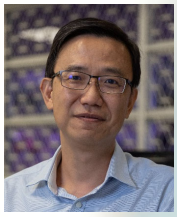
Cx1106
Computer Organization and Architecture

Cache



Oh Hong Lye
Senior Lecturer
School of Computer Science and Engineering, Nanyang Technological University.
Email: hloh@ntu.edu.sg

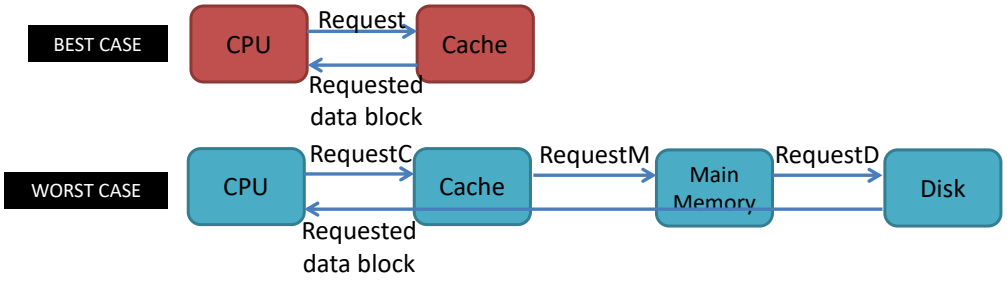
- This chapter is on Cache Memory Management. This is a module in the processor that is able to improve the overall system performance significantly by leveraging on the principle of locality exhibit by most application program. More details in this video.

Cache

- Issue: **CPU** speed is typically much **faster** than **external memory**.
  - CPU speed in Ghz region
  - External Memory Speed in 100s of Mhz region.
- **Need a fast memory** to act as a **buffer** between the Main memory and CPU.
- The purpose of cache memory is to speed up accesses by storing/fetching **recently used data** closer to the CPU instead of the main memory (slower access).
- Potentially able to **improve the overall system performance** drastically.

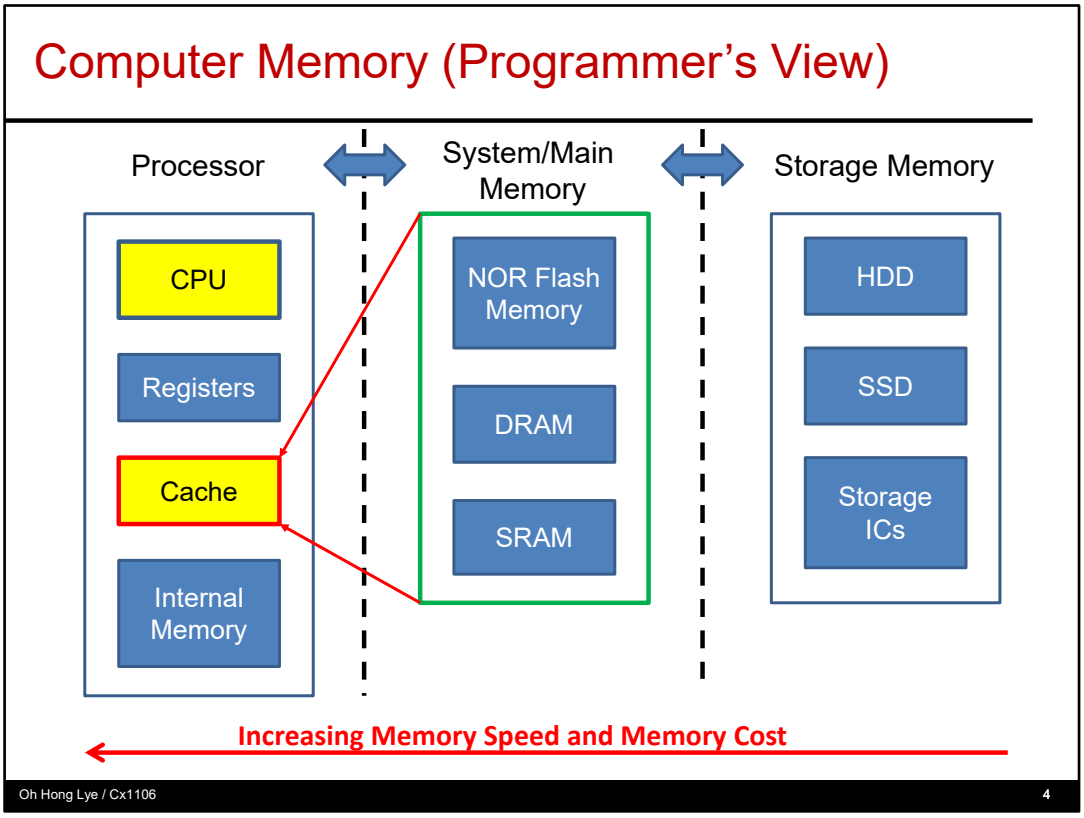
- Now, CPU typically run at a higher speed compared to the external memory.
  - So the external memory becomes the bottleneck and slows down the system performance.
- To increase the overall system performance, we need a fast memory to act as a buffer between the CPU and External Memory.
  - And that piece of fast memory is known as Cache.
- We'll see later that a well designed cache could potentially improved the system performance significantly.

CPU Memory Access



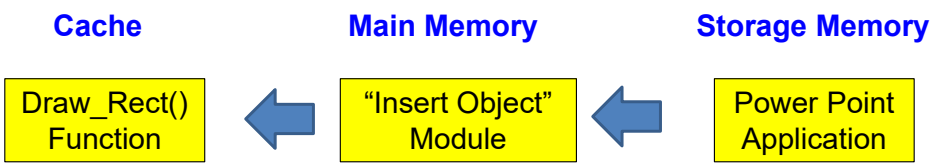
- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, a query is then sent to the main memory.
- If the data is not in main memory, then the request goes to disk.
- Once the data is located, the **required data and a number of its nearby data elements** are fetched into cache memory simultaneously.

- This slide shows you what really happen between the CPU, cache and main memory.
- In a processor with cache, the CPU will first visit the cache to see if it has the data the CPU needs.
    - If the data is there, then the requested data is transfer and the processor process with other operations.
    - If data is not in the cache, then CPU will need to look in the main memory or the HDD for the data.
        - Once data is found, it'll be fetched into the cache, together with some other nearby data elements.
- How the main memory data is stored in the cache depends on the mapping scheme used. And this is what we'll be going through in the next few slides.



- Let's re-visit a previous slide on computer memory from the perspective of a programmer.
- A program's code and data is stored entirely in the Storage memory.
- The specific sections of code and data is transferred to the System memory during runtime.
- A subset of the code and data which is recently or frequently used is stored in the cache for fast access by the CPU.
    - Different subset of code/data will be loaded to the cache as the program execute
        - The objective is to have the subset that CPU need most at that instance to minimise the time need to retrieve these information.
- Just to recap, CPU will first visit the cache to look for the information it needs, it will proceed to the System memory if it can't find the required information, and further check out the storage memory if it can't find the information in the System memory.
- You will see the system memory being reference by the name "main memory" in the rest of chapter. In this course, system memory and main memory are referring to the same piece of memory.

Cache Design - Introduction



- Example: Power Point Application
- Entire Application is stored in Storage Memory
- The user is trying to insert some object into the powerpoint slide
  - The corresponding code for object insert is transferred to Main memory for execution
- At the instance, the user is drawing a rectangle
  - The corresponding code for the Draw_Rect() function finds its way into the cache eventually.

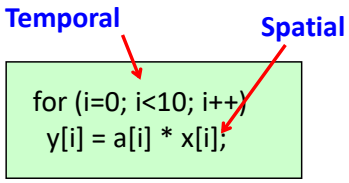
- This slide gives you an illustration of what happen in a Processor with cache when running a power point application.
- The entire powerpoint application is too large to be stored in the main memory so it is stored in the storage memory instead.
- User will only be using one of the many features so only those needed is loaded to the main memory.
  - In this case, it's the insert object module that is loaded.
- Even within the insert object module, user will only be using a sub-set of what the module provide
  - In the case, the user is trying to draw a rectangle.
  - The particular function that he needs will eventually find its way into the cache since it is the most recently and frequently used group of code.

Principle of Locality

- Now that we know cache only needs to contain a subset of the main memory to work
- In what way should the data be transferred between main memory and cache in order to maximise the efficiency of a cache?
- Efficiency meaning how much improvement can the cache brings to the system. This is related to the probability the CPU is able to find the information that it needs in the cache.
- For that, we need to understand the attributes of how information is organised and accessed when a program is executed.
- This is summarised in the concept called Principle of Locality

- From previous slides, we know that CPU only needs a subset of the entire program during runtime.
- Which is why cache size can be very small compared to system and storage memory.
- So the next question we need to answer is what data should be stored in the cache in order to maximise the resultant system performance?
    - To maximise the system performance, we need to maximise the cache efficiency.
    - The efficiency here is directly related to the probability of CPU finding what it needs in the cache.
- To enable that, we need to understand the attributes of how code and data is organised and accessed when a program is executed.
- This is summarised in the concept called principle of locality.

