Jordan Wu

U0900517

HOMEWORK ASSIGNMENT #2

1. **Do problem 2.1 from the book.**

2.1

For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, and i are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

```
f = g + (h – 5);
```

For this problem we would need to know how to add and subtract in MIPS. The following MIPS assembly language notation for add and subtract is in the table below (from FIGURE.21 page 64).

Instruction	Example	Meaning
add	add $s1, $s2, $s3	$s1 = $s2 + $s3
subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3

Looking at this notation the MIPS uses registers that is represented by dollar sign $ and two-characters. For this particular problem, this programs uses four variables (f, g, h, and i) which can be assign to specific registers. The textbook stated that arithmetic operation occur only on registers in MIPS instructions. We can assign these four variable into four registers, $s0 - $s3. We would need to store this value (h – 5) into a temporary register, $t0 for temporary register.

MIPS assembly code for f = g + (h – 5):

```
f = $s0, g = $s1, h = $s2, and i = $s3
$t0 = temporary register
```

```
addi  $t0, $s2, -5     #$t0 = $s2 - 5
add   $s0, $s1, $t0    #$s0 = $s1 + $t0
```

or

```
using variable f, g, h, i, and t
addi  t, h, -5         t = h - 5
add   f, g, t          f = g + t
```

2. **Repeat problem 2.1 on this C/Java code:**

```
g = (i - g) + (f - g);

h = f + i;
```

Looking at this problem there is four variables g, i, f, and h. We can assign each of these variables into four registers, $s0 - $s3. We will also use the table from the previous problem to add and subtract these variables. Note: that we will need to use two temporary registers to store the values (i - g) and (f – g), $t0 and $t1. The corresponding MIPS assembly code will be given below.

MIPS assembly code for g = (i – g) + (f – g) and h = f + i:

```
g = $s0, i = $s1, f = $s2, h = $s3
temporary register = $t0 and $t1
sub   $t0, $s1, $s0    #$t0 = $s1 - $s0
sub   $t1, $s2, $s0    #$t0 = $s2 - $s0
add   $s0, $t0, $t1    #$s0 = $t0 + $t1
add   $s3, $s2, $s1    #$s3 = $s2 + $s1
```

or

```
using variable g, i, f, and h along with t0 and t1.
sub   t0, i, g         t0 = i - g
sub   t1, f, g         t1 = f - g
add   g, t0, t1        g = t0 + t1
```

```
add   h, f, i          h = f + i
```

3. **Do problem 2.2 from the book.**

2.2

For the following MIPS assembly instruction below, what is a corresponding C statement?

```
add   f, g, h
add   f, i, f
```

The following MIPS assembly instruction only uses add arithmetic operation which there are only three register operands used. The following MIPS assembly instruction corresponds to the C statement below.

C Statement:

```
f = g + h;
f = i + f;
```

4. **Your answer from problem #1 is not yet MIPS assembly. Convert your answer from problem #1, and convert variable names to actual MIPS register names. Use registers $s0, $s1, $s2, $s3, and $t6 for the variables f, g, h, i, and t respectively.**

My solution to problem #1 was already in MIPS assembly code. The only difference now is that I will need to use the temporary register $t6 instead of $t0. The following MIPS assembly code will be given below with the minor changes.

MIPS assembly code for f = g + (h – 5):

```
f, g, h, and i are assigned to the registers $s0, $s1, $s2, and
$s3 (respectively)
temporary register is $t6
addi  $t6, $s2, -5     #$t6 = $s2 - 5
add   $s0, $s1, $t6    #$s0 = $s1 + $t6
```

