1. General Questions

(a) [4%] The similarity among human program, high-level language program, and assembly language program is that they are all:
step by step instructions to solve a problem.

The major difference among these programs is:
the languages/features (human/compiler/assembler) they use.

(b) [4%] In the context of *stored program architecture*, stored program means:
that the executable program and data are stored in memory;

and *instruction cycle* means:

instructions are executed using the same steps/phases repeatedly.

(c) [4%] The phase of an instruction cycle during which memory *must* be accessed is:
instruction fetch.

The two phases of the instruction cycle during which memory *may* be accessed are:
operand fetch and result store.

(d) [4%] Assembly language of a processor exposes its hardware capabilities since
there is a direct (one-to-one) equivalence between the assembly language features and the hardware features.

There ___can___ (can/cannot) be different assemblers for the same processor since
different assemblers/assembly languages symbolically specify the same hardware.

(e) [4%] A processor supports different addressing modes in order to:
address the operands flexibly and appropriately like high-level languages.

The difference between base addressing mode and indexed addressing mode is:
The base address is changeable in base addressing whereas the index address is changeable in index addressing mode.

(f) [4%] The difference between byte-addressable memory and word-addressable memory is: each address corresponds to a byte in byte-addressable memory, and to a word in word-addressable memory.

Byte-addressable memories are widely used in modern computers because:
they use the minimum amount of memory to store program and data.

(g) [4%] An instruction that converts an ASCII coded decimal digit stored in register `al` to its signed integer equivalent and stores it in `al` or vice versa is:
`xor al, 30h`.

An instruction that changes the case of an ASCII coded alphabet in register `bl` is:
`xor bl, 20h`.

(b)
```
    eax = 80808080
+   ebx = 8F7F7F80
  ----------------
      00000000
```

From the above, we see that the result becomes positive and overflow has occurred. (Adding two negative integers resulting in a positive integer!) Hence any value more negative than 8F7F7F80 in ebx will cause overflow.

(c) Most positive integer = $2^{31} - 1$.
Most negative integer = -2^{31}.

(d) (i) –64 = 11000000 (in sign and magnitude representation)
(ii) –100 = 11100100

5. (a) [00000100h] = FFFFFFFF.
In other words, contents of memory location starting from 00000100h will be changed from FFFFFFFF to 80808080h.

(b) ebx = [00000100] = FFFFFFFFh.

(c) ebx = [00000110] = FFFFFFFFh.

(d) ebx = 31303334h.

6. (a) (i) eax = 80808080 + 00000100 = 80808180
(S,Z,C,O) = (1,0,0,0)
(ii) ax = 8080 – 0100 = 8080 + (FF00) = 7F80
(S,Z,C,O) = (0,0,1,1)

(b) Being restricted to use only two operands in an instruction complicates programming effort. For example, if we wish to evaluate an arithematic expression involving 10 terms, it has to be programmed using up to 9 instructions. This restriction is caused by the fact that a physical system (computer hardware) must be built with fixed physical parameters. In choosing only two operands per instruction, it lowers the space cost incurred in an instruction (both the storage space of the instruction and the processor cost in execution such an instruction). Of course, one may design a machine that allows more than 2 operands to be specified. However, typically this number is restricted to 3 or fewer.

7. The following program assumes that a line may have up to 9 vowels.

```
segment .bss
   buffer  resb 128          ; reserve 128 bytes for line input
   count   resb 1

segment .text
   global _start
   _start: mov   eax, 3
           mov   ebx, 0
           mov   ecx, buffer
           int   0x80
```

(e) [4%] The addressing modes used in instructions `S3, S4, S5, and S6` are:

`S3:` ___register, based-indexed___

`S4:` ___register, immediate___

`S5:` ___register___

`S6:` ___program counter relative___

(f) [4%] Change instructions `S2` and `S5` so that instruction `S3` can be rewritten using register indirect addressing mode.

Changed instruction `S2` is: ___mov ebx, str+2___.

Changed instruction `S5` is: ___dec ebx___.

Changed instruction `S3` is: ___mov ax, [ebx]___.

