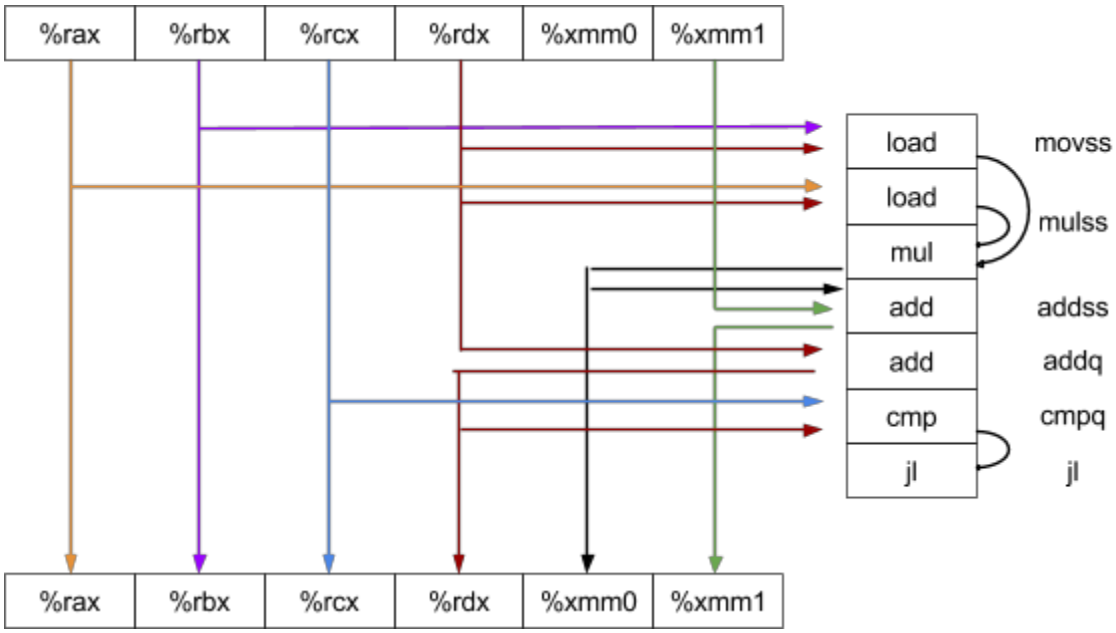
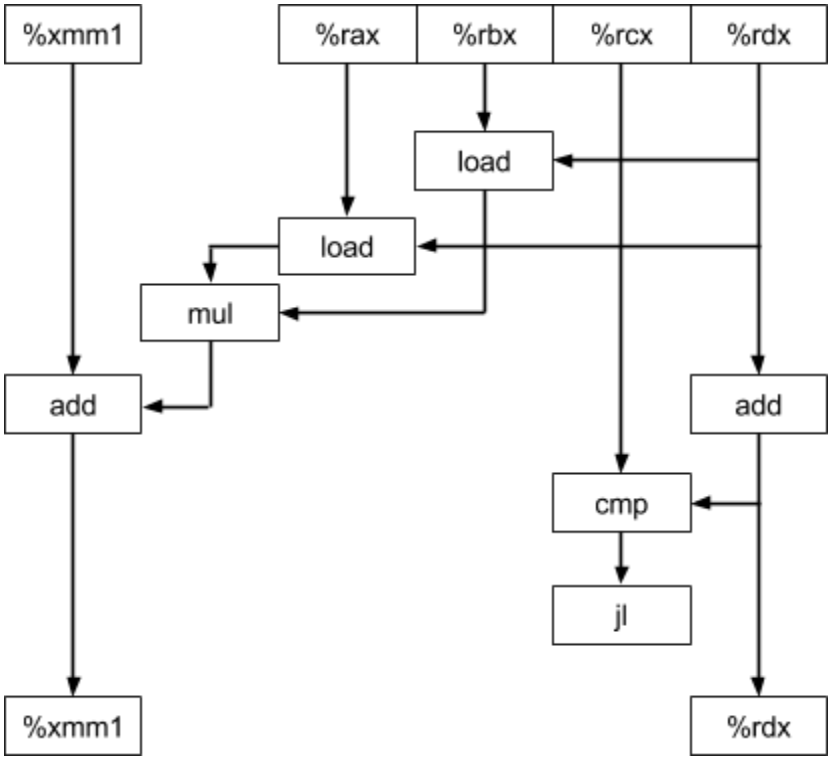
CSCI 2021 Spring 2015
Assignment IV Solutions

Problem 1.

