**191220154 张涵之 第5章作业**

1. 每次按同步方式从主存读取一个字，则不需要使用WMFC控制信号。
2. CALL指令功能：设置PC为子程序起始地址（从而跳转执行子程序）并将CALL指令后一条指令的首地址保存到栈中（便于子程序执行完毕后返回）。
3. 读取并执行CALL指令分为取指令译码、设置PC和保存返回地址三个阶段：
   1. PCout, MOVb, MARin // 通过总线B将PC的内容送到MAR的输入端
   2. read, PCout, b + 1, PCin // 取指令操作码，并设置PC = PC + 1
   3. MDRout, MOVb, IRin // 将MDR的内容通过总线B送到指令寄存器IR
   4. PCout, MOVb, MARin // 通过总线B将新PC的内容送到MAR的输入端
   5. read, PCout, b + 1, Yin // 取子程序地址，将下条指令地址送到寄存器Y
   6. MDRout, MOVb, PCin // 将MDR的内容通过总线B送到PC
   7. SPout, MOVb, MARin // 将栈指示器SP的内容送到MAR的输入端
   8. Yout, MOVb, MDRin // 将寄存器Y的内容送到MDR的输入端
   9. write, SPout, b + 1, SPin // 保存返回地址到栈顶，并更新SP = SP + 1

