第6章　MIPS 五段流水机制设计实验

6.1 理想流水线 CPU 设计实验

6.1.1 实验目的

学生了解 MIPS 五段指令流水线分段的基本概念，能设计流水接口部件，将课程实验中设计完成的单周期 CPU 改造成理想流水 CPU，并能正确运行无冒险冲突的理想流水线标准测试程序，能根据时空图简单分析流水 CPU 性能。

6.1.2 背景知识

MIPS 单周期 CPU 设计实现简单，控制器部分是纯组合逻辑电路，但该 CPU 所有指令执行时间均是一个相同的周期，即以速度最慢的指令作为设计其时钟周期的依据，如图6.1所示，单周期 CPU 的时钟频率取决于数据通路中的关键路径（最长路径），所以单周期 CPU 效率较低，性能不佳，现代处理器中已不再采用单周期方式，取而代之的多周期设计方式，而多周期 CPU 中流水 CPU 设计是目前的主流技术。

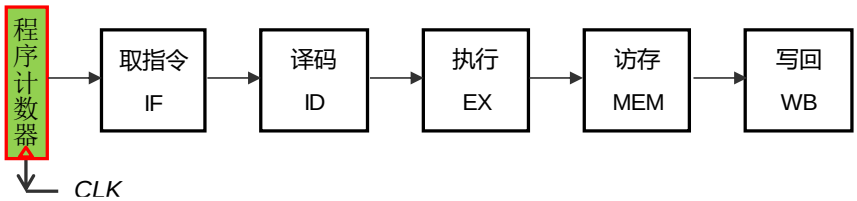

图 6.1 单周期 CPU 逻辑架构

流水线处理技术并不是计算机领域所特有的技术。在计算机出现之前，流水线技术已经在工业领域得到广泛应用，如汽车装配生产流水线等。计算机中的流水线技术是把一个复杂的任务分解为若干个阶段，每个阶段与其它阶段并行运行，其运行方式和工业流水线十分相似，因此被称为流水线技术。

把流水线技术应用于运算的执行过程，就形成了运算操作流水线，如浮点数加法运算可分解为求阶差、对阶、尾数加和规格化 4 个阶段。把流水线技术应用于指令的解释执行过程，就形成了指令流水线，如图 6.2 所示，MIPS 指令流水线通常将指令执行过程细分为取指令 IF、指令译码 ID、指令执行 EX、访存 MEM、写回 WB 共五个阶段，在每个阶段的后面都需要增加一个锁存器（又称为流水接口部件，用于锁存本段处理完成的数据或结果），以保证该阶段的执行结果给下一个阶段使用。

程序计数器、所有锁存器均采用公共时钟同步，每来一个时钟，各段逻辑功能部件处理完成的数据会锁存到锁存器中，作为下段的输入，指令的执行进入下一段。由于五个阶段逻辑延

迟时间并不一致，为保证指令流水线正确运行，最大时钟频率取决于五个阶段中最慢的阶段，所以分段时应该尽量让各阶段时间延迟相等，假设各段时间延迟均为 T。锁存器在公共时钟的驱动下可以锁存流水线前段加工完成的数据以及控制信号，锁存的数据和信号将用于后段的继续加工或处理。

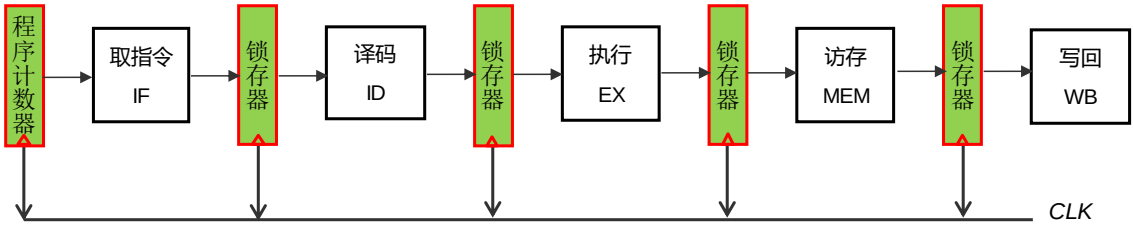

图 6.2 指令流水线逻辑架构

单周期 CPU 中由于各段组合逻辑完全串联，无法并行运行，其时空图如图 6.3 所示，图中横坐标表示时间，也就是连续指令进入流水线到流出流水线所经过的时间，当流水线中各段延迟时间相等时，横坐标被分割成相同长度的时间段，假设这里一个时间段为 T；纵坐标表示空间，即流水线的各段对应的功能部件。单周期 CPU 执行一条指令的时间为 5T。

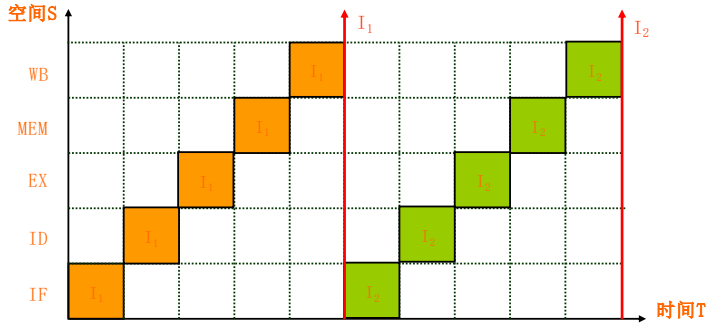

图 6.3 单周期 MIPS CPU 时空图

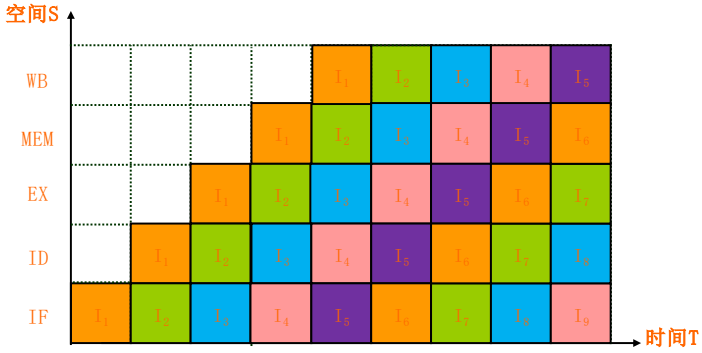

图 6.4 五段流水 MIPS CPU 时空图（无冲突）

MIPS 五段流水 CPU 中由于引入了锁存器，所以各段可以并行运行，如图 6.4 所示，当流水线充满后，每隔一个时钟周期 T，系统将流出一条执行完毕的指令，相比单周期 CPU 提升

5 倍。如需要执行 n 条指令，执行时间为(n+4)*T，当然这是理想情况，实际指令流水线存在很多的冲突冒险，本实验实现的是无冲突冒险的理想流水线，所以暂时可以不考虑这些冲突冒险。

6.1.3 实验内容

将课程实验完成的 MIPS 单周期 CPU 改造成支持无冲突冒险程序运行的理想流水线架构，标准测试程序如下，该程序在数据存储器 0、4、8、12（字节地址）位置依次写入数据 0、1、2、3，程序无任何数据冲突和分支冒险，所以设计流水线时无需考虑任何冲突冒险的处理。

```
//file: 理想流水线测试.asm
#理想流水线测试，所有指令均无相关性，一共17条指令，
#5段流水线执行周期数应该是5+(17-1）=21
  addi $s0,$zero, 0
  addi $s1,$zero, 0
  addi $s2,$zero, 0
  addi $s3,$zero, 0
  ori $s0,$s0, 0
  ori $s1,$s1, 1
  ori $s2,$s2, 2
  ori $s3,$s3, 3
  sw $s0, 0($s0)
  sw $s1, 4($s0)
  sw $s2, 8($s0)
  sw $s3, 12($s0)
  addi $v0,$zero,10          # system call for exit
  addi $s1,$zero, 0          #三条无用指令消除syscall与v0寄存器的相关性
  addi $s2,$zero, 0
  addi $s3,$zero, 0
  syscall                    # 停机
```

单周期 MIPS CPU 在一个时钟周期内完成取指令、指令译码、执行、访存、写回等一系列动作，如图 6.5 所示。各段直接串行连接，时钟周期等于各段的总延迟，性能较差。要将单周期架构改造成流水线架构，首先需要将 MIPS 指令执行过程严格分成 5 个阶段：取指令 IF 段、译码取数 ID 段、指令执行 EX 段、访存 MEM 段、写回 WB 段(注意不得简化成 4 段流水线)。其中 IF 段包括程序计数器 PC、NPC 下址逻辑、指令存储器等功能模块；ID 段包括控制器逻辑、取操作数逻辑；EX 段主要包括运算器模块；MEM 段主要包括内存读写模块；写回 WB 段主要包括寄存器写入控制模块。需要注意的是不是所有指令都需要经历完整的 5 个阶段。

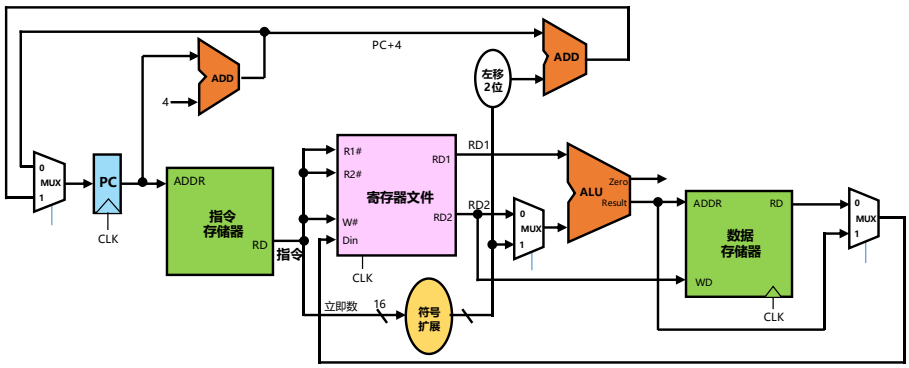

图 6.5 单周期 MIPS CPU 数据通路

不同阶段之间增加流水接口部件（锁存器），如图 6.6 所示，在单周期 CPU 实现基础上需要增加 IF/ID、ID/EX、EX/MEM、MEM/WB 共四个流水接口部件，四个流水接口均采用公共时钟进行同步，流水接口定义尽可能简化，其内部主要是若干寄存器，用于锁存段间数据，五段流水结构在第 1 章 数据表示实验中已经出现过，具体实现时可以参考数据表示实验中海明编码流水传输电路。

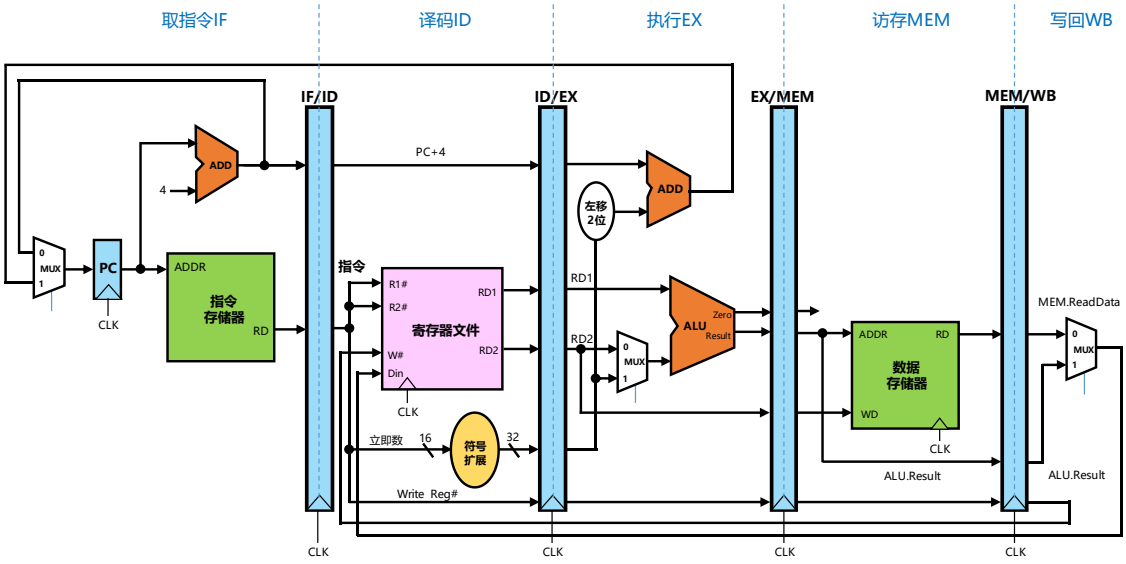

图 6.6 五段流水 MIPS CPU 数据通路

流水线通过流水接口部件为后段提供数据信息，控制信息，如图 6.7 所示，向前段传递反馈信息，流水线后段对数据的加工处理依赖于前段通过流水接口部件传递过来的信息。ID 段译码生成该指令的所有控制信号（图中蓝色线为控制信号），控制信号通过锁存器逐段向后传递，后段功能部件所需的控制信号不需要单独生成，直接从锁存器获取。注意单周期 CPU 中的控制器可以在 ID 段直接复用。不同的流水接口部件锁存的数据和控制信号不同，具体可根据前后段之间的交互信息进行考虑，以最为复杂的 ID/EX 接口部件为例，该锁存器锁存 ID 段由控制器产生的所有控制信号，同时还需要锁存由取操作数部件取出的寄存器值或立即数，ID/EX 部件设计完成后，其它各段流水接口部件可以直接复制后进行适当精简。

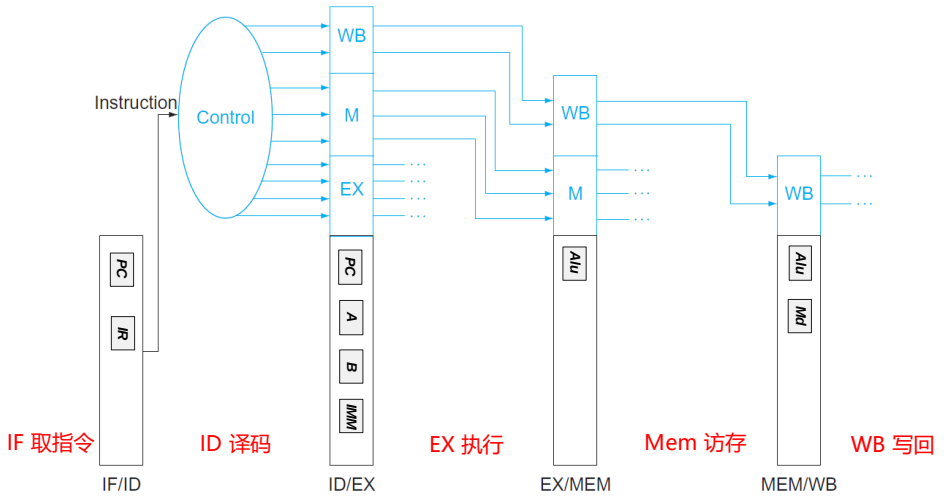

图 6.7 五段流水 MIPS CPU 数据与控制信号传递

五段流水 CPU 输入输出引脚如图 6.8 所示，请按相关要求进行流水线 CPU 设计，具体设计时请遵循如下注意事项。

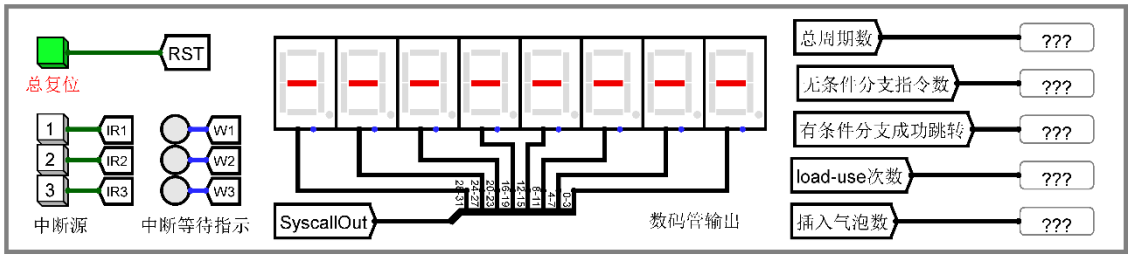

图 6.8 流水 CPU 输入输出引脚示意图

注意事项：

(1) 在 Logisim 平台实现时应尽量缩小原单周期 CPU 原理图中的功能部件尺寸，以方便布局绘图。

(2) 流水接口部件封装尺寸尽可能长一点，否则连线过多会给布线带来困扰。通过流水接口部件向后传输的控制信号应遵循越晚用到越靠近顶端的原则，便于腾出更多的空间进行流水功能段的电路布局；适当使用颜色标记关键功能部件和流水接口部件。

(3) 控制信号可以按使用段分为 EX 段使用、MEM 段使用、WB 段使用三类，ID 段可以将这三类信号利用分线器合并成三根多位宽总线，这样可以大大减少控制信号线的根数，绘图更简单，布局更清晰，需要使用的时候再用分线器拆开即可。

(4) 指令存储器 ROM 和数据存储器 RAM 必须在 main 电路中可见，不能封装在子电路中，以便于调试观察。

(5) 主要数据通路应直接连接，横向可连接的线缆应尽量直接连接，避免隧道的滥用，以保证原理图的可读性，连线一般不允许穿越其它功能部件的封装。

(6) 尽可能使用标签工具注释电路，包括控制信号、数据通路、显示模块、总线等，以便

于电路调试。注意标签以及注释的命名规范，过长的命名会影响电路布局。

(7) 可以使用任何 logisim 内建的组件构建电路，原运算器模块中使用了自定义 32 位加法器的模块，涉及多个子电路的嵌套，实际使用时系统最大频率很低，建议用 Logisim 内置的加法器进行替换。

