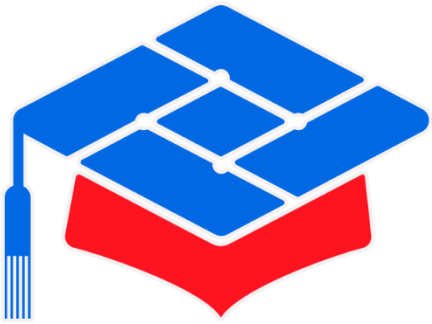

第八届

全国大学生集成电路创新创业大赛

报告类型：设计、仿真报告

参赛杯赛：中科芯杯

作品名称：Hydra-高速多端口共享缓存管理模块

队伍编号：CICC1651

团队名称：Hydra

目录

1. 概述

1.1 题目再述

赛题要求设计一款可对 SRAM 进行有效管理的 SRAM 控制器 IP。具体要求如下：

1、 支持管理至少 32 块 256K bit 的 SRAM 单元，总计至少 8Mbits 存储容量；

2、 支持 16 个端口同时进行读写操作，每个端口的传输带宽需达到 1Gbps；

3、 每个端口支持 8 个优先级队列，实现按队列进行数据缓存；

4、 支持按数据包进行缓存和调度，数据包长度范围为 64 到 1024 字节；

5、 支持多端口、多队列的动态共享缓存；

6、 支持多个端口同时写入和写出数据，每个端口独立操作，相互之间不受影响；

7、 支持数据校验，确保数据传输的准确性和完整性。

1.2 项目简介

Hydra 是一款高速多端口共享缓存管理模块，其基础功能包括但不限于：

1、 管理 16 个端口独立同时读写 32 块 SRAM；

2、 每端口 8 个队列按优先级调度数据包；

3、 所有端口所有队列动态分配缓存；

4、 对 SRAM 存储的数据进行 SEC 数据校验；

5、 提供严格优先级、WRR 两种调度模式。

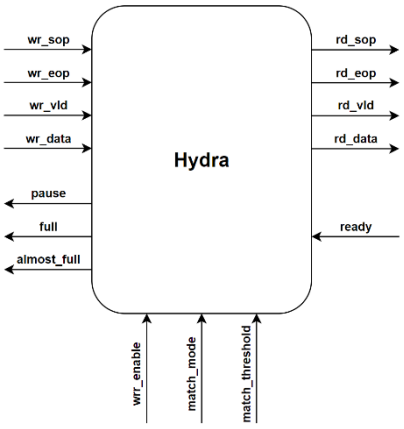

图 1 模块示意图

Hydra 在实现基础功能的同时，设计亮点包括但不限于：

1、 改进 Crossbar 架构交换机的诸多缺陷；

2、 引入跳转表的思想，降低维护优先级队列所需的资源；

3、 采用特殊的匹配机制，降低由于读出冲突导致的读延迟；

4、 添加暂停传输信号，支持断点续传；

5、 基于尾部预测的快速拼接，支持单数据包级别的边读边写；

6、 多种匹配模式支持，集合动静态分配功能于一体；

7、 高度保序，消除传统仲裁写入的缺陷；

8、 采用位掩码实现极低复杂度的 WRR 机制

9、 并行优化的数据校验加码-解码逻辑。

综上所述，Hydra 在完全实现题目基本要求的前提下，采用大量设计优化模块空间占用，降低读写延迟，提升模块性能，满足赛题背景中关于网络设备高速数据缓存的需求。

2. 基本参数

时钟频率：250Mhz

端口数量：16

SRAM 数量：32

单块 SRAM 规格：双口 256Kbits (16×16384)

读写带宽：

写入带宽（理论、测试）：3.725Gbps/端口

读出带宽（理论、测试）：3.725Gbps/端口

总计（理论、测试）：59.6Gbps

读写延迟：

按数据包头：8~24 周期+匹配阈值（可配置）

按数据包尾：0 周期（可提前/立即读出）

突发传输支持：

未拉高 pause 信号时，端口任意时刻均支持突发传输。在保持一直读出数据包的情况下，支持长期突发传输。压力测试已通过>260us 的突发传输。

存储资源占用情况（合计约 2Mbits，对应 88 块片上 BRAM）：

跳转表：32*32Kbits = 1Mbits (32*36Kbits BRAM)

空闲队列：32*22Kbits = 704Kbits (32*36Kbits BRAM)

ECC 校验信息：32*16Kbits = 512Kbits (32*18Kbits BRAM)

写入前端缓冲区：16*1Kbits = 16Kbits (16*18Kbits BRAM)

其他存储占用：< 8Kbits

功率：2.354W

3. 项目基本框架

3.1 动态缓存管理

3.1.1 SRAM 颗粒（页）划分

对于每个 SRAM，物理地址宽度为 14，要选取所有 SRAM 中一个半字需要 5+14=19 位宽的地址，不利于存储与管理。若对每半字的数据进行校验运算，由公式$2^r > n + r + 1$可知至少需要 5 位校验位，所有数据所需的校验资源高达总 SRAM 容量的 5/16。

为了便于存储地址、压缩校验功能所需的资源，Hydra 将 SRAM 的资源划分成 128 位的小颗粒，每颗粒称为 1 页，即每 8 半字为 1 页。此时指向 SRAM 中一页的地址被压缩成 14-3=11 位，选取所有 SRAM 中一页需要 5+11=16 位宽的地址，刚好与一半字对齐，便于存储。

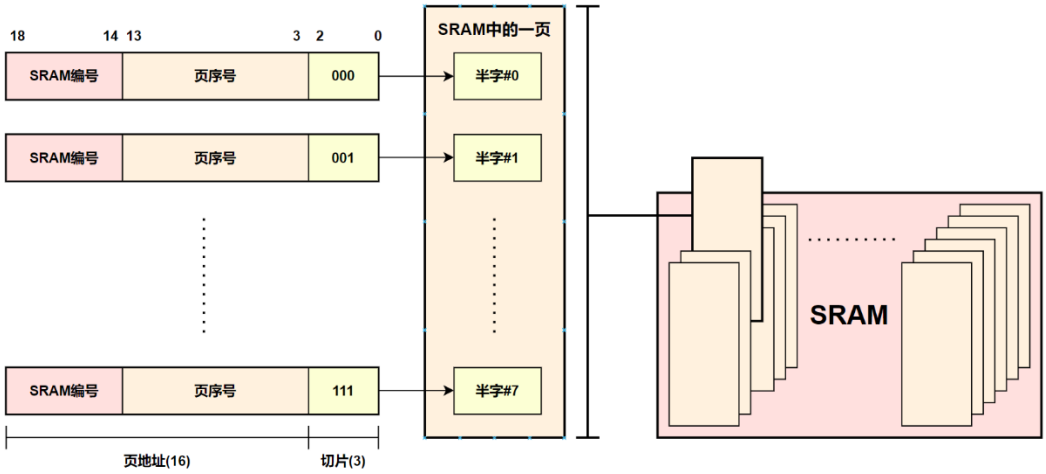

图 3.1.1(1) SRAM 颗粒划分

校验也以页为单位，根据公式，每 128 位生成 8 位校验码，生成所需时间相比于 16-5 时间上的差别可忽略不计(3 次异或运算的时间)，且校验占用的资源大幅下降，只需总 SRAM 容量的 1/16。

同时注意到一个数据包的长度折合后为 4~64 页/32~512 半字。即只需用 9 位描述一个数据包的长度，即一个数据包有多少字节。9 位长度信息与 3 位优先级信息、4 位目的端口号信息结合，刚好为一个半字(9+3+4=16)。将其作为控制信号的一部分输入，可以使数据包的控制信息与有效数据之间能清晰地划分开来，处理时无需进行二次切割、拼接，复杂度大大下降。

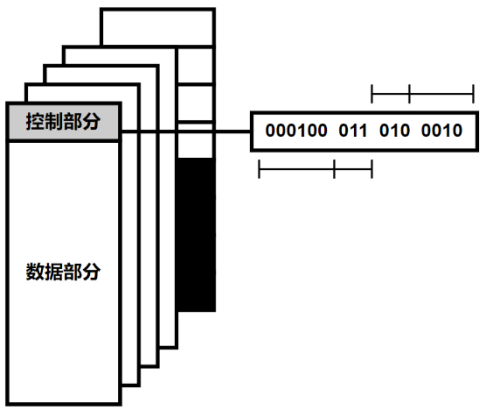

图 3.1.1(2) 数据包控制部分格式

数据包存储占用的最后一页可能并不会被使用完全，即有几半字的区域无有效数据。为了对齐页的划分，这些空缺的半字并不会被利用（但并非永远浪费，数据包读出，占用的页被回收后，它们仍可以被新的数据包装填），当数据包长度随机时，被浪费的空间小于 2%，即使数据包均为不利好页表划分的小数据包，被浪费的空间也小于 5%。

$$\text{(平均情况)}\ \frac{1}{512-32+1}\cdot\sum_{k=32}^{512}\frac{k}{8\cdot\left\lceil\frac{k}{8}\right\rceil}\times 100\% = 98.064\%$$

$$\text{(极端情况)}\ \frac{1}{128-32+1}\cdot\sum_{k=32}^{128}\frac{k}{8\cdot\left\lceil\frac{k}{8}\right\rceil}\times 100\% = 95.219\%$$

由于模块数据传输均以半字为单位，一页的数据交互需要八个周期，故需要 3 位的批次计数器（batch）记录当前处理到页中的第几个半字的数据。页地址与批次计数器拼接即可得到半字的地址，根据其线性映射关系，我们称页地址为线性地址，线性地址 11 位拼接后得到物理地址 14 位，与前面的数据是吻合的。

3.1.2 内存回收

传统的内存回收策略是为 SRAM 建立相应长度的位图（bitmap），其中每一位的数据分别对应 SRAM 中某单位（半字）是否有数据写入，1 表示被占用，0 表示未被占用，下称空闲。通过在写入数据时置 1，读出数据时置 0，即可描述 SRAM 的空闲位置，新来的数据只需直接写入空闲的位置。但是搜索位图中 0 的位置是一个时序性不良好的操作，即使经过独热码转化后，仍需通过遍历操作才能得到一个空闲位置。若要维护时序性，则需要等待较长的时间才能搜索到空闲位置，这与高速低延迟缓存管理模块的设计理念相悖。

Hydra 采用的方案是为每一个 SRAM 维护一个“空闲队列”，其本质是一个 FIFO，存储着空闲的线性地址，在写入数据时只需从队列头取出地址，读出数据时回收页，将地址插入队列尾，即可实现一个时间复杂度为 O(1)的回收机制。不过其可观的时序性需要牺牲一定资源。目前采用的空闲队列 FIFO 深度为 2048，宽度为 11（线性地址的宽度），32 块 SRAM 共需要 32*22Kb 的存储资源，合 32 块 BRAM。

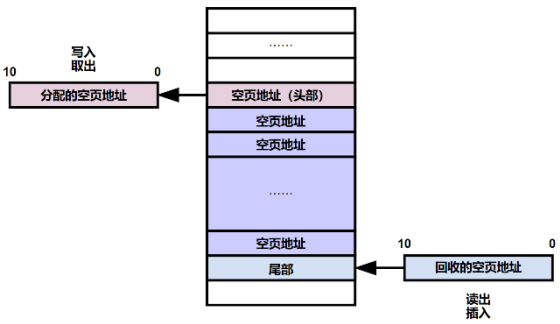

图 3.1.2 空闲队列实现内存回收与分配

Hydra 基于上述空闲队列指导的内存回收，实现了完全动态的空间分配机制，具体表现为可以在任意时刻（有空闲空间的时刻）无延时地申请一块空间，进行实时的数据存储。这使得只要还剩下空闲的空间，任意端口就可以写入数据，与传统的静态分配策略相比，带宽更高且更灵活（例如吞吐量大的端口占有更多的资源）。

3.2 数据包管理

3.2.1 链表维护优先级队列中的数据包

由于 Hydra 动态地分配内存，数据包每一部分存放的位置可能不连续，因此对于每个数据包，需要维护一个链表，记录每一页的地址。写入时通过空闲队列得到空闲页，写入数据后将其地址加入链表；读出时遍历链表中的节点即可。

