

Q1)

Part (i)

differentiate between pipelined and non-pipelined processors. What is advantage of pipelined over non-pipelines

Pipelining and non-pipelining are two different approaches to designing processors. Let's differentiate between the two:

Non-Pipelined Processors:

1. In a non-pipelined processor, each instruction is executed in a sequential manner. The processor fetches an instruction from memory, decodes it, executes it, and then moves on to the next instruction.
2. The entire instruction must complete its execution before the next instruction can be fetched and executed. This leads to a significant amount of idle time for the processor, as it waits for the completion of one instruction before moving on to the next.
3. Non-pipelined processors have a simpler design and are easier to implement compared to pipelined processors.
4. However, the lack of parallelism and idle time between instructions results in lower overall performance and throughput.

Pipelined Processors:

1. In a pipelined processor, the execution of instructions is divided into smaller stages, and multiple instructions are executed simultaneously in different stages of the pipeline.
2. Each stage of the pipeline performs a specific task, such as instruction fetch, instruction decode, execution, memory access, and write-back. As one instruction moves to the next stage, the subsequent instruction can enter the pipeline.
3. Pipelining allows for overlapping of instruction execution, which improves the throughput and performance of the processor.
4. By breaking down the instruction execution into smaller stages, pipelined processors can achieve higher clock frequencies, as each stage requires less time to complete.
5. Pipelining increases instruction throughput, reduces idle time, and takes advantage of instruction-level parallelism.
6. However, pipelining introduces additional challenges, such as data hazards, control hazards, and pipeline stalls, which need to be handled through techniques like forwarding, branch prediction, and instruction scheduling.

Advantages of Pipelined Processors over Non-Pipelined Processors:

1. **Increased Performance:** Pipelining allows for the execution of multiple instructions simultaneously, resulting in higher throughput and improved performance.
2. **Reduced Idle Time:** Pipelined processors minimize idle time by overlapping the execution of multiple instructions, thereby keeping the processor busy.
3. **Higher Clock Frequencies:** Pipelining enables shorter pipeline stages, which allows for higher clock frequencies and faster overall execution.

4. Efficient Resource Utilization: Pipelining allows for better utilization of processor resources by parallelizing instruction execution.

5. Support for Instruction-Level Parallelism: Pipelining takes advantage of instruction-level parallelism by executing multiple instructions simultaneously.

Overall, the advantage of pipelined processors over non-pipelined processors lies in their ability to achieve higher performance and throughput by exploiting parallelism and reducing idle time. However, pipelining introduces additional complexities in terms of handling hazards and stalls, which need to be addressed in the processor design.

Pipelined	Non-Pipeline
1) Overlapping execution of instructions	1) Sequential execution of instr.s.
2) Instruction pipelining - processor fetches next instr. while simultaneously decoding & executing current instr.	2) Fetch-Decode-Execute cycle - instr fetch from memory, decode to determine operation, executes & then fetches next instr.
3) Allow improved resource utilization & increased throughput.	3) During instr execution, processor resources like ALU & registers dedicated solely to that instr. When completes, then available for next execution.
4) Parallelism...	4) Limited Parallelism $\frac{P}{NP}$

Part 2

consider a pipeline having 4 phases with duration 60,50,90 and 80ns. give latch delay in 10ns. Calculate (a)non-pipeline execution time (b)speed up ratio, (c)pipeline time for 1000 tasks (d)sequential time for 1000 tasks (e)throughput

To calculate the required values, we'll use the given information about the pipeline phases and latch delay.

Pipeline Phases and Durations:

Phase 1: 60 ns

Phase 2: 50 ns

Phase 3: 90 ns

Phase 4: 80 ns

Latch Delay: 10 ns

Number of Tasks: 1000

(a) Non-pipeline Execution Time:

In a non-pipeline execution, each task has to go through all four phases without any overlap or parallelism. Therefore, the non-pipeline execution time for one task would be the sum of the durations of all four phases:

$$\text{Non-pipeline execution time} = 60 \text{ ns} + 50 \text{ ns} + 90 \text{ ns} + 80 \text{ ns} = 280 \text{ ns}$$

(b) Speed Up Ratio:

The speed-up ratio is the ratio of the non-pipeline execution time to the pipeline execution time. Since the non-pipeline execution time is given as 280 ns, we need to calculate the pipeline execution time first.

