

# 目录

[1 目录 2](#_Toc81504464)

[1 设计简介 4](#_Toc81504465)

[1.1 GadgetMIPS设计简介 4](#_Toc81504466)

[1.2 具体设计思路 4](#_Toc81504467)

[1.3 关于hhhhMIPS 5](#_Toc81504468)

[1.4 源码地址 5](#_Toc81504472)

[2 CPU设计 6](#_Toc81504473)

[2.1 CPU整体设计相关 6](#_Toc81504474)

[2.2 指令集支持 6](#_Toc81504475)

[2.3 流水线设计 7](#_Toc81504476)

[2.3.1 IF取指阶段 7](#_Toc81504477)

[2.3.2 IF/ID锁存器 9](#_Toc81504478)

[2.3.3 ID译码阶段 10](#_Toc81504479)

[2.3.4 ID/EX锁存器 11](#_Toc81504480)

[2.3.5 EX执行阶段 12](#_Toc81504481)

[2.3.6 EX/MEM锁存器 14](#_Toc81504482)

[2.3.7 MEM访存阶段 14](#_Toc81504483)

[3 总线设计 17](#_Toc81504484)

[3.1 类SRAM总线部分 17](#_Toc81504485)

[3.2 AXI总线部分 18](#_Toc81504486)

[3.2.1 乱序实现 19](#_Toc81504487)

[3.2.2 变长burst传输 19](#_Toc81504488)

[3.2.3 AXI部分信号 20](#_Toc81504489)

[3.2.4 内部状态转换 20](#_Toc81504490)

[3.2.5 读写优先级处理 21](#_Toc81504491)

[3.2.6 数据缓存与选择 22](#_Toc81504492)

[4 针对cache的流水化设计 24](#_Toc81504493)

[4.1 流水线化cache的状态机转换 24](#_Toc81504494)

[5 系统设计 26](#_Toc81504495)

[5.1 TLB设计 26](#_Toc81504496)

[5.2 查询流程 26](#_Toc81504497)

[5.3 TLB架构 26](#_Toc81504498)

[5.4 TLB相关表项和寄存器 29](#_Toc81504499)

[5.4.1 TLB表项 29](#_Toc81504500)

[5.4.2 EntryHi 30](#_Toc81504501)

[5.4.3 EntryLo 30](#_Toc81504502)

[5.4.4 Index 31](#_Toc81504503)

[5.5 TLB指令支持 31](#_Toc81504504)

[5.5.1 TLBP、TLBR 31](#_Toc81504505)

[5.5.2 TLBWI、TLBWR 32](#_Toc81504506)

[5.5.3 PMON 33](#_Toc81504507)

[5.6 运行命令 33](#_Toc81504508)

[5.7 μcore 35](#_Toc81504509)

[5.7.1 遗留问题 35](#_Toc81504510)

[5.7.2 测试程序 35](#_Toc81504511)

[6 其他部分 38](#_Toc81504512)

[6.1 个人体会 38](#_Toc81504513)

[7 参考文献 38](#_Toc81504514)

# 设计简介

## GadgetMIPS设计简介

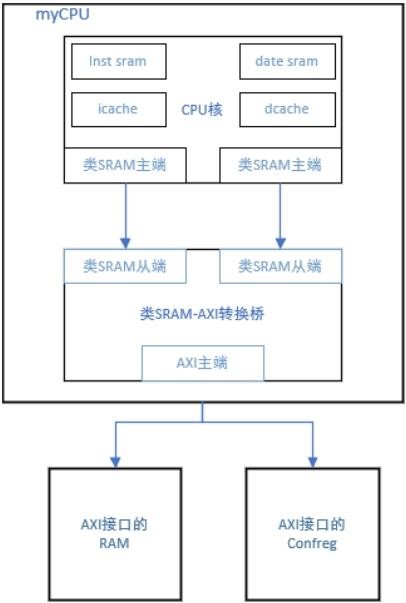
由于龙芯杯大赛的时间与我校小学期的时间重合，所以在吴磊老师的支持下我们选择以赛代练，通过完成龙芯杯的比赛来达到小学期的学习效果。

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. 在大赛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，还可以处理其他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。

## 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；



## 关于hhhhMIPS

由于后续决赛的查重的限制，我们组并没有进入决赛，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谭智文学长，龚怿和笔者去实现。



## 源码地址

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

hhhhMIPS：https://gitee.com/tzwkearn/ncut\_cpu\_2/tree/tlb

# CPU设计

## CPU整体设计相关

GadgetMIPS中使用的icache和CPU部分是基于去年参赛使用的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比赛的遗憾。

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比赛需求通过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。

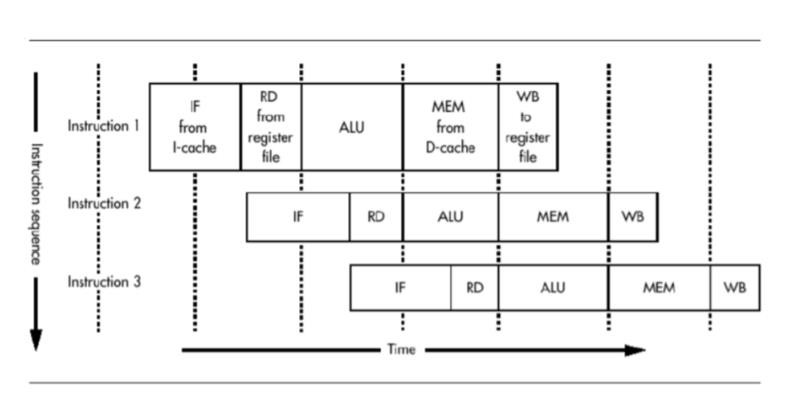
最后便是cache部分的修改和补充，由于AXI版本的dcache我们并没有成功实现，所以最终 我们还是只能采取SRAM的dcache，不过考虑到性能影响，最终还是采取了写回的方式配合 二路组相联的方式进行。同样我们在赛后也发现了一个问题，我们的dcache因为失误导致写 成了16k，这样导致我们二路组相联并没有实质性的提高性能，反而导致了降频。

