



ventus-gpgpu-verilog 设计文档

ventus-gpgpu-verilog-release-v1.0-spec

目录

1 VENTUS 硬件简介	3
2 VENTUS 信号描述	4
2.1 信号描述表	4
2.2 RTL 参数设置	6
3 CTA SCHEDULER	6
3.1 概述	6
3.2 信号描述	7
3.3 详细设计	8
3.3.1 <i>inflight_wg_buffer</i>	8
3.3.2 <i>allocator_neo</i>	12
3.3.3 <i>dis_controller</i>	14
3.3.4 <i>top_resource_table</i>	17
3.3.5 <i>GPU_interface</i>	22
4 PIPELINE	27
4.1 概述	27
4.2 详细设计	28
4.2.1 <i>WarpScheduler</i>	28
4.2.2 <i>InstrDecode</i>	33
4.2.3 <i>InstrBuffer</i>	35
4.2.4 <i>ScoreBoard</i>	37
4.2.5 <i>OperandCollector</i>	39
4.2.6 <i>Issue</i>	45
4.2.7 <i>SimtStack</i>	51
4.2.8 <i>sALU</i>	55
4.2.9 <i>vALU</i>	56
4.2.10 <i>vMUL</i>	57
4.2.11 <i>CSR</i>	62
4.2.12 <i>TensorCore</i>	64
4.2.13 <i>LSU</i>	66
4.2.14 <i>FPU</i>	76
4.2.15 <i>SFU</i>	83
4.2.16 写回单元	90
5 内存系统	90
5.1 概述	90
5.2 详细设计	91
5.2.1 <i>Icache</i>	91
5.2.2 <i>sharablemem</i>	93
5.2.3 <i>Dcache</i>	95
5.2.4 <i>L2 Cache</i>	103

1 Ventus 硬件简介

在硬件上，ventus 主要包括 CTA Scheduler、多个 SM 内核以及 L2 Cache 组成。其中，CTA 负责接收来自 host 的任务请求，并将任务分配给 SM；SM 内部包括执行流水线（pipeline）、共享存储器（Sharemem）以及 L1 Cache（L1\$D 和 L1\$I），pipeline 内部为 6 级流水；L2 Cache 负责从外部存储器中取指令和数据。

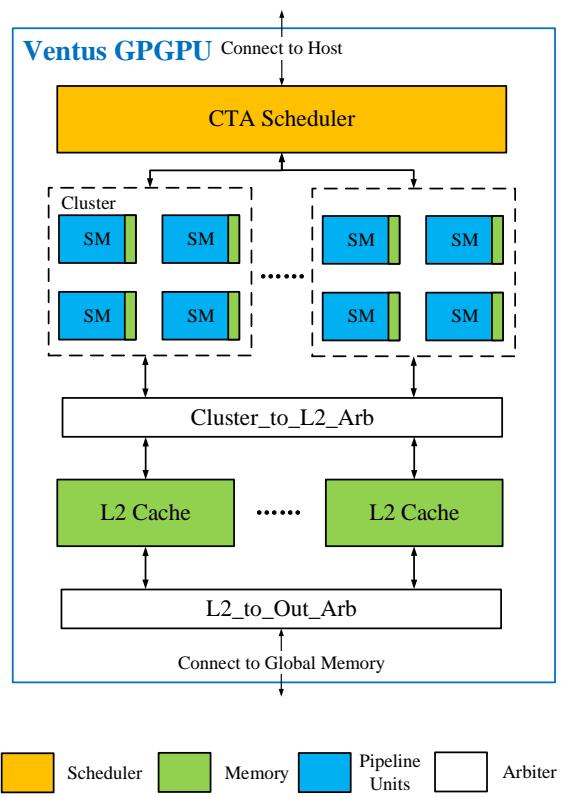


图 1-1 ventus 硬件结构框图

2 Ventus 信号描述

2.1 信号描述表

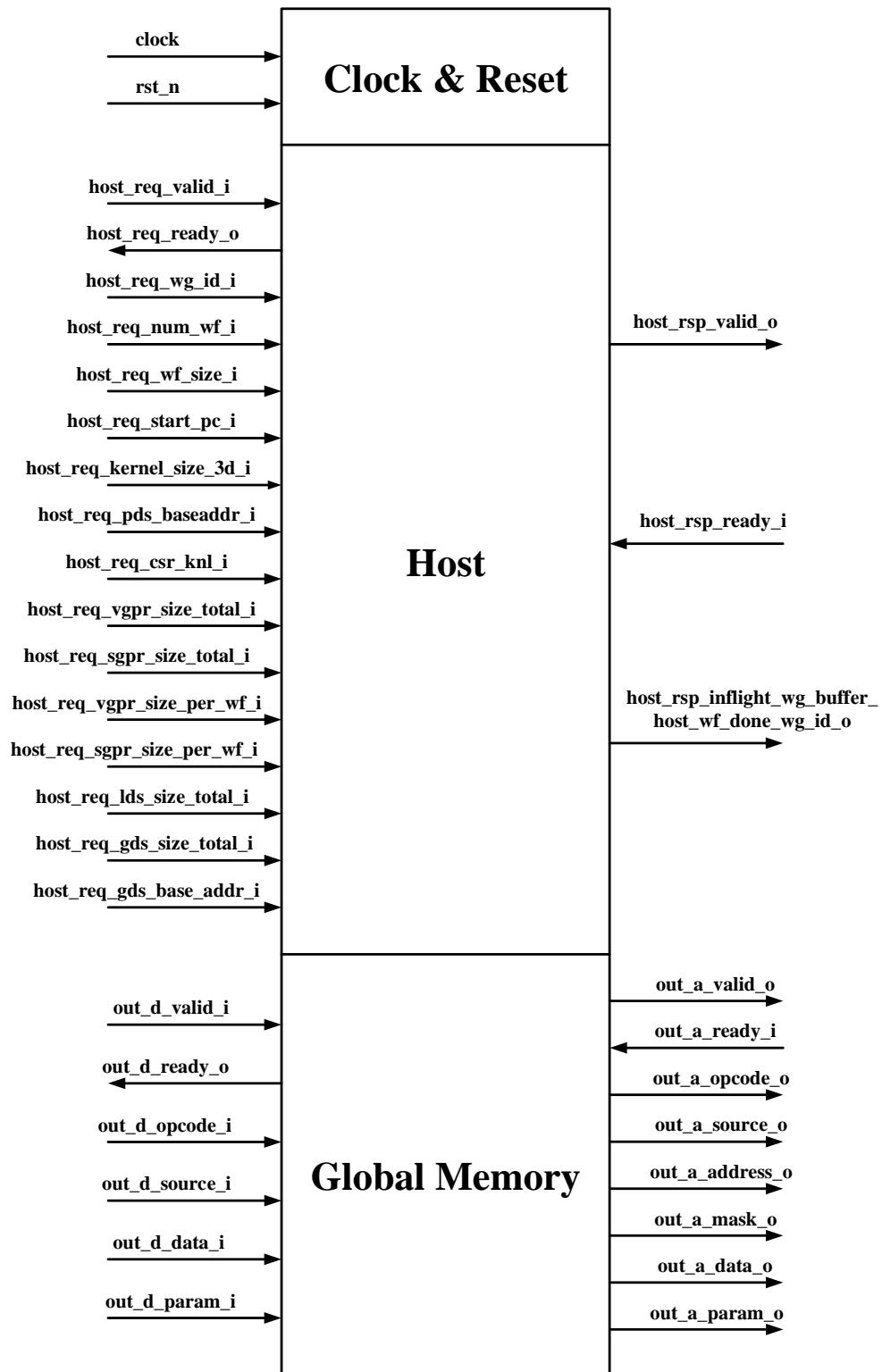


图 2-1 Ventus GPGPU 接口图

名称	类型	位宽	描述
时钟及复位信号			
clock	IN	1	系统时钟
rst_n	IN	1	全局复位，低有效
From Host			
host_req_valid_i	IN	1	host 请求有效信号
host_req_ready_o	OUT	1	GPGPU 就绪信号
host_req_wg_id_i	IN	6	workgroup id
host_req_kernel_size_3d_i	IN	10*3	该 workgroup 在 NDRANGE 中的 x id、y id 和 z id
host_req_num_wf_i	IN	4	该 workgroup 中 warp 数量
host_req_wf_size_i	IN	10	一个 warp 中 thread 数量
host_req_start_pc_i	IN	32	pc 开始地址
host_req_gds_baseaddr_i	IN	32	该 workgroup 分配的 global memory 的 baseaddr
host_req_vgpr_size_total_i	IN	11	该 workgroup 使用的 vgpr 个数
host_req_sgpr_size_total_i	IN	11	该 workgroup 使用的 sgpr 个数
host_req_lds_size_total_i	IN	18	该 workgroup 使用的 local memory 空间
host_req_gds_size_total_i	IN	11	该 workgroup 使用的 global memory 空间
host_req_pds_baseaddr_i	IN	32	该 workgroup 分配的 private memory 的 baseaddr
host_req_csr_knl_i	IN	32	该 workgroup 的 metadata buffer 的 baseaddr
host_req_vgpr_size_per_wf_i	IN	11	单个 warp 使用的 vgpr 个数
host_req_sgpr_size_per_wf_i	IN	11	单个 warp 使用的 sgpr 个数
To host			
host_rsp_valid_o	OUT	1	接收到 host 的输入所返回的响应
host_rsp_ready_i	IN	1	workgroup 完成信号
host_rsp_inflight_wg_buffer_host_wf_d one_wg_id_o	OUT	6	所完成 warp 的 workgroup id
To Mem			
out_a_valid_o	OUT	1	请求有效信号
out_a_ready_i	IN	1	Mem 就绪信号
out_a_opcode_o	OUT	3	操作码
out_a_size_o	OUT	3	传输字节数的对数
out_a_source_o	OUT	12	发射该请求的主端源标识符
out_a_address_o	OUT	32	地址
out_a_mask_o	OUT	8	字节掩码
out_a_data_o	OUT	64	数据
out_a_param_o	OUT	3	参数码
To Mem			

out_d_valid_i	IN	1	Mem 响应有效信号
out_d_ready_o	OUT	1	接收响应就绪信号
out_d_opcode_i	IN	3	操作码
out_d_source_i	IN	12	发射该请求的主端源标识符
out_d_data_i	IN	64	数据
out_d_param_i	IN	3	参数码

2.2 RTL 参数设置

名称	默认值	描述
NUM_THREAD	0x32	一个 warp 内的 thread 数
NUM_CLUSTER	0x1	Cluster 的个数
NUM_SM_IN_CLUSTER	0x2	一个 Cluster 内的 SM 数
NUM_WARP	0x8	一个 SM 内允许的最大 warp 数
NUM_BLOCK	0x8	一个 SM 内允许的最大 workgroup 数
NUM_LANE	0x32	硬件上一个 SM 的运算单元里能一次同时处理的 thread 数
NUM_FETCH	0x2	每次取指请求得到的指令个数
NUM_VGPR	0x400	一个 SM 内的向量通用寄存器个数
NUM_SGPR	0x400	一个 SM 内的标量通用寄存器个数

3 CTA Scheduler

3.1 概述

CTA Scheduler 模块的主要功能是任务分配, host (CPU) 发送给 GPU 的任务以 workgroup 为基本单位, 由 CTA Scheduler 接收, CTA Scheduler 会按 block 中包含的总 warp 数信息, 以及需要占用的 local memory、sharedmemory 大小, 将 block 对应分配到空闲 (即剩余资源足够) 的 SM 上。在硬件层面, 将按照 32 个 thread 组成一个 warp 的形式, 作为整体在 SM 硬件上进行调度。同一个 block 的 warp 只能在同一个 SM 上运行, 但是同一 SM 可以容纳来自不同 block 甚自不同 grid 的若干个 warp。

CTA Scheduler 以 warp 为单位逐个发送给 SM, 同一 workgroup 的 warp 会分配到同一 SM 中, warp_slot_id 的低位即表明了该 warp 在当前 workgroup 中的 id, 高位表明了 workgroup 本身所属的 id。相应的, SM 会通过此 id, 计算出当前 warp 在所属 workgroup 中的位置, 并将该值置于 CSR 寄存器中, 供软件使用。分配的 localmem baseaddr 需要通过 CSR 读取, 而 privatemem baseaddr 和 register base 则由硬件隐式映射。

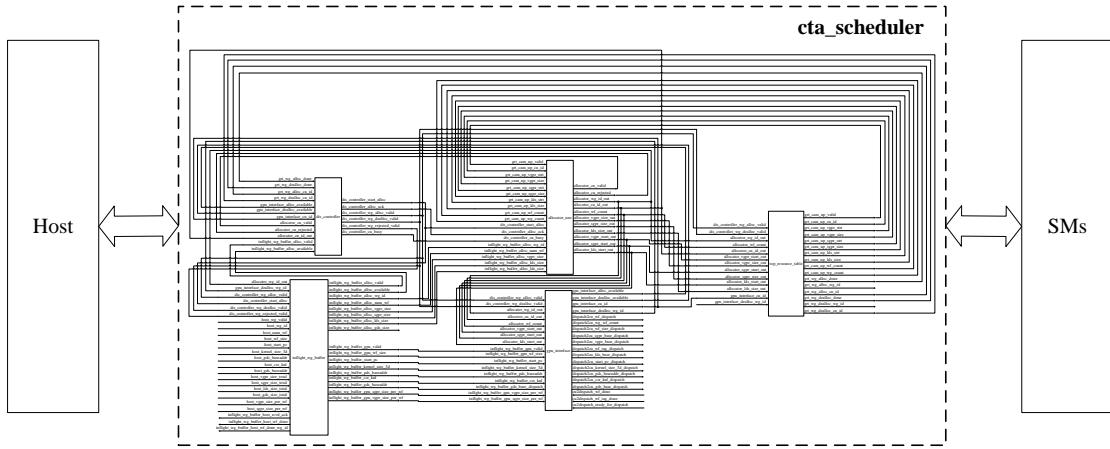


图 3-1 CTA Scheduler 的硬件框图

图 3-1 是 CTA 的硬件框图：

(1) host 发送任务给 CTA Scheduler 模块并接收执行成功的响应，CTA Scheduler 模块将 wg 分配给各个 SM，SM 以 warp 为单位执行并将执行完成信号发送给 CTA Scheduler 模块。

(2) CTA Scheduler 模块中主要分为五个模块，inflight_wg_buffer 模块负责缓存来自 host 的任务，allocator_neo 模块负责对比 SM 剩余资源与该任务所需资源，判断任务能否执行，dis_controller 模块负责产生其他模块的控制信号，top_resource_table 模块负责存储各个 SM 的剩余资源信息并在每一次任务后进行更新，gpu_interface 模块负责给 SM 发送任务并接收 SM 的任务完成信号。

3.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
host			
host_wg_valid_i	IN	1	发送使能
host_wg_id_i	IN	6	workgroup 的 id
host_kernel_size_3d_i	IN	10*3	该 workgroup 在 NDRange 中的 x id、y id 和 z id
host_num_wf_i	IN	4	该 workgroup 中 warp 数量
host_wf_size_i	IN	10	一个 warp 中 thread 数量
host_start_pc_i	IN	32	pc 开始地址
host_gds_baseaddr_i	IN	32	该 workgroup 分配的 global memory 的 baseaddr
host_vgpr_size_total_i	IN	11	该 workgroup 使用的 vgpr 个数
host_sgpr_size_total_i	IN	11	该 workgroup 使用的 sgpr 个数
host_lds_size_total_i	IN	18	该 workgroup 使用的 local memory 空间

host_gds_size_total_i	IN	11	该 workgroup 使用的 global memory 空间
host_pds_baseaddr_i	IN	32	该 workgroup 分配的 private memory 的 baseaddr
host_csr_knl_i	IN	32	该 workgroup 的 metadata buffer 的 baseaddr
host_vgpr_size_per_wf_i	IN	11	单个 warp 使用的 vgpr 个数
host_sgpr_size_per_wf_i	IN	11	单个 warp 使用的 sgpr 个数
inflight_wg_buffer_host_rcvd_ack_o	OUT	1	接收到 host 的输入所返回的响应
inflight_wg_buffer_host_wf_done_o	OUT	1	warp 完成信号
inflight_wg_buffer_host_wf_done_wg_id_o	OUT	6	所完成 warp 的 workgroup id
SM			
dispatch2cu_wf_dispatch_o	OUT	2	执行该 workgroup 的 SM id
dispatch2cu_wf_wf_count_o	OUT	4	workgroup 中 warp 数量
dispatch2cu_wf_size_dispatch_o	OUT	10	warp 中 thread 数量
dispatch2cu_sgpr_base_dispatch_o	OUT	11	sgpr 开始位置
dispatch2cu_vgpr_base_dispatch_o	OUT	11	vgpr 开始位置

3.3 详细设计

下面介绍 CTA Scheduler 的一些内部子模块。

3.3.1 inflight_wg_buffer

3.3.1.1 概述

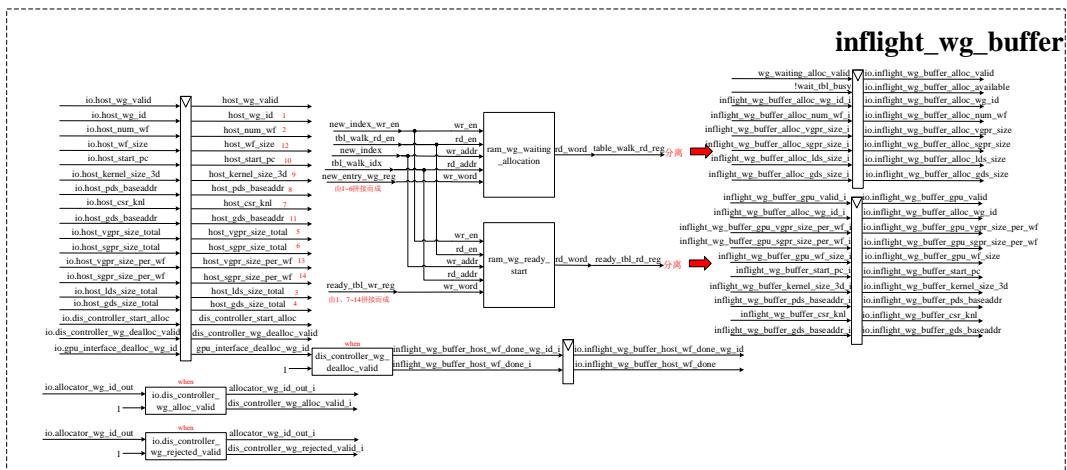


图 3-2 inflight_wg_buffer 的硬件框图

框图说明：

- (1) inflight_wg_buffer 模块中有两个 RAM，各有 NUMBER_ENTRIES 个存储空间（设置

为 2), 其中 `ram_wg_waiting_allocation` 存储的是发送给 `allocator_neo` 模块的数据 (与 SM 中剩余资源进行比较), `ram_wg_ready_start` 存储的是发送给 `gpu_interface` 模块的数据。

(2) 此模块的功能主要是接收 `host` 发来请求并进行缓存 (同时存入两个 `ram`, 将 `waiting_tbl_valid` 置高), 然后给 `host` 发送接收成功的响应, 其中存入 `ram` 时会选择空闲的最低地址。一旦 `ram` 中的 `workgroup` 被选中并发送给 `allocator_neo` 模块(选择方式为轮询), `inflight_wg_buffer` 模块就会向 `dis_controller` 模块发送信号, 标志着 `alloc` 开始。接着将所选择的 `workgroup` 置为 `waiting_tbl_pending` 状态, 表明需要等待 `allocator_neo` 模块的判断结果 (先发送给 `dis_controller` 控制模块), 若允许 `alloc` 则在 `ram` 中寻找该 `workgroup`, 将数据输出给 `gpu_interface` 模块同时清除该地址的数据, 若拒绝 `alloc` 则不清除 `ram` 中的数据, 等待下一次 `alloc`。

3.3.1.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位, 低有效
<code>host</code>			
<code>host_wg_valid_i</code>	IN	1	发送使能
<code>host_wg_id_i</code>	IN	6	<code>workgroup</code> 的 id
<code>host_kernel_size_3d_i</code>	IN	10*3	该 <code>workgroup</code> 在 <code>NDRange</code> 中的 x id、y id 和 z id
<code>host_num_wf_i</code>	IN	4	该 <code>workgroup</code> 中 warp 数量
<code>host_wf_size_i</code>	IN	10	一个 warp 中 thread 数量
<code>host_start_pc_i</code>	IN	32	pc 开始地址
<code>host_gds_baseaddr_i</code>	IN	32	该 <code>workgroup</code> 分配的 global memory 的 baseaddr
<code>host_vgpr_size_total_i</code>	IN	11	该 <code>workgroup</code> 使用的 vgpr 个数
<code>host_sgpr_size_total_i</code>	IN	11	该 <code>workgroup</code> 使用的 sgpr 个数
<code>host_lds_size_total_i</code>	IN	18	该 <code>workgroup</code> 使用的 local memory 空间
<code>host_gds_size_total_i</code>	IN	11	该 <code>workgroup</code> 使用的 global memory 空间
<code>host_pds_baseaddr_i</code>	IN	32	该 <code>workgroup</code> 分配的 private memory 的 baseaddr
<code>host_csr_knl_i</code>	IN	32	该 <code>workgroup</code> 的 metadata buffer 的 baseaddr
<code>host_vgpr_size_per_wf_i</code>	IN	11	单个 warp 使用的 vgpr 个数
<code>host_sgpr_size_per_wf_i</code>	IN	11	单个 warp 使用的 sgpr 个数
<code>inflight_wg_buffer_host_rcvd_ack_o</code>	OUT	1	接收到 <code>host</code> 的输入所返回的响应

inflight_wg_buffer_host_wf_done_o	OUT	1	warp 完成信号
inflight_wg_buffer_host_wf_done_wg_id_o	OUT	6	所完成 warp 的 workgroup id
dis_controller			
dis_controller_wg_alloc_valid_i	IN	1	允许 alloc
dis_controller_start_alloc_i	IN	1	开始 alloc
dis_controller_wg_dealloc_valid_i	IN	1	允许 dealloc
dis_controller_wg_rejected_valid_i	IN	1	拒绝 alloc
inflight_wg_buffer_alloc_valid_o	OUT	1	为 1 时表示 wg 已经发送给 allocator_neo
inflight_wg_buffer_alloc_available_o	OUT	1	为 1 时表示 inflight_wg_buffer 模块在等待 alloc 结果
allocator_neo			
inflight_wg_buffer_alloc_wg_id_o	OUT	6	workgroup id
inflight_wg_buffer_alloc_num_wf_o	OUT	4	该 workgroup 中 warp 数量
inflight_wg_buffer_alloc_vgpr_size_o	OUT	11	该 workgroup 使用的 vgpr 个数
inflight_wg_buffer_alloc_sgpr_size_o	OUT	11	该 workgroup 使用的 sgpr 个数
inflight_wg_buffer_alloc_lds_size_o	OUT	18	该 workgroup 使用的 local memory 空间
inflight_wg_buffer_alloc_gds_size_o	OUT	11	该 workgroup 使用的 global memory 空间
allocator_wg_id_out_i	IN	6	alloc 的 workgroup id
gpu_interface			
inflight_wg_buffer_gpu_valid_o	OUT	1	输出使能
inflight_wg_buffer_gpu_vgpr_size_per_wf_o	OUT	11	单个 warp 使用的 vgpr 个数
inflight_wg_buffer_gpu_sgpr_size_per_wf_o	OUT	11	单个 warp 使用的 sgpr 个数
inflight_wg_buffer_gpu_wf_size_o	OUT	10	一个 warp 中 thread 数量
inflight_wg_buffer_start_pc_o	OUT	32	pc 开始地址
inflight_wg_buffer_kernel_size_3d_o	OUT	10*3	该 workgroup 在 NDRANGE 中的 x id、y id 和 z id
inflight_wg_buffer_pds_baseaddr_o	OUT	32	该 workgroup 分配的 private memory 的 baseaddr
inflight_wg_buffer_csr_knl_o	OUT	32	该 workgroup 的 metadata buffer 的 baseaddr
inflight_wg_buffer_gds_baseaddr_o	OUT	32	该 workgroup 分配的 global memory 的 baseaddr
gpu_interface_dealloc_wg_id_i	IN	6	dealloc 的 workgroup id

3.3.1.3 设计原理

本模块主要包括两个状态机，分别负责和 host 以及 allocator_neo 模块交互。用三个向量（元素个数与 ram 存储空间一致）来表示 workgroup 的状态，waiting_tbl_valid 表示 ram

中该地址已经存入 workgroup，waiting_tbl_pending 表示该 workgroup 已经发送给 allocator_neo 模块，正在等待资源比较结果，valid_not_pending 表示 ram 中已有缓存但还未进行 alloc 判断。

下图所示为与 host 交互的状态机。

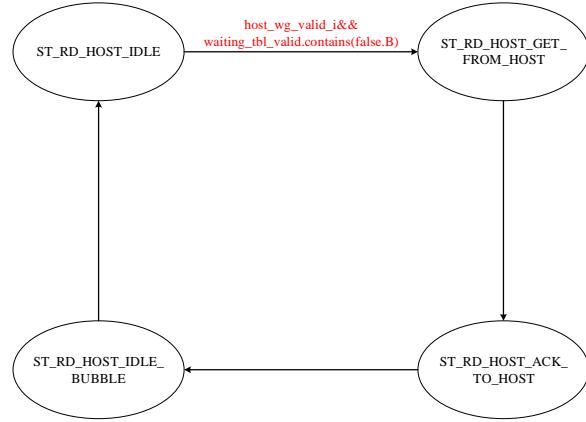


图 3-3 inflight_wg_buffer 与 host 交互的状态机跳转图

下图所示为与 allocator_neo 模块交互的状态机。

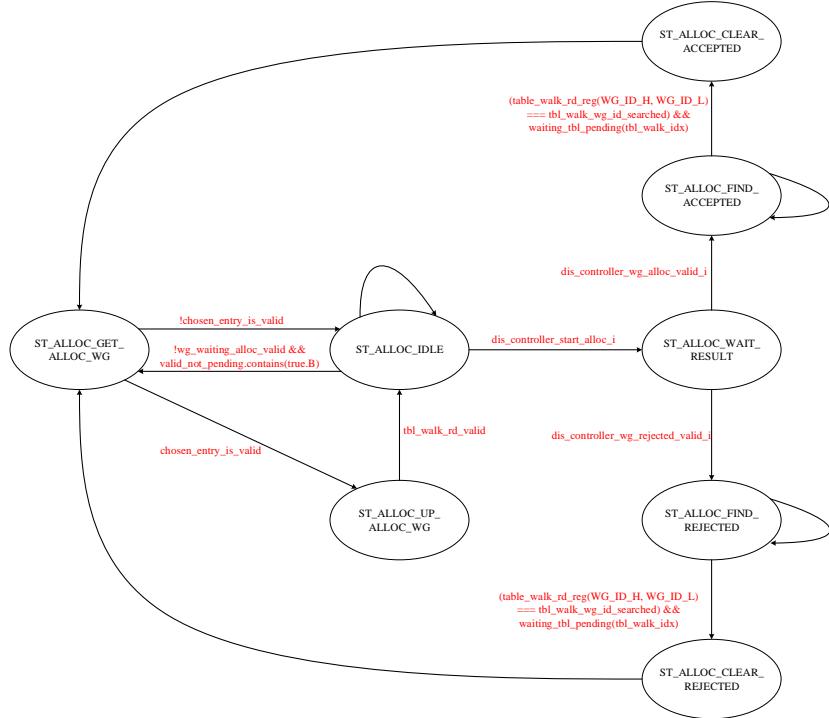


图 3-4 inflight_wg_buffer 与 allocator_neo 交互的状态机跳转图

3.3.2 allocator_neo

3.3.2.1 概述

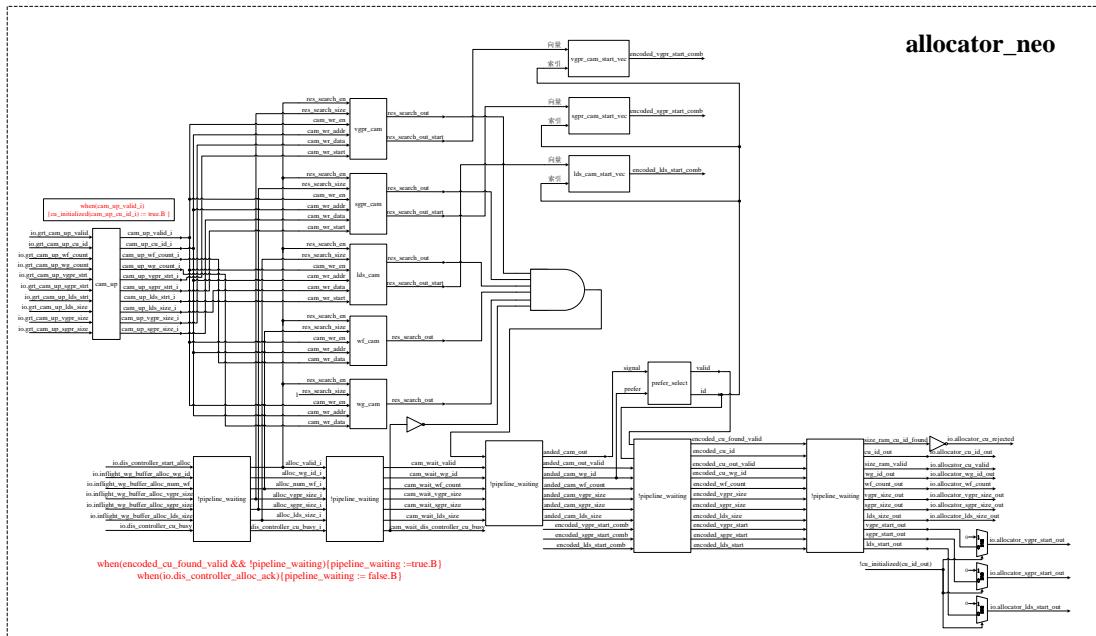


图 3-5 allocator_neo 的硬件框图

框图说明：

- (1) **cam_up** 是寄存器，目的是寄存来自 **top_resource_table** 模块的对应 SM 剩余资源。
- (2) **vgpr_cam**、**sgpr_cam**、**lds_cam**、**wf_cam** 和 **wg_cam** 这五个模块的作用是存储对应的剩余资源信息并与该次 **wg** 所需要的资源进行对比（有几个 SM 就需要比几次）。
- (3) 若没有 SM 满足要求，则拒绝该 **wg** 的执行，若有两个 SM 满足要求，通过 **prefer_select** 模块（轮询算法）选择一个 SM 执行。
- (4) **cu_initialized=0** 代表 SM 进行初始化，因此 **vgpr**、**sgpr** 和 **lds** 的起始位置从 0 开始，其他情况则是从 **top_resource_table** 模块计算得到的起始位置开始。
- (5) **pipeline_waiting** 初始值为 0，**allocator_neo** 模块正常运行，一旦找到可执行的 SM，**pipeline_waiting** 置 1，将当前流水线 stall 住，直到 **dis_controller** 控制模块发送的 **io.dis_controller_alloc_ack** 信号置 1 时流水线恢复。

3.3.2.2 信号说明

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
dis_controller			
dis_controller_start_alloc_i	IN	1	开始 alloc
dis_controller_alloc_ack_i	IN	1	alloc 响应

dis_controller_cu_busy_i	IN	1	SM 是否忙碌
allocator_cu_valid_o	OUT	1	输出有效使能
allocator_cu_rejected_o	OUT	1	拒绝 alloc
allocator_cu_id_out_o	OUT	2	该次分配的 SM id
inflight_wg_buffer			
inflight_wg_buffer_alloc_wg_id_o	IN	6	workgroup id
inflight_wg_buffer_alloc_num_wf_o	IN	4	该 workgroup 中 warp 数量
inflight_wg_buffer_alloc_vgpr_size_o	IN	11	该 workgroup 使用的 vgpr 个数
inflight_wg_buffer_alloc_sgpr_size_o	IN	11	该 workgroup 使用的 sgpr 个数
inflight_wg_buffer_alloc_lds_size_o	IN	18	该 workgroup 使用的 local memory 空间
inflight_wg_buffer_alloc_gds_size_o	IN	11	该 workgroup 使用的 global memory 空间
allocator_wg_id_out_i	OUT	6	alloc 的 workgroup id
top_resource_table			
grt_cam_up_valid_i	IN	1	cam 写入使能
grt_cam_up_cu_id_i	IN	2	SM id
grt_cam_up_vgpr strt_i	IN	10	vgpr 剩余最大空间的起始位置
grt_cam_up_vgpr_size_i	IN	11	vgpr 剩余最大空间的大小
grt_cam_up_sgpr strt_i	IN	10	sgpr 剩余最大空间的起始位置
grt_cam_up_sgpr_size_i	IN	11	sgpr 剩余最大空间的大小
grt_cam_up_lds strt_i	IN	17	lds 剩余最大空间的起始位置
grt_cam_up_lds_size_i	IN	18	lds 剩余最大空间的大小
grt_cam_up_wf_count_i	IN	4	workgroup 中剩余 warp 数量
grt_cam_up_wg_count_i	IN	4	剩余可分配的 workgroup 数量
allocator_wg_id_out_o	OUT	6	该次分配的 wg id
allocator_cu_id_out_o	OUT	2	该次分配的 SM id
allocator_wf_count_o	OUT	4	该次分配需要使用的 warp 数量
allocator_vgpr_size_out_o	OUT	11	该次分配需要使用的 vgpr 数量
allocator_sgpr_size_out_o	OUT	11	该次分配需要使用的 sgpr 数量
allocator_lds_size_out_o	OUT	18	该次分配需要使用的 lds 空间大小

allocator_vgpr_start_out_o	OUT	10	该次分配需要使用的 vgpr 开始位置
allocator_sgpr_start_out_o	OUT	10	该次分配需要使用的 sgpr 开始位置
allocator_lds_start_out_o	OUT	17	该次分配需要使用的 lds 开始位置

3.3.2.3 设计原理

vgpr、sgpr 和 lds 的 cam 模块接口如下图所示，其与 wf 和 wg 的主要区别在于多了起始位置。该模块中主要包含 1 个存储剩余资源数量的 ram 和 1 个存储剩余资源起始位置的 ram，每个 ram 的存储空间都是 NUMBER_CU 个，其中 NUMBER_CU 代表 SM 内核的个数，ventus 中设置为 2。每当有 wg 被 alloc 或 dealloc，意味着有资源被占用或者释放，top_resource_table 模块就会更新剩余最大资源以及其起始位置并发送给 allocator_neo 模块进行寄存，寄存地址为 SM 的 id。每当发送过来新的 wg，都会将其所需的资源与每个 SM 的剩余资源进行对比，从而输出一个 NUMBER_CU 位的数据（每一 bit 代表一个 SM 的该种剩余资源是否满足），只有五种资源都满足才代表该 SM 可以执行此 wg。

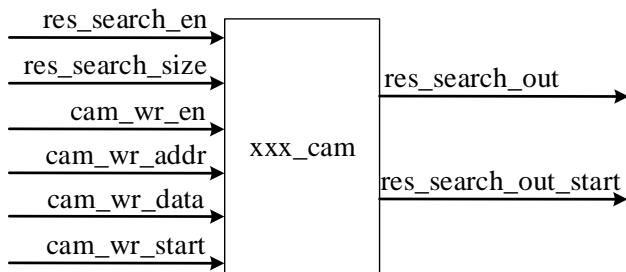


图 3-6 cam 的接口示意图

下表所示为 prefer_select 模块的逻辑，该模块的输入 signal 代表哪些 SM 有效，prefer 代表预期执行该 wg 的 SM 内核。同样以 RANGE=4 的情况为例，如下表所示。由于 for 循环是从 0 递增，因此高位地址的优先级高，由表可见，当初始 SM 为 0 时，SM 为 3 的优先级最高，即若位置 3 的 SM 有足够的资源就会优先执行该 SM，以此类推。

	prefer=0	prefer=1	prefer=2	prefer=3
signal	0123	0123	0123	0123
i+prefer_pre	0123	1230	2301	3012

3.3.3 dis_controller

3.3.3.1 概述

dis_controller 模块的作用主要是接收其他模块的工作状态并输出信号控制其他模块进行状态跳转。

3.3.3.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
inflight_wg_buffer			
inflight_wg_buffer_alloc_valid_i	IN	1	为 1 时表示 wg 已经发送给 allocator_neo
inflight_wg_buffer_alloc_available_i	IN	1	为 1 时表示 inflight_wg_buffer 模块在等待 alloc 结果
dis_controller_wg_alloc_valid_o	OUT	1	允许 alloc
dis_controller_start_alloc_o	OUT	1	开始 alloc
dis_controller_wg_dealloc_valid_o	OUT	1	允许 dealloc
dis_controller_wg_rejected_valid_o	OUT	1	拒绝 alloc
allocator_neo			
allocator_cu_valid_i	IN	1	输出有效使能
allocator_cu_rejected_i	IN	1	拒绝 alloc
allocator_cu_id_out_i	IN	2	该次分配的 SM id
dis_controller_start_alloc_o	OUT	1	开始 alloc
dis_controller_alloc_ack_o	OUT	1	alloc 响应
dis_controller_cu_busy_o	OUT	1	SM 是否忙碌
top_resource_table			
grt_wg_alloc_done_i	IN	1	alloc 结束
grt_wg_dealloc_done_i	IN	1	dealloc 结束
grt_wg_alloc_cu_id_i	IN	2	alloc 时的 SM id
grt_wg_dealloc_cu_id_i	IN	2	dealloc 时的 SM id
dis_controller_wg_alloc_valid_o	OUT	1	允许 alloc
dis_controller_wg_dealloc_valid_o	OUT	1	允许 dealloc
gpu_interface			
gpu_interface_alloc_available_i	IN	1	表示 gpu_interface 进入 alloc 状态
gpu_interface_dealloc_available_i	IN	1	表示 gpu_interface 进入 dealloc 状态
gpu_interface_cu_id_i	IN	2	需要 dealloc 的 SM id
dis_controller_wg_alloc_valid_o	OUT	1	允许 alloc
dis_controller_wg_dealloc_valid_o	OUT	1	允许 dealloc

3.3.3.3 设计原理

dis_controller 其主要的结构为状态机，如下图所示。

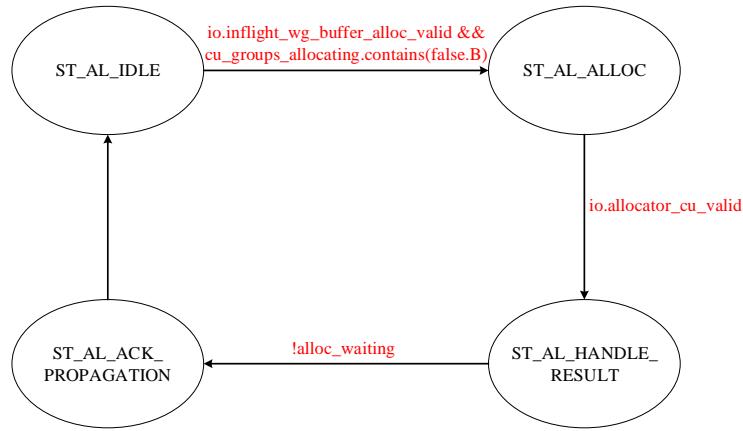


图 3-7 dis_controller 的状态机跳转图

初始为 ST_AL_IDLE 状态, cu_groups_allocating 信号为高时表示 resource_table 正在更新中, 因此当 allocator_neo 模块接收到 wg 任务 (io.inflight_wg_buffer_alloc_valid 为高) 且 resource_table 处于空闲状态 (cu_groups_allocating 为低) 时跳转到 ST_AL_ALLOC, 并且将输出给 allocator_neo 模块的信号 io_dis_controller_start_alloc 拉高, 通知 allocator_neo 模块进行剩余资源比较。

当 allocator_neo 模块完成对比并将结果输出 (io.allocator_cu_valid 为高), 将 alloc_waiting 拉高, 跳转进入 ST_AL_HANDLE_RESULT 状态。先判断是否为 deallocate 操作 (优先级比 alloc 高), 若是 deallocate 操作 (io.gpu_interface_dealloc_available 为高) 且 resource_table 处于空闲状态 (cu_groups_allocating 为低), 将输出信号 io_dis_controller_wg_dealloc_valid 和 cu_groups_allocating 拉高, 表明处于 deallocate 状态且 resource_table 正在更新。进一步判断该 alloc 操作是被拒绝还是接受, 若拒绝则将 alloc_waiting 拉低并将 io_dis_controller_wg_rejected_valid 拉高, 若接受则将 alloc_waiting 拉低并将 io_dis_controller_wg_alloc_valid 拉高, 同时还会让 resource_table 进行更新。

当处于 ST_AL_HANDLE_RESULT 状态且 alloc_waiting 为低时进入 ST_AL_ACK_PROPAGATION 状态, 最后回到 ST_AL_IDLE 状态。

3.3.4 top_resource_table

3.3.4.1 概述

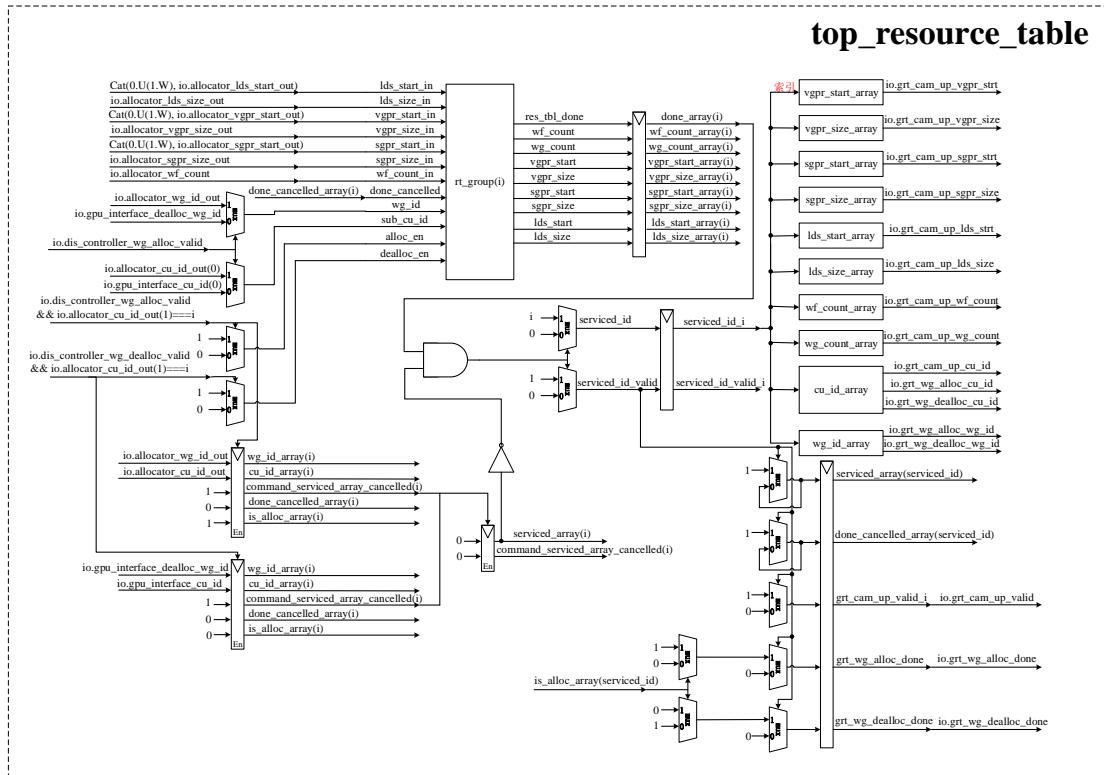


图 3-8 top_resource_table 的结构框图

框图说明：

(1) top_resource_table 模块的功能主要是计算并缓存 SM 内核中 vgpr、sgpr 和 lds 的剩余资源以及 workgroup 和 warp 的使用情况，其输入为来自 allocator_neo 模块的单次分配所需要的资源以及来自 gpu_interface 模块的释放已完成任务所占用资源，通过这些信号对 resource_table 进行更新，最后输出剩余资源以及开始位置。

