Alloc Module

2017年12月28日 17:30

Description

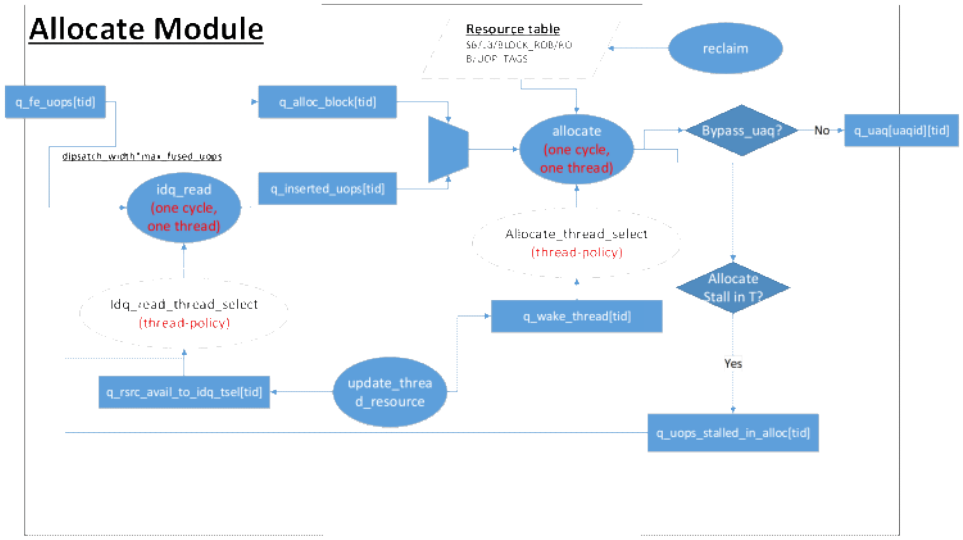
对于allocate的问题，主要解决如下几个问题

1. 分配后端需要使用的各种物理资源，比如load-buffer, store-buffer, ROB, uop-tag(暂时没明白干什么)， rs-entry, uaq-entry等
2. 解决指令之间的依赖关系，这里主要解决数据依赖以及逻辑上的依赖，比如lock
   a. 通过寄存器重命名解决
   b. 并不是所有的寄存器都可以通过renaming解决，需要区分哪些寄存器可以重命名，哪些不行(FCW，FC0-4)
   c. 寄存器的parital read/write问题
3. 解决memory renaming的问题，所谓memory renaming即访存地址上load/store之间有冲突的部分
4. 解决allocate在出现speculative-execution情况下，如何处理的问题---如何保证投机执行的"过时"指令不会影响到后面的pipeline
5. 处理当出现allocate stall情况时，如何stall/wakeup thread的问题

Related source file

allocate.h/.cc RAT/ROB top

整体流程



Structure

AllocateInfo

Stick in 1T/thread	
Num_dst_this_cycle	当前T内进行了几次dst reg的重命名，条件：不能是FP，不能是NULL REG，或者写EFLAGS
Num_loadop_this_cycle	判断当前T中有几个uop是load_op的fusing Micro-fusing OPT_4, OPT_6, OPT_17, OPT_19, OPT_15, OPT_16 Macro-fusing LOAD_OP_INT / LOAD_OP_FP
Num_uops_allocated_this_cycle	当前T allocate了多少个uop，对应next_entry_in_alloc_block这个值
Num_loadop_this_line	同num_loadop_this_cycle，角度不同，这次是站在fetch_line角度
Last_pos_allocated	表明当前T allocate中最后一次分配的uop在fetch line中的位置
Num_moves_bubbles_already_inserted_for	没有使用
Num_branches_allocated	当前T中branch的个数，目前没有使用
Next_is_fused	下一条uop是否是fused uop
Half_line	当前uop是否是当前一组uop_line发送到allocate中(同一个T发送的一组uops)的第一个
Last_uop_ended_allocation_block	针对fxch指令，这条指令必须为allocate中的最后一条指令
Num_rob_reads	表明当前T进行allocate分配的时候需要进行多少次rob read
Read_rob_entry[64]	记录当前T中哪些被重命名的rob被读取过
Read_rrf_reg[64]	记录当前T中哪些architecture register被读取过
Read_port_used[8]	记录rob read时候的Port使用情况
Restricted_stall	如果rob read的port使用在当T出现资源冲突，则设置
Allocating_inserted_uops	对于支持register partial update的Core，有时候需要人为插入新的uop同步partial flag的访问 比如： (1) movb 0x1, %al (2) movb 0x5, %ah <- no related with 1 (3) mov %rax, %rbx <- should wait 1, 2 complete 如果需要插入这种partial register的同步指令，则设置flag为1 NOTE: 当插入sync uop的时候，会导致当前allocate stall(不能再从q_alloc_block的SIM_Queue中获得任何uop) 这个标志表明sync的uop已经进入allocate模块，开始进行分配
Allocating_merge_flags_1_uops	同上，只不过dst register需要是eflags
Stick for 1 unfused uop	

Break_stat_event	当前没有使用
Std_dependency	load和之前的std之间有全地址的memory renaming关系，记录std的ROB ID
Sta_dependency	同上，记录与std相关的sta的ROB ID
Load_dpendency	记录load renaming关系的ROB id
Std_dependency_phytid	load和之前的std之间有全地址的memory renaming关系，记录std的thread id
Load_dependency_phytid	记录load renaming关系的load所在的logical thread id (从目前来看，一定是和当前load相同的logical thread id)
Retire_dependency	当Load和std之间有全地址memory renaming，且不能forwarding，那么load必须等到std结束，表明std的ROB ID
Retire_dependency_phytid	当Load和std之间有全地址memory renaming，且不能forwarding，那么load必须等到std结束，表明std的thread id
Partial_dst	当前的uop的dst是partial register，记录rat中dst的之前的映射，用于填写到rob.nsrcs[DEP_PARTIAL_INDEX]中，作为一个依赖，这个域仅仅在setting_seriesold_partial_register_updates=false的情况下使用
Partial_dst_phytid	同上，记录phythread id
Partial_flag	如果当前的uop需要写eflags，但是没有写入所有的flag标志，记录rat中eflags的之前映射，用于填写到 rob.nsrcs[DEP_PART_FLAG_INDEX]中，作为一个依赖，这个域仅仅在setting_enable_partial_flag_renaming=false的情况下使用
Partial_flag_phytid	同上
Std_depedency_s	load和之前的std之间有全地址的memory renaming关系，记录std的uop
Stall_reason	记录uop进行资源分配时，当不能进行分配后，导致allocate stall的原因 1. Scoreboard 与IF/VIF/DF等相关标志有关系 2. Fp_partial_flag 写入fc0-3 reg，当两者的写入者不同时stall 3. Fcw 写入fcw控制字 4. Brob Block rob，代表一次写入的rob的个数(setting_width)是否足够 5. Rob rob的个数是否满足当前uop请求的个数，即为1 6. Beuflushing a. backend开始做flush动作，allocate等待flush动作结束 b. backend必须等待mispredict的指令retire之后才能继续allocate 7. Br_checkpoint 8. Guaq 包含集中情况 a. Uaq不为空则stall，兼容之前的seriesold_style的机器 b. 如果uaq中已经没有空间，则stall 9. Color 10. Partial_reg 对于INT_REG, RBP-R15之间的通用寄存器,如果出现partial reg write/read的时候，需要插入merge uop a. 对于src reg 如果读取寄存器的width大于rename中寄存器width，需要插入同步指令 mov al, 0x5a / mov ax, 0x5a mov rbx, rax <- insert merge uop 特别的，对于 mov ah, 0x5a mov rbx, rax <- insert 2 merge uops，1 for al, 1 for ah，这里有点存疑，因为对于dst reg的处理已经保证了写入时的对齐属性 b. 对于dst reg 如果写入的是Byte寄存器，且之前的rename寄存器是partial reg，那么需要插入同步指令 mov ax, 0xa55a / mov ah, 0x5a mov al, 0xbb <- insert merge uop 当插入了merge uop之后，如果当前插入的sync uop还没有进入allocate模块，则当前的thread需要block其他的thread N(N=4) cycles 11. Partial_flags 对于shift_rotate的指令，不影响eflags 目前model中的eflags的组织情况 ALL_FLAG=0x8d5 Flag_group: CF, OF, SF/PF/AF/ZF, OTHERS 对于eflags的partial read来说， a. 如果读取的eflag标志不能由之前rename的flag group的子集覆盖，必须stall，等待之前的uop retire，比如 Inst1 修改ZF Inst2 读取SF, ZF 必须stall b. 如果读取的eflag标志完全来自于rename的flag group的子集，但是需要从多个group才能读取完整，必须stall，但是可以插入flag merge的uop进行eflag merge 当需要stall的时候，需要设置partial_flag_stall_cycle的值 12. Sb/Sab 当sb/sab中都没有空间，则stall 13. Lb 当lb中没有空间，则stall 14. Rs_full 当allocate模块中的uop越过uaq后，直接通过rs(scheduler)进行分配，如果支持uop操作的execport都无法提供足够的space，则stall 15. Uoptag 无法分配uoptag 16. Logical(指可以in flight在线的最大的architecture register，被rename过的)

