1. Skeleton bootloader
   1. org: 0x7c00
   2. bits: 16
   3. Core: cli hlt
   4. Padding
   5. Magic number 0xaa55
2. Add OEM Parameter Block:

TIMES 0Bh-$+start DB 0

bpbBytesPerSector: DW 512

bpbSectorsPerCluster: DB 1

bpbReservedSectors: DW 1

bpbNumberOfFATs: DB 2

bpbRootEntries: DW 224

bpbTotalSectors: DW 2880

bpbMedia: DB 0xF0

bpbSectorsPerFAT: DW 9

bpbSectorsPerTrack: DW 18

bpbHeadsPerCylinder: DW 2

bpbHiddenSectors: DD 0

bpbTotalSectorsBig: DD 0

bsDriveNumber: DB 0

bsUnused: DB 0

bsExtBootSignature: DB 0x29

bsSerialNumber: DD 0xa0a1a2a3

bsVolumeLabel: DB "MOS FLOPPY "

bsFileSystem: DB "FAT12 "

1. Implement function to print a string
   1. Call int 0x10
      1. ah: Output type. Set to 0x0e for teletype.
      2. al: The ASCII to be printed
   2. Put the message in si.
      1. lodsb loops through the message and puts the current character into al
      2. Quit when the current character/al has ASCII value 0.
   3. Before calling print, first null ds ad es
      1. ds and es don't accept immediates or other segment registers, so mov another zeroed general register into them.
      2. We can't do something like xor ds, ds either, because ds itself is a segment register, so it does not accept itself. Same goes for all other segment registers.
   4. After calling print, continue with the "cli hlt" infinite loop
2. Get amount of KB from BIOS
   1. Call int 0x12
      1. Clear ax before calling
3. Create stage 2 bootloader:
   1. org: 0
   2. bits: 16
   3. No OEM Parameter Block
   4. Set ds to cs
      1. mov ds, cs doesn't work, because ds and cs are both segment registers, and x86 does not support moving one segment register into another.
      2. In other words: no moving an immediate or a segment register into a segment register.
      3. Set ds to cs by pushing cs and popping to ds.
         1. Alternatively, use a general register such as ax as the intermediate register.
4. Housekeeping:
   1. cli
   2. Recall we load the stage 1 bootloader at 0x7C00. Under segment:offset, the segment registers must be set at 0x07C0.
      1. Notice this is 7C00 shifted 1 hex (4 bits) to the right.
      2. mov 0x07C0 into ds, es, fs, and gs. Use an intermediate register, say ax.
   3. Create the stack:
      1. Set ss (stack base) to 0. Again, use an intermediate register.
      2. Set sp (stack top) to 0xFFFF. No need for intermediate registers, sp accepts immediates.
   4. sti to restore interrupts
5. Load stage 2 bootloader:
   1. Utility: implement LBACHS
      1. Input:
         1. ax: sector number in LBA. The goal is to convert this into CHS format
      2. Recall CHS: Sector on head, head on track
      3. Absolute sector number on track:
         1. Get the remainder of the LBA sector number when divided by bpbSectorsPerTrack
         2. Use the div instruction
            1. div uses ax as the implicit dividend. It puts the quotient in ax (overriding the dividend) and the remainder in dx. In our case the remainder is only big enough to fill dl.
            2. Clear out dx before running div.
         3. Add 1 to dl because absolute sectors are 1-indexed.
         4. Move dl to [absoluteSector]. absoluteSector is a db because the 1-byte dl register is enough to hold the sector number.
      4. Absolute head and track numbers:
         1. After calculating the absolute sector number, ax still holds the division quotient, representing the number of *complete* tracks that come before our input sector number.
         2. Since tracks and heads are 0-indexed, ax already holds the track number we want. Do not clear out ax; clear out dx only.
         3. div ax by bpbHeadsPerCylinder
            1. Recall that bpbHeadsPerCylinder represents the number of heads per track
            2. The quotient of this division (held in ax) is the absolute track number, and the remainder (held in dx) is the absolute head number.
            3. Once again, the quotient and remainder are only big enough to fill al and dl respectively.
         4. mov dl to [absoluteHead], and al to [absoluteTrack]. Both are dbs like absoluteSector, for the same reason.
   2. Utility: implement ReadSectors:
      1. Inputs:
         1. ax: starting sector. In LBA format
         2. cx: number of sectors to read
         3. es:bx: buffer to read to
      2. Main loop:
         1. Allow, say, five tries per sector before giving up.
            1. Use di as the "remaining tries" counter
      3. Sector loop:
         1. This is part of the main loop.
            1. Upon successfully reading a sector, jump to the main loop so we have five tries for the new sector.
            2. Upon failing to read a sector, jump back to the sector loop, now with one fewer try.
         2. Push ax, bx, and cx
         3. Call LBACHS:
            1. Recall that LBACHS requires ax to hold the LBA sector number, and we already meet this requirement.
            2. LBACHS fills out absoluteSector, absoluteHead, and absoluteTrack
         4. Now that we know the CHS address we can call BIOS int 0x13 to read a sector.
            1. Inputs

