
Sirius 设计文档



目 录

| | | |
|----------|-------------------|----------|
| 1 | 设计概述 | 1 |
| 1.1 | 指令系统 | 1 |
| 1.2 | 内存管理 | 1 |
| 1.3 | 缓存 | 1 |
| 1.4 | 异常处理 | 2 |
| 1.5 | CP0 | 2 |
| 2 | Sirius CPU | 3 |
| 2.1 | 总体设计 | 3 |
| 2.2 | 接口定义 | 3 |
| 2.3 | 取指阶段 | 6 |
| 2.4 | 指令 FIFO | 6 |
| 2.5 | 译码阶段 | 7 |
| 2.6 | 执行阶段 | 9 |
| 2.7 | 访存阶段 | 12 |
| 2.8 | 写回阶段 | 12 |

第 1 章 设计概述

比赛的核心要求是实现一个部分兼容 MIPS32 体系结构的小端序 CPU，并在其基础上搭建 SoC。为了保证系统的简洁性，我们的 CPU 实现了最常用的部分 MIPS32 指令以及运行操作系统等高级应用所必须的协处理器 CP0 等特权资源。

我们实现的处理器命名为 Sirius，采用了双发射五级流水线的设计 [1]，并支持一级指令、数据缓存，以提升系统性能。一级缓存目前采用直接映射的方式，行宽 64 字节，大小为 8 KB。

通过运行功能测试以及性能测试，我们一定程度上证明了该处理器的正确性。

1.1 指令系统

根据大赛要求，我们实现的指令系统为 MIPS32[2][3] [4] 指令系统的子集，准确的说，我们实现的指令系统包含全部 MIPS I 中的非浮点部分以及 MIPS32 中为了实现从异常返回而添加的 ERET 指令。该指令系统包含以下几种指令类型：

- 算数运算指令 包括 ADD、SUB、SLT、MULT、DIV 等指令
- 逻辑运算指令 包括 AND、OR、XOR、NOR、LUI 等指令
- 移位运算指令 包括 SLL、SRL、SRA、SLLV、SRAV 等指令
- 分支跳转指令 包括 J、JR、JAL、B、BEQ 等指令
- 内存访问指令 包括 LB、LH、LW、SB、SW 等指令
- 系统控制指令 包括 SYSCALL、BREAK、MFC0、MTC0 等指令

大赛文档给出的 57 条指令，我们均已正常实现。CPU 中包含的 32 个通用寄存器，HI、LO 寄存器，以及协处理器 (CP0) 中实现的寄存器均按照 MIPS32r1 的规范实现。

基于性能方面的考虑，我们的 CPU 实现为非对称双发射五级流水线，通过对数据通路和控制模块的精心设计，我们尽可能地解决了因数据冲突、控制冲突以及结构冲突导致的流水线暂停。在我们的设计中，MIPS 传统的延迟槽技术得以保留，任一分支跳转指令后面均有深度为一条指令的延迟槽，可以用于编译器优化。

1.2 内存管理

处理器的 MMU（内存管理）模块负责将虚地址映射为对应的物理地址，我们的 CPU 目前只实现了 MIPS32r1 规范中规定的 kseg0 以及 kseg1 两段内存地址的映射，这两部分的映射方式很简单。仅仅将物理地址的高三位置为 0 即可。后续的版本会实现完整的内存地址映射。

1.3 缓存

在实际系统中，访存带宽与处理器运算能力的不匹配，使得访存系统的性能成为了处理器系统性能的瓶颈。考虑到缓存能够直接减少微处理器与慢速的内存设备之间的差距。我们的设计引入了缓存模块，用于进一步提升处理器的性能。

我们实现的 L1 Cache 共分为两部分：L1 指令缓存以及 L1 数据缓存。两者配置完全相同，大小为 8 KB，行宽 64 字节，采用直接映射的方式进行替换。当命中时 L1 Cache 可以在单周期内返回数据，否则就会暂停流水线与内存设备进行交互。L1 Cache 与 MMU 结合在一起，对外采用

