北京大学计算机学院本科生课程

计算机组成与系统结构实习

数据级并行

北京大学微处理器研发中心

2023-11-20

Outline

- DLP
- SIMD
- GPU

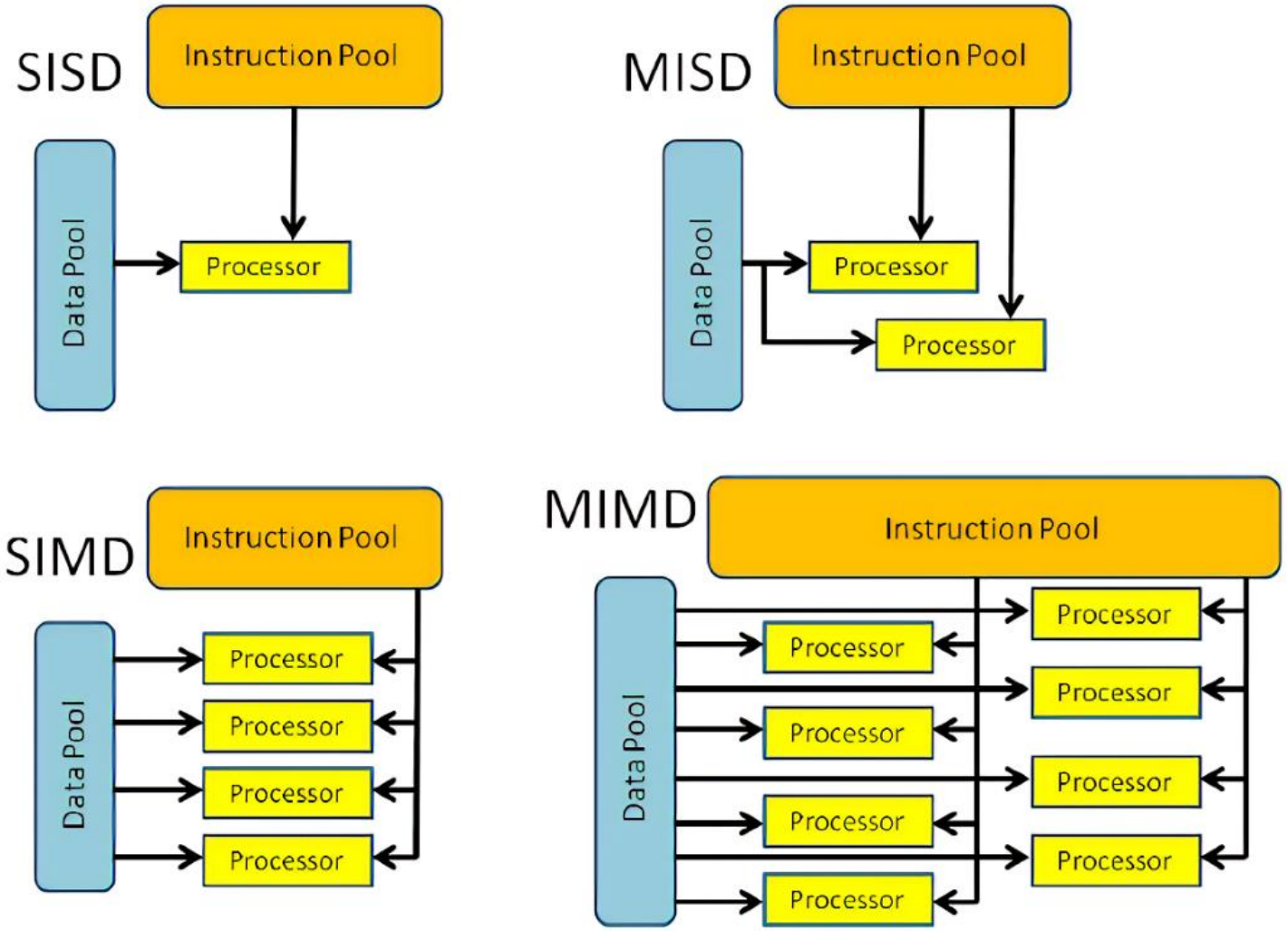
Flynn's Taxonomy

- Classified by data and control streams in 1966

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data (SIMD) (single PC: Vector, CM-2, SSE)
Multiple Instruction Single Data (MISD) (????)	Multiple Instruction Multiple Data (MIMD) (Clusters, SMP servers)

- SISD ⇒ Implicit Instruction Level Parallelism (ILP)
- SIMD ⇒ Data Level Parallelism (DLP)
- MIMD ⇒ Thread Level Parallelism (TLP)

Flynn's Taxonomy



Data Level Parallelism (DLP)

We call these algorithms data parallel because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control.

– W. Daniel Hillis and Guy L. Steele
"Data Level Algortihms", C. ACM

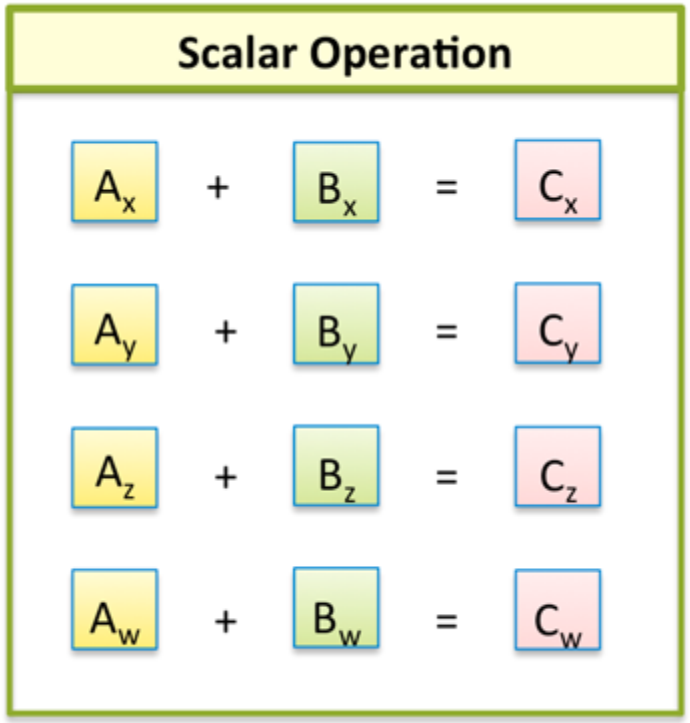
Vectors Motivation

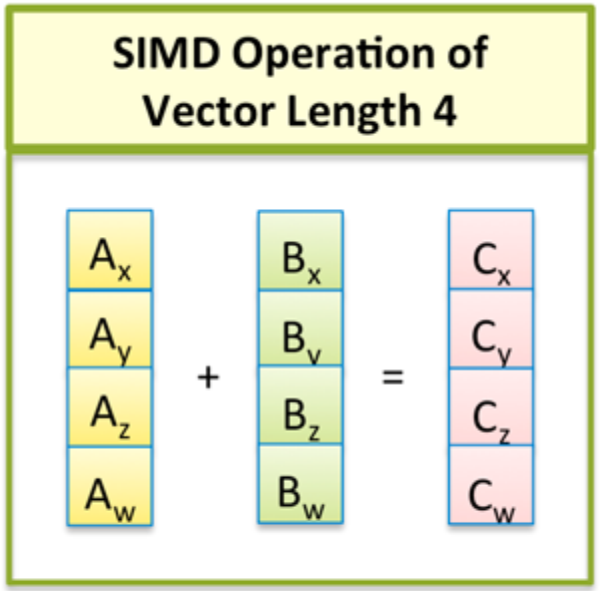
```
for (int i = 0; i < N; i++)
    A[i] = B[i] + C[i];
```

CPU Data-level Parallelism

- Let's make the execution unit (ALU) really wide
- Let's make the registers really wide too

```
for (int i = 0; i < N; i+= 4) {
    // in parallel
    A[i] = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
    A[i+3] = B[i+3] + C[i+3];
}
```





Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Examples

- PyTorch

```
1   import torch
2
3   SIZE = 100
4   a = torch.rand(SIZE)
5   b = torch.rand(SIZE)
6   c = a+b
7   d = a.sum()
```

- Scala

```
val vector = Vector(1, 2, 3, 4, 5, 6)
val plusOne = vector.map(x => x + 1)
val sum = vector.reduce((x, y) => x + y)
```

  - Functional programming
    - map , reduce
  - Compared with PyTorch
    - c=a+b -> ‘map’ (element-wise op)
    - d=a.sum() -> ‘reduce’ (reducing a vector into a scalar)
  - Scales to distributed systems

Examples

- X86 AVX intrinsics
  - Platform-dependent
  - Add (map operation)
  - _m256 contains float *8

```cpp
#include <immintrin.h>

constexpr int WIDTH = 256 / 8 / sizeof(float);

// c = a + b
// assumption:
//      1. a, b, c are aligned by 32 Bytes
//      2. N % WIDTH == 0
void vectorAdd(const float a[], const float b[], float c[], int N) {
    for (auto i: int = 0; i < N; i += WIDTH) {
        auto va: __m256 = _mm256_load_ps(p: &a[i]);
        auto vb: __m256 = _mm256_load_ps(p: &b[i]);
        auto vc: __m256 = va + vb;
        _mm256_store_ps(p: &c[i], a: vc);
    }
}
```

  - Sum (reduce operation)
  - Ignoring non-associativity of float

```cpp
// assumption:
//      1. a is aligned by 32 Bytes
//      2. N % WIDTH == 0
float vectorSum(const float a[], int N) {
    __m256 partialSum {};
    for (auto i: int = 0; i < N; i += WIDTH) {
        auto va: __m256 = _mm256_load_ps(p: &a[i]);
        partialSum += va;
    }
    auto res: float = 0.0f;
    // will be optimized to log(WIDTH) time by -Ofast
    for (auto i: int = 0; i < WIDTH; i++) {
        res += partialSum[i];
    }
}
```

DLP Alternatives

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography

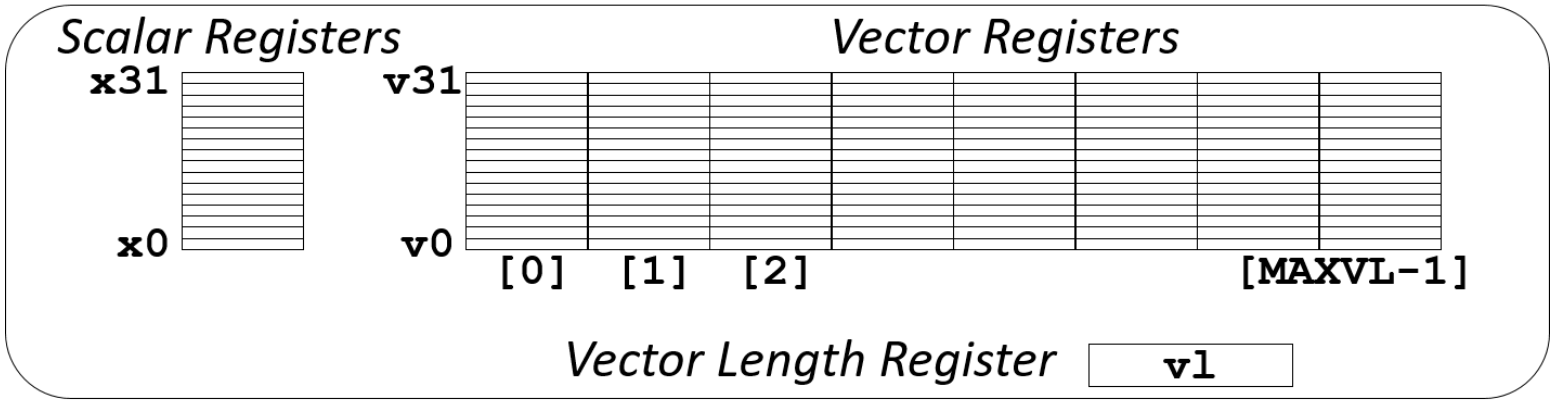
- All involve huge computations on large data set
- Supercomputers: CDC6600, CDC7600, Cray-1, …
- In 70s-80s, Supercomputer ≡ Vector Machine

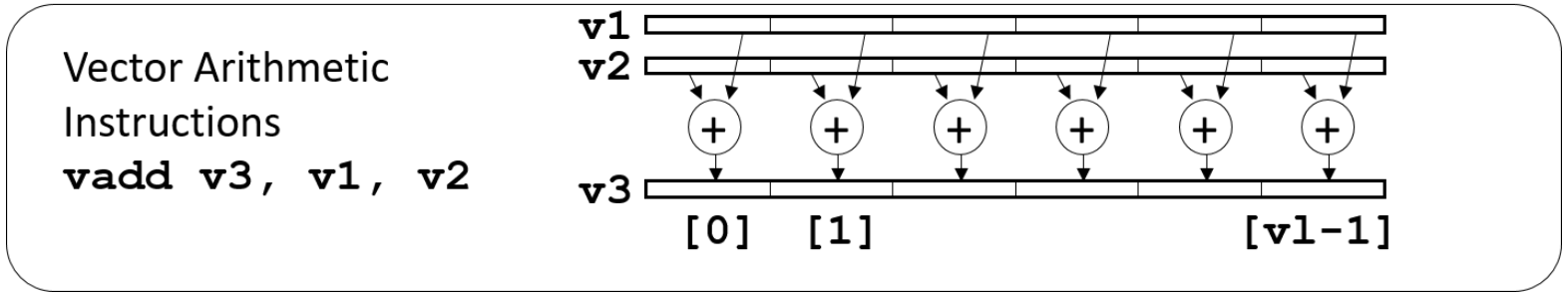
Vector Supercomputers

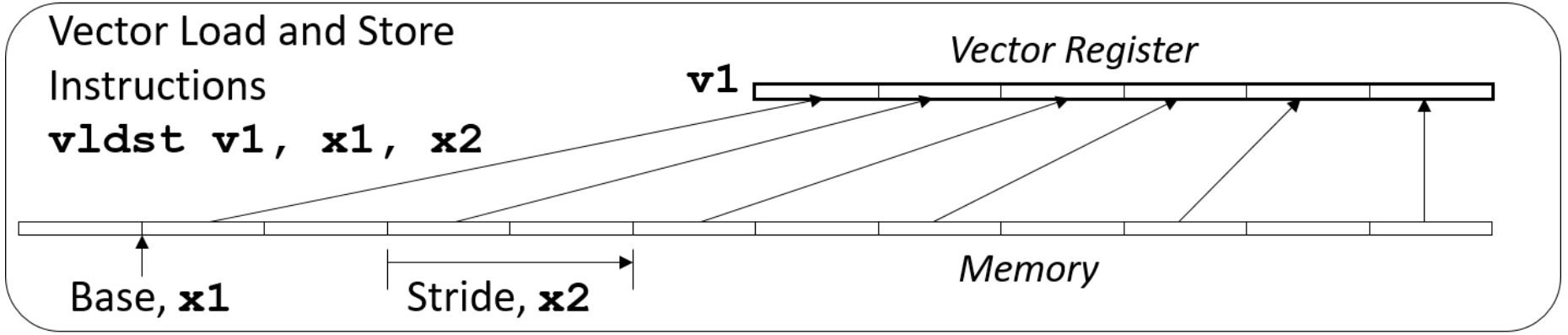
[©Cray Research, 1976]

