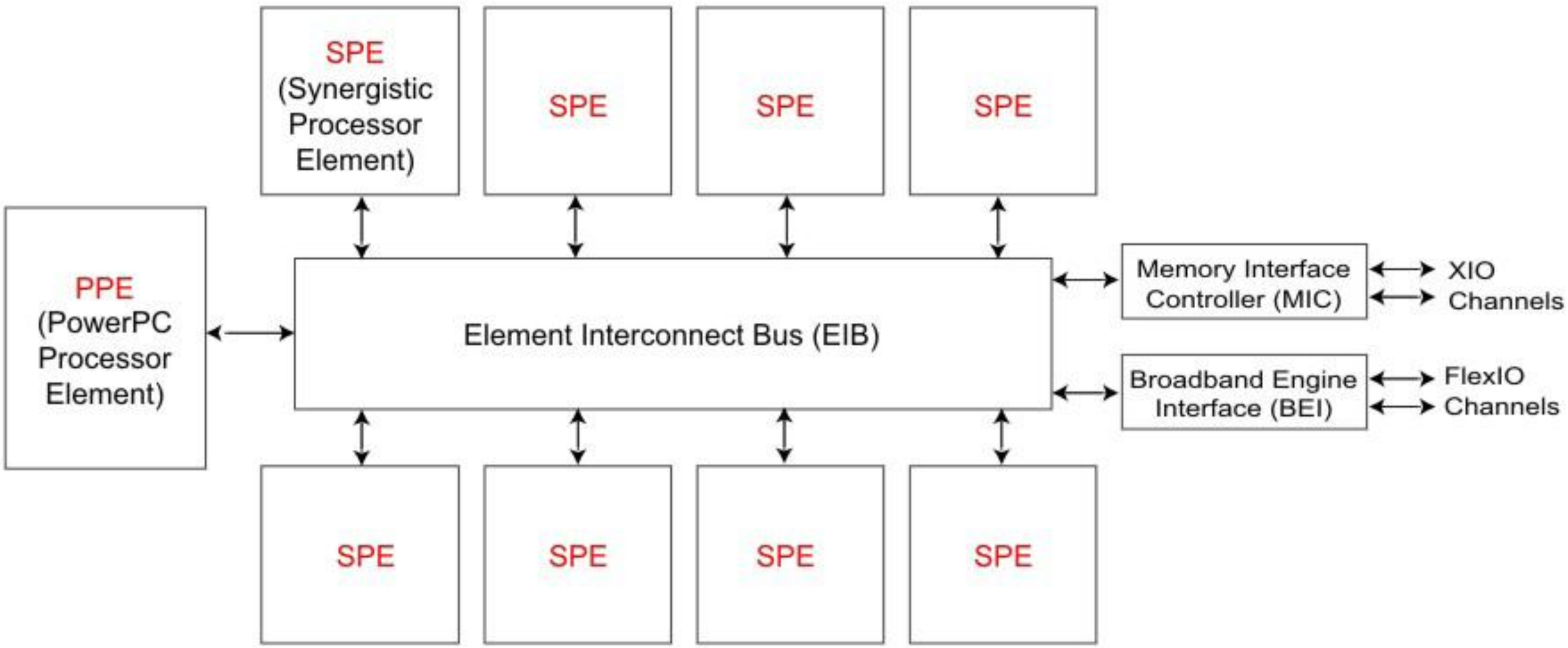
PS3 programming basics

Week 1. SIMD programming on PPE

Materials are adapted from the textbook

Overview of the Cell Architecture



XIO: Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels

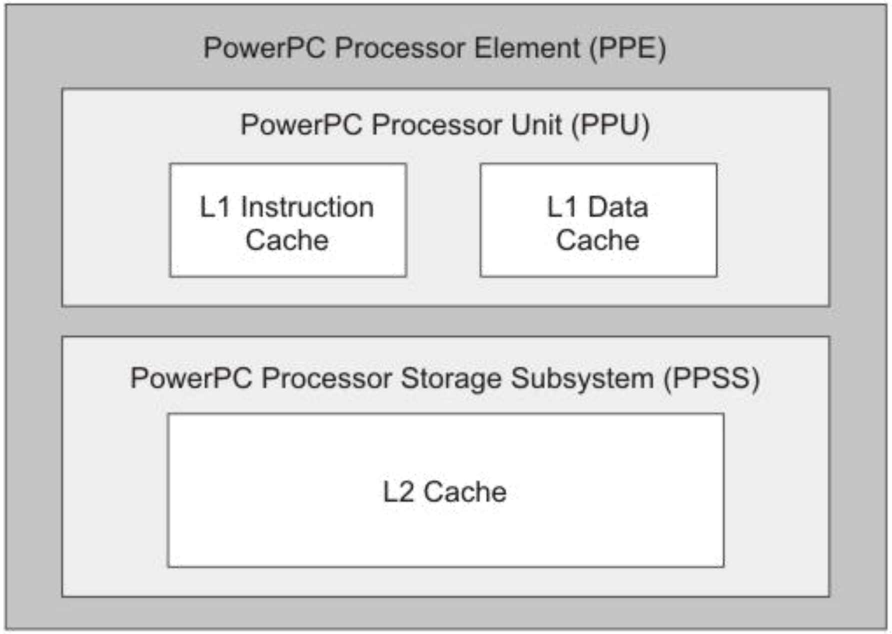
The PowerPC Processor Element

PPU

- Set of 64-bit registers
- 32 128-bit registers
- A 32-KB L1 I-cache
- A 32-KB L1 D-cache
- Two simultaneous threads of execution and can be viewed as a 2-way multiprocessor with shared dataflow.

