18-447: Computer Architecture Lecture 17: Virtual Memory II

Yoongu Kim
Carnegie Mellon University
Spring 2013, 2/27

Upcoming Schedule

- Friday (3/1): **Lab 3 Due**
- Friday (3/1): **Lecture/Recitation**
- Monday (3/4): **Lecture** – Q&A Session
- Wednesday (3/6): **Midterm 1**
  - 12:30 – 2:20
  - Closed book
  - One letter-sized cheat sheet
    - Can be double-sided
    - Can be either typed or written

Readings

- Required
  - P&H, Chapter 5.4
  - Hamacher et al., Chapter 8.8

- Recommended
  - *Denning, P. J.* ***Virtual Memory.*** *ACM Computing Surveys. 1970*
  - *Jacob, B., & Mudge, T.* ***Virtual Memory in Contemporary Microprocessors.*** *IEEE Micro. 1998.*

- References
  - Intel Manuals for 8086/80286/80386/IA32/Intel64

Review of Last Lecture

- The programer does not know *a priori* ...
  1. the **physical memory size** of the machine
     - what is the largest address that can be safely used?
  2. which **other programs** will be co-running on the machine
     - what if another program uses the same address?

- How to solve these two problems?

*"Any problem in computer science can be solved with another **level of indirection**." David Wheeler*

Review of Last Lecture (cont'd)

- **Virtual memory** is a level of indirection that ...
  1. Provides the illusion of a **large address space**
  2. This illusion is provided **separately** for each program

- Advantages of virtual memory
  1. Easier **memory management**
  2. Provides **memory isolation/protection**

*"At the heart [...] is the notion that 'address' is a concept **distinct** from 'physical location.'" Peter Denning*

Today's Lecture

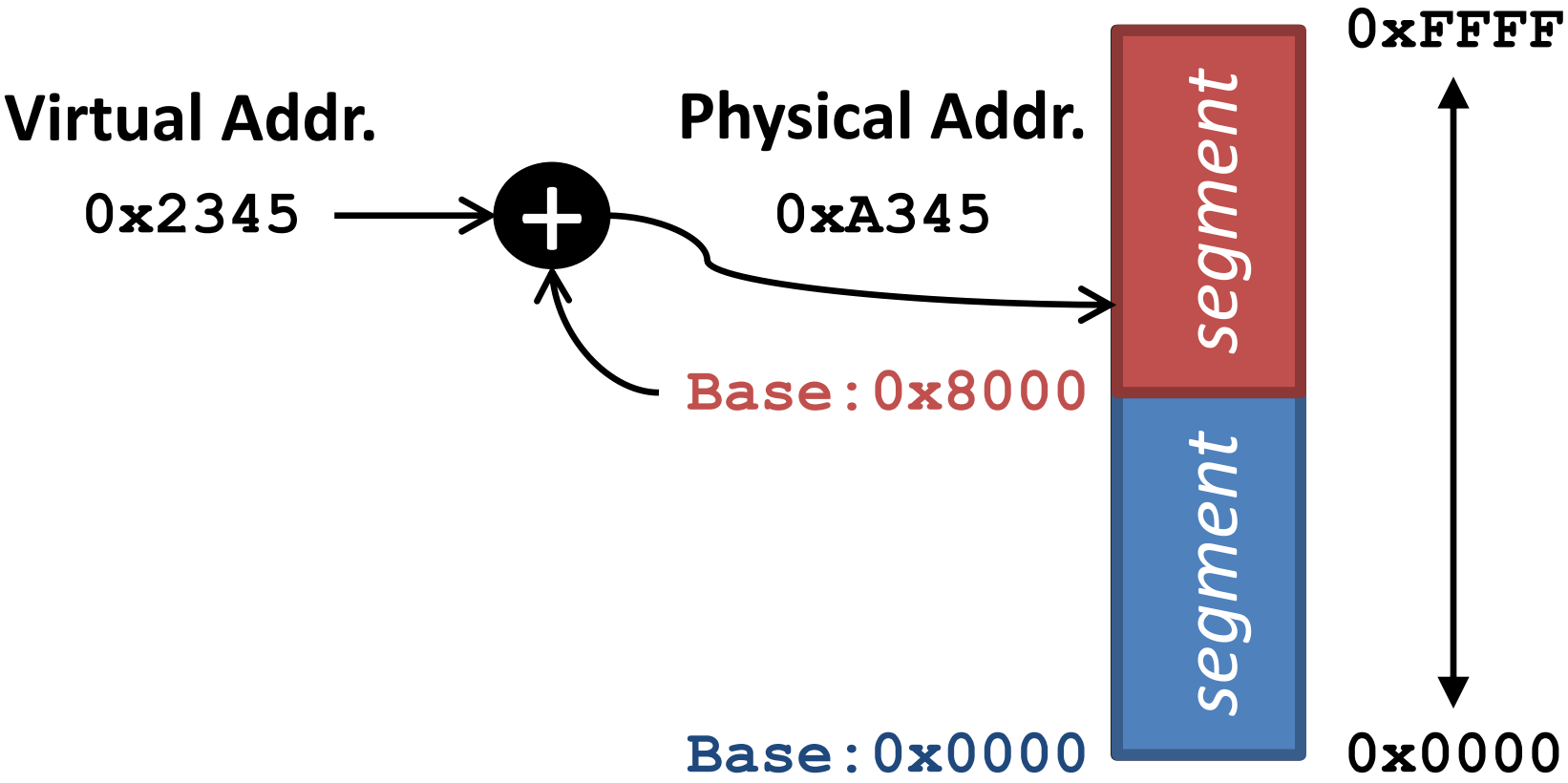
- Two approaches to virtual memory
  1. **Segmentation**
     - Not as popular today
  2. **Paging**
     - What is usually meant today by "virtual memory"

- Virtual memory requires HW+SW support
  - HW component is called the **MMU**
    - Memory management unit
  - How to **translate**: virtual ↔ physical addresses?

1. SEGMENTATION

Overview of Segmentation

- Divide the ***physical address space*** into ***segments***
  - The segments may overlap



Segmentation in Intel 8086

- ***Intel 8086*** (Late 70s)
  - 16-bit processor
  - 4 ***segment registers*** that store the ***base address***

