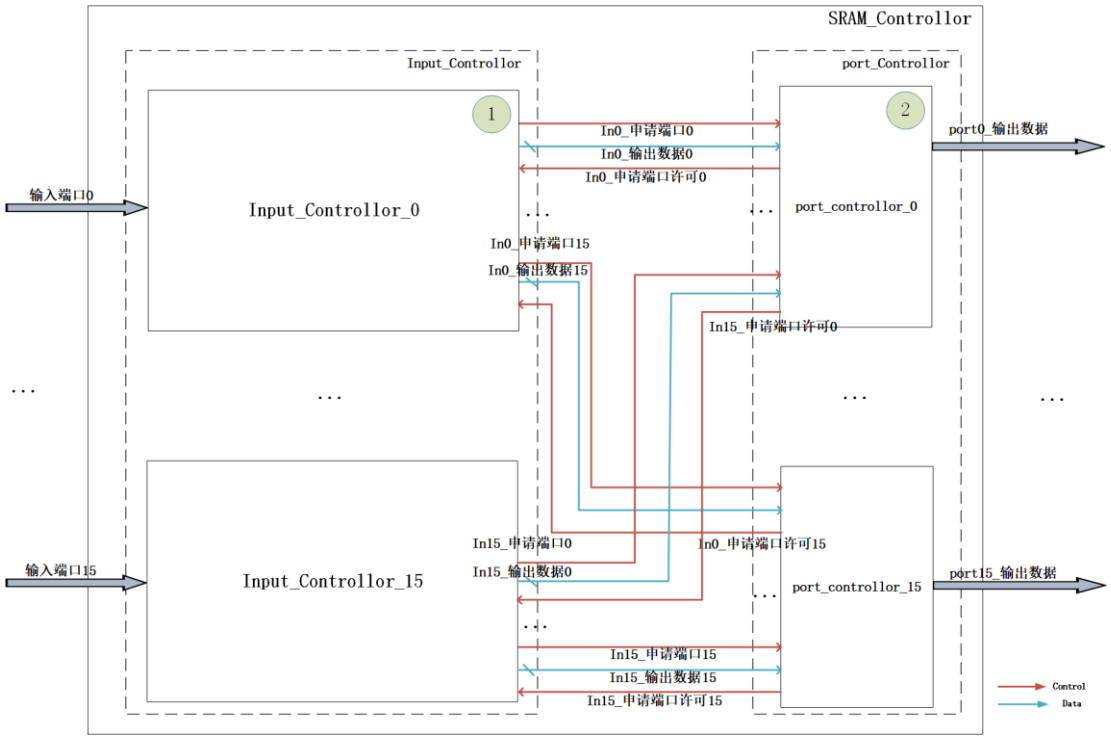


图 2.1.1 系统整体框图

系统整体上可以分为输入控制 Input_Controller 和端口控制 Port_Controller 两个部分。由于在 SRAM 的使用上，目前将 32 块 SRAM 进行了分割，划分为 16 组的 SRAM，并且端口 n(0<=n<=15)使用第 n 个组 SRAM，每个端口都不能使用其他组的 SRAM，因此每个端口的输入部分会进行的相对独立,可以将输入部分拆分为 16 个输入控制通道 Input_Controllor，如图中序号 1 所示的部分,每个输入控制通道的结构都比较类似。对于每个输出端口的管理，设计了一个端口控制模块 Port_Controller 来管理端口的状态，如图中序号 2 所示的部分。端口控制模块会对输入控制通道的请求进行仲裁，保证同一个时刻只能有一个 Input_Controllor 对本端口进行操作。每个端口控制模块的结构也类似。

下面将分别对 Input_Controller 模块和 Port_Controller 模块内部结构进

行介绍。

Input_Controller 模块包含了数据从端口输入到申请地址、写入 SRAM、记录数据到队列以及最后出队仲裁、请求的整个过程，其内部结构如下图 2.1.2 所示。

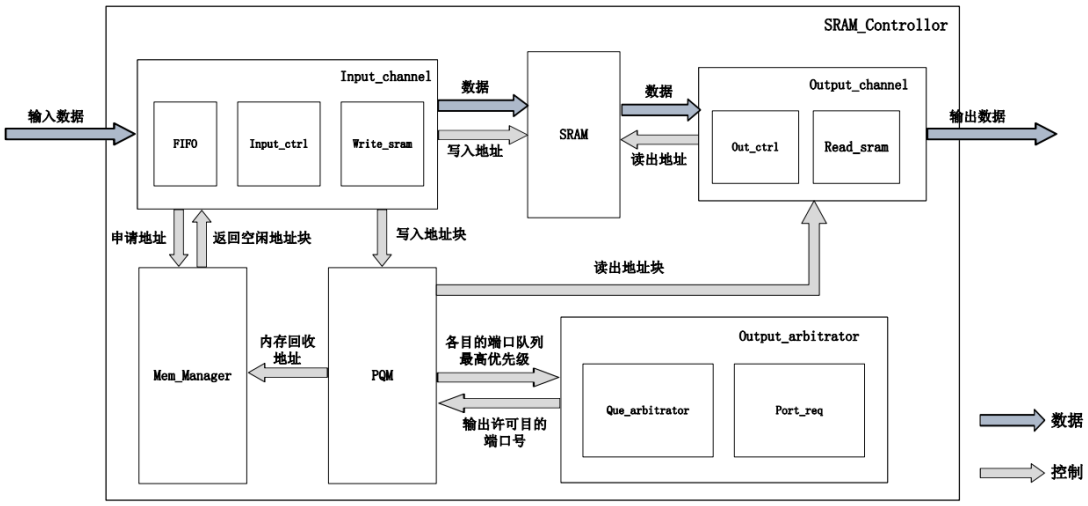


图 2.1.2 Input_Controller 模块内部结构

Input_Controller 模块包含了输入通道 Input_Channel、内存管理 Mem_Manager、队列管理 PQM、出队仲裁 Output_Arbitrator、输入通道 OutputChannel 等主要模块。

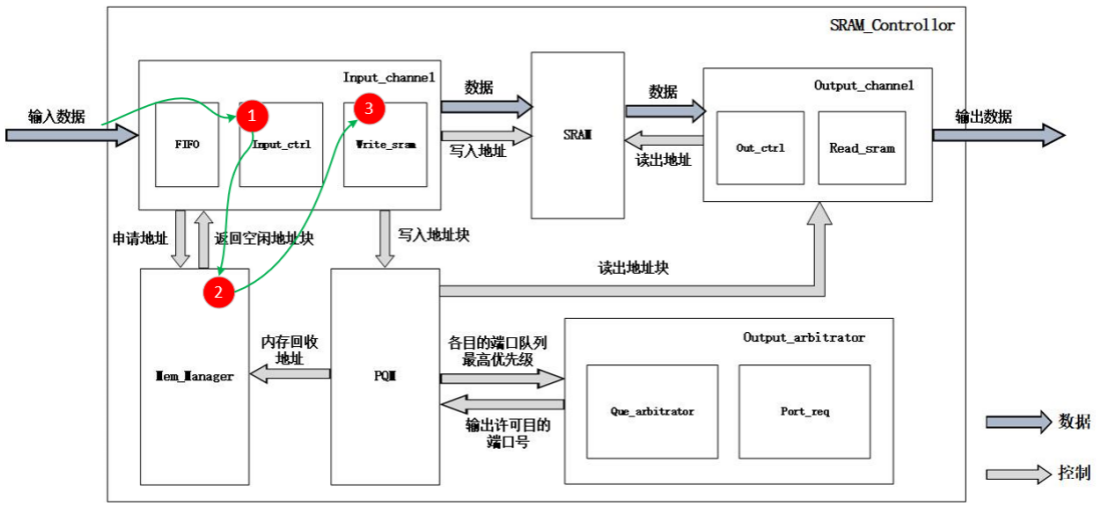


图 2.1.3 数据写入 SRAM 中的地址申请流程

进入端口的数据，会先在 FIFO 中进行暂存，其数据包头部数据会在进入 FIFO 的同时被送进 input_ctrl 模块。在这个模块中会解析数据的头部数据，将数据

包的目的地址 da、优先级 priority 信号告知 PQM 模块，同时会根据长度字段，计算需要请求的地址块个数(在地址管理上将 SRAM 进行分块，每 64 字节为一个块)，进而向内存管理模块 Mem_Manger 模块申请地址。申请的地址会告知 Input_ctrl 模块和 PQM 模块，Input_ctrl 模块根据地址控制 Write_sram 模块从 FIFO 中读取数据，进而写入到 SRAM 中，PQM 模块将该地址存入队列中，进行记录。

输出数据的处理流程如下图 2.1.4 所示。

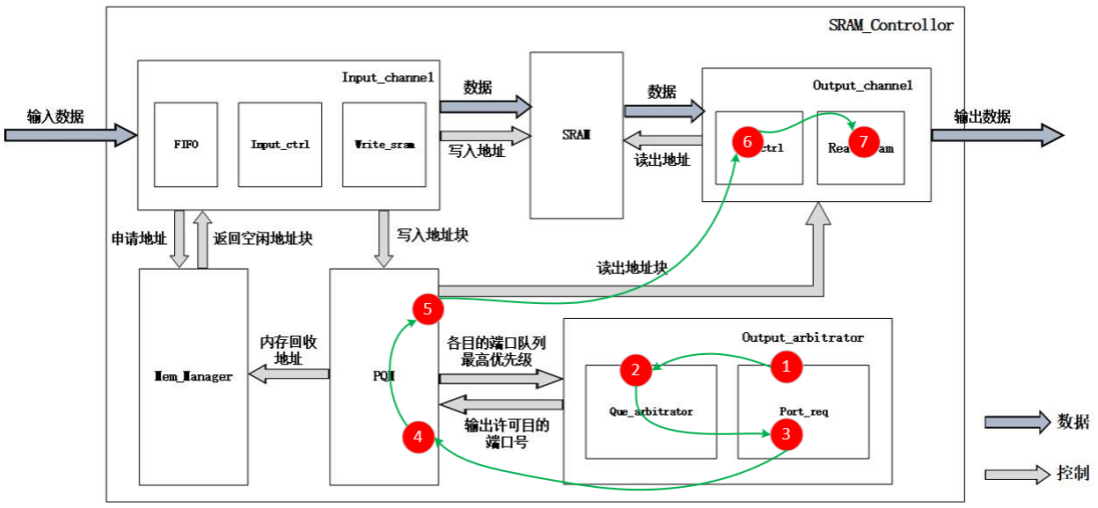


图 2.1.4 输出数据的处理流程

当 PQM 中的队列数据不为空并且 Port_rqe 模块自身处于空闲状态时，输出数据的流程就开始了。Port_req 模块首先使能 Que_arbitrator 模块，让其仲裁 PQM 当前的队列输出数据，从而得到一个当前最高优先级数据的目的地址信息，然后向这个目的端口发起请求，如果端口给予了响应，则将该目的地址信息，发送到 PQM 队列管理模块。PQM 队列管理模块会根据该目的地址信息，输出对应的 SRAM 存储块的地址数据到 Output_ctrl 模块中，进而驱动 Read_sram 模块读取对应地址块的数据输出。

Port_Controller 模块模块包含两个部分 Port_Arbitrator 端口仲裁和一个 16 选 1 的 MUX,其内部结构如下图 2.1.5 所示。

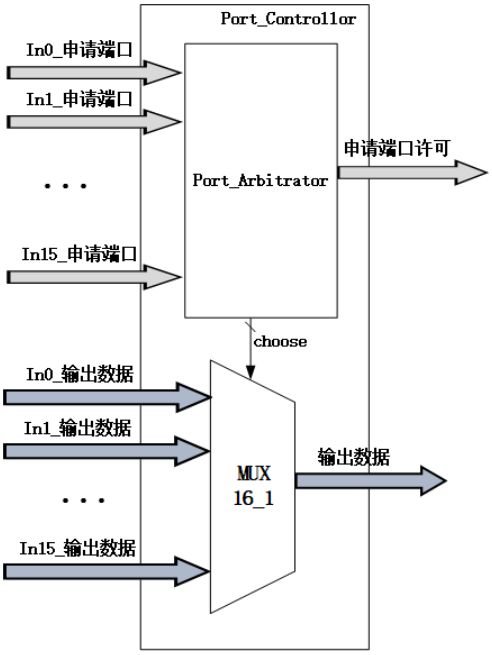


图 2.1.5 Port_Controller 模块内部结构

Port_Controller 内部的 Port_Arbitrator 端口仲裁模块会对输入控制通道的请求进行仲裁，保证同一个时刻只能有一个 Input_Controller 中的 Read_sram 模块对本端口进行操作。Port_Controller 内部的 MUX16_1 会根据仲裁的结果，选择对应输入数据进行输出。

2.2 数据位宽设计

根据赛题管理不少于 32 块 256Kbit 的 SRAM，即总容量不少于 8Mbit 的要求。本作品将 SRAM 的位宽定为 32bit，每个 SRAM 的深度为 8192。

输入输出端口的位宽设计为 32bit，接口时钟为 250MHz，每个端口的数据传输带宽为 8Gbps，满足赛题的 1Gbps 要求。

2.3 数据包格式设计

本作品的数据包格式，如下图 2.3.1 所示。

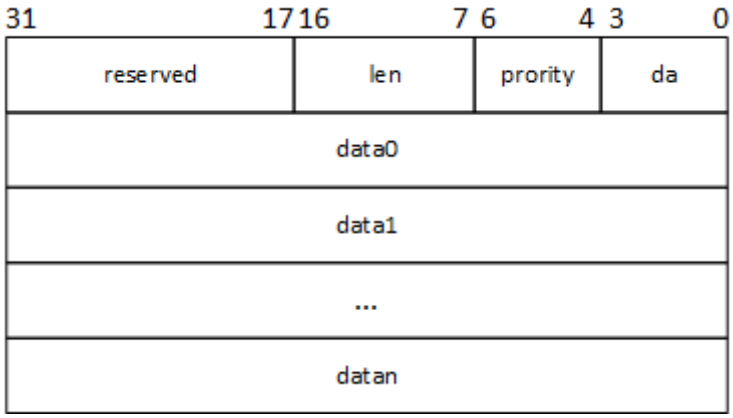


图 2.3.1 数据包格式

Len 字段为数据长度（64-1024Bytes），Priority 字段为数据包的优先级(0-7),DA 字段为数据包目的端口(0-15)，Reserved 字段暂时保留。

2.4 SRAM 选择与设计

本作品使用的 SRAM 是真双端口 SRAM，即两组端口同时进行读写操作。将要管理的 32 块 256kbit SRAM 进行分组，每组两块 SRAM。合并后，整个系统具有 16 个可以同时进行读写的 SRAM 组，每个输入端口管理一个 SRAM 组，即所有具有相同源地址的数据包会被放在同一个 SRAM 组内。SRAM 中两个完全独立的端口，即“a”和“b”端口，其中“a”端口用于写 SRAM，受读 SRAM 模块控制；“b”端口用于读 SRAM，受读 SRAM 模块控制。

在 SRAM 组内，对每个 SRAM 进行了分块处理，每个块 64 字节，后续的内存管理以一个块为基本管理单元，进行内存的分配与释放。

3 系统相关算法

3.1 优先级队列管理算法

优先级队列管理算法参考 PIEO，SP-PIFO 设计了一种链表算法管理队列，它能以较为简洁的链表形式创建队列，完成出队、入队操作，有助于减少运行时因多次查表引入的延迟和后续的硬件设计。

本设计为每一个输入端口建立一个队列，一个队列管理一组共享缓存，一个队列中包含 16x8 条子队列，每条子队列记录着特定输出端口的特定优先级的包信息，16x8 表示对于每个输出端口都有 8 条不同优先级的子队列。

优先级队列管理模块的算法模型如 图 3.1.1 所示，包括队列检索表和队列地址表。队列检索表中存储着子队列的首尾地址及是否创建队列标志位。队列检索表格式如表 3.1.1 所示。队列地址表中存有若干地址节点，地址节点由数据和下一跳地址构成，一组地址节点通过指针首尾相接，构成一条逻辑队列，队列地址表格式如表 3.1.2 所示。

