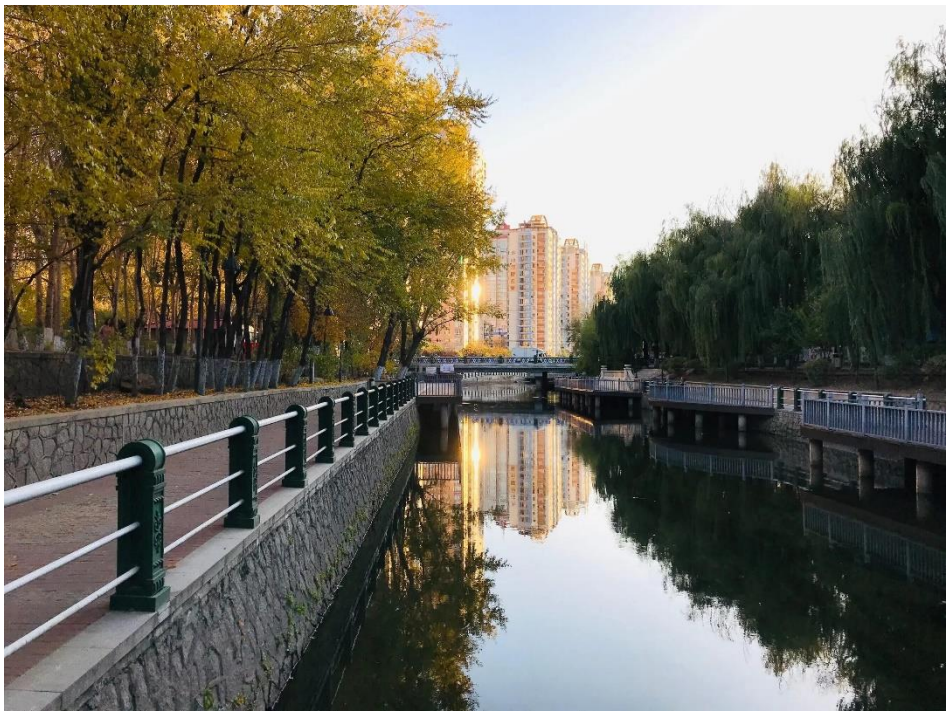
“龙芯杯”第七届全国大学生计算机系统能力培养大赛

哈尔滨工业大学“马家沟河”队

马家沟河项目
决赛设计报告



颜伟业　　耿琛博　　杜鑫　　木超楠

2023 年 8 月

目录

第一章　概述

1.1　项目背景

马家沟河（Ma-River）是第七届”龙芯杯“全国大学生计算机系统能力培养大赛（NSCSCC 2023）的参赛作品。

项目实现了基于 MIPS32 Release1 指令集的多发射乱序处理器软核——Ma-River CPU，并在此基础上自行构建了 SoC，自行设计实现了多个外设控制器，可以运行 U-Boot、ucore、Linux，并为其移植或自行实现了大量丰富应用，实现了一个完整的基于 FPGA 的嵌入式计算机系统。

1.2　项目概述

1.2.1　设计模式

Ma-River 的所有硬件设计使用 Verilog HDL 实现。

在项目前期的结构设计阶段，我们使用 C++ 编写了模拟 CPU 结构的周期精确模拟器，根据其在 benchmark 上的性能表现，进行精确的结构调整，使我们以极低代价得到具有优秀 IPC 的精简结构设计方案。后期直接据此进行 RTL 实现。

在项目后期的调试阶段，我们自行实现了指令自动生成器，可以根据不同场景生成大量随机指令，通过 trace 比对尽可能消除 CPU 核的潜在 bug。

1.2.2　CPU 架构

Ma-River CPU 是六发射超标量乱序处理器，支持 128 条 MIPS32 Release1 指令。主流水线 11 级，取指/提交宽度为 2，后端具有用于指令执行的 4 个整数功能单元和 2 个浮点功能单元，每周期最多发射执行 6 条指令。处理器核在整体上分为整数和浮点两个相对独立的部分，二者共享取指/访存/ROB/提交部件。

Ma-River CPU 正确实现了 CP0、TLB、精确异常等系统支持所需部件，能够支持 Linux 等操作系统的正常启动运行。

1.2.3　SoC 设计

为了充分利用实验箱上的提供的丰富外设、发挥 CPU 核的性能，我们基于 Vivado Block Design 设计了一套功能较完备的 SoC。我们通过自行设计的控制器和已有的 IP 核，能够操作板上的绝大多数功能模块，包括：

- VGA：控制器自行实现；分辨率 1024*768，支持字符和图像模式；支持 DMA，适配了 Framebuffer 驱动
- LCD：控制器自行实现；分辨率 480*800，支持字符和图像模式；支持 DMA，适配了 Framebuffer 驱动
- LCD 触摸：使用 AXI IIC IP 核实现，并使用自行实现的模块进行硬复位

- PS/2：控制器自行实现；支持 PS/2 键盘输入
- GPIO：控制器自行实现；支持以中断方式获取按键状态，支持控制 LED、数码管
- DRAM：使用 Xilinx MIG 7 Series IP 核实现
- 串口：使用 AXI UART16550 IP 核实现
- 以太网：使用 AXI Ethernetlite IP 核实现

同时，我们通过拓展 I/O 接口，可对板外 SD 卡进行读写操作。

1.2.4 系统软件

Ma-River 使用 U-Boot 进行引导，可以启动 ucore 和 Linux（最新的 6.4 版本）操作系统。在 Linux 上，我们自行实现和移植了 VGA Console、LCD、FrameBuffer、键盘等驱动程序，并且还实现了 Linux 终端的中文输入输出支持。

我们为 Linux 移植了 buildroot，通过网口挂载 NFS 文件系统，并在此基础上移植了 GCC、Python、QEMU 等软件，自行编写或移植了 LVGL GUI、Gif 动画播放器、3D 实时渲染器等图形应用。

第二章　CPU

2.1　概述

指令集　Ma-River CPU 实现了 MIPS32 Release1 的一个较完整子集，包括全部的算术逻辑指令、分支跳转指令（包括 likely）、陷阱指令、访存指令、浮点运算指令（CP1）、大部分特权指令（perf 和 sync 实现为 nop），共 128 条。

整体架构　Ma-River CPU 是取指/提交宽度为 2 的六发射超标量乱序处理器，分为整数和浮点（FPU）两个相对独立的部分，这两个部分共享取指部件和 ROB（32 表项）。整数部分的主流水线共 11 级，可大致划分为分支预测、取指（两级）、译码、寄存器重命名、分发、发射、执行、写回 ROB、提交（两级）。执行阶段具有 4 个并行工作的功能单元：ALU0/1（单周期，执行大部分算术逻辑指令）、访存（AGU+LSU，两级流水，执行访存指令）、MDU（多周期，执行乘除指令及其它杂项指令）。FPU 部分具有译码、寄存器重命名、分发、发射、执行（多周期，两个功能单元）、写回 ROB 的流水段，取指后的指令在经过第一个译码阶段后可被送入 FPU，运算结果写回 ROB 后与 CPU 指令统一提交。

CP0 与异常　为完成大赛要求并正确启动 Linux 操作系统，Ma-River CPU 实现了 18 个 CP0 寄存器（包括 Release 2 定义的 EBase），并正确实现了 12 种异常的精确支持。

Cache　Ma-River CPU 实现了 4KB×4 路的 i-Cache 和 4KB×3 路的 d-Cache，均为两级流水访问，并正确实现了 Cache 维护指令。为提升性能，d-Cache 还具备 1 个 Outstanding miss 支持以及修改可动态回滚的特性。

TLB MMU　Ma-River CPU 实现了具有 64 个表项的 TLB，并正确实现了全部 TLB 维护指令及异常，支持 4KB∼256MB 的动态页面大小配置（通过 CP0.PageMask 实现）。我们将 TLB 设计为对体系结构隐藏的两级分立结构，作为缓存的 L1 i-TLB 和 d-TLB 各具有 3 个表项。

分支预测　Ma-River CPU 的分支预测主要采取基于指令局部分支历史的 BTB（256 个表项）+LHT（12 位历史）的方式，此外对于函数返回指令使用容量为 4 的 RAS 硬件栈额外进行预测。

FPU　Ma-River CPU 正确实现了 MIPS32 Release1 定义的全部 30 条支持单精度/双精度/定点数据类型运算的浮点指令。我们自行设计实现了用于浮点加法、乘法、除法、平方根、取整、转换操作的动态多功能流水线运算部件，无需使用专用 IP。

性能　Ma-River 能够以 118MHz 的频率在大赛的性能测试中取得高达 110.50 的性能成绩，相对 GS132 的平均 IPC 比达到 46.71。Linpack 测得浮点算力达 3.23MFLOPS。

2.2 流水线结构

Ma-River CPU 的整体结构见图 2.1，下面对主要流水段进行简要说明：

- **分支预测** 根据当前的一对 PC 预测出下一对 PC 并更新。
- **取指 1** 对 PC 的高 20 位通过 L1 i-TLB 进行地址翻译，使用 PC 的低 12 位读取 i-Cache 每路的对应 tag 和数据。
- **取指 2** 对 i-Cache 的读取结果进行选路，检测 Cache miss/Uncached。并将选出的指令送入指令 FIFO。
- **译码** 从指令 FIFO 中取出指令进行译码（仅考虑需要整数部分处理的指令）。
- **寄存器重命名** 将指令写入 ROB 队尾，查询寄存器状态表，将指令的源/目的操作数寄存器编号替换为对应指令的 ROB 编号，并将需 FPU 处理的指令发射至 FPU。
- **分发** 根据指令类型（ALU、Mem、MDU）将指令存入对应功能单元的保留站中，指令在保留站中等待唤醒。
- **发射** 对于 4 组保留站，分别选出最合适的指令发射至功能单元。
- **执行（ALU）** 对于 ALU 指令，在一个周期内根据操作数值算出结果。
- **访存 1（AGU）** 访存指令执行阶段的第一级流水段，计算出地址并对其高 20 位进行翻译，使用其低 12 位读取 d-Cache 每路的对应 tag 和数据。
- **访存 2（LSU）** 对 d-Cache 的读取结果进行选路，将 store 结果写入 Cache 及回退队列，检测 Cache miss/Uncached。
- **写回 ROB** 将 4 个功能单元的执行结果写回到 ROB 中，并更新保留站中需要它们的指令操作数值。
- **提交 1** 从 ROB 队头取出一对指令，根据指令执行结果生成提交信号。
- **提交 2** 指令提交信号生效，修改处理器状态（更新 GPR、处理异常、清空流水线等），此时指令真正完成执行。

