CSC 252/452: Computer Organization
Fall 2024: Review Lecture

Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Class Review — Part 1

- Data representation
  - Binary/Bits
  - Integer
  - Floating Point
- Assembly programing
  - ISA basics
  - Assembly basics
    - Memory, architectural states, pointers
  - Instructions (mov, arithmetic, jump)
  - Condition codes
- Functions
  - Stack & frame
- Data structures and buffer overflow
- Instruction Encoding
  - How does the assembler work?

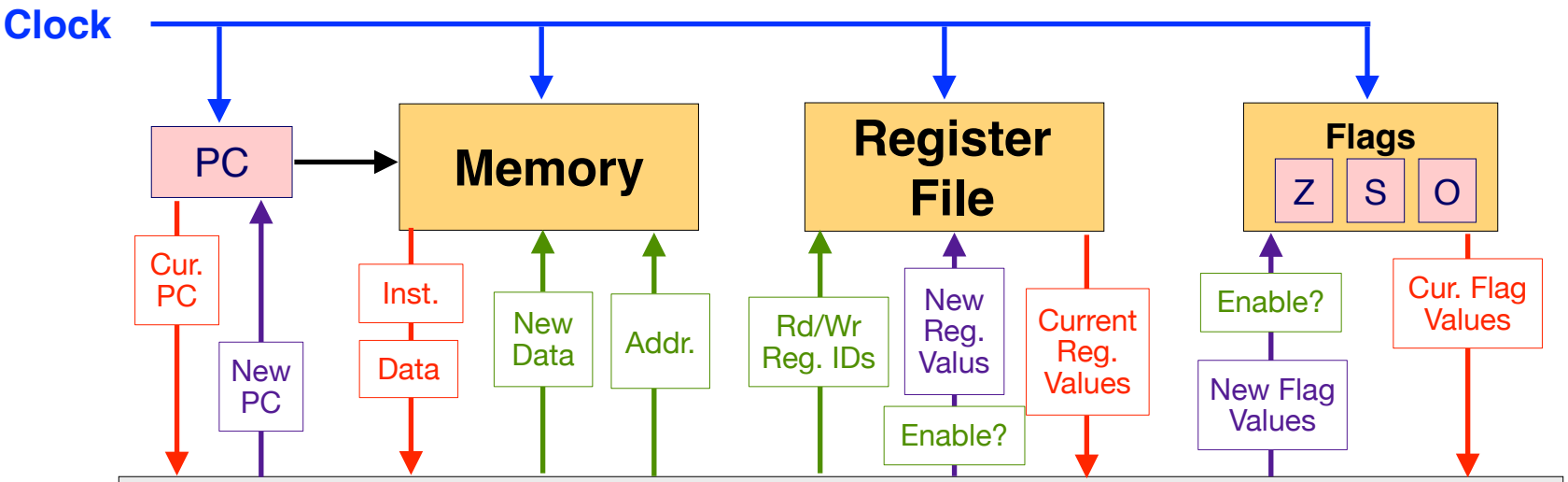
Class Review —Part 2

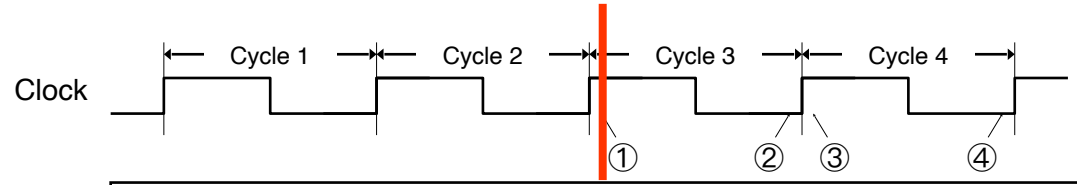
- CPU design
  - Single-cycle CPU
  - Pipelining
- Memory system
  - Memory technologies
  - Trade-offs
  - Memory hierarchy
  - Cache
    - Set-associative and direct-mapped cache
    - Replacement policy
- Processes
- Signals (not in final)
- Interrupts and exceptions (not in final)
- Virtual memory
  - Page table
  - TLB
- Dynamic memory allocation
  - Implicit allocator and explicit allocator
- Multi-threading

CPU Design

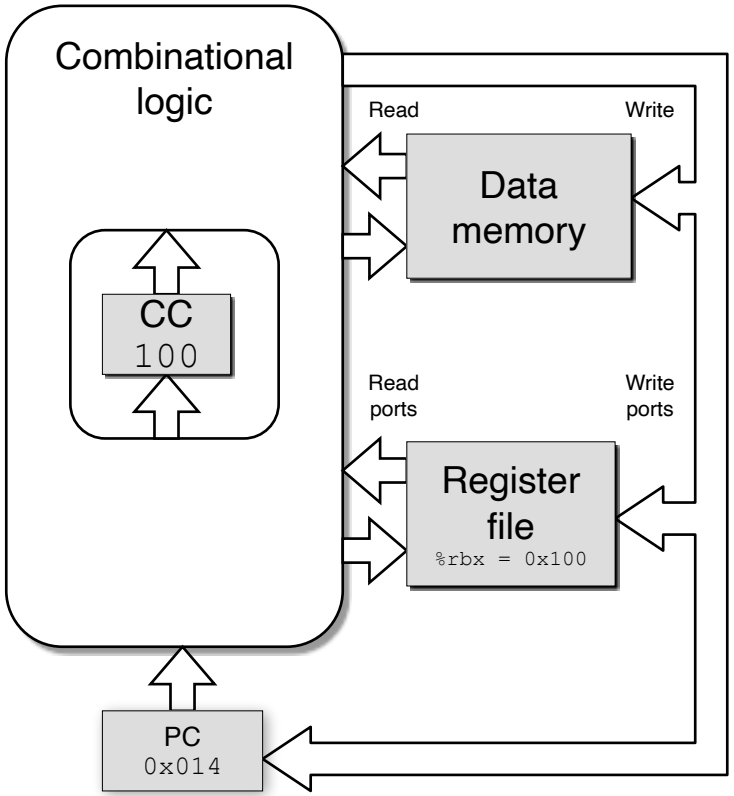
- Single-cycle CPU
  - How it works?
  - The drawback of it?
- Pipelining
  - Basic ideas
  - The advantage of it
  - Throughput and unbalanced pipeline
  - Problems to solve
    - Control dependency
    - Data dependency

Single-Cycle Microarchitecture

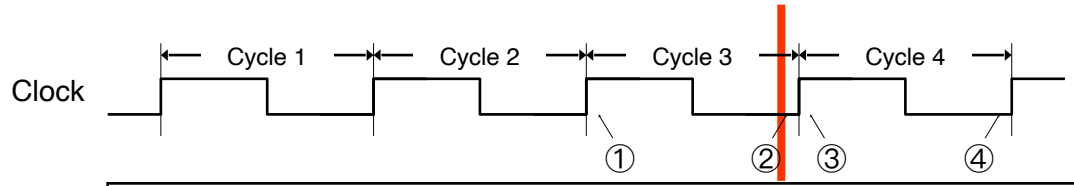




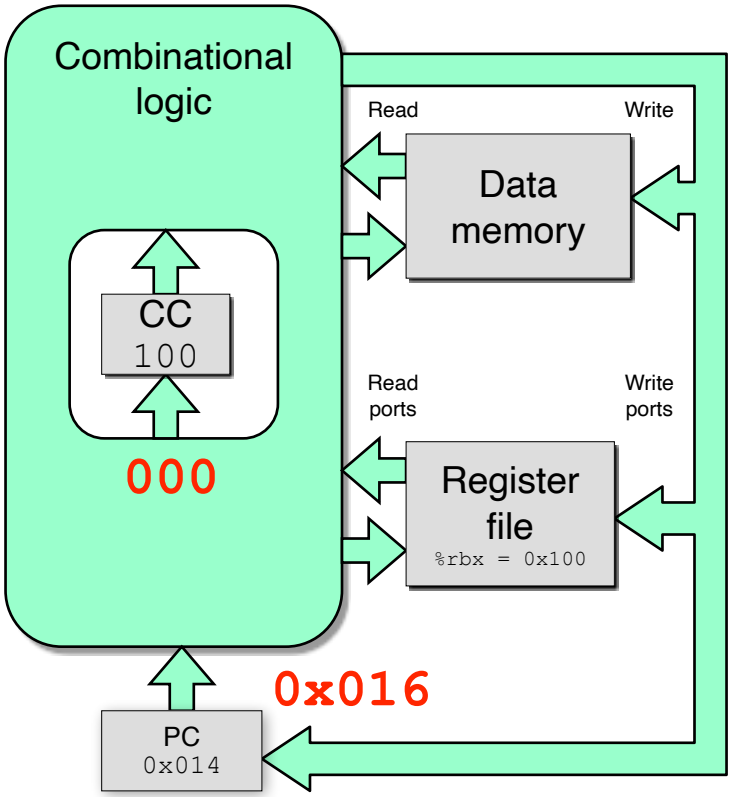
Cycle 1:	`0x000: irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a: irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014: addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016: je dest`	`# Not taken`
Cycle 5:	`0x01f: rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

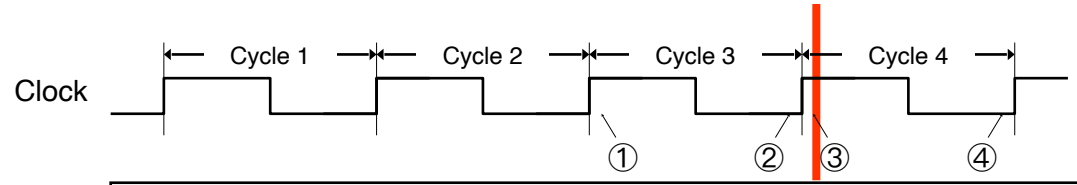


Cycle 1:	`0x000: irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a: irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014: addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016: je dest`	`# Not taken`
Cycle 5:	`0x01f: rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`

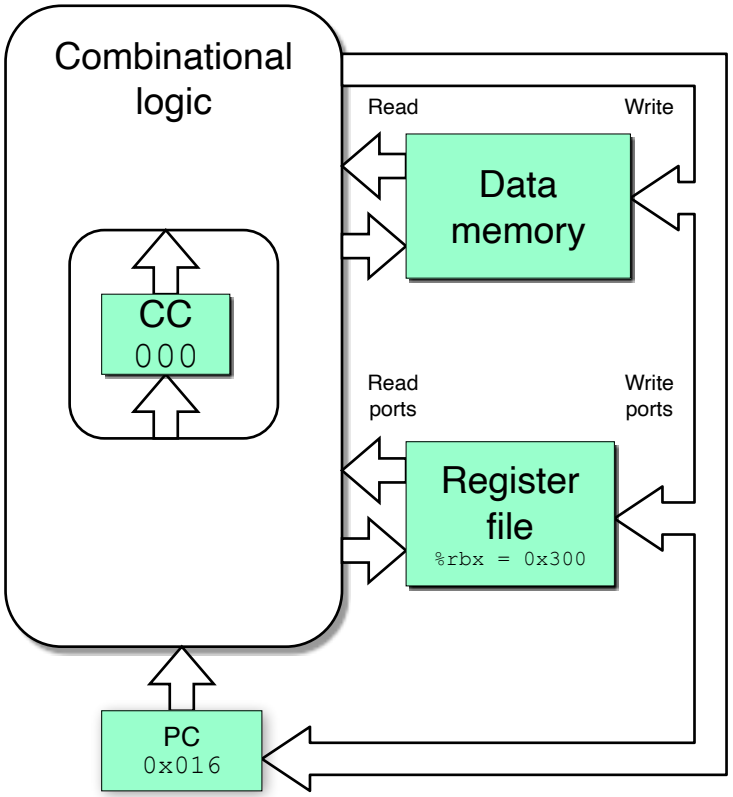


%rbx <-- 0x300

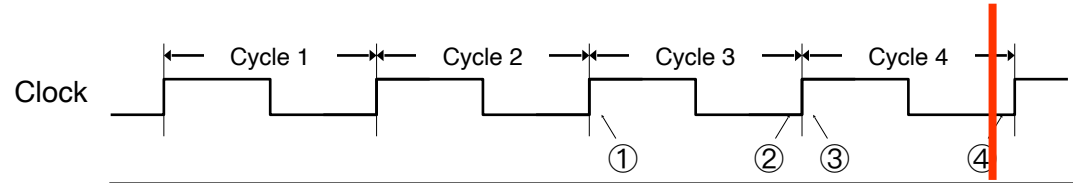
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



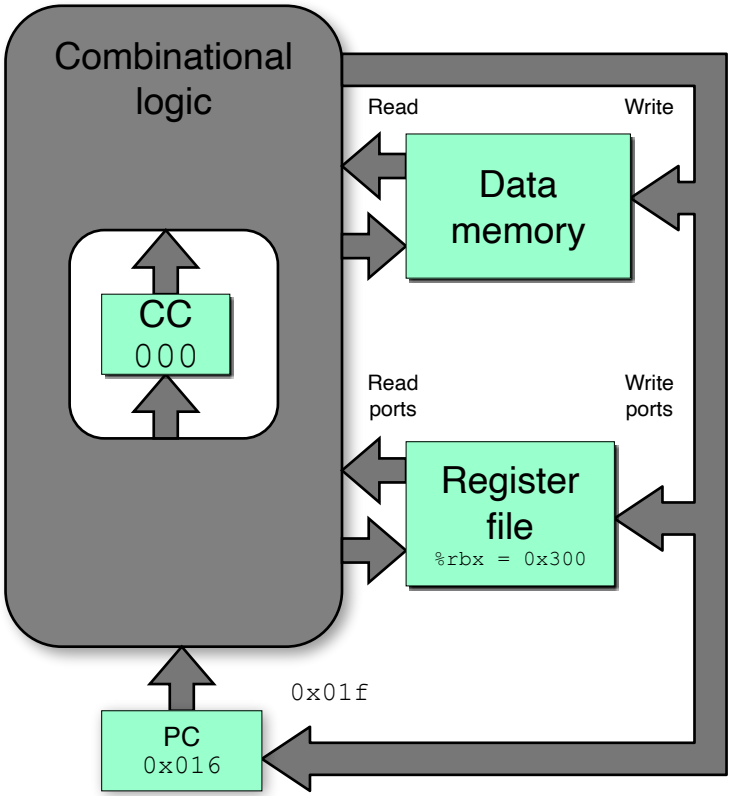
Cycle 1:	`0x000: irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a: irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014: addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016: je dest`	`# Not taken`
Cycle 5:	`0x01f: rmmovq %rbx,0(%rdx)`	`# M[0x200] <-- 0x300`



- state set according to `addq` instruction
- combinational logic starting to react to state changes



Cycle 1:	`0x000:`	`irmovq $0x100,%rbx  # %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx  # %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx  # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest  # Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

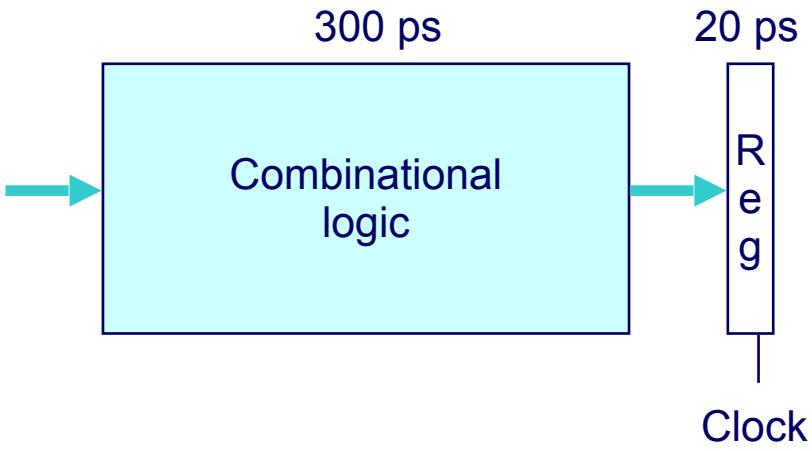


- state set according to `addq` instruction
- combinational logic generates results for `je` instruction

Limitations of a Single-Cycle CPU

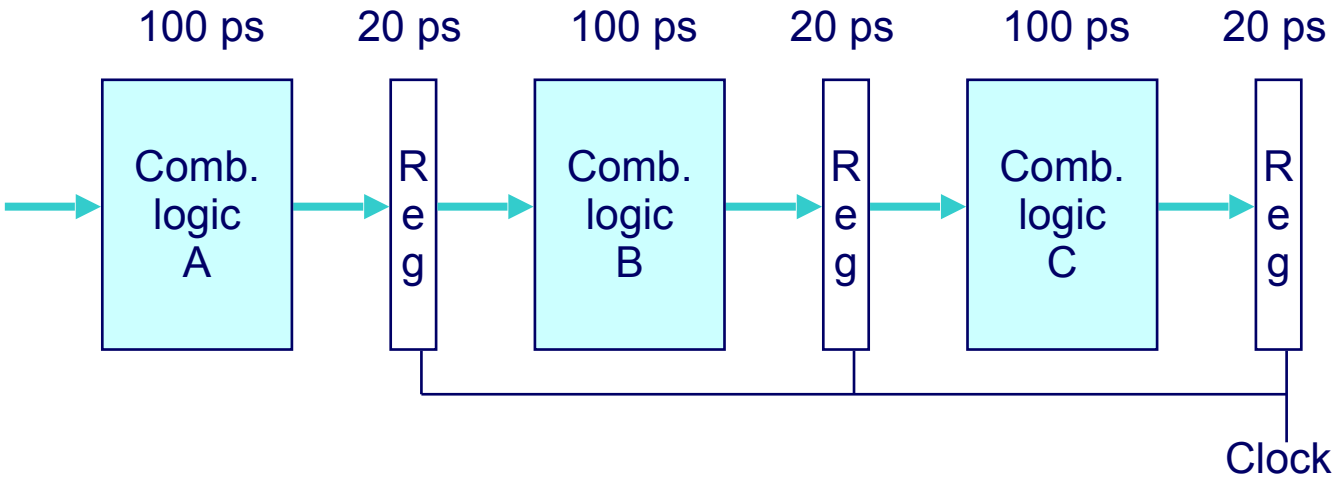
- Cycle time
  - Every instruction finishes in one cycle.
  - The absolute time takes to execute each instruction varies. Consider for instance an ADD instruction and a JMP instruction.
  - But the cycle time is uniform across instructions, so the cycle time needs to accommodate the worst case, i.e., the slowest instruction.
  - How do we shorten the cycle time (increase the frequency)?
- CPI
  - The entire hardware is occupied to execute one instruction at a time. Can’t execute multiple instructions at the same time.
  - How do execute multiple instructions in one cycle?

Pipelining



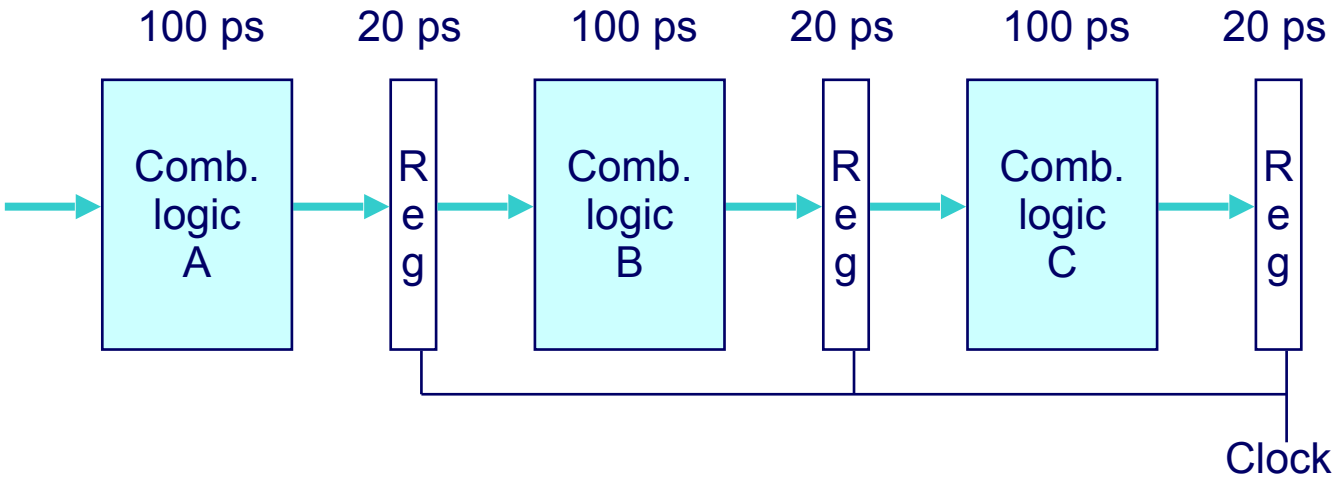
- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

3-Stage Pipelined Version

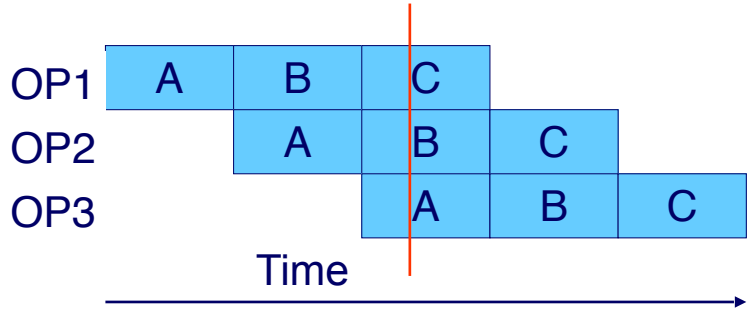


- Divide combinational logic into 3 stages of 100 ps each
- Insert registers between stages to store intermediate data between stages. These are call pipeline registers (ISA-invisible)
- Can begin a new instruction as soon as the previous one finishes stage A and has stored the intermediate data.
  - Begin new operation every **120 ps**
  - **Cycle time can be reduced to 120 ps**

3-Stage Pipelined Version



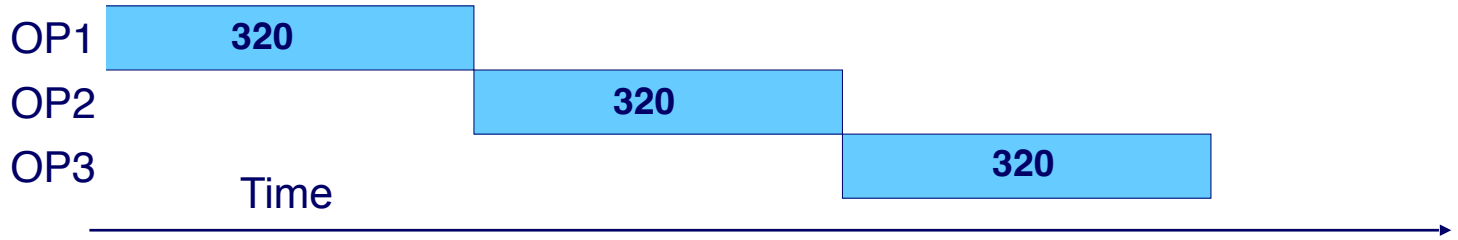
3-Stage Pipelined



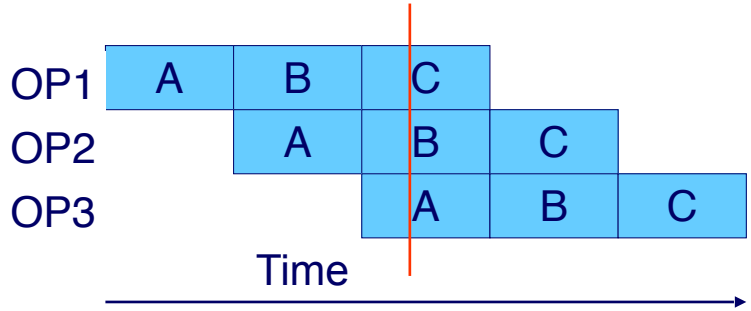
Comparison

Unpipelined

- Time to finish 3 insts = 960 ps
- Each inst.'s latency is 320 ps



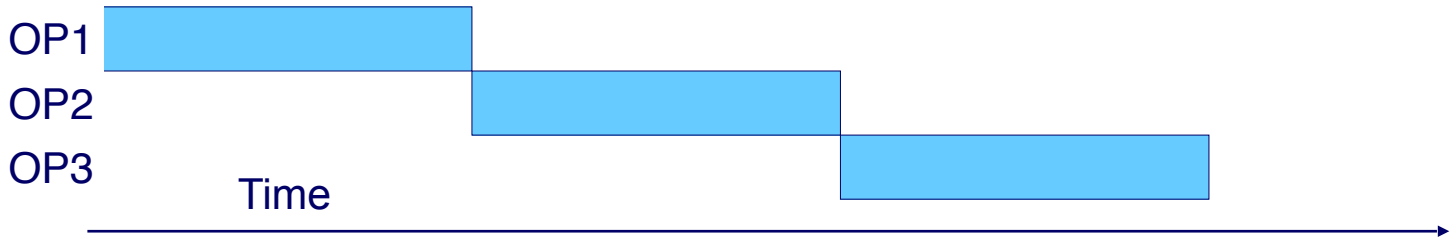
3-Stage Pipelined



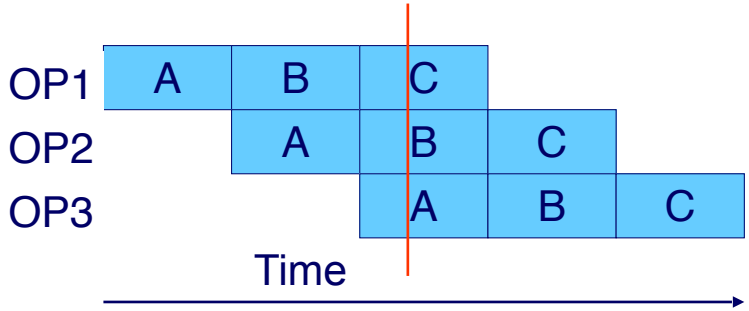
- Time to finish 3 insets = 120 * 5 = 600 ps
- But each inst.'s latency increases: 120 * 3 = 360 ps

Benefits of Pipelining

- Time to finish 3 insts = 960 ps
- Each inst.'s latency is 320 ps



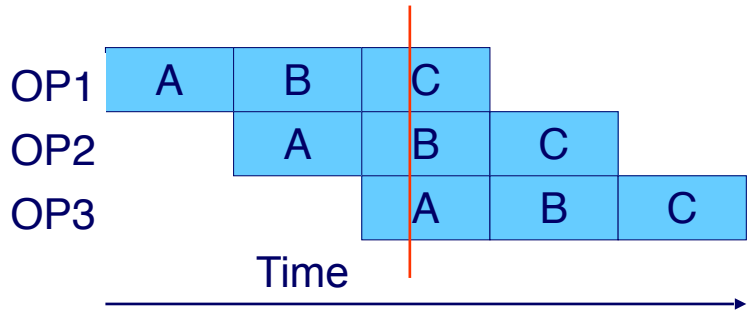
Reduce the cycle time from 320 ps to 120 ps



- Time to finish 3 insets = 120 * 5 = 600 ps
- But each inst.'s latency increases: 120 * 3 = 360 ps

One Requirement of Pipelining

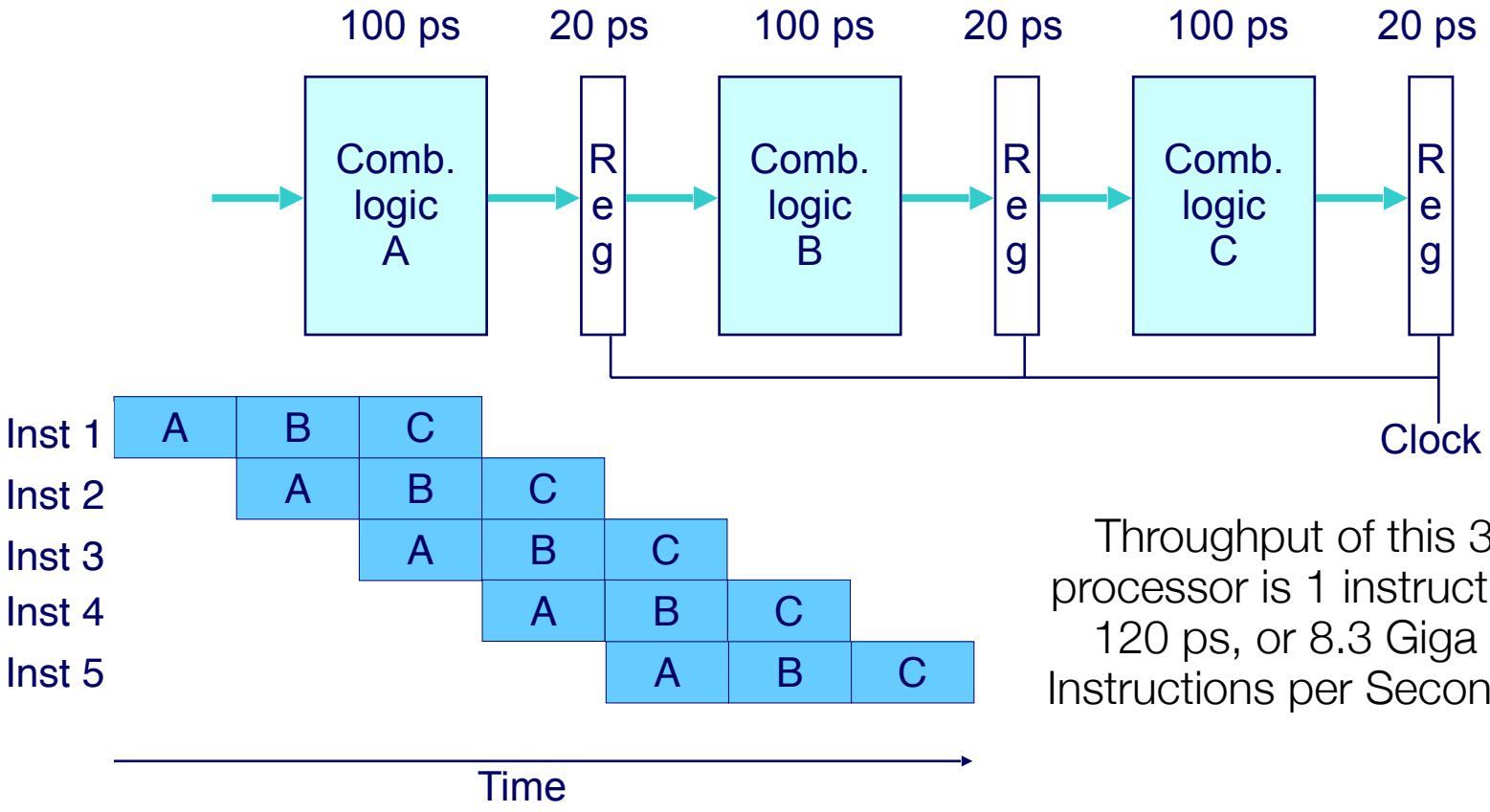
- The stages need to be using different hardware structures.
- That is, Stage A, Stage B, and Stage C need to exercise different parts of the combination logic.



- Time to finish 3 insets = 120 * 5 = 600 ps
- But each inst.'s latency increases: 120 * 3 = 360 ps

Throughput

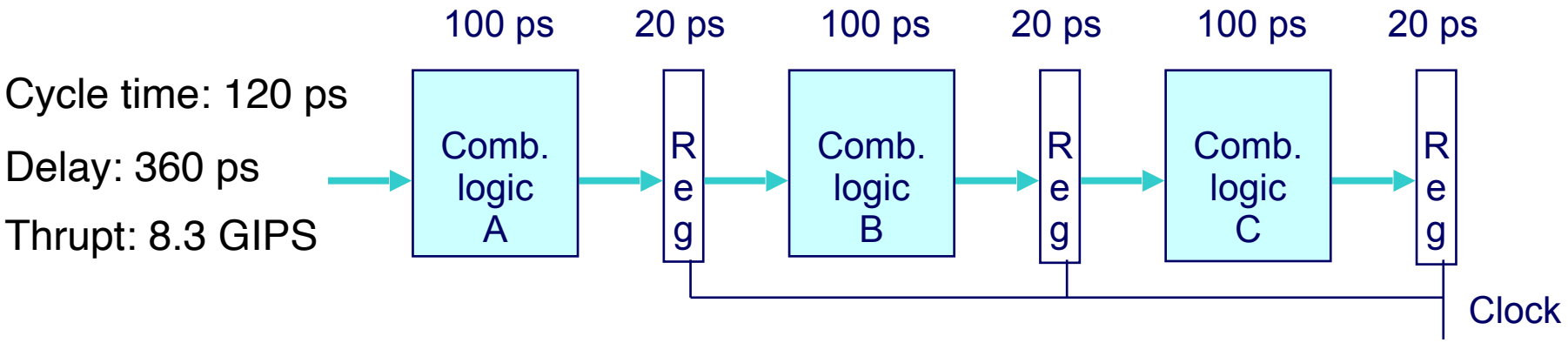
- The rate at which the processor can finish executing an instruction (at the steady state).



Throughput of this 3-stage processor is 1 instruction every 120 ps, or 8.3 Giga (billion) Instructions per Second (GIPS).

Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput



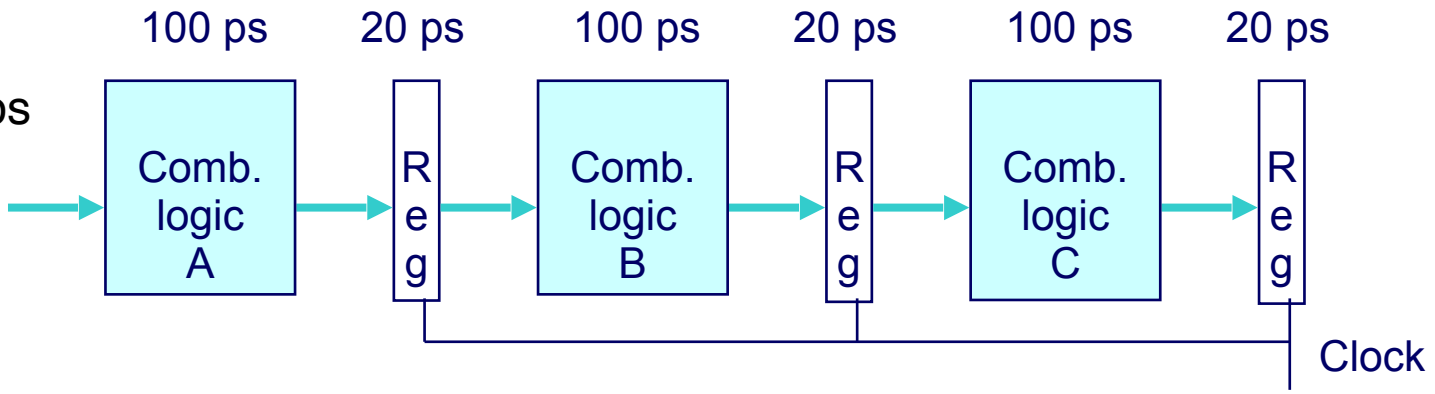
Unbalanced Pipeline

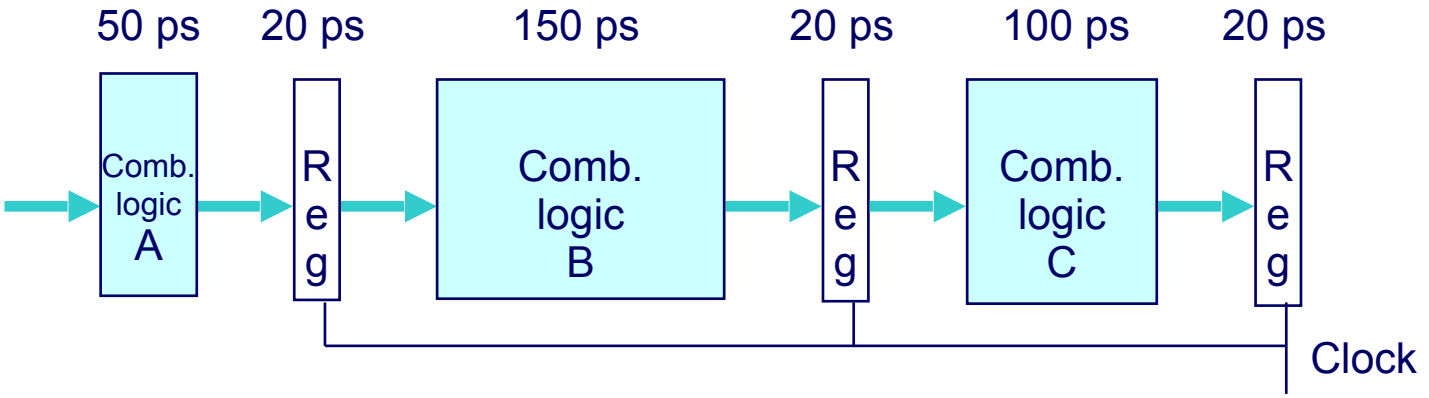
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

Cycle time: 120 ps

Delay: 360 ps

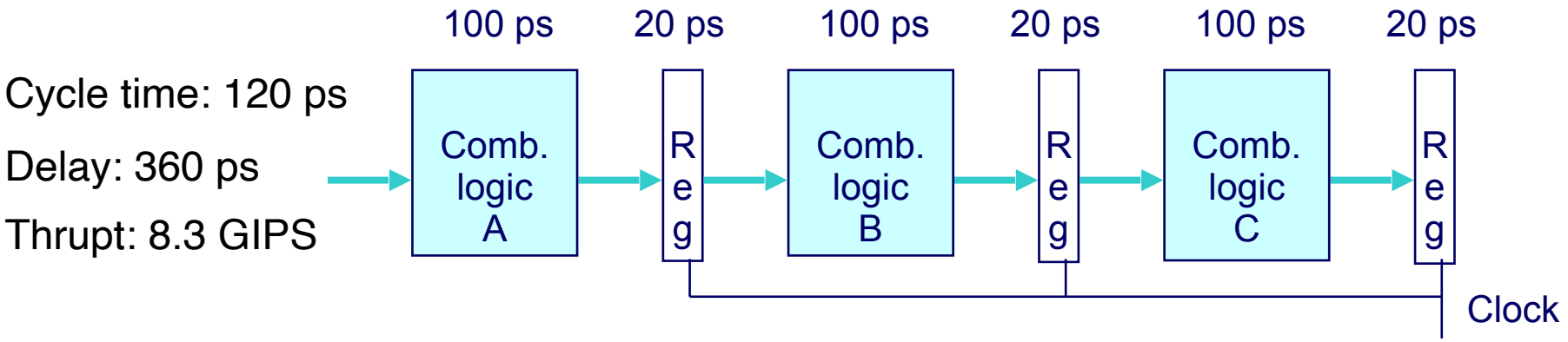
Thrupt: 8.3 GIPS

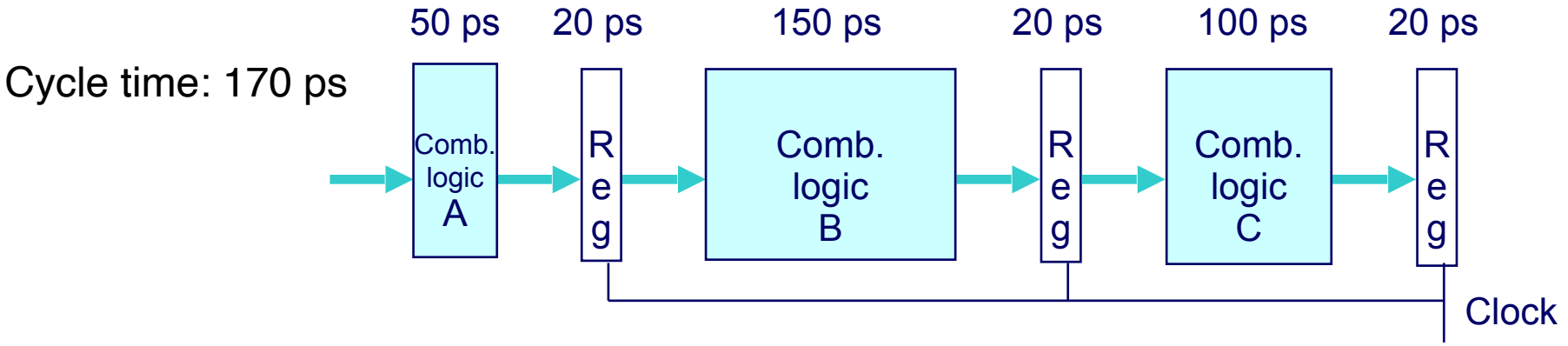




Unbalanced Pipeline

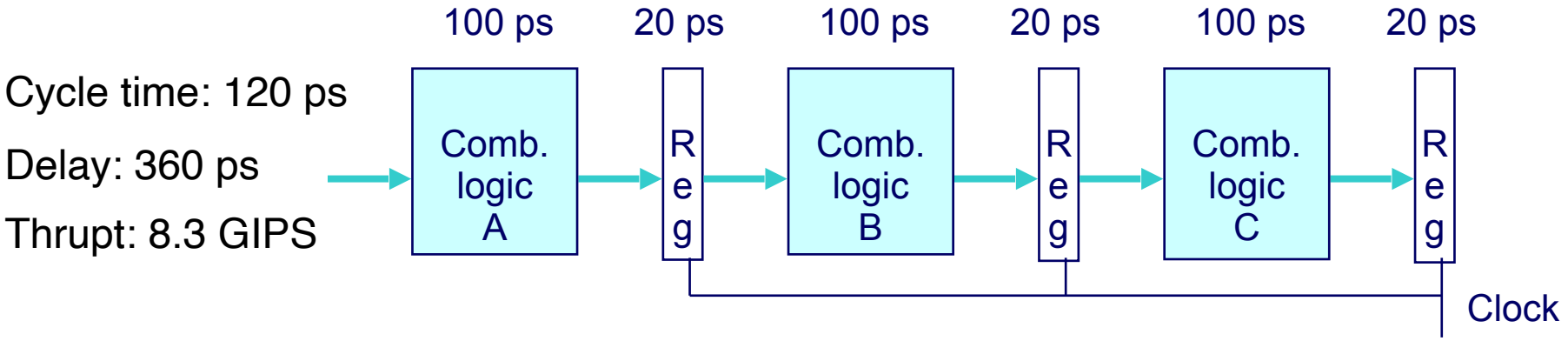
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

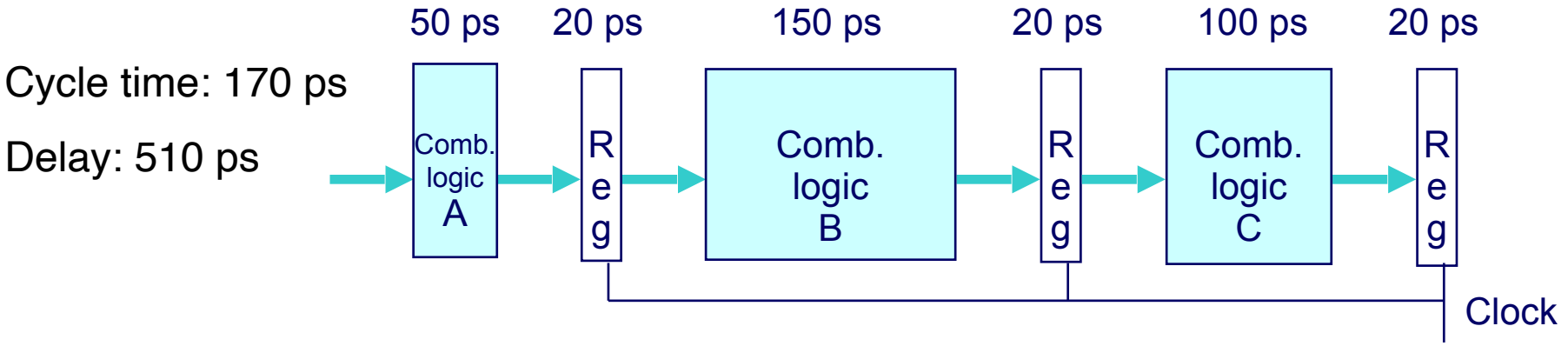




Unbalanced Pipeline

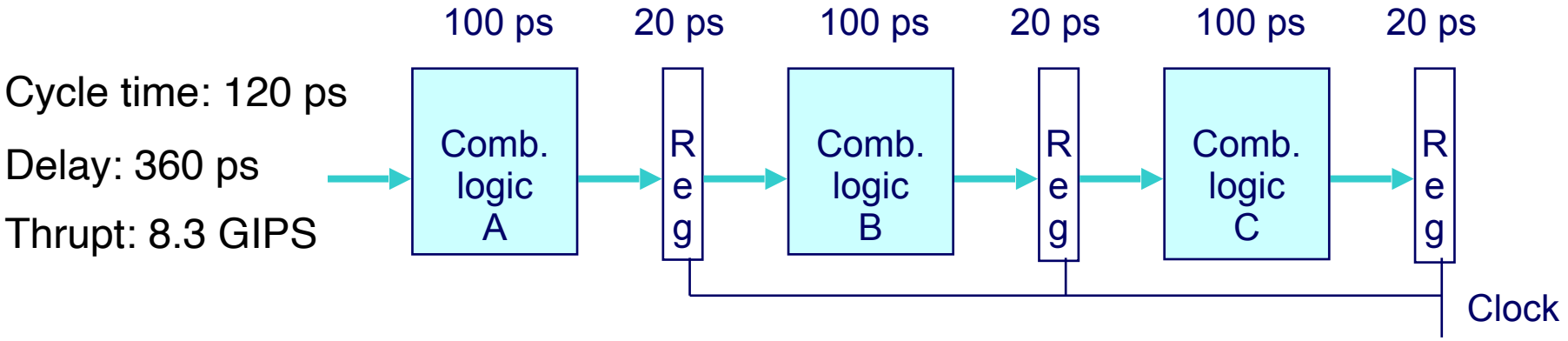
- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput

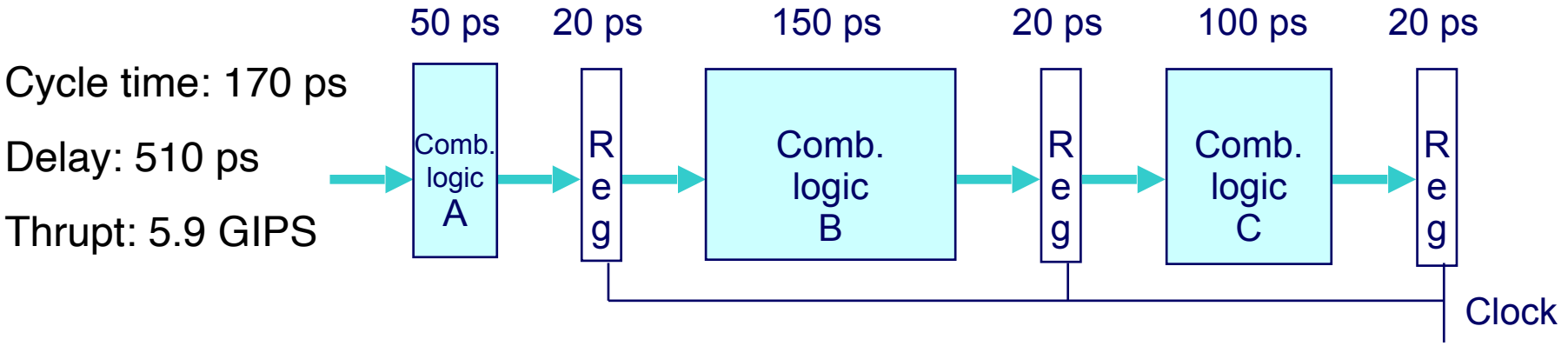




Unbalanced Pipeline

- A pipeline's delay is limited by the slowest stage. This limits the cycle time and the throughput





Pipeline Stages

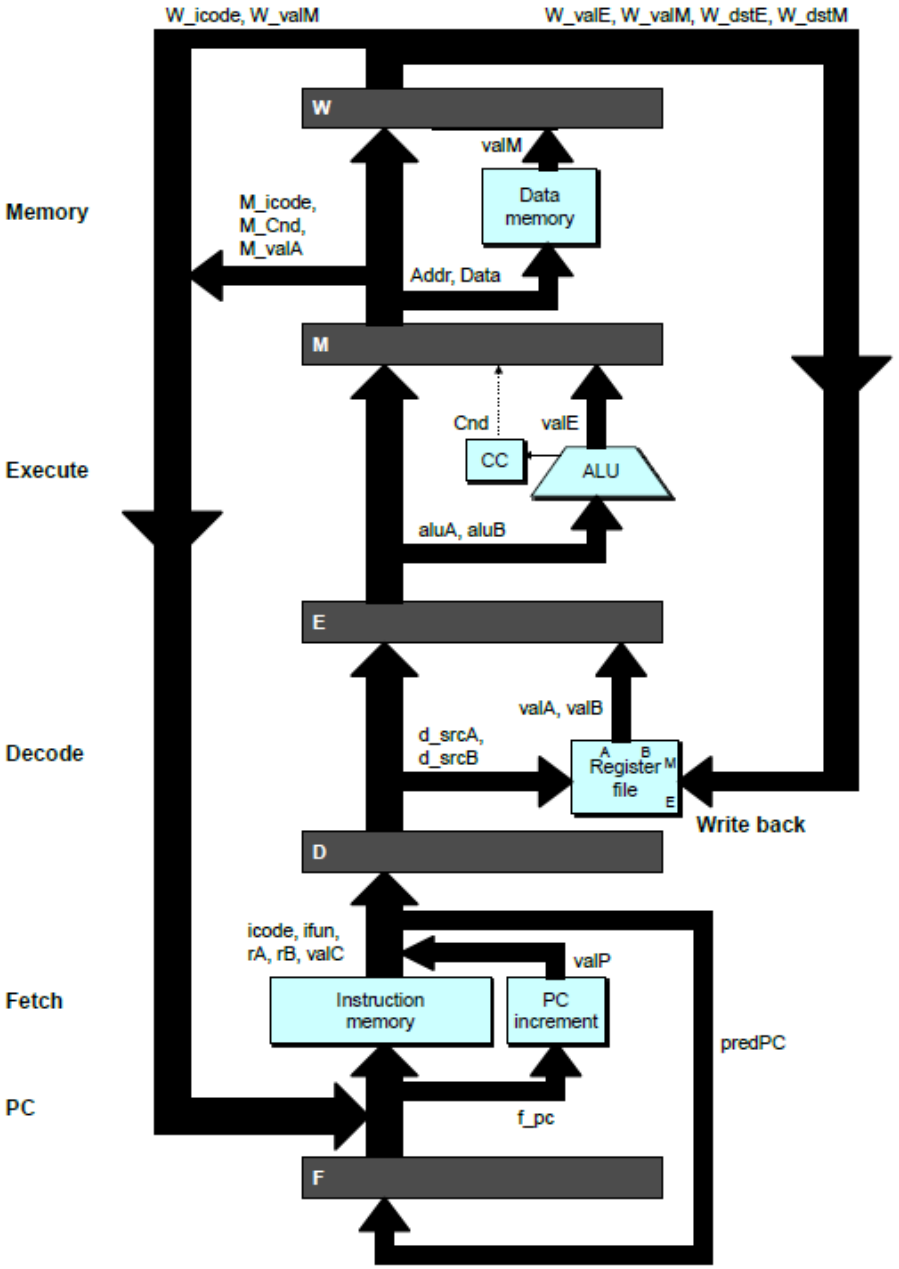
Fetch
- Select current PC
- Read instruction
- Compute incremented PC

Decode
- Read program registers

Execute
- Operate ALU

Memory
- Read or write data memory

Write Back
- Update register file



Stage Computation: Arith/Log. Ops

	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2 PC ← valP	Read instruction byte Read register byte Compute next PC PC ← valP
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result

Making the Pipeline Really Work

- Control Dependencies
  - What is it?
  - Software mitigation: Inserting Nops
  - Software mitigation: Delay Slots
- Data Dependencies
  - What is it?
  - Software mitigation: Inserting Nops

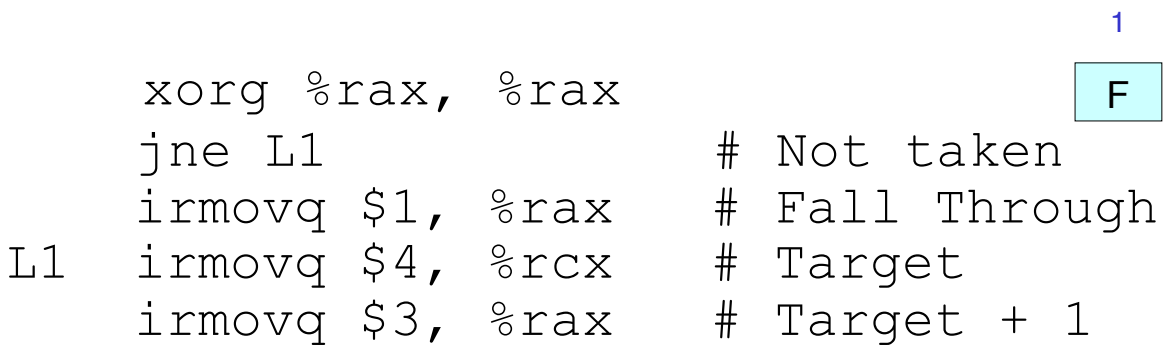
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
    xorg %rax, %rax
    jne L1              # Not taken
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```

Control Dependency

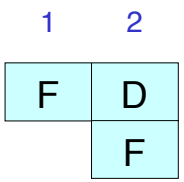
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

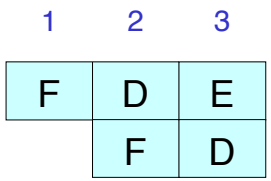
```
     xorg %rax, %rax
     jne L1              # Not taken
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

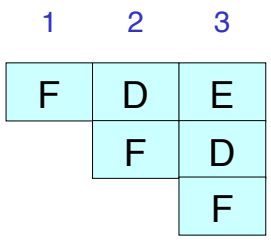
```
    xorg %rax, %rax
    jne L1              # Not taken
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

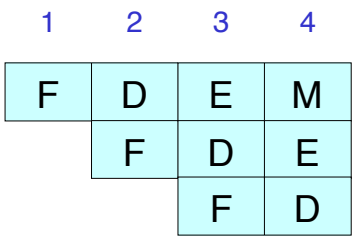
```
     xorg %rax, %rax
     jne L1              # Not taken
     nop
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

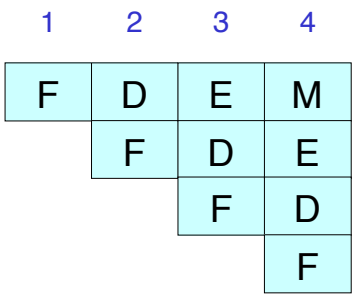
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

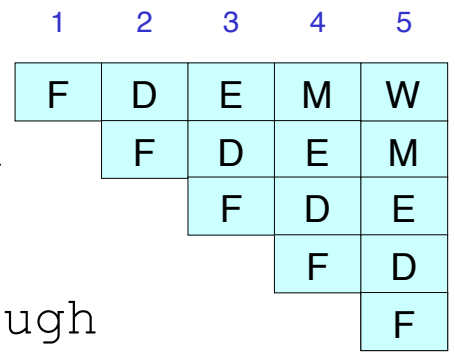
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

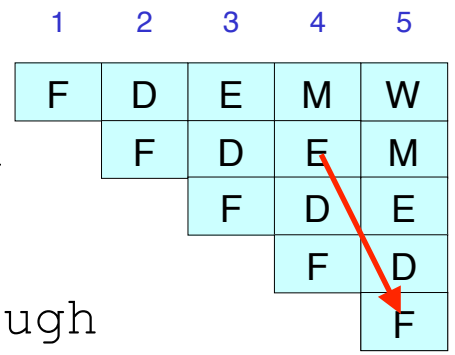
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

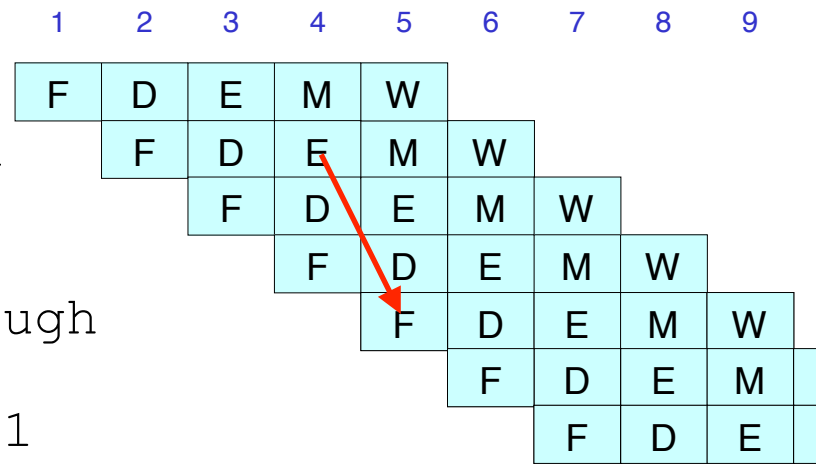
```
    xorg %rax, %rax
    jne L1              # Not taken
    nop
    nop
    irmovq $1, %rax     # Fall Through
L1  irmovq $4, %rcx     # Target
    irmovq $3, %rax     # Target + 1
```



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1                 # Not taken
     nop
     nop
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```

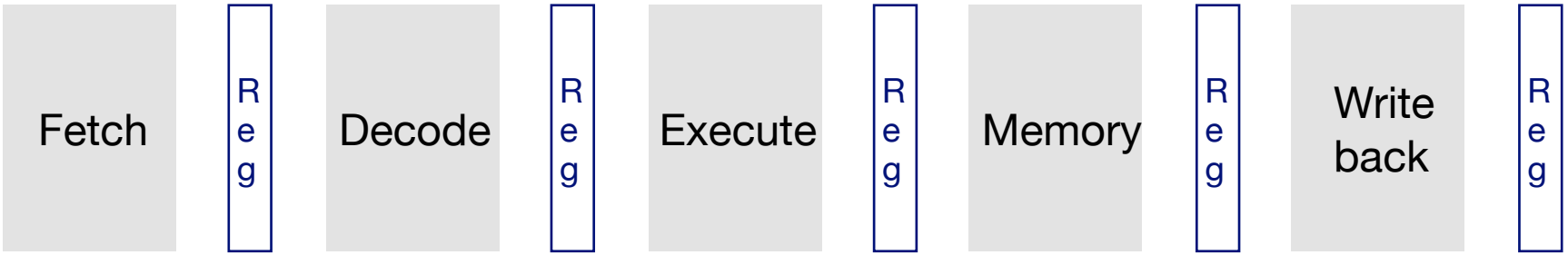


Resolving Control Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach
- Hardware mechanisms
  - Stalling (Think of it as hardware automatically inserting nops)
  - Branch Prediction
  - Return Address Stack

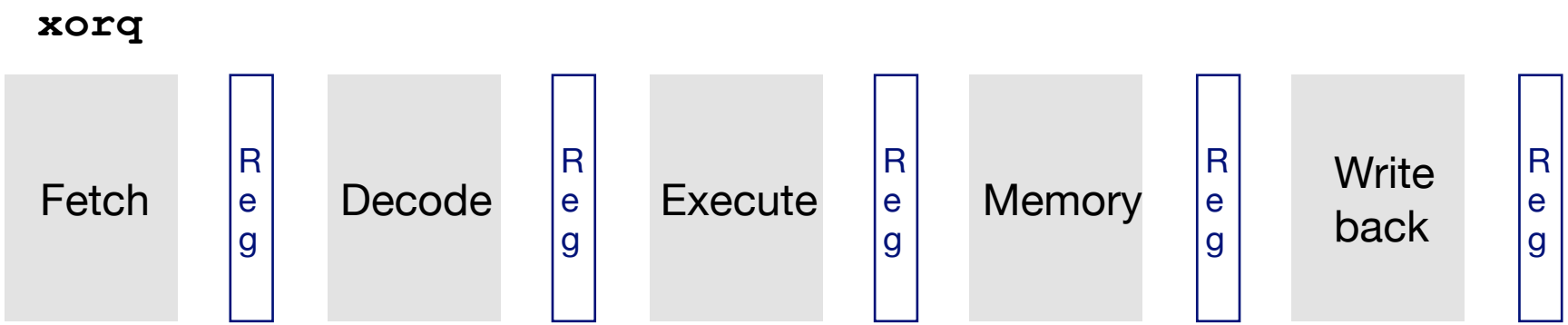
Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



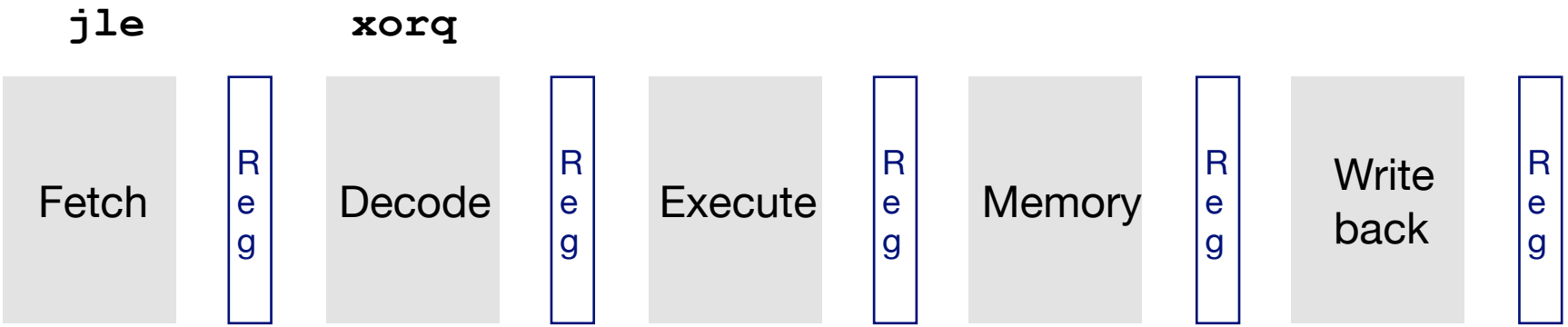
Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



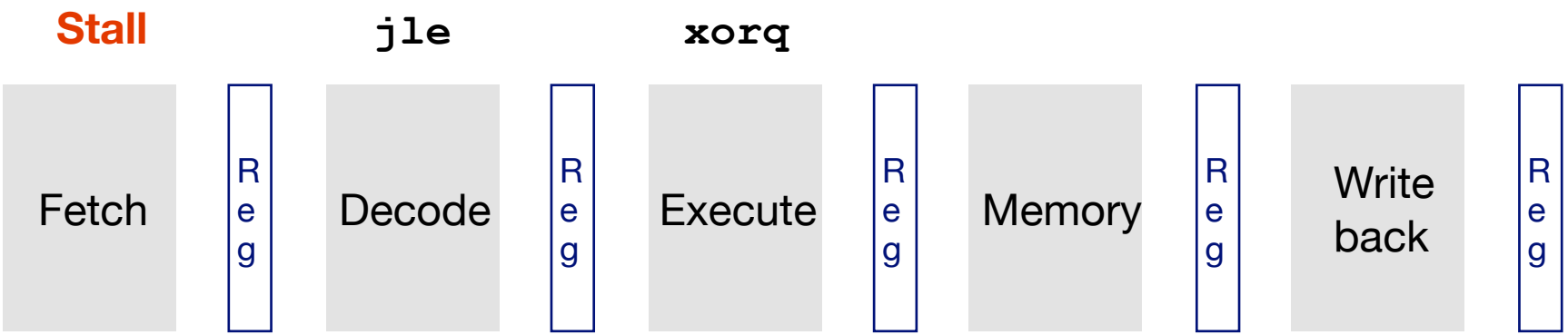
Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



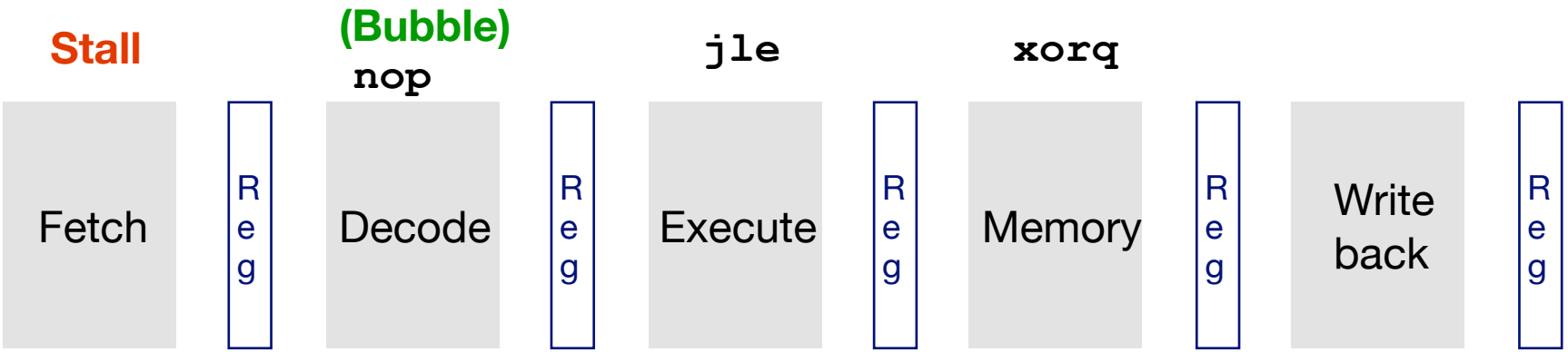
Hardware Generated Nops (Bubble and Stalling)

- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



Hardware Generated Nops (Bubble and Stalling)

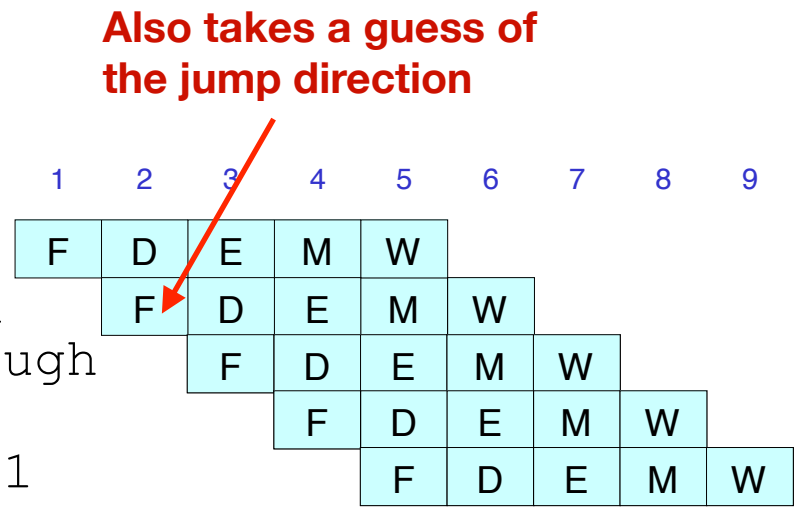
- **Stall** : the pipeline register shouldn't be written
- **Bubble** : signals correspond to a nop
- Why is it good for the hardware to do so anyways?



Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

```
     xorg %rax, %rax
     jne L1                 # Not taken
     irmovq $1, %rax        # Fall Through
L1   irmovq $4, %rcx        # Target
     irmovq $3, %rax        # Target + 1
```

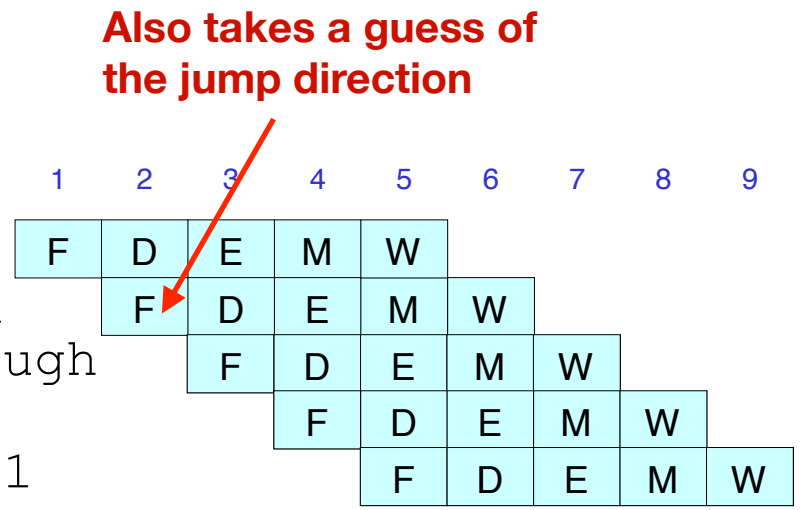


Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

```
    xorg %rax, %rax
    jne L1                  # Not taken
    irmovq $1, %rax         # Fall Through
L1  irmovq $4, %rcx         # Target
    irmovq $3, %rax         # Target + 1
```



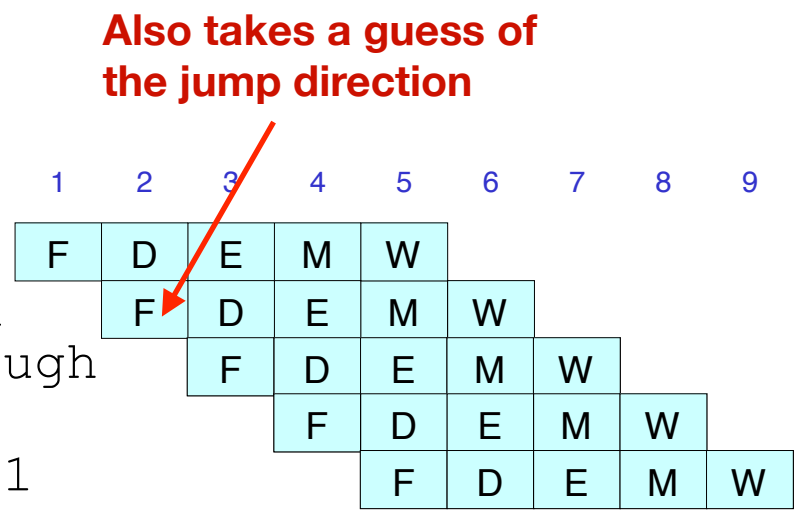
Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

```
    xorg %rax, %rax
    jne L1                  # Not taken
    irmovq $1, %rax         # Fall Through
L1  irmovq $4, %rcx         # Target
    irmovq $3, %rax         # Target + 1
```



Branch Prediction

Idea: instead of waiting, why not just guess the direction of jump?

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

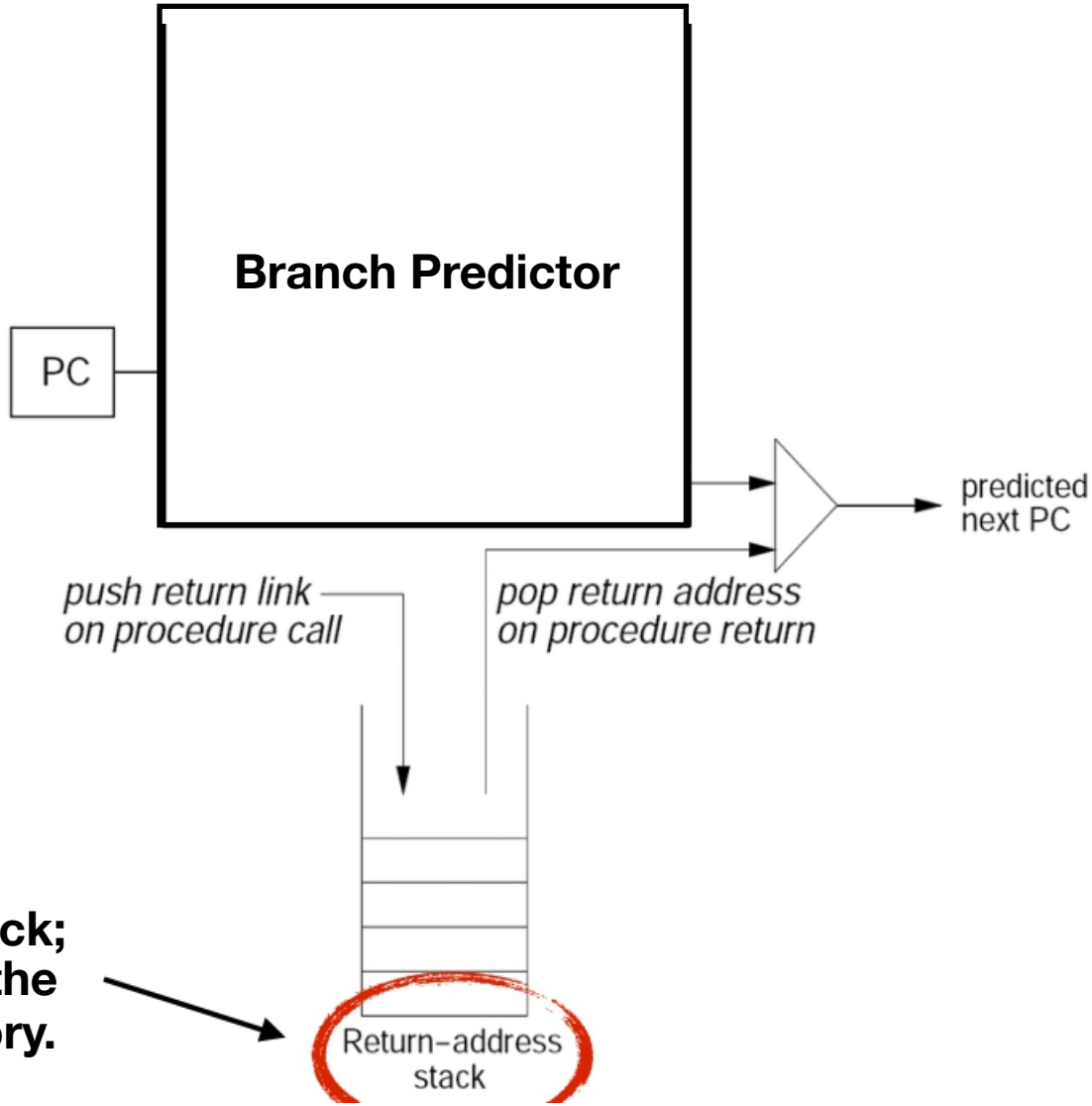
Static Prediction

- Always Taken
- Always Not-taken

Dynamic Prediction

- Dynamically predict taken/not-taken for each specific jump instruction

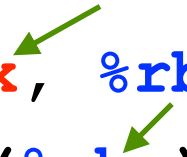
Return Address Stack (RAS)



A hardware stack; different from the stack in memory.

Data Dependencies

