

CS 110

Computer Architecture

Amdahl's Law, Data-level Parallelism

Instructor:
Sören Schwertfeger

<http://shtech.org/courses/ca/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

IEEE 754 Floating Point Standard (2/3)

- IEEE 754 uses “biased exponent” representation
 - Designers wanted FP numbers to be used even if no FP hardware; e.g., sort records with FP numbers using integer compares
 - Wanted bigger (integer) exponent field to represent bigger numbers
 - 2’s complement poses a problem (because negative numbers look bigger)
 - Use just magnitude and offset by half the range

IEEE 754 Floating Point Standard (3/3)

- Called **Biased Notation**, where bias is number subtracted to get final number
 - IEEE 754 uses bias of 127 for single prec.
 - Subtract 127 from Exponent field to get actual value for exponent

- **Summary (single precision):**



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023 (half, quad similar)

Representation for $\pm \infty$

- In FP, divide by 0 should produce $\pm \infty$, not overflow.
- Why?
 - OK to do further computations with ∞
E.g., $X/0 > Y$ may be a valid comparison
- IEEE 754 represents $\pm \infty$
 - Most positive exponent reserved for ∞
 - Significands all zeroes

Representation for 0

- Represent 0?

- exponent all zeroes

- significand all zeroes

- What about sign? Both cases valid

+0: 0 00000000 000000000000000000000000000000

-0: 1 00000000 000000000000000000000000000000

Special Numbers

- What have we defined so far?
(Single Precision)

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>???</u>
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	<u>nonzero</u>	<u>???</u>

- Clever idea:
 - Use $\text{exp}=0,255$ & $\text{Sig}\neq 0$

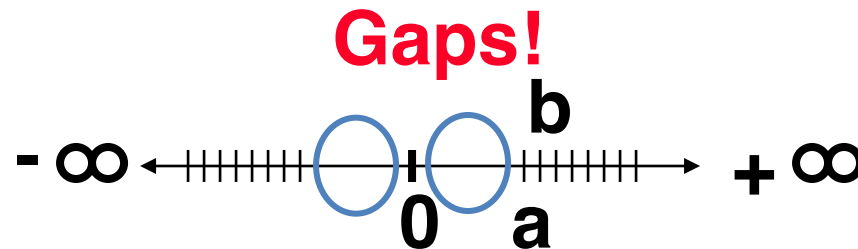
Representation for Not a Number

- What do I get if I calculate $\text{sqrt}(-4.0)$ or $0/0$?
 - If ∞ not an error, these shouldn't be either
 - Called Not a Number (NaN)
 - Exponent = 255, Significand nonzero
- Why is this useful?
 - Hope NaNs help with debugging?
 - They contaminate: $\text{op}(\text{NaN}, X) = \text{NaN}$
 - Can use the significand to identify which!

Representation for Denorms (1/2)

- Problem: There's a gap among representable FP numbers around 0
 - Smallest representable pos num:
 - $a = 1.0... 2 * 2^{-126} = 2^{-126}$
 - Second smallest representable pos num:
 - $b = 1.000.....1 2 * 2^{-126}$
 $= (1 + 0.00...12) * 2^{-126}$
 $= (1 + 2^{-23}) * 2^{-126}$
 $= 2^{-126} + 2^{-149}$
 - $a - 0 = 2^{-126}$
 - $b - a = 2^{-149}$

**Normalization
and implicit 1
is to blame!**



Representation for Denorms (2/2)

- **Solution:**

- We still haven't used Exponent = 0, Significand nonzero

- DENormalized number: no (implied) leading 1, **implicit exponent = -126.**

- Smallest representable pos num:

$$a = 2^{-149}$$

- Second smallest representable pos num:

$$b = 2^{-148}$$



Special Numbers Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	<u>Denorm</u>
1-254	anything	+/- fl. pt. #
255	<u>0</u>	<u>+/- ∞</u>
255	<u>nonzero</u>	<u>NaN</u>

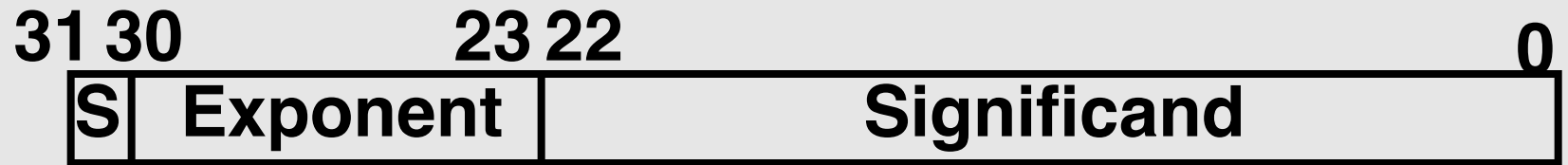
Conclusion

Exponent tells Significant how much (2^i) to count by (... , 1/4, 1/2, 1, 2, ...)

Can store NaN, $\pm \infty$

- Floating Point lets us:
 - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
 - Store **approximate** values for very large and very small #s.
- IEEE 754 Floating-Point Standard** is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

• Summary (single precision):



1 bit 8 bits 23 bits

• $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

• **Double precision identical, except with exponent bias of 1023 (half, quad similar)**

Admin

- Midterm II
 - May 17 (next Thursday!)
 - Topics: SDS, MIPS Pipelining, Caches, Floating Point, Parallelism (Till today's lecture, including today's lecture)
 - Same rules as Midterm I except:
 - 2 cheat A4 sheets are allowed (both handwritten in English by you)
 - MIPS green sheet will be provided again

New-School Machine Structures (It's a bit more complicated!)