2.3 分支预测阶段

Ma-River CPU 的流水线较长，难以及时精确更新 PC，因此需要依赖分支预测，使得 PC 在第一个流水段就能完成更新，且指令在整个流水线中始终处于推测执行状态，任何时候都可能被抹除，直到其完成提交。

由于分支预测逻辑较为复杂，我们将其和取指分离，单独构成一个流水段。Ma-River CPU 的分支预测策略是基于指令局部历史的，我们取 PC 的低 8 位（去掉最低两位的 0）作为 BTB 和 LHT 的索引，在 BTB 中存储该指令上一次提交时的转移目的地址以及其是否为分支指令，在 LHT 中存储该指令过去 12 次提交时的转移历史（以 12 位数表示）。我们在全局对每一种局部历史模式（共 4096 种）维护一个 2 位饱和计数器，将这条指令的局部历史模式查询该全局表预测出下一次是否转移。

分支指令在执行时可判断转移情况以及分支预测是否失败，在提交阶段更新分支预测器，对于预测失败的情况，令其延迟槽指令提交时通过一个统一信号清空流水线，重置 PC。

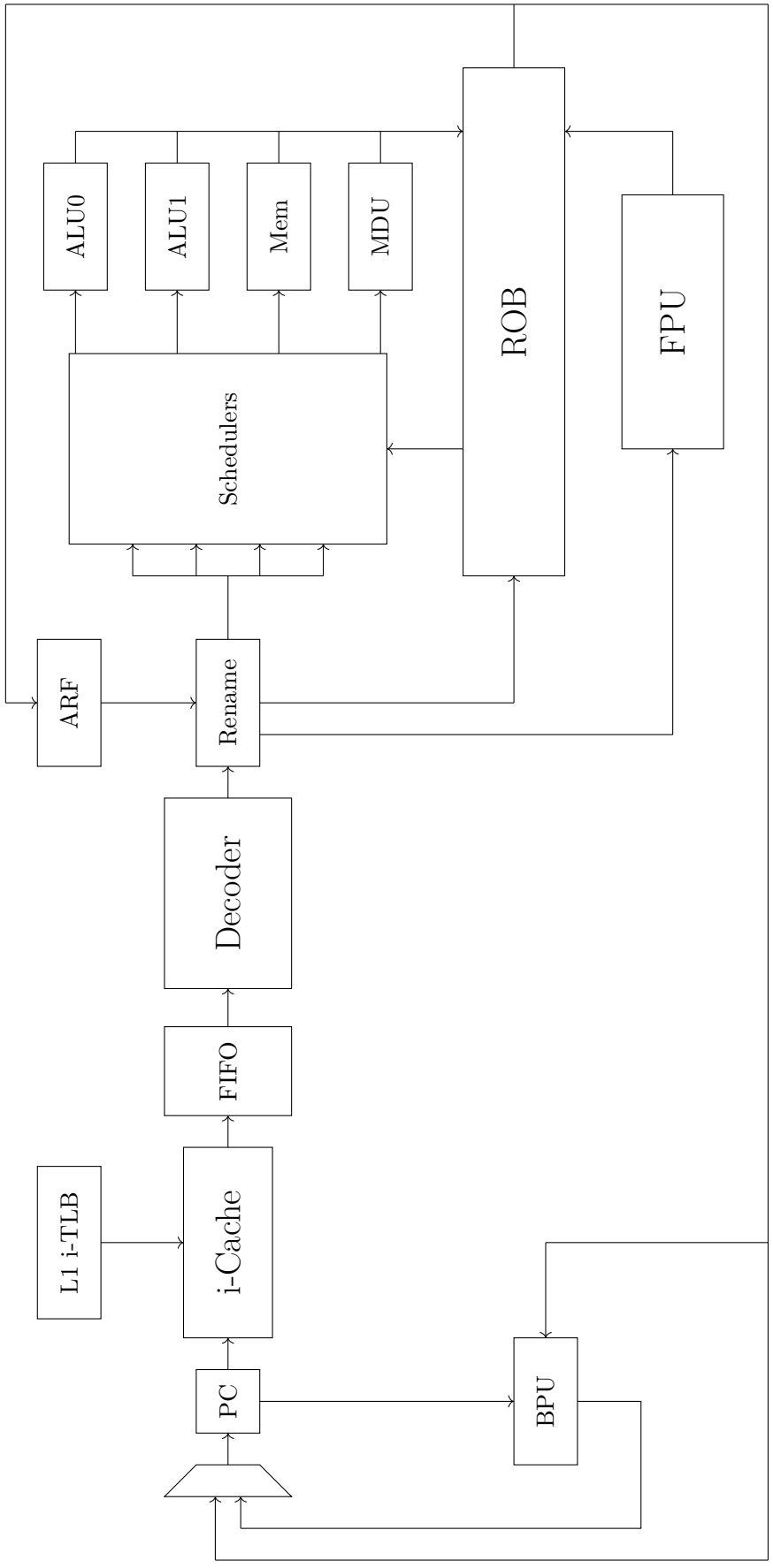


图 2.1: Ma-River CPU 整体结构

此外，我们还额外使用了容量为 4 的 RAS 硬件栈对函数返回指令（jr ra，间接转移，难以直接使用 BTB 预测）的转移目标进行预测。RAS 被实现为一个 64 位数，仅存地址的低 16 位。对于函数调用 jal x，在译码阶段后立刻将返回地址压入 RAS。在预测时，若这个地址索引曾在提交时被标记为 jr ra，则取 RAS 栈顶为目标地址并出栈。我们对于流水线的每一对指令都记录它们分支预测时 RAS 的原值，若它们提交时需要清空流水线，则将此原值重置 RAS（因为 RAS 可能已被推测错误的指令打乱了）。

表 2.1 为大赛性能测试 10 个 benchmark 的分支预测表现。

表 2.1: 分支预测表现

基准程序	预测失败率	IPC
bitcount	6.39%	1.45
bubble_sort	9.41%	1.31
coremark	11.95%	1.04
crc32	2.86%	1.62
dhrystone	5.11%	1.29
quick_sort	16.75%	1.01
select_sort	4.28%	1.53
sha	1.68%	1.60
streamcopy	1.51%	1.24
stringsearch	5.56%	1.25

2.4 取指阶段

在取指阶段，将通过一对指令的地址查询 i-Cache 和 L1 i-TLB，得到对应的一对指令，并处理 i-Cache miss/Uncached。

输入的两个地址一般是连续的，也可不连续。我们规定这两个地址的高 20 位必须是一致的（一定对应同一页面，这样 L1 i-TLB 只需 1 个读端口），否则只能输入第一个地址。它们可能对应不同的 Cache 块，因此需要令 i-Cache 提供双倍的读端口，且具备处理连续两次 miss/uncached 的机制。

Ma-River CPU 的 i-Cache 是 4 路组相联的，每路 4KB，每个 Cache 块 64 字节（AXI3 单次传输的极限），LRU 替换，采取了 VIPT 机制，使用虚拟地址的低 12 位读取每路对应的字，物理地址高 20 位作为 tag。而 Ma-River 支持的页大小最小为 4KB，因此虚拟地址低 12 位一定是与物理地址一致的。Tag 和数据都使用 BRAM 存储。

在取指的第一个流水段，使用地址的高 20 位查询 L1 i-TLB 进行地址翻译，使用地址的低 12 位读取每一路对应的 Tag 和 32 位字。在取指的第二个流水段，使用翻译得到的高 20 位物理地址（Tag）进行选路，得到指令字。若检测到 miss 或者 uncached，则阻塞流水线，启动 AXI 总线交互读取 Cache 块或单个指令字。在读取 Cache 块时若读到请求的指令字则立刻结束等待。

为将取指阶段与后面的阶段解耦，提升取指效率，取出的指令会被立刻放入一个容量为 16 的 FIFO 队列中，等待译码。

2.5 译码阶段

从 FIFO 队列中取出的指令在译码阶段通过两个完全相同的组合逻辑译码器生成指令在整数部分所需的控制信号。需要注意的是，我们在此处仅考虑需要整数部分执行的指令（非浮点指令、浮点访存或 GPR-FPR 传送指令），并不对纯粹的 FPU 指令进行译码。在启用了 FPU 的情况下，若一条指令在译码阶段未被识别，它会在译码阶段后直接被送入 FPU。这简化了译码器的设计。

2.6 寄存器重命名阶段

Ma-River CPU 在整数部分对于 GPR 的寄存器重命名是基于重排序缓冲（ROB）的（这与 FPU（FPR）不同，后文会提及）。在寄存器重命名阶段，当前这一对指令会被送入 ROB 队尾并得到其在 ROB 中的编号，之后这条指令及其目的操作数寄存器将会被直接使用该 ROB 编号进行定位。为简化设计，我们将 ROB 设计为两路奇偶分体的，每周期取出的两条指令必须位于 ROB 两路的相同位置，即便只有一条指令，也要分配两个表项（考虑到这种情况较少，不会造成太大浪费）。

我们使用 GPR 状态表记录每个 GPR 的重命名状态，即是否存在一条未提交的以其为目的操作数的指令。在寄存器重命名阶段，对指令的两个源操作数查询 GPR 状态表，若是这种情况，则将表中记录的那条指令的 ROB 编号作为源操作数的编号，需要那条指令的执行结果作为源操作数值。否则，直接读取指令提交后维护的寄存器文件（ARF）的值作为源操作数值。并且，对于指令的目的操作数，我们还要将当前指令的 ROB 编号记录到 GPR 状态表的对应位置中。