Principle of Locality



- The **principle of locality** tells us that once a byte in a program is accessed, it is likely a **nearby data element** will be needed soon.
- There are two principle of locality governing this behaviour
- Locality of **Space** (or Spatial Locality)
  - **Code/Data that is nearby each other** is likely to be accessed together
  - Transfer of data between main memory and cache is done in blocks to leverage on this behaviour
- Locality of **Time** (or Temporal Locality)
  - **Recently accessed code/data** is more likely to be accessed again
  - Used to decide which item to replace in the Cache

Oh Hong Lye / Cx1106 7

- When a byte of code or data is access within a program, it is very likely that its nearby code or data will be needed soon.
  - This attribute is known as the principle of locality
- There are two types of locality
  - Locality of space and Locality of time
  - Also known as Spatial and Temporal Locality
- Spatial locality refers to the scenario where the code/data that are located near to each other are likely to be accessed together
  - There leads to the cache design where if a particular byte required by the CPU is not in the cache, then the cache module will transfer that byte together with its neighbours to the cache. That is, any transfer between the cache and main memory always happen in blocks and not single byte. This is so that subsequent access will result in cache hit. Cache hit means CPU is able to find the required information in the cache.
- Temporal locality, on the other hand, refers to the scenario where the recently accessed code/data is likely to be accessed again.
  - This sort of justify why a cache will be effective even though its size is small.
  - It is also one of the considerations used when designing the cache replacement policy
- The small for loop shown here illustrate the two principles.
  - The for loop had 10 iterations, with each iteration performing a sum of product between array a and x, and storing the result to array y.
  - The sequential access of the arrays illustrate the concept of spatial locality.
    - When one element of an array is used, the chances of subsequent

elements being access is high.

- The 10 iterations implies the code for the sum of product will be executed 10 times, an illustration of temporal locality in operation.

Cache Memory Replacement Policy (need to move to after mapping)

- When there is a need to transfer a block of data to the cache but the cache is fully occupied, there is a need to decide which cache block to evict/purge in order to free up space for the new data.
- The algorithm used to decide which block gets evicted is designed based on some Cache Replacement Policy. Two examples illustrated below
- **Least recently used (LRU)** algorithm keeps track of the last time that a block was assessed and evicts the block that has been unused for the longest period of time. The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.
- **First-in, first-out (FIFO)** is a popular cache replacement policy. In FIFO, the block that has been in the cache the longest, regardless of when it was last used.

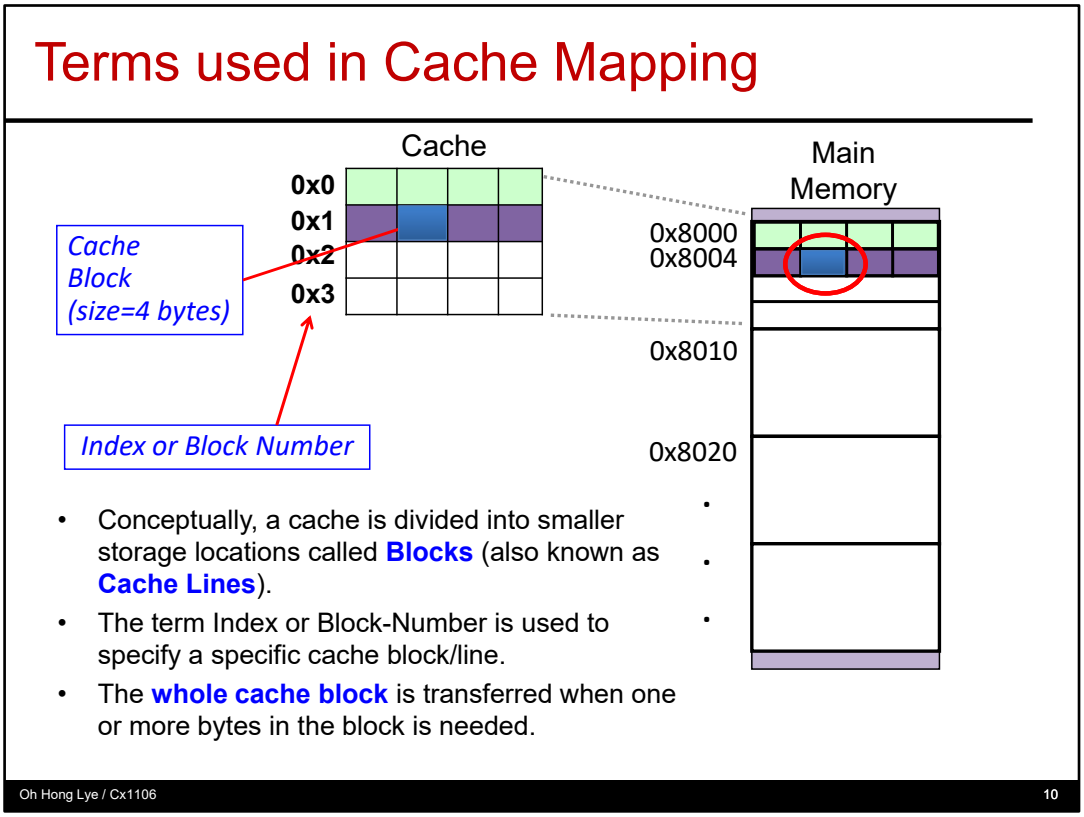
- The other part of cache storage design deals with replacement policy.
- While the direct mapped cache doesn't really need a cache replacement policy as its mapping is one-to-one, its good to understand the general concept behind replacement policy. Set associate or fully associative cache, which you will cover in Advanced Comp Arch module, will need cache replacement policy to decide which blocks gets replaced when the cache is full.
- In general, if the cache is full and a new block needs to be introduced into the cache, then the cache module will need to decide which cache block to evict to make space for the new block.
    - For Direct Mapped Cache, since the mapping is one-to-one, the block to evict is known before hand so there isn't a need to have any algorithm to manage the block replacement.
    - For Set Associate and Fully Associate Cache, which is not covered in this course, there is an option to choose from two or more blocks to evict. Hence, a replacement policy has to be put in place.
- Two of these methods are described here
    - Least Recently used algor. This scheme will evict the cache block that has not been used for the longest time. It's a very efficient scheme since the algor conform well to the locality principles. But implementation is more complex and costly as the cache needs to keep a access history for each block.
    - FIFO. Another popular scheme with simpler implementation. It basically replace the first block that comes into the cache. It conform approximately to the locality principles, just that it doesn't take into account of the scenario where a cache block is used multiple times. This typically result in lower

efficiency compared to LRU.

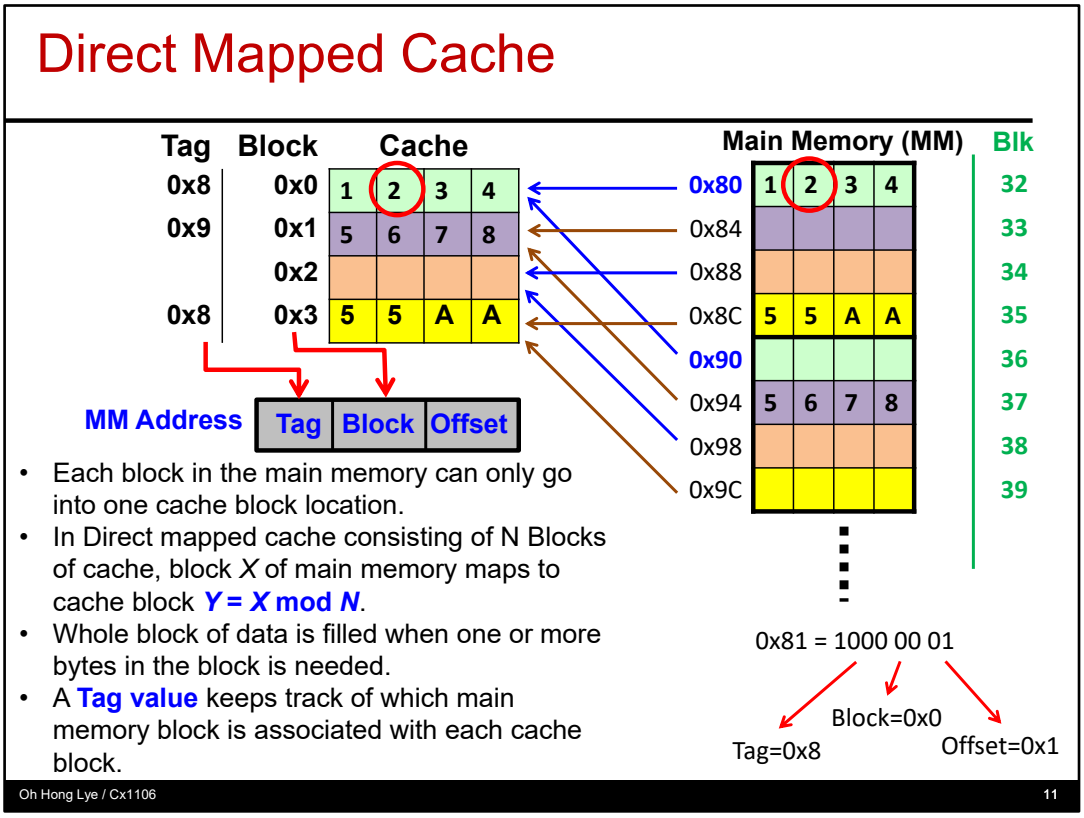
Cache Mapping Scheme

- We understand now that data transfer between main memory and cache is done in blocks instead of individual bytes to leverage on the principle of locality.
- Another attribute of cache design that affects the efficiency of the cache is the cache mapping scheme.
- Cache mapping scheme deals with how each main memory block is mapped to the a particular cache block, e.g. Main memory block #0x80 is mapped to cache block #0 (index 0).
- There are three basic cache mapping schemes
  - Direct Mapped
  - Set Associative
  - Fully Associative
- We will only touch on Direct Mapped Cache in this course, rest of the mapping scheme will be discussed in Advanced Computer Architecture Course.

- Another attribute of the cache design that will affect the efficiency is the cache mapping scheme
  - This deals with how each block in the main memory is mapped to the cache
- There are three basic types of mapping scheme
  - Direct Mapped
  - Set Associative and
  - Fully Associative
- For this course, we will only touch on the Direct Mapped Cache. The other two mapping scheme will be discussed in the Advanced Comp Arch module.



