23. Virtual Memory

Department of Computer Science
University of Victoria

MH: 7.6

HVZ: 6

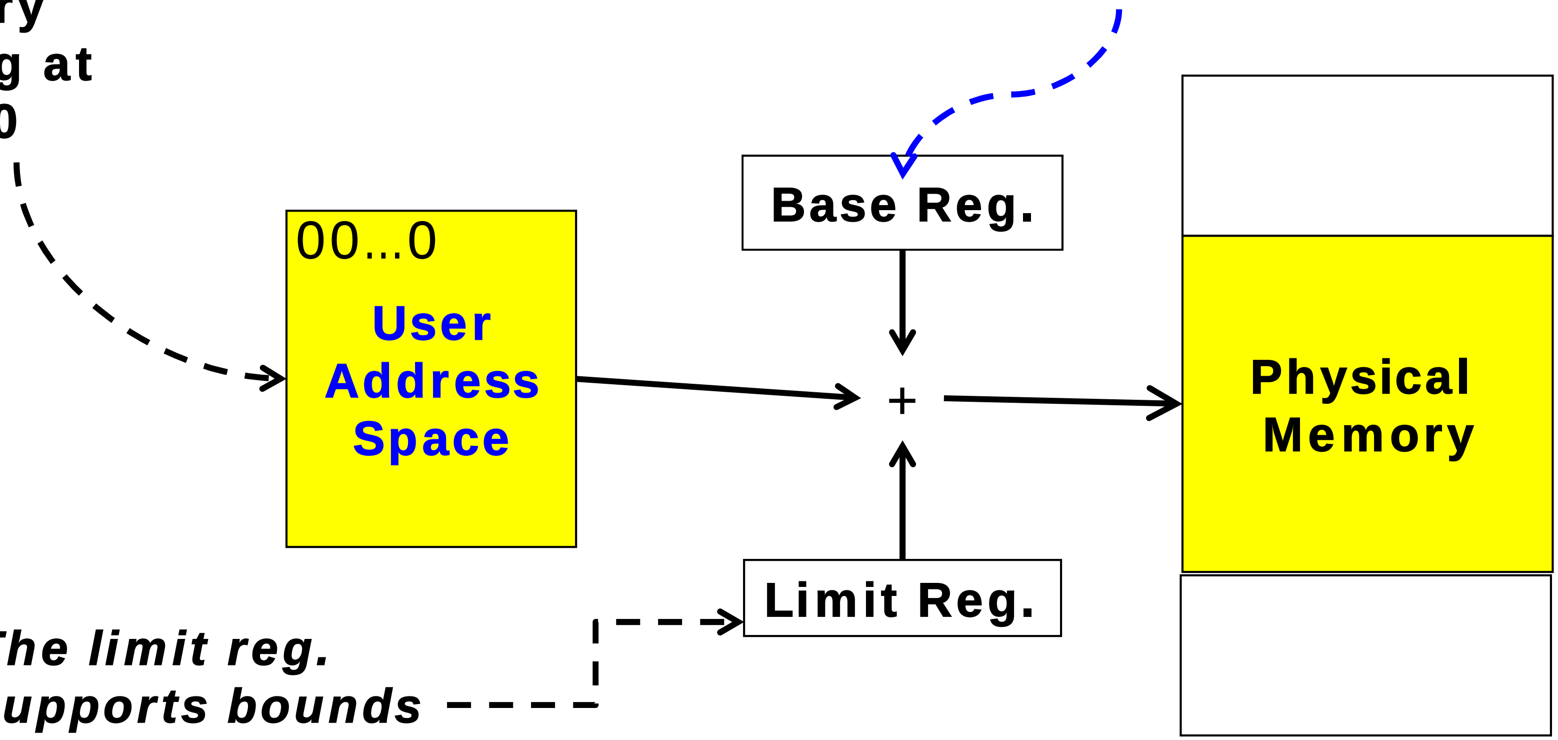
Stallings: 8.3, 8.5

(Some of the textbook subsections are more OS-oriented, but it will be helpful to you to read through them)

PROGRAM RELOCATABILITY

In multiple user systems, each user program 'appears' to use a contiguous block of memory beginning at location 0

Hardware maps user address space to the physical address space using registers that can only be set by the operating system



The limit reg. supports bounds checking

➔ This scheme provides easy program relocation

What is Virtual Memory?

Virtual memory is a memory management technique which virtualizes all storage system and makes them appear as one large memory device

The side effect is that the main (physical) memory can be seen to act as a cache for the secondary storage

Why Virtual Memory?

Efficient and safe sharing of memory between multiple programs

- Protection
- Active portion used is only a fraction

Decrease problems of having small amount of real memory

Reduce reliance on clever programming

What does Virtual Memory do?

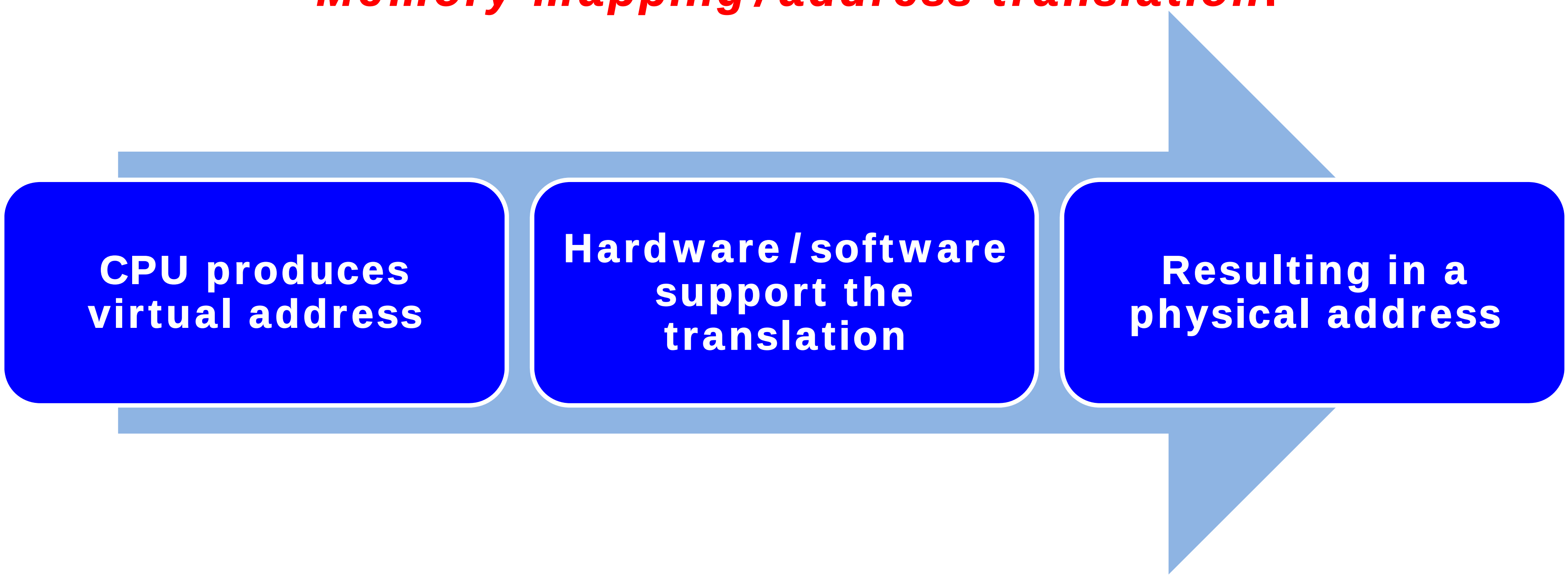
- Give an illusion of an essentially unbounded amount of memory
- Allow efficient and safe sharing of memory among multiple programs
- Remove the programming burdens of a small, limited amount of main memory
- Provide relocation, which simplifies loading the program for execution (programs can be loaded into any location)

How does Virtual Memory do it?

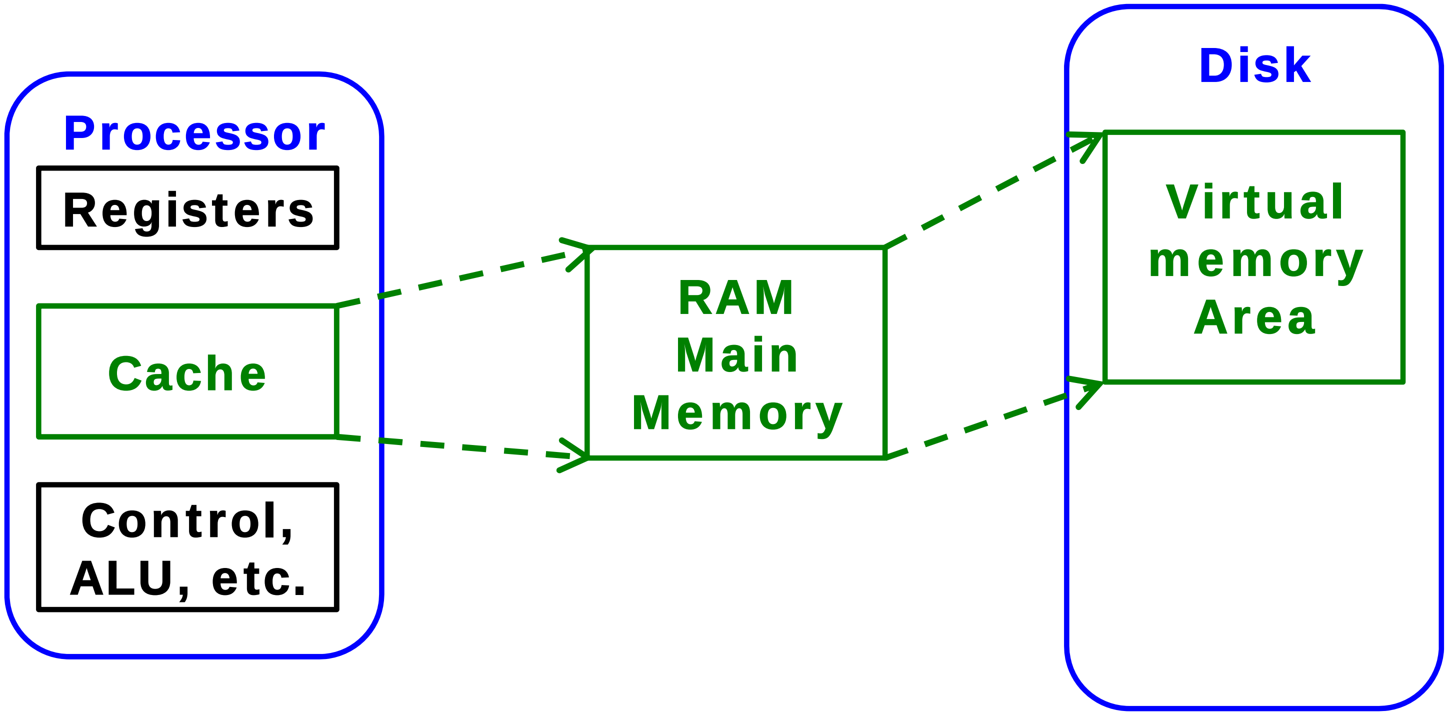
1. Each program is compiled to its address space

2. Virtual memory implements the translation of a program's address space to physical addresses

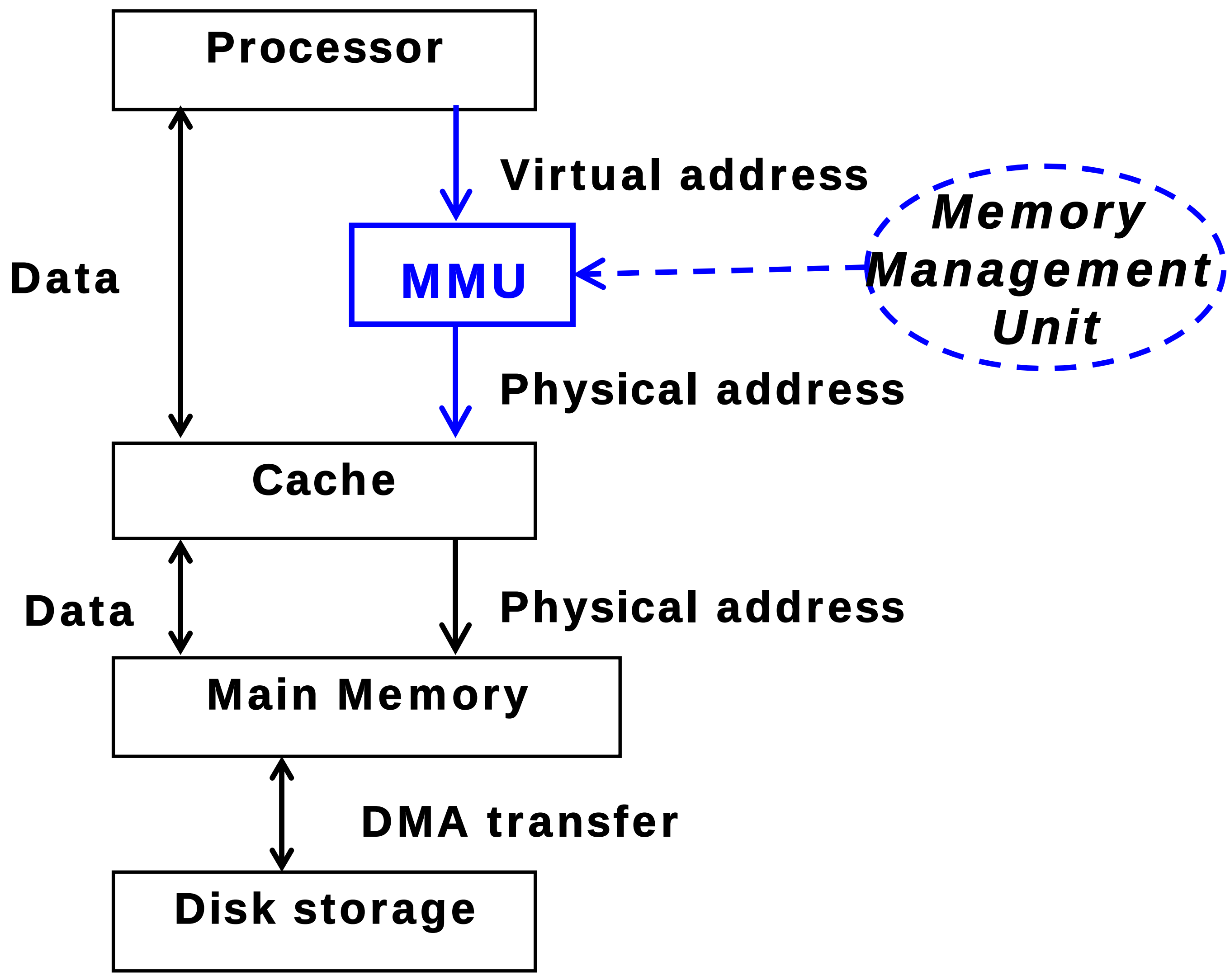
Memory mapping/address translation:



Overview


Processor
Registers
Cache
Control,
ALU, etc.
RAM
Main
Memory
Disk
Virtual
memory
Area

Virtual memory organization



How can programs make effective and efficient use of such large address space?

