# CS161 – Design and Architecture of Computer Systems

## Week 5 - Discussion

*some slides adapted from:

Prof Daniel Wong UCR – EE/CS)

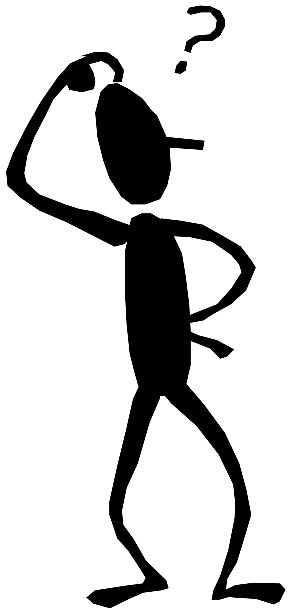
UNIVERSITY OF CALIFORNIA, RIVERSIDE

## Memory vs Storage

- Disk (aka: Secondary Storage):
  - used to store data of all your programs and everything else on the computer
  - often referred to as (when buying a computer):
    - "Storage"
    - "Hard Drive"
    - "Solid State Drive (SSD)"
    - "Flash memory" (for phones)
- Main memory:
  - used to hold programs while they are running
  - can be thought of as a "cache" for disk storage
  - often referred to as (when buying a computer):
    - "Memory"
    - "RAM"

## Memory vs Storage

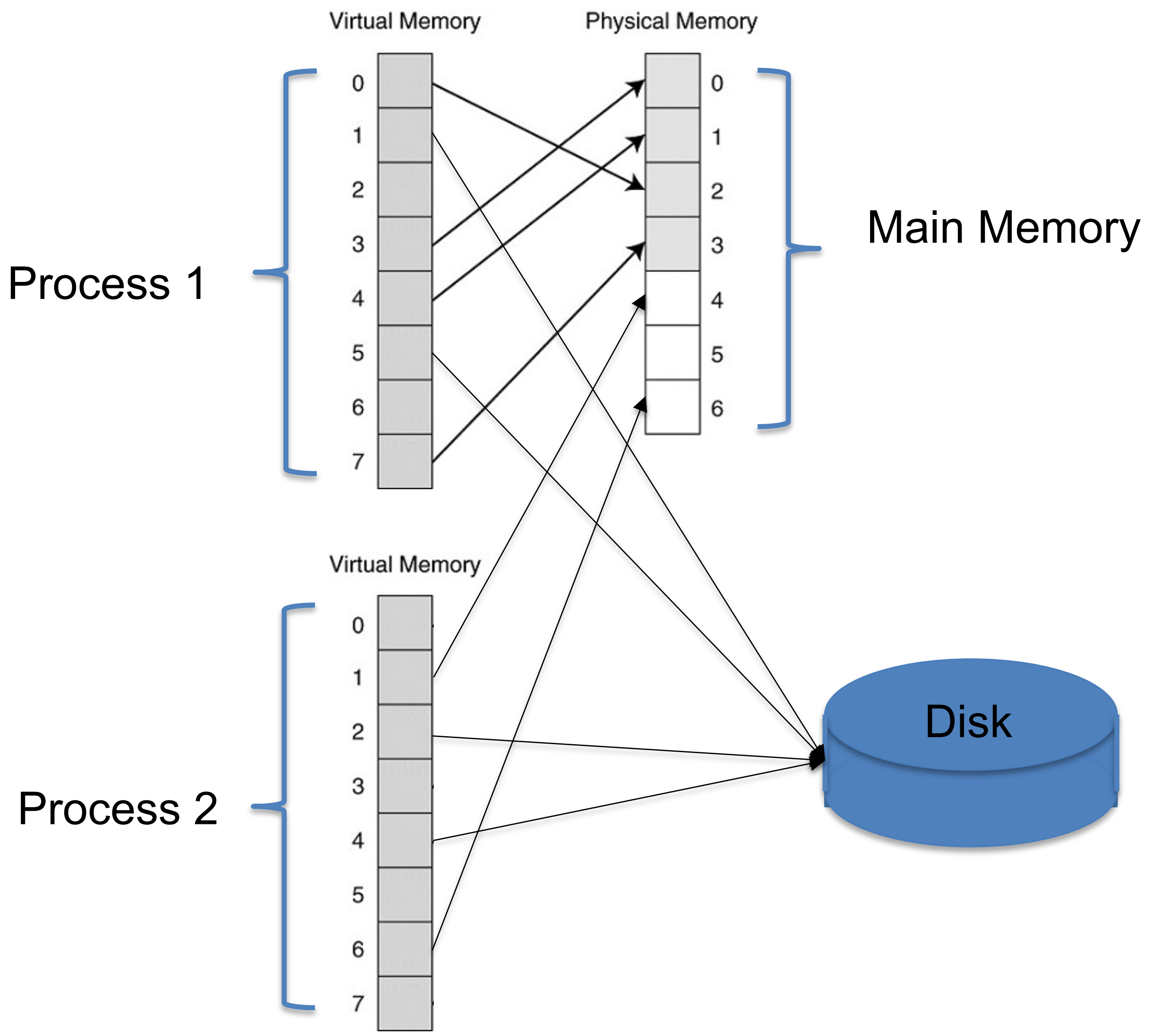
- Main memory:
  - used to hold programs while they are running
  - so programs are limited by size of memory??
  - and how do we switch programs running in memory??