Pipeline Execution Time:

In a pipelined execution, the execution time is determined by the longest pipeline phase, which is Phase 3 with a duration of 90 ns.

$$\text{Pipeline execution time} = \text{Duration of longest phase} + \text{Latch delay}$$

$$\text{Pipeline execution time} = 90 \text{ ns} + 10 \text{ ns} = 100 \text{ ns}$$

$$\text{Speed-up ratio} = \text{Non-pipeline execution time} / \text{Pipeline execution time}$$

$$\text{Speed-up ratio} = 280 \text{ ns} / 100 \text{ ns} = 2.8$$

(c) Pipeline Time for 1000 Tasks:

Since the pipeline can execute multiple tasks simultaneously, we need to calculate the pipeline time based on the duration of the longest phase.

$$\text{Pipeline time for 1000 tasks} = (\text{Number of tasks} - 1) * \text{Duration of longest phase} + \text{Latch delay}$$

$$\text{Pipeline time for 1000 tasks} = (1000 - 1) * 90 \text{ ns} + 10 \text{ ns}$$

$$\text{Pipeline time for 1000 tasks} = 899 * 90 \text{ ns} + 10 \text{ ns}$$

$$\text{Pipeline time for 1000 tasks} = 80910 \text{ ns} + 10 \text{ ns} = 80920 \text{ ns}$$

(d) Sequential Time for 1000 Tasks:

Sequential time is the time it would take to execute all the tasks sequentially without any parallelism. For 1000 tasks, it would be the product of the non-pipeline execution time and the number of tasks.

$$\text{Sequential time for 1000 tasks} = \text{Non-pipeline execution time} * \text{Number of tasks}$$

$$\text{Sequential time for 1000 tasks} = 280 \text{ ns} * 1000 = 280000 \text{ ns}$$

(e) Throughput:

Throughput is the number of tasks completed per unit of time. In this case, it would be the number of tasks divided by the pipeline time for 1000 tasks.

$$\text{Throughput} = \text{Number of tasks} / \text{Pipeline time for 1000 tasks}$$

$$\text{Throughput} = 1000 / 80920 \text{ ns}$$

Note: The throughput can also be calculated in units such as tasks per second by converting the pipeline time to seconds if required.

These calculations provide the values for the given pipeline configuration and tasks.

Q1

Part (2)

(a) Non-pipeline exec. time :-

$$60 + 50 + 90 + 80 = 280 \text{ ns}$$

(b) Speedup ratio :-

First $\rightarrow$ Pipeline exec. time

= longest phase + latch delay

$$= 90 \text{ ns} + 10 \text{ ns}$$

$$= 100 \text{ ns}$$

$$\text{Speedup ratio} = \frac{\text{Non-P}}{P} = \frac{280 \text{ ns}}{100 \text{ ns}} = 2.8$$

$$\begin{aligned} \text{(c) Pipeline for 1000 tasks} &= (\text{tasks} - 1) * \text{Pipeline time} + \text{latch delay} \\ &= (1000 - 1) * 90 + 10 \\ &= 89920 \text{ ns} \end{aligned}$$

$$\begin{aligned} \text{(d) Seq. (Non-Pipeline) 1000 tasks} &= \text{NP time} * \text{no. of tasks} \\ &= 280 * 1000 \\ &= 280000 \text{ ns} \end{aligned}$$

$$\begin{aligned} \text{(e) Throughput} &= \frac{\text{No. of tasks}}{\text{Pipeline time}} \\ &= \frac{1000}{89920 \text{ ns}} = 0.011 \text{ ns} \end{aligned}$$

OR

$$= \frac{1000 \times 10^9}{89920 \text{ s}} = 1.1 \times 10^{-5} \text{ s}$$

Part (i)

demonstrate the memory hierarchy of computer and define the concept of locality of reference in memory

Registers

Storage locations on processor

- Cache

High speed memory where data from frequently used main memory locations is temporarily stored

Connected to larger

- Main memory

medium speed memory complemented by very large

- Secondary memory

hard disk drives containing data not directly accessible by CPU

Hard disk drives can be magnetic or solid state (flash memory)

Forms of locality

- Three forms

- **Temporal locality**—Recently accessed items tend to be accessed again in the near future.

