GEM5 源码阅读：O3 处理器的访存流程与错误处理

maokelong – maokelong@qq.com

Table of Contents

如果你碰巧读到了这份文档，请不要过多地在意其中的细节，因为最开始我只是习惯性地写点什么，所以很多地方都没推敲，甚至没打算让别人明白。而且我的工作并非是如同文档般线性的，而是可能会随着我理解的加深与需求的改变而不断地跳跃，这进一步增加了本文档的混乱程度。

本文档对应 GEM5 版本为 `c428c220fd351626e2ee0005dda696940261793b`。

1. 蓝图

2018年/10/17 从 GEM5 官网上找到了 O3 处理器的文档。这个文档内容严重残缺，这是其中的部分内容：

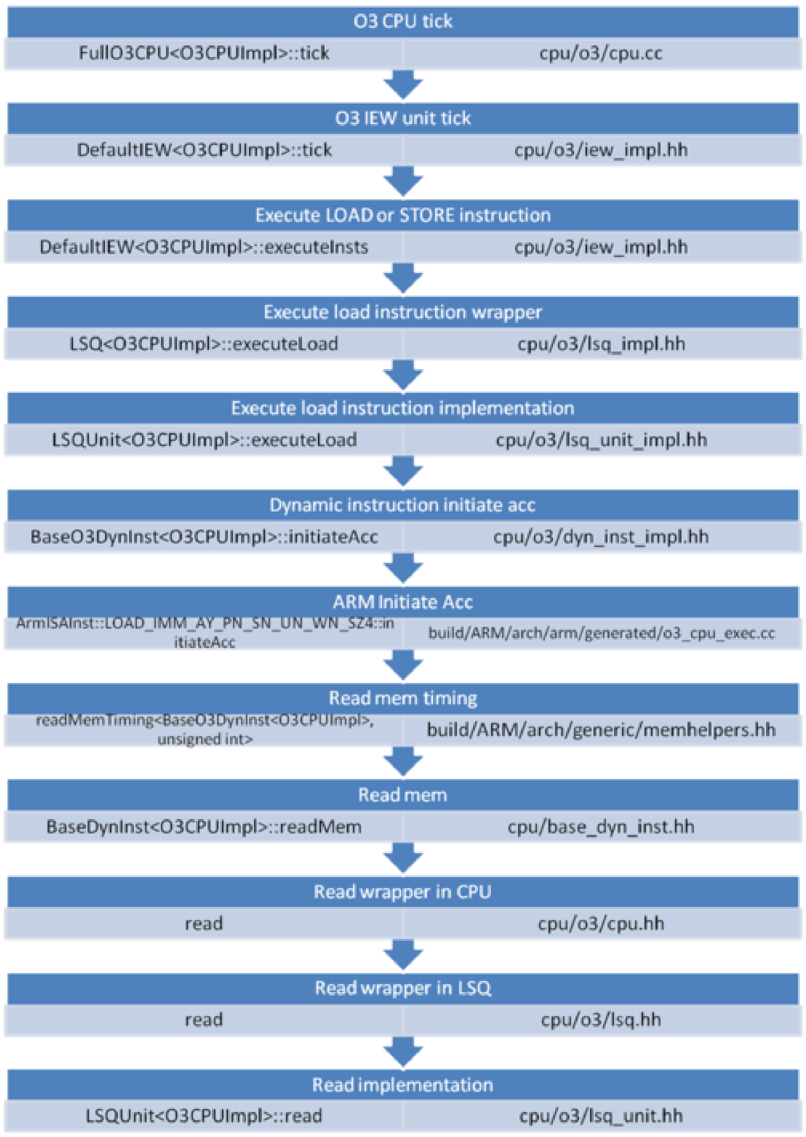
1.1. O3 流水阶段

- Fetch
  - 在每个时钟周期取指令，为哪个线程取取决于调度策略；
  - 该阶段是 `DynInst` 首先创建的阶段，`DynInst` 对象在原指令的基础上增加了运行时信息。也将完成分支预测。
- Decode
  - 在每个时钟周期译码指令；
  - 也将处理 PC-relative 的非条件分支跳转。
- Rename
  - 使用空闲链表中的物理寄存器为指令重命名，将在没有足够的寄存器时、或后台（计算）资源占尽时停顿；
  - 也将在此停顿处理一切 serializing instruction。
- Issue/Execute/Writeback
  - 执行与回写都将在 `execute()` 函数中完成， 所以将这三个阶段融合成一个阶段 → **IEW**；
  - 也会将指令派发到指令队列中；
  - 也会告诉指令队列发射指令，并执行与回写指令。
- Commit
  - 每周期提交指令，处理指令可能引起的一切异常；
  - 也将分支预测失败时重定向前台。

若觉得无法理解，那么肯定是对乱序处理器的理解不够到位。对此，可以参阅「1. 知识储备\Lesson 13 - Modern Intel Processors.pdf」。

1.2. O3 访存流程

原文档只给出了一个流程图：


O3 CPU tick
FullO3CPU<O3CPUImpl>::tick
cpu/o3/cpu.cc
O3 IEW unit tick
DefaultIEW<O3CPUImpl>::tick
cpu/o3/iew_impl.hh
Execute LOAD or STORE instruction
DefaultIEW<O3CPUImpl>::executeInsts
cpu/o3/iew_impl.hh
Execute load instruction wrapper
LSQ<O3CPUImpl>::executeLoad
cpu/o3/lsq_impl.hh
Execute load instruction implementation
LSQUnit<O3CPUImpl>::executeLoad
cpu/o3/lsq_unit_impl.hh
Dynamic instruction initiate acc
BaseO3DynInst<O3CPUImpl>::initiateAcc
cpu/o3/dyn_inst_impl.hh
ARM Initiate Acc
ArmISAInst::LOAD_IMM_AY_PN_SN_UN_WN_SZ4::initiateAcc
build/ARM/arch/arm/generated/o3_cpu_exec.cc
Read mem timing
readMemTiming<BaseO3DynInst<O3CPUImpl>, unsigned int>
build/ARM/arch/generic/memhelpers.hh
Read mem
BaseDynInst<O3CPUImpl>::readMem
cpu/base_dyn_inst.hh
Read wrapper in CPU
read
cpu/o3/cpu.hh
Read wrapper in LSQ
read
cpu/o3/lsq.hh
Read implementation
LSQUnit<O3CPUImpl>::read
cpu/o3/lsq_unit.hh
Figure 1. Load/store handling

感觉我还是亲自遍历一下上述路径较好。实践出真知，这个过程可以帮助我真正地熟悉 GEM5 的代码结构：我所困惑的问题肯定无法简单地通过看这个图就能解决，加上 GEM5 的文档又不全，我以后肯定得循着我要解决的问题充分阅读源码、充分发散，现在做最基本的准备肯定是没毛病的。

---

2. 蓝图初步还原

2.1. 源码追踪

这里从源代码中一一找出各方法所在的位置。这里不做进一步发散，只是为了勾勒出最基本的框架。

1. src\cpu\o3\cpu.cc :564 `FullO3CPU<Impl>::tick()`，处理器流水各阶段时钟周期迭代；各 buffers 推进。
2. src\cpu\o3\iew_impl.hh :1453 `DefaultIEW<Impl>::tick()`，处理器 IEW 阶段时钟周期迭代；
3. src\cpu\o3\iew_impl.hh :1176 `void DefaultIEW<Impl>::executeInsts()`，执行 instQueue 中一切可执行指令，其中在执行访存指令时：
    a. 若是 Load 指令，则在 ldstQueue 中执行该 Load 指令（`fault = ldstQueue.executeLoad(inst);`），若因遍历页表而需要延迟，则在 instQueue 中延迟该指令。特别地，预取数据不会产生 Fault；
    b. 若是 Store 指令，则在 ldstQueue 中执行该 Store 指令（`fault = ldstQueue.executeStore(inst);`），若因遍历页表而需要延迟，则在 instQueue 中延迟该指令。若期

间出错则标记为提交；

4. src\cpu\o3\lsq_unit_impl.hh :608 `Fault LSQUnit<Impl>::executeLoad(DynInstPtr &inst)`，开始执行 LSQ 中的指令，其中核心调用为 `load_fault = inst→initiateAcc();`

   Store → :657 `Fault LSQUnit<Impl>::executeStore(DynInstPtr &store_inst)`，其中核心调用为 `store_inst→initiateAcc();`

5. src\cpu\o3\dyn_inst_impl.hh :147 `Fault BaseO3DynInst<Impl>::initiateAcc()`，动态指令（包括指令的实时信息）→ 静态指令（ISA），其主要调用了 `this→staticInst→initiateAcc(this, this→traceData)`

6. src\arch\x86\isa\microops\ldstop.isa :120 `Fault %(class_name)s::initiateAcc(ExecContext * xc, Trace::InstRecord * traceData`，ISA 模板，其主要调用了 `fault = initiateMemRead(x5);`

   Store → :189 其核心调用为 `fault = writeMemTiming(7);`

7. src\arch\x86\memhelpers.hh :47 定义了 ini\tiateMemRead(x5)，表示在 timing 模式下发起读存操作，仅一行代码，执行 `return xc→initiateMemRead(addr, dataSize, flags)`

   Store → :176 其核心调用为 `xc→writeMem(x5);` 。这旁边还有一个简单封装了批写存的同名方法，不要为此（同名）而感到困惑。

8. src\cpu\base_dyn_inst.hh :893 `Fault BaseDynInst<Impl>::initiateMemRead(Addr addr, unsigned size, Request::Flags flags)`，执行地址转换，并发起访存。

   Store → :944 `Fault BaseDynInst<Impl>::writeMem(uint8_t *data, unsigned size, Addr addr, Request::Flags flags, uint64_t *res)`，同上。

9. src\cpu\o3\cpu.hh :747 `Fault read(x4)`，一个只有一行的 Wrapper

   Store → :755 同上。

10. src\cpu\o3\lsq.hh :336 `LSQ<Impl>::read(x4)`，一个只有两行的 Wrapper

    Store → :347 同上。

11. src\cpu\o3\lsq_unit.hh :554 `LSQUnit<Impl>::read(x4)` 至此，内存请求就可以发送到 Cache 端口了。（如果未在 StoreQueue 中命中）

    Store → :858 将 Store 请求丢到 storeQueue 中。

> 经过抽丝剥茧，发现 1-4 主要解决了 OOO 处理器如何进行指令调度，5-7 查找 Handler 用于在 timing mode 下执行 X86 指令，8-11 是真正执行 L/S 的地方。

