Name: Mike Xie

Computer Science 605.611

Problem Set 2

(Appendix A of the textbook defines and describes all MIPS instructions. Instructions not identified as pseudo-instructions in appendix A are true-op instructions.
To demonstrate your understanding of the concepts, your answers on problem sets should be produced manually without the use of a simulator or any outside source other than a calculator.

1. Recall that **"true-op"** instructions are those that can be translated by the assembler into a single 32-bit built-in machine instruction. True-op instructions can only use addressing modes that are supported by the built-in machine instructions and cannot use immediate operands outside the range -32768 to +32767.

a) (3) The goal of the instruction sequence below is to place the sum of $t0 plus the decimal constant +350000 into register $t0. Write in the missing decimal immediate operands required to accomplish the goal. Use only decimal immediate operands that are allowed in true-op instructions.

350000 to binary = 0000 0000 0000 0101 0101 0111 0011 0000
350000 to hex = 0x0005 5730

Upper 16 bits = 0101 = (5) decimal = 0x0005
Lower 16 bits = 0101 0111 0011 0000 = (22,320) decimal = 0x5730

```
lui     $1, 5
ori     $1, $1, 22320
addu    $t0, $t0, $1
```

b) (3) The goal of the instruction sequence below is to place the sum of $t0 plus the decimal constant -350000 into register $t0. Write in the missing decimal immediate operands required to accomplish the goal. Use only decimal immediate operands that are allowed in true-op instructions.

-350000 to binary with 2's complement = 1111 1111 1111 1010 1010 1000 1101 0000
-350000 to hex = 0xFFFA A8D0

Upper 16 bits = 1111 1111 1111 1010 = (-6) decimal = 0xFFFA
Lower 16 bits = 1010 1010 1101 0000 = (-22320) decimal = 0xA8D0

```
lui     $1, -6
ori     $1, $1, -22320
addu    $t0, $t0, $1
```

c) (3) The goal of the instruction sequence below is to place the sum of $t0 plus the decimal constant -32876 to register $t0. Write in the missing decimal immediate operands required to

accomplish the goal. Use only decimal immediate operands that are allowed in true-op instructions.
-32876 to binary with 2's complement = 1111 1111 1111 1111 0111 1111 1001 0100
-32876 to hex = 0xF329

Upper 16 bits = 1111 1111 1111 1111 = (-1) decimal = 0xFFFF
Lower 16 bits = 0111 1111 1001 0100 = (32660) decimal = 0x7F94

```
lui     $1, -1
ori     $1, $1, 32660
addu    $t0, $t0, $1
```

d) (3) The goal of the instruction sequence below is to place the sum of $t0 plus the decimal constant +32768 to register $t0. Write in the missing decimal immediate operands required to accomplish the goal. Use only decimal immediate operands that are allowed in true-op instructions.

32768 to decimal = 1000 0000 0000 0000 = 0x8000

```
ori     $1, $0, 32768
addu    $t0, $t0, $1
```

2. For each of the decimal integers listed below, what is the minimum number of bits required for the internal two's complement representation if only one sign bit is included?

a) (3) 30800

convert to binary
(30800) decimal = 0111 1000 0101 0000
A minimum of 16 bits are required.

b) (3) -32768

convert to binary using 2's complement
(-32768) decimal = 1000 0000 0000 0000
Minimum number of bits required is 16.

c) (3) -6

convert to binary using 2's complement
(-6) decimal = 1010
A minimum of 4 bits are required.

d) (3) 32768

convert to binary
(32768) decimal = 0 1000 0000 0000 0000
At least 17 bits minimum.
If we use 16 bits, then the msb will be one, and we will see a negative number.
So we need at least 17 bits at a minimum to represent this number.

3.(5) A system uses 16 bits to represent signed integers in excess-4000 form (i.e., the bias is 4000). Fill in the blanks to show the range of signed integers that can be represented in this excess-4000 system. The range is ______-4,000______ to ____61,535______.

A normal 16 bit range for signed integers is 2^(16) - 1 is 65535

With a bias of 4000. We need to subtract 4000 to get a new range.

