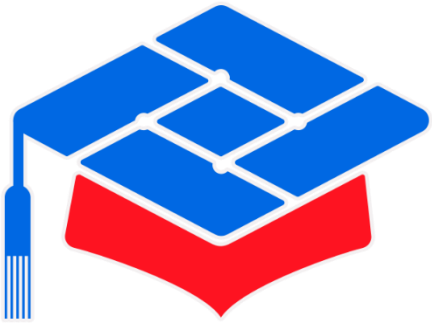


第八届

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，如右图。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；

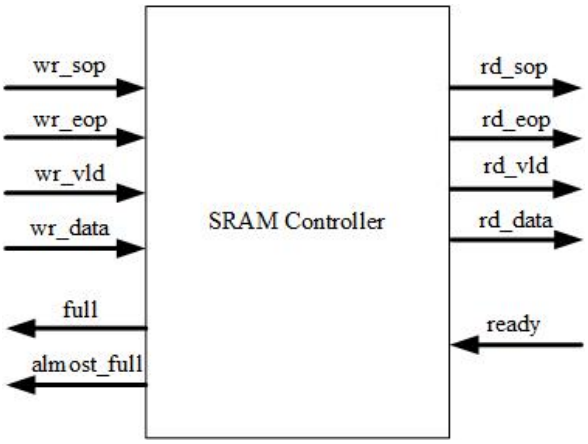


图 1.1 IP 示意图

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 缓存；

2、 每端口 8 个优先级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。

Hydra 在实现以上基础功能的同时，设计亮点有：

1、 引入页表管理的思想，优化逻辑的复杂程度，压缩模块空间占用；

2、 采用特殊的搜索机制，降低由于读出冲突导致的读延迟；

3、 引入链表的思想，降低维护优先级队列所需的资源；

4、 所有操作均流水化，在维护良好时序严谨性的同时提升性能；

5、 并行优化的数据校验加码-解码逻辑；

6、 采用位掩码实现极低复杂度的 WRR 机制。

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，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：伪双口/真双口 256Kbits (16×16384)

写入带宽：3.7Gbps/端口

写入延迟：32~48 周期

读出带宽：0.23~3.7Gbps/端口 （注：大多数情况下可跑满 3.7Gbps）

读出响应延迟：12~140 周期（注：大多数情况下仅需下限 12 周期）

存储资源占用情况（合计约 2Mbits，320 块片上 BRAM）：

跳转表：32*32Kbits = 1Mbits

空闲队列：32*22Kbits = 704Kbits

ECC 校验信息：32*16Kbits = 512Kbits

其他存储占用：< 8Kbits

3. 项目框架与策略

3.1 缓存管理与分配

3.1.1 页表管理

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（原因见第三节跳转表）。

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，将 SRAM 的资源进一步划分，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。

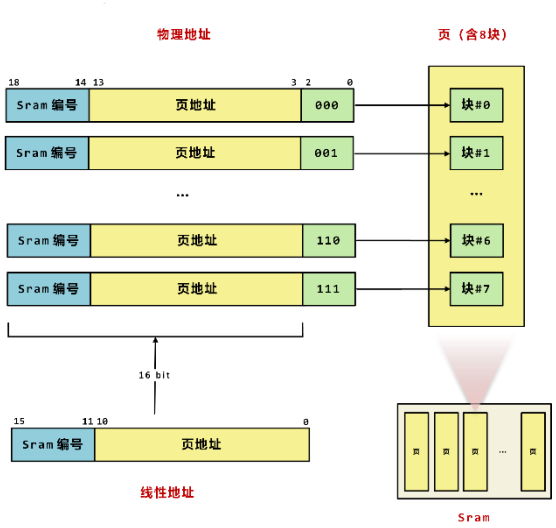


图 3.1.1 页表管理中地址的对应关系

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，处理时无需进行拼接操作，复杂度大大下降。

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更多资源。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 88KB 的存储资源。

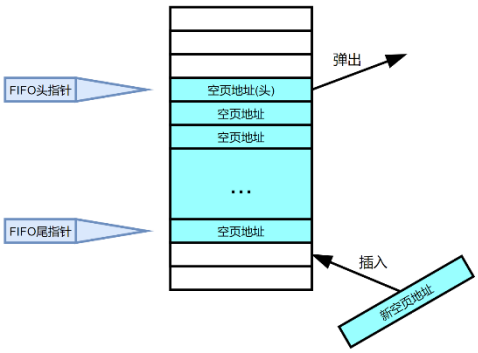


图 3.1.2 空闲链表运作机制

3.1.3 动态分配空间

Hydra 基于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。具体的表现为 Hydra 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。只要还剩下空闲的空间，任意端口就可以写入，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 匹配较优 SRAM

一个端口写入数据到 SRAM 时，有多个 SRAM 可用，若仅仅随机写入其中一个，会导致随着存入缓存的数据量增多，一个 SRAM 里会有各个端口（输出端口）的数据，不同端口同时输出数据需要访问 SRAM，此时会发生冲突，一些端口的请求需要等待别的端口请求完数据才被受理，导致读出的延迟极高（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，我们仍能尽量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 设计的框架规定一个数据包不得拆散在不同 SRAM 中，意味着 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，锁定分两种情况：

I）SRAM 正在被写；

伪双口 SRAM 无法同时进行多次写操作。

II）SRAM 已被其他端口搜索过程匹配。

不同端口可能同时认为一个 SRAM“最优”，导致写冲突和复杂的仲裁逻辑。

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数据较多；

2）包含的数据对应的端口总数较少。

尽可能把目的端口的数据聚集在一起，这样缓解了一块 SRAM 极多端口数据混杂的情况。

为了维护良好的时序性，端口匹配 SRAM 的过程需要 32 周期，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搜索过的最优的 SRAM，每次询问时，若硬性要求满足，则与当前最优的 SRAM 对比软性要求，若更优，则取而代之成为新的最优的 SRAM。搜索完成后，可以得到较优的 SRAM，接着再申请空闲页空间，开始 SRAM 的写入。

可见，从数据包开始进入 Hydra，到真正开始写入 SRAM，需要等待 32 周期的匹配过程，期间数据会流入端口前端模块的缓冲区中，在匹配完成后送入后端模块中完成写入。

为了防止出现多个端口同时询问一个 SRAM 是否匹配的情况（会导致冲突，需要额外的仲裁逻辑），Hydra 巧妙地设置了偏移量机制，保证每个端口每个周期搜索的 SRAM 刚好错开，互不干扰。