AXI 接口，支持猝发 (Burst) 传输，一次传输请求可以处理一整行缓存数据，以此提高总线的利用率。

1.4 异常处理

为了支撑操作系统的运行，提高处理器的通用性，Sirius 支持异常处理。根据 MIPS 规范的要求，Sirius 支持精确异常。即 CPU 能准确记录发生异常的指令位置（包括位于延迟槽中的指令），并确保异常发生之前的指令均完全执行，且让发生异常的指令及之后的指令取消执行。按照优先级排序，我们的 CPU 支持如下异常：

1. **中断** 外部中断、软中断、计时器中断。是否触发中断由协处理器 0 中的 Cause.IP 和 Status.IM 两个字段共同决定。
2. **地址错例外：取指** 在取址阶段地址不对齐时触发该异常。
3. **保留指令例外** 在解码阶段发现该指令无法解码时触发该异常。
4. **自陷、系统调用** 当执行了对应的指令时触发这些异常。
5. **整形溢出** 当执行 ADD、SUB、ADDI、SUBI 时发生溢出时触发该异常。
6. **地址错例外：访存** 在访存阶段地址不对齐时触发该异常。

异常发生后，PC 会被置为 0xBFC00380，开始异常处理流程。对于已经实现的异常，其处理流程遵循比赛要求的处理流程。中断会受到 CP0 寄存器的影响。Sirius 支持 6 个硬件中断，它们分别由 CP0 中的 Status.IM7 至 Status.IM2 位控制，对应位为 1 时中断启用。软中断则由 Status.IM1 和 Status.IM0 控制。中断仅在处理器正常状态下才可以发生。

1.5 CP0

CP0 本质上为一组寄存器。Sirius 实现了大赛文档要求的全部 CP0 寄存器。同时为了支撑后续的操作系统的实现，我们拟额外实现一些 MIPS32r1 规范中要求的寄存器。详见附录。

第 2 章 Sirius CPU

2.1 总体设计

Sirius CPU 设计为双发射五级流水线，分为取指、解码、执行、访存以及写回五个流水级。在取指阶段，PC 中的值被送往 MMU，并在 MMU 被翻译为对应的物理地址，并被送往 MMU 中取指相关的部分。如果该虚地址所在的区域可以被缓存，MMU 会先在内置的 L1 指令缓存内查询该指令。如果缓存命中则在单周期内返回指令数据，否则会发起访存请求充填流水线并暂停处理器。如果虚地址所在的区域不可被缓存，则直接发起访存请求并暂停流水线直到数据返回。取到的指令以及与之对应的 PC 地址会被送到指令 FIFO 内等待被执行。在译码阶段，指令 FIFO 的前两条指令会被取出，并被翻译为处理器内部使用的指令。最后双发射判断逻辑会根据情况来决定是否发射第二条指令，同时指令 FIFO 会根据双发射判断逻辑的结果确定是否删除刚刚被取出的两条指令。在执行阶段，最多两条指令会并行地在两个 ALU 内执行。同时在执行阶段，分支跳转等指令也在此执行，一旦确定需要跳转，需要跳转到的地址会被送回到 PC，PC 会在下一周期更新到目的地址。执行阶段产生的结果会被送往访存阶段，在访存阶段首先会处理异常，一旦有异常发生则刷新流水线。如果没有任何异常产生则正常进行访存操作。访存操作会将数据以及地址等控制信息发送到 MMU 内，MMU 依然负责虚拟地址转换，L1 数据缓存查找等操作。访存阶段的结果会被送到写回阶段。在写回阶段，运算/访存结果被写入寄存器中。

为了处理数据冲突问题，我们的处理器引入了旁路机制，各个阶段的运行结果都会被前推到解码阶段。当通过前推无法解决数据冲突时 (如 load-use 冲突)，流水线会暂停一拍等待结果返回。对于跳转指令导致的控制冲突，我们使用 MIPS 的延迟槽机制尽可能的使流水线不中断。在后续还将考虑引入分支预测器以及循环缓冲区进一步保证指令流不被打断。

为了遵循 MIPS32r1 的规范，Sirius CPU 实现了精确异常处理。其实现方法是当异常发生时，不直接处理，而是将之保存在流水线的寄存器内，直到访存阶段时统一处理。一旦发生异常，则刷新流水线更新 PC。

