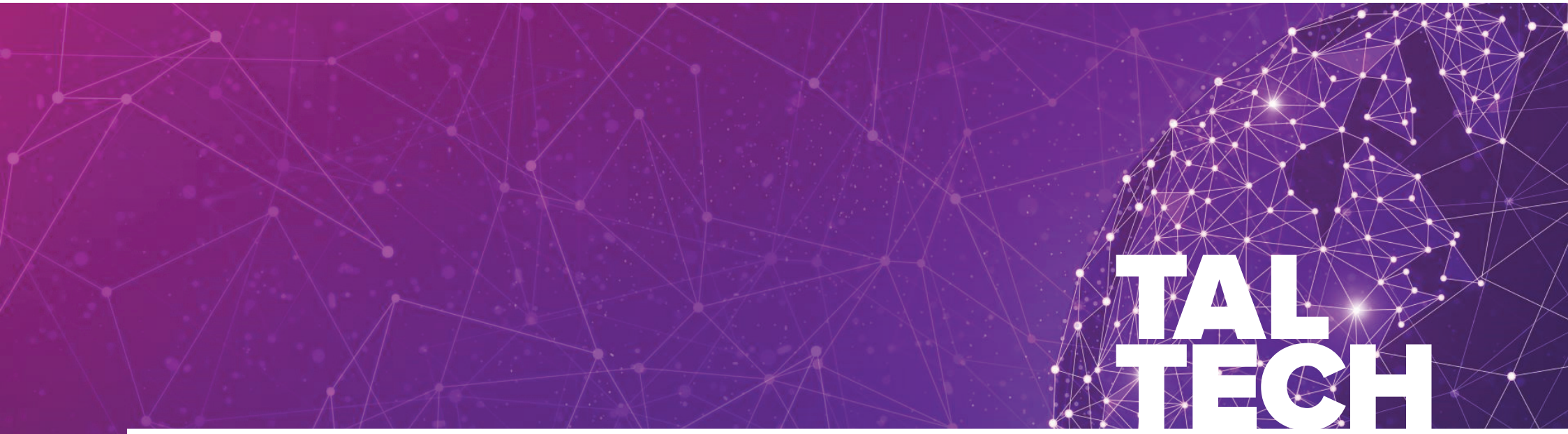


MICROPROCESSOR SYSTEMS (IAS0430)

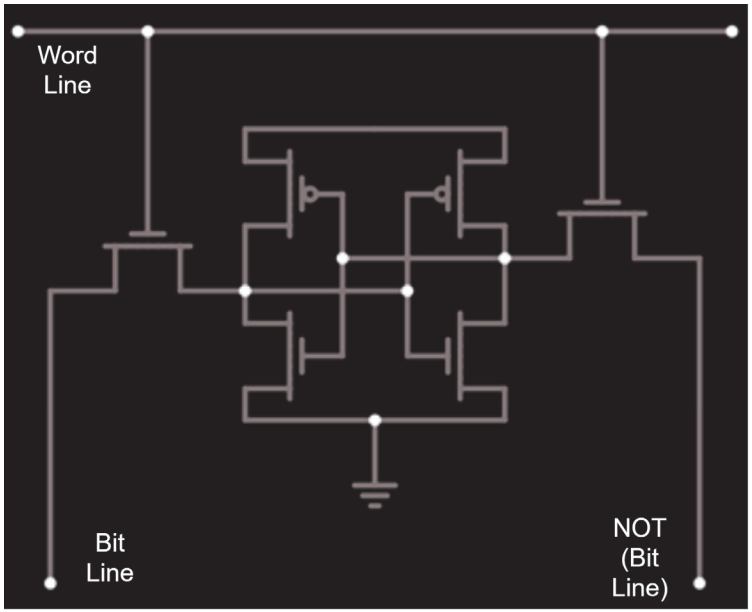
Department of Computer Systems
Tallinn University of Technology

29.10.2021

MEMORY HIERARCHY

- **What is computer memory?**
  - Computer memory is any **physical device designed using integrated circuits** for the purpose of **storing and retrieving data and/or instructions** for short term use (**temporarily**) or long term storage (**permanently**).