- Before we go into the details of cache mapping scheme, let's go through some basic terms that we will be using.
- First is the Cache Block or Cache Lines
  - Cache is divided into blocks consisting of multiple bytes
  - For this example, the cache block contains 4 bytes each.
- Each Cache block in the cache is numbered. This is known the cache block index or block number.
  - First cache block of the cache has a index 0.
- In cache analysis, the main memory is typically organised in the same format as the cache, i.e if the cache has a block size of 4 bytes, the main memory will also be arranged in blocks of 4 bytes each.
- When a particular byte in the main memory is needed, the entire block is transferred to the cache instead of the particular byte.
  - This as mentioned earlier, is to leverage on the principle of locality.



- For Direct mapped cache, each main memory block can only go into one specific block location in the cache.
- The way they are allocated is given by the formula X mod N, where X is the main memory block number and N is the total number of cache blocks available.
    - So MM block 0 goes to cache block 0, MM block 1 goes to cache block 1 and so on. MM Block 4 for this cache structure will wrap around and go to cache block 0.
    - In this example, MM BLK32 goes to Cache BLK 0, the mapping continue sequentially until MM BLK36 where it wraps around and get mapped to Cache BLK 0.
- As mentioned in the previous slides, although only one data byte is needed, the whole block of data is copied to the cache.
- Question here would be: How do we know which main memory block does the data in the cache correspond to?
    - For that, we need to look at the tag value, which is used to uniquely identify which MM block is in the cache.
        - For example, block starting 0x80 will be place in cache block 0 and have a tag value of 0x8, 0x8C will be placed in block 3 and have tag value of 0x8 as well.
        - 0x94, on the other hand, will be placed in block 1 and has a tag value 0x9.
- By now, you must be wondering how I derive these values.
    - The values are derived from the MM address.
    - Take for example 0x81.

- The first step is to partition the MM address into three fields consisting of TAG, BLK and OFFSET.
- To know how many bits the ‘offset’ field supposed to have, we need to understand the definition of the offset field.
    - An offset specify the position of the target byte from the start of the cache block.
    - For this example, the cache block size is 4, the offset field will consist of 2 bits as that is the number of bits needed to support 4 different address representation. For ease of calculation, you just need to use LOG2 of 4.
- Next is the ‘BLK’ field. Similarly, this field describe which block the target data is transferred to, so if the cache contains 4 block, 2 bits is required for the ‘BLK’ field.
- Lastly, the TAG field, which is used to uniquely identify each MM block, is derived from subtracting the bits allocated to offset and BLK. Fro this case, MM address range is 8 bits, so number of TAG bits = 8-2-2 = 4.

- After we partition the MM address according to the TAG, BLK and OFFSET field, we can easily determine which cache block this particular MM block will be transferred to by looking at the BLK field.
    - In this example, we can see that the MM block which contains 0x81 will be mapped to Cache Block 0 and the corresponding TAG value is 0x8.
    - We can also tell that the actual byte of interest is one byte away from the start of the cache block.

Cache Mapping (Elaboration on basic principle)

- Cache mapping
  - Allocates the data in the entire main memory into the cache which is of a much smaller size.
  - Able to uniquely identify each and every main memory location within the cache via the cache way of addressing: Block Index, Offset of the data within the block and the corresponding Tag Value of the block.
- To start doing the mapping, we need an attribute of the target information that is unique to it within the entire main memory, for that, the main memory address of the data is chosen as each target information's MM address is unique to itself.
- So the target information's MM address is partitioned into the three fields: TAG, BLOCK and OFFSET so that proper allocation to cache can be done.

- This slides elaborate on what we have learn so far regarding cache mapping scheme.
- Cache mapping involves the following tasks
  - Design a systematic way to map the main memory to a cache whose size is smaller than the main memory.
  - Allow the CPU to retrieve the information it needs from the cache, based on the MM address of the target information.
- The name 'target information' refers to the target code or data that the CPU needs.
- Since the CPU will use the MM address as the reference to retrieve the information, it is only natural that the MM address is used as the basis to enable the mapping process.

Cache Mapping (Elaboration on basic principle)

- The **number of bits allocated to each field** is a result of the structure of the cache.
  - **Offset** refers to the targeted data's location offset from the start of the cache block. Something like an address within the cache block. So if the size of a cache block is 16, one would need 4 bits in the OFFSET field to address these 16 locations.
  - **Block** refers to the index of the cache block that the targeted data will be mapped to. If there are 8 cache blocks for example, then one would need 3 bits in the BLOCK field to fully represent the cache block index.
  - **Tag** bits are the left over of target data's main memory address after partitioning for Offset and Block Index. Having this information will allow the cache system to **uniquely identify** the data in the cache.

- This slide contain detail description of each of the TAG, BLK and OFFSET field.
- The content has been discussed in previous slide so I will not elaborate further.
- You can pause the video here to review through the points again.

Direct-Mapped Cache Mapping Example

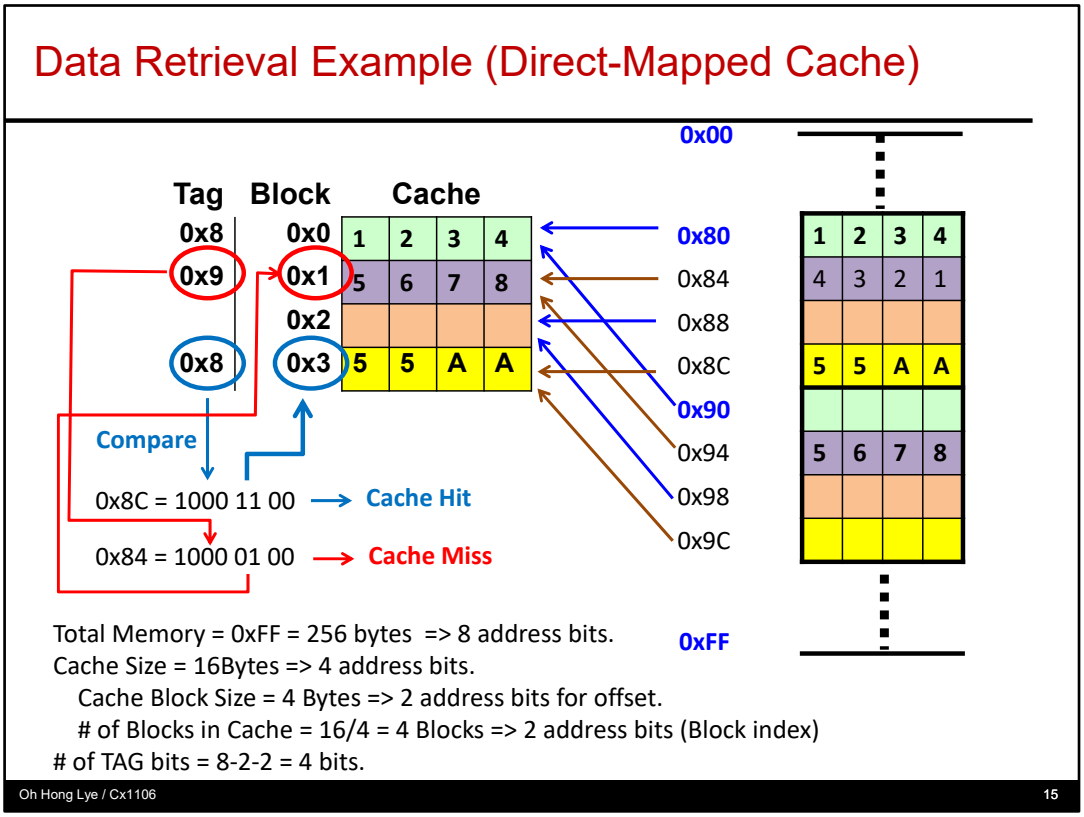
- Given a system with following attribute
  - **Main/System Memory size = 64KBytes**
  - **Cache Size = 256Bytes**
  - **Cache Block Size = 16Bytes**
- Where would Data at main memory address **0x1106** be mapped to?
- Derivation of cache mapping format
  - Cache Block Size = 16Bytes => #**Offset** bits = log2(16) = **4 bits**.
  - Number of Cache Blocks = 256/16 = 16 Blocks
    => #**Blk** bits = log2(16) = **4 bits**.
  - Main memory size = 64KBytes => log2(64*1024) = 16bits address.
    #**Tag** bits = 16-4-4 = **8 bits**
  - Mapping Format = 8:4:4
- Applying to the main memory address 0x1106
  - 0x1106 = **0001 0001 0000 0110b**
  - **BLK = 0x0**, **OFFSET=0x6**, **TAG=0x11**

Oh Hong Lye / Cx1106 14

- Next, let's go through a work example to check our understanding of the concepts we have learn so far for direct mapped cache mapping scheme.
- Given a system with
  - MM size 64KBytes,
  - Cache Size 256 Bytes
  - Cache Block Size 16Bytes
- Where would the Data at MM address 0x1106 be mapped to?
- Using the concept we learnt on derivation of the TAG:BLK:OFFSET fields for direct mapped cache,
  - Starting with the offset first, cache block size = 16 implies offset field is LOG2(16) = 4 bits.
  - Number of cache block is not given but that can be derived by dividing the cache size by the cache block size.
    - So there are 16 cache blocks in the cache. That implies BLK field is also 4 bits.
  - The TAG field is obtained by subtracting the MM address bit with OFFSET and BLK bits.
    - So number of TAG bits = 8.
  - That means the Mapping Format is 8:4:4
- When doing the cache mapping analysis, always remember to first convert the MM address to binary, performing analysis with hexadecimal MM address will very likely result in careless mistakes.
  - After converting 0x1106 to binary and applying the mapping format
  - We can see that the data at 0x1106 will be mapped to cache BLK 0 and has a

TAG value of 0x11.