Sirius CPU 的流水线设计图见图 2.1。需要注意的是，在这个设计图中，为了简化图片的复杂度，我们没有画出旁路、流水线控制以及流水寄存器等细节。

CPU 部分的 RTL 代码使用 SystemVerilog 写就，核心代码除了乘除法部分为了减少资源占用提升性能使用了 Xilinx 的 IP 核外，均为独立完成。代码存放于 [GitHub](#) 中（暂未公开）。

2.2 接口定义

Sirius 对外接口为标准的 AXI 接口。详见 ARM 的规范。

Sirius 内部主要分为两部分，MMU 以及核心部分，核心部分的接口如表 2.1 所示。其中 “inst_*” 与 “data_*” 信号是与 MMU 连接的信号，MMU 则视情况使用 AXI 的接口对外部设备发起访问请求。

当需要进行访存时，Sirius 会把对应的使能信号置为 1，同时在地址信号内给出需要访存的地址。如果数据可用，则对应的 “*_ok” 信号会被置为 1，如果对应数据不可用，流水线会被暂停直到数据可用。

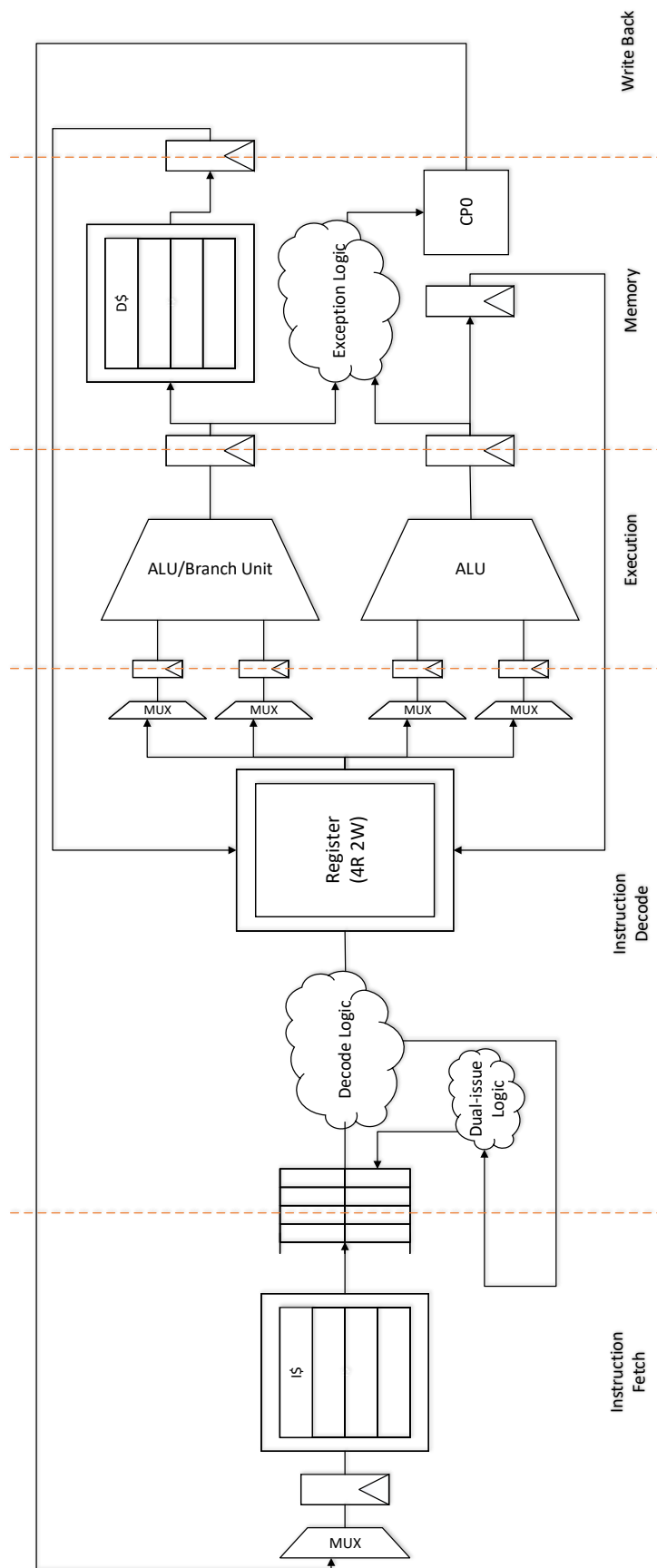


图 2.1: Sirius 流水线设计简图

| Signal Name | Width | Direction | Description |
|-------------|--------|-----------|--------------------------------------|
| clk | 1 | Input | 时钟信号 |
| rst | 1 | Input | 总体复位信号，高有效 |
| interrupt | 5...0 | Input | 中断信号，高有效 |
| inst_en | 1 | Output | 指令使能，该信号为 1 表示进行取指操作 |
| inst_addr | 31...0 | Output | 指令地址 |
| inst_ok | 1 | Input | 返回的指令是否有效，当该信号为 0 时表示指令暂未返回，需要暂停流水线 |
| inst_ok_1 | 1 | Input | 返回的第一条指令是否为有效指令 |
| inst_ok_2 | 1 | Input | 返回的第二条指令通常是当第一条指令位于缓存行的最后时，该信号会被置为 0 |
| inst_data_1 | 31...0 | Input | 读到的第一条指令数据 |
| inst_data_2 | 31...0 | Input | 读到的第二条指令数据 |
| data_en | 1 | Output | 数据使能，该信号为 1 表示进行取指操作 |
| data_wen | 3...0 | Output | 字节写使能，确定需要修改那个字节 |