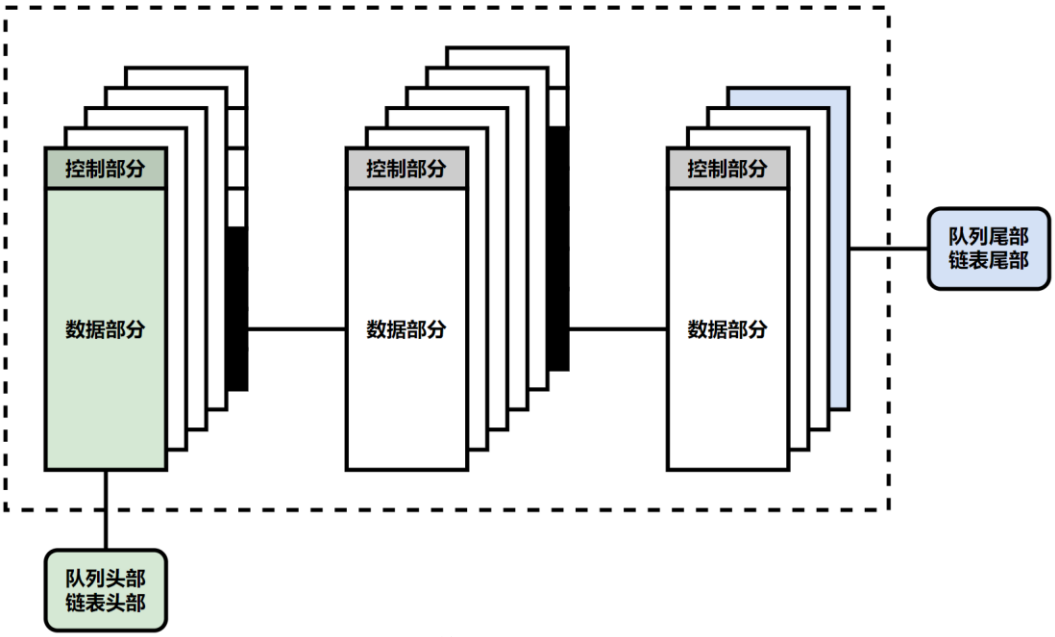

图 3.2.1 单向链表维护优先级队列

由于数据包读出时仅考虑优先级和写入时的先后顺序，因此一个优先级队列就可以用一个单向链表维护，其中存储的数据包是一条条“小链”。单向链表与传统的链表结构相比，每个结点仅存储指向下个结点的指针信息，易于管理与存储。

3.2.2 严格优先级调度

采用严格优先级调度时，端口先按队列优先级，再按写入时的先后顺序读出数据包。

3.2.3 WRR 调度

采用 WRR 调度时，端口引入了回合权重，不完全按队列优先级读出数据包。Hydra 支持一种经典的 WRR 调度，回合制处理读出队列中的数据包。对于每个端口，第一回合所有队列从高到低轮流读出数据包；第二回合优先度较高的七个队列从高到低轮流读出数据包；……以此类推，八个回合结束后从第一回合重新开始，在保证优先级高的队列享有较大读出带宽的同时，解决了低优先级被高优先级阻塞的问题。

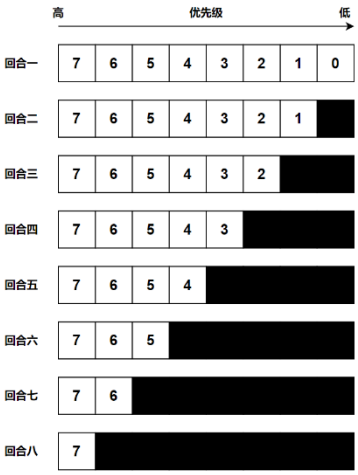

图 3.2.3 WRR 调度示意图

3.3 数据校验

3.3.1 汉明校验

Hydra 采用了(136,128)汉明校验，支持 SEC（single error correction，单错误纠错）。

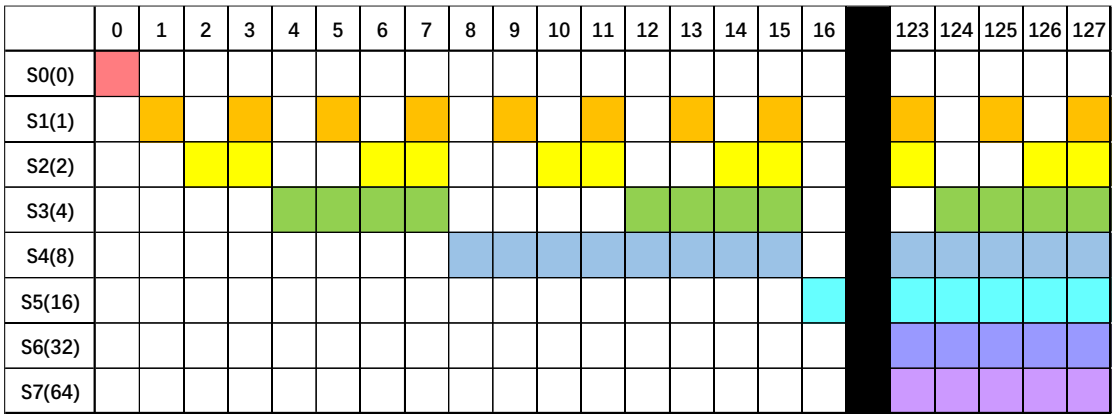

图 3.3.1 (136,128)汉明校验机制示意图（H-矩阵可视化）

3.3.2 校验信息存储

由于校验以页为单位，故校验信息的存储也以页为单位，每一个页地址指向了一个 8 位的校验码。每个 SRAM 都有一个校验存储空间，大小为 2048*8=2KB。由于同时只会有一个端口与 SRAM 进行交互，所以同时也只会有一个端口与校验存储空间进行交互，故校验存储可被置于片上资源的 Block RAM。

4. 项目亮点

4.1 结合 Crossbar 架构与总线架构

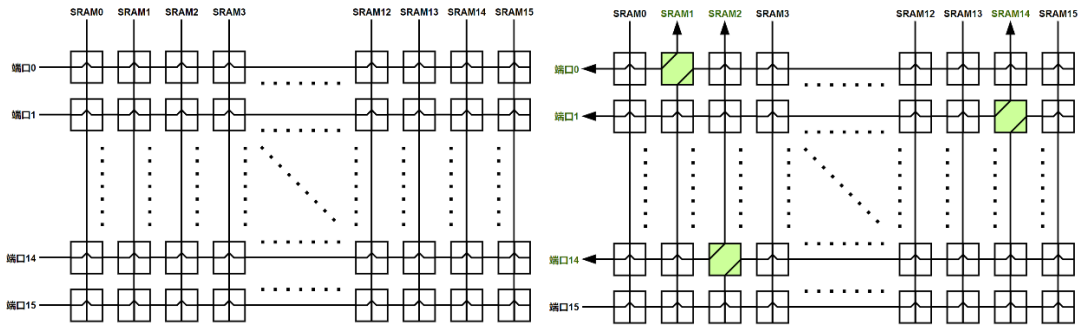

图 4.1(1) 经典 Crossbar 电路实现端口与 SRAM 间的数据传输

Hydra 借鉴了现代路由器微架构，采用 **Crossbar 架构**（见上图）实现了端口与 SRAM 之间的数据传输机制（见右图 16 个端口和 SRAM 之间自由建立的数据通道示意图），同时对 Crossbar 结构进行了一定的改进，并结合了**总线架构的优点**，包括但不限于：

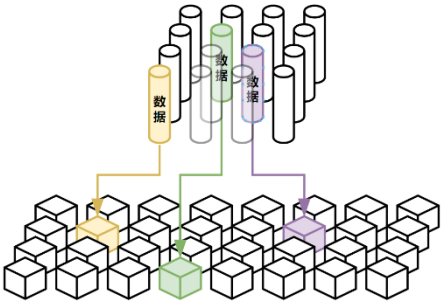

图 4.1(2) 端口与 SRAM 间自由建立的数据通道

1、 端口与 SRAM 之间可以点对点传输数据而不相互阻塞，支持各端口**独立无限制突发传输数据**直到无法找到可用的 SRAM；

2、 完全避免 Crossbar 和总线结构都具有的写入仲裁缺陷，同时有数据包写入时，Hydra 可**并行处理并传输**，无需建立仲裁机制逐个处理；

3、 点对点传输的建立也能具有较为**复杂的策略**：Hydra 在数据包写入前设置了较为复杂的匹配机制，在不影响点对点传输的前提下，缓解了读出冲突现象。

以上机制的具体实现见下文。

4.2 跳转表代替链表维护优先级队列

由于 Hydra 采用动态分配缓存的策略，故一个优先级队列中的数据包可能存放在不相邻的地方，因此需要专门存储所有数据包页地址。由于同一个优先级队列遵循先进先出的原则，可使用 FIFO 进行存储。

上述基础框架中提到的传统方案是给每个优先级队列建立一个 FIFO 管理队列中数据包的页地址，写入新的页时向 FIFO 末端插入页地址；读出页时从 FIFO 首端弹出页地址。但是由于动态分配的不平衡性，可能会存在一个优先级队列数据包极多，但是其他优先级队列几乎没有数据包的情况。若要支持最极端的条件（即所有空间都被一个优先级队列的数据包占用），每个队列的 FIFO 的深度需要 65536，每个元素宽 16 位，记载了一个带 SRAM 编号的

线性地址（5+11）。所有 FIFO 的存储资源共 128*65536*16bits=**16MB**，这是令人无法接受的。若酌情减少 FIFO 的深度，则会导致队列**有数据量限制**，**无法做到完全的动态分配**。

Hydra 采用的方案是为每个 SRAM 建立一个**“跳转表”**，基于链表数据结构的思想进一步改进传统方案中的单向链表：将队列中的页地址都视为一个结点，每个结点存储了下一结点的地址信息，这样就可以将**所有优先级队列存储在一起**，称为“跳转表”（因为每个结点都存储了跳转到下一个结点的信息）。读取队列中数据时，只需查找跳转表中当前页地址对应的内容，即可得到下一页的地址，再根据跳转表中下一页地址对应的内容，即可得到下下页的地址，以此类推，对于每个优先级队列，只需维护其队头的页地址，即可以按顺序访问队中的所有元素；向队列中新增数据时，只需将原来队尾页的跳转信息指向新的页地址，将新的页地址设置为新的末端即可。

从另一个角度看，单向链表与跳转表方案的区别在于维护队列的主体，前者是端口，后者是 SRAM，由于 SRAM 的空间固定，因此跳转表没有在极端情况下出现极高占用的弊端。

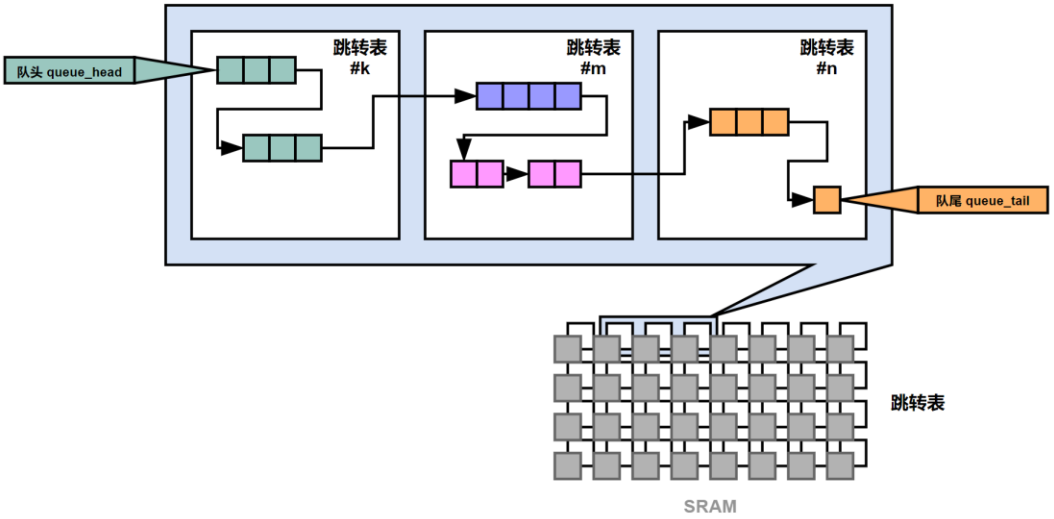

图 4.2(1) 跳转表维护队列示意图

值得注意的是，即使各端口同时独立地与 SRAM 进行高速数据交互，并不会存在同时访问/修改同一个 SRAM 的跳转表的问题，这是因为在 Crossbar 架构下，每个 SRAM 同时只会与一个端口进行交互。所以跳转表具有 RAM 的结构，从而占用非常小的电路体积。

在多个端口同时向一个优先级队列末端插入数据时，可能会有冲突的情况，因此对于每个数据包，写入第一页时，暂时不和队列末端拼接，但其后的所有页的跳转表信息正常更新，在数据包最后一页写完之后，数据包已经在跳转表中呈现为一条**“断链”**，只需将“断链”与队列末端拼接即可，原来队尾对应的页地址的跳转信息更新为“断链”首页地址，Hydra 还设计了对该入队、拼接过程加速的方案，具体请见 4.7。

