BIRISCV 技术文档

2023 年 8 月 3 日

温榆河项目组

目 录

1 处理器

1.1 简介

Biriscv 处理器由 Biriscv 微架构 + DDR + UART + Debug Module + Bootrom 组成。其中 Biriscv 微架构是 In-order 双发射结构设计，目前支持 M 扩展，Z 扩展（RV32IMZicsr）。Biriscv 处理器前端流水线包括分支预测单元，取指单元，译码单元，指令缓存单元。后端包括发射单元，执行单元。访存系统包括 LSU，MMU，数据缓存。
主要功能：

- 64-bit 取指，32-bit 数据访问
- 2 x ALU 单元
- 1 x LSU 单元
- 1 x 非流水线除法单元
- 最高支持每周期发射并完成两条指令
- 支持 User，Supervisor，Machine mode 特权等级
- 1 x MMU 单元
- 可 boot Linux

1.2 Biriscv 微架构

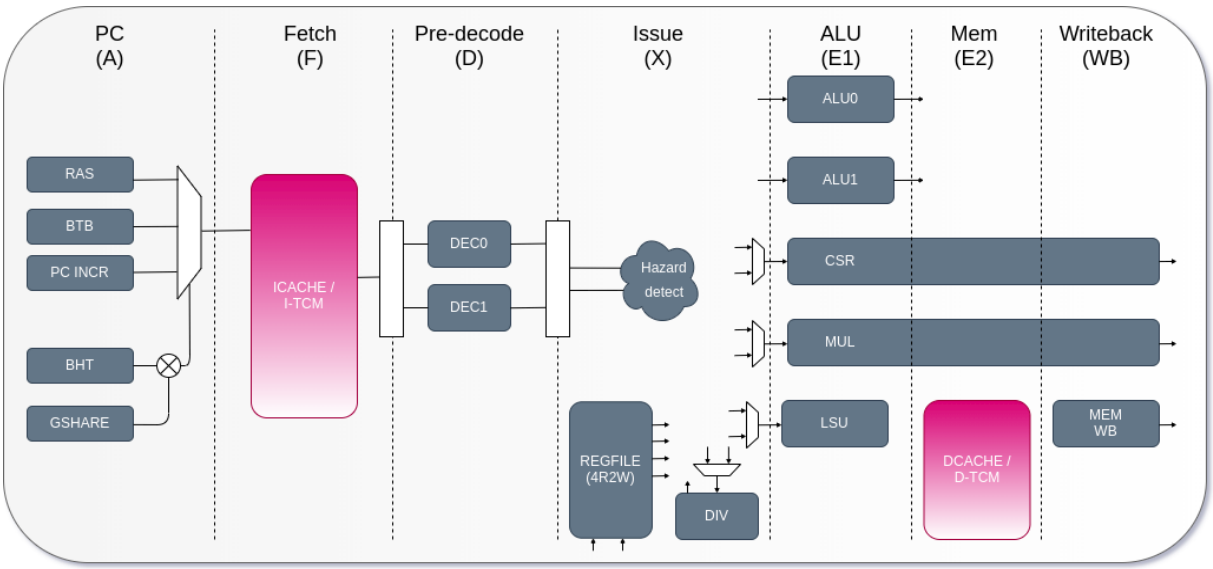


图 1.1 微架构示意图

1.3 前端

1.3.1 分支预测单元

输入端口：branch_info_request_i, branch_info_pc [31:0]

输出端口：next_pc_f_o [31:0]

单元功能介绍：分支预测算法采用 Gshare 两级预测架构，其主要组成部分包括 2 比特饱和计数器，全局历史寄存器（Global history register）。其主要算法为：使用全局历史寄存器结果异或 PC 值，得到 2 比特饱和计数器的索引值，根据 2 比特饱和计数器读出的结果判定分支预测的结果。同时，利用 PC 值作为分支目标缓冲（BTB）的索引值，读出对应的分支跳转地址，输出到取值模块。

1.3.2 取指单元

输入端口：next_pc_f_i [31:0], branch_request_i, branch_pc_i [31:0], icache_inst_i [63:0]

输出端口：fetch_pc_o [31:0], fetch_instr_o [63:0], icache_pc_o [31:0]

单元功能介绍：取指单元主要负责向处理器输送需要执行的指令 fetch_instr_o，取指单元首先收到来自分支预测单元的 PC 值 next_pc_f_i，输出地址到 MMU。如果 branch_request_i 信号被拉高，此时说明分支预测错误，取值单元将接受来自执行单元的 PC 值 branch_pc_i ，输出到 MMU。收到来自 MMU 的指令后，输出到译码单元。

1.3.3 译码单元

输入端口：fetch_pc_instr_i [31:0], fetch_in_instr_i [63:0], branch_request_i

输出端口：fetch_out0_instr_o [31:0], fetch_out0_pc_o [31:0], fetch_out1_instr_o [31:0], fetch_out1_pc_o [31:0]

单元功能介绍：译码单元主要负责将来自取指单元的 64 比特指令拆拆封成两路 32 比特指令 fetch_out0_instr_o，fetch_out1_instr_o，根据指令的 opcode 译码出该指令所对应的执行单元操作。将执行单元操作编译为 1bit 信号，输出到发射单元。

1.3.4 指令缓存

输入端口：axi_rdata_i[31:0],req_pc_i[31:0]

输出端口：fetch_out_pc_o[31:0]

单元功能介绍：指令缓存的作用是缓存指令，将要执行的指令和该指令邻近的指令都读取到指令缓存中缓存起来，要执行指令时先去指令缓存中查找，没有找到再去内存中读取。其中 axi_rdata_i 信号提供指令，req_pc_i 信号使用 Tag 进行比对，命中则将指令取出；未命中则请求写入，将 cache 中数据更新。fetch_in_inst_o 信号将查找到的指令发送给 MMU 单元。本设计采用两路组相联，大小为 16KB。

1.4 后端

1.4.1 发射单元

输入端口：fetch0_instr_i [31:0], fetch0_pc_i [31:0], fetch0_instr_i [31:0], fetch0_pc_i [31:0]

输出端口：opcode0_ra_operand_o[31:0], opcode0_rb_operand_o[31:0], opcode1_ra_operand_o[31:0], opcode1_rb_operand_o[31:0], csr_opcode_ra_operand_o[31:0], csr_opcode_rb_operand_o[31:0], mul_opcode_ra_operand_o[31:0], mul_opcode_rb_operand_o[31:0], lsu_opcode_ra_operand_o[31:0], lsu_opcode_rb_operand_o[31:0]

单元功能介绍：发射单元主要由发射队列，相关性检查，旁路网络组成。指令在发射阶段涉及到入队，选择，读寄存器堆，出队等操作。指令在发射单元将在发射队列中等候唤醒，同时对指令源操作数进行相关性检查，并读取寄存器堆。如果此时源操作数未就绪，指令将在发射队列中等候旁路网络前递源操作数，源操作数就绪后，两条指令将并行进入流水线。

1.4.2 执行单元

输入端口：alu_a_i[31:0], alu_b_i[31:0], opcode_ra_operand_i[31:0], opcode_rb_operand_i[31:0]

输出端口：alu_p_o[31:0], alu_p_o[31:0], csr_result_e1_value_o[31:0], writeback_value_o[31:0],

单元功能介绍：执行单元由两个 ALU 单元，一个 CSR 单元，一个 MUL 单元组成。ALU 单元主要执行整数运算操作，并将运算后的结果通过 alu_p_o 信号传输给旁路网络和寄存器堆。CSR 单元主要负责执行 CSR 相关操作。在 Briscv 中，CSR 操作将直接触发流水线停顿，操作将进行 2-3 个时钟周期。CSR 操作的结果将通过 csr_result_e1_value_o 写回到 CSR 寄存器堆。MUL 单元主要负责执行乘法操作，操作将进行 2-34 个时钟周期。MUL 结果将通过 writeback_value_o 写回。

1.5 访存系统

1.5.1 LSU

输入端口：opcode_ra_operand_i[31:0], opcode_rb_operand_i[31:0], mem_data_rd_i[31:0]

输出端口：writeback_value_o[31:0], mem_addr_o[31:0], mem_data_wr_o[31:0]

单元功能介绍：LSU 单元主要执行 LOAD 和 STORE 指令。LOAD 指令时，LSU 单元收到来自发射单元的源操作数 opcode_ra_operand_i 和 opcode_rb_operand_i，计算出访存地址 mem_addr_o 并输出给 MMU。访存结果将通过 mem_data_rd_i 端口输入到 LSU 单元，通过 mem_addr_o 端口写回。STORE 指令时，访存数据将通过 mem_data_wr_o 输出到 MMU。

1.5.2 MMU

输入端口：fetch_out_inst_i[63:0],fetch_in_pc_i[31:0], lsu_in_addr_i[31:0], lsu_out_data_wr_w[31:0],lsu_out_data_rd_i[31:0]，

输出端口：req_inst_o[63:0],fetch_out_pc_o[31:0], lsu_in_data_rd_o[31:0],lsu_out_addr_o[31:0],lsu_out_data_wr_o[31:0]

单元功能介绍：虚实地址翻译。将输入虚拟地址与 TLB 页表进行比对，如果命中，则取出物理地址，若是未命中，则进入 Page fault 状态，对 TLB 进行 update。其中 MMU 可分为 IMMU 和 DMMU，IMMU 功能为指令内存管理模块，DMMU 为数据内存管理模块。

1.5.3 数据缓存

输入端口：mem_addr_i[31:0],mem_data_wr_i[31:0]

输出端口：mem_data_rd_o[31:0]

单元功能介绍：dcache 功能与 icache 结构和功能相类似，但是 dcache 的作用是缓存数据，同样也会经过 Tag 比较，判命中最终 Data 阵列出数；未命中则请求写入，将 cache 中数据更新。

2 分支预测单元

2.1 简介

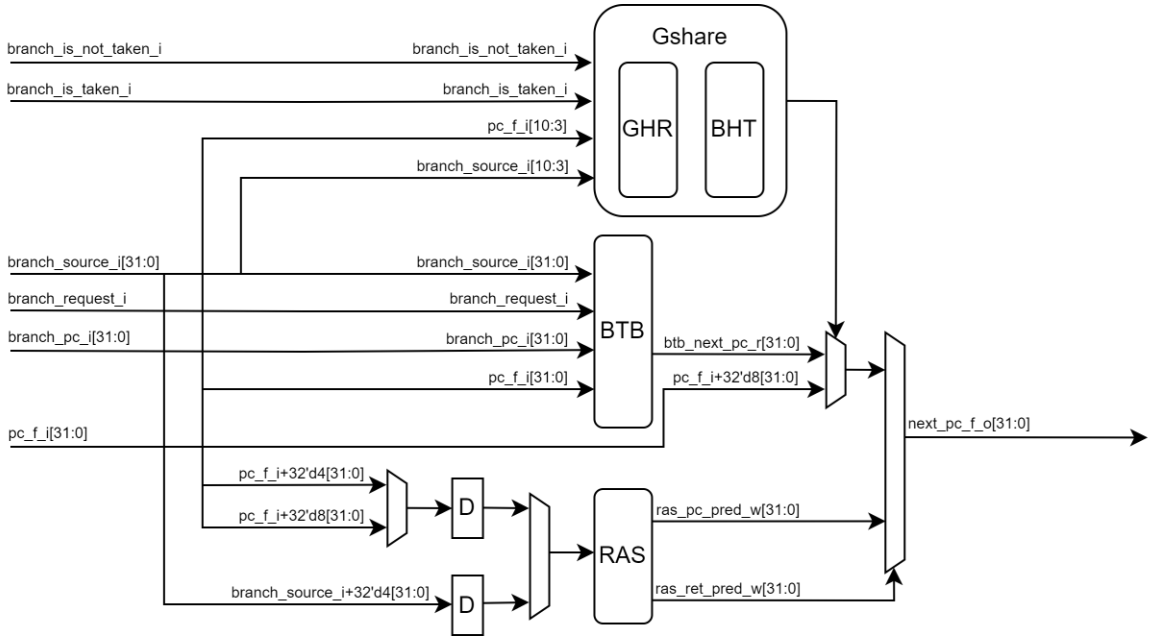


图 2.1 分支预测单元图示

在 Biriscv 的分支预测单元当中。此分支预测模块基于 BTB, BHT, GHR 和 RAS 给出下一个 PC 的地址。其中 RAS 模块是　用来存储 call 指令的 PC+4 作为 call 结束的返回地址，RAS 有两个栈指针，分别为 ras_index_real_q 代表真实的栈指针，ras_index_q 代表推测的栈指针。当 BTB 预测该指令是 call 时，ras_index_q 加 1；当 BTB 预测该指令是 return 时，ras_index_q 减 1。当 issue 返回预测错误并且该指令是 call 时，ras_index_real_q 加 1；当 issue 返回预测错误并且该指令是 return 时，ras_index_real_q 减 1。发生预测错误时，推测的栈指针 ras_index_q 会被置为真实的栈指针 ras_index_real_q。当推测该指令为 call 或者 Issue 返回预测错误并且该指令是 call 时，把下一条指令（PC+4）的值放到栈里面；当推测该指令为 return 或者 Issue 返回预测错误并且该指令是 return 时，弹出 ras_index_q 指针对应的值。

GHR 分为两个历史寄存器，global_history_real_q 用来存放 issue 返回的真实分支结果，global_history_q 用来存放预测的分支结果。当分支预测失败时，global_history_q 会被置为 global_history_real_q 的值并更新此次分支结果。

BHT 是一个深度为 512 的饱和计数器，采用两种不同算法寻址：

使用 gshare 算法寻址：用 GHR 和 PC 值的组合作为地址来查找或修改 BHT。

使用 PC 的一部分来寻址：使用 PC 的[11:2]位来寻址。

BTB 是一个深度为 32 的存储器，存储了当前 PC 值，目标 PC 值，call，return，jump 的标志位。当分支预测错误时，如果 BTB 中找到相应的 entry，那么就更新该 entry；如果没有找到相应的 entry，就用 lfsr 算法随机选择 BTB 中的一个 entry 进行替换。

2.1.1 Next PC 判定逻辑

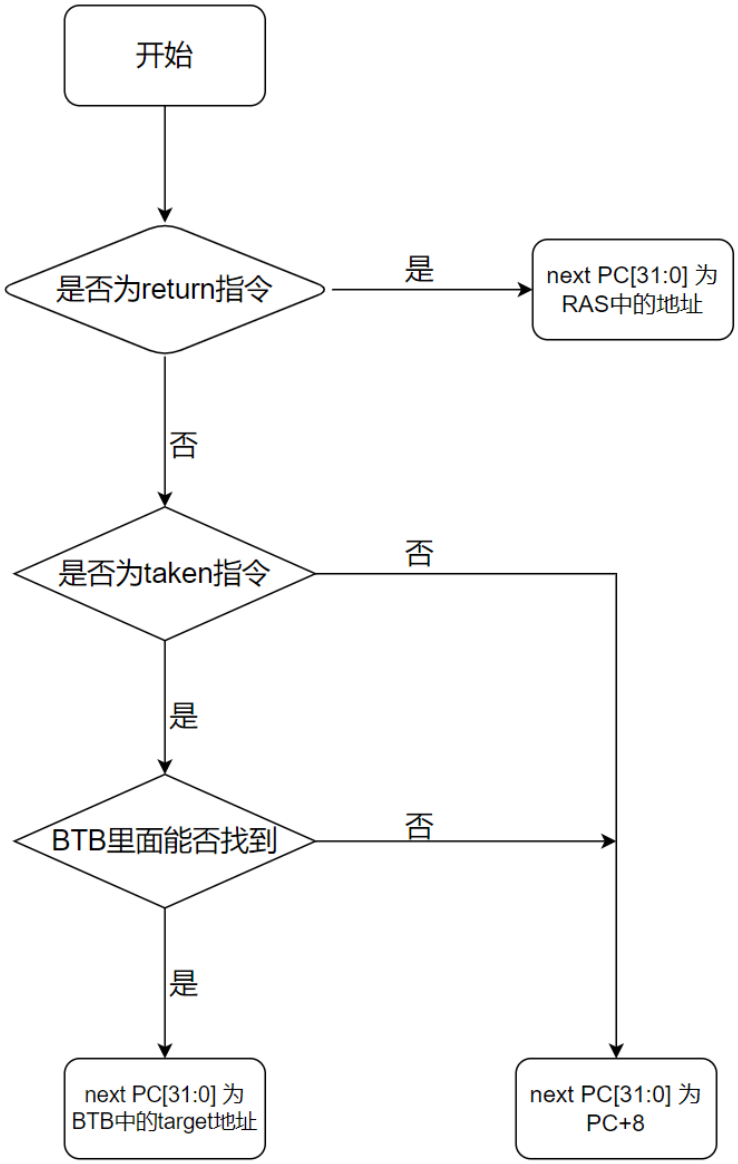


图 2.2 逻辑判定示意图

3 取指单元

3.1 简介

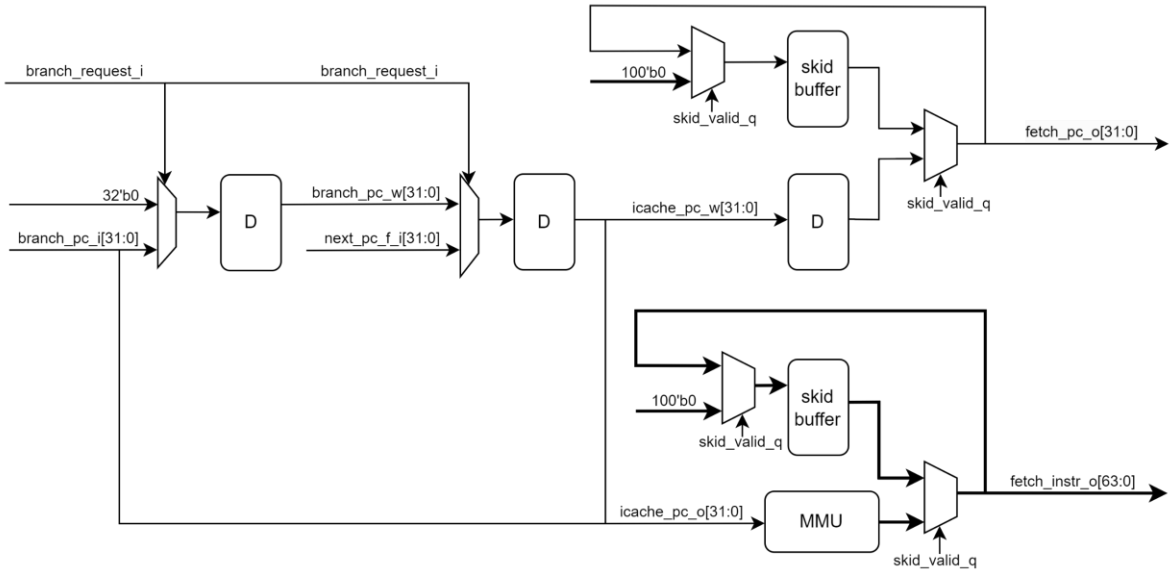


图 3.1 取指单元数据通路

取指单元主要负责向处理器输送需要执行的指令 fetch_instr_o，取指单元首先收到来自分支预测单元的 PC 值 next_pc_f_i，输出地址 MMU。如果 branch_request_i 信号被拉高，此时说明分支预测错误，取值单元将接受来自执行单元的 PC 值 branch_pc_i ，输出到 MMU。收到来自 MMU 的指令后，输出到译码单元。

4 译码单元介绍

4.1 简介

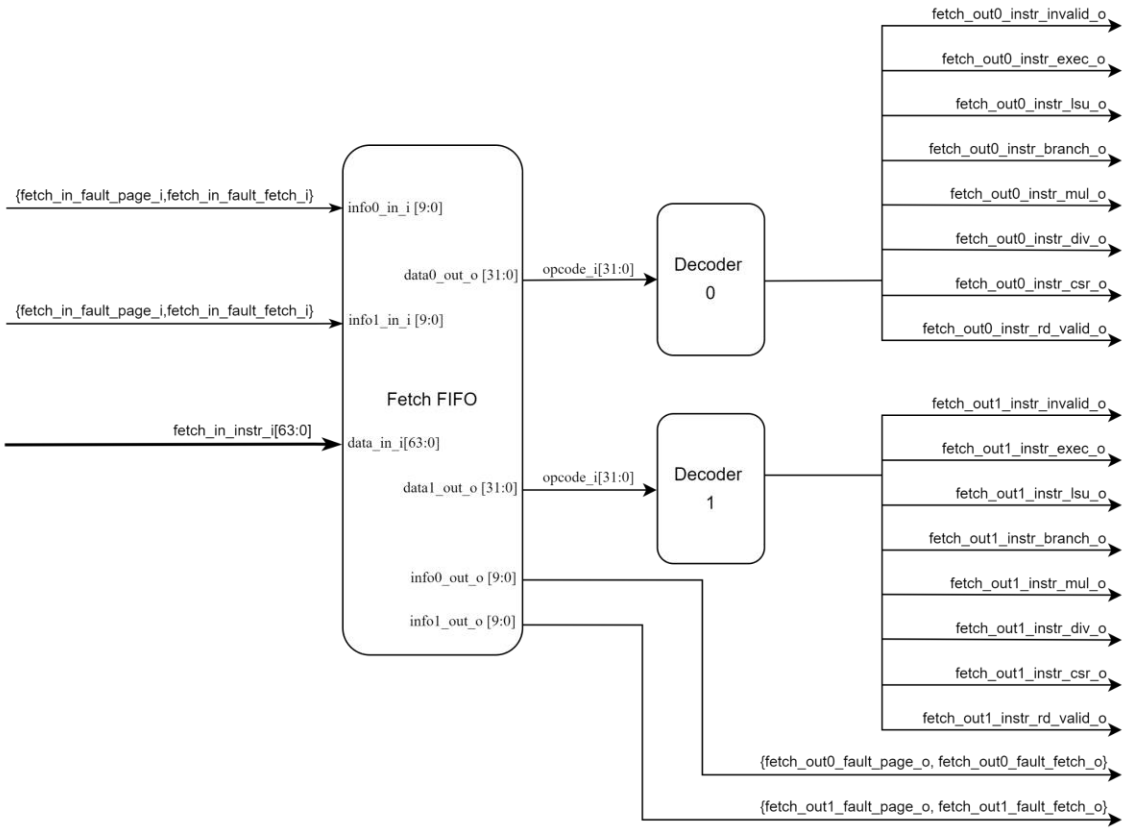


图 4.1 译码单元数据通路

译码单元主要负责将来自取指单元的 64 比特指令拆拆封成两路 32 比特指令 fetch_out0_instr_o，fetch_out1_instr_o，根据指令的 opcode 译码出该指令所对应的执行单元操作。将执行单元操作编译为 1bit 信号，输出到发射单元。