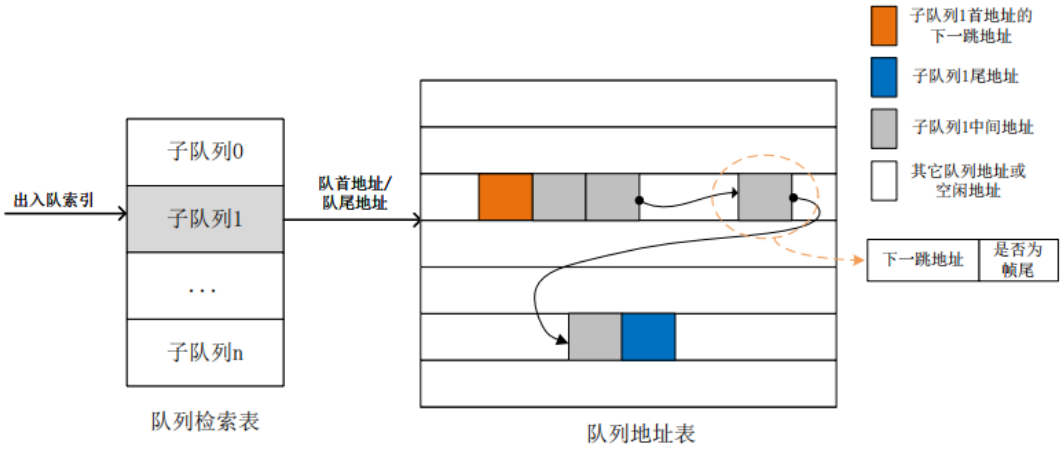


图 3.1.1 优先级队列管理模块算法模型图

表 3.1.1 队列检索表

子队列号	队首地址	队尾地址	是否创建队列
0	head_addr_0	tail_addr_0	vld0

1	head_addr_1	tail_addr_1	vld1
…	…	…	…
127	head_addr_127	tail_addr_127	vld127

表 3.1.2 队列地址表

地址	下一跳地址	是否为帧尾
0	next_addr_0	frame_last_0
1	next_addr_1	frame_last_1
…	…	…
n	next_addr_n	next_addr_n

出入队流程如下所示：

- **创建队列**

有报文入队时，input_ctrl 模块提取报头信息为索引，并从内存管理模块申请空闲地址送入 PQM，PQM 在队列检索表中检索子队列，若子队列创建标志位为 0，则表示该子队列未被创建，PQM 将地址写入此子队列的队首地址和队尾地址；当收到此条报文的下一个地址，PQM 将新收到的地址写入原队尾地址的下一跳地址内，并更新此子队列的队尾地址；直到收到此条报文的最后一个地址，PQM 同样需要将新收到的地址写入原队尾地址的下一跳地址内，并更新此子队列的队尾地址，另外，PQM 还要将子队列的创建标志位置为 1，将帧尾地址的是否为帧尾标志位置为 1。

- **入队**

根据报文信息携带的索引，在队列检索表中检索到指定子队列，若子队列创建标志位为 1，表示该子队列已被创建，此时 PQM 执行入队操作。将新收到的地址写入原队尾地址的下一跳地址内，并更新此子队列的队尾地址；当收到报文的最后一个地址，PQM 将新收到的地址写入原队尾地址的下一跳地址内，并更新此

子队列的队尾地址，同时将帧尾地址的是否为帧尾标志位置为 1。

- 出队

出队过程遵循刚优先级先出，同优先级先入先出的顺序。根据许可输出的端口号，找到此输出端口创建的最高优先级子队列，将子队列的首地址送至 read_sram 模块，待 read_sram 模块读完此地址块内的所有数据后，PQM 读取此地址的下一跳地址送入 read_sram 模块，直到帧尾标志位为 1 结束本次出队。

3.2 内存管理算法

内存管理算法参考了嵌入式 uCOS 常用的位图优先级算法，并在其基础上进行了改进和创新，以适用于基于寄存器的硬件平台。

位图优先级算法是多任务处理器在执行不同优先级的任务调度通过使用位图数据结构来表示各个任务的优先级，从而快速、高效地选择下一个要执行的任务。相比于传统的方法如从头到尾遍历就绪队列以及维护优先队列的顺序，按照优先级从高到低的顺序排列，位图优先级算法花费的时间和当前任务数无关，具有确定性，具有简单性和高效性。

在本次设计的内存管理算法中，用不同的比特位代表 SRAM 中某一个连续地址的实际存储区域，我们称之为块（Block），每个块的大小为 64Bytes，块的地址是其包含存储区的首地址。每个比特位在位图中的地址和 SRAM 中实际存储块地址相互映射，这样我们就可以用 N 比特的大小管理 N×64Bytes 的实际内存空间。

由于基于寄存器和组合逻辑电路的硬件结构和嵌入式处理器有很大区别，前者可以实现并行计算而后者不行，因此基于前者实现位图优先级查找的算法和基于 uCOS 上的位图优先级算法有很大区别。如下图 3.2.1 是本次设计的内存管理算法查找最高优先级空闲比特位的过程示意图：

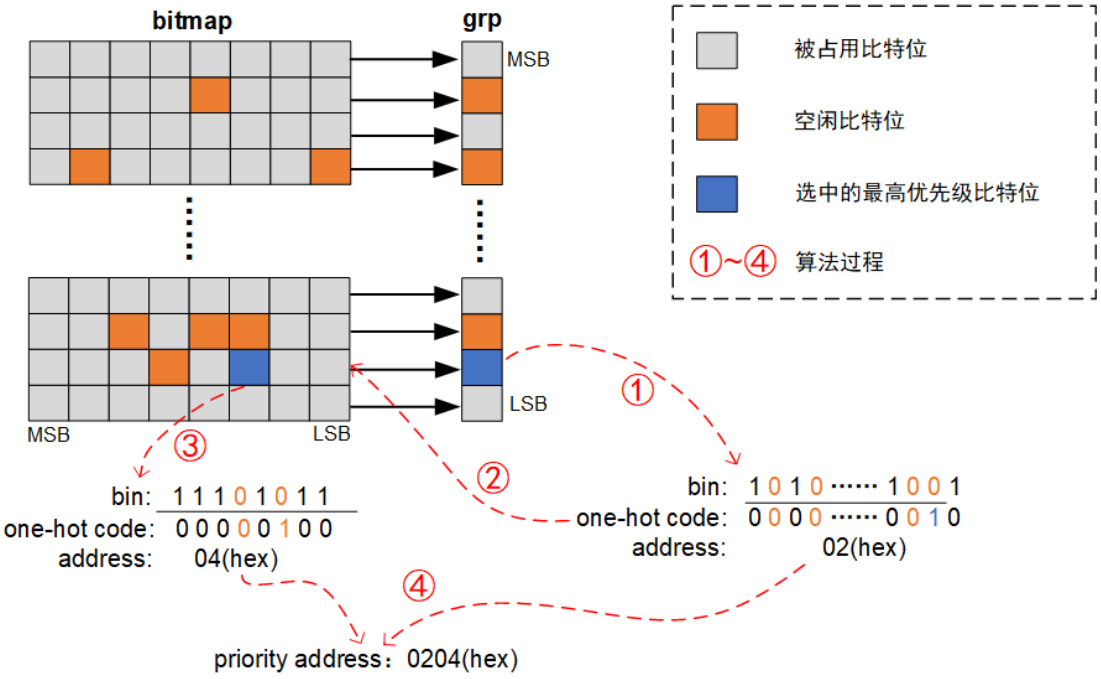


图 3.2.1 内存管理模块算法模型图

上图展示了该模块的工作过程：

- **位图（bitmap）表示任务状态**

图中左边的位图用于表示不同优先级内存块的状态。每个位置的比特位都对应了唯一的优先级，默认低位 LSB 优先级高于高位 MSB。灰色的方块表示该优先级比特位被占用（用 1 表示），橙色方块表示该优先级比特位是空闲的（用 0 表示）。位图中每一行对应一个优先级组。

- **优先级组选择（grp）**

位图中每个优先级组的状态是其中所有比特位求交集的结果，在右边的“grp”寄存器中的一个比特表示，grp 寄存器用来快速识别每一组中是否有空闲比特位。grp 的每一位表示对应行是否完全被占用。

- **查找最高优先级**

首先，通过扫描 grp 寄存器，找到存在空闲比特位的最高优先级的组（如图中步骤①所示）。在图中，grp 寄存器的值为 1010...0101，通过转化为独热码（one-hot）编码来找到的存在空闲比特位的最高优先级组（如图中步骤②所示），其中独热码置 1 的比特位为最高优先级的 0 位。然后，在对应的优先级组中查找最高优先级的比特位。

在这其中,基于 grp 和 grp 选中的最高优先级行的二进制数(bin,用 b 表示),将其转化成独热码（one-hot code,用 c 表示）的公式满足：

$$c = \overline{b} \bigcap (b + 1)$$

- **编码任务地址**

找到具体的最高优先级空闲比特位后，将其地址编码。首先将每组中的优先级 one-hot 编码表示出来（如图中步骤③所示）。最后，将组地址和行优先级地址组合成最终的优先级地址（如图中步骤④所示）。在图中，组合结果为 0204(hex)，即最高优先级空闲比特位的地址。

通过这个过程，该电路模块能够快速、有效地找到当前最高优先级的任务并进行调度。这种方法通过位图和优先级组寄存器的结合，大大提高了内存查找和分配的效率。

3.3 队列输出仲裁算法

从输入端口进入的数据会进行入队处理，出队需要根据优先级信息进行仲裁。队列管理模块内部是按照目的端口+优先级方式进行存储数据，存储结构如下图 3.3.1 所示。

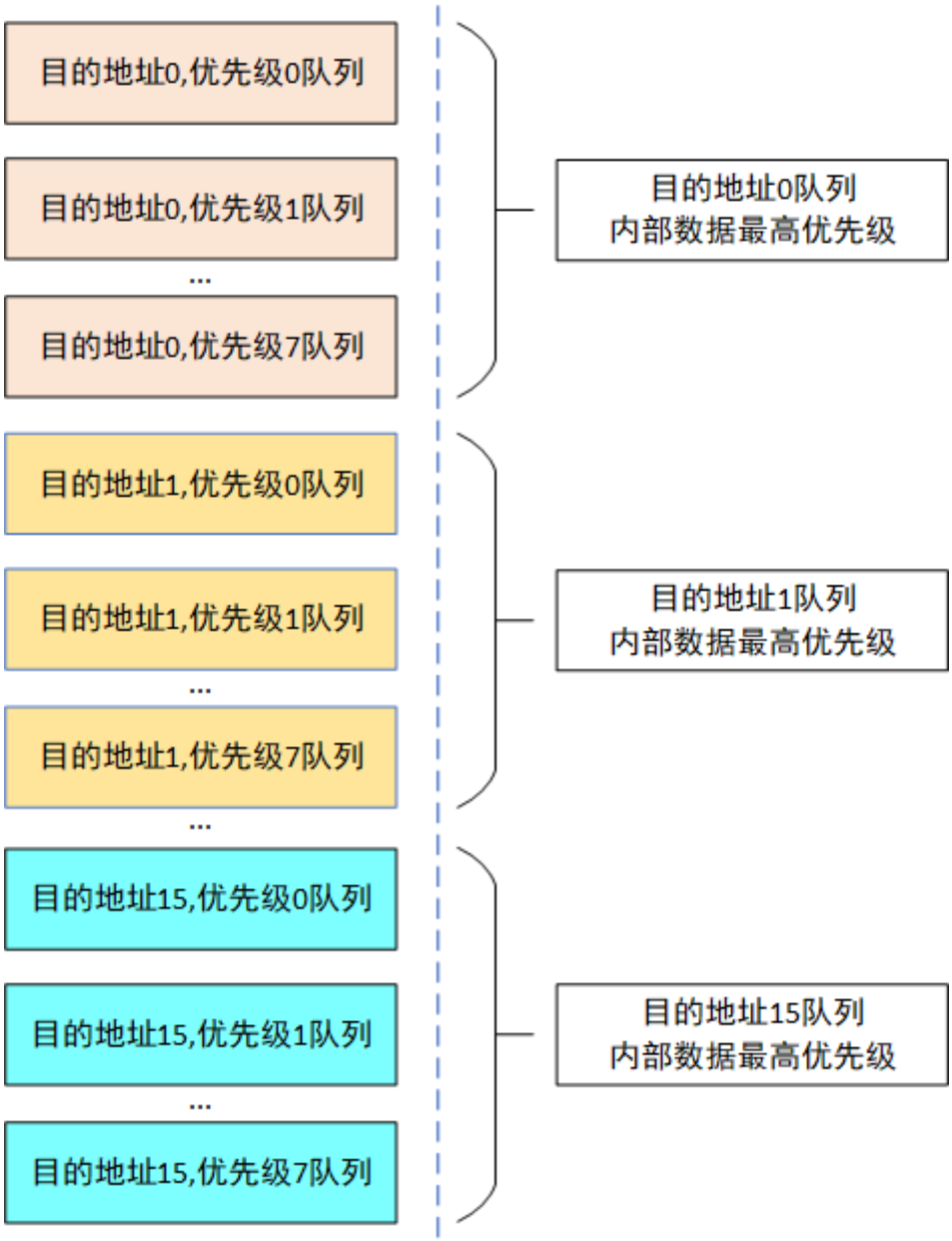


图 3.3.1 队列管理模块存储结构示意图

在队列管理模块内部会将相同目的端口、相同优先级的数据存储到同一个队列中，内部总共有 16x8 个队列。同时，队列管理模块内部维护了相同目的地址的不同优先级的 8 个队列中，当前优先级最高的、有数据的队列标号，总共会有 16 个优先级标号数据，以及 16 个数据是否有效的 vld 信号，该数据会送进仲裁模块。

因此，仲裁模块的任务可以详细描述为，现在有 16 个 3bit 的数据，以及这些数据的有效信号，需要从这些数据中选择出一个数值最大的数据(这里使用数值越大优先级越大的规定)。

首先将输入的数据逐 bit 与对应的 vld 信号进行与操作，例如：输入信号为 d[2],d[1],d[0]，对应的 vld 信号为 0，与操作之后为 0,0,0。消除无效信号的影响。之后，将与操作之后的数据进行独热编码，并将编码后的数据与对应的 vld 信号进行合并处理，然后按照行排列，具体结果如下图 3.3.2 所示。

行号	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	vld
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
2									
3									
4	0	1	0	0	0	0	0	0	1
5	1	0	0	0	0	0	0	0	1
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									

图 3.3.2 输出仲裁模块数据排列示意图

首先对每一列进行或操作，即判断每一列中是否有 1 存在，如果存在则表明存在对应列号优先级的数据，然后找出每一个列中从小序号到大序号的遍历的第一个 1 所在的行号(这里使用的策略，相同优先级的情况下小端口号的优先于大端口号)，同时需要记录该行号，即端口号。

经过上一步的处理，已经得到了各个优先级的一个端口号，此时只需要找出存在的最大优先级，然后读出上一步骤记录的对应的端口号。如果此时最高有效的优先级是 0，则从端口号较小的端口开始，找到第一个 vld 信号有效的端口。

4 子模块设计

4.1 DPRAM 模块

真双端口 RAM （DPRAM）指的是有两个读写端口，每个端口都可以同时发起读或者写，且彼此之间没有干扰的 RAM 设计。真双端口 RAM 有两组数据线和地址线，以及两个输出端口。因此，两个端口可以同时进行读写操作，无论是读和读、写和读，还是写和写，都可以在同一个时刻进行。

本作品需要管理至少 32 块 256Kbit 的 SRAM, 目前设定为 32 块。将 32 块 SRAM 分成 16 组，每组 2 个 SRAM，称为一个 DPRAM 块。DPRAM 模块的框图如下图 4.1.1 所示。

