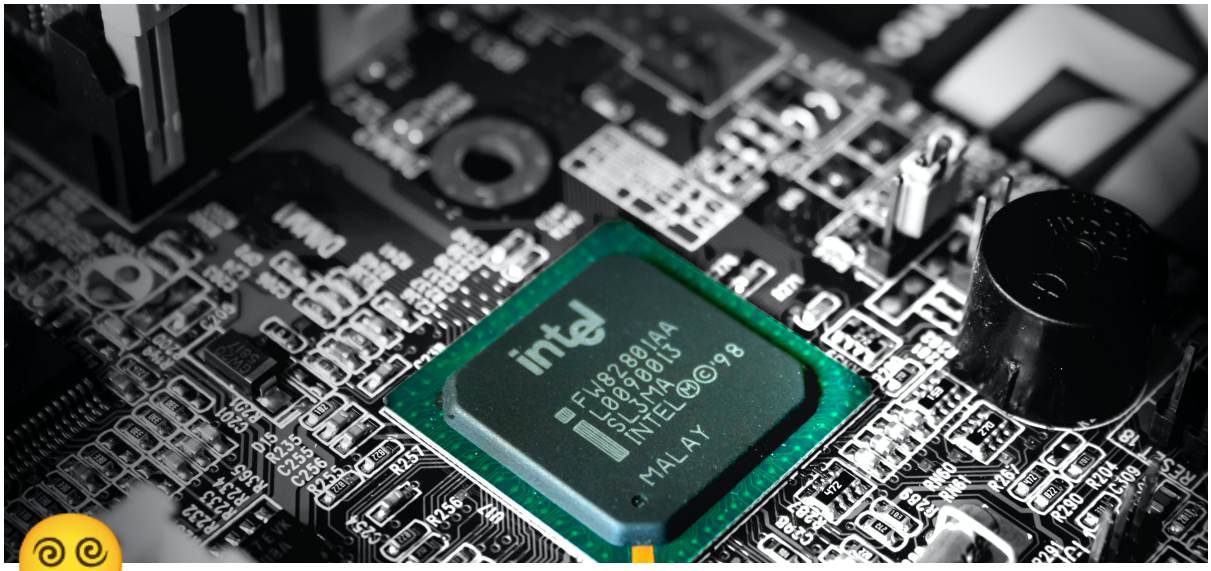


MSP Notes

Class	ET0708 MSP
Year	Year 3

Chapter 1: Intel Architecture

In the realm of computer architecture, understanding the fundamental components of Intel processors is pivotal. This chapter delves into the core aspects of Intel architecture, exploring the intricacies of address buses, data buses, control buses, memory and I/O interfacing, CPU registers, and execution units.

1. Intel Processors and Architecture:

- Intel processors are the driving force behind modern computing devices, encompassing a wide range of products catering to various computing needs.
- The architecture of Intel processors plays a critical role in determining their performance, compatibility, and capabilities.

2. Address Buses, Data Buses, and Control Buses:

- The architecture of Intel processors relies on three essential buses for communication: address bus, data bus, and control bus.

- The address bus carries memory addresses, indicating the location of data or instructions to be accessed.
- The data bus is responsible for transferring actual data between the processor and memory or I/O devices.
- The control bus manages signals that coordinate the operations of different components within the processor.

3. Memory and I/O Interfacing:

- Memory interfacing is a critical aspect of processor design, enabling efficient access to program instructions and data.
- Memory is organized into addresses, and the address bus is used to select specific memory locations for reading or writing.
- I/O interfacing involves connecting the processor to external devices like keyboards, displays, and sensors using specific memory addresses.

4. CPU Registers and Execution Units:

- CPU registers are small, high-speed storage locations within the processor that hold data temporarily during program execution.
- Registers play a vital role in the execution of instructions, data manipulation, and control flow.
- Execution units are functional blocks within the CPU responsible for executing specific types of instructions, such as arithmetic, logical, and memory-related operations.

5. Address Decoding and Bus Control:

- Address decoding is crucial for determining which component should respond to a particular memory or I/O address.
- Bus control logic manages the flow of data and control signals across the buses, coordinating operations between the processor, memory, and I/O devices.

6. Key Concepts in Intel Architecture:

- Understanding the various modes of operation, such as real mode and protected mode.
- Grasping the significance of instruction sets, including various data movement and manipulation instructions.

- Recognizing the role of interrupts and exceptions in managing unexpected events during program execution.

Chapter 2A: Intel Instructions

In the realm of computer architecture, Intel assembly language instructions serve as the foundational elements that drive the execution of programs on Intel processors. This chapter, "Intel Instructions," delves into the intricate world of instruction encoding, formats, and some of the fundamental instructions that shape the functionality of these processors.

1. Intel Assembly Language Instructions:

- Intel processors operate using a set of instructions written in assembly language, a human-readable form of machine code.
- Assembly language bridges the gap between low-level machine instructions and high-level programming languages.

2. Instruction Encoding and Formats:

- Instructions are encoded as binary patterns, with each instruction having a unique encoding.
- Instruction formats define the structure of an instruction, including fields for the operation code (opcode), operands, and addressing modes.

3. Basic Intel Instructions:

- The "MOV" instruction is fundamental, used to move data between registers and memory locations.
- "ADD" and "SUB" instructions perform addition and subtraction operations.
- These basic instructions form the building blocks for more complex operations and program flow control.

4. Data Movement and Manipulation Instructions:

- Intel architecture offers a wide array of instructions for moving, copying, and manipulating data.
- Instructions like "LEA" (Load Effective Address) calculate memory addresses for efficient addressing.

Chapter 2B: Programming

Building upon the foundation laid by instruction understanding, this chapter delves into the practical realm of assembly language programming for Intel processors. As you embark on this journey, you'll learn how to craft simple programs that harness the power of Intel instructions for memory manipulation and arithmetic operations.

1. Assembly Language Programming:

- Assembly language programming involves writing code using mnemonic instructions that correspond to machine-level instructions.
- Programs are written using a combination of instructions, registers, memory addresses, and data.

