ELE 655
Microprocessor System Design

Section 3 – Data Level Parallelism

Class 1 – VLIW

DLP

Data Level Parallelism

- SIMD overview
  - Leverage large parallel data problems
    - Matrix oriented scientific computing
    - Image and audio processing
  - Can be more energy efficient
    - Fewer instructions
    - Simpler logic than Dynamic scheduling with speculation, …

DLP

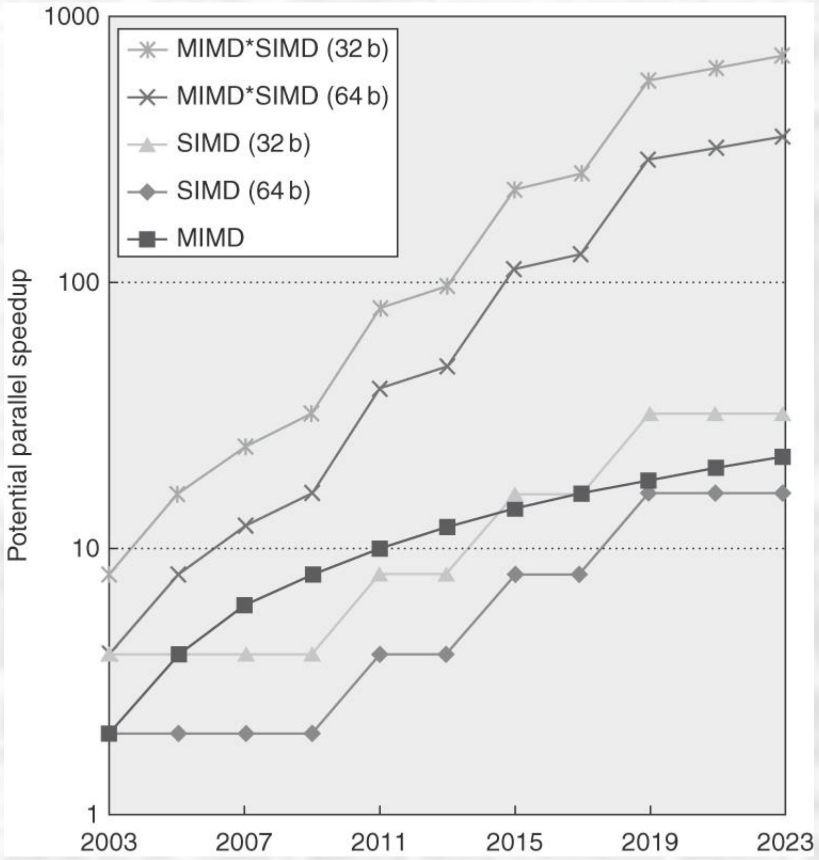
Data Level Parallelism

- SIMD overview
  - 3 variations
    - Vector processors
    - SIMD extensions to MIMD or SISD processors
    - Graphics Processor Units (GPUs)

DLP

Data Level Parallelism

- SIMD overview



MIMD expansion = 2 cores / 2 yrs

SIMD expansion = 2x ops/ 4 yrs

DLP

Data Level Parallelism

- Vector Architecture
  - Read sets of data into “vector” registers
  - Operate on registers
  - Store results back to memory

DLP

Data Level Parallelism

- Vector Architecture
  - Memory accesses
    - Memory naturally provides data serially
    - Vector registers read/write data serially then operate in parallel

DLP
Data Level Parallelism

- Vector Architecture

Pipelined
1 word/clk after latency

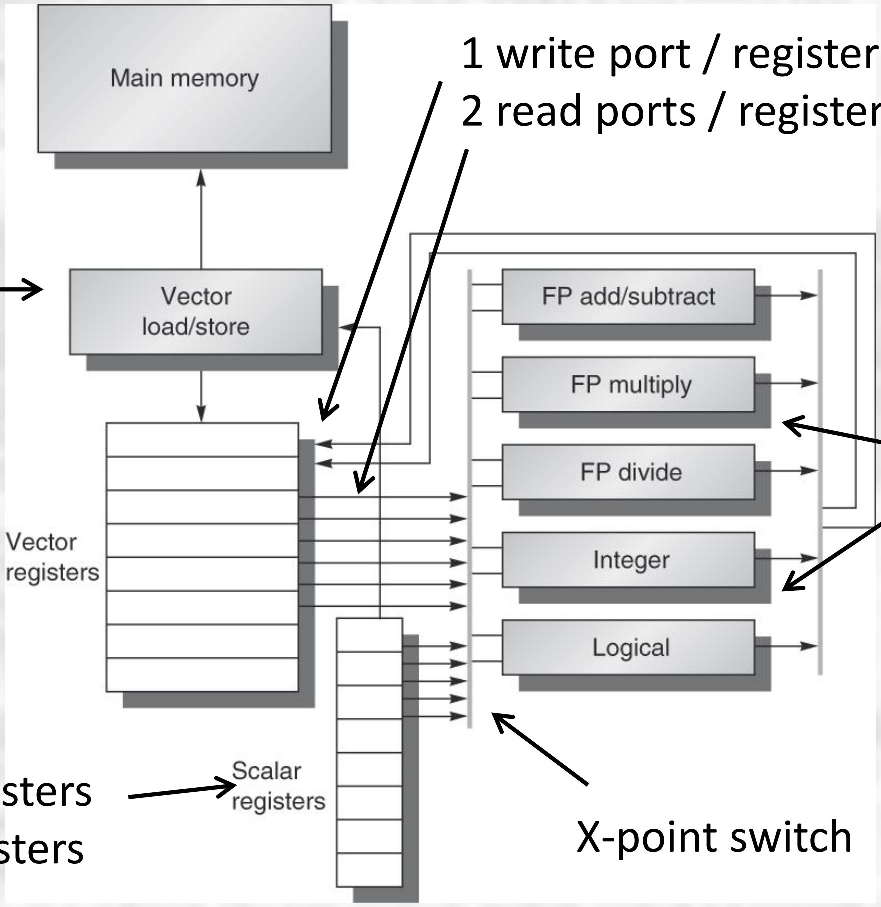
8 registers
64 elements / register
64 bits / element