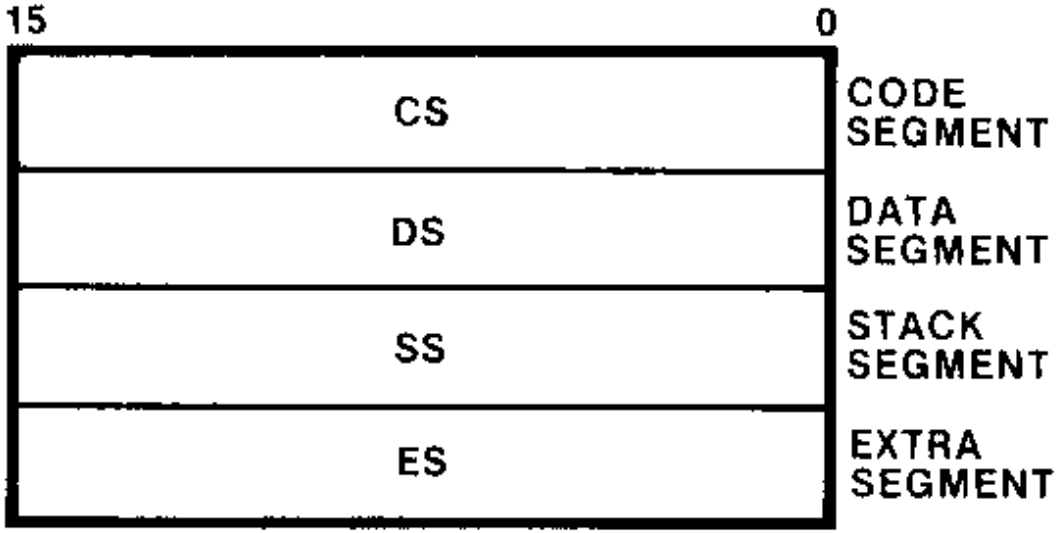
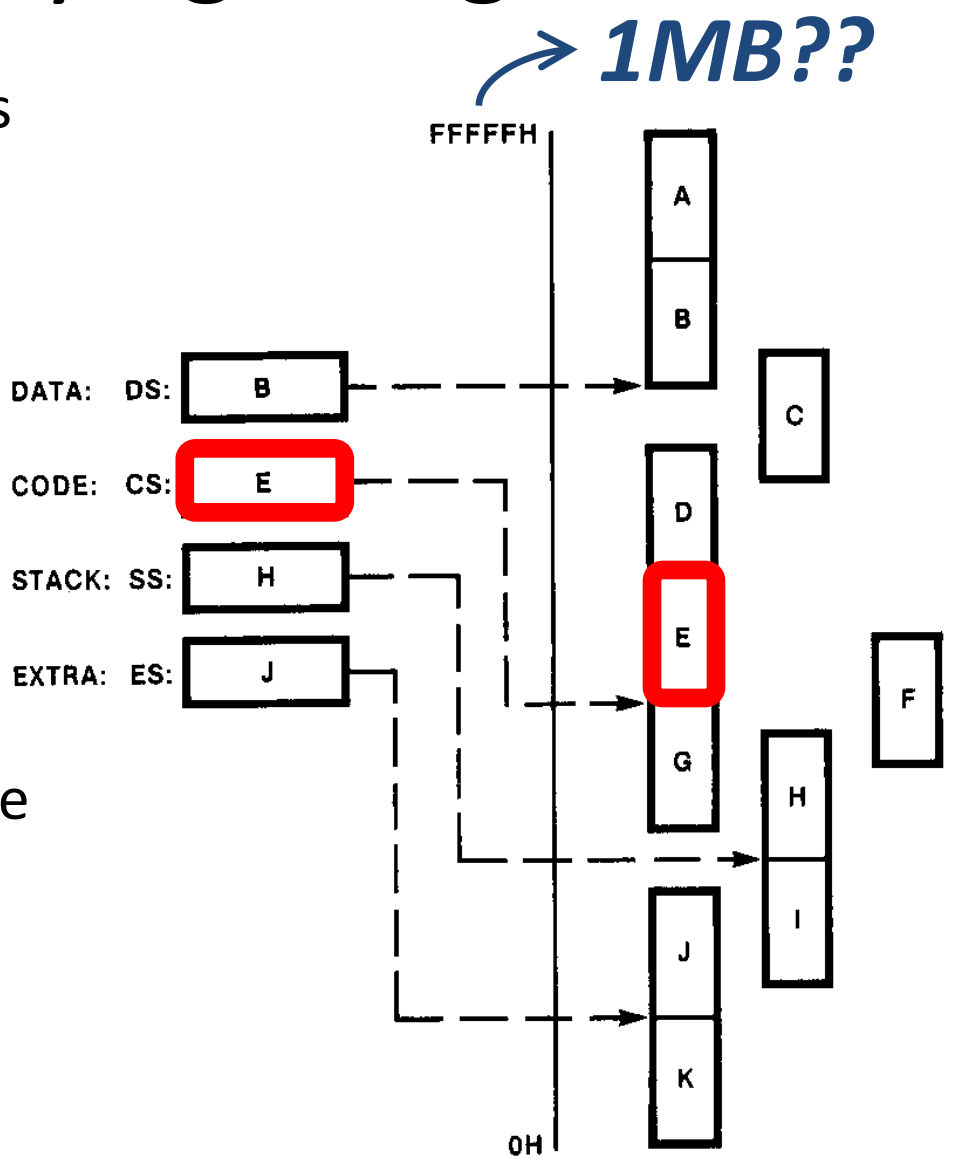


Figure 2-8. Segment Registers

Intel 8086: Specifying a Segment

- There can be many segments
- But only 4 of them are addressable at once
- Which 4 depends on the 4 segment registers
  - The programmer sets the segment register value
- Each segment is 64KB in size
  - Because 8086 is 16-bit



Intel 8086: Translation

- 8086 is a 16-bit processor ...
  - How can it address up to `0xFFFFF` (1MB)?

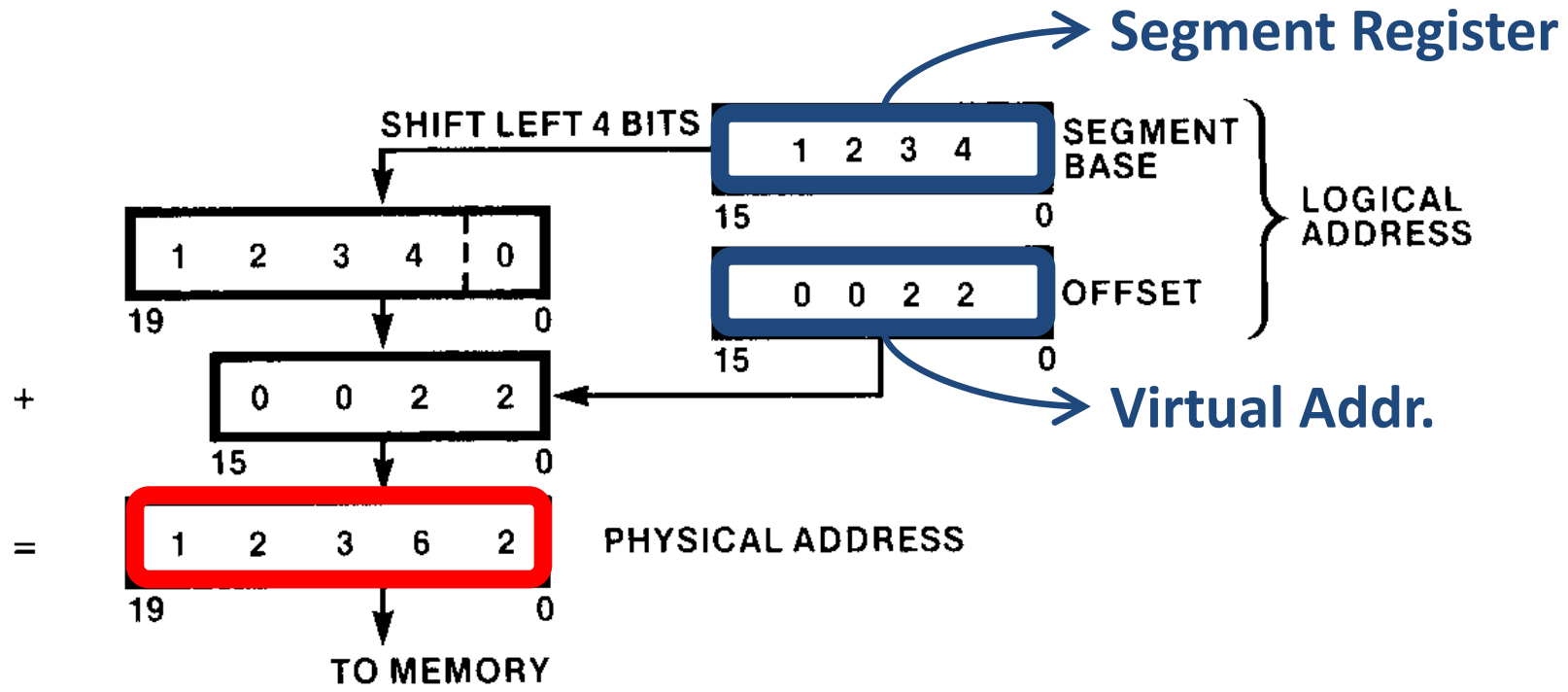


Figure 2-18. Physical Address Generation

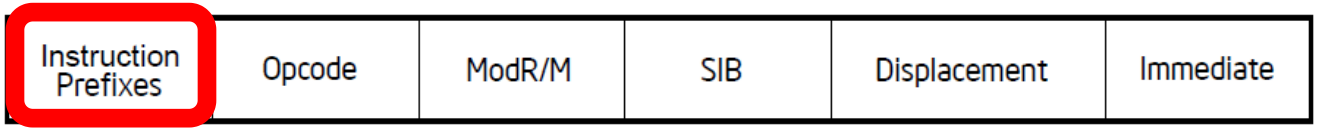
Intel 8086: Which Segment Register?