2. Writing Simple Programs:

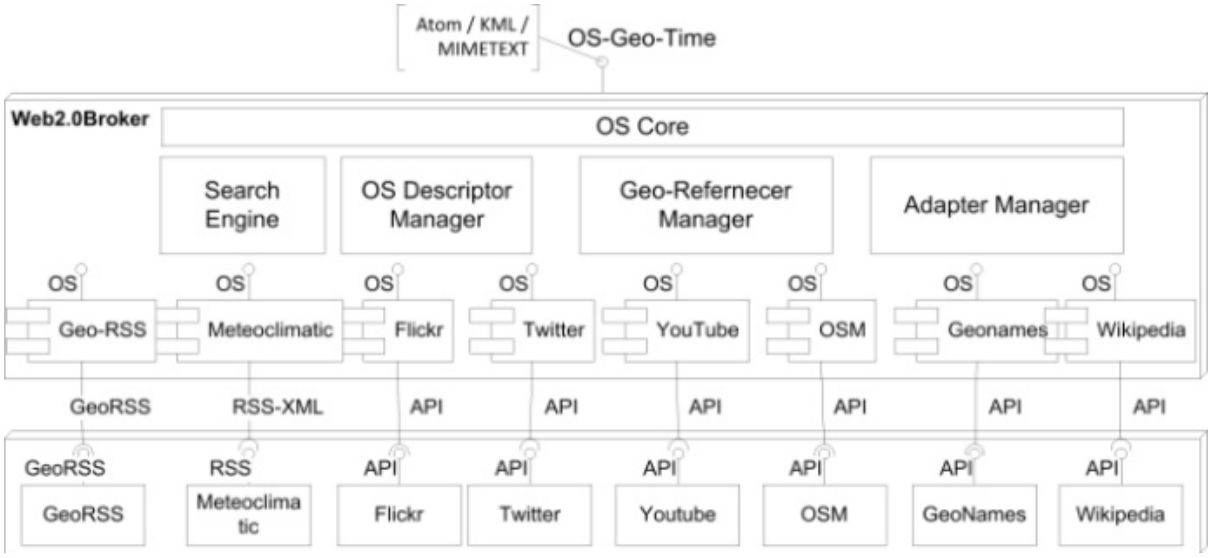
- Simple programs serve as stepping stones, allowing you to grasp the basics of assembly language programming.
- Programs might involve loading data, performing calculations, and displaying results.

3. Memory Manipulation and Arithmetic Operations:

- Assembly language programs often involve reading from and writing to memory locations.
- Arithmetic operations, such as addition, subtraction, multiplication, and division, are vital components of program logic.

Chapter 3: Intel Memory Interfacing and Memory Hierarchy

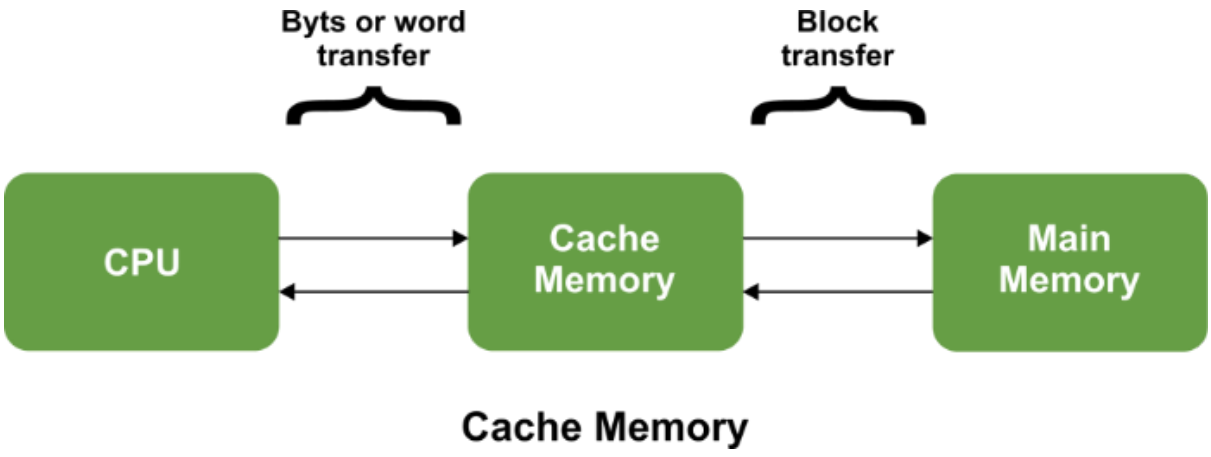
In computer architecture, the memory hierarchy separates computer storage into a hierarchy based on response time. This hierarchy includes various components such as cache memory, main memory, and secondary storage. The principle of the memory hierarchy is to minimize the time it takes to access memory. This is achieved by organizing data in such a way that the percentage of accesses to each successively lower level is reduced, thus reducing the time it takes on average to access memory. The memory hierarchy includes various levels (`m1`, `m2`, ..., `mn`) where each member `mi` is typically smaller and faster than the next highest member `mi+1` of the hierarchy.



The Intel Xeon Skylake-SP processor, for example, uses a mostly non-inclusive memory hierarchy. In this architecture, the Last Level Cache (LLC) content is largely independent of the private caches content. This change in policy from inclusive to mostly non-inclusive may have allowed Intel to redistribute cache chip area, by enlarging private caches and diminishing LLC.

Cache Organization and Memory Access

Cache memory is a special, high-speed memory that stores copies of data from frequently used main memory locations. It reduces the average time to access data from the main memory. The performance of cache memory is frequently measured in terms of a quantity called Hit ratio, which is the number of hits divided by the total number of accesses. Cache mapping techniques determine which memory addresses in the main memory are mapped to which locations in the cache memory. Examples include direct-mapped cache, set-associative cache, and fully associative cache.



In the case of Intel's memory interfacing, data in a cache is generally stored in fast access hardware such as RAM and may also be used in correlation with a software component. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer.

Memory-Mapped I/O and Use of IN and OUT Instructions

Memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) are two complementary methods of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices in a computer.

Memory-mapped I/O (MMIO):

- Memory-mapped I/O uses the same address space to address both main memory and I/O devices.
- The memory and registers of the I/O devices are mapped to (associated with) address values, so a memory address may refer to either a portion of physical RAM or to memory and registers of the I/O device.
- Thus, the CPU instructions used to access the memory can also be used for accessing devices.
- Each I/O device either monitors the CPU's address bus and responds to any CPU access of an address assigned to that device, connecting the system bus to the desired device's hardware register or uses a dedicated method.

Port-mapped I/O (PMIO):

- Port-mapped I/O often uses a special class of CPU instructions designed specifically for performing I/O, such as the IN and OUT instructions found on microprocessors based on the x86 architecture.
- Different forms of these two instructions can copy one, two or four bytes (`outb`, `outw` and `outl`, respectively) between the `EAX` register or one of that register's subdivisions on the CPU and a specified I/O port address which is assigned to an I/O device.
- I/O devices have a separate address space from general memory, either accomplished by an extra `I/O` pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

Memory Organization and Addressing Modes

Memory organization refers to the way data is stored in a computer's memory. It involves the layout of the memory components and the way these components interact with one another to process instructions.

Addressing modes define how the instruction uses its operand or operands. For instance, the operand could be stored directly in the instruction, it could be in a register, or in a memory location that the instruction points to.