## Virtual Memory: Big Picture

- Each program thinks it has entire computer disk to run on ("Virtual Memory")
- However, each program runs in shared main memory ("Physical Memory")
- But: main memory << disk
- And: multiple programs running at once (sharing main memory)
- So we have to convert virtual memory to actual physical memory or actual disk
- Thus: main memory is a "cache" for disk

# Shared Physical (Main) Memory



## Pages

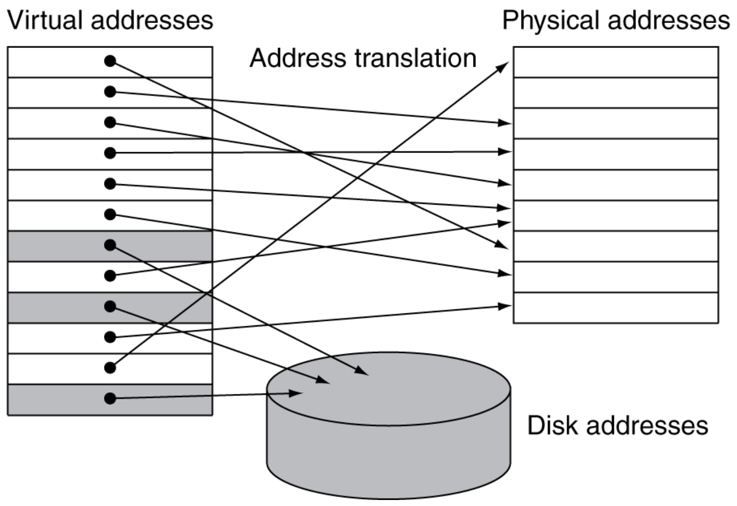
- Divide memory into equal sized “chunks” or pages (i.e., 4KB each)
- Any chunk of Virtual Memory can be assigned to any chunk of Physical Memory

- A page
  - a fixed-length contiguous block of virtual memory, described by a single entry in the page table
  - It is the smallest unit of data for memory management in a virtual memory operating system.
- page frame
  - the smallest fixed-length contiguous block of physical memory into which memory pages are mapped by the operating system.

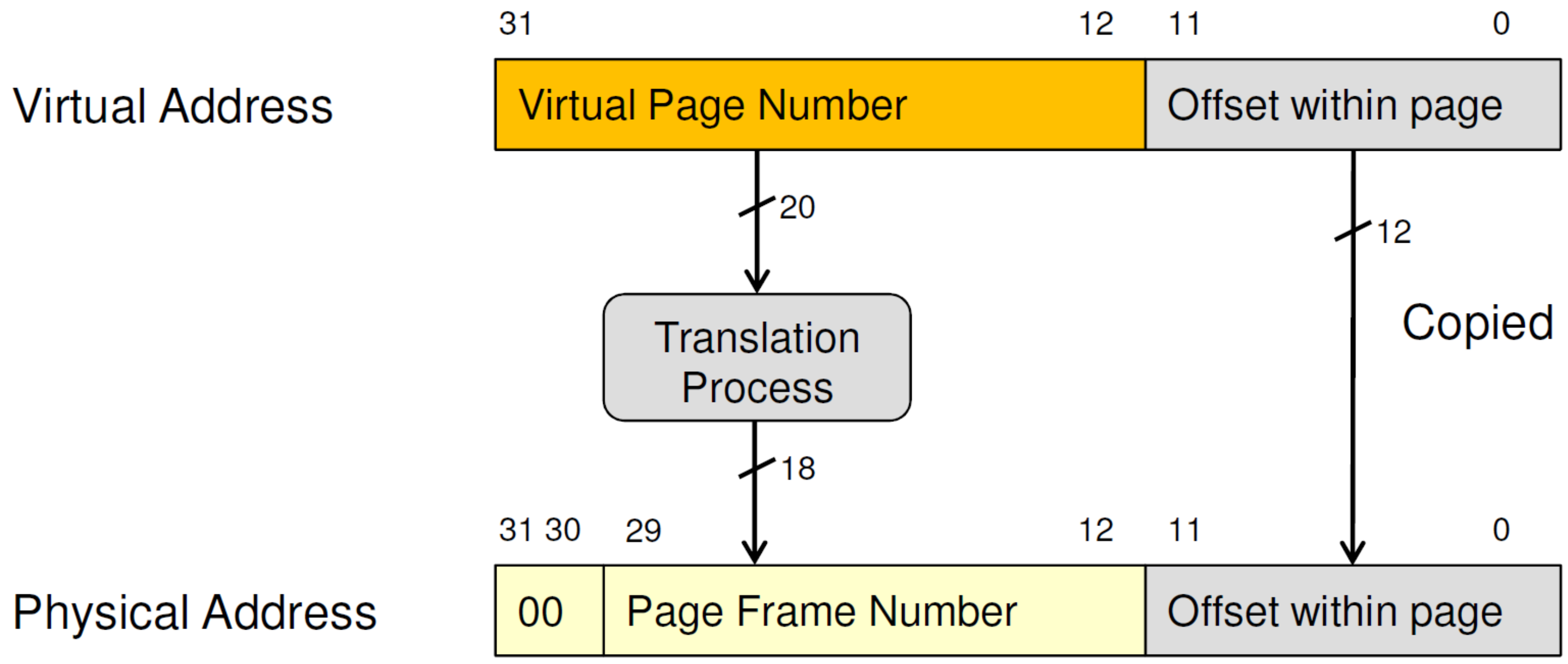
- A transfer of pages between main memory and an auxiliary store, such as a hard disk drive, is referred to as paging or swapping

# Page Fault

- Page missing in Main Memory
- Pages not in Main Memory are on disk:
  - swap-in/out