[-4,000, 61,535]

4. (5) Based on the instruction mnemonic (i.e., its name) and on the instruction's operands, indicate whether each of the following instructions is a true-op instruction or a pseudo-instruction.

```
li      $4, -72             # pseudo-instruction. Translates to lui and ori or addi
lui     $5, 0x8000          # true-op instruction.
addiu   $,3, $9, 51902      # pseudo-instruction. Immediate out of range.
```

```
ori     $7, $0, 0xCABE          #true-op
andi    $11, $6, -13634         #true-op
```

5. Assume floating point register $f8 contains the 32-bit floating point representation of the value -2.5. CPU register $4 contains the 32-bit two's complement representation of 30. Registers $f8 and $4 are set to these patterns prior to executing each of the instruction sequences listed below.

a) (5) The following pair of instructions places a 32-bit pattern into register $2. What <u>decimal</u> integer has the resulting pattern as its two's complement representation?

$f8 is a floating point register that contains the 32-bit floating point representation of -2.5.

Let's convert it to binary.
The IEEE – 754 single precision floating point format requires
1 sign bit, 8 exponent bits, and 23 fraction bits

-2.5 is a negative number, so the sign bit will equal 1
Sign bit = 1

We need to convert -2.5 to 2.5 in binary
(2.5) decimal = 10.1 in binary
Sanity check
(1*2^1) + (0*2^0) + (1*2^-1) = 2 + 0 + .5 =2.5

We need to convert it to scientific notation
10.1 = 1,01 *2^1

Assume we use excess-127 form
Exponent = 1 + 127
Exponent = 128
Convert to binary
(128) decimal = 1000 0000

The mantissa fraction
M = 0100 0000 0000 0000 0000 000

Therefore $f8 will contain the floating point in binary form as
1 1000 0000 0100 0000 0000 0000 0000 000
In hex form
0xC0200000

$4 contains the 32-bit two's complement representation of 30.
This is easier to compute, we just convert from decimal to binary
(30) decimal = 0000 0000 0000 0000 0000 0000 0001 1110
In hex form
0x0000001E

```
mfc1    $2, $f8
addu    $2, $2, $4
```

mfc1 $2, $f8
Move floating-point register $f8 to CPU register $2
When we move from a floating-point register to a CPU register, we just do a simple move.
in $f8 contains the hex value of 0xC0200000
Register $2 will now have the value in hex 0xC0200000.
Which can be represented with the decimal value of -1071644672.
The binary representation of -1071644672 in 2's complement is =
11000000001000000000000000000000

addu $2, $2, $4
register $2 will contain the sum of $2 and $4.
$2 = -1071644672 + 30
$2 = -1071644642
The decimal value of $2 is -1071644642

b) (5) The following pair of instructions places a 32-bit pattern into register $2. What decimal integer has the resulting pattern as its two's complement representation?

Assume that $f8 and $4 has not been updated and still has the same value
$f8 will contain the floating point value of -2.5 in binary form as
1 1000 0000 0100 0000 0000 0000 0000 000
In hex form
0xC0200000

$4 contains the 32-bit two's complement representation of 30.
This is easier to compute, we just convert from decimal to binary
(30) decimal = 0000 0000 0000 0000 0000 0000 0001 1110
In hex form
0x0000001E

Assume $0 has the value of 0.

```
mtc1    $4, $f8
xor     $2, $4, $0
```

mtc1 $4, $f8
move $4 to $f8. From CPU register to Coprocessor 1 register
So now register $f8 holds the value that register $4 had.
So register $f8 and $4 hold the value 0x0000001E

xor $2, $4, $0
do a logical XOR of $4 and $0 and store the value into $2
$4 has the value of 0x0000001E
$0 has the value of 0x00000000
When we do the XOR
Register $2 will have the value of 0x0000001E
Register $2 has the decimal value of 30

c) (5) The following instruction sequence places a 32-bit pattern into register $2. What decimal integer has the resulting pattern as its two's complement representation?

Assume that $f8 and $4 has not been updated and still has the same value
$f8 will contain the floating point value of -2.5 in binary form as
1 1000 0000 0100 0000 0000 0000 0000 000
In hex form
0xC0200000

$4 contains the 32-bit two's complement representation of 30.
This is easier to compute, we just convert from decimal to binary
(30) decimal = 0000 0000 0000 0000 0000 0000 0001 1110
In hex form
0x0000001E

Assume $0 has the value of 0.
Assume $f6 starts out with the value of 0

```
cvt.w.s  $f6, $f8
mfc1     $2, $f6
xor      $2, $4, $2
```

cvt.w.s $f6, $f8
convert single to integer. From $f8 convert to an integer and put it into $f6
In $f8 we have the decimal value -2.5 as a float with the hex value as 0xC0200000.
The float is turned into an integer, but still stored in the FPU.
$f6 will have the value of -2, but it will still be in the coprocessor register and stored in an integer 2's complement view rather than a floating point. So, its hex value will be 0xFFFFFFFE

mfc1 $2, $f6
move floating point register $f6 to CPU register $2.
When we do this move we only move the data. The value will stay the same.
$f6 has the value 0xFFFFFFFE.
$2 now has the value 0xFFFFFFFE. And the current decimal value will be -2.

xor $2, $4, $2
Do a logical xor of $4 and $2 and store the result into $2.