5. **Do problem 2.3 from the book.**

2.3

For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
B[8] = A[i-j];
```

Clarification: In arrays, elements in an array (of integers) are four bytes apart, so their memory addresses are four apart. Usually, the address of the first element is placed in a register. Addresses of later elements are computed (with additional instructions) or accessed by offset. Also, assume the first array position is position 0, for example: A[0].

For this problem we would need to understand the following terms, since this involves memory operands like an array (data structures are kept in memory which we need the memory address to access the correct data).

Data transfer instructions:

Load word (lw): word from memory to a register

Store word (sw): word from a register to memory

Instruction	Example	Meaning
Load word	lw $s1,20($s2)	$s1 = Memory[$s2+20]
Save word	sw $s1,20($s2)	Memory[$s2+20] = $s1

Note: In MIPS, words must start at addresses that are multiples of 4 this is called alignment restriction. Byte addressing also affects the array index. To get the proper byte address, the offset will be the array index multiplied by four.

For this problem we have to store a temporary value which is (i–j), so we would need to use $t0. Then we have to find the offset to get the correct address for the array A[i–j] because of the alignment restriction. After we load the value in A[i-j] which we would need to save into another temporary register $t1. Lastly we would need to store this reference value A[i-j] into B[8] (has the offset of 8*4).

Multiplying index by 4 to get byte address using:

```
sll  $t1, $t0, 2     #$t1 = $t0 * 4
```

MIPS assembly code for B[8] = A[i-j]:

Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively.

Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
sub   $t0, $s3, $s4    #$t0 = $s3 - $s4 or i - j
sll   $t0, $t0, 2      #$t0 = $t0 * 4 (shift to the left by 2)
lw    $t1, $t0($s6)    #Temporary reg $t1 gets A[i-j]
sw    $t1, (32)($s7)   #Stores A[i-j] into B[8]
```

6. **Repeat problem 2.3 on this C/Java code:**

```
f = B[f];

g = A[f] + f;
```

You may use $t0 as a temporary register.

First thing would be to find the offset for the array B[f] so that we will get the proper byte address. Once we done that we should be able to store B[f] into f. After we have to find the new offset since the value of f is changed. Then load word to find the value of A[f] (new offset since the value of f is changed) and after we should be able to use the addition operator to store the sum into g. The following MIPS code is given below.

MIPS assembly code for f = B[f] then g = A[f] + f:

Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively.

Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
sll   $t0, $s0, 2      #$t0 = $s0 * 4
lw    $t0, $t0($s7)    #Temporary reg $t0 gets B[f]
addi  $s0, $t0, 0      #$s0 = $t0 + 0
sll   $t0, $s0, 2      #$t0 = $s0 * 4
lw    $t0, $t0($s6)    #Temporary reg $t0 gets A[f]
```

