Introduction to Computer Systems
15-213/18-243, fall 2009
8th Lecture, Sep. 17th

Instructors:
Roger B. Dannenberg and Greg Ganger
## Last Time

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Used for linking</td>
</tr>
<tr>
<td>%r12</td>
<td>C: Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Last Time

- Procedures (x86-64): Optimizations
  - No base/frame pointer
  - Passing arguments to functions through registers (if possible)
  - Sometimes: Writing into the “red zone” (below stack pointer)
  - Sometimes: Function call using **jmp** (instead of **call**)
  - **Reason: Performance**
    - use stack as little as possible
    - while obeying rules (e.g., caller/callee save registers)
Last Time

- Arrays

```c
int val[5];

1 5 2 1 3
x x+4 x+8 x+12 x+16 x+20
```

- Nested

```c
int pgh[4][5];

1 5 2 0 6 1 5 2 1 3 1 5 2 1 7 1 5 2 2 1
76 96 116 136 156
```

- Multi-level

```c
int *univ[3]
```

```
univ
160 36
164 16
168 56

mit
16 20 24 28 32 36

ucb
36 40 44 48 52 56

univ
9 4 7 2 0
```

Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size

- **Programming**
  - Must do index computation explicitly

- **Performance**
  - Accessing single element costly
  - Must do multiplication

```c
int * new_var_matrix(int n)
{
    return (int *)
        calloc(sizeof(int), n*n);
}

int var_ele
    (int *a, int i, int j, int n)
{
    return a[i*n+j];
}
```

```assembly
movl 12(%ebp),%eax  # i
movl 8(%ebp),%edx  # a
imull 20(%ebp),%eax  # n*i
addl 16(%ebp),%eax  # n*i+j
movl (%edx,%eax,4),%eax  # Mem[a+4*(i*n+j)]
```
Dynamic Array Multiplication

- Per iteration:
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 for array indexing
    - 1 for loop index
    - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele
(int *a, int *b,
 int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result +=
            a[i*n+j] * b[j*n+k];
    return result;
}
```

- i-th row
- j-th column
Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to `-O2`

- **Code Motion**
  - Expression `i*n` can be computed outside loop

- **Strength Reduction**
  - Incrementing `j` has effect of incrementing `j*n+k` by `n`

- **Operations count**
  - 4 adds, 1 mult

```c
{ int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n+j] * b[j*n+k];
  return result;
}
```

```c
{ int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

- 4 adds, 3 mults

- 4 adds, 1 mult
Today

- Structures
- Alignment
- Unions
- Floating point
Structures

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

- **Accessing Structure Member**

```c
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

- **IA32 Assembly**

```assembly
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Structure Member

```c
struct rec { 
  int i;
  int a[3];
  int *p;
};
```

```c
int *find_a (struct rec *r, int idx) 
{
  return &r->a[idx];
}
```

Will disappear blackboard?

```asm
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax
leal 4(%eax,%edx),%eax
```

What does it do?
Generating Pointer to Structure Member

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

```c
int *find_a
    (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```asm
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax   # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Structure Referencing (Cont.)

- C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

What does it do?