Handling Data Transfer Between CPU and Peripherals

Data transfer between the CPU and peripherals can be handled in various ways, including:

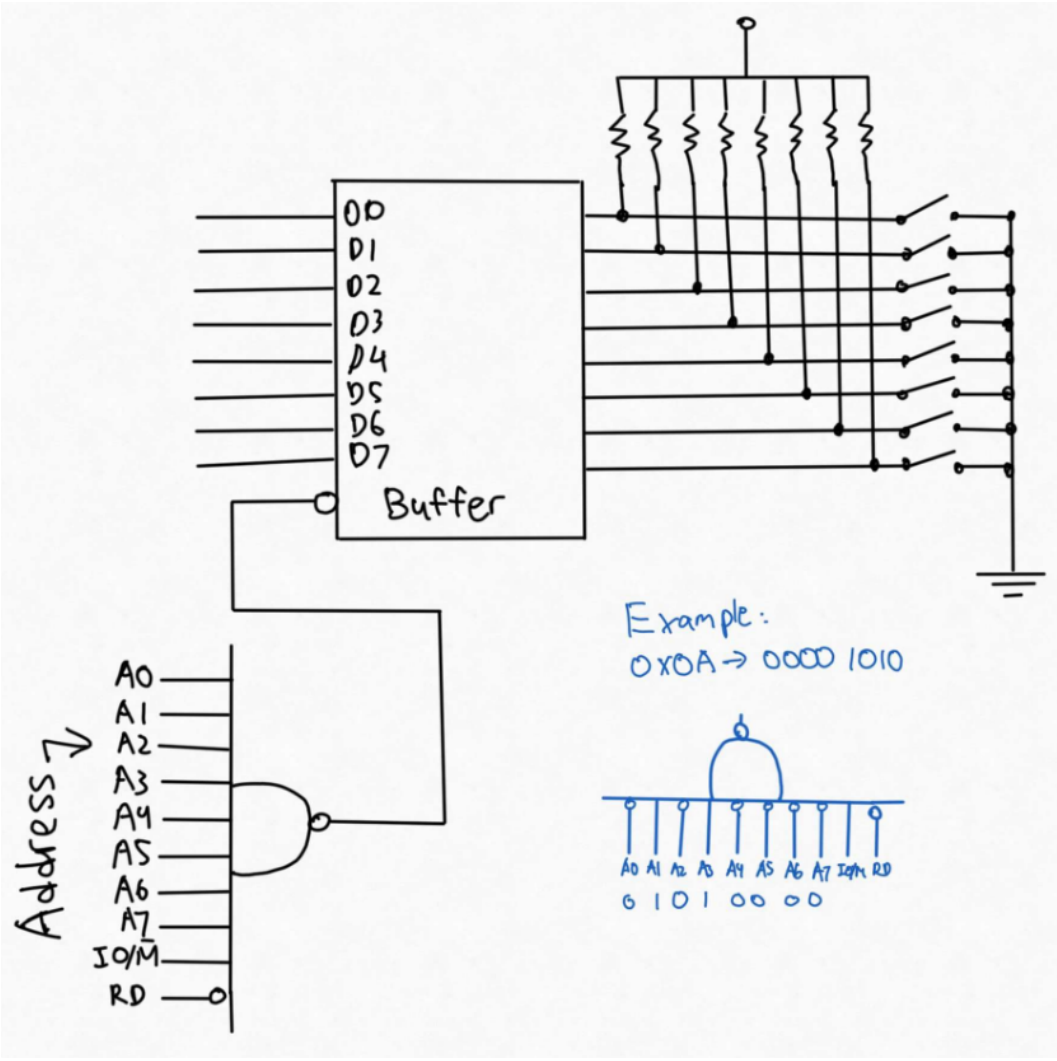
- **Programmed I/O:** The CPU is responsible for controlling all data transfer with the peripheral device. The CPU issues a command to the peripheral device to transfer data and then waits for the data transfer to complete.
- **Interrupt-driven I/O:** The CPU issues a command to the peripheral device to transfer data, but it does not wait for the data transfer to complete. Instead, the peripheral device sends an interrupt to the CPU when it has completed the data transfer.
- **Direct Memory Access (DMA):** A DMA controller takes over the control of the system bus from the CPU and transfers data directly between the peripheral device and memory. The CPU is only involved at the beginning and end of the transfer.

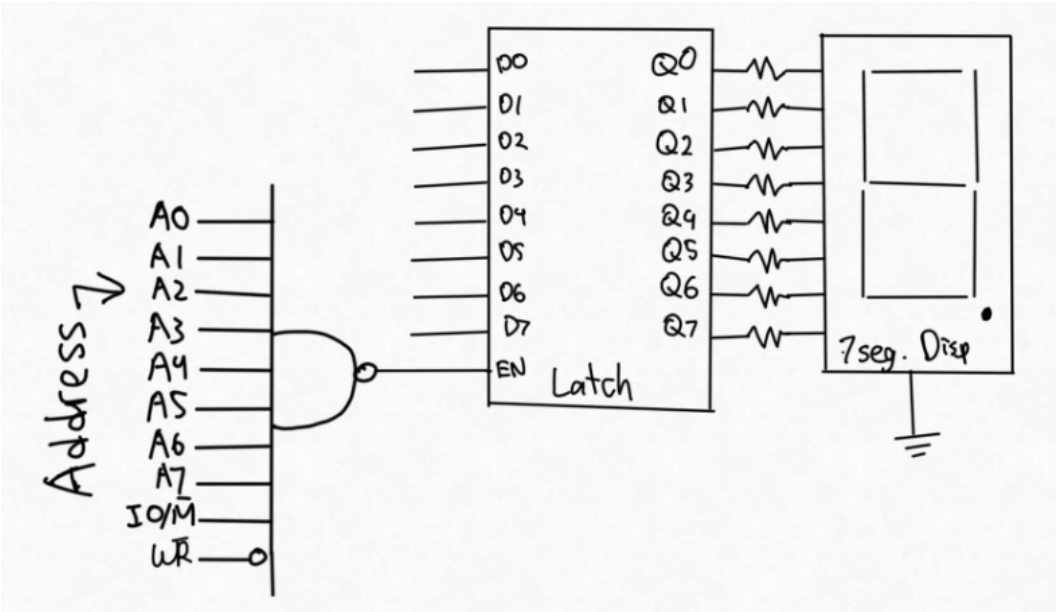
Basic Intel Instructions:

- The `MOV` instruction is fundamental, used to move data between register and memory locations.
- `ADD` and `SUB` instructions perform addition and subtraction operations.
- These basics instructions form the building blocks for more complex operations and program flow control.

IO Circuit

Design a IO circuit for the 8088 so that 8 switches (active low) are connected to a port
of address …. and a 7 segment LED is connected to another port of address … Draw the circuit showing how the address bus is being decoded and how the switches
and LEDs are to be connected to the data bus.





