Maynooth University
National University of Ireland Maynooth

CS433 Modern Architectures

Video 4

Modern Instructions sets and floating point

Microprocessor development cycle

Intel Tick–tock (2007)

tick: process technology change
tock: microarchitecture change

(2016)

Process: die shrink
Architecture: change code processed
Optimization: refinements

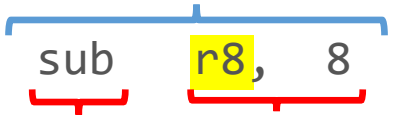
3Yr cycle

Tick	2014 Broadwell	Move from 22nm (optical lithography)to 14nm technology
Tock	2015 Skylake	Microarchitecture developed to reduced power for mobile apps

Process	2018 Cannon Lake	14nm-10nm
Architecture	2019 Willow cove	Development of the microarchitecture
Process	2020 Tiger lake	Refinement of 10nm process

Anatomy of an assembly language instruction (general purpose registers)

Assembly Language



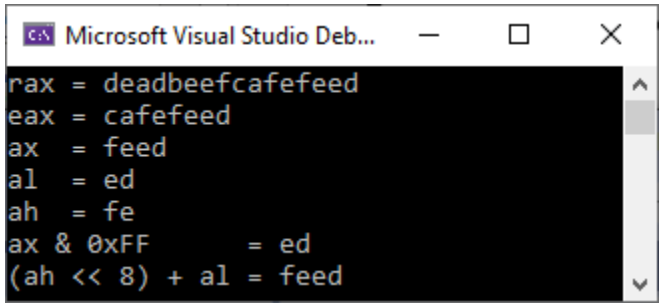
Operator Operands

	(63)MSB								(31)				(15)		(7) LSB(0)	
64-bit	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000 RAX
32-bit									0000	0000	0000	0000	0000	0000	0000	0000 EAX
16-bit													0000	0000	0000	0000 AX
8-bit															0000	0000 AL
8-bit															0000	0000 AH

#	Reg	64-bit register	Lower 32 bits	Lower 16 bits	Upper 8 bits	Lower 8 bits	Name
0	0000	rax	eax	ax	ah	al	Accumulator
1	0001	rbx	ebx	bx	bh	bl	Base
2	0010	rcx	ecx	cx	ch	cl	Counter
3	0011	rdx	edx	dx	dh	dl	Data
4	0100	rsi	esi	si	sih	sil	Source
5	0101	rdi	edi	di	dih	dil	Destination
6	0110	rbp	ebp	bp	dph	bpl	Base pointer
7	0111	rsp	esp	sp	sph	spl	Stack pointer
8	1000	r8	r8d	r8w		r8b	register 8
9	1001	r9	r9d	r9w		r9b	
10	1010	r10	r10d	r10w		r10b	
11	1011	r11	r11d	r11w		r11b	
12	1100	r12	r12d	r12w		r12b	
13	1101	r13	r13d	r13w		r13b	
14	1110	r14	r14d	r14w		r14b	
15	1111	r15	r15d	r15w		r15b	register 15

al, ah, ax, eax and rax share common bits, similar to a union in C.

```cpp
// https://stackoverflow.com/questions/27336734/how-to-tie-variables-in-c
#include <iostream>
#include <cstdint>
using namespace std;

union reg_t
{
    uint64_t rx;
    uint32_t ex;
    uint16_t x;
    struct {
        uint8_t l;
        uint8_t h;
    };
};

int main()
{
    reg_t a;

    a.rx = 0xdeadbeefcafefeed;

    cout << "rax = " << hex << a.rx << endl;
    cout << "eax = " << hex << a.ex << endl;
    cout << "ax  = " << hex << a.x << endl;
    cout << "al  = " << hex << (uint16_t)a.l << endl;
    cout << "ah  = " << hex << (uint16_t)a.h << endl;
    cout << "ax & 0xFF      = " << hex << (a.x & 0xFF) << endl;
    cout << "(ah << 8) + al = " << hex << (a.h << 8) + a.l << endl;
}
```



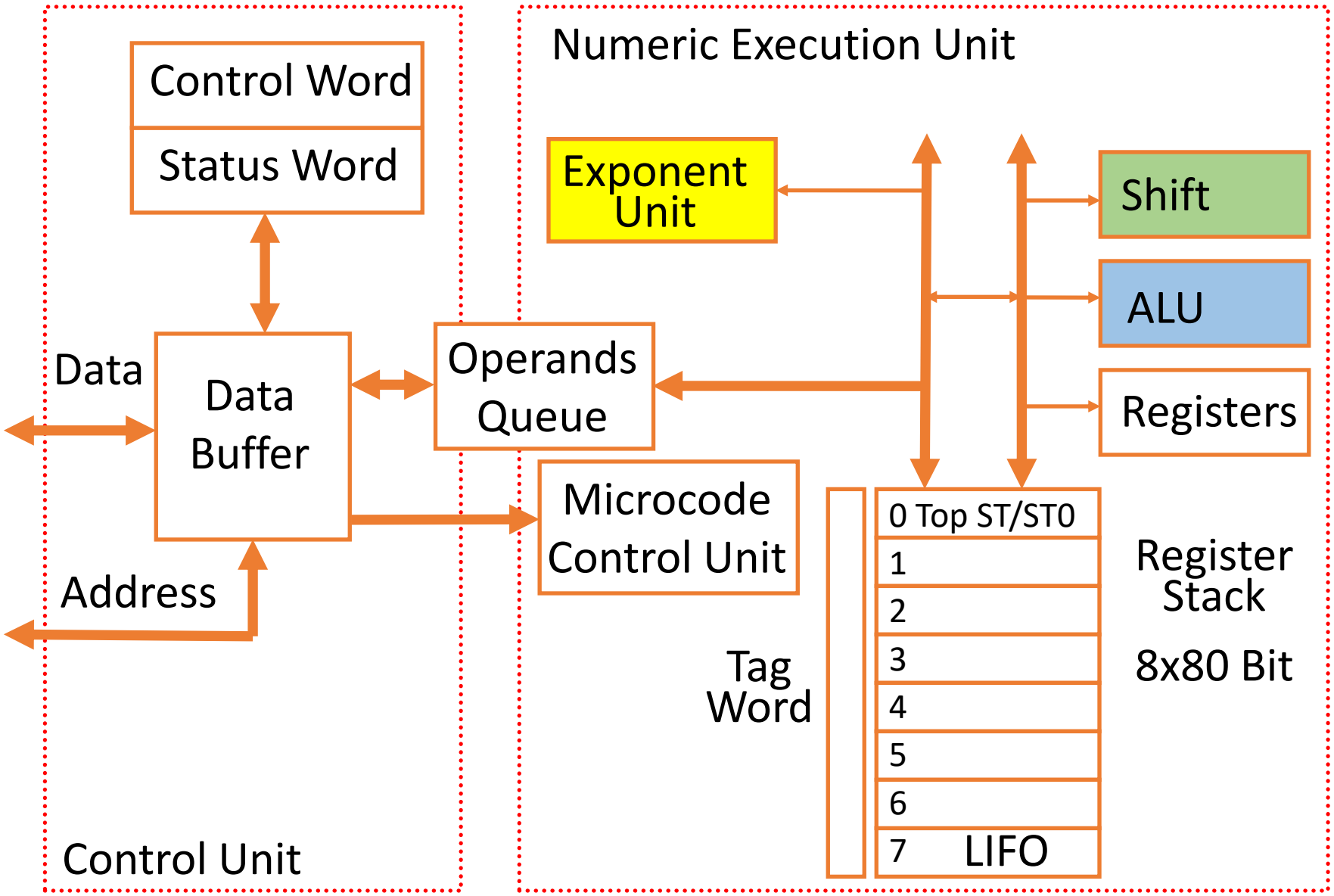
General additional registers

8-bit GP``	16-bit GP	32-bit GP	64-bit GP	80-bit x87	64-bit MMX	128-bit XMM	256-bit YMM	512-bit YMM	16-bit Segment	32-bit Control	32-bit Debug
AL	AX	EAX	RAX	ST0	MMX0	XMM0	YMM0	ZMM0	ES	CR0	DR0
CL	CX	ECX	RCX	ST1	MMX1	XMM1	YMM1	ZMM1	CS	CR1	DR1
DL	DX	EDX	RDX	ST2	MMX2	XMM2	YMM2	ZMM2	SS	CR2	DR2
BL	BX	EBX	RBX	ST3	MMX3	XMM3	YMM3	ZMM3	DS	CR3	DR3
AH, SPL1	SP	ESP	RSP	ST4	MMX4	XMM4	YMM4	ZMM4	FS	CR4	DR4
CH, BPL1	BP	EBP	RBP	ST5	MMX5	XMM5	YMM5	ZMM5	GS	CR5	DR5
DH, SIL1	SI	ESI	RSI	ST6	MMX6	XMM6	YMM6	ZMM6	-	CR6	DR6
BH, DIL1	DI	EDI	RDI	ST7	MMX7	XMM7	YMM7	ZMM7	-	CR7	DR7
R8L	R8W	R8D	R8	`	-	XMM8	YMM8	ZMM8	ES	CR8	DR8
R9L	R9W	R9D	R9	-	-	XMM9	YMM9	ZMM9	CS	CR9	DR9
R10L	R10W	R10D	R10	-	-	XMM10	YMM10	ZMM10	SS	CR10	DR10
R11L	R11W	R11D	R11	-	-	XMM11	YMM11	ZMM11	DS	CR11	DR11
R12L	R12W	R12D	R12	-	-	XMM12	YMM12	ZMM12	FS	CR12	DR12
R13L	R13W	R13D	R13	-	-	XMM13	YMM13	ZMM13	GS	CR13	DR13
R14L	R14W	R14D	R14	-	-	XMM14	YMM14	ZMM14	-	CR14	DR14
R15L	R15W	R15D	R15	-	-	XMM15	YMM15	ZMM15	-	CR15	DR15
						XMM16-31	YMM16-31	ZMM16-31			

x87	1987 (80387) Floating point extension to x86 processors
MMX	1997 SIMD Only integer operations (MMX)
SSE	1999 Streaming SIMD Extensions Integer and floating point (XMM)
AVX	2011 Advanced Vector Extensions (YMM0) floating point operations
AVX512	2013 512 bit Advanced Vector Extensions