A.





B. For data type `float`, the lower bound is 3 cycles.

C. For data type `int`, the lower bound is 1 cycle.

D. 2 floating point operations can have CPE of 3.00 because they are not on the critical path.

Problem 2

Part A

```
1  /* Unroll loop by 4, 4-way parallelism, reassociation  */
2  void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3  {
4     long int i;
5     int length = vec_length(u);
6     int limit = length-3;
7     data_t *udata = get_vec_start(u);
8     data_t *vdata = get_vec_start(v);
9     data_t sum0 = (data_t) 0;
10    data_t sum1 = (data_t) 0;
11    data_t sum2 = (data_t) 0;
12    data_t sum3 = (data_t) 0;
13
14    /* Accumulate 4 elements at a time */
15    for (i = 0; i < limit; i+=4) {
16         sum0 = sum0 + (udata[i] * vdata[i]);
17         sum1 = sum1 + (udata[i+1] * vdata[i+1]);
18         sum2 = sum2 + (udata[i+2] * vdata[i+2]);
19         sum3 = sum3 + (udata[i+3] * vdata[i+3]);
20    }
21
22    /* Finish off any remaining elements in case length % 4 != 0 */
23    for (; i < length; i++)
24         sum0 = sum0 + (udata[i] * vdata[i]);
25
26    *dest = sum0 + sum1 + sum2 + sum3;
27    }
```

Reassociation adds parentheses to define the order of operations explicitly. However, just like in normal arithmetic, in C, multiplication has a higher precedence compared to addition so there is no need to reassociate (although I have added parenthesis anyways to show how it could be done). But, even without the parenthesis, the order of operations will be the same.

Part B

Fetch	Integer	Floating Point
Without re-association	add	add
With re-association	add	add

The multiplication in each iteration can be performed w/out waiting for the accumulated value from the previous iteration. However, the sum variable creates a data dependency due to the accumulator via addition.

The critical path is used to get a latency-based lower bound on the code's performance. An integer would give ¼ CPE lower bound (b/c integers are low bounded by 1 cycle--see #1) an a float would give ¾ CPE lower bound (b/c floats are low bounded by 3 cycles--see #1). Each element will require 2 loads and the processor can only issue 1 load per cycle => each element requires at least 2 cycles. Therefore, the limiting factor is from the limitation on the throughput of the load unit.

See the figure shown on pg. 522 for a very similar problem. It has a data-flow diagram and shows the critical path.

Problem 3.

C = S * E * B
m = s + t + b
ln(S) = s
ln(B) = b

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	**32**	**1024**	**4**	**4**	**64**	**24**	**6**	**2**
2	**32**	**1024**	**4**	**256**	**1**	**30**	**0**	**2**
3	**32**	**1024**	**8**	**1**	**128**	**22**	**7**	**3**

4	**32**	**1024**	**8**	**128**	**1**	**29**	**0**	**3**
5	**32**	**1024**	**32**	**1**	**32**	**22**	**5**	**5**
6	**32**	**1024**	**32**	**4**	**8**	**24**	**3**	**5**

Problem 4.

64 KB = 64,000 bytes.
Direct mapped means it is 1-way associative (E = 1, one line per set)
*Each line is 4 bytes => block size = 4 bytes. Note: this is discounting the size of the tag and valid bit. Unless the problem explicitly says not to, you can discount the tag and valid bit size which is why the equation C = S * E * B holds.*
4-byte block means each block can store 1 pixel struct.

