Question 1: Pipelining, cont.

map:    beq $a0, $0, done
loop:   lw $t1, 0($a0)
        mul $t1, $t1, $a1
        lw $t2, 4($a0)
        add $t2, $t2, $t1
        sw $t2, 8($a0)
        lw $a0, 12($a0)
        bne $a0, $0, loop

        stall | forwA | forwB
            |       |       |
        c 1   b 10  a 11
        a 2   a 12
        c 3   b 13  a 14
        a 4   a 15  d 16
        a 5   a 17
        c 6   b 18  a 19

done:   jr $ra

Part (d) Compute how many cycles each loop iteration takes on average. Explain your answer for partial credit. (10 points)

Loop on average takes:
7 instructions +
6 stalls (3 of 2 cycles each) +
2 flushes (taken branch) =
15 cycles on average

Part (e) Re-schedule/re-write the function to make it faster. Faster code will achieve more points, but your answer must fit in the space below. (10 points)

map:    beq $a0, 0, done
loop:   lw $t1, 0($a0)
        lw $t2, 4($a0)
        lw $t3, 12($a0)
        mul $t1, $t1, $a1
        add $t2, $t2, $t1
        sw $t2, 8($a0)
        move $a0, $t3
        bne $a0, 0, loop

done:   jr $ra
Q2) struct unicorn is 6*4B = 24B

a) the best possible case is when all of the hooves point at the same hoof. After the first iteration which brings the hoof into the cache, we just get misses on the unicorn structure. 24B struct / 32B cache block = 3/4 of a miss per iteration.

b) The worst possible case is when the hooves all point to different memory locations and the hooves span cache block boundaries (so that ‘has_horseshoe’ and ‘shoe_size’ are in different cache blocks).

Miss on unicorn ¾ time.
Miss on unicorn[i].hooves[0]->has_horseshoe.
Miss on unicorn[i].hooves[0]->shoe_size.
Miss on unicorn[i].hooves[1]->has_horseshoe.
Miss on unicorn[i].hooves[1]->shoe_size.
Miss on unicorn[i].hooves[2]->has_horseshoe.
Miss on unicorn[i].hooves[2]->shoe_size.
Miss on unicorn[i].hooves[3]->has_horseshoe.
Miss on unicorn[i].hooves[3]->shoe_size.
8.75 misses/iteration.

Q3 a) 16KB cache with 32B blocks has 512 cache blocks. In original code, A traversal is ¼ or ½ miss per iteration (if A is 4 byte or 8 byte type, respectively) and B traversal is 2 misses per iteration (column-wise traversal, 5000 >> 512)

Just loop fission, followed by inverting the traversal of B.

```c
for (int i = 0 ; i < N ; i ++) {
    for (int j = 0 ; j < N ; j += 2) {
        A[i][j] = A[i][j+1];
    }
}

for (int j = 0 ; j < N ; j += 2) {
    for (int i = 0 ; i < N ; i ++) {
        B[j][i] = B[j+1][i];
    }
}
```

Now A loop is still ¼ or ½ miss and B loop is also ¼ or ½ miss per iteration (based on type size as above.)
Q3 b) Still 512 cache blocks in cache. In original code, no misses on i, j, temp registers. No misses on B[i][0] (reuse, 2-way set associative so can tolerate occasional conflicts with A[k][j][i]). 1 miss/iteration on A[k][j][i]; no locality. 1/N misses per iteration on C[i][j] (no locality, because A[k][j][i] overflows cache each inner loop). Alternatively, N misses on B, N^3 misses on A, N^2 misses on C.

First step, undo the use of the temporary variable to simplify code.

```c
for (int i = 0 ; i < N ; i ++) {
    for (int j = 0 ; j < N ; j ++) {
        C[i][j] = 0.0; // N^2/4 misses
        for (int k = 0 ; k < N ; k ++) {
            // 0 + N + N^3 misses
            C[i][j] += B[i][0] * A[k][j][i];
        }
    }
}
```

Using stripmining to separate i loop into two loops

```c
#define TILESIZE = 20;
for (int i = 0 ; i < N ; i += TILESIZE) {
    for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
        for (int j = 0 ; j < N ; j ++) {
            C[ii][j] = 0.0; // N^2/4 misses
            for (int k = 0 ; k < N ; k ++) {
                C[ii][j] += B[ii][0] * A[k][j][ii];
            }
        }
    }
}
```

Do loop fission to split out the initialization of C.

```c
for (int i = 0 ; i < N ; i += TILESIZE) {
    for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
        for (int j = 0 ; j < N ; j ++) {
            C[ii][j] = 0.0;
        }
    }
}
```

```c
for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
    for (int j = 0 ; j < N ; j ++) {
        C[ii][j] += B[ii][0] * A[k][j][ii];
    }
}
```
do loop inversion between the ii and j,k loops to complete the tiling

```c
for (int i = 0 ; i < N ; i += TILESIZE) {
    for (int j = 0 ; j < N ; j ++) {
        for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
            C[ii][j] = 0.0;   // still N^2/4 misses
            // we have reuse, because 20 blocks should fit
        }
    }
    for (int j = 0 ; j < N ; j ++) {
        for (int k = 0 ; k < N ; k ++) {
            for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
                // N^2/4     +     N      +   N^3/4 misses
                C[ii][j] +=  B[ii][0] * A[k][j][ii];
            }
        }
    }
}
```

as it stands, we now have: \(N^3/4 + N^2/2 + N\) misses

compared to the original: \(N^3 + N^2 + N\) misses

we can reduce it slightly more by loop fusing the two j loops together

```c
for (int i = 0 ; i < N ; i += TILESIZE) {
    for (int j = 0 ; j < N ; j ++) {
        for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
            C[ii][j] = 0.0;   // still N^2/4 misses
        }
        for (int k = 0 ; k < N ; k ++) {
            for (int ii = i ; ii < (i+TILESIZE) ; ii ++) {
                // 0     +     N      +   N^3/4 misses
                C[ii][j] +=  B[ii][0] * A[k][j][ii];
            }
        }
    }
}
```

down to: \(N^3/4 + N^2/4 + N\) misses

now a single iteration of the j loop does:

- loads 20 cache blocks containing elements of the C matrix
- keeps those 20 cache blocks fresh in the cache in the second loop
- reuses the same 20 cache blocks containing elements of B, keeping them fresh in the cache
- walks 5000 20-element contiguous lengths of matrix A with good spatial locality. Since we touch the C and B element during each of these 20-element lengths, the segments of A shouldn’t displace the elements of C and B.