- ❑ **Not long ago, main memory (RAM) was measured in MB or 1-2 Gb**
- ❑ **But theoretical address space is *much* larger**
  - ➔ **32-bit addresses can represent 4GB of byte-addressable space**
  - ➔ **33-bit addresses can represent 8GB of byte-addressable space**

Observation:

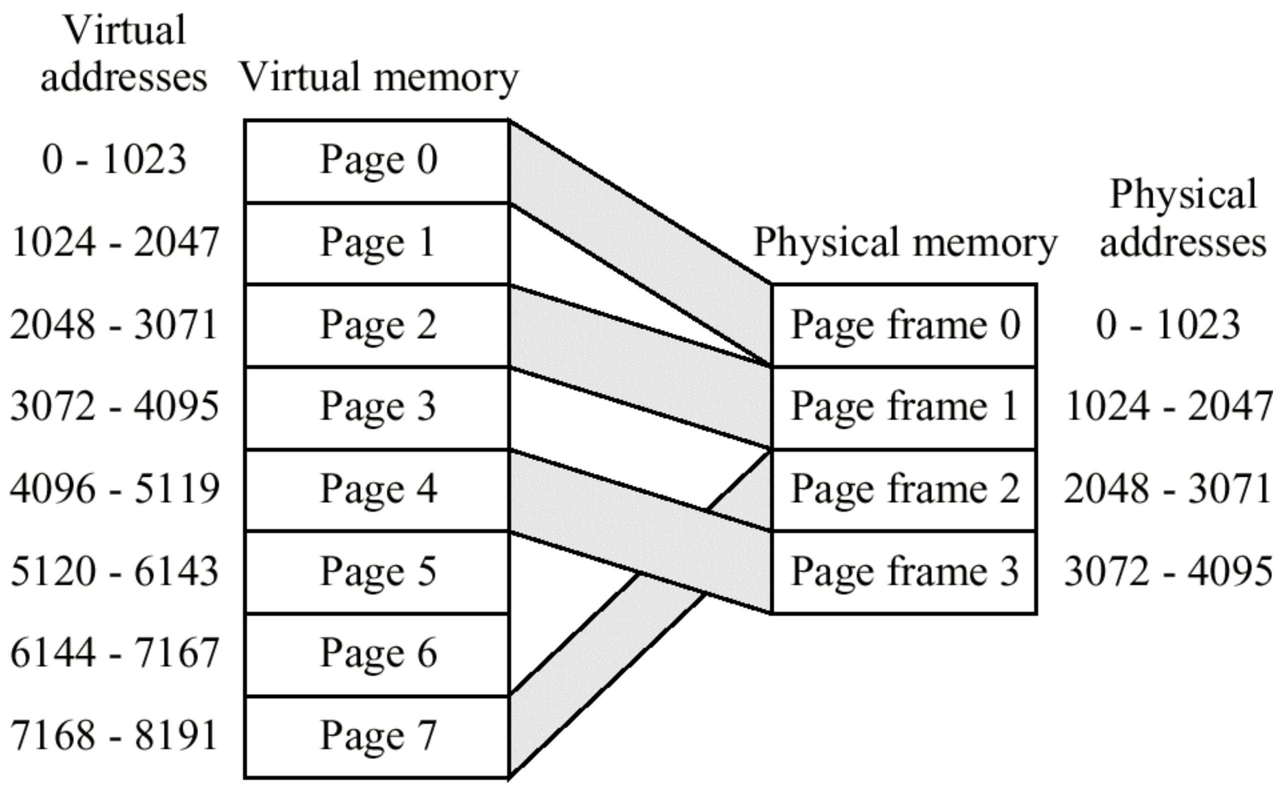
Few programs need all their code and data at once. In fact, most code and data are accessed rarely: the 90/10 or 80/20 "rules".

➔ Solution:

- **Only use RAM for "active" code and data.**
- **Store the rest on disk.**

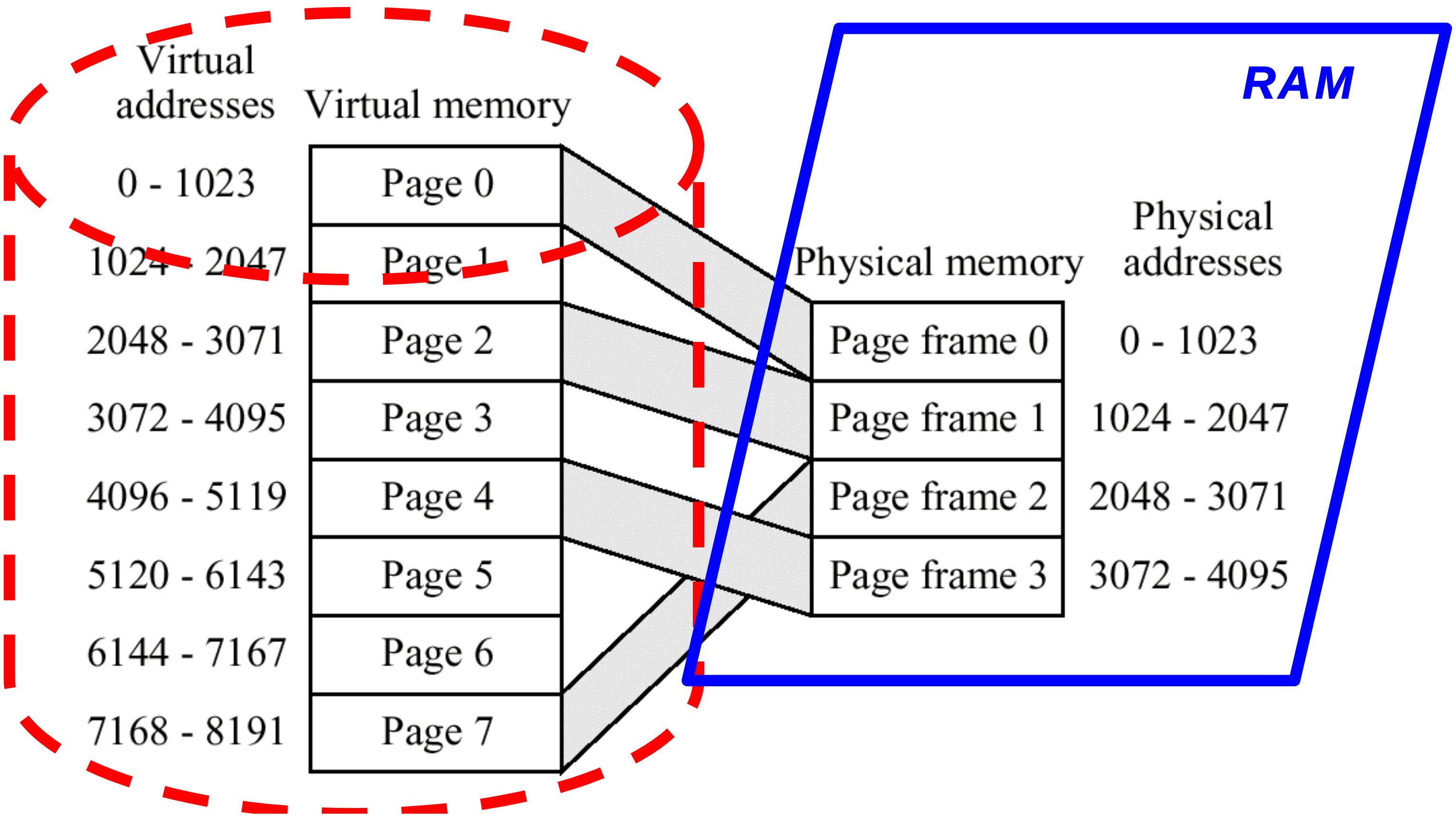
Virtual Memory: swap space

- Virtual memory is stored in a specialized hard disk image (often called a “swap” partition, or just “swap” space).
- The physical memory holds a small number of virtual *pages* in physical page *frames*.
- A mapping between a virtual and a physical memory:



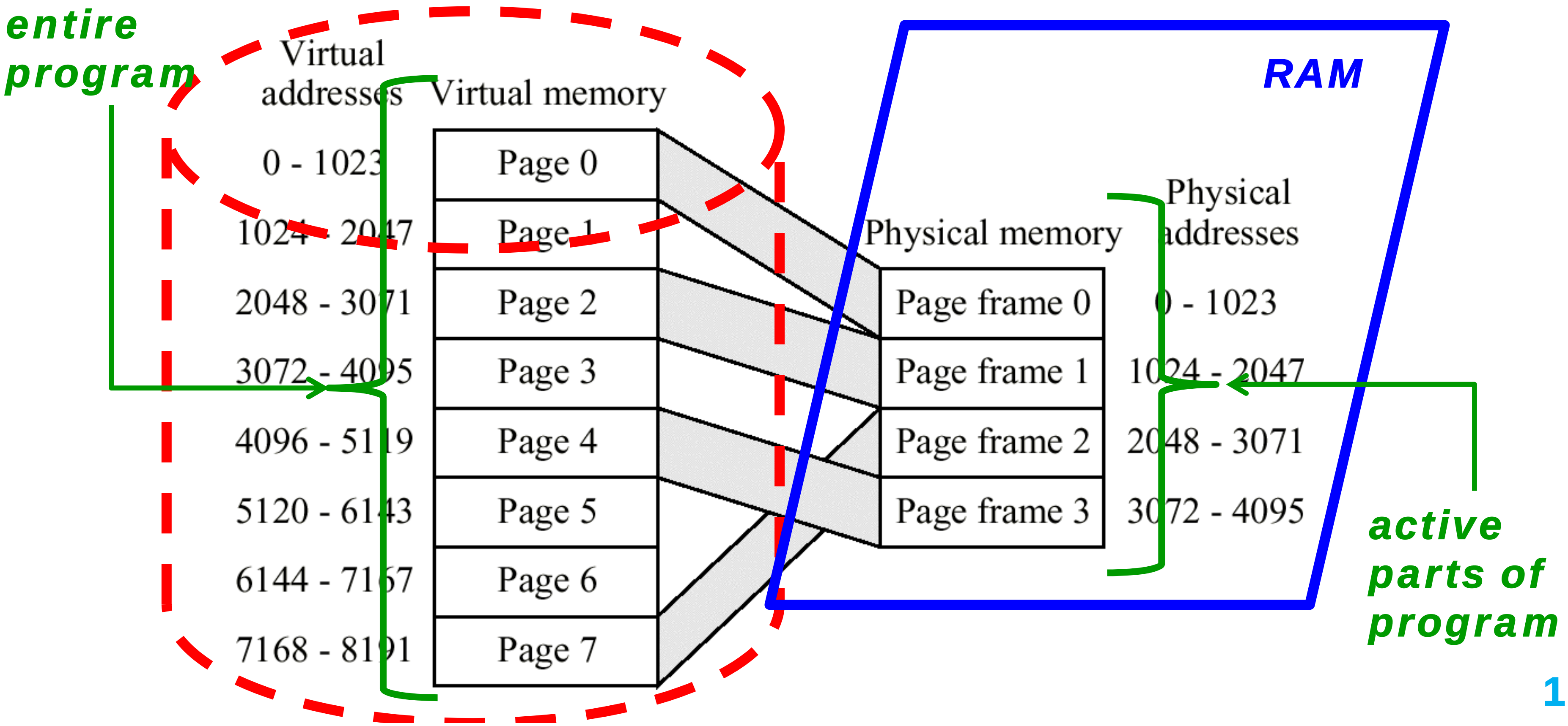
Virtual Memory: swap space

- Virtual memory is stored in a specialized hard disk image (often called a “swap” partition, or just “swap” space).
- The physical memory holds a small number of virtual *pages* in physical page *frames*.
- A mapping between a virtual and a physical memory:



Virtual Memory: swap space

- Virtual memory is stored in a specialized hard disk image (often called a "swap" partition, or just "swap" space).
- The physical memory holds a small number of virtual *pages* in physical page *frames*.
- A mapping between a virtual and a physical memory:



Questions on Virtual Memory

- **How much total virtual memory should be represented?**
  - How much disk space is available?
  - What is the ratio of allocated vs. active memory used by typical programs?
- **What granularity of pages to use?**
  - Smaller pages = more precision, less waste (later)
  - Smaller pages = more bookkepping and swapping
- **How to replace (“swap”) pages?**
  - Similar to caching, need a replacement policy: FIFO, LIFO, LRU, …?

What is the Page Table?

The page table (kept by the OS) records the mapping of virtual pages to physical frames

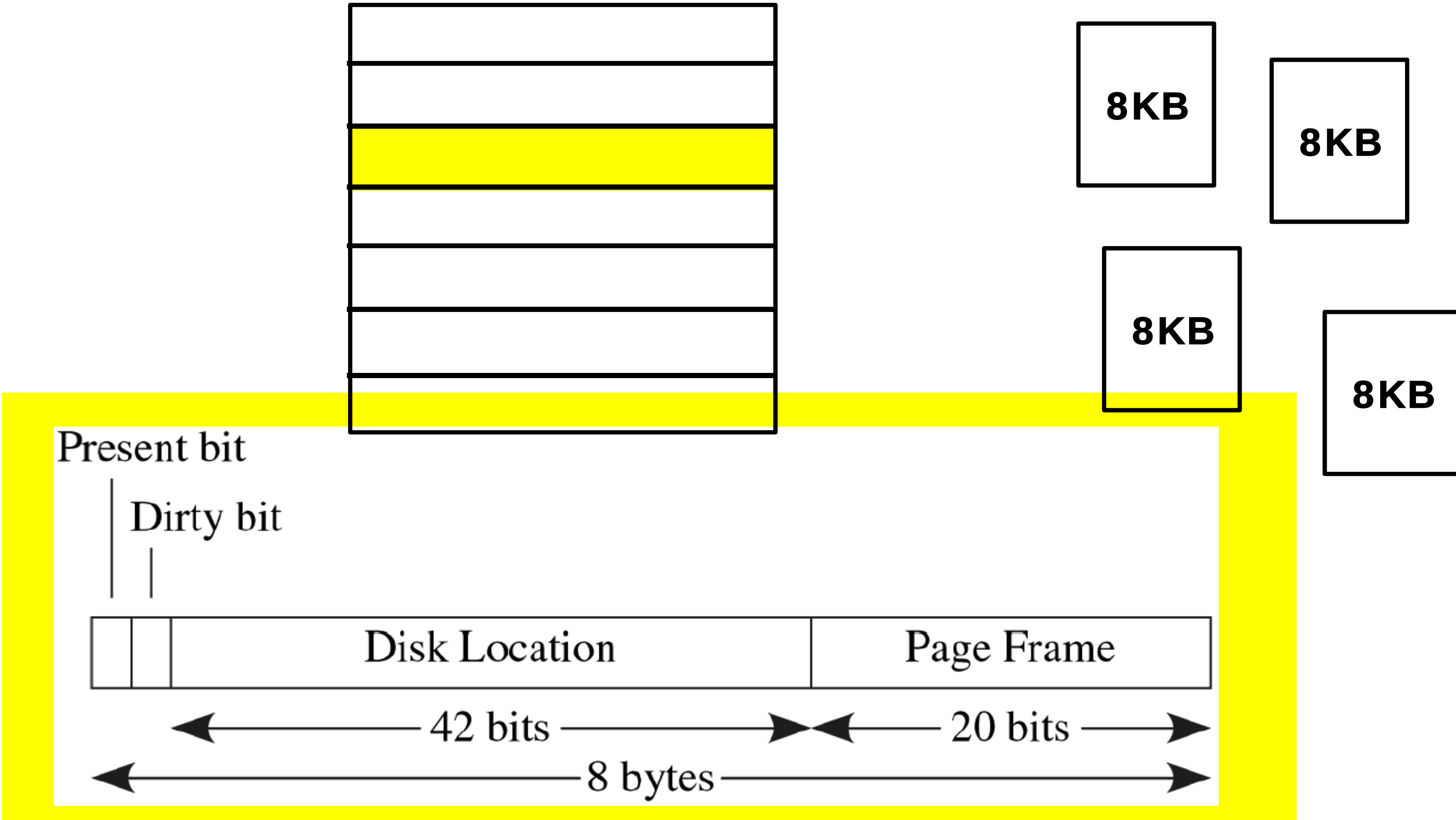
***Where* is the Page Table?**
➔ in RAM main memory

Present bit:
0: Page is not in physical memory
1: Page is in physical memory

Pag	Present bit	Disk address	Page frame
0	1	01001011100	00
1	0	11101110010	xx
2	1	10110010111	01
3	0	00001001111	xx
4	1	01011100101	11
5	0	10100111001	xx
6	0	00110101100	xx
7	1	01010001011	10

Page Table – Example.1

Consider a paging scheme that uses pages of size 8KB and a table with 64 bit entries



Page Table – Example.2

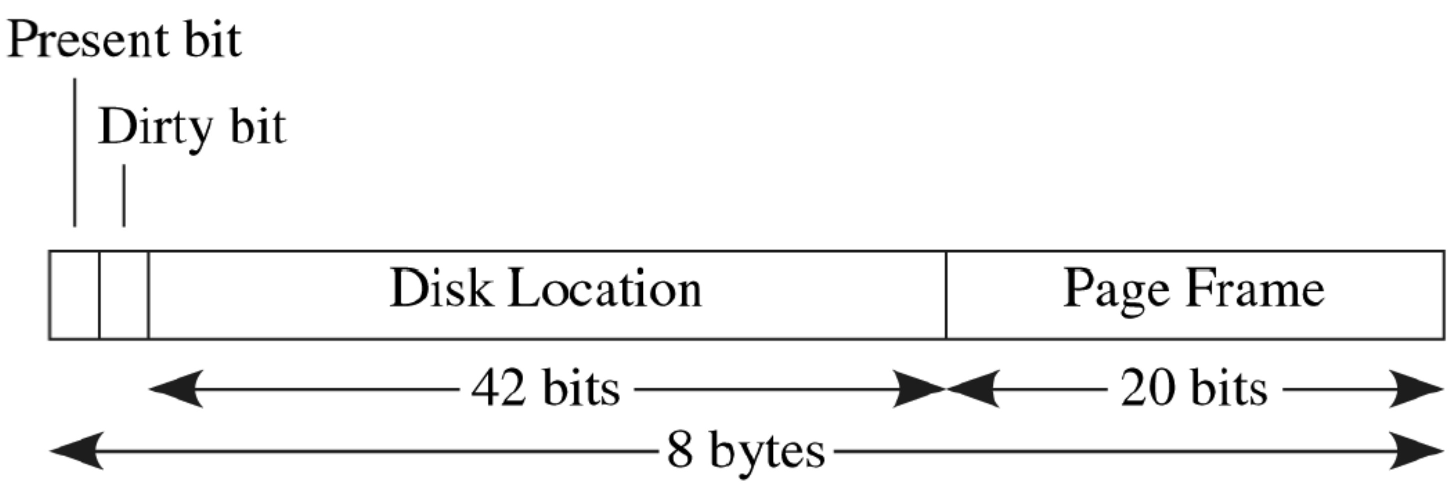
Consider a paging scheme that uses pages of size 8KB and a table with 64 bit entries

