第 5 章

指令发射

指令发射(Instruction Issue)的目的是将译码后的指令发送到处理器的运算单元，由运算单元完成指令的执行。根据每个时钟周期发射指令的数量，指令发射可分为单发射和多发射；根据指令发射的调度方式则可以分为顺序发射和乱序发射。本章首先介绍指令发射单元的基本概念和相关调度算法，然后以开源 Ariane 处理器核为例介绍指令发射单元设计的细节。

5.1 单发射和多发射

单发射是指处理器一个时钟周期只发射一条指令，而多发射是指一个时钟周期能够发射多条指令。多发射技术包括动态多发射和静态多发射。动态多发射由硬件决定发射的指令数。流水线通过增加运算单元，在同一个时钟周期内支持多条指令同时工作从而实现指令级并行。静态多发射通过编译器预先编排指令的方式来实现处理器内部指令的并行执行，如**超长指令字**(Very Long Instruction Word，VLIW)处理器，VLIW 的实现过程是由编译器在编译时找出指令间潜在的并行性，进行适当调度安排，把多个能并行执行的操作组合在一起，成为一条具有多个操作段的超长指令。

多发射相比单发射具有更宽的数据通路，流水线上每个阶段处理的指令数也更多，因此多发射相比单发射具有更高的数据吞吐率。但是，在多发射中数据的相关性要比单发射更加复杂，在同一阶段多条指令可能会存在数据相关。

5.2 顺序发射和乱序发射

顺序发射是指处理器按照程序原始二进制指令流的顺序将指令发射到执行单元，因此需要等到前序指令都已发出，同时源操作数和硬件资源已经就绪才会发射新的指令。乱序发射是指译码后的指令被分配到发射队列中，发射队列中的指令去除相关性后就可以优先发射而不需要等待按序发射，乱序发射队列通常可以分为分布式发射队列和统一发射队列。

分布式发射队列是指不同类型的**功能单元**(Function Units，FU)具有独立的发射队列，每个发射队列只负责向对应的功能单元发射指令，只要功能单元空闲和源操作数可用就可以执行发射。该方式可以降低设计复杂度，但效率较低。不同功能单元的使用率不同，导致对应的发射队列的使用率也会有所差异，如浮点发射队列已满，而整数队列出现空闲的情况。

与分布式发射队列不同，统一发射队列存储所有的发射指令，指令发射的过程可以分为发射前读取源操作数和发射后读取源操作数两种。发射前读取源操作数是指寄存器源操作数在指令发送到发送队列之前被读取，并存储在发射队列中，当操作数可用且功能单元空闲时，从发射队列发出指令用于执行，这种方式需要大量的硬件资源来存储源操作数。发射后读取源操作数是指源操作数在指令发射时读取，发射队列中存储寄存器的标号，当实际执行时根据标号读取实际的值，因此寄存器需要更多的读取端口，保证同时读取多个数据。

乱序发射可以采用保留站来实现，保留站是每个功能单元的专用缓冲区，用于存储将要在功能单元上执行的指令及操作数。同时保留站具有寄存器重命名的功能，在发射阶段将待用操作数寄存器重命名为保留站的名称，消除因名称相关而产生的冒险。最后根据保留站中的标志信息决定哪条指令可以发射执行。

5.3　指令动态调度

指令动态调度是指处理器通过硬件重新安排指令执行的顺序，减少因指令的相关性而出现流水线停顿。相比于指令静态调度使用编译器来去除指令的相关性更高效，硬件复杂度也显著提高。在多级流水线中指令从指令译码到指令发射执行会存在数据相关和结构相关，为了解决这些相关性的问题，可以采用动态调度算法。经典的动态调度算法包括记分板算法和 Tomasulo 算法；记分板算法来自 CDC6600 处理器的设计；Tomasulo 算法是由 R. Tomasulo 提出来的，应用于 IBM 360/91 处理器的设计，主要是通过对寄存器的动态重命名来处理指令的相关性。这里主要介绍记分板算法。