$4 has the value in hex form 0x0000001E
$2 has the value in hex form 0xFFFFFFFE

XOR the values in binary
0000 0000 0000 0000 0000 0000 0001 1110
1111 1111 1111 1111 1111 1111 1111 1110

1111 1111 1111 1111 1111 1111 1110 0000

Now that we have the binary value of $2. We need to convert it to decimal
1111 1111 1111 1111 1111 1111 1110 0000
invert
0000 0000 0000 0000 0000 0000 0001 1111
Plus 1
0000 0000 0000 0000 0000 0000 0010 0000
Which equals to 32
Because we did 2's complement backwards, the value must be -32.

Register $2 has the decimal value of -32

6. (5) Floating point register $f8 contains the 32-bit floating point representation of -8761. The following instruction is then executed:

```
cvt.s.w   $f6,$f8
```

Use eight hex digits to show the resulting bit pattern contained in $f6.
$f6 = **0xCE67DC70**

In register $f8, we need to convert -8761 to hex in floating point representation.

1 sign bit, 8 exponent bits, and 23 mantissa bits.

Sign bit = 1

Convert 8761 to binary
0010 0010 0011 1001

Convert to scientific notation
1.0001000111001 * 2^13

Using excess of 127
Exponent = 13 + 127
Exponent = 140
Convert to binary
Exponent bits = 1000 1100

Mantissa bits = 0001 0001 1100 1000 0000 000

The floating point representation of -8761 is
1100 0110 0000 1000 1110 0100 0000 0000
In hex it is
0xC608E400

So, register $f8, has the hex value 0xC608E400

cvt.s.w $f6, $f8
converts from integer to float. Convert $f8 as integer to a float and store into $f6

$f8 currently has the hex value 0xC608E400
We convert that to an integer representation.
0xC608E400 as an integer is -972495872
We now need to convert it back into a float.
Sign bit = 1
Convert to binary
0011 1001 1111 0111 0001 1100 0000 0000

Convert to Scientific notation
1.1 1001 1111 0111 0001 1100 0000 0000 *2^29
Exponent = 29 + 127
Exponent bits = 156 = 1001 1100

Mantissa = 1100 1111 1011 1000 1110 000
So the binary of the floating point representation is

1100 1110 0110 0111 1101 1100 0111 0000
Convert that to floating point binary to hex value.
0xCE67DC70
Then move that value into co-processor register $f6
So $f6 will have the value of 0xCE67DC70

7. (5) If each character is encoded as an 8-bit ASCII character, a 32-bit register can contain 4 characters. Assume register $3 contains the two's complement representation of +1094861636 and register $4 contains the two's complement representation of +589373213.#After the instruction addu $2,$3,$4 is executed, show the 4 ASCII characters represented by the 32-bit pattern produced in register $2. Display the ASCII characters (not their 8-bit codes) from left to right (i.e., from the high byte to the low byte within the register).

$3 contains the decimal value of 1094861636
$4 contains the decimal value of 589373213

Execute the addu instruction
addu $2, $3, $4
add register $3 and register $4 and put the sum into register $2.
1094861636 + 589373213 = 1684234849
Convert to binary and hex
= 0110 0100 0110 0011 0110 0010 0110 0001
= 0x64636261
Convert the hex values into ASCII characters from left to right.
0x64 = "d"
0x63 = "c"
0x62 = "b"
0x61 = "a"
0x64636261 = "dcba"
The ASCII characters in $2 are "dcba".