2.2. 日志追踪

以下节选自 SE 模式下打开部分 debug flags 后的打印内容。我有点忘记了我当时为什么要打印这些东西。

```
104013216: system.cpu.iew.lsq.thread0: Inserting load PC (0x7ffff7b352a0=>0x7ffff7b352a4).(0=>1), idx:14
[sn:447312]
104013216: system.cpu.iew.lsq.thread0: Executing load PC (0x7ffff7b352c7=>0x7ffff7b352cc).(0=>1),
[sn:447285]
104013216: system.cpu.iew.lsq.thread0: Read called, load idx: 10, store idx: -1, storeHead: 30 addr:
0x517ae
104013216: system.cpu.iew.lsq.thread0: Doing memory access for inst [sn:447285] PC
(0x7ffff7b352c7=>0x7ffff7b352cc).(0=>1)
104013216: system.cpu.iew.lsq.thread0: Executing load PC (0x7ffff7b352d9=>0x7ffff7b352de).(0=>1),
[sn:447291]
104013216: system.cpu.iew.lsq.thread0: Read called, load idx: 11, store idx: -1, storeHead: 30 addr:
0x517ac
104013216: system.cpu.iew.lsq.thread0: Doing memory access for inst [sn:447291] PC
```

```
(0x7ffff7b352d9=>0x7ffff7b352de).(0=>1)
104013216: system.cpu.iew.lsq.thread0: Executing load PC (0x7ffff7b352e8=>0x7ffff7b352ec).(0=>1),
[sn:447300]
104013216: system.cpu.iew.lsq.thread0: Read called, load idx: 12, store idx: -1, storeHead: 30 addr:
0x517b0
104013216: system.cpu.iew.lsq.thread0: Doing memory access for inst [sn:447300] PC
(0x7ffff7b352e8=>0x7ffff7b352ec).(0=>1)
104013216: system.cpu.iew.lsq.thread0: LQ size: 33, #loads occupied: 6
104013216: system.cpu.iew.lsq.thread0: SQ size: 33, #stores occupied: 0
104013549: system.cpu.iew.lsq.thread0: Inserting load PC (0x7ffff7b352b7=>0x7ffff7b352bb).(0=>1), idx:15
[sn:447326]
104013549: system.cpu.iew.lsq.thread0: Executing load PC (0x7ffff7b352a0=>0x7ffff7b352a4).(0=>1),
[sn:447312]
104013549: system.cpu.iew.lsq.thread0: Read called, load idx: 14, store idx: -1, storeHead: 30 addr:
0x517b8
104013549: system.cpu.iew.lsq.thread0: Doing memory access for inst [sn:447312] PC
(0x7ffff7b352a0=>0x7ffff7b352a4).(0=>1)
104013549: system.cpu.iew.lsq.thread0: LQ size: 33, #loads occupied: 7
104013549: system.cpu.iew.lsq.thread0: SQ size: 33, #stores occupied: 0
104013882: system.cpu.iew.lsq.thread0: Committing head load instruction, PC
(0x7ffff7b352b7=>0x7ffff7b352bb).(0=>1)
104013882: system.cpu.iew.lsq.thread0: LQ size: 33, #loads occupied: 6
104013882: system.cpu.iew.lsq.thread0: SQ size: 33, #stores occupied: 0
104014215: system.cpu.iew.lsq.thread0: Squashing until [sn:447281]!(Loads:6 Stores:0)
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352b7=>0x7ffff7b352bb).(0=>1)
squashed, [sn:447326]
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352a0=>0x7ffff7b352a4).(0=>1)
squashed, [sn:447312]
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352ef=>0x7ffff7b352f2).(0=>1)
squashed, [sn:447302]
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352e8=>0x7ffff7b352ec).(0=>1)
squashed, [sn:447300]
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352d9=>0x7ffff7b352de).(0=>1)
squashed, [sn:447291]
104014215: system.cpu.iew.lsq.thread0: Load Instruction PC (0x7ffff7b352c7=>0x7ffff7b352cc).(0=>1)
squashed, [sn:447285]
104014215: system.cpu.iew.lsq.thread0: LQ size: 33, #loads occupied: 0
104014215: system.cpu.iew.lsq.thread0: SQ size: 33, #stores occupied: 0
```

