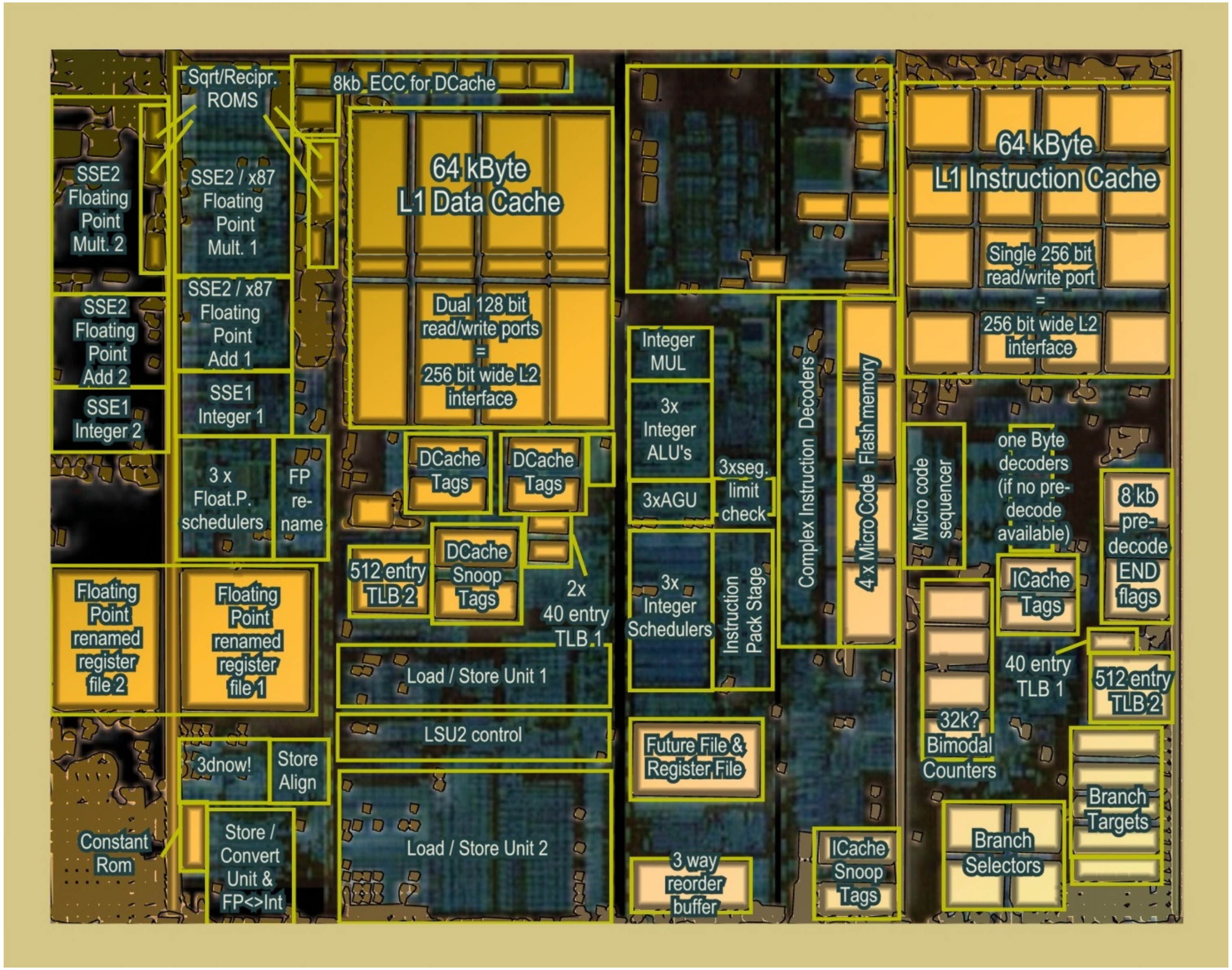
INF5063:
Programming heterogeneous multi-core processors

