Chapter 3

Subword parallelism

(This set contains adaptations of ten slides from MK Publishers´ originals and 25 new, complementary slides by Luiz Santos)

Streaming SIMD Extension 2 (SSE2)

- Adds 8 × 128-bit registers
  - Extended to 16 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision (*double* in C)
  - 4 × 32-bit single precision (*float* in C)
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data

Example: Matrix Multiplication

- C = C + A × B (DGEMM)
  - Double precision General Matrix Multiply
- Hypothesis:
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double c[][],
          double a[][], double b[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        c[i][j] = c[i][j]
                  + a[i][k] * b[k][j];
}
```

Matrix: vectorial representation

- For higher performance:
  - Single-dimensional representation of a matrix
  - Column-major transformation

n

c_{00}	c_{01}	c_{02}	
c_{10}	c_{11}	c_{12}	n
c_{20}	c_{21}	c_{22}	

Address arithmetic

c_{ij} **= C[i+j*n]**

i, j in [0,n-1]

c_{ij} **= C[i+j*3]**

i, j in [0,2]

C[0]	c_{00}	
C[1]	c_{10}	n
C[2]	c_{20}	
C[3]	c_{01}	
C[4]	c_{11}	n
C[5]	c_{21}	
C[6]	c_{02}	
C[7]	c_{12}	n
C[8]	c_{22}	

Example: c_{12} **= C[1+2*3] = C[7]**

Alternative representation

- Unoptimized code:

```
void dgemm (int n, double* A, double* B, double* C)
{
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j)
   {
    double cij = C[i+j*n]; /* cij = C[i][j] */
    for(int k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

Alternative representation

- x86 assembly code: (generated with gcc, 2014 version)

```
vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
mov %rsi,%rcx         # register %rcx = %rsi
xor %eax,%eax         # register %eax = 0
vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
add %r9,%rcx          # register %rcx = %rcx + %r9
vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1, element of A
add $0x1,%rax         # register %rax = %rax + 1
cmp %eax,%edi         # compare %eax to %edi
vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
jg 30 <dgemm+0x30>    # jump if %eax > %edi
add $0x1,%r11d        # register %r11 = %r11 + 1
vmovsd %xmm0,(%r10)   # Store %xmm0 into C element
```

(Inner loop only, i.e. it corresponds to lines 6 to 9 from source code)

Alternative representation

- x86 assembly code: (generated with gcc, 2014 version)

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10)  # Store %xmm0 into C element
```

(Inner loop only, i.e. it corresponds to lines 6 to 9 from source code)

Alternative representation

- x86 assembly code: (generated with gcc, 2014 version)

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

(Inner loop only, i.e. it corresponds to lines 6 to 9 from source code)

Alternative representation

- x86 assembly code: (generated with gcc, 2014 version)

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx         # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
element of A
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
```

(Inner loop only, i.e. it corresponds to lines 6 to 9 from source code)

Alternative representation

- x86 assembly code: (generated with gcc, 2014 version)

```
1. vmovsd (%r10),%xmm0   # Load 1 element of C into %xmm0
2. mov %rsi,%rcx     Cij (right)  # register %rcx = %rsi
3. xor %eax,%eax         # register %eax = 0
4. vmovsd (%rcx),%xmm1   # Load 1 element of B into %xmm1
5. add %r9,%rcx          # register %rcx = %rcx + %r9
6. vmulsd (%r8,%rax,8),%xmm1,%xmm1 # Multiply %xmm1,
element of A     aik          bkj   product
7. add $0x1,%rax         # register %rax = %rax + 1
8. cmp %eax,%edi         # compare %eax to %edi
9. vaddsd %xmm1,%xmm0,%xmm0 # Add %xmm1, %xmm0 sum-of-products
10. jg 30 <dgemm+0x30>   # jump if %eax > %edi    accumulator
11. add $0x1,%r11d       # register %r11 = %r11 + 1
12. vmovsd %xmm0,(%r10) # Store %xmm0 into C element
           Cij (left)
```

(Inner loop only, i.e. it corresponds to lines 6 to 9 from source code)

How to fully exploit AVX?

- Can be used for multiple FP operands
  - **4 × 64-bit double** precision (*double* in C)
  - 8 × 32-bit single precision (*float* in C)
- How to exploit vector operations?
  - Cannot compute scalar times scalar
  - Classic (scalar) matrix traversal inadequate...