## 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

## 流水线设计

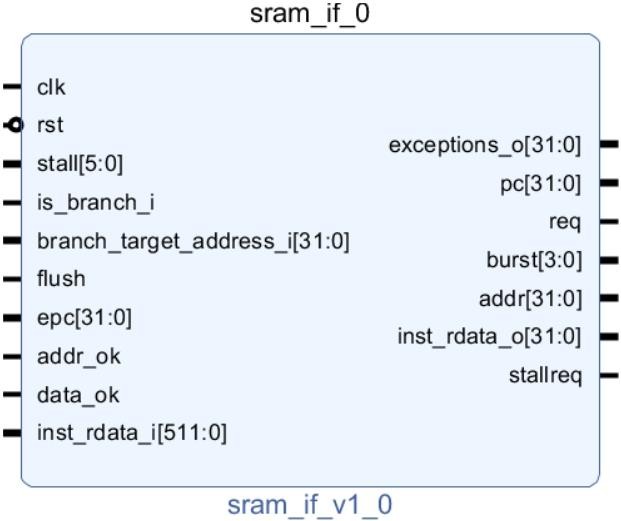
GadgetMIPS使用MIPS经典五级流水线架构，即取指 IF、译码 ID、执行 EX、访存 MEM 和 写回 WB，这使得CPU可以更充分地利用其内部不同功能的器件以提高CPU执行效率。



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。

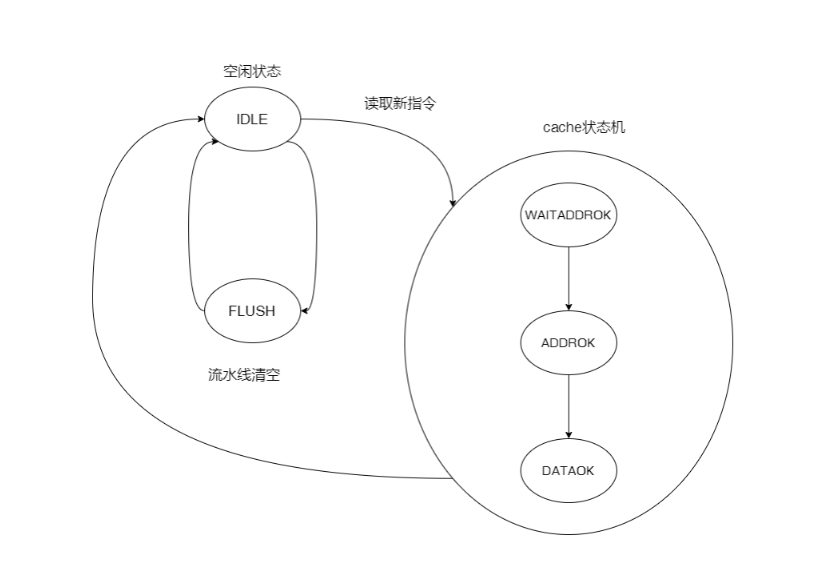
### IF取指阶段



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。

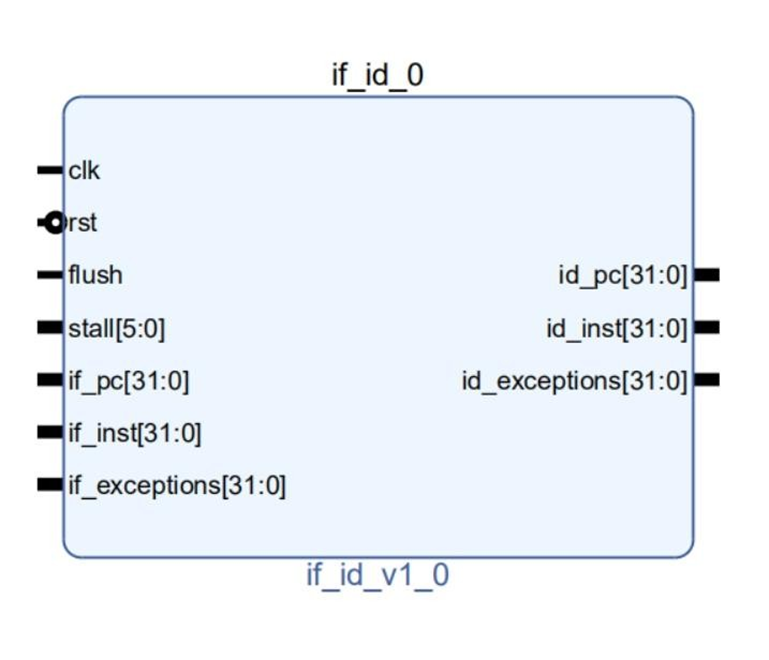
此外，该阶段还会对译码阶段的跳转指令给出响应：译码阶段给出一个跳转信号和一个新的 程序地址，该阶段响应该信号并将新地址写入PC，然后根据新的PC取指令。 IF模块内集成 了icache，设计为单块16个字、单路256块的架构，未命中时向外访问。由于取指阶段实际 上并不存在写操作，最终只需考虑读取数据并缓存的实现。