在每周期通过两条指令的情况下，需要额外考虑指令 2 和指令 1 的目的操作数相同或者指令 2 需要使用指令 1 的目的操作数的特殊情况，并且，GPR 状态表和 ARF 都需要 4 个读端口。

另外，对于 nop 指令（译码时被检测出），在这一阶段直接将其存入 ROB 并标记为执行完成，不令其流至执行阶段，它不会与其它指令竞争功能单元，考虑到 MIPS 编译器经常在程序中填充无用的 nop 指令，这一机制对性能是具有积极意义的。

2.7 分发阶段

Ma-River CPU 在整数部分具有 4 个用于执行指令的功能单元，两个 ALU，1 个访存（Mem，AGU+LSU），一个乘除用的 MDU，每个功能单元都具有 1 个缓存指令的保留站，考虑到不同类型指令的比例有所差别，为简化设计，我们令 ALU0/1 的保留站容量为 4，Mem 的保留站容量为 3，MDU 的保留站容量为 2，且每个保留站每周期最多接受/发射一条指令。表 2.2 列出了我们定义的指令功能单元分类。

在分发阶段，这两条指令将根据译码得到的指令类型存入对应的保留站，在此之后指令由顺序转为乱序。我们的分发规则如下：

- 若两条指令均为 ALU 指令，仅当 ALU0/1 保留站均未满才可分别存入 ALU0/1 的保留站，否则都要阻塞等待。
- 若只有一条 ALU 指令，则选一个未满的 ALU 保留站存入。若二者皆未满，将其存入剩余容量最大的保留站中，这样有助于负载均衡，提升 ALU 利用效率。
- 对于非 ALU 指令，当其功能单元的保留站未满时才可存入，否则需阻塞等待。若当前两条指令功能单元相同，则只可分发第一条。

表 2.2: 指令所需功能单元分类

功能单元类型	指令
ALU 指令（单周期）	（除 Mem、MDU 之外的所有需整数部分执行的指令）
Mem 指令（访存）	cache, lb, lbu, lh, lhu, ll, lw, lwl, lwr, sb, sc, sh, sw, swl, swr, ldc1, lwc1, sdc1, swc1
MDU 指令（Hi/Lo 相关、多周期或需要特殊端口的杂项指令）	mfc0, clo, clz, mfhi, mflo, mthi, mtlo, mul, multu, madd, maddu, msub, msubu, div, divu, cfc1, mfc1, movf.fmt, movt.fmt, movn.fmt, movz.fmt, mtc1

- 若一条指令可分发，另一条指令不可，则分发可分发的指令，阻塞流水线。

另外，在将指令存入保留站的过程中，需要尝试从 ROB 中读取未准备好的源操作数。我们对于 ROB 规定队头和队尾必须保持一定间距，这样，指令在提交后，它在 ROB 中记录的执行结果在几个周期后才会被位于寄存器重命名阶段的新指令覆盖，可以保证此时生成源操作数值的指令若已经写回 ROB，无论其是否被提交，一定可以读取到其结果。

2.8 发射阶段

在发射阶段，被唤醒（操作数已准备好或者可以保证在执行前准备好）的指令会与其操作数值将在功能单元未阻塞时被发射到功能单元进行执行。指令唤醒机制将在后文详细描述。

考虑到访存的复杂性，以及需要在 MDU 内部维护 Hi/Lo 的副本（将在后文详细说明），我们令 Mem 和 MDU 的保留站为 FIFO 结构，指令是严格顺序发射的，当最先进入的指令未准备好时，其它指令不可发射。对于两组 ALU 保留站采取乱序发射方式，每次选择最早进入保留站的（需要对保留站中指令维护一个时间戳）且被唤醒的指令进行发射。

2.9 执行阶段

指令进入执行阶段后，其操作数值都应是已知的，此时可以根据不同指令类型计算结果。

2.9.1 ALU

ALU 是单周期的，不会产生阻塞，用于大部分常规指令的执行。Ma-River CPU 的 ALU 具备普通算术逻辑运算功能，以及判断分支转移情况。

2.9.2 MDU

MDU 是多周期非流水的，只能处理一条指令。MDU 处理的指令种类比较杂，我们对其分类进行说明。

- Hi/Lo 维护（mfhi、mflo、mthi、mtlo），为简化设计，我们不令 Hi/Lo 参与寄存器重命名，而是在 MDU 中维护 Hi/Lo 的副本，令指令顺序通过 MDU 并对其直接进行更新或直接使用它们。指令在提交时才更新

体系结构可见的 Hi/Lo（这不如副本更新及时，但是最准确的）。一旦流水线被清空，MDU 持有的 Hi/Lo 副本会被重置为体系结构可见的 Hi/Lo。

- 乘法（mul、mult、multu、madd、maddu、msub、msubu），我们为 Ma-River CPU 自行设计了 3 周期的 32 位乘法器，基于 Wallace Tree。考虑到 Wallace Tree 消耗 LUT 较多，我们还提供了可选的基于片上 DSP 的乘法器（可修改配置参数启用）。此外，具有累加/累减操作的乘法指令需额外一个周期执行加减操作。

- 除法（div、divu），使用我们自行设计的普通 32 位移位除法器，需要 32 个周期。但考虑到大赛性能测试中除法的操作数往往较小，我们在作除法前对操作数前导 0 进行了缩减优化，使得较小数除法能用较少的周期实现。

- clo 和 clz，考虑到数前导 0 所需组合逻辑比较复杂且它们不常使用，这两条指令将在 MDU 中执行 3 个周期。

- mfc0，将其安排在 MDU 而非 ALU 中是为了仅设置一个 CP0 读端口，简化设计。

- GPR 与 FPR/CP1 交互（cfc1, mfc1, movf.fmt, movt.fmt, movn.fmt, movz.fmt, mtc1），它们需要通过专用数据通道和 FPU 交互，故安排在 MDU。具体实现机制将在 FPU 部分描述。

2.9.3 Mem（d-Cache）

Ma-River CPU 的 d-Cache 和 i-Cache 类似，都是 LRU、VIPT、每路 4KB、每块 64B，不同之处在于 d-Cache 是 3 路组相联的。访存单元是两级流水的：

- 第一级流水段（AGU），计算指令的访存虚拟地址，用地址高 20 位在 L1 d-TLB 中查询得到物理地址（Tag），用地址低 12 位读取 d-Cache 每路的 Tag 与数据字。

- 第二级流水段（LSU），使用 Tag 进行选路，得到结果数据字（还需要移位处理）或将结果写入 Cache BRAM 的对应位置，并检测 miss/uncached。

为提升访存性能，我们在设计时采取了两项策略对 d-Cache 的访问进行优化：

- **Outstanding miss** 对于在第二级流水段检测到 miss/uncached 的访存操作，立刻将其移入一个容量为 1 的 buffer 中启动 AXI 交互，在 AXI 交互的过程中（一般需要较长时间），访存流水线不会被阻塞，后续访存操作若 Cache hit 则可正常执行。这对于大赛性能测试中外设写较频繁的 benchmark（例如 Dhrystone）具有较好优化效果。

- **修改动态回滚** 考虑到指令在提交之前始终保持推测执行状态，随时可能会被抹除，对于 Cache hit 的 store 指令，我们令其在第二级流水段直接对 Cache 进行修改（这样无需等待其被提交，但可能是不正确的），并且将其在第一级流水段读取出的 Cache 对应位置的原值存入一个“回滚队列”中。当 store 指令被正确提交时，回滚队列可将队头弹出。当流水线被清空时，趁流水线正在重新建立中（这需要至少 6 个周期），将回滚队列中存的原值倒序写回 Cache BRAM，完成对不正确执行的 store 指令的修改撤销。

需要注意的是，访存指令若 Cache miss/uncached，则其必须在 buffer 中等待前面的指令全部提交完毕才可启动 AXI 交互，即数据访存的 AXI 交互必须是确定的，否则会导致对外设的错误修改。

2.10 写回 ROB 阶段和指令唤醒

指令在对应的功能单元中完成执行后，会将其结果（或异常状态）写回 ROB 并标记为执行完成，4 个功能单元的写回是并行的，它们会写到 4 个不同的 LUTRAM 中，在提交（或进入保留站读取 ROB 时）时正确选取 4 个之一作为结果。这一过程还需顺便将结果传递给保留站中需要它们的指令并唤醒它们，这需要规模较大的数据旁路网络（规模是功能单元数量 × 保留站容量级别的），也是整个处理器的时序瓶颈之处。

为了降低 RAW 相关带来的延迟，让指令尽快得到操作数值从而发射执行，需要尽可能提前对保留站中指令进行唤醒。这意味着，指令被唤醒并不意味着获得了操作数值，但是可以保证其若现在发射一定可以在执行需要时得到操作数值。我们采取了以下提前唤醒策略：

- 对于 ALU0/1 保留站这一周期选出发射的指令（或者正处于执行阶段还未算出结果的指令），令其唤醒两个 ALU 以及 Mem 保留站中的指令。即 ALU（单周期）前一条指令在这周期写回 ROB 的结果至少会立刻被这周期开始在 ALU 中计算结果或进入 Mem 第一级流水段的指令所使用，设置相应的数据传递旁路。
- 对于 load 指令，我们令其位于 Mem 第一个流水段时就试图唤醒 ALU0/1 和 Mem 保留站的指令，这意味着这条指令必须在两周期后顺利得到结果，需要保证：1. 此时位于第二级流水段的操作不被阻塞，2. 该指令一定 Cache hit。对于后者需要在第一级流水段就预测出指令的 cache hit 情况，我们利用访存局部性原理，维护最后访问的两个 Cache 块虚拟地址（当 Cache miss 时将其抹除），在 Mem 第一级流水段直接用虚拟地址匹配它们，若匹配成功（说明其一定 Cache hit）且前一个 Mem 指令未阻塞，则唤醒 ALU/Mem 保留站指令。
- 我们注意到在大赛性能测试的 StreamCopy 中有较为频繁的 “load addr1,Rx; store addr2,Rx” 的指令序列，这种情况下我们可以令 load 指令在发射时直接唤醒 Mem 保留站中使用其 load 结果作为数据字的 store 指令，在 load 指令位于 LSU 时将这周期选路结果直接传递到 AGU 的输入，供后面的 store 指令使用，达到背靠背执行。实测这确实会大幅提高 StreamCopy 的 IPC，但这一数据前递路径时延过高，会使 CPU 频率降低至少 10MHz，最终提交版本弃用了这一策略。