SSE2

x87 Architecture



Control word: configures calculator
Status word: returns information about calculations (e.g. errors)

Register stack: 80 bit 64 significand
3 bit stack pointer SP
SP=0 on start up ST(0)
SP wraps around 7->0

$1.0001 \times 2^4 = 10001$ (17)
$1.1 \times 2^{-1} = 0.11$ (0.625)

Add
10001.11 (aligned significands)

Multiply (17x0.625=10.625)
ALU: 1.0001x1.1=1.10011
Exponent unit: 4+(-1)=3
$1.10011 \times 2^3 = 1100.11$ (10.625 QED)

x87FPU Operators/Instructions (all CPU's 486 and after)

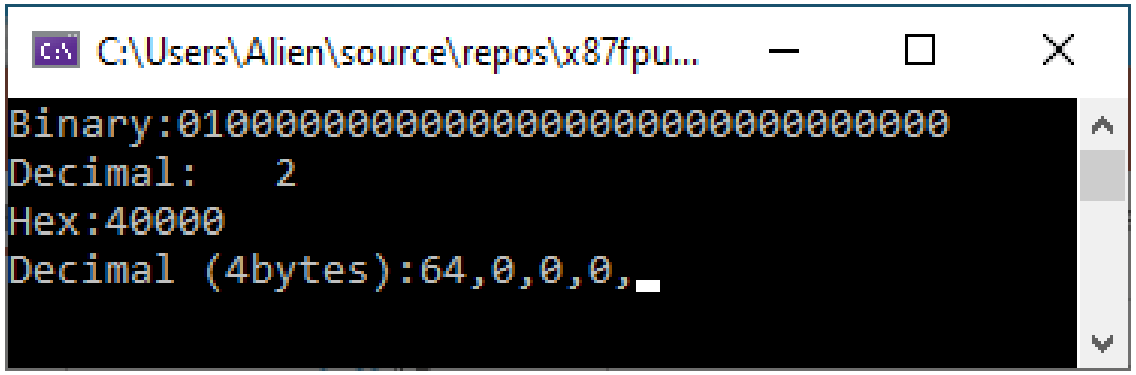
x87fpu	
FABS	Absolute Value
FADD	Add
FADDP	Add and Pop
FBLD	Load Binary Coded Decimal
FBSTP	Store BCD Integer and Pop
FCHS	Change Sign
FCLEX	Clear Exceptions
FCMOVB	FP Conditional Move - below (CF=1)
FCMOVBE	FP Conditional Move - below or equal (CF=1 or ZF=1)
FCMOVE	FP Conditional Move - equal (ZF=1)
FCMOVNB	FP Conditional Move - not below (CF=0)
FCMOVNBE	FP Conditional Move - below or equal (CF=0 and ZF=0)
FCMOVNE	FP Conditional Move - not equal (ZF=0)
FCMOVNU	FP Conditional Move - not unordered (PF=0)
FCMOVU	FP Conditional Move - unordered (PF=1)
FCOM	Compare Real
FCOM2	Compare Real
FCOMI	Compare Floating Point Values and Set EFLAGS
FCOMIP	Compare Floating Point Values and Set EFLAGS and Pop
FCOMP	Compare Real and Pop
FCOMP3	Compare Real and Pop
FCOMP5	Compare Real and Pop
FCOMPP	Compare Real and Pop Twice
FCOS	Cosine
FDECSTP	Decrement Stack-Top Pointer
FDIV	Divide
FDIVP	Divide and Pop
FDIVR	Reverse Divide
FDIVRP	Reverse Divide and Pop
FFREE	Free Floating-Point Register
FFREEP	Free Floating-Point Register and Pop
FIADD	Add
FICOM	Compare Integer
FICOMP	Compare Integer and Pop
FIDIV	Divide

FIDIVR	Reverse Divide
FILD	Load Integer
FIMUL	Multiply
FINCSTP	Increment Stack-Top Pointer
FINIT	Initialize Floating-Point Unit
FIST	Store Integer
FISTP	Store Integer and Pop
FISTTP	Store Integer with Truncation and Pop
FISUB	Subtract
FISUBR	Reverse Subtract
FLD	Load Floating Point Value
FLD1	Load Constant +1.0
FLDCW	Load x87 FPU Control Word
FLDENV	Load x87 FPU Environment
FLDL2E	Load Constant log2e
FLDL2T	Load Constant log210
FLDLG2	Load Constant log102
FLDLN2	Load Constant loge2
FLDPI	Load Constant π
FLDZ	Load Constant +0.0
FMUL	Multiply
FMULP	Multiply and Pop
FNCLEX	Clear Exceptions
FNINIT	Initialize Floating-Point Unit
FNOP	No Operation
FNSAVE	Store x87 FPU State
FNSTCW	Store x87 FPU Control Word
FNSTENV	Store x87 FPU Environment
FNSTSW	Store x87 FPU Status Word
FPATAN	Partial Arctangent and Pop
FPREM	Partial Remainder (for compatibility with i8087 and i287)
FPREM1	IEEE Partial Remainder
FPTAN	Partial Tangent
FRNDINT	Round to Integer
FRSTOR	Restore x87 FPU State

FSAVE	Store x87 FPU State
FSCALE	Scale
FSIN	Sine
FSINCOS	Sine and Cosine
FSQRT	Square Root
FST	Store Floating Point Value
FSTCW	Store x87 FPU Control Word
FSTENV	Store x87 FPU Environment
FSTP	Store Floating Point Value and Pop
FSTP1	Store Floating Point Value and Pop
FSTP8	Store Floating Point Value and Pop
FSTP9	Store Floating Point Value and Pop
FSTSW	Store x87 FPU Status Word
FSUB	Subtract
FSUBP	Subtract and Pop
FSUBR	Reverse Subtract
FSUBRP	Reverse Subtract and Pop
FTST	Test
FUCOM	Unordered Compare Floating Point Values
FUCOMI	Unordered Compare Floating Point Values and Set EFLAGS
FUCOMIP	Unordered Compare Floating Point Values and Set EFLAGS and Pop
FUCOMP	Unordered Compare Floating Point Values and Pop
FUCOMPP	Unordered Compare Floating Point Values and Pop Twice
FWAIT	Check pending unmasked floating-point exceptions
FXAM	Examine
FXCH	Exchange Register Contents
FXCH4	Exchange Register Contents
FXCH7	Exchange Register Contents
FXTRACT	Extract Exponent and Significand
FXRSTOR	Restore x87 FPU, MMX, XMM, and MXCSR State
FXSAVE	Save x87 FPU, MMX, XMM, and MXCSR State
FYL2X	Compute y × log2x and Pop
FYL2XP1	Compute y × log2(x+1) and Pop
WAIT	Check floating-point exceptions

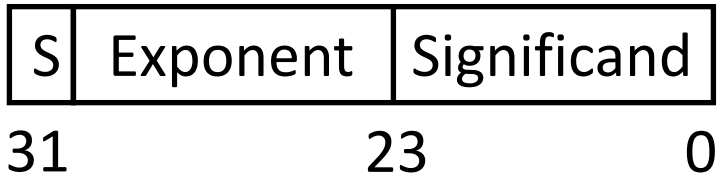
X87 Example 1 – Short real (floats)

```
#include <iostream>
#include <stdio.h>

void test(void);
int main()
{
    test();
    return 0;
}

void test()
{
    float A = 2, C = 0;
    unsigned short cntrl = 0x3FF, stat;
    __asm
    {
        FINIT
        FLDCW cntrl     ; Round even, Mask Interrupts
        FLD A           ; Push SX onto FP stack
        FSTP C          ; Copy result from stack C
    }
```




```
    // Binary representation of the 4 bytes
    printf("Binary:");
    unsigned char byt;
    for (int x = 3; x >= 0; x--)
    {
        byt = *((unsigned char*)&C + x);
        for (int y = 128; y > 0; y /= 2)
        {
            if ((y & byt) == 0) printf("0"); else printf("1");
        }
    }
    // Decimal format
    printf("\nDecimal: %3.0f", C);
    // Hex format
    printf("\nHex:");
    for (int x = 3; x >= 0; x--)
    {
        byt = *((unsigned char*)&C + x);

        printf("%x", (unsigned int)byt);
    }
    // Decimal 4 byte format
    printf("\nDecimal (4bytes):");
    for (int x = 3; x >= 0; x--)
    {
        byt = *((unsigned char*)&C + x);
        printf("%d,", (unsigned int)byt);
    }
    while (getchar() != 10);
}
```