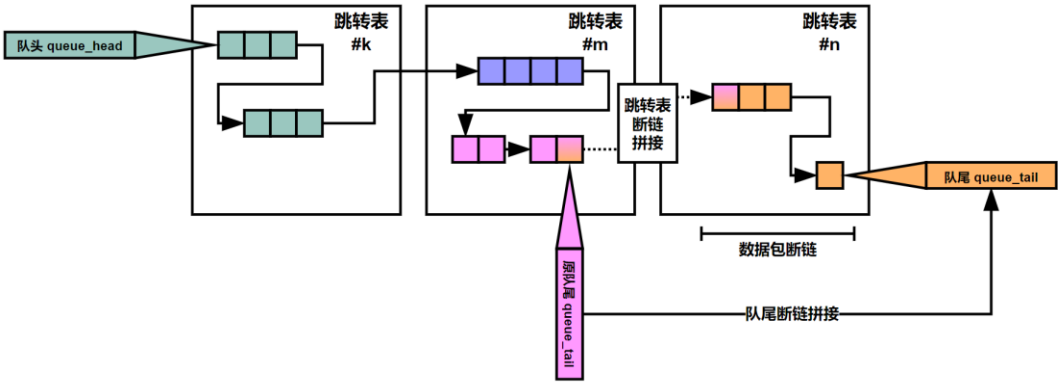

图 4.2(2) 新数据包写入时入队&拼接的过程

利用跳转表，Hydra 可以在不增加读入、写出页地址所需时间复杂度的情况下，解放队列的数据量限制，大大压缩地址管理所占用的资源。当前 Hydra 中跳转表占用资源为 32*2048*16=32*32Kbit，合 32 块 BRAM。

4.3 为数据包匹配较优 SRAM，缓解读出冲突

一个端口写入数据到 SRAM 时，一般有多个 SRAM 可用，若随机写入其中一个，随着存入缓存的数据量增多，会导致一个 SRAM 里会有去往多个端口的数据包，若这些端口同时发起读出请求，在访问该 SRAM 时会发生冲突，一些端口的请求需要等待别的端口请求完数据才被受理，从建立请求到真正获得数据，读出延迟极高（少则几百个周期多则几千个周期）。

动态分配在该方面的缺陷无法避免，端口占用资源的不平衡总会使一个 SRAM 里有不同端口的数据。即使如此，我们仍能缓解过高的读出延迟，将数据包开始存入 SRAM 前，Hydra 会为其**匹配一个较优的 SRAM**，具体的匹配规则如下：

硬性要求（不满足该规则的 SRAM 将会被忽略）

1、SRAM 容量充足：Hydra 规定一个数据包不得拆散在不同 SRAM 中，即 SRAM 剩余空闲空间必须大于等于新写入数据包的长度。

2、SRAM 未被其他端口锁定，即 SRAM 未正在被写且未被其他端口搜索过程匹配，前者是因为伪双口 SRAM 无法同时进行多次写操作，后者是因为若无此机制，不同端口可能同时认为一个 SRAM“最优”，导致写冲突，需要复杂的仲裁逻辑修正。

软性要求（越满足该规则的 SRAM 越被认为合适）

1、包含数据包目的端口的其他数据包较多；

2、包含的数据包对应目的端口的总数较少。

软性要求的目的在于尽可能把从同一端口读出的数据聚集在一起，降低一块 SRAM 极多端口数据混杂的可能性，从而缓解读取冲突。

下图是一个端口匹配的例子，红、黑、绿色 SRAM 为不符合硬性要求的 SRAM，匹

配时会直接跳过，白、蓝色 SRAM 为可匹配的 SRAM，根据上述软性规则，该端口对蓝色最深的 SRAM 偏好程度最高。

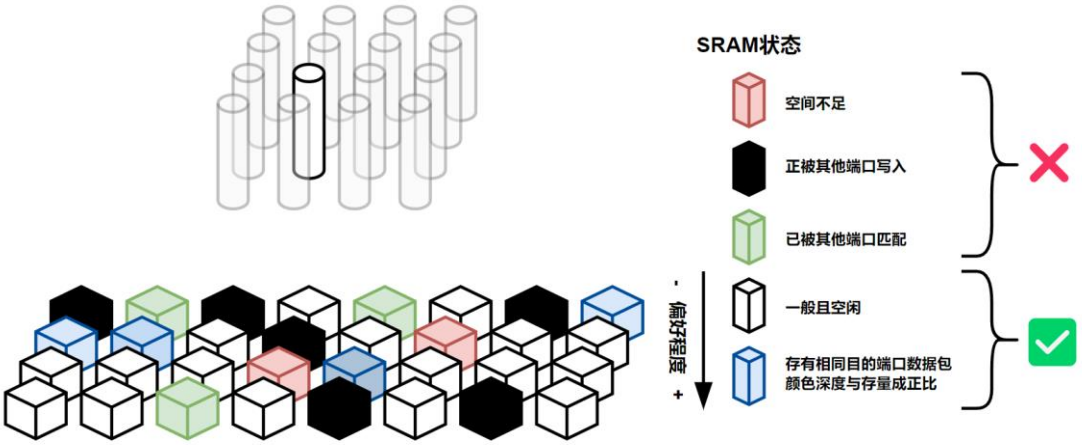

图 4.3(1) 端口匹配 SRAM 时，SRAM 状态影响着匹配结果

端口匹配 SRAM 的过程中，每个周期轮流询问一个 SRAM，并建立一个中间寄存器保存已经匹配到的最优的 SRAM。每次询问时，若硬性要求满足，则与当前已经匹配到的最优的 SRAM 对比软性要求，若新匹配的 SRAM 更优，则其成为新的最优的 SRAM。匹配完成后，可以得到较优的 SRAM，接着再启动 SRAM 的写入，具体机制见 4.4。

在保证较高的匹配效率的同时，为了维护良好的时序性，Hydra 采用**错位轮询匹配**的方式，经过巧妙的设计，Hydra 支持所有端口**无需仲裁地同时匹配 SRAM**，通过设置错位偏移量为 2，为数据选择器设置缓冲，降低匹配的组合逻辑。如下列轮询匹配中端口 0、1 轮流匹配 SRAM#3 的情况，可提前一周期缓存好匹配所需的 SRAM#3 有关的信号（MUX 周期），简化比较时（COM 周期）的组合逻辑。

全动态模式																															
PORT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1
2	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
3	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5
4	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7
5	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9
6	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11
7	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13
8	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
10	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
11	22	23	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
12	24	25	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
13	26	27	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
14	28	29	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
15	30	31	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

访问情况举例：SRAM#3与PORT#0和PORT#1																															
PORT#1	MUX	COM																													
PORT#0			MUX	COM																											

MUX: 多选器预选取
COM: 比较是否为更优的选择

图 4.3(2) 全动态分配模式下错位轮询匹配的一个回合

半动态模式																														
PORT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	1	0	3	0	5	0	7	0	9	0	11	0	13	0	15	0	17	0	19	0	21	0	23	0	25	0	27	0	29	0	31
1	2	3	2	5	2	7	2	9	2	11	2	13	2	15	2	17	2	19	2	21	2	23	2	25	2	27	2	29	2	31	2	1
2	4	5	4	7	4	9	4	11	4	13	4	15	4	17	4	19	4	21	4	23	4	25	4	27	4	29	4	31	4	1	4	3
3	6	7	6	9	6	11	6	13	6	15	6	17	6	19	6	21	6	23	6	25	6	27	6	29	6	31	6	1	6	3	6	5
4	8	9	8	11	8	13	8	15	8	17	8	19	8	21	8	23	8	25	8	27	8	29	8	31	8	1	8	3	8	5	8	7
5	10	11	10	13	10	15	10	17	10	19	10	21	10	23	10	25	10	27	10	29	10	31	10	1	10	3	10	5	10	7	10	9
6	12	13	12	15	12	17	12	19	12	21	12	23	12	25	12	27	12	29	12	31	12	1	12	3	12	5	12	7	12	9	12	11
7	14	15	14	17	14	19	14	21	14	23	14	25	14	27	14	29	14	31	14	1	14	3	14	5	14	7	14	9	14	11	14	13
8	16	17	16	19	16	21	16	23	16	25	16	27	16	29	16	31	16	1	16	3	16	5	16	7	16	9	16	11	16	13	16	15
9	18	19	18	21	18	23	18	25	18	27	18	29	18	31	18	1	18	3	18	5	18	7	18	9	18	11	18	13	18	15	18	17
10	20	21	20	23	20	25	20	27	20	29	20	31	20	1	20	3	20	5	20	7	20	9	20	11	20	13	20	15	20	17	20	19
11	22	23	22	25	22	27	22	29	22	31	22	1	22	3	22	5	22	7	22	9	22	11	22	13	22	15	22	17	22	19	22	21
12	24	25	24	27	24	29	24	31	24	1	24	3	24	5	24	7	24	9	24	11	24	13	24	15	24	17	24	19	24	21	24	23
13	26	27	26	29	26	31	26	1	26	3	26	5	26	7	26	9	26	11	26	13	26	15	26	17	26	19	26	21	26	23	26	25
14	28	29	28	31	28	1	28	3	28	5	28	7	28	9	28	11	28	13	28	15	28	17	28	19	28	21	28	23	28	25	28	27
15	30	31	30	1	30	3	30	5	30	7	30	9	30	11	30	13	30	15	30	17	30	19	30	21	30	23	30	25	30	27	30	29

访问情况举例：SRAM#3与PORT#0和PORT#1						
PORT#1	MUX	COM				MUX: 多选器预选取 COM: 比较是否为更优的选择
PORT#0			MUX	COM		

图 4.3(3) 半动态分配模式下错位轮询匹配的一个回合

同时 Hydra 支持用户**自定义匹配算法的激进程度**，即可以自定义数据包匹配 SRAM 过程的时间长度，称为**匹配阈值**。端口匹配 SRAM 过程中，每周期比较本次匹配持续时长和阈值，若时长达到阈值，且已有匹配结果，则完成匹配；如果无匹配结果，则进入超时匹配阶段，之后一旦有结果则立即完成匹配。

简单推理可知，匹配阈值较低时，平均匹配用时较少，写入延迟降低，但是可能无法得到最优的 SRAM，导致对读取冲突的缓解程度下降；匹配阈值较高时，平均匹配用时较长，写入延迟升高，但是可以尝试匹配更多的 SRAM，使得匹配结果质量上升，对读取冲突的缓解程度上升。从激进程度的角度考虑，**低阈值对应激进策略，高阈值对应保守策略**。

下图是匹配阈值等于 9 时可能出现的匹配情况，红色代表没有匹配结果，黄色、绿色代表有结果，绿色为最终匹配结果。情况 A 中，端口在第 2 周期先匹配到了某个 SRAM，后来在第 5 周期的匹配中，经过比对找到了更优的结果，最终在匹配时间达到阈值时结束匹配，发送匹配成功信号；情况 B 与 A 类似；情况 C 在达到阈值时仍未找到可用的 SRAM，进入超时匹配阶段，在第 13 周期匹配到了可用的 SRAM，同时立即结束匹配，触发匹配成功信号。

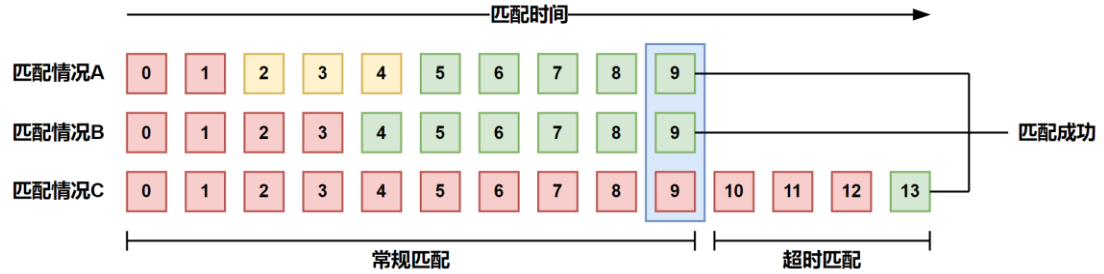

图 4.3(4) 匹配阈值等于 9 时可能出现的匹配情况举例

上述匹配机制仅需消耗数个周期即可缓解读取冲突问题，降低数百、数千周期的读取延迟，间接地大幅提升读取带宽。对于该机制造成的写入延迟（即匹配消耗的时间，一般是匹配阈值），Hydra 设置了基于尾部预测的快速拼接机制**抵消**了它，具体请见 4.5。

4. 4 设置写入前端缓冲结构，支持断点续传

由于匹配机制，数据包进入 Hydra 一段时间后才能真正开始被写入 SRAM，此时需要一个缓冲区用于存放匹配未完成时，已经进入模块的数据。

Hydra 将该缓冲区独立成模块，称为**“写入前端”**，前端与外界交互，并将数据暂时存放于缓冲区；通过建立与匹配模块的交互，在匹配结束的时启动向主模块（“后端”）的数据传输。缓冲区采用 FIFO 的结构，Hydra 使用三个指针维护缓冲区中数据的读写：xfer_ptr 和 wr_ptr 为 FIFO 的头部和尾部，end_ptr 则指向数据包结尾半字在 FIFO 中的位置，方便传输时切割不同的数据包。

