第5章

共享存储交换单元

共享存储(Shared Memory,SM)交换单元(或者称为交换结构),属于时分交换单元的一种,是以太网交换机、路由器等网络设备中广泛使用的基本电路。共享存储交换单元将所有等待输出的数据包(分组)先存储到一个公共存储区中,根据其输出端口和转发优先级进行排队,然后根据输出调度规则将分组读出并发送到目标输出端口中。共享存储交换单元本身是一个队列管理器,具有存储资源利用率高、结构简单、低时延的特性。共享存储交换单元曾广泛应用于大容量 ATM 交换机中,应用于 IP 交换机时,输入的 IP 包通常先被分割成定长的内部信元(典型值为 64 字节),然后进入共享存储交换单元。同一个 IP 包对应的内部信元被发送到交换机的出口处时,重新拼装成 IP 包并输出。

5.1 共享存储交换单元的工作原理

图 5-1 给出了 SM 交换结构的工作机制示意图。如果输入的是 ATM 信元,那么经过对输入信元的接收处理(信头校验检查、信元类别检查、转发表查找等),与该信元转发相关的内部控制信息会形成一个本地头(或者称为本地标签)。本地头中主要包括输出端口映射位图、转发优先级等字段。输出端口映射位图字段的位宽与输出端口数相同,每个比特对应一个输出端口,某个比特为 1 表示要从该端口输出,多个比特为 1 表示要从多个端口输出。

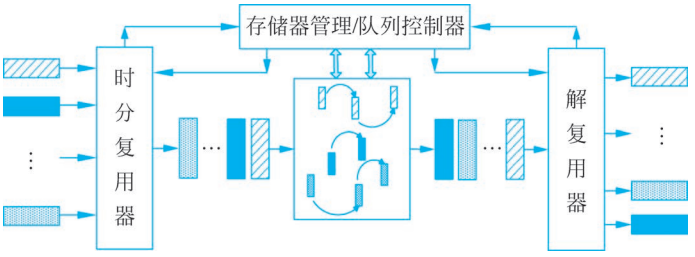

图 5-1 SM 交换结构的工作机制示意图

对于 IP 分组,可以在进行 IP 包输入处理后,为 IP 包加上包含包长度、输出端口映射位图、转发优先级等信息的本地头,然后进行定长分割,形成多个按序排列的内部信元。根据

具体实现方式的不同，与一个 IP 包对应的转发控制信息可以只出现在这个 IP 包的首信元头部，也可以添加到该 IP 包拆分后的每个信元头部。无论采用哪种方式，内部信元的长度都是固定的，典型值为 64 字节。

图 5-1 中，来自不同输入端口的信元经过时分复用器进行数据合路，合并为一路高速数据流，交由队列管理器处理。队列管理器对共享的数据缓冲区进行分割和管理，它根据信元本地头中的转发映射位图和转发优先级等信息将去往不同输出端口的信元存储在不同的逻辑输出队列中。去往不同端口或者去往相同端口但具有不同转发优先级的信元构成一个逻辑队列，逻辑队列的数量与端口数和优先级数都有关系。在某一时刻，业务量大的输出端口可能占用更多的数据缓冲区，业务量小的端口占用的缓冲区深度会相对较少。与为每个输出端口固定分配缓冲区的方式相比，共享缓冲区有利于提高存储空间的利用率，改善系统性能。

数据缓冲区可以根据业务优先级的不同，为分组分配不同的缓冲区使用权限。例如，低优先级业务在存储空间的占用超过一定门限时会被限制写入并丢弃。待转发业务具有多个优先级时，可以设置多个缓冲区管理门限，使得高优先级业务具有相对更高的缓冲区使用权。此外，可以将共享缓冲区划分为私有缓冲区和共享缓冲区。针对高优先级业务可以预留私有缓冲区，高优先级业务到达时，优先使用私有缓冲区，私有缓冲区被用尽后，可以占用共享缓冲区；低优先级业务不能占用私有缓冲区，只能占用共享缓冲区。这种机制可以使特定的高优先级业务得到基本的缓冲区保证。

存储器访问带宽是影响 SM 交换结构吞吐率的主要因素。进入交换结构的信元需要写入共享数据缓冲区中，从输出端口发送的信元需要从共享缓冲区中读出，缓冲区的读写访问带宽直接制约了交换结构的吞吐率。一般来说，交换机吞吐率的理论上限是存储器访问带宽的二分之一。例如，缓冲区读写数据总线位宽为 128 比特，读写时钟频率为 100MHz，则其峰值访问带宽为 12.8Gb/s。如果缓冲区采用单端口 RAM 实现，此交换单元的交换能力可以按照 6.4Gb/s 进行估算；如果采用双端口 RAM 实现，支持同时读、写，交换单元的交换能力可以按照 12.8Gb/s 进行估算。

图 5-2 给出一个包括 N 个端口的共享存储交换结构。其各个组成部分的基本功能如下。

(1) 输入接口。来自 N 个输入端口的 IP 包经过路由查找等前级处理后得到包括输出端口映射位图和转发优先级在内的转发控制信息。IP 包在进入交换结构之前需要进行分割，IP 包带有本地头(包含转发控制信息)，本地头中包含了数据包长度、输出映射位图、转发优先级等信息。来自不同输入端口、经过分割的数据包在交换结构的输入接口处进行合路，成为一路高速数据流(合路电路也可以出现在前级，先合路再进入交换结构，此时输入接口不需要实现合路功能)。此后，输入接口会根据当前 IP 包的转发优先级，检查信元存储器的使用量情况，判断是否接受当前输入的数据包。如果可以接受，则从空闲地址(指针)队列管理器中读出一个指针(地址)，然后将当前信元写入该存储空间，并将指针根据输出映射位图写入对应的队列控制器(Queue Controller，QC)中。如果不可以接受，那么输入接口将根据相关算法丢弃该当前 IP 包对应的所有信元。

(2) 空闲(自由)地址(指针)队列管理器。在交换结构进行初始化时，它存储着信元存储器的全部空闲(自由)地址指针，自由指针存储器(缓冲区)深度与信元存储器(缓冲区)能够存储的信元个数相同。有信元需要写入信元存储器时，外部电路首先从本电路读出一个自由指针，根据自由指针生成信元存储器的写入地址并进行写入操作；外部电路同时会根

据信元对应的 portmap 信息，判断其是单播信元还是组播信元，然后以自由指针为地址，向多播计数存储器写入此信元需要转发的次数。如果是单播信元，只转发 1 次，如果是多播信元，则写入具体需要转发的次数。交换结构进行信元写入操作的同时，将此自由指针写入相应输出端口的队列控制器(QC)。当交换结构从一个 QC 读出一个待发送信元的指针并将对应的信元从信元存储器中读出后，会根据当前信元的读指针读出相应多播计数器的值，如果转发次数为 1，则将当前指针写入空闲地址队列管理器；否则将当前的多播计数值减 1 后重新写入多播计数存储器。

(3) 队列控制器 QC。每个队列控制器对应着一个输出端口，管理着该端口的输出逻辑队列。有一些交换结构支持多个优先级，因此一个 QC 内部有多个具有不同优先级的子队列。

(4) 输出接口。它以公平轮询的方式检查各个 QC 是否有输出请求，如果某个 QC 有输出请求则取出 QC 提供的信元指针(信元在存储区中的存储地址)，从片外存储器中将信元读出并交给对应的输出端口。此后输出接口将指针交还给空闲地址队列管理器，由它根据该指针对应的组播计数器的值决定是否将指针归还到空闲地址队列。

(5) 信元存储器。信元存储器可以位于片外(如图 5-2 中所示)，也可位于片内。位于片外时，其存储空间可以很大，但读写访问速度慢；位于片内时，其读写速度可以达到最大，但存储容量通常较小。存储器的读写访问带宽直接决定了共享存储交换结构的吞吐率。

在共享存储交换单元内部，需要在多个电路中使用不同类型的调度器。如图 5-2 所示，图中的交换结构共使用了两种调度器，包括位于每个 QC 内部，对 8 个具有不同优先级的子队列进行输出调度的调度器(可采用 SP、WRR 等调度算法)和 N 个 QC 请求从存储器读入信元时使用的 RR 调度器，以及对数据存储器(图中的片外存储器)进行写入和读出仲裁的 RR 调度器。

