计算机系统 HW 3

SA25011049 李宇哲

T1（5.13）

••5.13 假设我们想编写一个计算两个向量 u 和 v 内积的过程。这个函数的一个抽象版本对整数和浮点数类型，在 x86-64 上 CPE 等于 14～18。通过进行与我们将抽象程序 combine1 变换为更有效的 combine4 相同类型的变换，我们得到如下代码：

```
1   /* Inner product.  Accumulate in temporary */
2   void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3   {
4       long i;
5       long length = vec_length(u);
6       data_t *udata = get_vec_start(u);
7       data_t *vdata = get_vec_start(v);
8       data_t sum = (data_t) 0;
9
10      for (i = 0; i < length; i++) {
11          sum = sum + udata[i] * vdata[i];
12      }
13      *dest = sum;
14  }
```

测试显示，对于整数这个函数的 CPE 等于 1.50，对于浮点数据 CPE 等于 3.00。对于数据类型 double，内循环的 x86-64 汇编代码如下所示：

```
   Inner loop of inner4.  data_t = double, OP = *
   udata in %rbp, vdata in %rax, sum in %xmm0
   i in %rcx, limit in %rbx
1  .L15:                                    loop:
2    vmovsd  0(%rbp,%rcx,8), %xmm1            Get udata[i]
3    vmulsd  (%rax,%rcx,8), %xmm1, %xmm1      Multiply by vdata[i]
4    vaddsd  %xmm1, %xmm0, %xmm0              Add to sum
5    addq    $1, %rcx                         Increment i
6    cmpq    %rbx, %rcx                       Compare i:limit
7    jne     .L15                             If !=, goto loop
```

假设功能单元的特性如图 5-12 所示。

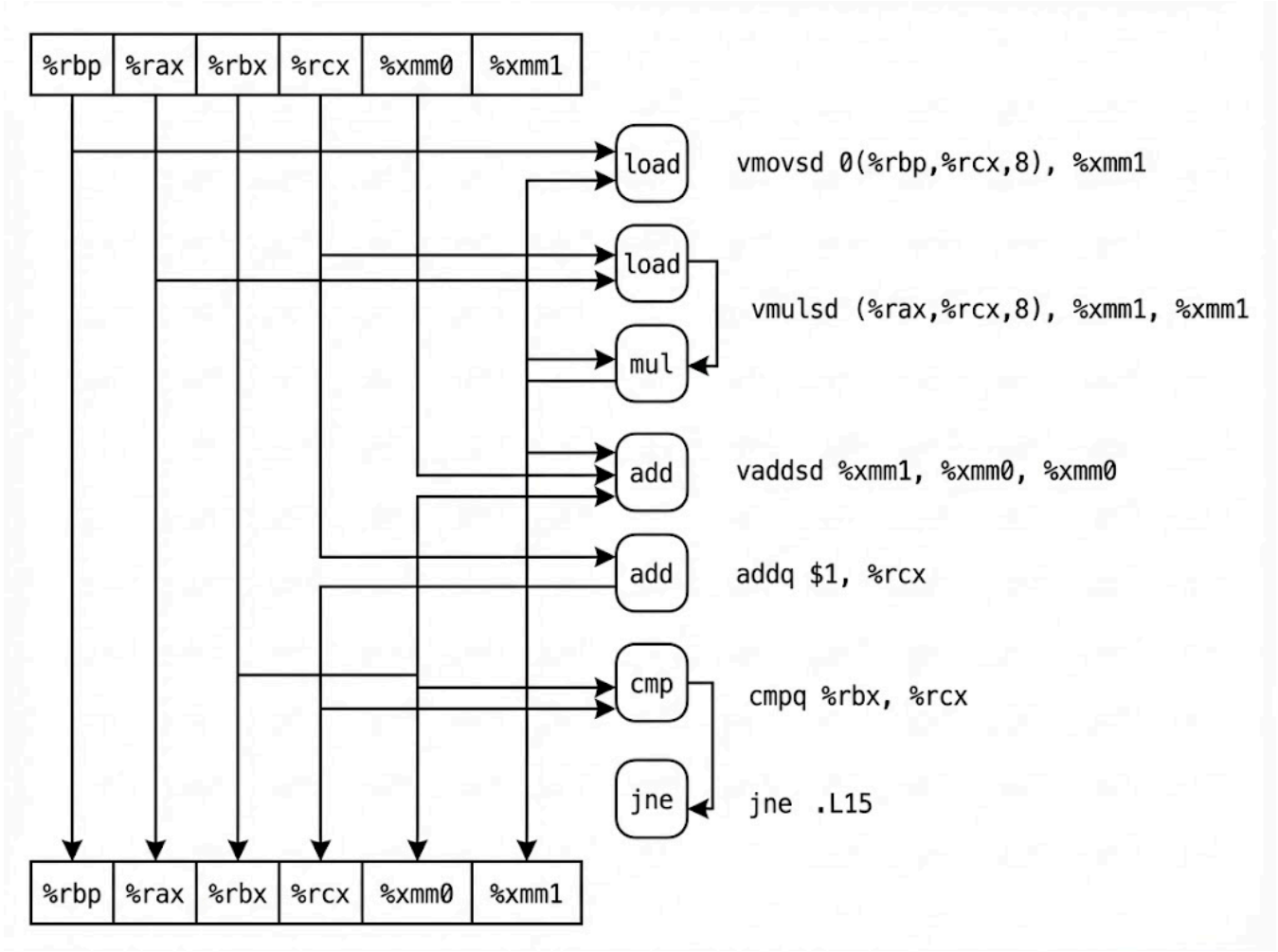
A. 按照图 5-13 和图 5-14 的风格，画出这个指令序列会如何被译码成操作，并给出它们之间的数据相关如何形成一条操作的关键路径。

