

# Processor Architecture

# Virtual Memory

Last modified 4/4/24

# Virtual Memory

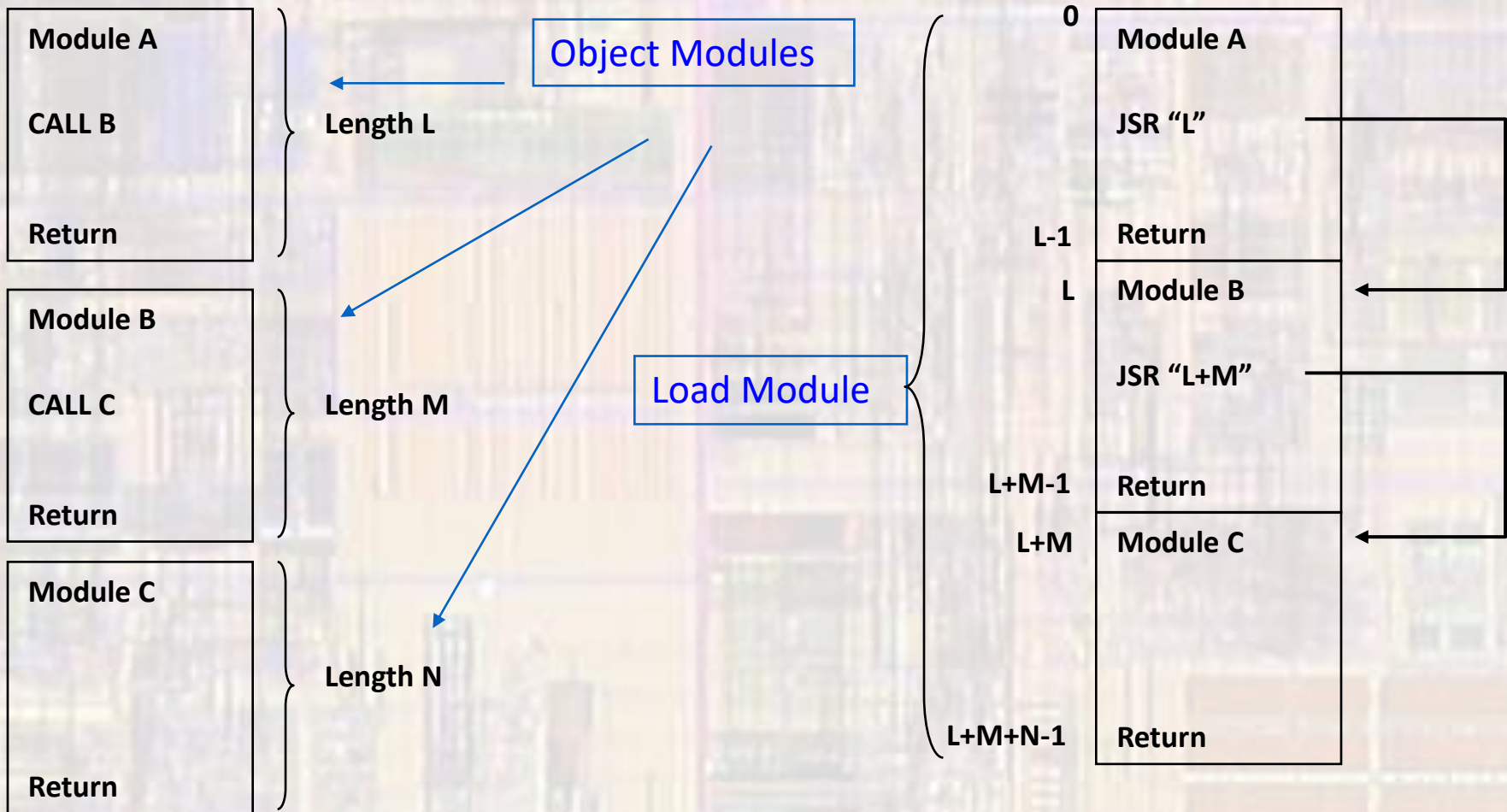
- Virtual Memory
  - Use main memory as a cache for secondary memory
    - Typically HDD for PC systems, Flash for mobile systems
  - Why?
    - Allow multiple programs (VMs) to share a common memory
    - Manage the limitations of main memory size
  - How?
    - Cache portions of the secondary memory in main memory
    - Allows different programs to be resident in main memory with-in different cache blocks (pages)

