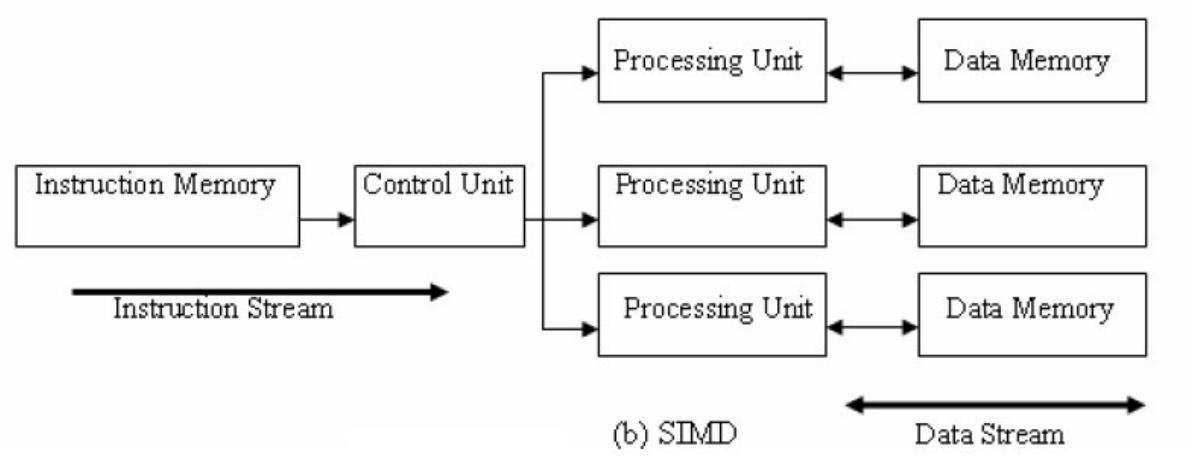
Vector Instruction Extension

SIMD architectures

- A data parallel architecture
- Applying the same instruction to many data
  - Save control logic
  - A related architecture is the vector architecture
  - SIMD and vector architectures offer high performance for vector operations.



Vector operations

- Vector addition Z = X + Y

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

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots\dots \\ x_n + y_n \end{pmatrix}$$

- Vector scaling Y = a * X

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

$$a * \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} a * x_1 \\ a * x_2 \\ \dots\dots \\ a * x_n \end{pmatrix}$$

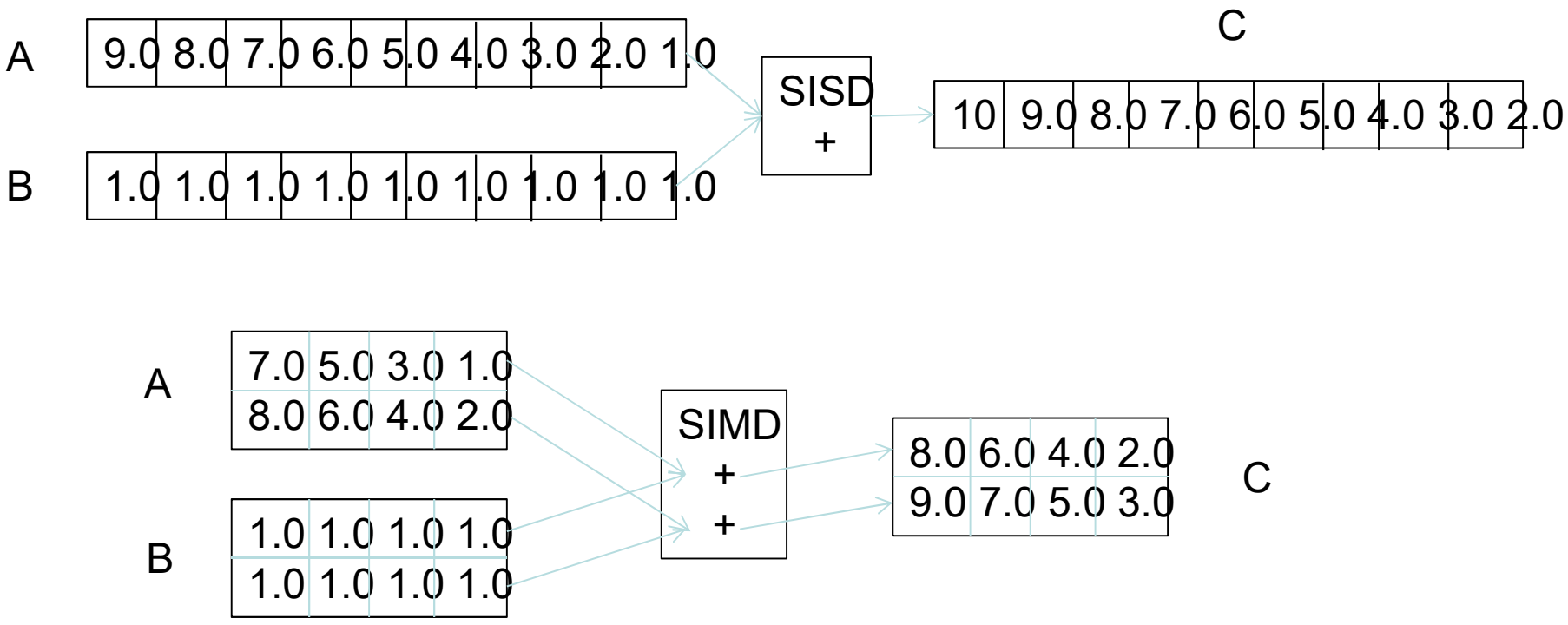
- Dot product

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

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} \bullet \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = x_1 * y_1 + x_2 * y_2 + \dots\dots + x_n * y_n$$