为了更好地解释记分板算法，将 2.1 节介绍的经典 5 级流水线中的指令译码流水级拆分成发射和读操作数两个阶段。记分板深度参与如下 4 个阶段。

1. 发射阶段

检测结构相关或写后写(Write After Write，WAW)相关。如果待发射指令使用的功能单元空闲(该功能单元没有被前序活动指令占用)，或者待发射指令所使用的目的寄存器空闲(该目的寄存器没有被前序活动指令占用)，则将指令发射到功能单元。否则，停止发射当前指令并等待功能单元或目的寄存器被释放。

2. 读操作数阶段

检测写后读(Read After Write，RAW)相关。如果当前指令要读取的源操作数寄存器是前序活动指令的目的寄存器，则需要停止读操作数并等待前序指令执行完成，更新寄存器值。

3. 执行阶段

根据指令的类型，使用对应的功能单元执行指令，完成后通知记分板。

4. 写回阶段

检测读后写(Write After Read,WAR)相关。记分板收到指令执行完成的通知后,如果检测该指令要写入的目的寄存器是前序活动指令的源操作数寄存器,并且前序活动指令还没有完成读操作数的动作,则需要停顿已经完成执行阶段的指令的写回动作。

以下面的 RSIC-V 指令代码片段为例来进一步介绍记分板算法的执行流程。记分板算法通过表 5.1 所示的表格来记录指令的执行状态并进行动态调度,该表记录了第二条 lw 指令已经完成执行阶段,即将进入写回阶段时的处理器状态。

```
lw    x6, 34(x12)        //读取寄存器 mem[Regs[x12]+ 34]的操作数到 x6
lw    x2, 45(x13)        //读取寄存器 mem[Regs[x13]+ 45]的操作数到 x2
mul   x1,x2,x4           /寄存器 x4 和 x2 的操作数做乘法运算,结果存到 x1
sub   x8,x6,x2           /寄存器 x2 和 x6 的操作数做减法运算,结果存到 x8
div   x10,x1,x6          /寄存器 x6 和 x1 的操作数做除法运算,结果存到 x10
add   x6,x8,x2           /寄存器 x2 和 x8 的操作数做加法运算,结果存到 x6
```

表 5.1 中指令状态子表指示的是每条指令所处的阶段,其中数字代表时钟周期,指令按照时钟周期进行调度,还未执行的阶段用空白表示。第一条 lw 指令在执行时其他指令无法发射(第 1～4 个时钟周期),因为第二条 lw 指令与第一条 lw 指令存在结构性相关,只能等待第一条 lw 指令结果写回后才能发射(第 5 个时钟周期)。

表 5.1　记分板结构表

指令状态				
指　　令	发射	读操作数	执行	写回结果
lw　x6,34(x12)	1	2	3	4
lw　x2,45(x13)	5	6	7	
mul　x1,x2,x4	6			
sub　x8,x6,x2	7			
div　x10,x1,x6				
add　x6,x8,x2				

续表

功能单元状态									
名称	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	NO								
Mult1	YES	mul	x1	x2	x4	Integer		NO	YES
Mult2	NO								
Add	YES	sub	x8	x6	x2		Integer	YES	NO
Divide	NO								

寄存器状态									
寄存器	x1	x2	x4	x6	x8	x10	x12	…	x31
功能单元	Mult	Integer			Add				

表 5.1 的功能单元状态子表指示的是功能单元和寄存器的状态，表中各参数含义如下。

（1）Busy：指示功能单元是否空闲。

（2）Op：对源操作数 1 和源操作数 2 执行的运算。

（3）F_i、F_j、F_k：F_i 表示目的寄存器编号，F_j、F_k 分别表示源寄存器 1 和源寄存器 2 的编号。

（4）Q_j、Q_k：产生源操作数 1 和源操作数 2 的功能单元。

（5）R_j、R_k：源操作数 1 和源操作数 2 是否就绪，YES 表示就绪，NO 表示未就绪。

表 5.1 记录的是第 7 个时钟周期的状态。此时第二条 lw 指令执行完成，但结果未写回，后面二条指令已完成发射，但两条指令中的源操作数 x2 不可用，需要等待第二条 lw 指令将结果写回，也就是存在 RAW 相关。因此，表 5.1 中 Mult 一行，Op 指示该功能单元被第三条 mul 指令占用，R_j 为 NO 指示源操作数 1 未就绪，Q_j 指示产生源操作数 1 的功能单元是 Integer 单元。Add 一行，Op 指示该功能单元被第四条 sub 指令占用，R_k 为 NO 指示源操作数 2 未就绪，Q_k 指示产生源操作数 2 的功能单元是 Integer 单元。