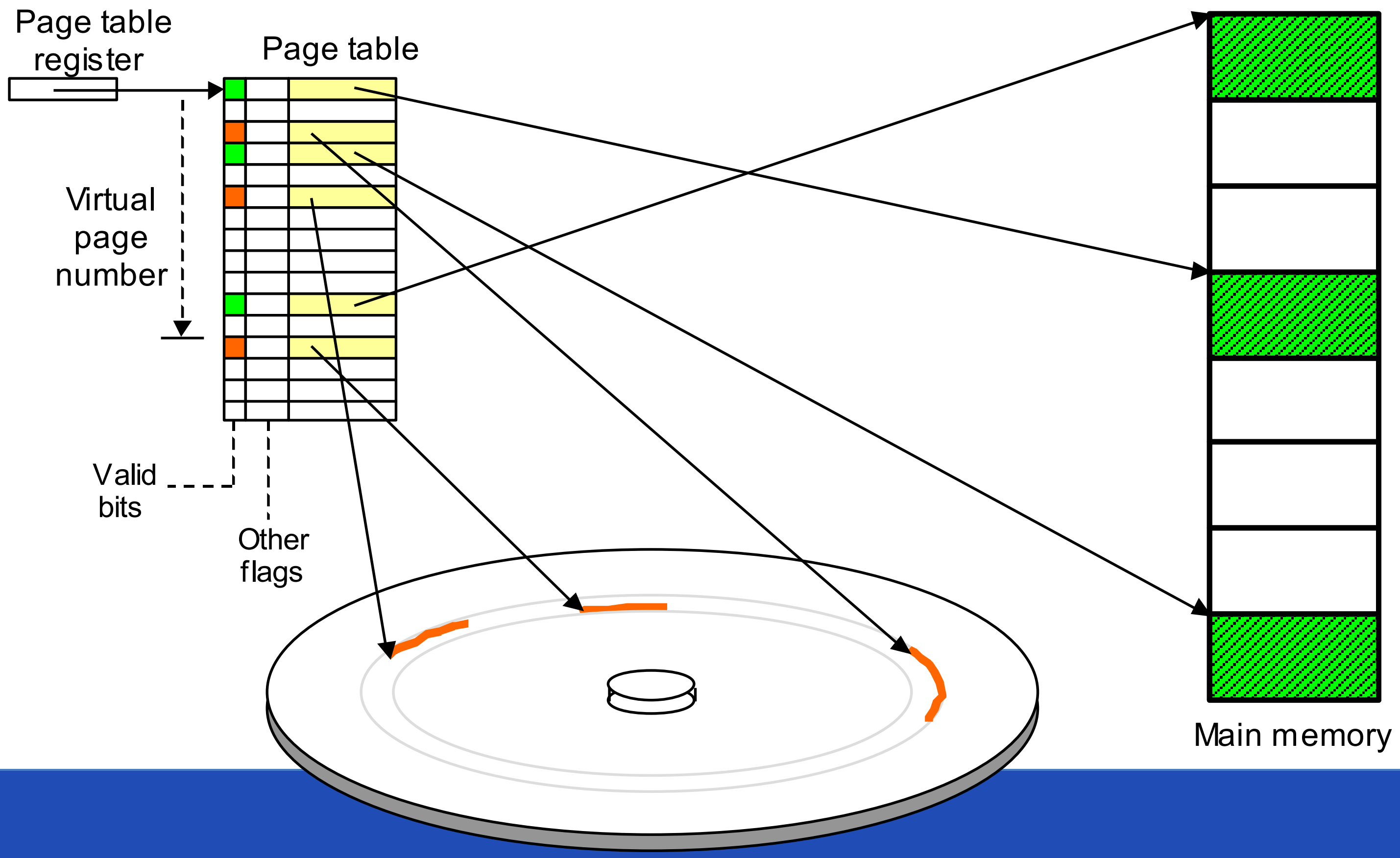


# Mapping Virtual to Physical Address



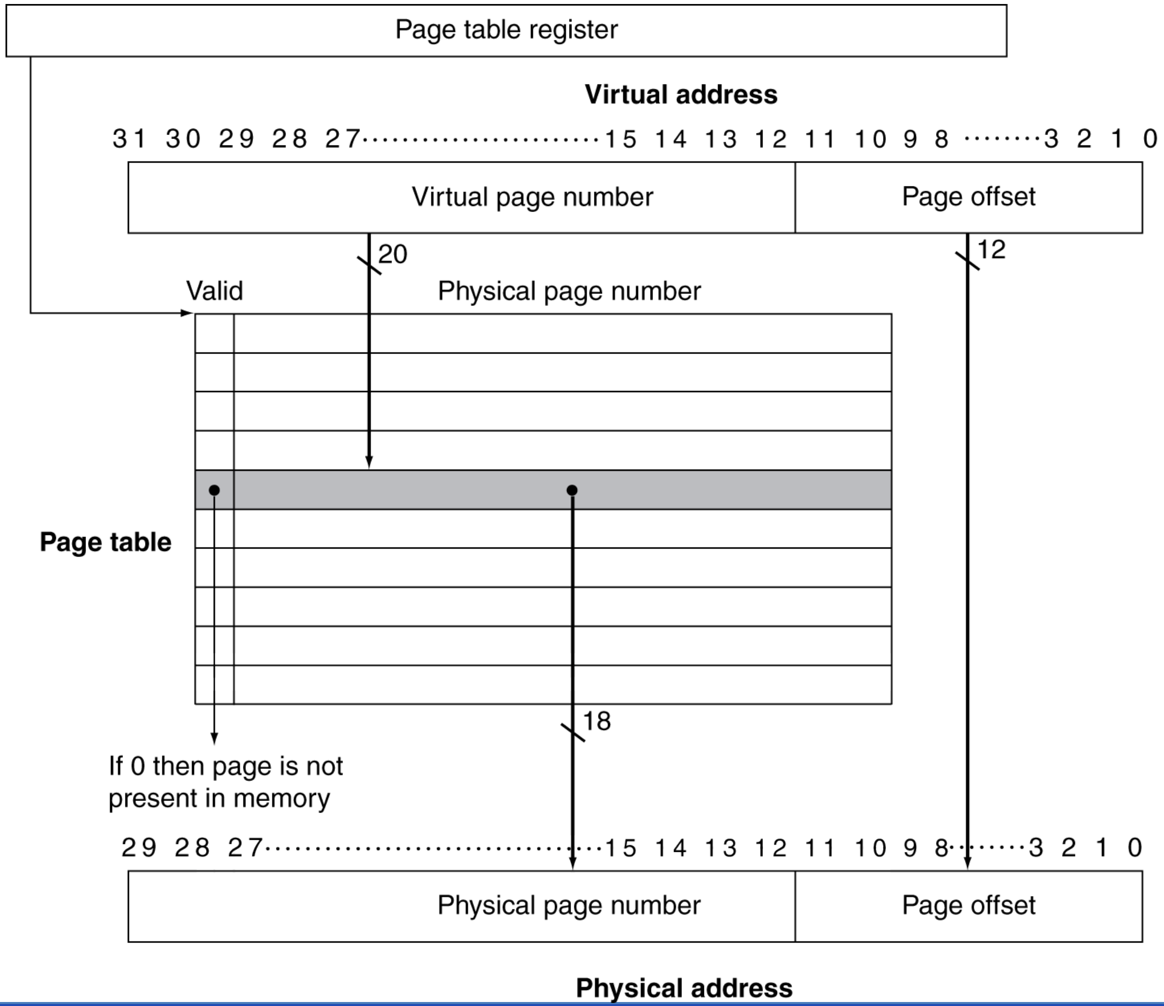
# Address Translation: Page Table

- A page table is a
  - data structure containing mapping of virtual to physical pages
  - each process has its own page table