Software

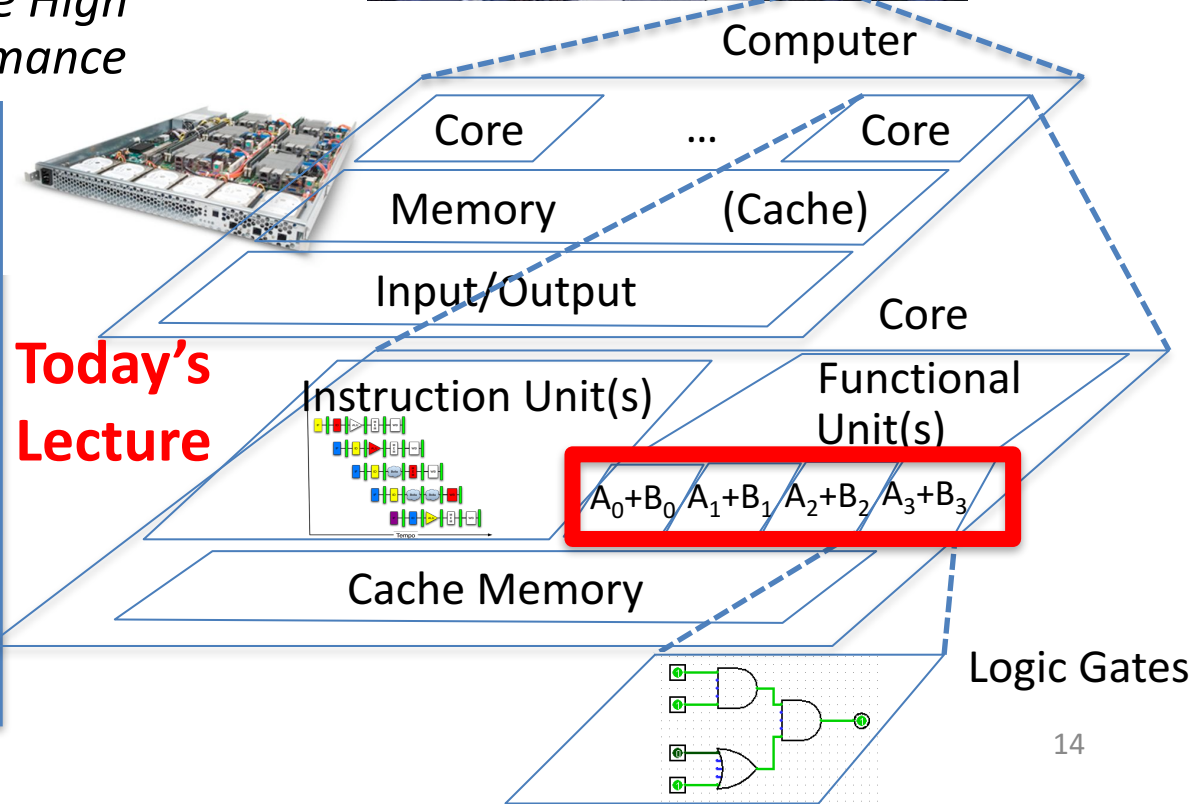
Hardware

Warehouse
Scale
Computer

Smart
Phone



*Harness
Parallelism &
Achieve High
Performance*



- Parallel Requests

Assigned to computer
e.g., Search "Katz"

- Parallel Threads

Assigned to core
e.g., Lookup, Ads

- Parallel Instructions

>1 instruction @ one time
e.g., 5 pipelined instructions

- Parallel Data

>1 data item @ one time
e.g., Add of 4 pairs of words

- Hardware descriptions

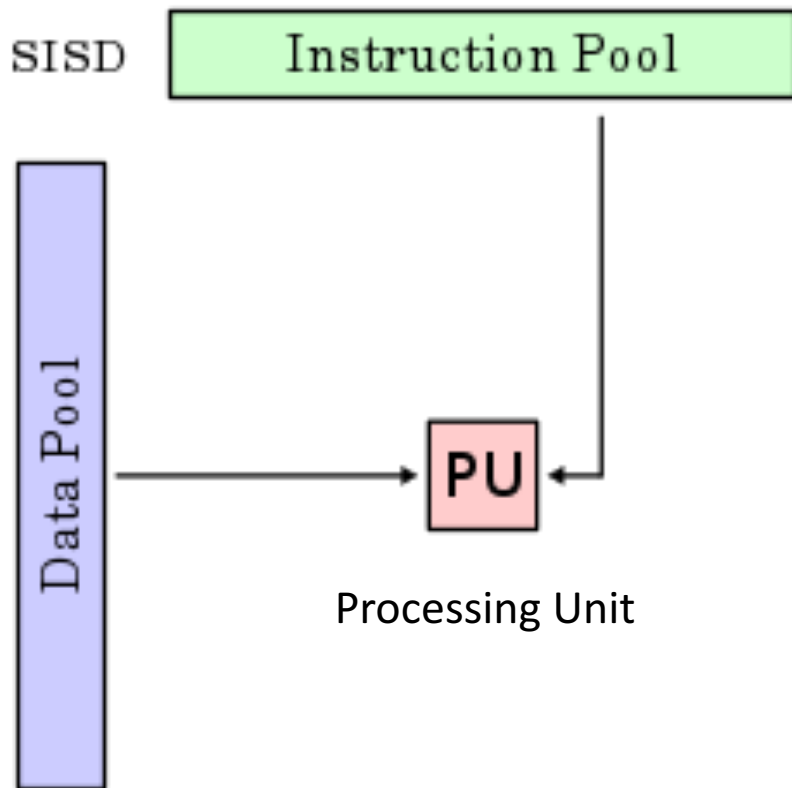
All gates @ one time

- Programming Languages

Using Parallelism for Performance

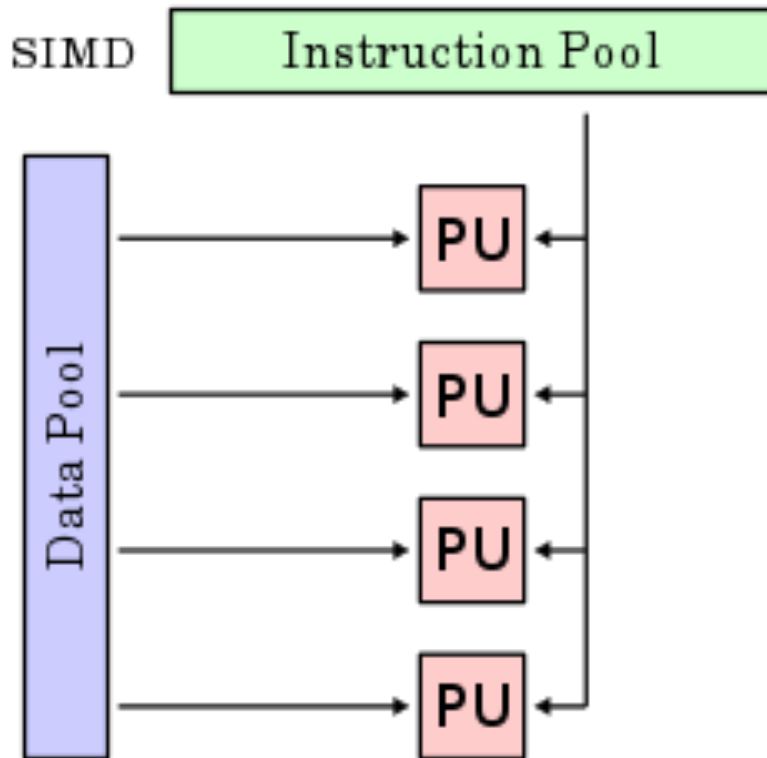
- Two basic ways:
 - Multiprogramming
 - run multiple independent programs in parallel
 - “Easy”
 - Parallel computing
 - run one program faster
 - “Hard”
- We’ll focus on parallel computing for next few lectures

Single-Instruction/Single-Data Stream (SISD)



