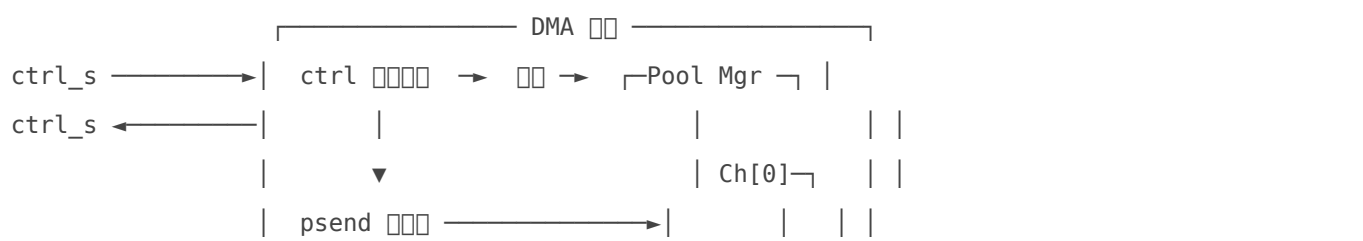


DMA

DMA ??????

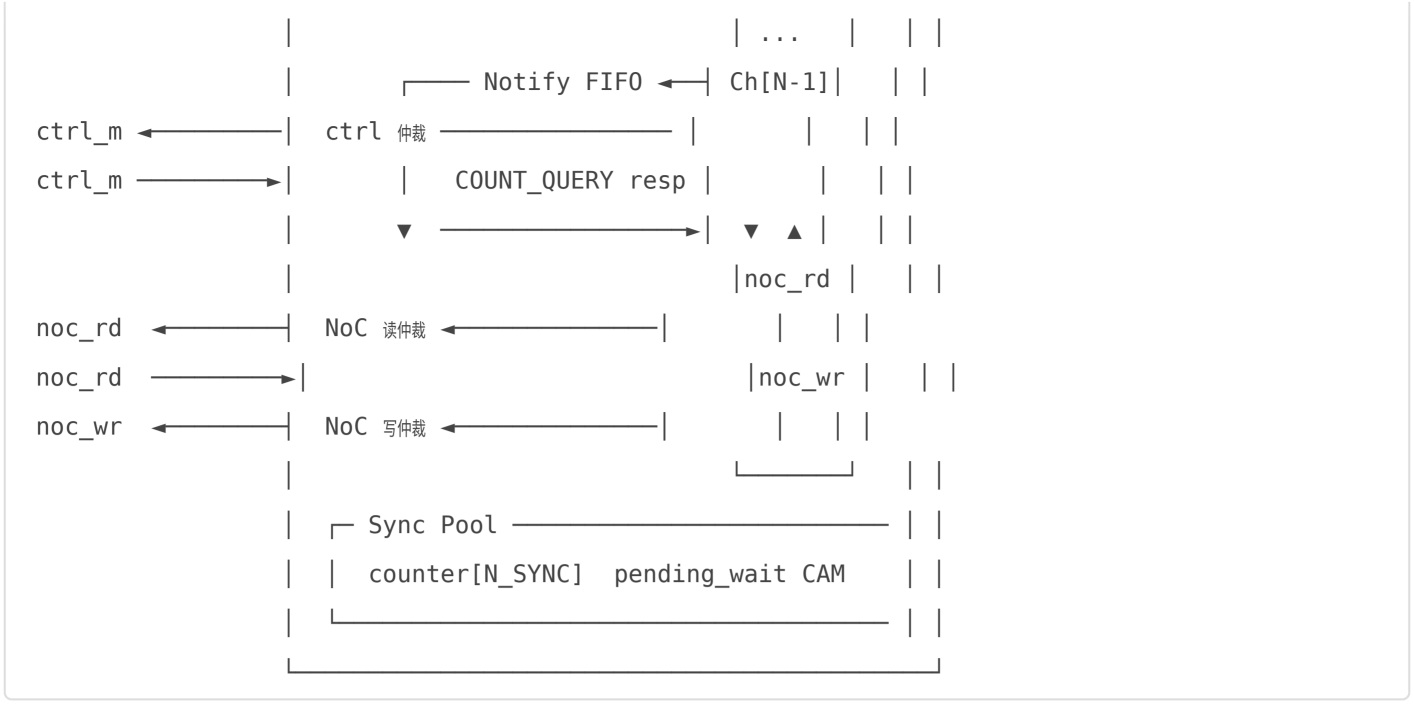
把 [[Pipe]] 第 6 节 §176 的 DMA 抽象一直展开到 RTL 起手前的最后一步。每条 channel = 一条 engine pipe；每个 sync counter = 一条 sync pipe。除非显式指出，假设单时钟域、同步复位、AXI-stream 风格的 valid/ready 握手。

