序号：

硬件课程设计报告 (2020-2021 学年)

姓　名：_____

学　号：_____

班　级：_____

专　业：_____

所在系：_____

指导教师：_____

年　月　日

总成绩：

评语：

指导教师签字：　　日期：

1 目录

1　设计简介

1.1　GadgetMIPS 设计简介

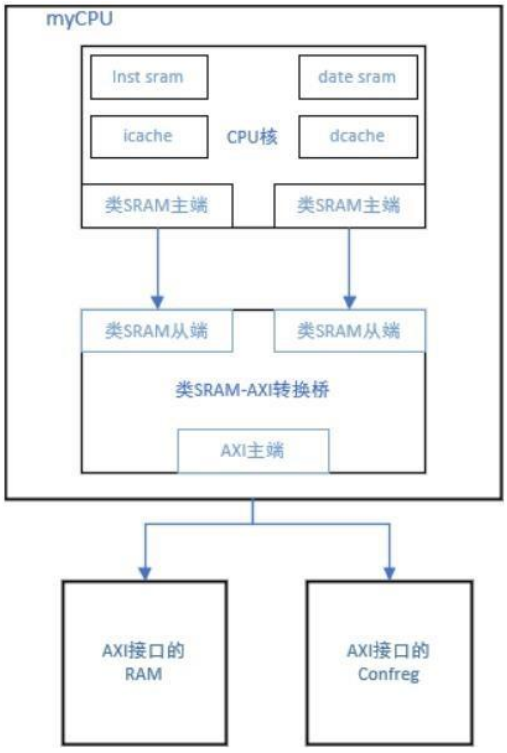
由于龙芯杯大赛的时间与我校小学期的时间重合，所以在吴磊老师的支持下我们选择以赛代练，通过完成龙芯杯的比赛来达到小学期的学习效果。

按照大赛要求和小学期要求设计开发了一个支持 MIPS 精简指令集的 GadgetMIPS 微系统，实现了功能如 下：

1) 在龙芯提供的开发板的基础上，设计实现了一个基于标准 32 位 MIPS 精简指令集的 CPU 并实现了五级流水。不仅可以支持大赛要求的 57 条指令也完美覆盖了小学期要求的相应指令，也可以处理其他 32 条复杂运算指令，设 计了 CP0，支持异常和冲突处理。
2) 设计内部实现类 SRAM 接口，并通过 AXI 转接桥与 CPU 外部进行连接。
3) 挂载指令 Cache 和数据 Cache。

1.2　具体设计思路

1) 五级流水线；
2) 基本指令及运算指令；
3) 异常、冲突处理；
4) 增设 icache、dcache 部分，并集成至取指、访存阶段；
5) SRAM & AXI 转接桥实现对外接口；



1.3 关于 hhhhMIPS

由于后续决赛的查重的限制，我们组并没有进入后续进程，所以基于本组 CPU 设计实际上只做到 了 Cache 部分，后续 TLB，PMON 和 μcore 部分由于二队部分选手几乎全程没有参与，所以以上模块又经由我们去实现。

1.4 源码地址

GadgetMIPS： https://github.com/whiteicey/loongson/tree/master

hhhhMIPS：https://gitee.com/tzwkearn/ncut_cpu_2/tree/tlb

2 CPU 设计

2.1 CPU 整体设计思路与问题

GadgetMIPS 中使用的 icache 和 CPU 部分是基于去年使用的 HikariMIPS 中的 icache 和进行的修改，CPU 和 icache 也是基于《自己动手写 CPU》进行实现，不过由于 HikariMIPS 设计的缺憾，基于 SRAM 架构和缺少 dcache 也成为了相应的掣肘，所以我们在初赛期间的主要任务在于通过《CPU 设计实战》的指导补全 dcache 和对 CPU 进行 AXI 化改造， 但是缺乏经验和鲁莽的选择导致了 CPU 制作的遗憾。

在 CPU 阶段，在 CPU 进行设计时假设其外部直接连接两个 RAM，一个 RAM 中存储着 CPU 要执行的所有指令，另一个 RAM 中存储着 CPU 在执行期间所要留存的数据。在经过了查找了一些资料后，我们决定将这两个 RAM 设置为异步 RAM。因为异步 RAM 是可以在一个周期内执行 完读写操作，这大幅度简化了我们写 CPU 接口的状态机时的难度。

基于前文所说的冒险失败，我们最终放弃了使用 AXI 架构 dcache 而最终采取 SRAM 架构的dcache 作为 CPU 部分的完善，此步骤中我们实现了类 SRAM 到 AXI 的总线桥结构，该总线桥的主要意义在于满足龙芯杯大赛的需求通过 AXI 接口让 CPU 进行数据交互。不过笔者此处也有必要做出一次说明，这样的 SRAM 信号和《CPU 设计实战》此书中采用的 AXI 信号需要非常细致的调整，其中对于总线桥的修改几乎是全盘推翻，所以如果时间充裕的情况下，建议不要将《自己动手写 CPU》和《CPU 设计实战》两书混合使用。

最后便是 cache 部分的修改和补充，由于 AXI 版本的 dcache 我们并没有成功实现，所以最终我们还是只能采取 SRAM 的 dcache，不过考虑到性能影响，最终还是采取了写回的方式配合 二路组相联的方式进行。同样我们在赛后也发现了一个问题，我们的 dcache 因为失误导致写 成了 16k，这样导致我们二路组相联并没有实质性的提高性能，反而导致了降频。

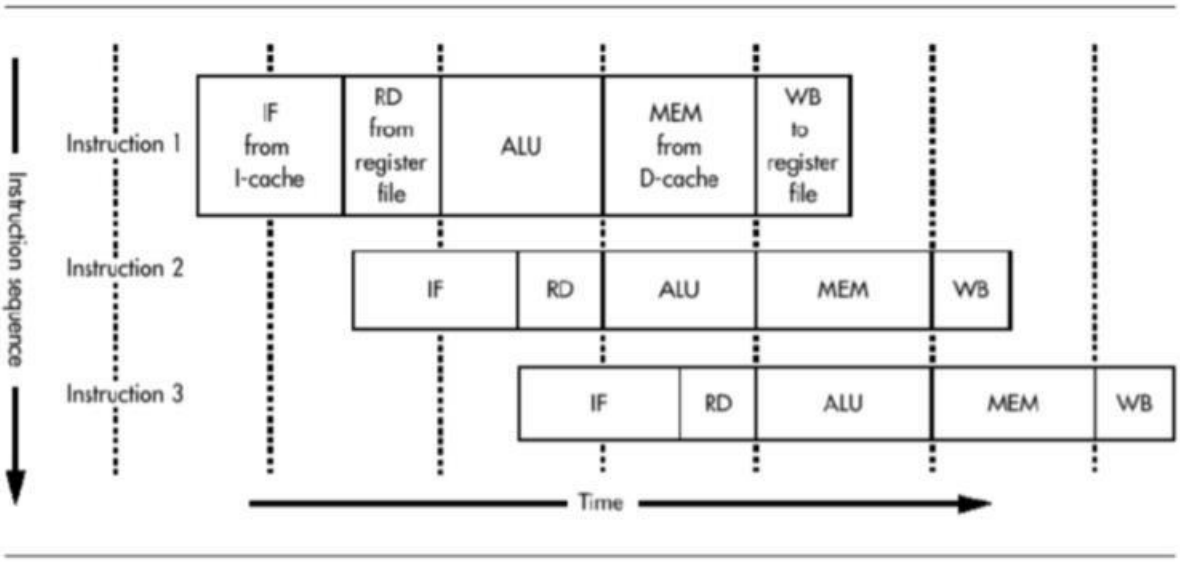
2.2 指令集支持

- 逻辑运算指令 OR, AND, XOR, NOR, ORI, ANDI, XORI, LUI
- 算术运算指令 ADD, ADDU, SUB, SUBU, SLT, SLTU, ADDI, ADDIU, SLTI, SLTIU, MULT, MULTU, MUL, DIV, DIVU
- 移位指令 SLL, SRL, SRA, SLLV, SRLV, SRAV
- 数据移动指令 MFHI, MFLO, MTHI, MTLO, MOVN, MOVZ

- 跳转/分支指令 J, JAL, JR, JALR, BEQ, BNE, BGTZ, BLEZ, BGEZ, BGEZAL, BLTZ, BLTZAL 加载/存储指令 LB, LBU, LH, LHU, LW, SB, SH, SW, LWL, LWR, SWL, SWR, LL, SC特权指令 MFC0, MTC0, ERET
- 自陷指令 BREAK, SYSCALL, TEQ, TNE, TGE, TGEU, TLT, TLTU, TEQI, TNEI, TGEI, TGEIU, TLTI, TLTIU

2.3 流水线设计

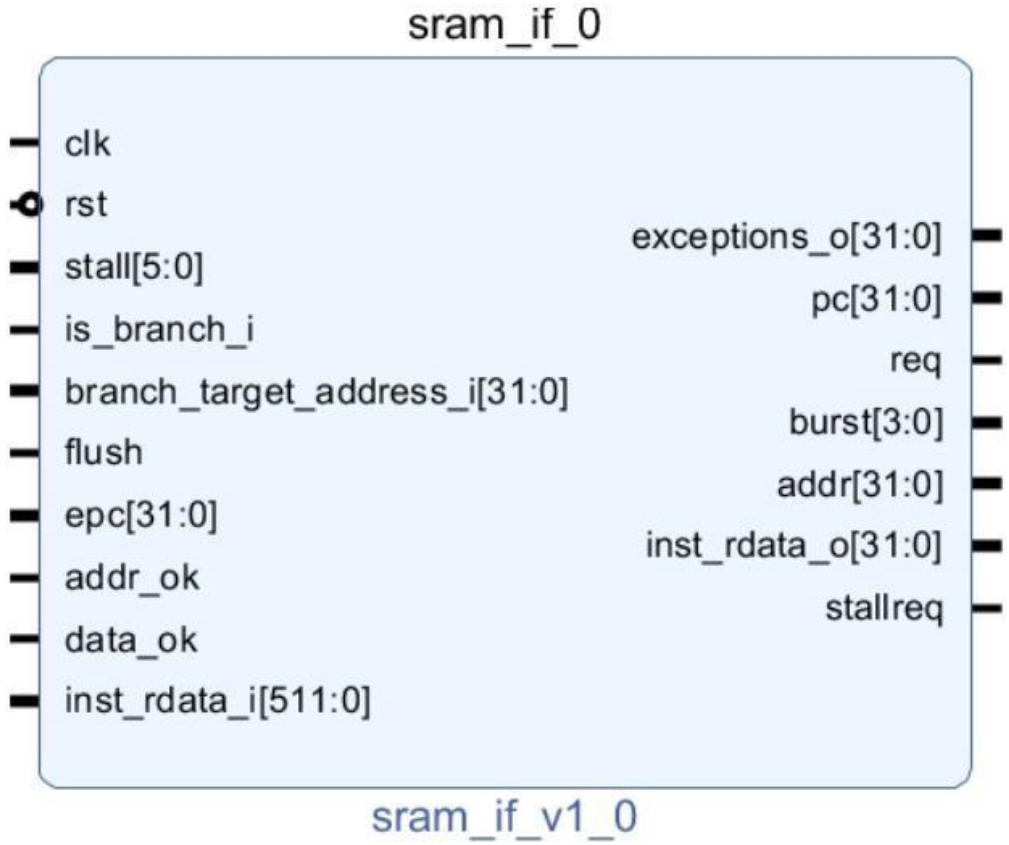
GadgetMIPS 使用 MIPS 经典五级流水线架构，即取指 IF、译码 ID、执行 EX、访存 MEM 和 写回 WB，这使得 CPU 可以更充分地利用其内部不同功能的器件以提高 CPU 执行效率。



其中 ALU 为 EX 中的一部分

GadgetMIPS 通过外设 ram 存储数据，而内部采用指令和数据分开存储的结构，各自采用独 立的 cache 进行数据缓存。

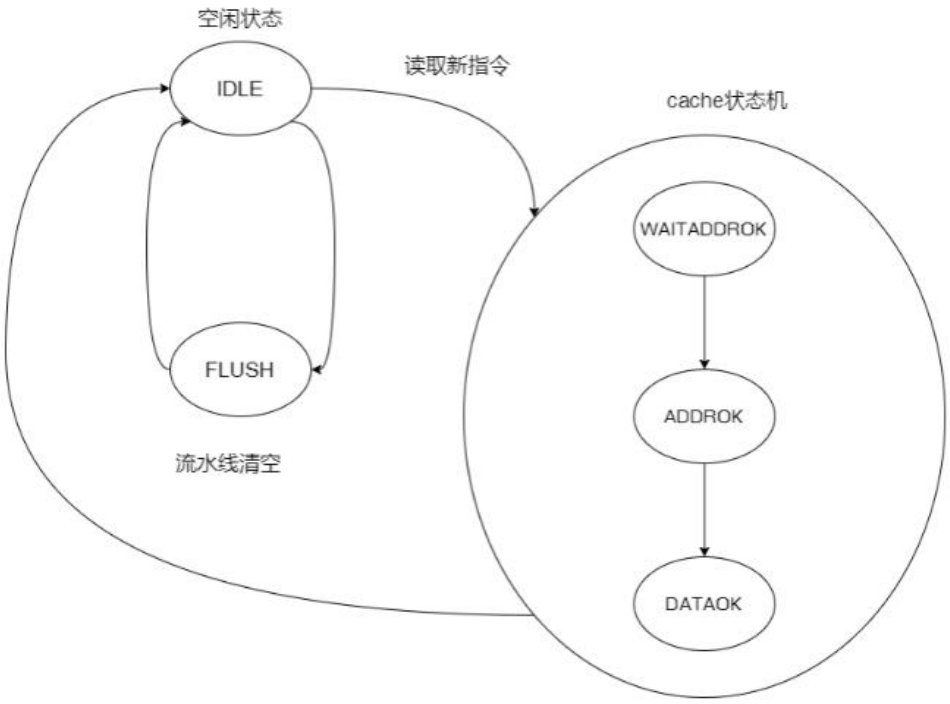
2.3.1 IF 取指阶段



在取指阶段，从外部 RAM/ROM 中读取一条指令，更新程序计数器的值，使其指向下一条指令。GadgetMIPS 通过字节寻址方式访问外部存储器，为了使访存与取指的地址对齐，每次 对齐读取 32 位，即访问[0x00,0x03]中的任意地址都会返回地址为 0x00~0x03 的 4 字节数据，这也是为了迎合 MIPS32R1 指令集中要求访存与取指的地址都必须对齐的要求。

此外，该阶段还会对译码阶段的跳转指令给出响应：译码阶段给出一个跳转信号和一个新的程序地址，该阶段响应该信号并将新地址写入 PC，然后根据新的 PC 取指令。 IF 模块内集成 了 icache，设计为单块 16 个字、单路 256 块的架构，未命中时向外访问。由于取指阶段实际上并不存在写操作，最终只需考虑读取数据并缓存的实现。

