# 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

$\boxed{1.1}$ SIMD is a form of instruction-level parallelism.

$\boxed{1.2}$ SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).

$\boxed{1.3}$ Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

$\boxed{1.4}$ Each hardware thread in the CPU uses a shared cache.

$\boxed{1.5}$ The number of hardware threads available can be more than the number of processor cores on the computer.

$\boxed{1.6}$ In thread-level parallelism, the amount of speedup is directly proportional to the increase in number of hardware threads.

2   Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.
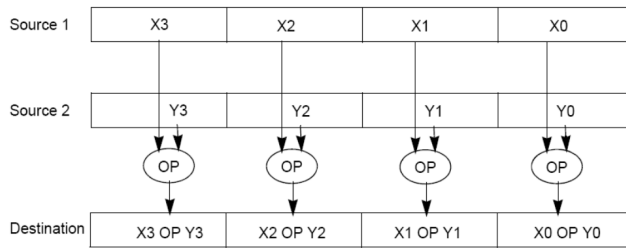
Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- __m128i _mm_set1_epi32(**int** i):

  Set the four signed 32-bit integers within the vector to i.
- __m128i _mm_loadu_si128(__m128i *p):

  Load the 4 successive ints pointed to by p into a 128-bit vector.
- __m128i _mm_mullo_epi32(__m128i a, __m128i b):

  Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- __m128i _mm_add_epi32(__m128i a, __m128i b):

  Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- **void** _mm_storeu_si128(__m128i *p, __m128i a):

  Store 128-bit vector a at pointer p.
- __m128i _mm_and_si128(__m128i a, __m128i b):

  Perform a bitwise AND of 128 bits in a and b, and return the result.
- __m128i _mm_cmpeq_epi32(__m128i a, __m128i b):

  The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

2.1   SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? Can we always SIMD-ize an entire array? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = _____;
    for (int i = 0; i < _____; i += ___) { // Vectorized loop
        prod_v = _____;
    }
    __mm_storeu_si128(_____, _____);
    for (int i = _____; i < _____; i++) { // Handle tail case
        result[0] *= _____;
    }
    return _____;
}
```

3   Thread-Level Parallelism

OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

- The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.

  ```
  #pragma omp parallel
  {
      ...
  }
  NOTE: The opening curly brace needs to be on a newline or else there
      will be a compile-time error!
  ```

- The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The following two code snippets are equivalent.

  ```
  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
      ...
  }
  ```

  ```
  #pragma omp parallel
  {
  #pragma omp for
      for (int i =0; i < n; i++) { ... }
  }
  ```

There are two functions you can call that may be useful to you:

- **int** `omp_get_thread_num()` will return the number of the thread executing the code

- **int** `omp_get_num_threads()` will return the number of total hardware threads executing the code

3.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume `arr` is an **int[]** of length n.

(a)
```
// Set element i of arr to i
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

(b)
```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

(c)
```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = 0;
```

(d)
```
// Set element i of arr to i;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    *arr = i;
    arr++;
```

3.2 What potential issue can arise from this code?