# Virtual Memory

- Virtual Memory
  - If we cache different programs into main memory, we create an addressing issue – processor uses the absolute address in main memory
    - How does the processor know where specific parts of code get cached to?
    - The location can change each time the program is started
    - The location can change if it gets swapped out and then back in again
  - Each program gets its own address space
    - This is a fictitious (virtual) address space
    - Only the program can access its address space
    - The address space is defined at compile time and does not change
  - The processor must translate(through HW and SW) from this “virtual address” to a physical address.

# Virtual Memory

- Linking Code



# Virtual Memory

- Terminology
  - Virtual Memory operates like a cache but some of the terms used are different for historical reasons
    - Address generated by the processor – virtual address
    - Blocks used in the VM system are called Pages
    - Misses in the VM system are called Page Faults
  - Physical Memory – typically refers to main memory (DRAM)
  - Address Translation – converting the virtual memory address created by the processor to a physical memory address associated with the DRAM, flash or disk
  - Swap Space – a copy of all the virtual memory space (required by a process) on the HDD – makes finding unloaded pages easier (16GB)

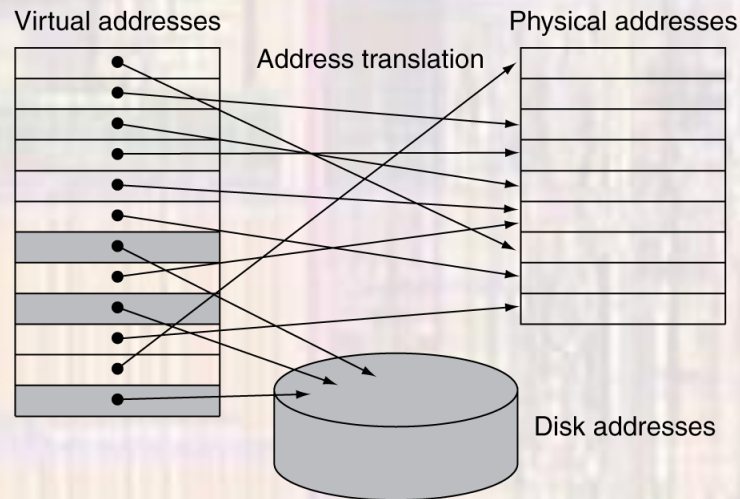
# Virtual Memory

- Address Translation
  - Each program has its own virtual memory space
  - When loading the program, the processor will map the virtual memory location loaded (used by the program) - into the corresponding physical address actually used to store the code/data
    - Not all of the program need be loaded – only the pages needed
  - Pages are fixed in size (remember these are cache blocks)
  - Pages can be loaded into any main memory location the processor chooses