32 gp registers
32 fp registers

1 write port / register
2 read ports / register

Fully pipelined

X-point switch



DLP

Data Level Parallelism

- Vector Architecture
  - Instruction Set
  - Fig 4.3

DLP
Data Level Parallelism

- Vector Architecture
  - Instruction Example

```
L.D          F0,a        ; load scalar a
LV           V1,Rx       ; load vector X
MULVS.D      V2,V1,F0    ; vector-scalar multiply
LV           V3,Ry       ; load vector Y
ADDVV        V4,V2,V3    ; add
SV           Ry,V4       ; store the result
```

  - 6 instructions vs approximately 600 in MIPS for 64 iterations
    - Even worse if we unroll loops

DLP
Data Level Parallelism

- Vector Architecture
  - Execution time
    - Length of operands
    - Structural hazards between operations
    - Data dependencies between operations
  - VMIPS
    - Execution units take 1 element / clock cycle
    - Pipelineing → execution time = fill + vector length ≅ vector length

DLP

Data Level Parallelism

- Vector Architecture

  - Convoy

    - Vector instructions that could potentially execute together
    - Limited by structural hazards
    - Limited by issue width

    - We add the restriction one convoy must finish before the next starts

  - Chaining
    - Within a convoy:
      - Vector operations start as soon as the first element (or any dependent element) of its source operand are available
      - Solves any RAW dependencies in a convoy

DLP
Data Level Parallelism

- Vector Architecture
  - Chime
    - Unit of time to execute one convoy
    - Simplified to ignore fill, chaining delays, and issue delays
    - Vector length of n, and m chimes
      - Requires approximately m x n clock cycles
  - Execution time = # of convoys X chime

DLP

Data Level Parallelism

- Vector Architecture

```
LV          V1,Rx       ;load vector X
MULVS.D     V2,V1,F0    ;vector-scalar multiply
LV          V3,Ry       ;load vector Y
ADDVV.D     V4,V2,V3    ;add two vectors
SV          Ry,V4       ;store the sum
```

Convoys:

1	LV	MULVS.D	chaining allows V1 RAW dependency
2	LV	ADDVV.D	chaining allows V2 RAW dependency
3	SV		

For 64 element vectors, requires 64 x 3 = 192 clock cycles

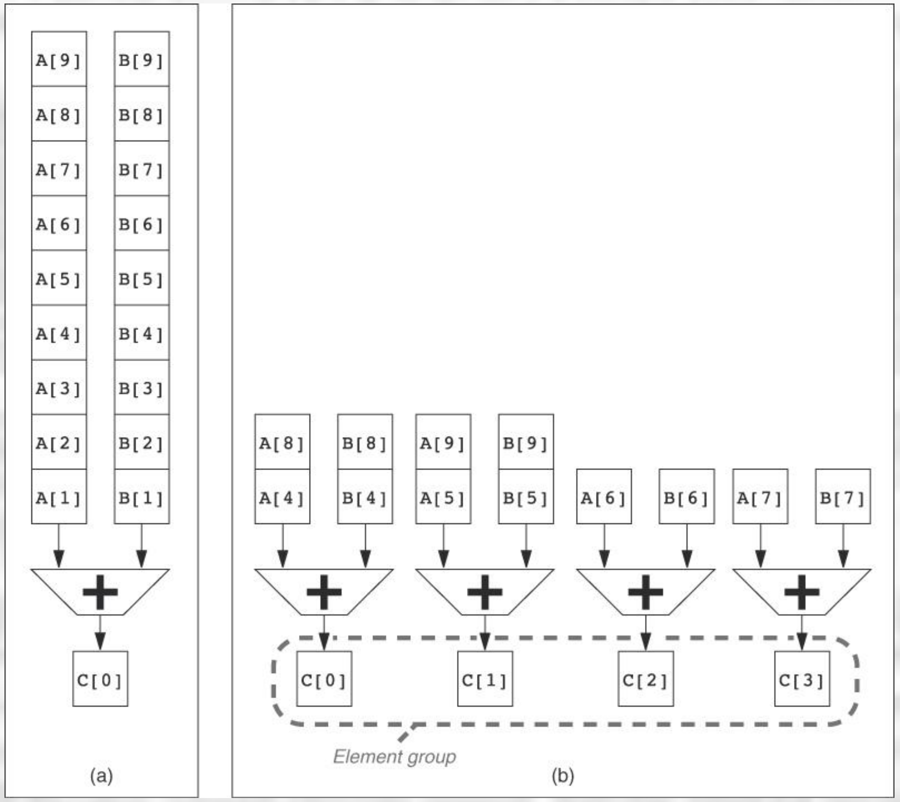
DLP
Data Level Parallelism

- Vector Architecture
  - Overhead not included in chime measurements
  - Issue width
    - Single issue requires an extra clock for a second instruction , …
  - Start up time
    - Pipeline fill
    - 6 clks FP add, 7 clks FP mult, 20 clks FP div, 12 for vector load
  - Chaining delay
    - Directly tied to start up time