- The target byte is located at byte 6 from the start of cache block.
- Byte 0 is the first byte of the cache block.

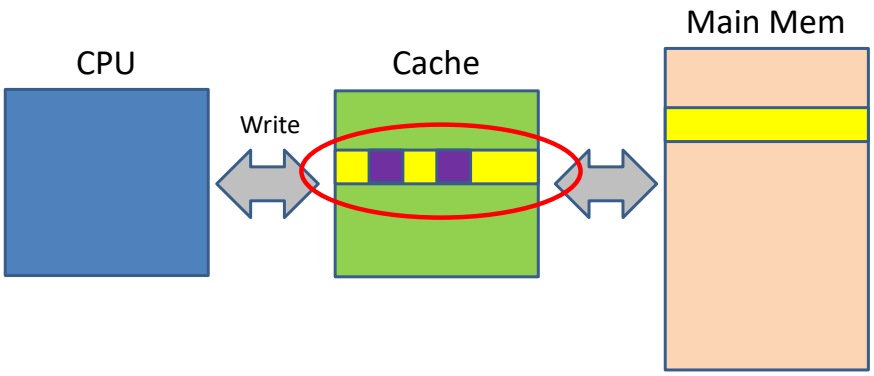


- After the work example, let's go through the data retrieval process to se how CPU evaluate whether a particular access is a cache hit or cache miss.
- Using the procedure mentioned in earlier slides, we can derive the mapping format for this cache structure
    - 4:2:2
    - Note that mapping format will change if the cache structure change e.g. change in cache size, cache block size and/or MM address range.
- Supposed after some data exchange, CPU wanted to retrieve data stored at location 0x8C.
    - Applying the 4:2:2 format on the 0x8C, BLK value is '3' so CPU proceed to check out cache block 3.
    - To find out whether the data stored at cache block 3 comes MM block that contains 0x8C's data, which btw is MM BLK 35.
        - It compare the TAG value of cache block 3 and the TAG value derived from 0x8C MM address.
        - Result is a match, that implies the data in cache block 3 does indeed comes from MM BLK 35. This is known as a Cache Hit.
- The CPU then proceed to access 0x84.
    - Applying the same procedure again, we realise that this time, the TAG value of cache block 1 is different from the TAG value derived from the 0x84 MM address. This means the data in cache block 1 does not come from MM block where 0x84's data is resided. This is known as a Cache Miss.
- If there is a Cache Hit, CPU will retrieve the data and proceed with other operations.
- If there is a Cache Miss, the corresponding MM block will be transferred into the

cache.

Cache Write Policy

- Locations in cache that are written into by CPU is known as **Dirty Block**
- Cache replacement must take into account **dirty blocks**, those blocks that have been updated while they were in the cache.
- Dirty blocks must be written back to memory. A **write policy** determines how this will be done.
- There are two types of write policies: **write through** and **write back**.

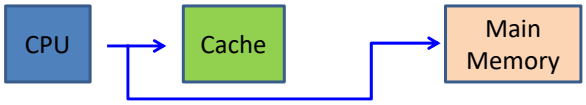


Oh Hong Lye / Cx1106 16

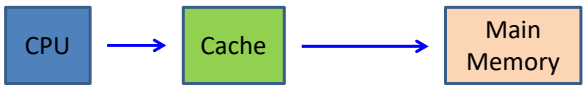
- So far, our discussion has always been related to CPU read operation.
  - In the next few slides, we will be looking at effects on cache when CPU perform write operation.
- A special marking is done to the cache block which the CPU writes into and these blocks are known as Dirty Blocks.
- Cache replacement needs to take into account of dirty blocks.
- E.g one main memory block was cached. And CPU write into some location in this block within the cache.
  - The content of the block in cache and main memory is now different.
  - Meaning that if we want to evict this block, we'll need to first write back this block to the main memory to maintain data coherency.
- There are two types of write policies: write through and write back.

Cache Write Policy (Write Hit)

- **Write through** updates cache and main memory **simultaneously** on every write.



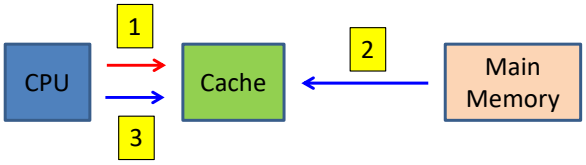
- **Write back** (also called copyback) updates memory **only** when the block is selected for replacement.



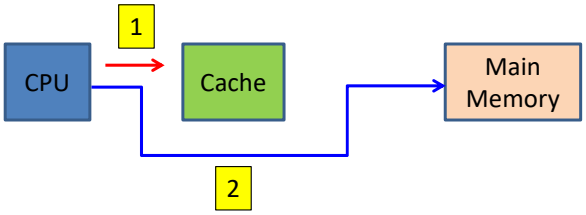
- Write through means if CPU write to a cache location, it'll also update the main memory at the same time.
- But for a Write back scheme, the main memory will be updated only when the cache block Is evicted.

Cache Write Policy (Write Miss)

- **Write allocate** => fetch on write. A write miss will cause the data block at the write-miss address to be **loaded to the cache**.



- **Write-no-allocate** => A write miss will not cause the data block to be loaded to the cache. Data **Write** is done **directly** to its location in **main memory**.



Oh Hong Lye / Cx1106 18

- Correspondingly, there are two ways in which CPU can handle write miss.
  - Write miss here refers to cases where CPU want to write to a certain main memory location but found that it is not in the cache.
  - A cache miss could be either a read miss or a write miss.
- The two ways in which CPU handle write miss are
  - Write allocate.
    - Under this scheme, the corresponding MM block will be loaded to the Cache when there is a write miss.
    - CPU will follow with a read to the cache to retrieve the data
  - Write no-allocate
    - With this scheme, CPU will write directly to the corresponding MM block if there is a Write Miss.

Cache Performance Related Terminologies

- **Cache Hit**
  - Data is found in the cache.
- **Cache Miss**
  - Data is not found in the cache.
- **Cache Hit rate (H)**
  - Percentage of time data found in the cache
- **Cache Miss rate**
  - Percentage of time data not found in cache.
- **Miss rate = 1 - Hit rate = (1 – H).**

- Some definition of the key parameters used for calculating Effective Access Time.
- We have encountered most of them previously, so I won't go through them again.
- You can pause the video briefly to revise the definitions.

Effective Access Time

Sequential Access of Cache and Main Memory, i.e. access do not overlap.

- The performance of hierarchical memory is measured by its effective access time (EAT).
- EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.
- The EAT for a two-level memory is given by

$$EAT = H \times Access_C + (1-H) \times Miss\ Penalty$$

- $Access_C$ = Access times for cache
- Miss Penalty = Time need to access the data when there is Cache Miss
- If we assume that data access to Cache and Main memory do not overlap, then Miss Penalty = $Access_C + Access_{MM}$, where $Access_{MM}$ is the access times for main memory

$$EAT = H \times Access_C + (1-H) \times (Access_C + Access_{MM})$$

- Effective Access Time, EAT in short, is the weighted average of the access time during cache hit and cache miss.
- Now, CPU always visit the cache first when it needs information.
    - If it can find the information in the cache, it's a cache hit and access time incurred will be the Cache access time.
    - If CPU cannot find the information in the cache, it's a cache miss and CPU will need to visit the Main Memory to retrieve the information.
        - If we further assume that access to cache and main memory happen sequentially and do not overlap,
            - Then access time during cache miss will be cache access time + main memory access time.
                - The cache access time is incurred when CPU first visit the cache to look for information.
                - The main memory access time is incurred when CPU realised that it is a cache miss and need an additional MM access time to retrieve the data.
        - As mentioned, the effective access time is a weighted average of the two access timing and is calculated by applying the probability of each of the scenario to the two access time component. Resulting in the expression you see in the slide.

Effective Access Time (Example)

Sequential Access of Cache and Main Memory

- Consider a system with a **main memory access time of 200ns** supported by a **cache** having a **10ns access time** and a **hit rate of 99%.**
- If the **accesses do not overlap**,

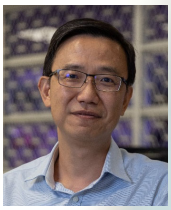
  The EAT is:

  0.99(10ns) + 0.01(10ns + 200ns) = 9.9ns + 2.1ns = 12ns.
- This equation for determining the effective access time can be extended to any number of memory levels.

- Applying what we learn about calculation of the EAT in one work example.
- System has a MM access time of 200ns, cache access time of 10ns and cache hit rate of 99%
- Remember the assumption that access to memories do not overlap.
- EAT is obtained by using the expression in the previous slide.
  - We will obtain 12ns.
- Here is a good example showing how a small cache can realise a significant improvement in Effective Access Time of the memory system.
- With that, we come to the end of the Cache Memory System module. Next lecture will be on Virtual memory management.

Cx1106
Computer Organization and Architecture

Virtual Memory



Oh Hong Lye
Senior Lecturer
School of Computer Science and Engineering, Nanyang Technological University.
Email: hloh@ntu.edu.sg

- This chapter touches on the concept of Virtual Memory Management in a Computer System.
- This is an important concept that is used widely in the design of a Operating System.
- We will cover the basics in this course and you will re-visit these concepts again when you take the Operating System Module.

MEMORY ADDRESS SPACE

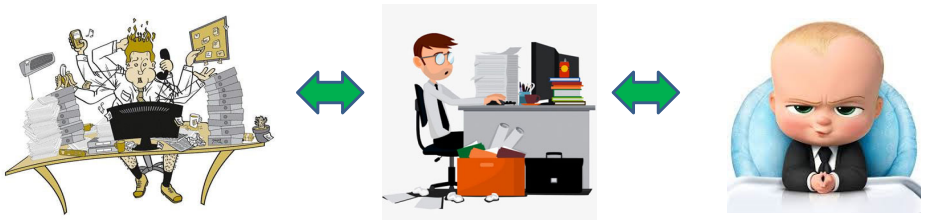
- Before we start on the Virtual Memory Management Topic, I would like to use a few slides to introduce the concept of memory address space in computing.

