Introduction to SIMD

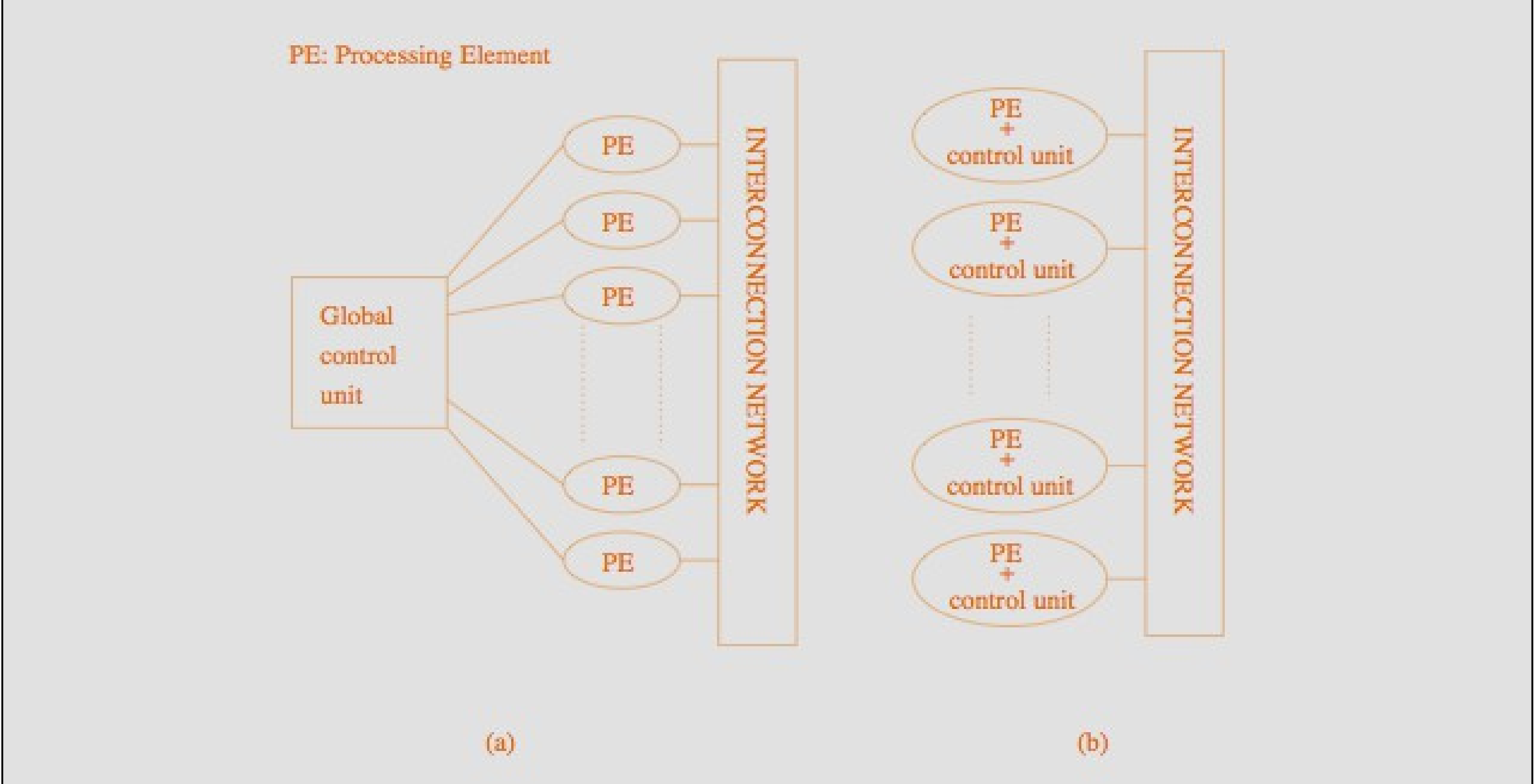
Parallel Hardware

- Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated.
- However, since this form of parallelism isn't usually visible to the programmer, we are treating both of them as extensions to the basic von Neumann model.

Review: Predominant Parallel Control Mechanisms

Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: `if ($myid == 0) { }`

SIMD and MIMD Architectures: What's the Difference?