下图是三个指针维护缓冲区的具体机制，数据包进入模块时，数据写入缓冲区，即图中绿色的位置，紫色 xfer_ptr 指向队头，粉色 wr_ptr 指向队尾，若匹配完成，启动前端向后端的传输过程，则对应上面一行三张图的情况，此时数据包前面已经写入缓冲区的部分随着 xfer_ptr 的移动被逐个传输至后端，后续还未写入部分同时有条不紊地进入缓冲区。数据包写入缓冲区完成后，将会留下橙色 end_ptr，当 xfer_ptr 与 end_ptr 重合时，前端将会发出数据包终止信号，并回到初始态。

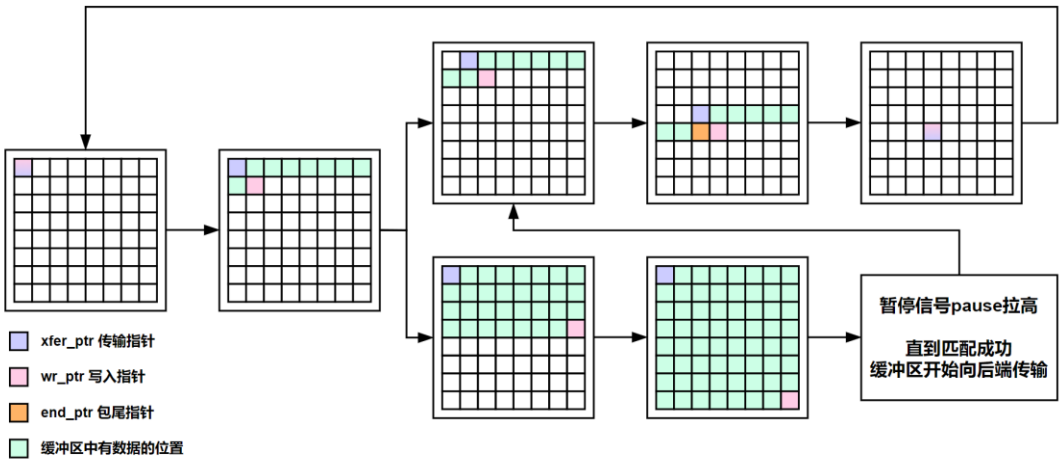

图 4.4(1) 三指针维护写入前端缓冲区的机制

当匹配超时较长时间，向后端的数据传输一直未能启动，数据将会在缓冲区中堆积，见图中第二行两张图的情况，缓冲区快满时，会对外界发送写入暂停的信号，即 pause 信号。值得一提的是，pause 信号会在完全暂停前若干拍提前拉高（见下方波形图），便于外界设备反应后及时暂停，且不用将已发送的数据冲刷/收回，可见 Hydra 具有不凡的用户友好性。pause 信号的设计使得 Hydra 支持**断点传输**：外界设备等待匹配完成，前端进入正常传输流

程，缓冲区腾出空闲空间，pause 信号拉低后仍然可以继续数据包的传输，无需重新发送整个包。

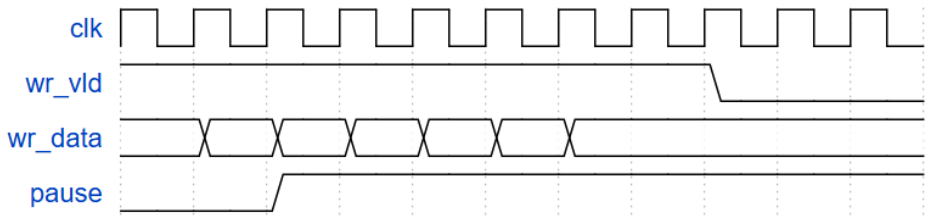

图 4.4(2) pause 信号提前拉高保证外接设备响应时数据无需冲刷/收回

4. 5 基于尾部预测的快速拼接机制，支持单包边读边写

由于匹配机制，使得数据包从外界写入到前端缓冲区，到传输至后端，真正写入 SRAM，再到入队、跳转表拼接过程完成后，变为可读取的状态之间存在若干周期的延迟，这使得一个数据包写入 Hydra 需要等待较长时间才能被读出。

详细分析上述过程每一步的延迟，可知从数据包最初写入到可以读出的时刻之间相距的总延时 Delay=传输延迟 XD+匹配阈值 T+数据包长度 L+入队拼接 C 个周期，为了缓解该延迟，Hydra 包含了**基于尾部预测的快速拼接机制**，可以**将 Delay 降低 60%~98%**。

以火车进入隧道比喻上述过程，数据包长度 L 为火车长度，传输延迟 XD+匹配阈值 T+入队拼接 C 为隧道长度，我们要得知火车的信息，实际上并不需要等待火车完全通过隧道，而只需要火车刚从隧道的另一边冒出车头时，查看车头的型号即可知道火车的信息。数据包也是一样，在数据包第一半字（控制部分）被传输至后端时，就可得知数据包的长度、优先级、目标端口。

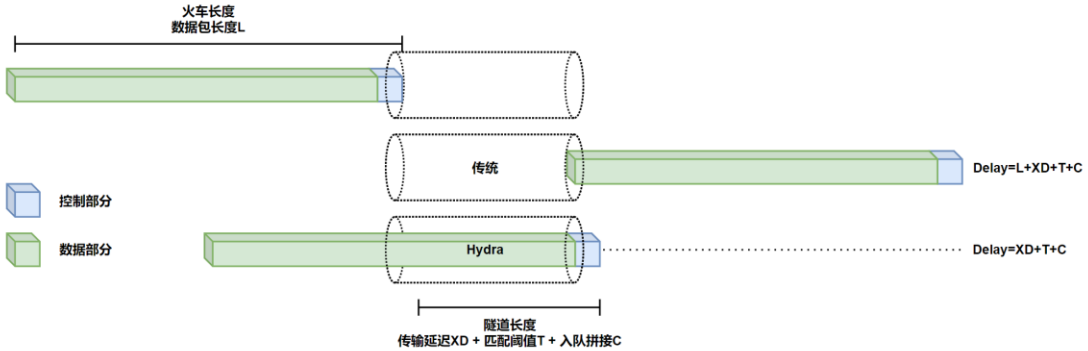

图 4.5(1) 火车进入隧道比喻 Hydra 如何降低传统数据包调度的读写延迟

又因为 Hydra 采用空闲队列回收与再分配内存（见 3.1.2），因此只需在数据包第一页写入时，将其在空闲队列中的地址与数据包长度相加即可得到数据包最后一页在空闲队列中的地址，以此提前预测数据包传输完毕后尾页地址，如右图。有已知的头和预测的尾即可完成数据包的入队、拼

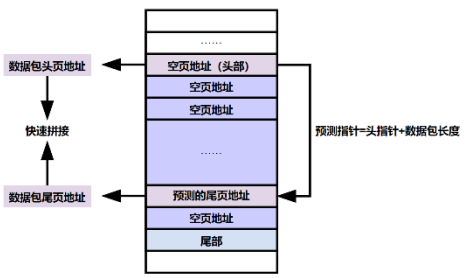

图 4.5(2) 尾部预测的具体实现

接操作，使其在完全写入完毕前就可被提前读取。

通过基于尾部预测的快速拼接机制，Hydra 直接消除了上述写入-读出延迟中的 L，而 XD+T+C 一般远小于 L。在匹配阈值小于 24 周期的情况下，几乎所有数据包都可以在外部设备写入完毕之前入队、拼接，即外部设备**写入完成后可立即读出**。Hydra **不仅抵消了匹配 SRAM 过程额外的时间开销，还使得数据包的读写更加灵活**。

4.6 多匹配模式支持，可设置全动态、半动态、静态模式

Hydra 不仅支持用户自定义匹配策略的激进程度，还支持用户定义匹配模式。目前有三种模式可用：全动态、半动态、全静态模式。

静态模式：16 个端口分别与 2 块 SRAM 绑定，不共享任何 SRAM，适用于中低速数据交换，读取延迟最低；

半动态模式：16 个端口分别与 1 块 SRAM 绑定，并共享剩余的 16 块 SRAM，适用于极端情况较少，需读取延迟较低的情况；

全动态模式：16 个端口完全共享 32 块 SRAM，适用于可能会出现极端情况的高速数据交换。

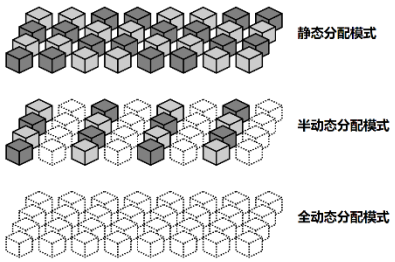

图 4.6 三种模式下缓存分配情况

值得注意的是，无论是哪种模式，Hydra 设计下的轮询匹配保证每个 SRAM 不会连续两个的周期被匹配，这使得每个端口可以提前从 SRAM 处抓取匹配过程需要的信号，降低匹配周期组合逻辑的复杂性。

4.7 保障保序性，消除传统仲裁的缺陷

当多个端口同时写入相同目的端口相同队列的数据包时，会同时向发出多个竞争的入队请求，此时需要仲裁模块逐个处理。传统仲裁具有一定缺陷，即无法判别请求的先后性。

举一个简单的例子，如右图，在某一周期端口 1、2、4 分别写入了目的端口 9 优先级 0 的数据包，第下一周期端口 3 写入了目的端口、优先级相同的数据包，假如第二周期处理完了端口 1 的请求，则此后仲裁选通信号中 2、3、4 位为高，接下来三周期依次处理 2、3、4 的请求。这时仲裁就出现了错误：本应后入队的来自端口 3 的数据包比端口 4 的数据包先被处理了，导致读出顺序错乱。

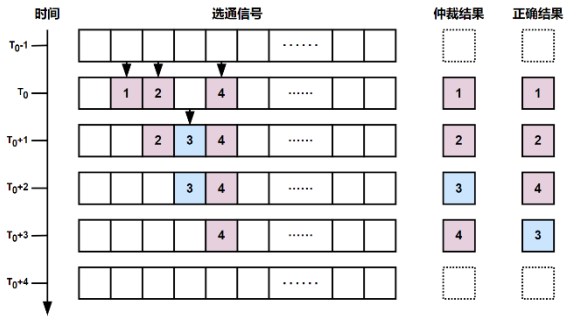

图 4.7(1) 传统仲裁的失序错误

为了维护数据包的有序性（称为**“保序”**），Hydra 引入了**时间序列**，一个存储时间戳的

FIFO 结构。在被写入数据包的 SRAM 向目的端口发起入队请求时，请求中会附带发起时间戳，主模块在有请求时将时间戳加入时间序列，每一周期根据时间序列中的顶部时间戳，生成对应的选通掩码，与选通信号经过简单逻辑仲裁得到应该处理的入队请求编号。

仍然使用刚刚的例子，第一周期由于存在请求，时间戳 T_0-1 被送入时间序列，第二周期同理，但在端口 1、2、4 的请求被处理完成前，时间序列顶一直都是 T_0-1，经过简单的判等和按位与操作，选通信号中端口 3 的请求被掩盖，从而搁置直到请求 1、2、4 处理完毕，时间序列弹出 T_0-1。

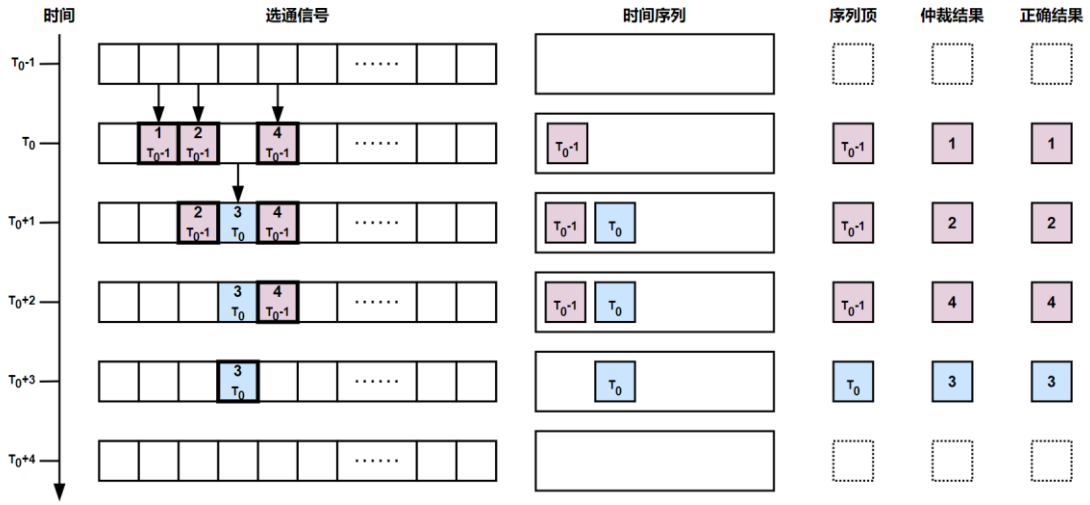

图 4.7(2) 时间序列保序机制运作原理举例

通过引入时间戳和时间序列，Hydra 保障了数据包读出的先后顺序。

4.8 “安抚”读取机制，降低读出延迟

虽然 Hydra 引入了匹配机制、基于尾部预测的快速拼接机制，但是无法完全避免读取冲突。当读取冲突发生时，传统的方案是仲裁阻塞读取，即按某种顺序处理读出请求，后续处理的请求必须等待先处理的请求完全结束之后才能开始。这会导致读取冲突时，被仲裁靠后读出的数据包，其读出延迟取决于前面读出数据包的长度，从请求刚建立（ready 拉高）到真正读出（rd_vld）延后了数百甚至上千周期。