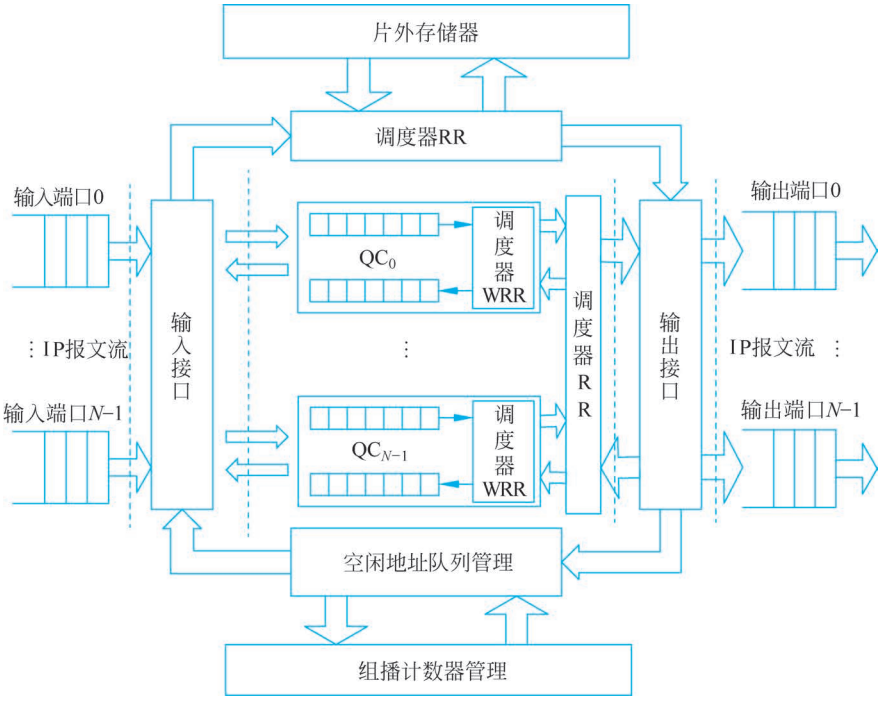

图 5-2 典型 N 端口共享存储交换结构

5.2　共享缓存交换结构及工作流程

图 5-3 是一个支持 4 个输出端口的共享缓存输出排队交换结构的电路框图，它同时是一个队列管理器。

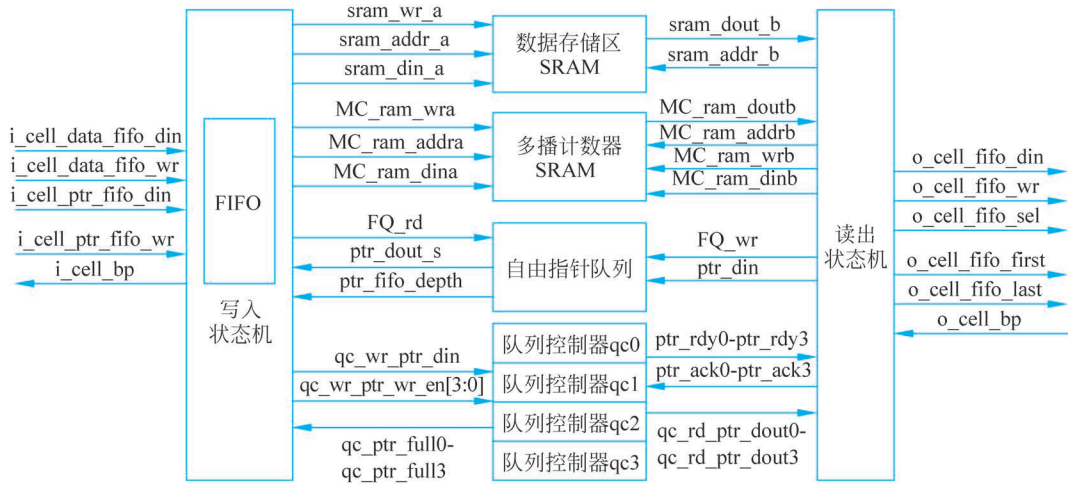

图 5-3　队列管理器电路内部结构图

下面分别介绍各组成部分的功能。

写入状态机：队列管理器的写入状态机负责本级电路和前级电路之间的数据交互。前级电路交给写入状态机的是完成数据包分割后得到的定长内部信元(长度为 64 字节)以及与该数据包对应的指针信息(包括该数据包分割后的信元数、信元转发优先级、输出端口映射位图等)。写入状态机管理着两个 FIFO(输入接口 FIFO)，用于存储输入数据包对应的信元和对应的指针。i_cell_data_fifo_wr 就是信元输入时对该 FIFO 的写信号。i_cell_data_fifo_din 是前级输入数据，在本电路中，我们将其位宽设置为 128 比特，通过这种方式，增大交换结构的内部带宽。i_cell_ptr_fifo_din 是输入的指针，位宽为 16，其中 i_cell_ptr_fifo_din [5:0]是当前写入的数据包中包括多少个信元；i_cell_ptr_fifo_din [11:8]是输出端口映射位图，用于选择从本电路的哪个端口输出；i_cell_ptr_fifo_din [14:12]是转发优先级，取值为 0～7，7 为最高优先级。i_cell_ptr_fifo_wr 是指针 FIFO 写入控制信号。i_cell_bp 是输入接口 FIFO 给前级的反压信号，当数据 FIFO 不能接收一个最大数据包或指针 FIFO "满"时其为 1，此时外部数据包不允许写入，以避免输入接口 FIFO 溢出。

数据存储区(SRAM)：SRAM 是本电路的数据存储区，也是整个交换单元的主存储区。

多播计数器：有的信元是去往多个输出端口的，每个共享数据缓冲区中的信元都有一个对应的多播计数器值，这个值是该信元要去往的输出端口总数。例如，某信元对应的多播计数值是 1，表示该信元要去往一个端口，即该信元为单播信元；如果是 4，就表示要去往 4 个端口，这个值是由信元输入时对应的 portmap 决定的。我们使用一个双端口 RAM 存储多播计数值，信元输入时，从 A 端口写入其对应的多播计数值；在信元从某端口输出后，我们从 B 端口对多播计数值进行更新。

自由指针队列：在SRAM主存储区中，每个64字节的数据块对应着一个自由指针，当有信元到达时，我们首先从自由指针队列中读取一个自由指针，然后依据这个指针将数据写入SRAM中。如果一个指针对应的信元被读出，并且多播计数器当前值为1，则该指针会被写入自由指针队列中。

队列控制器(Queue Controller，QC)：针对每个输出端口都有一个队列控制器，它采用链表结构管理从某个端口输出的分组，本设计中包括8个逻辑队列，对应8个输出优先级。

下面分析一个数据包进入队列管理器后的具体操作流程。例如，某个含有3个信元的数据包需要输入缓冲区管理器。缓冲区管理器会根据图5-3中写入状态机内部FIFO的数据深度判断输入缓冲区能否接收一个完整的最大数据包以及指针FIFO是否非满，若二者同时满足，则i_cell_bp为0，否则为1，表示有反压，无法接收当前数据帧对应的所有信元。当前数据包的所有信元都写入输入接口缓冲区的同时，数据帧对应的指针被写入接口指针缓冲区。

写入状态机在接口指针缓冲区非空、本电路所维护的共享数据缓冲区有剩余空间(自由指针缓冲区非空)时，会从自由指针队列中读出一个指针，从输入接口缓冲区中读出一个信元并写入指针所指向的64字节存储块中。这个指针被写入其portmap对应的QC中(portmap中的每个比特对应一个输出端口，同时对应该端口的QC)。同时，写入状态机会以该指针为地址，在多播计数器中写一个多播计数值，这个数值与portmap中1的个数相同。当某个输出端口对应的QC申请发送数据分组时，通过将自己的ptr_rdy信号置1表示有信元可以读出。队列管理器中的读出状态机在4个输出端口的QC之间进行轮询，若发现某个QC中有待发送的分组，就将该队列首部的指针取出，然后根据此指针，从SRAM中将对应的信元读出，通过o_cell_fifo_sel、o_cell_fifo_wr、o_cell_fifo_din三个信号，将其送往对应的输出端口，同时，根据该指针，从多播计数器RAM的B端口对其多播计数值进行更新，若当前多播计数值为1，表示该信元已经完成发送，可以将这个指针重新归还到自由指针队列当中。

需要说明的是，为了保证高优先级的业务能够获得更多的存储空间，电路中设置了8个反压门限，用于标记8个剩余缓冲区容量值。优先级越高，反压门限值越小，表示剩余缓冲区容量很小时，高优先级业务还可以被接受；优先级低的业务，在剩余缓冲区容量还相对较大时，就不能再被接受了。本设计中，不同优先级对应剩余缓冲区门限值如下，基本单位为内部信元个数。例如，PRI7_TH＝10'd32，表示剩余可用缓冲区深度小于32个信元时，不再接受优先级为7的信元(最高优先级信元)。

```
parameter       PRI7_TH = 10'd32,
                PRI6_TH = 10'd64,
                PRI5_TH = 10'd96,
                PRI4_TH = 10'd128,
                PRI3_TH = 10'd160,
                PRI2_TH = 10'd192,
                PRI1_TH = 10'd224,
                PRI0_TH = 10'd256;
```

图5-4是switch_core电路的符号图，其端口信号及具体定义如表5-1所示。

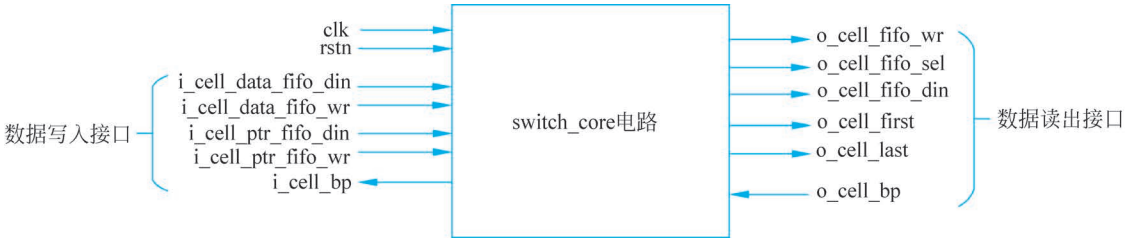

图 5-4　switch_core 电路符号图

表 5-1　switch_core 电路的外部端口及具体定义

端 口 名 称	I/O 类型	位宽/比特	含　义
clk	input	1	时钟
rstn	input	1	复位信号
i_cell_data_fifo_din	input	128	分组数据输入，位宽为 128 比特，每 4 个时钟周期写入一个 64 字节的内部信元
i_cell_data_fifo_wr	input	1	数据写信号，高电平有效
i_cell_ptr_fifo_din	input	16	当前写入数据包对应的指针， 比特[7:0]是当前数据包对应的信元数(实际只使用了低 6 位)； 比特[11:8]是输出端口映射位图； 比特[14:12]是分组转发优先级
i_cell_ptr_fifo_wr	input	1	当前数据包指针写信号，高电平有效
i_cell_bp	output	1	给前级电路的反压信号，为 1 时表示当前输入接口缓冲区不能接受一个最大帧，前级可能会因此将待写入的数据包丢弃
o_cell_fifo_wr	output	1	信元输出写信号
o_cell_fifo_sel	output	4	信元输出端口选择信号，哪个比特为 1 表示选择哪个输出端口
o_cell_fifo_din	output	128	信元数据输出
o_cell_first	output	1	输出首信元指示，输出一个分组的首信元时为 1
o_cell_last	output	1	输出尾信元指示，输出一个分组的尾信元时为 1
o_cell_bp	output	4	交换单元后级的 4 个输出端口电路给本级的反压信号，哪个比特为 1 表示该输出端口内部的数据缓冲区无法接受一个最大数据分组，此时队列管理器不应向该端口发送数据分组

5.2.1　switch_core 中的自由指针队列管理电路

图 5-3 中的 SRAM 是一个容量较大的存储器，是交换结构的共享缓冲区。一个可以存储 512 个信元的 SRAM(每个信元长度为 64 字节)的存储深度为 512(信元)，与之相对应的是一个深度同样为 512 的自由指针存储器，存储着指针值 0～511。本电路中，自由指针的位宽为 10 比特(实际使用了 9 比特，预留 1 比特是为了便于进行缓冲区扩展)，每个自由指针对应着一个在 SRAM 中可以存储一个完整信元的存储块。在初始化过程中，我们将 0～511 写入自由指针 FIFO。若 SRAM 位宽为 128 比特，则 4 个 128 比特的存储单元可以存