记分板算法的缺陷是对于 WAW 和 WAR 相关，需要等待相关性解除后才能发射指令，造成流水线停顿，效率较低。而 Tomasulo 算法能够使用寄存器重命名来解决 WAW、WAR 相关，寄存器重命名是由保留站提供，保留站用于缓冲待发射指令的

操作数，同时将待发射的操作数寄存器重命名。

5.4 指令发射单元设计

5.1～5.3 节对指令发射单元的功能及记分板算法进行介绍，本节将以开源 Ariane 处理器核为例介绍指令发射单元设计的细节。Ariane 的指令发射单元是 Issue_Stage 模块，其顶层模块源代码文件是 issue_stage.sv。本节首先从模块顶层对其进行分析，介绍指令发射单元的内部结构及其外围连接关系，然后对发射单元中的 Scoreboard 模块、Issue_Read_Operands 模块进行详细分析。

5.4.1 整体设计

图 5.1 为**指令发射单元**(Issue_Stage)整体框图，内部包括 Re_Name、Scoreboard、Issue_Read_Operands 3 个子模块。

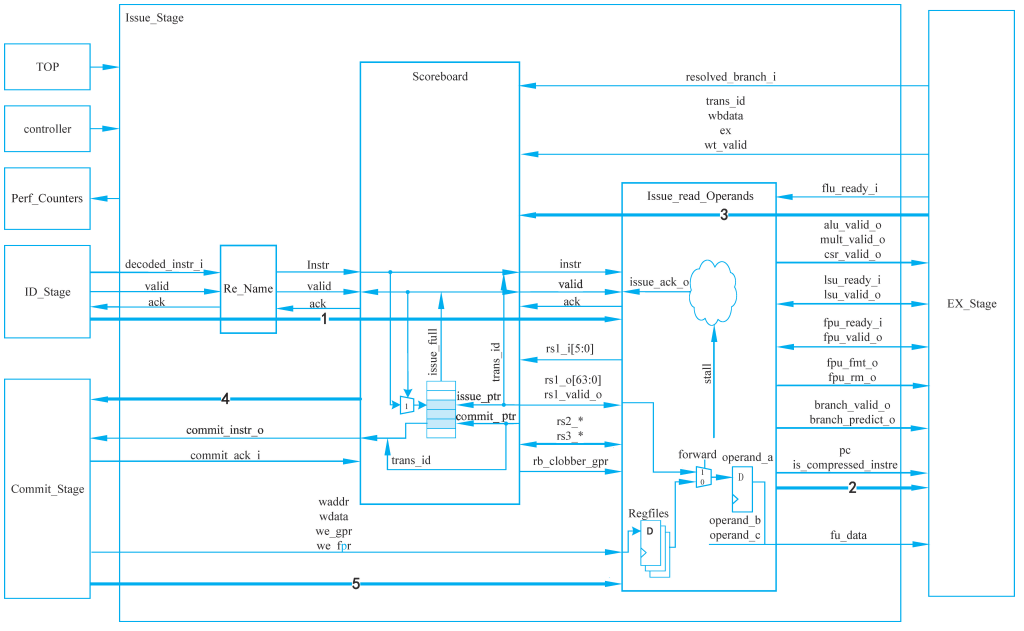

图 5.1 指令发射单元整体框图

（1）Re_Name 在 Regfiles 的索引上增加 1 位的重命名标志，但是目前的设计默认配置为关闭，因此相当于直接旁路。

（2）Scoreboard 维护一个发射队列。指令发射出去的同时被记录到 Scoreboard 中，该指令提交之后就从 Scoreboard 中撤销。指令执行单元（EX_Stage）执行后的结果被写入 Scoreboard。值得注意的是，由于不同指令执行的时间不一样，执行结果不是按照指令发射顺序写回 Scoreboard。

