Phone Khont Lu

Task 6.

a) $x = 65534$ in binary is $2^{16} - 2$

So 65534 → 1111 1111 1111 1110

10

~~signed bit~~ 0x F F F E

$x = \sim x$ will negate all the bits in x.

So, $x = \sim x$ results in

~~signed bit~~ → 0000 0000 0000 0001 and python flips the sign.

This is equal to 2's complement of the number 65535 ~~in~~ so for

10

print (x), python displays "-65535"

for print (formab (x, "04X"))

python distinguishes the sign bit, and reads 4 Msb so., it will print

including the sign bit!

"-1000"

b

~~I ask b~~

~~b)~~

b) for print (x), ~~it will print~~ it will print one as the bits are interpreted without the sign.

for print (format (x, "04X")),

it will print "0000" as there is no sign involved and it is an unsigned int.

Phone Khant Lu

Task 10

First convert the real numbers to fixed point format in 8.24

$$65 = 1 \times 2^6 + 1 \times 2^0$$

$$= 01000001_2 \quad (\text{length } 8)$$

$$0.75 = 1 \times 2^{-1} + 1 \times 2^{-2}$$

$$= .11\underbrace{000000\ldots00}_{22 \text{ zeroes}}{}_2$$

So 65.75 = 41 C0 00 00 in hexadecimal format in 8.24

In 12.20 format, similarly

$$65 = 0000\ 0100\ 0001_2$$

$$0.75 = 11\underbrace{000\ldots00}_{18 \text{ zeroes}}$$

So, 65.75 = 04 1C 00 00 in hex for 12.20

Task 10 (continued)

Hexdump (since it is in little-endian format)

00 AC 1400 : 00

00 AC 1401 : 00

00 AC 1402 : 00

00 AC 1403 : 00

00 AC 1404 : 00 ┐
00 AC 1405 : 00 │ first value
00 AC 1406 : C0 │
00 AC 1407 : 41 ┘

00 AC 1408 : 00

00 AC 1409 : 00

00 AC 140A : 00 ┐
00 AC 140B : 00 │ second value
00 AC 140C : 1C │
00 AC 140D : 04 ┘

00 AC 140E : 00

00 AC 140F : 00

Phone Khant Lu

Task 5,

We will solve the inner parentheses first, "or" it with b and then shift the resulting value.

	LsB	MsB
a:	0xE000	0xE001
b:	E002	E003
c:	E004	E005
temp1:	0000	0001

Assembly code

```
> LDA  $E004  $0x0E004
  AND  #$0x07
  STA  $0x0000
  LDA  $0x 0005
  AND  #$0x3F
  STA  $0x0001
  LDA  $0x0000
  ORA  $0xE002
  STA  $0x0000
  LDA  $0x0001
  ORA  $0xE003
  STA  $0x0001
```

(c&0x3F07)

(result|b)

Task 5

```
>LDA  $0x0000
 CLC
 ROL A
 STA  $0xE000
 LDA  $0x0001
 ROL A
 STA  $0xE001
```

a = result << 1

Phone Khant Lu

Task 8

"MFF"

∴ "A" = ~~0x4h~~ 0x41

"F" = ord("A") + 5

= 0x41 + 5

= 0x46

"M" = ord("A") + 12

= 0x41 + 12

= 0x4D

We will send MFF so

"M" + "F" + "F"

0100 1101 +0100 0110 +0100 0110

Each data "unit" is preceded by a start condition and followed by a stop condition

