Tutorial 3
CSE 112 Computer Organization

Ans 1

Part - a

Number = 1001100 Number of bits = 7

i. Unsigned representation = $1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 76$ Max value = $2^7 - 1 = 127$ (All bits are 1)
Min value = 0 (All bits are 0)

ii. signed magnitude:- We use the first bit as the sign of the number. For the given binary number, the first bit is 1, which means that the number is negative. We use the other bits to represent the magnitude of the number like an unsigned number.
Sign = negative
Magnitude = $0*2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 12$
Number = -12
Max_value = $2^6-1 = 63$ (First bit is 0 all other bits are 1)
Min value = $-(2^6-1) = -63$ (All bits are 1)

iii. 2's complement representation:- We can see the number is negative as the most significant bit (MSB) is 1.
We cannot directly find the value in decimal representation.

2's complement of 1001100 = 0110100
Sign = Negative

$-(0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0) = -52$
Max value = $(2^6 - 1) = 63$ (first bit 0 all other bits 1)
Min value = $-(2^6) = -64$ (first bit 1 all other bits 0)

Part -b

i. Unsigned representation = 12
Max value = 127 (All bits are 1)
Min value = 0 (All bits are 0)

ii. Sign = positive
Magnitude = $0*2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 12$
Number = 12
Max_value = $2^6-1 = 63$ (First bit is 0 all other bits are one)
Min value = $-(2^6-1) = -63$ (All bits are 1)

iii. 2's complement representation:- We can see the number is positive as the most significant bit (MSB) is 0
We can directly find the value in decimal representation.
Sign = Positive
$+(0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0) = 12$

Max value = (2^6 - 1)= 63 (first bit 0 all other bits 1)
Min value = -(2^6) = -64 (first bit 1 all other bits 0)

Part -c
It is not possible to get the equivalent decimal representation if we do not have the representation with us.