x86

September 17, 2010

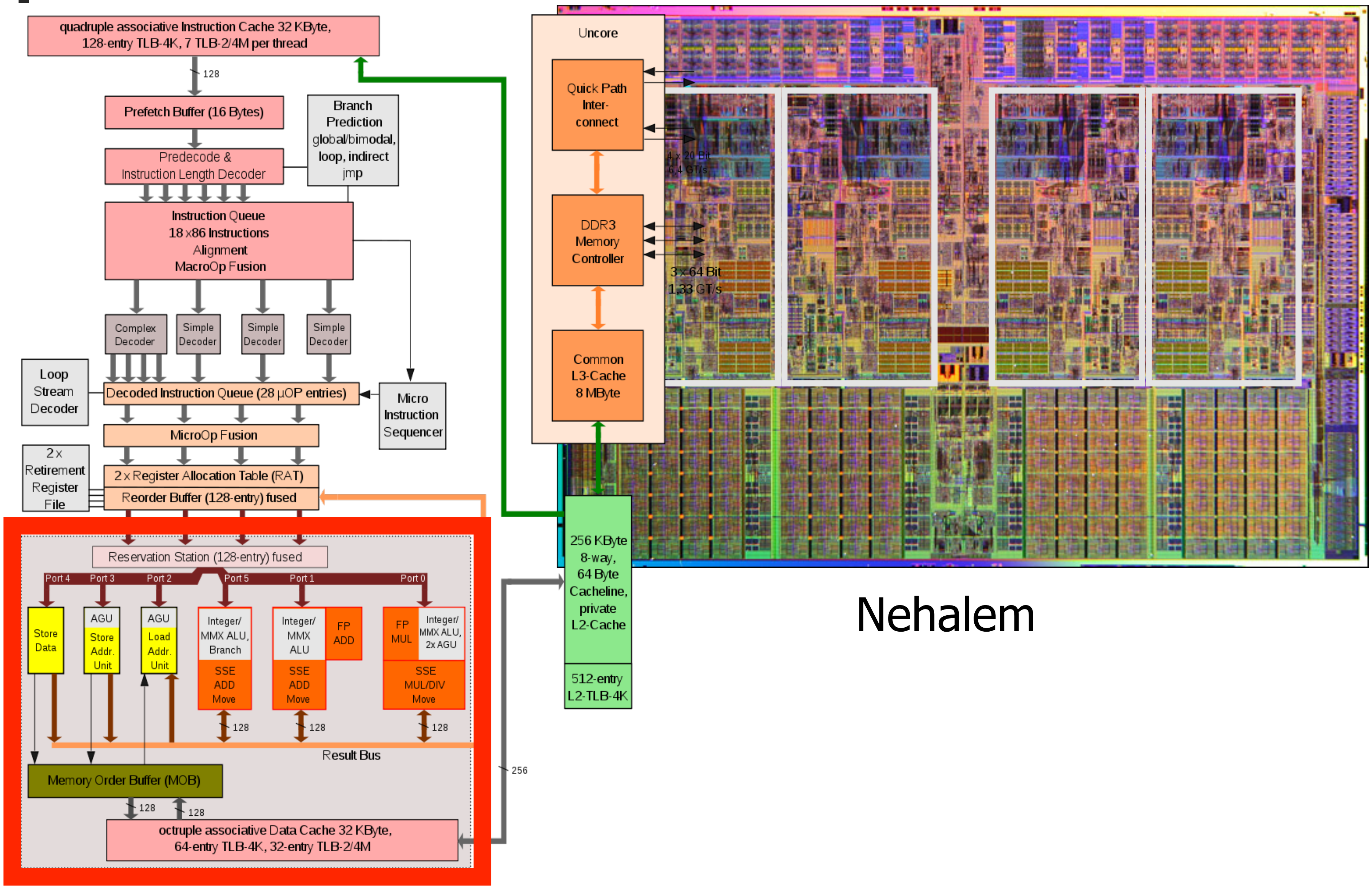
AMD K8



Sqrt/Recipr. ROMS
8kb ECC for DCache
64 kByte L1 Data Cache
Dual 128 bit read/write ports = 256 bit wide L2 interface
64 kByte L1 Instruction Cache
Single 256 bit read/write port = 256 bit wide L2 interface
SSE2 Floating Point Mult. 2
SSE2 / x87 Floating Point Mult. 1
SSE2 Floating Point Add 2
SSE2 / x87 Floating Point Add 1
SSE1 Integer 2
SSE1 Integer 1
3 x Float.P. schedulers
FP re-name
DCache Tags
DCache Tags
DCache Snoop Tags
512 entry TLB 2
2x 40 entry TLB 1
Integer MUL
3x Integer ALU's
3xAGU
3xseg. limit check
3x Integer Schedulers
Instruction Pack Stage
Complex Instruction Decoders
4 x Micro Code Flash memory
Micro code sequencer
one Byte decoders (if no pre-decode available)
8 kb pre-decode END flags
ICache Tags
40 entry TLB 1
512 entry TLB 2
32k? Bimodal Counters
Branch Targets
Branch Selectors
Floating Point renamed register file 2
Floating Point renamed register file 1
Load / Store Unit 1
LSU2 control
3dnow!
Store Align
Constant Rom
Store / Convert Unit & FP<>Int
Load / Store Unit 2
Future File & Register File
3 way reorder buffer
ICache Snoop Tags

