《计算机系统结构》作业

岳锦鹏

2024 年 3 月 17 日——2024 年 6 月 18 日

目录

第一章　计算机抽象及相关技术

1.3 [2]<1.3> 请描述高级语言 (例如 C) 编写的程序转化为能够直接在计算机处理器上执行的表示的具体步骤。

高级语言通过编译器（编译型的高级语言，如 C、C++）或解释器（解释型的高级语言，如 Python、JavaScript）转化成指令，再转化成二进制的机器码，或者直接转化成二进制的机器码（不同的硬件对应的机器码可能不同），机器码就是能够直接在计算机处理器上执行的表示。

1.5 [4]<1.6> 有 3 种不同的处理器 P1、P2 和 P3 执行同样的指令系统。P1 的时钟频率为 3GHz,CPI 为 1.5;P2 的时钟频率为 2.5GHz，CPI 为 1.0;P3 的时钟频率为 4GHz,CPI 为 2.2。

a. 以每秒执行的指令数为标准, 哪个处理器性能最高?

P1: $\frac{3\text{GHz}}{1.5\text{时钟周期数/指令}} = 2 \times 10^9$
P2: $\frac{2.5\text{GHz}}{1.0\text{时钟周期数/指令}} = 2.5 \times 10^9$
P3: $\frac{4\text{GHz}}{2.2\text{时钟周期数/指令}} \approx 1.818 \times 10^9$
所以处理器 P2 性能最高。

b. 如果每个处理器执行一个程序都花费 10 秒时间，求它们的时钟周期数和指令数。

	时钟周期数	指令数
P1	$3\text{GHz} \times 10s = 30 \times 10^9$	$\frac{30\times10^9}{1.5} = 20 \times 10^9$
P2	$2.5\text{GHz} \times 10s = 25 \times 10^9$	$\frac{25\times10^9}{1.0} = 25 \times 10^9$
P3	$4\text{GHz} \times 10s = 40 \times 10^9$	$\frac{40\times10^9}{2.2} \approx 18.1818 \times 10^9$

c. 我们试图把执行时间减少 30%，但这会引起 CPI 增大 20%。请问为达到时间减少 30% 的目标，时钟频率应达到多少?

$$\text{执行时间} \times 70\% = \text{指令数} \times (\text{CPI} \times 120\%)/\text{时钟频率}$$
$$\text{时钟频率} = \frac{\text{指令数} \times \text{CPI}}{\text{执行时间}} \times \frac{1.2}{0.7} \approx 1.714 \times \text{原始时钟频率}$$
所以时钟频率要达到原始时钟频率的大约 1.714 倍。

1.6 [20]<1.6> 同一个指令系统体系结构有两种不同的实现方式。根据 CPI 的不同将指令分成四类 (A、B、C 和 D)。P1 的时钟频率为 2.5GHz，CPI 分别为 1、2、3 和 3;P2 时钟频率为 3GHz，CPI 分别为 2、2、2 和 2。
给定一个程序，有 1.0×10^6 条动态指令，按如下比例分为 4 类:A，10%;B，20%;C，50%;D，20%。

a. 每种实现方式下的整体 CPI 是多少?

P1：$1 \times 10\% + 2 \times 20\% + 3 \times 50\% + 3 \times 20\% = 2.6$
P2：2

b. 计算两种情况下的时钟周期总数。

P1：$1.0 \times 10^6 \times 2.6 = 2.6 \times 10^6$
P2：$1.0 \times 10^6 \times 2 = 2 \times 10^6$

1.7 [15]<1.6> 编译器对应用程序性能有极深的影响。假定对于一个程序, 如果采用编译器 A，则动态指令数为 1.0×10^9，执行时间为 1.1s; 如果采用编译器 B，则动态指令数为 1.2×10^9，执行时间为 1.5s。

a. 在给定处理器时钟周期长度为 1ns 时，求每个程序的平均 CPI。

编译器 A：$\frac{1.1s}{1\times10^{-9}s}/1.0 \times 10^9 = 1.1$
编译器 B：$\frac{1.5s}{1\times10^{-9}s}/1.2 \times 10^9 = 1.25$

b. 假定被编译的程序分别在两个不同的处理器上运行。如果这两个处理器的执行时间相同，求运行编译器 A 生成之代码的处理器时钟比运行编译器 B 生成之代码的处理器时钟快多少。

$$\text{执行时间}_A = \text{执行时间}_B$$
$$\frac{\text{指令数}_A \times CPI_A}{\text{时钟频率}_A} = \frac{\text{指令数}_B \times CPI_B}{\text{时钟频率}_B}$$
$$\frac{\text{时钟频率}_A}{\text{时钟频率}_B} = \frac{\text{指令数}_A \times CPI_A}{\text{指令数}_B \times CPI_B} = \frac{1.0 \times 10^9 \times 1.1}{1.2 \times 10^9 \times 1.25} = \frac{11}{15} = 0.733$$

所以题意应该是错了，运行编译器 A 生成之代码的处理器时钟比运行编译器 B 生成之代码的处理器时钟慢，约 0.733 倍。

c. 假设开发了一种新的编译器，只需 6.0×10^8 条指令，程序平均 CPI 为 1.1。求这种新的编译器在原处理器环境下相对于原编译器 A 和 B 的加速比。

记这个新的编译器为 C。

$$\frac{\text{性能}C}{\text{性能}A}=\frac{\text{执行时间}A}{\text{执行时间}C}=\frac{\frac{\text{指令数}_A\times CPI_A}{\text{时钟频率}}}{\frac{\text{指令数}_C\times CPI_C}{\text{时钟频率}}}=\frac{\text{指令数}_A\times CPI_A}{\text{指令数}_C\times CPI_C}=\frac{1.0\times10^9\times1.1}{6.0\times10^8\times1.1}=\frac{5}{3}\approx1.667$$

同理，

$$\frac{\text{性能}C}{\text{性能}B}=\frac{\text{指令数}_B\times CPI_B}{\text{指令数}_C\times CPI_C}=\frac{1.2\times10^9\times1.25}{6.0\times10^8\times1.1}=\frac{25}{11}\approx2.273$$

所以这种新的编译器在原处理器环境下相对于原编译器 A 和 B 的加速比分别约为 1.667 和 2.273。

1.9 在某处理器中，假定算术指令、load/store 指令和分支指令的 CPI 分别是 1、12 和 5。同时假定某程序在单个处理器核上运行时需要执行 2.56×10^9 条算术指令、1.28×10^9 条 load/store 指令和 2.56×10^8 条分支指令，并假定处理器的时钟频率为 2GHz。现假定程序并行运行在多核上，分配到每个处理器核上运行的算术指令和 load/store 指令数目为单核情况下相应指令数目除以 $0.7\times p$（p 为处理器核数），而每个处理器的分支指令的数量保持不变。

1.9.1 [5]<1.7> 求出当该程序分别运行在 1、2、4 和 8 个处理器核上的执行时间，并求出其他情况下相对于单核处理器的加速比。

$$\text{多核执行时间}=\frac{\text{指令数}\times CPI}{\text{时钟周期}}=\frac{\frac{2.56\times10^9}{0.7\times p}\times1+\frac{1.28\times10^9}{0.7\times p}\times12+2.56\times10^8\times5}{2\times10^9}=\frac{16(p+20)}{25p}s$$

$$\text{单核执行时间}=\frac{\text{指令数}\times CPI}{\text{时钟周期}}=\frac{2.56\times10^9\times1+1.28\times10^9\times12+2.56\times10^8\times5}{2\times10^9}=\frac{48}{5}=9.6s$$

$$\text{多核与单核的加速比}=\frac{\text{单核执行时间}}{\text{多核执行时间}}=\frac{\frac{48}{5}}{\frac{16(p+20)}{25p}}=\frac{15p}{p+20}$$

处理器核数	执行时间	加速比
1	9.600	1.000
2	7.040	1.364
4	3.840	2.500
8	2.240	4.286

1.9.2 [10]<1.6,1.8> 如果算术指令的 CPI 加倍，对分别运行在 1、2、4 和 8 个处理器核上的执行时间有何影响?

$$\text{多核执行时间}=\frac{\frac{2.56\times10^9}{0.7\times p}\times2+\frac{1.28\times10^9}{0.7\times p}\times12+2.56\times10^8\times5}{2\times10^9}=\frac{16(7p+160)}{175p}s$$

$$\text{单核执行时间}=\frac{2.56\times10^9\times2+1.28\times10^9\times12+2.56\times10^8\times5}{2\times10^9}=\frac{272}{25}=10.88s$$

处理器核数	之前的执行时间	现在的执行时间
1	9.600	10.880
2	7.040	7.954
4	3.840	4.297
8	2.240	2.469

1.9.3 [10]<1.6,1.8> 如果要使单核处理器的性能与四核处理器相当，单处理器中 load/store 指令的 CPI 应该降低多少? 此处假定四核处理器的 CPI 保持原数值不变。

设单处理器中 load/store 指令的 CPI 应该降低到原 CPI 的 x 倍。

$$\text{单核处理器（}CPI\text{降低）的执行时间} = \text{四核处理器（原}CPI\text{）的执行时间}$$

$$\frac{2.56\times10^9\times1+1.28\times10^9\times12x+2.56\times10^8\times5}{2\times10^9}=\frac{16\times(4+20)}{25\times4}$$

$$\frac{192x}{25}+\frac{48}{25}=\frac{96}{25}$$

$$x=\frac{96-48}{192}=\frac{1}{4}=0.25$$

所以单处理器中 load/store 指令的 CPI 应该降低到原 CPI 的 0.25 倍，即 $12\times0.25=3$。

第二章　指令：计算机的语言

2.3 [5]<2.2,2.3> 对于以下 C 语句，请编写相应的 RISC-V 汇编代码。假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
B[8] = A[i-j];
```

这里假设数组 A 和 B 中的元素都是 64 位的，即“双字”，即 8 个字节。

```
sub     x30, x28, x29   // i-j，从 A 开始的双字偏移
slli    x30, x30, 3     // (i-j)*8，从 A 开始的字节偏移
add     x30, x30, x10   // A[i-j] 的地址
ld      x31, 0(x30)     // 加载 A[i-j]
sd      x31, 64(x11)    // 放到 B[8]，8 是双字偏移，转化成字节偏移就是 64
```

验证一下

使用命令 `riscv64-unknown-linux-gnu-gcc -S main.c -o main.s` 编译，左边是 C 代码，右边是汇编代码。

```
...
long long *A, *B, i,j;
B[8] = A[i - j];
...
```

```
...
ld      a4,-24(s0)  // 加载 i
ld      a5,-32(s0)  // 加载 j
sub     a5,a4,a5
slli    a5,a5,3
ld      a4,-40(s0)  // 加载 A
add     a4,a4,a5
ld      a5,-48(s0)  // 加载 B
addi    a5,a5,64
ld      a4,0(a4)
sd      a4,0(a5)
...
```

比较后可以发现基本差不多，但是最后一步赋值时这里先用了 addi 偏移 64 位，再存储，为什么不是直接 64(a5) 一步存储到从 a5 偏移 64 个字节的位置呢？

2.4 [10]<2.2,2.3> 对于以下 RISC-V 汇编指令，相应的 C 语句是什么？假设变量 f、g、h、i 和 j 分别分配给寄存器 x5、x6、x7、x28 和 x29。假设数组 A 和 B 的基地址分别在寄存器 x10 和 x11 中。

```
slli    x30, x5, 3      // x30 = f*8
add     x30, x10, x30   // x30 = &A[f]
slli    x31, x6, 3      // x31 = g*8
add     x31, x11, x31   // x31 = &B[g]
ld      x5, 0(x30)      // f = A[f]

addi    x12, x30, 8     // x12 = (&f + 1) = (原始的 f)&A[f + 1]
ld      x30, 0(x12)     // x30 = A[f + 1]
add     x30, x30, x5    // x30 = f + A[f + 1] = (原始的 f)A[f] + A[f + 1]
sd      x30, 0(x31)     // B[g] = A[f] + A[f + 1]
```

注释已写在上方，因此此汇编相应的 C 语句如下：

```
B[g] = A[f] + A[f + 1];
```

2.10 假设寄存器 x5 和 x6 分别保存值 0x8000000000000000 和 0xD000000000000000。

2.10.1 [5]<2.4> 以下汇编代码中 x30 的值是多少?