(2) 上图中的 rt_group 是对 resource_table_group 模块的调用，例化次数为 NUMBER_RES_TABLE，在 ventus 中设置为 1，其具体框图如下图所示。其中 wf_slot_id_gen 模块中有两个 2×8 的向量空间（2 代表 SM 个数，8 代表 workgroup 个数，一个空间存储开关状态，另一个存储 wg_id），alloc 时根据 cu_id 找到当前行中最低位的空闲位置并置 1，同时在相同位置存入此时的 wg_id，dealloc 时若 wg_id 的值相等就将相应位置的 1 置为 0。然后将此存储位置输出给后续模块，作为该次分配在 resource_table 中的索引。

(3) lds_res_tbl、vgpr_res_tbl、sgpr_res_tbl、wf_res_tbl 和 wg_throttling 这五个模块分别代表 lds、vgpr、sgpr、warp 和 workgroup 的使用情况，alloc 时占用资源，dealloc 时释放资源。

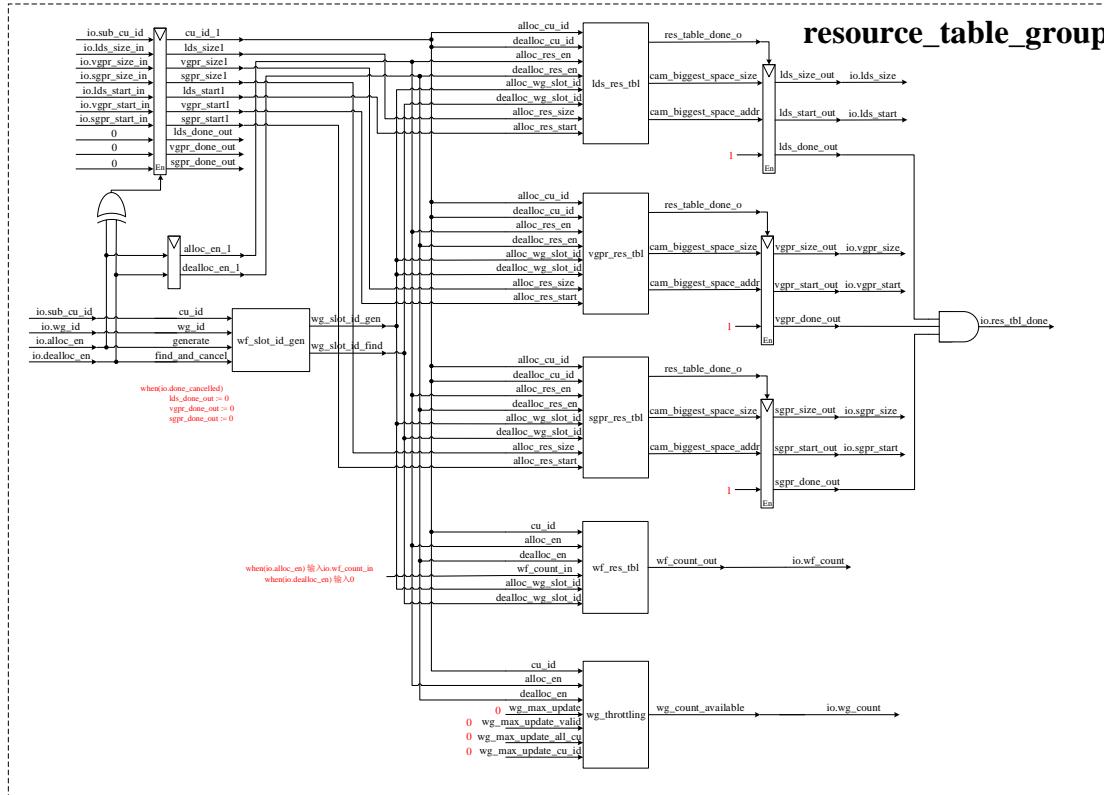


图 3-9 resource_table_group 的结构框图

3.3.4.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位, 低有效
dis_controller			
dis_controller_wg_alloc_valid_i	IN	1	允许 alloc
dis_controller_wg_dealloc_valid_i	IN	1	允许 dealloc
grt_wg_alloc_done_o	OUT	1	alloc 结束
grt_wg_dealloc_done_o	OUT	1	dealloc 结束
grt_wg_alloc_cu_id_o	OUT	2	alloc 时的 SM id
grt_wg_dealloc_cu_id_o	OUT	2	dealloc 时的 SM id
allocator_neo			
allocator_wg_id_out_i	IN	6	该次分配的 wg id
allocator_cu_id_out_i	IN	2	该次分配的 SM id
allocator_wf_count_i	IN	4	该次分配需要使用的 warp 数量
allocator_vgpr_size_out_i	IN	11	该次分配需要使用的 vgpr 数量

allocator_sgpr_size_out_i	IN	11	该次分配需要使用的 sgpr 数量
allocator_lds_size_out_i	IN	18	该次分配需要使用的 lds 空间大小
allocator_vgpr_start_out_i	IN	10	该次分配需要使用的 vgpr 开始位置
allocator_sgpr_start_out_i	IN	10	该次分配需要使用的 sgpr 开始位置
allocator_lds_start_out_i	IN	17	该次分配需要使用的 lds 开始位置
grt_cam_up_valid_o	OUT	1	cam 写入使能
grt_cam_up_cu_id_o	OUT	2	SM id
grt_cam_up_vgpr strt_o	OUT	10	vgpr 剩余最大空间的起始位置
grt_cam_up_vgpr_size_o	OUT	11	vgpr 剩余最大空间的大小
grt_cam_up_sgpr strt_o	OUT	10	sgpr 剩余最大空间的起始位置
grt_cam_up_sgpr_size_o	OUT	11	sgpr 剩余最大空间的大小
grt_cam_up_lds strt_o	OUT	17	lds 剩余最大空间的起始位置
grt_cam_up_lds_size_o	OUT	18	lds 剩余最大空间的大小
grt_cam_up_wf_count_o	OUT	4	workgroup 中剩余 warp 数量
grt_cam_up_wg_count_o	OUT	4	剩余可分配的 workgroup 数量
gpu_interface			
gpu_interface_cu_id_i	IN	2	dealloc 的 SM id
gpu_interface_dealloc_wg_id_i	IN	6	dealloc 的 workgroup id

3.3.4.3 设计原理

本模块的关键部分就是 lds_res_tbl、vgpr_res_tbl、sgpr_res_tbl、wf_res_tbl 和 wg_throttling 这五个剩余资源更新模块，接下来进行详细介绍。

首先 lds_res_tbl、vgpr_res_tbl、sgpr_res_tbl 都是调用的同一个模块，只是例化参数不同。其内部主要包括一个 ram 和四个状态机（main state machine、alloc state machine、dealloc state machine 和 find max state machine）。其中 ram 有 18 个存储空间，每个存储空间中的内容分为四部分，如下图所示。NEXT_ENTRY 为下一次分配的存储地址，PREV_ENTRY 为上一次分配的存储地址，位宽都是 4bit；RES_STRT 和 RES_SIZE 分别是该次分配所需要的该种资源的起始位置和数量。

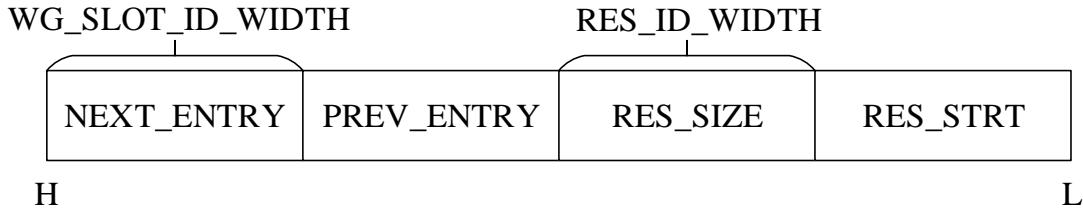


图 3-10 ram 的存储结构

主状态机如下图所示，其会根据输入的 alloc 和 dealloc 使能分别进入 alloc 状态机和 dealloc 状态机，接着进入 find max 状态机计算剩余最大空间的起始位置和大小。

Main state machine

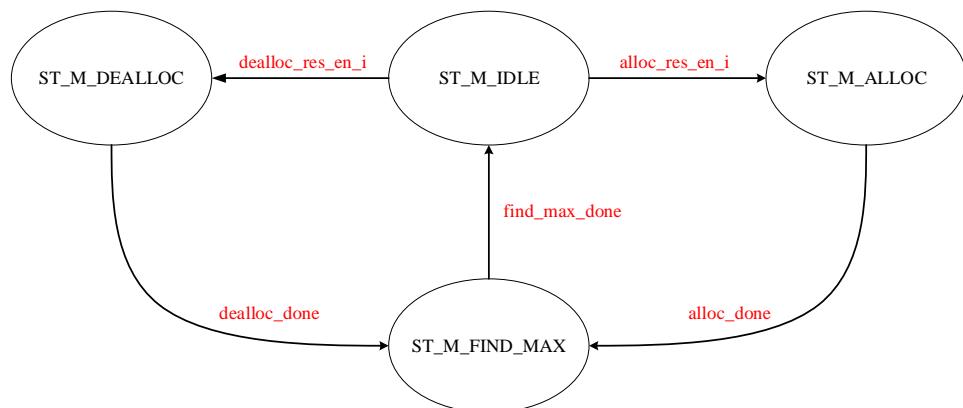
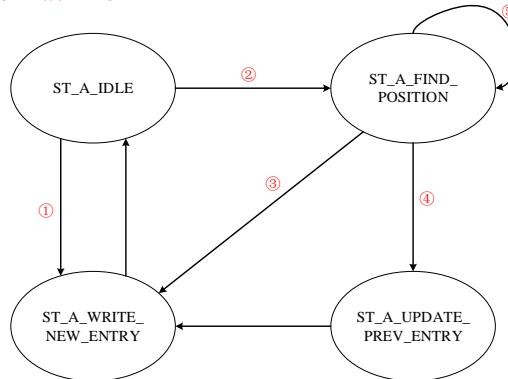


图 3-11 top_resource_table 的主状态机跳转图

Alloc 状态机如下图所示。

Alloc state machine



- ①: table_head_pointer_i === RES_TABLE_END_TABLE_U || !cu_initialized_i
- ②: !(table_head_pointer_i === RES_TABLE_END_TABLE_U || !cu_initialized_i)
- ③: (get_res_start(res_table_rd_reg) > alloc_res_start_i && get_prev_item_wg_slot(res_table_rd_reg) === RES_TABLE_HEAD_POINTER_U) || get_next_item_wg_slot(res_table_rd_reg) === RES_TABLE_END_TABLE_U
- ④: get_res_start(res_table_rd_reg) > alloc_res_start_i && !get_prev_item_wg_slot(res_table_rd_reg) === RES_TABLE_HEAD_POINTER_U
- ⑤: !get_res_start(res_table_rd_reg) > alloc_res_start_i && get_next_item_wg_slot(res_table_rd_reg) === RES_TABLE_END_TABLE_U

图 3-12 Alloc 状态机跳转图

Dealloc 状态机如下图所示。

Dealloc state machine

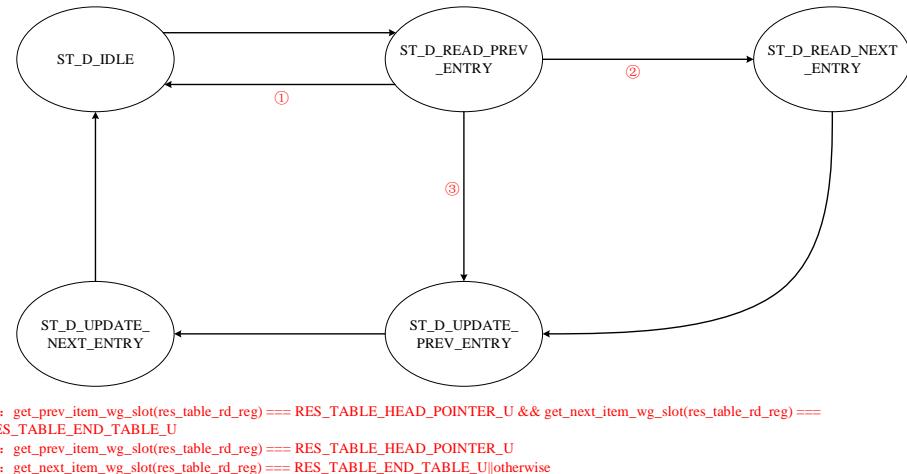


图 3-13 Dealloc 状态机跳转图

Find_max 状态机如下图所示。

Find max state machine

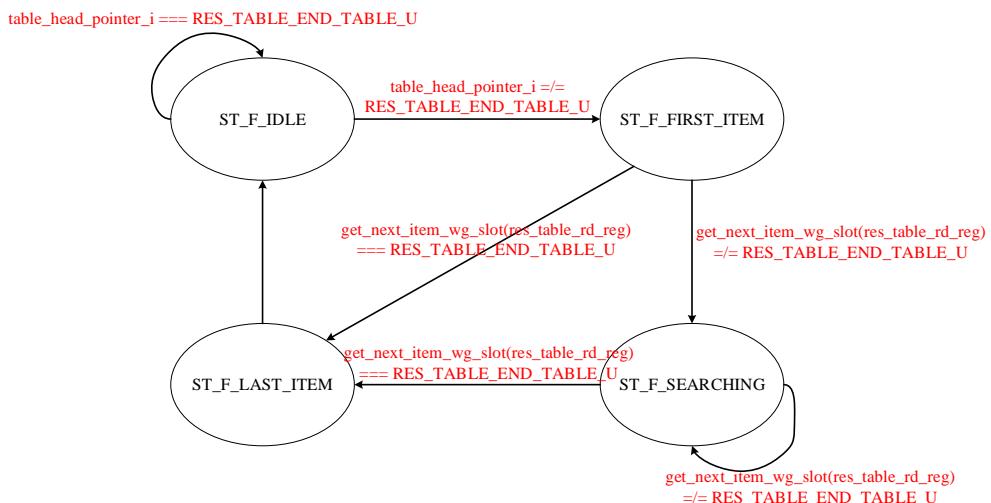


图 3-14 Find_max 状态机跳转图

wg_throttling 模块中有 2 个存储空间，对应 2 个 SM 的 workgroup 资源，当 alloc 时将可用 workgroup 资源减 1，当 dealloc 时将可用 workgroup 资源加 1。

wf_res_tbl 模块中有 2×8 个存储空间，分别对应 16 个 workgroup，其中存储着每个 workgroup 的 warp 剩余情况，当 alloc 时对应位置的剩余 warp 数量减 1，当 dealloc 时对应位置的剩余 warp 数量加 1。

dealloc 状态机的具体时序如下图所示，其中五个状态分别为 s0-s4，假设需要 dealloc 的 workgroup 的地址为 1，其前一个为 0，后一个为 2。

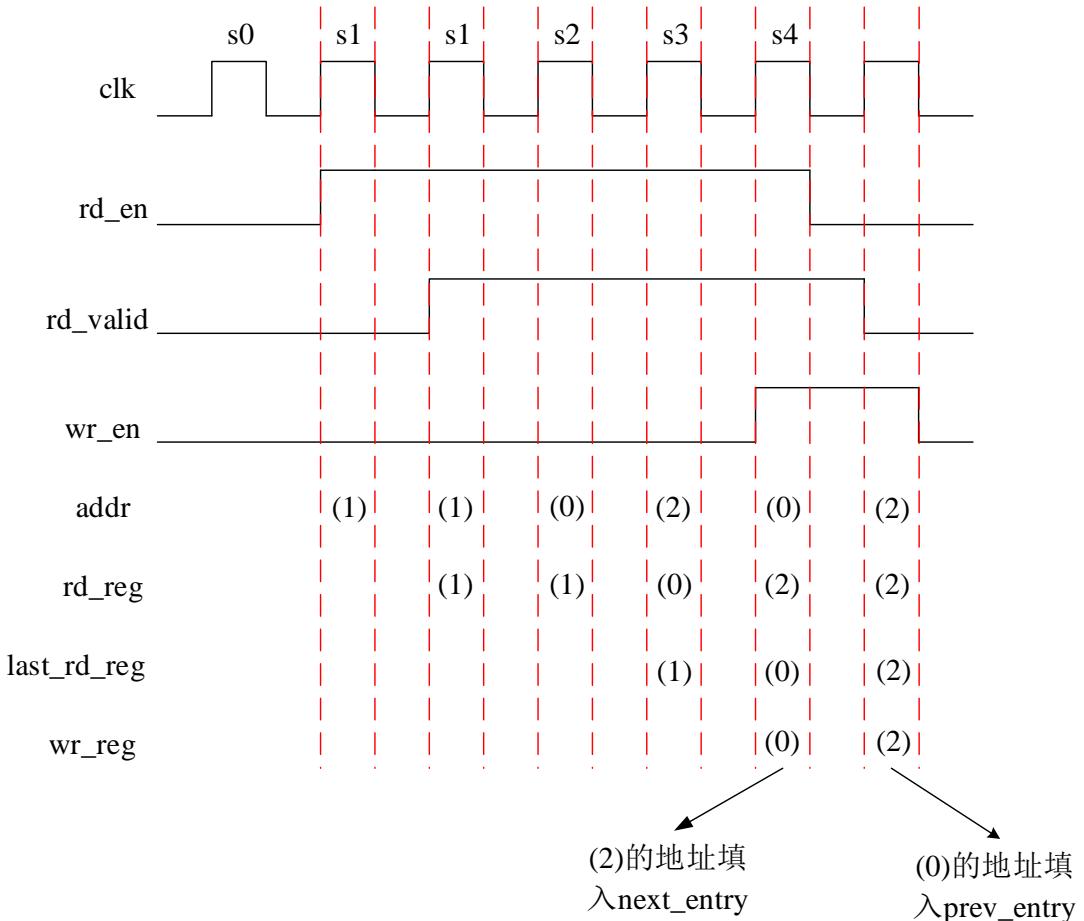


图 3-15 dealloc 状态机的时序图

3.3.5 GPU_interface

3.3.5.1 概述

GPU_interface 用于记录和更新已使用和未经使用过的 wg 的 id 信息、wf 信息。为 cta 提供下一次 dealloc 和 alloc 的资源状态分析。

3.3.5.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
Inflight_wg_buffer			
inflight_wg_buffer_gpu_valid_i	IN	1	Inflight 开启 alloc 的握手信号
inflight_wg_buffer_gpu_wf_size_i	IN	10	一个 warp 中的 thread 数量
inflight_wg_buffer_start_pc_i	IN	32	pc 开始的地址

inflight_wg_buffer_kernel_size_3d_i	IN	10*3	该 workgroup 在 NDRANGE 中的 x id、y id 和 z id
inflight_wg_buffer_pds_baseaddr_i	IN	32	该 workgroup 分配的 private memory 的 baseaddr
inflight_wg_buffer_csr_knl_i	IN	32	该 workgroup 的 metadata buffer 的 baseaddr
inflight_wg_buffer_gds_baseaddr_o	OUT	32	该 workgroup 分配的 global memory 的 baseaddr
inflight_wg_buffer_gpu_vgpr_size_per_wf_o	OUT	11	单个 warp 使用的 vgpr 个数
inflight_wg_buffer_gpu_sgpr_size_per_wf_o	OUT	11	单个 warp 使用的 sgpr 个数
allocator_neo			
allocator_wg_id_out_i	IN	6	该次分配的 wg id
allocator_cu_id_out_i	IN	2	该次分配的 SM id
allocator_wf_count_i	IN	4	该次分配需要使用的 warp 数量
allocator_vgpr_start_out_i	IN	10	该次分配需要使用的 vgpr 开始位置
allocator_sgpr_start_out_i	IN	10	该次分配需要使用的 sgpr 开始位置
allocator_lds_start_out_i	IN	17	该次分配需要使用的 lds 开始位置
discontrol			
dis_controller_wg_alloc_valid_i	IN	1	开始 alloc 的握手信号
dis_controller_wg_dealloc_valid_i	IN	1	开始 dealloc 的握手信号
gpu_interface_alloc_available_o	OUT	1	开始 alloc 的握手信号
gpu_interface_dealloc_available_o	OUT	1	开始 dealloc 的握手信号
discontrol & top_resoure_table			
gpu_interface_cu_id_o	OUT	1	deallc 的 cu_id
inflight_wg_buffer & top_resource_table			
gpu_interface_dealloc_wg_id_o	OUT	6	dealloc 的 wg_id
cta_scheduler			
dispatch2cu_wf_dispatch_o	OUT	2	执行该 workgroup 的 SM id
dispatch2cu_wg_wf_count_o	OUT	4	workgroup 中 warp 数量
dispatch2cu_wf_size_dispatch_o	OUT	10	warp 中 thread 数量
dispatch2cu_sgpr_base_dispatch_o	OUT	11	sgpr 开始位置
dispatch2cu_vgpr_base_dispatch_o	OUT	11	vgpr 开始位置
dispatch2cu_wf_tag_dispatch_o	OUT	7	warp id (高 3bit 索引 wg, 低 4bit 索引 warp)
dispatch2cu_lds_base_dispatch_o	OUT	18	lds 开始位置
dispatch2cu_start_pc_dispatch_o	OUT	32	pc 开始地址
dispatch2cu_kernel_size_3d_dispatch_o	OUT	10*3	该 workgroup 在 NDRANGE 中的 x id、y id 和 z id

dispatch2cu_pds_baseaddr_dispatch_o	OUT	32	该 workgroup 分配的 private memory 的 baseaddr
dispatch2cu_csr_knl_dispatch_o	OUT	32	该 workgroup 的 metadata buffer 的 baseaddr

3.3.5.3 设计原理

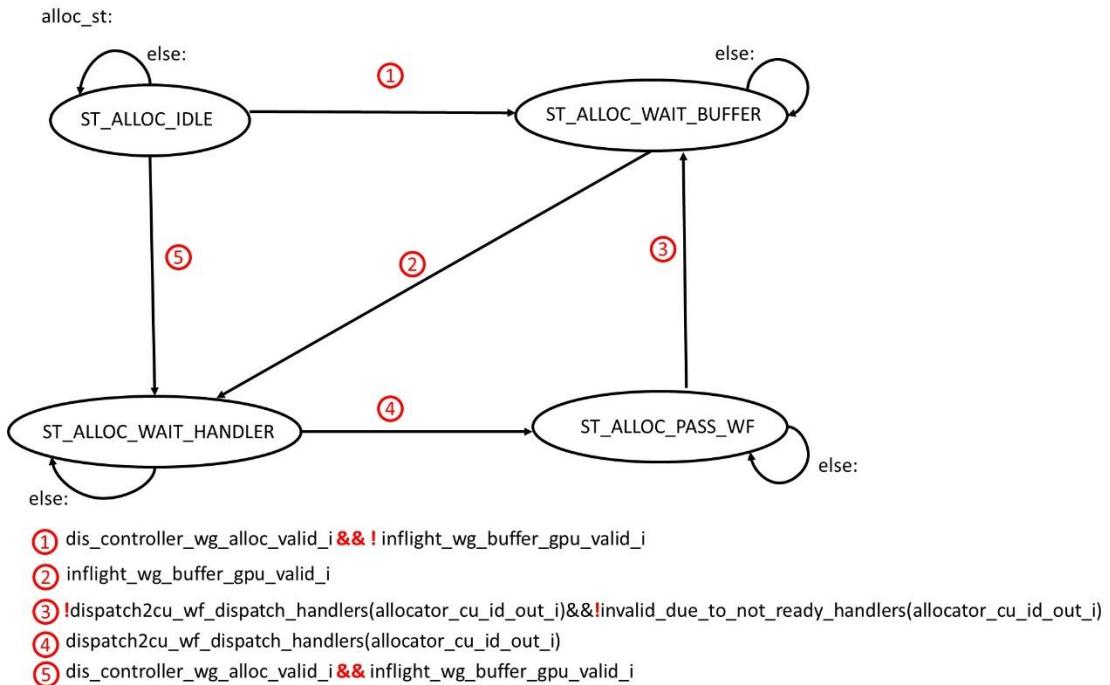


图 3-16 alloc_st 状态机跳转示意图

alloc_st 状态机说明：

(1) 默认为 idle (**ST_ALLOC_IDLE**) 状态。

在此状态下，先拉高寄存器 gpu_interface_alloc_available_i。再根据 dis_controller 的 wg_alloc 以及 inflight 的 wg_buffer 的有效状态，告知 cu_handler 可进入 alloc 状态、并向 cu_handler 分配 wg_id、wf_cnt，从此进入 **ST_ALLOC_WAIT_HANDLER** 状态；若 dis_controller 的 wg_alloc 的状态有效而 inflight 的 wg_buffer 处于无效，则进入 **ST_ALLOC_WAIT_BUFFER** 状态，

(2) **ST_ALLOC_WAIT_BUFFER** 状态。

处于此状态时，等待 inflight_wg_buffer 相应有效信号：inflight_wg_buffer_gpu_valid_i，从而如(1)所述，向 cu_handler 分配 wg_id、wf_cnt，再进入 **ST_ALLOC_WAIT_HANDLER** 状态。

(3) **ST_ALLOC_WAIT_HANDLER** 状态。

当已经将 wf、wg_id 信息派遣给 cu 之后，cu_handler 会将对应的 cu 的 dispatch2cu_wf_dispatch_handlers 信号拉高，处于此处的 **ST_ALLOC_WAIT_HANDLER** 状态会进一步将待派遣的 wf_tag、allocator_wf_count、wg_buffer 里的信息再发送给 cta_scheduler，并进入 **ST_ALLOC_PASS_WF** 状态。

(4) **ST_ALLOC_PASS_WF** 状态

若仍然处于派遣状态，则依据上一状态中的维持(3)中的 wf 和 wf_tag 的信息不变，派遣的 sgpr 和 vgpr 基地址需要额外加上从 wg_buffer 中获取的基地址；否则的话，派遣信

号优先级会根据 not_ready_handler 的状态, 将 dispatch2cu_wf_dispatch_i(含义是派遣的目标 cu)重新复位; 若此条 cu 的 not_ready_handler 返回值为低, 则继续准备 alloc、并回到 idle 状态。

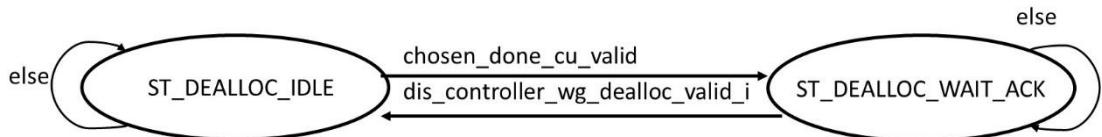


图 3-17 dealloc_st 状态机跳转示意图

dalloc_st 状态机说明:

dalloc 默认是处于 idle (ST DEALLOC_IDLE) 状态。

(1) ST DEALLOC_IDLE 状态。处于此状态下, 需要等待根据 cu_handler 返回来的 wg_done 的 dealloc 情况, 即是 chosen_done_cu_valid 信号, 进一步告知 inflight_wg_buffer 和 top_resource_table 的 dealloc 的 cu_id、对应 cu_id 的 wg_done 信号和 wg_id, 并进入 ST DEALLOC_WAIT_ACK 状态。

(2) ST DEALLOC_WAIT_ACK 状态。当 cu 不在 alloc 状态、且 cu 处于 dalloc 状态时, 需等待 dis_controller 发送一条拉高的 dis_controller_wg_dealloc_valid_i 信号, 让此状态回到 idle 状态; 若未应答到 dis_controller_wg_dealloc_valid_i, 则继续等待, 处于可 dalloc 状态。

cu_found 根据 cu_handler 返回来的 wg_done 的 dealloc 情况、选择出对应的 CU, 如下图所示。

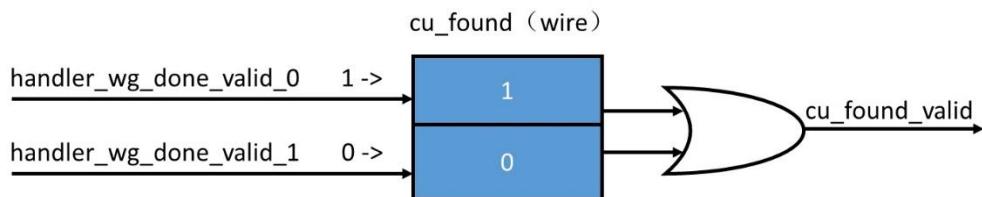


图 3-18 cu_found 硬件框图

3.3.5.3.1 CU_handler

cu_handler 此模块是详细标记和解标记 alloc 或者 dealloc 时的如 wg_id、wf_count 等信息, 用于及时的更新告知 inflight_wg_buffer 和 top_resource_table_i 哪些 wg_id、wf_tag 等被标记过而不能再利用、哪些 dealloc 过后能被再利用。

cu_handler 内部 wg_bitmap 的功能如下图所示, dealloc 的位置取决于 pending_wg_bitmap 的索引信息。

pending_wg_bitmap(regs)

逐个遍历->

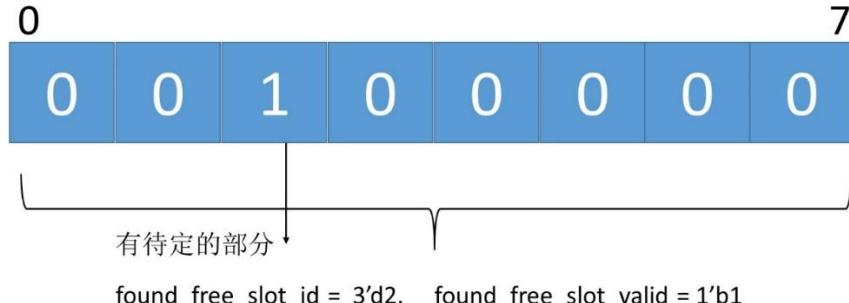


图 3-19 wg_bitmap 的索引方式

`dealloc` 的位置取决于 `pending_wg_bitmap` 的索引信息。`found_free_slot_id` 返回的是空闲位置的 id 索引。倘若 8 位索引中，均为低，`valid` 返回值为低。

used slot bitmap(regs)

逐个遍历->

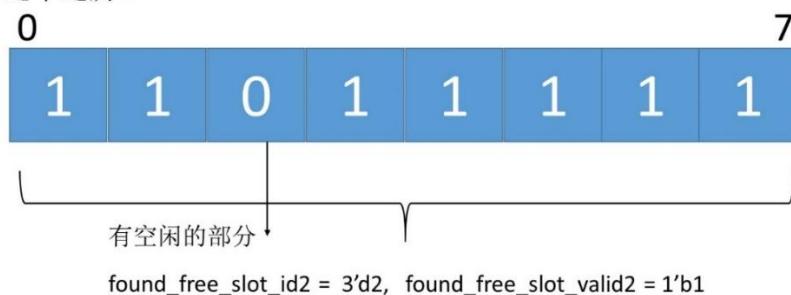


图 3-20 pending wg bitmap 的索引方式

`used_slot_bitmap` 会标记在工作状态对应的 `wg_id`, 硬件查找并返回的是下一个有 `slot` 位置的 `id`。倘若 8 位索引中, 均为高, `valid` 返回值为低。

在波形中可看到，测试输出的 found_free_slot_id2 值是第一个匹配!used_slot_bitmap 的数值。波形中，没有验切换 found_free_slot_id2 的情况。

`alloc` 状态机的状态图如下图所示。



图 3-21 alloc 状态机跳转示意图

(1) 初始化默认为 idle (**ST_ALLOC_IDLE**) 状态，在得到 workgroup 的分配使能且下一个资源位置有效时，进入 **STALLOCATING** 状态。当处于 idle 状态时，准许将 cu_handler 内部定义的 ram 记录分配的 wg_id 和 wf_cnt、在 used_slot_bitmap 中标记好被分配出去的 slot 位置。当前执行的 warp 的计数（猜测是 **wid**）、空闲位置也由 used_slot_bitmap 指定。

(2) 处于 **ST_ALLOCATING** 状态时, 根据当前分配的 warp 计数和 dispatch2cu 的准备情况, 检查 cu 不会操作两个不同的 warp 却具有相同的 tag、当前的分配的 warp 计数减一直到归零, 将接下来待分配的 wf_slot 和 wf_cnt-1 派遣给 cu、invalid_due_to_not_ready_i 拉低; 在未准备派遣的情况下, 发出未准备好的信号。在当前的分配的 warp 计数依然为 0 的情况下

下，使得状态回到 idle，并发出未准备好的信号。

dealloc 状态机的状态图如下图所示。

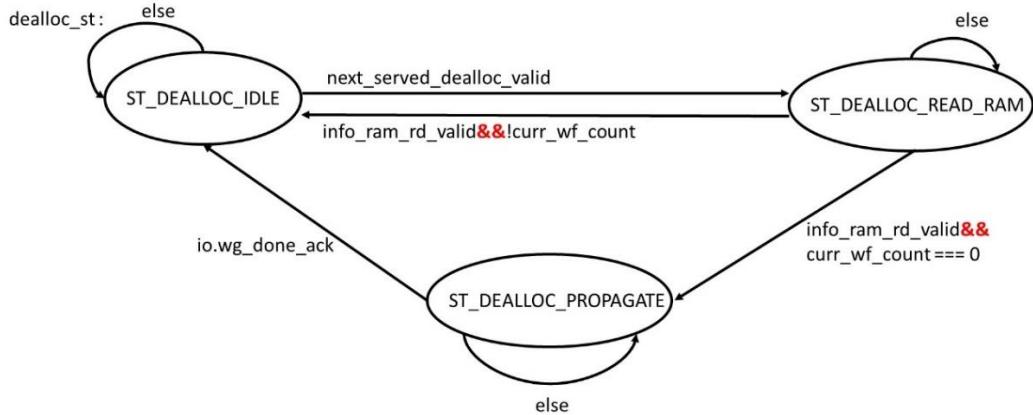


图 3-22 dealloc 状态机跳转示意图

(1) 初始化默认处于 idle (**ST DEALLOC_IDLE**) 状态，在 (`next_served_dealloc_valid`) 得到 `dealloc` 有效信号后，准许对自定义的 ram 进行读、从中获取 `wg_slot` 和 `wf_cnt` 作为当下要 `dealloc` 的对象，在 `pending_bitmap` 标记该部分(拉低)，并进入 **ST DEALLOC_READ_RAM** 状态。

(2) **ST DEALLOC_READ_RAM** 状态。根据当前的 `wf_cnt` 是否归零，判断对 wrokgroup 的释放是否完成、从 ram 中读出 alloc 时写入的 `wg_id` 并输出给 `gpu_interface`，进入 **ST DEALLOC_PROPAGATE** 状态、释放(拉低) `used_slot_bitmap` 的空闲 `wg_slot` 位置；`wf_cnt` 不是 0，则回到 idle 状态。

(3) **ST DEALLOC_PROPAGATE** 状态。根据返回的 wrokgroup 的完成应答状态，判断是否继续回到 idle 状态与否，否则的话，拉高并输出 wrokgroup 的完成有效信号。

4 Pipeline

4.1 概述

Ventus 的流水线包括六个流水级（译码之前还有取指），其结构框图如下图所示。

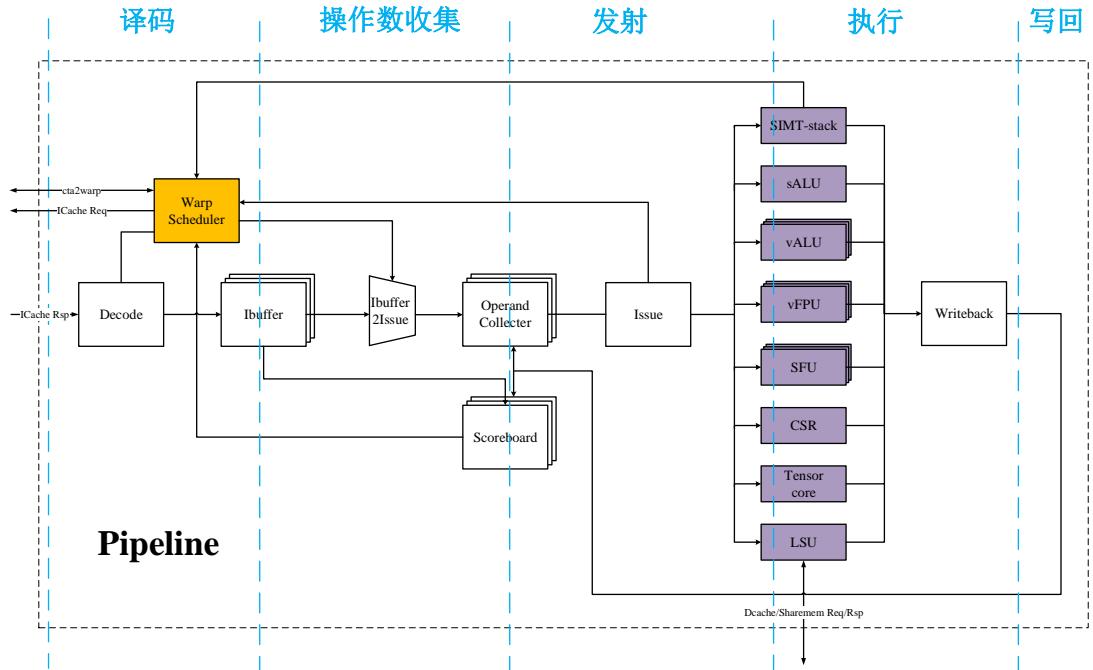


图 4-1 Pipeline 架构框图

4.2 详细设计

4.2.1 WarpScheduler

4.2.1.1 概述

warp 调度器主要功能如下：

- (1) 接收 CTA 调度器提供的 warp 信息，激活相应 warp 并标记所属的 block 信息。在 warp 执行完成后，将信息返回给 CTA 调度器以释放对应硬件。
- (2) 接收 issue 发送的 barrier 指令信息，将指定的 warp 锁住直到其所属的 block 中的所有活跃 warp 都执行到 barrier 指令。
- (3) 采用贪婪策略，选择发给 icache 进行取指的 warp，但当缓存相应 warp 指令的 ibuffer 已满时，会切换到下一个 warp 进行取指。发生 icache miss 或 ibuffer full 时会回退该 warp 的 pc，并等待下一次发射。
- (4) 选择未被 barrier 指令锁住、记分牌未显示冲突的活跃线程束，发给执行单元。

4.2.1.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟

<code>rst_n</code>	IN	1	全局复位，低有效， <code>rst_n</code> 触发后将在至少一个时钟周期后对核进行复位
From CTA			
<code>warpReq_valid_i</code>	IN	1	握手信号，为高电平时表示将分配 warp，需要接收信息。无需保持为高电平，直到 <code>ready</code> 信号为高电平
<code>warpReq_dispatch2cu_wf_tag_dispatch_i</code>	IN	TAG_WIDTH	workgroup id
<code>warpReq_wid_i</code>	IN	DEPTH_WARP	warp id
<code>warpReq_dispatch2cu_start_pc_dispatch_i</code>	IN	32	warp 起始 pc
to CTA			
<code>warpRsp_valid_o</code>	OUT	1	握手信号，为高电平时表示有 warp 执行完成。需要保持为高电平，直到 <code>ready</code> 信号为高电平
<code>warpRsp_ready_i</code>	IN	1	host 响应握手信号
<code>warpRsp_wid_o</code>	OUT	DEPTH_WARP	执行完成的 warp id
<code>wg_id_tag_i</code>	IN	TAG_WIDTH	执行 barrier 指令或执行完成的 warp 所属的 workgroup id
<code>wg_id_lookup_o</code>	OUT	DEPTH_WARP	执行 barrier 指令或执行完成的 warp id
To Icache			
<code>pc_req_valid_o</code>	OUT	1	握手信号，为高电平时表明 warp 调度器向 icache 发送取指请求。无需保持为高电平，直到 <code>ready</code> 信号为高电平
<code>pc_req_addr_o</code>	OUT	32	取指请求地址
<code>pc_req_mask_o</code>	OUT	NUM_FETCH	指令有效掩码
<code>pc_req_wid_o</code>	OUT	DEPTH_WARP	取指请求的 warp id
From Icache			
<code>pc_rsp_valid_i</code>	IN	1	握手信号，为高电平时表示 icache 发起取指响应。无需保持为高电平，直到 <code>ready</code> 信号为高电平。
<code>pc_rsp_addr_i</code>	IN	32	取指响应地址

pc_rsp_mask_i	IN	NUM_FETCH	指令有效掩码
pc_rsp_wid_i	IN	DEPTH_WARP	取指响应的 warp id
pc_rsp_status_i	IN	1	1 表示取指状态为 miss, 0 表示取指状态为 hit
From branch_back			
branch_valid_i	IN	1	握手信号, 为高电平时表示当前执行阶段发生了地址跳转。需要保持为高电平, 直到 ready 信号为高电平。
branch_ready_o	OUT	1	握手信号。
branch_wid_i	IN	DEPTH_WARP	执行分支跳转指令的 warp id
branch_jump_i	IN	1	跳转有效信号, 为高电平时表示需要跳转至新地址, 为低时不进行跳转
branch_new_pc_i	IN	32	需要跳转到的新地址
From Issue			
warp_control_valid_i	IN	1	握手信号, 为高电平时表示 warp 在执行 barrier 指令, warp 将被锁定或在执行 endprg 指令, warp 将执行完成。无需保持为高电平, 直到 ready 信号为高电平
warp_control_ready_o	OUT	1	握手信号
warp_control_simt_stack_op_i	IN	1	为 1 时表示 endprg 指令, warp 将执行完成; 为 0 时表示 barrier 指令, warp 将被锁定
warp_control_wid_i	IN	DEPTH_WARP	执行 barrier 指令或 endprg 指令的 warp id
From Scoreboard			
scoreboard_busy_i	IN	NUM_WARP	记分牌冲突信号
ibuffer_ready_i	IN	NUM_WARP	ibuffer 非满信号
warp_ready_o	OUT	NUM_WARP	未被 barrier 指令锁住、记分牌未显示冲突的活跃 warp
to ibuffer			

flush_valid_o	OUT	1	握手信号，为高电平时表示发生跳转或 warp 即将执行完成，需要清空译码单元和 ibuffer，无需保持为高电平，直到 ready 信号为高电平
flush_wid_o	OUT	DEPTH_WARP	发生跳转或即将执行完成的 warp id
flushCache_valid_o	OUT	1	握手信号，为高电平时表示 icache miss 或 ibuffer full，需要将后续取指响应无效化。无需保持为高电平，直到 ready 信号为高电平
flushCache_wid_o	OUT	DEPTH_WARP	发生 icache miss 或 ibuffer full 的 warp id

4.2.1.3 设计思路

接收 CTA scheduler 分配的 warp 信息，预设 CSR 寄存器值，并激活该 warp。在 warp 执行完成后，将该信息返回给 CTA scheduler（CTA scheduler 提供 warp id 以及其所属的 block id，还提供 warp 的起始 pc）。