2.11 提交阶段

在提交阶段，检查 ROB 队头的一对指令是否执行完成，若都执行完成（或第一条指令执行完成）则进行至多两条的指令提交，之后指令的执行效果将真正生效，实现对处理器状态的持久性修改（相当一部分特权指令是可被视作在此阶段 “执行” 的）。指令提交主要会作出这些行为：

- 具有写 GPR/FPR/Hi/Lo/FCC 行为的指令将结果持久性地写回 ARF/Hi/Lo/FCC，这是唯一需要考虑两条指令同时提交的情况，也是最普遍的情况，需要对上述结构设置两个并行的写端口。接下来的所有情况都涉及提交一条指令后的流水线清空，无需考虑两条指令同时修改的情况。
- 预测失败的分支指令提交时维护一个标记，在其延迟槽提交时，若此标记有效，则清空流水线，将 PC 重置为正确的分支目标地址。并且无论预测是否失败，都要对分支预测器作出修改。
- mtc0 指令或其它具有修改 CP0 行为的指令（tlb 维护等）提交时将直接修改 CP0 寄存器组并清空流水线从下一条指令重新开始执行。ctc1 对 CP1 控制寄存器的修改也如此处理。
- 检测指令异常以及此时的中断请求，若存在需要处理的中断/异常则计算目的地址、修改相应 CP0 寄存器并清空流水线实现跳转。

- 对于 L1 TLB 缺失的指令，尝试查询 L2 TLB，若查询到则清空流水线重新执行该指令，否则按 TLB 异常处理。
- Cache/TLB 维护指令对 Cache 和 TLB 作出相应修改。
- 对于 likely 分支指令直接清空流水线实现跳转（我们并不对 likely 分支指令进行预测，只提供最基本的实现）。
- 对于 movn/movz 指令，若其实际不写回则清空流水线重新开始执行下一条指令，这是因为它的不确定写回行为与我们的寄存器重命名机制违背，为简化设计我们在寄存器重命名时假定其一定写回。
- ll 和 eret 指令提交时修改 llbit，并清空流水线，这样我们只需要对整条流水线维护一个全局的 llbit，令 sc 指令在译码时就选择性处理为普通的写 0 指令或真的 store 指令。
- 对于 wait 指令，提交时清空流水线，设置一个阻塞标志，在下一次中断发生前阻止流水线的重新建立。

可见，Ma-River CPU 在设计上采取的是一种“遇事不决，清空流水线”的原则，很多特殊指令的特殊行为只需要放到提交阶段处理，这使得 CP0 寄存器等复杂结构的 RAW 冒险得到较为保守稳妥的解决，尽管这样带来了流水线重新建立的时间开销，但考虑到这些情况在程序正常运行时较少发生，并且能够大大简化设计的复杂程度，因此我们认为这是值得的。

由于提交阶段作出的行为较为复杂，且涉及到整个处理器的状态修改，我们将提交阶段拆分为两个流水段，第一级流水段取 ROB 队头并根据指令结果生成待生效的信号，第二级流水段将信号传递到处理器其它部件上生效。

2.12 中断、异常与 CP0 寄存器组

Ma-River CPU 支持 MIPS32 定义的 6 个硬件中断和 2 个软件中断，在无异常的指令提交时，若检测到 CP0.Cause 有未屏蔽的中断请求且此时中断使能开启，则清空流水线并重置 PC 为中断处理程序入口地址。

Ma-River CPU 支持精确异常，指令在流水线的取指、译码、执行阶段都会发生异常，需要让指令正常地流动下去，将异常信息写入 ROB，在指令提交时才处理其引发的异常。支持的异常如表 2.3 所示，覆盖了大赛功能测试及操作系统支持所需的所有异常。

Ma-River CPU 实现了 18 个 CP0 寄存器，它们及实现的字段如表 2.4 所示，覆盖了大赛功能测试及操作系统支持所需。

2.13 TLB MMU

Ma-River CPU 实现了具有 64 个表项（MIPS32 定义的上限）的全相联 TLB，用于实现虚拟地址到物理地址的快速翻译，并正确实现了全部 TLB 维护指令和异常以支持操作系统的内存管理。TLB 维护在指令提交时进行，为保守起见，这必定伴随着流水线的清空。

由于查询全相联 TLB 的时间代价较高，为了消除 TLB 带来的时序瓶颈，我们将 TLB 拆为对体系结构隐藏的两级分立结构，L1 d-TLB 和 i-TLB 作为整个 TLB 的容量为 3 表项的缓存，在取指/访存的第一个流水段会使用虚拟地址高 20 位匹配 L1 TLB 的 3 个表项的 VPN，这一查询的组合逻辑代价是较低的。若未匹配到表项，我们并不立刻阻塞流水线进行 L2 TLB 的查询，而是暂时将这条指令作为“TLB 缺失异常”处理，令其流动到提交阶段。在提交阶段对于 TLB 缺失异常并不直接进行跳转，而是先阻塞提交，在 L2 TLB 中进行

表 2.3: Ma-River CPU 支持的异常

异常	说明
Int	软/硬件中断
Mod	TLB 修改，当 store 指令作用于一个 D=0 的虚拟页时
TLBL	TLB Load 无效或缺失
TLBS	TLB Store 无效或缺失
AdEL	取指或 load 指令引起的地址错误（可能由于权限错误引起）
AdES	store 指令引起的地址错误
Sys	syscall 指令引起的系统调用
RI	未识别的保留指令（若启用 FPU，RI 是由 FPU 二次译码阶段负责检测的）
CpU	协处理器不可用（在用户态下使用特权指令访问 CP0，或在未启用 FPU 时执行浮点指令）
Ov	加法溢出
Tr	陷阱
FPE	浮点异常

表 2.4: Ma-River CPU 实现的 CP0 寄存器及字段

CP0 寄存器	实现的字段	说明
Index	全部	用于维护 TLB 时索引表项
Random	全部	用于随机替换 TLB 表项
EntryLo0	全部	用于读写 TLB 表项
EntryLo1	全部	用于读写 TLB 表项
Context	全部	用于 TLB 异常时快速定位页表项地址
PageMask	除 MaskX 外	指定页大小（支持 4KB-256MB 的所有合法取值）
Wired	全部	用于随机替换 TLB 表项
BadVaddr	全部	发生异常时记录引发异常的虚拟地址
Count	全部	用于时钟中断的计数器
EntryHi	除 VPN2X 外	用于读写 TLB 表项及维护 ASID
Compare	全部	用于设置时钟中断
Status	IE, EXL, KSU, IM, BEV, CU0, CU1	设置处理器关键状态
Cause	ExcCode, IP, IV, CE, TI, BD	发生异常时记录异常相关信息
EPC	全部	发生异常时记录指令地址
PRId	全部	固定为 0x00018000，告知操作系统本处理器具有和 MIPS 4Kc 类似的特性
EBase	全部	设置异常基地址（MIPS Release2 特性，但操作系统支持所需）
Config0	除 K23 和 KU 外	处理器基本信息以及设置 KSeg0 是否 Cached
Config1	全部	处理器 Cache、FPU 和其它部件的基本参数信息

一个 8 周期的查询，若查询到则不抛出异常，将该表项装入对应的 L1 TLB 进行替换（L1 TLB 的替换策略为 FIFO），并清空流水线，重新执行这条指令。若在 L1 TLB 中也查询不到则正常抛出异常即可。对于 TLB 修改指令（TLBWR/TLBWI），它们不仅要修改 L2 TLB 的对应位置，若对应位置原表项在 L1 TLB 中，也要写到 L1 TLB 的对应位置，这样两级 TLB 不会对正常的 TLB 维护造成任何影响，软件不会察觉，并且在保持较低缺页异常率的前提下（一次真正缺页异常的执行时间远高于 L2 TLB 的查询）实现了访存的快速地址翻译。

此外，我们还正确实现了 CP0.PageMask 中从 4KB-256MB 的所有合法的页大小取值，在 TLB 查询时会根据 PageMask 的取值进行地址匹配，这使得操作系统可以灵活选择页大小，提升内存管理效率。

2.14 FPU

Ma-River CPU 具有一个功能完备的 FPU，正确实现了 MIPS32 Release1 定义的全部 30 条支持单精度/双精度/定点数据类型运算的浮点指令，运算均符合 IEEE754 规范。在我们最终运行 Linux 的 SoC 上，FPU 耗费约 6k 个 LUT 和 FF，占整个 CPU 核的 23%。若不需要浮点运算，我们提供了对应的配置选项使得 FPU 不会被综合，以换取更多片上资源。