```assembly
# %edx = r
movl (%edx),%ecx    # r->i
leal 0(%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx)    # Update r->p
```
Today

- Structures
- Alignment
- Unions
- Floating point
Alignment

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

- **1 byte:** char, ...  
  - no restrictions on address

- **2 bytes:** short, ...  
  - lowest 1 bit of address must be $0_2$

- **4 bytes:** int, float, char *, ...,  
  - lowest 2 bits of address must be $00_2$

- **8 bytes:** double, ...  
  - Windows (and most other OS’s & instruction sets):  
    - lowest 3 bits of address must be $000_2$
  - Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes:** long double  
  - Windows, Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type
Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be $0_2$

- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be $00_2$

- **8 bytes: double, char *, ...**
  - Windows & Linux:
    - lowest 3 bits of address must be $000_2$

- **16 bytes: long  double**
  - Linux:
    - lowest 3 bits of address must be $000_2$
    - i.e., treated the same as a 8-byte primitive data type
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K =$ Largest alignment of any element
  - Initial address & structure length must be multiples of $K$

- **Example (under Windows or x86-64):**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Different Alignment Conventions

- **x86-64 or IA32 Windows:**
  - $K = 8$, due to `double` element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **IA32 Linux**
  - $K = 4$; `double` treated like a 4-byte data type
Saving Space

- Put large data types first

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```

- Effect (example x86-64, both have K=8)
Arrays of Structures

- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```c
short get_j(int idx) {
    return a[idx].j;
}
```

```assembly
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```
Today

- Structures
- Alignment
- Unions
- Floating point
Union Allocation

- Allocate according to largest element
- Can only use ones field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

![Diagram of union and struct allocation]
Using Union to Access Bit Patterns

typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u ?

Same as (unsigned) f ?
Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines

- **Big Endian**
  - Most significant byte has lowest address
  - PowerPC, Sparc

- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86
Byte Ordering Example

union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
    [0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
    [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```
Byte Ordering on IA32

Little Endian

Output on IA32:

- Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
- Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
- Long 0 == [0xf3f2f1f0]
# Byte Ordering on Sun

## Big Endian

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
<td>f4</td>
<td>f5</td>
<td>f6</td>
<td>f7</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Output on Sun:

- **Characters** 0–7: `[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]`
- **Shorts** 0–3: `[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]`
- **Ints** 0–1: `[0xf0f1f2f3, 0xf4f5f6f7]`
- **Long** 0: `[0xf0f1f2f3]`
Byte Ordering on x86-64

Little Endian

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
</tr>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSB</td>
<td>MSB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Output on x86-64:

- Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
- Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
- Long 0  == [0xf7f6f5f4f3f2f1f0]
Summary

Arrays in C
- Contiguous allocation of memory
- Aligned to satisfy every element’s alignment requirement
- Pointer to first element
- No bounds checking

Structures
- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions
- Overlay declarations
- Way to circumvent type system
Today

- Structures
- Alignment
- Unions

**Floating point**

- x87 (available with IA32, becoming obsolete)
- SSE3 (available with x86-64)
IA32 Floating Point (x87)

- **History**
  - 8086: first computer to implement IEEE FP
    - separate 8087 FPU (floating point unit)
  - 486: merged FPU and Integer Unit onto one chip
  - Becoming obsolete with x86-64

- **Summary**
  - Hardware to add, multiply, and divide
  - Floating point data registers
  - Various control & status registers

- **Floating Point Formats**
  - single precision (C `float`): 32 bits
  - double precision (C `double`): 64 bits
  - extended precision (C `long double`): 80 bits
FPU Data Register Stack (x87)

- FPU register format (80 bit extended precision)

- FPU registers
  - 8 registers %st(0) - %st(7)
  - Logically form stack
  - Top: %st(0)
  - Bottom disappears (drops out) after too many pushes
FPU instructions (x87)

- Large number of floating point instructions and formats
  - ~50 basic instruction types
  - load, store, add, multiply
  - sin, cos, tan, arctan, and log
    - Often slower than math lib

- Sample instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds Addr</td>
<td>push Mem[Addr]</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls Addr</td>
<td>%st(0) ← %st(0)*M[Addr]</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>%st(1) ← %st(0)+%st(1);pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>
FP Code Example (x87)

- **Compute inner product of two vectors**
  - Single precision arithmetic
  - Common computation

```c
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i] * y[i];
    return result;
}
```

```assembly
pushl %ebp              # setup
movl %esp,%ebp
pushl %ebx

movl 8(%ebp),%ebx       # %ebx=&x
movl 12(%ebp),%ecx      # %ecx=&y
movl 16(%ebp),%edx      # %edx=n
fldz                    # push +0.0
xorl %eax,%eax          # i=0
cmpl %edx,%eax          # if i>=n done
jge .L3

.L5:
   flds (%ebx,%eax,4)      # push x[i]
   fmuls (%ecx,%eax,4)     # st(0)*=y[i]
   faddp                   # st(1)+=st(0); pop
   incl %eax               # i++
   cmpl %edx,%eax          # if i<n repeat
   jle .L5