- **Q:** For a memory access, how does the machine know which of the 4 segment register to use?
  - **A:** Depends on the *type of memory access*

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction Fetch	CS	NONE	IP
Stack Operation	SS	NONE	SP
Variable (except following)	DS	CS,ES,SS	Effective Address
String Source	DS	CS,ES,SS	SI
String Destination	ES	NONE	DI
BP Used As Base Register	SS	CS,DS,ES	Effective Address

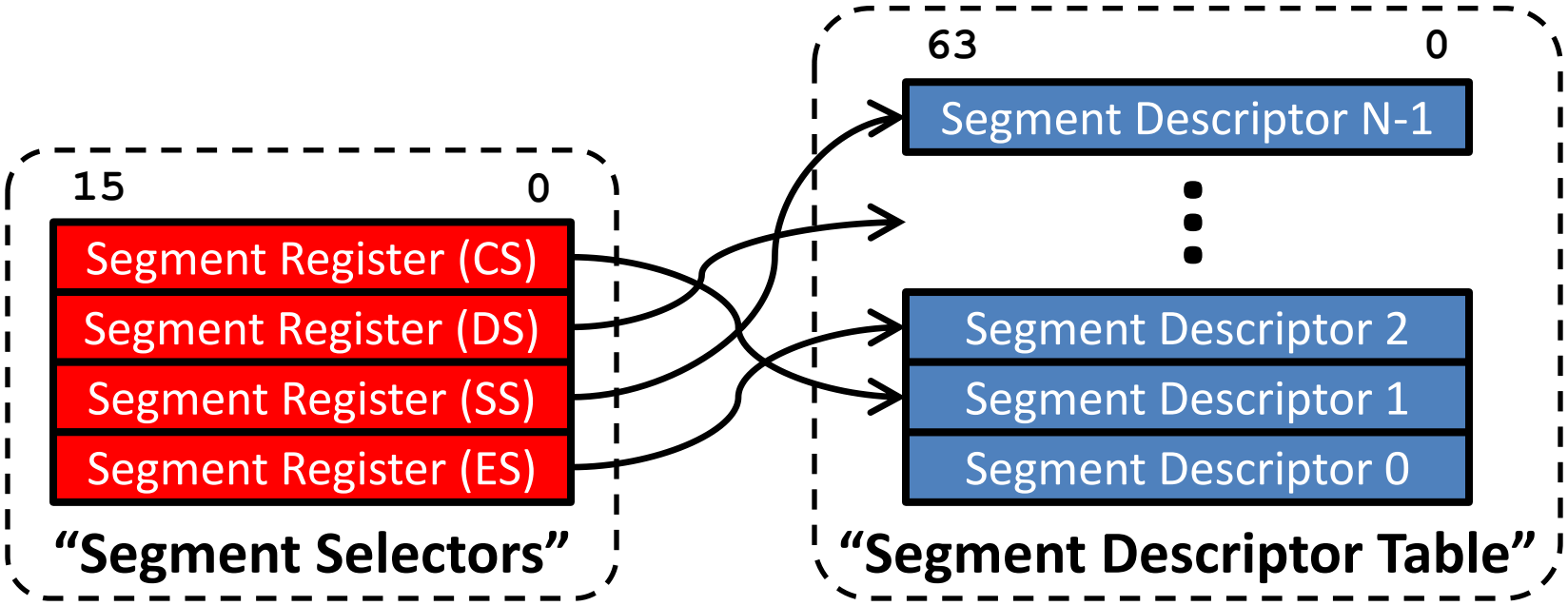
  - Can be overriden: `mov %AX,(%ES:0x1234)`

x86 Instruction



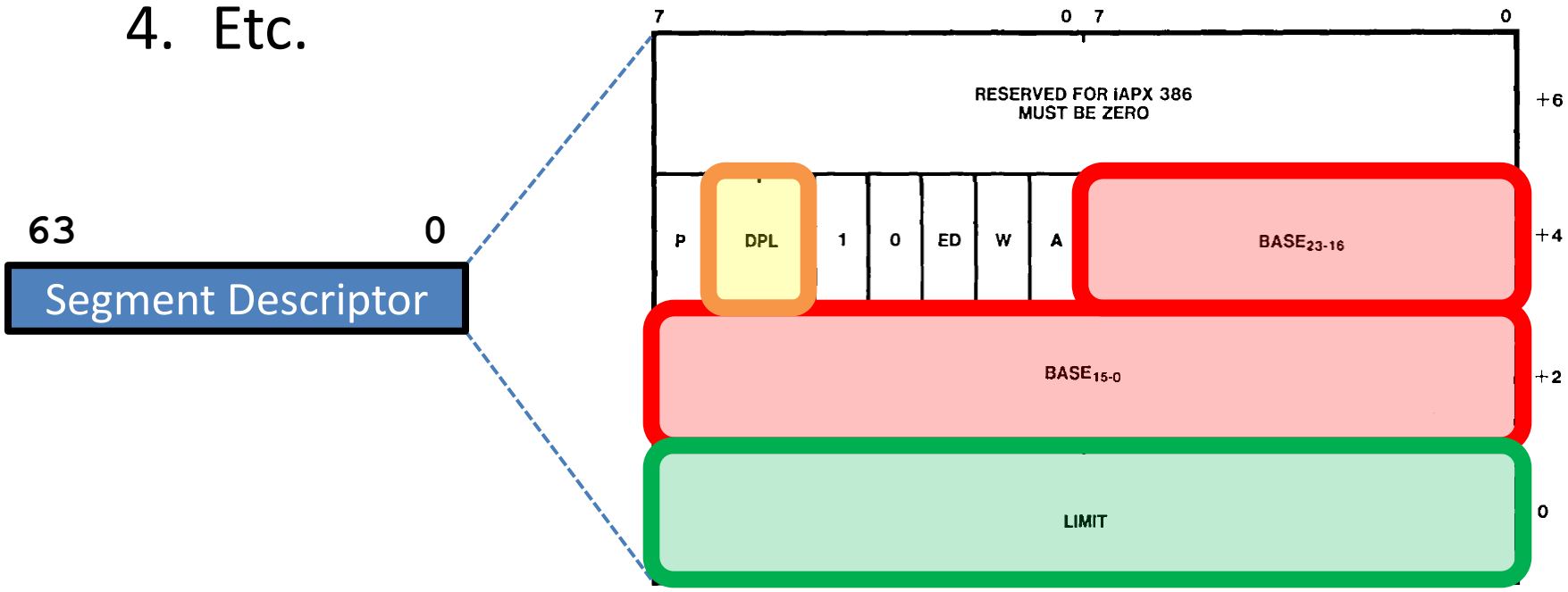
Segmentation in Intel 80286

- ***Intel 80286*** (Early 80s)
  - Still a 16-bit processor
  - Still has 4 segment registers that ...
    - stores the ***index*** into a table of base addresses
    - not the base address itself



Intel 80286: Segment Descriptor

- A ***segment descriptor*** describes a segment:
  1. **BASE**: Base address
  2. **LIMIT**: The size of the segment
  3. **DPL**: Descriptor Privilege Level (!!)
  4. Etc.