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}

a_{00}	**a_{01}**	**a_{02}**	**a_{03}**	**a_{04}**	**a_{05}**
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
b_{40}	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 0 **k = 0**

Vectorial traversal

$$
\begin{array}{cccccc}
c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & c_{05} \\
c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\
c_{20} & c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\
c_{30} & c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\
c_{40} & c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\
c_{50} & c_{51} & c_{52} & c_{53} & c_{54} & c_{55}
\end{array}
\qquad
\begin{array}{cccccc}
a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55}
\end{array}
\qquad
\begin{array}{cccccc}
\mathbf{b_{00}} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\
\mathbf{b_{10}} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\
\mathbf{b_{20}} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\
\mathbf{b_{30}} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\
\mathbf{b_{40}} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\
\mathbf{b_{50}} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55}
\end{array}
$$

$i = 0,1,2,3$ $j = 0$ **$k = 1$**

Vectorial traversal

$\mathbf{c_{00}}$	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}	$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$	$\mathbf{b_{00}}$	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	$\mathbf{b_{10}}$	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	$\mathbf{b_{20}}$	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	$\mathbf{b_{30}}$	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	$\mathbf{b_{40}}$	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	$\mathbf{b_{50}}$	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 0 **k = 2**

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
b_{40}	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 0 **k = 3**

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	b_{40}	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 0 **k = 4**

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	b_{40}	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 0 **k = 5**

Vectorial traversal

c_{00}	$\mathbf{c_{01}}$	c_{02}	c_{03}	c_{04}	c_{05}	$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$	b_{00}	$\mathbf{b_{01}}$	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	b_{10}	$\mathbf{b_{11}}$	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	b_{20}	$\mathbf{b_{21}}$	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	b_{30}	$\mathbf{b_{31}}$	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	b_{40}	$\mathbf{b_{41}}$	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	b_{50}	$\mathbf{b_{51}}$	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 1 **k = 0**

Vectorial traversal

$$
\begin{array}{cccccc}
c_{00} & \mathbf{c_{01}} & c_{02} & c_{03} & c_{04} & c_{05} \\
c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\
c_{20} & c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\
c_{30} & c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\
c_{40} & c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\
c_{50} & c_{51} & c_{52} & c_{53} & c_{54} & c_{55}
\end{array}
\qquad
\begin{array}{cccccc}
\mathbf{a_{00}} & \mathbf{a_{01}} & \mathbf{a_{02}} & \mathbf{a_{03}} & \mathbf{a_{04}} & \mathbf{a_{05}} \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55}
\end{array}
\qquad
\begin{array}{cccccc}
b_{00} & \mathbf{b_{01}} & b_{02} & b_{03} & b_{04} & b_{05} \\
b_{10} & \mathbf{b_{11}} & b_{12} & b_{13} & b_{14} & b_{15} \\
b_{20} & \mathbf{b_{21}} & b_{22} & b_{23} & b_{24} & b_{25} \\
b_{30} & \mathbf{b_{31}} & b_{32} & b_{33} & b_{34} & b_{35} \\
b_{40} & \mathbf{b_{41}} & b_{42} & b_{43} & b_{44} & b_{45} \\
b_{50} & \mathbf{b_{51}} & b_{52} & b_{53} & b_{54} & b_{55}
\end{array}
$$

$i = 0,1,2,3$ $\quad j = 1$ $\quad \mathbf{k = 1}$

Vectorial traversal

c_{00}	$\mathbf{c_{01}}$	c_{02}	c_{03}	c_{04}	c_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}

$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

b_{00}	$\mathbf{b_{01}}$	b_{02}	b_{03}	b_{04}	b_{05}
b_{10}	$\mathbf{b_{11}}$	b_{12}	b_{13}	b_{14}	b_{15}
b_{20}	$\mathbf{b_{21}}$	b_{22}	b_{23}	b_{24}	b_{25}
b_{30}	$\mathbf{b_{31}}$	b_{32}	b_{33}	b_{34}	b_{35}
b_{40}	$\mathbf{b_{41}}$	b_{42}	b_{43}	b_{44}	b_{45}
b_{50}	$\mathbf{b_{51}}$	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 1 **k = 2**