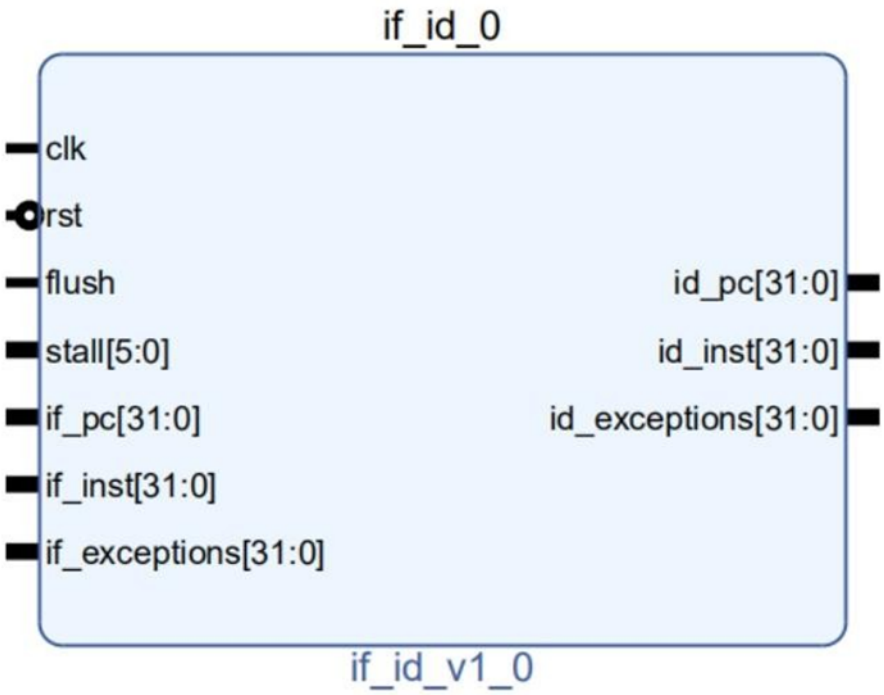
流程状态机转换如下图：icache 将会保持空闲状态，直到新的指令需求提出且未在 cache 内部找到，随后进入标准的 SRAM 握手流程接受内存发来的数据。除此之外，为了实现流水线清空，icache 需要接受控制模块 ctrl.v 发来的重置信号，且这个信号只能在空闲状态接受（无论如何，不应打断数据接收的握手流程）。




```
always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        state <= `IDLE;
        req <= 1'b0;
        flush_wait <= 1'b0;
    end else if (flush) begin
        if(state != `IDLE && state != `DATAOK) begin
            state <= `FLUSHWAIT;
            flush_wait <= 1'b1;
        end
    end else if (ce == 1'b1) begin
        case (state)
            `IDLE: begin
                if(pc[1:0] != 2'b00) begin
                    req <= 1'b0;
                end else if(!hit) begin//cache 读缺失
                    state <= `WAITADDROK;//进入等待地址确认状态
                    req <= 1'b1;
                end else begin
                    req <= 1'b0;
                end
            end
            `WAITADDROK: begin
                if(addr_ok == 1'b1) begin
                    req <= 1'b0;
                    state <= `ADDROK;
                end
            end
            `ADDROK: begin
                if(data_ok == 1'b1) begin
                    state <= `DATAOK;
                end
```

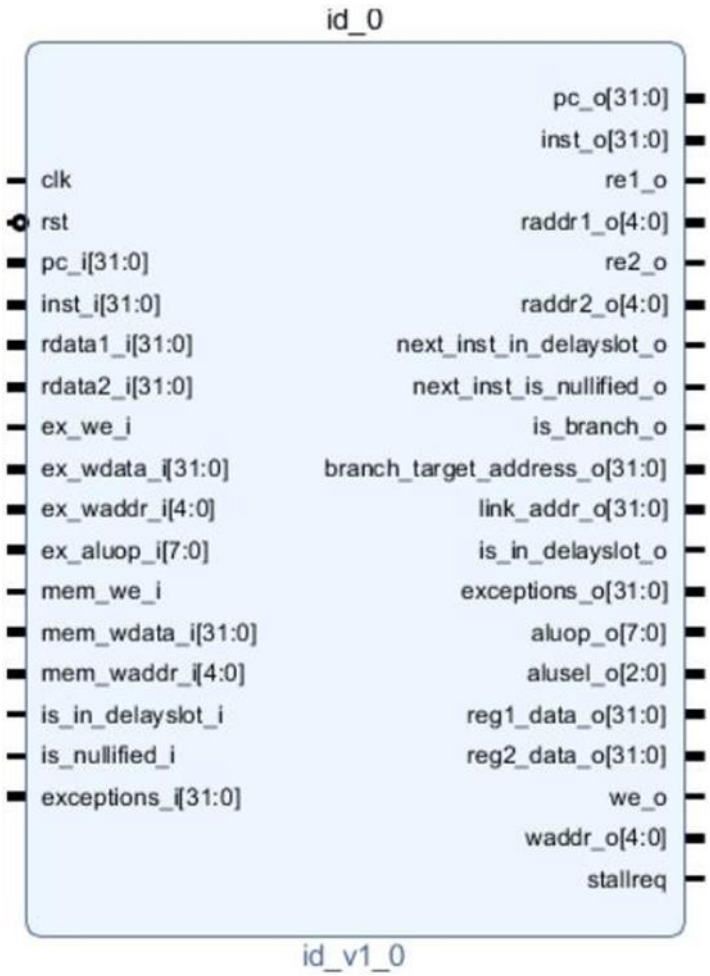
```
            end
            `DATAOK: begin
                state <= `IDLE;
            end
            `FLUSHWAIT: begin
                if(data_ok) begin
                    state <= `IDLE;
                    flush_wait <= 1'b0;
                end
            end
        endcase
    end
end
```

2.3.2 IF/ID 锁存器



取指阶段需要在一个时钟周期内取得指令并更新 PC 值，但如果在跳转指令计算新的 PC 值 时，因为硬件寄存器的延迟决定何时更新 PC，所以 PC 是否已经指向下一条指令不得而知， 那么跳转指令计算的新 PC 值也无法确定是否正确，为避免此情况发生，GadgetMIPS 在取指 和译码两阶段之间加入该寄存器，锁存译码阶段所需要的各种信号并保持一个周期。

2.3.3 ID 译码阶段

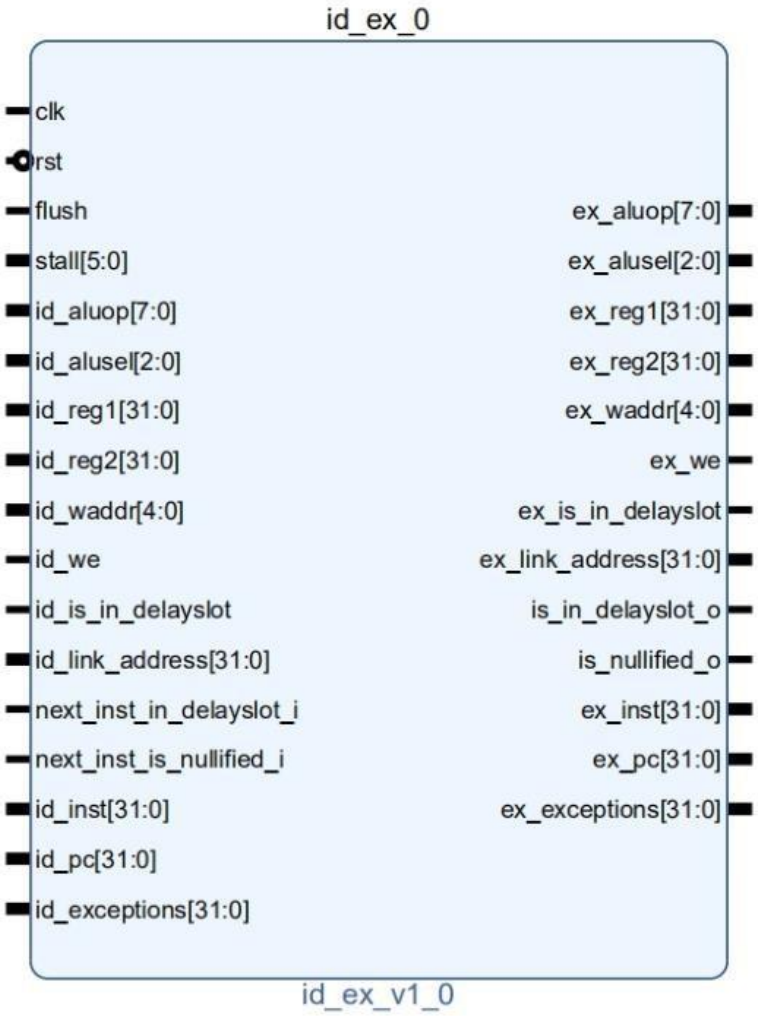


本阶段主要负责将取得的指令通过译码生成后续各阶段所需要的控制信号，同时，本阶段将得到执行阶段所需要的一个或两个操作数。此外，本阶段还负责对部分指令进行条件判断， 比如根据译码结果判断带条件数据转移指令是否需要进行数据转移。此处最重要的改动在于 TLB 的支持，因为 TLB 中可能出现 4 中异常，所以我们对原先的 32 位异常处理信号 exceptions_o 在高位进行了扩充（若出现一条未规定的指令我们一般直接解析为 NOP，这样 的处理方式便于后续的扩充）

```
always @ (posedge clk) begin
    if(rst == `RstEnable) begin
        exception_is_refetch = 1'b0;
    end else begin
        case (exception_is_refetch)
            1'b0: begin
                if(exception_is_tlbwi |
                   exception_is_tlbwr |
                   exception_is_tlbp  |
```

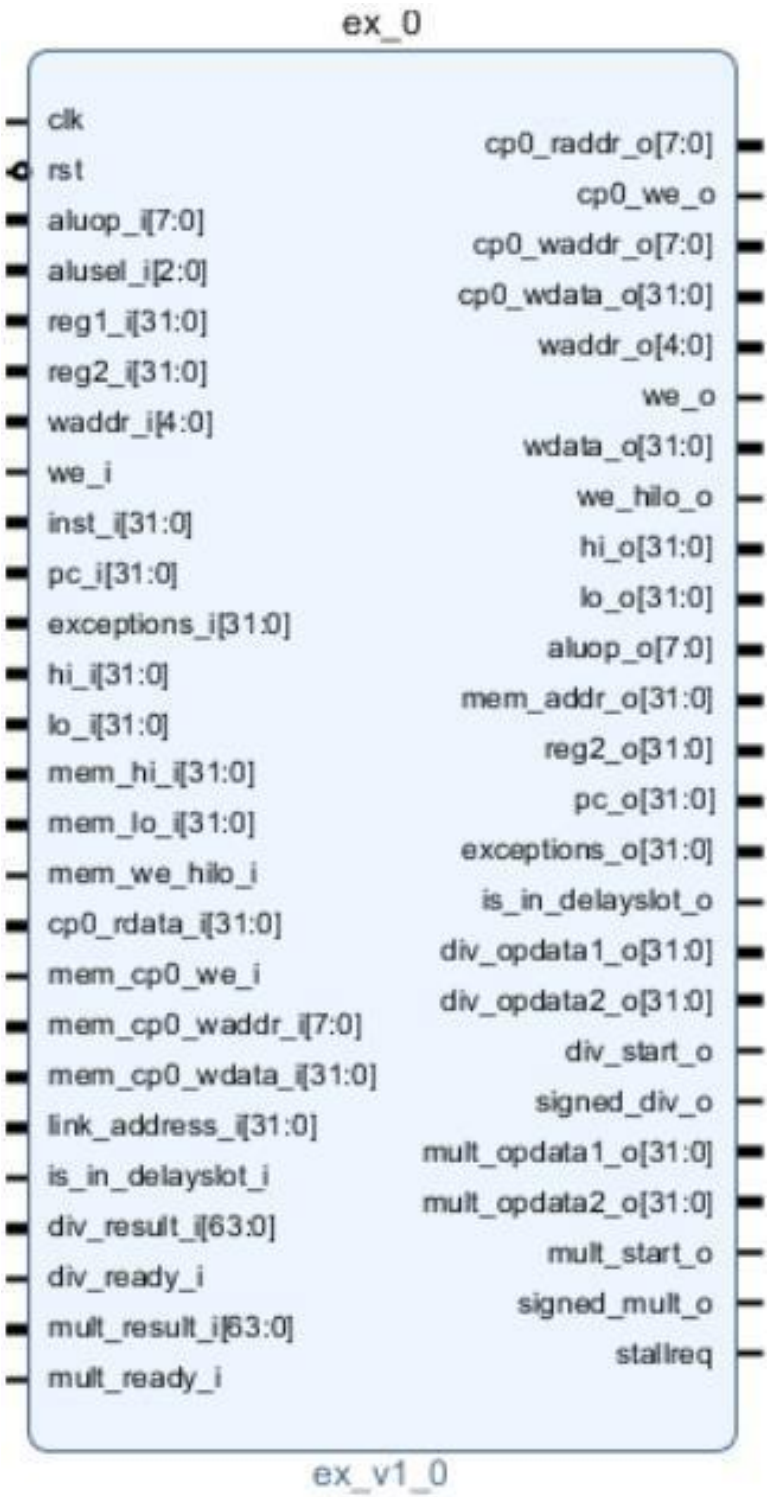
```
                    exception_is_tlbr) begin
                     exception_is_refetch <= 1'b1;
                end else begin
                    exception_is_refetch <= 1'b0;
                end
            end
            1'b1: begin
                exception_is_refetch <= 1'b0;
            end
            default: begin
            end
        endcase
    end
end

...
assign exceptions_o = {exception_is_tlbp,
                       exception_is_tlbr,
                       exception_is_tlbwi,
                       exception_is_tlbwr,
                       exception_is_refetch,
                       exceptions_i[26:5],
                       exception_is_syscall,
                       exception_is_break,
                       exception_is_eret,
                       inst_valid,
                       exceptions_i[0]};
```