8. Since the MIPS processor uses 32-bit addresses, it can reference any location within the 4-GB address space. The bytes in memory are numbered consecutively starting from 0x00000000 at the low end of memory up to 0xFFFFFFFF at the upper end of memory.

a) (5) The jump instruction j loop transfers control to the instruction to which the label "loop" is attached (i.e., it jumps to loop). Assume the machine code for this jump instruction is located at memory address 0x20CE88C0 and the label "loop" corresponds to memory address 0x2C4E088C. What is the 32-bit machine code for this jump instruction? Express your answer as an 8-digit hex number. The jump instruction is described in modules 1 and 2 as well as in appendix A of the textbook.

Let's figure out the format for a jump instruction.

The jump instruction is laid out like this
j target

the j / opcode requires 6 bits.
The target requires 26 bits.

The opcode for j instruction is 000010

The target that requires 26 bits.

We need to calculate how much forward we are moving in the program.
The left 4 bits of 0x20CE88C0 = 0x2
The left 4 bits of 0x2C4E088C = 0x2
So, they are in the same "256 MB region". A direct jump is possible.

We want the 26 bits that gives us the lower 28 bits after shifting left by 2 but ignoring the top 4 bits.

0x2C4E088C needs to be bit shifted by 2 bits to the right. (because the lowest 2 bits are always 0)
0x2C4E088C = 0010 1100 0100 1110 0000 1000 1000 1100
Right shift 2 to remove the always 0 of the last 2 bits
0000 1011 0001 0011 1000 0010 0010 0011 = 0x0B138223

We still have 28 bits, but we only need 26 bits. We need to reduce the bits down to 26 so that it will fit inside the machine instruction.
I'm just going to drop the left most bits until I get 26 bits.
0000 0011 0001 0011 1000 0010 0010 0011 = 0x03138223
Add in the op code of 0000 10
0000 1011 0001 0011 1000 0010 0010 0011 = 0x0B138223

Therefore, the machine instruction for the jump is 0x0B138223

b) (5) The conditional branch instruction beq $0, $0, exit always transfers control to the instruction to which the label "exit" is attached (since it compares register $0 with itself). Assume the branch instruction is located at memory address 0x20CE68C0 and the label "exit" corresponds to memory address 0x20CE88C4. What is the 32-bit machine code for this beq instruction? Express your answer as an 8-digit hex number. The beq instruction is described in modules 1 and 2 as well as in appendix A of the textbook.

The format of beq is 6 bits for opcode, 5 bits for rs, 5 bits for rs and 16 bits for offset.

The opcode for beq is 4
So the opcode in binary looks like this, 0001 00

The register $0 will have the value 0, since they are the zero registers.
And they will both have the same value of. 0000 0

We are jumping forwards.
The exit label has a higher memory address value
0x20CE68C0 < 0x20CE88C4

Another thing to note is that by the time the instruction is fetched, the PC counter would have incremented by 4. So, the PC of 0x20CE68C0, will be +4. And instead have a PC counter of 0x20CE68C4.

So, we need to find the offset between 0x20CE68C4 and 0x20CE88C4.

Let’s find the difference
0x20CE88C4
0x20CE68C4

0x00002000

That is the offset, 0x00002000.
Let’s turn that into binary
0000 0000 0000 0000 0010 0000 0000 0000
Let’s right shift 2, to get rid of the last 2 bits.
0000 0000 0000 0000 0000 1000 0000 0000
We just need 16 bits for the offset.
0000 1000 0000 0000 = 0x 0800
Now let’s add everything together.
0001 0000 0000 0000 0000 1000 0000 0000
The hex value will be, 0x10000800

c) (5) A jump instruction transfers control to some 32-bit memory address within the instruction memory. Assume the 32-bit machine instruction at that address is 0x00000000. This target instruction (0x00000000) is an R-type instruction. Indicate the ALU operation performed by this R-type instruction and indicate the value contained in the result register.

While the first 6 bits is all 0, and when that condition is met, then it is an R-type instruction.

However, the entire instruction is all 0s.
This indicates a no operation.
A no operation instruction (nop) has the machine instruction of 0x00000000, which matches the instruction given.

However, no op is not an R-type instruction. So, maybe that isn’t the answer you are looking for.
Another answer would be Shift Logical Left, or sll. That operation is an R-type instruction.
sll needs an op code of 0, and a function code of 0.
However, it won’t do anything since, its doing a bit shift of 0 to the left 0 times, and then storing in the register 0, which is hardcoded to have 0.
So the hardware is performing, $0 ← $0 << 0

The result register will have the value of 0.
The ALU operation performed will be a bit shift of 0 to the left by 0 bits.