为了缓解这种情况，Hydra 会**轮询处理读取请求**，即出现读取冲突时，每个请求**轮流处理一页**。这样每个端口将平均地享有读出带宽，从请求刚建立（ready 拉高）到真正读出（rd_vld）的延迟一般不多于 32 周期，最多仅有 128 周期（最极端情况，实际上在匹配策略下不可能出现）。由于该轮询机制类似于在“安抚”冲突时“不耐烦”的请求，故称之为**“安抚”读取机制**。

4.9 基于掩码集的无需复杂计算的快速 WRR

WRR 机制常常伴随着移位运算出现，为了避免体积庞大的移位电路，Hydra 预生成了

所有可能出现的 WRR 掩码情况（称为 WRR 掩码集）。在处理请求时，端口会生成 WRR 轮询信号，将下一次使用的 WRR 掩码切换到阵列中的指定位置，并与队列的非空信号进行简单的逻辑运算预生成下一次应当被读取的队列编号。实现了无需复杂计算的 WRR 机制。由于每次读取的队列编号都是预生成的，故 Hydra 能够在 ready 信号拉高一周期后立刻响应并发起数据包的读出请求。

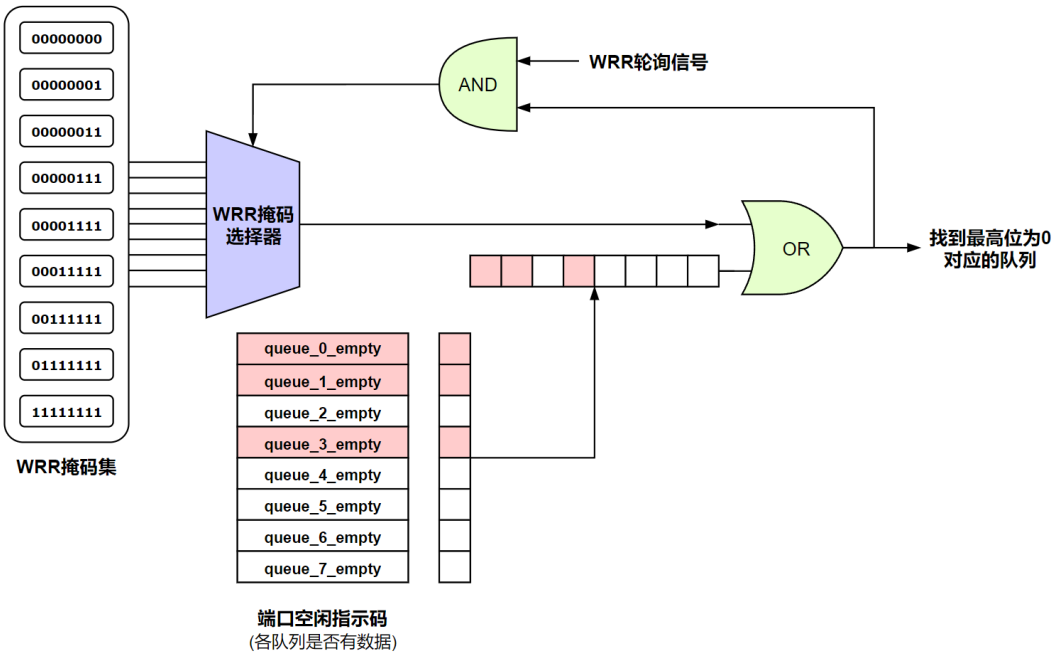

图 4.9 基于掩码集的快速 WRR

4.10 并行汉明校验，实现单周期编码解码

Hydra 根据异或操作的对称性，将串行异或运算转化为了并行异或运算，每一层执行多次异或运算并一层一层汇总，最终生成(128,8)SEC 编码。编码过程在 RTL 电路层面仅需 6 层 XOR 门，可在单周期内完成编解码。

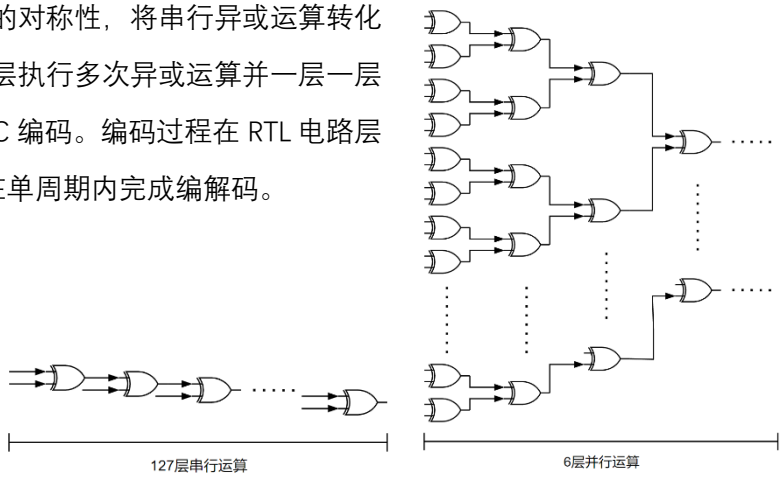

图 4.10 串并行汉明校验示意图

4.11 其他亮点

1、代码注释齐全，变量命名规范统一，大多数操作状态机化；

2、占用资源较大的结构全部 RAM 化，将被综合在 BRAM 中，降低布线压力；

3、机制流水线化，相邻请求处理可无缝衔接，无需等待信号重置；

4、严格制定每一步的逻辑复杂度，保障良好的时序严谨性；

5、采用 feature 验方式，一步一步细化进行功能调试与验证，使得 Hydra 具有良好的鲁棒性。

```
module port_wr_frontend(
    input clk,
    input rst_n,

    /*
     * 外界与前端交互的信号
     */
    input wr_sop,
    input wr_eop,
    input wr_vld,
    input [15:0] wr_data,
    output reg pause,

    /*
     * 向后端发送数据包的信号
     * |- xfer_data_vld - xfer_data是否有效
     * |- xfer_data - 当前周期传输的一半字数据
     * |- end_of_packet - 当前传输的半字是否为数据包最后半字
     */
    output ready_to_xfer,
    output reg xfer_data_vld,
    output reg [15:0] xfer_data,
    output reg end_of_packet,

    /*
     * 与匹配模块交互的信号
     * |- match_suc - 匹配完毕信号，可以开始发送缓冲区的数据
     * |- match_enable - 使能匹配进程的信号
     * |- new_dest_port, new_length - 被匹配的数据包的目标端口与长度(半字)
     *                                用于匹配时判断SRAM对该数据包的喜好程度
     */
    input match_suc,
    output reg match_enable,
    output reg [3:0] new_dest_port,
    output reg [8:0] new_length
);

/*
 * wr_state - 数据包写入前端缓冲区的状态:
 * |- 0 - 当前无数据包写入(初始态或wr_eop拉高后)
 * |- 1 - 数据包即将写入(wr_sop拉高后)
 * |- 2 - 数据包正在写入(wr_vld第一次拉高后)
 * |- 3 - 数据包完成写入(传输完毕所有半字后)
 */
reg [1:0] wr_state;
```

图 4.11 良好的代码规范

5. 模块具体实现细节

5.1 各模块说明

5.1.1 hydra(顶层模块)