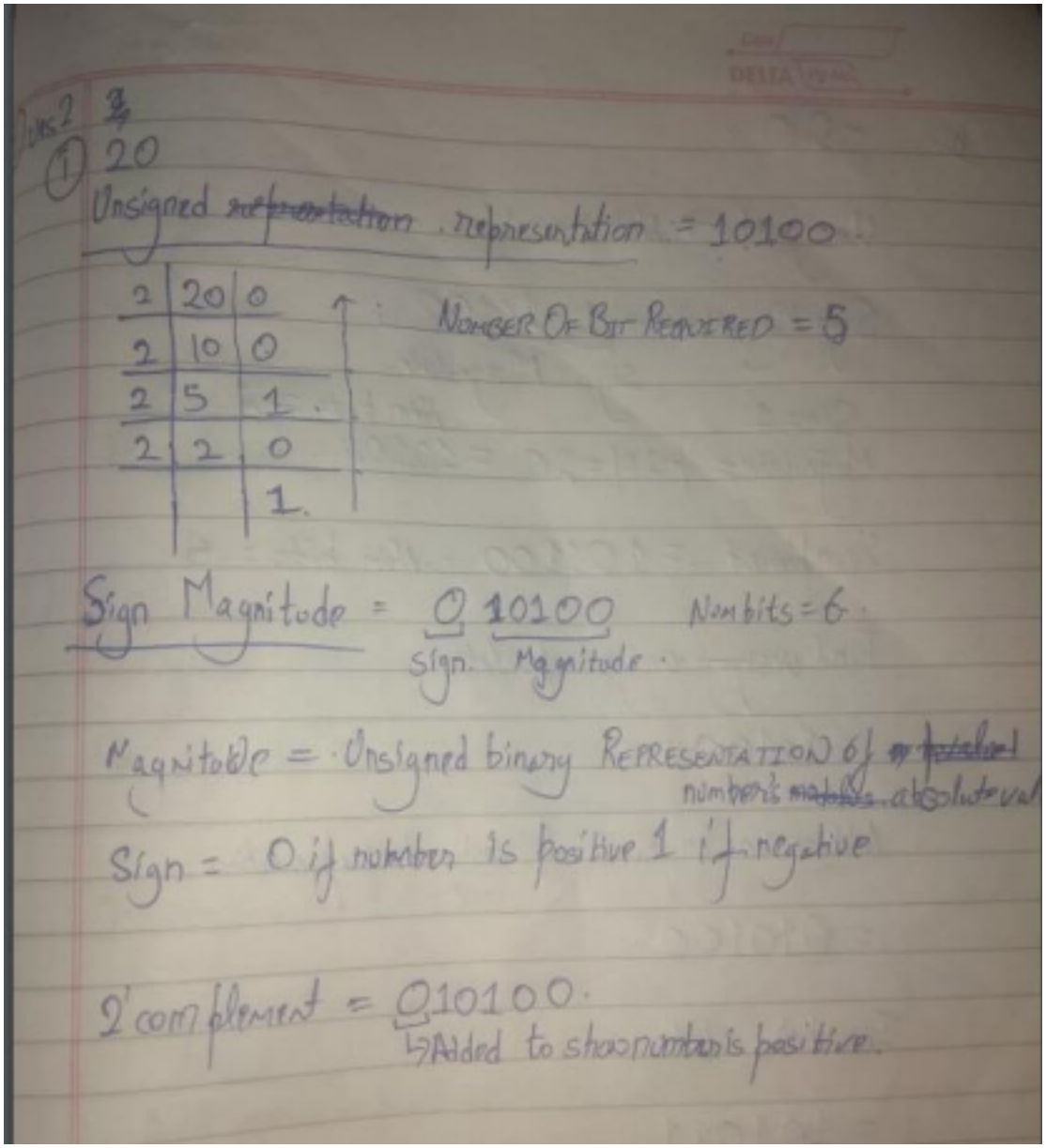
Ans 2

a. 01010
b. 001010
c. 10110
d. 110110

Suppose a number can be represented by k bits in 2's complement notation. Then the same number can be represented by m+k (m > 0) bits by simply making the m MSBs the same as the k^{th} bit. This is called sign extension.

NOTE to the TAs: The objective of this question is to familiarize students with sign extension, which is required in the next question.
Please discuss how this concept applies to variable casting as well. For example, how an 8 bit value is casted to 32 bits.