DLP

Data Level Parallelism

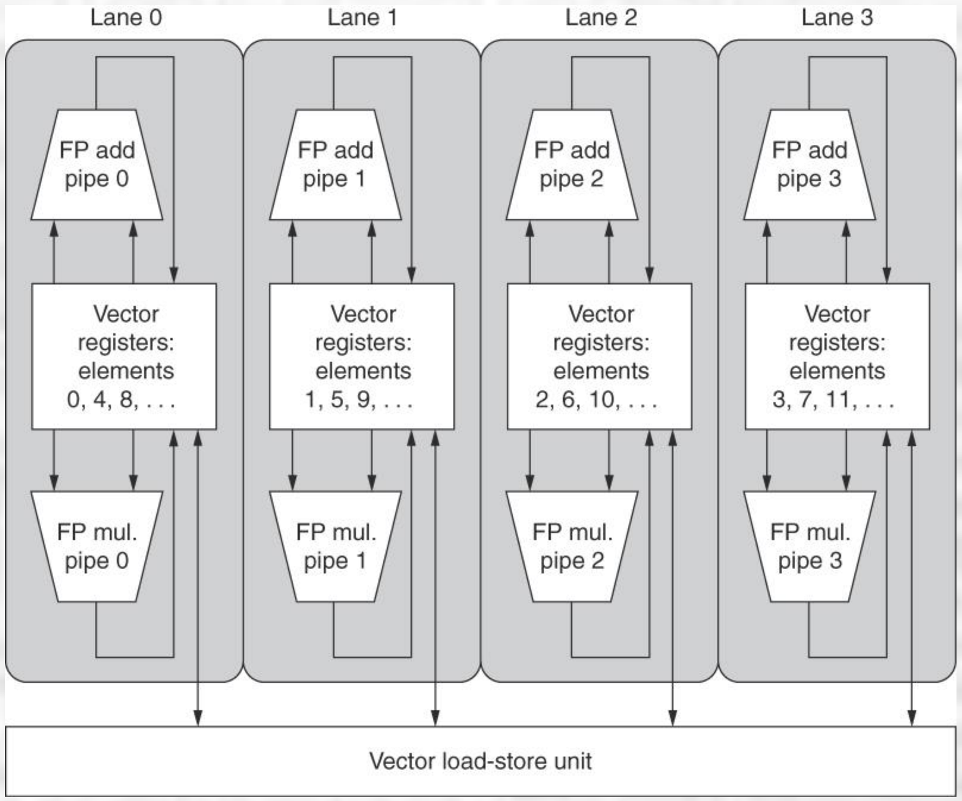
- Vector Architecture – Enhancements
  - Multiple Lanes



DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Multiple Lanes
  - No inter-lane communication
  - Optimize
    - Performance vs. cost
    - Clock speed vs. complexity


Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

DLP

Data Level Parallelism

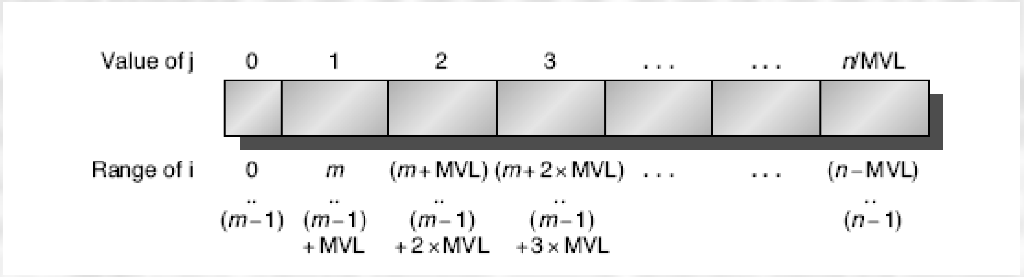
- Vector Architecture – Enhancements
  - Vector Length Register
  - Real code does not have data in nice 64 word segments
  - Real code may not even know the vector length until run time
  - VLR indicates the vector length

DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Strip Mining
    - Use VLR to manage vector lengths > hardware configuration
      - Break actual length into an integer number (n) of natural length + what's left
      - Execute the "what's left" by setting VLR < natural
      - Loop n times with VLR = natural

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
          for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
                    Y[i] = a * X[i] + Y[i] ; /*main operation*/
          low = low + VL; /*start of next vector*/
          VL = MVL; /*reset the length to maximum vector length*/
}
```



DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Vector Mask Register
    - Determines which elements in a vector actually get saved
    - 1 bit for each word in the vector
    - Operate in the vector as normal
    - Only writeback results for words with the mask set

DLP
Data Level Parallelism

- Vector Architecture – Enhancements
  - Vector Mask Register
    - Use VMR to skip elements in the vector
    - Consider:

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

    - Can’t be vectorized because of the “if”

DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Vector Mask Register
  - Use vector mask register to “disable” elements and allow the “if”

```
LV          V1,Rx        ;load vector X into V1
LV          V2,Ry        ;load vector Y
L.D         F0,#0        ;load FP zero into F0
SNEVS.D     V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2     ;subtract under vector mask
SV          Rx,V1        ;store the result in X
```

  - GFLOPS rate decreases!

DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Banked Memory
  - We can execute a new calculation every clock cycle via pipelining
  - But can we provide data at full clock rate
    - Caching
    - Memory banking
      - Allow multiple loads and stores in parallel
      - Scatter-Gather → independent bank addressing
      - Multiple processors → independent instruction streams
      - → Large number of banks
        - Cray 1 = 1024 banks

DLP

Data Level Parallelism

- Vector Architecture – Enhancements

  - Striding

  - How do we work with a multidimensional array structure
    - In memory the array is linear (row major or column major)
    - Subsequent element on the minor axis are separated by the major axis length – stride
    - Add a register to hold the stride value (computed at run time)

    - Can lead to bank contention
      - Separate stride accesses end up in the same bank → stall

$$\frac{Number\ of\ banks}{Least\ common\ multiple\ (Stride, \#\ of\ banks)} < Bank\ busy\ time$$

DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Scatter-Gather
    - Sparse matrices are very common
    - Use a vector to indicate the non-zero element of the matrix