	超过了设置的最大值 17. Rob_read 如果当前T需要读取的rob read多于设计，或者需要restrict顺序依次进行rob read，stall
Distance_stds	记录在同一个context_id下面的(预测分支路径上面)的在当前load之前的std的uop个数
Nb_lds	为num_older_loads的备份
Fwding_rid	从Predicted_fwding_rid赋值得到，这里有个问题没有考虑(load_forwarding问题，只考虑了std forwarding)
Fwding_phytid	当前physical thread id
Mrn_load_src	来自于uop的mrn_from_load，表明当前的uop memory renaming到load
Oracle_meet_mrn_cond	hit完美的memory renaming条件，满足如下条件 1. load的dst reg不是NULL 2. 不是lock load 3. 不是MSROM的uop或者MSROM完美执行 4. 不是fp指令或者enable mrn_on_fp选项 5. Enable SFB(store-forwarding-buffer)规则且fast_forward 或者 disable SFB规则且[std_addr, std_size) include [ld_addr, ld_size) 6. 是否允许int/fp之间的memory renaming
Is_load/is_fp/is_alu/is_sta/is_std/is_br	表明uop的类型
Is_fused_br	Fused uop是否是branch uop
Std_is_fp	全地址memory renaming的std是fp uop
Rob_req	当前uop需要几个rob entry，设置为1
Mrn_move_generated	表明是否有因为memory renaming而插入move uop指令，目前永远为0，setting_mrn_insert_moves的feature没有打开
Mrn_addr_move	当前没有使用
Mrn_int_mov	当前没有使用，是否是INT uop的mrn move
Mrn_fp_move	当前没用使用，是否是FP uop的mrn move
Oracle_fwd_rid	当Load和std之间有全地址memory renaming，std的ROB ID或者Load-load renaming发生后，load的ROB ID
Oracle_fwd_stdid	当Load和std之间有全地址memory renaming，std的thread id
Predicted_fwding_rid	Memory forwarding predictor预测的用来forwarding的ROB ID， memory forwarding predict仅仅是一个相对于当前store buffer的一个相对偏移
Fwd_from_load	Load data从之前的load forward
Fast_fwd	如果load和std全地址overlap，并且ld_addr == std_addr && ld_size <= std_addr或者enable perf_fwd，表明load可以快速从std获取data。这种情况对应于全地址的align forward 对于unaligned forward (前提：[std_addr, std_size) include [ld_addr, ld_size) && ld_addr > std_addr)，如下条件必须同时满足 1. Enable 32bit unaligned store forwarding或者std不是32bit store 2. Enable 128bit unaligned store forwarding或者std不是128bit store 3. 允许任意长度的unaligned forward或者naturally algined forward(std_addr % std_size == 0)或者std_addr 4B对齐
Can_mrn	通过memory forwarding predictor预测得到的memory renaming结果
Mrn_pred_bad_dist	Memory forwarding predictor预测的store buffer id没有ROB ID对应
Fwd_mispred	Memory forwarding predictor预测错误
Mrn_nomove	永远为false，表明mrn没有插入move uop
Seriesold_mob_hit_partial	表明load与之前的std之间存在partial地址(partial的地址范围由seriesold_partial_mob_match_mask决定)的data有关联，记录有关联的std的ROB ID(最近的std)
Hit_stack	表明memory renaming hit到stack上，目前没有使用
Move_count	永远为0，没有使用
Update_ctxt	表明当前uop要进行dest reg重定向
Num_older_loads	记录在同一个context_id下面的(预测分支路径上面)的在当前load之前的有memory renaming关系的load(如果enable setting_last_load_mrn选项，那么不管是否memory renaming，所有之前的load全部记录)
Mrn_load_fwd_found	是否存在和当前load存在load memory renaming关系的load，且load data可以forwarding到当前load 对于load forward来说，需要满足比较严苛的条件 1. Enable 完美的load forwarding选项 2. 找到之前有相关的load uop 3. Load uop的paddr和size必须完全相同
Mob_hit_partial	非常奇怪的判断方式，暂时没看明白
Stick 1T	
Block_alstall[threadnum]	如果allocate模块采用block allocate的方式(即每cycle Core支持的最大的allocate uop数量，即前端的发射宽度)，那么如果某类资源无法满足，则当前thread在当前T allocate stall
Uaqs_have_ready_uops[16]	对于不同的logical thread来说，uaq中ready的uop数量
Last_uop_was_fxch	Not used

Private Structure

Alloc_memreqtype_head	对于memory renaming需要占用uoptags的情况，每个std会占用一个memreqtype，这个是对应的指针
Allocate_memreqtype[64]	同上
Already_inserted_bubbles_for_moves	没有使用
Num_loadop_this_line	没有使用

Seriesold_partial_mob_match_mask	表明partial_mob_match的mask bits，等于 (1<<setting_seriesold_partial_mob_match_width[16]-1) & ~(setting_mob_linesize[16]-1)
Partial_stall_cycle	因为partial reg的问题导致的stall latency，delay = setting_partial_stall_latency[4]
Partial_flag_stall_cycle	因为partial flag的问题导致的stall latency，delay = setting_parital_flag_stall_latency[4]
q_inserted_uops	表明需要对于partial reg/flag插入 uop的时候，放入这个SIMQ
Rob_read_set	ROB entry的ARF bit，表明当前rob entry的dst value是否已经ready
Stdid_head	
Current_branch_color[MAX_THREADS]	当前的branch color
Phythread_sb_size	每个phythread的store buffer size
Phythread_ld_size	每个phythread的load buffer size
Last_alloc_time[MAX_THREADS]	上一次成功allocate的thread_cycle
Num_uoptags[MAX_THREADS]	每个phythread的uoptags的数量
Num_uoptags_addr[MAX_THREADS]	同上，没有使用
Num_uoptags_int[MAX_THREADS]	同上，没有使用
Num_non_bogus_allocated[MAX_THREADS]	不是错误路径上的uop的分配个数
Num_rob_block[MAX_THREADS]	Rob_block的个数 等于setting_rob_num/setting_width，表明一次性必须全部可以分配进入allocate模块
Sb_head[MAX_THREADS]	Store buffer的head指针
Sb_tail[MAX_THREADS]	Store buffer的tail指针
Stall_on_mrn_mispred[MAX_THREADS]	对于memory renaming的预测错误，需要stall的信息，目前没有使用，永远为false
q_bigflush_reclaim[MAX_THREADS]	对于retire阶段发送的bigblush的信息进入这个SIMQ，bigflush主要是因为写入控制寄存器导致的模式转换
Lock_predictor_table	Lock table的处理
Periodic_checkpoint_threshold	
Local_thread_priority	parking机制，表明下一个T可以使用allocate的thread ID
Thread_stalling_idq_thread	导致idq stall的thread ID
Last_cycle_idq_read_tsel	记录上一次获得执行idq_read的thread ID
Block_thread_select_tid	表明哪个thread需要独占allocate
Block_thread_select_count	当前某个thread需要独占allocate，阻止其他thread进行allocate，这个表明要独占的cycle数
q_rsrc_avail_to_idq_tsel[MAX_THREADS]	表明当前allocate的资源是否满足thread可以进行idq_read
q_uops_stalled_in_alloc[MAX_THREADS]	表明thread上一次的idq_read的uop还没有分配完成，必须stall idq_read
Last_cycle_rsrc_avail[MAX_THREADS]	记录上一个cycle的rsrc_avail的状态
Rsrc_avail[MAX_THREADS]	记录当前rsrc_avail的状态，可能根据当前的resource或是其他条件进行更新，表明当前cycle是否可以进行allocate
Pending_thread_wake[MAX_THREADS]	表明当前thread处于等待wake up的状态，与q_wake_thread配合使用
q_wake_thread[MAX_THREADS]	SIMQ，用于wakeup可以进行allocate的thread
Alloc_sleep_reason[MAX_THREADS]	allocate的logical thread sleep的原因 SLEEP_REASON_NONE 不需要sleep SLEEP_REASON_ROB ROB没有足够的空间 SLEEP_REASON_LB Load buffer没有足够的空间 SLEEP_REASON_SB Store buffer没有足够的空间 SLEEP_REASON_RS rs没有足够的空间(一次性都分配或者是只分配remaining) SLEEP_REASON_IDQ_EMPTY 前端没有解码的新指令 SLEEP_REASON_SCOREBOARD ？前端的scoreboard stall(和中断有关？) SLEEP_REASON_BR_COLOR ？Branch color的问题，解决什么问题
Rsp_cache_valid[MAX_THREADS]	Cached rsp的值是否是valid
Rsp_cache_valid_time[MAX_THREADS]	Cached rsp的valid的cycle点
Idq_read_active[MAX_THREADS]	表明thread是否可以进行idq_read
q_alloc_block[MAX_THREADS]	进行allocate的SIMQ
Orig_num_uopos_in_alloc_block[MAX_THREADS]	当前T初始在alloc_block的uop个数
Num_uops_in_alloc_block[MAX_THREADS]	当前T还在alloc_block的uop个数
Next_entry_in_alloc_block[MAX_THREADS]	当前T的allocate分配到alloc_block中第几个uop
Num_fxch_in_alloc_block[MAX_THREADS]	记录当前alloc_block中有几个fxch指令
Num_br_in_alloc_block[MAX_THREADS]	当前alloc_block中有几个branch指令
Last_locktable_entry[MAX_THREADS]	上一次分配的locktable entry的指针位置
Flag_stalled[MAX_THREADS]	操作EFLAGS是否导致stall