（3）Issue_Read_Operands 维护处理器核的 Regfiles，Regfiles 只能由指令提交单元（Commit_Stage）直接写入。该模块同时会读出操作数送给指令执行单元，根据 Scoreboard 的记录，操作数可能来自 Regfiles，也可能直接来自 Scoreboard。

图 5.1 中箭头标示指令发射单元（Issue_Stage）与指令译码单元（ID_Stage）、指令执行单元（EX_Stage）、指令提交单元（Commit_Stage）之间的数据交互。

（1）ID_Stage 将解码后的指令送到 Issue_Read_Operands 读取操作数。

（2）根据指令功能单元类型的不同，Issue_Read_Operands 将操作数发送到 EX_Stage 中对应的功能单元。

（3）EX_Stage 中的 FU 执行完后，将结果写回 Scoreboard 暂存，并将 valid 置 1。

（4）Scoreboard 中的 valid 被置 1 之后，指令可以被提交。

（5）Commit_Stage 返回 commit_ack，同时指令执行的结果直接被写入 Regfiles。

在流水线数据路径上，指令译码单元将译码后的指令，通过 decoded_instr_i 接口，以数据结构 scoreboard_entry_t 的形式发送给指令发射单元，关于该数据结构的详细定义可以见第 4 章。这两级流水线之间通过握手信号 valid、ack 进行数据交互的控制。指令发射单元检测待发射指令要使用的功能单元是否空闲，解除数据相关性，并进行操作数读取之后，将该指令发送到指令执行单元中对应的功能单元进行处理。

指令发射单元与指令执行单元的接口信号可以分成两类：数据接口和控制接口。数据接口是 fu_data_o，不管指令要被发射到哪个功能单元中执行，它们都通过该接口进行数据传递。fu_data_o 是 fu_data_t 类型的数据结构，其具体定义如下：

```
ariane_pkg.sv
typedef struct packed {
fu_t                        fu;
fu_op                       operator;
logic [63:0]                operand_a;
logic [63:0]                operand_b;
```

```
logic [63:0]                    imm;
logic [TRANS_ID_BITS-1:0]       trans_id;
  } fu_data_t;
```

接口定义中 fu 是指令占用的功能单元，被定义成 fu_t 的数据类型，具体定义如下。

```
ariane_pkg.sv
typedef enum logic[3:0] {
    NONE,               //0
    LOAD,               //1
    STORE,              //2
    ALU,                //3
    CTRL_FLOW,          //4
    MULT,               //5
    CSR,                //6
    FPU,                //7
    FPU_VEC             //8
} fu_t;
```

operator 是指令具体要执行的运算，例如该指令是要执行 ADD 运算，或者 SUB 运算，或者 DIV 运算等。operator 被定义成 fu_op 数据类型，具体定义的运算类型与 RISC-V 指令集的定义是一致的，代码比较多，读者可以自行查阅 ariane_pkg.sv 文件。

trans_id 是这条指令在 Scoreboard 中的索引号。由于不同功能单元的延时不一样，指令执行单元的结果写回 Scoreboard 可能是乱序的，需要通过 trans_id 索引来判断数据属于哪条指令，以及要写入 Scoreboard 中的哪个表项。operand_a、operand_b、imm 是从指令中解析出来或者从通用寄存器组读取出来的操作数和立即数，是功能单元执行运算的输入数据。

指令发射单元与指令执行单元的控制接口，根据功能单元的不同，可以分成以下 3 类。

(1) **固定延迟单元**(FLU)控制接口：所有 FLU 共用一个 ready 信号 flu_ready_i，高电平表示 FLU 空闲，指令可以发射；低电平表示 FLU 被占用，指令需要等待 FLU 被释放。根据指令所使用的功能单元的不同，alu_valid_o、branch_valid_o、mult_valid_o、csr_valid_o 中的一个会被拉高，指示 fu_data_o 数据已经准备好，对应的功能单元可以取数据进行运算。