Memory Address Space (Analogy)

Address Space 3 (Employee)

Address Space 2 (Manager)

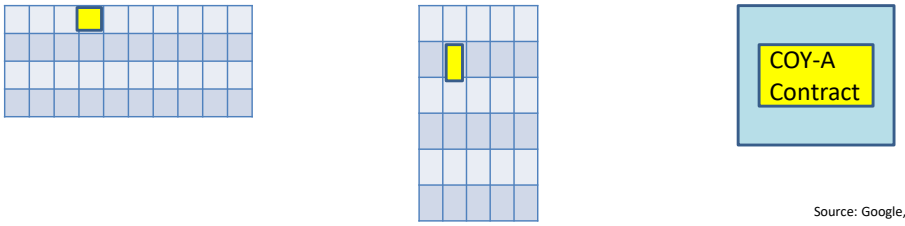
Address Space 1 (Boss)



Cabinet (Company Name)

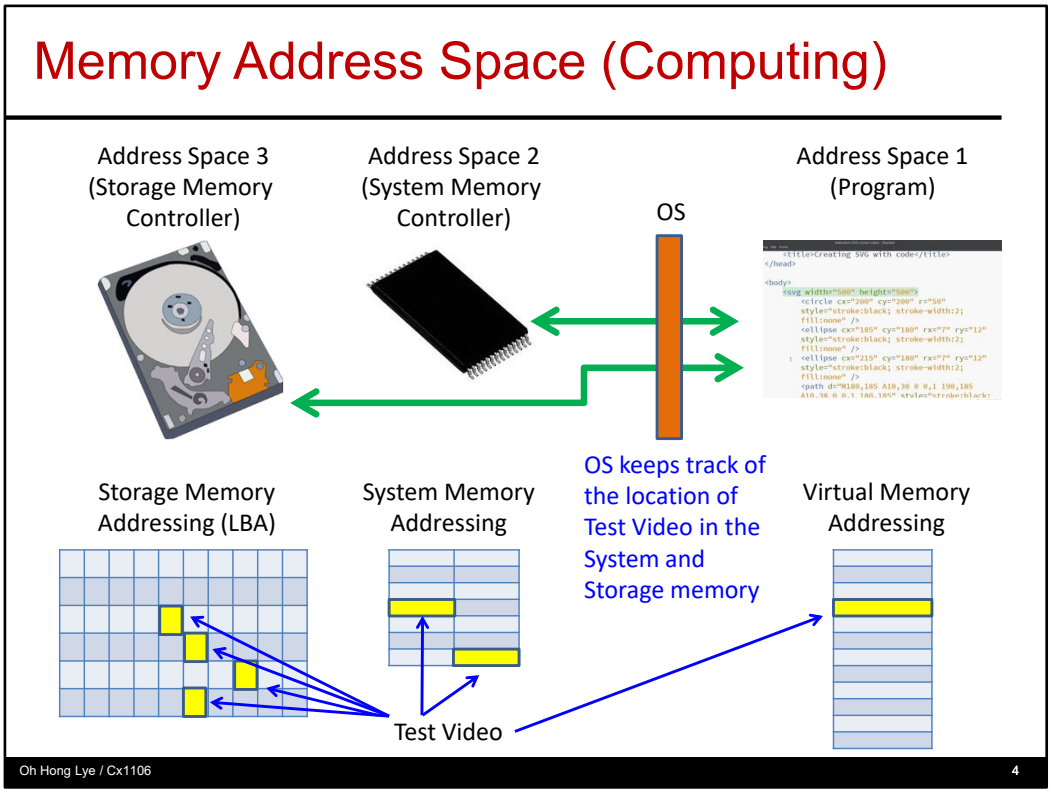
Cabinet (Market Segment)

Drawer (all-in-one)



Source: Google, Baby Boss.

- Let's start with an analog.
- Memory space are how a sub-system or module keep its information.
- Take for example, in a company,
    - the big boss will keep the most important documents in his drawer.
    - His manager will probably have a larger cabinet consisting of the key contracts and information specific to the market segments they supervise.
    - The subordinates would likely have their own cabinet which they kept documents corresponding to the specific customers they call on.
- The three parties do not know how each other store their information. But the company operation will still run smoothly, as long as, when the boss needs a specific piece of information, his managers or the subordinates under the managers can provide him the required data.
- As long as the boss is happy, everything is ok.



- Coming back to computing.
- Different entities within the computer stores information differently.
    - Each will have their own memory space and own addressing scheme.
- The program user developed uses the addresses generated by the compiler and further allocated by the Operating System,
    - the memory space correspond to the virtual memory address space of the program.
    - Note that each program has their own virtual memory address space.
- The actual program code and data are stored in the storage and system memory.
    - As mentioned previously, the entire program code/data resides in the storage memory while the system memory stores the subset of the program that is needed during runtime.
- As illustrated, the same piece of information, e.g. a test video clip, is stored in a different manner in the three entities.
    - In the program's virtual memory space, the test video could reside in contiguous piece of memory
    - In the System memory, it could be distributed across different frames in the system memory.
    - While in the storage memory, it may be allocated in random LBA blocks in the HDD.
- Similar to the analog in the previous slide, the program will still be able to function properly, as long as when it needs the test video, the corresponding data would be made available to the program.

- Since the test video is stored in different manner across the entities, that mean some form of address translation have to occur in order for the program to retrieve the data correctly. The address translation is done by the OS. And we will go into details of one of the scheme employed.

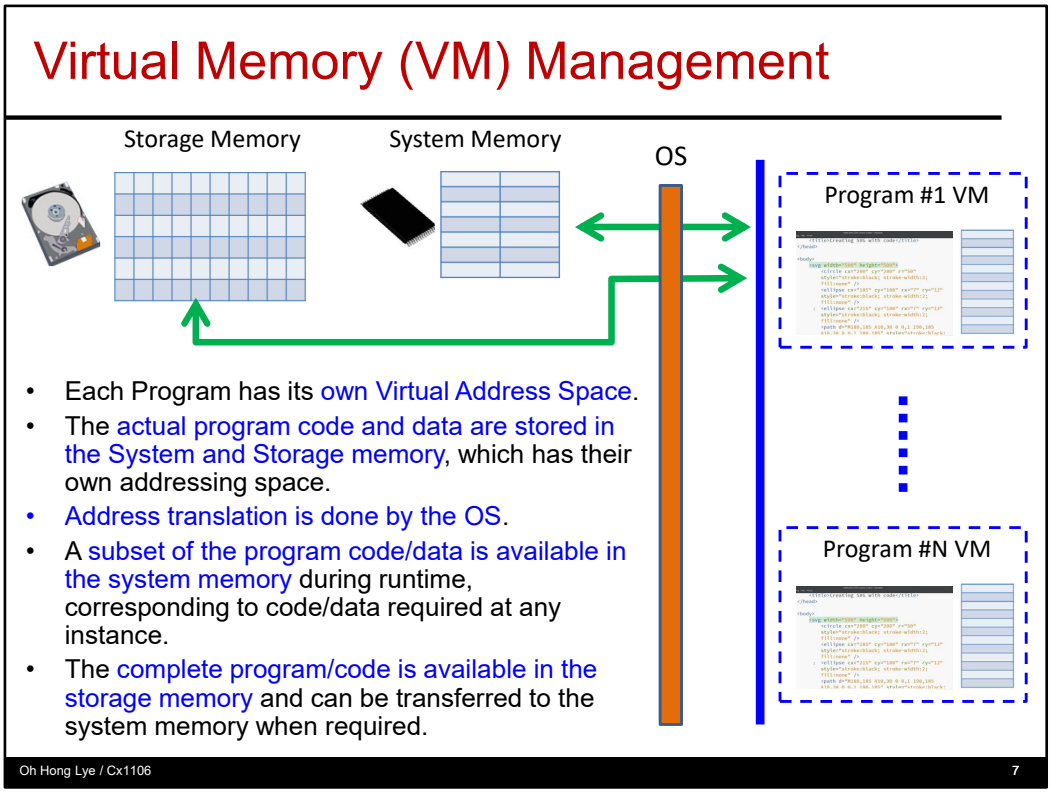
Basic Concepts

- In computing, address space is a range of discrete addresses corresponding to some physical or logical entity, e.g. program virtual memory, physical system memory, logical layout of a HDD etc.
- Every piece of data/code in a program is assigned an address by the compiler during the program compilation.
- When executing a program, the information it requires is obtained by issuing a request to the operating system (OS) using the addresses assigned by the compiler.
- The program doesn't need to know how the required information is organised in the system memory and rely on a middle man to do the corresponding address translation in order to fetch the correct information.
- In this case, the operating system is the middle man, translating the compiler generated address to the system memory address where the required information is stored.
- Program will still work as long as it gets the code/data it requested.

- To recap, address space is a range of address corresponding to a physical or logical entity.
- The address used by a program is generated by the compiler.
- During program execution, OS will translate the compiler generated address to the system memory address where the information is physically stored.
- The program doesn't need to know where the actual information is stored.
- It will work properly as long as it gets the information it needs when it request for it.

