
Unions

```c
struct A {
    int a;
    int b;
    int c;
};
```

...\[\text{sizeof}(A) = 12;\]
\[\text{align}(A) = 4;\]

```c
sizeof(union A) = 4;
align(union A) = 4;
```

Unions are dangerous: easy to achieve nasal demons. Think of manipulating unions as manipulating memory with casts.

```c
union A {
```
int a;
char c[47];

sizeof(A) = 48; //char array padded by 1 byte
align(A) = 4;

Assembly Control Flow
i.e. Complicated Data Access

f24.s

f:
    movl a, %eax
    movl (%eax), %eax
    ret

Parenthesis in ASM mean dereference.

f25.s

f:
    movl a, %eax
    movzbl (%eax), %eax
    ret

%eax is a scratch register. Even programs that don’t return a value can use it. If a program does something useless with %eax, it probably does not return a value.

In f25.s, we use %eax, and don’t store it in a global or anything, so it probably returns. a unsigned char*, return value likely unsigned char or unsigned.

Moves the 4 bytes of a into the register, then dereference the first byte pointed to by %eax and stores it in %eax, then return.

movzbl: b = byte z = fill with 0 In C:

extern signed char* a;

int f(void) {
    return a[0];
movsbl = sign-extended, movzbl = zero-extended

f :
    movl x, %eax
    movl a, %edx
    movzbl (%edx,%eax), %eax
    ret

    third line means add %eax to %edx and then dereference. In C:

    extern unsigned char* a;
    extern int x;

    unsigned f(void) {
        return a[x];
    }

Digression

War in ’80s, ’90s. CISC-RISC war. x86 is a complex instruction set. Alternate ways to build the machine used smaller instruction sets that do every instruction exactly explicitly. Smaller instruction sets prettier and easier to write a fast machine. No good arguments against RISC, but it lost. Because Intel. Programs that are compiled into CISC are smaller.

f28.s

(base, idx, sz)
base + idx * sz

... 

f :
    movl a, %eax
    movl x, %edx
    movl (%eax,%edx,4), %eax
    ret
In C:

```c
extern int * a;
extern int x;

int f(void) {
    return a[x];
}
```

This simply returns ‘a’.

```asm
f29.s
```

```asm
f:
    movl a, %eax
    ret

$f_a$ returns the address of a.

```asm
f30.s
```

```asm
f:
    movl $a, %eax
    ret

$a$ returns the address of a.

```asm
f32.s
```

```asm
f:
    movl 6161, %eax
    ret

returns the value at the address 0x6161

```asm
f33.s
```

```asm
f:
    movzbl (%eax,%edx,4), %eax
    ret

Looks like dereferencing an int array
```
struct four_bytes {
    unsigned char k;
    unsigned char l;
    unsigned char m;
    unsigned char n;
};
extern struct four_bytes* a;
extern int x;

int f(void) {
    return a[x].k;
}

f34.s

f:
    movl (%eax,%edx,8), %eax
    ret

object very likely to be an array because of the style of dereference.
In C:
struct two_words {
    unsigned k;
    unsigned l;
};
extern struct two_words* a;
extern int x;

int f(void) {
    return a[x].k;
}

f35.s

f:
    movl x, %eax
    sal  $4, %eax
addersl a, %eax
movl (%eax), %eax
ret

separate instructions for each step in the indexing.

dereferencing an array of structures. 4 ints in each structure. If sz in (base, idx, sz) is not 1, 2, 4, or 8, the compiler must write out the arithmetic explicitly.

In C:

```c
struct four_words {
    unsigned k;
    unsigned l;
    unsigned m;
    unsigned n;
};
extern struct four_words* a;
extern int x;
```

```assembly
int f(void) {
    return a[x].k;
}
```

f36.s

```
f:    
    movsbl 3(%eax,%edx,4), %eax

Actual form

off = 0(base, idx = 0, sz = 1)
off + base + idx*sz

a + 4*x + 3 constant 3 comes from asking for the 3rd element in a struct of ints.
```

f37.s