Ans 3

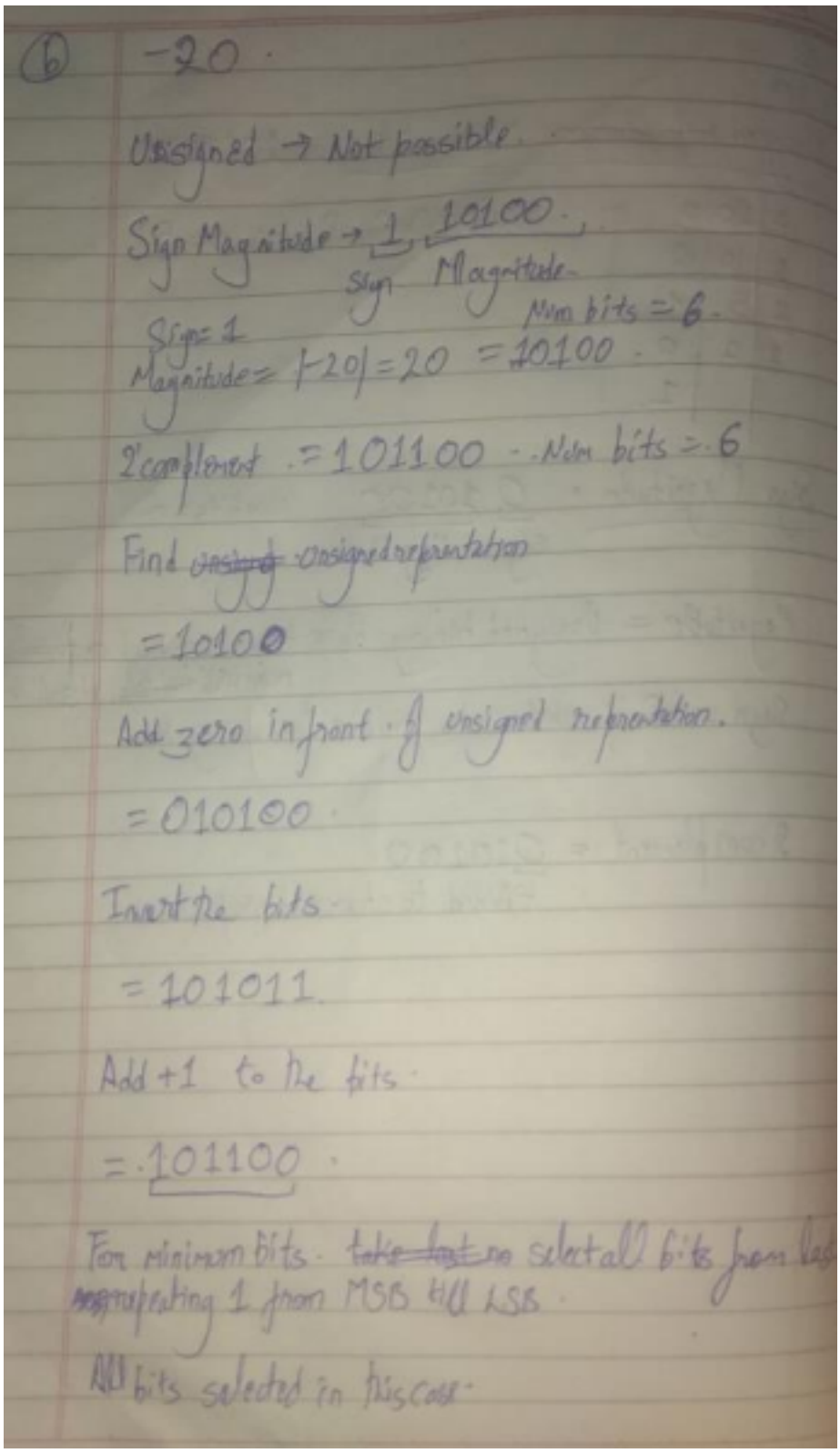


For signed magnitude:

Magnitude = Unsigned binary representation of the absolute value of the number

In this case number = 20, absolute value of number = 20

Hence magnitude = 10100



In case of 2's complement:
For the minimum number of bits, ignore sign extension, if present.

c) 32.

Unsigned := 100000 Num bits = 6.

Sign Magnitude := 0 100000 Num bits = 7.
Sign Magnitude.

2'complement = 0100000 Num bits = 7.

d) -32.

Unsigned Not possible.

Sign Magnitude = 1 100000 Num bits = 7.
Sign Magnitude

2's complement 100000 Num bits = 6.

Unsigned Representation of (-32) = 100000

Add zero infront = 0100000

Invert the bits = 1011111

Add+1 to the bits = 1100000
↳ Min bit representation.

Select last repeating 1 from MSB till LSB

Min bits 100000.

NOTE to the TAs: In 2's complement notation, repeated MSBs are redundant. For example:

1001 in 2's complement 4-bit notation
= 11001 in 2's complement 5-bit notation
= 111001 in 2's complement 6-bit notation
… and so on.

Therefore, to find the minimum number of bits for representation, we can take the bits from the last repeating MSB till the LSB, which is 1001.

Ans 4

a. Each character in the sequence can take one of k symbols. Therefore for a sequence of length n we can have at max k^ndifferent numbers.

b. We clearly need m values to represent each number in the set uniquely. A k-ary system sequence with q characters can represent at max k^qdifferent values. Therefore we need the smallest q, which satisfies $k^q >= m$. This means that $q = ceil(\log_k m)$.
For instance, if we are working with a 4-ary system (k = 4), then we have 4 symbols: 0, 1, 2 and 3. Now suppose the m = 20. Therefore we need a sequence of length 3 ($ceil(\log_4 20) = ceil(2.16) = 3$). With length 3 we can uniquely identify 4^3= 64 different elements. With length 2 we can uniquely identify 4^2= 16 different elements. Since we need to identify 20 elements, we need at least a 3 length sequence.

c. As k increases, we will need shorter and shorter sequences. Therefore the length decreases. For instance, binary sequences are longer than their corresponding decimal notation.

d. In most computer systems, different symbols are represented by different voltage levels. To distinguish between these symbols, we need to distinguish between different voltage levels. It is considerably easier to do this when we have just two symbols. A valid representation for such a case can be: GND for 0, and some non-zero voltage for 1 (and vice-versa). As we increase the number of symbols, the difference in voltage levels between two successive symbols decreases (for a fixed highest possible voltage), which increases the possibility of misrepresentation.

Ans 5

k bit 2's complement notation can represent 2^kdifferent numbers, whereas signed magnitude can represent 2^k-1 different numbers. This is because in signed magnitude notation, there are two symbols for 0, which represent +0 and -0. In arithmetic, 0 has no sign. Therefore, multiple representations for 0 can be seen as a disadvantage for signed magnitude notation.

A weighted number system uses a weighted sum to represent a number. An example of a weighted system is the decimal system. For example, if you have the number 123 in decimal, 1 is weighted with 100, 2 is weighted with 10 and 3 is weighted with 1, i.e. 123 = 1 * 100 + 2 * 10 + 3 * 1. The weights are well-defined for each position.

