1. Define or describe the following:

a. Assembly language
A language for which each instruction is a mnemonic for a corresponding machine instruction, with the exception of a few “pseudo-instructions” which translate into a very few machine instructions. Compilers translate high-level languages into assembly code, which an assembler then translates into machine code.

b. Instruction Set Architecture
The specification of the binary machine instructions that a machine can execute, including the exact layout in the bit pattern for an instruction of the opcodes, the addressing, and any registers used. The ISA provides the abstraction on which higher-level languages are built and for which different implementations of the machine can be built.

c. Benchmark Suite
A collection of programs that are used to test the performance of computers. The SPEC benchmark is one example of a benchmark suite developed by a consortium of manufacturers and users.

d. Register indirect (with base) addressing
An address (base) is accessed from a register and is added to an immediate part (offset) to give the address where the desired data item is stored. This is used in MIPS in load and store instructions as with
   `ls $r1, val($r2)`
where `$r2` is the register with the base address that is added to the immediate value `val`, the data then being stored in register `$r1`. 
e. Register
A storage unit for storing a string of bits, typically one “word,” in a computer. Usually comprised of flip-flops.

f. Hertz
One cycle per second.
This is the unit, usually in mega (millions) or giga (billions) that computer clock rate is measured.

g. Edge-triggered
A storage device such as a register or memory unit is edge-triggered if the state (bit) stored can change only on a clock edge, that is when the clock is changing level from high to low (falling edge-triggered) or low to high (rising edge triggered).

h. DRAM
Dynamic Random Access Memory. A bit is stored as a voltage in a capacitor controlled by a transistor. Since capacitors leak voltage over time, the bits stored need to be refreshed periodically (every few nanoseconds) by reading and rewriting, hence the term dynamic. This makes dynamic RAM slower than the more expensive static RAM (SRAM).
2. Suppose you have the following instruction set mix:

<table>
<thead>
<tr>
<th>Instr type</th>
<th>CPI</th>
<th>% in program</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>25</td>
</tr>
</tbody>
</table>

a. What is the average CPI for this mix?

\[
CPI = (1 \times 0.5) + (2 \times 0.25) + (4 \times 0.25) = 2.0
\]

b. If the clock rate is 800 MHz, what is the MIPS for this machine/instruction mix?

\[
MIPS = \frac{800 \text{ MHz}}{2.0 \text{ CPI}} = \frac{800 \times 10^6 \text{ cycles/second}}{2 \text{ cycles/instruction}} = 400 \times 10^6 \text{ instructions/second} = 400 \text{ MIPS}
\]

c. Why is MIPS not by itself a good basis for determining the performance of a given machine?

MIPS depends on the instruction mix used in testing -- a program with many type A instructions above will have a higher MIPS -- so it may be compiler dependent. MIPS also depend on the architecture. An architecture with simpler instructions which are fast but do not do as much as instructions on a machine with more complex instructions may have a higher MIPS for a program that runs in the same or slower speed.

3. a. Write MIPS assembly code that will execute the following C statement. Assume that the following registers are used to represent the variables.

\[
a = (a + b) - (c - 33)
\]

\[
\text{add } \$t0, \$s0, \$s1\\
\text{addi } \$t1, \$s2, -33 \quad \# \text{ note: there is no subi in MIPS}\\
\text{sub } \$s0, \$t0, \$t1 \quad \# \text{ note: } \$s1 \text{ and } \$s2 \text{ should not be used for intermediate results. They may be needed later in the code.}
\]
b. Write MIPS assembly code that will execute the following C statements. Assume that the following registers are used to represent the variables:

base address of list in register $s0
length of list (len) in register $s1
sum in register $s2
k in register $s3

```
sum = 0;
for (k = 0; k < len; k++)
    sum += list[k];
```

```
sub $s2, $s2, $s2  # sum = 0, could use add $s0, $zero,
sub $s3, $s3, $s3  # k = 0
for:        slt $t0, $s3, $s1  # k < len ?
            beq $t0, $zero, forend  # if not, end loop
            add $t1, $s3, $s3  # compute $t1 = 4*k (could also use sll)
            add $t1, $t1, $t1
            add $t1, $t1, $s0  # $t1 = addr(list[k])
            lw $t1, 0($t1)  # $t1 = list[k]
            add $s2, $s2, $t1  # sum += list[k]
            addi $s3, $s3, 1  # k++
            j for  # jump to beginning of loop
forend:
```
4. Consider the following function prototype for C++:

```c
int Combo(int a, int b);
```