一个完整的信元，这也说明一个信元在存入 SRAM 中时会占用 4 个 128 比特的存储单元，这时，我们除了使用自由指针外，还应该在其低位上加两位计数值，取值为 00～11，加在一起才是我们真正使用的 SRAM 地址。也就是说，我们申请的自由指针指向的是这个信元存入 SRAM 时占用的数据存储块编号，使用自由指针和两位计数值进行并位运算后的值作为地址时，指向的才是确切的数据存储位置，如图 5-5 所示。需要注意的是，存储信元时使用的计数值 00 到 11 并不是固定的，是与存储器位宽相关的。在本电路中，SRAM 的位宽为 128 比特，那么一个 64 字节的信元需要分 4 次存入 SRAM，此时的计数值为 00～11；若将 SRAM 位宽改为 64 比特，那么一个 64 字节的信元需要分 8 次才能完全存入，此时，计数值应当改为 000～111。

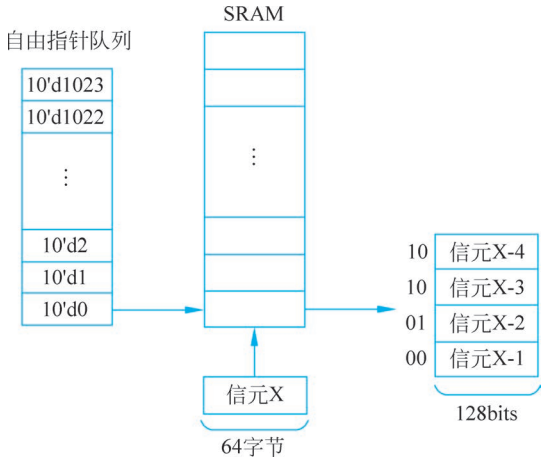

图 5-5　信元存储示意图

数据缓冲区是所有输出端口共享的。在某时刻，如果有大量数据包去往某端口，那么这些数据包可能占据较大的数据缓冲区，如果没有去往某个端口的数据，则该端口不会占用数据缓冲区。

在本设计中，实现自由指针队列功能的是 fq 模块，本质上是一个先入先出的 FIFO，下面是自由指针队列管理电路的设计代码。

```
`timescale 1ns / 1ps
module fq(
input                   clk,
input                   rstn,
input       [15:0]      ptr_din,
input                   FQ_wr,
input                   FQ_rd,
output      [9:0]       ptr_dout_s,
output      [9:0]       ptr_fifo_depth
);
reg     [2:0]       FQ_state;
reg     [9:0]       addr_cnt;
reg     [9:0]       ptr_fifo_din;
reg                 ptr_fifo_wr;
always@(posedge clk or negedge rstn)
    if(!rstn)
        begin
```

```
            FQ_state <= #2 0;
            addr_cnt <= #2 0;
            ptr_fifo_wr <= #2 0;
            end
        else begin
            ptr_fifo_wr <= #2 0;
            ptr_fifo_din <= #2 ptr_din[9:0];
            //在下面的状态机中增加了几个复位后的过渡状态,等待 FIFO 完成复位操作,正常工作时
            //不会再次进入这些状态
            case(FQ_state)
            0:FQ_state <= #2 1;
            1:FQ_state <= #2 2;
            2:FQ_state <= #2 3;
            3:FQ_state <= #2 4;
            //在状态 4 进行共享缓冲区可用指针初始化操作,将 0～511 共 512 个指针写入指针缓冲
            //区,这里指针位宽为 10 比特,最大可以支持 1024 个指针,此处只使用了 512 个指针
            4:begin
                ptr_fifo_din <= #2 addr_cnt;
                if(addr_cnt < 10'h1ff)
                    addr_cnt <= #2 addr_cnt + 1;
                if(ptr_fifo_din < 10'h1ff)
                    ptr_fifo_wr <= #2 1;
                else begin
                    FQ_state <= #2 5;
                    ptr_fifo_wr <= #2 0;
                    end
                end
            5:begin                 //归还自由指针
                if(FQ_wr)ptr_fifo_wr <= #2 1;
                end
            endcase
        end
    //注意,这里 sfifo_ft_w10_d512 表示此 FIFO 位宽为 10 比特,深度为 512,采用 fall through 模式的
    //FIFO,其读操作方式与通用 FIFO 不同
    sfifo_ft_w10_d512 u_ptr_fifo(
        .clk(clk),
        .srst(!rstn),
        .din(ptr_fifo_din[9:0]),
        .wr_en(ptr_fifo_wr),
        .rd_en(FQ_rd),
        .dout(ptr_dout_s[9:0]),
        .empty(),
        .full(),
        .data_count(ptr_fifo_depth[9:0])
        );
    endmodule
```

注意,在上面的电路中,系统复位后 FQ_state 经过几个过渡状态后才进入工作状态,原因是某些 FIFO IP 核的复位需要经过几个时钟周期才能完成,插入状态是为了等待其复位完成后再进行初始化。

5.2.2　队列控制器电路

本节将介绍队列控制器电路的工作机制。8 优先级队列控制器内部电路结构如图 5-6

所示，其在队列管理器中的位置如图 5-3 所示。

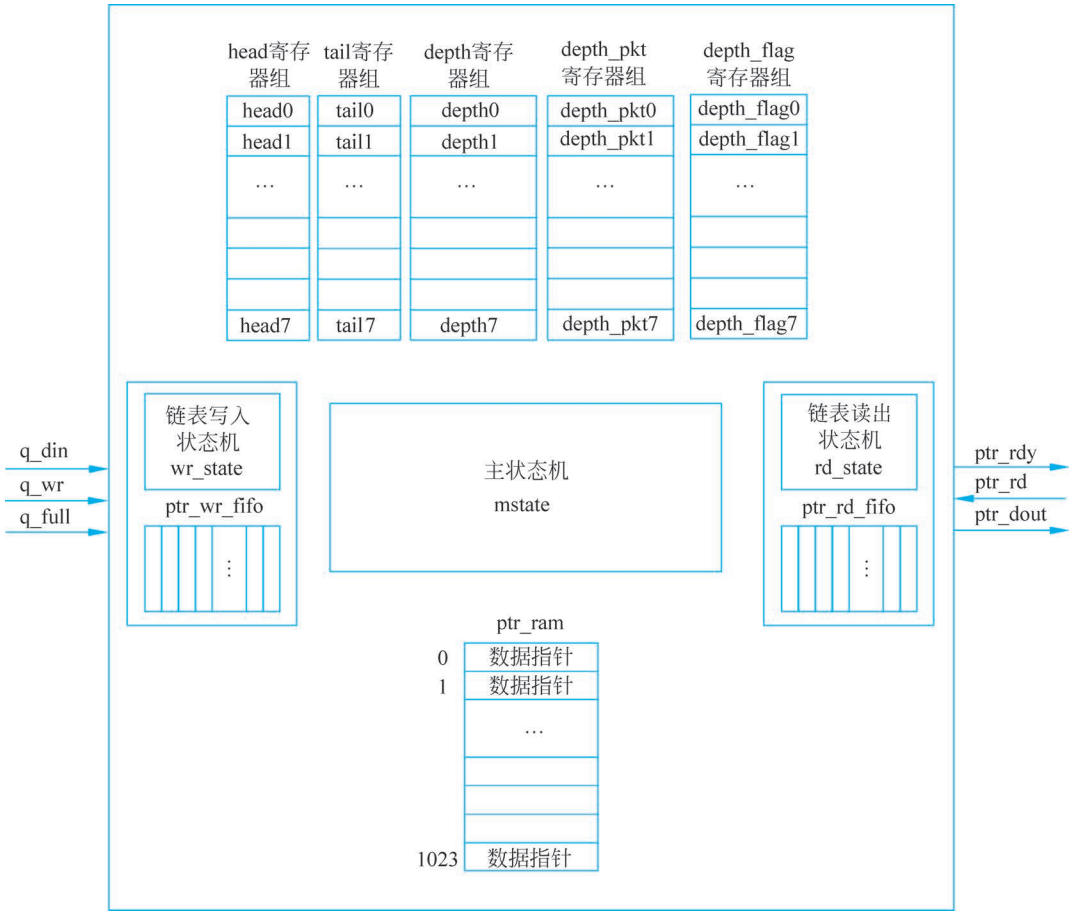

图 5-6　8 优先级队列控制器内部电路结构

在图 5-6 中包括三个状态机，一个是写入状态机，用于临时缓存队列管理器写入的指针，向主状态机发送写入请求；一个是读出状态机，用于向主状态机发送读出请求，临时缓存待输出的指针；另一个是主状态机，用于控制指针链表的建立和拆除，维护逻辑队列的状态，管理逻辑队列。

此处设计的队列控制器共维护 8 个逻辑队列，它们共用一个链表缓冲区。每个逻辑队列的头、尾、长度以及是否存储着完整的数据分组等信息，通过 head、tail、depth、depth_pkt、depth_flag 这 5 个寄存器进行维护。其中：