2.3.4 ID/EX 锁存器



该锁存器不仅需要锁存必要信号并保持一个周期，还要利用其一拍延迟锁存状态信号以提供 指示给下一条指令。比如下一条指令是否被无效化，是否在延迟槽中。

2.3.5 EX 执行阶段

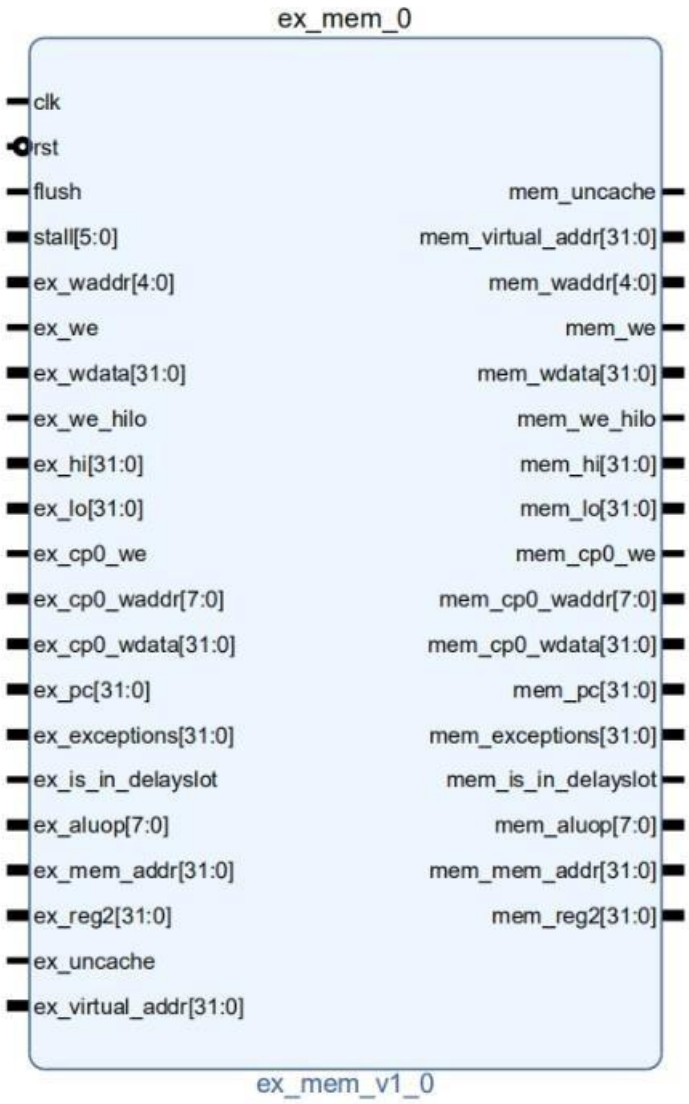


本阶段根据译码阶段给出的控制信号进行运算，包括逻辑运算，移位运算以及加减乘运算， 而除法运算则是构建多周期除法器进行运算，多周期运算需要暂停流水线等待运算完成。此 处增加的改动主要有两处，其一为：对 TLB 指令的执行、异常信号的判断处理；其二为：针 对流水线化 cache，对地址信号命中的预判断。

```
assign exceptions_o = {exceptions_i[31:12],
                       tlb_mapped & load_store ? (((tlb_hit & store) & (tlb_v & !tlb_d)) ? 1'b1 : 1'b0) : 1'b0,
                       tlb_mapped & load_store ? !tlb_hit : 1'b0,
                       tlb_mapped & load_store ? (tlb_hit & !tlb_v ? 1'b1 : 1'b0) : 1'b0,
                       exceptions_i[8:7],
                       trap_occured,
                       overflow_occured,
                       exceptions_i[4:0]};
```

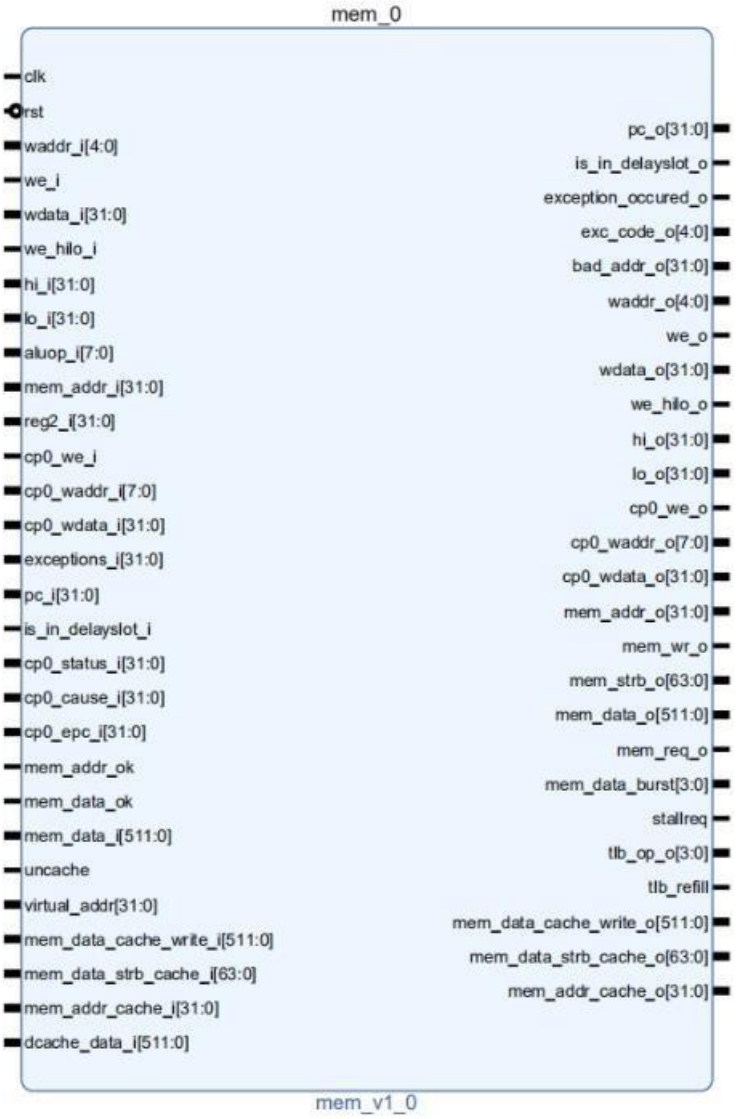
```
// 判断是否为数据操作
    wire load_store = (aluop_i == `MEM_OP_LB  |
                       aluop_i == `MEM_OP_LH  |
                       aluop_i == `MEM_OP_LWL |
                       aluop_i == `MEM_OP_LW  |
                       aluop_i == `MEM_OP_LBU |
                       aluop_i == `MEM_OP_LHU |
                       aluop_i == `MEM_OP_LWR |
                       aluop_i == `MEM_OP_SB  |
                       aluop_i == `MEM_OP_SH  |
                       aluop_i == `MEM_OP_SWL |
                       aluop_i == `MEM_OP_SW  |
                       aluop_i == `MEM_OP_SWR);

    wire store = (aluop_i == `MEM_OP_SB  |
                       aluop_i == `MEM_OP_SH  |
                       aluop_i == `MEM_OP_SWL |
                       aluop_i == `MEM_OP_SW  |
                       aluop_i == `MEM_OP_SWR);
```

2.3.6 EX/MEM 锁存器



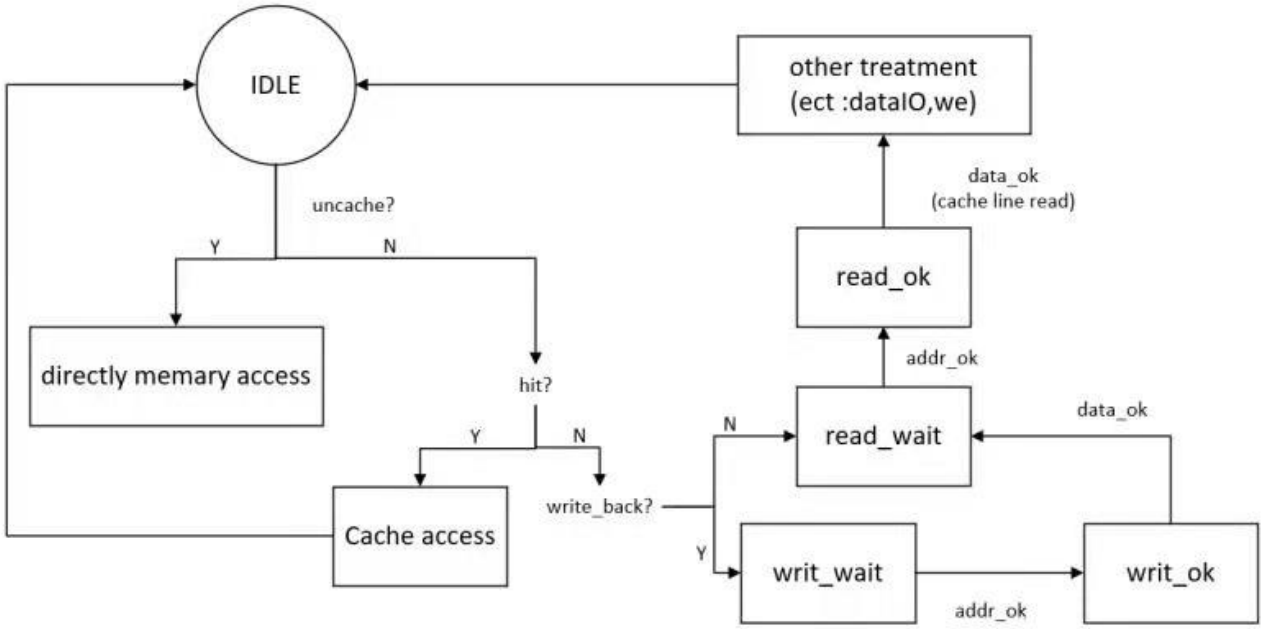
该锁存器需要锁存必要信号外，还需要在流水线暂停时为执行阶段提供多周期指令执行时状态机需要的信号。

2.3.7 MEM 访存阶段



该部分由于挂载 dcache，所以改动较大，且两个 CPU 由于 dcache 部分的思路不同所以区别也较大

首先是 GadgetMIPS，如前文所提到的，我们采取过《CPU 设计实战中》中的流水线化 CPU，但是由于线路过于庞大和缺乏经验，我们最终采取了二路组相联的 SRAM 版本的 dcache，而 hhhhMIPS，则是完整的流水线设计。



笔者在 GadgetMIPS 的 dcache 中采用了行写回、写优先的架构，其中在组相联中采取了随机数发生器用于替换（详见 GadgetMIPS 中的 `LSFR.v` 文件），此处笔者则列出 dcache 部分的状态机转换：

```
always @(*) begin
    if(rst == `RstEnable)begin
        stallreq <= 1'b0;
    end else begin
        case(state)
            `IDLE: begin
                if(mem_ce) begin
                    stallreq <= ~hit_total;
                end else begin
                    stallreq <= `False_v;
                end
            end
            `READWAIT: begin
                stallreq <= `True_v;
            end
            `READOK: begin
                stallreq <= `True_v;
            end
            `WRITEWAIT: begin
                stallreq <= `True_v;
            end
            `WRITEOK: begin
                stallreq <= `True_v;
            end
            `UNCACHEWAIT: begin
                stallreq <= `True_v;
            end
            `UNCACHEOK: begin
                if (!mem_data_ok) begin
                    // 数据握手不成功，原地等待
                    stallreq <= `True_v;
                end else begin
                    // 数据握手成功，立刻撤销流水线暂停
                    // 转入空闲阶段
                    stallreq <= `False_v;
                end
            end
            default: begin
```

```
                stallreq <= `False_v;
            end
        endcase
    end
end
```

此处值得注意的点在于：龙芯支持的指令一般位于 0xbfc00000 位置处开始执行（即 MIPS32 中的 kseg 段），这部分是属于 uncache 的部分，不允许外设强行在这个地方缓存数据，因此这里额外需要一个判断：当出现 uncache 时，单独走一个内存访问的流程，而不是在原来的基础上修改。

2.3.8 dcache 的暂停

由于此阶段设计外设访问，事物等待和握手，所以 mem 中 cache 是否命中问题是 CPU 中最容易拖慢时钟的部分，所以我们需要根据不同的情况做出不同的暂停，分析即可知道以下情况：

- 读且命中：这种情况可以直接从 cache 内获取数据，且当拍即可完成，自然不需要暂停；
- 写且命中：由于 cache 设计采用写回的模式，因此可以直接对 cache 内部的数据进行更改，并将其添加一个脏数据标记，而并非直接访问内存、消耗宝贵的时间。此设计结构可以保证写命中同样也可以一周期内完成，而写操作本身并不需要返回结果，完全可以令 cache 自己运行，并使其他模块进行下一条指令的准备工作；
- 最后一组数据返回：这种情形表示访问已经完全结束，不再需要进行其他的等待；相应地，给予的暂停信号应当得到撤销；
- 其他：数据还在等待传输的流程完成，需要使暂停延续下去。

2.3.9 写回

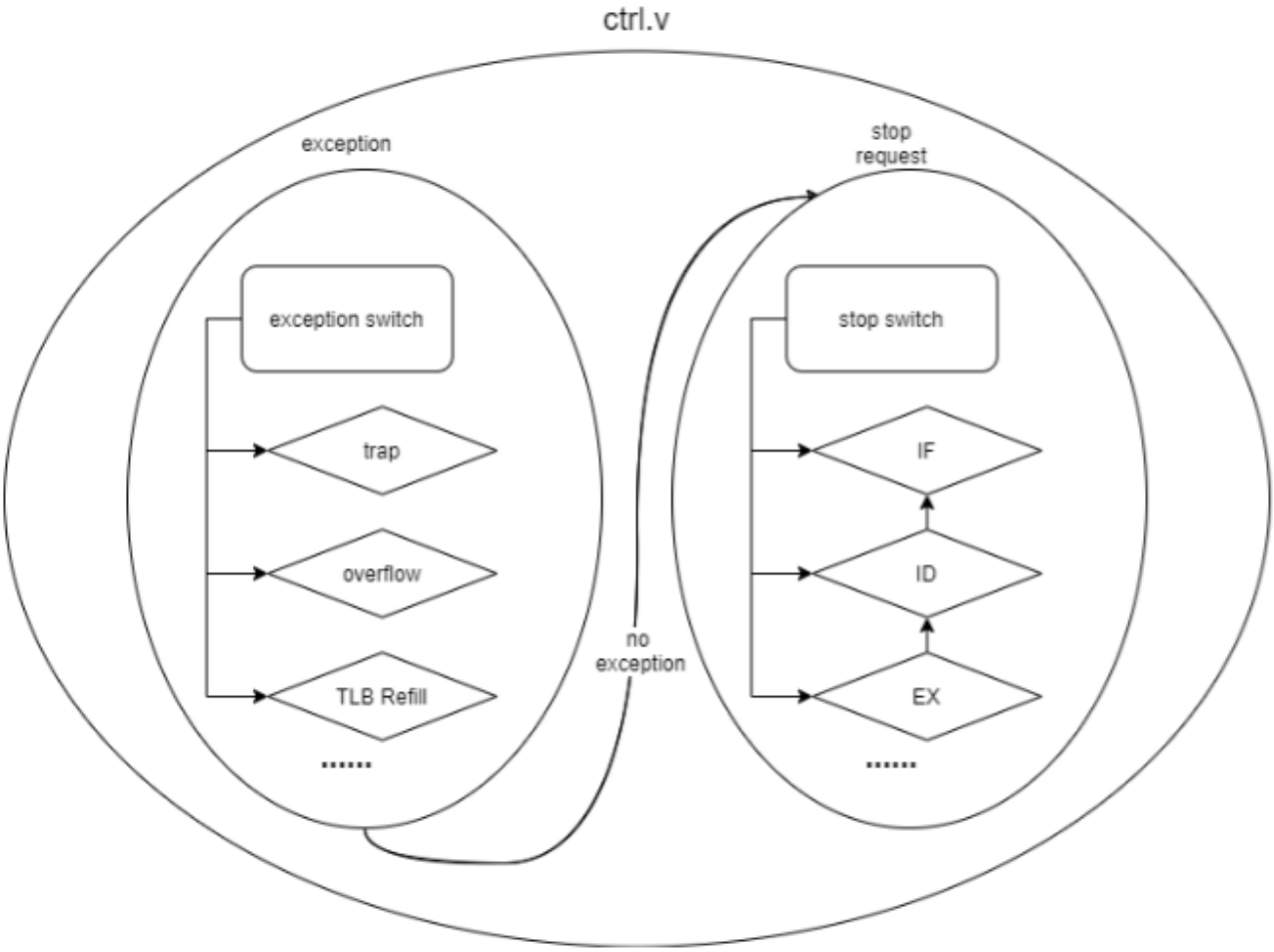
写回部分并不复杂详细的代码可以在源码中查看，其主要问题在于细节的注意，在此强调我们在实现此部分所遇到的问题：