Decoding floats



Decimal range $-1.2 \times 10^{-38} < |X| < 3.4 \times 10^{38}$

S = Sign, 0 +ve, 1 –ve
M = Mantissa/Significand (23 bits)
R = Base (2)
E = Exponent (8 bits)
O = Offset (-127)

float 32 bits

1 01111111 10000000000000000000000

S 1 -> -ve
E 01111111 -> 127 - 127(offset) -> E = 0
M 1 10000 -> Signifcicand =1.50…

$$Value = S.MR^{E-O} = -1.5 * 2^{127-127} = -1.5$$

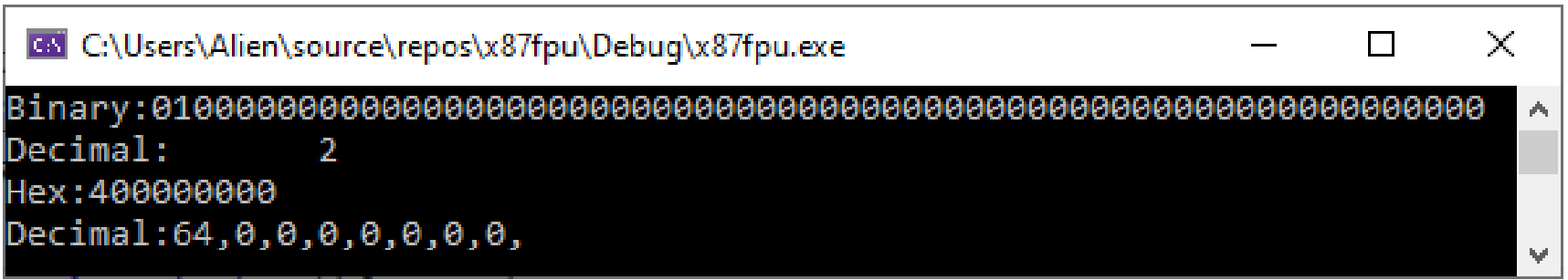
Internally the x87 uses 80bit double-extended precision floating point
S(1bit), E(15bits), M(64bits)
The 1. is included in the internal coding (it is not added back as above)
FLD loads external 32/64 bit numbers so not easy to access all 80 bits

X87 Example 2 – Long real (double)

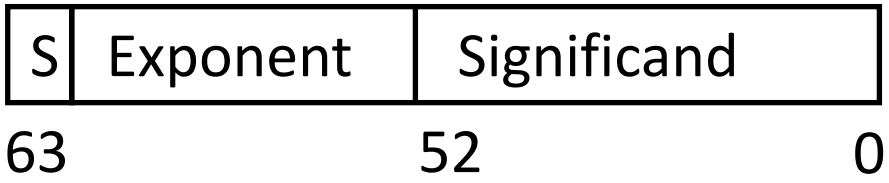
```
#include <iostream>
#include <stdio.h>

void test(void);
int main()
{
    test();
    return 0;
}

void test()
{
    double A = 2, C = 0;
    unsigned short cntrl = 0x3FF, stat;
    __asm
    {
        FINIT
        FLDCW cntrl; Round even, Mask Interrupts
        FLD A; Push SX onto FP stack;
        FSTP C; Copy result from stack C
    }
```

```
// Binary representation of the 8 bytes
printf("Binary:");
unsigned char byt;
for (int x = 7; x >= 0; x--)
{
    byt = *((unsigned char*)&C + x);
    for (int y = 128; y > 0; y /= 2)
    {
        if ((y & byt) == 0) printf("0"); else printf("1");
    }
}
// Decimal format
printf("\nDecimal: %7.0f", C);
// Hex format
printf("\nHex:");
for (int x = 7; x >= 0; x--)
{
    byt = *((unsigned char*)&C + x);

    printf("%x", (unsigned int)byt);
}
```

```
// Decimal 8 byte format
printf("\nDecimal:");
for (int x = 7; x >= 0; x--)
{
    byt = *((unsigned char*)&C + x);
    printf("%d,", (unsigned int)byt);
}
while (getchar() != 10);
```



Decoding doubles



Decimal range $-2.3 \times 10^{-308} < |X| < 1.7 \times 10^{308}$

double 64 bits

0 10000000000 0000000000000000000000000000000000000000000000000000

S 0 -> +ve

E 10000000000 -> 1024 - 1023(offset) -> E = 1

M 1 00000 -> Signifcicand =1.00…

$$Value = S.MR^{E-offset} = +1.0 * 2^{1024-1023} = 2$$

Added back (not stored in variable)

Reserved/Special values

0 = all bits set to zero

NaN = indefinite value = largest negative value possible -2^{31}

S = 1 bit

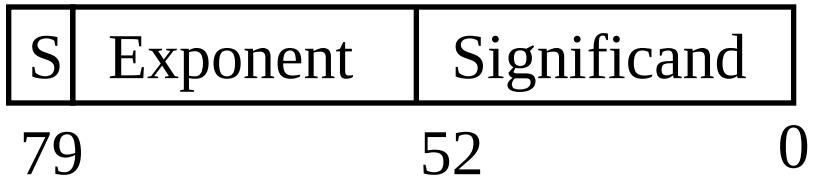
M = 53 bits

R = Base (2)

E = Exponent (11 bits)

O = Offset (-1023)

x87 Internal 80 bit – double extended



The 80 bit is used internally within the x87 FPU to maintain precision when working on 64 bit numbers.

The 80 bit format is not used on more recent architecture developments such as SSE and AVX.

Summary

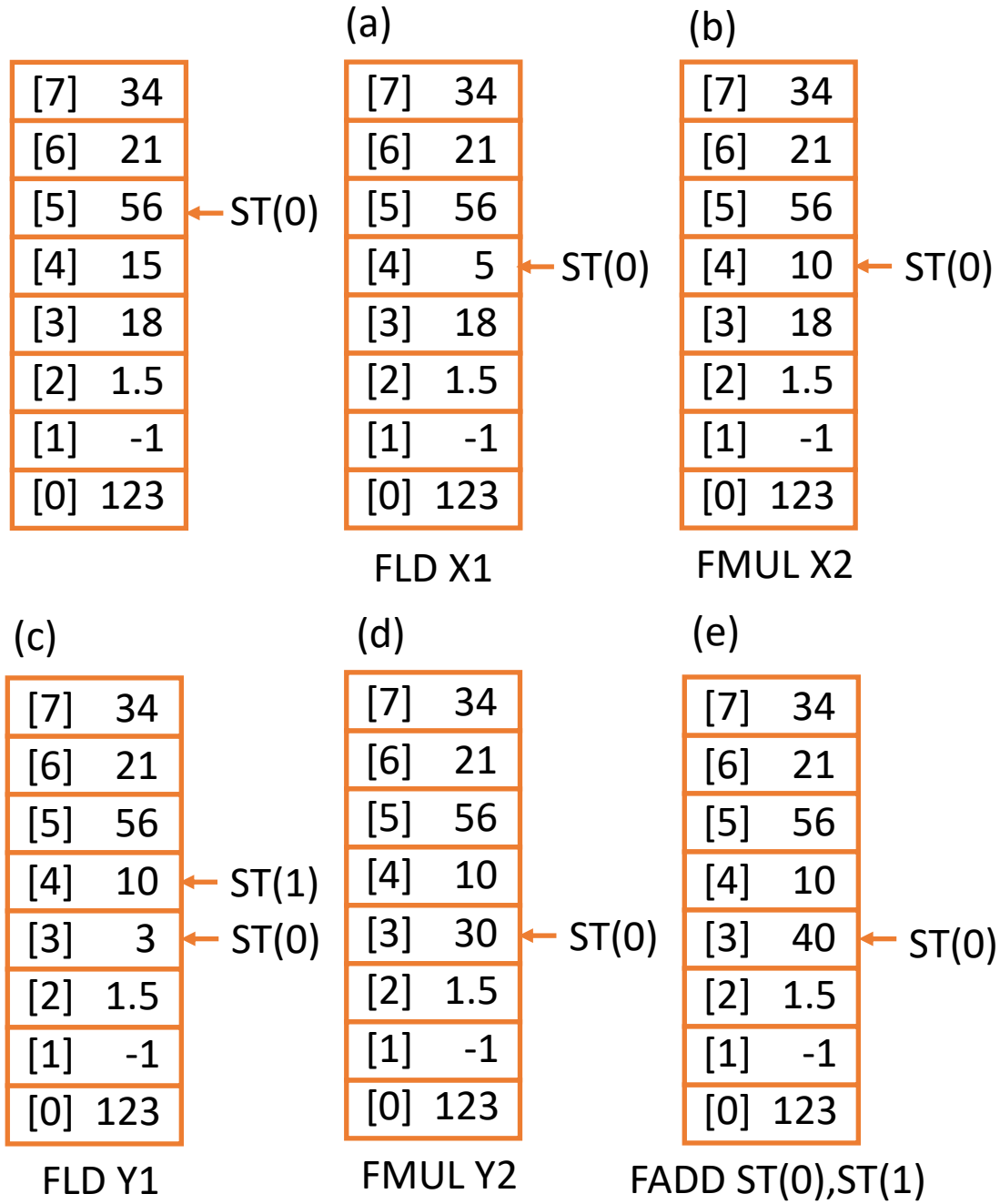
Integer

Fraction

1.001 =1 + 0(1/2) + 0(1/4) + 1(1/8) = 1.0125

Type	Bits	Add back 1.	Base	Sign (bits)	Mantissa (bits)	Exponent (bits)	Offset	Min (decimal)	Max (decimal)	Decimal digits
float	32	Y	2	1	23	8	-127	-1.2x10-38	3.4x10+38	7
double	64	Y	2	1	53	11	-1023	-2.3x10-308	1.7x10+308	16
double extended	80	N	2	1	64	15	-16383	-3.4x10-4932	1.2x10+4932	21
quad double	128	Y	2	1	113	15	-16383	-3.4x10-4932	1.2x10+4932	34

