Hui Shi Li

Section #6

TA: Daphne Gorman

Partner: Henry Pan

2/23/2015

Lab 4

Introduction:

The purpose of this lab is to understand how to convert two digit integers to half precision floating point, since LC-3 only has a word size of 16 bits. And to also multiply the resulting two floating point numbers together. We also take into consideration inputs of zero and negative integers.

Procedure:

- Integer Conversion:
  - The first step of this lab is to prompt the user to enter a sign and then the first and second digit of the integer. We started off by converting the sequence of characters to an integer. To do so, we stored each digit in a register and then multiplied the first input by 10. Taking the product of that and adding it to the second digit will result in our actual integer. We also stored the sign in another register and determined if a positive sign or negative sign was entered. We take the sign input and subtract 45 from it. Then we check if the result of the subtraction is zero or negative. If it is zero, then the sign is negative and we have 1 in the register holding the sign. If it is positive, then the sign is positive and we have 0 in the register holding the sign.
- Floating Point Conversion:
  - The second step is to convert the integer to half precision floating point. We first shift the integer to the correct position to be the mantissa/fraction part of the floating point. To do so, we shift the integer left until the first "1" aligns with the last bit of the exponent part by adding the integer to iteself. More precisely, you left shift until the "anding" of the integer and the mask of 1024 is positive. We also have a counter to keep track of the number of times that we have left shifted, which we will use to calculate the exponent. We take the exponent count and subtract that from 24 in order to get the exponent. We decided to subtract from 24 rather than 25 because we kept the leading "1" in the exponent component when calculating the mantissa. Now we shift the exponent left 10 times by adding the exponent to itself 10 times. Now that we have the exponent and mantissa components, we add them together. The final step is to add a "1" to the leftmost bit to signify a negative number if the integer was negative.
- Floating Point Multiplication:
  - The third step is to multiply the two floating points. First, we added the exponents together by loading two registers with the first exponent and the second exponent. We added them together and subtracted 15 from that sum in order to obtain the exponent of the product. Then we left shift it 10 times to get it back to its correct position. Next

we shift each mantissa to the right 4 times before we multiply them together. To right shift, we take the mantissa and divide by 2 four times. We take the mantissa and divide by 2, store the quotient from that division. Then make it the new dividend and divide by 2 again and continue to do this two additional times. Now we are ready to multiply the two mantissas. We basically added the first mantissa to the product as long as the second mantissa was still positive. Each time we add the first mantissa to the product, we decrement the second mantissa by 1. When the second mantissa becomes 0, we will have the product of multiply the two mantissas. The next step is to check how many times to right shift the product's mantissa back to the correct position. This depends on the result of "anding" the mask of x2000 with the product's mantissa. If the result is 0, then we right shift twice and if the result is not zero, then we right shift 3 times. We use the same right shift subroutine to right shift the product's mantissa, except we set the counter to the number of times we want to right shift, before calling the subroutine. After we right shift the mantissa to the correct position, we will remove the leading 1 by "anding" it with the mask of 1023. Finally, we determine what the sign of the product is. To do so, we add the two signs of the inputs together and add -1 to that. If the result of it is zero, the sign of the product is negative, in which we will add x8000, the exponent and mantissa component together. Otherwise, the sign of the product is positive and we only add the exponent and mantissa component together.

- Check for input of zero:
    - To check for the input of 0, we check if the sum of both digits are zero for each integer input. If the first integer input is 0(the sum is 0), then we remember that the first input is 0(storing it in a variable) and we skip the integer and floating point conversion. We continue to prompt the user for the second integer input. If it is zero, we skip to the end of our program (HALT). If the second integer input is not zero, we continue to do the integer and floating point conversion for the second integer. After the second input conversions, we recheck the variable that stored if the first input was zero or not. If it was zero, we skip the multiplication of the floating point numbers. If it was not zero, we continue with our multiplication.

Results: The integer conversion part of the lab was fairly easy to implement for us. As for the floating point conversion, the only problem that we encountered was that we left in the leading one of the mantissa, so we had to subtract from 24 to obtain the exponent.

Another problem we struggled with was checking for the zero input, at first, we implemented it to skip the conversions and multiplication, and halt the program. Then we realized that we forgotten about prompting for the second input, so we had to go back and implement the case where the first input was zero, but the second input was not zero and to do the conversions for the second input.