Chapter 4A: Intel Cache

Cache Memory Benefits and Organization

Cache memory is a very high-speed semiconductor computer memory used to speed up the central processing unit (CPU). It is primarily used to store and handle programs and applications which are more frequently used by processors, thereby enhancing the performance of the CPU. The response time for data exchange, access, and transfer is quite low, making cache memory an essential component for high performance in computer systems

Advantages of Cache Memory:

- Cache memory is faster than main memory due to the use of Static Random Access Memory (SRAM) as opposed to Dynamic Random Access Memory (DRAM) used in main memory.
- It stores instructions that may be required by the processor next time, allowing faster recovery of information compared to Random Access Memory (RAM).
- Cache memory stores frequently used data and instructions, thereby increasing the performance of the CPU

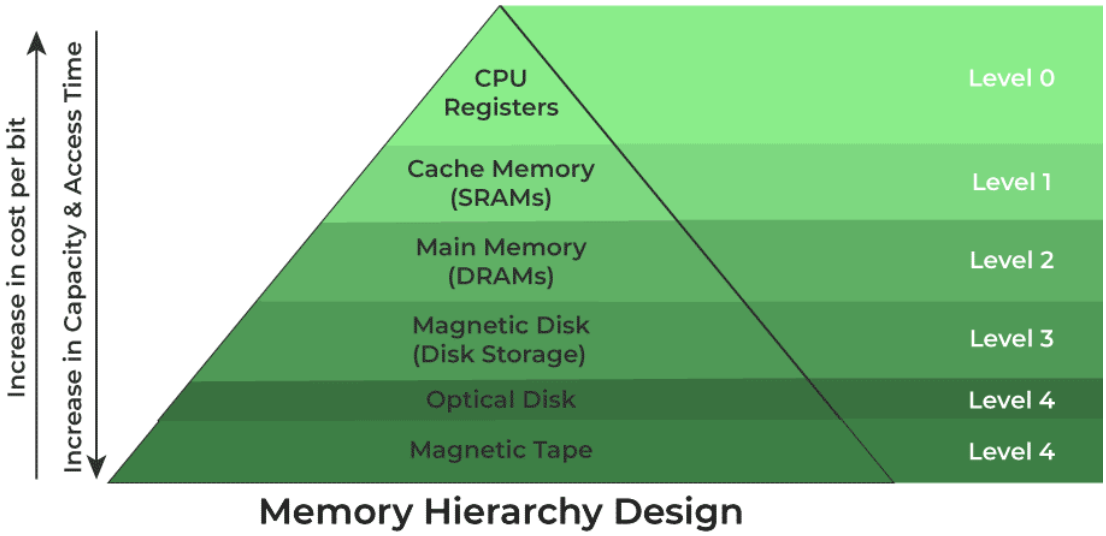
Cache Line Size and its Impact

Cache line size refers to the block of memory that is transferred to a memory cache. The cache line size is generally fixed, typically ranging from 16 to 256 bytes. The effectiveness of the line size depends on the application, and cache circuits may be configurable to a different line size by the system designer

The choice of cache line size involves trade-offs. Larger cache lines can result in more memory traffic for random access and more false sharing contention between parallel caches. Moreover, if different CPUs, each with its own cache, are accessing memory on the same cache line, that line will have to "bounce" back and forth between the caches. However, these problems can be avoided by tuning the software to want memory in chunks that are multiples of the cache line size.

Cache Memory Hierarchy

The memory hierarchy typically includes registers, cache memory, main memory, and secondary storage. Cache memory is positioned between the CPU and the main memory. It is usually divided into different levels, L1, L2, and L3, with L1 being the closest to the CPU and L3 being the farthest. L1 cache is the fastest but has the smallest capacity, while L3 cache is slower but has a larger capacity.



Memory Hierarchy Design

Principle of Locality

Cache is useful as the accesses maintain a locality of reference in a small range of memory.

- Temporal Locality: when a data is used it is likely that we will use the same data again.
- Spatial Locality: when a data is used, there is a high probability that the other data nearby will be needed soon.

Cache Miss and Hit

Hit means have in cache, miss means don’t have in cache.

Type of Cache Miss:

- Capacity Miss: Cache is not large enough to store all the needed values.
- Conflict Miss: Needed value was replaced during another recent miss.
- Cold miss: Memory location is being accessed for the first time.

Cache Mapping Techniques

Cache mapping techniques determine how the main memory blocks are loaded into the cache. There are three main types of cache mapping techniques:

- **Direct-mapped**: In this technique, each block of main memory maps to exactly one cache line. This technique is simple and inexpensive but can lead to a high miss rate if multiple memory blocks map to the same cache line.
- **Set-associative**: In this method, each block of main memory can map to any line in a given set of cache lines. This technique reduces the miss rate compared

to direct mapping but is more complex and expensive.

- **Fully-associative**: In this technique, each block of main memory can map to any line in the cache. This method has the lowest miss rate but is the most complex and expensive.

Cache Coherence and Performance Improvement

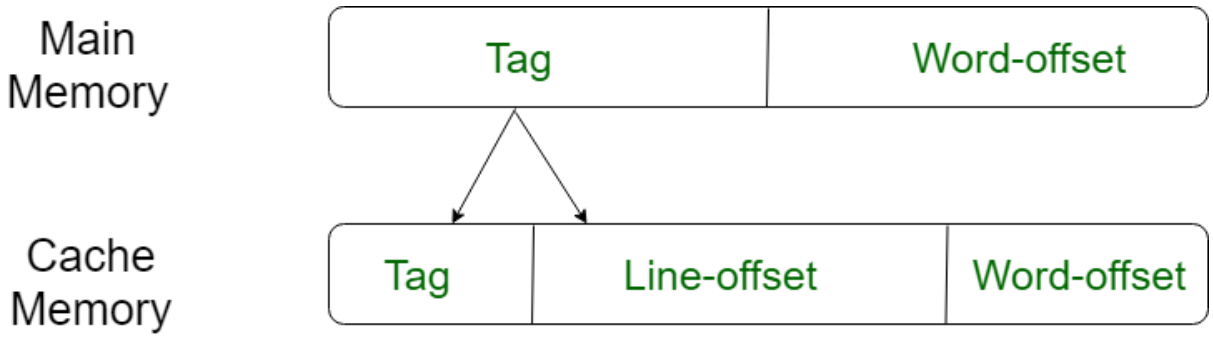
Cache coherence refers to the consistency of shared resource data in multiprocessor systems. It ensures that changes in the value of shared operands are propagated throughout the system in a timely fashion. Maintaining cache coherence is important for the correctness of a program and is typically handled through various coherence protocols.

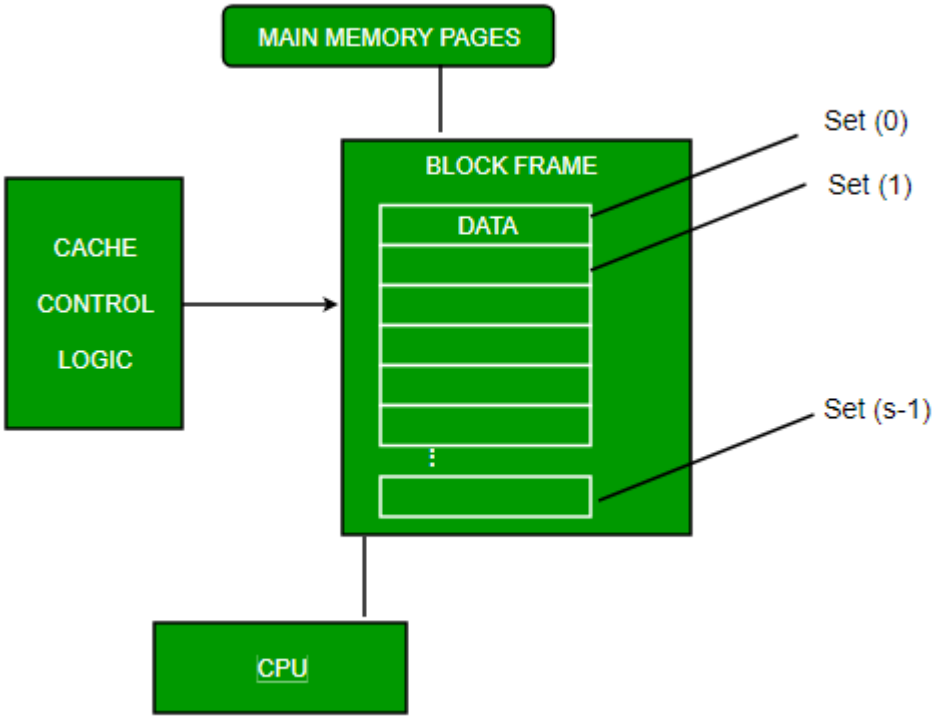
Improving cache performance can be achieved through various techniques such as increasing cache size, optimizing cache replacement and prefetching policies, and using cache compression. Prefetching and compression can have an interesting synergy: bad prefetches can bring blocks that end up polluting the cache; however, in a compressed cache these blocks may co-allocate with useful blocks, reducing the drawbacks of those bad prefetches.