2.14.1 FPU 整体结构

Ma-River CPU 的 FPU 整体结构如图 2 所示，它的架构类似于整数部分，也具有译码、寄存器重命名、分发、发射、执行、写回 ROB 的阶段。

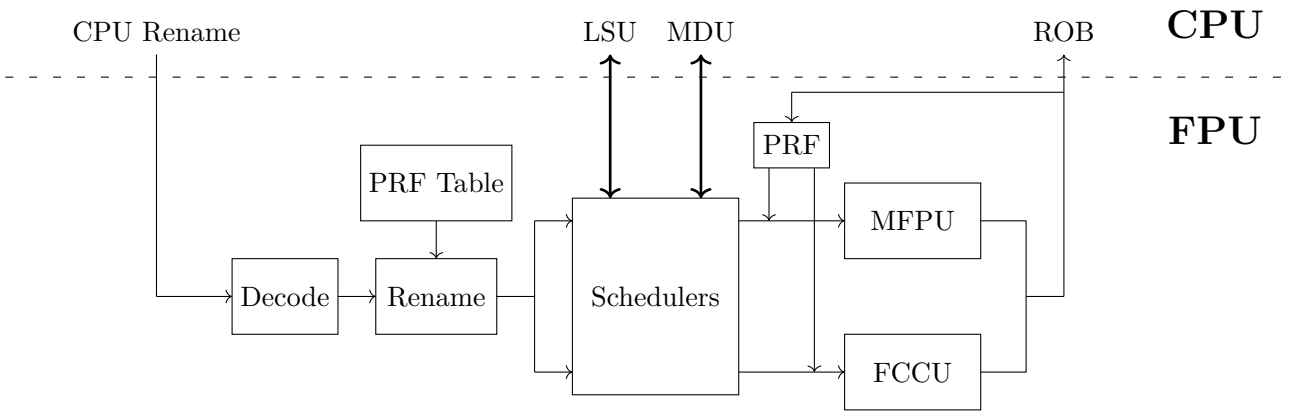


图 2.2: FPU 整体结构

FPU 部分没有自己的取指、访存和提交部件，这些是依赖于整数部分的。当指令通过整数部分的译码阶段后，若其需要在 FPU 部分被执行，或根本未被识别（整数部分的译码器仅对少数依赖整数部分的浮点指令进行译码，大部分浮点指令在 FPU 内译码），则在下一周期被发射到 FPU 中（此时其也在 ROB 中被分配表项，这是与整数部分共用的），FPU 每周期仅接受一条指令。当浮点指令执行结束后，会将结果写回 ROB，和其它指令一样被正常提交。

FPU 在执行阶段具有两个功能单元，一个用于执行大部分浮点运算（称为 MFPU），另一个用于执行涉及浮点条件码（FCC）的指令（称为 FCCU），例如浮点比较、条件传送和条件分支。二者的保留站容量皆为 4，MFPU 的保留站是乱序发射的，这和 ALU 是完全一致的。FCCU 的保留站是严格顺序发射的 FIFO 结构，这个思路和 MDU 一致，令 FCCU 持有 FCC 的临时版本，顺序通过 FCCU 的指令可直接修改或使用 FCCU 持有

的 FCC，并在提交时对 FCC 作出持久性修改，清空流水线时将 FCCU 持有的 FCC 重置为提交的版本。MFPU 和 FCCU 各自负责执行的指令如表 2.5 所示。

表 2.5: 浮点指令所需功能单元分类

功能单元类型	指令
MFPU	abs.fmt, add.fmt, ceil.fmt, cvt.d.fmt, cvt.s.fmt, cvt.w.fmt, div.fmt, floor.w.fmt, ldc1, lwc1, mfc1, mov.fmt, movn.fmt, movz.fmt, mtc1, mul.fmt, neg.fmt, round.fmt, sdc1, sqrt.fmt, sub.fmt, swc1, trunc.fmt
FCCU	bc1f, bc1t, c.cond.fmt, movf, movf.fmt, movt, movt.fmt

2.14.2 浮点寄存器（FPR）及其重命名

MIPS32 定义了 FPR[0]-FPR[31] 共 32 个 32 位的浮点寄存器，为提供双精度支持，FPR[2i] 和 FPR[2i+1] 可映射为一个 64 位浮点寄存器使用，浮点指令的操作数即为 32 位或 64 位的浮点寄存器。因此，Ma-River 的 FPU 对于浮点寄存器采取了奇偶分体的策略，源操作数和目的操作数都仅保留寄存器编号高 4 位，使用信号和写回信号都分为奇偶两组，这样简化了双精度浮点运算指令的“使用 4 个 FPR，写回 2 个 FPR”的复杂行为。

Ma-River CPU 在 FPU 部分的寄存器重命名采取基于物理寄存器（PRF）的策略，这和整数部分使用 ROB 不同，并不对 FPR 设置真正的寄存器文件，其提交值和未提交值都在 PRF 中。PRF 也是奇偶分体的，总共 64 个 32 位寄存器，奇 FPR 使用奇 PRF，偶 FPR 使用偶 PRF。我们记录每个 PRF 的状态及 FPR 对应的 PRF，在 FPU 的寄存器重命名阶段为目的操作数分配一个空闲的 PRF，并且查询源操作数的 PRF 编号。指令执行结束后将结果写到 PRF 中，发射时直接读取 PRF 值（由于 PRF 也保存提交后的值，此时一定是有效的），指令提交时，其写回的 PRF 值变为提交值，令目的操作数对应的上一个 PRF 空闲（此时不可能有使用它的指令了）。当流水线被清空时，未被提交的 PRF 将变为空闲状态。

2.14.3 浮点控制寄存器（FCR）

FCR 对于 FPU 犹如 CP0 寄存器对于 CPU，它们设置 FPU 的一般行为并反映 FPU 的状态。Ma-River CPU 实现了 MIPS32 定义的全部 5 个 FCR，如表 2.6 所示，可以通过 cfc1/ctc1 指令读写它们。

表 2.6: Ma-River CPU 支持的 FCR

FCR	说明
FIR	描述 FPU 实现的特性
FCSR	描述 FPU 的条件和异常状态，设置舍入/规格化规则
FCCR	即浮点条件码 FCC
FEXR	浮点异常位
FENR	异常屏蔽位，设置舍入/规格化规则

2.14.4 浮点异常

MIPS32（IEEE754）针对浮点运算中的一些特殊情况定义了如下所示的 6 种浮点异常，它们都对应于 CPU 的 FPE 异常，发生 FPE 时由 FCSR.Cause 指示异常原因。部分浮点异常是可屏蔽的，若被屏蔽则将结果置为特殊值（0、∞、NaN）。Ma-River CPU 正确实现了它们。

- **未实现的运算** 由于 Ma-River 实现了 MIPS32 Release1 定义的所有单/双精度和定点数运算，在指令被识别的情况下不会引发此异常。未被识别的指令引发保留指令异常。
- **非法运算** 运算操作数中有 NaN，或者执行了 $\frac{0}{0}$、$\frac{\infty}{\infty}$、对负数开根这样的结果未定义操作。若其被屏蔽则结果置为 NaN。
- **除 0** 当除法运算的分子非 0 且分母为 0。若其被屏蔽则结果置为 ∞。
- **上溢** 结果绝对值超出单/双精度的最大表示范围，若其被屏蔽则结果置为 ∞。
- **下溢** 当结果为非 0 非规格化数时。若其被屏蔽则按舍入规则处理。
- **结果不精确** 产生上溢或下溢时，或舍入导致结果不精确。若其被屏蔽则按舍入规则处理。

除了标准的浮点异常以外，Ma-River CPU 的 FPU 还可能会引发保留指令异常（在 FPU 被实现的情况下，保留指令是 FPU 的译码阶段负责检测的）和协处理器不可用异常（Status.CU1 置 0 但执行了浮点指令，一般是操作系统未为当前线程启用 FPU）。无论引发哪种异常，FPU 都会写回 ROB，在提交时按照 CPU 异常机制正常处理。

2.14.5 FPU 与 CPU 的数据交互

FPU 的访存指令需要使用整数部分的访存单元，此外还有一些指令具有在 FPR 和 GPR 间传送数据或者同时使用 GPR 和 FPR 的行为，无论如何，它们需要在整数部分的译码阶段就被识别，并且在此之后既需要发射到 FPU，又需要继续在整数部分的流水线中执行，即一条指令拆为两个微指令执行，这两个微指令（根据同一个 ROB 编号确定）需要在两个部分中“内外接应”，实现数据的交互。我们据此把浮点指令作出了如下分类：

- **不需 FPR**（cfc1, ctc1） 它们只读写 CP1 控制寄存器，作为正常 CPU 指令执行即可，处理方式同 mfc0/mtc0。
- **需要单个 FPR 值（或 FCC），写回 GPR**（movf, movt, mfc1） 我们将这些指令的整数部分微指令安排在 MDU 中执行，在整数部分设置一个以 ROB 编号寻址的 Buffer，MDU 保留站中需要 FPR 值的指令要等到 Buffer 对应位置有效才可发射。当 MFPU/FCCU 保留站的微指令被发射时，此时必然得到 FPR 操作数值（或者在 FCCU 执行指令时得到 FCC 值），将其通过通道传送到整数部分的 Buffer 中并尝试唤醒 MDU 保留站中的指令。这样，MDU 保留站发射指令时即可得到所需的 FPR 值，无论两个微指令在两条流水线中相对位置是什么样的。
- **需要单个 GPR 值，写回 FPR**（movn.fmt, movz.fmt, mtc1） 和上述情形相反，我们同样地设置一个 MDU 保留站到 MFPU 的数据通道与 Buffer 即可。
- **FPR 访存**（ldc1, lwc1, sdc1, swc1） 和上述情形类似，只不过换成了 Mem 保留站以及 LSU 的写回 ROB 阶段与 MFPU 的双向数据传输，需要设置相应的 Buffer。此外，ldc1 和 sdc1 是 64 位的访存指令，我们可在 Mem 保留站发射它们时将其进一步拆为两个连续发射的 32 位访存微指令。
- **仅使用 FPR** 剩余所有浮点指令，无需和整数部分交互。

2.14.6 浮点运算及其执行部件

表 2.7 列出了 MIPS32 定义的主要的浮点运算类型及 Ma-River 中的执行延迟（从发射到写回 ROB 的周期数），Ma-River CPU 正确地实现了它们，使用我们自行实现的运算部件（主要使用 MFPU，仅有浮点比较在 FCCU 中实现），无需额外 IP 核。由于浮点运算较为复杂，为避免影响 CPU 频率，我们采取了较为保守的长流水线设计，流水段拆分较细，执行延迟略高。

表 2.7: Ma-River CPU 实现的浮点运算类型

浮点运算类型	指令	执行延迟
加减	add.fmt, sub.fmt	12
变号	abs.fmt, neg.fmt	1
乘法	mul.fmt	13
除法	div.fmt	60
平方根	sqrt.fmt	60
比较	c.cond.fmt	3
转换为浮点（精度转换）	cvt.s, cvt.d	10
转换为定点（取整）	cvt.w, ceil.fmt, floor.fmt, round.fmt, trunc.fmt	6