A typical SIMD architecture (a) and a typical MIMD architecture (b).

Slide source: Grama et al., Introduction to Parallel Computing, http://www.users.cs.umn.edu/~karypis/parbook

Overview of SIMD Programming

- Vector architectures
- Early examples of SIMD supercomputers
- TODAY Mostly
  - Multimedia extensions such as SSE(Streaming SIMD Extensions) and AltiVec
  - Graphics and games processors (CUDA, stay tuned)
  - Accelerators (e.g., ClearSpeed)
- Is there a dominant SIMD programming model
  - Unfortunately, NO!!!
- Why not?
  - Vector architectures were programmed by scientists
  - Multimedia extension architectures are programmed by systems programmers (almost assembly language!)
  - GPUs are programmed by games developers (domain-specific libraries)

SIMD

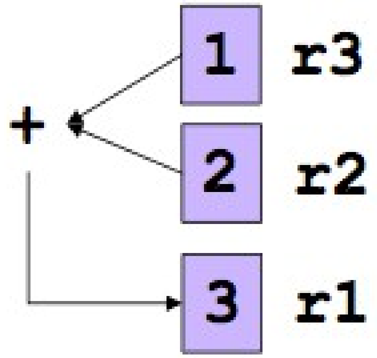
- SIMD systems are parallel systems
- Applying same instruction(operation) to multiple data items
- It has a single control unit and multiple ALUs
- An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item or it is idle.
- Ex: vector addition

2 arrays x and y, each with n elements

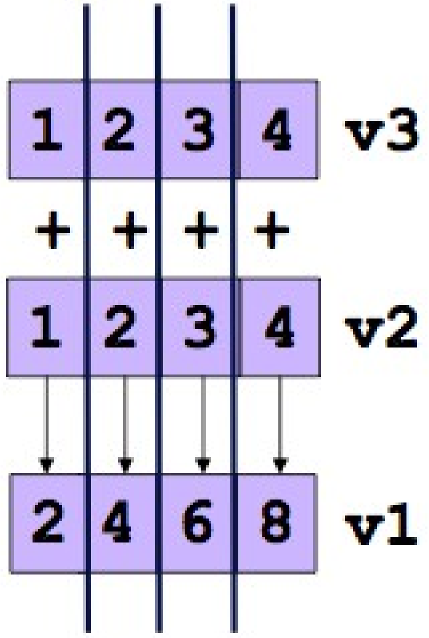
```
for (i=0; i<n; i++)
    x[i] += y[i]
```