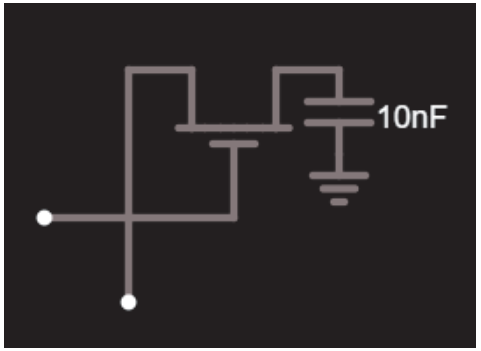
  - **Memory Types:**
    - **Bi-Polar memory:**
      - **SRAM – Static RAM**
      - DRAM – Dynamic RAM
    - Magnetic Memory
    - Optical Memory
  - **Memory Devices**
    - Register File



MEMORY HIERARCHY

- **What is computer memory?**
  - Computer memory is any **physical device designed using integrated circuits** for the purpose of **storing and retrieving data and/or instructions** for short term use (**temporarily**) or long term storage (**permanently**).

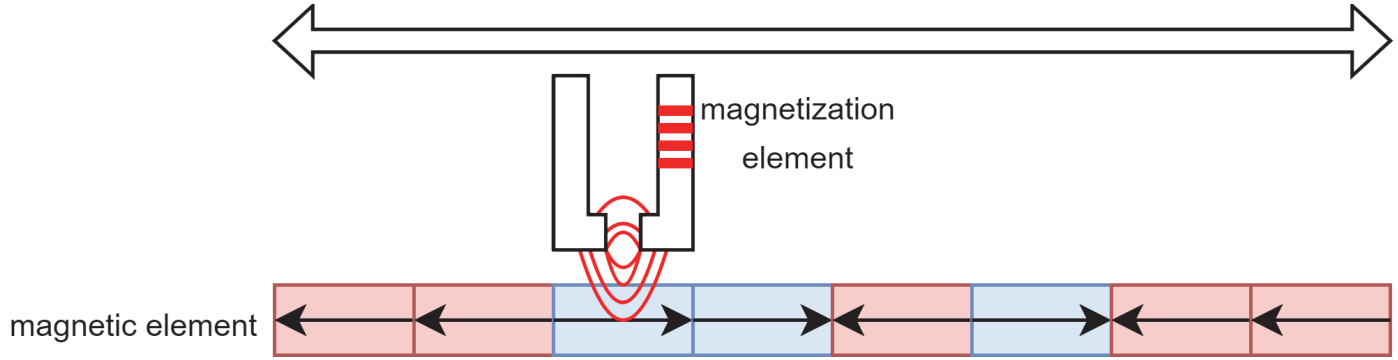
  - **Memory Types:**
    - **Bi-Polar memory:**
      - SRAM – Static RAM
      - **DRAM – Dynamic RAM**
    - Magnetic Memory
    - Optical Memory
  - **Memory Devices**
    - Register File



MEMORY HIERARCHY

- **What is computer memory?**
  - Computer memory is any **physical device designed using integrated circuits** for the purpose of **storing and retrieving data and/or instructions** for short term use (**temporarily**) or long term storage (**permanently**).

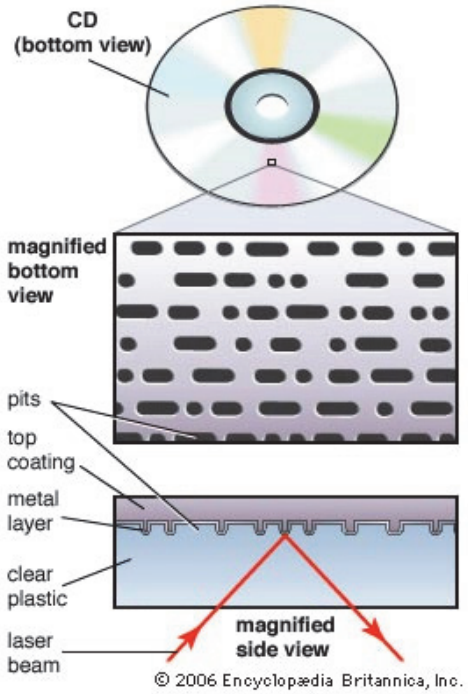
  - **Memory Types:**
    - Bi-Polar memory**:**
      - SRAM – Static RAM
      - DRAM – Dynamic RAM
    - **Magnetic Memory**
    - Optical Memory
  - **Memory Devices**
    - Register File



MEMORY HIERARCHY

- **What is computer memory?**
  - Computer memory is any **physical device designed using integrated circuits** for the purpose of **storing and retrieving data and/or instructions** for short term use (**temporarily**) or long term storage (**permanently**).