Intel 80286: Translation

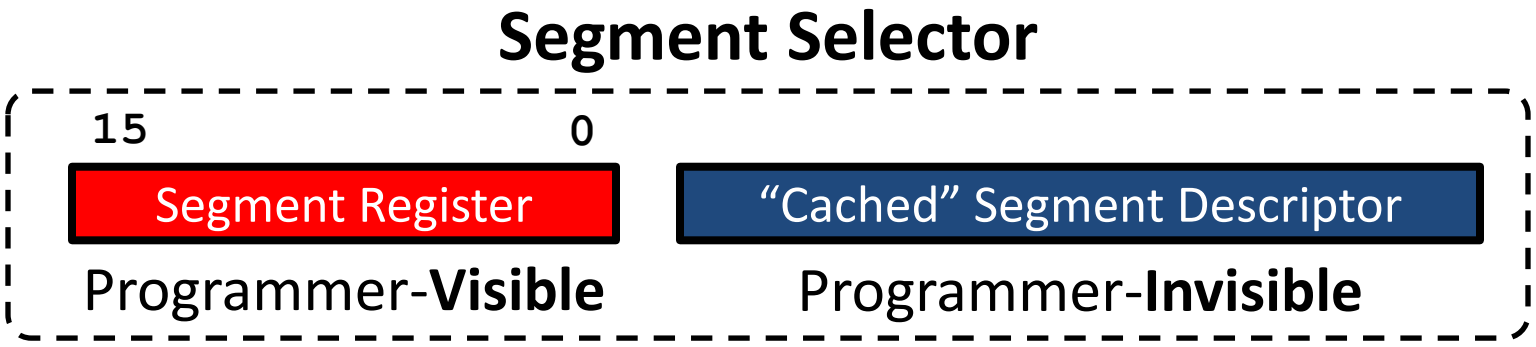
- Example: `mov %AX,(0x1234)`
  1. Accesses the **data segment** (unless otherwise specified)
  2. `DS` is the **segment selector** for the data segment
  3. `DS` points to a particular **segment descriptor** within the **segment descriptor table**
  4. The segment descriptor specifies **`BASE`** and **`LIMIT`**
     - Virtual address: `0x1234`
     - `assert(0x1234 ≤ LIMIT);`
     - Physical address: `BASE+0x1234`

- Also referred to as *"base-and-bound"*

Intel 80286: Accelerating Translation

- **Segment selectors**: stored in registers (fast)
- **Segment descriptors**: stored in memory (slow)
  - Before every memory access, always fetch the segment descriptor from memory? ➔ Large performance penalty

- **Solution**: "Cache" the segment descriptor as part of the segment selector



Intel 80286: Privilege Levels

- Four **privilege levels** in x86 (referred to as "**rings**")
  - Ring 0: Highest privilege (operating system)
  - Ring 1: Not widely used
  - Ring 2: Not widely used
  - Ring 3: Lowest privilege (user applications)

- Let us assume that you are currently at Ring 3 ...
  - In other words, your **Current Privilege Level** (CPL) = 3
  - Then, you can access only the segments whose **Descriptor Privilege Level** (DPL) is 3
    - You cannot access segments whose DPL < 3

Intel 80286: Privilege Levels (cont'd)

- What's my CPL?
  - Assume that the CS points to a segment descriptor
  - Assume that the DPL field in this segment descriptor is **N**
  - This means that your CPL is **N**
    - (Not really; CPL == DPL in the "cached" segment descriptor)
- What can I do if my CPL = 0?
  - You are in "kernel mode"
  - Can access all segments
  - Can execute all x86 instructions, even the privileged ones
- How do I change my CPL?
  - **System calls**: referred to as "software interrupts"
  - We will not go into detail

Fast Forward to Today (2013)

- Modern x86 Machines
  - 32-bit x86: Segmentation is similar to 80286
  - 64-bit x86: Segmentation is not supported *per se*
    - Forces the `BASE=0x0000000000000000`
    - Forces the `LIMIT=0xFFFFFFFFFFFFFFFF`
    - But `DPL` is still supported
- Side Note: Linux & 32-bit x86
  - Linux does not use segmentation *per se*
    - For all segments, Linux sets `BASE=0x00000000`
    - For all segments, Linux sets `LIMIT=0xFFFFFFFF`
  - Instead, Linux uses segments for privilege levels
    - For segments used by the kernel, Linux sets DPL = 0
    - For segments used by the applications, Linux sets DPL = 3

Summary of Segmentation

- **Summary**: Divide the address space into segments
  - Modularity: Different pieces of a program in different segments
  - Isolation: Different programs in different segments
  - Protection: Privilege levels
- **Advantages**
  - Translation is easy: Simple addition
  - Provides modularity, isolation, and protection
- **Disadvantages**
  - Susceptible to **fragmentation**
    - Segments are relatively large
    - Large contiguous regions of unoccupied memory may not be found
  - Only a few segments are addressable at the same time
  - Complicated management
    - Overlapping, differently-sized segments
    - Programmer has to change the value of the segment base/limit

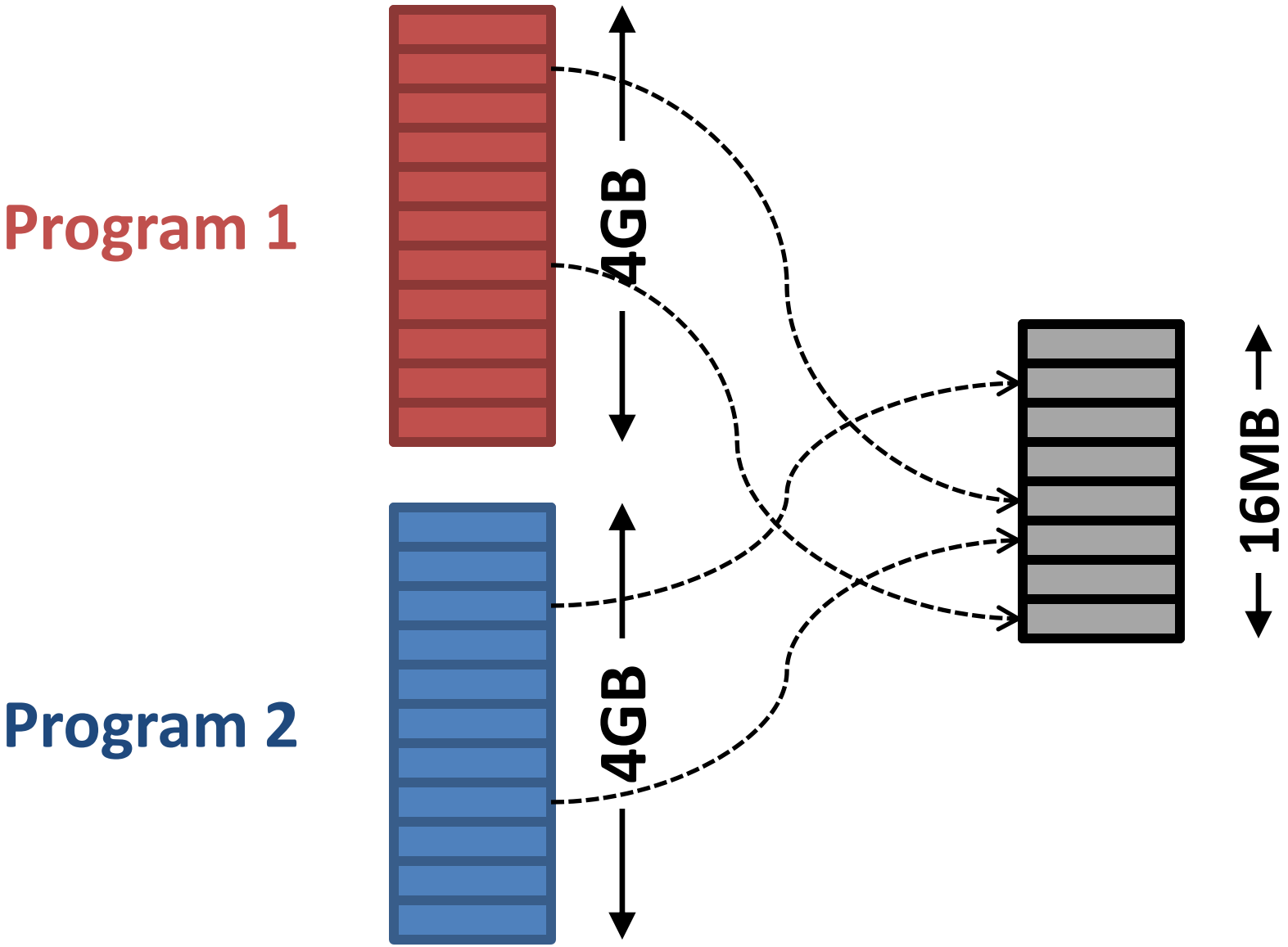
Today's Lecture

- Two approaches to virtual memory
  1. **Segmentation**
     - Not as popular today
  2. **Paging**
     - What is usually meant today by "virtual memory"

- Virtual memory requires HW+SW support
  - HW component is called the **MMU**
    - Memory management unit
  - How to **translate**: virtual ↔ physical addresses?