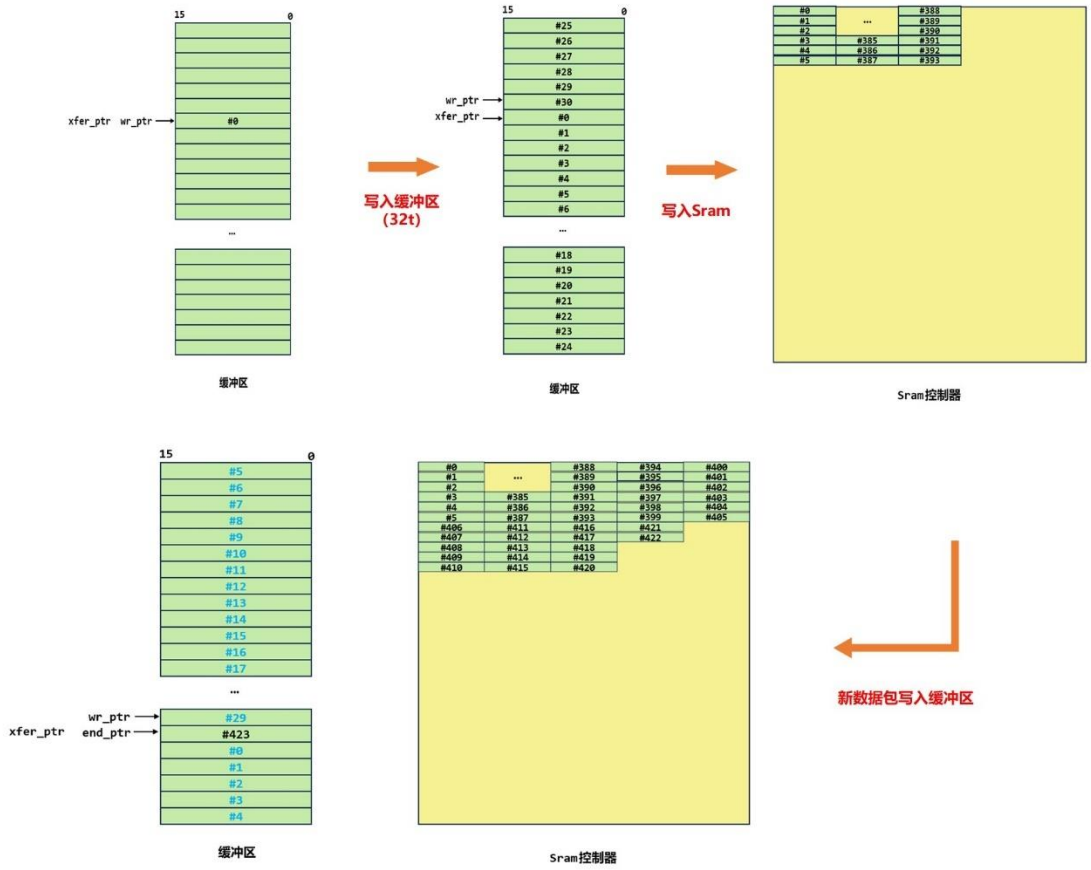


图 3.2.1 前端缓冲区机制

3. 2. 2 跳转表维护端口队列

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。

传统的方案便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，完全无法接受。若酌情缩短 FIFO 的深度，则无法做到完全的动态分配，各队列有数据量限制。

Hydra 采用的方案是为每个 SRAM 建立一个“跳转表”，将传统方案中的 FIFO 与链表数据结构的思想将结合，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放在一起存储。读取队列中数据时，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，每个优先级队列只需维护队头的页地址，即可以进入的顺序访问队中的所有元素；向队列中写入数据时，只需将原来末端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。

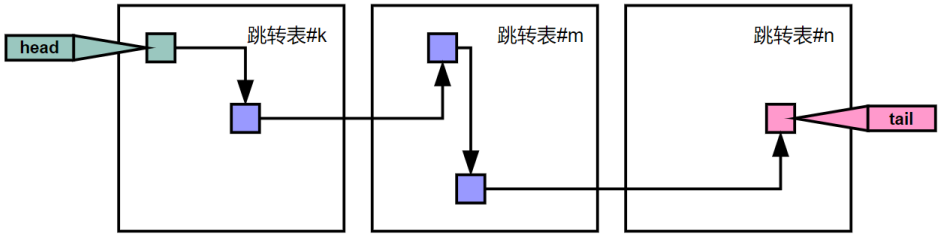


图 3.2.2(1) 跳转表维护优先级队列中的数据

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，这是因为每个 SRAM 同时只会与一个端口进行交互（见 SRAM 匹配机制）。所以以 SRAM 为单位划分的跳转表可以被整合到 BRAM 资源中。

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，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，为了避免同时来自多端口的“断链”拼接的问题，Hydra 采用了轮询机制，即每个端口轮流进行拼接操作，即可保证跳转表和队列末端有条不紊地更新。

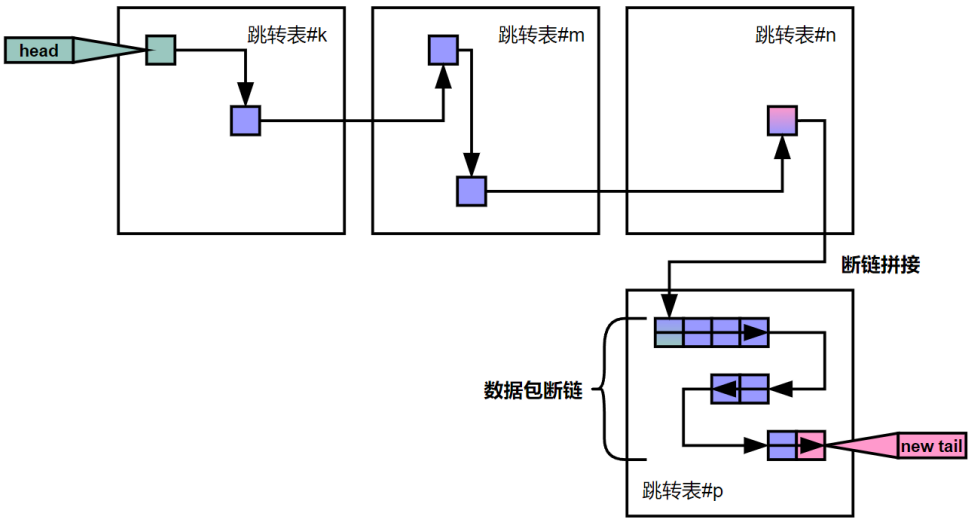


图 3.2.2(2) 并发写入情况下数据包入队机制

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，大大压缩地址管理所占用的资源，只需要 32*2048*16=128KB。

3.3 数据包调度

3.3.1 严格优先级调度

每个端口有 8 个优先级队列，Hydra 建立了一个 8 位宽的二进制数，指示每个队列是否有数据。根据严格优先级，读出数据时应访问有数据的队列中最优先的队列。要实现这个操作，只需将该 8 位数转化为独热码，即可得到最高位“1”的位置，即最优先的有数据的队列。

得到当前应当读出哪个队列的数据后，根据队列头页地址得到应当访问哪个 SRAM。考虑到可能会出现多个端口访问同一个 SRAM 的情况，此时需要引入轮询仲裁机制：得到端口应当输出哪个队列的数据后，向需要访问的 SRAM 端口号请求数据，每个 SRAM 都有一个 16 位宽的二进制数，指示了当前哪些端口正在请求数据，对其中为 1 的位进行轮询读取操作，每次轮询读出一页，宏观上表现即为访问同一 SRAM 的多个端口轮流输出 128 位数据，直至冲突结束。实现轮询操作使用了轮询位掩码，将其与请求指示数进行 AND 操作，即可得到当前应当处理哪个请求。

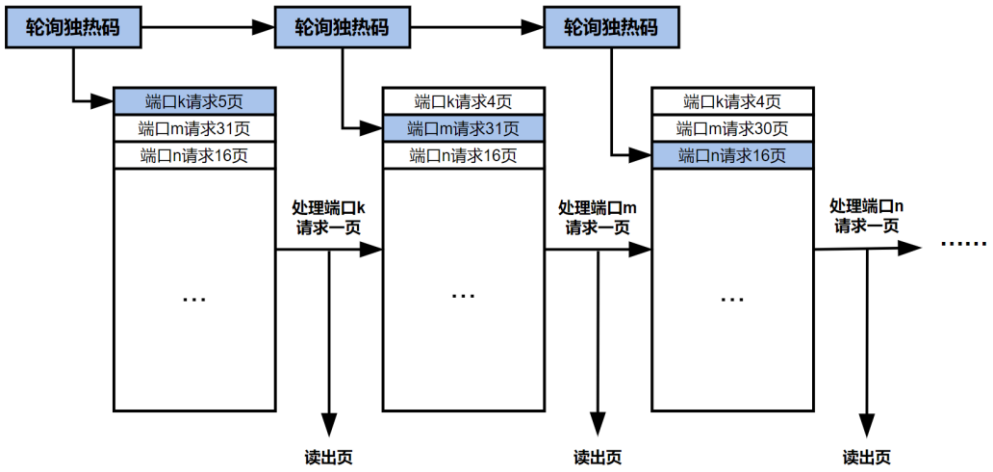


图 3.3.1 并发读取情况下轮询处理机制

值得注意的一点是，若多端口同时访问同一 SRAM 时，数据以页的方式轮流被读取，这样可以缓解传统仲裁中，后面的请求需要等待前面的请求完全处理完才能开始处理的情况，大大缩短了从 ready 信号拉高到第一个 rd_vld 拉高之间的延迟。