- Epitomized by Cray-1, 1976
- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Vector Programming Model







Vector Code Example

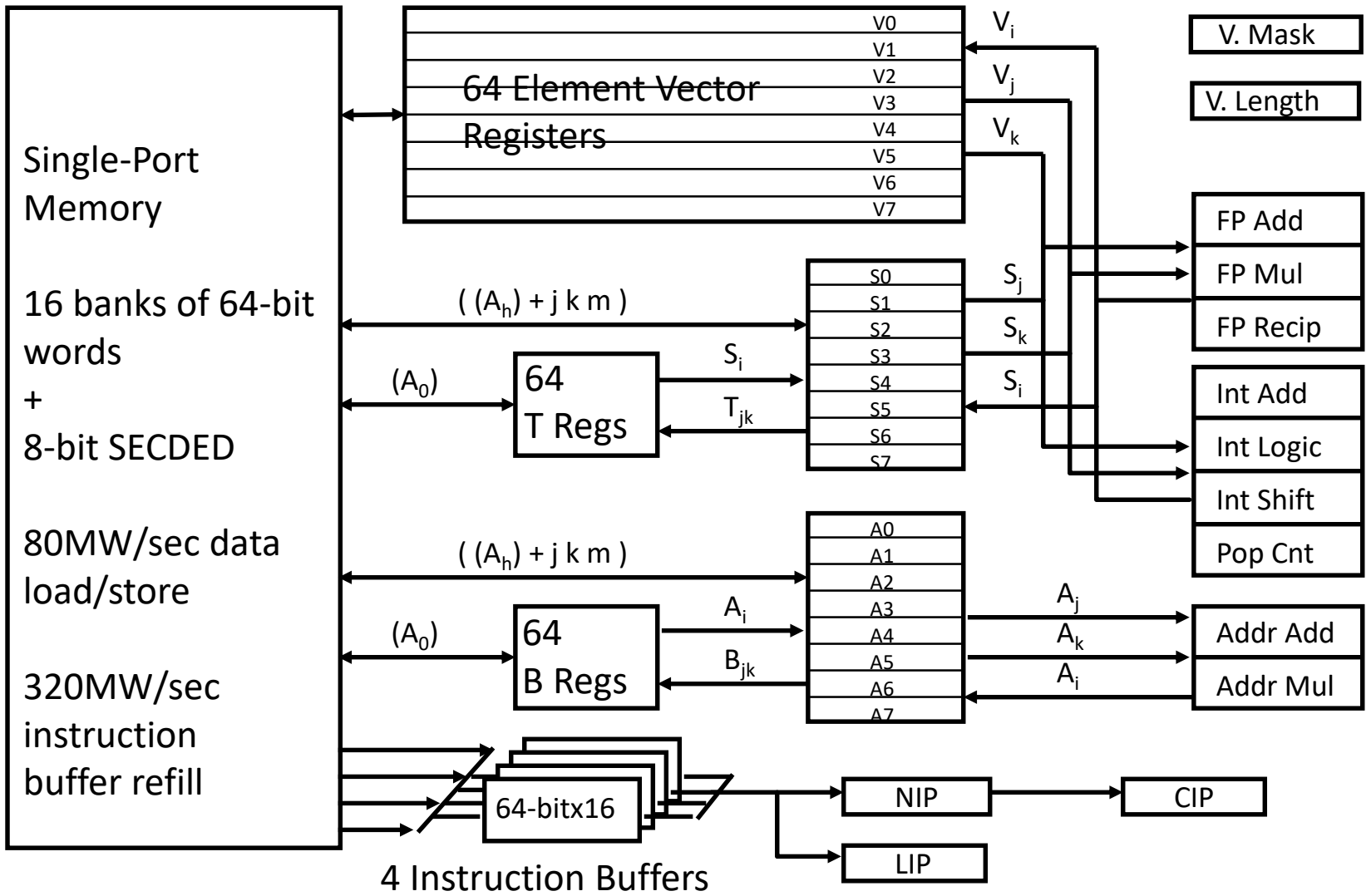
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, 8
  addi x2, 8
  addi x3, 8
  subi x4, 1
  bnez x4, loop
```