1. How much RAM can we have?

- **20 bits for the frame number (see below) imply**

  ➔the maximum number of entries (pages) = 2^{20}

  ➔ then 2^{20} x 8KB (size of each page) = 2^{33}B = 8GB of RAM



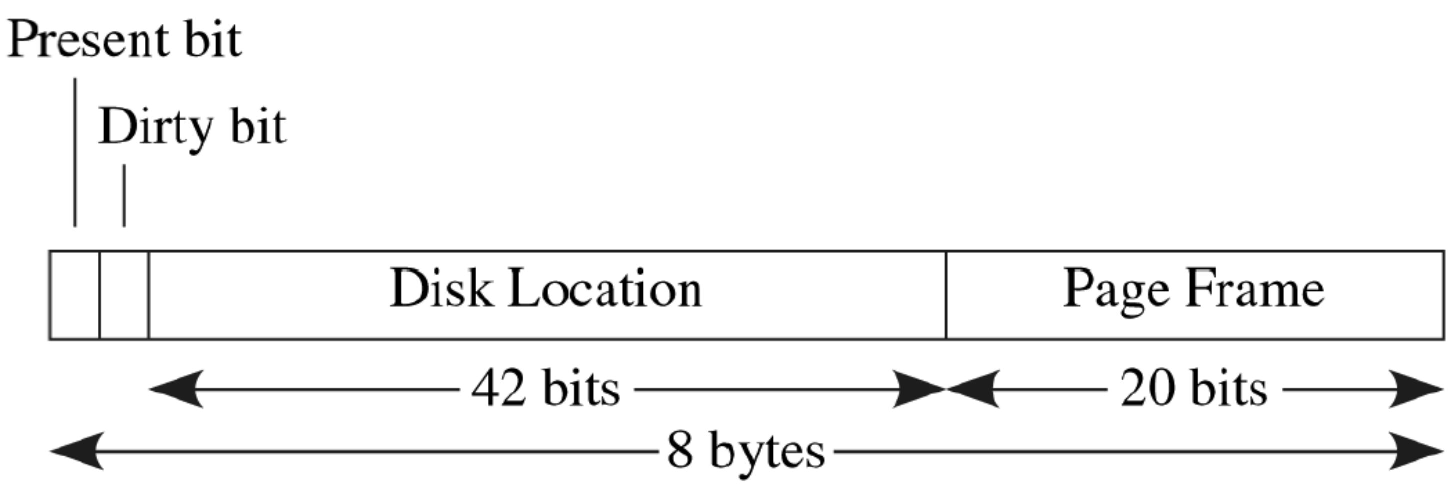
Page Table – Example.3

Consider a paging scheme that uses pages of size 8KB and a table with 64 bit entries

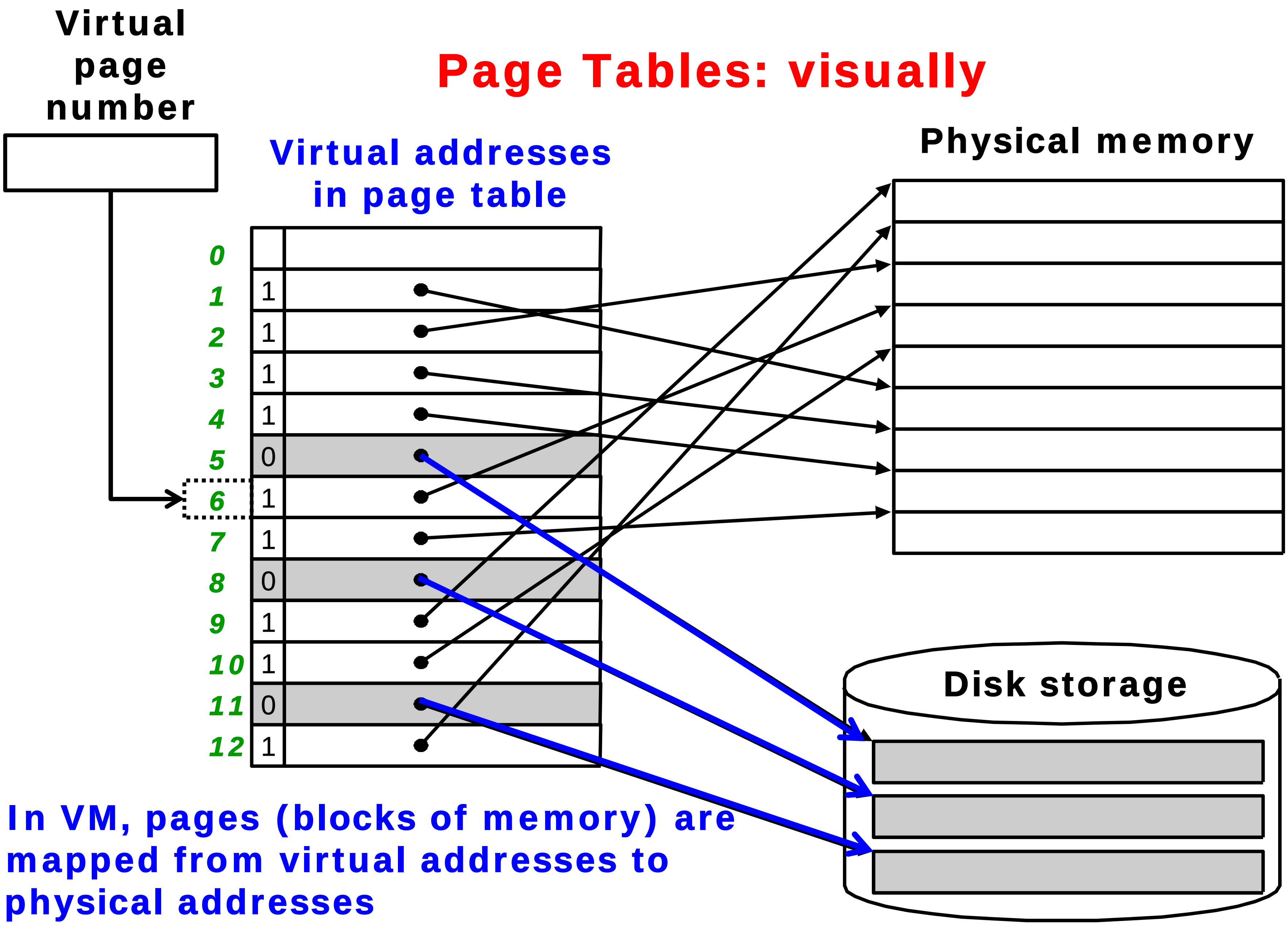
2. How much (maximum) virtual memory?

- **42 bits for page number on disk imply**

  ➔ 2^{42} x 8KB = 2^{55}B = 2^{15} TB! (maximum)



Page Tables: visually



In VM, pages (blocks of memory) are mapped from virtual addresses to physical addresses

NOTE: pages could map to disk

Page Tables – How they work

Page Table **is a table to index memory to locate pages**

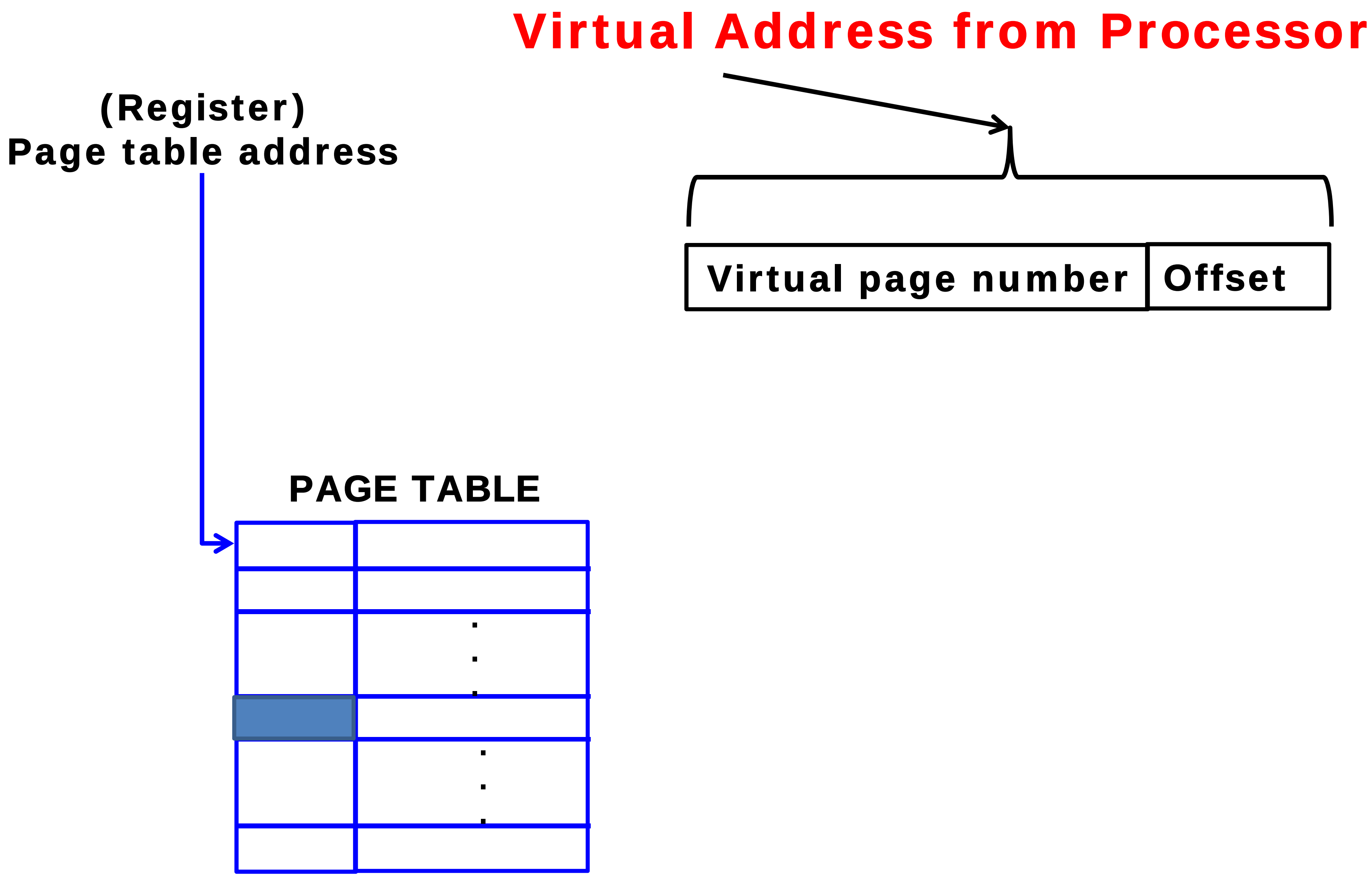
- **It may itself be in memory or in MMU**
- **Contains the corresponding physical page number**
- **May contain entries for pages not present in memory**
- **Indexed with the page number from the virtual address**
- **Each program has its own page table**

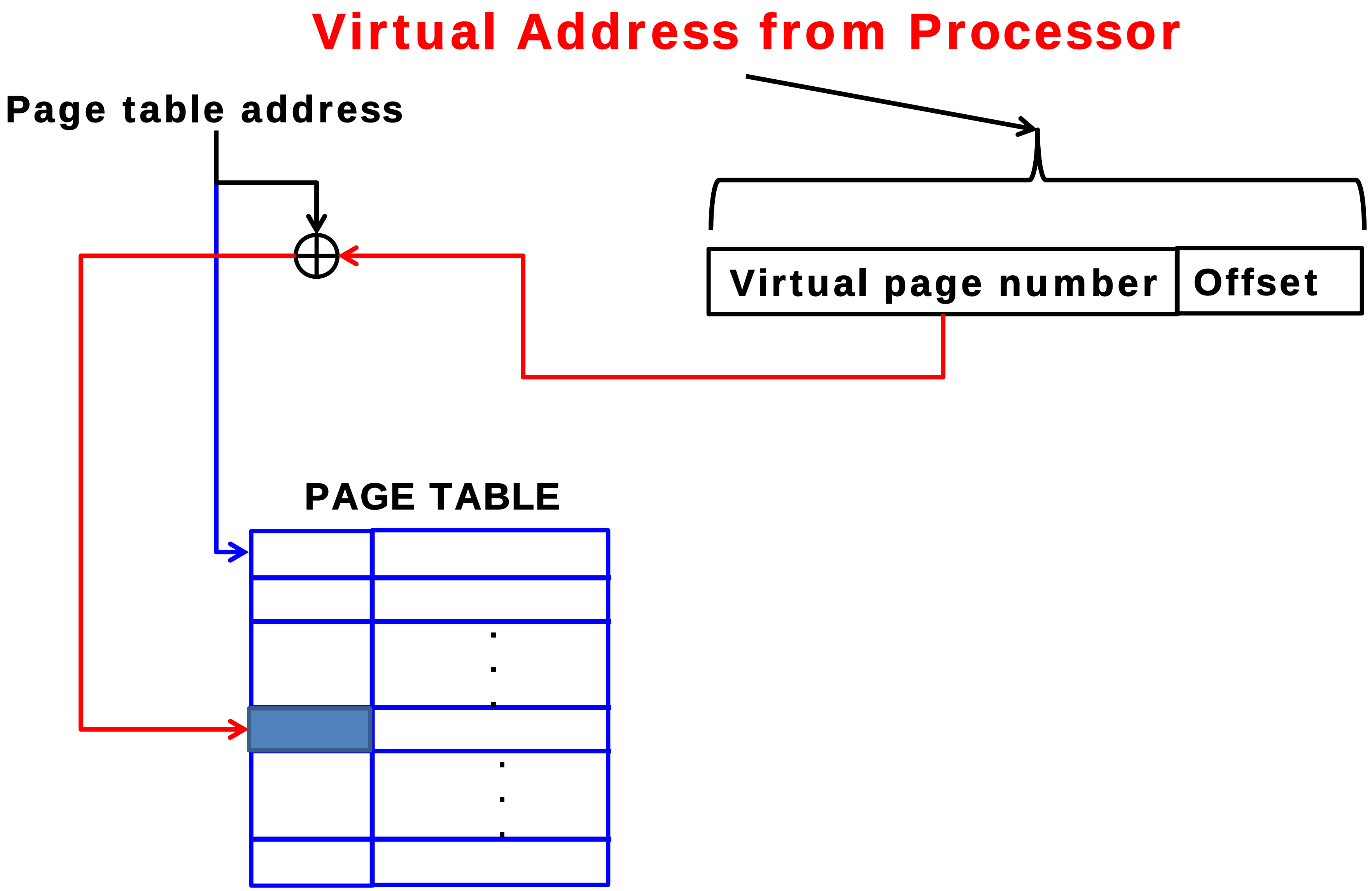
Page Tables – How they work

Hardware in the system includes a register that points to the start of the page table

- **A valid bit is used in each page table entry**
  - **a) if bit is *off*, the page is not in main memory**
    ➔page fault occurs ➔ must go to disk
  - **b) if bit is *on*, the page is valid**
    ➔ entry + offset = physical page number in memory

