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)

Dynamic Nested Arrays

- Strength
  - Can create matrix of any size
- Programming
  - Must do index computation explicitly
- Performance
  - Accessing single element costly
  - Must do multiplication

Dynamic Array Multiplication

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
# %edx = r
movl (%edx),%ecx
# r->i
leal 0(%ecx,4),%eax
leal 4(%edx,%eax),%eax
movl %eax,16(%edx)
```

Generating Pointer to Structure Member

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

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

Will disappear blackboard?

Generating Pointer to Array Element

- Offset of each structure member determined at compile time

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

```
# %edx = r
movl (%edx),%eax
```

Element i

Structure Referencing (Cont.)
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 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
- 4 bytes: int, float, char *, ...
  - lowest 2 bits of address must be 00
- 8 bytes: double, ...
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
      - 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
      - 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
- 4 bytes: int, float, ...
  - lowest 2 bits of address must be 00
- 8 bytes: double, char *, ...
  - Windows & Linux:
    - lowest 3 bits of address must be 000
- 16 bytes: long double
  - Linux:
    - lowest 3 bits of address must be 000
      - i.e., treated the same as an 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

Different Alignment Conventions
- x86-64 or IA32 Windows:
  - \( K = 8 \), due to double element
- IA32 Linux
  - \( K = 4 \), double treated like a 4-byte primitive data type

### Example (under Windows or x86-64)

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
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 $K=8$)

```
\begin{array}{cccc}
  p &=& 0 & 4 & 8 & 16 & 24 \\
\end{array}
```

Arrays of Structures

- Satisfy alignment requirement for every element

```
\begin{array}{cccc}
  a[0] & a[1] & a[2] & \cdots \\
  v & i[0] & i[1] & c
\end{array}
```

Accessing Array Elements

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

```
\begin{array}{cccc}
a[0] & \cdots & a[1] & \cdots \\
\end{array}
a+0 & a+12i
```

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

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

Union Allocation

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

```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

```
\begin{array}{cccc}
  sp+0 & sp+4 & sp+8 & sp+16 & sp+24 \\
  c & i[0] & i[1] & v
\end{array}
```

```
union U2 {
    float f;
    unsigned u;
} bit_float_t;
```

Using Union to Access Bit Patterns

```
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;
}
```

Today

- Structures
- Alignment
- Unions
- Floating point

Union Allocation

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

```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

```
\begin{array}{cccc}
  sp+0 & sp+4 & sp+8 & sp+16 & sp+24 \\
  c & i[0] & i[1] & v
\end{array}
```

```
union U2 {
    float f;
    unsigned u;
} bit_float_t;
```

Using Union to Access Bit Patterns

```
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

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

```
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**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

---

Byte Ordering on Sun

**Big Endian**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1f2f3,0xf4f5f6f7]
Ints 0-1 == [0xf0f1f2f3f4f5f6f7]
Long 0 == [0xf0f1f2f3f4f5f6f7]

---

Byte Ordering on x86-64

**Little Endian**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1f2f3,0xf4f5f6f7]
Ints 0-1 == [0xf0f1f2f3f4f5f6f7]
Long 0 == [0xf0f1f2f3f4f5f6f7f8f9]

---
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 requirement
- 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)
  - 79 78 64 63 0
    - exp frac
- 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>flda</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flda Addr</td>
<td>push Mem(Addr)</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>faddl</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
    - For (l = 0; l < n; l++)
      - result = x[l]*y[l];
      - return result;

  ```
  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;
  }
  ```

  ```
  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:
    fldl (%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
    jl .L5
  .L3:
    movl -4(%ebp),%ebx    # finish
    movl %ebp, %esp
    popl %ebp
    ret                   # st(0) = result
  ```
Inner Product Stack Trace

**Initialization**
1. `fld 0.0 %st(0)`

**Iteration 0**
2. `flds (%ebx,%eax,4) 0.0 %st(1)`
3. `fmuls (%ecx,%eax,4) 0.0 %st(1)`
4. `faddp %st(0)`

**Iteration 1**
5. `flds (%ebx,%eax,4) x[0] %st(1)`
6. `fmuls (%ecx,%eax,4) x[0] %st(1)`
7. `faddp %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:
    
    \[
    \begin{array}{c|c}
    \hline
    \text{integer} & \text{integer} \\
    \text{vector} & \text{vector} \\
    \text{operation} & \text{operation} \\
    \hline
    \end{array}
    \]
    
    “4-way”
- 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)

---

**SSE3 Registers**
- All caller saved
- 5 %xmm0 for floating point return value

---

**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></td>
<td></td>
</tr>
<tr>
<td>486</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium MMX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium III</td>
<td>SSE</td>
<td>2-way double precision fp</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>SSE2</td>
<td></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 Instructions: Examples

- Single precision 4-way vector add: `addps %xmm0 %xmm1` to `%xmm0`
- Single precision scalar add: `addss %xmm0 %xmm1` to `%xmm0`

SSE3 Instruction Names

- `addps` (packed vector) and `addss` (single precision scalar add)
- `addpd` and `addsd` (double precision)

SSE3 Basic Instructions

- Moves
  - Single `movs` to Double `movd` and Effect `D -> S`
  - Usual operand form: reg → reg, reg → mem, mem → reg

- Arithmetic
  - Single `addas` to Double `addsd` and Effect `D + S`

x86-64 FP Code Example

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

```
float ipf [float x[], float y[], int n] {
    int i; // Initialize loop counter
    float result = 0.0; // Initialize result
    for (i = 0; i < n; i++) // Loop through elements
        result += x[i] * y[i]; // Add corresponding elements
    return result; // Return final result
}
```

SSE3 Conversion Instructions

- Conversions
  - Same operand forms as moves

```
Instruction      Description
----------------------
cvtss2sd  single → double
cvtsd2ss  double → single
cvtss2sd  int → single
cvtssl2sd  int → double
cvtss2quad quad int → single
cvtss2quad quad int → double
cvtss2zl   single → int (truncation)
cvtss2zl   double → int (truncation)
cvtss2qd   single → quad int (truncation)
cvtss2qd   double → quad int (truncation)
```
x86-64 FP Code Example

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

```assembly
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

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

```assembly
# 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
```

Comments

- SSE3 floating point
  - Uses lower ½ (double) or ¼ (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

Checking Constant

- Previous slide: Claim
- Convert to hex format:
- Convert to double (blackboard?):
  - Remember: e = 11 exponent bits, bias = 2^10 - 1 = 1023

Vector Instructions

- Starting with version 4.1.1, gcc can autovectorize to some extent
  - -O3 or -fno-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