It effectively does nothing, and is equivalent to a no-op. but it is still an R-type instruction.

9. Assume CPU register $11 is refilled with the pattern 0xCAFEF00D prior to executing each of the instructions listed below. Use 8 hex digits to show the contents of the result register after each instruction is executed.

a) (3) addiu $10, $11, -2   contents of $10 = 0xCAFEF00B

We are adding $11 with -2, and then storing it into $10
$11 has the value 0xCAFEF00D,
Convert it to decimal
-889262067
Add them together
-889262067 + -2 = -889262069
Convert the sum to hex value
0xCAFEF00B

b) (3) addi $9, $11, -2   contents of $9 = 0xCAFEF00B

We are adding $11 with -2, and then storing it into $9
$11 has the value 0xCAFEF00D,
Convert it to decimal
-889262067
Add them together
-889262067 + -2 = -889262069
Convert the sum to hex value
0xCAFEF00B

c) (3) ori $8, $11, -2   contents of $8 = 0xFFFFFFFF

We need to do a logical or of 0xCAFEF00D and -2
Convert 0xCAFEF00D to binary
1100 1010 1111 1110 1111 0000 0000 1101

Convert -2 to binary
1111 1111 1111 1111 1111 1111 1111 1110

Do a logical OR
1100 1010 1111 1110 1111 0000 0000 1101
1111 1111 1111 1111 1111 1111 1111 1110

1111 1111 1111 1111 1111 1111 1111 1111

Convert the result to hex value
0xFFFFFFFF

d) (3) xori $7, $11, -2   contents of $7 = 0x35010FF3

We need to do a logical xor of 0xCAFEF00D and -2

Convert 0xCAFEF00D to binary
1100 1010 1111 1110 1111 0000 0000 1101

Convert -2 to binary
1111 1111 1111 1111 1111 1111 1111 1110

Do a logical XOR
1100 1010 1111 1110 1111 0000 0000 1101
1111 1111 1111 1111 1111 1111 1111 1110

0011 0101 0000 0001 0000 1111 1111 0011

Convert the result to hex value
0x35010FF3

e) (3) andi $6, $11, -2 contents of $6 = 0xCAFEF00C

We need to do a logical and of 0xCAFEF00D and -2
Convert 0xCAFEF00D to binary
1100 1010 1111 1110 1111 0000 0000 1101

Convert -2 to binary
1111 1111 1111 1111 1111 1111 1111 1110

Do a logical AND
1100 1010 1111 1110 1111 0000 0000 1101
1111 1111 1111 1111 1111 1111 1111 1110

1100 1010 1111 1110 1111 0000 0000 1100

Convert the result to hex value
0xCAFEF00C

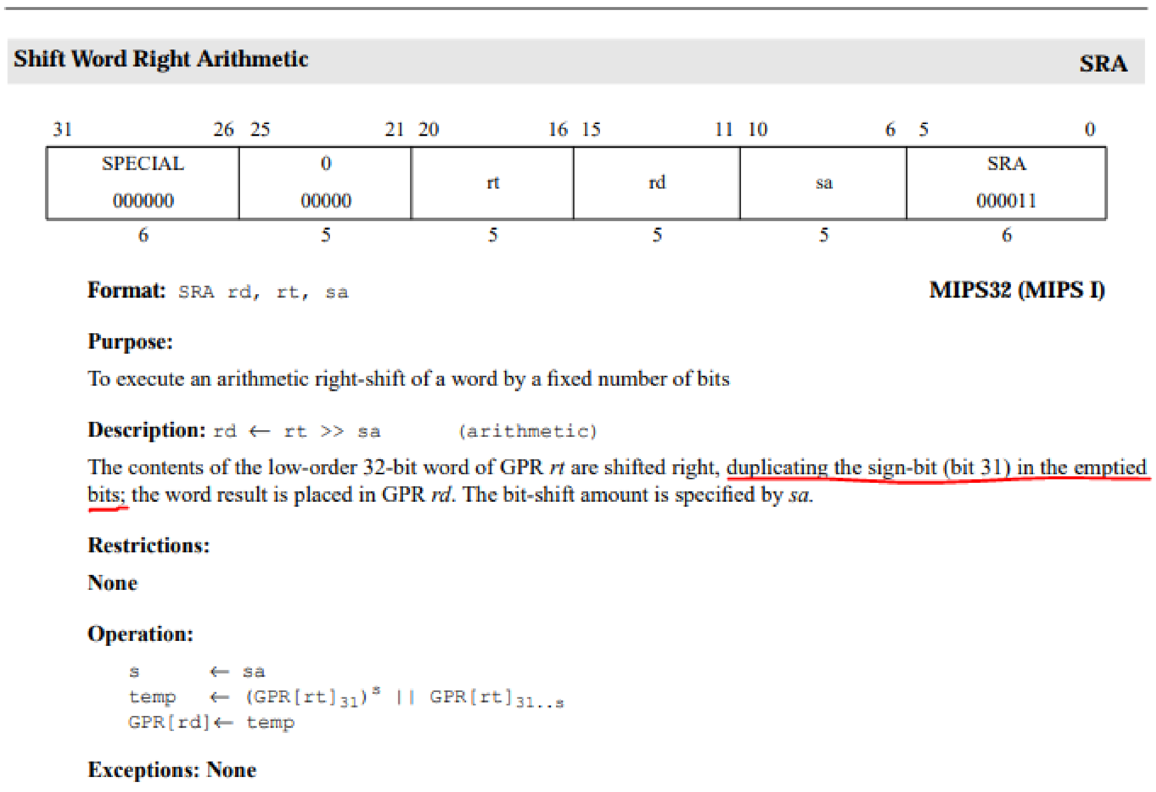
f) (3) sra $5, $11, 6 contents of $5 = 0xFF2BFBC0