- **Spatial locality**—Accesses tend to be clustered in the address space (for example, as in arrays or loops).

- **Sequential locality**—Instructions tend to be accessed sequentially.

Help us exploiting majority of memory accesses from small and fast memory

OR

Memory Hierarchy: The memory hierarchy of a computer consists of different levels of memory, including *CPU registers, cache memory, main memory (RAM), secondary storage, and tertiary storage*. Each level offers varying levels of performance and capacity, with the higher levels being faster but smaller and more expensive, while the lower levels provide larger capacity but slower access speeds.

Locality of Reference: Locality of reference refers to the tendency of a program to access memory locations that are close to each other in time. It is based on two primary types of locality:

1. Temporal Locality: Temporal locality refers to the reuse of recently accessed memory locations over a short period of time. Programs often access the same data or instructions repeatedly, benefiting from the retention of this data in faster levels of memory like caches.

2. Spatial Locality: Spatial locality refers to the access of memory locations that are near each other in physical address space. When a program accesses a memory location, it is likely to access nearby memory locations in the near future. This occurs due to factors like sequential execution, array traversals, or contiguous data structures.

The concept of locality of reference is crucial in memory systems as it allows for the optimization of memory access patterns. Caches exploit temporal and spatial locality by storing recently accessed data and nearby data, reducing the time required to fetch them from slower levels of memory. This improves the overall efficiency and performance of the memory hierarchy.

Part (ii)

consider a system having 32k bytes of main memory and 512 bytes of cache memory. the block size for both memories is 16 bytes. How many bits will be kept for offset, set number and tag fields for 8-way set associative cache

Part (ii)

Given:	Main memory = 32k bytes
	Cache memory = 512 bytes
	Block size = 16 bytes

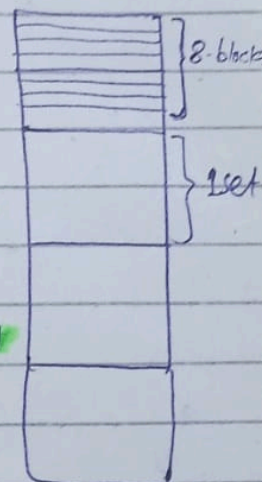
Soln.

$$\begin{aligned} \# \text{ of cache blocks} &= \frac{\text{cache memory size}}{\text{Block size}} \\ &= \frac{512}{16} = 32 \text{ cache blocks} \end{aligned}$$

⇒ As 8-way Set Associate

↳ each set will contain 8-cache blocks

$$\begin{aligned} \Rightarrow \# \text{ of cache sets} &= \frac{\# \text{ of cache blocks}}{\text{Associativity}} \\ &= \frac{32 \text{ blocks}}{8} \\ &= 4 \text{ sets with } 8\text{-blocks} \end{aligned}$$



① offset bits:-

$$\begin{aligned} \log_2 (\text{blocksize}) \\ &= \log_2 (16) = 4 \text{ bits to represent offset} \end{aligned}$$

② Set number bits:-

As in our 8-way set associativity, cache has 4 sets

$$\text{So, } \log_2 (4) = 2 \text{ bits to represent set \#}$$

③ Tag bits:-

$$\begin{aligned} &= \# \text{ of bits for addressing} - (\text{offset bits} + \text{set \# bits}) \\ &= \log_2 (32K) - (4 + 2) \\ &= 15 - 6 = 9 \text{ bits} \end{aligned}$$

$$\begin{aligned} 2^{15} &= 32,768 \\ 15 &= \log_2 (32,768) \end{aligned}$$

Assume that a processor has a direct mapped cache, the physical address is 20 bits long, the tag is 12 bits, and each block holds 32 bytes of data. How many lines are in this cache? Let's say CPU send requests for the following sequence of addresses: 0011 0101 0110 1101 1010, 0000 0011 1101 0111 0001, 0011 0101 0110 1100 0000, 1000 0000 1101 0111 0111, 1111 0010 0000 0000 1000, 1111 0010 0000 0001 0110, 1010 1010 1010 1100 0001, 1000 0000 1101 0111 0111. For each of these requests, state if it would result in a cache hit or a cache miss. Also show the final state of tag registers once all requests have been entertained. Assume that the whole cache is free at the start.