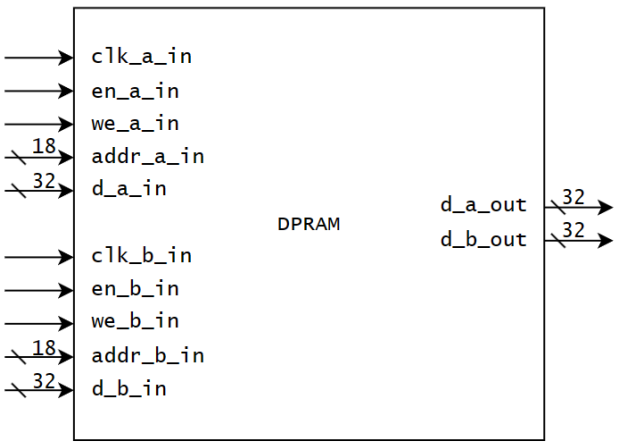


图 4.1.1 DPRAM 模块框图

DPRAM 模块的端口信号的说明如下表 4. 1. 1 所示。

表 4.1.1　DPRAM 模块的端口信号说明

端口名	方向	位宽	端口说明
en_a_in	输入	1b	端口 A 使能信号，高有效
we_a_in	输入	1b	端口 A 写使能信号，高有效
addr_a_in	输入	13b	端口 A 的 RAM 地址
d_a_in	输入	32b	端口 A 的写数据端口
clk_a_in	输入	1b	端口 A 的时钟信号
d_a_out	输出	32b	端口 A 的读数据端口

en_b_in	输入	1b	端口 B 使能信号，高有效
we_b_in	输入	1b	端口 B 写使能信号，高有效
addr_b_in	输入	13b	端口 B 的 RAM 地址
d_b_in	输入	32b	端口 B 的写数据端口
clk_b_in	输入	1b	端口 B 的时钟信号
d_b_out	输出	32b	端口 B 的读数据端口

4.2 输入控制模块(input_ctrl)

输入控制模块（input_ctrl）通过有限状态机来管理不同条件下的数据流，并根据不同的状态执行相应的操作，以控制从输入信号到 SRAM 的写入过程。input_ctrl 模块的框图如下图 2.4 所示。

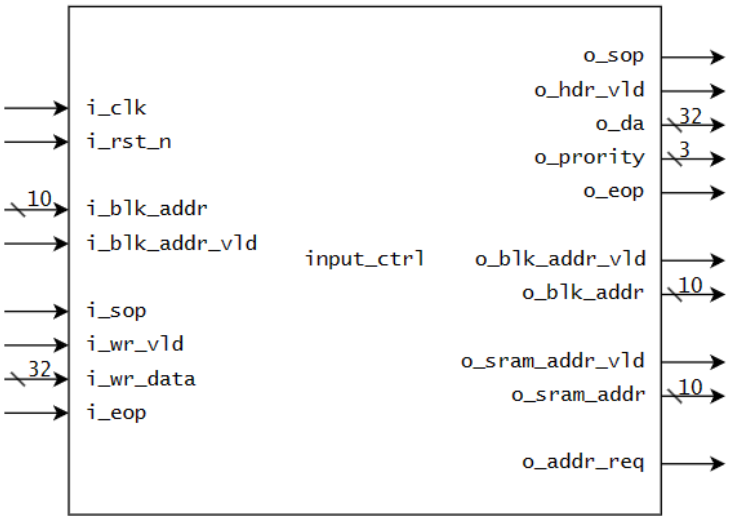


图 4.2.1 input_ctrl 模块框图

模块的接口描述如下表 4. 2. 1 所示：

表 4.2.1 input_ctrl 模块的端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_sop	输入	1b	新的一帧数据开始标志

i_wr_vld	输入	1b	写有效信号
i_wr_data	输入	32b	写入数据
i_eop	输出	1b	帧数据结束标志
i_blk_addr	输入	14b	存储数据的 block 地址
i_blk_addr_vld	输入	1b	存储数据的 block 地址输入有效
o_sop	输出	1b	输出帧开始标志
o_da	输出	4b	目的地址
o_prority	输出	3b	输出优先级
o_hdr_vld	输出	1b	输出 priority，da 有效
o_blk_addr	输出	14b	输出 block 地址
o_blk_addr_vld	输出	1b	输出 block 有效
o_eop	输出	1b	输出帧结束标志
o_sram_addr	输出	14b	SRAM 存储地址
o_sram_addr_vld	输出	1b	SRAM 存储地址有效
o_addr_req	输出	1b	向内存管理模块申请地址请求

输入控制模块和其他模块的连接如下图 4.2.2 所示

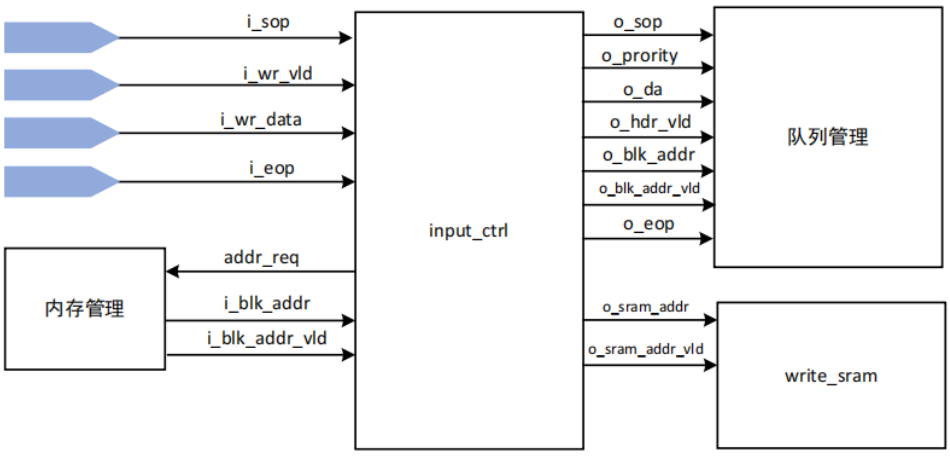


图 4.2.2 输入控制模块与其他模块的连接

输入控制模块主要实现以下三个部分的功能：

首先，当检测到端口有数据输入，向内存管理模块申请存储地址，申请的地址是 block 地址，然后把 block 地址转化为存储每一拍数据的实际 SRAM 地址；然后，解析帧头，讲解析得到的每个帧的目的地址(o_da)，优先级(o_priority)

以及申请到的存储地址发送给队列管理模块进行管理；最后，和写 SRAM 模块连接，把申请到的 SRAM 地址（o_sram_addr_vld）输入写 SRAM 模块。

输入控制模块内存状态机如下图 4.2.3 所示。

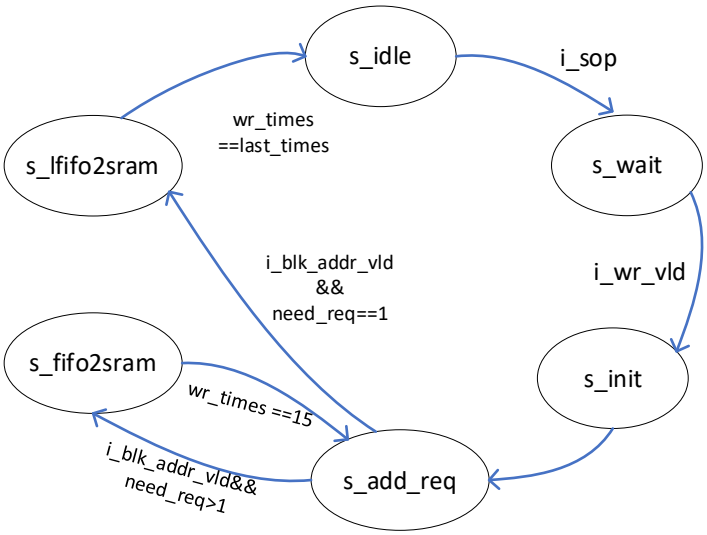


图 4.2.3 输入控制模块状态机

4.3 写 SRAM 模块(write_sram)

写 SRAM 模块(write_sram)根据输入信号 i_sram_addr_vld 决定是否从 FIFO 读取数据，并将数据写入指定的 SRAM 地址。

这个模块主要实现了以下功能：

1) 当输入的 SRAM 地址有效时，使能 FIFO 读取操作；

2) 将从 FIFO 读取的数据直接写入 SRAM；

3) 在时钟的上升沿或复位信号为低时，同步更新 SRAM 地址和写使能信号。

写 SRAM 模块的框图如图 4.3.1 所示。

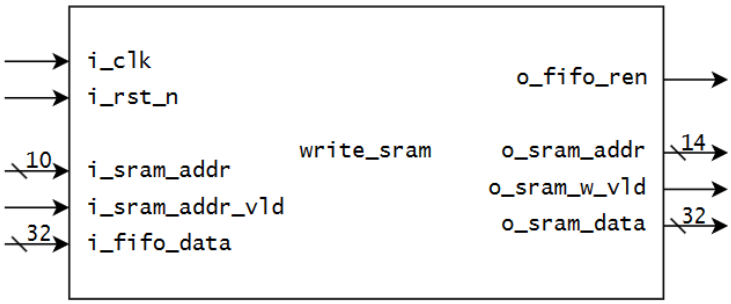


图 4.3.1 写 SRAM 模块框图

写 SRAM 模块的接口如下表 4.3.1 所示。

表 4.3.1 写 SRAM 模块的端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_sram_addr	输入	14b	输入 SRAM 地址
i_sram_addr_vld	输入	1b	输入 SRAM 地址有效
i_fifo_data	输入	32b	从 FIFO 读取的数据
o_fifo_ren	输出	1b	FIFO 读使能信号
o_sram_addr	输出	14b	输出写入 SRAM 地址
o_sram_w_vld	输出	1b	输出写入 SRAM 写使能
o_sram_data	输出	32b	输出写入 SRAM 数据

4.4 内存管理模块(mem_manager)

内存管理模块（mem_manager）是一个管理 SRAM 空间的模块，对 SRAM 内存空间进行分块（block）管理，采用位图的方法管理每个 block 从而高效实现内存灵活管理。其中包含了占用和释放 ROM 块的逻辑以及相应的状态机。

内存管理模块的框图如图 4.4.1 所示。

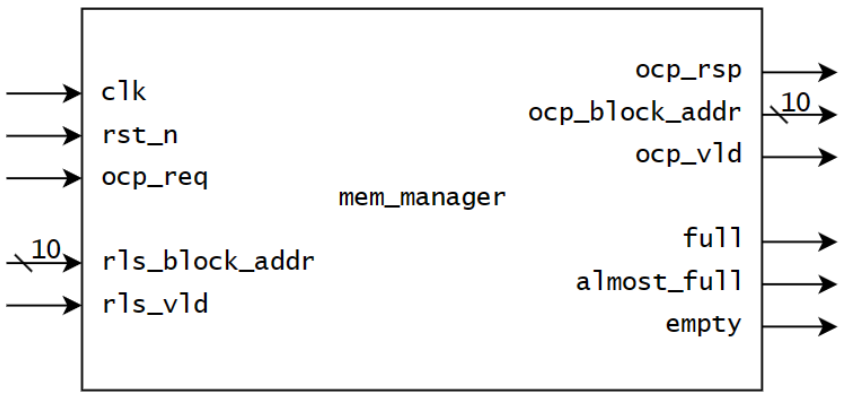


图 4.4.1 内存管理模块框图

内存管理模块的接口如下表 4.4.1 所示：

表 4.4.1 内存管理模块的端口信号说明

端口名	方向	位宽	端口说明
clk	输入	1b	时钟驱动信号
rst_n	输入	1b	同步复位信号（低电平复位）
ocp_req	输入	1b	请求占用 SRAM，申请可用 SRAM block 地址
ocp_rsp	输入	1b	响应占用 SRAM 请求，如果拉高表示成功
ocp_block_addr	输出	10b	输出可用 SRAM block 地址
ocp_vld	输出	1b	输出可用 SRAM block 地址有效信号
rls_block_addr	输入	10b	释放 block 地址
rls_vld	输入	1b	释放有效信号
full	输出	1b	SRAM 满信号
almost_full	输出	1b	SRAM 几乎满信号（阈值可配置）
empty	输出	1b	SRAM 空信号

内存管理模块的内部结构如下图 4.4.2 所示。

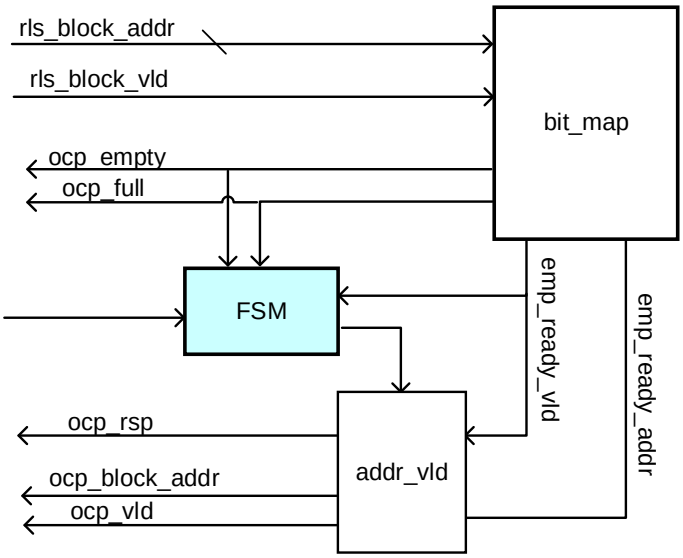


图 4.4.2 内存管理模块的内部结构图

对于每个 256Kb 大小的 SRAM 模块进行分块（block），每个 block 大小为 64bytes，最大可以存储下 16 拍来自端口的 32bit 位宽的数据。其中最主要的模块是 bit-map 模块，即位图管理器模块；通过位图优先级算法，通过查找第一个空闲位来实现空闲块的分配，并通过写使能信号和写值来更新内存。同时，该模

块还提供了空、满和几乎满的标志信号，以便于管理内存的状态。

下图 4.4.3 为 bit-map 模块内部架构图。

基于寄存器模块存储

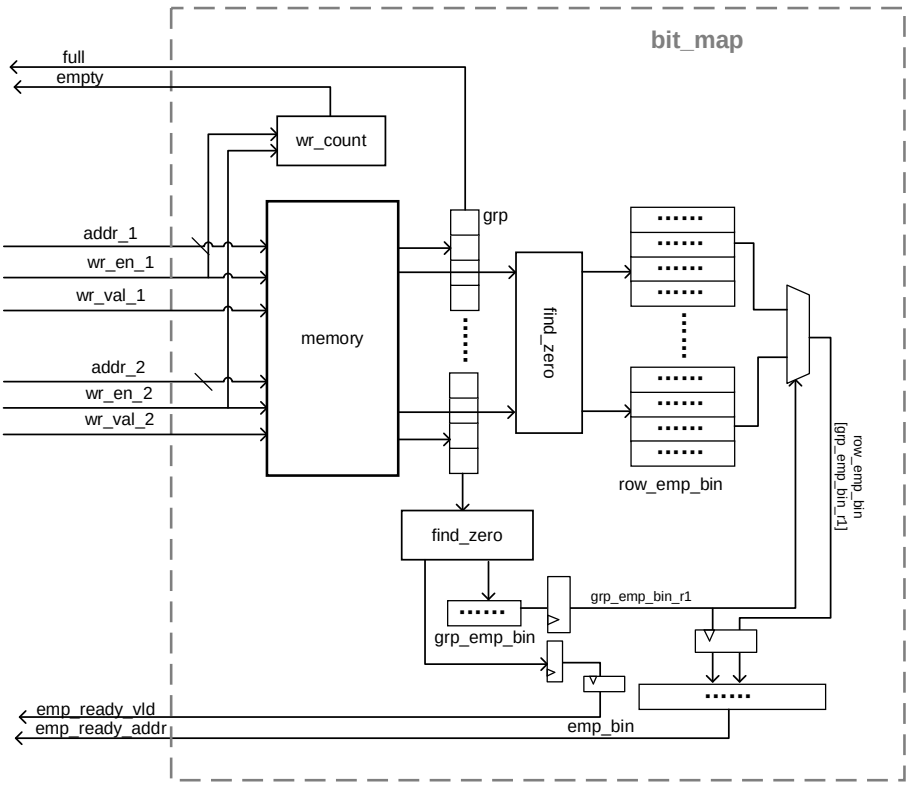


图 4.4.3 bit-map 模块内部架构图

4.5 优先级队列管理模块（PQM）

优先级队列管理（Priority Queue Manager,PQM）完成对报文信息的多优先级队列管理，优先级队列管理模块内部有两级管理，第一级为队列检索表，存储着子队列，即逻辑队列的首尾地址，第二级为队列地址表，存储着逻辑队列的所有地址。使用此结构有助于减少运行时因多次查表引入的延迟和硬件设计。