Vectorial traversal

$$
\begin{matrix}
c_{00} & \mathbf{c_{01}} & c_{02} & c_{03} & c_{04} & c_{05} \\
c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\
c_{20} & c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\
c_{30} & c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\
c_{40} & c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\
c_{50} & c_{51} & c_{52} & c_{53} & c_{54} & c_{55}
\end{matrix}
\qquad
\begin{matrix}
\mathbf{a_{00}} & \mathbf{a_{01}} & \mathbf{a_{02}} & \mathbf{a_{03}} & \mathbf{a_{04}} & \mathbf{a_{05}} \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55}
\end{matrix}
\qquad
\begin{matrix}
b_{00} & \mathbf{b_{01}} & b_{02} & b_{03} & b_{04} & b_{05} \\
b_{10} & \mathbf{b_{11}} & b_{12} & b_{13} & b_{14} & b_{15} \\
b_{20} & \mathbf{b_{21}} & b_{22} & b_{23} & b_{24} & b_{25} \\
b_{30} & \mathbf{b_{31}} & b_{32} & b_{33} & b_{34} & b_{35} \\
b_{40} & \mathbf{b_{41}} & b_{42} & b_{43} & b_{44} & b_{45} \\
b_{50} & \mathbf{b_{51}} & b_{52} & b_{53} & b_{54} & b_{55}
\end{matrix}
$$

i = 0,1,2,3   j = 1   **k = 3**

Vectorial traversal

c_{00}	$\mathbf{c_{01}}$	c_{02}	c_{03}	c_{04}	c_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}

$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

b_{00}	$\mathbf{b_{01}}$	b_{02}	b_{03}	b_{04}	b_{05}
b_{10}	$\mathbf{b_{11}}$	b_{12}	b_{13}	b_{14}	b_{15}
b_{20}	$\mathbf{b_{21}}$	b_{22}	b_{23}	b_{24}	b_{25}
b_{30}	$\mathbf{b_{31}}$	b_{32}	b_{33}	b_{34}	b_{35}
b_{40}	$\mathbf{b_{41}}$	b_{42}	b_{43}	b_{44}	b_{45}
b_{50}	$\mathbf{b_{51}}$	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 1 **k = 4**

Vectorial traversal

c_{00}	$\mathbf{c_{01}}$	c_{02}	c_{03}	c_{04}	c_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}

$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

b_{00}	$\mathbf{b_{01}}$	b_{02}	b_{03}	b_{04}	b_{05}
b_{10}	$\mathbf{b_{11}}$	b_{12}	b_{13}	b_{14}	b_{15}
b_{20}	$\mathbf{b_{21}}$	b_{22}	b_{23}	b_{24}	b_{25}
b_{30}	$\mathbf{b_{31}}$	b_{32}	b_{33}	b_{34}	b_{35}
b_{40}	$\mathbf{b_{41}}$	b_{42}	b_{43}	b_{44}	b_{45}
b_{50}	$\mathbf{b_{51}}$	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 1 **k = 5**

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	b_{00}	b_{01}	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	b_{20}	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	b_{30}	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	b_{40}	b_{41}	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	b_{50}	b_{51}	b_{52}	b_{53}	b_{54}	b_{55}

i = 0,1,2,3 j = 31 **k = 31**

Vectorial traversal

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}	c_{05}	$\mathbf{a_{00}}$	$\mathbf{a_{01}}$	$\mathbf{a_{02}}$	$\mathbf{a_{03}}$	$\mathbf{a_{04}}$	$\mathbf{a_{05}}$	b_{00}	$\mathbf{b_{01}}$	b_{02}	b_{03}	b_{04}	b_{05}
c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	b_{10}	$\mathbf{b_{11}}$	b_{12}	b_{13}	b_{14}	b_{15}
c_{20}	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	b_{20}	$\mathbf{b_{21}}$	b_{22}	b_{23}	b_{24}	b_{25}
c_{30}	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	b_{30}	$\mathbf{b_{31}}$	b_{32}	b_{33}	b_{34}	b_{35}
c_{40}	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	b_{40}	$\mathbf{b_{41}}$	b_{42}	b_{43}	b_{44}	b_{45}
c_{50}	c_{51}	c_{52}	c_{53}	c_{54}	c_{55}	a_{50}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	b_{50}	$\mathbf{b_{51}}$	b_{52}	b_{53}	b_{54}	b_{55}

i = 4,5,6,7 j = 0 **k = 0**

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
             _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
             _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```c
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
             _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
             _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized C code: (exploiting compiler intrinsics)

```
#include <x86intrin.h>
void dgemm (int n, double* A, double* B, double* C)
{
 for ( int i = 0; i < n; i+=4 )
  for ( int j = 0; j < n; j++ ) {
   __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
   for( int k = 0; k < n; k++ )
    c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
              _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
              _mm256_broadcast_sd(B+k+j*n)));
   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
  }
}
```

