16bit floating point converter for LC-3

Published By: Kyle Miller

04/15/2019

Purpose: This program takes a decimal input and outputs a 16bit binary floating value according to IEEE standards.

Expected Input:

It is expected that the user will input a decimal number, positive or negative, but no larger than 65,504 with up to four decimal places. The program will error check for you

Expected Output:

The output is the IEEE format for 16bit floating point numbers. Here is an example of how to interpret that number:

Sign Bit	Exponent (binary - #15)	Hidden Bit (usually 1)	Fraction (Mantilla)
1	11001	1.	1100 0011 00
Negative	25-15 = 10	Shift decimal by ten	1110 0001 1100 = -1358

Unsupported output:

This program does not support +/- infinity or NaN. It also does not support decimals less than 0.0001 though there are some numbers in this range which are allowed by IEEE standards. This may be added in a future update.

Registers/ Labels Used:

This program uses all registers at different times during the program. Most if not all of this information can be found in the comment lines of the program, but here are the general breakdowns of each major section of the program.

1. Print Welcome and reset variables (8:28)
    a. From lines 8-28 the program is initialized by clearing R0 to zero and then storing zero the vital memory locations of this program which include WN_LENGTH, DEC_LENGTH, SIGN, DECIMAL, DEC_SPACE, WHOLE_NUMBER, DECIMAL_NUMBER, FLOATING_POINT, EXPONENT, MANTILLA

    b. Label FIRST_MESASAGE to identifies where the program welcomes you
    c. Label BEGINNING identifies where the program restarts to if user input error is encountered or the user chooses to do another calculation

2. Get User Input (29:136)

a. This section of the program can be a little complicated because I want to error check the user input real-time as much as possible. For example, a user can enter a negative number, but that is only allowed during the first input. Another example would be decimal points, you can have decimal, but only once. And what if the decimal is pressed first – then I print a zero prior to the decimal. User input stops when the user either presses enter, or 4 decimal places are reached.
b. R5 stores the pointer to memory where the decimal numbers will be stored
c. Label MAIN_GET_WHOLE_NUM is the beginning of this routine and really is only used for the first user input
d. Label CONTINUE_INPUT is the loop portion of the input routine
e. Label MAIN_GET_DECIMAL is the loop where the decimal values are acquired. Once 4 decimals have been acquired then the program continues execution.
f. Label CHECK_VALID_INPUT_FIRST is the loop that is run only once to check for a decimal or negative
g. Label SET_NEGATIVE sets the negative bit of the floating point number in memory and is only set if the user enters a negative input
h. Label SET_WHOLE_NUM_ZERO is used if the user presses the decimal point as their first input. It sets the whole number to zero, prints a zero and a decimal and then carriers on with decimal input
i. Label PRINT_DECIMAL is used if a whole number has already been entered and now the user presses the decimal key. It prints a decimal to screen, sets the decimal point in memory so that it can't be printed again, converts the whole number from memory continues memory locations into a single binary number by calling 'DEC_TO_BIN' and then transfers to MAIN_GET_DECIMAL
j. Label VALIDATE_DEC is very important and is used to validate user input real-time. If a user presses a disallowed key then nothing is printed to the screen
k. Label BREAK is used when the user presses enter. When this occurs the program checks to see if there is a decimal number or not or if it is just a whole number. If there is a whole number then it stores the decimal first before moving on
l. Label STORE_DECIMAL is a subsection of break for storing the decimal portion, skipped if no decimal portion
m. Label MAIN_RET checks to see if a whole number is present yet. If not then it means they pressed enter on the first key press so it circles back to the beginning of the MAIN_GET_WHOLE_NUM loop.

3. Store an integer to consecutive memory locations from user input (137:157)
    a. The purpose of the STORE_INT subroutine is to store a validated integer input from the user to consecutive memory locations starter from USER_NUM_PTR. This memory location is stored in R5. R6 contains the ASCII offset x0030. The user input is decremented by R6 and then stored to R5. The length of the user input (whole number or decimal number) is incremented and the memory location in R5 is incremented.
    b. Labels INT_LEN_PLUS and DEC_LEN_PLUS are responsible for incrementing the length of the value we're creating, whether that be the whole number or the decimal.