```
add x30, x5, x6
```

由于后面全是 0，只需要考虑最高 4 位二进制。x5 为 8_{16} 即 1000_2，x6 为 D_{16} 即 1101_2，要注意最高位为 1 代表负数，由于负数已按照补码方式表示，所以可以直接相加。这里两个数都是负数，相加后为 $1\ 0101_2$，进位忽略，即 $0101_2 = 5_{16}$，此时 x30 的值为

0x5000000000000000

2.10.2 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

两个负数相加，结果的最高位为 0，表示正数，不是预期结果，即产生了溢出。

2.10.3 [5]<2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少?

```
sub x30, x5, x6
```

还是只考虑最高 4 位二进制，最高位为 1 代表负数，直接相减不方便，可以先按照补码方式转换为相反数再相加，按照补码方式 1101_2 的相反数为 0011，因此 $1000_2 - 1101_2 = 1000_2 + 0011_2 = 1011_2$，即 B_{16}。因此 x30 的值为

0xB000000000000000

2.10.4 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

两个负数相减，结果仍为负数，最高位是 1，为预期结果，无溢出。

2.10.5 [5]<2.4> 对于上面指定的寄存器 x5 和 x6 的内容，以下汇编代码中 x30 的值是多少?

```
add x30, x5, x6
add x30, x30, x5
```

第一行就是 2.10.1 的指令，之后执行第二行，按照最高位来看就是 $0101_2+1000_2=1101_2$，即 D_{16}，所以 x30 的值为

0xD000000000000000

2.10.6 [5]<2.4> x30 中的结果是否为预期结果，或者是否溢出?

结果的最高位为 1，虽然仍是负数但显然不是 x5 + x6 + x5 的结果，所以不是预期结果，产生了溢出。

2.27 [5]<2.7> 将以下循环转换为 C 代码。假设 C 语言级的整数 i 保存在寄存器 x5 中, x6 保存名为 result 的 C 语言级的整数，x10 保存整数 MemArray 的基址。

```
        addi    x6, x0, 0       // result = 0
        addi    x29, x0, 100    // x29 = 100
LOOP:   ld      x7, 0(x10)      // x7 = *MemArray
        add     x5, x5, x7      // i = i + *MemArray
        addi    x10, x10, 8     // MemArray++
        addi    x6, x6, 1       // result++
        blt     x6, x29, LOOP   // (result < 100)
```

注释已写在上方。根据题意和命名来看，MemArray 应该是整数数组而不是整数吧，不然总不能 (&MemArray)++ 吧。这里假设整数都是 64 位的，所以 MemArray++ 对应的汇编代码是增加 8 个字节，即 addi x10, x10, 8 。

根据注释分析出的结果，可以得到 C 代码:

```
for (int result = 0; result < 100; result++) {
    i += *(MemArray++);
}
```

为什么这里是 result 作为循环变量而 i 作为结果啊??? 但按照题意分析出来就是这样。

2.31 [20]<2.8> 将函数 f 转换为 RISC-V 汇编语言。假设 g 的函数声明是 `int g(int a,int b)`。函数 f 的代码如下:

```
int f(int a, int b, int c, int d) {
    return g(g(a,b), c+d);
}
```

注意:

(1) 调用函数时参数应该是从右往左计算的，但是需要保存的局部变量是从左到右保存的;

(2) 栈指针应该是 16 字节（四字）对齐的;

(3) 返回地址应该是调用者保存的。

```
f:                          // 保存自己要用到的保存寄存器，好像没有
                            // 此时 a,b,c,d 分别保存在 x10, x11, x12, x13 中
    addw    x5, x12, x13    // 计算 c+d, 4 字节整数的要加 w
    addi    sp, sp, -16     // 移动栈指针，栈指针应该是 16 字节对齐的？
    sd      x1, 8(sp)       // 保存 x1，返回地址
    sd      x5, 0(sp)       // 保存 x5，这里 int 类型 4 个字节，由于对齐产生了空位
                            // a 和 b 在调用 g(a,b) 后不会用到，所以不用保存
                            // *** 开始计算 g(a,b)
                            // a 和 b 已经在 x10 和 x11 中所以不需要移动
    jal     x1, g           // 跳转到 g
                            // g 的返回值在 x10 中，正好是下一次调用的第一个参数
    ld      x11, 0(sp)      // 恢复 x5，直接恢复到 x11 上避免再移动
                            // x1 不需要着急恢复，马上就是下一次调用了
                            // 只恢复了 x5，栈指针要 16 字节对齐不能移动
                            // *** 开始计算 g(g(a,b), c+d)
                            // 第二个参数已从 x5 恢复
                            // 第一个参数已经在 x10 中
    jal     x1, g           // 跳转到 g
                            // g 的返回值在 x10 中，不需要移动
    ld      x1, 0(sp)       // 恢复 x1
    addi    sp, sp, 16      // 恢复栈指针
                            // 恢复保存的保存寄存器，好像没有
    jalr    x0, x1          // 返回
```

验证一下

使用命令 `riscv64-unknown-linux-gnu-gcc -S main.c -o main.s` 编译。