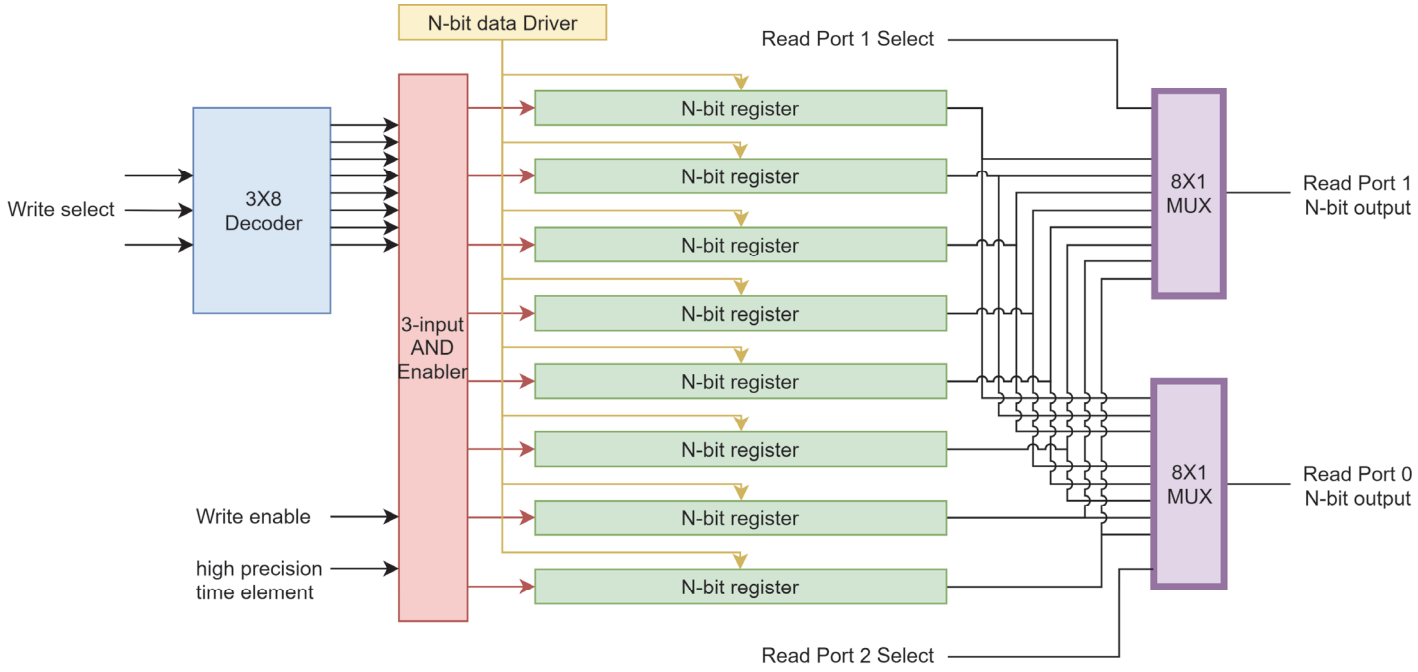
  - **Memory Types:**
    - Bi-Polar memory**:**
      - SRAM – Static RAM
      - DRAM – Dynamic RAM
    - Magnetic Memory
    - **Optical Memory**
  - **Memory Devices**
    - Register File



MEMORY HIERARCHY

- **What is computer memory?**
  - Computer memory is any **physical device designed using integrated circuits** for the purpose of **storing and retrieving data and/or instructions** for short term use (**temporarily**) or long term storage (**permanently**).

- **Memory Types:**
  - SRAM – Static RAM
  - DRAM – Dynamic RAM
  - Magnetic Memory
  - Optical Memory
- **Memory Devices**
  - **Register File**



MEMORY HIERARCHY

- **Caches recap**
  - The cache is a **static random access memory designed to operate and to be accesses by the CPU at a much faster rates than the Main Memory**
  - It is located right inside the CPU.
  - It bridges the speed gap between the Main memory and the CPU.
    - If data is found in the cache, this is called a cache **hit**
    - If data is not found in the cache, this is called a cache **Read miss**.
    - If data is being written into the cache in a location where data is already stored, this is called a cache **Write miss**. – we will get to see more types of misses later.
    - If a cache miss occurs, **the data is then requested from the slower main memory**

MEMORY HIERARCHY

- **Caches recap**
- The principle of **cache locality**:
  - **Spatial locality:**
    - When an address is accessed, it **is highly likely that the CPU wants to access the sequential memory locations located after that address as well**.
  - **Temporal locality**:
    - When an address is accessed**, it is highly likely that the CPU wants to access the that address again later**.

