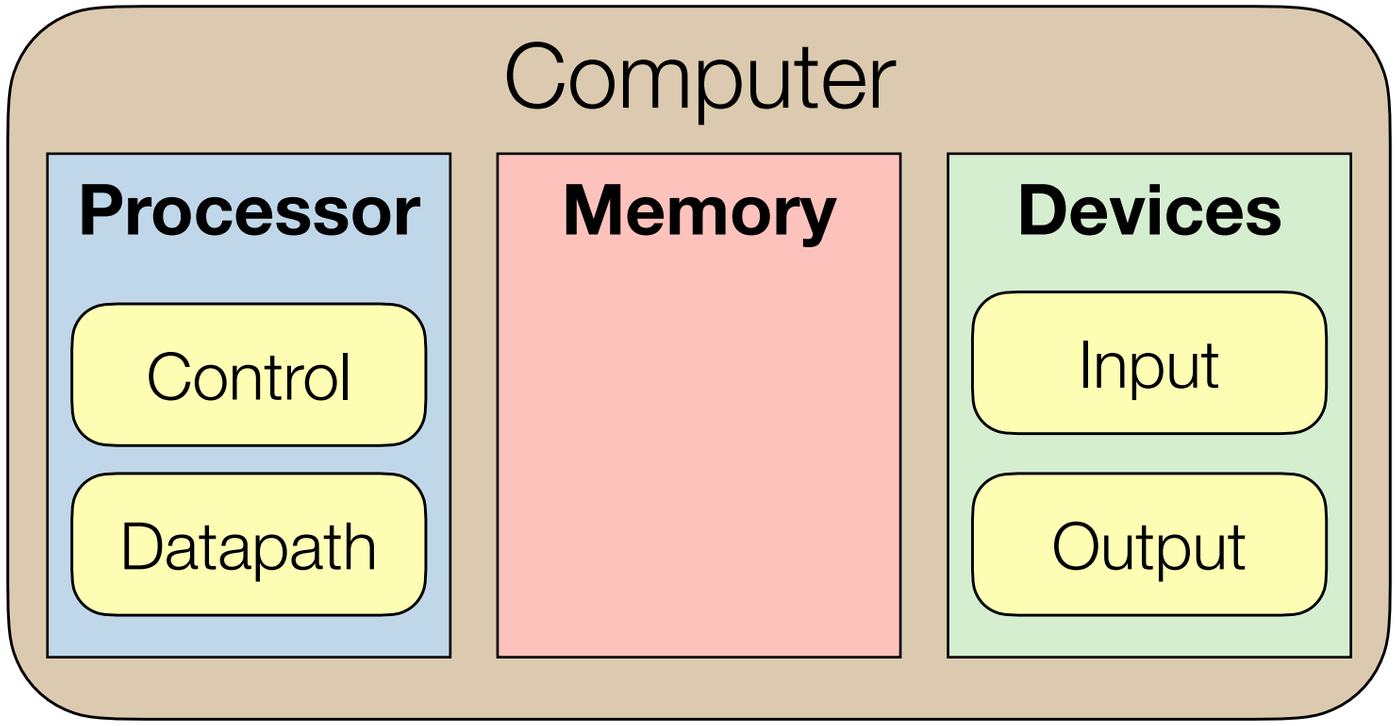
# Lecture 21: Virtual Memory

Spring 2024
Jason Tang

# Topics

- Virtual addressing
- Page tables
- Translation lookaside buffer

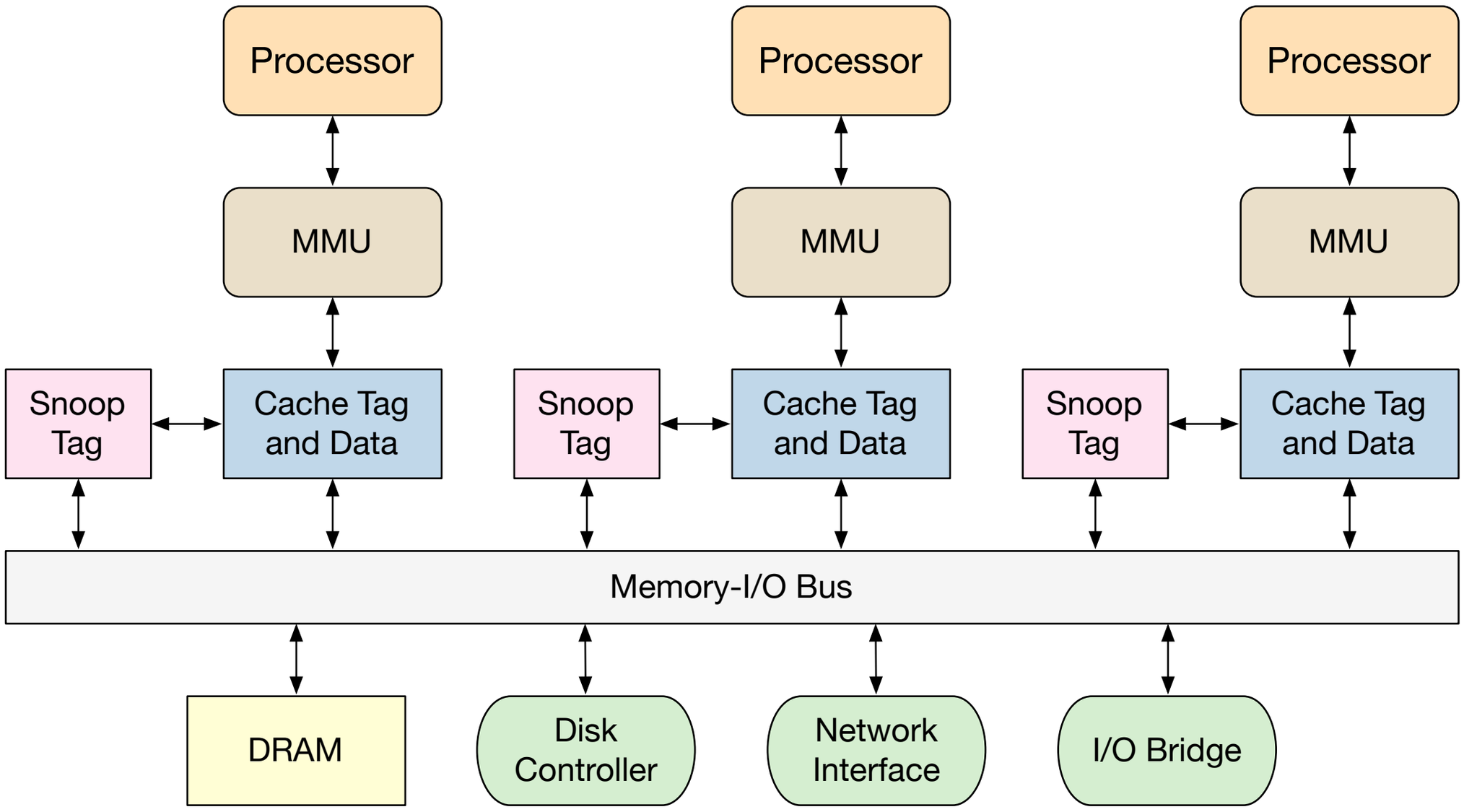
# Computer Organization




- Discussion thus far has concentrated upon the processor and then memory

- Will now start describing connection between processor and memory, processor and devices, and memory and devices