Scalar vs. SIMD in Multimedia Extensions

```
Scalar: add r1,r2,r3
```



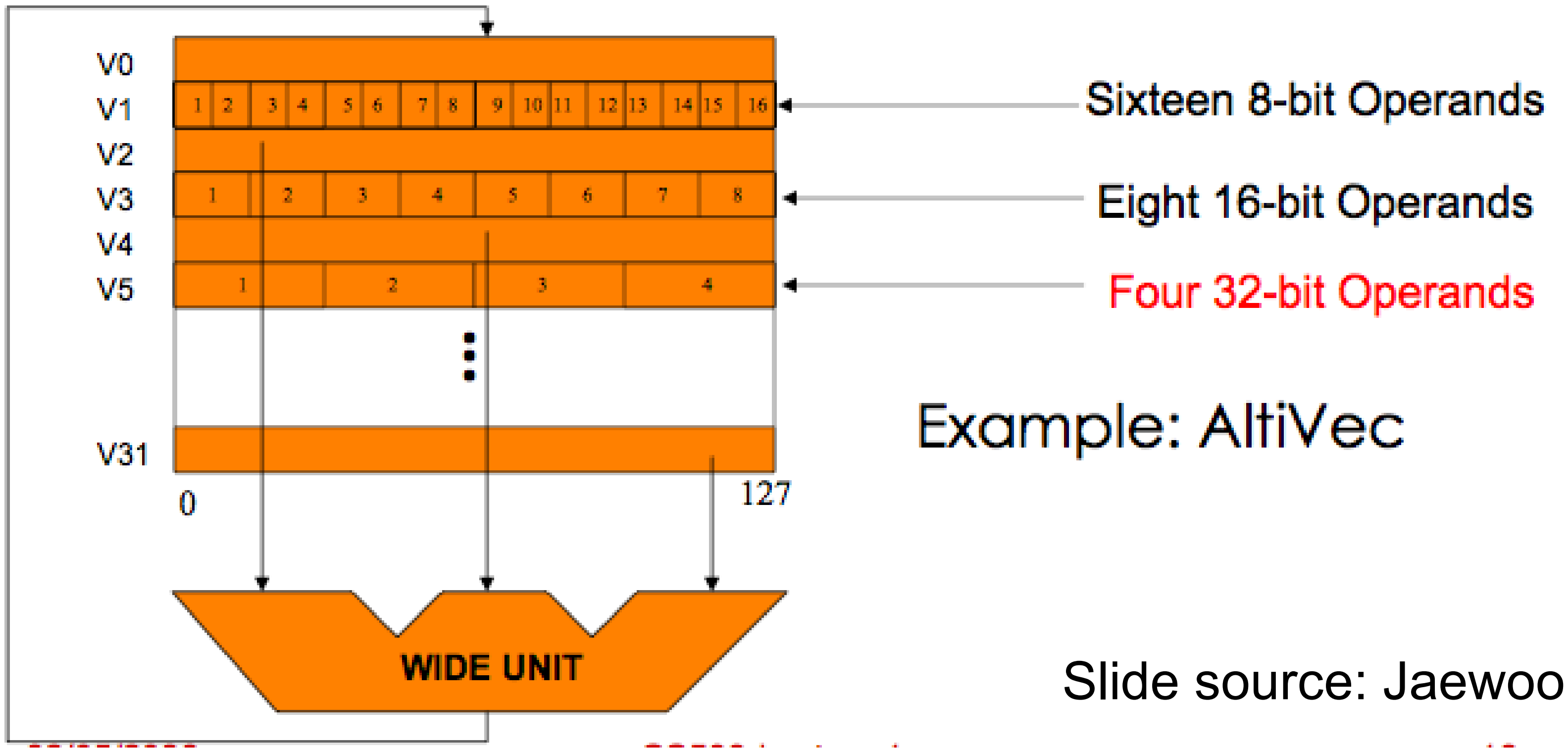
```
SIMD: vadd<sws> v1,v2,v3
```



Slide source: Sam Larsen

Multimedia Extension Architectures

- At the core of multimedia extensions
  - SIMD parallelism
  - Variable-sized data fields:
  - Vector length = register width / type size



Example: AltiVec

Slide source: Jaewook Shin

- Case 1:
  - Consider SIMD system has n ALUs.
  - Load x[i] and y[i] into the ith ALU
  - Have the ith ALU add y[i] to x[i]
  - Store the result in x[i]
-

- Case 2:
  - Consider SIMD system has m ALUs and m<n
  - Execute addittions in blocks of m elements at a time
  - If M =4; n =15
  - We can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14.
  - In the last group of elemnts, 12 to 14; it operates only on 3 elements of x and y
  - So one of the four ALUs will be idle.

Note: The requirement that all the ALUs execute the same instruction or are idle can seriously **degrade** the **overall performance** of a SIMD system.

1)For eg:, Suppose we only want to carry out addition if y[i] is positive:

```
for (i=0; i<n; i++)
    if y[i] > 0.0
        x[i] += y[i]
```

1)Load each element of y into an ALU and determine whether it's positive.

2)If y[i] is positive, carry out the addition

3)Else ALU storing y[i] will be idle while the other ALUs carry out the addition.

Implications of SIMD

- The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system.

- Classical SIMD systems : ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding.

- Further, ALUs have no instruction storage, so an ALU can't delay execution of an instruction by storing it for later execution.

Data Parallelism in SIMD systems

- SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data.
- Parallelism that obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called data parallelism

SIMD Parallelism

- SIMD systems work well for employing simple loops that operate on large arrays of data.
- SIMD Parallelism can be very efficient on large data parallel problems, but SIMD systems often don't do very well on other types of parallel problems.
- In the early1990s a marker of SIMD systems(Thinking Machines) was the largest manufacturer of parallel supercomputers.
- By the late 1990s the only widely produced SIMD systems were vector processors.
- More recently, GPUs and desktop CPUs are making use of aspects of SIMD computing.