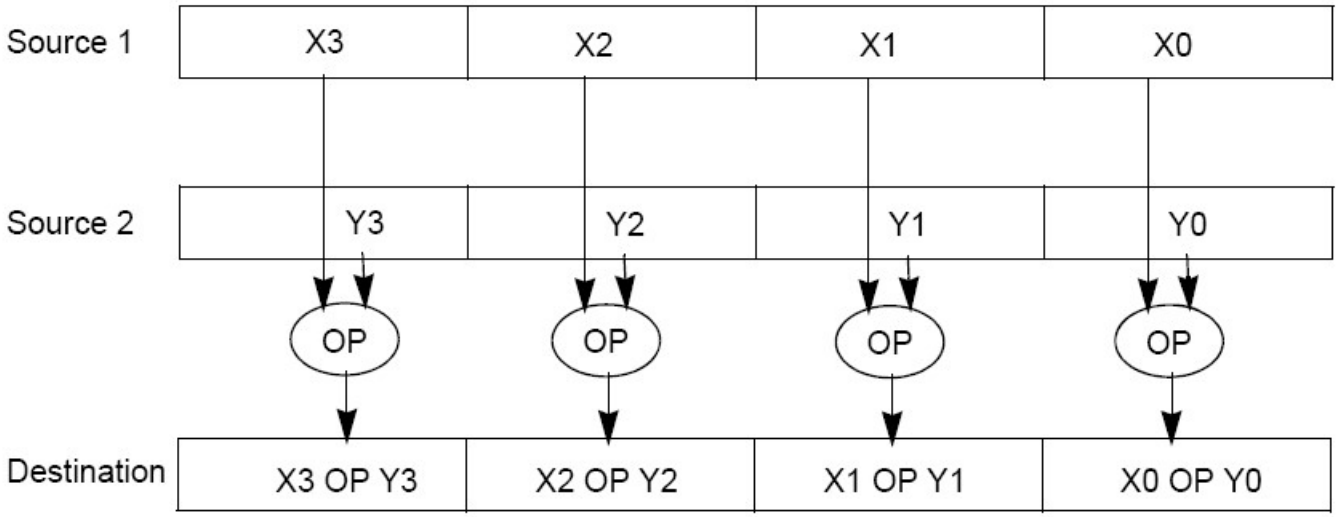
SISD and SIMD vector operations

- C = A + B
  - For (i=0;i<n; i++) c[i] = a[i] + b[i]



SIMD in IA-32 and IA-64

- To improve performance, Intel adopted SIMD (single instruction multiple data) instructions
  - MMX technology (Pentium II processor family)
  - SSE



x86 architecture SIMD support

- Both current AMD and Intel's x86 processors have ISA and microarchitecture support SIMD operations.
- ISA SIMD support
  - MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
    - See the flag field in /proc/cpuinfo
  - SSE (Streaming SIMD extensions): a SIMD instruction set extension to the x86 architecture
    - Instructions for operating on multiple data simultaneously (vector operations).
- Micro architecture support
  - Many functional units
  - 8 128-bit vector registers, XMM0, XMM1, …, XMM7

SSE programming

- Vector registers support three data types:
  - Integer (16 bytes, 8 shorts, 4 int, 2 long long int, 1 dqword)
  - single precision floating point (4 floats)
  - double precision float point (2 doubles).

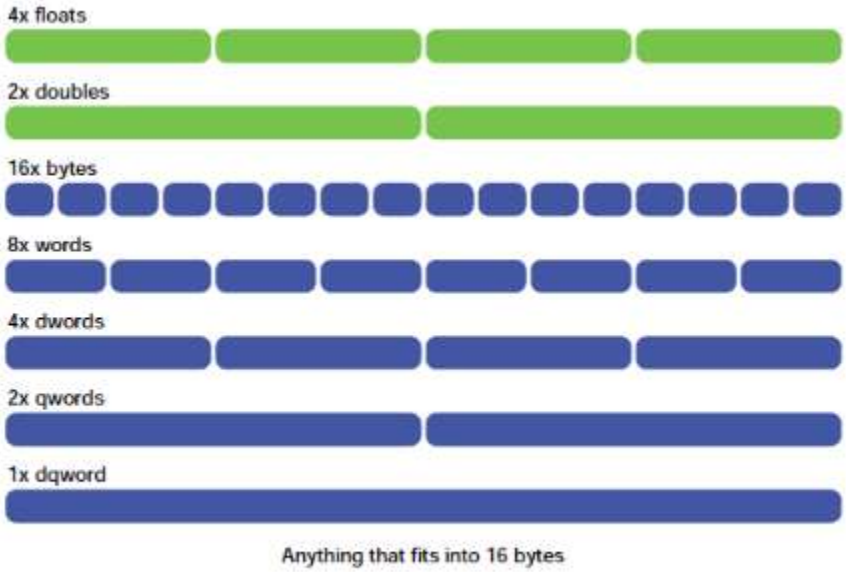
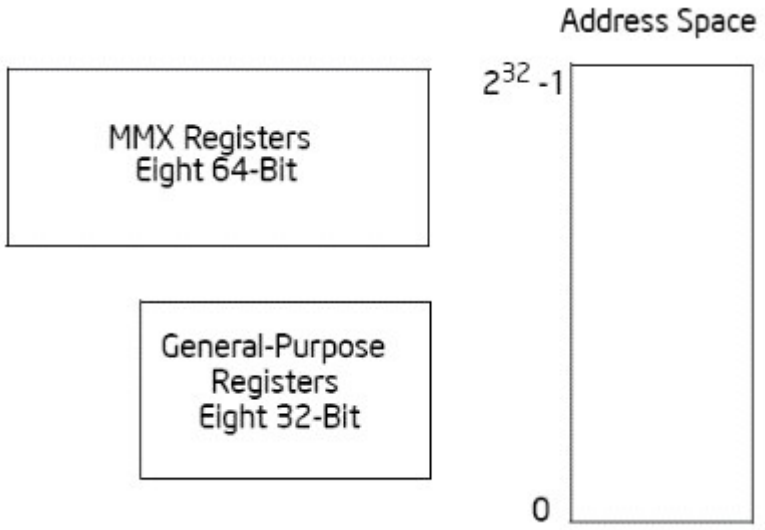