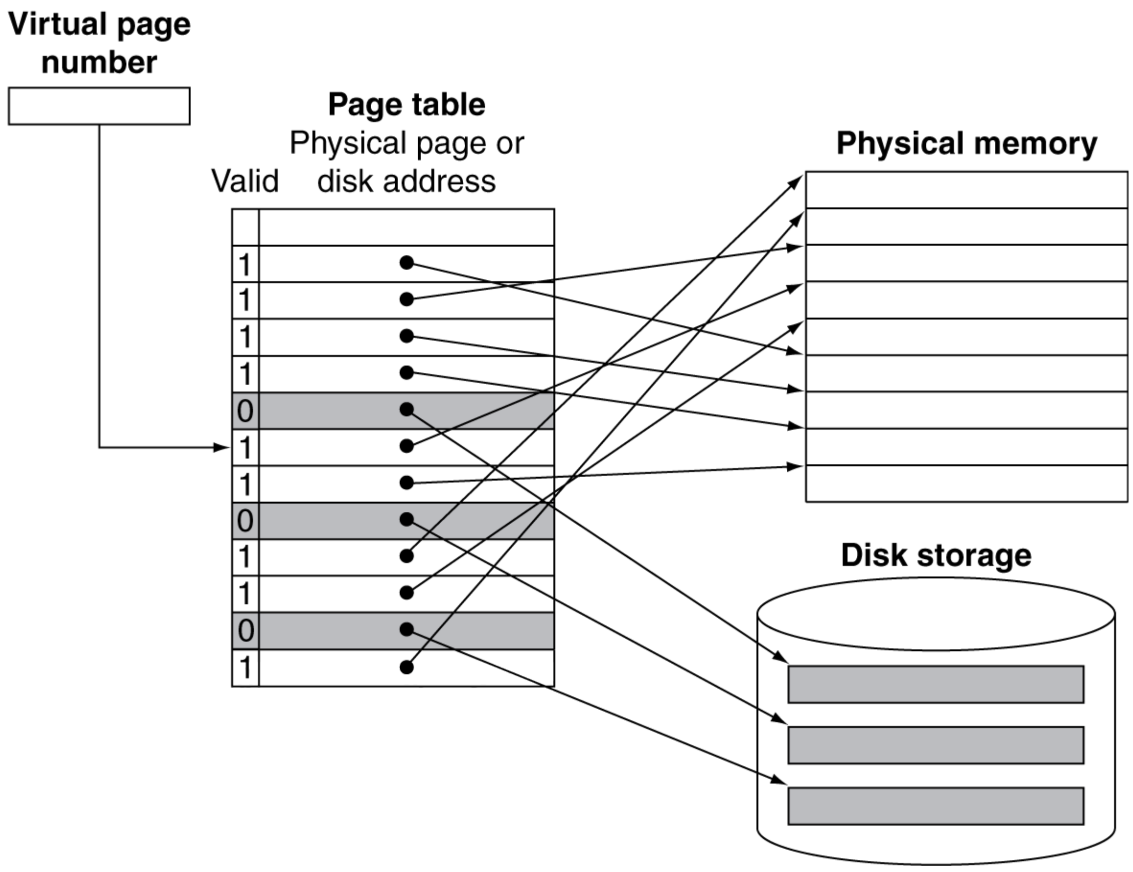


# Page Table

- Page table translates address



# Mapping Pages to Storage



# Optimizing VM

- Page Table too big!
  - virtual address space = 4GB= $2^2 2^{30}$ B = $2^{32}$ $B$
  - page size = 4 KB = $2^2 2^{10}$ B = $2^{12}$ $B$
  - PTE size = 4 B
    - number page table entries = $2^{32}$ - $2^{12}$ = $2^{20}$
    - **Page table size = 4 B * $2^{20}$ = 4MB**

      (just for Page Table of single process)
- Virtual Memory too slow!
  - Requires two memory accesses.
    - One to access page table to get the memory address
    - Another to get the real data