```
1     irmovq $50,   %rax
2     addq   %rax, %rbx
3     mrmovq 100(%rbx),   %rdx
```



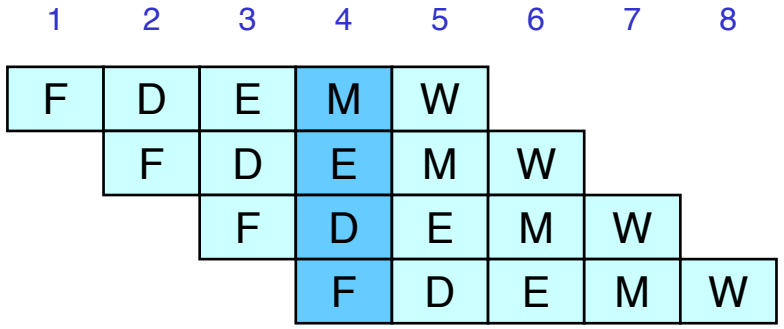
- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

Resolving Data Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution

Data Forwarding Example

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

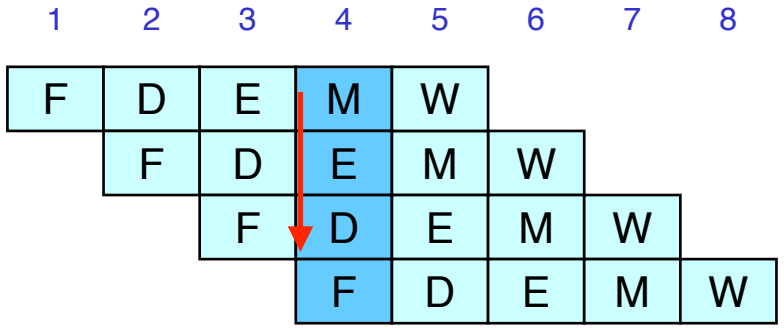
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

Data Forwarding Example

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
     0x016: halt
```



Register `%rdx`

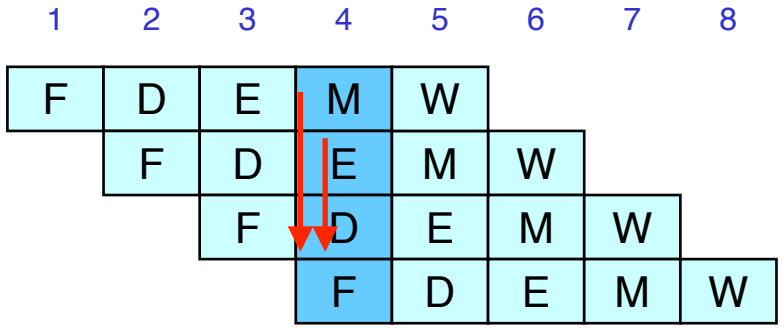
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