负责执行大部分运算的 MFPU 结构如图 3 所示。MFPU 采用单入单出的动态多功能流水线结构，除了除法和开根以外，所有运算部件都是完全流水化的，不同类型运算可重叠执行，但共用一部分流水线（例如舍入和规格化），这节省了部件。共用输出端的不同部件若同时输出，需按照复杂运算优先的原则进行仲裁，阻塞其它部件。

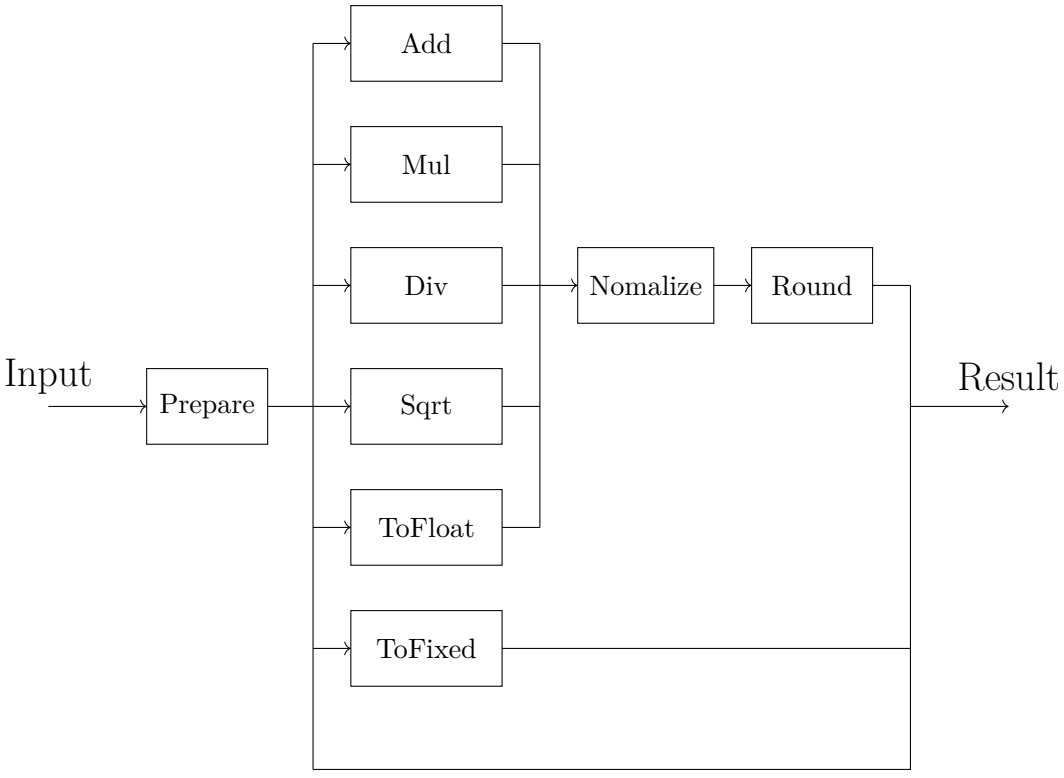


图 2.3: MFPU 结构

- **准备阶段** 对于单/双精度浮点类型的操作数，将其统一按双精度处理（这样无需再设计不同精度的运算部件），提取指数和尾数，判断 0、无穷和 NaN。对于一些简单的变号（abs、neg）或传送指令，在此阶段后可直接输出结果。
- **加法** 浮点加法器有 3 个流水段，第一个流水段进行指数减法，第二个流水段进行尾数对齐，第三个流水段进行尾数加法。
- **乘法** 浮点乘法器有 4 个流水段，同时进行 53 位的尾数乘法和指数加法，尾数乘法同样可通过配置选项选用基于 LUT 的 Wallace Tree 或基于 DSP 的分段乘法，由于 53 位乘法的 Wallace Tree 消耗巨量的 LUT，我们最终的提交版本是选用 DSP 的。
- **除法** 浮点除法器是非流水的，尾数使用朴素的 53 位移位除法。
- **平方根** 浮点平方根使用快速移位算法对尾数（需将指数调整为偶数再除 2）进行开根，过程和时间开销类似于除法。
- **定点转浮点** 定点数（整数）到浮点数转换有 2 个流水段，第一个流水段计算指数，第二个流水段移位得到尾数。
- **浮点转定点** 浮点数到定点数转换有 3 个流水段，第一个流水段进行尾数移位，第二个流水段进行舍入，第三个流水段进行取负和无效值检测。
- **规格化阶段** 规格化部件接受上述生成浮点数结果运算部件产生的符号、指数和 108 位尾数（此时还是中间状态），将尾数移位为 1.xxx 的规格化形式（即便结果实际上是非规格化数）。这一阶段有 3 个流水段，前 2 个流水段计算尾数前导 0 个数，第三个流水段对尾数进行移位。
- **舍入阶段** 舍入部件根据 FCSR.RM 指示的舍入规则，生成真正的单/双精度结果。这一阶段有 3 个流水段，第一个流水段计算真正的指数值，第二个流水段进行尾数舍入，第三个流水段进行结果调整并判断上下溢。

2.15 外部接口

Ma-River CPU 核对外仅通过一个用于访存的 AXI3 总线接口和 6 根高电平有效的中断相连。

取指部件和访存部件都具有自己的 AXI 接口，二者在内部通过一个简单的仲裁器合并为一条对外连接的总线。

2.16 性能

在大赛的性能测试中，Ma-River CPU 的频率最高可达 118MHz。在我们最终运行 Linux 的 SoC 上，我们将 CPU 和 SoC 频率皆设置为 100MHz。

Ma-River CPU 在大赛性能测试 10 个 benchmark 的表现如表 2.8 所示，相对 GS132 的平均加速比高达 110.50，平均 IPC 比为 46.71，平均 IPC 为 1.32。

此外，我们还在 Linux 环境下（CPU 和 SoC 频率均为 100MHz）对 Ma-River CPU 的浮点性能进行了测试，Linpack 测试结果为 3.23MFLOPS。

表 2.8: 性能测试表现

测试程序	**加速比**	**IPC**	**IPC 比值**
bitcount	112.54	1.45	47.06
bubble_sort	119.29	1.31	50.47
coremark	85.00	1.04	35.96
crc32	132.02	1.62	55.87
dhrystone	106.71	1.29	45.15
quick_sort	83.45	1.01	35.31
select_sort	118.81	1.53	50.27
sha	133.29	1.60	56.40
streamcopy	116.60	1.24	49.42
stringsearch	109.58	1.25	46.37
几何平均	110.50	1.32	46.71

第三章　SoC 与外设

3.1　概述

我们致力于构建一套能提供**真正 PC 机体验**的 SoC 与外设系统。目前我们已驱动 VGA、LCD、PS/2、GPIO、触摸、DDR、串口、以太网等一系列外设，初步达成了这一目标。其中，VGA、LCD、PS/2、GPIO 的控制器和对应驱动为团队自行实现。