Given information:

Physical address: 20 bits

Tag: 12 bits

Block size: 32 bytes

To calculate the number of lines in the cache for a direct-mapped cache, we need to determine the number of blocks.

Number of lines = Number of blocks

Block size = 32 bytes

Physical address = 20 bits

Number of blocks = $2^{(\text{Physical address} - \log_2(\text{Block size}))}$

Number of blocks = $2^{(20 - \log_2(32))}$

Number of blocks = $2^{(20 - 5)} = 2^{15} = 32,768$

Therefore, the cache has 32,768 lines.

Q2 Part (iii)

Direct-Mapped Cache

Given

Physical address = 20 bits

Tag = 12 bits

Block size = 32 bytes

No. of lines ??

⇒ # of lines = # of blocks

of blocks = $2^{(\text{Physical address} - \log_2(\text{block size}))}$

$$= 2^{(20 - \log_2(32))}$$

$$= 2^{(20 - 5)}$$

$$= 2^{15}$$

$$= 32,768$$

Hence cache has 32768 lines

Now, let's analyze the CPU requests and determine if they result in a cache hit or a cache miss, and update the tag registers accordingly:

1. 0011 0101 0110 1101 1010 (Address 0x356DA)

Cache Miss (No data in the cache)

2. 0000 0011 1101 0111 0001 (Address 0x03D71)

Cache Miss (No data in the cache)

3. 0011 0101 0110 1100 0000 (Address 0x356C0)

Cache Miss (No data in the cache)

4. 1000 0000 1101 0111 0111 (Address 0x80D77)

Cache Miss (No data in the cache)

5. 1111 0010 0000 0000 1000 (Address 0xF2008)

Cache Miss (No data in the cache)

6. 1111 0010 0000 0001 0110 (Address 0xF2016)

Cache Miss (No data in the cache)

7. 1010 1010 1010 1100 0001 (Address 0xAAAA1)

Cache Miss (No data in the cache)

8. 1000 0000 1101 0111 0111 (Address 0x80D77)

Cache Miss (No data in the cache)

Final State of Tag Registers: No data in the cache, so all tag registers remain empty.

All CPU requests result in cache misses since the cache is initially empty. The tag registers do not store any valid data at the end because no cache hits occurred.

Q3

Part (i)

while defining CPU performance, which parameters are required to consider? Differentiate between CPU performance and execution time. Discuss factors affecting execution time.

When defining CPU performance, several parameters are considered. These parameters help evaluate the effectiveness and efficiency of a CPU in executing tasks. The key parameters include:

1. Clock Rate: It refers to the speed at which the CPU operates and is measured in cycles per second (Hertz). A higher clock rate generally indicates faster performance, although other factors can influence overall performance.

2. Instructions Per Cycle (IPC): IPC represents the average number of instructions that can be executed in a single clock cycle. CPUs with higher IPC can perform more work per cycle and often offer better performance.

3. Cache Size and Latency: Cache memory plays a crucial role in CPU performance. A larger cache size and lower cache latency (access time) can reduce the time required to fetch data and instructions from memory, improving overall performance.

4. Pipelining: Pipelining allows for the concurrent execution of multiple instructions, increasing instruction throughput and overall performance. A deeper pipeline can potentially achieve higher performance, although it may introduce additional latency due to pipeline stalls.

5. Number of Cores: Multi-core CPUs have multiple processing units, enabling simultaneous execution of multiple tasks. More cores generally lead to better parallelism and higher performance in multi-threaded applications.

6. Instruction Set Architecture (ISA): The design and efficiency of the CPU's instruction set architecture can significantly impact performance. Optimized instruction sets, such as those supporting vector processing or specialized operations, can improve performance in specific tasks.