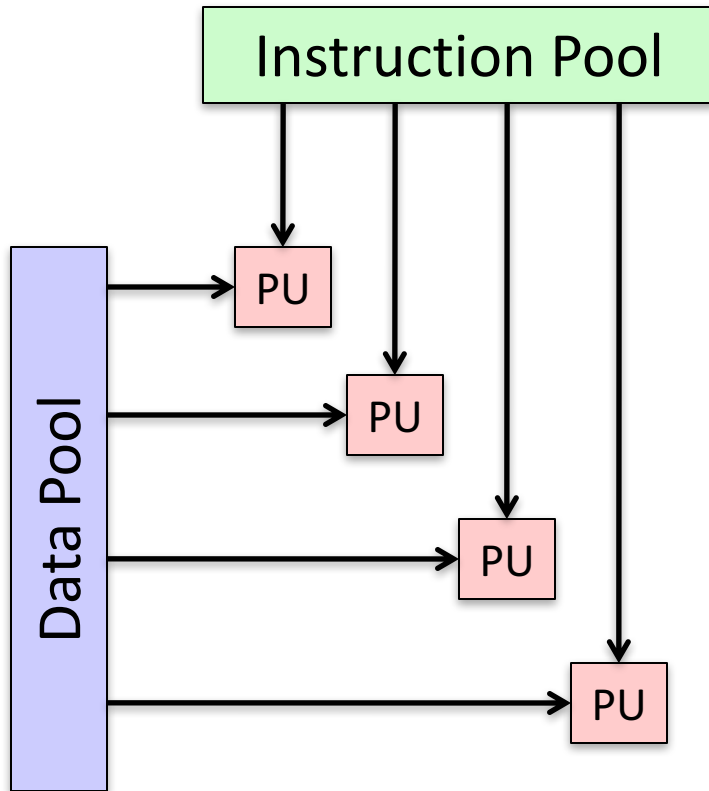
- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines

Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)



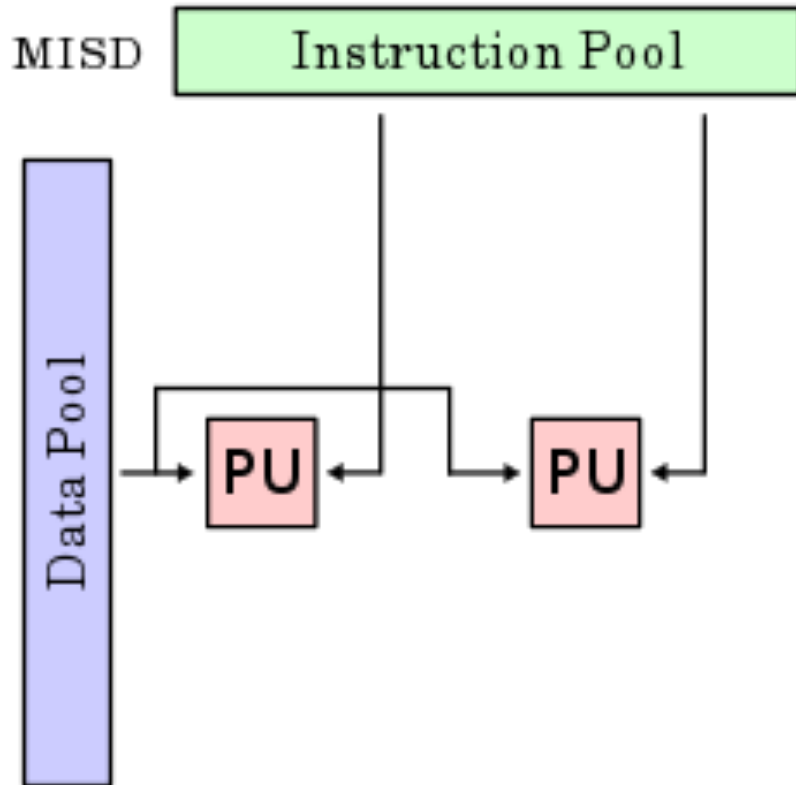
- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

Multiple-Instruction/Multiple-Data Streams (MIMD or “mim-dee”)



- Multiple autonomous processors simultaneously executing different instructions on different data.
 - MIMD architectures include multicore and Warehouse-Scale Computers

Multiple-Instruction/Single-Data Stream (MISD)



- Multiple-Instruction, Single-Data stream computer that exploits multiple instruction streams against a single data stream.
 - Rare, mainly of historical interest only

Flynn* Taxonomy, 1966

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 (Clovertown)

- Since about 2013, SIMD and MIMD most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data (“SPMD”)
 - Single program that runs on all processors of a MIMD
 - Cross-processor execution coordination using synchronization primitives
- SIMD (aka hw-level *data parallelism*): specialized function units, for handling lock-step calculations involving arrays
 - Scientific computing, signal processing, multimedia (audio/video processing)

*Prof. Michael Flynn, Stanford

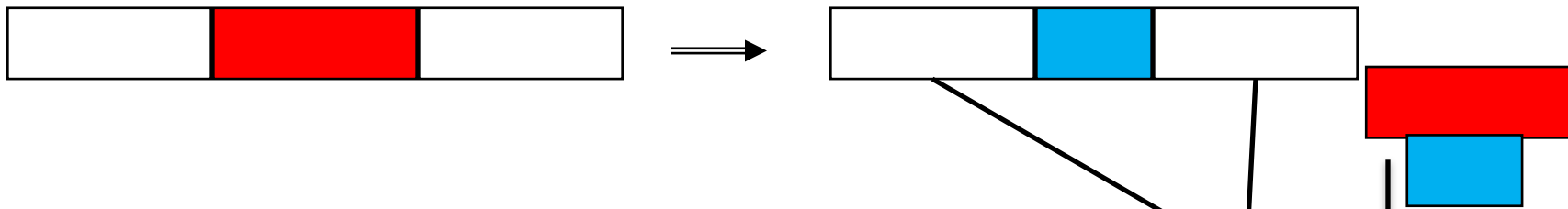


Big Idea: Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F < 1) of the task by a factor S (S > 1) and the remainder of the task is unaffected



$$\text{Execution Time w/ E} = \text{Execution Time w/o E} \times [(1-F) + F/S]$$

$$\text{Speedup w/ E} = 1 / [(1-F) + F/S]$$

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}$$

Non-speed-up part \rightarrow (1 - F) \leftarrow Speed-up part \leftarrow $\frac{F}{S}$

Example: the execution time of half of the program can be accelerated by a factor of 2.

What is the program speed-up overall?

$$\frac{1}{\frac{0.5 + 0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Example #1: Amdahl's Law