3.3.2 WRR 调度

Hydra 支持了使用最为广泛的一种 WRR 机制，即将输出数据包分为多个回合，第一回合八个优先级轮流输出，第二回合前七个优先级轮流输出，第三回合前六个周期优先级轮流输出……这样可以既考虑数据包的优先级，又缓解了低优先级数据包被一直堵塞无法输出的情况。

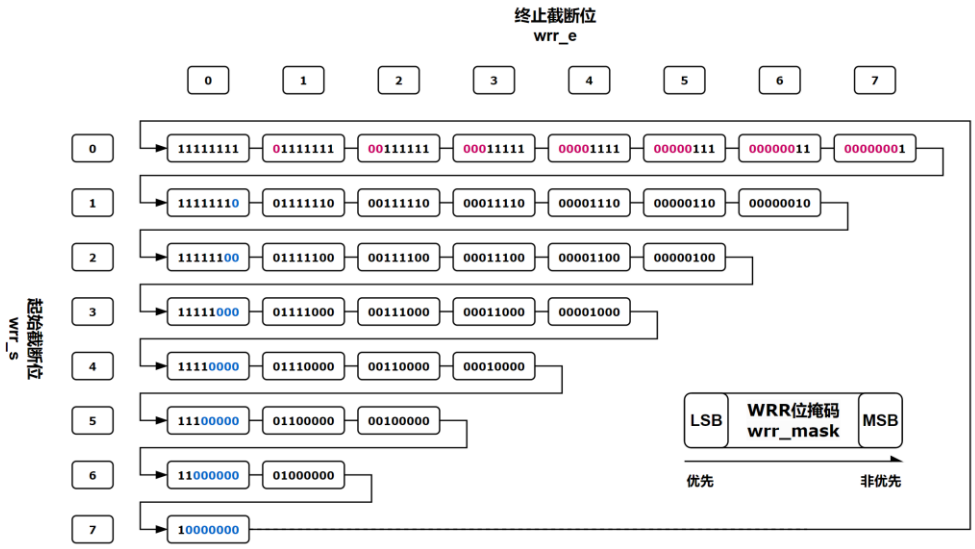


图 3.3.2(1) WRR 掩码状态机

实现 WRR 调度采用了位掩码的方案，每个端口都有一个 8 位的 WRR 位掩码，由掩码头、掩码尾维护，每次输出新数据包时，将 WRR 位掩码与队列指示码取 AND 操作取最高位，即可得到当前应输出哪个队列的数据包。

通过引入掩码机制，Hydra 成功将 WRR 调度从复杂的机制中解放，实现了占用资源极低（不到 16 位二进制数），时间复杂度极低（每次选择请求仅需 1 次 AND 运算所需的时间）。

