用户态中断
UINTC控制器实现

张宇轩@计96

https://github.com/yuxuan-z19/uintc

操作系统可以利用中断...

- 实时响应外设
- 计时
- 跨核通信

应用程序

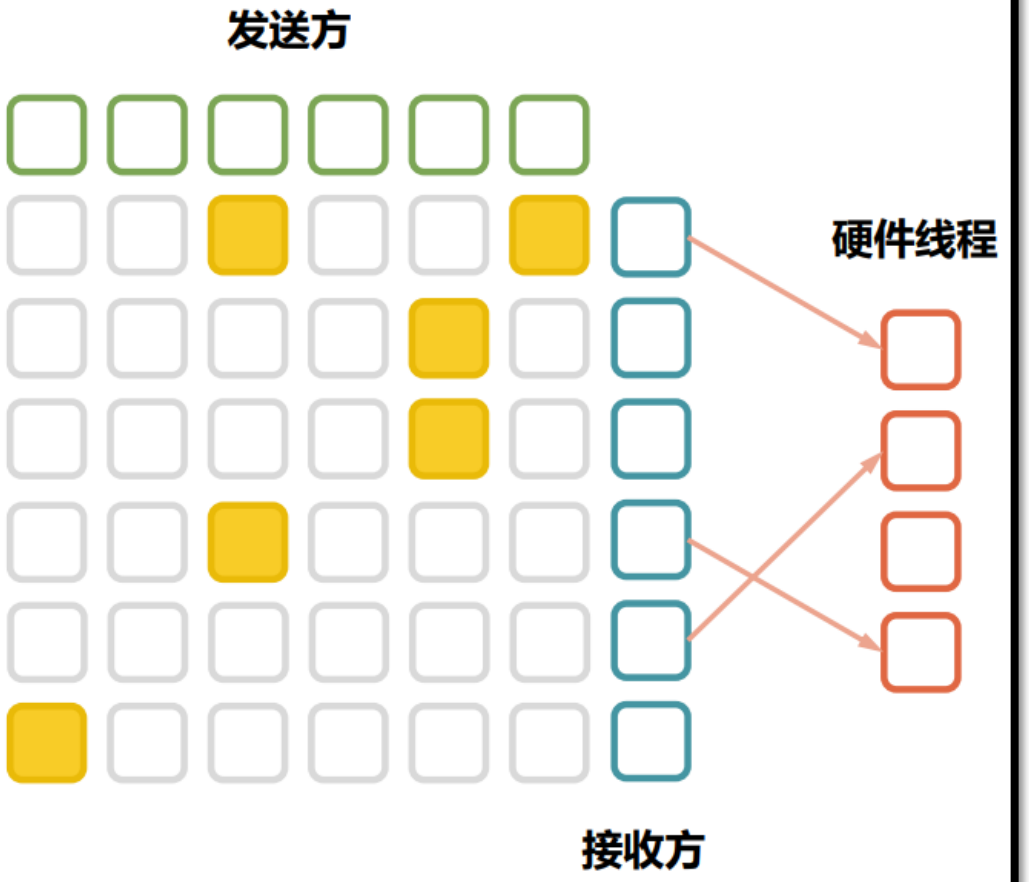
- 类似现用系统调用实现
  - 通过系统调用操作外设
  - 信号等 IPC 机制

- 用户态驱动
- 不用陷入内核的 IPC

IPC type	Relative Latency (normalized to User IPI)
User IPI	1.0
Signal	14.8
Eventfd	9.7
Pipe	16.3
Domain	17.3

UINTC-MAT

- 发送方数 × 接收方数 的 enable / pending 矩阵
- 每个发送方 / 接收方槽位记录一软件编号 uiid，对应应用程序申请的发送 / 接收口
- 每个上下文（对应硬件线程）记录监听的接收方编号 receiver_id



问题 即使基于reg，二维矩阵的索引会带来很大的查找开销
解决 每个接收方维护独自的消息队列，内容为发送方UIID/索引
思考 如何维护多个任务队列的资源？Block RAM => 地址映射？