```
add   $s1, $t0, $s0     #$s1 = $t0 + $s0
```

7. **Show how to convert the following two numbers to decimal three times: First, assume the numbers are unsigned binary and convert to decimal. Second, assume the numbers are twos-complement binary and convert to decimal. Finally, assume the numbers are 32-bit floating point numbers and convert to decimal.**

a.	**0x4396E000**
b.	**0xBFCCCCCD**

You should round the second number (b) to a reasonable result. Also remember that the 0x is not part of the number -- it just indicates 'hexadecimal'.) Please choose an appropriate level of detail when showing your conversions - we should be able to see how the conversion was accomplished, but we don't want to become lost in miniscule math operations.

In the textbook it states that C and Java use the notation 0xnnnnn for hexadecimal numbers. Assuming that these two numbers are in hexadecimal numbers, the first step would to convert them into binary numbers using the hexadecimal-binary conversion table on page 81 in the textbook.

Unsigned binary:

Hex for A	4	3	9	6	E	0	0	0
Binary for A	0100	0011	1001	0110	1110	0000	0000	0000

$$Decimal\ a = 1 \cdot 2^{13} + 1 \cdot 2^{14} + 1 \cdot 2^{15} + 1 \cdot 2^{17} + 1 \cdot 2^{18} + 1 \cdot 2^{20} + 1 \cdot 2^{23} + 1 \cdot 2^{24} + 1 \cdot 2^{25} + 1 \cdot 2^{30}$$
$$= 1133961216$$

Unsigned binary for a: 01000011100101101110000000000000 = 1133961216

Hex for B	B	F	C	C	C	C	C	D
Binary for B	1011	1111	1100	1100	1100	1100	1100	1101

$$Decimal\ b = 1\cdot 2^0 + 1\cdot 2^2 + 1\cdot 2^3 + 1\cdot 2^6 + 1\cdot 2^7 + 1\cdot 2^{10} + 1\cdot 2^{11} + 1\cdot 2^{14} + 1\cdot 2^{15} + 1\cdot 2^{18} + 1 \cdot 2^{19+} + 1\cdot 2^{22} + 1\cdot 2^{24} + 1\cdot 2^{25} + 1\cdot 2^{26} + 1\cdot 2^{27} + 1\cdot 2^{28} + 1\cdot 2^{29} + 1\cdot 2^{31} = 3217870029$$

Unsigned binary for b: 10111111110011001100110011001101 = 3217870029

Twos-complement binary

Two complement has both positive and negative numbers. For a 32 bit the positive range from 0 to 2,147,483,647 ($2^{31}-1$) and negative number -2,147,483,648 (-2^{31}). To tell whether or not the number is positive or negative we have to look at the most significant bit (MSB). If the MSB is a 1 then the number is negative, else the number is positive.

Since a = 4396E000 = 01000011100101101110000000000000. We know that this two's complement number is positive and should be the same as the unsigned.

$$Two's\ complement\ of\ a = 1133961216$$

Since b = BFCCCCCD = 10111111110011001100110011001101. We know that this two's complement number is negative.

$$Two's\ complement\ of\ b = 1\cdot 2^0 + 1\cdot 2^2 + 1\cdot 2^3 + 1\cdot 2^6 + 1\cdot 2^7 + 1\cdot 2^{10} + 1\cdot 2^{11} + 1\cdot 2^{14} + 1\cdot 2^{15} + 1\cdot 2^{18} + 1\cdot 2^{19+} + 1\cdot 2^{22} + 1\cdot 2^{24} + 1\cdot 2^{25} + 1\cdot 2^{26} + 1\cdot 2^{27} + 1\cdot 2^{28} + 1\cdot 2^{29} + 1\cdot -2^{31} = -1077097267$$

Note: The other way of getting two's complement of a negative number would be to invert then add 1.

32-bit floating point number:

We would need to know the following.

Sign	Exponent	Fraction
1 bit	8 bits	23 bits

$$floating-point\ numbers = (-1)^{Sign}\cdot(1+Fraction)\cdot 2^{Exponent-Bias}$$

Looking at 4396E000 = 0-10000111-00101101110000000000000.

We get that the sign is 0, exponent is 135, and the fraction is 0.1787109375 (for fraction use $d\cdot 2^{-i}$).

Putting them together into the equation

$$floating\ point\ a = (-1)^0\cdot(1+.1787109375)\cdot 2^{135-127} = 301.75$$

Looking at BFCCCCCD = 1-01111111-00110011001100110011101.

We get that the sign is 1, exponent is 127, and the fraction is .6

$$floating\ point\ b = (-1)^1 \cdot (1 + .6) \cdot 2^{127-127} = -1.6$$

8. **Show how to convert these two numbers to a 32-bit unsigned number, a 32-bit twos-complement number, and a 32-bit floating point number. Round to integers *where needed*, and write your answers in hexadecimal (include all 32 bits in your final answer).**