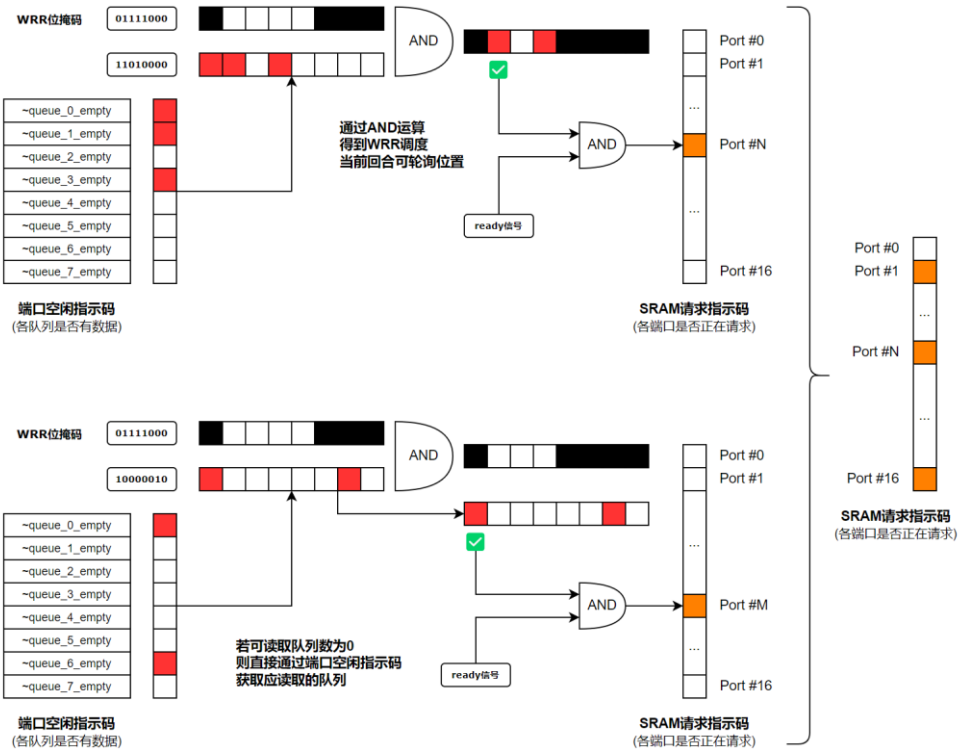


图 3.3.2(2) WRR 掩码结合优先级判定队列号

3.4 数据校验

3.4.1 汉明校验

Hydra 采用了(136,128)汉明校验，支持 SEC（单错误纠错）。

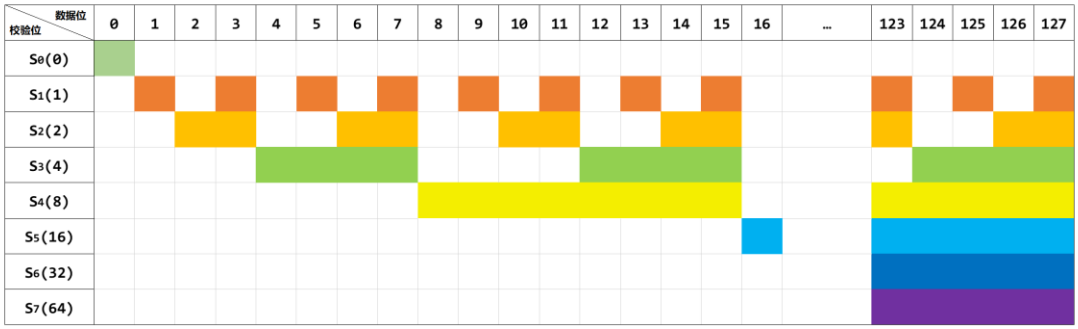


图 3.4.1(1) 汉明码计算原理（可视化 H 矩阵）

由于需要进行的异或运算较多，Hydra 将传统的串行计算改为并行计算，大大降低了编码所需的时间（只需 7 次异或门的时间），使得校验模块的时序性得到了保障。

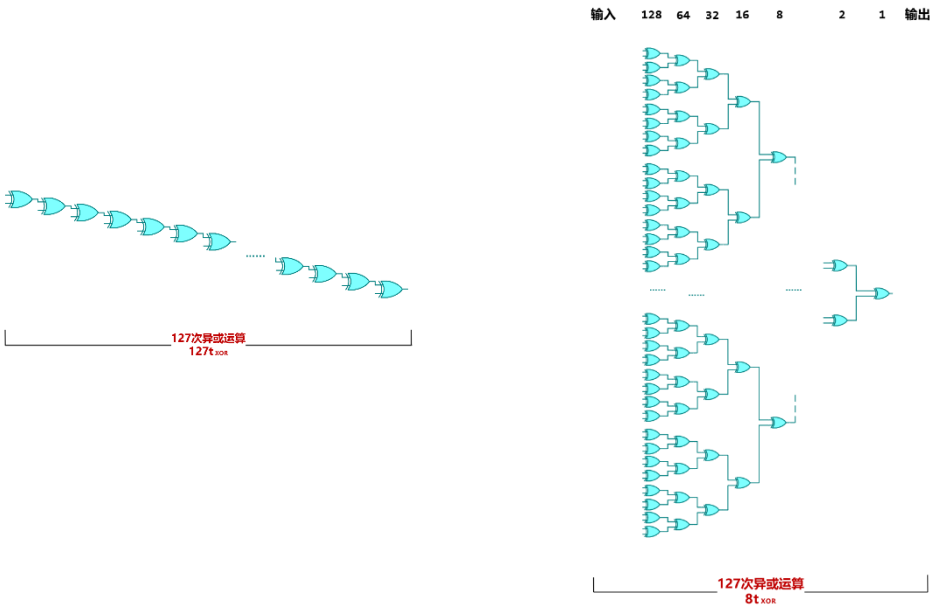


图 3.4.1(2) 串并行计算校验码耗时比较

3.4.2 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. 模块具体实现细节

4.1 各模块寄存器说明

4.1.1 controller(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。

表 4.1.1(1) controller 模块 IO 口介绍

类型	IO 口名称	含义	宽
input	clk	模块时钟信号	1
input	rst_n	模块复位信号	1
input	wr_sop	开始写入数据包信号	16*1
input	wr_eop	终止写入数据包信号	16*1
input	wr_vld	写入数据有效信号	16*1
input	wr_data	写入数据内容	16*16
output	full	端口无法为数据包分配更多空间	16*1
output	almost_full	端口即将无法为数据包分配更多空间	16*1
output	rd_sop	开始读出数据包信号	16*1
output	rd_eop	终止读出数据包信号	16*1
output	rd_vld	读出数据有效信号	16*1
output	rd_data	读出数据内容	16*16
input	ready	请求开始读出数据包信号	16*1
input	wrr_en	WRR 调度使能信号	16*1

表 4.1.1(2) controller 模块重要寄存器介绍

寄存器名称	含义	宽
searching_sram_index	端口正在匹配的 SRAM	16*5
searching_distribution	端口当前匹配的最优 SRAM	16*5
max_amount	端口当前最优 SRAM 的同端口数据量	16*11
search_cnt	端口匹配过程计数器	16*6
searching	端口是否正在匹配	16*1
queue_head_sram	端口各队列头页所在的 SRAM	16*8*5
queue_head_page	端口各队列头页的线性地址	16*8*11
queue_tail_sram	端口各队列尾页所在的 SRAM	16*8*5
queue_tail_page	端口各队列尾页的线性地址	16*8*11
queue_not_empty	端口各队列是否有数据包	16*8*1
handshake	读出后端向前端发出的握手信号	16*1

4.1.2 dual_port_sram

功能：伪双口 SRAM，提供独立工作的读写口各一个。

表 4.1.2(1) dual_port_sram 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	rst_n	模块复位信号（片选信号）	1
input	wr_en	写入数据使能信号	1
input	wr_addr	写入数据物理地址	14
input	din	写入数据内容	16
input	rd_en	读出数据使能信号	1
input	rd_addr	读出数据物理地址	14
output	dout	读出数据内容	16

表 4.1.2(2) dual_port_sram 模块重要寄存器介绍

寄存器名称	含义	宽
d_latches	SRAM 存储的数据	16384*16

4.1.3 ecc_encoder

功能：生成一页（128 位）数据的 8 位汉明校验码。

表 4.1.3 ecc_encoder 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	enable	编码使能信号	1
input	data_0	待校验数据#0	16
input	data_1	待校验数据#1	16
input	data_2	待校验数据#2	16
input	data_3	待校验数据#3	16
input	data_4	待校验数据#4	16
input	data_5	待校验数据#5	16
input	data_6	待校验数据#6	16
input	data_7	待校验数据#7	16
output	code	生成的校验码	8

4.1.4 ecc_decoder

功能：根据 8 位校验码对一页（128 位）待纠错数据纠错并输出。

表 4.1.4(1) ecc_decoder 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	enable	纠错使能信号	16
input	data_0	待纠错数据#0	16
input	data_1	待纠错数据#1	16
input	data_2	待纠错数据#2	16
input	data_3	待纠错数据#3	16
input	data_4	待纠错数据#4	16
input	data_5	待纠错数据#5	16
input	data_6	待纠错数据#6	16
input	data_7	待纠错数据#7	16
input	code	校验码	1
output	cr_data_0	已纠错数据#0	16
output	cr_data_1	已纠错数据#1	16
output	cr_data_2	已纠错数据#2	16
output	cr_data_3	已纠错数据#3	16
output	cr_data_4	已纠错数据#4	16
output	cr_data_5	已纠错数据#5	16
output	cr_data_6	已纠错数据#6	16
output	cr_data_7	已纠错数据#7	16

表 4.1.4(2) ecc_decoder 模块重要寄存器介绍

寄存器名称	含义	宽
cr_data	已纠错数据	128
cur_code	待纠错数据对应的校验码	8
wrong_pos	错误位置	8

4.1.5 fifo_null_pages

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。

表 4.1.5(1) fifo_null_pages 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	rst_n	模块复位信号	1
input	pop_head	弹出队头使能信号	1
output	head_addr	当前队头页地址	11
input	push_tail	插入队尾使能信号	1
input	tail_addr	插入的页地址	11

表 4.1.5(2) fifo_null_pages 模块重要寄存器介绍

寄存器名称	含义	宽
fifo	队列数据	2048*11
head_addr	队列头指针	11
tail_addr	队列尾指针	11

4.1.6 port_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。

表 4.1.6(1) port_frontend 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	wr_sop	开始写入数据包信号	1
input	wr_eop	终止写入数据包信号	1
input	wr_vld	写入数据有效信号	1
input	wr_data	写入数据内容	16
input	search_get	是否成功匹配到 SRAM	1
output	dest_port	当前写入数据包的目的端口	4
output	prior	当前写入数据包的优先级	3
output	length	当前写入数据包的长度	9
output	new_packet_into_buf	当前有新数据包写入（触发搜索）	1
output	data	传输至后端的数据内容	16
output	data_vld	传输至后端的数据有效信号	1

表 4.1.6(2) port_frontend 模块重要寄存器介绍

寄存器名称	含义	宽
buffer	被缓冲的数据	64*16
xfer_en	传输数据至后端使能	1
xfer_ptr	正在传输的数据位置指针	6
wr_ptr	正在写入的数据位置指针	6
end_ptr	未传输完的上一数据包结束位置指针	6