```
f:
.LFB0:
```

```
        .cfi_startproc
        addi    sp,sp,-32
        .cfi_def_cfa_offset 32
        sd      ra,24(sp)
        sd      s0,16(sp)
        .cfi_offset 1, -8
        .cfi_offset 8, -16
        addi    s0,sp,32
        .cfi_def_cfa 8, 0
        mv      a5,a0
        mv      a4,a3
        sw      a5,-20(s0)
        mv      a5,a1
        sw      a5,-24(s0)
        mv      a5,a2
        sw      a5,-28(s0)
        mv      a5,a4
        sw      a5,-32(s0)
        lw      a4,-24(s0)
        lw      a5,-20(s0)
        mv      a1,a4
        mv      a0,a5
        call    g
        mv      a5,a0
        mv      a3,a5
        lw      a5,-28(s0)
        mv      a4,a5
        lw      a5,-32(s0)
        addw    a5,a4,a5
        sext.w  a5,a5
        mv      a1,a5
        mv      a0,a3
        call    g
        mv      a5,a0
        mv      a0,a5
        ld      ra,24(sp)
        .cfi_restore 1
        ld      s0,16(sp)
        .cfi_restore 8
        .cfi_def_cfa 2, 32
        addi    sp,sp,32
        .cfi_def_cfa_offset 0
        jr      ra
        .cfi_endproc
```

第三章　计算机的算术运算

3.7 [5]<3.2> 假设带符号的 8 位十进制整数 185 和 122 以符号-数值形式存储。计算 185+122。是否上溢或下溢，或都没有?

185 转成 8 位二进制为 1011 1001，其最高位为 1，表示负数，即 $185-256=-71$；122 转成 8 位二进制为 0111 1010，其最高位为 0，表示正数。

$1011\ 1001+0111\ 1010=1\ 0011\ 0011$，只有 8 位，所以结果为 0011 0011，即 51。符合结果 $-71+122=51$，所以没有上溢或下溢。

3.22 [10] <3.5> 如果是浮点数，位模式 0x0C000000 表示的十进制数是什么? 使用 IEEE 754 标准。

这是 32 位的浮点数，即单精度浮点数。根据 IEEE 754 标准，即 1 位符号位、8 位指数位、23 位尾数位。

$$\underbrace{0}_{\text{符号位}}\,|\,\underbrace{000\ 1100\ 0}_{\text{指数位}}\,|\,\underbrace{000\ 0000\ 0000\ 0000\ 0000\ 0000}_{\text{尾数位}}$$

符号位是 0，表示正数；指数位是 0001 1000，转化为十进制为 24；尾数位是 0。

所以位模式 0x0C000000 表示的十进制数为

$$(-1)^{\text{符号}}\times(1+\text{尾数})\times 2^{\text{指数}-127}=(-1)^0\times(1+0.0)\times 2^{24-127}=2^{-103}$$

3.23 [10]<3.5> 假定采用 IEEE 754 单精度格式，写出十进制数 63.25 的二进制表示。

$$63.25=\frac{253}{4}=1111\ 1101_2\times 2^{-2}=1.111\ 1101_2\times 2^5=(-1)^0\times(1+0.111\ 1101)\times 2^{132-127}$$

根据 IEEE 754 标准，单精度格式为 1 位符号位、8 位指数位、23 位尾数位，132 转化为 8 位二进制为 1000 0100，所以十进制数 63.25 的二进制表示为

$$\underbrace{0}_{\text{符号位}}\,|\,\underbrace{100\ 0010\ 0}_{\text{指数位}}\,|\,\underbrace{111\ 1101\ 0000\ 0000\ 0000\ 0000}_{\text{尾数位}}$$

即

$$0100\ 0010\ 0111\ 1101\ 0000\ 0000\ 0000\ 0000$$

第四章　处理器

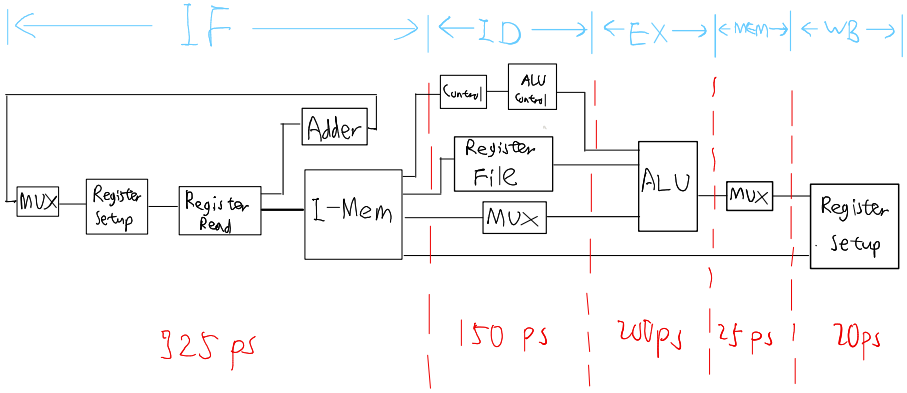
4.7 假设用来实现处理器数据通路的各功能模块延迟如下所示：

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

其中，寄存器读延迟指的是，时钟上升沿到寄存器输出端稳定输出新值所需的时间。该延迟仅针对 PC 寄存器。寄存器建立时间指的是，寄存器的输入数据稳定到时钟上升沿所需的时间。该数值针对 PC 寄存器和寄存器堆。

4.7.1 [5]<4.4>R 型指令的延迟是多少？比如，如果想让这类指令工作正确，时钟周期最少为多少？

R 型指令的步骤如下图所示。其中，在译码阶段，Control 的延迟为 50ps，Register File 的延迟为 150ps，Mux 的延迟为 25ps，由于这三个步骤可以同时执行，所以延迟取最大值，即 150ps。在访存（MEM）阶段，由于 R 型指令不需要访问内存，所以只需要通过一个多路选择器（MUX），所以延迟为 25ps。

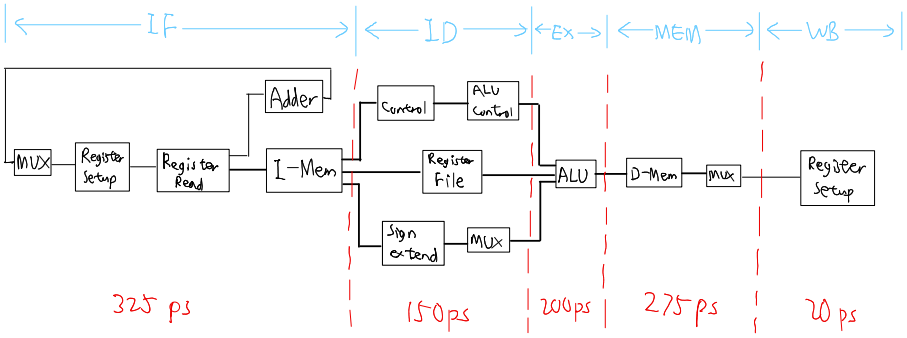


因此，延迟为 $325 + 150 + 200 + 25 + 20 = 720$ ps，即如果想让这类指令工作正确，时钟周期最少为 720 ps。

4.7.2 [10]<4.4>ld 指令的延迟是多少？仔细检查你的答案，许多学生会在关键路径上添加额外的寄存器。

ld 指令的步骤如下图所示，可以观察到在译码步骤中，虽然延迟的组成部分和上一

题不一样，但由于 Register File 的延迟较长，因此总的延迟还是由 Register File 决定，即 150ps。IF、ID、EX 步骤的延迟与上一题没有改变，但后面两个步骤有所改变。



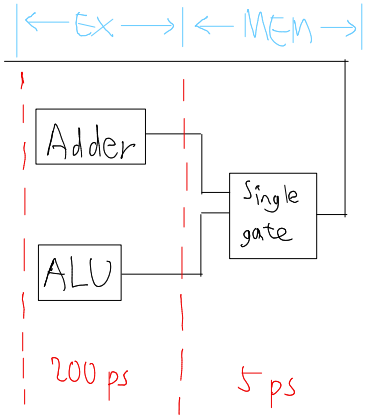
因此，延迟为 $325 + 150 + 200 + 275 + 20 = 970$ ps。

4.7.3 [10]<4.4>sd 指令的延迟是多少? 仔细检查你的答案，许多学生会在关键路径上添加额外的寄存器。

前三个步骤仍然与之前一样，在 MEM 中，只需要访问 D-Mem，即 250 ps，并且没有 WB 步骤，所以延迟为 $325 + 150 + 200 + 250 = 925$ ps。

4.7.4 [5]<4.4> beq 指令的延迟是多少?

前三个步骤仍然与之前一样，只需要在 MEM 中加入一个 Single gate 的延迟，并且没有 WB 步骤。



因此，延迟为 $325 + 150 + 200 + 5 = 680$ ps。

4.7.5 [5]<4.4>I 型指令的延迟是多少?

与 R 型指令类似，I 型指令只是在 ID 阶段需要在 MUX 前加入 Sign extend 的延迟，但仍然没有 Register File 的延迟大，所以 ID 步骤仍然需要 150 ps 的延迟，所以总延迟仍然为 720 ps。

4.7.6 [5]<4.4> 该 CPU 的最小时钟周期是多少?

由于延迟最长的指令为 ld 指令，所以该 CPU 的最小时钟周期为 ld 指令的延迟，即 970 ps。

4.8 [10]<4.4> 假设你能设计一款处理器并让每条指令执行不同的周期数。给定指令比例如下表所示，相比图 4-21 中的处理器，这款新处理器的加速比是多少?

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

R 型指令和 I 型指令没有 MEM 阶段，ld 指令 5 个阶段都有，sd 指令没有 WB 阶段，beq 指令没有 MEM 和 WB 阶段。可以认为一个阶段的执行需要一个时钟周期，如果所有指令执行相同的周期数，那么都需要 5 个时钟周期，而如果每条指令可以执行不同的周期数，那么各类指令的周期数之比为 R/I:4/5；ld:1；sd:4/5；beq:3/5。将各类指令的周期比按照指令比例加权平均即可得到周期比：

$$\frac{4}{5} \times 0.52 + 1 \times 0.25 + \frac{4}{5} \times 0.11 + \frac{3}{5} \times 0.12 = 0.826$$

取倒数即可得到加速比：$\frac{1}{0.826} = \frac{500}{413} \approx 1.21065375302663$

4.16 在本题中将讨论流水线如何影响处理器的时钟周期。假设数据通路的各个流水级的延迟如下:

IF	ID	EX	MEM	WB
250ps	350ps	150ps	300ps	200ps

同时，假设处理器执行的指令分布如下：

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

4.16.1 [5]<4.5> 在流水化和非流水的处理器中，时钟周期分别是多少?

流水化的处理器：按照最长的阶段，即 350ps；
非流水化的处理器：所有阶段延迟之和，即 $250 + 350 + 150 + 300 + 200 = 1250$ ps。

4.16.2 [10]<4.5 > 在流水化和非流水的处理器中，对于 ld 指令的延迟分别是多少?

ld 指令 5 个阶段都有，在流水化的处理器中，为 $350 \times 5 = 1750$ ps；
非流水化的处理器中，为所有阶段延迟之和，即 $250 + 350 + 150 + 300 + 200 = 1250$ ps。

4.16.3 [10]<4.5> 如果我们将数据通路中的一个流水级拆成两个新流水级，每一个新流水级的延迟是原来的一半，那么我们将拆分哪一级？新处理器的时钟周期是多少？

应该拆分最长的一级，即 ID 阶段，拆分之后最长的阶段延迟为 300 ps，所以新处理器的时钟周期是 300 ps。

4.16.4 [10]<4.5> 假设没有停顿或冒险，数据存储的利用率如何？

既然题目中出现了“停顿”“冒险”这种只有在流水化处理器中才会出现的情况，那么说明这里只需要考虑流水化的情况。数据存储对应的是 MEM 阶段，MEM 阶段只需要 300ps，但是为了满足流水线，延迟到了 350ps，所以 MEM 阶段的利用率为 $\frac{300}{350}$；而在题目给出的指令分布中，只有 Load 和 Store 指令会用到数据存储，即 $20\%+15\%$，所以数据存储的利用率为

$$\frac{300}{350}\times(0.2+0.15)=0.3$$

4.16.5 [10]<4.5> 假设没有停顿或冒险，寄存器堆的写口利用率如何？

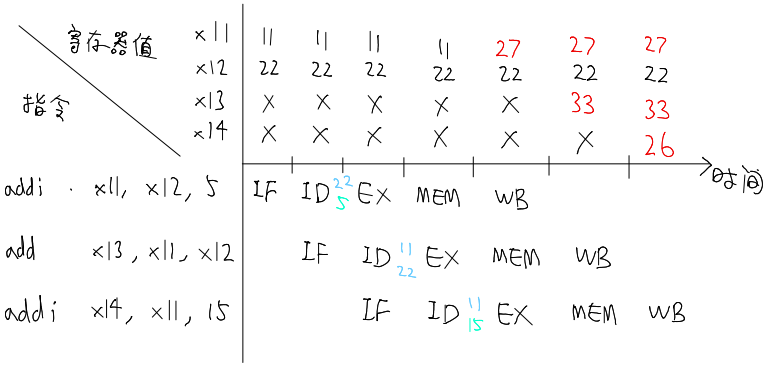
寄存器堆的写口对应的是 WB 阶段，WB 阶段的利用率为 $\frac{200}{350}$；在题目给出的指令分布中，使用到寄存器堆的写口的指令为 ALU/Logic 和 Load，即 $45\%+20\%$，所以寄存器堆写口的利用率为

$$\frac{200}{350}\times(0.45+0.2)=\frac{13}{35}\approx 0.371428571428571$$

4.18 [5]<4.5> 假设初始化寄存器 x11 为 11,x12 为 22，如果在 4.5 节中的流水线结构上执行下述代码，寄存器 x13 和 x14 中最终为何值？注：硬件不处理数据冒险，编程者需要在必要处插入 NOP 指令来解决数据冒险。