Figure 1. SSE/SSE2 data types

[Klimovitski 2001]

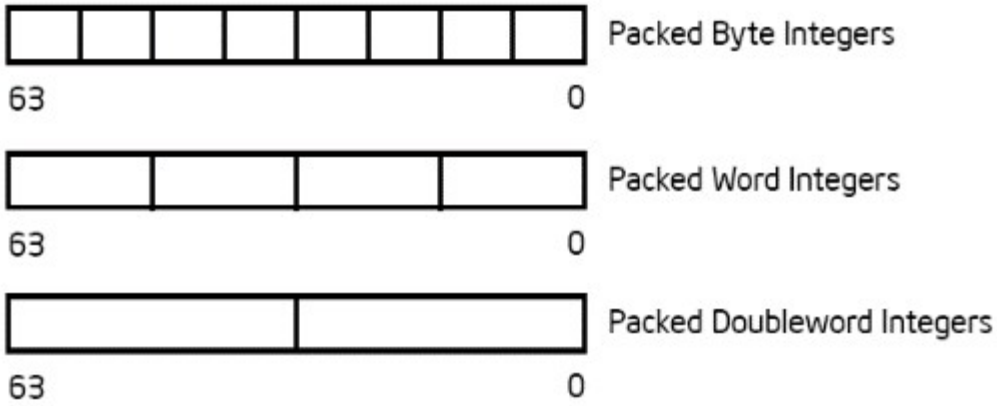
MMX

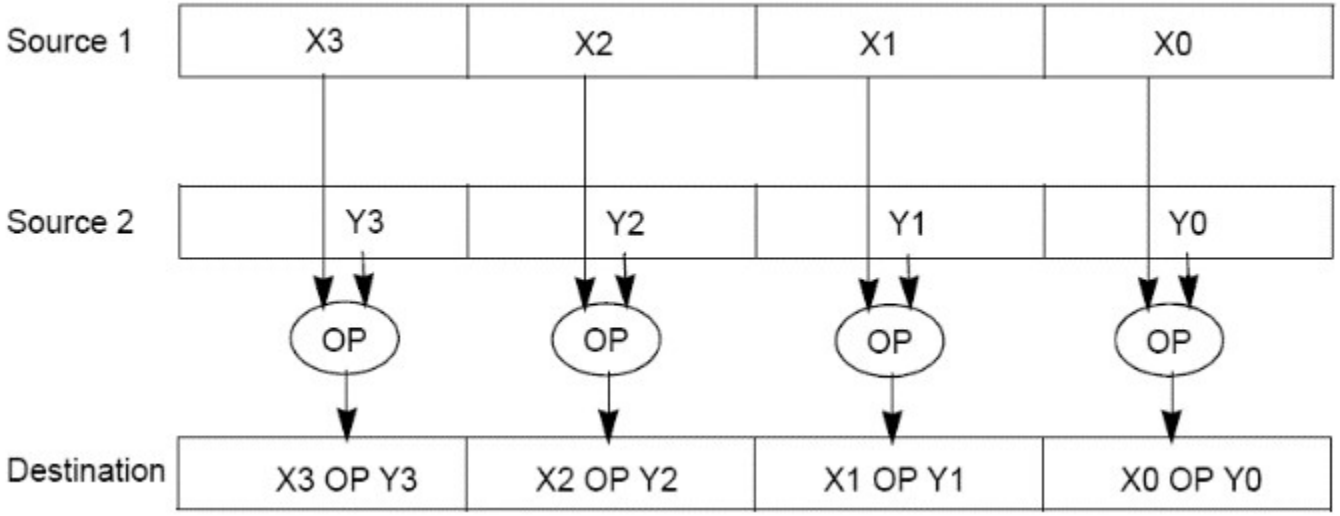
- MMX introduced
  - Eight new 64-bit data registers, called MMX registers
  - Three new packed data types:
    - 64-bit packed byte integers (signe and unsigned)
    - 64-bit packed word integers (signed and unsigned)
    - 64-bit packed double word integers (signed and unsigned)
  - Instructions that support the new data types