PPSS

- A unified 512-KB L2 I+D cache
- Various queues
- A bus interface unit



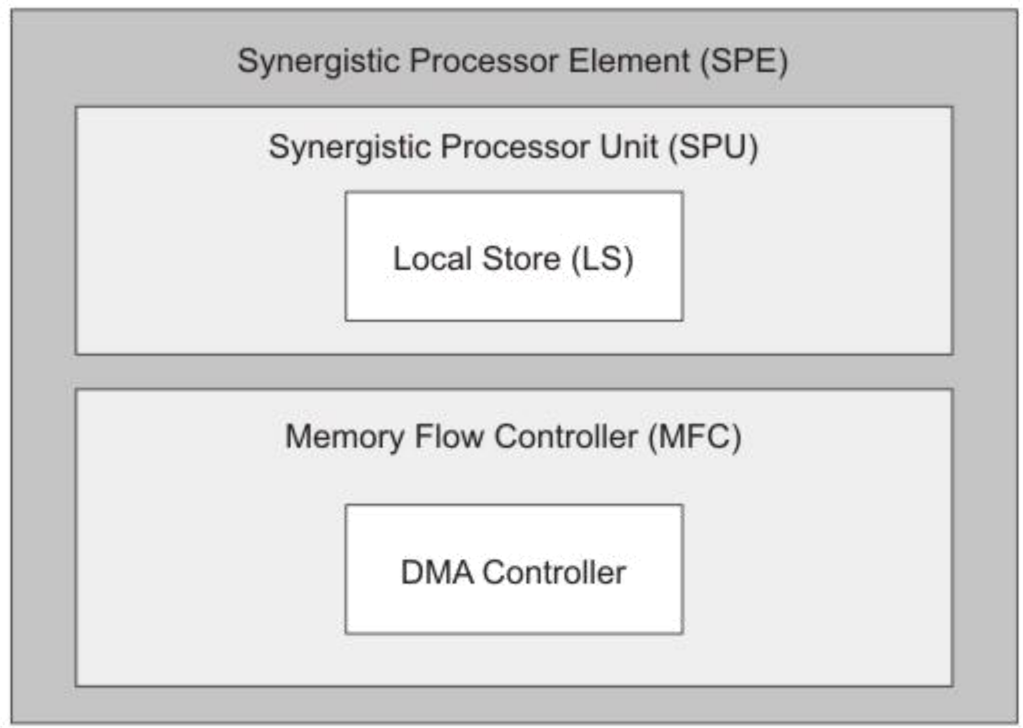
Synergistic Processor Elements

SPU

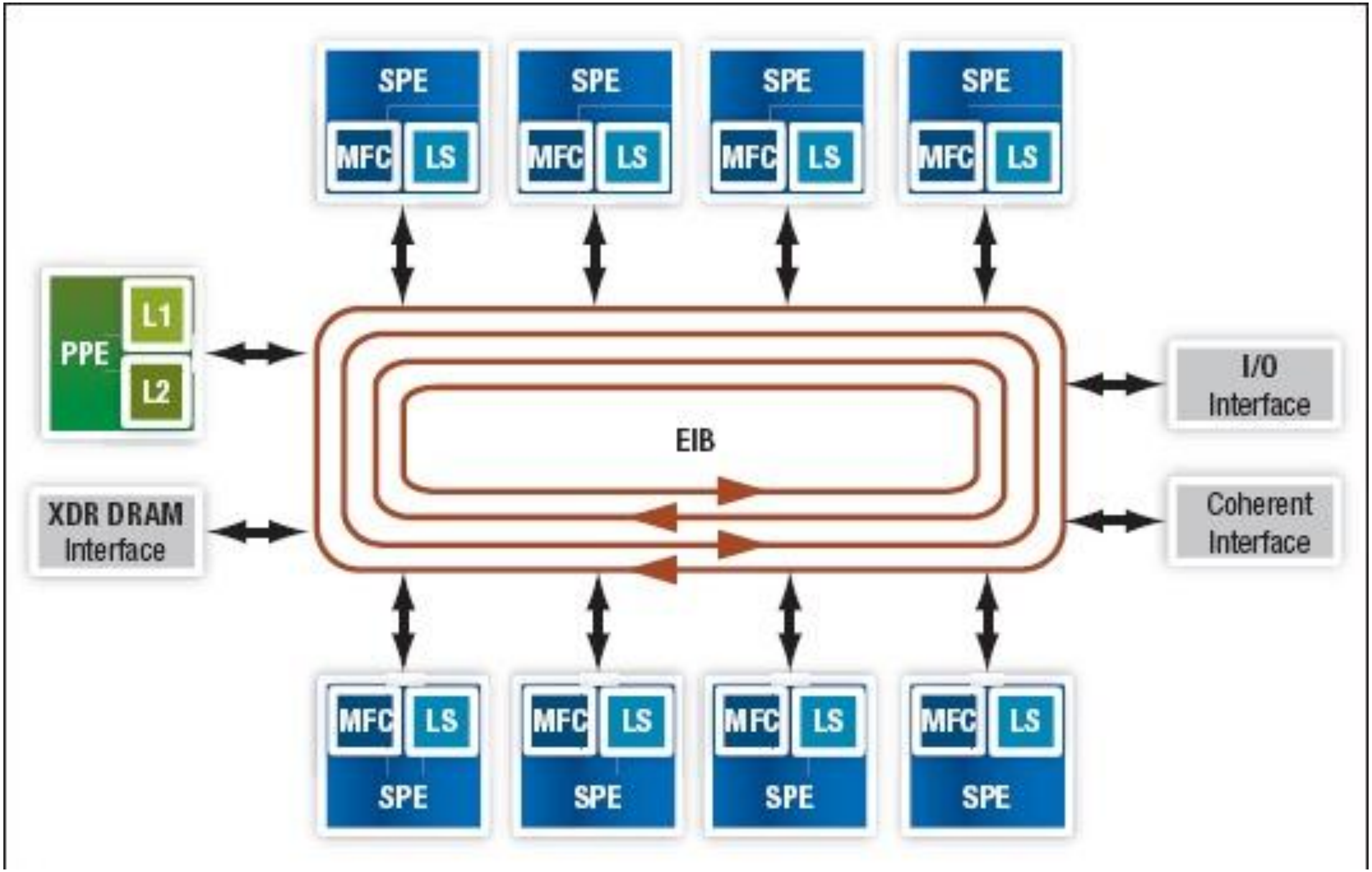
- 128 registers (each one 128 bits wide),
- 256-KB local store
- has its own program counter and is optimized to run SPE threads spawned by the PPE

MFC

- DMA transfers to move instructions and data between the SPU's LS and main storage.