.L3:
   movl -4(%ebp),%ebx      # finish
   movl %ebp, %esp
   popl %ebp
   ret                      # st(0) = result
```
Inner Product Stack Trace

Initialization

1. fldz
   0.0 %st(0)

Iteration 0

2.flds (%ebx, %eax, 4)
   0.0 %st(1)
   x[0] %st(0)

3. fmuls (%ecx, %eax, 4)
   0.0 %st(1)
   x[0] * y[0] %st(0)

4. faddp
   0.0 + x[0] * y[0] %st(0)

Iteration 1

5. flds (%ebx, %eax, 4)
   x[0] * y[0] %st(1)
   x[1] %st(0)

6. fmuls (%ecx, %eax, 4)
   x[0] * y[0] %st(1)
   x[1] * y[1] %st(0)

7. faddp
   x[0] * y[0] + x[1] * y[1] %st(0)
Today

- Structures
- Alignment
- Unions
- Floating point
  - x87 (available with IA32, becoming obsolete)
  - SSE3 (available with x86-64)
Vector Instructions: SSE Family

- **SIMD (single-instruction, multiple data) vector instructions**
  - New data types, registers, operations
  - Parallel operation on small (length 2-8) vectors of integers or floats
  - Example:
    - Floating point vector instructions
      - Available with Intel’s SSE (streaming SIMD extensions) family
      - SSE starting with Pentium III: 4-way single precision
      - SSE2 starting with Pentium 4: 2-way double precision
      - All x86-64 have SSE3 (superset of SSE2, SSE)
# Intel Architectures (Focus Floating Point)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Architectures</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>x86-16</td>
<td></td>
</tr>
<tr>
<td>286</td>
<td>x86-32</td>
<td></td>
</tr>
<tr>
<td>386</td>
<td>x86-32</td>
<td></td>
</tr>
<tr>
<td>486</td>
<td>x86-32</td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>Pentium MMX</td>
<td>MMX</td>
<td></td>
</tr>
<tr>
<td>Pentium III</td>
<td>SSE</td>
<td>4-way single precision fp</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>SSE2</td>
<td>2-way double precision fp</td>
</tr>
<tr>
<td>Pentium 4E</td>
<td>SSE3</td>
<td></td>
</tr>
<tr>
<td>Pentium 4F</td>
<td>x86-64 / em64t</td>
<td></td>
</tr>
<tr>
<td>Core 2 Duo</td>
<td>SSE4</td>
<td></td>
</tr>
</tbody>
</table>

*Our focus: SSE3*

used for scalar (non-vector) floating point
### SSE3 Registers

- All caller saved
- `%xmm0` for floating point return value

128 bit = 2 doubles = 4 singles

<table>
<thead>
<tr>
<th>%xmm0</th>
<th>Argument #1</th>
<th>%xmm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%xmm1</td>
<td>Argument #2</td>
<td>%xmm9</td>
</tr>
<tr>
<td>%xmm2</td>
<td>Argument #3</td>
<td>%xmm10</td>
</tr>
<tr>
<td>%xmm3</td>
<td>Argument #4</td>
<td>%xmm11</td>
</tr>
<tr>
<td>%xmm4</td>
<td>Argument #5</td>
<td>%xmm12</td>
</tr>
<tr>
<td>%xmm5</td>
<td>Argument #6</td>
<td>%xmm13</td>
</tr>
<tr>
<td>%xmm6</td>
<td>Argument #7</td>
<td>%xmm14</td>
</tr>
<tr>
<td>%xmm7</td>
<td>Argument #8</td>
<td>%xmm15</td>
</tr>
</tbody>
</table>
SSE3 Registers

- Different data types and associated instructions
  - Integer vectors:
    - 16-way byte
    - 8-way 2 bytes
    - 4-way 4 bytes
  - Floating point vectors:
    - 4-way single
    - 2-way double
  - Floating point scalars:
    - single
    - double
SSE3 Instructions: Examples

- Single precision 4-way vector add: \texttt{addps \%xmm0 \%xmm1}

- Single precision scalar add: \texttt{addss \%xmm0 \%xmm1}
SSE3 Instruction Names

- packed (vector)
  - addps
  - addpd

- single precision
  - addps
  - addpd

- double precision
  - addps
  - addpd

- single slot (scalar)
  - addss
  - addsd

this course
SSE3 Basic Instructions

- **Moves**

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>movss</td>
<td>movsd</td>
<td>D ← S</td>
</tr>
</tbody>
</table>

  - Usual operand form: reg → reg, reg → mem, mem → reg

- **Arithmetic**

<table>
<thead>
<tr>
<th>Single</th>
<th>Double</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>addss</td>
<td>addsd</td>
<td>D ← D + S</td>
</tr>
<tr>
<td>subss</td>
<td>subsd</td>
<td>D ← D − S</td>
</tr>
<tr>
<td>mulss</td>
<td>mulsd</td>
<td>D ← D × S</td>
</tr>
<tr>
<td>divss</td>
<td>divsd</td>
<td>D ← D / S</td>
</tr>
<tr>
<td>maxss</td>
<td>maxsd</td>
<td>D ← max(D,S)</td>
</tr>
<tr>
<td>minss</td>
<td>minsd</td>
<td>D ← min(D,S)</td>
</tr>
<tr>
<td>sqrtss</td>
<td>sqrtsd</td>
<td>D ← sqrt(S)</td>
</tr>
</tbody>
</table>
x86-64 FP Code Example

- Compute inner product of two vectors
  - Single precision arithmetic
  - Uses SSE3 instructions

```c
float ipf (float x[],
          float y[],
          int n) {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```assembly
ipf:
xorps  %xmm1, %xmm1
xorl   %ecx, %ecx
jmp    .L8
.L10:
movslq %ecx,%rax
incl   %ecx
movss (%rsi,%rax,4), %xmm0
mulss (%rdi,%rax,4), %xmm0
addss %xmm0, %xmm1
.L8:
    cmpl   %edx, %ecx
    jl     .L10
movaps %xmm1, %xmm0
ret
```

Will disappear Blackboard?
x86-64 FP Code Example

- Compute inner product of two vectors
  - Single precision arithmetic
  - Uses SSE3 instructions