以需要时间最长的read/write为时间周期设置标准，至少需要9个时钟周期。

1. 写出下列指令在指令执行阶段的控制信号序列，并说明需要几个时钟周期。
2. R[R1] ← R[R1] + imm16：不需要访问存储器，共需要3个时钟周期

MDRout, Yin // 将MDR的内容（即imm16）送到寄存器Y

R1out, add, (Zin) // 在ALU中R1与Y的内容相加，输出送到寄存器Z

Zout, R1in // 将寄存器Z中的内容送到R1

1. R[R1] ← R[R1] + M[imm16]：访问一次存储器，共需要5个时钟周期

MDRout, MARin // 将MDR的内容（imm16）送到MAR

read1, R1out, Yin // 取M[imm16]到MDR，R1的内容送到Y

read2 // 等待存储器访问操作完成

MDRout, add, (Zin) // 在ALU中MDR与Y的内容相加，输出送到Z

Zout, R1in // 将寄存器Z中的内容送到R1

1. R[R1] ← R[R1] + M[M[imm16]]：访问两次存储器，共需要8个时钟周期

MDRout, MARin // 将MDR的内容（imm16）送到MAR

read1 // 取M[imm16]到MDR

read2 // 等待存储器访问操作完成

MDRout, MARin // 再将MDR的内容（M[imm16]）送到MAR

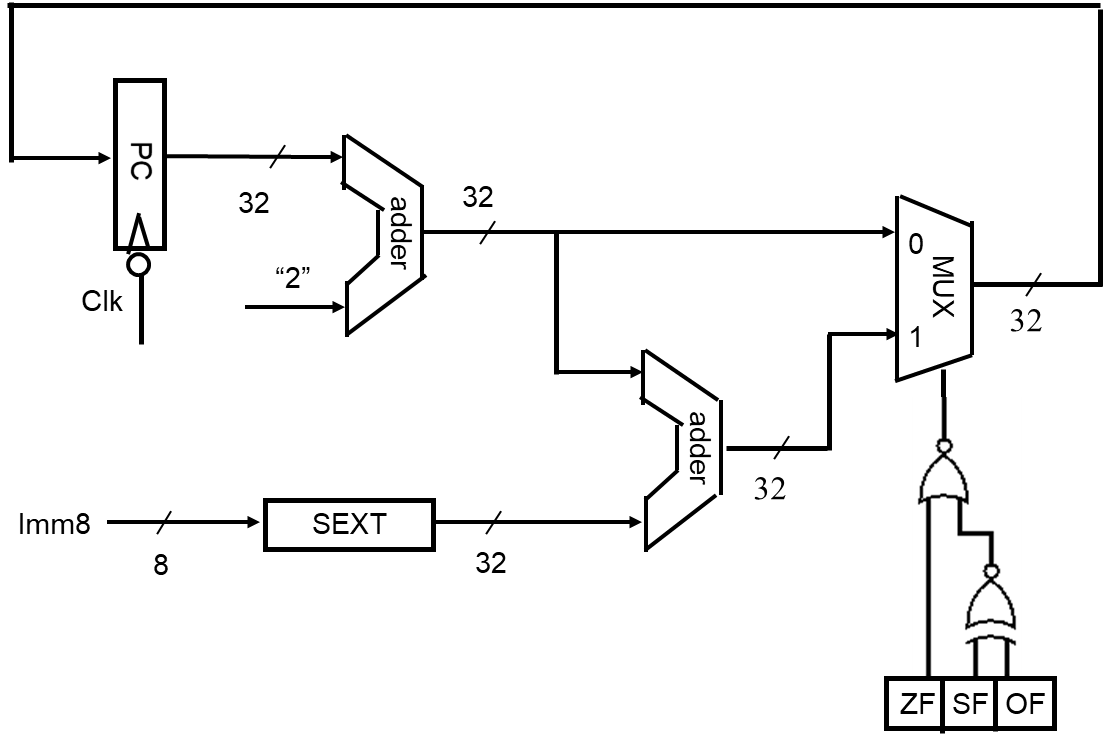
read1, R1out, Yin // 取M[M[imm16]]到MDR，R1的内容送到Y

read2 // 等待存储器访问操作完成

MDRout, add, (Zin) // 在ALU中MDR与Y的内容相加，输出送到Z

Zout, R1in // 将寄存器Z中的内容送到R1

1. 完成如下要求并回答问题：
2. 该计算机采用双字节定长指令字，当bgt指令条件不满足时，通过PC = PC + 2实现正常的顺序执行，则下一条指令的相对偏移量为2，即存储器按字节编址。
3. 如下图为实现bgt指令的数据通路。



1. RegWr = 0：无法写目的寄存器，所有R型、I型、U型和J型指令不能正确执行。

ALUASrc = 0：总是选busA作为ALU的A口操作数，J型指令不能正确执行。

Branch = 0：只有B型指令不能正确执行。

Jump = 0：只有J型指令不能正确执行。

MemWr = 0：总是不写信息到数据存储器，S型（Store）指令不能正确执行。

MemtoReg = 0：总是选ALU的输出送目的寄存器，Load指令不能正确执行。

1. RegWr = 1：总是写目的寄存器，S型（Store）和B型指令不能正确执行。

ALUASrc = 1：总是选PC作为A口操作数，除了J型指令和lui都不能正确执行。

Branch = 1：除了B型指令都不能正确执行。

Jump = 1：除了J型指令都不能正确执行。

MemWr = 1：总是写信息到数据存储器，除了S型（Store）指令都不能正确执行。

MemtoReg = 1：总是选数据存储器的输出送目的寄存器，除了S型和B型不写入目的寄存器，以及Load指令确实取的是数据存储器输出，其他指令都不能正确执行。

1. 若要在RV32I指令集中增加一条swap指令（功能是交换两个寄存器的内容）。
2. 要用伪指令方式实现“swap rs, rt”的指令序列，且不使用其他额外寄存器以免破坏这些寄存器的值，则可以用三次异或操作来实现两个寄存器内容的互换。

xor rs, rs, rt

xor rt, rs, rt

xor rs, rs, rt

1. 设swap指令在程序中所占比例为x（0 ≤ x ≤ 1），则采用伪指令方式（软件方式）增加swap指令后，程序运行所需时间为原先的1 – x + 3x = 1 + 2x倍；若采用硬件方式实现，所需时间为原先的1.1倍，令1 + 2x > 1.1，解得x > 0.05，即swap指令要在程序中占比超过5%，值得用硬件方式实现才比软件方式更经济实惠。
2. 在单周期数据通路中，一条指令必须在一个时钟周期内完成，且一个时钟周期内只能写一次寄存器。由于swap指令的功能是交换两个寄存器的内容，则至少写两次寄存器，不能在一个时钟周期内完成，也就无法在不对通用寄存器组进行修改（以支持同时写两个以上的寄存器）的情况下，直接采用硬件方式实现。