Element Interconnect Bus

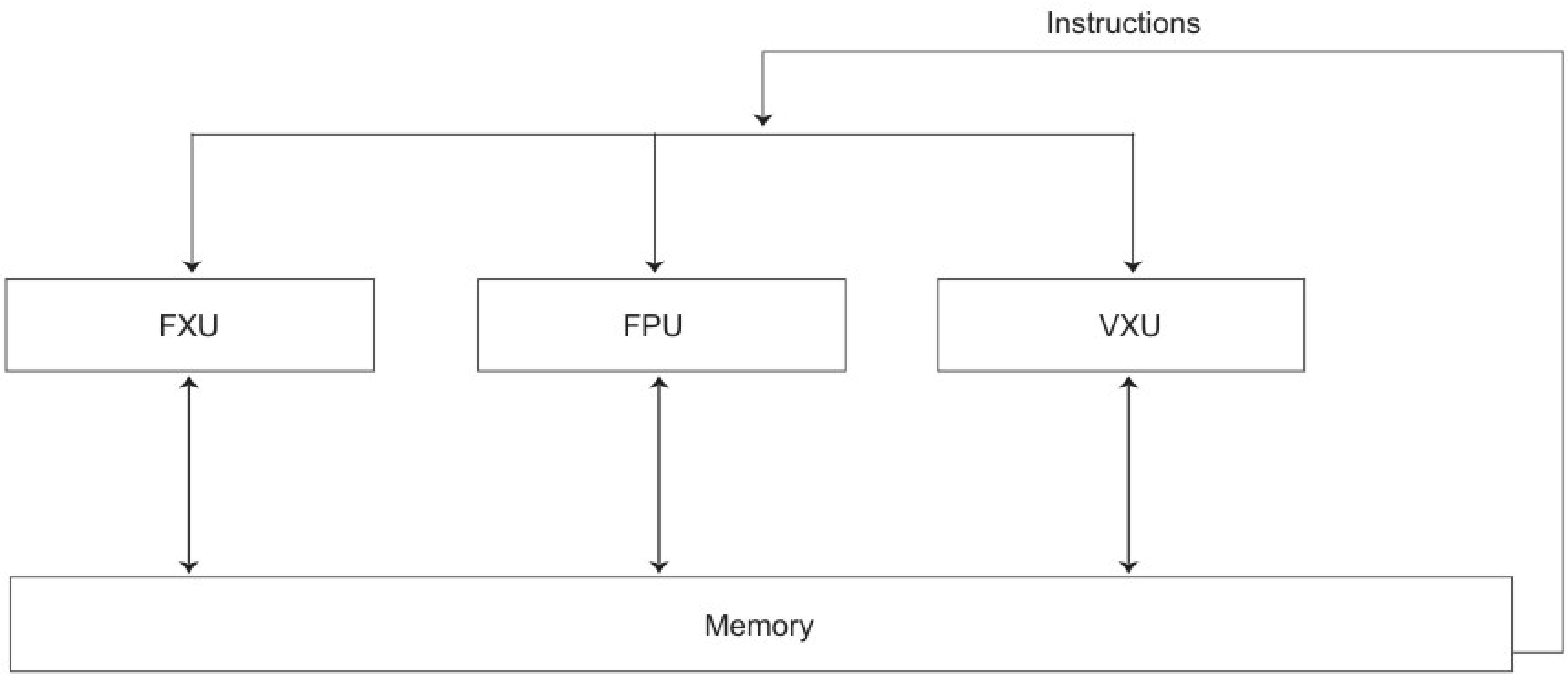


Threads and tasks

Term	Definition
PPE thread	A Linux thread running on a PPE.
SPE thread	A Linux thread running on an SPE. *Each such thread has its own SPE context which includes the 128 x 128-bit register file, program counter, and MFC Command Queues, and can communicate with other execution units (or with effective-address memory through the MFC channel interface).*
Cell Broadband Engine task	A task running on the PPE and SPE. *Each such task has one or more Linux threads. All the threads within the task share task's resources.*

Vector/SIMD Extension unit

- The 128-bit Vector/SIMD Multimedia Extension unit (VXU) operates concurrently with the PPU's fixed-point integer unit (FXU) and floating-point execution unit (FPU).



PPU SIMD PROGRAMMING BASICS

Vector instrinsic functions

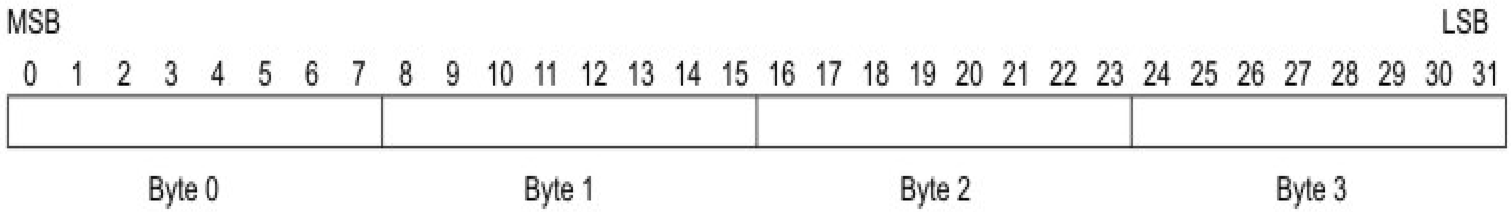
- **Specific**: have a 1-1 mapping with a single assembly-language instruction
  - EX: vec_abs(a)
- **Generic**: map to one or more assembly-language instructions
  - EX: vec_or(a,b),
- **Predicates**: compare values and return an integer that may be used directly for branching
  - EX: vec_all_eq(a,b), vec_any_eq(a,b)