(8) **严禁对时钟信号进行逻辑门操作，**停机操作也可以通过相关功能部件使能端完成。

6.1.4 实验步骤

(1) **设计流水接口部件**，首先考虑四个流水接口部件需要锁存的具体信息，然后设计最为复杂的 ID/EX 流水接口部件，封装设计完成后再复制生成 IF/ID、EX/MEM、MEM/WB 流水接口部件。注意流水接口部件封装的长度、标签注释以及颜色区分。

(2) **重新封装较大尺寸的功能部件**，如运算器、寄存器文件等。

(3) **重构数据通路**，将单周期 CPU 中的数据通路全部删除，重新进行电路布局，将各段功能部件与流水接口部件进行连接。

(4) **系统联调，功能测试**，加载标准测试程序进行功能调试，注意最终理想流水线测试程序会在数据存储器中写入数据，请自行检查后再提交检查。

6.2 气泡流水线 CPU 设计实验

6.2.1 实验目的

学生理解数据相关的基本原理，掌握插入气泡方式消除指令数据冲突的方法，能够在理想流水线的基础上增加逻辑电路进行数据冲突检测，可通过硬件插入气泡的方式消除数据冲突；学生理解控制冲突的基本原理，理解其引起的流水线停顿导致的性能下降，掌握控制冲突流水线处理机制，并能够增加相关逻辑使得流水线能正确处理分支指令引起的控制冲突；学生理解结构相关的基本原理，能分析五段流水 CPU 中存在的结构冲突，并能运用适当的方案进行解决，最终实现的五段流水 CPU 能正确运行单周期 CPU 实验中测试过的 benchmark 标准测试程序。

6.2.2 背景知识

理想流水线所有待加工对象均需要通过相同的部件（阶段），不同阶段之间无共享资源，且各段传输延迟一致，进入流水线的对象也不应受其它功能段的影响，但这仅仅适合工业生产流水线，计算机指令流水线存在较多的指令相关，会引起流水线的冲突和停顿。

所谓指令相关，是指指令流水线中，如果某指令的某个阶段必须等到它前面的某条指令的某个阶段完成才能开始，也即是两条指令间存在着某种依赖关系，则两条指令存在指令相关。指令相关包括数据相关、结构相关、控制相关，指令相关会导致流水线冲突/冒险（Hazzard）。流水线冲突是指由于指令相关的存在，导致指令流水线出现“断流”或“阻塞”，下一条指令不能

在预期的时钟周期加载到流水线中。流水线冲突包括数据冲突、结构冲突、控制冲突。

1) 数据冲突

当前指令要用到前面指令的操作结果，而这个结果尚未产生或尚未送达指定的位置，会导致当前指令无法继续执行，称为数据冲突。根据指令读访问和写访问的顺序，常见的数据冲突包括先写后读冲突（Read after Write，RAW）、先读后写冲突（Write after Read，WAR）、写后写冲突（Write after Write，WAW）。假定连续的两条指令 I_1 和 I_2，其中指令 I_1 在指令 I_2 之前载入流水线，两条指令之间可能引起的数据冲突如下：

(1)先写后读冲突 RAW

如果指令 I_2 的源操作数是指令 I_1 的结果操作数，这种数据冲突被称之为先写后读冲突（RAW）。当指令按照流水的方式执行的时候，由于指令 I_2 要用到指令 I_1 的结果，如果指令 I_2 在指令 I_1 将结果写入寄存器之前就在 ID 译码取数阶段读取了该寄存器的旧值，则会导致读取数据出错。

(2)先读后写冲突 WAR

如果指令 I_2 的结果操作数是指令 I_1 的源操作数，这种数据冲突被称之为先读后写冲突（WAR）。当指令 I_2 去写该寄存器的时候，指令 I_1 已经读取过该寄存器，所以这种数据相关对指令的执行不构成任何影响。

(3)写后写冲突 WAW

如果指令 I_2 和指令 I_1 的结果操作数是相同的，这种数据冲突被称之为写后写冲突（WAW）。当指令按照流水的方式执行的时候，如果指令 I_2 的写操作发生在指令 I_1 的写操作之后，这种 WAW 冲突对指令的执行没有影响，但在乱序调度的流水线中，有可能指令 I_2 的写操作发生在 I_1 指令的写操作之前，此时会发生写入顺序错误，结果单元中留下的是指令 I_1 的执行结果，而不是指令 I_2 的执行结果。由于本实验并未采用乱序发射技术，所以 WAW 冲突在本实验中不予考虑。

正常的程序都会存在着较多的 RAW 数据冲突，为了有效的避免结果出错，最简单的处理方法就是推后执行与其相关的指令，以保证指令和程序执行的正确性。如果利用软件方法解决就是在存在数据冲突的指令之间插入空指令，这需要编译器支持；如果采用硬件方法解决就是所谓插入“气泡”的方法，ID 段的指令如果与 EX、MEM 或 WB 段的指令存在数据冲突，则 ID 段正在执行的指令暂停一个时钟周期（通过阻塞 IF/ID，PC 实现），时钟到来时将 ID/EX 流水接口部件同步清零，也就是在 EX 段插入一条空指令（气泡），如果此时 ID 段仍然 MEM、WB 段数据相关，则继续阻塞 IF/ID 接口部件，下一个时钟到来时继续在 EX 段插入一个气泡，依次类推直至数据冲突消失。

2) 结构冲突

由于多条指令在同一时钟周期都需使用同一操作部件而引起的冲突称为结构冲突。假如流水线只有一个存储器，数据和指令都存放在同一个存储器中，当 Load 指令进入 MEM 段时，IF 段也同时需要访问存储器取出新指令，这时就会发生访存结构冲突。这样的结构冲突实际上在单周期 CPU 中就存在，解决方法是采用独立的指令存储器和数据存储器（哈佛结构），现代 CPU 中指令 cache 和数据 cache 分离也是这种结构，流水线中也可以采用同样的解决方案。

另外一个方案是在流水线中插入一个“气泡”，阻塞 PC 寄存器，使得 IF 段暂停一个时钟周期，下一个时钟周期到来时同步清空 IF/ID 接口部件，进入 ID 段的是一个气泡操作（空指令），等到 load 指令访存操作结束以后，IF 段再次重新启动，相比哈佛结构这种方案会使得流水线暂停一个时钟周期，引起性能的损失。

3) 控制冲突

当流水线遇到分支指令或其它会改变 PC 值的指令时，分支指令后续已经载入流水线的相邻指令可能不能进入执行阶段，这种冲突称为控制冲突，也称为分支冲突，由于分支指令跳转是否成功或改变后的 PC 值要等到 EX 或 MEM 段才能确定或计算出来（具体哪个阶段与设计相关），所以分支指令相邻后续若干指令已经预取进入流水线（后续预取指令条数称为预取深度），当分支指令成功跳转时，流水线预取的指令不能进入执行阶段，此时需要清空预取指令，同时修改 PC 的值，取出正确的分支目标地址的指令。发生控制冲突时，流水线会清空预取指令，浪费了若干时钟周期，引起流水线性能降低。

还可以采用基于软件或硬件的方法提前进行分支预测，在 IF 阶段就预取分支目标地址处的正确指令，以尽量减少由于控制相关而导致流水线性能下降。另外在 MIPS 中还采用了分支延迟基础上，也就是分支指令后的一条指令无论分支是否成功，都会进入执行阶段，这种方式也可以降低预取深度，本实验为了引入后续的动态分支预测技术，所以没有采用延迟分支技术。

6.2.3 实验内容

进一步改造理想流水线 MIPS CPU，增加数据相关检测逻辑，增加插入气泡逻辑，增加流水暂停逻辑，增加分支冲突处理逻辑，使得该流水线能处理数据冲突、控制冲突、结构冲突，并能正确运行单周期测试程序 benchmark.hex。

（1）**数据相关检测**，由于 ID 段需要取操作数，所以数据相关检测逻辑应设置在 ID 段。图 6.9 所示的五段流水结构中，数据只能在 WB 段写入，所以 ID 段正在处理的 ADD 指令与 EX、MEM、WB 段的指令都存在数据相关。本实验需要实验者设计数据相关检测逻辑，当存在数据相关时，可进行插入气泡的处理。数据相关的依据是比较当前指令是否存在源操作数和后续段的目的操作数是否相同。注意在 MIPS 指令集中不同指令所包含的源操作数是不同的，如 R 型指令涉及两个源操作数 Rs、Rt；I 型指令涉及一个或两个源操作数 Rs、Rt；分支指令（Beq，Bne）涉及两个源操作数 Rs、Rt；J 型指令无源操作数，但会产生控制冲突。注意比较时可能需要和后续三段的目的操作数均做比较，且需要考虑后续段的三条指令是否存在目的操作数，另外需要注意的是零号寄存器写入无意义。

（2）**数据冲突处理**，图 6.9 中 ID 段与 WB 段的数据相关可以通过寄存器先写后读的方式解决，具体方式是寄存器文件写入数据时采用下跳沿触发，由于寄存器的读出是组合逻辑，数据写入后立刻可获得寄存器新值，所以在下跳沿成功写入后，上跳沿触发流水线进行指令传送时，送入 ID/EX 的操作数已经是最新的寄存器值。而 ID 段与 EX 段、MEM 段的数据冲突则只能通过插入气泡的方式解决。

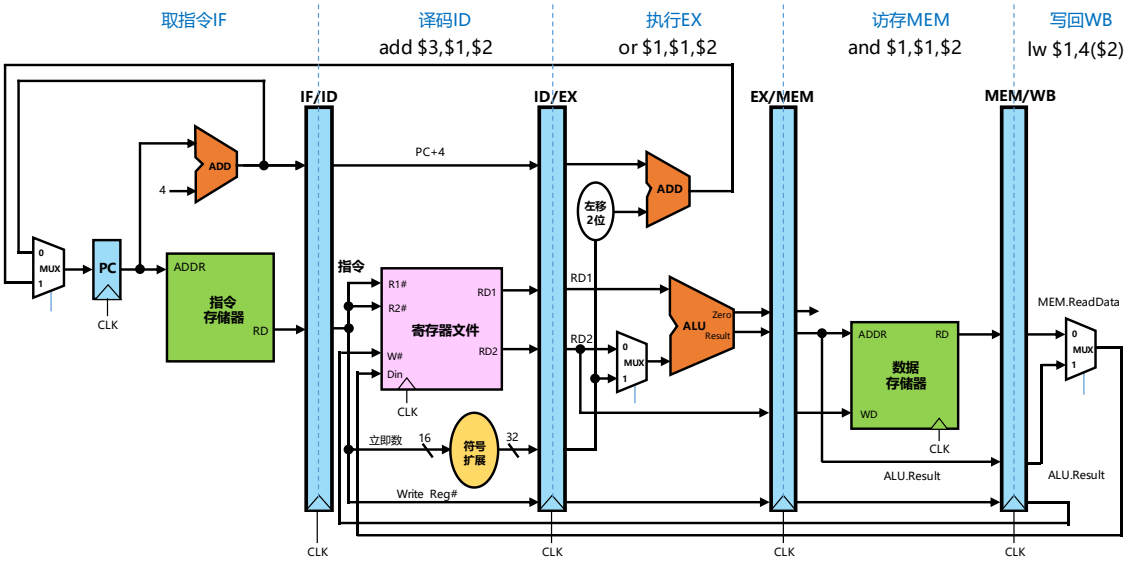

图 6.9 五段流水 MIPS CPU 中的数据相关

（3）**插入气泡逻辑**，出现数据相关时将在 EX 段插入气泡，IF、ID 段暂停。图 6.10 中 ID 段与 EX 段存在数据相关，假设相关检测逻辑已经检测到数据相关，应产生 stall（暂停）控制信号，阻塞 PC 以及 IF/ID 部件，暂停 IF、ID 段的工作以延缓取操作数的执行（可以通过控制程序计数器 PC 和 IF/ID 流水接口部件的使能端实现，这样时钟到来时寄存器的值不会改变），另外下一个时钟上跳沿到来时，需要向 EX 段插入一个气泡（空指令，MIPS 中空指令是全零的机器码，功能是 0 号寄存器左移零位送入 0 号寄存器），以避免 ID 段的指令向后传送。插入气泡可以通过 ID/EX 流水接口部件同步清零完成，为此需要为流水接口部件增加清零引脚。