实机效果如图 3.1 所示。

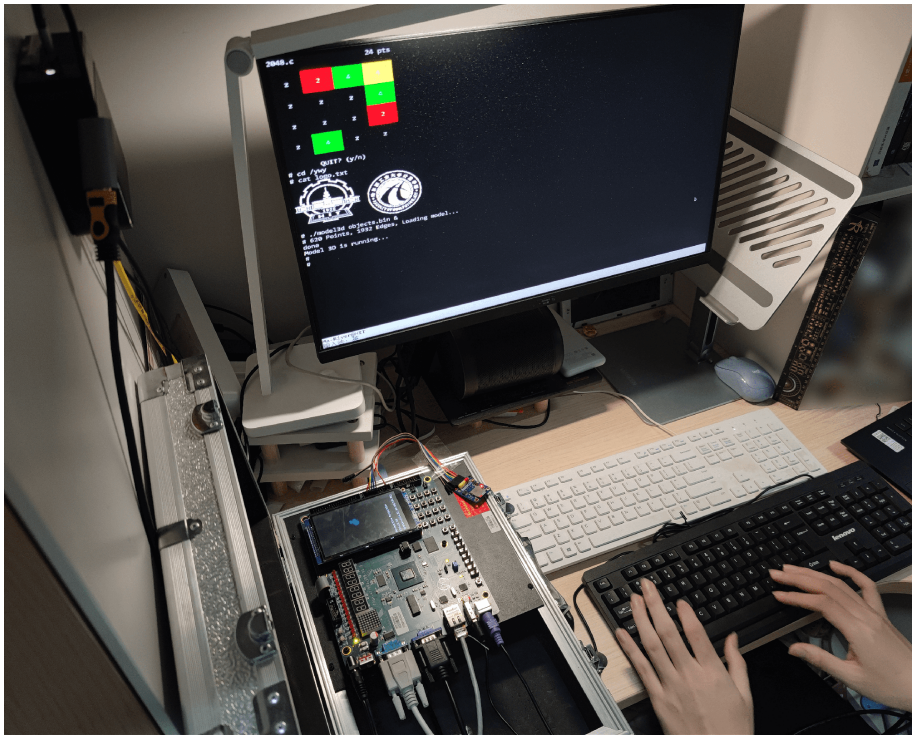


图 3.1: 实机效果展示

3.2 整体架构

SoC 整体架构如下图所示：

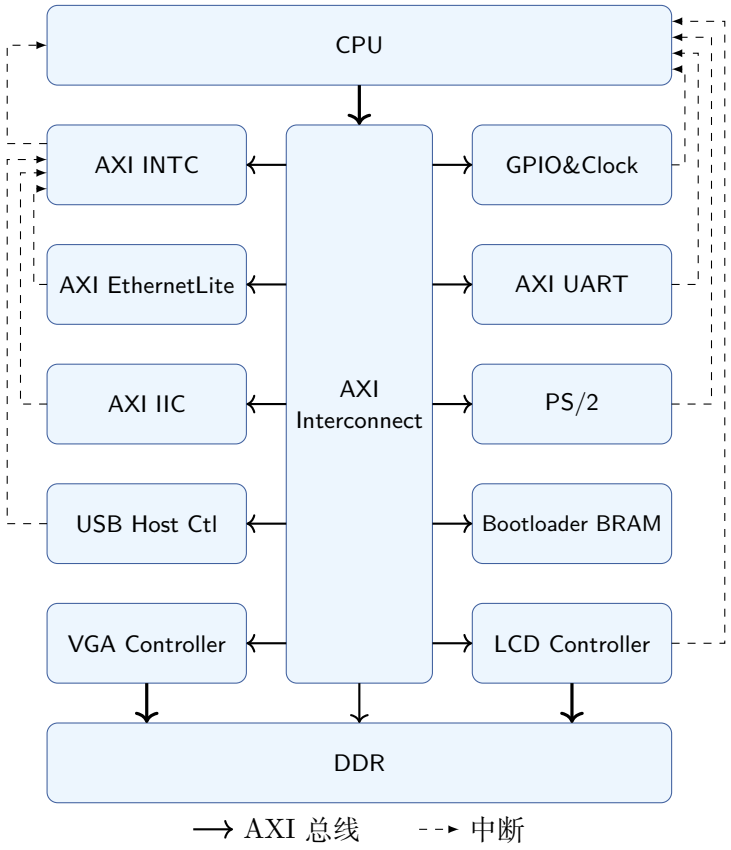


图 3.2: SoC 整体架构

3.3 资源分配

3.3.1 外设地址

表 3.1: 设备物理地址分配

设备名称	起始地址	设备名称	起始地址
DDR3	0x0000_0000	VGA 控制器	0x1FE0_0000
AXI IIC(触摸)	0x1FA0_0000	AXI UART16550	0x1FE4_0000
AXI INTC	0x1FB0_0000	AXI QUAD SPI	0x1FE8_0000
Bootloader ROM	0x1FC0_0000	AXI Ethernetlite	0x1FF0_0000
LCD 控制器	0x1FD0_0000		

3.3.2 外设中断

《MIPS 指令系统规范》[1]中要求实现 6 个硬件中断，但计时器中断会复用 HW5 硬件中断，故可以直接使用的硬件中断只有 5 个，且仅支持电平触发中断；为了拓展中断数量、处理部分边沿触发的中断，我们使用了 Xilinx AXI INTC IP 核。

中断的连接关系如下图所示，其中 AXI Ethernetlite 为边沿触发中断，其他均为电平触发中断。

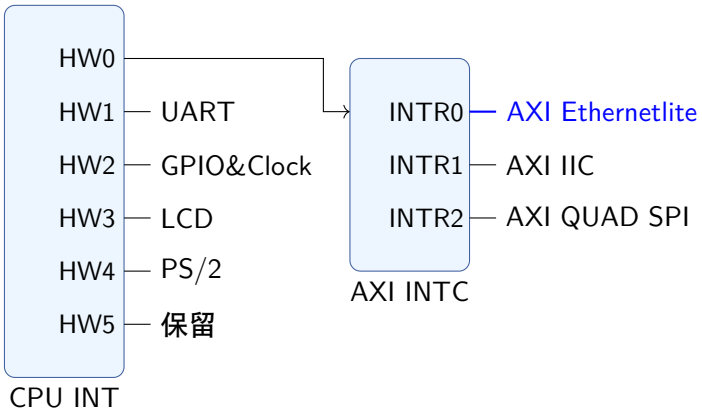


图 3.3: 中断的连接关系

3.4 外设说明

3.4.1 VGA

实验板提供了标准 VGA 接口，我们为此自行设计并实现了具有文本和图像两种工作模式的 VGA 控制器，其分辨率和刷新率固定为 1024×768@60Hz，位深度为 4，能实现文本或图像的输出。

在文本模式下，VGA 控制器按照 8×16 大小为一个格子，将屏幕划分为 48 行 128 列，每个格子可显示一个单色 ASCII 字符或半个汉字（两个格子显示一个汉字）。VGA 控制器为 ASCII 字符内置了点阵字库，只需要通过 AXI 总线向对应地址写入颜色信息和字符的 ASCII 码即可显示对应的字符。为支持汉字等高级字符集，文本模式支持向格子中写入基于位的二值化图像信息（每个像素用一个位表示），只要在软件字库中添加自定义的点阵信息，即可显示任何语言的字符。此外，文本模式还具有可配置的闪烁光标。文本模式的全部图像信息直接使用片上 BRAM 存储，不需要消耗内存带宽，性能友好。

在图像模式下，无法直接在片上存储的图像数据只能通过 DMA 方式传输。VGA 控制器会周期性地通过 AXI 总线从 DDR 中读取指定地址处的图像数据，显存地址和图像的显示区域由软件动态设置。基于图像模式和 DMA 机制，我们实现并移植了 FrameBuffer 驱动，能够实现虚拟终端、动画绘制、GUI 应用等。

我们为 U-Boot、ucore 和 Linux 都移植了基于 VGA 控制器文本模式的 Console 驱动程序，使得用户可以直接使用 VGA 与系统进行交互。

3.4.2 LCD 及触摸

实验板提供了分辨率为 480*800 的 LCD 屏幕，我们为其自行设计实现了 LCD 控制器实现文本与图像输出。LCD 控制器也具有文本模式和图像模式。

[1]参见龙芯杯 2023 发布包文档：《A03 “系统能力培养大赛” MIPS 指令系统规范 v1.01》

LCD 控制器的文本模式和 VGA 类似，故不再赘述。原则上系统 Console 也可使用 LCD 文本模式输出，但由于 LCD 屏幕太小，我们最终并未实现。

由于 LCD 本身具有显存，LCD 的图像模式支持直接向屏幕某位置写入像素值，这会在单个像素上花费较多周期，CPU 需要进行多次高代价 uncached 访存。为加速 LCD 图像绘制，我们也为 LCD 提供了 DMA 传输支持。软件可将像素数据准备在 DDR 中，设置 LCD 控制器的传输地址和范围即可启动单次 DMA 传输，LCD 控制器会连续向显存中写入像素，传输完成后通过中断告知 CPU。这使得我们基于 LCD 的动画应用可以以至少 20 30FPS 的帧率播放。基于其 DMA 机制，我们也实现并移植了和 VGA 类似的标准 FrameBuffer 驱动。

对于 LCD 的触摸功能，我们使用 Xilinx AXI IIC IP 核与 LCD 上的 GT1151Q 触摸芯片进行 I2C 通信，定时查询触摸状态，读取触摸坐标。

3.4.3 PS/2 键盘

实验板提供了 PS/2 物理接口，我们为此自行设计并实现了简单的 PS/2 键盘控制器。它能够解析输入的 PS/2 物理信号，在按键按下或松开时发起键盘中断，读取来自键盘的扫描码。我们为键盘分配了单独的中断信号并将其连接到 CPU 的外部中断信号上，CPU 可以通过 AXI 总线读取扫描码并响应中断。同时，我们使用了深度为 4 的 FIFO 对多个未读取的键盘扫描码进行缓冲。此外，对于不使用中断方式输入的系统（如 U-Boot），可使用轮询方式读取扫描码。

我们为 PS/2 键盘控制器编写了适配的驱动，结合 VGA 能达到实际 PC 机的体验。

3.4.4 GPIO

实验板上提供了 LED、数码管、拨码开关、按键等 GPIO 设备，我们为此自行设计并实现了 GPIO 控制器以对它们进行读写以及可编程的中断控制。

3.4.5 串口

实验板提供了 RS-232 串行通信接口，我们使用官方提供的 Xilinx UART16550 进行驱动。该控制器的中断信号为电平触发，我们将其直接连接到 CPU 的外部中断信号上。

U-Boot 和 Linux 源码中提供了相关驱动，可以直接使用。

3.4.6 以太网

实验板提供了以太网标准协议中的 MDIO 和 MII 接口。我们使用官方提供的 Xilinx EthernetLite 进行驱动。由于该 IP 仅提供上升沿触发的中断，我们使用中断控制器来转换中断类型。

U-Boot 和 Linux 源码中提供了相关驱动，可以直接使用。

3.4.7 SD 卡

我们通过实验板提供的拓展 GPIO，连接了提供 SPI 接口的 SD 卡模块，故我们使用 AXI QUAD SPI 进行交互。

Linux 源码中提供了相关驱动，可以直接使用。

第四章　系统软件

4.1　U-boot

4.1.1　背景

U-Boot 是一个启动引导程序，常见于嵌入式系统中，用于引导 Linux 等操作系统。在本系统的设计中，U-Boot 将作为引导程序，放置在 BRAM 中；同时，我们根据展示需求，修改了 U-boot 源码以增加更多功能。

4.1.2　移植内容

- 使用 U-boot v2023.07 稳定版本
- 修改底层驱动，使其支持 VGA Console+PS/2 键盘输入输出
- 添加如图 4.1 所示的菜单启动界面，能够自动通过网口加载并选择性启动 Linux 或 ucore 操作系统

图 4.1: U-Boot 菜单启动界面

4.2 ucore

4.2.1 背景

ucore 是清华大学的轻量级教学操作系统。

4.2.2 移植内容

- 修改底层驱动，使其支持 VGA Console+PS/2 键盘输入输出
- 在 U-Boot 启动菜单中增加 ucore 自启动选项。

4.3 Linux

4.3.1 背景

Linux 是最为著名的开源操作系统，有丰富的软硬件支持。我们以最新的稳定版本 Linux v6.4 (2023 年 6 月 26 日发布) 作为基线进行移植。

4.3.2 驱动程序

LCD 驱动

我们自行实现了 LCD 驱动，分为 3 个部分: 文本传输，普通图像传输和标准 FrameBuffer，它们分别对应/dev/下 3 个独立的字符设备。文本传输设备通过 write 在 LCD 的文本模式下顺序输出字符，类似串口。普通图像传输设备可通过 ioctl 接受 DMA 地址实现 DMA 快速传输 `dma_sync_single_for_device` 清理 Cache 实现数据同步，这允许了像素数据在传输前可以充分使用 Cache，提升处理性能。为减少阻塞，驱动程序对于多个未开始的 DMA 请求维护一个缓冲，当 DMA 中断发生时取下一个请求发送给控制器。