Vector data types

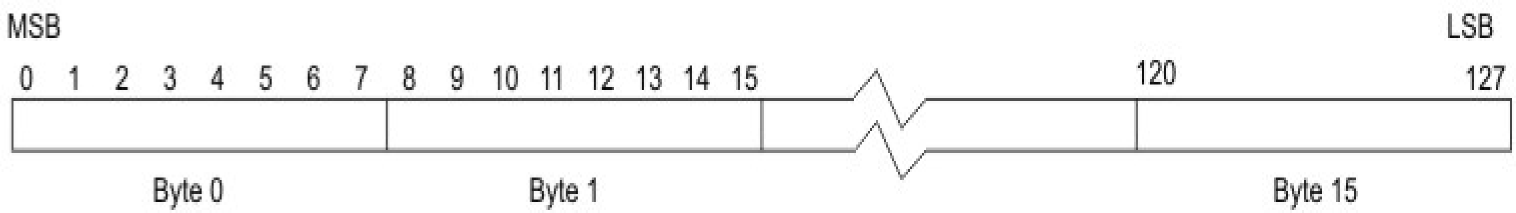
The vector registers are 128 bits and can contain

- Sixteen 8-bit values, signed or unsigned
  - EX: vector unsigned char
- Eight 16-bit values, signed or unsigned
  - EX: vector unsigned short
- Four 32-bit values, signed or unsigned
  - EX: vector unsigned int
- Four single-precision IEEE-754 floating-point
  - EX: vector float

Big-endian byte and bit ordering



Bit and Byte Order for a 32-bit Word



Bit and Byte Order for a 128-bit Register

A general approach to get data

- “typedefs” a union of an array of four ints and a vector of signed ints.

```
#include <stdio.h>
// Define a type that can be an array of ints or a vector.
typedef union {
    int iVals[4];
    vector signed int myVec;
} vecVar;
```

How to use it?

```
int main() {
    vecVar v1, v2, vConst; // define variables

    // load the literal value 2 into the 4 positions in vConst,
    vConst.myVec = (vector signed int){2, 2, 2, 2};
    // load 4 values into the 4 element of vector v1
    v1.myVec = (vector signed int){10, 20, 30, 40};
    // call vector add function
    v2.myVec = vec_add( v1.myVec, vConst.myVec );
    // see what we got!
    printf("\nResults:\nv2[0] = %d, v2[1] = %d, v2[2] = %d, v2[3]
    = %d\n\n", v2.iVals[0], v2.iVals[1], v2.iVals[2], v2.iVals[3]);
    return 0;
}
```

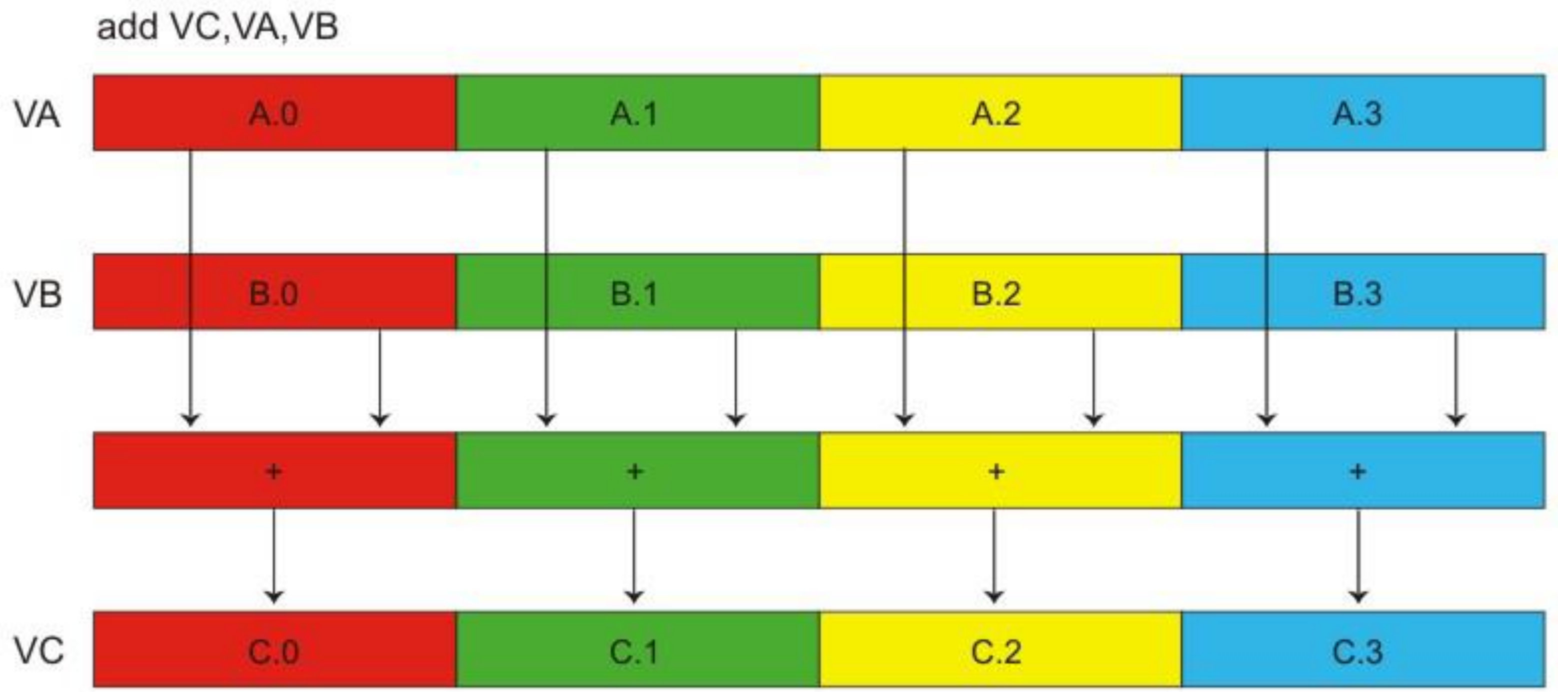
__attribute__(alligned(…))