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）。



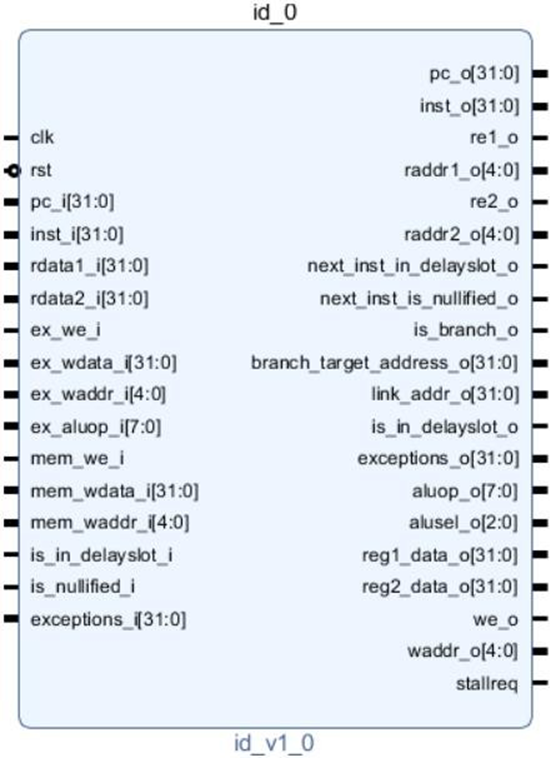
1. always @ (posedge clk) begin
2. **if** (rst == `RstEnable) begin
3. state <= `IDLE;
4. req <= 1'b0;
5. flush\_wait <= 1'b0;
6. end **else** **if** (flush) begin
7. **if**(state != `IDLE && state != `DATAOK) begin
8. state <= `FLUSHWAIT;
9. flush\_wait <= 1'b1;
10. end
11. end **else** **if** (ce == 1'b1) begin
12. **case** (state)
13. `IDLE: begin
14. **if**(pc[1:0] != 2'b00) begin
15. req <= 1'b0;
16. end **else** **if**(!hit) begin//cache读缺失
17. state <= `WAITADDROK;//进入等待地址确认状态
18. req <= 1'b1;
19. end **else** begin
20. req <= 1'b0;
21. end
22. end
23. `WAITADDROK: begin
24. **if**(addr\_ok == 1'b1) begin
25. req <= 1'b0;
26. state <= `ADDROK;
27. end
28. end
29. `ADDROK: begin
30. **if**(data\_ok == 1'b1) begin
31. state <= `DATAOK;
32. end
33. end
34. `DATAOK: begin
35. state <= `IDLE;
36. end
37. `FLUSHWAIT: begin
38. **if**(data\_ok) begin
39. state <= `IDLE;
40. flush\_wait <= 1'b0;
41. end
42. end
43. endcase
44. end
45. end

### 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。

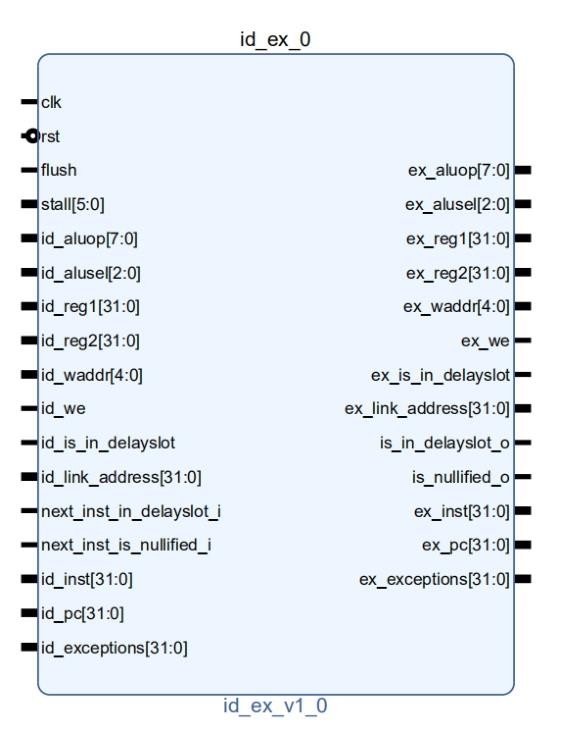
### ID译码阶段



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 对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）

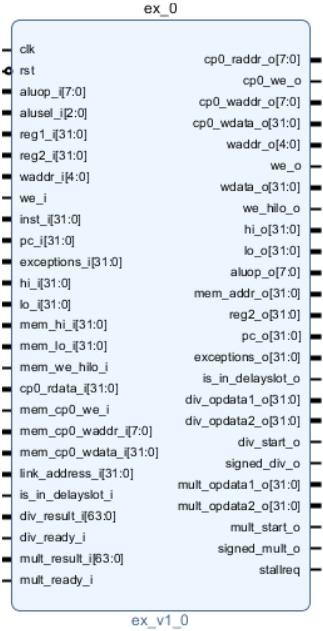
1. always @ (posedge clk) begin
2. **if**(rst == `RstEnable) begin
3. exception\_is\_refetch = 1'b0;
4. end **else** begin
5. **case** (exception\_is\_refetch)
6. 1'b0: begin
7. **if**(exception\_is\_tlbwi |
8. exception\_is\_tlbwr |
9. exception\_is\_tlbp  |
10. exception\_is\_tlbr) begin
11. exception\_is\_refetch <= 1'b1;
12. end **else** begin
13. exception\_is\_refetch <= 1'b0;
14. end
15. end
16. 1'b1: begin
17. exception\_is\_refetch <= 1'b0;
18. end
19. **default**: begin
20. end
21. endcase
22. end
23. end
25. ...
26. assign exceptions\_o = {exception\_is\_tlbp,
27. exception\_is\_tlbr,
28. exception\_is\_tlbwi,
29. exception\_is\_tlbwr,
30. exception\_is\_refetch,
31. exceptions\_i[26:5],
32. exception\_is\_syscall,
33. exception\_is\_break,
34. exception\_is\_eret,
35. inst\_valid,
36. exceptions\_i[0]};