```
addi    x11, x12, 5
add     x13, x11, x12
addi    x14, x11, 15
```

显然在硬件不处理数据冒险情况下，执行上述代码会出现数据冒险，以下是示意图，在 ID 指令旁边标注了实际取出的操作数。



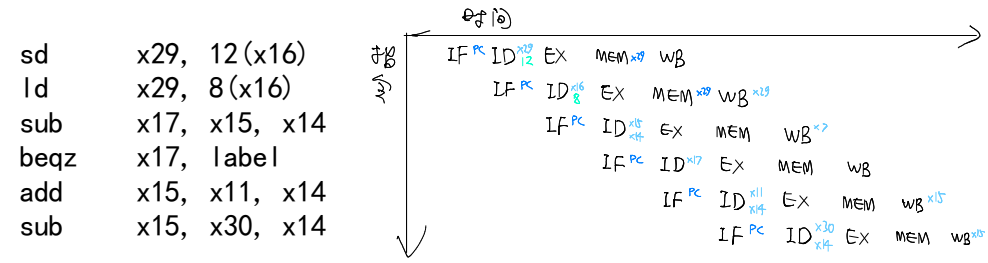
所以寄存器 x13 最终为 33，寄存器 x14 最终为 26。

4.22 [5]<4.5> 对于如下的 RISC-V 的汇编片段:

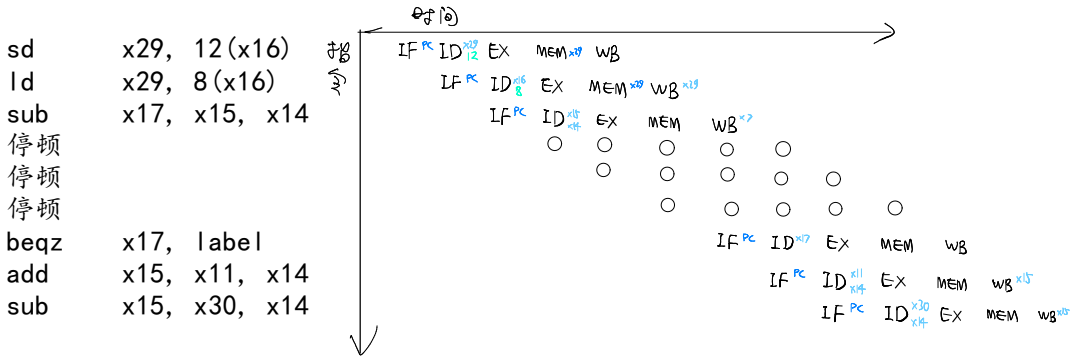
```
sd      x29, 12(x16)
ld      x29, 8(x16)
sub     x17, x15, x14
beqz    x17, label
add     x15, x11, x14
sub     x15, x30, x14
```

4.22.1 [5]<4.5> 请画出流水线图，说明以上代码会在何处停顿。

在加入停顿之前的流水线图是这样的，显然由于前面的指令的 MEM 阶段和后面的指令的 IF 阶段都需要访问存储器，会发生结构冒险。



加入停顿后流水线图变成了这样:



4.22.2 [5]<4.5> 是否可通过重排代码来减少因结构冒险而导致停顿的次数?

可以，由于只考虑结构冒险（即两条指令在同一个阶段访问寄存器堆或存储器的冒险），可以把第一行的 sd 指令放到第二三行的 ld 和 sub 指令后面，这样就可以减少一个停顿。

4.22.3 [5]<4.5> 该结构冒险必须用硬件来解决吗? 我们可以通过在代码中插入 NOP 指令来消除数据冒险，对于结构冒险是否可以相同处理? 如果可以，请解释原因。否则，也请解释原因。

不一定要用硬件来解决，可以通过插入 NOP 指令来消除结构冒险，因为 NOP 指令相当于一个停顿，只需要把上述的停顿换成 NOP 指令即可。

4.22.4 [5]<4.5> 在典型程序中，大约需要为该结构冒险产生多少个周期的停顿?(使用习题 4.8 中的指令分布)。

仔细观察可以发现，停顿的产生是由于 sd 和 ld 有 MEM 阶段，会和后续指令的 IF 阶段冲突，ld 和 R 型指令有 WB 阶段，会和后续指令的 ID 阶段冲突，那么可以认为一个 ld 导致 2 个停顿，一个 sd 导致 1 个停顿，一个 R 型导致 1 个停顿。所以大概产生的停顿周期数为：

$$1 \times 0.52 + 2 \times 0.25 + 1 \times 0.11 = 1.13$$

即产生 1.13 × 原始时钟周期数 个周期的停顿。

4.27 讨论下述指令序列，假设在一个五级流水的数据通路中执行:

```
add     x15, x12, x11
ld      x13, 4(x15)
ld      x12, 0(x2)
or      x13, x15, x13
sd      x13, 0(x15)
```

4.27.1 [5]<4.7> 如果没有前递逻辑或者冒险检测支持，请插入 NOP 指令保证程序正确执行。

```
add     x15, x12, x11       // 在第 5 个阶段写入 x15
nop
nop
ld      x13, 4(x15)         // 在第 2 个阶段读取 x15，在第 5 个阶段写入 x13
ld      x12, 0(x2)          // 在第 2 个阶段读取 x2，在第 5 个阶段写入 x12
nop
or      x13, x15, x13       // 在第 2 个阶段读取 x13 和 x15，在第 5 个阶段写入 x13
nop
nop
sd      x13, 0(x15)         // 在第 2 个阶段读取 x13 和 x15
```

4.27.2 [10]<4.7> 对代码进行重排，插入最少的 NOP 指令。假设寄存器 x17 可用来做临时寄存器。

```
add     x15, x12, x11
nop
nop
ld      x13, 4(x15)
ld      x12, 0(x2)
```

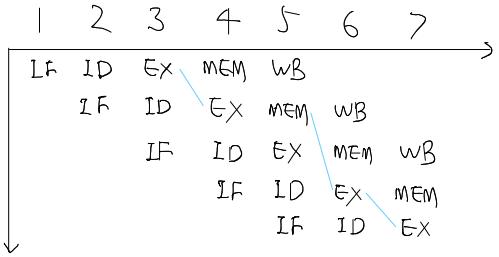
```
nop
or      x17, x15, x13
sd      x17, 0(x15)
```

4.27.3 [10]<4.7> 如果处理器中支持前递，但未实现冒险检测单元，上述代码段的执行将会发生什么?

不会发生数据冒险，因为在不存在加载后马上使用的情况，第二行加载到 x13 后在第 4 行才使用 x13，此时已经可以使用前递确保正确执行。

4.27.4 [20]<4.7> 以图 4-58 中的冒险检测和前递单元为例，如果执行上述代码，在前 7 个时钟周期中，每周期哪些信号会被它们置为有效?

```
add     x15, x12, x11
ld      x13, 4(x15)
ld      x12, 0(x2)
or      x13, x15, x13
sd      x13, 0(x15)
```



前递信号如图所示。所以在前 7 个时钟周期中，有效信号表示如下表:

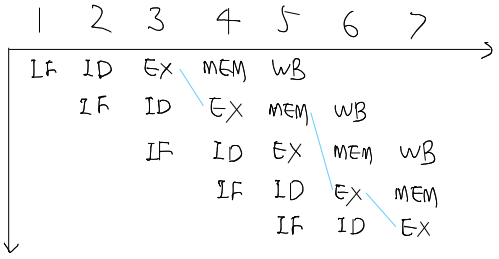
时钟周期	冒险检测	ForwardA	ForwardB
1	x	x	x
2	x	x	x
3	有效	有效	x
4	x	x	x
5	有效	x	有效
6	有效	有效	x
7	x	x	x

4.27.5 [10]<4.7> 如果没有前递单元，以图 4-58 中的冒险检测逻辑为例，需要为其增加哪些输入和输出信号?

第 3 个周期的前递，需要 IF/ID.Rs1 和 ID/EX.Rd 的输入信号，ForwardA 的输出信号；
第 5 个周期的前递，需要 IF/ID.Rs2 和 EX/MEM.Rd 的输入信号，ForwardB 的输出信号；
第 6 个周期的前递，需要 IF/ID.Rs1 和 ID/EX.Rd 的输入信号，ForwardA 的输出信号。
综上所述，需要增加 IF/ID.Rs1, IF/ID.Rs2, ID/EX.Rd, EX/MEM.Rd 的输入信号，ForwardA, ForwardB 的输出信号。

4.27.6 [20]<4.7> 如果使用习题 4.26.5 中的冒险检测单元，执行上述代码，在前 5 个时钟周期中, 每个周期哪些输出信号会有效?