8087 Stack



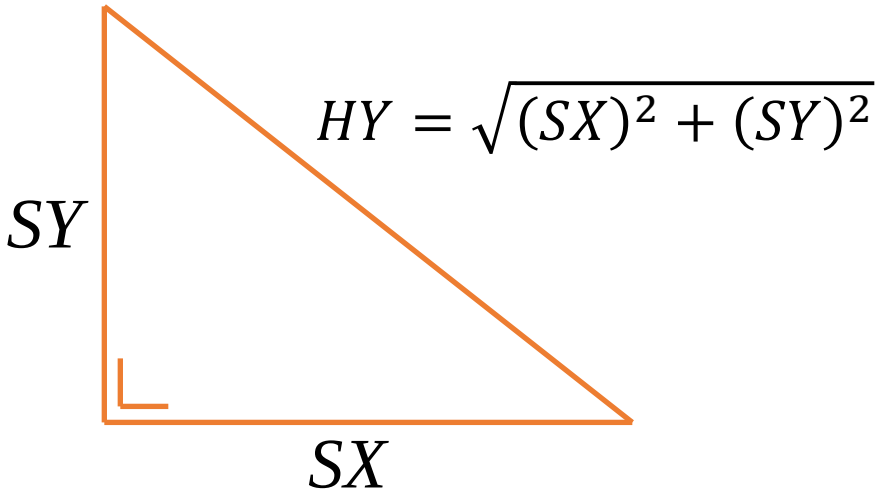

```
void test()
{
    long double X1=5,Y1=3,X2=2,Y2=10 , D = 0;
    unsigned short cntrl = 0x3FF, stat;
    unsigned char byt;

    __asm
    {
        FINIT
        FLDCW cntrl

  (a)   FLD  X1
  (b)   FMUL X2
__(c)   FLD  Y1
  (d)   FMUL Y2
  (e)   FADD ST(0),ST(1)

        FSTP D; Copy result from stack D
    }

    printf("Decimal: %4.0f", D);

    while (getchar() != 10);
}
```

Stack pointer decreases on each push.

The stack can go around the clock.

ST(0) = ST Stack top

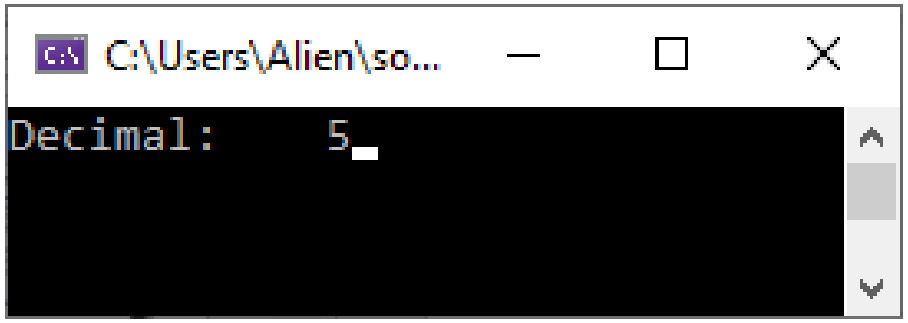
A Pythagorean Problem



$$SX = 3$$

$$SY = 4$$

$$HY = \sqrt{(3)^2 + (4)^2} = 5$$




```
void test()
{
    long double SX = 3, SY = 4, HY;
    unsigned short cntrl = 0x3FF, stat;
    unsigned char byt;
    __asm
    {
        FINIT
        FLDCW cntrl

        FINIT       ; Set FPU to default state
        FLDCW cntrl; Round even, Mask Interrupts

        FLD   SX          ; Push SX onto FP stack
        FMUL  ST, ST(0) ; Multiply ST* ST result on ST
        FLD   SY          ; Push SY onto FP stack
        FMUL  ST, ST(0) ; Multiply ST* ST
        FADD  ST, ST(1) ; ADD top two numbers on stack
        FSQRT             ; Square root number on stack

        FSTP HY           ; Copy result from stack into HY
    }
    printf("Decimal: %4.0f", HY);
    while (getchar() != 10);
}
```

A look at the disassembly

```
double SX = 3, SY = 4, HY=0;     004C1808 F2 0F 10 05 38 7B 4C 00 movsd       xmm0,mmword ptr [__real@40000000 (04C7B38h)]
                                 004C1810 F2 0F 11 45 F0          movsd       mmword ptr [SX],xmm0
                                 004C1815 F2 0F 10 05 40 7B 4C 00 movsd       xmm0,mmword ptr [string "1" (04C7B40h)]
                                 004C181D F2 0F 11 45 E0          movsd       mmword ptr [SY],xmm0
                                 004C1822 0F 57 C0                xorps       xmm0,xmm0
                                 004C1825 F2 0F 11 45 D0          movsd       mmword ptr [HY],xmm0

unsigned short cntrl=0x3FF,stat; 00301822 B8 FF 03 00 00          mov         eax,3FFh
                                 00301827 66 89 45 C4             mov         word ptr [cntrl],ax

unsigned char byt;
__asm
{
FINIT                            0030182B 9B                      wait
                                 0030182C DB E3                   fninit
FLDCW cntrl                      0030182E D9 6D C4                fldcw       word ptr [cntrl]
FINIT                            00301831 9B                      wait
                                 00301832 DB E3                   fninit
FLDCW cntrl;                     00301834 D9 6D C4                fldcw       word ptr [cntrl]
FLD   SX                         00301837 DD 45 F0                fld         qword ptr [SX]
FMUL  ST, ST(0)                  0030183A DC C8                   fmul        st(0),st
FLD   SY                         0030183C DD 45 E0                fld         qword ptr [SY]
FMUL  ST, ST(0)                  0030183F DC C8                   fmul        st(0),st
FADD  ST, ST(1)                  00301841 D8 C1                   fadd        st,st(1)
FSQRT                            00301843 D9 FA                   fsqrt
FSTP HY                          00301845 DD 5D D0                fstp        qword ptr [HY]
}
```

Pointer to compile time constant movsd mov scalar double precision

Wait dropped in by compiler x86 pauses until x87 done, older compiler put wait before every x87 instruction.

MMX

MMX was a technology added to Pentium processors in 1997 to allow them to do simultaneous operations on a group of data using a single instruction. This is a form of parallel processing known as SIMD (single instruction multiple data), that can run on a single processor.

The technology supports integer calculations only.

Note the original MMX registers are in fact the same as ST(0)(64 bit significand) used by the floating point processor (so you can't do both MMX and Floating Point at the same time).

To switch to MMX all you have to do is use an MMX instruction, to switch back there is a special instruction EMMS.

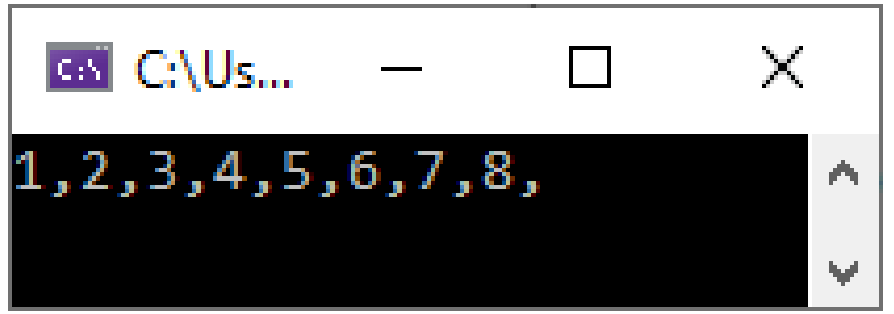
SSE1 supports but has also superseded MMX

MMX

```
void test()
{
    union mmx_word {
        unsigned char byte[8];
        unsigned __int64 value;
    };

    mmx_word NUM1 = { 0,1,2,3,4,5,6,7 };
    mmx_word NUM2 = { 1,1,1,1,1,1,1,1 };

    __asm
    {
        movq   mm0, NUM1
        movq   mm1, NUM2
        paddb mm0, mm1   // Add 8 bytes simultaneously
        movq   NUM1, mm0
    }

    for (int i = 0; i < 8; i++) printf("%d,", (unsigned int)NUM1.byte[i]);
    while (getchar() != 10);
}
```

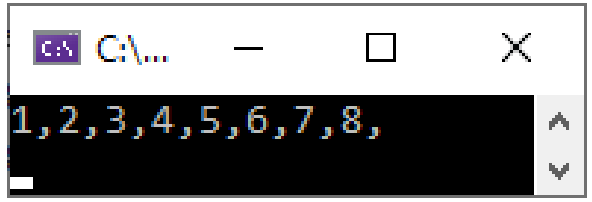
C compiler needs to reference the 64 bits as a single variable.