Multimedia / Scientific Applications

- Image
  - Graphics : 3D games, movies
  - Image recognition
  - Video encoding/decoding : JPEG, MPEG4
- Sound
  - Encoding/decoding: IP phone, MP3
  - Speech recognition
  - Digital signal processing: Cell phones
- Scientific applications
  - Array-based data-parallel computation, another level of parallelism

Vector Processors

- By the late 1990s the only widely produced SIMD systems were vector processors.
- They are used to operate over large arrays or vectors of data.
- While conventional CPUs operate over individual data elements or scalars.

Characteristics of Vector Processors

- Vector registers:
    - Used to store a vector of operands
    - They are capable of operating simulataneously on their contents.
    - Vector length may range from 4 to 128 64-bit elements.

- Vectorized & pipelined functional units:

    It contain pipelined functional units to support either of the following:

    - To apply the same operation to each element in the vector
    - To apply the same operation to each pair of corresponding elements in the 2 vectors

Characteristics of Vector Processors

- Vector instructions:
    - They are used to operate on vectors rather than scalars.
    - If the vector length is vector_length, these instructions have the great virtue that a simple loop such as

```
for(i=0; i<n' i++)
    x[i] + = y[i];
```

requires only a single load, add and store for each block of vector length elements

while a conventional system requires a load, add and store for each element.

-

Characteristics of Vector Processors

- Interleaved memory:
  - The memory system consists of multiple "banks" of memory, which can be accessed more or less independently.
  - After accessing one bank. There will be a delay before it can be re-accessed, but a different bank can be accessed much sooner
  - So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.

Characteristics of Vector Processors

- Strided memory access :
  - In strided memory access, the program accesses elements of a vector located at regular intervals.
  - These regular intervals are stated as stride value.

    For eg: accessing the 1st element, 5th element, the 9th element and so on, would be strided access with a stride of four.
  -

Characteristics of Vector Processors

- hardware scatter/gather:
  - Scatter refers to write operation and gather refers to read elements of a vector located at irregular intervals.
  - For eg. Accessing the 1st element, 3rd element, the 4th element, the 10th element and so on

Advantages of Vector Processors

- Vector systems are very fast
- They are easy to use
- Compiler vectorization works well at identifying code that can be vectorized.
  - Further, they identify loops that cannot be vectorized, and they often provide information about why a loop couldn't be vectorized.
  - The user can therby make informed decisions about whether it's possible to rewrite the loop so that it will vectorize.
- Vector processors have very high memory bandwidth.

Advantages of Vector Processors

- Vector processors have very high memory bandwidth.
    - Every data item that's loaded is actually used, unlike cache based systems that may not make use of every item in a cache line.
-

Disadvantages of Vector Processors

- Vector processors are not suitable to handle irregular data structures.
  - Works only if parallelism is regular (data/SIMD parallelism)
    - ++ Vector operations
    - -- very inefficient if parallelism is irregular
      - ➢ Searching a key in a linked list
- Scalability is limited
- Vector processors used to operate on long vectors are very expensive.

Graphics Processing Units (GPUs)

- To represent the surface of an object internally, GPUs use points, lines & triangles.
- All these internal representation is converted into an array of pixels using a graphics processing pipeline which can be sent to a computer screen.
-

Graphics Processing Units (GPUs)

- Shader Functions:
  - Used to describe the behavior of the programmable stages.
  - They are very short and can be implemented using a few lines of C code
  - Shader functions can be implemented to multiple elements at the same time since they are implicitely parallel.
  -