$$\text{Speedup } w/ E = 1 / [(1-F) + F/S]$$

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time

$$\text{Speedup } w/ E = 1 / (.75 + .25/20) = 1.31$$

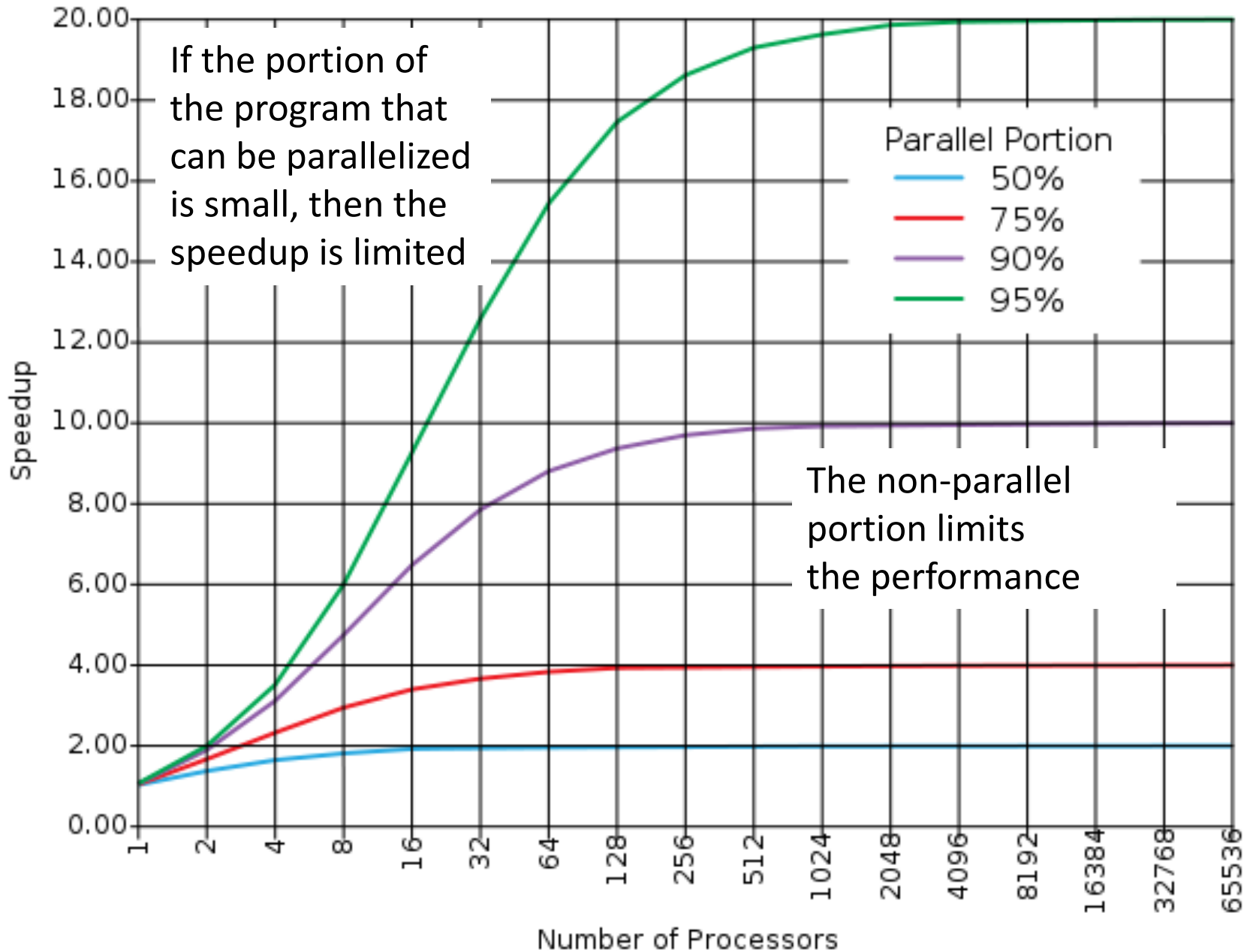
- What if its usable only 15% of the time?

$$\text{Speedup } w/ E = 1 / (.85 + .15/20) = 1.17$$

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

$$\text{Speedup } w/ E = 1 / (.001 + .999/100) = 90.99$$

Amdahl's Law



Strong and Weak Scaling

- To get good speedup on a parallel processor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
 - *Strong scaling*: when speedup can be achieved on a parallel processor without increasing the size of the problem
 - *Weak scaling*: when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
- **Load balancing** is another important factor: every processor doing same amount of work
 - Just one unit with twice the load of others cuts speedup almost in half

Question

Suppose a program spends 80% of its time in a square root routine. How much must you speedup square root to make the program run 5 times faster?