VIRTUAL MEMORY MANAGEMENT

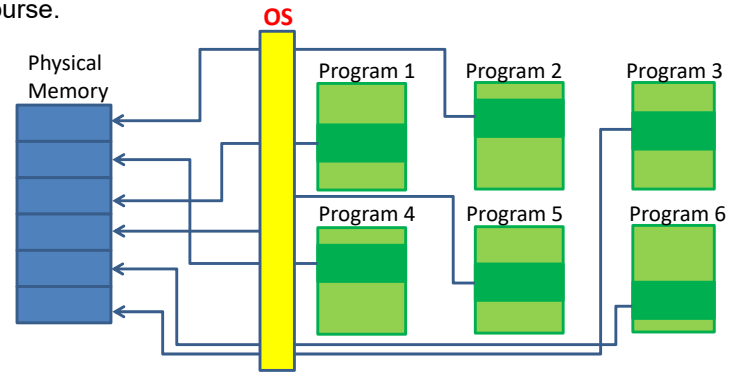
- Next, we get into the Virtual Memory Management discussion.



- As mentioned in previous slides, each program has its own virtual address space.
  - But the actual program code/data is stored in the system or storage memory, which has their own address space.
- Therefore, address translation needs to be done in order for the program to retrieve the information it needs, and this translation is done by the OS.
- Functionality wise, system memory stores the code/data required during any instance at runtime while the storage memory stores the entire program code/data.

Virtual vs Physical Memory Address Space

- The addresses used by the program is generated by the compiler and is known as the virtual address.
- The virtual address of the code/data is typically different from the Physical Memory Address that they will be resided.
- Address Translation is thus required and is done in hardware and/or software, managed by the Operating System (OS).
- Note that Physical, System and Main memory refers to the same memory for this course.

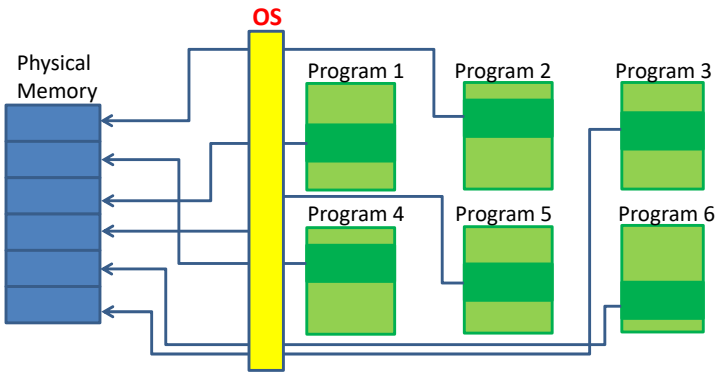


Oh Hong Lye / Cx1106 8

- Similar to the use of "Main Memory" in the Cache Chapter to refer to the System memory, in this chapter, System memory is sometimes referred to as the "Physical Memory", basically means the actual runtime memory the program code/data is stored in, this versus the Virtual Memory which is really a logical entity.
- The OS is responsible in managing the virtual memory, which includes allocating the subset of the program to the physical memory and performing the address translation from virtual to physical during program execution.
- The slide here shows the scenario where subset from multiple programs are allocated to the physical program simultaneously, with the OS handling the corresponding address translation.

Advantages of Virtual Memory

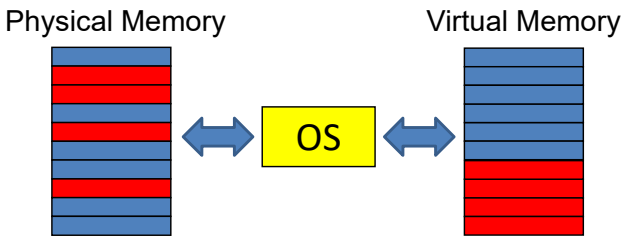
- OS is able to isolate each virtual memory space and prevent corruption between these spaces.
- Allow efficient and safe sharing among different programs within the shared physical memory.
- Allow one or more programs to run in the physical memory simultaneously even if the total size of all the programs is larger than the actual physical memory size.



- Some of the advantages of having a virtual memory management system are
  - Virtual Memory space of each program can be isolated from each other to prevent intended or unintended corruption between programs.
    - This allows efficient sharing among different programs with the shared physical memory
  - Virtual Memory management allows multiple programs to run simultaneously in the physical memory even if the total size of all the programs are larger than the physical memory size. This is illustrated in the diagram below where 6 programs shared a physical memory whose size is smaller than the total of the 6 programs.

Advantages of Virtual Memory

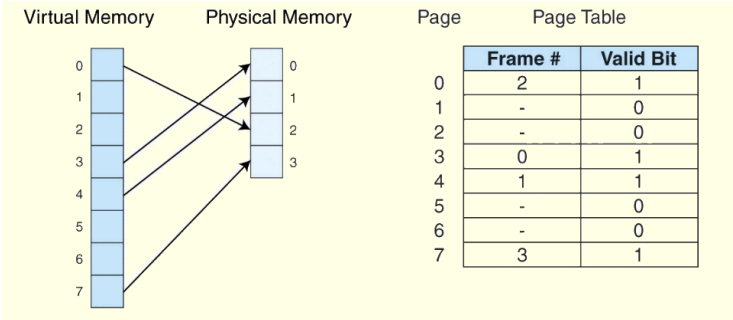
- OS is able to relocate virtual memory blocks to any physical memory blocks. This allows the virtual address space seen by the application to appear to be contiguous when it is actually spread across fragmented blocks in the physical memory.



- Another advantage of Virtual Memory Management is that the virtual address space seen by the program can appear to be contiguous when it is actually spread across different segments of physical memories.
- Having a contiguous piece of memory greatly simplify coding as the software does not need to do complicated boundary condition checking and management.

Address Mapping

- There are two common schemes used in the industry for address mapping
  - Paging. Memory space partitioned into Fixed sized blocks
  - Segmentation. Memory space partitioned into variable sized segments.
- For our course, we will only deal with paging with single level page table.



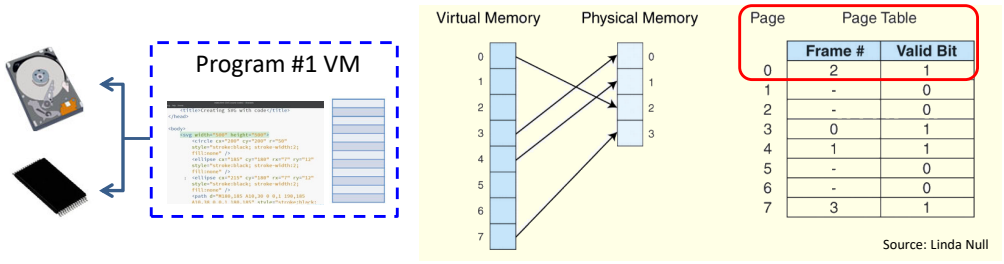
Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

Source: Linda Null

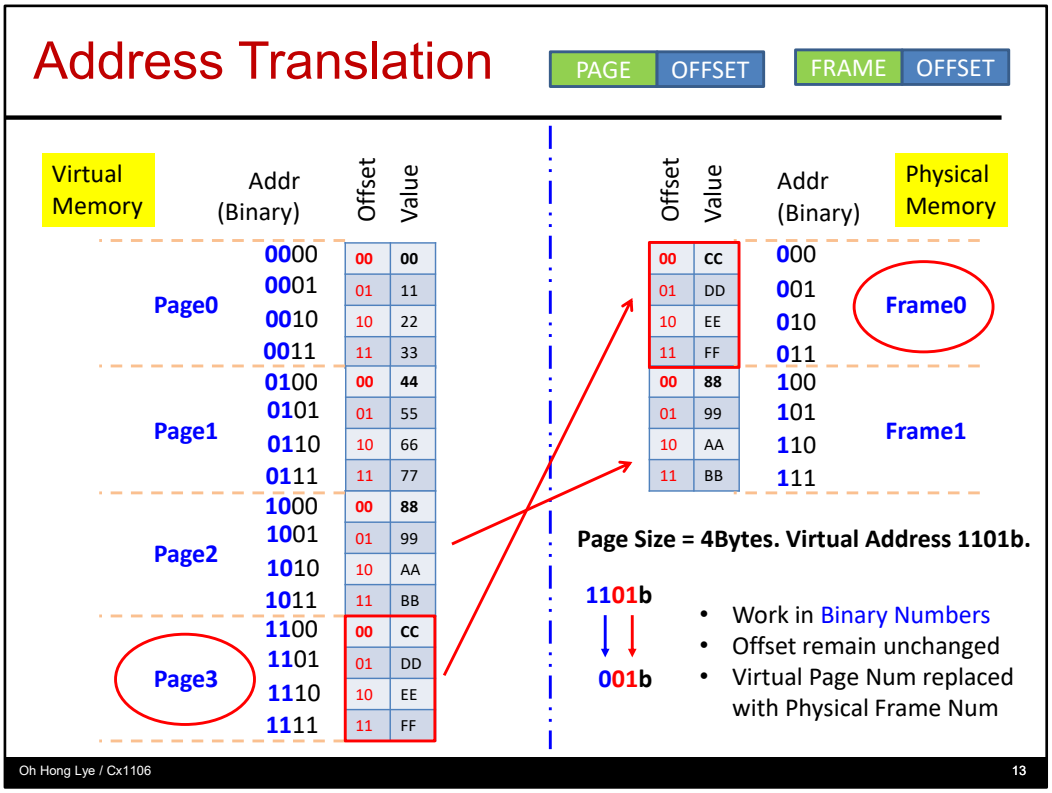
- There are two common scheme used in the industry for address mapping.
  - Paging and Segmentation.
- We will only cover Paging in the course.
- For Paging, the virtual and physical memory space are divided into fixed size pages and frames respectively.
  - We use the name 'Page' for Virtual Memory and 'Frame' for Physical Memory.
- The OS will map a virtual page to a particular physical frame and the translation information is store in a data structure known as a Page Table. More on this in later slides.
- Our course will only cover paging with single page table, you will progress with multi-level page table in your Operating System Course.