Intel Nehalem

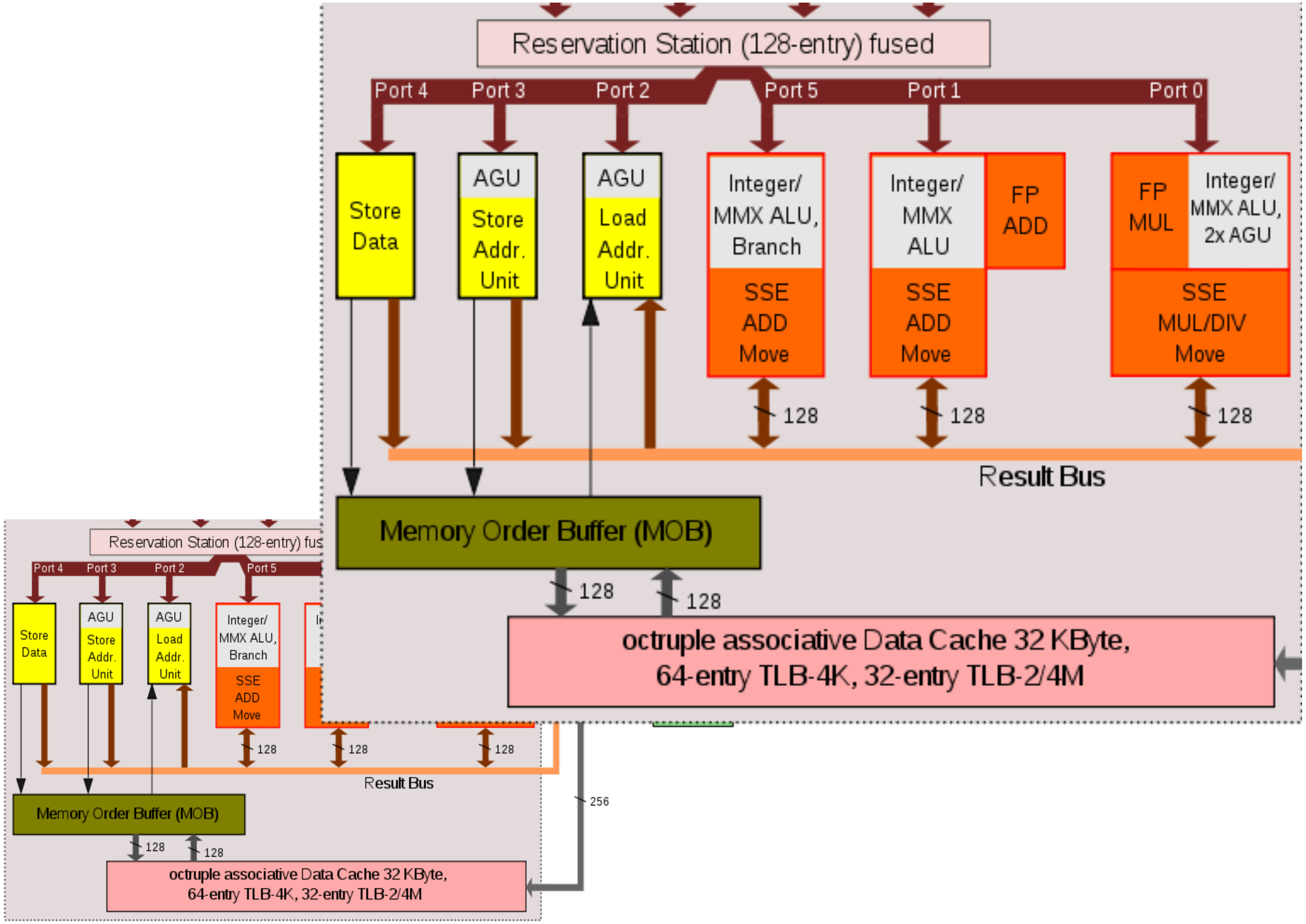
Intel Nehalem microarchitecture



Nehalem

GT/s: gigatransfers per second

Intel Nehalem



GT/s: gigatransfers per second

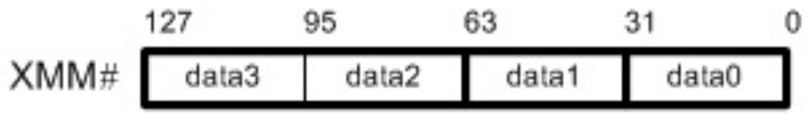
Special instructions…

- MMX
  - MMX is officially a meaningless initialism trademarked by Intel; unofficially,
    - MultiMedia eXtension
    - Multiple Math eXtension
    - Matrix Math eXtension
  - SIMD (Single Instruction, Multiple Data) computation processes multiple data in parallel with a single instruction, resulting in significant performance improvement; MMX gives 2 computations at once.
  - MMX defined 8 “new” 64-bit integer registers (mm0 ~ mm7), which were aliases for the existing x87 FPU registers – reusing 64 (out of 80) bits in the floating point registers.