# System Interconnect



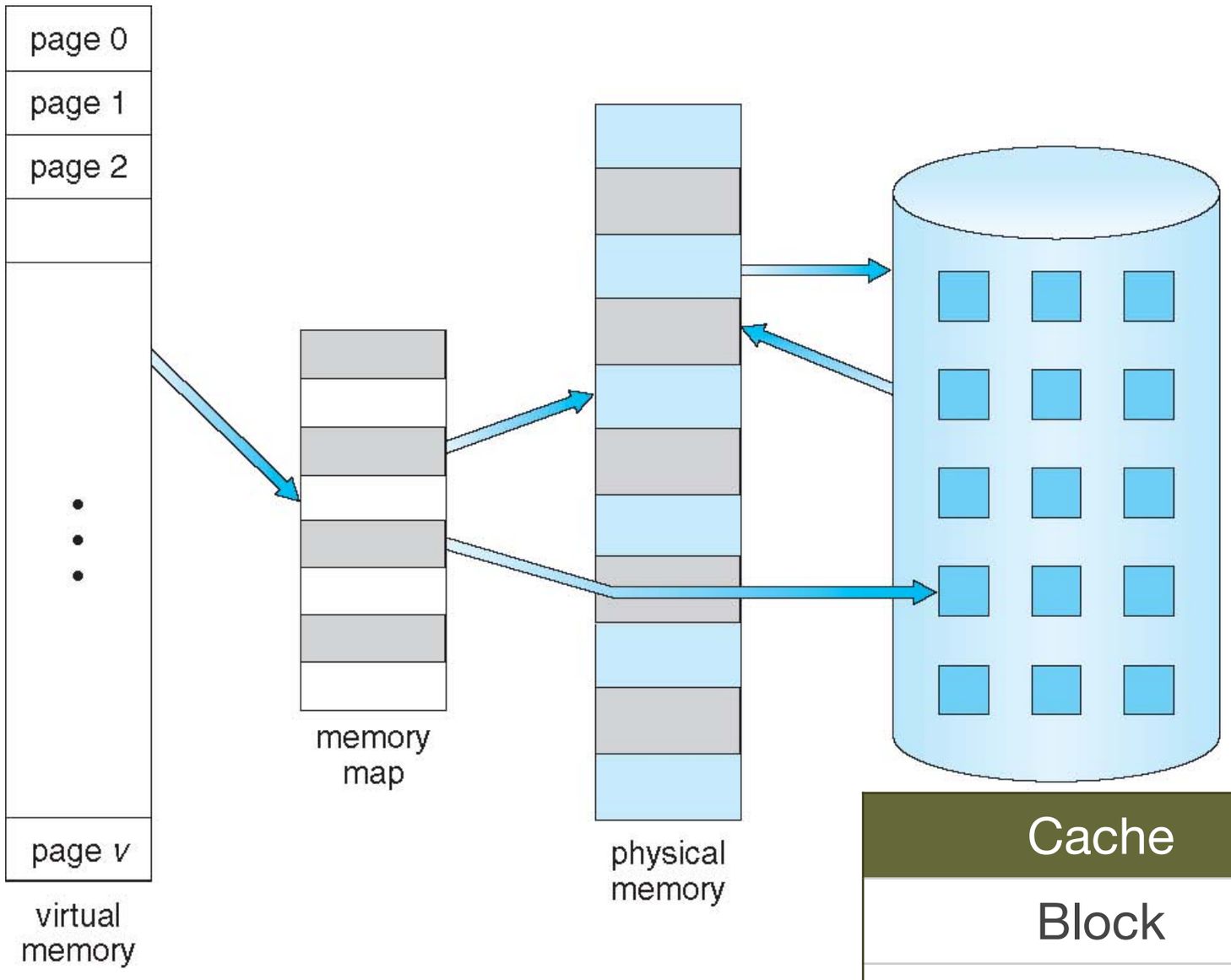
# Address Spaces

- Virtual Address (or logical address): address generated by CPU
- Physical Address: address calculated by memory management unit (MMU)
- DMA: direct memory access
- DMA Address (or bus address): addresses used by devices to send data between themselves
- DMA addresses do not always match physical address, and rarely match virtual address

# Virtual Address Space

- Separation of logical memory from physical memory, to allow for process isolation