4. Convert consecutive integers in memory to a single binary number (163:208)
    a. The DEC_TO_BIN subroutine converts a list of decimal numbers stored in consecutive memory locations and recreates the number into 1 binary number which is stored and returned into R0
    b. The memory locations are found in R5
    c. Scratch work is done in R6
    d. The most challenging part of this process involves using the multiplication subroutine. Each time through the loop you have to change the multiplier to use on the stored decimal number (1,10,100,1000 etc.)
    e. Labels DB_LOOP and BREAK_DB_LOOP are used to check if we've fully converted the number stored in memory. It deletes values as it backs through the number and when it's done the loop breaks and we return
    f. Here is an example of how this section works with a user number of 9,502:

Memory Location	Stored Number	Multiplier	Process order	Building the binary number
X4000	9	1000	4	9502
X4001	5	100	3	502
X4002	0	10	2	02
X4003	2	1	1	2

    g. The subroutine looks at x4003 first and multiplies it by the multiplier 1, then saves it to R6. Then it looks at x4002, multiplies it by 10 and adds it to R6, then x4001 which is five, multiplies it by 100 and stores it to R6 and so forth until you have the entire number condensed from multiple memory locations to a single register – R0

5. Multiply two numbers(196:208)
    a. The MULTIPLY subroutine multiplies the values stored in R1 and R2 and returns the result in R0.
    b. It does this by using MULTILOOP to add R2 to itself R1 number of times.

6. Make a number negative (212:214)
    a. The MAKE_NEGATIVE subroutine does just that. It's a quick way to make a number negative if necessary. I think I only use this once or twice

7. Check to see if the number is too large(217:246)
    a. INVALID_CHECK is a subroutine that checks to see if the user entered a whole number that is too large. It first checks to see if the length of the whole number is too large. IF that passes then it checks the digits entered one by one until it discovers that the number must be OK (INVALID_FALSE), and continues on or must be too large and throws an error and re-prompts the user for input (INVALID_TRUE)

8. Variables and memory locations(248:281)
    a. I store most of the variables and memory locations here as well as my user prompts

9. Convert a whole number to a floating point mantilla(283:325)

   a. WHOLE_TO_FLT takes an input at R0 that is a binary representation of the whole number that needs to be prepared for the floating point format.
   b. At label WTF_LOOP we check to see if the number is fully shifted to the left. We need to find the first one in the number. Each time we shift to the left we count that and then use that to build our exponent value.
   c. One it's fully shifted to the left (first 1 in the number) it jumps to WN_SHIFTED. We now create the exponent to be used in the floating point output. Then it naturally flows into the FORM_FLOATING_POINT
   d. Label WN_ZERO is the condition where the whole number is just zero. In this case we create the decimal portion of the mantilla first which is a little trickier when there is no whole number

10. Form a complete floating point number(330:365)
    a. This is where we start to form the Floating Point number. We first push the exponent onto our blank number, then start pushing on the whole number portion of the mantilla. After creating that we then calculate the decimal portion of the mantilla and push it onto the number. When the number is finished being created and stored into FLOATING_POINT we print it off

11. Print the floating point number(369:422)
    a. Print the floating point number. We cycle through each bit in the FLOATING_POINT from left to right and print it. We identify where we're at in the printing and now incorporate the proper spaces and implied bit during the printout
    b. Label PRINT0 and PRINT1 are used for printing the bit at the location we're checking
    c. Label SPACE_CHECK and PRINT_SPACE keeps track of where we're at in the printing and adds the space if needed
    d. We use PRINT_SHIFT to shift the number we're printing to the left each cycle through the loop so we always check the left most bit with an AND. If the result is negative then a bit is present
    e. There is also a small subroutine here that is activated if both the whole number and decimal number are zero. It sets the hidden bit to zero and then runs the print function

12. Convert a decimal number to its floating point equivalent(427:504)
    a. This is one of the hardest parts of the program – creating the decimal portion of the mantilla. How this number is created depends on whether or not there is a whole number or not which is why there are two ways in to the meat of this function – DEC_TO_FLT and DEC_TO_FLT_NWN (no whole number).
    b. MAKE_FRACTION is where we convert the decimal to its correct binary representation in IEEE formatting.
    c. Making the fraction is a little more complicated, let me try and explain how it works: The decimal, say 0.5432 is first converted to a whole number –> 5,432. It is then compared to the number 10,000. If our number (5432) when doubled is greater than 10,000 then this means that the bit to be stored is a 1. So our first bit would be 1. We

then subtract 10,000 from the doubling (10,864). This new number (864) becomes our next number to check and we repeat that process to create our fraction.

d. The loop does that 16 times to create the largest and most precise fraction possible. If there is a whole number then we store this to DECIMAL_NUMBER and return to building the FLOATING_POINT

e. At this point the loop checks to see if there is a whole number. If there isn't then we will need to shift the fraction until there is a one in the first most bit location. We count these shifts and build the exponent at this time.