### 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。

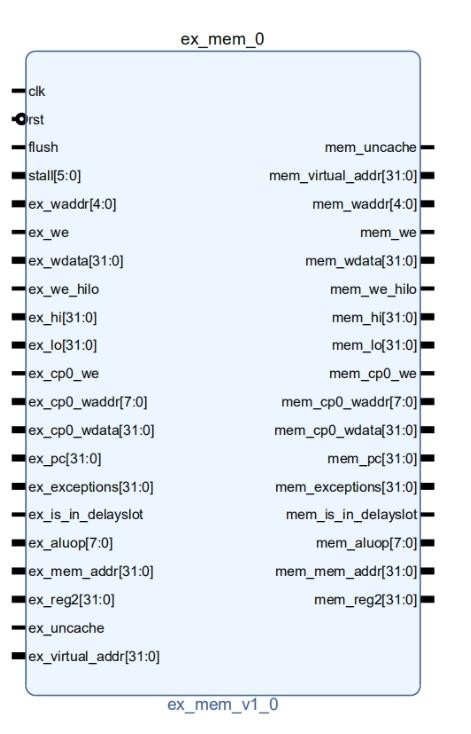
### EX执行阶段



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。

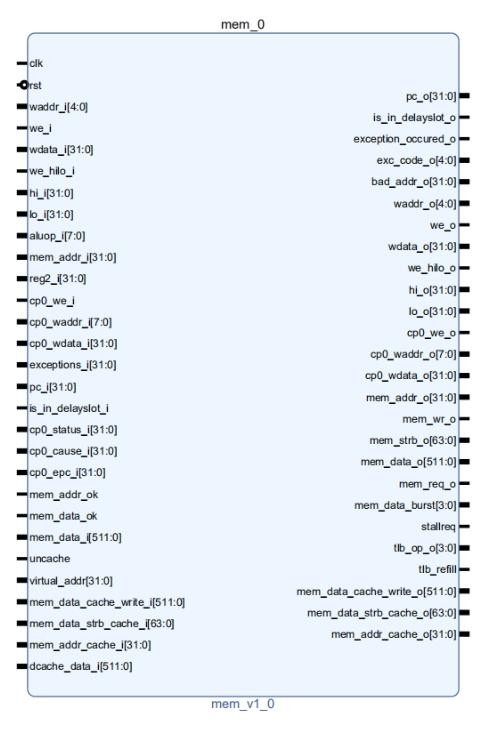
1. assign exceptions\_o = {exceptions\_i[31:12],
2. tlb\_mapped & load\_store ? (((tlb\_hit & store) & (tlb\_v & !tlb\_d)) ? 1'b1 : 1'b0) : 1'b0,
3. tlb\_mapped & load\_store ? !tlb\_hit : 1'b0,
4. tlb\_mapped & load\_store ? (tlb\_hit & !tlb\_v ? 1'b1 : 1'b0) : 1'b0,
5. exceptions\_i[8:7],
6. trap\_occured,
7. overflow\_occured,
8. exceptions\_i[4:0]};
9. // 判断是否为数据操作
10. wire load\_store = (aluop\_i == `MEM\_OP\_LB  |
11. aluop\_i == `MEM\_OP\_LH  |
12. aluop\_i == `MEM\_OP\_LWL |
13. aluop\_i == `MEM\_OP\_LW  |
14. aluop\_i == `MEM\_OP\_LBU |
15. aluop\_i == `MEM\_OP\_LHU |
16. aluop\_i == `MEM\_OP\_LWR |
17. aluop\_i == `MEM\_OP\_SB  |
18. aluop\_i == `MEM\_OP\_SH  |
19. aluop\_i == `MEM\_OP\_SWL |
20. aluop\_i == `MEM\_OP\_SW  |
21. aluop\_i == `MEM\_OP\_SWR);
23. wire store = (aluop\_i == `MEM\_OP\_SB  |
24. aluop\_i == `MEM\_OP\_SH  |
25. aluop\_i == `MEM\_OP\_SWL |
26. aluop\_i == `MEM\_OP\_SW  |
27. aluop\_i == `MEM\_OP\_SWR);

### 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。

### 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：

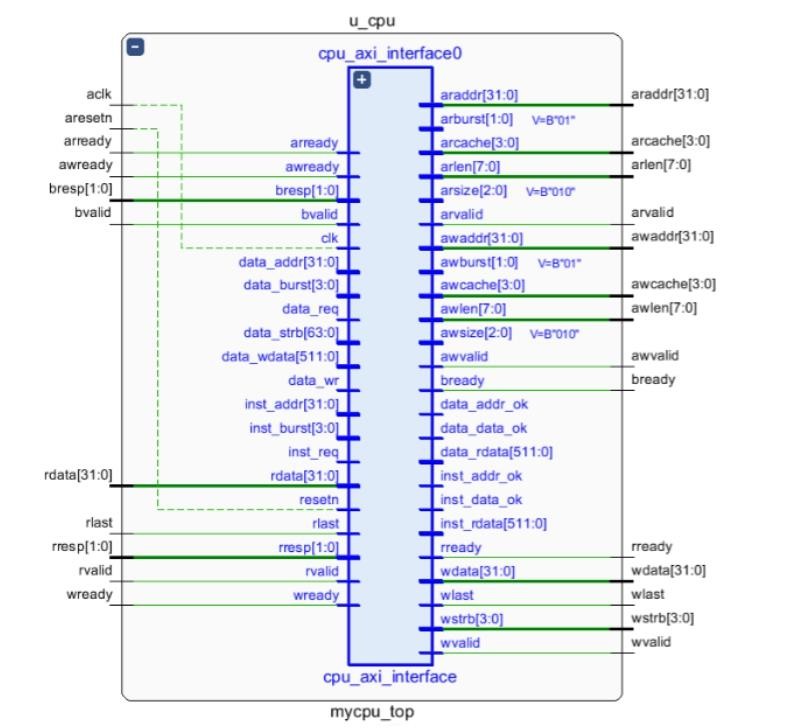
1. always @(\*) begin
2. **if**(rst == `RstEnable)begin
3. stallreq <= 1'b0;
4. end **else** begin
5. **case**(state)
6. `IDLE: begin
7. **if**(mem\_ce) begin
8. stallreq <= ~hit\_total;
9. end **else** begin
10. stallreq <= `False\_v;
11. end
12. end
13. `READWAIT: begin
14. stallreq <= `True\_v;
15. end
16. `READOK: begin
17. stallreq <= `True\_v;
18. end
19. `WRITEWAIT: begin
20. stallreq <= `True\_v;
21. end
22. `WRITEOK: begin
23. stallreq <= `True\_v;
24. end
25. `UNCACHEWAIT: begin
26. stallreq <= `True\_v;
27. end
28. `UNCACHEOK: begin
29. **if** (!mem\_data\_ok) begin
30. // 数据握手不成功，原地等待
31. stallreq <= `True\_v;
32. end **else** begin
33. // 数据握手成功，立刻撤销流水线暂停
34. // 转入空闲阶段
35. stallreq <= `False\_v;
36. end
37. end
38. **default**: begin
39. stallreq <= `False\_v;
40. end
41. endcase
42. end
43. 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。