Paging Method

- In a system that uses paging, the memory space is partitioned into fixed size blocks known as a Page/Frame.
- Information concerning the location of each page, whether on disk or in memory, is maintained in a data structure called a page table (shown below).
- In the Page Table below
  - Frame refers to the physical frame number in the main memory.
  - Page refers to the virtual page number used by program code.
  - Valid Bit (VB) indicates whether the Virtual Page is in the main memory (VB=1) or not (VB=0).



- The size of a Virtual Page is always the same as the size of a Physical Frame in Paging Scheme of address mapping.
- The OS will allocate a particular virtual page used by the program to be stored in a particular Frame in the physical memory during runtime.
  - The mapping information is kept in the Page Table shown.
- Within the Page table, you will see at least three parameters
  - Virtual Page column containing the virtual page number of the target virtual page.
  - Frame column containing the corresponding frame that the virtual page is mapped to.
  - And the Valid column showing whether the particular entry is valid or not.
    - If the valid bit is '1', this implies that corresponding mapping is valid and the information in that particular page is in the physical memory at that instance. E.g. in the page table shown, Virtual Page 0 information is resided in the Physical Frame 2 at that instance.

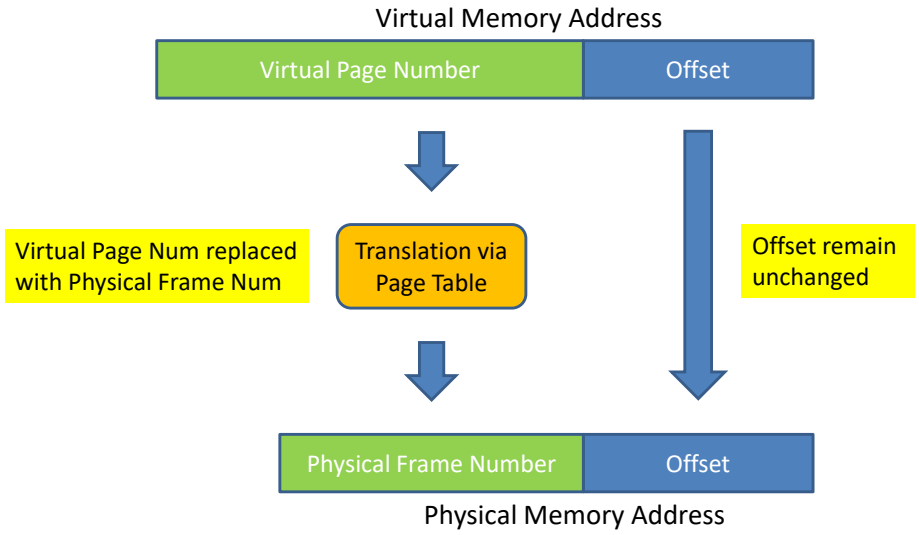


- This slide illustrate the process of address translation from Virtual to Physical Memory Space via a work example.
- The system shows here has the following attributes
    - Virtual Memory Space of 16 bytes.
    - Physical Memory Space of 8 bytes.
    - Virtual Page size of 4 bytes, meaning the Physical Frame size is also 4 bytes.
- First point to note is that when we map a virtual page to a physical frame, the relative position of the data in the page does not change.
    - You can see that when Page 3 is mapped to Frame 0, the offset of the 4 bytes of data in the Page does not change. E.g. offset of the data '0xEE' is 2 or 10 in binary in the Virtual Page3, the same data "0xEE" has an offset of 2 as well in the Physical Frame 0.
- With that, let's start the address mapping process.
    - First, convert the address to binary format. This is an important step, never work in hexadecimal format, there is a high possibility that you will make careless mistake.
    - Next, we need to partition the Virtual Address to two fields – Page and Offset.
        - This is done starting with the offset
            - Using the fact that page size is 4, then number of bits allocated to the Offset Field is 2. LOG2(4) = 2.
        - We can deduce the virtual page number from the upper address

bits of the virtual address and use that to look up the page table to find any matching Physical Frame Number.
        - If the entry for the particular page number is valid, the Corresponding Frame Number is used to replace the upper address bits to give the Physical Memory Address the virtual address is mapped to.
- Let's put some real values to illustrate the translation.
    - Starting with a virtual address of 1101 in binary.
    - Offset field is 2 bits so the address 1101 is stored in virtual page 3, 11b are the upper bits.
    - Page 3 is mapped to Frame 0 from the diagram.
        - So to derive the physical memory address, we replace '11' in binary with '0'.
        - This gives the Physical Memory address 001 in binary.
- You can try out other Virtual Memory Addresses in the is slide to reinforce your concept.

Address Translation

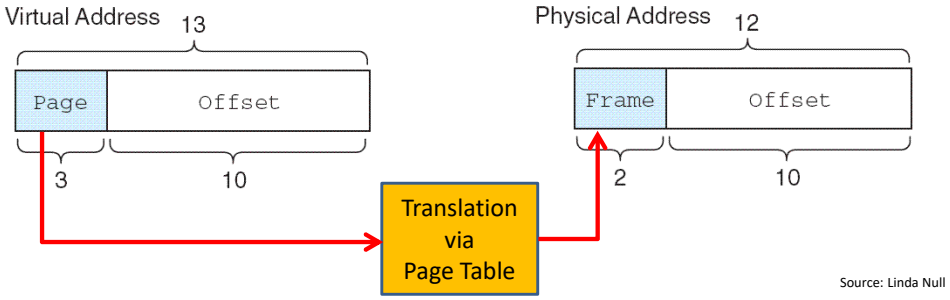
Work in Binary Numbers format



- A graphics illustration of the address translation process we discussed in previous slide.
- Convert the addresses to binary.
- Derive the number of bits in the offset field from the virtual page size.
- Partition the Virtual Memory Address to two fields: Virtual Page Number and Offset.
- The Offset remains unchanged during the translation.
- Use the Virtual Page Number to look up the Page Table for a valid mapping
- If a valid mapping is found, extract the corresponding Frame Number to replace the Virtual Page Number with the Physical Frame Number to have a Physical Memory Address.

Paging Example

- Consider a system with a virtual address space of 8K and a physical address space of 4K, and the system uses byte addressing.
- If page size is 1KByte, we have 8 virtual pages mapping to 4 physical frames.
- Note that Virtual Page Size is always equal to Physical Frame Size.
- A virtual address has 13 bits (8K = 2^{13}) with 3 bits for the page field and 10 for the offset, because the page size is 1024.
- A physical memory address requires 12 bits, the first two bits for the page frame and the trailing 10 bits the offset.

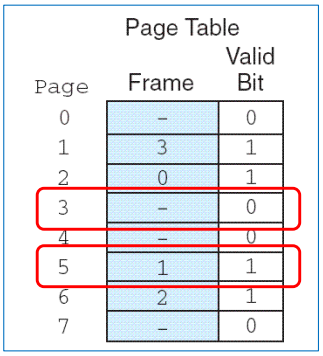


Source: Linda Null

- More work example.
- Consider a system with
  - Virtual Address Space 8KByte
  - Physical Address Space 4Kbyte
  - Page size of 1KByte
- From the attributes, we know that the system has 8 virtual pages and 4 physical frames.
- The page size is 1KByte, which is 2^10 bytes. That implies the OFFSET field is 10 bits.
- Virtual address is 13 bits and Physical Address is 12bits.
- So the translation process involves replacing the Upper 3 bits of the Virtual Address with the 2 bit Frame Number from the Page Table entries.

Paging Example

- Suppose we have the page table shown below.
- What happened when CPU access virtual address location
  - 0x1553
  - 0x0FA0



Page Table

Page	Frame	Valid Bit
0	–	0
1	3	1
2	0	1
3	–	0
4	–	0
5	1	1
6	2	1
7	–	0

Source: Linda Null

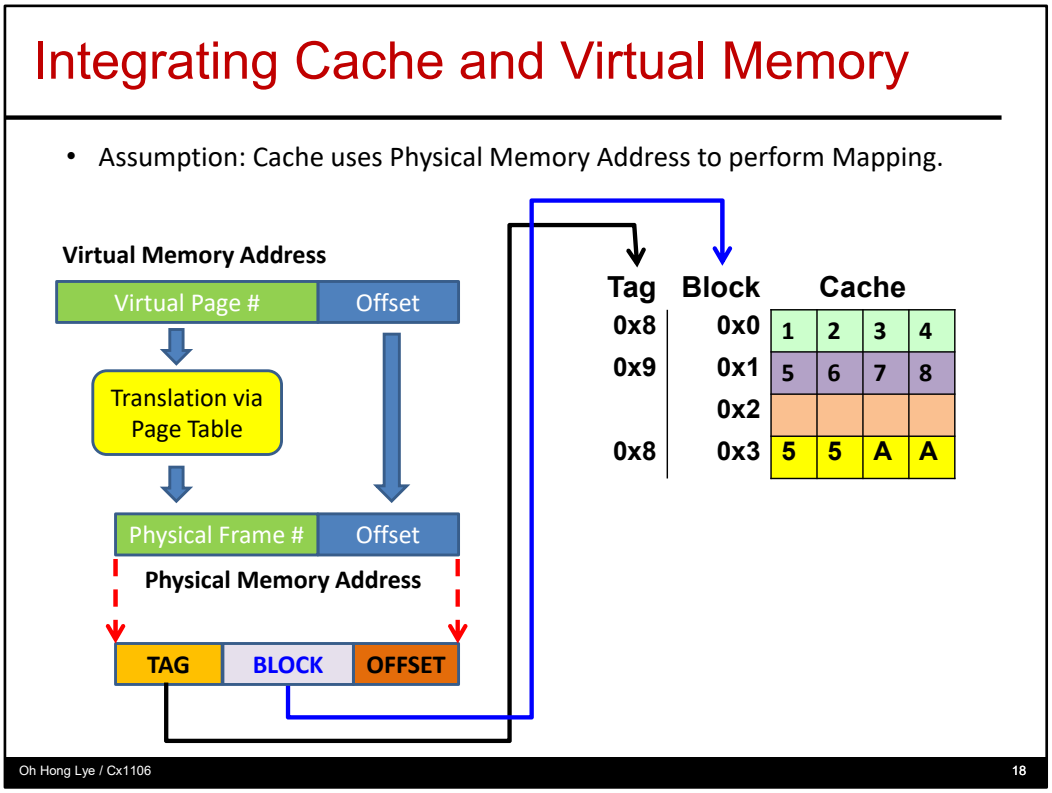
- 0x1553 = 1010101010011b
  - Data resides in Virtual Page 5
  - From Page Table, Virtual Page 5 is mapped to Physical Frame 1 and Valid bit is 1.
  - Physical Address = 010101010011b = **0x0553**
- 0x0FA0 = 0111110100000b
  - Data resides in Virtual Page 3
  - From Page Table, Valid bit is 0
  - Data not in Physical Memory
  - **Page Fault**