# Virtual Memory

- Address Translation

Virtual addresses – created by the compiler and used by the processor



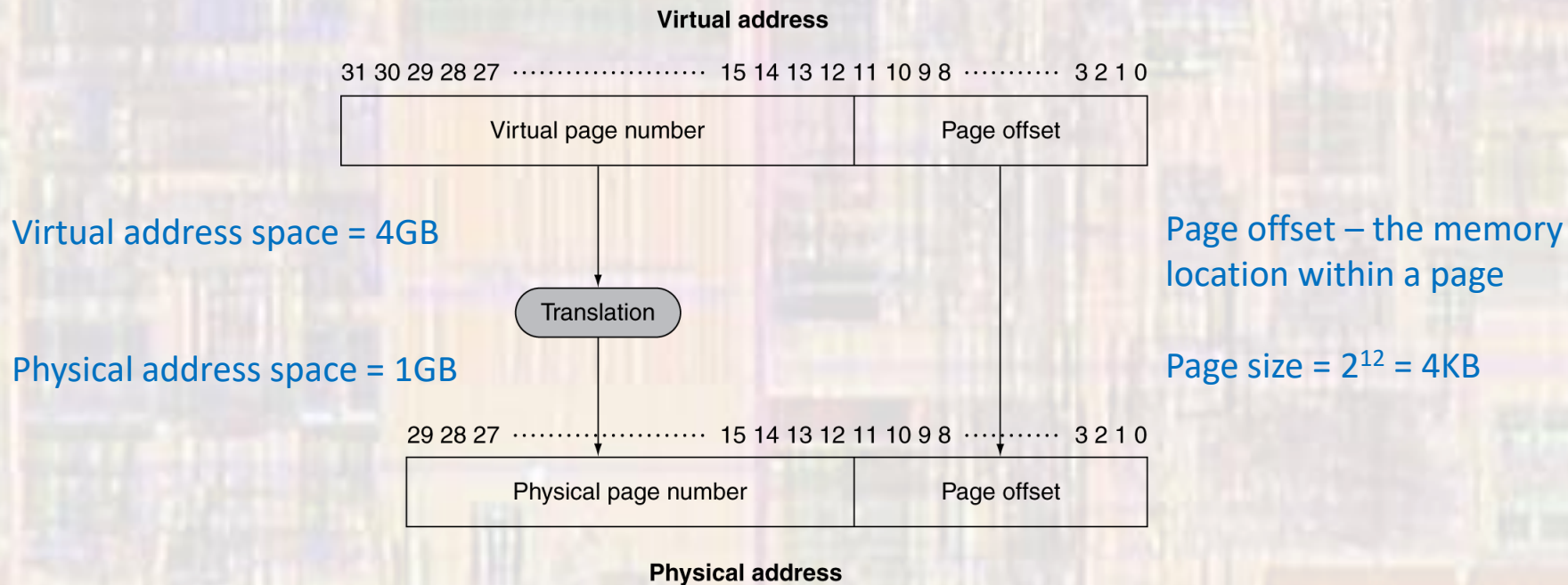
Physical addresses – The actual location of blocks (pages) in memory or on disk

- Overly simplistic example

Virtual address		Physical address
1234xxx	-->	dram 5463xxx
1235xxx	-->	dram 7638xxx
1236xxx	-->	hdd 1254xxx
1456xxx	-->	dram 5464xxx

# Virtual Memory

- Address Translation
  - Page offset is constant
  - Page must be translated

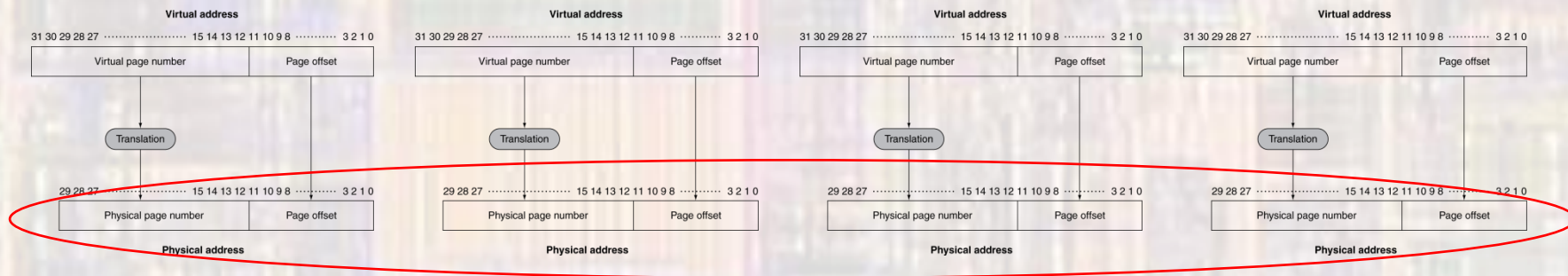




# Virtual Memory

- Address Translation

- Each program (process) will have its own virtual memory space



Only 1 physical memory

- But there is only 1 physical memory
- The VMM is responsible for physical memory management
  - Protects programs from overwriting each other
  - Allows programs to share