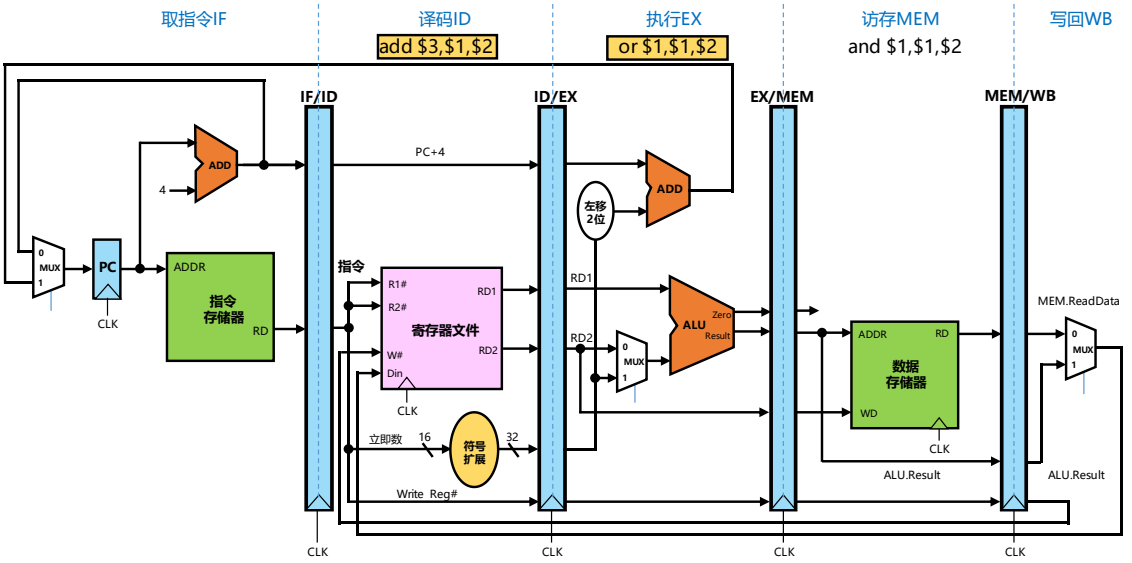

图 6.10 五段流水 MIPS CPU 中的数据相关---插入气泡前

时钟上跳沿到来后，将在 EX 段插入一个气泡，如图 6.11 所示，但此时数据相关检测逻辑

还可以检测到 ID 段和 MEM 段存在数据相关，此时继续重复前面的逻辑，在 EX 段继续插入气泡，暂停 IF 和 ID 段。

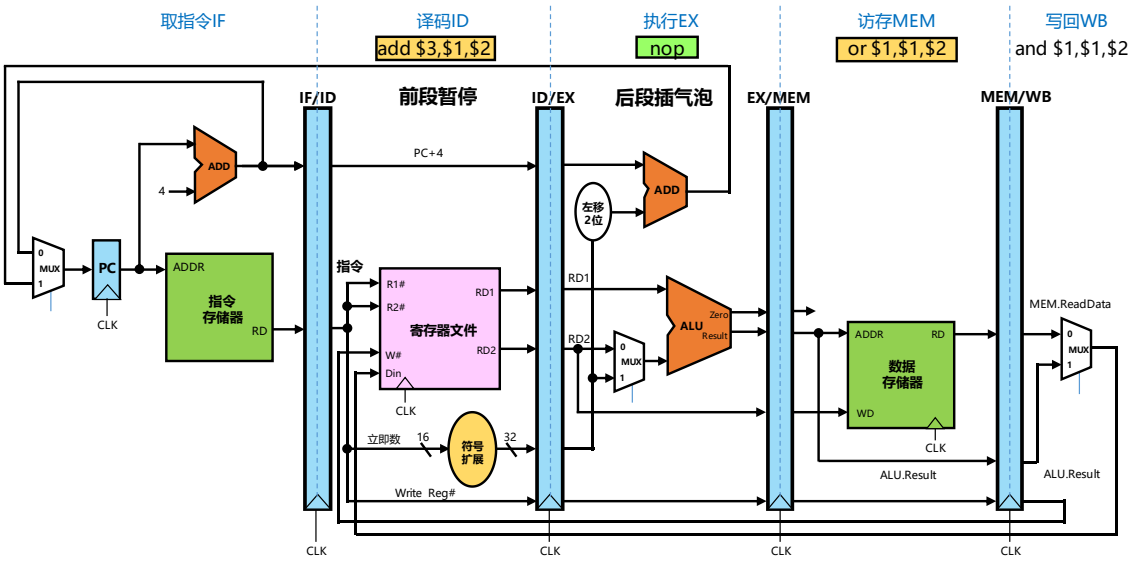

图 6.11 五段流水 MIPS CPU 中的数据相关---插入气泡后

下一个时钟上跳沿到来后，将在 EX 段继续插入一个气泡，如图 6.12 所示，此时数据相关消失，流水线暂停信号 stall 自动撤除，流水线重新恢复正常。

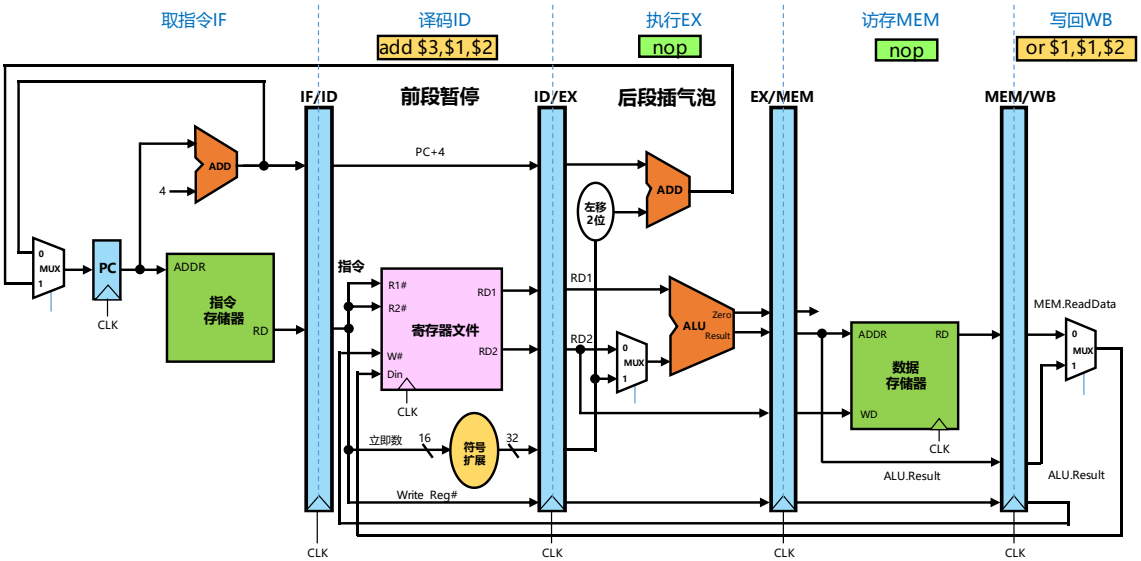

图 6.12 五段流水 MIPS CPU 中的数据相关---数据相关消除

下一个时钟上跳沿到来时，ID 段处理新的指令，存在数据相关的 ADD 指令通过 ID/EX 流水接口部件进入 EX 段，如图 6.13 所示。

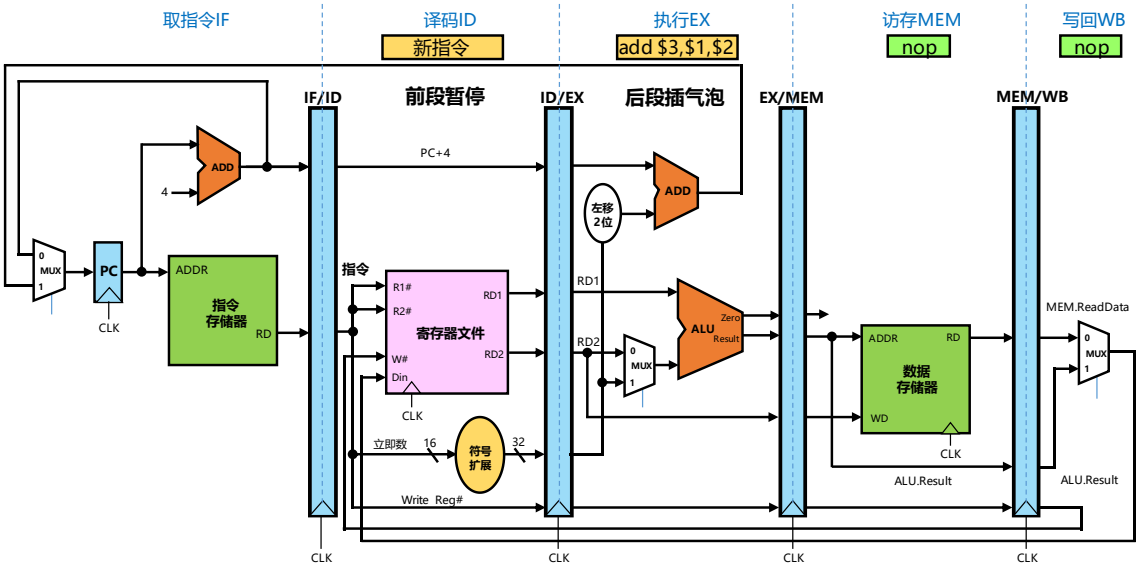

图 6.13 数据相关消除后的执行状态

（4）**结构冲突处理**，当多条指令在同一时钟周期都需使用同一操作部件而引起的冲突称为结构冲突，在流水线设计中也会存在各种结构冲突，如计算 PC+4、计算分支地址、运算器运算都需要使用运算器，访问指令和访问数据都需要使用存储器，解决方案是增设加法部件避免运算冲突，增设指令存储器避免访存冲突。另外 ID 段读寄存器与写回阶段写寄存器也存在结构冲突，可以采用先写后读的方式解决（下跳沿写入，与流水接口时钟触发相反）。

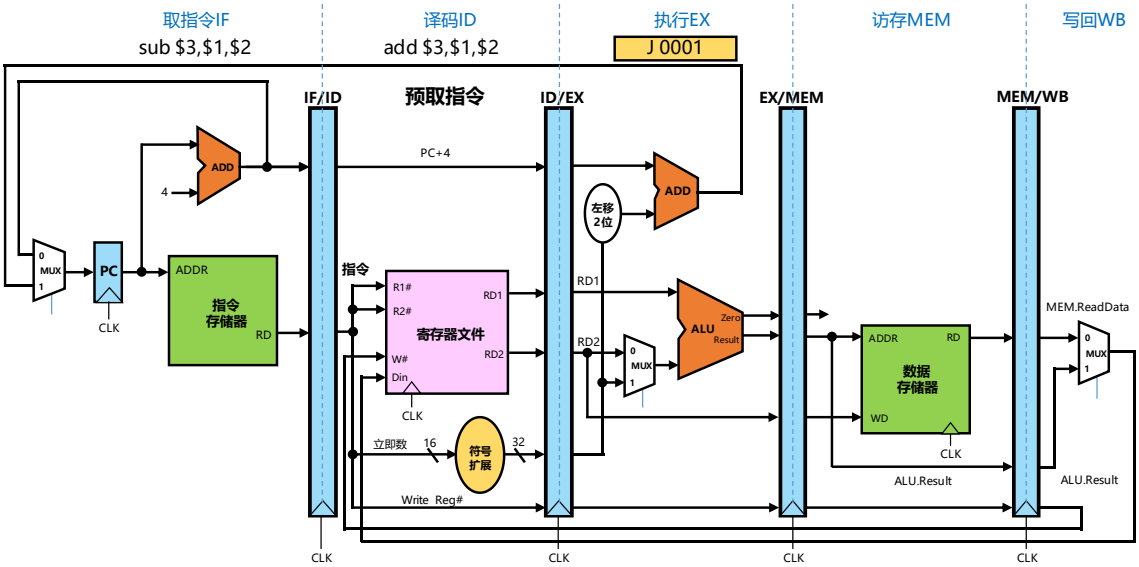

图 6.14 五段流水 MIPS CPU 中的分支相关

（5）**控制冲突处理**，如图 6.14 所示，在执行阶段出现了一条无条件分支指令 J 指令，这里假设无条件分支指令在 EX 段执行，J 指令执行时需要修改程序计数器 PC 的值，运算器运

算的分支目标地址的结果经多路选择器传送给 PC，下一个时钟上跳沿到来时锁存进入 PC，IF 段的 SUB 指令以及 ID 段的 ADD 指令都属于预取进入流水线的指令，都无法继续在流水线中执行，需要清除。

图 6.15 是时钟上跳沿到来后流水线的状态，从图可看出，IF/ID、ID/EX 接口部件所有数据和控制信号全部被清零，全零的 MIPS 指令是 sll $0,$0,0 指令，不会改变 CPU 状态，等同与空操作 nop，预取指令成功清除，所以这里发生控制冲突时应该给出 IF/ID、ID/EX 段的清空信号，时钟上跳沿到来时完成预取指令清空。

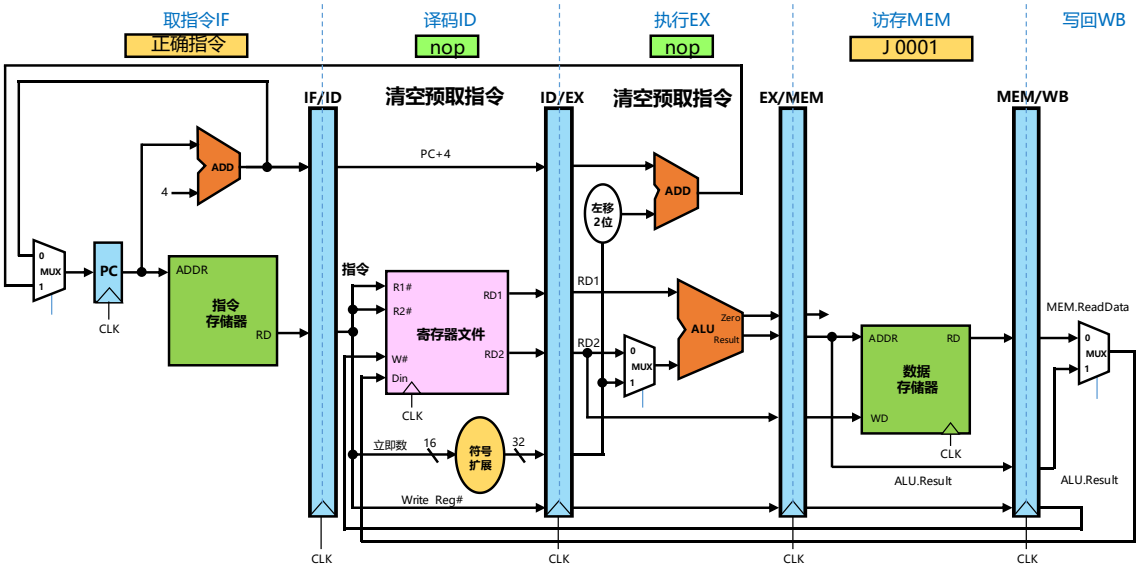

图 6.15 五段流水 MIPS CPU 中的分支相关正确处理

注意：真实 MIPS 流水线中采用了分支延迟槽技术，也就是分支指令后面的一条指令无论跳转成功与否均会被执行，为简化实验，我们取消分支延迟槽技术，所以 MIPS 指令规范中所有 PC+8 的要求在实验中应全部改成 PC+4。

另外分支指令的执行也可安排在 ID 段，MEM 段，甚至 WB 段完成，不同段执行预取深度不一样，在执行 EX 段进行分支处理的预取深度为 2，预取深度越大，对流水性能造成的影响也越大，实验时可以酌情考虑在哪个阶段完成分支处理，有条件分支指令由于涉及计算，建议放在 EX 段完成，这也有利于后续数据重定向机制的统一实现。

图 6.16 为 MIPS 五段流水 CPU 处理数据相关和分支相关的流水线时空转置图，此图和普通的时空图略有不同，横坐标是空间，代表各功能部件段，纵坐标是时间，每一格代表一个时钟周期，图中单元格的数字代表流水线功能部件的指令序号，采用这种方法展示时空图更有利于大家结合具体电路设计进行分析验证。

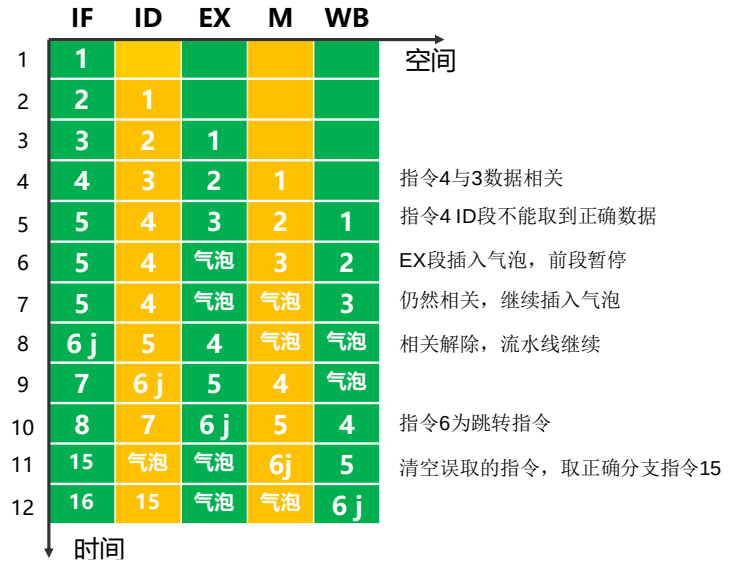

图 6.16 存在数据相关和分支相关的流水线时空转置图

6.2.4 实验步骤

(1) **设计相关检测逻辑**，分析实验中需要支持的所有 MIPS 指令的指令格式，根据需要设计数据相关检测逻辑，用于判断 ID 段的指令与 EX 以及 MEM 段指令之间的数据相关性，相关检测逻辑封装完成后应利用测试用例中的相关测试程序进行测试，检查几个关键点的相关性能否检查出来，注意该测试程序只用到了很少的指令，所以只能简单的验证，并不能完全测试相关性检测逻辑的正确性。

```
//file: 相关测试程序.asm
 #用于数据相关性测试，程序最终完成等差数列求和运算
 #计算0+1+2+3+4+5+6+7，运算中间结果存入内存4,8,c,10,14,1c
 #程序一共包含38条指令
 .text
 addi $s1, $0, 4
 sw $s1, 0($s1)
 lw $s2, 0($s1)
 addi $s2,$s2,-4          #s2 load-use相关
 addi $s0,$0,0            #数列初值
 addi $s1,$s0,1           #计算下一个数，等差值为1     $s0和上条指令相关
 add $s0,$s0,$s1          #求累加和    $s0和上上条指令相关，$s1和上条相关
 add $s2,$s2,4            #地址累加
 sw $s0,0($s2)            #存累加和    $s2和上条指令相关
```

```
addi $s1,$s1,1
add $s0,$s0,$s1          #求累加和
add $s2,$s2,4            #地址累加
sw $s0,0($s2)            #存累加和
…
addi $v0,$zero,10        # system call for exit
addi $s0,$zero, 0        #消除相关性
addi $s0,$zero, 0        #syscall 隐含操作数V0也可能引起相关，需要注意
addi $s0,$zero, 0
syscall                  # we are out of here.
```

(2) **改造流水接口部件，**使其能实现流水暂停以及同步清零插入气泡的功能。

(3) **实现插入气泡逻辑，**在 ID 段的控制器中实现插入气泡逻辑，并使用相关测试程序进行测试，此时要求功能正确。

(4) **实现控制相关逻辑，**增加相关逻辑，使得无条件分支指令，有条件分支指令能在流水线上正确运行，并分别使用实验资料包中的 JMP 测试程序以及 B 指令测试程序进行功能测试。

(5) **流水综合调试，**加载标准测试程序进行功能调试，注意统计时钟周期数，无条件分支数，有条件分支成功分支次数，插入气泡数等参数，以便进行正确性的判断。

(6) **流水性能分析，**根据参考答案，结合自己的实际方案，分析流水线执行总周期数与单周期执行程序周期数之间的关系，一般情况下应符合如下公式：**气泡流水周期数=单周期执行周期数 ＋（流水冲满时间 -1） ＋ J 指令*预取深度 ＋ 条件分支成功次数*预取深度 ＋ 气泡数。** 注意由于相关检测逻辑实现的误差，可能少插或多插气泡，导致气泡数不一致，请准确核对自己的相关检测逻辑。

(7) **提交检查。**

6.2.5 实验思考

(1) 插入气泡能否直接使用寄存器的异步清零信号，为什么？

(2) 插入气泡的个数是否需要用电路进行控制？为什么？

(3) MIPS 零号寄存器是比较特殊的寄存器，如果源操作数是零号寄存器时是否存在数据相关？你是如何解决的？气泡指令之间是否存在相关？

(4) 采用气泡方式解决数据相关后测试标准测试程序 benchmark 时钟周期为什么反而比单周期 CPU 多了很多，难道流水线还不如单周期 CPU 吗？

6.3 重定向流水线 CPU 设计实验

6.3.1 实验目的

学生理解数据重定向的基本原理，理解 Load-use 相关的处理机制，能够在气泡流水线 CPU 的基础上增加相应的逻辑功能部件，使得除了 Load-use 相关的所有数据冲突都可以采用重定向的方式进行冲突处理，最终实现的 CPU 能正确运行单周期 CPU 实验中测试过的 benchmark 标准测试程序。

6.3.2 背景知识

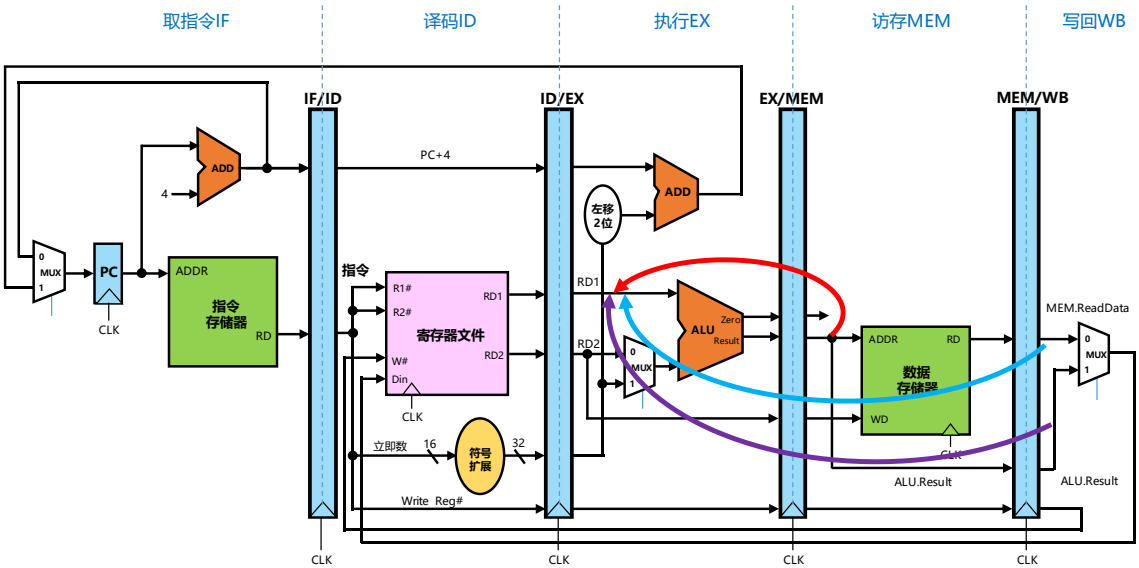

图 6.17 数据重定向示意图

气泡流水线通过延缓 ID 段取操作数动作的方式解决数据冲突问题，但大量气泡的插入会严重影响指令流水性的性能，还有一种思路是先不考虑 ID 段所取的操作数是否正确，而是等到实际需要使用这些操作数时再考虑正确性问题，如图 6.17 所示，EX 段的指令可能与 MEM 段，WB 段的两条指令均存在数据相关，此时 EX 段取得的操作数应该是错误的数据，正确的数据分别存放在 EX/MEM 以及 MEM/WB 流水接口部件中，还未写回到寄存器中，此时可以直接将正确数据从其所在位置重定向（Forwarding）到 EX 段合适的位置（也称为旁路 Bypass），这样就可以避免插入气泡引起的流水线性能下降，重定向方式可以解决大部分的数据相关问题，可大大优化流水线性能。

