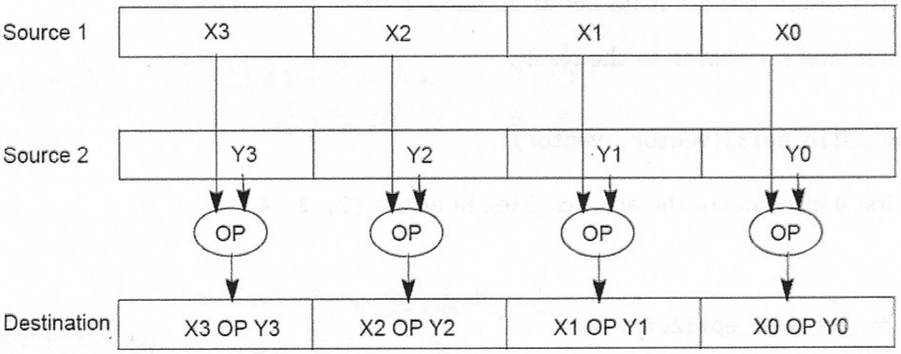
CS 61C
Fall 2018

Data-level Parallelism
Discussion 8: October 15, 2018

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, or 16 shorts/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)`:
  Return the 128-bit vector stored at pointer `p`.
- `__m128i _mm_mullo_epi32(__m128 a, __m128 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(__m128 a, __m128 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)`:
  Compare packed 32-bit integers in a and b for equality, set return vector to 0xFFFFFFFF if equal and 0 if not.

0.1 You have an array and 128-bit vector as follows:

```
int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
__m128i vector = _mm_loadu_si128((__m128i *) arr);
```

For each of the following tasks, fill in the correct arguments for each SIMD instruction, and where necessary, fill in the appropriate SIMD function. Assume they happen independently, i.e. the results of Part A do not at all affect Part B.

(a) Multiply vector by itself, and set vector to the result.

```
__m128i vector = _mm_mullo_epi32(vector, vector);
```

look at table to find a Match

(b) Add 1 to each of the first 4 elements of the arr, resulting in arr = {2, 3, 4, 5, 5, 6, 7, 8}

```
__m128i vector_ones = _mm_set1_epi32(1);
__m128i result = _mm_add_epi32(vector, vector_ones);
_mm_storeu_si128((__m128i *) arr, vector);
```

make vector of 1's

add our vector + the 1's vector

store it back to where we got it.

(c) Add the second half of the array to the first half of the array, resulting in arr = {1 + 5, 2 + 6, 3 + 7, 4 + 8, 5, 6, 7, 8} = {6, 8, 10, 12, 5, 6, 7, 8}

add the two vectors together

load in the second half of the array

```
__m128i result = _mm_add_epi32(_mm_loadu_si128((__m128i *) (arr + 4)), vector);
_mm_storeu_si128((m128i*) arr, result);
```

override the original front part of the array.

(d) Set every element of the array that is not equal to 5 to 0, resulting in arr = {0, 0, 0, 0, 5, 0, 0, 0}. Remember that the first half of the array has already been loaded into vector.

first half

```
__m128i fives = _mm_set1_epi32(5);
__m128i mask = _mm_cmpeq_epi32(vector, fives);
__m128i result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) arr, result);
```

second half

```
vector = _mm_loadu_si128((__m128i *) (arr + 4));
mask = _mm_cmpeq_epi32(vector, fives);
result = _mm_and_si128(mask, vector);
_mm_storeu_si128((__m128i *) (arr + 4), result);
```

make vector full of 5's.

compare the 5's vector to the input vector.

Remember, this will then output all 1's or all 0's ie 0xFFFFFFFF or 0. Because of this, we just re-apply the mask to get back just 5's or zeros.

Finally we store our result back to the array

This is Repeating what we did for the first part

0.2 Implement the following function, which returns the product of all of the elements in an array.

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

```
static int product_vectorized(int n, int *a) {
    int result[4];
  1 __m128i prod_v = __mm_set1_epi32(1);
  2 for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
  3     prod_v = __mm_mullo_epi32(prod _v, __mm_loadu_si128((__m128i *) (a + i)));
    }
 ~3 _mm_storeu_si128((__m128i *) result, prod_v);
  2 for (int i = n/4 * 4; i < n; i++) { // Handle tail case
  3     result[0] *= a[i];
    }
3 4 return result[0] * result[1] * result[2] * result[3];
}
```

Initialize stays the same,

← We prepare result

← init product to 1

← work on data sets of multiple 4.

↑ load in array

↖ do our products together.

↑ tail case for $n\%4 \neq 0$.

← store our product vector to an array.

↑ multiply the results back together

Remember $n/4$ is floor division so $n/4 \cdot 4$ will chop off the remainder bits (ie $9 \rightarrow 8$)

This impl can have faster overflow issues so be careful!