$$\text{Speedup } w/ E = 1 / [(1-F) + F/S]$$

A: 5

B: 16

C: 20

D: 100

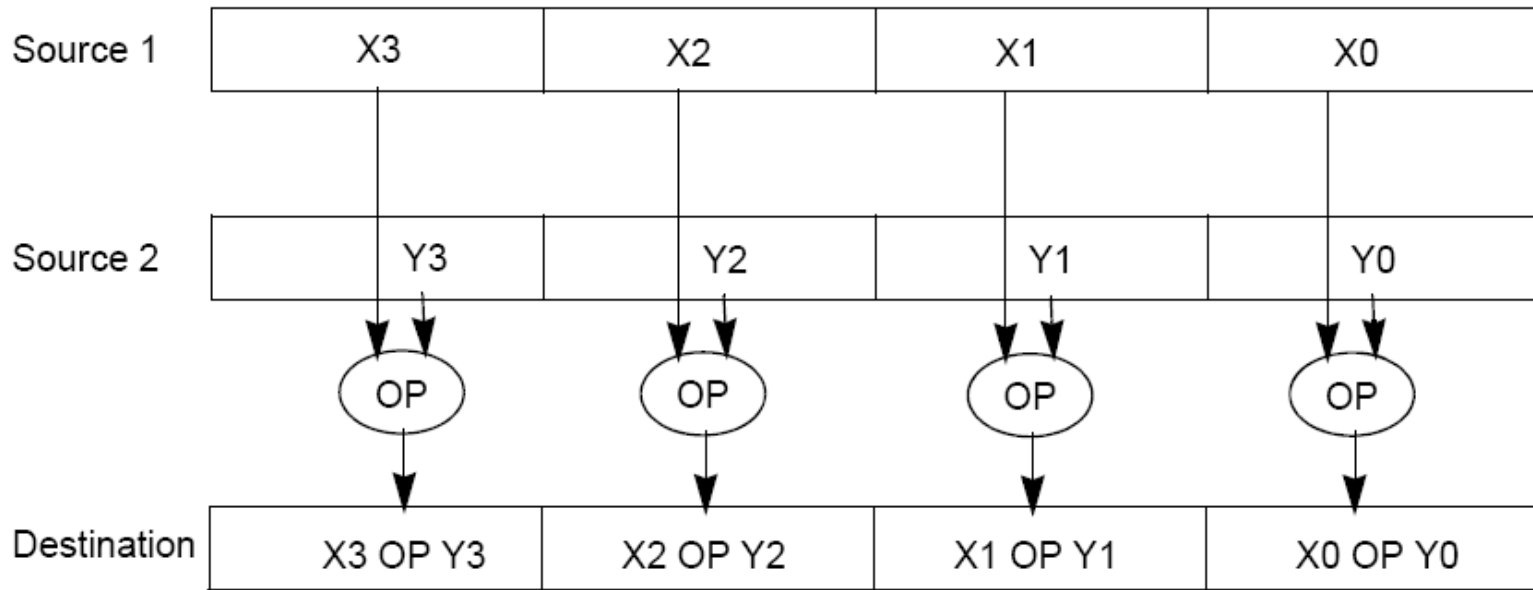
E: None of the above

SIMD Architectures

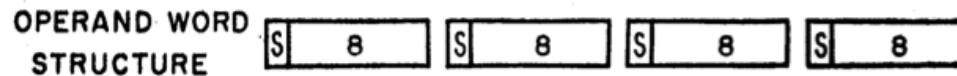
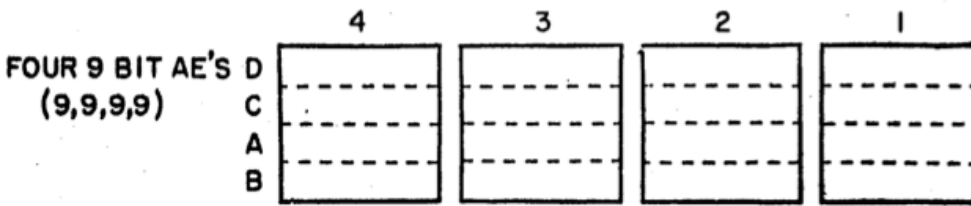
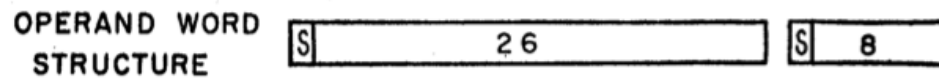
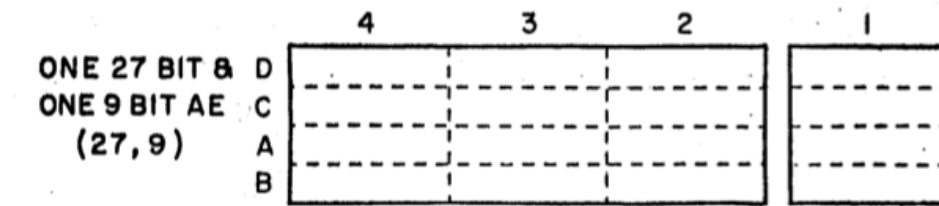
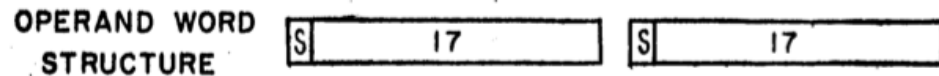
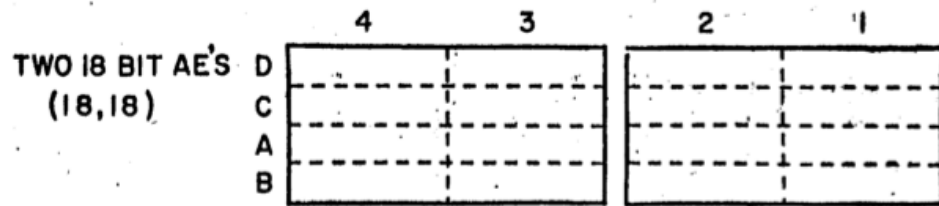
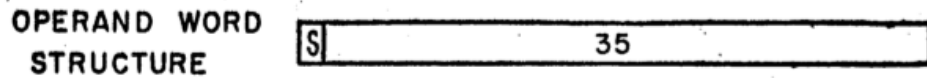
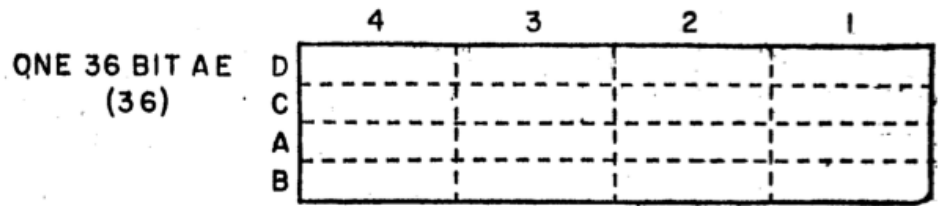
- *Data parallelism*: executing same operation on multiple data streams
- Example to provide context:
 - Multiplying a coefficient vector by a data vector (e.g., in filtering)
$$y[i] := c[i] \times x[i], \quad 0 \leq i < n$$
- Sources of performance improvement:
 - One instruction is fetched & decoded for entire operation
 - Multiplications are known to be independent
 - Pipelining/concurrency in memory access as well

Intel “Advanced Digital Media Boost”

- To improve performance, Intel’s SIMD instructions
 - Fetch one instruction, do the work of multiple instructions



First SIMD Extensions: MIT Lincoln Labs TX-2, 1957



Intel SIMD Extensions

- MMX 64-bit registers, reusing floating-point registers [1992]
- SSE2/3/4, new 128-bit registers [1999]
- AVX, new 256-bit registers [2011]
- AVX-512: 512-bit registers [2016]
 - Space for expansion to 1024-bit registers

XMM Registers

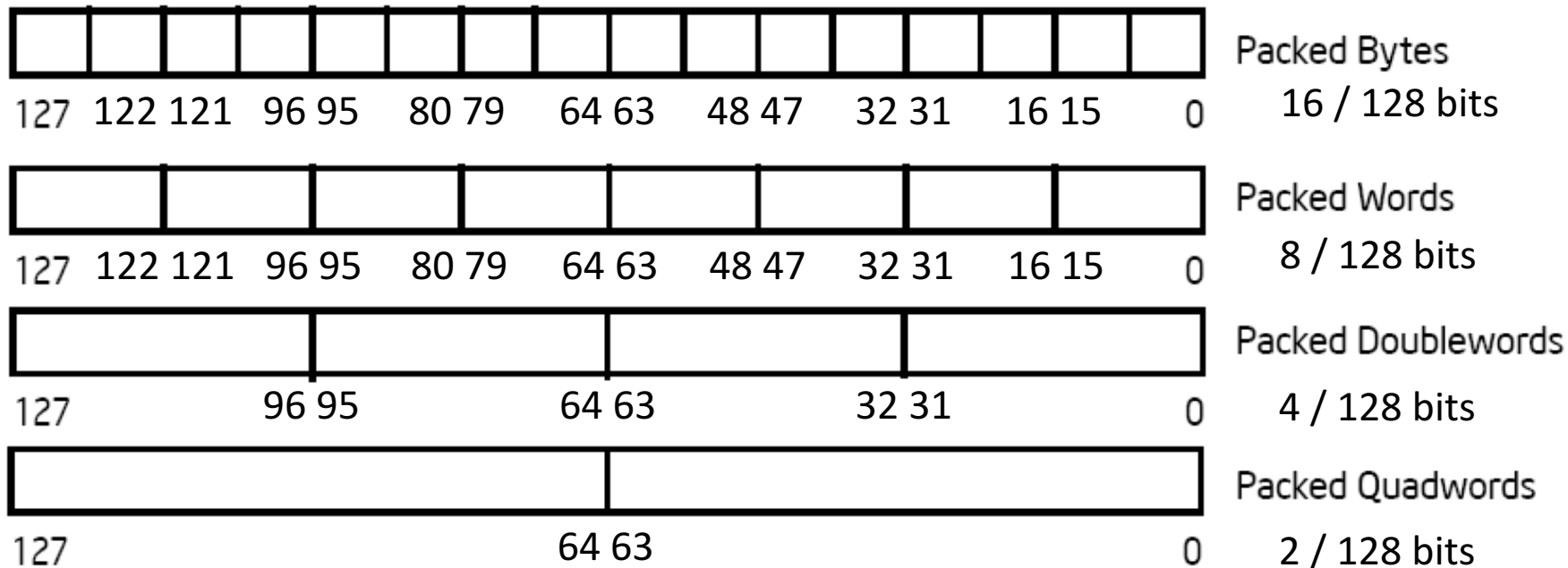
127		0
	XMM7	
	XMM6	
	XMM5	
	XMM4	
	XMM3	
	XMM2	
	XMM1	
	XMM0	