---

3. 蓝图精细还原

3.1. 存取队列

根据上面的追踪结果，可以发现访存的实际发起阶段就在最后一个方法里。下面就将对其进行细致的分析。

src|cpu|o3|lsq_unit.hh

```
template <class Impl>
Fault
LSQUnit<Impl>::read(const RequestPtr &req,
                    RequestPtr &sreqLow, RequestPtr &sreqHigh,
                    int load_idx)
{
    DynInstPtr load_inst = loadQueue[load_idx];

    assert(load_inst);

    assert(!load_inst->isExecuted());

    // 如果是必须保持顺序的指令。（指令之间有些存在 happen-before 关系）
    // 当指令执行时未处于 LSQ 队首时即认为其乱序，此时需要将其进行重调度。
    // Make sure this isn't a strictly ordered load
    // A bit of a hackish way to get strictly ordered accesses to work
    // only if they're at the head of the LSQ and are ready to commit
    // (at the head of the ROB too).
    if (req->isStrictlyOrdered() &&
        (load_idx != loadHead || !load_inst->isAtCommit())) {
```

```
        iewStage->rescheduleMemInst(load_inst);
        ++lsqRescheduledLoads;
        DPRINTF(LSQUnit, "Strictly ordered load [sn:%lli] PC %s\n",
                load_inst->seqNum, load_inst->pcState());

        return std::make_shared<GenericISA::M5PanicFault>(
            "Strictly ordered load [sn:%llx] PC %s\n",
            load_inst->seqNum, load_inst->pcState());
    }

    // Check the SQ for any previous stores that might lead to forwarding
    int store_idx = load_inst->sqIdx;

    int store_size = 0;

    DPRINTF(LSQUnit, "Read called, load idx: %i, store idx: %i, "
            "storeHead: %i addr: %#x%s\n",
            load_idx, store_idx, storeHead, req->getPaddr(),
            sreqLow ? " split" : "");

    // 如果是 LLSC 指令。（LLSC 用于原子访存）
    // 为了确保原子性，将不再临时记录指令结果，以免在执行过程中满足了别的指令的完成条件。在执行 LLSC 指令之后重新开启记录。
    if (req->isLLSC()) {
        assert(!sreqLow);
        // Disable recording the result temporarily.  Writing to misc
        // regs normally updates the result, but this is not the
        // desired behavior when handling store conditionals.
        load_inst->recordResult(false);
        TheISA::handleLockedRead(load_inst.get(), req);
        load_inst->recordResult(true);
    }

    // 如果是访问映射到物理内存上的处理器内部寄存器。
    // 则，略
    if (req->isMmappedIpr()) {
        assert(!load_inst->memData);
        load_inst->memData = new uint8_t[64];

        ThreadContext *thread = cpu->tcBase(lsqID);
        Cycles delay(0);
        PacketPtr data_pkt = new Packet(req, MemCmd::ReadReq);

        data_pkt->dataStatic(load_inst->memData);
        if (!TheISA::HasUnalignedMemAcc || !sreqLow) {
            delay = TheISA::handleIprRead(thread, data_pkt);
        } else {
            assert(sreqLow->isMmappedIpr() && sreqHigh->isMmappedIpr());
            PacketPtr fst_data_pkt = new Packet(sreqLow, MemCmd::ReadReq);
            PacketPtr snd_data_pkt = new Packet(sreqHigh, MemCmd::ReadReq);

            fst_data_pkt->dataStatic(load_inst->memData);
            snd_data_pkt->dataStatic(load_inst->memData + sreqLow->getSize());

            delay = TheISA::handleIprRead(thread, fst_data_pkt);
            Cycles delay2 = TheISA::handleIprRead(thread, snd_data_pkt);
            if (delay2 > delay)
                delay = delay2;

            delete fst_data_pkt;
            delete snd_data_pkt;
        }
        WritebackEvent *wb = new WritebackEvent(load_inst, data_pkt, this);
        cpu->schedule(wb, cpu->clockEdge(delay));
        return NoFault;
    }

    // 考虑当前 Load 指令可以从别的 Store 获取 Data 的情况
    // 此时 Load 指令可以直接 Forward，详情，略
    while (store_idx != -1) {
        // End once we've reached the top of the LSQ
        if (store_idx == storeWBIdx) {
            break;
        }

        // Move the index to one younger
        if (--store_idx < 0)
            store_idx += SQEntries;

        assert(storeQueue[store_idx].inst);

        store_size = storeQueue[store_idx].size;

        // 如果无需存储数据，或 SC，或 Cache 维护指令，则直接 continue
        if (!store_size || storeQueue[store_idx].inst->strictlyOrdered() ||
            (storeQueue[store_idx].req &&
             storeQueue[store_idx].req->isCacheMaintenance())) {
            // Cache maintenance instructions go down via the store
            // path but they carry no data and they shouldn't be
            // considered for forwarding
            continue;
        }

        assert(storeQueue[store_idx].inst->effAddrValid());

        // Check if the store data is within the lower and upper bounds of
        // addresses that the request needs.
        bool store_has_lower_limit =
            req->getVaddr() >= storeQueue[store_idx].inst->effAddr;
        bool store_has_upper_limit =
            (req->getVaddr() + req->getSize()) <=
            (storeQueue[store_idx].inst->effAddr + store_size);
        bool lower_load_has_store_part =
            req->getVaddr() < (storeQueue[store_idx].inst->effAddr +
                               store_size);
        bool upper_load_has_store_part =
            (req->getVaddr() + req->getSize()) >
            storeQueue[store_idx].inst->effAddr;

        // If the store's data has all of the data needed and the load isn't
        // LLSC, we can forward.
        if (store_has_lower_limit && store_has_upper_limit && !req->isLLSC()) {
            // Get shift amount for offset into the store's data.
            int shift_amt = req->getVaddr() - storeQueue[store_idx].inst->effAddr;

            // Allocate memory if this is the first time a load is issued.
            if (!load_inst->memData) {
                load_inst->memData = new uint8_t[req->getSize()];
            }
            if (storeQueue[store_idx].isAllZeros)
                memset(load_inst->memData, 0, req->getSize());
            else
                memcpy(load_inst->memData,
                    storeQueue[store_idx].data + shift_amt, req->getSize());

            DPRINTF(LSQUnit, "Forwarding from store idx %i to load to "
                    "addr %#x\n", store_idx, req->getVaddr());

            PacketPtr data_pkt = new Packet(req, MemCmd::ReadReq);
            data_pkt->dataStatic(load_inst->memData);

            WritebackEvent *wb = new WritebackEvent(load_inst, data_pkt, this);

            // We'll say this has a 1 cycle load-store forwarding latency
            // for now.
            // @todo: Need to make this a parameter.
            cpu->schedule(wb, curTick());

            ++lsqForwLoads;
            return NoFault;
        } else if (
                (!req->isLLSC() &&
                 ((store_has_lower_limit && lower_load_has_store_part) ||
                  (store_has_upper_limit && upper_load_has_store_part) ||
                  (lower_load_has_store_part && upper_load_has_store_part))) ||
                (req->isLLSC() &&
                 ((store_has_lower_limit || upper_load_has_store_part) &&
                  (store_has_upper_limit || lower_load_has_store_part)))) {
            // This is the partial store-load forwarding case where a store
            // has only part of the load's data and the load isn't LLSC or
            // the load is LLSC and the store has all or part of the load's
            // data

            // If it's already been written back, then don't worry about
            // stalling on it.
            if (storeQueue[store_idx].completed) {
                panic("Should not check one of these");
                continue;
            }

            // Must stall load and force it to retry, so long as it's the oldest
            // load that needs to do so.
            if (!stalled ||
                (stalled &&
                 load_inst->seqNum <
                 loadQueue[stallingLoadIdx]->seqNum)) {
                stalled = true;
                stallingStoreIsn = storeQueue[store_idx].inst->seqNum;
                stallingLoadIdx = load_idx;
            }

            // Tell IQ/mem dep unit that this instruction will need to be
            // rescheduled eventually
            iewStage->rescheduleMemInst(load_inst);
            load_inst->clearIssued();
            ++lsqRescheduledLoads;

            // Do not generate a writeback event as this instruction is not
            // complete.
            DPRINTF(LSQUnit, "Load-store forwarding mis-match. "
                    "Store idx %i to load addr %#x\n",
                    store_idx, req->getVaddr());

            return NoFault;
        }
    }

    // If there's no forwarding case, then go access memory
    DPRINTF(LSQUnit, "Doing memory access for inst [sn:%lli] PC %s\n",
            load_inst->seqNum, load_inst->pcState());

    // Allocate memory if this is the first time a load is issued.
    if (!load_inst->memData) {
        load_inst->memData = new uint8_t[req->getSize()];
    }

    // if we the cache is not blocked, do cache access
    bool completedFirst = false;
    PacketPtr data_pkt = Packet::createRead(req);
    PacketPtr fst_data_pkt = NULL;
    PacketPtr snd_data_pkt = NULL;

    data_pkt->dataStatic(load_inst->memData);

    LSQSenderState *state = new LSQSenderState;
    state->isLoad = true;
    state->idx = load_idx;
    state->inst = load_inst;
    data_pkt->senderState = state;

    if (!TheISA::HasUnalignedMemAcc || !sreqLow) {
        // Point the first packet at the main data packet.
        fst_data_pkt = data_pkt;
    } else {
        // Create the split packets.
        fst_data_pkt = Packet::createRead(sreqLow);
        snd_data_pkt = Packet::createRead(sreqHigh);
        fst_data_pkt->dataStatic(load_inst->memData);
        snd_data_pkt->dataStatic(load_inst->memData + sreqLow->getSize());

        fst_data_pkt->senderState = state;
        snd_data_pkt->senderState = state;

        state->isSplit = true;
        state->outstanding = 2;
        state->mainPkt = data_pkt;
    }

    // For now, load throughput is constrained by the number of
    // load FUs only, and loads do not consume a cache port (only
    // stores do).
    // @todo We should account for cache port contention
    // and arbitrate between loads and stores.
    bool successful_load = true;
    if (!dcachePort->sendTimingReq(fst_data_pkt)) {
        successful_load = false;
    } else if (TheISA::HasUnalignedMemAcc && sreqLow) {
        completedFirst = true;

        // The first packet was sent without problems, so send this one
        // too. If there is a problem with this packet then the whole
        // load will be squashed, so indicate this to the state object.
        // The first packet will return in completeDataAccess and be
        // handled there.
        // @todo We should also account for cache port contention
        // here.
        if (!dcachePort->sendTimingReq(snd_data_pkt)) {
            // The main packet will be deleted in completeDataAccess.
            state->complete();
            // Signify to 1st half that the 2nd half was blocked via state
            state->cacheBlocked = true;
            successful_load = false;
        }
    }

    // If the cache was blocked, or has become blocked due to the access,
    // handle it.
    if (!successful_load) {
        if (!sreqLow) {
            // Packet wasn't split, just delete main packet info
            delete state;
            delete data_pkt;
        }

        if (TheISA::HasUnalignedMemAcc && sreqLow) {
            if (!completedFirst) {
                // Split packet, but first failed.  Delete all state.
                delete state;
                delete data_pkt;
                delete fst_data_pkt;
                delete snd_data_pkt;
                sreqLow.reset();
                sreqHigh.reset();
            } else {
                // Can't delete main packet data or state because first packet
                // was sent to the memory system
                delete data_pkt;
                delete snd_data_pkt;
                sreqHigh.reset();
            }
        }

        ++lsqCacheBlocked;

        iewStage->blockMemInst(load_inst);

        // No fault occurred, even though the interface is blocked.
        return NoFault;
    }

    return NoFault;
}

// 说是执行 SQ 中指定索引的 store，其实只是将数据写入 store queue 中。
// TODO：Store Queue 当 Data 就绪后才会真正执行。
template <class Impl>
Fault
LSQUnit<Impl>::write(const RequestPtr &req,
                     const RequestPtr &sreqLow, const RequestPtr &sreqHigh,
                     uint8_t *data, int store_idx)
{
    assert(storeQueue[store_idx].inst);

    DPRINTF(LSQUnit, "Doing write to store idx %i, addr %#x"
            " | storeHead:%i [sn:%i]\n",
            store_idx, req->getPaddr(), storeHead,
            storeQueue[store_idx].inst->seqNum);

    storeQueue[store_idx].req = req;
    storeQueue[store_idx].sreqLow = sreqLow;
    storeQueue[store_idx].sreqHigh = sreqHigh;
    unsigned size = req->getSize();
    storeQueue[store_idx].size = size;
    bool store_no_data = req->getFlags() & Request::STORE_NO_DATA;
    storeQueue[store_idx].isAllZeros = store_no_data;
    assert(size <= sizeof(storeQueue[store_idx].data) || store_no_data);

    // Split stores can only occur in ISAs with unaligned memory accesses.  If
    // a store request has been split, sreqLow and sreqHigh will be non-null.
    if (TheISA::HasUnalignedMemAcc && sreqLow) {
        storeQueue[store_idx].isSplit = true;
    }

    if (!(req->getFlags() & Request::CACHE_BLOCK_ZERO) && \
        !req->isCacheMaintenance())
        memcpy(storeQueue[store_idx].data, data, size);

    // This function only writes the data to the store queue, so no fault
    // can happen here.
    return NoFault;
}
```

看了其中的代码之后发觉其主要作用为，处理 StrictlyOrdered、LLSC、MmappedIpr、forwarding case 等特殊情况，真正发起访存的逻辑就是将数据包发送到缓存端口，当失败时重复发送，再失败就只能阻塞访存指令。

看代码的时候意识到，在给定物理地址的情况下去访问内存系统端口是不会返回错误的，所以页错误应当是在更之前的阶段返回的。再之前也就只有地址转换了。因此接下来我应当深入地址转换的代码，其中必然有我所需要的发起页错误的逻辑。

根据 GEM5 的文档，地址转换分为这几个主要的部分：TLB management、Delayed translation、Page table walkers。