  - Virtual address space size often differs than physical memory size
- From perspective of a running program on the processor, load and store instructions are performed relative to that program's virtual address space
  - MMU maps between virtual and physical addresses
  - Avoid need for contiguous memory blocks
  - Allows programs to be loaded at any physical memory location (relocation)

# Virtual Memory



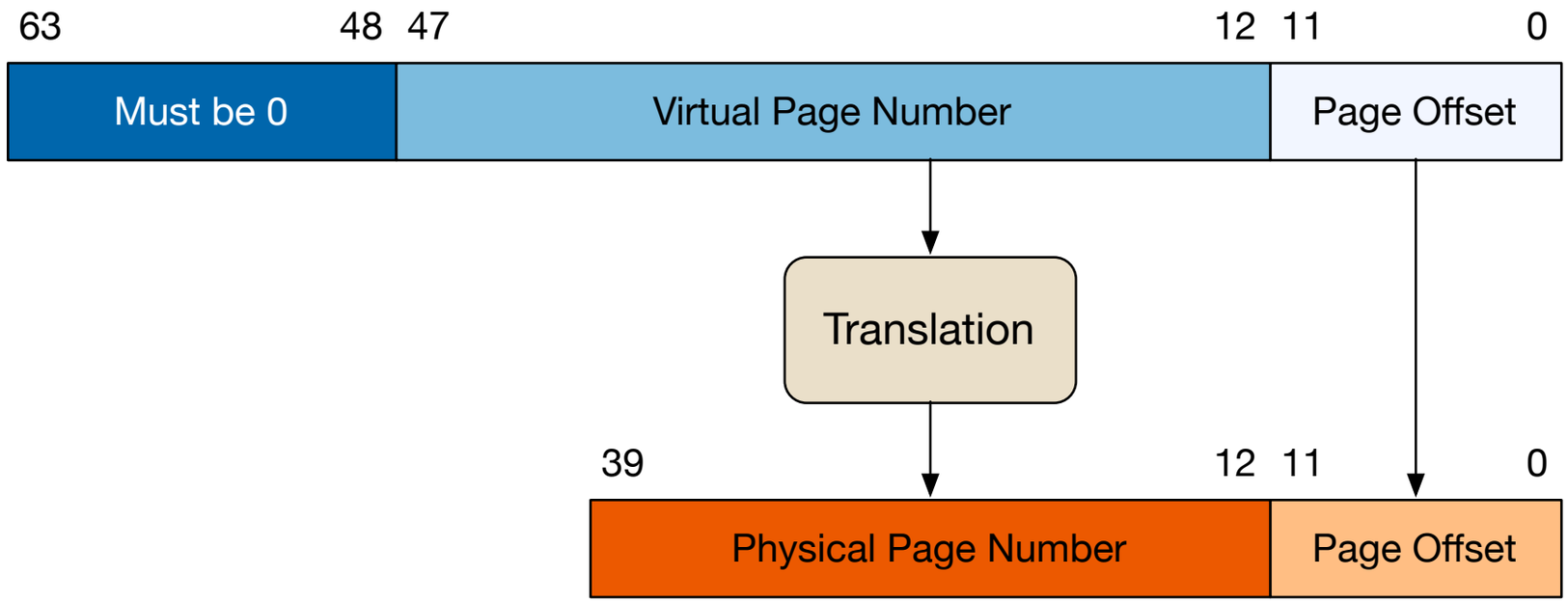

| Cache | Virtual Memory |
|---|---|
| Block | Page |
| Cache Miss | Page Fault |
| Block Addressing | Address Translation |