The most difficult part of the lab was multiplying the two floating points together. Originally, we wrote the entire part of the multiplication without checking whether or not each subroutine worked. So, when it came to testing our program, we got results that we did not expect and we did not know what went wrong where. This become a huge mess and we had to rewrite it, debugging each subroutine before moving onto another. This allowed us to easily debug and correctly implement the multiplication part of this lab.

Another problem that we had was that our prompts and some variables were too far from where we were using them, so we had to move them closer to where we used them.

Discussion:

The largest number that can be represented in half-precision floating point format is 65,504, which is 0 11110 1111111111.

The smallest positive number that can be represented in half-precision floating point format is 6.10 x 10^-5 which is approximately 2^-14.

JSR jumps to a location, similar to the unconditional branch, BRnzp and saves the current PC in r7. The RET, JMP r7 will get us back to where we called the routine.

;**Args**: R1(sign), R2(first input), R3(second input), R4(integer)

;converts the two digit input to an integer by multiplying r2(first input) by 10 and adding that to r4(integer), then adding that to r3(second input) to r4(integer)

;sets r1 to 1 or 0, if the sign is + or -

TOINT

;**return** r4(integer)

;**Args**: R1(integer)

;left shift r1(integer) by anding with r5(mask), branch INCREMENT until 0, increment r3(exponent counter) every time

;clear r5 and store mantissa to r5, set r4 to be 24

;invert r3(exponent counter) and add 1, clear r2 and store exponent component to r4

;left shift r4(exponent component) 10 times and add r1 to r4, store in r1

;and r6(sign) with itself to check if sign is negative, if so load r0 with signbit and add that to r1(float)

TOFLOAT

;**return** r1(float)

;**Args**: r1(mantissa), r2(divisor)

;invert r2 and add 1, add -1 to r5(counter)

;branch to DIVIDE2 if r5 is zero or positive

;increment r3(quotient), add r2 to r4(mantissa), branch DIVIDE2 as long as r4 is positive

;add r3 to r4 to be new dividend, and r3 with 0 to clear quotient

RIGHTSHIFT

;**return** r4(mantissa)

;**Args**: R1(mantissa1)

;add r1 to r3(product), add -1 to r2(mantissa2), branch MMLOOP as long as r2 is positive

MMLOOP

;**return** r3(mantissap)

;**Args**: R3(mantissa)

;and r3 and r4(mask2), if the result is 0, branch to SHIFT2, otherwise branch to SHIFT3

CHECKPRODUCT

;**return** r4(shifted mantissa)

;**Args**: r4(exponent)

;add r4 to itself 10 times

SHIFTEXPONENT

;**return** r4

;**Args**: r4(mantissashifted)

;and r1(mmask) and r4

REMOVE

;**return** r4

;**Args**: r1(first input's sign), r2(second input's sign),

;add r1 and r2 to r3(sum of signs), branch to PRODUCTSN if zero, otherwise branch to PRODUCTSP

PRODSIGN

;**return** r1(product)

Conclusion: In this lab, I learned how to convert to half precision floating point and how to multiply two half precision floating point numbers together. I also learned the importance of debugging each subroutine before implementing everything and debugging altogether. The hassle of implementing everything then debugging makes it hard to figure out where in the program, you implemented something incorrectly.

Henry Pan
Lab 4
Section 6
TA Daphne Gomez

Receive input 1 → check for ascii (+ / -)

Anything else ↙ if (+) ↘ if (-)

Sign set to (0) | Sign set to (1)

Receive input 2 | Receive input 2

Convert ASCII to Decimal | Convert ASCII to decimal

Multiply by 10 | Multiply by 10

Receive input 3 | Receive input 3

Convert ASCII to Decimal | Convert ASCII to decimal

Add the two numbers | Add the two numbers

→ if result is zero ←

↙ zero | ↙ nonzero

result = 0 | concatenate sum to sign

→ result

MZ 2/10/14

input → check for zero ↙ nonzero ↘ result = 0

MZ 2/10/14

Convert ASCII to Decimal | Convert ASCII to decimal

Add the two numbers | Add the two numbers

→ if result is zero ←

↙ zero | ↙ nonzero

result = 0 | concatenate sum to sign

→ result

input → check for zero ↙ nonzero ↘ result = 0

AND 1024 and input

↓ 0

add input to input

↓

increment exp

↓ 1024

Subtract exp from 24 for exponent component

↓

Add exp. comp. to itself 10 times

↓

Add exp. comp. to input (shifted)

↓

check for sign

↙ if (+) | ↘ if (-)

result | add 32768 (this is the sign bit)

↓

result

2/10/14