1. ????

参数	含义	默认	限制
`N_CH`	channel 实例数	16	≤ 64
`N_SYNC`	sync counter 数	32	≤ 256
`Q_MAX`	单 channel 队列硬件上限	16	2 的幂
`K_IN / K_OUT / K_SIG`	cmd_desc 中三个 pipe 列表的最大长度	4 / 4 / 4	编码受硬件限制
`COUNT_W`	所有 count 寄存器位宽	32	wrap 不报错
`HANDLE_W`	pipe handle 位宽	16 = unit_addr(8) + pipe_id(8)	
`ADDR_W`	系统地址位宽	48	
`NOC_W`	NoC 数据 beat 位宽	512	
`STRIDE_DIM`	地址生成最大维度	3	内 + 间 + batch
`STRIDE_W / SHAPE_W`	地址 step / 维度大小位宽	32 / 24	
`DESC_W`	cmd_desc SRAM 字宽	1024	取字段总和向上对齐
`N_PEND_WAIT`	全单元未决 `pwait` 数	16	
`TILE_BUF_BYTES`	每 channel transpose buffer	4 KiB	决定最大可 transpose tile

2. ????

```
                  ┌──────────────── DMA 单元 ────────────────┐
ctrl_s ─────────►│ ctrl 接收解码 ─► 路由 ─► ┌─Pool Mgr ─┐ │
ctrl_s ◄─────────│       │                    │            │ │
                 │       ▼                    │ Ch[0]─┐    │ │
                 │ psend 写队列 ─────────────►│       │  │ │
```



各子模块:

1. **ctrl** 接收解码 + 路由：单一从端口；按 opcode + handle 路由到 Pool Mgr / Channel / Sync Pool
2. **Pool Mgr**：channel 的 alloc / free + 元数据表
3. **Channel × N_CH**：每条独立，含队列、FSM、地址生成、transform、通知 FIFO、完成计数器
4. **Sync Pool**：N_SYNC 个独立 counter + pending wait CAM + 独立的 alloc bitmap
5. **NoC** 仲裁：所有 channel 共享一个 read master + 一个 write master，round-robin
6. **ctrl** 主端仲裁：所有 channel 的 Notify FIFO + Sync Pool 的回 ack 走同一对外 master，round-robin

3. ????

信号组	方向	描述
`clk, rst_n`	in	单时钟，异步 assert / 同步 deassert 复位
`ctrl_s_*`	slave	CPU → DMA 的控制总线从端
`ctrl_m_*`	master	DMA → 其他单元的控制总线主端
`noc_rd_*`	master	NoC 读 master
`noc_wr_*`	master	NoC 写 master
`dbg_*`	out	可观测点（见 §12）

3.1 `ctrl_s_*` ????

信号	方向	位宽	说明
`ctrl_s_req_valid`	in	1	

信号	方向	位宽	说明
`ctrl_s_req_ready`	out	1	
`ctrl_s_req_op`	in	4	opcode（§5.2）
`ctrl_s_req_handle`	in	HANDLE_W	目标 pipe handle；palloc 时不用
`ctrl_s_req_beat_first`	in	1	多 beat cmd_desc 写的首尾标志
`ctrl_s_req_beat_last`	in	1	
`ctrl_s_req_payload`	in	128	每 beat 128 bit；DESC_W = 8 beat
`ctrl_s_resp_valid`	out	1	
`ctrl_s_resp_ready`	in	1	
`ctrl_s_resp_data`	out	max(HANDLE_W, COUNT_W) = 32	palloc 返 handle / pread 返 count
`ctrl_s_resp_tag`	out	4	关联请求顺序（in-order 也保留以便 pipeline）

3.2 `ctrl_m_*`????