5 执行单元介绍

5.1 简介

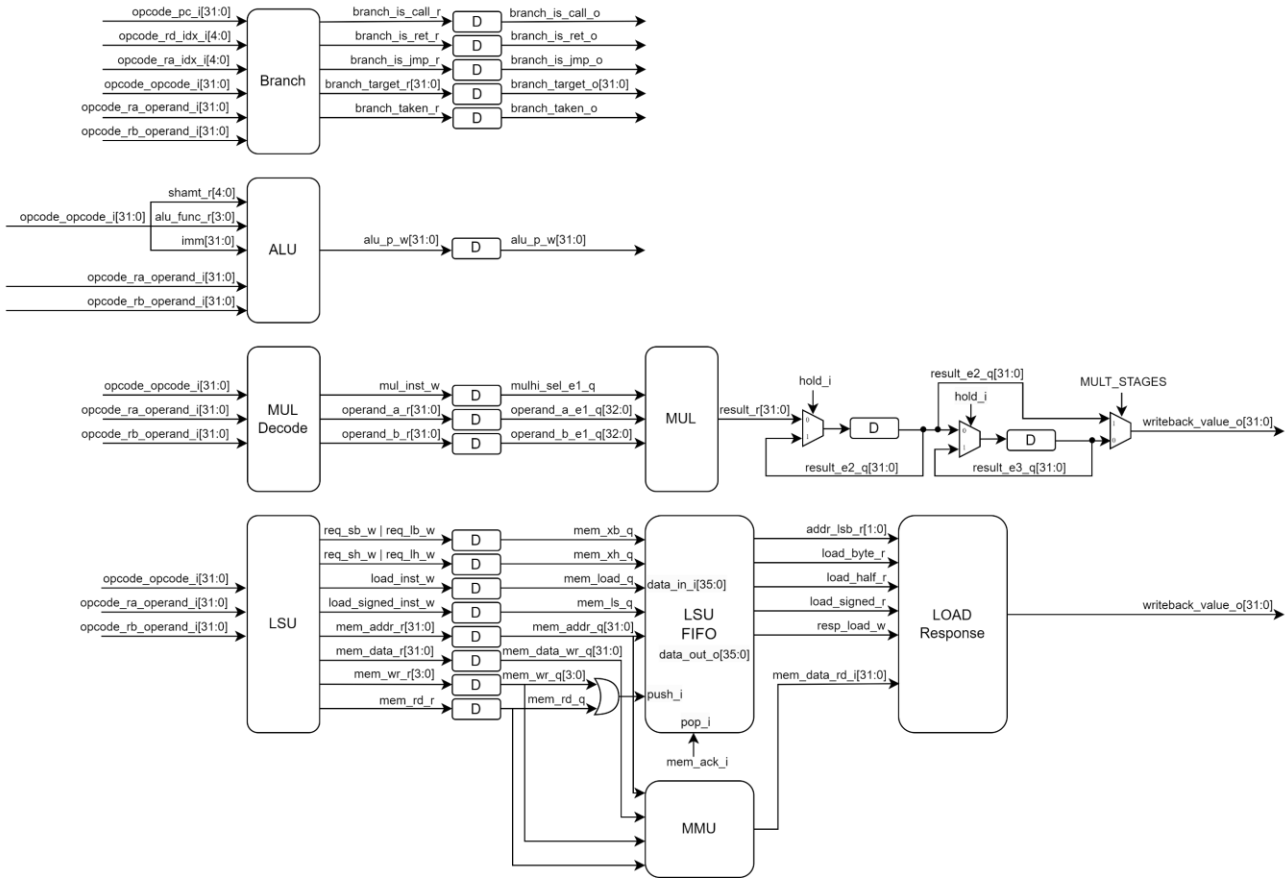


图 5.1 执行单元数据通路

在 Biriscv 当中，执行单元主要包括 ALU，Branch，MUL，LSU 模块。在 ALU 模块中主要负责进行整数运算操作，ALU 模块的源操作数来源于执行单元根据得到的指令判断对应的指令操作，获取操作数或者从指令中取到的立即数。

Branch 模块主要负责判断指令是否跳转，是什么类型的跳转指令以及跳转地址。MUL 模块中主要负责乘法运算操作，根据 issue 得到的指令判断对应的指令操作，获取对应的操作数进行乘法运算。在 LSU 当中，主要由 DECODE，FIFO，和 RESPONSE 模块组成，在 DECODE 模块中，LSU decode 模块通过指令解析出对应的 Load 和 Store 指令，包括这些指令对应的读写信息。然后根据指令信息和源操作数解析出对应的 load/store 的地址以及 store 指令对应的数据信息。LSU FIFO 深度为 2，将 store 指令和 load 指令对应的地址和指令状态信息存储在起来，接收到 mmu 的 ack 信号弹出数据。从 fifo 中弹出数据后，如果指令是 load 指令，通过 load 指令解析出来的信息，将 mmu 传过来的数据结果写给 issue。在 LOAD RESPONSE 模块中，从 fifo 中弹出数据后，如果指令是 load 指令，通过 load 指令解析出来的信息，将 mmu 传过来的数据结果写给 issue。

6 发射单元介绍

6.1 简介

Issue 单元主要由 PC 单元，发射选择(Issue select)单元，指令译码(Instruction decoder)单元，发射逻辑(Issue logic)单元，旁路网络(Bypass Network)单元组成，流水线控制(pipeline control)单元，分支预测更新(branch predictor info)单元。下文将逐一介绍这些模块。

6.1.1 PC 单元

在 Brisicv 的 Issue 单元中，如下图所示，PC 模块的用途是存储正确的 PC 值。此 PC 值被存放在 pc_x_q 寄存器当中。RST 时 pc_x_q 置零。当遇 CSR 跳转请求时，pc_x_q 更新为 CSR 跳转地址。当遇到 Branch 跳转请求时，pc_x_q 更新为 Branch 跳转地址，这个 Branch 跳转地址是由 ALU 单元计算出来的。如果以上情况都不符合，那么判断此时 CPU 是双发射(Dual Issue)模式或者单发(Single Issue)模式，分别对应 PC+8 和 PC+4。经过更新后，输出最新的 pc_x_q [31:0], 此信号为 PC 的期待值。

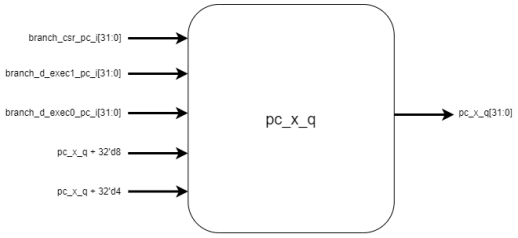


图 6.1 pc_x_寄存器

6.1.2 发射选择单元

在 Briscv 当中的发射选择单元主要是负责判断分支预测错误的发生。上文我们提到过 pc_x_q 寄存器，这个寄存器存储着正确的 PC 值，在发射选择单元中，会首先将 fetch0 指令的 PC 值和 pc_x_q 寄存器的值对比，若一致，则将对应的 slot0_valid 信号置 1。如果发现 fetch0 指令的 PC 与 pc_x_q 的 pc 值不一致，则比较 fetch1 指令。如果 fetch1 指令有效的 PC 值正确，将 slot1_valid 置 1。如果两条指令的 pc 地址与 pc_x_q 均不一致，同时指令有效，那么将触发 Mispredict 信号，通知前端冲刷流水线。除了 Mispredict，CSR 跳转也会触发冲刷流水线的操作。Mispredict 和 CSR 跳转由 branch_request_o 信号向前端传递。跳转的 PC 地址由 branch_pc_o 向前端传递。

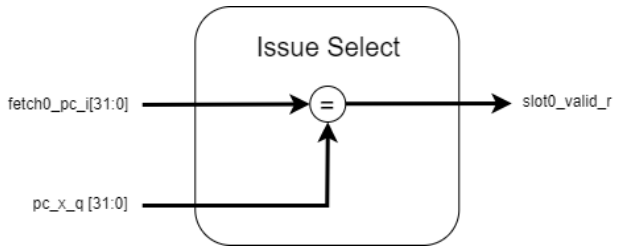


图 6.2 Issue Select 单元

6.1.3 指令译码单元

指令译码单元主要负责对指令进行译码，分发。其中译码部分主要是从指令中提取源寄存器 1，源寄存器 2，和目标寄存器。分发部分则是根据 slot0，和 slot1 的 valid 值来决定是否将指令分配给 slot0/1。在 Biriscv 处理器当中，CSR 指令只能经过 slot0 进行发射，所以 slot1 的输出会有一条通路是连接到 slot0 上边的。下文将通过代码示例对此单元功能进行详细描述。

```
// Word 0 (and possibly slot 1) are valid instructions
if (slot0_valid_r)
begin
    opcode_a_valid_r = 1'b1;
    opcode_b_valid_r = fetch1_valid_i;
    opcode_a_r       = fetch0_instr_i;
    opcode_a_pc_r    = fetch0_pc_i;
    opcode_a_fault_r = {fetch0_fault_page_i, fetch0_fault_fetch_i};
    opcode_b_r       = fetch1_instr_i;
    opcode_b_pc_r    = fetch1_pc_i;
    opcode_b_fault_r = {fetch1_fault_page_i, fetch1_fault_fetch_i};
end
```

图 6.3 Slot0 部分代码

在上图中，如果 slot0_valid_r 信号被拉高，此时说明 pc_x_q 和 fetch0 指令的 pc 地址一致，那么第一条指令一定是正确的，因为在比较 pc 值的时候我们也对 fetch0 的 valid 信号进行了检查。但是我们不能保证第二条指令也是正确的，所以此时 opcode_b_valid_r 不能被直接写入 1’b1，而是要写入 fetch1_valid_i。这部分代码同时也对每条指令对应的各个（pc 值，指令，错误代码）寄存器进行赋值。

```
// Word 1 valid - mux to first issue slot
// Note: Some instruction types can only issue in slot0, hence this muxing
else if (slot1_valid_r)
begin
    opcode_a_valid_r = 1'b1;
    opcode_b_valid_r = 1'b0;
    opcode_a_r       = fetch1_instr_i;
    opcode_a_pc_r    = fetch1_pc_i;
    opcode_a_fault_r = {fetch1_fault_page_i, fetch1_fault_fetch_i};
    opcode_b_r       = 32'b0;
    opcode_b_pc_r    = 32'b0;
    opcode_b_fault_r = 2'b0;
end
```

图 6.4 Slot1 部分代码

在上图中，如果 slot1_valid_r 信号被拉高，此说明 fetch0 指令的 pc 值是错误的，fetch1 指令的 pc 值是正确的。那么此时我们舍弃掉 fetch0 指令，只执行 fetch1 指令。将 fetch1 指令赋值给 opcode_a 相关的寄存器。将 opcode_b 相关的寄存器置 0。

6.1.4 发射逻辑单元

如下图所示当一条指令经过了指令译码单元之后，会得到这条指令的 ra, rb, rd 也就是两个源寄存器和一个目标寄存器的 index。此时这条指令会进入发射逻辑单元当中，发射逻辑单元的主要任务分为三项：

- 对指令的操作数进行相关性检查；
- 判断是否可以进行双发射；
- 如果流水线当中正在处理执行周期较长的指令，那么后续同样执行周期较长的指令将被延迟发射。

首先，在进行指令源操作数的相关性检查的时候，发射逻辑单元主要利用 Scoreboard 来判定当前指令的操作数是否被占用。Scoreboard 的数据结构是一个 32 位数组。当一条指令进入发射逻辑单元后，发射逻辑单元将利用目标寄存器的 index 作为索引，将 Scoreboard 中对应的空位置 1。后续指令可以通过查找 Scoreboard 来判定当前指令的寄存器是否被占用。当指令通过相关性检查后，对应的 valid 信号将被置 1。

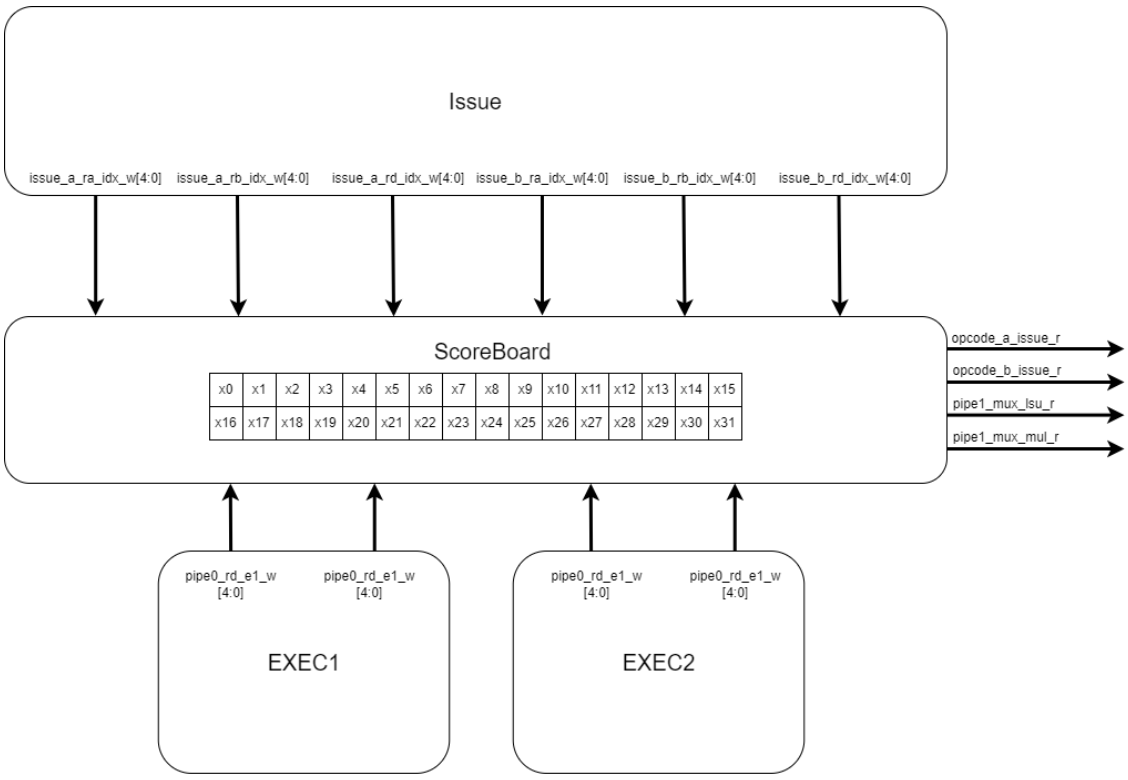


图 6.5 Scoreboard 示例图

发射逻辑单元要做的第二项工作是判断是否可以双发射，这里将贴出部分代码作为示例:

```
// Check instructions can be issued in the second execution unit
wire pipe1_ok_w      = issue_b_exec_w | issue_b_branch_w | issue_b_lsu_w | issue_b_mul_w;

// Is this combination of instructions possible to execute concurrently.
// This excludes result dependencies which may also block secondary execution.
wire dual_issue_ok_w =   enable_dual_issue_w &&  // Second pipe switched on
                         pipe1_ok_w &&           // Instruction 2 is possible on second exec unit
                        (((issue_a_exec_w | issue_a_lsu_w | issue_a_mul_w) && issue_b_exec_w)   ||
                         ((issue_a_exec_w | issue_a_lsu_w | issue_a_mul_w) && issue_b_branch_w) ||
                         ((issue_a_exec_w | issue_a_mul_w) && issue_b_lsu_w)                    ||
                         ((issue_a_exec_w | issue_a_lsu_w) && issue_b_mul_w)
                         ) && ~take_interrupt_i;
```

图 6.6 双发射仲裁逻辑代码

通过上图我们可以看到，双发射仲裁的第一步是判断指令是否可以在 pipe1（第二条流水线）当中执行。可以在 pipe1 中执行的指令有：ALU 相关指令，跳转指令，Load/Store 指令，乘法指令。通过第一步判断后，我们还需要判断 pipe1 和 pipe0 是否可以同时开始执行。在 Birsicv 当中，允许双发射的指令组合有如下几种：

表 6.1 Biriscv 双发射指令组合

Issue_a	Issue_b
ALU 指令	ALU 指令
Load/Store 指令	ALU 指令

乘法指令	ALU 指令
ALU 指令	跳转指令
Load/Store 指令	跳转指令
乘法指令	跳转指令
ALU 指令	Load/Store 指令
乘法指令	Load/Store 指令
ALU 指令	乘法指令
Load/Store 指令	乘法指令

下一步发射逻辑单元所要做的事情就是根据发射指令类型，对后续的指令进行延迟发射。具体实现如下：

```
// Do not start multiply, division or CSR operation in the cycle after a load (leaving only ALU operations and branches)
if ((pipe0_load_e1_w || pipe0_store_e1_w || pipe1_load_e1_w || pipe1_store_e1_w ) && (issue_a_mul_w || issue_a_div_w || issue_a_csr_w))
    scoreboard_r = 32'hFFFFFFFF;
```

图 6.7 Biriscv 双发射指令延迟逻辑

如上图所示，如果发射逻辑单元检测到上一个周期发射的指令为 Load 指令或 Store 指令，当前周期要发射指令 Mul，Div，CSR 中的任何一个，当前周期停止发射。

旁路网络单元：

旁路网络单元主要功能是对源操作数的 index 进行监控，当源操作数就绪的时候进行数据前递。在 Biriscv 当中，双发射的两条指令是由不同的流水线控制单元进行管理的。所以 slot0 和 slot1 的两个旁路网络单元要对两个流水线控制单元(pipe0 & pipe1)进行监控。当源操作数就绪以后，此时指令已经经过了 PC 检查，指令译码，相关性检查，源操作数前递四个步骤。当全部就绪后，指令就可以发送给流水线控制单元了。

```
// Bypass - E1
if (pipe0_rd_e1_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = writeback_exec0_value_i;
if (pipe0_rd_e1_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = writeback_exec0_value_i;

if (pipe1_rd_e1_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = writeback_exec1_value_i;
if (pipe1_rd_e1_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = writeback_exec1_value_i;
```

图 6.8 Exec 1 数据前递逻辑

```
// Bypass - E2
if (pipe0_rd_e2_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = pipe0_result_e2_w;
if (pipe0_rd_e2_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = pipe0_result_e2_w;

if (pipe1_rd_e2_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = pipe1_result_e2_w;
if (pipe1_rd_e2_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = pipe1_result_e2_w;
```

图 6.9 Exec 2 数据前递逻辑

```
// Bypass - WB
if (pipe0_rd_wb_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = pipe0_result_wb_w;
if (pipe0_rd_wb_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = pipe0_result_wb_w;

if (pipe1_rd_wb_w == issue_a_ra_idx_w)
    issue_a_ra_value_r = pipe1_result_wb_w;
if (pipe1_rd_wb_w == issue_a_rb_idx_w)
    issue_a_rb_value_r = pipe1_result_wb_w;
```

图 6.10 写回阶段数据前递逻辑

上图所示的分别是，Exec1，Exec2，WB 阶段的数据前递逻辑。其原理是通过对比各个阶段的目标寄存器 Index，和发射阶段的源寄存器 Index。如果发现一致，则将数据进行前递。

6.1.5 流水线控制单元

在 Biriscv 处理器当中，流水线总共分为 6 级或 7 级(可自定义 decode stage)。其中后三级流水线：exec1，exec2，wb 都是由流水线控制单元进行管理的。流水线控制单元包含三个模块 E1、E2、E3，分别负责地址分配、存储结果、将结果寄存器中的值写回到寄存器堆。下面将逐一介绍这些模块。