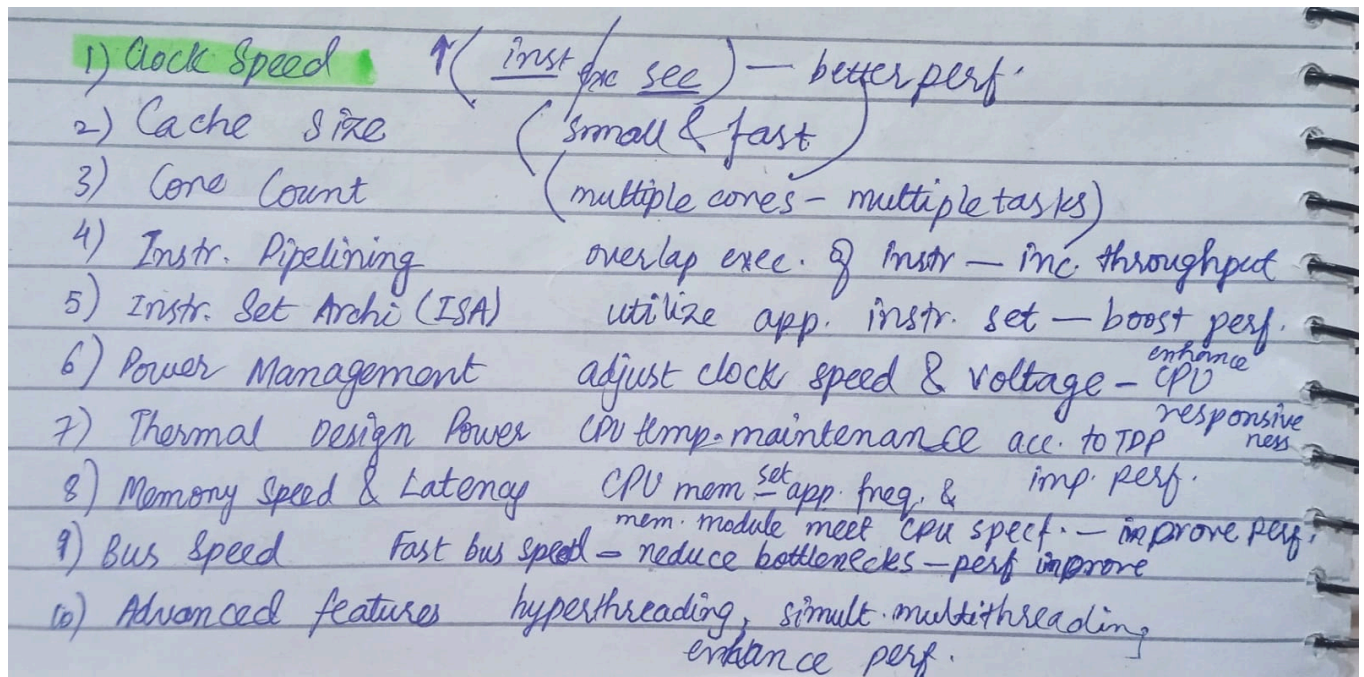
CPU performance refers to the overall capability and efficiency of a CPU in executing tasks. It is a measure of how effectively a CPU can process instructions and complete computational tasks.

On the other hand, execution time refers to the time taken by a CPU to complete a specific task or program. It is the actual time required for the CPU to execute a set of instructions or perform a specific computation.

Factors Affecting Execution Time:

- 1. Clock Rate:** A higher clock rate generally leads to shorter execution times. CPUs with faster clock speeds can execute instructions more quickly.
- 2. Instruction Count:** The total number of instructions in a program directly impacts execution time. A larger instruction count will generally result in longer execution times.
- 3. CPI (Cycles Per Instruction):** CPI represents the average number of clock cycles required to execute a single instruction. A lower CPI value indicates faster execution times.
- 4. Memory Access Time:** Memory access times can significantly impact execution time. Accessing data from main memory or cache can introduce latency, causing stalls in instruction execution.
- 5. Dependencies and Branches:** Dependencies between instructions and frequent branching in the program flow can lead to pipeline stalls and longer execution times.
- 6. Parallelism:** Utilizing parallel execution, such as through multi-threading or SIMD (Single Instruction, Multiple Data), can reduce execution time by executing multiple instructions simultaneously.
- 7. Data Dependencies:** Dependencies between instructions, especially those requiring data from previous instructions, can introduce delays and increase execution time.
- 8. Cache Performance:** Cache hits and misses can affect execution time. Higher cache hit rates result in faster access to data, reducing the time spent waiting for data to be fetched from slower memory levels.