（2）**加载和存储单元**(LSU)控制接口：lsu_ready_i、lsu_valid_o，具体含义与 FLU 的 ready、valid 类似。

（3）**浮点处理单元**(FPU)控制接口：fpu_ready_i、fpu_valid_o，具体含义与 FLU 的 ready、valid 类似。

指令执行单元的功能单元执行完对应的运算之后，将结果通过写回端口先写回 Scoreboard 暂存，写回端口如下：

```
issue_stage.sv
input logic    [NR_WB_PORTS-1:0][TRANS_ID_BITS-1:0]  trans_id_i,
input bp_resolve_t                                   resolved_branch_i,
input logic    [NR_WB_PORTS-1:0][63:0]               wbdata_i,
input exception_t   [NR_WB_PORTS-1:0]                ex_ex_i,
input logic    [NR_WB_PORTS-1:0]                     wt_valid_i,
```

wt_valid_i 是写回数据有效的指示信号，当 wt_valid_i 置高时，分支解析结果 resolved_branch_i、运算结果 wbdata_i、异常信息 ex_ex_i 被写入 Scoreboard 中 trans_id 所索引到的表项中。需要注意的是，指令执行单元写回 Scoreboard 有 4 组独立的端口，其中，加载单元 load_unit、存储单元 stroe_unit、浮点处理单元 fpu 各自有自己独立的写回端口，而所有的 FLU 单元则共享同一个写回端口。

暂存在 Scoreboard 中的写回数据，需要通过指令提交单元写回寄存器组 Regfiles，由于处理器核的 Regfiles 也是放在指令发射单元中，因此指令发射模块与指令提交模块也有另外一组接口信号用于数据写回。当指令被指令提交模块确认可以提交时，结果通过下面这组接口直接写入 Regfiles 中：

```
issue_stage.sv       //commit port
input logic [NR_COMMIT_PORTS-1:0][4:0]    waddr_i,
input logic [NR_COMMIT_PORTS-1:0][63:0]   wdata_i,
input logic [NR_COMMIT_PORTS-1:0]         we_gpr_i,
input logic [NR_COMMIT_PORTS-1:0]         we_fpr_i,
```

waddr_i 是通用寄存器组的索引，wdata_i 是写回的数据，we_gpr_i 是写回整数 Regfiles 的指示信号，we_fpr_i 是写回浮点 Regfiles 的指示信号。wdata_i 在 we_gpr_i、we_fpr_i 的指示下，被写入 Regfiles 中 waddr_i 所索引到的寄存器中。表 5.2 为 Issue_Stage 模块与各模块之间的接口列表。

表 5.2　Issue_Stage 模块与各模块之间的接口

信　　号		方向	位宽/类型	描　　述
TOP	clk_i	输入	1	时钟
	rst_ni	输入	1	复位
Controller	flush_unissued_instr_i	输入	1	flush 信号
	flush_i	输入	1	flush 信号
Perf_Counters	sb_full_o	输出	1	Scoreboard 信号
EX_Stage	fu_data_o	输出	fu_data_t	给 FU 的数据结构,包含操作数
	pc_o	输出	VLEN	指令 PC 值
	is_compressed_instr_o	输出	1	压缩指令标志位
	flu_ready_i	输入	1	FU 握手信号
	alu_valid_o	输出	1	FU 握手信号
	branch_valid_o	输出	1	FU 握手信号
	branch_predict_o	输出	branch_predict_sbe_t	分支预测信息
	resolve_branch_i	输入	1	没有被使用
	lsu_ready_i	输入	1	FU 握手信号
	lsu_valid_o	输出	1	FU 握手信号
	mult_valid_o	输出	1	FU 握手信号
	fpu_ready_i	输入	1	FU 握手信号
	fpu_valid_o	输出	1	FU 握手信号
	fpu_fmt_o	输出	2	浮点指令 fmt 信息
	fpu_rm_o	输出	3	浮点指令 rm 信息
	csr_valid_o	输出	1	FU 握手信号
	resolved_branch_i	输入	bp_resolve_t	EX_Stage 反向返回真实分支执行情况
	trans_id_i	输入	NR_WB_PORTS * TRANS_ID_BITS	FU 写回端口索引
	wbdata_i	输入	NR_WB_PORTS * 64	FU 写回端口数据
	ex_ex_i	输入	NR_WB_PORTS	FU 写回端口异常
	wt_valid_i	输入	NR_WB_PORTS	FU 写回端口 valid 信号