表 6.2 E1 模块

名称	位宽	信号类型	说明
issue_stall_i	1	input	流水线停顿信号
issue_lsu_i	1	input	lsu 信号
issue_csr_i	1	input	csr 信号
issue_div_i	1	input	div 信号
issue_mul_i	1	input	mul 信号
issue_accept_i	1	input	accept 信号
issue_valid_i	1	input	issue 有效信号
squash_e1_e2_i	1	input	流水线冲刷信号
issue_rd_valid_i	1	input	目标寄存器信号

issue_branch_i	1	input	跳转信号
take_interrupt_i	1	input	中断信号
branch_misaligned_w	1	input	分支预测未对准
squash_e1_e2_o	1	input	流水线冲刷信号
issue_branch_taken_i	1	input	分支预测被采用
issue_exception_i	6	input	异常信号
issue_pc_i	32	input	pc 地址信号
issue_branch_target_i	32	input	分支预测 next pc 地址信号
issue_opcode_i	32	input	操作码信号
issue_operand_ra_i	32	input	操作数
issue_operand_rb_i	32	input	操作数
valid_e1_q	1	output	E1 有效信号
opcode_e1_q	32	output	E1 中操作码信号
pc_e1_q	32	output	E1 中 pc 地址
npc_e1_q	32	output	E1 中 next pc 地址
operand_ra_e1_q	32	output	E1 中操作数
operand_rb_e1_q	32	output	E1 中操作数
ctrl_e1_q	`PCINFO_W	output	E1 中控制信号
exception_e1_q	`EXCEPTION_W	output	E1 中异常信号

1. 流水线正常工作时

pc_e1_q[31:0]、opcode_e1_q[31：0]、operand_ra_e1_q[31:0]、operand_rb_e1_q[31:0]的值分别由输入信号 issue_opcode_i[31:0]、issue_operand_ra_i[31:0]、issue_pc_i[31:0]、issue_operand_rb_i[31:0]赋予。npc_e1_q[31:0]为下一周期的 pc 地址，在输入信号 issue_branch_taken_i 为 1 表示分支预测正确时，取 issue_branch_target_i[31:0]的值，否则取 issue_pc_i[31:0]+31’b4。
exception_e1_q[5:0]存储异常信息，只要输入信号 issue_exception_i[5:0]中有 1 信号，就将其值赋予 exception_e1_q[5:0]。否则检查 branch_misaligned_w 信号，为 1 时对 exception_e1_q[5:0]赋 `EXCEPTION_MISALIGNED_FETCH，为 0 时赋 6’b0。

ctrl_e1_q[9：0]为控制信号，每一位信号分别对应一个功能单元，从小到达依次为 ALU、LOAD、STORE、CSR、DIV、MUL、BRANCH、RD_VALID、INTR、COMPLETE。在不发生中断（take_interrupt_i 信号不为 1）时，前八位信号继承来自 issue 的控制信号，INTR 位取 0，COMPLETE 位取 1。

2. 流水线中发生异常/中断时

发生中断时，ctrl_e1_q[7：1]为 0，INTR 位和 COMPLETE 位为 1。

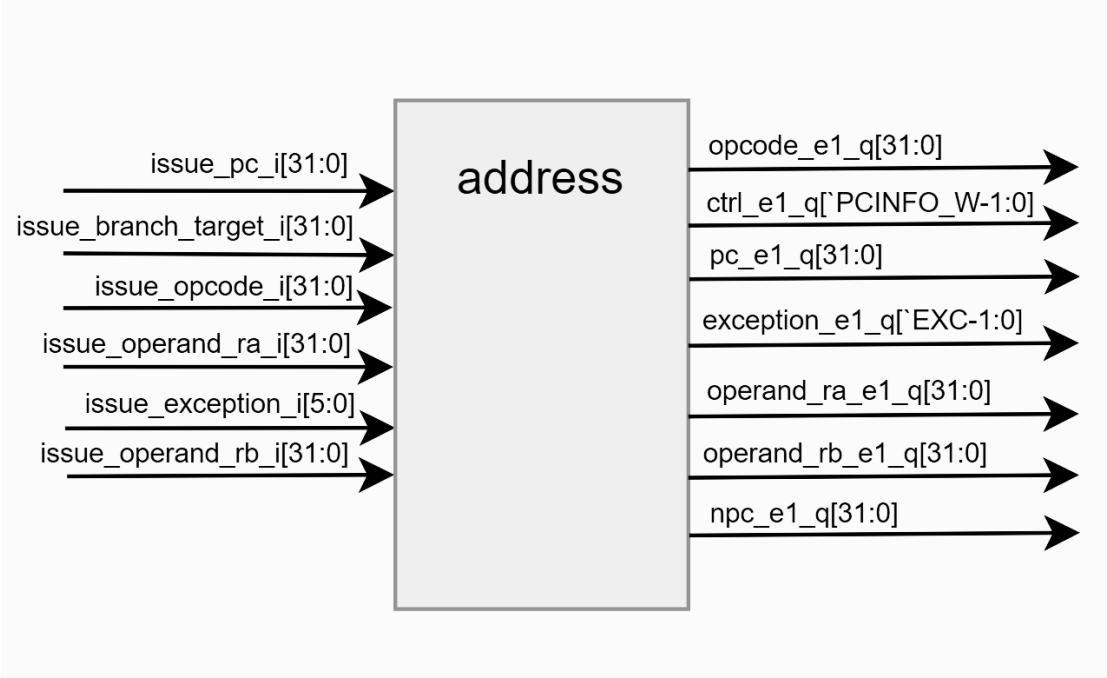


图 6.11 端口示意图

表 6.3 E2 模块

名称	位宽	信号类型	说明
squash_e1_e2_i	1	input	流水线冲刷信号
issue_stall_i	1	input	流水线停顿信号
squash_e1_e2_o	1	input	流水线冲刷信号
csr_result_write_e1_i	1	input	写回有效信号
valid_e1_q	1	input	E1 有效信号
csr_result_exception_e1_i	6	input	csr 异常信号
pc_e1_q	32	input	来自 E1 的 pc 地址
operand_ra_e1_q	32	input	来自 E1 的操作数
operand_rb_e1_q	32	input	来自 E1 的操作数
npc_e1_q	32	input	来自 E1 的 next pc 地址
csr_result_wdata_e1_i	32	input	要写入 csr 寄存器的值
div_result_i	32	input	写回的 div 值
csr_result_value_e1_i	32	input	csr 读取结果
alu_result_e1_i	32	input	alu 结果
ctrl_e1_q	`PCINFO_W	input	来自 E1 的 ctrl 信号
exception_e1_q	`EXCEPTION_W	input	来自 E1 的异常信号
valid_e2_q	1	output	E2 有效信号
csr_wr_e2_q	1	output	csr 写回信号
result_e2_q	32	output	写回数据
csr_wdata_e2_q	32	output	要写入 csr 寄存器的值

pc_e2_q	32	output	E2 中 pc 地址
npc_e2_q	32	output	E2 中 next pc 地址
opcode_e2_q	32	output	E2 中操作码
operand_ra_e2_q	32	output	E2 中操作数
operand_rb_e2_q32	32	output	E2 中操作数
exception_e2_q	`EXCEPTION_W	output	E2 中异常
ctrl_e2_q	`PCINFO_W	output	E2 中控制信号

3. 流水线正常工作时

csr_result_exception_e1_i[5:0]的值载入 exception_e2_q[5:0]。如果 ctrl_e1_q[9:0]中 DIV 位为 1，result_e2_q[31:0]赋 div_result_i[31:0]，否则检查 CSR 位信号，为 1 时将 csr_result_value_e1_i[31:0]赋予 result_e2_q[31:0]，为 0 时将 alu_result_e1_i[31:0]赋予 result_e2_q[31:0]。

4. 流水线中发生异常/中断时

当系统中存在异常时，冲洗掉模块中的信号。如果 ctrl_e1_q[9:0]中 INTR 位为 1,exception_e2_q[5:0]赋 6'h20，否则检查前端 exception_e1_q[5:0]是否存在异常，如果有则将 exception_e1_q[5:0]赋予 exception_e2_q[5:0]、valid_e2_q 赋 0。

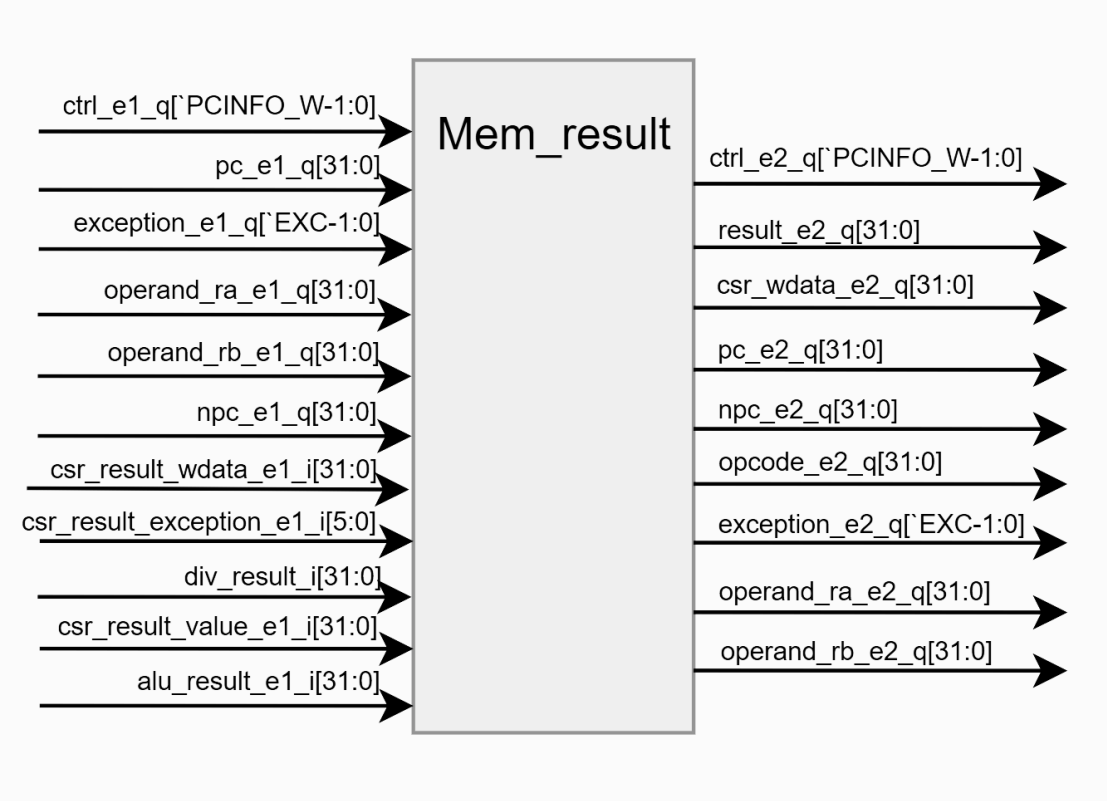


图 6.12 端口示意图

流水线控制单元输出的 result_e2_o[31:0]在 mul_result_e2_i[31:0]、result_e2_q[31:0]、mem_result_e2_i[31:0]三个信号当中做出仲裁选择。SUPPORT_LOAD_BYPASS、valid_e2_w 为 1，ctrl_e2_q 信号中的 LOAD 位或 STORE 位为 1 时赋 mul_result_e2_i[31:0]的值。其次在 SUPPORT_MUL_BYPASS、valid_e2_w、ctrl_e2_q 信号中的 MUL 位为 1 时赋 mul_result_e2_i[31:0]的值，否则赋 result_e2_q[31:0]的值。

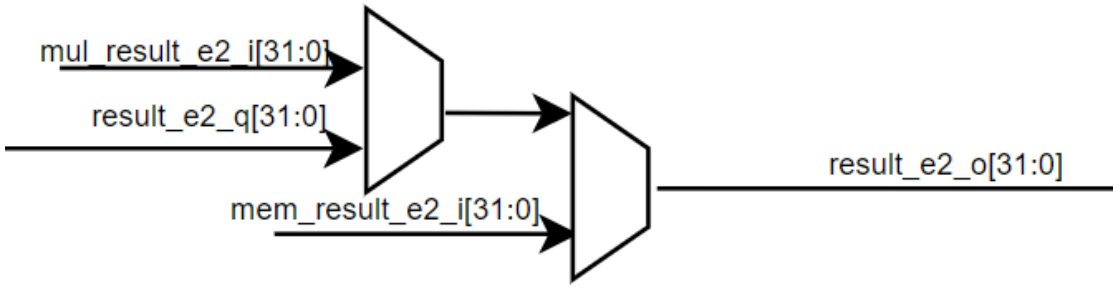


图 6.13 数据通路

表 6.4 E3 模块

名称	位宽	信号类型	说明
squash_wb_i	1	input	流水线冲刷信号
valid_e2_w	1	input	E2 有效信号
issue_stall_i	1	input	流水线停顿信号，为 1 时，流水线停止运行
valid_e2_q	1	input	E2 有效信号
csr_wr_e2_q	1	input	csr 写回信号
mem_result_e2_i	32	input	写回缓存结果
mul_result_e2_i	32	input	写回的 mul 结果
result_e2_q	32	input	来自 E2 的写回结果
csr_wdata_e2_q	32	input	要写入 csr 寄存器的值
pc_e2_q	32	input	来自 E2 的 pc 地址
npc_e2_q	32	input	来自 E2 的 next pc 地址
opcode_e2_q	32	input	来自 E2 操作码
operand_ra_e2_q	32	input	来自 E2 操作数
operand_rb_e2_q	32	input	来自 E2 操作数
exception_e2_q	`EXCEPTION_W	input	来自 E2 的异常信号
ctrl_e2_q	`PCINFO_W	input	来自 E2 的控制信号
csr_write_wb_q	1	output	写回信号
valid_wb_q	1	output	写回有效信号
opcode_wb_q	32	output	写回操作码
pc_wb_q	32	output	写回 pc 地址
result_wb_q	32	output	写回结果
operand_ra_wb_q	32	output	写回操作数

operand_rb_wb_q	32	output	写回操作数
csr_wdata_wb_q	32	output	写回 csr 寄存器数据
npc_wb_q	32	output	写回 next pc 地址
exception_wb_q	`EXCEPTION_W	output	写回异常信号
ctrl_wb_q	`PCINFO_W	output	写回控制信号

5. 流水线正常工作时

pc_wb_o[31:0]、opcode_wb_o[31:0]、operand_ra_wb_o[31:0]、operand_rb_wb_o[31:0]、exception_wb_o[31:0]、csr_wdata_wb_o[31:0]、ctrl_wb_q[31:0]等信号分别由 pc_e2_q[31:0]、opcode_wb_q[31:0]、operand_ra_e2_q[31:0]、operand_rb_e2_q[31:0]、exception_e2_r[31:0]、csr_wdata_e2_q[31:0]\ctrl_e2_q[31:0]的值赋予。

6. 流水线中发生异常/中断时

当系统中存在异常时，冲洗掉模块中的信号,ctrl_wb_o[9:0]接受 ctrl_e2_q[9:0]中除 RD_VALID 位以外的信号。valid_wb_q 信号在 E2 模块中发生 load 为对准、load 失败、store 未对准、store 失败、page fault load 和 page fault store 等异常时置零。

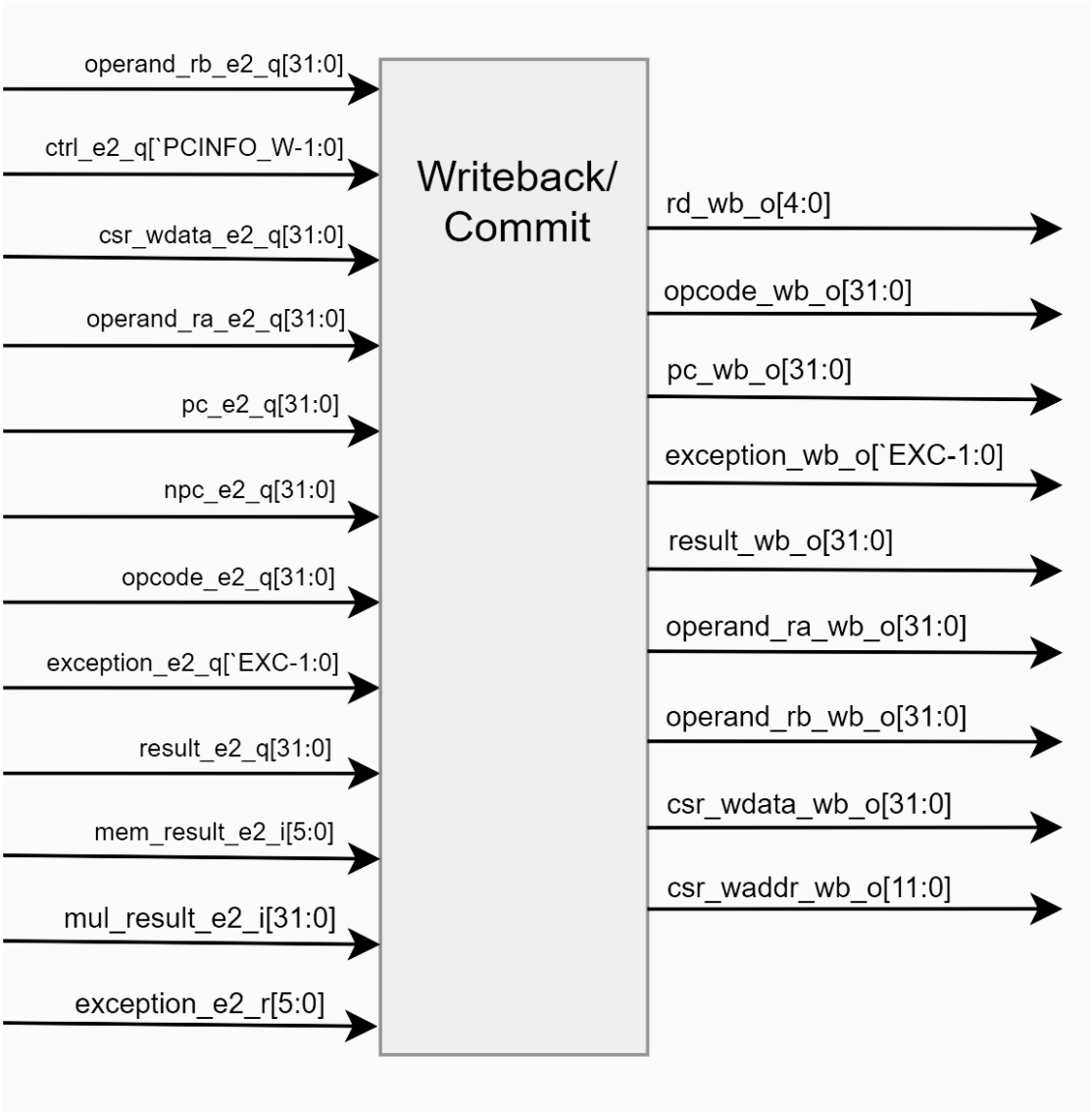


图 6.14 Writeback/Commit 示意图

输出信号 rd_wb_o[4:0]的值由 vslid_wb_o、ctrl_wb_q[9:0]信号的 RD_VALID 位、stall_o 的非信号的逻辑与运算的 5 位拓展值与 opcode_wb_q[31:0]信号的[11:7]位信号的与运算结果得出。

6.1.6 分支预测更新单元

分支预测更新单元的主要工作是把最新的跳转信息更新给前端的 BPU 单元。BPU 接收到来自 Issue 的最新跳转信息就可以对饱和计数器和 RAS 做相应的更改。下图是分支预测更新单元的代码示例。如图所示，分支预测更新单元会根据 pipe0 和 pipe1 的执行情况，选择对应的数据，向 BPU 发送。

```
//-------------------------------------------------------------
// Branch predictor info
//-------------------------------------------------------------
// This info is used to learn future prediction, and to correct
// BTB, BHT, GShare, RAS indexes on mispredictions.
assign branch_info_request_o      = mispredicted_r;
assign branch_info_is_taken_o     = (pipe1_branch_e1_w & branch_exec1_is_taken_i)     | (pipe0_branch_e1_w & branch_exec0_is_taken_i);
assign branch_info_is_not_taken_o = (pipe1_branch_e1_w & branch_exec1_is_not_taken_i) | (pipe0_branch_e1_w & branch_exec0_is_not_taken_i);
assign branch_info_is_call_o      = (pipe1_branch_e1_w & branch_exec1_is_call_i)      | (pipe0_branch_e1_w & branch_exec0_is_call_i);
assign branch_info_is_ret_o       = (pipe1_branch_e1_w & branch_exec1_is_ret_i)       | (pipe0_branch_e1_w & branch_exec0_is_ret_i);
assign branch_info_is_jmp_o       = (pipe1_branch_e1_w & branch_exec1_is_jmp_i)       | (pipe0_branch_e1_w & branch_exec0_is_jmp_i);
assign branch_info_source_o       = (pipe1_branch_e1_w & branch_exec1_request_i)      ? branch_exec1_source_i : branch_exec0_source_i;
assign branch_info_pc_o           = (pipe1_branch_e1_w & branch_exec1_request_i)      ? branch_exec1_pc_i     : branch_exec0_pc_i;
```