VGA Console 驱动

我们仿照内核默认的`dummy console`以及自带的`vga console`驱动程序,实现了用于自己 VGA 控制器的`consw`结构体及文本输出/滚屏/光标设置等的底层接口函数，它们都基于 VGA 控制器文本模式的操作。我们修改了内核启动代码，将默认 Console（`conswitchp`）设置为我们自己的，这样在 Linux 内核初始化完成后，可直接使用基于我们 VGA Console 驱动的`tty0`虚拟终端。由于 VGA 控制器的文本模式仅使用片上存储，不会与 CPU 竞争内存带宽，相比于常规的基于 FrameBuffer 的 Console 实现方式，这可以为系统带来更高的性能。

Linux 使用 VGA Console 的启动效果如图 4.2 所示。

PS/2 键盘驱动

Linux 启动后会使用虚拟终端，它的输入最终基于一个完备的 input 子系统，这给我们自行实现的键盘驱动带来了便利。我们只需要令键盘中断处理程序读取扫描码，通过 `input_report_key` 向内核的 input 子系统上报键盘输入事件即可。

图 4.2: Linux 启动界面

Framebuffer 驱动

为了能够支撑更多 GUI 应用，我们还为 VGA 和 LCD 控制器实现了 Framebuffer 驱动。Framebuffer 驱动的实现包含如下部分：一，开辟一块内存区域，存放每个像素的颜色信息；二，以合适的方式向显示设备传送内存中的颜色信息。由于我们的 VGA 控制器和 LCD 模块本身均采用 RGB565 颜色格式，故在实现驱动时使用同样的格式进行像素信息的存储，因而每个像素需要 2 字节空间。由于我们实现的 VGA、LCD 控制器均支持 DMA，故我们仅需向控制器传送内存区域首地址，由控制器直接进行数据的读取，即可完成图像显示。其中，VGA 控制器可自行进行不间断的数据读取；但 LCD 控制器完成一次数据传输后便会停止，所以我们使用计时器机制，每隔 50ms 传送一次内存区域首地址，以 20Hz 刷新率显示图像。

SD 驱动

由于我们通过 AXI QUAD SPI IP 核对 SD 卡进行操作，所以直接使用了该 IP 核对应的 SPI 总线驱动；同时，我们也直接使用了 Linux 内核主线中使用 SPI 总线的 MMC 驱动。在设备树中记录设备相关信息后，即可在进入 Linux 内核后将 SD 卡分区挂载到目录树中，进行读写操作。

4.3.3 用户态组件

Linux 内核本身并不能提供任何用户态组件，因此我们需要手工编译这一部分。我们选择了著名的嵌入式 Linux 开发工具套件 buildroot 来协助完成用户程序的构建。构建过程中，我们选择了 glibc 作为系统的 C/C++ 标准库实现，并使用 busybox 来提供大部分的命令行工具。由于构建的 rootfs 较大，无法使用 initramfs 等方式直接加载，而 Flash 的读写速度也较慢并不灵活，因此我们使用 NFS 协议通过网络挂载系统的根分区。实践证明，在网络稳定的情况下，系统的响应速度并不会受到影响。我们成功移植了如下内容：

GCC 由于 buildroot 理念在于生成体积更小的根文件系统，所以不支持 GCC 移植，我们参考 linux-from-scratch[1] 对 GCC 进行移植，并对 glibc 以及 binutils 利用相同方法移植到系统中，此外，由于 Ma-River

[1] https://www.linuxfromscratch.org/

CPU 实现了 FPU，所以在编译时我们添加了支持硬件浮点的选项，经过测试，GCC 可以成功编译出浮点指令并成功运行。我们在 Ma-River 上用移植的 GCC 编译了小游戏 2048，并成功在 Ma-River 上运行，以此突出我们的移植工作。

QEMU 我们成功移植了著名开源项目 QEMU, 并生成了可以模拟 RISC-V 架构的系统级模拟器 qemu-system-riscv64，但可惜的是实验板内存只有 128MB，无法满足系统级模拟器的运行需求。

Python 我们移植了 Python3, 并且添加了数据处理相关的库 (numpy,scipy 等)。我们还使用 Python 搭建了简单的 http 服务器，浏览器可以在局域网中访问板载服务器上的网页。以此证明我们的 CPU 能够在板载资源更充足的情况下运行更加复杂的应用，体现出我们工作的实用性。

Busybox 使用 Busybox 提供大部分命令行工具。

网络工具 移植了常用网络工具 (ip,ping,wget,nc,ssh) 大大提高了后续的移植以及测试工作的效率。

4.3.4 新增内容

中文支持

我们在 VGA Console 和键盘驱动程序的基础上，为 Linux 终端增加了基本的中文输入输出能力，基于标准的 UTF-8 编码，不仅可以输入/显示汉字，还可以将文件名等设为中文进行操作。在我们自己实现的 CPU 及计算机系统上支持中文汉字处理，也是一件颇有意义的事情。

我们的 VGA 控制器在文本模式下具有二值化图像显示能力，这使得软件理论上可基于字库绘制复杂字符。我们向 VGA Console 驱动程序中加入了使用 Unicode 索引的汉字点阵字库（宋体），收录了符合 GBK 规范的所有汉字。这样驱动程序可在屏幕某位置显示占两格的汉字。我们还对 Linux 内核中的虚拟终端负责输出字节流解码的部分作出了简要修改，使其不会对汉字对应的 UTF-8 编码报错，并正确地将汉字对应的 Unicode 码值传递给底层 VGA Console 驱动。有趣的是，我们还将哈工大校徽分解为 16*16 的不同色块存于字库中，映射到 Unicode 编码中未被汉字使用的区域，这样可以以“显示文本”的方式在屏幕任意位置显示哈工大校徽。

对于中文输入，我们自行实现了简单的拼音输入法，它基于 Trie 树进行查询匹配，支持多字输入、单字选择和简单的拼音分割，操作和普通的拼音输入法类似。我们将 VGA 文本模式的最下两行供输入法使用，键盘驱动程序在读取到扫描码后，会率先交由输入法模块进行处理，若输入法处于输入状态则拦截这一按键事件，不提交至 input 子系统。当输入完成后，输入法模块通过 `tty_insert_flip_string` 将汉字字符串的 UTF-8 编码字节流送入虚拟终端的 tty_port 里。

GUI 应用支持

我们对 LVGL 官方演示程序进行了移植，使其能够通过 Framebuffer 在 LCD 上输出 GUI 界面、通过输入子系统读取触摸信息。于是，我们便可以直接通过触摸来操作该演示程序提供的开关等控件，并看到即时的显示反馈。

GIF 动画播放器

我们实现了一个简单的 GIF 动画播放器，可以在 LCD 上显示 QQ 动态表情等。程序读取指定 GIF 文件并逐帧解析，使用 LZW 压缩算法库 [2] 对像素数据进行解压缩，通过 DMA 将每帧图像轮番显示在 LCD 屏幕上，帧率可达 20-30FPS。该动画播放器支持通过触摸屏移动图像。

[2] `https://github.com/jefftime/lzw`

3D 模型实时渲染

由于 Ma-River CPU 具有完备的浮点运算能力，并且我们的 LCD 控制器支持 DMA 加速绘制，这使得我们可以实现一些较复杂的图形学应用，例如 3D 模型渲染。

我们使用 C 语言编写了一个简单 3D 模型软渲染器（只使用纯 CPU 指令），使用 Blender 建立较复杂的 3D 模型，将其三角面信息以 stl 格式导出，渲染器据此得出模型顶点的三维坐标，将其乘以视图矩阵和投影矩阵进行投影变换（这需要大量浮点运算），投影到二维平面上，再绘制顶点间的线条，通过 DMA 将其显示到 LCD 屏幕上。我们的软渲染器是动态实时的，在 Linux 下运行时帧率可达 12FPS 左右，视角可自动随物体转动，也支持通过触摸移动视角，具有很好的交互性。

附录

致谢

我们对以下团体或个人表示由衷的感谢，没有他们，马家沟河项目是无法走到今天的。

- 感谢龙芯中科提供的设备与比赛平台支持，衷心祝愿他们能够在国产 CPU 道路上越走越远。
- 感谢舒燕君和刘国军两位指导老师的帮助、指导和关心。
- 感谢胡光辉等隔壁团队同学们的无私帮助。
- 感谢张清钰、王永琪、郑翔宇、丛日东等学长的帮助。
- 感谢清华大学 ZenCove、NonTrivalMips、重庆大学 CDIM 等往届团队，他们开源的往届作品及经验分享对我们的工作起到了极大帮助和启发作用。
- 感谢姚永斌老师的《超标量处理器设计》一书，此书给予了我们很大启发，没有它就没有 Ma-River 现在的微架构设计。

参考资料

本项目参考了包括但不限于下列书籍、资料、网站或开源项目：

- 姚永斌. 超标量处理器设计 [M]. 清华大学出版社: 201404.
- 唐朔飞. 计算机组成原理（第三版）[M]. 哈尔滨工业大学出版社: 202010.
- 汪文祥, 邢金璋. CPU 设计实战 [M]. 机械工业出版社: 202101.
- CEMU 文档：`http://cemu.cyyself.name/`
- CDIM 项目：`https://github.com/Maxpicca-Li/CDIM`
- ZenCove 项目：`https://github.com/zencove-thu/`
- NonTrivalMips 项目：`https://github.com/trivialmips/nontrivial-mips`
- SHIT Core 项目：`https://github.com/Superscalar-HIT-Core/Superscalar-HIT-Core-NSCSCC2020`
- TinyVGA：`http://tinyvga.com/`
- Framebuffer_driver_rpi：`https://github.com/dsoastro/framebuffer_driver_rpi`