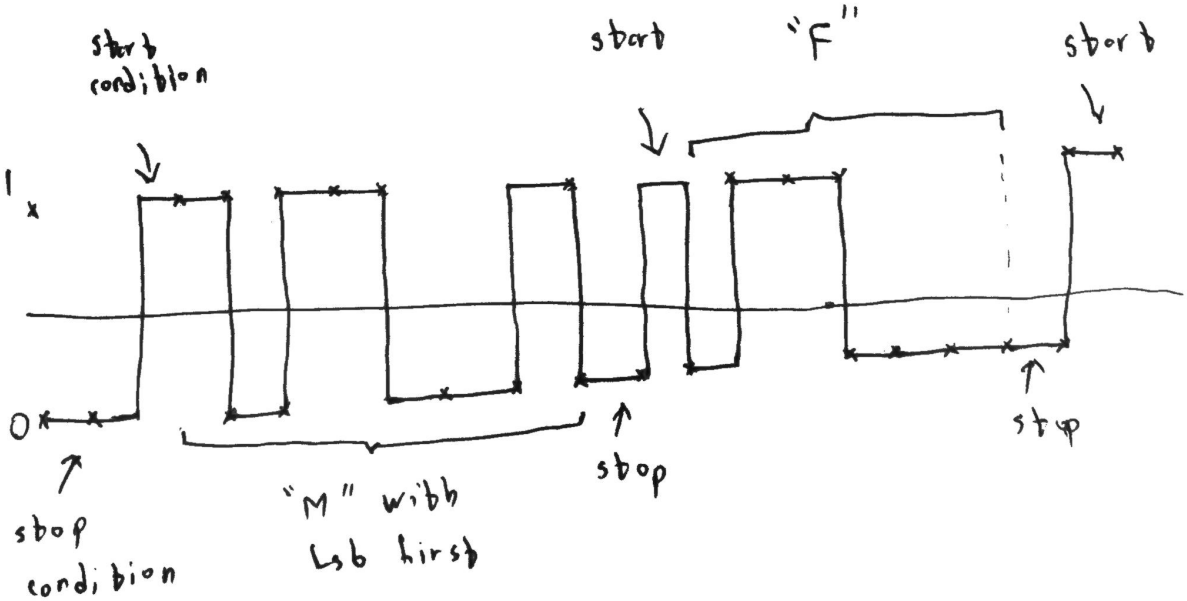
~~So, th~~

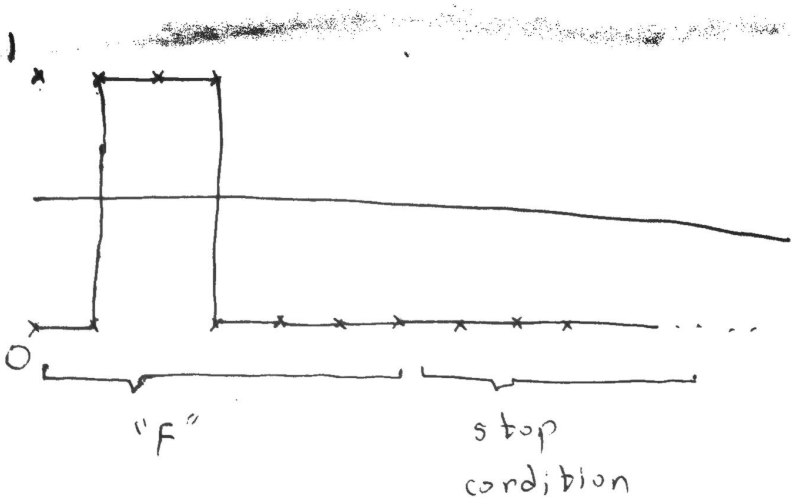
~~I will divide~~ tl

Since the RS-232 line can send only 7-bits, the Msb is truncated.