# 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。



## 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：



## 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；

### 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。

### 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：

1. 2'b01: begin
2. // 写AR通道进行AXI地址握手
3. **if** (arready && arvalid) begin
4. // AR握手成功
5. read\_counter <= (4'b1111 - arlen[3:0]); // 清零counter
6. // 如果arlen是burst传输，则低位是f，减去后counter正好为0
7. // 如果不是burst传输，保证最后一次传输一定写在31:0处
8. arvalid <= 1'b0; // 撤销握手信号
9. read\_status <= 2'b10;
10. rready <= 1'b1; // 准备接收数据
11. end **else** begin
12. arvalid <= 1'b1; // 保持AR握手
13. read\_status <= 2'b01;
14. rready <= 1'b0;
15. end
16. end
18. 2'b10: begin
19. // 等待R通道的数据握手
20. **if** (rready && rvalid) begin
21. // 本次握手成功
22. read\_result[read\_counter] <= rdata;
23. read\_counter <= read\_counter + 1;
25. // 最后一个则结束传输
26. **if** (rlast) begin
27. // 这里设置读结束，下一个周期应该关闭读状态机使能
28. // 从而中断状态机的执行，否则就原地等待
29. read\_if\_or\_mem[0] <= 1'b1; // 表示读结束
30. rready <= 1'b0;
31. end
32. end
33. end

### AXI部分信号

此部分并非全部AXI信号，仅为此项目中定义的

1. //inst sram-like
2. input wire inst\_req ,
3. input wire[3:0] inst\_burst , // 0000 -> 1 word, 1111 -> 16 words
4. input wire[31:0] inst\_addr ,
5. output reg [511:0] inst\_rdata ,
6. output wire inst\_addr\_ok ,
7. output reg inst\_data\_ok ,
8. //data sram-like
9. 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

### 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；

### 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：

1. // SRAM握手
2. always @ (posedge clk) begin
3. **if** (!resetn) begin
4. ...
5. end **else** begin
6. // 正常逻辑
7. **if** (!write\_en) begin
8. // 当前没有写操作
9. **if** (data\_wr) begin
10. // 如果上一个是写
11. data\_data\_ok <= 1'b0; // 清除数据握手
12. end
13. **if** (data\_req && data\_wr) begin
14. ...
15. end **else** begin
16. // 不写则保证写状态机关闭
17. write\_en <= 1'b0;
18. end
19. end **else** begin
20. // 当前有写操作
21. **if** (write\_done) begin
22. // 写完了
23. data\_data\_ok <= 1'b1; // 进行数据握手
24. write\_en <= 1'b0; // 关闭写状态机
25. end **else** begin
26. // 还在写
27. data\_data\_ok <= 1'b0; // 不握手
28. write\_en <= 1'b1; // 保持写状态机打开
29. end
30. end
32. **if** (!read\_en) begin
33. // 当前没有读操作
34. **if** (!data\_wr) begin
35. // 如果上一个是读
36. data\_data\_ok <= 1'b0; // 清除数据握手
37. end
38. inst\_data\_ok <= 1'b0; // 清除数据握手
39. **if** (data\_req && !data\_wr) begin
40. // data要读
41. // 记录读信息
42. ...
44. read\_en <= 1'b1; // 启动读状态机
45. end **else** **if** (inst\_req) begin
46. // inst要读
47. ...
49. read\_en <= 1'b1; // 启动读状态机
50. end **else** begin
51. // 不写则保证读状态机关闭
52. read\_en <= 1'b0;
53. end
54. end **else** begin
55. // 当前有读操作
56. **if** (read\_if\_or\_mem[1]) begin
57. // 读data
58. **if** (read\_if\_or\_mem[0]) begin
59. // 读完了
60. ...
61. data\_data\_ok <= 1'b1; // 进行数据握手
62. read\_en <= 1'b0; // 关闭读状态机
63. end **else** begin
64. // 还在写
65. data\_data\_ok <= 1'b0; // 不握手
66. read\_en <= 1'b1; // 保持读状态机打开
67. end
68. end **else** begin
69. // 读inst
70. **if** (read\_if\_or\_mem[0]) begin
71. // 读完了
72. ...
74. inst\_data\_ok <= 1'b1; // 进行数据握手
75. read\_en <= 1'b0; // 关闭读状态机
76. end **else** begin
77. // 还在写
78. inst\_data\_ok <= 1'b0; // 不握手
79. read\_en <= 1'b1; // 保持读状态机打开
80. end
81. end
82. end
83. end
84. end

### 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。

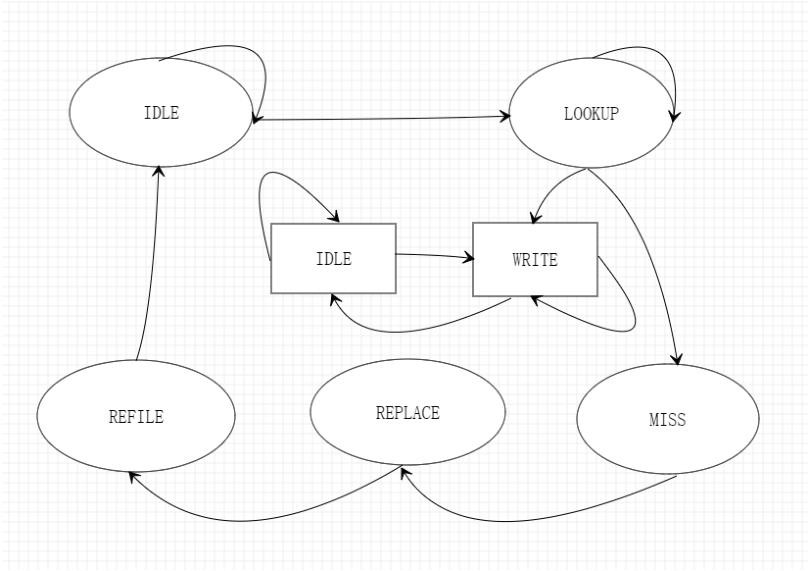
只有当数据被全部缓存时，addr\_ok有效信号才会传回给CPU； 同样，当数据被全部交换完毕后，总线桥才会CPU将data\_ok设为有效。

