Q1

Better understanding of architecture issues

Improve algorithm development skills

Personal satisfaction that comes with learning something complex.

Understanding of computer systems

Assembler – program that translates assembly language code to machine lang code e.g.
NASM - Netwide
MASM - Microsoft

TASM – Borland Turbo

ROM non-volatile – you cannot erase or modify it when the computer system is turned off

Q2

Unconditional branching – transfers control to the specified target

Absolute address and pc relative address – we can move code from one block of memory to another without changing target address

Use of Procedural calls
Facilitates modular programming – we have return control to the calling program after execution of procedures

Requires 2 pieces of information i.e., End of procedure, Return address

1) Advantages of HLL
   Program development is faster – due to structures (iterative, sequential, selection)
2) Easy to maintain -
3) Portable – can work on any computer

Macro - are used as a shorthand notation for a group of statements

permit the assembly language programmer to name a group of statements and refer to the group by the macro name

Stack – A group of memory locations in read and write memory of any computer and is used to store contents of register operand and memory address.

3 uses of stack

1) as a scratchpad to temporarily store data,
2) for transfer of program control,
3) and for passing parameters during a procedure call.

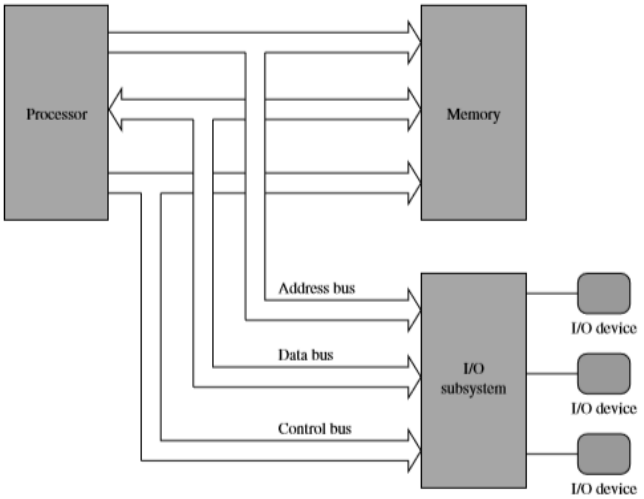

Figure 2.2 Simplified block diagram of a computer system.

b) Master – initiator of a transaction

Slave – target of the transaction

Three-address machines

- Instructions carry all three addresses explicitly
- One address used for destination & 2 addresses for source

Zero-address machines

- Location of both operands are assumed to be at a default location
- Use stack as source of input operands & result goes back into the stack

Q4

Process registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. The registers can be divided into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set. The number of addresses used in instructions partly influences the number of data registers and their use. For example, in three- and two-address machines, there is no need for the internal data registers. However, having a few internal registers improves performance by

cutting down the number of memory accesses required to execute a program. RISC processors typically have a large number of registers.

Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.

Flow of control

Program execution, by default, proceeds sequentially. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC. When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in the MIPS processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern. However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as if-then-else and while statements to alter the execution behavior based on some run-time conditions.

Q5

Immediate addressing - data are specified as part of the instruction itself. As a result, even though the data are in memory, it is located in the code segment, not in the data segment. This addressing mode is typically used in instructions that require at least two data items to manipulate.

Direct Addressing Mode

Operands specified in a memory-addressing mode require access to the main memory, usually to the data segment. As a result, they tend to be slower than either of the two previous addressing modes.

Indirect Addressing - The direct addressing mode can be used in a straightforward way but is limited to accessing simple variables. For example, it is not useful in accessing the second element of table1 as in the following C statement:

table1[1] = 99

The indirect addressing mode remedies this deficiency. In this addressing mode, the offset or effective address of the data is in one of the general registers. For this reason, this addressing mode is sometimes referred to as the register indirect addressing mode.

Q6

INC - inc (INCrement) instruction adds one to its operand

The operand can be either in a register or in memory

The general format of these instructions is
inc destination

XLAT

The xlat (translate) instruction can be used to perform character translation.
xlatb
To use this instruction, the EBX register must to be loaded with the starting address of the translation table and AL must contain an index value into the table. The xlat instruction adds contents of AL to EBX and reads the byte at the resulting address.

ADD

The add instruction can be used to add two 8-, 16- or 32-bit operands. The syntax is
add destination, source
The semantics of the add instruction are
destination = destination + source

XCHG

The xchg instruction exchanges 8-, 16-, or 32-bit source and destination operands. The syntax is similar to that of the mov instruction. Some examples are
xchg EAX,EDX
xchg [response],CL
xchg [total],DX

Q7