Introduction to Computer Systems

15-213/18-243, spring 2009
9th Lecture, Feb. 10th

Instructors:

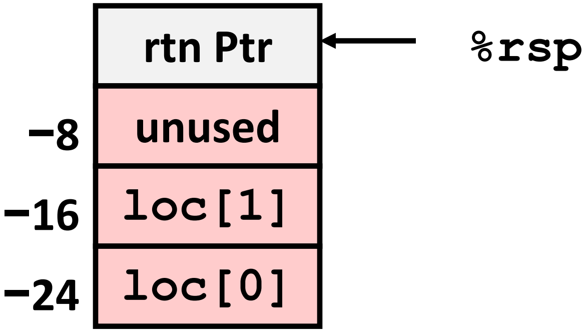
Gregory Kesden and Markus Püschel

Last Time

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Callee saved
`%rdx`	Argument #3	`%r11`	Used for linking
`%rsi`	Argument #2	`%r12`	C: Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

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)

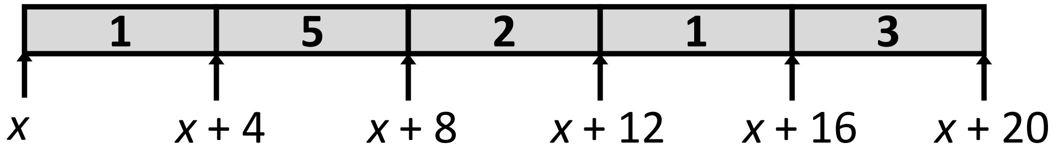


  - Sometimes: Function call using `jmp` (instead of `call`)
  - **Reason: Performance**
    - **use stack as little as possible**
    - **while obeying rules (e.g., caller/callee save registers)**

Last Time

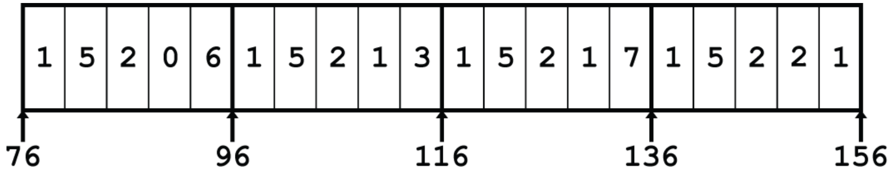
- Arrays

  `int val[5];`



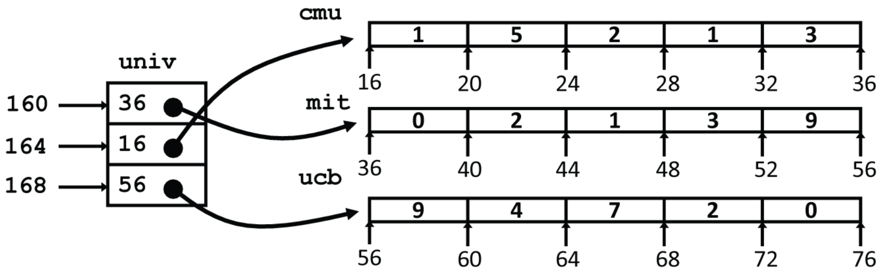
- Nested

  `int pgh[4][5];`



- Multi-level

  `int *univ[3]`



Dynamic Nested Arrays

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

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

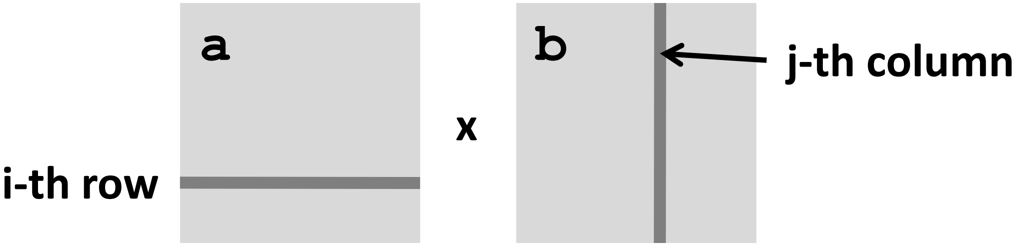
```
int var_ele
  (int *a, int i, int j, int n)
{
  return a[i*n+j];
}
```

```
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

```
/* 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;
}
```



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

4 adds, 3 mults

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

4 adds, 1 mult

```
{
  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;
}
```

Today

- Structures
- Alignment
- Unions
- Floating point

Structures

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

Memory Layout

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

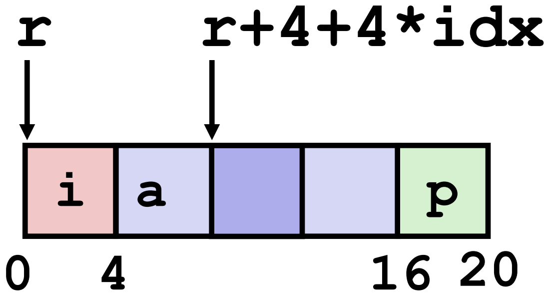
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32 Assembly