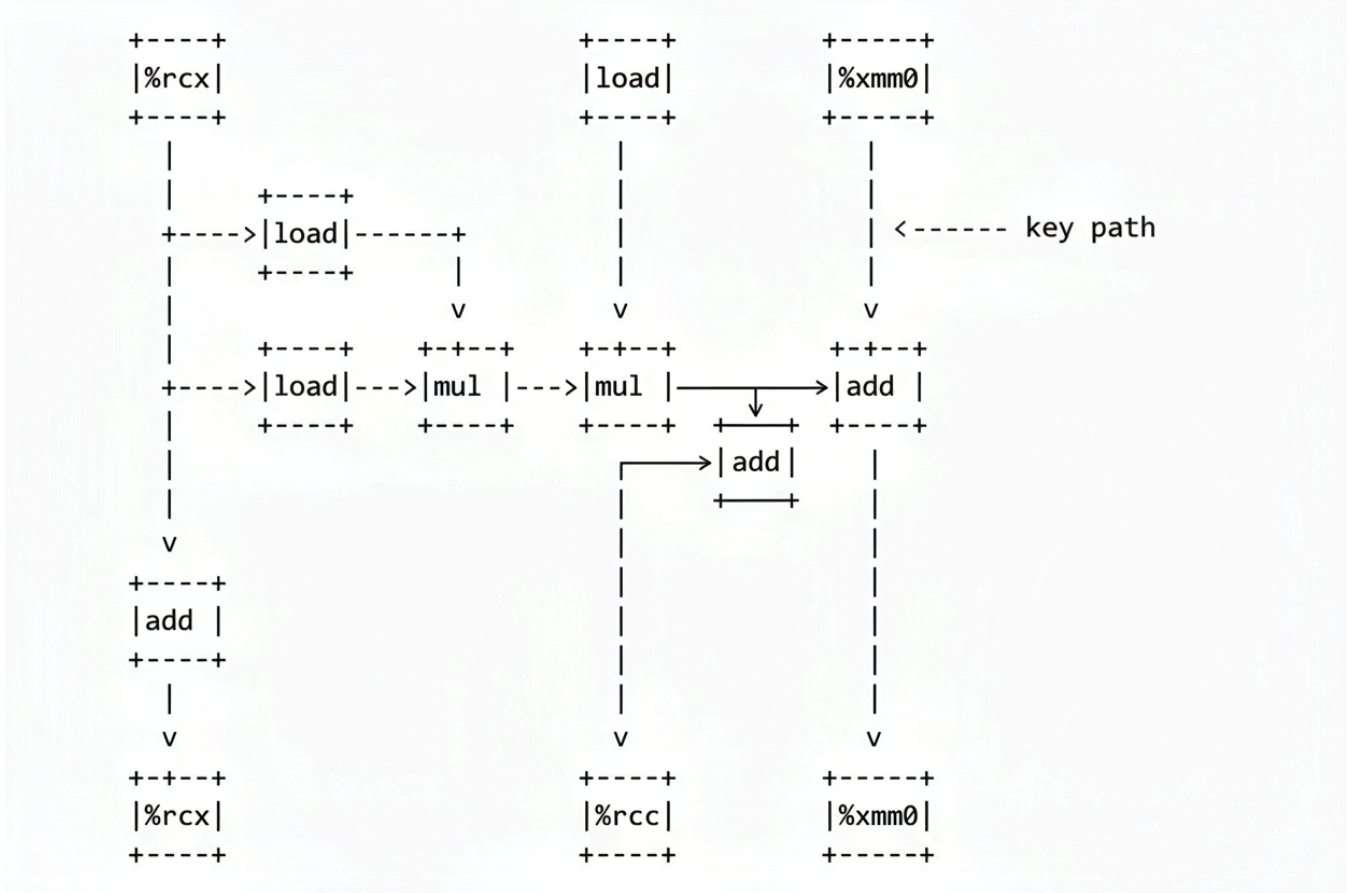
B. 对于数据类型 double，这条关键路径决定的 CPE 的下界是什么？

C. 假设对于整数代码也有类似的指令序列，对于整数数据的关键路径决定的 CPE 的下界是什么？

D. 请解释虽然乘法操作需要 5 个时钟周期，但是为什么两个浮点版本的 CPE 都是 3.00。

A.


%rbp
%rax
%rbx
%rcx
%xmm0
%xmm1
load
vmovsd 0(%rbp,%rcx,8), %xmm1
load
vmulsd (%rax,%rcx,8), %xmm1, %xmm1
mul
add
vaddsd %xmm1, %xmm0, %xmm0
add
addq $1, %rcx
cmp
cmpq %rbx, %rcx
jne
jne .L15
%rbp
%rax
%rbx
%rcx
%xmm0
%xmm1


%rcx
load
%xmm0
load
<------ key path
load
mul
mul
add
add
add
%rcx
%rcc
%xmm0

B.

CPE 的下界由浮点加法的延迟决定，CPE 是 3.0

C.

CPE 的下界由整数加法的延迟决定。CPE 是 1.0

D.

浮点乘法不在循环所在的关键路径上，每一轮迭代中乘法操作只依赖于当前轮加载的数据，不依赖于上一轮的计算结果。所以 cpu 可以利用流水线并行蛆执行多个 乘法操作，因此CPE与浮点乘无关，还是跟浮点加有关

T2（5.17）

** 5.17 库函数 memset 的原型如下：

```
void *memset(void *s, int c, size_t n);
```

这个函数将从 s 开始的 n 个字节的内存区域都填充为 c 的低位字节。例如，通过将参数 c 设置为 0，可以用这个函数来对一个内存区域清零，不过用其他值也是可以的。

下面是 memset 最直接的实现：

```
1  /* Basic implementation of memset */
2  void *basic_memset(void *s, int c, size_t n)
3  {
4      size_t cnt = 0;
5      unsigned char *schar = s;
6      while (cnt < n) {
7          *schar++ = (unsigned char) c;
8          cnt++;
9      }
10     return s;
11 }
```

实现该函数一个更有效的版本，使用数据类型为 unsigned long 的字来装下 8 个 c，然后用字级的写遍历目标内存区域。你可能发现增加额外的循环展开会有所帮助。在我们的参考机上，能够把 CPE 从直接实现的 1.00 降低到 0.127。即，程序每个周期可以写 8 个字节。

