

END SEMESTER ASSESSMENT (ESA) B. TECH IV SEMESTER- June 2020

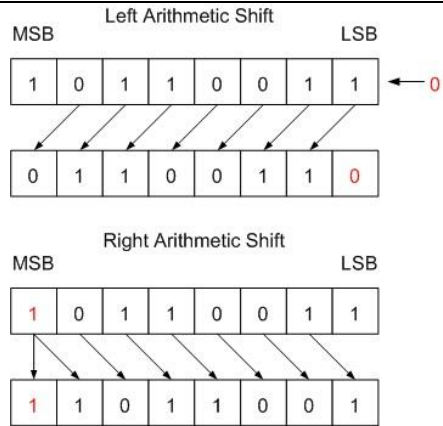
UE18CS253 – Microprocessor and Computer Architecture

Time: 3 Hrs	solution set	Max Marks: 100
-------------	--------------	----------------

Note: All answers must be precise and to the point.

1.	a)	<p>Differentiate the following.</p> <ol style="list-style-type: none"> i. Microprocessor and Microcontrollers. ii. RISC and CISC with respect to processor architecture. <p>Solution:</p> <p>Microprocessor:</p> <ul style="list-style-type: none"> A microprocessor is a computer processor that incorporates the functions of a computer's <u>central processing unit</u> (CPU) on a single <u>integrated circuit</u> (IC). Multi purpose <u>programmable</u> device. Accepts <u>digital data</u> as input, processes it according to instructions stored in its memory, and provides results as output. Brain of the computer. Multiple microprocessors, working together, are the "hearts" of datacenters, super-computers, communications products, and other digital devices. Don't have RAM, ROM, and other peripheral on the chip Desktop PC's, Laptops, notepads . Intel's Pentium 1,2,3,4, core 2 duo, i3, i5 etc <p>Microcontroller:</p> <ul style="list-style-type: none"> Microcontroller has a CPU, in addition with a fixed amount of RAM, ROM and other peripherals all embedded on a single chip. Also termed as a mini computer or a computer on a single chip. Designed to perform specific tasks. Requires small resources like RAM, ROM, I/O ports etc. and hence can be embedded on a single chip. Reduces the size and the cost. Applications : Washing machine, Digicam, Pendrive, Remote, Microwave, Cars, Bikes, Telephone, Mobiles, Watches, etc. Manufacturers : ATMEL, Microchip, Texas Ins., Freescale, Philips, Motorola etc., <p>RISC and CISC:</p>	4
----	----	---	---

	<table><tr><th>CISC</th><th>RISC</th></tr><tr><td>Emphasis on hardware</td><td>Emphasis on software</td></tr><tr><td>Multiple instruction sizes and formats</td><td>Instructions of same set with few formats</td></tr><tr><td>Less registers</td><td>Uses more registers</td></tr><tr><td>More addressing modes</td><td>Fewer addressing modes</td></tr><tr><td>Extensive use of microprogramming</td><td>Complexity in compiler</td></tr><tr><td>Instructions take a varying amount of cycle time</td><td>Instructions take one cycle time</td></tr><tr><td>Pipelining is difficult</td><td>Pipelining is easy</td></tr></table>	CISC	RISC	Emphasis on hardware	Emphasis on software	Multiple instruction sizes and formats	Instructions of same set with few formats	Less registers	Uses more registers	More addressing modes	Fewer addressing modes	Extensive use of microprogramming	Complexity in compiler	Instructions take a varying amount of cycle time	Instructions take one cycle time	Pipelining is difficult	Pipelining is easy	
CISC	RISC																	
Emphasis on hardware	Emphasis on software																	
Multiple instruction sizes and formats	Instructions of same set with few formats																	
Less registers	Uses more registers																	
More addressing modes	Fewer addressing modes																	
Extensive use of microprogramming	Complexity in compiler																	
Instructions take a varying amount of cycle time	Instructions take one cycle time																	
Pipelining is difficult	Pipelining is easy																	
b)	<p>Explain logical and arithmetic shift instructions with an example.</p> <p>Logical Shift</p> <ul style="list-style-type: none">• A Left Logical Shift of one position moves each bit to the left by one. The vacant least significant bit (LSB) is filled with zero and the most significant bit (MSB) is discarded.• A Right Logical Shift of one position moves each bit to the right by one. The least significant bit is discarded and the vacant MSB is filled with zero. <div><p>Left Logical Shift</p><p>MSB </p></div>																	



- c) **Write a general structure of subroutine. Write a Recursive Program to find the factorial of a Number using subroutine.**

2+4

Solution:

Structure of Subroutine:

main:

...

// call

...

bl subr

// call subroutine

ret_addr: add r0, r1, r2

// return

...

subr:

...