ah: Function mode. Set to 0x02 for reading a sector.

al: Number of sectors to read. Set to 0x01 because we are only reading one at a time.

ch: Lower eight bits of the cylinder number. For our purposes this is just the entire cylinder number. Set to [absoluteTrack]

cl: Absolute sector number. Set to [absoluteSector]

dh: Head number. Set to [absoluteHead]

dl: Drive number. Set to [bsDriveNumber]

* + - * 1. Call int 0x13 once all these registers have been set up
      1. 0x13 sets the CF flag upon failure, and clears the flag upon success.
      2. jnc to the success-handling code:
         1. mov the success message into si and call the print string function
         2. Pop cx, bx, and ax
         3. Add [bybBytesPerSector] to bx.

Recall that es:bx is the buffer location. We are in effect moving the buffer pointer one sector over, because we are about to read the next sector.

* + - * 1. Inc ax.

Recall that ax holds the LBA sector to read. This sets us up for reading the next sector.

* + - * 1. loop to main

The "loop" instruction uses cx as the loop counter. This is why we use cx to hold the number of sectors to read.

* + - * 1. ret
      1. Error-handling code:
         1. Call BIOS int 0x13 to reset the disk

ah: Function number. Set to 0 for disk reset.

In other words, clear out ax then call int 0x13.

* + - * 1. dec di. Recall di holds the "number of tries left".
        2. Pop cx, bx, and ax
        3. If we still have tries left, jump back to sector loop

Use the jnz instruction

* + - * 1. Otherwise call int 0x18

No registers required.

0x18 signals to the BIOS that the boot failed, and lets the BIOS handle the cleanup.