3. Program Execution and Control Structures

Consider the following program fragment

```
        mov   ecx, 1000
again:  loop  again
```

(a) [4%] If the `mov` instruction takes 4 clock cycles and the `loop` instruction takes 5 clock cycles to execute, the number of clock cycles required to execute the above program fragment is:

___4+5×1000 = 5004 clocks___.

(b) [4%] If the above program fragment is run on a processor with 1 GHz clock (in which each clock cycle takes 10^{-9} seconds), the total execution time of the above program is:

___5004×10^{-9} seconds___

(c) [4%] For the program to execute for *about* 1 second, the `mov` instruction above should be changed to: `mov ecx,` ___200,000,000___.

(d) [4%] The given loop _does not_ (does/does not) have any instruction(s) in its body.

A general use of such loops is to program ___time delay/waiting___

(e) [4%] Write a high-level language `for` statement equivalent to the above NASM program fragment.

Answer: `for i =` ___1000___ `to` ___1___ `by` ___-1___ `{`_____`}`.

(f) [4%] Write a NASM program fragment which is equivalent to the following high-level language statement: `for i = 1 to 1000 by 1 {}`.
Hint: You need to use the compare and conditional jump operations instead of `loop`.

```
        mov   ecx, 1          ; i = 1
again:                        ; {}
        inc   ecx             ; by 1
        cmp   ecx, 1000       ; to 1000
        jle   again           ; loop
```

4. Assembly Language Programming

A *palindrome* is a character string which reads the same left-to-right and right-to-left ignoring spaces and punctuations. For example, software is not a palindrome whereas racecar is one.

For this problem, you are going to write an assembly language program that reads a word (character string without any spaces or punctuations), checks if that word is a palindrome or not (that is, whether it reads the same left-to-right and right-to-left) and prints an appropriate message like one of following.

software is not a palindrome

racecar is a palindrome

(a) [6%] Given the declarations

```
segment .data
   pmsg    db   ' is a palindrome.', 0xA
   plen    equ  $-pmsg
   nmsg    db   ' is not a palindrome.', 0xA
   nlen    equ  $-nmsg

segment .bss
   string  db   40
```

write the sequence of instructions to read the input string and store it at string.

```
mov   eax, 3          ; read from
mov   ebx, 0          ; from standard input device
mov   ecx, string     ; and store the input string
mov   edx, 40
int   0x80
```

```
; A:  insert the indicated code from (d) here to print the input string
```

(b) [6%] Write a sequence of instructions to get the address of the first character in the input string in register `esi`, get the address of the last character in string in register `edi`, compare the characters pointed to by `esi` and `edi`.

```
        mov   esi, string     ; address of first character
        mov   edi, esi        ; address of last character
        add   edi, edx
        dec   edi
next:   mov   al, [esi]       ; load next character from left into al
        cmp   al, [edi]       ; compare with the next character from right
```

(c) [6%] Write a simple program loop that will check whether the input string is a palindrome by comparing the first character with the last, the second with the last but one, , appropriately. You should save the address and length of either `pmsg` or `nmsg` into registers `ecx` and `edx` registers before jumping to label `print`.

```
        jne   npali           ; not a palindrome if different
        inc   esi             ; update address of next character from left
        dec   edi             ; address of next character from right
        cmp   esi, edi        ; proceed if the next character from left is
        jl    next            ;   before the next character from right
        mov   ecx, pmsg       ; palindrome; load the address and length
        mov   edx, plen       ;   of 'palindrome' message
        jmp   print           ; jump to print
npali:  mov   ecx, nmsg       ; not a palindrome; load address and length
        mov   edx, nlen       ;   of 'not palindrome' message
```

(d) [6%] Complete the required program by adding the necessary instructions to print the output message and exit.

```
; the following must be inserted at point A in (a) to print the input string
        mov   edx, eax        ; load length
        dec   edx             ; and skip the line feed
        mov   ecx, string     ; load address of input string
        mov   eax, 4          ; and print it
        mov   ebx, 1          ; on the standard output device
        int   0x80
; end insert

print:  mov   eax, 4          ; print the message
        mov   ebx, 1          ; on the standard output device
        int   0x80
        mov   eax, 0          ; exit
        int   0x80
```

****** END OF EXAMINATION ******