信号	方向	位宽	说明
`ctrl_m_req_valid`	out	1	
`ctrl_m_req_ready`	in	1	
`ctrl_m_req_op`	out	4	COUNT_DELTA / COUNT_QUERY
`ctrl_m_req_handle`	out	HANDLE_W	目标单元的 pipe
`ctrl_m_req_payload`	out	COUNT_W	delta（有符号）或 threshold
`ctrl_m_req_tag`	out	log2(N_CH+1)	用于将 query 回应路由回发起 channel
`ctrl_m_resp_valid`	in	1	仅 COUNT_QUERY 响应
`ctrl_m_resp_ready`	out	1	
`ctrl_m_resp_tag`	in	log2(N_CH+1)	

3.3 `noc_rd_*` / `noc_wr_*`

标准 AXI-stream / AXI4 风格读写 master。要点：

- `id` 字段宽度 = log2(N_CH)，回 resp 时由 NoC 仲裁还原路由
- 每个读请求带 `addr / len / id / qos`，每个 beat 带 `data / id / last`
- 写请求带 `addr / len / id / qos`；写数据 channel 带 `data / strb / last`

- 不要求写 ack（fire-and-forget），但 NoC 必须保证同 id 同地址的写最终落地后 count_delta 才能被发送

4. cmd_desc ????

总宽 824 bit，对齐到 DESC_W = 1024 bit（高位保留）。

偏移	位宽	字段	说明
0	4	`op`	0=COPY，1=BCAST，2=GATHER，3=SCATTER，余保留
4	4	`xform_flags`	bit0 transpose，bit1 pad，bit2 slice，bit3 deslice
8	48	`src_base`	字节地址（落在某条 pipe 内）
56	96	`src_stride[3]`	每维 32 bit signed，单位字节
152	48	`dst_base`	
200	96	`dst_stride[3]`	
296	72	`shape[3]`	每维 24 bit unsigned
368	64	`xform_params`	pad amounts / slice offsets / transpose 块大小
432	4	`n_in`	实际使用的 in 表项数（≤ K_IN）
436	4	`n_out`	
440	4	`n_sig`	
444	4	reserved	
448	96	`in_list[4]`	每项 {handle:16, delta:8} = 24 bit
544	96	`out_list[4]`	
640	96	`sig_list[4]`	
736	88	reserved	
824	200	padding	高位置 0

`n_in / n_out / n_sig` 为 0 表示该侧无端点（保留以兼容未来扩展；DMA 在本设计中数据端必为 pipe，n_in / n_out 至少为 1）。

5. ??????

5.1 opcode ?

op	名	方向	**payload** 用途	多 **beat**

0x0	PALLOC	s	{type, queue_depth}	否
0x1	PFREE	s	—	否
0x2	PSEND	s	cmd_desc	是（8 beat)
0x3	PREAD	s	—	否
0x4	PWAIT	s	threshold	否 (响应可能挂起)
0x5	PSET_COUNT	s	value	否
0x6	PSIGNAL	s	delta	否
0x8	COUNT_DELTA	s/m	delta	否
0x9	COUNT_QUERY	s/m	threshold	否

5.2 ? beat PSEND ???

- `req_op = PSEND` 在每beat上保持; CPU必须在8个连续beat内完成cmd_desc推送, DMA在`beat_last=1`那一拍才提交到目标channel队列
- 期间若`req_ready`拉低, CPU必须暂停; DMA内部用8 × 128 bit移位寄存器累积, 不要求双缓冲（带宽够时一拍接一拍）
- 不允许多个PSEND交错; 若CPU必须背靠背发不同channel的PSEND, 需排队串行

5.3 ??