13. Build the floating point through bit shifting one register into another(510:532)
    a. This subroutine takes an input of R0, reads the left most bit and pushes that onto the right side of a source register R2. A counter is stored in R1 so we only shift the amount of bits that we want to. R2 is returned, and R2 could already contain a value when the subroutine begins.
    b. The purpose of this subroutine is to allow us to take a value like the exponent, push it onto our FLOATING_POINT, then take the whole number Mantilla and push it onto the FLOATING_POINT.
    c. This subroutine does a final rounding check at the very end of pushing the decimal onto the FLOATING_POINT

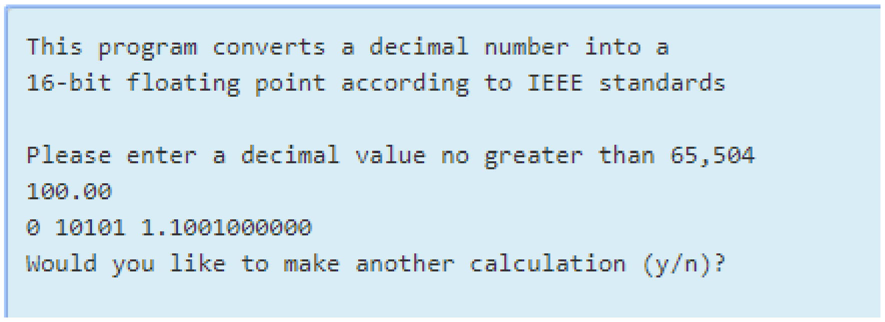
14. Restart the program(540:590)
    a. Prompt the user if they want to do another calculation. Only 'y' or 'n' are allowable input. If 'y' is selected then restart the program. If 'n' is selected then thank the user and halt the processor.

Output examples:

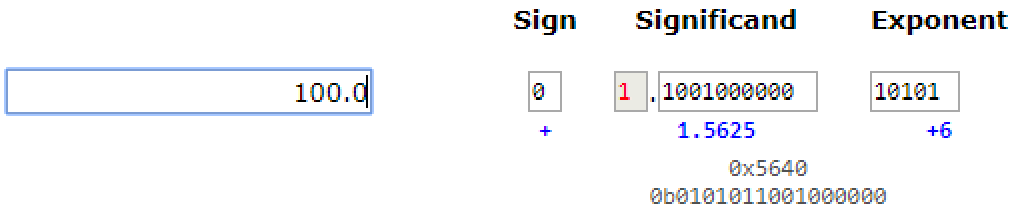
This program can handle input from -65504 to 65504 and decimals up to 4 points long. I cross checked my work with a published IEEE converter found here: http://weitz.de/ieee/, and I will show screen shots from both my LC-3 program and that converter.

INPUT: 100

OUTPUT: 0 10101 1.1001000000



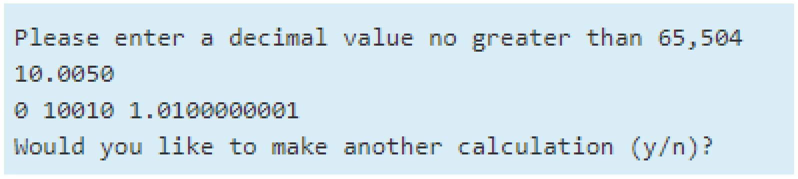
```
This program converts a decimal number into a
16-bit floating point according to IEEE standards

Please enter a decimal value no greater than 65,504
100.00
0 10101 1.1001000000
Would you like to make another calculation (y/n)?
```



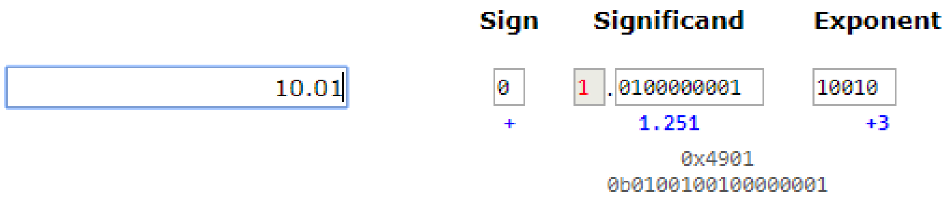
```
                Sign   Significand        Exponent
100.0           0      1 .1001000000      10101
                +        1.5625           +6
                          0x5640
                   0b0101011001000000
```