```
 # %eax = val
 # %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Structure Member

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



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

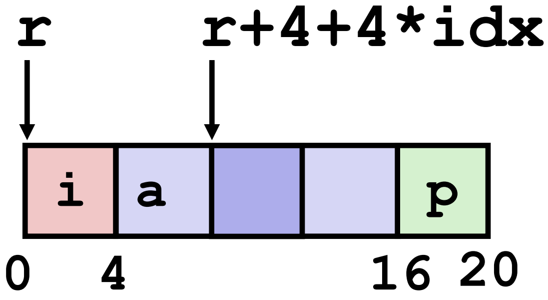
What does it do?

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

Will disappear blackboard?

Generating Pointer to Structure Member

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



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

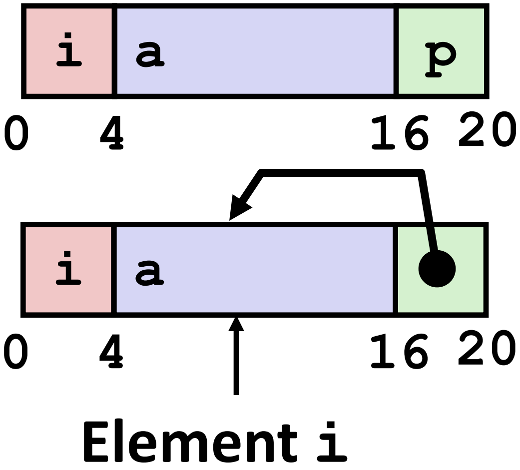
```
int *find_a
  (struct rec *r, int idx)
{
   return &r->a[idx];
}
```

```
 # %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

