Computer Architecture
Machine Representation of Programs: Structures, Unions, and Floats

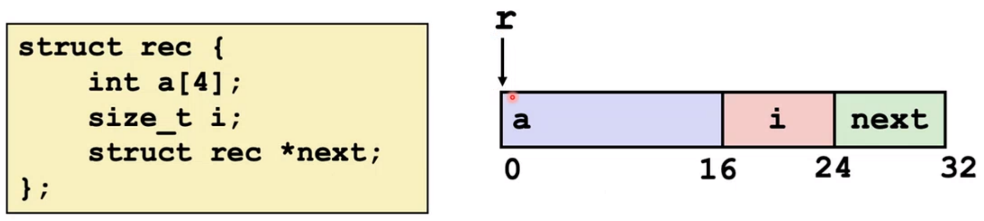
Cain Susko

Queen's University
School of Computing

February 28, 2022

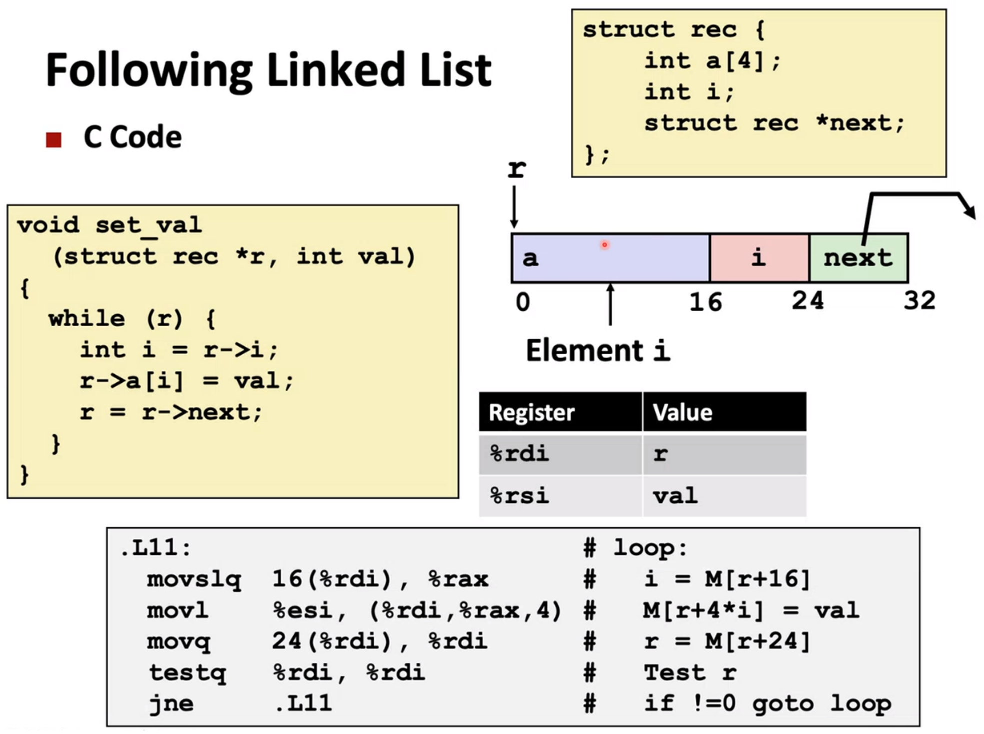
Structures

A structure is represented as a block of memory which is big enough to hold all its fields. Fields are ordered by declaration and the compiler determines the overall size and position of them.



```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

An example of a struct is a Linked List:



Following Linked List

- C Code

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

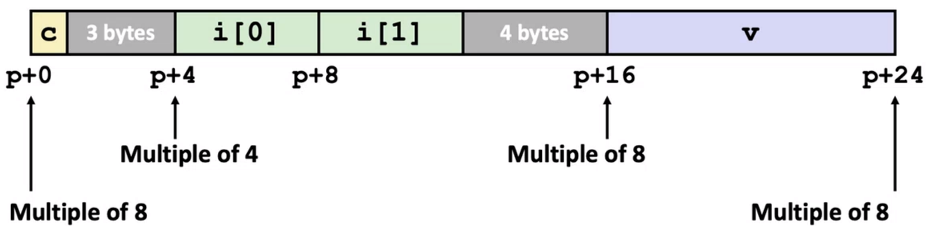
r
a
i
next
0
16
24
32
Element i

Register	Value
%rdi	r
%rsi	val

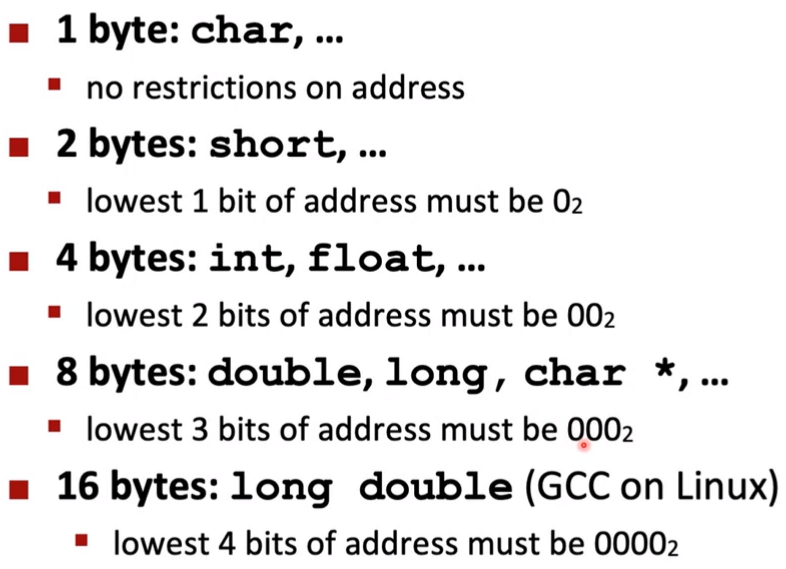
```
.L11:                              # loop:
  movslq  16(%rdi), %rax          #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)     #   M[r+4*i] = val
  movq    24(%rdi), %rdi          #   r = M[r+24]
  testq   %rdi, %rdi              #   Test r
  jne     .L11                    #   if !=0 goto loop
```

Structures & Alignment

An unaligned structure is a structure like in the previous example, where the amount of allocated space is not uniform. A struct with Aligned data is one where each field is a multiple of a given number. (typically 2, 4, or 8)



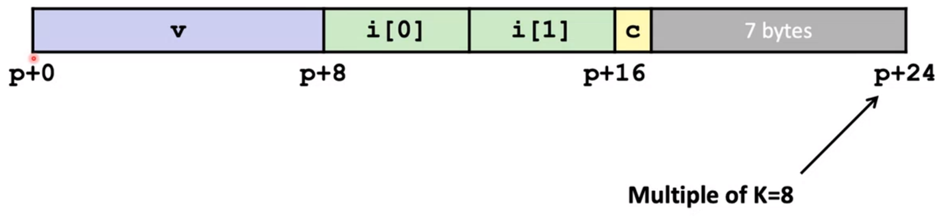
The reason this is done is that, without alignment, it is very difficult to read and write to a struct. Note, the grey blocks are added by the compiler to 'pad' the smaller *int* datatypes. Alignment is required on some machines but is only advised for x86-64. The alignment rules for the primitive types of C are:



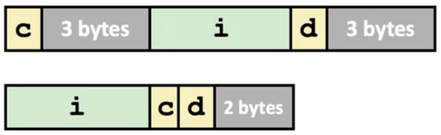
- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2
- **16 bytes: `long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

Note: the number K that is chosen to be the multiple for the rest of the fields is the number of bytes in the largest field. (see the above example with `v`)

Changing the order of the fields in the structure declaration changes their position in memory. The convention is to declare the largest field first and progress to the smallest. This can be seen in the reordering of the previous example:



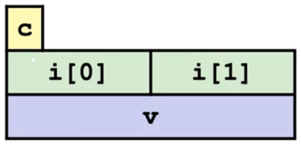
This convention can (but not always) result in less space in memory being used by the structure.



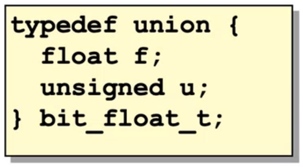
Unions

A structure is always contiguous, whereas a **Union** is not. Space is allocated according to the largest element in the Union like so:

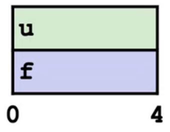
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

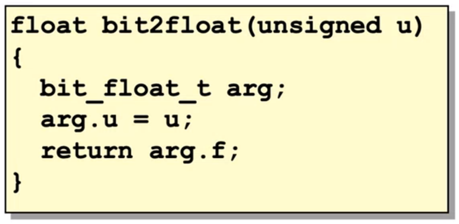


As an example, below is a program that uses a union to access the bit pattern of a float:



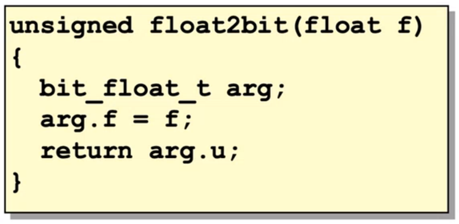
```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```





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

Same as (float) u ?



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

Same as (unsigned) f ?

In essence, a union stores 1 sequence of bits and calling its fields define how the stored sequence should be interpereted.
The common usage of this datatype is to translate between machines / architectures. It is also very useful for switching between signed and unsigned ints.