| data_addr | 31...0 | Output | 数据地址 |
| data_wdata | 31...0 | Output | 写数据 |
| data_ok | 1 | Input | 返回的数据是否有效，当该信号为 0 时表示数据暂未返回，需要暂停流水线 |
| data_data | 31...0 | Input | 读到的数据 |

表 2.1: Sirius 核心接口定义

| Signal Name | Width | Direction | Description |
|-------------------|--------|-----------|--------------------------------------|
| clk | 1 | Input | 时钟信号 |
| rst | 1 | Input | 总体复位信号，高有效 |
| pc_en | 1 | Input | PC 使能，当该信号为 0 时，PC 寄存器不能被修改 |
| inst_ok_1 | 1 | Input | 返回的第一条指令是否为有效指令 |
| inst_ok_2 | 1 | Input | 返回的第二条指令通常是当第一条指令位于缓存行的最后时，该信号会被置为 0 |
| fifo_full | 1 | Input | 指令 FIFO 是否为满，当 FIFO 满载时，PC 寄存器也不得被修改 |
| branch_taken | 1 | Input | 后续流水级是否发生了跳转 |
| branch_address | 31...0 | Input | 跳转的目的地址 |
| exception_taken | 1 | Input | 后续流水级是否发生异常 |
| exception_address | 31...0 | Input | 发生异常时跳转到的地址 |
| pc_address | 31...0 | Output | 输出的 PC 地址 |

表 2.2: PC 接口定义

2.3 取指阶段

取指阶段主要就是 PC 寄存器 (**pc.sv**)，该模块的接口定义见图 2.2。

PC 寄存器内包含要被取指的下一条指令的地址。正常情况下，PC 会根据两个 inst_ok 信号确定下一周期时候 PC 是加 4 或者是加 8。当异常发生时，PC 被置为异常模块指定的地址，通常发生异常时该模块给出的地址为 0xBFC00380，但是当发出异常信号的指令为 ERET 时，异常模块给定的地址为 EPC 寄存器内的值。

在取指阶段得到的指令会和该指令对应的 PC 值一并放入指令 FIFO 内。

在取指过程中，可能会发生取指地址不对齐等异常。当在此发生异常时，我们不会立即处理响应异常，而是在后续的访存流水级统一处理，此时我们要做的仅仅是将发生的异常记录下来等待后续处理。