1. reg[31:0] read\_addr;
2. reg[3:0] read\_burst;
3. reg[31:0] read\_result[15:0];
4. // [1]: 0 inst, 1 data; [0]: 0 reading, 1 done.
5. reg[1:0] read\_if\_or\_mem;
6. reg read\_en; // 使能读状态机，为1时读状态机开始启动
8. reg[31:0] write\_addr;
9. reg[3:0] write\_burst;
10. reg[31:0] write\_data[15:0];
11. reg[3:0] write\_strb[15:0];
12. reg write\_done; // 0 writing, 1 done
13. reg write\_en; // 使能写状态机，为1时写状态机开始启动
15. ...
17. **if** (read\_if\_or\_mem[1]) begin
18. // 读data
19. **if** (read\_if\_or\_mem[0]) begin
20. // 读完了
21. data\_rdata[511:480] <= read\_result[0];
22. data\_rdata[479:448] <= read\_result[1];
23. data\_rdata[447:416] <= read\_result[2];
24. data\_rdata[415:384] <= read\_result[3];
25. ...
26. end **else** begin
27. // 读inst
28. **if** (read\_if\_or\_mem[0]) begin
29. // 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];

# 针对cache的流水化设计

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学长和龚怿的讲解，如有纰 漏，还请指出，在此写出此部分权当提醒与纪念。

## 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\_ yalid== 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；



1. **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）；
2. **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；
3. **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。

# 系统设计

## 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。

## 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].Mask ))) 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
12. c ← TLB[i].C1
13. d ← TLB[i].D1
14. endif
15. **if** v = 0 then
16. SignalException(TLBInvalid, reftype)
17. endif
18. **if** (d = 0) and (reftype = store) then
19. SignalException(TLBModified)
20. endif
21. # pfn 19..0 corresponds to pa 31..12
22. pa ← pfn19..0 || va11..0
23. found ← 1
24. **break**
25. endif
26. endfor
28. **if** found = 0 then
29. SignalException(TLBMiss, reftype)
30. endif

## 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

1. module tlb\_mmu(
2. input wire clk,
3. input wire rst,
4. ///////////////////////////////////////////////////////////
5. // tlb\_op:
6. // 3: tlbP
7. // 2: tlbr
8. // 1: tlbwi
9. // 0: tlbwr
10. ///////////////////////////////////////////////////////////
11. input wire[3:0] tlb\_op,
12. // icache读tlb接口: if
13. input wire[`RegBus] inst\_virtual\_pc\_i,
14. output reg[`RegBus] inst\_physical\_pc\_o,
15. output reg inst\_tlb\_hit,
16. output reg[2:0] inst\_c,
17. output reg inst\_d,
18. output reg inst\_v,
19. output reg inst\_mapped,
20. // mem
21. input wire[`RegBus] data\_virtual\_pc\_i,
22. output reg[`RegBus] data\_physical\_pc\_o,
23. output reg data\_tlb\_hit,
24. output reg[2:0] data\_c,
25. output reg data\_d,
26. output reg data\_v,
27. output reg data\_mapped,
28. // cp0 output
29. input wire[`RegBus] index\_i,
30. input wire[`RegBus] random\_i,
31. input wire[`RegBus] entryLo0\_i,
32. input wire[`RegBus] entryLo1\_i,
33. input wire[`RegBus] entryHi\_i,
34. input wire[`RegBus] pageMask\_i,
35. // cp0 input
36. output reg[`RegBus] index\_o,
37. output reg[`RegBus] entryLo0\_o,
38. output reg[`RegBus] entryLo1\_o,
39. output reg[`RegBus] entryHi\_o,
40. output reg[`RegBus] pageMask\_o
41. );

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：

1. // inst 读
2. always @ (\*) begin
3. **if**(rst == `RstEnable) begin
4. inst\_tlb\_hit <= `TLBMiss;
5. inst\_physical\_pc\_o <= `ZeroWord;
6. inst\_c <= 3'b000;
7. inst\_d <= 1'b0;
8. inst\_v <= 1'b0;
9. inst\_mapped <= 1'b0;
10. end **else** begin
11. inst\_tlb\_hit <= `TLBMiss;
12. inst\_physical\_pc\_o <= `ZeroWord;
13. inst\_c <= 3'b000;
14. inst\_d <= 1'b0;
15. inst\_v <= 1'b0;
16. inst\_mapped <= 1'b0;
17. **if**(inst\_virtual\_pc\_i[31:30] == 2'b10) begin
18. // 此时地址处于unmapped区域
19. inst\_tlb\_hit <= `TLBHit;
20. inst\_physical\_pc\_o <= {3'b000, inst\_virtual\_pc\_i[28:0]};
21. inst\_c <= inst\_virtual\_pc\_i[29] ? 3'b010:3'b011;                // 'uncached' or 'cacheable noncoherent'
22. inst\_d <= 1'b0;                                                 // default as 'clean'
23. inst\_v <= 1'b1;                                                 // default as 'valid'
24. inst\_mapped <= 1'b0;
25. end **else** begin
26. inst\_tlb\_hit <= (is\_hit\_inst ? `TLBHit:`TLBMiss);
27. inst\_physical\_pc\_o <= {inst\_pfn\_result, inst\_virtual\_pc\_i[`VA\_OFFSET]};
28. inst\_c <= inst\_c\_result;
29. inst\_d <= inst\_d\_result;
30. inst\_v <= inst\_v\_result;
31. inst\_mapped <= 1'b1;
32. end
33. end
34. end
36. // data 读
37. always @ (\*) begin
38. **if**(rst == `RstEnable) begin
39. data\_tlb\_hit <= `TLBMiss;
40. data\_physical\_pc\_o <= `ZeroWord;
41. data\_c <= 3'b000;
42. data\_d <= 1'b0;
43. data\_v <= 1'b0;
44. data\_mapped <= 1'b0;
45. end **else** begin
46. data\_tlb\_hit <= `TLBMiss;
47. data\_physical\_pc\_o <= `ZeroWord;
48. data\_c <= 3'b000;
49. data\_d <= 1'b0;
50. data\_v <= 1'b0;
51. data\_mapped <= 1'b0;
52. **if**(data\_virtual\_pc\_i[31:30] == 2'b10) begin
53. // 此时地址处于unmapped区域
54. data\_tlb\_hit <= `TLBHit;
55. data\_physical\_pc\_o <= {3'b000, data\_virtual\_pc\_i[28:0]};
56. data\_c <= data\_virtual\_pc\_i[29] ? 3'b010:3'b011;                // 'uncached' or 'cacheable noncoherent'
57. data\_d <= 1'b0;                                                 // default as 'clean'
58. data\_v <= 1'b1;                                                 // default as 'valid'
59. data\_mapped <= 1'b0;
60. end **else** begin
61. data\_tlb\_hit <= (is\_hit\_data ? `TLBHit:`TLBMiss);
62. data\_physical\_pc\_o <= {data\_pfn\_result, data\_virtual\_pc\_i[`VA\_OFFSET]};
63. data\_c <= data\_c\_result;
64. data\_d <= data\_d\_result;
65. data\_v <= data\_v\_result;
66. data\_mapped <= 1'b1;
67. end
68. end
69. 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。