优先级队列管理模块的框图如下图 4.5.1 所示。

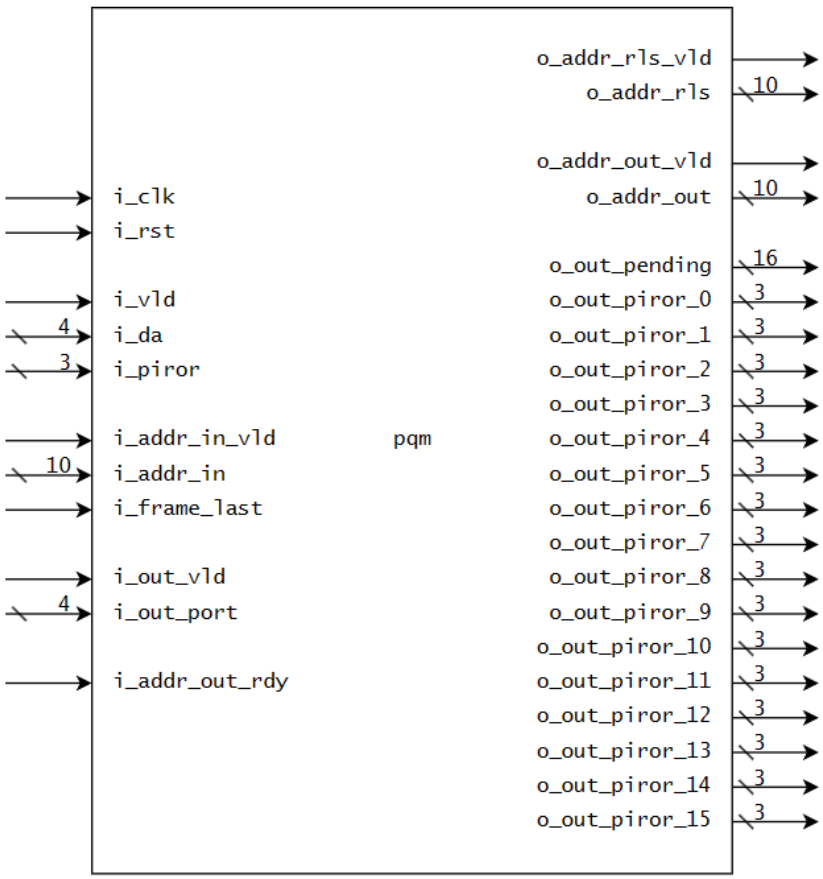


图 4.5.1 优先级队列管理模块框图

优先级队列管理模块的接口如下表 4.5.1 所示。

表 4.5.1 优先级管理模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_vld	输入	1b	输入目的地址和数据优先级有效信号
i_da	输入	4b	输入目的地址
i_prior	输入	3b	输入数据优先级
i_addr_in_vld	输入	1b	输入地址有效信号
o_addr_in_rdy	输出	1b	输入地址准备信号
i_addr_in	输入	10b	输入地址
i_frame_last	输入	1b	输入一帧最后一个地址信号
o_addr_rls_vld	输出	1b	释放地址有效信号

o_addr_rls	输出	10b	释放地址
i_addr_out_rdy	输入	1b	输出地址准备信号
o_addr_out_vld	输出	1b	输出地址有效信号
o_addr_out	输出	10b	输出地址
o_frame_last	输出	1b	输出一帧最后一个地址
i_out_vld	输入	1b	输出端口许可信号
i_out_port	输入	4b	输出端口
o_out_pending	输出	16b	对 16 个输出端口的排队信号
o_out_prior_{x}	输出	3b	对 16 个输出端口的数据最高优先级 x:1~16

优先级队列管理模块的内部结构如下图所示。

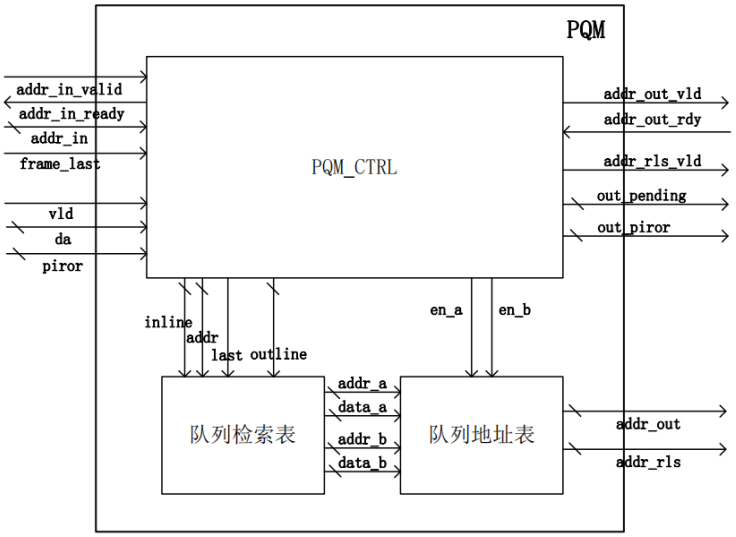


图 4.5.2　优先级队列管理模块内部结构

优先级队列管理模块内部包含三个子模块 PQM_CTRL、队列检索表和队列地址表。PQM_CTRL 模块控制队列的创建与更新。队列检索表模块记录一个队列，本设计为每个输出端口的每种优先级包都创建了一条独立的子队列，即该队列中包含 128（16x8）条子队列，每条子队列记录了该子队列是否被创建以及逻辑队列的首尾地址。队列地址表存储着逻辑队列，一条子队列对应着一条逻辑队列，逻辑队列记录着特定输出端口特定优先级的数据包的所有地址。

优先级队列管理的入队状态机如下图所示。

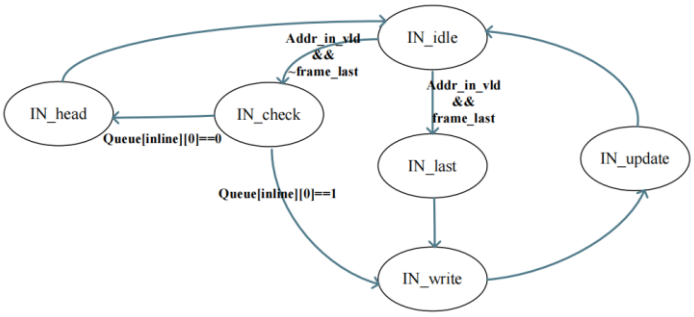


图 4.5.3 PQM **队列输入状态机**

每当输入控制模块收到新帧时，会向 PQM 发送此帧的目的端口和优先级，PQM 将目的端口和优先级存入寄存器，直到收到下一帧数据更新寄存器。基于此，当 PQM 收到新地址时，根据此时的目的端口和优先级寄存器确定此地址应该记录在哪条子队列，首先判断该条子队列是否被创建，若没有，则将此地址写入子队列的队首和队尾然后返回空闲状态；若已被创建，则先将地址写入原队尾地址的下一跳地址，然后更新队尾地址，再返回空闲状态。当收到此帧的最后一个地址时，在逻辑队列中将此地址的帧尾标志位置为 1，然后把此地址写入原队尾地址的下一跳地址，最后更新队尾地址返回空闲状态。

优先级队列管理的出队状态机如下图所示：

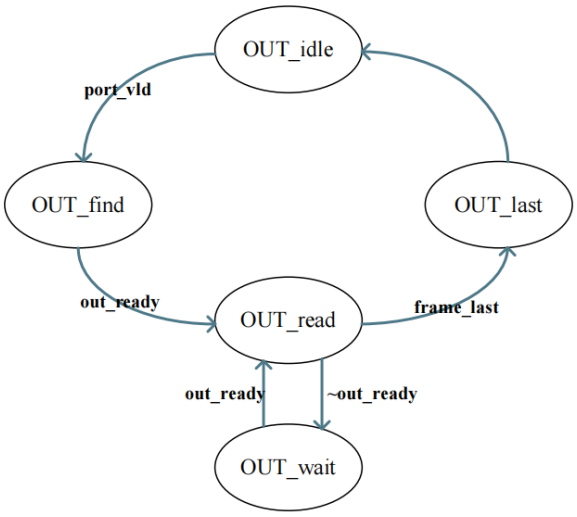


图 4.5.4　PQM **队列输出状态机**

当 PQM 收到输出使能信号时，首先根据输出端口号找到此输出端口的最高优先级队列，将此队列的队首地址发送出去，然后更新队首地址，直到发出帧尾地址，结束本轮出队回到空闲状态。

4.6 出队仲裁模块（que_arbitrator）

出队仲裁模块的框图如图所示。

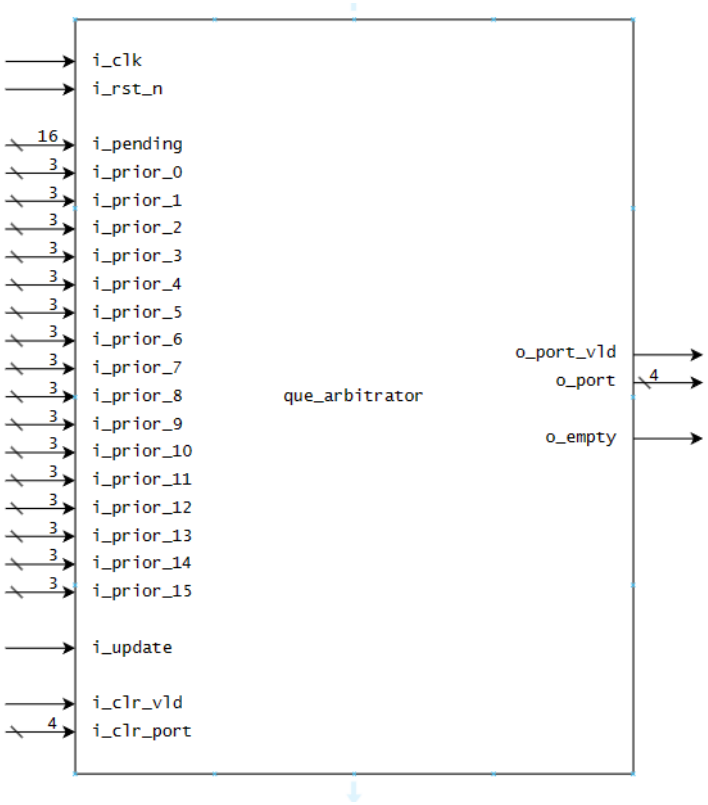


图 4.6.1 出队仲裁模块框图

出队仲裁模块的接口如下表所示：

表 4.6.1 出队仲裁模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_pending	输入	16b	端口挂起标志，每个比特代表一个端口
i_piror × 16	输入	4b	每个端口的优先级
i_update	输入	1b	更新信号
i_clr_port	输入	4b	需要清除的端口编号
i_clr_vld	输出	1b	清除有效标志
o_port	输出	4b	被仲裁选中的端口编号

o_port_vld	输出	1b	端口有效信号
o_empty	输出	1b	空标志，如果所有端口都没有挂起则置高电平

4.7 输出请求模块（port_req）

输出请求模块的框图如图 4.7.1 所示。

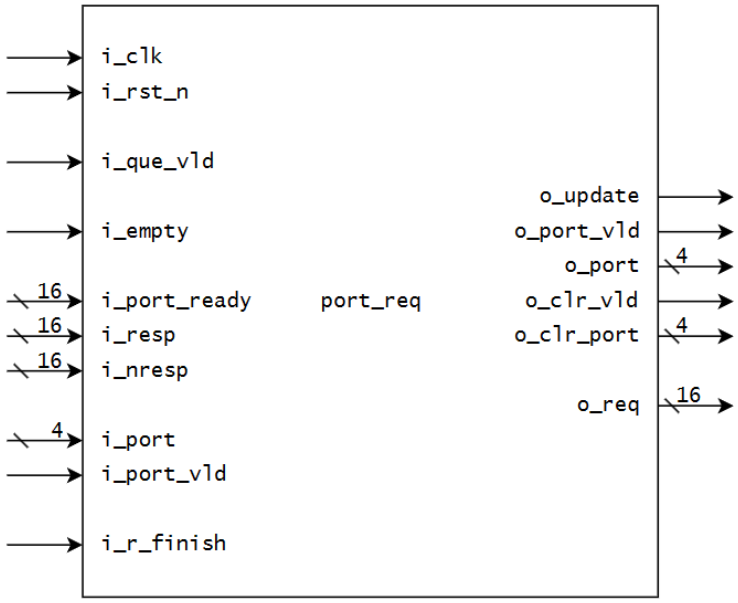


图 4.7.1 输出请求模块框图

输出请求模块的端口描述如下表所示：

表 4.7.1 输出请求模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_que_vld	输入	1b	队列有效信号
i_empty	输入	1b	空信号
i_port_ready	输入	16b	每个端口的准备状态，每个比特对应相应端口
i_resp	输入	16b	每个端口的响应信号
i_nresp	输入	16b	每个端口的非响应信号

i_port	输入	4b	被仲裁选中的端口信号
i_port_vld	输入	1b	被仲裁选中的端口信号有效标志
i_r_finish	输入	1b	读操作完成信号
o_update	输出	1b	更新信号
o_port	输出	4b	输出端口编号
o_port_vld	输出	1b	输出端口编号有效
o_clr_port	输出	4b	需要清除的端口编号
o_clr_vld	输出	1b	清除有效信号
o_req	输出	16b	请求信号，向各个的端口发送请求

4.8 输出控制模块(output_ctrl)

输出控制模块的框图如图 4.8.1 所示。

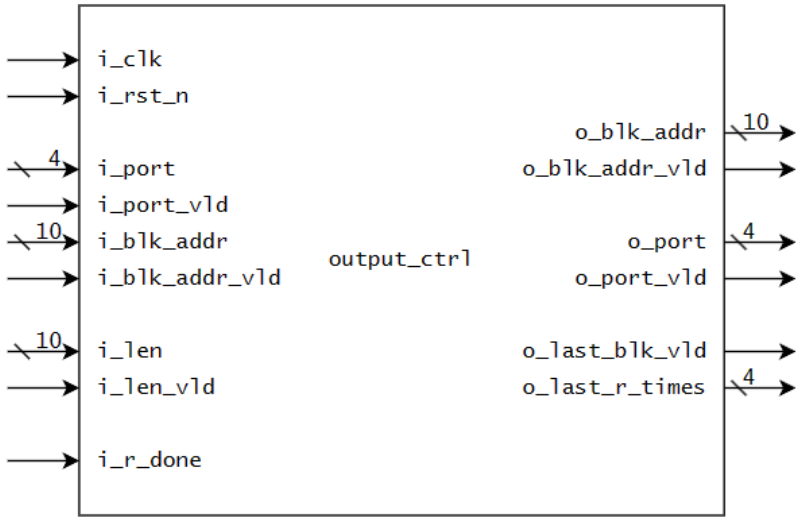


图 4.8.1 输出控制模块框图

输出控制模块的接口如下表所示：

表 4.8.1 输出控制模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）

i_port	输入	4b	用于选择输入端口号编号
i_port_vld	输入	14b	输入端口有效信号
i_blk_addr	输入	1b	用于输入 SRAM 存储块 block 地址
i_blk_addr_vld	输入	10b	block 地址有效信号
i_len	输入	10b	记录帧长度
i_len_vld	输出	1b	帧长度有效信号
i_r_done	输出	1b	读取完成信号
o_blk_addr	输出	10b	用于输出 SRAM 存储块 block 地址
i_blk_addr_vld	输入	1b	输出 block 地址有效信号
o_port	输出	1b	用于选择输出端口号编号
o_port_vld	输出	1b	输出端口有效信号
o_last_blk_vld	输出	1b	每个帧末尾（小于 64bytes 部分）存储 SRAM 存储块 block 地址
o_last_r_times	输出	4b	每个帧末尾具体占传输位宽的个数； 例如：帧末尾有 32bytes，不满足 64bytes 却也要占用一个大小为 64bytes 的 block 存储区，传输位宽为 32bit，则 32bytes=8×32bit，大小为 8

4.9 读 SRAM 模块（read_sram）

读 SRAM 模块（read_sram） 负责从内存 SRAM 读取数据，并通过 FIFO 缓冲区将数据输出到指定的接口。同时，其中包括对数据块和单元的管理以及对帧头和 CRC 校验的处理。