Cache Mapping Techniques

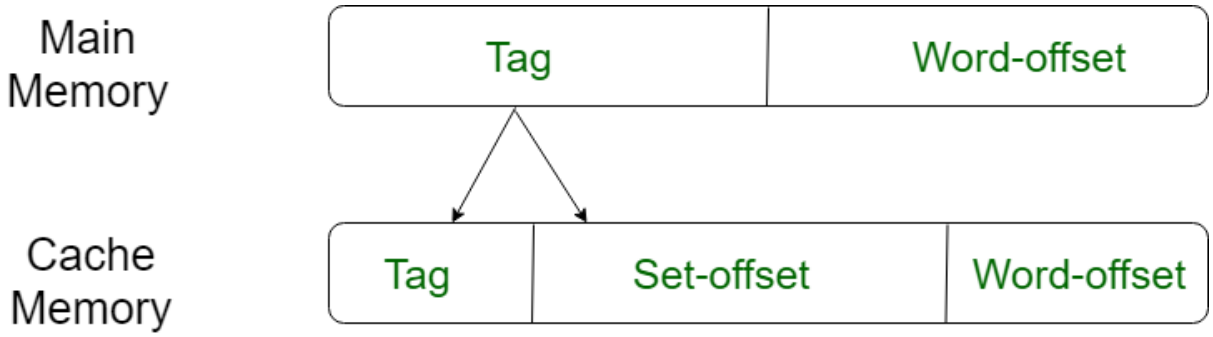
Cache mapping techniques determine how the main memory blocks are loaded into the cache. There are three main types of cache mapping techniques:

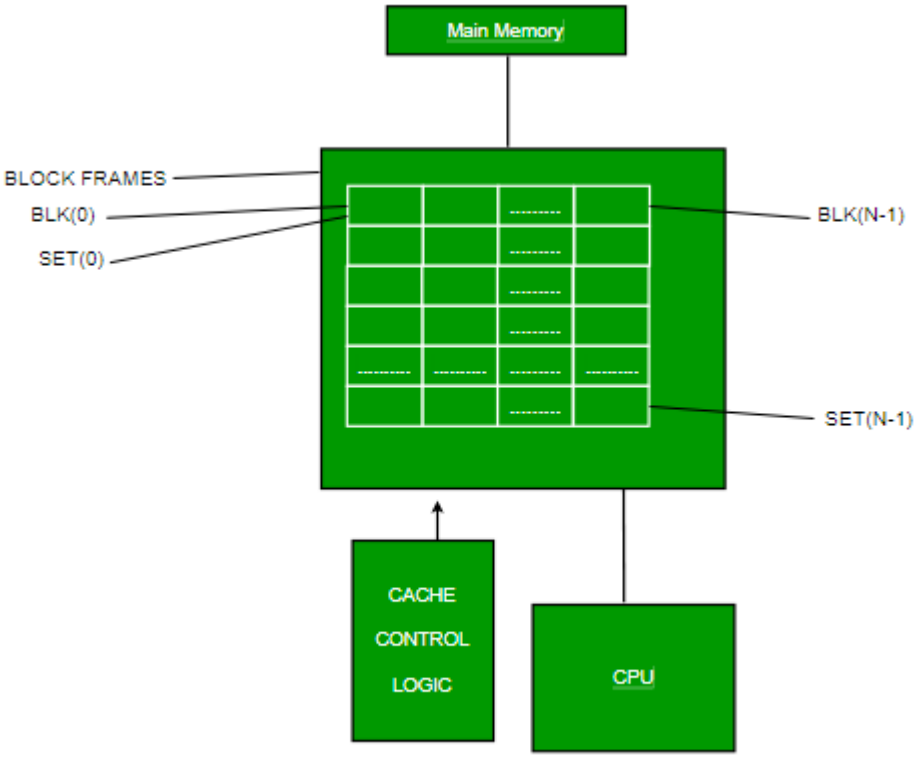
- **Direct Mapped Cache**: In this technique, each memory block can only be mapped to one specific block in the cache. This means there is a one-to-one correspondence between memory and cache blocks. The cache location is determined by using the lower-order bits of the memory address. While this method is simple and easy to implement, it can lead to high levels of conflict misses when multiple memory blocks map to the same cache block.



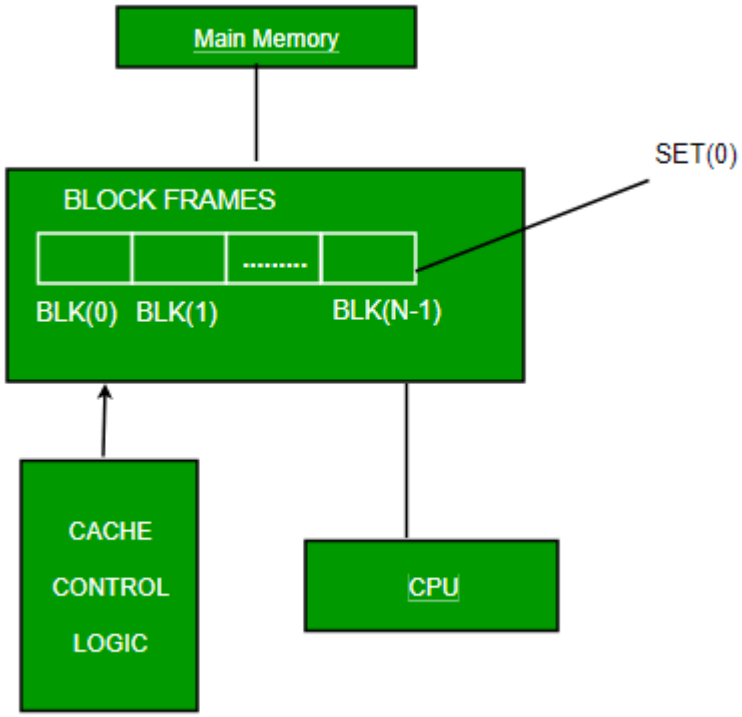


- **Set-Associative Mapping**: This cache mapping technique eliminates the limitation of direct mapping by mapping multiple lines together instead of mapping each line one by one. The entire memory block can be mapped to any of the cache lines. It's essentially a hybrid of direct and fully associative mapping techniques, providing a balance between them.