MMX

- Packed integer types allow operations to be applied on multiple ir

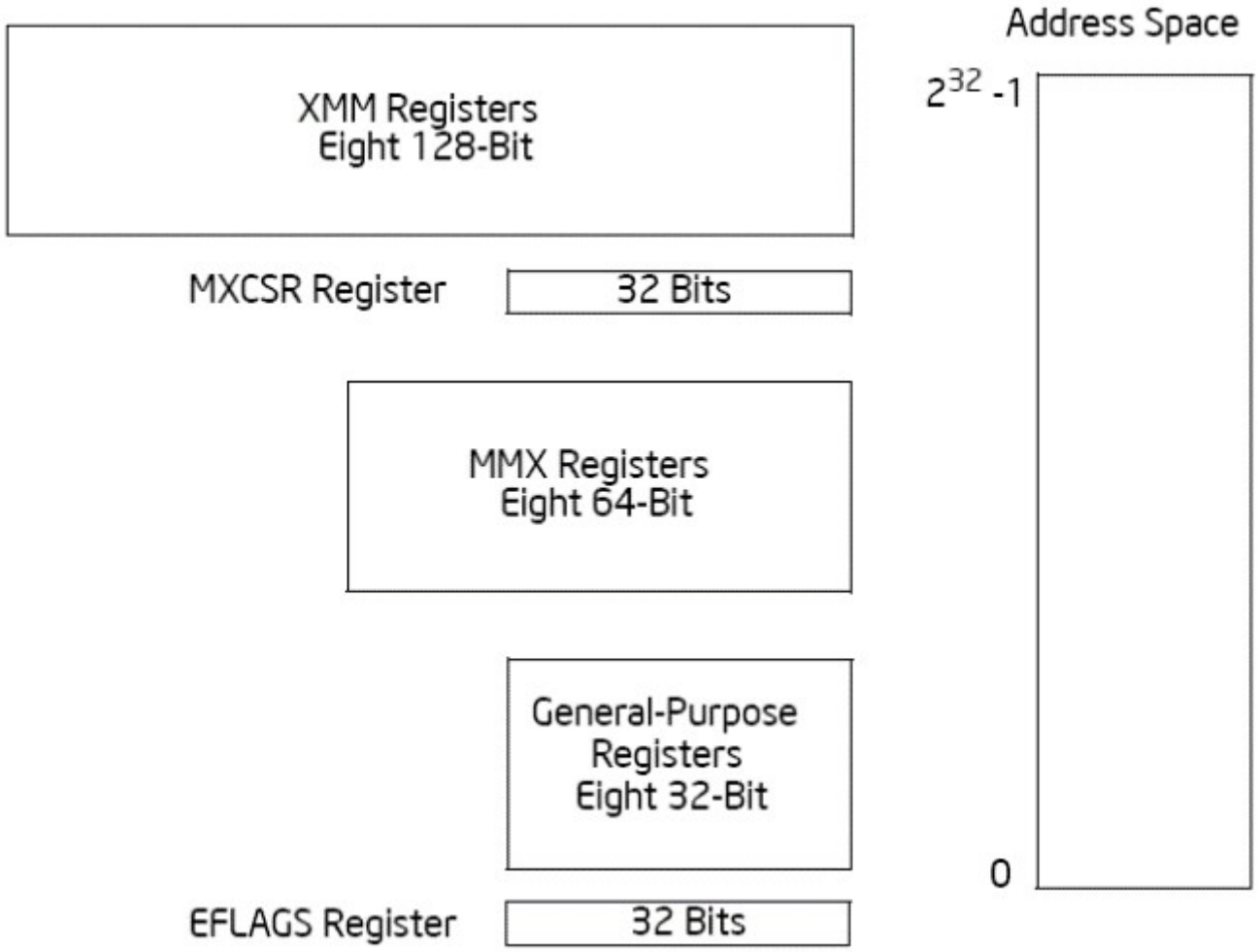




SSE

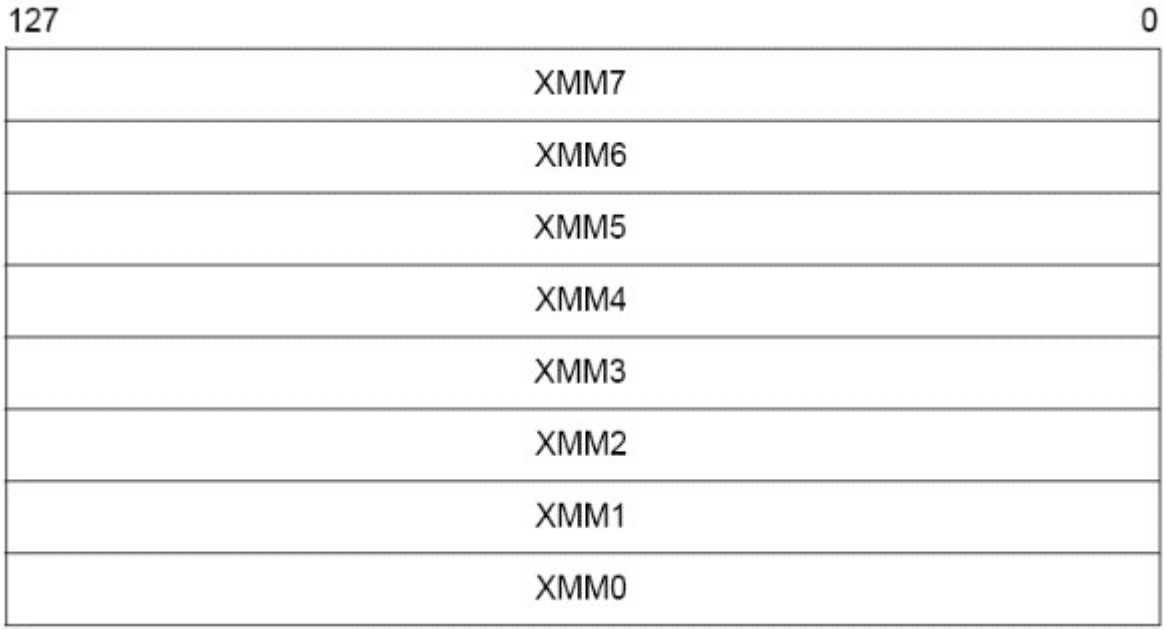
- SSE introduced eight 128-bit data registers (called XMM registers)
  - In 64-bit modes, they are available as 16 64-bit registers
  - The 128-bit packed single-precision floating-point data type, which allows four single-precision operations to be performed simultaneously
    - They can be used in parallel with MMX registers

SSE Execution Environment



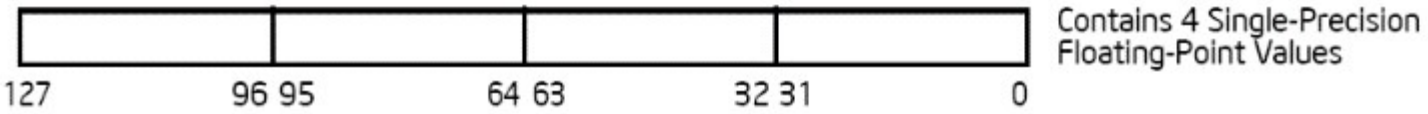
XMM Registers

- In certain modes, additional eight 64 bit registers are also available (XMM8 - XMM15)