Overall, execution time is affected by a combination of factors such as clock rate, instruction count, CPI, memory access times, dependencies, parallelism, and cache performance. Improving these factors can lead to reduced execution time and improved CPU performance.



Part (ii)

You are on a design team for a new processors. The clock of processors runs at 200MHz. the following table gives instruction frequencies for a given program, as well as how many cycles instruction take, for the different classes of instructions. Instruction Type, Frequency, Cycles Load and Stores, 30%, 6 Arithmetic instructions, 50%, 4 All others, 20%, 3

(a) Calculate the average CPI. (

b) if given program gets executed in 11seconds, calculate the no. of instructions in a program.

(c) What clock rate would make the machine to execute the same program in 10 sec?

(a) To calculate the average CPI (Cycles Per Instruction), we need to consider the instruction frequencies and the cycles required for each instruction type.

Given information:

Clock rate: 200 MHz (cycles per second)

Instruction Type: Load and Stores, Arithmetic instructions, All others

Load and Stores:

Frequency = 30%

Cycles per Instruction = 6 cycles

Arithmetic instructions:

Frequency = 50%

Cycles per Instruction = 4 cycles

All others:

Frequency = 20%

Cycles per Instruction = 3 cycles

To calculate the average CPI, we can use the following formula:

Average CPI = (Frequency₁ * CPI₁ + Frequency₂ * CPI₂ + Frequency₃ * CPI₃) / 100

Substituting the values:

Average CPI = (30 * 6 + 50 * 4 + 20 * 3) / 100

Average CPI = (180 + 200 + 60) / 100

Average CPI = 440 / 100

Average CPI = 4.4

Therefore, the average CPI for the given program is 4.4.

(b) To calculate the number of instructions in the program, we can use the formula:

Number of instructions = Execution time / (Clock rate * Average CPI)

Given:

Execution time = 11 seconds

Clock rate = 200 MHz

Average CPI = 4.4

Number of instructions = 11 / (200 * 10⁶ * 4.4)

Number of instructions = 11 / (880 * 10⁶)

Number of instructions $\approx 12,500,000$ instructions

Therefore, the number of instructions in the program is approximately 12,500,000 instructions.

(c) To calculate the required clock rate to execute the same program in 10 seconds, we can rearrange the formula from part (b) as follows:

$$\text{Clock rate} = \text{Execution time} / (\text{Number of instructions} * \text{Average CPI})$$

Given:

$$\text{Execution time} = 10 \text{ seconds}$$

$$\text{Number of instructions} = 12,500,000$$

$$\text{Average CPI} = 4.4$$

$$\text{Clock rate} = 10 / (12,500,000 * 4.4)$$

$$\text{Clock rate} \approx 180 \text{ MHz}$$

Therefore, to execute the same program in 10 seconds, the clock rate would need to be approximately 180 MHz.

Q3

Part (ii)

(a) CPI ?

→ clock rate = 200 MHz

i) Load & Store :

freq = 30%

cycle = 6

(ii) Arithmetic instr.

freq = 50%

cycles = 4

(iii) All other :

freq = 20%

cycles = 3

avg. CPI ?

$$\text{avg CPI} = \frac{f_1 \times \text{CPI}_1 + f_2 \times \text{CPI}_2 + f_3 \times \text{CPI}_3}{100}$$

$$= \frac{(30 \times 6) + (50 \times 4) + (20 \times 3)}{100}$$

$$= \frac{180 + 200 + 60}{100} = \frac{440}{100} = 4.4$$

(b) Exec time = 11 sec.