Data Forwarding Example

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
     0x016: halt
```



Register `%rdx`

- Forward from the memory stage

Register `%rax`

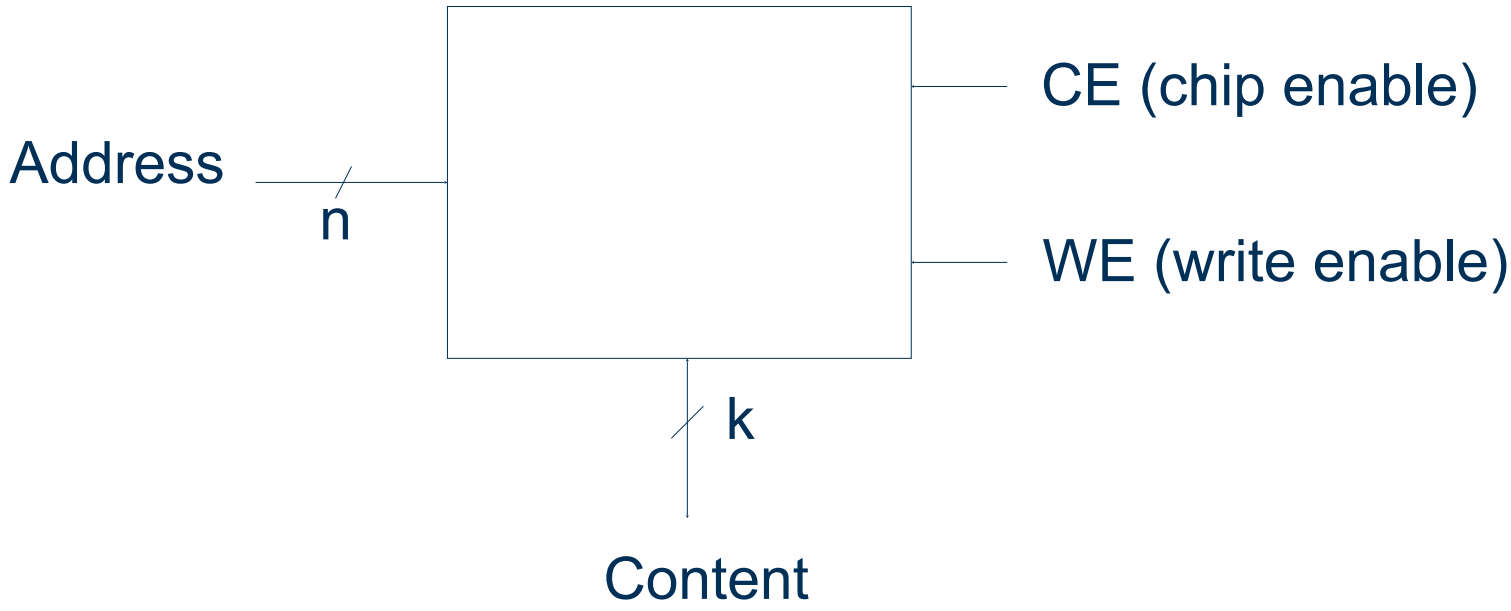
- Forward from the execute stage

Memory System

- Memory technology
- Memory trade-offs
- Memory hierarchy
- Cache
  - Basic ideas
  - Different design choices
  - Replacement policy

Memory Technology: RAM

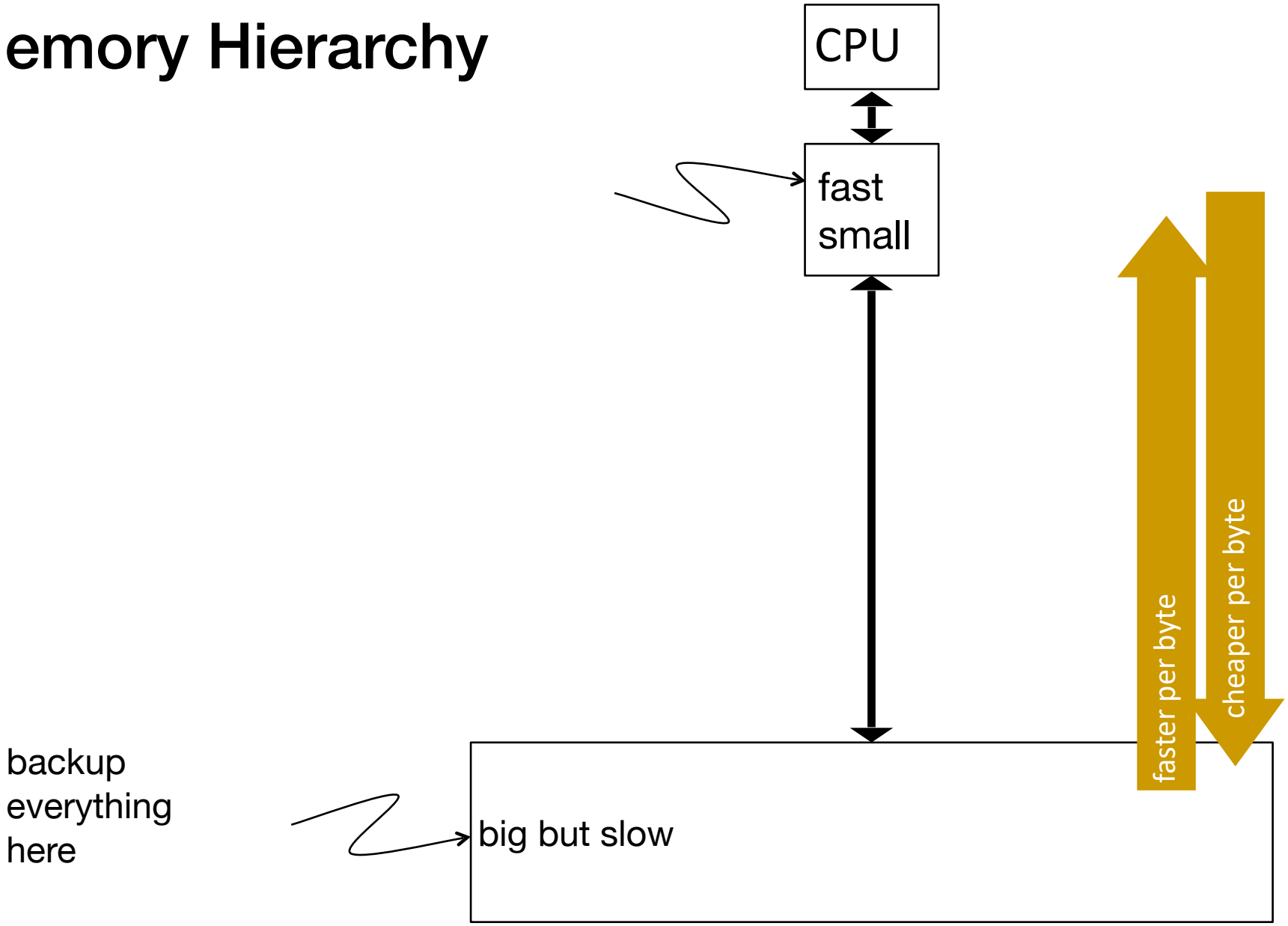
- Random access memory
- Random access means you can supply an arbitrary address to the memory and get a value back



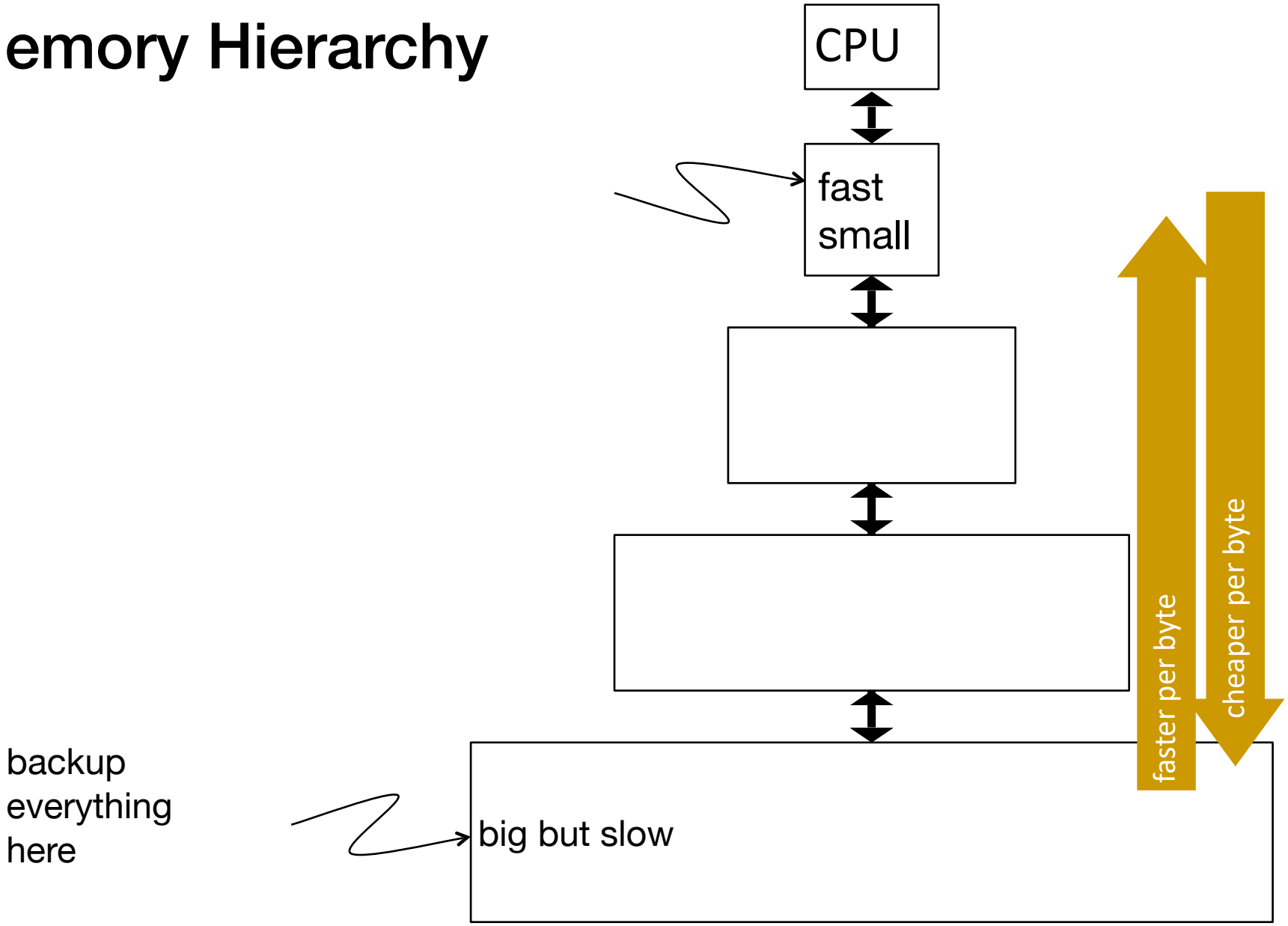
Summary of Trade-Offs

- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
- Larger capacity is slower
  - Flip-flops/Small SRAM, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Other technologies have their place as well
  - PC-RAM, MRAM, RRAM

Memory Hierarchy

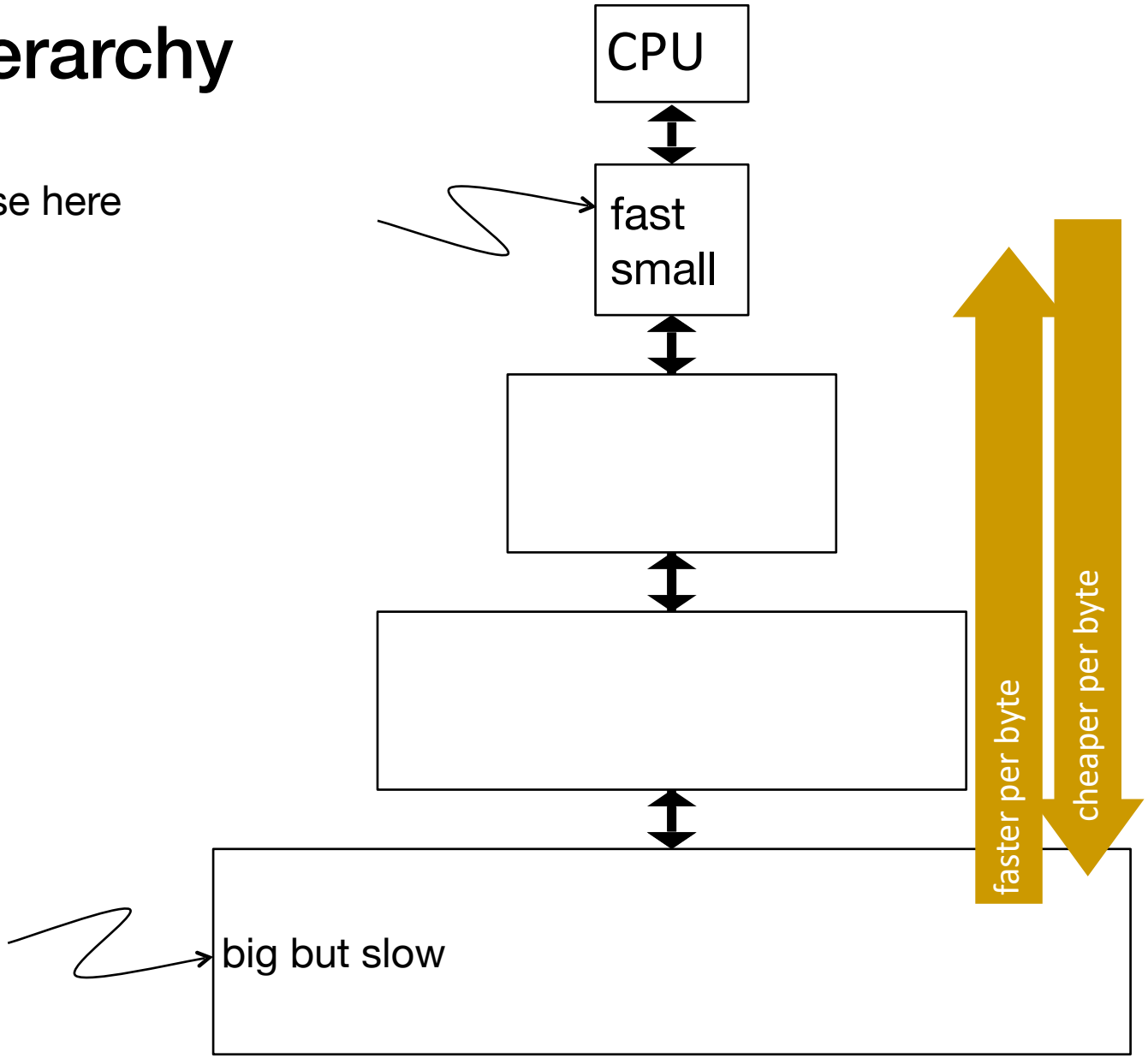


Memory Hierarchy



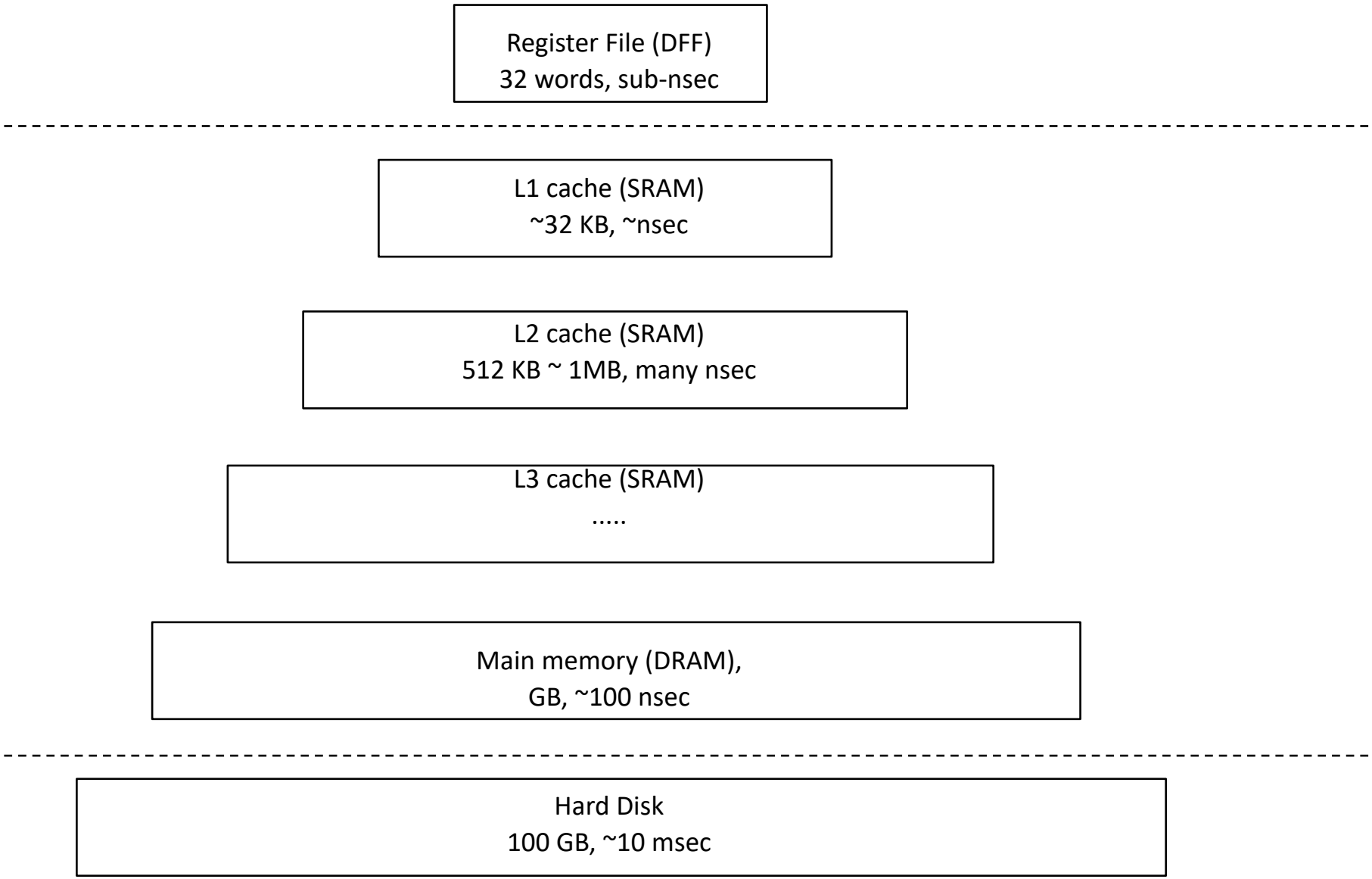
Memory Hierarchy

move what you use here

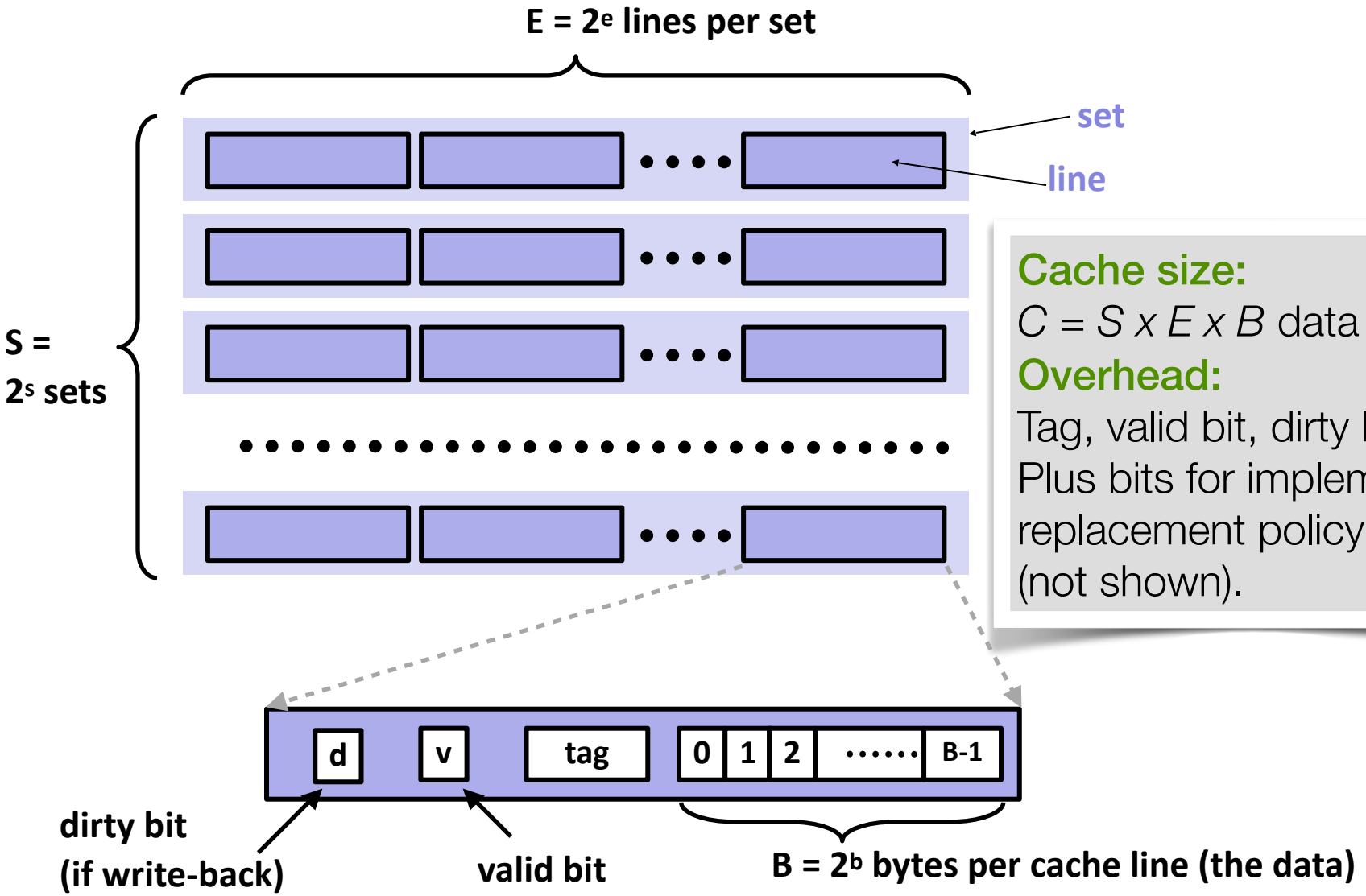


backup
everything
here

A Modern Memory Hierarchy



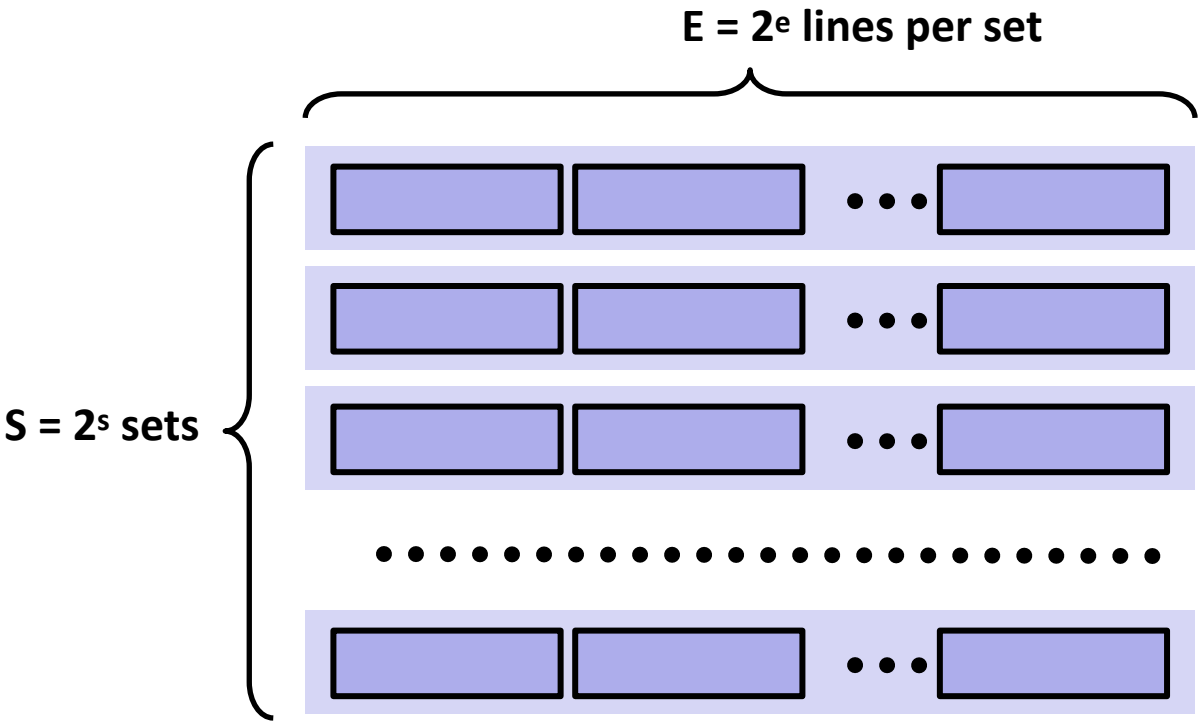
General Cache Organization (S, E, B)



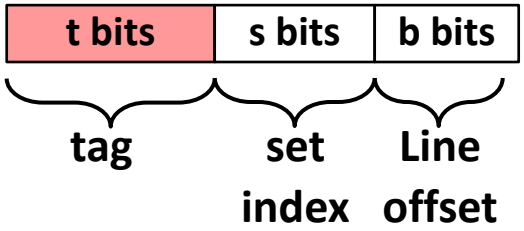
Cache size:
$C = S \times E \times B$ data bytes

Overhead:
Tag, valid bit, dirty bit.
Plus bits for implementing replacement policy (not shown).

Cache Access

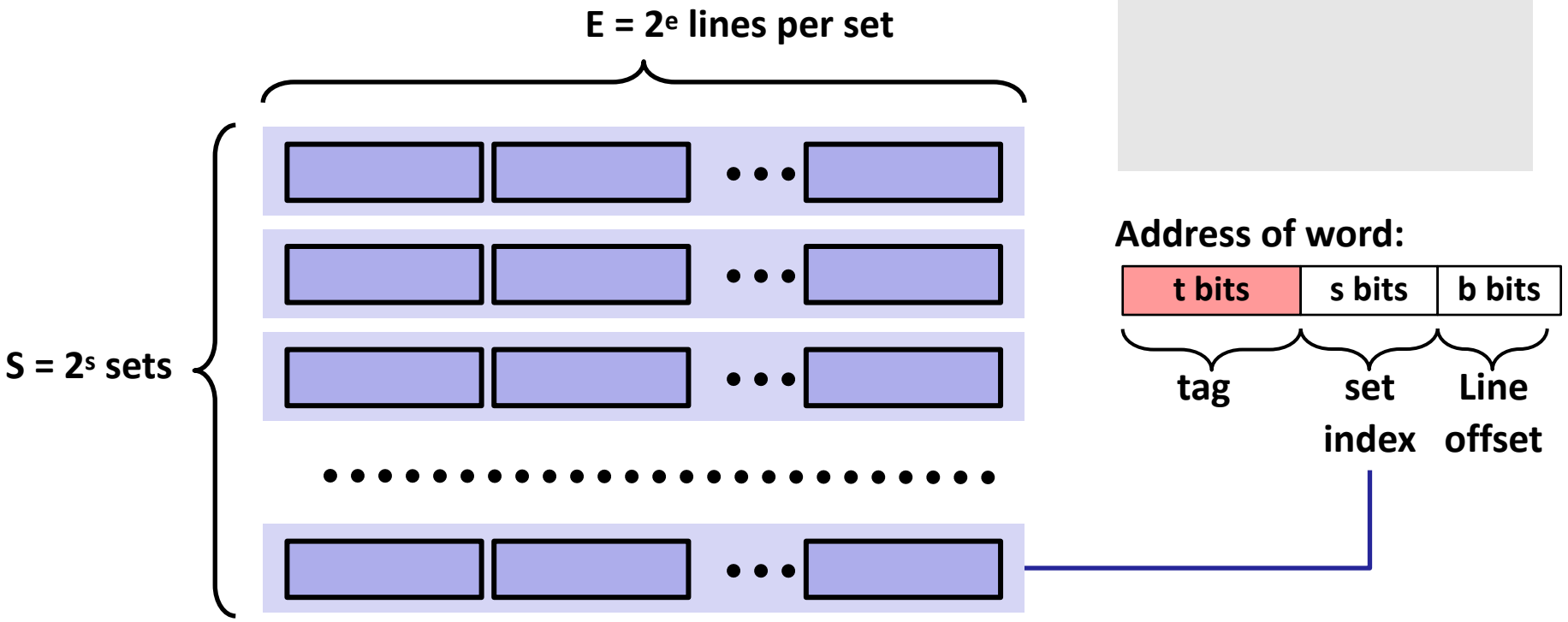


Address of word:



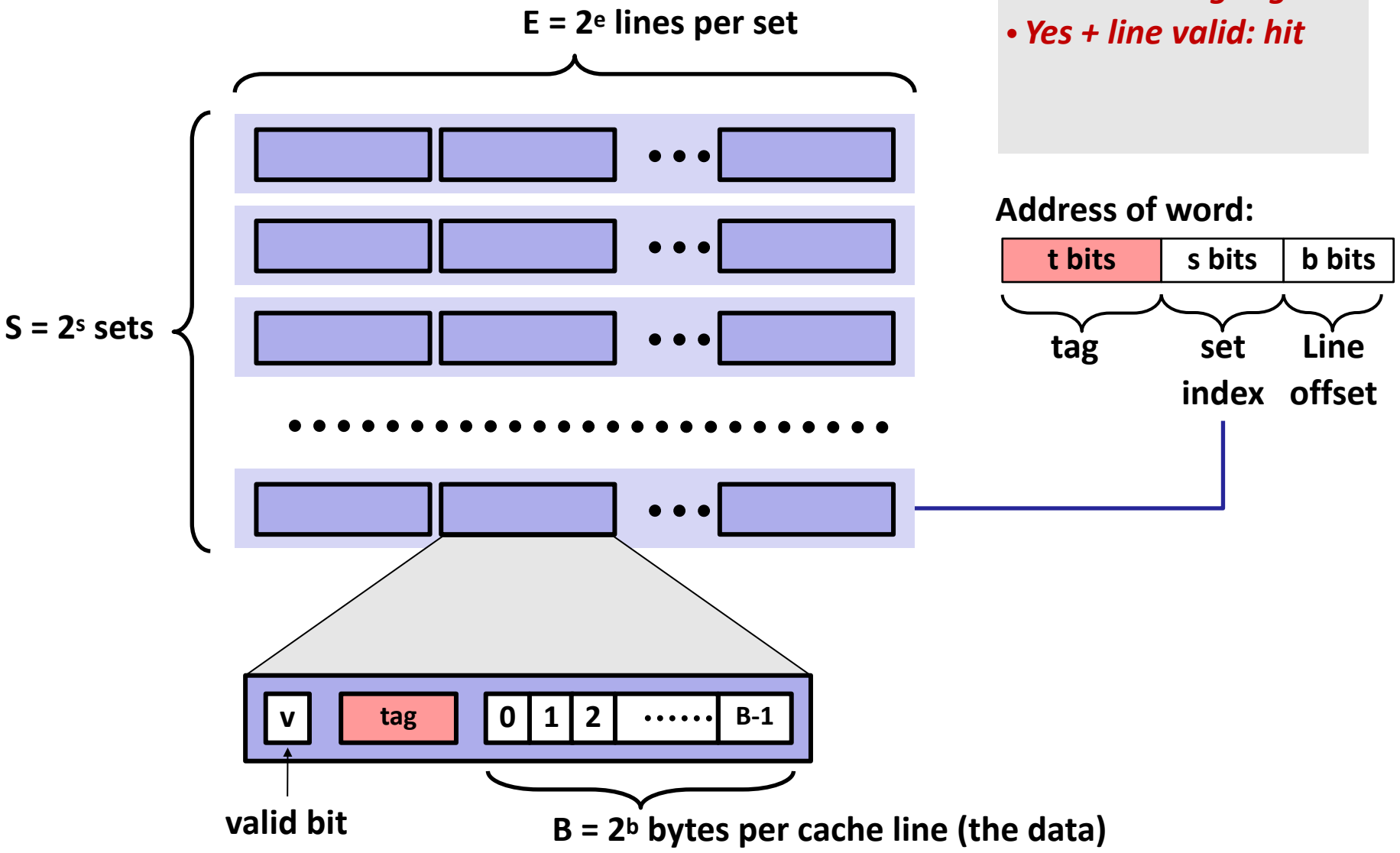
Cache Access

- *Locate set*



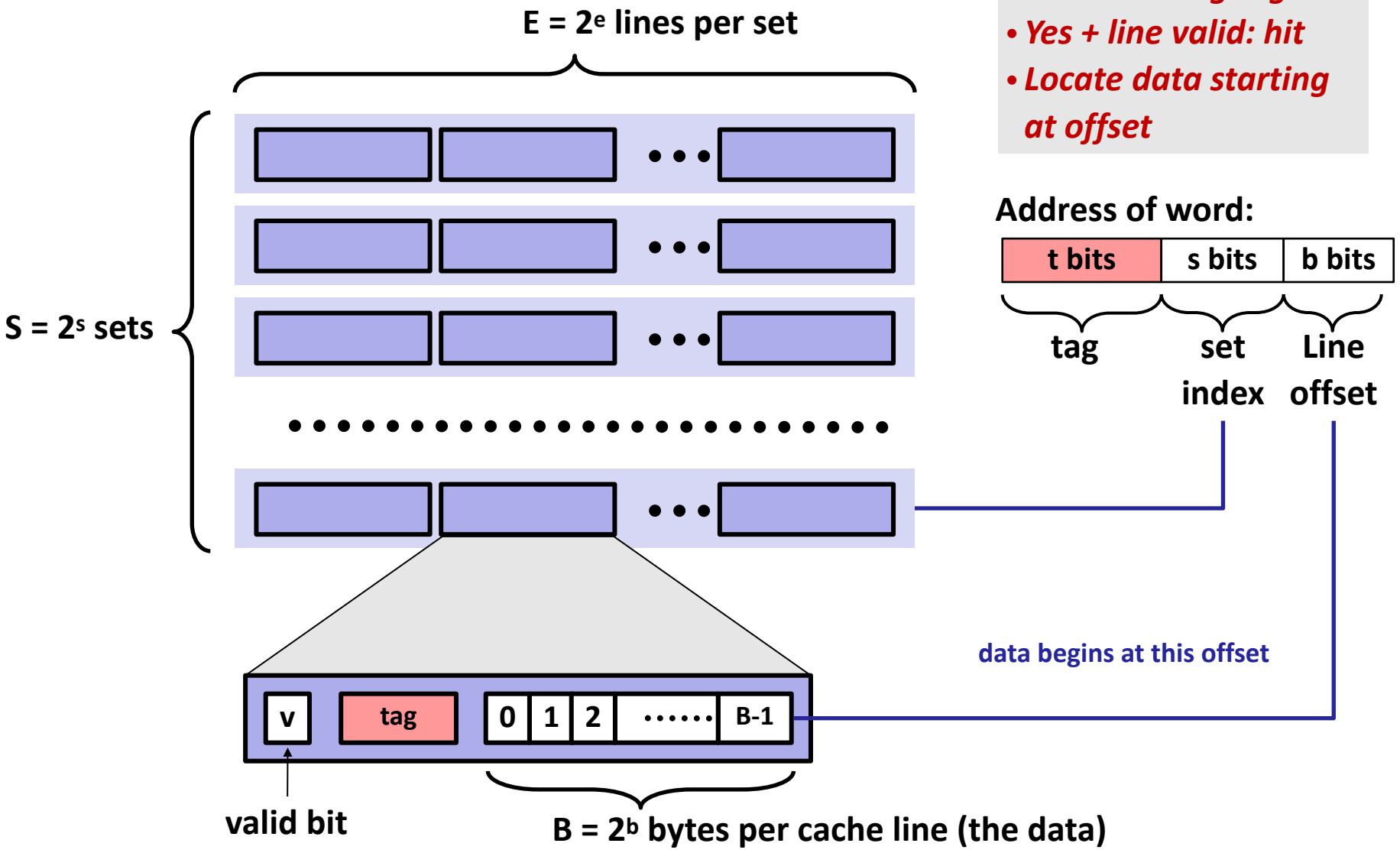
Cache Access

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*



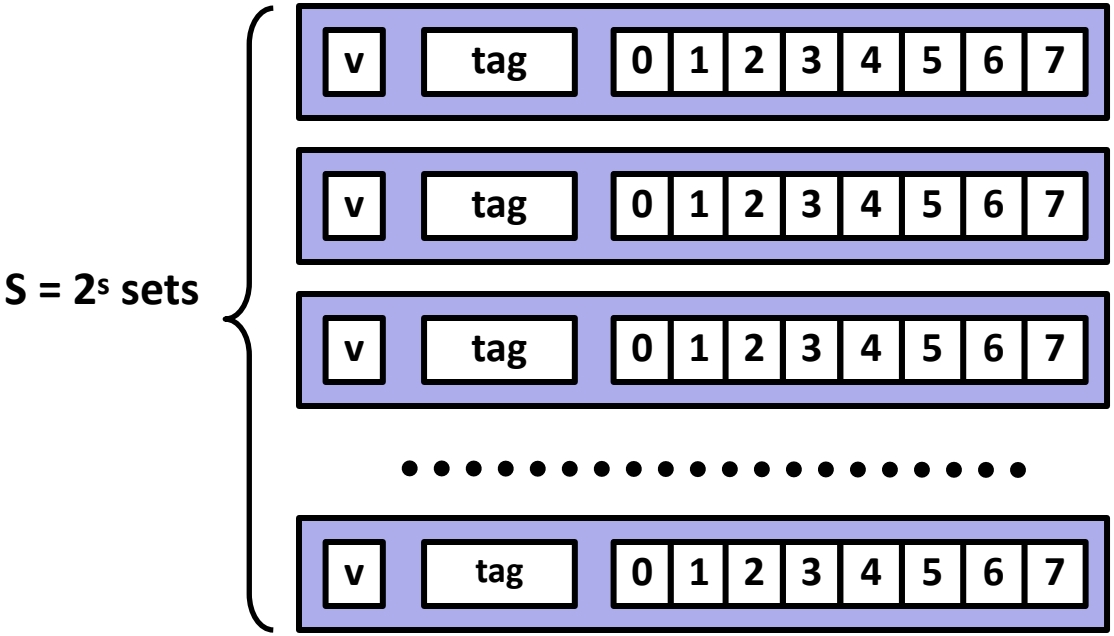
Cache Access

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

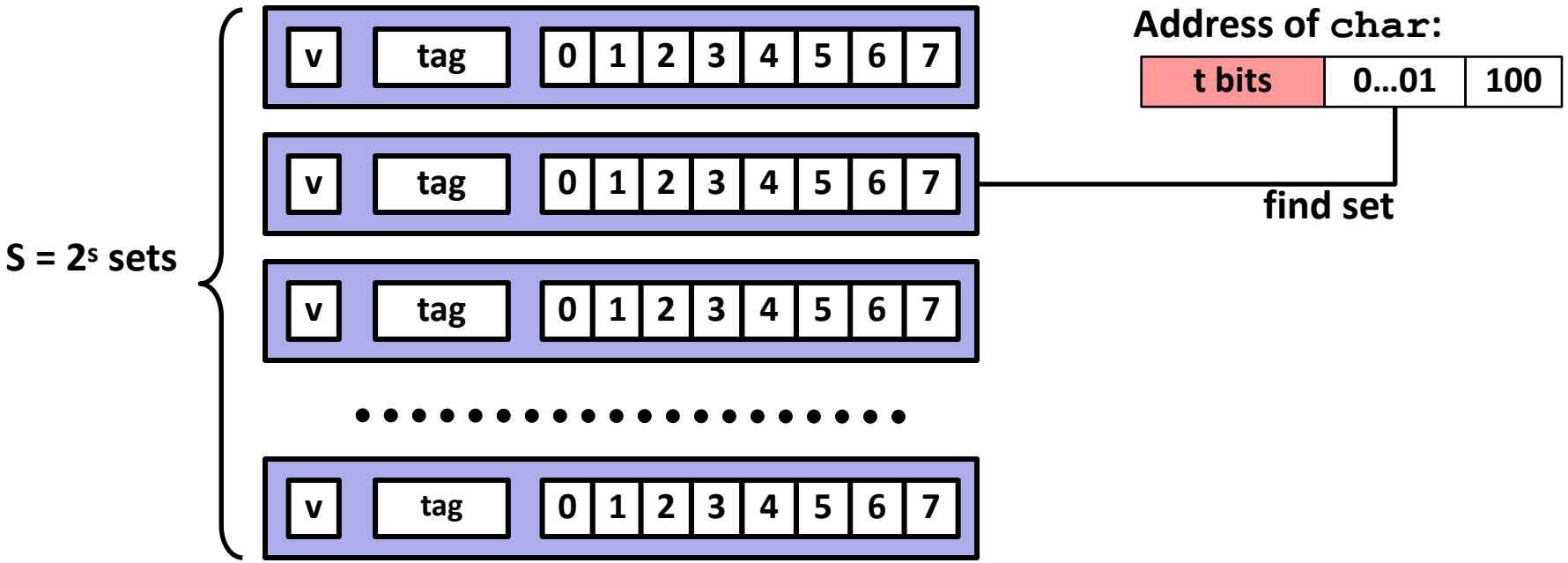


Address of `char`:

t bits	0…01	100

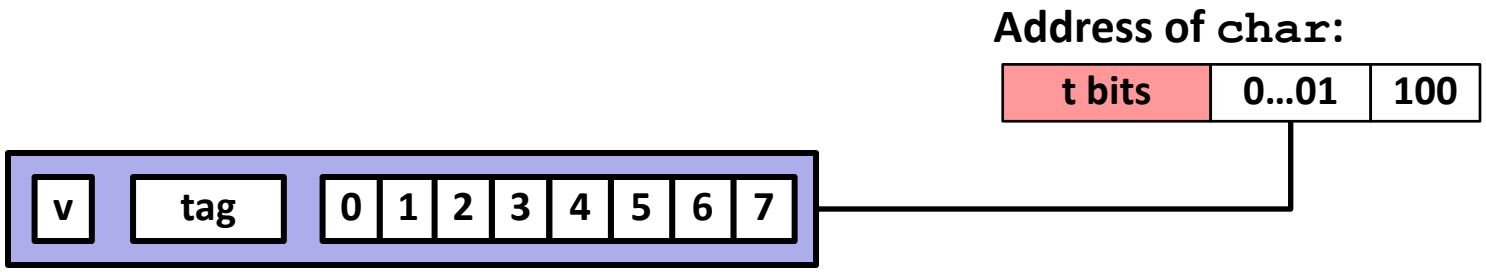
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



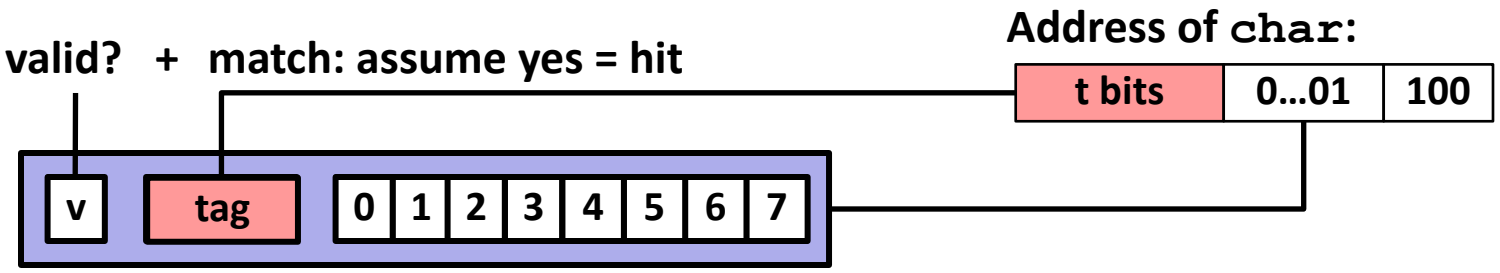
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



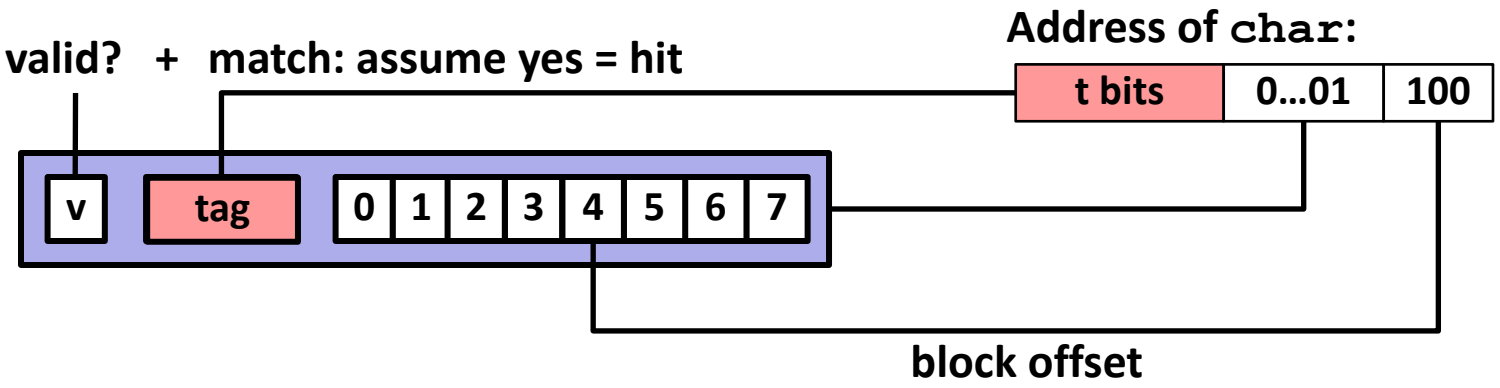
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



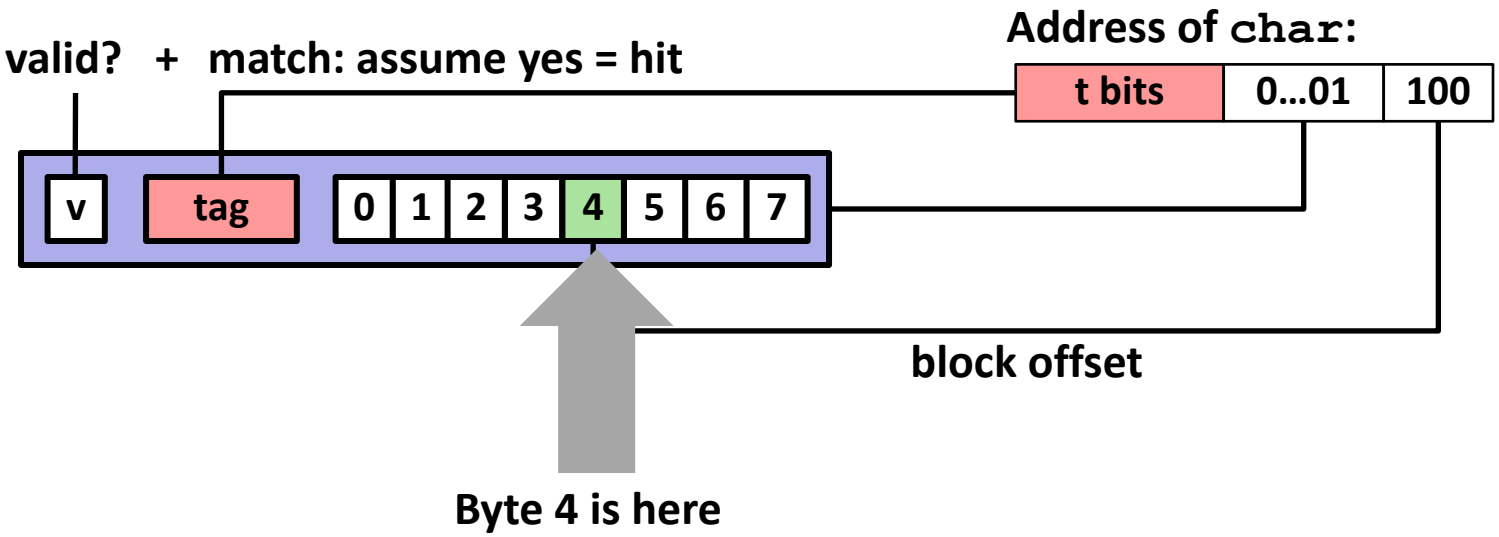
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



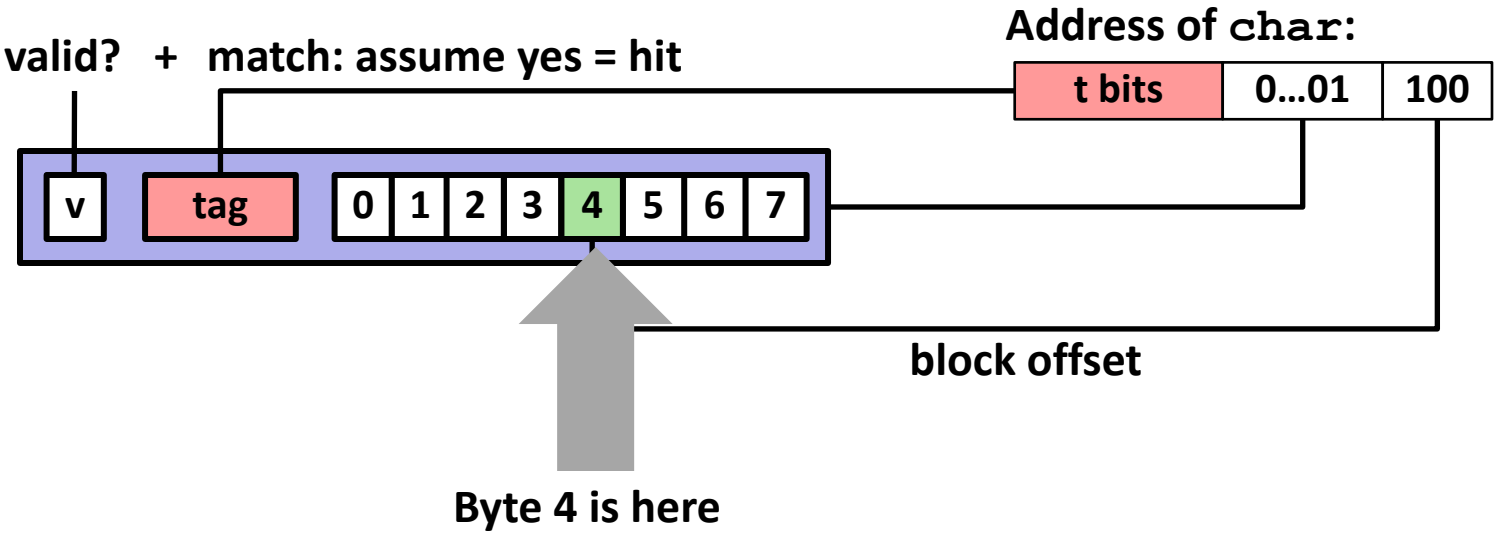
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes



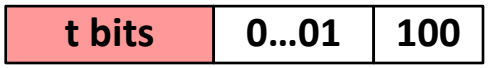
If tag doesn't match: old line is evicted and replaced

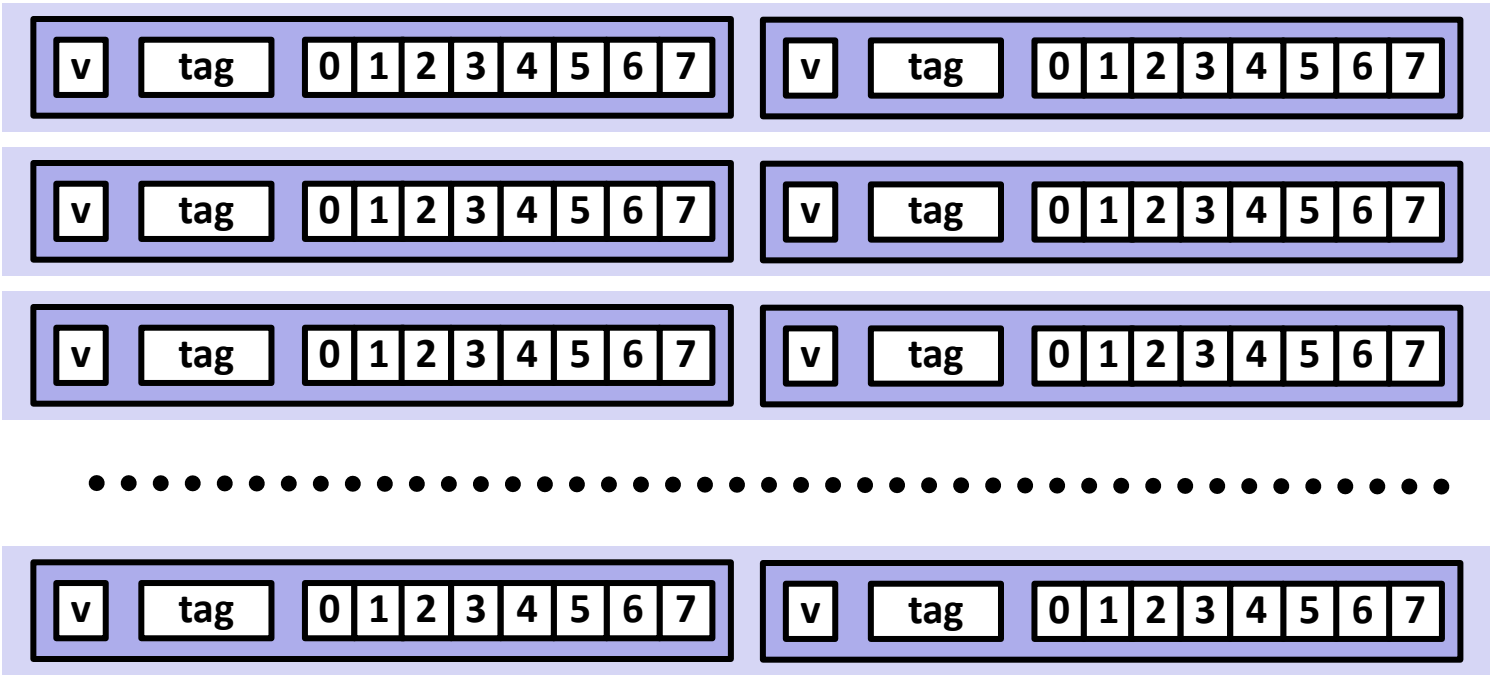
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache line size 8 bytes

Address of `short int`:

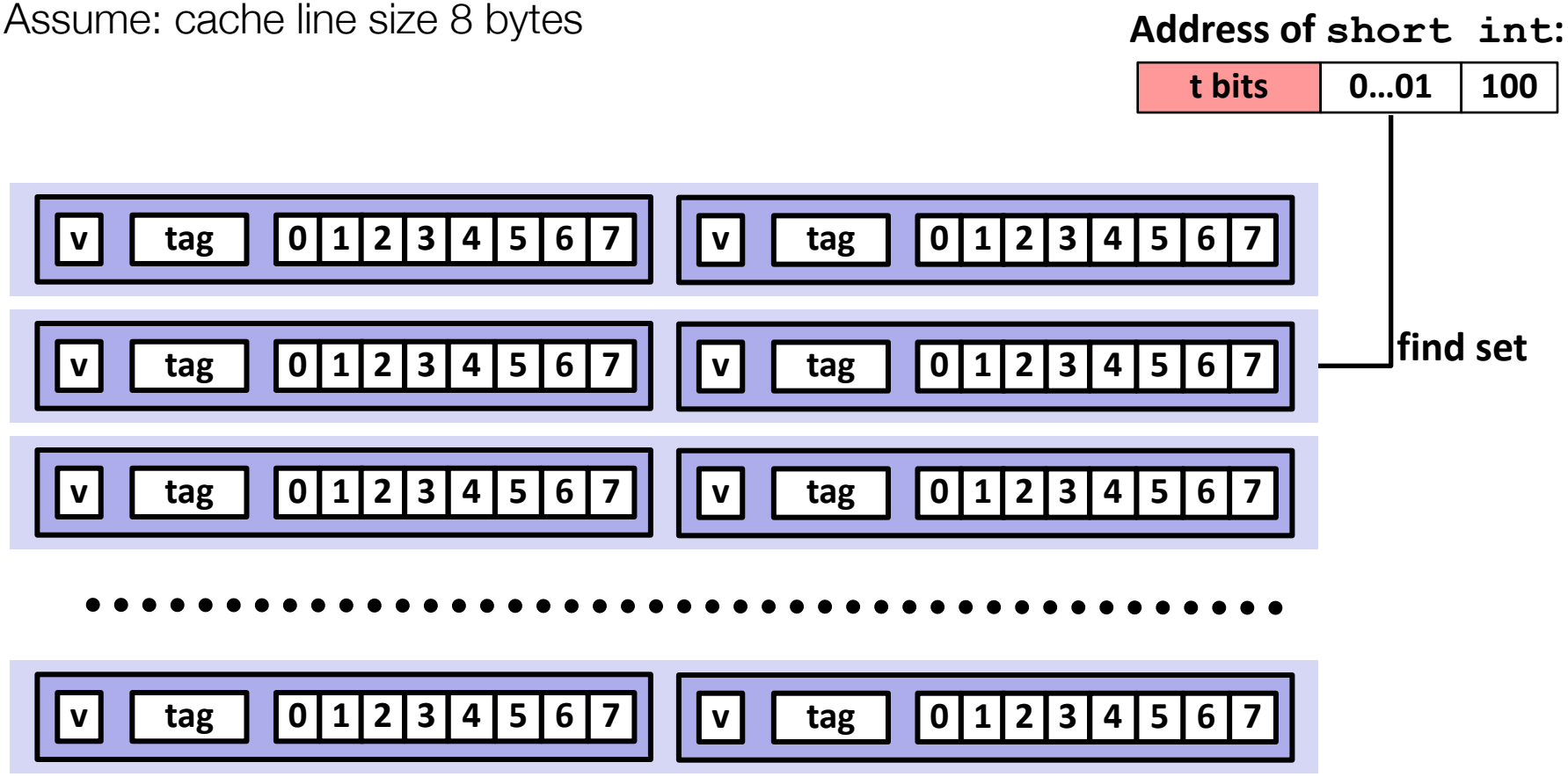




E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

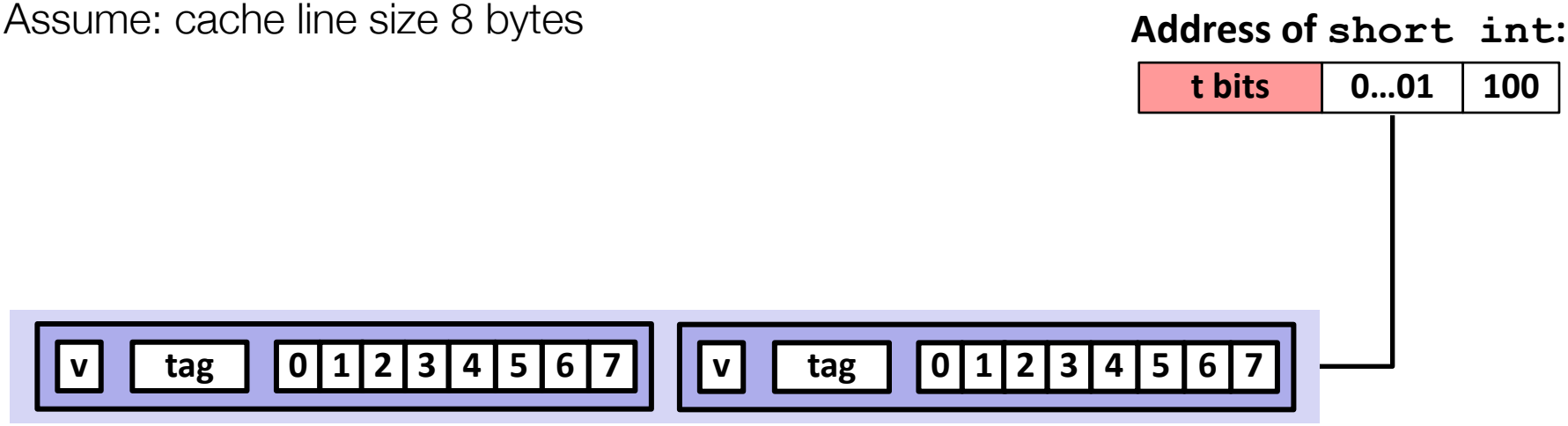
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

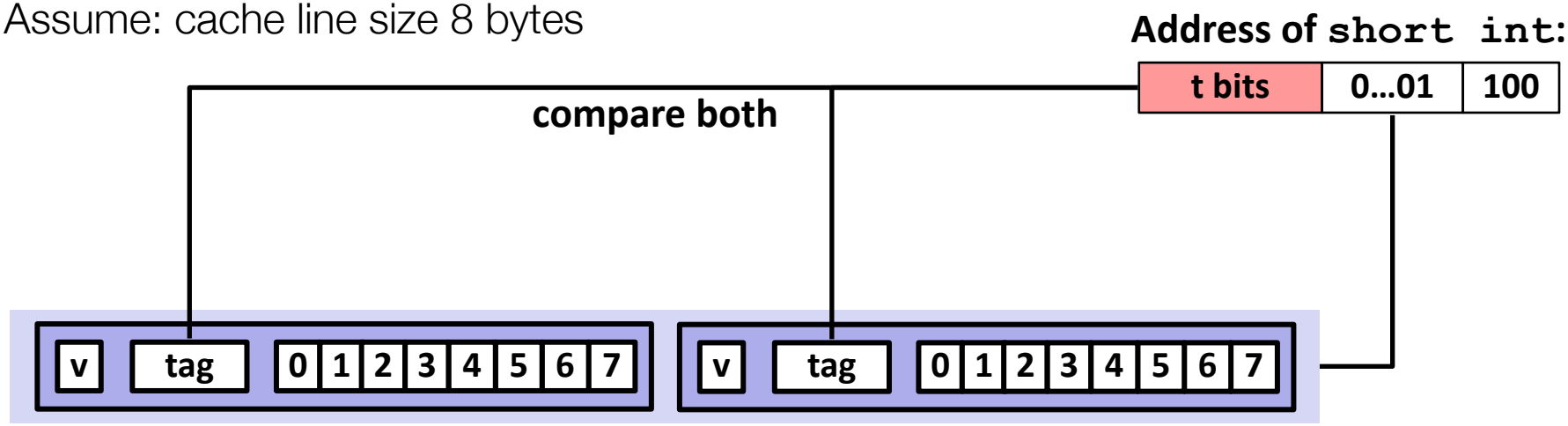
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

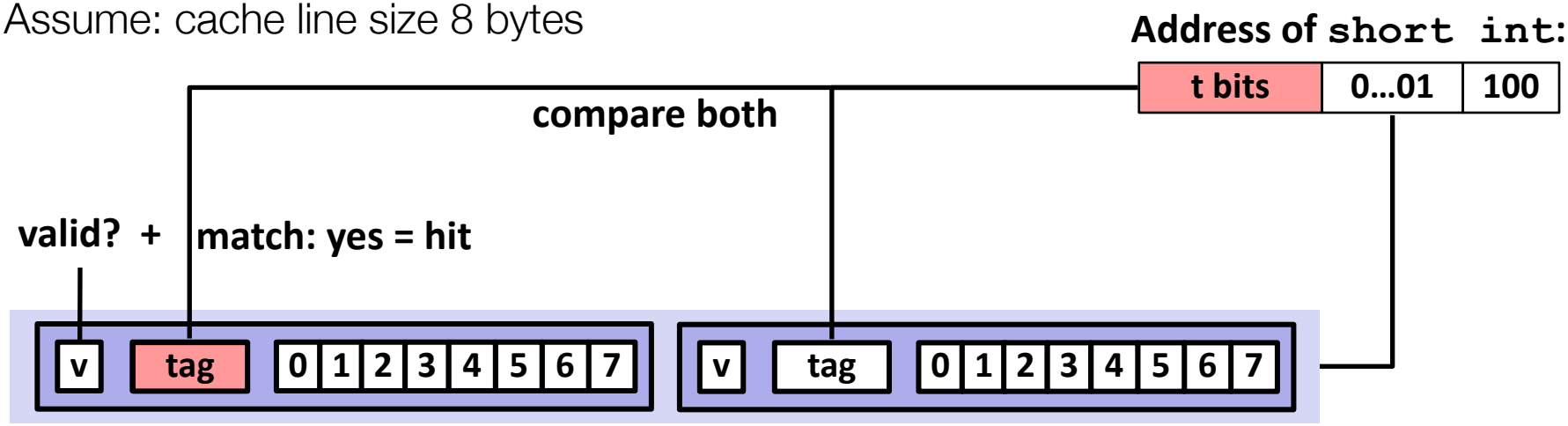
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

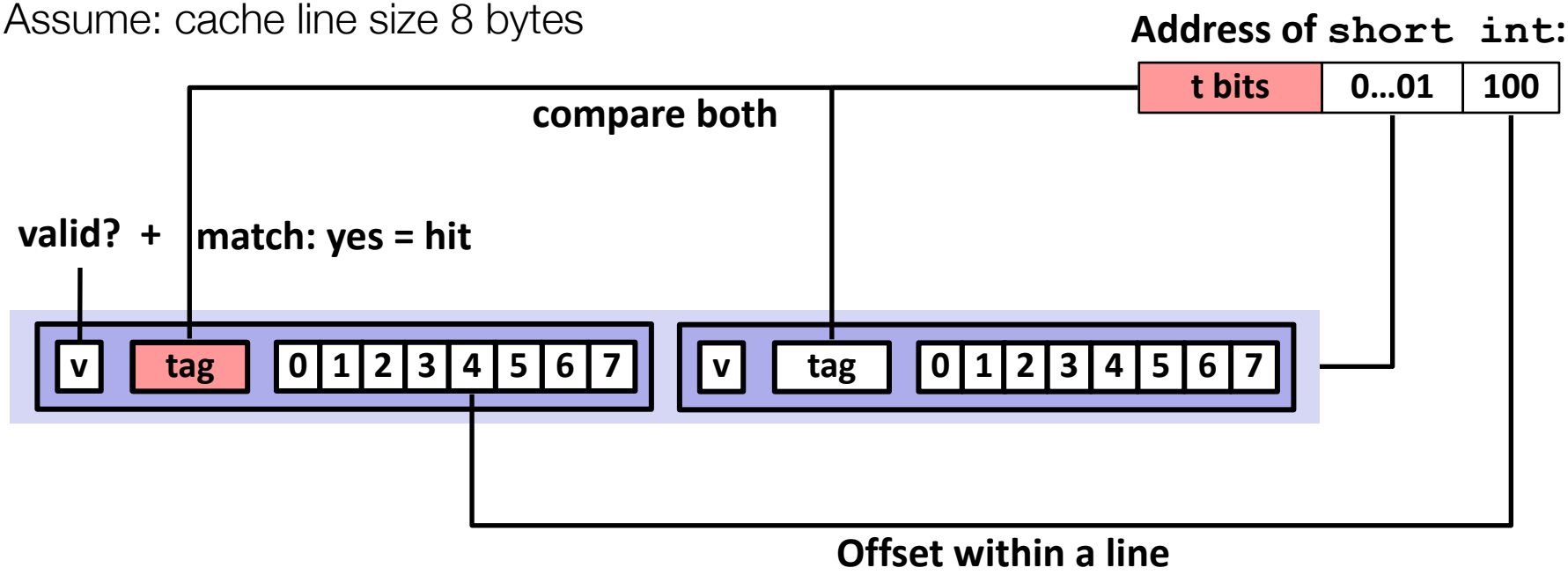
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

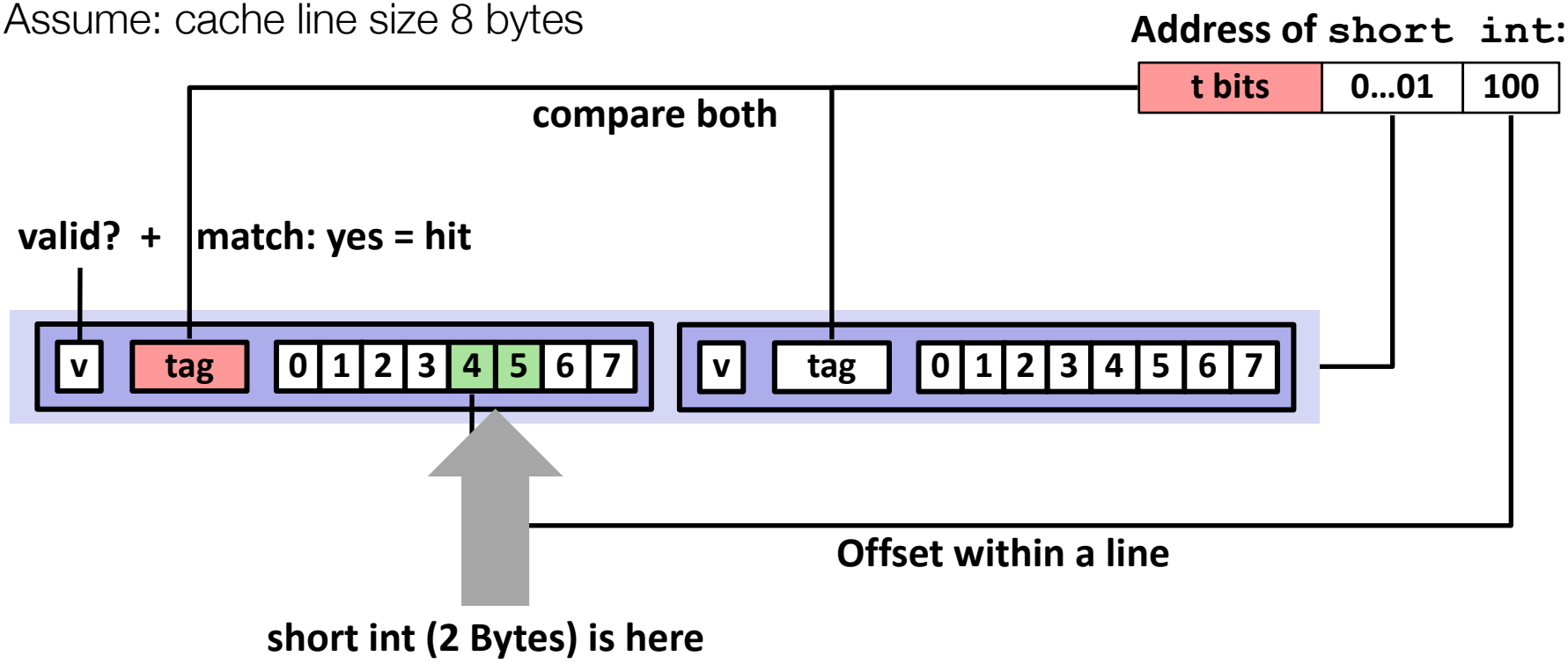
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

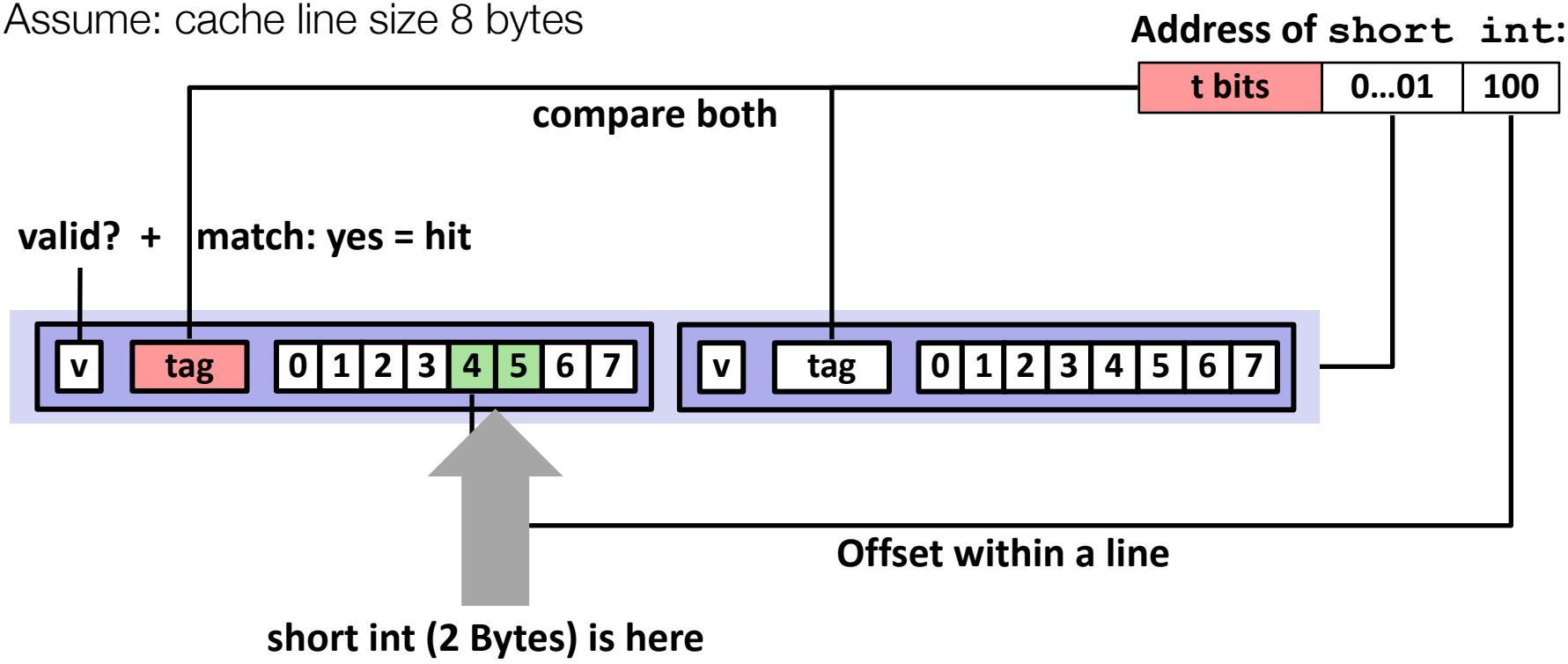
Assume: cache line size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache line size 8 bytes



No match:

- **One line in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU), …**

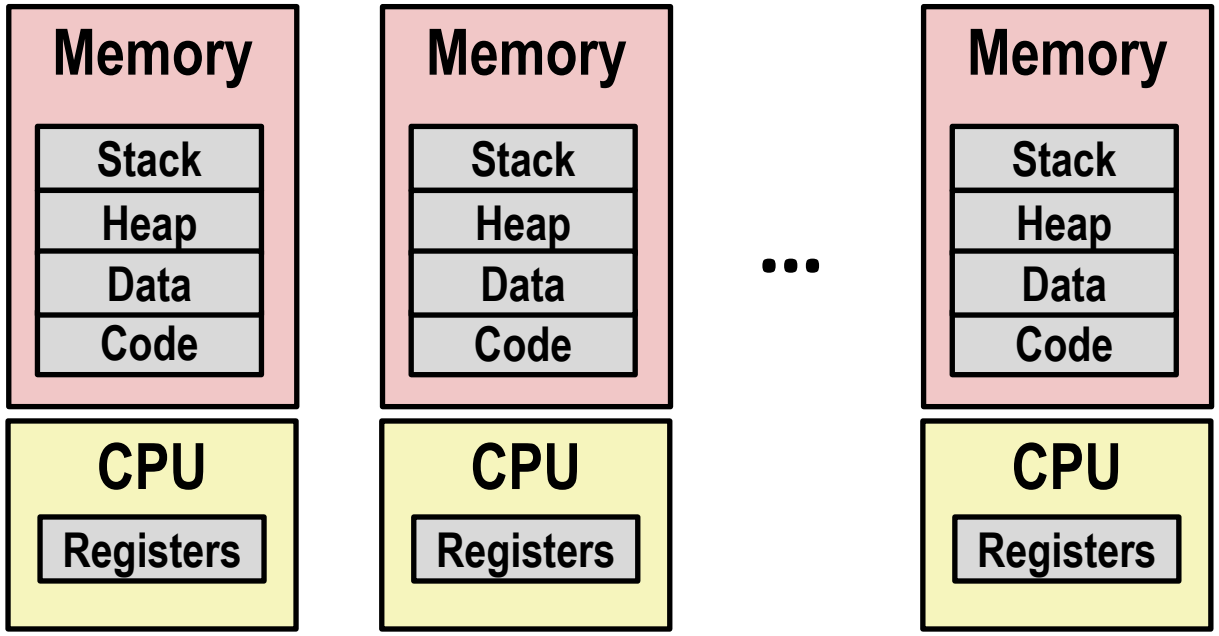
Eviction/Replacement Policy

- Which cache line should be replaced for associative cache?
  - Any invalid cache line first
  - If all are valid, consult the replacement policy
  - Randomly pick one???
  - Ideally: Replace the cache line that's least likely going to be used again
    - Approximation: Least recently used (LRU)
      - Essentially have to track the ordering of all cache lines
      - In reality, true LRU is never implemented. Too complex.
      - "Pseudo-LRU" is usually used in real processors.

Processes

- Definition
- Create a process
- Terminate a process
- Reap a process

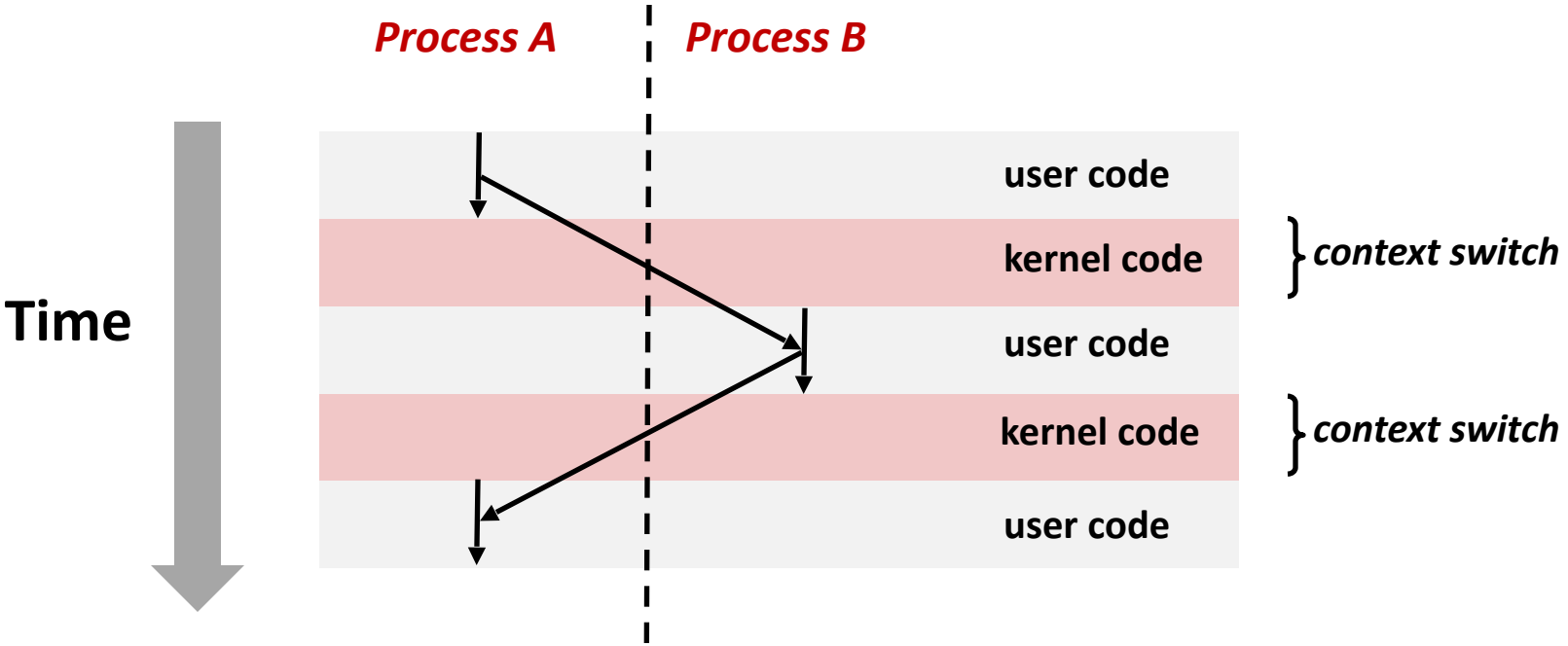
Multiprocessing: The Illusion



- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*



Creating Processes

- Parent process creates a new running child process by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent's (virtual) address space (i.e., same stack copies, code, etc.)
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called once but returns twice

Terminating Processes

- Process becomes terminated for one of three reasons:
  - Receiving a signal whose default action is to terminate
  - Returning from the `main` routine
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- `exit` is called once but never returns.

`wait`: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

`execve`: Loading and Running Programs

Executes "`/bin/ls -lt /usr/include`" in child process using current environment:

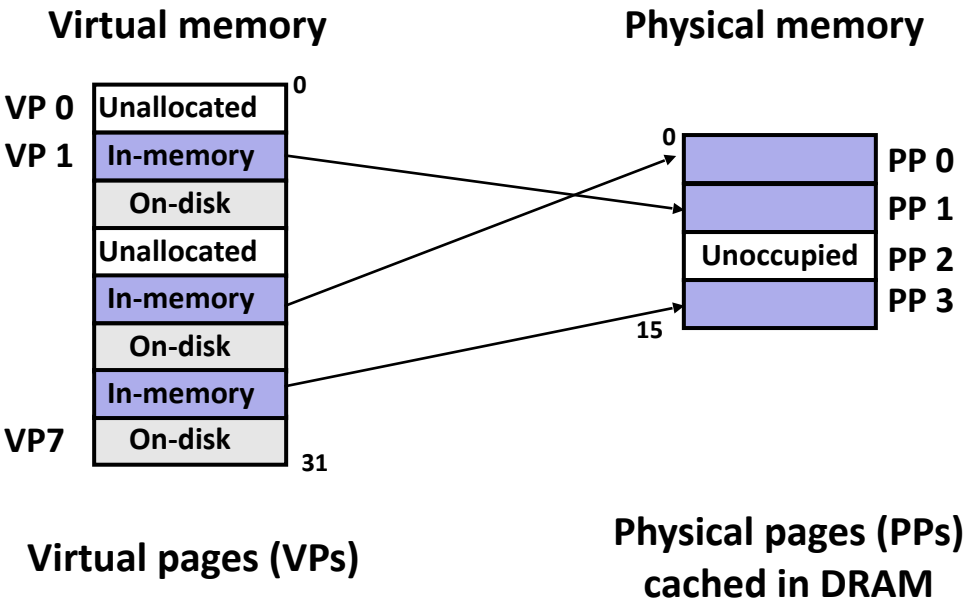
```
char *myargv[] = {"/bin/ls", "-lt", "/usr/include"};
char *environ[] = {"USER=droh", "PWD="/usr/droh"};