```
for (i=0; i<n; i++)
        A[K[i]] = A[K[i]] + C[M[i]];
```

Where A and C are the matrices and K and M are vectors of the non-zero elements

- Special instructions LVI, SVI to gather and scatter
  - Use one operand (non-zero element vector) as an index to a base address

DLP

Data Level Parallelism

- Vector Architecture – Enhancements
  - Scatter-Gather
    - Add elements of two sparse matrices

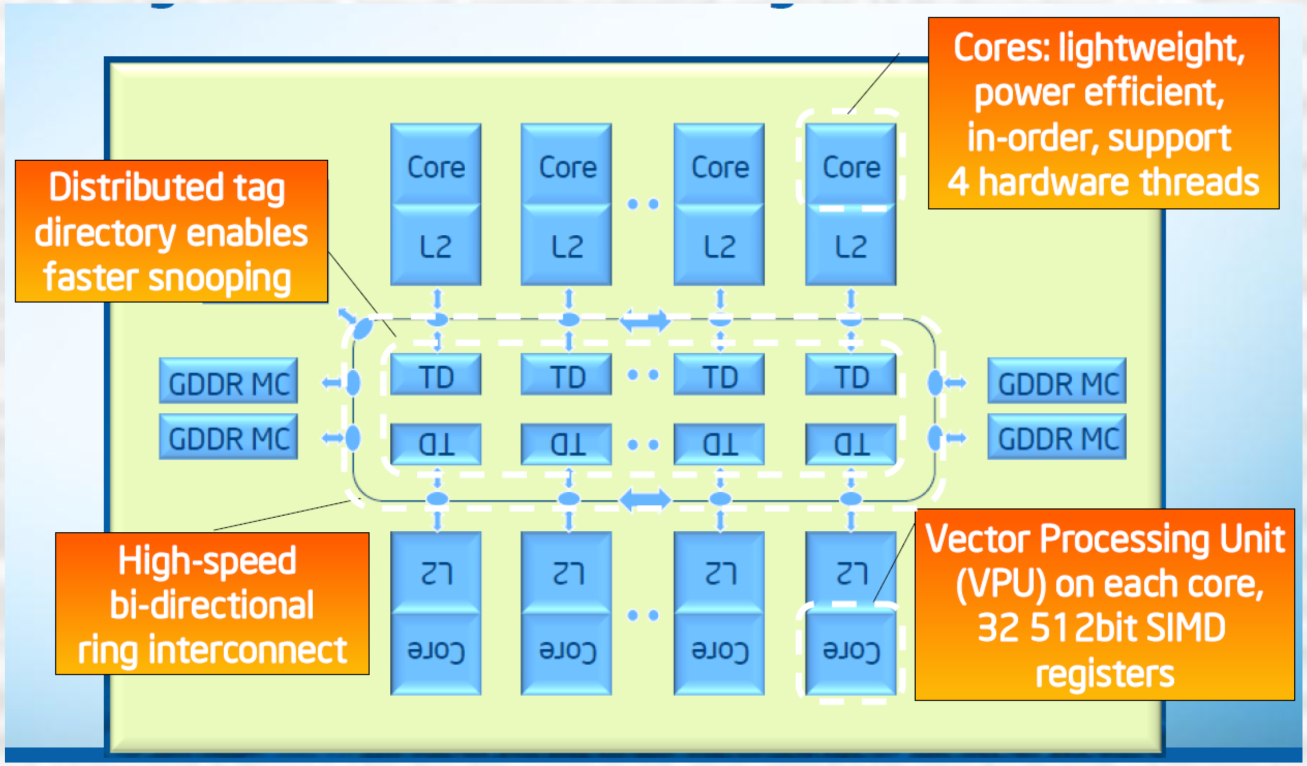
Ra, Rc, Rk, Rm are the base address for the vectors