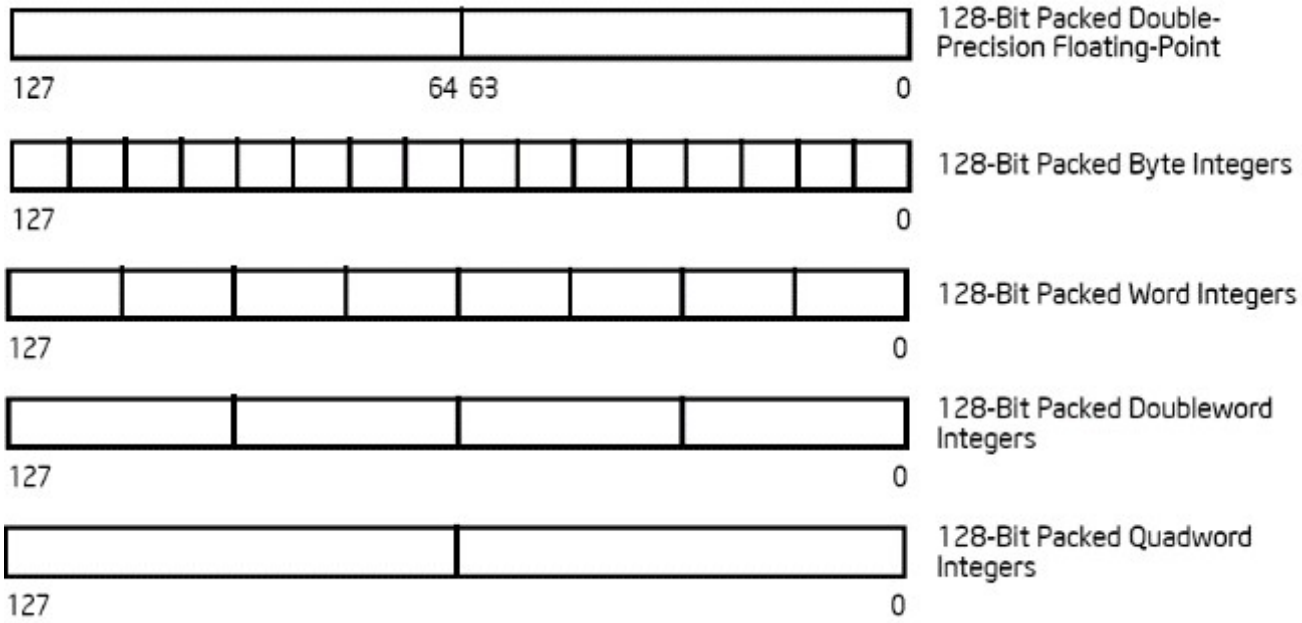


SSE Data Type

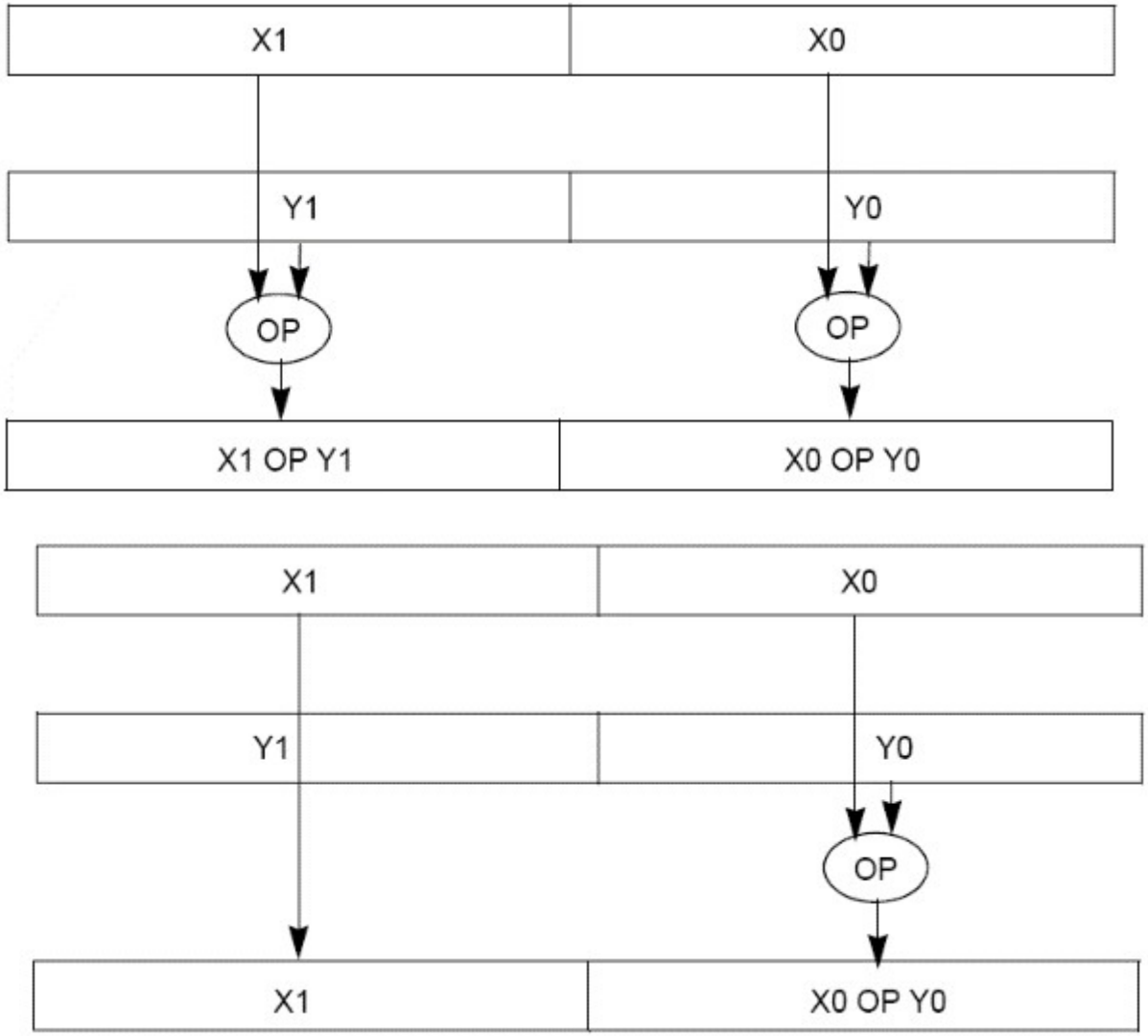
- SSE extensions introduced one new data type
  - 128-Bit Packed Single-Precision Floating-Point Data Type