* 1. Load the root directory table
     1. The goal is to call ReadSectors. Recall the registers we need:
        1. ax: Sector location.
        2. cx: Number of sectors to read
        3. es:bx: buffer location
     2. For ax and cx, we need to know the size and starting location of the root directory table, measured in sectors.
     3. Size:
        1. 32 bytes per entry (0x0020 in hex).
        2. Number of entries in the root directory table is bpbRootEntries
        3. 1) x 2) gives us the size of the root directory table in bytes, but we need the size in sectors.
        4. Number of bytes per sector is bpbBytesPerSector.
           1. Use the div instruction: div WORD [bpbBytesPerSector]
        5. Put the size in cx.
     4. Starting address:
        1. The root directory table starts after reserved sectors (including the boot sector) and the two FATS.
        2. Number of FATs: bpbNumberOfFATs
        3. Sectors per FAT: bpbSectorsPerFAT
        4. Size of reserved sectors, including the boot sector: bpbReservedSectors
        5. Put the starting address in ax
     5. One last thing before loading: we'll need the data sector's starting address for when we actually load the stage 2 bootloader.
        1. Add both ax and cx to the 0-initialized [datasector] (size: 1 word).
     6. At this point, ax and cx are ready for ReadSectors. It remains to prepare es and bx, which hold the buffer location.
        1. Let's use 7C00:0200 as the buffer location.
        2. es has already been set at 7C00 earlier in the bootloader, so we don't have to do anything here.
        3. Set bx to 0x0200.
     7. Now call ReadSectors
  2. First make sure the stage 2 bootloader even exists on the floppy. Find the stage 2 bootloader in the root directory.
     1. mov [bpbRootEntries] into cx.
        1. It's no coincidence that we using the counter register cx. [bpbRootEntries] is the number of entries to loop through in the linear search for the stage 2 bootloader's name.
     2. mov 0x0200 into di.
        1. Recall 0x0200 is where we loaded the root directory table.
     3. Loop for the linear search:
        1. push cx. We are about to call the assembly version of strncmp(), and we want cx to hold the "n".
        2. mov the name of the stage 2 bootloader into si.
           1. Recall that variable names in assembly are pointers. Move the pointer to the name. Do not dereference.
        3. Push di
        4. rep cmpsb
           1. cmpsb compares bytes aka characters.
           2. rep runs cmpsb repeatedly, effectively doing strncmp(). It uses cx as the "n".
        5. Pop di
        6. If we have a match, proceed to the next step, which is loading the FAT.
           1. je LOAD\_FAT
        7. Otherwise:
           1. Pop cx. Now cx holds the remaining number of root entries to search.
           2. Add 0x0020 to di.

0x0020 is 32 in decimal. Recall that root directory table entries are 32 bytes each. We are in effect moving to the start of the next entry.

* + - * 1. "loop" back to the loop

Once again, the loop instruction uses cx as the loop counter.

* + - * 1. jmp to failure-handling code if there are no more tries
    1. Failure handling code (we didn't find the stage 2 bootloader anywhere in the root directory table):
       1. mov the failure message to si
       2. Call print string
       3. Reboot upon a keypress. First wait for a keypress
          1. Do this by calling int 0x16.

ah: Function type. Set to 0 for "read key press".

* + - 1. Call int 0x19 to reboot
         1. No registers
  1. Look up the stage 2 bootloader's starting cluster in the FAT. To do this we have to load the FAT first.
     1. First some cosmetics. Print a blank line.
        1. mov the string 0x0D 0x0A 0x00 into si and call print string.
           1. 0x0D is the ASCII for \r
           2. 0x0A is the ASCII for \n
           3. 0x00 is just the null terminator
     2. The goal is to call ReadSector on the FAT. Recall the registers we need:
        1. ax: Sector location.
        2. cx: Number of sectors to read
        3. es:bx: buffer location
     3. Before we get started, do some prepping for loading the stage 2 bootloader.
        1. In particular we want to calculate the starting cluster of the stage 2 bootloader.
        2. Recall that LOAD\_FAT is called by the code that searches for the stage 2 bootloader.
           1. When LOAD\_FAT is called, the base address of the root directory table entry for the stage 2 bootloader is stored in di.
        3. Recall that bytes 26-27 of each root directory table entry stores the first cluster.
           1. There aren't enough clusters to need two bytes. We only need the lower byte, which is byte 26 because x86 is small-endian.
        4. mov [di + 0x001A] into dx. 0x001A is the hex for 26.
           1. dx now stores the starting cluster number
        5. mov dx into [cluster].
           1. Later on, when we load the stage 2 bootloader, we will retrieve the starting cluster from [cluster].
     4. Now we can prep the registers for ReadSectors. Start with the number of sectors. Recall that ReadSectors expects this in cx.
        1. Why not fill out ax first?
           1. To compute the number of sectors, we need to multiply the number of FATs with the number of sectors per FAT. This requires the "mul" instruction.
           2. mul is hardcoded to multiply the value held in ax. There is no way to change this.
           3. If we filled out ax first, it would just get overwritten when we fill out cx.
        2. Clear out ax
        3. mov [bpbNumberOfFATs] into al
        4. mul WORD [bpbSectorsPerFAT]
           1. Notice that ax isn't explicitly specified here. It doesn't need to be, because it's the only possible register to use.
           2. Strictly speaking, when we call mul on a WORD, ax is taken, and the multiplication result is stored in dx:ax.