以 ID/EX.RD1 为例，此输出会送到 ALU 的第一个操作数端，但可能不是最新的值，所以应在 ID/EX. RD1 的输出端增加一个多路选择器，此多路选择器的默认输入来源为 ID/EX.RD1、另外也可能来自 EX/MEM.AluResult 的重定向，也可能是来自 MEM/WB.AluResult 或

MEM/WB.ReadData 的重定向，同样 ID/EX. RD2 输出端也可以再增加一个多路选择器进行同样的重定向处理，当然新加入的多路选择器会增加一个 forward 的选择控制信号，这个信号应可以在译码段进行数据相关检测时自动生成，如图 6.18 所示。

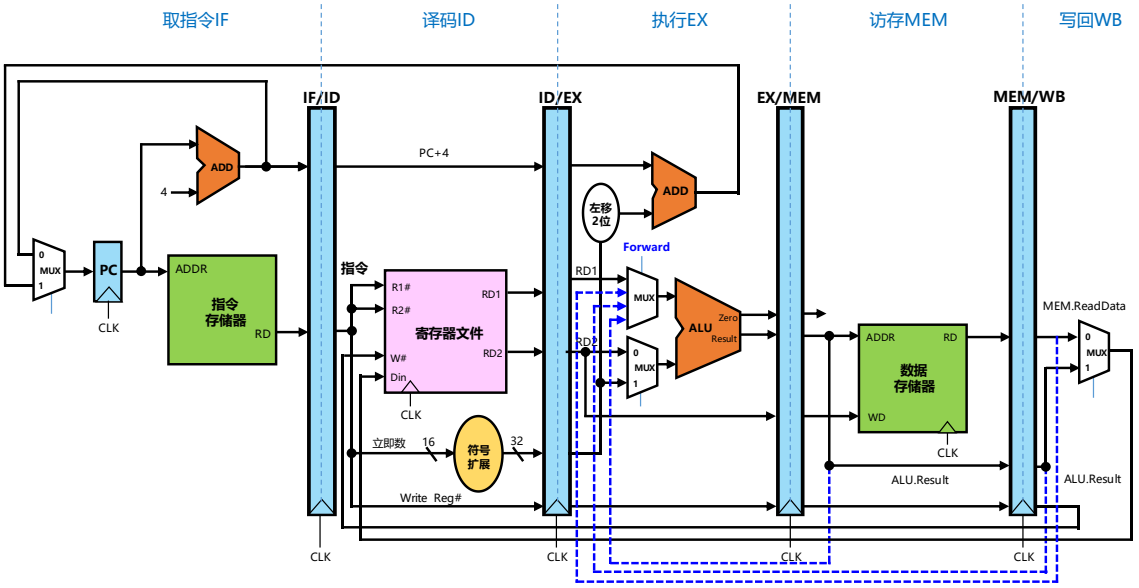

图 6.18 数据重定向数据通路

但是如果出现相邻两条指令存在数据相关，且前一条指令是访存指令时（称为 Load-Use 相关），不能采用重定向方式进行处理，如图 6.19 所示。

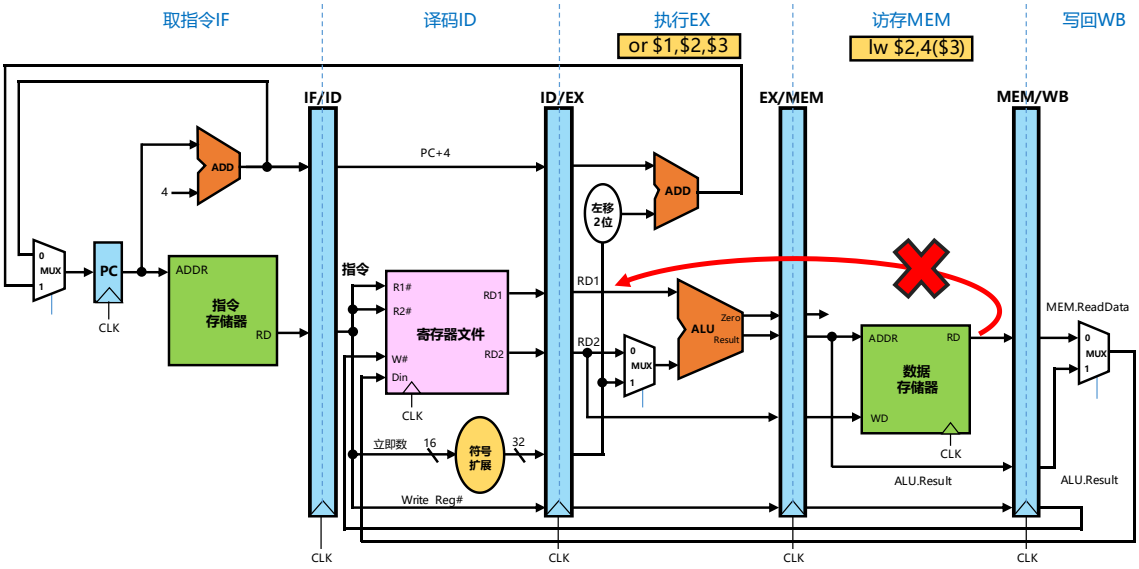

图 6.19 Load-Use 相关

EX 段 or 指令需要使用 1 号寄存器，而 MEM 段目的寄存器也是 1 号寄存器，这时 1 号寄存器的值并没有锁存在 EX/MEM 中，而是必须等待数据存储器读操作完成后才会出现在 Read data 引脚上，如果将该引脚直接重定向到 EX 段，逻辑上也可以成立，甚至在 Logisim 以及

FPGA 开发板上也可以成功实现，但实际上这样会使得 EX 段的延迟变成了访存延迟+运算器延迟，违背了流水线分段尽量使各段延迟相等的原则，会使得系统最大时钟频率降低一倍。所以对于 Load-Use 数据相关，应在 ID 段就及时检出，并强制插入一个气泡，消除这种相关，如图 6.20 所示。

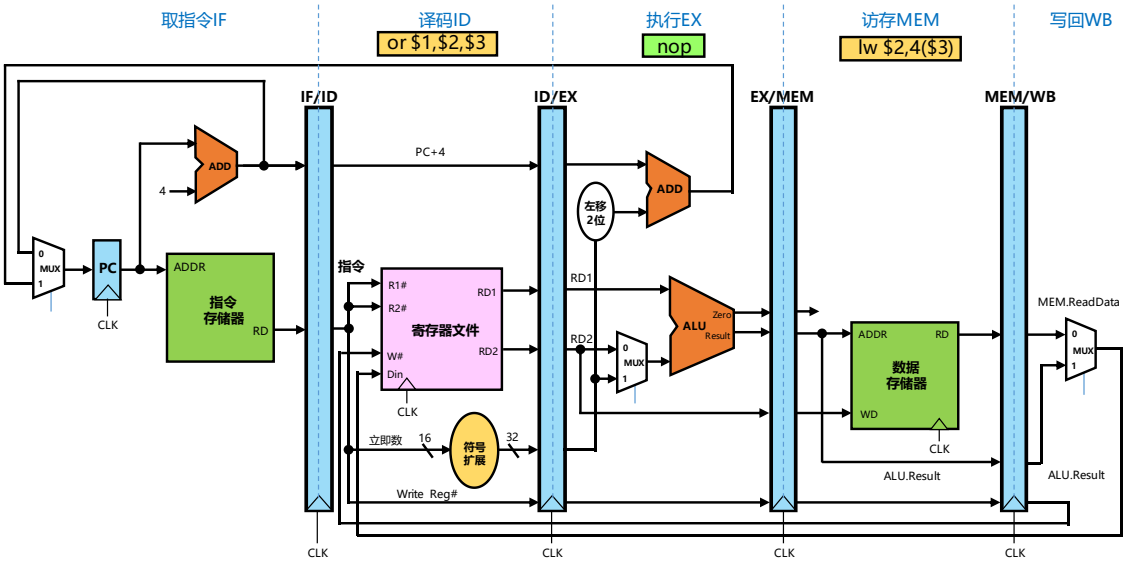

图 6.20 插入气泡消除 Load-Use 相关

6.3.3 实验内容

进一步改造气泡式流水线 MIPS CPU，增加重定向机制，增加 Load-Use 数据相关检测机制，使得流水线能在不插入气泡的情况下处理大部分数据相关问题，最终能正确运行单周期测试程序 benchmark.hex。

6.3.4 实验步骤

（1）**增加重定向数据通路**，如 EX/MEM 中锁存的运算器运算结果、MEM/WB 中锁存的运算器运算结果、访存数据等，重定向到 EX 段，在合适的位置增加多路选择器，增加选择控制信号。

（2）**构造重定向逻辑**，首先改造数据相关检测逻辑，增加 Load-Use 数据相关检测逻辑，如出现 Load-use 相关执行原有插入气泡的逻辑进行处理。如出现非 Load-Use 数据相关，则由重定向逻辑直接生成操作数来源选择信号，对进入运算器或存储器的操作数正确来源进行选择。这里重定向逻辑可以放置在 ID 段，也可以放置在 EX 段，二者各有利弊，大家可自行权衡。

（3）流水综合调试，首先加载数据相关测试程序进行简单数据相关功能验证测试，测试验证无误后再加载标准测试程序进行功能调试，注意统计时钟周期数，Load-Use 冲突数等运行

参数。

（4）流水性能分析。根据参考答案，结合自己的实际方案，分析流水线执行总周期数与单周期执行程序周期数之间的关系，一般情况下应符合如下公式：**重定向周期数= 指令条数 +（流水冲满时间 -1） + 分支冲突次数*预取深度 +LOAD-USE 数**。由于程序中 Load-Use 部分是固定的，所以重定向周期数的答案应该是标准的，如果出现偏差，请分析原因。

（5）提交检查。

6.3.5 实验思考

(1) 重定向逻辑放在流水线哪个阶段更好，为什么？

6.4 基于 FPGA 的流水 CPU 实验

6.4.1 实验目的

学生掌握 N4-DDR FPGA 开发板的使用，能将课程实验中设计完成的流水 CPU 在该平台上具体实现，并能正确运行标准测试程序。

6.4.2 实验内容

将支持重定向机制的五段流水 MIPS CPU 从 Logisim 移植到 N4-DDR FPGA 平台，采用硬件描述语言重新设计所有功能部件，并进行数据通路重构，在 FPGA 平台上实现五段流水 CPU。

实验要求：

(1) **支持显示功能切换**，能在 FPGA 开发板上利用 8 个八段数码管进行正确的显示输出，可以通过拨码开关实现显示功能的切换，应包括正常数据输出显示、不同内存地址内存值的观察、PC 值观察、时钟周期数观察、其它运行参数统计观察等（需要设置显示功能切换拨码开关---用于切换显示功能，另外还需要设置内存地址输入拨码开关）。

(2) **支持频率切换**，最高频率以及演示频率（保证输出显示效果与 Logisim 一致，肉眼可观察），需要设置频率切换拨码开关。

(3) **支持总复位按钮**，按下总复位按钮后，系统复位，所有寄存器，存储器值清零，从 0 号地址开始重新执行程序。

6.5 动态分支预测机制设计实验

6.5.1 实验目的

学生研究动态分支预测相关原理，掌握相联存储器设计机理，并最终应用相关机制为已经实现的五段流水 CPU 添加动态分支预测机制，最终方案应比原有方案性能更优。

6.5.2 背景知识

采用重定向机制后，指令流水线中数据相关基本不需要插入气泡就可解决，只有少数 Load-Use 冲突需要插入一个气泡解决冲突问题，指令流水线性能得到较好的提升。此时指令流水线中结构冲突（分支冒险）对流水线性能影响很大，分支指令会使指令流水线暂停，因为分支使得 IF 段以及 ID 段执行的指令（预取指令）被清空，这部分流水线性能损失称为分支延迟。指令预取深度越长，分支延迟越大。为减少分支延迟，应在流水线中尽早判断出分支是否成功跳转，并尽早计算出分支目标地址，比如将分支指令放在 ID 段完成，这样预取深度就是 1，分支延迟也只有一个时钟周期。进一步降低分支延迟的主要方法有**静态分支处理**与**动态分支预测**两种。

静态分支预测主要是基于编译器的编译信息对分支指令进行预测，预测信息是静态的不能改变，与分支的实际执行情况无关，通常是采用一些简单的策略进行预测或处理，具体如下：

1) 预测分支失败，这是缺省逻辑，与无分支预测的指令流水方案相同。
2) 预测分支成功，这个逻辑效果和第一种应该差不了太多。
3) 延迟分支，由编译器将一条有用的指令放在分支指令后，作为分支指令的延迟槽 delay slot，不论分支指令是否跳转，都要按顺序执行延迟槽指令，如果分支指令放在 ID 段执行，延迟槽技术可以完全消除分支延迟，但这需要编译器进行指令调度，取决于编译器能否将有效指令放入延迟槽，对编译器有较高的要要求。在本实验中没有考虑延迟槽的概念。

动态分支预测依据分支指令的分支跳转历史，不断的对预测策略进行更新，具有较高的预测准确率，存放分支跳转历史统计信息的逻辑必须用硬件实现，现代处理器中均支持动态分支预测算法。最简单的动态分支预测策略是分支预测分支历史表 (branch history table，BHT)，也称为分支预测缓冲器（Branch Prediction Buff），BHT 表中存放分支指令的地址，分支目标地址，历史跳转信息描述位（分支预测历史位），如表 6.1 所示，其中 valid 位表示对应表项是否有效，如果不设置 valid 位，无法区分哪些表项是有效数据，比如初始化时表格全零，也会被当做是 0 号地址是分支指令，分支目标地址是 0。

分支指令执行时，会将分支指令的分支指令地址，计算得到的分支目标地址，是否成功跳转的信息送 BHT 表，BHT 表以当前分支指令的地址作为关键字，在 BHT 表内做全相联并发比较，如果数据缺失，需要将当前分支指令的相关信息载入 BHT 表中，并设置合适的分支预测历史位初值。注意载入过程中可能涉及到淘汰策略，这里 BHT 表本质上就是一个 cache。如

果有相符的表项，数据命中，表明当前分支指令的信息已经存放在 BHT 表中，这时只需要根据当前是否成功跳转的信息调整对应表项的分支预测历史位，并且处理 cache LRU 计数标记信息即可。

表 6.1 BHT 格式

Valid（1：有效）	分支指令地址	分支目标地址	分支预测历史（2 位）	LRU 置换标记
1	XXX	XXX	11,10 预测跳转	酌情设置
1	XXX	XXX	XXX	XXX
1	XXX	XXX	XXX	XXX
1	XXX	XXX	XXX	XXX
0（无效）	空	空	空	空
0（无效）	空	空	空	空
0（无效）	空	空	空	空
0（无效）	空	空	空	空

最早分支预测历史位采用 1 位数据表示，为 1 表示跳转，为 0 表示不跳转，科学研究表明，双预测位可在较低的成本下实现很高的预测准确率，所以现在普遍采用双位预测，双位预测的状态转换图如图 6.21 所示。当状态位为 00、01 时预测不跳转，为 10、11 时预测跳转。当前指令分支跳转成功与否将会决定状态的变迁。注意，初始值的设置也很重要，对于无条件分支指令，初始值如果是 00，则预测失败次数是 2 次，实际设计时应适当调整初始值。

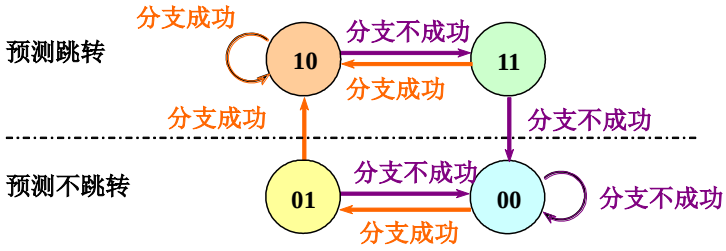

图 6.21 采用双位预测的 BHT 状态转换

BHT 表会放在取指令阶段，利用 PC 的值作为关键字进行全相联比较（此过程应与取指令过程并发），BHT 表命中表示当前指令一定是一条分支指令，此时可以根据 BHT 表中的历史预测位决定下条指令的地址是 PC+4 还是分支目标地址，如果预测正确，指令流水线不会停顿，如果预测失败，则分支指令在实际执行阶段还是应该清空预取的指令。当然如果 BHT 表缺失，表明当前指令可能不是一条分支指令或者是不经常使用的分支指令，则按照 PC+4 取下条指令。BHT 表的引入使得取指令 IF 阶段可以在指令并未取出的情况下进行分支预测，由于双位预测的高准确率，可以消除指令流水线中的大多数的分支延迟。