- Putting in some values.
- 0x1553 correspond to virtual page number 5.
  - Looking at the page table, we see that virtual page 5 has its valid bit set, meaning that it is mapped to some physical memory and from the same table, we see that the its frame #1.
  - Translation process involves replacing the upper 3 bits (101b) in Virtual Memory Address with 1.
- For 0x0FA0, it corresponds to virtual page number 3.
  - When we look at the page table, we can see that its valid bit is 0. Meaning no physical memory is mapped to this virtual page.
  - This generates a page fault that triggers the OS to load the required page from storage memory to the Physical Memory.

Paging Example

- In the previous slide, when accessing virtual address 0x1553 (which is in virtual page 5), the page table shows the valid bit is 1. From the page table, physical frame 1 will be used for subsequent address translation needed to access the actual data.
- When accessing virtual address 0xFA0 (virtual page 3), the page table shows that the valid bit is zero. If the valid bit is zero in the page table entry for the virtual address, this means that the page is not in memory and must be fetched from Storage Memory.
  - This is a page fault.
  - If necessary, a page is evicted from memory and is replaced by the page retrieved from storage memory, and the valid bit is set to 1.
- For both cases above, the data is then accessed by appending the offset to the physical frame number (address translation is simply replacing the virtual page number with the corresponding physical frame number in the page table).

- This slide summarises the discussion we did in the previous slide.
- You can pause the video for a while to review the points again.

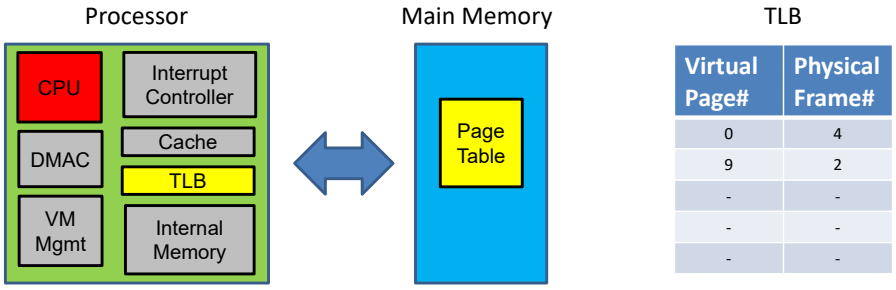


- In a typical computer system, especially one with an operating system, both cache and virtual memory management will be implemented.
- Therefore, there is a need to look at integration of these two modules.
- This slide shows one of the ways in which the processor may retrieve the information required when executing a program
- With a software program, we would always start with the virtual memory address.
- If we assume that the cache in this processor uses physical memory address to perform the indexing and mapping, then we following procedure will be carried out within this computer system.
    - First, the virtual memory address will be translated to its corresponding physical memory address using the methods we discussed in previous slides.
    - The CPU then anaylse if there is a cache hit or cache miss using the target's physical address.
    - If it is a cache Hit, then CPU will proceed to retrieve the information from the cache and proceed.
    - If it is a cache miss, then cache management module will proceed to perform the transfer of data from System/Storage memory.
- Note that the sequence will be slightly different if the Cache is using Virtual Memory Address to do the indexing and mapping.
    - Starting with the virtual address used by the program again, the CPU will visit the cache first to check if it is a cache hit or cache miss.
    - If it is a cache hit, CPU will retrieve the data and proceed.

    - If it is a cache miss, that means the block containing the required data needs to be fetched from the Main Memory.
        - But the OS will have to translate the virtual memory address to physical memory address before the data can be fetched.
- As seen, the sequence of access and translation is different but the concept behind each of the cache mapping and virtual memory translation is the same.

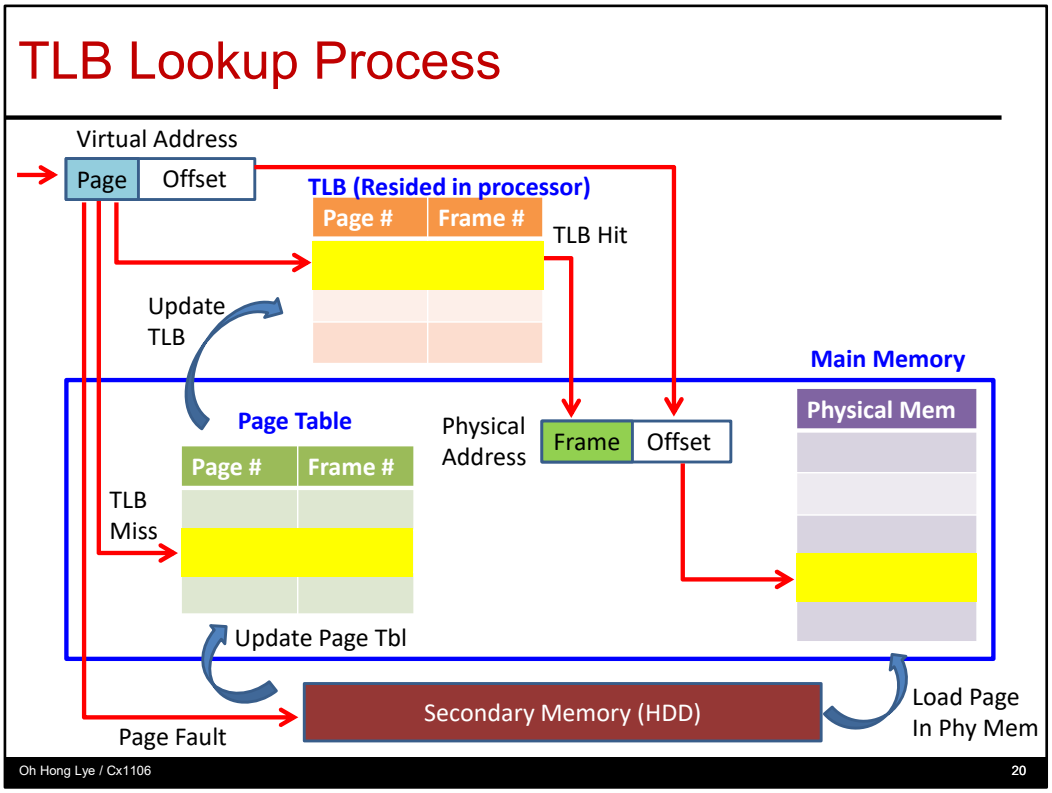
Translation Lookaside Buffer (TLB)

- Page Table is located in Main Memory so access is relatively slower than internal memory access.
- To speed up the page translation process, a page table cache (TLB) is used to stored a list of most recently/frequently used address translation entries in the page table.
- TLB is implement with fast memory such as SRAM and reside within the processor.
- Each TLB entry includes a Virtual Page number and its corresponding Frame Number.



Oh Hong Lye / Cx1106 19

- Due to its size, which could go up to a few MBs, Page table is located in the Main Memory, which is relatively slower compared to the internal memory.
- Since the process of translating Virtual Memory Address to Physical Memory Address happen very frequently, it makes sense to optimise this process.
    - One way is to use a fast internal memory, called a TLB (Translation Lookaside Buffer) to store frequently used page table information.
    - And get the OS to refer to the TLB first when they are looking for address translation information.
    - If the hit rate of the TLB is high, the overall speed of the address translation will speed up significantly.
- TLB is implemented with fast memory like SRAM and resides in the processor.
    - It stores a subset of the Page Table information.
    - Each TLB entry includes the Virtual Page Number and the corresponding Physical Frame it is mapped to.
        - All entries in the TLB are Valid entries.
- So in some way, TLB function like a cache to the Page Table.
- But the difference between a TLB and the Cache we discussed previously is that a TLB stores address translation information i.e. which virtual page gets mapped to which physical frame. While a regular cache stores the actual code and data of a program.



- The slide here illustrate the 3 ways that the virtual to physical memory address translation could be carried out.
- Starting with the Virtual Memory Address.
    - The OS will check the TLB for a matching entry.
        - If the virtual page being access has an entry in the TLB, then it is a TLB hit.
        - The OS can extract the corresponding Physical Frame Number the virtual page is mapped to and proceed to derive the Physical Memory Address and access the data.
    - If the require entry is not in the TLB, then it is a TLB miss.
        - OS will proceed to check the Page Table resided in the Main Memory for the information.
        - If a valid entry if found in the Page Table, it is a Page Table Hit, the OS proceed to update the TLB, compute the physical address and access the data.
    - Under the worst case scenario, the entry in the Page Table is invalid, that means the required data is not in the Physical memory and the OS needs to transfer the data from the Storage memory to the Physical Memory and update the page table.
    - Whether the TLB gets updated or not will depend on the Virtual Memory Management design.
    - That is the end of the chapter on Virtual memory management.