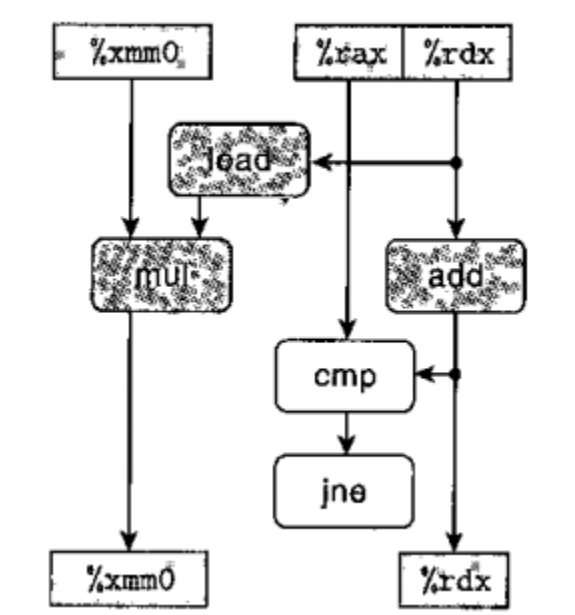
1. For the following assembly, draw a data flow graph and identify the critical path.

```
.L1:
        vmulsd      (%rdx), %xmm0, %xmm0
        addq        $8, %rdx
        cmpq        %rax, %rdx
        jne         .L1
```

Given the following table, what is the lowest bound latency to execute n iterations of this loop for integer operations? Floating point operations?

Latency Table (CPE)	Integer	Double
Arithmetic (except mul)	1	3
Multiply	2	4
Load Store	1	1



The critical path is determined by the multiply operation in the loop. For integer operations, from the table, we can see that the latency is 2 cycles for multiplication and for n iterations of the loop, it would take 2n cycles to execute. Similarly for floating point operations, the latency is 4 cycles, and it would take 4n cycles to execute n iterations of the loop.

1. Suppose our memory can store 64 bytes of data, addressed between 6'b000000 and 6'b111111, and that we have stored a 4 x 4 array of ints in memory. This could be declared by:

```
int arr[4][4] = {
      {0,1,2,3},
      {4,5,6,7},
      {8,9,10,11},
      {12,13,14,15}
};
```

In addition, suppose we have a **fully associative cache** (only one set) with a single line of **16 bytes**.

Then, our addresses can be partitioned into **2 tag bits** along with **4 block offset bits**. For example, 6'b000000 would be formatted as:

tag: 00	block offset: 0000

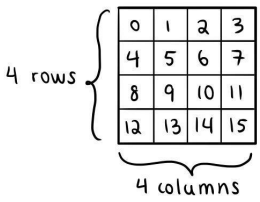
A. Now, say we accessed the array using the following loop, and our cache is cold (empty) to start. How many cache hits/misses would there be? (Think about how the array is stored in memory, and how data will be cached).

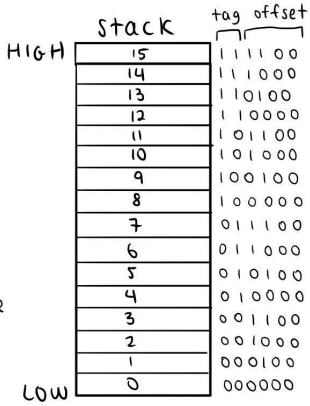
```
let sum = 0;
for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
            sum += arr[i][j];
      }
}
```

B. Again, assume our cache is empty to start. How many cache hits/misses would there be if we used this loop instead?

```
let sum = 0;
for (int j = 0; j < 4; j++) {
      for (int i = 0; i < 4; i++) {
            sum += arr[i][j];
      }
}
```

we can imagine our array is stored in memory like this:





Recall that an int is 4 bytes, hence the binary addresses indicated

our cache looks like this:

set	valid	tag	block of 16 bytes
0	0		

now consider the loops:

A) accesses in row major order...
- → first read at arr[0][0] (MISS) stores the first row in the cache, so reads arr[0][1] through arr[0][3] are cache hits!
- → so there is 1 miss and 3 hits for every 4 accesses for a total of 4 misses, 12 hits

B) accesses in column major order
- → first read at arr[0][0] (MISS) caches arr[0][0] through arr[0][3]
- → however, our next read is at arr[1][0]! so we have another miss, and evict the first row to cache arr[1]
- → ... repeat ä
- → 16 cache misses, 0 hits

2. Suppose we have a direct-mapped cache with **2 sets, 1 line per set, 4 bytes per line**, and 4 bit addresses that look like:

tag bits: t = 1	set index: s = 1	offset bits: b = 2

Then our address space will look like:

Address (decimal)	Tag bits	Index bits	Offset bits
0	0	0	00
1	0	0	01
2	0	0	10
3	0	0	11
4	0	1	00
5	0	1	01
6	0	1	10
7	0	1	11
8	1	0	00
9	1	0	01
10	1	0	10
11	1	0	11
12	1	1	00
13	1	1	01
14	1	1	10
15	1	1	11

Initially, our cache is cold (empty), and we may represent it like this:

Set	**Valid**	**Tag**	**Byte 0**	**Byte 1**	**Byte 2**	**Byte 3**
0	0					
1	0					

A. Now, suppose we want to read **1-byte words** from memory, where the word located at memory address i is indicated by M[i]. What happens to the cache when the CPU performs the following sequence of reads?

read words at memory address: 0, 14, 3, 11

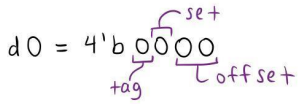
✯ Good things to note:

- first, we note that our blocks (data that gets cached together) are uniquely identified by the concatenation of the tag and index bits
  → blocks 2'b00, 2'b01, 2'b10, 2'b11
  d0 d1 d2 d3
- furthermore, we have two cache sets (each capable of fitting one block) and 4 blocks, which means that multiple blocks must map to the same cache set (same set index)
  ↳ these blocks are differentiated by their tag bits

initially, our cache is cold:

SET	VALID	TAG	B0	B1	B2	B3
0	0					
1	0					

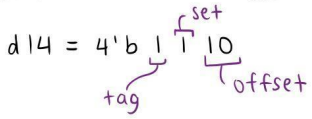
read word at address 0:



set 0's line has a 0 valid bit, so cache miss! fetch block 0 from memory and store in set 0

SET	VALID	TAG	B0	B1	B2	B3
0	1	0	m[0]	m[1]	m[2]	m[3]
1	0					

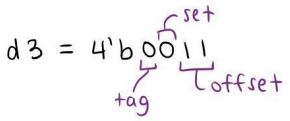
read word at address 14:



set 1's line has a 0 valid bit, so cache miss! fetch block 3 from memory and store in set 1

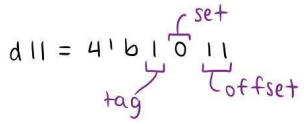
SET	VALID	TAG	B0	B1	B2	B3
0	1	0	m[0]	m[1]	m[2]	m[3]
1	1	1	m[12]	m[13]	m[14]	m[15]

read word at address 3:



set 0's line has a valid bit of 1, and the tags match, so we have a cache hit! the cache's state does not change

read word at address 11:



set 0's line has a valid bit of 1, but the tags do not match, so we have a cache miss! we must evict the data from set 0 and cache block 2 from memory

SET	VALID	TAG	B0	B1	B2	B3
0	1	1	m[8]	m[9]	m[10]	m[11]
1	1	1	m[12]	m[13]	m[14]	m[15]