- 从端: CPU同向发出的请求严格按到达顺序处理（in-order总线）；响应也按发出顺序
- 主端: 本单元发出的`COUNT_DELTA`之间保序（FIFO），不同发起channel之间不保序

5.4 ???????

- 错的handle、count溢出、队列满、超借都不产生硬件error; 按特性6由CPU软件保证

6. Channel Pool ???

6.1 ??

存储	形式	容量	端口
`alloc_bitmap`	FF	N_CH bit	1R/1W, 单cycle
`meta[N_CH]`	1RW SRAM或register file	N_CH × (log2(Q_MAX)+1+1) bit	1RW
`count[N_CH]`	register file	N_CH × COUNT_W	1R/1W（与§7.2共享, 见下）

注: `count`寄存器物理上在每条channel内（§7.2）, Pool Mgr只在PALLOC时复位它。

6.2 alloc ????

```
PALLOC req → priority_encoder(alloc_bitmap) → free_id
           ├── alloc_bitmap[free_id] ← 1
           ├── meta[free_id] ← {queue_depth, valid=1}
           ├── count[free_id] ← 0
           └── resp ← {unit_addr, free_id}    # HANDLE_W bits
```

- 优先编码器: N_CH = 16 时 4 级 OR 树, 单 cycle
- 整体 PALLOC 延迟: 2 cycle (解码 1 + 写回 1) + 响应回总线 1 cycle

6.3 free ????

```
PFREE req(handle.pipe_id = id) → alloc_bitmap[id] ← 0
                               → meta[id].valid ← 0
```

不复位 count (下次 PALLOC 时再清); 不检查队列空 (特性 6)。

7. Channel ?????×N_CH?

每条 channel 是完全独立的硬件块。下列描述一条 channel; N_CH 条复用对外仲裁器。

7.1 ????

项	实现
存储	1R/1W SRAM, 深度 Q_MAX, 宽度 DESC_W
`head` 指针	log2(Q_MAX) + 1 bit (高位作 wrap 标志)
`tail` 指针	log2(Q_MAX) + 1 bit
空	head == tail
满	head[high] != tail[high] && head[low] == tail[low]
写端	PSEND 多 beat 完成后单 cycle 写 SRAM (地址 = head.low), `head++`
读端	执行 FSM 在 FETCH 态读 SRAM (地址 = tail.low); 命令完成后 `tail++`

