CPU 设计文档报告

哈尔滨工业大学(深圳)　winter spring 队
朱冬春 罗彦恒 郑建川 陈梓晗

一、设计简述

CPU 使用 verilog 语言开发，使用 chiplab 测试框架仿真，实现了基于 LoongArch 基准指令集的七级流水线顺序双发射 CPU。同时 cpu_clk 主频达到 73MHZ，通过所有功能测试点以及性能测试下板实现， myCPU 嵌入到性能测试环境 SoC_AXI_Lite 里综合实现后 WNS 非负值，即不存在违约路径，符合大赛的设计要求。拥有大于 8KB 的指令存储器与数据存储器，实现了大赛要求的基准指令。

二、具体设计

1. 设计简图

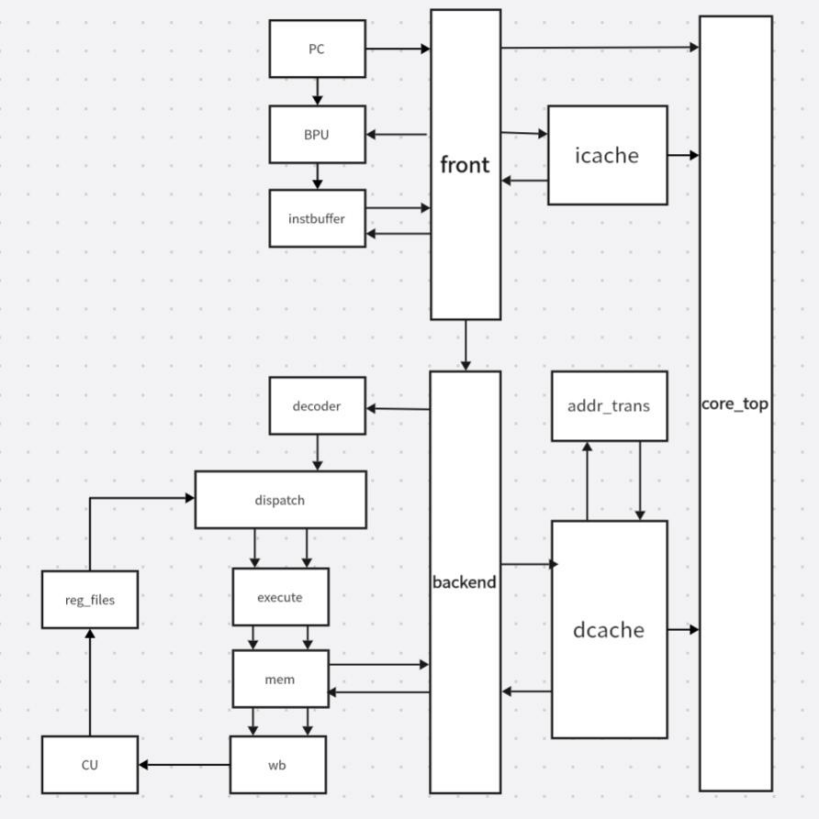

图 1

2. 实现指令

基本运算指令：ADD.W SUB.W ADDI.W SLT SLTU SLTI SLTUI MUL.W MULH.W MULH.WU DIV.W DIV.WU MOD.W MOD.WU

逻辑运算指令：AND OR XOR NOR ANDI ORI XORI SLL.W SRL.W SRA.W

跳转与分支指令：JIRL B BL BEQ BNE BLT BGE BLTU BGEU

访存类指令： LD.B LD.H LD.W LD.BU LD.HU STB STH STW

原子访存指令： LL.W SC.W

特殊立即数加载指令：LU12I PCADDU12I

计数器指令： RDCNTID RDCNTVHW

CSR 相关指令： CSR CSRRD CSRWR

缓存控制指令： CACOP

系统控制指令： CPUCFG DBAR IBAR BREAK SYSCALL IDLE ERTN

3. 各模块设计

(1) 指令存储器 (icache)

① 状态机控制（FSM）

localparam IDLE = 3'b000;

localparam DEALING1 = 3'b001;

localparam DEALING2 = 3'b010;

localparam REFILL = 3'b011;