- **Fully Associative Mapping**: In this technique, each memory block can be mapped to any line in the cache. This allows for greater flexibility but also increases complexity and cost.



Direct Mapped Cache

Direct Mapped Cache is the simplest form of cache mapping where each block of main memory can be mapped to only one specific cache location. The cache is organized into multiple sets with a single cache line per set. Based on the address of the memory block, it can only occupy a single cache line. The cache can be framed as an `n × 1` column matrix.

The working process of the direct mapped cache involves a read admittance to the cache. This takes the central portion of the address and is called the index. It is usually used as the row number. At the same time, both the tag and the data are also looked up. In the next step, the tag and the upper part of the address are compared in order to find out whether or not the line is valid and whether or not it comes from the same address range in the memory.

Some advantages of Direct-Mapped Cache include:

- Simplicity in implementation
- Lower power consumption
- Constant and deterministic access time
- Straightforward replacement procedure
- Cost-effectiveness.

However, the direct-mapped cache may not be suitable for all memory access patterns and workloads due to its higher conflict and miss rates and limited associativity.

Set-Associative Cache

Set-Associative Cache is a trade-off between direct-mapped and fully associative cache. The cache is divided into 'n' sets and each set contains 'm' cache lines. A memory block is first mapped onto a set and then placed into any cache line of the set. The range of caches from direct-mapped to fully associative is a continuum of levels of set associativity.

Set-associative cache can be anywhere from 2 sets to eight sets wide. Data is stored in them all, but the cache distributes it to each set in sequence, rather than randomly. In most cases, data from each set is also read sequentially, speeding up the reading process just a little.

The advantages of Set-Associative Cache include:

- Flexibility of placing memory block in any of the cache lines within a set

- Flexibility of using replacement algorithms if a cache miss occurs.

Fully Associative Cache

In a Fully Associative Cache, the cache is organized into a single cache set with multiple cache lines. A memory block can occupy any of the cache lines. The cache organization can be framed as 1 × m row matrix.

Advantages of Fully Associative Cache include:

- Full utilization of the cache as it provides the flexibility of placing memory block in any of the cache lines
- It offers the flexibility of utilizing a wide variety of replacement algorithms if a cache miss occurs.

However, the fully associative cache is more complex and costly compared to the other two cache mapping techniques.

Calculating Index:

$$2^{index} = \frac{CacheSize}{LineSize \times SetAssociativity}$$

Example:

A Direct Mapped cache has the following specifications :

Main Memory: 4GB - 2^{32}

Cache Size: 2KB - 2^{11}

Cache Line Size: 8B - 2^{3}

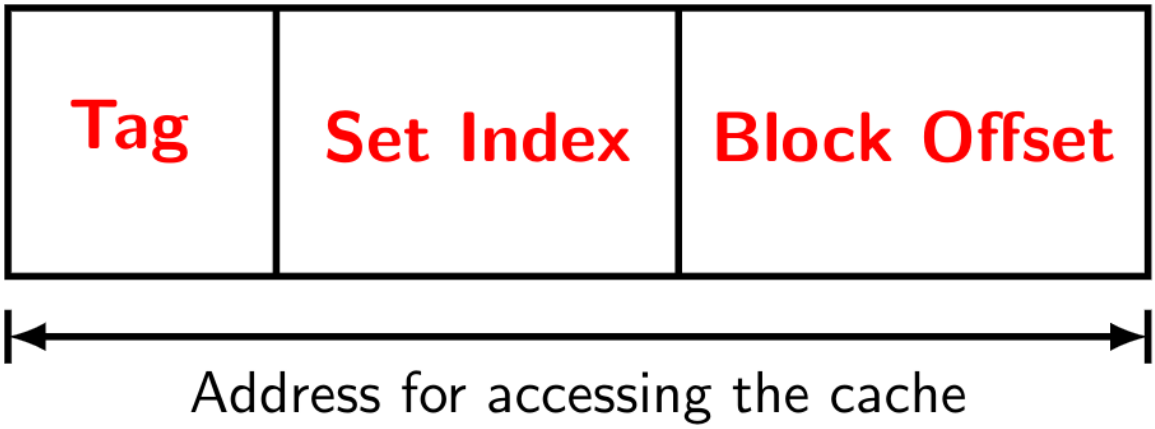
Show how the address bus will be subdivided into the tag, index and byte offset. Show
clearly the number of bits required for each section.
Use a diagram to show how a byte with address **0000000AH** and data `055h` and another
byte with address **00000011H** and data `0AAh` will be stored in this cache. Your diagram should show clearly the index and offset.

