1.

	1	2	3	4	5	6
Instruction 1	FI	DI	EI	WO		
Instruction 2		FI	DI	EI	WO	
Instruction 3			FI	DI	EI	WO

For Three instructions each instruction having 4 stages, we would require 6 units of time to complete the 3 instructions. Here we assume that each stage in a instruction takes a unit time and also we assume each stage can run in parallel.

2.

An x86 instruction can have 6 stages in the instruction processing. They are

1. Fetch instruction (FI): In this step we will fetch the instruction and store it in the buffer.
2. Decode instruction (DI): We will decode the instruction and determine the opcode and operand.
3. Calculate operands (CO): We will calculate the effective address of each source operand.
4. Fetch operands (FO): In this stage we will fetch the operand from the memory. Operands from the registers need not be fetched
5. Execute instruction (EI): Here we will execute the specified instruction.
6. Write operand (WO): Store the results in the memory.

Let us consider the instruction SUB [ESI], EAX. In this instruction we are actually taking two operands from the [ESI] and EAX, subtracting both the operands and storing the results in [ESI]. The various stages of execution is as follows.

FI	DI	CO	FO	EI	WO
yes	yes	no	yes	yes	yes

Fetch Instruction: In the fetch instruction stage we are fetching the instruction SUB [ESI], EAX. The PC contains the address of the instruction to be fetched. The instruction is read to the MBR and then moved to the IR.

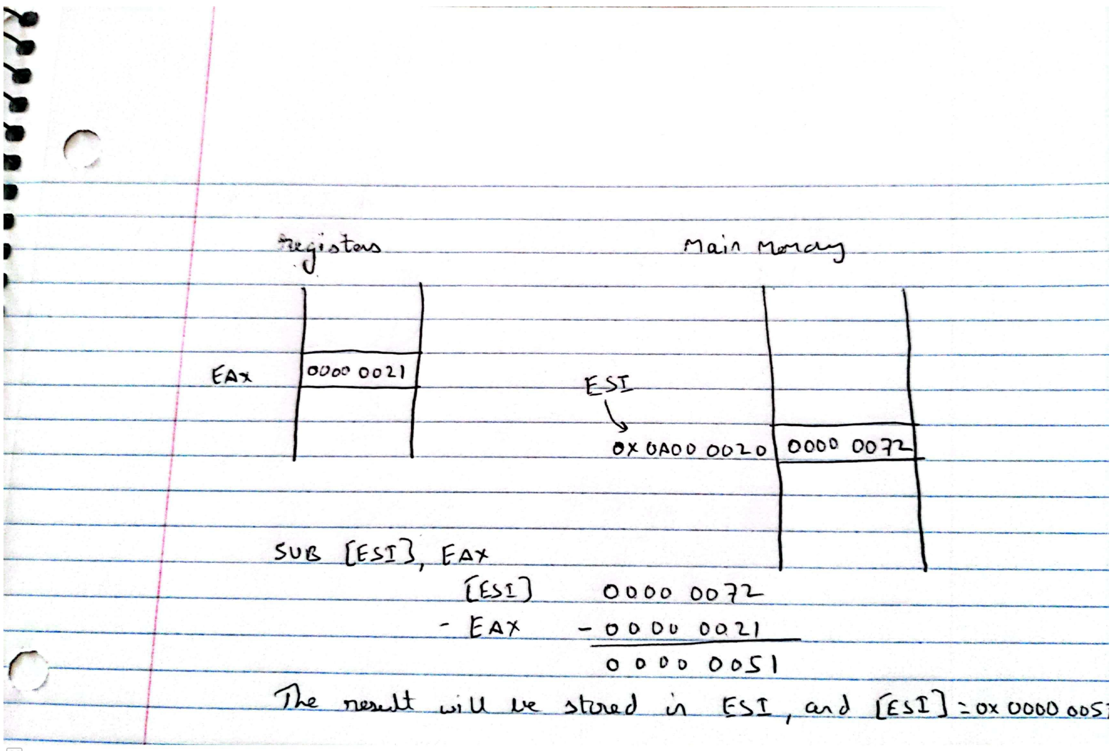
Decode Instruction: The instruction which is available is decoded to get the opcode and operand. In the instruction SUB [ESI], EAX, SUB is referred the opcode and operand are present in the location [ESI] and EAX.

Calculate Operands: This stage is not present for this instruction, because we are not calculating any effective address.

Fetch Operand: We need to fetch the operand from the location [ESI]. Let us assume that ESI = 0x 0A00 0020, we will fetch the data which is present at the location 0x 0A00 0020. Let us assume that the data present at the location is [ESI] = 0x 0000 0072, data is two words long. We need not fetch the operand from EAX, it will be available since it's a register. Let us say EAX = 0x 0000 0021.

Execute Instruction: In this stage we will be performing the subtraction operation of the two operands. The subtraction will be 0x 0000 0072 - 0x 0000 0021, which will result in 0x 0000 0051.