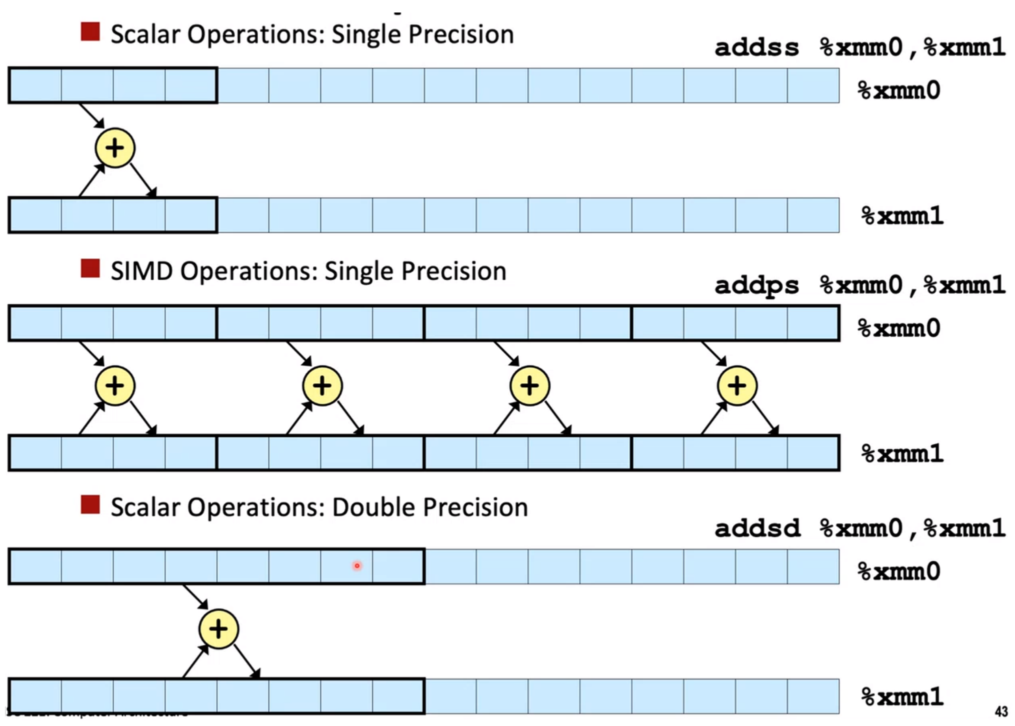
Floating Point

Historically, there have been many ways of storing Floats.

- x87 FP - very ugly & old
- SSE FP - supported by x86 since 2000. Particularly good with vectors
- AVX FP - hot off the presses! Similar to SSE

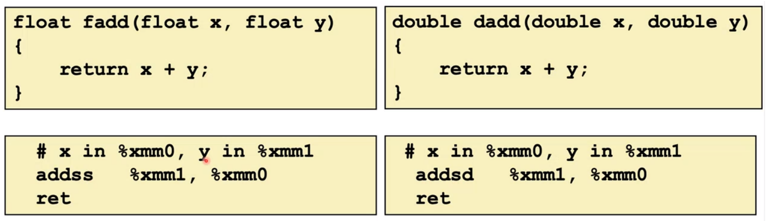
Within SSE there are 16 registers (`%xmm0,...,%xmm15`) to store information. Each register can hold anywhere from up to 16 single byte ints to up to 2 double precision floats.

Operations are applied to floats like so:


Scalar Operations: Single Precision
addss %xmm0,%xmm1
%xmm0
%xmm1
SIMD Operations: Single Precision
addps %xmm0,%xmm1
%xmm0
%xmm1
Scalar Operations: Double Precision
addsd %xmm0,%xmm1
%xmm0
%xmm1
43

Float Basics

Within floating point operations, arguments are passed as `%xmm0,...,%xmmN`. The results are returned in `%xmm0`. Note that all `XMM` registers are caller saved.



```
float fadd(float x, float y)
{
    return x + y;
}
```

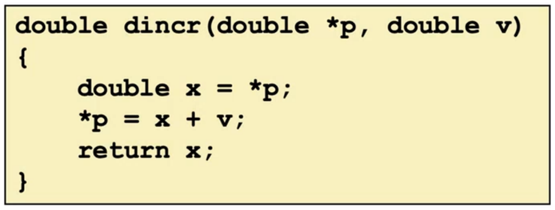
```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

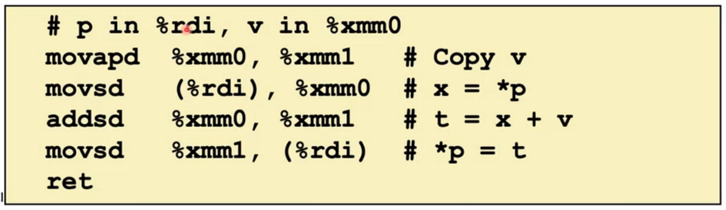
```
  # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

Float Memory Referencing

in FP arithmetic, int and float arguments are passed in regular registers. FP values are passed in XMM registers. To move data between the two, there are specific `mov` instructions for doing so.



```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```



```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1    # Copy v
  movsd   (%rdi), %xmm0   # x = *p
  addsd   %xmm0, %xmm1    # t = x + v
  movsd   %xmm1, (%rdi)   # *p = t
  ret
```

more info about Floating Points There are alot of different operations and formats of operations for FP's so it can get complicated. Comparisons between FP's are done with `ucomiss, ucomisd` but the flags used (CF, ZF, PF) are the same as previously covered. There are also ways of making an XMM register be constant.