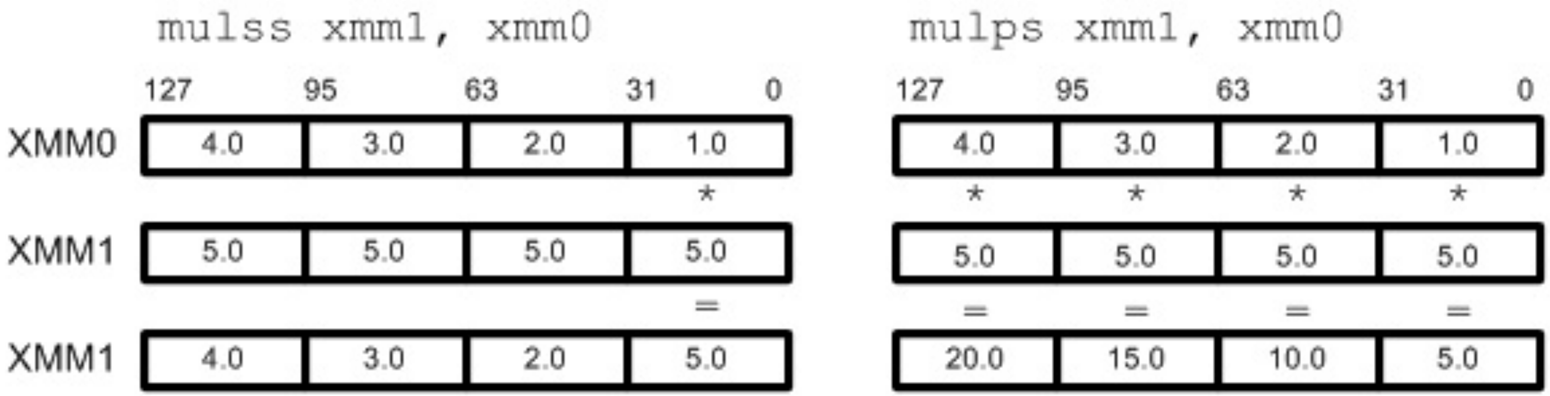
Special instructions…

- SSE
  - Streaming SIMD Extensions (SSE)
  - SIMD; 4 computations at once.

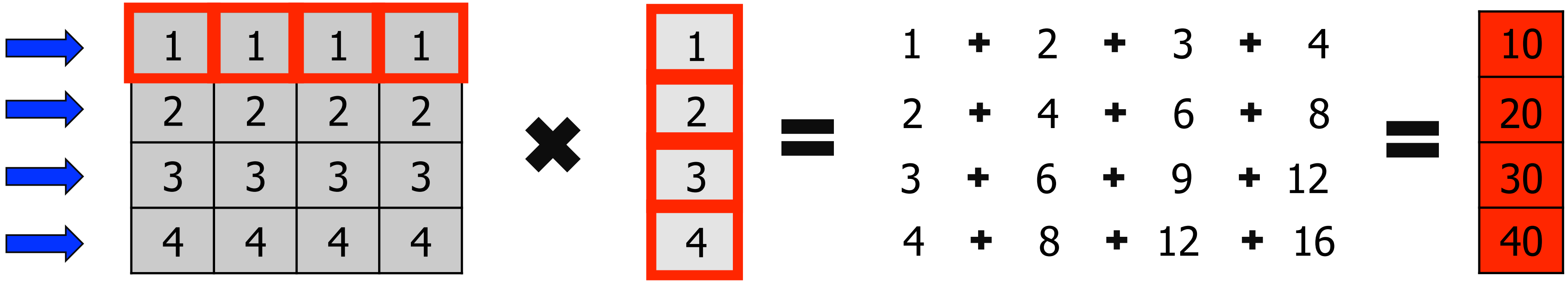
  - SSE defines 8 **new** 128-bit registers (xmm0 ~ xmm7) for single-precision floating-point computations. Since each register has 128-bit long, we can store total 4 of 32-bit floating-point numbers (1-bit sign, 8-bit exponent, 23-bit mantissa).