功能：主控制器模块，连接所有子模块并处理核心逻辑，如匹配 SRAM，维护队列等操作。

5.1.2 port_wr_frontend

功能：根据 8 位校验码对一页（128 位）待纠错数据纠错并输出。

5.1.3 port_wr_matcher

功能：维护存储空闲页地址的“空闲队列”，实现 O(1)的内存回收机制。

5.1.4 port_rd_frontend

功能：对写入模块的数据进行初步处理并缓冲，并指导总控制模块进行匹配操作。

5.1.5 port_rd_dispatch

功能：对写入模块的数据进行初步处理并缓冲，并指导总控制模块进行匹配操作。

5.1.6 sram_state

功能：管理 ECC 存储、跳转表和 SRAM 端口数据量。

5.1.7 sram

功能：伪双口 SRAM，提供独立工作的读写口各一个。

5.1.8 ecc_encoder

功能：管理 ECC 存储、跳转表和 SRAM 端口数据量。

5.1.9 ecc_decoder

功能：管理 ECC 存储、跳转表和 SRAM 端口数据量。

5.2 重要逻辑说明

5.2.1 数据包写入流程

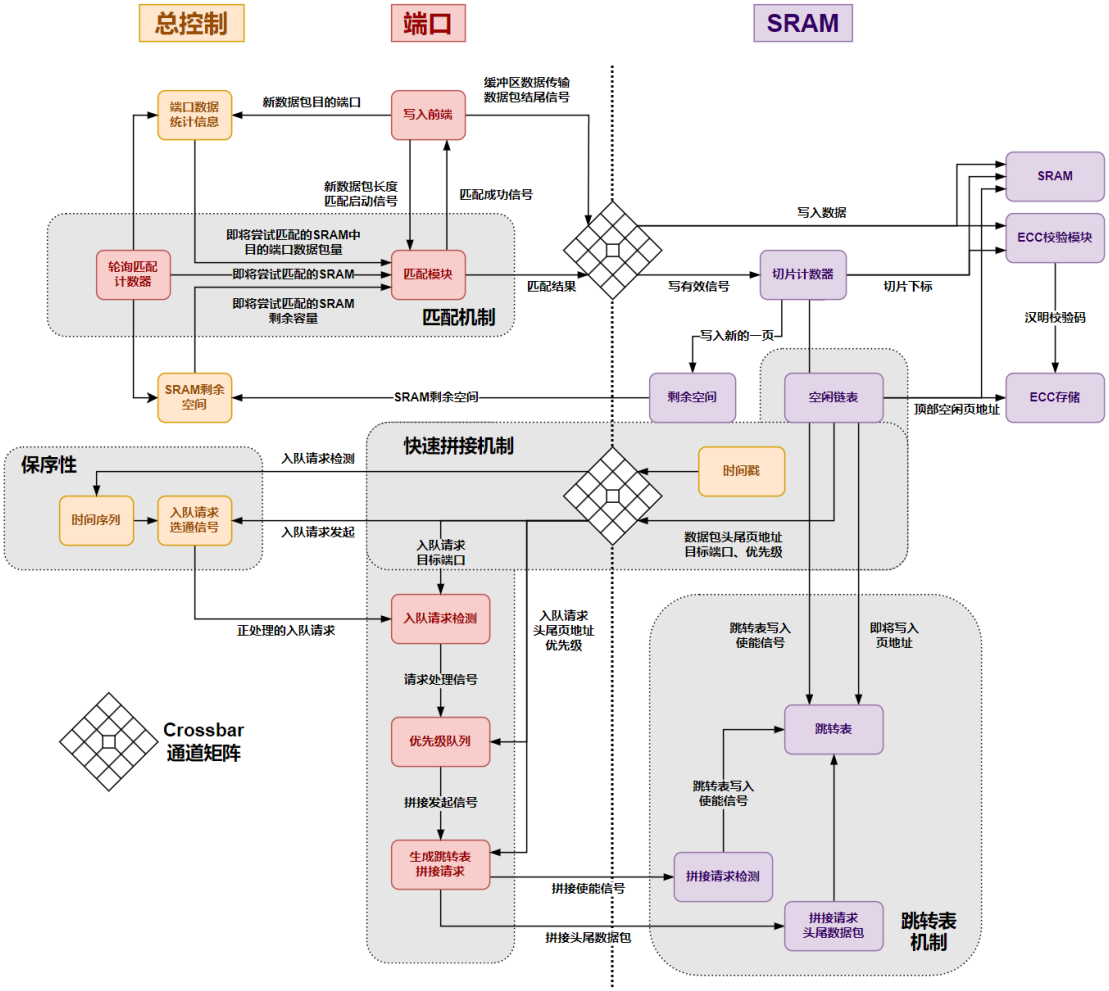

图 5.2.1 数据包写入流程核心部分示意图

5. 2. 2 数据包读出流程

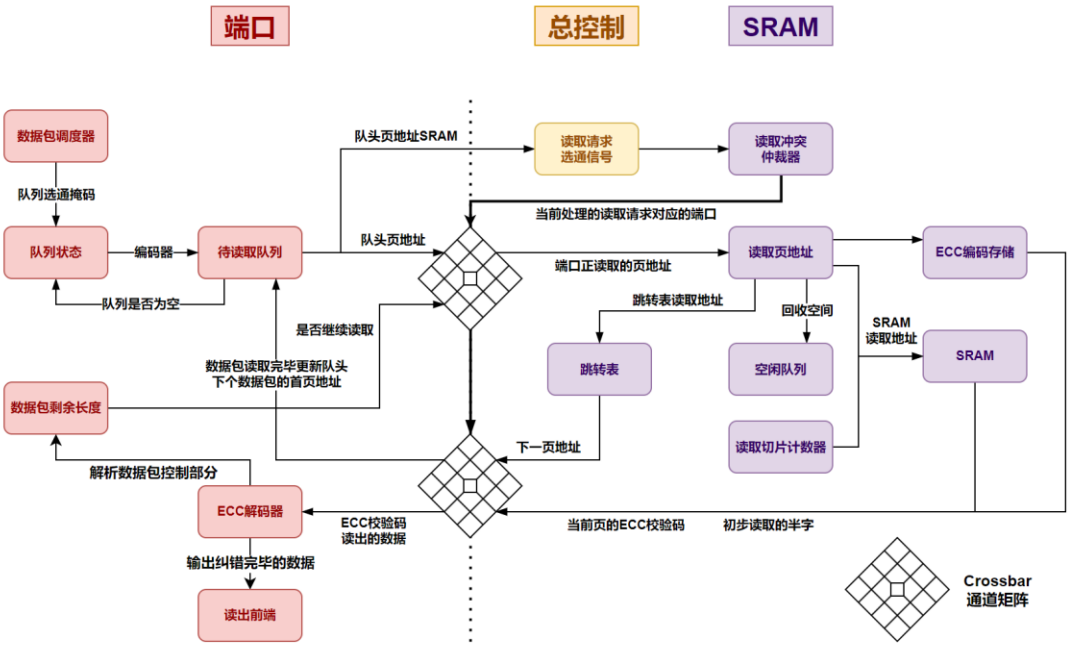

图 5.2.2 数据包读出流程核心部分示意图

6. 接口与配置

6.1. 写控制 IO 口

6.1.1 IO 口介绍

wr_sop：拉高 1 周期表示开始写入数据包。

wr_eop：拉高 1 周期表示结束写入数据包。

wr_vld：表示当前通过 wr_data 写入的一半字数据有效。

wr_data：写入一半字数据的内容，仅在 wr_vld 为高的时候被认为有效。

6.1.2 使用方法

数据包格式：

控制部分（1 半字）和数据部分（31~511 半字）。

其中控制部分为 16 位，描述三个数据包信息：

1）高 9 位表示数据包的长度（半字）；

2）低 4 位表示目的端口；

3）剩余 3 位表示优先级。

注意事项：

1）wr_sop 和 wr_eop 不能同时拉高。

2）wr_vld 只能在 wr_sop 为高后、wr_eop 为高前的一段时间内为高。

6.2 写反馈 IO 口

6.2.1 IO 口介绍

full：当所有 SRAM 被占用/满时拉高。

almost_full：当所有端口均满足以下至少一个条件时拉高：

1、 被占用/写满；

2、 剩余空间少于 25%。

pause：当前端口缓冲区被占满或上个包已写入缓冲区完毕，但仍处于超时匹配时拉高。

6.2.2 使用方法

当某个端口的 pause 拉高时，立即停止数据传输，直到 pause 拉低即可继续传输。

当 almost_full 拉高时，可以考虑拉高 ready 读出缓存中的数据。

6.3 读控制 IO 口

6.3.1 IO 口介绍

ready：请求读出一个数据包的信号。

6.3.2 使用方法

ready 拉高时，若该端口存在可读取的数据包，rd_sop 将会在下一周期立即拉高，表示开始读出数据包。读出数据包的过程中拉高 ready 无效，即只有在前一个数据包 rd_eop 拉高后，拉高 ready 才会触发后一个数据包的读出。

6.4 读反馈 IO 口

6.4.1 IO 口介绍

rd_sop：拉高 1 周期表示开始写入数据包。

rd_eop：拉高 1 周期表示结束写入数据包。

rd_vld：表示当前通过 rd_data 写入的一半字数据有效。

rd_data：写入一半字数据的内容，仅在 rd_vld 为高的时候被认为有效。

6.4.2 使用方法

数据包格式与 sop、eop、vld 和 data 信号线的机制与写入时相同。

6.5 配置选项

wrr_enable：每个端口各 1 个，表示是否开启该端口的 WRR 调度模式。

match_mode：**缓存分配模式**，0-静态分配；1-半动态分配；2/3-全动态分配。当需要高速读写且有可能出现极端情况时，建议使用全动态分配模式；需要高速读写且端口使用程度较为平均时，建议使用半动态分配模式；仅需要中低速读写时，建议使用静态分配模式。

match_threshold：**匹配阈值**，当匹配时长超过该值后，一旦检测到可用的 SRAM 即完成匹配。匹配阈值的大小涉及匹配策略的激进程度，越小匹配所需时间越短，写入延迟降低，但可能降低匹配策略对读取冲突的缓解程度；越大匹配所需时间越长，提升匹配策略对读取冲突的缓解程度，但会使写入延迟提升。静态分配模式下匹配阈值最大为 0；半动态分配模式下匹配阈值最大为 16；全动态分配模式下匹配阈值最大为 30。

7. 验证方法

7.1 RTL 级仿真

7.1.1 RTL 级电路图

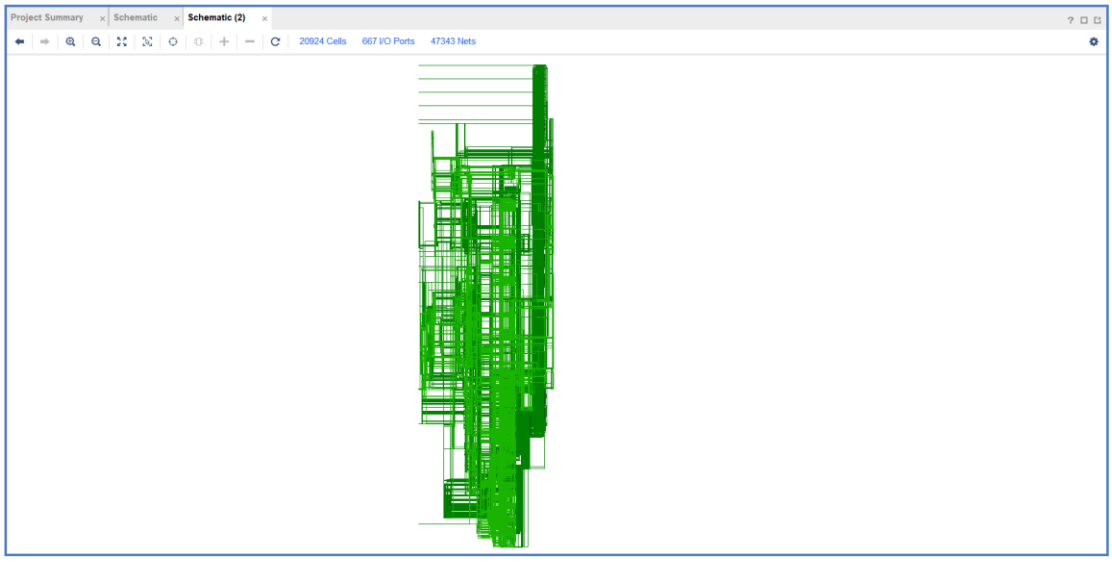

图 7.1.1 RTL 级电路图

7.1.2 Feature 验证

注：Hydra 完成了大量 Feature Test，因篇幅有限，只展示最重要测试的波形图。

1、 单数据包写入读出

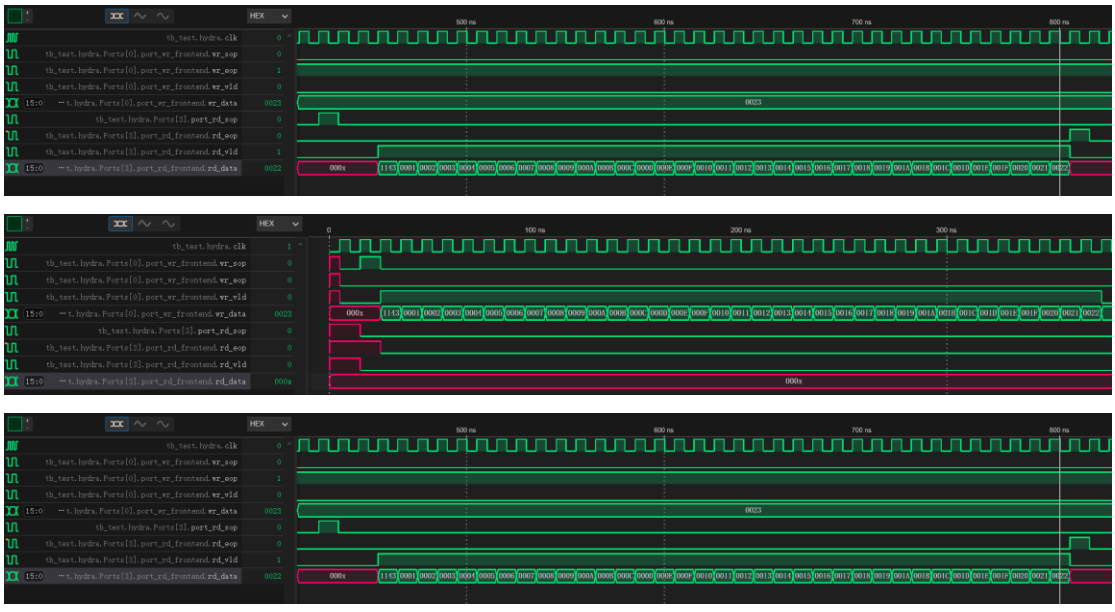

图 7.1.2(1) 单数据包写入读出

上图展示了端口 0 向端口 3 发送了优先级为 4，长度为 34 半字的数据包时，两个端口部分 IO 口的波形图，第一张图为整理情况，第二三张图为写入、读出阶段分别放大后的情况，对比 wr_data 和 rd_data 信号可验证数据传输的正确性。

2、 前端缓冲与匹配过程

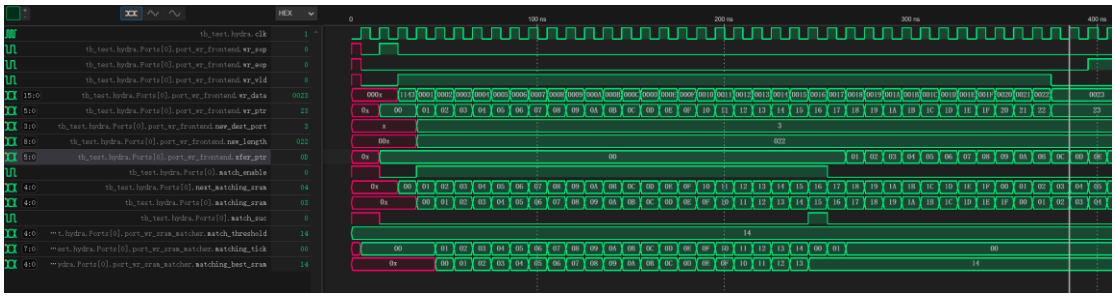

图 7.1.2(2) 数据包进入前端后触发匹配过程的波形图

上图中，端口 0 向端口 3 发送了优先级为 4，长度为 34 半字的数据包，数据包先进入前端缓冲区，对应尾指针 wr_ptr 不断往后移动。第一半字刚进入时，前端解析其中的控制信息，得到了数据包的目的端口 new_dest_port 和长度 new_length，并拉高启动匹配信号 match_enable。负责匹配的模块收到信号后，每周期匹配相应的 SRAM（matching_sram），且匹配时长 matching_tick 不断增加，最终达到匹配阈值 matching_threshold 时，发出匹配成功信号 match_suc。前端得知匹配成功后，开始像后端传输信号，对应传输指针 xfer_ptr 不断往后移动。数据包最终完整地穿过缓冲区，被传输至后端存入 SRAM。