  - **Locality** can help **determine how the cache should be accessed** and what **methods to use in cache replacement policies** for different programs– remember this for later on…

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing **refers to the way in which a cache address is divided**.
  - Caches stores information as chunks of data called **blocks**.
    - A block of data is a collection of **words** that are stored one after the other.
    - The size of a block is largely determined when the cache is designed as part of the whole system.
      - It is always measured in Bytes
    - Meaning that the block size of the cache must correspond to other system design decisions.
  - Let us assume that we have 32-bit long data. If we have 8 words inside a block.
    - **What is the size of a block in Bytes?**

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing **refers to the way in which a cache address is divided**.
  - Caches stores information as chunks of data called **blocks**.
    - A block of data is a collection of **words** that are stored one after the other.
    - The size of a block is largely determined when the cache is designed as part of the whole system.
      - It is always measured in Bytes
    - Meaning that the block size of the cache must correspond to other system design decisions.
  - Let us assume that we have 32-bit long data. If we have 8 words inside a block.
    - **What is the size of a block in Bytes?**
    - We multiply the size of each word, times the total number of words inside the block
      - Words are 32-bit which is 4 bytes
      - So the size of the block is 4 * 8 is 32 Bytes

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing **refers to the way in which a cache address is divided**.
  - Caches stores information as chunks of data called **blocks**.
    - A block of data is a collection of **words** that are stored one after the other.
    - The size of a block is largely determined when the cache is designed as part of the whole system.
      - It is always measured in Bytes
    - Meaning that the block size of the cache must correspond to other system design decisions.
  - Let us assume that we have 32-bit long data. If we have 8 words inside a block.
    - **How do we know where the data is?**

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing **refers to the way in which a cache address is divided**.
  - Caches stores information as chunks of data called **blocks**.
    - A block of data is a collection of **words** that are stored one after the other.
    - The size of a block is largely determined when the cache is designed as part of the whole system.
      - It is always measured in Bytes
    - Meaning that the block size of the cache must correspond to other system design decisions.
  - Let us assume that we have 32-bit long data. If we have 8 words inside a block.
    - **How do we know where the data is?**
      - Since we have 8 words inside the block, we will need 3 bits to locate data inside it, those are the location bits – these bits are called the **offset bits**

8-word, 32-Byte cache block	
offset	data
000	32-bit long word
001	32-bit long word
010	01000010101110101010101000001010
...	.....
111	10110110001010101010101000111101

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing refers to the way in which a cache address is divided.
  - But, where do we find a block inside the cache?
    - Since the cache is very small memory, we need to use it in the most efficient way
    - We divide the cache into areas where blocks can be stored
      - These areas of the cache are called **sets**
    - Lets take the block from the pervious slide:
      - **How many blocks of 32-byte size can we fit inside a 128-Byte cache?**

MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing refers to the way in which a cache address is divided.
  - But, where do we find a block inside the cache?
    - Since the cache is very small memory, we need to use it in the most efficient way
    - We divide the cache into areas where blocks can be stored
      - These areas of the cache are called **sets**
    - Lets take the block from the pervious slide:
      - **How many blocks of 32-byte size can we fit inside a 128-Byte cache?**
        - We divide the size of the cache by the size of the block
        - 128 / 32 = 4 blocks can be stored inside this cache