读 SRAM 模块的框图如图 4.9.1 所示。

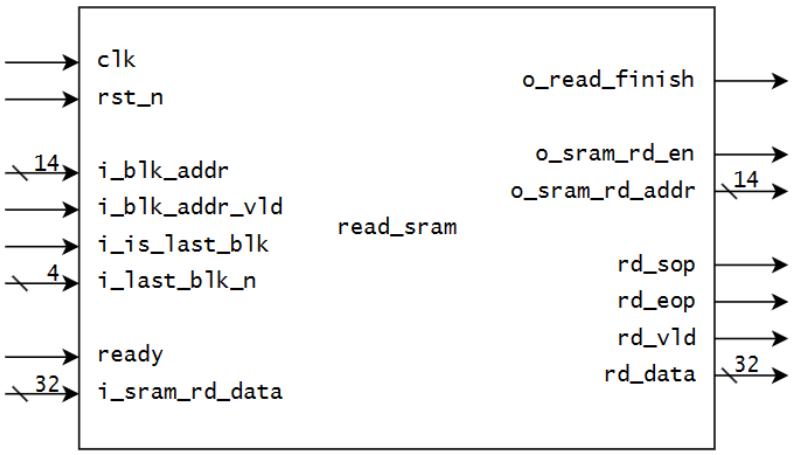


图 4.9.1 读 SRAM 模块框图

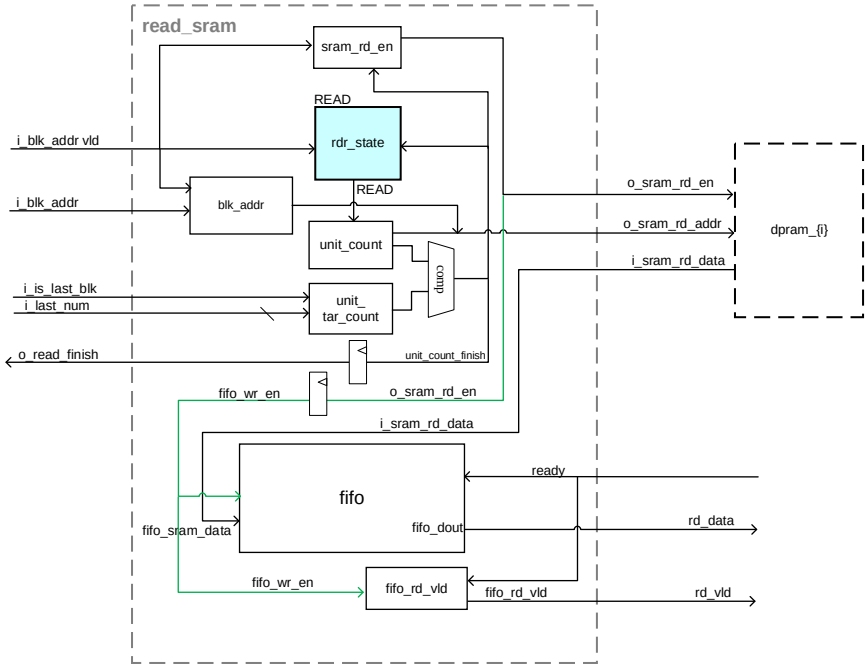
读 SRAM 模块的接口如下表所示：

表 4.9.1 读 SRAM 模块端口信号说明

端口名	方向	位宽	端口说明
clk	输入	1b	时钟驱动信号
rst_n	输入	1b	同步复位信号（低电平复位）
i_blk_addr	输入	1b	输入读取 SRAM block 地址
i_blk_addr_vld	输入	14b	输入读取 SRAM block 地址有效
i_is_last_blk	输入	1b	是否 i_blk_addr_vld 为最后一个 block
i_last_blk_n	输入	4b	最后一个 block 中存储的有效数据深度
i_start_packet	输入	1b	是否为帧头
o_read_finish	输出	1b	读取 SRAM 完成信号
o_sram_rd_en	输出	1b	读取 SRAM 使能
o_sram_rd_addr	输出	14b	读取 SRAM 地址
i_sram_rd_data	输入	32b	输入 SRAM 读取数据
rd_sop	输出	1b	输出帧起始信号
rd_eop	输出	1b	输出帧结束信号
rd_vld	输出	1b	输出数据有效
rd_data	输出	32b	输出数据
phead	输出	32b	帧头信息

phead_vld	输出	1b	帧头信息有效信号
crc_correct	输出	1b	crc 校验正确信息
crc_correct_vld	输出	1b	crc 校验正确有效信号

读 SRAM 模块的结构如下所示



4.10 输出仲裁模块（port_arbitrator）

输出仲裁模块需要对输出请求模块的请求进行仲裁，保证同一个时刻只能有一个输入端口在向本端口写数据。输出仲裁模块的框图如图所示。

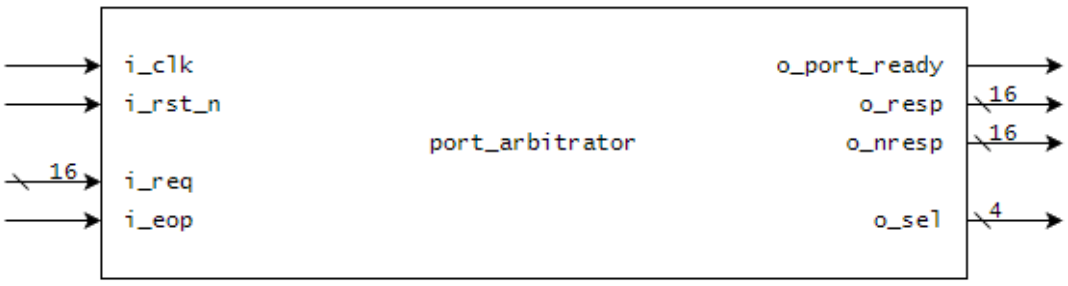


图 4.10.1 输出仲裁模块框图

输出仲裁模块的接口如下表 4.10.1 所示：

表 4.10.1 输出仲裁模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_req	输入	16b	16 个输入通道的请求信号
i_eop	输入	1b	输入本端口的 eop 信号
o_port_ready	输出	1b	本端口的 ready 信号
o_resp	输出	16b	仲裁的响应结果
o_nresp	输出	16b	仲裁的拒绝结果
o_sel	输出	4b	仲裁的响应的端口标号，给输出的 mux 使用。

输出仲裁模块的结构比较简单，内部实际上使用了一个优先编码器。输出仲裁模块状态机如下图 4.10.2 所示。

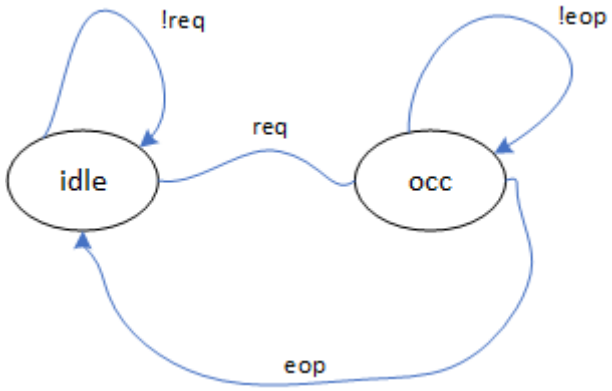


图 4.10.2 输出仲裁模块状态机

输出仲裁模块只有两个状态空闲态 idle,以及占用状态 occupied(简写为 occ)。在 idle 状态，如果有请求数据就会进行仲裁，仲裁可以在一个周期内完成，因此下一个状态会进 occ 状态。在 ooc 状态下，只有接收到 eop 信号，即本端口的数据写入已经完成标志，之后会重新进入 idle 状态，等待请求。

4.11 CRC 校验模块（crc32_d32）

CRC 校验模块，用来对存入 SRAM 的数据进行校验，数据存储 SRAM 之前计算其 CRC 校验值，数据读出 SRAM 后进行 CRC 校验。CRC 校验模块的框图如图所示。

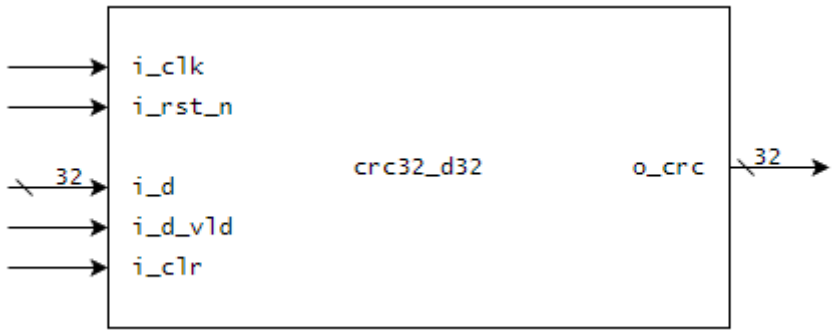


图 4.11.1 CRC 校验模块框图

CRC 校验模块的接口如下表所示：

表 4.11.1 输出仲裁模块端口信号说明

端口名	方向	位宽	端口说明
i_clk	输入	1b	时钟驱动信号
i_rst_n	输入	1b	同步复位信号（低电平复位）
i_d	输入	32b	输入的待计算 CRC 的数据
i_d_vld	输入	1b	输入数据有效信号
i_clr	输入	1b	结束本次计算，清除 crc32_d32 内部寄存器。
o_crc	输出	32b	计算的 crc32 校验值

CRC 校验模块的输入输出位宽为 32bit，与本作品的输入输出端口位宽以及 SRAM 读写位宽相匹配，可以方便的实现数据校验功能。输入的数据在送进 SRAM 存储的同时也会送进 CRC 校验模块计算 CRC 校验值，输出的数据在输出的同时，也会对其进行校验。

5 验证平台

5.1 验证平台整体设计

本作品搭建了基于受约束随机化策略的 SystemVeriog 分层验证平台，验证平台系统框图如下图 5.1.1 所示。

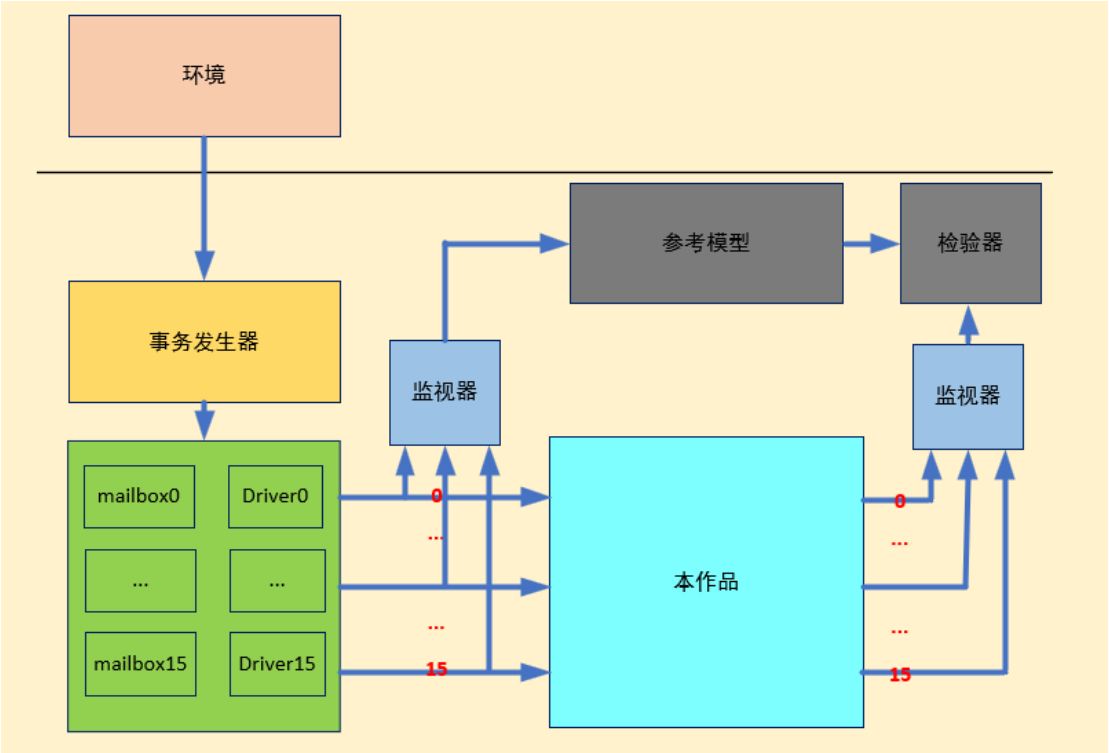


图 5.1.1 基于 SystemVerilog 的分层验证平台框架

验证平台包含了验证环境，事务发生器，驱动器，监视器，参考模型以及检验器模块。各个模块的功能如下所示：

1) 验证环境模块对随机化的部分参数进行了设置，并串联起整个验证流程，包含创建对象、构建模块、运行以及最后的统计。

2) 事务发生器中产生发送的数据包，并对数据包进行受约束的随机化，然后将数据包传递到驱动器模块。

3) 驱动器模块负责和待测平台(DUT)进行实际连接，将事务发生器产生的数据包根据实际接口的时序要求，送进 DUT 中。

4) 监视器模块有两个类型，输入监视器和输出监视器。输入监视器将驱动器发送至 DUT 的输入信号，打包后发送至参考模型。输出监视器，将 DUT

的输出信号，打包后送至检验器模块。

5) 参考模型，将输入的数据包进行处理，得到期望的输出结果，这个这个结果数据将会送进后续的检验器模块，进行后续的处理。(这里的参考模型进行了简化处理，后面后详细介绍)

6) 检验器模块,将参考模型的输出数据和输出监视器的输出数据进行比较，输出最终的校验结果。

整个系统启动后,事务发送器会根据验证环境中的配置产生相应约束和个数的数据包，然后驱动器模块将数据包从对应的端口输入进 DUT 中，在 DUT(本作品)中进行调度处理。同时，输入监视器会将输入 DUT 的信号打包发送到参考模型中，经过参考模型的处理，结果会输入到检验器，输出监视器会将 DUT 输出的信号进行打包发送到检验器中。检验器模块会实时对比参考模型和 DUT 的输出结果,并将比对结果进行日志输出。

5.2 验证环境设计

在顶层的验证环境中，主要包含两个主要任务:系统配置以及整个验证流程的启动。

在系统配置中,目前可以配置端口使能参数 port_enable 和端口发送的数据包数量参数 port_ncells。port_enable 参数是一个长度为 16 的数组，数组的每个元素的值可以为 1 或者 0，1 表示启用这个端口。参数 port_ncells 也是一个长度为 16 的数组,数组的每个元数表示在仿真过程中该端口发送的数据包数量。

在验证流程的启动中，主要包含 3 个过程，过程的示意图如下图 5.2.1 所示。

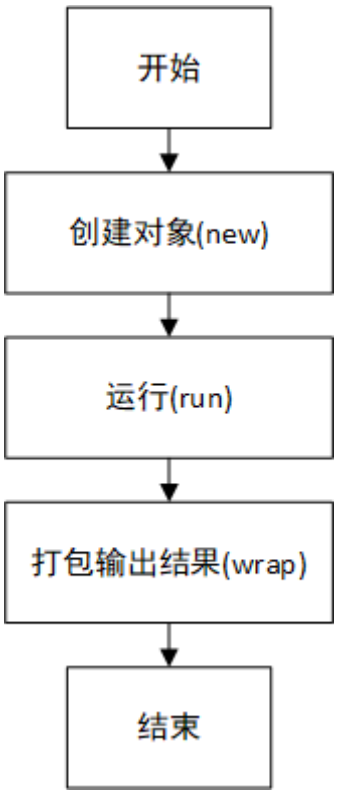


图 5.2.1 验证启动流程

首先，对验证平台各个子模块对象进行创建，包含事务发生器对象、驱动器对象、监视器对象、参考模型对象以及检验器对象。在运行阶段，启动了多个线程来运行各个验证平台子模块，其中有 16 个线程来运行 16 个事务发生器和驱动器，分别驱动 DUT 的 16 个端口。在打包输出结果阶段，会对校验器的统计结果进行统一的输出。