a.	**-300**
b.	**81.2**

Again, choose an appropriate level of detail when showing your work. Also, you can double check your answers to this and previous problems using Google: "123456 in twos complement" or "123456 in single floating point", etc.

32-bit unsigned number:

You cannot assign a negative value to an unsigned number so -300 doesn't not have a 32-bit unsigned number.

Since 81.2 is not an integer, I will round this number to 81. To find the binary number for 81 I would use the method of dividing by two, and writing a 0 for every even number, and a 1 for every odd number (which I will subtract the current number by 1 to make it even again):

Decimal	Binary
81	1
40	0
20	0
10	0
5	1
2	0
1	1

So the decimal number 81 is 1010001. We will now need to convert that number into a 32-bit unsigned binary number.

$$81 = 0000 - 0000 - 0000 - 0000 - 0000 - 0000 - 0101 - 0001 = 00000051_{hex}$$

32-bit two's complement number:

First we find out the binary sequence for 300 then we use the following equation below to get -300.

$$-x = \bar{x} + 1$$

Decimal	Binary
300	0
150	0
75	1
37	1
18	0
9	1
4	0
2	0
1	1

x = 100101100 now we will invert this number and add one.

$$-300 = 011010011 + 1 = 011010100$$

Since this is a negative number, I would have to add 1 to the left.

-300 in 32-bits two's complement = 11111111111111111111011010100

$$-300 = 1111 - 1111 - 1111 - 1111 - 1111 - 1110 - 1101 - 0100 = FFFFFED4_{hex}$$

Now assuming that 81.2 is 81, everything should be the same as the previous section (32-bit unsigned number).

$$81 = 0000 - 0000 - 0000 - 0000 - 0000 - 0000 - 0101 - 0001 = 00000051_{hex}$$

32-bit floating point number:

Using some resources online. http://www.madirish.net/240 and https://www.youtube.com/watch?v=MIrQtuoT5Ak

1 bit	8 bits	23 bits
Sign	Exponent	Mantissa

Steps:

1. Determine the sign bit (1 for negative and 0 for positive)
2. Convert your decimal digit to binary
    a. Convert the integer (unsigned) first
    b. Then the decimals
        i. This sequence is $d \cdot 2^{-1} + d \cdot 2^{-2}$ *and so on*..this series is the same as ½ + ¼ + 1/8 + 1/16 ….
3. Normalize this number (no leading zero's and have 1 at the right of binary point, important to count the number of left shift)
4. The number of left shift is then add to 127 (the maximum number you can express with 8 bits). Express this number as an 8 bit binary for the exponent.
5. The mantissa is everything number to the left of the binary point in the normalize form.
6. Put all of this together to get the bit sequence for floating point number

For -300 we know that this is a negative number so the sign bit is 1

Sign	Exponent	Mantissa
1		

Next we will have to convert 300 into binary

300 = 100101100

Since this number does not have any decimal.

100101100.0000

Now we normalize this number by shift the binary point to the left 8 position

1.001011000000

Now we have our Mantissa which is everything to the left of the binary point

Sign	Exponent	Mantissa
1		00101100000000000000000

Now we add the number of left shift to 127 to get our exponent value

8 + 127 = 135 = 10000111

Now we have our floating point sequence

Sign	Exponent	Mantissa
1	10000111	00101100000000000000000

Now for our solution we have to convert to hexadecimal

Floating point for -300 = 1100-0011-1001-0110-0000-0000-0000-0000 = $C396000_{hex}$

Now for 81.2, we would follow the same steps

81.2 is a positive number so the sign bit is 0

Sign	Exponent	Mantissa
0		

Now converting 81 to binary

81 = 1010001

Now converting the decimal 0.2 into binary, multiply by 2 until we reach 0

0.2*2 = 0.4	0
0.4*2 = 0.8	0
0.8*2 = 1.6	1
0.6*2 = 1.2	1
0.2*2 = 0.4	repeated

0.2 = .0011001100…

Now we put them together

81.2 = 1010001.0011001100…

Now normalize this number by shifting to the left 6 position

1.0100010011001100

Using the shift number 6 add it to 127 then represent that number in binary

6+127 = 133 = 10000101