  - Single or packed scalar operations: `__SS` vs `__PS`



Example: Matrix Multiplication



Example: Matrix Multiplication - C

```
#include <stdio.h>

float elts[4][4] = {1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4};
float vin[4] = {1,2,3,4};
float vout[4];

void main(void)
{
   vout[0] =  elts[0][0] * vin[0]  +  elts[0][1] * vin[1] +
              elts[0][2] * vin[2]  +  elts[0][3] * vin[3];

   vout[1] =  elts[1][0] * vin[0]  +  elts[1][1] * vin[1] +
              elts[1][2] * vin[2]  +  elts[1][3] * vin[3];

   vout[2] =  elts[2][0] * vin[0]  +  elts[2][1] * vin[1] +
              elts[2][2] * vin[2]  +  elts[2][3] * vin[3];

   vout[3] =  elts[3][0] * vin[0]  +  elts[3][1] * vin[1] +
              elts[3][2] * vin[2]  +  elts[3][3] * vin[3];

   printf("%f %f %f %f\n", vout[0], vout[1], vout[2], vout[3]);
}
```

Example: Matrix Multiplication – SSE

```c
#include <stdio.h>

float elts[4][4] = {1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4};
float vin[4] = {1,2,3,4};
float vout[4];

void main(void)
{
    vout[0] =  elts[0][0] * vin[0]  +  elts[0][1] * vin[1] +
               elts[0][2] * vin[2]  +  elts[0][3] * vin[3];

    vout[1] =  elts[1][0] * vin[0]  +  elts[1][1] * vin[1] +
               elts[1][2] * vin[2]  +  elts[1][3] * vin[3];

               elts[2][0] * vin[0]  +  elts[2][1] * vin[1] +
               elts[2][2] * vin[2]  +  elts[2][3] * vin[3];

               elts[3][0] * vin[0]  +  elts[3][1] * vin[1] +
               elts[3][2] * vin[2]  +  elts[3][3] * vin[3];

    printf(... vout[1], vout[2], vout[3]);
}
```

```
__asm {
    mov       esi, VIN
    mov       edi, VOUT

    // load columns of matrix into xmm4-7
    mov       edx, ELTS
    movups    xmm4, [edx]
    movups    xmm5, [edx + 0x10]
    movups    xmm6, [edx + 0x20]
    movups    xmm7, [edx + 0x30]

    // load v into xmm0.
    movups    xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps       xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups      xmm1, xmm0
    shufps      xmm1, xmm1, 0x00
    mulps       xmm1, xmm4
    addps       xmm2, xmm1

    // repeat the process for y, z and w
    movups      xmm1, xmm0
    shufps      xmm1, xmm1, 0x55
    mulps       xmm1, xmm5
    addps       xmm2, xmm1
    movups      xmm1, xmm0
    shufps      xmm1, xmm1, 0xAA
    mulps       xmm1, xmm6
    addps       xmm2, xmm1
    movups      xmm1, xmm0
    shufps      xmm1, xmm1, 0xFF
    mulps       xmm1, xmm7
    addps       xmm2, xmm1

    // write the results to vout
    movups    [edi], xmm2
}
```

Assuming **elts** in a COLUMN-MAJOR order:

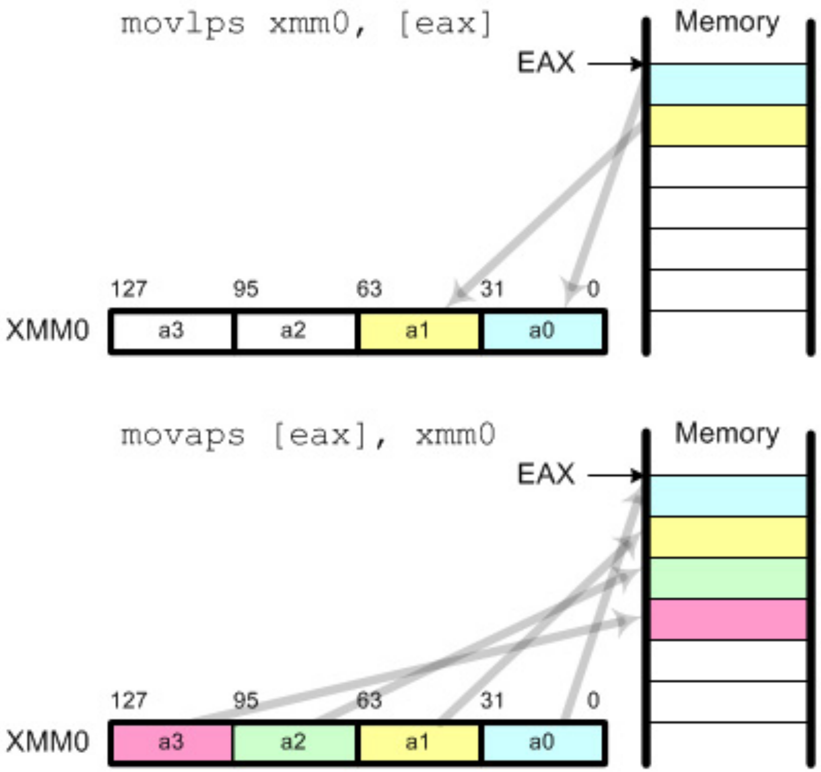
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	e	i	n	b	f	j	n	c	g	k	o	d	h	l	p