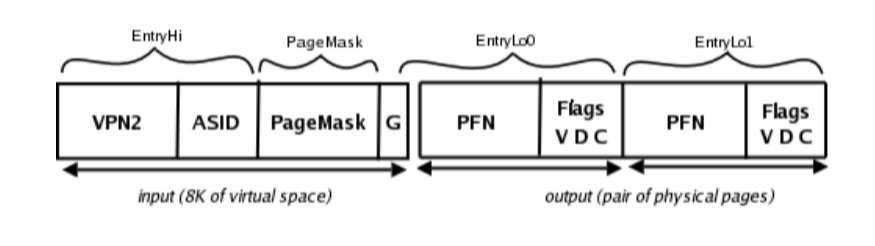
所以最好的一个办法是利用逐位或运算，让电路并行完成比较任务，这就意味着效率是多倍 的提升。

1. 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][`TLB\_G]);
2. 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][`TLB\_G]);
3. 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][`TLB\_G]);
4. 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][`TLB\_G]);

7. wire hit\_inst[0:15];
8. wire is\_hit\_inst;
9. assign is\_hit\_inst = hit\_inst[ 0] | hit\_inst[ 1] | hit\_inst[ 2] | hit\_inst[ 3] |
10. hit\_inst[ 4] | hit\_inst[ 5] | hit\_inst[ 6] | hit\_inst[ 7] |
11. hit\_inst[ 8] | hit\_inst[ 9] | hit\_inst[10] | hit\_inst[11] |
12. hit\_inst[12] | hit\_inst[13] | hit\_inst[14] | hit\_inst[15];
14. wire hit\_data[0:15];
15. wire is\_hit\_data;
16. assign is\_hit\_data = hit\_data[ 0] | hit\_data[ 1] | hit\_data[ 2] | hit\_data[ 3] |
17. hit\_data[ 4] | hit\_data[ 5] | hit\_data[ 6] | hit\_data[ 7] |
18. hit\_data[ 8] | hit\_data[ 9] | hit\_data[10] | hit\_data[11] |
19. hit\_data[12] | hit\_data[13] | hit\_data[14] | hit\_data[15];

## TLB相关表项和寄存器

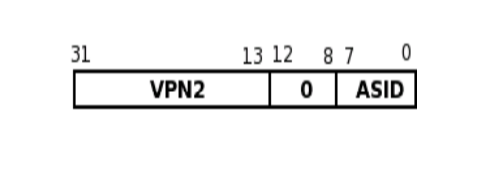
### TLB表项



单个TLB表项长度为90位，它包括：

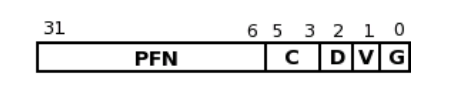
* **VPN2：**19位，虚拟地址高位。虚拟地址同样有该部分，查询TLB过程实际上即为虚拟地址匹配该字段、同时返回对应信息的过程；
* **ASID：**8位，地址空间标识。当查得的TLB表项不为全局可用时（即**G**不为有效），还需要对此字段进行新的比较过程；
* **PageMask：**12位掩码
* **G：**1位，全局属性标识符。它表示这个TLB表项是否全局可用，若无效，则使用该TLB表项时还要额外核对**ASID**字段；
* **EntryLo：**共有两个部分，和EntryLo0、EntryLo1进行数据交换。

### ****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。

### 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。

### 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。

## TLB指令支持

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



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。

### 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;
4. wire[`TLBWayBus]   hit\_hi\_match\_switch;
5. assign hit\_hi\_match\_switch =    (hit\_hi\_match[0] ? 0:0)        |   (hit\_hi\_match[1] ? 1:0)         |
6. (hit\_hi\_match[2] ? 2:0)        |   (hit\_hi\_match[3] ? 3:0)         |
7. (hit\_hi\_match[4] ? 4:0)        |   (hit\_hi\_match[5] ? 5:0)         |
8. (hit\_hi\_match[6] ? 6:0)        |   (hit\_hi\_match[7] ? 7:0)         |
9. (hit\_hi\_match[8] ? 8:0)        |   (hit\_hi\_match[9] ? 9:0)         |
10. (hit\_hi\_match[10] ? 10:0)        |   (hit\_hi\_match[11] ? 11:0)         |
11. (hit\_hi\_match[12] ? 12:0)        |   (hit\_hi\_match[13] ? 13:0)         |
12. (hit\_hi\_match[14] ? 14:0)        |   (hit\_hi\_match[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\_index][`TLB\_G]};
20. entryLo1\_o <= {6'b000000, tlb\_ram[tlb\_index][24:0],  tlb\_ram[tlb\_index][`TLB\_G]};
21. pageMask\_o <= {7'b0000000, tlb\_ram[tlb\_index][`TLB\_PageMask], 13'b0\_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