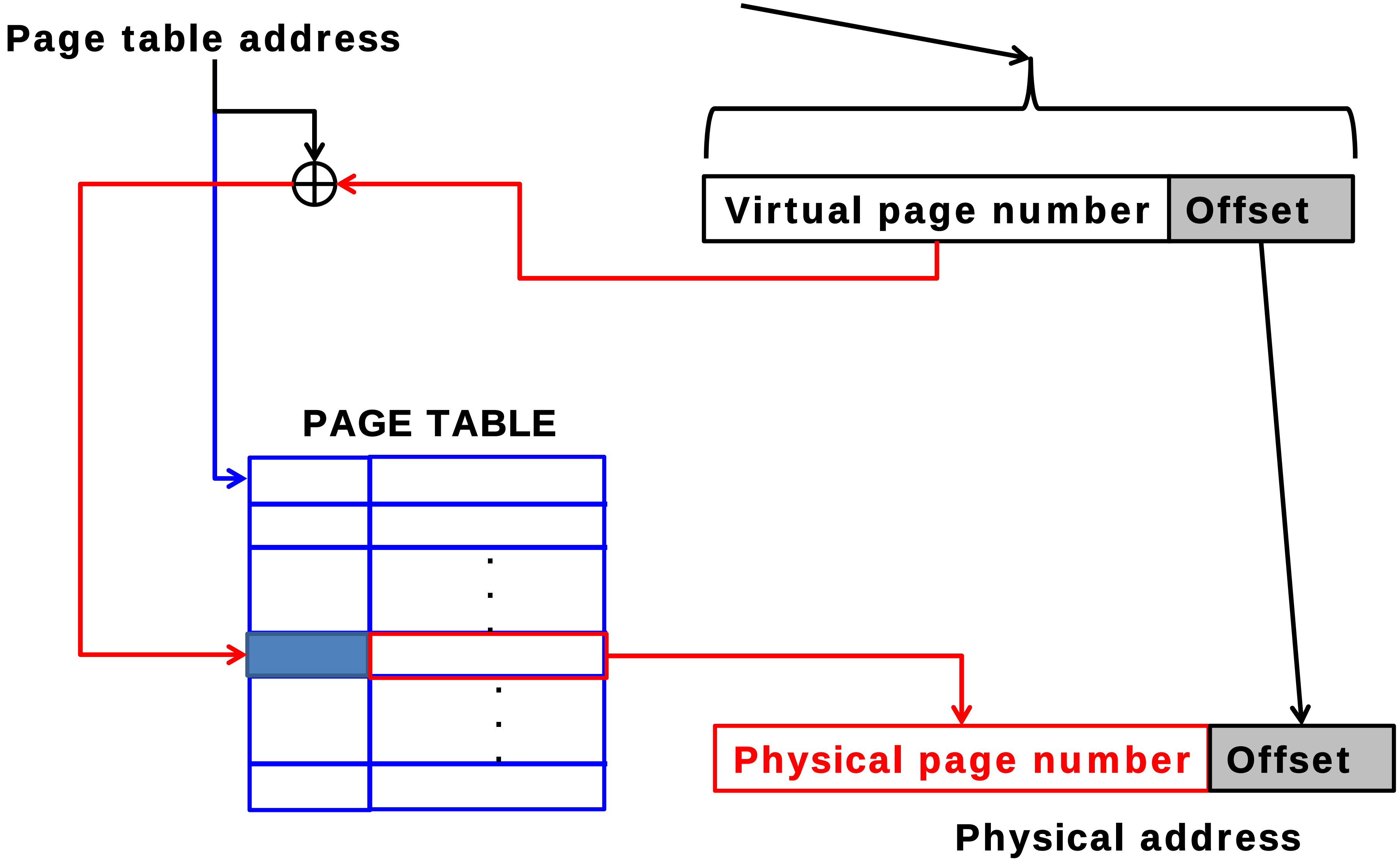
- **No tags are required:**
  page table contains a mapping for every possible page


Virtual Address from Processor
(Register)
Page table address
Virtual page number
Offset
PAGE TABLE



Virtual Address from Processor
Page table address
Virtual page number
Offset
PAGE TABLE

Virtual Address from Processor



Example from Spring 2009 final exam (1)

Consider a computer which has a virtual memory system with the following characteristics. The computer has a 32-bit address space with byte addressable memory; this implies a virtual address space which is 2^{32} bytes or 4 GB in size. The main memory (RAM memory) is 1 GB in size. Pages are 4 KB in size.

How many entries can the page table have?

Each page is 4 KB = $2^2 \times 2^{10} = 2^{12}$

Total address space = 2^{32} bytes

➔ $2^{32} / 2^{12} = 2^{20}$ pages

➔ it will need 20 bits to give a number entry to each page (5 hex digits)

Example from Spring 2009 final exam (2)

Suppose that the CPU generates the sequence of memory addresses shown below (written in hexadecimal).

time

```
0x00001020
0x00006FF8
0x00006004
0x00001024
0x00005FF8
0x00001028
```

Example from Spring 2009 final exam (3)

Suppose that the page table contains the entries shown on the left. Invalid (unmapped) entries in the page table are shown as empty.

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04

.....

Example from Spring 2009 final exam (4)

Show the corresponding sequence of addresses in main memory which the virtual addresses are mapped to.

If any address cannot be mapped, show your answer as the words ‘page fault’.

Example from Spring 2009 final exam (5)

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04

0x00001020

0x00006FF8

0x00006004

0x00001024

0x00005FF8

0x00001028

Virtual Addresses from processor

Corresponding Real Addresses

Example from Spring 2009 final exam (6)

Consider the first address = 0x00001020

It was previously decided that there are 2^{20} pages and we will need 20 bits to give a number entry to each page

$$\texttt{0x}\underbrace{\texttt{00001}}\texttt{020}$$

5 Hex digits = 20 bits = 0001 → page entry number

$$\texttt{0x00001}\underbrace{\texttt{020}}$$

Remaining 3 Hex digits = 12 bits = 020 → offset

Example from Spring 2009 final exam (7)

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04

0x00001020

0x00006FF8

0x00006004

0x00001024

0x00005FF8

0x00001028

Virtual Addresses from processor

015C2020

Corresponding Real Addresses

Example from Spring 2009 final exam (8)

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04

Virtual Addresses from processor	Corresponding Real Addresses
0x00001020	015C2020
0x00006FF8	3EF04FF8
0x00006004	
0x00001024	
0x00005FF8	
0x00001028	

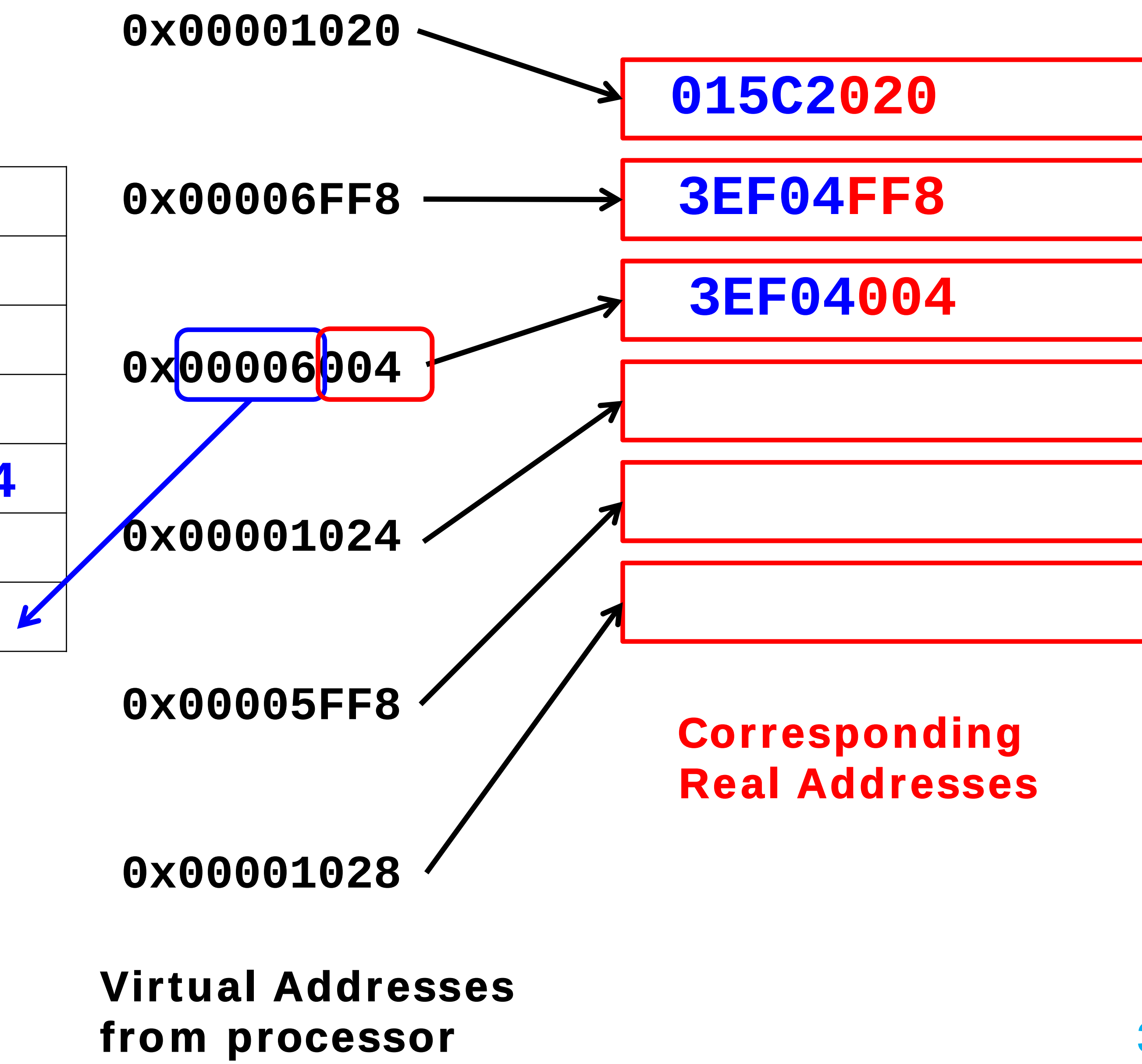
Corresponding Real Addresses

Virtual Addresses from processor

Example from Spring 2009 final exam (9)

Page Table

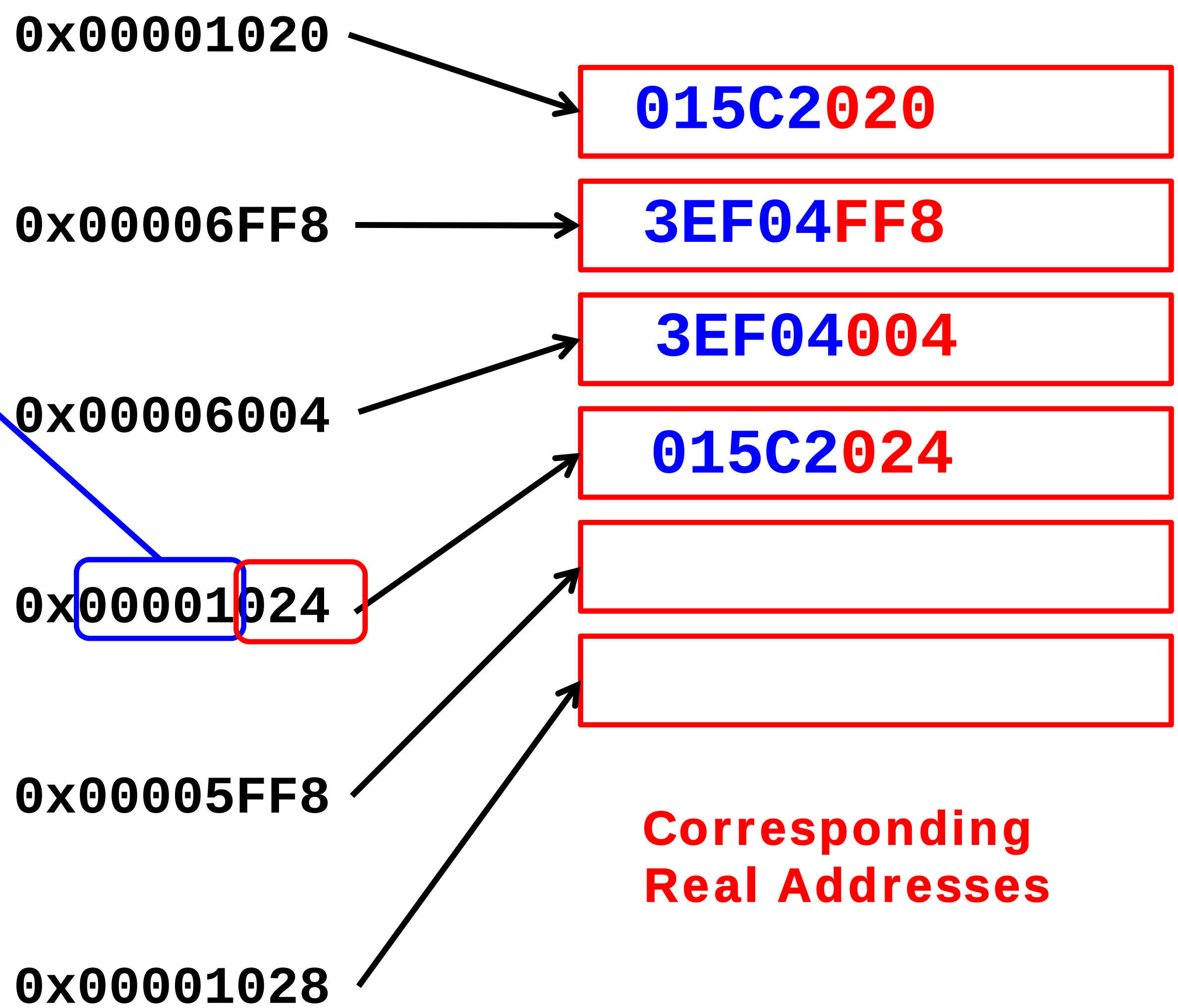
0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04



Example from Spring 2009 final exam (10)

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04



Corresponding Real Addresses

Virtual Addresses from processor

Example from Spring 2009 final exam (11)