4.1.7 sram_state

功能：管理 ECC 存储、跳转表和 SRAM 端口数据量。

表 4.1.7(1) sram_state 模块 IO 口介绍

类型	名称	含义	线宽
input	clk	模块时钟信号	1
input	rst_n	模块复位信号	1
input	ecc_wr_en	ECC 存储写入使能信号	1
input	ecc_wr_addr	ECC 存储写入页地址	11
input	ecc_din	ECC 存储写入校验码	8
input	ecc_rd_en	ECC 存储读出使能信号	1
input	ecc_rd_addr	ECC 存储读出页地址	11
output	ecc_dout	ECC 存储读出校验码	8
input	jt_wr_en	跳转表写入使能信号	1
input	jt_wr_addr	跳转表写入页地址	11
input	jt_din	跳转表写入结点内容	16
input	jt_rd_en	跳转表读出使能信号	1
input	jt_rd_addr	跳转表读出页地址	11
output	jt_dout	跳转表读出结点内容	16
input	wr_op	写单页信号	1
input	wr_or	写数据包信号	1
input	wr_port	写入数据包的输出端口	4
input	delta_free_space	写入数据包时剩余空间改变量	12
input	delta_page_amount	写入数据包时端口统计该变量	12
input	rd_op	读单页信号	1
input	rd_port	读出数据包的输出端口	4
input	rd_addr	读出的页地址	11
input	request_port	请求查询数据量的输出端口	4
output	page_amount	查询到的打开看看数据量	11
output	null_ptr	空闲页指针	11
output	free_space	剩余可写页	11

表 4.1.7(2) sram_state 模块重要寄存器介绍

寄存器名称	含义	宽
ecc_storage	ECC 存储	2048*8
jump_table	跳转表	2048*16
port_amount	各端口的数据量	16*11