if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Virtual Memory

- Page table
  - Page hit
  - Page fault
- TLB
- Address translation process
- Multi-level page tables

VM Concepts

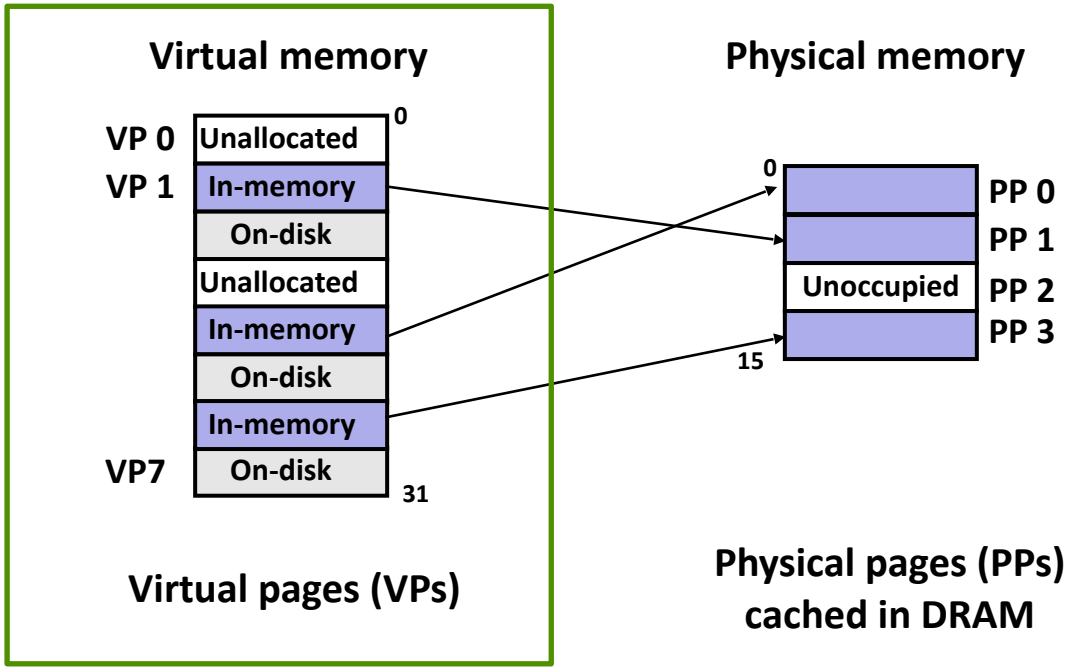
- *Virtual memory* is an array of N contiguous *pages* (each page has a certain amount of continuous bytes).
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").



VM Concepts

- *Virtual memory* is an array of N contiguous *pages* (each page has a certain amount of continuous bytes).
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

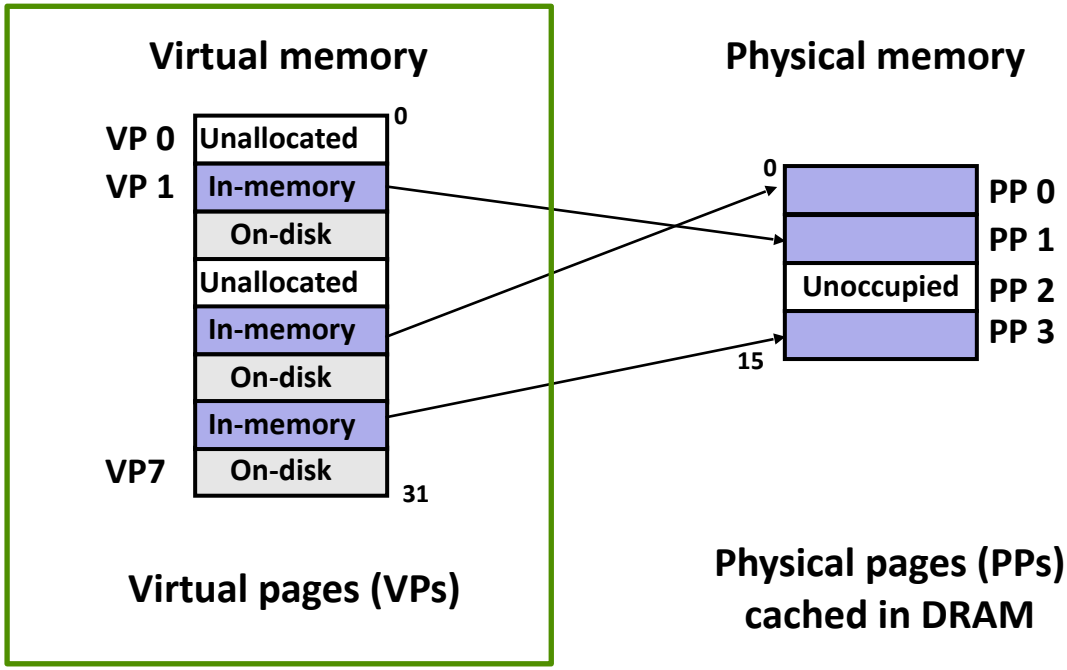
What programmers see



VM Concepts

- *Virtual memory* is an array of N contiguous *pages* (each page has a certain amount of continuous bytes).
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see

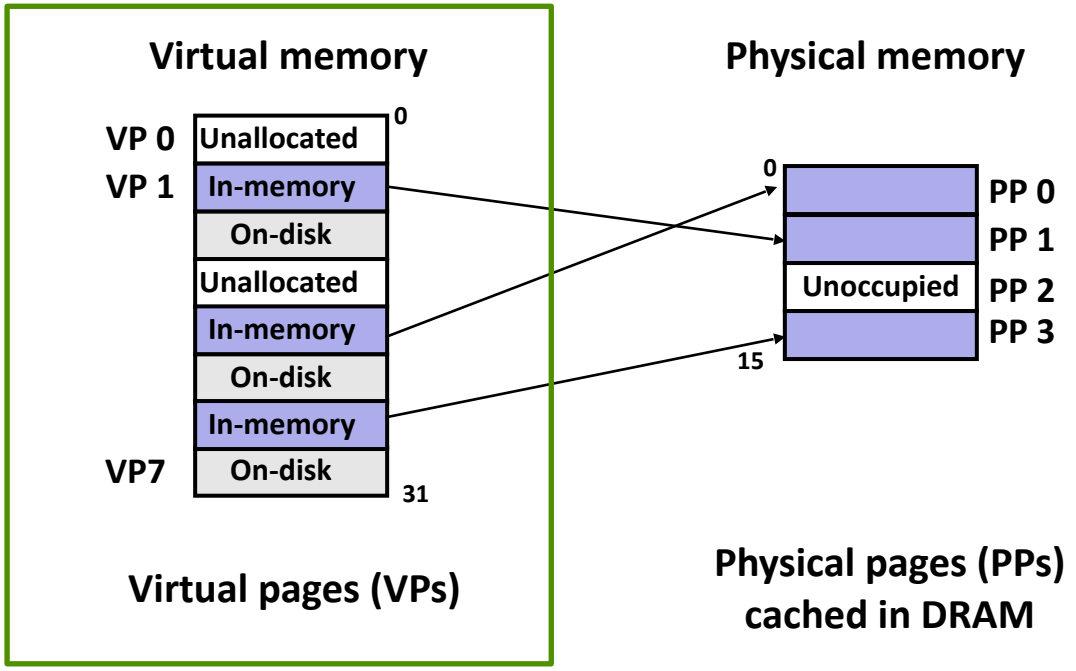


Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B

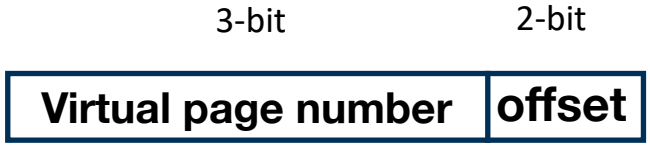
VM Concepts

- *Virtual memory* is an array of N contiguous *pages* (each page has a certain amount of continuous bytes).
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

What programmers see



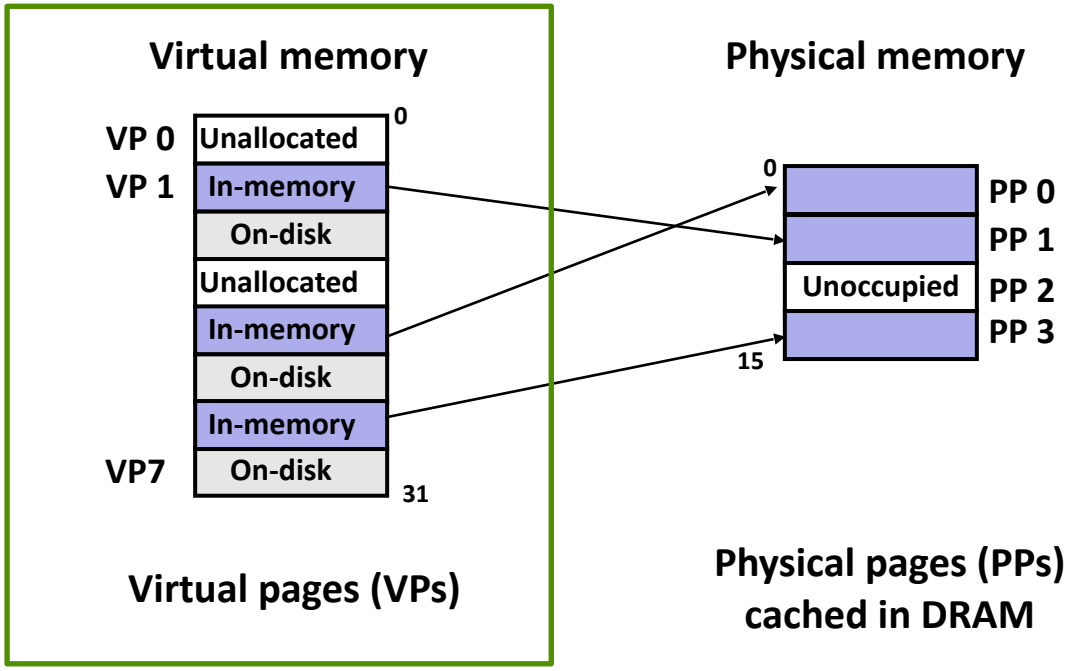
Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B



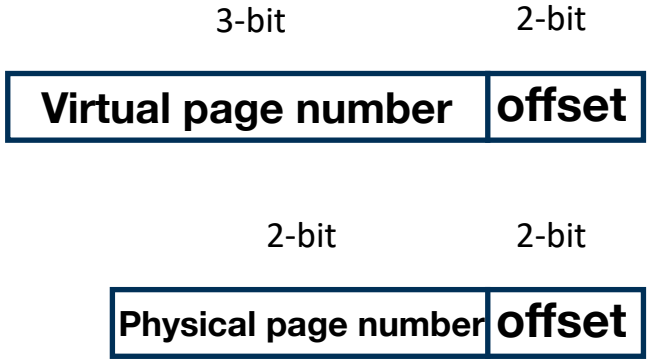
VM Concepts

- *Virtual memory* is an array of N contiguous *pages* (each page has a certain amount of continuous bytes).
- Physical memory is also divided into pages. Each physical page (sometimes called *frames*) has the same size as a virtual page. Physical memory has way fewer pages.
- A page can either be on the ("uncached") disk or in the physical memory ("cached").

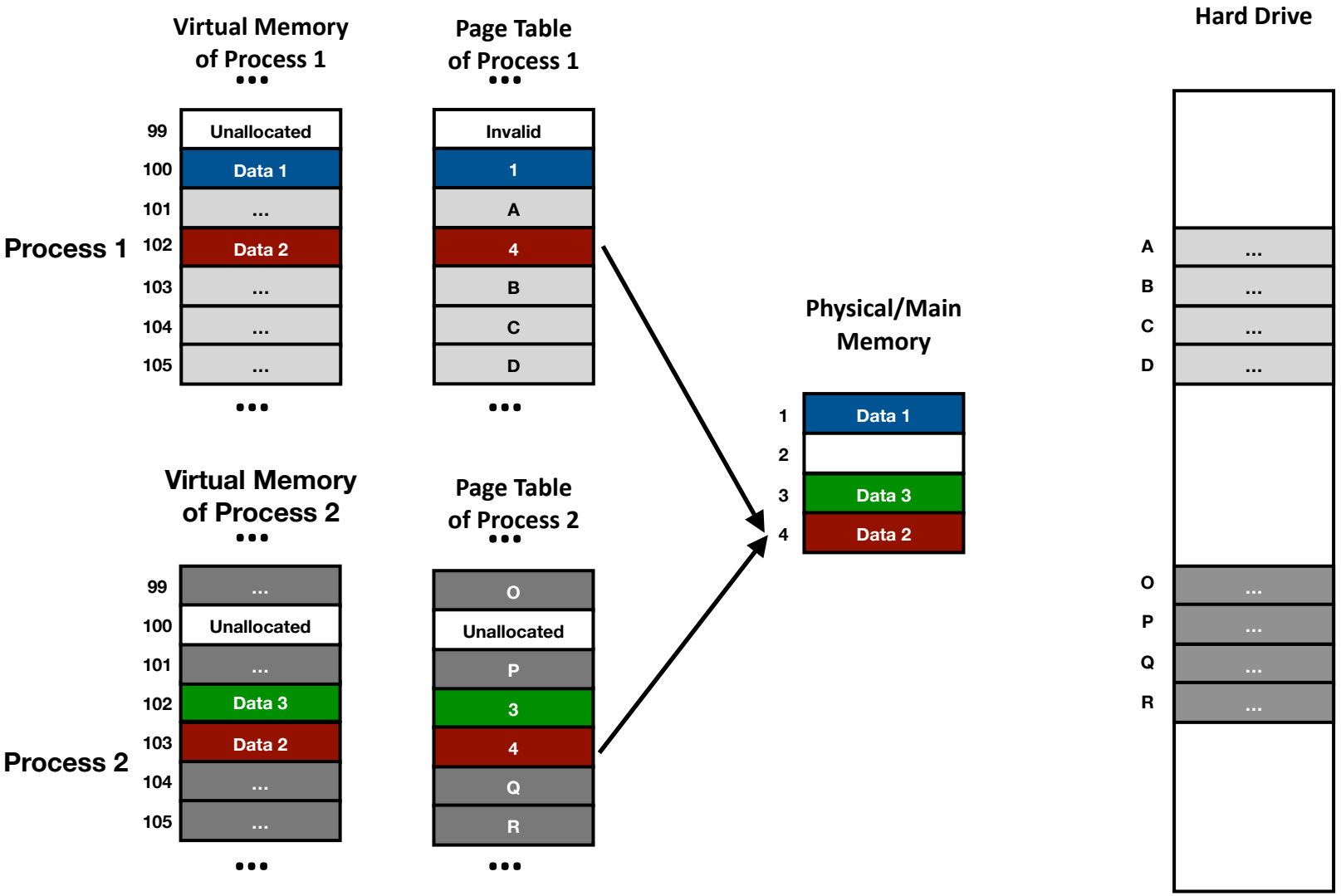
What programmers see



Assuming page size is 4B
Virtual memory size is 32B
Physical memory size is 16B

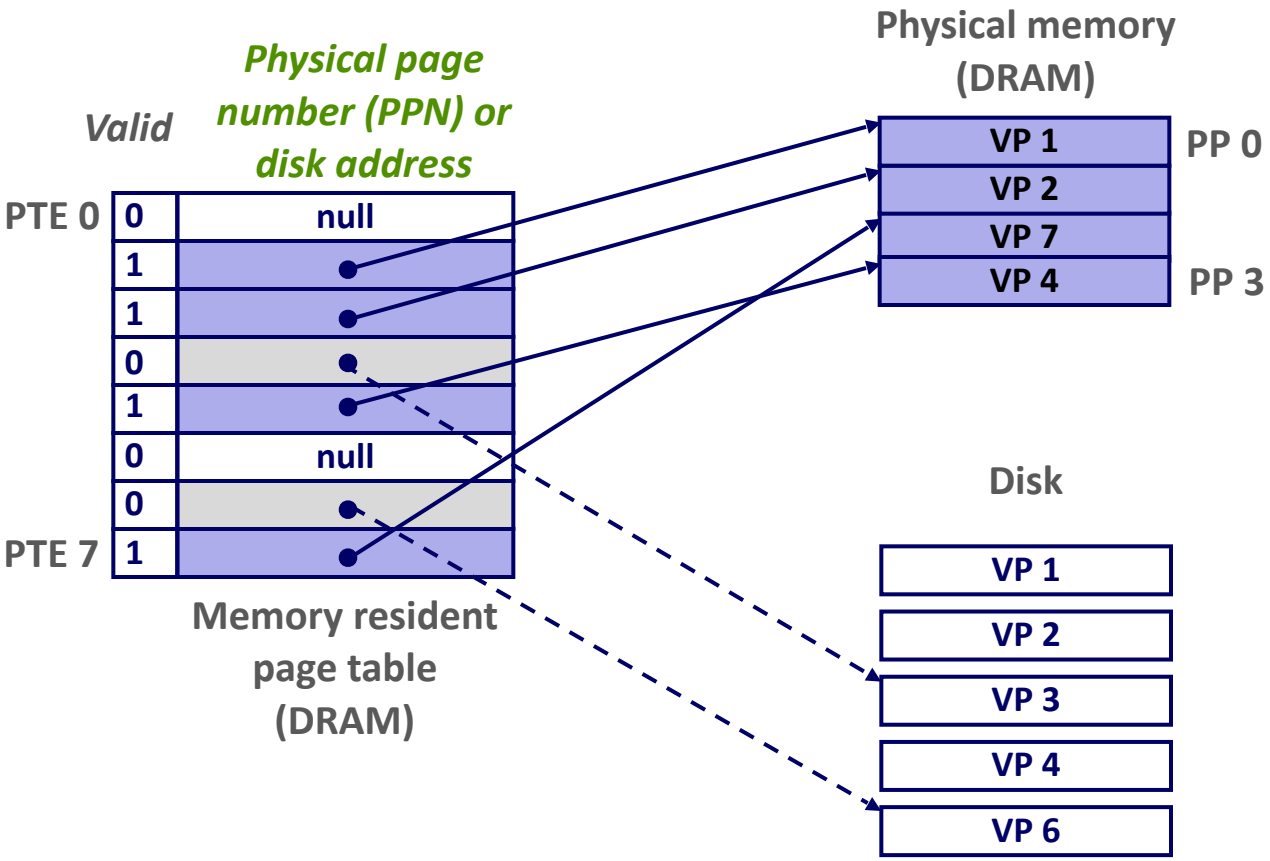


Page Table


Virtual Memory of Process 1
Page Table of Process 1
Process 1
99 Unallocated
100 Data 1
101 ...
102 Data 2
103 ...
104 ...
105 ...
Invalid
1
A
4
B
C
D
Physical/Main Memory
1 Data 1
2
3 Data 3
4 Data 2
Hard Drive
A ...
B ...
C ...
D ...
O ...
P ...
Q ...
R ...
Virtual Memory of Process 2
Page Table of Process 2
Process 2
99 ...
100 Unallocated
101 ...
102 Data 3
103 Data 2
104 ...
105 ...
O
Unallocated
P
3
4
Q
R

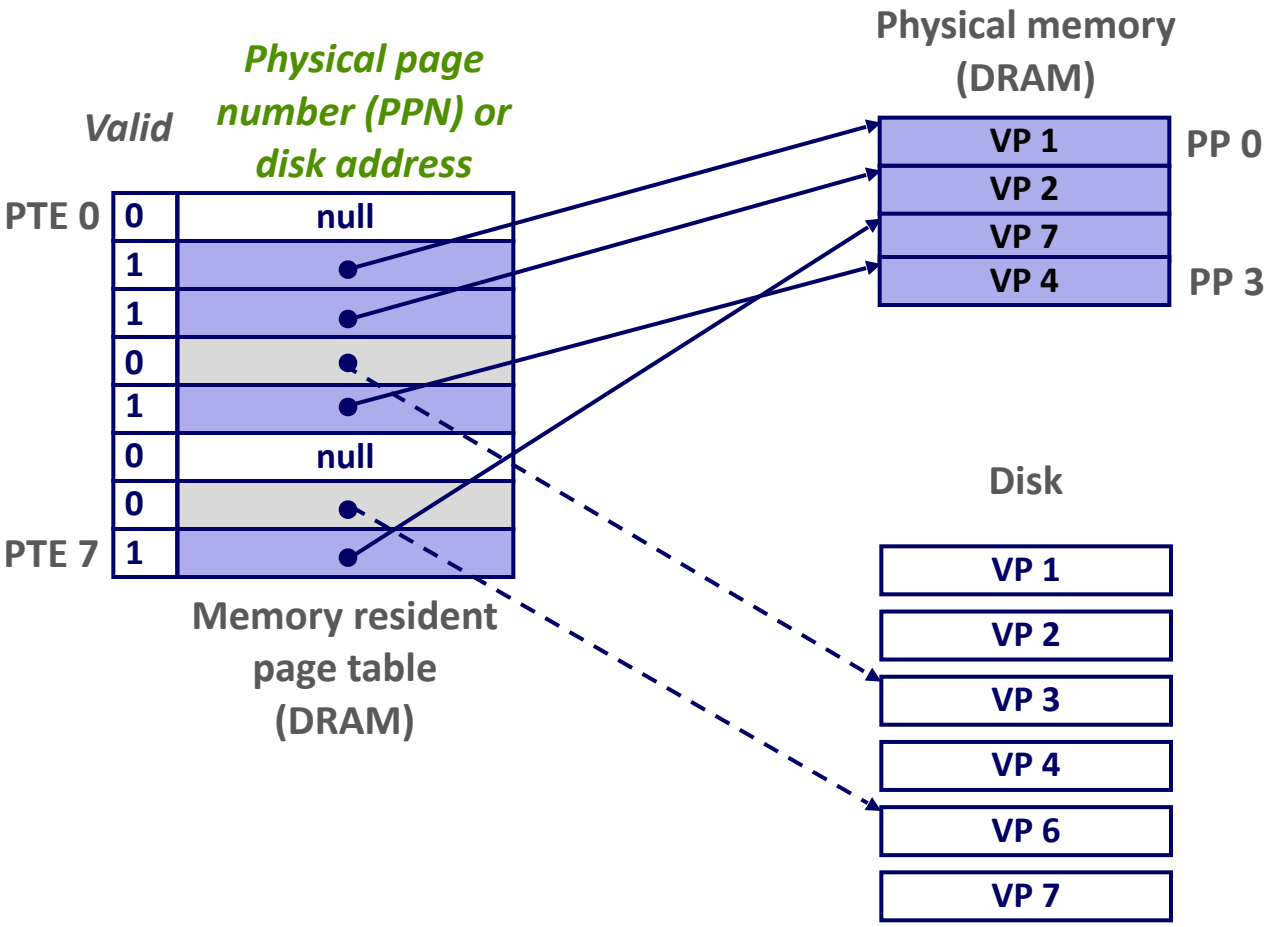
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



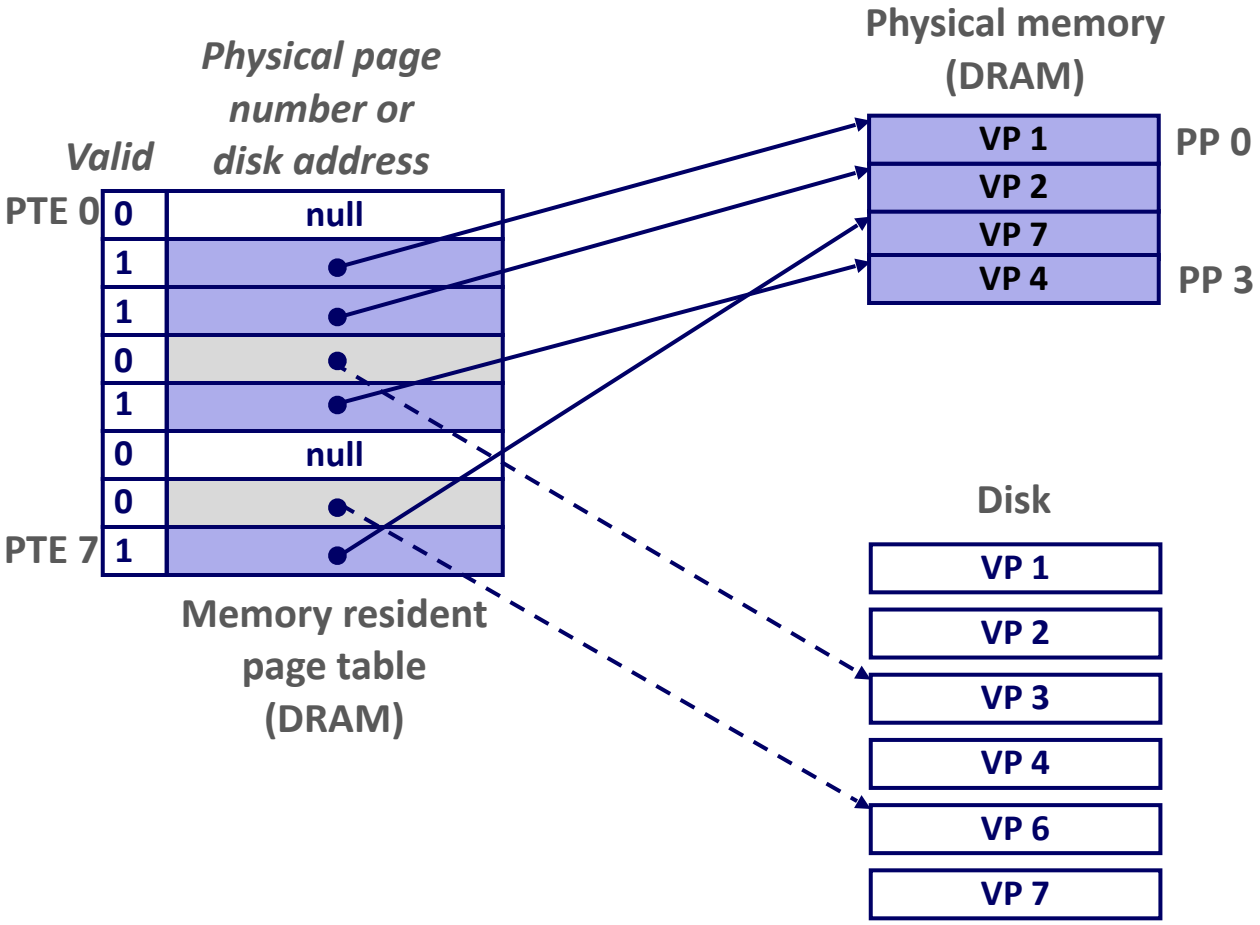
Enabling Data Structure: Page Table

- A page table is an array of page table entries (PTEs) that maps every virtual page to its physical page, i.e., virtual to physical address translation.
- One PTE for each virtual page.