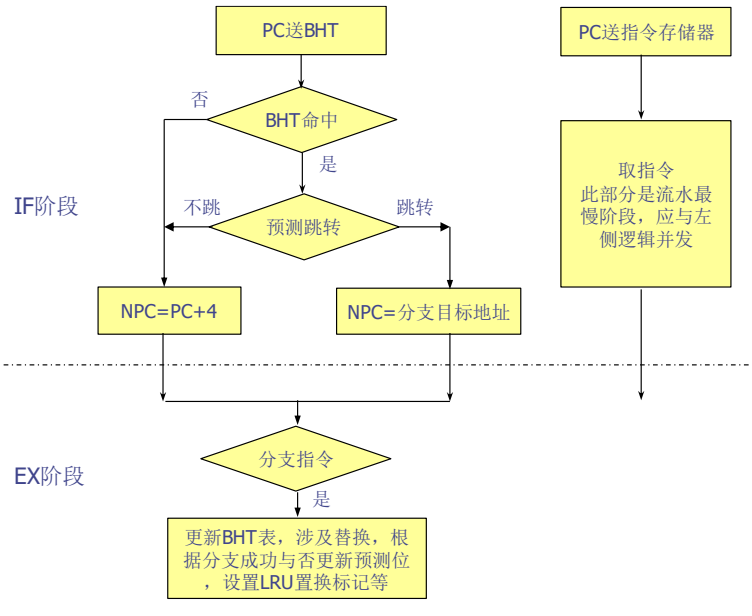

图 6.22 动态分支预测流程流程

6.5.3 实验内容

为流水 CPU 增加动态分支预测逻辑，取指令阶段可以以 PC 的值为关键字查询 BHT 表（Branch History Table），根据历史统计信息直接预测下条指令的正确地址，通过提升预测准确率，尽量避免分支指令引起的流水线清空暂停现象，从而优化指令流水线性能。

实际处理流程中取指令阶段以 PC 为关键字到 BHT 表做全相联比较查找，如果命中，直接根据预测位决定是否跳转，并给出下一条指令正确地址（取指令阶段，与取指令并行）。分支指令在指令执行 EX 阶段会根据跳转与否更新 BHT 中的表项，如当前执行指令不在 BHT 中，需要载入当前指令分支信息，如 BHT 表已满，需要进行淘汰。如 BHT 表命中，需要根据当前跳转成功与否情况更新双预测位，同时将 LRU 置换计数信息清零，以提升下次预测的准确性，具体流程如图 6.22 所示。

6.5.4 实验步骤

(1) 设计双位预测状态机组合逻辑电路，BHT 表中会使用该模块。输入为原预测状态位，实际跳转情况，输出为新的状态位。

(2) 设计 BHT 表，设计全相联并发查找机制，也就是增加多路并发比较机制，根据比较的结果获取分支地址以及分支预测位。

(3) 为 BHT 表增加写入逻辑，能够将一条新的分支指令信息加入 BHT 表，当有空位时能将表项内容添加到空位中，注意 BHT 的写入是在分支指令执行时实施的。

(4) 实现 BHT 表的 LRU 置换算法。

(5) BHT 功能测试：修改分支指令执行逻辑，如当前分支指令不在 BHT 表中，需要将当前指令相关信息载入 BHT 表，并设置合适的预测位初始值，载入过程可能涉及淘汰；如当前分支指令已在 BHT 表中，则根据跳转成功与否更新预测位以及 LRU 置换信息。利用分支预测测试程序测试 BHT 表的载入过程，淘汰策略是否正确，验证对应分支指令在执行时是否正确载入，初始预测位是否设置合理。该步骤只考虑 BHT 表数据的载入，并不进行分支预测。

```
//file: 分支预测测试程序.asm
#####################################################################
#用于BHT表才载入过程测试，LRU淘汰策略测试，BHT表预测功效测试
#####################################################################
.text
 addi $s1,$zero, 5   #循环次数
 j jmp_next1          #载入BHT,BHT有1个表项
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next1:
 j jmp_next2        #载入BHT,BHT有2个表项
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next2:
 j jmp_next3        #载入BHT,BHT有3个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next3:
 j jmp_next4        #载入BHT,BHT有4个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next4:
 j jmp_next5        #载入BHT,BHT有5个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next5:
 j jmp_next6        #载入BHT,BHT有6个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next6:
 j jmp_next7        #载入BHT,BHT有7个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
```

```
 addi $s2,$zero, 2
jmp_next7:
 j jmp_next8          #载入BHT,BHT有8个表项,后续会多次执行，应预测成功
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next8:
 addi $s1,$s1, -1
 bne $s1,$0, jmp_next2
 #载入BHT,BHT满，应淘汰 j jmp_next1，后续多次执行，跳转有成功有失败
 addi $s0,$zero,1
 addi $s2,$zero,255
 addi $s1,$zero,1
 addi $s3,$zero,3
 beq $s0, $s2, jmp_next9   #载入BHT,BTH已满，淘汰j jmp_next2
 beq $s0, $s0, jmp_next9   #载入BHT,BTH已满，淘汰j jmp_next3
 addi $s1,$zero, 1
 addi $s2,$zero, 2
jmp_next9:
 bne $s1, $s1, jmp_next10   #载入BHT,BTH已满，淘汰j jmp_next4
 bne $s1, $s2, jmp_next10   #载入BHT,BTH已满，淘汰j jmp_next5
 addi $s1,$zero,1    #不会执行
 addi $s2,$zero,2    #不会执行
jmp_next10:
 jal func                   #载入BHT,BTH已满，淘汰j jmp_next6
  addi   $v0,$zero,10
 syscall                    #程序退出，停机

func: addi $s0,$zero, 0     #子函数
      addi $s0,$s0, 1
      add    $a0,$0,$s0
       addi   $v0,$0,34
       syscall
       jr $31               #载入BHT,BTH已满，淘汰j jmp_next7
```

(6) **增加预测功能：**修改取指令 IF 阶段的下条指令 NPC 逻辑，利用 BHT 进行预测下条指令的地址，预测跳转信息应经过流水线向后传递，方便分支指令执行时进行判断；修改分支指令执行逻辑，根据预测是否正确决定是否需要清空预取指令。

(7) **系统联调，功能测试。**先调试分支预测测试程序，运行成功后再调试 benchmark 程序。

第7章 输入输出系统实验

7.1 AHB-Lite 总线设计实验

7.1.1 实验目的

学生掌握 AHB-Lite 总线协议，能通过 AHB-Lite 总线对存储器、简单接口（例如 LED 指示灯、七段数码管、开关、键盘等）进行访问，读取数据或输出信息。

7.1.2 背景知识

总线是处理器、存储器和 I/O 设备之间信息传递的通路，是计算机体系结构的重要组成部分。CPU 通过总线传送运行程序所需要的地址、数据及控制（指令）信息，也是各种外部设备接口电路的直接连接对象。

现代计算机的总线出现了许多新结构，推出了各种新标准与新技术，其中 AMBA 总线标准主要用于片内总线。AMBA（Advanced Microcontroller Bus Architecture）是由 ARM 公司推出的片上总线规范，它提供了一种特殊的机制，可将 RISC 处理器集成在其它 IP 核和外设中。第一代 AMBA 标准（AMBA1.0）定义了二组总线规范：AMBA 系统总线 ASB（Advanced System Bus）和 AMBA 外设总线 APB（Advanced Peripheral Bus）。AMBA2.0 在些基础上增加了 AHB（Advanced High-performance Bus）高性能总线。AMBA3.0 规范则进一步定义了一组四个接口协议，这些协议针对要求高数据吞吐量、低带宽通信，要求低门数、低功耗以及执行片上测试和调试访问的数据集中处理的组件，提出了片上数据通信要求。这四个总线协议除 AHB、ASB、APB 外，第四个就是 AXI（Advanced eXtensible Interface）总线协议，它丰富了现有的 AMBA 标准内容，满足超高性能和复杂的片上系统（SoC）设计的需求。AMBA4.0 规范在 AMBA3.0 规范的基础上又另外新增了三个接口协议：AXI4、AXI4-Lite 和 AXI4-Stream。

作为 AMBA 总线规范之一，AHB 总线是一个开源总线接口规范，广泛应用于嵌入式系统。AHB 总线能够方便微处理器连接多个设备或外设。AHB-Lite 是 AHB 总线规范的简化版本，仅支持单一的总线主设备。

AHB-Lite 总线主要由时钟（HCLK）、写使能（HWRITE）、地址（HADDR，32 位）、写数据（HWDATA，32 位）和读数据（HRDATA，32 位）组成，通常根据地址译码得到片选（CS）信号来进行输入输出设备的选择，如图 7.1 所示。

一个 AHB-Lite 总线传送过程包括 2 个时钟周期：第一个时钟周期为地址周期，第二个是数据周期。在地址周期，处理器作为主设备送出要访问的地址（HADDR），如果是写数据则使能读写控制信号（HWRITE），否则使该信号无效；在数据周期，如果是写数据，则在总线上送出要写的数据（HWDATA），否则读取总线上由从设备送入的数据（HRDATA）。具体读写过

程如图 7.2 和图 7.3 所示。

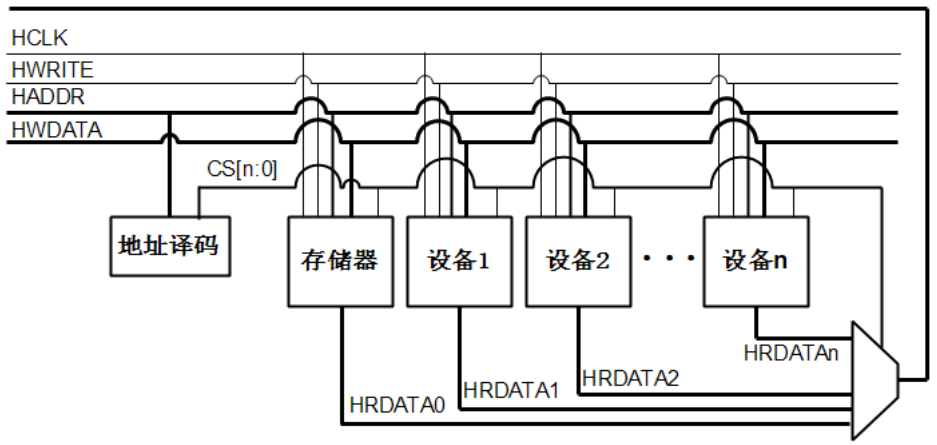

图 7.1 AHB-Lite 总线的构成

AHB-Lite 总线是通过地址译码来实现总线仲裁的，地址译码实际上是一个优先级译码器，它根据固定的地址范围优先级来确定设备访问的先后顺序。AHB-Lite 总线的优点是简单、高效，容易学习、掌握和使用，比较适合作为高速的系统总线使用；最大的缺点是数据的读写必须在 2 个时钟周期内完成，对于写操作还可以通过信号的缓存来解决，但是读操作中如果从设备不能在主设备给出地址后及时送出数据会引起读操作失败，另外该总线协议只支持单一的主设备，因此 AHB 总线如果作为种类繁多、速度各异的外部设备的总线接口并不是十分合适。

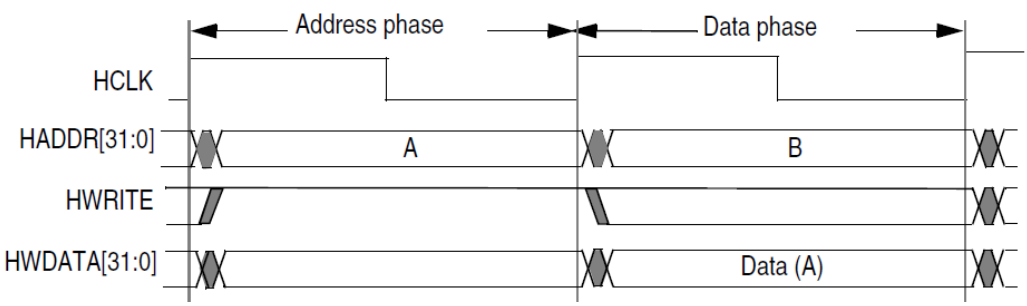

图 7.2 AHB-Lite 总线写数据的时序图

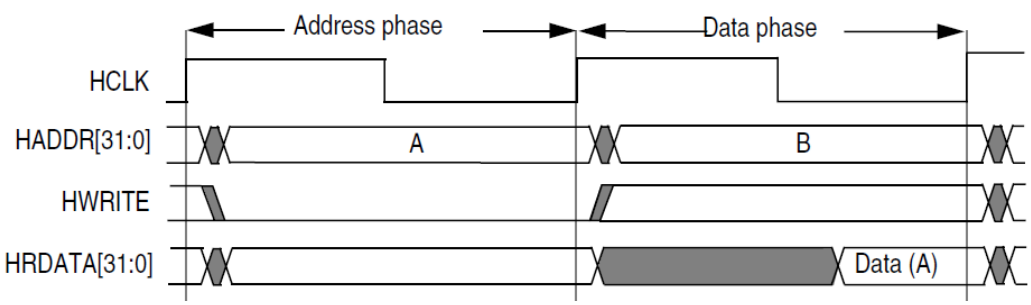

图 7.3 AHB-Lite 总线读数据的时序图

7.1.3 实验内容

通过 AHB-Lite 总线对存储器、LED 指示灯、键盘的输入、键盘的状态以及文本显示输出进行访问，能够正确存取存储器中的数据，控制 LED 指示灯的熄灭，根据键盘的状态读取键盘的输入信息，将希望的信息输出到文本显示器显示。AHB-Lite 总线实验的输入输出引脚如图 7.4 所示。

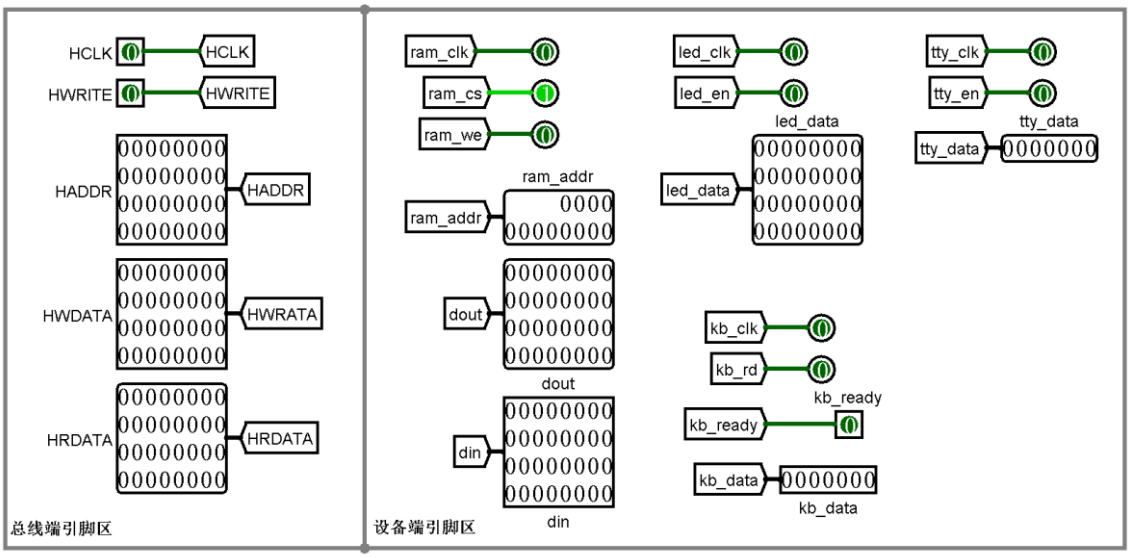

图 7.4 AHB-Lite 总线实验输入输出引脚

7.1.4 实验步骤

（1） **AHB-Lite 总线协议的实现**，参照图 7.1 给出的总线构成以及图 7.2 和图 7.3 所示的总线读写访问时序，按照图 7.4 所示的输入输出引脚，完成总线读写访问的电路。

（2） **AHB-Lite 总线测试**，AHB-Lite 总线实现后，将其放到图 7.5 所示的测试电路中进行测试，验证总线实现的正确性，同时观察总线是如何动作的。

