a. (20) Diagram how this instruction sequence would be decoded into operations and show the data dependencies between them. Use Figure 5.14 as a guide. Include your diagram in your solutions document.

The x86-64 assembly code for the inner loop is as follows:

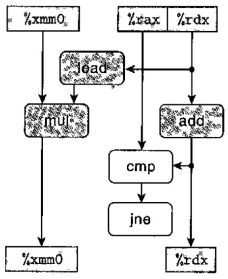
```
# u in %rbx, v in %rax, length in %rcx, i in %rdx, sum in %xmm1
.L87:
    movss (%rbx, %rdx, 4), %xmm0  # Get u[i]
    mulss (%rax, %rdx, 4), %xmm0  # Multiply by v[i]
    adds %xmm0, %xmm1             # Add to sum
    addq $1, %rdx                 # Increment i
    cmpq %rcx, %rdx               # Compare i to length
    jl .L87                       # If <, keep looping
```

load load

Registers to include:
rax, rbx
rdx, rcx
xmm0, xmm1

From the textbook:

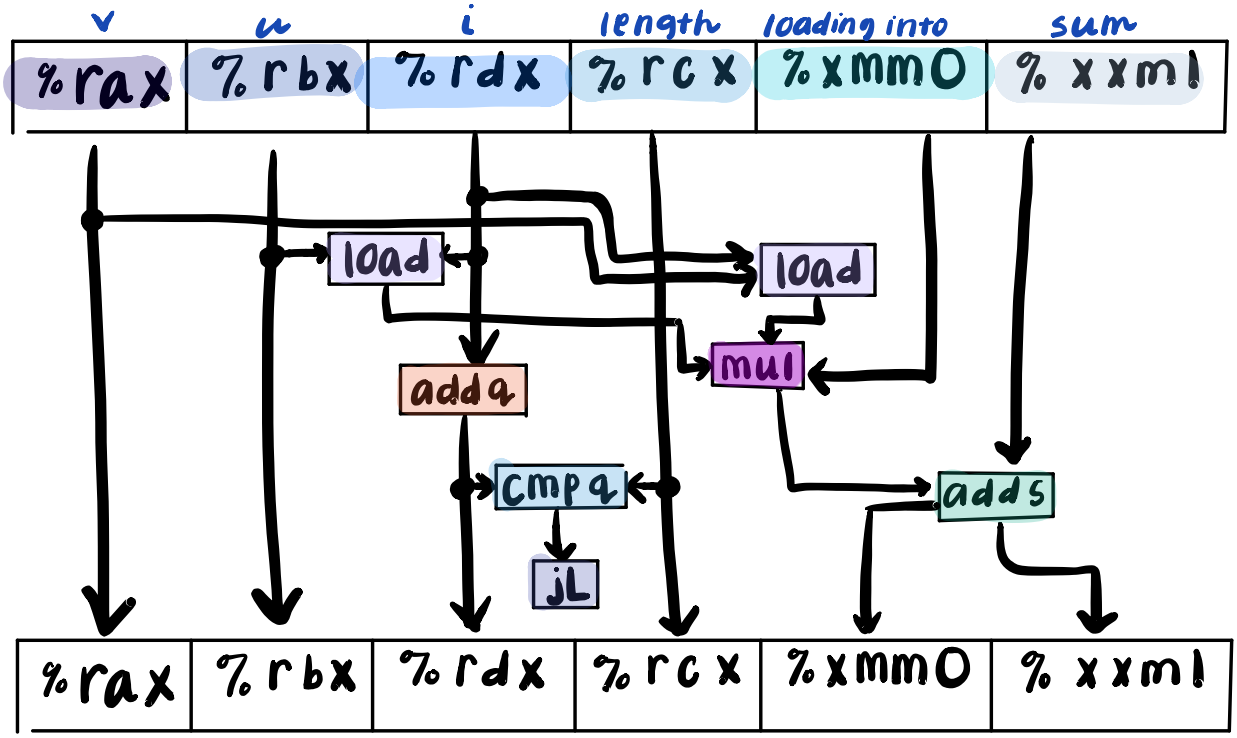
Figure 5.14 Abstracting `combine4` operations as a data-flow graph. We rearrange the operators of Figure 5.13 to more clearly show the data dependencies (a), and then further show only those operations that use values from one iteration to produce new values for the next (b).



(a)

following this example.

Instructions:
load
addq, adds
cmpq



1) 2)

b. (20) Which operation(s) in the loop can NOT be pipelined? Why? What are the latencies of these operations? 3) Based on this, what is the lower latency bound (in terms of CPE) of the procedure? Assume that *float* addition has a latency of 3, *float* multiplication has a latency of 5, and all integer operations have a latency of 1. Hint: think about which operation(s) depend on a result from the previous loop iteration. Write your answers in your solutions document.

1) The operations in the loop that cannot be pipelined are those that rely on the previous / returned values. An example of this in the scenario above would be the incrementer i → %rdx, and the two add instructions use i to iterate through the loop and compare require i so these all can NOT be pipelined.

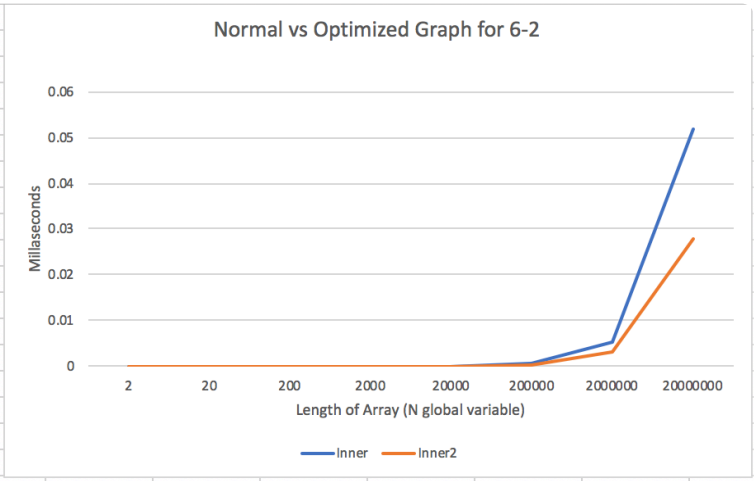
to reiterate, the add instructions are what cannot be pipelined.

2) The two latencies for these operations are 1 for the integer latency (adds) and 3 for the float addition latency (addq).

3) The lower bound latency is 3.

d. (20) Using your code from part c, collect data on the execution times of *inner* and *inner2* with varying array lengths. Summarize your findings and argue whether *inner* or *inner2* is more efficient than the other (or not). Create a graph using appropriate data points to support your argument. Include your summary and graph in your solutions document.

Length of Array	Inner	Inner2
2	0	0
20	0	0
200	0.000001	0
2000	0.000006	0.000003
20000	0.000058	0.000028
200000	0.000533	0.000287
2000000	0.005343	0.003174
20000000	0.051957	0.027708



Based on the collected data, my inner2 function
compared to inner. As the length of the array in
apparent as its difference grows exponentially. I