SIMQueue

SIMQueue	Size	Latency	Name	dataType	IsPtr	Description

q_inserted_uops[threadnum]	UOP_NSRC(10)*2	0		Int *	false	Only enable when partial register update serially, or partial flag renaming & merge
q_bigflush_reclaim[threadnum]	setting_bigflush_latency+1 (setting_bpmiss_latency-setting_fetch_to_alloc_latency-setting_alloc_to_exec_latency)	Setting_bigflush_latency (setting_bpmiss_latency[30]-setting_fetch_to_alloc_latency[16]-setting_alloc_to_exec_latency[8])		Int *	False	
q_rsrc_avail_to_idq_tsel[threadnum]	setting_alloc_rsrs_avail_to_idq_tsel_latency/setting_alloc_clock+2	setting_alloc_rsrs_avail_to_idq_tsel_latency[4]				
q_uops_stalled_in_alloc[threadnum]	setting_allocate_tsel_alstall_latency/setting_alloc_clock+2	setting_allocate_tsel_alstall_latency[4]				
q_wake_thread[threadnum]	1	Wake_latency (setting_alloc_rsrc_avail_compute_latency[2]-setting_alloc_clock[2]		Int*	False	
q_alloc_block[threadnum]	Max_uops_per_fused*setting_width	0		Int *	False	
q_reclaim_sb	Setting_max_rob_size	Setting_meu_reclaim_sb_latency[2]		Int*	false	
q_reclaim_lb	Setting_max_rob_size	Setting_meu_reclaim_lb_latency[4]		Int*	False	
q_reclaim_rob_block	Setting_max_rob_size	Setting_reclaim_rob_latency[4]		Int*	False	
q_reclaim_rob	Setting_max_rob_size	Setting_reclaim_rob_latency[4]		Int*	False	
q_reclaim_uoptags	Setting_max_rob_size	Setting_reclaim_rob_latency[4]		Int*	False	

Allocate需要分配的资源

- Rob
- Store: store buffer, sab
- Load: load buffer
- Uaq: uop allocated queue
- Schedule: schedule entries (RS)
- Uoptags:
- Br color

Implementation

Flow

allocate block: 表示每个cycle内从IDQ读到的需要占用ROB entry的uop个数

allocate的pipeline flow

idq_read

idq_read的前置条件

- idq没有被后面的allocate stall，这个stall表明后面的allocate会stall所有thread的idq_read，thread_stalling_idq_read (这个会被 allocate stall的唯一条件是需要allocate insert uop)
- q_fe_uop中有ready的uop
- rsrc_avail(表明当前cycle的thread是否因为某些allocate原因sleep，是否可以进行allocate)， uop_stalled_in_alloc(表明已经可以allocate的uop是否全部都allocate完毕)满足条件

将uop从decoder buffer读入到内部的buffer中，主要完成如下几个功能

- 对于branch uop，判断是否支持restrict_branch_alloc_per_clock选项，如果支持，检查当前cycle的branch个数，超过，不读取
- 对于fxch，如果碰到，且fxch_alloc_restriction设置，则fxch必须为当前T allocate的最后一条指令
- 将q_fe_uop的uop读入到q_alloc_block，实际上对于HW来说，应该是相同的queue

Reclaim

主要是和retire stage进行交互，接收来自retire stage的信息(SIMQ)，更新当前可以用于分配的资源

- 回收store buffer
- 回收load buffer
- 回收rob/rob block
- 回收uoptags
- 对share的logical processor而言，判断切换条件

Allocate_thread_select

选择当前cycle使用allocate的模块

- Rsrc_avail必须ready
- 如果没有任何一个thread的rsrc_avail ready，那么选择上个cycle进行idq_read的thread
- 如果某个thread需要独占allocate，那么直接返回，并且阻止idq_read

Allocate_update_thread_resource

处理所有的sleep的thread的状态更新，使其可以参与到下一个cycle的allocate的arbitration，主要检查sleep_reason是否已经可以meet了

Update_alstall_counters

更新所有allocate stall的counter，包括

- thread上的al_stall, al_stall_bubble
- Block_thread_select_count

Compute_block_alstall

查看当前使用allocate的thread是否真的可以进行allocate，对于rs/sb/lb/rob等是否进行block方式的分配，一个cycle内的必须全部都进入或都不进入

Allocate_early_check

allocate前期检查

- 当前选中的thread没有al_stall

- 当前需要进行分配的uop个数，没有超过发射宽度
- 如果uop要求结束当前cycle的allocate(fxch指令)，或是compute_block_alstall中的检查不满足要求
- 没有指令需要进行allocate，或者需要进行partial reg/flag read/write，但是q_inserted_uops为空(代表没有sync的uop插入)

Get_uop_to_allocate_from_frontend

拿到一个等待allocate的uop，两个来源, q_allloc_block(正常路径上的uop)和q_inserted_uops(针对partial reg/flag的sync uop)

对于fused uop的每个uop来说，执行如下步骤

1. Compute_memory_dependencies

   检查常规load uop memory renaming的在ROB中的std指令，这里主要包括几种情况

   a. Partial match
      i. 一种是load 或是 std跨mobline的情况，这种情况没有看明白，mob_hit_partial
      ii. Hit in 16bit, Load/std在一个mobline内partial hit，但是full address不一定相交， seriesold_mob_hit_partial
      iii. 出现上述两种情况，都需要load等到std retire之后才能执行

   b. Full match, setting_enable_forwarding[true]
      i. 对于load / std的起始地址相同，std size >= load size的情况，算作fast_fwd，可以实现store buffer forwarding的功能
      ii. 对于load被std内含的情况，算作unaligned forwarding的情况，如果enable的unaligned forwarding的各种处理情况feature
      iii. 否则，load依赖于store，等到std retire之后才可以执行
      iv. 如果没有hit任何的std，如果hit了old load，如果使能这个feature，load也可以依赖于old load
      v. 扫描所有已经retire，但是还没有写回的store，如果match，则标记load hit post-retire store

   c. 对于支持memory predicator的model，查看load是否被预测hit 某个std，并和full match中计算获得的真正依赖的std对比，看是否是mispredict

2. Merge_partial_at_exec if merge_partials_at_exec

   对于partial reg的写，如果使能merge_partial_at_exec的选项，那么将dst partial reg的parent作为一个源操作数放到uop中，1)本身原来的src中有与dst或是dst.parent相同的reg；2)单操作数，直接将parent作为一个源操作数