在多周期数据通路中，一个指令可通过多个时钟周期来完成，因此也可以写多次寄存器，即使不对通用寄存器组进行修改，也可以直接采用硬件方式实现。

1. PcWr = 0：取指令阶段无法将下一条指令地址送PC，则所有指令都不能正确执行。

MARWr = 0：无法写入Load/Store指令的地址，则Load/Store指令不能正确执行。

RegWr = 0：无法写入寄存器，则R型指令、I型运算指令、Load指令不能正确执行。

BMUX = 0：无法满足RExec状态的要求，R型指令不能正确执行。

PCout = 0：无法取指令，则所有指令都不能正确执行。

1. PcWr = 1：每个时钟都不停地更新PC，则所有的指令执行都会失去控制。

MARWr = 1：每个时钟都更新MAR，但除了Load/Store指令都不会送入总线；而对于Load/Store指令，MAR的内容将在“投机”计算指令地址暂存的下一个周期，即lwExec和swExec状态下被送入总线，此时MAR还没来得及被更新。由于lwFinish状态不需要读MAR，Load仍能正确执行；swFinish状态需要MAR，则Store不能正确执行。

RegWr = 1：总是写目的寄存器，S型（Store）和B型指令不能正确执行。

BMUX = 1：无法满足IExec状态的要求，I型指令不能正确执行。

PCout = 1：无法满足lwExec、SwExec的要求，Load/Store指令都不能正确执行。

1. 假定在一个5级流水线（图5.43）处理器中：回答以下问题。
2. 执行阶段EX所用的ALU操作时间缩短20%不能加快流水线的执行速度。因为根据流水线设计的原则，流水段的长度以最复杂的操作（此处为存储单元, 200ps）为准，ALU操作时间缩短20%，流水段的长度仍为200ps，执行速度不变。
3. 若ALU操作时间增加20%，对流水线的性能没有影响。因为ALU的操作时间原本为150ps，增加20%为180ps，仍小于200ps，流水段长度不变，执行速度不变。
4. 若ALU操作时间增加40%，流水线的性能会变差，因为ALU的操作时间增进40%为210ps，大于200ps，流水段长度也应增加到210ps，执行耗时增加了5%。
5. A: 80ps，B: 30ps，C: 60ps，D: 50ps，E: 70ps，F: 10ps，寄存器延迟：20ps。
6. 插入1个流水段寄存器，得到一个2级流水线：

在C和D之间插入，时钟周期取max(80 + 30 + 60, 50 + 70 + 10) + 20= 190ps，指令吞吐率为1/190ps = 5.26G条/秒，指令执行时间为2 × 190 = 380ps

1. 插入2个流水段寄存器，得到一个3级流水线：

分别在B和C、D和E之间插入，时钟周期取max(80 + 30, 60 + 50, 70 + 10) + 20 = 130ps，指令吞吐率1/130ps = 7.69G条/秒，指令执行时间为3 × 130 = 390ps

1. 插入3个流水段寄存器，得到一个4级流水线：

分别在A和B、C和D、D和E之间插入，时钟周期取max(80, 30 + 60, 50, 70 + 10) + 20 = 110ps，指令吞吐率1/110ps = 9.09G条/秒，指令执行时间为4 × 110 = 440ps

1. 吞吐量最大的流水线：

各部分延迟中最大的为80ps，则时钟周期最小为80 + 20 = 100ps，观察相邻的各部分中，E和F合并后仍不超过80ps，则插入4个流水段寄存器，分别在A和B、B和C、C和D、D和E之间，得到一个5级流水线。此时时钟周期取100ps，指令吞吐率1/100ps = 10G条/秒，指令执行时间为5 × 100 = 500ps