5.3 事务发生器模块设计

事务发生器模块用于产生用于测试的数据包，产生数据的方式使用受约束的随机化策略。事务发送器的工作流程如下图 5.3.1 所示。

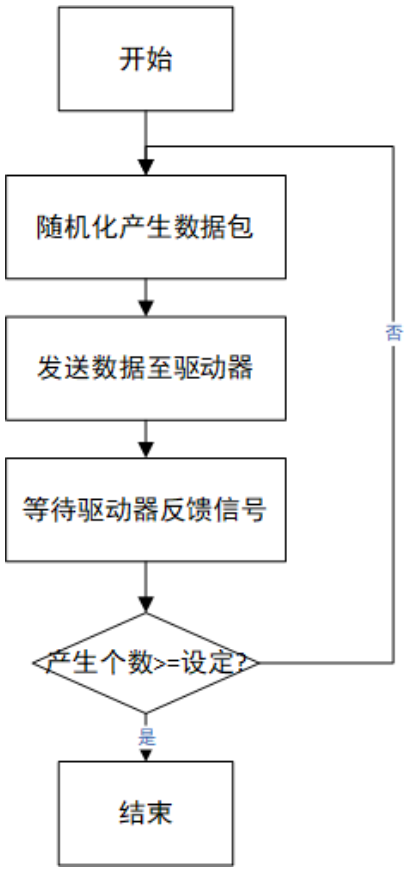


图 5.3.1 事务发生器工作流程

事务发生器启动后，首先产生数据包，然后通过 SystemVerilog 的信箱机制将数据包发送驱动器模块，然后等待驱动器发送完该数据后给出的反馈信号，之后判定产生的个数是否已经达到设定的要求，并决定是否结束任务。

事务发生器和驱动器之间同步机制采用的是双向通信机制。事务发生器向驱动器发送 1 个数据包后，需要等待驱动器发送完该数据后的反馈信号，然后才会产生下一个数据。这样考虑的原因是避免事务发生器的发送速率和驱动器的接收速率不匹配，导致驱动器在短时间内产生大量数据，导致存储数据包占用的空间过大。

数据包事务的格式定义以及随机化约束如下图 5.3.2 所示。

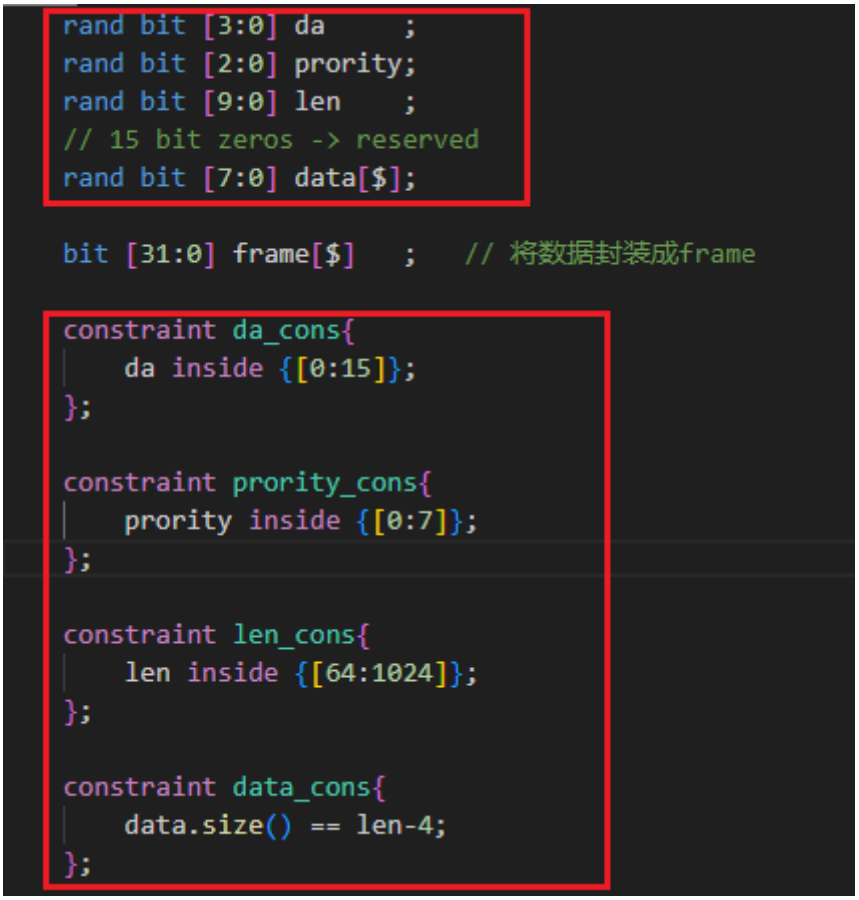


图 5.3.2 随机化变量及约束

随机化过程主要是对数据包的目的地址，优先级，数据长度和数据进行随机化，因此定义了 4 个随机变量，并根据要求对其添加了约束。

5.4 驱动器模块设计

驱动器模块主要负责将事务发生器产生的数据包，转换为符合端口时序要求的端口信号。驱动器模块的工作流程如下图 5.4.1 所示

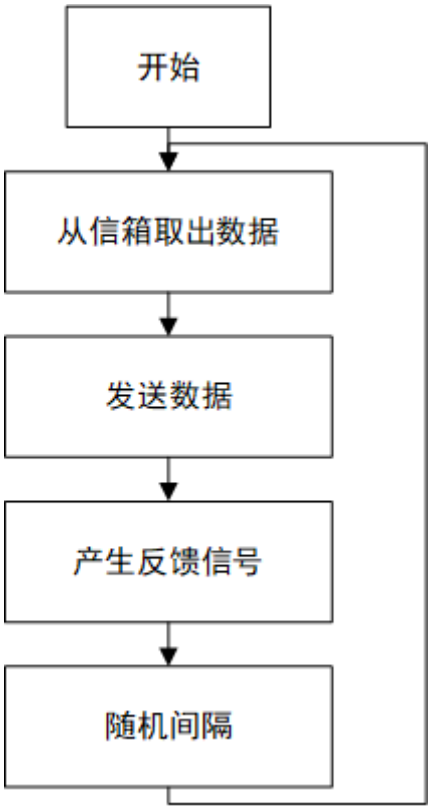


图 5.4.1 驱动器工作流程

驱动器模块工作时首先从连接事务发生器的信箱中取出数据，由于SystemVerilog 内置的信箱机制从信箱中取数据是阻塞模式，因此退出取出数据状态时一定能够保证拿到了数据。后续根据端口时序发送数据，然后向事务发生器模块发送反馈信号，通知发生器可以产生下一个数据包，随机等待一段时间，模拟数据包发送时间的随机性，之后进入下一轮处理。

5.5 监视器模块设计

监视器模块的主要功能是驱动器将发送给DUT 和DUT 最终的输出数据重新进行打包，转换为事务的格式，进行后续的检验处理。

驱动器发送给 DUT 的数据，经过监视器打包后会发送给参考模型，DUT 输出的数据，经过经过监视器打包后会发送给检测器。监视器和参考模型以及监视器和检验器之间的通信方式都是采用 SystemVerilog 的信箱完成。

5.6 参考模型设计

参考模型的作用是产生期望的 DUT 结果，用于后续检验器的参考数据来源。

本作品高速多端口共享缓存管理模块，内部包含复杂的控制逻辑以及优先仲裁处理，因此本作品的参考模型实现起来非常复杂、困难。目前，采取了一种简单的策略来粗略的对 DUT 进行验证测试。

目前，数据包的头部数据中只使用了 0-16 的 17bit 数据，高 15bit 数据是保留状态。不影响 DUT 正常功能的情况下，在设计的数据包格式基础上进行了简单修改，增加了 4bit 的源地址字段 src，占用原头部数据的 17-20 的保留字段区域。修改的报文格式如下图 5.6.1 所示。

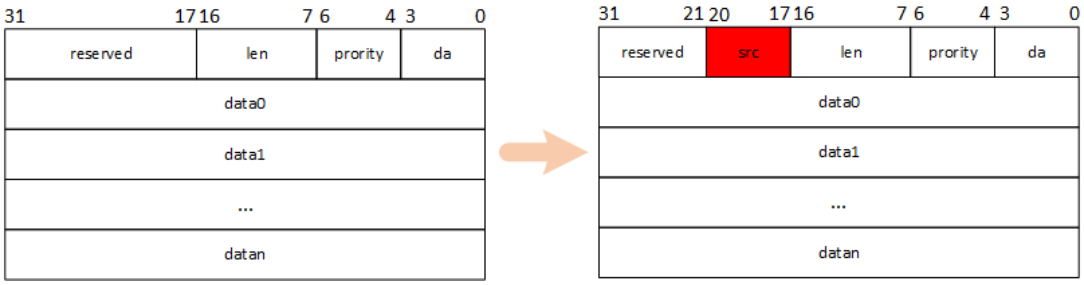


图 5.6.1 验证报文格式修改

在参考模型中会对监视器送入的数据包进行处理，形成一条 85bit 长度的记录，记录的格式如下图 5.6.2 所示。

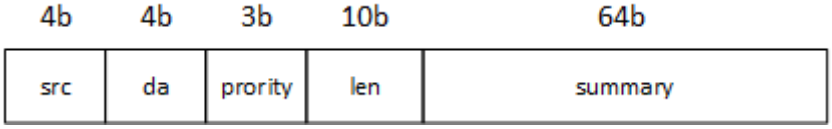


图 5.6.2 参考模型记录格式

记录的前 4 个字段字段，分别是发送数据的源端口，目的端口，优先级，长度信息，后面的 64bit 数据是对数据包中的数据(不包含头部数据)，使用摘要算法(如 md5，sha 等)形成的摘要数据(目前使用 crc 算法进行简易验证)。

驱动器每向 DUT 发送一个数据包激励，都会产生一个对应的记录数据。DUT 的输出数据也会先进行记录信息计算，然后在输入记录中进行搜索，如果可以找到则表明该数据包没有问题，如果没有找到则表明出现了错误。

5.7 检验器模块设计

检验器模块用来对参考模型输出的期望值与 DUT 输出的实际值进行比对，得出验证的结果。检验器模块内部维护了 16 个根据数据包源地址分配的记录队列，

参考模型输出的相同源地址的记录存储在同一个队列中。当新得到 DUT 的输出数据时，处理的流程如下图 5.7.1 所示。

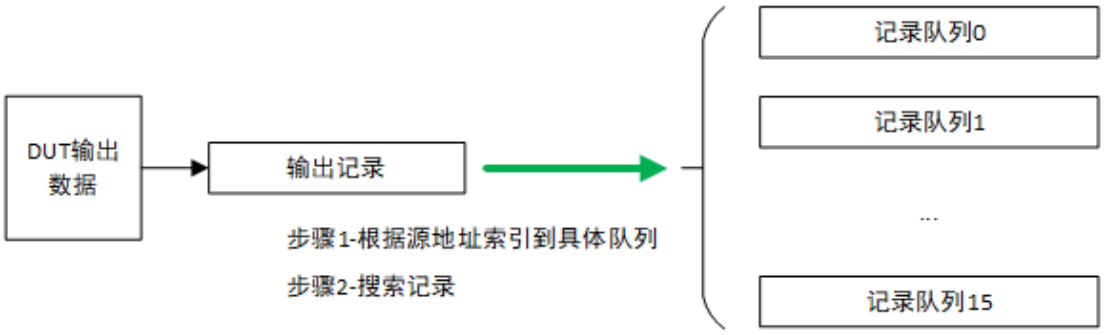


图 5.7.1 检验器处理示意图

首先对 DUT 的输出数据进行记录计算，得到一条记录数据，然后根据记录数据的源地址索引到存储对应源地址记录的队列，在该队列中进行新生成记录的搜索，如果没有找到对应的记录则输出对应的日志并记录信息，如果找到了对应的记录，则表明数据包正确，然后从队列中删除匹配的记录。

6 仿真测试

6.1 测试计划

仿真测试平台：QuestaSim10.6c/ModelSim10.5。测试过程分为两个大的部分：子模块的测试和整体测试。

1）子模块测试

针对各个子模块编写相应的 TestBench 进行功能性的测试。

2）整体测试

整体测试计划分为 4 个步骤：

a） 完成 1 个端口向 1 个端口发送数据的测试。

b） 完成 1 个端口向 16 个端口发送数据的测试。

c） 完成 16 个端口向 1 个端口发送数据的测试。

d） 完成 16 个端口向 16 个端口随机发送数据的测试。

6.2 子模块测试

6.2.1 DPRAM 模块仿真测试

对 DPRAM 模块的读写进行测试，测试波形图如下图 4.1 所示，测试输出如下图 6.2.1 所示。

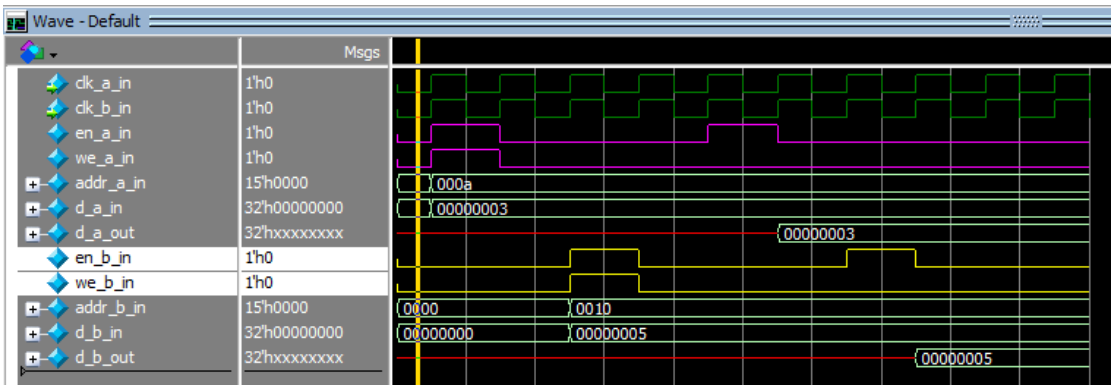


图 6.2.1 DPRAM 模块仿真波形

```
VSIM 10> run
# @5: PortA write at 000a with value          3 !
# @25: PortB write at 0010 with value          5 !
# @55: PortA read at 000a with value :          3!
# @75: PortB read at 0010 with value :          5!
```

图 6.2.2 DPRAM 模块测试输出

对 DPRAM 模块进行读写仿真测试，均可以得到正确的结果，符合预期功能。

6.2.2 输入控制模块仿真测试

对输入控制模块进行软件仿真测试，输入控制模块主要功能是 1.检测到端口有数据输入，向内存管理模块申请存储地址，并 block 地址转化为存储每一拍数据的实际 SRAM 地址；2.解析帧头，讲解析得到的每个帧的目的地址（o_da），优先级（o_priority）以及申请到的存储地址发送给队列管理模块进行管理；3.和写 SRAM 模块连接，把申请到的 SRAM 地址（o_sram_addr_vld）输入写 SRAM 模块。

下图 6.2.3 所示是输入控制模块检测到起始帧标志“i_sop”，开始向 mem_nanager 申请数据，拉高“o_addr_req”并且接收到 block 地址；解析帧头数据“32'h0001ff14”,解析得到目的地址“o_da”为 4，优先级“o_priority”为 1，发送到 pqm 模块进行队列管理。波形如下图所示。

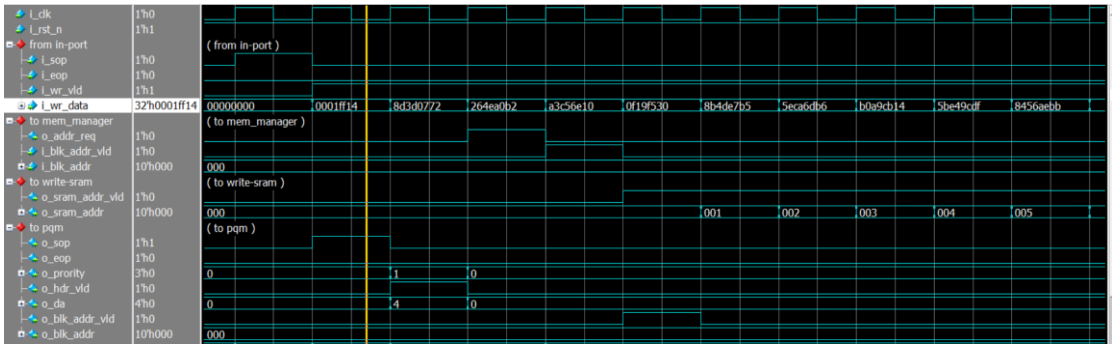


图 6.2.3 输入控制模块仿真波形 1

最后，和 write_sram 模块进行交互，把输入端口的连续帧根据切分成多个 block，多次向 mem_manager 模块请求 block 地址并把基于 block 的实际 SRAM 地址输出到写 SRAM 模块。如下图展示了和写 SRAM 模块交互的过程的波形图。