# Page Fault Handling

- Main memory can be considered as a cache of disk contents
- Upon a page fault, the system [sometimes] loads the missing data from disk (“swap in”)
  - Very costly, can take millions of cycles to process
- Pages should be large enough to reduce page fault rate
  - Modified data are cached as memory pages and then later flushed to disk (write back), as that constantly writing to disk is too slow (write through)
- When selecting a victim page to evict, LRU is most common algorithm

# Virtual Addressing

- Address is broken into a virtual page number and a page offset
- MMU translates virtual page number to a physical page number
- Example: ARMv8-A has 64-bit virtual addresses, but only lower 48 bits (256 TiB) are used. Suppose that physical address space is only 40 bits (1 TiB), and a page is 4 KiB. Then:



# Memory Maps

- In a set-associative cache, a particular memory block is mapped to some particular set

- With paging, the operating system maintains the memory mappings between virtual addresses and physical addresses

- Multiple virtual addresses map to the same physical address to support shared memory

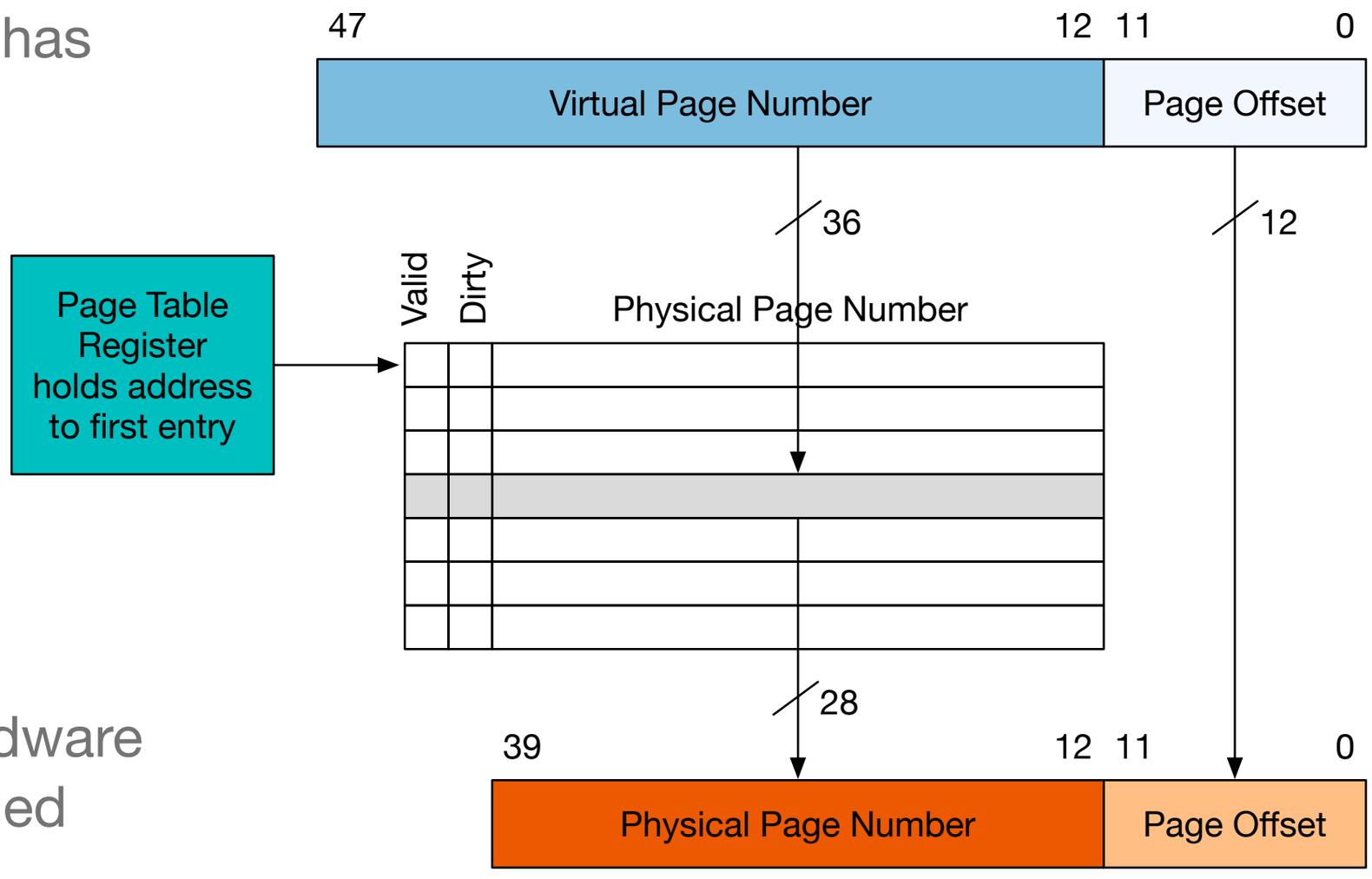
- Multiple physical addresses map to the same virtual address to support process forking (or after a copy-on-write page has been modified)

# Page Table

- [Usually] Stored in main memory
- Page Table Register holds the address to the beginning of the page table
- Operating system maintains a different page table for each process
- Each entry in table gives a mapping between from a virtual to a physical address
    - If a computed load/store virtual address does not have an entry within table, then MMU raises an interrupt (a page fault)