7.1.5 实验思考

(1) AHB-Lite 总线是如何实现总线仲裁的？

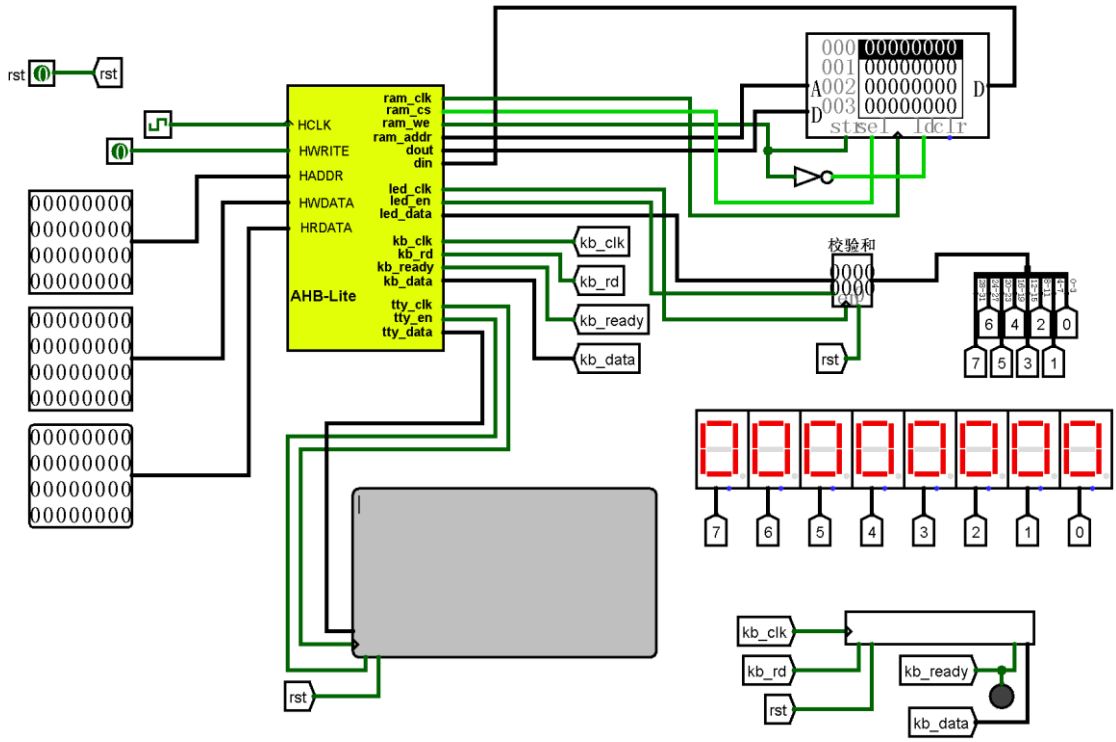

图 7.5 AHB-Lite 总线测试电路

7.2 程序查询控制方式编程实验

7.2.1 实验目的

学生掌握程序查询控制方式基本原理，能编写 MIPS 汇编程序利用轮询的方式与外部设备进行数据交互，理解程序查询控制方式中 CPU 占用率问题。

7.2.2 背景知识

程序查询控制方式是最原始、最简单的方式。其基本思想是，CPU 直接执行一段输入输出程序来实现主机与外设的数据交换，程序执行一次只能传送一个数据（字或字节）。每次传送前都要查询设备所处的状态，只有当设备准备就绪后才可进行传送。否则，主程序作循环查询等待操作。因此数据传送速度很慢，对于慢速外设，CPU 的利用率很低。

图 7.6 程序查询控制方式接口的示意图，从中可以看出程序查询控制方式接口的主要组成部分包括：地址译码器、数据缓冲器 DBR、设备状态寄存器 DSR 和有关控制逻辑等组成。

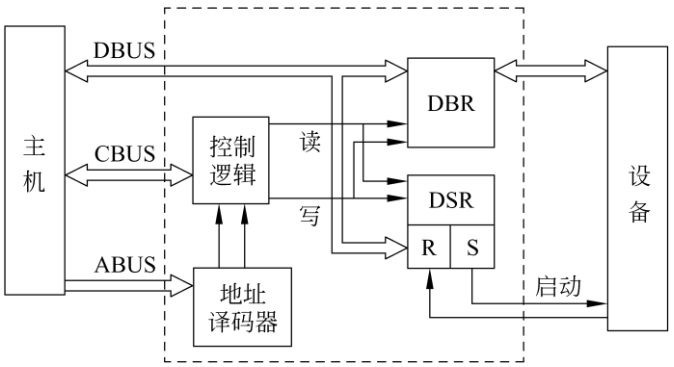

图 7.6 程序查询控制方式接口

地址译码器用来识别设备并完成设备选择功能；数据缓冲寄存器(DBR)用于主机与外设之间数据的缓冲。执行输入操作时，DBR 存放从 I/O 设备读取的数据，该数据将被 CPU 读取；当执行输出操作时，DBR 存放 CPU 送来的数据，该数据将输出至外设；设备状态寄存器(DSR)用于记录设备的状态，常见的状态信息如“忙”、“准备就绪”、“错误”等。在程序查询输入/输出方式下，CPU 通过执行程序查询设备状态位来判断设备的状态，以确定程序下一步流程。

程序查询方式中数据输入/输出流程如图 7.7 所示。外设被启动后，CPU 通过查询设备状态来检查设备是否准备好，若设备准备好，则 CPU 与外设之间进行一次数据交换；反之，CPU 继续查询设备的状态，直到设备准备好。在整个输入/输出过程中，CPU 只能执行与 I/O 操作有关的操作，即要么通过 I/O 指令查询外设的状态，要么通过 I/O 指令执行数据输入/输出。

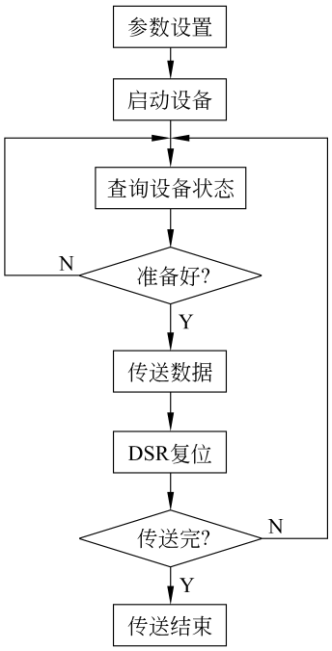

图 7.7 程序查询 I/O 流程

7.2.3 实验内容

编写 MIPS 汇编程序，利用程序查询方式接收 MARS 仿真器中键盘与字符显示设备仿真插件中的键盘按键动作，并自动将按键对应的 ASCII 编码输出到字符显示设备，实验效果如图 7.8 所示。

键盘与字符显示设备仿真插件提供键盘和字符显示两个外部设备的仿真，详细请参考第 10 章 第 10.6.6 节，该插件采用内存映射的方式与 MIPS 处理器进行数据交互，仿真设备内部 I/O 端口与内存映射情况如表 7.1 所示。

表 7.1 虚拟设备内存映射地址

设备	寄存器（8 位）	内存映射地址	备注
键盘设备	数据缓冲寄存器 DBR	0xffff0004	
键盘设备	设备状态寄存器 DSR	0xffff0000	最低位为 Ready 就绪位 次低位为中断使能位
字符显示设备	数据缓冲寄存器 DBR	0xffff000c	
字符显示设备	设备状态寄存器 DSR	0xffff0008	最低位为 Ready 位 次低位为中断使能位

该插件与 MIPS 处理器连接后，MIPS 程序可以通过 MMIO（内存映射 I/O）方式接收该插件键盘区域的按键动作，当有按键动作时，按键对应的字符会直接回显在插件窗口下方的中的键盘文本区，同时对应按键的键值会存放在数据缓冲寄存器 DBR，并将设备状态寄存器 DSR 的 Ready 位置 1，表示按键就绪。当 MIPS 程序使用 LW 指令读取数据缓冲寄存器 DBR 中的按键数据时，设备状态寄存器 DSR 的 Ready 位自动清 0。

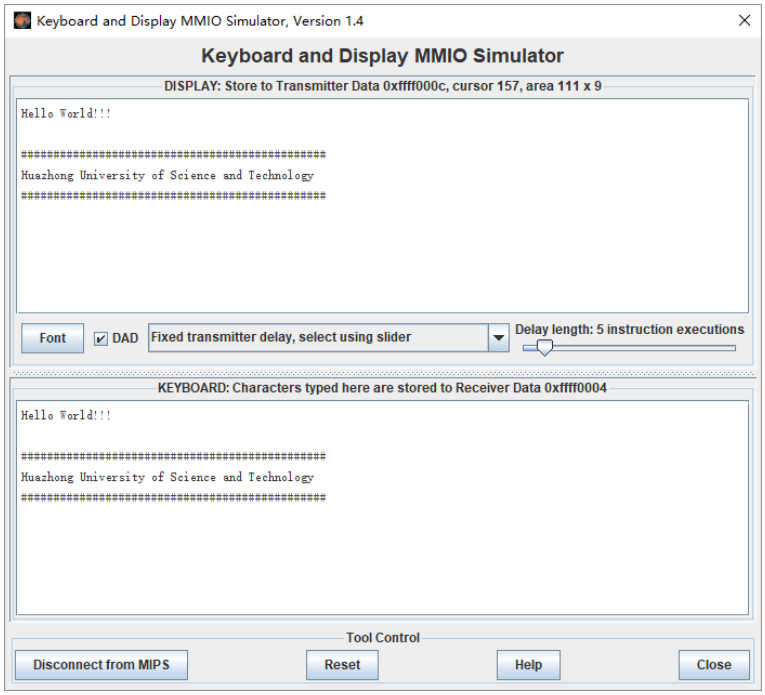

图 7.8 轮询键盘并回显按键

当字符显示设备状态寄存器 DSR 的 Ready 位为 1 时，MIPS 程序才可以对字符显示设备进行写操作，此时 MIPS 程序可以利用 SW 指令将待显示的 ASCII 字符写入数据缓冲寄存器，相应字符就会在插件字符显示区域显示，该动作会触发显示设备状态寄存器 DSR 的 Ready 位清 0，并延迟一段时间以仿真慢速字符显示的处理过程，数据显示完毕后再将 DSR 的 Ready 位置 1，具体延迟时间可以通过显示区域的滑动条进行配置。需要注意的是，只有在仿真插件中点击“连接到 MIPS”或 Reset 按钮才能将字符显示设备状态寄存器 DSR 的 Ready 位置 1，运行程序时需要注意。

在程序查询 I/O 方法中，MIPS 程序只能不停的循环测试对应设备的设备状态寄存器，根据状态寄存器的值确认后续操作，这种方式中 CPU 除了轮询设备状态，无法完成其它有意义的功能，CPU 被完全占用。

7.3 中断服务程序编程实验

7.3.1 实验目的

学生掌握中断处理机制基本原理，能编写 MIPS 汇编程序利用中断处理方式与外部设备进行数据交互，理解中断处理机制中 CPU 占用率问题。

7.3.2 背景知识

程序查询控制方式中，CPU 必须在特权态且关闭外部中断的情况下，不断的执行程序对设备状态进行循环查询（轮询），直至设备就绪，才能完成一个数据的传输。以输入为例，当 C 语言程序运行到 getchar()函数时，需要从键盘接收数据输入。如果采用程序查询方式，只能不断的查询是否有键盘按下，等待按键动作，等待过程中 CPU 由于外部中断关闭且处于特权态，所以不能处理其它事务。当键盘状态寄存器显示有按键事件时，后续程序从键盘缓冲区取走对应键值，CPU 才能执行后续语句，这种方式常见于早期的单任务的系统，对于慢速设备，其 CPU 占用率较高，但高速设备则没有这个问题，所以现在也有一些高速设备采用程序查询方式进行数据传输。

相比程序查询控制方式，中断机制可以实现主机和外设并行工作，当 CPU 需要和外部设备进行数据交互时，首先启动外部设备。当慢速的外部设备准备数据时，CPU 仍然可以执行其它有意义的程序：由操作系统将等待外部设备的进程挂起，并切换到其他进程执行。当外部设备数据准备就绪时，通过外部设备中断请求告知 CPU；CPU 在执行完每条指令后判断当前是否存在中断请求，如果有中断请求，则暂停主程序的执行，由硬件系统进行中断识别（判断是何种设备或事件引起的中断），并通过中断向量表等机制跳转到对应设备或事件的中断服务程序（中断识别以及跳转的过程需要时间开销），然后由 CPU 执行中断服务程序完成与设备的数据交互后执行进程调度，唤醒之前被挂起的进程。

还是以 getchar()函数为例，若采用中断机制，执行 getchar()函数的进程在执行到该函

数后就会被挂起，进入键盘 I/O 等待队列，CPU 从就绪队列中调度其他进程运行。当按键发生时，由硬件产生键盘中断，CPU 暂停当前程序，转向键盘中断服务子程序执行，键盘中断服务子程序从键盘缓冲区拷贝对应键值到系统缓冲区，唤醒处于挂起状态的执行 getchar()函数的进程，将其加入就绪队列，并最后转进程调度。

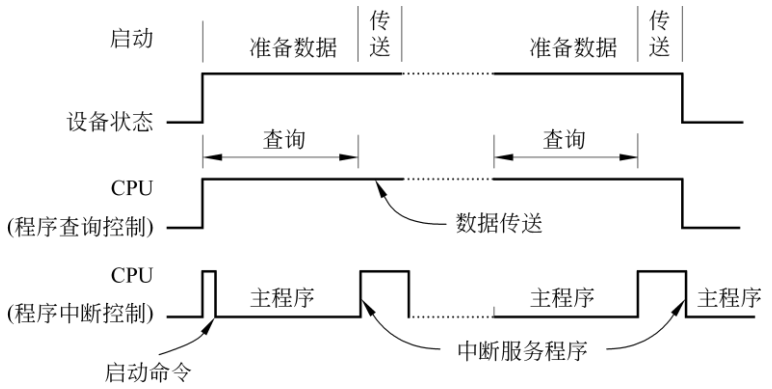

图 7.9 两种方式中 CPU 运行的轨迹

图 7.9 将 CPU 使用中断和不使用中断两种情况的轨迹进行了比较。在这个过程中才能不难发现，使用中断技术后，CPU 原来用于查询外设状态的时间被充分地利用起来，工作效率得到了显著的提高。

中断服务程序和函数调用类似，二者本质上都是函数体，MIPS 中采用 JAL 指令进行函数调用，与中断服务程序的执行类似，二者都需要将程序计数器 PC 保存到某个寄存器，然后修改 PC 的值。不同的是 JAL 将 PC 保存到 31 号通用寄存器，中断则将其保存到 EPC 中；修改后 PC 的新值，在 JAL 中由指令给出，中断则直接由硬件识别给出中断服务程序入口地址；另外 JAL 是一条指令，由用户在程序中显示调用，当 PC 指向该指令时执行，中断则不是一条指令，它是在满足中断响应条件时执行，是由随机事件触发的，并没有显示的调用位置。

另外二者的函数体中都需要保护现场，恢复现场，现场包括所有可能会在函数体中覆盖旧值的寄存器，一般不同的操作系统会通过 ABI 接口（Application Binary Interface）事先约定好哪些属于调用者函数保护，哪些属于被调用函数保护，以便编译器生成代码。由于中断服务程序的执行并没有调用者，所以中断服务程序中还应该额外保存一部分本应该由调用者保护的寄存器。最后函数调用返回与中断服务程序返回使用的指令不同，前者一般通过 jr $ra 返回，中断通过 eret 指令返回。

7.3.3 实验内容

编写 MIPS 汇编程序，利用中断处理方式接收 MARS 仿真器中键盘与字符显示设备仿真插件中的键盘按键动作，并自动将按键对应的 ASCII 编码输出到字符显示设备，要求主程序执行一段能运行较长时间且有意义的工作。实验思路如下：

1） 打开实验报中的 keyboard.asm 程序，找到该程序入口位置，开启中断处理机制的程

序：利用访存指令设置键盘虚拟设备的状态寄存器 DSR 中的中断使能位为 1，当状态寄存器 DSR 中的 Ready 位为 1 时，如果中断使能位为 1，则触发外部中断。

2） MARS 在每条指令执行完毕后，会检测是否存在外部中断，当检测到外部中断时，会将将异常代码 0(代表外设中断)写入协处理器 CP0 的 CAUSE 寄存器($13)的 2~6 位，并将第 8 位设置为 1,标识中断源为键盘,将程序计数器 PC 的值存入 EPC 寄存器($14)，并检查 0x80000180 处是否存在中断/异常处理程序。如果存在，则将程序计数器设置为该地址。否则程序执行将终止，并向"RUN I / O"选项卡发送消息。中断处理机制允许 MIPS 主程序执行有用的任务，而不是在轮询设备状态的循环中空转。

3） 编写键盘中断处理程序，中断处理程序应包括保护现场，中断服务，恢复现场，开中断，中断返回四部分，由于 MARS 只支持简单的中断机制，开中断部分可以忽略，具体代码流程可以参考第 10 章 10.7 节。要求中断处理程序必须对 Keyboard.asm 中频繁使用的寄存器进行改写，所以中断处理程序开始时保护现场过程中应将该寄存器压栈保存，中断返回时应该将该寄存器恢复，以避免影响主程序正常工作。中断服务部分要求将键盘键值直接回显在该插件的字符显示设备中。

7.4 单级中断机制设计实验

7.4.1 实验目的

学生掌握单级中断处理机制，能在单周期 CPU 中设计单级中断处理机制，能处理多个外部中断事件，能正确的暂停主程序的执行，转为为按钮事件服务的中断服务程序，中断服务程序执行完毕后应返回主程序继续运行，不同的按钮会进入不同的中断服务程序。

7.4.2 背景知识

计算机系统运行时，若系统外部、内部或当前程序本身出现某种非预期的事件，CPU 将暂时停下当前程序，转向为该事件服务的中断服务子程序，待事件处理完毕，再恢复执行原来被暂停的程序，这个过程称为中断。产生中断的事件对 CPU 来说是随机发生的，中断技术把有序的程序运行和无序的中断事件统一起来，大大增强了计算机系统的处理能力和灵活性。中断是现代计算机普遍采用的一项重要技术，可以实现主机和外设并行工作，以提高了整个系统的工作效率，可以方便计算机进行程序调试、人机交互、故障处理、实时处理

(1) 中断分类

根据引起中断的事件来自于CPU内部还是CPU外部，可分为内中断(也称为软中断)和外中断(也称为硬件中断)；根据进入中断的方式可分为自愿中断和强迫中断；根据其重要性可分为可屏蔽中断和不可屏蔽中断，如图7.10所示。

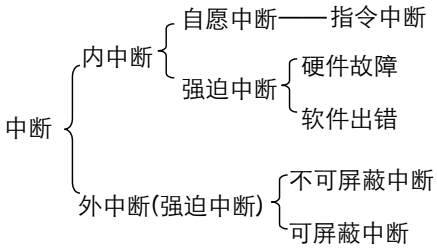

图 7.10 中断分类

(2) 中断优先级

当多个外部设备同时产生中断请求时，就存在着优先级的问题，优先级高的先响应，优先级低的后响应，中断优先级分为响应优先级和处理优先级。响应优先级是指 CPU 对各设备中断请求进行响应的先后次序，它根据中断事件的重要性和迫切性而定，是在硬件电路上进行固定的，无法更改。处理优先级是指 CPU 实际完成中断服务程序的先后次序，可以通过中断屏蔽技术动态调整外部设备的处理优先级，如果不采用中断屏蔽技术，处理优先级和响应优先级相同。