IDLE: 空闲态，等待新请求

DEALING1/2: 正在处理 miss，需要读内存

REFILL: 从 dev 接口补 cache 内容

② 地址拆解与 cache 索引结构

index = addr[11:5]; // 共 7 位，可表示 128 个 cache line

tag = addr[31:12];　　// 剩余 20 位作为 tag

offset = addr[4:2];　// 指令在 256 位 block 中的位置（8 个指令）

这代表：

每条 cache line = 256 bits = 8 指令

128 个 cache line，分布在 ram1 / ram2 中

③ hit 检测机制

hit_ram1_index1 = (tag == ram1_tag) & 有效位

两个 tag_ram，分别存 tag 和 valid 位（21 位）

两个 data_ram，分别存 256 位数据块

use_bit[]: 表示替换策略（LRU 伪装替换）

④ cache 替换与 refill

如果 miss，根据 use_bit[index] 决定填入 ram1 还是 ram2

使用 cpu_ren 和 cpu_raddr 发出读请求到下一级 dev

返回后写入 ram，并更新 tag 和 valid 位

if(!use_bit[index]) -> 写 ram1

else -> 写 ram2

⑤ 双发射支持

模块每周期支持两个指令地址 inst_addr1 和 inst_addr2：

分别通过 offset1 和 offset2 从 hit 数据块中选出对应 32 位指令

输出为 inst_out1, inst_out2

⑥ CACOP 接口支持

input wire icacop_en;

input wire [1:0] cacop_mode;

若使能 icacop_en，通过 cacop_index 进行 tag 清除或管理

由 tag_ram1 和 tag_ram2 通过 cacop_flush 实现逻辑

⑦ 异常信息与预测信息传播

output [31:0] pred_addr, pred_taken;

output [6:0] pc_exception_cause_out1/2;

用于将 IF 阶段接收的异常或 BPU 信息，保持到下一周期向后传递

⑧ flush 与异常处理

外部发出 flush 时，状态机回到 IDLE，清除内部状态

flush_flag 用于等待 dev_rvalid 完成后真正清空

支持处理异常冲刷过程中刚好 miss 的情况

(2) 数据存储器 (dcache)

① 核心结构设计

该缓存采用 2 路组相联结构，通过分层存储和标签匹配实现高效数据访问，核心结构包括：

1) 存储组织

数据存储：2 个数据 RAM（data_ram1 和 data_ram2），每个容量为 128 组×256 位（32 字节 / 块），对应 2 路组相联的两个“路”（way）。每块包含 8 个 32 位字（256 位 = 32 字节），支持按字节粒度更新（通过 wen 字节使能）。

标签存储：2 个标签 RAM（tag_bram1 和 tag_bram2），每个存储 128 组 ×21 位标签，其中：

20 位为地址标签（对应虚拟地址[31:12]）；

1 位为有效位（标记该块是否有效）。

辅助标记：

dirty[127:0][1:0]：每个组的 2 路各 1 个脏位，标记该块是否被 CPU 修改（与主存不一致）。

use_bit[127:0]：替换策略标记，用于选择替换哪一路（推测为最近最少使用 LRU 的简化版）。

2) 地址拆分

虚拟地址（vaddr）被拆分为三部分，用于寻址和匹配：

索引（index）：7 位（vaddr[11:5]），对应 128 个组（2^7=128），用于定位缓存组。

偏移（offset）：3 位（vaddr[4:2]），对应块内 8 个 32 位字（2^3=8），用于从块中选择目标字。

标签（tag）：20 位（vaddr[31:12]），用于与缓存标签匹配，确认是否命中。

② 核心工作流程（状态机驱动）

缓存通过状态机（state）管理从请求到完成的全流程，关键状态及转换逻辑如下：

状态	功能描述
IDLE	空闲状态，等待 CPU 请求。检测命中 / 未命中、非缓存访问或缓存操作，跳转至对应状态。
DIRTY_WRITE	未命中且存在脏块时，将脏块写回主存（确保主存数据一致性）。
ASKMEM	向主存请求数据（未命中且无脏块，或脏块写回后）。
REFILL	主存返回数据后，填充缓存（更新数据和标签），回到空闲状态。
UNCACHE	处理非缓存访问（直接与主存交互，不经过缓存）。
CACOP	处理缓存维护操作（如刷新、清理）。

核心逻辑：

命中（hit1/hit2）：直接读写缓存，写操作标记脏位。

未命中：若对应组有脏块，先写回（DIRTY_WRITE），再请求主存（ASKMEM）并填充（REFILL）。

③ 关键功能设计

1) 命中检测与数据读写

命中判断：通过比较请求地址的标签与两路缓存的标签（ram_tag1_choose/ram_tag2_choose），结合有效位（tag[20]），确定是否命中（hit1/hit2）。