# Page Table

- Every page table entry has control flags:
  - Valid Bit: If set, then proceed with memory access; otherwise raise interrupt
  - Dirty Bit: Set by hardware when page is modified




- In simplest case, virtual page number is an index number into page table

# Page Table Size

- Many systems default to 4 KiB pages (12 bits)

- For a 32-bit virtual address, that means upper 20 bits give a virtual page number, and lower 12 bits are an offset into the resulting physical page

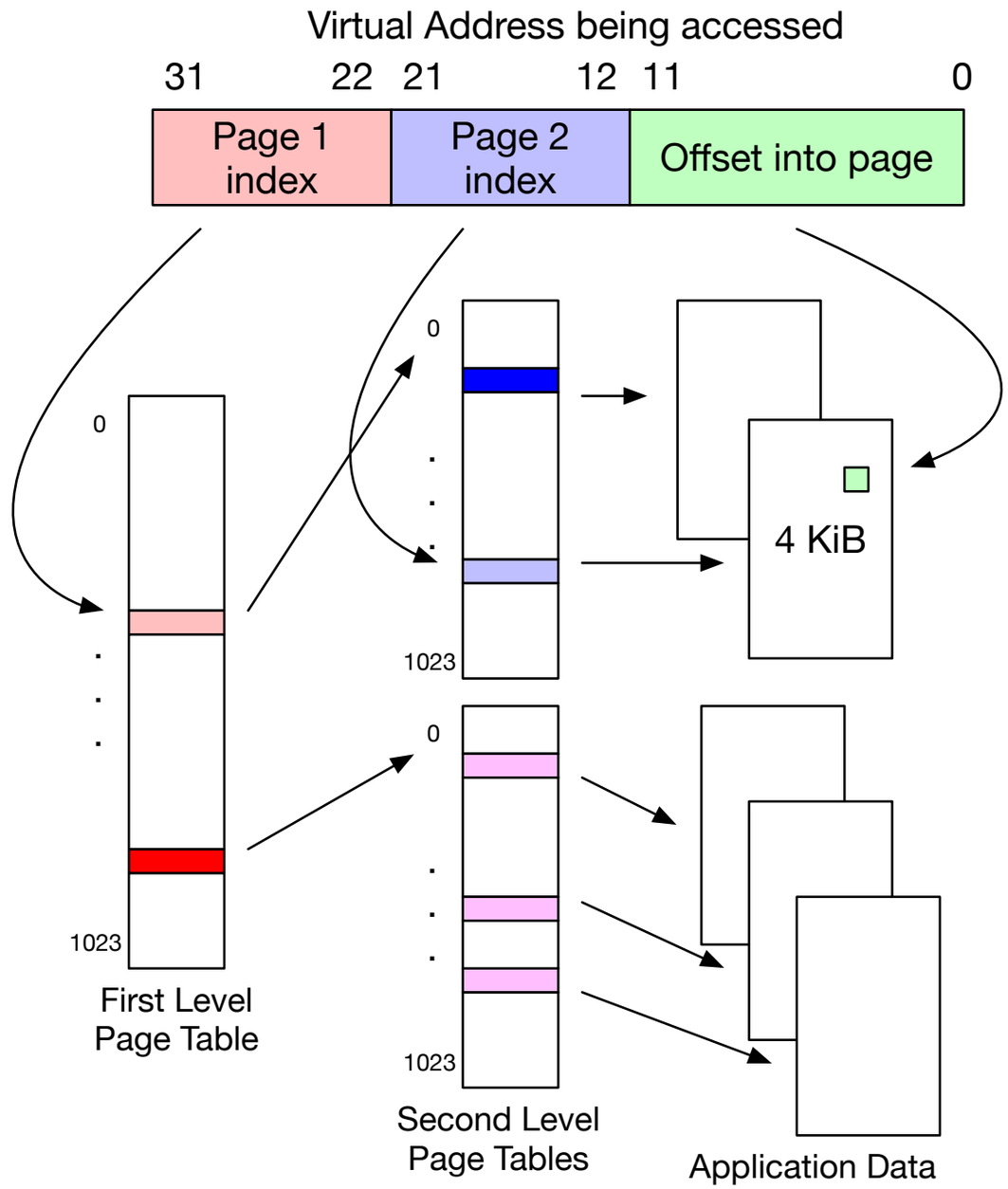
- To fully utilize entire 4 GiB address space (32 bits), number of page table entries needed is $\frac{2^{32}}{2^{12}} = 2^{20}$

- If each page table entry is 32 bits (4 bytes), then entire page table requires $2^{22}$ bytes = 4 MiB (which is greater than a page itself)

  - Need to either increase page sizes ("huge pages"), or use multi-level paging

# Multi-Level Page Table

- Instead of using virtual page number as an index into a single page table, split those bits to specify a first level and second level page
- Example: Let there be a system with 32-bit virtual addresses and 4 KiB pages. The first 10 bits give an index within a first level page table. That entry gives the physical address of a second level page table. Let the next 10 bits give an index within that second level page table. Use that entry as the physical address of the final page.