### 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。

1. wire[`TLBWayBus] tlb\_index;
2. assign tlb\_index = tlb\_op[0] ? random\_i[`TLBWayBus] : index\_i[`TLBWayBus];
4. // 写入TLB操作: tlbwi tlbwr
5. always @ (posedge clk) begin
6. **if**(tlb\_write\_en) begin
7. tlb\_ram[tlb\_index][`TLB\_VPN2] <= entryHi\_i[`EntryHiVPN2];
8. tlb\_ram[tlb\_index][`TLB\_ASID] <= entryHi\_i[`EntryHiASID];
9. tlb\_ram[tlb\_index][`TLB\_PageMask] <= pageMask\_i[`PageMask\_Mask];
10. tlb\_ram[tlb\_index][`TLB\_G] <= entryLo0\_i[`EntryLoG] & entryLo1\_i[`EntryLoG];
11. tlb\_ram[tlb\_index][`TLB\_PFN0] <= entryLo0\_i[`EntryLoPFN] & (~pageMask\_i[`PageMask\_Mask]);
12. tlb\_ram[tlb\_index][`TLB\_PFN0\_C] <= entryLo0\_i[`EntryLoC];
13. tlb\_ram[tlb\_index][`TLB\_PFN0\_D] <= entryLo0\_i[`EntryLoD];
14. tlb\_ram[tlb\_index][`TLB\_PFN0\_V] <= entryLo0\_i[`EntryLoV];
15. tlb\_ram[tlb\_index][`TLB\_PFN1] <= entryLo1\_i[`EntryLoPFN] & (~pageMask\_i[`PageMask\_Mask]);
16. tlb\_ram[tlb\_index][`TLB\_PFN1\_C] <= entryLo1\_i[`EntryLoC];
17. tlb\_ram[tlb\_index][`TLB\_PFN1\_D] <= entryLo1\_i[`EntryLoD];
18. tlb\_ram[tlb\_index][`TLB\_PFN1\_V] <= entryLo1\_i[`EntryLoV];
19. tlb\_valid[tlb\_index] <= 1'b1;
20. end
21. **else** begin
22. end
23. end

### 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

1. dirty\_ram dirty0(
2. .a(index),
3. //.d(hit),错误代码
4. .d(~cache\_ok),//正确写法
5. .clk(clk),
6. .we(wr | cache\_ok),
7. .spo(dirty\_out)
8. );

## 运行命令

$ 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

## μcore

### 遗留问题

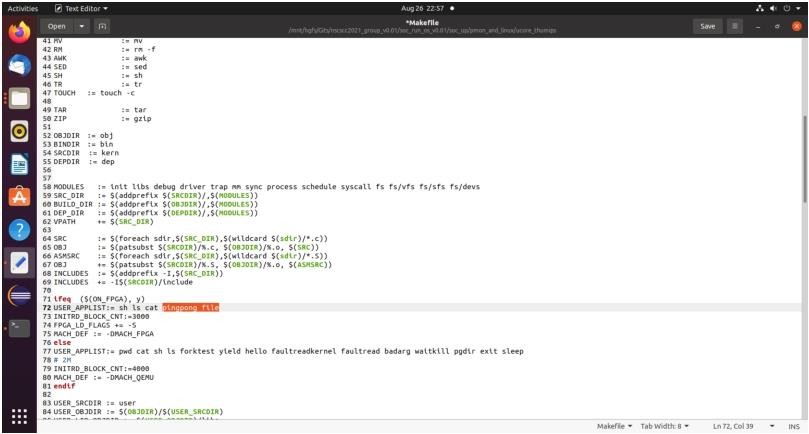
其实μcore部分的问题更像是一种细节错误，我们在源码中尚不能运行的部分中加入了一行 换行输出，这样就通过运行，不过关于此错误的问题为什么能修正我们也由于时间紧迫并没 有做出合理的解释，只能说猜想可能在于先前的时序部分带有一些未能发现的遗留问题，所 以在ddr3还没协会就做出了访问？

### 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。

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\*/
3. #include <stdio.h>
5. **int** main() {
6. cprintf(" \_   \_  \_\_\_\_\_ \_\_\_\_\_  \_\_\_\_\_ \_\_\_\_\_  \_\_\_\_\_  \_\_\_\_\_  \_\_\_\_\_  \_\_\_\_\_  \_\_  \n");
7. cprintf("| \\ | |/  \_\_\_/  \_\_ \\/  \_\_\_/  \_\_ \\/  \_\_ \\/ \_\_  \\|  \_  |/ \_\_  \\/  | \n");
8. cprintf("|  \\| |\\ `--.| /  \\/\\ `--.| /  \\/| /  \\/`' / /'| |/' |`' / /'`| | \n");
9. cprintf("| . ` | `--. \\ |     `--. \\ |    | |      / /  |  /| |  / /   | | \n");
10. cprintf("| |\\  |/\\\_\_/ / \\\_\_/\\/\\\_\_/ / \\\_\_/\\| \\\_\_/\\./ /\_\_\_\\ |\_/ /./ /\_\_\_\_| |\_\n");
11. cprintf("\\\_| \\\_/\\\_\_\_\_/ \\\_\_\_\_/\\\_\_\_\_/ \\\_\_\_\_/ \\\_\_\_\_/\\\_\_\_\_\_/ \\\_\_\_/ \\\_\_\_\_\_/\\\_\_\_/\n");
13. **return** 0;
14. }

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。

1. /\*多线程测试：pingpong.c\*/
3. #include <ulib.h>
4. #include <stdio.h>
5. #include <string.h>
6. #include <dir.h>
7. #include <file.h>
8. #include <stat.h>
9. #include <dirent.h>
10. #include <unistd.h>
11. // #include "../kern/fs/sysfile.h"
13. #define printf(...)                     fprintf(1, \_\_VA\_ARGS\_\_)
14. **int**
15. main(**int** argc, **char** \*argv[])
16. {
17. **char** info;
18. **int** ret;
19. **int** p;
21. **if** ((ret = fork()) == 0)
22. {
23. printf("%d: received ping\n", getpid());
24. write(p, &info, 1);
26. close(p);
27. }
28. **else**
29. {
31. read(p, &info, 1);
32. printf("%d: received pong\n", getpid());
34. close(p);
35. }
36. exit(0);
37. }

如图运行成功即可



# 其他部分

## 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。

# 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