接收流水线发送的 barrier 指令信息，将指定的 warp 暂停直到其所属的 block 中所有活跃 warp 都执行到 barrier 指令（barrier 指令需要显式插入，线程达到栅栏后需等待线程块（block/workgroup）内所有线程才可继续执行）。

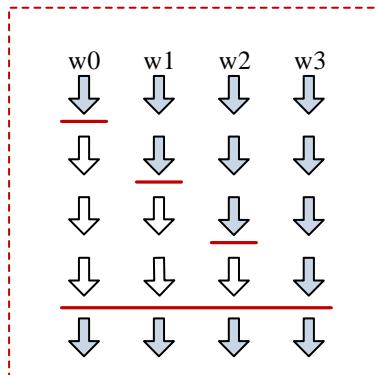


图 4-2 barrier 指令示意图

采用贪婪调度策略(Greedy-Then-Oldest,GTO)，选择发给 icache 的 warp。当发生 icache miss 或 ibuffer 已满的情况会将该 warp 的 pc 回退，同时 ibuffer 已满的情况会切换到下一个 warp。

采用轮询调度策略(Round-Robin,RR)，选择发给执行单元的 warp。选择出当前指令缓冲有效、记分牌未显示冲突的指令。切换 warp 仅需要一个周期。

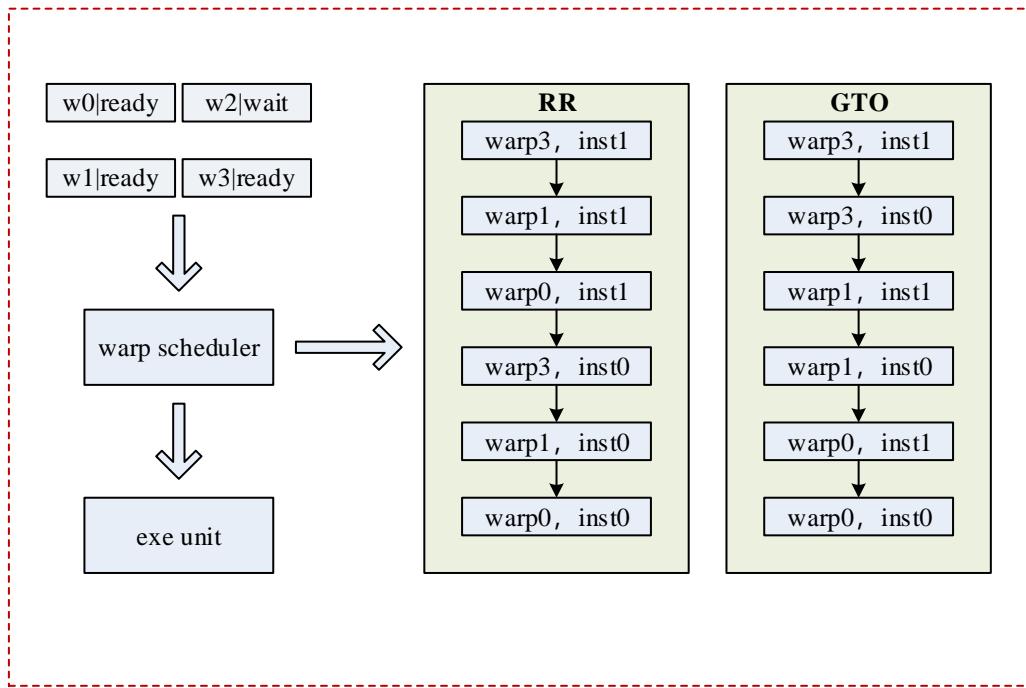


图 4-3 轮询策略(RR)、贪婪策略(GTO)示意图

内部子模块 pcControl 的设计框图图下图所示。

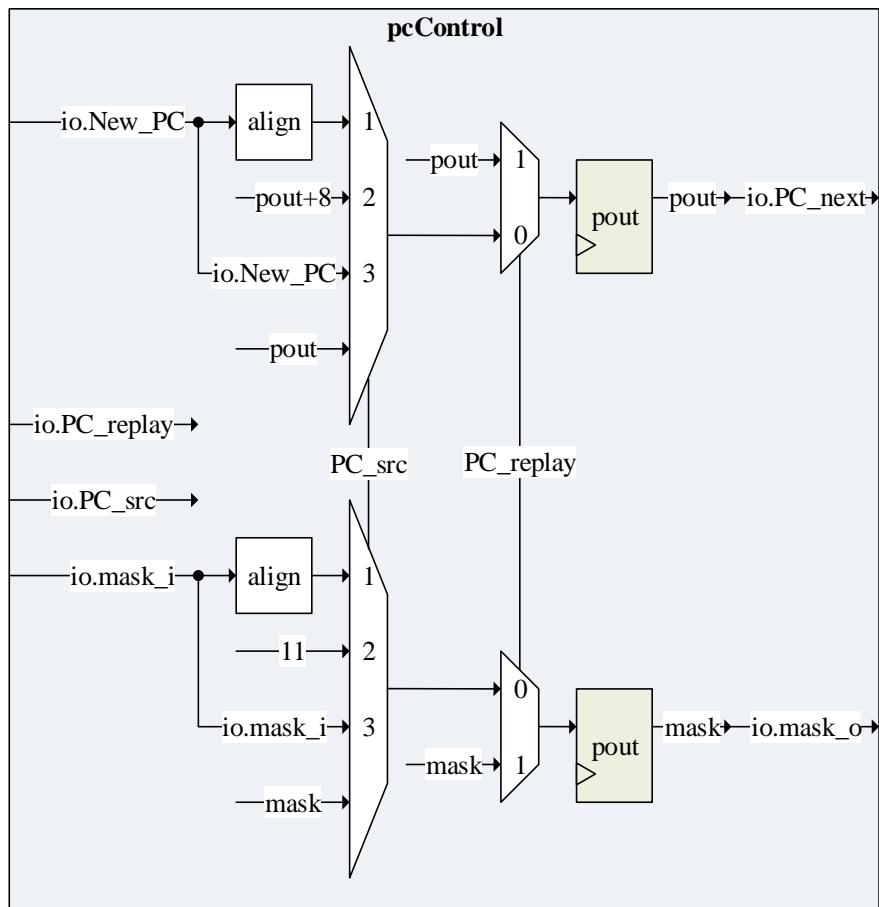


图 4-4 pcControl 的硬件框图

4.2.2 InstrDecode

4.2.2.1 概述

Decode 模块是将 32 位指令，按照不同种类的代码，进一步译码成控制（control signals）信号，属于 pipeline 的开始模块。每一组 control signals 还配备了自己 control_mask 信息。供给 ibuffer 模块调整发射的顺序。

4.2.2.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	时钟
RST_N	IN	1	下降沿复位
From Icache			
inst_0_i	IN	32	32 位指令
inst_1_i	IN	32	32 位指令
inst_mask_0_i	IN	1	32 位指令掩码
inst_mask_1_i	IN	1	32 位指令掩码
pc_i	IN	32	pc 地址
wid_i	IN	3	用于工作的 warp_id
From warp_scheduler			
flush_wid_i	IN	3	用于 flush 的 warp_id
flush_wid_valid_i	IN	1	flush 开始信号
From ibuffer			
ibuffer_ready_i	IN	`NUM_WARP	空满信号
To ibuffer			
control_mask_0_o	OUT	1	control 信号掩码信息
control_mask_1_o	OUT	1	control 信号掩码信息
control_Signals_inst_1_o	OUT	32	参见 Control 信号。
control_Signals_wid_1_o	OUT	3	
control_Signals_fp_1_o	OUT	1	
control_Signals_branch_1_o	OUT	2	
control_Signals_simt_stack_1_o	OUT	1	
control_Signals_simt_stack_op_1_o	OUT	1	
control_Signals_barrier_1_o	OUT	1	
control_Signals_csr_1_o	OUT	2	
control_Signals_reverse_1_o	OUT	1	
control_Signals_sel_alu2_1_o	OUT	2	
control_Signals_sel_alu1_1_o	OUT	2	
control_Signals_sel_alu3_1_o	OUT	2	

control_Signals_isvec_1_o	OUT	1
control_Signals_mask_1_o	OUT	1
control_Signals_sel_imm_1_o	OUT	4
control_Signals_mem_whb_1_o	OUT	2
control_Signals_mem_unsigned_1_o	OUT	1
control_Signals_alu_fn_1_o	OUT	6
control_Signals_force_rm_rtz_1_o	OUT	1
control_Signals_is_vls12_1_o	OUT	1
control_Signals_mem_1_o	OUT	1
control_Signals_mul_1_o	OUT	1
control_Signals_tc_1_o	OUT	1
control_Signals_disable_mask_1_o	OUT	1
control_Signals_custom_signal_0_1_o	OUT	1
control_Signals_mem_cmd_1_o	OUT	2
control_Signals_mop_1_o	OUT	2
control_Signals_reg_idx1_1_o	OUT	8
control_Signals_reg_idx2_1_o	OUT	8
control_Signals_reg_idx3_1_o	OUT	8
control_Signals_reg_idxw_1_o	OUT	8
control_Signals_wvd_1_o	OUT	1
control_Signals_fence_1_o	OUT	1
control_Signals_sfu_1_o	OUT	1
control_Signals_readmask_1_o	OUT	1
control_Signals_writemask_1_o	OUT	1
control_Signals_wxd_1_o	OUT	1
control_Signals_pc_1_o	OUT	32
control_Signals_imm_ext_1_o	OUT	6
control_Signals_atomic_1_o	OUT	1
control_Signals_aq_1_o	OUT	1
control_Signals_rl_1_o	OUT	1
control_Signals_inst_0_o	OUT	32
control_Signals_wid_0_o	OUT	3
control_Signals_fp_0_o	OUT	1
control_Signals_branch_0_o	OUT	2
control_Signals_simt_stack_0_o	OUT	1
control_Signals_simt_stack_op_0_o	OUT	1
control_Signals_barrier_0_o	OUT	1
control_Signals_csr_0_o	OUT	2
control_Signals_reverse_0_o	OUT	1
control_Signals_sel_alu2_0_o	OUT	2
control_Signals_sel_alu1_0_o	OUT	2
control_Signals_sel_alu3_0_o	OUT	2
control_Signals_isvec_0_o	OUT	1

control_Signals_mask_0_o	OUT	1	
control_Signals_sel_imm_0_o	OUT	4	
control_Signals_mem_whb_0_o	OUT	2	
control_Signals_mem_unsigned_0_o	OUT	1	
control_Signals_alu_fn_0_o	OUT	6	
control_Signals_force_rm_rtz_0_o	OUT	1	
control_Signals_is_vls12_0_o	OUT	1	
control_Signals_mem_0_o	OUT	1	
control_Signals_mul_0_o	OUT	1	
control_Signals_tc_0_o	OUT	1	
control_Signals_disable_mask_0_o	OUT	1	
control_Signals_custom_signal_0_0_o	OUT	1	
control_Signals_mem_cmd_0_o	OUT	2	
control_Signals_mop_0_o	OUT	2	
control_Signals_reg_idx1_0_o	OUT	8	
control_Signals_reg_idx2_0_o	OUT	8	
control_Signals_reg_idx3_0_o	OUT	8	
control_Signals_reg_idxw_0_o	OUT	8	
control_Signals_wvd_0_o	OUT	1	
control_Signals_fence_0_o	OUT	1	
control_Signals_sfu_0_o	OUT	1	
control_Signals_readmask_0_o	OUT	1	
control_Signals_writemask_0_o	OUT	1	
control_Signals_wxd_0_o	OUT	1	
control_Signals_pc_0_o	OUT	32	
control_Signals_imm_ext_0_o	OUT	6	
control_Signals_atomic_0_o	OUT	1	
control_Signals_aq_0_o	OUT	1	
control_Signals_rl_0_o	OUT	1	
rm_0_o	OUT	3	舍入模式
rm_is_static_0_o	OUT	1	是否是静态舍入模式
rm_1_o	OUT	3	舍入模式
rm_is_static_1_o	OUT	1	是否是静态舍入模式

4.2.3 InstrBuffer

4.2.3.1 概述

ibuffer 模块是将 decode 模块的 control signals 存储在内部的 fifo 中。fifo 的深度为 size_ibuffer，一次可将 num_fetch 个控制信息存储。

4.2.3.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
From Decode			
ibuffer_in_control_mask_i	IN	NUM_FETCH*1	控制信号掩码
ibuffer_in_control_Signals_inst_i	IN	NUM_FETCH*32	控制信号，含义 详情见 CtrlSigs 类信号列表
ibuffer_in_control_Signals_wid_i	IN	NUM_FETCH*3	
ibuffer_in_control_Signals_fp_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_branch_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_simt_stack_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_simt_stack_op_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_barrier_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_csr_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_reverse_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_sel_alu2_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_sel_alu1_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_sel_alu3_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_isvec_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_mask_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_sel_imm_i	IN	NUM_FETCH*4	
ibuffer_in_control_Signals_mem_whb_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_mem_unsigned_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_alu_fn_i	IN	NUM_FETCH*6	
ibuffer_in_control_Signals_force_rm_rtz_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_is_vls12_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_mem_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_mul_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_tc_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_disable_mask_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_custom_signal_0_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_mem_cmd_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_mop_i	IN	NUM_FETCH*2	
ibuffer_in_control_Signals_reg_idx1_i	IN	NUM_FETCH*8	
ibuffer_in_control_Signals_reg_idx2_i	IN	NUM_FETCH*8	
ibuffer_in_control_Signals_reg_idx3_i	IN	NUM_FETCH*8	
ibuffer_in_control_Signals_reg_idxw_i	IN	NUM_FETCH*8	
ibuffer_in_control_Signals_wvd_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_fence_i	IN	NUM_FETCH*1	

ibuffer_in_control_Signals_sfu_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_readmask_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_writemask_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_wxd_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_pc_i	IN	NUM_FETCH*32	
ibuffer_in_control_Signals_imm_ext_i	IN	NUM_FETCH*6	
ibuffer_in_control_Signals_atomic_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_aq_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_rl_i	IN	NUM_FETCH*1	
ibuffer_in_control_Signals_rm_i	IN	NUM_FETCH*3	舍入模式
ibuffer_in_control_Signals_rm_is_static_i	IN	NUM_FETCH*1	是否是静态舍入模式

4.2.4 ScoreBoard

4.2.4.1 概述

记分牌模块用于记录流水线中各种类型的冒险冲突，从而控制流水线的停顿，确保流水化的指令能够正常进行。在“承影”架构中，每个线程束拥有各自独立的记分牌模块，当某个线程束指令流水线因数据相关被停顿时，其他线程束的指令仍可以被调度执行。记分牌模块为线程束中的每个通用寄存器（vectorReg/scalarReg）、跳转操作（beqReg）、操作数收集（opcolReg）和 fence 操作（fenceReg）都分配 1bit 用于记录相应操作的完成状态。

目前记分牌模块解决的冲突类型包括以下八种类型：

(1) 待发射指令的源 1 寄存器与正在执行指令的目的寄存器相同，流水线需要停顿等待该目的寄存器写回(read_rs1)。

(2) 待发射指令的源 2 寄存器与正在执行指令的目的寄存器相同，流水线需要停顿等待该目的寄存器写回(read_rs2)。

(3) 待发射指令的源 3 寄存器与正在执行指令的目的寄存器相同，流水线需要停顿等待该目的寄存器写回(read_rs3)。

(4) 待发射指令是带掩码的向量指令，流水线需要停顿等待向量掩码寄存器写回(read_mask)。

(5) 待发射指令的目的寄存器与正在执行指令的目的寄存器相同，流水线需要停顿等待该目的寄存器写回(read_wb)。

(6) 待发射指令是跳转指令，流水线需要停顿并清空(read_beq)。

(7) 当前指令正在操作数收集器中收集，流水线需要停顿等待操作数收集完成并发射(read_opcol)。

(7) 待发射指令是访存指令，流水线需要停顿等待 fence 操作完成(read_fence)。

4.2.4.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟
rst_n	IN	1	全局复位，低有效
ibuffer 端口			
ibuffer_if_sel_alu1_i	IN	2	源寄存器 1 选择信号
ibuffer_if_sel_alu2_i	IN	2	源寄存器 2 选择信号
ibuffer_if_sel_alu3_i	IN	2	源寄存器 3 选择信号
ibuffer_if_reg_idx1_i	IN	8	位数扩展后的源寄存器 1
ibuffer_if_reg_idx2_i	IN	8	位数扩展后的源寄存器 2
ibuffer_if_reg_idx3_i	IN	8	位数扩展后的源寄存器 3
ibuffer_if_reg_idxw_i	IN	8	位数扩展后的目的寄存器
ibuffer_if_isvec_i	IN	1	进行 vALU 或 SIMD 计算
ibuffer_if_readmask_i	IN	1	当指令为 VL table 中后 16 条时为 1, 其余指令为 0
ibuffer_if_branch_i	IN	2	01 为跳转指令, 10 为 jal 指令, 11 为 jalr 指令
ibuffer_if_mask_i	IN	1	为 1 时 mask 使能, 为 0 时 mask 不使能
ibuffer_if_wxd_i	IN	1	写回的目的寄存器是标量寄存器
ibuffer_if_wvd_i	IN	1	写回的目的寄存器是向量寄存器
ibuffer_if_mem_i	IN	1	是否 LSU 指令和原子指令（这两个都需要存储器读写）
ibuffer2issue 端口			
if_reg_idxw_i	IN	8	位数扩展后的目的寄存器
if_wxd_i	IN	1	写回的目的寄存器是标量寄存器
if_wvd_i	IN	1	写回的目的寄存器是向量寄存器
if_branch_i	IN	2	01 为跳转指令, 10 为 jal 指令, 11 为 jalr 指令
if_barrier_i	IN	1	仅限于 barrier、barriersub 和 ednprg 指令时拉高（具体指令功能见文档）
if_fence_i	IN	1	是否 fence 指令
if_fire_i	IN	1	端口握手信号
writeback 端口			
wb_v_reg_idxw_i	IN	8	位数扩展后的向量目的寄存器
wb_v_wvd_i	IN	1	写回的目的寄存器是向量寄存器
wb_v_fire_i	IN	1	向量寄存器写回握手信号
wb_x_reg_idxw_i	IN	8	位数扩展后的标量目的寄存器
wb_x_wxd_i	IN	1	写回的目的寄存器是标量寄存器
wb_x_fire_i	IN	1	标量寄存器写回握手信号
lsu 端口			
fence_end_i	IN	1	fence 请求完成

operand_collector 端口			
op_col_in_fire_i	IN	1	操作数开始收集握手信号
op_col_out_fire_i	IN	1	操作数完成收集握手信号
warp_sche 端口			
br_ctrl_i	IN	1	跳转指令完成信号（三种情况）
delay_o	OUT	1	流水线停顿信号

4.2.5 OperandCollector

4.2.5.1 概述

OperandCollector 的主要功能是在寄存器文件中收集操作数，并将其发送给执行单元，其硬件框图如下图所示。

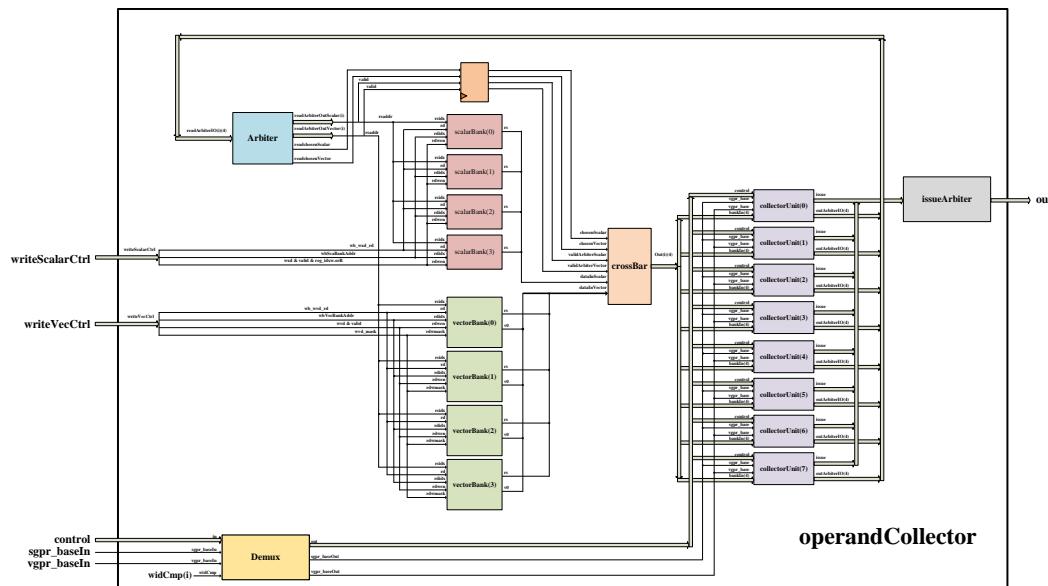


图 4-5 OperandCollector 的硬件框图

Demux 例化于 instDemux 模块，连接外部输入端口至收集单元，用于接收外部指令的译码信息和寄存器的基址信息，接着将此信息同步至每个操作数收集单元，并选择可用的收集单元进行指令的操作数收集过程。

collectorUnits 例化于 collectorUnit 模块，连接交叉开关至仲裁发射端口，用于接收交叉开关读取的操作数，待该指令的操作数收集完成后输出至仲裁发射端。根据 num_collectorUnit 参数例化 8 个。

Arbiter 例化于 operandArbiter 模块，连接收集单元至寄存器板块和交叉开关，对发送到寄存器板块的读请求进行仲裁控制。

vectorBank 例化于 FloatRegFileBank 模块，是连接仲裁器至交叉开关的向量寄存器板块，用于存储向量操作数的寄存器文件，根据 num_bank 参数例化 4 个。

scalarBank 例化于 RegFileBank 模块，是连接仲裁器至交叉开关的标量寄存器板块，用于存储标量操作数的寄存器文件，根据 num_bank 参数例化 4 个。

crossBar 例化于 crossBar 模块，连接寄存器板块至收集单元，控制将读取到的结果送至收集单元。

`issueArbiter` 是指令发射仲裁器，连接收集单元至外部输出端，负责将完成操作数收集的指令仲裁后发射出去执行。

4.2.5.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
<code>clk</code>	IN	1	系统时钟
<code>rst_n</code>	IN	1	全局复位，低有效
<code>ibuffer2Issue</code> 端口			
<code>in_valid_i</code>	IN	1	端口有效信号
<code>in_ready_o</code>	OUT	1	端口就绪信号
<code>in_wid_i</code>	IN	DEPTH_WARP	线程束 ID
<code>in_inst_i</code>	IN	32	指令
<code>in_imm_ext_i</code>	IN	6	立即数扩展后的高 6bit
<code>in_sel_imm_i</code>	IN	4	立即数类型选择信号
<code>in_reg_idx1_i</code>	IN	8	位数扩展后的源寄存器 1
<code>in_reg_idx2_i</code>	IN	8	位数扩展后的源寄存器 2
<code>in_reg_idx3_i</code>	IN	8	位数扩展后的源寄存器 3
<code>in_branch_i</code>	IN	2	01 为跳转指令, 10 为 <code>jal</code> 指令, 11 为 <code>jalr</code> 指令
<code>in_custom_signal_0_i</code>	IN	1	只有 <code>setrpc</code> 指令时为 1, 将分支汇合地址写入 <code>rpc</code> 的 <code>csr</code> 寄存器
<code>in_isvec_i</code>	IN	1	进行 vALU 或 SIMT 计算
<code>in_readmask_i</code>	IN	1	当指令为 VL table 中后 16 条时为 1, 其余指令为 0
<code>in_sel_alu1_i</code>	IN	2	源寄存器 1 选择信号
<code>in_sel_alu2_i</code>	IN	2	源寄存器 2 选择信号
<code>in_sel_alu3_i</code>	IN	2	源寄存器 3 选择信号
<code>in_pc_i</code>	IN	32	程序指针
<code>in_mask_i</code>	IN	1	为 1 时 mask 使能, 为 0 时 mask 不使能
<code>in_fp_i</code>	IN	1	表示是否是浮点数操作
<code>in_simt_stack_i</code>	IN	1	为 1 时需要使用 <code>simt_stack</code> 堆栈, 为 0 时不需要
<code>in_simt_stack_op_i</code>	IN	1	区分 <code>join</code> (为 1) 与其他分支指令(<code>endprg</code> 指令此信号也为 1)
<code>in_barrier_i</code>	IN	1	仅限于 <code>barrier</code> 、 <code>barriersub</code> 和 <code>ednprg</code> 指令时拉高(具体指令功能见文档)
<code>in_csr_i</code>	IN	2	控制 <code>csr</code> 工作状态, 00 为非 <code>csr</code> 指令, 01 为 W(对应 CSRRW),

			10 为 S (对应 CSRRS), 11 为 C (对应为 CSRRC)
in_reverse_i	IN	1	操作数调换, 将 in2 赋给 a, 将 in1 赋给 b
in_mem_whb_i	IN	2	表示 mem 的操作类型 (字、半字和字节操作)
in_mem_unsigned_i	IN	1	加载或存储的数据是否是无符号数
in_alu_fn_i	IN	6	指令操作码
in_force_rm_rtz_i	IN	1	
in_is_vls12_i	IN	1	是否长立即数偏移的向量访存指令
in_mem_i	IN	1	是否 LSU 指令和原子指令 (这两个都需要存储器读写)
in_mul_i	IN	1	是否乘法指令
in_tc_i	IN	1	是否张量运算指令
in_disable_mask_i	IN	1	区分是否为自定义访存指令
in_mem_cmd_i	IN	2	为 2 时写存储器, 为 1 时读存储器, 为 0 时不是存储器操作
in_mop_i	IN	2	指示访存地址模式 (详见 V 扩展文档 P30)
in_reg_idxw_i	IN	8	位数扩展后的目的寄存器
in_wvd_i	IN	1	写回的目的寄存器是向量寄存器
in_fence_i	IN	1	是否 fence 指令
in_sfу_i	IN	1	是否使用 sfу 的指令 (除法、开根、取余)
in_wxd_i	IN	1	写回的目的寄存器是标量寄存器
in_atomic_i	IN	1	是否原子操作
in_aq_i	IN	1	acquire, 在这条指令之后的访存指令必须等到该指令完成后才开始执行
in_rl_i	IN	1	release, 在这条指令之前的访存指令的结果必须在该指令执行之前被观察到
in_rm_i	IN	2	
in_rm_is_atatic_i	IN	1	
writeback 标量端口			
writeScalar_valid_i	IN	1	端口有效信号
writeScalar_ready_o	OUT	1	端口就绪信号
writeScalar_rd_i	IN	32	写回标量目的寄存器数据

writeScalar_wxd_i	IN	1	写回标量信号
writeScalar_idxw_i	IN	8	写回标量目的寄存器地址
writeScalar_wid_i	IN	DEPTH_WARP	线程束 ID
writeback 向量端口			
writeVector_valid_i	IN	1	端口有效信号
writeVector_ready_o	OUT	1	端口就绪信号
writeVector_rd_i	IN	32* NUM_THREAD	写回向量目的寄存器数据
writeVector_wvd_i	IN	1	写回向量信号
writeVector_idxw_i	IN	8	写回向量目的寄存器地址
writeVector_wid_i	IN	DEPTH_WARP	线程束 ID
writeVector_wvd_mask_i	IN	NUM_THREAD	向量寄存器掩码
csrfile 端口			
sgpr_base_i	IN	11*NUM_WARP	标量寄存器基址
vgpr_base_i	IN	11*NUM_WARP	向量寄存器基址
issue 端口			
out_valid_o	OUT	1	端口有效信号
out_ready_i	IN	1	端口就绪信号
out_wid_o	OUT	DEPTH_WARP	线程束 ID
out_inst_o	OUT	32	指令
out_branch_o	OUT	2	01 为跳转指令, 10 为 jal 指令, 11 为 jalr 指令
out_custom_signal_0_o	OUT	1	只有 setrpc 指令时为 1, 将分支汇合地址写入 rpc 的 csr 寄存器
out_isvec_o	OUT	1	进行 vALU 或 SIMD 计算
out_pc_o	OUT	32	程序指针
out_fp_o	OUT	1	表示是否是浮点数操作
out_simt_stack_o	OUT	1	为 1 时需要使用 simt_stack 堆栈, 为 0 时不需要
out_simt_stack_op_o	OUT	1	区分 join(为 1) 与其他分支指令(endprg 指令此信号也为 1)
out_barrier_o	OUT	1	仅限于 barrier、barriersub 和 ednprg 指令时拉高(具体指令功能见文档)
out_csr_o	OUT	2	控制 csr 工作状态, 00 为非 csr 指令, 01 为 W(对应 CSRRW), 10 为 S(对应 CSRRS), 11 为 C(对应为 CSRRC)
out_reverse_o	OUT	1	操作数调换, 将 in2 赋给 a, 将 in1 赋给 b
out_mem_whb_o	OUT	2	表示 mem 的操作类型(字、半字和字节操作)

out_mem_unsigned_o	OUT	1	加载或存储的数据是否是无符号数
out_alu_fn_o	OUT	6	指令操作码
out_force_rm_rtz_o	OUT	1	
out_is_vls12_o	OUT	1	是否长立即数偏移的向量访存指令
out_mem_o	OUT	1	是否 LSU 指令和原子指令（这两个都需要存储器读写）
out_mul_o	OUT	1	是否乘法指令
out_tc_o	OUT	1	是否张量运算指令
out_disable_mask_o	OUT	1	区分是否为自定义访存指令
out_mem_cmd_o	OUT	2	为 2 时写存储器，为 1 时读存储器，为 0 时不是存储器操作
out_mop_o	OUT	2	指示访存地址模式（详见 V 扩展文档 P30）
out_reg_idxw_o	OUT	8	位数扩展后的目的寄存器
out_wvd_o	OUT	1	写回的目的寄存器是向量寄存器
out_fence_o	OUT	1	是否 fence 指令
out_sfu_o	OUT	1	是否使用 sfu 的指令（除法、开根、取余）
out_wxd_o	OUT	1	写回的目的寄存器是标量寄存器
out_atomic_o	OUT	1	是否原子操作
out_aq_o	OUT	1	acquire，在这条指令之后的访存指令必须等到该指令完成后才开始执行
out_rl_o	OUT	1	release，在这条指令之前的访存指令的结果必须在该指令执行之前被观察到
out_rm_o	OUT	2	
out_rm_is_atatic_o	OUT	1	

4.2.5.3 设计思路

操作数收集器通过允许尽可能多的指令同时访问操作数，利用多板块的并行性提高访问效率，其内部包含多个收集单元，每个收集单元包含一条线程束指令所需要的所有源操作数缓冲空间，如下图所示。

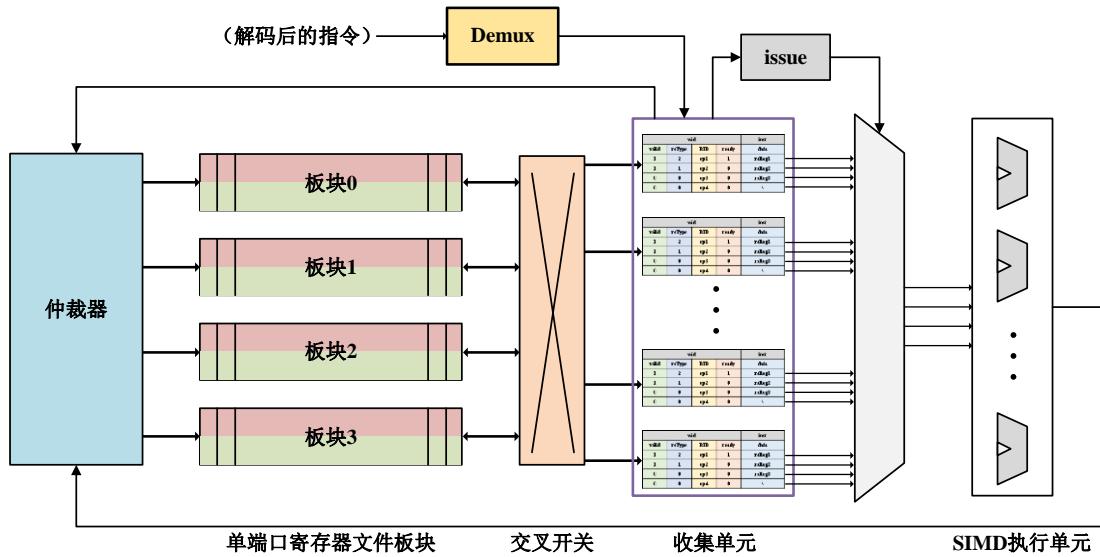


图 4-6 操作数收集器的结构框图

“承影”架构中，操作数收集器会接收来自指令缓冲中的请求，依据所需数据类型，经由 crossbar 向寄存器堆访问获取数据，获取后即进入待发射状态。

每个收集单元设置 4 个条目，除了源 1、源 2、源 3 操作数，还包括第 4 个用于保存向量掩码的操作数。每个条目包括有效位、就绪位、寄存器地址、板块 ID、操作数类型、操作数数据字段。收集单元的结构如下表所示。

wid				inst
valid	rsType	RID	ready	data
1	2	op1	1	rsReg1
1	1	op2	0	rsReg2
0	0	op3	0	rsReg3
0	0	op4	0	\

4.2.6 Issue

4.2.6.1 概述

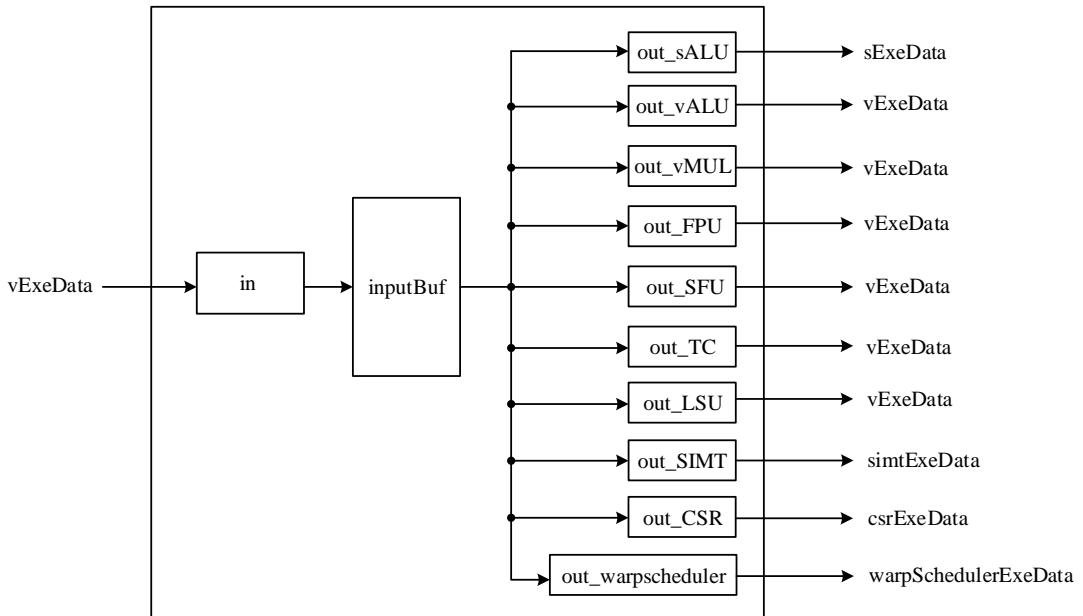


图 4-7 Issue 的架构框图

框图说明：

- (1) issue 内部主要是连线，通过 ctrl 信号来判断输出（out_）的 valid 使能，起到一个类似于数据分配的作用。
- (2) inputBuf 在代码中是一个深度为 0 的队列，所以，issue 内部仅包括组合逻辑。

4.2.6.2 信号描述

名称	类型	位宽	描述
From Exe_data			
issue_in_warps_control_Signals_inst_i	IN	32	控制信号，含义详情见 CtrlSigs 类信号列表
issue_in_warps_control_Signals_wid_i	IN	3	
issue_in_warps_control_Signals_fp_i	IN	1	
issue_in_warps_control_Signals_branch_i	IN	2	
issue_in_warps_control_Signals_simt_stack_i	IN	1	
issue_in_warps_control_Signals_simt_stack_op_i	IN	1	
issue_in_warps_control_Signals_barrier_i	IN	1	
issue_in_warps_control_Signals_csr_i	IN	2	
issue_in_warps_control_Signals_reverse_i	IN	1	
issue_in_warps_control_Signals_sel_alu2_i	IN	2	
issue_in_warps_control_Signals_sel_alu1_i	IN	2	
issue_in_warps_control_Signals_sel_alu3_i	IN	2	
issue_in_warps_control_Signals_isvec_i	IN	1	

issue_in_warps_control_Signals_mask_i	IN	1	
issue_in_warps_control_Signals_sel_imm_i	IN	4	
issue_in_warps_control_Signals_mem_whb_i	IN	2	
issue_in_warps_control_Signals_mem_unsigned_i	IN	1	
issue_in_warps_control_Signals_alu_fn_i	IN	6	
issue_in_warps_control_Signals_force_rm_rtz_i	IN	1	
issue_in_warps_control_Signals_is_vls12_i	IN	1	
issue_in_warps_control_Signals_mem_i	IN	1	
issue_in_warps_control_Signals_mul_i	IN	1	
issue_in_warps_control_Signals_tc_i	IN	1	
issue_in_warps_control_Signals_disable_mask_i	IN	1	
issue_in_warps_control_Signals_custom_signal_0_i	IN	1	
issue_in_warps_control_Signals_mem_cmd_i	IN	2	
issue_in_warps_control_Signals_mop_i	IN	2	
issue_in_warps_control_Signals_reg_idx1_i	IN	8	
issue_in_warps_control_Signals_reg_idx2_i	IN	8	
issue_in_warps_control_Signals_reg_idx3_i	IN	8	
issue_in_warps_control_Signals_reg_idxw_i	IN	8	
issue_in_warps_control_Signals_wvd_i	IN	1	
issue_in_warps_control_Signals_fence_i	IN	1	
issue_in_warps_control_Signals_sfu_i	IN	1	
issue_in_warps_control_Signals_readmask_i	IN	1	
issue_in_warps_control_Signals_writemask_i	IN	1	
issue_in_warps_control_Signals_wxd_i	IN	1	
issue_in_warps_control_Signals_pc_i	IN	32	
issue_in_warps_control_Signals_imm_ext_i	IN	6	
issue_in_warps_control_Signals_atomic_i	IN	1	
issue_in_warps_control_Signals_aq_i	IN	1	
issue_in_warps_control_Signals_rl_i	IN	1	
issue_in_warps_control_Signals_rm_i	IN	3	舍入模式类型
issue_in_warps_control_Signals_is_static_i	IN	1	是否为静态舍入
From OPC			
issue_in_valid_i	IN	1	握手信号
issue_in_vExeData_in1_i	IN	NUM_THREAD *32	操作数 1
issue_in_vExeData_in2_i	IN	NUM_THREAD *32	操作数 2
issue_in_vExeData_in3_i	IN	NUM_THREAD *32	操作数 3
issue_in_vExeData_mask_i	IN	NUM_THREAD *1	线程掩码
To OPC			

Issue_in_ready_o	OUT	1	握手信号
To sALU			
issue_out_sALU_valid_o	OUT	1	握手信号
issue_out_sALU_sExeData_in1_o	OUT	32	操作数 1
issue_out_sALU_sExeData_in2_o	OUT	32	操作数 2
issue_out_sALU_sExeData_in3_o	OUT	32	操作数 3
issue_out_sALU_warps_control_Signal_s_wid_o	OUT	3	操作的 warp_id
issue_out_sALU_warps_control_Signal_s_branch_o	OUT	2	跳转指示接口
issue_out_sALU_warps_control_Signal_s_alu_fn_o	OUT	6	计算操作码
issue_out_sALU_warps_control_Signal_s_reg_idxw_o	OUT	8	目标寄存器
issue_out_sALU_warps_control_Signal_s_wxd_o	OUT	1	标量寄存器写回标志
From sALU			
issue_out_sALU_ready_i	IN	1	握手信号
To vALU			
issue_out_vALU_valid_o	OUT	1	握手信号
issue_out_vALU_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1
issue_out_vALU_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_vALU_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_vALU_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_vALU_warps_control_Signal_s_wid_o	OUT	3	操作的线程束 id
issue_out_vALU_warps_control_Signal_s_reverse_o	OUT	2	跳转指示接口
issue_out_vALU_warps_control_Signal_s_alu_fn_o	OUT	6	计算操作码
issue_out_vALU_warps_control_Signal_s_reg_idxw_o	OUT	8	目标寄存器
issue_out_vALU_warps_control_Signal_s_wvd_o	OUT	1	向量寄存器写回标志
From vALU			
issue_out_vALU_ready_i	IN	1	握手信号
To vFPU			
issue_out_vFPU_valid_o	OUT	1	握手信号
issue_out_vFPU_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1

issue_out_vFPU_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_vFPU_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_vFPU_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_vFPU_warps_control_Signal_s_wid_o	OUT	3	操作的 warp_id
issue_out_vFPU_warps_control_Signal_s_reverse_o	OUT	2	操作数调转
issue_out_vFPU_warps_control_Signal_s_force_rm_rtz_o	OUT	1	指令类型是否为 RTZ
issue_out_vFPU_warps_control_Signal_s_alu_fn_o	OUT	6	计算模式操作码
issue_out_vFPU_warps_control_Signal_s_reg_idxw_o	OUT	8	目的寄存器
issue_out_vFPU_warps_control_Signal_s_wvd_o	OUT	1	向量寄存器写回标志
issue_out_vFPU_warps_control_Signal_s_wxd_o	OUT	1	标量寄存器写回标志
issue_out_vFPU_warps_control_Signal_s_rm_o	OUT	3	舍入模式
issue_out_vFPU_warps_control_Signal_s_rm_is_static_o	OUT	1	是否是静态舍入
From vFPU			
issue_out_vFPU_ready_i	IN	1	握手信号
To LSU			
issue_out_LSU_valid_o	OUT	1	握手信号
issue_out_LSU_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1
issue_out_LSU_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_LSU_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_LSU_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_LSU_warps_control_Signals_wid_o	OUT	3	操作的 warp_id
issue_out_LSU_warps_control_Signals_isvec_o	OUT	1	向量操作类型
issue_out_LSU_warps_control_Signals_mem_whb_o	OUT	2	mem 的操作长短
issue_out_LSU_warps_control_Signals_mem_unsigned_o	OUT	1	有无符号数
issue_out_LSU_warps_control_Signals	OUT	6	计算操作码

<u>_alu_fn_o</u>			
issue_out_LSU_warps_control_Signals_is_vls12_o	OUT	1	是否为长立即数访问的向量指令
issue_out_LSU_warps_control_Signals_disable_mask_o	OUT	1	区分为自定义访存指令
issue_out_LSU_warps_control_Signals_mem_cmd_o	OUT	2	对存储器的操作类型
issue_out_LSU_warps_control_Signals_mop_o	OUT	1	指示访存地址模式
issue_out_LSU_warps_control_Signals_reg_idxw_o	OUT	8	目的寄存器
issue_out_LSU_warps_control_Signals_wvd_o	OUT	1	向量寄存器写回
issue_out_LSU_warps_control_Signals_atomic_o	OUT	1	是否原子指令
issue_out_LSU_warps_control_Signals_aq_o	OUT	1	原子指令的获取权限
issue_out_LSU_warps_control_Signals_rl_o	OUT	1	原子指令的放下权限
To LSU			
issue_out_LSU_ready_i	IN	1	握手信号
To SFU			
issue_out_SFU_valid_o	OUT	1	握手信号
issue_out_SFU_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1
issue_out_SFU_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_SFU_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_SFU_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_SFU_warps_control_Signals_wid_o	OUT	3	操作的 warp_id
issue_out_SFU_warps_control_Signals_fp_o	OUT	1	是否是浮点数操作
issue_out_SFU_warps_control_Signals_reverse_o	OUT	2	操作数调转
issue_out_SFU_warps_control_Signals_is_vec_o	OUT	1	指令类型是否为 RTZ
issue_out_SFU_warps_control_Signals_alu_fn_o	OUT	6	计算模式操作码
issue_out_SFU_warps_control_Signals_reg_idxw_o	OUT	8	目的寄存器
issue_out_SFU_warps_control_Signals	OUT	1	向量寄存器写回标志