# ARMv8-A Multi-Level Page Table

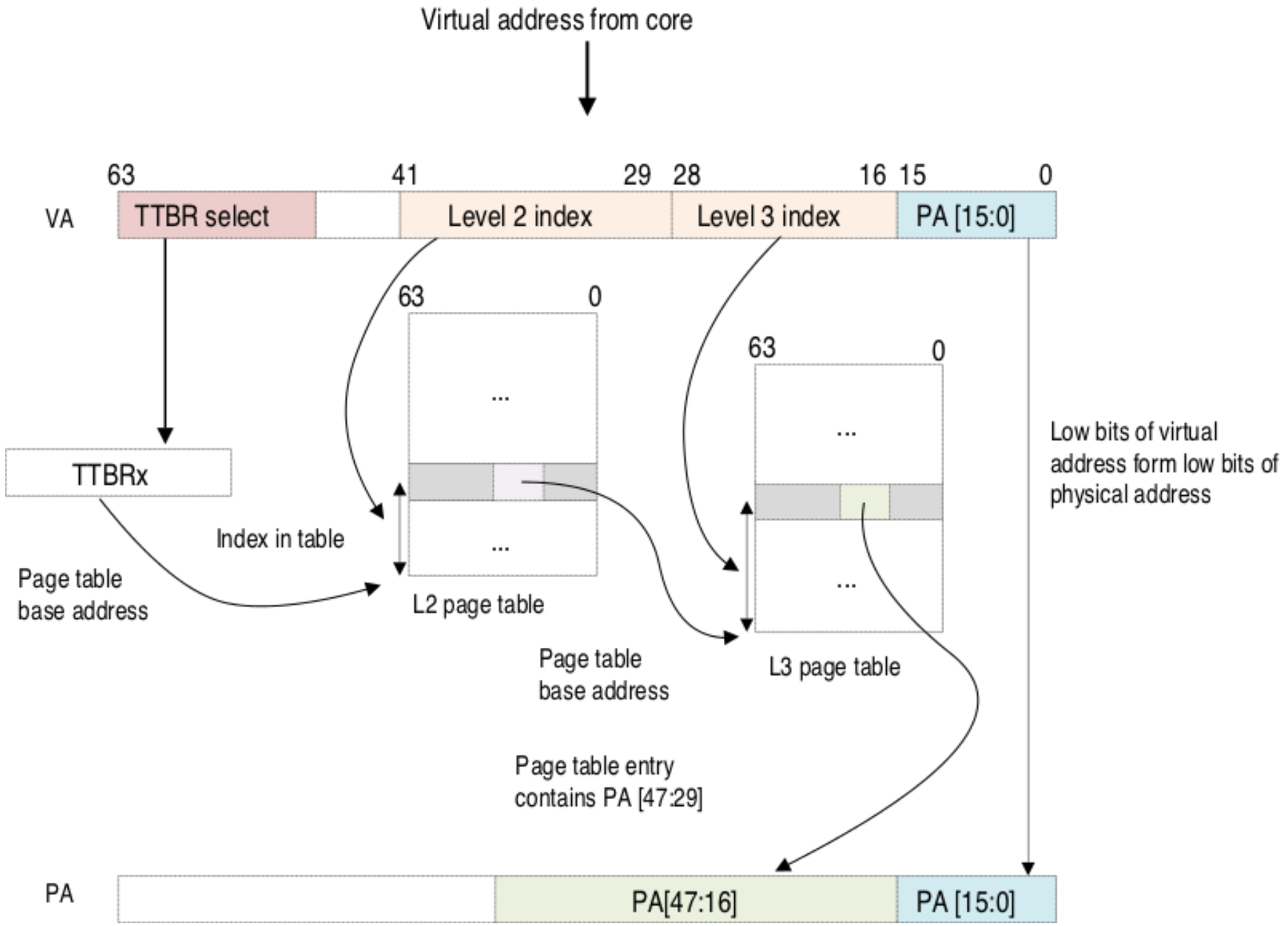
- ARMv8-A has 64-bit virtual addresses, and allows for various page sizes

  - For 4 KiB pages ("granules"), bits [47:39] specify one of 512 entries within the L0 table; each entry gives an address of a L1 table. Because each entry is 8 bytes, 512 × 8 = 4096 bytes, which is the page size. Bits [38:30] gives index within L1 table, bits [29:21] gives index with L2 table, bits [20:12] gives index within L3 table.

  - For 64 KiB pages, bits [47:42] are index into L1 table, bits [41:29] are index into L2 table, and then bits [28:16] are index into L3 table. Bits [15:0] are the page offset.