Task 8

Assumming the start condition is 1 and stop condition is zero,





Phone Khant Lu

Task 9

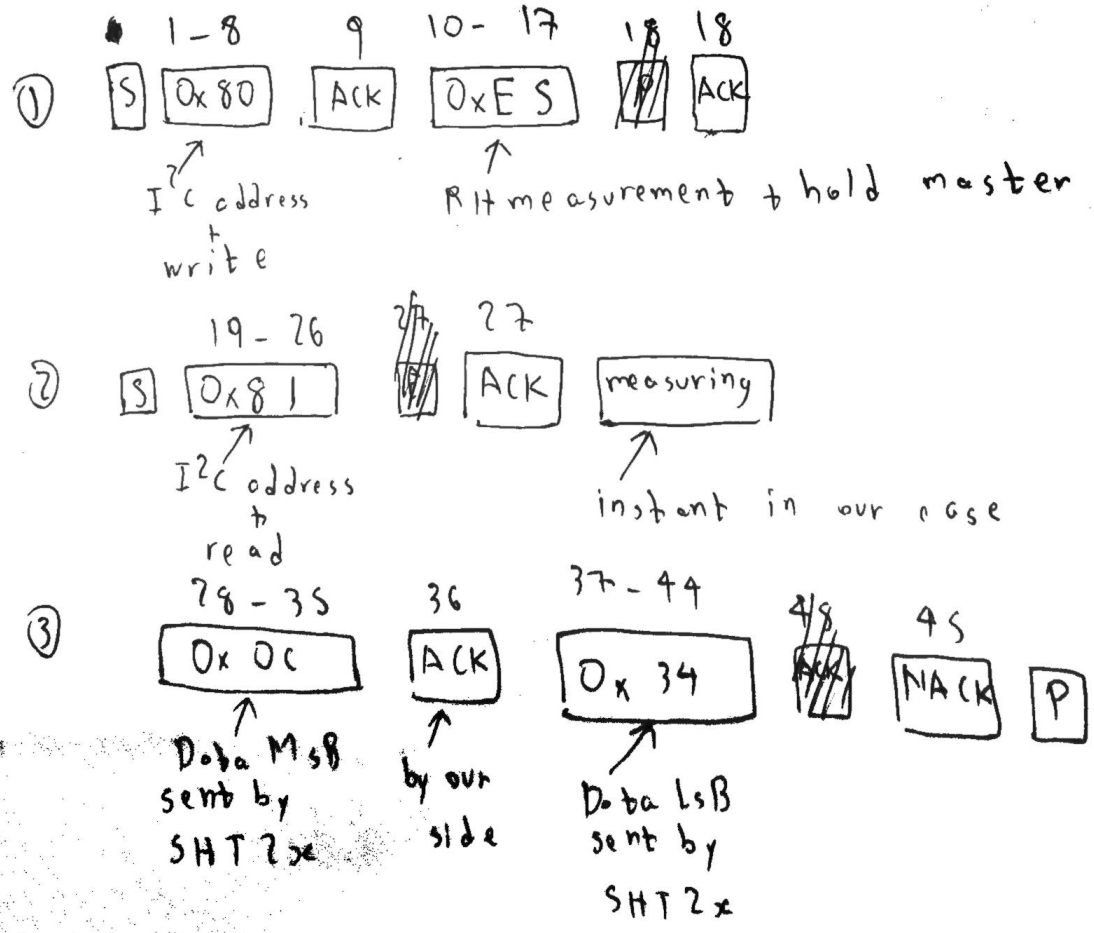
Since the measurement is instant, it does not matter whether we use "hold master" or "no hold master"

I will use "hold master" since I think it is simpler.

Let S be transmission start condition,

Let P be transmission stop condition.

In hexadecimal, $3124 = 12 \times 16^2 + 3 \times 16 + 4 = 0x0C34$



Note: we did not receive a
check sum so it was left out.

The hex transmission above can be summarized into

① Send I^2C address + write and then specify the humidity command,

② Send I^2C read + address

③ Normally, our device will be forced into wait state but in our case, the measurement is instant.

④ SHT 2x send us MSB and LSB of data

Phone Khant Lw

Task 7

Subtraction with borrow and subtract with carry are used when we want to perform subtraction on data whose size is bigger than the word size of the CPU.

In subtract with borrow,

there is a borrow flag in the flag register of the CPU.

when we perform subtraction between 2 words (lower bytes) of data and we need to borrow from the higher bytes, the borrow flag is set to 1.

When we perform subtraction with borrow on higher bytes of our data, the CPU subtracts the higher byte plus one such that

Divide A, B into A_{LSB}, A_{MSB}, B_{LSB}, B_{MSB}

If $B_{LSB} - A_{LSB}$ requires borrow, borrow flag is set.

In $B_{MSB} - A_{MSB}$, If borrow flag is set,

result = $B_{MSB} - A_{MSB} - 1$

Task 7

Subtract with carry works similarly to subtract with borrow except that it uses the carry flag with inverted logic.

~~carry~~
borrow flag = ~ carry flag.

The advantage is that we can reuse the same flag for both subtraction and addition.

However, for both implementations, we must set the relevant flag to its default state before starting any new subtraction or the results will be wrong. (SEC for subtract with carry, ~~etc for sub~~ clear borrow flag for subtract with borrow.)

The second implementation is more common.

Phone Khant Lu

Task 2 (X)

If we look at the directory starting from 0009800, we can see second.txt at 0x9880.

According to the section 4 of the document, we can deduce the inode number of second.txt by looking at the 1st 2 bytes.

Namely, 04 00

So the inode number is 4

So, it must be the ~~2nd~~ third inode after the inode of the root directory namely at 0x1060.

By looking at byte 04 - 07 of this inode entry as indicated in the table in section 3, we can know the size.

size of second.txt = 0e 00 00 00

Interpreted in little-endian,

size = $16^0 \times 14$ = 14 bytes

Task 2(X)

Now, the contents, by looking at the "zone" bytes of the inode entry, we can see that there is only 1 zone, namely ~~zero~~ zone0 with contents 1c 00.