```
LV          Vk, Rk          ; load Rk
LVI         Va, (Ra + Vk)   ; load A[K[ ]] – gather
LV          Vm, Rm          ; load Rm
LVI         Vc, (Rc + Vm)   ; Load C[M[ ]] – gather
ADDVV.D     Va, Va, Vc      ; add vectors
SVI         (Ra + Vk), Va   ; store A[K[ ]] - scatter
```

DLP

Data Level Parallelism

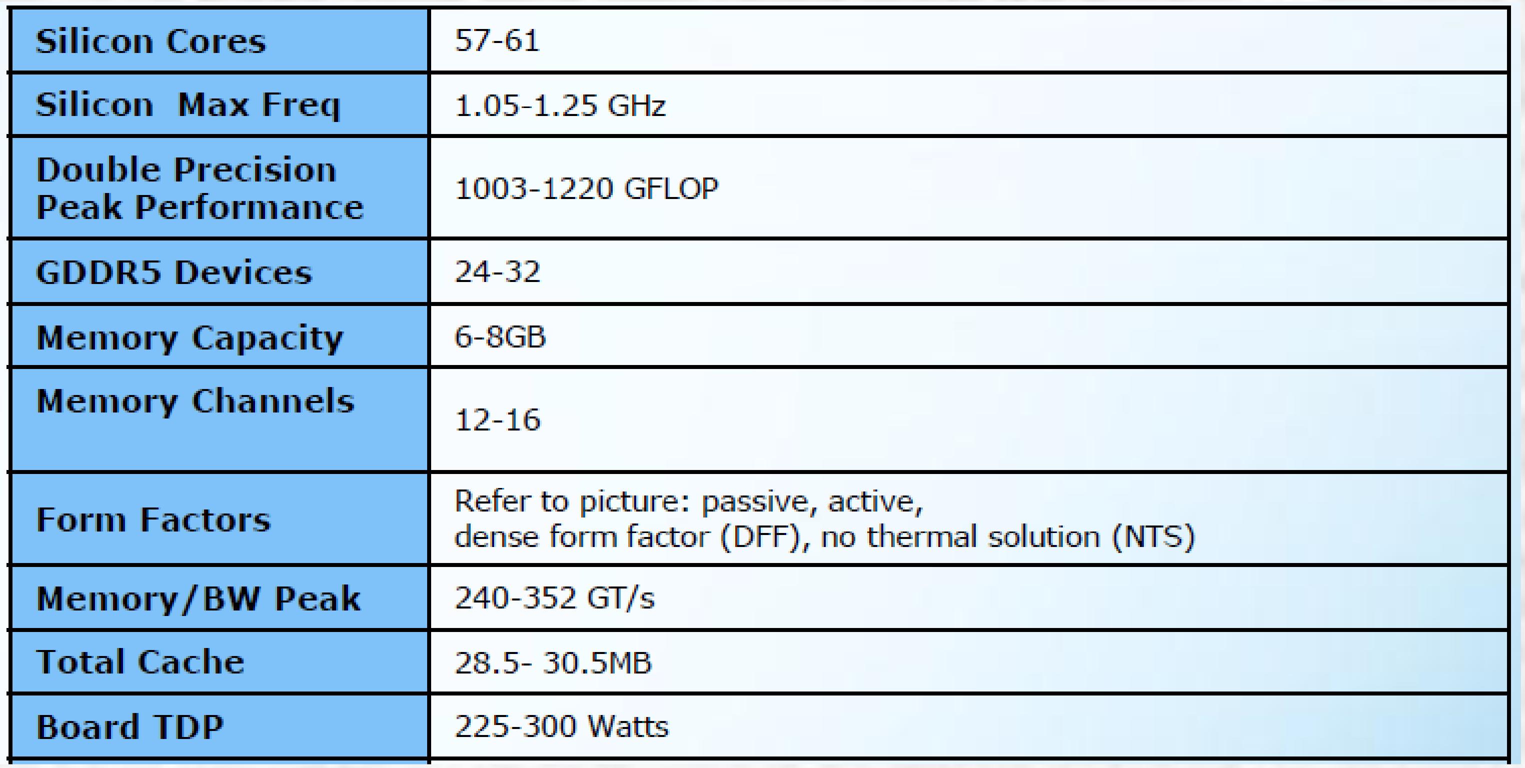
- Vector Processor – modern example
  - Intel Phi Co-processor


Distributed tag directory enables faster snooping
Core
L2
TD
GDDR MC
Cores: lightweight, power efficient, in-order, support 4 hardware threads
High-speed bi-directional ring interconnect
Vector Processing Unit (VPU) on each core, 32 512bit SIMD registers

DLP

Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor

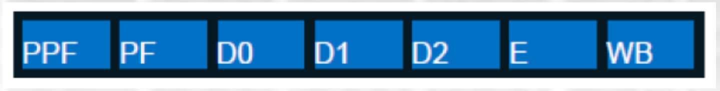