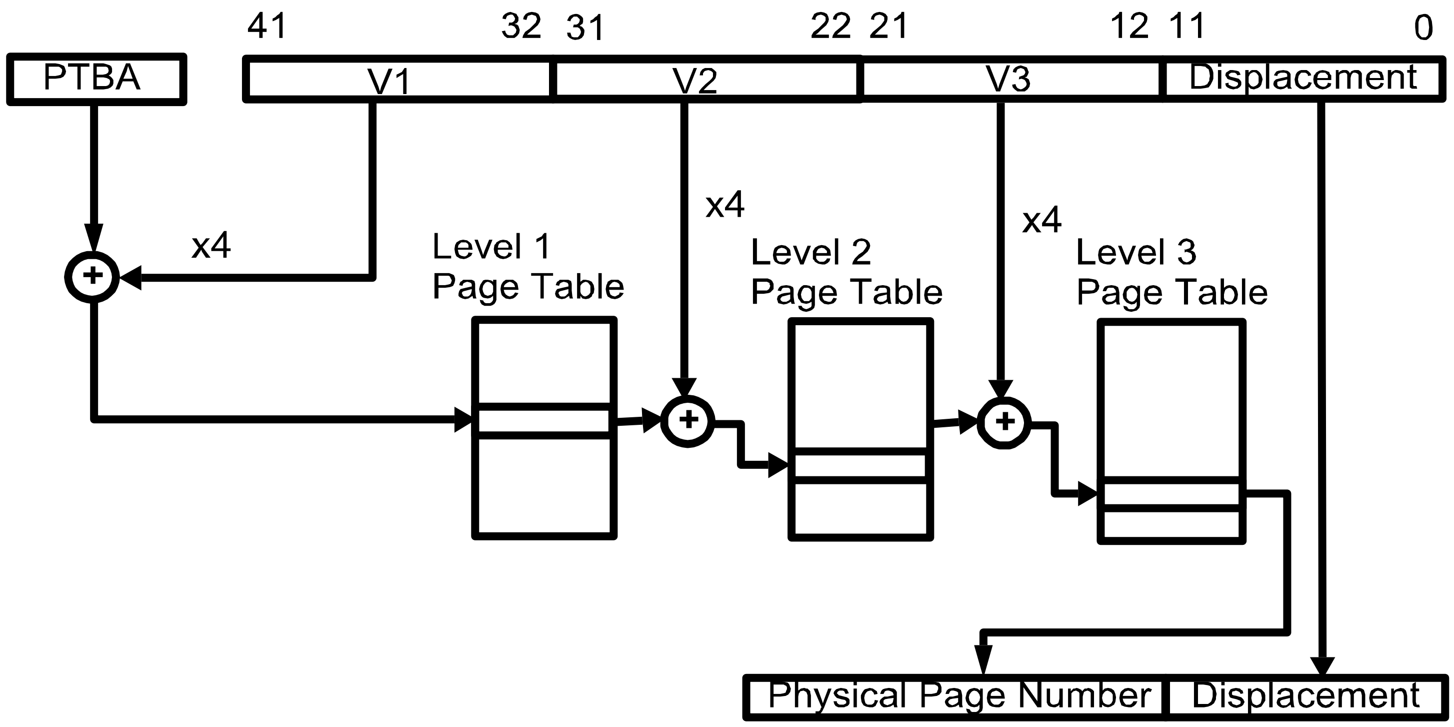
# Multi-level Page Table

- Problem: 1-level page table too expensive for large virtual address space
  - Need to reduce the size of the page table

- Solution: Multi-level page table, Paging page tables, etc.
  - To create small page tables for virtual memory
  - The virtual address is now split into multiple chunks to index a page table “tree”

# Multi-level Page Table

- Virtual page number broken into fields to index each level of multi-level page table



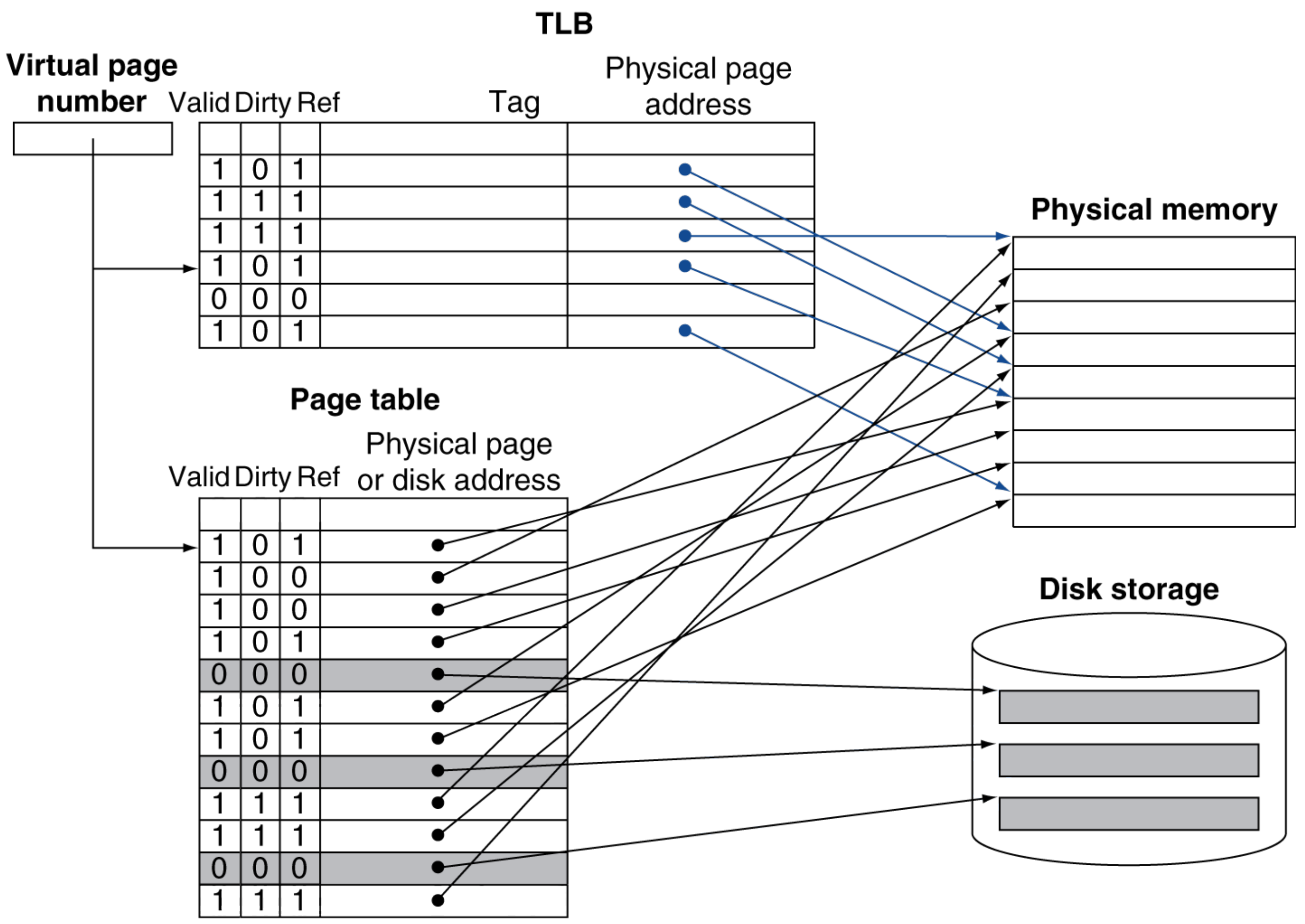
# Fast Address Translation

- Problem: Virtual Memory requires two memory accesses!
  - one to translate Virtual Address into Physical Address (page table lookup)
  - one to transfer the actual data (cache hit)
  - But Page Table is in physical memory! => 2 main memory accesses!