2. PAGING

Overview of Paging



Overview of Paging (cont'd)

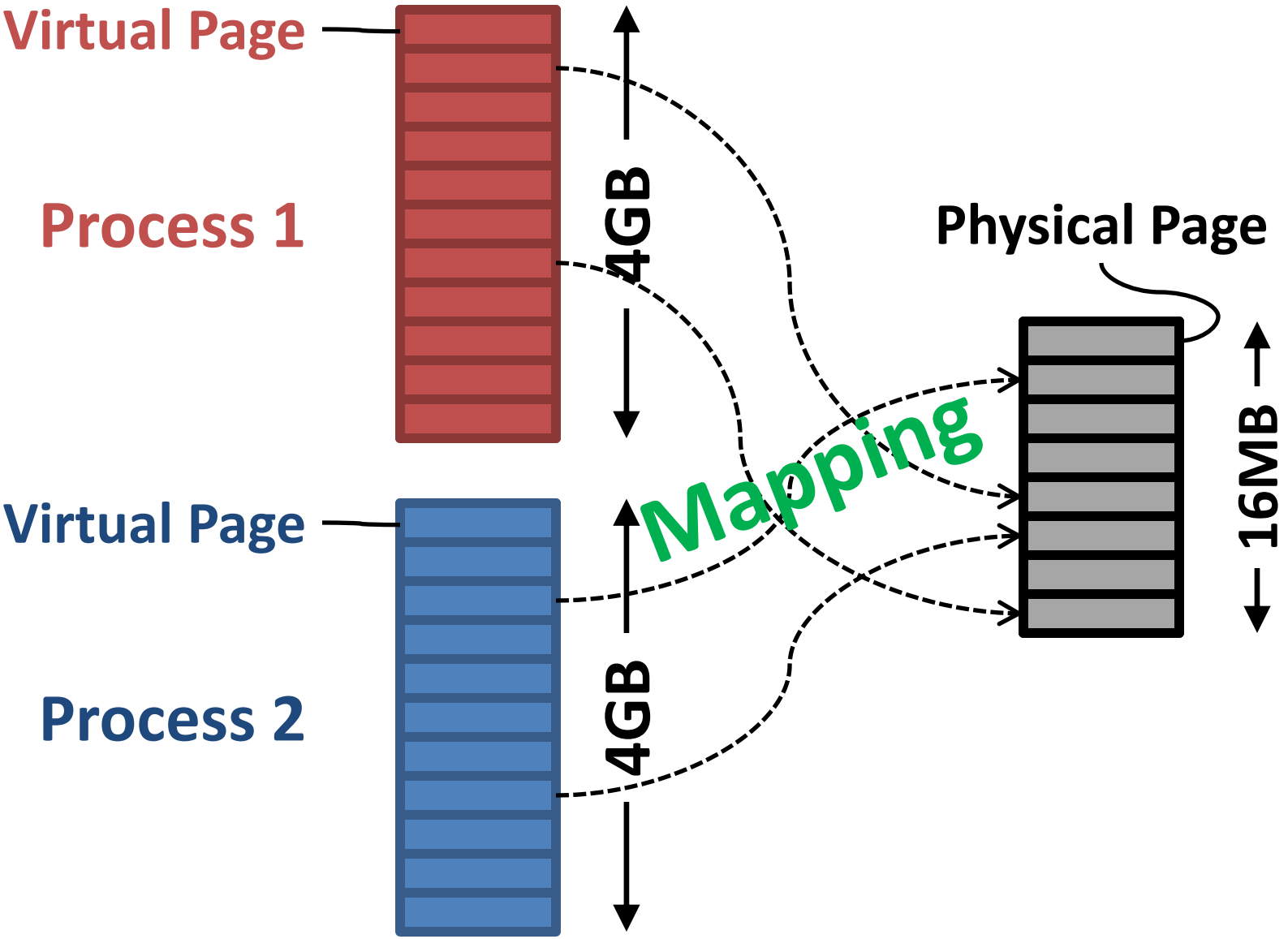
1. Based on the notion of a **virtual address space**
   - A large, contiguous address space that is only an illusion
     - Virtual address space >> Physical address space
   - Each "program" gets its own separate virtual address space
     - Each **process**, not each thread

2. Divide the address spaces into fixed-sized **pages**
   - **Virtual page**: A "chunk" of the virtual address space
   - **Physical page**: A "chunk" of the physical address space
     - Also called a **frame**
   - Size of virtual page == Size of physical page

Overview of Paging (cont'd)

3. **Map** virtual pages to physical pages
   - By itself, a virtual page is merely an illusion
     - Cannot actually store anything
     - Needs to be backed-up by a physical page
   - Before a virtual page can be accessed …
     - It must be paired with a physical page
     - I.e., it must be **mapped** to a physical page
     - This mapping is stored somewhere
   - On every subsequent access to the virtual page …
     - Its mapping is looked up
     - Then, the access is directed to the physical page

Overview of Paging (cont'd)