INPUT: 10.0050

OUTPUT: 0 10010 1.010000001



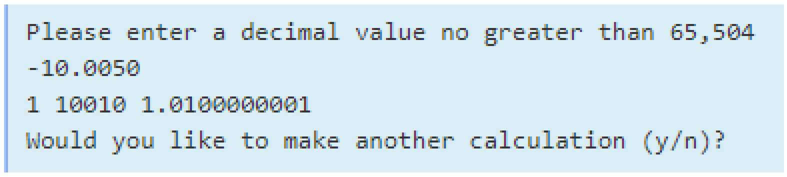
```
Please enter a decimal value no greater than 65,504
10.0050
0 10010 1.0100000001
Would you like to make another calculation (y/n)?
```



```
                Sign   Significand        Exponent
10.01           0      1 .0100000001      10010
                +        1.251            +3
                          0x4901
                   0b0100100100000001
```

INPUT: -10.0050

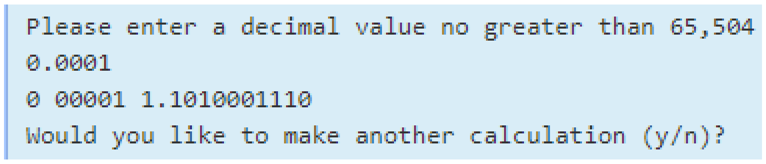
OUTPUT: 1 10010 1.010000001



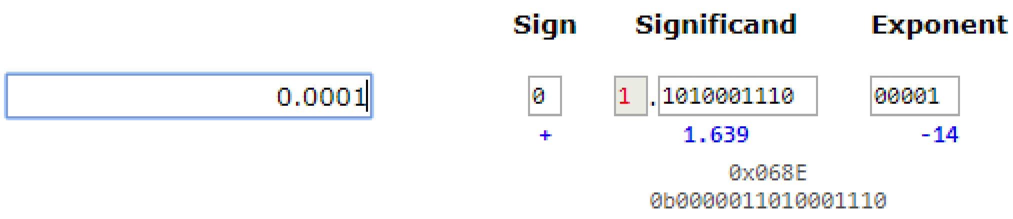
```
Please enter a decimal value no greater than 65,504
-10.0050
1 10010 1.0100000001
Would you like to make another calculation (y/n)?
```

INPUT: 0.0001

OUTPUT: 0 00001 1.1010001110



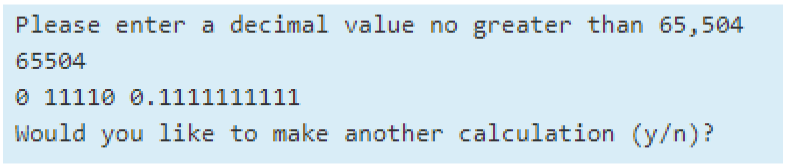
```
Please enter a decimal value no greater than 65,504
0.0001
0 00001 1.1010001110
Would you like to make another calculation (y/n)?
```



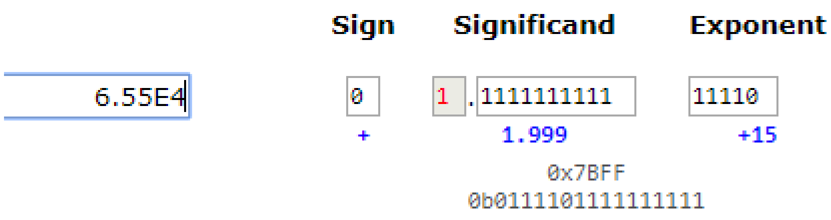
```
                         Sign    Significand        Exponent
0.0001                   0       1 .1010001110      00001
                         +       1.639              -14
                                    0x068E
                                 0b0000011010001110
```

INPUT: 65504

OUTPUT:



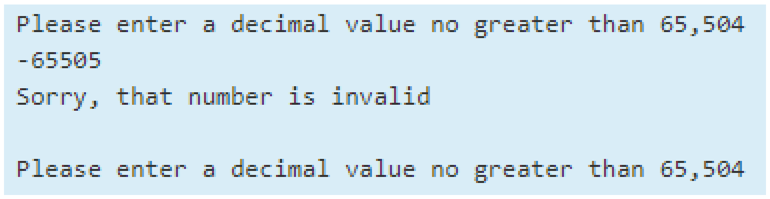
```
Please enter a decimal value no greater than 65,504
65504
0 11110 0.1111111111
Would you like to make another calculation (y/n)?
```



```
                   Sign    Significand        Exponent
6.55E4             0       1 .1111111111      11110
                   +       1.999              +15
                              0x7BFF
                           0b0111101111111111
```