PC 在初始化时被置为 0xBFC0_0000。

2.4 指令 FIFO

指令 FIFO(**instruction_fifo.sv**) 的作用是分割取指流水级以及后续的译码流水级。事实上，指令 FIFO 也将整个处理器划分为两个部分，即用于尽可能地获取连续的指令流的前端以及执行指令的后端。

指令 FIFO 的实现原理见图 2.2。其核心部分就是一个用于存储指令的环形缓冲区以及分别用于读写缓冲区的指针。在我们的实现中为了支持双发射相关的逻辑，我们拓展了读写指针，使得该指令 FIFO 最多可以同时读取和存储两条指令。同时为了支撑 MIPS 要求的延迟槽机制的正确运行，该 FIFO 还有还实现有一段区域用于在因分支指令清空 FIFO 时暂存本应位于延迟槽内的指令。

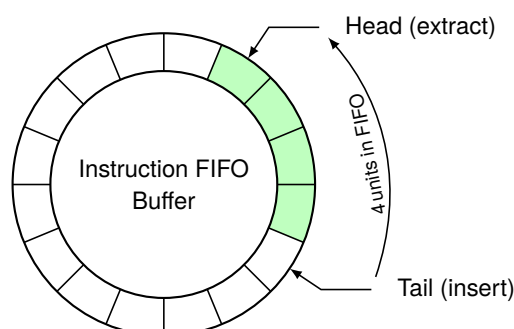


图 2.2: 指令 FIFO 原理图

2.5 译码阶段

译码阶段主要负责指令解码，通用寄存器读取，双发射判断等工作。用于处理数据冲突的旁路模块也在此处运行。

译码阶段全部逻辑均为组合逻辑。

2.5.1 指令解码模块

指令解码模块由两个.sv 文件构成，分别为 **decoder_alpha.sv** 以及 **decoder_ctrl.sv**。

decoder_alpha 模块负责将完整的指令按照 MIPS 的编码方式切分为不同的字段，以供译码阶段的各个模块使用，**decoder_ctrl** 模块则根据将指令转换为 Sirius 内部的控制信号，并提供给后续的流水级使用。**\$0** 寄存器。

2.5.2 寄存器文件模块

寄存器模块在解码阶段以及写回阶段均被使用。寄存器文件内部为 32 个 32 位的寄存器，实现了 4 个读端口以及 2 个写端口。需要注意的是，在 MIPS32r1 规范中要求 **\$0** 寄存器读出的数必须为 0，但是我们没有在寄存器文件内实现这一机制，我们把处理这一要求的逻辑放置在了后续的旁路模块内。

寄存器文件的读端口为组合逻辑，可以在单周期内返回需要的数据；写端口部分为时序逻辑，写的数据会在下一周期存放到寄存器内。当对一个寄存器号同时进行读写操作时，读出的为要写入的值。

2.5.3 旁路模块

在译码阶段我们最多可能需要读四个通用寄存器的数值，因此我们需要实例化四个旁路模块，全部旁路模块使用的都是同一套代码，代码位于 **forwardin_unit.sv**，其接口定义见表 2.3。

因为引入了双发射技术，对于主流水线和辅助流水线之间的前推的优先级显得尤为重要，我们实现的旁路模块的各个阶段的优先级如图 2.3。旁路模块会按照优先级顺序确定各个阶段是否有写寄存器文件的请求以及写寄存器号是否与读的寄存器号相同，如果相同则将该阶段的结果直接作为寄存器读的结果送入下一流水级。如果各个各个流水级都无需前推，则直接将从寄存器文件内读出的值送入下一流水级。

需要注意的是，如同上一节所叙，我们将处理读取 **\$0** 寄存器的逻辑放在了这一模块内。具体的做法是：当判断当前读的寄存器号为 **\$0** 时，直接返回 0，不做任何前推处理。