读操作：命中时，根据 offset 从块中提取 32 位字（hit_data_word_choose），输出 rdata 并置位 rdata_valid。

写操作：命中时，根据 wen（字节使能）更新块中对应字节（attach_write_data），并标记对应路的脏位（dirty[index_2][0/1] = 1）。

2) 未命中处理与缓存填充

脏块管理：未命中时，若 dirty[index_2][dirty_index] = 1（存在脏块），需先将脏块通过写总线写回主存（cpu_wen/cpu_waddr/cpu_wdata），写完成后清除脏位。

数据请求与填充：脏块写回后，通过读总线向主存请求数据（cpu_ren/cpu_raddr），主存返回数据（dev_rdata）后，根据 use_bit 选择替换路，更新数据 RAM 和标签 RAM。

3) 非缓存访问（Uncache）

针对不需要缓存的数据（如 I/O 设备），支持直接访问主存：

当 duncache_en 有效时，进入 UNCACHE 状态，通过 uncache_ren/uncache_wen 直接与主存交互，不更新缓存。

4) 缓存一致性维护（CACOP）

支持缓存维护命令，确保缓存与主存数据一致：

清理：标记缓存块为无效（cacop_mode=2'b00），适用于主存数据被外部修改的场景。

写回：将脏块写回主存并清除脏位（cacop_mode=2'b01），确保主存数据最新。

刷新：写回指定脏块并标记为无效（cacop_mode=2'b10），强制缓存与主存同步。

5)　替换策略

采用基于 use_bit 的简化替换策略：

use_bit[index]标记最近使用的路。填充新块时，替换 use_bit 指示的路，并更新 use_bit。

(3) 分支预测器 (bpu)

该分支预测器（BPU）的设计针对双发射超标量处理器，结合了分支目标缓冲区（BTB）和分支历史表（BHT）的核心机制，通过“预测 - 验证 - 更新”的闭环逻辑实现分支指令的高效预测。

针对双发射处理器一次处理两条指令（if_pc1 和 if_pc2）的特点，实现并行预测：

①　跳转方向预测

对两条指令分别判断是否预测跳转：

```
assign pred_taken1 = if_tag1 == tag[index1] & valid[index1] == 1'b1 & history[index1][1] == 1'b1;
assign pred_taken2 = if_tag2 == tag[index2] & valid[index2] == 1'b1 & history[index2][1] == 1'b1;
```

标签匹配（if_tag == tag[index]）：确保条目属于当前指令；

条目有效（valid[index] == 1）：条目存储了有效历史和地址；

历史计数器最高位为 1（history[1] == 1）：2 位计数器的最高位表示“强跳转”或“弱跳转”（计数器值 10 或 11 时，最高位为 1）。

②　目标地址选择

优先选择第一条预测跳转的指令的目标地址，若均不跳转则取下一条指令地址：

```
assign pred_addr = pred_taken1 ? addr[index1] : pred_taken2 ? addr[index2] : if_pc1 + 8;
```

当分支指令进入执行阶段（EX）后，根据实际执行结果（是否跳转、实际目标地址）更新 BTB 和 BHT，确保预测器动态优化。更新逻辑分为三类操作：

① 新增条目（当条目无效且实际跳转时）

触发条件：add1 = ex_valid1 & !valid[ex_index1] & real_taken1（同理 add2）；

操作：初始化条目为 “弱跳转” 状态（history = 2'b10），标记有效（valid = 1），写入标签和实际目标地址（tag = ex_tag，addr = real_addr）。

② 更新历史（当条目有效、标签匹配且为分支指令时）

触发条件：update1 = ex_valid1 & valid[ex_index1] & tag[ex_index1]==ex_tag1 & ex_is_bj_1（同理 update2）；

操作：通过 2 位饱和计数器更新历史（核心逻辑）：

若实际跳转（real_taken=1）：计数器递增（00→01→10→11→11）；

若实际不跳转（real_taken=0）：计数器递减（11→10→01→00→00）。

（饱和计数器的设计避免了单一位预测的抖动，提高稳定性）

③ 替换条目（当条目有效但标签不匹配且实际跳转时）

触发条件：replace1 = ex_valid1 & valid[ex_index1] & real_taken1 & tag[ex_index1]!=ex_tag1（同理 replace2）；