3.2. 地址转换

跟踪代码发现，地址转换的核心代码位于 TLB::translate 中。下面对其进行细致的分析。

src|arch|x86|tlb.cc

```
Fault
TLB::translate(const RequestPtr &req,
        ThreadContext *tc, Translation *translation,
        Mode mode, bool &delayedResponse, bool timing)
{
    Request::Flags flags = req->getFlags();
    int seg = flags & SegmentFlagMask;
    bool storeCheck = flags & (StoreCheck << FlagShift);

    delayedResponse = false;

    // 处理访问 Internal Memory Space 的情况。
    // 映射到映射区的 PM 不会是嵌入在处理器中的，所以我无需理解这种情况。
    // If this is true, we're dealing with a request to a non-memory address
    // space.
    if (seg == SEGMENT_REG_MS) {
        return translateInt(req, tc);
    }

    Addr vaddr = req->getVaddr();
    DPRINTF(TLB, "Translating vaddr %#x.\n", vaddr);

    HandyM5Reg m5Reg = tc->readMiscRegNoEffect(MISCREG_M5_REG);

    // 当运行在保护模式下。
    // 不在保护模式就是实模式了。
    // 实模式中 VA=PA，该模式在操作系统运行后就没有用了，我无需处理。
    // If protected mode has been enabled...
    if (m5Reg.prot) {
        DPRINTF(TLB, "In protected mode.\n");
        // 当不运行在 64 位模式，那就是 32 位喽，什么年代了还用 32 位。
        // If we're not in 64-bit mode, do protection/limit checks
        if (m5Reg.mode != LongMode) {
            DPRINTF(TLB, "Not in long mode. Checking segment protection.\n");
            // Check for a NULL segment selector.
            if (!(seg == SEGMENT_REG_TSG || seg == SYS_SEGMENT_REG_IDTR ||
                        seg == SEGMENT_REG_HS || seg == SEGMENT_REG_LS)
                    && !tc->readMiscRegNoEffect(MISCREG_SEG_SEL(seg)))
                return std::make_shared<GeneralProtection>(0);
            bool expandDown = false;
            SegAttr attr = tc->readMiscRegNoEffect(MISCREG_SEG_ATTR(seg));
            if (seg >= SEGMENT_REG_ES && seg <= SEGMENT_REG_HS) {
                if (!attr.writable && (mode == Write || storeCheck))
                    return std::make_shared<GeneralProtection>(0);
                if (!attr.readable && mode == Read)
                    return std::make_shared<GeneralProtection>(0);
                expandDown = attr.expandDown;

            }
            Addr base = tc->readMiscRegNoEffect(MISCREG_SEG_BASE(seg));
            Addr limit = tc->readMiscRegNoEffect(MISCREG_SEG_LIMIT(seg));
            bool sizeOverride = (flags & (AddrSizeFlagBit << FlagShift));
            unsigned logSize = sizeOverride ? (unsigned)m5Reg.altAddr
                                            : (unsigned)m5Reg.defAddr;
            int size = (1 << logSize) * 8;
            Addr offset = bits(vaddr - base, size - 1, 0);
            Addr endOffset = offset + req->getSize() - 1;
            if (expandDown) {
                DPRINTF(TLB, "Checking an expand down segment.\n");
                warn_once("Expand down segments are untested.\n");
                if (offset <= limit || endOffset <= limit)
                    return std::make_shared<GeneralProtection>(0);
            } else {
                if (offset > limit || endOffset > limit)
                    return std::make_shared<GeneralProtection>(0);
            }
        }
        // 同理
        if (m5Reg.submode != SixtyFourBitMode ||
                (flags & (AddrSizeFlagBit << FlagShift)))
            vaddr &= mask(32);

        // 如果开启了分页，则执行转换。
        // If paging is enabled, do the translation.
        if (m5Reg.paging) {
            DPRINTF(TLB, "Paging enabled.\n");
            // 虚拟地址已经提供了段页信息，此处只需查找 TLB 即可。
            // The vaddr already has the segment base applied.
            TlbEntry *entry = lookup(vaddr);
            if (mode == Read) {
                rdAccesses++;
            } else {
                wrAccesses++;
            }

            // 若未查找到，则在 FS 模式下调用页表 Walker 继续查找页表项，并将页表项写入 TLB。SE 模式暂不是工作重点，后续可能会继续调研。
            if (!entry) {
                DPRINTF(TLB, "Handling a TLB miss for "
                        "address %#x at pc %#x.\n",
                        vaddr, tc->instAddr());
                if (mode == Read) {
                    rdMisses++;
                } else {
                    wrMisses++;
                }
                if (FullSystem) {
                    Fault fault = walker->start(tc, translation, req, mode);
                    if (timing || fault != NoFault) {
                        // This gets ignored in atomic mode.
                        delayedResponse = true;
                        return fault;
                    }
                    entry = lookup(vaddr);
                    assert(entry);
                } else {
                    Process *p = tc->getProcessPtr();
                    const EmulationPageTable::Entry *pte =
                        p->pTable->lookup(vaddr);
                    if (!pte && mode != Execute) {
                        // Check if we just need to grow the stack.
                        if (p->fixupStackFault(vaddr)) {
                            // If we did, lookup the entry for the new page.
                            pte = p->pTable->lookup(vaddr);
                        }
                    }
                    if (!pte) {
                        return std::make_shared<PageFault>(vaddr, true, mode,
                                                           true, false);
                    } else {
                        Addr alignedVaddr = p->pTable->pageAlign(vaddr);
                        DPRINTF(TLB, "Mapping %#x to %#x\n", alignedVaddr,
                                pte->paddr);
                        entry = insert(alignedVaddr, TlbEntry(
                                p->pTable->pid(), alignedVaddr, pte->paddr,
                                pte->flags & EmulationPageTable::Uncacheable,
                                pte->flags & EmulationPageTable::ReadOnly));
                    }
                    DPRINTF(TLB, "Miss was serviced.\n");
                }
            }

            DPRINTF(TLB, "Entry found with paddr %#x, "
                    "doing protection checks.\n", entry->paddr);
            // Do paging protection checks.
            // 检查是否在用户态
            bool inUser = (m5Reg.cpl == 3 &&
                    !(flags & (CPL0FlagBit << FlagShift)));
            CR0 cr0 = tc->readMiscRegNoEffect(MISCREG_CR0);
            // 检查是否有写的权限
            bool badWrite = (!entry->writable && (inUser || cr0.wp));
            // 若用户态访问管态才能访问的地址，或向不具有写权限的地址写
            if ((inUser && !entry->user) || (mode == Write && badWrite)) {
                // The page must have been present to get into the TLB in
                // the first place. We'll assume the reserved bits are
                // fine even though we're not checking them.
                return std::make_shared<PageFault>(vaddr, true, mode, inUser,
                                                   false);
            }
            // 或者当传递的模式不是写，但检查出来是写模式，且向不具有写权限的地址写
            if (storeCheck && badWrite) {
                // This would fault if this were a write, so return a page
                // fault that reflects that happening.
                return std::make_shared<PageFault>(vaddr, true, Write, inUser,
                                                   false);
            }

            // 实际物理地址 = 物理页基址 + 页内偏移（区分大小页）
            Addr paddr = entry->paddr | (vaddr & mask(entry->logBytes));
            DPRINTF(TLB, "Translated %#x -> %#x.\n", vaddr, paddr);
            req->setPaddr(paddr);
            if (entry->uncacheable)
                req->setFlags(Request::UNCACHEABLE | Request::STRICT_ORDER);
        } else {
            //Use the address which already has segmentation applied.
            DPRINTF(TLB, "Paging disabled.\n");
            DPRINTF(TLB, "Translated %#x -> %#x.\n", vaddr, vaddr);
            req->setPaddr(vaddr);
        }
    } else {
        // Real mode
        DPRINTF(TLB, "In real mode.\n");
        DPRINTF(TLB, "Translated %#x -> %#x.\n", vaddr, vaddr);
        req->setPaddr(vaddr);
    }

    return finalizePhysical(req, tc, mode);
}
```

根据代码，异常仅有可能在执行 Walker（成功会放入 TLB） 及检查 TLB Entry 的时候发起，由于检查 TLB Entry 相关的逻辑都在本文件里，而这部分已经阅读过了，因此我仅需额外深入 Walker 方法即可。

> 现在回过头来看的确是如此。以 GEM5 原本的代码而言，PageFault 的确只会在 PageWalk 的时候发起。在通过 PageWalk 获得了 VM→PM 之后，后续的访存操作只会返回 NoFault。