```
add     x15, x12, x11
ld      x13, 4(x15)
ld      x12, 0(x2)
or      x13, x15, x13
sd      x13, 0(x15)
```



前递信号如图所示。所以在前 5 个时钟周期中，有效信号表示如下表：

时钟周期	冒险检测	ForwardA	ForwardB
1	x	x	x
2	x	x	x
3	有效	有效	x
4	x	x	x
5	有效	x	有效

第五章　大而快：层次化存储

5.2 cache 对于为处理器提供高性能存储器层次结构非常重要。下面是 64 位存储器地址访问顺序列表，以字地址的形式给出。

0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c, 0xba, 0xbd

5.2.1 [10]<5.3> 对于一个有 16 个 cache 块、每个块大小为 1 个字的 cache，标识这些引用的二进制字地址、标签和索引。此外，假设 cache 最初为空，列出每个地址的访问会命中还是失效。

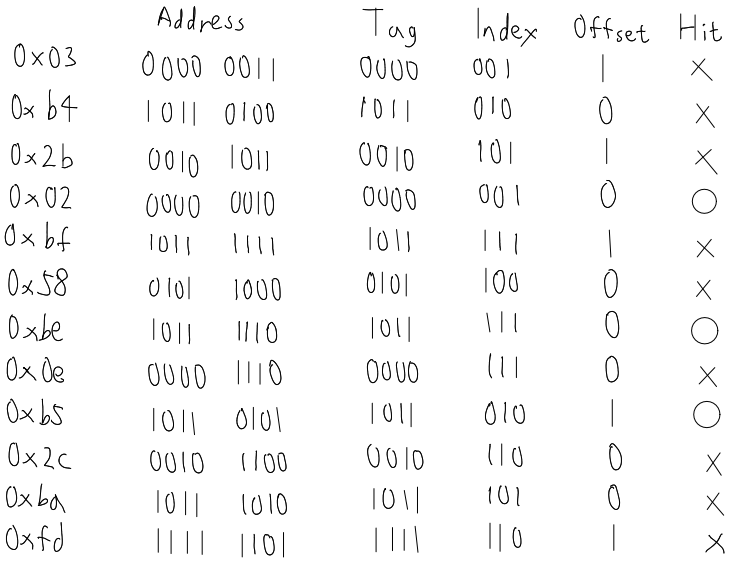
缓存块有 16 个，也就是 2^4，所以索引的大小为 4 个位，给出的地址为两个十六进制位，即 8 位二进制，所以索引实际上就是低 4 位地址，标签实际上是就是高 4 位地址，是否命中只需要看在某一行之前是否出现过相同的标签和索引。

	Address		Tag	Index	Hit
0x03	0000	0011	0000	0011	×
0xb4	1011	0100	1011	0100	×
0x2b	0010	1011	0010	1011	×
0x02	0000	0010	0000	0010	×
0xbf	1011	1111	1011	1111	×
0x58	0101	1000	0101	1000	×
0xbe	1011	1110	1011	1110	×
0x0e	0000	1110	0000	1110	×
0xb5	1011	0101	1011	0101	×
0x2c	0010	1100	0010	1100	×
0xba	1011	1010	1011	1010	×
0xfd	1111	1101	1111	1101	×

5.2.2 [10]<5.3> 对于一个有 8 个 cache 块、每个块大小为 2 个字的 cache，标识这些引用的二进制字地址、标签、索引和偏移。此外，假设 cache 最初为空，列出每个地址的访问会命中还是失效。

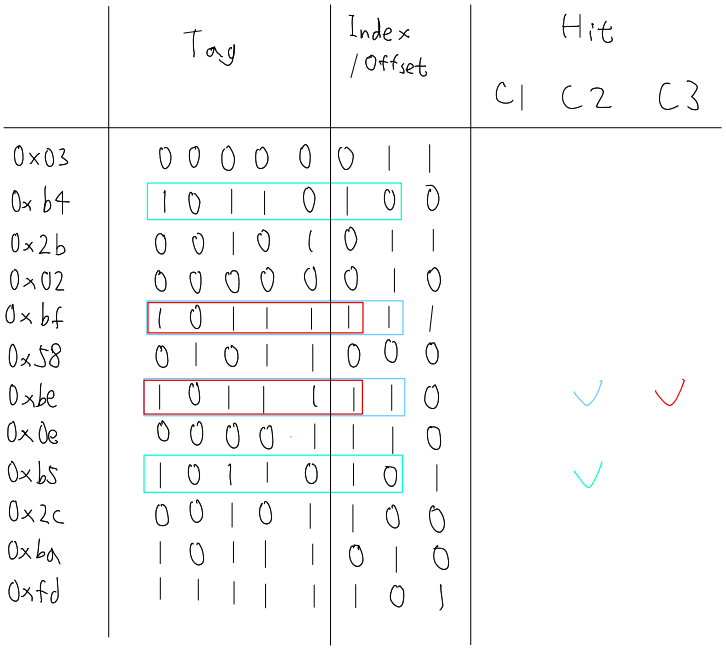
每个缓存块的大小变大了，缓存块的数量变小了，所以偏移实际上就是把索引的最

低位移过来了。同样是否命中只需要看在某一行之前是否出现过相同的标签和索引，并且在这之后没有索引相同但标签不同的访问（不然就被置换了），不需要考虑偏移。这里的命中一列，不命中使用 × 表示，而命中使用○表示（用○比 √从视觉上更好区分）。



5.2.3 [20]<5.3,5.4> 请针对给定的访问顺序优化 cache 设计。这里有三种直接映射 cache 的设计方案, 每种方案都可以容纳 8 个字的数据:

- C1 的块大小为 1 个字
- C2 的块大小为 2 个字
- C3 的块大小为 4 个字



可见 C2 的缓存命中次数最多，所以这里应该选择 C2 方案。

5.3 按照惯例, cache 以它包含的数据量来进行命名（例如，4KiB cache 可以容纳 4KiB 的数据），但是 cache 还需要 SRAM 来存储元数据，如标签和有效位等。本题研究 cache 的配置如何影响实现它所需的 SRAM 总量以及 cache 的性能。对所有的部分，都假设 cache 是字节可寻址的，并且地址和字都是 64 位的。

5.3.1 [10]<5.3> 计算实现每个块大小为 2 个字的 32KiB cache 所需的总位数。

32KiB 即 $2^5 \times 2^{10} = 2^{15}$B，每个块大小为 2 个字即 $2 \times 64/8 = 2^4$B，由于字节寻址所以偏移为 4 位，共有 $2^{15-4} = 2^{11}$ 个块，因此索引为 11 位，那么地址为 64 位，所以标签就是 $64 - 4 - 11 = 49$ 位，有效位为 1 位。每个块都有一个标签和一个有效位，所以 SRAM 存储的元数据需要 $2^{11} \times (49 + 1) = 102400$ 位。再加上存储数据的 2^{15+3} 位，即$\mathbf{2^{11} \times (49 + 1) + 2^{15+3} = 364544}$ **位**。

5.3.2 [10]<5.3> 计算实现每个块大小为 16 个字的 64KiB cache 所需的总位数。这个 cache 比 5.3.1 中描述的 32KiB cache 大多少?(请注意，通过更改块大小，我们将数据量增加了一倍，但并不是将 cache 的总大小加倍。)

64KiB 即 $2^6 \times 2^{10} = 2^{16}$B，每个块的大小为 16 个字即 $16 \times 64/8 = 2^7$B，所以偏移为 7 位。

共有 $2^{16-7} = 2^9$ 个块，索引为 9 位。 标签为 $64 - 7 - 9 = 48$ 位，有效位为 1 位。

所以需要的总位数为 $2^9 \times (48+1) + 2^{16+3} = \mathbf{549376}$ 位，比 5.3.1 大了 $\frac{549376-364544}{364544} = 0.507022471910112$，即约大了 $\mathbf{50.70}\%$。

5.3.3 [5]<5.3> 解释为什么 5.3.2 中的 64KiB cache 尽管数据量比较大，但是可能会提供比 5.3.1 中的 cache 更慢的性能。

5.3.2 中的 64KiB cache 每个块的大小比 5.3.1 大，所以每次缓存失效时都需要把更多的数据读入缓存或写回存储，所以失效代价会更大。而且总块数更少，所以命中率会更低。

5.3.4 [10]<5.3,5.4> 生成一系列读请求，这些请求需要在 32KiB 的两路组相联 cache 上的失效率低于在 5.3.1 中描述的 cache 的失效率。

在 5.3.1 中标签 49 位，索引 11 位，偏移 4 位。若改成两路组相联，那么每个块就只有 1 个字了，且索引为 10 位，标签为 50 位，每个块都各自有一个标签和一个有效位。所以只需要构造索引相同的地址，但是标签不同，即可让 5.3.1 的直接映射失效，而两路组相联不失效。所以构造的读请求地址如下：**0x00000, 0x10000, 0x00000, 0x10000, 0x00000, 0x10000, ……**，前两次请求都未命中缓存，但从第三次请求开始，直接映射会一直失效，而两路组相联会一直命中，所以符合要求。

5.5 对一个 64 位地址的直接映射 cache 的设计，地址的以下位用于访问 cache。

标签	索引	偏移
63 ~ 10	9 ~ 5	4 ~ 0

5.5.1 [5]<5.3>cache 块大小为多少（以字为单位）？

偏移是 0 到 4，也就是 5 位，所以 cache 块的大小是 $2^5 = 32$B，一个字是 4B，所以 cache 块的大小是 **8** 个字。

5.5.2 [5]<5.3 >cache 块有多少个?

索引从 5 到 9，也就是 5 位，所以 cache 块有 $2^5 = \mathbf{32}$ 个。

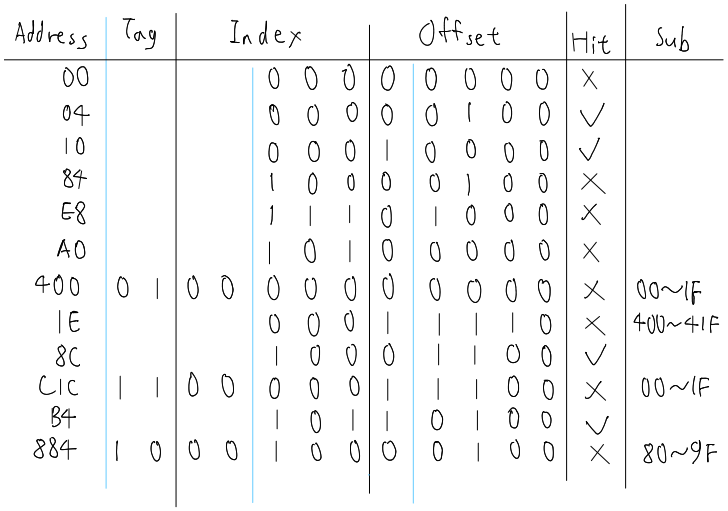
5.5.3 [5]<5.3> 这种 cache 实现所需的总位数与数据存储位之间的比率是多少?

标签是 10 到 63，也就是 54 位，还需要一位有效位，所以实现所需的总位数是 $2^5 \times (54 + 1) + 2^5 \times 2^5 \times 8 = 9952$，数据存储位是 $2^5 \times 2^5 \times 8 = 8192$。所以这种 cache 实现所需的总位数与数据存储位之间的比率是 $\frac{9952}{8192} \times 100\% = \mathbf{121.484375}\%$。

5.5.4 下表记录了从上电开始 cache 访问的字节地址。

十六进制	00	04	10	84	E8	A0	400	1E	8C	C1C	B4	884
十进制	0	4	16	132	232	160	1024	30	140	3100	180	2180

[20]<5.3> 对每一次访问，列出：它的标签、索引和偏移；指出命中还是失效；替换了哪个字节 (如果有的话)。



图中的 Address 表示十六进制的地址，Tag 为标签,，Index 为索引，Offset 为偏移，Hit 列中勾表示命中，叉表示失效；Sub 表示替换的字节地址范围。

5.5.5 [5]<5.3> 命中率是多少?

12 次访问中有 4 次命中，所以命中率为 $\frac{1}{3} \approx$ **33.33**%。

5.5.6 [5]<5.3> 列出 cache 的最终状态，每个有效表项表示为 < 索引，标签，数据 > 的记录。例如:

```
<0, 3, Mem[0xC00]-Mem[0xC1F]
```

对于某个索引，从后往前看 Index，首次找到这个索引的地方所在的标签即为最终状态的标签，之后把偏移全填为 0，再把地址转成十六进制（已按 4 位一组用蓝色线分隔），即为起始地址；把偏移量全填为 1，再把地址转成十六机制，即为结束地址。

例如，对于 0 号索引，最后一次出现是在地址 C1C 的地方，它的标签为 11，也就是十六进制的 3。之后看这一行的地址为 1100 0001 1100，将偏移量全填成 0，也就是 1100 0000 0000，这就是 0xC00，将偏移量全填成 1，也就是 1100 0001 1111，这就是 0xC1F，所以结果为 `<0, 3, Mem[0xC00]-Mem[0xC1F]` 。

所以 cache 的最终状态如下:

```
<0, 3, Mem[0xC00]-Mem[0xC1F]>
<4, 2, Mem[0x880]-Mem[0x89F]>
<5, 0, Mem[0x0A0]-Mem[0x0BF]>
<7, 0, Mem[0x0E0]-Mem[0x0FF]>
```

5.7 考虑以下的程序和 cache 行为:

每 1000 条指令的数据读次数	每 1000 条指令的数据写次数	指令 cache 失效率	数据 cache 失效率	块大小（字节）
250	100	0.30%	2%	64

5.7.1 [10]<5.3,5.8> 假设一个带有写直达、写分配 cache 的 CPU 实现了 2 的 CPI，那么 RAM 和 cache 之间的读写带宽（用每个周期的字节数进行测量）是多少?(假设每个失效都会生成一个块的请求。)

CPI 为 2，每条指令 2 个周期，那么每个周期就是 0.5 条指令，那么指令的读带宽就是 $0.5 \times 0.30\% \times 64 = 0.096$ 字节/周期，而数据的读带宽就是 $0.5 \times \frac{250}{1000} \times 2\% \times 64 = 0.16$ 字节/周期。

由于写直达，所以不管是否失效，都需要将某个寄存器的数据写入到 RAM 中，RISC-V 中的寄存器为 64 位，也就是 8 个字节，那么写入 RAM 的带宽为 $0.5 \times \frac{100}{1000} \times 8 = 0.4$ 字节/周期。

由于写分配，所以当失效时，需要先（后 *）将 RAM 中的数据放回到寄存器中，这时会产生读带宽 $0.5 \times \frac{100}{1000} \times 2\% \times 64 = 0.064$ 字节/周期。

(* 如果先取回数据，那么需要同时写入 cache 和 RAM；如果后取回数据，那么就只写入 RAM，取回时就已经是最新的数据了。)

所以总的读带宽为 0.096+0.16+0.064 = **0.32**字节/周期，总的写带宽为 **0.4**字节/周期。

5.7.2 [10]<5.3,5.8> 对于一个写回、写分配 cache 来说，假设替换出的数据 cache 块中有 30% 是脏块，那么为了实现 CPI 为 2，读写带宽需要达到多少?

指令的读带宽仍为 0.096 字节/周期，数据的读带宽仍为 0.16 字节/周期。但是这里的“写分配”似乎没用？写的时候如果不失效，那 cache 中就是最新的；如果失效，那先替换旧的块，之后再写入 cache，也不会出现分配的情况。

对于写回策略，当写不失效时，不需要在 cache 与 RAM 中传输数据。当写失效时，如果被替换的 cache 块是“脏”块，也就是被修改过，那么需要将这个块写入到 RAM 中。当读失效时，仍然会产生 cache 块的替换，所以如果是“脏”块也需要写入 RAM。并且这里没有提及缓冲的事，所以认为没有写缓冲，那么写带宽为 $0.5\times\left(\frac{250}{1000}+\frac{100}{1000}\right)\times 2\%\times 30\%\times 64 = 0.0672$。

所以总的读带宽为 **0.096** 字节/周期，总的写带宽为 **0.0672** 字节/周期。

5.9 cache 块大小 (B) 可以影响失效率和失效延迟。假设一台机器的基本 CPI 为 1，每条指令的平均访问次数（包括指令和数据）为 1.35，给定以下各种不同 cache 块大小的失效率，找到能够最小化总失效延迟的 cache 块大小。

8:4%	16:3%	32:2%	64:1.5%	128:1%

5.9.1 [10]<5.3> 失效延迟为 20×B 周期时，最优块大小是多少?

总失效延迟为 $1.35\times$ 失效率 $\times 20\times B$ 周期，所以所求问题为

$$\arg\min_i \quad 1.35\times 失效率_i \times 20 \times B_i$$

$$\{(B_i, 失效率_i)\} = \{(8, 0.04), (16, 0.03), (32, 0.02), (64, 0.015), (128, 0.01)\}$$

枚举可得

B_i	失效率$_i$	总失效延迟
8	0.04	8.64
16	0.03	12.96
32	0.02	17.28
64	0.015	25.92
128	0.01	34.56

所以最优块大小是 8。

5.9.2 [10]<5.3> 失效延迟为 24+B 周期时，最优块大小是多少?

总失效延迟为 $1.35 \times$ 失效率 $\times (24 + B)$ 周期，所以所求问题为

$$\arg\min_i \quad 1.35 \times \text{失效率} \times (24 + B)$$

$$\{(B_i, \text{失效率}_i)\} = \{(8, 0.04), (16, 0.03), (32, 0.02), (64, 0.015), (128, 0.01)\}$$

枚举可得

B_i	失效率$_i$	总失效延迟
8	0.04	1.728
16	0.03	1.62
32	0.02	1.512
64	0.015	1.782
128	0.01	2.052

所以最优块大小是 32。

5.9.3 5.9.3[10]<5.3> 失效延迟为定值时，最优块大小是多少?

总失效延迟为 $1.35 \times$ 失效率 $\times$ 定值，所以失效率越小，总失效延迟就越小，**所以最优块大小是 128。**

5.10 本题研究不同 cache 容量对整体性能的影响。通常, cache 访问时间与 cache 容量成正比。假设主存访问需要 70ns，并且在所有指令中有 36% 的指令访问数据内存。下表显示了两个处理器 P1 和 P2 中每个处理器各自的 L1 cache 的数据。

	L1 大小	L1 失效率	L1 命中时间
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

5.10.1 [5]<5.4> 假设 L1 命中时间决定 Pl 和 P2 的时钟周期时间，它们各自的时钟频率是多少?

P1: $\frac{1}{0.66\times10^{-9}} \approx 1.515 \times 10^9\text{Hz} = 1.515\text{GHz}$。

P2: $\frac{1}{0.90\times10^{-9}} \approx 1.111 \times 10^9\text{Hz} = 1.111\text{GHz}$。

5.10.2 [10]<5.4>P1 和 P2 各自的 AMAT (平均内存访问时间) 是多少 (以周期为单位)?

这里的平均内存访问时间应该是每条指令的，而且是已知产生内存访问的情况下计算的，所以为

$$\text{P1:} \quad 1 + 8\% \times \left\lceil \frac{70}{0.66} \right\rceil = 9.56\text{周期}$$

$$\text{P2:}\quad 1+6\%\times\left\lceil\frac{70}{0.90}\right\rceil=5.68\text{周期}$$

5.10.3 [5]<5.4> 假设基本 CPI 为 1.0 而且没有任何内存停顿，那么 P1 和 P2 的总 CPI 是多少? 哪个处理器更快?(当我们说 “基本 CPI 为 1.0” 时，意思是指令在一个周期内完成，除非指令访问或者数据访问导致 cache 失效。)

这里计算的总 CPI 仍然是每条指令的周期数，但是并没有已知产生内存访问，所以

$$\text{P1:}\quad 1+8\%\times\left\lceil\frac{70}{0.66}\right\rceil+36\%\times 8\%\times\left\lceil\frac{70}{0.66}\right\rceil=12.6416\text{周期/指令}$$

$$\text{P2:}\quad 1+6\%\times\left\lceil\frac{70}{0.90}\right\rceil+36\%\times 6\%\times\left\lceil\frac{70}{0.90}\right\rceil=7.3648\text{周期/指令}$$

所以 P2 更快。

对于接下来的三个问题，我们将考虑向 P1 添加 L2 cache(可能弥补其有限的 Ll cache 容量)。解决这些问题时，请使用上一个表中的 Ll cache 容量和命中时间。L2 失效率表示的是其局部失效率。

L2 大小	L2 失效率	L2 命中时间
1 MiB	95%	5.62 ns

5.10.4 [10]<5.4> 添加 L2 cache 的 P1 的 AMAT 是多少? 在使用 L2 cache 后,AMAT 变得更好还是更差?

$$1+8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times 95\%\times\left\lceil\frac{70}{0.66}\right\rceil=9.852\text{周期}$$

显然使用 L2 cache 后，AMAT 变得更差。

5.10.5 [5]<5.4> 假设基本 CPI 为 1.0 而且没有任何内存停顿，那么添加 L2 cache 的 P1 的总 CPI 是多少?

$$1+8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times 95\%\times\left\lceil\frac{70}{0.66}\right\rceil+36\%\times\left(8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times 95\%\times\left\lceil\frac{70}{0.66}\right\rceil\right)=13.03872$$

5.10.6 [10]<5.4> 为了使具有 L2 cache 的 P1 比没有 L2 cache 的 P1 更快，需要 L2 的失效率为多少?

设 L2 的失效率最大为 x，则

$$1+8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times x\times\left\lceil\frac{70}{0.66}\right\rceil=9.56$$

解得 $x=\frac{98}{107}\approx 0.9159=91.59\%$，所以 L2 的失效率需要小于 91.59%。

5.10.7 [15]<5.4> 为了使具有 L2 cache 的 P1 比没有 L2 cache 的 P2 更快，需要 L2 的失效率为多少?

设 L2 的失效率最大为 x，由于涉及到 P1 和 P2，所以这里需要比较的是实际的执行时间，即 CPI × 每条指令执行的时间，由于 5.10.1 的假设，所以每条指令执行的时间就是 L1 命中时间。

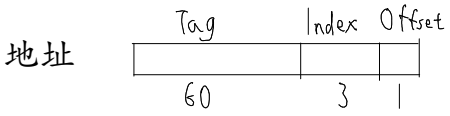