4.2 重要逻辑说明

4.2.1 写前端流水线

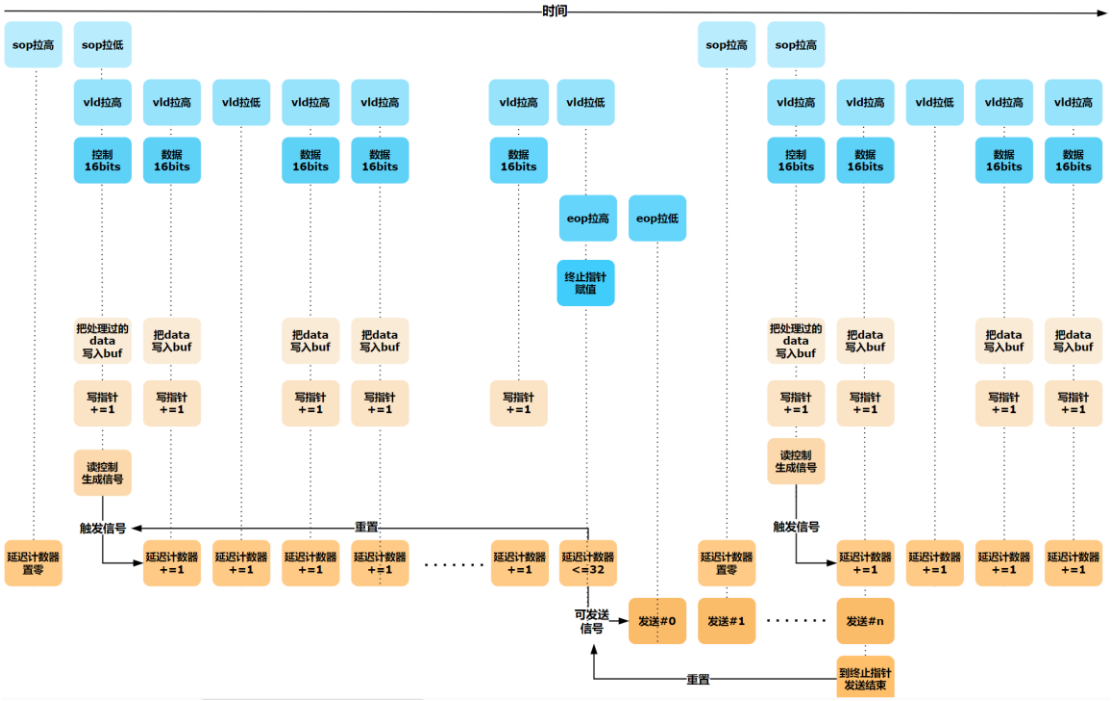


图 4.2.1 写前端流水线示意图 写控制 IO 口 前端逻辑

4.2.2 写后端流水线

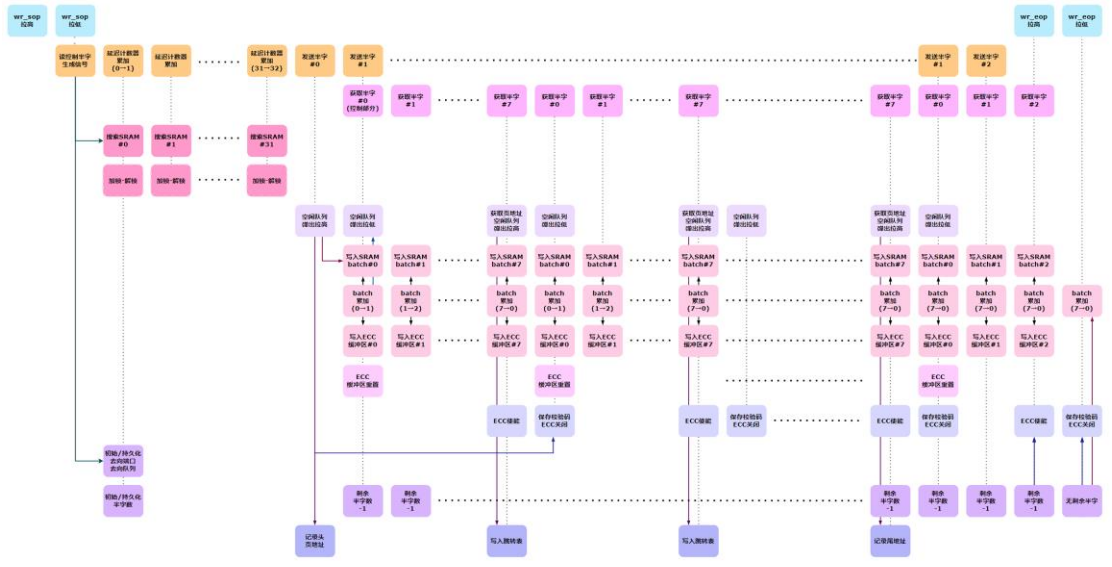


图 4.2.2 写后端流水线示意图 写控制 IO 口 前端信号 后端逻辑

4.2.3 读前后端流水线

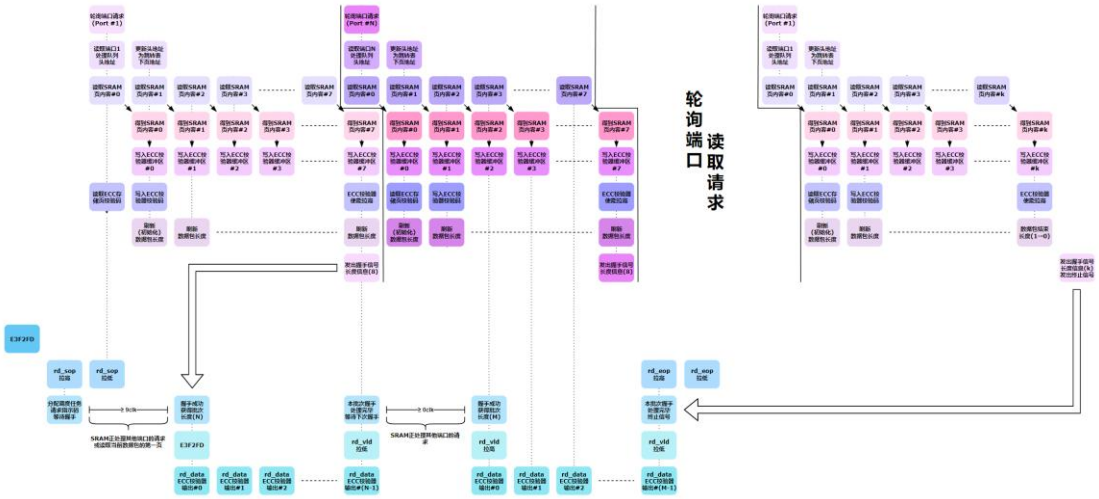


图 4.2.3 读前后端流水线示意图 读控制 IO 口/前端逻辑 后端逻辑

5. 接口与配置

5.1. 写控制 IO 口

5.1.1 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。

5.1.2 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。

5.2 写反馈 IO 口

5.2.1 IO 口介绍

full：当所有 SRAM 被占用/满时拉高，或端口无法匹配到可存储新包的 SRAM 时为高。

almost_full：当端口没有占用 SRAM，且可用 SRAM 数量小于端口数量时拉高。

5.2.2 使用方法

当某个端口的 full 拉高时，端口不应写入任何数据，正在写入的数据包应当立即停止写入，在合适的时机从头重新写入（已经写入的在缓冲里的数据将被“冲刷”）。

当某个端口的 almost_full 拉高时，则做好准备在写入过程中拉高 full 的“冲刷”操作。

5.3 读控制 IO 口

5.3.1 IO 口介绍

ready：请求读出一个数据包的信号。

5.3.2 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。

5.4 读反馈 IO 口