项目进展

W01~02 确定选题

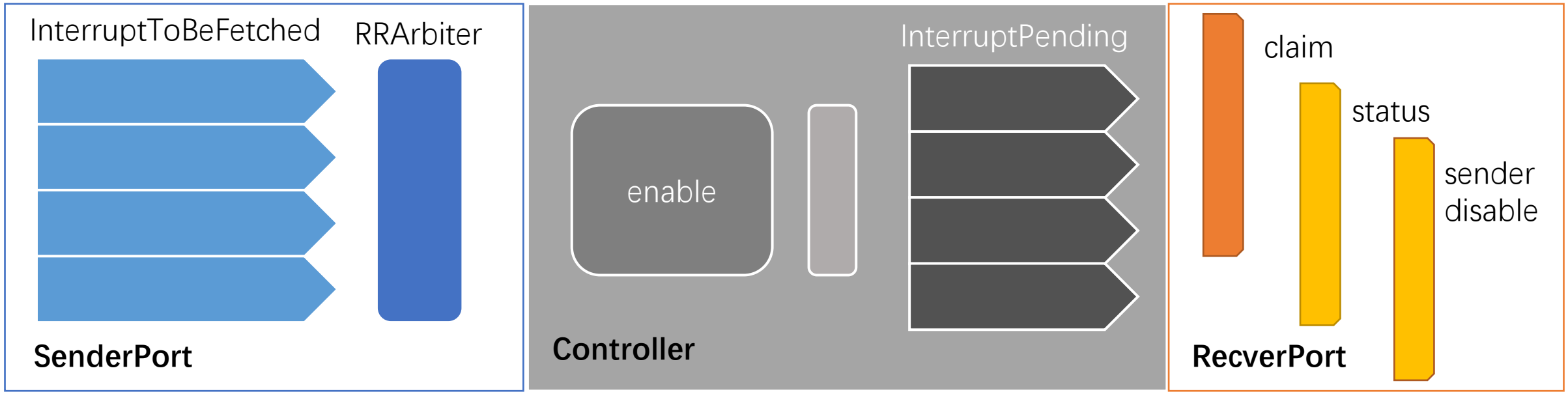
W03~05 在ZCU104上复现labeled（未运行rCore-N）

W05~06 调研rocket chip的PLIC、CSR和核间中断

W07~08 按照规范实现邻接矩阵的UINTC控制器

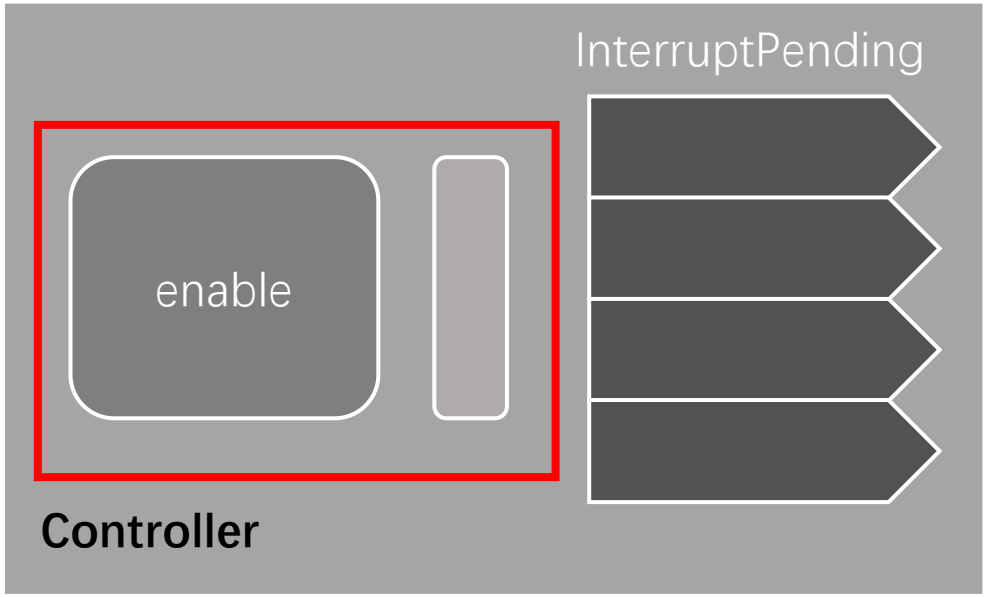
W09 完成UINTC控制器的”邻接表”实现，准备接入BRAM

未来 rCore-N + UIPI with labeled + UINTC



与UINTC相同 维护enable（允许中断请求）、claim（接收中断请求）、status（查询中断请求是否领取），控制器在Block RAM上实现

与UINTC不同 pending采用邻接(表)队列实现，新增sender disable来强制不响应历史上发出的中断请求



操作码规范如下：

op[2]	op[1:0]
`0` 设置enable，`1` 设置pending	`00` 取消设置，`01` 设置，`11` 清空

采用握手信号和冗余位来保证安全性

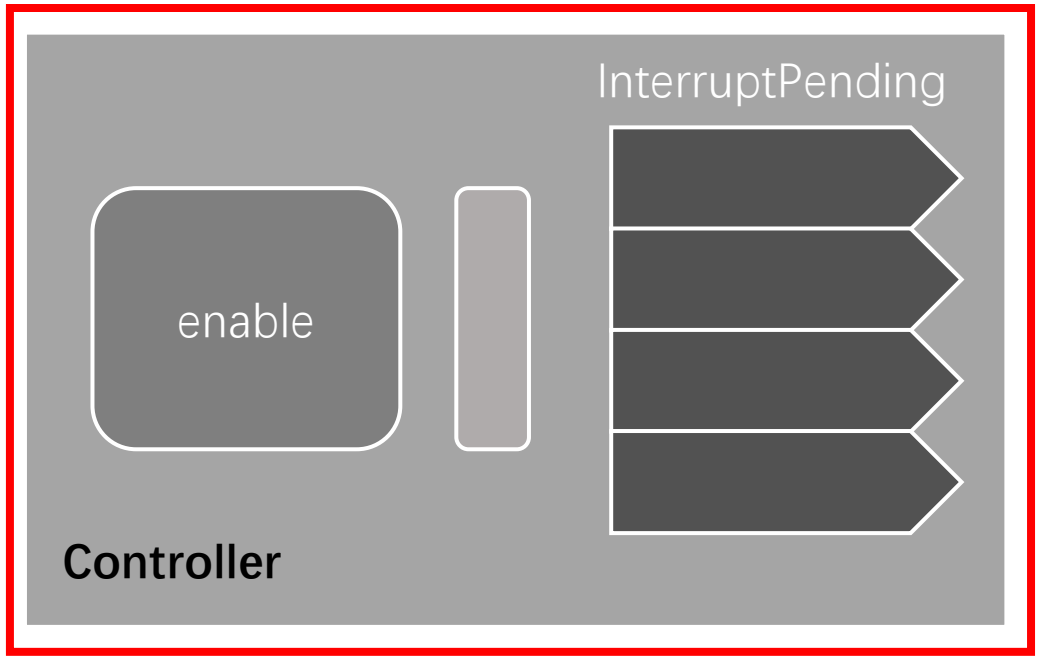
> 清空请求发生 ⇔ 切换上下文 ⇒ 清空当前的 `enable` 和 `pending` ，因此**特别约定**当低2位全为1时清空 `enable` 和 `pending`

```scala
class EnableMap(implicit p: config.Parameters) extends Module {
    val io = IO(new Bundle {
        val src_request = Flipped(Valid(new Request))
        val sink_request = Valid(new Request)
        val sink_enable = Output(Bool())
    })

    val src_id = io.src_request.bits.idx.src
    val dst_id = io.src_request.bits.idx.dst

    val bitmap = RegInit(
      VecInit(Seq.fill(p(SOCKET_CNT))(0.U((1 << log2Ceil(p(SOCKET_CNT))).W)))
    )
    val bit_set = 1.U << dst_id
    val op = io.src_request.bits.op(1, 0)
    when(io.src_request.valid && !io.src_request.bits.op(2)) {
        when(op === "b01".U) {
            bitmap(src_id) := bitmap(src_id) | bit_set // set
        }.elsewhen(io.src_request.bits.op(1, 0) === "b11".U) {
            bitmap(src_id) := 0.U // clear
        }.elsewhen(op === "b00".U) {
            bitmap(src_id) := bitmap(src_id) & ~bit_set // unset
        }.otherwise {}
    }

    io.sink_request := RegNext(io.src_request)
    io.sink_enable := RegNext(bitmap(src_id)(dst_id))
}
```