- Variables are aligned at a boundary corresponding to its datatype size
  - The datatype size of vector is 16 (bytes)
- When declaring a variable, you can assign its alignment by __attribute__(aligned(…))
  - EX: int var ___attribute__(aligned(8))
  - A valid address will be like 0x0FFFFFF8 or 0x0FFFFFF0

Vector Add Operations

vector signed int VA,VB,VC;

VC = vec_add(VA,VB);



Example 1: array-summing

- Traditional approach

```
// 16 iterations of a loop
int rolled_sum(unsigned char bytes[16]) {
    int i; int sum = 0;
    for (i = 0; i < 16; ++i) { sum += bytes[i]; }
    return sum;
}
```

Vector Version (no loop)

```
// Vectorized for Vector/SIMD Multimedia Extension
int vectorized_sum(unsigned char data[16]) {
    vector unsigned char temp;
    union { int i[4]; vector signed int v; } sum;
    vector unsigned int zero = (vector unsigned int){0};
    // Perform a misaligned vector load of the 16 bytes.
    temp = vec_perm(vec_ld(0, data), vec_ld(16, data),
                        vec_lvsl(0, data));
    // Sum the 16 bytes of the vector
    sum.v = vec_sums((vector signed int)vec_sum4s(temp, zero),
    (vector signed int)zero);
    // Extract the sum and return the result.
    return (sum.i[3]);
}
```

Function Description

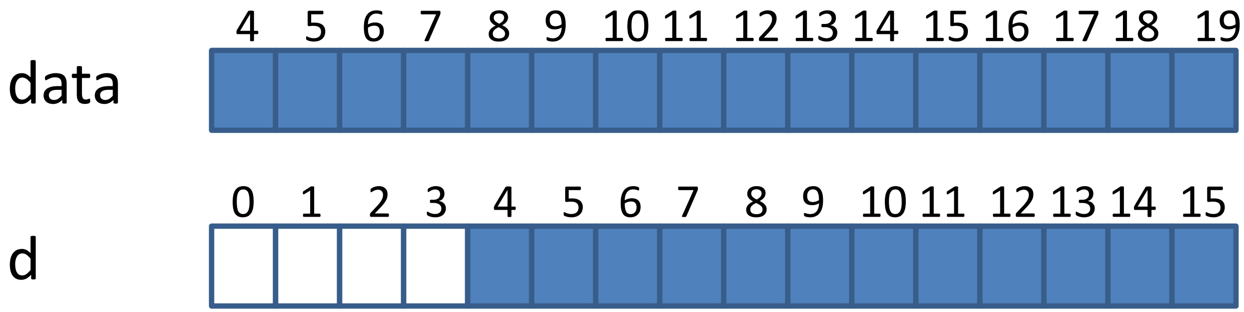
Functions	Explanation
d = vec_perm(a,b,c)	Vector Permute
d = vec_ld(a,b)	Vector Load Indexed
d = vec_lvsl(a,b)	Vector Load for Shift Left
d = vec_sums(a,b)	Vector Sum Saturated
d = vec_sum4s(a,b)	Vector Sum Across Partial (1/4) Saturated

d = vec_ld(a,b)

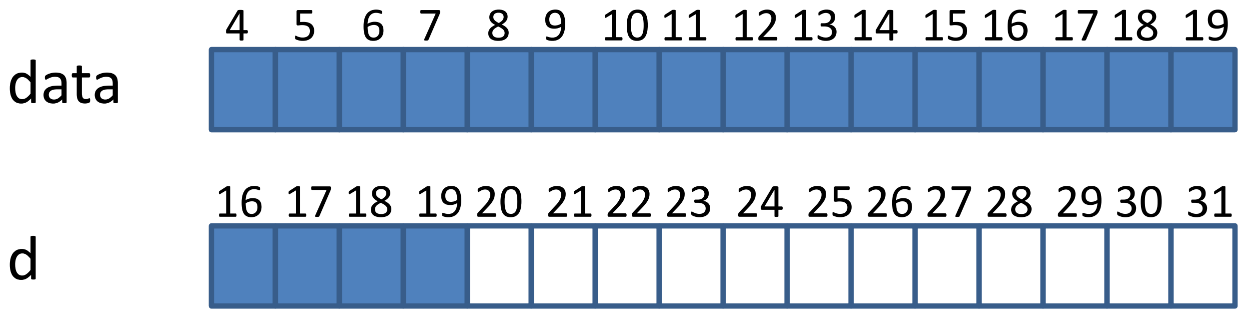
- Load 16 bytes from memory and return to **d**
- **a** (an integer) is added to the address of **b** (a pointer), and the sum is truncated to a multiple of 16 bytes. The result is the contents of the 16 bytes of memory starting at this address.
  - If the address is not aligned on a 16 bytes boundary, d is loaded from the next-lowest 16 byte boundary

Example

- d = vec_ld(0, data);



- d = vec_ld(16, data);