# Virtual Memory

- Page Fault
  - The processor addresses a given virtual address
    - If the address maps to the main memory – HIT
    - If the address maps to the secondary memory – Miss = page fault
  - Page fault - requires a page to be read from secondary memory
    - At 10ms access times (HDD) = 10M clock cycles to get the first byte when running at 1GHz
    - Need to transfer 4KB
  - Must reduce page faults to the lowest possible value

# Virtual Memory

- Page Fault
  - Approaches to reduce page fault impact
    - Page size – large enough to capture large portions of the program
      - 4KB, 16KB typical – but growing
      - 1KB for mobile – smaller apps
    - Fully associative placement
    - Use SW to manage page faults
      - Small overhead compared to the delay of a miss
      - Can be very sophisticated in placement and replacement
    - Use write-back
      - Only when necessary (dirty)

# Virtual Memory

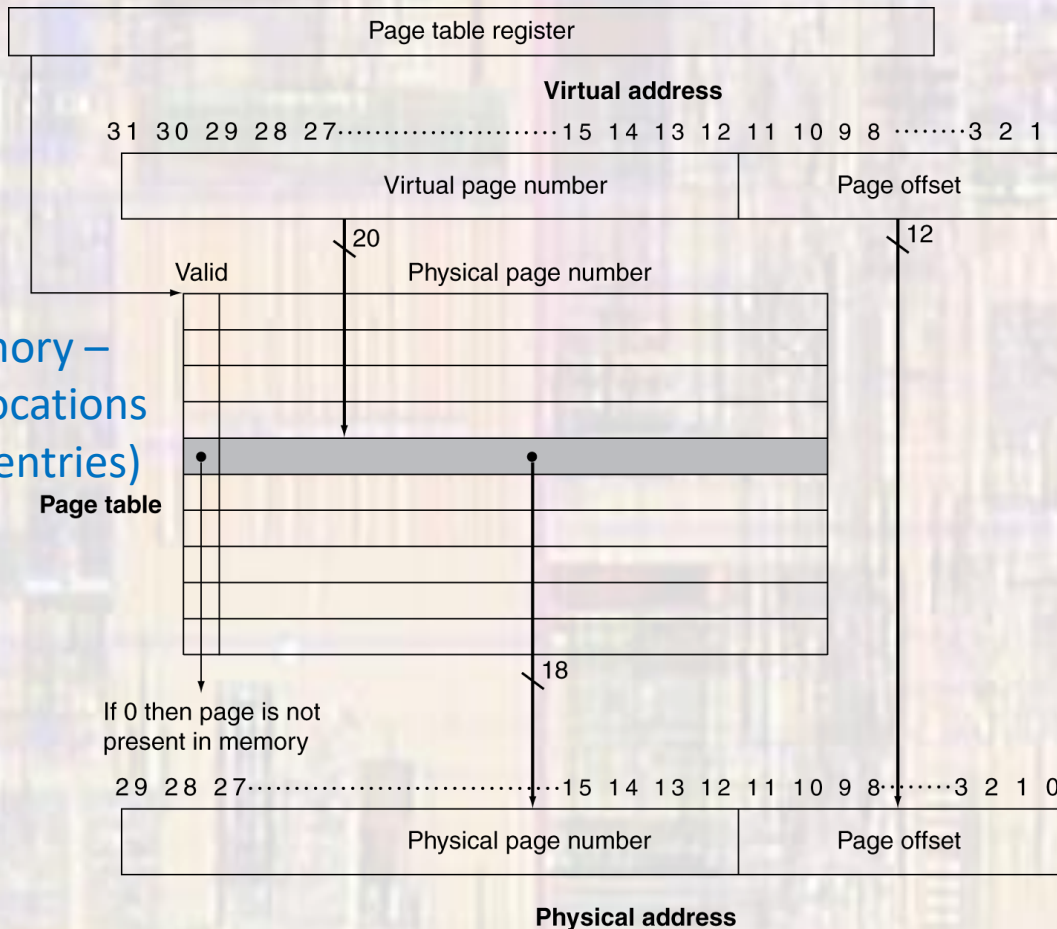
- Page Tables
  - Remember – fully associative placement is costly due to the time and circuitry needed to find the block (page) you are looking for
  - VM uses the existing memory addressing/decoding capability built into main memory
    - Each program (process) is allocated a space in main memory to place the address translation information for that process – Page Table
    - The processor includes a register to point to the location of the page table for the currently running program – Page Table Register
  - The page table maps each possible virtual memory location to a physical memory location