head：链表(逻辑队列)的头指针；

tail：链表(逻辑队列)的尾指针；

depth：链表(逻辑队列)中包括的内部信元数量；

depth_pkt：链表(逻辑队列)中包括的完整分组数量；

depth_flag：链表(逻辑队列)中是否有完整的数据分组，1 表示有，0 表示无。

下面是队列控制器的设计代码。

```
//=====================================================================
// q_din[15:0]的数据结构:
```

```
//q_din[15]: 为尾指针指示比特,其为 1 表示当前指针指向一个帧的最后一个信元
//q_din[14]: 为头指针指示比特,其为 1 表示当前指针指向一个帧的第一个信元,此处的指针位宽
//为 16 比特,低 9 位有效
//=========================================================================
`timescale 1ns / 1ps
module qc_8_ch(
input                       clk,
input                       rstn,
//指针写入端口
input       [15:0]          q_din,
input                       q_wr,
output                      q_full,
//指针读出端口
output                      ptr_rdy,
input                       ptr_rd,
output      [15:0]          ptr_dout
);
reg    [15:0]    ptr_din;
reg    [2:0]     ptr_dlp;
reg              ptr_wr_req;
reg              q_rd;
wire   [15:0]    q_dout;
wire             q_empty;
//指针写入 FIFO,用于对写入的指针进行本地缓冲
sfifo_ft_w16_d32 u_ptr_wr_fifo (
.clk(clk),
.srst(!rstn),
.din(q_din[15:0]),
.wr_en(q_wr),
.rd_en(q_rd),
.dout(q_dout),
.full(q_full),
.empty(q_empty),
.data_count()
);
//=========================================================================
//本电路中使用了三个状态机,一个为 wr_state,用于进行链表写入申请; 一个为 rd_state,用于进
//行指针读出申请; 一个为 mstate,用于对链表进行维护。这样做是因为链表存储于 SRAM 中,不能
//同时对链表存储区进行读写操作,因此指针写入和读出操作都需要使用请求 - 应答方式
//=========================================================================
reg    [1:0]      wr_state;
reg               ptr_wr_ack;
always@(posedge clk or negedge rstn)
    if(!rstn)begin
        ptr_din<=#2  0;
        ptr_dlp<=#2  0;
        ptr_wr_req<=#2 0;
        q_rd<=#2  0;
        wr_state<=#2  0;
        end
    else begin
        //本状态机负责从输入指针 FIFO 中读出指针,以请求 - 应答方式通过 mstate
        //写入链表中
        case(wr_state)
```

```
            0:begin
                if(!q_empty)begin
                    q_rd        <= #2  1;
                    wr_state    <= #2  1;
                    end
                end
            1:begin
                q_rd            <= #2  0;
                ptr_din         <= #2  {q_dout[15:14],4'b0,q_dout[9:0]};
                ptr_dlp         <= #2  q_dout[12:10];
                ptr_wr_req      <= #2  1;
                wr_state        <= #2  2;
                end
            2:begin
                if(ptr_wr_ack)begin
                    ptr_wr_req      <= #2  0;
                    wr_state        <= #2  0;
                    end
                end
            endcase
            end
    //=====================================================================
    //ptr_rd_fifo_din: 从链表读出的指针
    //ptr_rd_fifo_ack: mstate 给 rd_state 的读请求应答信号,同时作为输出指针写入输出指针 FIFO
    //的写控制信号
    //head: 链表头指针寄存器
    //tail: 链表尾指针寄存器
    //depth: 当前链表中完整信元的个数
    //depth_pkt: 当前链表中完整数据包的个数
    //depth_flag: 当前链表中有完整数据包时为 1,否则为 0
    //depth_frame: 当前链表中完整数据包数
    //=====================================================================
    reg  [15:0]   head, tail;
    reg  [13:0]   depth,depth_pkt;
    reg           depth_flag;
    reg  [15:0]   head0,tail0;
    reg  [15:0]   head1,tail1;
    reg  [15:0]   head2,tail2;
    reg  [15:0]   head3,tail3;
    reg  [15:0]   head4,tail4;
    reg  [15:0]   head5,tail5;
    reg  [15:0]   head6,tail6;
    reg  [15:0]   head7,tail7;
    reg  [13:0]   depth0,depth1,depth2,depth3,
                  depth4,depth5,depth6,depth7;
    reg  [13:0]   depth_pkt0,depth_pkt1,depth_pkt2,depth_pkt3,
                  depth_pkt4,depth_pkt5,depth_pkt6,depth_pkt7;
    reg           depth_flag0,depth_flag1,depth_flag2,depth_flag3,
                  depth_flag4,depth_flag5,depth_flag6,depth_flag7;
    reg      [15:0]   ptr_ram_din;
    wire     [15:0]   ptr_ram_dout;
    reg               ptr_ram_wr;
    reg      [9:0]    ptr_ram_addr;
    reg      [2:0]    ptr_rd_dlp;
```

```
reg      [2:0]    ptr_rd_dlp_reg;
reg               ptr_rd_fifo_ack;
wire              ptr_rd_fifo_full;
wire              ptr_rd_fifo_empty;
reg               ptr_rd_req;
reg      [15:0]   ptr_rd_fifo_din;
reg      [3:0]    mstate;
always@(posedge clk or negedge rstn)
    if(!rstn)  begin
        mstate <= #2  0;
        ptr_ram_wr <= #2  0;
        ptr_wr_ack <= #2  0;
        ptr_rd_fifo_ack <= #2  0;
        ptr_ram_din <= #2  0;
        ptr_ram_addr <= #2  0;
        depth_flag <= #2 0;
        head <= #2 0;       tail <= #2 0;
        depth <= #2 0;      depth_pkt <= #2 0;        depth_flag <= #2 0;
        head0 <= #2 0;      tail0 <= #2 0;
        depth0 <= #2 0;     depth_pkt0 <= #2 0;       depth_flag0 <= #2 0;
        head1 <= #2 0;      tail1 <= #2 0;
        depth1 <= #2 0;     depth_pkt1 <= #2 0;       depth_flag1 <= #2 0;
        head2 <= #2 0;      tail2 <= #2 0;
        depth2 <= #2 0;     depth_pkt2 <= #2 0;       depth_flag2 <= #2 0;
        head3 <= #2 0;      tail3 <= #2 0;
        depth3 <= #2 0;     depth_pkt3 <= #2 0;       depth_flag3 <= #2 0;
        head4 <= #2 0;      tail4 <= #2 0;
        depth4 <= #2 0;     depth_pkt4 <= #2 0;       depth_flag4 <= #2 0;
        head5 <= #2 0;      tail5 <= #2 0;
        depth5 <= #2 0;     depth_pkt5 <= #2 0;       depth_flag5 <= #2 0;
        head6 <= #2 0;      tail6 <= #2 0;
        depth6 <= #2 0;     depth_pkt6 <= #2 0;       depth_flag6 <= #2 0;
        head7 <= #2 0;      tail7 <= #2 0;
        depth7 <= #2 0;     depth_pkt7 <= #2 0;       depth_flag7 <= #2 0;
        end
    else begin
        ptr_wr_ack      <= #2  0;       //给 ptr_wr_ack 赋默认值
        ptr_rd_fifo_ack <= #2  0;       //给 ptr_rd_ack 赋默认值
        ptr_ram_wr      <= #2  0;       //给 ptr_ram_wr 赋默认值
        case(mstate)
        0:begin
            if(ptr_rd_req) begin
                case(ptr_rd_dlp)
                3'b000: begin
                    head <= #2 head0;       tail <= #2 tail0;
                    depth <= #2 depth0;     depth_pkt <= #2 depth_pkt0;
                    depth_flag <= #2 depth_flag0;
                    ptr_ram_addr <= #2 head0[9:0];
                    end
                3'b001: begin
                    head <= #2 head1;       tail <= #2 tail1;
                    depth <= #2 depth1;     depth_pkt <= #2 depth_pkt1;
                    depth_flag <= #2 depth_flag1;
                    ptr_ram_addr <= #2 head1[9:0];
```

```
            end
        3'b010: begin
            head <= #2 head2;       tail <= #2 tail2;
            depth <= #2 depth2;     depth_pkt <= #2 depth_pkt2;
            depth_flag <= #2 depth_flag2;
            ptr_ram_addr <= #2 head2[9:0];
            end
        3'b011: begin
            head <= #2 head3;       tail <= #2 tail3;
            depth <= #2 depth3;     depth_pkt <= #2 depth_pkt3;
            depth_flag <= #2 depth_flag3;
            ptr_ram_addr <= #2 head3[9:0];
            end
        3'b100: begin
            head <= #2 head4;       tail <= #2 tail4;
            depth <= #2 depth4;     depth_pkt <= #2 depth_pkt4;
            depth_flag <= #2 depth_flag4;
            ptr_ram_addr <= #2 head4[9:0];
            end
        3'b101: begin
            head <= #2 head5;       tail <= #2 tail5;
            depth <= #2 depth5;     depth_pkt <= #2 depth_pkt5;
            depth_flag <= #2 depth_flag5;
            ptr_ram_addr <= #2 head5[9:0];
            end
        3'b110: begin
            head <= #2 head6;       tail <= #2 tail6;
            depth <= #2 depth6;     depth_pkt <= #2 depth_pkt6;
            depth_flag <= #2 depth_flag6;
            ptr_ram_addr <= #2 head6[9:0];
            end
        3'b111: begin
            head <= #2 head7;       tail <= #2 tail7;
            depth <= #2 depth7;     depth_pkt <= #2 depth_pkt7;
            depth_flag <= #2 depth_flag7;
            ptr_ram_addr <= #2 head7[9:0];
            end
        endcase
        ptr_rd_dlp_reg <= #2 ptr_rd_dlp;
        mstate <= #2 4;
        end
    else if(ptr_wr_req) begin
        mstate <= #2 1;
        case(ptr_dlp)
        3'b000: begin
            head <= #2 head0;   tail <= #2 tail0;
            depth <= #2 depth0; depth_pkt <= #2 depth_pkt0;
            depth_flag <= #2 depth_flag0;
            end
        3'b001: begin
            head <= #2 head1;   tail <= #2 tail1;
            depth <= #2 depth1; depth_pkt <= #2 depth_pkt1;
            depth_flag <= #2 depth_flag1;
            end
```

```
            3'b010: begin
                head <= #2 head2;  tail <= #2 tail2;
                depth <= #2 depth2; depth_pkt <= #2 depth_pkt2;
                depth_flag <= #2 depth_flag2;
                end
            3'b011: begin
                head <= #2 head3;  tail <= #2 tail3;
                depth <= #2 depth3; depth_pkt <= #2 depth_pkt3;
                depth_flag <= #2 depth_flag3;
                end
            3'b100: begin
                head <= #2 head4;  tail <= #2 tail4;
                depth <= #2 depth4; depth_pkt <= #2 depth_pkt4;
                depth_flag <= #2 depth_flag4;
                end
            3'b101: begin
                head <= #2 head5;  tail <= #2 tail5;
                depth <= #2 depth5; depth_pkt <= #2 depth_pkt5;
                depth_flag <= #2 depth_flag5;
                end
            3'b110: begin
                head <= #2 head6;  tail <= #2 tail6;
                depth <= #2 depth6; depth_pkt <= #2 depth_pkt6;
                depth_flag <= #2 depth_flag6;
                end
            3'b111: begin
                head <= #2 head7;  tail <= #2 tail7;
                depth <= #2 depth7; depth_pkt <= #2 depth_pkt7;
                depth_flag <= #2 depth_flag7;
                end
            endcase
            end
        end
//================================================================
//状态 1、2 控制链表写入
//================================================================
1:begin
    //如果当前队列非空,将指针加入链表尾部
    if(depth[9:0]) begin
        ptr_ram_wr <= #2  1;
        ptr_ram_addr[9:0]<= #2  tail[9:0];
        ptr_ram_din[15:0]<= #2  ptr_din[15:0];
        tail <= #2 ptr_din;
        end
    //如果当前队列空,将指针同时作为链表的头和尾
    else begin
        ptr_ram_wr <= #2  1;
        ptr_ram_addr[9:0]<= #2  ptr_din[9:0];
        ptr_ram_din[15:0]<= #2  ptr_din[15:0];
        tail <= #2  ptr_din;
        head <= #2  ptr_din;
        end
    //增加当前链表信元深度,如果是一个分组的最后一个信元,则增加队列中的数据分组数
    depth <= #2 depth + 1;
```

```
        if(ptr_din[15]) begin
            depth_pkt <= #2 depth_pkt + 1;
            depth_flag <= #2 1;
            end
        mstate <= #2  2;
        end
    2:  begin
        ptr_wr_ack <= #2  1;
        case(ptr_dlp)
        3'b000: begin
            head0 <= #2 head;       tail0 <= #2 tail;
            depth0 <= #2 depth;     depth_pkt0 <= #2 depth_pkt;
            depth_flag0 <= #2 depth_flag;
            end
        3'b001: begin
            head1 <= #2 head;       tail1 <= #2 tail;
            depth1 <= #2 depth;     depth_pkt1 <= #2 depth_pkt;
            depth_flag1 <= #2 depth_flag;
            end
        3'b010: begin
            head2 <= #2 head;       tail2 <= #2 tail;
            depth2 <= #2 depth;     depth_pkt2 <= #2 depth_pkt;
            depth_flag2 <= #2 depth_flag;
            end
        3'b011: begin
            head3 <= #2 head;       tail3 <= #2 tail;
            depth3 <= #2 depth;     depth_pkt3 <= #2 depth_pkt;
            depth_flag3 <= #2 depth_flag;
            end
        3'b100: begin
            head4 <= #2 head;       tail4 <= #2 tail;
            depth4 <= #2 depth;     depth_pkt4 <= #2 depth_pkt;
            depth_flag4 <= #2 depth_flag;
            end
        3'b101: begin
            head5 <= #2 head;       tail5 <= #2 tail;
            depth5 <= #2 depth;     depth_pkt5 <= #2 depth_pkt;
            depth_flag5 <= #2 depth_flag;
            end
        3'b110: begin
            head6 <= #2 head;       tail6 <= #2 tail;
            depth6 <= #2 depth;     depth_pkt6 <= #2 depth_pkt;
            depth_flag6 <= #2 depth_flag;
            end
        3'b111: begin
            head7 <= #2 head;       tail7 <= #2 tail;
            depth7 <= #2 depth;     depth_pkt7 <= #2 depth_pkt;
            depth_flag7 <= #2 depth_flag;
            end
        endcase
        mstate <= #2 3;
        end
    3: mstate <= #2 0;
```

```
//================================================================
//                读操作相关状态
//================================================================
4:  begin
    ptr_rd_fifo_din <= #2 head;
    ptr_rd_fifo_ack <= #2 1;
    mstate          <= #2 5;
    end
5:  begin
    head            <= #2 ptr_ram_dout;
    if(depth > 1)   depth <= #2 depth - 1;
    else            depth <= #2 0;
    if(head[15]) begin
        depth_pkt <= #2 depth_pkt - 1;
        if(depth_pkt > 1)depth_flag <= #2 1;
        else depth_flag <= #2 0;
        end
    mstate  <= #2 6;
    end
6:  begin
    case(ptr_rd_dlp_reg)
    3'b000:begin
        head0 <= #2 head;
        depth0 <= #2 depth;
        depth_pkt0 <= #2 depth_pkt;
        depth_flag0 <= #2 depth_flag;
        end
    3'b001:begin
        head1 <= #2 head;
        depth1 <= #2 depth;
        depth_pkt1 <= #2 depth_pkt;
        depth_flag1 <= #2 depth_flag;
        end
    3'b010:begin
        head2 <= #2 head;
        depth2 <= #2 depth;
        depth_pkt2 <= #2 depth_pkt;
        depth_flag2 <= #2 depth_flag;
        end
    3'b011:begin
        head3 <= #2 head;
        depth3 <= #2 depth;
        depth_pkt3 <= #2 depth_pkt;
        depth_flag3 <= #2 depth_flag;
        end
    3'b100:begin
        head4 <= #2 head;
        depth4 <= #2 depth;
        depth_pkt4 <= #2 depth_pkt;
        depth_flag4 <= #2 depth_flag;
        end
    3'b101:begin
        head5 <= #2 head;
        depth5 <= #2 depth;
```

```
                depth_pkt5 <= #2 depth_pkt;
                depth_flag5 <= #2 depth_flag;
                end
            3'b110:begin
                head6 <= #2 head;
                depth6 <= #2 depth;
                depth_pkt6 <= #2 depth_pkt;
                depth_flag6 <= #2 depth_flag;
                end
            3'b111:begin
                head7 <= #2 head;
                depth7 <= #2 depth;
                depth_pkt7 <= #2 depth_pkt;
                depth_flag7 <= #2 depth_flag;
                end
            endcase
            mstate <= #2 0;
            end
        endcase
        end
//================================
//          指针链表存储区
//================================
sram_w16_d1k u_ptr_ram (
  .clka(clk),
  .wea(ptr_ram_wr),
  .addra(ptr_ram_addr[9:0]),
  .dina(ptr_ram_din),
  .douta(ptr_ram_dout)
);
//=============================================================================
//本状态机在输出指针 FIFO 非满,链表中有完整的数据帧时申请按照优先级读出指针并缓冲在 FIFO
//中,供外部电路读取
//=============================================================================
reg      [1:0]      rd_state;
wire     [7:0]      depth_flag_req;
assign   depth_flag_req =   {depth_flag0,depth_flag1,
                             depth_flag2,depth_flag3,
                             depth_flag4,depth_flag5,
                             depth_flag6,depth_flag7};
wire last_cell_ptr;
assign  last_cell_ptr = ptr_rd_fifo_din[15];
always@(posedge clk or negedge rstn)
    if(!rstn) begin
        ptr_rd_req    <= #2 0;
        ptr_rd_dlp    <= #2 0;
        rd_state <= #2 0;
        end
    else
        begin
        case(rd_state)
        0:begin
            if((depth_flag_req > 8'b0)& !ptr_rd_fifo_full) begin
                casex(depth_flag_req)
```

```
                8'bxxxx_xxx1: ptr_rd_dlp <= #2 7;
                8'bxxxx_xx10: ptr_rd_dlp <= #2 6;
                8'bxxxx_x100: ptr_rd_dlp <= #2 5;
                8'bxxxx_1000: ptr_rd_dlp <= #2 4;
                8'bxxx1_0000: ptr_rd_dlp <= #2 3;
                8'bxx10_0000: ptr_rd_dlp <= #2 2;
                8'bx100_0000: ptr_rd_dlp <= #2 1;
                8'b1000_0000: ptr_rd_dlp <= #2 0;
                endcase
                ptr_rd_req    <= #2 1;
                rd_state      <= #2 1;
                end
            end
        1:begin
            if(ptr_rd_fifo_ack)begin
                ptr_rd_req    <= #2  0;
                if(last_cell_ptr) rd_state <= #2  3;
                else rd_state  <= #2 2;
                end
            end
        2:begin
            if(!ptr_rd_fifo_full)begin
                ptr_rd_req    <= #2 1;
                rd_state  <= #2 1;
                end
            end
        3:rd_state <= #2  0;
        endcase
        end
    //输出缓冲,使用的是 fall through 模式的 FIFO
    sfifo_ft_w16_d32 u_ptr_rd_fifo (
    .clk    (clk),
    .srst   (!rstn),
    .din    (ptr_rd_fifo_din[15:0]),
    .wr_en  (ptr_rd_fifo_ack),
    .rd_en  (ptr_rd),
    .dout   (ptr_dout[15:0]),
    .full   (ptr_rd_fifo_full),
    .empty  (ptr_rd_fifo_empty),
    .data_count  ()
    );
    assign  ptr_rdy = !ptr_rd_fifo_empty;
    endmodule
```