```
# Vector Code
  li x4, 64
  setvl x4
  vld v1, x1
  vld v2, x2
  vadd v3,v1,v2
  vst v3, x3
```

区别在哪儿？？？

Cray-1 (1976)



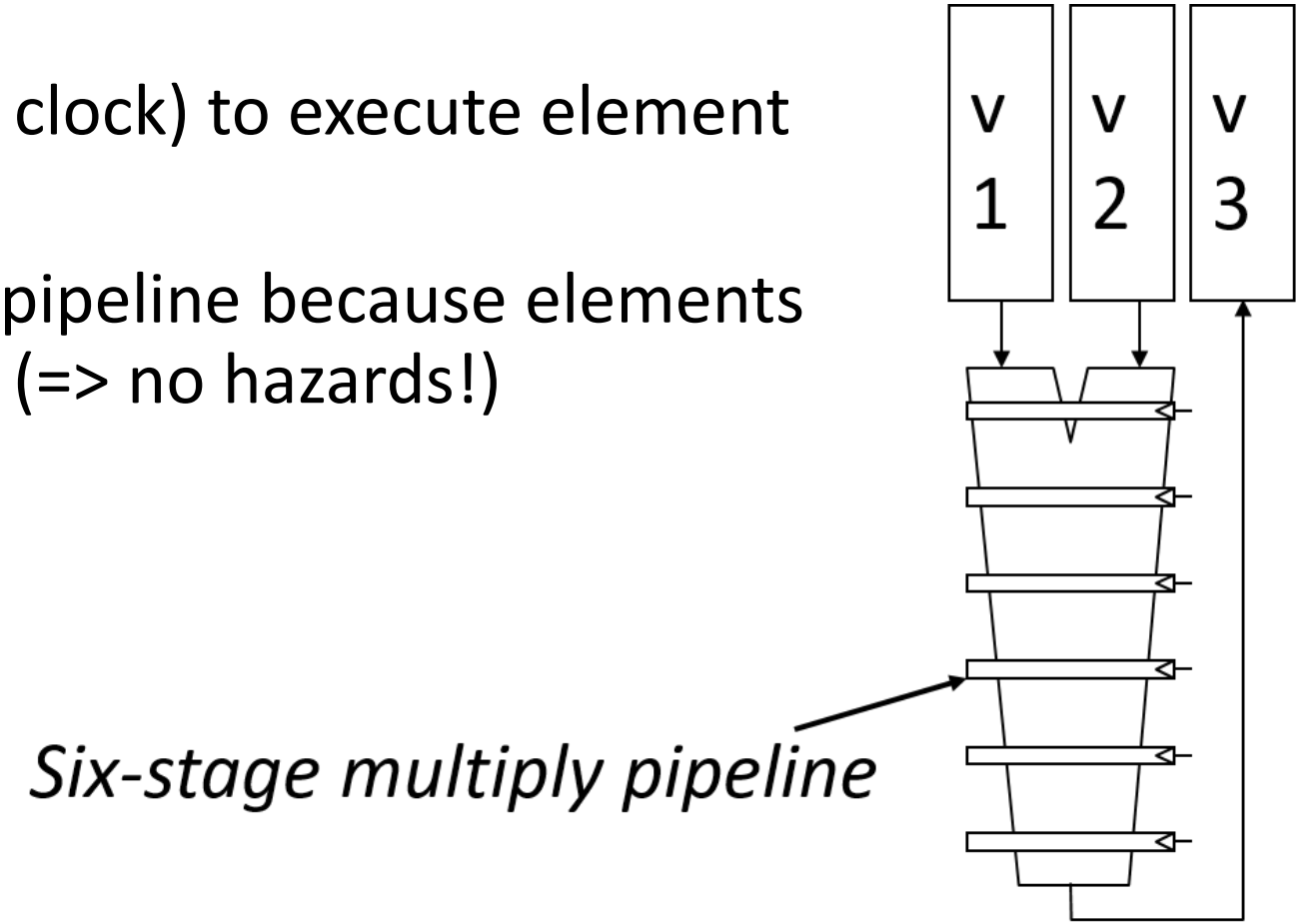
- *memory bank cycle* 50 ns
- *processor cycle* 12.5 ns (80MHz)

Vector Instruction Set Advantages

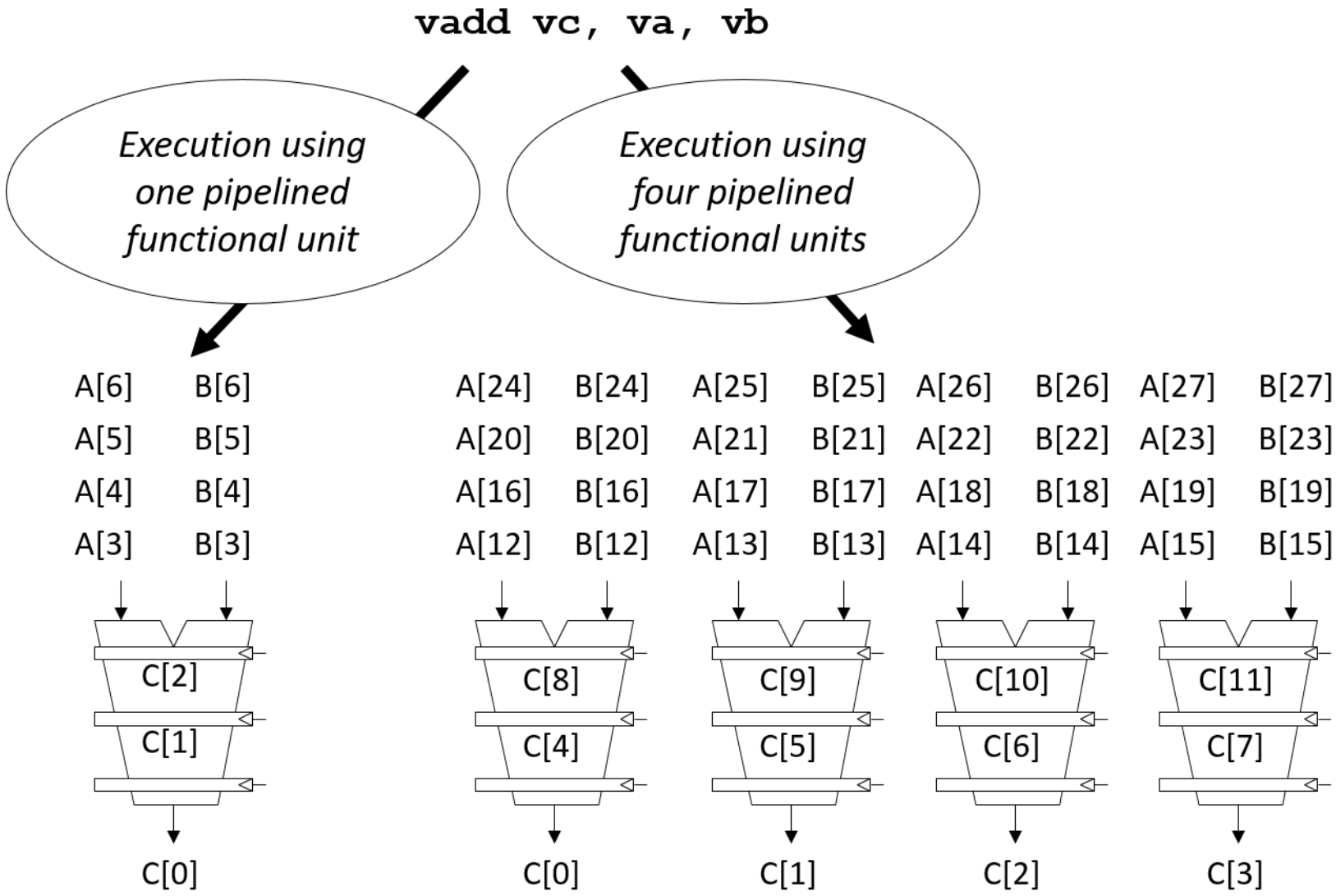
- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same code on more parallel pipelines (lanes)

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)

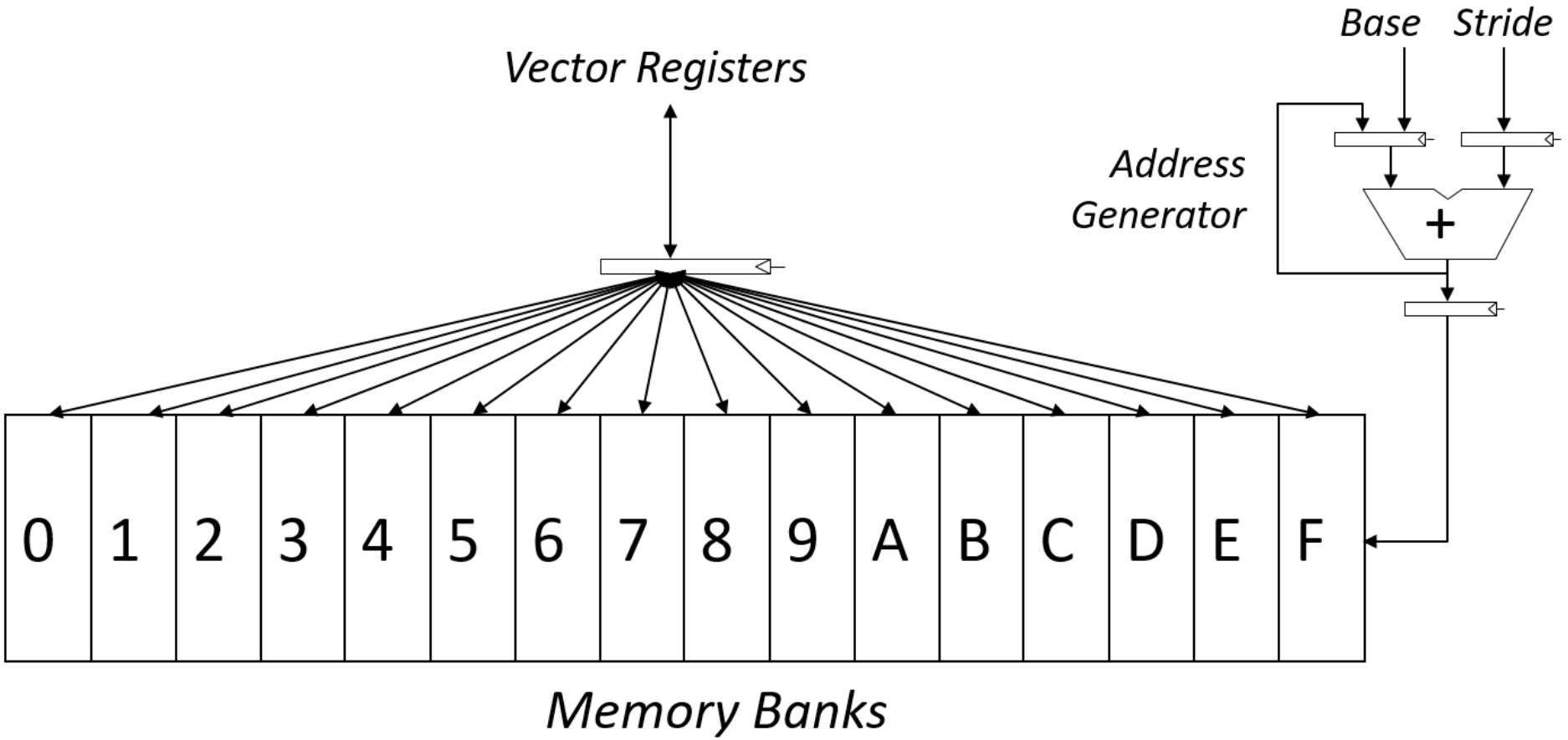


Vector Instruction Execution

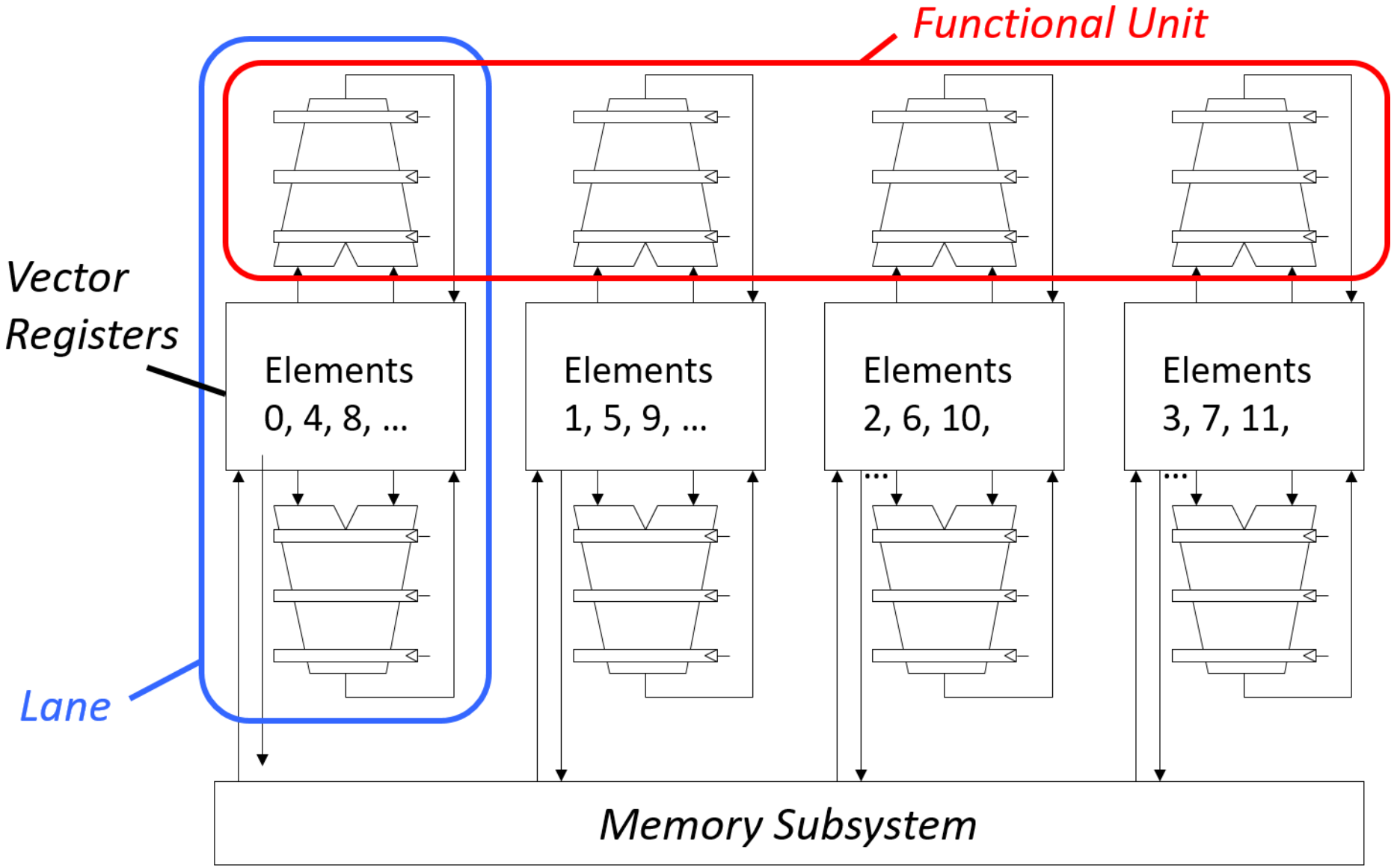


Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

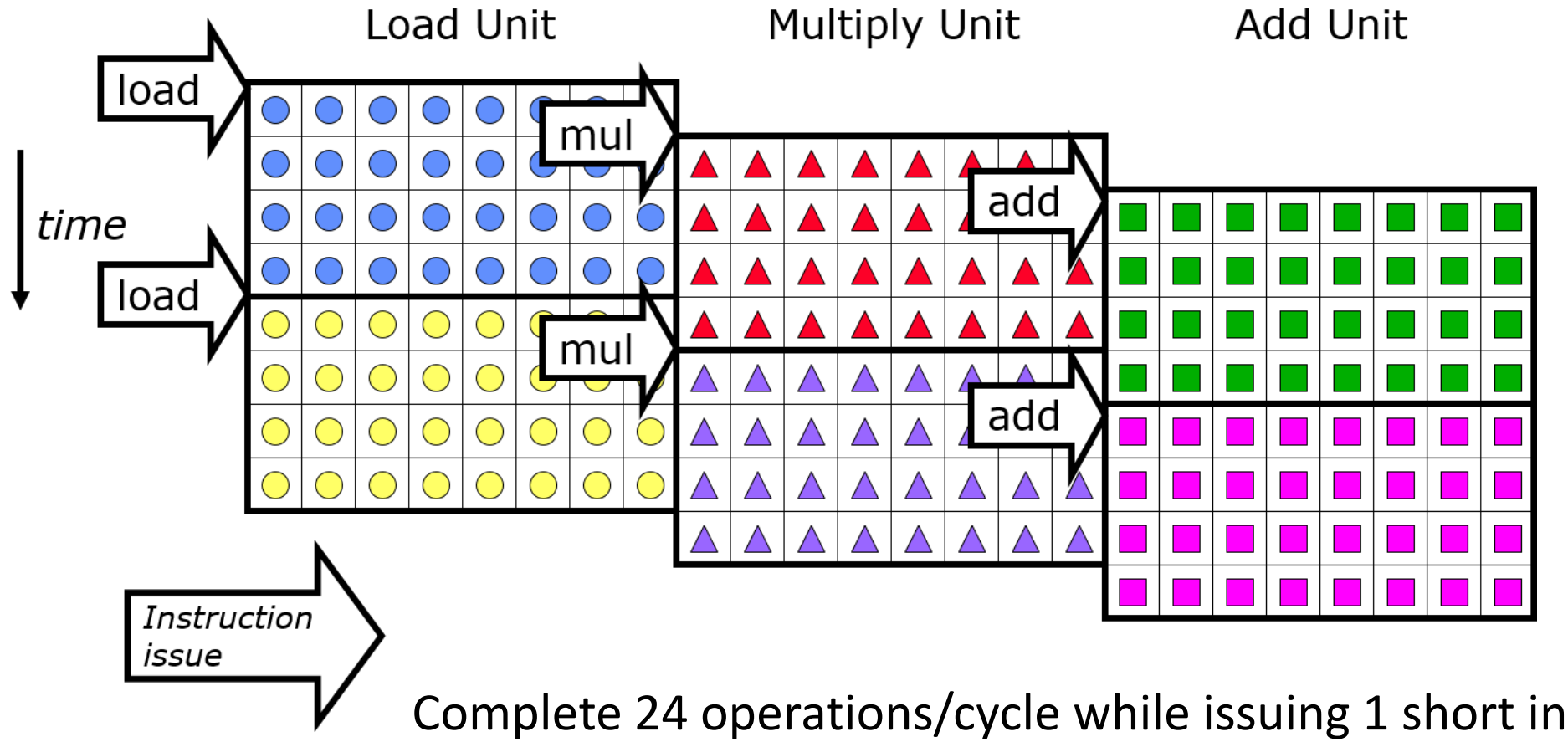


Vector Unit Structure



Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes

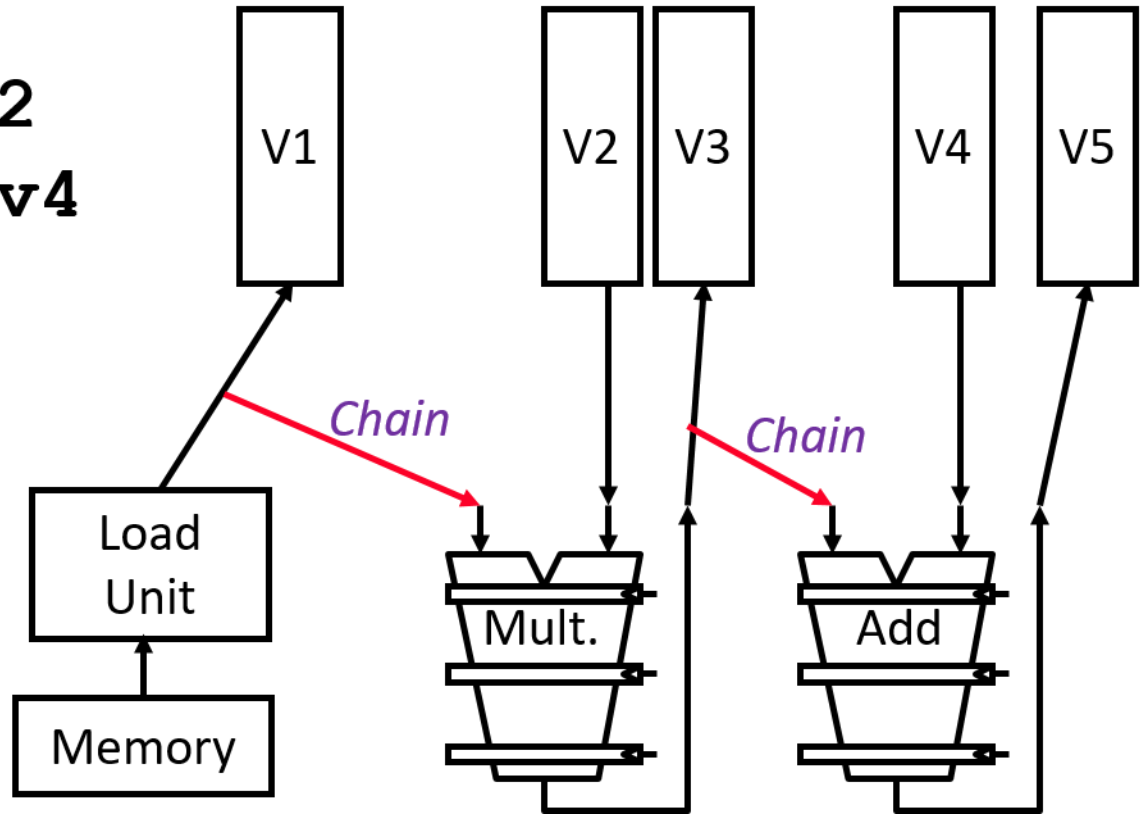


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

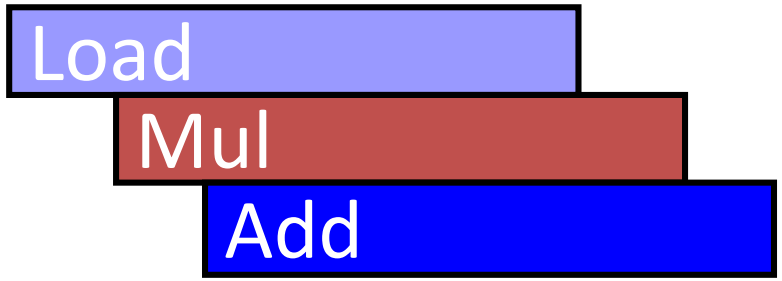
```
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```



Vector Chaining Advantage

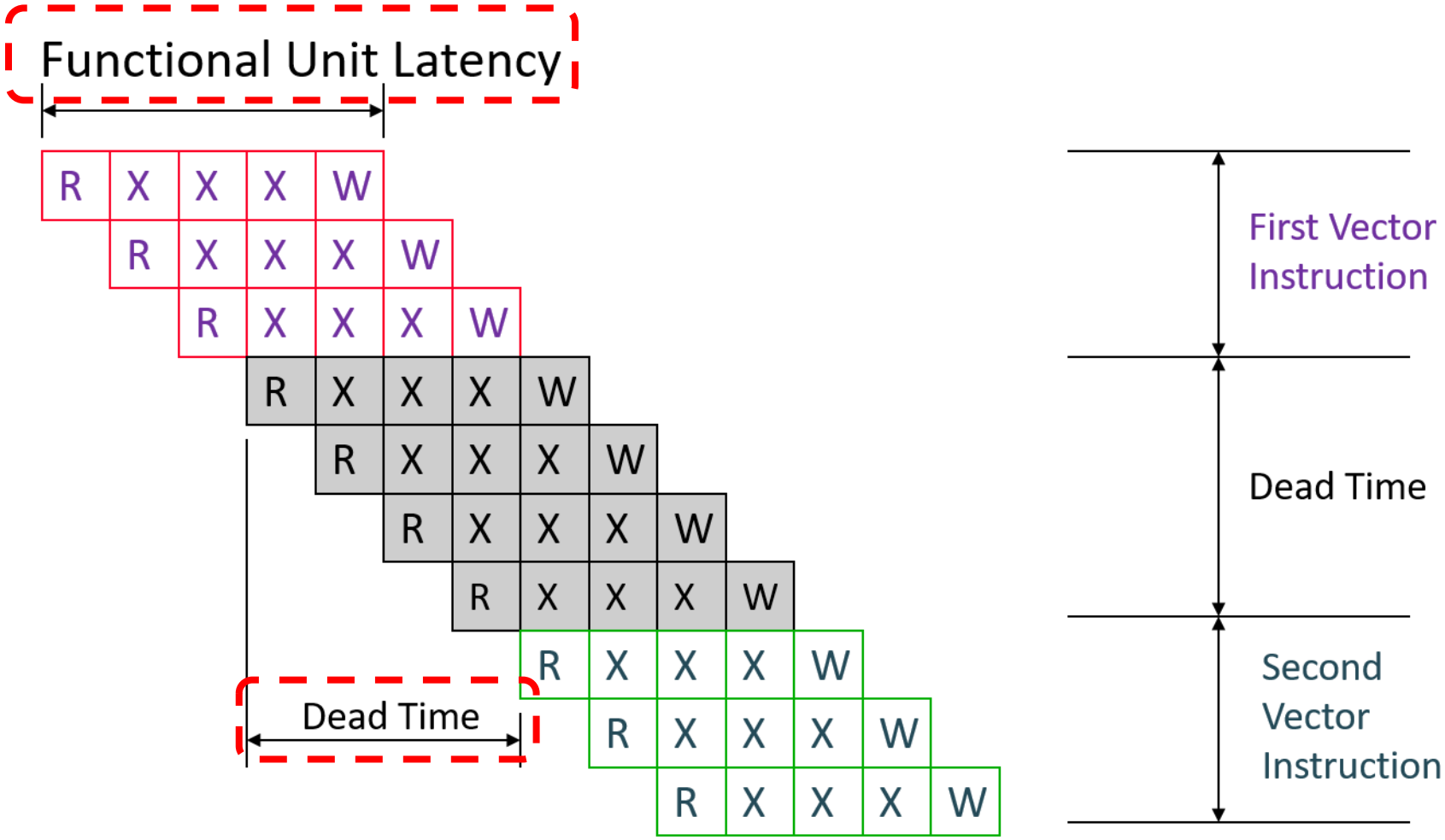
- Without chaining, must wait for last element of result to be written before starting dependent instruction