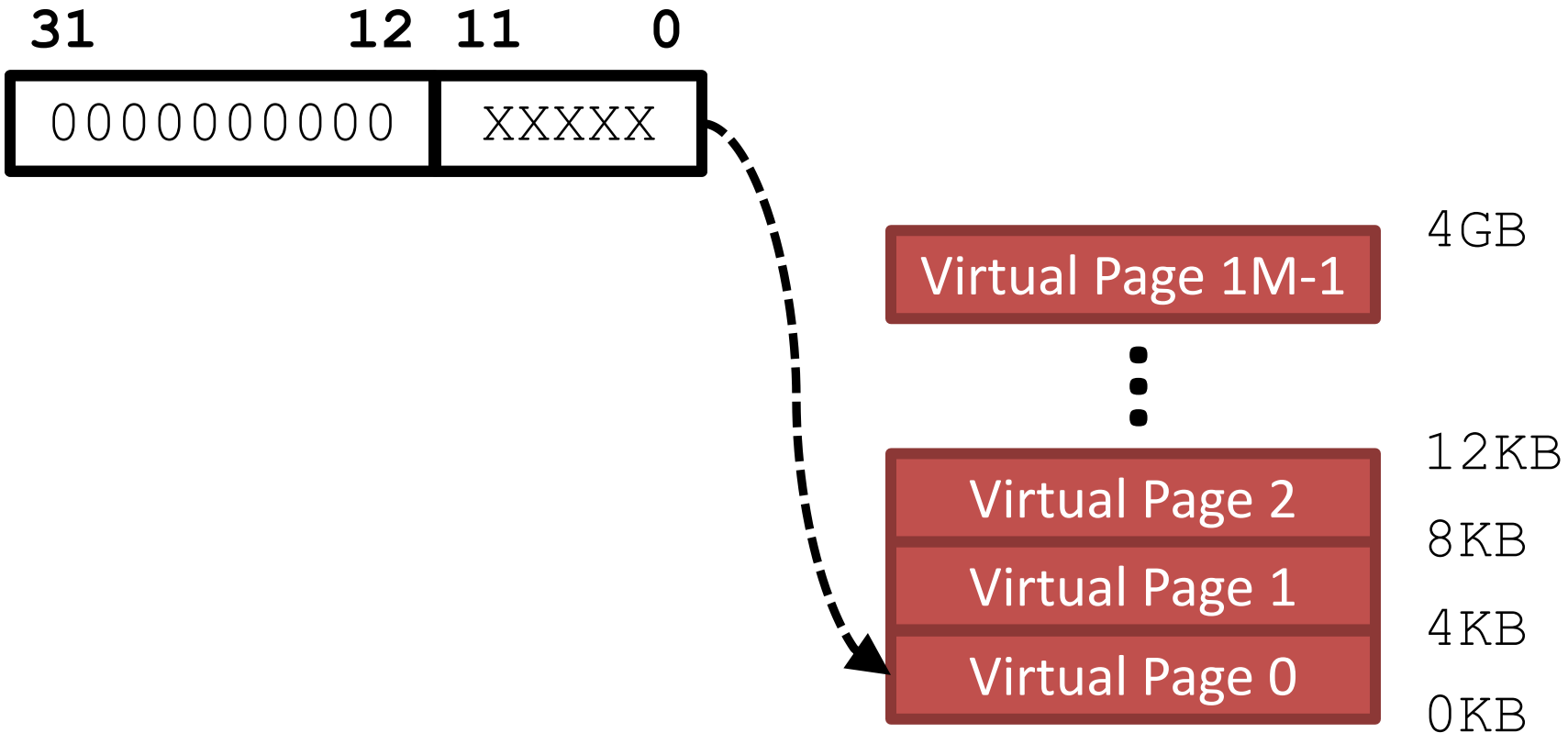


Paging in Intel 80386

- ***Intel 80386*** (Mid 80s)
  - 32-bit processor
  - 4KB virtual/physical pages
- **Q:** What is the size of a virtual address space?
  - **A:** 2^32 = 4GB
- **Q:** How many virtual pages per virtual address space?
  - **A:** 4GB/4KB = 1M
- **Q:** What is the size of the physical address space?
  - **A:** Depends… but less than or equal to 4GB
- **Q:** How many physical pages in the physical address space?
  - **A:** Depends… but less than or equal to 1M
  - But let us assume that physical addresses are still 32 bits