The higher-order bits are in dx, the lower-order bits are in ax.

* + - * 1. The multiplication result is small enough that dx will be 0. So for our purposes, the result is just stored in ax.
      1. mov ax into cx.
    1. *Now* we can fill out ax, which holds the starting sector.
       1. mov [bpbReservedSectors] into ax and we are done.
          1. Recall that the FATs come immediately after reserved sectors, and that sectors are 0-indexed.
          2. Therefore [bpbReservedSectors] is also the starting sector of the FATs
    2. Lastly, fill out bx, the base address of the buffer
       1. Let's reuse 0x0200: mov bx, 0x0200
    3. Now call ReadSectors
  1. The starting cluster from the FAT is its index in the table, but we want the physical (aka Cylinder/Head/Sector or CHS) address. Convert to CHS in two steps:
     1. Convert cluster index to logical block address (LBA)
        1. Not all cluster values in the FAT represent cluster indices. Relevant to our current purpose, cluster values 0x00 and 0x01 do not represent cluster indices:
           1. 0x00: free cluster
           2. 0x01: reserved cluster
        2. Therefore cluster indices start from 0x02 (and end at 0xFEF. Values higher than 0xFEF also don't represent cluster indices):
           1. 0x02 marks the 0th index
           2. 0x03 marks the 1st index
           3. Etc.
        3. To convert a sector's location from FAT sector value to LBA, first subtract 2 to get the cluster's index, then multiply by the number of sectors per cluster (1 for our OS).
  2. Actually load all sectors of the stage 2 bootloader, following the FAT's linked list.
     1. General strategy: call ReadSectors on each sector of the stage 2 bootloader.
     2. Recall the registers we need:
        1. ax: Sector location.
        2. cx: Number of sectors to read
        3. es:bx: buffer location
     3. The values of ax and cx change with each sector read, but es:bx does not. Set es and bx before we start the main loop.
        1. First print an empty line for cosmetics.
        2. Let's use 0050:0000 as the buffer location
        3. mov 0x0050 into es. Remember to use an intermediate register.
        4. Clear out bx
        5. push bx
           1. The reason is the main loop.
           2. In the main loop we are calling ReadSectors and looking up the next cluster. Both tasks use bx.
           3. Therefore, in each iteration, bx gets pushed onto the stack after calling ReadSectors, and popped back after looking up the next cluster. This is how we "remember" the 0x0000 value for bx.
           4. The popping happens at the beginning of the next iteration.
           5. Edge case: there's a pop at the start of the 0th iteration. We need to ensure that the popped value isn't garbage.
           6. This is why we push the valid bx value of 0x0000 before the main loop, so 0x0000 gets popped back into bx in the 0th iteration.
     4. Main loop:
        1. Recall we want to fill out ax and cx.
        2. ax: recall that [cluster] stores the current cluster index.
           1. Back when we were loading the FAT, we moved the first cluster index of the stage 2 bootloader into [cluster].
           2. We will update [cluster] with every sector read, by looking up the FAT.
        3. ax continued: ax needs the LBA location, but [cluster] stores the cluster index. First implement the translation function ClusterLBA:
           1. Recall that cluster indices 0x00 and 0x01 have special meanings, and do not represent actual clusters. 0x02 represents cluster 0.
           2. So first subtract ax (which holds the cluster index) by 2.
           3. Now we have the cluster number, but we want the sector number. We need to multiply by the number of sectors per cluster.
           4. Clear out cx
           5. mov [bpbSectorsPerCluster] into cl
           6. mul cx.