# Virtual Memory

- Page Tables

Starting address of the page table in main memory – process dependent

Main memory – memory locations ( $2^{20} = 2M$  entries)



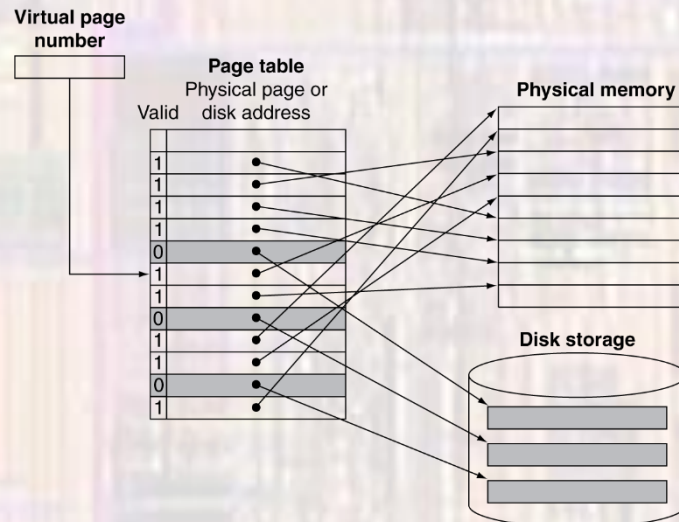
Address known to the program and the processor

Physical address to read from / write to

# Virtual Memory

- Page Tables

- Logical configuration



- In most systems – 1 page table but separate data structures
  - 1 for main memory
  - 1 for secondary memory
    - Always need to know where the page is in secondary memory – never changes

# Virtual Memory

- Page Fault - read
  - Page is not in main memory and the valid bit is NOT set in the page table
    - Processor throws an exception → gives control to the OS
    - OS finds the page on disk (using the HDD portion of the page table)
      - Located in the swap space
    - If necessary – the OS determines which page to write back to the HDD if the memory was full
      - Must be dirty
      - LRU is typical – but with a sophisticated process
      - Often include a reference bit (indicates the page has been referenced recently)
    - OS returns control to the program and re-executes the read
      - This time it finds the page in main memory

# Virtual Memory

- Page Fault - write
  - Page is not in main memory and the valid bit is NOT set in the page table
    - Processor throws an exception → gives control to the OS
    - OS finds the page on disk (using the HDD portion of the page table)
    - If necessary – the OS determines which page to write back to the HDD if the memory was full
    - OS reads the page in
    - OS returns control to the program and re-executes the write
    - Uses a write-back approach – only writes when it is swapped back out and dirty