// body of the subroutine

...

bx lr

// return to caller

Or

mov pc, lr

Program:

MOV R0,#5

MOV R1,R0

MOV R2,#1

factorial:

STMFD R13!, { R1, R2,LR}

CMP R0,#1

BEQ ret

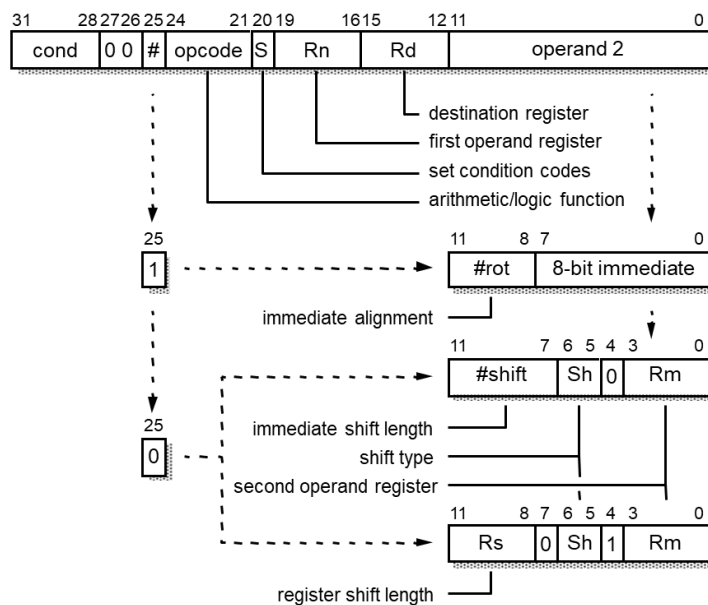
MOV R1, R0

SUB R0, R0, #1

BL factorial

MUL R2, R1, R0

	MOV R0, R2 LDMFD R13!, { R1, R2,LR} MOV PC,LR ret: MOV R0, #1 LDMFD R13!, { R1, R2,LR} MOV PC,LR	
d)	<p>i. Explain how to encode data processing instructions .</p> <p>Solution: <u>Machine code</u> is what computer processors run on: <u>binary</u> representations of simple instructions. All ARM processors</p> <ul style="list-style-type: none"> Any instruction with bits 27 and 26 as 00 is data processing. The four-bit opcode field in bits 24–21 defines exactly which instruction this is: add, subtract, move, compare, and so on. 0100 is ADD Bit 25 is the "immediate" bit. If it's 0, then operand 2 is a register. If it's set to 1, then operand 2 is an immediate value. Note that operand 2 is only 12 bits. That doesn't give a huge range of numbers: 0–4095, or a byte and a half. Not great when you're mostly working with 32-bit numbers and addresses. <p>i. Encode the ARM instruction: ADDS R1, R0, R2 LSR R4 (opcode for ADD is 0100)</p> <p>Solution: 1110 000 0100 S Rn Rd Shift Rm 1110 000 0100 S Rn Rd 00010 01 0 Rm </p>	6



- Any instruction with bits 27 and 26 as 00 is data processing. The four-bit opcode field in bits 24–21 defines exactly which instruction this is: add, subtract, move, compare, and so on. 0100 is ADD
- Bit 25 is the "immediate" bit. If it's 0, then operand 2 is a register. If it's set to 1, then operand 2 is an immediate value.
- Note that operand 2 is only 12 bits. That doesn't give a huge range of numbers: 0–4095, or a byte and a half. Not great when you're mostly working with 32-bit numbers and addresses.

i. Encode the ARM instruction: ADDS R1, R0, R2 LSR R4 (opcode for ADD is 0100)

Solution:

1110 000 0100 S Rn Rd Shift Rm

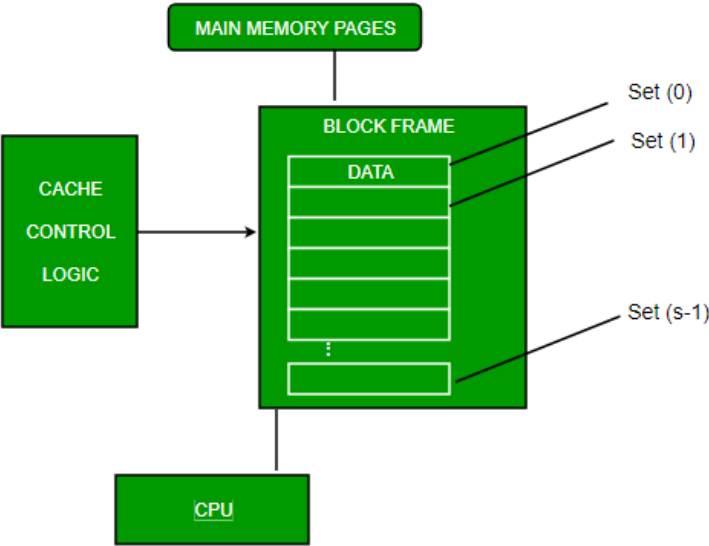
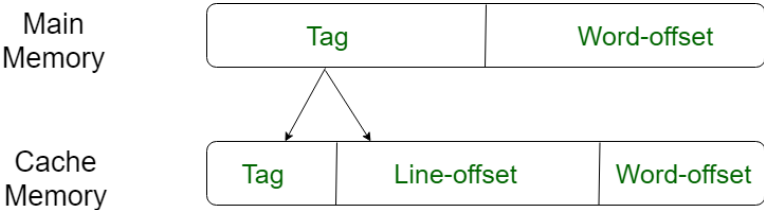
1110 000 0100 S Rn Rd 00010 01 0 Rm