3. Signal_alstall

   在真正进行allocate和rename之前，进行allocate stall的检查，主要有如下几类的stall的原因(这个stall只影响本次allocate的行为，是否导致thread sleep视情况而定)

   a. Scoreboard --- Sleep thread
      与IF/VIF/DF等相关标志有关系, wait_on_scoreboard_alloc[true]
   b. Fp_partial_flag
      Merge fc0-3 reg，当fc1和fc023的写入者不同且enable fp_partial_flag_stall feature[no]
   c. Fcw
      写入fcw控制字时，如果要求顺序写入，且达到最大的并行写入的number
   d. Brob --- Sleep thread
      Block rob，代表当前T一次写入的rob的个数(setting_width)是否足够，allocate_block_rob[no]
   e. Rob --- Sleep thread
      rob的个数是否满足当前uop请求的个数，包括正常需要allocate的uop，即为1和因为memory renaming导致需要加入的move uop，在现在的Model实现中，这个mrn_move_generated永远为0， setting_mrn_insert_moves[no], setting_mrn_fused_moves[no]
   f. Beuflushing
      i. backend发现mispred，设置beuflush，准备开始进行pipeline flush，allocate等待flush动作结束，enable_beuflush_alloc_stall[no]
      ii. backend必须等待mispredict的指令retire之后才能继续allocate，alstall_until_mispred_retire[no] (branch X mispred after younger branch Y mispred, then allocate wait branch X retire)
   g. Br_checkpoint
   h. Guaq
      包含几种情况
      i. Uaq不为空则stall，兼容之前的seriesold_style的机器， setting_uaq_empty_alloc_stall[yes], but modeled CPU not include uaq
      ii. 如果uaq中已经没有空间，则stall
   i. Color --- Sleep thread
      表示对于mispred的branch来说，有几笔mispred的branch可以不stall allocate。如果当前需要flush pipeline，且当flush pipeline的时候，不能立即回收bogus分配的资源，且当前的branch color不是最新的branch color，stall
   j. Partial_reg/partial_flags
      对于插入了sync uop的uop，如果partial_merge int reg或是partial merge flag，那么需要设置partial_stall_cycle = thread_cycle + setting_partial_stall_latency[4]
      对于INT_REG, RSP-R15之间的通用寄存器,如果出现partial reg write/read的时候，需要插入merge uop
      当插入merge uop之后，如果当前插入的sync uop还没有进入allocate模块，则当前的thread需要block其他的thread N(N=4) cycles，并独占allocate模块
      Setting_seriesold_partial_register_updates[no]
      i. 对于src reg
         如果读取寄存器的width大于rename中寄存器width，需要插入同步指令

         ```
         mov al, 0x5a / mov ax, 0x5a
         mov rbx, rax          <- insert merge uop
         ```

         特别的，对于

         ```
         mov ah, 0x5a
         mov rbx, rax          <- insert 2 merge uops，1 for al, 1 for ah，这里有点存疑，因为对于dst reg的处理已经保证了写入时的对齐属性
         ```

      ii. 对于dst reg
         如果uop的source没有出现partial reg的问题，写入的是Byte寄存器，且之前的rename寄存器是partial reg，那么需要插入同步指令

         ```
         mov sp, 0xa55a / mov spl, 0x5a
         mov esp, 0xbb          <- insert merge uop
         ```

      iii. Partial_flags
         Setting_enable_partial_flag_renaming[yes]
         uop需要读取eflags
         对于shift_rotate的指令，不影响eflags
         目前model中的eflags的组织情况， ALL_FLAG=0x8d5， Flag_group: CF, OF, SF/PF/AF/ZF, OTHERS
         对于eflags的partial read来说，
         1) 如果读取的eflag标志不能由之前rename的flag group的子集覆盖，必须stall，等待之前的uop retire，比如
            Inst1 修改ZF
            Inst2 读取SF, ZF 必须stall
         2) 如果读取的eflag标志完全来自于rename的flag group的子集，但是需要从多个group才能读取完整，必须stall，但是可以插入flag merge的uop进行eflag merge
         当需要stall的时候，需要设置partial_flag_stall_cycle的值
   k. Sb/Sab --- {sb} Sleep thread
      当sb/sab中都没有空间，则stall

l. Lb --- Sleep thread
当lb中没有空间，则stall
m. Rs_full --- Sleep thread
Setting_bypass_uaq[yes]
当allocate模块中的uop越过uaq后，直接通过rs(scheduler)进行分配，如果支持uop操作的execport都无法提供足够的space，则stall
n. Uoptag
Setting_alloc_stall_for_uoptags[no]
当lb/sb/sab没有stall，无法分配uoptag
o. Logical(指可以in flight在线的最大的architecture register，被rename过的)
Setting_max_alive_logical[0]
超过了设置的最大值
p. Rob_read
Setting_enable_rob_read_ports[yes]，表明issue之前需要读取rob
Setting_rob_read_ports[3]，一个T内最多支持3个不同rob entry read
Setting_rob_read_restrict_ports[yes]，
如果当前T需要读取的rob read多于设计，或者rob read的时候出现资源冲突(不同uop在用一个T占用相同的port，或者同一个uop的不同src占用相同的port)，stall
Compute_rob_read_stall
对于rename之后的architecture register，会rename到某个rob entry上面，对于某个uop而言，如果当前所需要的src已经rename了或者rename了，但是retire了(new value在rrf中)，则在issue之前需要先进行register read
1) 首先，统计当前uop的所有src，对于目前的uop设计，uop的src不会超过3(fused uop)/2(unfused uop)
2) 一个uop出现>=2个相同的src，则只处理一次
3) 获得每个src rename对应的rob entry，对于已经rename到某个rob entry的src
a) Setting_dst_rob_ready_reduction[yes](表示 src依赖的rob是当T分配的，则src不用rob read) 或是 setting_rat_rob_read_reduction[yes](表明当前src可以通过依赖的rob获得需要的数据)
i) s_remove_arf_bit_write[no](实验feature，去掉rob上的arf bit，直接根据一个固定的latency判断是否需要进行rob read)
ii) 查看entry上对应的ARF bit(表示当前rob对应的写value是否已经ready)，如果已经set，则需要进行rob read
b) 总是进行rob read
4) 判断当前的src在当前T是否已经进行过rob read，如果进行过，不再read，可以来自于rob/rrf
5) 对于rsp的read，如果rsp_caching使能且rsp_cache_valid，则不进行rob read。 Setting_rsp_cacheing[false]
6) 按照既有的port read的连接关系分配rob read，如果出现冲突，则结束
Src_num = assign_source_numbers() 按照规则给Uop的每个src进行编号，见后面rob read port分析
Uop_slot表示当前allocate uop属于当T进行allocate的uop中的第几个， first_uop_slot表明当T第一个进行分配的uop属于setting_width中的哪一个
按照uop_slot和src_num进行read port的绑定 setting_robrp_binding_scheme[2](3 128-bit read port)
7) 记录进行过rob read的src
8) 对于rsp来说，如果当前src读取的是完整的rsp，且没有出现资源冲突，则rsp进行缓存
9) 对于出现memory renaming依赖的load而言(如果memory renaming的条件非常完美)，则load可以直接从依赖的std的dst寄存器进行read，这个也需要占用read port
10) 如果需要stall allocate，对于当前运行allocate的thread，设置thread[tid].al_stall_bubble = setting_alloc_stall_bubble[0]，表明多少个T后thread才能继续进行allocate

对于预测memory renmaing到std的load且setting_mrn_unfuse_loadop[false]且是fusing的parent，则unfuse
a. 对于原来的parent，设置mrn_unfused_parent
b. 对于新的parent，设置mrn_unfused_child

如果当前T需要进行pipeline flush，但是branch_color还有空间可用，则分配新的branch_color，不stall allocate

处理br_checkpoint

lock相关的处理
Setting_do_seriesold_base_locks[no](基本的串行lock的处理方式) || (setting_do_CPU1_fast_locks[yes] && setting_CPU1_checkpoint_penalty[0](CPU1 checkpoint penalty)
a. 对于lock load(加了lock前缀或是带有lock属性的uop)，加入当前logical的lock_table中，并标记当前load会block后面所有的load，记录下当前lock_load的执行信息到当前thread的结构中，更新当前的lock_table的指针，当前uop记录对应的lock_table中的entry
b. 对于支持setting_do_CPU1_fast_locks的情况，
i. 如果num_locks == setting_number_CPU1_fast_locks[1]，且setting_do_single_CPU1_fast_locks[1](只允许一个CPU1 fast lock每个T)，如果lock_predictor_table中没有对应的lock_load记录，标记当前lock_load为speculative_lock
ii. 如果!setting_do_single_CPU1_fast_locks， 且lock_predictor_table中没有对应的lock_load记录，标记当前lock_load为speculative_lock
c. 当前的lock load不是错误路径上的指令且没有错误，那么标记当前thread需要look_for_unlock_store
d. 对于当前的thread，如果有lock_load存在，且设置了look_for_unlock_store
i. 如果是sta uop，则标记sta uop为sta_unlock，标记sta的lock_table entry 指针为上一个lock_load所指向的
ii. 如果是std uop，则标记std uop为allocator_unblocking_store，同时取消thread上的look_for_unlock_store标记