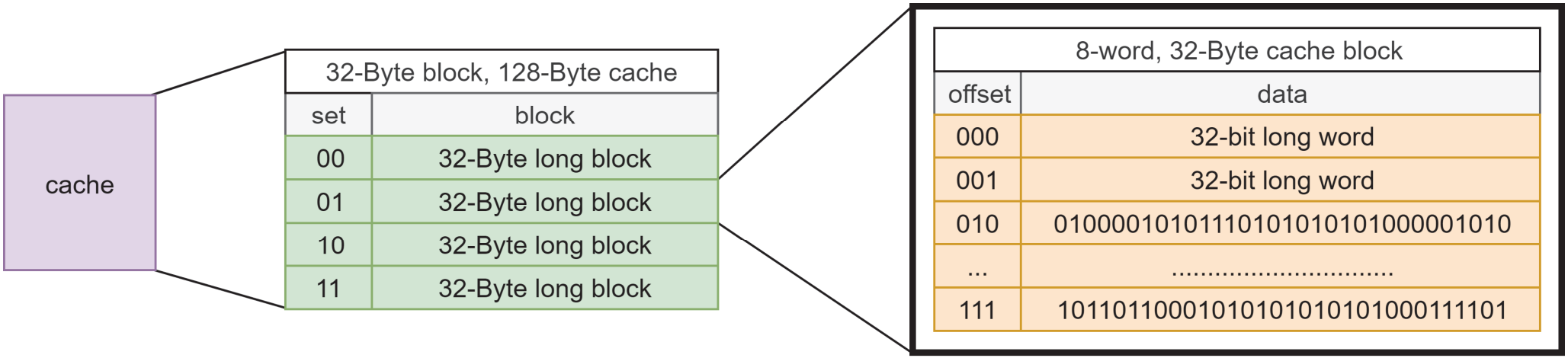
MEMORY HIERARCHY

- **Cache Addressing**
  - Cache addressing refers to the way in which a cache address is divided.
  - But, where do we find a block inside the cache?
    - Since the cache is very small memory, we need to use it in the most efficient way
    - We divide the cache into areas where blocks can be stored
      - These areas of the cache are called **sets**
    - Lets take the block from the pervious slide:
      - **How many blocks of 32-byte size can we fit inside a 128-Byte cache?**
        - We divide the size of the cache by the size of the block
        - 128 / 32 = 4 blocks can be stored inside this cache
      - **How do we know where a block is located inside the cache?**
        - Since we have 4 blocks inside the cache, we will need 2 bits to locate a block inside it – these bits are called the **set bits**

32-Byte block, 128-Byte cache	
set	block
00	32-Byte long block
01	32-Byte long block
10	32-Byte long block
11	32-Byte long block

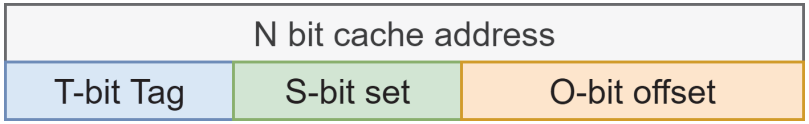
MEMORY HIERARCHY

- **Cache Addressing**
  - We end up with the following:
    - A cache is divided into sets
      - These sets hold Blocks of data
    - Blocks are divided into words
      - These words are the actual data we are storing



MEMORY HIERARCHY

- **Cache Addressing**
  - The cache address itself is also divided into three parts as follows:



    - The **T-bit Tag** bits:
      - The tag bits specify which block the data we are looking for.
      - The number of those bits specifies the number of blocks that can be addressed.
        - If we have 4-bit tag, we can address 2^4 = 16 blocks
        - These blocks can be **inside** the cache – a **cache hit**
        - Or they can be **not inside** the cache – a **cache miss**
          - Must be retrieved from higher levels of memory
    - The **S-Bit set** bits:
      - The set bits tells us which set of the cache a block is located.
    - The **O-bit Offset** bits:
      - The Offset bits tell us which word inside the block that we want to locate.

MEMORY HIERARCHY

- Cache Addressing
  - If we have a program made out of 155 instructions. Each instruction is 16 bits long.
  - What is the word size of these instructions if we want to fit each instruction into one single word?
  - How many blocks this program can be divided into knowing that a block of data is 16 Bytes in size?
  - How many blocks can we fit inside a 128 Byte cache?
  - How long should the address be to be able to address all these blocks?

MEMORY HIERARCHY

- **Cache Addressing**
  - **If we have a program made out of 155 instructions. Each instruction is 16 bits long.**
  - **What is the word size of these instructions if we want to fit each instruction into one single word?**
    - Since we can fit one instruction in a word, the word will be the same length as the instruction
      - 16 bit long words
      - But since we need to fit those into blocks, we convert into bytes
        - 16 bits = **2 bytes**

MEMORY HIERARCHY

- **Cache Addressing**
  - 155 instruction. 2-byte words.
  - **How many blocks this program can be divided into knowing that a block of data is 16 Bytes in size?**
    - Since a block is 16 bytes
      - 16 bytes divided by the size of words
        - 16/2 = **8 words per block**
    - Since we have 155 instructions, we need to find how many blocks needed to fit all these instructions
      - Each block can fit 8 words = 8 instructions
      - **155 / 8 = 19.3**
        - But we can not have blocks as fractions
          - We round up the result into the higher number
          - The number of blocks needed is **20 blocks**