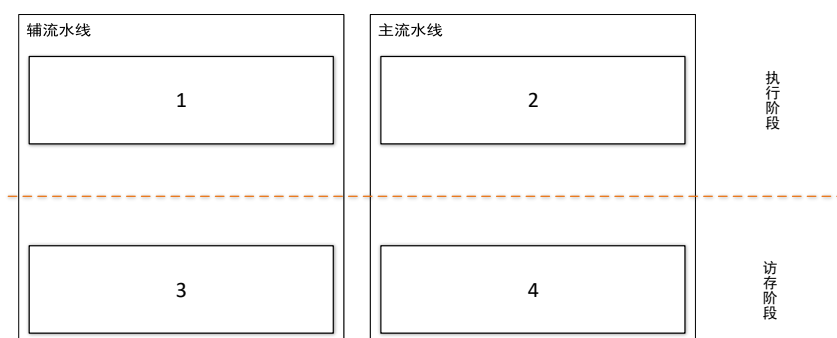


图 2.3: 前推模块的优先级

| Signal Name | Width | Direction | Description |
|-------------------|--------|-----------|--|
| slave_ex_reg_en | 1 | Input | 寄存器使能, 当前位于辅流水线执行阶段的指令需要写寄存器时, 该信号置为 1 |
| slave_ex_addr | 4...0 | Input | 当前位于辅流水线执行阶段的指令需要写的寄存器号 |
| slave_ex_data | 31...0 | Input | 当前位于辅流水线执行阶段的结果 |
| master_ex_reg_en | 1 | Input | 寄存器使能, 当前位于主流水线执行阶段的指令需要写寄存器时, 该信号置为 1 |
| master_ex_addr | 4...0 | Input | 当前位于主流水线执行阶段的指令需要写的寄存器号 |
| master_ex_data | 31...0 | Input | 当前位于主流水线执行阶段的结果 |
| slave_mem_reg_en | 1 | Input | 寄存器使能, 当前位于辅流水线访存阶段的指令需要写寄存器时, 该信号置为 1 |
| slave_mem_addr | 4...0 | Input | 当前位于辅流水线访存阶段的指令需要写的寄存器号 |
| slave_mem_data | 31...0 | Input | 当前位于辅流水线访存阶段的结果 |
| master_mem_reg_en | 1 | Input | 寄存器使能, 当前位于主流水线访存阶段的指令需要写寄存器时, 该信号置为 1 |
| master_mem_addr | 4...0 | Input | 当前位于主流水线访存阶段的指令需要写的寄存器号 |
| master_mem_data | 31...0 | Input | 当前位于主流水线访存阶段的结果 |
| reg_addr | 4...0 | Input | 解码阶段需要读的寄存器号 |
| reg_data | 31...0 | Input | 从寄存器文件中读出的结果 |
| result_data | 31...0 | Output | 前推后的结果 |

表 2.3: 旁路单元接口定义

| Signal Name | Width | Direction | Description |
|----------------------------|-------|-----------|-----------------------------|
| id_priv_inst_master | 1 | Input | 主流水线当前在解码的指令是否为特权指令 |
| id_wb_reg_dest_master | 4...0 | Input | 主流水线当前解码的指令要写的寄存器号 |
| id_wb_reg_en_master | 1 | Input | 主流水线当前解码的指令的寄存器文件写使能信号 |
| id_is_hilo_accessed_master | 1 | Input | 主流水线当前解码的指令是否需要使用到 HiLo 寄存器 |
| id_opcode_slave | 5...0 | Input | 辅流水线当前解码指令的 OpCode |
| id_rs_slave | 4...0 | Input | 辅流水线的源寄存器号 |
| id_rt_slave | 4...0 | Input | 辅流水线的另一个源寄存器号 |
| id_mem_type_slave | 1...0 | Input | 辅流水线当前解码的是否需要访存 |
| id_is_branch_instr_slave | 1 | Input | 辅流水线当前解码的是否是分支跳转指令 |
| id_priv_inst_slave | 1 | Input | 辅流水线当前解码的是否是特权指令 |
| id_is_hilo_accessed_slave | 1 | Input | 辅流水线当前指令是否需要访问 HiLo |
| fifo_empty | 1 | Input | 指令 FIFO 是否为空 |
| fifo_almost_empty | 1 | Input | 指令 FIFO 是否仅剩一条指令 |
| enable_master | 1 | Input | 主流水线是否在正常执行指令 |
| enable_slave | 1 | Input | 是否允许辅流水线当前解码的指令发射 |