clock rate = 200 MHz

Avg CPI = 4.4

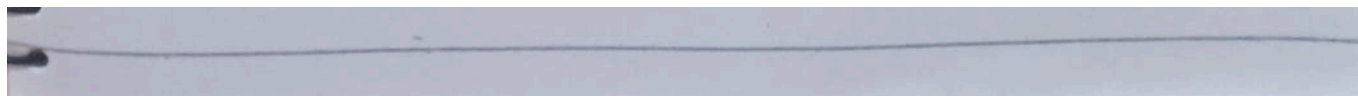
of instr.s = 11

$$\frac{(200 \times 10^6 \times 4.4)}{11} = 12,500,000 \text{ instructions}$$

(c) clock rate = Exec. time

of instr. × avg CPI

$$= \frac{10}{(12500000 \times 4.4)} = 18 \text{ MHz}$$



Q4)

Part(i)

Using a loop and indexed addressing, write code that rotates the member of a 32-bit integer array forward one position. The value at end of array must wrap around to the first position. For example , the array, [10,20,30,40] would be transformed into [40,10,20,30].

; Include Irvine32 library

include irvine32.inc

.data

array DWORD 10, 20, 30, 40

.code

main PROC

; Initialize console

call Clrscr

; Get the size of the array

mov ecx, LENGTHOF array

; Rotate the array elements

mov esi, OFFSET array ; Source index

mov edi, esi ; Destination index

add edi, 4 ; Move destination index forward by 4 bytes (32 bits)

dec ecx ; Decrement the loop counter

mov eax, [esi] ; Store the first element in EAX

rotateLoop:

mov ebx, [edi] ; Store the current destination element in EBX

mov [esi], ebx ; Move the destination element to the source position

add esi, 4 ; Move source index forward by 4 bytes (32 bits)

add edi, 4 ; Move destination index forward by 4 bytes (32 bits)

loop rotateLoop ; Repeat the loop until ECX is zero

; Move the stored first element to the last position

mov [esi], eax

; Display the rotated array

mov ecx, LENGTHOF array

mov esi, OFFSET array

```

displayLoop:
mov eax, [esi]
call WriteInt
call Crlf
add esi, 4
loop displayLoop
; Exit program
call WaitMsg
call ExitProcess
main ENDP
END main

```

Part (ii)

Write a non-recursive version of the factorial procedure that uses a loop . Write a short program that interactively test your factorial procedure. Let the user enter the value of n. If overflow occurs in your loop when calculating each factorial value, your program should display an error message. If no overflow occurs, display the calculated factorial. write assembly language program that uses irvine32 library

```

; Include Irvine32 library
include irvine32.inc

.data
prompt BYTE "Enter a positive integer (n): ", 0
result BYTE "Factorial: ", 0
error BYTE "Overflow occurred!", 0

.code
main PROC
; Initialize console
call Clrscr
; Display prompt and get user input
mov edx, OFFSET prompt
call WriteString
mov eax, 0 ; Initialize factorial accumulator
mov ebx, 1 ; Initialize loop counter
mov ecx, 1 ; Initialize multiplication counter
inputLoop:

```

```

call ReadInt ; Read user input
mov edx, eax ; Save the value of 'n' in edx
; Calculate factorial
mov eax, 1 ; Reset accumulator to 1
factorialLoop:
cmp ecx, edx ; Compare multiplication counter with 'n'
jg displayResult ; If multiplication counter > 'n', display result
imul eax, ecx ; Multiply accumulator by multiplication counter
inc ecx ; Increment multiplication counter
jno factorialLoop ; Jump if no overflow occurred
jmp displayError ; Jump if overflow occurred
displayResult:
; Display the calculated factorial
mov edx, OFFSET result
call WriteString
mov eax, edx ; Move the factorial value to EAX
call WriteInt ; Display the factorial
call Crlf
jmp exitProgram
displayError:
; Display overflow error message
mov edx, OFFSET error
call WriteString
call Crlf
jmp exitProgram
exitProgram:
; Exit program
call WaitMsg
call ExitProcess
main ENDP
END main

```

Q5

Part(i)