		1110 0000 1001 0000 0001 0001 0010 0010 E0901122	
2	a)	<p>Define Pipeline processing. Explain various stages in a Pipeline execution with neat diagram.</p> <p>Solution:</p> <p>Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.</p> <p>The 5 stages of instruction execution in a pipelined processor are:</p> <p>Stage 1 (Instruction Fetch) : The instruction is fetched from memory and placed in the instruction pipeline. Update the PC to the next sequential PC by adding 4 to PC.</p> <p>Stage 2 (Instruction Decode) : The instruction is decoded and register operands read from the register files. There are 3 operand read ports in the register file so most ARM instructions can source all their operands in one cycle.</p> <p>Stage 3 (Instruction Execute):In this stage, ALU operations are performed.</p> <p>Stage 4 (Memory Access): Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle. If the instruction is a LOAD, the memory does a read using effective address computed in the previous cycle. If the instruction is a STORE, then the memory writes the data from the second register read using the effective address.</p> <p>Stage 5 (Write Back):In this stage, computed/fetched value is written back to the register present in the instructions.</p>	6

	<p style="text-align: center;">Ideal Pipelined Instruction Processing Representation (i.e no stall cycles)</p> <p style="text-align: center;">← Pipeline Fill cycles = 5 - 1 = 4 →</p> <p style="text-align: center;">Time 2 3 4 5 6 7 8 9 10</p> <p style="text-align: center;">1 2 3 4 5 IF ID EX MEM WB</p> <p>I1 IF ID EX MEM WB</p> <p>I2 IF ID EX MEM WB</p> <p>I3 IF ID EX MEM WB</p> <p>I4 IF ID EX MEM WB</p> <p>I5 IF ID EX MEM WB</p> <p>I6 IF ID EX MEM WB</p> <p>Program Flow ↓</p> <p>Any individual instruction goes through all five pipeline stages taking 5 cycles to complete Thus instruction latency = 5 cycles</p> <p>Here $n = 5$ pipeline stages or steps</p> <p>Number of pipeline fill cycles = Number of stages - 1 Here $5 - 1 = 4$</p> <p>After fill cycles: One instruction is completed every cycle (Effective CPI = 1) (ideally)</p> <p style="text-align: right;">EECC550 - Shaaban</p> <p style="text-align: right;">#9 Final Exam Review Winter 2009 2-18-2010</p> <p><u>Ideal pipeline operation without any stall cycles</u></p>	
b)	<p>Define Latency and throughput with respect to pipeline. Give one example</p> <p>Pipeline throughput: instructions completed per second.</p> <p>Pipeline latency: how long does it take to execute a single instruction in the pipeline.</p> <p style="text-align: center;"> </p> <p>Pipeline throughput:</p> $= 1 \text{ instr} / \max[\text{lat}(\text{IF}), \text{lat}(\text{ID}), \text{lat}(\text{EX}), \text{lat}(\text{MEM}), \text{lat}(\text{WB})]$ $= 1 \text{ instr} / \max[5 \text{ ns}, 4 \text{ ns}, 5 \text{ ns}, 10 \text{ ns}, 4 \text{ ns}]$ $= 1 \text{ instr} / 10 \text{ ns} \quad (\text{ignoring pipeline register overhead})$ $L = \text{lat}(\text{IF}) + \text{lat}(\text{ID}) + \text{lat}(\text{EX}) + \text{lat}(\text{MEM}) + \text{lat}(\text{WB})$ <p>Pipeline Latency= $= 5 \text{ ns} + 4 \text{ ns} + 5 \text{ ns} + 10 \text{ ns} + 4 \text{ ns} = 28 \text{ ns}$</p>	2
c)	<p>Consider a non-pipelined processor with a clock rate of 2.5 gigahertz and average cycles per instruction of 4. The same processor is upgraded to a pipelined processor with five stages but due to the internal pipeline delay, the clock speed is reduced to 2 gigahertz. Assume there are no stalls in the pipeline. What is the speed up achieved in this pipelined processor?</p> <p>Solution:</p>	4