不做满检查 —— 软件保证。

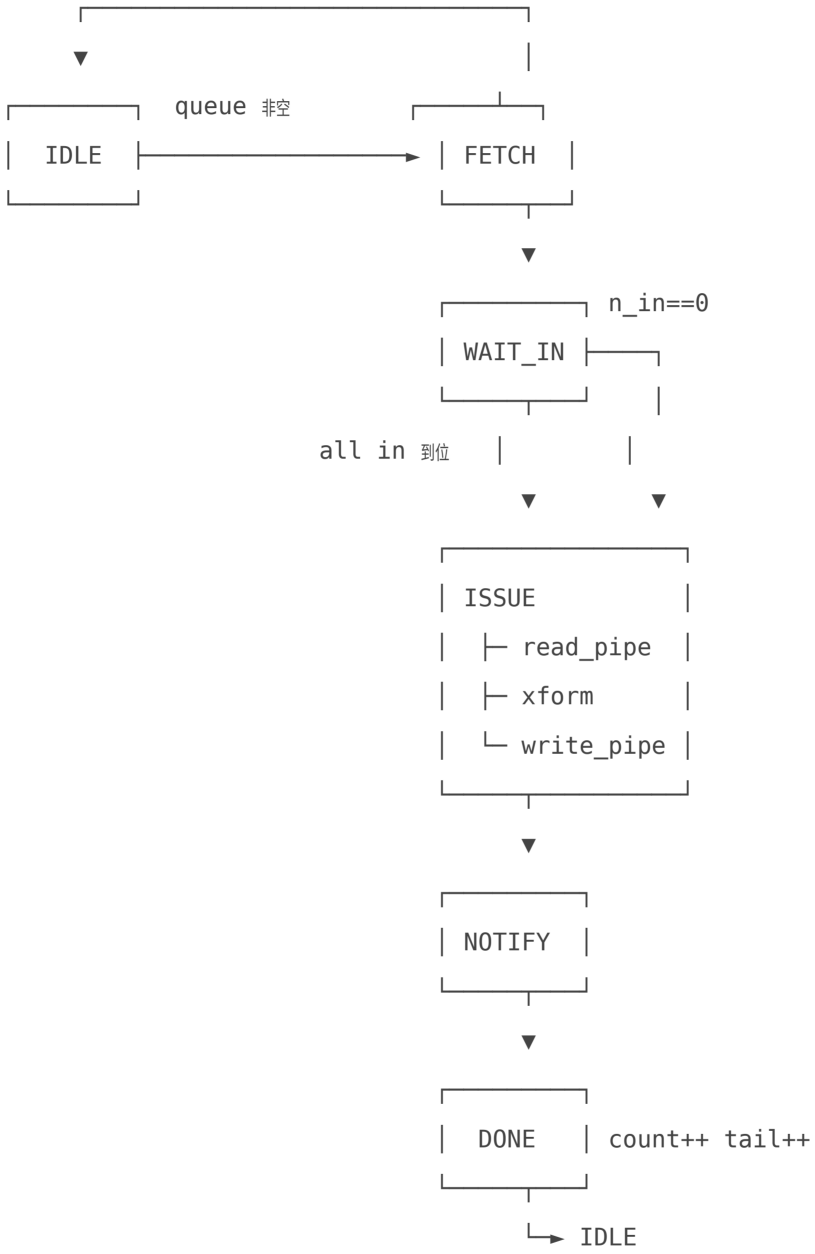
7.2 ?????

```
reg [COUNT_W-1:0] count;
always @(posedge clk) begin
  if (!rst_n)            count <= 0;
  else if (alloc)        count <= 0;            // PALLOC 时复位
  else if (cmd_complete) count <= count + 1;    // 命令出队时 +1
```

```
    else if (set_count)    count <= set_val;      // PSET_COUNT
  end
```

`pread` 直接组合输出；`pwait` 见 §10.2 的统一 pending wait 机制（channel 自己的 count 也走同一个 CAM）。

7.3 ?? FSM



状态	时长	行为
IDLE	1 cycle	监视 queue empty 标志
FETCH	1 cycle	从 queue SRAM 读 cmd_desc 到工作寄存器组

状态	时长	行为
WAIT_IN	不定	对 `in_list[0..n_in-1]` 顺序发 COUNT_QUERY；收到 ack 后步进；全部到位后转 ISSUE
ISSUE	多 cycle，pipeline	read / xform / write 三级流水（见 §7.5）
NOTIFY	n_in + n_out + n_sig cycle	顺序入队 Notify FIFO（§7.6）
DONE	1 cycle	`count++`，`tail++`

工作寄存器组：1 份 cmd_desc 拷贝 + 当前进度（in 检查索引、地址生成器状态、notify 索引）。

7.4 src / dst ?????

每个 channel 持 2 个独立实例（src + dst）。

累加器实现，避免乘法器：

```
state:
  base                  ADDR_W
  stride[STRIDE_DIM]    STRIDE_W each
  shape[STRIDE_DIM]     SHAPE_W each
  idx[STRIDE_DIM]       SHAPE_W each
  cur_addr              ADDR_W

init (ISSUE 入口):
  idx <= 0
  cur_addr <= base

step (每发出一个 beat):
  idx[0]++
  cur_addr += stride[0]
  if idx[0] == shape[0]:
    idx[0] = 0
    idx[1]++
    cur_addr += stride[1] - stride[0] * shape[0]   # 这里有乘法 → 预算
    ...

done:
  idx[STRIDE_DIM-1] == shape[STRIDE_DIM-1]
```