<u>_wvd_o</u>			
issue_out_SFU_warps_control_Signals	OUT	1	标量寄存器写回标志
From SFU			
issue_out_SFU_ready_i	IN	1	握手信号
To warp_scheduler			
issue_out_warps_valid_o	OUT	1	握手信号
issue_out_warps_control_Signals_wid_o	OUT	3	执行的线程束 id
issue_out_warps_control_Signals_simt_stack_op_o	OUT	1	simt_stack 执行类型
From warp_scheduler			
issue_out_warps_ready_i	IN	1	握手信号
To CSR			
issue_out_CSR_valid_o	OUT	1	握手信号
issue_out_CSR_csrExeData_in1_o	OUT	32	操作数
issue_out_CSR_warps_control_Signals_inst_o	OUT	32	指令
issue_out_CSR_warps_control_Signals_wid_o	OUT	3	执行的线程束 id
issue_out_CSR_warps_control_Signals_csr_o	OUT	2	CSR 操作类型
issue_out_CSR_warps_control_Signals_isvec_o	OUT		是否是向量操作
issue_out_CSR_warps_control_Signals_custom_signal_0_o	OUT		分支会合方式
issue_out_CSR_warps_control_Signals_reg_idxw_o	OUT		目的寄存器
issue_out_CSR_warps_control_Signals_wxd_o	OUT		标量寄存器写回
From CSR			
issue_out_CSR_ready_i	IN	1	握手信号
To MUL			
issue_out_MUL_valid_o	OUT	1	握手信号
issue_out_MUL_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1
issue_out_MUL_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_MUL_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_MUL_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_MUL_warps_control_Signals_wid_o	OUT	3	操作的 warp_id

issue_out_MUL_warps_control_Signal_s_reverse_o	OUT	2	操作数调转
issue_out_MUL_warps_control_Signal_s_alu_fn_o	OUT	6	计算模式操作码
issue_out_MUL_warps_control_Signal_s_reg_idxw_o	OUT	8	目的寄存器
issue_out_MUL_warps_control_Signal_s_wvd_o	OUT	1	向量寄存器写回标志
issue_out_MUL_warps_control_Signal_s_wxd_o	OUT	1	标量寄存器写回标志
From MUL			
issue_out_MUL_ready_i	IN	1	握手信号
To TC			
issue_out_TC_valid_o	OUT	1	握手信号
issue_out_TC_vExeData_in1_o	OUT	NUM_THREAD *32	操作数 1
issue_out_TC_vExeData_in2_o	OUT	NUM_THREAD *32	操作数 2
issue_out_TC_vExeData_in3_o	OUT	NUM_THREAD *32	操作数 3
issue_out_TC_vExeData_mask_o	OUT	NUM_THREAD	线程掩码
issue_out_TC_warps_control_Signals_wid_o	OUT	3	执行的线程束 id
issue_out_TC_warps_control_Signals_reg_idxw_o	OUT	8	目的寄存器
From TC			
issue_out_TC_ready_i	IN	1	握手信号
To simtExeData			
issue_simtExeData_valid_o	OUT	1	握手信号
issue_simtExeData_opcode_o	OUT	3	操作码
issue_simtExeData_wid_o	OUT	3	操作的线程束 id
issue_simtExeData_PC_branch_o	OUT	32	分支跳转 pc
issue_simtExeData_PC_execute_o	OUT	32	分支执行 pc
issue_simtExeData_mask_init_o	OUT	1	初始化
From simtExeData			
issue_simtExeData_ready_i	IN	1	握手信号

4.2.7 SimtStack

4.2.7.1 概述

simt stack 是一个特殊的执行单元，负责管理线程分支。

执行分支管理相关的向量指令时，`simt stack` 会接收相关指令的执行信息，并读取 `csr` 中的 `rpc`。得到指令执行结果时，`simt stack` 首先判断是否有分支分歧发生，如果没有分支分歧发生时，不会产生压栈行为，跳过不必要的程序段的执行。如果有分支分歧发生，`simt stack` 会进行判断，优先执行线程数量少的分歧路径，并将另一条分歧路径的 `npc`、`rpc` 压入栈中。

当执行 `join` 指令时，`simt stack` 接收到执行信息后，会进行弹栈，将栈中保存的另一条分歧路径的 `pc` 弹出，并发送到 `warp scheduler` 进行取指。

由 `simt stack` 设置的隐式 `mask` 会在该 `warp` 执行过程中一直生效，直到有其它分支管理支持对其进行修改。该 `mask` 与 `rvv` 软件形式的 `mask` 可以叠加生效。

4.2.7.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
<code>clk</code>	IN	1	系统时钟
<code>rst_n</code>	IN	1	全局复位，低有效， <code>rst_n</code> 触发后将在至少一个时钟周期后对核进行复位
<code>branch_ctl_valid_i</code>	IN	1	握手信号，为高电平时表示将执行分支管理相关的向量指令，需要接收信息。需要保持为高电平，直到 <code>ready</code> 信号为高电平
<code>branch_ctl_ready_o</code>	OUT	1	握手信号
<code>branch_ctl_opcode_i</code>	IN	1	1 为 <code>join</code> 指令，0 为分支指令
<code>branch_ctl_wid_i</code>	IN	DEPTH_WARP	执行分支管理相关向量指令的 <code>warp id</code>
<code>branch_ctl_pc_branch_i</code>	IN	32	<code>else</code> 分支路径 <code>pc</code>
<code>branch_ctl_pc_execute_i</code>	IN	32	正在执行的 <code>pc</code>
<code>branch_ctl_mask_init_i</code>	IN	NUM_THREAD	活跃线程掩码
<code>if_mask_valid_i</code>	IN	1	握手信号，为高电平时表示分支指令已执行得到结果。需要保持为高电平，直到 <code>ready</code> 信号为高电平
<code>if_mask_ready_o</code>	OUT	1	握手信号
<code>if_mask_mask_i</code>	IN	NUM_THREAD	分支路径活跃线程掩码
<code>if_mask_wid_i</code>	IN	DEPTH_WARP	执行分支指令的 <code>warp id</code>

RPC 信息			
pc_reconv_valid_i	IN	1	握手信号，为高电平时表示读取的 rpc 有效。无需保持为高电平，直到 ready 信号为高电平。
pc_reconv_i	IN	32	csr 中读取的 rpc
To WarpScheduler			
fetch_ctl_valid_o	OUT	1	握手信号，为高电平时表示将发送跳转信息到 warp scheduler。需要保持为高电平，直到 ready 信号为高电平
fetch_ctl_ready_i	IN	1	握手信号
fetch_ctl_wid_o	OUT	DEPTH_WARP	跳转信息所属的 warp 的 id
fetch_ctl_jump_o	OUT	1	跳转有效信号，为高电平时表示需要跳转至新地址，为低时不进行跳转
fetch_ctl_new_pc_o	OUT	32	需要跳转到的新地址

4.2.7.3 设计思路

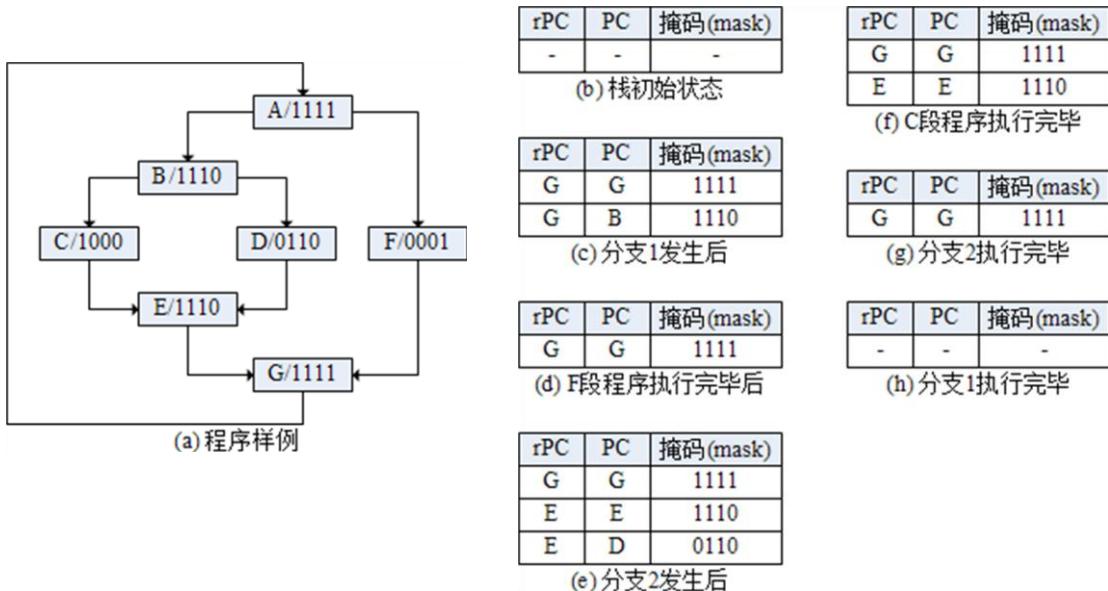


图 4-8 SimtStack 程序样例

- 图(a)为程序样例，A-G 表示代码块，右侧数字是线程活跃掩码信息，表示执行代码块的线程数量。代码块 A 中的指令有四个线程执行，代码块 B 中的指令只有三个线程执行，剩下的一个线程执行代码块 F。
- 执行图(a)时，simt stack 中的变化：
 - ①. 如图(b)为 simt stack 初始状态，因为未发生分支，未进行过压栈操作，所以栈内无任何数据；
 - ②. 当执行到 A 中最后一条指令时，产生分支。优先执行线程数量少的分支 F，所以将分支 B 的 rPC、起始 PC、掩码压入栈中，同时将该分支的分支重聚点的 rPC、起始 PC 和掩码也压入栈中，分支重聚点的起始 PC 即 rPC，如图(c)所示。
 - ③. 分支 F 执行完后，将执行代码块 G 中的第一条指令，该指令是 join 指令，当栈顶 rPC 与正在执行的 PC 相同时，join 指令会将栈顶的 rPC、起始 PC、掩码也就是图(c)中最后一行的信息弹出，将 PC 设置为 B 起始 PC，然后开始执行分支 B，此时栈内如图(d)所示，因为栈顶的信息弹出，所以此时栈顶变为图(c)中的倒数第二行。
 - ④. 执行到分支 B 中最后一条指令时，会再次发生分支，同样如步骤 2，优先执行线程数量少的分支 C，将分支 D 和分支重聚点 E 的 rPC、起始 PC、掩码压入栈中，如图(e)所示。
 - ⑤. 执行完分支 C 后同样执行代码块 E 的第一条 join 指令，将分支 D 的 rPC、起始 PC、掩码弹出，开始执行分支 D，栈内如图(f)所示。
 - ⑥. 分支 D 执行完成后将执行 E(join)，弹出分支重聚点 E 的 rPC、起始 PC、掩码，执行代码块 E，栈内如图(g)所示。
 - ⑦. 执行完 E 后，执行 G(join)，弹出分支重聚点 G 的 rPC、起始 PC、掩码，此时栈内如图(h)所示，程序执行完毕。
- 图(i)为运行图(a)程序的流程示意图，蓝色箭头表示执行该代码块的线程。

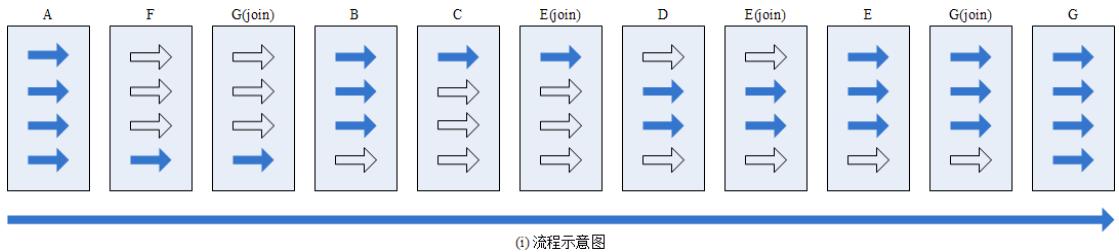


图 4-9 样例程序执行的流程示意图

4.2.8 sALU

4.2.8.1 概述

代码中的 ScalarALU (sALU) 是一个标量 ALU 模块，用于处理标量指令（加减、移位、布尔逻辑、比大小，以及跳转指令）。

下图中的 `result` 和 `result_br` 都是深度为 1 的队列，负责接收 ScalarALU 处理后的数据，其中 `io.out` 会被送到 `writeback` 中，`io.out2br` 会被送到 `branch_back` 中。

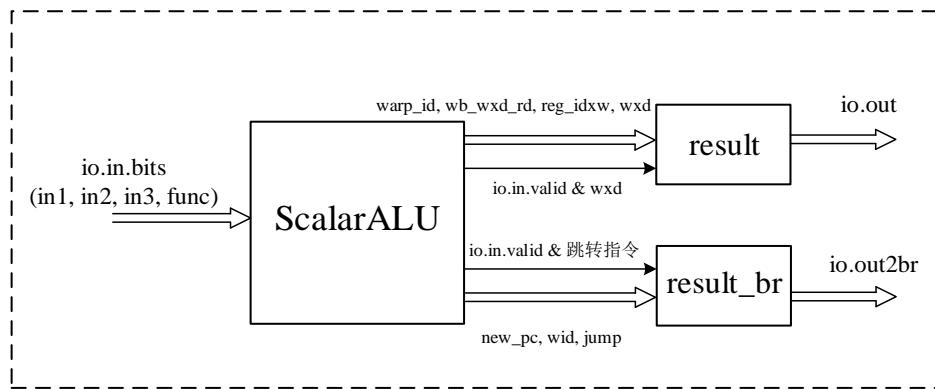


图 4-10 sALU 的架构框图

4.2.8.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
输入操作数及控制信号			
<code>in_valid_i</code>	IN	1	握手信号
<code>in_ready_o</code>	OUT	1	握手信号
<code>in1_i</code>	IN	32	操作数 1
<code>in2_i</code>	IN	32	操作数 2
<code>in3_i</code>	IN	32	操作数 3
<code>ctrl_wid_i</code>	IN	3	warp 的 id
<code>ctrl_reg_idxw_i</code>	IN	8	目的寄存器的 id

ctrl_wxd_i	IN	1	写回的目的寄存器是标量寄存器
ctrl_alu_fn_i	IN	6	指令操作码
ctrl_branch_i	IN	2	01 为跳转指令, 10 为 jal 指令, 11 为 jalr 指令
向 writeback 模块输出			
out_valid_o	OUT	1	握手信号
out_ready_i	IN	1	握手信号
wb_wxd_rd_o	OUT	32	计算结果
wxd_o	OUT	1	写回的目的寄存器是标量寄存器
reg_idxw_o	OUT	8	目的寄存器的 id
warp_id_o	OUT	3	warp 的 id
向 branch_back 模块输出			
out2br_valid_o	OUT	1	握手信号
out2br_ready_i	IN	1	握手信号
br_wid_o	OUT	3	warp 的 id
br_jump_o	OUT	1	跳转使能
br_new_pc_o	OUT	32	新 pc

4.2.9 vALU

4.2.9.1 概述

vALU 是一个向量 ALU 模块，用于处理向量的加减、移位、布尔逻辑、比大小，以及跳转指令。典型运算消耗 1cycle，支持折叠功能。下图中 softThread 是 GPGPU SIMD 流水线的宽度，而 hard_thread 是实际运算单元的数量，此时为 softThread = hardThread 的情况。

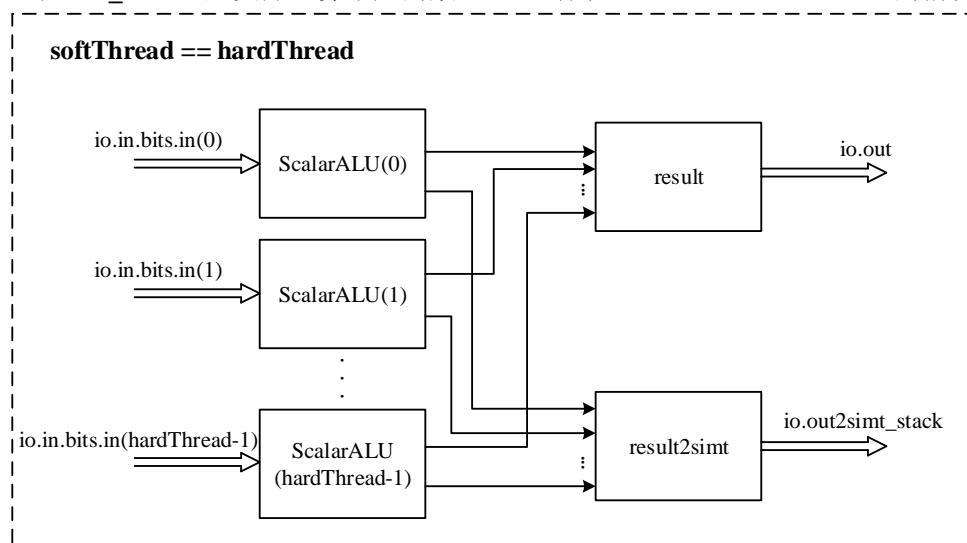


图 4-11 softThread = hardThread 时 vALU 的架构图

当 softThread != hardThread（根据定义，softThread 必须大于 hardThread）时，硬件无法同时处理所有的线程。此时处理的基本思路是采用一个 Reg 寄存外部过来的数据，在硬

件处理完前 hardThread 个线程的数据后，再传递随后的 hardThread 个线程的数据给硬件以此类推。完成计算后，再采用一个 Reg 将结果一起输出，如下图所示。

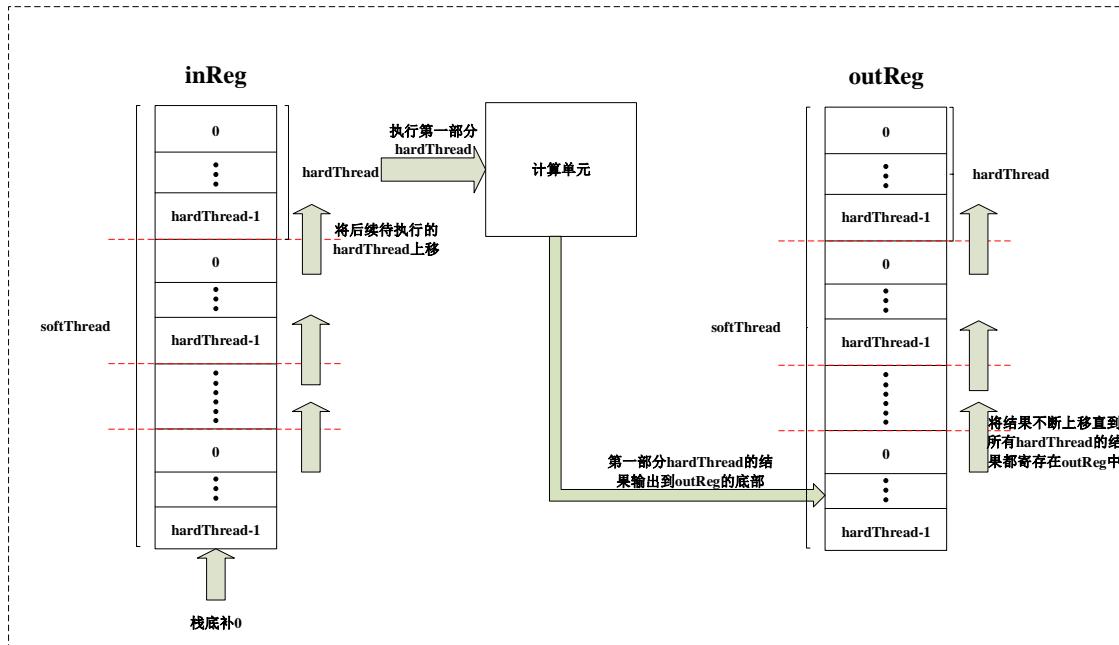


图 4-12 `softThread` \neq `hardThread` 时 vALU 的架构图

4.2.9.2 信号描述

vALU 模块调用了 `NUM_THREAD` 个 sALU 模块，因此信号是相同的，区别在于信号的位宽是 sALU 的 `NUM_THREAD` 倍。

4.2.10 vMUL

4.2.10.1 概述

vMUL 是一个向量乘法模块，用于处理向量/标量的整形乘法指令。典型乘法消耗 2cycle。下图中的 `ArrayMultiplier` 是一个 32-bit 的乘法运算单元，此模块也有类似 vALU 的折叠功能，标量乘法指令的执行结果由第 0 个乘法单元输出。`result_x` 和 `result_v` 是两个深度为 1 的 FIFO，分别输出标量和向量指令的结果。

4.2.10.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
输入操作数及控制信号			
in_valid_i	IN	1	握手信号

in_ready_o	OUT	1	握手信号
in1_i	IN	NUM_THREAD*32	操作数 1
in2_i	IN	NUM_THREAD*32	操作数 2
in3_i	IN	NUM_THREAD*32	操作数 3
mask_i	IN	NUM_THREAD	线程 mask
ctrl_wid_i	IN	3	warp 的 id
ctrl_reg_idxw_i	IN	8	目的寄存器的 id
ctrl_wxd_i	IN	1	写回的目的寄存器是标量寄存器
ctrl_alu_fn_i	IN	6	指令操作码
ctrl_wvd_i	IN	1	写回的目的寄存器是向量寄存器
ctrl_reverse_i	IN	1	操作数调换
标量输出			
outx_valid_o	OUT	1	握手信号
outx_ready_i	IN	1	握手信号
outx_wb_wxd_rd_o	OUT	32	标量计算结果
outx_wxd_o	OUT	1	写回的目的寄存器是标量寄存器
outx_reg_idxw_o	OUT	8	目的寄存器的 id
outx_warp_id_o	OUT	3	warp 的 id
向量输出			
outv_valid_o	OUT	1	握手信号
outv_ready_i	IN	1	握手信号
outv_wb_wxd_rd_o	OUT	NUM_THREAD*32	向量计算结果
outv_wvd_o	OUT	1	写回的目的寄存器是向量寄存器
outv_reg_idxw_o	OUT	8	目的寄存器的 id
outv_warp_id_o	OUT	3	warp 的 id

4.2.10.3 设计思路

乘法的计算过程、乘影乘法器的计算流程和其中 addOneColumn 的结构如下图所示。其核心思想利用 addOneColumn 模块对每一列的数据个数进行压缩。

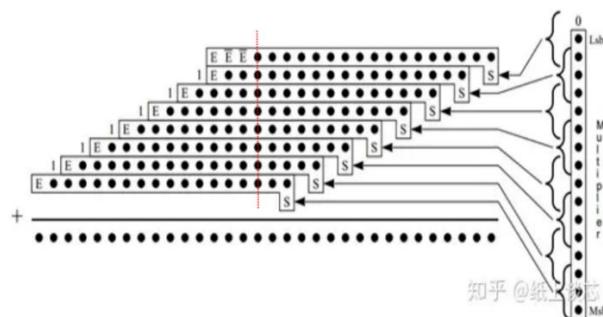


图 4-13 乘法的典型运算流程

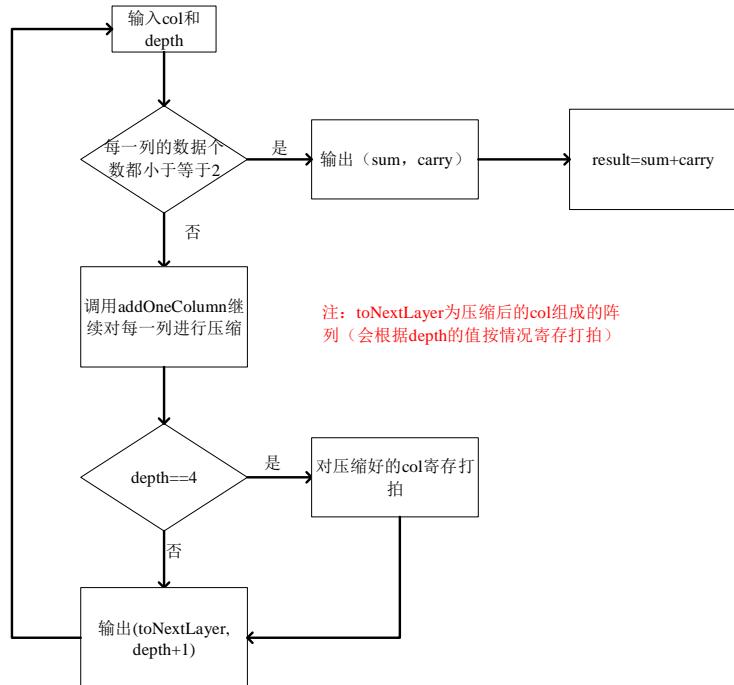


图 4-14 乘影乘法器的计算流程

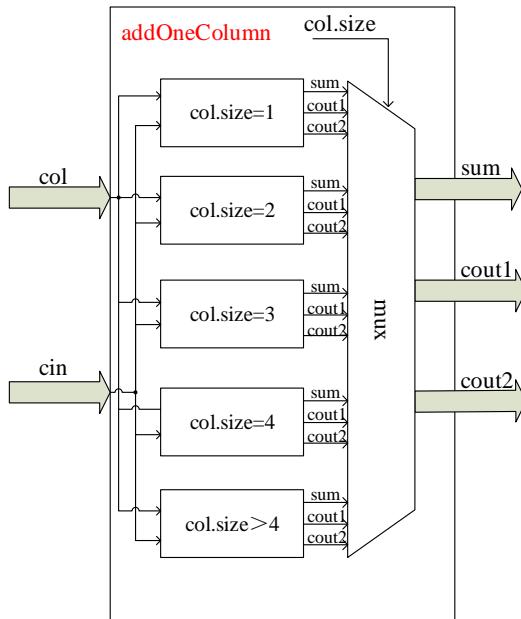


图 4-15 addOneColumn 的结构示意图

乘法模块中使用了 booth 编码+Wallace 树+CSA 进位保存加法器的组合来实现快速乘法运算，下面介绍各算法原理。

(1) booth 编码

使用 booth 编码主要是为了减少运算的次数，若计算 $M=A \times B$ ，其中 A 和 B 都是 5bit 的数，若 B 中有 n 个 bit 为 1，则要进行 n 次移位加法操作。

一个有符号位的二进制数的补码可以用下式表示：

$$B = -2^{n-1}B_{n-1} + 2^{n-2}B_{n-2} + \dots + B_0 \quad (5-1)$$

它等价于：

$$B = 2^{n-1}(-B_{n-1} + B_{n-2}) + 2^{n-2}(-B_{n-2} + B_{n-3}) + \dots + (-B_0 + B_{-1}) \quad (5-2)$$

$$B = 2^{n-2}(-2B_{n-1} + B_{n-2} + B_{n-3}) + 2^{n-4}(-2B_{n-3} + B_{n-4} + B_{n-5}) + \dots + (-2B_1 + B_0 + B_{-1}) \quad (5-3)$$

公式 (5-3) 即是基 4booth 编码的表示形式，将 (5-1) 改写成 (5-3) 后，可以大大减少非零行的数目。乘数按三位一组进行划分，相互重叠一位。把公式 (5-1) 重写为公式 (5-3) 后，每一组按下表编码，并形成一个部分积。以上述提到的 5bit 乘法为例，可以将加法操作减少到 3 次。

输入(乘数位)			部分积
B_{i+1}	B_i	B_{i-1}	PP_i
0	0	0	0
0	0	1	A
0	1	0	A
0	1	1	2^*A
1	0	0	-2^*A
1	0	1	$-A$
1	1	0	$-A$
1	1	1	0

示例： $B=-7=1111_1001$, $A=-3=1111_1101$, 则 $M=A*B=21=0001_0101$

首先将乘数 A 按照 booth 编码进行转换：

$A=1111_1101$, $-A=0000_0011$, $+2A=1111_1010$, $-2A=0000_0110$

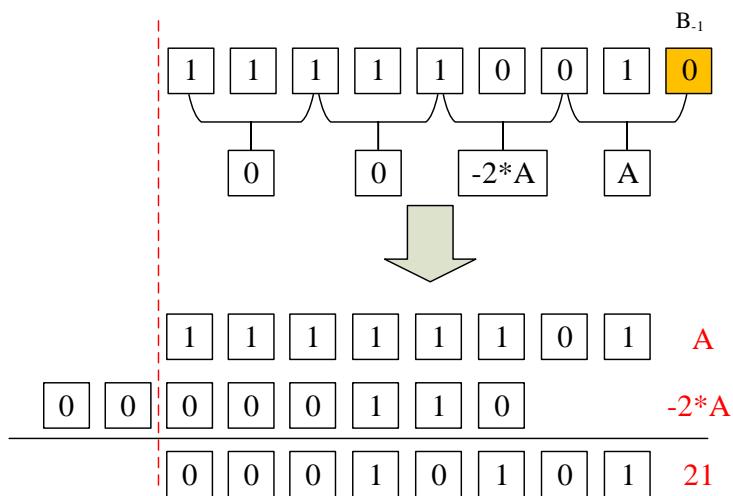


图 4-16 采用 booth 编码的乘法样例

(2) Wallace 树

Wallace 树是一种高效快速的加法树结构，简单地讲即许多个加数求和，每 3 个加数分为一组，压缩至 2 个加数，循环往复，如图所示。这种高并行度的优化结构大大加快了乘法器的计算速度，因为直接用加法器计算多位加法时，这样计算每一级逻辑是多位加法器延时，而采用 Wallace 树结构每一级逻辑相当于 1 位全加器延时。这种 3: 2 的加法器也叫 32 压缩器，将两个 32 压缩器组合可以构成 53 压缩器，如下图所示。

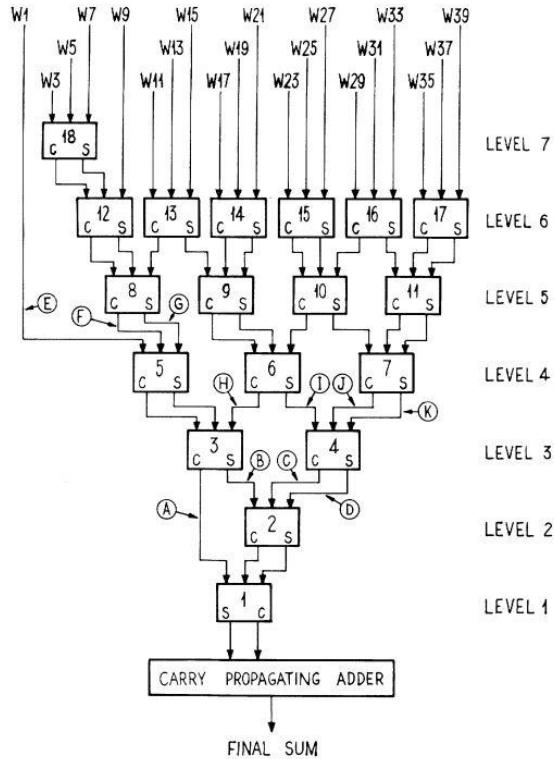


图 4-17 Wallace 树

(3) 进位保存加法器 CSA

在加法运算中，当加数的个数大于 2 时，最直接的计算方法是先计算两个数的和，再将和与第三个数做加法，以此类推，如图所示。

对于 m 个数相加，每个数 n 比特宽，总共需要 $m-1$ 次加法。假如使用超前进位加法器 LCA 的话，直接相加法总共需要的门延迟为 $O(m\lg n)$ 。使用进位保存加法器 CSA 结构则可以将门延迟降到更低，其结构如图所示，它将 3 个数相加转换为 2 个数相加，最后达到 LCA 的加数位宽为 $n+m-2$ ，LCA 的门延迟为 $O(\lg(n+m))$ ，则总门延迟为 $O(m+\lg(n+m))$ 。

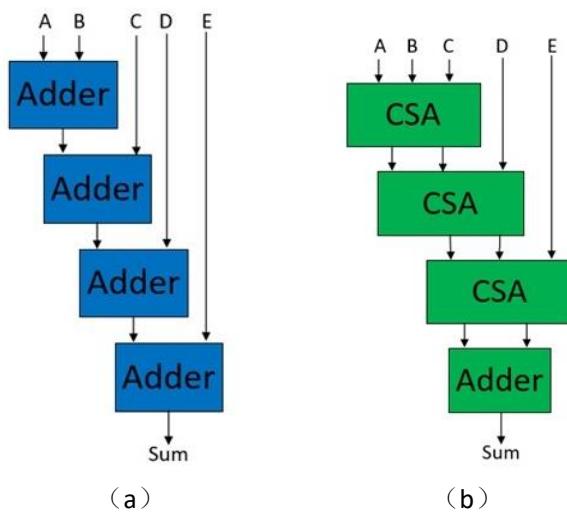


图 4-18 (a) 直接相加；(b) CSA 加法

4.2.11 CSR

4.2.11.1 概述

在 warp 启动时，对应的 CSR 会设置好应用程序所需的一些值，包括 thread id 等，vsetl 也在此计算。其余与 riscv cpu 的 CSR 功能一致。

下图中，CTA2csr 由 CTA 发出，在程序开始时对 CSRFile 进行设置。in 包括 ctrl 信号和输入数据；out 向外输出向量 CSR 寄存器的值。

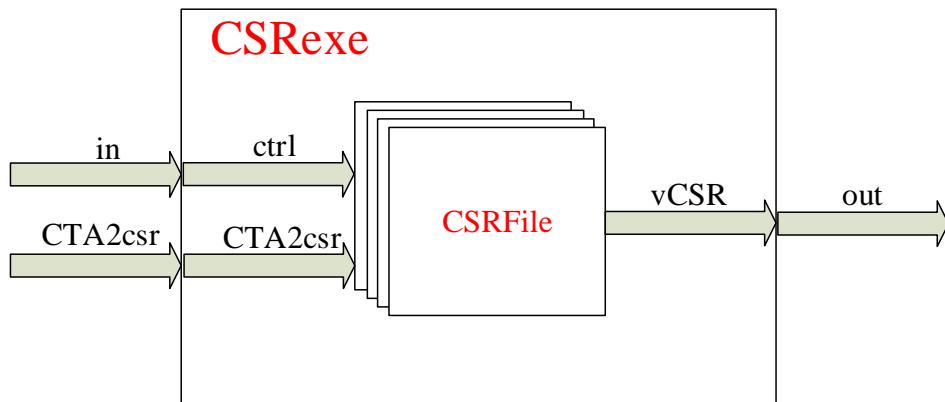


图 4-19 CSR 架构框图

4.2.11.2 信号描述

该章节将对 CSR 模块的信号及其相关细节进行描述。

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
CTA2csr			
名称	类型	位宽	描述
CTA2csr_valid_i	IN	1	有效信号
dispatch2cu_wg_wf_count_i	IN	4	该 workgroup 中 warp 数量
dispatch2cu_wf_size_dispatch_i	IN	10	该 warp 中 thread 数量
dispatch2cu_sgpr_base_dispatch_i	IN	11	sgpr 开始位置
dispatch2cu_vgpr_base_dispatch_i	IN	11	vgpr 开始位置
dispatch2cu_wf_tag_dispatch_i	IN	7	warp 的 id (高 3bit 索引 wg, 低 4bit 索引 warp)
dispatch2cu_lds_base_dispatch_i	IN	18	lds 开始位置
dispatch2cu_start_pc_dispatch_i	IN	32	pc 开始地址
dispatch2cu_pds_base_dispatch_i	IN	32	该 workgroup 分配的 private memory 的 baseaddr
dispatch2cu_gds_base_dispatch_i	IN	32	该 workgroup 分配的 global memory 的 baseaddr

dispatch2cu_csr_knl_dispatch_i	IN	32	该 workgroup 的 metadata buffer 的 baseaddr
dispatch2cu_wgid_x_dispatch_i	IN	10	该 workgroup 在 NDRange 中的 x 轴坐标
dispatch2cu_wgid_y_dispatch_i	IN	10	该 workgroup 在 NDRange 中的 y 轴坐标
dispatch2cu_wgid_z_dispatch_i	IN	10	该 workgroup 在 NDRange 中的 z 轴坐标
dispatch2cu_wg_id_i	IN	32	workgroup 的 id
wid_i	IN	3	warp 的 id
控制信号			
ctrl_inst_i	IN	32	指令
ctrl_csr_i	IN	2	csr 工作模式
ctrl_custom_signal_0_i	IN	1	只有 setrpc 指令时为 1, 将分支汇合地址写入 rpc 的 csr 寄存器
ctrl_isvec_i	IN	1	是否为向量
ctrl_reg_idxw_i	IN	8	目的寄存器的 id
ctrl_wxd_i	IN	1	写回的目的寄存器是标量寄存器
ctrl_wid_i	IN	3	warp 的 id
数据输入接口			
in_valid_i	IN	1	握手信号
in_ready_o	OUT	1	握手信号
in1_i	IN	32	32bit 数据
rm_wid_i	IN	3	warp 的 id
lsu_wid_i	IN	3	warp 的 id
simt_wid_i	IN	3	warp 的 id
数据输出接口			
out_valid_o	OUT	1	握手信号
out_ready_i	IN	1	握手信号
wb_wxd_rd_o	OUT	32	输出数据
wxd_o	OUT	1	写回的目的寄存器是标量寄存器
reg_idxw_o	OUT	8	目的寄存器的 id
warp_id_o	OUT	3	warp 的 id
rm_o	OUT	9	舍入模式
sgpr_base_o	OUT	88	所有 warp 的标量寄存器地址的拼接
vgpr_base_o	OUT	88	所有 warp 的向量寄存器地址的拼接
lsu_tid_o	OUT	32	输出给 lsu 模块的线程 id
lsu_pds_o	OUT	32	输出给 lsu 模块的 private memory 的 baseaddr

lsu_numw_o	OUT	32	输出给 lsu 模块的 warp 总数
simt_rpc_o	OUT	32	输出给 SIMT 堆栈的分支重聚点 pc

4.2.12 TensorCore

4.2.12.1 概述

TensorCore 用于特定格式下的张量计算。TCDotProduct 模块的作用是计算原理图中单个窗口的卷积结果，通过对 TCDotProduct 模块例化调用 DimM×DimK 次得到整体卷积的结果。

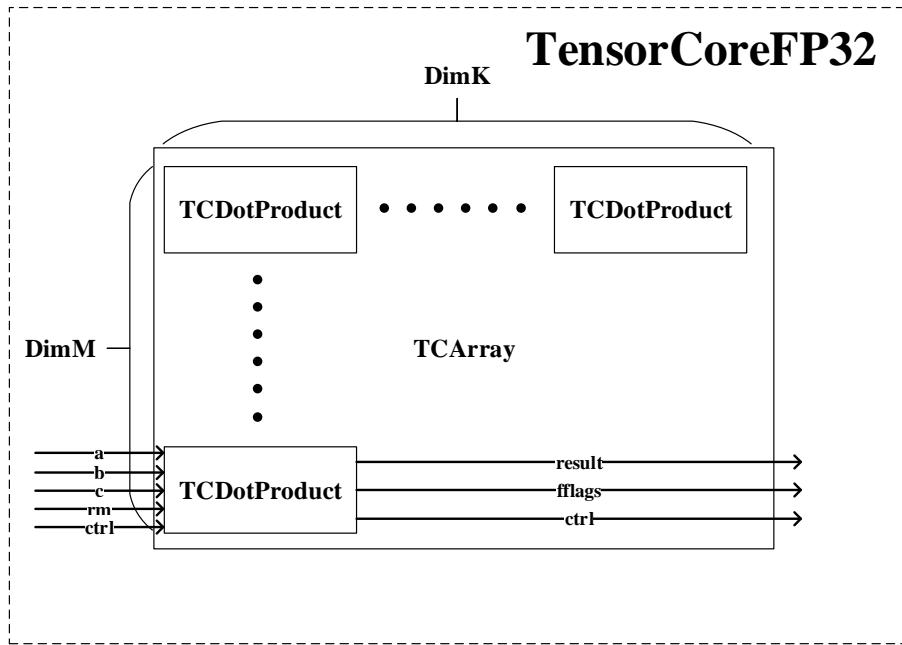


图 4-20 TensorCore 的架构框图

目前，在 GPGPU 上进行卷积计算最常用的方式是将卷积运算转化为通用矩阵乘法操作，充分利用 GPGPU 软硬件对通用矩阵乘法已有的各种优化来获取卷积计算的加速。其具体原理如下图所示。这里展示了一张 5×5 的输入特征图与一个 3×3 的卷积核进行卷积且结果与偏置值累加。基本流程为将输入特征图和卷积核中的元素一一对应地进行乘法运算，然后将得到的积进行累加，再与偏置值相加，得到该位置的最终卷积结果。

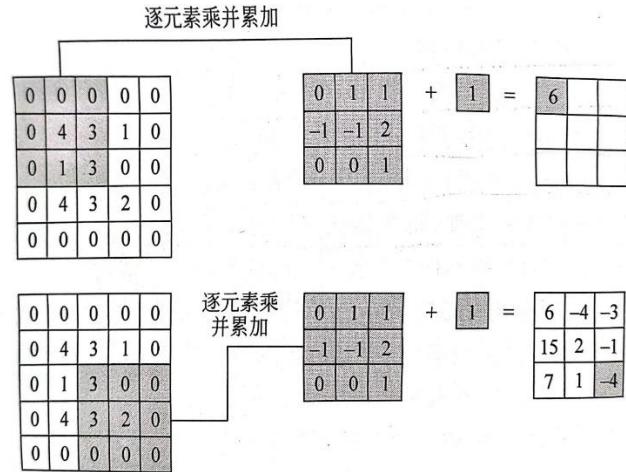


图 4-21 直接卷积转化为矩阵相乘

4.2.12.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位, 低有效
输入操作数及控制信号			
in_valid_i	IN	1	握手信号
in_ready_o	OUT	1	握手信号
in1_i	IN	NUM_THREAD*32	操作数 1
in2_i	IN	NUM_THREAD*32	操作数 2
in3_i	IN	NUM_THREAD*32	操作数 3
ctrl_wid_i	IN	3	warp 的 id
ctrl_reg_idxw_i	IN	8	目的寄存器的 id
rm_i	IN	3	舍入模式
标量输出			
out_valid_o	OUT	1	握手信号
out_ready_i	IN	1	握手信号
wb_wxd_rd_o	OUT	NUM_THREAD *32	计算结果
wvd_o	OUT	1	写回的目的寄存器是向量寄存器
reg_idxw_o	OUT	8	目的寄存器的 id
warp_id_o	OUT	3	warp 的 id
wvd_mask_o	OUT	NUM_THREAD	线程 mask (全 1)