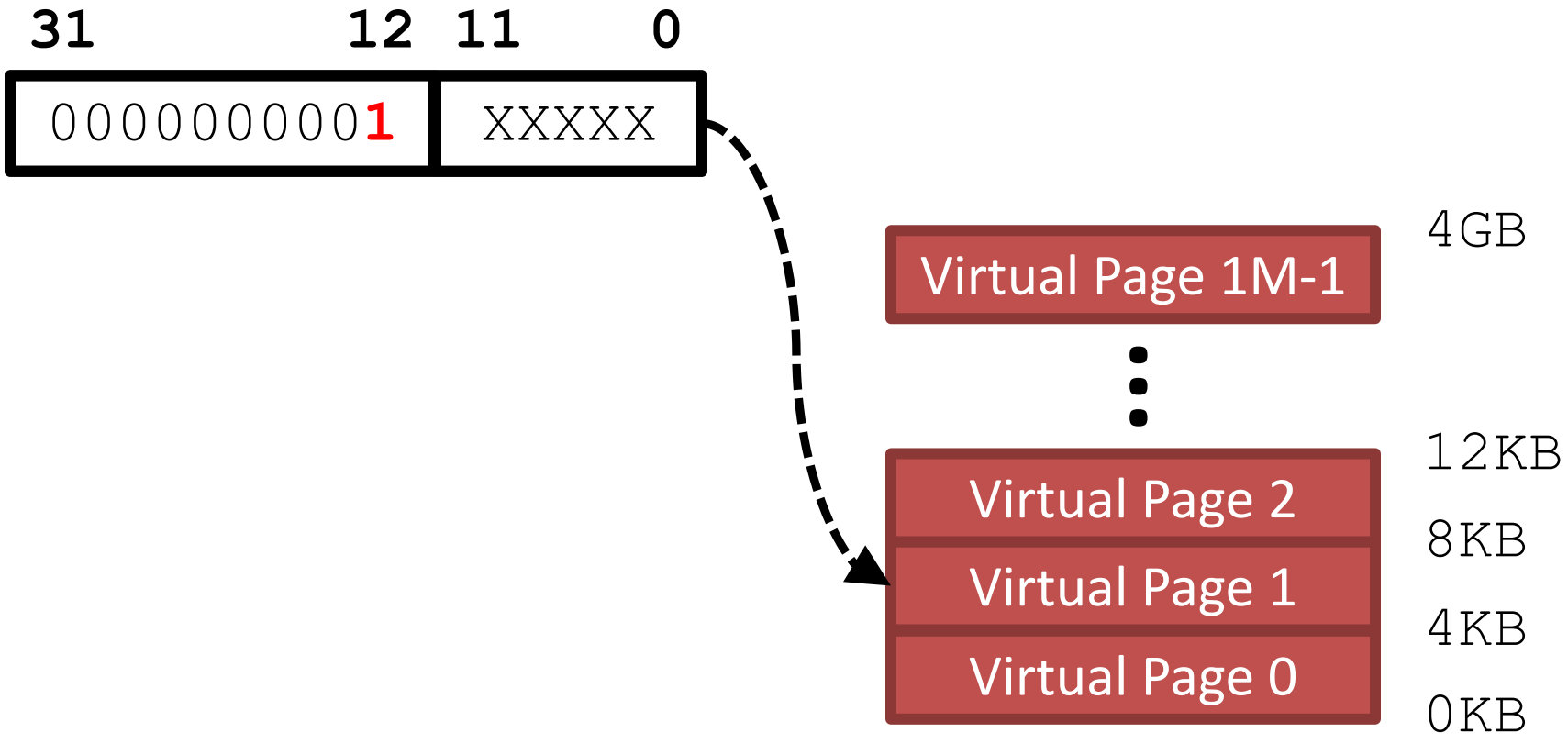
Intel 80386: Virtual Pages

32-bit Virtual Address



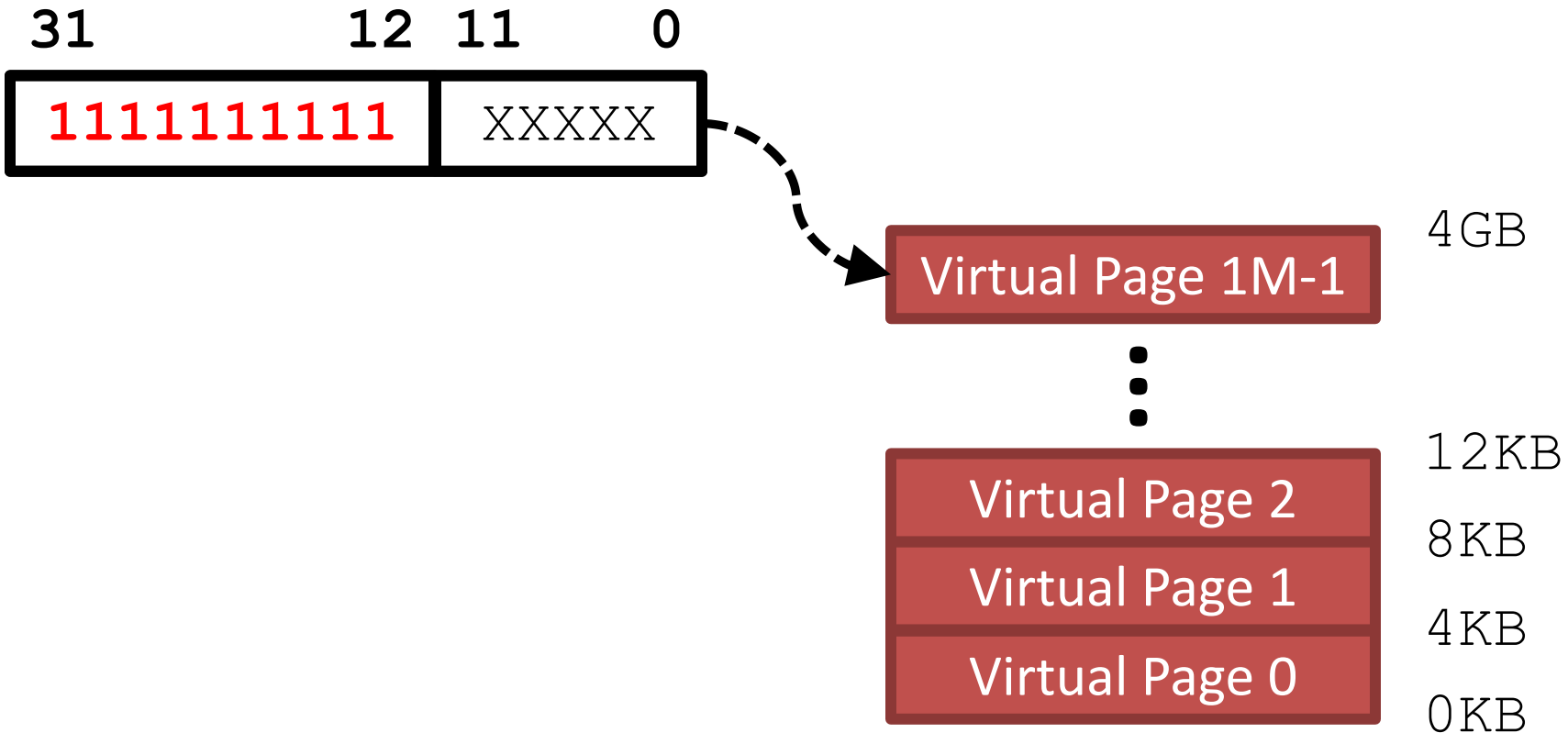
Intel 80386: Virtual Pages

32-bit Virtual Address



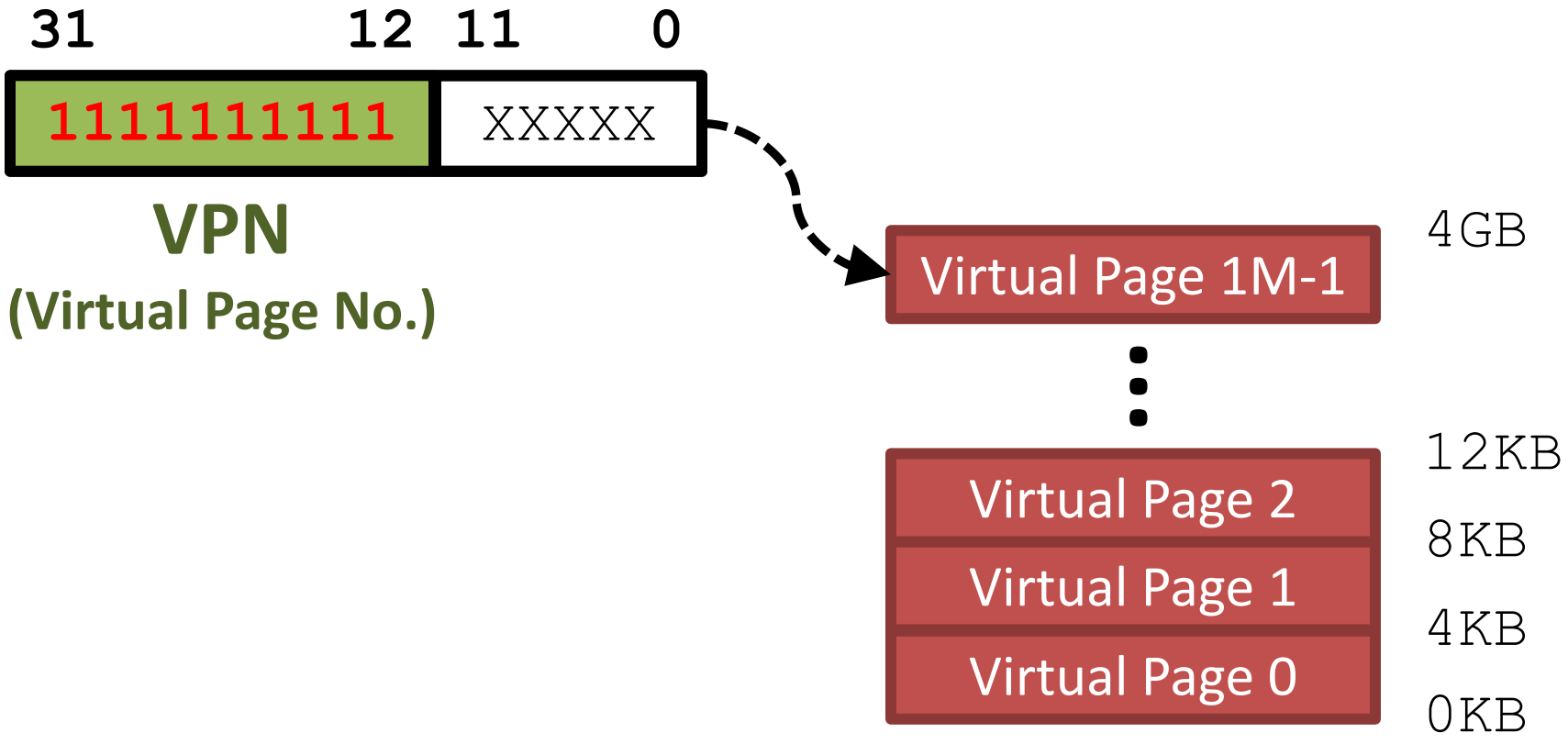
Intel 80386: Virtual Pages

32-bit Virtual Address



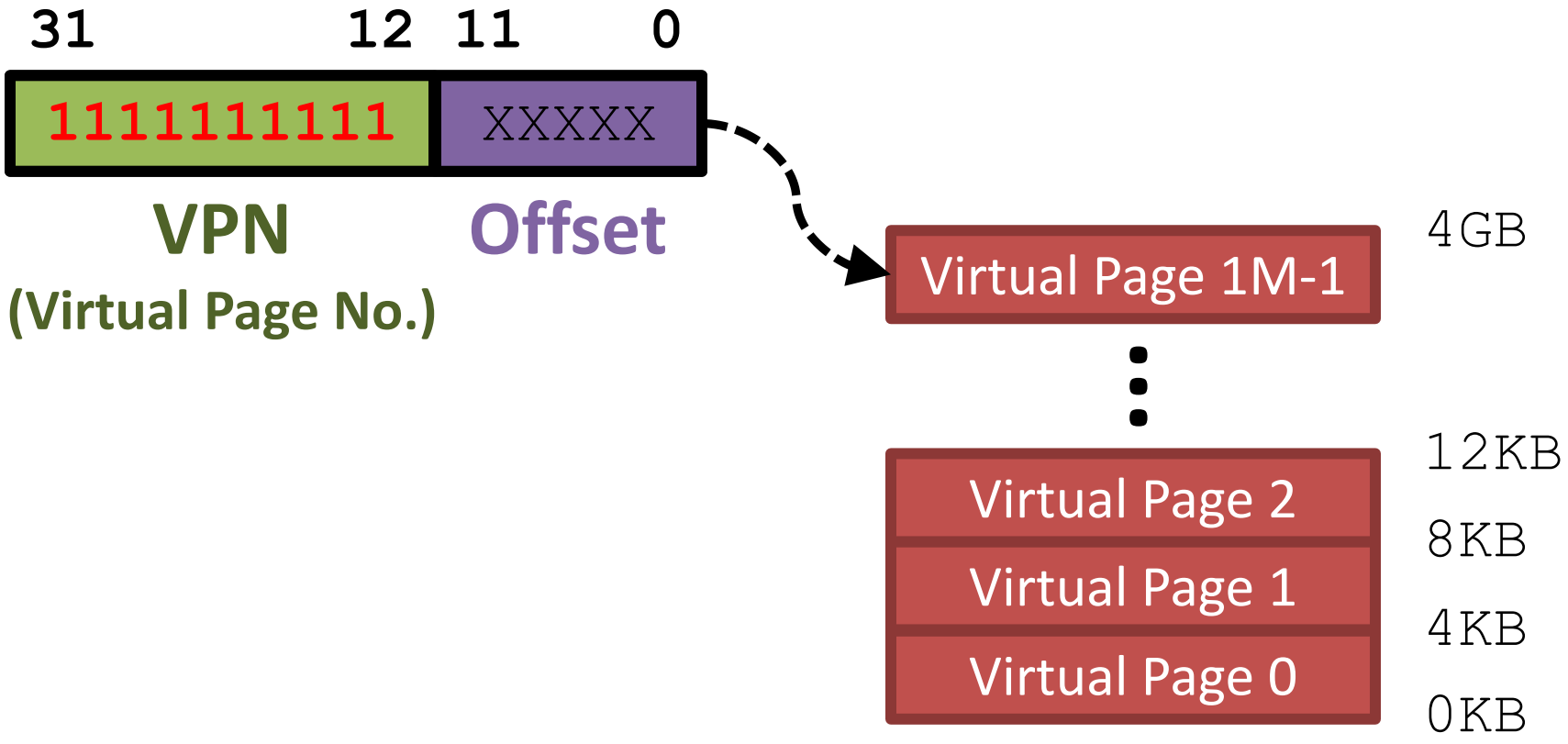
Intel 80386: Virtual Pages

32-bit Virtual Address



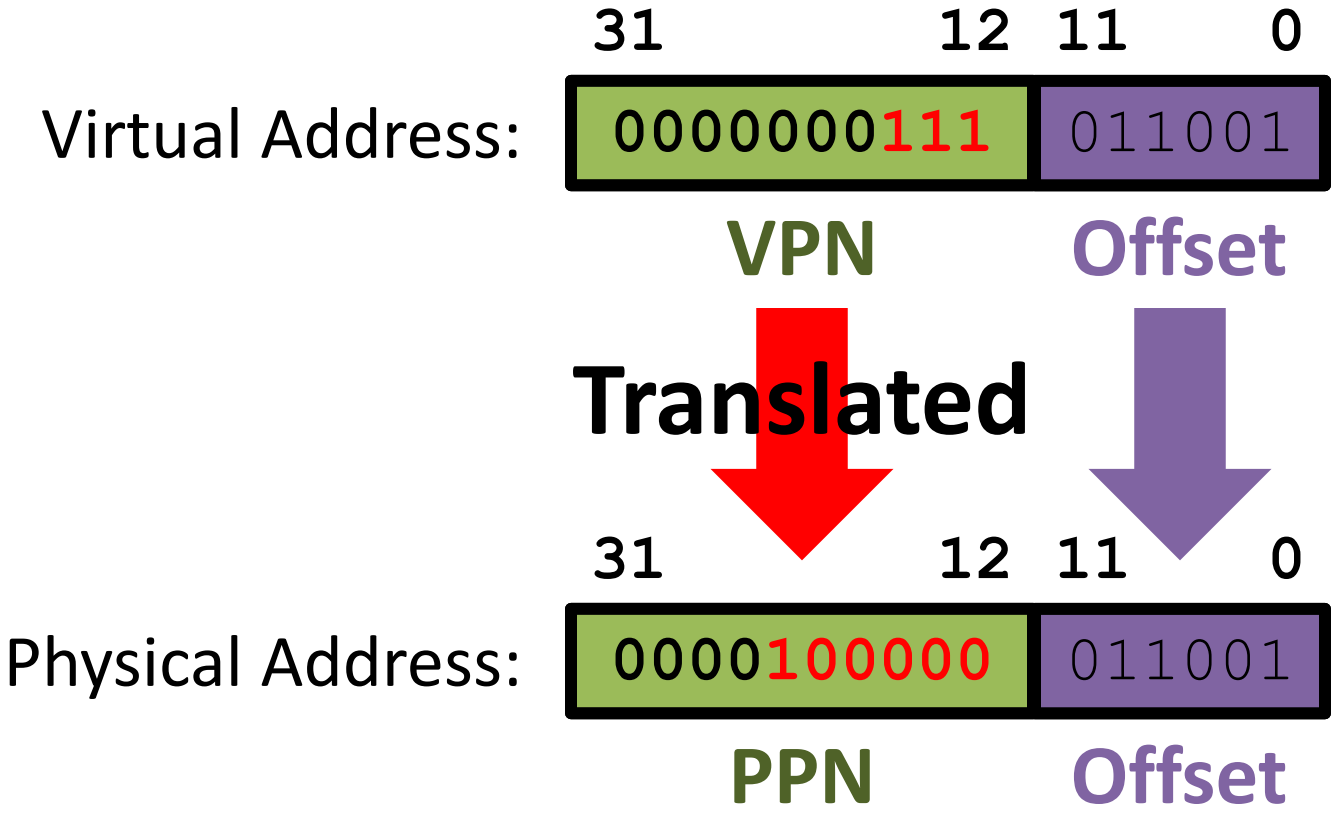
Intel 80386: Virtual Pages

32-bit Virtual Address



Intel 80386: Translation

- Assume: Virtual Page 7 is mapped to Physical Page 32
- For an access to Virtual Page 7 …



Intel 80386: VPN → PPN

- How to keep track of VPN → PPN mappings?
  - VPN 65 → PPN 981,
  - VPN 3161 → PPN 1629,
  - VPN 9327 → PPN 524, …
- **Page Table:** A "lookup table" for the mappings
  - Can be thought of as an array
  - Each element in the array is called a **page table entry** (PTE)

```
uint32 PAGE_TABLE[1<<20];
PAGE_TABLE[65]=981;
PAGE_TABLE[3161]=1629;
PAGE_TABLE[9327]=524; ...
```

Intel 80386: Two Problems

- Two problems with page tables

- **Problem #1: Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s

- Problem #2: Page table is stored in memory
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**

Intel 80386: Page Table Too Large

- Typically, the vast majority of PTEs are empty

```
PAGE_TABLE[0]=141;
...
PAGE_TABLE[532]=1190;
PAGE_TABLE[534]=NULL;
...                                  empty
PAGE_TABLE[1048401]=NULL;
PAGE_TABLE[1048402]=845;
...
PAGE_TABLE[1048575]=742; // 1048575=(1<<20)-1;
```

  - **Q:** Why? - **A:** Virtual address space is extremely large

- Typically, empty PTEs are clustered together
  - **Q:** Why? - **A:** Stack vs. heap

Intel 80386: Page Table Too Large

- **Solution:** "Unallocate" the empty PTEs to save space

```
PAGE_TABLE[0]=141;
...
PAGE_TABLE[532]=1190;
```

Unallocated

```
PAGE_TABLE[1048402]=845;
...
PAGE_TABLE[1048575]=742; // 1048575=(1<<20)-1;
```