Exploiting subword parallelism

- Optimized x86 assembly code:

```
vmovapd (%r11),%ymm0         # Load 4 elements of C into %ymm0
mov %rbx,%rcx                # register %rcx = %rbx
xor %eax,%eax                # register %eax = 0
vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
add $0x8,%rax                # register %rax = %rax + 8
vmulpd (%rcx),%ymm1,%ymm1    # Parallel mul %ymm1,4 A elements
add %r9,%rcx                 # register %rcx = %rcx + %r9
cmp %r10,%rax                # compare %r10 to %rax
vaddpd %ymm1,%ymm0,%ymm0     # Parallel add %ymm1, %ymm0
jne 50 <dgemm+0x50>          # jump if not %r10 != %rax
add $0x1,%esi                # register % esi = % esi + 1
vmovapd %ymm0,(%r11)         # Store %ymm0 into 4 C elements
```

[Generated with gcc (2014 version) when using C Intrinsics to induce full AVX exploitation]

(Inner loop only, i.e. it corresponds to lines 6 to 11 from source code)

Impact of subword parallelism

- Experiment: 32 x 32 matrices
  - 2.6GHz Intel Core i7 (Sandy Bridge)
  - Using a single core
- Unoptimized DGEMM
  - 1.7 GigaFLOPS
- Optimized DGEMM
  - 6.4 GigaFLOPS
- Speed up: 3.85 times as fast!

Chapter 3

Subword parallelism

Arithmetic for Multimedia

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- Saturating operations
  - On overflow, result is largest representable value
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Floating Point (brief review)

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$

normalized

not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Subword Parallellism

- Graphics and audio applications can take advantage of performing simultaneous operations on short vectors
  - Example: 128-bit adder:
    - Sixteen 8-bit adds
    - Eight 16-bit adds
    - Four 32-bit adds
- Also called data-level parallelism, vector parallelism, or Single Instruction, Multiple Data (SIMD)

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from TOS: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

x86 FP Instructions

Data transfer	Arithmetic	Compare	Transcendental
FILD mem/ST(i) FISTP mem/ST(i) FLDPI FLD1 FLDZ	FIADDP mem/ST(i) FISUBRP mem/ST(i) FIMULP mem/ST(i) FIDIVRP mem/ST(i) FSQRT FABS FRNDINT	FICOMP FIUCOMP FSTSW AX/mem	FPATAN F2XMI FCOS FPTAN FPREM FPSIN FYL2X

- Optional variations
  - I: integer operand
  - P: pop operand from stack
  - R: reverse operand order
  - But not all combinations allowed

Chapter 3

Arithmetic for Computers

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

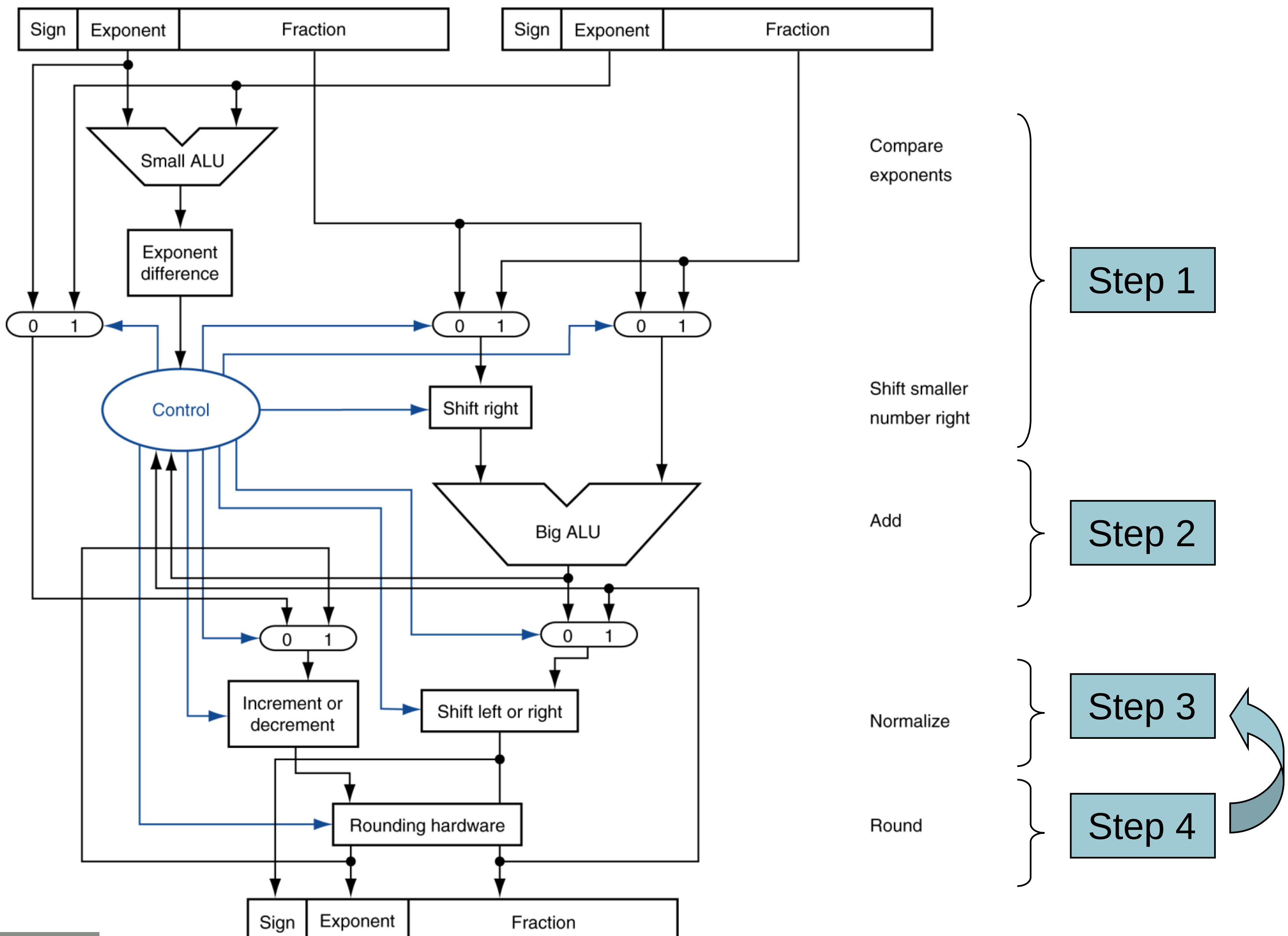
Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625$