		<p>Frequency of the clock = 2.5 gigahertz</p> <p>Cycle time = 1 / frequency</p> $= 1 / (2.5 \text{ gigahertz})$ $= 1 / (2.5 \times 10^9 \text{ hertz})$ $= 0.4 \text{ ns}$ <p><u>Non-Pipeline Execution Time-</u></p> <p>Non-pipeline execution time to process 1 instruction = Number of clock cycles taken to execute one instruction</p> $= 4 \text{ clock cycles}$ $= 4 \times 0.4 \text{ ns}$ $= 1.6 \text{ ns}$ <p><u>Cycle Time in Pipelined Processor-</u></p> <p>Frequency of the clock = 2 gigahertz</p> <p>Cycle time = 1 / frequency</p> $= 1 / (2 \text{ gigahertz})$ $= 1 / (2 \times 10^9 \text{ hertz})$ $= 0.5 \text{ ns}$ <p><u>Pipeline Execution Time-</u></p> <p>Since there are no stalls in the pipeline, so ideally one instruction is executed per clock cycle. So,</p> <p>Pipeline execution time = 1 clock cycle</p> $= 0.5 \text{ ns}$ <p><u>Speed Up-</u></p> $\text{Speed up} = \text{Non-pipeline execution time} / \text{Pipeline execution time}$ $= 1.6 \text{ ns} / 0.5 \text{ ns}$ $= 3.2$	
	d)	<p>Explain in detail, the pipeline hazards.</p> <p>Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles.</p> <p>Any condition that causes a stall in the pipeline operations can be called a hazard.</p> <p>There are primarily three types of hazards:</p> <p>i. Data Hazards</p>	8

		<p>ii. Control Hazards or instruction Hazards</p> <p>iii. Structural Hazards.</p> <p>i. Data Hazards:</p> <p>A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result of which some operation has to be delayed and the pipeline stalls. Whenever there are two instructions one of which depends on the data obtained from the other.</p> <p>$A=3+A$</p> <p>$B=A*4$</p> <p>For the above sequence, the second instruction needs the value of 'A' computed in the first instruction.</p> <p>Thus the second instruction is said to depend on the first.</p> <p>If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.</p> <p>ii. Structural Hazards:</p> <p>This situation arises mainly when two instructions require a given hardware resource at the same time and hence for one of the instructions the pipeline needs to be stalled.</p> <p>The most common case is when memory is accessed at the same time by two instructions. One instruction may need to access the memory as part of the Execute or Write back phase while other instruction is being fetched. In this case if both the instructions and data reside in the same memory. Both the instructions can't proceed together and one of them needs to be stalled till the other is done with the memory access part. Thus in general sufficient hardware resources are needed for avoiding structural hazards.</p> <p>iii. Control hazards:</p> <p>The instruction fetch unit of the CPU is responsible for providing a stream of instructions to the execution unit. The instructions fetched by the fetch unit are in consecutive memory locations and they are executed.</p> <p>However the problem arises when one of the instructions is a branching instruction to some other memory location. Thus all the instruction fetched in the pipeline from consecutive memory locations are invalid now and need to be removed (also called flushing of the pipeline). This induces a stall till new instructions are again fetched from the memory address specified in the branch instruction.</p> <p>Thus the time lost as a result of this is called a branch penalty. Often dedicated hardware is incorporated in the fetch unit to identify branch instructions and compute branch addresses as soon as possible and reducing the resulting delay as a result.</p>	
3.	a)	<p>With figure explain about direct mapping cache memory.</p> <p>Solution:</p> <p>The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or</p>	4

	<p>In Direct mapping, assigne each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.</p> <p>$i = j \text{ modulo } m$</p> <p>where</p> <p>i=cache line number</p> <p>j= main memory block number</p> <p>m=number of lines in the cache</p> <div>  <p>The diagram illustrates the direct mapping cache architecture. It features a central 'BLOCK FRAME' containing multiple 'DATA' slots. To the left, a 'CACHE CONTROL LOGIC' block is connected to the 'BLOCK FRAME'. Above the 'BLOCK FRAME', a 'MAIN MEMORY PAGES' block is connected. Below the 'BLOCK FRAME', a 'CPU' block is connected. On the right side, specific sets within the 'BLOCK FRAME' are labeled: 'Set (0)', 'Set (1)', and 'Set (s-1)'.</p> </div> <p>For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory. In most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of s-r bits (most significant portion) and a line field of r bits. This latter field identifies one of the m=2^r lines of the cache</p> <div>  <p>This diagram shows how a main memory address is mapped to cache memory. The 'Main Memory' address is divided into a 'Tag' and a 'Word-offset'. The 'Tag' is mapped to the 'Tag' field of the 'Cache Memory', and the 'Word-offset' is mapped to the 'Word-offset' field. The 'Line-offset' field in the 'Cache Memory' is also shown, which is derived from the 'Tag' field.</p> </div>	
b)	<p>A computer system uses 16-bit memory addresses. It has a 2K-byte cache organized in a direct-mapped manner with 64 bytes per cache block. Assume that the size of each memory word is 1 byte.</p> <ol style="list-style-type: none"> Calculate the number of bits in each of the Tag, Block, and Word fields of the memory address. When a program is executed, the processor reads data sequentially from the following word addresses: 128, 144, 2176, 2180, 128, 2176 	6

All the above addresses are shown in decimal values. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Solution:

i. Block size = 64 bytes = 2^6 bytes = 2^6 words (since 1 word = 1 byte)

Therefore, Number of bits in the Word field = 6

Cache size = 2K-byte = 2^{11} bytes

Number of cache blocks = Cache size / Block size = $2^{11} / 2^6 = 2^5$

Therefore, Number of bits in the Block field = 5

Total number of address bits = 16

Therefore, Number of bits in the Tag field = $16 - 6 - 5 = 5$

For a given 16-bit address, the 5 most significant bits, represent the Tag, the next 5 bits represent the Block, and the 6 least significant bits represent the Word.

ii.