(3) 中断嵌套

根据设备中断服务子程序能否被其它中断请求再次中断，可以分为单级中断和多重中断。如果一个中断服务程序被执行，则 CPU 不响应其它中断请求（包括比本中断服务程序优先级更高的中断），而只有在该中断服务程序执行完成后才能响应其它中断请求，这种中断就是单级中断。在中断服务程序执行过程中，如果允许 CPU 响应其它中断请求，则这种中断称为多重中断，也称中断嵌套。中断嵌套中高优先级中断可以中断低优先级中断。

(4) 中断仲裁

同一时刻可能有多个外部设备同时发出中断请求，如何选择适当的设备进行中断响应，这就是中断仲裁需要解决的问题，中断仲裁方法和集中式与总线仲裁方式类似，仲裁方式与中断请求与 CPU 的连接关系有关，主要分为链式查询、独立请求、分组链式三种。

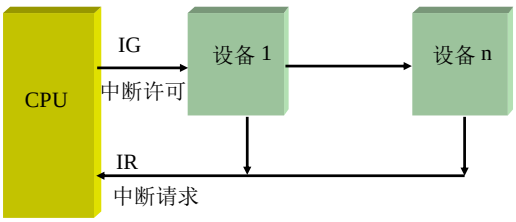

图 7.11 链式请求

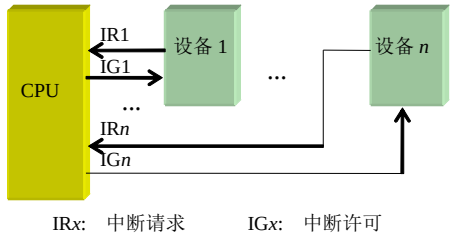

图 7.12 独立请求

链式请求方式中 CPU 只有一个中断请求输入 IR，所有设备都连接到 CPU 的 IR 引脚，需要发出中断请求时将 IR 置 1，CPU 收到中断请求 IR 后，通过中断许可信号 IG 对设备进行授权，授权信号 IG 首先传递给设备 1，如设备 1 没有中断请求，设备 1 会将授权信号沿请求链转发给下一个设备，直至正确的中断设备,中断服务程序处理完成后再将对应的 IR 请求清除。这种方式成本较低，优先级固定，离 CPU 近的设备中断优先级高，当优先级高的设备过于频

繁的请求中断时，末端低优先级设备可能存在着饥饿现象；链式请求中断响应较慢，中断授权信号传递需要多个不确定的时钟周期才能实现中断授权；另外这种方式存在单点故障，可扩展性较差。

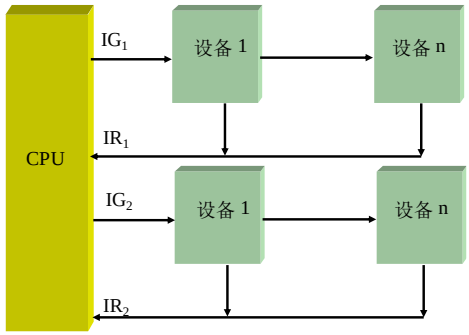

图 7.13 二维分组链式请求

独立请求方式中每一个外部设备都有一组中断请求 IR 与中断授权 IG 信号与 CPU 相连，CPU 可以根据响应优先级直接进行中断响应，处理灵活，但 CPU 引脚过多。受限于引脚资源，实际系统中往往采用分组链式的折中结构，如图 7.13 所示。二维分组链式结构中每一组 IR，IG 信号被多个设备采用链式结构共享，称为中断共享，每一个中断请求链共享一个中断号。

(5) 中断识别

中断识别的任务是确定中断请求是由哪个外部设备（中断源）发出的，不同的中断请求连接方式的中断识别方法不同，目前主要有程序查询、硬件查询和独立请求三种中断识别的方法。独立请求方式识别最为简单，链式或分组链式相对较为复杂。以独立请求方式为例，来自中断请求寄存器 IR 的多个中断请求信号与中断屏蔽寄存器 INM 逻辑与后进入优先编码器，如图 7.14 所示，被屏蔽的中断请求将无法抵达 CPU，优先编码器根据响应优先级生成中断请求的编号 xy（中断号），完成中断识别，并将优先编码器的所有 IR 输入信号逻辑或后得到中断请求信号，最终与中断使能寄存器 IE（由开中断，关中断指令控制）逻辑与后送 CPU，CPU 执行完一条指令后如果发现有中断信号则进入中断处理流程。注意中断使能寄存器 IE 为零表示关中断，此时 CPU 中断请求信号将为零，所有外部设备的中断请求均无法抵达 CPU，也就意味着 CPU 不会响应任何外部中断。

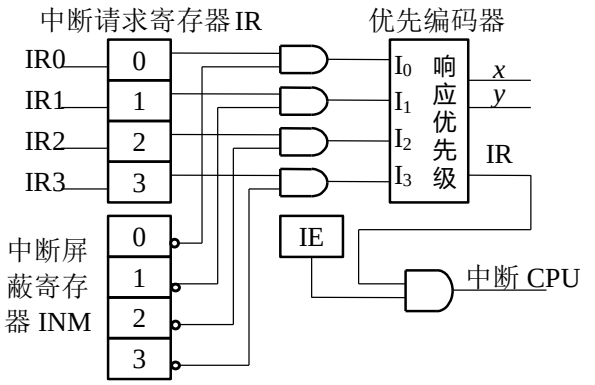

图 7.14 中断屏蔽与中断识别

在完成中断源的识别后，还需要获得对应的中断服务程序入口地址，这样 CPU 才能转为执行该中断服务程序。获得中断服务程序入口地址的方法主要有两种：向量中断法和非向量中断法。其中向量中断法对应独立请求中断方式，**向量中断法**将中断向量（中断服务程序的入口地址和程序状态字 PSW）以数组形式构成一个**中断向量表**，用中断号作为下标进行索引，本质上是中断服务程序入口地址的查找表。

向量中断法中断响应流程如下：先将各个中断服务程序的中断向量组织成中断向量表；中断响应时，通过相应方法识别中断源获得中断号，然后通过中断向量表入口地址计算得到该中断的具体中断向量地址，再根据中断向量地址访问中断向量表，从存储器中读出中断服务程序的入口地址，并装入程序计数器 PC 中和条件状态寄存器中，CPU 开始执行中断服务程序。

当一个中断号被多个同类设备共享时（链式、二维链式结构），引起该中断的具体设备必须采用非向量中断法获取具体中断服务程序入口地址。非向量中断法的中断响应方式为：CPU 在响应中断请求时，只产生一个固定的地址，该地址是中断查询程序的入口地址，通过执行该查询程序来确定中断服务程序的入口地址，然后执行相应的中断服务程序。

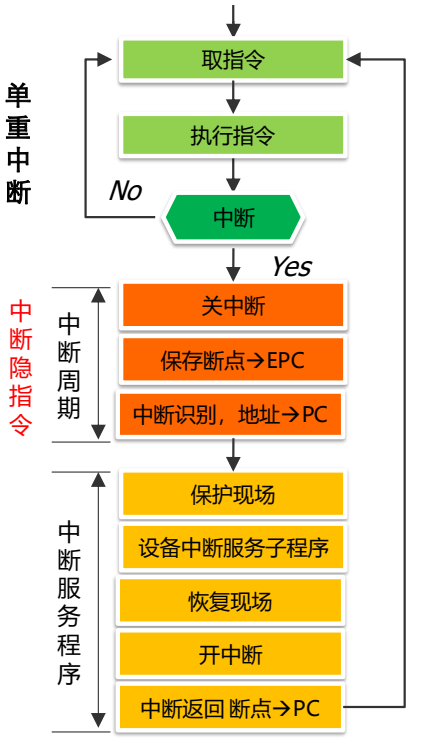

图 7.15 单级中断流程

(6) 中断响应

CPU 响应中断的时机是指令执行周期结束后，如图 7.15 所示，当 CPU 中断请求信号 IR 为高电平时表明有中断，暂停当前程序的执行，进入中断隐指令周期。此时 CPU 首先通过将中断使能寄存器 IE 清零的方式关中断，同时保存程序断点---指令寄存器 PC 的值，MIPS 中是将 PC 送异常程序计数器 EPC (exception program counter，CP0 中的一个寄存器)，同时进行中断源识别，获取中断服务程序入口地址，并将中断服务程序入口地址送 PC，**这一系列动作全**

部由硬件逻辑完成，需要一定的时间开销（可能需要若干时钟周期，此阶段 CPU 并不能执行指令），相关动作对应一系列数据通路，通常将这部分动作称为中断隐指令（实际它并不是 ISA 指令集的一部分），中断服务程序入口地址送 PC 后，CPU 便完成了暂停现有程序，转去中断服务程序的过程。

值得注意的是，缺页中断的中断时机是一个例外，发生缺页中断时指令并不能执行完毕，所以缺页中断的时机并不是指令执行周期结束后的公操作，而且这条未能执行的指令中断结束后应该重新执行，所以其保存的断点 PC 的值也不一样。

(7) 中断服务

中断服务程序首先是通过堆栈压栈的方式保护现场，现场包括所有可能被破坏的通用寄存器，状态字等，与普通函数调用中的保护现场不同，中断服务程序并没有调用者，所以本应该由调用者函数负责保存的部分寄存器可能也需要中断服务程序保存，凡是中断服务程序会写入的寄存器都需要保护。保护现场完成后正式进入为设备服务的中断服务子程序，进行设备数据交互，该程序结束后通过堆栈出栈的方式恢复现场，然后开中断，允许 CPU 响应其它中断，最后执行中断返回指令 eret 返回主程序，其任务是将断点地址送 PC。值得注意的是，**开中断应该与中断返回同时完成**，否则开中断之前如果已有中断请求在等待，中断返回 eret 指令将会被中断，保存断点的 EPC 寄存器将会被破坏，这里关中断应由硬件自动完成。另外中断返回时当前中断事件已经处理完毕，应该将当前中断服务程序对应的中断请求信号 IR 清零。

中断隐指令部分是由硬件实现，中断服务部分是由 CPU 执行中断服务子程序实现，其中包含保护现场、恢复现场涉及较为复杂的访存操作，这些都需要占用 CPU 时间，一次中断过程只能传输一个字，传输效率较低，所以中断机制比较适合为外设的随机事件服务，不太适合高速的数据传输。

7.4.3 实验内容

为已经实现的单周期 MIPS CPU 增加单级中断处理机制，引入 3 个外部中断源，如图 7.16 左下角所示，该 CPU 支持 3 个 Logisim 按钮触发的中断源，分别对应编号为 1、2、3 的三个按钮，3 个 LED 指示灯 IR1、IR2、IR3 分别表示对应中断源的中断请求信号，中断处理完成时熄灭，3 个中断源分别对应不同的中断优先级，其优先级顺序为：3>2>1>CPU。为方便实验检查，要求按下 1 号按键后 W1 指示灯点亮，并能进入数字 1 的走马灯演示子程序，中断演示程序运行结束后 W1 指示灯熄灭；按下 2 号按键后能进入数字 2 的走马灯演示子程序，按下 3 号按键后进入数字 3 的走马灯演示子程序，指示灯点亮熄灭规则与 1 号键相同。

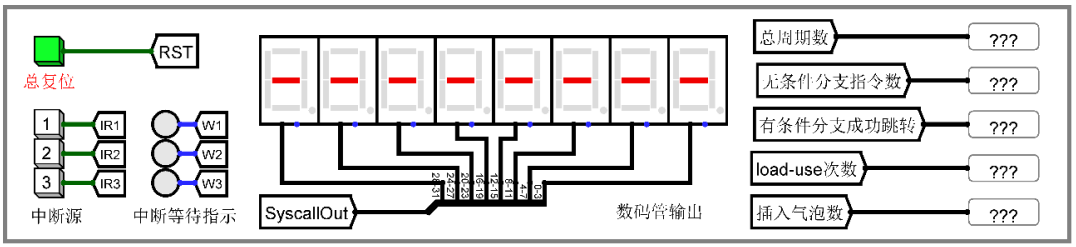

图 7.16 支持单级中断的单周期 MIPS CPU 输入输出引脚

走马灯演示程序如下，作为中断服务程序使用时还需要进行适当修改。

```
#file: 中断演示V2.0.asm
#############################################################################
#中断演示程序，简单走马灯测试，按下1号键用数字1循环移位测试
#                                按下2号键用数字2循环移位测试
#最右侧显示数据是循环计数
#这只是中断服务程序演示程序，方便大家检查中断嵌套，
#设计时需要增加开中断，关中断，保护现场，恢复现场的相关指令
#############################################################################
.text

addi $s6,$zero,1        #中断号为1，根据中断源不同进行修改
addi $s4,$zero,6        #循环次数初始值
addi $s5,$zero,1        #计数器累加值
#############################################################################
#逻辑左移，每次移位4位
#显示区域依次显示0x00000016 0x00000106 0x00001006 ... 10000006    00000006
#############################################################################
IntLoop:
add $s0,$zero,$s6

IntLeftShift:
sll $s0, $s0, 4
or $s3,$s0,$s4
add      $a0,$0,$s3             #display $s0
addi     $v0,$0,34              # display hex
syscall                         # we are out of here.

bne $s0, $zero, IntLeftShift
sub $s4,$s4,$s5                 #循环次数递减
bne $s4, $zero, IntLoop

addi     $v0,$zero,10           # system call for exit
syscall                         # we are out of here.
```

7.4.4 实验步骤

（1） **设计中断按键信号产生电路**，具体可参考如图 7.17 所示，这里 IR1 就是中断请求寄存

器，其中同步清零信号用于清除中断请求信号，注意中断请求信号必须等待中断服务程序执行到中断返回时才能清除。

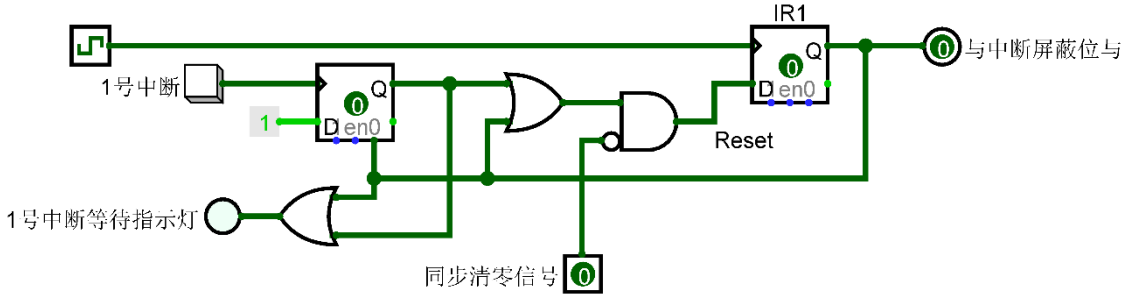

图 7.17 中断按键采样电路

（2） **设计中断识别逻辑**，能实现实验要求的中断响应优先级，能正确识别 1~3 号中断源，并设计向量中断机制，可由中断号寻找中断程序入口地址，为简化实现，中断向量表可以直接用硬逻辑实现。中断识别部分设计可参考图 7.14，由于不需要动态调整中断处理优先级，所以中断屏蔽寄存器部分电路可以省略。

（3） **存储区间规划**，考虑数据存储器存储区间如何划分，数据区和堆栈区如何分配，其中堆栈区主要用于中断服务程序中的保护现场和恢复现场的堆栈操作，需要考虑堆栈指针$sp 如何进行初始化，堆栈放在哪个地址合适；另外需要考虑指令存储器的区间规划，中断服务子程序存放在哪里比较合适，堆栈指针在哪里进行初始化？

（4） **实现 CP0 寄存器组中与中断相关的寄存器，**包括中断使能寄存器 IE，异常程序计数器 EPC，同时增加开关中断的硬件实现逻辑。

（5） **设计中断隐指令数据通路**，包括保存断点至 EPC、中断识别、中断服务程序入口地址送 PC 等，可参考图 7.15 的流程。

（6） **增加中断返回指令 eret 的支持**，需要重新设计控制器，注意 eret 指令应该同时开中断。否则 EPC 的值有可能会被破坏。

（7） **编写中断服务程序，**修改中断演示程序使得不同的中断源演示效果有区别；增加保护现场、恢复现场的代码，保证中断程序不破坏主程序的正常运行；最后程序结尾还需要增加中断返回代码；

（8） **系统联调，功能测试**，在主程序运行时任意点击 1、2、3 号按键中的任何一个，均能正常进入中断演示程序，处理完毕后能正确返回主程序，如中断服务程序执行期间又有新的按键被按下，中断返回时应直接进入新的中断服务。

7.4.5 实验思考

(1) 不同的中断请求存储在哪里，何时消失？

不同的中断请求通常保存在 CP0 寄存器组中的中断请求寄存器 IR 中，其寄存器的每 1 位对应一个中断请求。对应中断请求应在中断服务处理完毕时清除，这里应该由硬件完成这一清零动作。

(2) 硬件响应优先级用什么电路实现，为什么要有处理优先级？

硬件响应优先级一般采用优先编码器实现，同一时刻优先级最高的中断请求被 CPU 响应。只有没有更高优先级的中断请求时，电路才会把较低优先级的中断请求送给 CPU，优先编码器在 logisim 中有现成的模块。

(3) 中断使能寄存器 IE 是干什么用的？

中断使能寄存器就像是一个开关，用来打开或关闭 CPU 的中断请求，外部设备的中断请求信号会与这个开关进行逻辑与操作。只有中断使能有效时，CPU 才能接收到中断请求，无效时 CPU 无法接收到任何中断请求，也就无法响应中断。