Assume that the function `Combo` has six local variables stored in registers s0, s1, s2, s3, t0, and t1, with the usual conventions. Assume that the function `Combo` makes calls to other functions. Outline the steps needed to execute the following function call in assembly code, assuming the variables are assigned to registers, \( x \) in $s4, \( a \) in $s0, \( b \) in $s1.

```assembly
x += Combo(a, b);
```

You may answer by explaining the steps in words, or by writing assembly code with comments explaining the purpose of each statement, or a mix of the two. If you are unsure whether something needs to be done, put it in. Include all operations done by the caller function (the function where the above line of code occurs) and the callee function (the function `Combo`) that are needed to make the function call work. Indicate the body of the function `Combo` with a comment `<body of function>`.
In caller function
1. Place parameters in registers
   - add $a0, $s0, $zero
   - add $a1, $s1, $zero
2. Jump and link (putting return address into $ra and jumping to function)
   - jal Combo
8. Use returned value
   - add $s4, $v0, $s4

In function Combo
3.a. Adjust stack pointer
   - addi $sp, $sp, -20
   b. and save state
      (s registers used and ra register)
      - sw $s0, 16($sp)
      - sw $s1, 12($sp)
      - sw $s2, 8($sp)
      - sw $s3, 4($sp)
      - sw $ra, 0($sp)
4. Move parameters into locals
   - add $s0, $a0, $zero
   - add $s1, $a1, $zero
5. Execute function, placing return value in v register(s)
   <body of function>
   - add $v0, xxx
6.b. Restore state
   (s registers used and ra register)
   - lw $s0, 16($sp)
   - lw $s1, 12($sp)
   - lw $s2, 8($sp)
   - lw $s3, 4($sp)
   - lw $ra, 0($sp)
   a. Readjust stack pointer
   - addi $sp, $sp, 20
7. Return to caller
   - jr $ra
5. Draw the transistor circuit for a 3-way **AND** gate.
6. Draw the logic gate circuit for a decoder with 3 input lines and the appropriate number of output lines. Where might such a circuit be used?
7. Consider the following diagram.

![Diagram of a circuit with a clock input, data input, and two latches labeled Q and ~Q.]

a. The two parts of the circuit that are boxed are identical. Explain what one of these circuits by itself is and how it works.

*Each of these circuits is a D-latch. When the clock is high, whatever value is on the Data In line is stored and is also on the data out line. When the Clock is low, the stored value remains captured on the Out (Q) line, regardless of any changes on the Data In line. The ~Q line has the opposite of the Q line.*

b. The complete circuit forms a standard unit. Describe what it is, how it works, and where it might be used in a CPU.

*The whole Circuit is a flip-flop, which also stores a single bit value. However, in a flip-flop, the value on the Data-In line is stored during the falling edge of the clock. While the clock is high, the first (master) latch is open and captures the data value. When the clock falls through the falling edge to low, the second (slave) latch (using the negated clock) is open, capturing the signal on the first latch. If the data In signal changes while the clock is low, it will not affect the stored value because the master latch is closed and will not change. If the data value changes while the clock is high, it will change the master latch, but not the slave. It is only the value on the end of the high signal, kept on while the clock falls to low that is captured.*
8. Consider the hex number 0x33104000.
   a. Write this as a binary number

   0011 0011 0001 0000 0100 0000 0000 0000

   b. Write this as an integer base ten, assuming two's complement representation. (Write the value as a sum of powers of two, with the appropriate sign, e.g. \(2^27 + 2^22 + \ldots + 2^5\))

   \[+ 2^{29} + 2^{28} + 2^{25} + 2^{24} + 2^{20} + 2^{14}\]

   c. Write this number as a decimal, assuming it is in IEEE-754 floating point representation for 32-bit floats.

   *Sign is +*

   *Exp is 01100110 in bias 127 this is (2+4+32+64)-127 = -25*

   *Fraction is 0010000010…, so significand is 1.001000001 = 1 + 1/8 + 1/512 = 577/512*

   *Final is +577 \times 2^{-34}*

   d. Write the instruction that this represents. (Hint: The opcode 001100 is for the instruction andi)

   *andi $24$, $16$, 0x4000 is sufficient. This is also

   *andi $t8$, $s0$, 2^{15})*