The cache is initially empty. Therefore, all the cache blocks are invalid.

Access # 1:

Address = $(128)_{10} = (0000000010000000)_2$

(Note: Address is shown as a 16-bit number, because the computer uses 16-bit addresses)

For this address, Tag = 00000, Block = 00010, Word = 000000

Since the cache is empty before this access, this will be a cache miss

After this access, Tag field for cache block 00010 is set to 00000

Access # 2:

Address = $(144)_{10} = (0000000010010000)_2$

For this address, Tag = 00000, Block = 00010, Word = 010000

Since tag field for cache block 00010 is 00000 before this access, this will be a cache hit

(because address tag = block tag)

Access # 3:

Address = $(2176)_{10} = (0000100010000000)_2$

For this address, Tag = 00001, Block = 00010, Word = 000000

Since tag field for cache block 00010 is 00000 before this access, this will be a cache miss (address tag \neq block tag)

After this access, Tag field for cache block 00010 is set to 00001

Access # 4:

Address = $(2180)_{10} = (0000100010000100)_2$

For this address, Tag = 00001, Block = 00010, Word = 000100

Since tag field for cache block 00010 is 00001 before this access, this will be a cache hit (address tag = block tag)

Access # 5:

Address = $(128)_{10} = (0000000010000000)_2$

For this address, Tag = 00000, Block = 00010, Word = 000000

Since tag field for cache block 00010 is 00001 before this access, this will be a cache miss (address tag \neq block tag) After this access, Tag field for cache block 00010 is set to 00000

Access # 6:

Address = $(2176)_{10} = (0000100010000000)_2$

For this address, Tag = 00001, Block = 00010, Word = 000000

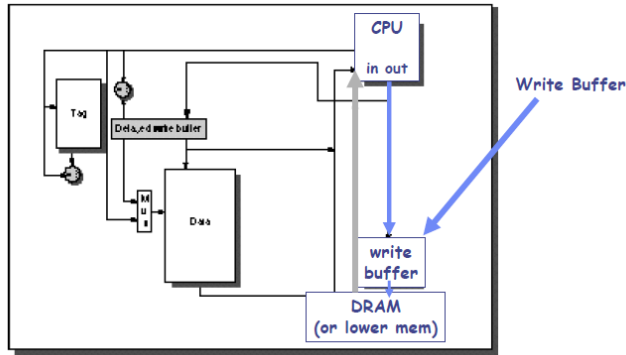
Since tag field for cache block 00010 is 00001 before this access, this will be a cache miss (address tag \neq block tag) After this access, Tag field for cache block 00010 is set to 00001

- Cache hit rate = Number of hits / Number of accesses = $2/6 = 0.333$

c)	<p>What is cash miss? Explain the different categories of misses.</p> <p>Cache miss: A cache miss is an event in which a system or application makes a request to retrieve data from a cache, but that specific data is not currently in cache memory.</p> <p>Different categories of misses:</p> <ol style="list-style-type: none"> 1. Compulsory: On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses. 2. Capacity: Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (program working set is much larger than cache capacity). 3. Conflict: In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame; also called collision misses or interference misses. 	4
d)	<p>Consider a 2-way set associative cache memory with 4 sets and total 8 cache blocks (0-7) and a main memory with 128 blocks (0-127). What memory blocks will be present in the cache after the following sequence of memory block references if LRU policy is used for cache block replacement. Assuming that initially the cache did not have any memory block from the current job?</p> <p>0 5 3 9 7 0 16 55</p> <p>Solution:</p> <p>2-way set associative cache memory, i.e $K = 2$.</p> <p>No of sets is given as 4, i.e. $S = 4$ (numbered 0 - 3)</p> <p>No of blocks in cache memory is given as 8, i.e. $N = 8$ (numbered from 0 - 7)</p> <p>Each set in cache memory contains 2 blocks.</p> <p>The number of blocks in the main memory is 128, i.e $M = 128$. (numbered from 0 - 127)</p> <p>A referred block numbered X of the main memory is placed in the set numbered $(X \bmod S)$ of the the cache memory. In that set, the block can be placed at any location, but if the set has already become full, then the current referred block of the main memory should replace a block in that set according to some replacement policy. Here the replacement policy is LRU (i.e. Least Recently Used block should be replaced with currently referred block).</p> <p>X (Referred block no) and the corresponding Set values are as follows:</p> <p>$X \rightarrow \text{set no } (X \bmod 4)$</p> <p>0$\rightarrow$0 (block 0 is placed in set 0, set 0 has 2 empty block locations, block 0 is placed in any one of them)</p> <p>5\rightarrow1 (block 5 is placed in set 1, set 1 has 2 empty block locations, block 5 is placed in any one of them)</p> <p>3\rightarrow3 (block 3 is placed in set 3, set 3 has 2 empty block locations, block 3 is placed in any one of them)</p> <p>9\rightarrow1 (block 9 is placed in set 1, set 1 has currently 1 empty block location,</p>	6