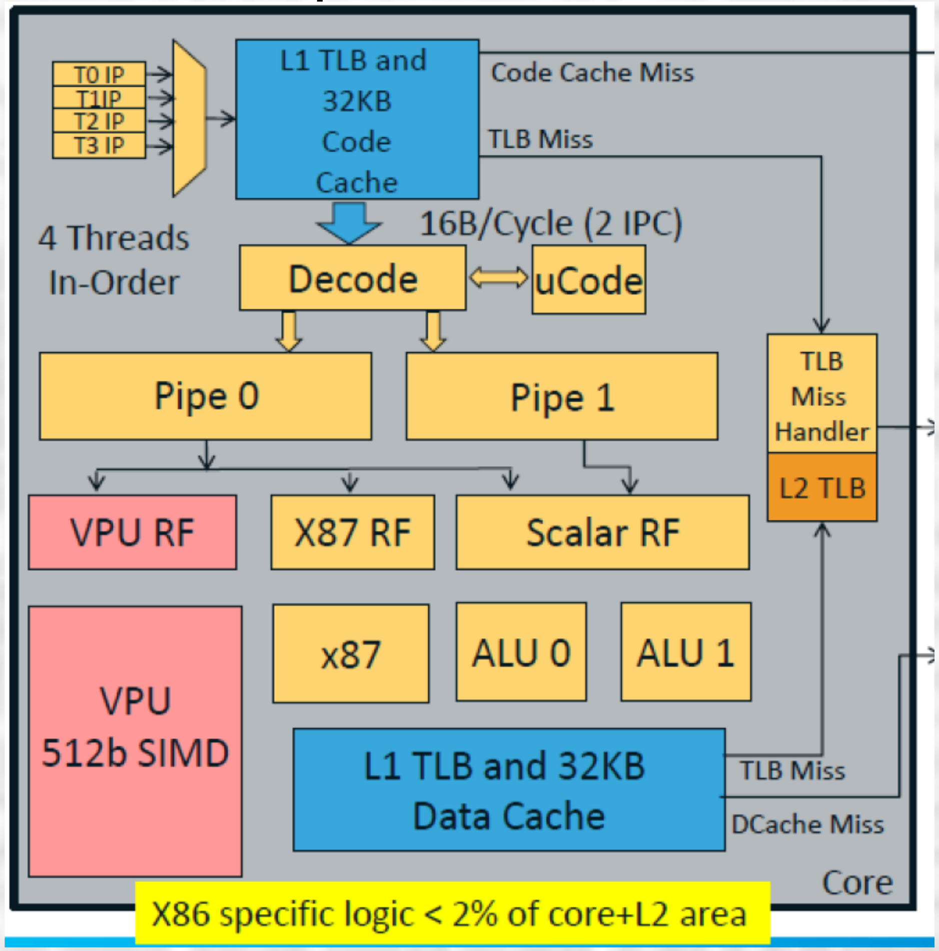
Silicon Cores	57-61
Silicon Max Freq	1.05-1.25 GHz
Double Precision Peak Performance	1003-1220 GFLOP
GDDR5 Devices	24-32
Memory Capacity	6-8GB
Memory Channels	12-16
Form Factors	Refer to picture: passive, active, dense form factor (DFF), no thermal solution (NTS)
Memory/BW Peak	240-352 GT/s
Total Cache	28.5- 30.5MB
Board TDP	225-300 Watts

DLP

Data Level Parallelism

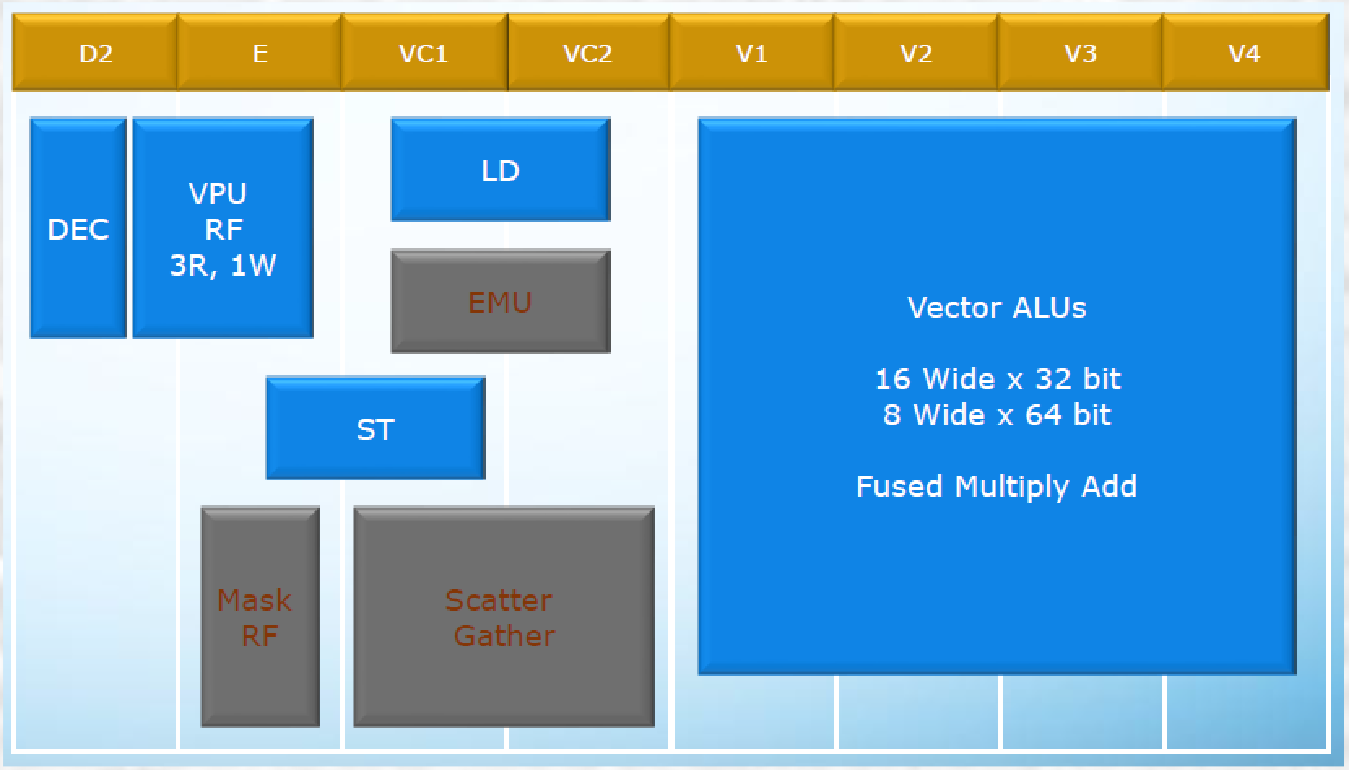
- Vector Processor – modern example
  - Intel Phi Co-processor





DLP
Data Level Parallelism

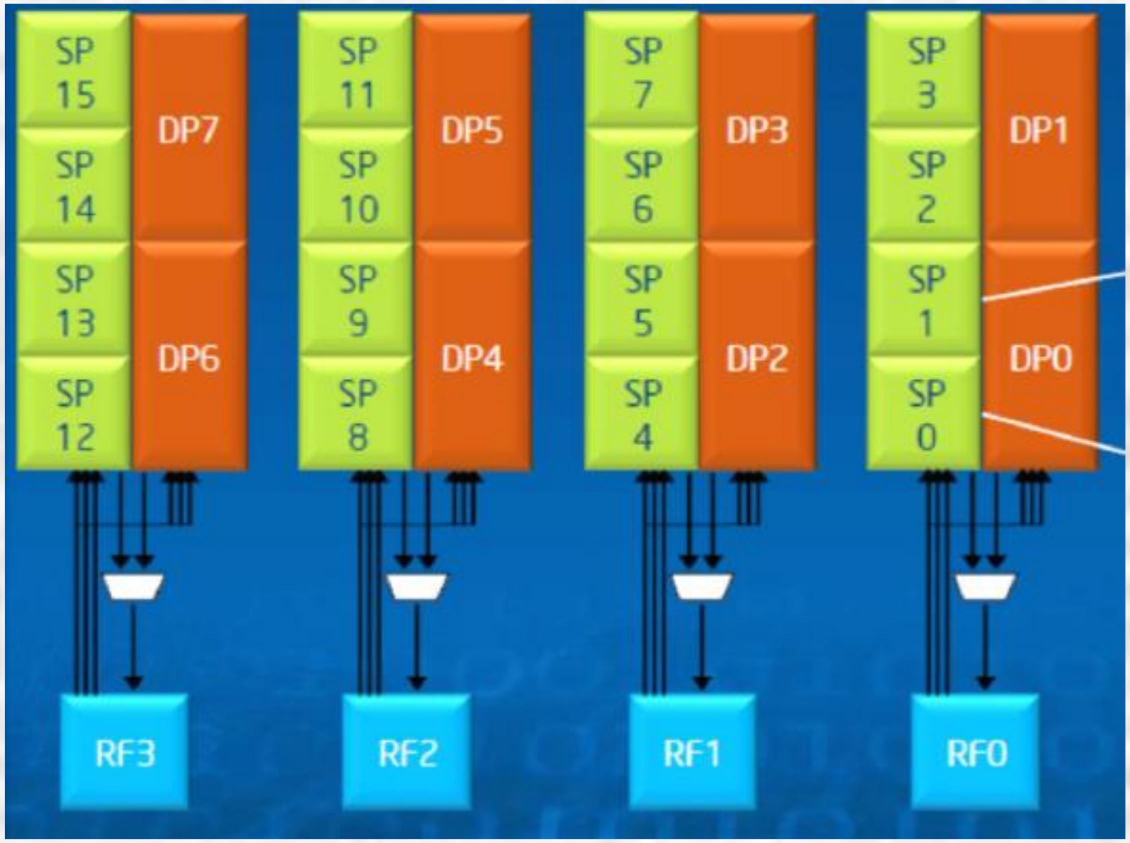
- Vector Processor – modern example
  - Intel Phi Co-processor



DLP

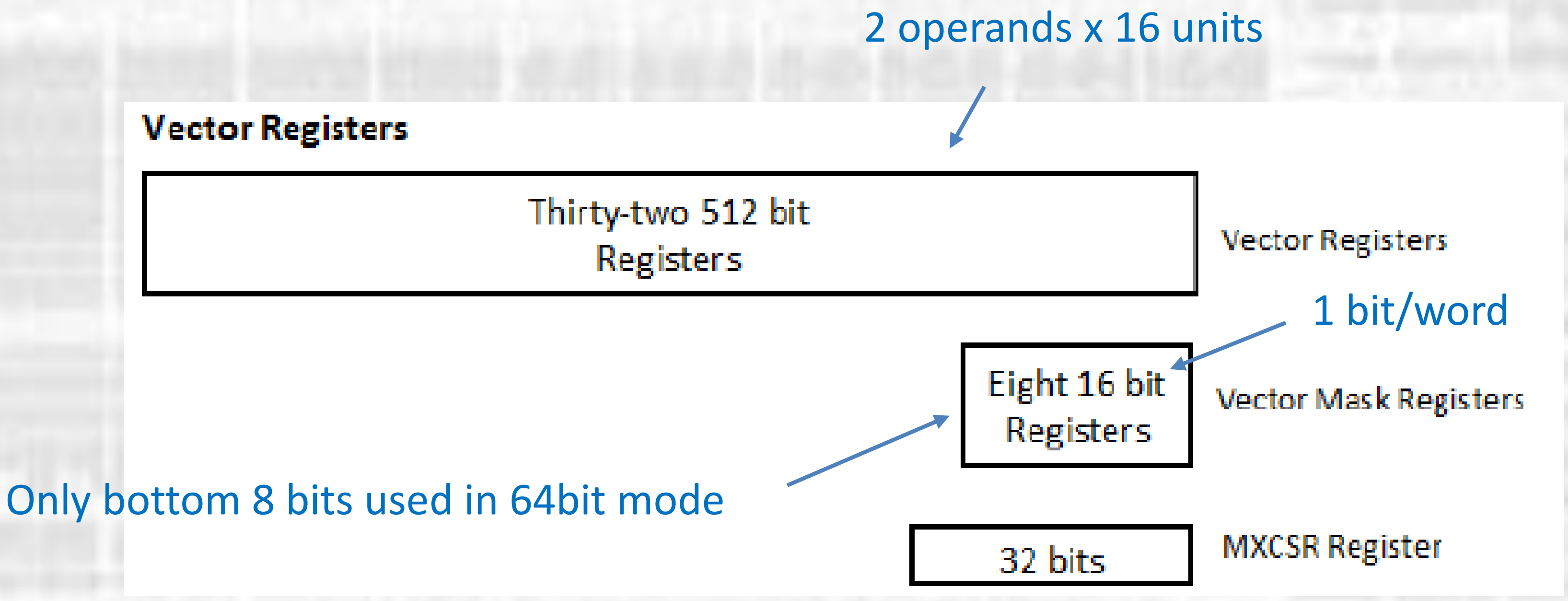
Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor
  - 16 x 32bit