- With chaining, can start dependent instruction as soon as first result appears

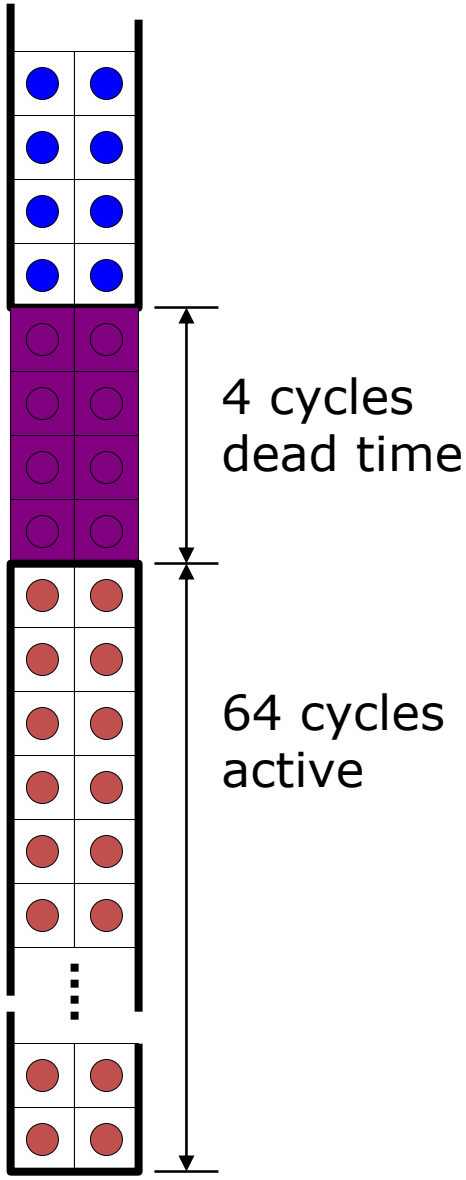


Vector Startup

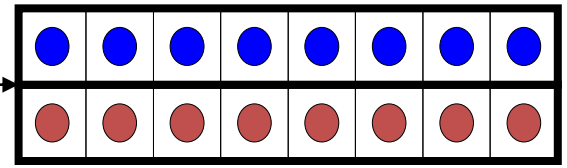
- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline)



Dead Time and Short Vectors



No dead time

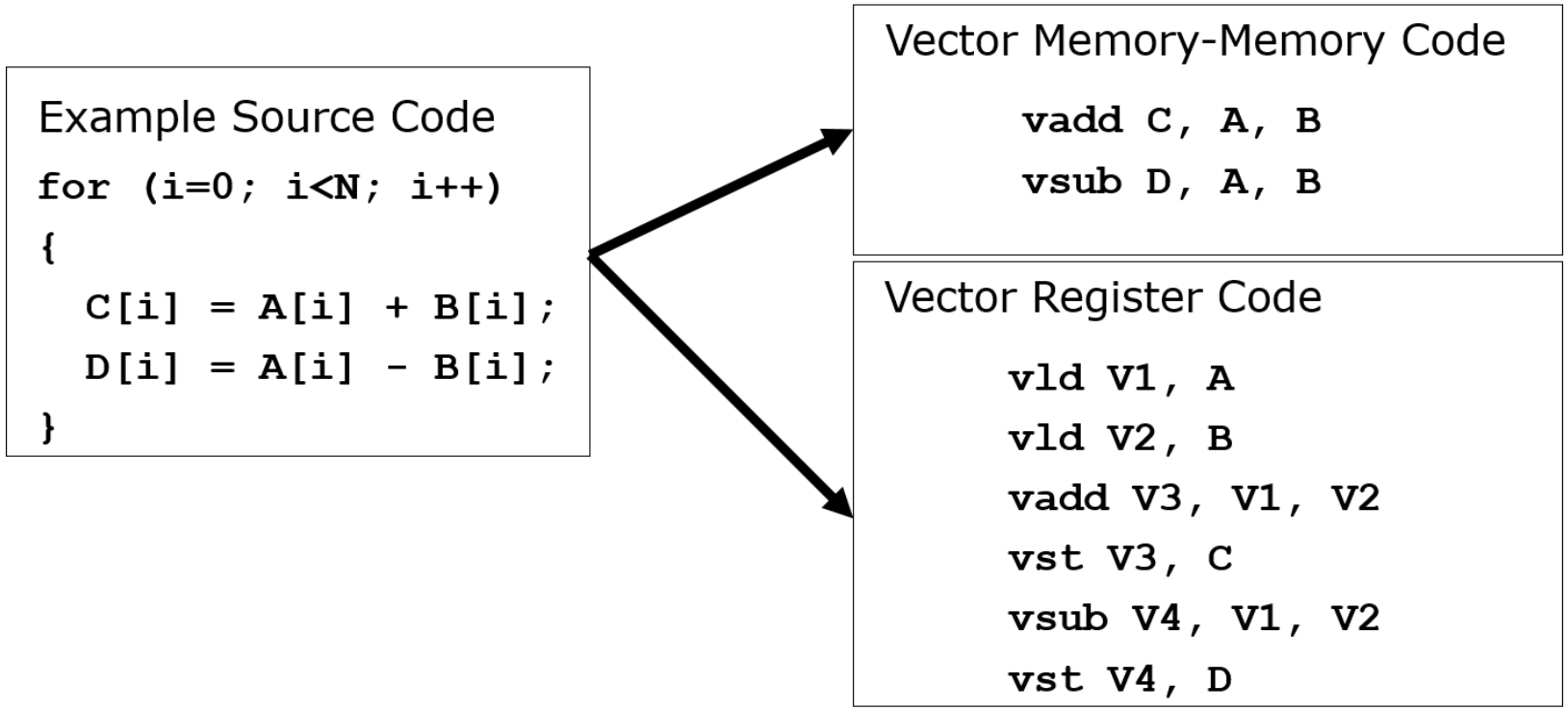


T0, Eight lanes
No dead time
100% efficiency with 8-element vectors

Cray C90, Two lanes
4-cycle dead time
Maximum efficiency 94% with 128-element vectors

Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 (1973) and TI ASC (1971), were memory-memory machines
- Cray-1 (1976) was first vector register machine



Vector Memory-Memory versus Vector Register Machines

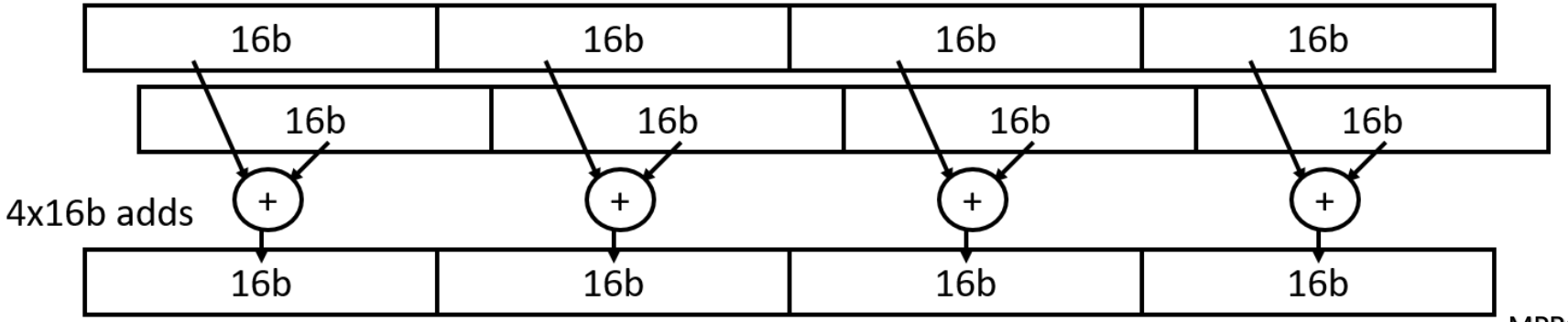
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2-4 elements
- Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
- (we ignore vector memory-memory from now on)

Outline

- DLP
- SIMD
- GPU

Packed SIMD Extensions

- Short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX
- Single instruction operates on all elements within register



SIMD in x86

- MMX (Multimedia Extension)
  - 1996, Pentium, 57 instructions
  - 64-bit MMX registers (borrowed from FP registers)
  - eight 8-bit / four 16-bit / two 32-bit operation
- SSE ~ SSE4 (Streaming SIMD Extension)
  - 1999~, Pentium III
  - 128-bit registers
  - FP: two 32-bit single precision, two 64-bit double precision
  - Int: four 32-bit, eight 16-bit, sixteen char
- AVX ~AVX2 (Advanced Vector Extensions)
  - 2011~, Sandy Bridge
  - AVX: 256-bit registers, AVX2: 512-bit registers
  - Three-operand, relaxed alignment of memory operands

MMX

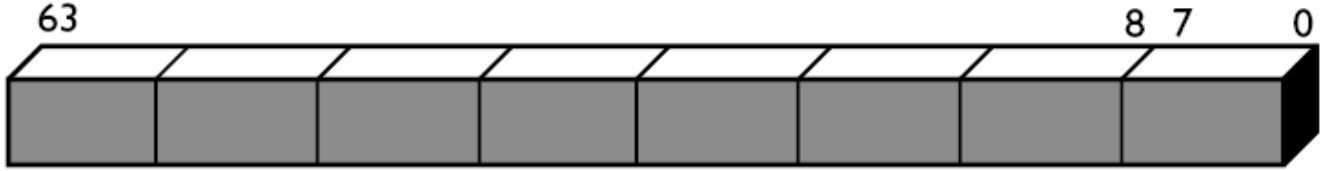
- After analyzing a lot of existing applications such as graphics, MPEG, music, speech recognition, game, image processing, they found that many multimedia algorithms execute the same instructions on many pieces of data in a large data set

- Typical elements are small, 8 bits for pixels, 16 bits for audio, 32 bits for graphics and general computing

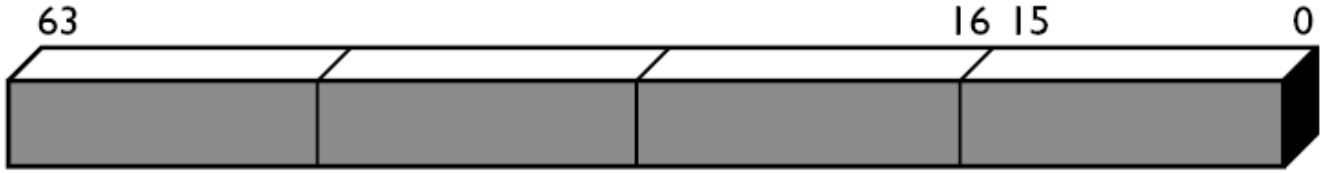
- New data type: 64-bit packed data type. Why 64 bits?
  - Good enough
  - Practical

MMX Data Types

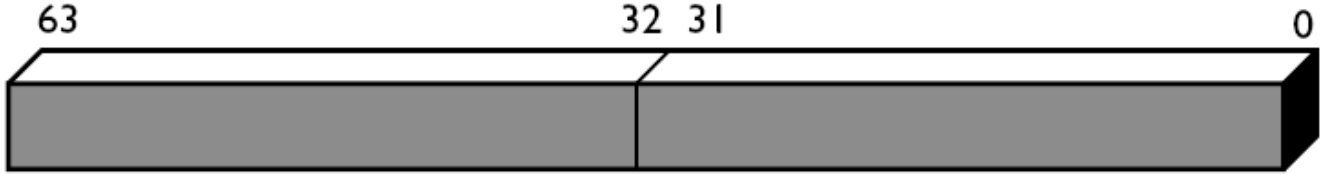
Packed Byte: 8 bytes packed into 64 bits



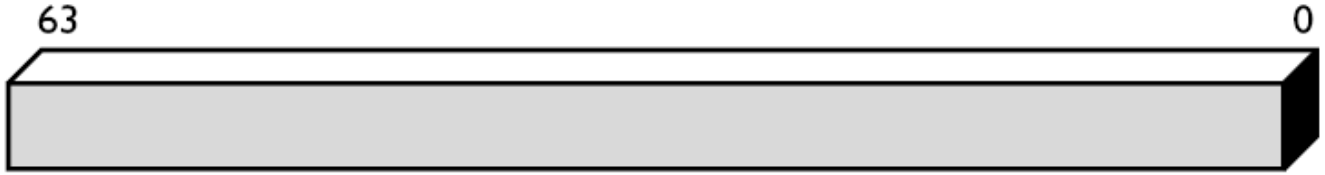
Packed Word: 4 words packed into 64 bits



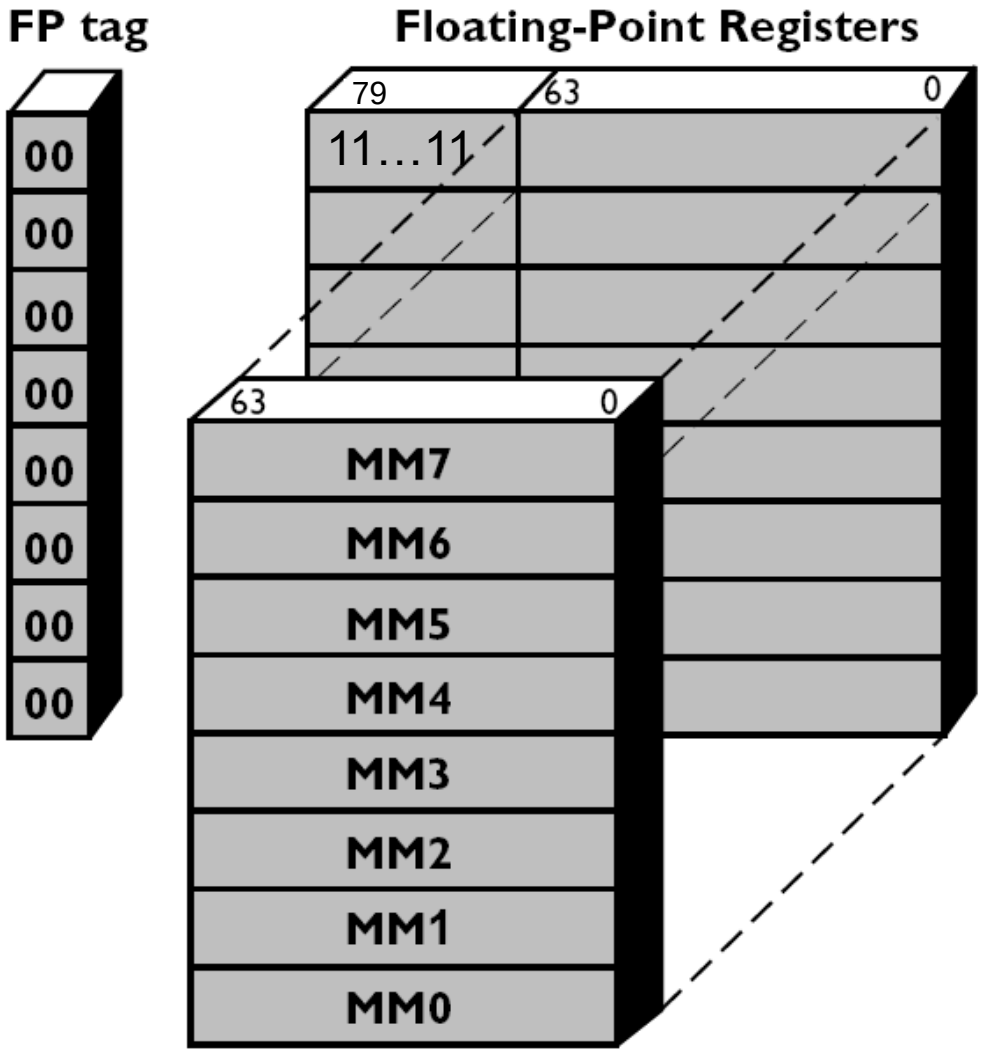
Packed Doubleword: 2 doublewords packed into 64 bits



Packed Quadword: One 64-bit quantity



MMX Registers



8 **MMX Registers** MM0~MM7

No new mode or state was created. Hence, for context switching, no extra state needs to be saved.