We need to shift the bits of register $11 6 times to the right. Then store the resultant into register $5.

Convert 0xCAFEF00D to binary
1100 1010 1111 1110 1111 0000 0000 1101

Shift 6 times to the right, keeping the signed bit
1111 1111 0010 1011 1111 1011 1100 0000

According to the MIPS architecture for programmers, when you do a right shift, you must duplicate the sign bit, every time you do a right shift. This makes it so that the MSB, which is the sign bit, stays on and is sticky when you shift right.



Shift Word Right Arithmetic **SRA**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format: `SRA rd, rt, sa` **MIPS32 (MIPS I)**

Purpose:

To execute an arithmetic right-shift of a word by a fixed number of bits

Description: `rd ← rt >> sa (arithmetic)`

The contents of the low-order 32-bit word of GPR *rt* are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR *rd*. The bit-shift amount is specified by *sa*.

Restrictions:

None

Operation:

```
s       ← sa
temp    ← (GPR[rt]_31)^s || GPR[rt]_31..s
GPR[rd]← temp
```

Exceptions: None

So, the result, when we keep the sign bit is.
1111 1111 0010 1011 1111 1011 1100 0000

Then we convert to hex value
0xFF2BFBC0

10. (5) Based on Booth's algorithm, how many additions and how many subtractions should be performed in multiplying the decimal integer 9040 by the decimal multiplier -592 .

Multiplicand = 9040
Multiplier = -592

Convert 9040 to binary
0010 0011 0101 0000

Convert -592 to binary
1111 1101 1011 0000

We need to examine each pair of bits
(bi, bi-1) = (0,1), then we add the multiplicand to the partial product
(bi, bi-1) = (1,0), then we subtract the multiplicand to the partial product
(bi, bi-1) = (0,0) or (1,1) we do not add or subtract.

We also should perform an arithmetic right shift every time, whether or not we added or subtracted or neither.

So, the total number of additions is how many times we see an 01 pattern.
The total number of subtractions is how many times we see an 10 pattern.

The additions that fit our pattern
1111 11$\underline{01}$ 1$\underline{01}$1 0000
0000 00+10 0+100 0000
There are only 2 additions that we do.

The subtractions that fit our pattern
1111 1$\underline{10}$1 $\underline{10}$1$\underline{1\ 0}$000
0000 0-100 -100-1 0000
There are only 3 subtractions that we do.
0000 0-1+10 -1+10-1 0000
So, we do a total of 2 additions and 3 subtractions using booths algorithm.

11. (3) Write down a MIPS assembly language instruction that adds the immediate operand -592 to register $t0. Use hex to express the immediate operand.

We can use the addi to add the immediate to register $t0.
Convert -592 to binary
592 in binary = 0000 0010 0101 0000
Convert to negative using one's complement
1111 1101 1010 1111
Use two's complement
1111 1101 1011 0000
Convert to hex value
0xFDB0

addi $t0, $t0, 0xFDB0