```c
float ipf (float x[], float y[], int n) {
    int i;
    float result = 0.0;

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

    return result;
}
```

ipf:

```
xorps   %xmm1, %xmm1  # result = 0.0
xorl    %ecx, %ecx    # i = 0
jmp     .L8           # goto middle
.L10:
    movslq  %ecx,%rax   # icpy = i
    incl    %ecx        # i++
    movss   (%rsi,%rax,4), %xmm0  # t = y[icpy]
    mulss  (%rdi,%rax,4), %xmm0   # t *= x[icpy]
    addss  %xmm0, %xmm1        # result += t
.L8:
    cmpl    %edx, %ecx     # i:n
    jl      .L10          # if < goto loop
    movaps  %xmm1, %xmm0   # return result
    ret
```
## SSE3 Conversion Instructions

- **Conversions**
  - Same operand forms as moves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvtss2sd</td>
<td>single → double</td>
</tr>
<tr>
<td>cvtsd2ss</td>
<td>double → single</td>
</tr>
<tr>
<td>cvtsi2ss</td>
<td>int → single</td>
</tr>
<tr>
<td>cvtsi2sd</td>
<td>int → double</td>
</tr>
<tr>
<td>cvtsi2ssq</td>
<td>quad int → single</td>
</tr>
<tr>
<td>cvtsi2sdq</td>
<td>quad int → double</td>
</tr>
<tr>
<td>cvttss2si</td>
<td>single → int (truncation)</td>
</tr>
<tr>
<td>cvttsd2si</td>
<td>double → int (truncation)</td>
</tr>
<tr>
<td>cvttss2siq</td>
<td>single → quad int (truncation)</td>
</tr>
<tr>
<td>cvttss2siq</td>
<td>double → quad int (truncation)</td>
</tr>
</tbody>
</table>
x86-64 FP Code Example

double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}

a %xmm0 double
x %xmm1 float
b %xmm2 double
i %edi int

funct:
    cvtss2sd %xmm1, %xmm1
    mulsd %xmm0, %xmm1
    cvtsi2sd %edi, %xmm0
    divsd %xmm0, %xmm2
    movsd %xmm1, %xmm0
    subsd %xmm2, %xmm0

ret
x86-64 FP Code Example

double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}

a %xmm0 double
x %xmm1 float
b %xmm2 double
i %edi int

funct:
cvtss2sd %xmm1, %xmm1  # %xmm1 = (double) x
mulsd  %xmm0, %xmm1    # %xmm1 = a*x
cvtsi2sd %edi, %xmm0    # %xmm0 = (double) i
divsd  %xmm0, %xmm2    # %xmm2 = b/i
movsd  %xmm1, %xmm0    # %xmm0 = a*x
subsd  %xmm2, %xmm0    # return a*x - b/i
ret
Constants

double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}

# Constant declarations
.LC2:
    .long 3435973837     # Low order four bytes of 1.8
    .long 1073532108     # High order four bytes of 1.8
.LC4:
    .long 0              # Low order four bytes of 32.0
    .long 1077936128     # High order four bytes of 32.0

# Code
cel2fahr:
    mulsd .LC2(%rip), %xmm0     # Multiply by 1.8
    addsd .LC4(%rip), %xmm0     # Add 32.0
    ret

Here: Constants in decimal format
- compiler decision
- hex more readable
Checking Constant

- Previous slide: Claim
  
  `.LC4:
  
  .long 0          # Low order four bytes of 32.0
  .long 1077936128 # High order four bytes of 32.0

- Convert to hex format:
  
  `.LC4:
  
  .long 0x0        # Low order four bytes of 32.0
  .long 0x40400000 # High order four bytes of 32.0

- Convert to double (blackboard?):
  
  - Remember: $e = 11$ exponent bits, bias = $2^{e-1}-1 = 1023$
Comments

- **SSE3 floating point**
  - Uses lower ½ (double) or ⅛ (single) of vector
  - Finally departure from awkward x87
  - Assembly very similar to integer code

- **x87 still supported**
  - Even mixing with SSE3 possible
  - Not recommended

- For highest floating point performance
  - Vectorization a must (but not in this course😊)
  - See next slide
Vector Instructions

- Starting with version 4.1.1, gcc can autovectorize to some extent
  - -O3 or -ftree-vectorize
  - No speed-up guaranteed
  - Very limited
  - icc as of now much better
  - Fish machines: gcc 3.4

- For highest performance vectorize yourself using intrinsics
  - Intrinsics = C interface to vector instructions
  - Learn in 18-645

- Future
  - Intel AVX announced: 4-way double, 8-way single