It means MMX and FPU cannot be used at the same time. Big overhead to switch

MMX Instructions

Category		Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADDB, PADDW, PADDD	PADDSB, PADDSW	PADDUSB, PADDUSW
	Subtraction	PSUBB, PSUBW, PSUBD	PSUBSB, PSUBSW	PSUBUSB, PSUBUSW
	Multiplication	PMULL, PMULH		
	Multiply and Add	PMADD		
Comparison	Compare for Equal	PCMPEQB, PCMPEQW, PCMPEQD		
	Compare for Greater Than	PCMPGTPB, PCMPGTPW, PCMPGTPD		
Conversion	Pack		PACKSSWB, PACKSSDW	PACKUSWB
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ		
	Unpack Low	PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ		

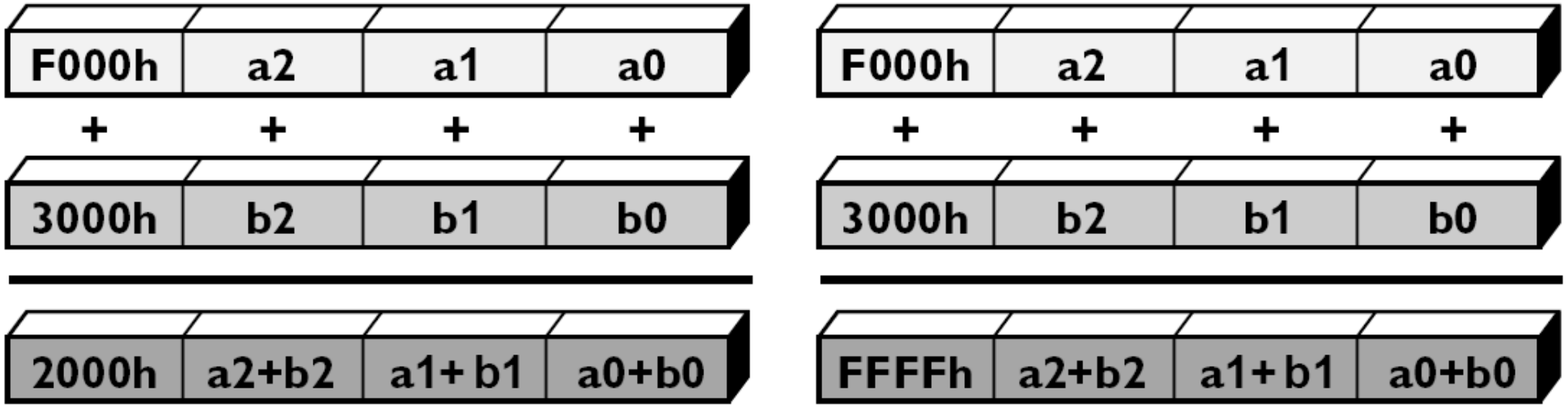
MMX Instructions

		Packed	Full Quadword
Logical	And And Not Or Exclusive OR		PAND PANDN POR PXOR
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD	PSLLQ PSRLQ
		Doubleword Transfers	**Quadword Transfers**
Data Transfer	Register to Register Load from Memory Store to Memory	MOVD MOVD MOVD	MOVQ MOVQ MOVQ
Empty MMX State		EMMS	

Call it before you switch to FPU from MMX;
Expensive operation

Saturation arithmetic

- Useful in Graphics Applications



wrap-around

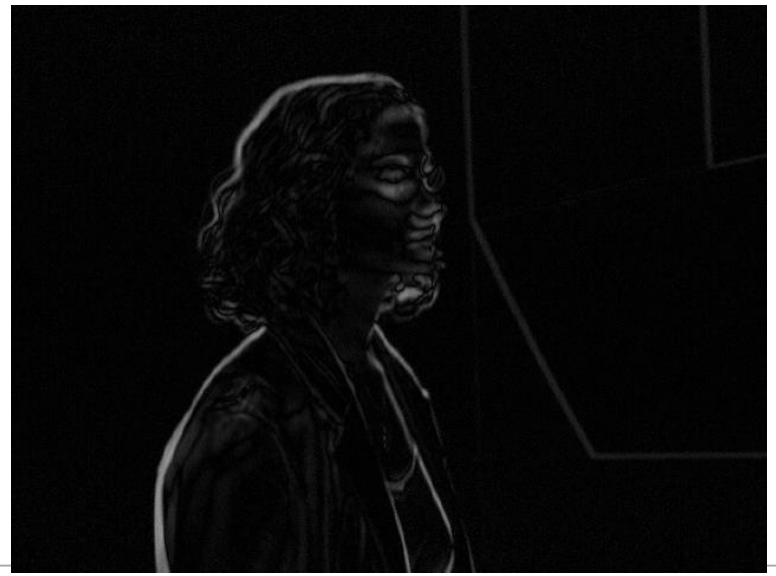
saturating

Example: add a constant to a vector

```
char d[]={5, 5, 5, 5, 5, 5, 5, 5};
char clr[]={65,66,68,...,87,88}; // 24 bytes
__asm{
    movq mm1, d
    mov cx, 3
    mov esi, 0
L1: movq mm0, clr[esi]
    paddb mm0, mm1
    movq clr[esi], mm0
    add esi, 8
    loop L1
    emms
}
```

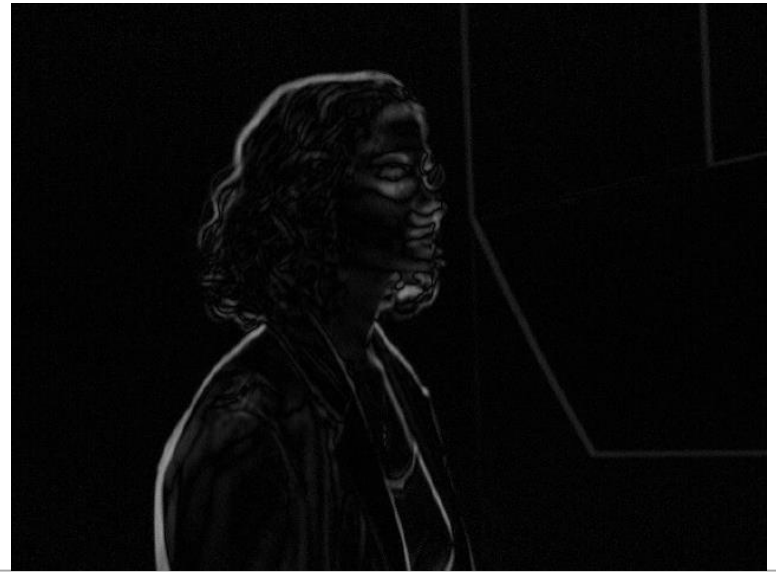
Application: frame difference

|A-B|



Application: frame difference

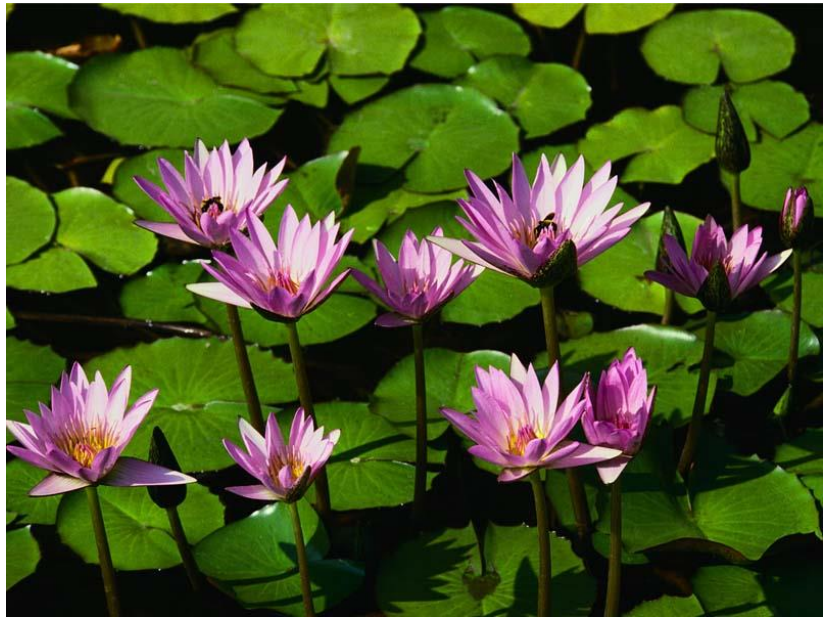
(A-B) or (B-A)



Application: frame difference

```
MOVQ      mm1, A //move 8 pixels of image A
MOVQ      mm2, B //move 8 pixels of image B
MOVQ      mm3, mm1 // mm3=A
PSUBSB    mm1, mm2 // mm1=A-B
PSUBSB    mm2, mm3 // mm2=B-A
POR       mm1, mm2 // mm1=|A-B|
```

Example: image fade-in-fade-out



A

B

$$A*\alpha+B*(1-\alpha) = B+\alpha(A-B)$$

α=1

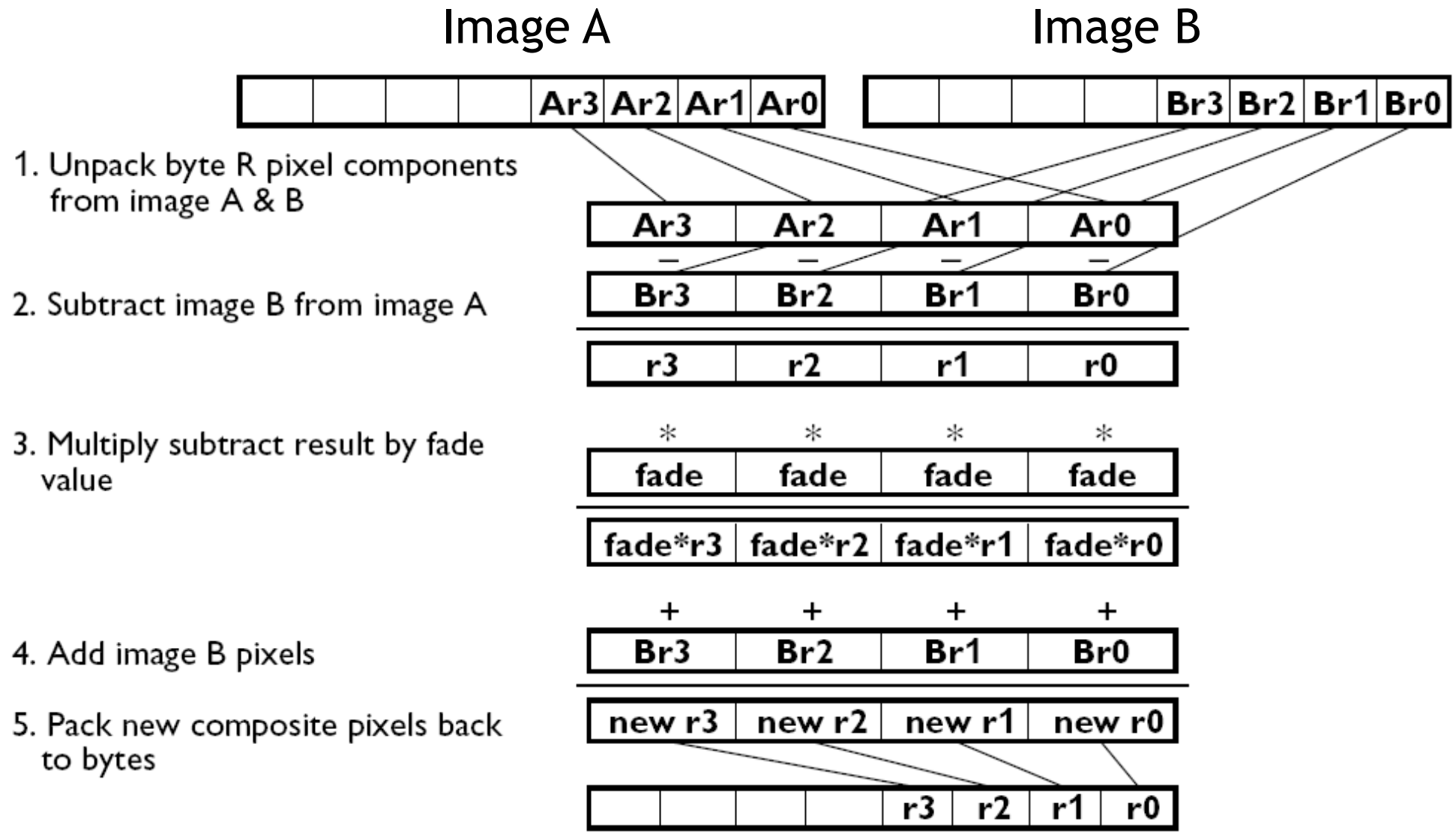
α=0.75

α=0.5

α=0.25

α=0

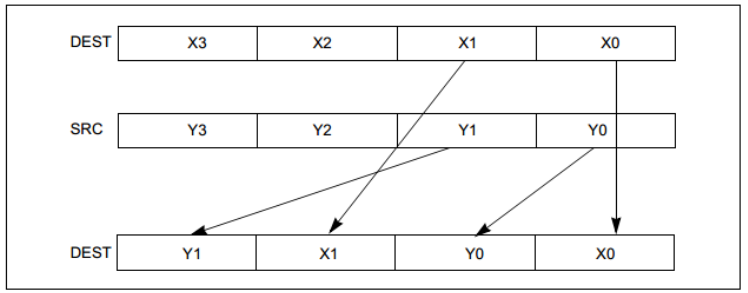
Example: image fade-in-fade-out



Example: image fade-in-fade-out

```
MOVQ       mm0, alpha//4 16-b zero-padding α
MOVD       mm1, A //move 4 pixels of image A
MOVD       mm2, B //move 4 pixels of image B
PXOR       mm3, mm3 //clear mm3 to all zeroes
//unpack 4 pixels to 4 words
PUNPCKLBW mm1, mm3 // Because B-A could be
PUNPCKLBW mm2, mm3 // negative, need 16 bits
PSUBW      mm1, mm2 //(B-A)
PMULHW     mm1, mm0 //(B-A)*fade/256
PADDW      mm1, mm2 //(B-A)*fade + B
//pack four words back to four bytes
PACKUSWB   mm1, mm3
```

