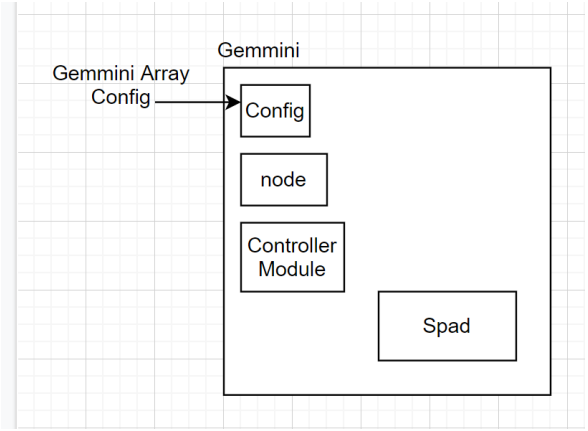
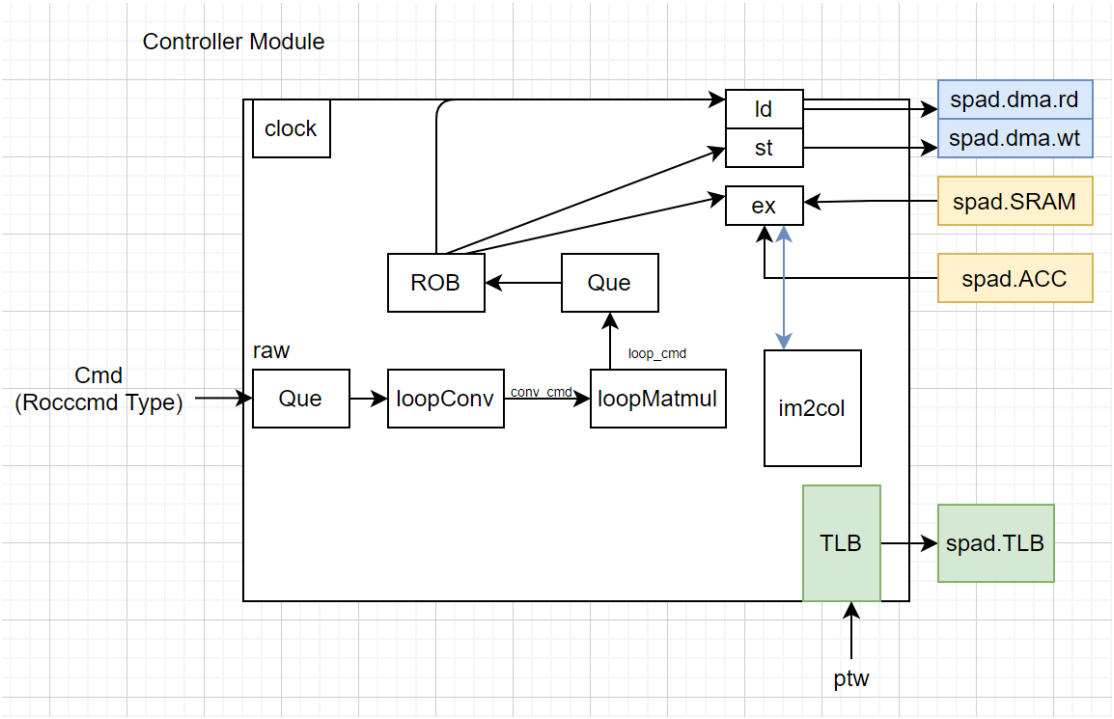
部分总结

上个星期主要阅读了 gemmini 源码的 controller, load controller, Mesh 和 Spad 部分。大致理解了整个 gemmini 的构成，大致如下：



1. **Controller Module 部分：**



该部分主要负责 Rocc 指令的展开，Controller 接收 cmd 后存入缓存队列，之后 cmd 通过 LoopConv 和 LoopMatmul 函数展开(暂时不知道展开的具体过程和结果，猜想是将一个矩阵乘法指令拆解为一系列 load，store 和 execute 的 rocc 指令)，展开后的结果依旧是 rocc 指令，进入重排序缓存端元（ROB）Rocc 的定义信息可以在"rocket-chip/src/main/scala/tile/LazyRoCC.scala"中找到，成分如下：

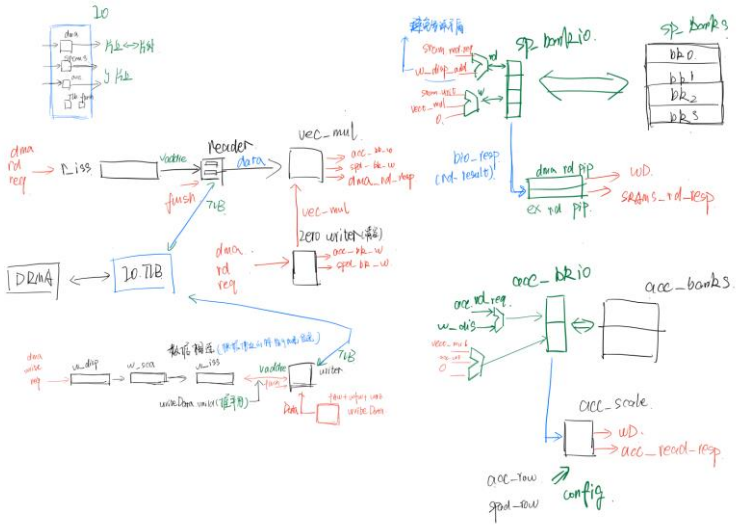
```
class RoCCInstruction extends Bundle {
  val funct = Bits(7.W)
  val rs2 = Bits(5.W)
  val rs1 = Bits(5.W)
  val xd = Bool()
  val xs1 = Bool()
  val xs2 = Bool()
  val rd = Bits(5.W)
  val opcode = Bits(7.W)
}

class RoCCCommand(implicit p: Parameters) extends CoreBundle()(p) {
  val inst = new RoCCInstruction
  val rs1 = Bits(xLen.W)
  val rs2 = Bits(xLen.W)
  val status = new MStatus
}
```