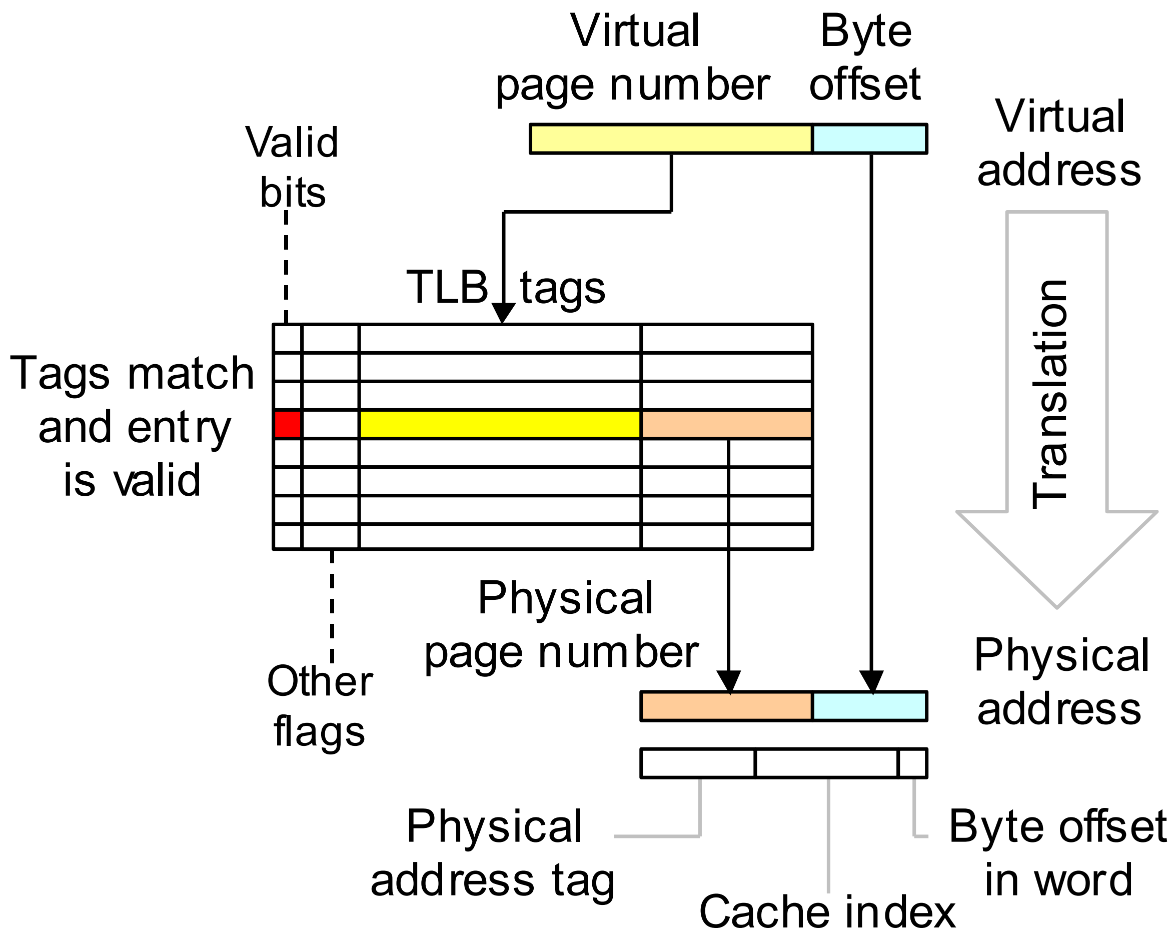
- Solution: Why not create a cache of virtual to physical address translations to make translation fast? (smaller is faster)

- For historical reasons, such a "page table cache" is called a Translation Lookaside Buffer, or TLB