Recall that mul implicitly uses ax. Therefore ax now holds the starting *sector* number.

* + - * 1. Right now ax holds the sector number relative to the start of the data sector, but we need the sector number relative to the start of the *disk*.

Add [datasector] to ax.

* + - * 1. Return
      1. Now call ClusterLBA, which turns the value in ax to the starting LBA sector number relative to the start of the disk, exactly what ReadSectors needs.
      2. Now fill out cx, which holds the number of sectors to read.
         1. Clear out cx.
         2. mov [bpbSectorsPerCluster] into cl.
      3. Call ReadSectors
      4. Push bx onto the stack
         1. As mentioned, we are about to look up the next sector, and this task uses bx as well.
         2. So save the current bx on the stack, and pop it back once we are done with the lookup.
      5. Look up the next cluster
         1. First get the base address of the current cluster's address.

Add this base address to the base address of the loaded FAT (recall we loaded the FAT at 0x0200), to obtain the absolute base address of the cluster's FAT entry.

* + - * 1. The current cluster's index is stored in [cluster], but we need the number of *bytes*.
        2. Recall that each FAT entry is 12 bits, or 1.5 bytes.

Multiply the cluster index by 1.5 to get the number of bytes from the FAT's base address.

* + - * 1. mov WORD [cluster] into ax. Then mov ax into cx and dx.

We'll use ax to test whether the cluster index is even or odd (see below).

We'll divide dx by 2 and add back to cx. This gives up the 1.5x ax that we want.

* + - * 1. shr dx by 0x0001

Shift right by 1 bit, aka divide by 2

* + - * 1. add dx to cx. Now cx holds the number of bytes from the FAT's base address.
        2. To obtain the absolute address, we need to add the base address of the FAT itself. We loaded the FAT at 0x0200.

Add both cx and 0x0200 to bx. Now bx holds the absolute address of the FAT entry of the current cluster.

* + - * 1. Recall that each FAT entry is 12 bits long. We must read in a word rather than a byte, because a byte isn't big enough.

mov WORD [bx] into dx.

bx is the absolute address. [bx] dereferences the pointer, and gives us the two bytes that include the lookup result (aka next cluster's index).

* + - * 1. But a word is *too* big. We only need either the first or the last 12 bits, depending on the cluster.
        2. The treatment is different for even and odd clusters.

test ax 0x0001

The "test" instruction does a bitwise AND on ax and 0x0001, and sets the zero flag based on the result.

If ax is odd then jump to the odd-cluster handling code.

jnz .ODD\_CLUSTER

* + - * 1. Even clusters:

Even clusters use all of the first byte, and the lower half of the second. We want to get rid of the upper half of the second byte

Because of small-endianness, the upper half of the second byte is just the highest four bits of the word.

Therefore apply a mask 0x0FFF

and dx, 0x0FFF

* + - * 1. Odd clusters:

Odd clusters use the higher half of the first byte, and all of the second byte.

Just shift the word four bits to the right.

shr dx, 0x0004

* + - * 1. Now we have the 12-bit index for the next cluster in dx. mov dx into [cluster]
        2. Not all dx values signify a valid next cluster:

We already saw that 0x00 and 0x01 do not.

Values greater than or equal to 0x0FF0 do not either. Exit the loop if dx is one of these values.

* + - * 1. Test whether we should exit the loop:

cmp dx, 0x0FF0

jb <start of loop>

Think of jb as the unsigned version of jl.

* 1. The stage 2 bootloader is loaded
     1. Print a blank line
     2. Return to the stage 2 bootloader
        1. This is not a ret but a retf, which is a far return.
        2. A far return returns to an address on a different segment. Recall that we loaded the stage 2 bootloader at 0050:0000.
        3. How to let the CPU know this return address? The CPU looks at the top two entries on the stack as the segment and offset.
        4. push WORD 0x0050
        5. push WORD 0x0000
        6. Now that the return address is on the stack:
           1. retf