$$\left[1+8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times x\times\left\lceil\frac{70}{0.66}\right\rceil+36\%\times\left(8\%\times\left\lceil\frac{5.62}{0.66}\right\rceil+8\%\times x\times\left\lceil\frac{70}{0.66}\right\rceil\right)\right]\times 0.66$$
$$=7.3648\times 0.90$$

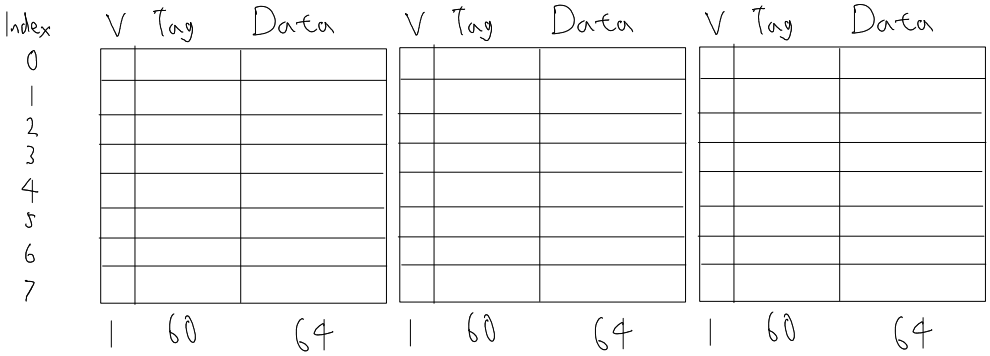
解得 $x=\frac{27719}{40018}\approx 0.6927=69.27\%$，所以 L2 的失效率需要小于 69.27%。

5.11 本题研究不同 cache 设计的效果，特别是将组相联 cache 与 5.4 节中的直接映射 cache 进行比较。有关这些练习，请参阅下面显示的字地址序列:

0x03, 0xb4, 0x2b, 0xbe, 0x58, 0xbf, 0x0e, 0x1f, 0xb5, 0xba, 0x2e, 0xce

5.11.1 [10]<5.4> 绘制块大小为 2 字、总容量为 48 字的三路组相联 cache 的组织结构图。图中应有类似于图 5-18 的样式，还应该清楚地显示标签和数据字段的宽度。





块大小为 $2=2^1$ 字，所以地址中的偏移量为 1 位。$48\div 3\div 2=8$，所以索引有 $8=2^3$ 行，所以地址中的索引有 3 位。所以标签为 $64-3-1=\mathbf{60}$ 位。一个字应该是 32 位，并且块大小为 2 字，那么数据字段就是 $2\times 32=\mathbf{64}$ 位。

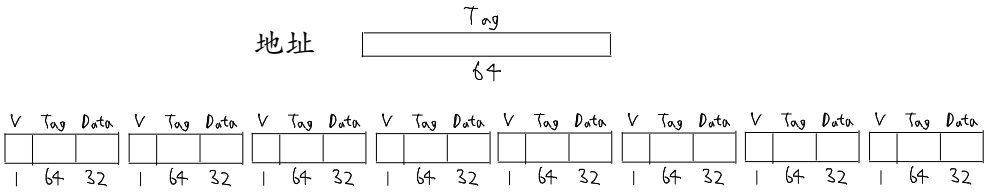
5.11.2 [10]<5.4> 从 5.11.1 中记录 cache 的行为。假设 cache 使用 LRU 替换策略。对于每一次 cache 访问，确定：

- 二进制字地址。
- 标签。
- 索引。
- 偏移。
- 访问会命中还是失效。
- 在处理访问后，cache 每一路中有哪些标签。

Hexadecimal Address	Binary Address			Hit	Way 1	Way 2	Way 3
	Tag	Index	Offset		Index - Tag	Index - Tag	Index - Tag
03	0000	00	11	×	1-0		
b4	1011	01	00	×	1-0, 2-b		
2b	0010	10	11	×	1-0, 2-b, 5-2		
02	0000	00	10	○	1-0, 2-b, 5-2		
be	1011	11	10	×	1-0, 2-b, 5-2, 7-b		
58	0101	10	00	×	1-0, 2-b, 4-5, 5-2, 7-b		
bf	1011	11	11	○	1-0, 2-b, 4-5, 5-2, 7-b		
0e	0000	11	10	×	1-0, 2-b, 4-5, 5-2, 7-b	7-0	
1f	0001	11	11	×	1-0, 2-b, 4-5, 5-2, 7-b	7-0	7-1
b5	1011	01	01	○	1-0, 2-b, 4-5, 5-2, 7-b	7-0	7-1
bf	1011	11	11	○	1-0, 2-b, 4-5, 5-2, 7-b	7-0	7-1
ba	1011	10	10	×	1-0, 2-b, 4-5, 5-2, 7-b	5-b, 7-0	7-1
2e	0010	11	10	×	1-0, 2-b, 4-5, 5-2, 7-b	5-b, 7-2	7-1
ce	1100	11	10	×	1-0, 2-b, 4-5, 5-2, 7-b	5-b, 7-2	7-c

图中的 Hit（命中）列用蓝色圈表示命中，命中时在上一行会有蓝色框表示命中了哪个缓存。红色下划线表示在这一时刻产生了缓存替换，标记了新的缓存放在哪个位置。

5.11.3 [5]<5.4> 绘制块大小为 1 字、总容量为 8 字的全相联 cache 的组织结构图。图中应有类似于图 5-18 的样式，还应该清楚地显示标签和数据字段的宽度。



地址是按字索引的，块大小是 1 字，所以没有偏移量。由于是全相联，所以没有索引位，所以标签为 64 位。总容量为 8 字，所以 8 个块排成一行。数据字段就是块大小即 1 字 = 32 位。

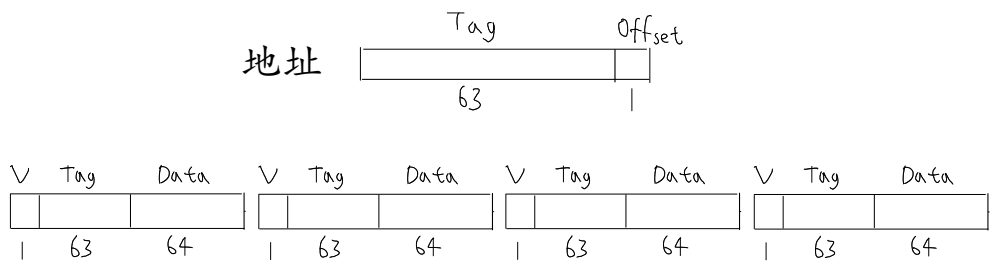
5.11.4 [10]<5.4> 从 5.11.3 中记录 cache 的行为。假设 cache 使用 LRU 替换策略。对于每一次 cache 访问，确定：

- 二进制字地址。
- 标签。
- 索引。
- 偏移。
- 访问会命中还是失效。
- 在处理访问后，cache 每一路中有哪些标签。

Hexadecimal Address	Binary Address Tag	Hit	Way 0	1	2	3	4	5	6	7
03	00000011	×	03							
b4	10110100	×	03	b4						
2b	00101011	×	03	b4	2b					
02	00000010	×	03	b4	2b	02				
be	10111110	×	03	b4	2b	02	be			
58	01011000	×	03	b4	2b	02	be	58		
bf	10111111	×	03	b4	2b	02	be	58	bf	
0e	00001110	×	03	b4	2b	02	be	58	bf	0e
1f	00011111	×	1f	b4	2b	02	be	58	bf	0e
b5	10110101	×	1f	b5	2b	02	be	58	bf	0e
bf	10111111	○	1f	b5	2b	02	be	58	bf	0e
ba	10111010	×	1f	b5	ba	02	be	58	bf	0e
2e	00101110	×	1f	b5	ba	2e	be	58	bf	0e
ce	11001110	×	1f	b5	ba	2e	ce	58	bf	0e

图中的 Hit（命中）列用蓝色圈表示命中，命中时在上一行会有蓝色框表示命中了哪个缓存。红色下划线表示在这一时刻产生了缓存替换，标记了新的缓存放在哪个位置。

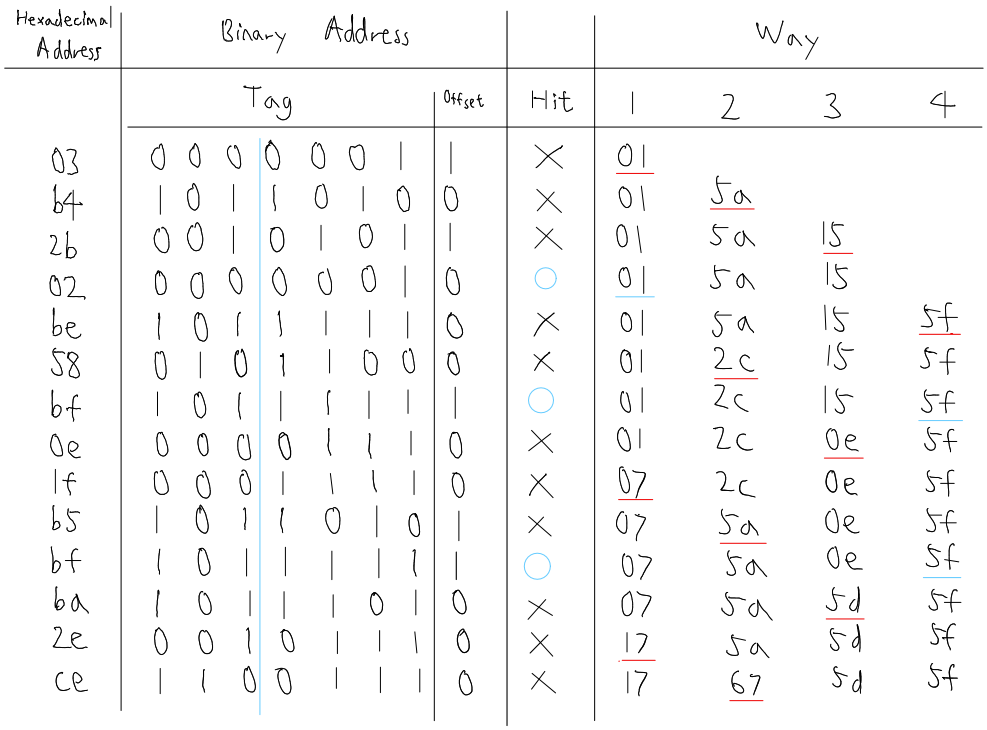
5.11.5 [5]<5.4> 绘制块大小为 2 字、总容量为 8 字的全相联 cache 的组织结构图。图中应有类似于图 5-18 的样式，还应该清楚地显示标签和数据字段的宽度。



块大小为 2 字，所以偏移量为 1 位，一个字是 32 位，所以数据大小为 64 位。总容量为 8 字，所以有 4 个块，全相联所以 4 个块排成一行，没有索引位，所以标签为 $64 - 1 = 63$ 位。

5.11.6 [10]<5.4> 从 5.11.5 中记录 cache 的行为。假设 cache 使用 LRU 替换策略。对于每一次 cache 访问，确定:

- 二进制字地址。
- 标签。
- 索引。
- 偏移。
- 访问会命中还是失效。
- 在处理访问后，cache 每一路中有哪些标签。



图中 Tag 列中的蓝色线表示 4 位分隔，便于二进制转十六进制。Hit（命中）列用蓝色圈表示命中，在命中时右侧的蓝色下划线表示命中了哪一路的标签，同时也代表产生了一次访问（相当于 LRU 把它放到链表最后）。在未命中时右侧的红色下划线表示替换了哪一路的标签，同时也代表产生了一次访问。

所以，对于每一行，填写方法是：先把 Tag 转成十六进制，再查看 Way（这里全相联就是四路组相联，Way 表示路）中是否有这个十六进制地址，如果有，表示命中，那就把上一行的 Way 复制下来，并在命中的地址上划一条蓝色下划线；如果没有，表示未命中，就从这行开始向上查看最近的哪条下划线距离最远（表示最久没访问），那么就替换这个地址，其他地址不变，替换后在这个地址上划一条红色下划线。

5.12 多级 cache 是一种重要的技术，可以在克服一级 cache 提供的有限空间的同时仍然保持速度。考虑具有以下参数的处理器:

无内存停顿的基本 CPI	处理器速度	主存访问时间	每条指令的 L1 cache 的失效率 *	L2 直接映射 cache 速度	L2 直接映射 cache 全局失效率	L2 八路组相联速度	L2 八路组相联 cache 全局失效率
1.5	2GHz	100ns	7%	12cycles	3.5%	28cycles	1.5%

*Ll cache 失效率是针对每条指令而言的。假设 Ll cache 的总失效数量（包括指令和数据）为总指令数的 7%。

这题的参考答案有误，答案当成局部失效率计算了。（全局失效率与局部失效率的定义在课本第 290 页）

5.12.1 [10]<5.4> 使用以下方法计算表中处理器的 CPI: 仅有 Ll cache; 使用 L2 直接映射 cache; 使用 L2 八路组相联 cache。如果主存访问时间加倍，这些数据会如何变化?(将每个更改作为绝对 CPI 和百分比更改。) 请注意 L2 cache 可以隐藏慢速内存影响的程度。

先计算主存访问的时钟周期，$2 \times 10^9\text{Hz} \times 100 \times 10^{-9}\text{s} = 200\text{cycles}$。

全局失效率是指访问 L2 并且 L2 失效的指令数量与全部指令数量的比值；局部失效率是指访问 L2 并且 L2 失效的指令数量与访问 L2 的指令的数量的比值。

- 仅有 L1 cache 时，CPI 为 $1.5 + 7\% \times 200 = \mathbf{15.5}$ 周期；
- 使用 L2 直接映射 cache 时，CPI 为 $1.5 + 7\% \times 12 + 3.5\% \times 200 = \mathbf{9.34}$ 周期；
- 使用 L2 八路组相联 cache 时，CPI 为 $1.5 + 7\% \times 28 + 1.5\% \times 200 = \mathbf{6.46}$ 周期。

如果主存访问时间加倍，

- 仅有 L1 cache 时，CPI 为 $1.5 + 7\% \times 400 = \mathbf{29.5}$ 周期，增加了 $29.5 - 15.5 = \mathbf{14}$ 个周期，增加了 $14/15.5 = \mathbf{90.3225806451613}\%$；
- 使用 L2 直接映射 cache 时，CPI 为 $1.5 + 7\% \times 12 + 3.5\% \times 400 = \mathbf{16.34}$ 周期，增加了 $16.34 - 9.34 = \mathbf{7}$ 个周期，增加了 $7/15.5 = \mathbf{45.1612903225806}\%$；
- 使用 L2 八路组相联 cache 时，CPI 为 $1.5 + 7\% \times 28 + 1.5\% \times 400 = \mathbf{9.46}$ 周期，增加了 $9.46 - 6.46 = \mathbf{3}$ 个周期，增加了 $3/6.46 = \mathbf{46.4396284829721}\%$。

5.12.2 [10]<5.4> 可能有比两级更多的 cache 层次结构吗? 已知上述处理器具有 L2 直接映射 cache, 设计人员希望添加一个 L3 cache，访问时间为 50 个时钟周期，并且该 cache 将具有 13% 的失效率。这会提供更好的性能吗? 一般来说，添加 L3 cache 有哪些优缺点?