4.2.13 LSU

4.2.13.1 概述

核心的访存单元，适用于标量向量和标量访存指令。

通过判断地址范围将请求发送给 Sharemem 或 Dcache。

在发送向量请求时，会将同一个 Cacheline 的请求合并。

内部有 MSHR，可以记录多个 LSU 请求，并收集从 ShareMem 和 Dcache 返回的数据，收集完成后将应答返回给流水线。

LSU 还会记录现有的访存请求信息，以实现 fence 指令。当遇到该指令后，会让所属 warp 的所有访存请求处理完成后再发送新的访存请求。

其硬件结构主要包括 input_fifo、addrcalculate、mshr、rsp_arb、lsu2wb 以及 shiftboard 构成，如下图所示。其中，input_fifo 用于接收来自流水线的请求信息；addrcalculate 用于计算访存地址，并向 Dcache/Sharemem 发送请求；rsp_arb 用于仲裁来自 Dcache/Sharemem 的响应；mshr 用于收集 Dcache/Sharemem 的响应信息，并将响应返回给 lsu2wb；shiftboard 用于 fence 指令；lsu2wb 负责区分 mshr 的响应信息是向量还是标量指令，并将其返回给流水线。

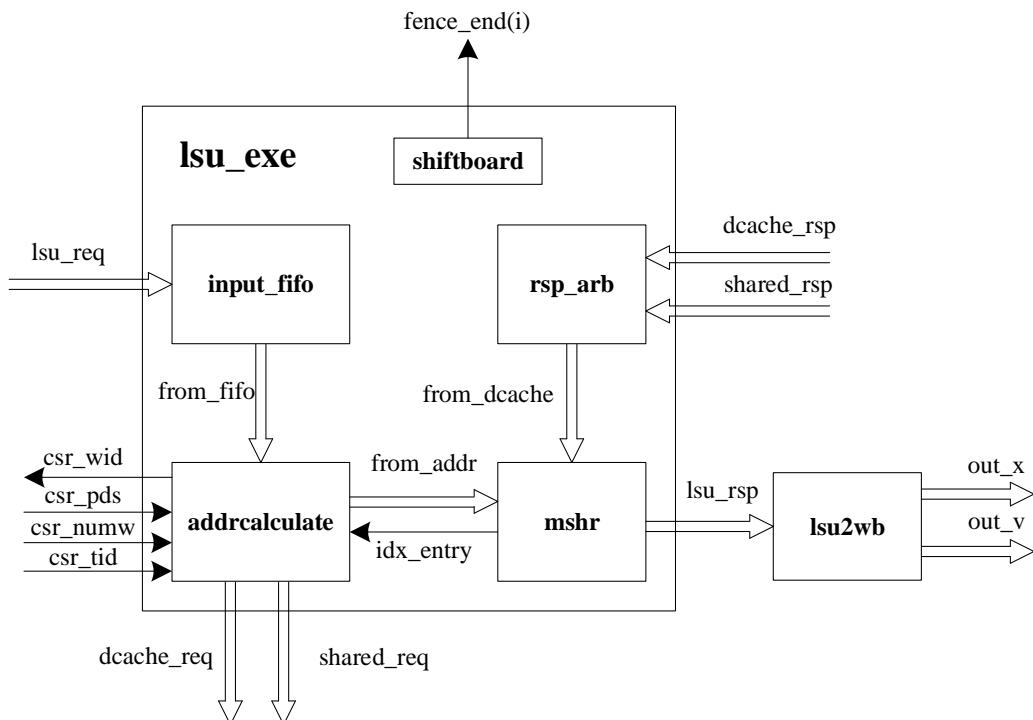


图 4-22 LSU 结构框图

4.2.13.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟

<code>rst_n</code>	IN	1	复位信号, 低电平有效, 对内部寄存器进行复位。
流水线请求接口			
<code>lsu_req_valid_i</code>	IN	1	请求有效信号
<code>lsu_req_ready_o</code>	OUT	1	lsu 内部 ready 信号
<code>lsu_req_in1_i</code>	IN	128	操作数 1, 位宽随 NUM_THREAD 变化
<code>lsu_req_in2_i</code>	IN	128	操作数 2, 位宽随 NUM_THREAD 变化
<code>lsu_req_in3_i</code>	IN	128	操作数 3, 位宽随 NUM_THREAD 变化
<code>lsu_req_mask_i</code>	IN	4	线程掩码, 位宽随 NUM_THREAD 变化
<code>lsu_req_wid_i</code>	IN	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
<code>lsu_req_isvec_i</code>	IN	1	表示是否为向量指令
<code>lsu_req_mem_whb_i</code>	IN	2	表示访存操作类型(字、半字或字节操作)
<code>lsu_req_unsigned_i</code>	IN	1	表示访存的数据是否为无符号数
<code>lsu_req_alu_fn_i</code>	IN	6	指令操作码, 在 lsu 中用于原子指令
<code>lsu_req_is_vls12_i</code>	IN	1	表示是否为长立即数偏移的向量访存指令
<code>lsu_req_disable_mask_i</code>	IN	1	表示是否为自定义访存指令
<code>lsu_req_mem_cmd_i</code>	IN	2	访存请求类型, 2 是写请求, 1 是读请求
<code>lsu_req_mop_i</code>	IN	2	访存地址模式
<code>lsu_req_reg_idx_i</code>	IN	8	目的寄存器地址
<code>lsu_req_wvd_i</code>	IN	1	表示写回目的寄存器是向量寄存器
<code>lsu_req_fence_i</code>	IN	1	表示是否为 fence 指令
<code>lsu_req_wxd_i</code>	IN	1	表示写回目的寄存器是标量寄存器
<code>lsu_req_atomic_i</code>	IN	1	表示是否为原子指令
<code>lsu_req_aq_i</code>	IN	1	acquire, 在这条指令之后的访存指令必须等到该指令完成后才开始执行
<code>lsu_req_rl_i</code>	IN	1	release, 在这条指令之前的访存指令的结果必须在该指令执行之前被观察到
流水线响应接口			
<code>out_x_valid_o</code>	OUT	1	标量指令访存指令响应有效信号

out_x_ready_i	IN	1	表示标量写回端口是否就绪
out_x_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_x_wxd_o	OUT	1	写回目的寄存器是标量寄存器
out_x_reg_idxw_o	OUT	8	标量寄存器地址
out_x_wb_wxd_rd_o	OUT	32	需要写回的数据
out_v_valid_o	OUT	1	向量指令访存指令响应有效信号
out_v_ready_i	IN	1	表示向量写回端口是否就绪
out_v_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_v_wvd_o	OUT	1	写回目的寄存器是向量寄存器
out_v_reg_idxw_o	OUT	8	向量寄存器地址
out_v_wvd_mask_o	OUT	4	线程掩码, 位宽随 NUM_THREAD 变化
out_v_wb_wxd_rd_o	OUT	128	需要写回的数据, 位宽随 NUM_THREAD 变化
csr 交互接口			
csr_pds_i	IN	32	private memory 的 base addr
csr_numw_i	IN	32	当前 workgroup 的 warp 数量
csr_tid_i	IN	32	warp 内最小的线程 id
csr_wid_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
fence 完成接口			
fence_end_o	OUT	8	指示当前 warp 是否已完成前序的访存指令, 位宽随 workgroup 内的最大 warp 数量变化
Dcache 请求接口			
dcache_req_valid_o	OUT	1	Dcache 请求有效信号
dcache_req_ready_i	IN	1	Dcache 请求就绪信号
dcache_req_instrid_o	OUT	3	来自 lsu mshr 内部的 id 信号
dcache_req_setidx_o	OUT	5	当前 Cacheline 的 index/setidx
dcache_req_tag_o	OUT	24	当前 Cacheline 的 tag
dcache_req_activemask_o	OUT	4	线程掩码
dcache_req_blockoffset_o	OUT	4	各线程在当前 Cacheline 的 blockoffset
dcache_req_wordoffset1h_o	OUT	16	各线程的字节使能
dcache_req_data_o	OUT	128	写入的数据 (用于 store 指令)
dcache_req_opcode_o	OUT	3	访问 Dcache 的类型
dcache_req_param_o	OUT	4	访问 Dcache 类型的补充信号
Dcache 响应接口			

dcache_rsp_valid_i	IN	1	Dcache 响应有效信号
dcache_rsp_ready_o	OUT	1	Dcache 响应就绪信号
dcache_rsp_instrid_i	IN	3	id 信号, 与 dcache_req_instrid_o 对应
dcache_rsp_data_i	IN	128	读出的数据 (用于 load 指令)
dcache_rsp_activemask_i	IN	4	线程掩码
Sharemem 请求接口			
shared_req_valid_o	OUT	1	Sharemem 请求有效信号
shared_req_ready_i	IN	1	Sharemem 请求就绪信号
shared_req_instrid_o	OUT	3	来自 lsu mshr 内部的 id 信号
shared_req_iswrite_o	OUT	1	是否为写操作
shared_req_setidx_o	OUT	5	当前 Cacheline 的 index/setidx, (Sharemem 内部没有 Cacheline 的概念, 但是其请求格式与 Dcache 一致)
shared_req_tag_o	OUT	24	当前 Cacheline 的 tag
shared_req_activemask_o	OUT	4	线程掩码
shared_req_blockoffset_o	OUT	4	各线程在当前 Cacheline 的 blockoffset
shared_req_wordoffset1h_o	OUT	16	各线程在当前 Cacheline 的 blockoffset
shared_req_data_o	OUT	128	写入的数据 (用于 store 指令)
Sharemem 响应接口			
shared_rsp_valid_i	IN	1	Sharemem 响应有效信号
shared_rsp_ready_o	OUT	1	Sharemem 响应就绪信号
shared_rsp_instrid_i	IN	3	id 信号, 与 shared_req_instrid_o 对应
shared_rsp_data_i	IN	128	读出的数据 (用于 load 指令)
shared_rsp_activemask_i	IN	4	线程掩码

4.2.13.3 设计原理

在 lsu 内部, input_fifo 是一个 FIFO, rsp_arb 和 lsu2wb 分别是一个仲裁器和译码器, 下面介绍 addrcalculate、mshr 和 shiftboard 的工作原理。

4.2.13.3.1 addrcalculate

addrcaculate 用于接收来自 input_fifo 的信号, 计算出 Load/Store 指令的访存地址, 并根据地址是否大于 SHARED_ADDR_MAX (Sharedmem 的最大地址) 向 Dcache/Sharemem 发起访存请求。其内部有一个状态机, 其状态如下:

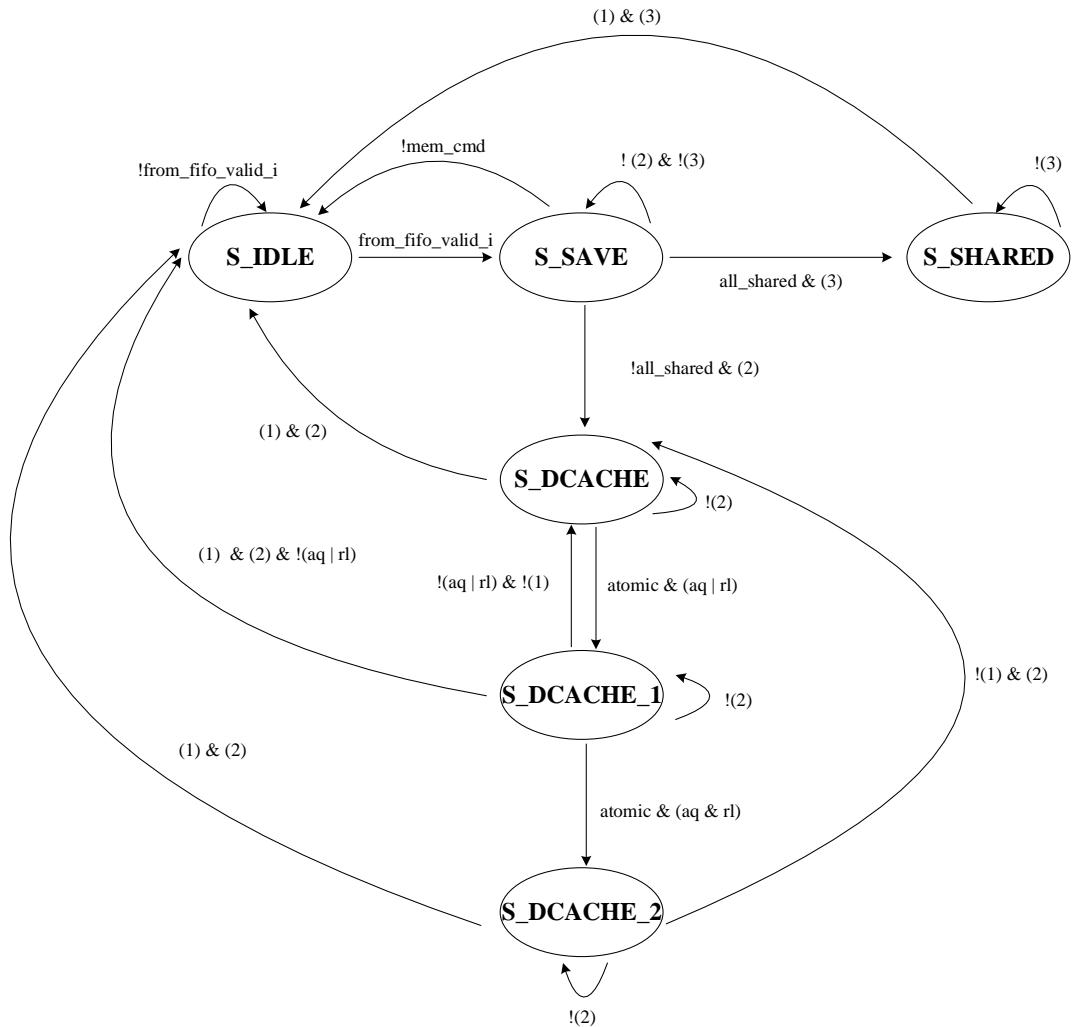


图 4-23 addrcalculate 的状态机示意图

注:

- (1) $\text{cnt} \geq \text{NUM_THREAD} \mid\mid \text{mask_next} == 0$ (计数器溢出或者 mask 全为 0)
- (2) $\text{dcache_req_valid_o} \&\& \text{dcache_req_ready_i}$
- (3) $\text{shared_req_valid_o} \&\& \text{shared_req_ready_i}$

框图说明:

- (1) **S_IDLE** 为空闲状态, 当 **from_fifo_valid_i** 为高时进入 **save** 状态。
- (2) 在 **S_SAVE** 状态下, 模块利用 **reg_save** 寄存来自 **input_fifo** 的信号, 并根据 **in1** 和 **in2** 计算出访存地址 **addr**, 判断每个线程的访存地址 **addr(x)** 是否都小于 **SHARED_ADDR_MAX**, 如果都小于, 则进入 **S_SHARED** 状态; 否则, 则进入 **S_DCACHE** 状态。除此之外, **S_SAVE** 状态下, 模块还要将 **tag** 信息输出给 **mshr**, 并从 **mshr** 中读取 **idx_entry**。
- (3) **S_SHARED** 状态下, 向外发出 **shared_req** (共享存储器访存请求)。
- (4) **S_DCACHE** 状态下, 向外发出 **dcache_req** (全局和局部存储器访存请求)。如果是原子操作, 则最多分为三个周期进行完成 (对应 **Dcache** 有三个状态: **S_DCACHE**、**S_DCACHE_1**、**S_DCACHE_2**)。这三种状态对应的操作如下所示 (**invalid** 表示向 **dcache** 发起 **invalid** 请求):

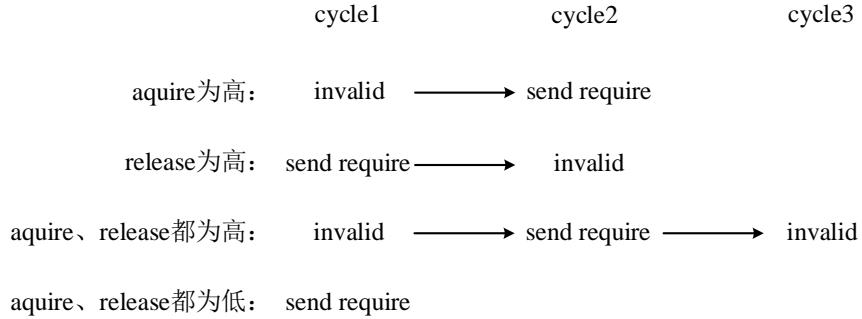


图 4-24 amo 指令在不同周期下的操作

(5) 模块内部有一个计数器（最大值为 NUM_THREAD），当计数器超过最大值或 mask_next 为 0 时，S_DCACHE 和 S_SHARED 状态都会返回到 S_IDLE 状态。

LSU 发送请求的地址格式如下表所示，其中，blockoffset 会随着 Cacheline 大小的变化而变化，setIdx 会随着 L1\$D 中的 Cacheline 数量和组相联数量变化而变化，对应地，tag 也会随着 blockoffset 和 setIdx 的位宽变化而变化。

31	12 11	7 6	2 1	0
tag			setIdx	blockoffset

wordoffset

考虑到 risc-v 的向量指令有多种寻址模式，在乘影 GPGPU 中，其寻址方式与控制信号里面的 disable_mask (在向量指令中，如果是自定义访存指令，则 disable_mask 拉高)、vls12 和 mop (寻址模式) 有关，其 addr(x) 的组合逻辑如下表所示：

isvec	disable_mask	vls12	mop	指令类型	addr
1	1	1	x	自定义访存指令，对标标量访存的长立即数版本	vs1 + offset
1	1	0	x	自定义访存指令，专用于访问 private memory	$(vs1+imm)*num_thread_in_wg + thread_idx_csr_pds$
1	0	0	11	indexed-ordered 寻址模式	$rs1 + vs2[i]$
1	0	0	00	unit-stride 寻址模式	$rs1 + i*4$
1	0	0	10/ 01	constant-strided 寻址模式	$rs1 + i*vs2[i]$
0	0	x	x	标量存储指令	—

在输出非 amo/fence 时，不同线程的 tag 和 setidx 相等时，可以同时输出多个线程的请求，如下所示：

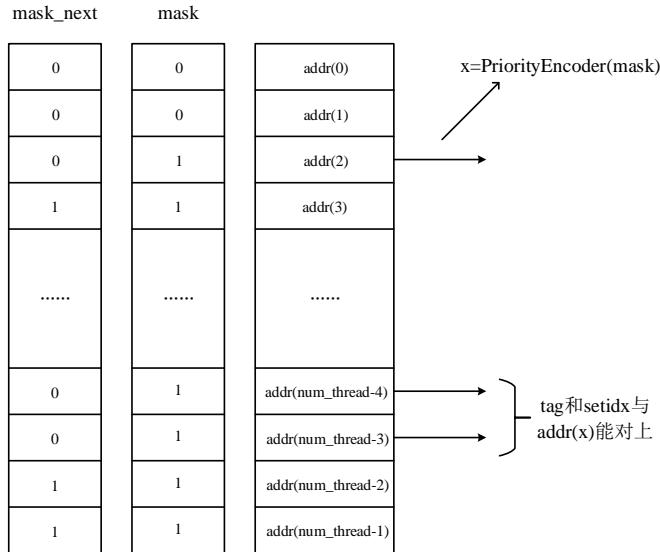


图 4-25 非 amo/fence 指令的请求合并方式

而在 amo 指令下，合并不同线程的请求是不被允许的，如下图所示：

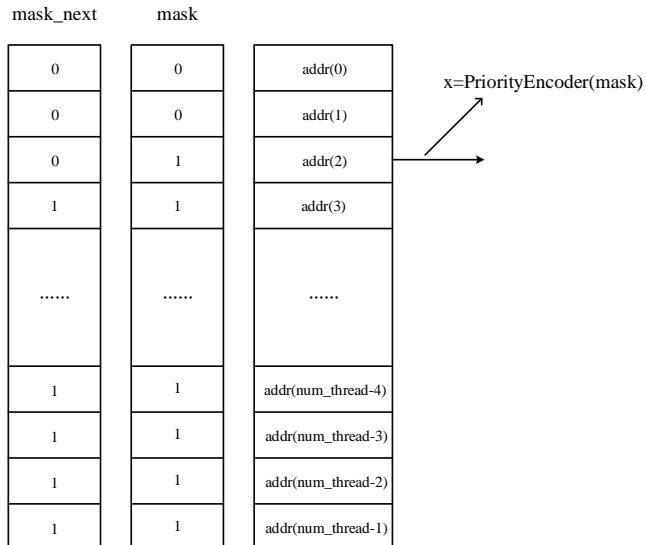


图 4-26 amo 指令的请求方式

在输出请求信号给 Dcache 时，其 **opcode** 和 **param** 的作用如下表所示：

opcode	param_wire	DCache 执行的操作
0	0	regular read
0	1	LR 指令
1	0	regular write
1	1	SC 指令
2	--	AMO 指令（操作类型根据 param_wire 来定）
3	0	invalid
3	1	flush
3	2	waitMSHR

为了处理 **param** 的输出，模块内采用到了 **param_wire_alt**，根据 **ctlr.alu_fn**（操作类型）赋予其不同的值，并在 **atomic** 为高时输出给 **param_wire**，最后输出给 **Dcache**, **param_wire_alt**

的组合逻辑如下所示：

reg_save_ctrl_alu_fn	param_wire_alt
FN_SWAP	4'b1111
FN_AMOADD	4'b0000
FN_XOR	4'b0001
FN_AND	4'b0011
FN_OR	4'b0010
FN_MIN	4'b0100
FN_MAX	4'b0101
FN_MINU	4'b0110
FN_MAXU	4'b0111
else	4'b0001

最后在不同 state 下 dcache_req_opcode_o 和 dcache_req_param_o 的组合如下

atomic	aq	rl	fence	state	dcache_req_opcode_o	dcache_req_param_o
1	1	1	x	S_DCACHE_2	3	0
				S_DCACHE_1	2	param_wire_alt
				S_DCACHE	3	0
1	1	0	x	S_DCACHE_1	2	param_wire_alt
				S_DCACHE	3	0
1	0	1	x	S_DCACHE_1	3	0
				S_DCACHE	2	param_wire_alt
1	0	0	x	S_DCACHE	ctrl.alu_fn == FN_ADD	mem_cm d(1)
					ctrl.alu_fn != FN_ADD	2
					ctrl.mem_cmd(1)	param_wire_alt
0	x	x	1	S_DCACHE	3	0
0	x	x	0	S_DCACHE	ctrl.mem_cmd(1)	0

4.2.13.3.2 mshr

mshr 的设计框图如下：

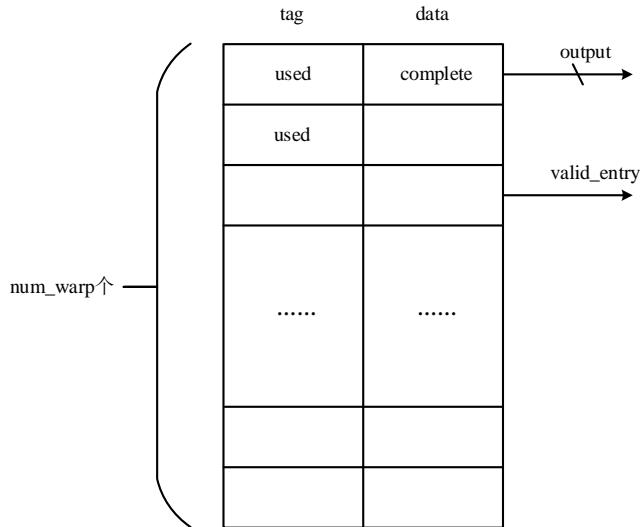


图 4-27 mshr 的设计框图

框图说明：

(1) pipeline 中的 mshr 不包括对 bank conflict 的处理，也不包括对 tag 未命中时的处理（这部分由 DCache/Sharemem 自行处理）。所以，mshr 模块的核心就是一个 tag 缓冲（用于缓存 wid、mask 等信息）和 data 缓冲，用于向 pipe 返回 lsu_rsp 以及指令的控制信号。

(2) tag 缓冲用于缓存来自 addrCalculate 的控制信号，当模块接收到一组 tag 信息时，对应 entry id 的 used 会拉高。

(3) data 缓冲用于缓存来自 DCache/Sharemem 的访问响应，当模块接收完一次响应后，对应 entry id 的 complete 会拉高，此时会将对应 entry 的 data 和 tag 信息发送给 lsu2wb。

MSHR 内部有一个状态机，包括三个状态，`S_IDLE`、`S_ADD` 和 `S_OUT`。状态机的跳转如下图所示。

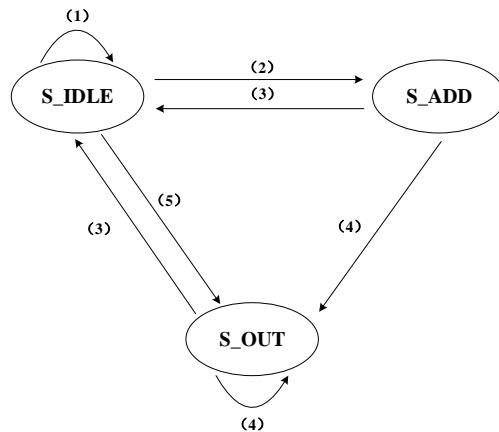


图 4-28 mshr 的状态机跳转

注：

- (1) !(from_dcache_valid_i & from_dcache_ready_o) | (3)
- (2) (from_dcache_valid_i & from_dcache_ready_o) & (from_addr_valid_i & from_add_ready_o)
- (3) !(|complete & to_pipe_ready_i)
- (4) |complete & to_pipe_ready_i

(5) ((from_dcache_valid_i & from_dcache_ready_o) & to_pipe_ready_i & current_mask == activemask)| (4)

状态机的说明如下：

(1) S_IDLE 为空闲状态，只有在这个状态下，才会接收来自 addrcaculate 的 tag 信息，且只有这个状态才会向外输出 valid_entry。当来自 DCache/Sharemem 的响应接收完成后，进入 S_OUT 状态；如果 DCache/Sharemem 的响应和 addrcaculate 的 tag 信息同时到来，则进入 S_ADD 状态，在这个状态下接收 tag 信息，并将对应 entry 的 data 置零。

(2) S_ADD 状态用于存储 tag 信息，在接收完成后，在条件(3)下跳转回 S_IDLE 状态，在条件(4)下跳转到 S_OUT 状态。

(3) S_OUT 状态用于将 lsu_rsp 发送给 lsu2wb，当传输完成后，重新进入 S_IDLE 状态。不同的情况可以见下图的解释：

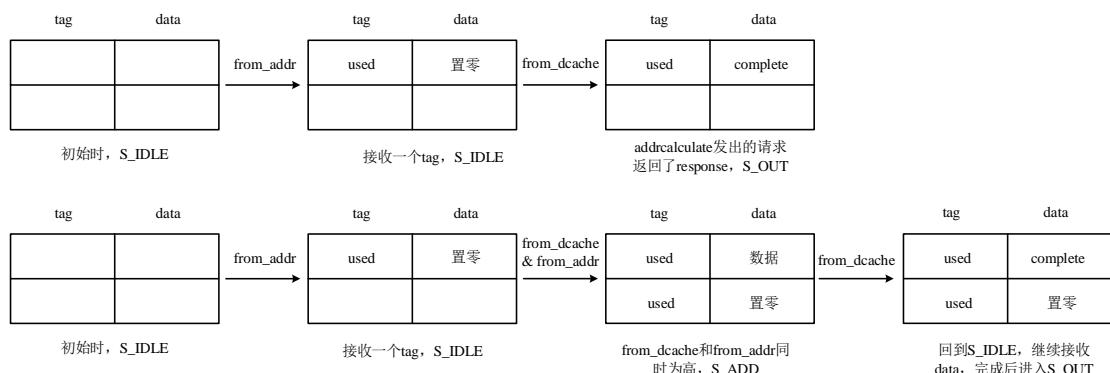


图 4-29 mshr 在不同状态下 tag 和 data 的操作

上面展示了主要的两种情况。可以看出，当 DCache/Sharemem 的响应和 addrcaculate 的 tag 信息同时到来时，它们要写入的不是同一个 entry，S_ADD 这个状态的设置就是为了多一个时钟周期来存储 tag 信息。

4.2.13.3.3 shiftboard

ShiftBoard 内有一个元素个数为 depth 的向量，每个元素都是 1bit 的寄存器，在接收一组 lsu_req 后左移一位；在返回一组 lsu_rsp 后右移一位。左移时最低位补 1，右移时最高位补 0，如下图所示。所以，当寄存器的最低位是 0 时，说明所有访存 require 都得到了 response，这用于 fence 指令。

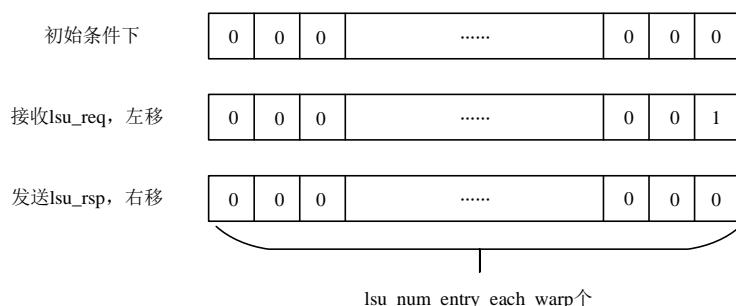


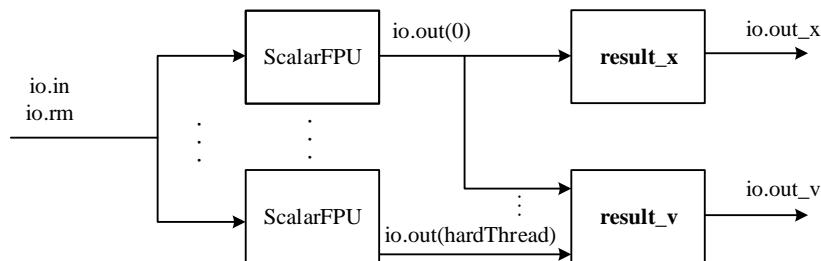
图 4-30 ShiftBoard 的工作原理

4.2.14 FPU

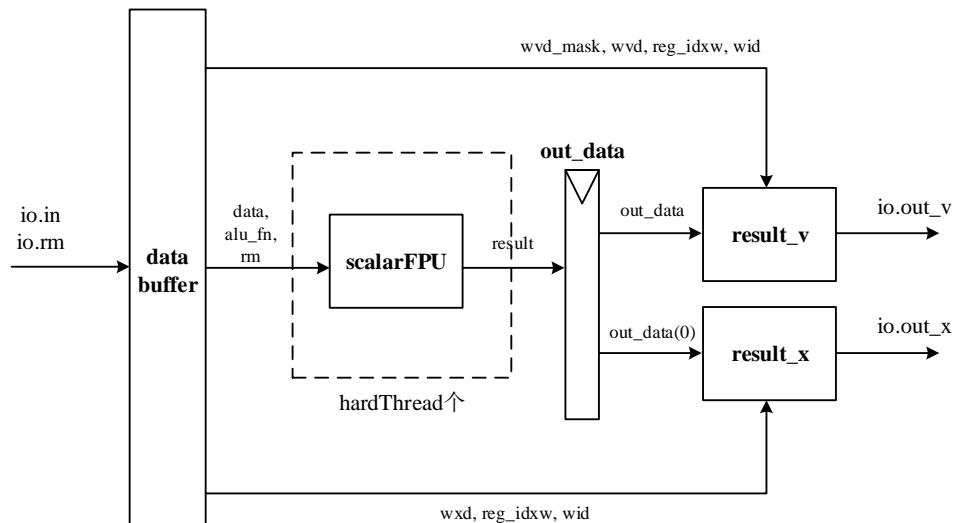
4.2.14.1 概述

FPU 是乘影 GPGPU 中的浮点数处理模块，标量和向量浮点运算都在这个模块进行。

FPU 的硬件数量暂未决定，目前 FPU 硬件数量(hardThread)为 NUM_THREAD(softThread) 的 $1/2^m$ 倍， $m=0, 1, 2, \dots$ 在 softThread 等于/不等于 hardThread 的条件下，FPU 的结构如下图所示。



(a)



(b)

图 4-31 (a) softThread 等于 hardThread (b) softThread 不等于 hardThread

标量 FPU (scalarFPU) 内部有 5 个子模块：用于乘法、加法和乘加的 fma；用于符号注入和分类的 fpmv；用于浮点数比较的 fcmp；用于浮点转整型的 fp2int；用于整型转浮点的 int2fp。其结构框图如下图所示。

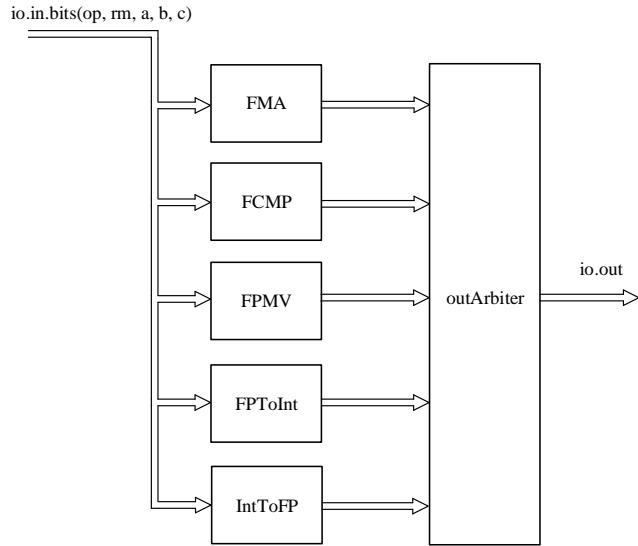


图 4-32 scalarFPU 硬件框图

4.2.14.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟
rst_n	IN	1	复位信号, 低电平有效, 对内部寄存器进行复位。
流水线请求接口			
in_valid_i	IN	1	请求有效信号
in_ready_o	OUT	1	sfu 内部 ready 信号
in_in1_i	IN	128	操作数 1, 位宽随 NUM_THREAD 变化
in_in2_i	IN	128	操作数 2, 位宽随 NUM_THREAD 变化
in_in3_i	IN	128	操作数 3, 位宽随 NUM_THREAD 变化
in_mask_i	IN	4	线程掩码, 位宽随 NUM_THREAD 变化
in_wid_i	IN	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
in_reverse_i	IN	2	操作数调换信号, 在高电平时, 将 in2 赋给 a, 将 in1 赋给 b
in_alu_fn_i	IN	6	操作码
in_reg_idxw_i	IN	8	目的寄存器地址
in_wvd_i	IN	1	目的寄存器是向量寄存器
in_wxd_i	IN	2	目的寄存器是标量寄存器

舍入模式输入信号			
in_rm_i	IN	3	舍入模式控制信号
流水线应答接口			
out_x_valid_o	OUT	1	标量指令访存指令响应有效信号
out_x_ready_i	IN	1	表示标量写回端口是否就绪
out_x_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_x_wxd_o	OUT	1	写回目的寄存器是标量寄存器
out_x_reg_idxw_o	OUT	8	标量寄存器地址
out_x_wb_wxd_rd_o	OUT	32	需要写回的数据
out_v_valid_o	OUT	1	向量指令访存指令响应有效信号
out_v_ready_i	IN	1	表示向量写回端口是否就绪
out_v_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_v_wvd_o	OUT	1	写回目的寄存器是向量寄存器
out_v_reg_idxw_o	OUT	8	向量寄存器地址
out_v_wvd_mask_o	OUT	4	线程掩码, 位宽随 NUM_THREAD 变化
out_v_wb_wxd_rd_o	OUT	128	需要写回的数据, 位宽随 NUM_THREAD 变化

4.2.14.3 设计原理

4.2.14.3.1 FMA

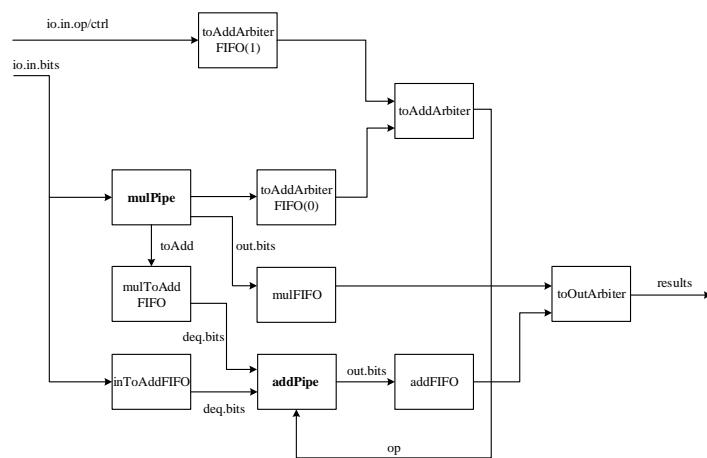


图 4-33 FMA 的结构框图

FMA 模块的框图如上图所示，其框图说明如下：

(a) FMA 用于 FPU 的乘法、加法和乘加运算，通过一系列的握手信号来决定浮点运算单元是否进行计算，同时，通过仲裁器来决定输出哪个模块的数据。

(b) 乘法需要 3 个 cycle 的时间，加法需要 2 个 cycle 的时间，所以，乘加运算一共需

要 5 个 cycle。

(c) **mulPipe** 是浮点乘法单元，计算出的结果可能会直接输出，也可能会传递给浮点加法单元进行乘加运算。

(d) **addPipe** 是浮点加法单元，它接受来自外部 IO 和 **mulPipe** 的数据，并根据操作码 **op** 来判断操作数来自哪里。

4.2.14.3.1.1 mulPipe

FMA 的 **mulPipe** 采用了浮点乘法器的一般结构，如下图所示。该浮点乘法器主要由一个处理尾数乘法的定点乘法器组成，以及处理指数和特殊值（ ± 0 、 $\pm \infty$ 、 NaN ）的电路组成。

该算法的流程如下：

- (a) 计算两带偏差的指数之和，并从中减去偏差，以计算出暂定指数。
- (b) 计算尾数乘法，对于两个在区间 [1,2] 上的无符号有效数的乘积将落在 [1,4) 上。因此，可能需要将结果右移 1 位（同时调整指数）以完成标准化。
- (c) 舍入后再进行一次标准化一位与指数调整。

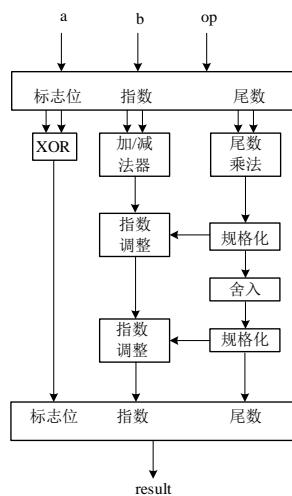


图 4-34 mulPipe 的计算流程

mulPipe 的硬件结构如下图所示。

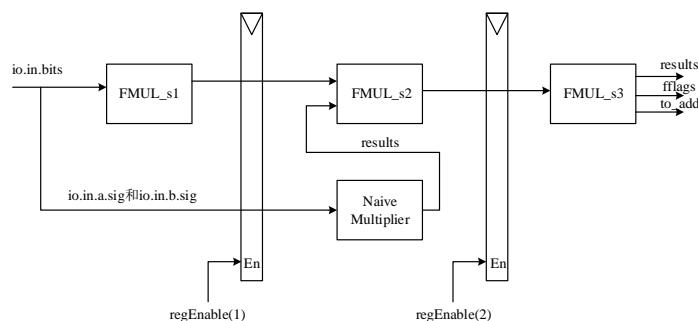


图 4-35 mulPipe 的硬件结构

- (a) 该浮点数乘法内部采用了三级流水（计算结果需要三个 cycle）。
- (b) **FMUL_s1** 用于做指数加减法，计算出暂定指数，预判断指数是否溢出，并分析一些特殊情况。
- (c) **NaiveMultiplier** 用于计算尾数乘法。
- (d) **FMUL_s2** 用于收集从 **FMUL_s1** 和 **NaiveMultiplier** 的信号，并将其传递给下一级流水

线。

(e) FMUL_s3 将尾数计算结果规格化后，调整指数，随后再进行舍入操作，将舍入结果规格化后进行输出。

4.2.14.3.1.2 addPipe

addPipe 的计算流程如图所示，其步骤为：(1) 求阶差；(2) 对阶；(3) 尾数相加；(4) 结果规格化并判断溢出；(5) 舍入；(6) 再次规格化。

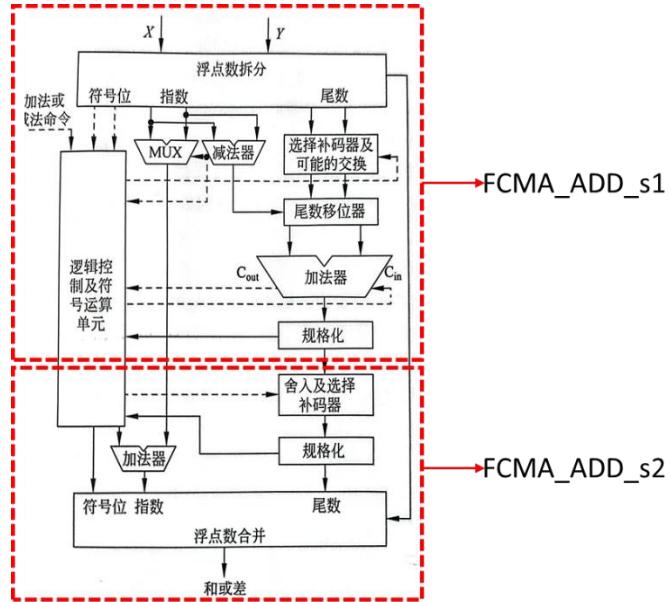


图 4-36 addPipe 的计算流程

addPipe 的硬件实现框图如下图所示。

- (a) 完成该浮点数加法器需要两个时钟周期。
- (b) addPipe 模块对两个浮点操作数进行加法/减法运算，由于指令中有乘加操作，因此先判断操作数 b 是否为乘法结果，若为乘加操作，将 b_inter_valid 置高，同时输入乘法操作得到的异常标志。
- (c) FCMA_ADD_s1 模块的作用是完成浮点数拆分、尾数预移位、加法和后移位的操作，该模块为了加速浮点数加法，采用了双路径的设计思路，同时还需要考虑特殊情况。
- (d) FCMA_ADD_s2 模块的作用是完成浮点数舍入、规格化和浮点数合并的操作，也需要定义特殊情况下结果输出。

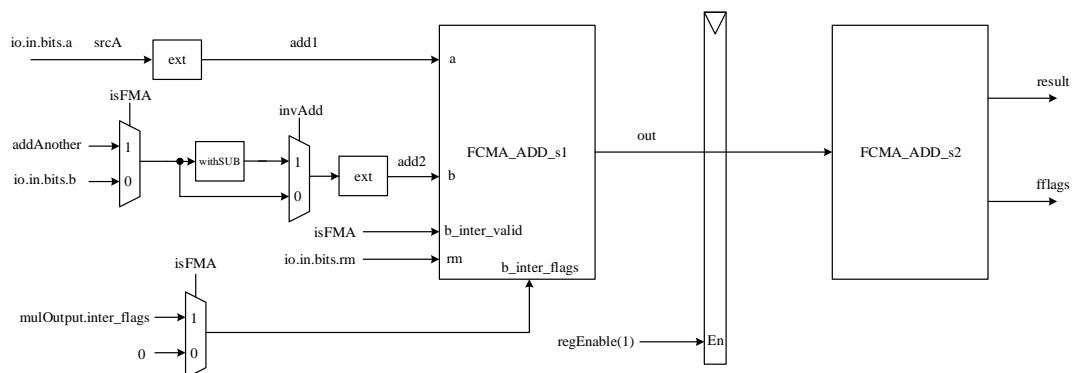


图 4-37 addPipe 的硬件框图

其中，FCMA_ADD_S1 分成了双路径进行计算，根据两操作数的指数差将预移位和后移

位转发到不同的路径上，当指数差 $\text{expDiff} \geq 2$ 时为远路径 Farpath，当指数差 $\text{expDiff} < 2$ 时为近路径 Nearpath。这样区分是为了减小移位的次数。