```
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?

```
 # %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

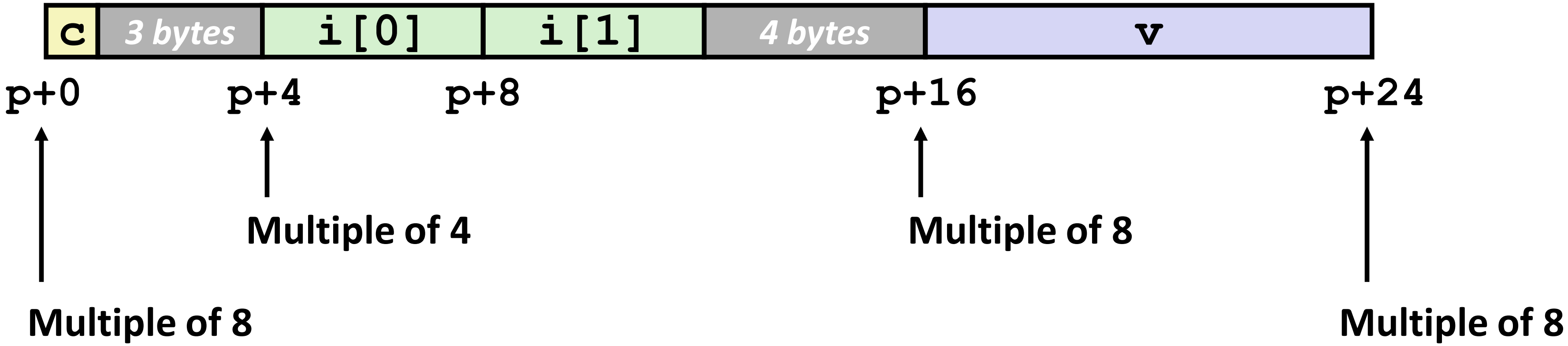
Specific Cases of Alignment (x86-64)

- **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

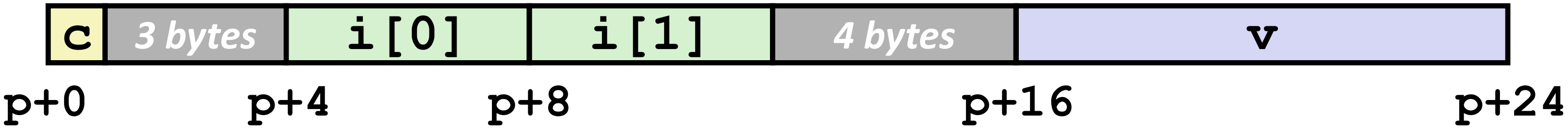
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



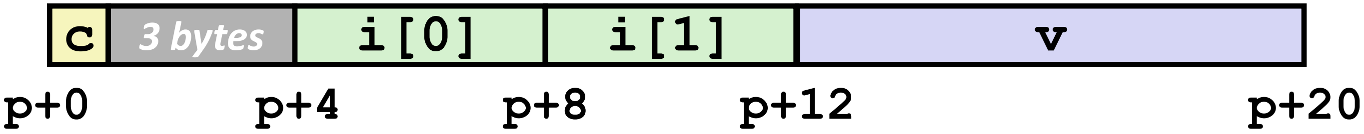
Different Alignment Conventions

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

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



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



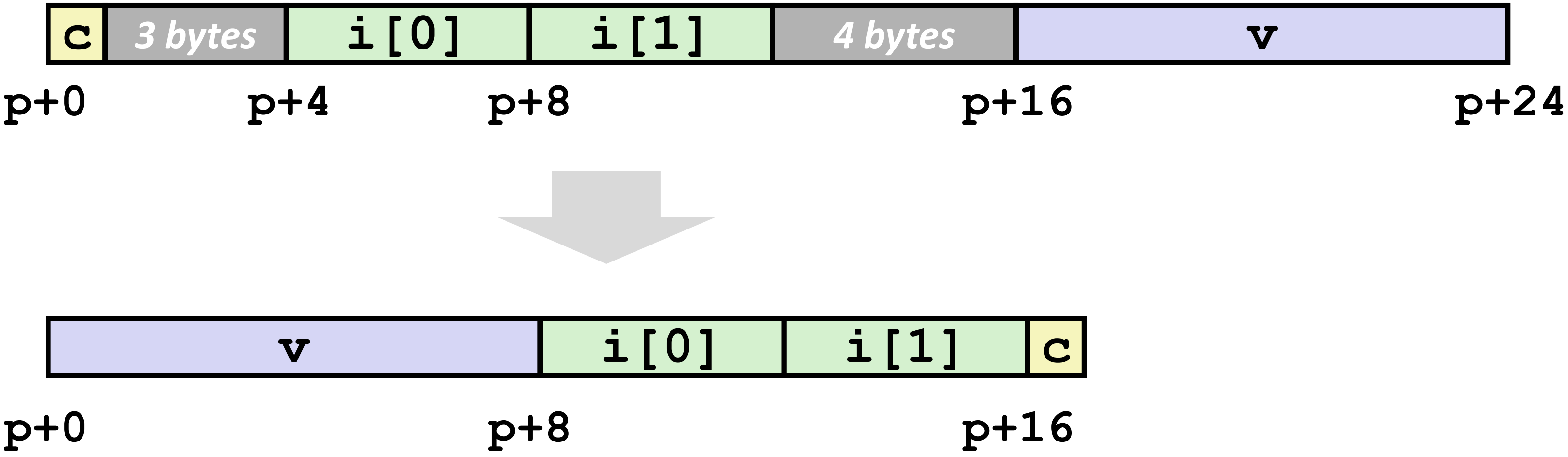
Saving Space

- 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;
```

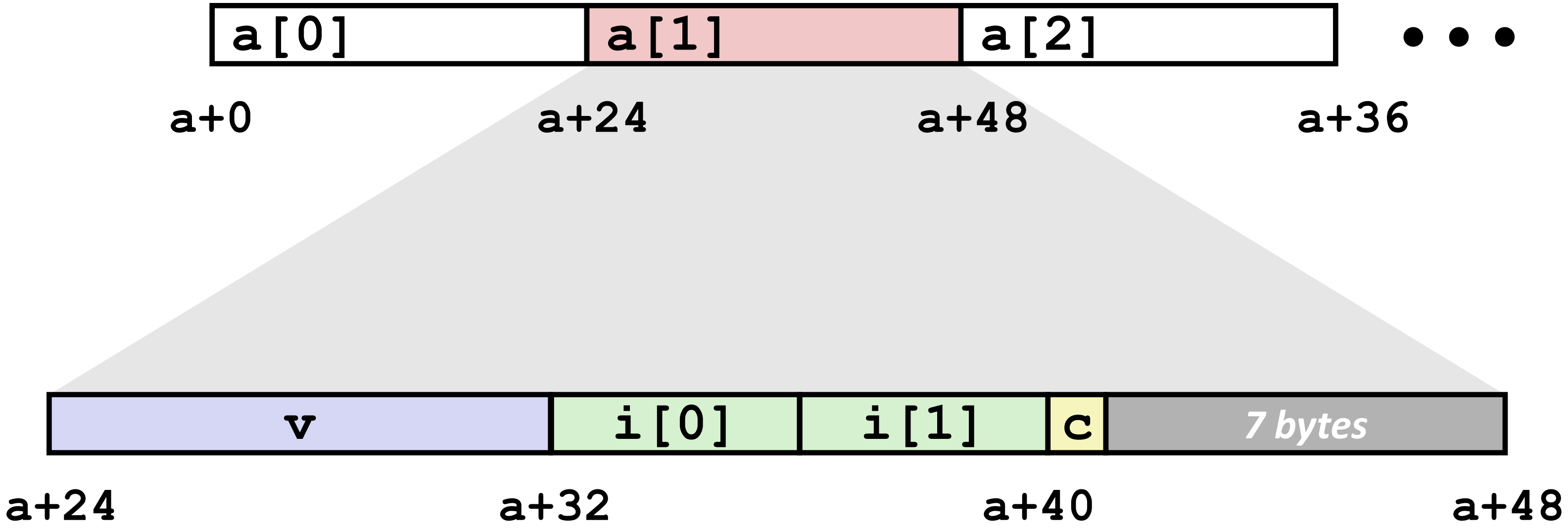
- Effect (example x86-64, both have *K=8)*