3、 前端传输数据至后端，再进一步被写入 SRAM

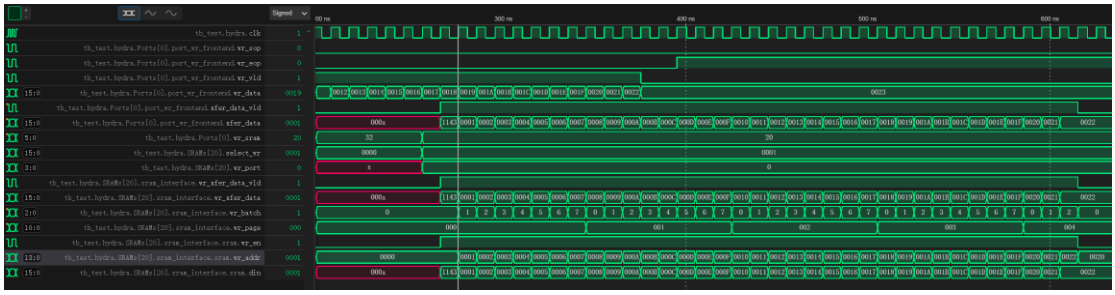

图 7.1.2(3) 数据包传输至后端并被存入 SRAM

匹配结束后，Crossbar 矩阵中端口和 SRAM 的结点会被激活，上图中来自端口 0 的数据包匹配到了 SRAM20，选通信号 select_wr 的第零位置 1，经过编码器，SRAM 得到正在向自己传输数据的端口号 wr_port，图中为 0，并通过其选择端口数据通道中的数据信号 wr_xfer_data_vld 和 wr_xfer_data，它们与前端传输至后端的数据信号同步。同时，SRAM 封装模块会生成页地址 wr_page 并启动切片计数器 wr_batch，它们组成 SRAM 的写地址 wr_addr。在数据包结束时，切片计数器清零，并停止写 SRAM，对应 wr_en 拉低。

4、入队请求的生成与时间序列管理

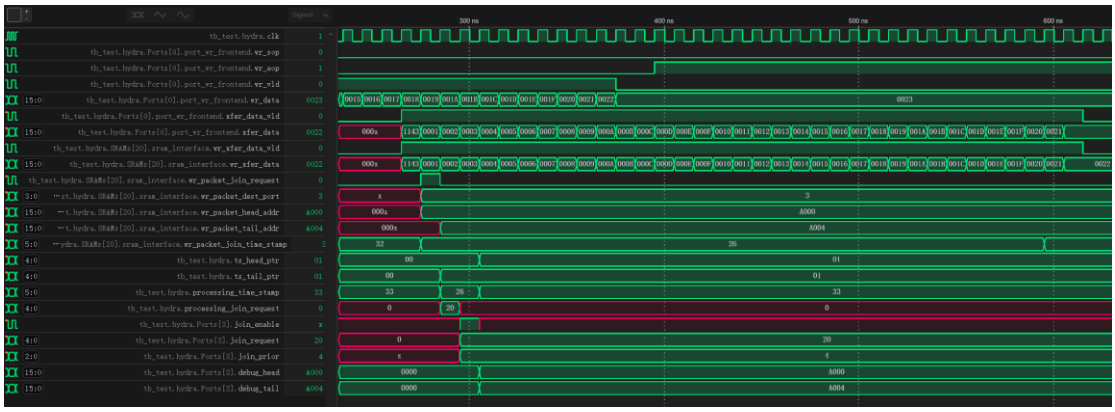

图 7.1.2(4) SRAM 生成入队请求到数据包最终成功入队过程

数据包传输至后端，被写入 SRAM 时可得知其首页地址和尾页地址，此时即可建立入队请求，图中控制信号 1143 解析完毕的下一个周期，拉高入队有效信号 wr_packet_join_request，更新数据包的目的端口（3）和头页地址 wr_packet_head_addr（A000），同时为请求盖上时间戳 wr_packet_join_time_stamp（26）。主控制器中检测到当前周期有入队请求，向时间序列中加入当前时间戳，对应 ts_tail_ptr 改变，在下一个周期观测时间序列顶部作为正在处理的时间戳 processing_time_stamp（26），经过逻辑运算和选择编码得到应处理哪个 SRAM 的请求 processing_join_request（20），最后由数据包目的端口拾起该请求，刷新请求缓存 join_enable、join_request、join_prior，完成数据包断链入队。debug_head 和 debug_tail 信号最终分别变为数据包头尾页地址（A000 ~ A004），证明了该机制的正确性。

5、基于尾部预测的快速拼接

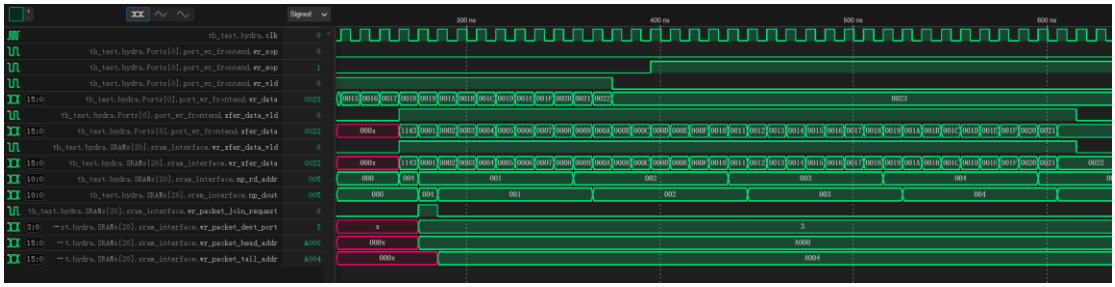

图 7.1.2(5) 基于尾部预测的快速拼接过程

数据包刚进入 SRAM 时，可通过其长度和头页地址计算得到指向空闲队列中数据包尾页的指针，此时更新空闲队列的读取地址 np_rd_addr（004），即可在下个周期预知数据包尾部地址，并将其当作入队请求的尾部 wr_packet_tail_addr（A004）。经过该过程，Hydra 得以在数据包刚被写入时就启动入队、拼接过程，大大降低读取延迟。

6、 多数据包的保序性

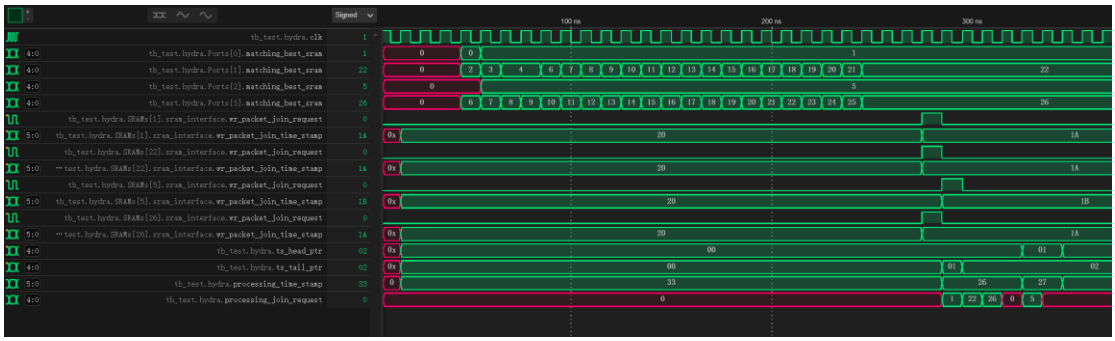

图 7.1.2(6) 一组入队请求的保序处理

上图是一个经典的容易在保序方面出错的入队请求组合，SRAM1、22、26 同时发起入队请求（数据包来自端口 0、1、3），SRAM5 在后一个周期发起入队请求（数据包来自端口 2），由于时间序列在接下来的三个周期只处理时间戳为 26 的请求，故 SRAM5 的请求会被遮盖，不会被处理直到主控制模块将时间戳为 26 的请求全部处理完毕，开始处理时间戳为 27 的请求。图中 processing_join_request 即为入队请求处理顺序（1、22、26、5）。

7、 基于快速拼接的无延迟/提前读取

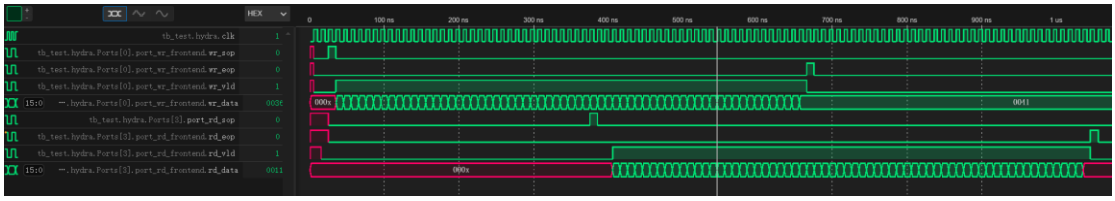

图 7.1.2(7) 提前读取数据包

上图是提前读取数据包的案例，在外界设备还没完成写入时，Hydra 就已经允许读出该数据包了。这样可以消除平均 80%的读出延迟，因为可读出实际不再需要被数据包的长度影响。

7. 1. 3 压力测试

压力测试方案：

1、单端口间歇线性映射写入共 8208 个 64 字节数据包并完全读出

2、16 个端口同时间歇线性映射写入共 8208 个 64 字节数据包并完全读出

3、16 个端口同时连续线性映射写入共 8208 个 64 字节数据包并完全读出

4、16 个端口同时连续随机映射写入共 8208 个 64 字节数据包并完全读出

5、16 个端口同时连续随机映射写入共 8208 个随机长度数据包并完全读出

7.2 硬件实现验证

7.2.1 验证环境

Vivado 版本：2018.3

芯片型号：xcku115-flvb2104-2-e

综合策略：Flow_AreaOptimized_high

布线策略：Performance_NetDelay_high

7.2.2 综合布线结果

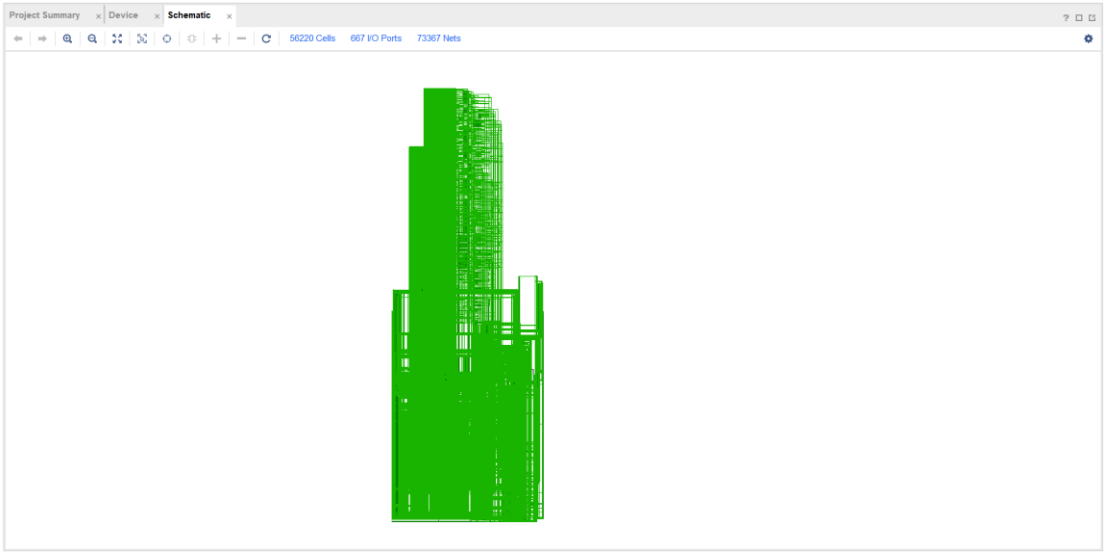

图 7.2.2(1) 布线结果电路图

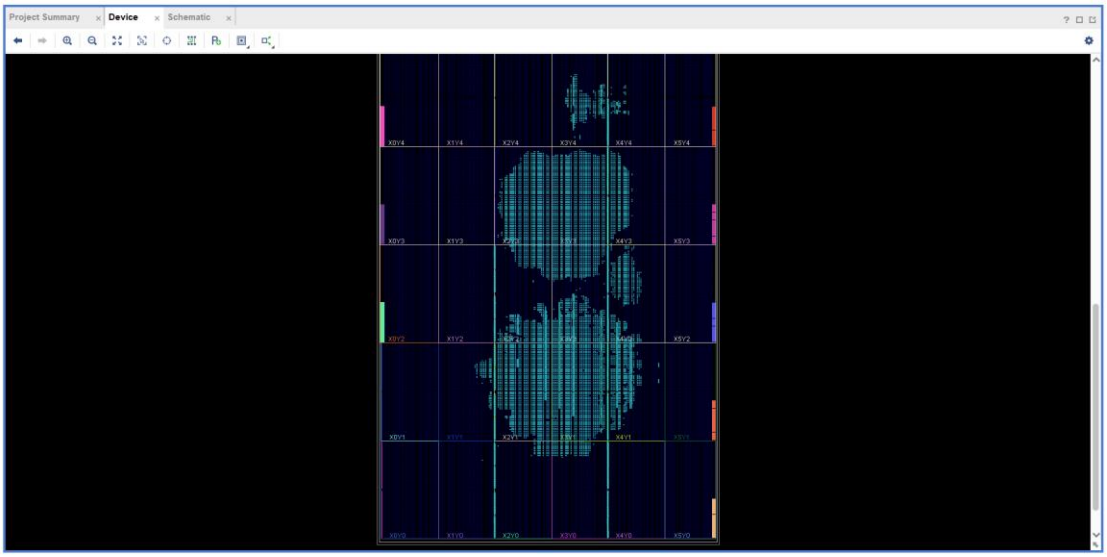

图 7.2.2(2) Device（布线阶段）

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP	Start	Elapsed	Run Strategy
synth_1	constrs_1	synth_design Complete!								52796	22334	88.00	0	0	7/23/24, 3:50 PM	00:14:39	Flow_AreaOptimized_high (Vivado Synthesis 2018)
impl_2	constrs_1	route_design Complete!	0.310	0.000	0.020	0.000	0.000	2.354	0	52544	22350	88.00	0	0	7/23/24, 4:13 PM	00:28:50	Performance_NetDelay_high (Vivado Implementation 2018)