Example: Matrix Multiplication – SSE

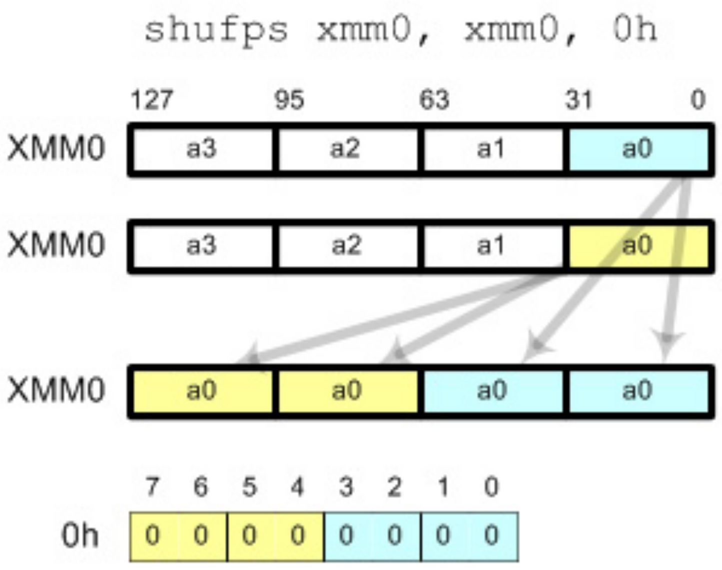
```
__asm {
        mov          esi, VIN
        mov          edi, VOUT

        // load columns of matrix into xmm4-7
        mov       edx,   ELTS
        movups    xmm4,  [edx]
        movups    xmm5,  [edx + 0x10]
        movups    xmm6,  [edx + 0x20]
        movups    xmm7,  [edx + 0x30]

        // load v into xmm0.
        movups    xmm0,  [esi]

        // we'll store the final result in xmm2; initialize it
        // to zero
        xorps       xmm2,  xmm2

        // broadcast x into xmm1, multiply it by the first
        // column of the matrix (xmm4), and add it to the total
        movups      xmm1,  xmm0
        shufps      xmm1,  xmm1,  0x00
        mulps       xmm1,  xmm4
        addps       xmm2,  xmm1

        // repeat the process for y, z and w
        movups      xmm1,  xmm0
        shufps      xmm1,  xmm1,  0x55
        mulps       xmm1,  xmm5
        addps       xmm2,  xmm1

        movups      xmm1,  xmm0
        shufps      xmm1,  xmm1,  0xAA
        mulps       xmm1,  xmm6
        addps       xmm2,  xmm1

        movups      xmm1,  xmm0
        shufps      xmm1,  xmm1,  0xFF
        mulps       xmm1,  xmm7
        addps       xmm2,  xmm1

        // write the results to vout
        movups    [edi],  xmm2
}
```