$xLen$为常量 64

ROB 会将指令发送至 load_controller 单元，store_controller 单元以及 execute_controller 单元。Load_controller 和 store_controller 内部完成解读 rocc 指令的过程。其中 load controller 和 store controller 分别和 Spad 的 dma 控制单元交互。Spad 的 dma 单元则通过内部 writer 与 reader 单元处理 dma 读写请求，wrtier 与 reader 单元通过 TLB 页表缓存来将输入的虚地址（vaddr）与 DRAM 物理地址对应，从而完成片上与片外的交互。Execute_controller 单元的流程还未分析，已知包含 mesh 阵列，完成脉动阵列计算，其接收来自 Spad 的数据。

2. Spad 部分：



Spad 的 IO 由 3 个控制单元：dma，srams 和 acc，TLB 页表缓存和 flush 信号构成。Dma，srams 和 acc 单元都有类似的 req 和 resp 两部分构造，req 接收来自外部的请求，在 spad 内部完成后由 resp 输出信号。

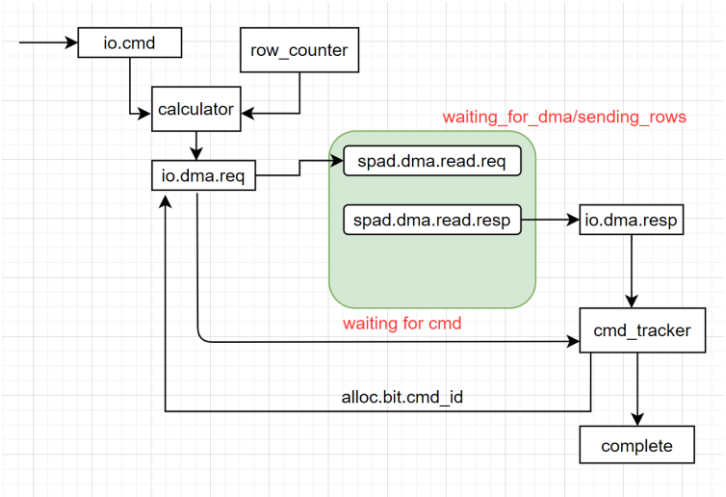
Spad 内部包含 Spad_banks, acc_banks 两种 bank，与片外存储交互的 reader 和 writer 单元，vect_mul 单元和 zero_writer 单元。

Spad_banks 处理来自 srams unit 的读写请求，acc_banks 处理来自 acc unit 的请求。其输出会通过延迟队列分别连接到 srams 的 resp 和 acc 的 resp。另外，banks 的输出都会通过 WriteData 单元输出给 writer 单元。

Dma unit 的读写指令会先通过缓冲队列，缓冲队列会判断输入指令是否为 garbage 指令来判断是否需要 0 延迟传递 garbage 指令。之后如上文所述，dma 读写指令会进入对应的 writer 和 reader 单元完成数据的读写。其中，reader 会后接 vec_mul 单元（应该是在一定条件下会进行向量乘法运算），vec_mul 单元

会将结果输给 sp_bank 和 acc_bank 单元的写输入。dma 读请求也会发送给 zero_writer 单元，zero_writer 单元接 acc_banks 和 spad_banks 的写单元进行对应地址的清零操作，其输出会发送给 vec_mul 单元。

3. Controller 中的 load controller 部分：



内部通过有限状态机控制单元的运作流程，controller 的 IO 接口主要为指令输入和 dma IO 单元。Dma IO 单元在 controller 的代码文件中完成和 spad.dma.read IO 单元的连接。

Controller 内部主要为 cmd_tracker 单元负责缓存 cmd 指令。有限状态机有三个状态等待指令（waiting_for_cmd）、等待 dma 请求处理（waiting_for_dma_req）和发送行（sending_rows）。

在 waiting_for_cmd 阶段，load controller 会根据指令的 funct 参数判断是配置自己还是执行加载任务，在通过 spad.dma unit 执行加载任务之前，会将指令包含的总数据量（单位为字节）写入 cmd_tracker，进行登记。若内部有空间存储，则 cmd_tracker 会在空闲的队列中存储该指令的总数据量并返回位置信息，cmd_tracker 的输出作为 spad.dma.read.req 的一个特征之一存储(用于标记)。若 cmd_tracker 无内部空闲，则状态机保持直至 cmd_tracker 有空闲。完成登记后机器根据上一个 dma.read.req 是否处理完毕进入 waiting_for_dma_req 和 sending_row 阶段

在 waiting_for_dma_req 阶段，load Controller 会等待 SRAM 处理上一个 dma.read.req，完成后进入 sending_row 阶段。

在 sending_row 阶段，controller 会根据是否最后一行来判断是否进入 waiting_for_cmd 阶段。Controller 内部有 row_counter 变量用来帮助 controller 判断是否为最后一行。row_counter 通过 wrap add 的机制将存入数据的位置控制在相对于 vaddr 下一定范围的栈内（但初始位置不确定）。每完成指令内部一行数据的 load 任务，dma.resp 会从 cmd_tracker 中减去对应数量的字节数。

4.