These problems cover floating point in Chapter 2.4. The problems are similar to the practice problems in the text, which have solutions at the end of the chapter.

Question #1:

Do problem 2.86 in the text (representation of numbers). You must show your work to receive full credit.

Question #2:

Do problem 2.87 in the text (more representation). You must show your work to receive full credit.

Question #3:

Work out "long hand" the steps to add "e" 2.71828 to "c" 299792458 (the speed of light in m/s) as single precision numbers. You should first convert the two numbers to single precision floating point numbers and show their representation, indicating the sign, exponent and fractional part. Then, show the individual steps and intermediate products to add them (i.e. adjust the exponents, show the adjusted values and then do the addition and adjust the final result).

You can use a C program and 'gdb' to assist in the conversion of the numbers of single precision floating point if you like. Alternatively, you can print out the memory corresponding to the floating point values using code similar to show\_bytes in Figure 2.4

15. Suppose we wish to write a procedure that computes the inner product of two vectors u and v. An abstract version of the function has a CPE of 16–17 with x86-64 and 26–29 with IA32 for integer, single-precision, and double-precision data. By doing the same sort of transformations we did to transform the abstract program combine1 into the more efficient combine4, we get the following code:

1 /\* Accumulate in temporary \*/

2 void inner4(vec\_ptr u, vec\_ptr v, data\_t \*dest)

3 {

4 long int i;

5 int 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 }

Our measurements show that this function has a CPE of 3.00 for integer and floating-point data. For data type float, the x86-64 assembly code for the inner loop is as follows:

inner4: data\_t = float udata in %rbx, vdata in %rax, limit in %rcx,i in %rdx, sum in %xmm1

1 .L87: loop:

2 movss (%rbx,%rdx,4), %xmm0 Get udata[i]

3 mulss (%rax,%rdx,4), %xmm0 Multiply by vdata[i]

4 addss %xmm0, %xmm1 Add to sum

5 addq $1, %rdx Increment i

6 cmpq %rcx, %rdx Compare i:limit

7 jl .L87 If <, goto loop

A. Diagram how this instruction sequence would be decoded into operations and show how the data dependencies between them would create a critical path of operations, in the style of Figures 5.13 and 5.14.

To diagram this instruction sequence like Figures 5.13 and 5.14, The first step is to label each of the different registers on the top, %rax, %rbx, %rcx, %rdx, %xmm0, and %xmm1.

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

Then the next step is write out how the instructions are executing, in terms of loading, storing, multiplying, dividing, adding, comparing, branching, or jumping. In this problem:

- movss is basically a load operation

- mulss is basically two steps, loading the value and multiplying

- addss is one step, an add operation

- addq is one step, an add operation

- cmpq is one step, a comparison

- jl is one step, a jump if less than operation.

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

load

load

mul

add

add

cmp

jump

movss

mulss

addss

cmpq

addss

jl

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

To draw the first diagram, if the object needs the value in that operation then it points to that operation. If the operation has values that come out and are used in other operations, that also needs to be documented.

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

load

load

mul

add

add

cmp

jump

movss

mulss

addss

addq

cmpq

jl

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

For modeling with Figure 5.14, the arrows will stay the same except there's more organization on how data flows:

%rax

%rbx

%rcx

%rdx

%xmm00

%xmm1

addss

mulss

load

load

addq

cmpq

jl

%rdx

%xmm1

B. For data type float, what lower bound on the CPE is determined by the critical path?

4 + 3 / 3= 7 / 3 = 2.33

C. Assuming similar instruction sequences for the integer code as well, what lower bound on the CPE is determined by the critical path for integer data?

3 + 1 / 3 = 4 / 3 = 1.33

D. Explain how the two floating-point versions can have CPEs of 3.00, even though the multiplication operation requires either 4 or 5 clock cycles.

5.16 Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling. For x86-64, our measurements of the unrolled version give a CPE of 2.00 for integer data but still 3.00 for both single and double precision.

A. Explain why any version of any inner product procedure cannot achieve a

CPE less than 2.00.

1 /\* Accumulate in temporary \*/

2 void inner4(vec\_ptr u, vec\_ptr v, data\_t \*dest)

3 {

4 long int i;

5 int 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 }

5.17 Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling with four parallel accumulators. Our measurements for this function with x86-64 give a CPE of 2.00 for all types of data.

A. What factor limits the performance to a CPE of 2.00?