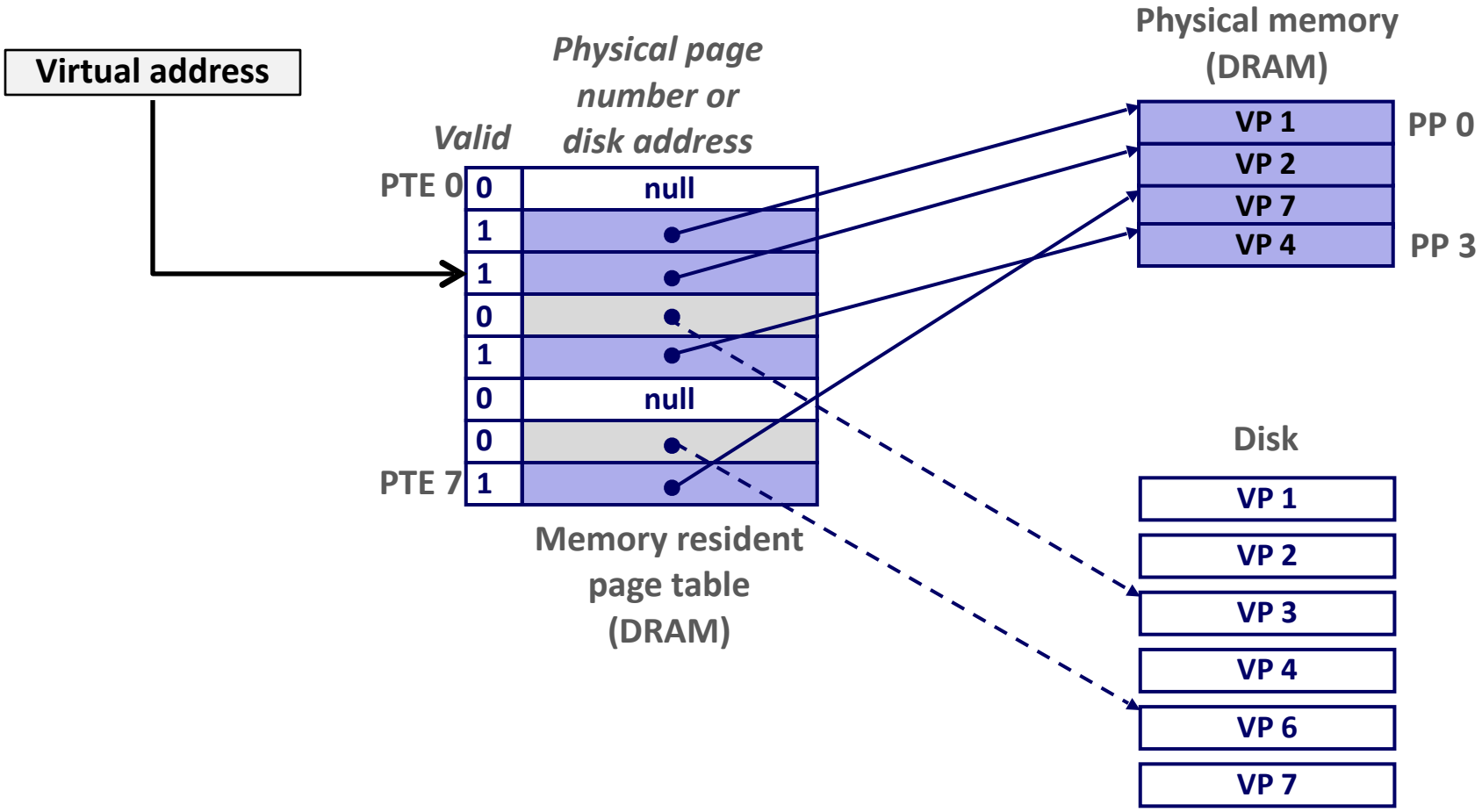
Page Hit

- *Page hit:* reference to VM word that is in physical memory



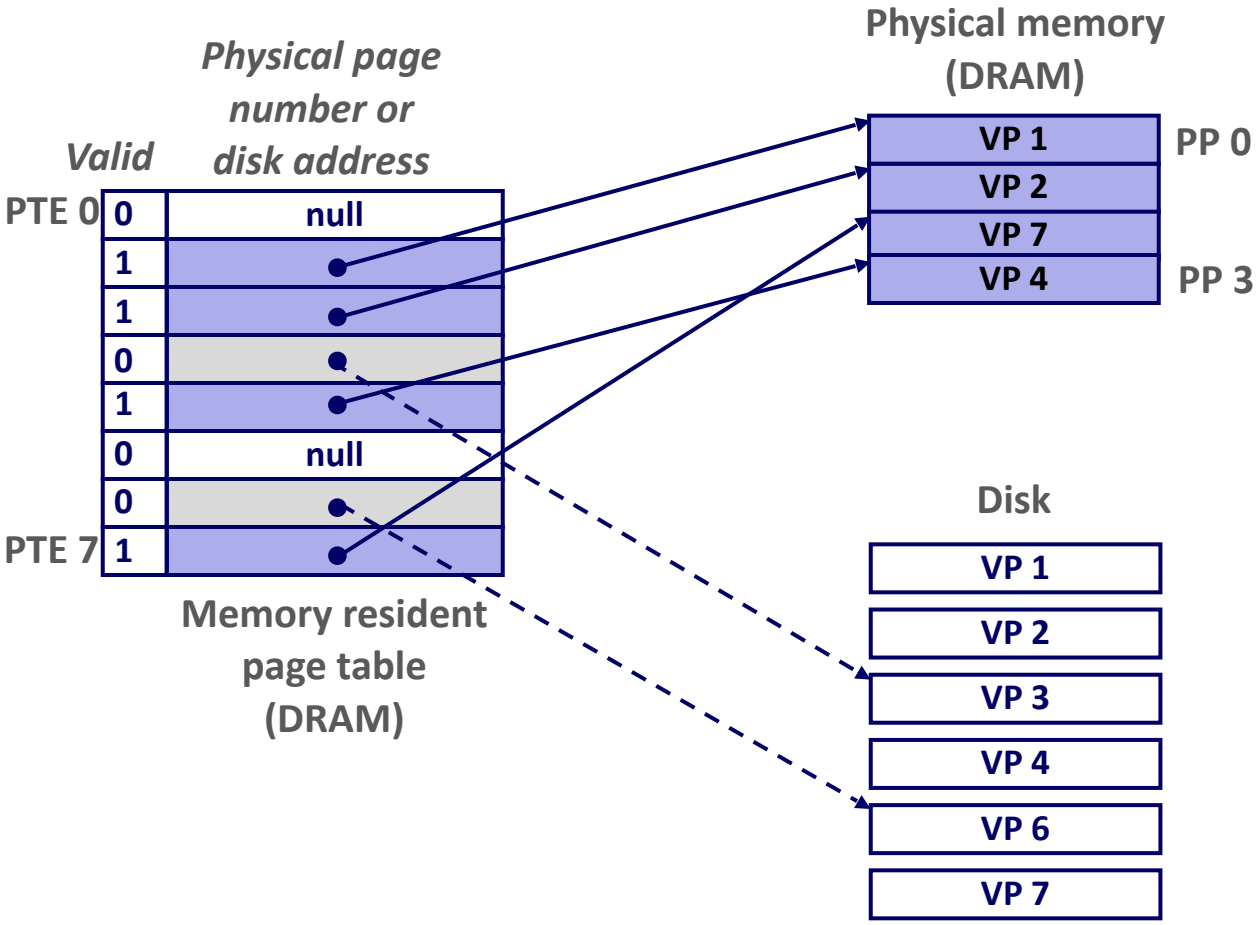
Page Hit

- *Page hit:* reference to VM word that is in physical memory



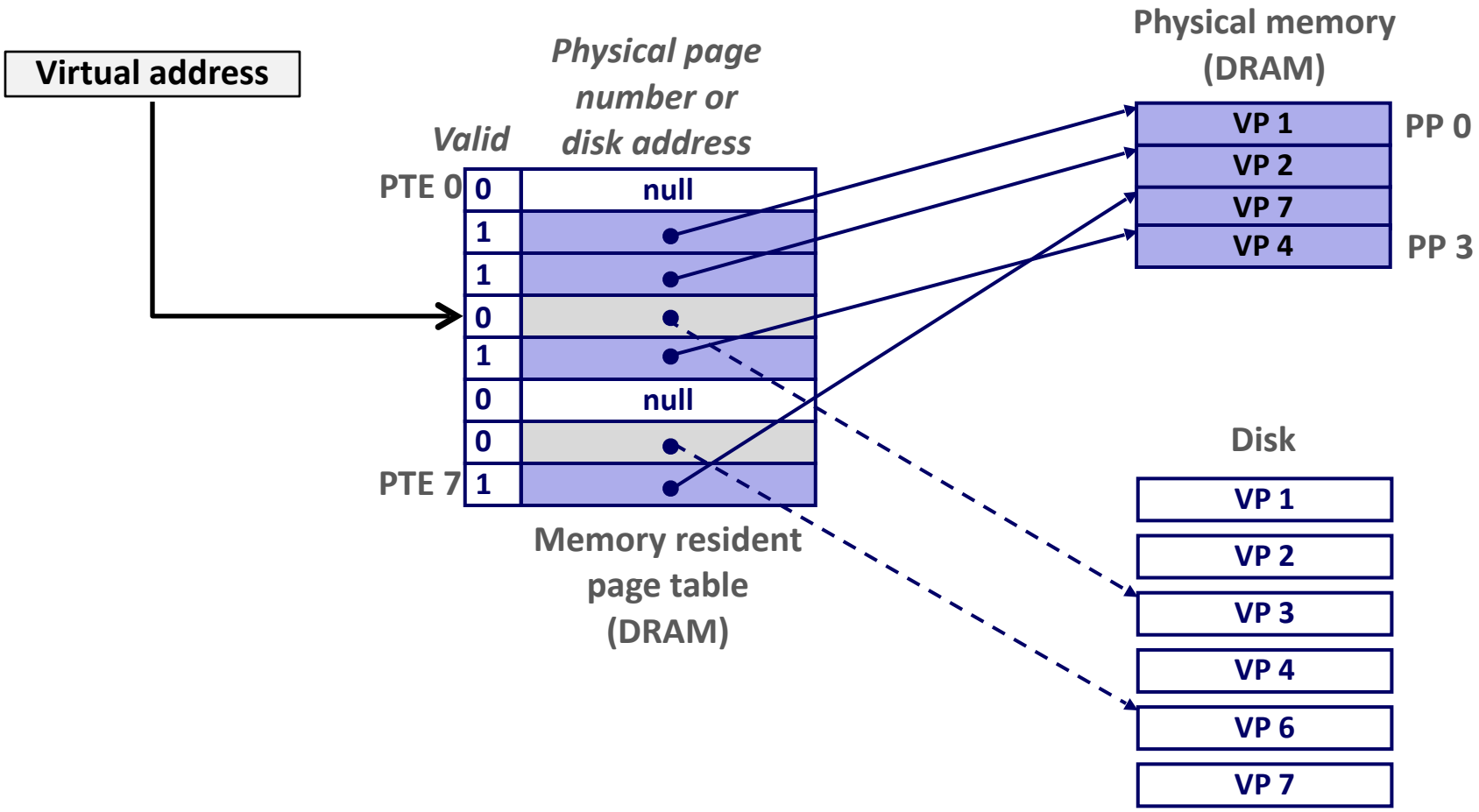
Page Fault

- *Page fault:* reference to VM word that is not in physical memory



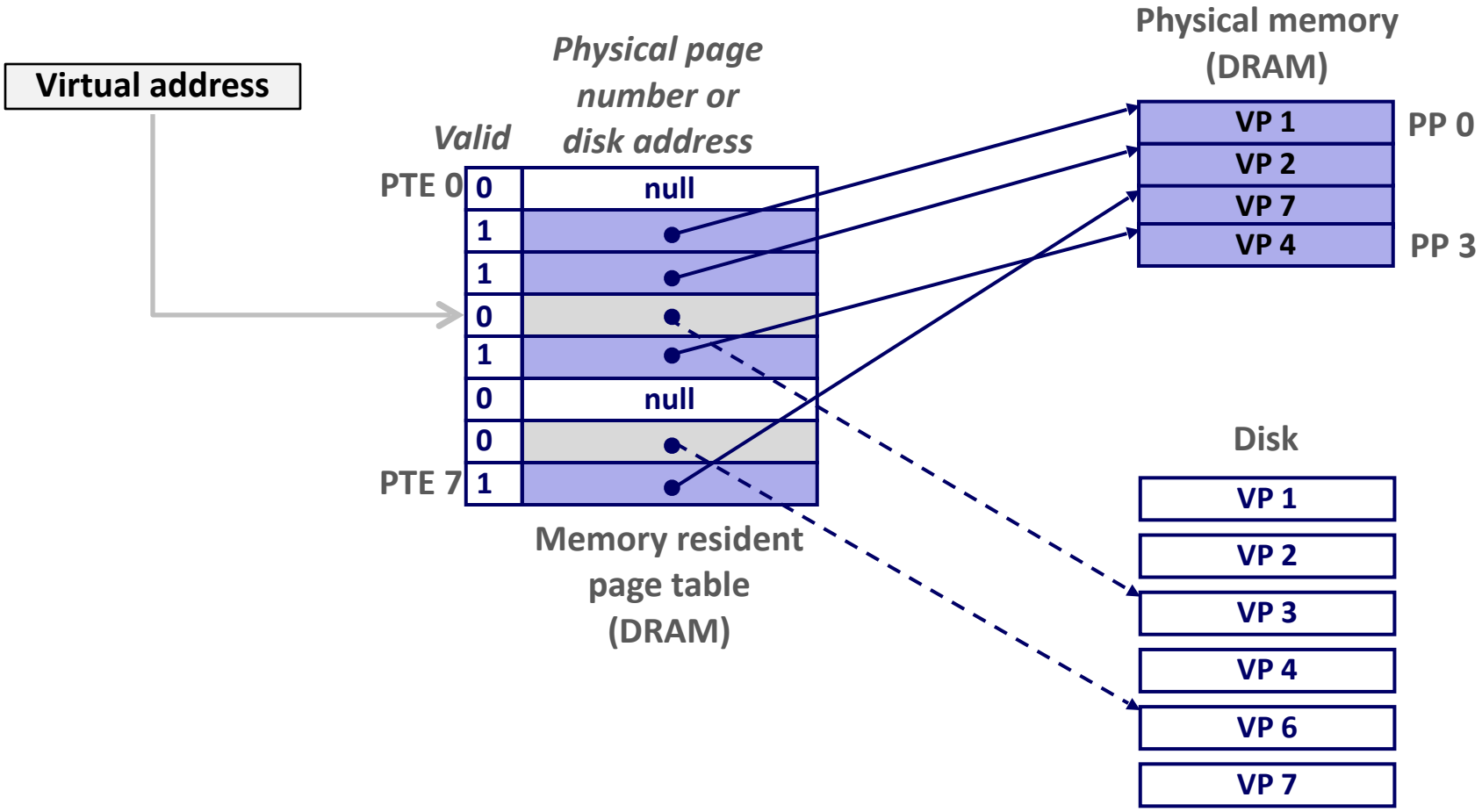
Page Fault

- *Page fault:* reference to VM word that is not in physical memory



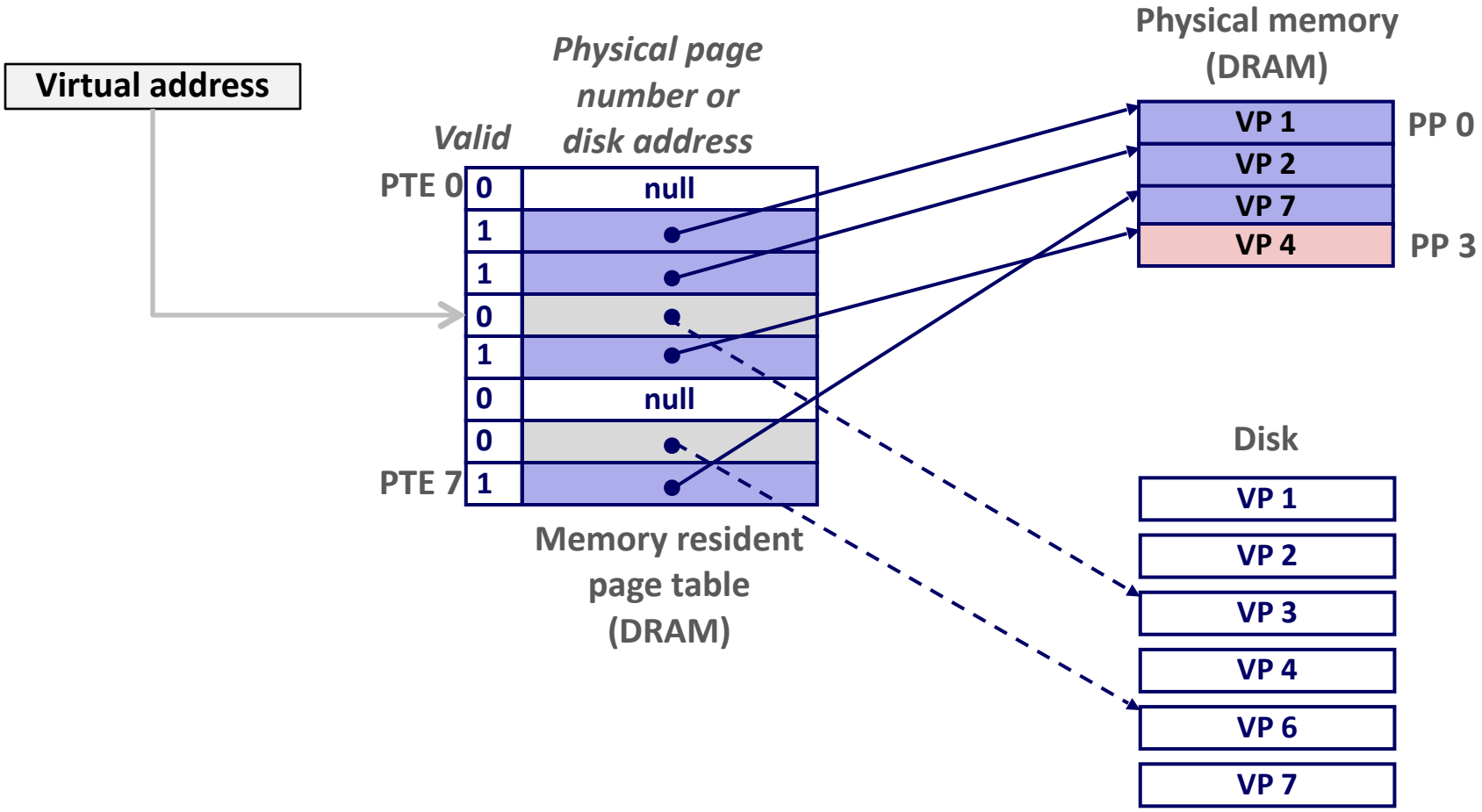
Handling Page Fault

- Page miss causes ***page fault (an exception)***



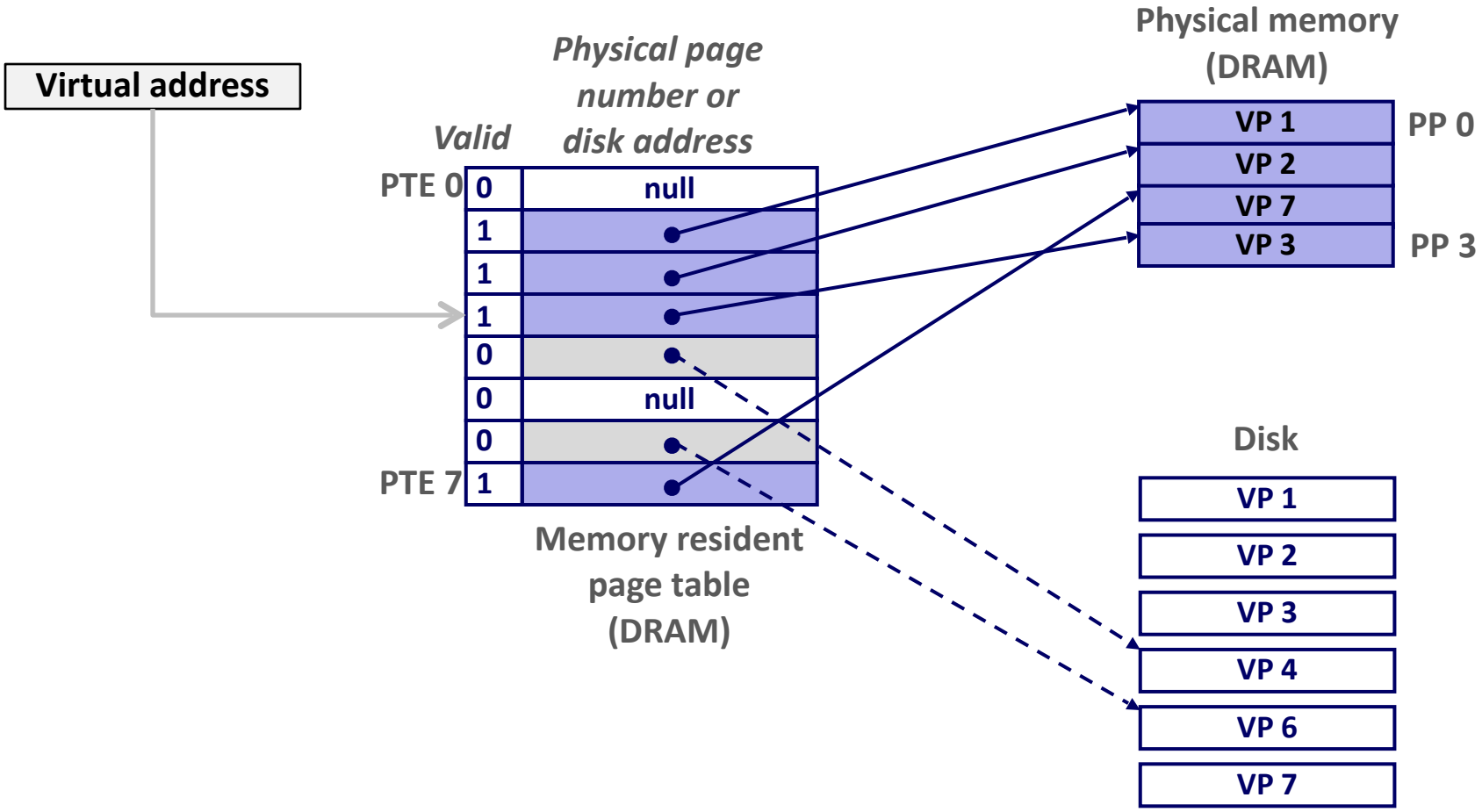
Handling Page Fault

- Page miss causes ***page fault (an exception)***
- Page fault ***handler*** selects a victim to be evicted (here VP 4)



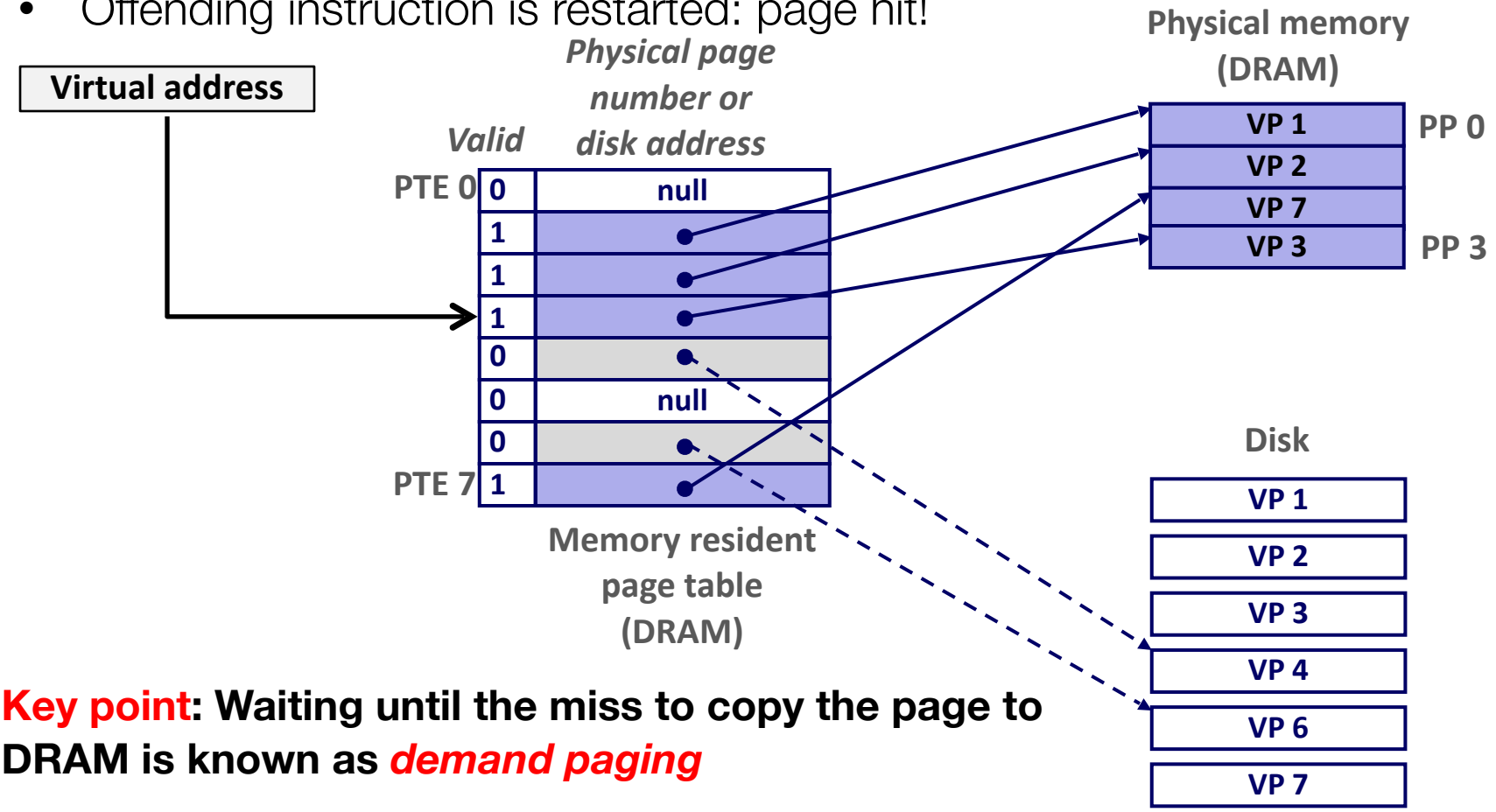
Handling Page Fault

- Page miss causes ***page fault (an exception)***
- Page fault ***handler*** selects a victim to be evicted (here VP 4)



Handling Page Fault

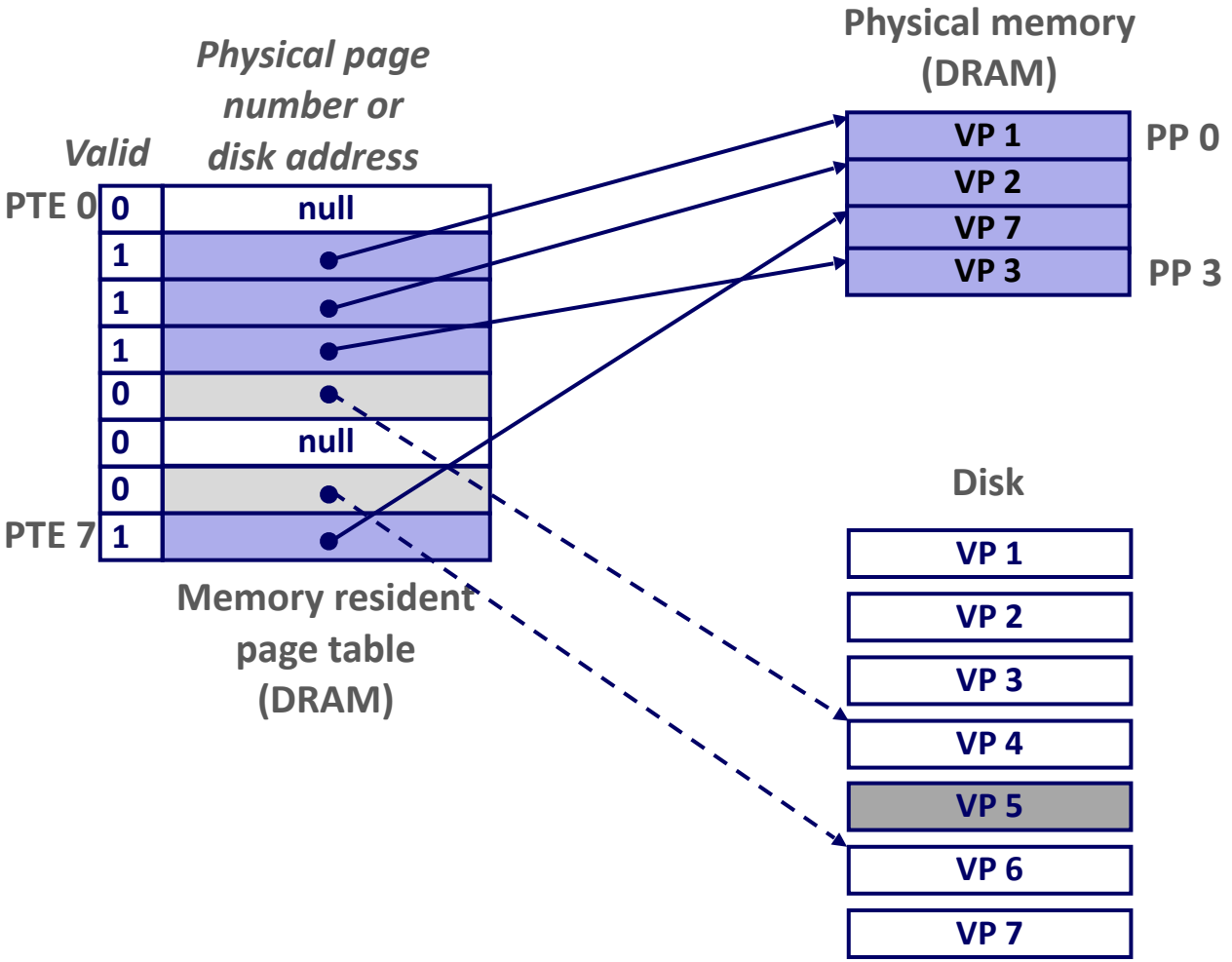
- Page miss causes ***page fault (an exception)***
- Page fault ***handler*** selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!