对于 qc_8_ch.v 的仿真分析,可以和整个交换单元共同进行,这里不再单独给出。

5.2.3　switch_core 电路

本节将介绍队列管理器电路的工作机制。队列管理器电路的结构在图 5-3 中已经给出,其中的写入状态机和读出状态机控制着队列管理器 switch_core 的写入和读出操作。

写入状态机用于接收并存储前级传输的信元,主要完成申请自由指针、进行多播计数、

存储信元、将自由指针写入对应的 QC 等工作。读出状态机用于轮询各输出端口对应的 QC，读出待发送分组的信元指针，从数据存储区中将信元读出并发送给后级电路，主要完成读取输出信元指针、读取信元、修改多播计数器、归还自由指针等工作。

写入和读出状态机中都会涉及对多播计数器的操作，这里做进一步的说明。数据包在前级电路中，通过路由查找，可以得到输出端口映射位图并存放于本地头中。在队列管理器中，对于单播数据包，每将一个信元从输出端口输出后，都会立即将对应的指针写入自由指针队列。当遇到多播信元时，每将多播信元输出一次，都需要根据多播计数值做进一步的判断以决定是否应该归还指针。例如，信元 a 需要从 4 个端口中输出，当该信元从端口 1 输出后，我们难以确定其是否已经从其余 3 个端口输出过了。此时，需要使用多播计数器记录输出次数。例如，某信元输入时，输出端口位图为“4'b1101”，表示该信元需要从端口 0、端口 2 和端口 3 输出。在申请到自由指针后，我们在多播计数器中以该自由指针为地址，写入计数值 3(表示需要从 3 个端口输出)。每当该信元从某个端口输出一次，就将多播计数器中的数值减 1。当计数器值减 1 后为 0 时，表明该信元已经从三个端口输出，此时应当归还指针。需要说明的是，在信元写入缓冲区时，其对应的指针应该被写入 3 个输出端口对应的队列控制器中。