(a) There is always a cold miss (accessing the .r field) because the desired data is not in the cache. After fetching from memory, the next 3 accesses (.g, .b, .a) will be cache hits because they were in the block that was fetched.
Miss rate: 1/4 = 25%

(b) The array traversal in this problem is via pointer arithmetic (a subject that was introduced in the beginning of the semester). You'll notice that the access pattern is the similar as the one above—they're just using pointers here, which is why the problem may seem more complex.
Miss rate: 1/4 = 25%

(c) Each access is for 4 bytes since it's using an integer pointer. Since a block can only hold 4 bytes, there will always be a compulsory miss.
Miss rate: 1/1 = 100%

(d) Loop C. All 4 bytes are traversed at once in each iteration.

Problem 5.

$Cache\ size\ = 64K = 64 * 1024 = 2^{16} bytes$

```
void copy_matrix(int dest[ROWS][COLS], int src[ROWS][COLS])
{
      int i, j;
      for (i=0; i<ROWS; i++) {
            for (j=0; j<COLS; j++) {
                  dest[i][j] = src[i][j];
            }
      }
}
```

1. What is the cache miss rate if ROWS = 128 and COLS = 128?
   $Matrix\ size\ = 2^7 * 2^7 * 2^2 = 2^{16} bytes\ = size\ of\ the\ cache$

   `dest[i][j]` and `src[i][j]` will be mapped to the same cache line.
   Miss rate = 100%
2. What is the cache miss rate if ROWS = 128 and COLS = 192?
   $Matrix\ size\ = (2^7 * 2^7 + 2^7 * 2^6) * 2^2 = 2^{16} + 2^{15} bytes = 1.5 *\ size\ of\ the\ cache$

   `dest[i][j]` and `src[i][j]` will be mapped to a different cache line. The access pattern utilizes the spatial locality (1 miss, 3 hits per cache line).
   Miss rate = 25%
3. What is the cache miss rate if ROWS = 128 and COLS = 256?
   $Matrix\ size\ = (2^7 * 2^8) * 2^2 = 2^{17} bytes = 2 *\ size\ of\ the\ cache$

   `dest[i][j]` and `src[i][j]` will be mapped to the same cache line.
   Miss rate = 100%

```
void copy_n_flip_matrix1(int dest[ROWS][COLS], int src[ROWS][COLS])
{
      int i, j;
      for (i=0; i<ROWS; i++) {
            for (j=0; j<COLS; j++) {
                  dest[i][COLS - 1 - j] = src[i][j];
            }
      }
}
```

4. What is the cache miss rate if ROWS = 128 and COLS = 128?
   `dest[i][COLS-1-j]` and `src[i][j]` will be mapped to a different cache line. The access pattern utilizes the spatial locality (1 miss, 3 hits per cache line).
   Miss rate = 25%

5. What is the cache miss rate if ROWS = 128 and COLS = 192?
   `dest[i][COLS-1-j]` and `src[i][j]` will be mapped to a different cache line. The access pattern utilizes the spatial locality (1 miss, 3 hits per cache line).
   Miss rate = 25%

```
void copy_n_flip_matrix2(int dest[ROWS][COLS], int src[ROWS][COLS])
{
      int i, j;
      for (j=0; j<COLS; j++) {
            for (i=0; i<ROWS; i++) {
                  dest[i][COLS - 1 - j] = src[i][j];
            }
      }
}
```

6. What is the cache miss rate if ROWS = 128 and COLS = 128?
   Each element that is accessed will be mapped to a new cache line, but the access pattern utilizes the spatial locality.
   Miss rate = 25%
7. What is the cache miss rate if ROWS = 192 and COLS = 128?
   Miss rate = 75%

Problem 6.
Each *pixel_t* has a size of 4 bytes. The pixel array is 16x16 and holds elements of type *pixel_t*.

(a) 8-byte blocks mean that each block can store 2 pixel_t structs.
The first access to pixel[0][0].r results in a cold miss because the cache is empty. This fetches a block which contains pixel[0][0] and pixel[0][1]. That means the rest of the accesses for pixel[0][0] will be a hit as well as the entirety for pixel[0][1].
However, the next pixel will require fetching another block due to a cold miss. We can see a pattern here: there is a cold miss and then 7 subsequent cache hits.
Miss rate: 1/8 = **12.5%**

(b) Set associativity impacts the performance in terms of how long it may take to access data since associativity determines how the cache is laid out and thus how indexing will take place. However, it does not affect the number of hits/misses. ***What is important here is that the block size is halved***. Remember, blocks are what is transferred between levels of cache. Therefore, if we can cache less data, we should expect the cache misses to increase. Halving the block size in this straightforward access pattern leads to doubling the cache misses.

