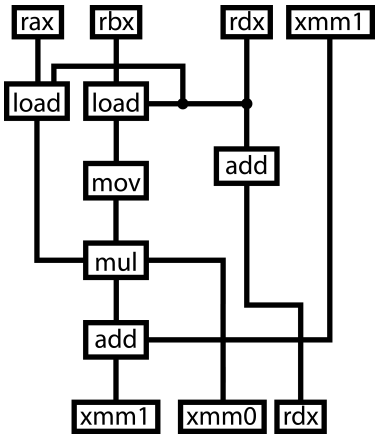
CS4400 *Alex Clemmer, u0458675*

Problem Set 6

1 Problem 5.15

1. Diagram of data dependencies:



2. Lower bound on the CPE for `float` type:

> 6.

3. Lower bound on the CPE for integral types:

> 3.

4. Why in practice is the CPE 3.00 when floating point multiplication operations take 4 or 5 cycles apiece?

> Throughput is smaller than the specific yield, and we take advantage of this. Specifically, in pipelining our floating point operations, we increase the utilization factor of our hardware. We "stack" the floating point operations on top of each other, which has the observed effect of dramatically reducing total latency.

2 Problem 5.16

The code for the four-way unrolled loop:

```
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long int i;
    int length = vec_length(u);
    int limit = length-4;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t acc = (data_t) 0;

    for (i = 0; i < limit; i+=4) {
        acc = acc + ((udata[i] * vdata[i]) *
                     (udata[i+1] * vdata[i+1]) *
                     (udata[i+2] * vdata[i+2]) *
                     (udata[i+3] * vdata[i+3]));
    }

    for(; i < length; i++) {
        acc = acc + (udata[i] * vdata[i]);
    }
    *dest = acc;
}
```

1. The Load unit can load only one value from memory each cycle. Computing the inner product of two different arrays will always require two loads from memory, and thus, they will at least have a 2 CPE minimum bound.

2. Mainly the problem is that we can't take advantage of a lot of tricks that we could in the case of integers (specifically reassociation transformation) because floating point arithmetic operations are not associative. In this case, we are left with vanilla pipelining, but since we're already maximizing our utilization factor here, we don't actually gain any CPEs by unrolling the loop (though we may still gain a performance advantage).

3 Problem 5.17

The four-way unrolled loop with 4 accumulators:

```
void inner4(vec_ptr u, vec_ptr v, data_t *dest)
{
    long int i;
    int length = vec_length(u);
    int limit = length-4;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t acc0 = (data_t) 0;
    data_t acc1 = (data_t) 0;
    data_t acc2 = (data_t) 0;
    data_t acc3 = (data_t) 0;

    for (i = 0; i < limit; i+=4) {
        acc0 = acc0 + (udata[i] * vdata[i]);
        acc1 = acc1 + (udata[i+1] * vdata[i+1]);
        acc2 = acc2 + (udata[i+2] * vdata[i+2]);
        acc3 = acc3 + (udata[i+3] * vdata[i+3]);
    }

    for(; i < length; i++) {
        acc0 = acc0 + (udata[i] * vdata[i]);
    }
    *dest = acc0 * acc1 * acc2 * acc3;
}
```

1. Same answer basically as 5.16.a. Loading values from memory. The Load unit can load only one value from memory each cycle.

2. Almost certainly register spilling. IA32 has a small number of registers to hold accumulators, and the only option is to "spill" the extras onto the stack.

4 Problem 5.22

Which of the two choices should we optimize?

Let's say we're doing 100 "units" (for some arbitrary unit) of work. If we speed up some part of the pipeline by a factor of x, then the total amount of work to be done decreases by the same factor. Particularly, if we can speed up a process that accounts for 30% of the total work (*i.e.*, 30 units) by a factor of 3, then that portion of work takes $\frac{30}{3} = 10$ work units. Similarly, if we speed up a process that accounts for 50% of the total work by a factor of 1.5, then the total work to take at present is $\frac{50}{1.5} = 33\frac{1}{3}$ work units.
Using simple addition, we can see that for the speedup of the 30%, our total work goes from 100 to $10 + 70 = 80$ work units. The total work of the speed up over 50% of the work goes from 100 to $33\frac{1}{3} + 50 = 83\frac{1}{3}$. **If our goal is to maximize speed: we pick B**.