4.2.14.3.2 FPMV

FPMV 模块中包含了符号注入和分类等共 4 条指令，主要是通过输入的操作码 op 选择不同的输出（符号位改变后的操作数 a 或者操作数 a 的类别）。其硬件实现框图如下。

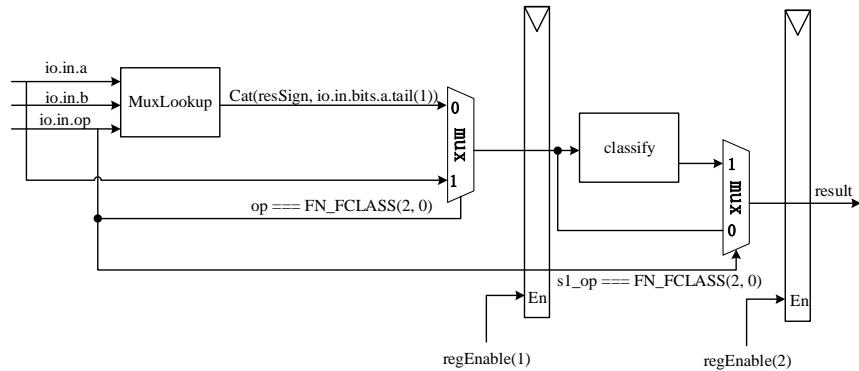


图 4-38 FPMV 的硬件框图

(a) MuxLookup 模块为一个查找表，其根据输入的操作码 op 的后三位检索输出操作数 a 的新符号位，若为 110，输出操作数 b 的符号；若为 101，输出操作数 b 的符号取反；若为 100，输出操作数 a 的符号与操作数 b 的符号的异或。

(b) classify 模块的功能为分类，其根据输入的浮点数类别输出相应结果，如下表所示。

$x[rd]$ 位	含义
0	$f[rsI]$ 为 $-\infty$ 。
1	$f[rsI]$ 是负规格化数。
2	$f[rsI]$ 是负的非规格化数。
3	$f[rsI]$ 是 -0 。
4	$f[rsI]$ 是 $+0$ 。
5	$f[rsI]$ 是正的非规格化数。
6	$f[rsI]$ 是正的规格化数。
7	$f[rsI]$ 为 $+\infty$ 。
8	$f[rsI]$ 是信号(signaling)NaN。
9	$f[rsI]$ 是一个安静(quiet)NaN。

4.2.14.3.3 FCMP

FCMP 模块为浮点数比较模块，其中包含了最大值、最小值、相等、不等、小于和小于等于这 6 条比较指令，需要注意的是可能存在输入操作数为 invalid 情况，此时输出结果为 0。其硬件框图如下图所示。

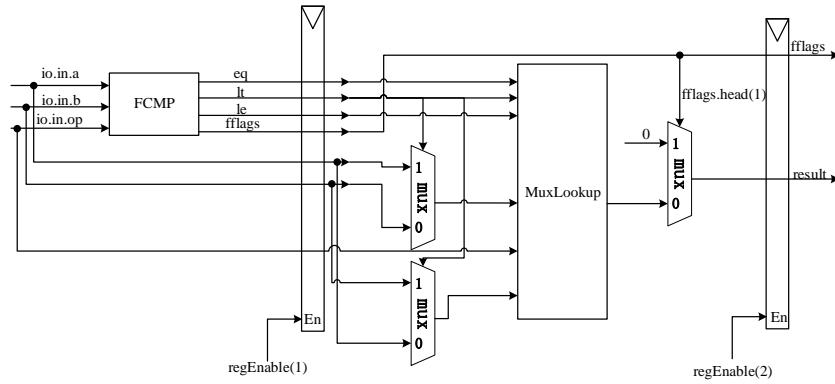


图 4-39 FCMP 的硬件框图

(a) FCMP 将输入的浮点数 a 和 b 进行划分，通过 decode 模块分析操作数是否为特殊情况（0、NaN 和 SNaN）并考虑将 fflags 的最高位置高，同时该模块还比较了 a 和 b 的大小，并输出 eq（相等）、lt（小于）和 le（小于等于）三个结果。

(b) MuxLookup 模块是一个查找表，其根据输入的 op 检索输出相应的比较结果，当操作数无效时输出结果为 0。

4.2.14.3.4 FPToInt

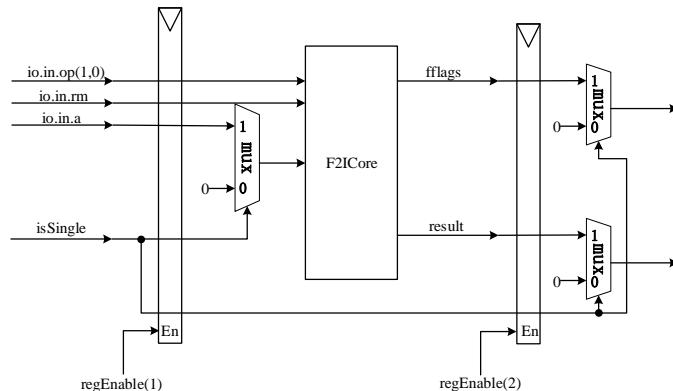


图 4-40 FPToInt 的硬件框图

(a) 信号 isSingle 是通过操作码 op 判断的输入浮点数是否为单精度（实际设计中也未考虑双精度指令）

(b) F2ICore 模块为单精度浮点数转整型和无符号整型的具体算法模块，其通过操作码 op 的最后两位判断转化成有符号还是无符号整型，再通过舍入模式进行舍入。

二进制浮点数的标准形式为 $(-1)^s \times 1.f \times 2^{e-b}$ ，其中 s 为符号位，指数位 e 需要减去偏移量 b ($b=2^{\text{len}(e)-1}-1$)，而 32bit 二进制无符号数的上限为 $2^{32}-1$ ，因此指数位 e 最大只能为 $b+31$ 。接下来需要对指数位 e 分情况讨论，若 $e \geq b+pc$ (pc 为尾数 f 的位宽)，意味着尾数 f 可以在都左移至小数点左边的基础上再左移 $e-b-pc$ 位，需要考虑溢出情况；若 $e < b+pc$ ，说明尾数 f 只有部分能左移至小数点左边，有 $b+pc-e$ 位在小数点的右边，由于此时不是整数，因此需要同时考虑舍入和溢出的情况。

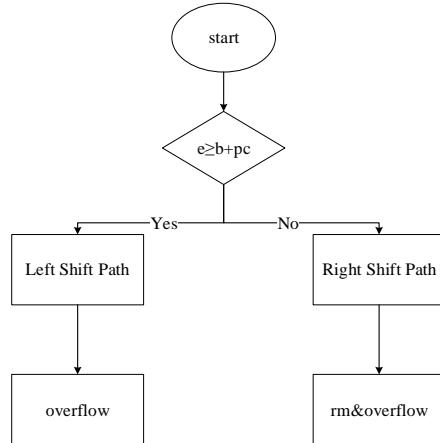


图 4-41 浮点转整型的计算流程

4.2.14.3.5 IntToFP

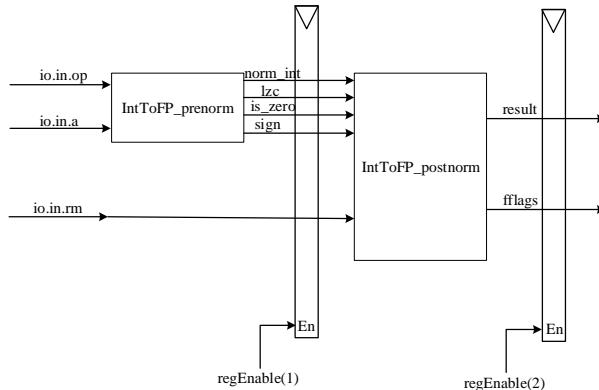


图 4-42 IntToFP 的硬件结构

- (a) IntToFP_prenorm 模块根据输入的 op 判断二进制数是否有符号，并且分情况对二进制数进行前导零预测，然后移位使得最高位为 1，去掉最高位即为浮点数的尾数。
- (b) IntToFP_postnorm 模块对尾数进行舍入，并根据是否进位得到浮点数的指数，再和符号位一起拼接成浮点数格式。

二进制转浮点数首先需要根据二进制的符号分情况对其进行前导零预测，目的是找到二进制中最高位的“1”的位置并将其左移到最高位，从而得到浮点数的尾数部分。然后再对多余尾数位宽的部分进行舍入，根据舍入的结果得到浮点数的指数部分。由于乘影目前没有转化成 64 位整型的指令，代码中的 long_int 恒为 0。

4.2.15 SFU

4.2.15.1 概述

目前 SFU 只支持了 RV 定义的整数除法、浮点除法和浮点平方根功能。

SFU 中的运算单元少于 NUM_THREAD，需要较长的周期数来完成。

其硬件结构如下图所示。其中包括：int_div 用于整数除法；float_div_sqrt 用于浮点除法/平方根；databuffer 是一个深度为 1 的队列，用于寄存输入数据；arbiter 用于仲裁是整

数还是浮点的输出数据；`out_data` 用于寄存输出数据，等待所有线程都完成计算后将其返回给流水线。

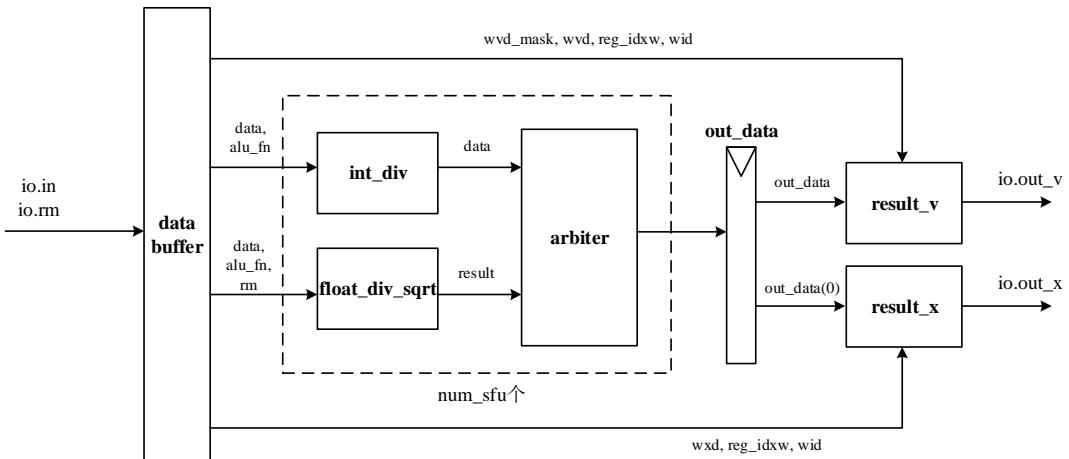


图 4-43 SFU 的硬件结构

4.2.15.2 信号描述

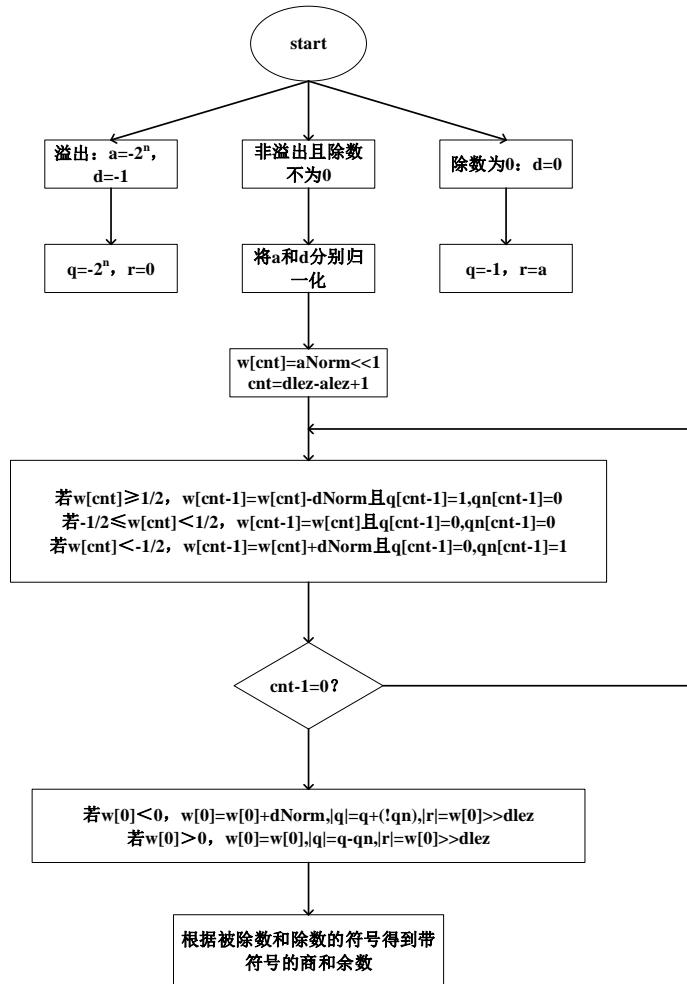
名称	类型	位宽	描述
时钟及复位信号			
<code>clk</code>	IN	1	系统时钟
<code>rst_n</code>	IN	1	复位信号, 低电平有效, 对内部寄存器进行复位。
流水线请求接口			
<code>in_valid_i</code>	IN	1	请求有效信号
<code>in_ready_o</code>	OUT	1	sfu 内部 ready 信号
<code>in_in1_i</code>	IN	128	操作数 1, 位宽随 NUM_THREAD 变化
<code>in_in2_i</code>	IN	128	操作数 2, 位宽随 NUM_THREAD 变化
<code>in_in3_i</code>	IN	128	操作数 3, 位宽随 NUM_THREAD 变化
<code>in_mask_i</code>	IN	4	线程掩码, 位宽随 NUM_THREAD 变化
<code>in_wid_i</code>	IN	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
<code>in_fp_i</code>	IN	1	是否是浮点操作
<code>in_reverse_i</code>	IN	2	操作数调换信号, 在高电平时, 将 <code>in2</code> 赋给 <code>a</code> , 将 <code>in1</code> 赋给 <code>b</code>
<code>in_isvec_i</code>	IN	1	是否向量操作
<code>in_alu_fn_i</code>	IN	6	操作码
<code>in_reg_idxw_i</code>	IN	8	目的寄存器地址

in_wvd_i	IN	1	目的寄存器是向量寄存器
in_wxd_i	IN	2	目的寄存器是标量寄存器
舍入模式输入信号			
in_rm_i	IN	3	舍入模式控制信号
流水线应答接口			
out_x_valid_o	OUT	1	标量指令访存指令响应有效信号
out_x_ready_i	IN	1	表示标量写回端口是否就绪
out_x_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_x_wxd_o	OUT	1	写回目的寄存器是标量寄存器
out_x_reg_idxw_o	OUT	8	标量寄存器地址
out_x_wb_wxd_rd_o	OUT	32	需要写回的数据
out_v_valid_o	OUT	1	向量指令访存指令响应有效信号
out_v_ready_i	IN	1	表示向量写回端口是否就绪
out_v_warp_id_o	OUT	3	warp id, 当前 warp 在 workgroup 里面的位置, 位宽随 workgroup 内的最大 warp 数量变化
out_v_wvd_o	OUT	1	写回目的寄存器是向量寄存器
out_v_reg_idxw_o	OUT	8	向量寄存器地址
out_v_wvd_mask_o	OUT	4	线程掩码, 位宽随 NUM_THREAD 变化
out_v_wb_wxd_rd_o	OUT	128	需要写回的数据, 位宽随 NUM_THREAD 变化

4.2.15.3 设计原理

4.2.15.3.1 int_div

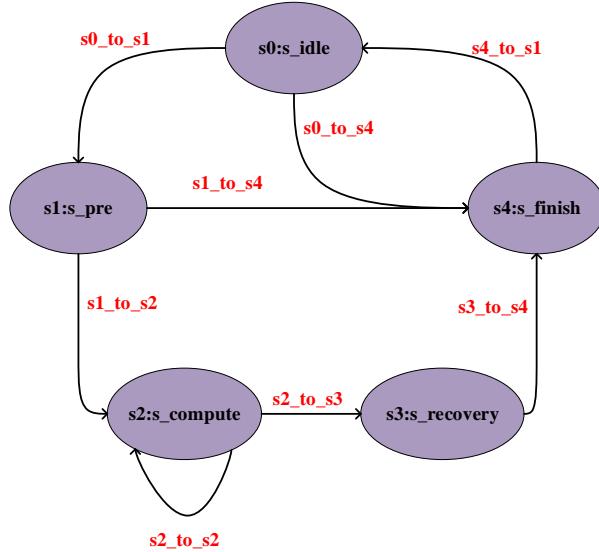
整数除法器的计算流程如下图所示。



注: aNorm 和 dNorm 为被除数 a 和除数 d 归一化后的结果, 此操作先将 a 和 d 取绝对值, 然后向左移位直至最高位为 1 (移位位数分别为 alez 和 dlez), 在移位后的 a 的最高位添加两 bit 的 0, 在移位后的 d 的最高位和最低位分别添加 1 bit 的 0

图 4-44 整数除法计算的流程

下图为整数除法器工作时的状态转移图, s_{idle} 为初始状态, 给寄存器赋 0 作为初始值, 若输入的被除数以及除数为特殊值 (溢出以及除数为 0) 直接到达 s_{finish} 状态, 同时输出商和余数, 其他情况则进入 s_{pre} 状态; s_{pre} 主要是对被除数和除数进行归一化, 若判断出被除数小于除数, 同样也直接到达 s_{finish} 状态并输出商和余数; $s_{compute}$ 状态中通过基 2 SRT 算法求得商; $s_{recovery}$ 状态实现了余数的恢复; 最后通过 s_{finish} 状态输出商和余数。



附各状态转移条件:

```

s0_to_s1: io.in.fire() && (!overflow || !divByZero)
s0_to_s4: io.in.fire() && (overflow || !divByZero)
s1_to_s2: unsignedAReg >= unsignedDReg
s1_to_s4: unsignedAReg < unsignedDReg
s1_to_s2: cnt_next /= 0.U
s2_to_s3: cnt_next === 0.U
s3_to_s4: true
s4_to_s1: io.out.fire()

```

图 4-45 int_div 内部

下面介绍整数除法中用到的基 2 SRT 算法:

基 2 SRT 算法是在数字递归算法基础公式上改善得到的，使用数字递归算法（Digit Recurrence Algorithms）进行除法操作时迭代 n 次，每次迭代中产生基 r 的商，其中商的最高位先产生。经过 j+1 次迭代后，商表示如下：

$$q_{j+1} = \sum_{i=1}^j q_i r^{-i} \quad (5-4)$$

经过 n 次迭代后除法完成，产生了 n 个商数，商 q 表示为：

$$q = q_n = \sum_{i=1}^n q_i r^{-i} \quad (5-5)$$

最终 q 的误差需小于 ulp，所以：

$$0 \leq \varepsilon_q = \left| \frac{x}{d} - q_n \right| < r^{-n} = ulp \quad (5-6)$$

在第 j+1 次迭代中，每一步中产生的误差为：

$$\varepsilon_{j+1} = \left| \frac{x}{d} - q_{j+1} \right| < r^{-(j+1)} \quad (5-7)$$

重新组合上式，两式各乘以 d 和 r 的 j+1 次得：

$$r^{(j+1)} |x - dq_{j+1}| < d \quad (5-8)$$

以上左式定义一个新的中间变量 w[j+1] 如下：

$$w_{j+1} = r^{(j+1)} |x - dq_{j+1}| \quad (5-9)$$

w[j+1] 为部分余数，其递归过程如下式所示：

$$w_{j+1} = rw_j - dq_{j+1}, \text{ where } w_0 = x \quad (5-10)$$

此处部分余数 w[j+1] 可以表示为：

$$-d < w[j+1] < d \quad (5-11)$$

因为 w[j+1]=r*w[j]-d*q[j+1]，这表示在第 j+1 次迭代后得到的 q[j+1] 需使 w[j+1] 满足以上

条件。下面是 $w[j+1]$ 的迭代流程图：

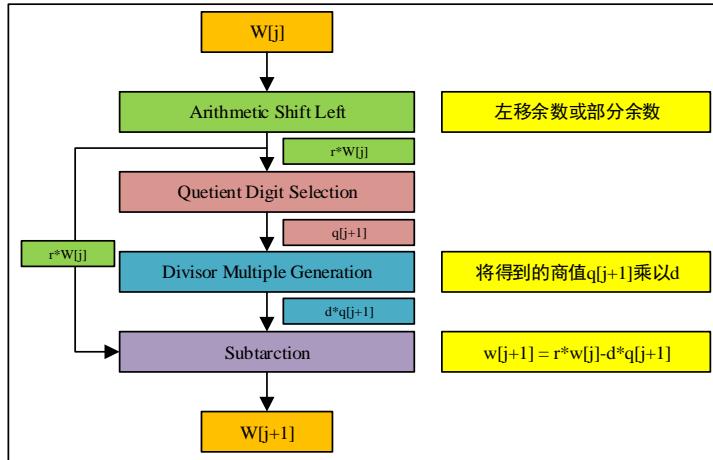


图 4-46 $w[j+1]$ 的迭代流程图

$w[j+1]$ 迭代框图是根据输入 $w[j]$, r 和 d 联合计算而来, 初始的 $w[j]=x$, 即被除数。此即为商数选择法 (Quotient Digit Selection, QDS)。

二进制恢复余数法的 QDS 函数:

$$q_{i+1} = \begin{cases} 0, & \text{if } 2w[j] < d \\ 1, & \text{if } d \leq 2w[j] < 2d \end{cases} \quad (5-12)$$

二进制非恢复余数法的 QDS 函数:

$$q_{i+1} = \begin{cases} -1, & \text{if } 2w[j] < 0 \\ 1, & \text{if } 2w[j] \geq 0 \end{cases} \quad (5-13)$$

而 SRT 算法旨在加速非恢复二进制除法, 在商数选择集中引入了 0, 且将 QDS 函数修改如下:

$$q_{i+1} = \begin{cases} -1, & \text{if } 2w[j] < -d \\ 0, & \text{if } -d \leq 2w[j] < d \\ 1, & \text{if } 2w[j] \geq d \end{cases} \quad (5-14)$$

基 2 SRT 将 0 引入到商集中, 部分迭代则可只需要移位操作, 减少了除法模块的平均延时。但它与非恢复余数法的问题依旧是 $2w[j]$ 与 $-d$ 和 $+d$ 的比较需要全精度才能得出 q 值。所以将除数 d 归一化小数表示至区间 $[1/2, 1]$, 引入新的分界点 $-1/2$ 和 $1/2$ 来替代 $-d$ 和 $+d$, 下式可说明 $-d$ 与 $+d$ 的小数取值:

$$-d \leq -\frac{1}{2} \leq 2w[j] < \frac{1}{2} \leq d \quad (5-15)$$

所以 QDS 将更新如下:

$$q_{i+1} = \begin{cases} -1, & \text{if } 2w[j] < -\frac{1}{2} \\ 0, & \text{if } -\frac{1}{2} \leq 2w[j] < \frac{1}{2} \\ 1, & \text{if } 2w[j] \geq \frac{1}{2} \end{cases} \quad (5-16)$$

换成列表形式:

$[d_i, d_{i+1})$	$2w[j]$	q_{i+1}
$[\frac{1}{2}, 1)$	$[\frac{1}{2}, 1)$	1
	$[-\frac{1}{2}, \frac{1}{2})$	0

	$(-1, -\frac{1}{2})$	-1
--	----------------------	----

d 属于区间 $[\frac{1}{2}, 1)$; 部分商 $w[j+1]$ 被限制在 $[-\frac{1}{2}, \frac{1}{2}]$ 区间内, $2w[j]$ 被归一化且其二进制补码表示如下:

$$2w[j] = u_0 u_1 u_2 \cdots u_{-(n+1)} \quad (5-17)$$

其中 u_0 为符号位。因此, 在 $\{-1, 0, 1\}$ 三个可能的值中选取合适的商 q 值。

q 的选择有三种情况:

1. 当 $2w[j] \geq 1/2$, 则 q 值为 1, 即最高两比特为 0.1;
2. 当 $2w[j] < -1/2$, 则 q 值为 -1, 即最高两比特为 1.0;
3. 其他情况, q 值为 0;

其具体计算过程已经体现在计算流程图中, 下面举例说明。

设被除数a=58, 除数d=6, 则 $58/6=9\dots4$										
init	a=58	0	0	1	1	1	0	1	0	
	d=6	0	0	0	0	0	1	1	0	
Normalization	aNorm	0	0	1	1	1	0	1	0	0
	dNorm	0	1	1	0	0	0	0	0	0
	dNegNorm	1	0	1	0	0	0	0	0	0
step0	w[4]	0	0	1	1	1	0	1	0	0
	2*w[4]	0	1	1	1	0	1	0	0	0
	+(-d)	1	0	1	0	0	0	0	0	0
step1	w[3]	0	0	0	1	0	1	0	0	0
	2*w[3]	0	0	1	0	1	0	0	0	0
	+0	0	0	0	0	0	0	0	0	0
step2	w[2]	0	0	1	0	1	0	0	0	0
	2*w[2]	0	1	0	1	0	0	0	0	0
	+(-d)	1	0	1	0	0	0	0	0	0
step3	w[1]	1	1	1	1	0	0	0	0	0
	2*w[1]	1	1	1	0	0	0	0	0	0
	+0	0	0	0	0	0	0	0	0	0
step4	w[0]	1	1	0	0	0	0	0	0	<0
	+d	1	1	0	0	0	0	0	0	
	w[0]	1	0	0	0	0	0	0	0	
step5	r =w[0]>>5	0	0	0	0	0	1	0	0	r =4
	q	1	0	1	0					
	qn	0	0	0	0					
	q =q+(!qn)	1	0	0	1					q =9
										因为被除数和除数都为正数, 所以商为9, 余数为4, 符合预期

4.2.15.3.2 float_div_sqrt

这部分采用了开源项目 fpnew 的浮点除法/开根号模块。参考网址：
[pulp-platform/fpu_div_sqrt_mvp: \[UNRELEASED\] FP div/sqrt unit for transprecision \(github.com\)](https://github.com/pulp-platform/fpu_div_sqrt_mvp)

4.2.16 写回单元

这一部分包括用于数据写回的模块 `writeback`（连接执行单元和 `operand_collector`），以及分支写回的模块 `branch_back`（连接 `simt_stack/alu` 和 `warp_sche`）。

这一部分的硬件结构比较简单，这里不做介绍。

5 内存系统

5.1 概述

下图展示了 Cluster 内部的结构（目前在代码中 `NUM_CLUSTER=1`，所以所有的 SM 共同访问同一块 L2 Cache）。每个 SM 有单独的 L1 内存子系统，包括共享存储（Shared Memory）、指令缓存（ICache）和数据缓存（DCache）。其中，ICache 通过取指级访问，其它存储器均通过 LSU 进行访问。

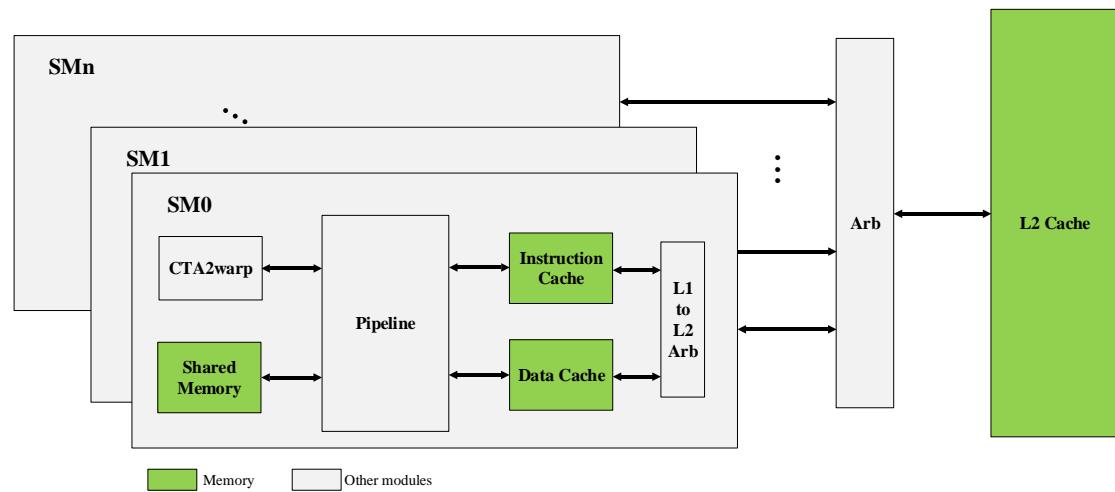


图 5-1 Ventus 内部的存储系统（`NUM_CLUSTER=1`）

5.2 详细设计

5.2.1 Icache

5.2.1.1 概述

Icache 的特性如下：

- (1) 2 路(way)组相联（具体数量可配置）；
- (2) 每次 L1-L2 之间的数据读写的最大宽度均为一个缓存行的宽度；
- (3) 替换策略支持 LRU 替换策略；
- (4) 支持对正在进行的请求进行无效化（flush）；
- (5) 可以同时存在多个个未完成的 L2 访问请求（具体数量可配置）；
- (6) 支持相同缓存行的 L2 访问请求合并（具体数量可配置）。

5.2.1.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟
rst_n	IN	1	全局复位，低有效， rst_n 触发后将在至少 一个时钟周期后对核进 行复位
invalid_i	IN	1	全局无效化使能信号， 使 icache 保存的所有 cacheline 的 tag 无效
core_req_valid_i	IN	1	握手信号，为高电平时 表明 warp 调度器向 icache 发送取指请求。 无需保持为高电平，直 到 ready 信号为高电平
core_req_addr_i	IN	32	取指请求地址
core_req_mask_i	IN	NUM_FETCH	指令有效掩码
core_req_wid_i	IN	DEPTH_WARP	取指请求的 warp id
flush_pipe_valid_i	IN	1	握手信号，为高电平时 表示 icache miss 或 ibuffer full，需要将后 续取指响应无效化。无

			需保持为高电平，直到 ready 信号为高电平
flush_pipe_wid_i	IN	DEPTH_WARP	发生 icache miss 或 ibuffer full 的 warp id
<hr/>			
core_rsp_valid_o	OUT	1	握手信号，为高电平时表示 icache 发起取指响应。无需保持为高电平，直到 ready 信号为高电平。
core_rsp_addr_o	OUT	32	取指响应地址
core_rsp_data_o	OUT	NUM_FETCH*32	取指响应指令
core_rsp_mask_o	OUT	NUM_FETCH	指令有效掩码
core_rsp_wid_o	OUT	DEPTH_WARP	取指响应的 warp id
core_rsp_status_o	OUT	1	1 表示取指状态为 miss, 0 表示取指状态为 hit
<hr/>			
mem_rsp_valid_i	IN	1	握手信号，为高电平时表示 l2cache 发起读响应。需要保持为高电平，直到 ready 信号为高电平
mem_rsp_ready_o	OUT	1	握手信号
mem_rsp_d_source_i	IN	DEPTH_WARP	请求发送主机辨识符
mem_rsp_d_addr_i	IN	32	读响应地址
mem_rsp_d_data_i	IN	DCACHE_BLOC_KWORDS*32	读响应数据，即指令
<hr/>			
mem_req_valid_o	OUT	1	握手信号，为高电平时表示发起读缺失请求。需要保持为高电平，直到 ready 信号为高电平
mem_req_ready_i	IN	1	握手信号
mem_req_a_source_o	OUT	DEPTH_WARP	请求发送主机辨识符
mem_req_a_addr_o	OUT	32	读缺失请求地址

5.2.1.3 设计原理

icache 由 3 级流水构成，如下图所示。

(1) 流水级 0 接收取指请求或二级缓存响应，接收取指请求时向标签存储阵列发起 tag 检查请求；接收二级缓存响应时，向 mshr 发起读请求获取地址等信息。

(2) 流水级 1 得到取指请求的 tag 检查结果, 若 tag hit 则向数据存储阵列发起读请求, 获取指令; 若 tag miss, 则向 mshr 发起写请求, 记录该次缺失。此外, 流水级 1 得到二级缓存响应对 mshr 发起的读请求的结果, 同时对标签存储阵列和数据存储阵列发起写请求, 同时清空 mshr 中记录该次缺失的 entry。

(3) 流水级 2 将得到取指响应, 同时 mshr 会在二级缓存允许时向二级缓存发起读请求; 二级缓存响应的内容也在标签存储阵列和数据存储阵列中更新完毕。

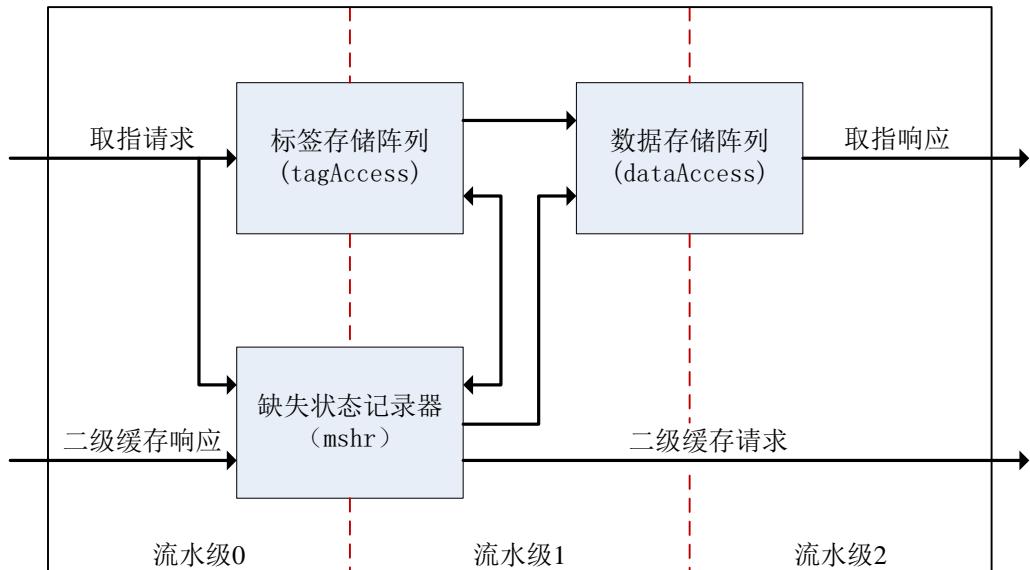


图 5-2 icache 结构框图

5.2.2 sharemem

5.2.2.1 概述

Sharemem 是一个直接映射的 memory, 其缓存行大小为 blocksize 个 word(通常情况下, blocksize=NUM_THREAD), 其内部包括 NSets(可配置, 目前是 128)个缓存行。所以, Sharemem 的总容量为 NSets*blocksize 个 word。

Sharemem 主要由用于处理板块冲突的 bankconfarb、SRAM (dataAccess) 组成以及用于读写的交叉开关构成。如下图所示。

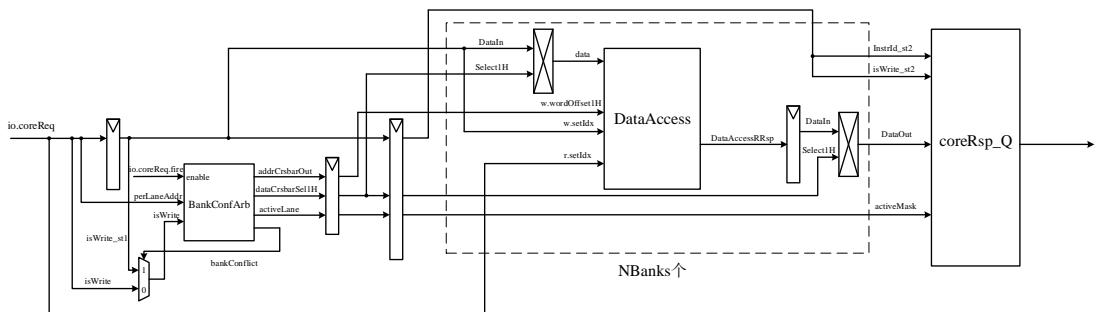


图 5-3 Sharemem 的硬件结构

Sharemem 的读写可能会产生板块冲突, 在发生板块冲突时, 由内部 bankconfarb 模块将单个请求拆分成多次请求。

5.2.2.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟
rst_n	IN	1	复位信号, 低电平有效, 对内部寄存器进行复位。
lsu 请求接口			
core_req_valid_i	IN	1	Sharemem 请求有效信号
core_req_ready_o	OUT	1	Sharemem 请求就绪信号
core_req_instrid_i	IN	3	来自 lsu mshr 内部的 id 信号
core_req_iswrite_i	IN	1	是否为写操作
core_req_setidx_i	IN	5	当前 Cacheline 的 index/setidx, (Sharemem 内部没有 Cacheline 的概念, 但是其请求格式与 Dcache 一致)
core_req_tag_i	IN	24	当前 Cacheline 的 tag
core_req_activemask_i	IN	4	线程掩码
core_req_blockoffset_i	IN	4	各线程在当前 Cacheline 的 blockoffset
core_req_wordoffset1h_i	IN	16	各线程在当前 Cacheline 的 blockoffset
core_req_data_i	IN	128	写入的数据 (用于 store 指令)
lsu 应答接口			
core_rsp_valid_o	OUT	1	Sharemem 响应有效信号
core_rsp_ready_i	IN	1	Sharemem 响应就绪信号
core_rsp_instrid_o	OUT	3	id 信号, 与 shared_req_instrid_o 对应
core_rsp_data_o	OUT	128	读出的数据 (用于 load 指令)
core_rsp_activemask_o	OUT	4	线程掩码

5.2.2.3 设计原理

Sharemem 内部 SRAM 的组织形式如下图所示。

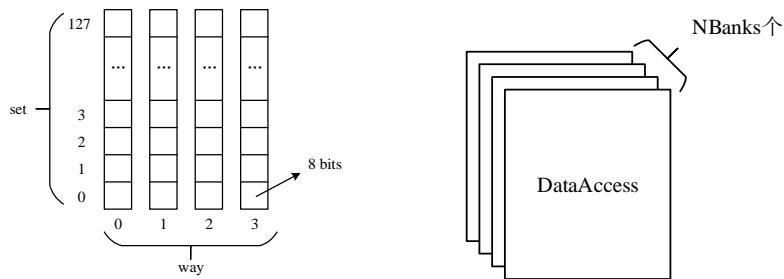


图 5-4 Sharemem 内部 SRAM 的组织形式

对 Sharemem 的读写请求可能会产生板块冲突，这是因为有多个线程访问到了同一个 dataAccess，在这种情况下，bankconfarb 会将来自 lsu 的请求拆分成多次请求，以保证每一个 dataAccess 在同一周期下最多只有一个线程对其进行读写。

Sharemem 的读写时序如下图所示，假定板块冲突只持续了一个周期。

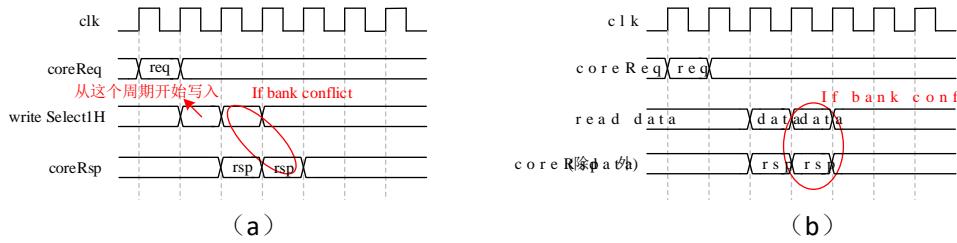


图 5-5 (a) 写时序; (b) 读时序

5.2.3 Dcache

5.2.3.1 概述

Dcache 采用 2 路组相联，缓存行的大小为 blockword (word)，总容量为 $32 \times 2 \times \text{blockword}$ (word)，一般的 blocksize = num_thread (目前 Dcache_blockword=lcache_blockword 相等)。

每次 LSU 访问的最大返回宽度为 blocksize；每次 L1-L2 之间数据读写的最大宽度均为 blockword (一个缓存行)。

写策略为写回-写不分配；替换策略支持 LRU 替换策略。

支持对整个数据高速缓存的无效和清除操作，暂不支持对单条缓存行的无效和清除操作。

支持相同缓存行的 L2 访问请求合并，目前合并数为 2。

目前缓存系统暂不支持原子指令，后续会进行更改。

Dcache 的流水线示意图如下图所示，浅蓝色图标表示队列，深蓝色图标表示寄存器，浅绿色图标表示 SRAM。常规读写命中需要 3 个时钟周期返回 response。

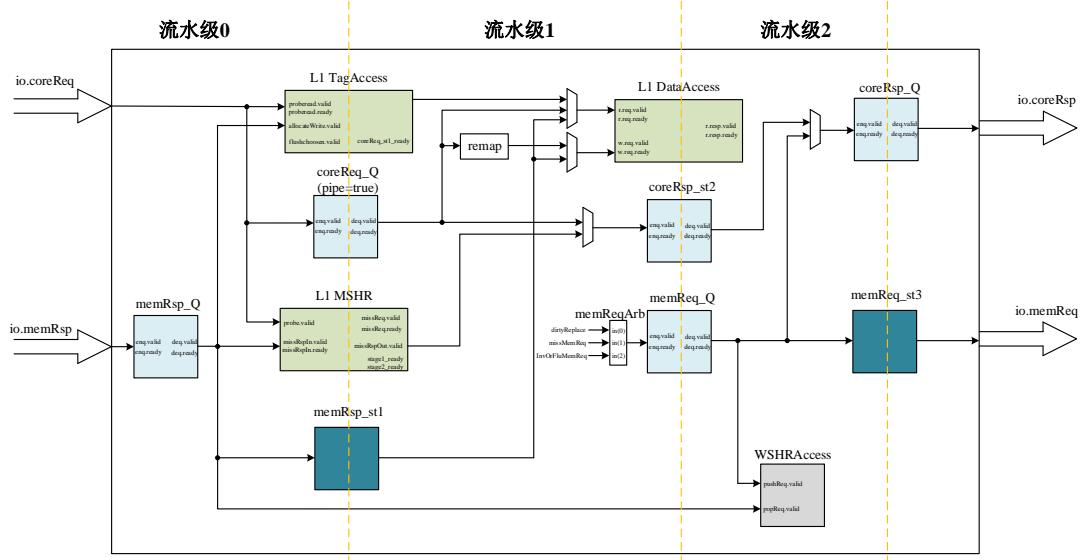


图 5-6 Dcache 的流水线示意图

5.2.3.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
clk	IN	1	系统时钟
rst_n	IN	1	复位信号, 低电平有效, 对内部寄存器进行复位。
Dcache 请求接口			
core_req_valid_i	IN	1	lsu 请求有效信号
core_req_ready_o	OUT	1	Dcache 就绪信号
core_req_instrid_i	IN	3	来自 lsu mshr 内部的 id 信号
core_req_setidx_i	IN	5	当前 Cacheline 的 index/setidx
core_req_tag_i	IN	24	当前 Cacheline 的 tag
core_req_activemask_i	IN	4	线程掩码
core_req_blockoffset_i	IN	4	各线程在当前 Cacheline 的 blockoffset
core_req_wordoffset1h_i	IN	16	各线程的字节使能
core_req_data_i	IN	128	写入的数据 (用于 store 指令)
core_req_opcode_i	IN	3	访问 Dcache 的类型
core_req_param_i	IN	4	访问 Dcache 类型的补充信号
Dcache 响应接口			
core_rsp_valid_i	OUT	1	Dcache 响应有效信号
core_rsp_ready_o	IN	1	lsu 就绪信号
core_rsp_instrid_i	OUT	3	id 信号, 与 dcache_req_instrid_o 对应
core_rsp_data_i	OUT	128	读出的数据 (用于 load 指令)
core_rsp_activemask_i	OUT	4	线程掩码
L2 Cache 请求接口			
mem_req_valid_o	OUT	1	Dcache 请求有效信号
mem_req_ready_i	IN	1	L2 Cache 就绪信号
mem_req_a_opcode_o	OUT	3	请求操作码
mem_req_a_param_o	OUT	3	请求操作码的补充信息
mem_req_a_source_o	OUT	10	id 信息, 来自请求操作类型、mshr/wshr 以及 setIdx
mem_req_a_addr_o	OUT	32	Cacheline 的起始地址
mem_req_a_data_o	OUT	64	写入的数据, 位宽随 Cacheline 的大小变化
mem_req_a_mask_o	OUT	8	写数据的字节 mask, 位宽随 Cacheline 的大小变化
L2 Cache 响应接口			
mem_rsp_valid_i	IN	1	Sharemem 响应有效信号
mem_rsp_ready_o	OUT	1	Sharemem 响应就绪信号

mem_rsp_d_opcode_i	IN	3	响应操作码
mem_rsp_d_source_i	IN	128	id 信息，对应 mem_req_a_source_o
mem_rsp_d_addr_i	IN	32	Cacheline 的起始地址
mem_rsp_d_data_i	IN	64	读出的数据