写回寄存器时，之后的指令执行读取操作可能会发生写后读相关，因此指令在向寄存器堆读取数据之前，会同时观察这些模块，观察它们正在写入的寄存器编号。一旦确认与目前读的位置一致，读取过程将会选择写回等模块所采用的数据，而非从寄存器读取的数据，作为该指令最终读取的结果。

2.3.10 总控制器

总控制器主要负责两方面的控制：

1. 根据异常更改 PC，取指过程入口转向对应的异常处理程序
2. 接受暂停信号，仅允许发起信号模块之后余下的流程完成，其余部分暂停实施



异常处理程序是实现高级功能中最为基础的部分。ctrl.v 不会自我保存异常地址入口，而是通过外部模块告知实现，这种设计是为了便于兼容及日后开发。

除此之外：异常处理流程中保存了 0xbfc00380 这个位置，这是因为很多 MIPS32 设计都采取这个位置作为默认的处理入口。因此，我们也保留了这个默认的异常程序地址。

一切正常时，ctrl.v 会继续检查暂停判断。不同的模块有可能在同一时间发起暂停请求，接受以后暂停该模块的运行。与异常处理不同的一点是：暂停不仅会影响当前模块，还会影响流水线之前的那一些模块（堵塞了后续过程，前面的指令不应该实施）。最终，控制器将会暂停最排后模块之前的所有执行步骤，直至得到响应、暂停请求得以终止。

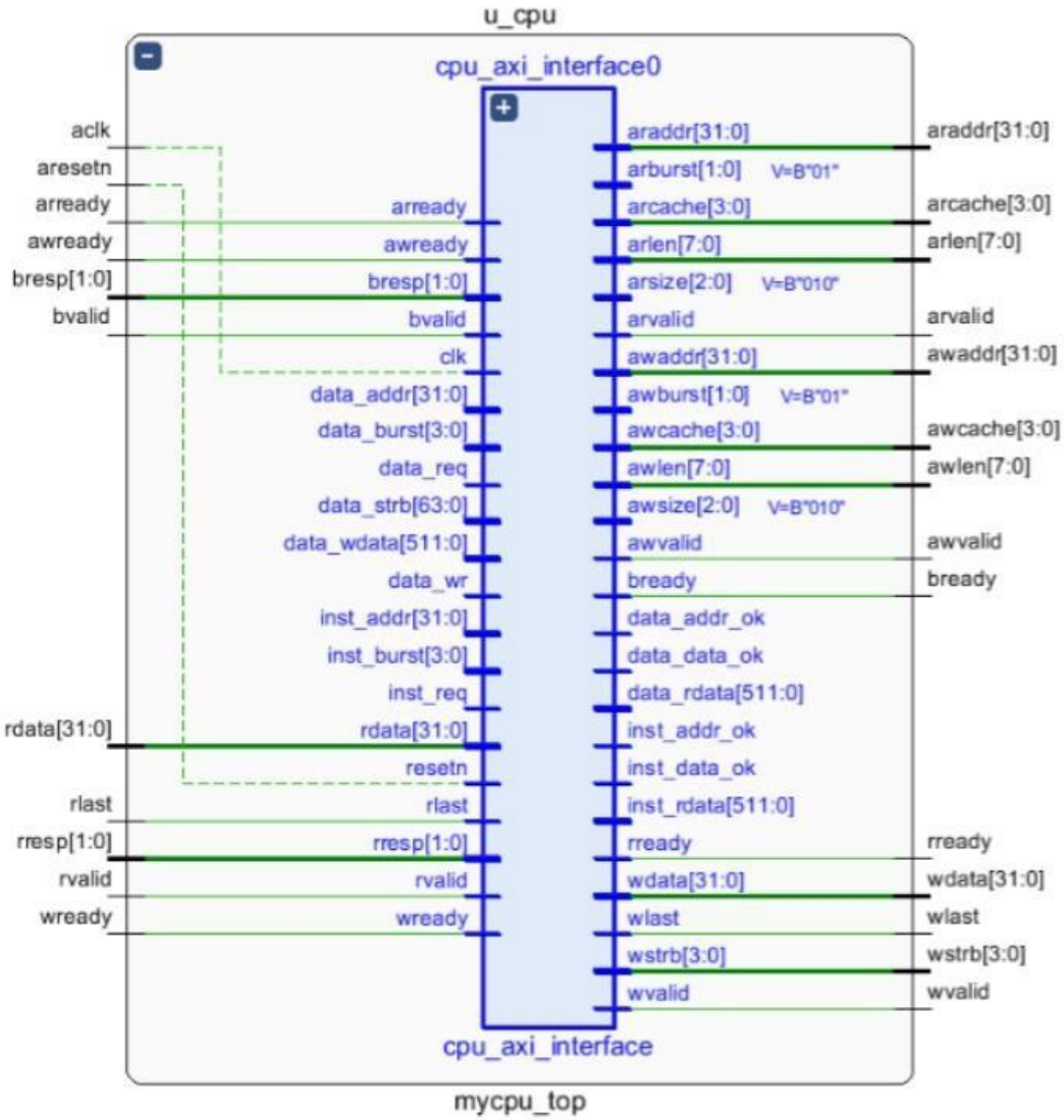
```
always @ (*) begin
        stall <= 6'b000000;
        flush <= 1'b0;
        if(rst == `RstEnable) begin
            epc_o <= `ZeroWord;
        end else if(exception_occured_i) begin
            // 有异常
            flush <= 1'b1;
            case (exc_code_i)
                // 根据异常类型判断 pc 要写入的值
                5'h10: begin
                    // ERET 调用
                    epc_o <= cp0_epc_i;
                end
                default: begin
                    // 其他异常统一入口
```

```
                    epc_o <= 32'hBFC00380;
                end
            endcase
        end else if(stallreq_from_mem == `Stop) begin
            stall <= 6'b011111;
        end else if(stallreq_from_ex == `Stop) begin
            stall <= 6'b001111;
        end else if(stallreq_from_id == `Stop) begin
            stall <= 6'b000111;
        end else if(stallreq_from_if == `Stop) begin
            stall <= 6'b000111; // ID 跟着暂停，应该可以解决跳转的问题
            // TODO 与 MEM 竞争被仲裁等待时，ID 是否应该额外暂停？
        end else begin
            stall <= 6'b000000;
        end
    end
```

3 总线设计

在设计之初我们设想过纯 AXI 总线的方案，但是由于接口的细节以及 verilog 软件综合和布线 后对于代码有不同程度的优化，所以导致逻辑正确而综合后出现错误的情况。

所以我们最终还是将总线设计为了：对内为 SRAM 总线，对外为 AXI 总线，并通过桥将信号连接，详细的总线文件可在 GadgetMIPS 中的 cpu_axi_interface.v 文件中看到。



3.1 类 SRAM 总线部分

此类 SRAM 总线与主流的 32 位设计为了对内 512 数据位宽，这是为了兼容 cache 单行 16 个字的设计。并且 `cpu_axi_interface.v` 兼顾一个数据缓存的功能，将一行数据单独存起来用 于保存，功效等同于 32 位类 SRAM 的理念设计。

设计的类 SRAM 的信号规定如下：

信号	宽度	方向	功能	对应
req	1	master->slave	发起读/写请求。高电平有效	inst_req data_req
wr	1	master->slave	判断为读操作或者是写操作。高电平为写	inst_wr data_wr
size	4	master->slave	传输数据大小，以32位长的字为单位	inst_burst data_burst
addr	32	master->slave	访问的32位地址	inst_addr data_addr
wstrb	64	master->slave	写使能信号，每位高电平代表对应字节写有效	inst_strb data_strb
wdata	512	master->slave	写出的数据	wdata
addr_ok	1	slave->master	响应：当前数据对应地址已经接受	inst_addr_ok data_addr_ok
data_ok	1	slave->master	响应：当前数据已全部收/发完毕	inst_data_ok data_data_ok
rdata	512	slave->master	返回的数据	inst_rdata data_rdata

3.2 AXI 总线部分

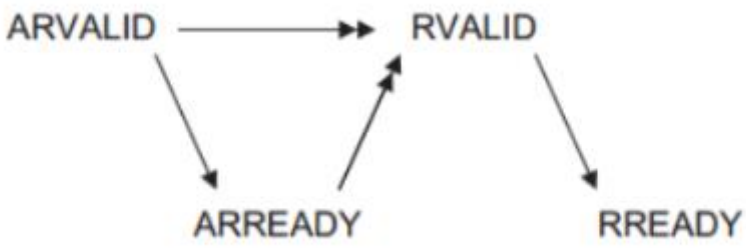
GadgetMIPS 设计的 AXI 总线参照于 `[AMBA AXI Protocol]` 。此协议基于突发传输（突发传传输能明显的提高效率），并且此 AXI 协议允许：

1) 允许在实际数据传输之前发送地址信息；
2) 支持多个读写交替（outstanding）传输；
3) 支持乱序(out-of-order)传输；

3.2.1 事物握手

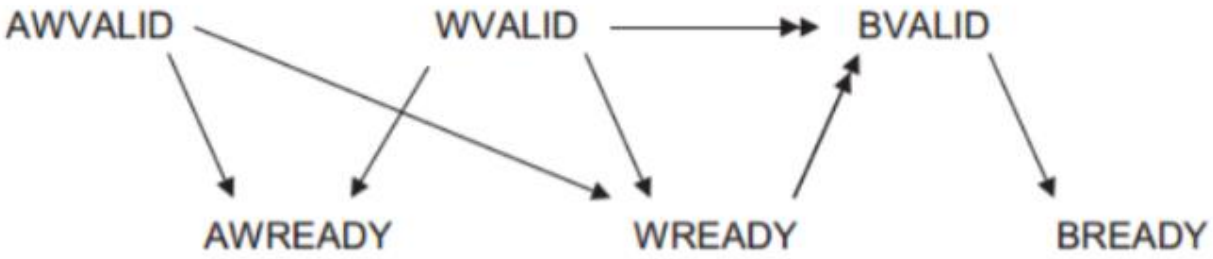
类 SRAM 总线具有握手：只有当请求（req）有效且地址接收响应（addr_ok）有效才表明地址交互完成（称为一次握手）；在这之后只有数据传输响应（data_ok）有效才表明事务交互完成。

AXI 原理与之类似：以读请求握手为例为例：当请求（如 arvalid）有效且地址有效（如 arready）完成一次握手；在这之后，双方开始准备下一个握手：从方发送数据，在发送的同时伴随置起读有效的发送信号（如 rvalid）；主方则在准备妥当后置起读准备的信号（如 rready），只有两者同时有效时，双方才能确认这个数据得到了接收。



写握手状态转换同理，但信号更为复杂：写结束后还要进行一次写确认握手（bvalid 与 bready），同样只有双方均置为有效时，才认定这个事务得到了结束。

基于以上理论我们可以看出，AXI 最大的特点就是一切交互都必须通过双方完整握手来实现，虽然过程描述繁琐复杂，但由于握手和握手之间没有必然的依赖联系，实际拆分相当容易。



3.2.2 乱序实现

GadgetMIPS 由于是采用的五级流水线，所以具有流水线暂停及清空的的特性，基于此特性发生后靠前指令需继续执行而之后的指令将会舍弃。

笔者此处需要强调 GadgetMIPS 并非绝对的乱序传输，为了避免出现写后读相关，同地址的情形下，前条指令的访存过程必须在后条指令之前执行完毕。正因上述情况 GadgetMIPS 设计为了访存优先、写优先，即访存操作、写操作必须严格在取值操作、读操作之前完成。

3.2.3 变长 burst 传输

GadgetMIPS 的一般采用两种长度访问：

1) 对于 uncache 字段的访问或是单独的请求，进行单个数据的访问（burst = 4'b0000）；
2) 对于 cache 更新一整行的操作，进行行长度即 16 个数据的访问（burst = 4'b1111）。
每个数据规定长恒为一个字（size = 4'b0011）。

实际上， cpu_axi_interface.v 支持 1 至16 个字的任意长度突发。在原先通过数据缓存存储数据后，总线桥将会启用一个计数器保存 burst 要求的传输数目；向外设发送数据时，计数器将逐步更新，确保所有的（且仅限）要求的数据被发送成功。

此处为一个读操作样例：

```
2'b01: begin
    // 写 AR 通道进行 AXI 地址握手
    if (arready && arvalid) begin
        // AR 握手成功
        read_counter <= (4'b1111 - arlen[3:0]); // 清零 counter
        // 如果 arlen 是 burst 传输，则低位是 f，减去后 counter 正好为 0
        // 如果不是 burst 传输，保证最后一次传输一定写在 31:0 处
        arvalid <= 1'b0; // 撤销握手信号
        read_status <= 2'b10;
        rready <= 1'b1; // 准备接收数据
    end else begin
        arvalid <= 1'b1; // 保持 AR 握手
        read_status <= 2'b01;
        rready <= 1'b0;
    end
end

2'b10: begin
    // 等待 R 通道的数据握手
    if (rready && rvalid) begin
        // 本次握手成功
        read_result[read_counter] <= rdata;
        read_counter <= read_counter + 1;

        // 最后一个则结束传输
        if (rlast) begin
            // 这里设置读结束，下一个周期应该关闭读状态机使能
            // 从而中断状态机的执行，否则就原地等待
            read_if_or_mem[0] <= 1'b1; // 表示读结束
            rready <= 1'b0;
        end
    end
end
```