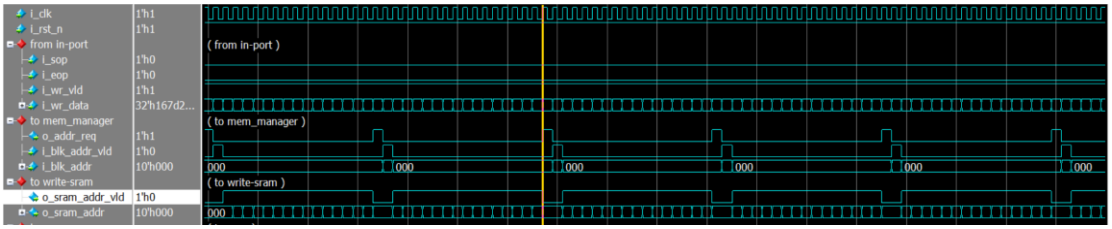


图 6.2.4 输入控制模块仿真波形 2

6.2.3 写 SRAM 模块仿真测试

对写 SRAM 模块进行软件仿真测试，写 SRAM 主要功能是读取从端口输入到输入 FIFO 中的数据，并根据输入控制模块提供的 SRAM 地址把数据写入相应的 SRAM 地址中进行存储。

先向 FIFO 中输入 32 拍数据，数据位宽位 32bit。波形图如下图所示。

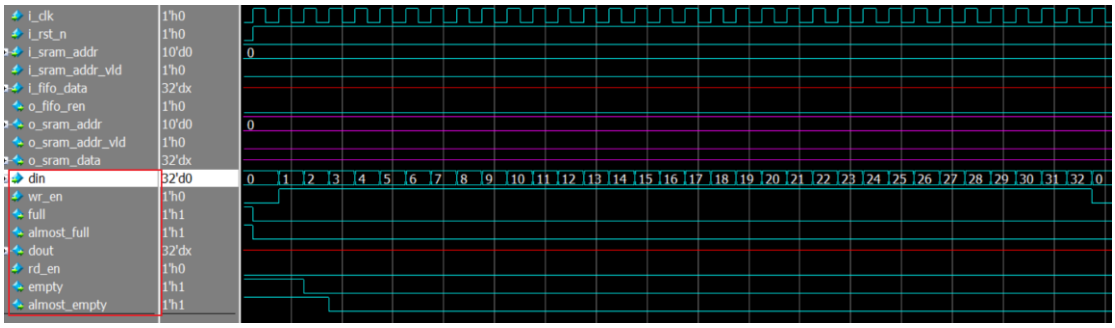


图 6.2.5 写 SRAM 模块仿真波形 1

当 write_sram 模块接收到输入控制模块提供的 SRAM 地址，会开启 FIFO 读使能，同步把 FIFO 中的数据存入 SRAM 中的对应地址。如下图 6.2.6 所示。

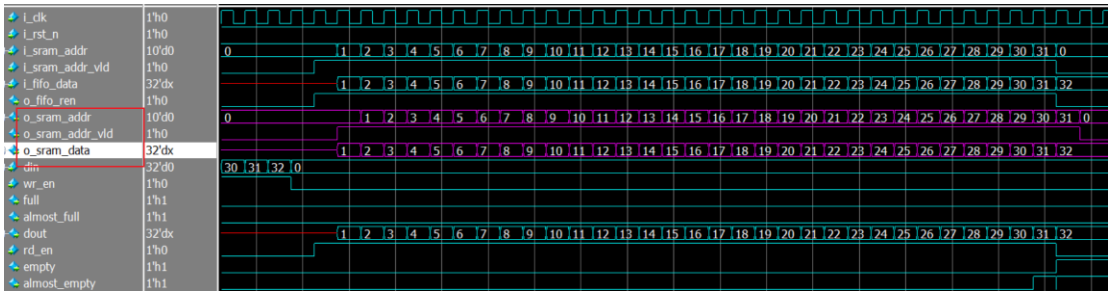


图 6.2.6 写 SRAM 模块仿真波形 2

6.2.4 内存管理模块仿真测试

内存管理模块主要是当有模块需要申请 block 地址，即拉高“ocp_req”，内存管理模块会在 2 拍以内回复“ocp_rsp”，如果有空闲内存块，就拉高“ocp_rsp”，

否则拉高“full”信号表示没有多余的空闲地址可以进行分配了。如下面波形图所示，当分配完 03ff(hex)地址的 block 块之后，内存没有新的可以进行分配的地址，因此下次再进行申请，模块不会再拉高“ocp_rsp”。

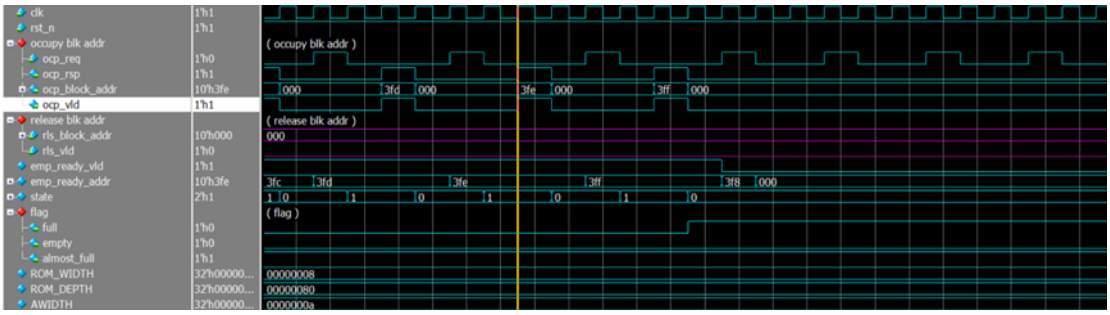


图 6.2.7 内存管理模块仿真波形 1

在内存分配满的情况下，如果释放了部分内存块，如图所示，释放了 block 地址为 007、013、029 的内存块之后，下次再向该模块申请新的内存地址，模块分按照优先级提供地址为 007、013、029 的内存进行响应。结果如下图所示。

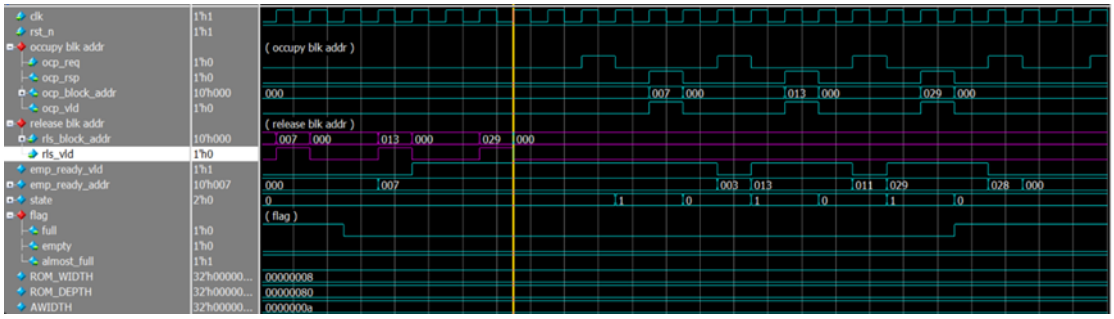


图 6.2.8 内存管理模块仿真波形 2

6.2.5 优先级队伍管理模块仿真测试

当有一帧数据进入 SRAM_Controllor,input_ctrl 模块会提取帧头中的 DA 和 Piror 字段给 PQM(①框),同时向 MEM_Manager 申请空闲地址,当 MEM_Manager 返回一个空闲地址后，input_ctrl 模块会将此地址送入 PQM（②框）。PQM 根据 DA 和 Piror 寄存器检查对应的子队列是否被创建，若没有，则将此地址写入子队列的队首和队尾地址，同时写入逻辑队列（③框）。

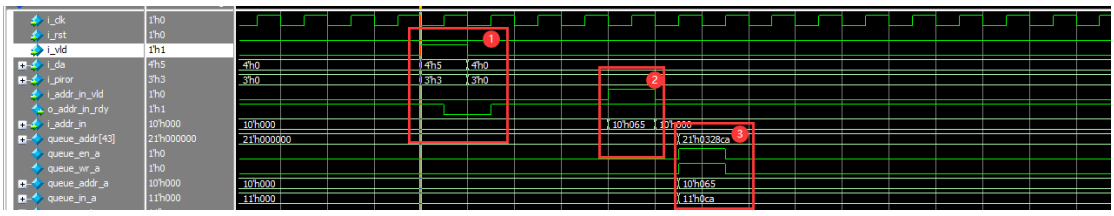


图 6.2.9 PQM 模块仿真波形 1

当 PQM 收到帧尾地址时（①框），PQM 根据 DA 和 Piror 寄存器更新子队列队尾地址并读取原队尾地址（②框），然后将新地址写入原队尾地址的下一跳地址（③框）。

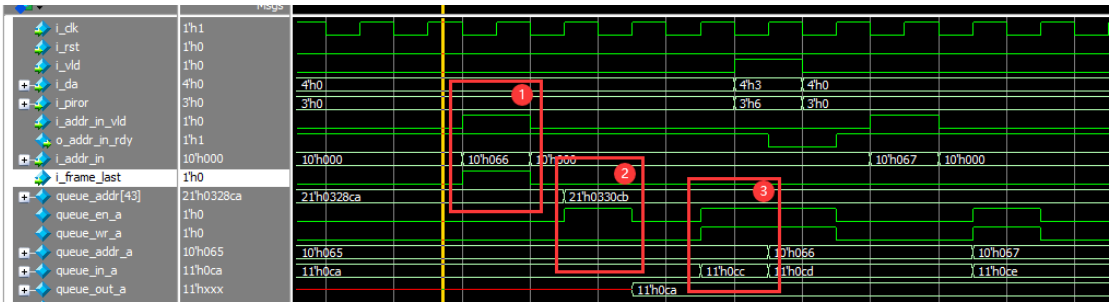


图 6.2.10 PQM 模块仿真波形 2

当输出许可信号拉高时，PQM 找到此输出许可的最高优先级子队列（①框），然后读取此队列的队首地址的下一跳地址并将队首地址输出出去（②框），等 output_ctrl 处理完上一个地址块内的所有有效数据将 addr_out_rdy 信号拉高时，PQM 将释放刚送出去的那个地址，同时将读到的下一跳地址发送出去并更新子队列的队首地址（③框），直到发送完帧尾地址后停止。

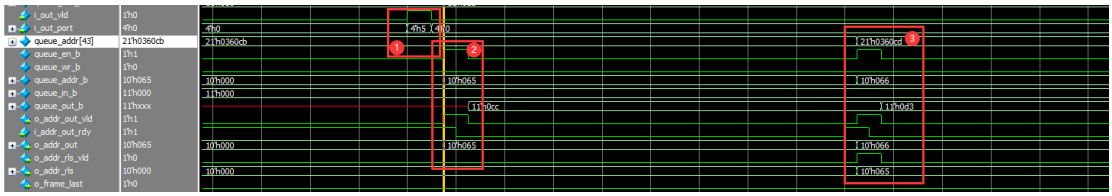


图 6.2.11 PQM 模块仿真波形 3

6.2.6 出队仲裁模块仿真测试

如下图所示，出队仲裁仲裁的仿真波形如下所示。

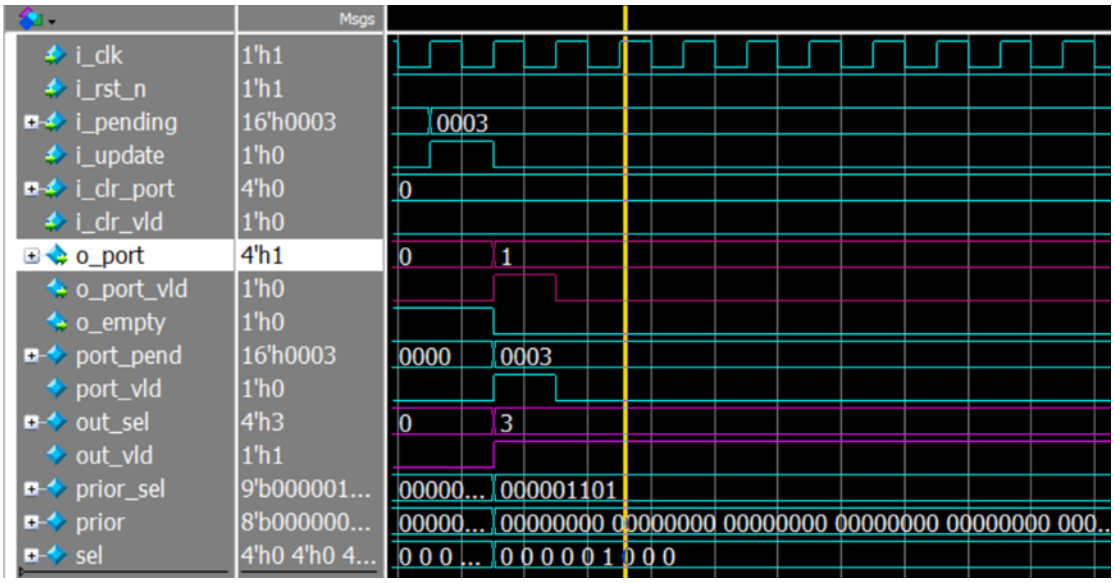


图 6.2.12 出队仲裁模块模块仿真波形

6.2.7 输出请求模块仿真测试

输出请求模块功能是其他模块向该模块请求可以输出数据的端口，如果请求成功，该模块会输出请求端口有效。如下图波形图所示，输入该模块的请求信号有效请求端口号为 f（hex），输出有效请求信号为 f（hex），表示请求成功。

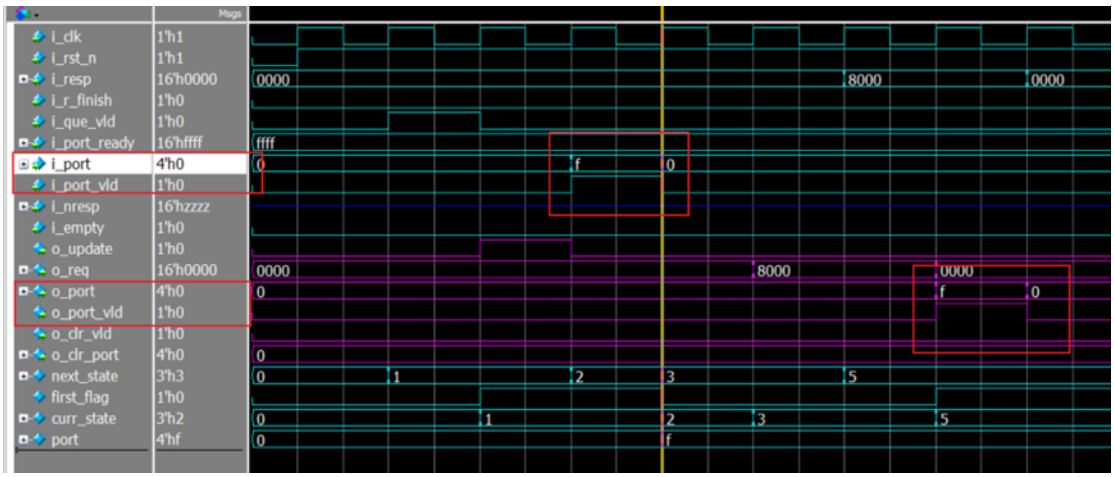


图 6.2.13 输出请求模块仿真波形

6.2.8 输出控制模块仿真测试

该模块波形图 6.2.13 如下所示。

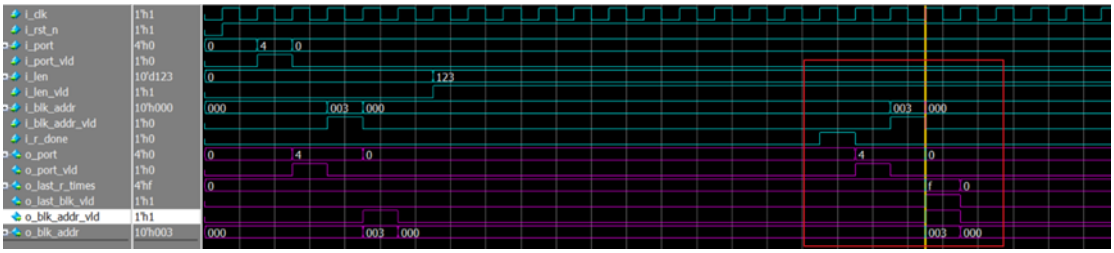


图 6.2.14 输出仲裁模块仿真波形

输出控制模块用于根据 PQM 和读 SRAM 模块的地址、目的端口地址、长度信息生成读取信息，如上图所示，当长度信息为 123byets 时，那么一共占用两个 blk 地址（每个 block 大小为 64bytes，为了方便这里的 testbench 提供的 blk 地址是常数），由于 SRAM 实际存储位宽为 32bit=4bytes，那么第二次需要读 59bytes，也就是 15 次，“o_last_r_times” 有效输出为 f（hex）=15。

