

Unix x86 Assembly Primer

CS423

Sections

Unix x86 breaks programs into *sections* (a.k.a. segments)

- `.data` - for *global* user data, readable *and* writable
- `.rodata` - for *global* user data, read *only*
- `.text` - for executable code

Declaring a section:

- `.section <type>`
- `<type>` is one of the above

Data Section Example

A very simple data section is as follows:

```
.section .data
.align 4 /* begin at 4 byte boundary */
a: .skip 4 /* associate label a with 4 bytes */
arr: .skip 64
```

The only thing that ends a section is:

- declaring another section
- end of file

Text (Code) Section Example

A very simple text section is as follows:

```
        .section .text
.globl  main
main:
/* comments work just like in C */
    pushl   %ebp
    movl    %esp, %ebp
    ...
    leave
    ret
```

x86 Registers

4 byte (1 word / integer) registers include:

- `%eax` -- general purpose (aka accumulator)
- `%ebx` -- general purpose (aka base)
- `%ecx` -- general purpose (aka counter)
- `%edx` -- general purpose (aka data)
- `%esp` -- top of stack pointer
- `%ebp` -- current ARP
- `%esi` -- source index register
- `%edi` -- destination index register

There are others, but :

- using them has side effects
- you shouldn't need more than the first six above
- (my solution didn't use `%esi` or `%edi`)

x86 Memory References

All x86 memory references have the form:

$\text{immediate}(\text{base}, \text{index}, \text{scale})$

Where:

- *immediate* is a constant
- *base* is an address, as a register
- *index* is the number of units from the base (register)
- *scale* is the size of one unit (number: 1, 2, 4, or 8)
- any of the above may be omitted

and the resulting address is calculated as:

$\text{base} + \text{immediate} + \text{scale} * \text{displacement}$

x86 Memory References

Examples:

- `-8(%ebp)`
 - go 8 bytes “above” the `ebp`
 - default scale is 1
- `(%eax, %ebx, 4)`
 - `eax` might be the base of an array
 - `ebx` the index into array, each element of size 4
- `myvariable`
 - OK to omit `()`'s entirely!
 - looks up a “global variable”

x86 Instructions

Basic rules for an x86 assembly instruction

- data moves from left to right
- no more than 1 memory reference allowed
- rightmost operand cannot be a constant

Simple x86 Instructions

- `movl src, dst` /* dst = src */
- `addl src, dst` /* dst = dst + src */
- `subl src, dst` /* dst = dst - src */
- `negl dst` /* dst = - dst */
- `notl dst` /* dst = bitwise_not(dst) */
- `cmpl src, dst` /* compares dst to src, sets flags */
- `je label` /* jump if flags say equal */
- `jne label` /* jump if flags say not equal */
- `jg label` /* jump if flags say greater */
- `jl label` /* jump if flags say less than */
- `jmp label` /* jump always */
- `nop` /* no operands - do nothing */

Ugly x86 Instructions

Multiplying:

```
imull  src /* only ONE operand! */
```

- always calculates $\%eax * src$
- result may overflow - overflow is stored in $\%edx$
- most compilers ignore overflow data

Dividing - the reverse of multiplying?

```
idivl  src /* again, ONE operand! */
```

- always calculates $\langle \%edx:\%eax \rangle / src$
 - division result in $\%eax$
 - remainder in $\%edx$
 - need a way to setup $\%edx$ with appropriate value
- ```
cddl /* no operands, sign extends %eax into %edx */
```

# Quick Example

calculate  $\%ebx = |\%ecx / (7 * 81)|$

```
movl $7,%eax /* $ means 7 is literal, not memory ref. */
```

```
movl $81,%ebx
```

```
imull %ebx /* eax=7*81 */
```

```
movl %eax,%ebx
```

```
movl %ecx,%eax /* setup half of numerator */
```

```
ctdl /* setup edx for division (other half) */
```

```
idivl %ebx /* eax = ecx / (8*71) */
```

```
cmpl 0,%eax
```

```
jg done /* if eax > 0 (from cmpl), jump */
```

```
negl %eax
```

```
done:
```

```
nop
```

# x86 “Function” Instructions

`pushl src /* only ONE operand! */`

- pushes value of `src` onto stack, i.e.  
`movl src, -4(%esp)`  
`subl $4, %esp`
- can be used for parameter passing

`call label`

- pushes return address onto stack and jumps

`ret /* no operands */`

- pops return address off of stack into `%eip`

# Actually ~~Compile~~ Assembling

Simply use gcc:

- make sure assembly code filename ends in “.s”
- `gcc -gstabs -o run_me file.s`
- the “-gstabs” allows use of the debugger (gdb / ddd)
- that’s “all” there is to it ...