5.2.3.3 设计原理

Dcache 内部 SRAM 的组织形式如下图所示，其内部由 blockwords 个 dataAccess 组成。

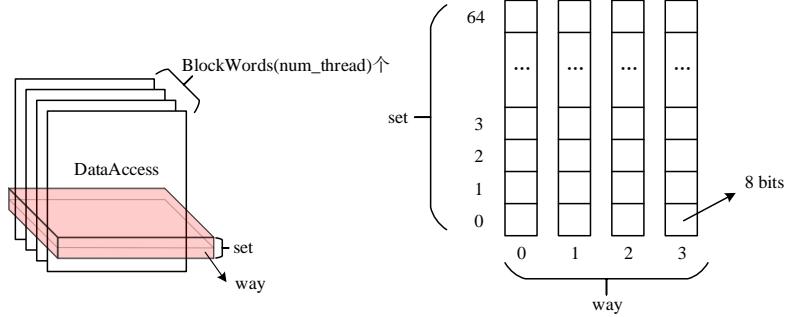


图 5-7 Dcache 内部 SRAM 的组织形式

在添加了有效位、脏位和替换策略信息后，缓存空间的结构如下表所示。

setIdx	V	D	LRU	Tag	Block 0				Block 1			
					0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00
0	1	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
1	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
2	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
3	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	0	0	0x000	0x000000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

整个 cache 分为对 LSU 接入的两组接口 coreReq 和 coreRsp，以及对更高层 cache (L2) 接入的两组接口 memReq 和 memRsp。coreReq 支持的请求类型包括：普通读请求（标量/向量）、普通写请求（标量/向量）、fence 请求、预留性读请求、条件写请求、原子操作请求。这些请求在 cache 中经历的处理操作如下图。

(图 5-8—图 5-14 来源：金楚丰，通用图形处理器缓存子系统关键技术与实现。参考链接：[清华大学学位论文服务系统 \(tsinghua.edu.cn\)](http://tsinghua.edu.cn))。

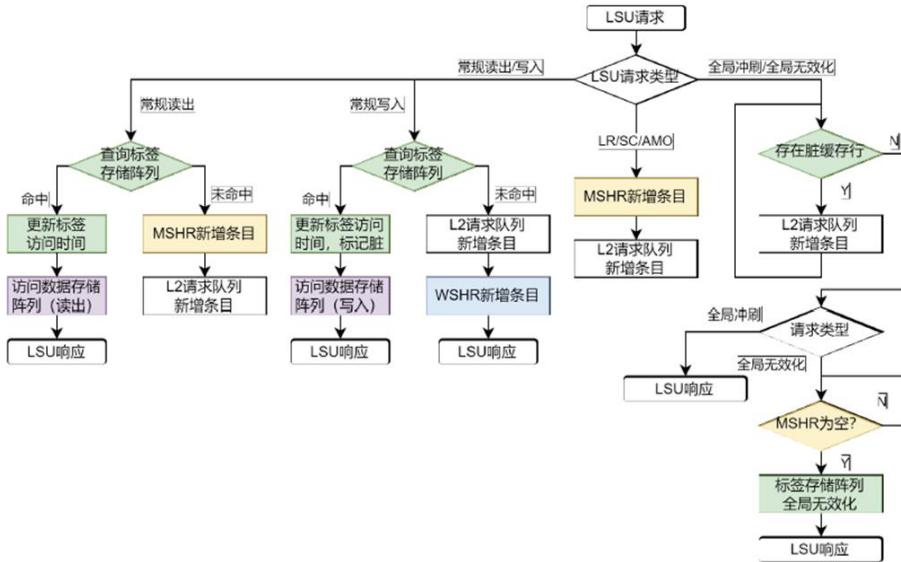


图 5-8 不同请求类型对应的 Dcache 处理过程

这些请求操作大多数与指令一一对应。AMO 指令是一个例外，AMO 指令允许带有.aq 和 /或.rl 标识符（acquire 和 release），对于携带此类标识符的访存指令，LSU 会将其分割为一个不携带标识符的访存请求和一条对应的 fence 请求（冲刷或无效化请求）。

memRsp 支持的响应类型包括：常规读取缺失的响应和 LR、SC、原子操作的响应。这些响应在 cache 中经历的处理操作如下图：

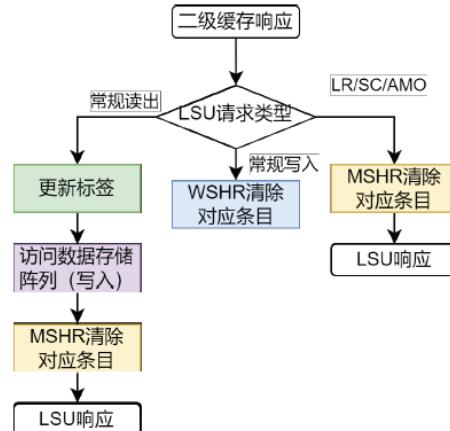


图 5-9 不同 L2 响应对应的 Dcache 处理过程

整个 cache 由三个流水级构成，标签存储阵列、缺失状态记录器和数据存储阵列模块在当前周期接收有效的读出请求，在一个时钟上升沿之后输出响应，其他信号在跨越流水级时使用流水线寄存器。二级缓存响应端口、LSU 响应端口、二级缓存请求端口采用了队列，用于前后级流水线暂时缓冲。一级数据缓存流水线示意图如下：

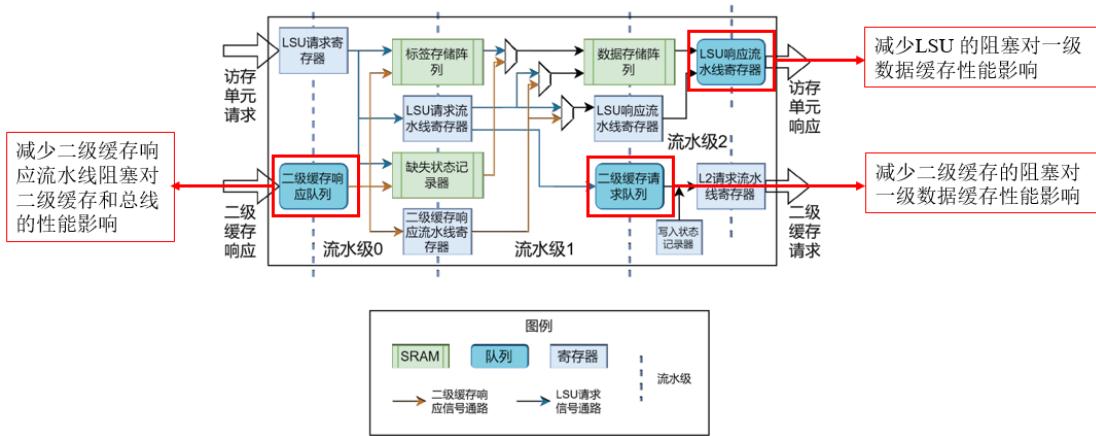


图 5-10 Dcache 的流水线路径

对于访存单元请求流水线路径：

- (1) 流水级 0：发出标签存储阵列 tag 和缺失状态记录器 mshr 的 SRAM 访问请求；
- (2) 流水级 1：根据请求类型和查询结果 tag 决定后续操作方式：压栈二级缓存请求队列 memReq_Q 和缺失状态条目并或/与发出数据存储阵列 data 访问请求；
- (3) 流水级 2：压栈 LSU 响应队列条目 coreRsp_Q；

各类型请求在访存单元请求流水线中的操作步骤如下表：

访存单元请求类型	常规读命中	常规读缺失	常规写命中	常规写缺失	LR/SC/AMO	全局冲刷	全局无效化
流水级0	发出tag读请求 发出MSHR探测	发出tag读请求 发出MSHR探测	发出tag读请求 发出MSHR探测	发出tag读请求 发出MSHR探测	发出MSHR探测	发出tag脏位查询	发出tag脏位查询
流水级1	得到tag结果 发出time更新请求 发出data读出请求	根据tag结果 触发MSHR请求 压栈memReq	得到tag结果 发出time更新请求 发出data写入请求 发出脏位标记请求	得到tag结果 压栈memReq 发出WSHR记录请求	根据检查结果 压栈memReq	压栈memReq	压栈memReq 清空MSHR tag无效化
流水级2	完成time更新 完成data读出 压栈coreRsp		压栈coreRsp	压栈coreRsp			

对于二级缓存响应流水线路径：

- (1) 流水级 0：发出缺失状态记录器 mshr 的 SRAM 访问请求；
- (2) 流水级 1：根据请求类型和 mshr 查询结果，更新标签存储阵列 tag，准备压栈 LSU 响应队列条目；
- (3) 流水级 2：压栈 LSU 响应队列条目；

各类型请求在二级缓存响应流水线中的操作步骤如下表：

二级缓存请求类型	常规读缺失	常规写缺失	LR/SC/AMO
流水级0	发出MSHR查询请求	发出MSHR查询请求	发出MSHR查询请求
流水级1	得到查询结果 发送tag更新请求 发送time更新请求 发送data写入请求 发起MSHR清除请求	得到查询结果 发起WSHR清除请求	得到查询结果 发起MSHR清除请求 压栈coreRsp
流水级2	完成tag更新请求 完成time更新请求 完成data写入请求 完成MSHR清除请求 压栈coreRsp	完成WSHR清除请求	完成MSHR清除请求

下面介绍 Dcache 内部的两个核心子模块：l1_tagaccess 和 l1_mshr。

5.2.3.3.1 l1_tagaccess

l1_tagaccess 模块的设计框图如下图所示。图中，不同的 Access 采用了不同颜色进行表示，为了方便呈现时序器件的流水线多周期行为，会将同一个存储器不同端口绘制在多个图例实体中。CAM 表示按内容寻找，WriteBus 表示写入，readBus 表示读出。tag 存储阵列用

于记录缓存内所有行的存在状态、替换判据(图中 timeAccess 被替换成了 lru_matrix 模块)。其对外端口主要有三组：探测读、写分配和 invalid/flush，其内部共有 3 个流水级，但是第 2 流水级只用于更新内部的寄存器和 SRAM。

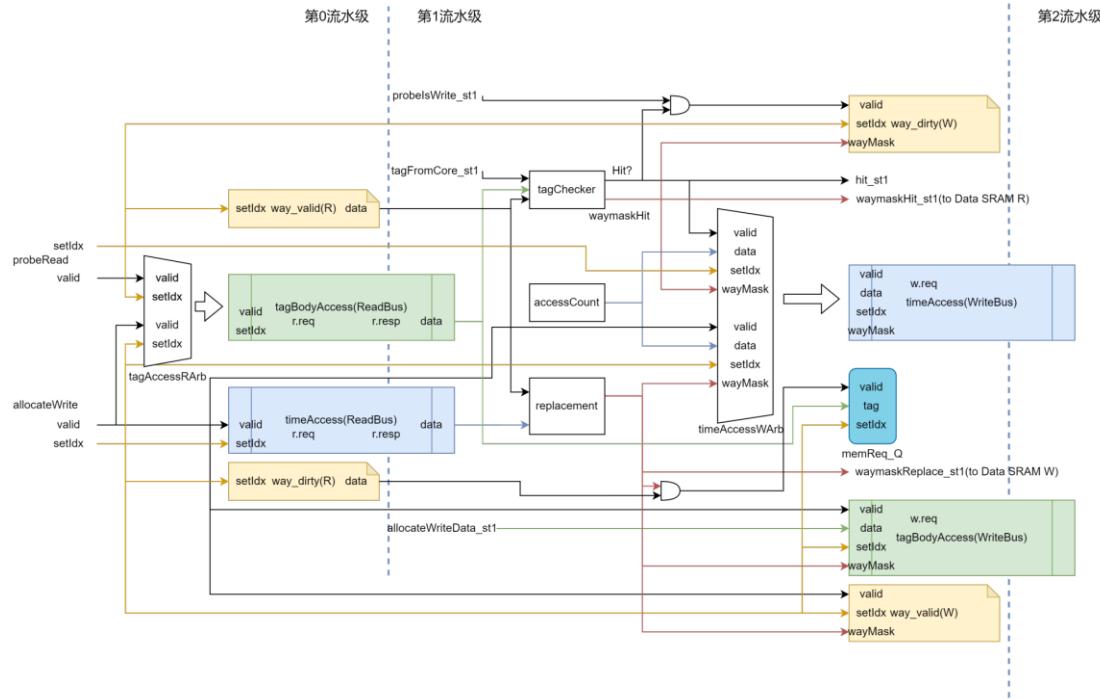


图 5-11 L1_tagaccess 的硬件框图

下面介绍标签存储阵列的三组输入输出信号：

(1) 在 LSU 请求类型中，常规读和常规写请求会触发标签存储阵列的探测读功能。这两种请求使用探测读功能来获知所访问的缓存行是否位于缓存中，即本次访存请求是命中，还是缺失。探测读在发生缓存命中时，还会告知命中的路索引，供后续的数据存储阵列读出来使用。

(2) 在二级缓存响应流程中，如果响应类型是常规读出缺失，会触发标签存储阵列的分配写功能。分配写流程完成了缓存行的填充，并在发生替换时发送新的二级缓存写入请求。

(3) invalidate/flush: invalidateAll 将所有缓存行的 valid 位都置 0，flush 将对应行的 dirty 位置 0，并输出对应缓存行的地址。

5.2.3.3.2 L1_mshr

L1_mshr 用于当 L1 Cache 内部没有 LSU 请求的缓存行时，存储对应缓存行的信息，下图为 L1_mshr 的整体框图。

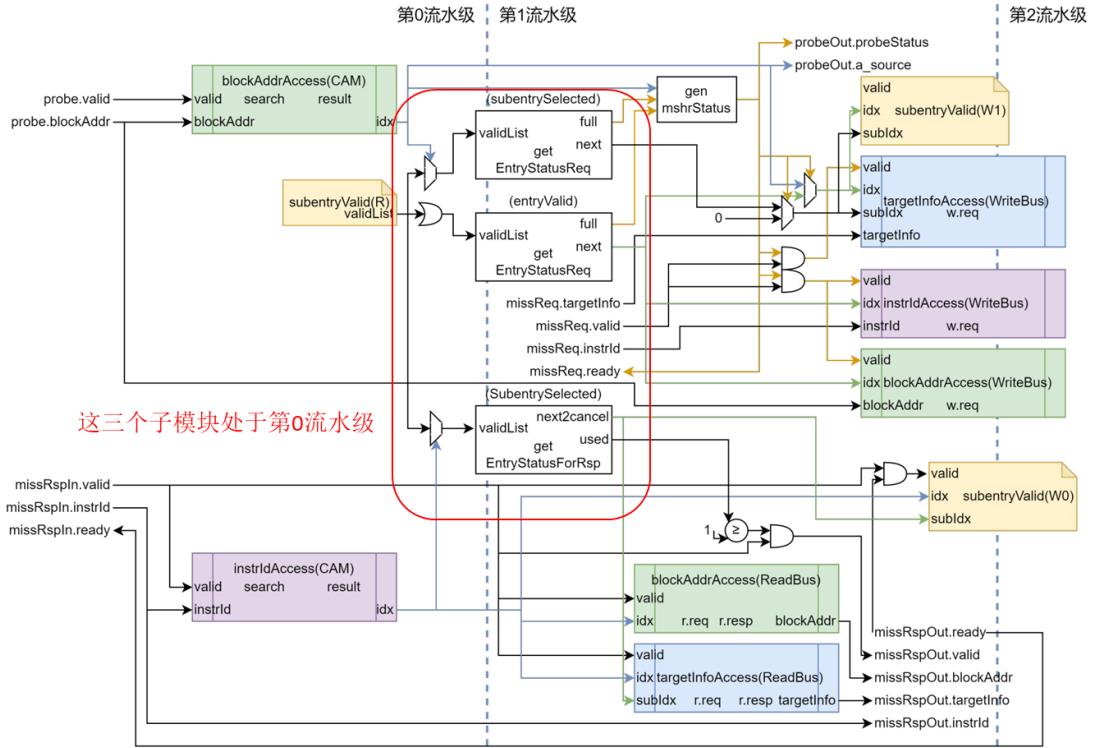


图 5-12 l1_mshr 的硬件框图

l1_mshr 记录了 L2 响应需要恢复成 LSU 响应的所有信息。其缓存结构如下，为了支持合并功能，一个主条目有多个副条目（目前是 2 个）。返回的缓存行携带 L2 请求时分配的 ID 信息。缓存行地址缓存了地址的 (tag+setIdx) 信息，目标信息存储的信息包括从 lsu 发送的 instrid、activemask、blockoffset 和 wordoffset1h 信息，如下图所示。在实践中，缺失请求总是会先填充第一个副条目。

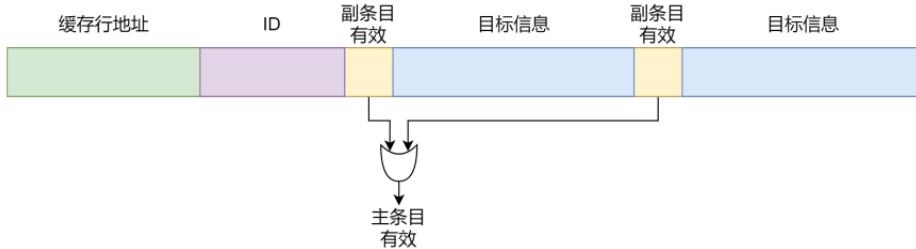


图 5-13 l1_mshr 的缓存结构

lsu 请求到来后，常规读出请求会触发缺失状态记录器的探测流程，在下一个周期从 l1_tagaccess 获取是否缺失的信息后，决定是否触发 MSHR 的缺失请求流程。

探测流程会将所有 LSU 请求的缓存行地址和 MSHR 中所有有效主条目缓存行地址进行一次匹配对比，对比流程如下图所示（代码中还有 secondary_full_return 这个状态，这个状态是为了保证当前主 entry 的所有副条目响应都返回给 lsu）。

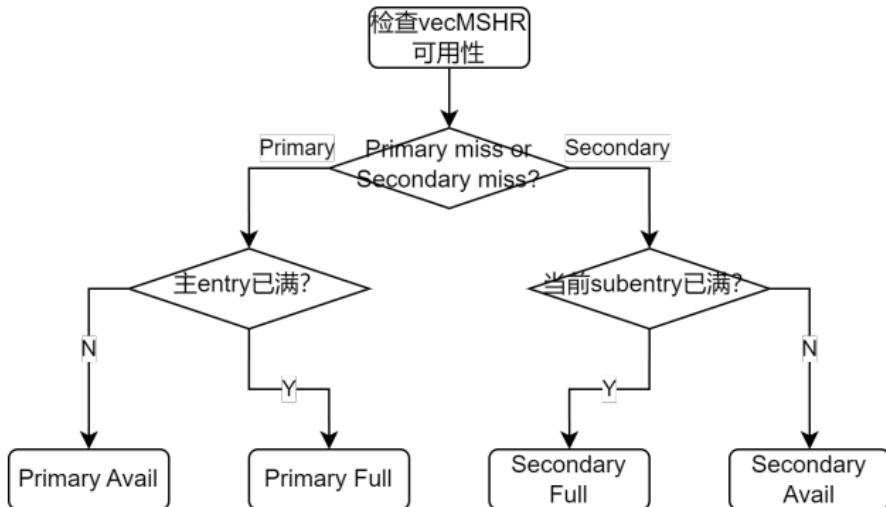


图 5-14 l1_mshr 探测请求流程图

l1_mshr 的流水线有停顿风险，其风险主要来自于下述 2 个方面：

- (1) 目前 LSU 请求的缓存行地址与所有主条目都对不上，且所有主条目都已有效；
- (2) 目前 LSU 请求的缓存行地址与其中一个主条目能对上，但该主条目的所有副条目都已有效。

5.2.3.3.3 特殊请求的设计

为了 Dcache 能正常工作，有几种特殊情况需要处理。下面简述目前需要注意的点。

- (1) lsu_req 和 l2_rsp 同时到来：

从硬件框架图中可以看出，lsu 的 req 和 l2 的 rsp 可能会引起资源上的冲突，在它们同时到来时，会优先处理来自 l2 的 rsp，在这个时候利用 core_req_stx_valid/ready 这组信号阻塞 coreReq 的通路。

- (2) dirtyReplace 和 flush/invalidate 操作：

从 coreReq 到 memReq 一半只需要 3 个时钟周期，这是因为 memReq 大都是因为 Cache Miss 导致的，不会从 l1_dataAccess 里面读数据。但是，在发生 dirty 替换或 flush/invalidate 时，需要将 l1_dataAccess 的脏数据读出，并以写请求的形式发送给 L2，这会多消耗一个时钟周期。在这种情况下，addr、mask、opcode 等信息需要跟随数据多寄存一个时钟周期。

- (3) 写对应的缓存行在之前已经读 miss，且这个读 miss 请求仍未得到响应：

目前的做法是，在遇到这种情况时直接阻塞 coreReq 通路，直到 mshr 内部清空（更合理的做法是只要对应缓存行得到了 l2_rsp，流水线就可以继续执行）。这么做是因为，coreReq 通路处理请求是顺序执行的，如果这个写请求不能执行，后续的 coreReq 通路本来就会阻塞。

5.2.4 L2 Cache

5.2.4.1 概述

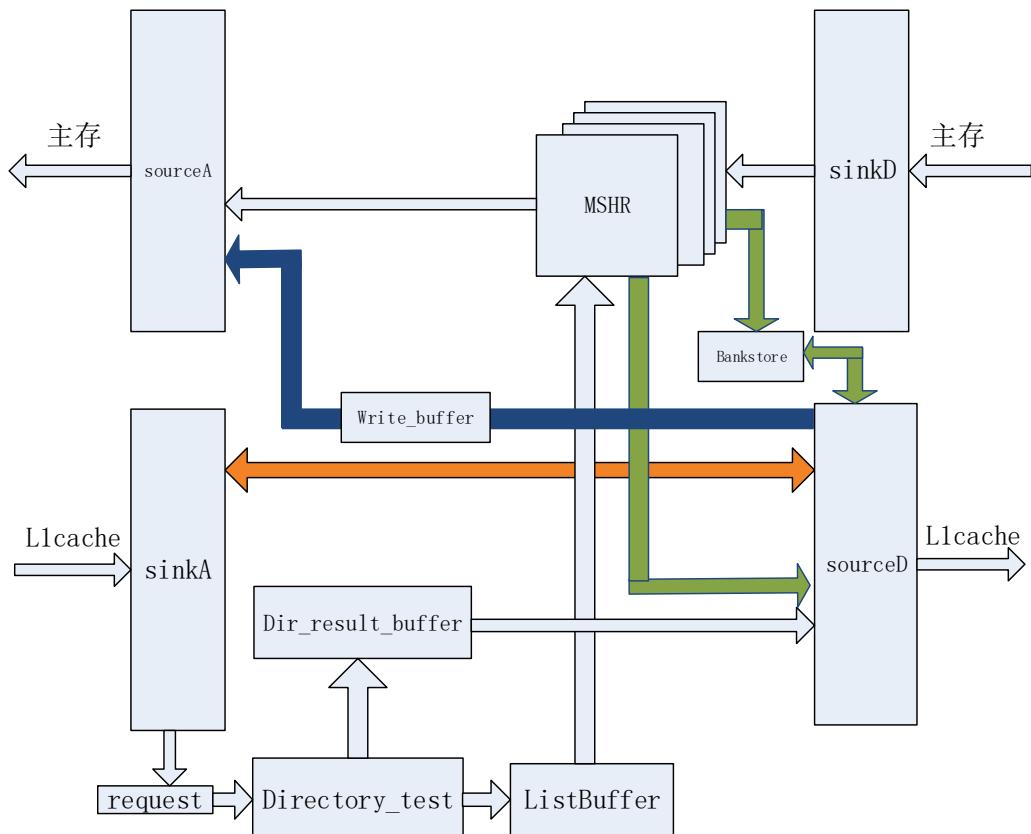


图 5-15 L2 Cache 的结构框图

5.2.4.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位, 低有效
From L1cache			
sche_in_a_valid_i	IN	1	握手信号
sche_in_a_ready_o	OUT	1	握手信号
sche_in_a_opcode_i	IN	3	操作码
sche_in_a_source_i	IN	12	发射该请求的主端源标识符
sche_in_a_address_i	IN	32	地址
sche_in_a_mask_i	IN	8	字节掩码
sche_in_a_data_i	IN	64	数据
sche_in_a_param_i	IN	3	参数码

sche_out_a_ready_i	IN	1	握手信号
To L1cache			
sche_out_a_valid_o	OUT	1	握手信号
sche_out_a_opcode_o	OUT	3	操作码
sche_out_a_size_o	OUT	3	传输字节数的对数
sche_out_a_source_o	OUT	12	发射该请求的主端源标识符
sche_out_a_address_o	OUT	32	地址
sche_out_a_mask_o	OUT	8	字节掩码
sche_out_a_data_o	OUT	64	数据
sche_out_a_param_o	OUT	3	参数码
From mem			
sche_out_d_valid_i	IN	1	握手信号
sche_out_d_ready_o	OUT	1	握手信号
sche_out_d_opcode_i	IN	3	操作码
sche_out_d_source_i	IN	12	发射该请求的主端源标识符
sche_out_d_data_i	IN	64	数据
sche_out_d_param_i	IN	3	参数码
sche_in_d_ready_i	IN	1	握手信号
To mem			
sche_in_d_valid_o	OUT	1	握手信号
sche_in_d_opcode_o	OUT	3	操作码
sche_in_d_source_o	OUT	12	接受该请求的主端源标识符
sche_in_d_data_o	OUT	64	数据
Sche_in_d_param_o	OUT	3	参数码

5.2.4.3 设计原理

Ventus 中的 L2 cache 主要是采用的 TileLink 总线协议的 TileLink 无缓存重量级标准 (TileLink Uncached Heavyweight, TL-UH)，TileLink 总线协议是和 RISC-V 一起诞生的开源片上总线协议。该总线定义了一个主-从点对点传输规范。在规范中设计了从 E 到 A 五种基于消息的优先级。TileLink 总线协议允许在消息和消息之间进行乱序，并针对无死锁和扩展性做了特殊设计。相比 ARM 公司的商用 AXI 总线协议，TileLink 总线协议精简干练，设计复杂度显著降低，具体内容可参考协议文档 (参考链接: [tilelink-spec-1.7.1-draft.zh.pdf\(cnrv.io\)](#))。

TL-UH 提供了 6 种消息类型，并且只用了五种消息优先级中的两个 (A 和 D, D 优先级大于 A)，如下表所示，ventus 目前使用了其中的四种消息类型，消息名称后的数字为对应的操作码 opcode，消息包含的其余参数可以参考协议文档。

消息 (操作码)	用途	响应消息 (操作码)	备注
Get(4)	读出	AccessAckData(1)	
PutFullData(0)	写入 (全部)	AccessAck(0)	
PutPartialData(1)	写入 (部分)	AccessAck(0)	
ArithmeticData(2)	原子操作 (算术)	AccessAckData(1)	未使用
LogicalData(3)	原子操作 (逻辑)	AccessAckData(1)	未使用
intent(5)	冲刷/全局无效	HintAck(2)	

Ventus 中的存储结构如下图所示, 其使用的缓存地址映射方式为组相连, 原理可见 [Cache 的基础知识 - 知乎 \(zhihu.com\)](#)。该 L2 cache 一共分成了 2 个 set 和 4 个 way, 共 8 个缓存行 (cache line), 每个缓存行有 16 个 block, 每个 block 包含 1 个字节, 因此整个缓存行是 128bit。综上所述, 32bit 的地址可以分成四部分, 1bit 的 set 索引、4bit 的 block 索引和 27bit 的 tag (组相连不需要 way 索引, 因为需要和 1 个 set 中所有缓存行的 tag 进行比较来判断是否命中)。每个缓存行还带有一个 valid 位和 dirty 位, 前者表示当前缓存行是否有效, 后者表示该缓存行是否被修改, 若被修改的话当该缓存行被替换时需要被更新到主存。

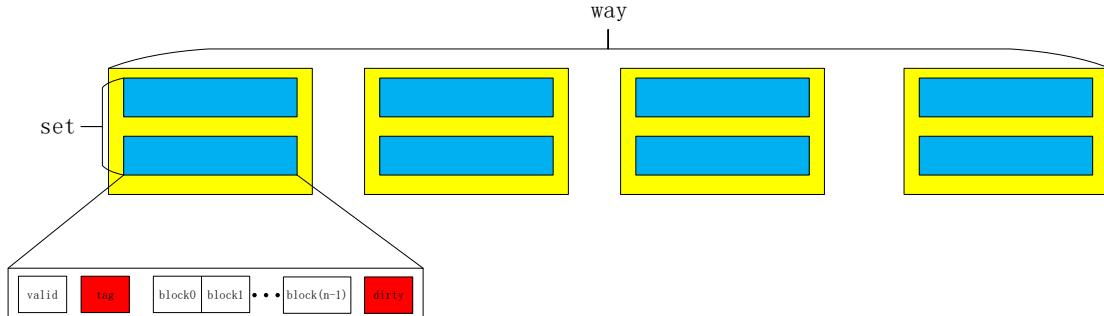


图 5-16 L2 Cache 的存储结构

Ventus 中 L2 cache 的更新策略是写回+写不分配, 具体流程如下图所示, 写回表示当 CPU 执行 store 指令并在 cache 命中时只更新 cache 中的数据, 同时将 dirty 位置位, 主存中的数据只会在 cache line 被替换时更新。写不分配指当 CPU 写数据发生 cache 未命中时, 首先从主存中加载数据对 cache line 进行替换, 然后再对该 cache line 进行写操作。(代码中的情况是不会更新 L2 cache, 而是对来自主存的 cache line 进行写操作后直接传回主存, 与承影文档矛盾)。

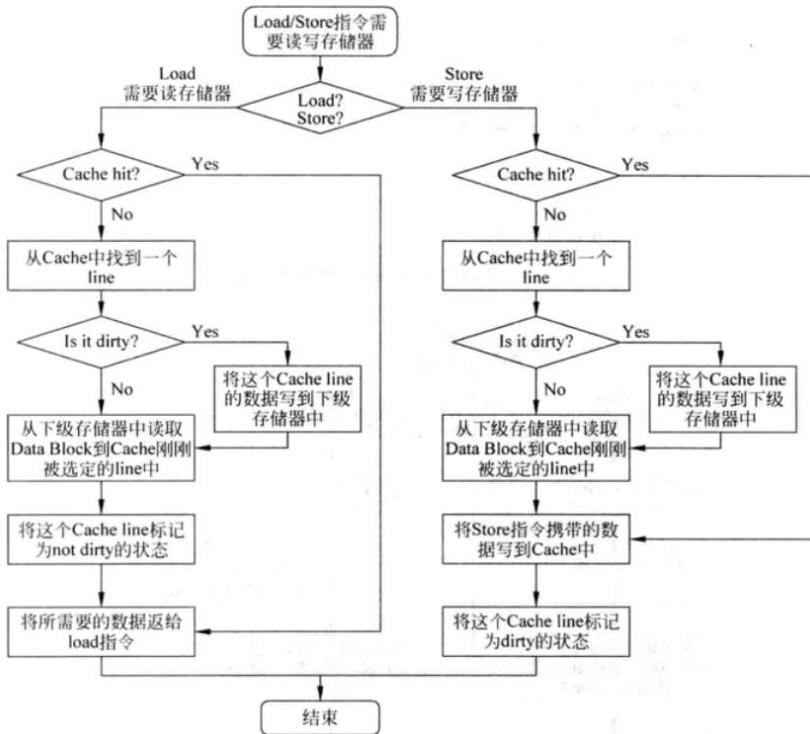


图 5-17 L2 Cache 的流程图

无论是写入或者读出 L2 cache 时发生了缺失, 都需要从对应的 set 中找到一个 way, 来存放下级存储器读出的数据, 如果此时这个 set 内所有的 way 都被占用, 那么就需要替换一个, 如何从这些有效的 cache line 中找到一个并替换之, 这就是替换策略。Ventus 中 L2 cache

的替换策略是 LRU 替换（具体原理见 Directory_test 模块），即找一个使用次数最少的 way 进行替换。

下面介绍在不同条件下 L2 Cache 的对应操作。

(1) 读命中

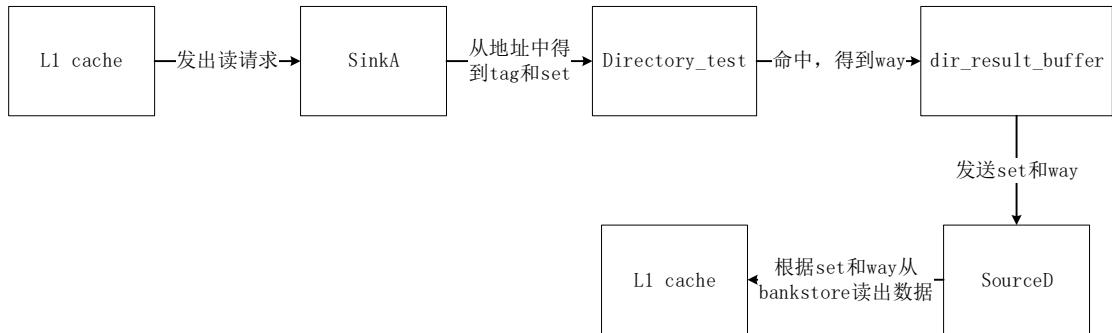


图 5-18 读命中时 L2 Cache 的操作流程

当来自 L1 cache 的请求为 Get 操作时，首先经过 SinkA 模块发出请求，并从 32bit 的地址得到该请求的 tag 和 set，若 Directory_test 模块判断命中，就将唯一确定的地址（set 和 way）发送给 dir_result_buffer 和 SourceD 模块，SourceD 模块根据地址从 BankStore 中读出响应缓存行并发送给 L1 cache。

(2) 写命中



图 5-19 写命中时 L2 Cache 的操作流程

当来自 L1 cache 的请求为 PutFullData 和 PutPartialData 时，与读命中不同的是，数据最后会根据 4bit 的 mask 写入 BankStore 中某个缓存行，每一 bit 对应 32bit 数据。

(3) 读不命中

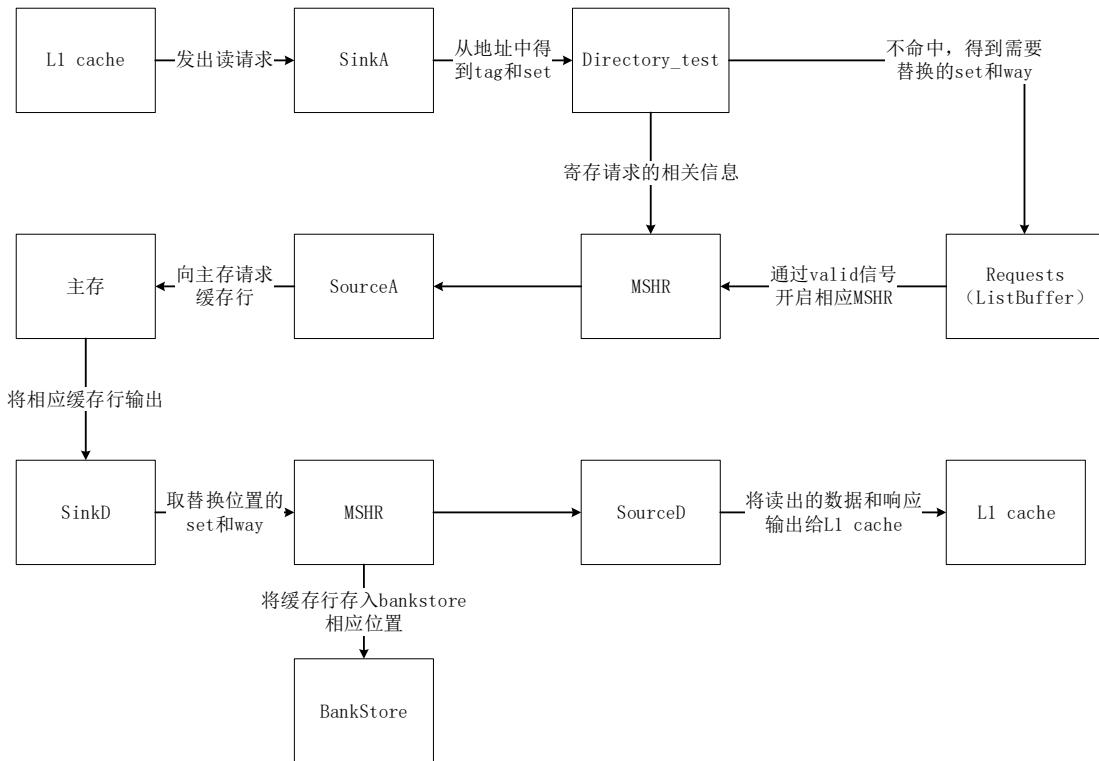


图 5-20 读不命中时 L2 Cache 的操作流程

(4) 写不命中

当 Directory_test 模块判断读不命中时，将该请求发送给 Requests 并将空闲的 MSHR 使能，从而 MSHR 能接收并寄存来自 Directory_test 模块的未命中请求的信息，然后 MSHR 通过 SourceA 模块向主存发送请求（请求中的 Source 会替换成 MSHR 的索引），主存将缓存行发送给 SinkD 模块，通过 Source 可以索引到对应的 MSHR 并取得替换地址（set 和 way），因此可以将缓存行存入 BankStore，同时还通过 SourceD 将读数据和响应发送给 L1 cache。

(5) 不命中且被替换行带有脏数据

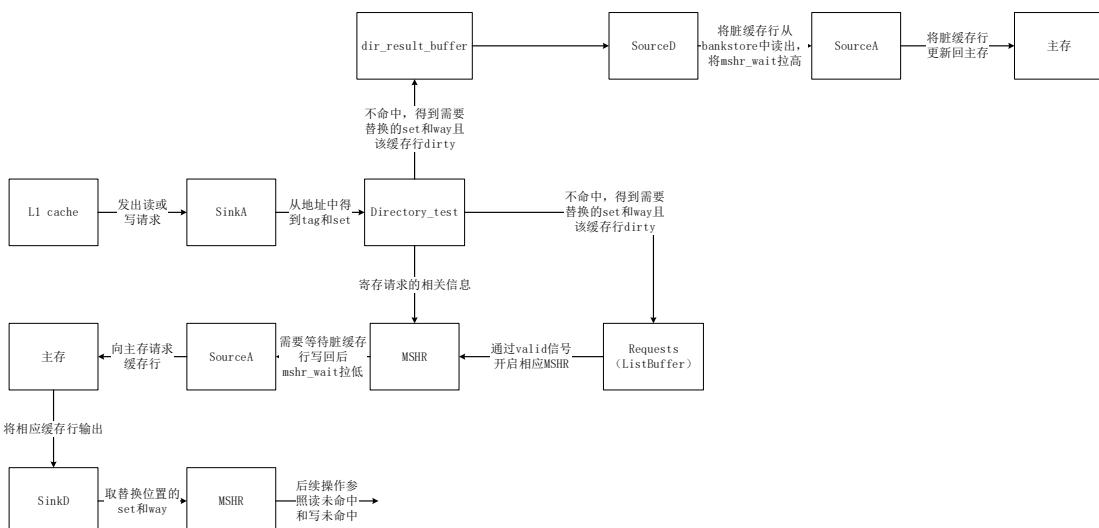


图 5-21 不命中且被替换行带有脏数据时 L2 Cache 的操作流程

首先，只有不命中的情况下才会需要替换 BankStore 中的缓存行，因此才需要考虑需要替换的缓存行是否为脏。当 Directory_test 模块判断未命中且脏时，会将需要替换的缓存行的地址（set 和 way）发送给 SourceD，并将 mshr_wait 为高，MSHR 就

无法将请求发送给 SourceA，需等待脏缓存行写回主存。后续的流程与读不命中/写不命中一致。

(6) 冲刷/全局无效

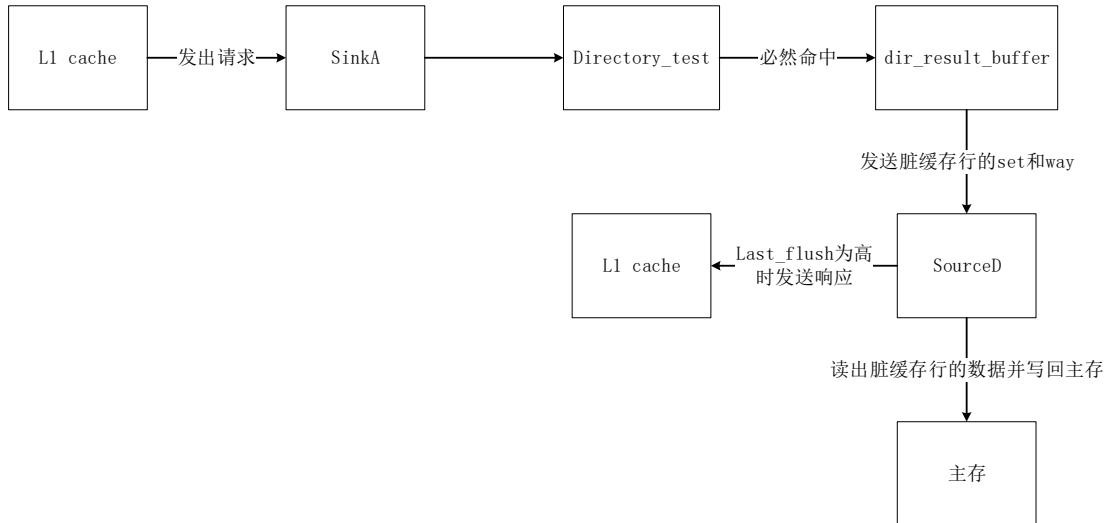


图 5-22 冲刷/无效化时 L2 Cache 的操作流程

Ventus 中的 Hint 指令的功能主要是实现 flush 和 invalidate，当 param 为 0 时为 flush 操作，将 L2cache 中所有脏缓存行写入主存；当 param 为 1 时为 invalidate 操作，将 L2cache 中所有脏缓存行写入主存并且将所有缓存行的 valid 位拉低。此操作经过 directory_test 后会发出 NUM_WAY*NUM_SET 个请求，遍历所有缓存行，然后给 L1cache 发送响应。需要注意的是，一旦 L2cache 接收到 flush/invalidate 请求，需要停止接受外部请求直到冲刷完成，同时需要等待内部 MSHR 清空才能执行 flush/invalidate。