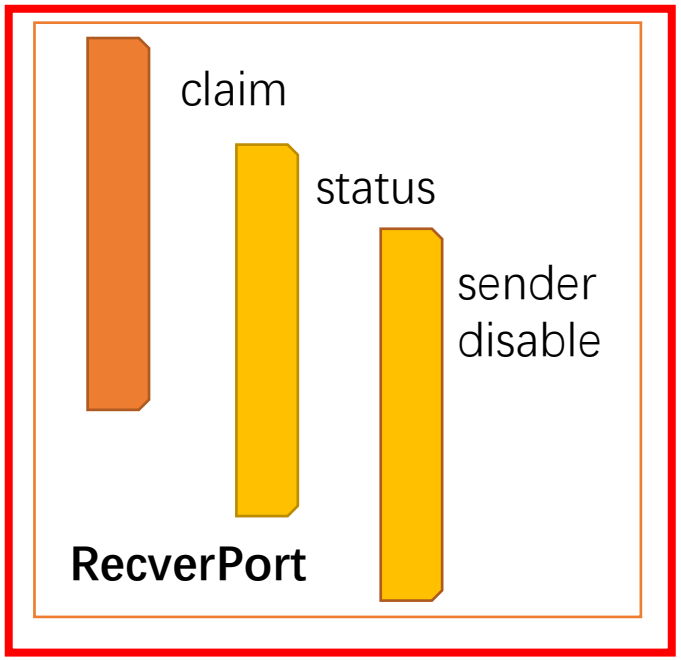

```scala
class Controller(implicit p: config.Parameters) extends Module {
    val io = IO(new Bundle {
        val request_sent = Flipped(Decoupled(new Request))
        val request_recv =
            Vec(p(SOCKET_CNT), Decoupled(UInt(p(LG2_SOCKET_CNT).W)))
    })

    val request_queue = Queue(io.request_sent, 16)

    /* enable reg stage */
    val enable_map = Module(new EnableMap)
    enable_map.io.src_request.valid := request_queue.valid
    enable_map.io.src_request.bits := request_queue.bits
    request_queue.ready := enable_map.io.sink_request.valid
    // once the pipereg is set, we could pop the queue

    /* enable reg stage */
    val pending_queues = Seq.fill(p(SOCKET_CNT)) {
        Module(new Queue(UInt(p(LG2_SOCKET_CNT).W), 16))
    }
    // ? act as adjacency list

    val request = enable_map.io.sink_request.bits
    pending_queues.zipWithIndex.foreach {
        case (q, idx) => {
            q.io.enq.bits := request.idx.src
            when(
                enable_map.io.sink_request.valid && enable_map.io.sink_enable &&
                request.idx.dst === idx.U
            ) {
                q.io.enq.valid := enable_map.io.sink_request.valid && request.op(2)
            }.otherwise { q.io.enq.valid := false.B }
            io.request_recv(idx) <> q.io.deq
        }
    }
}
```




```scala
class RecverPort(implicit p: config.Parameters) extends Module {
    val io = IO(new Bundle {
        val request_recv =
            Flipped(Vec(p(SOCKET_CNT), Decoupled(UInt(p(LG2_SOCKET_CNT).W))))
        val request_disable =
            Input(Vec(p(SOCKET_CNT), Bool())) // maintain by cores
        val recver_claim =
            Vec(p(SOCKET_CNT), Decoupled(UInt(p(LG2_SOCKET_CNT).W)))
        val sender_status = Output(Vec(p(SOCKET_CNT), Bool()))
    })

    (io.recver_claim zip io.request_recv) foreach {
        case (claim, recv) => {
            claim.bits := recv.bits
            claim.valid := recv.valid && io.request_disable(recv.bits) === false.B      You,
            recv.ready := Mux(io.request_disable(recv.bits) === true.B, true.B, claim.ready)
                // if it's disabled just ignore it
        }
    }

    io.sender_status := DontCare
    for (port <- io.recver_claim)
        io.sender_status(port.bits) :=
            RegNext(port.valid && port.ready, init = false.B)
}
```