Key point: Waiting until the miss to copy the page to DRAM is known as *demand paging*

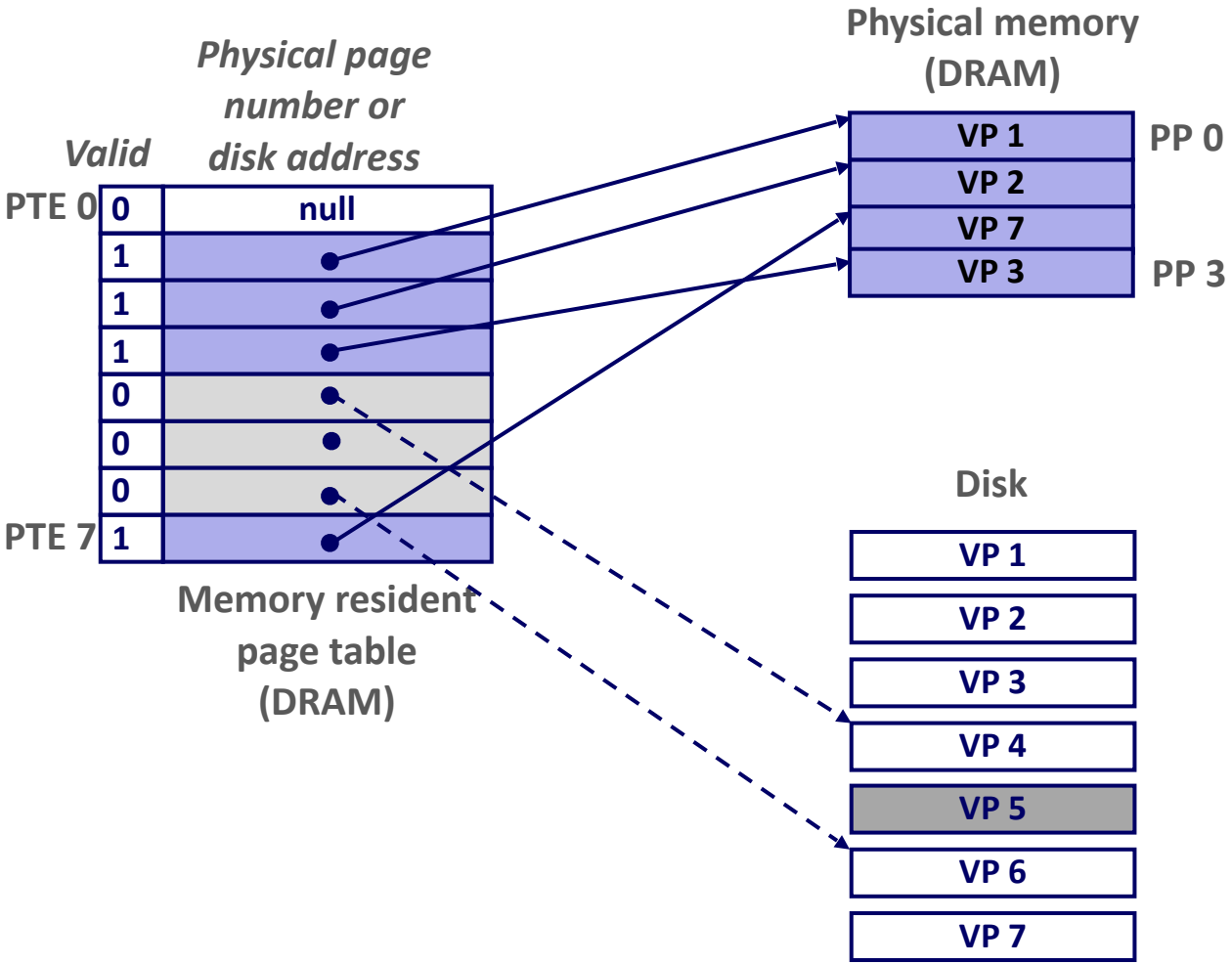
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



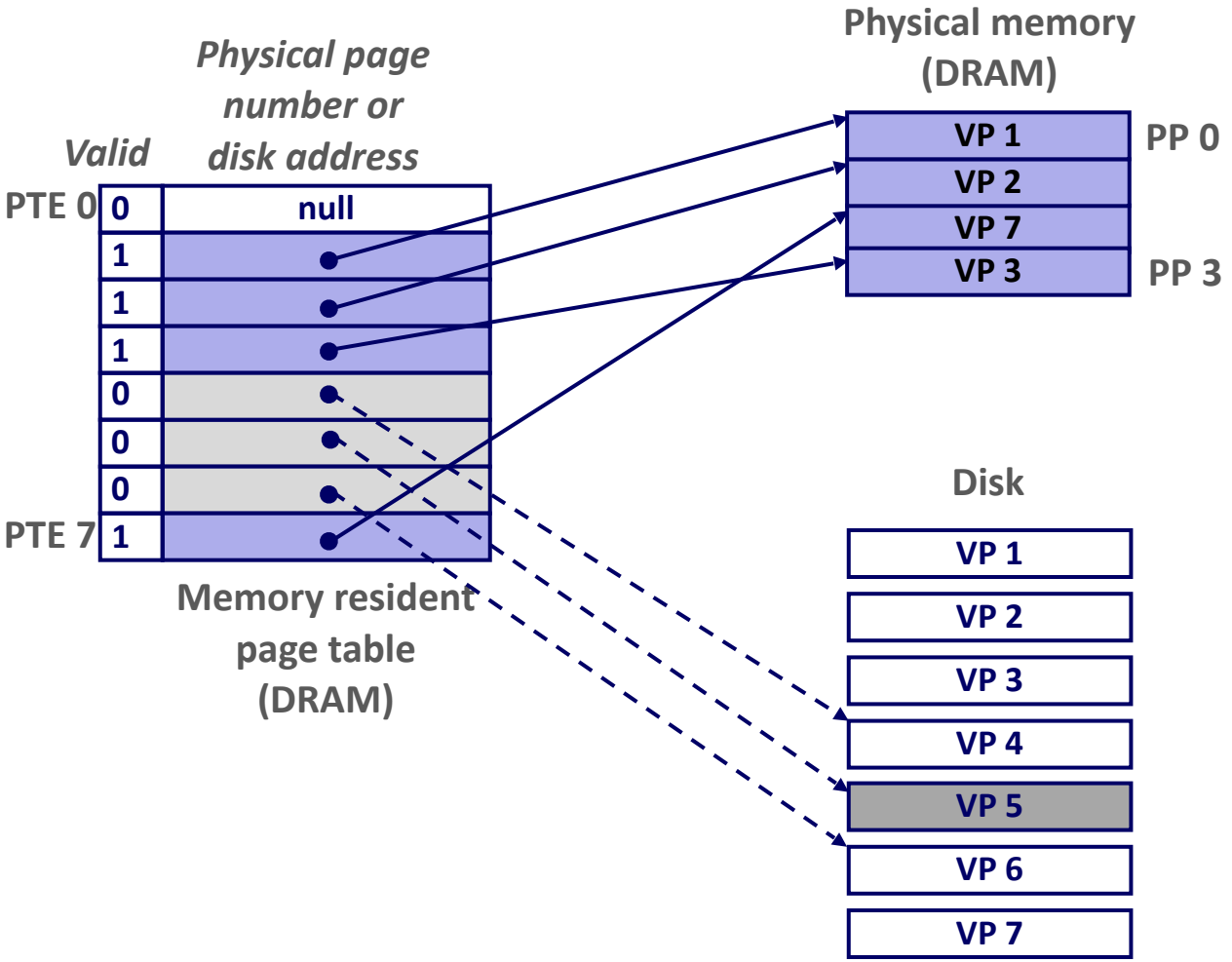
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



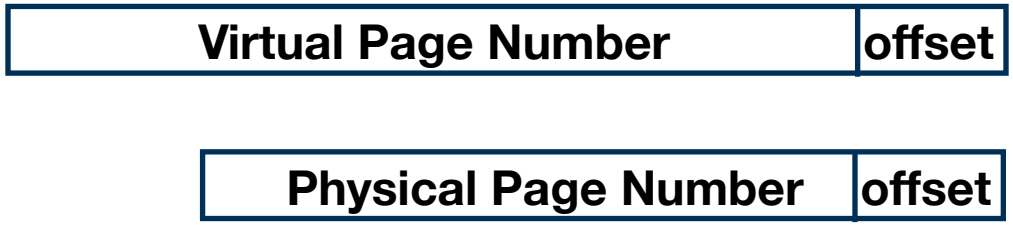
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.



Calculate Bits in VA and PA

- In a 64-bit machine, VA is 64-bit long. Assuming PM is 4 GB. Assuming 4 KB page size.
- How many bits for page offset?
  - 12. Same for VM and PM
- How many bits for Virtual Page Number?
  - 52, i.e., 2^{52} virtual pages
- How many bits for Physical Page Number?
  - 20, i.e., 2^{20} physical pages

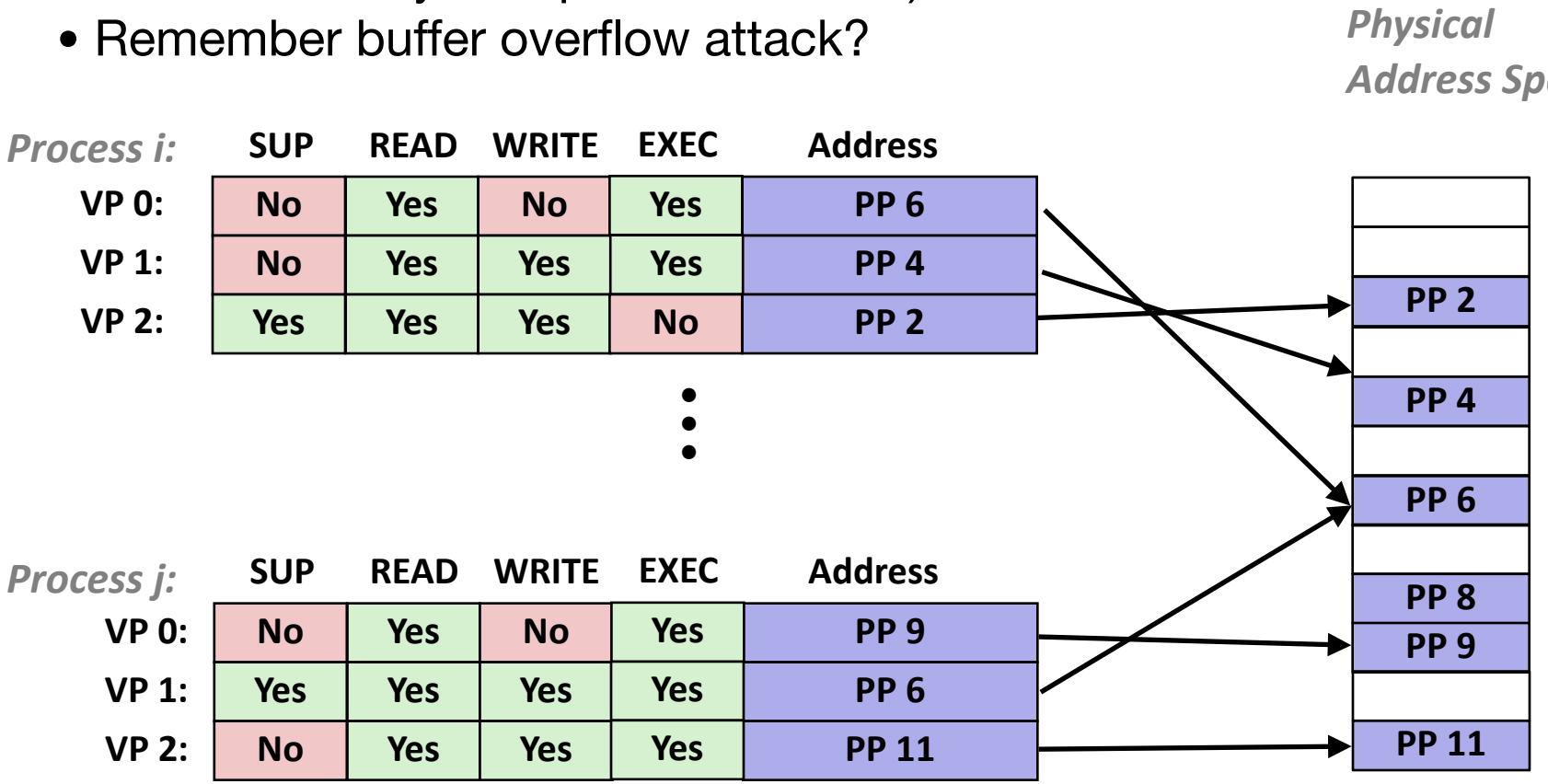


Calculate the Page Table Size

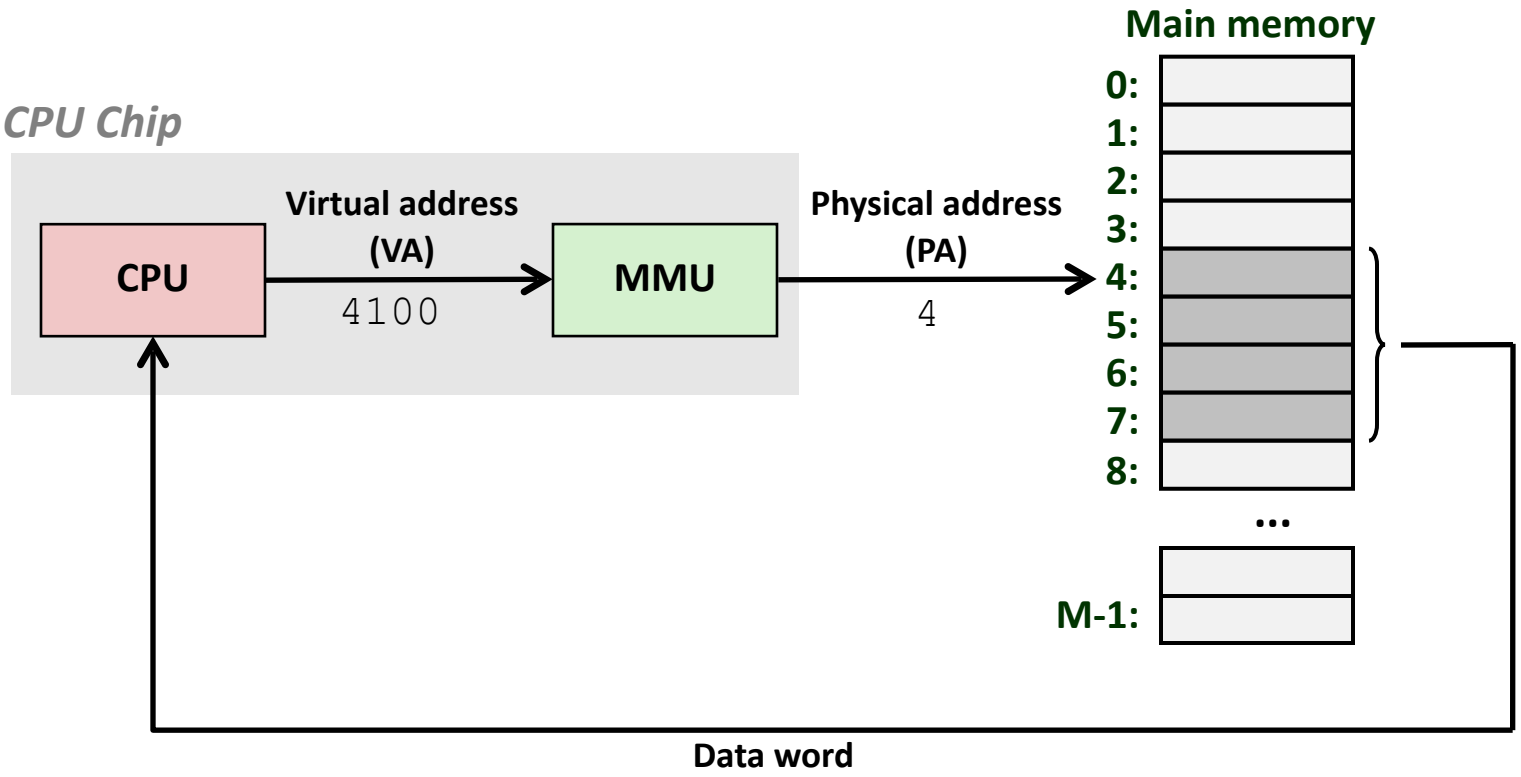
- Assume 4KB page, 4GB virtual memory, each PTE is 8 Bytes
  - 4GB/4KB = 1M virtual pages
  - 1M PTEs in a page table
  - 8MB total size per page table

VM Provides Further Protection Opportunities

- Extend PTEs with permission bits
- MMU checks these bits on each access (read/write/executable/ accessible only in supervisor mode?)
- Remember buffer overflow attack?

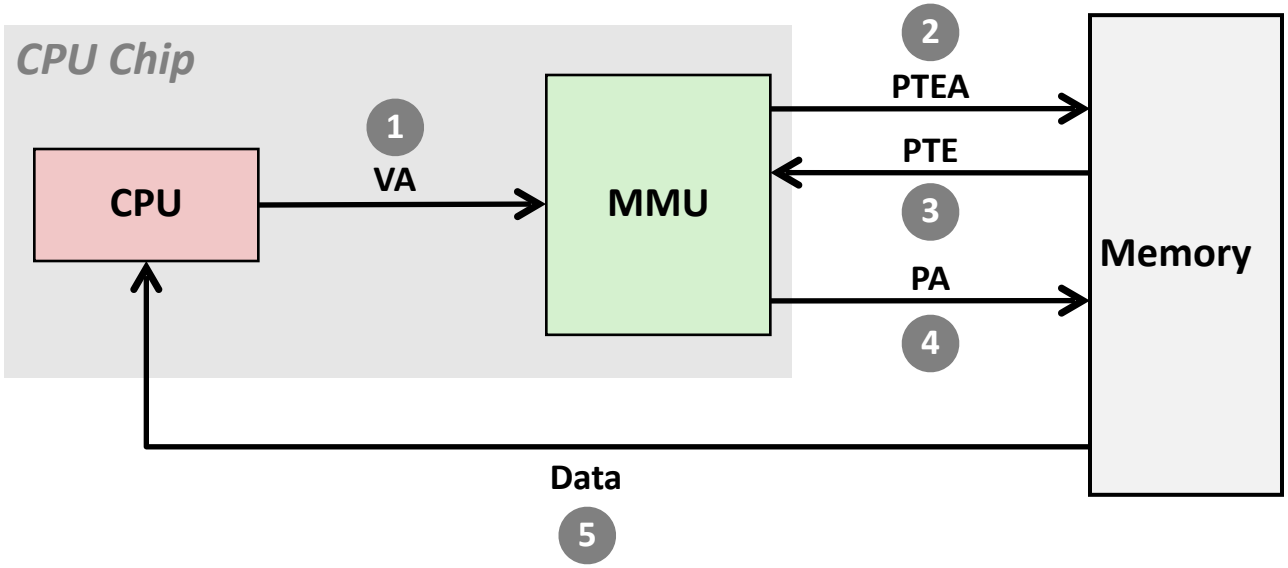


A System Using Virtual Memory



- The memory management unit (MMU) does the VA to PA translation, and moves data between physical memory and disk.

Address Translation: Page Hit



1) Processor sends virtual address to MMU

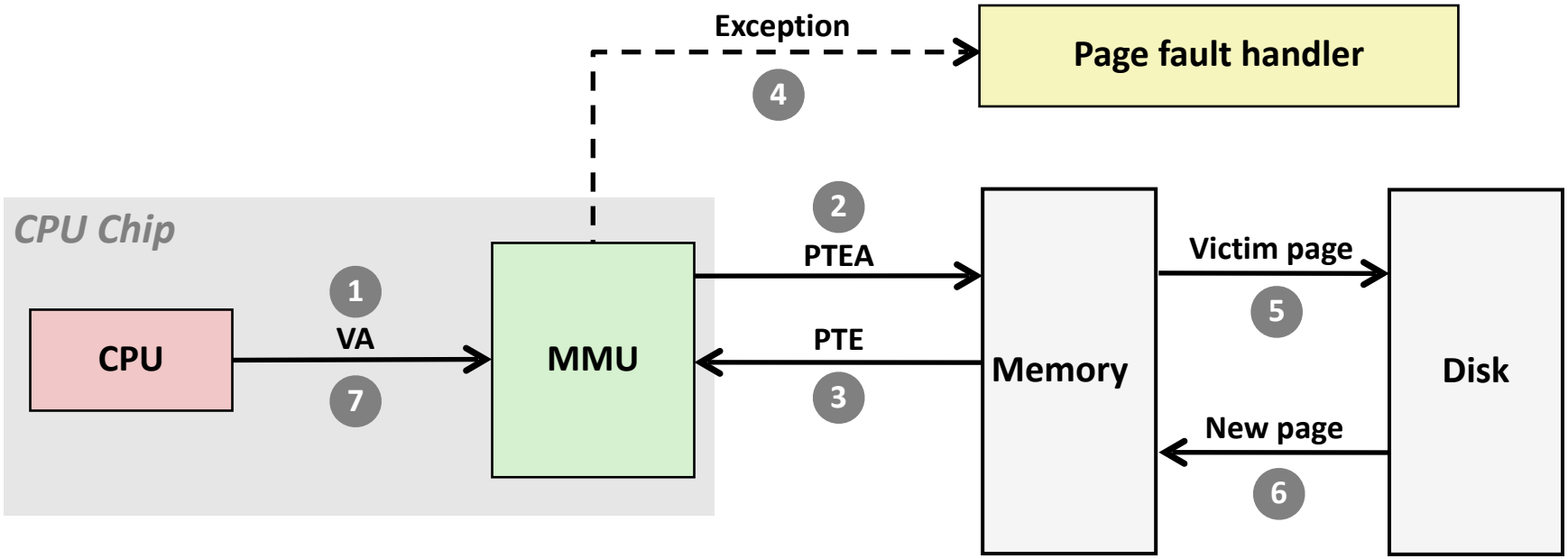
2-3) MMU fetches PTE from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Address Translation: Page Fault



1) Processor sends virtual address to MMU

2-3) MMU fetches PTE from page table in memory

4) Valid bit is zero, so MMU triggers page fault exception

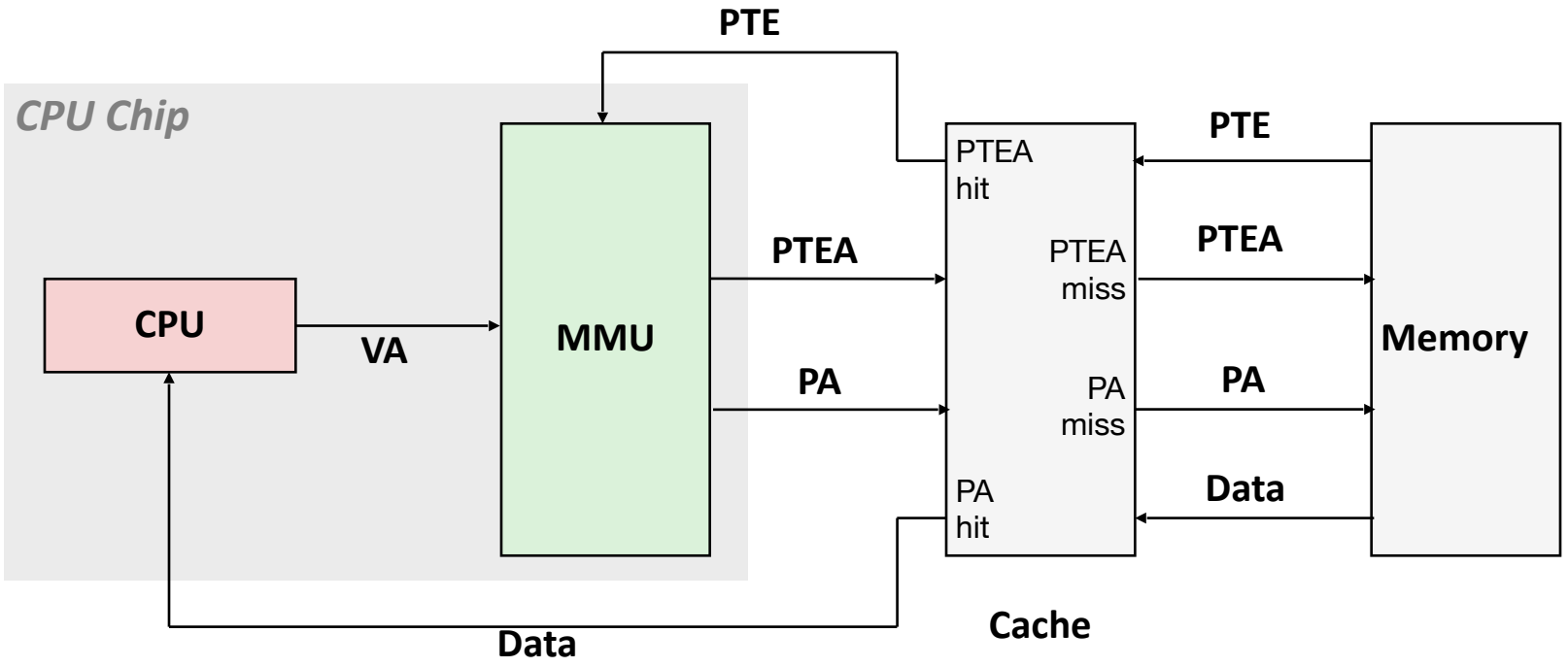
5) Handler identifies victim (and, if dirty, pages it out to disk)

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Integrating VM and Cache



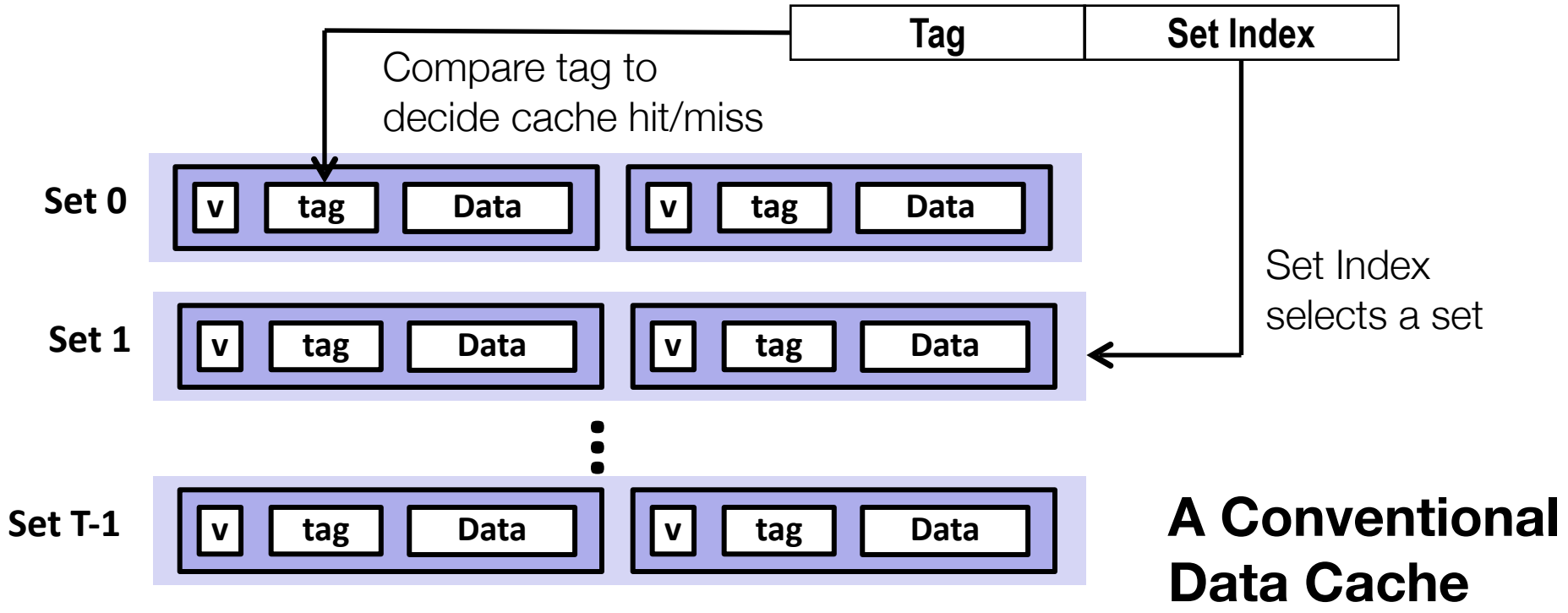
VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

Speeding up Address Translation

- Problem: Every memory load/store requires two memory accesses: one for PTE, another for real
  - The PTE access is kind of an overhead
  - Can we speed it up?
- Page table entries (PTEs) are already cached in cache like any other memory data. But:
  - PTEs may be evicted by other data references
  - PTE hit still requires a small cache delay

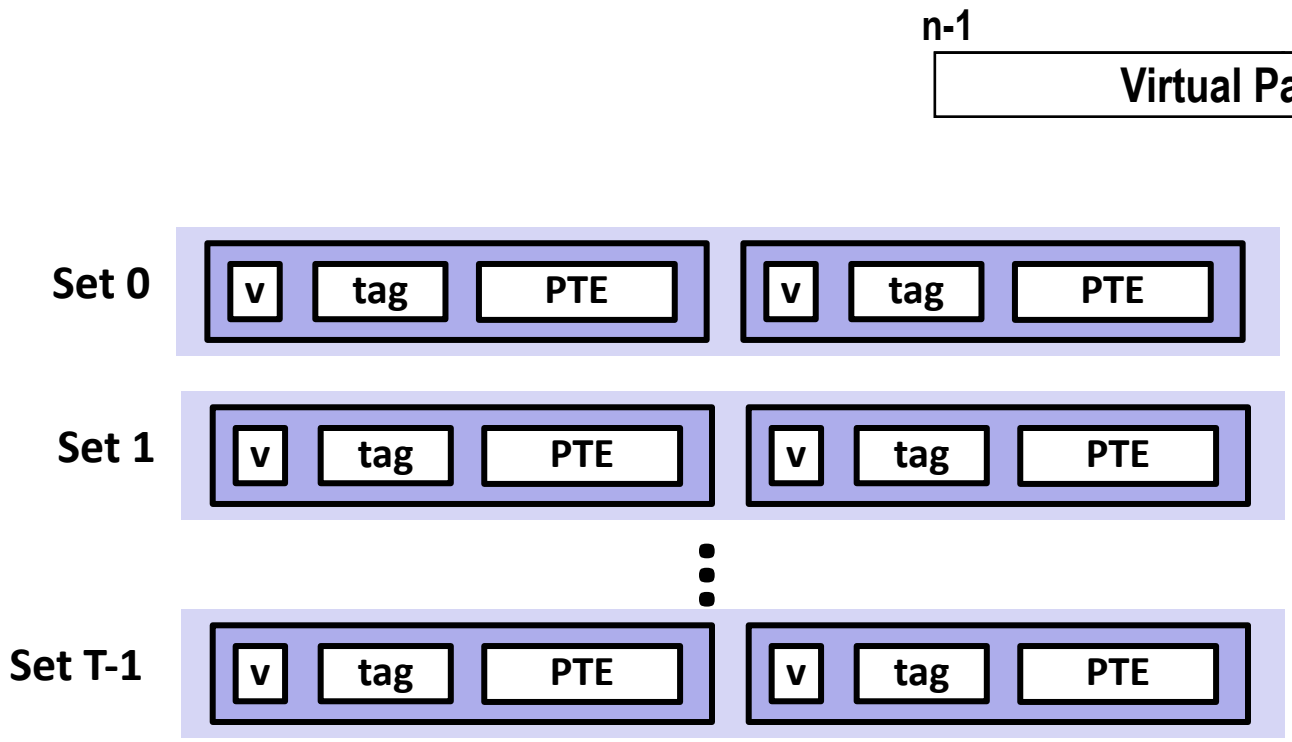
Speeding up Translation with a TLB

- Solution: *Translation Lookaside Buffer* (TLB)
  - Think of it as a dedicated cache for page table
  - Small set-associative hardware cache in MMU
  - Contains complete page table entries for a small number of pages



Accessing the TLB

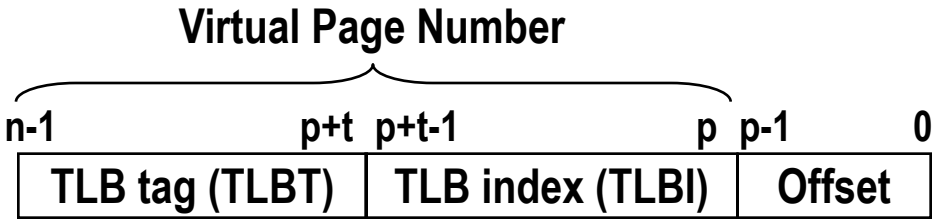
- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

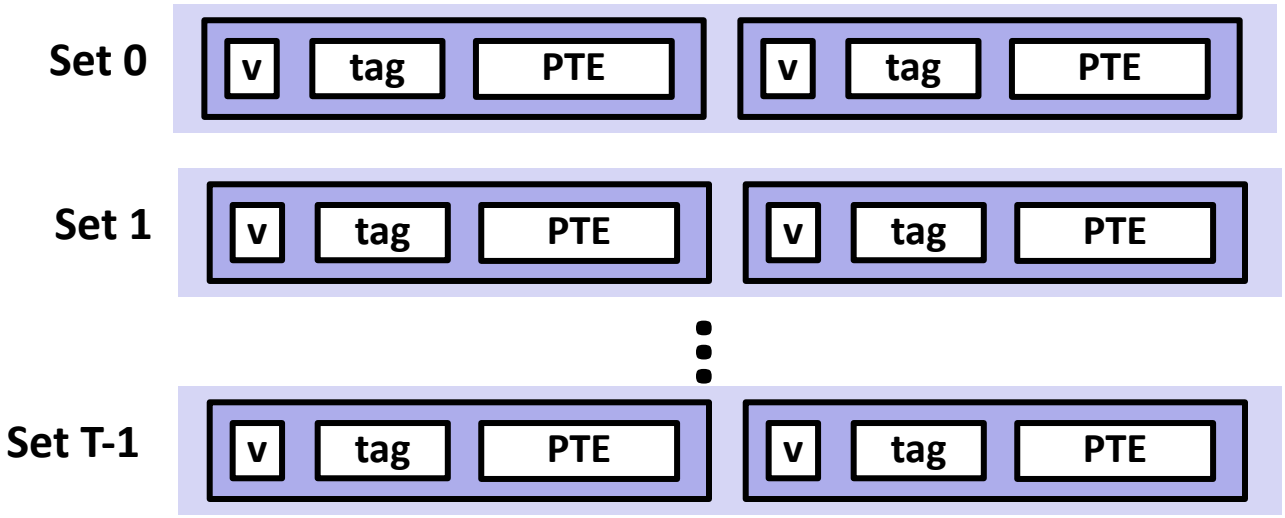


A Page Table Cache

Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

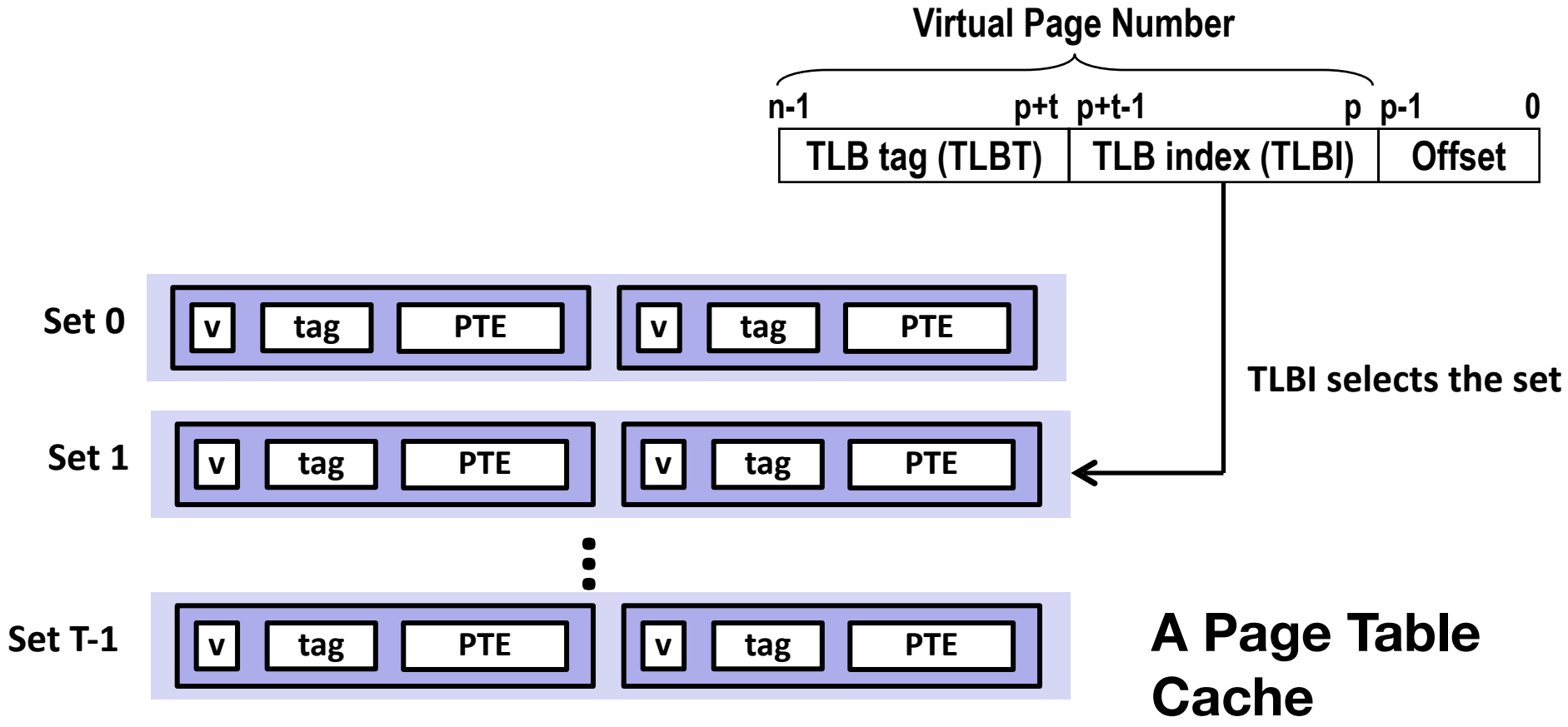




A Page Table Cache

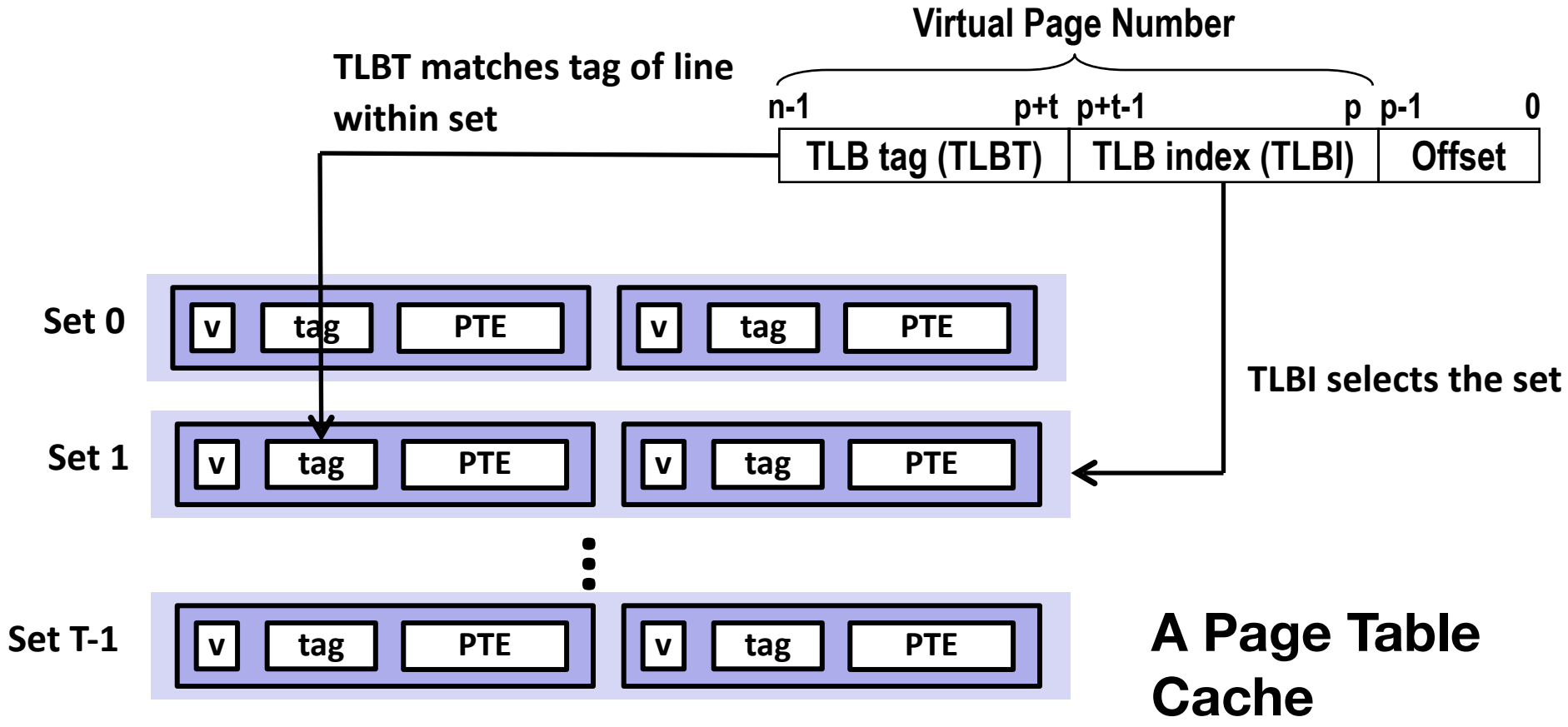
Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