src|arch|x86|pagetable_walker.cc

```
/*
 * 注意，Page Table Walker 是一个状态机，所有的状态在长模式下包括：Ready、Waiting、LongPML4、LongPDP、LongPD、LongPTE，显然，其首先分为初始化好、等待执行、开始执行这几个小步，执行之后 Walker 会分别进入四级页表。
 * 调用关系：start->startWalk->setupWalk->recvPacket->stepWalk
 */

// #69
Fault
Walker::start(ThreadContext * _tc, BaseTLB::Translation *_translation,
              const RequestPtr &_req, BaseTLB::Mode _mode)
{
    // TODO: in timing mode, instead of blocking when there are other
    // outstanding requests, see if this request can be coalesced with
    // another one (i.e. either coalesce or start walk)
    // 在这个时候初始化 Walker，读写与否（_mode），Timing 模式与否，均在此时记录到 Walker 中。
    WalkerState * newState = new WalkerState(this, _translation, _req);
    newState->initState(_tc, _mode, sys->isTimingMode());

    // 如果当前有别的 Walker 在运行，则仅将当前 Walker 放入队列尾部
    if (currStates.size()) {
        assert(newState->isTiming());
        DPRINTF(PageTableWalker, "Walks in progress: %d\n", currStates.size());
        currStates.push_back(newState);
        return NoFault;
    } else {
        // 否则将当前 Walker 放入队列尾部，然后开始执行 Walker
        currStates.push_back(newState);
        Fault fault = newState->startWalk();
        // 如果不是 Timing Mode，说明就是 Atomic Mode 喽，这个时候相关请求会立即得以响应，而不必等待回调以驱动状态机，所以可以马上删除状态机。
        if (!newState->isTiming()) {
            currStates.pop_front();
            delete newState;
        }
        return fault;
    }
}

// #221
Fault
Walker::WalkerState::startWalk()
{
    Fault fault = NoFault;
    assert(!started);
    started = true;

    // Setup Walk
    setupWalk(req->getVaddr());

    // 如果是 Timing Mode，则调整 Walker 的状态机为「Waiting」，下个状态为 LongPML4，然后发送包并等待响应。响应包的方法
在 #593。
    if (timing) {
        nextState = state;
        state = Waiting;
        timingFault = NoFault;
        sendPackets();
    } else {
        do {
            walker->port.sendAtomic(read);
            PacketPtr write = NULL;
            fault = stepWalk(write);
            assert(fault == NoFault || read == NULL);
            state = nextState;
            nextState = Ready;
            if (write)
                walker->port.sendAtomic(write);
        } while (read);
        state = Ready;
        nextState = Waiting;
    }
    return fault;
}

// #275
Fault
Walker::WalkerState::stepWalk(PacketPtr &write)
{
    assert(state != Ready && state != Waiting);
    Fault fault = NoFault;
    write = NULL;
    // 页表项，定义在 pagetable.hh #140，是一个 BitUnion
    PageTableEntry pte;
    if (dataSize == 8)
        pte = read->get<uint64_t>();
    else
        pte = read->get<uint32_t>();
    VAddr vaddr = entry.vaddr;
    bool uncacheable = pte.pcd;
    Addr nextRead = 0;
    bool doWrite = false;
    bool doTLBInsert = false;
    bool doEndWalk = false;
    bool badNX = pte.nx && mode == BaseTLB::Execute && enableNX;
    switch(state) {
      case LongPML4:
        DPRINTF(PageTableWalker,
                "Got long mode PML4 entry %#016x.\n", (uint64_t)pte);
        nextRead = ((uint64_t)pte & (mask(40) << 12)) + vaddr.longl3 * dataSize;
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = pte.w;
        entry.user = pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        entry.noExec = pte.nx;
        nextState = LongPDP;
        break;
      case LongPDP:
        DPRINTF(PageTableWalker,
                "Got long mode PDP entry %#016x.\n", (uint64_t)pte);
        nextRead = ((uint64_t)pte & (mask(40) << 12)) + vaddr.longl2 * dataSize;
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = entry.writable && pte.w;
        entry.user = entry.user && pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        nextState = LongPD;
        break;
      case LongPD:
        DPRINTF(PageTableWalker,
                "Got long mode PD entry %#016x.\n", (uint64_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = entry.writable && pte.w;
        entry.user = entry.user && pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        if (!pte.ps) {
            // 4 KB page
            entry.logBytes = 12;
            nextRead =
                ((uint64_t)pte & (mask(40) << 12)) + vaddr.longl1 * dataSize;
            nextState = LongPTE;
            break;
        } else {
            // 2 MB page
            entry.logBytes = 21;
            entry.paddr = (uint64_t)pte & (mask(31) << 21);
            entry.uncacheable = uncacheable;
            entry.global = pte.g;
            entry.patBit = bits(pte, 12);
            entry.vaddr = entry.vaddr & ~((2 * (1 << 20)) - 1);
            doTLBInsert = true;
            doEndWalk = true;
            break;
        }
      case LongPTE:
        DPRINTF(PageTableWalker,
                "Got long mode PTE entry %#016x.\n", (uint64_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = entry.writable && pte.w;
        entry.user = entry.user && pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        entry.paddr = (uint64_t)pte & (mask(40) << 12);
        entry.uncacheable = uncacheable;
        entry.global = pte.g;
        entry.patBit = bits(pte, 12);
        entry.vaddr = entry.vaddr & ~((4 * (1 << 10)) - 1);
        doTLBInsert = true;
        doEndWalk = true;
        break;
      case PAEPDP:
        DPRINTF(PageTableWalker,
                "Got legacy mode PAE PDP entry %#08x.\n", (uint32_t)pte);
        nextRead = ((uint64_t)pte & (mask(40) << 12)) + vaddr.pael2 * dataSize;
        if (!pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        nextState = PAEPD;
        break;
      case PAEPD:
        DPRINTF(PageTableWalker,
                "Got legacy mode PAE PD entry %#08x.\n", (uint32_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = pte.w;
        entry.user = pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        if (!pte.ps) {
            // 4 KB page
            entry.logBytes = 12;
            nextRead = ((uint64_t)pte & (mask(40) << 12)) + vaddr.pael1 * dataSize;
            nextState = PAEPTE;
            break;
        } else {
            // 2 MB page
            entry.logBytes = 21;
            entry.paddr = (uint64_t)pte & (mask(31) << 21);
            entry.uncacheable = uncacheable;
            entry.global = pte.g;
            entry.patBit = bits(pte, 12);
            entry.vaddr = entry.vaddr & ~((2 * (1 << 20)) - 1);
            doTLBInsert = true;
            doEndWalk = true;
            break;
        }
      case PAEPTE:
        DPRINTF(PageTableWalker,
                "Got legacy mode PAE PTE entry %#08x.\n", (uint32_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = entry.writable && pte.w;
        entry.user = entry.user && pte.u;
        if (badNX || !pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        entry.paddr = (uint64_t)pte & (mask(40) << 12);
        entry.uncacheable = uncacheable;
        entry.global = pte.g;
        entry.patBit = bits(pte, 7);
        entry.vaddr = entry.vaddr & ~((4 * (1 << 10)) - 1);
        doTLBInsert = true;
        doEndWalk = true;
        break;
      case PSEPD:
        DPRINTF(PageTableWalker,
                "Got legacy mode PSE PD entry %#08x.\n", (uint32_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = pte.w;
        entry.user = pte.u;
        if (!pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        if (!pte.ps) {
            // 4 KB page
            entry.logBytes = 12;
            nextRead =
                ((uint64_t)pte & (mask(20) << 12)) + vaddr.norml2 * dataSize;
            nextState = PTE;
            break;
        } else {
            // 4 MB page
            entry.logBytes = 21;
            entry.paddr = bits(pte, 20, 13) << 32 | bits(pte, 31, 22) << 22;
            entry.uncacheable = uncacheable;
            entry.global = pte.g;
            entry.patBit = bits(pte, 12);
            entry.vaddr = entry.vaddr & ~((4 * (1 << 20)) - 1);
            doTLBInsert = true;
            doEndWalk = true;
            break;
        }
      case PD:
        DPRINTF(PageTableWalker,
                "Got legacy mode PD entry %#08x.\n", (uint32_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = pte.w;
        entry.user = pte.u;
        if (!pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        // 4 KB page
        entry.logBytes = 12;
        nextRead = ((uint64_t)pte & (mask(20) << 12)) + vaddr.norml2 * dataSize;
        nextState = PTE;
        break;
      case PTE:
        DPRINTF(PageTableWalker,
                "Got legacy mode PTE entry %#08x.\n", (uint32_t)pte);
        doWrite = !pte.a;
        pte.a = 1;
        entry.writable = pte.w;
        entry.user = pte.u;
        if (!pte.p) {
            doEndWalk = true;
            fault = pageFault(pte.p);
            break;
        }
        entry.paddr = (uint64_t)pte & (mask(20) << 12);
        entry.uncacheable = uncacheable;
        entry.global = pte.g;
        entry.patBit = bits(pte, 7);
        entry.vaddr = entry.vaddr & ~((4 * (1 << 10)) - 1);
        doTLBInsert = true;
        doEndWalk = true;
        break;
      default:
        panic("Unknown page table walker state %d!\n");
    }
    if (doEndWalk) {
        if (doTLBInsert)
            if (!functional)
                walker->tlb->insert(entry.vaddr, entry);
        endWalk();
    } else {
        PacketPtr oldRead = read;
        //If we didn't return, we're setting up another read.
        Request::Flags flags = oldRead->req->getFlags();
        flags.set(Request::UNCACHEABLE, uncacheable);
        RequestPtr request = std::make_shared<Request>(
            nextRead, oldRead->getSize(), flags, walker->masterId);
        read = new Packet(request, MemCmd::ReadReq);
        read->allocate();
        // If we need to write, adjust the read packet to write the modified
        // value back to memory.
        if (doWrite) {
            write = oldRead;
            write->set<uint64_t>(pte);
            write->cmd = MemCmd::WriteReq;
        } else {
            write = NULL;
            delete oldRead;
        }
    }
    return fault;
}

// #554
/* Setup Walk
 * 更改状态机当前状态：四级页表第一级
 * 更改状态机 NextState: Ready
 * 准备好 read 包
 */
void
Walker::WalkerState::setupWalk(Addr vaddr)
{
    VAddr addr = vaddr;
    CR3 cr3 = tc->readMiscRegNoEffect(MISCREG_CR3);
    // Check if we're in long mode or not
    Efer efer = tc->readMiscRegNoEffect(MISCREG_EFER);
    dataSize = 8;
    Addr topAddr;
    if (efer.lma) {
        // Do long mode.
        state = LongPML4;
        topAddr = (cr3.longPdtb << 12) + addr.longl4 * dataSize;
        enableNX = efer.nxe;
    } else {
        // We're in some flavor of legacy mode.
        CR4 cr4 = tc->readMiscRegNoEffect(MISCREG_CR4);
        if (cr4.pae) {
            // Do legacy PAE.
            state = PAEPDP;
            topAddr = (cr3.paePdtb << 5) + addr.pael3 * dataSize;
            enableNX = efer.nxe;
        } else {
            dataSize = 4;
            topAddr = (cr3.pdtb << 12) + addr.norml2 * dataSize;
            if (cr4.pse) {
                // Do legacy PSE.
                state = PSEPD;
            } else {
                // Do legacy non PSE.
                state = PD;
            }
            enableNX = false;
        }
    }

    nextState = Ready;
    entry.vaddr = vaddr;

    Request::Flags flags = Request::PHYSICAL;
    if (cr3.pcd)
        flags.set(Request::UNCACHEABLE);

    RequestPtr request = std::make_shared<Request>(
        topAddr, dataSize, flags, walker->masterId);

    read = new Packet(request, MemCmd::ReadReq);
    read->allocate();
}
```

```
// #593
bool
Walker::WalkerState::recvPacket(PacketPtr pkt)
{
    assert(pkt->isResponse());
    assert(inflight);
    assert(state == Waiting);
    inflight--;
    if (pkt->isRead()) {
        // should not have a pending read it we also had one outstanding
        assert(!read);

        // @todo someone should pay for this
        pkt->headerDelay = pkt->payloadDelay = 0;

        state = nextState;
        nextState = Ready;
        PacketPtr write = NULL;
        read = pkt;
        timingFault = stepWalk(write);
        state = Waiting;
        assert(timingFault == NoFault || read == NULL);
        if (write) {
            writes.push_back(write);
        }
        sendPackets();
    } else {
        sendPackets();
    }
    if (inflight == 0 && read == NULL && writes.size() == 0) {
        state = Ready;
        nextState = Waiting;
        if (timingFault == NoFault) {
            /*
             * Finish the translation. Now that we know the right entry is
             * in the TLB, this should work with no memory accesses.
             * There could be new faults unrelated to the table walk like
             * permissions violations, so we'll need the return value as
             * well.
             */
            bool delayedResponse;
            Fault fault = walker->tlb->translate(req, tc, NULL, mode,
                                                 delayedResponse, true);
            assert(!delayedResponse);
            // Let the CPU continue.
            translation->finish(fault, req, tc, mode);
        } else {
            // There was a fault during the walk. Let the CPU know.
            translation->finish(timingFault, req, tc, mode);
        }
        return true;
    }

    return false;
}
```

4. 小关卡攻坚

4.1. 硬件是如何发起页错误的

现在我知道 IEW 阶段发起页错误的具体位置了，接下来需要进一步调查页错误是如何处理的。感觉不会需要很多工作量，这部分应该没什么特别地。

4.1.1. 发起过程

根据官方文档，所有的错误都是在 Commit 阶段处理的，所以我们首先应深入该阶段。由于错误具体是如何被 trap 的这对于我来说并不重要，因此这里只是简单地分析一下调用栈。

- src\cpu\o3\commit_impl.hh :809 `DefaultCommit<Impl>::commit()` 流水中提交阶段。具体提交位置为 :923。
- src\cpu\o3\commit_impl.hh :964 `DefaultCommit<Impl>::commitInsts()` 提交指令，如果此时 ROB 中位于首部的指令未 Squash，则试着 Commit 该指令；
- src\cpu\o3\commit_impl.hh :1139 `DefaultCommit<Impl>::commitHead()` Commit 之前检查到 Fault，陷入 trap。具体位置为 :1228 cpu→trap()
- src\cpu\o3\cpu.cc :991 `FullO3CPU<Impl>::trap()` 中最终发起了 `Fault: fault→invoke(this→threadContexts[tid], inst);`

ok，现在可以深入错误的 invoke 代码了。

首先，得明白 Fault 之间的派生关系。最终得到的 Fault 是 PageFault，其派生关系为：FaultBase → X86ISA:X86FaultBase → X86ISA:X86Fault → X86ISA:PageFault。所以虚函数 fault→invoke() 执行的是哪个就很显然了。