Programming Multimedia Extensions

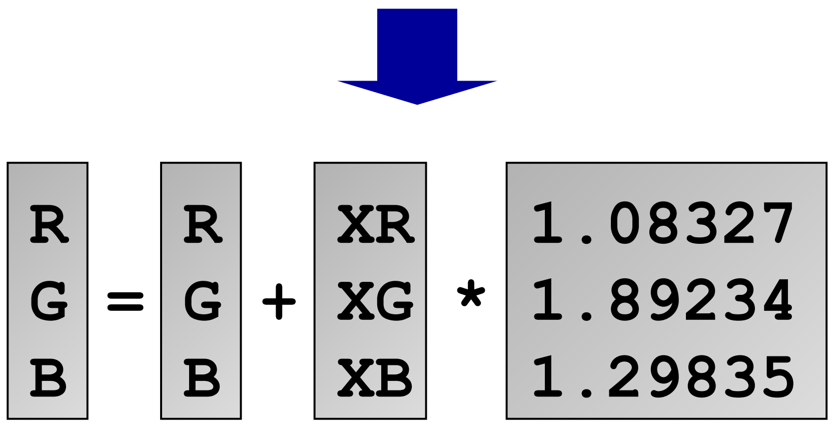
- Language extension
  - Programming interface similar to function call
  - C: built-in functions, Fortran: intrinsics
  - Most native compilers support their own multimedia extensions
    - GCC: -faltivec, -msse2
    - AltiVec: dst= vec_add(src1, src2);
    - SSE2: dst= _mm_add_ps(src1, src2);
    - BG/L: dst= __fpadd(src1, src2);
    - No Standard !
- Need automatic compilation
  - PhD student Jaewook Shin wrote a thesis on locality optimizations and control flow optimizations for SIMD multimedia extensions

Exploiting SLP with SIMD Execution

- Benefit:
  - Multiple ALU ops $\rightarrow$ One SIMD op
  - Multiple ld/st ops $\rightarrow$ One wide mem op
- What are the overheads:
  - Packing and unpacking:
    - rearrange data so that it is contiguous
  - Alignment overhead
    - Accessing data from the memory system so that it is aligned to a "superword" boundary
  - Control flow
    - Control flow may require executing all paths

1. Independent ALU Ops

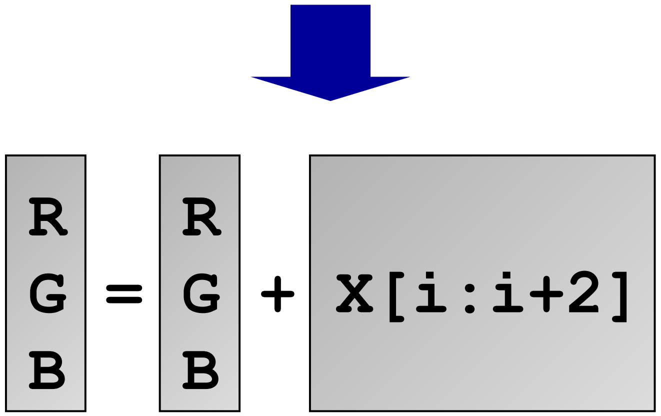
```
R = R + XR * 1.08327
G = G + XG * 1.89234
B = B + XB * 1.29835
```



Slide source: Sam Larsen

2. Adjacent Memory References

```
R = R + X[i+0]
G = G + X[i+1]
B = B + X[i+2]
```



Slide source: Sam Larsen

3. Vectorizable Loops

```
for (i=0; i<100; i+=1)
   A[i+0] = A[i+0] + B[i+0]
```

Slide source: Sam Larsen

3. Vectorizable Loops

```
for (i=0; i<100; i+=4)
    A[i+0] = A[i+0] + B[i+0]
    A[i+1] = A[i+1] + B[i+1]
    A[i+2] = A[i+2] + B[i+2]
    A[i+3] = A[i+3] + B[i+3]
```

↓

```
for (i=0; i<100; i+=4)
    A[i:i+3] = B[i:i+3] + C[i:i+3]
```

Slide source: Sam Larsen

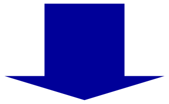
4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=1)
    L = A[i+0] - B[i+0]
    D = D + abs(L)
```

Slide source: Sam Larsen

4. Partially Vectorizable Loops

```
for (i=0; i<16; i+=2)
    L = A[i+0] – B[i+0]
    D = D + abs(L)
    L = A[i+1] – B[i+1]
    D = D + abs(L)
```