    or
  - 8 x 64bit



DLP
Data Level Parallelism

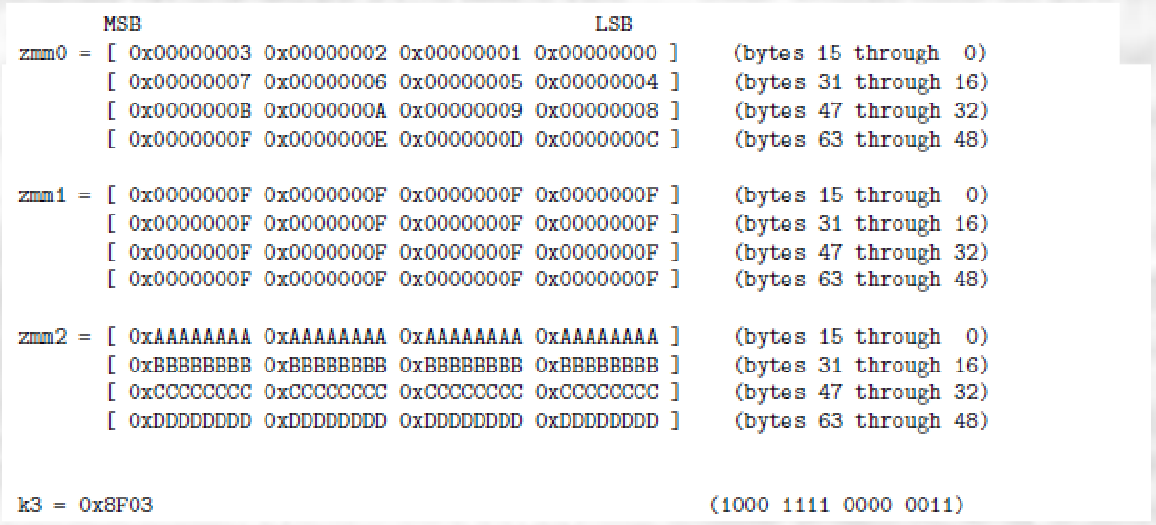
- Vector Processor – modern example
  - Intel Phi Co-processor



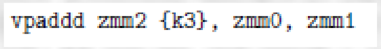
DLP

Data Level Parallelism

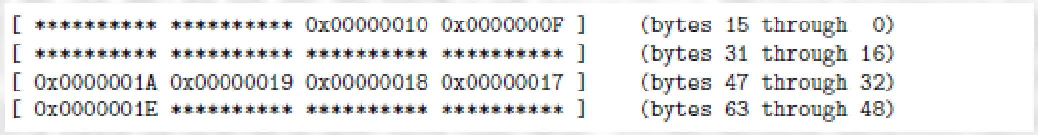
- Vector Processor – modern example
  - Intel Phi Co-processor
    - Vector Mask



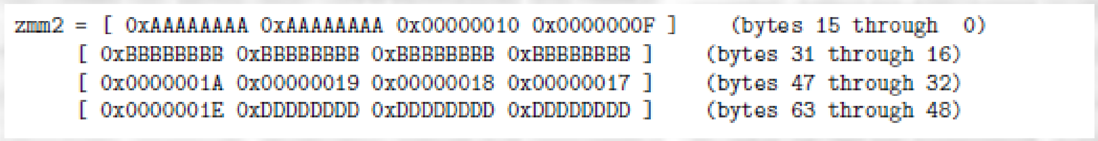
```
       MSB                                            LSB
zmm0 = [ 0x00000003 0x00000002 0x00000001 0x00000000 ]    (bytes 15 through  0)
       [ 0x00000007 0x00000006 0x00000005 0x00000004 ]    (bytes 31 through 16)
       [ 0x0000000B 0x0000000A 0x00000009 0x00000008 ]    (bytes 47 through 32)
       [ 0x0000000F 0x0000000E 0x0000000D 0x0000000C ]    (bytes 63 through 48)