这里是一些额外的指导原则。在此，假设 K 表示你运行程序的机器上的 sizeof(unsigned long)的值。

- 你不可以调用任何库函数。
- 你的代码应该对任意 n 的值都能工作，包括当它不是 K 的倍数的时候。你可以用类似于使用循环展开时完成最后几次迭代的方法做到这一点。
- 你写的代码应该无论 K 的值是多少，都能够正确编译和运行。使用操作 sizeof 来做到这一点。
- 在某些机器上，未对齐的写可能比对齐的写慢很多。(在某些非 x86 机器上，未对齐的写甚至可能会导致段错误。)写出这样的代码，开始时直到目的地址是 K 的倍数时，使用字节级的写，然后进行字级的写，(如果需要)最后采用用字节级的写。
- 注意 cnt 足够小以至于一些循环上界变成负数的情况。对于涉及 sizeof 运算符的表达式，可以用无符号运算来执行测试。(参见 2.2.8 节和家庭作业 2.72。)

```
1 #include <stddef.h> // for size_t
2 #include <stdint.h> // for uintptr_t
3 #include <stdio.h>  // for printf
```

```c
#include <string.h> // for memcmp

void *effective_memset(void *s, int c, size_t n) {
    unsigned char *schar = (unsigned char *)s;
    unsigned char uc = (unsigned char)c;
    size_t K = sizeof(unsigned long);

    // Alignment
    while (n > 0 && ((uintptr_t)schar % K != 0)) {
        *schar++ = uc;
        n--;
    }
    if (n >= K) {
        unsigned long c_long = 0;
        for (size_t i = 0; i < K; i++) {
            c_long |= ((unsigned long)uc << (8 * i));
        }
        unsigned long *slong = (unsigned long *)schar;
        size_t cnt = n / K;
        size_t limit = cnt - 7;
        while (cnt >= 8) {
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            *slong++ = c_long;
            cnt -= 8;
        }
        // Remain
        while (cnt > 0) {
            *slong++ = c_long;
            cnt--;
        }
        schar = (unsigned char *)slong;
        n = n % K;
    }
    while (n > 0) {
        *schar++ = uc;
        n--;
    }

    return s;
}
```

对上述函数我进行了 4 个 testcase 的 测试，结果如下

```c
// Test main function
int main() {
    // Test 1: Basic functionality
    char buf1[100];
```

```c
    effective_memset(buf1, 0xAA, 100);
    int test1_passed = 1;
    for (int i = 0; i < 100; i++) {
        if ((unsigned char)buf1[i] != 0xAA) {
            test1_passed = 0;
            break;
        }
    }
    printf("Test 1 (basic memset): %s\n", test1_passed ? "PASSED" : "FAILED");

    // Test 2: Small buffer
    char buf2[5] = {1, 2, 3, 4, 5};
    effective_memset(buf2, 0xCC, 5);
    int test2_passed = 1;
    for (int i = 0; i < 5; i++) {
        if ((unsigned char)buf2[i] != 0xCC) {
            test2_passed = 0;
            break;
        }
    }
    printf("Test 2 (small buffer): %s\n", test2_passed ? "PASSED" : "FAILED");

    // Test 3: Alignment test (unaligned start)
    char buf3[100];
    char *unaligned = buf3 + 3;
    effective_memset(unaligned, 0xBB, 50);
    int test3_passed = 1;
    for (int i = 0; i < 50; i++) {
        if ((unsigned char)unaligned[i] != 0xBB) {
            test3_passed = 0;
            break;
        }
    }
    printf("Test 3 (unaligned start): %s\n", test3_passed ? "PASSED" : "FAILED");

    // Test 4: Return value
    char buf4[10];
    void *ret = effective_memset(buf4, 0xDD, 10);
    printf("Test 4 (return value): %s\n", (ret == buf4) ? "PASSED" : "FAILED");

    return 0;
}
```

测试结果：

```
✗ innerpeace@innerpeace ~/csapp/hw/hw3 clang 5_17.c
innerpeace@innerpeace ~/csapp/hw/hw3 ./a.out
Test 1 (basic memset): PASSED
Test 2 (small buffer): PASSED
Test 3 (unaligned start): PASSED
Test 4 (return value): PASSED
```