Bootsector (512 byte) x86 programs

Recently I wrote an x86 assembler in 512 bytes of machine code: https://github.com/kvakil/0asm. This is called a "bootsector" program, because it fits in a typical hard-drive sector. This zine will give you pointers on writing bootsector x86 programs of your own, assuming familiarity with x86 assembly.

x86 resources I found useful:

- 80x86 is an octal machine: https://bit.ly/2OYLHLI Good pseudocode, and information about the ISA.
- 80386 reference manual: https://bit.ly/2OELlLr (particularly Chapter 17, and the appendices)

Ten Tips:

1. **Baby's first bootsector:** You could write a bootsector from scratch, but I've made a bootsector skeleton to extract common scaffolding: https://github.com/kvakil/boot-skel
   It also provides some nice debugging features–see the repository for details!

2. **Use registers for their purpose:**

- `sp`: use as stack pointer, too good to pass up.
- `ax`: aim for comparisons to operate on `ax`, many instructions are shorter when they use `ax`. (For example, `cmp ax,1` is shorter than `cmp bx,1`.)
- `cx`: useful as a loop counter: see `loop`, `rep`, and `jcxz` instructions.
- `si`, `di`: use as source and destination of memory accesses (respectively): used by some instructions, especially `lodsb`, `stosb`, ... Prefer these to `mov` since they are shorter and increment the pointers!
- `bx`, `bp`: can be used for addressing, like `mov ax,[bx]`. Generally `bx` is better than `bp` because the common zero offset case like `mov ax,[bx+0]` is a byte shorter than `mov ax,[bp+0]` (Table 17-2 in manual).
- `dx`: used by `div` and `mul`, otherwise not useful.

3. **Know the instruction set:** here is a non-exhaustive tier ranking of instructions you probably haven't seen.

- Useful: `lodsb`, `stosb`, `inc`, `dec`, `xchg`.
- Sometimes useful: `cbw`, `scasb`, `movsb`, `loop`, `stc`, `clc`, `neg`, `not`, carry flag stuff like `adc`.
- Usually useless: anything else (especially BCD instructions like `aaa`).

4. **Use `FLAGS`:** almost all instructions affect the `FLAGS` register (Appendix C of manual). Because conditional jumps rely on `FLAGS`, aim to have your function return boolean results in `FLAGS`. Instructions like `stc` let you manipulate flags manually, but try to have your code correctly modify `FLAGS` without them to save bytes.

5. **Forget calling conventions:** you should think of functions as common snippets of code which may affect many registers. Using any "result" register may be useful. Any time you use `push`/`pop` or `leave` should be suspect. This also typically makes it easier to reuse code snippets between functions.

6. **Know the idioms:** there are many "peephole" optimizations possible, I'll just list the ones I find most useful. The best way to find them is by reading through other people's code or by mucking around with the instruction set.

- *Zeroes:* rather than `mov ax,0` (3 bytes) use `xor ax,ax` (2 bytes). Similarly instead of `cmp ax,0` (3 bytes) use `test ax,ax` (2 bytes).
- *Prefer `xchg` to `mov`:* If you are moving a register to or from `ax`, consider using `xchg` (1 byte) instead of using `mov` (2 bytes). This is also useful since some instructions must use `ax` or have shorter encodings when they do.
- *Shifts to multiply:* You can use bitshifting to multiply or divide by powers of two.
- *Set register to -CF:* `sbb bl,bl` (2 bytes) sets `bl = -carry flag`. If you are OK with setting `al` instead of another register, you can use the undocumented instruction `salc` (1 byte). Since `-1` has all bits set and `0` has no bits set (in two's complement), this is useful as a bitmask.
- *Tail call*: if you have `call F` & `ret` (typically at the end of a function), you can replace this with just `jmp F`, saving up to two bytes. You can also remove the `jmp` completely by moving `F` inline, but of course you can only inline once.

7. **Beware `rel16` jumps:** if you jump farther than 127 bytes, you use a long jump (costing an extra byte), and your assembler won't tell you! Check the assembly listing to monitor for these, and reorder your code as appropriate.

8. **Beware unconditional jumps:** jumping using a condition (like `jc`) doesn't cost more than jumping unconditionally. Unconditional jumps with a nearby conditional jump are usually a sign that the code can be refactored to a single conditional jump (typically by rewriting a `while`-loop as a `do-while`-loop).

9. **Self-modifying code:** rarely useful, but very cool when it works. One use is global variables (saving one byte over the naïve solution). For example, this creates a counter starting at 1234:

```
LA: mov ax,1234 ; initial value is 1234
    ; LA+1 is the address of 1234
    mov bx,LA+1
    ; use ax as counter value
    inc word [bx] ; increment counter
```

10. **Ignore these rules:** these are just guidelines which I've found *typically* reduce code size. It's very difficult to write a small program, so all of these are really just heuristics. Happy hacking!