Miss rate: 1/4 = 25%

(c) 8-byte blocks. Arrays, as noted in the problem, are sorted in row-major order, so the elements inside blocks will be fetched in row-major order. However, note that the access pattern in this problem is column-wise. Therefore, there will be a cold miss in trying to access pixel[0][0].r and the next 3 accesses for the struct will be hits after being fetched frim memory. But since the access pattern is columnwise, the next element to be accessed is pixel[1][0] which is not in the cache! This will result in a cache miss followed by 3 cache hits after being fetched from memory.
Miss rate: 2/8 = 25%

(d) Once again, we can only fit 1 pixel struct in a block at a time. Therefore, column-wise access doesn't have an effect. There will just be misses followed by cache hits.
Miss rate: 1/4 = 25%

Problem 7.

$$Miss\ rate\ =\ Total\ number\ of\ misses\ /\ Total\ number\ of\ accesses$$

Total number of accesses:

- # accesses for `array[i][j]:` $1023 + 1022 + ... + 1 = 1023\ (1023 + 1) / 2$
- # accesses for `array[i-1][j-1]:` $1023 + 1022 + ... + 1 = 1023\ (1023 + 1) / 2$
- # accesses for `array[i-1][j]:` $1023 + 1022 + ... + 1 = 1023\ (1023 + 1) / 2$

$$Total\ number\ of\ accesses\ =\ 3 * 1023 * (1023 + 1) / 2$$

Question 1.

$$Cache\ size = 64KB = 64 * 1024\ = 2^{16}$$

$$B = 32 = 2^5$$

$$E = 2$$

$$S = 2^{16} / 2^6 = 2^{10} = 1024$$

Total number of misses:

- # misses for `array[i][j]:`

$$(128 * 7) + [(127 * 8) + (126 * 8) + ... + (1 * 8)]$$

$$(128 * 7) + 8 * (127\ (127 + 1) / 2)$$

$$(128 * 7) + 4 * (127 * 128)\ =\ 515 * 128$$

- # misses for `array[i-1][j-1]:128`

- # misses for `array[i-1][j]:` 0 (since the data is already loaded by the access to `array[i-1][j-1]`)

$$Total\ number\ of\ misses\ = (515 * 128) + 128 = (516 * 128)$$

$$Miss\ rate = (516 * 128) / (3 * 1023 * 512) = 4.2\%$$

Question 2.

$$Cache\ size = 3KB = 3 * 1024$$

$$B = 32 = 2^5$$

$$E = 2$$

$$S = (3 * 2^{10}) / 2^6 = 3 * 2^4 = 48$$

1 Row has 1024 int, 8 ints per set, will thus use the 48 sets, more than twice.
Which means that access to all elements of a row will fill up the cache.
So for `array[i-1][j-1]` and `array[i-1][j]` don't benefit from the loading of `array[i][j]` until fits into 48 sets and less or (48 * 8 = 384 elements).

Total number of misses:

- # misses for `array[i][j]:`

$$(128 * 7) + [(127 * 8) + (126 * 8) + ... + (1 * 8)]$$

$$(128 * 7) + 8 * (127\ (127 + 1) / 2)$$

$$(128 * 7) + 4 * (127 * 128)\ =\ 515 * 128$$

- # misses for `array[i-1][j]:`

$$(128 * 7) + [(127 * 8) + (126 * 8) + ... + (49 * 8)]$$

$$(128 * 7) + 8 * [(127 + 49) * 79 / 2]$$

$$(128 * 7) + 4 * (176 * 79) = (128 * 7) + (176 * 316)$$

- # misses for `array[i-1][j-1]:` 0

$$Total\ number\ of\ misses\ = (515 * 128) + (128 * 7) + (176 * 316)$$

$$Miss\ rate = ((522 * 128) + (176 * 316)) / (3 * 1023 * 512) = 7.79\%$$

Question 3.
Use blocking to reduce cache misses for 3KB cache.