5.4.1 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。

5.4.2 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。

5.5 WRR 使能配置

wrr_en：每个端口各 1 个，表示是否开启该端口的 WRR 调度模式。

6. 验证方法

6.1 RTL 级仿真

6.1.1 RTL 级电路图

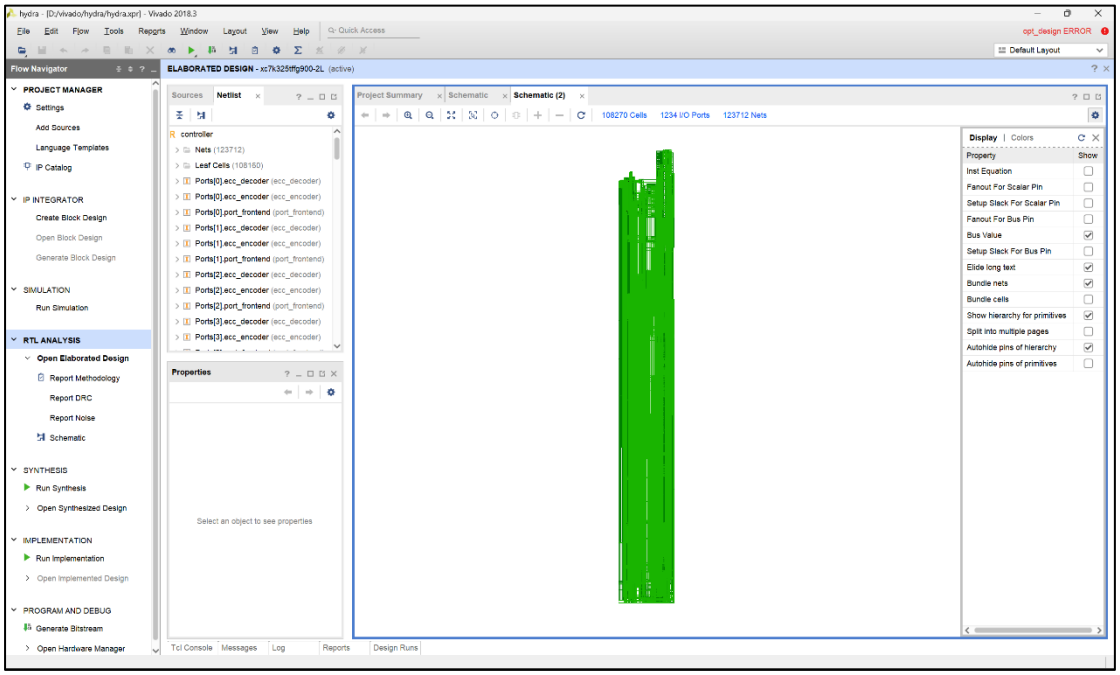


图 6.1.1 RTL 级电路

6.1.2 单模块仿真

见附件中的.tb 测试文件和.wlf/.do 仿真波形文件。

6.1.3 总模块仿真

见附件中的.tb 测试文件和.wlf/.do 仿真波形文件。

6.2 综合验证

6.2.1 综合环境

操作系统：Microsoft Windows 11

Vivado 版本：2018.3

板卡系列：Kintex-7

板卡型号：xc7k325tffg900-2L

片上资源：**IO Pin**：900　　**Block RAMs**：445 * 36Kbits

LUT：203800　　**Ultra RAMs**：0

FlipFlops：407600　　**DSPs**：840

6.2.2 综合结果

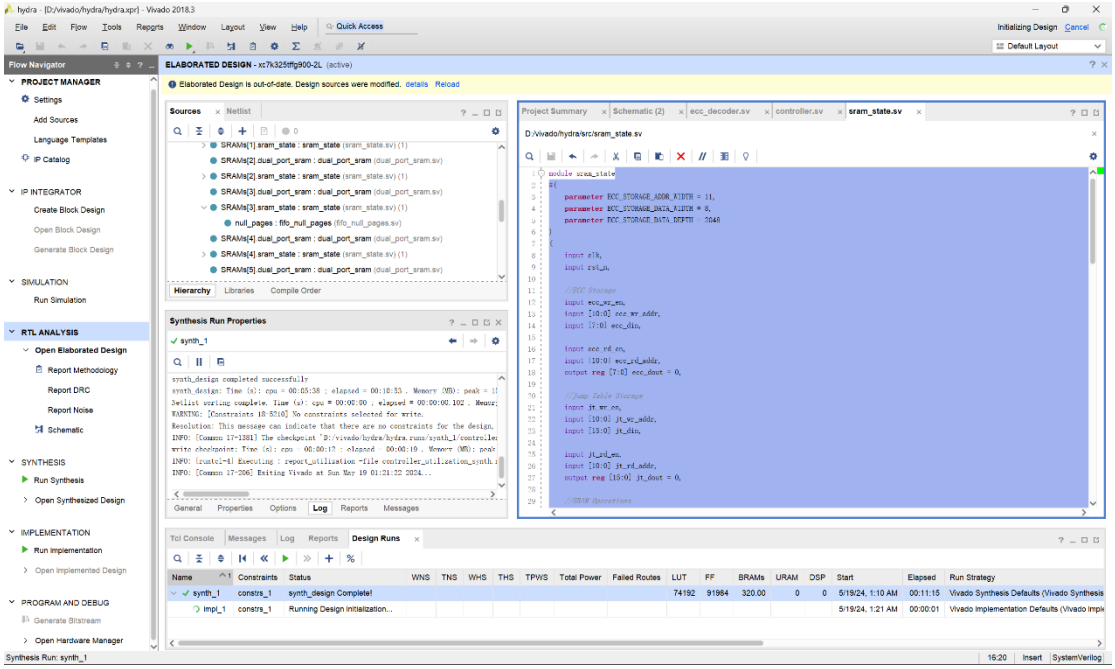


图 6.2.2(1) 综合成功结果

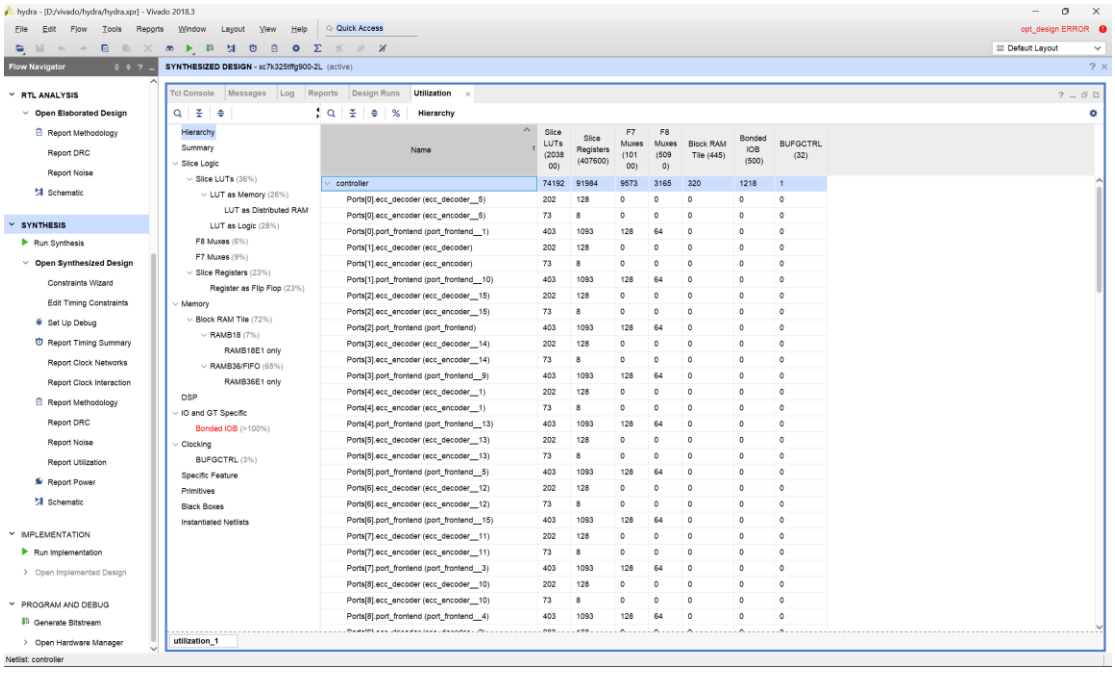


图 6.2.2(2) 模块资源占用

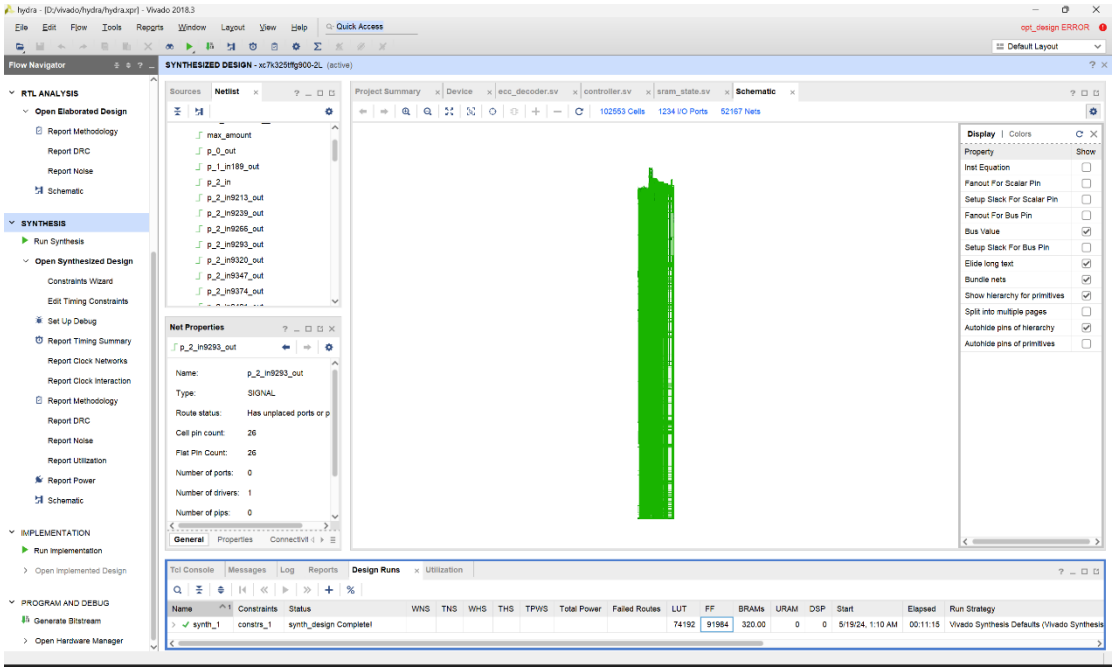


图 6.2.2(3) 综合电路（全貌）

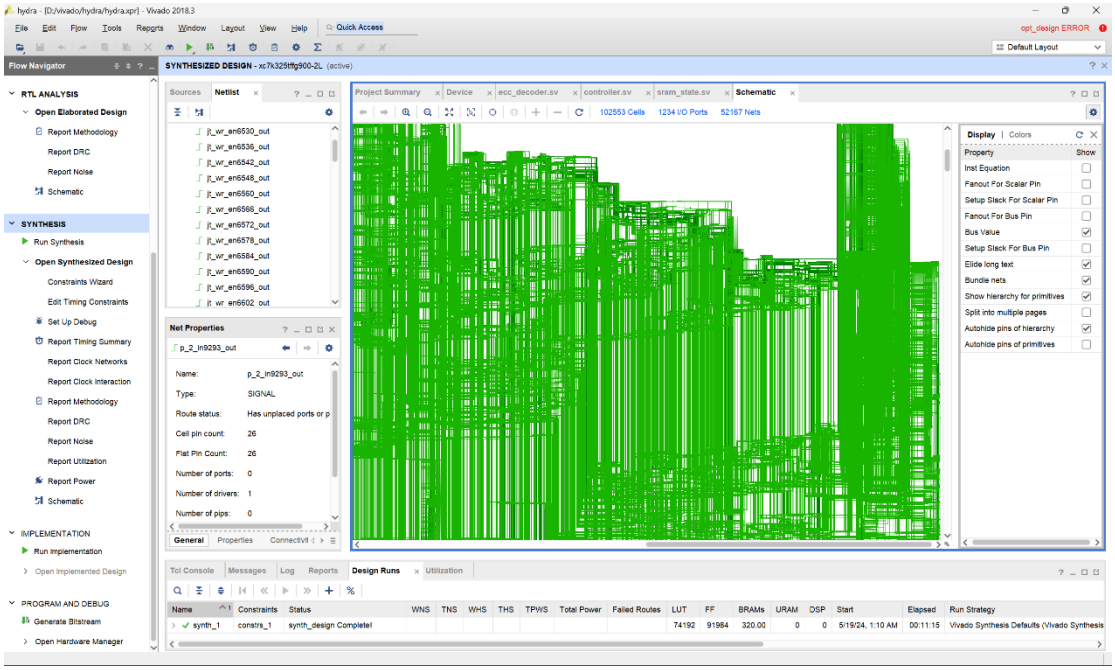
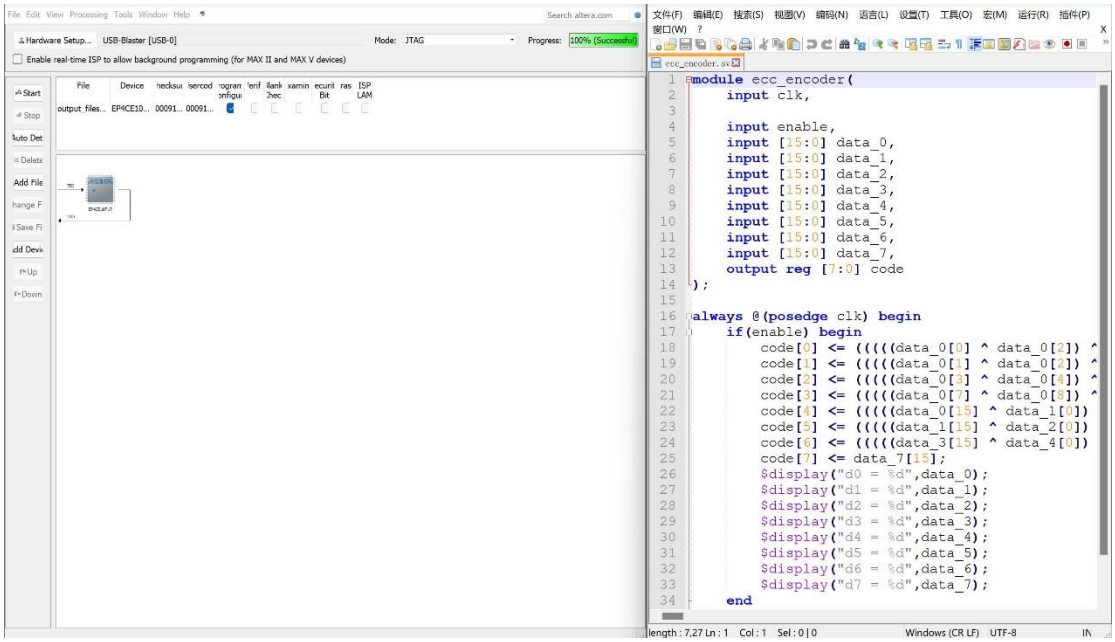