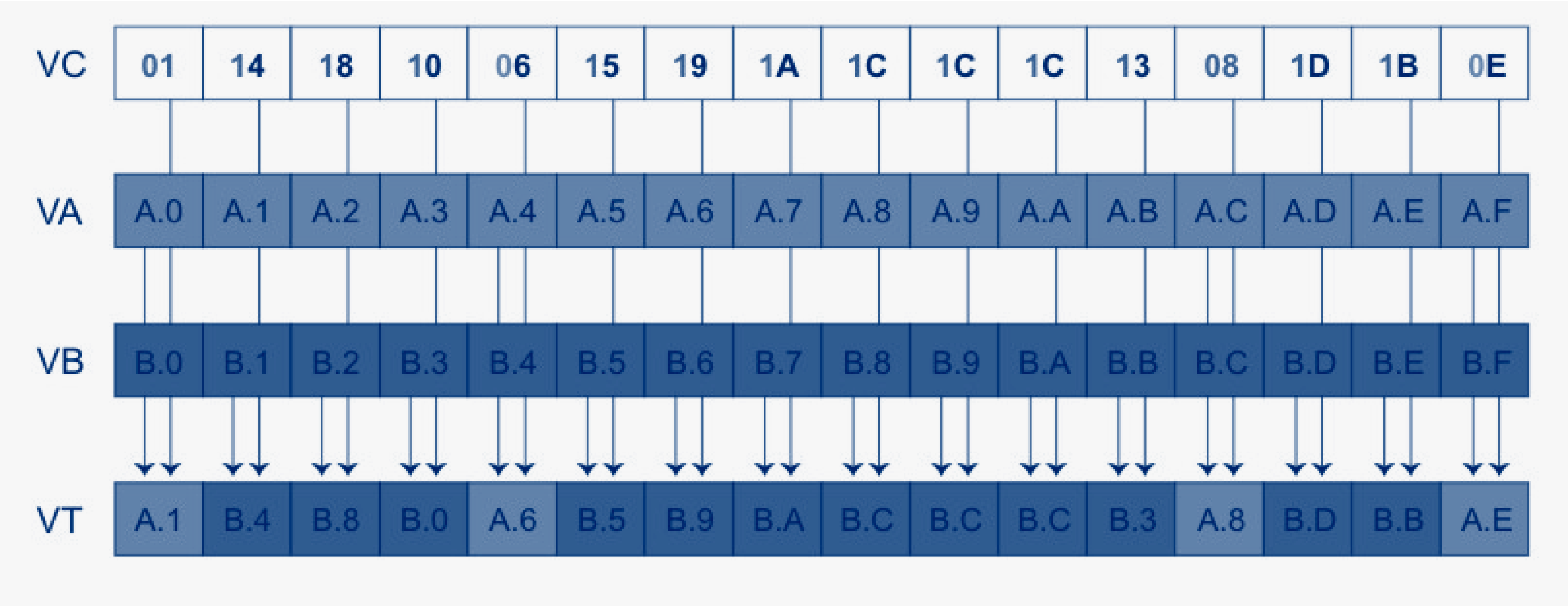
d = vec_lvsl(a,b)

- Does not perform any loading at all!!!
- Can be use to determine whether the pointer is aligned relative to the 16 byte vector boundary.
  - d = vec_lvsl(4,data)

	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
data																
d	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

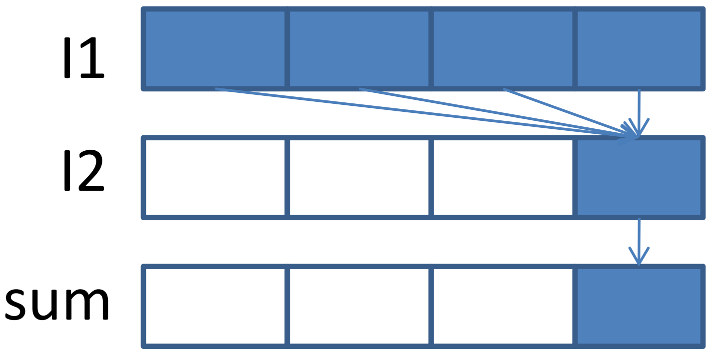
d = vec_perm(a,b,c)

- Think [a,b] is a 32 byte long vector. The indices of the bytes in b is from 16 to 31.
- c is an index array.

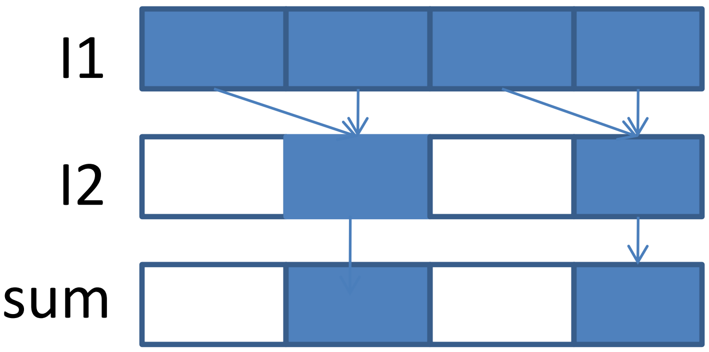


vec_sums, vec_sum2s, vec_sum4s

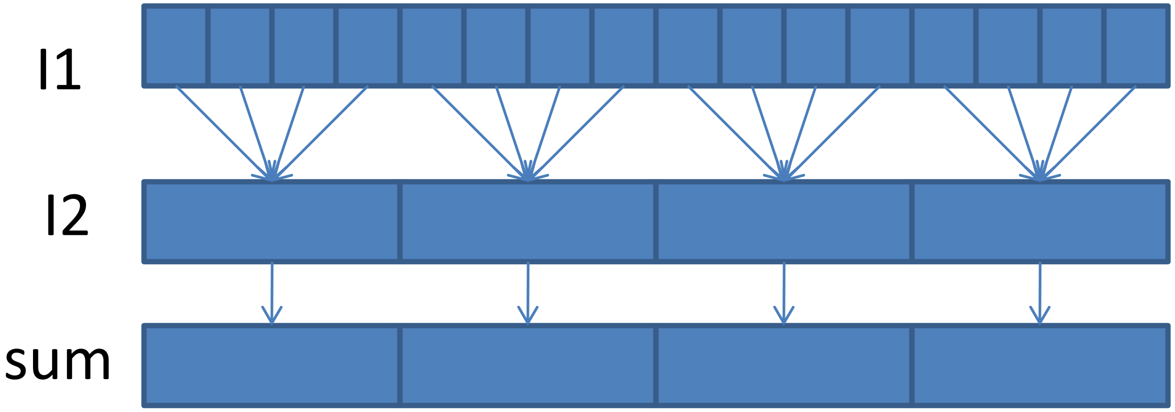
- sum = vec_sums(I1,I2)