```
for (i=0; i<16; i+=2)
    L0
    L1  = A[i:i+1] – B[i:i+1]
    D = D + abs(L0)
    D = D + abs(L1)
```

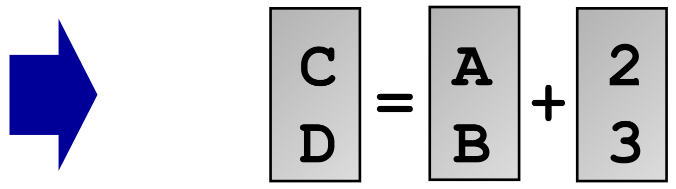
Slide source: Sam Larsen

Programming Complexity Issues

- High level: Use compiler
  - may not always be successful
- Low level: Use intrinsics or inline assembly tedious and error prone (certain subroutines are written in C/C++ called as intrinsic functions, which are built in to the compiler)
- Data must be aligned,and adjacent in memory
  - Unaligned data may produce incorrect results
  - May need to copy to get adjacency (overhead)
- Control flow introduces complexity and inefficiency
- Exceptions may be masked

Packing/Unpacking Costs

```
C = A + 2
D = B + 3
```

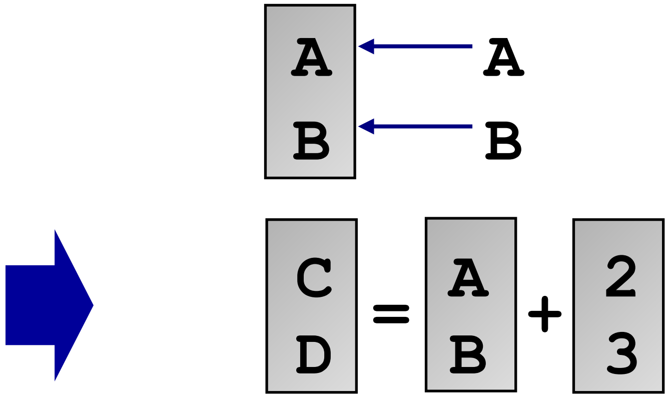


Slide source: Sam Larsen

Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory

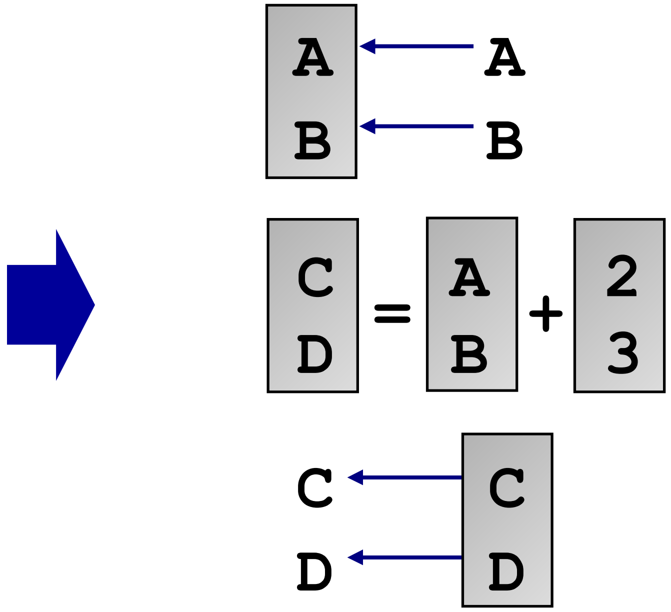
```
A = f()
B = g()
C = A + 2
D = B + 3
```



Packing/Unpacking Costs

- Packing source operands
  - Copying into contiguous memory
- Unpacking destination operands
  - Copying back to location

```
A = f()
B = g()
C = A + 2
D = B + 3
E = C / 5
F = D * 7
```

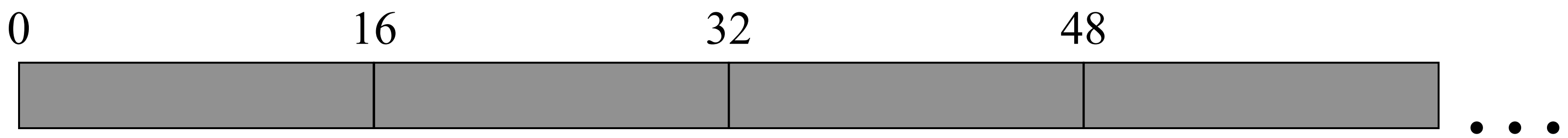


Slide source: Sam Larsen

Alignment Code Generation

- Aligned memory access
  - The address is always a multiple of 16 bytes
  - Just one superword load or store instruction