操作：覆盖原条目，更新标签为当前指令标签，重置历史为 “弱跳转”（history=2'b10），写入实际目标地址。

(4) Instbuffer 设计

实例化了两个 FIFO 作为双发射指令缓冲队列，从前端取得的两条指令及其他相关信息分别存入两个 FIFO 中，发射时同时发出。FIFO 中的一条信息包含了：指令 PC、指令的二进制码、分支预测的信息以及例外信息。当队列满时，会暂停前端取指，当队列空时，会向后端发射空指令，以此来保证前后端工作的独立性。

(5) 译码模块设计

译码阶段收取前端传来的两条指令分别进行译码，例化两个 id 模块，指令进入 id 模块后送入 7 个译码单元进行译码，最终通过 valid 信号选取唯一有效的译码单元进行输出。

同时译码阶段设计了一个发射队列，译码完成的指令进入队尾等待发射，若队列满则发出信号使流水线暂停。

(6) 发射(dispatch)模块设计

① 有效性判断与发射控制逻辑

```
assign send_double = (!mem_inst) && (!data_hazard_inst) && (!cnt_inst) && (!privilege_inst) && (&inst_valid);
assign send_en = (send_double == 1'b1) ? 2'b11 : (inst_valid[0] ? 2'b01 : (inst_valid[1] ? 2'b10 : 2'b00));
```

send_double：判断是否双发射

禁止双发射条件：load/store、计数器指令、特权指令、数据冒险、指令无效

send_en：决定实际发射哪条（双发射/只发 1/不发）

② 旁路转发处理（Forwarding）

在下述几个阶段进行旁路优先读数：

WB 阶段旁路

MEM 阶段旁路

EX 阶段旁路

对于每个源寄存器（两条指令各两个），都优先从最近的阶段取数据：

if (wb_pf_write_en && read_en && addr_match) -> wb_data;

else if (mem_pf_write_en && ...) -> mem_data;

else if (ex_pf_write_en && ...) -> ex_data;

else -> from_regfile 或 imm

此部分在流水线中用于消除 RAW 冒险。

③ CACOP 指令识别

识别 ALU_CACOP 类型后，提取目标寄存器来解析其操作类型：

```
cacop_opcode = reg_write_addr
icacop_en = (opcode[2:0] == 3'b000)
dcacop_en = (opcode[2:0] == 3'b001)
```

此处是为 I/D Cache 操作指令做准备。

④ Load-Use Hazard 检测

当前 EX 阶段正在执行 load 指令，且此条 load 的目的寄存器刚好是本周期某条指令的源操作数地址

此时发射暂停，等待 load 指令结果就绪

⑤ 异常标记与异常链传递

每条指令都传递异常原因的四种来源：

pc_exception_cause

instbuffer_exception_cause

decoder_exception_cause

dispatch_exception_cause

⑥ CSR 数据处理

根据是否启用了 csr_read_en 控制信号，判断是否需要发 csr_read_data：

csr_read_data = csr_read_data_i;

同时将最终 csr 地址输出到 EX。

⑦ 寄存器锁存与控制流

若 rst 或 flush 或 dispatch_pause：清空所有发射输出

否则正常发射

若 pause：保持上一个周期的值

(7) 执行(execute)模块设计

该模块例化了两个 ALU 模块分别处理两条指令，同时 ALU 内根据执行指令的类别分为四个单元分别执行一般指令，乘法，除法指令以及跳转指令。乘法 ALU 使用 FPGA 板上资源进行乘法运算；除法 ALU 使用前导 0 算法计算除法在输入延迟一个周期之后，通常约 5 个周期可得出结果。分支跳转 ALU 根据不同的分支跳转指令和预测跳转信息，生成分支刷新信号和实际跳转地址，并输出给前端。同时 ALU 还会处理访存指令，输出读/写使能和对应的地址给 dcache，并且判断访存地址是否对齐，生成例外信号。

(8) 访存(mem)模块设计

访存模块根据执行阶段的指令信息判断接受 Load 指令的访存结果，当前为访存指令且 DCache 返回有效时，访存模块将返回数据记录，传递至写回模块；若 Dcache 未返回有效， 则访存模块会发出暂停请求，等待 DCache 结果的返回。

(9) 控制(cu)模块设计

控制模块控制流水线重定向信号，流水线暂停信号的生成。流水线传递至此的例外信息会被按照优先级统一处理，中断请求会在该阶段被附在指令上，例外处理结果会决定该指令的执行结果是否被提交，以及其结果是否被写入寄存器堆，以此实现例外的精确处理。

三、参考说明

csr 模块的实现参考了 openla500。

后端的设计参考了 2024 年春日影队伍的设计。

参考文献：《超标量处理器设计》。