Using union structure to load 8x8bit integers into the 64 bit mmx word

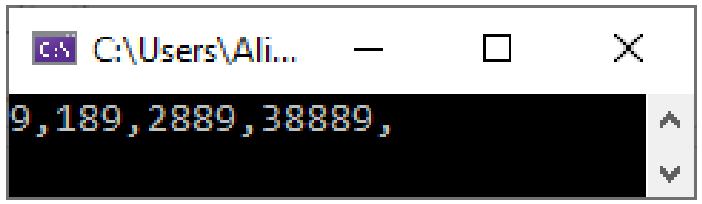


MMX Examples

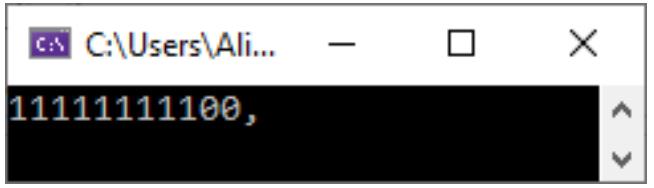
```
mmx_word NUM1 = { 0,1,2,3,4,5,6,7 };
mmx_word NUM2 = { 1,1,1,1,1,1,1,1 };
__asm
{
 movq   mm0, NUM1
 movq   mm1, NUM2
 paddb  mm0, mm1   // Add 8 bytes simultaneously
 movq   NUM1, mm0
}
```



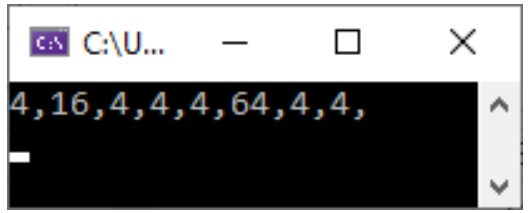

```
 union mmx_word {
        unsigned __int16 word[4];
        unsigned __int64 value;
};
mmx_word NUM1 = { 10,200,3000,40000};
mmx_word NUM2 = { 1,11,111,1111};
__asm
{
 movq   mm0, NUM1
 movq   mm1, NUM2
 psubw  mm0, mm1    // subtract words
 movq   NUM1,mm0
}
```



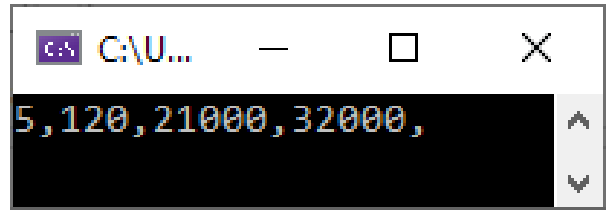

```
 union mmx_word {
        unsigned __int64 value;
};
mmx_word NUM1 = { 1234567890};
mmx_word NUM2 = { 9876543210};
__asm
{
 movq   mm0, NUM1
 movq   mm1, NUM2
 paddq  mm0, mm1    // add quadwords
 movq   NUM1, mm0
}
```




```
mmx_word NUM1 = { 1,4,1,1,1,16,1,1 };
__asm
{
 movq   mm0, NUM1
 psllq  mm0, 2   // shift left quadword 2 bits
 movq   NUM1, mm0
}
```

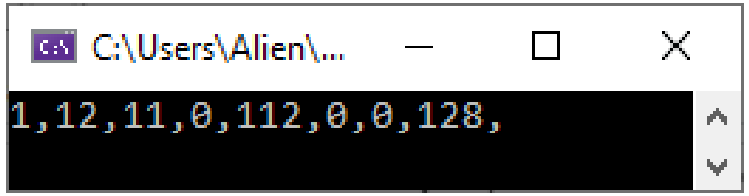



```
mmx_word NUM1 = { 1,20,300,400};
mmx_word NUM2 = { 5,6,70,80};
__asm
{
 movq   mm0, NUM1
 movq   mm1, NUM2
 pmullw  mm0, mm1 // multiply 4 words (low 16 bits)
 movq   NUM1, mm0
}
```

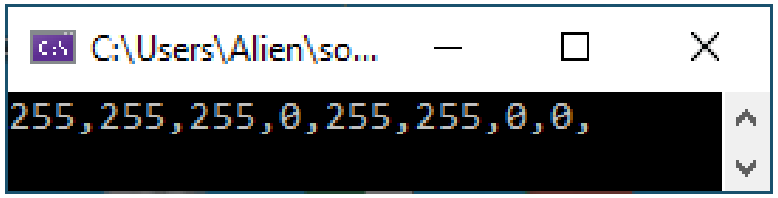


MMX Examples

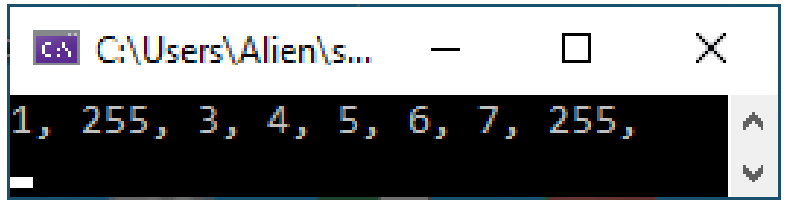
```
mmx_word NUM1 = { 1,12,123,17,123,34,124,128 };
mmx_word NUM2 = { 0xFF,255,15,0,240,1,1,128 };
__asm
{
  movq  mm0, NUM1
  movq  mm1, NUM2
  pand mm0, mm1    // and
  movq  NUM1, mm0
}
```




```
mmx_word NUM1 = { 1,12,123,-17,123,34,-124,128 };
mmx_word NUM2 = { 0,255,-15,0,240,1,1,128 };
__asm
{
 movq  mm0, NUM1
 movq  mm1, NUM2
 pcmpgtb mm0, mm1    // parallel compare greater than
 movq  NUM1, mm0
}
```




```
mmx_word NUM1 = { 1, 256, 3, 4};
mmx_word NUM2 = { 5, 6, 7, 300};
    __asm
    {
        movq  mm0, NUM1
        movq  mm1, NUM2       // Packed with Unsigned Saturation
        packuswb mm0, mm1     // 8 words in 2 register -> 8 bytes
        movq  NUM1, mm0       // in one register, saturate above 255
    }
```

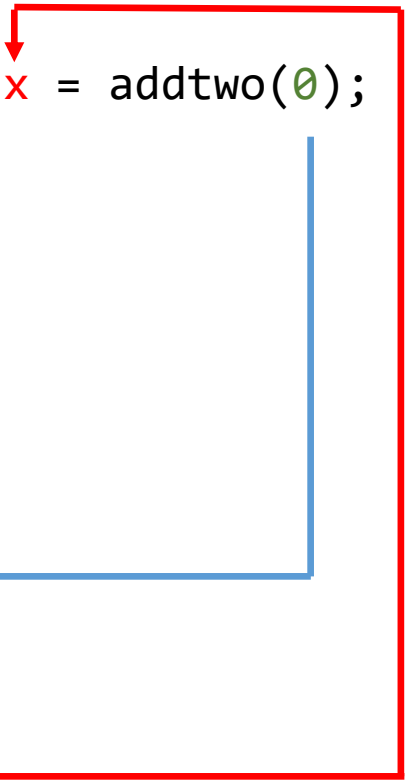


Opcode	Description
padd[b/w/d]	add 8 bytes, 4 words or two double words
psub[b/w/d]	subtract
pcmpeq[b/w/d]	compare equal
pcmpgt[b/w/d]	compare greater than
pmullw	multiply low order bits areresult
Pmulhw	multiply high order bit are result
pmaddwd	multiply and add
psra[w/d]	Shift right arithmetic
psll[w/d/q]	Shilft left logical
psrl[w/d/q]	Shilft right logical
punpckl[bw/wd/dq]	Unpack/interleave merge
punpckh[bw/wd/dq]	Unpack
packss[wb/dw]	pack words to bytes, double w to words
pand	and
pandn	not and
por	or
pxor	exor
pov[d/q]	move to or from register to memory
emms	Empty FP tag bit, return to x87fpu

C/C++ Calling conventions

```
extern "C" { unsigned long __stdcall addtwo(unsigned long);}

int main()
{
    unsigned int x = addtwo(0);
}
```

```
.DATA

.CODE
addtwo PROC input
    mov rax, rcx
    add rax, 2

    ret
addtwo ENDP

end
```



To make use of SSE and AX instruction you must put the code in a separate function. You can not put inline unsafe code in an x64 program so we are obliged to do this (can with 32 bit x86 code).

call pushes instruction pointer, rip, onto the stack
ret pops instruction pointer off the stack

push = push a value on the stack, push rax – pushes 8 bytes onto stack
pop = puts value on top of stack into register

rbp - register base pointer (start of stack)
rsp - register stack pointer (current location in stack, growing downwards)

Stack grows down, push rsp=rsp-8, pop rsp=rsp+8

You may subtract from rsp to provide space for an array.

rcx holds data for first parameter passed either an address (array) or the value itself (unsigned long)
rax holds the return value
You can pass data in arrays to and from the function

C/C++ Calling conventions

```
int main()
{
                                48 83 EC 28          sub         rsp,28h ; Add 40d bytes to stack
unsigned int x = addtwo(0);     33 C9                xor         ecx,ecx
                                E8 15 00 00 00       call        addtwo (07FF6C0121020h)

}
                                33 C0                xor         eax,eax
                                48 83 C4 28          add         rsp,28h
                                C3                   ret


                                .CODE
                                addtwo PROC input
          07FF6C0121020h →      55                   push        rbp       ; inherit base pointer
                                                                           ; save on stack
                                48 8B EC             mov         rbp,rsp   ; save stack pointer
                                48 8B C1             mov         rax,rcx
                                48 83 C0 02          add         rax,2
                                C9                   leave                 ; move rsp,rbp, pop rbp
                                C3                   ret                   ; ret
```