- Architecture extended with eight 128-bit data registers: XMM registers
 - x86 64-bit address architecture adds 8 additional registers (XMM8 – XMM15)

Intel Architecture SSE2+ 128-Bit SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
 - Single-precision FP: Double word (32 bits)
 - Double-precision FP: Quad word (64 bits)

Fundamental 128-Bit Packed SIMD Data Types



SSE/SSE2 Floating Point Instructions

Move
does
both
load
and
store

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

xmm: one operand is a 128-bit SSE2 register

mem/xmm: other operand is in memory or an SSE2 register

{SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: four 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register

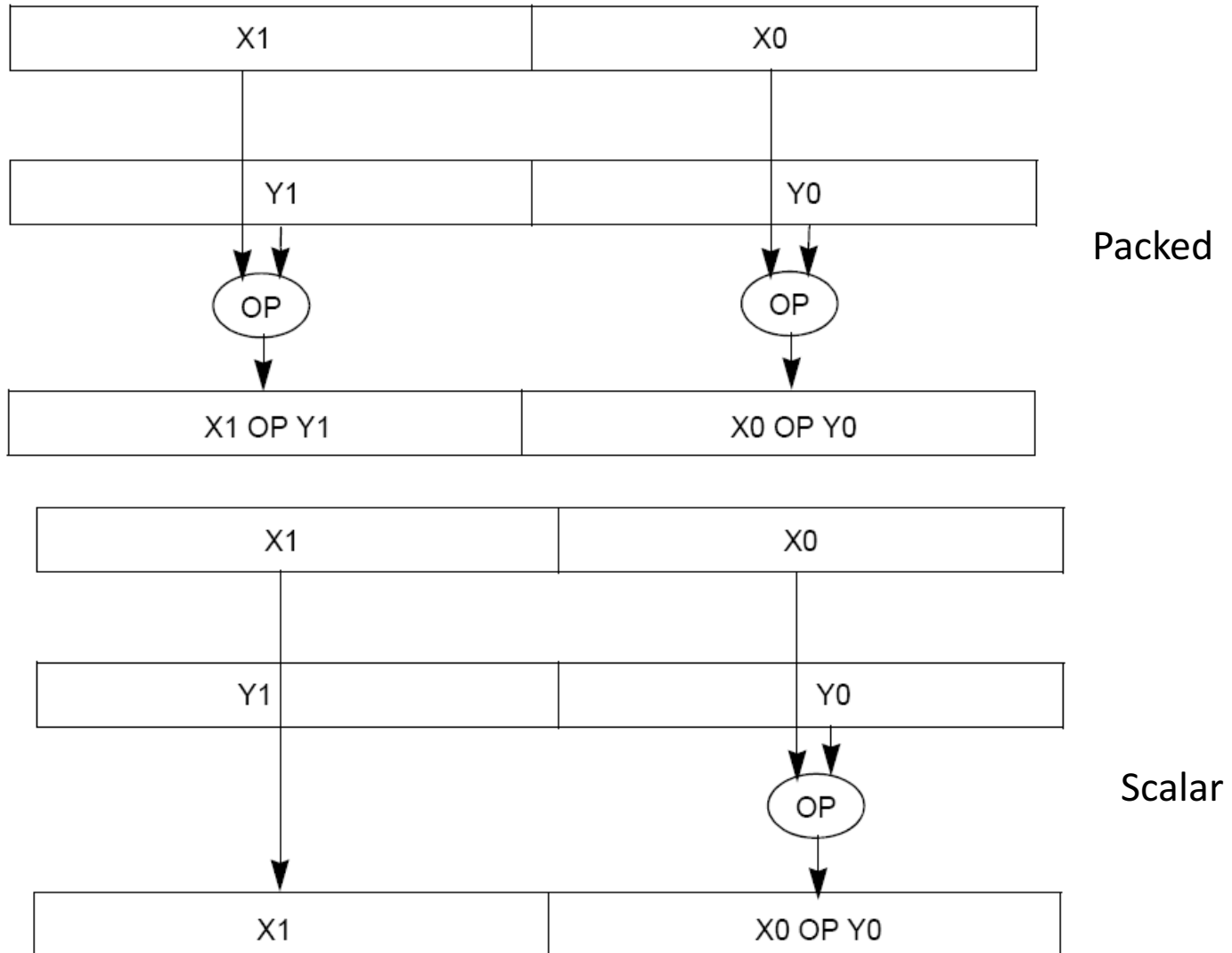
{A} 128-bit operand is aligned in memory

{U} means the 128-bit operand is unaligned in memory

{H} means move the high half of the 128-bit operand

{L} means move the low half of the 128-bit operand

Packed and Scalar Double-Precision Floating-Point Operations



Example: SIMD Array Processing

```
for each f in array  
  f = sqrt(f)
```

```
for each f in array  
{  
  load f to the floating-point register  
  calculate the square root  
  write the result from the register to memory  
}
```

```
for each 4 members in array  
{  
  load 4 members to the SSE register  
  calculate 4 square roots in one operation  
  store the 4 results from the register to memory  
}
```

SIMD style

Data-Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops
for(i=1000; i>0; i=i-1)
x[i] = x[i] + s;
- How can reveal more data-level parallelism than available in a single iteration of a loop?
- *Unroll loop* and adjust iteration rate

Looping in MIPS

Assumptions:

- \$t1 is initially the address of the element in the array with the highest address
- \$f0 contains the scalar value s
- 8(\$t2) is the address of the last element to operate on

CODE:

```
Loop: 1. l.d      $f2,0($t1)      ; $f2=array element
      2. add.d   $f10,$f2,$f0     ; add s to $f2
      3. s.d     $f10,0($t1)     ; store result
      4. addiu  $t1,$t1,#-8      ; decrement pointer 8 byte
      5. bne    $t1,$t2,Loop     ; repeat loop if $t1 != $t2
```