src|arch|x86|faults.cc

```
    // :55
    void X86FaultBase::invoke(ThreadContext * tc, const StaticInstPtr &inst)
    {
        if (!FullSystem) {
            FaultBase::invoke(tc, inst);
            return;
        }

        PCState pcState = tc->pcState();
        Addr pc = pcState.pc();
        DPRINTF(Faults, "RIP %#x: vector %d: %s\n",
                pc, vector, describe());
        using namespace X86ISAInst::RomLabels;
        HandyM5Reg m5reg = tc->readMiscRegNoEffect(MISCREG_M5_REG);
        MicroPC entry;
        if (m5reg.mode == LongMode) {
            if (isSoft()) {
                entry = extern_label_longModeSoftInterrupt;
            } else {
                entry = extern_label_longModeInterrupt;
            }
        } else {
            entry = extern_label_legacyModeInterrupt;
        }
        tc->setIntReg(INTREG_MICRO(1), vector);
        tc->setIntReg(INTREG_MICRO(7), pc);
        if (errorCode != (uint64_t)(-1)) {
            if (m5reg.mode == LongMode) {
                // 长模式且返回错误码情况下的中断入口
                // 微码定义在 src\arch\x86\isa\insts\romutil.py :194，其原格式定义在 : 29
                entry = extern_label_longModeInterruptWithError;
            } else {
                panic("Legacy mode interrupts with error codes "
                        "aren't implementde.\n");
            }
            // Software interrupts shouldn't have error codes. If one
            // does, there would need to be microcode to set it up.
            assert(!isSoft());
            tc->setIntReg(INTREG_MICRO(15), errorCode);
        }
        pcState.upc(romMicroPC(entry));
        pcState.nupc(romMicroPC(entry) + 1);
        tc->pcState(pcState);
    }

    // :137
    void PageFault::invoke(ThreadContext * tc, const StaticInstPtr &inst)
    {
        if (FullSystem) {
            /* Invalidate any matching TLB entries before handling the page fault */
            tc->getITBPtr()->demapPage(addr, 0);
            tc->getDTBPtr()->demapPage(addr, 0);
            HandyM5Reg m5reg = tc->readMiscRegNoEffect(MISCREG_M5_REG);
            X86FaultBase::invoke(tc);
            /*
             * If something bad happens while trying to enter the page fault
             * handler, I'm pretty sure that's a double fault and then all
             * bets are off. That means it should be safe to update this
             * state now.
             */
            if (m5reg.mode == LongMode) {
                tc->setMiscReg(MISCREG_CR2, addr);
            } else {
                tc->setMiscReg(MISCREG_CR2, (uint32_t)addr);
            }
        } else {
            PageFaultErrorCode code = errorCode;
            const char *modeStr = "";
            if (code.fetch)
                modeStr = "execute";
            else if (code.write)
                modeStr = "write";
            else
                modeStr = "read";

            // print information about what we are panic'ing on
            if (!inst) {
                panic("Tried to %s unmapped address %#x.\n", modeStr, addr);
            } else {
                panic("Tried to %s unmapped address %#x.\nPC: %#x, Instr: %s",
                      modeStr, addr, tc->pcState().pc(),
                      inst->disassemble(tc->pcState().pc(), debugSymbolTable));
            }
        }
    }
```

果然需要深入汇编代码，懒得看了...

4.1.2. 错误码的生成过程

ErrorCode 系 X86FaultBase 类的成员，在构造的时候使用简单组装参数而成，所以想要知道 ErrorCode 是怎么生成的，还需要回头去看 PageFault 是在哪构造的。

src|arch|x86|faults.hh :322

```
PageFault(Addr _addr, bool present, BaseTLB::Mode mode,
              bool user, bool reserved) :
            X86Fault("Page-Fault", "#PF", 14, 0), addr(_addr)
      {
            PageFaultErrorCode code = 0;
            code.present = present;
            code.write = (mode == BaseTLB::Write);
            code.user = user;
            code.reserved = reserved;
            code.fetch = (mode == BaseTLB::Execute);
            errorCode = code;
      }
```

下面查看 Walker 源码寻找答案。

src|arch|x86|faults.hh :706

```
Fault
Walker::WalkerState::pageFault(bool present)
```

```
{
    DPRINTF(PageTableWalker, "Raising page fault.\n");
    HandyM5Reg m5reg = tc->readMiscRegNoEffect(MISCREG_M5_REG);
    if (mode == BaseTLB::Execute && !enableNX)
        mode = BaseTLB::Read;
    return std::make_shared<PageFault>(entry.vaddr, present, mode,
                                       m5reg.cpl == 3, false);
}
```

那么 pageFault() 发起的位置呢？在 Walk 的时候每次判定 `badNX || !pte.p` 都会结束 Walk 并发起 PageFault。根据网传资料及源码分析，badNX 项不用理会，因这在预取代码段时才会用到，而我要管理的内存一定在 Memory-mapping Segment。

src\arch\x86\pagetable.hh :140

```
BitUnion64(PageTableEntry)
    // Disable  or  enable  instrucFon  fetches  from  the  child  page, limiting  execution to text
segment only
    Bitfield<63> nx;
    // level 1 - level 3: 40 most significant bits of physical page table address (forces page tables to
be 4KB aligned)
    // level 4: 40 most significant bits of physical page address (forces pages to be 4KB aligned)
    Bitfield<51, 12> base;
    Bitfield<11, 9> avl;
    // Global page (don’t evict from TLB on task switch)
    Bitfield<8> g;
    // level 1 - level 3: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).
    Bitfield<7> ps;
    // level 4: Dirty bit (set by MMU on writes, cleared by software)
    Bitfield<6> d;
    // Reference bit (set by MMU on reads and writes, cleared by software)
    Bitfield<5> a;
    // Caching disabled or enabled for the child page table
    Bitfield<4> pcd;
    // Write-through or write-back cache policy for the child page table
    Bitfield<3> pwt;
    // user or supervisor (kernel) mode access permission for all reachable pages
    Bitfield<2> u;
    // Read-only or read-write access access permission for all reachable pages
    Bitfield<1> w;
    // Child page table present in physical memory (1) or not (0).
    Bitfield<0> p;
EndBitUnion(PageTableEntry)
```

4.2. Flush 指令是否是 Store 指令的一种

仔细一想有点问题，我能在 L/S 路径上返回一个 PageFault，那么 Flush 指令怎么办？它是否实现为了 Store 指令的一种？

4.2.1. 如何判断指令是 Load 还是 Store

1. src\cpu\o3\iew_impl.hh :1176 `DefaultIEW<Impl>::executeInsts()`，执行 instQueue 中一切可执行指令；
2. src\cpu\base_dyn_inst.hh :505-507 `isMemRef/Load/Store()`，直接调用静态指令相应方法；
3. src\cpu\static_inst.hh :143-145 `isMemRef/Load/Store()`，直接返回 `flags[IsMemRef/Load/Store]`；

那么这些 flags 是如何设置的呢？这就需要去查看源码了。

结果查找结果很令我困惑，在以 IsMemRef 为关键词进行查找的时候，我只得到了如下两个结果：

- src\arch\x86\isa\formats\monitor_mwait.isa :84 `MwaitInst` 类构造时即设置 `flags[IsMemRef] = 1; flags[IsLoad] = 1;`

- src\arch\x86\isa\operands.isa :209 `def operands {{ 'Mem': ('Mem', 'uqw', None, ('IsMemRef', 'IsLoad', 'IsStore'), 300) }};`

> Instruction = mnemonic + Operands

对于后者，需要了解 GEM5 的 Code parsing 才行，其中从前到后四个元素的含义分别为：1) operand class; 2) default type; 3) specifier; 4) instruction flags; 5) priority。

> *If the flag operand is a triple, the **first** element is unconditional, the **second** is inferred when the operand is a source, and the **third** when it is a destination.*
>
> — *GEM5 Code parsing*
> *地址见附录*

根据上面的引用，我们可以知道：判断指令是否是 Load/Store 指令的关键在于，确定其操作数是 Source 还是 Target。

- src\arch\isa_parser.py :396 `Operand` 类在构造时即通过构造参数传递其是否 src 或 dest
- src\arch\isa_parser.py :978 `MemOperand` 类派生自 `Operand` 类
- src\arch\isa_parser.py :1032 `OperandList` 类包含名为 `memOperand` 的成员，姑且认为是用来保存 MemOperand 类型的成员的，其在构造时表明当 code 中匹配到 `[xxx] =` 或 `=` 时即认为是 dest，否则为 src
- src\arch\isa_parser.py :1315 `InstObjParams` 类

```
class InstObjParams(object):
    def __init__(self, parser, mnem, class_name, base_class = '',
                 snippets = {}, opt_args = []):
        self.mnemonic = mnem
        self.class_name = class_name
        self.base_class = base_class
        if not isinstance(snippets, dict):
            snippets = {'code' : snippets}
        // 将 snippets 转换为使用 ' ' 隔开各项的字符串
        compositeCode = ' '.join(map(str, snippets.values()))
        self.snippets = snippets

        self.operands = OperandList(parser, compositeCode)
```

- src\arch\x86\isa\microops\ldstop.isa :575 我查找了实例化这个类的地方，这里以 Store 为例

```
    def defineMicroStoreOp(mnemonic, code, completeCode="", mem_flags="0",
                           implicitStack=False):
        global header_output
        global decoder_output
        global exec_output
        global microopClasses
        Name = mnemonic
        name = mnemonic.lower()

        # Build up the all register version of this micro op
        iop = InstObjParams(name, Name, 'X86ISA::LdStOp',
                            { "code": code,
                              "complete_code": completeCode,
                              "ea_code": calculateEA,
                              "memDataSize": "dataSize" })
```

接下来我继续查找实例化 defineMicroStoreOp 的地方，结果直接查找到了定义 `Clflushopt` 微码的位置，所以现在我将直接深入 FLUSH 指令分析。

4.2.2. FLUSH 指令深入分析

- src\arch\x86\isa\insts\general_purpose\cache_and_memory_management.py :61 可以发现，CLFLUSH 这一 MACRO 指令实现为 clflushopt + mfence 的复合 MICRO 指令；
- src\arch\x86\isa\microops\ldstop.isa :642 定义了微指令 clflushopt

```
defineMicroStoreOp('Clflushopt', 'Mem = 0;',
                   mem_flags="Request::CLEAN | Request::INVALIDATE" +
                   " | Request::DST_POC")
```

- src\arch\x86\isa\microops\ldstop.isa :575 至此，与上一小节成功对接

ok，现在总结上述所有调用

1. Clflushopt 定义时简单调用 defineMicroStoreOp

```
defineMicroStoreOp('Clflushopt', 'Mem = 0;',
                   mem_flags="Request::CLEAN | Request::INVALIDATE" +
                   " | Request::DST_POC")
```

2. defineMicroStoreOp 中 InstObjParams 定义

```
iop = InstObjParams(clflushopt, Clflushopt, 'X86ISA::LdStOp',
                    { "code": 'Mem = 0;',
                        "complete_code": "",
                        "ea_code": calculateEA,         // 略
                        "memDataSize": "dataSize" })    // 略
```

3. InstObjParams 之中 OperandList 初始化

```
self.operands = OperandList(parser, "Mem = 0; 略略")
```

4. OperandList 中 is_dest 的设置

```
// :1061 匹配 "[xx] =" 或 "=", Store 下成功
is_dest = (assignRE.match(code, match.end()) != None)

// :1044 匹配所有 Operand，根据 src\arch\x86\isa\operands.isa，Store 给的参数 "Mem" 是 Operand，而 Load 给的参数 "Data" 不是 Operand
match = parser.operandsRE.search(code, next_pos)
```