初始化当前的rob entry

给当前uop分配资源，对于不同的uop类型，分配不同的资源
a. Portout/storeaddr
i. 分配store buffer
ii. 分配store buffer id(sab)，setting_disamb_disable[true]，记录当前uop信息到allocate模块的sab buffer中
b. Portin/load
i. 分配load buffer
ii. 记录最近一次分配的sab
iii. 记录memory renaming的信息，如果和某个std存在memory renaming，记录最新一次std所在的store buffer的位置，标记当前load和std指令所需要的memory renaming信息
c. Storedata
i. 记录最近一次分配的sab
ii. 分配store buffer
iii. 记录store buffer和rob entry间的对应关系
iv. 进行uaq的分配，如果当前uop在rat执行完毕，则不需要分配到uaq中

d. Others
   i. 进行uaq的分配，如果当前uop在rat执行完毕，则不需要分配到uaq中
e. 对于所有类型uop，
   i. 如果没有设置Setting_bypass_uaq[yes]， 分配进入portingbinding描述的uaq中，push进入q_uaq SIMQ
   ii. 否则，setting_pb_dynamic_binding_for_ldsta[no](对于ld/sta而言，动态执行port绑定)，执行port绑定(setting_bind_execport_at_alloc[yes])
f. 对于所有类型uop，分配rob entry(如果当前uop不是fused uop或是fused uop的第一条指令)

更新allocate的scoreboard

a. 设置al_scoreboard为当前uop的number，如果当前uop为SETSCORE且(不支持flag_renaming或是!NOSCORE_FLAGRENAMING)

更新uoptag信息

a. 如果dst不是NULL REG，更新uoptags的number

映射重命名的src操作数

将rat映射中存在的映射关系copy到对应的ROB entry中，这样rob就知道src需要到哪个rob中进行索引，对于某个src的映射的reg是因为之前的ld通过memory renaming获得的，那么进行标注

重命名memory

对于memory renaming的load更新rob中的memory renaming信息

对于使用setting_mrn_uoptags[0]，即memory renaming需要占用uoptags，那么在memory predictor中预测会进行forward的std会分配一个entry在allocate_memreqtype中，如果allocate_memreqtype没有预测的std结果，则预测结果失效

如果需要memory forwarding，则设置mrn_fwding_rid为对应的std的rob entry

★ dst重命名

a. 对于fxch的指令，直接修改rat映射表上的映射关系(exchange一下)，然后结束本次allocate　　fxch_alloc_restriction[yes](fxch is last in alloc phase)
b. 对于load-op的fused uop，如果load-op的load和std memory renaming，并且load-op已经unfused，如果setting_mrn_load_op_src_update[no](memory renaming的load需要的src直接从std的src获取)，那么将load-op中的op的src重命名到std的src上，对于已经retire的std，则src映射到NULL
c. 对于memory renaming到load的情况，这里认为是false path，处理的方式是将当前Load的dst直接重映射到memory renaming的load所在ROB，代码的条件写的有问题(多写了1个!)
d. 对于memory renaming到std的情况
   i. 如果setting_mrn_load_op_src_update && load-op已经unfused，且是load-op中的load，不做任何处理
   ii. 对于不需要进行partial_series_update的INT_REG/eflags，如果本次dst是partial update，保存之前的dst的映射到info中，对应naive的partial register update
   iii. 对于setting_mrn_insert_moves[no](对于memory renaming的uop插入move)
      1) False path， 生成新的MRN_MOV的uop，dst=uop_dst, src=std的src，并更新对应的ROB entry，并于memory renaming的load形成新的fuse关系(micro-fusing)
         a) 不支持seriesold update partial reg的情况，标记新的mov指令的dep_partial_reg/flag标记
         b) 支持seriesold update partial reg的情况，更新dst对应的rat到生成的move uop所在的ROB hawk: 这个方法看起来有些功能错误，因为std, ld之间可能插入别的core的store，这样如果直接从std拿data，不一定是最新的data，这个疑问本身有问题
      2) 按照不同的memory_dep_level的设置，进行当前load的dst重命名， setting_mrn_dep_level[DEP_LVL_SRC]
         a) DEP_LVL_SRC
            使用std的src对dst进行重命名(如果std的src有映射，且没有retire)，映射完成后，分别标记dst和std的src表明经过memory renaming的重命名
         b) DEP_LVL_STD
            使用std本身进行重命名，过程同上
         c) DEP_LVL_LOAD
            使用load本身进行重命名，过程同上
      3) 对于上述两种情况，去除memory renaming的依赖，去除ii中保存的映射，对于setting_mrn_sta_spec[false](allow renamed load go ahead of STA)，同时去掉sta_dependency的依赖
e. 不属于上述的情况，进行dst重命名到当前uop所在的ROB
   i. 保存之前的dst映射到info中
   ii. 重命名分为几种情况
      1) Setting_bypass_zero_marks[no](对于dst等于0的情况)，取消重命名，最新值指向rrf
      2) Setting_bypass_moves[0](对于move指令)，直接映射到src上
      3) 否则映射到当前ROB
   iii. 更新RAT的映射表
      1) 对于使用partial_register update方式的CPU来说，进行partial reg的串行化更新，如果需要插入partial merge的指令，则插入
      2) 否则直接更新RAT对于的映射表项
   iv. 更新ROB中的ARF bit，对于enable rob_read_port的选项
   v. 更新ROB中的表项，主要针对memory renaming和partial reg write
   vi. 根据memory renaming的forwarding条件，更新需要forwarding的std的forwarding type，所有的情况不考虑hit到已经retire的store
      1) STD_FFWD_NOMRN
         Store 进行fast forward，但是没有进行memory renaming
      2) STD_FFWD_MRN
         store进行fast forward，且进行了memory renaming
      3) STD_SFWD
         Unaligned forwarding或是与std有dependency关系
   vii. 更新EFLAGS对应的RAT表项

Update_rob_entry

主要更新branch color信息

Update_rob_block_count

如果uop是按照rob block方式分配，计算rob block的counter　rob_block = rob_num / setting_width

Update_fp_control_dsts

Alloc_update_fe_scoreboard

Allocate_std

做些必要的检查，有些指令可能会插入uop_merge sync uop(最多2个)；查看对应的sta的访问属性(对于streaming类型的操作如NT)，设置memory访问属性为WC

对于std指令，如果memory renaming的feature需要占用uoptag且预测为memory renaming，则更新对应的uoptag为std的ROB ID

Copy_rat

对于mispred的uop，copy rat的信息到对应spawn_cid中

Dequeue_frontend_uops

从q_insert或q_alloc_block中pop一个uop

Compute_thread_switch_blocking

如果当前正在执行sta uop，则设置al_prevent_switch为true

Check_if_next_uop_is_fused

检查q_insert或是q_alloc_block中的uop是否是fused

Allocate_into_rs

uop分配到RS(scheduler)中，对于fusing uop处理方式不同，分为

1) New model　只有当所有的fusing uop的所有Uop进入RS后，才会更新unfused counter
2) 原始model　对于std和fusing uop不计算unfused counter

Update_lrob_for_mrn_moves

对于使能memory renaming move的uop，因为是插入的指令，所以需要对ROB index递增

Update rob index

当前T的thread进行allocate分配之后，

Allocate_update_thread_prority

当setting_allocate_tsel_perfect[no]，且当前T执行allocate的thread allocate了uop或是插入了新的uop，那么切换当前local_thread_priority到下一个thread

Allocate_update_next_alloc_block_entry

更新下个T待分配的uop的个数，主要查看next_entry_in_alloc_block[tid](表明q_alloc_block中的uop处理到第几个)，如果q_alloc_block中的uop全部处理完毕，则开始下一轮的allocate，否则当前thread不能进行allocate，必须等到q_alloc_block处理完毕，例外，设置了setting_allocate_reuse_block_entries_after_stall[no](表明即使出现了allocate stall，依然可以进行idq_read，这时q_alloc_block会持续增大，相当于frontend和allocate之间增加了隐形buffer，buffer大小不确定)

ROB entry的依赖关系构建

	0..UOP_NSRC	DEP_STA_INDEX(UOP_NSRC)	DEP_PARTIAL_INDEX	DEP_PART_FLAG_INDEX	DEP_STD_INDEX	DEP_RET_INDEX	MAX_SRC
Nsrcs[]	Function model上根据uISA对应的每个uop可以建立的uReg依赖	对于load uop，Memory renaming情况下，load依赖的full match std对应的sta的ROB ID	针对没有enable seriesold_partial_reg_update的情况，这种情况下，所有的partial register访问全部会映射为对full register的访问，所以对于同一个reg_name的partial/full访问会建立依赖关系，这个域针对一种特殊情况作处理 mov rax, 0xa5a5　(1) mov al, 0x5a　(2) <- 因为是write，所以对于RAT来说会将rax重命名到当前的指令，但是这个指令只操作了al，所以高位部分还在上一条指令中 所以，对于这种情况，需要(2)在当前域上标记依赖(1)	同DEP_PARTIAL_INDEX的情况，这种情况下需要标记之前的dst到当前域建立依赖关系	对于load uop， Memory renaming情况下，load依赖的full match std对应的ROB ID	对于load uop，memory renaming情况下，load依赖的std对应的ROB ID，这个依赖表明对应的std必须retire之后，load才能执行 存在两种情况 • Partial match，则load必须等待Partial match的std • Full match，load和std之间无法建立任何forwarding关系	Sentinel标志