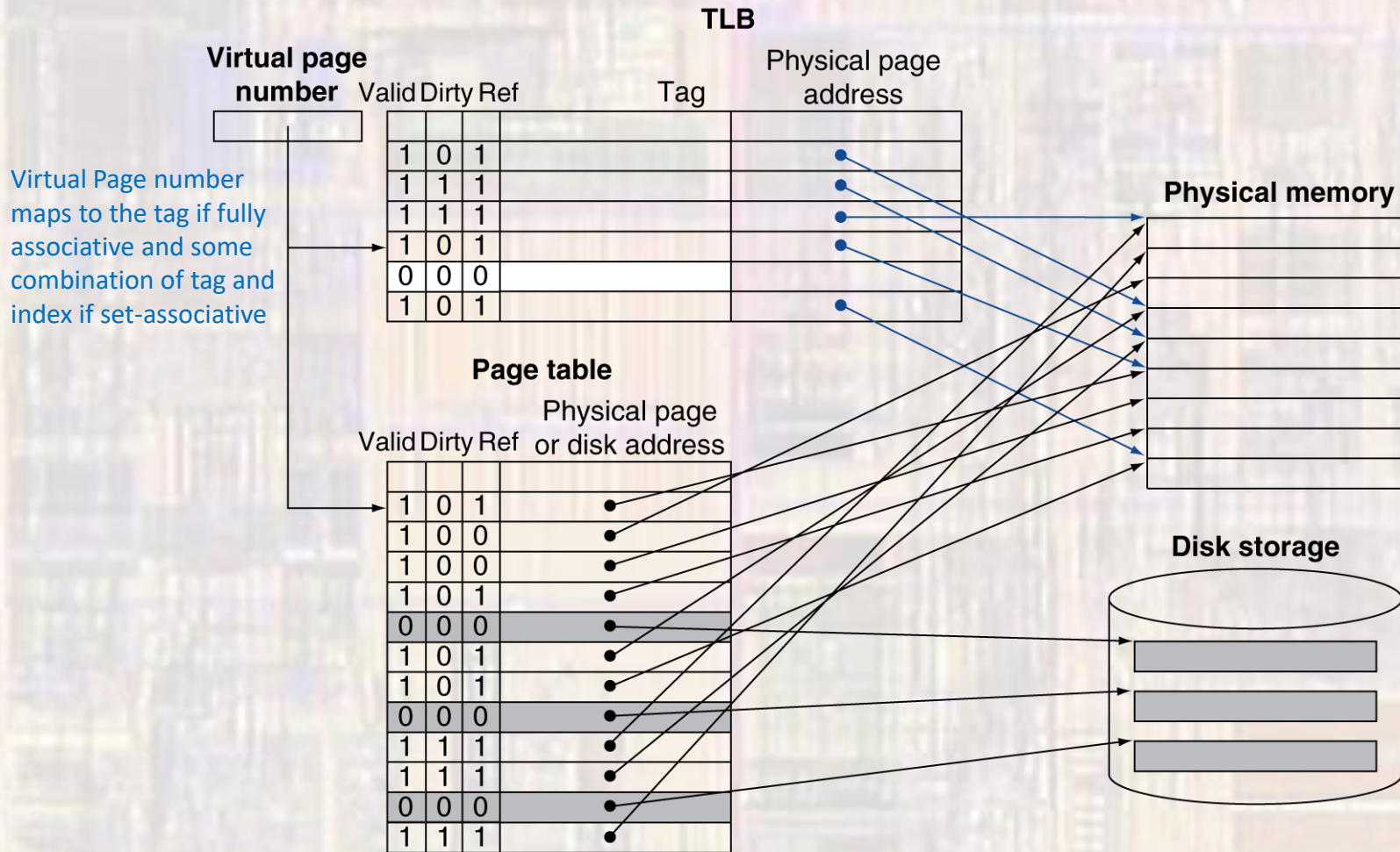


# Virtual Memory

- Page Table Issue
  - Every memory access requires 2 accesses
    - 1 – for the page table lookup (translation) – in memory
    - 1 – for the actual memory read/write – in memory
  - Fortunately – Pages have high temporal and spatial locality
  - Create a cache of the translation entries
    - Translation-Lookaside Buffer (TLB)

