简单的类 MIPS 多周期流水化处理器实现

目的

完成多周期类 MIPS 流水处理器，通过检测竞争并插入停顿机制解决数据冒险、结构冒险、控制冒险。

设计思路

将处理器结构重新安排，具体包括：

拆散所有原有模块，将处理器划分为 IF、ID、EX、MEM、WB 五大阶段，每个阶段封装为一个模块，每两个阶段间有一组寄存器，作为流水线的数据暂存，命名为 IFID、IDEX、EXMEM、MEMWB；

规定除 reset 和 clock 信号外的其他连线均只能连接相邻层，如 IF 只与 IFID 连接、ID 只与 IFID、IDEX 连接；

规定所有阶段内操作在时钟上升沿进行，所有阶段间寄存器的读写在时钟下降沿进行。

为避免控制冒险，进行如下调整：

在 IF（取指令）阶段判断指令是否为 beq 指令，若是，则加入长度为 3 个周期的 STALL，等待 beq 指令结果。在此期间 IF 只能发出 nop 指令，且不更新 PC 的值；

在 IF（取指令）阶段判断指令是否为 j 指令，若是，则直接更新对应的 PC，并使 IF 发出的指令为 nop。

为避免数据冒险，进行如下调整：

在 IF（取指令）阶段加入一组计数器（每个计数器为 3 位，即计数为 0 至 7），记录每个寄存器的最后一个写入操作还有多久可以完成；

计数器的初值均为 0，每个时钟上升沿时将值非 0 的计数器的值加 1；

对每个指令，IF 先查询指令中需要读的寄存器对应的计数器的值，若为 0，则可以正常发布指令，并将此指令需要写入的寄存器对应的计数器的值设为 4（即计时 4 个周期后认为写入操作已完成），否则发出 nop 指令，且不更新 PC。

仿真描述

预先在数据存储器的 0 号至 7 号地址分别写入 1 至 8，共 8 个数据；在指令存储器内写入指令如下：

100011 00000 00001 0000000000000100 //lw $1,4($0)

000100 00000 00001 0000000000000010 //beq $0,$1,2（跳过下 2 条指令，若未解决 hazard，则此时$0 与$1 均为 0，会跳过）

100011 00000 00010 0000000000000000 //lw $2,0($0)

000000 00001 00010 00011 00000 100000 //add $3,$1,$2

000000 00001 00010 00100 00000 100010 //sub $4,$1,$2

000000 00001 00010 00101 00000 100100 //and $5,$1,$2

000000 00001 00010 00110 00000 100101 //or $6,$1,$2

000000 00001 00010 00111 00000 101010 //slt $7,$1,$2

000000 00010 00001 01000 00000 101010 //slt $8,$2,$1

000100 00111 00000 0000000000000010 //beq $7,$0,2

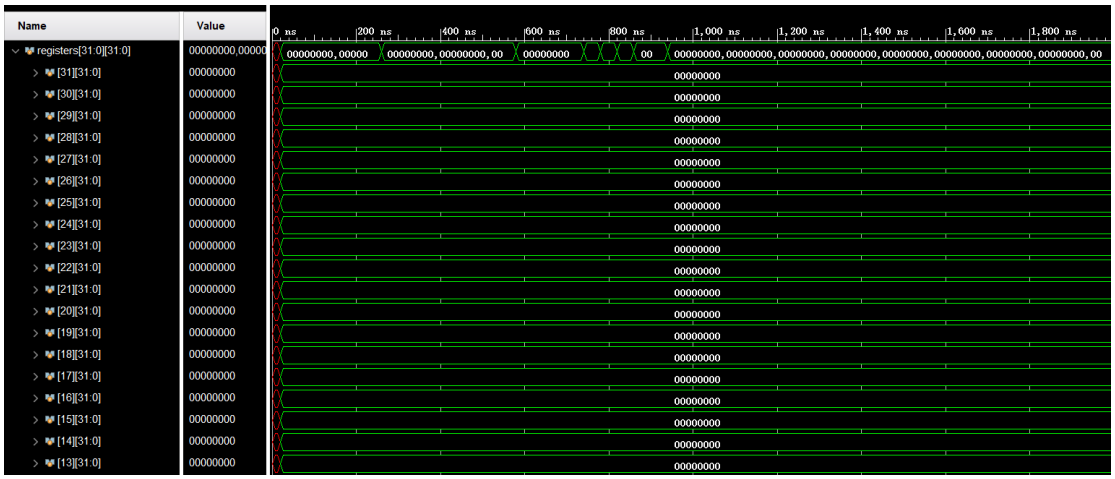
000000 00100 01000 11111 00000 100000 //add $31,$4,$8

000000 00101 00110 11110 00000 100000 //add $30,$5,$6

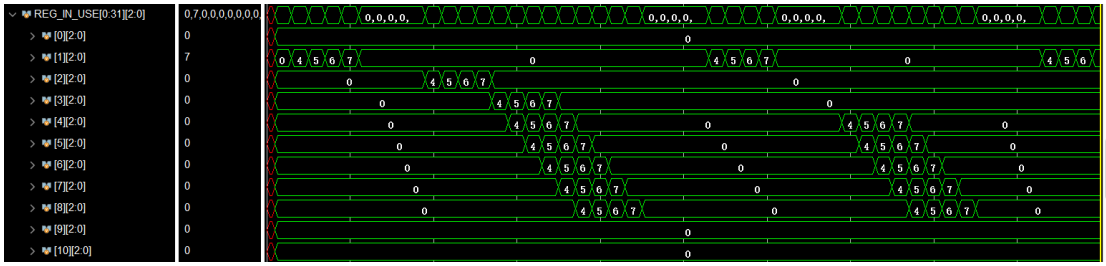
000010 0000000000 0000 0000 0000 0000　　//j 0
若正常运行，则每个循环结束时结果如下（值为 16 进制表示）：
$1 5
$2 1
$3 6
$4 4
$5 1
$6 5
$7 0
$8 1
其余寄存器均为 0。

实验结果

仿真波形如下所示（由于软件显示问题，显示的值可能并非完整值）。
寄存器：



寄存器对应的计数器（值非 0 表示会产生冲突）：



通过分析相应仿真波形，结果正确。

总结

通过加入在 IF（取指令）阶段自动加入 nop 防止了冲突，确保了程序运行结果的正确性，然而也会产生一些问题。

最突出的问题在于，IF 阶段对指令进行了预分析，会使得此阶段过于臃肿，进而导致流水线各阶段用时差距大，难以保证效率，然而，考虑到即使在加入这些逻辑后，IF 相对于其他阶段，尤其是 EX 和 MEM，仍然不会过于庞大，因此可能带来的效率影响不会太过严重。也可以考虑利用 IF 阶段的指令预分析结果，减少 ID 阶段的工作量。

由于条件限制，在实验中没有加入对于计数器的具体实现，因此看似大量的加法会成为 IF 的负担，然而在实际情况下，计数器的硬件实现可以通过 D 触发器的级联实现，也不会造成太大的效率影响。

通过 bypassing、forwarding、分支预测等方式，可以减少冲突时的等待时间，或者通过由编译器调整指令的顺序，尽量避免或消除等待。