RAT的机制和结构

在model中采用的重命名机制是重命名到ROB entry

```
Mapping rat[Uop_Regnum_MAX];                    // 包括所有的arch、uarch的寄存器列表，包括alias寄存器(比如rax, eax, ah, al)
Struct {
   int ptr;              // 重命名到的ROB ID
   uint32 mask;          // 操作的大小和mask
} partial_flags[6];                             // EFLAGS partial相关，rat中应该已经包括的EFLAGS的定义
```

mapping的结构

```
Struct Mapping {
   int pointer;                                 // 全寄存器重命名到的ROB entry
   int l_ptr;                                   // x寄存器重命名到的ROB entry
   int h_ptr;                                   // xh寄存器重命名到的ROB entry
   int x_ptr;                                   // xx寄存器重命名到的ROB entry
   int is_zero;                                 // 当前寄存器的指是否是0，全寄存器
   int is_x_zero;                               // 同上，xx
   int is_h_zero;                               // 同上，xh
   int phytid;                                  // 当前寄存器重命名所对应的phythread ID
?  int reg_num;                                 // 需要再看下
   bool src_mrned;                              // 表明当前RAT entry的值是否是因为memory renaming猜测获得的
   bool local_uoptag_invalid;                   // 设置为false，暂时没有使用
};
```

两种类型的partial register renaming方式

1. Seriesold_update style (setting_seriesold_partial_register_updates[true] || (setting_enable_partial_flag_renaming[true] && setting_enable_partial_flag_merge_uops[false]))

   Seriesold_partial_register_write

   进行register的更新，根据当前写入的register的宽度和之前的register重命名状态进行分析

   Seriesold_partial_register_read

   根据register重命名状态和当前的读入宽度返回重命名的结果

   Seriesold_insert_partial_stalls

   对于当前的register read，是否需要插入merge uop进行register的合并

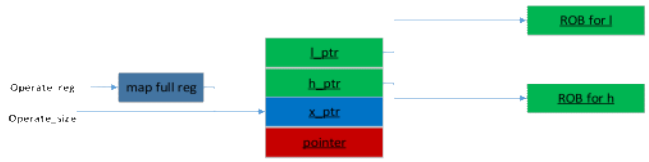
   Insert_partial_stall_on_write

   在进行register写入的时候，是否需要插入merge uop进行register的合并

   Seriesold_insert_merge_uop

   按照对应的规则，插入register merge的uop，并放入q_insert SIMQ中

Seriesold_partial_register_update



原则：对于 partial reg的renaming只能有一个是有效的，
意味l_ptr,h_ptr/x_ptr只能是之一有效

- Partial register的合并可以在两个点发生：read时或是write时，目前在read时进行合并
- 对于register read，如果当前read register的宽度大于最后rename的宽度，需要插入merge uop进行partial register合并；否则，读取当前rat中的信息，哪个映射不空，返回哪个
- 对于register write，更新规则如下(有一个特定的featuer：zero_mask register)
  - 当前的write宽度 >= rat中已有映射的宽度，设置对应的映射指针，清除小于当前宽度的映射
  - 当前的write宽度 < rat中已有映射的宽度，如果已有映射是zero_mask，设置已有映射为新的映射；否则对于已经是partial mapping的情况，将已有的partial mapping映射为宽度更小的映射
  - 如果当前uop的dst register是zero_mask，根据当前rat中的zero_mask标志标记对应映射的zero_mask flag
  - ? 在model 中实际处理中，当发现某个uop的dst register是partial register，会首先查看rat当前映射是否是partial mapping，如果是，插入merge uop。个人觉得这里有些over-design，上面的设计已经能保证正确性

2. Naïve style
   这种方式，对于partial register的访问同样转换到full register后映射，这样，partial register的所有访问之间都有依赖关系，是串行更新的，一种特殊需要考虑的情况是，当前的register write是一个partial write，这时需要将之前的register映射作为当前uop的依赖

Naïve update

- 对于register read， 查找对应的RAT表找到full reg对应的ROB ID，直接标记依赖
- 对于register write
  - Full register write，直接更新RAT表为当前 ROB ID
  - Partial register write，标记目前的映射的ROB ID为当前ROB ID的一个依赖项[DEP_PARTIAL_INDEX]，更新RAT表为当前ROB ID

Partial register update stall的处理

Partial Flag renaming

Partial_flags_read
读取uop需要的flag分组，保证只有一个flag分组有效
Partial_flags_write
uop更新的eflags中，和所有partial flag分组有overlap的分组全部rename到当前uop所在的ROB，对于rol/rcl/ror/rcl等移位指令，因为eflags的更新依赖于cl的值，所以这种情况下，表明当前分组所有flag标志无效(这里的无效指的是必须stall等待uop更新eflags)
Insert_partial_flags_stall
在uop需要读取partial flag的情况下，如果当前分组重命名的partial flag有效标志不能覆盖uop需要的标志或者uop需要从多于一个分组读取eflags标志，则表明partial eflag stall。如果支持partial flag merge，对于需要从多于一个分组读取eflags的标志的情况，可以插入一个mov uop来merge EFLAGS标志
Partial flag的分组情况(最大分组为6个)，每个分组可以单独进行rename

CF	OF	SF/PF/AF/ZF	Others

对于flag read， 有如下若干情况需要处理

- 如果partial flag的处理需要完全串行化(等待ROB空)，需要等待back-end为空
- 如果uop命中的某个partial flag register需要读取的bit多于当前partial flag有效修改的bit，需要等待当前partial flags register retire之后才能执行
- 满足2的条件下，如果uop需要读取的eflags的bit跨越N(N> =2)个partial flags register
  - 不支持enable_partial_flag_merge_uops，必须等待N-1个partial flags retire之后才能执行
  - 支持enable_partial_flag_merge_uops，依次插入若干条uop mov指令，将所有uop需要读取的eflags bit分布的partial flag register逐一递增merge，最后一个插入的merge uop的write_flags标志包含所有uop需要读取的eflags标志，并将所有的eflags的rename全部映射到最后的merge uop上

对于flag write，见partial_flags_write

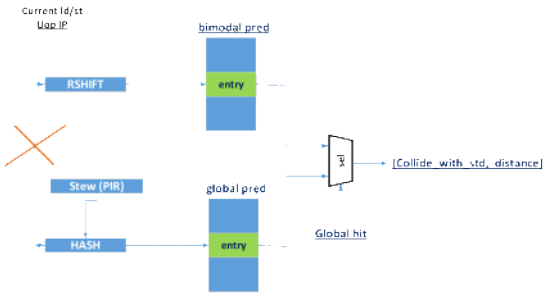
Memory forwarding predictor

MFP的预测主要是预测的是可以forwarding的load/std之间的关系
MFP Entry

Lip	tag信息，代表ld/st的IP va地址, 只在global_pred中使用
Distance	ld和依赖的st间的distance
Confidence	饱和计数器，2bit饱和计数器，confidence>2表示strong_confidence
Plr	Pipeline recorder
Cycle	更新时间
Collide_with_std	是否和某个std 冲突，可以进行forwarding
Violation[MAX_ALIAS(8)]	表明是否和std间有冲突
Mrn_from_store	当前entry是否曾经通过std forwarding

MFP的各个参数

Model	Structure	Index function
Bimodal pred	Set(4096), Ways(1), ENTRIES(1)	Right shift, xargv_mtf_mrn_shift[2](表示某个ld可以和多大范围内的std预测产生依赖)
Global pred	Set(1), Ways(0), ENTRIES(1)	Hash function,



MFP的访问

在model中是在decode结束之后，立刻调用MFP的接口来探测load的std之间是否存在memory renaming的关系，这个时候还无法发现是否存在forwarding的关系。MFP预测ld_mrnable，那么等到allocate阶段会设置到load对应的ROB entry中。MFP的访问，目前只有bimodal方式，这个方式是一个直接相连的cache。当confidence的饱和计数器>2，且当前load不是lock_load的时候，采信MFP中collide_with_std的bit