图 6.15 分支预测更新单元代码示例

7 MMU 单元介绍

MMU 单元待修改

8 指令缓存单元介绍

8.1 简介

I-Cache 采取两路组相连结构，分为 Tag 部分和 Data 部分。

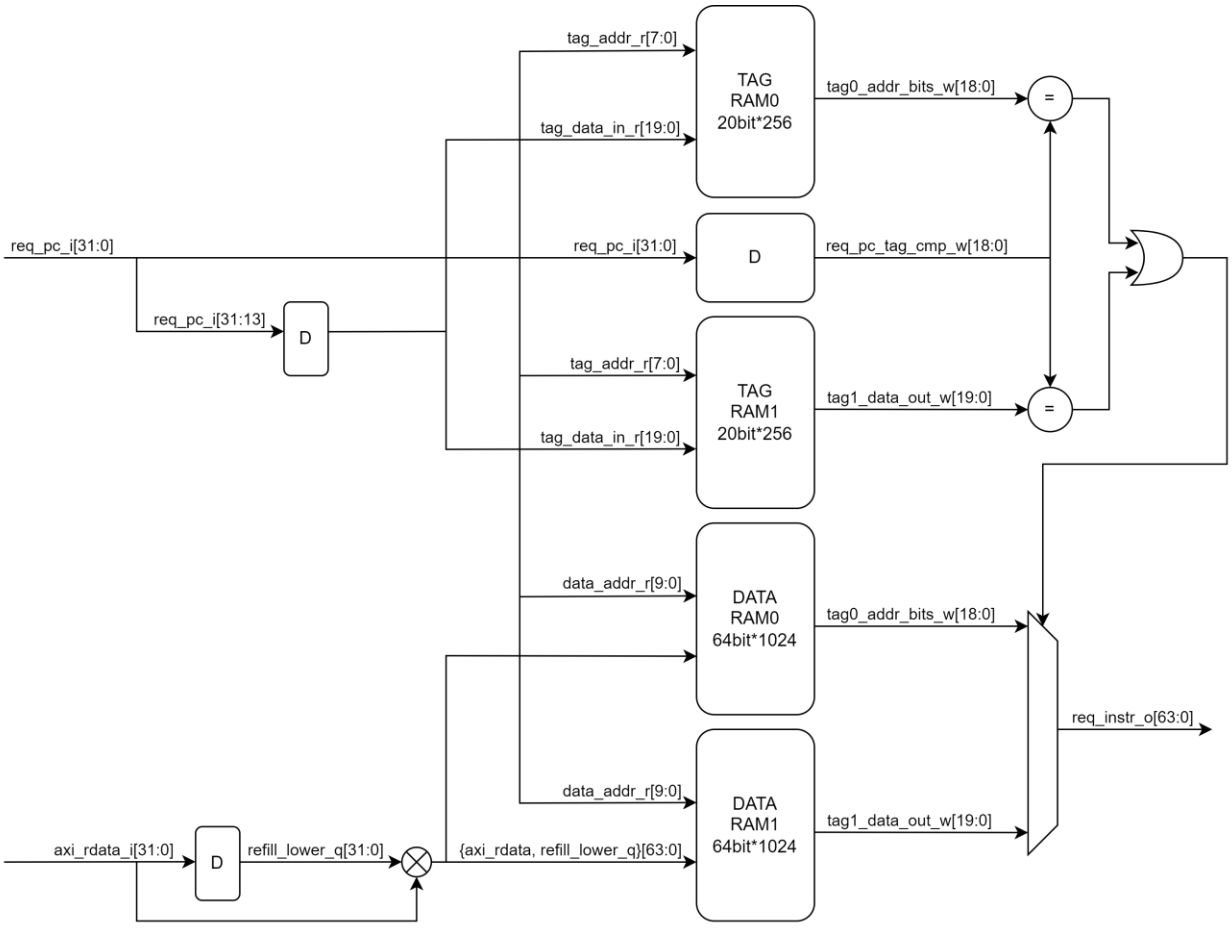


图 8.1 I-Cache 数据通路

8.1.1 TAG RAM

Tag RAM 的深度为 256，数据宽度为 20bit，包含 1 位 valid 和 19 位的 Tag，这里的 Tag 是指 req_pc_i 的[31:13]位。req_pc_i 的[12:5]位作为 index，用来在 Tag RAM 中寻址。用 index 寻址后分别在两路 Tag RAM 中比较 Tag 的值，如果有一个相同则代表 hit 成功。

8.1.2 Data RAM

Data RAM 的深度为 1024，数据宽度为 64bit，使用 PC 的[12:3]寻址。

9 数据缓存单元介绍

9.1 简介

整 0 个 D-cache 分为 5 个模块：Core、Uncache、Mux、Pmem Mux、AXI。
Mux：对输入的数据进行选择与控制。
Core 模块：使用 cache 时候的数据模块，包含两个 DATA RAM 和 TAG RAM。
Uncache 模块：不使用数据时候的数据模块。
Pmem Mux：对输出信号进行选择与控制。
AXI：AXI 总线控制模块。

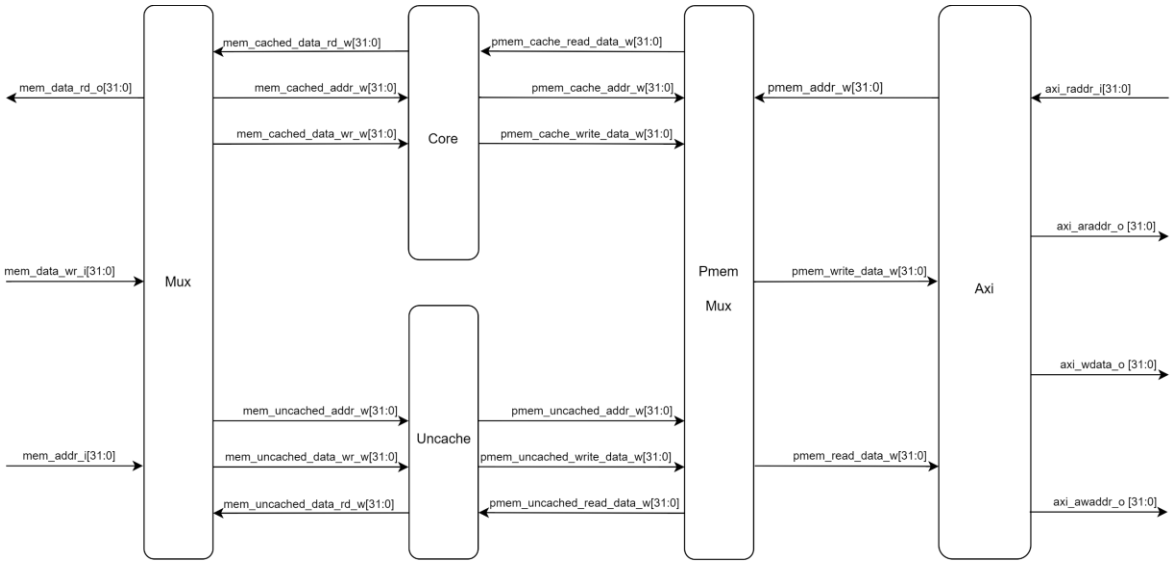


图 9.1 D-Cache 数据通路

表 9.1 端口列表

Name	Description
axi_rdata_i [31:0]	从外部(L2cache 或者存储器)输入的读数据
mem_addr_i [31:0]	由处理器内部 MMU 模块输入的地址
mem_data_wr_i [31:0]	由处理器内部 MMU 模块输入的写数据
axi_wdata_o [31:0]	写数据输出
axi_araddr_o [31:0]	读地址输出
axi_awaddr_o [31:0]	写地址输出
mem_data_rd_o [31:0]	命中后直接读取输出数据输出到 MMU 中

10 处理器状态转换

S 模式转换 M 模式：

- Trap 发生
- 查询 medeleg/mideleg 寄存器，判断是否需要委托当前异常/中断
- 设置 mepc 为当前 pc 值，mepc 为 mret 跳转地址
- 设置 mcause 寄存器，写入当前异常/中断错误代码
- 设置 mstatus.MIE 为 0，禁用中断
- 设置 mstatus.MPP 为 01，代表原始特权等级为 Supervisor
- 设置 mstatus.MPIE 为 1，代表原始中断使能为 1
- 设置 mtval 寄存器为 0 或由软件设置为相关信息
- 读取 mtvec 寄存器，赋值给 pc 寄存器
- 进入 Machine 模式

10.1 M 模式寄存器

10.1.1 Machine ISA Register (misa)

misa CSR 是一个 WARL 读写寄存器，报告 hart 支持的 ISA。该寄存器在任何实现中都必须是可读的，但是可以返回 0 值来表示未实现 misa 寄存器，如果 misa 为零，寄存器的宽度则有一个固定值。

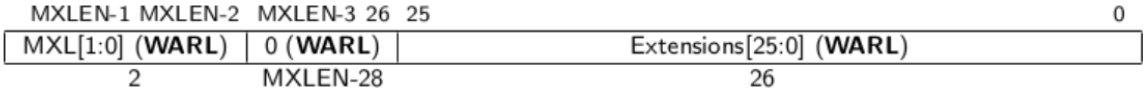


图 10.1 misa 寄存器

MXL[1:0]字段编码了 M 模式下寄存器宽度（32，64，128），MXL 部分的编码如下表。

表 10.1 MXL 部分的编码

MXL	XLEN
1	32
2	64
3	128

如果对 misa 的写入导致 MXLEN 发生变化，则 MXL 的位置移动到 misa 在新宽度处的最重要的两位。如果 misa 为零，也可以通过将立即数 4 放入寄存器中，然后每次将寄存器向左移动 31 位来找到基宽。如果一次移位后为零，则宽度为 RV32。如果两次移位后为零，则宽度为 RV64，否则为 RV128。Extension[25:0] 字段对标准扩展名的存在进行编码，每个字母使用一个 bit，bit0 代表“A”扩展，bit1 代表“B”扩展，以此类推，bit25 代表“Z”扩展。如果“I”扩展和“E”扩展同时被使能，那么“I”扩展将被采用，“E”扩展将被忽略。如果支持 User 和 Supervisor 模式，“U”和“S”位将分别被置 1。如果有任何非标准扩展，则 “X”位置 1。

Extension 字段的每一位的详细对应关系如下表。

表 10.2 Extension 字段

Bit	Character	Description
0	A	Atomic extension
1	B	Tentatively reserved for Bit-Manipulation extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Reserved
7	H	Hypervisor extension
8	I	RV32I/64I/128I base ISA
9	J	Tentatively reserved for Dynamically Translated Languages extension
10	K	Reserved
11	L	Reserved
12	M	Integer Multiply/Divide extension
13	N	Tentatively reserved for User-Level Interrupts extension
14	O	Reserved
15	P	Tentatively reserved for Packed-SIMD extension
16	Q	Quad-precision floating-point extension
17	R	Reserved
18	S	Supervisor mode implemented
19	T	Reserved
20	U	User mode implemented
21	V	Tentatively reserved for Vector extension
22	W	Reserved
23	X	Non-standard extensions present
24	Y	Reserved
25	Z	Reserved

“E”位为只读。除非 misa 全为零，否则“E”位总是与“I”位相反。同时支持 RV32E 和 RV32I 的实现可以通过 “I”位置 0 来选择 RV32E。

如果一个 ISA 扩展 x 依赖于一个 ISA 扩展 y，那么尝试启用扩展 x 而禁用扩展 y 会导致两个扩展都被禁用。例如，设置“F”=0，“D”=1，结果将禁用“F”和“D”。

10.1.2 Machine Vendor ID Register (mvendorid)

mvendorid CSR 是一个 32 位只读寄存器，提供核心提供商的 JEDEC 制造商 ID。这个寄存器在任何实现中都必须是可读的，但是可以返回 0 值来表示该字段没有实现，或者这是一个非商业实现。

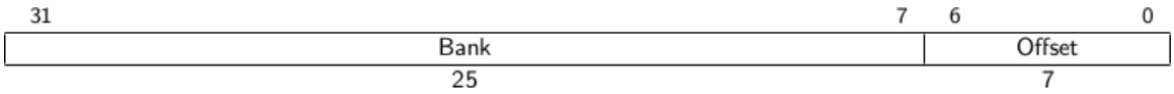


图 10.2 mvendorid 寄存器

10.1.3 Machine Architecture ID Register (marchid)

marchid CSR 是一个 mxlen 位的只读寄存器，编码 hart 的基本微体系结构。该寄存器在任何实现中都必须是可读的，但是可以返回 0 值来表示未实现该字段。mvendorid 和 marchid 的组合应该唯一地标识所实现的 hart 微架构的类型。

图 10.3 marchid 寄存器

10.1.4 Machine Implementation ID Register (mimpid)

mimid CSR 提供了处理器实现版本的唯一编码。这个寄存器在任何实现中都必须是可读的，但是可以返回 0 值来表示该字段没有实现。实现值应该反映 RISC-V 处理器本身的设计，而不是任何周围的系统。

图 10.4 mimpid 寄存器

10.1.5 Hart ID Register (mhartid)

hartid CSR 是一个 mxlen 位的只读寄存器，包含运行代码的硬件线程的整数 ID。这个寄存器在任何实现中都必须是可读的。在多处理器系统中，Hart ID 不一定连续编号，但至少有一个 Hart ID 必须为零。Hart id 在执行环境中必须是唯一的。

10.1.6 Machine Status Registers (mstatus)

mstatus 寄存器是一个 mxlen 位可读写寄存器，RV32 和 RV64 的格式分别如图所示。状态寄存器跟踪并控制 hart 的当前运行状态。

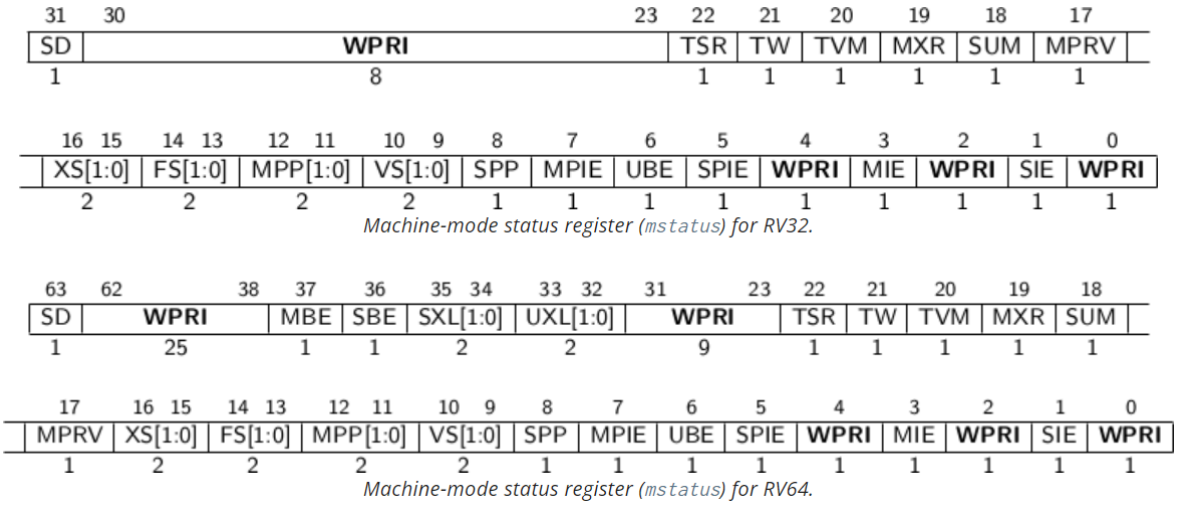


图 10.5 mstatus 寄存器

mstatush 是一个 32 位读写寄存器，仅存在于 RV32 中。mstatush 的 30:4 位通常包含与 RV64 的 mstatus 的 62:36 位相同的字段。mstatush 中不存在 SD、SXL 和 UXL 字段。

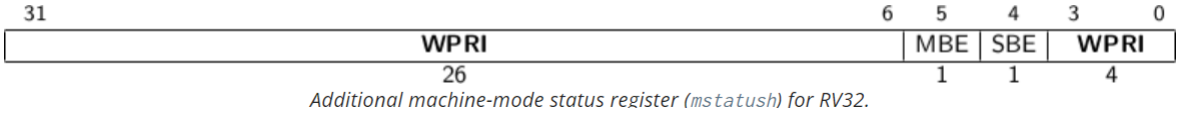


图 10.6 mstatus 寄存器

1. 特权和全局中断使能堆栈 MIE 和 SIE 位

分别为 M 模式和 S 模式提供全局中断使能。这些位主要用于保证当前特权模式下中断处理程序的原子性。

当 hart 以特权模式 x 执行时，当 xIE=1 时全局启用中断，当 xIE=0 时全局禁用中断。无论为低特权模式设置了任何全局 wIE 位,低特权模式的中断 w<x 总是全局禁用的。无论高特权模式的全局 yIE 位设置如何，高特权模式的中断 y>x 始终是全局启用的。

在将控制权交给低特权模式之前，高特权级别的代码可以使用单独的每个中断启用位来禁用所选的高特权模式中断。xPIE 保留在 trap 发生之前激活的中断启用位的值，xPP 保留之前的特权模式。MPP 的值是两位宽（代表 M=11，S=01，U=00），SPP 是一位宽（S=1，U=0）。xPP 字段是 WARL 字段，只能包含特权模式 x 和任何实现的特权模式低于 x。如果没有实现特权模式 x，则 xPP 必须为只读 0。

MRET 或 SRET 指令分别用于从 m 模式或 s 模式返回。在执行 xRET 指令时，假设 xPP 值是 y，则将 xIE 置为 xPIE;将特权模式变为 y;xPIE 设置为 1; xPP 设置为支持的最小特权模式(如果支持 U-

mode，则为 U，否则为 M)。xPP≠M 时，xRET 也设置 MPRV=0。对于 RV64 系统，SXL 和 UXL 字段是分别控制 S 模式和 U 模式下 XLEN 值的 WARL 字段。这些字段的编码与 misa 的 MXL 字段相同，如下表所示。S 模式和 U 模式下的有效 XLEN 分别称为 SXLEN 和 UXLEN。

表 10.3 MXL 字段

SXL/UXL	XLEN
1	32
2	64
3	128

对于 RV32 系统，SXL 和 UXL 字段不存在，并且 SXLEN=32 和 UXLEN=32。对于 RV64 系统，如果不支持 S 模式，则 SXL 为只读零。否则，它是一个编码 SXLEN 当前值的 WARL 字段。具体来说，可以使 SXL 成为只读字段，其值始终确保 SXLEN=MXLEN。

对于 RV64 系统，如果不支持 U 模式，则 UXL 为只读零。否则，它是一个编码 UXLEN 当前值的 WARL 字段。具体来说，实现可以使 UXL 成为只读字段，其值始终确保 UXLEN=MXLEN 或 UXLEN=SXLEN。

无论在何种模式下，只要将 XLEN 设置为小于所支持的最宽 XLEN 的值，所有操作都必须忽略配置的 XLEN 以上的源操作数寄存器位，并且必须对结果进行符号扩展以填充目标寄存器中所支持的整个最宽 XLEN。类似地，大于 XLEN 的 pc 位被忽略，当写入 pc 时，它被符号扩展以填充支持的最宽 XLEN。

2. 存储特权 MPRV(Modify PRiVilege)位

修改有效特权模式，即执行加载和存储时的特权级别。当 MPRV=0 时，加载和存储正常运行，使用当前特权模式的转换和保护机制。当 MPRV=1 时，加载和存储指令的内存地址将按照当前特权模式（MPP）进行地址转换和保护，并且应用字节序。这意味着当 MPRV 为 1 时，加载和存储指令将使用 MPP 特权模式的地址转换和保护机制，并且按照 MPP 的字节序进行数据操作。如果不支持 U 模式，则 MPRV 位是只读的，并且始终为 0。如果 MRET 或 SRET 指令将特权模式更改为特权低于 M 的模式，也会设置 MPRV=0。

当前特权模式和 MPP 指定的特权模式可能具有不同的 XLEN 设置。当 MPRV=1 时，加载和存储内存地址被视为当前 XLEN 被设置为 MPP 的 XLEN。MXR(Make eXecutable Readable)位修改加载访问虚拟内存的权限。当 MXR=0 时，只可以读取标记为可读的页。当 MXR=1 时，可以读取标记为可读或可执行(R=1 或 X=1)的页。当基于页的虚拟内存无效时，MXR 不起作用。如果不支持 S 模式，则 MXR 为只读 0。SUM(允许 Supervisor User Memory access)位修改 S 模式加载和存储访问虚拟内存的权限。当 SUM=0 时，S 模式访问 U 模式可访问的页将出错。当 SUM=1 时，允许这些访问。
当基于页的虚拟内存无效时，SUM 不起作用。

虽然 SUM 在不以 S 模式执行时通常会被忽略，但当 MPRV=1 且 MPP=S 时，它会生效。如果不支持 s 模式，SUM 为只读 0。MODE 为只读 0。

3. 端序控制 MBE、SBE 和 UBE 位

是 WARL 字段，它们控制内存访问(指令获取除外)的端序。指令读取总是小端顺序的。MBE 控制 M 模式(假设 mstatus.MPRV=0)进行的非指令获取内存访问是小端(MBE=0)还是大端(MBE=1)。