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg’s
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in $f12, result in $f0, literals in global memory space
- Compiled MIPS code:

```
f2c: lwc1   $f16, const5($gp)
     lwc2   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

FP Example: Array Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2

FP Example: Array Multiplication

- MIPS code:

```
    li   $t1, 32         # $t1 = 32 (row size/loop end)
    li   $s0, 0          # i = 0; initialize 1st for loop
L1: li   $s1, 0          # j = 0; restart 2nd for loop
L2: li   $s2, 0          # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5     # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1   # $t2 = i * size(row) + j
    sll  $t2, $t2, 3     # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2   # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)     # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5     # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1   # $t0 = k * size(row) + j
    sll  $t0, $t0, 3     # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0   # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)    # $f16 = 8 bytes of z[k][j]
```

…

FP Example: Array Multiplication

…

```
sll   $t0, $s0, 5         # $t0 = i*32 (size of row of y)
addu  $t0, $t0, $s2       # $t0 = i*size(row) + k
sll   $t0, $t0, 3         # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0       # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)        # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16      # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1         # $k k + 1
bne   $s2, $t1, L3        # if (k != 32) go to L3
s.d   $f4, 0($t2)         # x[i][j] = $f4
addiu $s1, $s1, 1         # $j = j + 1
bne   $s1, $t1, L2        # if (j != 32) go to L2
addiu $s0, $s0, 1         # $i = i + 1
bne   $s0, $t1, L1        # if (i != 32) go to L1
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Right Shift and Division

- Left shift by *i* places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., –5 / 4
    - 11111011_2 >> 2 = 11111110_2 = –2
    - Rounds toward –∞
  - c.f. 11111011_2 >>> 2 = 00111110_2 = +62

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - “My bank balance is out by 0.0002¢!” ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

bla

$$
\begin{array}{cccccc}
c_{00} & c_{01} & c_{02} & c_{03} & c_{04} & c_{05} \\
c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\
c_{20} & c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\
c_{30} & c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \\
c_{40} & c_{41} & c_{42} & c_{43} & c_{44} & c_{45} \\
c_{50} & c_{51} & c_{52} & c_{53} & c_{54} & c_{55}
\end{array}
\quad
\begin{array}{cccccc}
a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\
a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\
a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\
a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\
a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55}
\end{array}
\quad
\begin{array}{cccccc}
b_{00} & b_{01} & b_{02} & b_{03} & b_{04} & b_{05} \\
b_{10} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\
b_{20} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\
b_{30} & b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\
b_{40} & b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\
b_{50} & b_{51} & b_{52} & b_{53} & b_{54} & b_{55}
\end{array}
$$