为消除内层乘法：在 FETCH 阶段预计算 `wrap_delta[i] = stride[i] - stride[i-1] * shape[i-1]`，存到工作寄存器；step 阶段只做加法。FETCH 因此多消耗 (STRIDE_DIM - 1) 个 cycle

做串行乘加，或硬连接STRIDE_DIM - 1个并行乘法器（4 cycle总延迟换面积）。

7.5 ISSUE ????

ISSUE内三段，背靠背流水：

```
read 端              xform 端             write 端
────────            ───────             ────────
地址 → noc_rd_req     data_in  → 缓冲       地址 → noc_wr_req
     ↓                         ↓                ↓
  noc_rd_resp ────────► xform_buf ────────► noc_wr_data
                      (transpose: 双缓冲 tile_buf)
```

- 无transpose：xform仅做pad/slice/deslice的拍数级处理（组合+一拍寄存），read→write实际2 cycle延迟
- 有transpose：xform用tile_buf SRAM（TILE_BUF_BYTES，2R/2W或双bank ping-pong），先填满整tile再换轴读出；read与write不能完全重叠，但tile N的写可以与tile N+1的读重叠（双缓冲）
- backpressure：write端被NoC阻塞→xform暂停→read端暂停（valid/ready反向传播）

每条channel的ISSUE在不抢仲裁前提下能跑满NoC单master带宽。

7.6 ?? FIFO

项	实现
存储	1R/1W SRAM或reg array，深度K_IN + K_OUT + K_SIG = 12，宽度HANDLE_W + COUNT_W + 1bit(op)
写端	NOTIFY态按顺序写入in_list（op=DELTA, delta=负）、out_list（正）、sig_list（正）
读端	全channel共享的ctrl_m仲裁器（§9.2）轮询拉取

ISSUE完成即开始NOTIFY；DONE不等Notify FIFO排空（FIFO是channel内的，已经记录了"我要通知谁"）。Channel复位/free时必须等FIFO排空。

8. Sync Pipe Pool

8.1 counter ??

存储	规格
`sync_count[N_SYNC]`	N_SYNC × COUNT_W FF（小，全FF阵列）
`sync_alloc[N_SYNC]`	N_SYNC bit

每个counter单cycle R/W，支持`+=delta`、`=val`、读出。

8.2 pending wait CAM

全单元 (含 channel 自身 count + sync count) 共用一个 pending wait 表。

项	实现
容量	N_PEND_WAIT
每项	{valid, target_handle (HANDLE_W), threshold (COUNT_W), resp_tag (4 bit)}
写端	收到 PWAIT 时入表 (若 count 已达, 立即响应不入表)
比较端	每次任何 count 更新后, 对所有 valid 项广播 `count >= threshold`; 命中项发 `resp` 并 invalidate

广播比较: N_PEND_WAIT = 16 时是 16 个 COUNT_W 比较器, 单 cycle。

8.3 alloc / free

与 channel pool 完全独立的另一个 priority encoder + bitmap, 结构同 §6。

9. ??????

9.1 NoC ? / ?

两个 round-robin 仲裁器, 输入 N_CH 路, 输出 1 路 master。

- 仲裁粒度: burst (一条 NoC 请求 = 一段连续 beat), 不要在 burst 中间切
- pending request 数: 每 channel 同时挂 OUT_RD 个未完成请求; 总单元 OUT_RD × N_CH 个 entry 的回 resp 路由表 (由 `id` 字段索引)

9.2 ctrl_m ????

输入: N_CH 路 Notify FIFO 头 + Sync Pool 的 ack 路径 = N_CH + 1 路。Round-robin 单 cycle 仲裁。

9.3 ctrl_s ??

```
case (req_op)
  PALLOC, PFREE:      → Pool Mgr (channel pool) / Sync Pool
  PSEND:              → Channel[req_handle.pipe_id].queue
  PREAD/PWAIT/PSET:   → 根据 handle.type 路由
  PSIGNAL:            → Sync Pool
  COUNT_DELTA:        → 根据 handle.type 路由到 Channel.count 或 Sync Pool.sync_count
  COUNT_QUERY:        → 同上, 比较后立即回 resp (必命中或失败)
endcase
```

PALLOC 的 type 字段区分 engine (→ Pool Mgr) 还是 sync (→ Sync Pool)。

10. ????????