下面是支持 4 个输出端口，每个端口包括 8 个优先级队列的队列管理器代码。

```
`timescale 1ns / 1ps
module switch_core_pri(
input                          clk,
input                          rstn,
//=============================================================================
//下面给出的是与前级电路的接口信号,前级电路在本电路反压为 0(i_cell_bp 为 0)时,可以连续地
//将一个完整的数据帧对应的信元写入本电路内部的数据 FIFO 中,然后将对应的指针按照规定的格式
//写入本电路内部的指针 FIFO 中。注意,写入的数据包长度为 64 字节的整数倍,不足 64 字节时,需
//要进行填充补足。i_cell_ptr_fifo_din 是当前数据包对应的指针,定义如下:
//i_cell_ptr_fifo_din[5:0]:数据包具有的内部信元数
//i_cell_ptr_fifo_dout[11:8]: 输出端口映射位图
//i_cell_ptr_fifo_dout[14:12]: 转发优先级
//=============================================================================
input      [127:0]      i_cell_data_fifo_din,
input                   i_cell_data_fifo_wr,
input      [15:0]       i_cell_ptr_fifo_din,
input                   i_cell_ptr_fifo_wr,
output  reg             i_cell_bp,
//=============================================================================
//下面给出的是与后级电路的接口信号,后级电路包括 4 个独立的输出端口处理电路,这些电路共用
//o_cell_fifo_din、o_cell_first 和 o_cell_last 信号,通过 o_cell_fifo_sel 确定当前输出数据属
//于哪个输出端口,o_cell_bp 是来自外部输出端口处理电路的反压信号
//=============================================================================
output  reg                o_cell_fifo_wr,
output  reg  [3:0]         o_cell_fifo_sel,
output       [127:0]       o_cell_fifo_din,
output                     o_cell_first,
output                     o_cell_last,
input        [3:0]         o_cell_bp
);
//双端口 RAM 接口信号,存储用户数据
```

```
wire    [127:0]  sram_din_a;          //sram 输入信号
wire    [127:0]  sram_dout_b;         //sram 输出信号
wire    [11:0]   sram_addr_a;         //sram a 口地址信号
wire    [11:0]   sram_addr_b;         //sram b 口地址信号
wire             sram_wr_a;           //sram a 口写信号
//输入缓冲 FIFO 接口信号,临时存储来自前级的数据包和指针
reg              i_cell_data_fifo_rd;
wire    [127:0]  i_cell_data_fifo_dout;
wire    [8:0]    i_cell_data_fifo_depth;
reg              i_cell_ptr_fifo_rd;
wire    [15:0]   i_cell_ptr_fifo_dout;
wire             i_cell_ptr_fifo_full;
wire             i_cell_ptr_fifo_empty;
reg     [5:0]    cell_number;
reg              i_cell_last;
reg              i_cell_first;
//自由指针队列接口信号
reg     [15:0]   FQ_din;              //输出信元时,寄存从 qc 模块读取的自由指针
reg              FQ_wr;
reg              FQ_rd;
reg     [9:0]    FQ_dout;             //写入信元时,寄存从 fq 模块读出的自由指针
wire    [9:0]    FQ_depth;
// ========================================================================
// sram_cnt_a、sram_cnt_b 是 2 比特的计数器,产生信元读写所需的低位地址 2'b00～2'b11,与指针
//并位,产生实际信元读写地址。
//从 sram 中读数据不需要读信号,sram_rd 在这里用作读操作指示。
// sram_rd_dv 是将 sram_rd 延迟一个时钟周期得到的信号,其为 1,表示 sram 当前输出的是有效数据
// ========================================================================
reg     [1:0]    sram_cnt_a;
reg     [1:0]    sram_cnt_b;
reg              sram_rd;
reg              sram_rd_dv;          //sram 输出数据有效指示
//写入状态机相关信号
reg     [3:0]    wr_state;            //写入状态机
reg     [3:0]    qc_portmap;
reg     [3:0]    qc_wr_ptr_wr_en;     //队列控制器写入信号
wire    [9:0]    ptr_dout_s;          //从自由指针队列中申请的指针
reg     [15:0]   qc_wr_ptr_din;       //准备写入 qc 的自由指针
//多播计数器相关信号
wire    [8:0]    MC_ram_addra;        //多播计数器 a 口地址信号
wire    [3:0]    MC_ram_dina;         //多播计数器 a 口输入
reg              MC_ram_wra;          //多播计数器 a 口读写信号
reg              MC_ram_wrb;          //多播计数器 b 口读写信号
reg     [3:0]    MC_ram_dinb;         //多播计数器 b 口输入信号
wire    [3:0]    MC_ram_doutb;        //多播计数器 b 口输出
//输入信元缓冲区,采用 fall through 模式
sfifo_ft_w128_d256 u_i_cell_fifo(
  .clk(clk),
  .srst(!rstn),
  .din(i_cell_data_fifo_din[127:0]),
  .wr_en(i_cell_data_fifo_wr),
  .rd_en(i_cell_data_fifo_rd),
  .dout(i_cell_data_fifo_dout[127:0]),
  .full(),
```

```
  .empty(),
  .data_count(i_cell_data_fifo_depth[8:0])
);
always @(posedge clk)
    i_cell_bp <= #2 (i_cell_data_fifo_depth[8:0]> 161) | i_cell_ptr_fifo_full;
//输入指针缓冲区,采用 fall through 模式
sfifo_ft_w16_d32 u_ptr_fifo (
  .clk(clk),                          // input clk
  .srst(!rstn),                       // input rst
  .din(i_cell_ptr_fifo_din),          // input [15 : 0] din
  .wr_en(i_cell_ptr_fifo_wr),         // input wr_en
  .rd_en(i_cell_ptr_fifo_rd),         // input rd_en
  .dout(i_cell_ptr_fifo_dout),        // output [15 : 0] dout
  .full(i_cell_ptr_fifo_full),        // output full
  .empty(i_cell_ptr_fifo_empty),      // output empty
  .data_count()                       // output [5 : 0] data_count
);
// ================================================== //
//                    写入控制状态机
// ================================================== //
wire     [2:0]    i_cell_pri;
reg      [2:0]    i_cell_pri_reg;
assign   i_cell_pri =   i_cell_ptr_fifo_dout[13:11];
parameter        PRI7_TH = 10'd32,
                 PRI6_TH = 10'd64,
                 PRI5_TH = 10'd96,
                 PRI4_TH = 10'd128,
                 PRI3_TH = 10'd160,
                 PRI2_TH = 10'd192,
                 PRI1_TH = 10'd224,
                 PRI0_TH = 10'd256;
wire[9:0]pri_th;
assign  pri_th =   (i_cell_pri[2:0] == 3'b000)? PRI0_TH:
                   (i_cell_pri[2:0] == 3'b001)? PRI1_TH:
                   (i_cell_pri[2:0] == 3'b010)? PRI2_TH:
                   (i_cell_pri[2:0] == 3'b011)? PRI3_TH:
                   (i_cell_pri[2:0] == 3'b100)? PRI4_TH:
                   (i_cell_pri[2:0] == 3'b101)? PRI5_TH:
                   (i_cell_pri[2:0] == 3'b110)? PRI6_TH:PRI7_TH;
wire      fq_bp;
reg       fq_bp_reg;
assign    fq_bp = (FQ_depth <= pri_th)?1:0;
wire qc_ptr_full0, qc_ptr_full1, qc_ptr_full2, qc_ptr_full3;
wire [3:0]qc_ptr_bull;
assign    qc_ptr_bull = {qc_ptr_full3, qc_ptr_full2, qc_ptr_full1, qc_ptr_full0};
always@(posedge clk or negedge rstn)
    if(!rstn)
        begin
        wr_state    <= #2  0;
        FQ_rd       <= #2  0;
        MC_ram_wra  <= #2  0;
        sram_cnt_a  <= #2  0;
        i_cell_data_fifo_rd <= #2 0;
        i_cell_ptr_fifo_rd <= #2 0;
```

```
        qc_wr_ptr_wr_en <= #2 0;
        qc_wr_ptr_din <= #2 0;
        FQ_dout <= #2 0;
        qc_portmap <= #2 0;
        cell_number <= #2 0;
        i_cell_last <= #2 0;
        i_cell_first <= #2 0;
        fq_bp_reg <= #2 0;
        end
    else begin
        MC_ram_wra <= #2 0;
        FQ_rd <= #2  0;
        qc_wr_ptr_wr_en <= #2  0;
        i_cell_ptr_fifo_rd <= #2  0;
        case(wr_state)
        0:begin
            sram_cnt_a <= #2  0;
            i_cell_last <= #2 0;
            i_cell_first <= #2 0;
            if(!i_cell_ptr_fifo_empty & (qc_ptr_bull == 4'b0))begin
                i_cell_data_fifo_rd <= #2  1;
                i_cell_ptr_fifo_rd  <= #2  1;
                qc_portmap          <= #2  i_cell_ptr_fifo_dout[11:8];
                fq_bp_reg           <= #2  fq_bp;
                FQ_rd               <= #2  !fq_bp;
                FQ_dout             <= #2  ptr_dout_s;
                cell_number[5:0]    <= #2  i_cell_ptr_fifo_dout[5:0];
                i_cell_pri_reg      <= #2  i_cell_ptr_fifo_dout[14:12];
                i_cell_first        <= #2  1;
                if(i_cell_ptr_fifo_dout[5:0] == 6'b1) i_cell_last <= #2 1;
                wr_state <= #2 1;
                end
            end
        1:begin
            cell_number <= #2 cell_number - 1;
            sram_cnt_a <= #2 1;
            //注意: 写入队列控制器中的指针的[15:14]两个比特分别源自 i_cell_last 和
            //i_cell_first 两个信号
            qc_wr_ptr_din <= #2  { i_cell_last,i_cell_first,
                                   1'b0,i_cell_pri_reg[2:0],
                                   FQ_dout[9:0]};
            if(qc_portmap[0])qc_wr_ptr_wr_en[0]<= #2  !fq_bp_reg;
            if(qc_portmap[1])qc_wr_ptr_wr_en[1]<= #2  !fq_bp_reg;
            if(qc_portmap[2])qc_wr_ptr_wr_en[2]<= #2  !fq_bp_reg;
            if(qc_portmap[3])qc_wr_ptr_wr_en[3]<= #2  !fq_bp_reg;
            MC_ram_wra <= #2  !fq_bp_reg;
            wr_state <= #2  2;
         end
        2:begin
            sram_cnt_a <= #2 2;
            wr_state <= #2 3;
         end
        3:begin
            sram_cnt_a <= #2 3;
```

```
                wr_state <= #2 4;
            end
            4:begin
                i_cell_first <= #2  0;
                if(cell_number) begin
                    FQ_rd        <= #2  !fq_bp_reg;
                    FQ_dout      <= #2  ptr_dout_s;
                    sram_cnt_a   <= #2  0;
                    wr_state     <= #2  1;
                    if(cell_number == 1) i_cell_last <= #2 1;
                    else i_cell_last <= #2 0;
                    end
                else begin
                    i_cell_data_fifo_rd <= #2 0;
                    wr_state            <= #2 0;
                    end
                end
            default:wr_state <= #2  0;
            endcase
            end
assign  sram_wr_a = i_cell_data_fifo_rd & !fq_bp_reg;
assign  sram_addr_a = {FQ_dout[9:0],sram_cnt_a[1:0]};
assign  sram_din_a = i_cell_data_fifo_dout[127:0];
assign  MC_ram_addra = FQ_dout[8:0];
assign  MC_ram_dina  =  qc_portmap[0] + qc_portmap[1] + qc_portmap[2] + qc_portmap[3];
//=============================================================================
//                        读出控制状态机
//=============================================================================
reg      [3:0]    rd_state;
wire     [15:0]   qc_rd_ptr_dout0, qc_rd_ptr_dout1, qc_rd_ptr_dout2, qc_rd_ptr_dout3;
reg      [1:0]    RR;
reg      [3:0]    ptr_ack;
wire     [3:0]    ptr_rd_req_pre;
wire     ptr_rdy0, ptr_rdy1, ptr_rdy2, ptr_rdy3;
wire     ptr_ack0, ptr_ack1, ptr_ack2, ptr_ack3;
assign   ptr_rd_req_pre = {ptr_rdy3, ptr_rdy2, ptr_rdy1, ptr_rdy0} & (～o_cell_bp);
assign   {ptr_ack3, ptr_ack2, ptr_ack1, ptr_ack0} = ptr_ack;
assign   sram_addr_b = {FQ_din[9:0], sram_cnt_b[1:0]};
//=============================================================================
//FQ_din 寄存的是队列控制器输出的指针,其中比特[15] 是尾信元指示信号,比特[14]是头信元指
//示信号。指出当前信元是一个完整数据帧的头信元还是尾信元,供后级电路将收到的信元重新拼
//接成完整数据包使用
//=============================================================================
assign  o_cell_last = FQ_din[15];
assign  o_cell_first = FQ_din[14];
assign  o_cell_fifo_din[127:0] =  sram_dout_b[127:0];
always@(posedge clk or negedge rstn)
    if(!rstn)begin
        rd_state <= #2  0;
        FQ_wr <= #2  0;
        FQ_din <= #2  0;
        MC_ram_wrb <= #2  0;
        MC_ram_dinb <= #2  0;
        RR <= #2  0;
```

```
        ptr_ack <= #2  0;
        sram_rd <= #2  0;
        sram_rd_dv <= #2  0;
        sram_cnt_b <= #2  0;
        o_cell_fifo_wr <= #2  0;
        o_cell_fifo_sel <= #2  0;
        end
    else begin
        FQ_wr <= #2  0;
        MC_ram_wrb <= #2  0;
        o_cell_fifo_wr <= #2 sram_rd;
        case(rd_state)
        0:begin
            sram_rd <= #2  0;
            sram_cnt_b <= #2  0;
            //当任意一个队列控制器中有准备好的数据包时,开始读出
            if(ptr_rd_req_pre)rd_state <= #2  1;
            end
        1:begin
            rd_state <= #2  2;
            sram_rd <= #2  1;
            RR <= #2 RR + 2'b01;
            //采用公平轮询的机制,轮流对 4 个端口进行发送轮询
            case(RR)
            0:begin
                casex(ptr_rd_req_pre[3:0])
                4'bxxx1:begin
                    FQ_din <= #2  qc_rd_ptr_dout0;
                    o_cell_fifo_sel <= #2  4'b0001;
                    ptr_ack <= #2  4'b0001;
                    end
                4'bxx10:begin
                    FQ_din <= #2  qc_rd_ptr_dout1;
                    o_cell_fifo_sel <= #2  4'b0010;
                    ptr_ack <= #2  4'b0010;
                    end
                4'bx100:begin
                    FQ_din <= #2  qc_rd_ptr_dout2;
                    o_cell_fifo_sel <= #2 4'b0100;
                    ptr_ack <= #2  4'b0100;
                    end
                4'b1000:begin
                    FQ_din <= #2  qc_rd_ptr_dout3;
                    o_cell_fifo_sel <= #2  4'b1000;
                    ptr_ack <= #2  4'b1000;
                    end
                endcase
                end
            1:begin
                casex({ptr_rd_req_pre[0],ptr_rd_req_pre[3:1]})
                4'bxxx1:begin
                    FQ_din <= #2  qc_rd_ptr_dout1;
                    o_cell_fifo_sel <= #2  4'b0010;
                    ptr_ack <= #2  4'b0010;
```

```
            end
        4'bxx10:begin
            FQ_din <= #2  qc_rd_ptr_dout2;
            o_cell_fifo_sel <= #2  4'b0100;
            ptr_ack <= #2  4'b0100;
            end
        4'bx100:begin
            FQ_din <= #2  qc_rd_ptr_dout3;
            o_cell_fifo_sel <= #2  4'b1000;
            ptr_ack <= #2  4'b1000;
            end
        4'b1000:begin
            FQ_din <= #2  qc_rd_ptr_dout0;
            o_cell_fifo_sel <= #2  4'b0001;
            ptr_ack <= #2  4'b0001;
            end
        endcase
    end
    2:begin
        casex({ptr_rd_req_pre[1:0],ptr_rd_req_pre[3:2]})
        4'bxxx1:begin
            FQ_din <= #2  qc_rd_ptr_dout2;
            o_cell_fifo_sel <= #2  4'b0100;
            ptr_ack <= #2  4'b0100;
            end
        4'bxx10:begin
            FQ_din <= #2  qc_rd_ptr_dout3;
            o_cell_fifo_sel <= #2  4'b1000;
            ptr_ack <= #2  4'b1000;
            end
        4'bx100:begin
            FQ_din <= #2  qc_rd_ptr_dout0;
            o_cell_fifo_sel <= #2  4'b0001;
            ptr_ack <= #2  4'b0001;
            end
        4'b1000:begin
            FQ_din <= #2  qc_rd_ptr_dout1;
            o_cell_fifo_sel <= #2  4'b0010;
            ptr_ack <= #2  4'b0010;
            end
        endcase
        end
    3:begin
        casex({ptr_rd_req_pre[2:0],ptr_rd_req_pre[3]})
        4'bxxx1:begin
            FQ_din <= #2  qc_rd_ptr_dout3;
            o_cell_fifo_sel <= #2  4'b1000;
            ptr_ack <= #2  4'b1000;
            end
        4'bxx10:begin
            FQ_din <= #2  qc_rd_ptr_dout0;
            o_cell_fifo_sel <= #2  4'b0001;
            ptr_ack <= #2  4'b0001;
            end
```

```
                4'bx100:begin
                    FQ_din <= #2  qc_rd_ptr_dout1;
                    o_cell_fifo_sel <= #2  4'b0010;
                    ptr_ack <= #2  4'b0010;
                    end
                4'b1000:begin
                    FQ_din <= #2  qc_rd_ptr_dout2;
                    o_cell_fifo_sel <= #2  4'b0100;
                    ptr_ack <= #2  4'b0100;
                    end
                endcase
                end
            endcase
            end
        2:begin
            ptr_ack      <= #2  0;
            sram_cnt_b   <= #2  sram_cnt_b + 1;
            rd_state  <= #2  3;
          end
        3:begin
            sram_cnt_b <= #2  sram_cnt_b + 1;
            MC_ram_wrb <= #2  1;
            if(MC_ram_doutb == 1)  begin
                MC_ram_dinb <= #2  0;
                FQ_wr <= #2  1;
                end
            else MC_ram_dinb <= #2  MC_ram_doutb - 1;
            rd_state <= #2  4;
          end
        4:begin
            sram_cnt_b <= #2  sram_cnt_b + 1;
            rd_state <= #2  5;
          end
        5:begin
            sram_rd <= #2  0;
            rd_state <= #2  0;
          end
        default:rd_state <= #2  0;
        endcase
        end
//=============================================================================
//例化自由指针队列管理电路
//=============================================================================
fq u_fq (
    .clk(clk),
    .rstn(rstn),
    .ptr_din({6'b0,FQ_din[9:0]}),
    .FQ_wr(FQ_wr),
    .FQ_rd(FQ_rd),
    .ptr_dout_s(ptr_dout_s),
    .ptr_fifo_depth(FQ_depth)
);
//多播计数器存储器
dpsram_w4_d512 u_MC_dpram (
```

```
  .clka(clk),
  .wea(MC_ram_wra),
  .addra(MC_ram_addra[8:0]),
  .dina(MC_ram_dina),
  .douta(),
  .clkb(clk),
  .web(MC_ram_wrb),
  .addrb(FQ_din[8:0]),
  .dinb(MC_ram_dinb),
  .doutb(MC_ram_doutb)
);
qc_8_chu_qc0(
    .clk      (clk              ),
    .rstn     (rstn             ),
    .q_din    (qc_wr_ptr_din    ),
    .q_wr     (qc_wr_ptr_wr_en[0]),
    .q_full   (qc_ptr_full0     ),
    .ptr_rdy  (ptr_rdy0         ),
    .ptr_rd   (ptr_ack0         ),
    .ptr_dout (qc_rd_ptr_dout0  )
    );
qc_8_ch u_qc1(
    .clk      (clk              ),
    .rstn     (rstn             ),
    .q_din    (qc_wr_ptr_din    ),
    .q_wr     (qc_wr_ptr_wr_en[1]),
    .q_full   (qc_ptr_full1     ),
    .ptr_rdy  (ptr_rdy1         ),
    .ptr_rd   (ptr_ack1         ),
    .ptr_dout (qc_rd_ptr_dout1  )
    );
qc_8_ch u_qc2(
    .clk      (clk              ),
    .rstn     (rstn             ),
    .q_din    (qc_wr_ptr_din    ),
    .q_wr     (qc_wr_ptr_wr_en[2]),
    .q_full   (qc_ptr_full2     ),
    .ptr_rdy  (ptr_rdy2         ),
    .ptr_rd   (ptr_ack2         ),
    .ptr_dout (qc_rd_ptr_dout2  )
    );
qc_8_ch u_qc3(
    .clk      (clk              ),
    .rstn     (rstn             ),
    .q_din    (qc_wr_ptr_din    ),
    .q_wr     (qc_wr_ptr_wr_en[3]),
    .q_full   (qc_ptr_full3     ),
    .ptr_rdy  (ptr_rdy3         ),
    .ptr_rd   (ptr_ack3         ),
    .ptr_dout (qc_rd_ptr_dout3  )
    );
//数据存储区,使用双端口 RAM 实现
dpsram_w128_d2k u_data_ram (
  .clka(clk),
```

```
  .wea(sram_wr_a),
  .addra(sram_addr_a[10:0]),
  .dina(sram_din_a),
  .douta(),
  .clkb(clk),
  .web(1'b0),
  .addrb(sram_addr_b[10:0]),
  .dinb(128'b0),
  .doutb(sram_dout_b)
);
endmodule
```