C/C++ Calling conventions

; Uses calling convention Microsoft x64 calling convention "Fastcall" similar to Ubuntu "V fastcall“

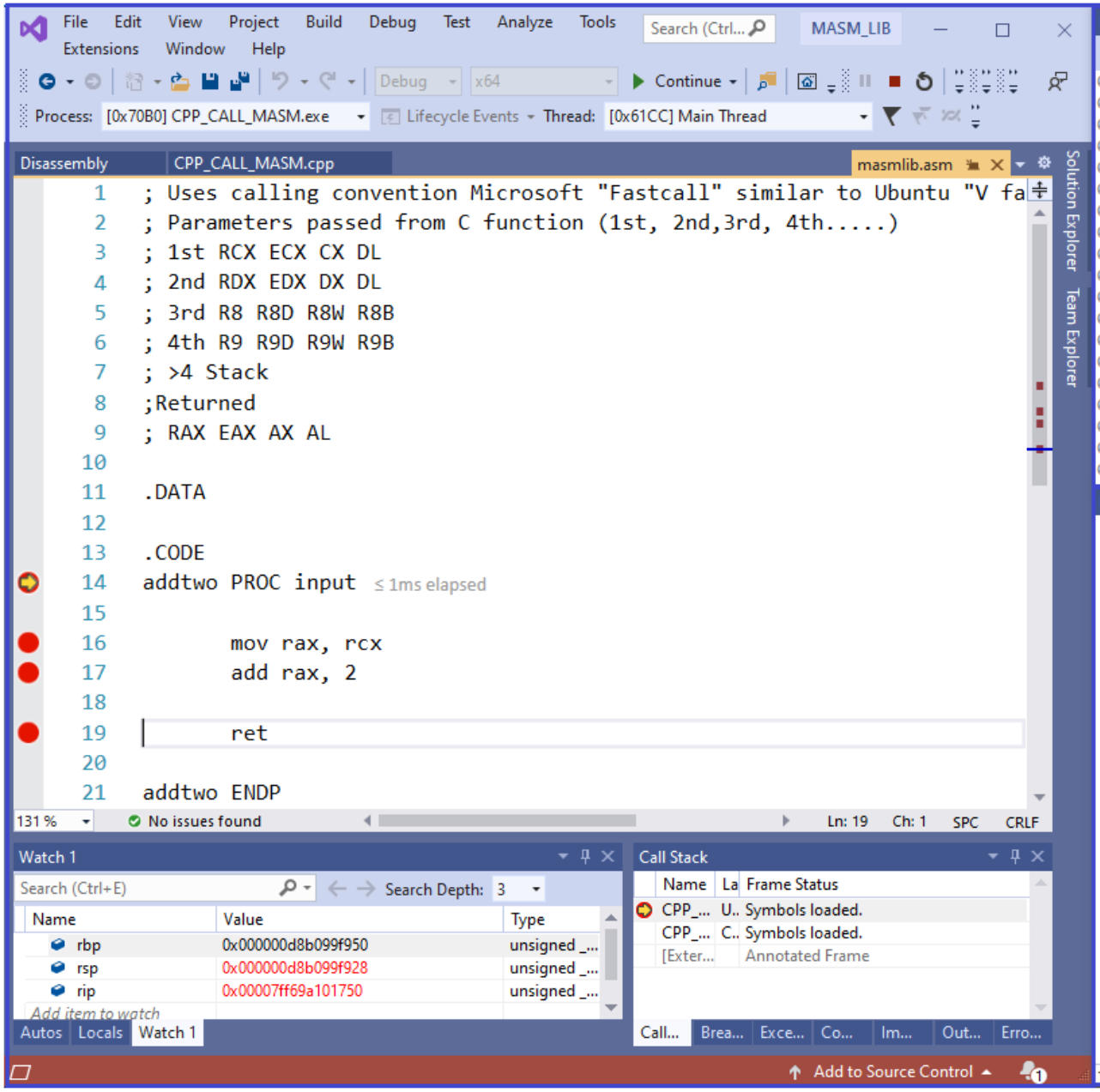
; Parameters passed from C function (1st, 2nd,3rd, 4th.....)
; 1st RCX ECX CX DL
; 2nd RDX EDX DX DL
; 3rd R8 R8D R8W R8B
; 4th R9 R9D R9W R9B
; >4 Stack
;Returned
; RAX EAX AX AL

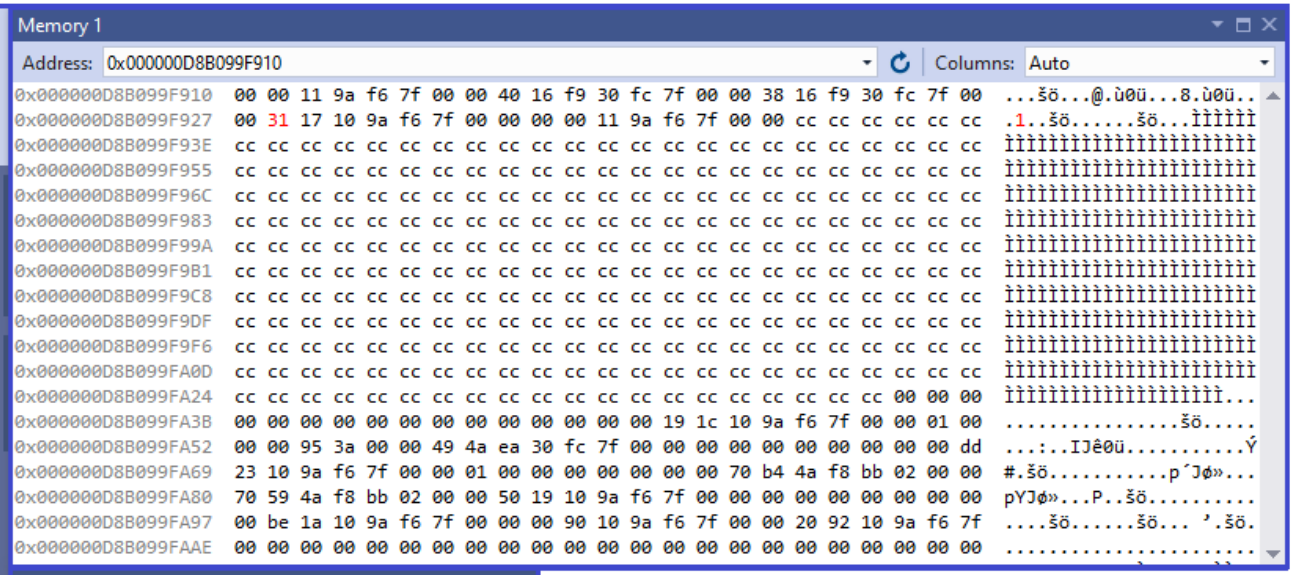
Registers used for first four parameters,
Stack used when there are more than four parameters.

```
.CODE
addtwo PROC input
55                  push        rbp      ; inherit base pointer, save on stack
48 8B EC            mov         rbp,rsp  ; save stack pointer
48 8B C1            mov         rax,rcx
48 83 C0 02         add         rax,2
C9                  leave                 ; move rsp,rbp, pop rbp
C3                  ret                   ; ret
```

https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

C/C++ Calling conventions – debugging the stack through a call cycle





You can use the debugger to watch registers and memory. You will need to add some break points after the program has started.

C/C++ Calling conventions – sequence of events

```
int main()
{
00007FF69A101700 40 55                 push        rbp ; save a copy of the stack base frame pointer on the stack, new stack grown from rbp
00007FF69A101702 57                    push        rdi ; we will use rdi so save it as it may be in use in the calling function
00007FF69A101703 48 81 EC 08 01 00 00  sub         rsp,108h      ; rsp=rsp-264d
00007FF69A10170A 48 8D 6C 24 20        lea         rbp,[rsp+20h] ; rbp = 32 bytes shaddow space (rdx, rcx, r8, r9)+ 264 bytes (load effecitive address)
00007FF69A10170F 48 8B FC              mov         rdi,rsp ; Set destination register rdi = rsp as it can be used with stoc instruction
00007FF69A101712 B9 42 00 00 00        mov         ecx,42h ; ecx = 66d
00007FF69A101717 B8 CC CC CC CC        mov         eax,0CCCCCCCCh
00007FF69A10171C F3 AB                 rep stos    dword ptr [rdi] ; repeat "store string fill" stack space with CC (allows stack management)
00007FF69A10171E 48 8D 0D DB E8 00 00  lea         rcx,[__D7728E49_CPP_CALL_MASM@cpp (07FF69A110000h)] ; program entry point to pass to debugger
00007FF69A101725 E8 5D F9 FF FF        call        __CheckForDebuggerJustMyCode (07FF69A101087h) ;Call the VS2019 debugger to get it started

unsigned int x = addtwo(0);
00007FF69A10172A 33 C9                 xor         ecx,ecx
00007FF69A10172C E8 01 F9 FF FF        call        addtwo (07FF69A101032h)  ; Microsoft x64 calling convention cdecl
00007FF69A101731 89 45 04              mov         dword ptr [x],eax
}
00007FF69A101734 33 C0                 xor         eax,eax
00007FF69A101736 48 8D A5 E8 00 00 00  lea         rsp,[rbp+0E8h]
00007FF69A10173D 5F                    pop         rdi
00007FF69A10173E 5D                    pop         rbp
00007FF69A10173F C3                    ret

.CODE
addtwo PROC input
00007FF69A101750 55                    push        rbp
00007FF69A101751 48 8B EC              mov         rbp,rsp
00007FF69A101754 48 8B C1              mov         rax,rcx
00007FF69A101757 48 83 C0 02           add         rax,2
00007FF69A10175B C9                    leave       ; mov rsp,rbp pop rbp
00007FF69A10175C C3                    ret
```

There are many other calling conventions

System V AMD64 ABI: on a Mac/Linux the calling convention is different there are six registers in the order rdi, rsi,rdx, rcx, r8, r9. After that the stack is used.