		<p>block 9 is placed in that, now set 1 is full, and block 5 is the least recently used block)</p> <p>7--->3 (block 7 is placed in set 3, set 3 has 1 empty block location, block 7 is placed in that, set 3 is full now, and block 3 is the least recently used block)</p> <p>0--->block 0 is referred again, and it is present in the cache memory in set 0, so no need to put again this block into the cache memory.</p> <p>16--->0 (block 16 is placed in set 0, set 0 has 1 empty block location, block 0 is placed in that, set 0 is full now, and block 0 is the LRU one)</p> <p>55--->3 (block 55 should be placed in set 3, but set 3 is full with block 3 and 7, hence need to replace one block with block 55, as block 3 is the least recently used block in the set 3, it is replaced with block 55.</p> <p>Hence the main memory blocks present in the cache memory are : 0, 5, 7, 9, 16, 55 .</p>	
4.	a)	<p>Explain the cache optimization: Giving priority to Read misses over Write misses to reduce miss penalty.</p> <p>Solution:</p> <p>Consider the following code sequence.</p> <p>Ex: STR R3, 512 (R0) LDR R1, 1024 (R0) LDR R2, 512 (R0)</p> <p>Assume Direct mapped:</p> <p>Write-through cache that maps 512 and 1024 to the same block. Four word write buffer that is not checked on a read miss. Will the value in R2 always be equal to the value in R3?</p> <ul style="list-style-type: none"> • R3-> Write buffer • R1<- M[1024]-same cache index; read miss • R2<- M[512], if write not completed, old value. <p>• Hence, $R3 \neq R2$</p> <ul style="list-style-type: none"> • This is a read-after-write data hazard in memory. • The data in R3 are placed into the write buffer after the STR. • The following LDR instruction uses the same cache index and is therefore a miss. • The second LDR instruction, tries to put the value in location 512 into the register R2. • This also results in a miss. • If the write buffer hasn't completed writing to location 512 in memory, • The read of location 512 will put the old, wrong value into the cache block and then into R2. • Without proper precautions, R3 would not be equal to R2! 	4

Read Priority over Write on Miss



6

The simplest way for this dilemma is for the read miss to wait until the write buffer is empty.

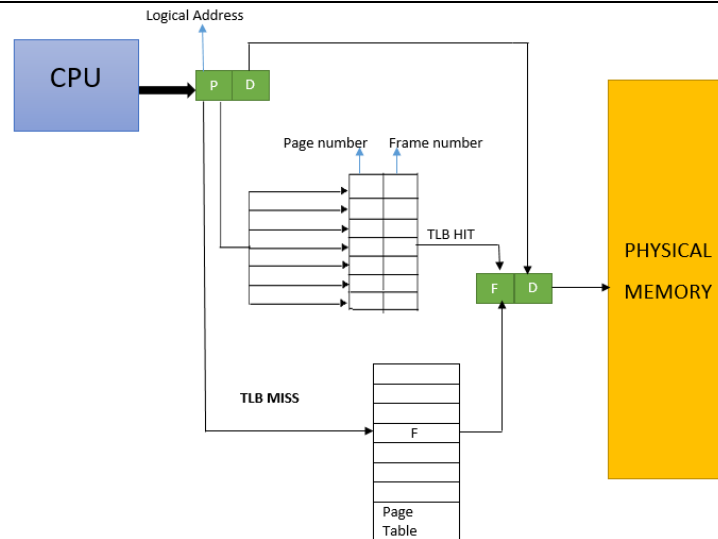
- The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts, and the memory system is available, let the read miss continue.
- Virtually all the desktop and server processors use the later approach, giving reads priority over writes.
- The cost by the processor in write-back cache can also be reduced.
- If a read miss occurs, the processor can either stall until the buffer is empty or check the address of the words in the buffer for conflicts

b) Write a short note on TLB

4

Answer:

- A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location.[1][2] It is a part of the chip's memory-management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache. A TLB may reside between the CPU and the CPU cache, between CPU cache and the main memory or between the different levels of the multi-level cache. The majority of desktop, laptop, and server processors include one or more TLBs in the memory-management hardware, and it is nearly always present in any processor that utilizes paged or segmented virtual memory.
- The TLB is sometimes implemented as content-addressable memory (CAM). The CAM search key is the virtual address, and the search result is a physical address. If the requested address is present in the TLB, the CAM search yields a match quickly and the retrieved physical address can be used to access memory. This is called a TLB hit. If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page walk. The page walk is time-consuming when compared to the processor speed, as it involves reading the contents of multiple memory locations and using them to compute the physical address. After the physical address is determined by the page walk, the virtual address to physical address mapping is entered into the TLB.



c) **Explain in detail the Flynn's classification of computers.**

Answer:

Flynn's classification divides computers into four major groups that are:

1. Single instruction stream, single data stream (SISD)
2. Single instruction stream, multiple data stream (SIMD)
3. Multiple instruction stream, single data stream (MISD)
4. Multiple instruction stream, multiple data stream (MIMD)

SISD	SIMD
Single Instruction, Single Data	Single Instruction, Multiple Data
MISD	MIMD
Multiple Instruction, Single Data	Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

	<p>Single Instruction, Multiple Data (SIMD)</p> <p>A type of parallel computer Single instruction: All processing units execute the same instruction at any given clock cycle</p> <p>Multiple data: Each processing unit can operate on a different data element</p> <p>This type of machine typically has an instruction dispatcher, a very high- bandwidth internal network, and a very large array of very small-capacity instruction units. Best suited for specialized problems characterized by a high degree of regularity, such as image processing.</p> <p>Synchronous (lockstep) and deterministic execution Two varieties: Processor Arrays and Vector Pipelines</p> <p>Multiple Instruction, Single Data (MISD)</p> <p>A single data stream is fed into multiple processing units.</p> <p>Each processing unit operates on the data independently via independent instruction streams.</p> <p>Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).</p> <p>Some conceivable uses might be: – multiple frequency filters operating on a single signal stream multiple cryptography algorithms attempting to crack a single coded message.</p> <p>Multiple Instruction, Multiple Data (MIMD)</p> <p>Currently, the most common type of parallel computer. Most modern computers fall into this category.</p> <p>Multiple Instruction: every processor may be executing a different instruction stream</p> <p>Multiple Data: every processor may be working with a different data stream</p> <p>Execution can be synchronous or asynchronous, deterministic or non-deterministic</p> <p>Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.</p>	
d)	<p>Explain Amdahl's law and Gustafson's law.</p> <p>Answer:</p> <p>Amdahl's Law: states that potential program speedup is defined by the fraction of code (P) that can be parallelized:</p>	6

Amdahl's law (or **Amdahl's argument**^[1]) is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours to complete using a single thread, but a one hour portion of the program cannot be parallelized, therefore only the remaining 19 hours ($p = 0.95$) of execution time can be parallelized, then regardless of how many threads are devoted to a parallelized execution of this program, the minimum execution time cannot be less than one hour. Hence, the theoretical speedup is limited to at most 20 times the single thread performance, .

Amdahl's law can be formulated in the following

g way:

$$OverallSpeedup = \frac{1}{(1-f) + \frac{f}{s}}$$

where

- S = speed up factor
- F = fraction of program which can be optimized or speed up factor can be applied.
- $(1-f)$ = fraction of program on which speed up factor cannot be applied.

Gustafon's Law

Gustafson's law (or **Gustafson–Barsis's law**^[1]) gives the theoretical speedup in latency of the execution of a task *at fixed execution time* that can be expected of a system whose resources are improved.

Rather than assuming that the problem size is fixed, assume that the parallel execution time is fixed. As problem size is increases, increase the parallel processing units (N).

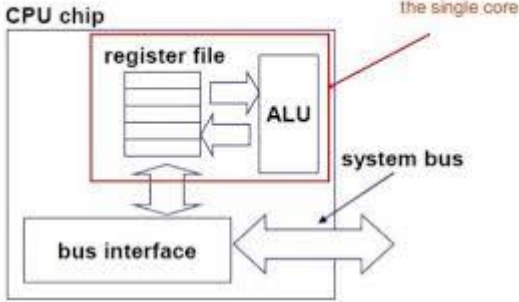
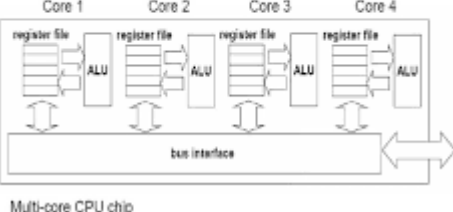
Gustafson, makes the case that the serial section of the code does not increase with the problem size.

The speed up factor is:

$$S_s(n) = \frac{s + np}{s + p}$$

$$= s + np$$

		$= n + (1 - n)s$ <p>Where $p = 1 - s$</p> <p>The speed up factor is called Scaled Speed-up Factor</p>	
5.	a)	<p>Define the following terms:</p> <p>ILP: Instruction-level parallelism (ILP) is a measure of how many operations in a computer program can be performed "in-parallel" at the same time. ILP allows the compiler and the processor to overlap the execution of multiple instructions or even to change the order in which instructions are executed.</p> <p>VLIW: A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.</p> <p>superscalar processor: A superscalar processor is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor. It therefore allows for more throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an arithmetic logic unit.</p> <p>Speculation: Speculation is an approach that allows the compiler or the processor to guess about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction.</p>	4
	b)	<p>What is loop unrolling? Write about limits of loop unrolling.</p> <p>Answer:</p> <p>Loop Unrolling: An important compiler technique to get more performance from loops is loop unrolling, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations</p> <p>Limits: Amount of loop overhead amortized with each unroll</p> <ul style="list-style-type: none"> • Unroll 4 times → 2 out of 14 cycles are overhead → 0.5 cycles per iteration • Unroll 8 times → 0.25 cycles per iteration • Growth in code size • Larger loops, code size growth will be a concern • Large code size may increase instruction cache miss rate • Potential shortfall in registers that is created by aggressive unrolling and 	4