Page Table

0:	0x00443
1:	0x015C2
2:	-
3:	0x28AE1
4:	0x016DE4
5:	-
6:	0x3EF04

0x00001020

0x00006FF8

0x00006004

0x00001024

0x00005FF8

0x00001028

Virtual Addresses from processor

015C2020

3EF04FF8

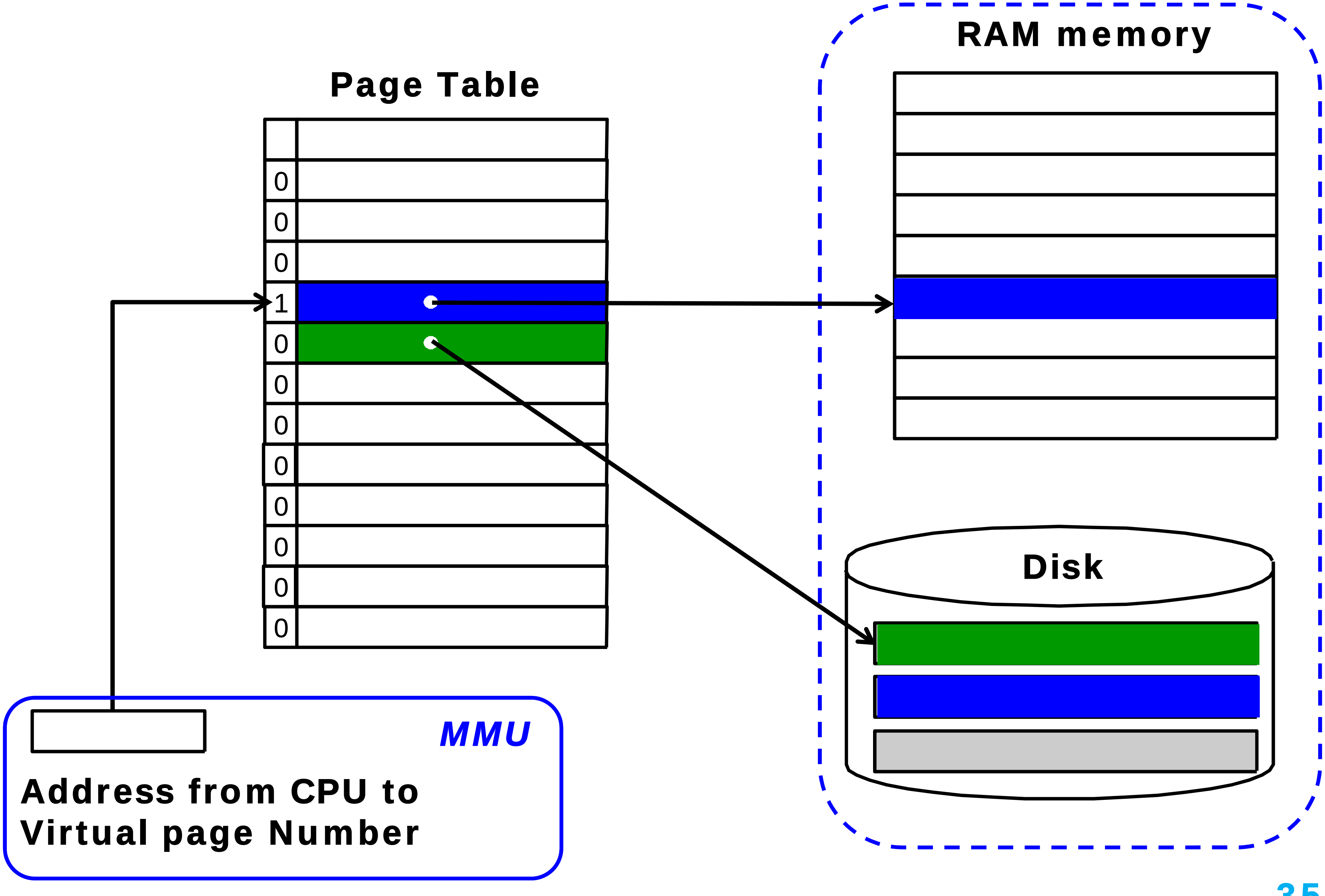
3EF04004

015C2024

Page fault

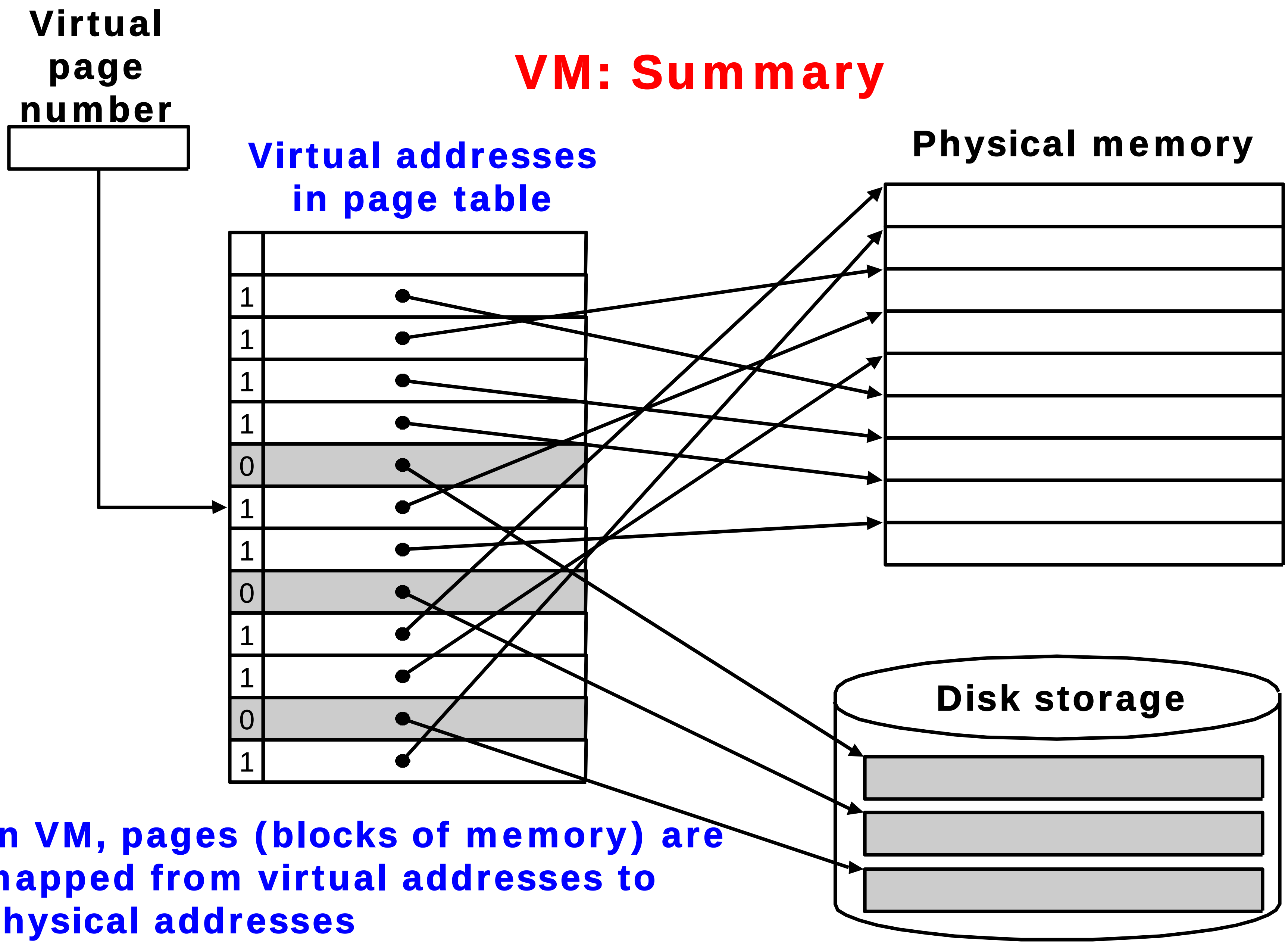
Corresponding Real Addresses

VM: Summary



RAM memory
Page Table
0
0
0
1
0
0
0
0
0
0
0
0
Disk
MMU
Address from CPU to
Virtual page Number

VM: Summary



In VM, pages (blocks of memory) are mapped from virtual addresses to physical addresses

NOTE: pages could map to disk

Page Faults: : the data is not in memory, retrieve it from disk

The OS finds the page in the next level of the hierarchy and decides where to place it in the main memory

Design challenges:

- ***Huge miss penalty***
  - **pages should be fairly large (e.g. 32 KB)**
  - **optimize the page placement**
- ***Reducing page fault rate is important***
  - **allow fully associative placement of pages**
- ***Reduce the overhead***
  - **handle the faults in software or hardware?**
  - **allow using clever algorithms?**
- ***using write-through is too expensive since it takes too long***
  - **use write back**
  - **a dirty bit is used to decide whether to write back or not**

Making Address Translation Faster: The TLB

Every memory access by a program can have two parts:

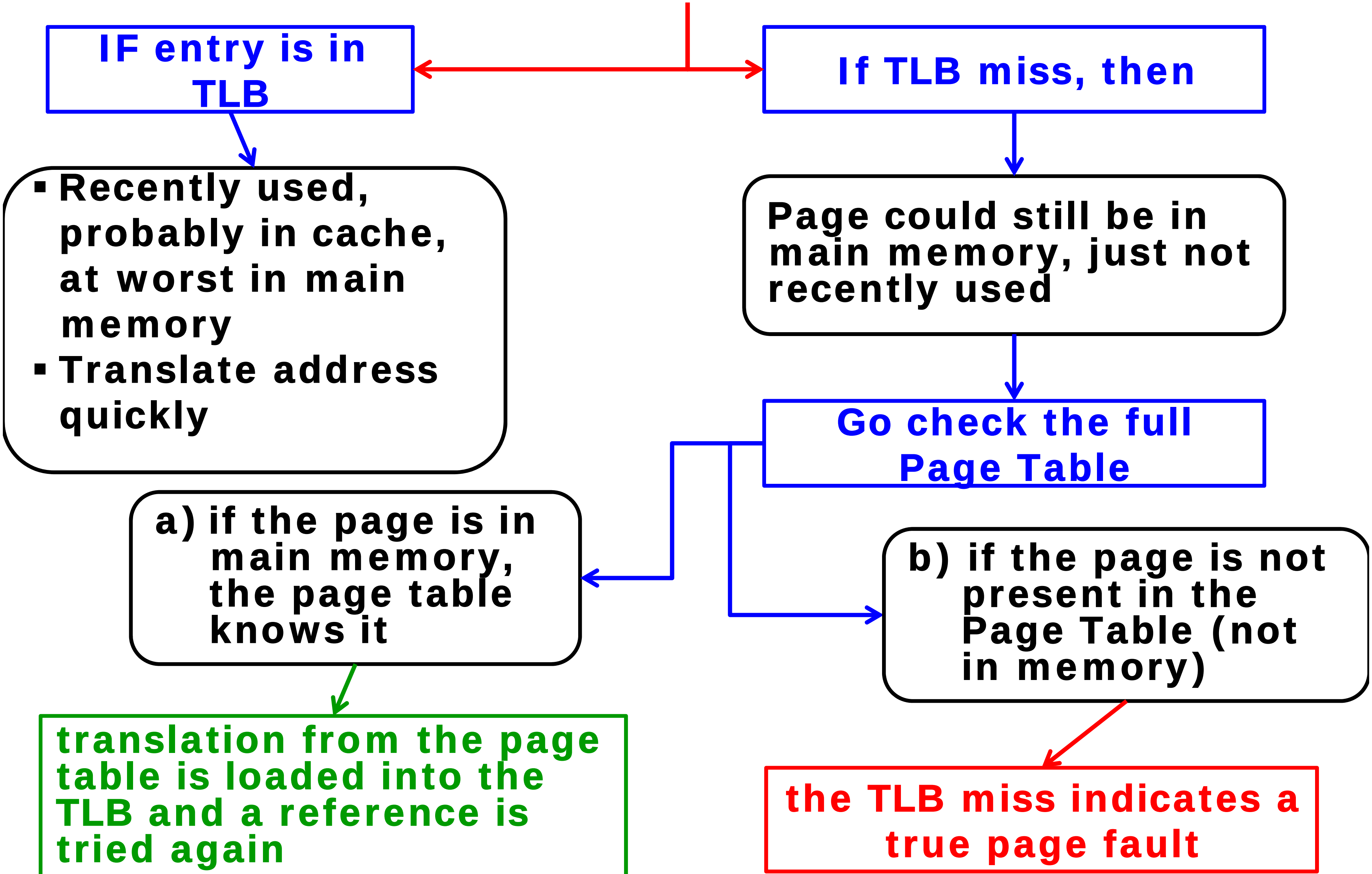
1. **one memory access to obtain the physical address**
2. **a second access to get the data**

To speed up step 1 (translation) use a TLB = Translation-lookaside buffer (TLB):

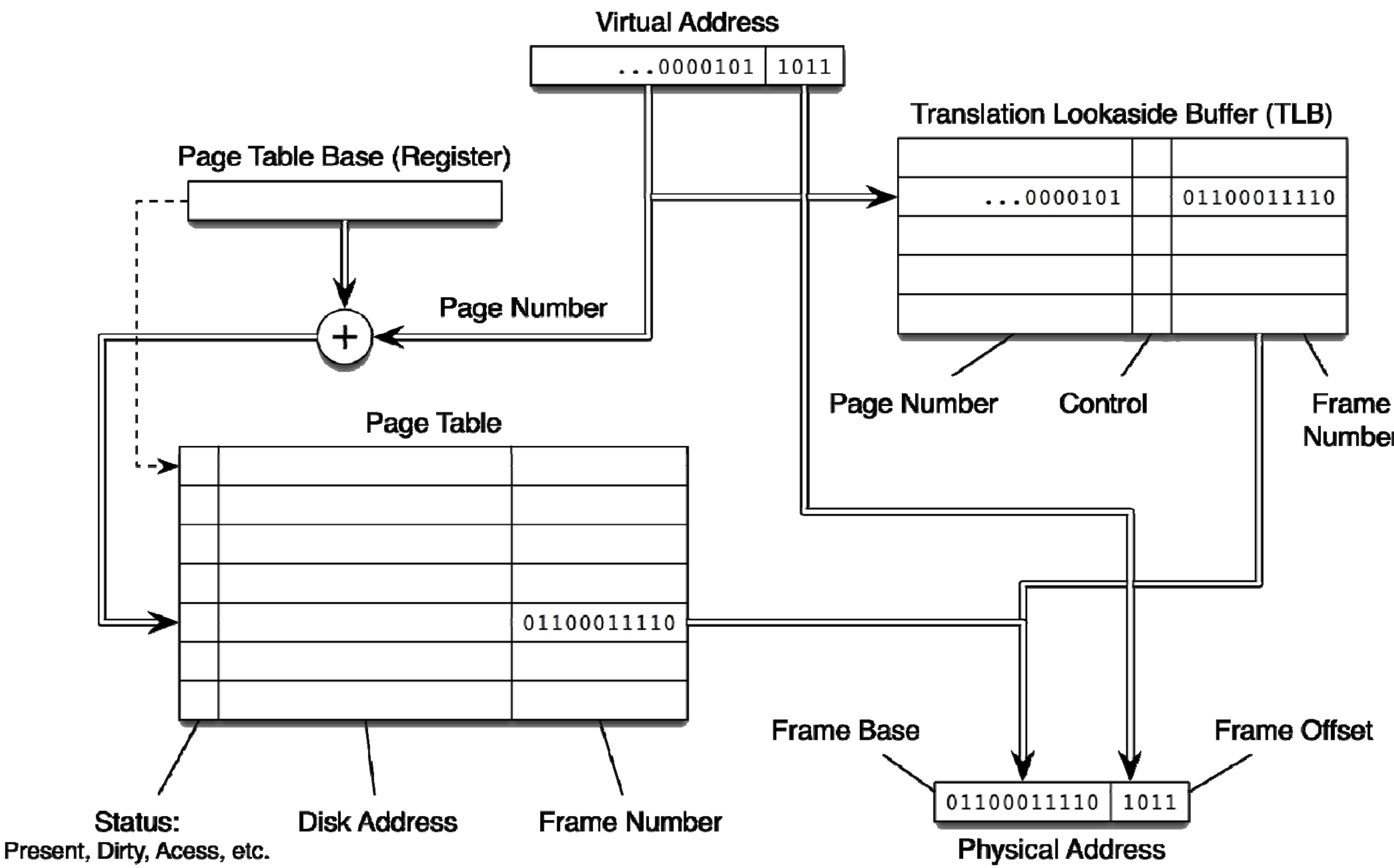
a cache for recent address translations (recent page entries)

- **TLB can be: fully associative, set associative, or direct mapped**
- **TLBs are usually small, typically not more than 128 - 256 entries**
- **TLB access time should be comparable to cache access time**

Making Address Translation Faster: The TLB



Address Translation



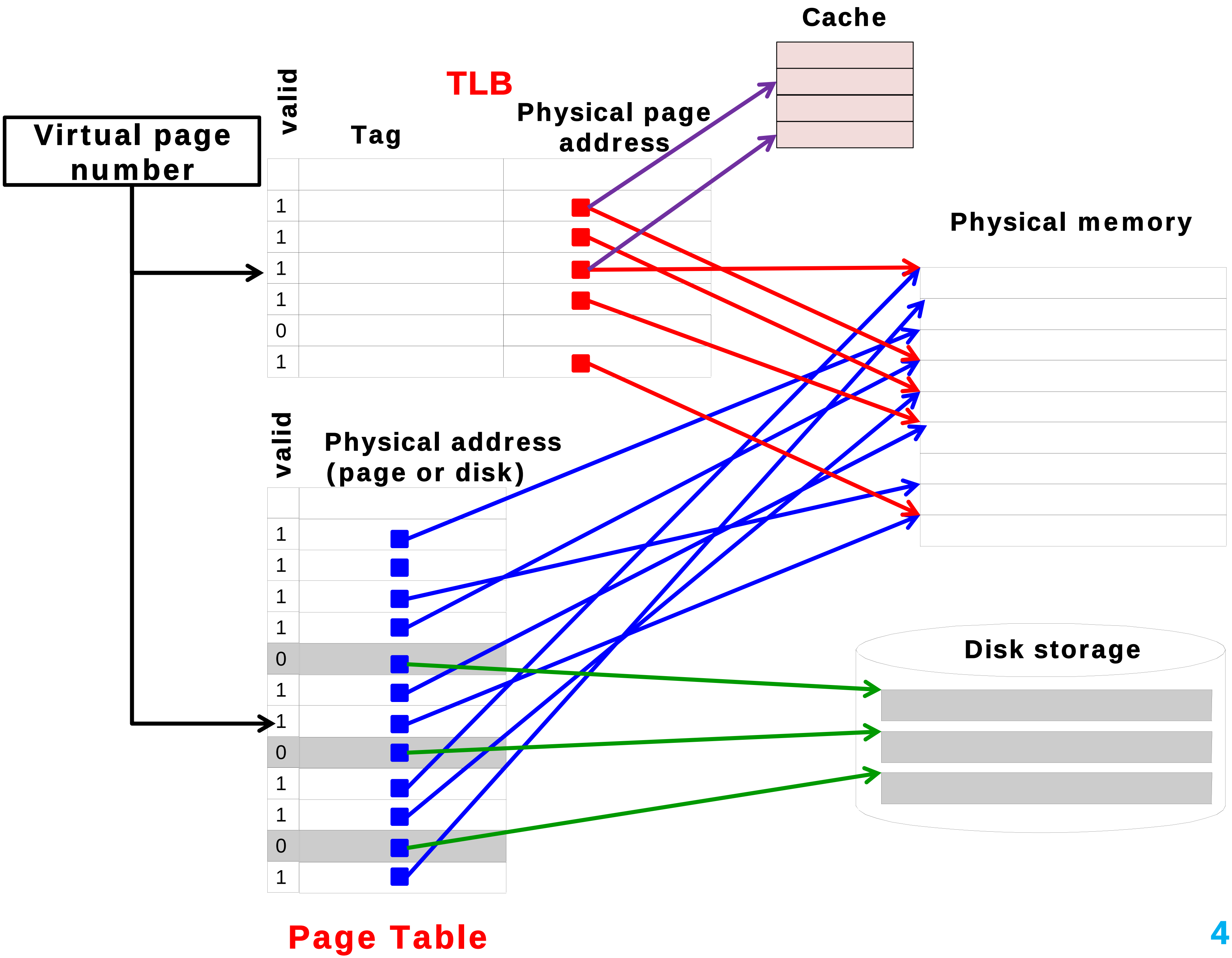
Translation Lookaside Buffer (TLB)