3.2.4 AXI 部分信号

此部分并非全部 AXI 信号，仅为此项目中定义的

```
//inst sram-like
input wire inst_req ,
input wire[3:0] inst_burst , // 0000 -> 1 word, 1111 -> 16 words
input wire[31:0] inst_addr ,
output reg [511:0] inst_rdata ,
output wire inst_addr_ok ,
output reg inst_data_ok ,
//data sram-like
input wire data_req ,
```

```
10. input wire[3:0] data_burst , // 0000 -> 1 word, 1111 -> 16 words
11. input wire data_wr ,
12. input wire[63:0] data_strb ,
13. input wire[31:0] data_addr ,
14. input wire[511:0] data_wdata ,
15. output reg [511:0] data_rdata ,
16. output wire data_addr_ok ,
17. output reg data_data_ok ,
18. //axi
19. //ar
20. output wire[31:0] araddr ,
21. output wire[7 :0] arlen ,
22. output wire[2 :0] arsize ,
23. output wire[1 :0] arburst ,
24. output wire[3 :0] arcache ,
25. output reg arvalid ,
26. input wire arready ,
27. //r
28. input wire[31:0] rdata ,
29. input wire[1 :0] rresp ,
30. input wire rlast ,
31. input wire rvalid ,
32. output reg rready ,
33. //aw
34. output wire[31:0] awaddr ,
35. output wire[7 :0] awlen ,
36. output wire[2 :0] awsize ,
37. output wire[1 :0] awburst ,
38. output wire[3 :0] awcache ,
39. output reg awvalid ,
40. input wire awready ,
41. //w
42. output wire[31:0] wdata ,
43. output wire[3 :0] wstrb ,
44. output wire wlast ,
45. output reg wvalid ,
46. input wire wready ,
47. //b
48. input wire[1 :0] bresp ,
49. input wire bvalid ,
50. output reg bready
```

3.2.5 内部状态转换

GadgetMIPS 转换遵循几个原则：

1) 多个请求冲突时，写优先、访存阶段优先；
2) 内部实现数据缓存，暂存未经发送的数据；
3) 默认以字为传输单位，但经由 AXI 部分传输的字数量可以更改；

3.2.6 读写优先级处理

基于此部分我们需要明确其中的几个操作发生的场合：

1) 写操作仅有访存阶段才可能发生，仅需写请求发生即可开始运行；
2) 读操作则可能在取指阶段与访存阶段发生，需要写请求开始初步运行，且优先判断访存阶段的请求；
3) 在这之后，读状态机还需等待写操作完成，这个操作才能继续进行，否则读请求将会被阻塞；

基于以上场景，我们提出以下握手方式：

```
// SRAM 握手
always @ (posedge clk) begin
    if (!resetn) begin
        ...
    end else begin
        // 正常逻辑
        if (!write_en) begin
            // 当前没有写操作
            if (data_wr) begin
                // 如果上一个是写
                data_data_ok <= 1'b0; // 清除数据握手
            end
            if (data_req && data_wr) begin
                ...
            end else begin
                // 不写则保证写状态机关闭
                write_en <= 1'b0;
            end
        end else begin
            // 当前有写操作
            if (write_done) begin
                // 写完了
                data_data_ok <= 1'b1; // 进行数据握手
                write_en <= 1'b0; // 关闭写状态机
            end else begin
                // 还在写
                data_data_ok <= 1'b0; // 不握手
                write_en <= 1'b1; // 保持写状态机打开
            end
        end

        if (!read_en) begin
            // 当前没有读操作
            if (!data_wr) begin
                // 如果上一个是读
                data_data_ok <= 1'b0; // 清除数据握手
            end
            inst_data_ok <= 1'b0; // 清除数据握手
            if (data_req && !data_wr) begin
                // data 要读
                // 记录读信息
                ...

                read_en <= 1'b1; // 启动读状态机
            end else if (inst_req) begin
                // inst 要读
                ...

                read_en <= 1'b1; // 启动读状态机
            end else begin
                // 不写则保证读状态机关闭
                read_en <= 1'b0;
            end
        end else begin
            // 当前有读操作
            if (read_if_or_mem[1]) begin
                // 读 data
                if (read_if_or_mem[0]) begin
                    // 读完了
                    ...
                    data_data_ok <= 1'b1; // 进行数据握手
                    read_en <= 1'b0; // 关闭读状态机
                end else begin
```

```
                    // 还在写
                    data_data_ok <= 1'b0; // 不握手
                    read_en <= 1'b1; // 保持读状态机打开
                end
            end else begin
                // 读 inst
                if (read_if_or_mem[0]) begin
                    // 读完了
                    ...

                    inst_data_ok <= 1'b1; // 进行数据握手
                    read_en <= 1'b0; // 关闭读状态机
                end else begin
                    // 还在写
                    inst_data_ok <= 1'b0; // 不握手
                    read_en <= 1'b1; // 保持读状态机打开
                end
            end
        end
    end
end
```

3.2.7 数据缓存与选择

类 SRAM 的 addr_ok 信号一般用来确认地址传输完毕，但也会用来传输其他相关数据。GadgetMIPS 在内部设置了多组 16 个字长的数据缓存块，在接收到请求、接受地址的同时开 始接收数据。缓存部分同样具有在两个收发端之间进行数据选择的功能。

只有当数据被全部缓存时，addr_ok 有效信号才会传回给 CPU；同样，当数据被全部交换完毕后，总线桥才会 CPU 将 data_ok 设为有效。

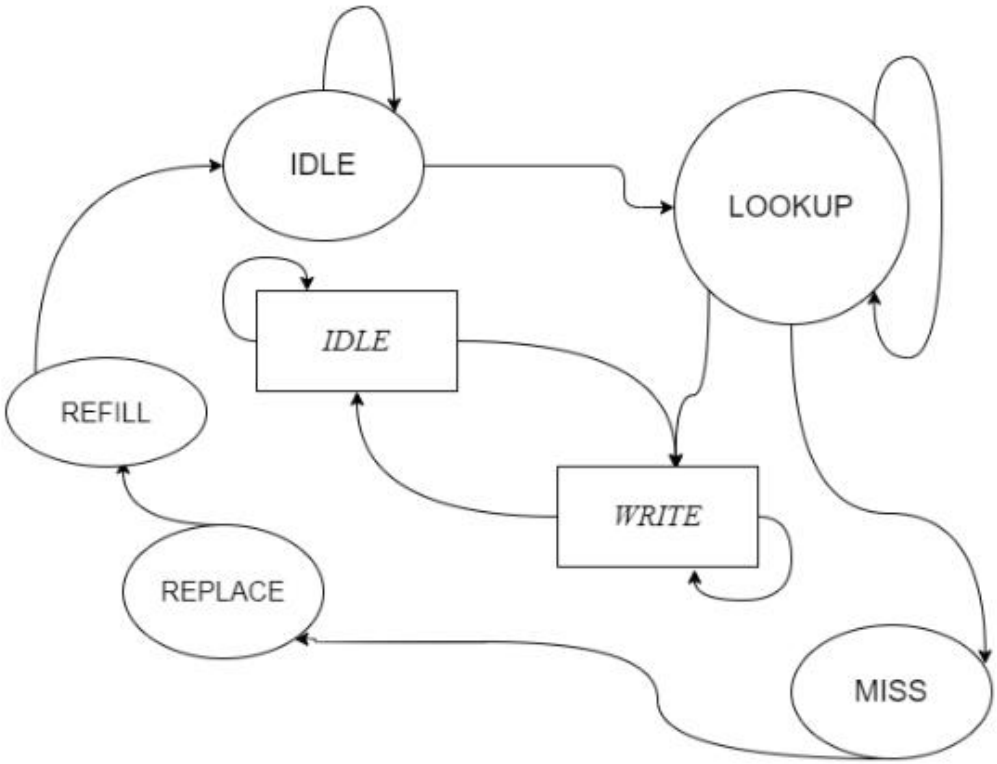
```
reg[31:0] read_addr;
reg[3:0] read_burst;
reg[31:0] read_result[15:0];
// [1]: 0 inst, 1 data; [0]: 0 reading, 1 done.
reg[1:0] read_if_or_mem;
reg read_en; // 使能读状态机，为 1 时读状态机开始启动

reg[31:0] write_addr;
reg[3:0] write_burst;
reg[31:0] write_data[15:0];
reg[3:0] write_strb[15:0];
reg write_done; // 0 writing, 1 done
reg write_en; // 使能写状态机，为 1 时写状态机开始启动

...

if (read_if_or_mem[1]) begin
    // 读 data
    if (read_if_or_mem[0]) begin
        // 读完了
        data_rdata[511:480] <= read_result[0];
        data_rdata[479:448] <= read_result[1];
        data_rdata[447:416] <= read_result[2];
        data_rdata[415:384] <= read_result[3];
    ...
    end else begin
    // 读 inst
    if (read_if_or_mem[0]) begin
        // 读完了
```

```
30.         inst_rdata[511:480] <= read_result[0];
31.         inst_rdata[479:448] <= read_result[1];
32.         inst_rdata[447:416] <= read_result[2];
33.         inst_rdata[415:384] <= read_result[3];
```

4 针对 cache 的流水线化设计

由于笔者并未参与此处的工作，所以此处的报告更多来源于参考代码部分理解，如有纰漏，还请指出，在此写出此部分权当提醒与纪念。

4.1 流水线化 cache 的状态机转换



主状态机各状态下的状态转移

1) IDLE→IDLE：这一拍，流水线没有新的 Cache 访问请求，或者有请求，但因该请求与 Hit Write 冲突而无法被 Cache 接收；
2) IDLE→LOOKUP：这一拍，Cache 接收了流水线发来的一个新的 Cache 访问请求 (必定与Hit Write 无冲突)；
3) LOOKUP→IDLE：当前处理的操作是 Cache 命中的，且这拍流水线没有 新的 Cache 访问请求，或者有请求但因该请求与 Hit Write 冲突而无法被 Cache 接收；

4) **LOOKUP→LOOKUP**：当前处理的操作是 Cache 命中的，且这一拍 Cache 接收了流水线发来的一个新的 Cache 访问请求，且与 Hit Write 无冲突；
5) **LOOKUP→MISS**：当前处理的操作是 Cache 缺失的；
6) **MISS→MISS**：AXI 总线提口模块反馈回来的 id 为 0；
7) **MISS→REPLACE**：16 字节写缓存为空，AXI 总线接口模块反馈回来的 wr_rdy 为 1(表示AXI 总线内部 可以接收 wr_req)。当看到 wr_rdy 为 1 时，会对 Cache 发起替换的读请求，并 转到 REPLACE 状态
8) **REPLACE→REPLACE**：AXI 总线接口模块反馈回来的 rd_rdy 为 0。刚进入 REPLACE 的第一拍，会得到被替换的 cache 行数据，并发起 wr_req 至 AXI 总线接口（由于 wr_rdy 为 1，故 wr_req 一定会被接收）；同时，对 AXI 总线发起缺失 Cache 的读请求；
9) **REPLACE→REFILL**：AXI 总线接口模块反馈回来的 rd_rdy 为 1，表示对 AXI 总线 发起的缺失 Cache 的读请求将被接收；
10) **REFILL→REFILL**：缺失 Cache 行的最后一个 32 位数据（即 ret_valid==1 && ret_last==1 ）尚 未 返 回；
11) **REFILL→IDLE**：缺失 Cache 行的最后一个 32 位数据从 AXI 总线接口模块返回。

WriteBuffer 状态机里的各状态间的转换条件说明如下：

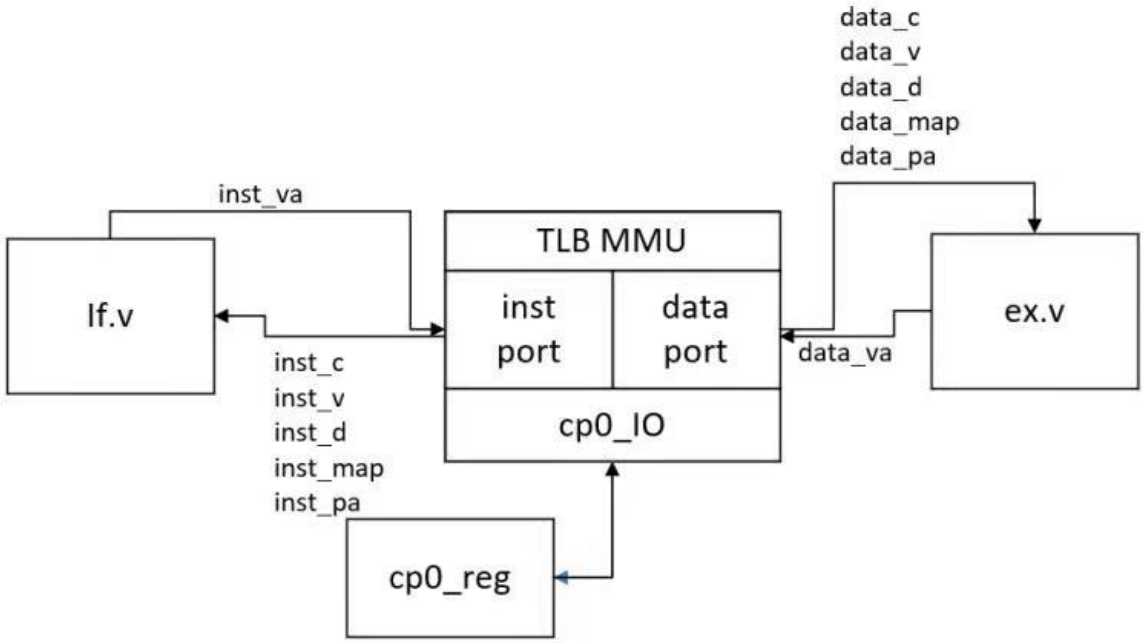
1) **IDLE→IDLE**：这一拍，Write Buffer 没有待写的数据，并且主状态机没有新的 Hit Write；
2) **IDLE→WRITE**：这一拍，Write Buffer 没有待写的数据，并且主状态机发现新的 Hit Write（主状态机处于 LOOKUP 状态且发现 Store 操作命中 Cache）；
3) **WRITE→WRITE**：这一拍，Write Buffer 有待写的数据，并且主状态机发现新的 Hit Write；
4) **WRITE→IDLE**：这一拍，Write Buffer 有待写的数据，并且主状态机没有新的 Hit Write。