```
f:    
    leal 3(%eax,%edx,4), %eax
```
load effective address: compute the effective address and then don’t dereference it; just move the address into the destination argument.

Exactly the same as f36.s, except return &(thing)

f38.s

Everything is a number to the compiler. It will also use leal with anything that can be most easily computed using that process, even if it’s not an address.

\[
\text{return } a[x].n; \\
// \text{is equivalent to}
\]

\[
\text{return } a + 4x + 3;
\]

f39.s

\[
f: \\
movl x, %eax \quad // \text{load x into } %eax \\
leal (%eax,%eax), %edx \quad // \text{add x to itself, store in } %edx \\
addl a, %eax \quad //
\]

\[
\text{leal } 3(%eax,%edx), %eax \\
\text{ret}
\]

Moving instruction pointer around

f40.s

\[
\text{.LFB0:} \\
... \\
cmpl %edx, %eax \\
jge .L2 \\
movl %edx, %eax \\
.L2: \\
\text{ret}
\]

\[
\text{.LFE0} \quad ... \\
\text{if } (a > b) \\
\text{ret } a
\]
else
    ret b

    Compiler changed the order of things. jge corresponds to else.

f:
    %edx = a
    %eax = b
    if (%edx >= %eax) // ?
        return b;
    else
        return a;

WRONG

cmpl x, y
    // is equivalent to

        subl x, y == y -= x
        + test if result >= 0

f:
    %edx = a
    %eax = b
    if (%eax - %edx >= 0)
        b - a >= 0
        return b;
    else
        return a;

    cmpl does the same subtraction as subl, but throws away the result, with
    the exception of storing metadata about the subtraction in special registers
    called flags. Jump instructions then check the flag registers, so cmpl changes
    those for jge to look at.

    Like cmpl, subl ALSO changes all of the flags, so the compiler will some-
    times use a normal operation like subl.

f41.s

f:
    movl b, %eax
    cmpl x, %eax
jne .L2
  movl a, %eax
.L2:
  ret
jne x, %reg: jump if x not equal to value in %reg
f42.s
je x, %reg: jump if x is equal to value in %reg
f44.s
f:
  cmpl \$0, a
  fs e t e %al
  movzbl %al, %eax
  ret
sete: extracts the equal flag, which is true if a == 0
f45.s
Same
  Note: cannot compare two things from memory. Only understands registers.

f46.s
testl: like cmpl, but with a bitwise & instead. All backward jumps are loops
f:
  testl %eax, %eax
  je .L4 // true if and only if %eax == 0
  // return 0 if x == 0
  ...
  cmpl %ecx, %edx
  jne .L3 // loop if
  // C code
  if (x == 0)
  return 0;
rv = 0; %edx = a;
while (%edx!= &a[x])
{
   rv += *%edx;
   %edx += 4;
}
return rv;
...
int rv = 0;
int *a; int x;
for (int i = 0; i != x; ++i)
   rv += a[i];
return rv;
...
int rv = 0;
int *i = a;
int *end = &a[x];
while (a != end)
{
   rv += *i;
   i++;
}
return rv;

Computes the sum of the elements of an array of ints.
leal does not set flags.

**Local Variables**

Local variables stored on the stack.
   In asm, %esp is the stack pointer.
   At the beginning of a function, there is at least one thing on the stack:
   the return address.
   Even chars are 4-bytes big as stack arguments.

```
unsigned f(unsigned i)
{
```
return  i ;
}

Arguments stored on the stack immediately after the return address.

f48.s

Two arguments. Sum function. Returns the sum of its two arguments.

f49.s

Has 8 arguments, only uses the first two, asm exactly the same as f48.s

f50.s

f :  
push %ebp  
movl %esp , %ebp  
subl $8 , %esp  
call g  
leave  
ret

Need to preserve the value of %ebp.