表 2.4: 双发射判断模块接口定义

2.5.4 双发射判断模块

双发射判断模块 (**dual_engine.sv**) 的主要作用是用于判断在当前的情况下是否可以进行双发射。该模块的接口定义见表 2.4。

在目前，确定辅流水线是否发射仅需要遵循如下 5 条简单的规则：

1. 主流水线不发射，辅流水线必定不发射
2. 指令 FIFO 仅剩一条指令或者无指令，辅流水线必定不发射
3. 当主辅流水线存在访问 HiLo 寄存器的指令或者特权指令时，辅流水线必定不发射
4. 当辅流水线中的指令辅流水线无法执行时，辅流水线必定不发射
5. 当主流水线中的指令与辅流水线的指令之间存在写后读 (RAW) 相关时，辅流水线必定不发射

主流水线以及辅流水线是否会发射的信号会被送回到指令 FIFO 内，指令 FIFO 后续会根据这两个信号更新读指针的位置。

2.6 执行阶段

执行阶段负责执行分支指令，完成算术逻辑运算以及 (对于访存指令) 生成访存地址等工作。

大多数的简单的算术逻辑运算可以在 1 周期内完成，但是对于比较 **DIV**、**MULT** 等较为复杂的指令，需要 4~34 个周期来完成，此时就会阻塞流水线等待指令执行完毕。不过好在多周期指令出现的频率极低，因此直接阻塞不会较大地对影响性能。

我们的乘除法均使用 Xilinx® 提供的 IP 核实现，其中乘法使用 FPGA 内置的 DSP Slice 实现，

| Signal Name | Width | Direction | Description |
|-----------------|--------|-----------|----------------|
| en | 1 | Input | 使能信号 |
| pc_address | 31...0 | Input | 当前执行的指令的 PC 地址 |
| instruction | 31...0 | Input | 当前执行指令 |
| is_branch_instr | 1 | Input | 该指令是否为跳转分支类指令 |
| branch_type | 2...0 | Input | 跳转类型 |
| data_rs | 31...0 | Input | RS 寄存器内的值 |
| data_rt | 31...0 | Input | RT 寄存器内的值 |
| branch_taken | 1 | Output | 是否跳转 |
| branch_address | 31...0 | Output | 跳转目的地址 |

表 2.5: 分支单元接口定义

4 个周期即可返回结果。除法则需要 34 个周期。

2.6.1 分支单元

分支单元 (**branch.sv**) 用于处理分支指令。其接口见表 2.5。如果该模块发现当前的指令为分支指令的话, 就会根据指令的类型来判断是否进行跳转, 其输出信号会被回送到 PC 以及指令 FIFO 中。

2.6.2 ALU

ALU 是整个执行阶段, 乃至整个 CPU 中最重要的模块之一。Sirius 内置了两个 ALU, 分别用于主辅流水线。在这里我们主要介绍在主流水线上使用的 ALU, 即 ALU α , 用于辅流水线上的 ALU β 为 ALU α 取出乘除法以及特权指令后得来的。不再详细介绍。

ALU α 的代码位于 **alu_alpha.sv**, 其接口定义见表 2.6。

对于大多数比较简单的算术逻辑指令, 我们直接通过调用 SystemVerilog 内置的运算符即可完成, 但是对于乘、除法这种操作而言, 直接调用运算符进行计算必然会导致执行周期的周期过长, 无法有效地提升主频。因此我们在这里使用 Xilinx® 提供的 IP 核调用对应的片上资源进行运算, 并辅以计数器以及运算前/后的求补器完成各种有符号/无符号乘法运算。

当 ALU 执行过程中出现异常时, 对应的 “exp_*” 信号会被置为 1, 并随着流水线进入到下一层级, 并在下一层级统一处理。

2.6.3 HiLo 寄存器

HiLo 寄存器对外表现为一对寄存器, 即 Hi、Lo 两个 32 位的寄存器。在我们的实现内, Hi、Lo 寄存器被作为一个整体实现, 即实现为一个 64 位的寄存器。Hi 寄存器为这一寄存器的高 32 位, Lo 寄存器为该寄存器的低 32 位的寄存器。