5 系统设计

5.1 TLB 设计

首先，MIPS32 是基于页表的页式存储管理，通过虚实地址切换的过程实现。而在地址转换中以 4KB 大小为单位，称为页。页内的低 12 位地址只是从虚拟地址简单地传递到物理地址。转换表中每一项含有一个页的虚拟地址(VPN，即虚拟页号)和一个物理页地址(PFN，代表页 帧号)。当程序给出一个虚拟地址时，该地址和 TLB 中的每个 VPN 做比较，如果和某项匹配 就给出相应的 PFN。

其次 TLB 是一种内容寻址的存储器，是按照内容来选择某一项。其中每一项都有内建的比较器，这就直接导致 TLB 的复杂度和性能扩展性很差，所以典型的 TLB 只有 16 到 64 项，而我们此处也采取了 32 项。



5.2 查询流程

伪代码描述如下：

```
1.  found ← 0
2.  for i in 0...TLBEntries-1
3.      if ( (TLB[i].VPN2 and not (TLB[i].Mask )) = (va 31..13 and not (TLB[i].Mas
k ))) and (TLB[i].G or (TLB[i].ASID = EntryHi.ASID )) then
4.          if va12 = 0 then
5.              pfn ← TLB[i].PFN0
6.              v ← TLB[i].V0
7.              c ← TLB[i].C0
8.              d ← TLB[i].D0
9.          else
10.         pfn ← TLB[i].PFN1
11.             v ← TLB[i].V1
```

```
            c ← TLB[i].C1
            d ← TLB[i].D1
        endif
        if v = 0 then
            SignalException(TLBInvalid, reftype)
        endif
        if (d = 0) and (reftype = store) then
                SignalException(TLBModified)
        endif
        # pfn 19..0 corresponds to pa 31..12
        pa ← pfn19..0 || va11..0
        found ← 1
        break
    endif
endfor

if found = 0 then
    SignalException(TLBMiss, reftype)
endif
```

5.3 TLB 架构

考虑到取指以及访存阶段同时需要访问内存，因此 TLB 模块为两个部分各自单独设计了端口：输入虚拟地址，返回物理地址；

同时，为了减少相关信号的依赖，将 C，D，V，TLB 命中等信号位传出用于判断异常；将是否映射传出用于判断是否应采用查询的物理地址。

另外，为支持异常及 TLB 指令，TLB 需要与 CP0 寄存器组进行交互，及时更新数据

```
module tlb_mmu(
input wire clk,
input wire rst,
//////////////////////////////////////////////////////////////
// tlb_op:
// 3: tlbP
// 2: tlbr
// 1: tlbwi
// 0: tlbwr
//////////////////////////////////////////////////////////////
input wire[3:0] tlb_op,
// icache 读 tlb 接口: if
input wire[`RegBus] inst_virtual_pc_i,
output reg[`RegBus] inst_physical_pc_o,
output reg inst_tlb_hit,
output reg[2:0] inst_c,
output reg inst_d,
output reg inst_v,
output reg inst_mapped,
// mem
input wire[`RegBus] data_virtual_pc_i,
output reg[`RegBus] data_physical_pc_o,
output reg data_tlb_hit,
output reg[2:0] data_c,
output reg data_d,
output reg data_v,
output reg data_mapped,
// cp0 output
input wire[`RegBus] index_i,
input wire[`RegBus] random_i,
input wire[`RegBus] entryLo0_i,
input wire[`RegBus] entryLo1_i,
input wire[`RegBus] entryHi_i,
```

```
input wire[`RegBus] pageMask_i,
// cp0 input
output reg[`RegBus] index_o,
output reg[`RegBus] entryLo0_o,
output reg[`RegBus] entryLo1_o,
output reg[`RegBus] entryHi_o,
output reg[`RegBus] pageMask_o
);
```

取指/访存阶段的输出根据两个 always 组合逻辑块实现。对于位于 kseg0/1 的部分及寻找失败的情形进行特判，并用默认值填充：

```
// inst 读
always @ (*) begin
    if(rst == `RstEnable) begin
        inst_tlb_hit <= `TLBMiss;
        inst_physical_pc_o <= `ZeroWord;
        inst_c <= 3'b000;
        inst_d <= 1'b0;
        inst_v <= 1'b0;
        inst_mapped <= 1'b0;
    end else begin
        inst_tlb_hit <= `TLBMiss;
        inst_physical_pc_o <= `ZeroWord;
        inst_c <= 3'b000;
        inst_d <= 1'b0;
        inst_v <= 1'b0;
        inst_mapped <= 1'b0;
        if(inst_virtual_pc_i[31:30] == 2'b10) begin
            // 此时地址处于 unmapped 区域
            inst_tlb_hit <= `TLBHit;
            inst_physical_pc_o <= {3'b000, inst_virtual_pc_i[28:0]};
            inst_c <= inst_virtual_pc_i[29] ? 3'b010:3'b011;                //
 'uncached' or 'cacheable noncoherent'
            inst_d <= 1'b0;                                                 //
 default as 'clean'
            inst_v <= 1'b1;                                                 //
 default as 'valid'
            inst_mapped <= 1'b0;
        end else begin
            inst_tlb_hit <= (is_hit_inst ? `TLBHit:`TLBMiss);
            inst_physical_pc_o <= {inst_pfn_result, inst_virtual_pc_i[`VA_OFFS
ET]};
            inst_c <= inst_c_result;
            inst_d <= inst_d_result;
            inst_v <= inst_v_result;
            inst_mapped <= 1'b1;
        end
    end
end

// data 读
always @ (*) begin
    if(rst == `RstEnable) begin
        data_tlb_hit <= `TLBMiss;
        data_physical_pc_o <= `ZeroWord;
        data_c <= 3'b000;
        data_d <= 1'b0;
        data_v <= 1'b0;
        data_mapped <= 1'b0;
    end else begin
        data_tlb_hit <= `TLBMiss;
        data_physical_pc_o <= `ZeroWord;
        data_c <= 3'b000;
        data_d <= 1'b0;
        data_v <= 1'b0;
        data_mapped <= 1'b0;
```

```
        if(data_virtual_pc_i[31:30] == 2'b10) begin
            // 此时地址处于 unmapped 区域
            data_tlb_hit <= `TLBHit;
            data_physical_pc_o <= {3'b000, data_virtual_pc_i[28:0]};
            data_c <= data_virtual_pc_i[29] ? 3'b010:3'b011;                //
 'uncached' or 'cacheable noncoherent'
            data_d <= 1'b0;                                                 //
 default as 'clean'
            data_v <= 1'b1;                                                 //
 default as 'valid'
            data_mapped <= 1'b0;
        end else begin
            data_tlb_hit <= (is_hit_data ? `TLBHit:`TLBMiss);
            data_physical_pc_o <= {data_pfn_result, data_virtual_pc_i[`VA_OFFS
ET]};
            data_c <= data_c_result;
            data_d <= data_d_result;
            data_v <= data_v_result;
            data_mapped <= 1'b1;
        end
    end
end
```

此处我们所遇到的麻烦主要集中于：如何与 32 个表项比较？

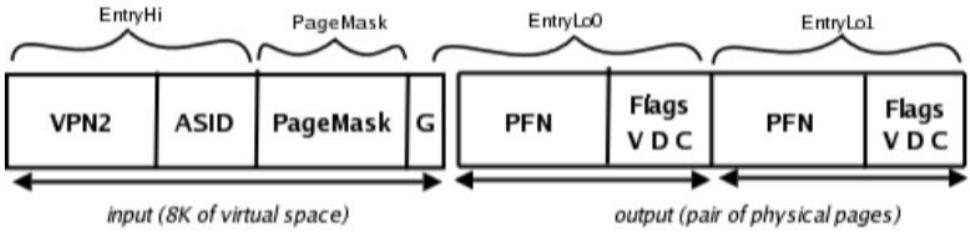
如果我们决定逐一进行比对，这就意味着需要 32 个时钟周期，或者被迫在一个周期内进行大量等待，直到先前比较完成才执行，这就意味着我们会有大量的时间消耗。

所以最好的一个办法是利用逐位或运算，让电路并行完成比较任务，这就意味着效率是多倍的提升。

```
assign hit_inst[0] = tlb_valid[0] & (inst_virtual_pc_i[`VA_VPN2] == tlb_ram[0]
[`TLB_VPN2]) && ((entryHi_i[`EntryHiASID] == tlb_ram[0][`TLB_ASID]) || tlb_ram[0][`TL
B_G]);
assign hit_inst[1] = tlb_valid[1] & (inst_virtual_pc_i[`VA_VPN2] == tlb_ram[1]
[`TLB_VPN2]) && ((entryHi_i[`EntryHiASID] == tlb_ram[1][`TLB_ASID]) || tlb_ram[1][`TL
B_G]);
assign hit_inst[2] = tlb_valid[2] & (inst_virtual_pc_i[`VA_VPN2] == tlb_ram[2]
[`TLB_VPN2]) && ((entryHi_i[`EntryHiASID] == tlb_ram[2][`TLB_ASID]) || tlb_ram[2][`TL
B_G]);
assign hit_inst[3] = tlb_valid[3] & (inst_virtual_pc_i[`VA_VPN2] == tlb_ram[3]
[`TLB_VPN2]) && ((entryHi_i[`EntryHiASID] == tlb_ram[3][`TLB_ASID]) || tlb_ram[3][`TL
B_G]);

wire hit_inst[0:15];
wire is_hit_inst;
assign is_hit_inst = hit_inst[ 0] | hit_inst[ 1] | hit_inst[ 2] | hit_inst[ 3]
 |
                hit_inst[ 4] | hit_inst[ 5] | hit_inst[ 6] | hit_inst[ 7] |
                hit_inst[ 8] | hit_inst[ 9] | hit_inst[10] | hit_inst[11] |
                hit_inst[12] | hit_inst[13] | hit_inst[14] | hit_inst[15];

wire hit_data[0:15];
wire is_hit_data;
assign is_hit_data = hit_data[ 0] | hit_data[ 1] | hit_data[ 2] | hit_data[ 3]
 |
                hit_data[ 4] | hit_data[ 5] | hit_data[ 6] | hit_data[ 7] |
                hit_data[ 8] | hit_data[ 9] | hit_data[10] | hit_data[11] |
                hit_data[12] | hit_data[13] | hit_data[14] | hit_data[15];
```

5.4 TLB 相关表项和寄存器

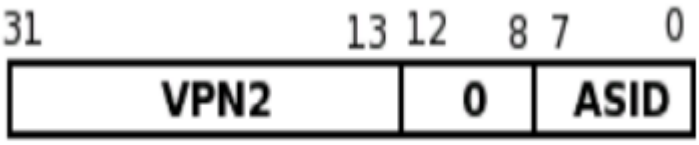
5.4.1 TLB 表项



单个 TLB 表项长度为 90 位，它包括：

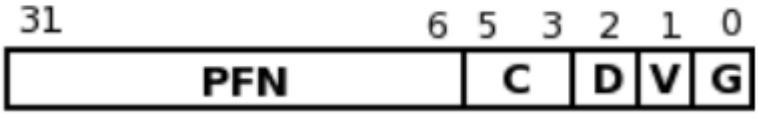
- **VPN2**：19 位，虚拟地址高位。虚拟地址同样有该部分，查询 TLB 过程实际上即为虚拟地址匹配该字段、同时返回对应信息的过程；
- **ASID**：8 位，地址空间标识。当查得的 TLB 表项不为全局可用时（即 G 不为有效），还需要对此字段进行新的比较过程；
- **PageMask**：12 位掩码
- **G**：1 位，全局属性标识符。它表示这个 TLB 表项是否全局可用，若无效，则使用该 TLB 表项时还要额外核对 ASID 字段；
- **EntryLo**：共有两个部分，和 EntryLo0、EntryLo1 进行数据交换。

5.4.2 EntryHi



MIPS32 中 EntryHi 只用到了高位和低位。它作用主要是作为一个进程指示，保存当前进程所用到的虚拟页号以及地址标识符。每当进程切换时，该寄存器的内容都会改变，同时保证不同进程运行时不会有相同的数据，借此保证了安全性。

5.4.3 EntryLo



EntryLo 实际上有两个寄存器，编号为 0 和 1，对应 TLB 低 50 位共 2 个部分。虚拟地址的高 19 位作为 VPN2，低 12 位作为 4KB 的页内偏移原封不动，余下的第 12 位即为选择位，选择 0 或 1 对应的内容。两个寄存器只有遇见特定的 TLB 指令才会发生更改。

数据格式如下：

- **PFN**：26 位，物理页号。TLB 表项中的 PFN 查询后被输出，与虚拟地址余下的偏移量拼接，得到 38 位的物理地址结果 ；
- **C**：2 位，决定 cache 属性。这个属性主要由软件以及系统用来优化，决定该页指向的数据允许缓存（cacheable noncoherent，2'b11）抑或是非缓存（uncache，2'b10）；
- **D**：1 位，脏位。表示该页是否发生写更改；
- **V**：1 位，有效位；
- **G**：1 位，全局标识位。当两个 寄存器写入 TLB 表项时，TLB 表项的 G 即为两个寄存器的 G 与运算的结果。

此处需要做出的提醒在于：

> 虽然在 MIPS32 标准下，拼接后最多允许 256GB 空间的访问，但鉴于对外接口是 32 位的，最终 **GadgetMIPS** 以及 **hhhhMIPS** 仍采用 32 位而非 38 位输出，EntryLo 的 PFN 相应位置被默认置 0。

5.4.4 Index

MIPS32 将其归为一个很“空闲”的寄存器，仅在特殊查询时启用。它的低位保存匹配所需的 TLB 表项号（因我们设为了 32 项，所以采取 5 位）。最高位用于在查询异常时作为指示位。

5.5 TLB 指令支持

MIPS32 规定了共 4 条 TLB 指令：**TLBP**、**TLBR**、**TLBWI**、**TLBWR**：

指令	功能
TLBP	于TLB表中查询EntryHi对应内容
TLBR	将TLB表中Index寄存器对应的表项内容输入EntryLo寄存器
TLBWI	将EntryLo寄存器内容写回TLB表中Index寄存器对应的表项
TLBWR	类似TLBWI，但表项编号来自随机数生成器而不是index寄存器

以上指令主要是针对于 OS 阶段编写的，在测试时不需要考虑该部分。详细代码可参考

tlb_mmi.v 文件。

5.5.1 TLBP、TLBR

TLBP、TLBR 本质上都是读操作且均向 CP0 输出结果，因此选择集成于一个组合逻辑代码块内处理。发现操作即会改变对外输出信号，此后下一拍 CP0 将读取这些信号并更新。