Breakpoint (before)	RIP	RSP	RBP	Memory
unsigned int x = addtwo(0);	00007FF69A10172A	0000007FCD5AF760	00000012876FF680	0x0000007FCD5AF740 00 00 11 9a f6 7f 00 00 40 16 4e 31 fc 7f 00 00 38 16 4e 31 fc 7f 00
xor ecx,ecx				0x0000007FCD5AF757 00 2a 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
call addtwo(07FF69A101032)	00007FF69A10172C			
addtwo PROC input/push rbp	00007FF69A101750	00000012876FF658	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 38 16 f9 30 fc 7f 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
				RSP 0x...F658 ↑ RSP 0x...9F960 ↑ RSP 0x...F950 ↓
mov rbp,rsp	00007FF69A101751	00000012876FF658	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
				↑ RSP 0x...F958
mov rax,rcx	00007FF69A101754	00000012876FF658	00000012876FF650	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
add rax, 2	00007FF69A101757	00000012876FF658	00000012876FF650	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
Leave = mov rsp,rbp, pop rbp	00007FF69A10175B	00000012876FF658	00000012876FF650	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
free the stack frame, empty stack				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
		move rsp,rbp	pop rbp	
ret	00007FF69A10175C	00000012876FF658	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
				RSP 0x...9F958 ↑ ↑ RSP 0x...F960
mov dword ptr [x],eax	00007FF69A101731	00000012876FF660	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
lea rsp,[rbp+0E8h]	00007FF69A101736	00000012876FF660	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
		[rsp+E8] main() sf		
pop rdi	00007FF69A10173D	00000012876FF768	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
				↑ RSP 0x...F968
pop rbp	00007FF69A10173E	00000012876FF770	00000012876FF680	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
ret	00007FF69A10173F	00000012876FF778	0000000000000000	0x00000012876FF640 00 00 11 9a f6 7f 00 00 40 16 f9 30 fc 7f 00 00 80 f6 6f 87 12 00 00
				0x00000012876FF657 00 31 17 10 9a f6 7f 00 00 00 00 11 9a f6 7f 00 00 cc cc cc cc cc cc
add rsp,48h	00007FF69A101C19	00000012876FF780	0000000000000000	
ret	00007FF69A101C1D	00000012876FF7C8	0000000000000000	
call _c_exit (07FF69A1012CBh)	Debugger stops			
mov eax,dword ptr [rsp+2Ch]				
jmp $LN18+35h (07FF69A101B29h)				

```
unsigned int x = addtwo(0);                    Call and return addresses
00007FF69A10172A 33 C9              xor        ecx,ecx
00007FF69A10172C E8 01 F9 FF FF     call       addtwo (07FF69A101032h)
00007FF69A101731 89 45 04           mov        dword ptr [x],eax
```

SSE Streaming SIMD Extensions

SSE1 Introduced with Pentium *!!!* in 1999

SSE1 Introduced floating point instructions eight registers xmm0-xmm77

SSEE1 8 x 128 bit Registers xmm0 to xmm1 each containing four 32 bit floating point numbers

SSE2 Introduced Integer calculation and largely replaced MMX.

Floating point x87fpu and SSE instructions can be mixed, but MMX and x87fpu can not.

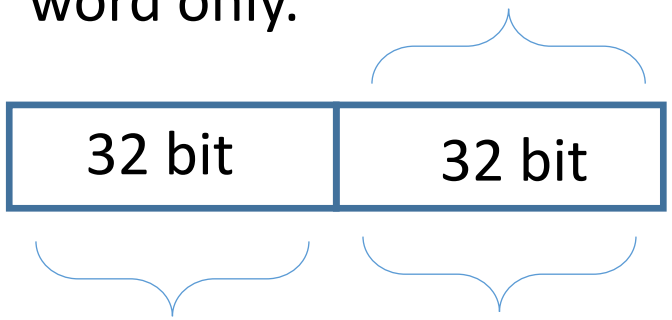
128 bits = 1

SSE Streaming SIMD Extensions

XMM registers can be configured as follow

- four 32 bit single precision numbers (floats) SSE1
- two 64-bit double-precision floating point numbers.
- two 64-bit integers or
- four 32-bit integers or
- eight 16-bit short integers or
- sixteen 8-bit bytes or characters

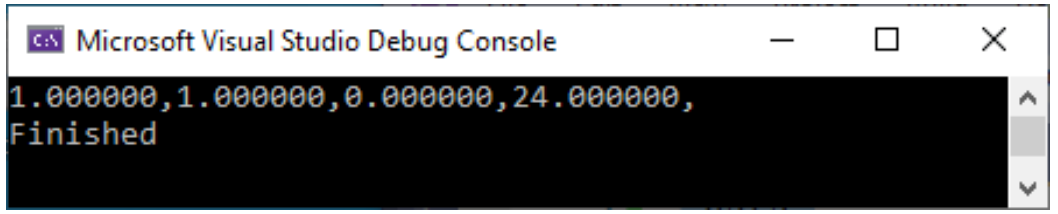
Scalar – works on lower word only.



Packed – works in parallel on all words.

SSE Streaming SIMD Extensions – VS2019

```
#include <stdio.h>
#include <xmmintrin.h>  // SIMD intrinsics

using namespace std;

extern "C"
{
  void sse_add(float* inputArray, float* outputArray);
}


int main(int argc, char* argv[])
{
// Arrays containing multiples of 4 floats
// (4x32=128bits,16 bytes) xmm0, xmm1
float inputArray[8] = { 1.0,  2.0,  3.0,  4.0,
                        0.0, -1.0, -3.0, 20.0 };
float outputArray[4] = {};

sse_add(inputArray, outputArray);   // Test function

for (int i = 0; i < 4; i++) printf("%f,", outputArray[i]);

printf("\nFinished\n");

return 0
```




```
.data
.code

    sse_add PROC C
    ; Parameters passed in from the C program call
    ; inputArray address in RCX.
    ; outputArray address in RDX.

    ; xmm0 is a 128 bit wide register,
    ;each float is 32 bits, 4 floats in total

    ; Move Unaligned Packed Single-Precision
    ; Floating-Point Values
    movups xmm0, [rcx]    ; Load xmm0 with bits 0-31
    movups xmm1, [rcx+16] ; Load xmm1 with bits 32-63

    addps  xmm1, xmm0     ; xmm1=xmm1+xmm0
    ;mulps  xmm1, xmm0     ; xmm1=xmm1*xmm0

    movups [rdx], xmm1 ; Store result.

    RET
    sse_add ENDP


end
```

SSE Instructions

Arithmetic	
addps	Adds 4 single-precision (32bit) floating-point values to 4 other single-precision floating-point values.
addss	Adds the lowest single-precision values, top 3 remain unchanged.
subps	Subtracts 4 single-precision floating-point values from 4 other single-precision floating-point values.
subss	Subtracts the lowest single-precision values, top 3 remain unchanged.
mulps	Multiplies 4 single-precision floating-point values with 4 other single-precision values.
mulss	Multiplies the lowest single-precision values, top 3 remain unchanged.
divps	Divides 4 single-precision floating-point values by 4 other single-precision floating-point values.
divss	Divides the lowest single-precision values, top 3 remain unchanged.
rcpps	Reciprocates (1/x) 4 single-precision floating-point values.
rcpss	Reciprocates the lowest single-precision values, top 3 remain unchanged.
sqrtps	Square root of 4 single-precision values.
sqrtss	Square root of lowest value, top 3 remain unchanged.
rsqrtps	Reciprocal square root of 4 single-precision floating-point values.
rsqrtss	Reciprocal square root of lowest single-precision value, top 3 remain unchanged.
maxps	Returns maximum of 2 values in each of 4 single-precision values.
maxss	Returns maximum of 2 values in the lowest single-precision value. Top 3 remain unchanged.
minps	Returns minimum of 2 values in each of 4 single-precision values.
minss	Returns minimum of 2 values in the lowest single-precision value, top 3 remain unchanged.
pavgb	Returns average of 2 values in each of 8 bytes.
pavgw	Returns average of 2 values in each of 4 words.
psadbw	Returns sum of absolute differences of 8 8bit values. Result in bottom 16 bits.
pextrw	Extracts 1 of 4 words.
pinsrw	Inserts 1 of 4 words.
pmaxsw	Returns maximum of 2 values in each of 4 signed word values.
pmaxub	Returns maximum of 2 values in each of 8 unsigned byte values.
pminsw	Returns minimum of 2 values in each of 4 signed word values.
pminub	Returns minimum of 2 values in each of 8 unsigned byte values.
pmovmskb	builds mask byte from top bit of 8 byte values.
pmulhuw	Multiplies 4 unsigned word values and stores the high 16bit result.
pshufw	Shuffles 4 word values. Takes 2 128bit values (source and dest) and an 8-bit immediate value

SSE Instructions

https://softpixel.com/~cwright/programming/simd/sse.php

Logic	
andnps	Logically ANDs 4 single-precision values with the logical inverse (NOT) of 4 other single-precision values.
andps	Logically ANDs 4 single-precision values with 4 other single-precision values.
orps	Logically ORs 4 single-precision values with 4 other single-precision values.
xorps	Logically XORs 4 single-precision values with 4 other single-precision values.