Arrays of Structures

- Satisfy alignment requirement for every element

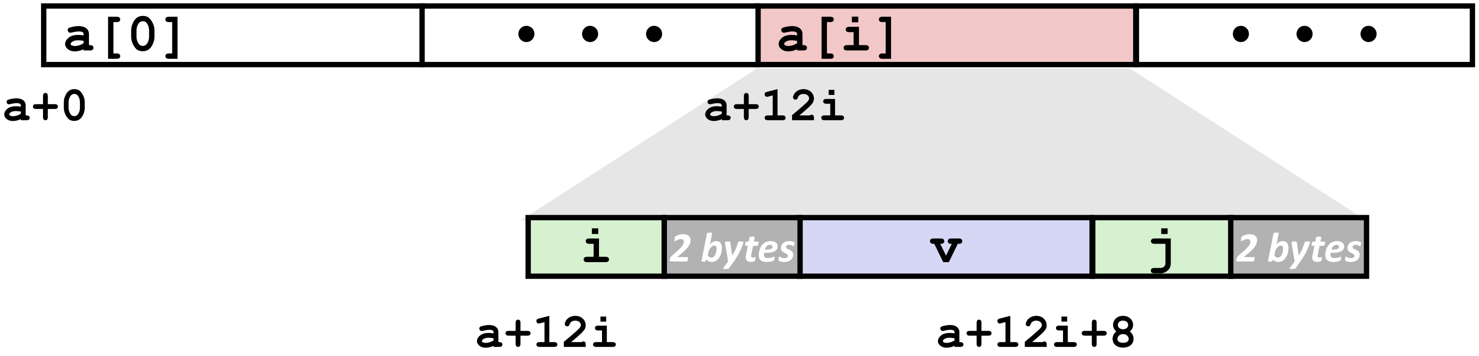
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

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



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

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

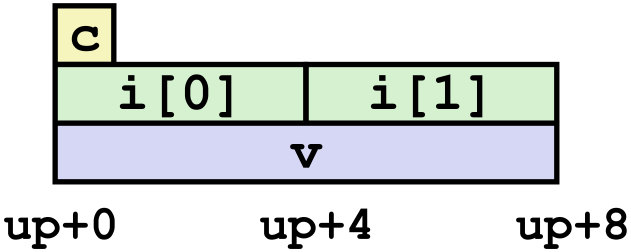
Today

- Structures
- Alignment
- **Unions**
- Floating point

Union Allocation

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

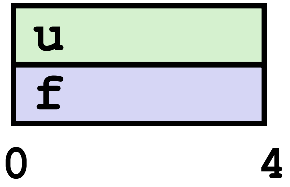
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



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

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` ?

Summary

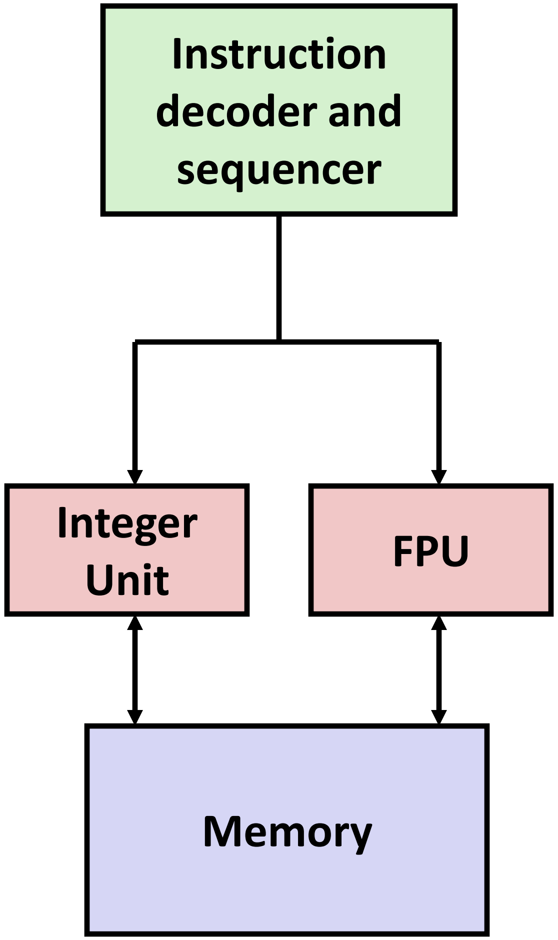
- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - 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)