INPUT:-65505

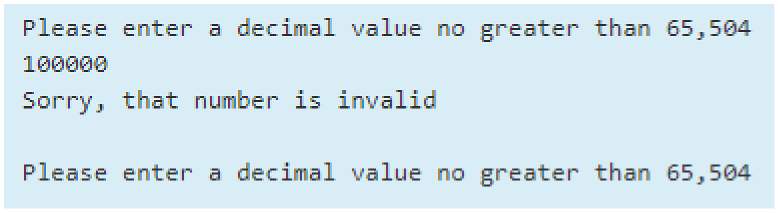
OUTPUT: N/A



```
Please enter a decimal value no greater than 65,504
-65505
Sorry, that number is invalid

Please enter a decimal value no greater than 65,504
```

INPUT: 100000

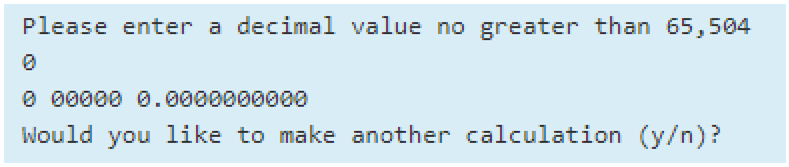
OUTPUT: N/A



```
Please enter a decimal value no greater than 65,504
100000
Sorry, that number is invalid

Please enter a decimal value no greater than 65,504
```

INPUT: 0

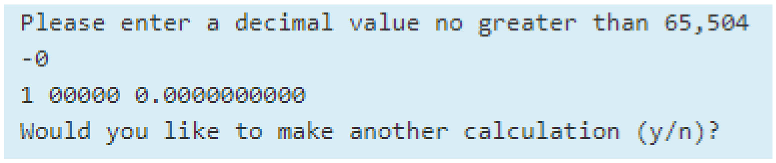
OUTPUT: 0 00000 0.0000000000



```
Please enter a decimal value no greater than 65,504
0
0 00000 0.0000000000
Would you like to make another calculation (y/n)?
```

INPUT: -0

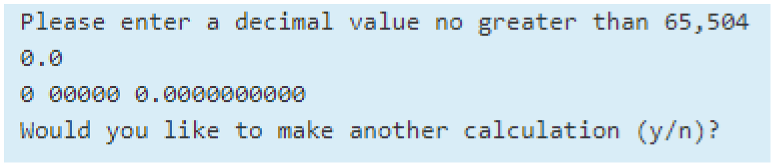
OUTPUT: 1 00000 0.0000000000



```
Please enter a decimal value no greater than 65,504
-0
1 00000 0.0000000000
Would you like to make another calculation (y/n)?
```

INPUT: 0.0

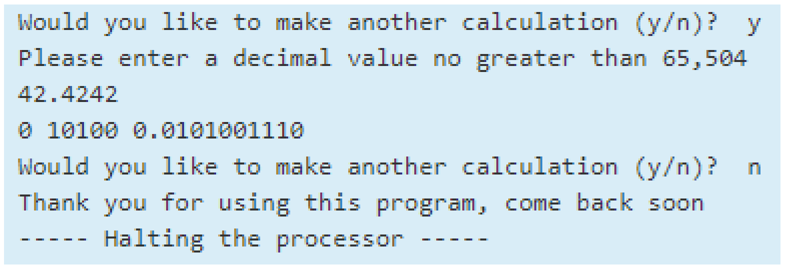
OUTPUT: 0 00000 0.0000000000



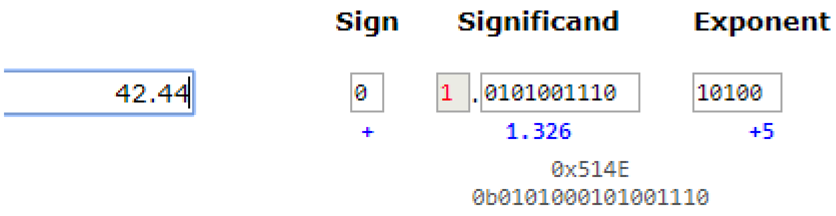
```
Please enter a decimal value no greater than 65,504
0.0
0 00000 0.0000000000
Would you like to make another calculation (y/n)?
```

INPUT: 42.4242

OUTPUT: 0 10100 1.0101001110



```
Would you like to make another calculation (y/n)?  y
Please enter a decimal value no greater than 65,504
42.4242
0 10100 0.0101001110
Would you like to make another calculation (y/n)?  n
Thank you for using this program, come back soon
----- Halting the processor -----
```



```
                     Sign    Significand      Exponent

42.44                 0      1 .0101001110     10100
                      +          1.326           +5
                                   0x514E
                             0b0101000101001110
```