Example: Matrix Multiplication – SSE

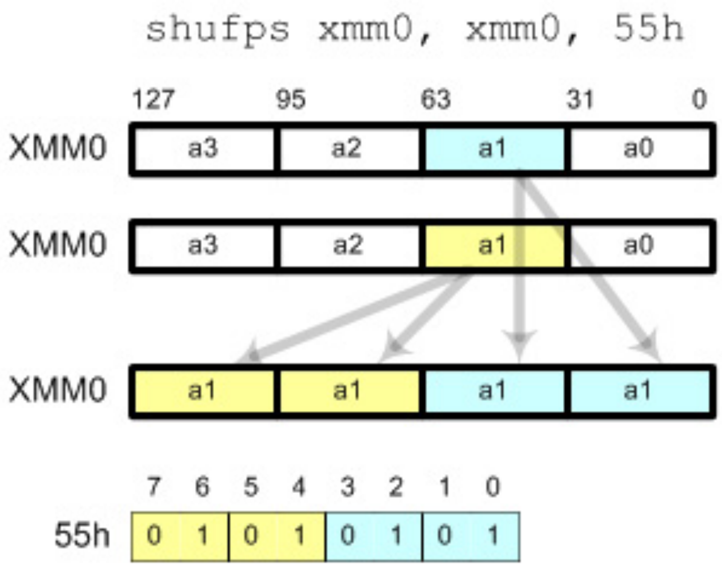


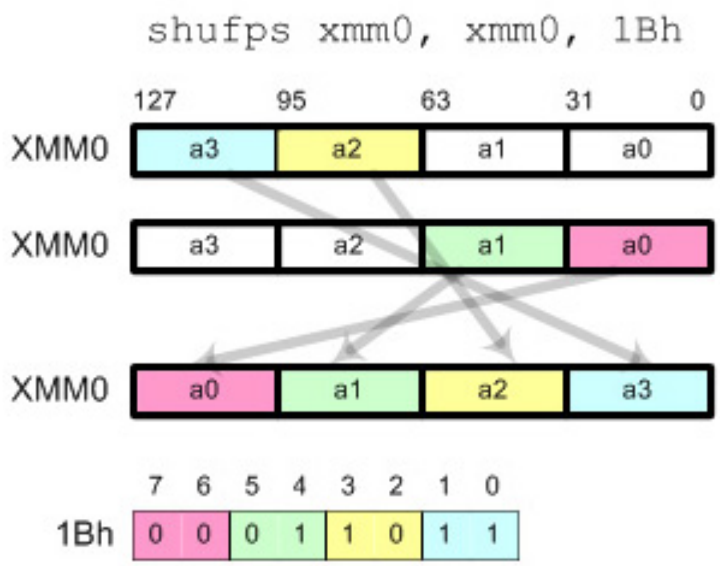

```
__asm {
        mov           esi, VIN
        mov           edi, VOUT

        // load columns of matrix into xmm4-7
        mov        edx,   ELTS
        movups     xmm4,  [edx]
        movups     xmm5,  [edx + 0x10]
        movups     xmm6,  [edx + 0x20]
        movups     xmm7,  [edx + 0x30]

        // load v into xmm0.
        movups     xmm0,  [esi]

        // we'll store the final result in xmm2; initialize it
        // to zero
        xorps        xmm2,  xmm2

        // broadcast x into xmm1, multiply it by the first
        // column of the matrix (xmm4), and add it to the total
        movups       xmm1,  xmm0
        shufps       xmm1,  xmm1,  0x00
        mulps        xmm1,  xmm4
        addps        xmm2,  xmm1

        // repeat the process for y, z and w
        movups       xmm1,  xmm0
        shufps       xmm1,  xmm1,  0x55
        mulps        xmm1,  xmm5
        addps        xmm2,  xmm1

        movups       xmm1,  xmm0
        shufps       xmm1,  xmm1,  0xAA
        mulps        xmm1,  xmm6
        addps        xmm2,  xmm1

        movups       xmm1,  xmm0
        shufps       xmm1,  xmm1,  0xFF
        mulps        xmm1,  xmm7
        addps        xmm2,  xmm1

        // write the results to vout
        movups     [edi], xmm2
}
```

Example: Matrix Multiplication – SSE

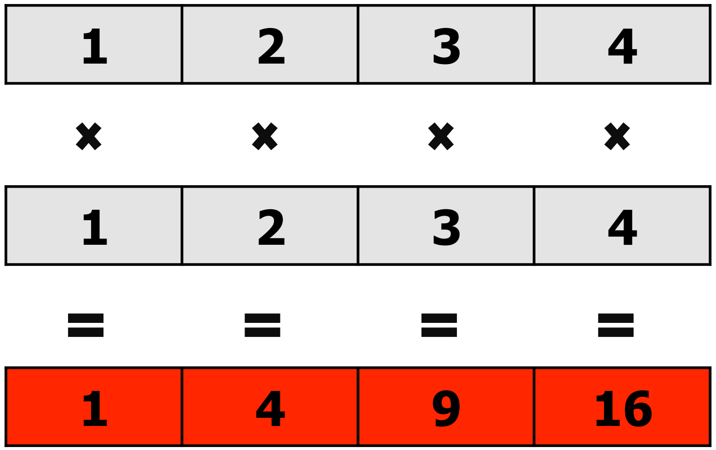



```
__asm {
    mov         esi, VIN
    mov         edi, VOUT

    // load columns of matrix into xmm4-7
    mov       edx,  ELTS
    movups    xmm4, [edx]
    movups    xmm5, [edx + 0x10]
    movups    xmm6, [edx + 0x20]
    movups    xmm7, [edx + 0x30]

    // load v into xmm0.
    movups    xmm0, [esi]

    // we'll store the final result in xmm2; initialize it
    // to zero
    xorps       xmm2, xmm2

    // broadcast x into xmm1, multiply it by the first
    // column of the matrix (xmm4), and add it to the total
    movups      xmm1, xmm0
    shufps      xmm1, xmm1, 0x00
    mulps       xmm1, xmm4
}
```