```
1  // Decrements element i of arr. n is a multiple of omp_get_num_threads()
2  #pragma omp parallel
3  {
4      int threadCount = omp_get_num_threads();
5      int myThread = omp_get_thread_num();
6      for (int i = 0; i < n; i++) {
7          if (i % threadCount == myThread) arr[i] -= 1;
8      }
9  }
```

4   Concurrency

The benefits of multi-threading programming come only after you understand concurrency. Here are two of the most common concurrency issues:

1. **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. This can be solved by bypassing the cache and writing directly to memory, i.e. using volatile keywords in many languages, or by using a coherency protocol such as MOESI.

2. **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-threading programming, the **interleaving** of R, M, W stages often produces a lot of issues.

4.1   MOESI Protocol

Parallel processing allows individual cores of a CPU to operate as independent units with their own caches. However, for this to be the case, the machine must be able to coordinate the information flow of all cores and all caches so that this information is reliable to some degree. Therefore, we impose **cache states**, composing of the **valid**, **dirty** and **shared** bits, to denote status of the cache data at a specific cache block. These cache states are used when there is a cache **miss** or **write** to a certain core's cache so that if the information is modified in one place, the other caches are informed. In summary, we don't want two caches with different data both saying that they have the most up-to-date data, because that simply can't be true. In other words, from the perspective of the **host processor**, their cache line states may be updated due to actions taken by **proxy processor** execution.

Consider this visual representation of the addressing of a cache block and the updated construction of the block itself:

Address		
Tag	Index	Offset

$\longrightarrow$

Contents			
State			
Valid	Dirty	Shared	Data
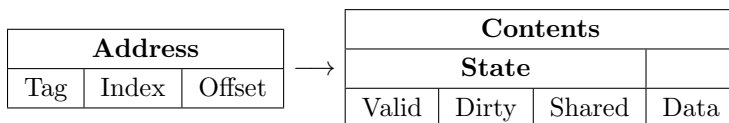
Each state describes a specific set of conditions, on a **single cache block**, in respect to the overall memory system(all caches and main memory).

4.1   Match all conditions below with their corresponding state(s).

Note: Some conditions can apply to multiple states!

(a) data in host cache up-to-date

(b) data in main memory is outdated

(c) data in main memory up-to-date

(d) if evicted, host cache must write this line's data back to main memory

(e) no copies exist in other (proxy) caches

(f) copies may exist in other (proxy) caches

(g) access from processor will result in a miss

1. **Modified**(M)

2. **Owned**(O)

3. **Exclusive**(E)

4. **Shared**(S)

5. **Invalid**(I)

4.2 Atomic Instructions

In order to solve the problems created by Read-modify-write, we have to rely on the idea of uninterrupted execution, also known as **atomic** execution.

In RISC-V, we have two categories of atomic instructions:

1. **Amoswap**: allows for uninterrupted memory operations within a single instruction

2. **Load-reserve, store-conditional**: allows us to have uninterrupted execution across multiple instructions

Both of these can be used to achieve atomic primitives. Here we'll focus on the former with this example:

```
Test-and-set

Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

amoswap rd, rs2, (rs1): Atomically, loads the word starting at address **rs1** into **rd** and puts **rs2** into memory at address **rs1**. Data races are avoided using the *aq* and *rl* flags, which *acquire* a lock that forces multiple threads to wait their turn until the lock is *released*.

Test-and-set: We have a lock stored at the address specified by `a0`. We utilize `amoswap` to put in 1 and get the old value. If the old value was a 1, we would not have changed the value of the lock and we will realize that someone currently has the lock. If the old value was a 0, we will have just "locked" the lock and can continue with the critical section. When we are done, we put a 0 back into the lock to "unlock" it.

4.2 | We've experimented with data synchronization across threads in C, but now let's take a look at how to parallelize and avoid data races in RISC-V!

We want to parallelize a program that finds the sum of the integers in an array pointed to by `a0` (array length = `a2`) and places it in memory at address `a1`. There is a free word of memory initialized to zero (i.e. result of `calloc(4, 1)`) pointed to by `a3`. For the sake of simplicity, assume there is a function `get_thread_num` that returns the current thread's number and a function `get_num_threads` that returns the total number of threads.

Here is some skeleton code to parallelize this operation. Note the use of amoswap. Before filling out the skeleton code, answer questions 4.3 and 4.4 first.

```
1       #Prologue
2       ...
3       mv s0 a0        #s0 points to the array
4       mv s1 a1        #s1 points to the global sum
5       mv s2 a2        #s2 has the length of array
6       jal get_num_threads
7       mv s3 a0        #s3 has the total number of threads
8       jal get_thread_num
9       mv s4 a0        #s4 has the current thread number
10      li t0 0
11  Loop:
12      bge _____
13      slli t1 s4 2
14      add t1 s0 t1    #index into array
15      lw t2 0(t1)
16      add t0 t0 t2    #add to local sum
17      add _____
18      j Loop
19  Exit:
20      _____
21  Try:
22      lw t1 0(a3)     #Check if work is being done in another thread
23      bnez t1 Try
24      amoswap.w.aq _____
25      _____
26      _____
27      add t2 t2 t0    #add local sum to total
28      _____
29
30      amoswap.w.rl _____
31      #Epilogue
32      ...
```

4.3   Why do we want to use an atomic instruction in our parallelized implementation?

4.4   Between which lines in the program above should threads start to run in parallel on separate copies of code? (Equivalent to where we put **#pragma** omp parallel in C)