2's complement is a weighted system.
Weights of a n bit 2's complement number are **$-(2^{n-1}), 2^{n-2}, \ldots, 2^1, 2^0$.**
Example of 4 bit 2'complement number 1111 = $1 * -(2^3) + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = -1$
Sign magnitude is an unweighted system. The first bit in sign magnitude does not have any weight assigned, hence unweighted.

Now you can try to figure out other properties of the 2's complement system. For example, why sign extension works:

Suppose $n = b_{x-1} * -(2^{x-1}) + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$, where b_i is the i^{th} bit in the 2's complement notation of n.
Suppose we sign extend it by k bits.
$b_{x-1} * -(2^{x+k-1}) + \ldots. + b_{x-1} * 2^x + b_{x-1} * 2^{x-1} + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0 =$
$b_{x-1} * \{-(2^{x+k-1}) + \ldots. + 2^x + 2^{x-1}\} + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$
$= b_{x-1} * 2^{x-1}\{-(2^{k-1}) + 2^{k-2} + \ldots. + 2^1 + 2^0\} + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0 =$
$b_{x-1} * 2^{x-1}\{-(2^{k-1}) + 2^{k-1} - 1\} + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$
$= b_{x-1} * 2^{x-1}\{- 1\} + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$
$= b_{x-1} * -(2^{x-1}) + b_{x-2} * 2^{x-2} + \ldots + b_1 * 2^1 + b_0 * 2^0$
$= n$

Additional link for the properties of 2's complement

NOTE to the TAs: The last part is just for extra information. Please see to it accordingly.

Ans 6

If we add 01111111 + 00000001 then there will be an overflow. Since registers are only 8 bit wide, therefore the largest number we can store is $2^7-1 = 127$. However, we are computing 127 + 1 (= 128), which requires 9 bits in 2's complement notation. As a result, this computation will overflow and will generate 10000000 = -128, the wrong answer. This happened because we needed more than 8 bits to store the result of the operation in 2's complement notation. That's why the result in Z will not be correct. 128 has the representation of 010000000. Z will take the 8 LSBs of this, i.e. 10000000.
You can detect this by checking for overflows. This can be done easily by looking at the signs of the operands and the sign of the result. If they do not make sense, then there was an overflow. For example, in this case, the sign of both the operands was +ve, but the sign of the result was - ve. This does not make sense.
Most ISAs have a special register, often called the **flags** or the **status register**, which tracks the state information of the machine. A part of this state can be whether the last operation overflowed or not. In such a case, the flags register can have a bit dedicated to detecting overflows. If the bit is set to 1, this means the last operation overflowed (or vice-versa).

Ans 7

a. The highest possible sum is d x (2^n-1) (at worst). This can be represented in ceil($\log_2$(d x (2^n-1))) = n + ceil($\log_2$d). For example, if we add four 2-bit binary numbers 11 + 11 + 11 + 11 = 3+3+3+3 (in decimal), we get 12. We need 4 bits to represent 12 (2 + ceil($\log_2$(4))).
b. We can consider the multiplication of two n and m bit numbers as (2^m-1) (at worst) repeated addition of n bit numbers. Therefore, we need n + ceil($\log_2(2^m-1)$) = m+n bits.

NOTE to the TAs: The objective of this question is to promote analytical skills by forcing students to evaluate how unsigned notation will work for arbitrary length numbers.

Ans 8

Way 1: We can throw an exception whenever an overflow occurs. This will force the programmer to write code that explicitly defines what to do in case of overflows.

Way 2: Another alternative is to define an ISA whose semantics are invariant of the size of a number. As a result, the ISA is expected to handle numbers of arbitrary length correctly. One way to do this might be storing a number initially using x bits. Whenever an overflow occurs, x is

incremented, and the operation which caused the overflow is re-executed. This incrementing of x is done until no overflow occurs.

Use 1: If the programmer wants to write code for a critical application, then he/she might not want unexpected and unhandled overflows. In such cases, unexpected and unhandled overflows might result in an undefined state, which might be dangerous.

Use 2: Suppose the programmer works with very large numbers and is worried about the memory footprint of the application. If a number is small, then it does not make sense to waste redundant bits for its representation, just to support larger numbers. The Way 2 described above can be used to handle such scenarios. In such microarchitectures, larger numbers are given more bits and smaller numbers less bits, resulting in a more optimal memory usage.

NOTE to the TAs: This is an open ended question. That's why we have kept it in the end. There is no fixed answer for this. The objective of this question is to promote problem solving skills.

Ans 9.

Solution:

```
Add r3, r1, r2
Sub r3, r3, r4
Sub r3, r3, r7
Add r3, r3, r3
Sub r3, r3, r1
Sub r2, r2, r1
```