So, we can look at sector 1c to view ~~the~~ the contents.

This is address 0x7000, i.e ~~3rd~~ the 2nd sector after 0x6800. (0x6800 + 0x0800)

The contents are

49 27 6c 6c 20 62 65 20 62 61 63

6b 21 0a.

As conveniently shown in the hexdump, we can decode this to

"I'll be back!"

Note: We can verify the length of this file by adding the num of characters * (1 byte) + 1 byte ~~for~~ from the ~~EOL ch~~ EOF character.
So, in total, 14 bytes.

Phone Khant Lu

Task (1)

If we look at the documentation we can see that superblock is offset 1024 bytes from the start of the disk. So ~~the first b~~ we have to look at 00000400 in the hexdump.

We can clearly see that the first 2 bytes in the superblock indicate the number of nodes,

So, it is c0 02h.

Since it is in little-endian format,

no of inodes = $12 \times 16^0 + 0 \times 16^1 + 0 \times 16^2 + 1 \times 16^3$

= 12 + 4096

= 4108

no. of inodes = $0 \times 16^0 + c \times 16^1 + 7 \times 16^2 + 0 \times 16^3$

= 192 + 512

= 704

Phone Khant 44

Task 13 (X)

```
def PrintBlocks(name: str, inodeOffset : int):
    #I will write the assumed function of the
     function I use because I don't remember the
     names exactly
    #returns a file object
    f = open (name, "r")
    #reads offset + size of inode
    raw_read = f.read (offset + 32)
    #slices bytes so that we can inspect
     the inode
    inode = raw_read [offset : ]
    # I isolate zone bytes
    zones = inode [14:]
    for i in range (0, 2):
        isolated_bytes = zones [i : i+2]
    if zones[0] != 0x00 or zones [1] != 0x00:
        print ("0")
    if zones [2] != 0x00 or zones [3] != 0x00:
        print ("1")
    if zones [4] != 0x00 or zones [5] != 0x00:
        print ("2")
```

Task 3

```
if zones[6] != 0x00 or zones[7] != 0x00:
    print("3")
if zones[8] != 0x00 or zones[9] != 0x00:
    print("4")
if zones[10] != 0x00 or zones[11] != 0x00:
    print("5")
if zones[12] != 0x00 or zones[13] != 0x00:
    print("6")
if zones[14] != 0x00 or zones[15] != 0x00:
    LSB = int(zones[14])
    MSB = int(zones[15]) * 16**2
    location = LSB + MSB
    raw_read = f.read(location + 1024)
    zones = raw_read[location:],
    for i in range(1024):
        zone_base_number = 7
    for i
```

Phone Khant Lu

Task 3

```
for i in range (512):
    if zones [2*i] != 0x00 or zones [2*i+1]
                                != 0x00:
        ~~print ("str(zone_base_number)+~~
        ~~str(i~~
        print (str(zone_base_number +
                   i))

    return n
```

End of function

Phone Khant Lu

Task 9

```
def GetLongestFreeBlock(bitmap bytes) -> int:
    # I will treat bytes as an array of bytes as
      I am not familiar with this object

    Spotted = False
    Maxlength = 0
    Max index = None
    i = 0, offset = 0, current_length = None

    while ~~true~~ i < len(bitmap):
        ~~If flag 1 = True: spotted 1 = True~~
        ~~for j in range(7):~~
        byte = bitmap[i]
        If ~~flag 1 = Tru~~ spotted != True
            for j in range (0, 8):
                if (byte << j) == 0x00
                    Spotted = True
                    offset = j + (i * 8)
                    ~~byte~~ current length = 8 - * j
                    i = j + 1
                    continue

                i = i + 1
                continue
```

Task 4

Else :

~~for j in range(0,8):~~

~~if (byte >> j) == 0x00:~~

if byte == 0x00 :

current_length += 8

i = i+1

continue

~~for j in range (0,8):~~

~~if (byte >> j) == 0x00~~

elif byte == 0xFF :

if ~~ma~~ current_length > Maxlength:

Maxlength = current_length

~~off~~

Max index = offset

~~cor current_length~~ = ?

~~else:~~

i = i+1

Spotted = False

~~continue.~~

current_length = 0

continue.

Task 4

```
else:
    for j in range(0, 8):
        if (byte >> j) == 0x00:
            current_length += j
            if current_length > Max length:
                Max length = current_length
                Max index = offset

            i = i+1
            Spotted = False
            current_length = 0

            continue

return Max index
```