上文中的框图简单介绍了 L2 cache 的结构，接下来详细介绍各个模块的内部结构以及实现方法。由于 SourceA 模块和 SinkD 模块只是简单的输入输出相连，SinkA 模块中的 putbuffer 与 ListBuffer 原理基本一致，因此不单独介绍。

5.2.4.3.1 Directory_test

5.2.4.3.1.1 概述

Directory_test 在 L2Cache 当中，是所有数据流必经过的模块，如上可见框图。Directory_test 主要承担了以下四个功能：(1) Flush 和 invalidate 功能；(2) cacheline 的状态位 valid 或者 dirty 标记；(3) LRU 更换 way；(4) 判断命中与否。

5.2.4.3.1.2 信号说明

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
From MSHR			
名称	类型	位宽	描述
dr_write_valid_i	IN	1	握手信号
dr_write_ready_o	OUT	1	握手信号

dr_write_way_i	IN	2	待写的 way 的 id
dr_write_tag_i	IN	28	待写的 tag
dr_write_set_i	IN	1	待写的 set 的 id
From sinkA			
dir_read_valid_i	IN	1	握手信号
dir_read_ready_o	OUT	1	握手信号
dir_read_set_i	IN	1	要读的 set 的 id
dir_read_opcode_i	IN	3	要读的操作码
dir_read_size_i	IN	3	要读的规模大小
dir_read_source_i	IN	12	要读的主端源标识符
dir_read_tag_i	IN	28	要读的 tag 标识
dir_read_offset_i	IN	3	要读的地址偏移量
dir_read_put_i	IN	2	要读的请求字段地址
dir_read_data_i	IN	64	要读的缓存行数据
dir_read_mask_i	IN	8	要读的缓存行数据掩码
dir_read_param_i	IN	3	要读的参数符
To SourceD			
dir_result_valid_o	OUT	1	result 的握手信号
dir_result_ready_i	IN	1	result 的握手信号
dir_result_victim_tag_o	OUT	28	要替换的 tag 信息
dir_result_way_o	OUT	2	待操作的 way 的 id
dir_result_hit_o	OUT	1	命中信号
dir_result_dirty_o	OUT	1	dirty 标志位信号
dir_result_flush_o	OUT	1	flush 信号
dir_result_last_flush_o	OUT	1	flush/invalidate 的结束信号
dir_result_set_o	OUT	1	待操作的 set 的 id
dir_result_opcode_o	OUT	3	待操作的操作码
dir_result_size_o	OUT	3	待操作的 size 信息
dir_result_source_o	OUT	12	待操作的 source 信息
dir_result_tag_o	OUT	28	待操作的 tag 的信息
dir_result_offset_o	OUT	3	待操作的 offset 信息
dir_result_put_o	OUT	2	待操作的 put 的 id
dir_result_data_o	OUT	64	待操作的缓存行数据
dir_result_mask_o	OUT	8	待操作的缓存行的掩码信息
dir_result_param_o	OUT	3	待操作的参数码信息
To scheduler			
dir_ready_o	OUT	1	Directory_test 可处理请求的握手信号
From scheuler			
dir_flush_i	IN	1	开始 flush
dir_invalidate	IN	1	开始 invalidate

dir_tag_match_i	IN	1	表示 tag 匹配
-----------------	----	---	-----------

5.2.4.3.1.3 设计原理

Directory_test 内部 sram 存储定义：

codeBits 对应 cacheline 的位宽、singport 是否单端口、cc_dir 是对应的 sram 的名字。sram 大小为 set*way 条 cacheline, 支持旁路写(读写地址相同时, 直接输出而不是先存 sram)、支持 reset。

cc_dir 的内部存储示意, 每一条 way 对应了一条各自的 tag, 一共的 tag 数量是 num_set*num_way。tag 和主存的地址相关。实际 cache 的存储空间大小占用 tag 部分, 但不包括这部分。L2cache 每次通过判断比较 sram 中是否有相同的 tag, 来决定是否命中。根据 tag 在 sram 中进一步存储的位置决定命中的 way 的 id。

Directory_test 内部实现的功能如下：

(1) Flush 和 invalidate 功能

directory_test 需要根据 request 的预取 (Hint) 指令作 invalidate 和 flush 的功能。

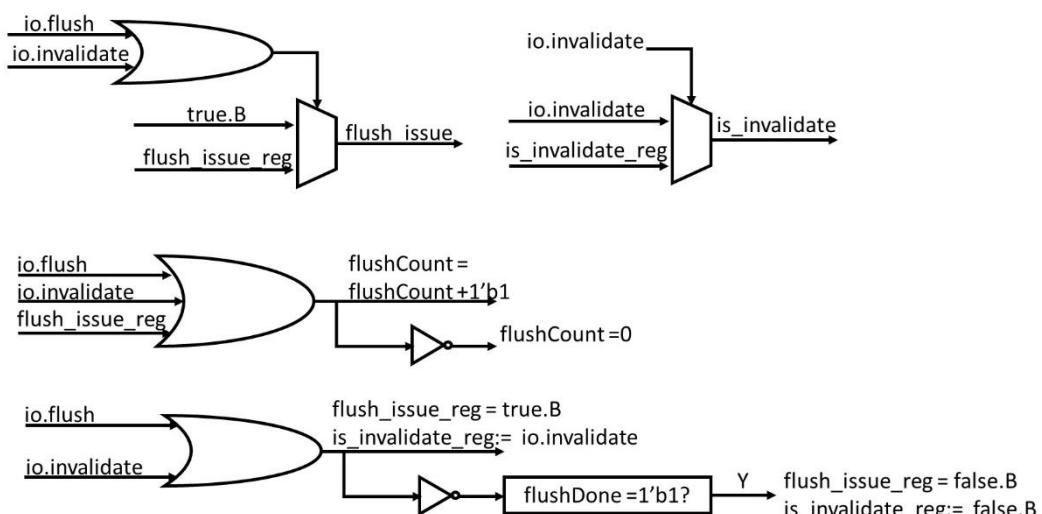


图 5-23 Directory_test 判断 flush/invalidate 的过程

flush 和 invalidate 可作全局清零, flush 和 invalidate 过程中默认命中, 输出的是对应 flush 过程中的 way 和 set。invalidate 标记全局 cacheline 的 valid 状态位为 invalid。invalidate 功能包括了 flush 功能, 他们是以 param 的状态标记区分的。invalidate 和 flush 功能均需要执行多个周期。

(2) cacheline 的状态位 valid 或者 dirty 标记。

v 位和 d 位更新规则如下:

- 在 wipedone 结束后, v 和 d 位全部清零
- 在 flush 过程中, 逐个清零对应 way 的 d, 但需要根据 invalidate 信号进一步清对应 way 的 v 位。
- 在写命中对应 way 时, 置位此 way 的 d
- 除此之外, 在未命中对应的 way、set, 且不是写 miss 时, (读 miss)进行 d 和 v 位的清零
- 在 directory 的 write.valid (实际上是读到了请求主存返回来的数据) 到对应 set 和 way 时, 置位对应的 v。
- 写 miss 不走 directory, 只获得相应用于替换的 set 和 tag。

(3) LRU 替换 way 策略

miss 且发现脏数据时需要替换含脏位的缓存行。替换策略这里使用的是 LRU (Least

recently used) 产生待替换 way 的 id, 以达到将最近最少使用的 way 替换出去的方法。LRU 内部存有标记访问过的 way 的 id 矩阵, 每次访问过后标记, 在需要发生替换掉最少的 way 时, 将标记最少的 way 替换掉。在所有的 way 未被轮询标记完前, 优先输出未被标记的 way id。

(4) 判断命中。

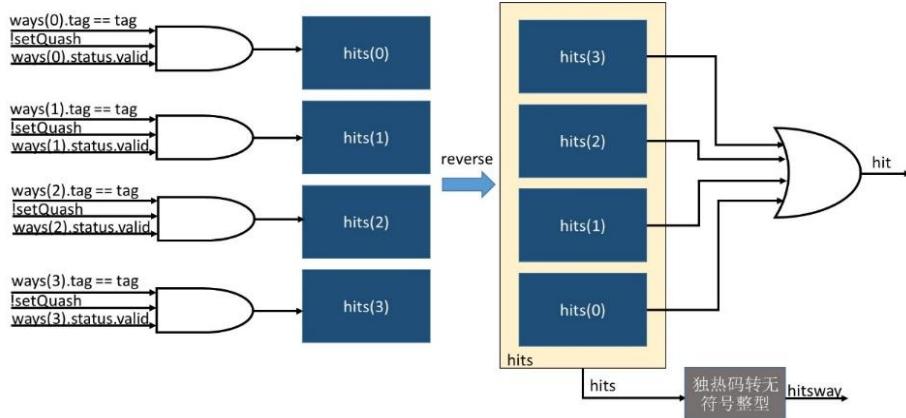


图 5-24 Directory_test 判断是否 hit 的过程

硬件通过判断每次请求来的 tag 是否在内部 sram 读的一整个 set 的 tag 中, 来判断是否命中, 根据命中的 hit 的 id 来判断具体命中的 way 的 id。hit 表示 hits 的或、hitway 表示 hits 命中对应的 (独热码转 UInt 类型) way。setQuash 和 tagmatch 表示写数据还在总线上, 但还未写入存储当中, 这种情况和前一个周期或者本周期的 read 数据比较, 若相等则也是认为是命中的。

tag 和 set 是上一次读使能过后从 read 端口寄存的 tag 和 set。

setQuash、setQuash_1、tagMatch、tagMatch_1, 表示写 tag (或写 set) 与上次或者待读的 tag (或者 set) 一致。

5.2.4.3.2 ListBuffer

5.2.4.3.2.1 概述

ListBuffer 用于存储来自 L1 Cache 的请求, 其存储结构如下图所示。

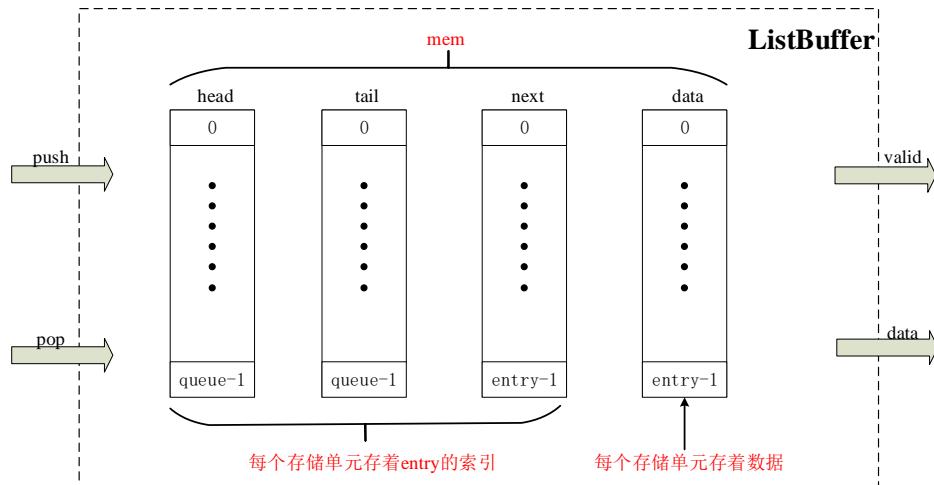


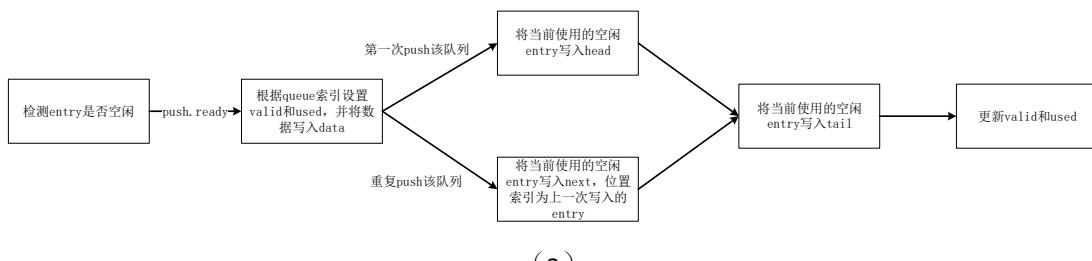
图 5-25 ListBuffer 的存储结构

5.2.4.3.2.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
To directory_test			
List_buffer_push_ready_o	OUT	1	往 List_buffer 内部传输的握手信号
List_buffer_valid_o	OUT	4	表明 List_buffer 内部是否有剩余空间
From directory_test			
List_buffer_push_valid_i	IN	1	往 List_buffer 内部传输的 push 握手信号
List_buffer_push_index_i	IN	2	List_buffer 内部深度索引
List_buffer_push_data_data_i	IN	64	List_buffer 内部存储数据
List_buffer_push_data_mask_i	IN	8	List_buffer 内部数据掩码
List_buffer_push_data_put_i	IN	2	List_buffer 内部 push 的位置
List_buffer_push_data_opcode_i	IN	3	List_buffer 内部 push 的操作码
List_buffer_push_data_source_i	IN	12	List_buffer 内部 push 的主端源标识符
List_buffer_pop_valid_i	IN	1	往 List_buffer 外部 pop 传输的握手信号
List_buffer_pop_data_i	IN	2	pop 的位置索引
To MSHR			
List_buffer_data_data_o	OUT	64	pop 的数据行
List_buffer_data_mask_o	OUT	8	pop 的数据掩码
List_buffer_data_put_o	OUT	2	pop 的位置深度索引
List_buffer_data_opcode_o	OUT	3	pop 的操作码
List_buffer_data_source_o	OUT	12	pop 出的主端源标识符

5.2.4.3.2.3 设计原理

ListBuffer 的 Push/pop 流程如下图所示。



(a)

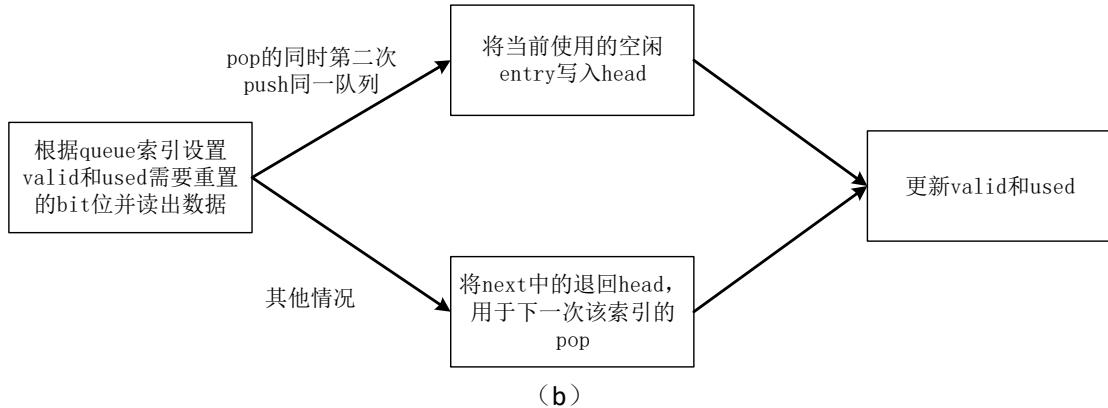


图 5-26 ListBuffer 的 (a) Push 和 (b) pop 流程

总的来说，存储器 tail 用来存 push 时的 entry，存储器 head 用来存 pop 时的 entry，由于可以重复给同一个 queue 进行 push 操作（使用不同的 entry），因此设置了存储器 next 来存储重复的 push 操作的 entry。valid 信号的每一 bit 对应一个 queue 是否有数据，used 信号的每一 bit 对应一个 entry 是否被占用，**当需要进行 push 操作时要检查 used 是否有空闲位置**，每次完成 push 或者 pop 操作后更新 valid 和 used 的相应 bit 位。

5.2.4.3.3 BankStore

5.2.4.3.3.1 概述

BankStore 是 L2 Cache 用于存储数据的模块，其硬件框图如下图所示。

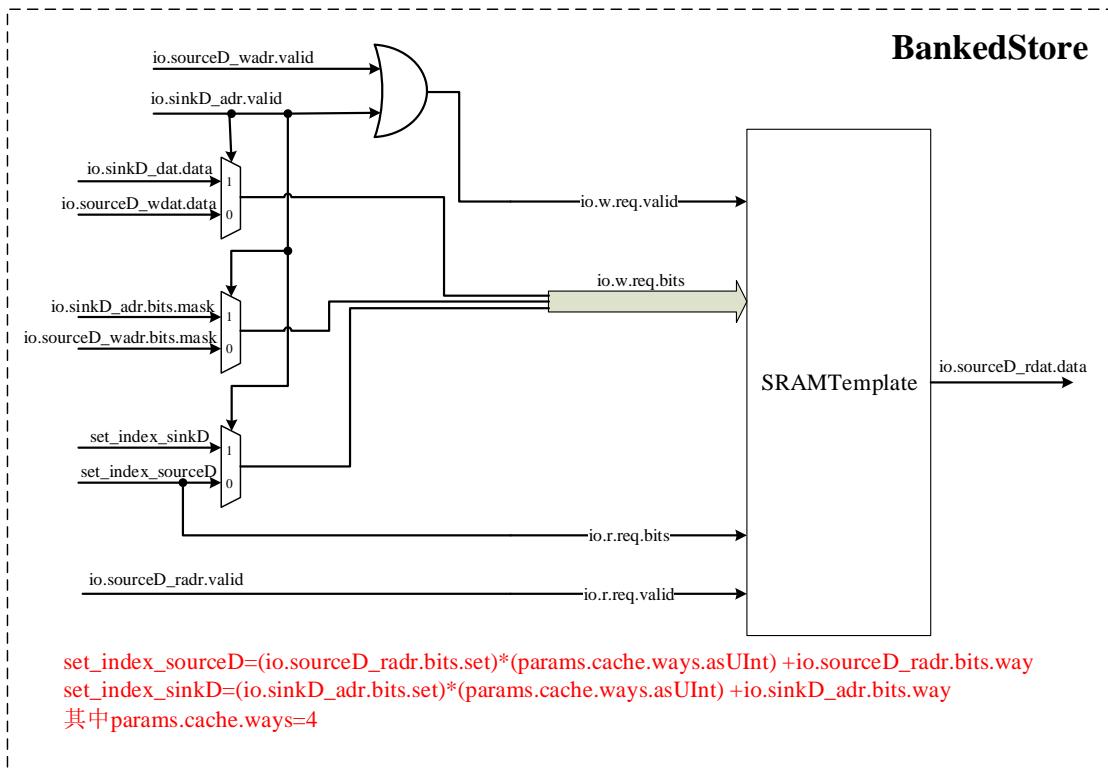


图 5-27 BankStore 的硬件框图

其中 SRAMTemplate 的结构如下图所示。

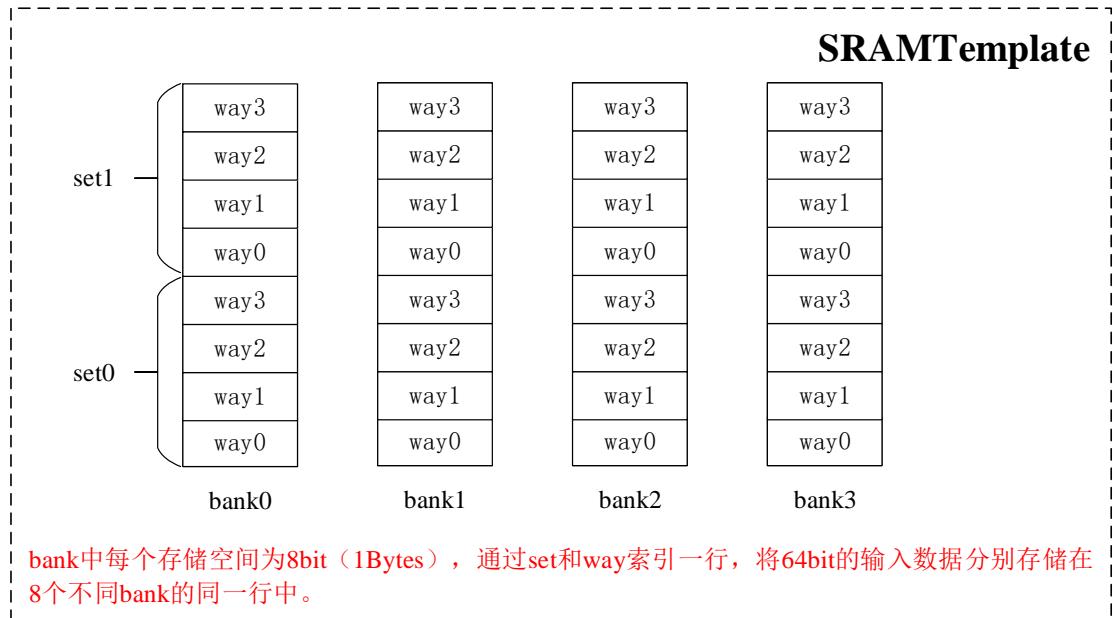


图 5-28 SRAMTemplate 的结构框图

5.2.4.3.3.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
From sinkD			
sinkD_adr_way_i	IN	2	索引的 way id
sinkD_adr_set_i	IN	1	索引的 set id
sinkD_adr_mask_i	IN	8	索引的数据掩码
sinkD_dat_data_i	IN	64	索引的数据
sinkD_adr_valid_i	IN	1	握手信号
To sinkD			
sinkD_adr_ready_o	OUT	1	握手信号
From SourceD			
sourceD_radr_way_i	IN	2	读的 way id
sourceD_radr_set_i	IN	1	读的 set id
sourceD_radr_mask_i	IN	8	读的数据掩码
sourceD_radr_ready_i	IN	1	读的握手信号
sourceD_wadr_way_i	IN	2	写 way id
sourceD_wadr_set_i	IN	1	写 set id
sourceD_wadr_mask_i	IN	8	写数据掩码
sourceD_wadr_data_i	IN	64	写的数据行
sourceD_wadr_valid_i	IN	1	写的握手信号
To SourceD			
sourceD_rdat_data_o	OUT	64	读出的数据缓存行
sourceD_wadr_ready_o	OUT	1	写的握手信号

5.2.4.3.3.3 设计原理

BankedStore 模块接收来自 sinkD 的写地址、写数据和来自 sourceD 的写地址、写数据、读地址，并且给 sourceD 输出读数据，因此其内部主要包含一些组合逻辑和一个 SRAM。来自 sinkD 和 sourceD 的地址由 set (1bit)、way (2bit) 和 mask (8bit) 构成，如上图所示，通过 set 和 way 可以选中一行 (64bit)，再通过 mask 掩码将数据分成四部分存到相应的 bank 中，因此 bank 中每个存储空间为 8bit。

5.2.4.3.4 MSHR

5.2.4.3.4.1 概述

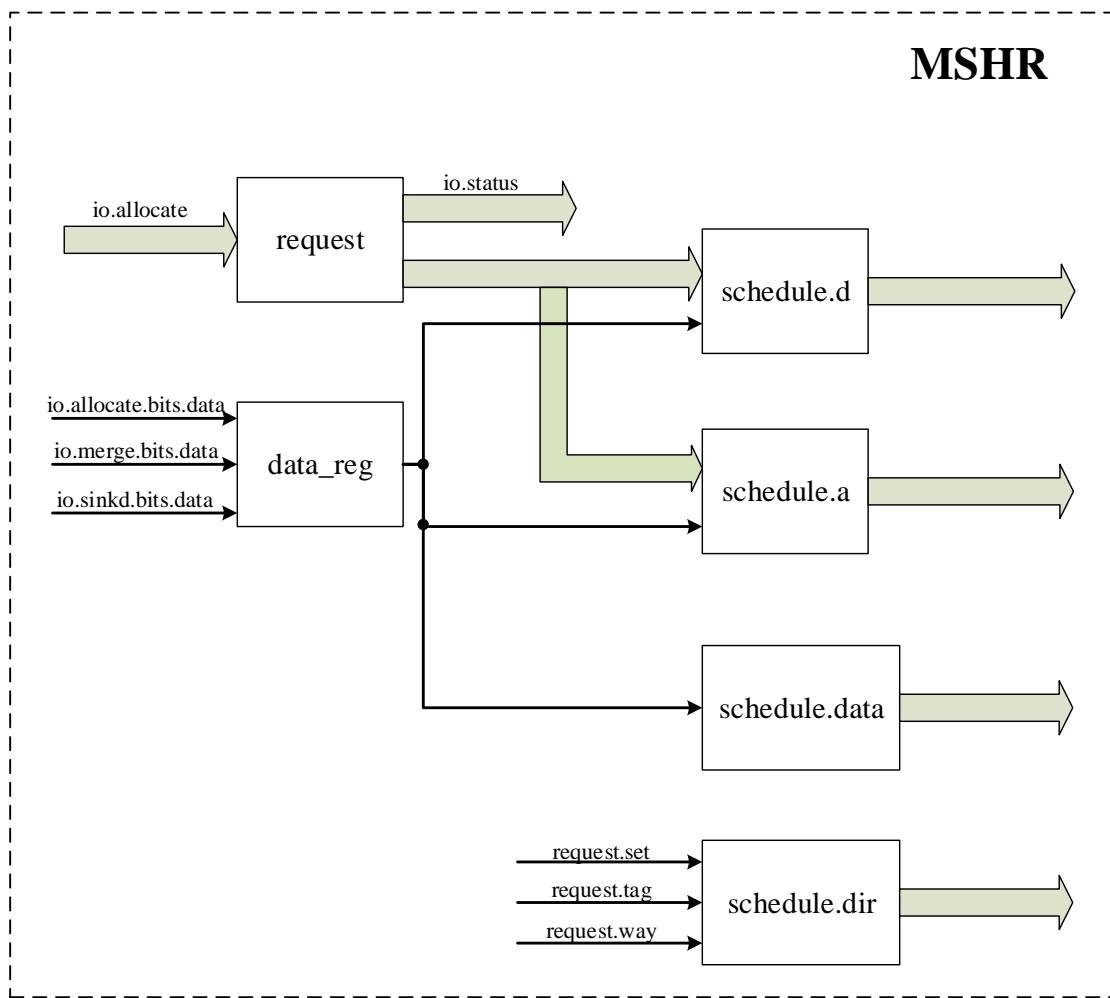


图 5-29 L2 MSHR 的结构框图

(1) MSHR 主要是实现非阻塞缓存功能，即记录未命中时请求的信息，因此模块中主要结构是寄存器，通过不同的 valid 信号进行寄存，上图中的 request 寄存未命中请求的各类信息，data_reg 寄存数据。

(2) MSHR 的输入主要有三类：①来自 Derectory_test 模块的 allocate 类，当缓存未命中时将请求的信息发送给 MSHR；②来自 ListBuffer 模块的 merge 类，当写未命中时完成输入数据对主存来的缓存行的写入；③来自 sinkD 模块的 SinkDResponse 类，用于从主存拿到数据后进入 MSHR 获得替换的缓存行位置 (way 和 set)。

(3) MSHR 的输出主要为 schedule 类，其可以分成 a、d、dir 和 data 四个类，分别输

出到 sourceA、sourceD、directory_test 和 bankstore 模块。

5.2.4.3.4.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位，低有效
From sinkD			
mshr_alloc_valid_i	IN	1	alloc 的握手信号
mshr_alloc_way_i	IN	2	alloc 的 way id
mshr_alloc_dirty_i	IN	1	alloc 的 dirty 与否
mshr_alloc_flush_i	IN	1	alloc 的 flush 信号
mshr_alloc_last_flush_i	IN	1	alloc 的 flush 的结束信号
mshr_alloc_set_i	IN	1	alloc 的 set id
mshr_alloc_opcode_i	IN	3	alloc 的操作码
mshr_alloc_size_i	IN	3	alloc 的传输字节数的对数
mshr_alloc_source_i	IN	12	alloc 的主端源标识符
mshr_alloc_tag_i	IN	28	alloc 的 tag 标识字段
mshr_alloc_offset_i	IN	3	alloc 的地址偏移量
mshr_alloc_put_i	IN	2	alloc 的地址索引深度
mshr_alloc_data_i	IN	64	alloc 的数据行
mshr_alloc_mask_i	IN	8	alloc 的数据掩码
mshr_alloc_param_i	IN	3	alloc 的参数字段
To Scheduler			
mshr_status_set_o	OUT	1	用于匹配 tag 相等的 set
mshr_status_opcode_o	OUT	3	用于判断此时的状态码
mshr_status_tag_o	OUT	28	用于匹配 tag 相等
From List_buffer			
mshr_valid_i	IN	1	mshr 接受是否有效
mshr_merge_valid_i	IN	1	请求是否是一次写操作
mshr_merge_mask_i	IN	8	写操作的数据掩码
mshr_merge_data_i	IN	64	写的数据行
mshr_merge_opcode_i	IN	3	写的操作码
mshr_merge_put_i	IN	2	写的位置深度
mshr_merge_source_i	IN	12	写的主端源标识符
From SourceD			
mshr_wait_i	IN	1	让 mshr 等待当前请求处理完毕
mshr_schedule_d_ready_i	IN	1	与 sourceD 的握手信号
From SourceA			
mshr_schedule_a_ready_i	IN	1	与 sourceA 的握手信号
From Sheduler			
mshr_mixed_i	IN	1	mshr 混合处理信号

From directory_test			
mshr_schedule_dir_ready_i	IN	1	与 directory_test 的握手信号
From sinkD			
mshr_sinked_valid_i	IN	1	和 sinkD 的握手信号
mshr_sinked_data_i	IN	1	和 sinkD 的回应数据
To Scheduler			
mshr_schedule_a_valid_o	OUT	1	优先处理的 mshr 数据部分
mshr_schedule_a_set_o	OUT	1	
mshr_schedule_a_opcode_o	OUT	1	
mshr_schedule_a_size_o	OUT	3	
mshr_schedule_a_source_o	OUT	12	
mshr_schedule_a_tag_o	OUT	28	
mshr_schedule_a_offset_o	OUT	3	
mshr_schedule_a_put_o	OUT	2	
mshr_schedule_a_data_o	OUT	64	
mshr_schedule_a_mask_o	OUT	8	
mshr_schedule_a_param_o	OUT	3	
mshr_schedule_d_valid_o	OUT	1	
mshr_schedule_d_hit_o	OUT	1	
mshr_schedule_d_way_o	OUT	2	
mshr_schedule_d_dirty_o	OUT	1	
mshr_schedule_d_flush_o	OUT	1	
mshr_schedule_d_last_flush_o	OUT	1	
mshr_schedule_d_set_o	OUT	1	
mshr_schedule_d_opcode_o	OUT	3	
mshr_schedule_d_size_o	OUT	3	
mshr_schedule_d_source_o	OUT	12	
mshr_schedule_d_tag_o	OUT	28	
mshr_schedule_d_offset_o	OUT	3	
mshr_schedule_d_put_o	OUT	2	
mshr_schedule_d_data_o	OUT	64	
mshr_schedule_d_mask_o	OUT	8	
mshr_schedule_d_param_o	OUT	3	
mshr_schedule_data_o	OUT	64	
mshr_schedule_dir_valid_o	OUT	1	
mshr_schedule_dir_way_o	OUT	2	
mshr_schedule_dir_data_tag_o	OUT	28	
mshr_schedule_dir_set_o	OUT	1	

5.2.4.3.4.3 设计原理

Ventus 中的 L2 cache 具有四个 MSHR，通过 LitBuffer 输出的 4bit 的 valid 信号控制，并通过 mshr_insertOH 得到空闲的 MSHR（优先级从 0 到 3 依次降低，可参考下表所示案例）。

当 `directory_test` 判断请求未命中后将请求信息 push 入 `ListBuffer`, 若为写请求将需要写入的数据作为 `merge_data` 一同存入。 `ListBuffer` 根据存入位置将输出的 `valid` 信号相应 bit 拉高, 从而控制相应的 MSHR 开启, 接收并寄存来自 `directory_test` 的未命中请求信息(`allocate`类)。

	初始状态	push 第一个未命中请求	push 第二个未命中请求	第一个未命中请求执行结束 pop	第二个未命中请求执行结束 pop
io.valid	0000	0001	0011	0010	0000
mshr_insertOH	0001	0010	0100	0001	0001

接着 MSHR 将未命中请求发送给 sourceA, sourceA 会将请求发送给主存, 主存通过 sinkD 将缓存行发送回 MSHR 取得需要替换的位置 (`set` 和 `way`)。若为读请求, MSHR 会通过 `io.schedule.dir` 端口输入 `bankstore` 进行缓存行替换, 并将读地址发送给 sourceD; 若为写请求, MSHR 直接将改写后的数据发送给 sourceD, 同时将该条未命中请求的信息从 `ListBuffer` 中 pop 出。

5.2.4.3.5 SourceD

5.2.4.3.5.1 概述

SourceD

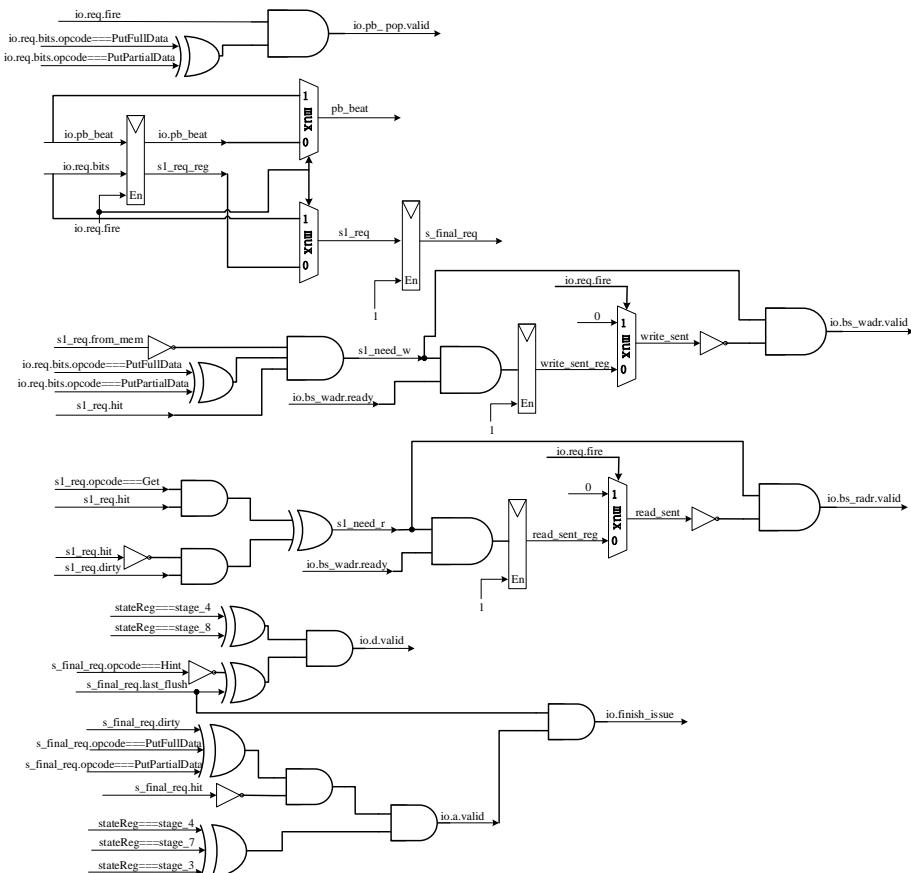


图 5-30 SourceD 的结构框图

(1) SourceD 中主要结构为状态机, 根据不同状态与 bankstore、sourceA 和 L1 cache 进行数据交互, 图中 `io.bs_wadr.valid` 和 `io.bs_radr.valid` 是对 bankstore 进行读写的使能, `io.a.valid` 是对 sourceA 进行数据传输的使能, `io.d.valid` 是对 L1 cache 进行数据传输的使能。

(2) 在 ventus 中, 只有写命中时会将 `io.bs_wadr.valid` 拉高, 从而对 `bankstore` 进行更新, 写不命中时直接将写好的数据传输给 `sourceA` (主存)。当读命中或不命中且需要替换的缓存行为脏时将 `io.bs_radr.valid` 拉高, 这两种情况都需要将 `bankstore` 中的数据读出, 需要区分的是, 读命中将数据输出给 `L1 cache`, 缓存行为脏时将数据输出给主存。

5.2.4.3.5.2 信号描述

名称	类型	位宽	描述
时钟及复位信号			
CLK	IN	1	系统时钟
RST	IN	1	全局复位, 低有效
From MSHR or directory_test			
<code>req_from_mem_i</code>	IN	1	请求是否和 mem 相关
<code>req_hit_i</code>	IN	1	请求是否命中
<code>req_way_i</code>	IN	2	请求的 way id
<code>req_dirty_i</code>	IN	1	请求是否为脏
<code>req_flush_i</code>	IN	1	请求 flush 信号
<code>req_last_flush_i</code>	IN	1	请求 flush 的结束信号
<code>req_set_i</code>	IN	1	请求的 set id
<code>req_opcode_i</code>	IN	3	请求的操作码
<code>req_size_i</code>	IN	3	请求的字节规模大小
<code>req_source_i</code>	IN	12	请求的主端源标识符
<code>req_tag_i</code>	IN	28	请求的 tag 信息
<code>req_offset_i</code>	IN	3	请求的 offset 信息
<code>req_put_i</code>	IN	2	请求的 put 信息
<code>req_data_i</code>	IN	64	请求的缓存行信息
<code>req_mask_i</code>	IN	8	请求的缓存行掩码信息
<code>req_param_i</code>	IN	3	请求的参数码字段信息
<code>req_valid_i</code>	IN	1	请求的握手信号
To MSHR			
<code>req_ready_o</code>	OUT	1	请求的握手信号
To Mem			
<code>d_address_o</code>	OUT	32	地址信息
<code>d_opcode_o</code>	OUT	3	操作码
<code>d_size_o</code>	OUT	3	字节规模大小
<code>d_source_o</code>	OUT	12	主端源标识符
<code>d_data_o</code>	OUT	64	数据缓存行
<code>d_param_o</code>	OUT	3	参数码
<code>d_valid_o</code>	OUT	1	握手信号
From			
<code>d_ready_i</code>	IN	1	握手信号
To sinkA			
<code>pb_pop_index_o</code>	OUT	2	待 pop 出的索引
<code>pb_pop_valid_o</code>	OUT	1	握手信号

From sinkA			
pb_beat_data_i	IN	64	pop 出的数据
pb_beat_mask_i	IN	8	pop 出的数据掩码
To bankstore			
bs_radr_way_o	OUT	2	读的 way id
bs_radr_set_o	OUT	1	读的 set id
bs_radr_mask_o	OUT	8	读的数据掩码
bs_radr_valid_o	OUT	1	读的握手信号
bs_wadr_way_o	OUT	2	写的 way id
bs_wadr_set_o	OUT	1	写的 set id
bs_wadr_mask_o	OUT	8	写的数据掩码
bs_wadr_valid_o	OUT	1	写的握手信号
bs_wadr_data_o	OUT	64	写出的数据缓存行
From bankstore			
bs_radr_ready_i	IN	1	读的握手信号
bs_wadr_ready_i	IN	1	写的握手信号
To sourceA			
a_set_o	OUT	1	发送给主存的 set id
a_opcode_o	OUT	3	发送给主存的操作码
a_size_o	OUT	3	字节数大小
a_source_o	OUT	12	主端源标识符
a_tag_o	OUT	28	tag 信息
a_offset_o	OUT	3	偏移量
a_data_o	OUT	64	数据缓存行
a_mask_o	OUT	8	数据掩码
a_param_o	OUT	3	参数码
a_valid_o	OUT	1	握手信号
From L1cache			
a_ready_i	IN	1	握手信号
To MSHR			
mshr_wait_o	OUT	1	让 MSHR 等待信号
finish_issue_o	OUT	1	Flush 结束信号

5.2.4.3.5.3 设计原理

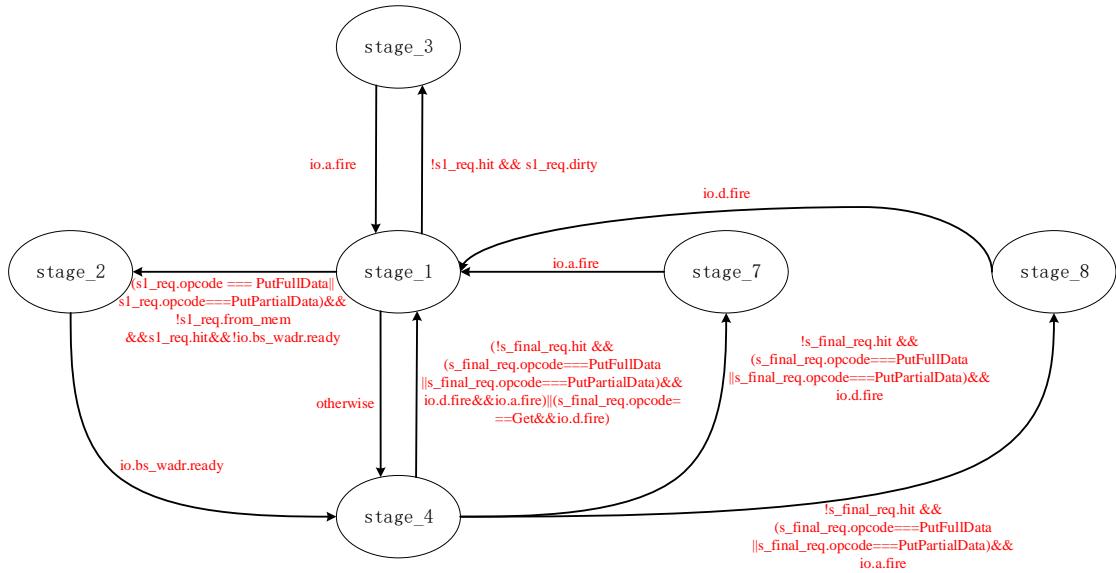


图 5-31 SourceD 的状态机跳转图

SourceD 中的状态转移图如上图所示，其中 `state_1` 为初始状态，`stage_2` 为等待写入 bankstore 的状态，`stage_3` 为将脏缓存行写回主存的状态，`stage_4` 为对命中或不命中或冲刷作出响应的状态，`stage_7` 为等待 sourceA 接收的状态，`stage_8` 为等待 L1 cache 接收的状态。

从图中可以看出，当请求不命中且需要替换的缓存行为脏时进入 `stage_3`，直到 `io.a.fire`（脏缓存行已经输出给 sourceA）后回到初始状态 `state_1`；当为命中的写请求且 bankstore 还未准备好写入时进入 `stage_2`，直到 bankstore 准备完毕进入 `stage_4`；`otherwise` 代表剩余其他情况都会进入 `stage_4`，直到不命中的写请求执行完毕（输出给 sourceA 和 L1 cache）或读请求执行完毕（输出给 L1 cache）后回到初始状态 `state_1`，若只输出给 L1 cache 则进入 `stage_7`，同理只输出给 sourceA 则进入 `stage_8`。当为 invalid 或 flush 请求时进入 `stage4`，若此时为 `dirty`，需要将 `io.a.valid` 拉高，从而将脏缓存行写回主存；若此时为 `last_flush`，需要将 `io.d.valid` 拉高，从而给 L1cache 请求完成的响应。