HiLo 寄存器内也实现了一个规模较小的旁路机制, 即访存阶段以及写回阶段还未写入到寄存器内部的数值也可以被前推到执行阶段需要访问 HiLo 寄存器时候获得。

| Signal Name | Width | Direction | Description |
|--------------|--------|-----------|---------------------------------|
| clk | 1 | Input | 时钟信号 |
| rst | 1 | Input | 复位信号 |
| flush_i | 1 | Input | 流水线刷新信号 |
| alu_op | 5...0 | Input | ALU 操作码 |
| src_a | 31...0 | Input | 源操作数 1 |
| src_b | 31...0 | Input | 源操作数 2 |
| src_hilo | 63...0 | Input | HiLo 寄存器的值 |
| rd | 4...0 | Input | rd 地址 |
| sel | 2...0 | Input | sel 内容 |
| cop0_addr | 7...0 | Output | 如果该指令为 MFC0, 该信号指示从 CP0 中读取什么结果 |
| cop0_data | 31...0 | Input | 从 CP0 中读到的结果 |
| cop0_wen | 1 | Output | CP0 写使能 |
| exp_overflow | 1 | Output | 是否发生了溢出异常 |
| exp_eret | 1 | Output | 是否正在执行 ERET 指令 |
| exp_syscal | 1 | Output | 是否正在执行 ERET 指令 |
| exp_break | 1 | Output | 是否正在执行 break 指令 |
| hilo_wen | 1 | Output | HiLo 寄存器写使能 |
| hilo_result | 63...0 | Output | HiLo 寄存器输出结果 |
| result | 31...0 | Output | 运算结果 |
| stall_o | 1 | Output | 流水线是否需要暂停 |

表 2.6: ALU α 接口定义

2.7 访存阶段

访存阶段主要负责访问存储器、处理异常。

访存阶段在获取执行阶段送来的访存信号以及地址信息后，根据这些信息向 MMU 发出对应的请求。因为内存访问通常是按字对齐的，但是 MIPS 除了支持 LW、SW 等直接操作整个字的指令外，还需要支持 LH、SH、LB、SB 等操作半字乃至字节的指令甚至是一直作为 MIPS 特色的 LWL、SWL 等非对其访问指令。为了支持这些指令，访存单元在获取所需信号后会将操作转换为 (操作类型，字节使能，操作地址，写数据 (如果有)) 的一个四元组，并发送到 MMU 中，MMU 再根据情况决定是对 Data Cache 进行操作还是开始动用 AXI 总线访问外部设备。

在进行实际的访存指令之前，异常模块会先行检查是否存在异常，如果存在异常则取消当前的指令并把对应的 PC 值按照 MIPS 的规范放入 CP0 的 EPC 寄存器内，同时设置好处理异常所需的各个 CP0 寄存器。中断作为特殊的异常也会在此被处理，异常处理逻辑内存在对应的逻辑来处理异常可能会打断正常的流水线执行等问题。

辅流水线在访存阶段什么都不做，仅仅是在未发生异常的情况下直接将所需信号传送到下一流水级等待处理。

2.8 写回阶段

写回阶段是 Sirius 流水线的最后一个阶段，在该阶段进行的主要是向寄存器文件以及 HiLo 寄存器写入结果。不再赘述。

参考文献

- [1] PATTERSON D A, HENNESSY J L. Computer organization and design, fifth edition: The hardware/software interface[M]. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- [2] Mips architecture for programmers volume i-a: Introduction to the mips32 architecture[M]. Revision 3.02 ed. 955 East Arques Avenue Sunnyvale, CA 94085-4521: MIPS Technologies, Inc., 2011.
- [3] Mips architecture for programmers volume ii-a: The mips32 instruction set[M]. Revision 3.02 ed. 955 East Arques Avenue Sunnyvale, CA 94085-4521: MIPS Technologies, Inc., 2011.
- [4] Mips architecture for programmers volume iii: The mips32 and micromips32 privileged resource architecture[M]. Revision 3.02 ed. 955 East Arques Avenue Sunnyvale, CA 94085-4521: MIPS Technologies, Inc., 2011.