ARM Cortex-A Series Programmer's Guide, §12.4.2

# ARMv8-A Multi-Level Page Table

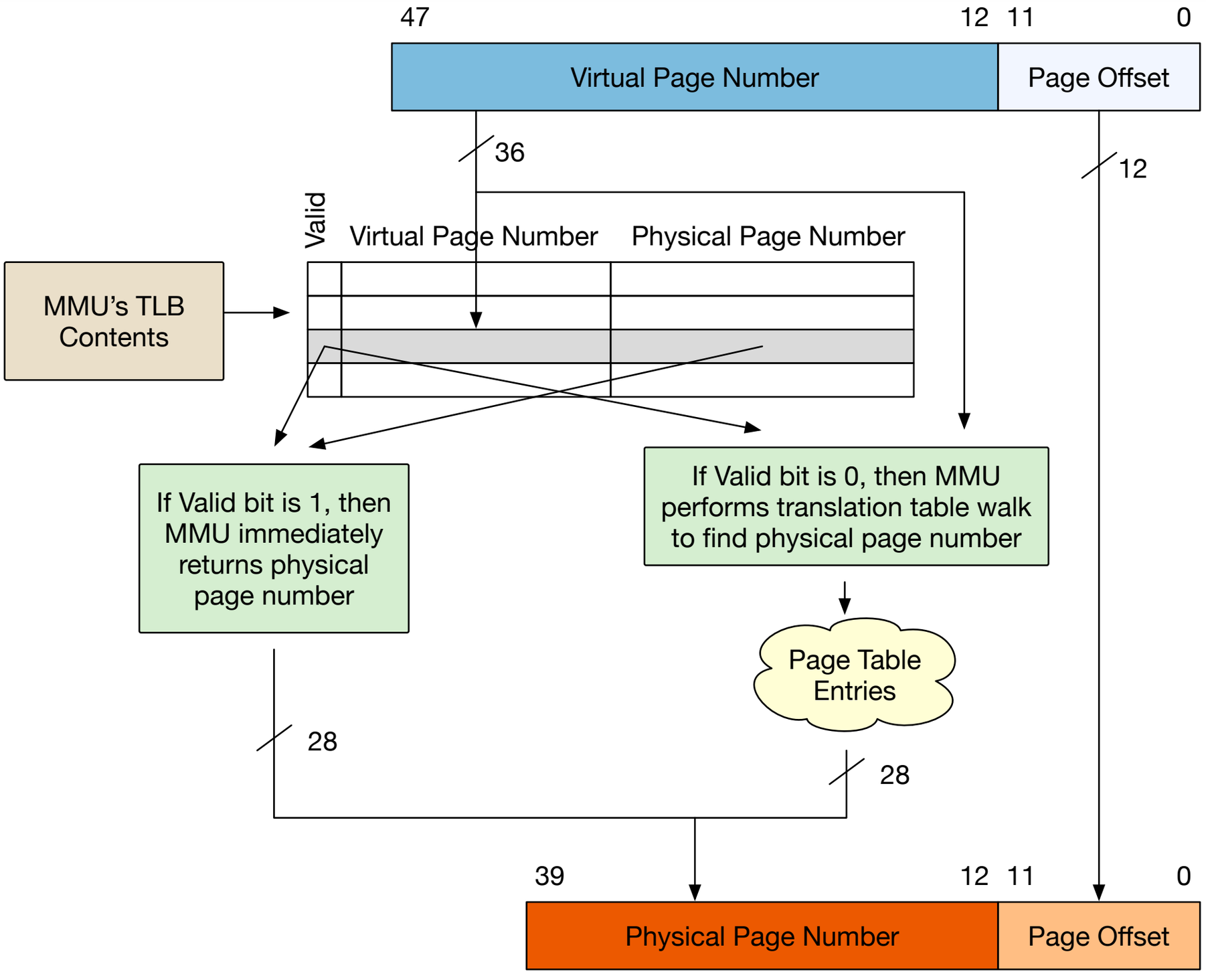



ARM Cortex-A Series Programmer's Guide, §12.3

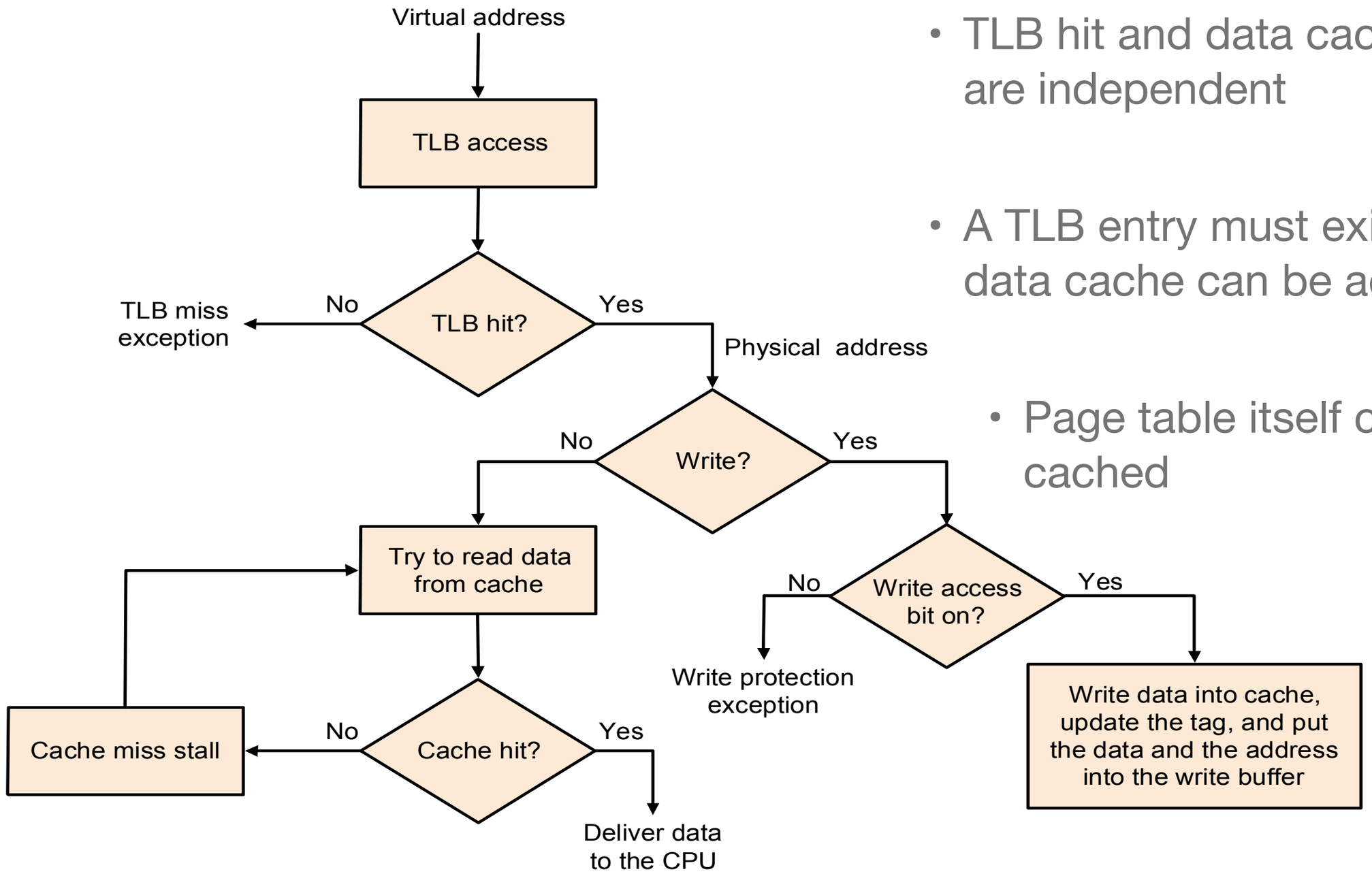
# TLB

- Each lookup into a page table incurs more memory overhead
- Translation Lookaside Buffer (TLB): cache within MMU that stores recent translation lookups
  - TLB Hit: Translation of a virtual address resolved by a cached TLB entry
  - TLB Miss: Incurs a translation table walk to resolve virtual address
- TLB is limited, so MMU replaces old entries (either automatically or via software) as part of TLB miss
  - Like normal data cache, TLB entries can be invalidated to force a refetch of the page table entries

# TLB Design


47
12
11
0
Virtual Page Number
Page Offset
36
12
Valid
Virtual Page Number
Physical Page Number
MMU's TLB Contents
If Valid bit is 1, then MMU immediately returns physical page number
If Valid bit is 0, then MMU performs translation table walk to find physical page number
Page Table Entries
28
28
39
12
11
0
Physical Page Number
Page Offset

# TLB and Cache




- TLB hit and data cache hit are independent
- A TLB entry must exist before data cache can be accessed
- Page table itself can be cached

# Memory Exceptions Summary

- Cache miss: referenced block not in cache; needs to be fetch from main memory
- TLB miss: referenced page of virtual address needs to be check in page table
- Page fault: reference page not in main memory (may need to load from disk)