Write Operation: The result of the subtraction of the two operands will be stored back to the location ESI. Now we will write 0x 0000 0051 to the location of ESI and we will have [ESI] = 0x 0000 0051.



Let us consider another instruction
  ADD EAX, [ESI]

In this instruction we are actually taking two operands from the EAX and [ESI], adding both the operands and result will be present in EAX. The various stages of execution is as follows.

FI	DI	CO	FO	EI	WO
yes	yes	no	yes	yes	no

Fetch Instruction: In the fetch instruction stage we are fetching the instruction ADD EAX, [ESI]. The PC contains the address of the instruction to be fetched. The instruction is read to the MBR and then moved to the IR.

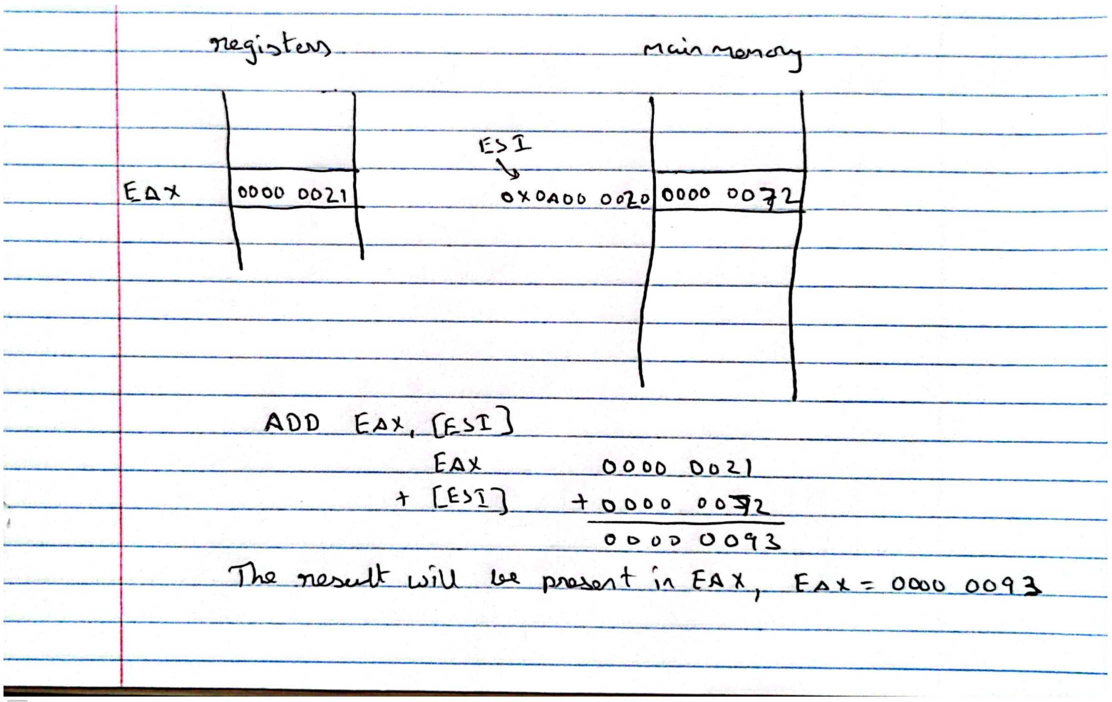
Decode Instruction: The instruction which is available is decoded to get the opcode and operand. In the instruction ADD EAX, [ESI]; ADD is referred the opcode and operand are present in the location EAX and [ESI].

Calculate Operands: This stage is not present for this instruction, because we are not calculating any effective address.

Fetch Operand: We need to fetch the operand from the location [ESI]. Let us assume that ESI = 0x 0A00 0020, we will fetch the data which is present at the location 0x 0A00 0020. Let us assume that the data present at the location is [ESI] = 0x 0000 0072, data is two words long. We need not fetch the operand from EAX, it will be available since it's a register. Let us say EAX = 0x 0000 0021.

Execute Instruction: In this stage we will be performing the addition operation of the two operands. The addition will be 0x 0000 0021 + 0x 0000 0072 , which will result in 0x 0000 0093. The result will be present in EAX. Now EAX = 0x 0000 0093.

Write Operation: There is no write operation for this instruction since the result of addition will be present in the register. We need write operation to store data in memory, it's not necessary in this instruction.



ADD EAX, [ESI]

```
     EAX        0000 0021
   + [ESI]    + 0000 0072
              -----------
                0000 0093
```

The result will be present in EAX, EAX = 0000 0093

3.

Pipeline suffers from the penalty of a branch instruction. When a branch instruction occur the processor is not sure which instruction to be fetched in the pipeline until the execution of the branch instruction, it may even make a wrong choice. A brute force approach to deal with this problem would be to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams.
There are two problems with this approach

- With multiple pipelines there are contention delays for access to the registers and to memory, because both streams will try to access the registers and memory at the same time.
- One of the two additional branch instructions may enter the pipeline before the original branch decision is resolved. In that case each such instruction needs an additional stream.