```
float a[64];
for (i=0; i<64; i+=4)
   Va = a[i:i+3];
```



Text Book Example

- An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle. As an example, suppose we want to carry out a "vector addition."
- suppose we have two arrays x and y, each with n elements, and we want to add the elements of y to the elements of x:

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

Text Book Example (Cont…)

- Suppose further that our SIMD system has n ALUs. Then we could load x[i] and y[i] into the ith ALU, have the ith ALU add y[i] to x[i], and store the result in x[i].

Text Book Example (Cont…)

- If the system has m ALUs and m < n, we can simply execute the additions in blocks of m elements at a time.
- For example, if m = 4 and n = 15, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14.
- Note that in the last group of elements in our example—elements 12 to 14—we're only operating on three elements of x and y, so one of the four ALUs will be idle

Text Book Example (Cont…)

- What is the requirement?

  All the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system.

- For example, suppose we only want to carry out the addition if y[i] is positive:

```
    for (i = 0; i < n; i++)
        if (y[i] > 0.0) x[i] += y[i];
```

- Here, load each element of y into an ALU and determine whether it's positive.

Text Book Example (Cont…)

- If y[i] is positive, we can proceed to carry out the addition. Otherwise,
- the ALU storing y[i] will be idle while the other ALUs carry out the addition.

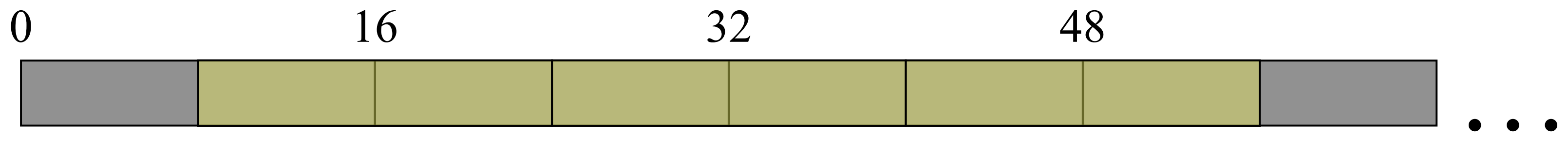
Alignment Code Generation (cont.)

- Misaligned memory access
  - The address is always a non-zero constant offset away from the 16 byte boundaries.
  - Static alignment: For a misaligned load, issue two adjacent aligned loads followed by a merge.

```
float a[64];
for (i=0; i<60; i+=4)
   Va = a[i+2:i+5];
```

$\Rightarrow$

```
float a[64];
for (i=0; i<60; i+=4)
   V1 = a[i:i+3];
   V2 = a[i+4:i+7];
   Va = merge(V1, V2, 8);
```



Compiler Optimizations

- Statically align loop iterations

```
float a[64];
for (i=0; i<60; i+=4)
  Va = a[i+2:i+5];

float a[64];
Sa2 = a[2]; Sa3 = a[3];
for (i=2; i<62; i+=4)
  Va = a[i:i+3];
```

Alignment Code Generation (cont.)

- Unaligned memory access
  - The offset from 16 byte boundaries is varying or not enough information is available.
  - Dynamic alignment: The merging point is computed during run time.

```
float a[64];
start = read();
for (i=start; i<60; i++)
   Va = a[i:i+3];
```

```
float a[64];
start = read();
for (i=start; i<60; i++)
   V1 = a[i:i+3];
   V2 = a[i+4:i+7];
   align = (&a[i:i+3])%16;
   Va = merge(V1, V2, align);
```

"classical" SIMD system

- ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcastbefore proceeding
- Further, the ALUs have no instruction storage, so an ALU can't delay execution of an instruction by storing it for later execution

Data-parallelism

- SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data

History of SIMD

- In the early 1990s a maker of SIMD systems (Thinking Machines) was the largest manufacturer of parallel supercomputers.
- However, by the late 1990s the only widely produced SIMD systems were vector processors.
- More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.