SBE 控制 S 模式进行的显式读取和存储是小端(SBE=0)还是大端(SBE=1)。如果不支持 S 模式，则 SBE 为只读 0。

UBE 控制 U 模式进行的显式读取和存储是小端(UBE=0)还是大端(UBE=1)。如果不支持 U 模式，则 UBE 为只读 0。

对于隐式访问超级级别的内存管理数据结构（例如页表），字节序始终由 SBE 控制。当改变 SBE 时，会影响对这些数据结构的解释方式。如果在 SBE 发生变化时仍然使用这些数据结构，M 模式的软件必须执行一条 SFENCE.VMA 指令，并将 rs1 和 rs2 都设置为 x0，以确保内存管理数据结构的一致性。

如果支持 S 模式，则实现可以使 SBE 成为 MBE 的只读副本。如果支持 U 模式，则实现可以使 UBE 成为 MBE 或 SBE 的只读副本。

4. 虚拟存储控制 TVM（Trap Virtual Memory）位

是一个 WARL 字段，它支持拦截监控虚拟内存管理操作。当 TVM=1 时，在 S 模式下尝试读写 satp CSR 或执行 SFENCE.VMA 或 SINVAL.VMA 指令执行时将引发非法指令异常。当 TVM=0 时，在 S 模式下允许这些操作。不支持 S 模式时，TVM 为只读 0。

5. TW(Timeout Wait)位

是一个 WARL 字段，它支持拦截 WFI 指令。当 TW=0 时，如果不被其他原因阻止，WFI 指令可以在较低权限模式下执行。当 TW=1 时，如果 WFI 在任何低特权模式下执行，并且它没有在特定的有限时间内完成，则 WFI 指令会导致非法指令异常。时间限制可能一直为 0，在这种情况下，当 TW=1 时，WFI 总是在较低特权模式下导致非法指令异常。没有比 M 模式更低的模式时，TW 为只读 0。如果存在 S 模式，在 U 模式下执行 WFI 会导致非法指令异常，除非它在特定于实现的有限时间限制内完成。

6. TSR (Trap SRET)位

是一个 WARL 字段，它支持拦截异常返回指令 SRET。当 TSR=1 时，在 S 模式下执行 SRET 将引发非法指令异常。当 TSR=0 时，表示允许。扩展上下文状态

FS[1:0]和 VS[1:0] WARL 字段和 XS[1:0]只读字段分别用于通过设置和跟踪浮点单元和任何其他用户模式扩展的当前状态来降低上下文保存和恢复的成本。FS 字段编码浮点单元状态的状态，包括浮点寄存器 f0-f31 和 fcsr、frm 和 fflags。VS 字段用于编码向量扩展状态的状态，包括向量寄存器 v0 ~ v31 和 csr 寄存器 vcsr、vxrm、vxsat、vstart、vl、vtype 和 vlenb。XS 字段对其他用户模式扩展的状态和相关状态进行编码。

上下文切换例程可以检查这些字段，以快速确定是否需要状态保存或恢复。如果需要保存或恢复，则通常需要额外的指令和 csr 来影响和优化该流程。

FS、VS 和 XS 字段使用下表的状态编码，四个可能的状态值分别是 Off、Initial、Clean 和 Dirty。根据浮点状态位，可以判断上下文切换的时候，是否需要保存浮点相关寄存器。 当扩展的状态设置为 Off 时，任何试图读取或写入相应状态的指令都将导致非法指令异常。当状态为 Initial 时，对应的状态应有一个初始常数值。当状态为 Clean 时，相应的状态可能与初始值不同，但与存储在上下文交换中的最后一个值匹配。当状态为 Dirty 时，对应的状态自上次上下文保存以来可能已经被修改。

表 10.4 状态编码

Status	FS and VS Meaning	XS Meaning

0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

Current State Action	Off	Initial	Clean	Dirty
At context save in privileged code				
Save state? Next state	No Off	No Initial	No Clean	Yes Clean
At context restore in privileged code				
Restore state? Next state	No Off	Yes, to initial Initial	Yes, from memory Clean	N/A N/A
Execute instruction to read state				
Action? Next state	Exception Off	Execute Initial	Execute Clean	Execute Dirty
Execute instruction that possibly modifies state, including configuration				
Action? Next state	Exception Off	Execute Dirty	Execute Dirty	Execute Dirty
Execute instruction to unconfigure unit				
Action? Next state	Exception Off	Execute Initial	Execute Initial	Execute Initial
Execute instruction to disable unit				
Action? Next state	Execute Off	Execute Off	Execute Off	Execute Off
Execute instruction to enable unit				
Action? Next state	Execute Initial	Execute Initial	Execute Initial	Execute Initial

图 10.7 状态转换图

SD 位是一个只读位，它总结 FS、VS 或 XS 字段是否表示存在一些需要将扩展用户上下文保存到内存中的脏状态。当 FS、VS 或 XS 位编码 Dirty 状态(即 SD=(FS==11)或(XS==11)或(VS==11))时设置。这允许特权代码快速确定除了整数寄存器集和 PC 之外何时不需要额外的上下文保存。
如果 FS、XS 和 VS 都是只读零，那么 SD 也总是零。

10.1.7 Machine Trap-Vector Base-Address Register (mtvec)

mtvec 寄存器是一个 mxlen 位的 WARL 读/写寄存器，它保存 trap 向量配置，包括一个向量基址(base)和一个向量模式(mode)。

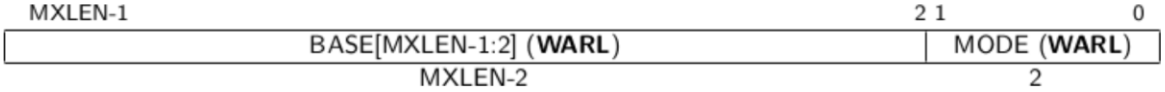


图 10.8 mtvec 寄存器

当 MODE=Direct 时，所有进入机器模式的陷阱都会导致 pc 被设置为 BASE 字段中的地址。当 MODE= vector 时，所有进入机器模式的同步异常导致 pc 被设置为 BASE 字段中的地址，而中断导致

pc 被设置为 BASE 字段中的地址加上中断原因编号的四倍。例如，机器模式定时器中断导致 pc 被设置为 BASE+0x1c。

表 10.5 MODE 字段表

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	Reserved

10.1.8 Machine Trap Delegation（medeleg/mideleg）

默认状态下，所有 trap 指令都由机器模式处理。尽管机器模式的 trap handler 可以利用 MRET 指令将 trap 重新委托给更低特群等级。利用 medeleg 和 mideleg 当中的读写位可以指定特定的异常和中断由特定的特权等级执行。当一个 Trap 被委托给 S 模式处理时我们需要进行如下操作：
需要将异常或中断错误代码写入 scause 寄存器。将发生异常 pc 地址写入 sepc 寄存器。stval 寄存器写入异常处理相关信息。mstatus.SPP 写入 Trap 发生时，CPU 的特权等级将 mstatus.SIE 的值写入到 mastatus.SPIE 清空，mstatus.SIE。mcause 寄存器，mepc 寄存器，mtval 寄存器，mstatus.MPP，mstatus.MPIE 不需要写入。

medeleg 和 mideleg 寄存器不应该由任何只读比特位，任何支持被委托的 Trap 也应该支持不被委托。需要注意的是，Trap 不允许由高特权等级向低特权等级传送。例如，如果 M 模式将 Trap 委托给了 S 模式，之后在 M 模式下运行的软件触发了非法指令异常，那么此时 Trap 应在 M 模式处理。相同的情况下，如果 S 模式下运行的软件触发了非法指令异常，那么 Trap 应该在 S 模式执行。被委托的中断会被发出委托的等级忽略，例如，如果 Supervisor timer interrupt（STI）被 M 模式委托给了 S 模式。那么当机器在 M 模式运行的时候，STI 请求将被忽略。相反，如果 STI 没有被委托，那么无论机器在任何模式运行的时候，STI 请求都会被接受，并且将机器模式转为 M 模式。

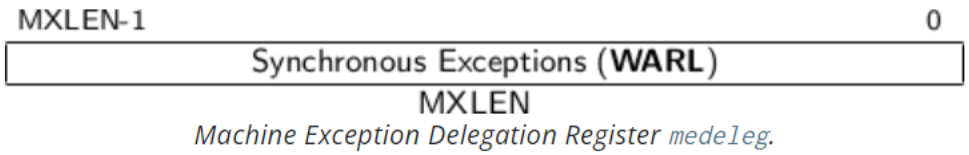


图 10.9 Medeleg 寄存器

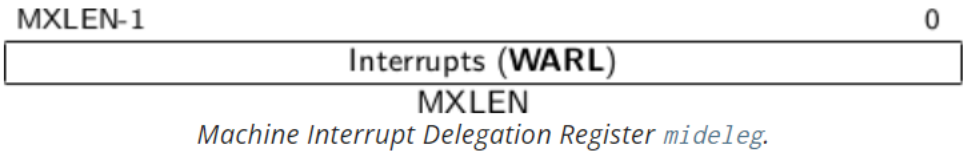


图 10.10 Mideleg 寄存器

medeleg 寄存器利用每一个比特的索引值来代表异常的错误代码。通过比较 medeleg 的索引值和 mcause 的返回值，来判断当前异常是否被委托给了其他特权等级。mideleg 寄存器的比特位排布和 mip 寄存器的比特位排布相同。例如 STIP 在 mip 当中由比特位 5 控制，在 mideleg 当中同样有比特位 5 控制。

10.1.9 Machine Interrupt Register（mip/mie）

mip 寄存器存储了等候中的中断请求的信息。mie 寄存器存储了中断使能比特位。mip 与 mie 寄存器当中的比特位对应 mcause 当中相同的比特位。[15:0]位为普通终端使用，[31:16]位为用户自定义。

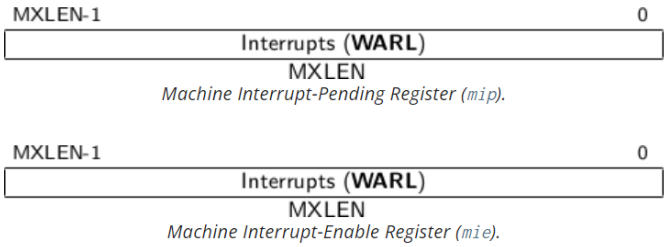


图 10.11 mie/mip 寄存器

当满足以下情况是，中断请求信号会陷入到 M 模式当中：

当前机器运行模式为 M，mstatus.MIE 为 1 或者当前特权等级小于 M 模式 mie 和 mip 当中，对应当前中断请求的信号为 1。当前中断请求信号在 mideleg 当中没有被委托需要注意的是，当终端在 mip 当中等候时，需在有限时间内对当前中断是否可以 trap 做出判断。在 xRET 指令执行后立刻判定。陷入到 M 模式的中断请求比陷入到其他模式当中的中断请求拥有更高优先级。mip 寄存器当中的任何一个 bit 都可以设置为可写或只读。假设 mip 寄存器当中的 bit i 是可写的，那么等候中的中断请求 i 可以通过将 mip 当中的 bit i 置零来清除。如果中断请求 i 是等候状态但是 mip 中的 bit i 是只读的，那么具体架构实现必须提供其他方法来清除等候中的中断请求。如果一个中断请求可以被 pending，那么 mie 当中对应的 bit 位就可以置为可写状态。mie 当中的只读位必须设置为只读 0。

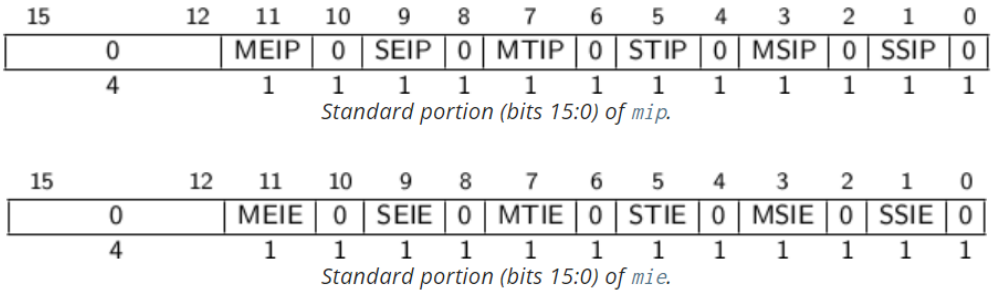
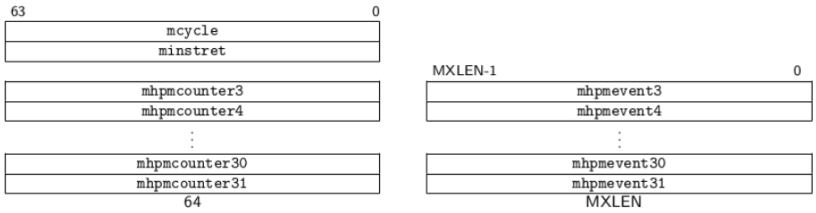


图 10.12 mip/mie 寄存器 bit 位

mip.MEIP 和 mip.MEIE 分别对应机器模式中外部中断的 pending 和 enable。其中 mip.MEIP 为只读，只能被特定的中断控制器赋值和清除。mip.MEIP 和 mie.MEIE 分别对应时钟中断的中断等候和中断使能信号 mip.MTIP 为只读位，可以通过写入 memory-mapped 机器模式时钟对比寄存器来清除。mip.MSIP 和 mie.MSIE 是分别对应机器模式的软件中断等候和中断使能信号。mip.MSIP 为只读信号。可通过 memory-mapped 控制寄存器写入。通常在多 hart 场景下用来提供 machine-level 的处理器内部中断信号。每一个 hart 可以通过相同的 memory-mapped 控制寄存器来写入 mip.MSIP。对于单 hart 系统来说，或者对于一个支持通过外部中断来发送 machine-level 中断的架构来说，mip.MSIP 和 mie.MSIE 可以同时为只读恒零。如果架构不支持 supervisor 模式，mip.SEIP，mip.STIP，mip.SSIP，mie.SIE，mie.STIE，mie.SSIE 可以为只读恒零。如果架构支持 supervisor 模式 mip.SEIP 和 mie.SEIE 分别为 supervisor-level 的外部中断等候和使能信号。其中 mip.SEIP 为可写位，也可以通过机器模式软件写入。除此之外，架构层面的中断控制器也可以产生 supervisor-level 的外部中断信号。Supervisor-level 的外部中断通过将软件可写的 mip.SEIP 位和外部中断控制器信号进行 logical-OR 操作来实现终端等候。当 mip 寄存器通过 CSR 指令读取的时候，mip.SEIP 通过 rd 寄存器发挥的值是软件可写位和中断控制器信号的 logical-OR 结果。但是中断控制器信号并不参与计算 SEIP 的写入值。只有软件可写 mip.SEIP 位参与 CSRRS 指令和 CSRRC 指令的 read-modify-write 操作。当 supervisor 模式启用的时候，mip.STIP 和 mie.STIE 分别对应 supervisor-level 时钟中断的终端等候和中断使能信号。mip.STIP 为只读位，同时也可以通过 M 模式软件写入来向 S 模式发送时钟中断信号。当 supervisor 模式启用的时候，mip.SSIP 和 mip.SSIE 分别对应 supervisor-level 的软件中断等候和中断使能信号。其中 mip.SSIP 为可写位，同时也可以被 platform 中断控制器置 1。多个 M 模式同时中断通过以下优先级处理：MEI，MSI，MTI，SEI，SSI，STI。如果中断被委托给 S 模式，那么此中断在 sip 寄存器可被发现，也可以通过 sie 寄存器忽略。除此之外，sip 和 sie 中对应的 bit 位为只读恒零。

10.1.10 Hardware Performance Monitor

M 模式包括基本的硬件性能监测单元。mcycle 寄存器存储了当前处理器的周期数。minstret 存储了退休指令的数量。mcycle 和 minstret 寄存器在 rv32 和 rv64 系统当中均有 64-bit 精度。计数器寄存器在 hart 重置时候会有一个随意数值，也可以被写入指定数值。任何 CSR 写入都会在写入指令完成后生效。mcycle 寄存器可以在同一个 core 的不同 hart 之间共享，这种情况下，写入 mcycle 的操作也会被不同 hart 看到。Platform 应支持查询 hart 和 mcycle 的共享关系。硬件性能监测单元包括 29 个额外的事件计数器：mhpmcounter3-mhpmcounter31，事件选择 CSR：mhpmevent3-mhpmevent31。这些寄存器负责控制事件和对应计数器的增长关系。这些事件的定义是由 platform 决定的。但是事件 0 代表没有事件。所有计数器都应该被实现，一个符合规定的实现方法就是将计数器及其对应的时间选择器置为只读恒零。



Hardware performance monitor counters.

图 10.13 硬件性能监控计数器

Mhpcounter 为 WARL 寄存器，在 rv32 和 rv64 系统中最高支持 64bit 精度。当寄存器位宽为 32 位时，读取 mcycle，minstret，mhpmcountern 将返回 31-0 位的值。同时写操作也将写入 31-0 位。读取 mcycleh，minstreth，mhpmcounterh 将返回 63-32 位，写操作将写入 63-32 位。

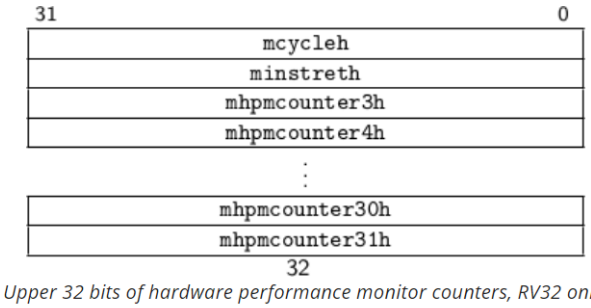


图 10.14 上部 32 位硬件性能监控计数器

10.1.11 Machine Counter-Enable Register（mcounteren）

Counter-enable 寄存器 mcountern 是一个 32 位寄存器。主要负责控制更低特权等级的硬件监测计数器的使能信号。

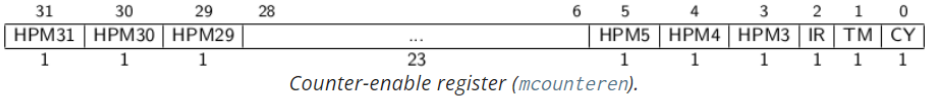


图 10.15 mcounteren 寄存器

Mcounteren 寄存器的设置之负责控制 accessibility。读操作和写操作并不会影响潜在的计数器。计数器数值会持续增加无论是否 accessible。当 CY，TM，IR 或 HPMn 比特位被清空，任何在 U 模式或 S 模式尝试读取 cycle，time，instret，或 hpmcountern 寄存器的操作将会被视为违法指令异常。当其中任何一个比特位被置 1 后，对应的寄存器访问操作会在更低的特权等级被允许。cycle，instret，hpmcountern CSR 是 mcycle，minstret，mhpmcounter 的只读映射。同理，time CSR 是 memory-mapped mtime 寄存器的只读映射。以上映射关系在 RV32I 当中也是同理。timeh CSR 是 mtime 寄存器 63-32 位的只读映射，而 time 则是 31-0 位的只读映射。在实现了 U 模式的系统当中，mcounteren 寄存器必须存在。但是所有位都遵循 WARL，而且可能是只读恒零。这代表了当机器在更低特权等级运行时，读取相应计数器将导致非法指令异常。在没有实现 U 模式的系统当中，mcounteren 寄存器不应该存在。

10.1.12 Machine Counter-Inhibit Register（mcountinhibit）

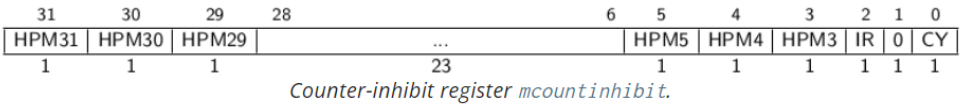


图 10.16 mcountinhibit 寄存器

mcountinhibit 是一个 32 位 WARL 寄存器。主要负责控制硬件监测寄存器数值的增长。寄存器的设置只负责控制寄存器数值增长与否。寄存器的 accessibility 并不会被 mcountinhibit 寄存器的设置所影响。当 CY，IR，或 HPMn 比特位被清除后，cycle，instre，hpmcountern 寄存器的数值增长不会被影响。当 CY，IR，HPMn 比特被置 1 后，相应寄存器的数值增长停止。mcycle 寄存器可以在同一个 core 上被不同的 hart 共享。在这种情况下，mcountinhibit.CY 也会被共享，写入 mcountinhibit.CY 也会被其他 hart 发现。如果 mcountinhibit 寄存器没有被实现，那么相应的操作会被置 0。

10.1.13 Machine Scratch Register（mscratch）

Mscratch 寄存器是一个 MXLEN 长度的读写寄存器。主要用于机器模式下，程序临时保存一些数据。

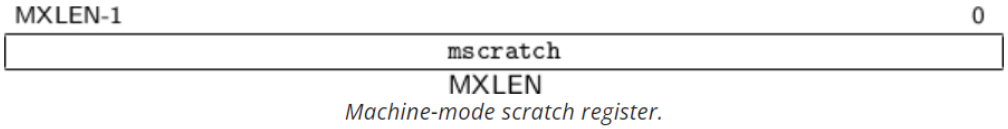


图 10.17 mscratch 寄存器

10.1.14 Machine Exception Program Counter (mepc)

mepc 是一个 MXLEN-bit 读/写寄存器，格式如图所示。mepc 的低位（mepc[0]）始终为零。在仅支持 IALIGN=32 的实现中，两个低位（mepc[1:0]）始终为零。