5. 判定 Clflushopt 的 Operand 是 target，所以该静态指令的 flag 将设置为 store。所以：

> **Clflushopt 是 Store 指令的一种！！！！**

4.3. 动态指令（仅 Store）生命周期

在之前的源码阅读中，我发现 Fault 是作为动态指令的上下文的，那么问题来了，动态指令的生命周期是什么样的？什么时候产生（不在乎），什么时候登记中断，什么时候销毁。了解了这些我才能知道怎么优雅地在 Walk 之外的地方优雅地插入 Fault。

> 接下来的分析就以 Store 为主了。

4.3.1. 错误是在哪写进动态指令的

- src\cpu\o3\iew_impl.hh :1258 `fault = ldstQueue.executeStore(inst);` 前面已经讨论了第一次返回 Fault 的位置；
- src\cpu\o3\dyn_inst_impl.hh :156 `this→fault = this→staticInst→initiateAcc(this, this→traceData);` 实际上这里才是 Fault 真正写入动态指令的位置。

4.3.2. 动态指令是何时设置为执行完成的

1. src\cpu\o3\iew_impl.hh :1218 准备执行却发现已经 Squash 时（无需考虑）
2. src\cpu\o3\iew_impl.hh :1277 当指令进入 StoreQueue 之前就出错时（进入提交队列）
3. src\cpu\o3\lsq_unit_impl.hh :1120 StoreQueue 中的 Store 已经提交到别的内存对象并收到返回的 Packet 之后；（进入提交队列）

---

5. 写存流程分析

2018/11/27 从写文档到现在，我主要沿着官方给出的蓝图，再现了 O3 处理器中读操作的路径。在这个过程之中，我充分了解了：

- O3 处理器访存的流程。Macro 转换为 Micro 指令，然后准备 dispatch 到 scheduler 中，scheduler 乱序 issue 指令，其中 Store 指令将首先缓存在 StoreQueue 中，而 Load 指令首先尝试命中 StoreQueue 然后再尝试发起对 Cache Port 的访问。
- O3 处理器中页错误发起的时机。在访存过程中，只有地址转换时才会发起 PageFault，这显然不符合我的预期，我需要对此进行修改；
- X86 中 FLUSH 指令的实现细节。其实际上是处理为一种特殊的 Store 指令，只是额外带有一些 Request::FLAGS。

现在假设写请求已经最终抵达 WriteQueue 了，接下来我将看看这个写请求是怎么穿过 Cache 并抵达内存的。

5.1. StoreQueue → Cache

1. src\cpu\o3\iew_impl.hh :1453 IEW 周期迭代期间利用一切可用的带宽，尝试回写 Store。`ldstQueue.writebackStores();`；
2. src\cpu\o3\lsq_impl.hh :296 `void LSQ<Impl>::writebackStores()` 尝试将所有活动线程的 Store 回写，核心调用为 `thread[tid].writebackStores();`
3. src\cpu\o3\lsq_unit_impl.hh :783 `void LSQUnit<Impl>::writebackStores()` LSQ 执行单元进行回写。在确定不是空写指令后，将占用 Store 端口并调用 `sendStore(data_pkt)` 发送写请求；
4. src\cpu\o3\lsq_unit_impl.hh :1212 `bool LSQUnit<Impl>::sendStore(PacketPtr data_pkt)` 直接发送 PKG 到缓存端口。核心调用为 `dcachePort→sendTimingReq(data_pkt)`；

5.2. Cache → MSHR

5.3. MSHR → XBar

5.4. XBar → MemCtrl

> 尽管听起来不符合常识，但 GEM5 中是没有通常意义中的 Memory Controller 的。下面所述的 DRAM Controller 不过只能控制一个 Channel。如果要配置多通道以进行 Interleaving 从而进一步提高 Low-level parallelism，还需要从 XBar 入手。GEM5 这样做的原因是，其不打算模拟出系统的每一个细节，更多地只是关注其时序约束。

> Bank-level Parallelism 可 Overlapped 执行同一个芯片内的多个 DRAM Access；Channel-level Parallelism 更厉害了，就连总线传输都可以是 Overlapped。GEM5 中 Interleave 与否是在 add_range 时设置的。

- src\mem\dram_ctrl.cc :616 `DRAMCtrl::recvTimingReq(PacketPtr pkt)` 运行调度器标注权重之后将读写操作添加到相应队列中；
- src\mem\dram_ctrl.cc :518 `DRAMCtrl::addToWriteQueue(PacketPtr pkt, unsigned int pktCount)` 注释中写到，此处仅仅添加到写队列中。请求将在设置 readyTime 并调用 schedule() 之后最终完成；
- src\mem\dram_ctrl.cc :917 `DRAMCtrl::accessAndRespond(x2)`
- src\mem\abstract_mem.cc :327 `AbstractMemory::access(PacketPtr pkt)`当 isRead 的时候完成实际的内存访问，当 isInvalidate、isClean 的时候（但不得是写）什么也不做，当 isWrite 的时候完成实际的内存访问；

> 2018/11/29 其中有些细节看不懂，主要是注释说的功能和它实现的功能不一样，给我整懵了。想着魏博可能看过源码就去问了，结果了解到 ThyNVM 之前的做法并不是这样，以前是自己搞了一个 DRAMBanks 类，其简单地实现了 DRAM 中的一些重要概念，比如 Bank、RowBuffer。而且 DRAMCtrl 的实现太复杂，看它还不如看 NVMain。
> 最后，魏博给我建议说，还是根据我的需求决定使用什么样的工具，我想了一下，我最深只需要用到不同 Bank 之间能并行访问的特点，从这一层意义来说 ThyNVM 之前的实现就能够满足我的需求了。所以我决定停止该部分源码的阅读。

5.5. Cache → CPU

1. src\cpu\o3\lsq_impl.hh :344 `bool LSQ<Impl>::recvTimingResp(PacketPtr pkt)` 调用了 `thread[cpu→contextToThread(pkt→req→contextId())].completeDataAccess(pkt);` 开始完成整次访问；
2. src\cpu\o3\lsq_unit_impl.hh :94 `void LSQUnit<Impl>::completeDataAccess(PacketPtr pkt)` 调用了 `writeback(inst, pkt);` 表示回写完成；
3. src\cpu\o3\lsq_unit_impl.hh :1108 `void LSQUnit<Impl>::writeback(DynInstPtr &inst, PacketPtr pkt)` 将指令设置为已执行并送入提交队列。

5.6. CPU

但是！在 `writeback` 中又调用了 `inst→completeAcc(pkt)`，这是可以返回错误的！好激动，开始分析了！

- src\cpu\o3\dyn_inst_impl.hh :165 `BaseO3DynInst<Impl>::completeAcc(PacketPtr pkt)`，其中有这样一句调用！`this→fault = this→staticInst→completeAcc(pkt, this, this→traceData);` 在指令上下文中设置错误！调用静态指令的 complete 方法！
- src\arch\x86\isa\microops\ldstop.isa :211 只是简单地对 Store 指令的一个包装，默认不会出错。

```
%(op_decl)s;
%(op_rd)s;
%(complete_code)s;
%(op_wb)s;
return NoFault;
```

6. GEM5 中断概览

6.1. 中断控制器的设置

附录中有一篇「i8259A中断控制器分析」，我看了这篇博客之后有一种茅塞顿开的感觉，希望也能对你有所启发。

根据 Intel 手册，x86 平台中 APIC 主要分为两类：local APIC，I/O APIC。从位置上讲，前者集成于 Processor（的 Uncore 部分），而后者坐落于 System Chip Set。从逻辑关系上讲，后者可以作为前者的中断源之一，前者还能接受 Locally connected I/O devices 等 local interrupt sources。

> *The local APIC can also receive interrupts from externally connected devices through the I/O APIC.*
>
> — *Intel® 64 and IA-32 Architectures Software Developer's Manual #3017*

6.1.1. 南桥中断控制器

1. src\dev\x86\SouthBridge.py :46 南桥中有几个重要的成员如下，此前阅读代码时觉得很困惑的几个芯片在此出现。

   ```
   X86ISA::I8254 * pit;
   X86ISA::I8259 * pic1;
   X86ISA::I8259 * pic2;
   X86ISA::Cmos * cmos;
   X86ISA::Speaker * speaker;
   X86ISA::I82094AA * ioApic;
   ```

2. src\dev\x86\SouthBridge.py :89 在 `def attachIO()` 中上述各芯片建立了彼此之间的连接。其中：ioApic 是 pic1 的主片（Master）,pci1 是 pic2 的主片。键鼠直接连接在了 ioApic 的引脚上。

   ```
   self.io_apic.external_int_pic = self.pic1
   ```

这些芯片位于南桥，属于 ChipSet 的组成部分，作为 Platform 上单独模组的处理器不能直接将引脚接到这些芯片上。

6.1.2. 控制器内部中断控制器

python routine

1. src\arch\x86\X86LocalApic.py `cxx_class = 'X86ISA::Interrupts` 据此可判断 GEM5 中 Interrupts 类即 LocalApic 的实现；

2. configs\common\CacheConfig.py :50 `def config_cache` 在配置 Cache 的时候创建实例化 X86LocalApic 并完成端口的连接；

   a. src\cpu\BaseCPU.py :242 `createInterruptController()` 方法实际完成 X86LocalApic 的创建。具体而言产生了 numThreads 个（无超线程时等于处理器数）LocalApic，并指定了其各自的时钟，且地址映射了其寄存器地址使得能够直接通过 mov 读写该寄存器。

      ```
      self.interrupts = [X86LocalApic(clk_domain = self.apic_clk_domain,
                                      pio_addr=0x2000000000000000)
                         for i in xrange(self.numThreads)]
      _localApic = self.interrupts
      ```

> *Each local APIC consists of a set of APIC registers and associated hardware that control the delivery of interrupts to the processor core and the generation of IPI messages. The APIC registers are memory mapped and can be read and written to using the MOV instruction.*
>
> — *Intel® 64 and IA-32 Architectures Software Developer's Manual #2017*

b. configs\common\CacheConfig.py :159 端口连接，将所有缓存端口连接在 cached_bus（左参数，此处为 system.tol2bus），将所有非缓存端口连接在非缓存端口上（右参数，此处为 system.membus）。所有 slave 都将连接在 bus.master 上，所有 master 都将连接在 bus.slave 上。其中 system.membus 实现起来是一个 coherent_xbar。

```
if options.l2cache:
    system.cpu[i].connectAllPorts(system.tol2bus, system.membus)
elif options.external_memory_system:
    system.cpu[i].connectUncachedPorts(system.membus)
else:
    system.cpu[i].connectAllPorts(system.membus)
```

c. src\cpu\BaseCPU.py :235 这里仔细分辨可以发现只是将要执行的代码放入列表，这些代码最终将在 被 connectAllPorts 调用执行。

```
_uncached_slave_ports = []
_uncached_master_ports = []
if buildEnv['TARGET_ISA'] == 'x86':
    _uncached_slave_ports += ["interrupts[0].pio",
                              "interrupts[0].int_slave"]
    _uncached_master_ports += ["interrupts[0].int_master"]
```