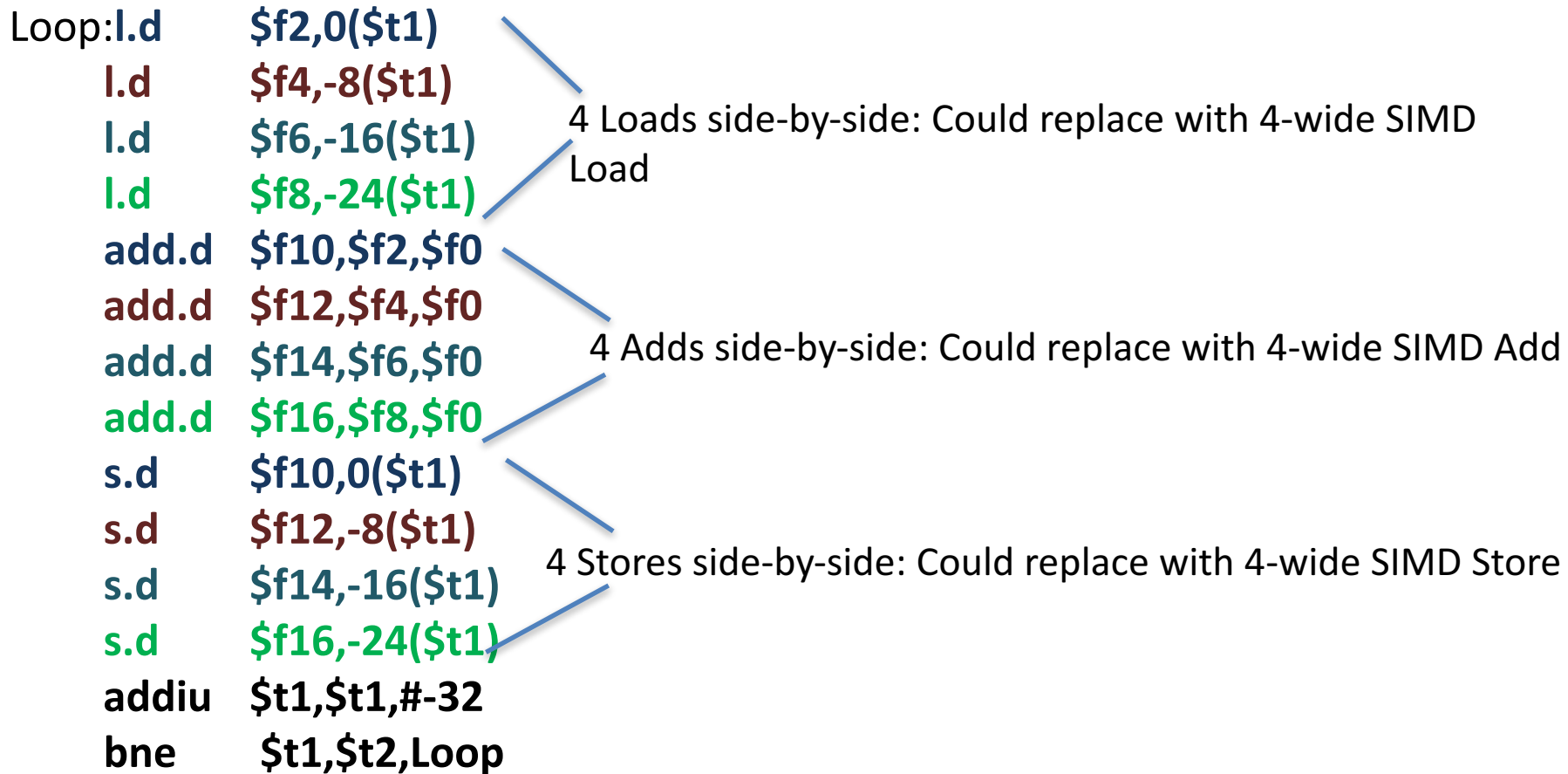
Loop Unrolled

```
Loop: l.d    $f2,0($t1)
      add.d  $f10,$f2,$f0
      s.d    $f10,0($t1)
      l.d    $f4,-8($t1)
      add.d  $f12,$f4,$f0
      s.d    $f12,-8($t1)
      l.d    $f6,-16($t1)
      add.d  $f14,$f6,$f0
      s.d    $f14,-16($t1)
      l.d    $f8,-24($t1)
      add.d  $f16,$f8,$f0
      s.d    $f16,-24($t1)
      addiu  $t1,$t1,#-32
      bne   $t1,$t2,Loop
```

NOTE:

1. Only 1 Loop Overhead every 4 iterations
2. This unrolling works if
 $\text{loop_limit}(\text{mod } 4) = 0$
3. Using different registers for each iteration eliminates data hazards in pipeline

Loop Unrolled Scheduled



Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C

```
for(i=1000; i>0; i=i-1)  
    x[i] = x[i] + s;
```

- Could be rewritten What is downside of doing it in C?

```
for(i=1000; i>0; i=i-4) {  
    x[i] = x[i] + s;  
    x[i-1] = x[i-1] + s;  
    x[i-2] = x[i-2] + s;  
    x[i-3] = x[i-3] + s;  
}
```


Generalizing Loop Unrolling

- A loop of **n iterations**
- **k copies** of the body of the loop
- **Assuming $(n \bmod k) \neq 0$**

Then we will run the loop with 1 copy of the body **$(n \bmod k)$** times and with k copies of the body **$\text{floor}(n/k)$** times

Example: Add Two Single-Precision Floating-Point Vectors

Computation to be performed:

```
vec_res.x = v1.x + v2.x;  
vec_res.y = v1.y + v2.y;  
vec_res.z = v1.z + v2.z;  
vec_res.w = v1.w + v2.w;
```

mov a ps : **move** from mem to XMM register,
memory **aligned**, **packed** single precision

add ps : **add** from mem to XMM register,
packed single precision

mov a ps : **move** from XMM register to mem,
memory **aligned**, **packed** single precision

SSE Instruction Sequence:

(Note: Destination on the right in x86 assembly)

```
movaps address-of-v1, %xmm0  
    // v1.w | v1.z | v1.y | v1.x -> xmm0  
addps address-of-v2, %xmm0  
    // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x -> xmm0  
movaps %xmm0, address-of-vec_res
```

Intel SSE Intrinsics

- Intrinsics are C functions and procedures for inserting assembly language into C code, including SSE instructions
 - With intrinsics, can program using these instructions indirectly
 - One-to-one correspondence between SSE instructions and intrinsics

Example SSE Intrinsics

Intrinsics:

Corresponding SSE instructions:

- Vector data type:

`_m128d`

- Load and store operations:

`_mm_load_pd`

MOVAPD/aligned, packed double

`_mm_store_pd`

MOVAPD/aligned, packed double

`_mm_loadu_pd`

MOVUPD/unaligned, packed double

`_mm_storeu_pd`

MOVUPD/unaligned, packed double

- Load and broadcast across vector

`_mm_load1_pd`

MOVSD + shuffling/duplicating

- Arithmetic:

`_mm_add_pd`

ADDPD/add, packed double

`_mm_mul_pd`

MULPD/multiple, packed double

Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

$$C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^2 A_{i,k} \times B_{k,j}$$

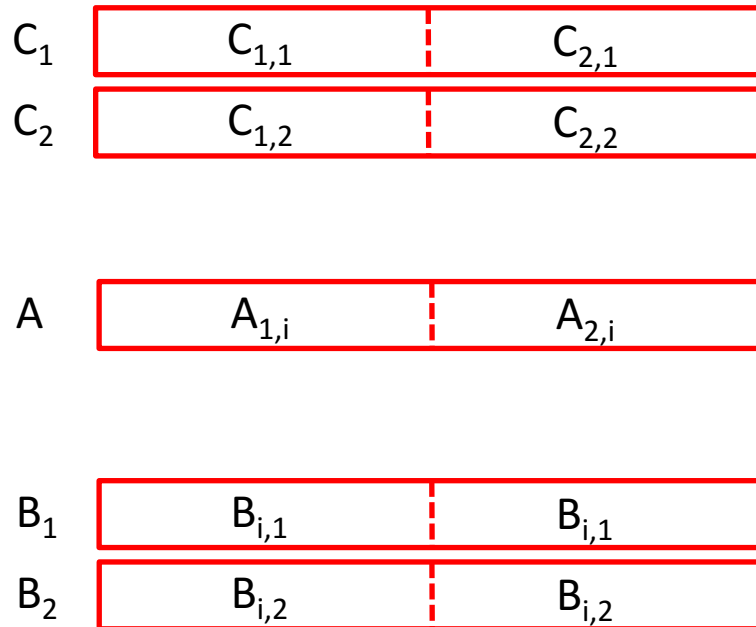
$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$\begin{aligned}
 C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
 C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \\
 C_{1,1} &= 1*1 + 0*2 = 1 & C_{1,2} &= 1*3 + 0*4 = 3 \\
 C_{2,1} &= 0*1 + 1*2 = 2 & C_{2,2} &= 0*3 + 1*4 = 4
 \end{aligned}$$