| Cache | TLB | Page | Condition |
|---|---|---|---|
| miss | hit | hit | Possible, but page table is unchecked if TLB hits |
| hit | miss | hit | TLB miss, but entry found in page table and data in cache |
| miss | miss | hit | TLB miss, but entry found in page table and date miss in cache |
| miss | miss | miss | TLB miss, followed by page fault; data missed in cache |
| miss | hit | miss | Impossible - no translation possible if page not in memory |
| hit | hit | miss | Impossible - no translation possible if page not in memory |
| hit | miss | miss | Impossible - data cannot be cached if page is not in memory |

# Memory Protection Bits

- Page table entries support protection bits in addition to valid and dirty bits:
  - Read-only (*RO*) or read-write (*RW*)
  - Executable (*X*) or non-executable (*NX*)
  - User mode or privileged mode

| Memory Region | Protection |
|---|---|
| Stack | RW, NX |
| Heap | RW, NX |
| Text | RO, X |
| Constants | RO, NX |

- Operating system uses protection bits to enforce separation of data between system and user processes

# Handling TLB Misses and Page Faults

- TLB Miss: handled by hardware
  - Check if page is in memory; if is valid then update TLB
  - Raise page fault interrupt if page is not in memory
- Page Fault: handled by operating system
  - OS interrupt handler determines cause of page fault
  - If OS needs to load referenced data from disk, it will restart instruction after disk I/O completes; until then OS keeps the process suspended