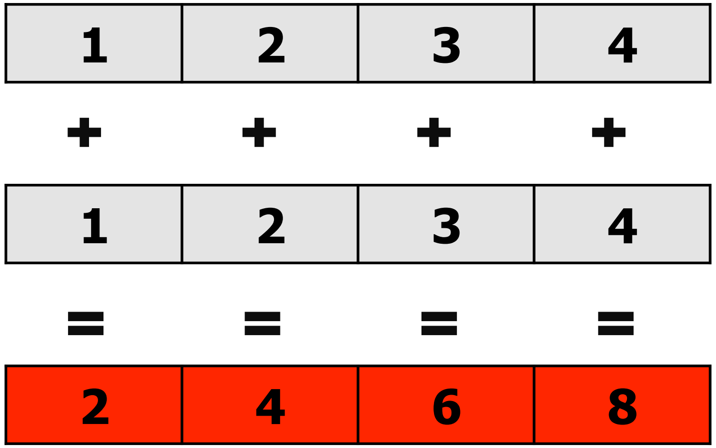


Example: Matrix Multiplication – SSE




```
__asm {
        mov          esi, VIN
        mov          edi, VOUT

        // load columns of matrix into xmm4-7
        mov       edx,  ELTS
        movups    xmm4, [edx]
        movups    xmm5, [edx + 0x10]
        movups    xmm6, [edx + 0x20]
        movups    xmm7, [edx + 0x30]

        // load v into xmm0.
        movups    xmm0, [esi]

        // we'll store the final result in xmm2; initialize it
        // to zero
        xorps       xmm2, xmm2

        // broadcast x into xmm1, multiply it by the first
        // column of the matrix (xmm4), and add it to the total
        movups      xmm1, xmm0
        shufps      xmm1, xmm1, 0x00
        mulps       xmm1, xmm4
        addps       xmm2, xmm1

        // repeat the process for y, z and w
        movups      xmm1, xmm0
        shufps      xmm1, xmm1, 0x55
        mulps       xmm1, xmm5
        addps       xmm2, xmm1

        movups      xmm1, xmm0
        shufps      xmm1, xmm1, 0xAA
        mulps       xmm1, xmm6
        addps       xmm2, xmm1

        movups      xmm1, xmm0
        shufps      xmm1, xmm1, 0xFF
        mulps       xmm1, xmm7
        addps       xmm2, xmm1

        // write the results to vout
        movups    [edi], xmm2
}
```

Example: Matrix Multiplication – SSE




```
__asm {
        mov             esi, VIN
        mov             edi, VOUT

        // load columns of matrix into xmm4-7
        mov         edx,   ELTS
        movups      xmm4, [edx]
        movups      xmm5, [edx + 0x10]
        movups      xmm6, [edx + 0x20]
        movups      xmm7, [edx + 0x30]

        // load v into xmm0.
        movups      xmm0, [esi]

        // we'll store the final result in xmm2; initialize it
        // to zero
        xorps           xmm2, xmm2

        // broadcast x into xmm1, multiply it by the first
        // column of the matrix (xmm4), and add it to the total
        movups          xmm1, xmm0
        shufps          xmm1, xmm1, 0x00
        mulps           xmm1, xmm4
        addps           xmm2, xmm1

        // repeat the process for y, z and w
        movups          xmm1, xmm0
        shufps          xmm1, xmm1, 0x55
        mulps           xmm1, xmm5
        addps           xmm2, xmm1

        movups          xmm1, xmm0
        shufps          xmm1, xmm1, 0xAA
        mulps           xmm1, xmm6
        addps           xmm2, xmm1

        movups          xmm1, xmm0
        shufps          xmm1, xmm1, 0xFF
        mulps           xmm1, xmm7
        addps           xmm2, xmm1

        // write the results to vout
        movups      [edi], xmm2
}
```

Example: Matrix Multiplication – SSE

```
__asm {
        mov             esi, VIN
        mov             edi, VOUT

        // load columns of matrix into xmm4-7
        mov         edx,   ELTS
        movups      xmm4, [edx]
        movups      xmm5, [edx + 0x10]
        movups      xmm6, [edx + 0x20]
        movups      xmm7, [edx + 0x30]

        // load v into xmm0.
        movups      xmm0, [esi]

        // we'll store the final result in xmm2; initialize it
        // to zero
        xorps         xmm2, xmm2

        // broadcast x into xmm1, multiply it by the first
        // column of the matrix (xmm4), and add it to the total
        movups        xmm1, xmm0
        shufps        xmm1, xmm1, 0x00
        mulps         xmm1, xmm4
        addps         xmm2, xmm1

        // repeat the process for y, z and w
        movups        xmm1, xmm0
        shufps        xmm1, xmm1, 0x55
        mulps         xmm1, xmm5
        addps         xmm2, xmm1

        movups        xmm1, xmm0
        shufps        xmm1, xmm1, 0xAA
        mulps         xmm1, xmm6
        addps         xmm2, xmm1

        movups        xmm1, xmm0
        shufps        xmm1, xmm1, 0xFF
        mulps         xmm1, xmm7
        addps         xmm2, xmm1

        // write the results to vout
        movups      [edi], xmm2
}
```