图 7.2.2(3) 综合、布线结果

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**2.354 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.9°C**
Thermal Margin:	73.1°C (86.5 W)
Effective ϑJA:	0.8°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

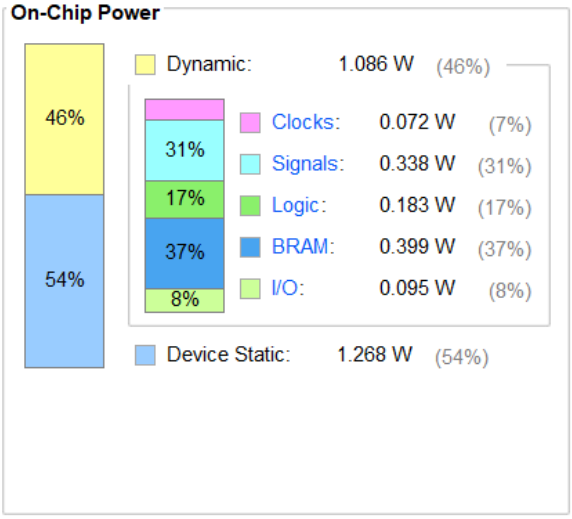

图 7.2.2(4) Power Summary（布线阶段）

Hierarchy

Name	CLB LUT...	Block RAM Tile (2160)	Bonded IOB (7...	HPIOB (598)	HRIO (104)	GLOBAL CLOCK BUFFERs (1248)	CLB Register...	CARRY8 (82920)	F7 Muxe...	F8 Muxe...	CLB (82...	LUT as Logic ...	LUT as Memo...
hydra	52544	88	667	563	104	1	22350	336	16120	5220	8743	52536	8
decoder_32_5 (decoder_32_5)	20	0	0	0	0	0		0	0	0	10	20	0
decoder_concatenate (decoder_16_4_124)	589	0	0	0	0	0		0	64	32	254	589	0
Ports[0].port_rd_dispatch (port_rd_dispatch)	133	0	0	0	0	0		0	5	0	40	133	0
Ports[0].port_rd_frontend (port_rd_frontend)	4	0	0	0	0	0		0	0	0	4	4	0
Ports[0].port_wr_frontend (port_wr_frontend)	551	0.5	0	0	0	0		2	0	0	233	551	0
Ports[0].port_wr_sram_matcher (port_wr_sram_matcher)	21	0	0	0	0	0		1	0	0	7	21	0
Ports[1].port_rd_dispatch (port_rd_dispatch_24)	133	0	0	0	0	0		0	5	0	39	133	0
Ports[1].port_rd_frontend (port_rd_frontend_25)	3	0	0	0	0	0		0	0	0	3	3	0
Ports[1].port_wr_frontend (port_wr_frontend_26)	59	0.5	0	0	0	0		2	0	0	14	59	0
Ports[1].port_wr_sram_matcher (port_wr_sram_matcher_27)	20	0	0	0	0	0		1	0	0	8	20	0
Ports[2].port_rd_dispatch (port_rd_dispatch_28)	131	0	0	0	0	0		0	16	0	41	131	0
Ports[2].port_rd_frontend (port_rd_frontend_29)	4	0	0	0	0	0		0	0	0	3	4	0
Ports[2].port_wr_frontend (port_wr_frontend_30)	58	0.5	0	0	0	0		2	0	0	14	58	0
Ports[2].port_wr_sram_matcher (port_wr_sram_matcher_31)	20	0	0	0	0	0		1	0	0	7	20	0
Ports[3].port_rd_dispatch (port_rd_dispatch_32)	133	0	0	0	0	0		0	16	0	40	133	0
Ports[3].port_rd_frontend (port_rd_frontend_33)	4	0	0	0	0	0		0	0	0	4	4	0
Ports[3].port_wr_frontend (port_wr_frontend_34)	58	0.5	0	0	0	0		2	0	0	14	58	0
Ports[3].port_wr_sram_matcher (port_wr_sram_matcher_35)	19	0	0	0	0	0		1	0	0	7	19	0
Ports[4].port_rd_dispatch (port_rd_dispatch_36)	133	0	0	0	0	0		0	16	0	39	133	0
Ports[4].port_rd_frontend (port_rd_frontend_37)	4	0	0	0	0	0		0	0	0	4	4	0
Ports[4].port_wr_frontend (port_wr_frontend_38)	59	0.5	0	0	0	0		2	0	0	15	59	0
Ports[4].port_wr_sram_matcher (port_wr_sram_matcher_39)	20	0	0	0	0	0		1	0	0	6	20	0

图 7.2.2(5) Utilization（布线阶段）

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.310 ns	Worst Hold Slack (WHS):	0.020 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	50555	Total Number of Endpoints:	50555

All user specified timing constraints are met.

图 7.2.2(6) Timing Summary（布线阶段）

7.3 FPGA 验证

7.3.1 Ecc 校验编解码模块

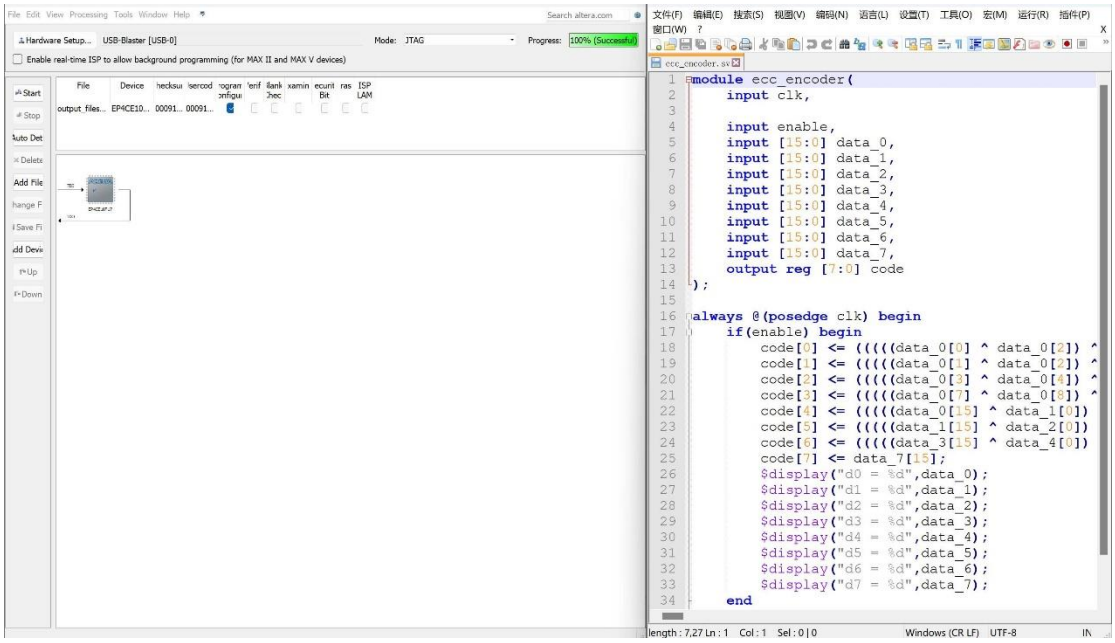

图 7.3.1(1) 程序烧录截图

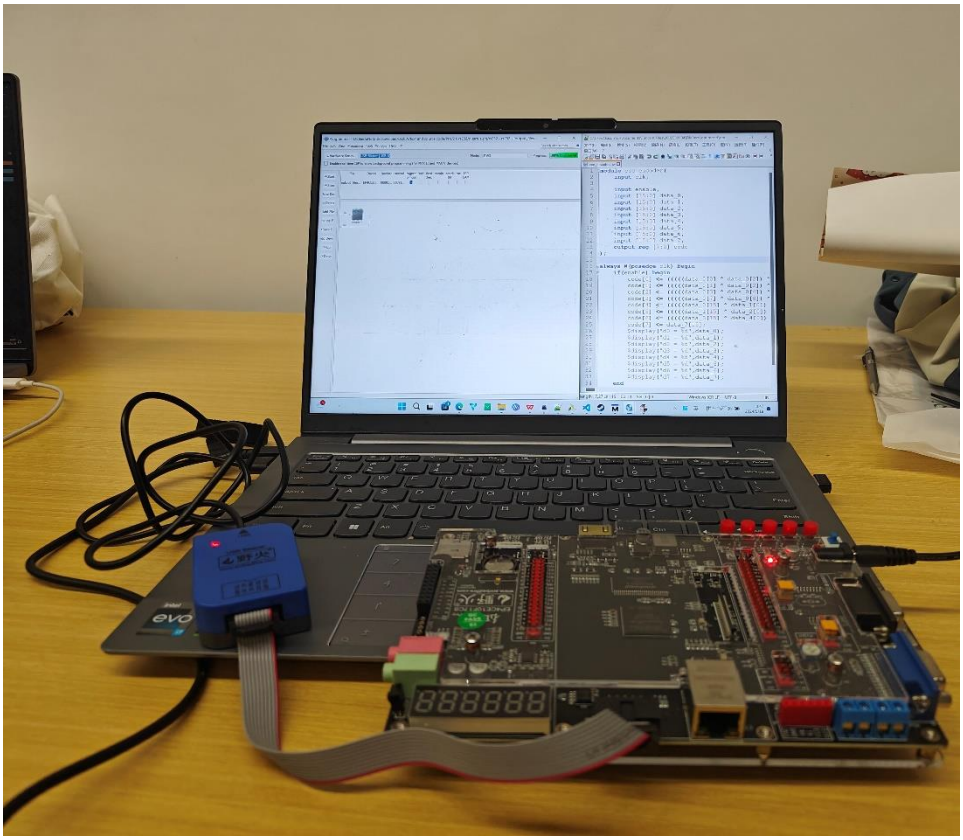

图 7.3.1(2) FPGA 板运行图

8. 设计优缺点

8.1 优点

1）完全动态分配内存，各端口、队列数据量无限制；

2）引入数据结构与算法思想，降低逻辑复杂度；

3）引入页表管理的思想，进一步划分 SRAM，减少了存储时需要的控制信息；

4）时序性良好，使用跳转表替代时序性差的位图，各模块机制流水化；

5）偏好性存储降低读取延迟，使得大多数情况下端口读取带宽可以达到最大值；

6）在有可用 SRAM 时保证端口写入带宽总为最大值；

7）极大压缩模块的存储资源占用（相较于传统方案而言）；

8）极低的读写延迟，在写入未完毕时即可开始读出；

9）代码风格良好，严格遵循命名规范，注释清晰。

8.2 缺点

1）Crossbar 架构对模块布线压力稍大。

注：若压力稍大，可以选择降频。由于 Crossbar 架构的优越性，Hydra 在低频率下带宽仍能远超总线高频率下的带宽，具体数据见下表。

频率(MHz)	总带宽(Gbps)	
	Crossbar架构	总线架构
25	5.96	0.37
50	11.92	0.75
100	23.84	1.49
125	29.80	1.86
200	47.68	2.98
250	59.60	3.73
400	95.37	5.96

图 8.2 SRAM 宽度为 16 时两种架构在不同频率下的理论带宽

9. 后续开发计划

9.1 进一步优化读写延迟

继续简化读写逻辑，降低传输延迟 XD。

改良快速拼接机制，降低入队拼接延迟 C。

9.2 更全面的压力测试

模拟更加极端的情况，测试时间更长，吞吐量更高的情况。

编写全自动压力测试脚本，高级语言按照测试策略随机生成输入数据，testbench 读取输入数据并保存模块输出至文件，高级语言再读取输出文件进行对拍。

9.2 继续提升代码质量

进一步整理代码，丰富注释。