Write an assembly code to convert a binary string, declared in data segment, into an unsigned value and store it in EAX register. For example, if given string is '1110' the EAX register must have 14 in it. If the binary string is greater than 32 digits length, a value 0 must be placed in EAX. (Hint: SHL and OR).

```
; Include Irvine32 library
include irvine32.inc

.data
binaryString BYTE "1110", 0
binaryLength EQU $ - binaryString - 1

.code
main PROC
; Initialize console
call Clrscr

; Display the binary string
mov edx, OFFSET binaryString
call WriteString
call Crlf

; Convert binary string to unsigned value
mov eax, 0 ; Clear EAX register
mov ecx, binaryLength ; Set loop counter to string length
convertLoop:
cmp ecx, 32 ; Check if string length exceeds 32 digits
jg endProgram ; Jump to end if string length > 32 digits
movzx edx, byte ptr [edx] ; Load current character as zero-extended byte
cmp edx, '0' ; Compare current character with '0'
jb endProgram ; Jump to end if invalid character (less than '0')
cmp edx, '1' ; Compare current character with '1'
ja endProgram ; Jump to end if invalid character (greater than '1')
shl eax, 1 ; Shift left EAX by 1
add eax, edx ; Add the current bit (0 or 1) to EAX
inc edx ; Move to the next character in the binary string
loop convertLoop ; Repeat the loop until ECX is zero
endProgram:

; Display the result
mov edx, OFFSET binaryString
```

```

call WriteString
cmp ecx, 32 ; Check if string length exceeds 32 digits
jg displayOverflow ; Jump to display overflow message if string length > 32 digits
mov edx, OFFSET binaryString
call WriteString
call Crlf
mov edx, OFFSET binaryString
call WriteString
mov eax, 0 ; Place value 0 in EAX if overflow occurred
call WriteInt
exitProgram:
; Exit program
call WaitMsg
call ExitProcess
displayOverflow:
; Display overflow message
mov edx, OFFSET binaryString
call WriteString
call Crlf
jmp exitProgram
main ENDP
END main

```

Part (ii)

Write an assembly language procedure which will count the number of times vowel characters occur within a NULL terminated lowercase string. The procedure will be called with EDI having the address of string to be searched and should return with the count in CX. For example, if the procedure is called for the following string, it must return 3 in CX: str byte "hello world", 0 The procedure should return with all registers preserved (except CX, which contains the return value). You are not allowed to refer to str variable inside the procedure. NOTE: A NULL terminated string is a sequence of ASCII characters with the end of the sequence indicated by a byte containing the value 0

```

; Include Irvine32 library
include irvine32.inc

.data

str BYTE "hello world", 0
vowels BYTE "aeiou", 0

```

```

.code
countVowels PROC
push ebx ; Preserve EBX register
push esi ; Preserve ESI register
xor ecx, ecx ; Initialize count to zero
mov esi, edi ; Copy the address of the string to ESI
countLoop:
lodsb ; Load the next character into AL from DS:ESI
cmp al, 0 ; Check if end of string (NULL terminator)
je endCountLoop ; Jump to end the count loop if end of string
mov ebx, OFFSET vowels ; Load the address of the vowels string into EBX
findVowel:
cmp al, [ebx] ; Compare the current character with the current vowel
je incrementCount ; Jump to increment the count if a vowel is found
inc ebx ; Move to the next vowel character
cmp byte ptr [ebx], 0 ; Check if end of vowels string
jne findVowel ; Jump to continue searching for vowels
incrementCount:
inc ecx ; Increment the count of vowels
jmp countLoop ; Jump to continue counting
endCountLoop:
mov eax, ecx ; Move the count value to EAX for returning
pop esi ; Restore ESI register
pop ebx ; Restore EBX register
ret
countVowels ENDP
main PROC
; Initialize console
call Clrscr
; Call the countVowels procedure
lea edi, str ; Load the address of the string into EDI
call countVowels
; Display the result

```

```
mov edx, OFFSET str
call WriteString
call Crlf
mov edx, OFFSET str
call WriteString
mov eax, ecx ; Move the count value from CX to EAX for displaying
call WriteInt
call Crlf
; Exit program
call WaitMsg
call ExitProcess
main ENDP
END main
```