6.2.9 读 SRAM 模块仿真测试

读 SRAM 模块的功能是根据输出控制模块的信息产生实际地址，访问 DPRAM（SRAM）读取数据，内部有一个 FIFO，可以缓存数据并输出。如下图所示，当“i_start_packet”信号置位高电平，说明接下来的帧起始，因此从 DPRAM 读取数据后输出拉高了“o_rd_sop”信号。如下图 6.2.15 所示。

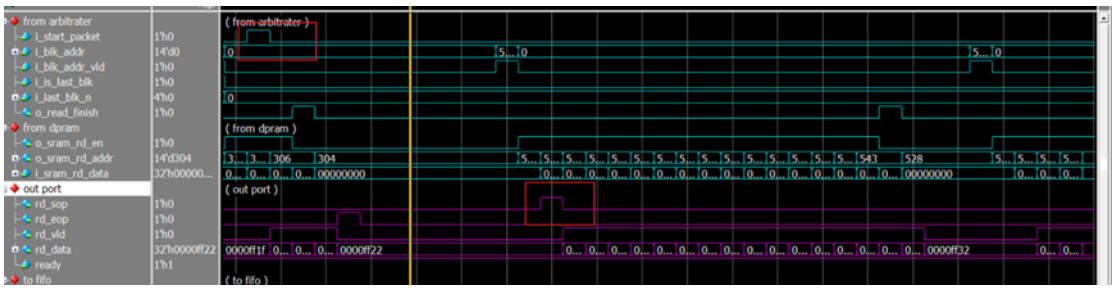


图 6.2.15 读 SRAM 模块仿真波形 1

当 output 发送信号标记为最后一个 block 信号，表面接下来读取的 SRAM 地址中存放的是帧结尾，且“i_last_blk_n”=2 ；在读 SRAM 模块输出端“o_rd_eop”置位为 1，且读了 3 个周期（2+1=3），因为最后一个存储空间不可能为 0，故当“i_last_blk_n”=0 时，需要从 SRAM 读取 1 个周期的数据。结果如下图所示。

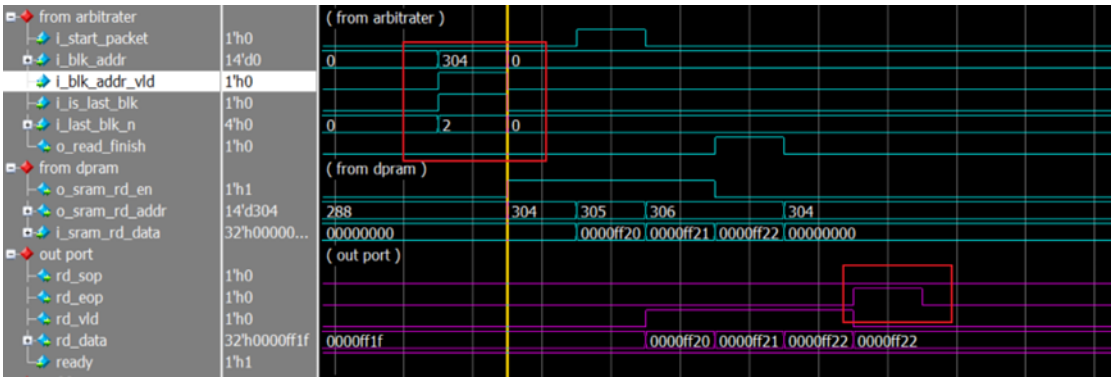


图 6.2.16 读 SRAM 模块仿真波形 2

6.3 整体测试

整体测试方案如下所示。

```
: @0,the [          4] driver begin to work
: @0,the [          3] driver begin to work
: @0,the [          2] driver begin to work
: @0,the [          1] driver begin to work
: [frame info]-> da =  4,prority= 1,len =  216
: @0,generator new cell
: @0,the [          0] driver begin to work
: @150000,[port          0] send          0 cell
: driver tr datalen:        212
: frame len :          54
: @5850000 [TEST NOTE]: simulation finish~~~~~~~~~~~~~~~~~~~~
```

图 6.3.1 整体测试输入帧参数

向 SRAN_Controllor 模块输入一个 DA 为 4，优先级为 1，长度为 216 的数据包（①框），然后输出这个数据包（②框）

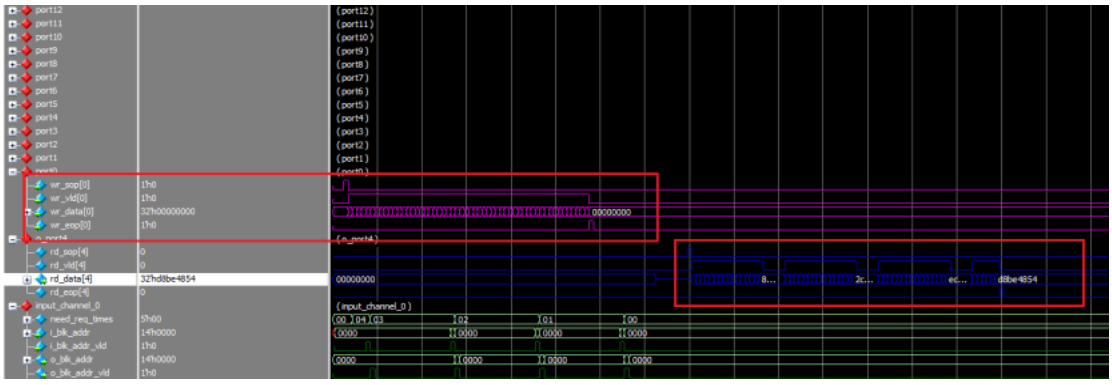


图 6.3.2 输入输出波形图

放大①框和②框，输出端口之前，输出数据包与输入数据包相同，整体测试正确。

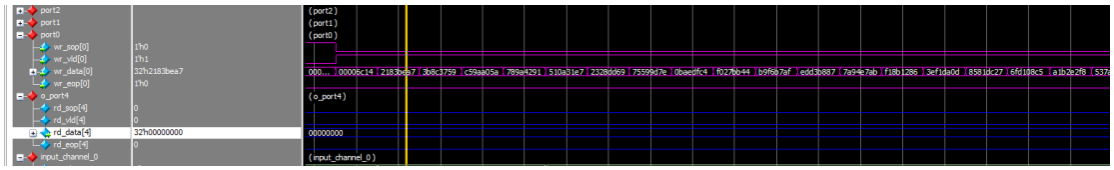


图 6.3.2 输入波形图

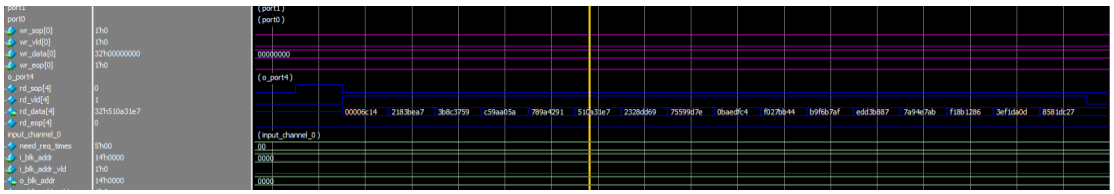


图 6.3.4 输出波形图

7 Synthesis&Implementation

7.1 开发平台

平台使用米联客 MZ7100FC 开发板，它搭载 Xilinx 公司开发的 Zynq-7000 系列芯片，型号为 XC7Z100FFG900-2。

图 7.1.1 米联客 MZ7100FC 开发板实物图

该 FPGA 平台的资源情况如下表 7.1.1 所示

表 7.1.1 FPGA 资源情况

FPGA 主要参数	架构	Kintex-7
	LUT	277400
	Block RAM	26.5Mb
	DSP	2020
	DFF	554800
	GTX 收发器	16 对

使用的开发工具和开发语言如下所示：

1）开发工具：Vivado2022.1

2）开发语言：SystemVerilog

7.2 资源使用情况

对本作品进行 Synthesis 后，查看其资源使用情况，如下图 7.2.1 所示。

Resource	Utilization	Available	Utilization %
LUT	114202	277400	41.17
LUTRAM	384	108200	0.35
FF	70736	554800	12.75
BRAM	280	755	37.09
IO	1124	362	310.50

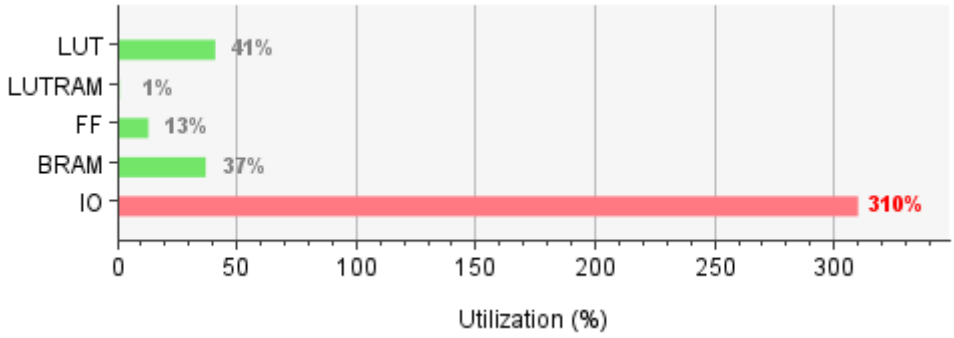


图 7.2.1 资源使用情况

LUT、FF、BRAM 等资源的使用情况，MZ7100FC 开发板能够满足所需，但是对于 IO 资源的使用，大大超出了所能使用的资源。

16 个输入端口和 16 个输出端口，每个端口的数据总线位宽为 32bit，仅仅考虑数据端口，就很少有 FPGA 资源能够满足 IO 的资源需要，因此对于 FPGA 原型验证不能直接下板测试，需要重新设计方案。

7.3 FPGA 原型验证方案

由于本作品的输入输出端口较多，如果直接下板测试需要大量的 IO 资源，很少有 FPGA 板卡能够满足，因此测试数据不能由外部产生，只能由 FPGA 内部产生，设计的 FPGA 原型验证方案如下图 7.3.1 所示。

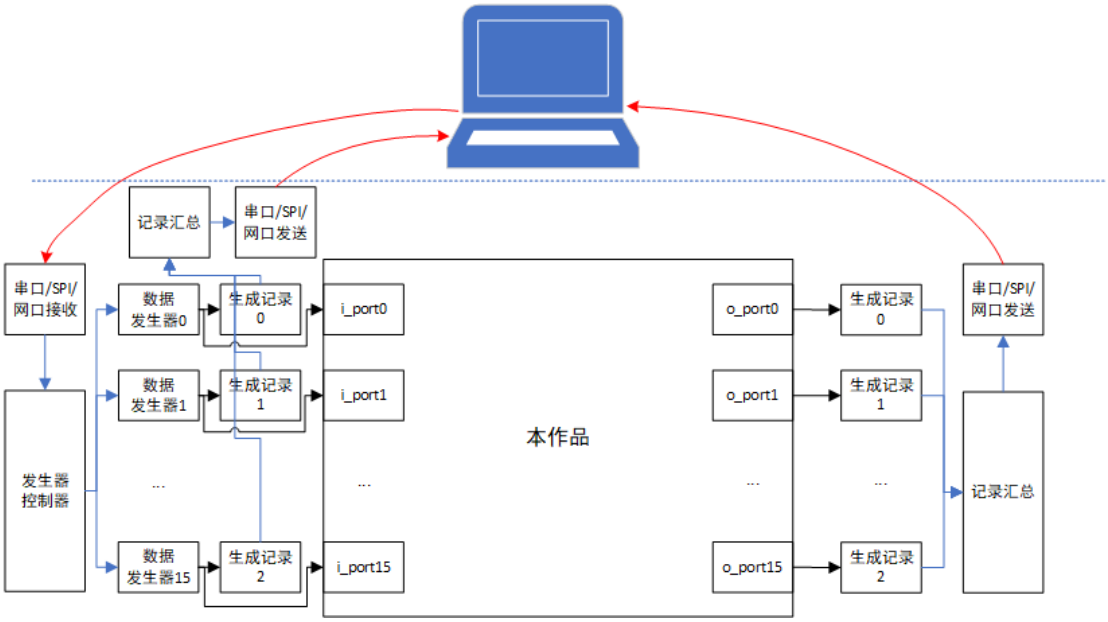


图 7.3.1 FPGA 原型验证方案

图中虚线以下的部分都是在 FPGA 上实现(包括本作品)，内部包含 16 个数据发生器和 16 个数据接收器。数据发送器用于产生用于测试的数据，数据接收器用于对产生的数据进行收集。

其中数据发生器是受发生器控制器控制的，控制的主要内容包含数据包的目的地址、优先级、数据包长度等信息，这些数据实际上是从上位机(PC 端)通过串口/SPI/网口等接口发送给发生器控制器的。

本作品的数据端口会连接生成记录模块，将接收的数据形成一条记录，记录的格式和本次使用的验证平台的参考模型实现有着较大相似之处，记录格式如下图 7.3.2 所示。

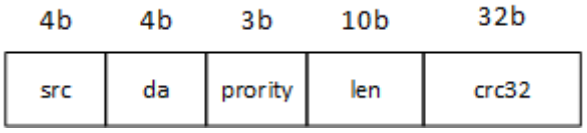


图 7.3.2 FPGA 原型验证记录格式

同时，数据发生器产生的数据也会经过生成记录模块，产生的记录进行汇总后通过串口/SPI/网口发送模块，传输至上位机系统，进行比对。

在整个 FPGA 原型验证的过程中，PC 端的上位机发送的是产生数据的简单指令，接收到的是发送数据的记录和接收数据的记录。

这个方案可以极大程度的减少对 FPGA IO 资源的消耗。

8 不足与改进

1) **不足与改进 1：**目前将 SRAM 分成了 16 个组，端口 n(0<=n<=15)使用第 n 个组 SRAM，每个端口都不能使用其他组的 SRAM，这样会导致一个端口的 SRAM 组满了，但是其他 15 个 SRAM 组仍然空闲的情况下，该端口不可以继续传输数据。

   后续改进方向：端口首先使用本端口对应的 SRAM 组，在 SRAM 组满了，但是其 SRAM 组仍然有空闲的情况下，可以向其他组的 SRAM 进行写入。

2) **不足与改进 2：**在数据出队的过程中，向 PQM 队列管理模块请求地址时，目前是当一个地址块完全读取完成后再去请求地址，这样导致读取完成和请求地址之间有 2 个时钟周期的空闲。

   后续改进方向：实现地址的提前请求。

3) **不足与改进 3：**数据校验部分，考虑使用 ECC 纠错带来的巨大空间开销，目前仅仅使用了 CRC 校验，不具备纠错功能。

   后续改进方向：对数据包的头部使用 ECC 纠错，对数据包的数据部分使用 CRC 校验。

4) **不足与改进 4：**参考模型目前不具备优先级验证功能

   后续改进方向：优化参考模型