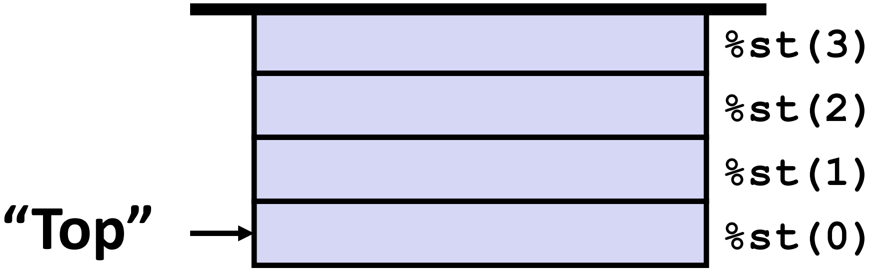
- **History**
  - 8086: first computer to implement IEEE FP
    - separate 8087 FPU (floating point unit)
  - 486: merged FPU and Integer Unit onto one chip
  - Becoming obsolete with x86-64
- **Summary**
  - Hardware to add, multiply, and divide
  - Floating point data registers
  - Various control & 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:**

Instruction	*Effect*	*Description*
`fldz`	`push 0.0`	Load zero
`flds` *Addr*	`push` Mem[*Addr*]	Load single precision real
`fmuls` *Addr*	`%st(0)` ← `%st(0)*`M[*Addr*]	Multiply
`faddp`	`%st(1)` ← `%st(0)+%st(1);pop`	Add and pop

FP Code Example (x87)

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

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

Inner Product Stack Trace

```
eax = i
ebx = *x
ecx = *y
```

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`

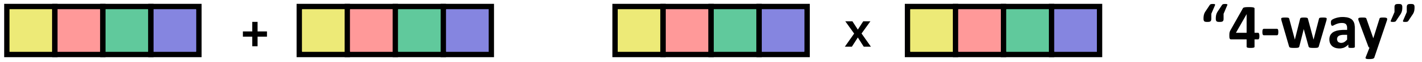
x[0]*y[0]+x[1]*y[1]	%st(0)

Today

- Structures
- Alignment
- Unions
- **Floating point**
  - x87 (available with IA32, becoming obsolete)
  - SSE3 (available with x86-64)