- sum = vec_sum2s(I1,I2)



- sum = vec_sum4s(I1,I2)



Example 2: strcmp

- int strcmp(const char* str1, const char* str2);
  - Returns + if str1>str2, 0 if str1==str2, and - if str1<str2

```
int strcmp ( const char * str1, const char * str2 ){
    int size1 = strlen(str1); int size2 = strlen(str2);
    int N = min(size1,size2);
    for (int i =0; i<N; i++){
        if (str1[i]>str2[i]) return 1;
        else if (str1[i]<str2[i]) return -1;
    }
    if (size1==size2) return 0;
    if(size1>size2) return 1;   return -1;
}
```

Vector Version

- Let’s assume that both str1 and str2 are aligned at 16 boundaries.
- Basic idea:
  - (1) Check the equality of two vectors
  - (2) If not, then check element by element.
- Use vec_all_eq for (1)
  - vec_all_eq(a,b) returns 1 if all the element of a and b are equal. Otherwise, it returns 0

Example3: Insertion Sort

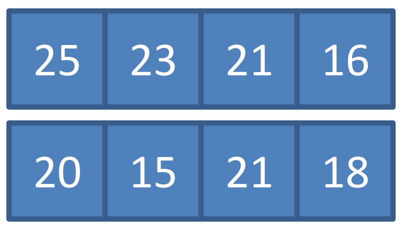
- EX: sort an array num[] in ascending order
  – Insert num(i) to the sorted list num(1:i-1)

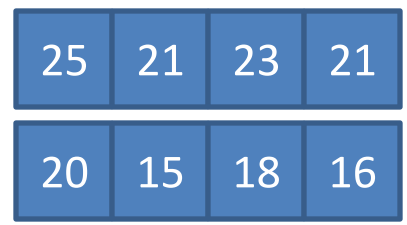
```
for(i=1; i<N; i++)
    for(j=i;j>0;j--)
        if(num(j-1)>num(j))
            swap(num(j-1),num(j));
        else break;
```

Vector Version

- Replace scalar variable `num(i)` by a vector
- How to perform the swap function?
  tmp=num(j-1);num(j)=num(j-1);num(j)=tmp;
  - Use vec_ld and vec_st
  - EX: vec_ld(vec,j*16, num); vec_st(vec,j*16, num)
  - What if num is not aligned on a 16 byte boundary?
- How about the comparison?
  - Can vec_all_gt work?

Two stages

1. Order the vectors, such that all larger elements in one vector and all smaller elements in another. (Inter-vector sorting)
   - EX: turn 

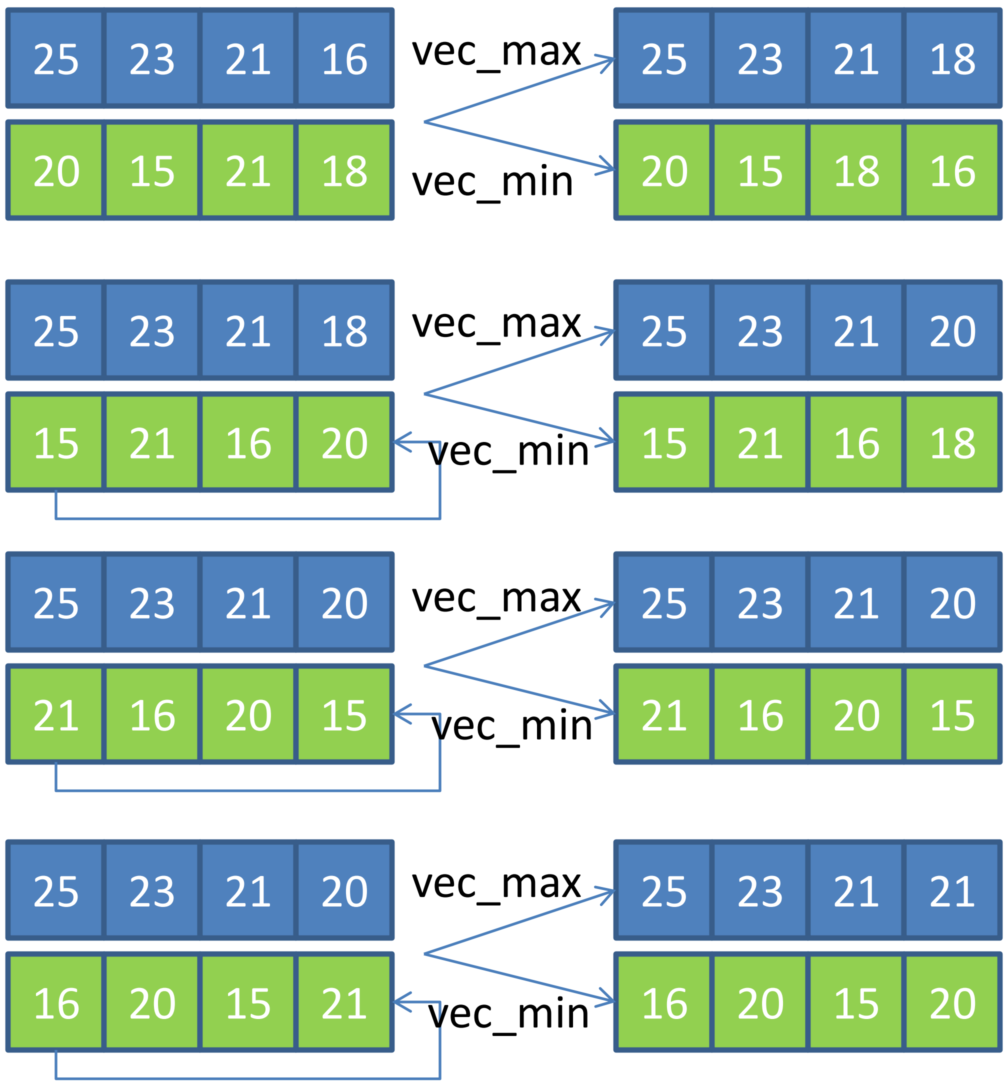
into 

   - What is the sequential code to do that?
2. Order the elements inside the individual vectors. (Intra-vector sorting)