MEMORY HIERARCHY

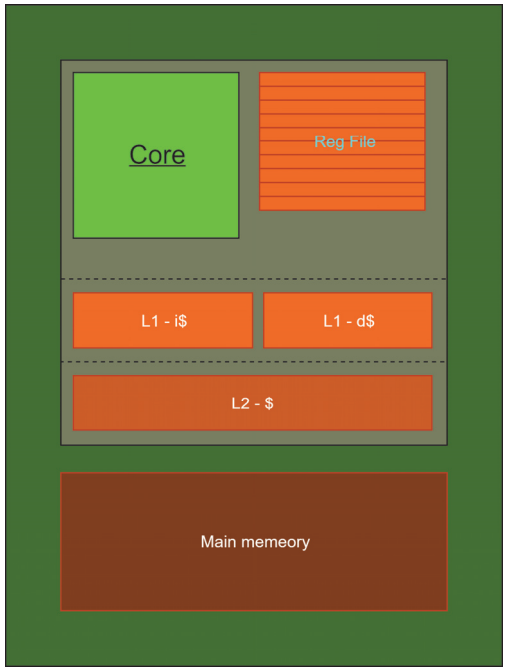
- **Cache Addressing**
  - 155 instruction. 2-byte words. 8 words per block. 20 blocks
  - **How many blocks can we fit inside a 128 Byte cache?**
    - Since a block is 16 bytes
      - We divide the size of the cache by the size of the blocks
      - **128 / 16 = 8**
        - We can store 8 blocks inside this cache
          - This means that we need **8 sets**

MEMORY HIERARCHY

- **Cache Addressing**
  - 155 instruction. 2-byte words. 8 words per block. 20 blocks. 8 sets in the cache
  - **How long should the address be to be able to address all these blocks?**
    - **The Tag bits** will have to represent all the blocks that we can address.
      - This means that we need bits to tag 20 blocks
      - We need **5 bits** to address 20 blocks
    - The set bits will have to represent the number of sets we can store blocks in
      - This means we need bits to represent 8 sets
      - We need **3 bits** to represent 8 sets
    - The offset bits will have to represent which word within a block we want to address
      - This means we need bits to represent 8 words per block
      - We need **3 bits** to represent 8 words

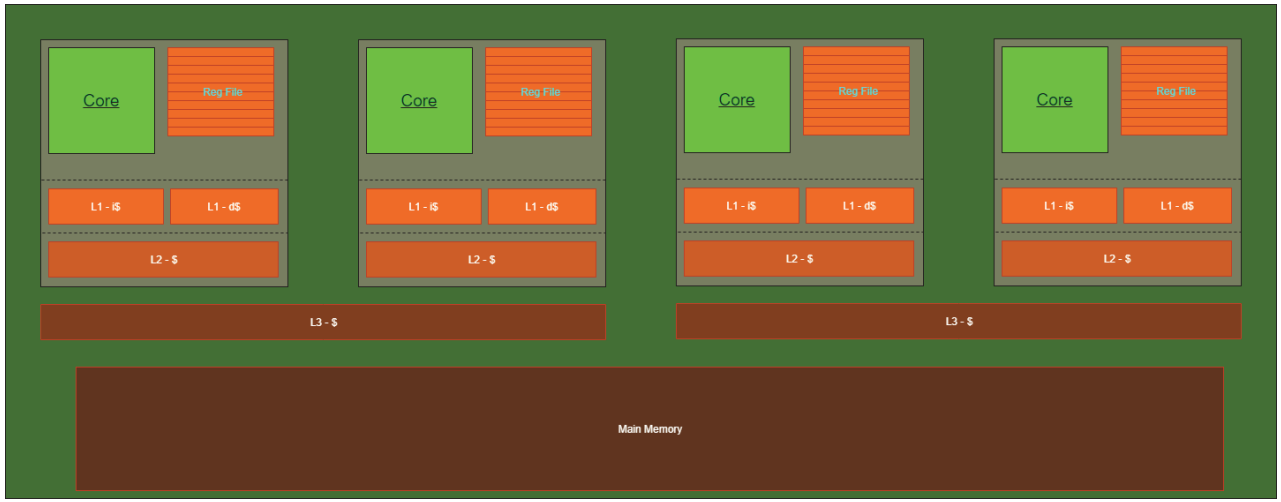
MEMORY HIERARCHY

- **Multi Level caching**
  - Because the cache is a very fast memory and its proximity to the CPU allows the CPU to perform better.
  - This can be taken advantage of by adding more caches into the system
  - The cache system is then divided into levels of caching
    - The lower caches are called L1, the next level is L2. and so on
  - The **lower the cache levels**, the more specialized the caches are
    - Lowest level (L1) has two specialized caches
      - **The Instruction cache** – also know as the I-cache (i$)
        - Is a cache dedicated to storing instructions
      - **The Data cache** – also know as the D-cache (d$)
        - Is a cache dedicated to storing program data
  - The higher the cache levels, the more general purpose they are
    - The L2 cache are normally shared caches
      - Storing both instructions and data