  - SSE 2 introduced five data types



Packed and Scalar Double-Precision Floating-Point Operations



SSE instructions

- Assembly instructions
  - Data movement instructions
    - moving data in and out of vector registers
  - Arithmetic instructions
    - Arithmetic operation on multiple data (2 doubles, 4 floats, 16 bytes, etc)
  - Logical instructions
    - Logical operation on multiple data
  - Comparison instructions
    - Comparing multiple data
  - Shuffle instructions
    - move data around SIMD registers
  - Miscellaneous
    - Data conversion: between x86 and SIMD registers
    - Cache control: vector may pollute the caches
    - State management:

SSE Instructions

- SSE Data Transfer Instructions

MOVAPS	Move four aligned packed single-precision floating-point values between XMM registers or between and XMM register and memory
MOVUPS	Move four unaligned packed single-precision floating-point values between XMM registers or between and XMM register and memory
MOVHPS	Move two packed single-precision floating-point values to an from the high quadword of an XMM register and memory
MOVHLPS	Move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register
MOVLPS	Move two packed single-precision floating-point values to an from the low quadword of an XMM register and memory
MOVLHPS	Move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register
MOVMSKPS	Extract sign mask from four packed single-precision floating-point values
MOVSS	Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory

SSE instructions

- Arithmetic instructions
  - pd: two doubles, ps: 4 floats, ss: scalar
  - ADD, SUB, MUL, DIV, SQRT, MAX, MIN, RCP, etc
    - ADDPS – add four floats, ADDSS: scalar add
- Logical instructions
  - AND, OR, XOR, ANDN, etc
    - ANDPS – bitwise AND of operands
    - ANDNPS – bitwise AND NOT of operands
- Comparison instruction:
  - CMPPS, CMPSS – compare operands and return all 1's or 0's

SSE Packed Arithmetic Instructions

ADDPS	Add packed single-precision floating-point values
ADDSS	Add scalar single-precision floating-point values
SUBPS	Subtract packed single-precision floating-point values
SUBSS	Subtract scalar single-precision floating-point values
MULPS	Multiply packed single-precision floating-point values
MULSS	Multiply scalar single-precision floating-point values
DIVPS	Divide packed single-precision floating-point values
DIVSS	Divide scalar single-precision floating-point values
RCPPS	Compute reciprocals of packed single-precision floating-point values
RCPSS	Compute reciprocal of scalar single-precision floating-point values

SSE Packed Arithmetic Instructions

SQRTPS	Compute square roots of packed single-precision floating-point values
SQRTSS	Compute square root of scalar single-precision floating-point values
RSQRTPS	Compute reciprocals of square roots of packed single-precision floating-point values
RSQRTSS	Compute reciprocal of square root of scalar single-precision floating-point values
MAXPS	Return maximum packed single-precision floating-point values
MAXSS	Return maximum scalar single-precision floating-point values
MINPS	Return minimum packed single-precision floating-point values
MINSS	Return minimum scalar single-precision floating-point values

SSE instructions

- Shuffle instructions
  - SHUFPS: shuffle number from one operand to another
  - UNPCKHPS - Unpack high order numbers to an SIMD register. Unpckhps [x4,x3,x2,x1][y4,y3,y2,y1] = [y4, x4, y3, x3]
  - UNPCKLPS
- Other
  - Data conversion: CVTPS2PI mm,xmm/mem64
  - Cache control
    - MOVNTPS stores data from a SIMD floating-point register to memory, bypass cache.
  - State management: LDMXCSR load MXCSR status register.

SSE Comparison, Logical, and Shuffle Instructions

CMPPS	Compare packed single-precision floating-point values
CMPSS	Compare scalar single-precision floating-point values
COMISS	Perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register
UCOMISS	Perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register
ANDPS	Perform bitwise logical AND of packed single-precision floating-point values
ANDNPS	Perform bitwise logical AND NOT of packed single-precision floating-point values
ORPS	Perform bitwise logical OR of packed single-precision floating-point values
XORPS	Perform bitwise logical XOR of packed single-precision floating-point values
SHUFPS	Shuffles values in packed single-precision floating-point operands
UNPCKHPS	Unpacks and interleaves the two high-order values from two single-precision floating-point operands
UNPCKLPS	Unpacks and interleaves the two low-order values from two single-precision floating-point operands

SSE2 Instructions

ADDPD	Add packed double-precision floating-point values
ADDSD	Add scalar double precision floating-point values
SUBPD	Subtract scalar double-precision floating-point values
SUBSD	Subtract scalar double-precision floating-point values
MULPD	Multiply packed double-precision floating-point values
MULSD	Multiply scalar double-precision floating-point values
DIVPD	Divide packed double-precision floating-point values
DIVSD	Divide scalar double-precision floating-point values
SQRTPD	Compute packed square roots of packed double-precision floating-point values
SQRTSD	Compute scalar square root of scalar double-precision floating-point values
MAXPD	Return maximum packed double-precision floating-point values
MAXSD	Return maximum scalar double-precision floating-point values
MINPD	Return minimum packed double-precision floating-point values
MINSD	Return minimum scalar double-precision floating-point values