(4) 开中断，关中断如何实现？

X86 指令集中的 STI（开中断）、CLI（关中断）指令就是用于控制中断使能寄存器的；MIPS 处理器中没有专门的开中断、关中断指令，具体实现时是利用 CP0 寄存器组访问指令 MFC0、MTC0 实现的，用 MFC0 指令将 IE 的值读取到一个通用寄存器，然后修改这个通用寄存器（根据需要置 0 或 1），最后用 MTC0 指令将这个值从通用寄存器写回 CP0，从而完成对 IE 开关控制。

(5) CPU 如何判断当前有中断需要响应？

CPU 判断当前有中断响应的依据是连接 CPU 的中断请求信号 IR，当该信号为高电平时，CPU 必须进行中断响应。对具体外部设备而言，其中断请求要被 CPU 响应，必须在该中断请求没有被屏蔽，无更高优先级的中断请求，且中断使能寄存器 IE 有效，CPU 才能收到最终的中断请求信号，当 CPU 正在执行的指令执行完毕进入公操作阶段时，CPU 才会响应该设备的中断请求。

(6) CPU 发现当前有中断请求后要做什么动作，什么时候响应中断？哪些是硬件完成，哪些是软件完成？由硬件完成的动作需要多少个时钟周期，此时 CPU 能否执行指令？

CPU 发现当前有中断请求需要响应后，会进行如下一系列动作：①保存断点，中断程序即将执行的下一条指令的地址，也就是当前 PC 的值；②中断识别并跳转，根据中断源找到中断服务程序入口地址，修改 PC 值转到中断服务程序；③利用堆栈保护现场；④执行中断服务程序；⑤恢复现场，完成中断返回前的准备工作；⑥中断返回。 其中①和②由硬件完成，也就是所谓中断隐指令完成的任务，这部分可能需要多个时钟周期，具体与实现有关，此时 CPU 不能执行其它指令。其它部分则是由软件实现。

(7) 单级中断中的断点保存在哪里？

单级中断中的断点通常保存在一个专用的寄存器中，就 MIPS CPU 而言，是保存在 CP0 中的 EPC 寄存器中。

(8) 中断服务程序入口地址如何查找？

本实验中有 3 个中断源，资源不受限，最简单的方法就是采用独立式中断请求方式进行中断仲裁，利用优先编码器电路即可实现中断源的识别，再增加中断向量表逻辑实现中断号到中断入口地址的转换，真实的中断向量表涉及内存数据访问，逻辑较为复杂，为简化实验设计，中断向量表可以用组合逻辑电路固化。

(9) 中断处理程序中的现场有哪些，我们实验中需要考虑保存哪些现场？

中断响应时 CPU 要转去执行另外一个程序，执行完后再返回继续执行被暂停的程序，因此在改变 CPU 状态的任何操作之前，要被修改的状态都需要保存起来，中断返回时再恢复。CPU 的状态都是通过寄存器实现的，所以所谓保护现场就是要备份所有即将修改的寄存器的值。实验中需要考虑保存的现场主要有：中断返回地址 EPC、中断屏蔽字、以及在中断服务程序中需要写入的的通用寄存器。记住一个原则，你即将修改什么，就保存什么。

(10) 中断服务程序存放在指令存储器中的那个位置，如何载入到 ROM 中？

中断服务程序存放在指令存储器的一个特定位置，它的起始地址必须与中断响应时 CPU 赋予 PC 的值相同，按照习惯中断处理程序的入口地址通常是存储器最低的地址，即 0x00000000。中断服务程序一般是与主程序一起编译，然后统一做成存储器文件，加载到指令存储器中，当然具体存放地址可以根据需要进行调整。

(11) 数据堆栈放在哪里？MIPS 如何访问堆栈？

数据堆栈通常放在数据存储器中。MIPS 中并不存在类似 X86 中的 PUSH，POP 堆栈指令，具体使用堆栈是利用内存访问指令配合堆栈指针寄存器 SP 控制实现的，MIPS 处理器的堆栈是向下生长的，每压栈保存一个值，SP 要减 4，出栈时则 SP 需要加 4，具体例子如下：

```
# 设置 SP 指针
li sp, STACK_BASE_ADDR

# 入栈
addi sp, sp, -4
sw   ra, 0(sp)

# 出栈
lw   ra, 0(sp)
addiu sp, sp, 4
```

(12) 按键中断是电平触发还是跳变触发？连续按键如何处理？实际系统中是如何处理的？

按键一般是边沿触发，但是处理边沿中断触发通常较为困难，因此常用锁存器将边沿触发转换为电平信号，然后用电平中断触发。连续按键最简单的处理办法是在第一次按键没有处理

完之前，忽视后续的按键动作，即发生按键时锁存器锁存，不再接受新的按键信息，只有当 CPU 将对应按键中断处理完毕后，再开放这个锁存器。实际系统中这个锁存器是一个缓存，可以保存按键的队列信息，这样 CPU 就可以一个一个的处理，直至保存的按键缓存队列全部被处理。实际系统运行时中断处理响应非常快，操作系统也规定中断服务程序不能太长，连续按键都可以得到及时响应。

(13) 实验中的中断机制问什么要使用 CP0 寄存器组？在我们的实验中如何简化？

不使用 CP0 也可以实现满足实验任务的要求，但使用 CP0 寄存器组实现中断机制的目的是为了方便程序的汇编，实验中我们采用 MARS 汇编器来汇编程序，因此符合 MIPS 规范的实现更加方便。但 MIPS 处理器中 CP0 中寄存器位的定义较为复杂，具体实现时可以根据需要尽量进行简化，只要能实现中断相关机制即可。

(14) 最终实现的单级中断处理器中中断隐指令的开销是多少个时钟周期？

(15) 最终完成的单级中断处理器，进行中断处理时，你是否发现中断过程很慢？为什么？

7.5 多级中断机制设计实验

7.5.1 实验目的

学生掌握多级嵌套中断处理机制，能在单周期 CPU 中设计多级嵌套中断处理机制，能处理多个外部中断事件，能正确的终止主程序的执行，转为为按钮事件服务的中断服务程序，中断服务程序执行完毕后应返回主程序继续运行，不同的按钮会进入不同的中断服务程序，高优先级中断可以打断低优先级中断，高优先级中断服务程序执行完毕后应能返回被中断的中断服务程序，直至主程序。

7.5.2 背景知识

多级中断与单级中断的区别在于中断可以嵌套，其在中断隐指令阶段的逻辑与单级中断完全一致，区别在于中断服务程序流程略有区别，具体处理流程如图 7.18 所示，中断服务程序保护现场部分还应该包括中断屏蔽字的保护，方便动态调整中断优先级，另外由于中断服务程序需要实现嵌套，CP0 中 EPC 寄存器的值可能被破坏，所以 EPC 寄存器也应作为现场进行保护，保护现场结束后立即开中断（软件指令完成）以使 CPU 系统可以响应新的中断，中断服务程序结束后，要进行恢复现场的操作，注意恢复现场应该是原子操作，必须首先关中断（软件指令完成），恢复现场完成后进入中断返回，同时开中断（硬件实现）。

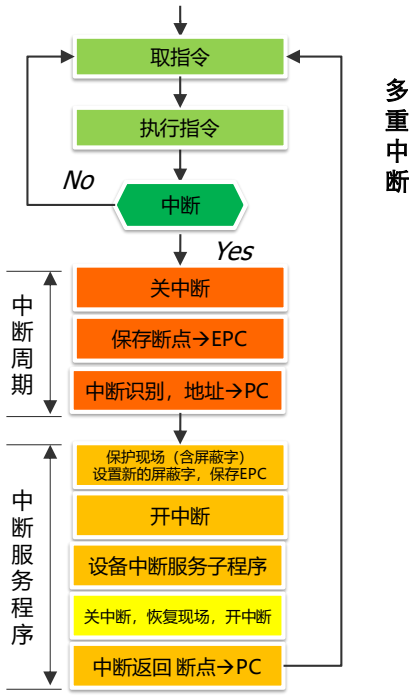

图 7.18 多级嵌套中断流程

7.5.3 实验内容

为已经实现的单周期 MIPS CPU 增加多级中断处理机制，引入 3 个外部中断源，如图 7.19 左下角所示，该 CPU 支持 3 个 Logisim 按钮触发的中断源，分别对应编号为 1、2、3 的三个按钮，3 个 LED 指示灯 IR1、IR2、IR3 分别表示对应中断源的中断请求信号，中断处理完成时熄灭，3 个中断源分别对应不同的中断优先级，其优先级顺序为：3>2>1>CPU。为方便实验检查，要求按下 1 号按键后 W1 指示灯点亮，并能进入数字 1 的走马灯演示子程序，中断演示程序运行结束后 W1 指示灯熄灭；按下 2 号按键后能进入数字 2 的走马灯演示子程序，按下 3 号按键后进入数字 3 的走马灯演示子程序，指示灯点亮熄灭规则与 1 号键相同。**高优先级中断应该正确中断低优先级中断服务子程序**。

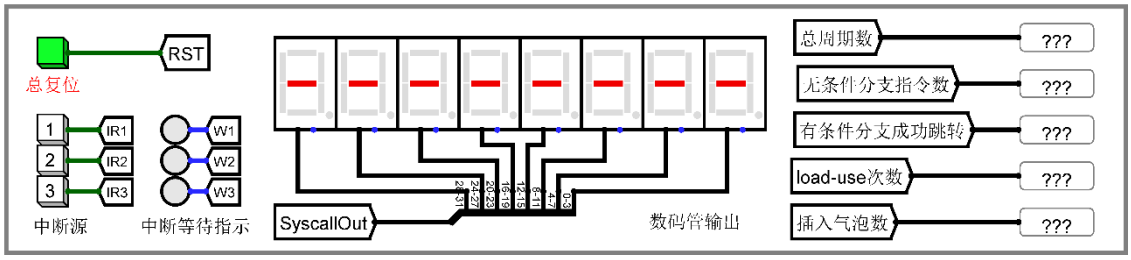

图 7.19 支持多级嵌套中断的单周期 MIPS CPU 输入输出引脚

7.5.4 实验思考

(1) 多级嵌套中断的断点如何处理？

单级中断中的断点保存在 CP0 中的 EPC 寄存器中，但 EPC 只是一个寄存器，进行多级中断嵌套时，会被后面的中断破坏，所以中断服务程序中应将 EPC 作为现场保存起来，EPC 实际上也是一个被调用者保存寄存器，类似 JAL 嵌套调用，需要保护$ra 寄存器一样。

(2) 高优先级中断服务程序执行过程中，有新的按键事件发生，如何处理？

由硬件判断新的按键事件的优先级高低。如比当前中断服务程序的优先级更高，就暂停当前正在执行的中断服务程序（类似于被暂停的主程序），转而执行为这个更高优先级的中断服务的中断服务程序;否则需要等待当前中断服务程序执行完毕并返回后,才能够进行中断响应;注意 CPU 响应不响应中断请求取决与能否收到中断请求信号，如果收不到就不响应。

(3) 中断屏蔽寄存器有什么作用，本实验是否需要中断屏蔽寄存器？

中断屏蔽寄存器的作用是保存中断屏蔽字，中断屏蔽字的用途是动态的调整中断处理优先级；正常情况下，优先级高的中断请求在被处理时，即当它的中断服务程序在执行时，是不允许被它同级或更低优先级的中断请求打断的。但是通过设置中断屏蔽字，可以允许它被低优先级的中断请求打断，从而先完成低优先级中断请求的中断服务程序，然后再执行完它自己的中断服务程序。中断屏蔽字是在中断处理过程中设置的，在真实计算机环境下，通常是由中断服务程序进行设置。由于本实验中并不需要动态调整中断处理优先级，因此可以不设置中断屏蔽寄存器。

7.5.5 实验步骤

（1） **增加软件开关中断机制，**需要增加 CP0 访问指令的支持，实现 MFC0、MTC0 两条指令，CP0 是独立于通用寄存器组的另外一组寄存器，这些寄存器不能直接被 MIPS 指令集引用，MIPS 指令集中由 MFC0，MTC0 两条指令负责访问，中断使能位 IE，中断屏蔽位，EPC 寄存器等均在 CP0 通用寄存器组中有详细约定，但其实现较为复杂，为简化实验，我们可以不遵循 MIPS CP0 的约定，根据本实验的需求进行最大的简化，能实现中断机制即可。

（2） **修改中断服务程序**，EPC 必须作为现场进行保护，注意 EPC 必须通过 MFC0 指令进行访问，恢复现场时也应恢复 EPC；保护现场结束后应执行 MFC0、MTC0 指令设置 CP0 中的中断使能位 IE 开中断，恢复现场之前也需要执行 MFC0、MTC0 指令进行关中断以保证恢复现场的原子性。

（3） **系统联调，功能测试**。功能测试流程如下:

- **单级中断功能检查**：在主程序运行时任意点击 1~3 号按键中的任何一个，均能正常进入中断演示程序，处理完毕后能正确返回主程序。

- **嵌套中断功能检查**：在主程序运行时依次点击 1~3 号按键，能正确演示 2 号中断打断 1 号中断服务程序，3 号中断打断 2 号中断服务程序，3 号中断服务程序执行完毕恢复执行 2 号中断服务程序，2 号中断服务程序执行完毕后再恢复执行 1 号中断服务程序，直至最后恢复执行主程序的嵌套过程。
- **嵌套特殊流程检查**：点击 2 号中断，进入中断服务程序后再点击高优先级的 3 号中断，进入 3 号中断服务程序后再点击 1 号中断，当 3 号中断服务程序执行完毕后应该能恢复执行 2 号中断服务程序，最后再响应 1 号中断请求，如果嵌套中断逻辑实现不佳，这里可能直接响应 1 号中断服务程序，2 号中断无法返回。

7.6 流水中断机制设计实验

7.6.1 实验目的

学生理解单周期中断和流水中断的差异，能应用流水相关知识，中断相关知识，为流水 CPU 增加中断处理机制，最终能正常运行并能正确运行标准测试程序，并能演示 3 个外部中断源的中断机制。

7.6.2 实验内容

为已经实现好的支持重定向机制的五段流水 CPU 增加中断异常处理机制（单级中断，中断嵌套不限），引入 3 个外部中断源，如图 7.20 图 7.19 左下角所示，该 CPU 支持 3 个 Logisim 按钮触发的中断源，分别对应编号为 1、2、3 的三个按钮，3 个 LED 指示灯 IR1、IR2、IR3 分别表示对应中断源的中断请求信号，中断处理完成时熄灭，3 个中断源分布对应不同的中断优先级，其中 3>2>1>CPU。为方便实验检查，要求按下 1 号按键后能进入数字 1 的走马灯演示子程序，按下 2 号按键后进入数字 2 的走马灯演示子程序，按下 3 号按键后进入数字 3 的走马灯演示子程序。

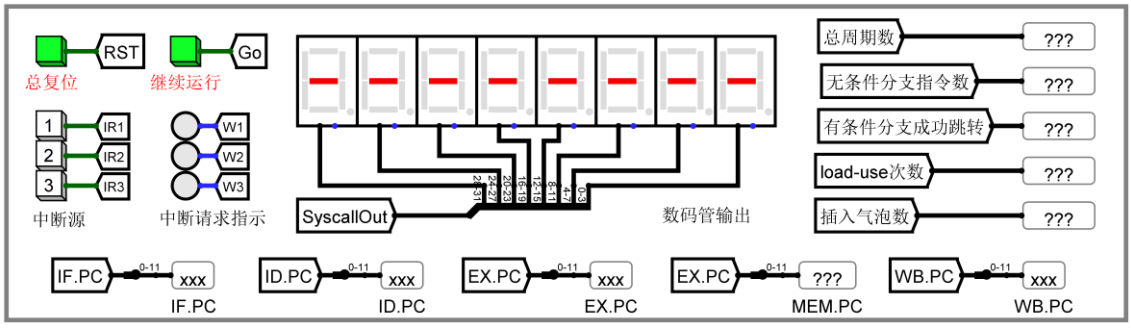

图 7.20 支持中断机制的流水 MIPS CPU 输入输出引脚

7.6.3 实验思考

(1) **单周期 CPU 中断处理和流水中断处理有何区别？**

在单周期 CPU 中，每条指令执行完毕后，CPU 就可以去判断是否有中断需要处理， 如果有就进行中断处理，如果没有则正常执行下一条指令。流水线中相对较为复杂，主要原因是在流水线中同时有 5 条指令在执行，这 5 级指令分别处于不同的处理阶段，即 IF、ID、EXE、MEM、WB 各有一条指令在执行，另外可能有多条指令都进入了结束阶段，比如无条件分支指令可能安排在 ID 段，有条件分支指令可能安排在 EX 段执行，中断时到底中断哪条指令的执行，同时执行的其它指令怎么办都需要考虑。

理论上，任意一个流水段上都可以处理中断，为了实现的方便通常会选择一个段来处理中断，可供选择的段常常是 EX 或者是 WB。但是，不管选择那个流水段来处理，都要做下面的事情，(以在 EX 段处理中断为例，其实在 WB 段处理更加方便)：①EX 段前面的 2 条指令要继续执行完；②EX 段之后进入流水线的 2 条指令要取消；③处于 EX 段本身的这条指令有两种选择，如果这条指令允许继续执行完，保存的断点地址应该是该指令的 PC+4；如果不允许这条指令继续执行，返回的断点就是这条指令的 PC;（通常做法是不允许这条指令继续执行）；另外如有指令在 EX 段之前完成，比如无条件分支指令在 ID 段完成，此时逻辑又有区别；④暂停流水线进行中断响应；（根据具体的实现方法也可以不暂停）；⑤重新启动流水线从新的 PC 值处取指令（新的 PC 即是中断入口地址）。