下面是 switch_core_pri. v 的仿真代码。

```
module switch_core_pri_tb;
reg             clk;
reg             rstn;
reg  [127:0]    i_cell_data_fifo_din;
reg             i_cell_data_fifo_wr;
reg  [15:0]     i_cell_ptr_fifo_din;
reg             i_cell_ptr_fifo_wr;
reg  [3:0]      o_cell_bp;
wire            i_cell_bp;
wire            o_cell_fifo_wr;
wire [3:0]      o_cell_fifo_sel;
wire [127:0]    o_cell_fifo_din;
wire            o_cell_first;
wire            o_cell_last;
//生成系统工作时钟
always #5 clk = ~clk;
switch_core_pri u_swtich_core (
    .clk(clk),
    .rstn(rstn),
    .i_cell_data_fifo_din(i_cell_data_fifo_din),
    .i_cell_data_fifo_wr(i_cell_data_fifo_wr),
    .i_cell_ptr_fifo_din(i_cell_ptr_fifo_din),
    .i_cell_ptr_fifo_wr(i_cell_ptr_fifo_wr),
    .i_cell_bp(i_cell_bp),
    .o_cell_fifo_wr(o_cell_fifo_wr),
    .o_cell_fifo_sel(o_cell_fifo_sel),
    .o_cell_fifo_din(o_cell_fifo_din),
    .o_cell_first(o_cell_first),
    .o_cell_last(o_cell_last),
    .o_cell_bp(o_cell_bp)
);
initial begin
    clk = 0;
    rstn = 0;
    i_cell_data_fifo_din = 0;
    i_cell_data_fifo_wr = 0;
    i_cell_ptr_fifo_din = 0;
    i_cell_ptr_fifo_wr = 0;
    o_cell_bp = 0;
    #100;
    rstn = 1;
```

```
        #10_000;
        send_frame(127,0,4'b0001);
        send_frame(128,1,4'b0010);
        send_frame(129,2,4'b0100);
        send_frame(1518,3,4'b1000);
        #1000;
    end
    task send_frame;
    input    [11:0]  len;
    input    [2:0]   pri;
    input    [3:0]   portmap;
    // cell_num 是内部使用的寄存器,记录当前帧包括的信元数
    reg      [5:0]   cell_num;
    reg      [5:0]   i,j;     //任务内部使用的寄存器
    begin
        i = 0;
        j = 0;
        //下面的代码根据帧长计算其包括多少个 64 字节的内部信元
        if(len[5:0] == 6'b0) cell_num[5:0] = len[11:6];
        else begin
            cell_num[5:0] = len[11:6];
            cell_num[5:0] = cell_num[5:0] + 1;
            end
        repeat(1)@(posedge clk);
        #2;
        //下面的 while 语句用于在交换单元有反压时进行等待
        while(i_cell_bp) repeat(1)@(posedge clk);
        #2;
        //=====================================================================
        //下面的循环体用于产生用户数据包并写入被测试电路中。
        //i 用于循环体控制; j 从 0 开始累加,每写入一个数据增加 1,在仿真分析时作为写入的用户数据
        //=====================================================================
        for(i = 0;i < cell_num;i = i + 1)begin
            //本设计中,第一个信元的第一个写入数据中包括了本地头,此处,本地头较为简单,由分
            //组长度值和端口映射位图组成
            if(i == 0) begin
                i_cell_data_fifo_din = {len[11:0],portmap[3:0], 112'h0};
                i_cell_data_fifo_wr = 1;
                repeat(1)@(posedge clk);
                #2;
                j = 1;
                i_cell_data_fifo_din = j;
                i_cell_data_fifo_wr = 1;
                repeat(1)@(posedge clk);
                #2;
                j = 2;
                i_cell_data_fifo_din = j;
                i_cell_data_fifo_wr = 1;
                repeat(1)@(posedge clk);
                #2;
                j = 3;
                i_cell_data_fifo_din = j;
                i_cell_data_fifo_wr = 1;
                repeat(1)@(posedge clk);
```

```
            #2;
            end
        else begin
            j = j + 1;
            i_cell_data_fifo_din = j;
            i_cell_data_fifo_wr = 1;
            repeat(1)@(posedge clk);
            #2;
            j = j + 1;
            i_cell_data_fifo_din = j;
            i_cell_data_fifo_wr = 1;
            repeat(1)@(posedge clk);
            #2;
            j = j + 1;
            i_cell_data_fifo_din = j;
            i_cell_data_fifo_wr = 1;
            repeat(1)@(posedge clk);
            #2;
            j = j + 1;
            i_cell_data_fifo_din = j;
            i_cell_data_fifo_wr = 1;
            repeat(1)@(posedge clk);
            #2;
            end
        end
    i_cell_data_fifo_wr = 0;
    i_cell_ptr_fifo_din = {1'b0,pri[2:0],portmap[3:0],2'b0,cell_num[5:0]};
    i_cell_ptr_fifo_wr = 1;
    repeat(1)@(posedge clk);
    #2;
    i_cell_ptr_fifo_wr = 0;
    i_cell_ptr_fifo_din = 0;
    end
endtask
endmodule
```