$$2^{index} = \frac{2^{11}}{2^{3} \times 1} = 2^{8}$$

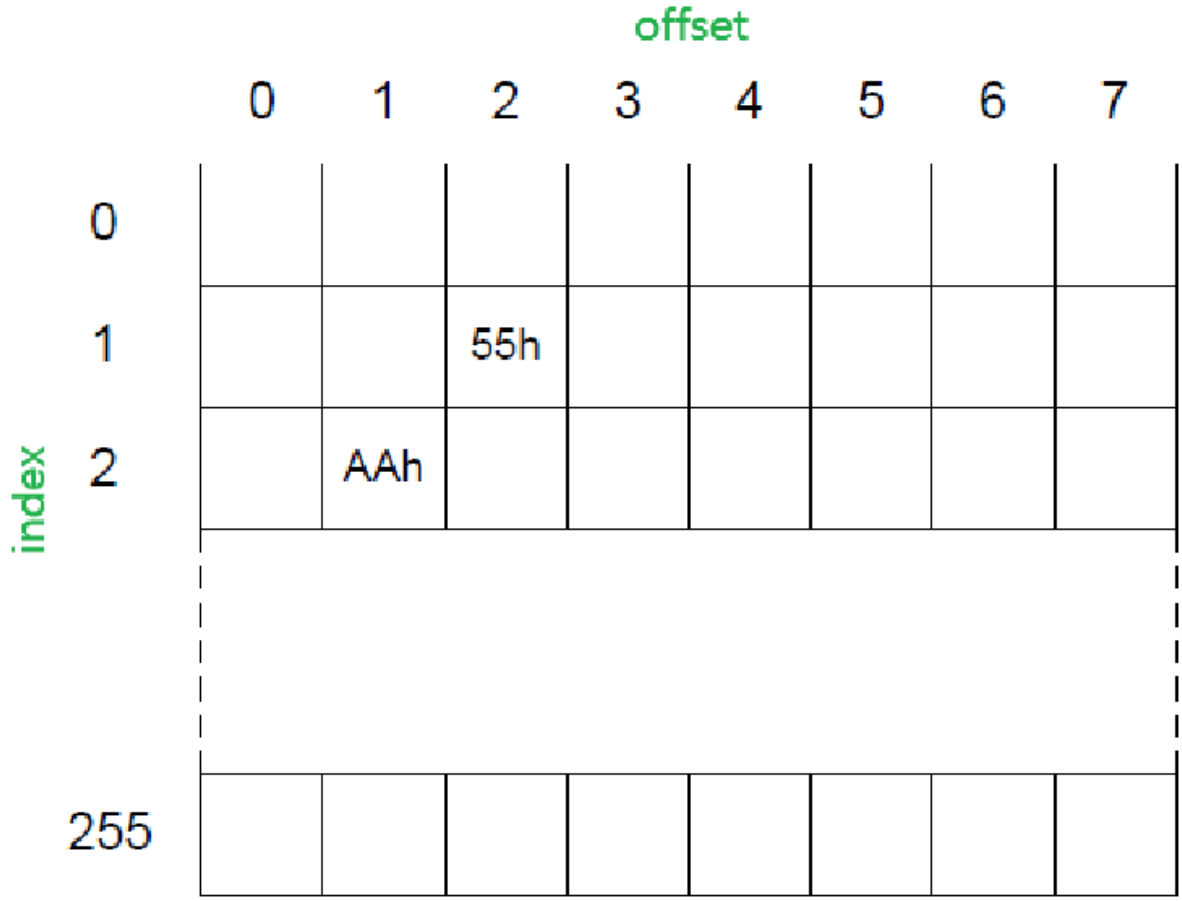
Index: 8 bits

Offset: 3 bits (from the line size)

Tag: $32 - 8 - 3 = 21$



0000000AH → … … ***0000 1***010 so **index is 1** and offset is 2 will contain `055H`
00000011H → … … ***0001 0***001 so ***index is 2*** and offset is 1 will contain `0AAH`

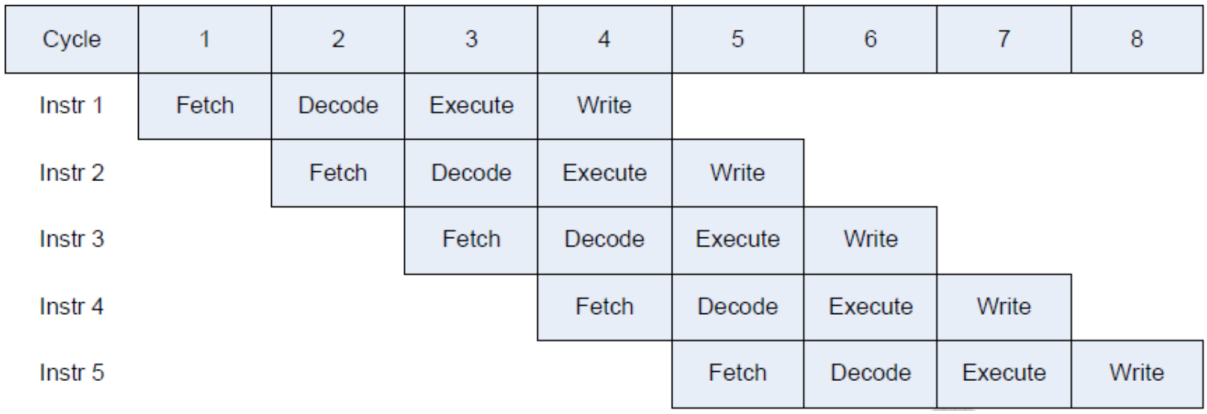


index has 256 bits from 2^8, offset has 8 bits from 2^3

Chapter 4B: Intel Pipeline

The "Intel Pipeline" module continues to delve into the realm of pipelining, offering an even deeper understanding of its architecture, stages, hazards, and advanced concepts such as branch prediction and forwarding. This module is designed to provide you with a comprehensive grasp of how pipelines function, the challenges they pose, and the strategies employed to overcome those challenges.

This section delves into the architecture and stages of a pipeline, addressing the various types of hazards that can hinder its smooth operation. It also introduces the concept of instruction and data forwarding, pivotal techniques that maintain pipeline efficiency.



Pipeline Architecture and Stages:

- A pipeline is a sequence of stages through which instructions progress.
- Stages include instruction fetch, decode, execute, memory access, and write-back.
- Pipelining enhances processor throughput by allowing multiple instructions to be in various stages simultaneously.

Hazards and Solutions:

- **Data Hazards:** Arise when an instruction depends on the result of a previous instruction. They can lead to incorrect results.
    - **Solution:** Techniques like data forwarding or stall (waiting) can address data hazards.

- **Control Hazards:** Emerge due to branch instructions, which alter program flow. They can lead to pipeline flushing and reduced efficiency.
    - **Solution:** Branch prediction and delayed branching are used to mitigate control hazards.
- **Structural Hazards:**
    - **Bus:** When different stages of the pipeline wants to use the data bus, it will cause a structural hazard. During the 4th cycle, the instruction 1 is writing back to memory while the CPU is also trying to fetch instruction 4. Since both cannot be using the data bus at the same the instruction 4 will be delayed
    - **Execute:** `DIV` instruction will take many cycles. if it takes 6 cycles the next instruction will have to wait.

Instruction and Data Forwarding:

- **Instruction Forwarding:** Also known as register forwarding, it enables the result of an instruction in one pipeline stage to be directly forwarded to another stage, preventing data hazards.
- **Data Forwarding:** Data from a previous instruction is forwarded to a subsequent stage before it's written back to a register, addressing data hazards.

Branch Prediction and Its Purpose:

- **Branch Prediction:** Predicts the outcome of branch instructions before they are resolved.
- **Purpose:** To improve pipeline efficiency by minimizing the impact of control hazards caused by conditional branches.
- Branch Prediction is used by the CPU to predict in advance on the possible outcome of a branch instruction. This will enable the pipeline to be full most of the time.
  Branch Prediction can be either static or dynamic. In static prediction, the instructions are being analyzed before being executed. In dynamic prediction, the prediction is done using a one or two bits state machine with is updated when the program is being executed.
- An instruction buffer is used to hold the instructions before passing it to the pipeline. A branch unit will check if the next instruction is a branch. If it is a branch, both possible instructions will be fetched. Either part of the instructions

will be ready for execution without any extra waiting time.

Chapter 5: Intel Pentium

This module will cover the Intel Pentium processor, its ability to execute multiple instructions, and the use of instruction-level parallelism. We will discuss the features and advancements in Intel Pentium processors, and delve into the concepts of superscalar architecture, branch prediction, speculative execution, and out-of-order execution.

This chapter dives into the intricacies of Intel Pentium processors, showcasing their advancements and contributions to modern computing architectures. It introduces the concept of superscalar architecture and explores sophisticated techniques that boost performance by predicting and executing instructions optimally.

Multiple instruction execution and Instruction-level parallelism in Intel Pentium

Instruction-level parallelism (ILP) is the simultaneous execution of a sequence of instructions in a computer program. In the context of the Intel Pentium processor, ILP is exploited to increase the performance of the processor. Intel Pentium processors are equipped with multiple processing units that allow them to handle several instructions in parallel in each processing stage. This ability to execute multiple instructions simultaneously gives Intel Pentium processors an execution throughput of more than one instruction per cycle, making them superscalar processors.