unpack



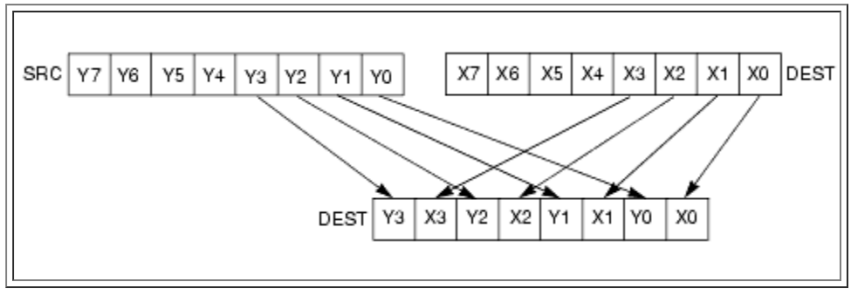
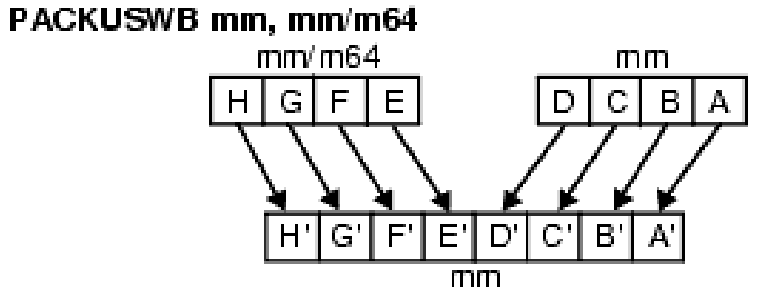


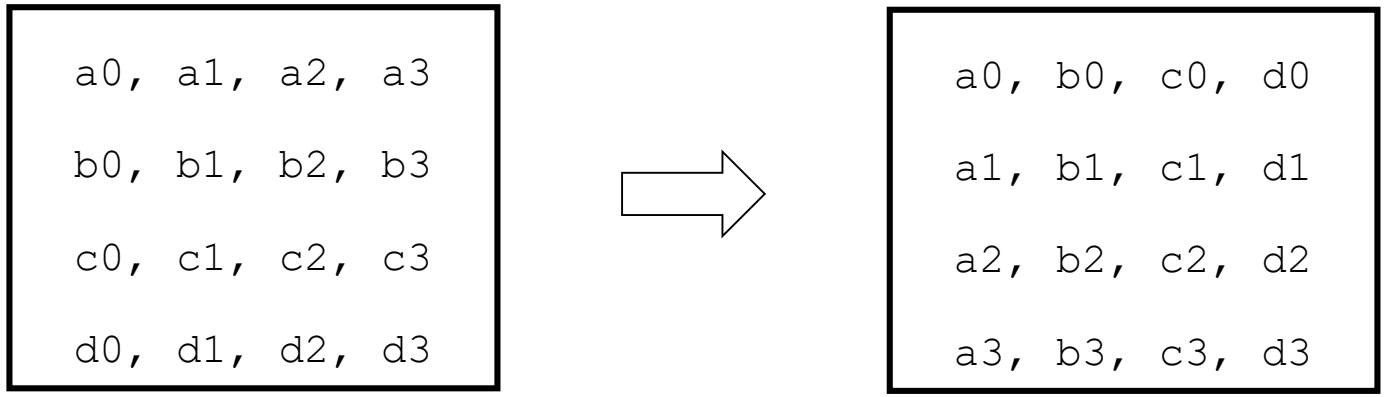
图 3-15. PUNPCKLBW 指令操作



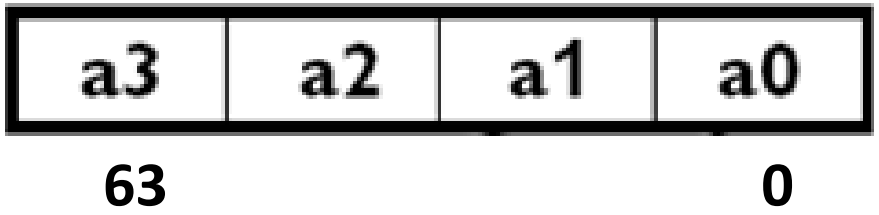
3006014

Application: matrix transpose

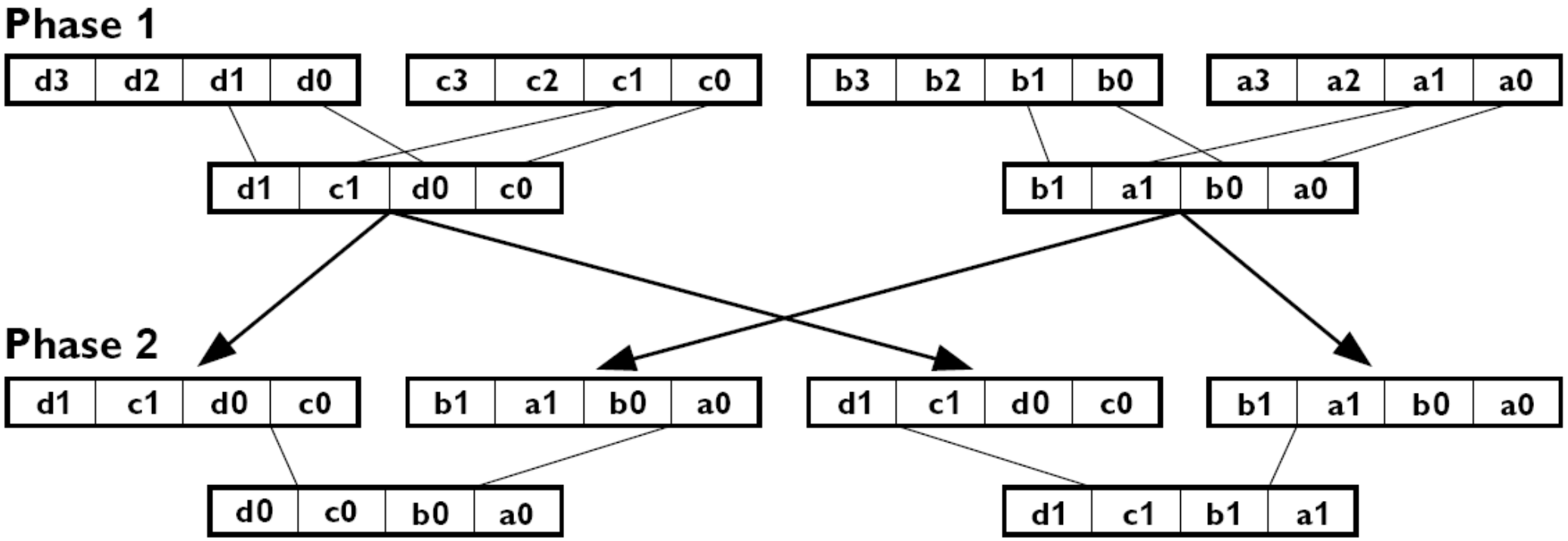
Elements: 16-bit intergers



Little endian: LSB => Lower address



Application: matrix transpose

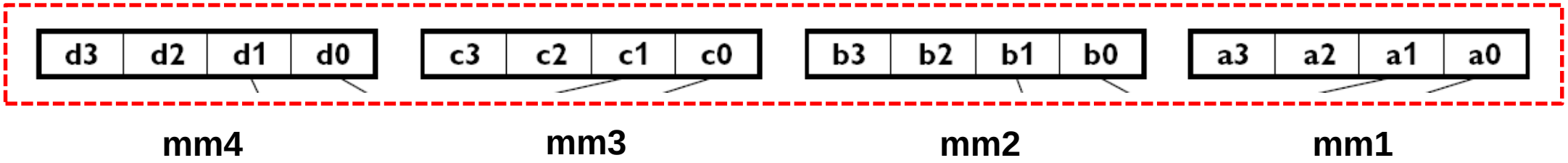


Note: Repeat for the other rows to generate ($[d_3, c_3, b_3, a_3]$ and $[d_2, c_2, b_2, a_2]$).

Application: matrix transpose

MMX code sequence operation:

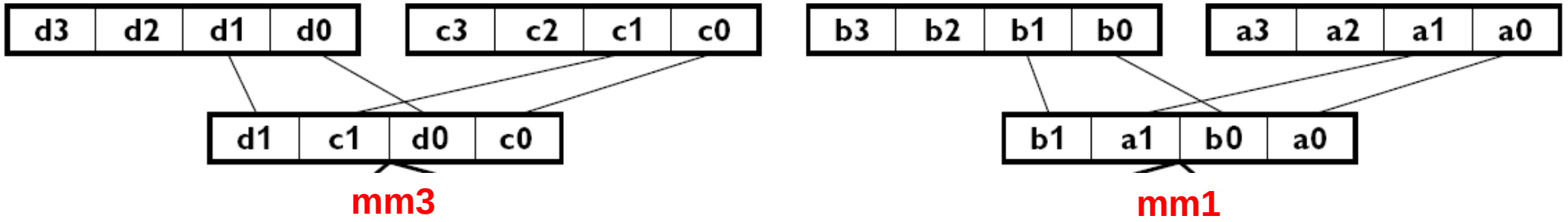
```
movq        mm1, row1     ; load pixels from first row of matrix
movq        mm2, row2     ; load pixels from second row of matrix
movq        mm3, row3     ; load pixels from third row of matrix
movq        mm4, row4     ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2      ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd   mm3, mm4      ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq        mm5, mm1      ; copy mm1 to mm5
punpckldq   mm1, mm3      ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq   mm5, mm3      ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```



Application: matrix transpose

MMX code sequence operation:

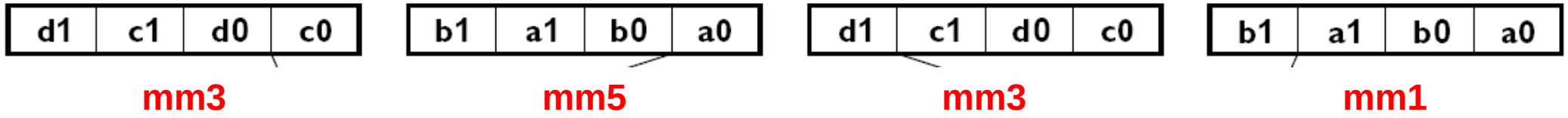
```
movq        mm1, row1     ; load pixels from first row of matrix
movq        mm2, row2     ; load pixels from second row of matrix
movq        mm3, row3     ; load pixels from third row of matrix
movq        mm4, row4     ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2      ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd   mm3, mm4      ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq        mm5, mm1      ; copy mm1 to mm5
punpckldq   mm1, mm3      ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq   mm5, mm3      ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```



Application: matrix transpose

MMX code sequence operation:

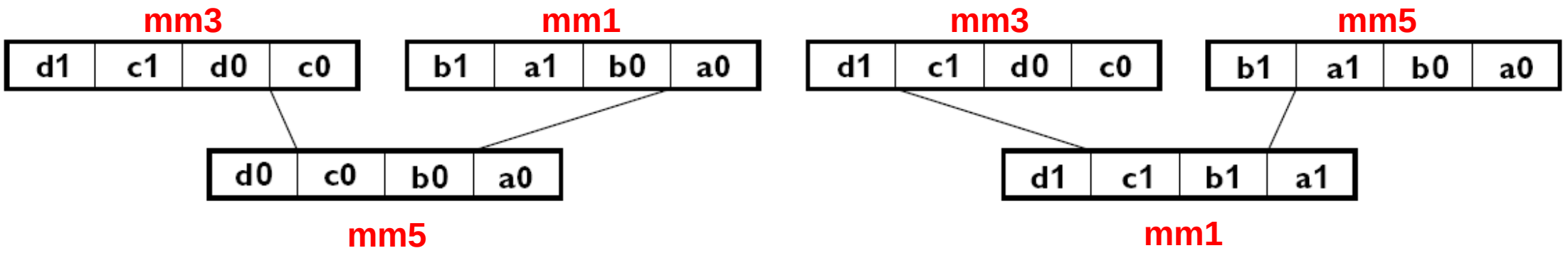
```
movq        mm1, row1      ; load pixels from first row of matrix
movq        mm2, row2      ; load pixels from second row of matrix
movq        mm3, row3      ; load pixels from third row of matrix
movq        mm4, row4      ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2       ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd   mm3, mm4       ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq        mm5, mm1       ; copy mm1 to mm5
punpckldq   mm1, mm3       ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq   mm5, mm3       ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```



Application: matrix transpose

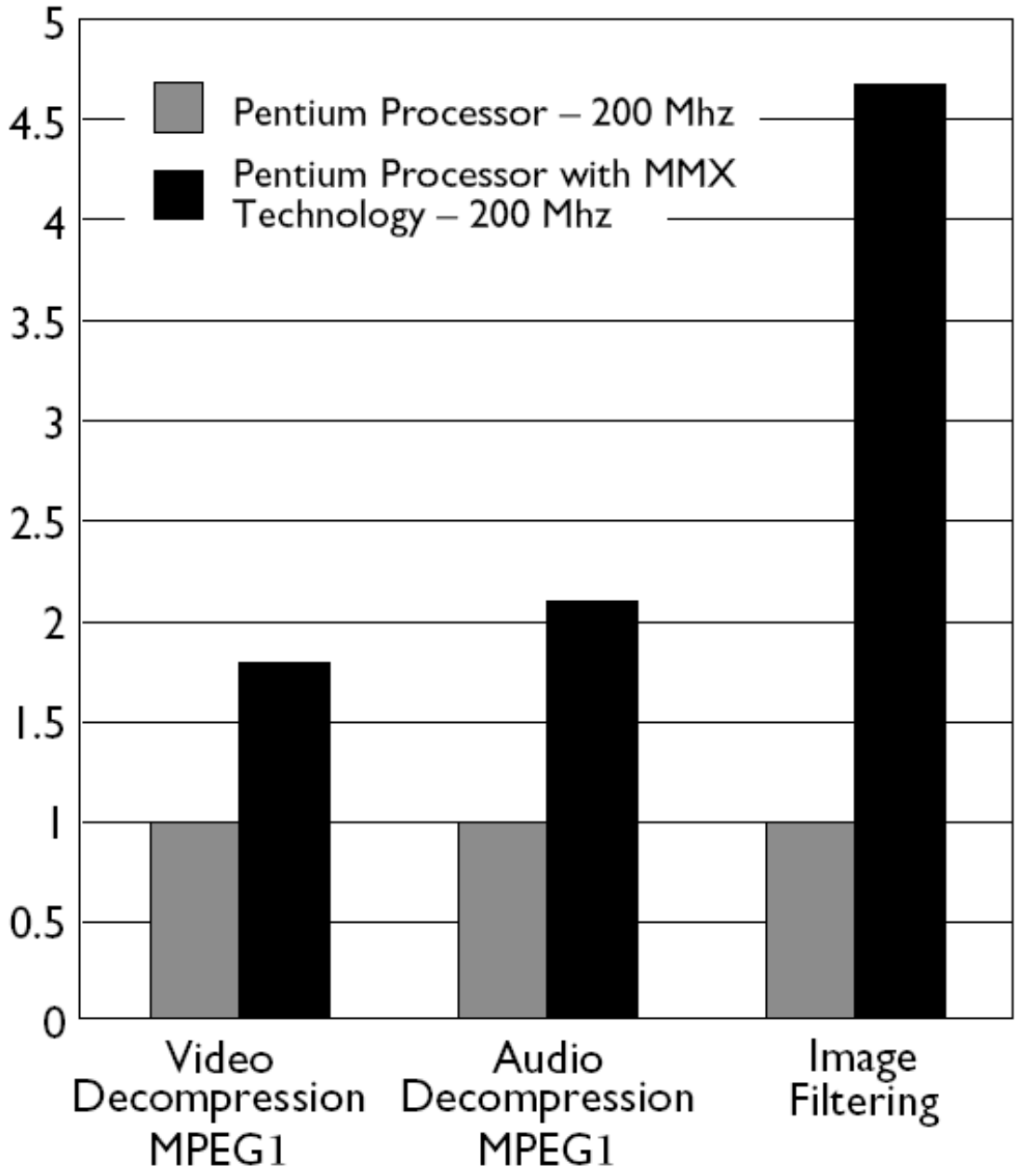
MMX code sequence operation:

```
movq        mm1, row1     ; load pixels from first row of matrix
movq        mm2, row2     ; load pixels from second row of matrix
movq        mm3, row3     ; load pixels from third row of matrix
movq        mm4, row4     ; load pixels from fourth row of matrix
punpcklwd   mm1, mm2      ; unpack low order words of rows 1 & 2, mm 1 = [b1, a1, b0, a0]
punpcklwd   mm3, mm4      ; unpack low order words of rows 3 & 4, mm3 = [d1, c1, d0, c0]
movq        mm5, mm1      ; copy mm1 to mm5
punpckldq   mm1, mm3      ; unpack low order doublewords -> mm2 = [d0, c0, b0, a0]
punpckhdq   mm5, mm3      ; unpack high order doublewords -> mm5 = [d1, c1, b1, a1]
```

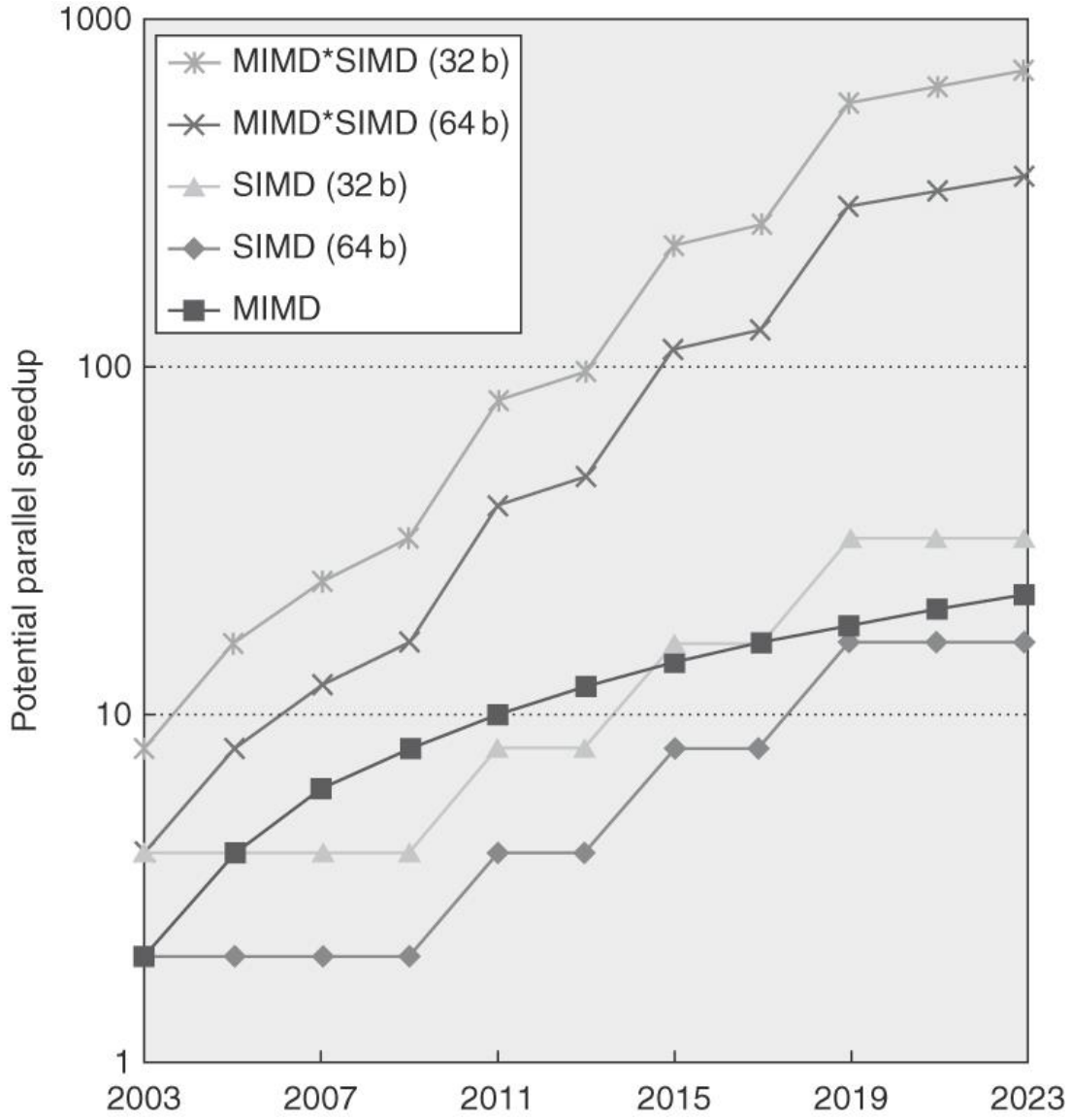


Performance Boost

- Benchmark kernels: FFT, FIR, vector dot-product, IDCT, motion compensation.
- >80% performance gain
- Lower the cost of multimedia programs by removing the need of specialized DSP chips



DLP important for conventional CPUs



- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - ***Note: Educated guess, not Intel product plans!***
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years

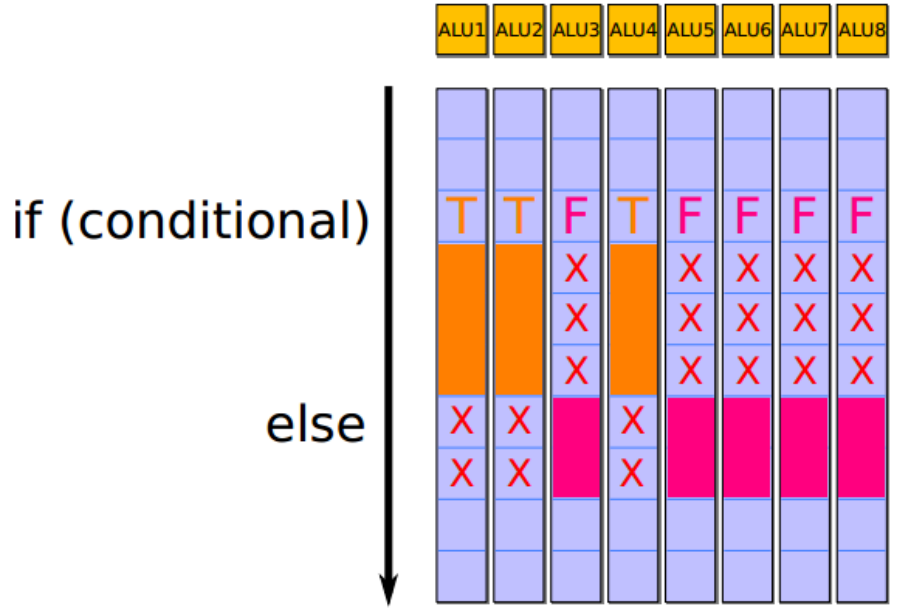
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

SIMD Drawbacks

- Restricted degree of parallelism: 64-bit ~ 512-bit
  - Continuous extension to ISA?
- Restricted conditions
  - Effective on all elements, no flexibility on conditional cases
- Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

SIMD ≠ DLP !



Packed SIMD versus Vectors

- Limited instruction set
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b) , adding scatter/gather
  - New ARM SVE/MVE vector ISA closer to traditional vector designs

Outline

- DLP
- SIMD
- GPU

Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