1. 发生数据相关的指令对是1-2（s3）、2-3（t2）、2-4（t2）以及3-4（t1）。如果不采用“转发”技术，需要在分别在第2、第3、第4条指令前各插入3条nop指令才能避免数据冒险。如果采用“转发技术”，则第3和第4条指令之间的数据冒险不能完全解决，仍需要在第4条指令前加入1条nop指令，才能使这段RV32I程序不发生数据冒险。
2. RV32I指令序列在图5.43所示的流水线数据通路中执行：回答以下问题。
3. 第1条指令在EX阶段结束时已经得到s3的新值，可以直接从EX/M流水段寄存器取出送到ALU的输入端，这样在第二条指令执行ALU时用的就是s3的新值；

第2条指令在EX阶段结束时已经得到t2的新值，可以直接从EX/M流水段寄存器取出送到ALU的输入端，供第3条指令执行ALU时使用；

同样，第4条指令在ALU中用到的t2可以直接从M/WB流水段寄存器获得；

第3条指令在M段结束时得到t1（准备WB段存入）的新值，从M/WB寄存器转发到第4条指令的ALU输入端（但是来不及到达，发生Load-use数据冒险）。

1. 如上，第3和第4条指令之间存在Load-use数据冒险。
2. 假设处理器提供对转发技术和Load-use数据冒险的支持，第5周期结束时：

第1条指令：处于WB阶段，寄存器s3将被写入数据。

第2条指令：处于M阶段，对于sub指令来说，此时什么也不用做。

第3条指令：处于EX阶段，正在从EX/M流水段寄存器读取被转发的t2新值。

第4条指令：处于ID阶段，从t1和t2寄存器读取数据。考虑到检测出Load-use冒险，需要插入气泡来作阻塞处理，还可能正在将控制信号/指令/PC写使能清0。

第5条指令：处于IF阶段，正在从PC取指令并计算PC + 4，但由于在此之前已经发生了Load-use冒险，PC可能将不会被写入PC + 4的值，而是维持不变。

1. 假定有一个程序的指令序列为“lw，add，lw，add，…”。回答以下问题。
2. 每条lw和紧跟其后的add之间都发生Load-use冒险，则每个lw-add指令对中间都需要插入一个气泡（阻塞一个时钟周期），则CPI为3/2 = 1.5
3. 寄存器写口和读口分别在一个时钟周期的前、后半个周期内独立工作。每两条相邻指令之间有数据依赖，都需要插入两个气泡（阻塞两个时钟周期），CPI为3。
4. 流水线带转发，则只需要解决1-2、6-7两个指令对的Load-use数据冒险即可。考虑从选择两条执行顺序不影响的指令，分别插入第1和2、6和7条指令之间，如下：

1 lw s2, 100 (s6)

**4 add s6, s4, s7**

2 add s2, s2, s3

3 lw s3, 200 (s7)

6 lw s2, 300 (s8)

**5 sub s3, s4, s6**

7 beq s2, s8, Loop

将与第2和第3条指令无关的第4条指令插入到第1和第2条指令之间，与第6条指令无关的第5条指令插入到第6和第7条之间，则调整后的指令序列全都可以顺利进行转发，而不会出现Load-use数据冒险，执行时的性能达到最好。

1. 检测相减结果是否为0的操作在执行阶段进行，则分支延迟损失时间片（分支延迟槽）为C = 2。考虑数据转发，lw和bne指令之间发生Load-use数据冒险，需要阻塞1个时钟周期；bne和add指令之间（当转移条件满足时）发生控制冒险，需要阻塞2个时钟周期；j指令的阻塞时间与何时PC更新操作有关，若在译码/取数阶段更新PC，则需要阻塞1个时钟周期，否则若在执行阶段更新PC，需要阻塞2个时钟周期。
2. 通过分别计算三种实现方式下每条指令的（平均）执行时间来比较快慢。
3. 单周期方式：时钟周期取最复杂指令lw的指令周期，为PC锁存延迟 + 取指令时间 + 寄存器取数时间 + ALU延迟 + 存储器取数时间 + 寄存器建立时间 + 时钟偏移，PC锁存延迟和时钟偏移忽略不计，得200 + 50 + 100 + 200 + 50 = 600ps。单周期方式下每条指令在这个时钟周期内完成，则每条指令的执行时间为600ps
4. 多周期方式：CPI = 7 × 25% + 6 × 10% + 5 × 52% + 4 × 11% + 4 × 2% = 5.47，又时钟周期取存储操作时间的一半即100ps，每条指令的平均执行时间为547ps
5. 流水线方式：下面先依次分析每一类指令的CPI。