10.1 channel ? pending wait CAM

- channel 的 count 更新（DONE 态、PSET、外部 COUNT_DELTA）作为广播事件
- 物理实现: channel 把 `count_update_pulse + new_count + handle` 送进 CAM 的比较口
- 多个 channel 同 cycle 更新: CAM 端串行接受（每 cycle 1 路），channel 端 backpressure；冲突概率低（DONE 不会很频繁）

10.2 channel ? ctrl_m

- channel Notify FIFO 头部直接挂仲裁器; 不需要中间 buffer
- ctrl_m 主端被对端拉低 ready 时, 整体停发; channel 内部 FIFO 自然填满后 backpressure 到 NOTIFY 状态

10.3 channel ? NoC

- read: channel 发 req → NoC 仲裁 → 远端; resp 通过 `id` 直接回到正确 channel 的 read fifo
- write: channel 同时驱动 wr_addr 和 wr_data; 数据宽度对齐由地址生成器保证

11. ?????

信号 / 状态	复位值
`alloc_bitmap` (channel + sync)	全 0
`meta[*]`	invalid
所有 `count` 寄存器	0
FSM	IDLE
queue head / tail	0
pending wait CAM	全 invalid
Notify FIFO	空
NoC outstanding 表	空

复位释放后, 单元立刻可接受 PALLOC。不需要软件 "init" 指令。

12. ??????

信号	宽度	用途
`dbg_ch_state[N_CH]`	N_CH × 3	每 channel FSM 状态
`dbg_ch_count[N_CH]`	N_CH × COUNT_W	每 channel 完成计数
`dbg_queue_occ[N_CH]`	N_CH × log2(Q_MAX)+1	每 channel 队列填充
`dbg_pend_wait_occ`	log2(N_PEND_WAIT)+1	未决 wait 数
`dbg_noc_rd_outstanding`	log2(OUT_RD×N_CH)+1	NoC 读未决数

信号	宽度	用途
`dbg_alloc_bitmap`	N_CH + N_SYNC	pool 占用

所有 dbg 信号同步组合输出，外部可通过 APB-like 单独的 debug 总线读出。生产环境可参数化掉。

13. ??????

下列由芯片产品定义决定，不在本设计文档锁定：

1. `N_CH / N_SYNC / Q_MAX / N_PEND_WAIT` 的实际数值
2. NoC 协议细节（具体是 AXI4 / CHI / 自研）
3. `TILE_BUF_BYTES` 与最大 transpose tile 大小
4. ctrl_s / ctrl_m 总线宽度与是否允许多 outstanding
5. 是否例化 `xform_flags.pad / slice / deslice` 的全部组合，还是某些组合非法
6. `OUT_RD`（每 channel NoC 未决读请求数），影响吞吐与回路 buffer 面积
7. 调试总线接口形式

14. ??? Pipe ??

抽象层概念	本文档落点
engine pipe 契约（输入序列 → 输出序列 + credit）	§7.1 队列 + §7.2 count + §7.3 FSM
sync pipe 契约	§8.1 counter + §8.2 pending wait
落地 A（每单元独立 ID）	§6.1 + §8.3 各自独立 bitmap
落地 B（控制/数据总线分开）	§3.1/3.2 vs §3.3
落地 D（sync 独立硬件）	§8 与 §6/§7 不共用任何状态
特性 6（硬件零检查）	§5.4 + §6.3 + §7.1 满空不检
特性 7（实例创建即不可变）	§6.2 alloc 后参数定，§7 内部状态命令级
特性 8（三类 pipe）	engine 在 §7，sync 在 §8

Revision #1
Created 2026-05-13 13:49:41 UTC by Colin
Updated 2026-05-13 13:49:41 UTC by Colin