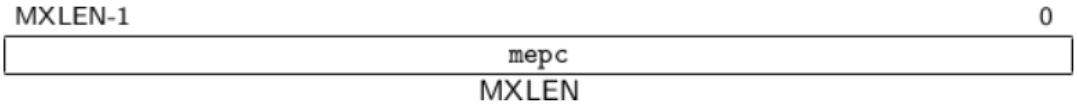


图 10.18 mepc 寄存器

如果一个实现允许 IALIGN 为 16 或 32（例如，通过更改 CSR misa），那么，每当 IALIGN=32 时，位 mepc[1]在读取时被屏蔽，从而看起来为 0。这种屏蔽也发生在 MRET 指令的隐式读取中。尽管被屏蔽，但当 IALIGN=32 时，mepc[1]仍然是可写的。

mepc 是一个 WARL 寄存器，必须能够保存所有有效的虚拟地址。它不需要能够保存所有可能的无效地址。在编写 mepc 之前，实现可能会将一个无效地址转换为 mepc 能够保持的其他无效地址。当地址转换无效时，虚拟地址和物理地址是相等的。因此，mepc 必须能够表示的地址集包括可以用作有效 pc 或有效地址的物理地址集。

当 trap 进入 M 模式时，mepc 将使用被中断或遇到异常的指令的虚拟地址写入。否则，mepc 永远不会实现编写，尽管它可能是由软件明确编写的。

10.1.15 Machine Cause Register (mcause)

mcause 寄存器是一个 MXLEN-bit 读写寄存器，其格式如图所示。当 trap 进入 M 模式时，mcause 会编写一个代码，指示导致 trap 的事件。否则，mcause 永远不会进行编写，尽管它可能是由软件显式编写的。

如果 trap 是由中断引起的，则会拉高 mcause 寄存器中的中断位。异常代码字段包含一个标识最后一个异常或中断的代码。表[mcauses]列出了可能的机器级异常代码。异常代码是一个 WLRL 字段，因此只能保证包含支持的异常代码。

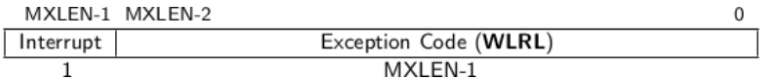


图 10.19 mcause 寄存器

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

图 10.20 中断异常编码

Load 和 load-reserved 指令会生成加载异常，而 store，store-conditional 和 AMO 指令会生成 store/AMO 异常。

中断可以通过 mcause 寄存器值符号上的单个分支与其他 trap 分离。左移可以移除中断位，并缩放异常代码以索引到陷阱向量表中。我们不区分特权指令异常和非法操作码异常。这简化了体系结构，还隐藏了实现支持哪些更高权限指令的细节。为 trap 提供服务的特权级别可以实现一个策略，决定是否需要区分这些操作码，如果需要，则将给定的操作码视为非法操作码还是特权操作码。

如果一条指令可能引发多个同步异常，则表[exception-priority]的优先级递减顺序指示在 mcause 中执行和报告哪个异常。任何自定义同步异常的优先级都是由具体实现定义的。

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
		During instruction address translation:
	12, 1	First encountered page fault or access fault
		With physical address for instruction:
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/store/AMO address breakpoint
		Optionally:
	4, 6	Load/store/AMO address misaligned
		During address translation for an explicit memory access:
	13, 15, 5, 7	First encountered page fault or access fault
		With physical address for an explicit memory access:
	5, 7	Load/store/AMO access fault
		If not higher priority:
Lowest	4, 6	Load/store/AMO address misaligned

图 10.21 异常优先级

当虚拟地址被转换为物理地址时，地址转换算法确定可能引发的特定异常。Load/store/AMO 地址未对齐异常的优先级可能高于或者低于 load/store/AMO 页面故障和访问故障异常。

Load/store/AMO 地址未对齐和页面错误异常的相对优先级是为了灵活地满足两个设计点而定义的。从不支持未对齐访问的实现可以无条件地引发未对齐地址异常，而无需执行地址转换或保护检查。仅支持对某些物理地址进行未对齐访问的实现必须转换并检查地址，然后才能确定是否可以继续进行未对齐的访问，在这种情况下，引发页面错误异常或访问更合适。

指令地址断点与数据地址断点（也称为观察点）和环境中断异常（由 EBREAK 指令引发）具有相同的原因值，但优先级不同。

指令地址未对齐异常是由目标未对齐的控制流指令引发的，而不是由获取指令的行为引发的。因此，这些异常的优先级低于其他指令地址异常。

10.1.16 Machine Trap Value Register (mtval)

mtval 寄存器是一个 MXLEN 位读写寄存器，格式如图所示。当 trap 进入 M 模式时，mtval 要么设置为零，要么写入异常特定信息，以帮助软件处理 trap。否则，mtval 永远不会由实现编写，尽管它可能是由软件显式编写的。硬件平台将指定哪些异常必须以信息方式设置 mtval，哪些可以无条件地将其设置为零。如果硬件平台指定没有异常将 mtval 设置为非零值，则 mtval 为只读零。

如果在指令获取、加载或存储中发生断点、地址未对齐、访问错误或页面错误异常时，mtval 是用非零值写入的，则 mtval 将包含出错的虚拟地址。

当启用基于页面的虚拟内存时，即使对于物理内存访问故障异常，也会使用错误的虚拟地址写入 mtval。这种设计降低了大多数实现的数据路径成本，尤其是那些具有硬件页表遍历器的实现。

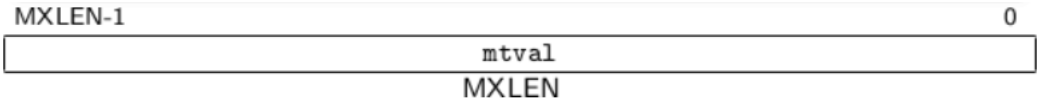


图 10.22 mtval 寄存器

如果在未对齐的加载或存储导致访问故障或页面故障异常时，mtval 是用非零值写入的，则 mtval 将包含导致故障的访问部分的虚拟地址。

如果在具有可变长度指令的系统上发生指令访问故障或页面故障异常时，mtval 是用非零值写入的，则 mtval 将包含导致故障的指令部分的虚拟地址，而 mepc 将指向指令的开头。

mtval 寄存器还可以选择性地用于在非法指令异常（mepc 指向内存中的错误指令）时返回错误指令位。如果在发生非法指令异常时，mtval 是用非零值写入的，则 mtval 将包含以下值中的最短值：实际错误指令。出错指令的前 ILEN 位。错误指令的前 MXLEN 位。

在非法指令异常时加载到 mtval 中的值是右对齐的，并且所有未使用的高位都被清除为零。在 mtval 中捕获出错的指令可以减少指令仿真的开销，从而在指令未对齐的情况下可以避免若干部分指令载入，以及在使用加载将指令取入数据寄存器时可能出现的 data cache 未命中或 slow uncached accesses 。在动态翻译系统中如果另一个代理正在操作指令内存，也存在原子性问题。一个要求是在捕获 trap 之前将整个指令（或至少第一个 MXLEN 位）提取到 mtval 中。这不应该限制实现，实现通常会在尝试解码指令之前获取整个指令，并避免使软件处理程序复杂化。

mtval 中的值为零表示不支持该功能，或者提取了非法的零指令。来自 mepc 指向的指令内存的加载可以用于区分这两种情况（或者，可以询问系统配置信息，以便在运行时之前安装适当的陷阱处理）。对于其他 trap，mtval 设置为零，但未来的标准可能会重新定义 mtval 对其他陷阱的设置。如果 mtval 不是只读零，那么它是一个 WARL 寄存器，必须能够保存所有有效的虚拟地址和零值。它不需要能够保存所有可能的无效地址。在写入 mtval 之前，实现可以将无效地址转换为 mtval 能够保存的其他无效地址。如果实现了返回错误指令位的功能，mtval 还必须能够保持小于 2N 的所有值，其中 N 是 MXLEN 和 ILEN 中的较小值。

10.1.17 Machine Configuration Pointer Register (mconfigptr)

mconfigptr 是一个 MXLEN 位只读 CSR，格式如图所示，它保存配置数据结构的物理地址。软件可以遍历此数据结构，以发现有关 harts、平台及其配置的信息。

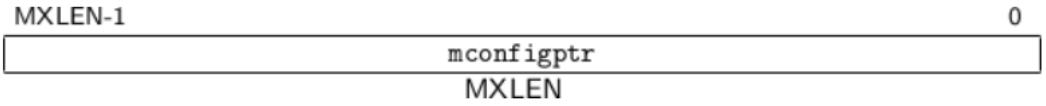


图 10.23 mconfigptr 寄存器

以位为单位的指针对齐必须不小于支持的最大 MXLEN：即，如果支持的最大 MX LEN 为 8 × n、则 mconfigptr[log2n-1:0]必须为零。

必须实现 mconfigptr，但它可能为零，表示配置数据结构不存在，或者必须使用替代机制来定位它。配置数据结构的格式和模式尚未标准化。虽然 mconfigptr 在某些实现中只是硬接线的，但其他实现可能提供一种方法来配置 CSR 读取时返回的值。例如，mconfigptr 可能会在引导过程开始时显示由平台或 M 模式软件编程的内存映射寄存器的值。

10.1.18 Machine Environment Configuration Registers (menvcfg and menvcfgh)

menvcfg CSR 是一个 MXLEN 位读/写寄存器，格式化为 MXLEN=64，如下图所示，用于控制特权低于 M 的模式的执行环境的某些特性。

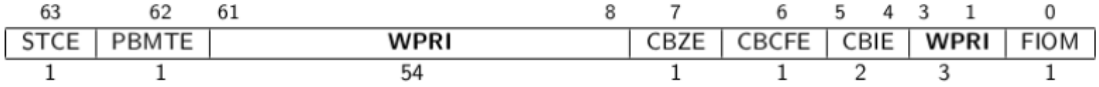


图 10.24 menvcfg 寄存器（MXLEN=64）

如果在 menvcfg 中将位 FIOM（fence of I/O 意味着内存）置为 1，则在权限低于 M 的模式下执行的 Fence 指令将被修改，因此对设备 I/O 的顺序访问要求也意味着对主内存的顺序访问的要求。表 https://www.five-embeddev.com/riscv-isa-manual/latest/machine.html#tab:menvcfg-FIOM 详细说明了当 FIOM=1 时，对于特权低于 M 的模式，FENCE 指令位 PI、PO、SI 和 SO 的修改解释。

类似地，当 FIOM=1 时，对于特权低于 M 的模式，如果访问按设备 I/O 排序的区域的原子指令设置了 aq and/or rl 位，则该指令的排序就好像它同时访问设备 I/O 和内存一样。如果不支持 S 模式，或者 satp.MODE 为只读零（始终为 Bare），则实现可能会使 FIOM 为只读零。

表 10.6 指令位及其含义

指令位	含义

PI	前置设备输入和存储器读取（隐含 PR）
PO	前置设备输出和内存写入（隐含 PW）
SI	后续设备输入和内存读取（隐含 SR）
SO	后续设备输出和内存写入（软件隐含）

在 menvcfg 中需要位 FIOM，因此 M 模式可以模拟第[hypervisor]章的 hypervisor 扩展，该扩展在 hypervisor CSR henvcfg 中具有等效的 FIOM 位。PBMTE 位控制 Svpbmt 扩展是否可用于 S 模式和 G 级地址转换（即，用于指向 satp 或 hgatp 的页表）。当 PBMTE=1 时，Svpbmt 可用于 S 模式和 G 阶段地址转换。当 PBMTE=0 时，表现为未执行 Svpbmt。如果未实现 Svpbmt，则 PBMTE 为只读零。此外，对于具有 hypervisor 扩展的实现，如果 menvcfg.PBMTE 为零，则 henvcfg.PBMTE 为只读零。STCE 字段的定义将由即将到来的 Sstc 扩展提供。在批准扩展之前，其在 menvcfg 内部的分配可能会发生变化。

CBZE 字段的定义将由即将到来的 Zicboz 扩展提供。在批准扩展之前，其在 menvcfg 内部的分配可能会发生变化。CBCFE 和 CBIE 字段的定义将由即将到来的 Zicbom 扩展提供。在批准这一扩展之前，它们在 menvcfg 内部的分配可能会发生变化。当 MXLEN=32 时，menvcfg 包含与 MXLEN=64 时 menvcfg 的位 31:0 相同的字段。此外，当 MXLEN=32 时，menvcfgh 是一个 32 位读/写寄存器，它包含与 MXLEN=64 时 menvcfg 的位 63:32 相同的字段。当 MXLEN=64 时，寄存器 menvcfgh 不存在。如果不支持 U 模式，则寄存器 menvcfg 和 menvcfgh 不存在。

10.1.19 Machine Security Configuration Register (mseccfg)

mseccfg 是一个可选的 MXLEN-bit 读/写寄存器，格式如下图所示，用于控制安全功能。仅当 MXLEN=32 时，mseccfgh 是一个 32 位读/写寄存器，其包含与 MXLEN=64 时的 mseccfg 位 63:32 相同的字段。

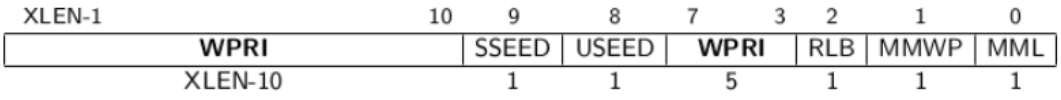


图 10.25 mseccfg 寄存器

SSEED 和 USEED 字段的定义将由即将到来的熵源（entropy-source ）扩展 Zkr 提供。在批准扩展之前，它们在 msecfg 内的分配可能会发生变化。

RLB、MMWP 和 MML 字段的定义将由即将进行的 PMP-enhancement 扩展 Smepmp 提供。在批准扩展之前，它们在 msecfg 内的分配可能会发生变化。

10.1.20 Machine Timer Register(mtime and mtimecmp)

mtime 寄存器是平台提供的一个实时计数器，用作内存映射机器模式读写寄存器。mtime 必须以恒定的频率递增，并且平台必须提供用于确定 mtime tick 的周期的机制。如果计数溢出，mtime 寄存器将 wrap around。

在 RV32 和 RV64 系统中 mtime 寄存器均有 64bit 的精度。平台提供了一个 64 位内存映射的机器模式定时器比较寄存器（mtimecmp）。每当 mtime 包含大于或等于 mtimecmp 的值时，机器计时器中断就会等候，并将这些值视为无符号整数。中断一直保持发布状态，直到 mtimecmp 大于 mtime（通常是写入 mtimecmp 的结果）。只有当中断被使能并且 MTIE 位被设置在 mie 寄存器中时，才会进行中断。

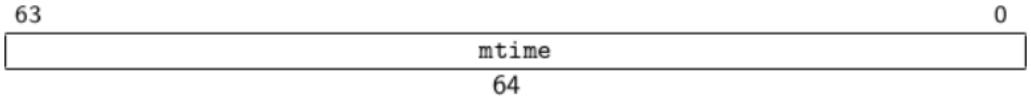


图 10.26 mtime 寄存器

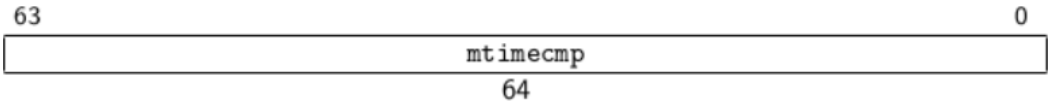


图 10.27 mtimecmp 寄存器

定时器设施被定义为使用 wall-clock 时间而不是循环计数器来支持以高度可变的时钟频率运行的现代处理器，以通过动态电压和频率缩放来节省能量。提供精确的实时时钟（RTC）相对昂贵（需要晶体或 MEMS 振荡器），并且即使在系统的其余部分无使能时也必须运行，因此系统中通常只有一个位于与处理器不同的频率/电压域中。因此，RTC 必须由系统中的所有 harts 共享，并且对 RTC 的访问可能会导致 voltage-level-shifter 和 clock-domain crossing 的惩罚。因此，将 mtime 公开为内存映射寄存器比公开为 CSR 更合理。

较低的特权级别没有自己的 timecmp 寄存器。相反，机器模式下软件可以通过将下一个定时器中断多路复用到 mtimecmp 寄存器中，在 hart 上实现任意数量的虚拟定时器。

简单的固定频率系统可以使用单个时钟进行周期计数和 wall-clock 计时。对 mtime 和 mtimecmp 的写入最终一定会反映在 MTIP 中，但不一定会立即反映出来。

如果中断处理程序增加 mtimecmp 然后立即返回，则可能会发生伪定时器中断，因为在此期间 MTIP 可能尚未拉低。所有软件都应该被编写为假设该事件是可能的，但大多数软件应该假设该事件极不可能发生。与轮询 MTIP 直到其失败相比，偶尔引起虚假计时器中断似乎总是更具性能。在 RV32 中，对 mtimecmp 的内存映射写入只修改寄存器的一个 32 位部分。以下代码序列设置一个 64 位的 mtimecmp 值，而不会由于比较器的中间值而错误地生成计时器中断：

```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)     # No smaller than old value.
sw a1, 4(t1)     # No smaller than new value.
sw a0, 0(t1)     # New value.
```

图 10.28 示意图

用于在 RV32 中设置 64 位时间比较器的示例代码，假设采用小端序存储系统，并且寄 存器位于 Strong Ordered I/O 区域中。将-1 存储到 mtimecmp 的低位可以防止 mtimecmp 暂时小于旧值和新值中的较小值。对于 RV64，额外支持对 mtime 和 mtimecmp 寄存器的自然对齐 64 位内存访问，并且是原子访问。

10.2 S 模式寄存器

10.2.1 Supervisor Status Register (sstatus)

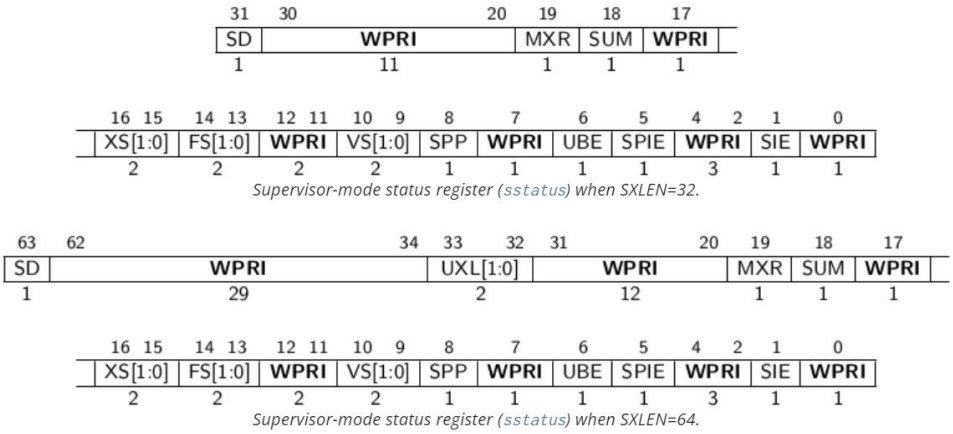


图 10.29 sstatus 寄存器

SPP:SPP 位表示 hart 在进入 supervisor 模式之前正在执行的特权级别。当接收到 trap 时，如果该 trap 来自 user 模式，则 SPP 设置为 0，否则设置为 1。当执行一条 SRET 指令(从 trap 处理程序返回时，如果 SPP 位为 0，则特权级别被设置为用户模式，如果 SPP 位为 1，则特权级别被设置为 supervisor 模式;SPP 设置为 0。

SIE:SIE 位在 supervisor 模式下启用或禁用所有中断。当 SIE 为 0 时，在 supervisor 模式下禁用中断。当 hart 在 user 模式下运行时，SIE 中的值将被忽略，并且启用管理员级中断。supervisor 可以使用 sie CSR 禁用单个中断源。

SPIE:SPIE 位表示在进入 supervisor 模式之前是否使能了 supervisor 中断。当 trap 进入 supervisor 模式时，SPIE 设置为 SIE, SIE 设置为 0。当执行一条 SRET 指令时，SIE 被设置为 SPIE，然后 SPIE 被设置为 1。

UXL: UXL 字段控制 user 模式的 XLEN 值。当 SXLEN=32 时，UXL 字段不存在，UXLEN=32。当 SXLEN=64 时，它是一个编码 UXLEN 当前值的 WARL 字段。

MXR: MXR 位控制加载访问虚拟内存的权限。当 MXR=0 时，只能访问可读的内存区域。当 MXR=1 时，可以访问可读和可操作的内存区域。如果虚拟内存无效，MXR 没有用处。

SUM:SUM 修改 supervisor 模式加载和存储访问虚拟内存的权限, SUM 机制防止管理软件无意中访问用户内存。当 SUM=0 时，supervisor 模式内存访问 user 模式可访问的页面将出错。当 SUM=1 时，允许这些访问。当基于页面的虚拟内存无效时，SUM 不起作用，当以 user 模式执行时也不起作用。无论 SUM 的状态如何，s 模式永远不能执行来自用户页面的指令