MEMORY HIERARCHY

- **Multi Level caching**
  - This can be done on multi core level as well.
  - A L3 cache is added as shared cache among cores



  - Higher levels of cache are Larger is size. Usually, a L2 cache will be 4 times larger than an L1 cache. An L3 cache will be 8 times larger than an L2 cache, and so on

  - Larger caches, can store more blocks, meaning that a cache miss doesn't always require to go the main memory to be mitigated

MEMORY HIERARCHY

- **Multi Level caching**
  - When a block is moved into lower levels of cache, it must be copied to all the cache levels before it.
    - Example: Let us assume we have 2 levels of cache and a block called **BlockX** in the main memory. What happens when the core requests a an instruction in that block

MEMORY HIERARCHY

- **Multi Level caching**
  - When a block is moved into lower levels of cache, it must be copied to all the cache levels before it.
    - Example: Let us assume we have 2 levels of cache and a block called **BlockX** in the main memory. What happens when the core requests a an instruction in that block
      - The core asks the **register file**, but the word is not in the register file
      - The register file requests it from the **L1 cache**, but it is not there – it is an **L1 cache miss**
      - L1 asks from **L2**, but it is not there – it is an **L2 cache miss**
      - L2 cache asks it from main memory – it is in main memory, the L2 cache receives an exact **copy** of the **BlockX** – the original copy of **BlockX** is still in main memory.
      - L2 send an exact copy of **BlockX** to L1 because it asked for it earlier
      - L1 sends the word that the core asked for to the register file
      - The register file delivers it to the core
  - An exact copy of **BlockX** now exists inside all the cache levels, but the original is in the main memory

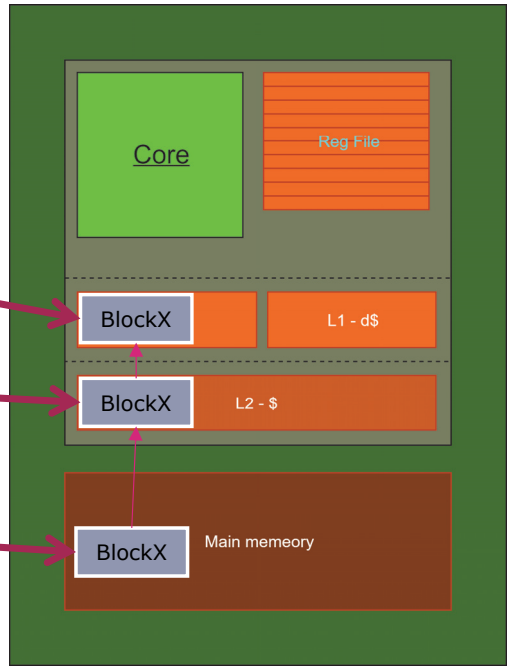
MEMORY HIERARCHY

- **Multi Level caching**
  - When a block is moved into lower levels of cache, it must be copied to all the cache levels before it.
    - Example: Let us assume we have 2 levels of cache and a block called **BlockX** in the main memory. What happens when the core requests a an instruction in that block
      - The core asks the register file, but the word is not in the register file
      - The register file requests it from the L1 cache, but it is not there – it is an L1 cache miss
      - L1 asks from L2, but it is not there – it is an L2 cache miss
      - L2 cache asks it from main memory – it is in main memory, the L2 cache receives an exact **copy** of the **BlockX** – the original copy of **BlockX** is still in main memory.
      - L2 send an exact copy of **BlockX** to L1 because it asked for it earlier
      - L1 sends the word that the core asked for to the register file
      - The register file delivers it to the core
  - *An exact copy of **BlockX** now exists inside all the cache levels, but the original is in the main memory*

This is an issue

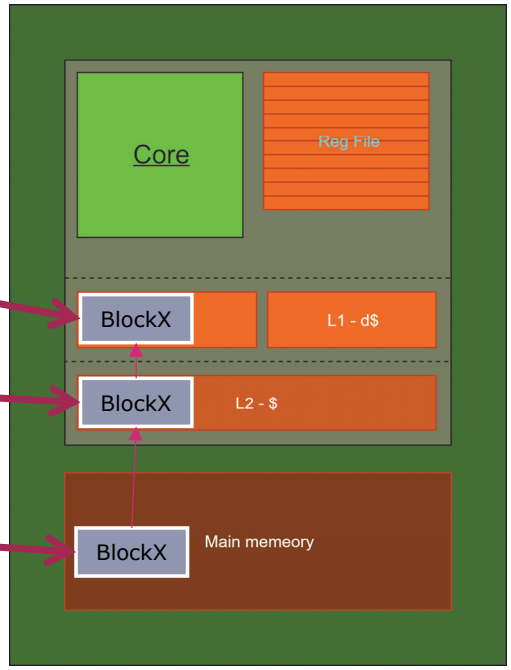
MEMORY HIERARCHY

- **Multi Level caching**
  - *An exact copy of **BlockX** now exists inside all the cache levels, but the original is in the main memory*
  - *L2 send an exact copy of **BlockX** to L1*
  - *L2 cache asks it from main memory – it is in main memory, the L2 cache receives an exact copy of the **BlockX***
  - ***BlockX** is in Main memory*