对于取数指令，若其后第一条指令与之存在依赖关系需要阻塞两个时钟周期，若其后第一条指令不存在而第二条存在依赖关系，需要阻塞一个时钟周期，其他的情况均不需要阻塞。则CPI = 1/2 × 3 + (1 – 1/2) × 1/4 × 2 + 3/8 × 1 = 17/8 = 2.125

\* 我理解这个“Load指令与后续各指令之间存在依赖关系的概率分别为1/2、1/4、1/8……”中的1/2、1/4……是各自独立的，所以这里2乘的是“取数指令后第一条指令不依赖而第二条依赖”的概率，3/8是后面两条都不依赖的概率。

对于存数指令均不需要阻塞，CPI = 1；

对于ALU指令采用“转发”技术处理数据冒险，也不需要阻塞，CPI = 1；

对于分支指令，预测准确率为75%，分支延迟损失片为2，即预测失败时需要阻塞两个时钟周期，则CPI = 75% × 1 + (1 – 75%) × 3 = 1.5；

对于跳转指令最早在译码阶段确定跳转地址，需要阻塞两个时钟周期，CPI = 3；

综上，CPI = 2.125 × 25% + 1 × 10% + 1 × 52% + 1.5 × 11% + 3 × 2% = 1.37625，又时钟周期取存储操作时间的一半即100ps，每条指令的平均执行时间为138ps

则三种实现方式流水线最快，多周期次之，单周期最慢，其中多周期其实只比单周期略快（10%不到），而流水线方式执行速度分别是单周期的4.35和多周期的3.96倍。

1. 分析并给出以下急着预测方案的预测准确率。
2. 静态预测，总是预测转移（taken）。预测准确率为15 / 25 = 60%

* 分支指令1：预测正确3次，错误0次；
* 分支指令2：预测正确0次，错误4次；
* 分支指令3：预测正确3次，错误3次；
* 分支指令4：预测正确4次，错误1次；
* 分支指令5：预测正确5次，错误2次。

1. 静态预测，总是预测不转移（not taken）。预测准确率为10 / 25 = 40%

* 分支指令1：预测正确0次，错误3次；
* 分支指令2：预测正确4次，错误0次；
* 分支指令3：预测正确3次，错误3次；
* 分支指令4：预测正确1次，错误4次；
* 分支指令5：预测正确2次，错误5次。

1. 一位动态预测，初始预测转移（taken）。预测准确率为13 / 25 = 52%

* 分支指令1：预测T – T – T，正确3次，错误0次；
* 分支指令2：预测T – N – N – N，正确3次，错误1次；
* 分支指令3：预测T – T – N – T – N – T，正确1次，错误5次；
* 分支指令4：预测T – T – T – T – N，正确3次，错误2次；
* 分支指令5：预测T – T – T – N – T – T – N，正确3次，错误4次。

1. 两位动态预测，初始预测弱转移（taken）。预测准确率为18 / 25 = 72%

\* 以下wT/wN (weak)表示弱转移/不转移，sT/sN (strong)表示强转移/不转移。

* 分支指令1：预测wT – sT – sT，正确3次，错误0次；
* 分支指令2：预测wT – wN – sN – sN，正确3次，错误1次；
* 分支指令3：预测wT – sT – wT – sT – wT – ST，正确3次，错误3次；
* 分支指令4：预测wT – sT – sT – sT – wT，正确4次，错误1次；
* 分支指令5：预测wT – sT – sT – wT – sT – sT – wT，正确5次，错误2次。