# Virtual Memory

- Translation-Lookaside Buffer



# Virtual Memory

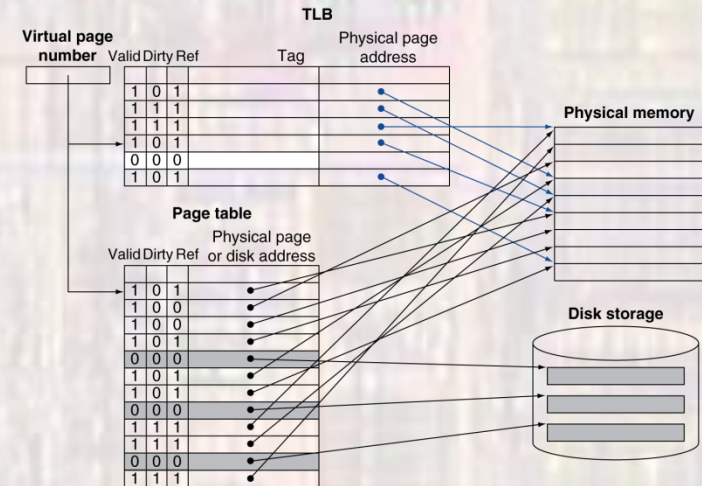
- TLB Hit

- Page is in main memory

- Tag matches
- Valid bit is 1

- Physical address is provided to the cache

- Ref bit is set
- If a write – dirty bit is set



# Virtual Memory

- TLB Miss

- Page is not in the TLB

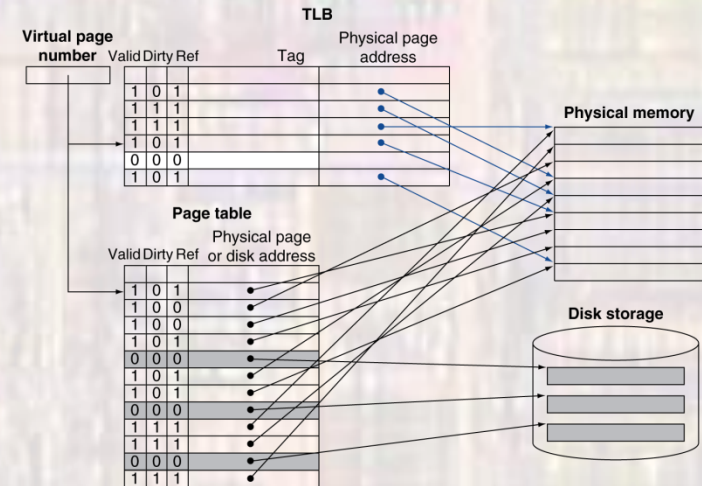
- Check the page table

- Hit

- load the page info into the TLB
  - Must have a replacement policy
- Provide the physical address to the cache

- Miss → Page Fault

- Throw an exception ...
- Once the page is loaded in memory and the entry added to the TLB – re-issue the read/write



# Virtual Memory

- TLB Miss

- Page is not in the TLB

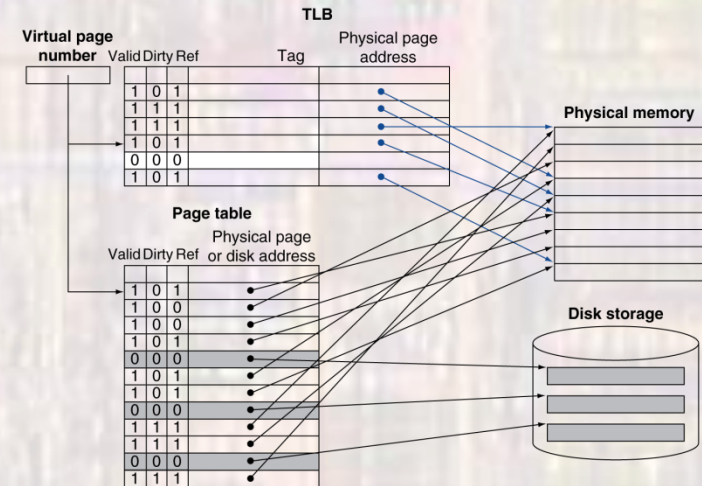
- Check the page table

- Hit

- load the page info into the TLB
  - Must have a replacement policy
- Provide the physical address to the cache

- Miss → Page Fault

- Throw an exception ...
- Once the page is loaded in memory and the entry added to the TLB – re-issue the read/write



# Virtual Memory

- TLB