为了方便测试，我们也对 TLBP 增加了新功能，即发现寻找异常，则会将 Index 低位全置为 1，此操作能有效帮助 debug 过程。

```
1.  wire[`TLBWayBus]   hit_hi_match_NotFound;
2.  assign hit_hi_match_NotFound = is_hit_hi_match ? 5'b00000:5'b11111;
3.
4.  wire[`TLBWayBus]   hit_hi_match_switch;
5.  assign hit_hi_match_switch =    (hit_hi_match[0] ? 0:0)        |   (hit_hi_mat
ch[1] ? 1:0)         |
6.                                  (hit_hi_match[2] ? 2:0)        |   (hit_hi_mat
ch[3] ? 3:0)         |
7.                                  (hit_hi_match[4] ? 4:0)        |   (hit_hi_mat
ch[5] ? 5:0)         |
8.                                  (hit_hi_match[6] ? 6:0)        |   (hit_hi_mat
ch[7] ? 7:0)         |
9.                                  (hit_hi_match[8] ? 8:0)        |   (hit_hi_mat
ch[9] ? 9:0)         |
10.                                 (hit_hi_match[10] ? 10:0)        |   (hit_hi_m
atch[11] ? 11:0)         |
11.                                 (hit_hi_match[12] ? 12:0)        |   (hit_hi_m
atch[13] ? 13:0)         |
12.                                 (hit_hi_match[14] ? 14:0)        |   (hit_hi_m
atch[15] ? 15:0)         |
13.                                 hit_hi_match_NotFound;
14. //并行提高效率
15. always @(*) begin
16.     case(tlb_op)
17.         4'b0100: begin
18.             entryHi_o <= {tlb_ram[tlb_index][`TLB_VPN2], 5'b00000, tlb_ram[tlb
_index][`TLB_ASID]};
19.             entryLo0_o <= {6'b000000, tlb_ram[tlb_index][49:25], tlb_ram[tlb_i
ndex][`TLB_G]};
20.             entryLo1_o <= {6'b000000, tlb_ram[tlb_index][24:0],  tlb_ram[tlb_i
ndex][`TLB_G]};
21.             pageMask_o <= {7'b0000000, tlb_ram[tlb_index][`TLB_PageMask], 13'b
0_0000_0000_0000};
22.         end
23.         4'b1000: begin
24.             index_o[31] <= ~is_hit_hi_match;
25.             index_o[30:5] = 26'd0;
26.             index_o[`TLBWayBus] <= hit_hi_match_switch[`TLBWayBus];
27.         end
28.         default: begin
29.         end
30.     endcase
31. end
```

5.5.2 TLBWI、TLBWR

这两个同为写操作的命令的区别仅仅在于使用的下标发生了变化，因而放在了一块处理。当确认为 TLBWR 时，访问所用的下标将从外界的随机数生成器中取得，否则仍选择采用 Index 寄存器的输入。

```
wire[`TLBWayBus] tlb_index;
assign tlb_index = tlb_op[0] ? random_i[`TLBWayBus] : index_i[`TLBWayBus];

// 写入 TLB 操作: tlbwi tlbwr
always @ (posedge clk) begin
    if(tlb_write_en) begin
        tlb_ram[tlb_index][`TLB_VPN2] <= entryHi_i[`EntryHiVPN2];
        tlb_ram[tlb_index][`TLB_ASID] <= entryHi_i[`EntryHiASID];
        tlb_ram[tlb_index][`TLB_PageMask] <= pageMask_i[`PageMask_Mask];
        tlb_ram[tlb_index][`TLB_G] <= entryLo0_i[`EntryLoG] & entryLo1_i[`Entr
yLoG];
        tlb_ram[tlb_index][`TLB_PFN0] <= entryLo0_i[`EntryLoPFN] & (~pageMask_
i[`PageMask_Mask]);
        tlb_ram[tlb_index][`TLB_PFN0_C] <= entryLo0_i[`EntryLoC];
        tlb_ram[tlb_index][`TLB_PFN0_D] <= entryLo0_i[`EntryLoD];
        tlb_ram[tlb_index][`TLB_PFN0_V] <= entryLo0_i[`EntryLoV];
        tlb_ram[tlb_index][`TLB_PFN1] <= entryLo1_i[`EntryLoPFN] & (~pageMask_
i[`PageMask_Mask]);
        tlb_ram[tlb_index][`TLB_PFN1_C] <= entryLo1_i[`EntryLoC];
        tlb_ram[tlb_index][`TLB_PFN1_D] <= entryLo1_i[`EntryLoD];
        tlb_ram[tlb_index][`TLB_PFN1_V] <= entryLo1_i[`EntryLoV];
        tlb_valid[tlb_index] <= 1'b1;
    end
    else begin
    end
end
```

5.6 PMON

基于龙芯杯开源的 gs132 内核为我们搭建了一个小的 soc，该 soc 拥有串口、falsh 芯片和指令数据 ram。通过此 soc 在 FPGA 上生成 bit 流文件即可在线编程。

不过此处也不得不提一下先前遇到的问题，即 cache 中的 cachetag 被意外清空，所以此处代码的修改为

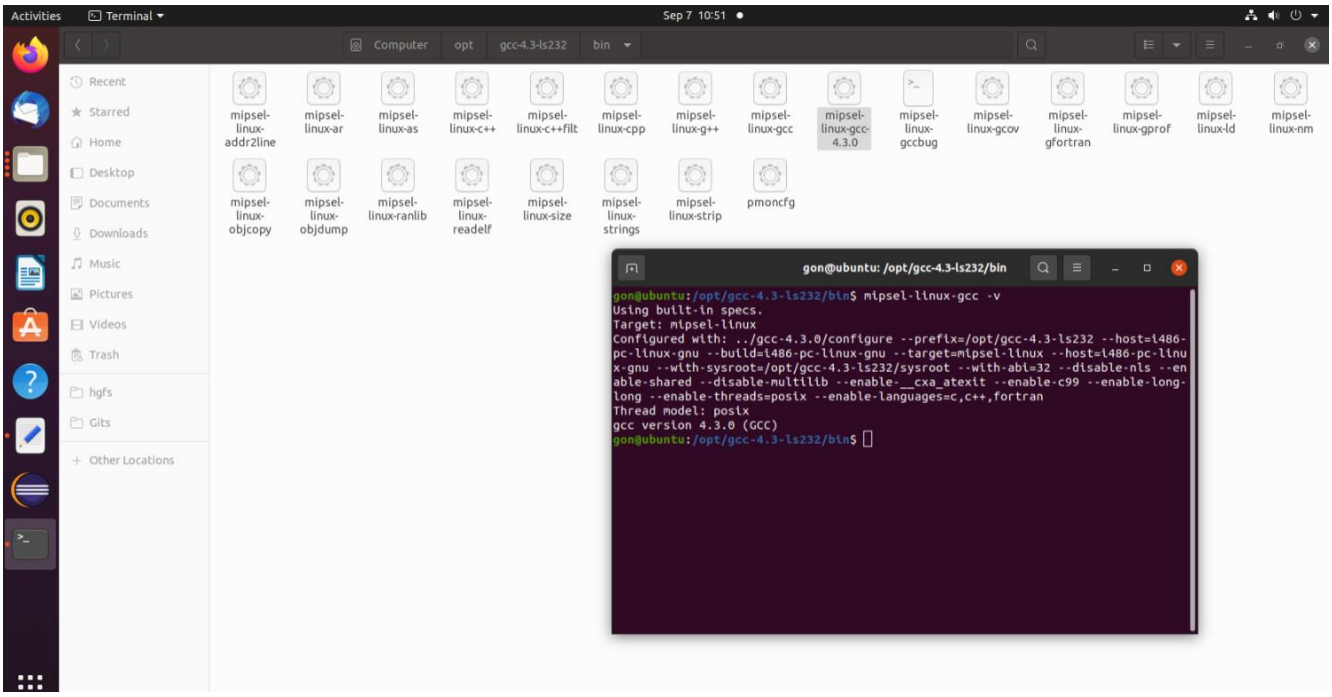
```
dirty_ram dirty0(
        .a(index),
        //.d(hit),错误代码
        .d(~cache_ok),//正确写法
        .clk(clk),
        .we(wr | cache_ok),
        .spo(dirty_out)
);
```

5.6.1 工具链

在此篇章开始前，对于环境我们申明均为 Linux 操作系统。

关于 PMON，我们已经拥有了官方提供的 gzrom.bin 二进制文件，但是由于竞赛时间的限制所以对于 GadgetMIPS 设计的初衷并非是按照着能运行操作系统的 CPU 去设计，基于此原因，我们需要对源代码进行逐一排查进行修改。

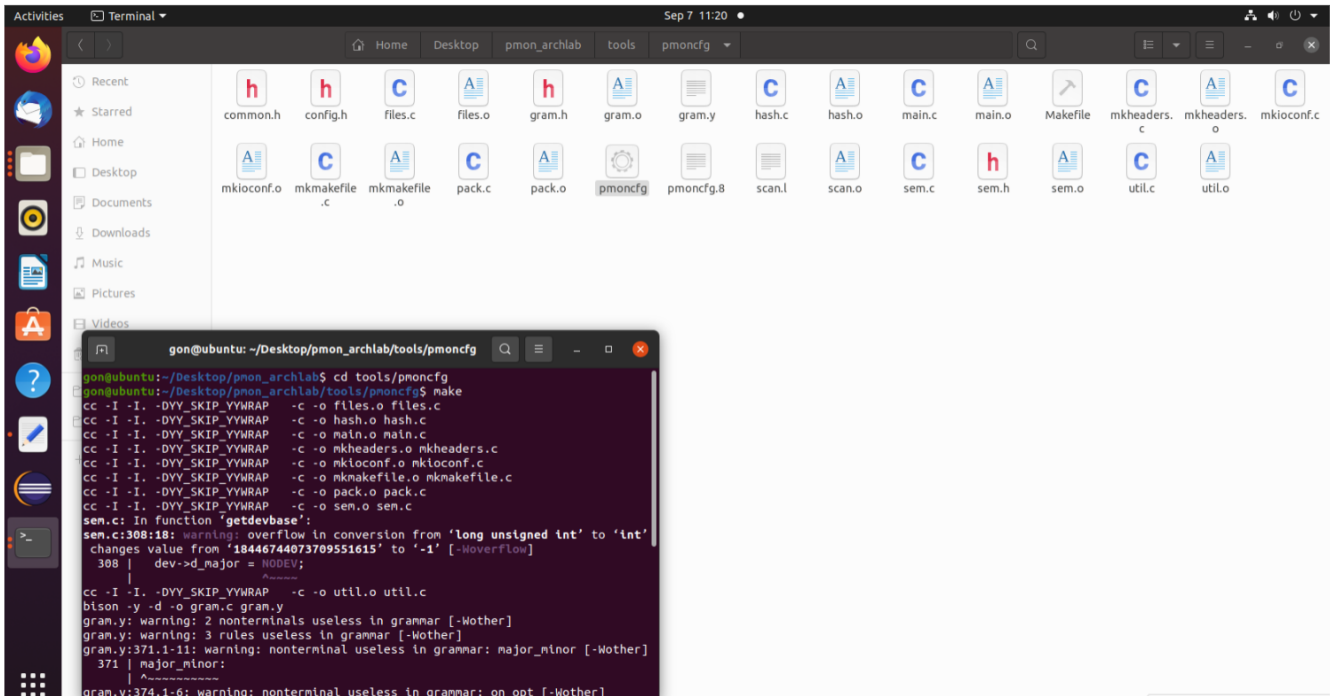
注意：代码的运行都需要其对应的编辑工具和环境，所以我们需要先安装官方要求的工具链 gcc-4.3-ls132，此工具链能够辅助进行对 MIPS32 小端序的程序进行编译支持，而 PMON 的源码由 C 语言编写，所以我们选择 mipsel-linux-gcc 工具链去编译 PMON。（将工具链加入环境变量中能大幅简化其使用的复杂度）



5.6.2 内核编译

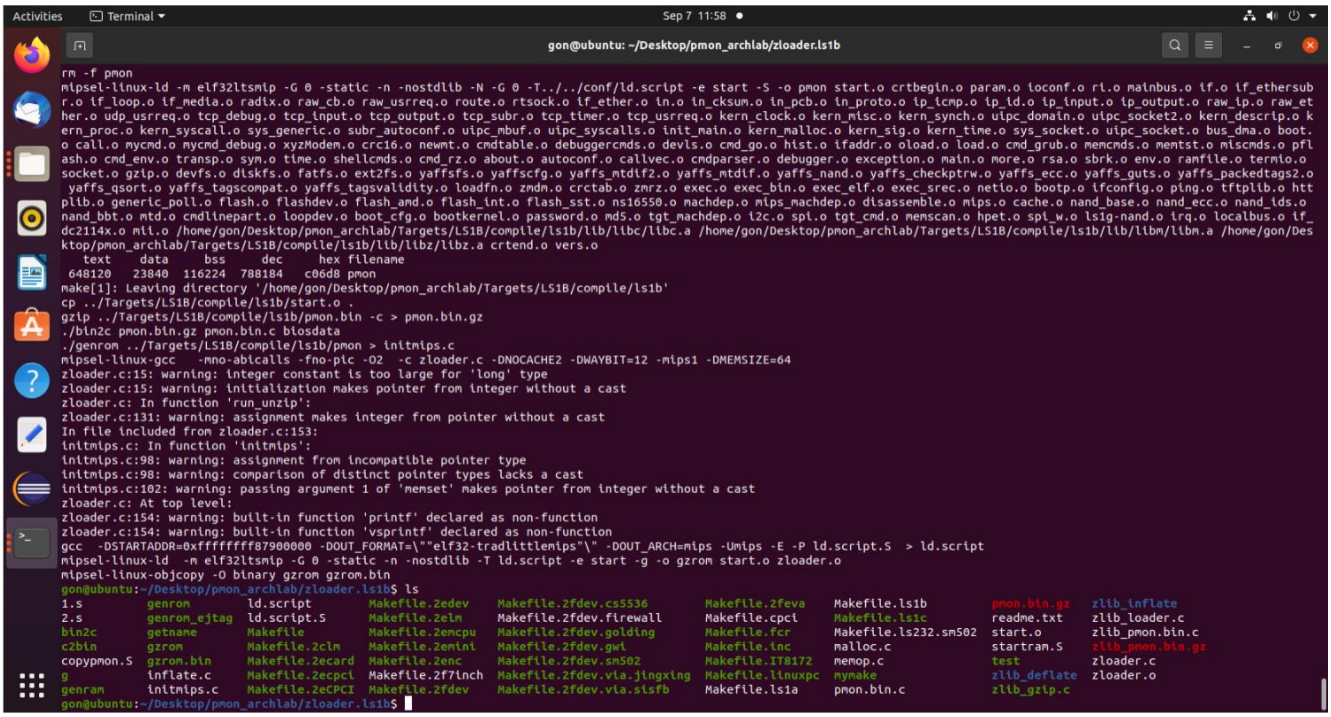
由于 PMON 本身设计较早存在时代限制带来的问题并没有允许用户自己设计 Makefile 文件中的参数，所以我们能做的就是依据其设计的不同环境的各个版本进行选择。