- Unallocating every single empty PTE is tedious
  - Instead, unallocate only long stretches of empty PTEs

Intel 80386: Page Table Too Large

- To allow PTEs to be “unallocated” …
  - the page table must be restructured
- Before restructuring: **flat**

```
uint32 PAGE_TABLE[1024*1024];
uint32 PAGE_TABLE[0]=423;
uint32 PAGE_TABLE[1023]=381;
```

- After restructuring: **hierarchical**

```
uint32 *PAGE_DIRECTORY[1024];
PAGE_DIRECTORY[0]=malloc(sizeof(uint32)*1024);
PAGE_DIRECTORY[0][0]=423;
PAGE_DIRECTORY[0][1023]=381;
PAGE_DIRECTORY[1]=NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[2]=NULL; // 1024 PTEs unallocated
```

Intel 80386: Two Problems

- Two problems with page tables
- **Problem #1: Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s
  - **Solution:** Hierarchical page table
- **Problem #2: Page table is in memory**
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**

Intel 80386: Accelerating Translation

- Retrieving PTEs from the memory is slow …

- **Solution**: "Cache" the PTEs inside the processor
  - **Translation Lookaside Buffer** (TLB)
    - "Lookaside Buffer" is an old term for cache
  - 32-entry TLB for 80386
  - Each TLB entry consists of a **tag** and **data**
    1. Tag: 20-bit VPN + 4-bit metadata
    2. Data: 20-bit PPN

Intel 80386: Two Problems

- Two problems with page tables
- **Problem #1: Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (because 4B ≈ 20-bit PPN)
  - Page table = 4MB (!!)
    - very expensive in the 80s
  - **Solution:** Hierarchical page table
- **Problem #2: Page table is in memory**
  - Before every memory access, always fetch the PTE from the slow memory? ➔ **Large performance penalty**
  - **Solution:** Translation Lookaside Buffer

Next Lecture

- More on paging:
  - Trade-offs in page size
  - PTEs, PDEs & Page-level protection
  - Demand paging & Page faults
  - Thrashing & Replacement
  - Handling TLB Misses
  - Context switches & Homonyms