zmm1 = [ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ]    (bytes 15 through  0)
       [ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ]    (bytes 31 through 16)
       [ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ]    (bytes 47 through 32)
       [ 0x0000000F 0x0000000F 0x0000000F 0x0000000F ]    (bytes 63 through 48)

zmm2 = [ 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA 0xAAAAAAAA ]    (bytes 15 through  0)
       [ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ]    (bytes 31 through 16)
       [ 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC 0xCCCCCCCC ]    (bytes 47 through 32)
       [ 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ]    (bytes 63 through 48)


k3 = 0x8F03                                             (1000 1111 0000 0011)
```



```
vpaddd zmm2 {k3}, zmm0, zmm1
```



```
[ ********** ********** 0x00000010 0x0000000F ]    (bytes 15 through  0)
[ ********** ********** ********** ********** ]    (bytes 31 through 16)
[ 0x0000001A 0x00000019 0x00000018 0x00000017 ]    (bytes 47 through 32)
[ 0x0000001E ********** ********** ********** ]    (bytes 63 through 48)
```



```
zmm2 = [ 0xAAAAAAAA 0xAAAAAAAA 0x00000010 0x0000000F ]    (bytes 15 through  0)
       [ 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB 0xBBBBBBBB ]    (bytes 31 through 16)
       [ 0x0000001A 0x00000019 0x00000018 0x00000017 ]    (bytes 47 through 32)
       [ 0x0000001E 0xDDDDDDDD 0xDDDDDDDD 0xDDDDDDDD ]    (bytes 63 through 48)
```

DLP

Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor
  - Peak performance

Clock freq x 8 lanes (64bit mode) x 2 FLOPS/clock x # cores

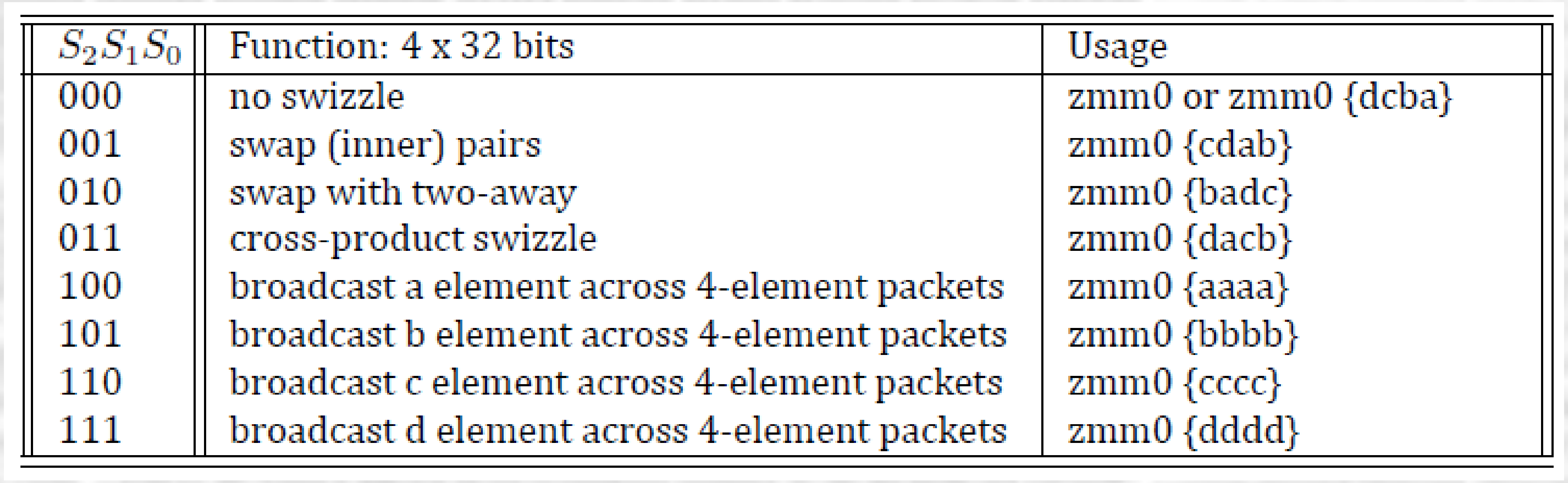
1Ghz x 8 lanes x 2 Flops/clock = 16Gflops / core

60 cores → 960Gflops

DLP

Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor
    - Swizzle / Broadcast
      - 4 word permutations (16 bytes in 32 bit mode, 32 bytes in 64 bit mode)
      - Convers when the operand is loaded



$S_2S_1S_0$	Function: 4 x 32 bits	Usage
000	no swizzle	zmm0 or zmm0 {dcba}
001	swap (inner) pairs	zmm0 {cdab}
010	swap with two-away	zmm0 {badc}
011	cross-product swizzle	zmm0 {dacb}
100	broadcast a element across 4-element packets	zmm0 {aaaa}
101	broadcast b element across 4-element packets	zmm0 {bbbb}
110	broadcast c element across 4-element packets	zmm0 {cccc}
111	broadcast d element across 4-element packets	zmm0 {dddd}

DLP
Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor
    - SP transcendental instructions supported in hardware
      - Exponent
      - Logarithm
      - Reciprocal
      - Square root operations.

DLP

Data Level Parallelism

- Vector Processor – modern example
  - Intel Phi Co-processor
    - Standard instruction format

```
vop   v0{mask}, v1, v2|mem{swizzle},
```