UBE:UBE 位用于控制 user 模式下是否启动错误指令异常，当设置为 1 时，用户模式下的错误指令会引发异常，设置为 0 时，用户模式下的错误指令会被忽略，不会引发异常。

10.2.2 Supervisor Trap Vector Base Address Register(stvec)

stvec 寄存器是一个 sxlen 位的读/写寄存器，它保存 trap 向量配置，由一个向量基址(base)和一个向量模式(mode)组成。

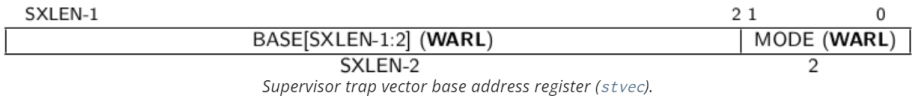


图 10.30 stvec 寄存器

stvec 中的 BASE 字段是一个 WARL 字段，它可以保存任何有效的虚拟地址或物理地址。

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	*Reserved*

图 10.31 MODE 字段编码

MODE 字段的编码如表所示。当 MODE=Direct 时，所有进入 supervisor 模式的 trap 都会将 pc 设置为 BASE 字段中的地址。当 MODE= vector 时，所有进入 supervisor 模式的同步异常导致 pc 被设置为 BASE 字段中的地址，而中断导致 pc 被设置为 BASE 字段中的地址加上中断原因编号的四倍。

10.2.3 Supervisor Interrupt Registers (sip and sie)

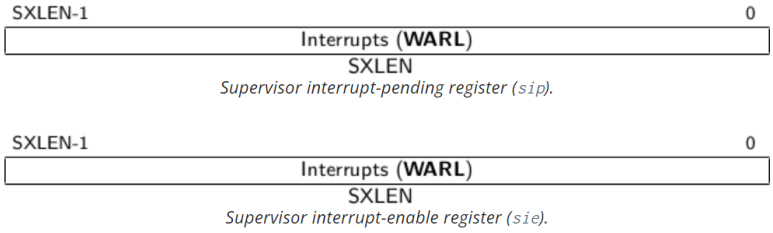


图 10.32 sip/sie 寄存器

sip 寄存器是一个包含挂起中断信息的 sxlen 位读/写寄存器，而 sie 是对应的包含中断使能位的 sxlen 位读/写寄存器。15:0 bit 位分配给中断原因。16bit 以及以上的位用于自定义使用。

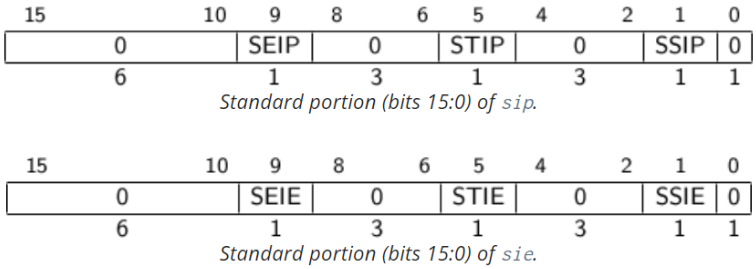


图 10.33 sip/sie 寄存器[15:0] bit

sip.SEIP 和 sie.SEIE 用于 supervisor 级外部中断的中断挂起和中断使能位。
sip.STIP 和 sie.STIE 是 supervisor 级定时器中断的中断挂起和中断启用位。
sip.SSIP 和 sie.SSIE 是 supervisor 级软件中断的中断挂起和中断启用位。

10.2.4 Counter-Enable Register (scounteren)

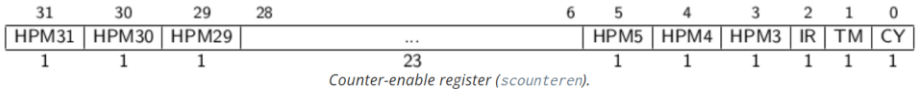


图 10.34 scounteren 寄存器

counter-enable 寄存器 scounteren 是一个 32 位寄存器，它控制硬件性能监控计数器到 user 模式的可用性。当计数器寄存器中的 CY、TM、IR 或 HPMn 位清除时，在 user 模式下执行时试图读取周期、时间等寄存器将导致非法指令异常。当这些位之一被设置时，就允许访问相应的寄存器。

10.2.5 Supervisor Scratch Register (sscratch)

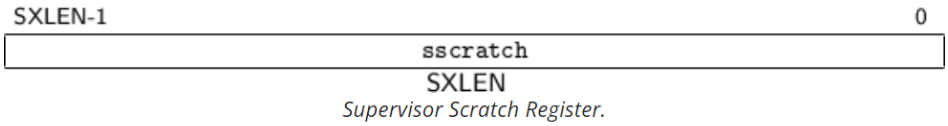


图 10.35 sscratch 寄存器

Supervisor Scratch Register（SSR）是一个用于 Supervisor 级别软件的临时存储寄存器。它提供了一个供 Supervisor 模式的软件使用的、不受保存和恢复规则限制的工作区域。Supervisor Scratch Register 可以用于保存 Supervisor 级别软件在执行期间需要暂时保存的数据、状态或临时变量。由于它不受保存和恢复规则的限制，Supervisor 级别的软件可以自由地使用这个寄存器，而不必担心影响到其他寄存器的值。

10.2.6 Supervisor Exception Program Counter (sepc)

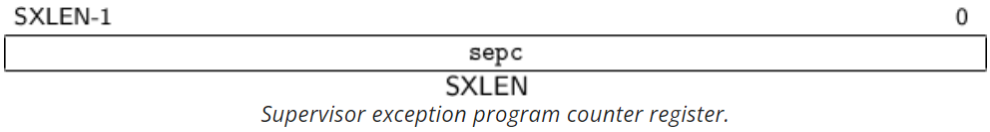


图 10.36 spec 寄存器

Supervisor 模式下的 Supervisor Exception Program Counter（SEPC）用于存储发生异常时的下一条指令的地址。当在 Supervisor 模式下发生异常（如中断或故障）时，处理器会将 SEPC 设置为发生异常的指令的地址，以便在异常处理程序执行完毕后能够正确返回到异常指令的下一条指令。

10.2.7 Supervisor Cause Register (scause)

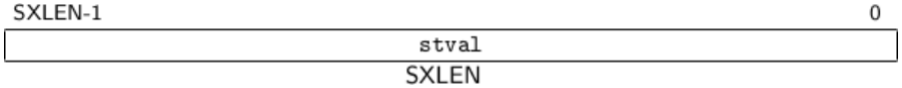
Supervisor 模式下的 Supervisor Cause Register（SCAUSE）用于存储异常或中断的原因。

Supervisor Cause register scause.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

图 10.37 scause 寄存器

10.2.8 Supervisor Trap Value (stval) Register



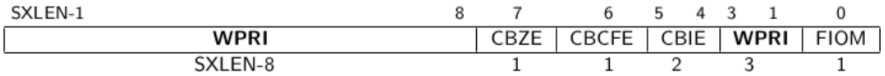
Supervisor Trap Value register.

图 10.38 stval 寄存器

Supervisor Trap Value Register（STVAL）用于存储导致异常或中断的指令或访问的引发的地址或值。例如在在指令获取、加载或存储中发生断点、地址不对齐、访问错误或页面错误异常等。

10.2.9 Supervisor Environment Configuration Register (senvcfg)

senvcfg CSR 是一个读/写寄存器，它控制 user 模式执行环境的配置。



Supervisor environment configuration register (senvcfg).

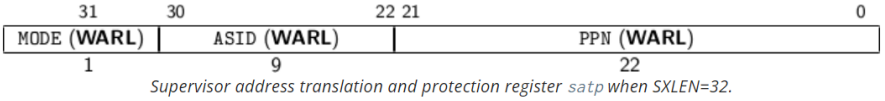
图 10.39 senvcfg 寄存器

FIOM:设置为 1，在 user 模式下执行的 Fence(用于控制内存访问的顺序和可见性)指令会被修改,可以控制对设备 I/O 的访问顺序要求也包括对主存访问的顺序要求。user 模式下 Fence 指令为 PI,PO,SI 和 SO 和修改解释如下。

Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

图 10.40 Fence 指令

10.2.10 Supervisor Address Translation and Protection (satp) Register



Supervisor address translation and protection register satp when SXLEN=32.

图 10.41 satp 寄存器

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section **??**).

图 10.42 MODE 位描述

Satp 寄存器用于配置和管理 Supervisor 模式下的地址转换和保护机制。satp 寄存器的作用是指定用于虚拟地址到物理地址转换的页表根据址（Page Table Root）以及当前的地址转换模式。

satp 寄存器的位字段解释如下：

MODE（位 31）：用于指示地址转换模式，0 表示禁用地址转换（Bare 模式），1 表示启用 Sv32 模式。

ASID（位 30:22）：用于指定地址空间标识符，用于与 TLB（转换查找缓冲）相关联，以区分不同的地址空间。

PPN（位 21:0）：用于指定页表根据址（Page Table Root）的物理页号，用于虚拟地址到物理地址的转换。

satp 寄存器在 Supervisor 模式下的使用非常重要，它确定了地址转换和内存保护的机制。通过配置合适的页表根据址和地址空间标识符，satp 寄存器允许 Supervisor 模式下的代码和数据访问正确的物理内存位置，并提供内存保护功能，以确保不同的地址空间之间的隔离和安全性。

11 Biriscv 实现

11.1 M-mode

11.1.1 csr_mstatus

mstatus 是 32 位的读写寄存器，用于跟踪和控制当前 hart 的操作状态。

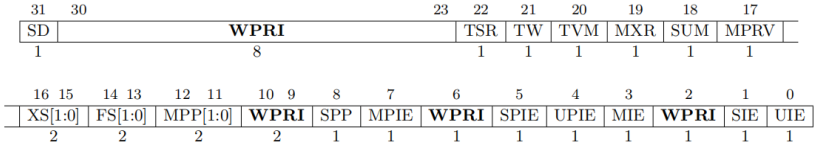


Figure 3.6: Machine-mode status register (`mstatus`) for RV32.

图 11.1 mstatus 寄存器

7. SR_xIE(xIE)

用于控制全局中断的启用和关闭。

8. SR_xPIE,SR_xPP(xPIE,xPP)

当触发 x 模式的中断异常时，xIE 的值被设置为 0，xPIE 用于保存之前 xIE 的使能状态，xPP 用于保存之前所处的特权模式。当执行 xRET 时，xIE 被设置为 xPIE 的值，特权模式改变为 xPP 中保存的特权模式，并且 xPP 被设置为 0。

9. SR_SUM（SUM）

用于修改 S 模式加载和存储访问虚拟内存的权限，当 SUM=0 时，S 模式存储器对 U 模式可访问的页面进行访问时将发生故障。当 SUM=1 时允许这些访问，当基于页面的虚拟内存无效时，SUM 无效。

10. SR_MPRV（MPRV）

MPRV 位修改在所有特权模式下加载和存储执行的特权级别。当 MPRV=0 时，加载和存储行为正常，使用当前特权模式的转换和保护机制。当 MPRV＝1 时，使用 MPP 中的特权模式的转换和保护机制。

11. SR_MXR（MXR）

用于修改加载访问虚拟内存的权限，当 MXR=0 时，我们可以 load 标记为可读（R=1）的页面。当 MXR＝1 时，我们可以 load 标记为可读或者可执行的（R=1 或 X=1）的页面。

11.1.2 csr_mcause（mcause）

当触发一个 trap 并进入 m 模式时，mcause 中会保存触发的中断异常的编码。

11.1.3 csr_mtval(mtval)

当触发一个 trap 并进入 m 模式时，mtval 要么设置为零，要么写入异常特定信息，以帮助软件处理陷阱。由硬件平台将指定哪些异常必须设置 mtval，哪些异常可以无条件地将其设置为零。下图是 mtval 的设置规则。在 biriscv 中，对于非法指令，依然将内存访问的地址存入 mtval 中。

```
case (exception_i)
`EXCEPTION_MISALIGNED_FETCH,
`EXCEPTION_FAULT_FETCH,
`EXCEPTION_PAGE_FAULT_INST:     csr_mtval_r = exception_pc_i;
`EXCEPTION_ILLEGAL_INSTRUCTION,
`EXCEPTION_MISALIGNED_LOAD,
`EXCEPTION_FAULT_LOAD,
`EXCEPTION_MISALIGNED_STORE,
`EXCEPTION_FAULT_STORE,
`EXCEPTION_PAGE_FAULT_LOAD,
`EXCEPTION_PAGE_FAULT_STORE:    csr_mtval_r = exception_addr_i;
default:                        csr_mtval_r = 32'b0;
endcase
```

图 11.2 代码示例

11.1.4 csr_mtvec(mtvec)

用于保存陷阱矢量的配置。

图 11.3 mtvec 寄存器

如果 MODE 域的值为 0，则所有陷入机器模式的 trap 都会将 pc 设置为 BASE 字段中的地址。如果 MODE 域的值为 1，那么所有陷入机器模式的 trap 都会将 pc 的值设置为 BASE+4×cause
csr_mtvec 中保存着 trap_vector 的地址

11.1.5 csr_mip（mip）,csr_mie(mie)

mip 中包含着挂起中断的信息，mie 包含着中断使能的相应信息。在 biriscv 中，mie 仅有 MSIE,MTIE,MEIE 三位可读可写，mip 仅有 MSIP,MTIP.MEIP 三位可读可写。

11.1.6 csr_mscratch(mscratch)

mscratch 寄存器是专用于机器模式的 MXLEN 位读/写寄存器。通常，它用于保存指向机器模式 hart 本地上下文空间的指针，这里 mscratch 中保存着机器模式的 sp 指针，并在进入 M 模式陷阱处理程序时与用户寄存器交换。

11.1.7 csr_mcycle，csr_mcycle_h，csr_mtimecmp(mcycle,mtimecmp)

mcycle CSR 用于统计处理器内核执行的时钟周期数,mcycle 有 64 位,在 biriscv 中用 mcycle 的低 32 位与 csr_mtimecmp 进行比较，相等时发起定时中断。

```
if (SUPPORT_MTIMECMP && csr_mcycle_q == csr_mtimecmp_q)
begin
    if (csr_mideleg_q[`SR_IP_MTIP_R])
        csr_mip_next_r[`SR_IP_STIP_R] = csr_mtime_ie_q;
    else
        csr_mip_next_r[`SR_IP_MTIP_R] = csr_mtime_ie_q;
    csr_mtime_ie_r  = 1'b0;
end

csr_mip_r = csr_mip_r | csr_mip_next_r;
```

图 11.4 代码示意图

11.1.8 csr_medeleg，csr_mideleg(medeleg,mideleg)

```
  uintptr_t interrupts = MIP_SSIP | MIP_STIP | MIP_SEIP;
  uintptr_t exceptions =
    (1U << CAUSE_MISALIGNED_FETCH) |
    (1U << CAUSE_FETCH_PAGE_FAULT) |
    (1U << CAUSE_BREAKPOINT) |
    (1U << CAUSE_LOAD_PAGE_FAULT) |
    (1U << CAUSE_STORE_PAGE_FAULT) |
    (1U << CAUSE_USER_ECALL);

  write_csr(mideleg, interrupts);
  write_csr(medeleg, exceptions);
  assert(read_csr(mideleg) == interrupts);
  assert(read_csr(medeleg) == exceptions);
}
```

图 11.5 代码示意图

如上图所示，这里将 S 模式中断和大多数异常直接委托给 S 模式

11.1.9 csr_mepc(mepc)

在进入机器模式中断异常后保存进入中断异常之前 pc 指针的值

11.1.10 csr_misa(misa)

由硬件指定，表明该核支持 rv32I 基本指令集，并支持 M 扩展（这里 SUPPORT_MULDIV 默认值为 1）

```
wire [31:0] misa_w = SUPPORT_MULDIV ? (`MISA_RV32 | `MISA_RVI | `MISA_RVM): (`MISA_RV32 | `MISA_RVI)
```

图 11.6 代码示意图

11.1.11 csr_mhartid(mhartid)

由于 biriscv 中只有一个硬件线程，mhartid 的值恒为零。Mode 在 biriscv 中没有定义 sstatus,在进入 s 模式的中断异常前，在硬件上，将 mstatus 中 SPP 域的值更新为进入中断异常之前的特权模式，mstatus 的 SPIE 更新为 SIE 的值，SIE 清零。

11.1.12 csr_sepc（sepc）

在进入 S 特权模式中断异常前保存进入中断异常之前 pc 指针的值

11.1.13 csr_stvec(stvec)

在 csr_stvec 中保存 handle_exception 的地址,在进入 S 模式中断异常时，从 handle_exception 开始执行。

11.1.14 csr_scause(scause)

在进入 S 模式中断异常前，scause 中会保存触发的中断异常的编码

11.1.15 csr_stval(stval)

当触发一个 trap 并进入 s 模式时，stval 要么设置为零，要么写入异常特定信息，以帮助软件处理陷阱。由硬件平台将指定哪些异常必须设置 stval，哪些异常可以无条件地将其设置为零。下图是 biriscv 中 stval 的设置规则。

```
case (exception_i)
`EXCEPTION_MISALIGNED_FETCH,
`EXCEPTION_FAULT_FETCH,
`EXCEPTION_PAGE_FAULT_INST:     csr_stval_r = exception_pc_i;
`EXCEPTION_ILLEGAL_INSTRUCTION,
`EXCEPTION_MISALIGNED_LOAD,
`EXCEPTION_FAULT_LOAD,
`EXCEPTION_MISALIGNED_STORE,
`EXCEPTION_FAULT_STORE,
`EXCEPTION_PAGE_FAULT_LOAD,
`EXCEPTION_PAGE_FAULT_STORE:    csr_stval_r = exception_addr_i;
default:                        csr_stval_r = 32'b0;
endcase
```

图 11.7 代码示意图

11.1.16 csr_satp(satp)

当 MODE 位为 0 时，虚拟地址不会进行转化，此刻的虚拟地址就是物理地址。MODE 为不同值时，会根据 MODE 位选择不同结构的页表。ASID（Address Space Identifier， 地址空间标识符）域是可选的，它可以用来降低上下文切换的开销。PPN 存放的是最高级页表的起始页号（页目录），根据相应的 page 大小进行可以得到最高级页表（页目录）的物理地址。这样 cpu 就可以告诉虚拟内存地址从哪里翻译成物理内存地址。

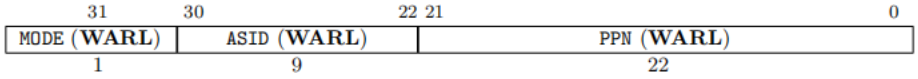


图 11.8 satp 寄存器

11.1.17 csr_sscratch

sscratch 用于在 hart 执行用户代码时保持指向 hart 本地 S-mode 上下文的指针，在进入中断异常后，sscratch 与用户态的 sp 指针的值进行交换，此时 sp 指向 S-mode 堆栈。

11.1.18 csr_sie,csr_sip

在 biriscv 中，sie 仅有 SSIE,STIE,SEIE 三位可读可写，sip 仅有 SSIP,STIP,SEIP 三位可读可写

12 局部中断控制器（CLINT）

Birsicv 的中断控制器模块 CLINT 主要负责处理软件中断。 CLINT 中断控制器占据 64KB 地址空间。模块由两个模块组成：CLINT 单元和 CLINT 寄存器单元。其中 CLINT 单元为顶层模块，CLINT 寄存器单元负责实现 MSIP 寄存器，和 SSIP 寄存器。

12.1 IPI 中断

CLINT 模块可以用于生成 IPI 中断。CLINT 中实现了两个 32 位 msip 和两个 32 位 ssip 寄存器。软件中断通过配置地址映射的软件中断配置寄存器进行中断信号的控制。具体映射地址可查询表 12.1 CLINT 寄存器的存储器映射地址。

12.1.1 寄存器

M（Machine）模式和 S（Supervisor）模式的软件中断分别由 msip 和 ssip 寄存器进行控制。

该寄存器只有最低位为有效位，有效位将作为软件中断信号发送给 CPU 核。当软件将 msip 寄存器的有效位置 1 后，CSR 寄存器 mip 中的 msip 域会置 1 表示当前处于中断等待状态。软件也可以通过写 0 置寄存器来清除软件中断。对于软件中断，CLINT 模块将读取 msip 寄存器的最低位，当最低位为 1 的时候，CLINT 模块将触发 IPI 中断信号。

1. msip

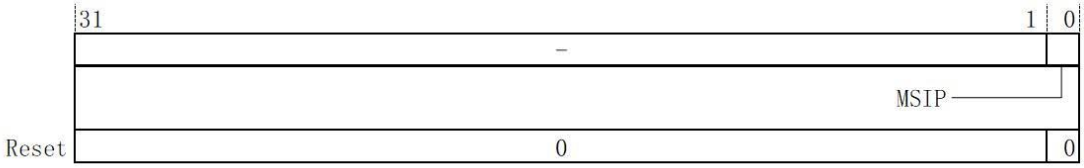


图 12.1 msip 寄存器

表 12.2 msip 寄存器描述

比特位	字段名称	属性	解释
[0:0]	msip	RW	M 模式软件中断是通过写入存储器映射的控制寄存器 msip 产生的。msip 寄存器是一个 32 位宽的 WARL 寄存器，其中高 31 位绑定到 0。最低有效位可用于驱动 RISC-V 核中 CSR 寄存器 mip 的 MSIP 位。msip 寄存器中的其他位硬连线到 0。重置时，msip 寄存器被清除为零。