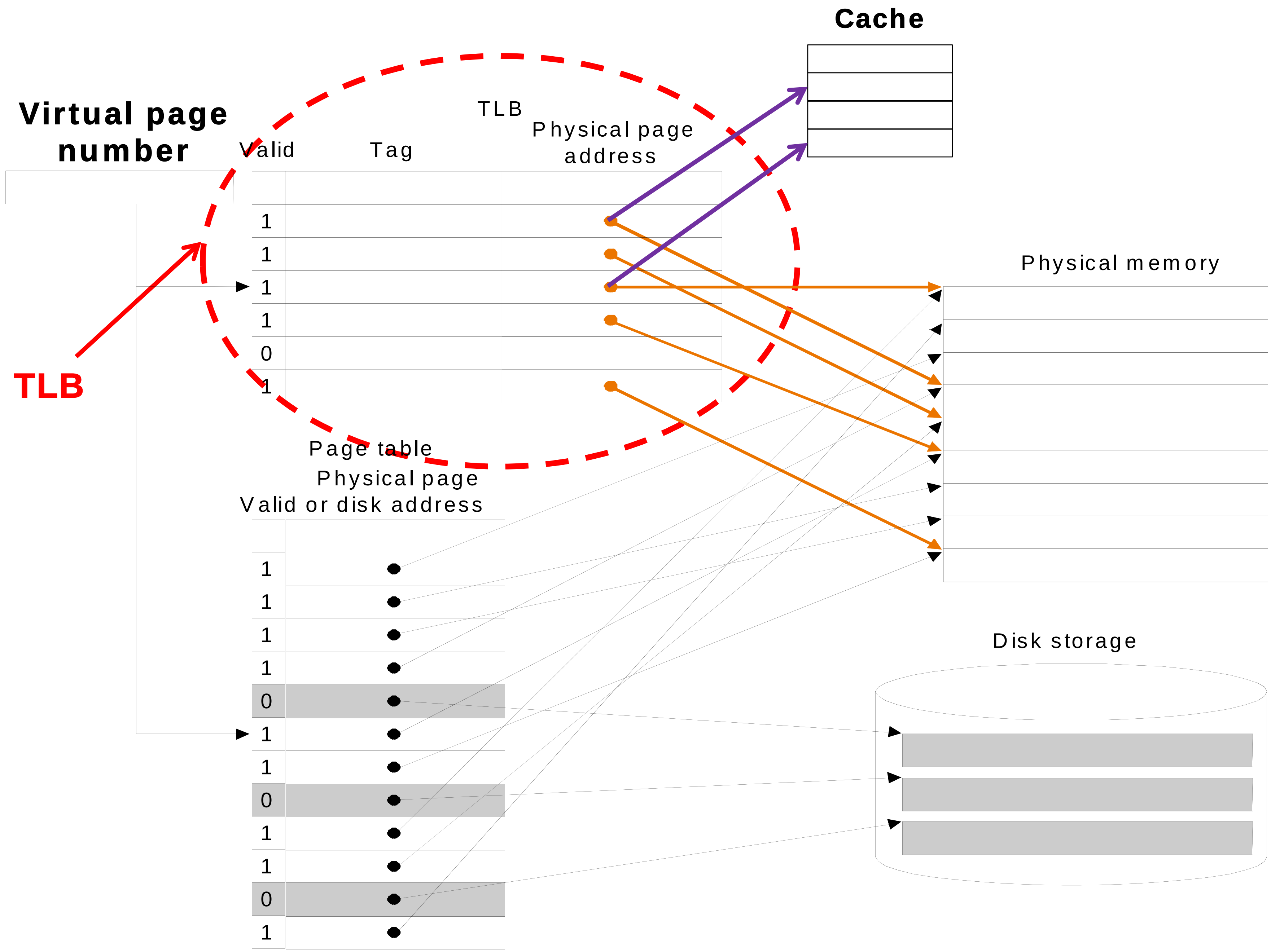
Used to speed virtual address translation, a "cache" of page table entries: the TLB (implemented in hardware)

- **TLB holds recent page to frame mappings**
- **Normally resides in MMU**

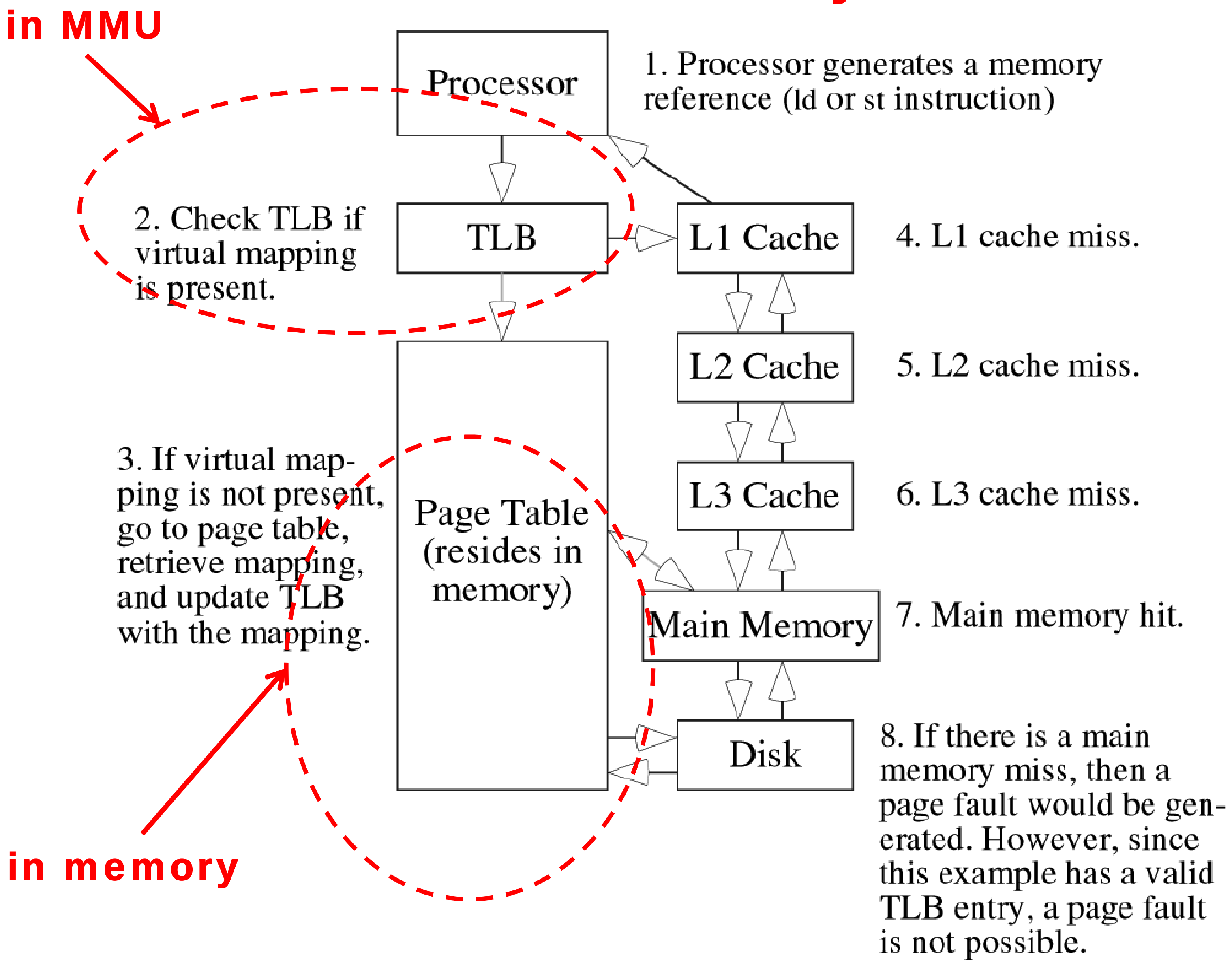
Entries may be invalid when pages get ejected (swapped) out

Valid	Virtual page number	Physical page number
1	0 1 0 0 1	1 1 0 0
1	1 0 1 1 1	1 0 0 1
0	- - - - -	- - - -
0	- - - - -	- - - -
1	0 1 1 1 0	0 0 0 0
0	- - - - -	- - - -
1	0 0 1 1 0	0 1 1 1
0	- - - - -	- - - -

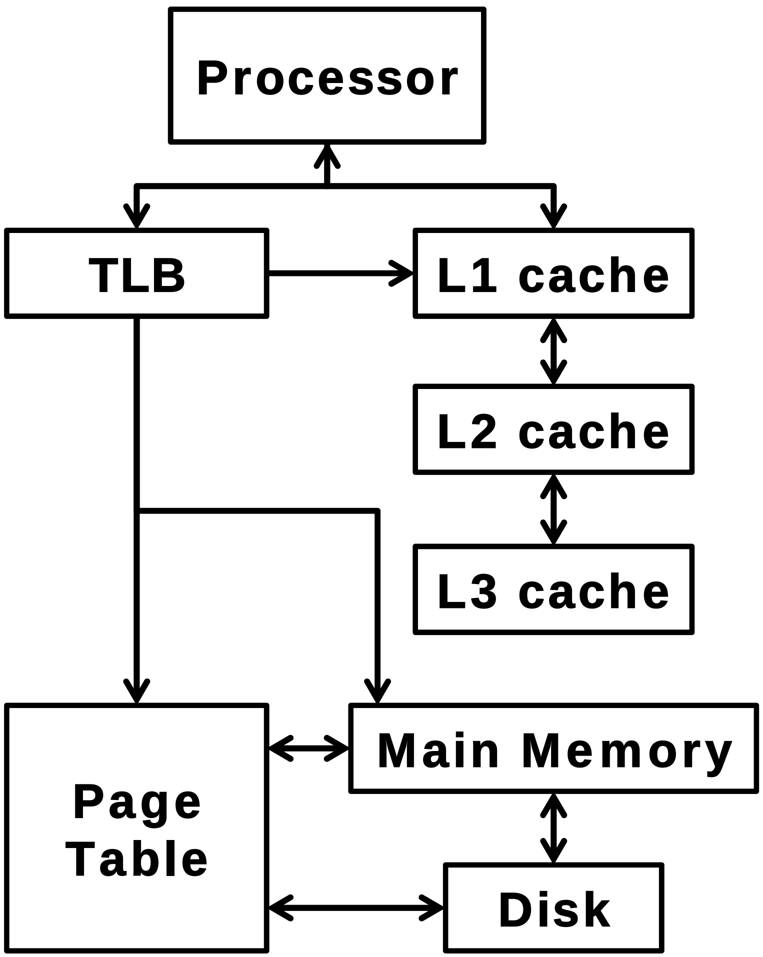

Virtual page number
valid
Tag
TLB
Physical page address
Cache
Physical memory
valid
Physical address (page or disk)
Disk storage
Page Table


Cache
Virtual page number
TLB
Valid
Tag
Physical page address
1
1
1
1
0
1
TLB
Physical memory
Page table
Physical page
Valid or disk address
1
1
1
1
0
1
1
0
1
1
0
1
Disk storage

Putting it All Together: example when data is in memory



Putting it All Together: more details about TLB and cache



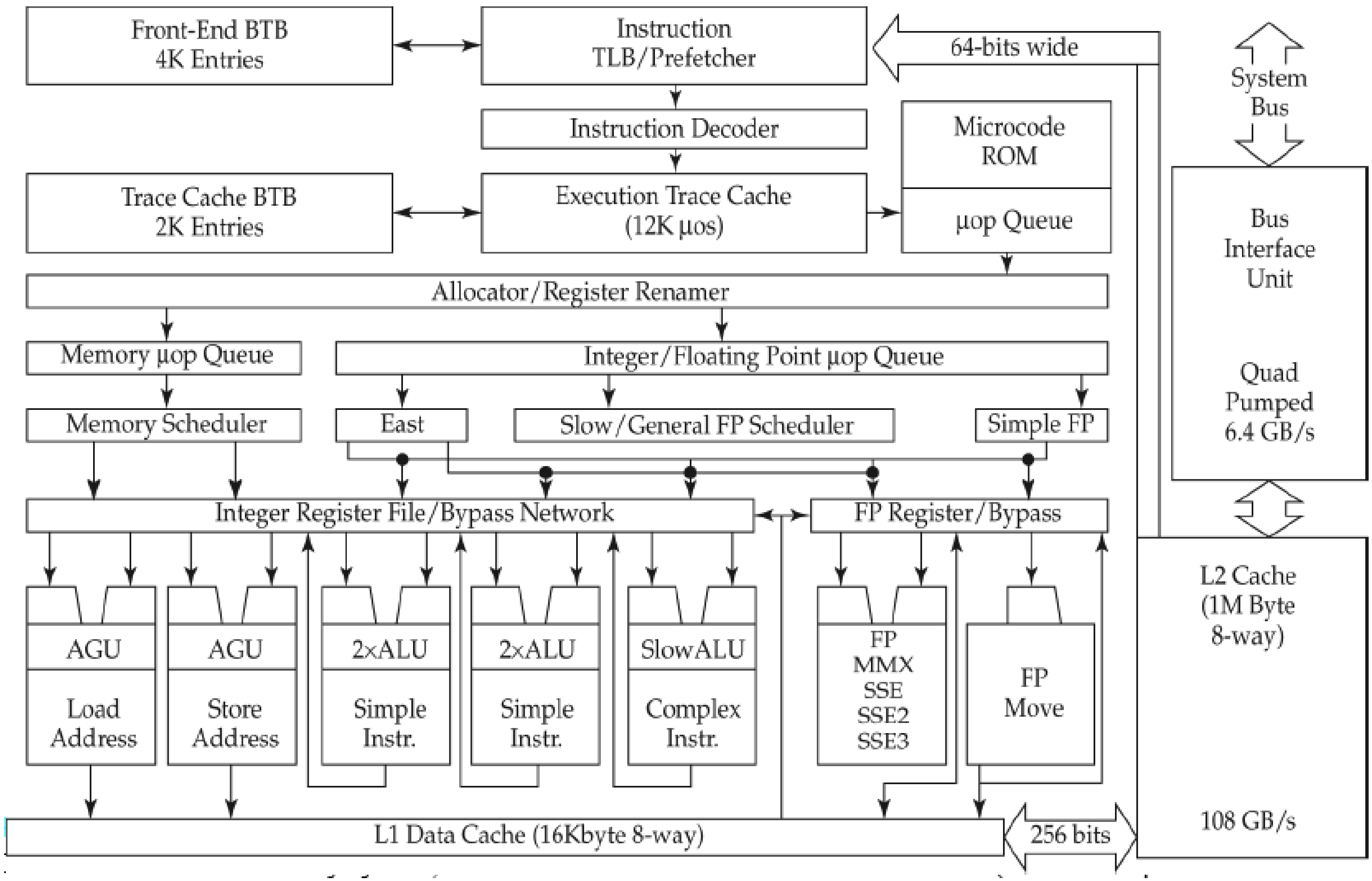
1. Initial address sent in parallel to TLB and L1 cache
2. If in L1
   ➔ cache hit, done
3. Else parallel search continues in TLB and L2 and L3
   ➔if quick answer with hit comes from TLB or L2 or L3
   ➔ done
4. Else possibly
   ➔TLB hit to memory
5. Etc. etc.