Now we have our floating point sequence

Sign	Exponent	Mantissa
0	10000101	01000100110011001100110

Floating point for 81.2 = 0100-0010-1010-0010-0110-0110-0110-0110= $42A26666_{hex}$

9. **Consider 32-bit floating point. What are the first two base ten integers greater than or equal to 10^8 (one hundred million) that can be *exactly* represented by a 32-bit floating point number? Show your work and/or reasoning before giving your answers.**

This problem we are looking at integers that is greater than or equal to 10^8. We know that the floating point exponent can hold 8 bits which as large as $2.0_{ten} \times 10^{38}$. To solve this we would have to convert the integers into 32-bit floating point to see if it can exactly represented by it.

For 10^8:

This is a positive number so the sign bit is 0

Sign	Exponent	Mantissa
0		

10^8 in binary is equal to 00000101111101011110000100000000

Since we do not have any decimals this binary becomes 00000101111101011110000100000000.00

Now we have to normalize this binary number which we shift 26 position to the left

1.01111101011110000100000000

Now add the number of left shift to 127

26+127 = 153 which in unsigned binary is 10011001

Putting this together we the 32-bit floating point sequence

Sign	Exponent	Mantissa
0	10011001	01111101011110000100000

Now put this into the equation below to see if it represent 10^8 exactly

$$floating - point\ numbers = (-1)^{Sign} \cdot (1 + Fraction/Mantissa) \cdot 2^{Exponent-Bias}$$

$$Mantissa = 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-8} + \cdots$$
$$= 0.4901161193847656$$

$$10^8 = (-1)^0 \cdot (1 + 0.4901161193847656) \cdot 2^{153-127} = 10^8$$

For 10^9:

0.4901161193847656

This number is positive so the sign bit is 0

Sign	Exponent	Mantissa
0		

10^8 in binary is equal to 11101110011010110010100000000

Since we do not have any decimals this binary becomes 11101110011010110010100000000.00

Now we have to normalize this binary number which we shift 29 position to the left

1.11011100110101100101000000000

Now add the number of left shift to 127

29+127 = 156 which in unsigned binary is 10011100

Putting this together we the 32-bit floating point sequence

Sign	Exponent	Mantissa
0	10011100	11011100110101100101000

$$Mantissa = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-9} + \cdots = 0.862645149230957$$

$$10^9 = (-1)^0 \cdot (1 + 0.862645149230957) \cdot 2^{153-127} = 10^9$$

For 10^{10}:

Doing the same thing we would get this 32-bit floating point sequence

Sign	Exponent	Mantissa
0	10100000	00101010000001011111001

$$10^{10} = (-1)^0 \cdot (1 + 0.1641532182693481) \cdot 2^{160-127} = 10^{10}$$

For 10^{11}:

Doing the same thing we would get this 32-bit floating point sequence

Sign	Exponent	Mantissa
0	10100011	01110100100001110110111

$$10^{11} = (-1)^0 \cdot (1 + 0.4551914930343628) \cdot 2^{163-127} = 9.9999998 \cdot 10^{10}$$

Answer: the first two base ten integers greater than or equal to 10^8 (one hundred million) that can be *exactly* represented by a 32-bit floating point number is 10^9 and 10^{10}. The problem is that the Mantissa cannot be that accurate with only 23 bits for very large numbers which is why we can't represent some number exactly for example 10^{11}.

10. **Find a definition for a floating point ULP, then explain it. Finally, given any number *n*, how would you determine the weight of one ULP for *n's* floating point representation?**

In the textbook ULP is the units in the last place. The number of bits in error in the least significant place of significand between the actual number and the number that can be represented. In the worst case the rounding of the actual number is halfway between two floating-point representations. To find the weight of one ULP would the difference of THE LEAST SIGNIFICANT BITS of the (n) actual number and the (n's floating point representation) number represented. For example 2.37_{ten} and 2.36_{ten} the ulp is off by 1.