# Fast Translation Using a TLB

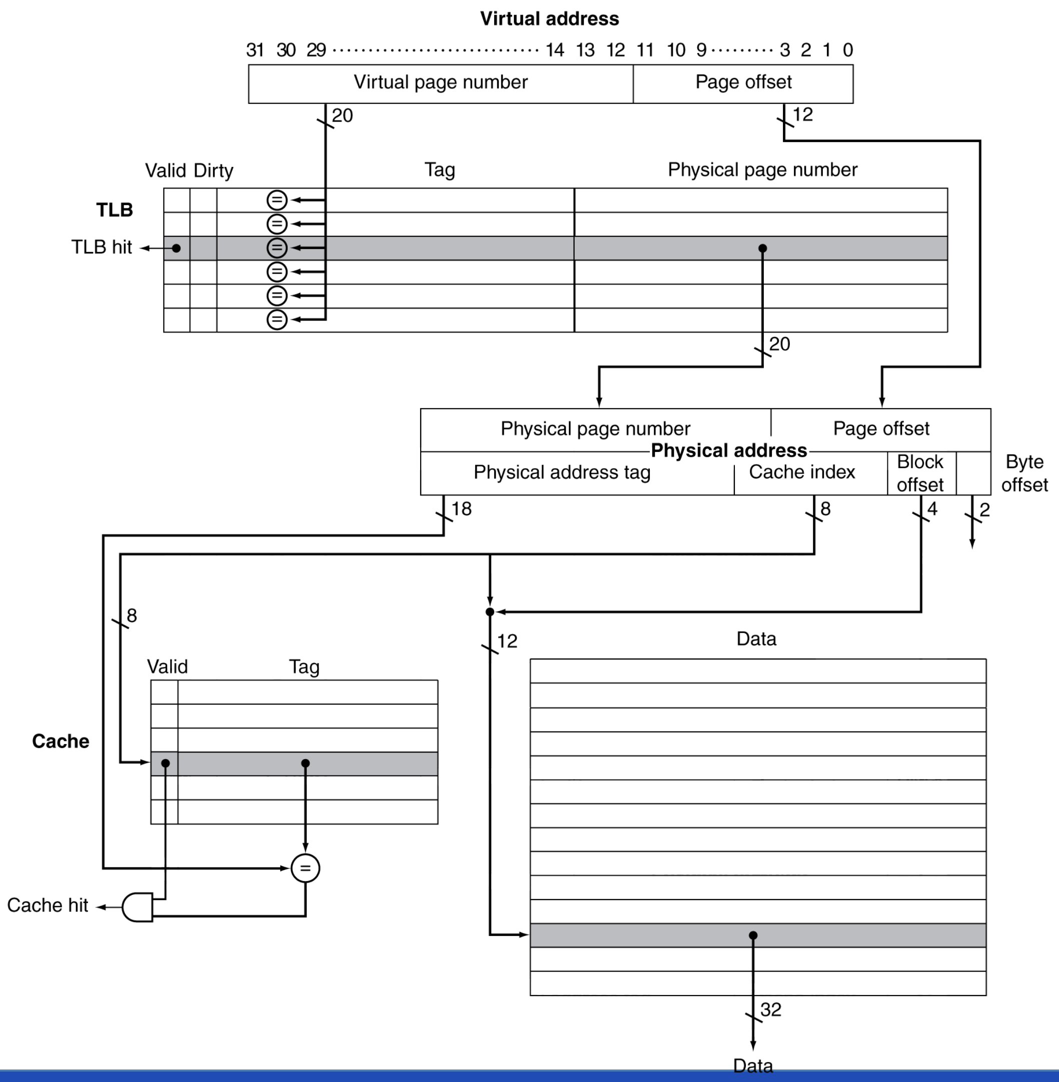


# TLB Translation



Virtual-to-physical address translation by a TLB and how the resulting physical address is used to access the cache memory.

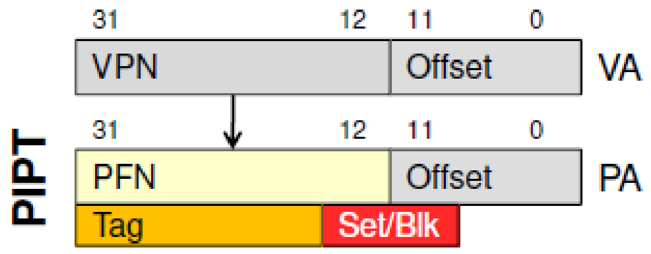
# TLB and Cache Interaction

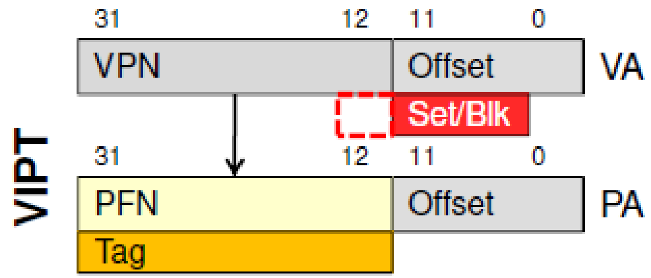


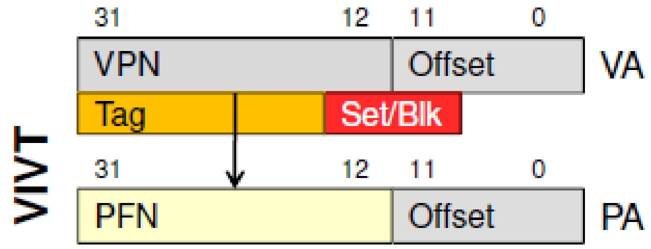
- If cache tag uses physical address
    - Need to translate before cache lookup
- Physically Indexed, Physically Tagged

# Cache Index/Tag Options

- Physically indexed, physically tagged (PIPT)
  - Wait for full address translation
  - Then use physical address for both indexing and tag comparison
- Virtually indexed, physically tagged (VIPT)
  - Use portion of the virtual address for indexing then wait for address translation and use physical address for tag comparisons
  - Easiest when index portion of virtual address w/in offset (page size) address bits, otherwise aliasing may occur
- Virtually indexed, virtually tagged (VIVT)
  - Use virtual address for both indexing and tagging…No TLB access unless cache miss
  - Requires invalidation of cache lines on context switch or use of process ID as part of tags







## Summary

- Virtual Memory overcomes main memory size limitations
- VM supported through Page Tables
- Multi-level Page Tables enables smaller page tables in memory
- TLB enables fast address translation

## Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

a) For a single-level page table, how many page table entries (PTEs) are needed?

# Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

a) For a single-level page table, how many page table entries (PTEs) are needed?

virtual address = 43 bits

page size = 4 KB = $2^2 2^{10}$B = $2^{12}$B ➔ offset bits = 12

virtual page number (VPN) bits = virtual address bits – offset bits

= 43 – 12

= 31

entries in page table = $2^{31}$ = $2^1 2^{30}$ = 2 G entries

## Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

b) How much physical memory is needed for storing the page table?

# Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

b) How much physical memory is needed for storing the page table?

page table size = (PTE size in bytes)*(number entries in page table)

= (2 G) * (4 B)

= $(2^{31})$ * $(2^{2}$B)

= $2^{33}$B

= $2^{3}2^{30}$B

= 8 GB

## Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

c) What is the size of secondary storage?

# Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

c) What is the size of secondary storage?

secondary storage size (in bytes) = $2^{vitual\ address\ bits}$ bytes

= $2^{43}$ bytes

= $2^{3}2^{40}$ bytes

= 8 TB

## Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

d) How many bits of physical address are needed to address DRAM (main memory)?