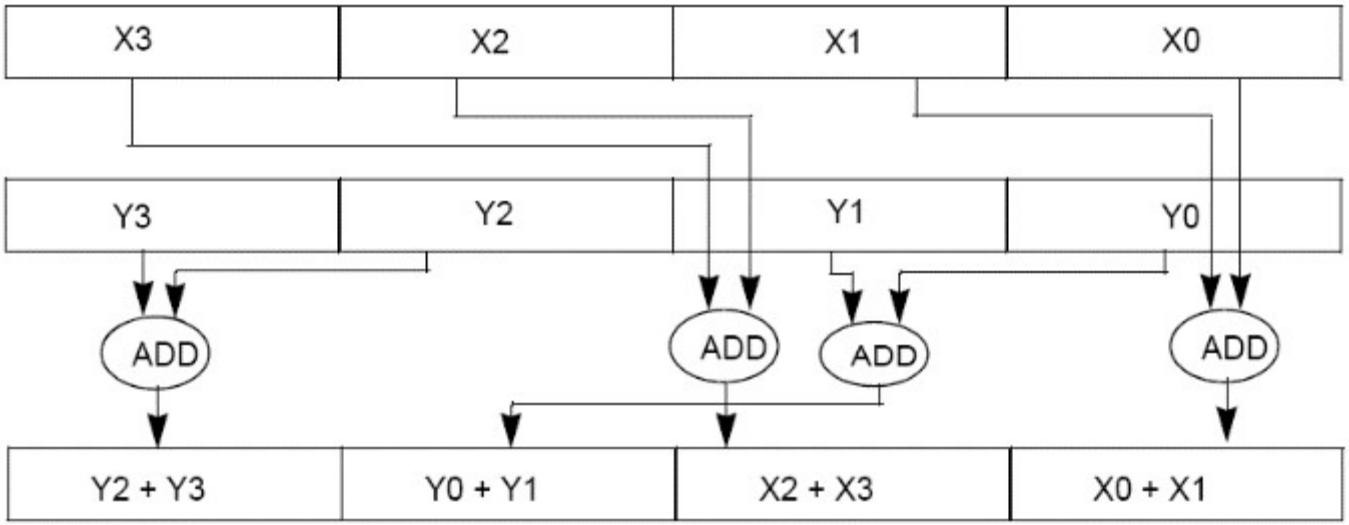
SSE2 128-Bit SIMD Integer Instructions

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword
MOVQ2DQ	Move quadword integer from MMX to XMM registers
MOVDQ2Q	Move quadword integer from XMM to MMX registers
PMULUDQ	Multiply packed unsigned doubleword integers
PADDQ	Add packed quadword integers
PSUBQ	Subtract packed quadword integers
PSHUFLW	Shuffle packed low words
PSHUFHW	Shuffle packed high words
PSHUFD	Shuffle packed doublewords
PSLLDQ	Shift double quadword left logical
PSRLDQ	Shift double quadword right logical
PUNPCKHQDQ	Unpack high quadwords
PUNPCKLQDQ	Unpack low quadwords

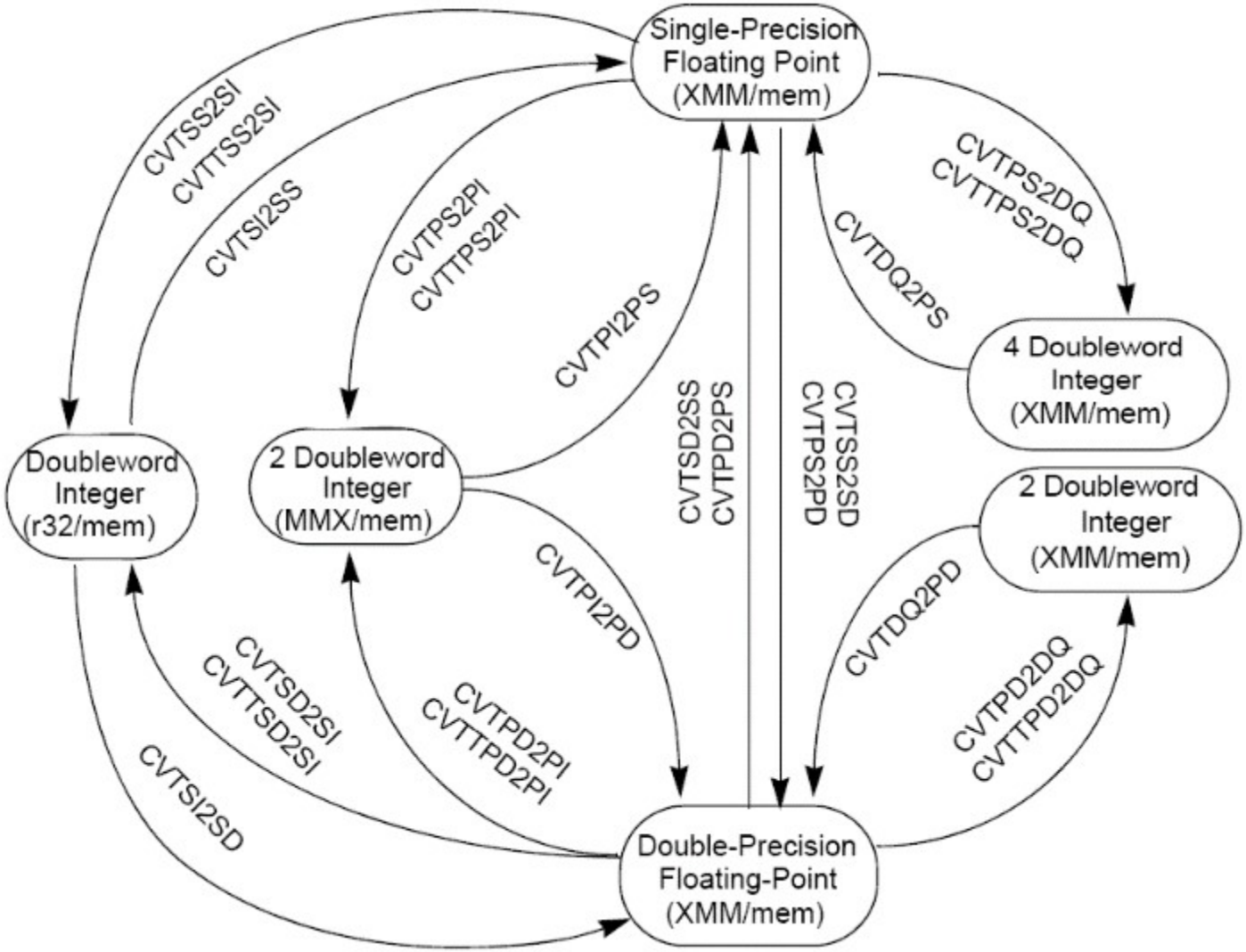
Horizontal Addition/Subtraction

PHADDW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
PHADDSW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD	Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.
PHSUBW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.
PHSUBSW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
PHSUBD	Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