上面这些看着比较乱，这里稍微整理一下。在实例化 BaseCPU 之时，即有代码碎片表明集成在处理器之中的 local apic（区别于南桥上的 io apic）的端口的连接方式，但此时只是将代码封装进了字符串，而并执行。直到继续补充系统细节的时候，若选项中开启了 l2 Cache，才创建 local apic 并真正执行连接 local apic 端口的代码。

cpp routine

1. src\cpu\base.cc :128 构造 `BaseCPU` 的时候使用 Python 代码传递过来的参数构造 `std::vector<TheISA::Interrupts*> interrupts;` 成员，也即 local_apic，此后为每个 Thread 绑定中断控制器，`interrupts[tid]→setCPU(this);`。当然，后面也会在 Thread 切换处理器时作相应切换；
2. src\arch\x86\interrupts.cc ：592 这里是该类的构造方法，其中发生的事情可以理解为芯片的 Reset。表示当前正在处理的中断 pendingxxi 及其向量 xxVector 全部指令，并将仅用于处理可屏蔽中断（阅读代码中的注释得，除 ExtInt 之外的所有中断均为UnmaskableInt）的 ISRV 及 IRRV 设置为 0，表示当前没有任何 ExtInt 中断请求，且没有服务任何中断。

6.2. 中断的发起

脑壳疼，看起来 local apic 中发起中断的接口有很多。只能通过类比来排除了。

经调研，发现 IO Apic（I82094）调用消息接口向 local apic 发送中断请求。

```
intMasterPort.sendMessage(apics, message, sys->isTimingMode());
```

然而这是基于 Membus 的 通信方式，不是我所需要关注的。我关注的应是类似于直接级联的方式。

PIC1（I8259）向 I82094 发送中断的接口为 `raiseInterruptPin`，其继续调用 `signalInterrupt`，接着调用 `requestInterrupt`，实质上等同于调用了 `output→raise();` 。看起来有点意思了，那么这个 output 是在哪里设置连接的呢？

src\dev\x86\SouthBridge.py :92

```
[X86IntLine(source=self.pic1.output, sink=self.io_apic.pin(0)),
```

是在南桥里作为 io_apic.pin[0] 的 source。接下来最终由 I82094 中的 `raiseInterruptPin` 方法调用 `signalInterrupt`，其读取该 line 相应 Interrupt Command Register (ICR) 中的 entry，组装成消息，通过 membus 向 local apic 发送中断请求。

至此，我觉得流程已经很明了了。我应该关注 `raiseInterruptPin` → `signalInterrupt` 这一部分的实现。需要注意的是这部分在 local apic 中未实现。

6.3. 中断的处理

1. src\cpu\o3\commit_impl.hh :809 `DefaultCommit<Impl>::commit()` 首先检查是否存在中断 `cpu→checkInterrupts(cpu→tcBase(0))`，存在才传播。
   a. src\cpu\base.hh :247 `checkInterrupts`，`return FullSystem && interrupts[tc→threadId()]→checkInterrupts(tc);`
   b. src\arch\x86\interrupts.cc :610 `X86ISA::Interrupts::checkInterrupts()`

```
bool
X86ISA::Interrupts::checkInterrupts(ThreadContext *tc) const
{
    RFLAGS rflags = tc->readMiscRegNoEffect(MISCREG_RFLAGS);
    if (pendingUnmaskableInt) {
        DPRINTF(LocalApic, "Reported pending unmaskable interrupt.\n");
        return true;
    }
    if (rflags.intf) {
        if (pendingExtInt) {
            DPRINTF(LocalApic, "Reported pending external interrupt.\n");
            return true;
        }
        if (IRRV > ISRV && bits(IRRV, 7, 4) >
               bits(regs[APIC_TASK_PRIORITY], 7, 4)) {
            DPRINTF(LocalApic, "Reported pending regular interrupt.\n");
            return true;
        }
    }
    return false;
}
```

2. src\cpu\o3\commit_impl.hh :784 `DefaultCommit<Impl>::propagateInterrupt` 当提交状态不为 TrapPending、 interrupt 等时，通过 `interrupt = cpu→getInterrupts();` 获取异常。
   a. src\cpu\o3\cpu.cc :966 `return this→interrupts[0]→getInterrupt(this→threadContexts[0]);`。为什么是 0 呢？因为在 GEM5 中至少全系统模式下处理器是不支持超线程的，也即一个处理器仅支持单线程。

      错误登记在 src\cpu\base.hh :211 `std::vector<TheISA::Interrupts*> interrupts;`

   b. src\arch\x86\interrupts.cc :640 `getInterrupt()`，`return std::make_shared<ExternalInterrupt>(IRRV)`

c. src\arch\x86\faults.hh :381 `class ExternalInterrupt : public X86Interrupt`

> `ExternalInterrupt` 有唯一的构造函数，其参数为 vector，据查，构造时使用 IRRV。

3. src\cpu\o3\commit_impl.hh :964 `DefaultCommit<Impl>::commitInsts` 在提交指令的时候检查中断情况 `interrupt != NoFault`，并在发生错误的时候执行 `handleInterrupt`
4. src\cpu\o3\commit_impl.hh :730 `DefaultCommit<Impl>::handleInterrupt` 处理中断
5. src\cpu\o3\cpu.cc :974 `FullO3CPU<Impl>::processInterrupts` 中最终发起中断 `this→trap(interrupt, 0, nullptr);`
    - `trap` 之前的动作为 `updateIntrInfo()`，其将 IRRV 赋值给 ISRV，在通过 APIC 地址 APIC_IN_SERVICE_BASE 设置 APIC 中 ISRV 寄存器后，清除 IRRV 寄存器并设置新值。

6.4. 部分特殊类

在实现过程中，有些概念我有点搞不清，这里把它们整理一下。

IntLine

src\dev\x86\intdev.hh :277 其对应中断系统中的「中断线」概念，是连接两个中断控制器 Source 及 Sink 的信号线的实体。看了它的声明，其唯一的方法就是其构造方法，将执行 `source→addSink(sink);`，从而在构造时即建立脱离 IntLine 控制的 Source 及 Sink 连接关系。

中断线的概念贯穿中断控制器的设计（Local APIC 除外，其没有相关实现），i8259 中 ISR 的低七位分别对应一个中断线，其相应位置高表示正在服务该中断。

IntSourcePin

src\dev\x86\intdev.hh :185 中断线的 Source 端，其主要包括三个方法：`addSink(IntSinkPin *sink)`、`raise()`、及 `lower()`，分别完成：在 sinks 成员（一个向量）中保存参数相应 sink、使得 sinks 中所有项执行 `pin.device→raiseInterruptPin(pin.number);`、使得 sinks 中所有项执行 `pin.device→lowerInterruptPin(pin.number);`。

IntSinkPin

src\dev\x86\intdev.hh :185 其作用小得多，看了下其成员方法，也就需要主要起 device 成员在构造方法中设置。构造时，构造参数从 src\dev\x86\X86IntPin.py :44 中获取，实际位置一般都是某具体中断控制器创建的时候。所以定义 pin 方法的中断控制器都必定对应一个 sink 端。

```
def pin(self, line):
    return X86IntSinkPin(device=self, number=line)
```

Appendix A: 常用位置

impl

此处为 O3CPUImpl，定义在 src\cpu\o3\impl.hh :53

DynInst

其模板定义在 src\cpu\o3\dyn_inst.hh :60。另外其继承了 BaseDynInst，其定义在 src\cpu\base_dyn_inst.hh :79；

Appendix B: 名词解释

B.1. 处理器相关

iTB & dTB

O3 处理器中分别为指令和数据实现了不同的 TLB，即为 iTB 和 dTB。

ROB

The re-order buffer

MSHR

Miss Status and handling Register. cache miss handler

B.2. 内存相关

Ipr

Internal Processor Registers。当寄存器映射到物理地址上时，这里称之为 MmappedIpr。

LLSC

The request is a Load locked/store conditional。对应内存一致性中 Safety-net 部分，用于实现内存原子读取/写入。

UnalignedMemAcc

Unaligned Memory Access。形式化表示即为 `addr % N != 0`。

Internal address space

The internal address space corresponds to that processor’s own internal address space, Ureg registers. - ADSP-TS201 TigerSHARC ® Processor Hardware Reference

Protected mode

操作系统课提到的实模式、保护模式、虚拟 8086 模式中的第二种。保护模式增加了分段分页机制，为实现 VM 提供了硬件支持。

B.3. 中断相关

Interrupt

通常分为三类：Exception、Interrupt Request (IRQ) or Hardware Interrupt、Software Interrupt。详情参考 OSDev

- Exceptions：分文三类，Faults、Traps、Aborts。
  - Faults: These can be corrected and the program may continue as if nothing happened.
  - Traps: Traps are reported immediately after the execution of the trapping instruction.
  - Aborts: Some severe unrecoverable error.

APIC

Advanced Programmable Interrupt Controller

IRR

the interrupt request register (IRR)；后缀 v 则表示 vector

ISR

or in-service register (ISR)

> *The 256 bits in these registers represent the 256 possible vectors; vectors 0 through 15 are reserved by the APIC*
>
> — *intel*
> *Intel® 64 and IA-32 Architectures Software Developer's Manual #3045*

SMI

SMM is entered through activation of an external system nterrupt pin (SMI#), System management mode (SMM) which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.

> *The IA-32 architecture supports three operating modes and one quasi-operating mode: Protected mode, Real-address mode, System management mode (SMM), Virtual-8086 mode, IA-32e mode*
>
> — *intel*
> *Intel® 64 and IA-32 Architectures Software Developer's Manua #2723*

NMI

Exception or non-maskable interrupt (NMI)

EOI

end-of interrupt, 该寄存器只写

LVT

local vector table (LVT), which allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers. 详见《手册》#3028。

Appendix C: 资源整理

- GEM5 对 O3 处理器的介绍
- gem5 源代码文件结构作用介绍
- 操作系统名词词典
- GEM5 Code parsing
- ThyNVM: 怎么实现混合内存
- ThyNVM: 旧版实现
- Priority Among Simultaneous Exceptions and Interrupts，见 《Intel® 64 and IA-32 Architectures Software Developer's Manua 》#2848
- Protected-Mode Exceptions and Interrupts，见 《Intel® 64 and IA-32 Architectures Software Developer's Manua 》#2842

- i8259A中断控制器分析

Last updated 2019-01-20 15:17:15 中国标准时间