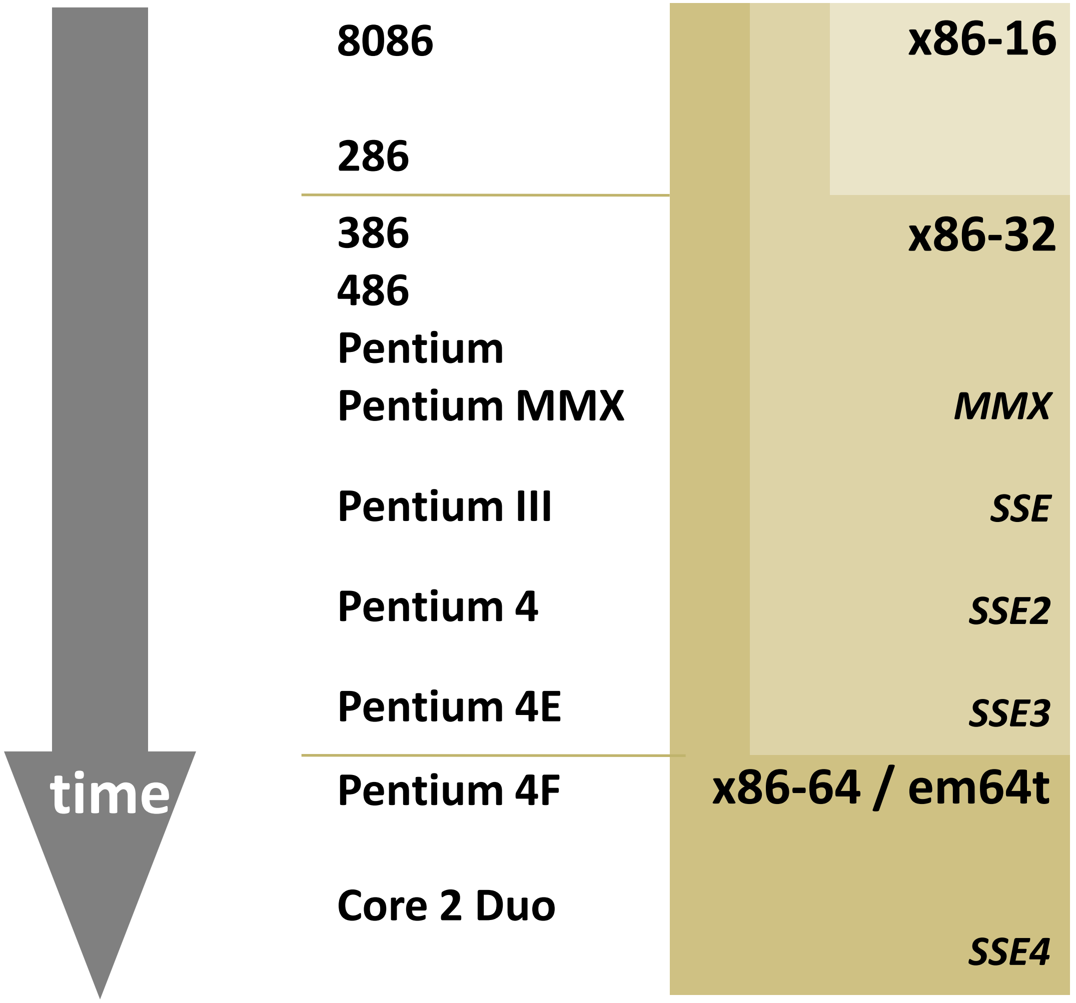
Vector Instructions: SSE Family

- **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)



Processors	Architectures	Features
8086	x86-16	
286		
386	x86-32	
486		
Pentium		
Pentium MMX	*MMX*	
Pentium III	*SSE*	*4-way single precision fp*
Pentium 4	*SSE2*	*2-way double precision fp*
Pentium 4E	*SSE3*	
Pentium 4F	x86-64 / em64t	
Core 2 Duo		
	SSE4	