Inter-vector Sort

- Two functions: vec_min and vec_max
  - Returns a vector containing min(or max) elements in each position
  - EX: vec_max({25,23,21,16}{20,15,21,18})
    ={25,23,21,18}
  - EX: vec_min({25,23,21,16}{20,15,21,18})
    ={20,15,21,16}
- Almost is what we need, except…

Rotate a Vector



- We can use vec_perm to rotate a vector
- The index vector is {4,5,6,7,8,9,10,11,12,13,14,15,0,1,2,3}

Intra-vector Sort

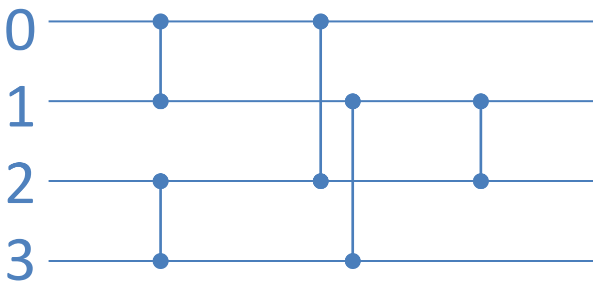
- Rely on four functions
  - d = vec_cmpgt(a,b): compares elements of **a** and **b**, if **a**[i]>=**b**[i], **d**[i]=F^8. Otherwise, **d**[i]=0,for i=0,1,2,3.
  - d = vec_and(a,b): d[i] = a[i]&b[i]
    - bit level AND
  - d = vec_and(a,b): d[i] = a[i]+b[i]
  - d = vec_perm(a,b,c): we had learned it.
- How to do that?
  - For example, sort {12,7,-5,9}

Some Analysis

- How many comparisons do we need?
  - (0,1),(0,2),(0,3),(1,2),(1,3),(2,3)
- Which can be compared (sorted) in parallel?
  - For example: {(0,1), (2,3)}, {(0,2), (1,3)},{(0,3), (1,2)}
- What can we get if {(0,1), (2,3)} is sorted first?
  - We get $A[0] \leq A[1]$ and $A[2] \leq A[3]$. What's next?
- What can we get after {(0,2), (1,3)} is sorted?
  - $A[0] \leq A[1]$, $A[2] \leq A[3]$ (why?) $A[0] \leq A[2]$, $A[1] \leq A[3]$.
- What do we miss?

Sorting Network

- Step 1: {(0,1)(2,3)}
- Step 2: {(0,2)(1,3)}
- Step 3: {(1,2)}



- Exercise: what's the sorting network if we sort {(0,3), (1,2)} first? And {(0,2), (1,3)} first?
- How to make comparison of {…}?
  – Need to compare elements using vec_cmpgt
  – Need to exchange data according to the result

EX: Compare {(0,1),(2,3)}

b=vec_perm(a,a,{4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11});

//b[0]=a[1], b[1]=a[0], b[2]=a[3], b[3]=a[2]

d = vec_cmpgt(a,b)

// For a={12,7,-5,9}, b={7,12,9,-5} ➔ d={F^8,0,0,F^8}

- Exercise: what is the index array if we want to compare {(0,2),(1,3)} or {(1,2)}?
  - For {(0,2),(1,3)}, {8,9,10,11,0,1,2,3,12,13,14,15,4,5,6,7}
  - For {(1,2)}, {0,1,2,3,8,9,10,11,4,5,6,7,12,13,14,15}

Vector Comparison Functions

- We need to exchange data if d[0]=F^8 or d[2]=F^8
- The only way to exchange data is by vec_perm.
  - How to design the index array c?
    - If d=={0,F^8,0,F^8}, c={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
    - If d=={F^8,0,0,F^8}, c={4,5,6,7,0,1,2,3,8,9,10,11,12,13,14,15}
    - If d=={0,F^8,F^8,0}, c={0,1,2,3,4,5,6,7,12,13,14,15,8,9,10,11}
    - If d=={F^8,0,F^8,0}, c={4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11}
  - One possible way to generate c = base+mask
    - base can be {0,1,2,3,0,1,2,3,8,9,10,11,8,9,10,11}
    - mask can be and(d,{4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4})

Exercises

- How to design the index array for {(0,2)(1,3)}?
  - base={0,1,2,3,5,6,7,8, 0,1,2,3,5,6,7,8}
  - mask={8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8}
- How to design the index array for {(0,3)(1,2)}?
  - base={0,1,2,3,5,6,7,8, 5,6,7,8,0,1,2,3}
  - mask={12,12,12,12,4,4,4,4,4,4,4,4,12,12,12,12}
- How to design the index array for {(1,2)}?
  - base={0,1,2,3,5,6,7,8, 5,6,7,8,12,13,14,15}
  - mask={0,0,0,0, 4,4,4,4,4,4,4,4,0,0,0,0}

Homework

- Read textbook chap 9.
- Implement "quick sort" or "merge sort"
  - Implement the sequential code
  - Use vectorized statements.
  - Compare the performance for different implementations and to the insertion sort in the textbook