图 6.2.2(4) 综合电路（放大）

6.3 FPGA 验证

6.3.1 Ecc 校验编解码模块



```
module ecc_encoder(
    input clk,

    input enable,
    input [15:0] data_0,
    input [15:0] data_1,
    input [15:0] data_2,
    input [15:0] data_3,
    input [15:0] data_4,
    input [15:0] data_5,
    input [15:0] data_6,
    input [15:0] data_7,
    output reg [7:0] code
);

always @(posedge clk) begin
    if(enable) begin
        code[0] <= (((((data_0[0] ^ data_0[2]) ^
        code[1] <= (((((data_0[1] ^ data_0[2]) ^
        code[2] <= (((((data_0[3] ^ data_0[4]) ^
        code[3] <= (((((data_0[7] ^ data_0[8]) ^
        code[4] <= (((((data_0[15] ^ data_1[0])
        code[5] <= (((((data_1[15] ^ data_2[0])
        code[6] <= (((((data_3[15] ^ data_4[0])
        code[7] <= data_7[15];
        $display("d0 = %d",data_0);
        $display("d1 = %d",data_1);
        $display("d2 = %d",data_2);
        $display("d3 = %d",data_3);
        $display("d4 = %d",data_4);
        $display("d5 = %d",data_5);
        $display("d6 = %d",data_6);
        $display("d7 = %d",data_7);
    end
```

图 6.3.1(1) 程序烧录截图

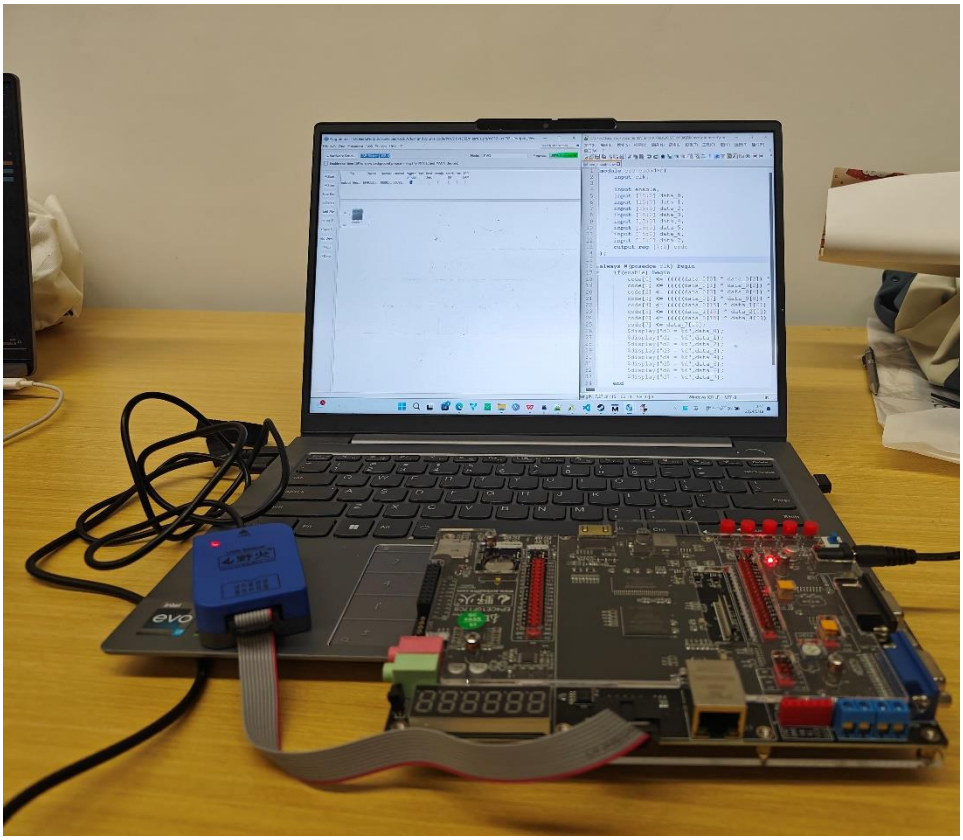


图 6.3.1(2) FPGA 板运行图

7. 设计优缺点

7.1 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理论峰值；

6）在有可用 SRAM 时保证端口写入带宽总为理论峰值；

7）极大压缩模块的存储资源占用（相较于传统方案而言）；

8）代码风格良好，严格遵循命名规范，注释清晰。

7.2 缺点

1）匹配过程使得数据包写入延迟 32~48 周期；

2）页表管理导致存储数据包时，SRAM 部分空间（<5%）被暂时性浪费；

3）未匹配到 SRAM 时，写入强行终止，已被写入前端缓冲区的一部分数据会被冲刷。

8. 后续开发计划

8.1 pause 接口

见缺点 3)，当一个数据包开始发送时，Hydra 会启动搜索，尝试匹配合适的 SRAM 对其进行存储，若未匹配到，则会拉高端口的 full 接口，强行终止此次发送，已发送进缓冲区的数据将会被冲刷。

Hydra 未来会引入 pause 接口，一个新的写反馈接口，当出现上述情况时，pause 将会被拉高，暂停本次传输，并一直启动搜索，直到原来被占用/满的 SRAM 被释放，匹配到合适的 SRAM，pause 被拉低，本次传输继续。这样不会出现在传输过程中拉高 full 接口，避免模块内部数据冲刷、模块外部设备重新发送数据包的复杂逻辑。

8.2 读写延迟优化

Hydra 未来会进一步优化流水线，并简化冗余的时序逻辑，预计可降低完全动态分配下读写延迟。

8.3 半动态模式

Hydra 未来会引入新模式，旨在提供一种介于完全动态分配与静态分配内存之间的一种策略，即半动态模式。具体的设计为每个端口绑定一块 SRAM，并共享剩余的 16 块 SRAM。

半动态模式在保证带宽不变的前提下，牺牲单端口数据容量，极大减少读写延迟。在部分场合比完全动态模式表现更佳。

8.4 模块化框架

Hydra 未来将会拆解主控制模块，进一步提升代码的层次性与可读性。