Our focus: SSE3
used for scalar (non-vector)
floating point

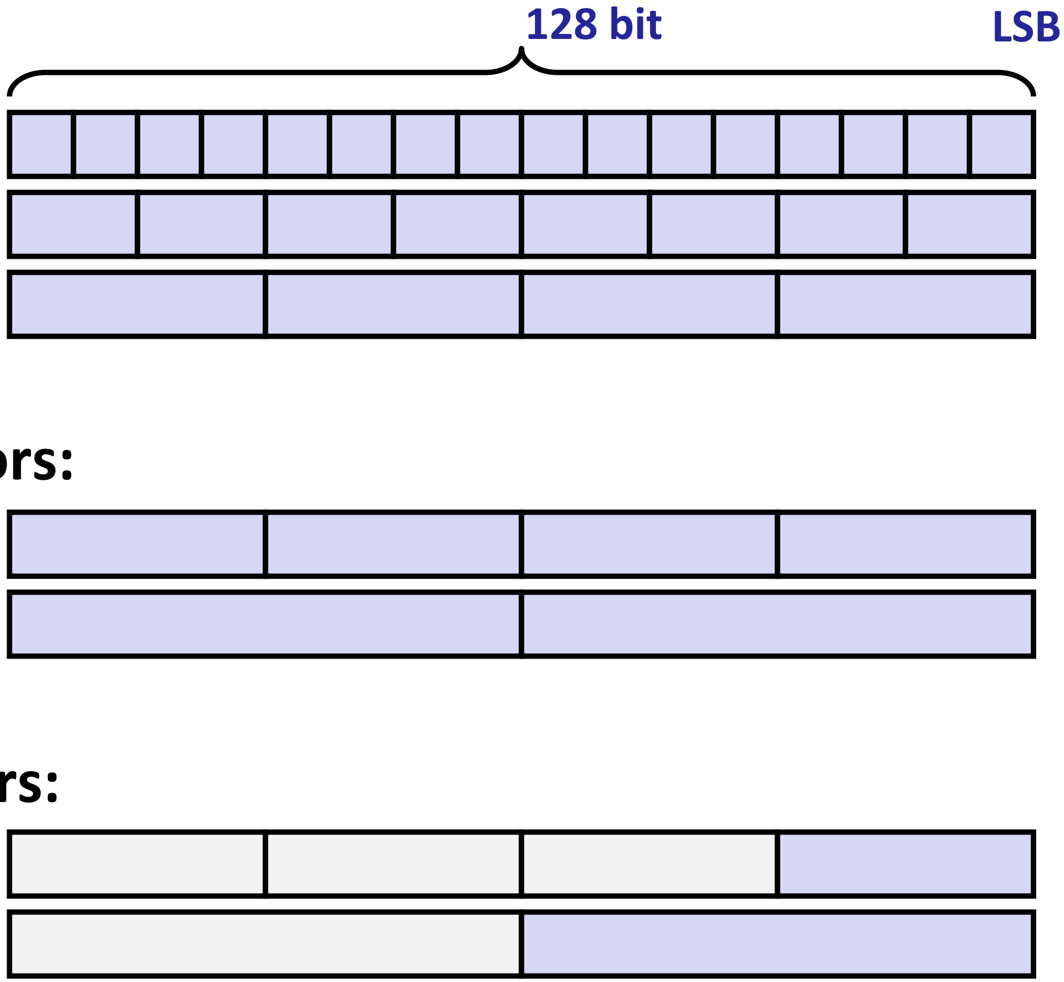
SSE3 Registers

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

128 bit = 2 doubles = 4 singles

`%xmm0`	Argument #1	`%xmm8`
`%xmm1`	Argument #2	`%xmm9`
`%xmm2`	Argument #3	`%xmm10`
`%xmm3`	Argument #4	`%xmm11`
`%xmm4`	Argument #5	`%xmm12`
`%xmm5`	Argument #6	`%xmm13`
`%xmm6`	Argument #7	`%xmm14`
`%xmm7`	Argument #8	`%xmm15`

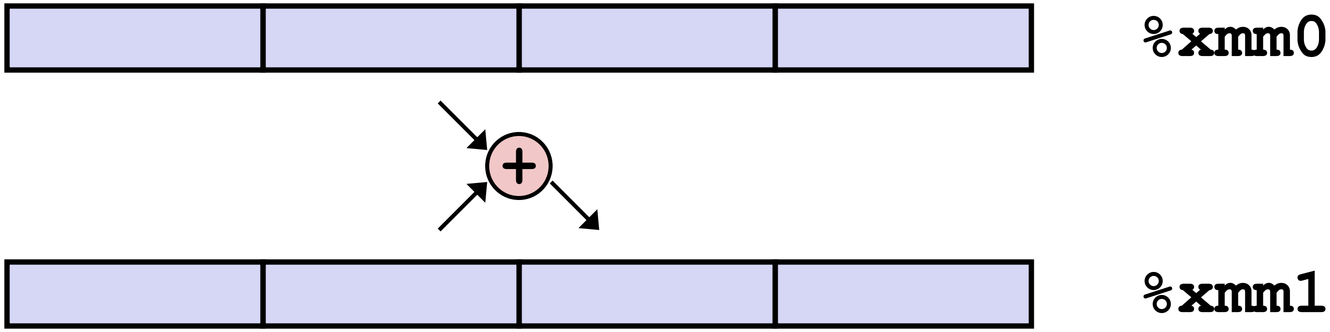
SSE3 Registers



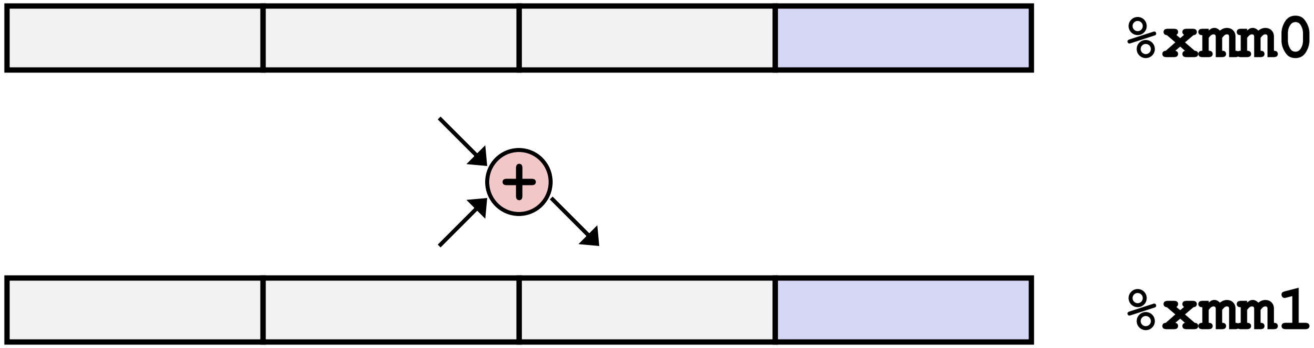
- Different data types and associated instructions
- Integer vectors:
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
- Floating point vectors:
  - 4-way single
  - 2-way double
- Floating point scalars:
  - single
  - double