Features and Advancements in Intel Pentium Processors:

- Intel Pentium processors represent a leap in computational power, efficiency, and performance.
- Advancements include increased clock speeds, improved instruction sets, enhanced cache designs, and optimized microarchitecture.
- The Intel Pentium series introduced the use of superscalar techniques, enabling multiple instructions to be executed in parallel. The Pentium Pro further enhanced this by introducing features like register renaming, branch prediction, data flow analysis, speculative execution, and more pipeline stages. Advanced optimization techniques in microcode were also added along with a level 2 cache. The Pentium II incorporated Intel MMX technology for efficient processing of video, audio, and graphics data. The Pentium III added streaming extensions (SSE) for supporting 3D graphics software. The Pentium 4 implemented third-

generation address translation and other floating-point enhancements for multimedia.

Superscalar Architecture:

- Superscalar architecture enables processors to execute multiple instructions in parallel during a single clock cycle.
- It incorporates multiple execution units that work simultaneously, optimizing instruction-level parallelism.
- In a superscalar processor like the Intel Pentium, several instructions start execution in the same clock cycle. This is achieved by equipping the processor with multiple processing units to handle several instructions in parallel in each processing stage. The compiler plays a crucial role in avoiding many hazards through the judicious selection and ordering of instructions. However, the detrimental effect on performance of various hazards becomes even more pronounced in a superscalar architecture.

Branch Prediction, Speculative Execution, and Out-of-Order Execution:

- **Branch Prediction:** Techniques that predict the outcome of branch instructions before they are resolved, used in processors to guess the outcome of a conditional operation before the condition is resolved. If the prediction is correct, the processor can continue executing instructions without having to wait. However, if the prediction is incorrect, the processor has to discard the speculative work and restart from the branch. The Pentium Pro introduced the use of branch prediction.
- **Speculative Execution:** Executes instructions speculatively based on predictions to maintain pipeline efficiency, used in processors to execute instructions before it is known whether these instructions should be executed or not. This can increase performance as the processor can keep executing instructions without having to wait for previous instructions to complete. However, speculative execution can lead to security vulnerabilities, as was seen in the Meltdown and Specter vulnerabilities.
- **Out-of-Order Execution:** Executes instructions in an order that maximizes utilization of execution units, improving overall performance, used in processors to execute instructions not in the order they appear in the program but in an order that is determined by the availability of input data and execution units. This can increase performance by allowing the processor to keep executing instructions without having to wait for previous instructions to complete.

Chapter 6: RISC

In this module, we will explore the Reduced Instruction Set Computer (RISC) architecture, its comparison to Complex Instruction Set Computer (CISC) architecture, the advantages of RISC CPUs, and compiler optimizations for RISC processors.

This chapter delves into the core of RISC architectures, shedding light on their distinguishing characteristics and advantages over traditional CISC architectures. The chapter's content explores the intrinsic features of RISC CPUs and the role of compiler optimizations in harnessing their power.

RISC vs. CISC Architecture:

- **RISC (Reduced Instruction Set Computing):** Emphasizes a simple, streamlined set of instructions that execute in a single clock cycle.
- **CISC (Complex Instruction Set Computing):** Features a complex instruction set with variable execution times for instructions.

CISC devices take more time to execute a single operation, so the architecture doesn't support the parallel processing and pipelining of instructions. On the other hand, in RISC architecture, every instruction is completed in one clock cycle, promoting the pipelining of instructions.

CISC architecture tries to boost CPU performance by finishing a task in fewer lines of assembly code, while RISC focuses on reducing the execution time of each instruction. For example, a CISC device like an Intel 8086 processor might take around 70 to 77 clock cycles to perform multiple operations on two 8-bit numbers, while a RISC device like a PIC microcontroller may only take up to 38 clock cycles. As a result, the RISC device is twice as fast as its CISC counterpart.

Advantages of RISC CPUs:

- RISC CPUs boast several advantages, including improved performance, simplified instruction pipeline design, and efficient use of hardware resources.
- They provide the use of space on microprocessors and energy saving, making them ideal for devices like cell phones.
- They allow high-level language compilers to create more efficient code due to their set of instructions.
- They utilize registers to pass parameters and store local parameters.
- Their instructions use limited arguments, making them easy to pipeline.

- They can increase operation speed while reducing execution time.

However, it's worth noting that RISC CPUs require very fast memory systems to feed various instructions, thus a large memory cache is required.

RISC Features:

- **Simplified Instruction Set:** RISC CPUs prioritize a smaller set of instructions, allowing for faster execution and efficient pipelining.
- **Load-Store Architecture:** RISC architectures rely on load and store instructions to access memory, reducing memory access times.

  The load-store architecture means that instructions that accessed the memory are limited mainly to the move (MOV) instructions only. Arithmetic instructions will not be able to directly access the memory.
- The RISC architecture is characterized by its simplicity, which allows for efficient use of space on microprocessors and energy saving. This makes it ideal for use in devices like cell phones (ARM processors).
- RISC processors utilize registers to pass parameters and store local parameters. They use a limited number of arguments, which allows them to use a fixed-length instruction, making the instructions easy to pipeline. This architecture can increase operation speed while reducing execution time.

Compiler Optimizations for RISC Processors:

- RISC architectures depend on compilers to optimize code for execution efficiency.
- Compiler optimizations include instruction scheduling, loop unrolling, and instruction combining.
- Examples:
  - Compiler can re-arrange branches and loops so they are more efficient.
  - Compiler can assign variables to registers.
  - Simpler and fewer instructions means it is easier to design the compiler.
  - Copy Propagation
- The performance of RISC processors can depend significantly on the compiler. The compiler plays a crucial role when translating the CISC code to RISC code.
- RISC architecture allows high-level language compilers to produce more efficient code due to its set of instructions. However, the performance of RISC

processors can depend on the compiler or the programmer, as some instructions might rely on the previous instruction to finish their execution.

- The RISC instruction set requires one to write more efficient software (e.g., compilers or code) with fewer instructions. This need for efficient software makes RISC a more complex compiler design compared to CISC.

Difference between Von Neumann and Harvard architecture

The major difference between the two architectures is that in a Von Neumann architecture all memory is capable of storing all program elements, data and instructions; in a Harvard architecture the memory is divided into two memories, one for data and one for instructions.

Von Neumann Architecture:
Has no difference between the program and the data memories.
Advantages: Greatly simplifies the memory interface.
Disadvantages: Capacity between memory and the CPU is limited, one can increase the capacity of this channel by making it wider or faster, but the CPU is limited in its ability to access the memory.
Harvard Architecture:
Has instructions and data stored in separate memories.
Advantages: Able to handle different word lengths and formats of code and data. Provides some level of security by not allowing execution of data (access right needed), and write inadvertently over the program codes (to protect memory zone).
Disadvantages: Memory interfacing is more complicated.