- All supported by hardware until Main memory hit
- OS software takes over for page Fault and access to disk

Typical values for a TLB

Size:	16-512 entries
Block size:	1-2 page table entries of typically 4-8 bytes each
Hit time:	0.5 – 1 clock cycles
Miss penalty:	10-100 clock cycles
Miss rate:	0.01% - 1%

Putting it All Together



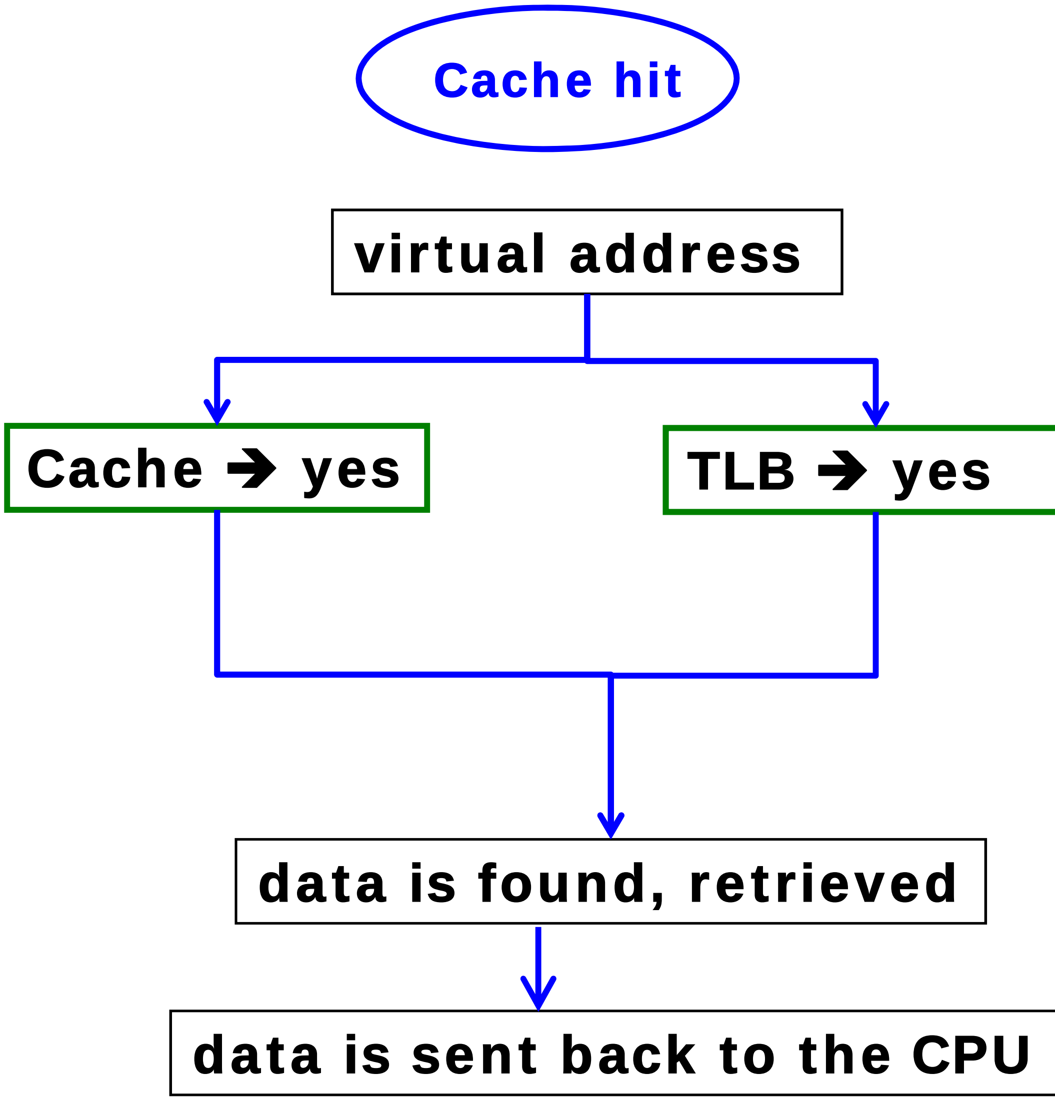
ARM's Virtual Memory System Architecture

MMU within processor

Defines 1Kb, 4KB and 64Kb pages

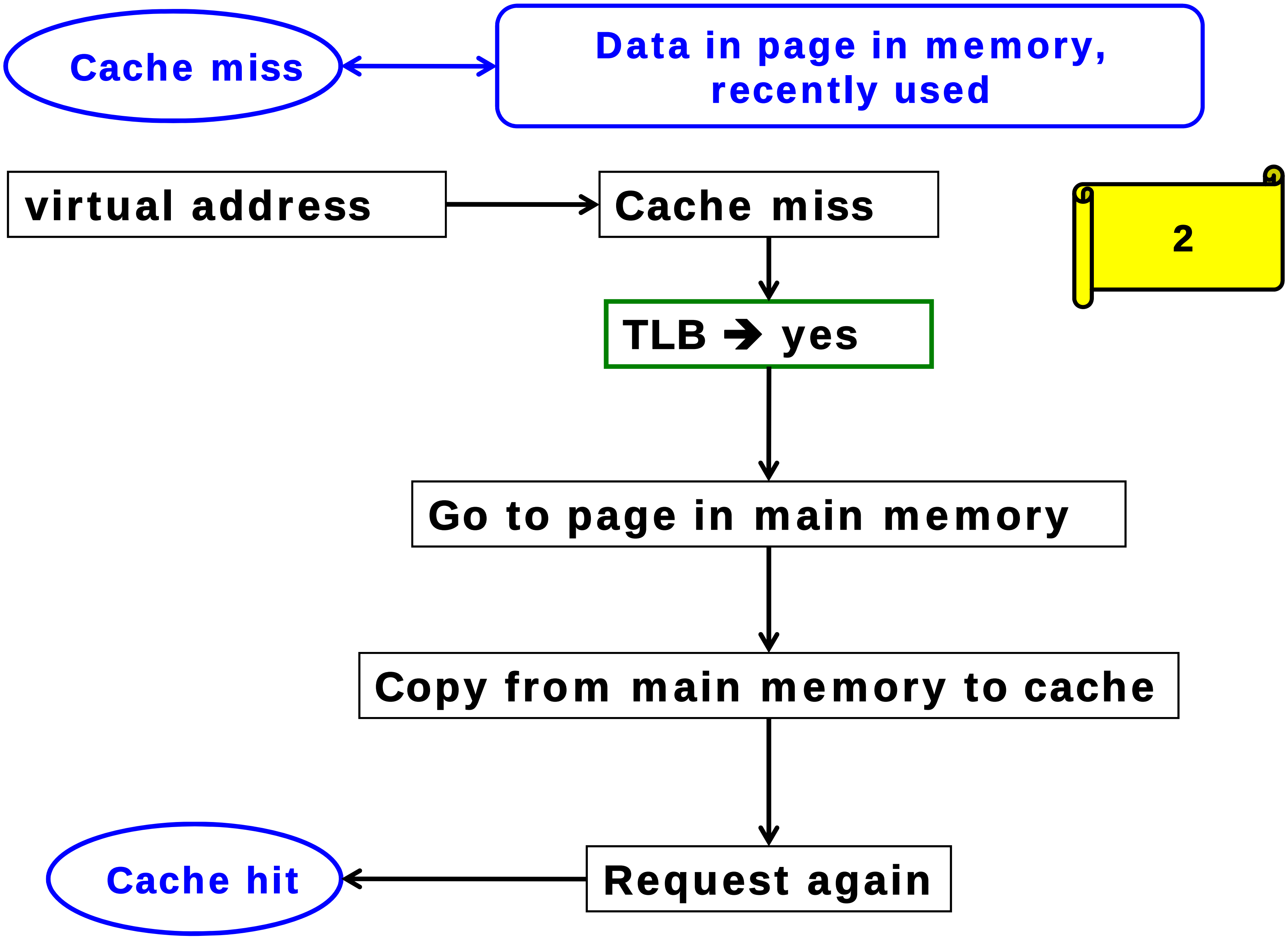
TLB all hardware

Integrating Virtual Memory, TLBs, and Caches

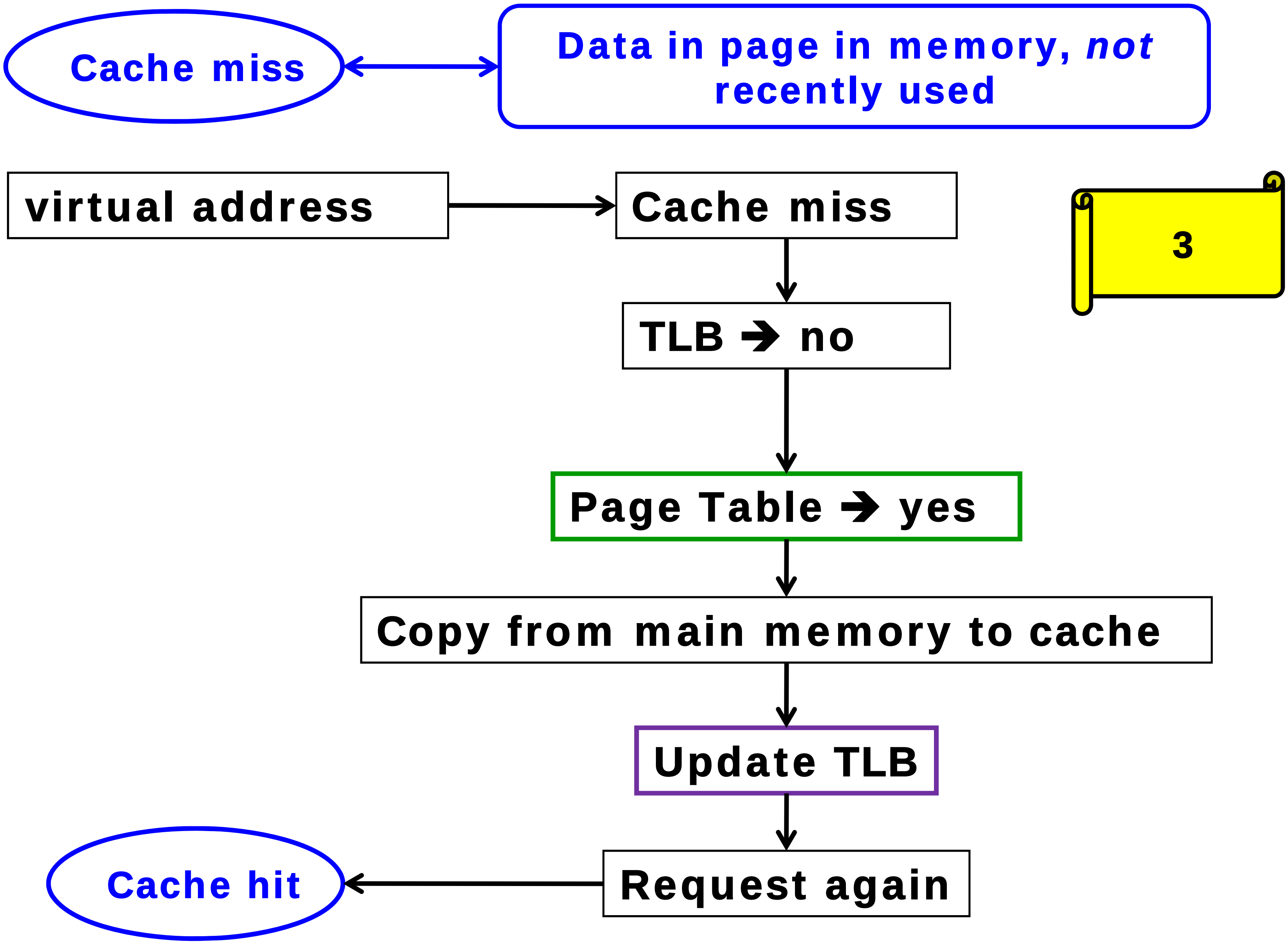


1

Integrating Virtual Memory, TLBs, and Caches



Integrating Virtual Memory, TLBs, and Caches



Integrating Virtual Memory, TLBs, and Caches

Cache miss ↔ Data on disk

4

virtual address → Cache miss

↓

TLB and Page Table ➔ all miss

↓

Page Fault

↓

Data cannot be in the cache unless it is present in main memory

↓

The OS flushes the contents of any page from the cache

↓

The OS migrates that page from disk

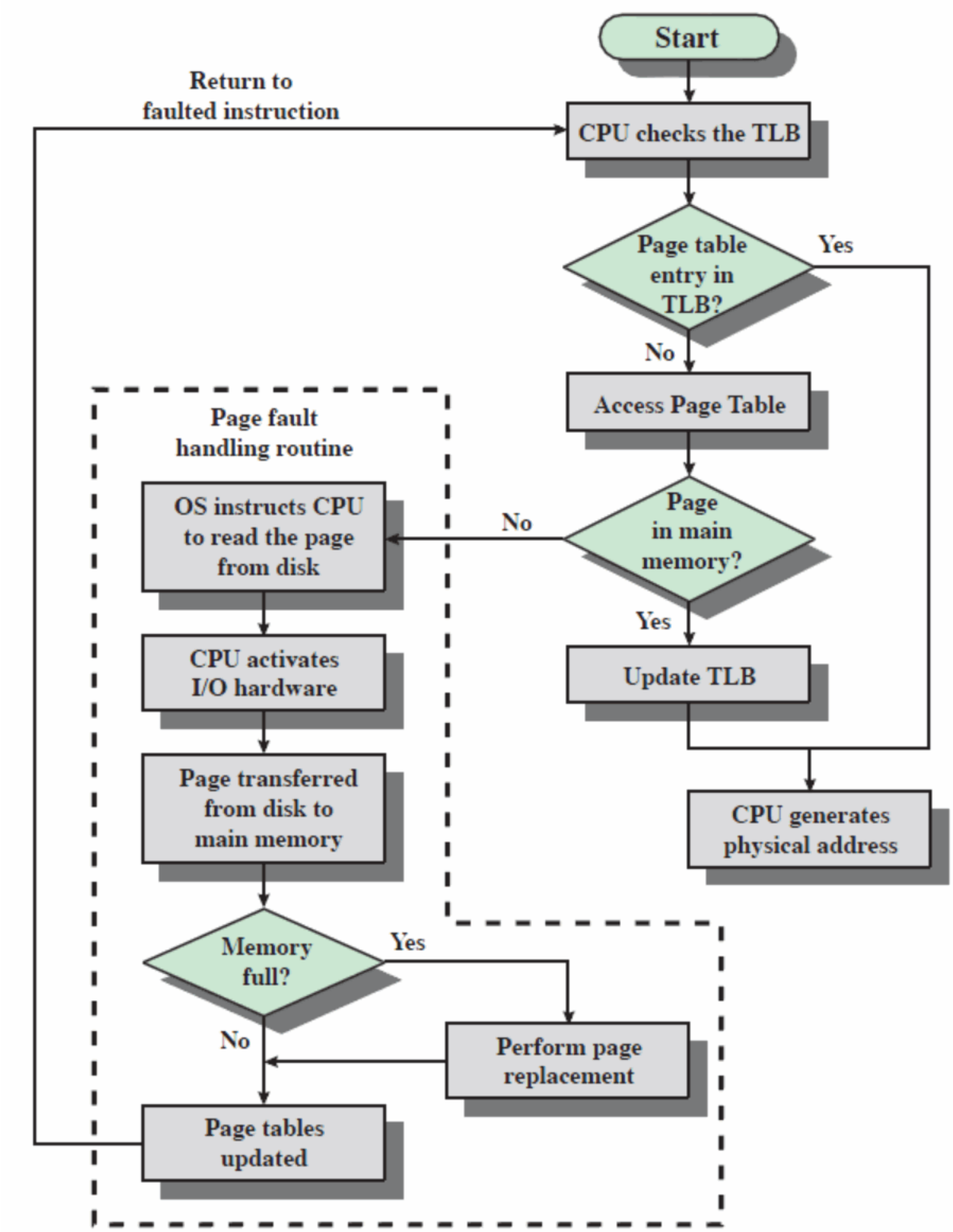
While the notion of cache and virtual memory seem similar there are differences:

	Cache	Virtual Memory
definition	cache is a subset of main memory	memory is a subset of what is on disk
purpose	speed	expand memory (provide relocatability)
data unit	line / block	page
implemented	hardware	hardware / software

Integrating Virtual Memory, TLBs, and Caches

Cache	TLB	Virtual memory	Possible? If so, under what circumstance?
miss	hit	hit	Possible, although the page table is never really checked if TLB hits.
hit	miss	hit	TLB misses, but entry found in page table; after retry, data is found in cache.
miss	miss	hit	TLB misses, but entry found in page table; after retry, data misses in cache.
miss	miss	miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
miss	hit	miss	Impossible; cannot have a translation in TLB if page is not present in memory.
hit	hit	miss	Impossible; cannot have a translation in TLB if page is not present in memory.
hit	miss	miss	Impossible; data cannot be allowed in cache if the page is not in memory.