One of the drawback could be that, we are wasting our efforts by processing the two streams, as we are going to use only one stream at the end.
Despite these two drawbacks this can improve system performance.

4.
The outcome of a conditional branch cannot be predicted before, it is an unpredictable event. We need to know which instruction is going to be executed next to fetch the instruction. In case of conditional branch since we cannot predict which instruction is going to be executed next we need to wait until the execution of a conditional branch instruction before fetching the instruction and this slows down the instruction pipeline. Some ways to deal with conditional branching is as follows.

Multiple Streams: In this approach we will follow a brute force method where we will replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. In this case we will have two streams for the two instructions. There are few potential challenges with this approach.
1. With multiple pipelines there are contention delays for access to the registers and to memory
2. One of the two additional branch instructions may enter the pipeline before the original branch decision is resolved. In that case each such instruction needs an additional stream.
Despite these drawbacks this can improve system performance.

Prefetch buffer target: When a conditional branch is recognized, the target of the branch is prefetched, and also the instruction following the branch. Until the branch instruction is executed the target will be saved, if the branch is taken the saved target is used.

Loop Buffer: A loop buffer is maintained by the instruction fetch stage of the pipeline and contains the *n* most recently fetched instructions. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. Benefits of loop buffer are
1. Instructions fetched in sequence will be available without the usual memory access time.
2. Useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences.
3. This strategy is particularly well suited to dealing with loops, or iterations. Instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

Branch Prediction: Predict whether a branch will be taken or not. The common techniques are:
- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.The first two approaches either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target.The third approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others.Taken or not taken approach stores the history bits whether the branch is taken or not.The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements, the address of a branch instruction, some number of history bits that record the state of use of that instruction, and information about the target instruction.

Delayed branch: It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

5.
History bits are used in Dynamic branch strategies to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. One or more bits can be associated with

each conditional branch instruction to represent the history of the instruction. These bits are referred to as a taken/not taken switch that helps the processor to make a decision when the instruction is encountered again. The history bits are not associated with the instruction in main memory, they are kept in temporary high speed storage. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. A single bit can be used to record whether a branch is taken or not taken last time. Using a single bit has short coming in case of a loop instruction, an error in prediction will occur twice for each use of the loop, once on entering the loop, and once on exiting. If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. The use of history bits has one drawback, if the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded.

6.
A loop buffer is a small and high speed memory maintained by the instruction fetch stage of the pipeline and contains the *n* most recently fetched instructions. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.
Benefits of loop buffer are
1. By prefetching, loop buffer will contain instructions sequentially ahead of the current instruction fetch address. Instructions fetched in sequence will be available without the usual memory access time.
2. If a branch points to a target whose location is ahead of the address of the branch instruction, the target will be available in the buffer. This is useful for common occurrence like if and if-else sequences.
3. This strategy is particularly well suited to dealing with loops. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched only once from the memory, and is available in buffer for all next iterations.

The loop buffer is similar in principle to a cache dedicated to instructions, but loop buffer only retains instructions in sequence and is much smaller in size.

Loop Buffer is maintained by the instruction fetch stage of the pipeline.

7.
Let there be k stages and n instructions which needed to be executed.

Let us assume that each stage will take a time τ_m +d , where τ_m is the maximum stage delay.

Let τ be the time taken by each stage $\tau = \tau_m$ +d

In real life scenario τ_m >> d so we can ignore d and $\tau = \tau_m$

Let *Tk*, *n* be the total time required for a pipeline with *k* stages to execute *n* instructions. Then

$T_{k,n}$ = [*k* + (*n* - 1)]τ

Since the first instruction will require k stages and the rest n-1 instructions need n-1 stages extra. So the total time will be $T_{k,n}$ = [*k* + (*n* - 1)]τ

Now let us consider a processor which does not have pipeline architecture, in that case the total time taken will be $T_{1,n}$ = knτ

Speedup = $S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{\mathrm{kn\tau}}{[\mathrm{k + (n - 1)}]\tau}$

$$\boldsymbol{S_k} = \frac{\mathbf{kn}}{[\boldsymbol{k} + (\boldsymbol{n} - \mathbf{1})]}$$

For the case where n -> ∞

$S_k = \frac{\mathrm{kn}}{[k + (n-1)]}$, taking n common and canceling it out we have,

$$S_k = \frac{\mathrm{k}}{[\frac{\mathrm{k}}{\mathrm{n}} + (1 - \frac{1}{\mathrm{n}})]}$$

$$\lim_{n -> \infty} S_k = \lim_{n -> \infty} \frac{\mathrm{k}}{[\frac{\mathrm{k}}{\mathrm{n}} + (1 - \frac{1}{\mathrm{n}})]}$$

We know that $\lim_{x -> \infty} \frac{1}{x} = 0$, using this we have

$$\lim_{n -> \infty} S_k = \frac{\mathrm{k}}{[0 + (1 - 0)]}$$

$$\lim_{\boldsymbol{n} -> \infty} \boldsymbol{S_k} = \mathbf{k}$$

As n tends to ∞ speed up will be k.