MFP的更新

在load retire的时候，更新MFP，更新的时候需要如下信息：[ld对应的uop， 实际load是否与std mrn，与sta是否冲突， 距离当前st_buffer的distance，实际的是否存在memory mrn]；更新规则如下：

1) MFP对应entry 预测为mrn，如果load存在memory mrn，那么检查dist和预测结果是否正确，都正确，confidence+1，剩余情况，则confidence计数器清0
2) 预测为no-mrn，如果load与std mrn，那么confidence计数器清0，否则confidence+1

Allocate Info上面的各种memory mrn的标志

Store Partial Match

mob_hit_partial	看起来像是标志std/load 跨mob_linesize的情况处理，但是判断条件感觉有问题
seriesold_mob_hit_partial	std/load的pa的局部地址(16bit)一样，且mob_linesize之内的byte之间有overlap

Load Match

load_dependency/load_dependency_phytid	forward load的ROB ID和PHYTHREAD ID
nb_lds	forward load和当前load的load 距离(隔了多少个load), 对于setting_last_load_mrn(依赖最老的load)，距离很远
mrn_load_fwd_found	是否已经找到一个forward load，主要是一个bool 变量控制找到
fwd_from_load	表明load从load forwarding

Store Full Match

std_dependency/std_dependency_phytid	表明和load有memory renaming关系的最近的std uop ROB ID和PHYTHREAD ID，且是full match
sta_dependency	表明有mrn的std关联的sta uop的ROB ID, full match
retire_dependency/retire_dependency_phytid	表明有mrn的std必须要到retire之后才能继续执行load，包含如下几种情况： 1. load与std match，但是不符合forwarding条件 2. load与std存在partial match，且partial match的std不是full match的std，那么=partial match的std

MFP predicted

can_mrn	load是否mrn到之前的std(预测的结果是之前mrn的store到当前STORE BUFFER的距离)
predicted_fwding_rid	预测的std的ROB ID
predicted_fwding_sbid	预测的std的STORE BUFFER ID
mrn_pred_bad_dist	预测mrn，但是预测std已经store back
fwd_mispred	MFP的预测结果与功能执行结果不match

Forwarding相关

fast_fwd	std.pa=load.pa，且std.size >= load.size
oracle_fwd_rid	load forwarding的ROB ID，可能是ld或是std
oracle_fwd_stdid	load forwarding的STORE BUFFER ID
oracle_meet_mrn_cond	表明当前的load是否满足memory renaming的条件 1. load的dst 寄存器不能使NULL REG 2. 不能是lock load 3. 不是来自于MS的uop或者MSROM的uop完美执行 4. 允许FP uop操作上的memory renaming 5. 使用SFB(store-forwarding buffer)规则，且是fast_fwd 或者 (不使用SFB规则且 (完美forwarding或者load.pa >= std.pa && load.pa+load.size <= std.pa+std.size))
fwding_rid / fwding_phytid	表明load预测从哪个std forwarding
mrn_load_src	表明load从哪个load fowarding
mrn_nomove	没有使用

Simuop上面的各种memory mrn的标志

MFP预测结果

ld_collide_with_std	MFP预测load与std之间有forwarding
ld_mrnable	MFP预测load会有memory renaming, 但是有相应的条件需要判断， memforward_predict.cc:393

需要判断的结果

mem_renamed	load已经完成std的renaming，体现在了reg renaming table上，而不仅仅是标记load / std间的dependency关系
unaligned_fwd	非perfect forwarding或是pa相同，store_size >= load_size，即不是fast_fwd
fwd_mispred	forwarding预测错误，实际上就是要fowarding的entry与真实情况不符合
mrn_mispred	mfp的forwarding预测错误，前提，mfp预测memory renaming，但是forwarding entry预测有问题
pred_sta_indep	mfp预测load与std之间没有forwarding，但是confidence很高，说明和某些independ sta有依赖关系
ld_mem_viol	

统计用

mrn_nomove	=info->mrn_nomove, 没有使用，本意表示当不能插入新的mov指令时，memory renaming 不能进行

mrn_hit_postret	memory renaming，与下一项是相同含义
ld_hit_postret	表明当前load与还没有回写的std有关联
mob_partial_replay	mob_hit_partial != -1 && mob_hit_partial != std_dependency
mob_partial_noreplay	std_dependency == -1 && mob_hit_partial != -1

ROB上面的各种memory mrn的标志

fwding_rid	= info->predicted_fwding_rid
bad_dist	=info->mrn_pred_bad_dist
can_mrn	=info->can_mrn
oracle_fwd_rid	=info->oracle_fwd_rid
oracle_fwd_stdid	=info->oracle_fwd_stdid
fwd_from_load	=info->fwd_from_load
fast_fwd	=info->fast_fwd
mrn_from_stack	=info->hit_stack
mob_partial_hit	info->seriesold_mob_hit_partial && info->seriesold_mob_hit_partial != info->std_dependency
Collide	表明std与后面的load之间存在forwarding
Int2fp	forwarding的类型
Fp2int	同上
fwd_from_int	load forwarding从std所在的int scheduler，这里有点问题，和fp2int如何解释?
std_src_of_ffwd(fast-forward)	=info->oracle_meet_mrn_cond
dependent_on_older_stas	在预测时，表明load依赖于之前的sta

Memory renaming的各种情况 (compute_memory_dependencies())

1) 首先，只有load存在memory mrn，且load不能是uncachable或是sw_prefetch指令
2) 从新到老扫描ROB中的ld/std uop，检查memory renaming情况
   a. 对于load，情况简单，只支持一种memory renaming和forwarding， fwd_ld.pa==ld.pa && fwd_ld.size==ld.size
   b. 对于std，情况复杂很多
      i. Partial match
         1) mob_hit_partial model中也没有enable这条路径， 这里判断有些不清楚，看起来是想处理跨mob_linesize的情况，但是判断条件貌似不对
         2) seriesold_mob_hit_partial (std.pa&partial_addr_mask == ld.pa&partial_addr_mask) && (std.mob_byte_mask & ld.mob_byte_mask)，认为是partial match，即在mob_line的范围内两者overlap，且pa地址的低Nbit(不包括mob_line_size对应的地址bit)一致
      ii. Full match
         load与最近的某笔std的pa出现overlap，即认为两者存在memory renaming。对于full match而言，从功能上来说，load确实需要从memory renaming的std forwarding数据或者等待其完成
         1) Fast_fwd
            Ld.pa==std.pa && ld.size<=std.size 或者perfect forwarding
         2) Unaligned_fwd
            ld可以从std进行forward，但是这种forward不是aligned forwarding
            有若干条件限制：
            a) 支持非aligned的forwarding
            b) Ld.pa>std.pa && ld.pa+ld.size <= std.pa+std.size
            c) 支持32bit store forwarding或者std.size != 4
            d) 支持128bit store forwarding或者std.size != 16
            e) 支持任意的store forwarding 或者 支持naturally_aligned的方式(std.pa % std.size == 0) 或者std.pa % 4 == 0
         3) Retire_dependency/phytid
            不满足任何forwarding的条件，只能等待std retire
      iii. No match
         表明在程序执行过程中，该笔load不会和任何std发生memory renaming
   c. 对于之前已经检测到partial match的情况，如果full match的std和partial match的std不同，则retire dependency建立在partial match的std上
3) 对于已经retire的std [store_head, store_tail]，查看load是否与store queue中的std之间产生memory renaming，对于这种情况，仅仅支持2种情况
   a. mrn_hit_postret 表明load与某个已经retire的std之间存在pa间的overlap
   b. fast_fwd 和上面描述的fast_fwd条件一致
4) 对于使用了MFP的情况
   a. MFP预测load mrn，获得预测对应std的ROB ID，如果预测的std已经store back，则预测失败
   b. MFP预测load no-mrn，预测的std的ROB ID赋值为full match情况下的std ROB ID
   c. 针对a, b，查看MFP预测的时候正确(最终结果)

对于预测为memory renaming的load-op形式的fusing uop，需要进行unfused的处理，将load进行unfuse，而其他的uop还是处于fuse状态。标记load为unfused_parent, 其他的uop为unfused_child。问题：为什么需要进行unfused???