可能有比两级更多的 cache 层次结构。这里的 13% 失效率没有说局部还全局。那么，如果它是全局失效率，那么肯定要比 L2 的全局失效率低，但是这里它比 L2 的全局失效率高，所以只能是局部失效率。

那么加入 L3 cache 后的 CPI 为：$1.5 + 7\% \times 12 + 3.5\% \times (50 + 13\% \times 200) = 5$ 周期 < 9.34 周期，所以**会**提供更好的性能。

添加 L3 cache 的优点是能用更小的全局失效率兜底，隐藏慢速内存影响的程度，减小总体的 CPI；缺点是一旦全部缓存都失效，必须访问主存时，会产生很大的延迟。

5.12.3 [20]<5.4> 在较老的处理器中，例如 Intel Pentium 或 Alpha 21264，L2 cache 在主处理器和 Ll cache 的外部 (位于不同芯片上)。虽然这种做法使得大型 L2 cache 成为可能，但是访问 cache 的延迟也变得很高，并且因为 L2 cache 以较低的频率运行，所以带宽通常也很低。假设 512KiB 的片外 L2 cache 的失效率为 4%，· 如果每增加一个额外的 512KiB cache 能够降低 0.7% 的失效率，并且 cache 的总访问时间为 50 个时钟周期，那么 cache 容量必须多大才能与上面列出的 L2 直接映射 cache 的性能相匹配?

这里 4% 的失效率应该为局部失效率。设有 $x+1$ 个 512 KiB 的片外 L2 cache。那么

$$1.5+7\%\times 12+3.5\%\times 200=1.5+7\%\times(50+(4\%-0.7\%x)\times 200)$$

解得 $x=-\frac{270}{7}=-38.5714285714286$，但 x 应 $\geqslant 0$，所以**不存在合适的 cache 容量与上面列出的 L2 直接映射 cache 的性能相匹配。**

5.16 如 5.7 节所述，虚拟内存使用页表来跟踪虚拟地址到物理地址的映射。本题显示了在访问地址时必须如何更新页表。以下数据构成了在系统上看到的虚拟字节地址流。假设有 4KiB 页，一个 4 表项全相联的 TLB，使用严格的 LRU 替换策略。如果必须从磁盘中取回页，请增加下一次能取的最大页码:

十进制	4669	2227	13916	34587	48870	12608	49225
十六进制	0x123d	0x08b3	0x365c	0x871b	0xbee6	0x3140	0xc049

TLB

有效位	标签	物理页号	上次访问时间间隔
1	0xb	12	4
1	0x7	4	1
1	0x3	6	3
0	0x4	9	7

页表

索引	有效位	物理页号/在磁盘中
0	1	5
1	0	在磁盘中
2	0	在磁盘中
3	1	6
4	1	9
5	1	11
6	0	在磁盘中
7	1	4
8	0	在磁盘中
9	0	在磁盘中
a	1	3
b	1	12

5.16.1 [10]<5.7> 对于上述每一次访问，列出:

- 本次访问在 TLB 会命中还是失效。
- 本次访问在页表中会命中还是失效。
- 本次访问是否会造成缺页错误。
- TLB 的更新状态。

4KiB 即 2^{12}Bytes，所以对于一个十六进制地址，右侧三位十六进制表示页内偏移，左侧一位表示标签。

Address		TLB	PT	Page				
Tag	Offset	Hit	Hit	Fault	V	Tag	Physical Page Number	Last Access Interval
1	2 3 d	×	√	√	1	b	12	5
					1	7	4	2
					1	3	6	4
					1	1	13	0
0	8 b 3	×	√	×	1	0	5	0
					1	7	4	3
					1	3	6	5
					1	1	13	1
3	6 5 c	√	√	×	1	0	5	1
					1	7	4	4
					1	3	6	0
					1	1	13	2
8	7 1 b	×	√	√	1	0	5	2
					1	8	14	0
					1	3	6	1
					1	1	13	3
b	e e 6	×	√	×	1	0	5	3
					1	8	14	1
					1	3	6	2
					1	b	12	0
3	1 4 0	√	√	×	1	0	5	4
					1	8	14	2
					1	3	6	0
					1	b	12	1
c	0 4 9	×	×	√	1	c	15	0
					1	8	14	3
					1	3	6	1
					1	b	12	2

- TLB 是否失效，只需要查看是否在上一行的 Tag 中出现过;
- 页表是否失效，只需要看标签是否在页表中出现;
- 如果某个标签在磁盘中，或者不在页表中，都会造成缺页错误;
- 每次访问后，TLB 中的上次访问时间间隔都需要加一（用蓝色表示），如果 TLB 未命中，则需要替换上次访问时间间隔最大的那一行（用红色表示），不管是否命中都需要把当前访问到的 Tag 所在的那行的上次访问时间间隔改为 0（用红色表示）;

- 题目中的“如果必须从磁盘中取回页，请增加下一次能取的最大页码”的意思是发生缺页错误时，分配的物理页号是当前最大的物理页号加一。
- 答案中的 last access 是相对顺序，每次替换序号最小的一行，而这里是指访问时间间隔，所以每次替换最大的一行。

5.16.2 [15]<5.7> 重复 5.16.1，但这次使用 16KiB 页而不是 4KiB 页。拥有更大页大小的优势是什么？有什么缺点？

16KiB = 2^{14}Bytes，所以右侧 14 位二进制位表示页内偏移，左侧 2 位二进制位表示页号。

Tag	Offset binary	Offset hexadecimal	TLB Hit	PT Hit	Page Fault	V	Tag	Physical Page Number	Last Access Interval
0 0	0 1	23d	×	√	×	1	b	12	
						1	7	4	
						1	3	6	
						1	0	5	
0 0	0 0	8b3	√	√	×	1	b	12	
						1	7	4	
						1	3	6	
						1	0	5	
0 0	1 1	65c	√	√	×	1	b	12	
						1	7	4	
						1	3	6	
						1	0	5	
1 0	0 0	71b	×	√	√	1	2	13	
						1	7	4	
						1	3	6	
						1	0	5	
1 0	1 1	ee6	√	√	×	1	2	13	
						1	7	4	
						1	3	6	
						1	0	5	
0 0	1 1	140	√	√	×	1	2	13	
						1	7	4	
						1	3	6	
						1	0	5	
1 1	0 0	049	√	√	×	1	2	13	
						1	7	4	
						1	3	6	
						1	0	5	

由于标签只有 2 位，所以一共 4 行的 TLB 不会出现很多次置换，这里的上次访问间隔就省略了（直接按照原始的 TLB，先置换第 4 行，再置换第 1 行，再置换第 3 行，再置换第 2 行），每次置换的行仍然用红色表示。

拥有更大页大小的优势是有更高的快表命中率，缺点是会降低内存使用率（产生了更多内零头）。

5.16.3 [15]<5.7> 重复 5.16.1，但这次使用 4KiB 页和一个两路组相联 TLB。

一个页是 4KiB，两个页组成一行。右侧 12 位表示页内偏移，中间 1 位表示索引，左侧 3 位表示标签。页号仍然是左侧 4 位。

Page Number	Address			Physical Page Number	TLB Hit	PT Hit	Page Fault	Way 1				Way 2			
	Tag	Index	Offset					V	Tag	Physical Page Number	Last Access Interval	V	Tag	Physical Page Number	Last Access Interval
1	000	1	234	13	×	√	√	1	6	12	4	1	3	6	3
								1	7	4	1	1	0	13	0
0	000	0	8b3	5	×	√	×	1	0	5	0	1	3	6	4
								1	7	4	2	1	0	13	1
3	001	1	65c	6	×	√	×	1	0	5	1	1	3	6	5
								1	1	6	0	1	0	13	2
8	100	0	71b	14	×	√	√	1	0	5	2	1	4	14	0
								1	1	6	1	1	0	13	3
b	101	1	eE6	12	×	√	×	1	0	5	3	1	4	14	1
								1	1	6	2	1	5	12	0
3	001	1	140	6	√	√	×	1	0	5	4	1	4	14	2
								1	1	6	0	1	5	12	1
c	110	0	049	15	×	×	√	1	6	15	0	1	4	14	3
								1	1	6	1	1	5	12	2

5.16.4 [15]<5.7> 重复 5.16.1，但这次使用 4KiB 页和一个直接映射 TLB。

一个页是 4KiB，右侧 12 位表示页内偏移，中间 2 位表示索引，左侧 2 位表示标签。页号仍然是左侧 4 位。

Page Number	Physical Page Number	Tag	Index	Offset	TLB Hit	PT Hit	Page Fault	V	Tag	Physical Page Number
1	13	0 0	0 1	23d	X	√	√	1	b	12
								1	0	13
								1	3	6
								0	4	9
0	5	0 0	0 0	8b3	X	√	X	1	0	5
								1	0	13
								1	3	6
								0	4	9
3	6	0 0	1 1	65c	X	√	X	1	0	5
								1	0	13
								1	3	6
								1	0	6
8	14	1 0	0 0	71b	X	√	√	1	2	14
								1	0	13
								1	3	6
								1	0	6
b	12	1 0	1 1	ee6	X	√	X	1	2	14
								1	0	13
								1	3	6
								1	2	12
3	6	0 0	1 1	140	X	√	X	1	2	14
								1	0	13
								1	3	6
								1	0	6
c	15	1 1	0 0	049	X	X	√	1	3	15
								1	0	13
								1	3	6
								1	0	6

5.16.5 [10]<5.4,5.7> 讨论为什么 CPU 必须使用 TLB 才能实现高性能。如果没有 TLB，如何处理虚拟内存访问?

为了便于编写程序，写代码时不需要指定某段数据放在哪个物理地址中，出现了虚拟地址，为了将虚拟地址存放到实际的物理地址中，需要有个表存放这个映射关系，这就是页表，而页表存放在内存中，访问比较慢。但是每次访问虚拟地址时，都需要访问一次页表再访问实际的数据，也就是访问两次内存，所以就出现了 TLB（快表）作为页表的缓存，在 TLB 命中时只需要访问一次内存，从而提升性能。

如果没有 TLB，每次虚拟内存访问就需要访问两次内存，第一次访问页表，第二次再访问实际的数据。