编译不同版本的相关参数需要编辑获得，其中 pmoncfg 由/tools/pmoncfg 生成，生成依赖工具包 bison 和 flex，执行 make 后将文件复制到交叉编译链处即可。



因为版本原因，所以我们需要采用的是 ls1b 版本，此版本需要采用新的工具链，所以我们需要修改 Makefile 文件以此获得正确的编译方式。将目标文件生成于/zloader/Makefile.inc/中，并将 mips-elf 设置为 mips-linux 即可。

回到 ls1b 目录下，运行 make 指令即可，等待编译过程后即可获得我们需要的 gzrom.bin 文件，此文件即可获得支持 elf 的操作，并让 FPGA 读取其运行：



5.6.3 运行命令

由于管脚设计和外设连接比较麻烦，所以龙芯提供了一个 soc 环境作为辅助，配合使用烧写的 flash，所以我们通过 flash 将二进制文件上传即可。

```
$ minicom -s
```

进入 Serial port setup，进行配置

选择 E 行对波特率调整为 57600，随后复位试验箱运行 PMON 即可

```
.....................
 OK,Booting Bios
FREQ
RTC time invalid, reset to epoch.
FREI
DONE
DEVI
ENVI
MAPV
in envinit
nvram=bfc00000
NVRAM is invalid!
NVRAM@bfc00000
STDV
80100000:  memory between 82fff800-83000000  is already been allocated,heap is already above this point
SBDD
DINI
==gpio: gpio status 0
NETI
RTCL
====before configure
in configure
mainbus0 (root)
localbus0 at mainbus0
dmfe0 at localbus0: address 00:98:76:64:32:19
in if attach
loopdev0 at mainbus0out configure
====before init ps/2 kbd
devconfig done.
ifinit done.
domaininit done.
init_proc....
HSTI
SYMI
SBDE

 * PMON2000 Professional *
Configuration [FCR,EL,NET]
Version: PMON2000 2.1 (ls1b) #362: 2017年04月13日星期四09:55:07 CST commit 77efeacd131b8eacede0ba1ec6e311
Supported loaders [srec, elf, bin]
Supported filesystems [mtd, net, fs/yaffs2, fat, fs, disk, socket, tty, ram]
This software may be redistributed under the BSD copyright.
Copyright 2000-2002, Opsycon AB, Sweden.
Copyright 2005, ICT CAS.
CPU GODSON1 @ 99.99 MHz / Bus @ 99.99 MHz
Memory size 128 MB (128 MB Low memory,   0 MB High memory) .
Primary Instruction cache size 4kb (64 line, 1 way)
Primary Data cache size 4kb (64 line, 1 way)

BEV1
BEV2
BEV3
BEV0
BEV in SR set to zero.

NAND DETE
NAND device: Manufacturer ID: 0xec, Chip ID: 0xf1 (Samsung NAND 128MiB 3,3V 8-bit)
NAND_ECC_NONE selected by board driver. This is not recommended !!
Scanning device for bad blocks
Bad eraseblock 191 at 0x017e0000
Bad eraseblock 415 at 0x033e0000
Bad eraseblock 714 at 0x05940000
Bad eraseblock 898 at 0x07040000
NANDFlash info:
erasesize       131072 B
writesize       2048 B
oobsize         64 B
PMON>
```

5.6.4 遗留问题和相关解决

PMON 的问题主要还是在于 cache，让我印象最深的两点分别为：cache 指令初始化导致崩溃和读取内存导致崩溃两点。

其一，由于时间问题，我们没有时间再去实现 cache 指令去进行初始化，所以我们对于 cache 指令的实现均解析为了 NOP，将此步骤交由复位过程让 FPGA 手动实现，虽然这种形式能够有效的解决崩溃问题，但是相应地，这样的设计会导致运行的繁琐和对设计思路的违背，所以如果由机会，我们还是希望在此部分做出改进。

其二，此步虽然并非是 PMON 的崩溃，是后续 μ core 崩溃与此部分密切相关，我们需要检测 cache 大小，大小和预设不符会极大的妨碍我们操作系统的运行，在 MIPS32 手册后我们修改了 cp0 寄存器中的初
始值，随后此 bug 得以解决。

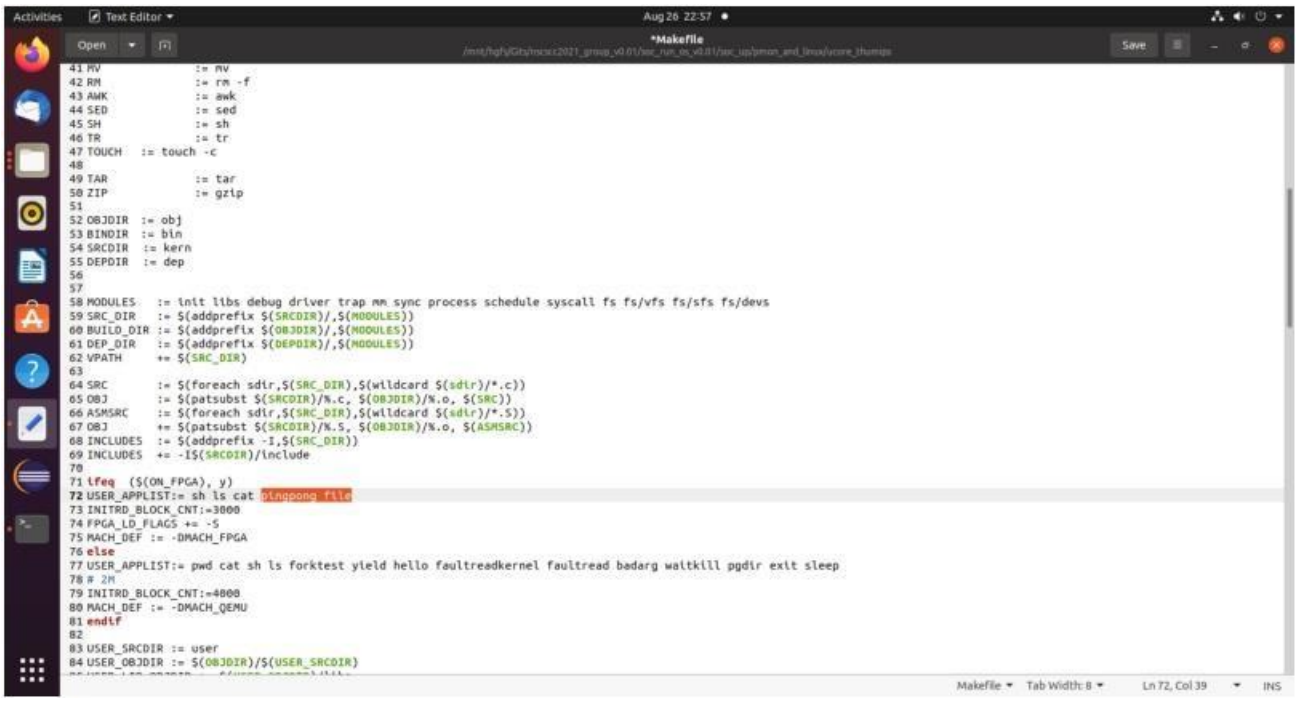
5.7 μcore

5.7.1 测试程序

测试程序均需要放在 /user/ 文件夹下，并修改相关 Makefile 文件进行加载。

5.7.2 Makefile 文件的修改

由于需要新增程序，所以我们需要在 Makefile 文件的低 72 行添加我们所编写的代码文件进入其中，并将第 73 行的 INITRD_BLOCK_CNT 扩大到对应数值（建议为 3000），最终在 bash 下针对 FPGA 上板的运行 m 模式进行 make 即可（此处命令为 make ON_FPGA=Y ）。



第一个代码为简单的输出打印测试：

```
1.  /*输出测试：file.c*/
2.
3.  #include <stdio.h>
4.
5.  int main() {
6.      cprintf(" _    _  _____ _____  _____ _____  _____  _____  _____  __
\n");
7.      cprintf("| \\ | |/  ___/  __ \\/  ___/  __ \\/  __ \\/ __  \\|  _  |/ __
\\/  | \n");
8.      cprintf("|  \\| |\\ `--.| /  \\/\\ `--
.| /  \\/| /  \\/` ' / /'| |/' |`' / /'`| | \n");
```

```
    cprintf("| . ` | `--. \\ |     `--
. \\ |    | |      / /   |  /| |   / /    | | \n");
    cprintf("| |\\  |/\\__/ / \\__/\\/\\__/ / \\__/\\| \\__/\\./ /___\\ |_/ /.
/ /____| |_\n");
    cprintf("\\_| \\_/\\____/ \\____/\\____/ \\____/ \\____/\\_____/ \\___/ \\
_____/\\___/\n");

    return 0;
}
```

第二个代码为多线程的 pingpong 测试，不过由于 μcore 并没有实现管道，所以需要在代码中额外做一层嵌套。

```
/*多线程测试：pingpong.c*/

#include <ulib.h>
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <file.h>
#include <stat.h>
#include <dirent.h>
#include <unistd.h>
// #include "../kern/fs/sysfile.h"

#define printf(...)                     fprintf(1, __VA_ARGS__)
int
main(int argc, char *argv[])
{
    char info;
    int ret;
    int p;

    if ((ret = fork()) == 0)
    {
        printf("%d: received ping\n", getpid());
        write(p, &info, 1);

        close(p);
    }
    else
    {

        read(p, &info, 1);
        printf("%d: received pong\n", getpid());

        close(p);
    }
    exit(0);
}
```

如图运行成功即可

```
round-trip min/avg/max = 1.521/1.960/2.649 ms
PMON> load tftp://169.254.8.103/ucore-kernel-initrd
Loading file: tftp://169.254.8.103/ucore-kernel-initrd (elf)
0x80000000/1409424 + 0x80158190/13072(z) + 302 syms-
Entry address is 80000000
PMON> g
   zero      at       v0       v1       a0       a1       a2       a3
 00000000 00000000 00000000 00000000 00000001 a7dffda8 a7dffdb0 870af5b0
    t0       t1       t2       t3       t4       t5       t6       t7
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    s0       s1       s2       s3       s4       s5       s6       s7
 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
    t8       t9       k0       k1       gp       sp       s8       ra
 00000000 00000000 00000000 00000000 00000000 a7dffd88 00000000 8707ad48
++setup timer interrupts
Initrd: 0x8005e190 - 0x8015818f, size: 0x000fa000, magic: 0x2f8dbe2a
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x80000128 (phys)
  etext 0x8002B400 (phys)
  edata 0x80158190 (phys)
  end   0x8015B4A0 (phys)
Kernel executable memory footprint: 1217KB
memory management: buddy_pmm_manager
memory map:
    [80000000, 82000000]

freemem start at: 8019C000
free pages: 00001E64
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
726F6300
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
--------------------- END ---------------------
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: RR_scheduler
ramdisk_init(): initrd found, magic: 0x2f8dbe2a, 0x000007d0 secs
sfs: mount: 'simple file system' (153/97/250)
vfs: mount disk0.
kernel_execve: pid = 2, name = "sh".
user sh is running!!!
$ ls
 0 is  [directory] 2(hlinks) 9(blocks) 2304(bytes) : 0'.'
    [d]    2(h)        9(b)     2304(s)   .
    [d]    2(h)        9(b)     2304(s)   ..
    [-]    1(h)       21(b)    85097(s)   pingpong
    [-]    1(h)       21(b)    85139(s)   cat
    [-]    1(h)        1(b)       21(s)   test.txt
    [-]    1(h)       22(b)    89406(s)   sh
    [-]    1(h)       21(b)    85086(s)   file
    [-]    1(h)       21(b)    85265(s)   ls
    [-]    1(h)       21(b)    85119(s)   pwd
lsdir: step 4
$ cat test.txt
hello world! Haha...
$
```

```
$ cat test.txt
hello world! Haha...
$ pingpong
5: received pong
6: received ping
$ pwd
disk0:/
$ file
```

```
$
```

5.7.3 遗留问题与相关解决

其实 μcore 部分的问题更像是一种细节错误，我们在源码中尚不能运行的部分中加入了一行

换行输出，这样就通过运行，不过关于此错误的问题为什么能修正我们也由于时间紧迫并没有做出合理的解释，只能说猜想可能在于先前的时序部分带有一些未能发现的遗留问题，所以在ddr3 还没协会就做出了访问？

其次是由于 TLB 异常处理导致时钟变慢的问题，由于我们的处理方法是修改 exception 和译码阶段这样的改动也相应的让异常向译码传输的最长通路加长，不过讨论后认为此问题可以通过增加流水线层数解决。

6 其他部分

6.1 个人体会

不得不说，龙芯确实是计算机硬件领域中含金量较高的一项比赛，它和操作系统大赛，编译器大赛被我们戏称为“铁人三项“。对比于其他比赛，纵向对比其他工程赛龙芯杯无疑是难度高，专业性强，再横向对比 ICPC/CCPC 的话，由于近年来 ICPC/CCPC 奖牌超发的情况下，龙芯杯的奖牌又更强的“抗通胀”价值。

不过这次比赛对于我而言只能说一言难尽吧，先说收获，由于先前我学习的主要方向是算法 和 AI，并且在拿到 CCPC 铜牌和论文之后我对于硬件方向的兴趣并不浓厚，甚至对这个领域中的了解也知之甚少。在龚怿带着我打完操作系统大赛后才开始恶补硬件相关的知识。不过很可惜，直到最后也没有能完全赶上龚怿的进度去分担代码压力，所以只能在初期写文档以 图加快大家的学习速度，在初赛收尾阶段才终于能接手龚怿的 dcache 并参与其中，决赛也只能作为 debug 的辅助去阅读修改源码以及通过 C 语言编写简单的测试程序。不过即使如此也算是重新学习了基于 MIPS 架构的 CPU 知识。

但是同样的，遗憾确实太多了，从一开始的选人和和工期的进展都出现了不同程度的问题，可以说这些问题最终导致我们冲击二等奖的失败了，我们在一开始的选人上就错过了徐绍峰学长的技术支持，而后续蝴蝶效应般的工期灾难更是直接导致了我们在系统阶段任务的完成上过于紧张难以维系合理的工程进展，所以最后的名次，只能是由我们自己吞下苦果了。不过不论如何也只能说吸取教训然后继续走下去，只是希望这样的失误不要再度上演。

7 参考文献

[1]雷思磊.《自己动手写CPU》[M].北京:电子工业出版社,2014.9.

[2]李亚民.《计算机原理与设计：Verilog HDL 版》[M].清华大学出版社

[3]汪文祥.《CPU设计实战》[M].北京:机械工业出版社,2021.1

[4] David A. Patterson《计算机组成与设计 硬件/软件接口》[m].北京:机械工业出版社,2020.4

[5]姚勇斌《超标量处理器设计》[m].北京:清华大学出版社,2011