Figure 8.18



To figure 8.19

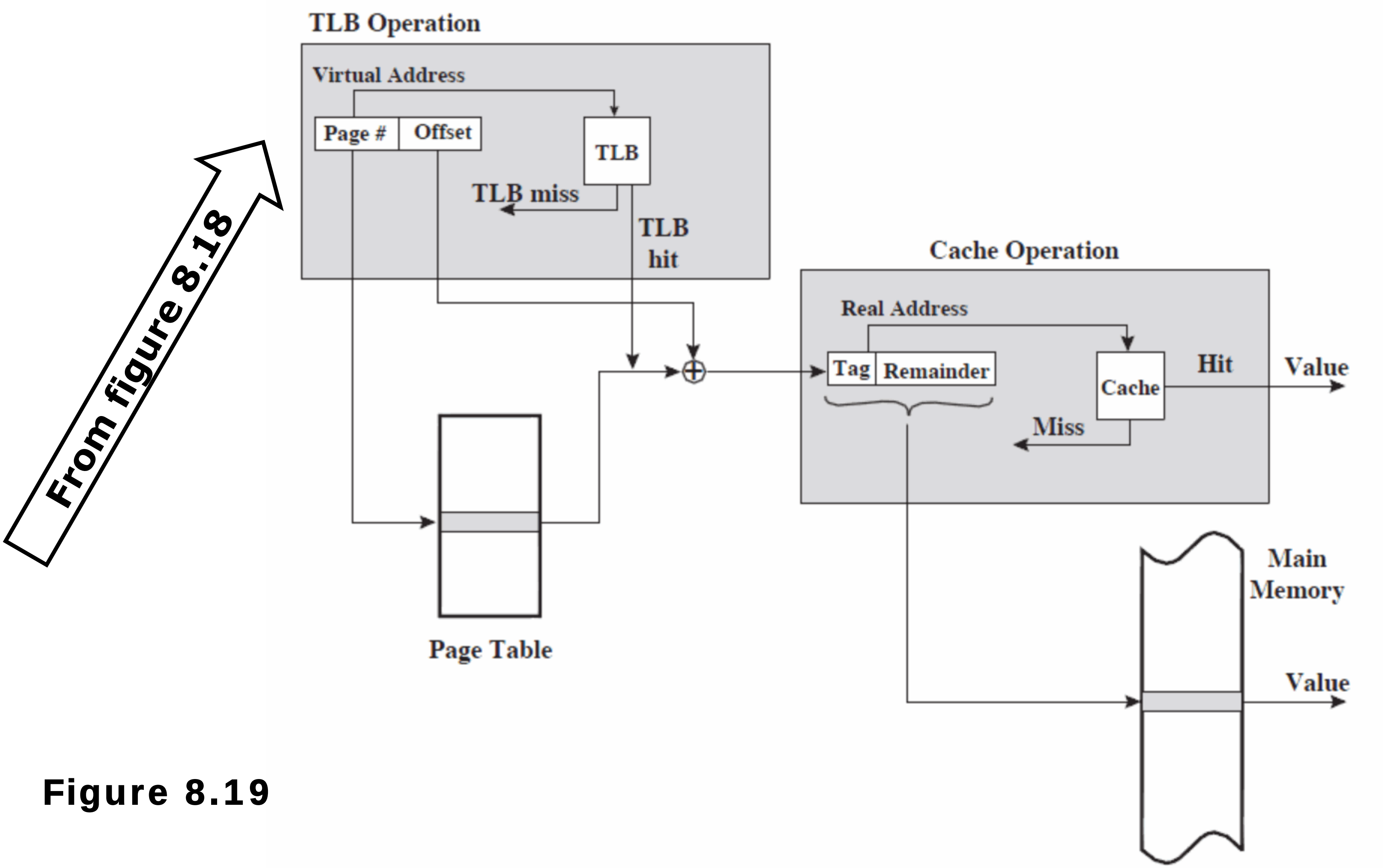
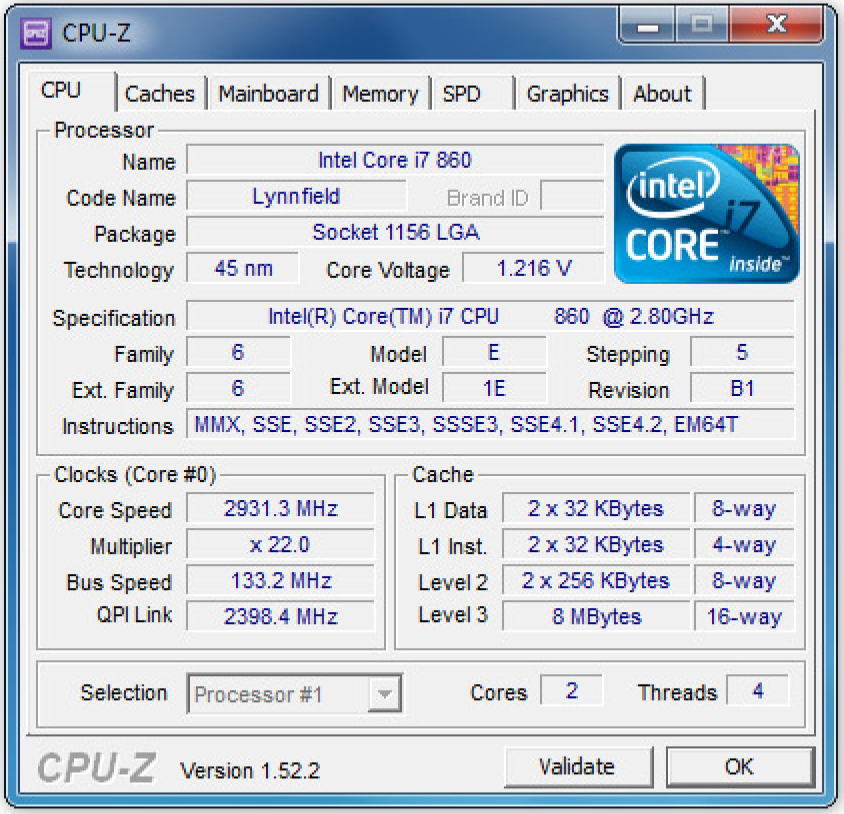

From figure 8.18
TLB Operation
Virtual Address
Page #
Offset
TLB
TLB miss
TLB
hit
Page Table
Cache Operation
Real Address
Tag
Remainder
Cache
Hit
Value
Miss
Main
Memory
Value

Figure 8.19

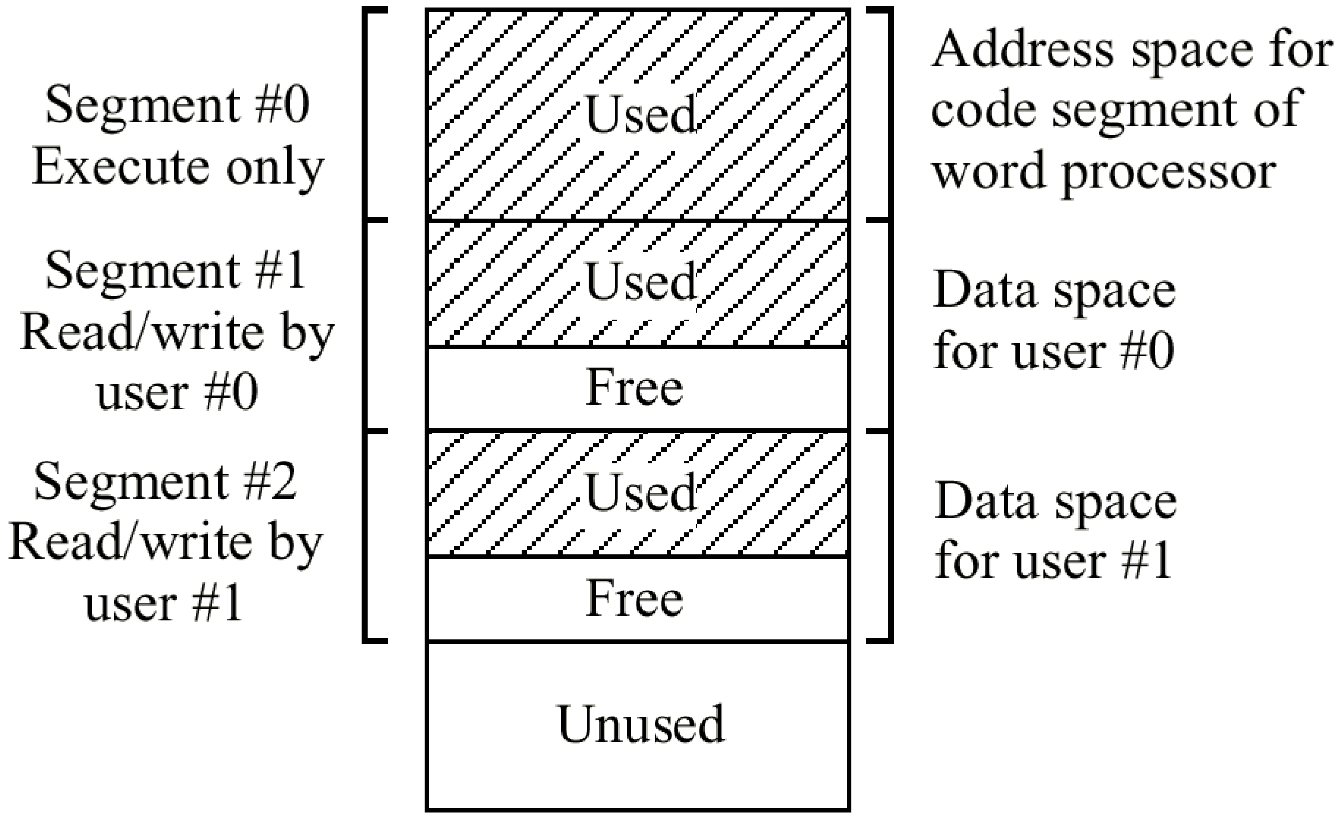
Useful utility: CPUZ



http://www.cpuid.com/softwares/cpu-z.html

Segmentation

A segmented memory allows two users to share the same word processor code, with different data spaces:

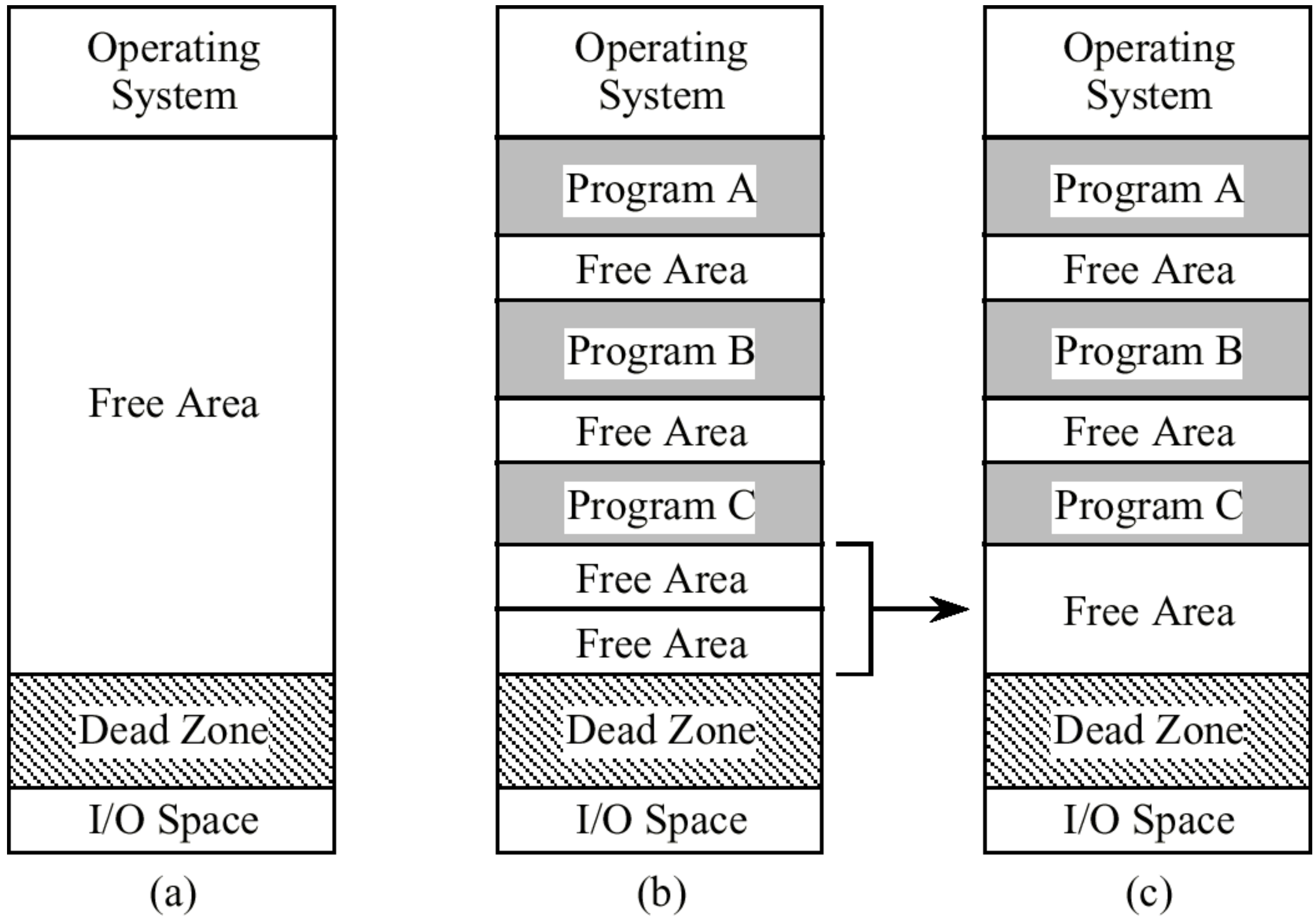


Fragmentation

(a) Free area of memory after initial-ization;

(b) after fragment-ation;

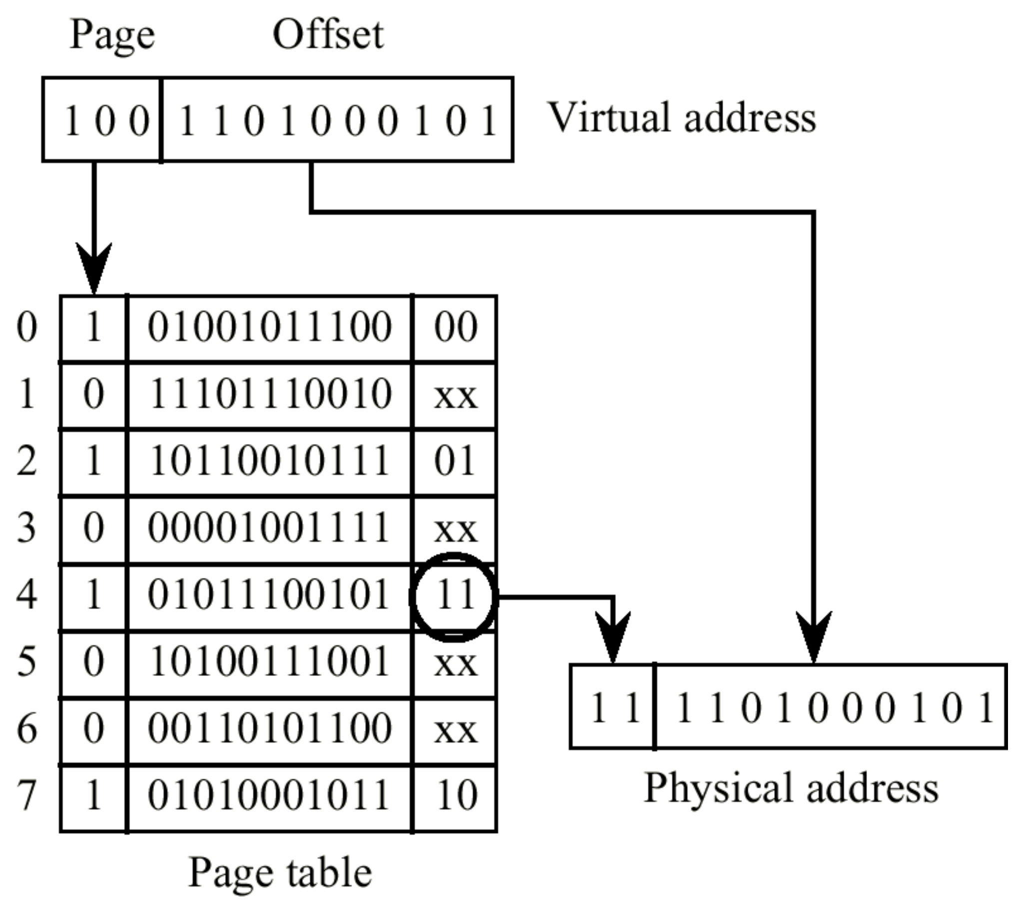
(c) after coalescing.



Using the Page Table (read from MH)

Virtual addresses are translated to physical addresses by breaking them into two parts:
1. page number
2. offset

- **The page number identifies a location in the page table which contains the current frame for the page**
- **The frame number and offset are combined to form a physical address.**



Using the Page Table - Example

Assume sequential access to pages 1, 2, 3, 4, and 5 (with 4 frames).

- **Table is initially empty**
- **When page is not in table, trigger a "page fault" and load the needed page**
- **Replace old pages when out of frames**

0	0	01001011100	xx
1	1	11101110010	00
2	0	10110010111	xx
3	0	00001001111	xx
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

After fault on page #1

0	0	01001011100	xx
1	1	11101110010	00
2	1	10110010111	01
3	0	00001001111	xx
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

0	0	01001011100	xx
1	1	11101110010	00
2	1	10110010111	01
3	1	00001001111	10
4	0	01011100101	xx
5	0	10100111001	xx
6	0	00110101100	xx
7	0	01010001011	xx

After fault on page #3

0	0	01001011100	xx
1	0	11101110010	xx
2	1	10110010111	01
3	1	00001001111	10
4	1	01011100101	11
5	1	10100111001	00
6	0	00110101100	xx
7	0	01010001011	xx