## Example 1

1. The following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

d) How many bits of physical address are needed to address DRAM (main memory)?

physical DRAM size = $2^{physical\ address\ bits}$ bytes

= 16 GB

= $2^4 2^{30}$ B

= $2^{34}$ B. ➔ 34 bits needed

# Example 2

2. Using a multi-level page table with the following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

a) How many levels of page tables will be needed in this case?

# Example 2

2. Using a multi-level page table with the following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

a) How many levels of page tables will be needed in this case?

number of page table entries that fit in a page = $\frac{page\ size}{PTE\ size} = \frac{4\ KB}{4\ B}$ = 1 K

= $2^{10}$ entries in a page

➔ so we need 10 bits to find the PTE in a page

number of levels = $\frac{bits\ of\ single\ level\ page\ table\ entries}{bits\ of\ multilevel\ page\ table\ entries} = \frac{31}{10}$ = 3.1 = 4 levels

# Example 2

2. Using a multi-level page table with the following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

b) How many memory references are needed for address translation if missing in TLB?

## Example 2

2. Using a multi-level page table with the following list provides parameters of a virtual memory system:

| Virtual Address (bits) | Physical DRAM Installed | Page Size | PTE Size (bytes) |
|---|---|---|---|
| 43 | 16 GB | 4 KB | 4 |

b) How many memory references are needed for address translation if missing in TLB?

if there is a miss in the TLB for address translation, then 4 memory references are required, one for each level

# Example 3

3. Given the following parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (i.e., a [12,8] encoding):

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

a) decode the bits for the value 0x9A.

# Example 3

3. Given the following parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (i.e., a [12,8] encoding):

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

a) decode the bits for the value 0x9A.

0x9A = 1001 1010 = 10011010

So there's eight data bits (8) and we're using 12 bit parity encoding, so we will have 4 bits for parity bits, which we will put in spots 1, 2, 4, and 8, so we leave a blank there for now:

_ _1_001_1010

**p1** checks every other spot (1 position in between), so positions 1,3,5,7,9, and 11, the positions are highlighted in red:

_ _1_001_1010

since there are four 1's highlighted, then we should set the first bit (i.e., the **p1** bit) to 0 to keep the group's parity at an even number:

0_1_001_1010

# Example 3

3. Given the following parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (i.e., a [12,8] encoding):

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

a) decode the bits for the value 0x9A.

next we look at the case for **p2**, checking every two bits (highlighted in red), so we're checking positions 2,3,6,7,10,and 11:

0_1_001_1010

since there are three 1's highlighted, then we should set the second bit (i.e., the **p2** bit) to 1 to make the group's parity at an even number:

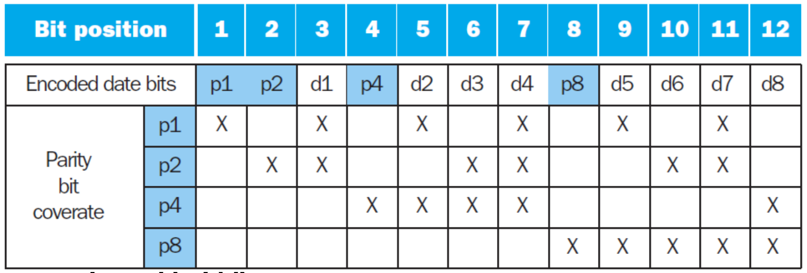
011_001_1010

next we look at the case for **p4**, checking every four bits (highlighted in red), so we're checking positions 4,5,6,7,and 12:

011_001_1010

# Example 3

3. Given the following parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (i.e., a [12,8] encoding):



| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

a) decode the bits for the value 0x9A.

since there is only one 1 highlighted, then we should set the fourth bit (i.e., the **p4** bit) to 1 to make the group's parity at an even number:

0111001_1010

finally, we look at the case for **p8**, checking every eight bits (highlighted in red), so we're checking positions 8,9,10,11,and 12:

0111001_1010

since there are two 1's highlighted, then we should set the eighth bit (i.e., the **p8** bit) to 0 to keep the group's parity at an even number:

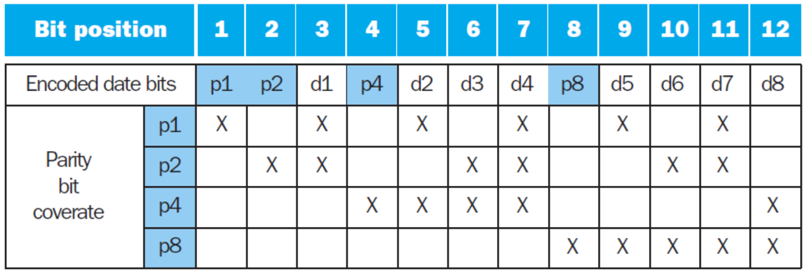
011100101010

this leaves us with the following final code word:

011100101010

# Example 3

3. Given the following parity bits, data bits, and field coverage in a Hamming ECC code for eight data bits (i.e., a [12,8] encoding):



| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded date bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 |
| Parity bit coverate | p1 | X | | X | | X | | X | | X | | X | |
| | p2 | | X | X | | | X | X | | | X | X | |
| | p4 | | | | X | X | X | X | | | | | X |
| | p8 | | | | | | | | X | X | X | X | X |

a) decode the bits for the value 0x9A.

but let's assume that the following was received instead (bit 10 was flipped):

011100101110

so to check if there was an error, we would check the parity and compare them with their respective parity bits:

p1: 011100101110 → four 1's (even) without parity bit and parity bit is 0 (even) → no error
p2: 011100101110 → four 1's (even) without parity bit and parity bit is 1 (odd) → error
p4: 011100101110 → one 1's (odd) without parity bit and parity bit is 1 (odd) → no error
p8: 011100101110 → three 1's (odd) without parity bit and parity bit is 0 (even) → error

since parity bits 2 (p2) and 8 (p8) are incorrect, then bit 10 must be wrong (2+8=10)

011100101110 → 011100101010