[31:1]	Reserved	RW	读取时返回 0，写入时无效。

12. ssip

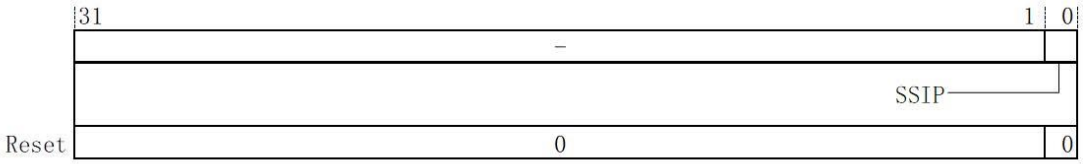


图 12.2 ssip 寄存器

表 12.3 ssip 寄存器描述

比特位	字段名称	属性	解释
[0:0]	ssip	RW	S 模式软件中断是通过写入存储器映射的控制寄存器 ssip 产生的。ssip 寄存器是一个 32 位宽的 WARL 寄存器，其中高 31 位绑定到 0。最低有效位可用于驱动 RISC-V 核中 CSR 寄存器 sip 的 SSIP 位。ssip 寄存器中的其他位硬连线到 0。重置时，ssip 寄存器被清除为零。
[31:1]	Reserved	RW	读取时返回 0，写入时无效。

12.1.2 中断处理过程

1. RV Linux Kernel 发起中断请求

内核程序调用 smp 相关函数，这里以 Linux 进程调度为例，首先 Linux 内核调度程序调用 smp_send_reschedule()函数，该函数将调用 send_ipi_single()函数，并将 CLINT 中 msip 寄存器最低位置 1，触发软件中断。

13. CLINT 接收中断

软件将中断控制器中 msip 寄存器相应 MSIP 位置 1。CLINT 通过读取 msip 寄存器接受中断请求，并通过 softirq 引脚发送中断请求给 CPU Core

14. CSR 更改相应寄存器

mip.MSIP 位被置 1 后，同时 mie.MSIE 为 1，则开始处理中断。此时，处理器执行下列操作：
•处理器执行完当前指令，保存下一条指令的 PC 到 mepc 中。
•设置 mcause 的中断标记为 1，将中断编号写入 mcause，并更新 mtval 为 0。
•将 mstatus 的中断使能位 MIE 保存到 MPIE 中，将 MIE 清零，禁止响应中断。

•将发生中断之前的权限模式保存到 mstatus 的 MPP 中，切换到机器模式。
•（mtvec.Mode=0，直通中断）：PC 从 mtvec 处执行。通常，取回的指令是一条跳转指令，跳转至顶层处理函数。该函数通过分析 mcause 获取中断编号，并调用该编号对应的处理函数。
•（mtvec.Mode=1，矢量中断）：PC 从 mtvec.Base + 4 * 中断编号处执行。通常，取回的指令是一条跳转指令，跳转至相应中断的处理函数。

15. 中断处理程序执行

CPU 跳转至软件中断处理函数入口，软件判断中断为 IPI 中断，跳转至 IPI 中断处理函数入口，根据 IPI 中断 ID 跳转至相应的 IPI 处理函数并执行。

16. 中断处理程序执行完毕

中断返回执行 mret 指令可以实现中断返回。此时，处理器执行下列操作：
• 将 mepc 恢复到 PC。（mepc 保存的是下一条指令的 PC，所以无需调整）
• 将 mstatus.MPIE 恢复到 mstatus.MIE。
• 从 mstatus.MPP 恢复发生中断之前的权限模式。

13 中断

13.1 时钟中断

在 Biriscv 当中，时钟中断由 CPU Core 内部的 CSR 单元触发。Biriscv 的每个 Hart 都定义了一个 mtime 寄存器和 mtimecmp 寄存器。由操作系统负责寄存器值的加减，并触发相应的时钟中断，在触发时钟中断后，操作系统将根据 jiffie 值对 mtimecmp 寄存器的值更新，准备下一次时钟中断。

13.1.1 寄存器

1. mtime

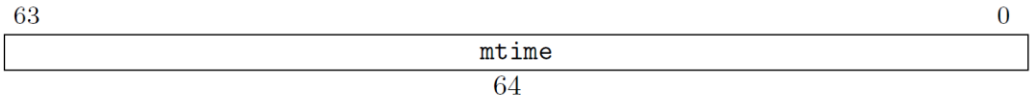


图 13.1 mtime 寄存器

表 13.1 mtime 寄存器描述

比特位	字段名称	属性	解释
[63:0]	mtime	RW	多核系统中仅存在一个 64 位系统计时器 mtime。mtime 位于 CSR 中，用于跟踪从任意时间点开始计数的周期数。它是一个自由运行的计数器，每次循环数递增一次。

2. mtimecmp

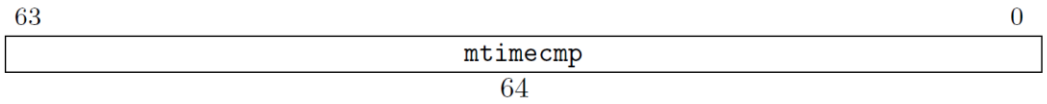


图 13.2 mtimecmp 寄存器

表 13.2 mtimecmp 寄存器描述

比特位	字段名称	属性	解释
[63:0]	mtimecmp	RW	位于 CSR 中，保存一个 64 位的值。每当 mtime 大于或等于 mtimecmp 寄存器中的值时，计时器中断就会挂起。定时器中断用于驱动 CSR 寄存器 mip 的 MTIP 位。

13.1.2 中断处理过程

1. CLIINT 发起时钟中断请求

CLINT 比较 mtimecmp[63:0]≤mtime[63:0]时，产生中断。

2. CSR 更改相应寄存器

mip.MSIP 位被置 1 后，同时 mie.MSIE 为 1，则开始处理中断。此时，处理器执行下列操作：

- 处理器执行完当前指令，保存下一条指令的 PC 到 mepc 中。
- 设置 mcause 的中断标记为 1，将中断编号写入 mcause，并更新 mtval 为 0。
- 将 mstatus 的中断使能位 MIE 保存到 MPIE 中，将 MIE 清零，禁止响应中断。
- 将发生中断之前的权限模式保存到 mstatus 的 MPP 中，切换到机器模式。
- mtvec.Mode=0，直通中断）：PC 从 mtvec 处执行。通常，取回的指令是一条跳转指令，跳转至顶层处理函数。该函数通过分析 mcause 获取中断编号，并调用该编号对应的处理函数。
- （mtvec.Mode=1，矢量中断）：PC 从 mtvec.Base + 4 * 中断编号处执行。通常，取回的指令是一条跳转指令，跳转至相应中断的处理函数。

3. 中断处理程序执行

RV Linux Kernel 接收到来自 CPU Core 的时钟中断请求，进入 IRQ 处理函数并执行时钟中断相关处理函数，读取 mtimecmp 中的值，并根据系统 jiffie 值进行更新，并写回 mtimecmp 寄存器。（待完善）

4. 中断处理程序执行完毕

中断返回执行 mret 指令可以实现中断返回。此时，处理器执行下列操作：

- 将 mepc 恢复到 PC。（mepc 保存的是下一条指令的 PC，所以无需调整）
- 将 mstatus.MPIE 恢复到 mstatus.MIE。
- 从 mstatus.MPP 恢复发生中断之前的权限模式。

机器模式下拥有修改访问所有计时器中断相关寄存器的权限；超级用户模式下仅具有访问修改超级用户模式时钟计时器比较值寄存器（stimecmp）的权限；普通用户模式没有权限。

13.2 UART 中断

在 Biriscv 当中，只有一个外部中断源 UART，UART 中断通过外部中断引脚直接接入到 CPU Core 内部。具体 UART 实现方法不在这里赘述，可查询 16550A 资料，以下是 Biriscv 当中的 UART 中断处理流程。

13.2.1 UART 核发起中断请求

UART 通过相应 IRQ 引脚向 CPU 核发送中断请求。

13.2.2 CSR 更改相应寄存器

mip.MSIP 位被置 1 后，同时 mie.MSIE 为 1，则开始处理中断。此时，处理器执行下列操作：

- 处理器执行完当前指令，保存下一条指令的 PC 到 mepc 中。
- 设置 mcause 的中断标记为 1，将中断编号写入 mcause，并更新 mtval 为 0。
- 将 mstatus 的中断使能位 MIE 保存到 MPIE 中，将 MIE 清零，禁止响应中断。
- 将发生中断之前的权限模式保存到 mstatus 的 MPP 中，切换到机器模式。
- （mtvec.Mode=0，直通中断）：PC 从 mtvec 处执行。通常，取回的指令是一条跳转指令，跳转至顶层处理函数。该函数通过分析 mcause 获取中断编号，并调用该编号对应的处理函数。
- mtvec.Mode=1，矢量中断）：PC 从 mtvec.Base + 4 * 中断编号处执行。通常，取回的指令是一条跳转指令，跳转至相应中断的处理函数。

1. 中断处理程序执行

RV Linux Kernel 接收到来自 UART 中断请求，进入 IRQ 处理函数并执行 UART 中断相关处理函数。（待完善）

2. 中断处理程序执行完毕

中断返回执行 mret 指令可以实现中断返回。此时，处理器执行下列操作：

- 将 mepc 恢复到 PC。（mepc 保存的是下一条指令的 PC，所以无需调整）
- 将 mstatus.MPIE 恢复到 mstatus.MIE。

- 从 mstatus.MPP 恢复发生中断之前的权限模式。

表 13.3 CLINT 寄存器的存储器映射地址

地址	宽度	属性	名称	初始值
0x0200_0000	4B	RW	msip0	0x00000000
0x0200_0004	4B	RW	msip1	0x00000000
Reserved	-	-	Reserved	
0x0200_4000	4B	RW	mtimecmp0_low	0xffffffff
0x0200_4004	4B	RW	mtimecmp0_high	0xffffffff
0x0200_4008	4B	RW	mtimecmp1_low	0xffffffff
0x0200_400C	4B	RW	mtimecmp1_high	0xffffffff
Reserved	-	-	Reserved	
0x0200_BFF8	4B	RW	mtime_low	0x00000000
0x0200_BFFC	4B	RW	Mtime_high	0x00000000
0x0200_C000	4B	RW	ssip0	0x00000000
0x0200_C004	4B	RW	ssip1	0x00000000
Reserved	-	-	Reserved	
0x0200_D000	4B	RW	stimecmp0_low	0xffffffff
0x0200_D004	4B	RW	stimecmp0_high	0xffffffff
0x0200_D008	4B	RW	stimecmp1_low	0xffffffff
0x0200_D00C	4B	RW	stimecmp1_high	0xffffffff
Reserved	-	-	Reserved	

表 13.4 UART 寄存器的存储器映射地址

地址	宽度	属性	名称	初始值
0x9200_0000	1B	RW	Divisor Latch Byte 1	0x00
0x9200_0001	1B	RW	Divisor Latch Byte 2	0x00

0x9200_0000	1B	R	Receiver Buffer	0x00
0x9200_0000	1B	W	Transmitter Holding Register	0x00
0x9200_0001	1B	RW	Interrupt Enable	0x00
0x9200_0002	1B	R	Interrupt Identification	0xC1
0x9200_0002	1B	W	FIFO Control	0xC0
0x9200_0003	1B	RW	Line Control Register	0x03
0x9200_0004	1B	W	Modem Control Register	0x00
0x9200_0005	1B	R	Line Status Register	0x00
0x9200_0006	1B	R	Modem Status Register	0x00

13.3 端口列表

表 13.5 端口列表

信号名	I/O	初始值	时钟域	功能描述
timer_irq_o[1:0]	O			时钟中断信号
ipi_o[1:0]	O			核间中断信号
Exc_o	O			外部中断信号

14 Wishbone 总线

14.1 时序

附录 1 信号列表

signal	Dri	Description
分支预测单元信号		
branch_info_request_i	in	分支跳转请求信号
branch_info_pc [31:0]	in	分支跳转 PC 地址
next_pc_f_o [31:0]	out	取指地址
取指单元		
next_pc_f_i [31:0]	in	取指地址
branch_request_i	in	分支跳转请求信号
branch_pc_i [31:0]	in	分支跳转 PC 地址
icache_inst_i [63:0]	in	I-cache 输入指令
译码单元		
fetch_pc_instr_i [31:0]	in	PC 地址
fetch_in_instr_i [63:0]	in	指令
branch_request_i	in	分支跳转请求
fetch_out0_instr_o [31:0]	out	发射 1 指令
fetch_out0_pc_o [31:0]	out	发射 1PC 地址
fetch_out1_instr_o [31:0]	out	发射 2 指令
fetch_out1_pc_o [31:0]	out	发射 2PC 地址

指令缓存		
axi_rdata_i[31:0]	in	AXI 总线数据
req_pc_i[31:0]	in	指令 PC 地址
fetch_out_pc_o[31:0]	out	指令 PC 地址
发射单元		
clk_i	in	时钟
rst_i	in	复位信号
fetch0_valid_i	in	译码单元 FIFO valid 信号，FIFO 工作正常
fetch0_instr_i [31:0]	in	发射 0 指令
fetch0_pc_i [31:0]	in	发射 0PC 地址
fetch0_fault_fetch_i	in	AXI Error
fetch0_fault_page_i	in	特权等级错误
fetch0_instr_exec_i	in	Exec 单元指令
fetch0_instr_lsu_i	in	LSU 单元指令
fetch0_instr_branch_i	in	Branch 指令
fetch0_instr_mul_i	in	Mul 指令
fetch0_instr_div_i	in	Div 指令
fetch0_instr_csr_i	in	CSR 指令
fetch0_instr_rd_valid_i	in	指令有目标寄存器，且目标寄存器有效
fetch0_instr_invalid_i	in	非法指令

fetch1_valid_i	in	译码单元 FIFO valid 信号，FIFO 工作正常
fetch1_instr_i [31:0]	in	发射 1 指令
fetch1_pc_i [31:0]	in	发射 1PC 地址
fetch1_fault_fetch_i	in	AXI Error
fetch1_fault_page_i	in	特权等级错误
fetch1_instr_exec_i	in	Exec 单元指令
fetch1_instr_lsu_i	in	LSU 单元指令
fetch1_instr_branch_i	in	Branch 指令
fetch1_instr_mul_i	in	Mul 指令
fetch1_instr_div_i	in	Div 指令
fetch1_instr_csr_i	in	CSR 指令
fetch1_instr_rd_valid_i	in	指令有目标寄存器，且目标寄存器有效
fetch1_instr_invalid_i	in	非法指令
branch_exec0_request_i	in	Branch 指令（跳转或不跳转）
branch_exec0_is_taken_i	in	Branch 指令跳转
branch_exec0_is_not_taken_i	in	Branch 指令不跳转
branch_exec0_is_call_i	in	跳转指令
alu_p_o[31:0]	in	ALU 单元写回结果
csr_result_e1_value_o[31:0]	in	CSR 单元写回结果
writeback_value_o[31:0]	in	MUL 单元写回结果

writeback_value_o[31:0]	in	LSU 单元写回结果
branch_pc_o[31:0]	out	分支跳转地址
branch_request_o	out	分支跳转请求
branch_info_pc_o[31:0]	out	BPU 更新 PC 跳转地址
branch_info_request_o	out	BPU 更新分支跳转请求
opcode0_ra_operand_o[31:0]	out	ALU0 源操作数 1
opcode0_rb_operand_o[31:0]	out	ALU0 源操作数 2
opcode1_ra_operand_o[31:0]	out	ALU1 源操作数 1
opcode1_rb_operand_o[31:0]	out	ALU1 源操作数 2
csr_opcode_ra_operand_o[31:0]	out	CSR 源操作数 1
csr_opcode_rb_operand_o[31:0]	out	CSR 源操作数 2
mul_opcode_ra_operand_o[31:0]	out	MUL 源操作数 1
mul_opcode_rb_operand_o[31:0]	out	MUL 源操作数 2
lsu_opcode_ra_operand_o[31:0]	out	LSU 源操作数 1
lsu_opcode_rb_operand_o[31:0]	out	LSU 源操作数 2
执行单元		
alu_a_i[31:0]	in	ALU0 源操作数 1
alu_b_i[31:0]	in	ALU0 源操作数 2
opcode_ra_operand_i[31:0]	in	CSR/MUL/LSU 源操作数 1
opcode_rb_operand_i[31:0]	in	CSR/MUL/LSU 源操作数 2

mem_data_rd_i[31:0]	in	访存结果读取
alu_p_o[31:0]	out	ALU 写回结果
csr_result_e1_value_o[31:0]	out	CSR 写回结果
writeback_value_o[31:0]	out	MUL 写回结果
writeback_value_o[31:0]	out	LSU 写回结果
mem_addr_o[31:0]	out	访存地址
mem_data_wr_o[31:0]	out	访存写入数据
MMU 信号		
fetch_out_inst_i[63:0]	in	输入 I-MMU 指令
fetch_in_pc_i[31:0]	in	输入 I-MMU 地址
req_inst_o[63:0]	out	I-MMU 输出指令
fetch_out_pc_o[31:0]	out	I-MMU 输出地址
lsu_in_addr_i[31:0]	in	输入 D-MMU 地址
lsu_out_data_wr_w[31:0]	in	输入 D-MMU 的读数据
lsu_out_data_rd_i[31:0]	in	输入 D-MMU 的写数据
lsu_in_data_rd_o[31:0]	out	D-MMU 输出数据
lsu_out_addr_o[31:0]	out	D-MMU 输出地址
lsu_out_data_wr_o[31:0]	out	D-MMU 输出写数据
指令缓存信号		
axi_rdata_i[31:0]	in	输入总线指令
req_pc_i[31:0]	in	输入地址
fetch_out_pc_o[31:0]	out	I-cache 输出指令

数据缓存信号		
mem_addr_i[31:0]	in	输入地址
mem_data_wr_i[31:0]	in	输入指令
mem_data_rd_o[31:0]	out	输出指令

附录 2 参考文献

Github 下载地址: https://github.com/ultraembedded/biriscv
BootLoader 下载地址: https://github.com/ultraembedded/riscv-linux-boot
Kernel image 下载地址: https://github.com/ultraembedded/riscv-linux-prebuilt

附录 3 寄存器列表/属性

Register Name	Attribute
User State	
x0/zero	Zero
x1/ra	Return Address
x2/sp	Stack Pointer
x3/gp	Global Pointer
x4/tp	Thread Pointer
x5/t0	Temporary Register
x6/t1	Temporary Register
x7/t2	Temporary Register
x8/s0	Saved Register/frame pointer
x9/s1	Saved Register
x10/a0	Function Arguments/return values
x11/a1	Function Arguments/return values
x12/a2	Function Arguments
x13/a3	Function Arguments
x14/a4	Function Arguments
x15/a5	Function Arguments

x16/a6	Function Arguments
x17/a7	Function Arguments
x18/s2	Saved Register
x19/s3	Saved Register
x20/s4	Saved Register
x21/s5	Saved Register
x22/ss6	Saved Register
x23/s7	Saved Register
x24/s8	Saved Register
x25/s8	Saved Register
x26/s10	Saved Register
x27/s11	Saved Register
x28/t3	Temporary Register
x29/t4	Temporary Register
x30/t5	Temporary Register
x31/t6	Temporary Register
pc	Program Counter
Supervisor State	
sstatus	Supervisor status register
stvec	Supervisor Trap Vector Base Address Register

sie	Supervisor Interrupt-Enable Register
sip	Supervisor Interrupt-Pending Register
scounteren	Counter-Enable Register
sscratch	Supervisor Scratch Register
sepc	Supervisor Exception Program Counter
scause	Supervisor Cause Register
stval	Supervisor Trap Value Register
senvcfg	Supervisor Environment Configuration Register
satp	Supervisor Address Translation and Protection Register
Machine State	
misa	Machine ISA register
mvendorid	Machine Vendor ID register
marchid	Machine Architecture ID register
mimpid	Machine Implementation ID register
mhartid	Hart ID register
mstatus	Machine Status Register
mstatush	Machine Status Register
mtvec	Machine Trap-Vector Base-Address Register
medeleg	Machine Exception Delegation Register
mideleg	Machine Interrupt Delegation Register

mip	Machine Interrupt-Pending Register
mie	Machine Interrupt-Enable Register
mcycle	Machine clock cycle count register
minstret	Machine instruction retired count register
mcounteren	Machine Counter-Enable Register
mcountinhibit	Machine Counter-Inhibit register
mscratch	Machine Scratch Register
mepc	Machine Exception Program Counter register
mcause	Machine Cause Register
mtval	Machine Trap Value Register
mconfigptr	Machine Configuration Pointer Register
mseccfg	Machine Configuration Pointer Register
mtime	Machine time register (memory -mapped control register)
mtimecmp	Machine time compare register (memory -mapped control register)