Example: 2 x 2 Matrix Multiply

- Using the XMM registers
 - 64-bit/double precision/two doubles per XMM reg



Stored in memory in Column order

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

C_1 C_2

Example: 2 x 2 Matrix Multiply

- Initialization

C_1	0	0
C_2	0	0

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$

- Initialization

C_1	0	0
C_2	0	0

- $l = 1$

A	$A_{1,1}$	$A_{2,1}$
---	-----------	-----------

`_mm_load_pd`: Load 2 doubles into XMM reg, Stored in memory in Column order

B_1	$B_{1,1}$	$B_{1,1}$
-------	-----------	-----------

B_2	$B_{1,2}$	$B_{1,2}$
-------	-----------	-----------

`_mm_load1_pd`: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$

- First iteration intermediate result

$$\begin{array}{l} C_1 \\ C_2 \end{array} \begin{array}{|c|c|} \hline 0 + A_{1,1}B_{1,1} & 0 + A_{2,1}B_{1,1} \\ \hline 0 + A_{1,1}B_{1,2} & 0 + A_{2,1}B_{1,2} \\ \hline \end{array}$$

```
c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));  
c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
```

SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- $i = 1$

$$A \begin{array}{|c|c|} \hline A_{1,1} & A_{2,1} \\ \hline \end{array}$$

`_mm_load_pd`: Stored in memory in Column order

$$\begin{array}{l} B_1 \\ B_2 \end{array} \begin{array}{|c|c|} \hline B_{1,1} & B_{1,1} \\ \hline B_{1,2} & B_{1,2} \\ \hline \end{array}$$

`_mm_load1_pd`: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$ $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
 $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$ $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

- First iteration intermediate result

$$\begin{array}{l}
 C_1 \quad \boxed{0 + A_{1,1}B_{1,1} \quad | \quad 0 + A_{2,1}B_{1,1}} \\
 C_2 \quad \boxed{0 + A_{1,1}B_{1,2} \quad | \quad 0 + A_{2,1}B_{1,2}}
 \end{array}$$

`c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));`
`c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));`
 SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- $l = 2$

$$A \quad \boxed{A_{1,2} \quad | \quad A_{2,2}}$$

`_mm_load_pd`: Stored in memory in Column order

$$B_1 \quad \boxed{B_{2,1} \quad | \quad B_{2,1}}$$

$$B_2 \quad \boxed{B_{2,2} \quad | \quad B_{2,2}}$$

`_mm_load1_pd`: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

$$\begin{array}{cc}
 & C_{1,1} & C_{2,1} \\
 C_1 & \boxed{A_{1,1}B_{1,1}+A_{1,2}B_{2,1} \quad | \quad A_{2,1}B_{1,1}+A_{2,2}B_{2,1}} \\
 C_2 & \boxed{A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \quad | \quad A_{2,1}B_{1,2}+A_{2,2}B_{2,2}} \\
 & C_{1,2} & C_{2,2}
 \end{array}$$

```

c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));

```

SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- $l = 2$

$$A \quad \boxed{A_{1,2} \quad | \quad A_{2,2}}$$

`_mm_load_pd`: Stored in memory in Column order

$$B_1 \quad \boxed{B_{2,1} \quad | \quad B_{2,1}}$$

$$B_2 \quad \boxed{B_{2,2} \quad | \quad B_{2,2}}$$

`_mm_load1_pd`: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)

Example: 2 x 2 Matrix Multiply (Part 1 of 2)

```
#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in
// comments as v1 = [ a | b]
// where v1 is a variable of type __m128d and
// a, b are doubles

int main(void) {
    // allocate A,B,C aligned on 16-byte boundaries
    double A[4] __attribute__((aligned(16)));
    double B[4] __attribute__((aligned(16)));
    double C[4] __attribute__((aligned(16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1,c2,a,b1,b2;
```

```
// Initialize A, B, C for example
/* A =                (note column order!)
    1 0
    0 1
*/
A[0] = 1.0; A[1] = 0.0; A[2] = 0.0; A[3] = 1.0;

/* B =                (note column order!)
    1 3
    2 4
*/
B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

/* C =                (note column order!)
    0 0
    0 0
*/
C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
```

Example: 2 x 2 Matrix Multiply (Part 2 of 2)

```
// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a =
       i = 0: [a_11 | a_21]
       i = 1: [a_12 | a_22]
    */
    a = _mm_load_pd(A+i*lda);
    /* b1 =
       i = 0: [b_11 | b_11]
       i = 1: [b_21 | b_21]
    */
    b1 = _mm_load1_pd(B+i+0*lda);
    /* b2 =
       i = 0: [b_12 | b_12]
       i = 1: [b_22 | b_22]
    */
    b2 = _mm_load1_pd(B+i+1*lda);
```

```
    /* c1 =
       i = 0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
       i = 1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
    */
    c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
    /* c2 =
       i = 0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
       i = 1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
    */
    c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
}

// store c1,c2 back into C for completion
_mm_store_pd(C+0*lda,c1);
_mm_store_pd(C+1*lda,c2);

// print C
printf("%g,%g\n%g,%g\n",C[0],C[2],C[1],C[3]);
return 0;
}
```

Inner loop from gcc -O -S

```
L2: movapd    (%rax,%rsi), %xmm1 //Load aligned A[i,i+1]->m1
    movddup  (%rdx), %xmm0      //Load B[j], duplicate->m0
    mulpd    %xmm1, %xmm0       //Multiply m0*m1->m0
    addpd    %xmm0, %xmm3       //Add m0+m3->m3
    movddup  16(%rdx), %xmm0    //Load B[j+1], duplicate->m0
    mulpd    %xmm0, %xmm1       //Multiply m0*m1->m1
    addpd    %xmm1, %xmm2       //Add m1+m2->m2
    addq     $16, %rax           // rax+16 -> rax (i+=2)
    addq     $8, %rdx           // rdx+8 -> rdx (j+=1)
    cmpq    $32, %rax          // rax == 32?
    jne     L2                 // jump to L2 if not equal
    movapd  %xmm3, (%rcx)      //store aligned m3 into C[k,k+1]
    movapd  %xmm2, (%rdi)      //store aligned m2 into C[l,l+1]
```

And in Conclusion, ...

- Amdahl's Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
 - Exploit data-level parallelism in loops
 - One instruction fetch that operates on multiple operands simultaneously
 - 128-bit XMM registers
- SSE Instructions in C
 - Embed the SSE machine instructions directly into C programs through use of intrinsics
 - Achieve efficiency beyond that of optimizing compiler