Horizontal Data Movements



Conversion Between Different Types



SEE programming in C/C++

- Map to *intrinsics*
  - An *intrinsic* is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsic functions are inherently more efficient than called functions because no calling linkage is required.
  - Intrinsics provides a C/C++ interface to use processor-specific enhancements
  - Supported by major compilers such as gcc

SSE intrinsics

- Header files to access SEE intrinsics
  - #include <mmintrin.h> // MMX
  - #include <xmmintrin.h> // SSE
  - #include <emmintrin.h> //SSE2
  - #include <pmmintrin.h> //SSE3
  - #include <tmmintrin.h> //SSSE3
  - #include <smmintrin.h> // SSE4
- MMX/SSE/SSE2 are mostly supported
- SSE4 are not well supported.
- When compile, use –msse, -mmmx, -msse2 (machine dependent code)
  - Some are default for gcc.
- A side note:
  - Gcc default include path can be seen by ‘cpp –v’
  - On linprog, the SSE header files are in
    - /usr/local/lib/gcc/x86_64-unknown-linux-gnu/4.3.2/include/

SSE intrinsics

- Data types (mapped to an xmm register)
  - __m128: float
  - __m128d: double
  - __m128i: integer
- Data movement and initialization
  - _mm_load_ps, _mm_loadu_ps, _mm_load_pd, _mm_loadu_pd, etc
  - _mm_store_ps, …
  - _mm_setzero_ps

SSE intrinsics

- Data types (mapped to an xmm register)
  - __m128: float
  - __m128d: double
  - __m128i: integer
- Data movement and initialization
  - _mm_load_ps, _mm_loadu_ps, _mm_load_pd, _mm_loadu_pd, etc
  - _mm_store_ps, …
  - _mm_setzero_ps
  - _mm_loadl_pd, _mm_loadh_pd
  - _mm_storel_pd, _mm_storeh_pd

SSE intrinsics

- Arithemetic intrinsics:
  - _mm_add_ss, _mm_add_ps, …
  - _mm_add_pd, _mm_mul_pd
- See examples

SSE intrinsics

- Data alignment issue
  - Some intrinsics may require memory to be aligned to 16 bytes.
    - May not work when memory is not aligned.
- Writing more generic SSE routine
  - Check memory alignment
  - Slow path may not have any performance benefit with SSE

GCC Inline Assembly

- GCC inline assembly allows us to insert inline functions written in assembly
  - GCC provides the utility to specify input and output operands as C variables
  - Basic inline

```
asm("movl %ecx %eax"); /* moves the contents of ecx to eax */
__asm__("movb %bh (%eax)"); /*moves the byte from bh
                              to the memory pointed by eax */
```

```
asm ( assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

GCC Inline Assembly

- Some examples

```
int a=10, b;
asm ("movl %1, %%eax;
      movl %%eax, %0;"
     :"=r"(b)          /* output */
     :"r"(a)           /* input */
     :"%eax"           /* clobbered register */
     );

int main(void)
{
        int foo = 10, bar = 15;
        __asm__ __volatile__("addl  %%ebx,%%eax"
                             :"=a"(foo)
                             :"a"(foo), "b"(bar)
                             );
        printf("foo+bar=%d\n", foo);
        return 0;
}
```

GCC Inline Assembly

```
static inline char * strcpy(char * dest,const char *src)
{
int d0, d1, d2;
__asm__ __volatile__(  "1:\tlodsb\n\t"
                       "stosb\n\t"
                       "testb %%al,%%al\n\t"
                       "jne 1b"
                     : "=&S" (d0), "=&D" (d1), "=&a" (d2)
                     : "0" (src),"1" (dest)
                     : "memory");
return dest;
}
```

GCC Inline Assembly

```
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile (  "int $0x80" \
                  : "=a" (__res) \
                  : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                    "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}
```

```
asm("movl $1,%%eax;          /* SYS_exit is 1 */
     xorl %%ebx,%%ebx;       /* Argument is in ebx, it is 0 */
     int  $0x80"             /* Enter kernel mode */
     );
```

Example: Inner Product

- The inner product of two vectors x = (x1 , x2 , . . . , xk) and y = (y1 , y2 , . . . , yk) is y = x1 y1 + x2 y2 + . . . + xk yk .

- *float x[k]; float y[k]; // vectors of length k*
- *float inner_prod = 0.0;*
- *for(i = 0; i < k; i++)*
  - *inner_prod += x[i] * y[i];*

Example: Inner Product in SSE

- float x[k]; float y[k]; // vectors of length k
- __m128 X, Y; // 128-bit values
- __m128 acc = _mm_setzero_ps(); // set to (0, 0, 0, 0)
- float inner_prod, temp[4];
- for(i = 0; i < k - 4; i += 4) {
  - X = _mm_load_ps(&x[i]); // load chunk of 4 floats
  - Y = _mm_load_ps(y + i); // alternate way, pointer arithmetic
  - acc = _mm_add_ps(acc, _mm_mul_ps(X, Y));
- }
- _mm_store_ps(&temp[0], acc); // store acc into an array of floats
- inner_prod = temp[0] + temp[1] + temp[2] + temp[3];
- // add the remaining values
- for(; i < k; i++)
  - inner_prod += x[i] * y[i];