Allocate导致的sleep原因

SLEEP_REASON_NON	
SLEEP_REASON_ROB	block_rob或者rob都没有可用的空间可以使用
SLEEP_REASON_LB	load buffer没有可用的空间
SLEEP_REASON_SB	store buffer中没有可用的空间(这个主要针对std，对于sta会分配sab)
SLEEP_REASON_RS	uop可以分配的scheduler中没有RS的可用空间，或者在block level分配下，scheduler的空间不能满足所有的uop数量
SLEEP_REASON_IDQ_EMPTY	在alloc_block和q_insert的SIMQ中都没有等待分配的uop存在
SLEEP_REASON_SCOREBOARD	等待fe_scoreboard[tid]不被占用
SLEEP_REASON_BR_COLOR	allocate有一个需要处理的branch mispred，且没有可用的branch color可以使用(最多4个branch color)

ROB port read的方式

Source number的分配问题

对于model而言，常规的uop最多只有2个src register，对于fused uop最多有3个src register。一个uop最多可以有10个操作数，src register在操作数中的位置依据不同的uop设计而不同

这里，对于不同的uop type，需要单独讨论source number的分配问题

- Non-fusing uop

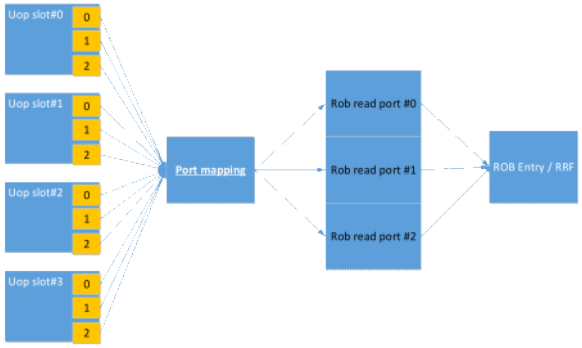
Portout	最多2个reg，dx, eax	分别返回1、0
Portin	最多1个reg, dx	返回0
Load/sta/lea/lea1	(base，index)	base返回0，index返回2

Std / fe_fp	最多1个reg	返回0
Std / opcode_std	最多1个reg	返回0
Std / std_xxx(neg, add)	最多2个reg	分别返回1， 0
Br / jcc, ujcc, ujmp_onedec, ujmp_ms_count_dec	最多1个reg， eflags	返回0
Br / jmp_indirect, call_indirect, return_indirect, jmpnear_indirect	最多1个reg，*(reg)	返回1
Br / ujmp_indirect	最多1个reg，*(reg)	返回1
Br / ujcc_indirect	最多2个reg， eflags, *(reg)	分别返回1、0
Seg_concat	最多4个reg	对于src_index<2，返回0； 否则src_index-1
fusing的首条指令，micro-fusing，分配到不同的execution port	最多2个reg	分别返回2， 0
其他普通指令	最多2个reg	分别返回1， 0

- Fusing uop

Br 对应macro-fusing，op-jcc的情况	最多2个reg，对应op上面的reg操作数	分别返回1, 0
	对于jcc和load fusing的情况，最多只有1个reg	返回1
std， sta/std micro-fusing	最多1个reg	返回1
其他的fusing情况	最多1个reg	返回1

ROB read port的分配方式



Port mapping的方式

scheme (current use 2)	uop slot	source_num	map port
0 (GNRJ-like, conservative mode)	#0	0	0
		1	1
		2	2
	#1	0	1
		1	0
		2	2
	#2	0	1
		1	0
		2	2
	#3	0	0
		1	1
		2	2
1 (3 128-bit ROB read port)	#0	0	0
		1	1
		2	2
	#1	0	2
		1	0
		2	1
	#2	0	1
		1	2
		2	0
	#3	0	0
		1	1
		2	2
2 (3 128-bit ROB read port)	#0	0	0
		1	1
		2	2
	#1	0	2
		1	1
		2	0
	#2	0	1
		1	2
		2	0
	#3	0	0
		1	2
		2	1

在RAT执行的uop

- dead_at_rat uop
  - esp folding uop(插入的esp sync指令)
- Fxchg, setting_bypass_fxchg
- Zero_mark, setting_bypass_zero_marks
- Register mov指令(GP register, XMM/MMX/SSE, FP stack register(stX, stpush/stpop), setting_bypass_moves(0, disable; 1, uop; 2, instr)

Speculative Recovery

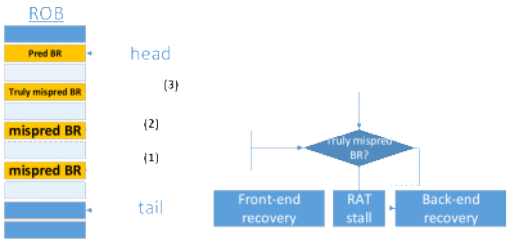
在目前model的实现中，mispred的branch指令在execution完成之后，会执行beuflush动作，表明需要做mispred-flush。beuflush的时候需要完成如下方面的若干动作

- 标记当前rob中branch之后的指令为bogus
- 清空uaq，scheduler

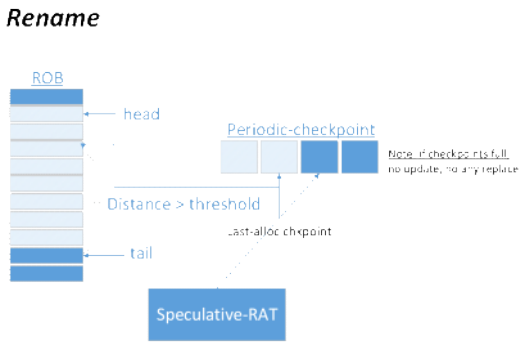
- 清空RAT中的queue，消除对应的lock table的设置
- 清空frontend
- RAT table需要重置到某个stable状态

对于RAT table重置到某个stable状态，model中实现如下的方法

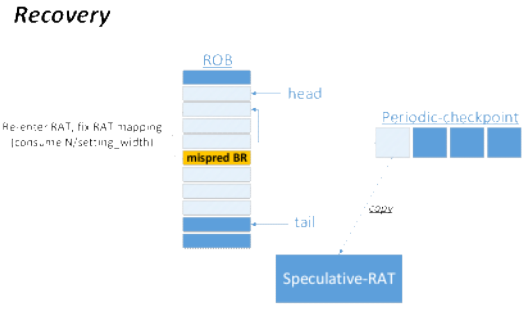
- Sequential-update
  - Frontend/RAT stall until mispred branch retire，due to different RAT mechanism
    - Rename to ROB, no need further action, RAT all point to -1, model follow this
    - Merged register file, copy RRAT(retirement-RAT) to RAT
  - Another method (doubt whether HW can implement?)
    Hold allocation stage when the first truly mispred uop do flush



- Checkpoint
  - Br-checkpoint
    针对每个branch进行RAT备份，当发现mispred的branch之后，不需要等待branch retire，直接从最近的checkpoint恢复RAT，可以马上开始新的uop rename
    - initialize阶段
      checkpoint的个数是core的资源，在core内的physical thread之间均匀分配
    - rename阶段
      对于是branch的uop尝试分配checkpoint
      - 如果checkpoint有空余空间，直接分配
      - 如果没有空余空间，则进行替换
        - first，对于branch的confidence强，且最老的checkpoint进行替换
        - second，如果checkpoint中都是confidence弱的branch
          - 当前branch是强confidence的uop，则不设置checkpoint
          - 否则，找最老的进行替换
    - Recovery阶段 (当前uop不是bogus uop)
      - 如果当前uop没有mispred，则直接回收其对应的checkpoint
      - 如果当前uop mispred，则clean该uop对应ROB之后的所有checkpoint
        - 如果当前uop有对应的checkpoint，则直接更新RAT
        - 没有对应的checkpoint，则退化为Sequential-update方式处理
    - retire或是bigflush阶段
      - Retire不需要特殊处理
      - 对于bigflush的情况，clean引起bigflush的uop对应ROB之后的所有checkpoint(更新的checkpoint)
  - Periodic-checkpoint
    周期性进行RAT的备份，当发现mispred的branch之后，不需要等待branch retire，直接从最近的checkpoint恢复RAT，并step-over最近的checkpoint到branch之间的uop rename的状态。具体做法如下：
    - initialize阶段
      checkpoint的个数是core的资源，在core内的physical thread之间均匀分配；threshold(用于判断是否应该checkpoint)等于core的ROB/checkpoints，core的ROB资源也在physical thread间均分
    - rename阶段
      只在当前T rename的第一个uop才会进行checkpoint的检查



    - recovery阶段
      不需要等待mispred BR retire，但是需要等待checkpoint到BR之间的uop恢复RAT的时间，这段时间RAT stage只能用来做fix，不能rename新的uop



- retire或是bigflush阶段
  - 对于分配了checkpoint的uop，如果是retire，则将对应checkpoint clean即可，对应checkpoint的rob指针指向invalid
  - 对于bigflush的情况，clean引起bigflush的uop对应ROB之后的所有checkpoint(更新的checkpoint)