图 5-7 是 fq.v 中进行指针初始化的仿真波形。可以看出，系统复位后，FQ_state 进入状态 4，向内部的自由指针 FIFO 中写入可用的自由指针，自由指针的取值从 0 开始，一直到 511，共 512 个自由指针。

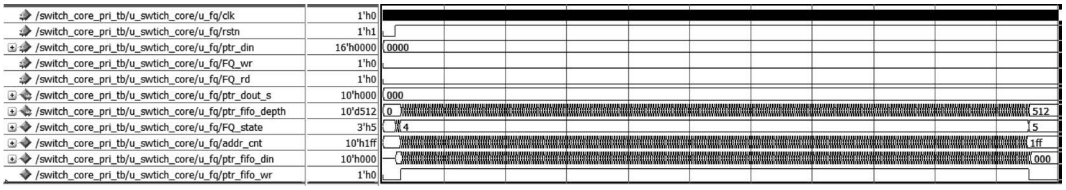

图 5-7　自由指针缓冲区初始化仿真波形

图 5-8 是从信元输入到写入主缓冲区的仿真波形，该数据包包括两个信元。从图中可以看出，i_cell_data_fifo_wr 连续 8 个时钟周期为 1，将 8 个 128 比特的数写入 switch_core_pre 模块的输入信元缓冲区中，然后通过 i_cell_ptr_fifo_din 和 i_cell_ptr_fifo_wr 写入与之

相应的指针。此后，switch_core_pre 的主状态机会读出两个自由指针，如图中所示，指针值分别为 0x000 和 0x001，然后将信元读出并通过 sram_wr_a、sram_addr_a 和 sram_din_a 接口写入信元主缓冲区中。图中的 i_cell_first 和 i_cell_last 用于指出当前信元是一个分组的首信元和尾信元。

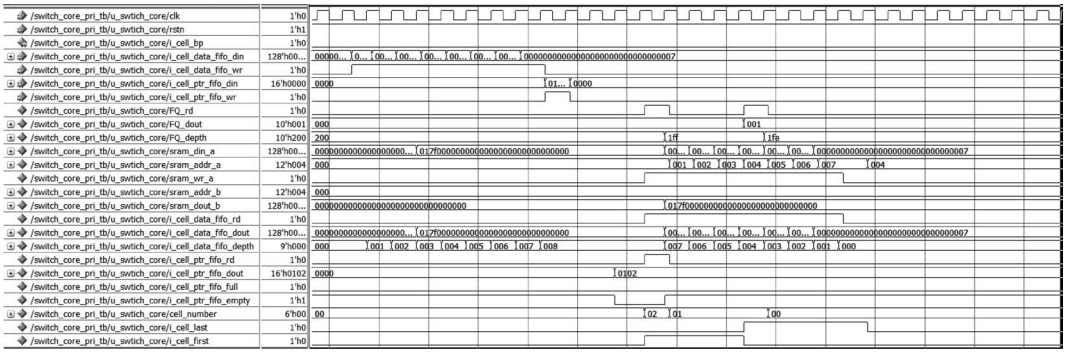

图 5-8　从信元输入到写入主缓冲区的仿真波形

图 5-9 是读出一个完整分组的仿真波形。上述包括两个信元的分组写入主缓冲区后，可以看到 ptr_rdy0 由 0 跳变为 1，switch_core_pri 内部的读出控制状态机轮询发现后，通过将 ptr_ack 置 1，将该分组首信元的指针 0x4000 读出，这里的 qc_rd_ptr_dout0[14]为 1，表示其指向一个分组的首信元。此后，switch_core_pri 中的读出控制状态机(rd_state)基于该指针生成主缓冲区读地址，将相应信元数据读出后，通过与后级的接口信号 o_cell_fifo_wr、o_cell_fifo_din、o_cell_fifo_sel 等，将数据写入相应的输出端口。此处，o_cell_fifo_sel 取值为 4'b0001，表示从端口 0 输出。同时可以看出，信元输出时，信元对应的指针通过 FQ_wr 和 FQ_din 写入自由指针队列。

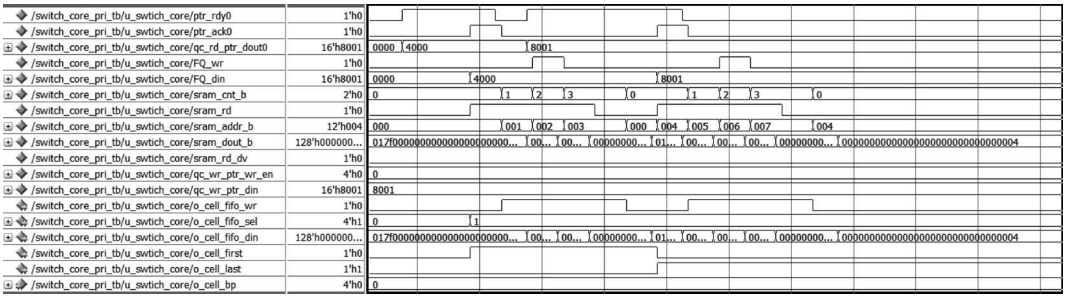

图 5-9　信元从主缓冲区读出的仿真波形

图 5-10 是写入 4 个去往不同输出端口数据包的仿真波形，通过 o_cell_fifo_sel 的取值分别为 4'h1、4'h2、4'h4、4'h8 可以看出，写入的 4 个分组分别被写入输出端口 0、1、2 和 3。

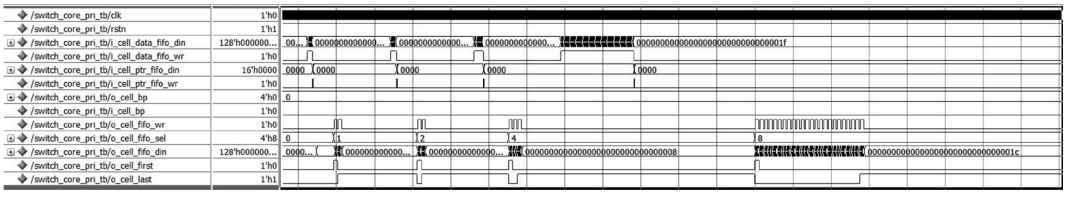

图 5-10　写入 4 个去往不同输出端口数据包的仿真波形