MEMORY HIERARCHY

- **Multi Level caching**
  - *An exact copy of **BlockX** now exists inside all the cache levels, but the original is in the main memory*
  - *L2 send an exact copy of **BlockX** to L1*
  - *L2 cache asks it from main memory – it is in main memory, the L2 cache receives an exact copy of the **BlockX***
  - ***BlockX** is in Main memory*
  - ***What happens if the data inside BlockX is changed?***



MEMORY HIERARCHY

- **Multi Level caching**
  - Since information in the cache is a copy of the information in other caches or in the main memory, once the data changes or updated, the rest of the locations where that particular cache information must be changed or updated accordingly.
  - There is are two main methods of updating a block of memory:
    - **WRITE BACK**:
      - Once the data is updated inside the cache address, it is synchronized with all the copies in all the other locations in all the other memories.
        - All the copies are changed at once, with no delay!
    - **WRITE THROUGH**:
      - Once the data is updated inside the cache address, it is not updated anywhere else, unless the block is being replaced in memory or updated somewhere else.
        - No copies are changed, unless the changed copy is being accessed or replaced in memory.
    - **What are the benefits of each of those methods?**

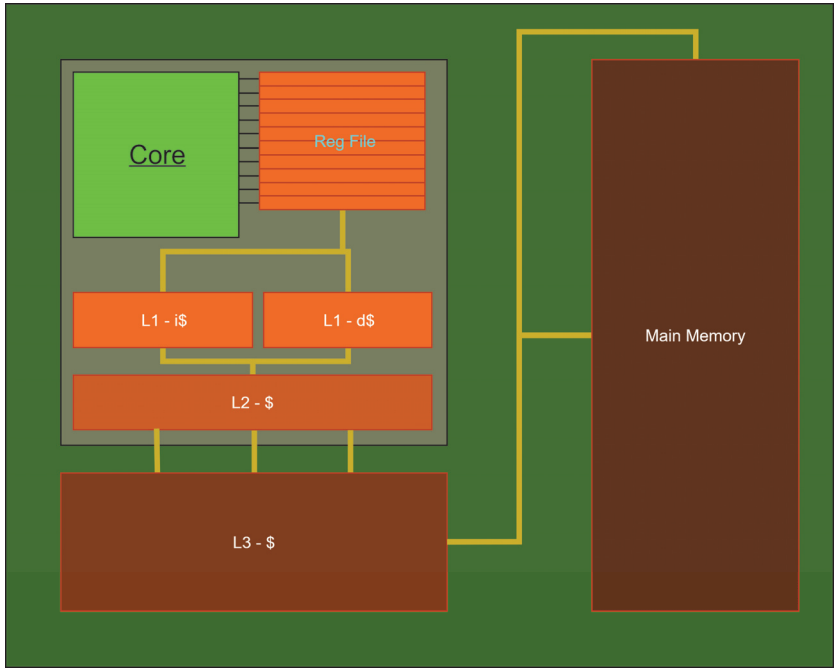
MEMORY HIERARCHY

- **Multi Level caching**
  - Memory Hierarchy structure:
    - Hierarchical Access Memory Organization
      - Explained in the previous 5 slides
    - **Simultaneous Access Memory Organization**
      - The CPU communicates directly with the different memory levels
      - This allows faster delivery of data to the CPU
      - This also requires extra management to make sure that all memory levels need to be in synchronization all the time.
      - It works like this:
        - The CPU ask the Reg file for data, if reg files does not have it, then the CPU, *NOT THE REG FILE*, asks the L1 cache for data.
        - If the cache does not have the data, the CPU, *NOT THE L1 CACHE*, asks the L2 cache for data…. And so on
        - The CPU takes care of communication and no communication needs to be done between memory devices.
        - Once data is found, it is copies from the memory device it is found in, to all the lower levels of the memory

MEMORY HIERARCHY

- **Multi Level caching**

Hierarchical Access Memory Organization



Simultaneous Access Memory Organization