	<p>scheduling strategy</p> <ul style="list-style-type: none"> • Register pressure • Scheduling code to increase ILP causes the number of live values to increase, thus generates shortage of registers 	
c)	<p>Explain single core and multicore processor with necessary diagrams.</p> <p>Single core processor: A single-core processor is a <u>microprocessor</u> with a single core on a chip, running a single <u>thread</u> at any one time.</p> <p>Single core processors have only one processor in die to process instructions. All the processor developed by different manufacturers till 2005 were single core. In today's computers we use multicore processors but single core processor also perform very well. Single core processors have been discontinued in new computers, so these are available at very cheap rates.</p> <p>Single-core CPU chip</p>  <p>Multicore processors:</p> <p>A multi-core processor is a <u>computer processor integrated circuit</u> with two or more separate <u>processing units</u>, called <u>cores</u>, each of which reads and executes <u>program instructions</u>, as if the computer had several processors.</p> <p>Multicore processors are the latest processors which became available in the market after 2005. These processors use two or more cores to process instructions at the same time by using hyper threading. The multiple cores are embedded in the same die. The multicore processor may look like a single processor but actually it contains two (dual-core), three (tricore), four (quad-core), six (hexa-core), eight (octa-core) or ten (deca-core) cores. Some processors even have 22 or 32 cores. Due to power and temperature constraints, multicore processors are only a practical solution for increasing the speed of future computers.</p> <p>Multi-Core Architectures</p> <ul style="list-style-type: none"> • Replicate multiple processor cores on a single die. 	6
d)	<p>What is dynamic scheduling? Explain the Dynamic scheduling techniques.</p> <p>Dynamic Scheduling:</p>	6

Dynamic scheduling, is a method in which the hardware determines which instructions to execute, as opposed to a statically scheduled machine, in which the compiler determines the order of execution. In essence, the processor is executing instructions out of order.

Dynamic scheduling offers several *advantages*:

- It enables handling some cases when dependencies are unknown at compile time (e.g., because they may involve a memory reference);
- It simplifies the compiler;
- It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

Dynamic scheduling: The Idea

A major limitation of the pipelining techniques is that they use in-order instruction issue: *if an instruction is stalled in the pipeline, no later instructions can proceed*. Thus, if there is a dependency between two closely spaced instructions in the pipeline, it will stall. For example:

```
DIVD    F0, F2, F4
ADDD    F10, F0, F8
SUBD    F12, F8, F14
```

SUBD instruction cannot execute because the dependency of ADDD on DIVD causes the pipeline to stall; yet SUBD is not data dependent on anything in the pipeline. This is a performance limitation that can be eliminated by not requiring instructions to execute in order.

To allow SUBD to begin executing, we must separate the instruction issue process into two parts: checking the structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in order instruction issue. However, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do *out-of-order execution*, which implies *out-of-order completion*.

In introducing out-of-order execution, we have essentially split the ID pipe stage into two stages:

- **Issue** - Decode instructions, check for structural hazards;
- **Read operands** - Wait until no data hazards, then read operands.

An instruction fetch proceeds with the issue stage and may fetch either into a single-entry latch or into a queue; instructions are then issued from the latch or queue. The EX stage follows the read operands stage, just as in the DLX pipeline. As in the DLX floating-point pipeline, execution may take multiple cycles, depending on the operation. Thus, we may need to distinguish when an instruction begins execution and when it completes execution; between the two times, the instruction is in execution. This allows multiple instructions to be in execution at the same time.

Scoreboarding is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies; it is named after the CDC 6600 scoreboard, which developed this capability.

The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection.

Every instruction goes through the scoreboard, where a record of the data dependencies is constructed; this step corresponds to instruction issue and replaces part of the ID step in the DLX pipeline. The scoreboard then determines when the instruction can read its operands and begin execution.

Tomasulo Approach is another scheme to allow execution to proceed in the presence of hazards developed by the IBM 360/91 floating-point unit. This scheme combines key elements of the scoreboarding scheme with the introduction of register renaming.

In the loop unrolling section we showed how a compiler could rename registers to avoid WAW and WAR hazards. In Tomasulo's scheme this functionality is provided by the reservation stations, which buffer the operands of instructions waiting to issue, and by issue logic.

The basic idea is that a **reservation station** fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register appear, only the last one is actually used to update the register.

As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station in a process called **register renaming**. This combination of issue logic and reservation stations provides renaming and eliminates WAW and WAR hazards.

This additional capability is the major conceptual difference between scoreboarding and Tomasulo's algorithm. Since there can be more reservation stations than real registers, the technique can eliminate hazards that could not be eliminated by a compiler.