General-Purpose GPUs (GP-GPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU, which supported a new programming language CUDA (in 2007)
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

Early 90s – Pre GPU

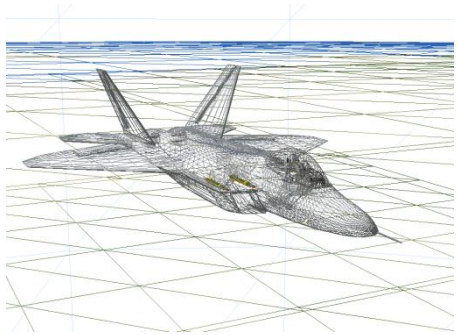
Wolfenstein 3D, 1992

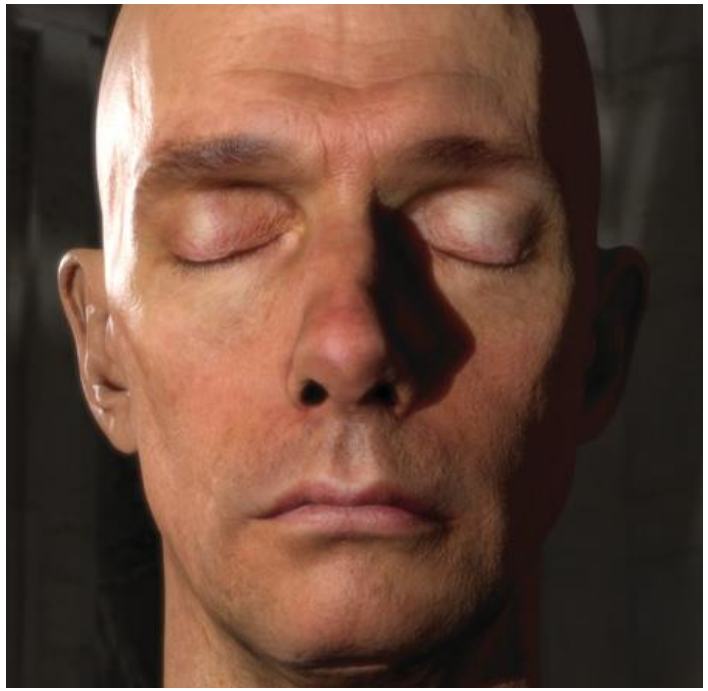
Doom I, 1993

- Interactive software rendering (no GPUs yet)
- NOTE: SGI was building interactive rendering supercomputers, but this was beginning of interactive 3D graphics on PC

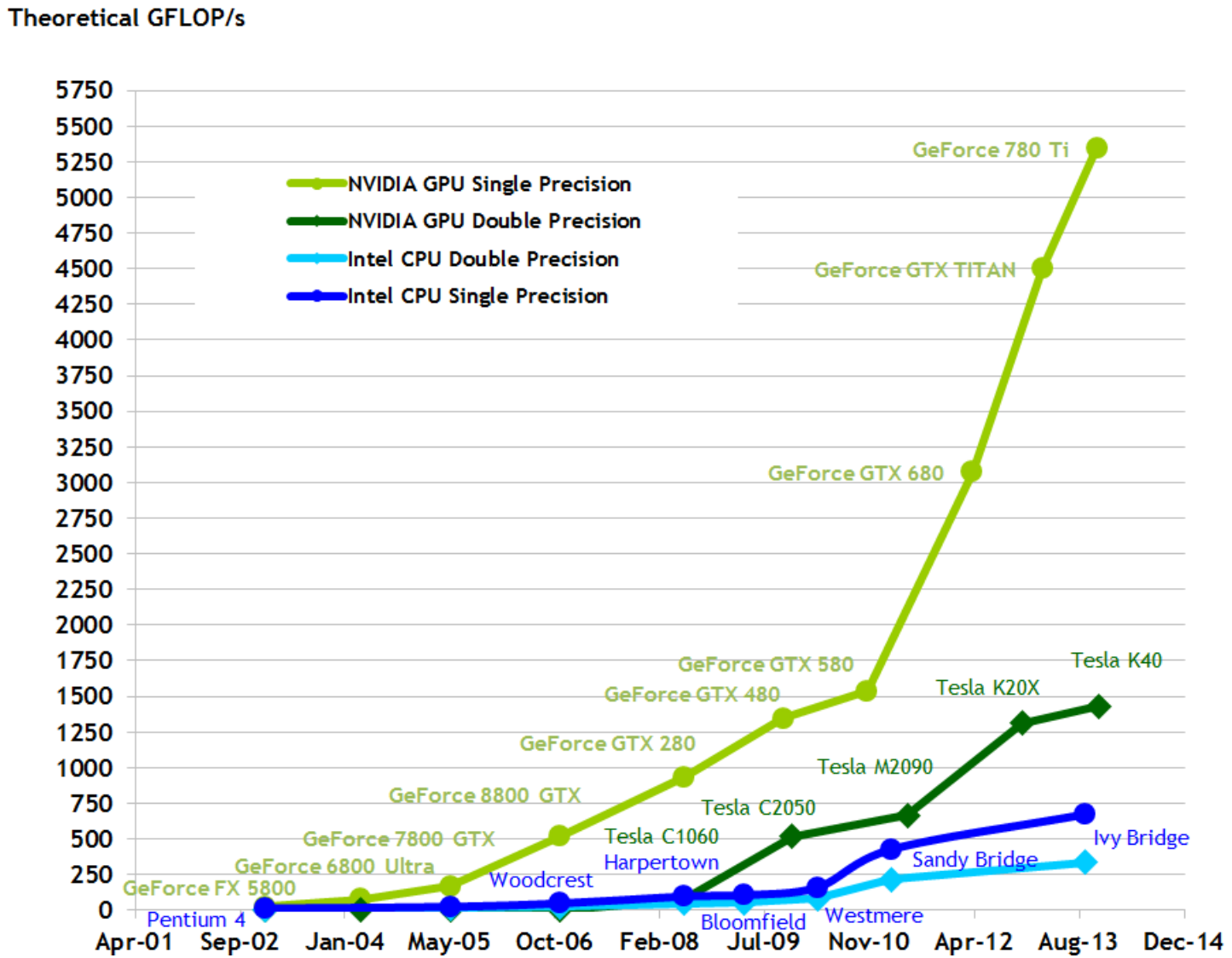
Graphics Workloads

- Triangles/vertices and pixels/fragments

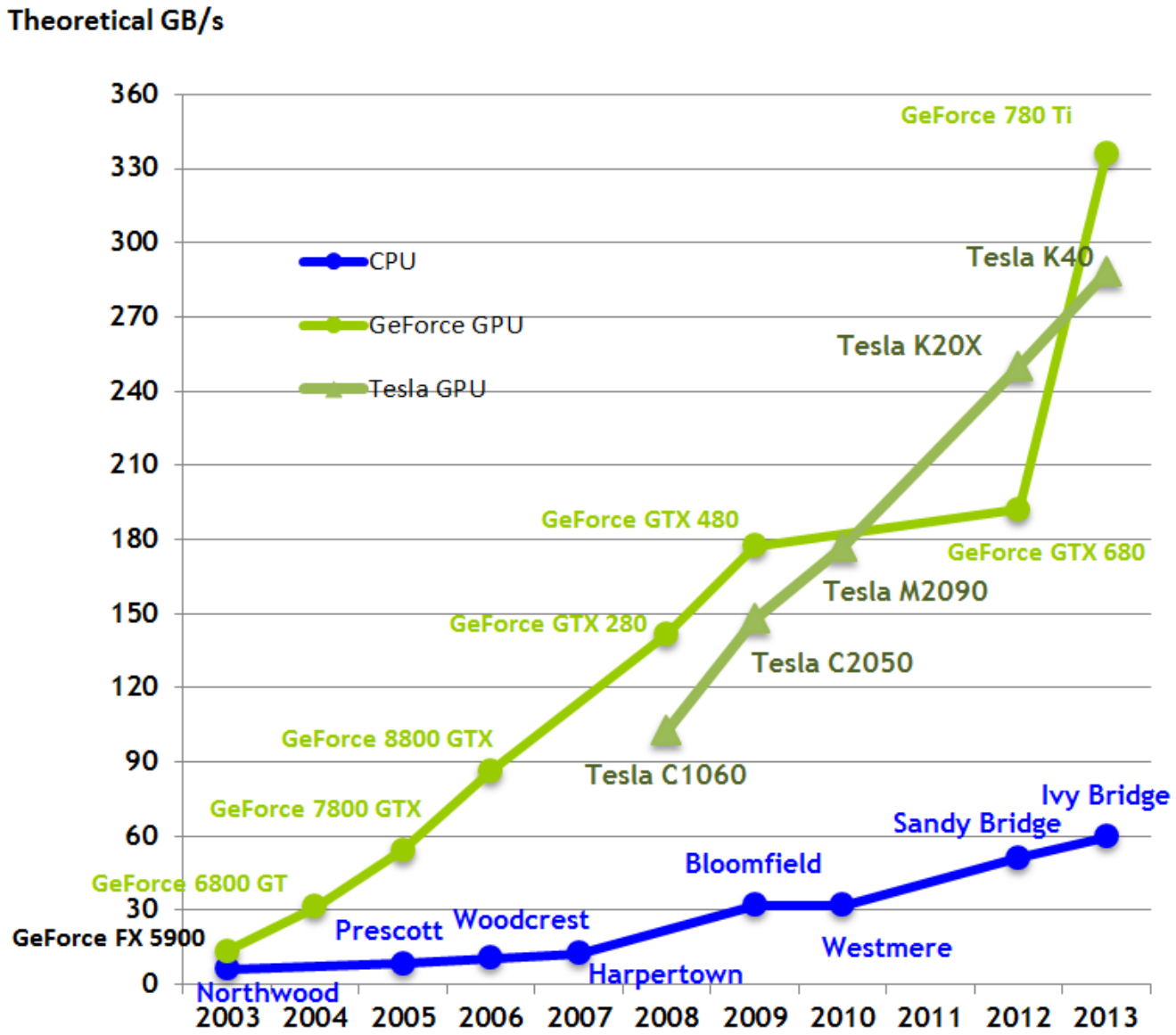




CPU and GPU Trends



CPU and GPU Trends

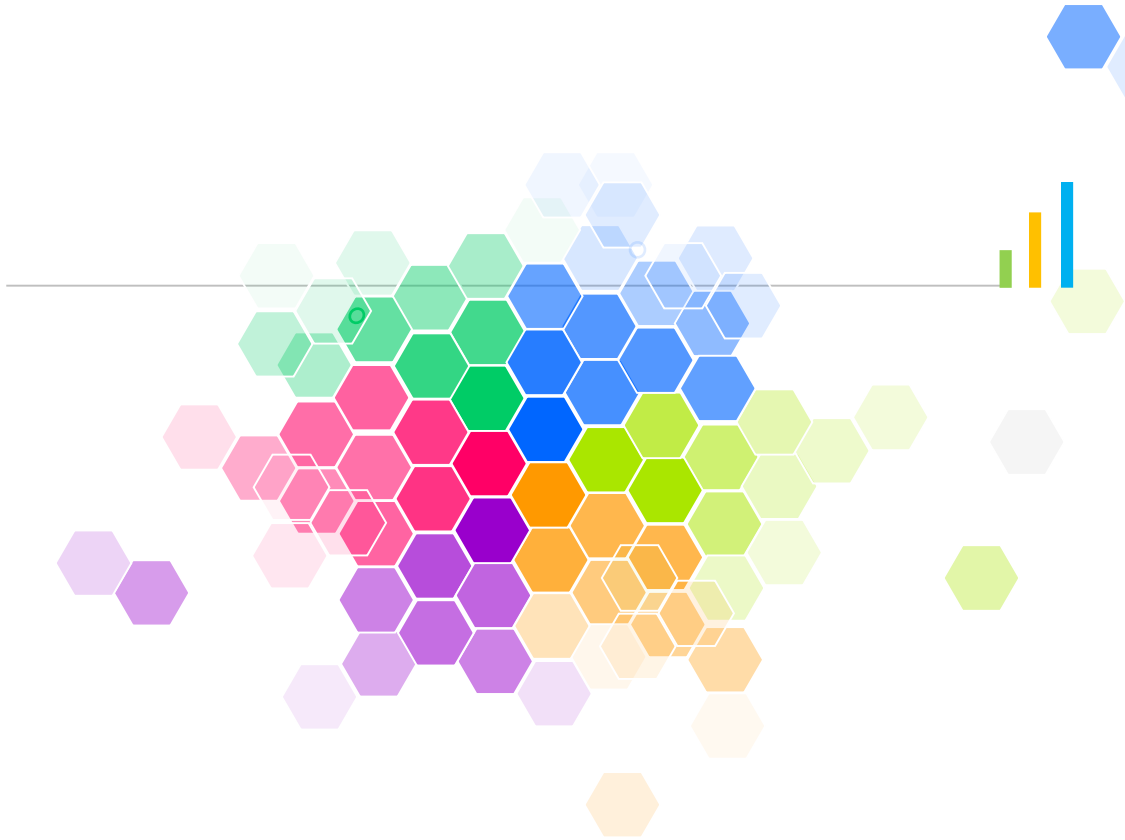


NIPS 2017 NVIDIA 发布最新GPU （Titan V)

- Volta 架构 GV100，拥有 12GB 显存
- 峰值浮点性能为 110 Tflops，是Titan Xp（12T）的 9 倍
- 在 fp32 下是 Titan Xp 的 1.5 倍，在 fp16 下是 Titan Xp 的 3 倍
- 售价 $2999（大概2w RMB）
- 采用台积电（TSMC）12nm FFN 制程
- 5120 个 CUDA 核心
- 晶体管数量： 211 亿
- 面积：815mm^2（约等于一块apple watch的面积）

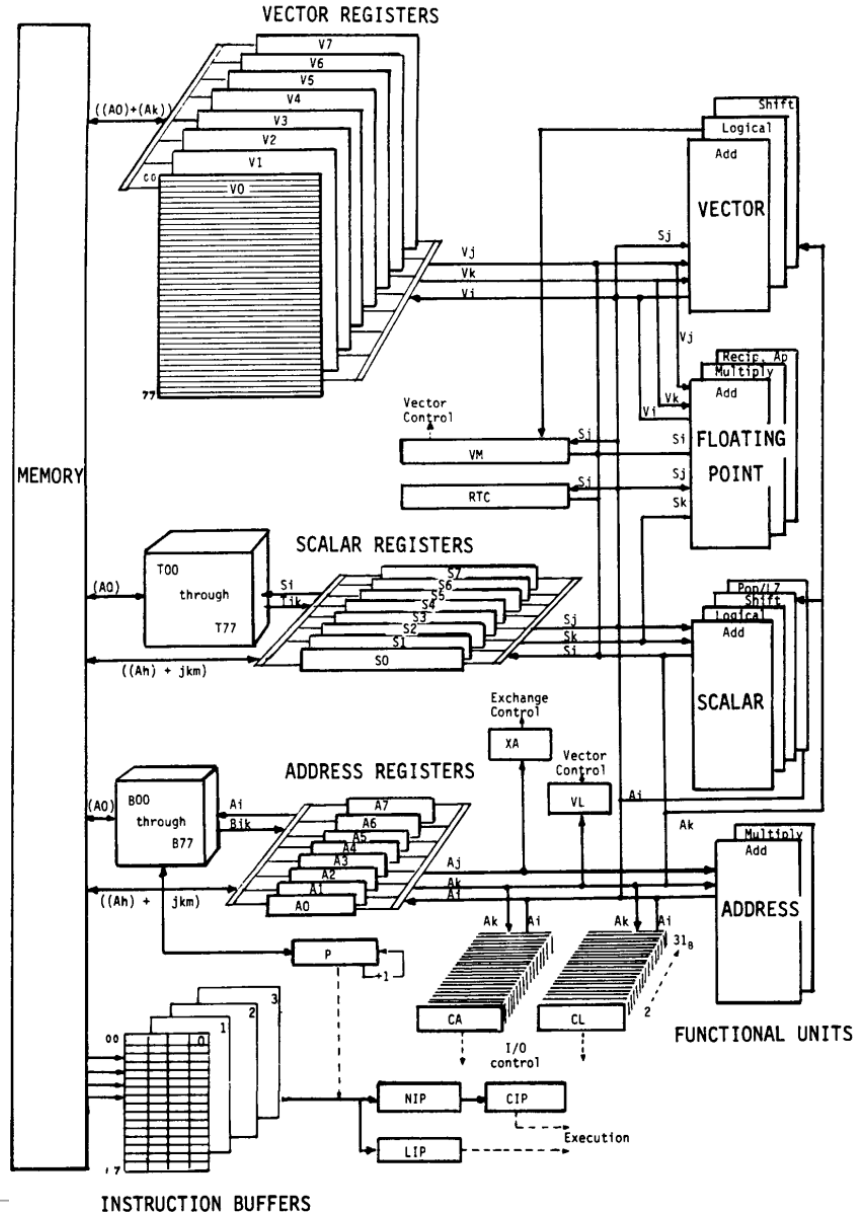
Important of Machine Learning for GPUs

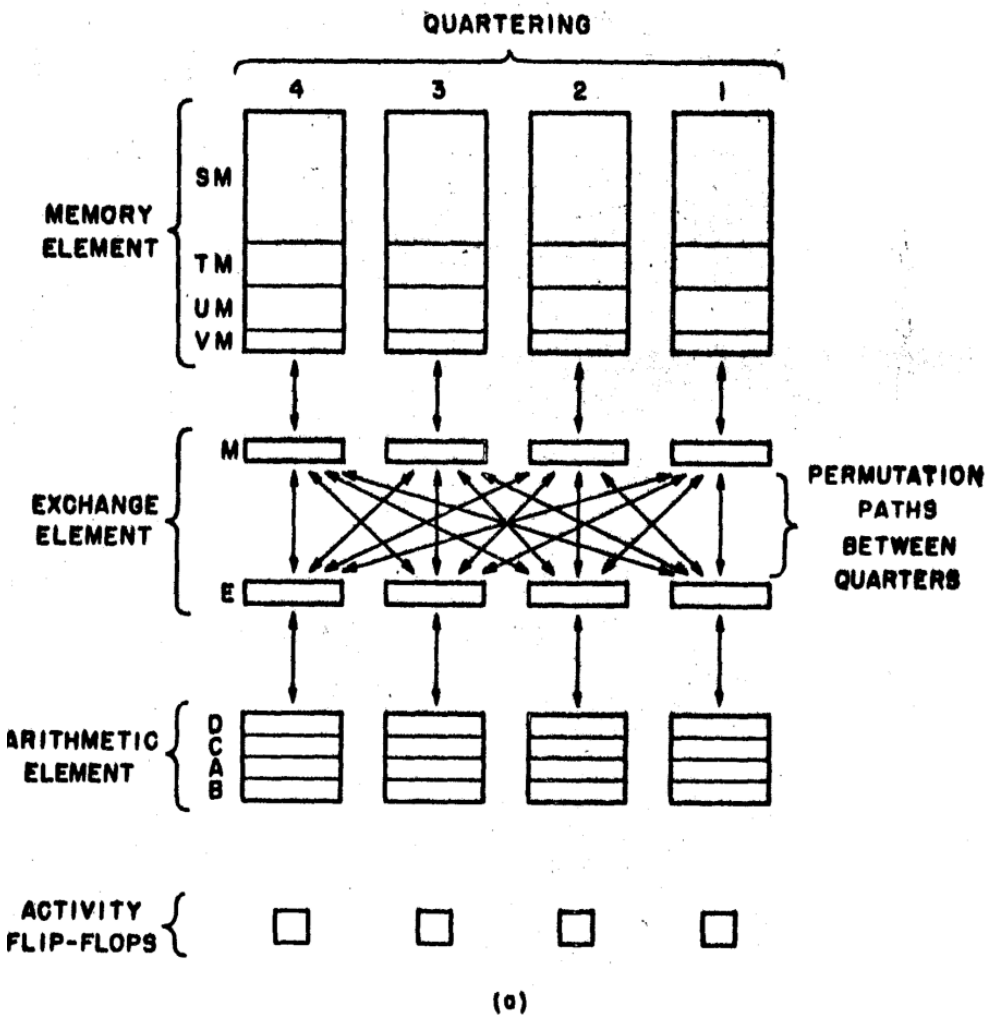
NVIDIA stock price 40x in 9 years (since deep learning became important)



欢迎提问

Vector vs. SIMD





Vector vs. GPU

- GPU
  - Discrete accelerator running in own memory space
    - More recently support sharing of physical, or pinned virtual, but still in different memory hierarchy and no page faults
  - Terrible at scalar code
  - Only effective on very large data-parallel tasks (>10,000s)
  - ISA/microarchitecture evolved from graphics shader needs, not general compute
  - Only seems efficient compared to out-of-order scalar core
- Traditional vector
  - Coprocessor running in same memory hierarchy as scalar
  - Tightly coupled to scalar core
  - Effective with loop counts of 2 or more
  - ISA evolved from general computing needs
  - Very efficient