Compare	
cmpxxps	Compares 4 single-precision values.
cmpxxss	Compares lowest 2 single-precision values.
comiss	Compares lowest 2 single-recision values and stores result in EFLAGS.
ucomiss	Compares lowest 2 single-precision values and stores result in EFLAGS. (QNaNs don't throw exceptions with ucomiss, unlike comiss.)
xx above	
eq	Equal to.
lt	Less than.
le	Less than or equal to.
ne	Not equal.
nlt	Not less than.
nle	Not less than or equal to.
ord	Ordered.
unord	Unordered.

Conversion	
cvtpi2ps	Converts 2 32bit integers to 32bit floating-point values. Top 2 values remain unchanged.
cvtps2pi	Converts 2 32bit floating-point values to 32bit integers.
cvtsi2ss	Converts 1 32bit integer to 32bit floating-point value. Top 3 values remain unchanged.
cvtss2si	Converts 1 32bit floating-point value to 32bit integer.
cvttps2pi	Converts 2 32bit floating-point values to 32bit integers using truncation.
cvttss2si	Converts 1 32bit floating-point value to 32bit integer using truncation.

SSE Instructions

https://softpixel.com/~cwright/programming/simd/sse.php

State	
fxrstor	Restores FP and SSE State.
fxsave	Stores FP and SSE State.
ldmxcsr	Loads the mxcsr register.
stmxcsr	Stores the mxcsr register.

Shuffling	
shufps	Shuffles 4 single-precision values. Complex.
unpckhps	Unpacks single-precision values from high halves.
unpcklps	Unpacks single-precision values from low halves.

Cache Control	
prefetchT0	Fetches a cache-line of data into all levels of cache.
prefetchT1	Fetches a cache-line of data into all but the highest levels of cache.
prefetchT2	Fetches a cache-line of data into all but the two highest levels of cache.
prefetchNTA	Fetches data into only the highest level of cache, not the lower levels.
sfence	Guarantees that all memory writes issued before the sfence instruction are completed

Load/Store	
movaps	Moves a 128bit value.
movhlps	Moves high half to a low half.
movlhps	Moves low half to upper halves.?
movhps	Moves 64bit value into top half of an xmm register.
movlps	Moves 64bit value into bottom half of an xmm register.
movmskps	Moves top bits of single-precision values into bottom 4 bits of a 32bit register.
movss	Moves the bottom single-precision value, top 3 remain unchanged if the destination is another xmm register, otherwise they're set to zero.
movups	Moves a 128bit value. Address can be unaligned.
maskmovq	Moves a 64bit value according to a mask.
movntps	Moves a 128bit value directly to memory, skipping the cache, NT stands for "Non Temporal".
movntq	Moves a 64bit value directly to memory, skipping the cache.

AVX Advanced Vector Extensions (Sandy Bridge)

YMM registers are 256 bit and can be configured as follows

- eight 32-bit single-precision floating point numbers
- four 64-bit double-precision floating point numbers

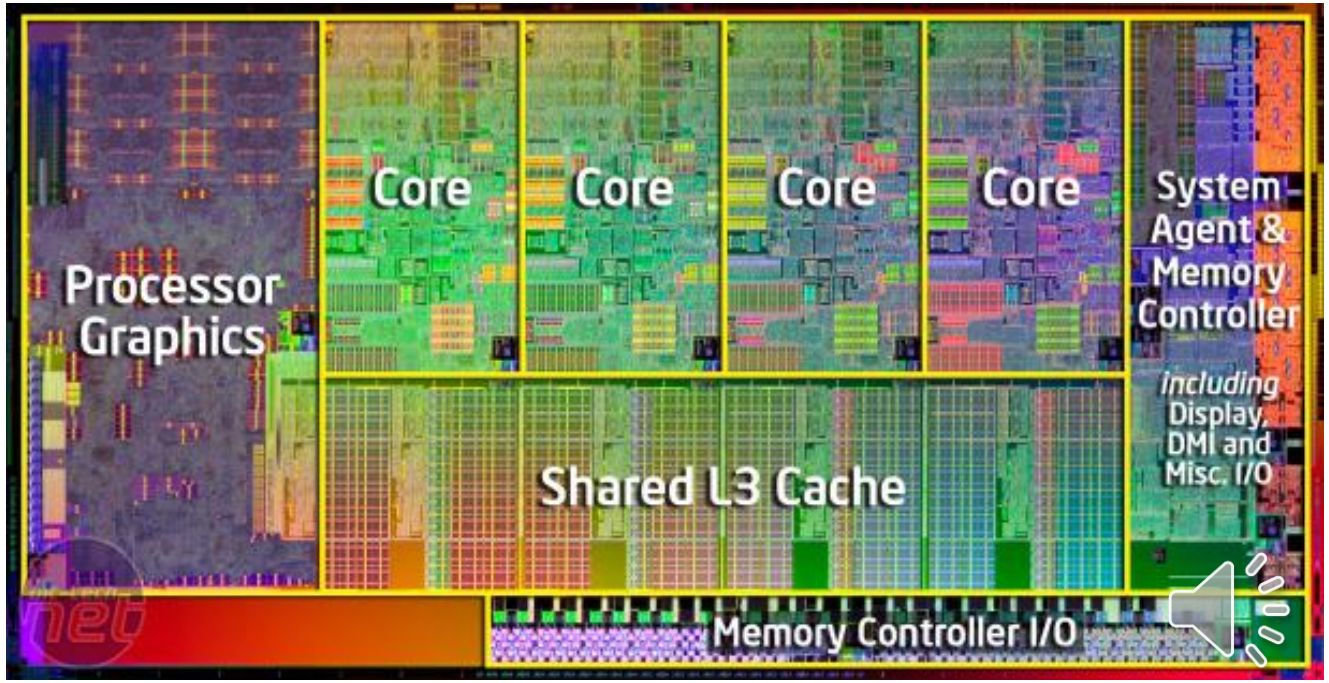
AVX uses a three-operand SIMD instruction format
 - one destination register and two source operands

AVX only uses the ymm registers

AVX 512 – 512 bit registers, zmm0 to zmm31

Each core has its own AVX registers

I5 –Sandy Bridge



AVX Instructions

Similar to SSE3 except prefixed with a v e.g. movup (SSE) becomes vmovup(AVX)

There are other instructions also, including the following.

Process Control	
VBROADCAST[S|D|F128]	Copies a 32-128 bit operand to all fields of a register.
VINSERTF128	Replaces the top or bottom half of a YMM register
VEXTRACT128	Copies the top or bottom half of a YMM register
VMASKMOVP[S|D]	Conditional move
VPERMILP[S|D]	"In-Lane" shuffle (can't shuffle across 128-bit boundaries).
VPERM2F128	Shuffles 2 sources into a single destination.
VZEROALL	Sets all YMM registers to zero.
VZEROUPPER	Clears the top half of all YMM registers.

Arithmetic	
vaddsubpd	Adds the top two doubles and subtracts the bottom two.
vaddsubps	Adds top singles and subtracts bottom singles.
vhaddpd	Top double is sum of top and bottom, bottom double is sum of second operand's top and bottom.
vhaddps	Horizontal addition of single-precision values.
vhsubpd	Horizontal subtraction of double-precision values.
vhsubps	Horizontal subtraction of single-precision values.

AVX Instructions

```
#include <stdio.h>      // printf
#include <xmmintrin.h>  // SIMD intrinsics

#define array_length 16 // multiples of 8 to fit 256 AVX register size

extern "C"
{
void avx_poly(float* inputArray, float* outputArray, long ary_len);
}


int main(int argc, char* argv[])
{
// Arrays containing containing 2x8x32bit floats = 2 x 256bit
float inputArray[array_length] = { 0.0, 1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,
                                   8.0, 9.0, -1.0, -2.0, -3.0, -4.0, -5.0, -6.0 };

float outputArray[array_length] = {};

avx_poly(inputArray, outputArray, array_length);   // Test function

for (int j = 0; j < (array_length/8); j++)
{
for (int i = 0; i < 8; i++) printf("% 3.1f, ", outputArray[i + (8 * j)]);
printf("\n");
}

return 0;
}
```

AVX Instructions

```
; Rebuild on each edit to implement changes
.data
; Code operates on 8 floats in parallel
; The constant 2 needs to stored as 8xfloat
A dd 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0
.code
avx_poly PROC C
; Parameters passed in by C
; inputArray address in RCX   1st
; outputArray address in RDX  2nd
; ary_len in R8.  3rd
; ymm0 is a 256 bit wide register, each float is a 32 bits, 8 floats in total
; for each group of 8 in our input array calculate the polynomial

loop1:
vmovups ymm0, [rcx+r8*4-32] ; Load X starting 8 floats from end of array hence -32
                            ; R8 is 32 bit (4x32=256, 8x32bit floats)
                            ; rcx is the base address i.e. &inputArray[0]
vmulps ymm1, ymm0, ymm0     ; Calculate X^2.
vmovups ymm0,[A]            ; Set ymm0 so all floats are 2
vaddps ymm0, ymm0, ymm1     ; Calculate 2 + X^2.

vmovups [rdx+r8*4-32], ymm0 ; Store result.
sub r8, 8
jg  loop1

vzeroall  ; Indirect method of saying AVX finished

RET
avx_poly ENDP
end
```