SSE3 Instructions: Examples

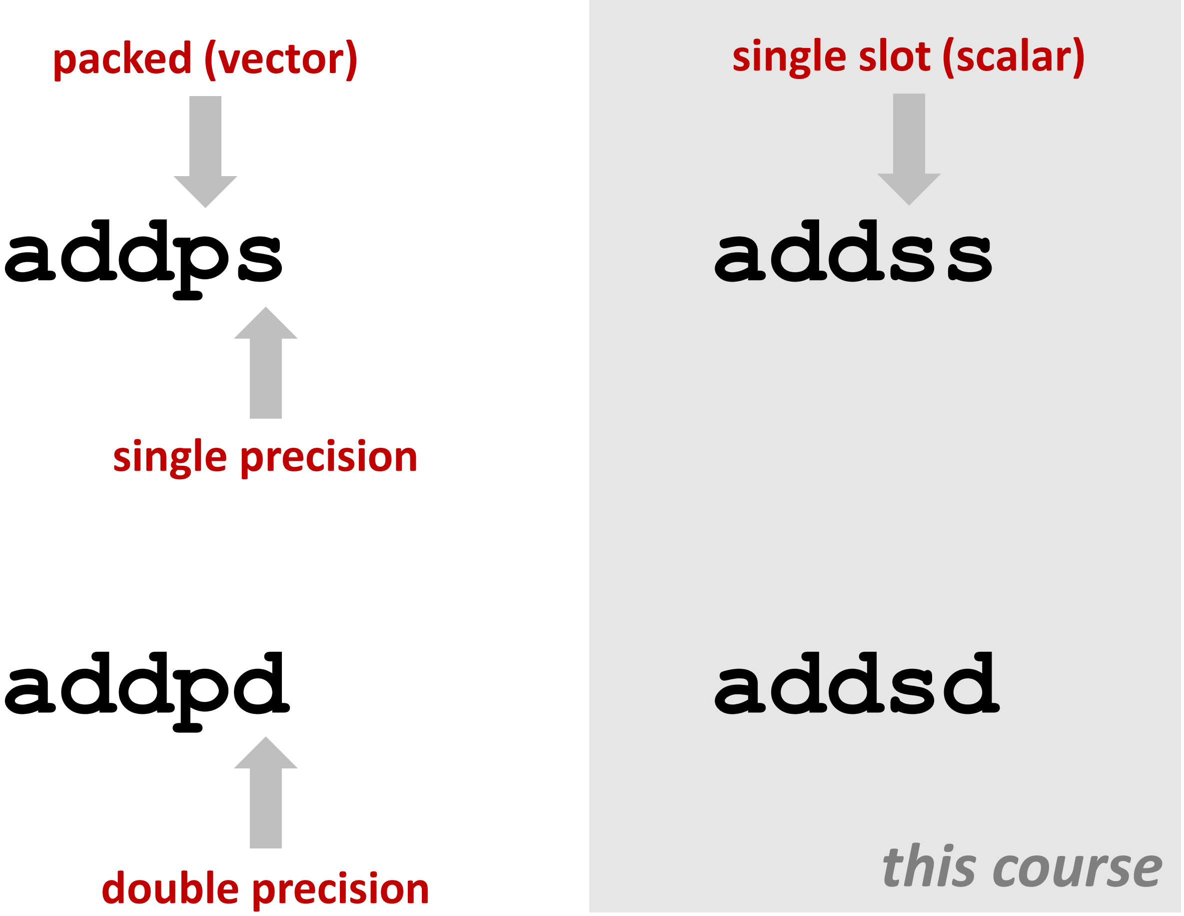
- **Single precision 4-way vector add:** `addps %xmm0 %xmm1`



- **Single precision scalar add:** `addss %xmm0 %xmm1`



SSE3 Instruction Names



packed (vector)

`addps`

single precision

`addpd`

double precision

single slot (scalar)

`addss`

`addsd`

this course

SSE3 Basic Instructions

- **Moves**

Single	*Double*	*Effect*
`movss`	`movsd`	D ← S

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

- **Arithmetic**

Single	*Double*	*Effect*
`addss`	`addsd`	D ← D + S
`subss`	`subsd`	D ← D – S
`mulss`	`mulsd`	D ← D x S
`divss`	`divsd`	D ← D / S
`maxss`	`maxsd`	D ← max(D,S)
`minss`	`minsd`	D ← min(D,S)
`sqrtss`	`sqrtsd`	D ← sqrt(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;
 float result = 0.0;

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

```
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

```
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:                            # loop:
   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:                             # middle:
   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

Instruction	*Description*
`cvtss2sd`	single → double
`cvtsd2ss`	double → single
`cvtsi2ss`	int → single
`cvtsi2sd`	int → double
`cvtsi2ssq`	quad int → single
`cvtsi2sdq`	quad int → double
`cvttss2si`	single → int (truncation)
`cvttsd2si`	double → int (truncation)
`cvttss2siq`	single → quad int (truncation)
`cvttss2siq`	double → quad int (truncation)

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
```

Will disappear
Blackboard?

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;
}
```

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

```
# 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
```

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 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

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