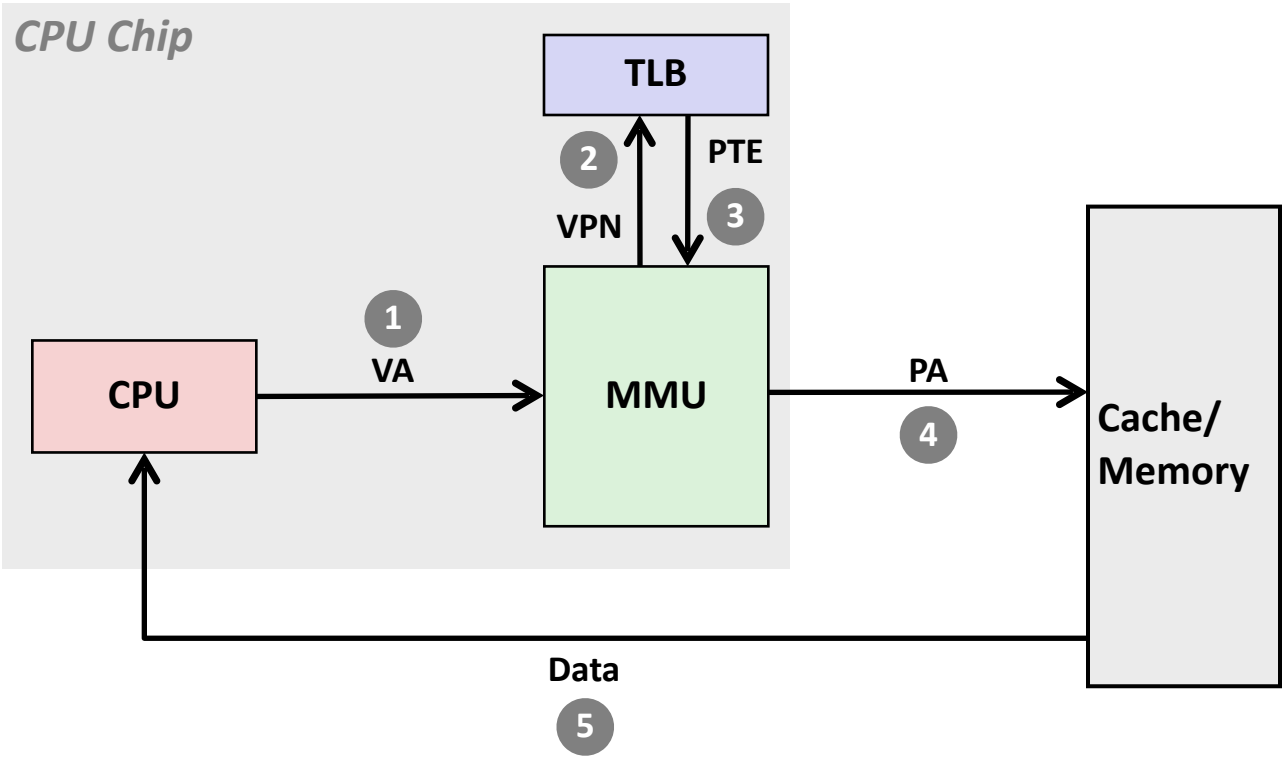


Accessing the TLB

- MMU uses the Virtual Page Number portion of the virtual address to access the TLB:

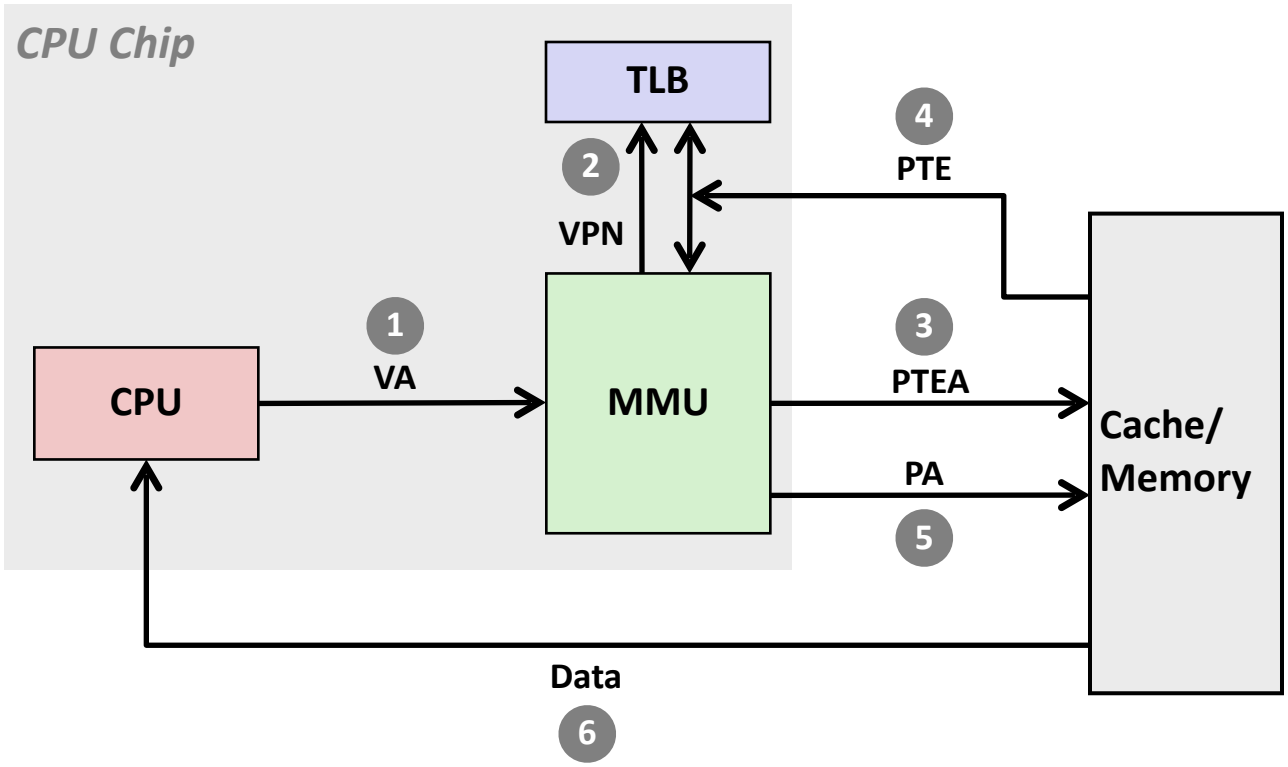


TLB Hit



A TLB hit eliminates a memory access

TLB Miss

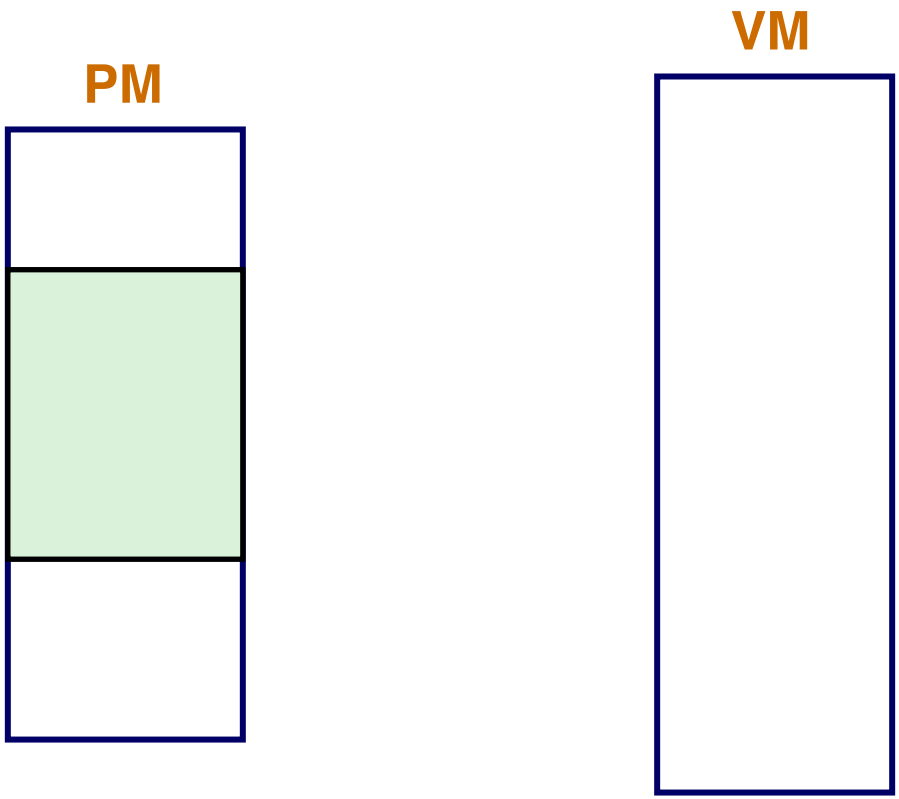

CPU Chip
TLB
4
PTE
2
VPN
1
VA
3
PTEA
CPU
MMU
Cache/
Memory
PA
5
Data
6

Where Does Page Table Live?

- It needs to be at a specific location where we can find it
  - In main memory, with its start address stored in a special register (PTBR)
- Assume 4KB page, 48-bit virtual memory, each PTE is 8 Bytes
  - 2^{36} PTEs in a page table
  - 512 GB total size per page table??!!
- Problem: Page tables are huge
  - One table per process!
  - Storing them all in main memory wastes space

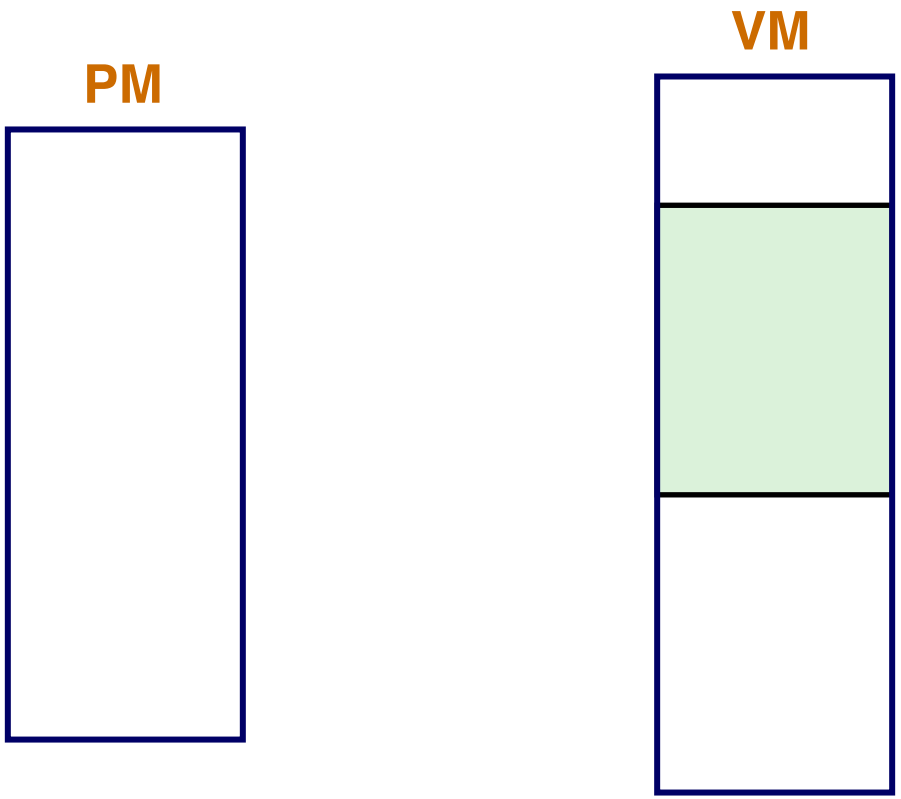
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



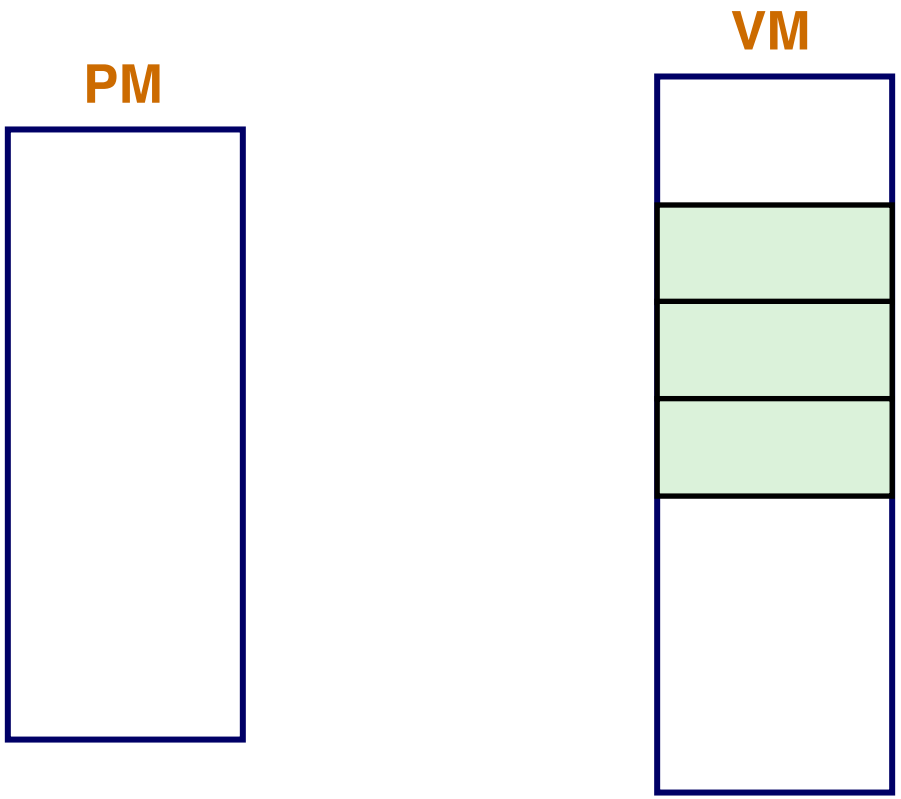
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



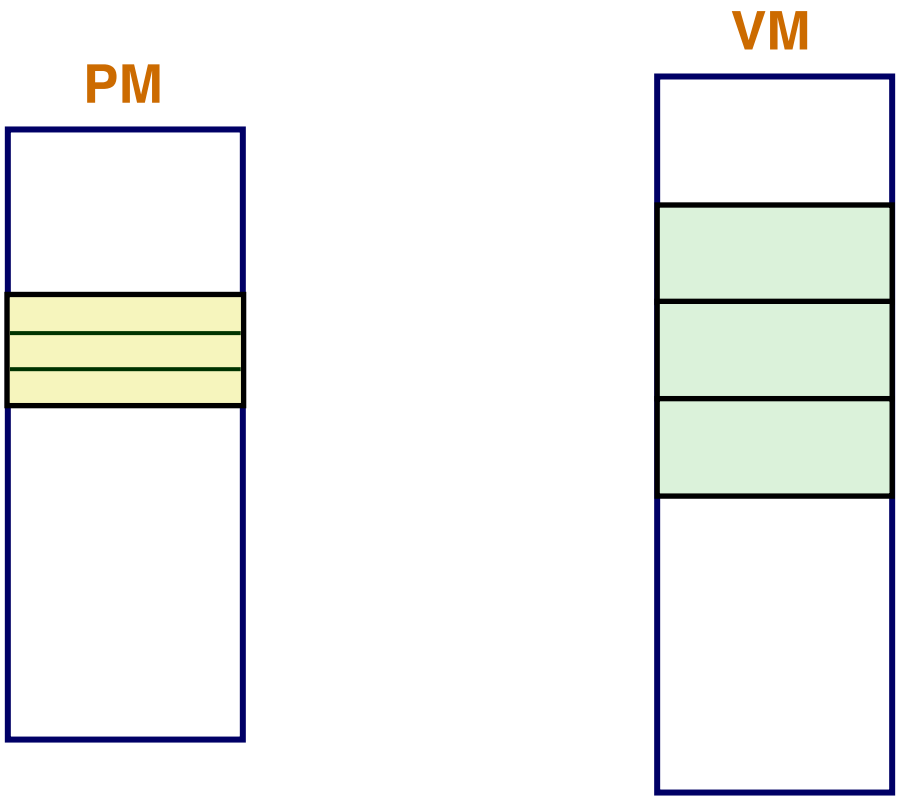
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



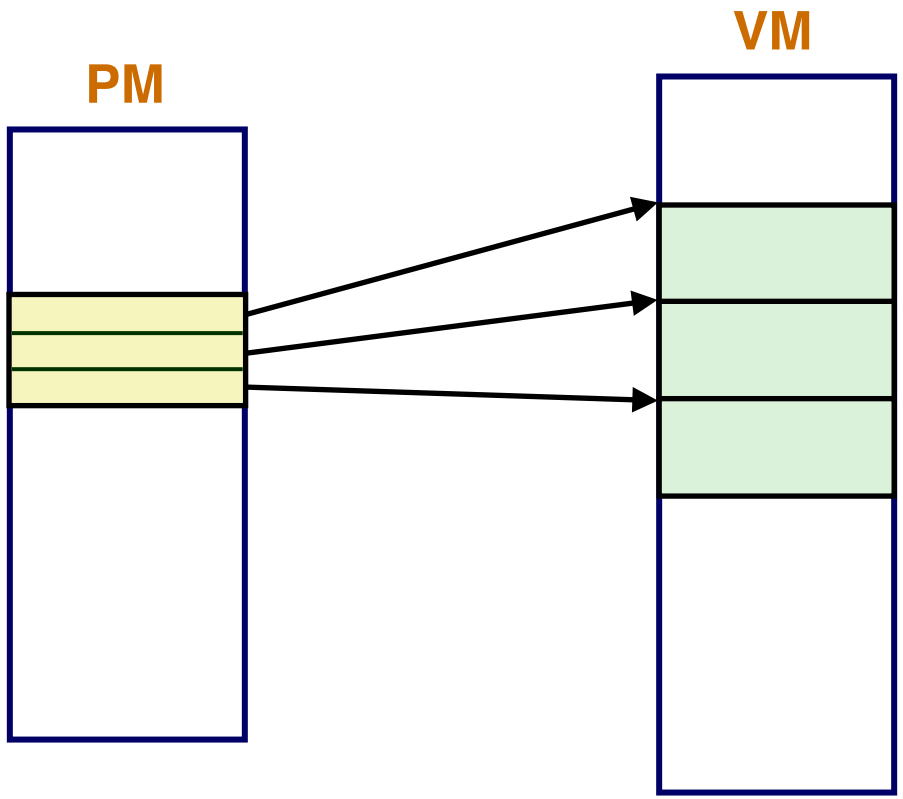
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



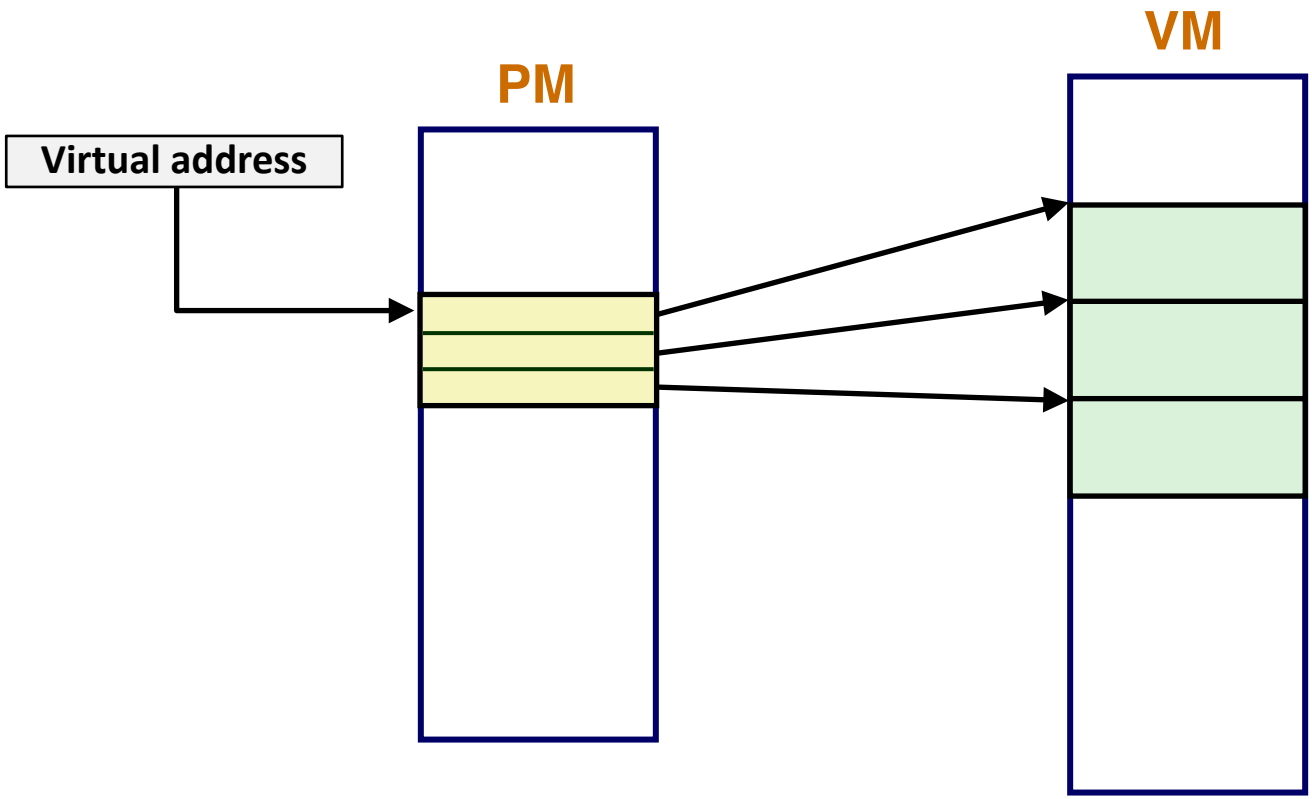
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



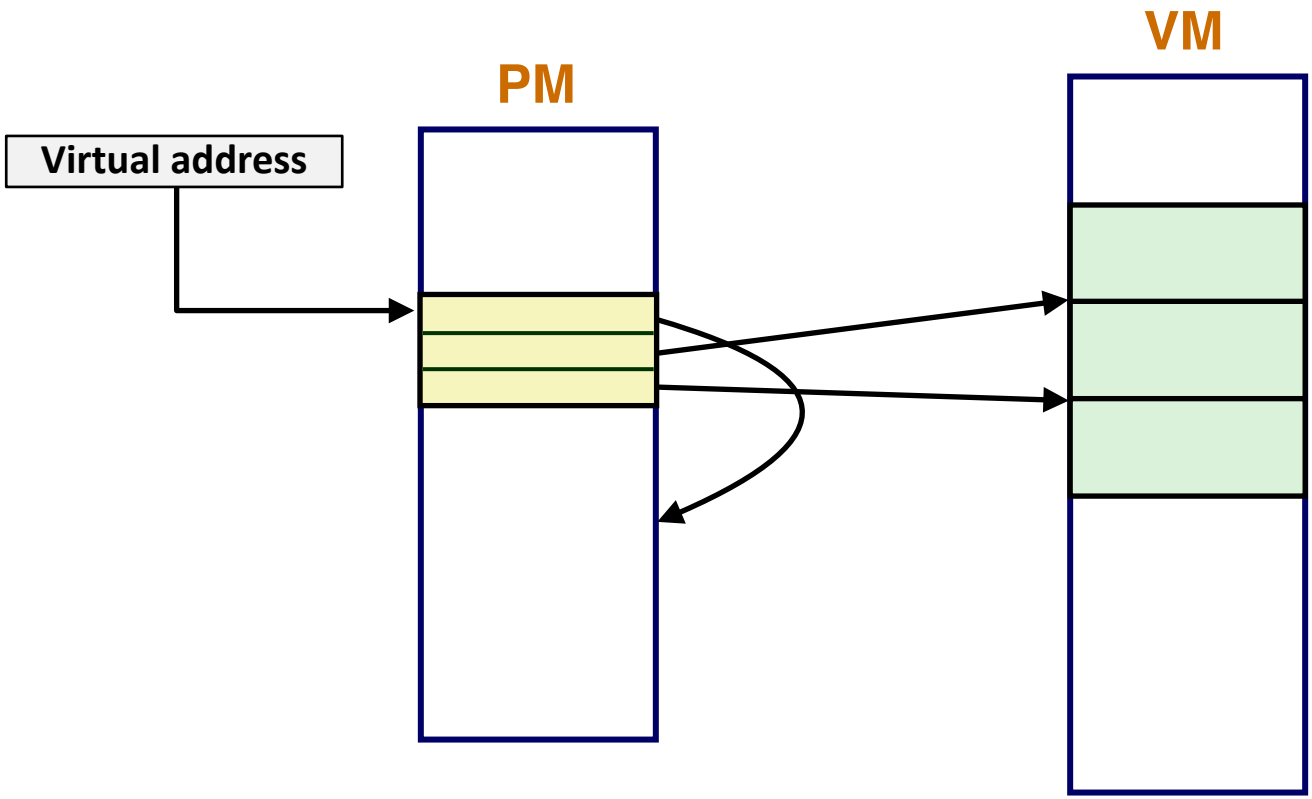
Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data

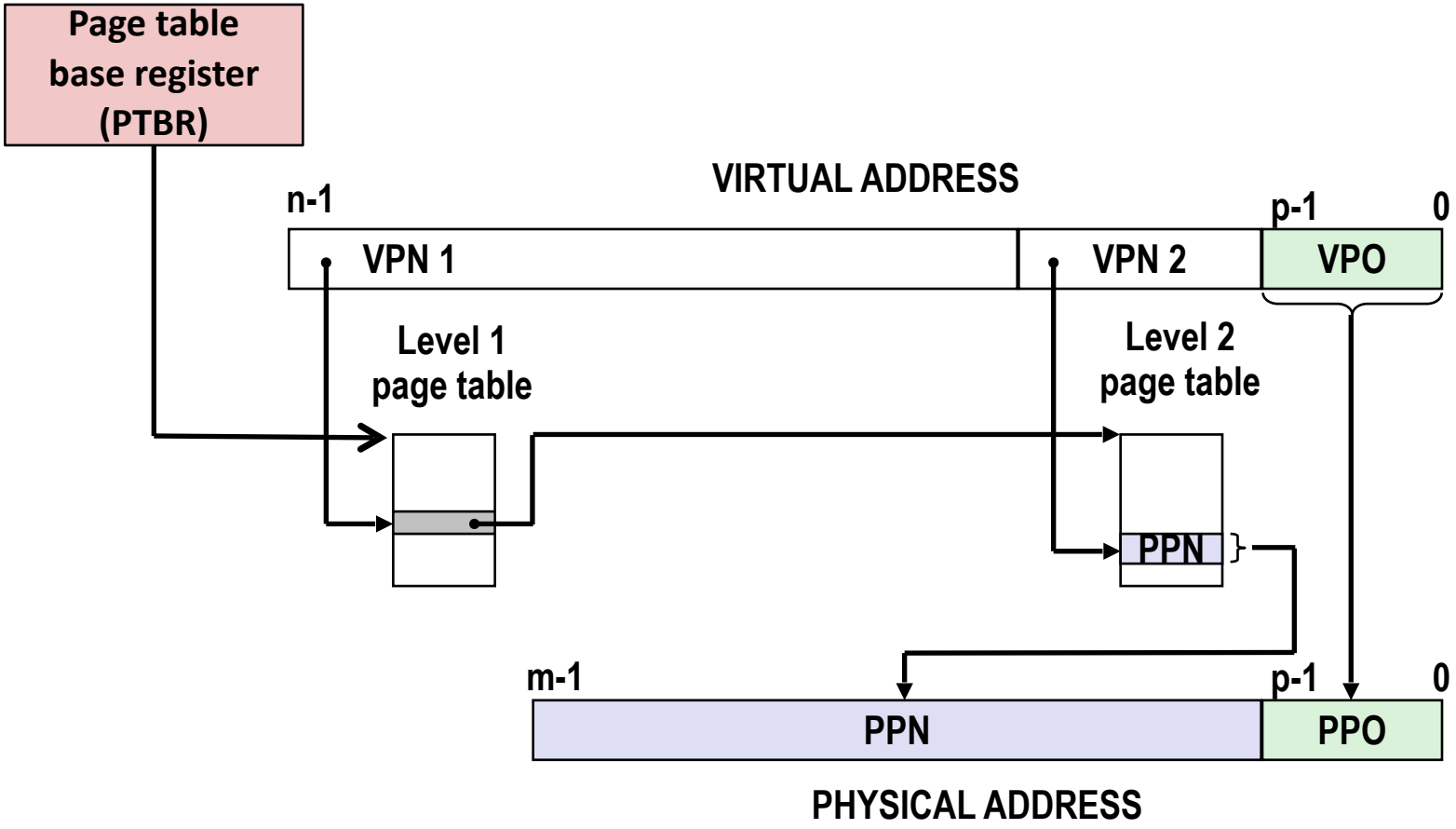


Solution: Page the Page Table

- Observation: Only a small number of pages (working set) are accessed during a certain period of time, due to locality
- Put only the relevant page table entires in main memory
- Idea: Put page table in Virtual Memory and swap it just like data



How to Access a 2-Level Page Table?

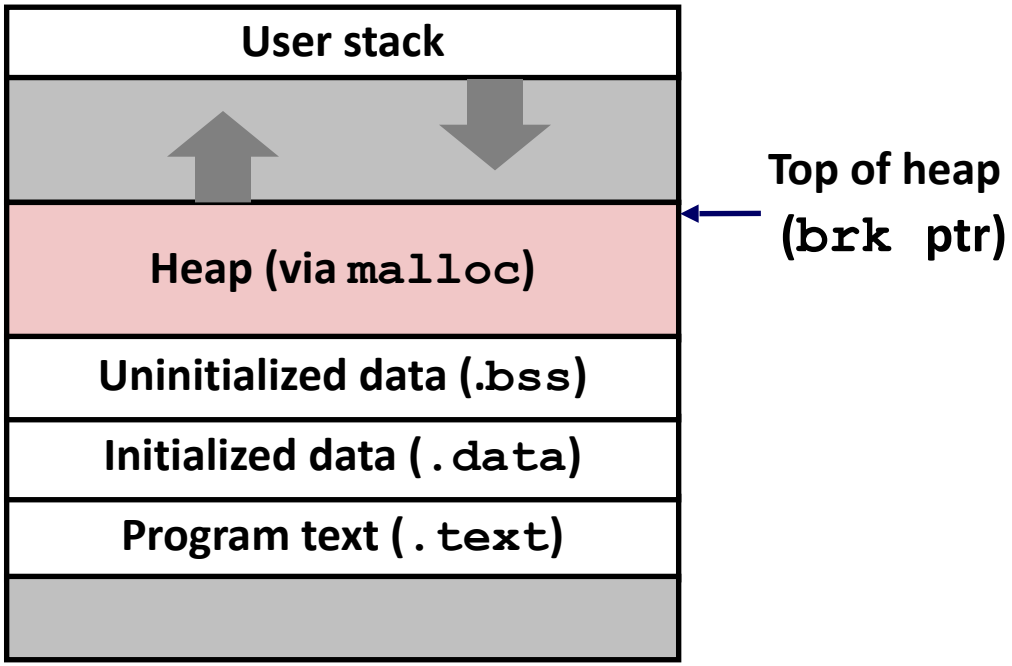


Dynamic Memory Allocation

- General idea
- Explicit allocator
- Implicit allocator

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



The `malloc/free` Functions

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- `p` must come from a previous call to `malloc` or `realloc`

Other functions

- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc`: Changes the size of a previously allocated block.
- `sbrk`: Used internally by allocators to grow or shrink the heap

Explicit Allocator

- Key Issues
  - Free:
    - How do we know how much memory to free given just a pointer?
    - How do we keep track of the free blocks?
      - The advantages and disadvantages of each free list?
    - How do we reinsert freed block?
  - Allocation:
    - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
    - How do we pick a block to use for allocation -- many might fit?
  - Performance Issues
    - Throughput
    - Memory utilization (fragmentation)

Implicit Allocator

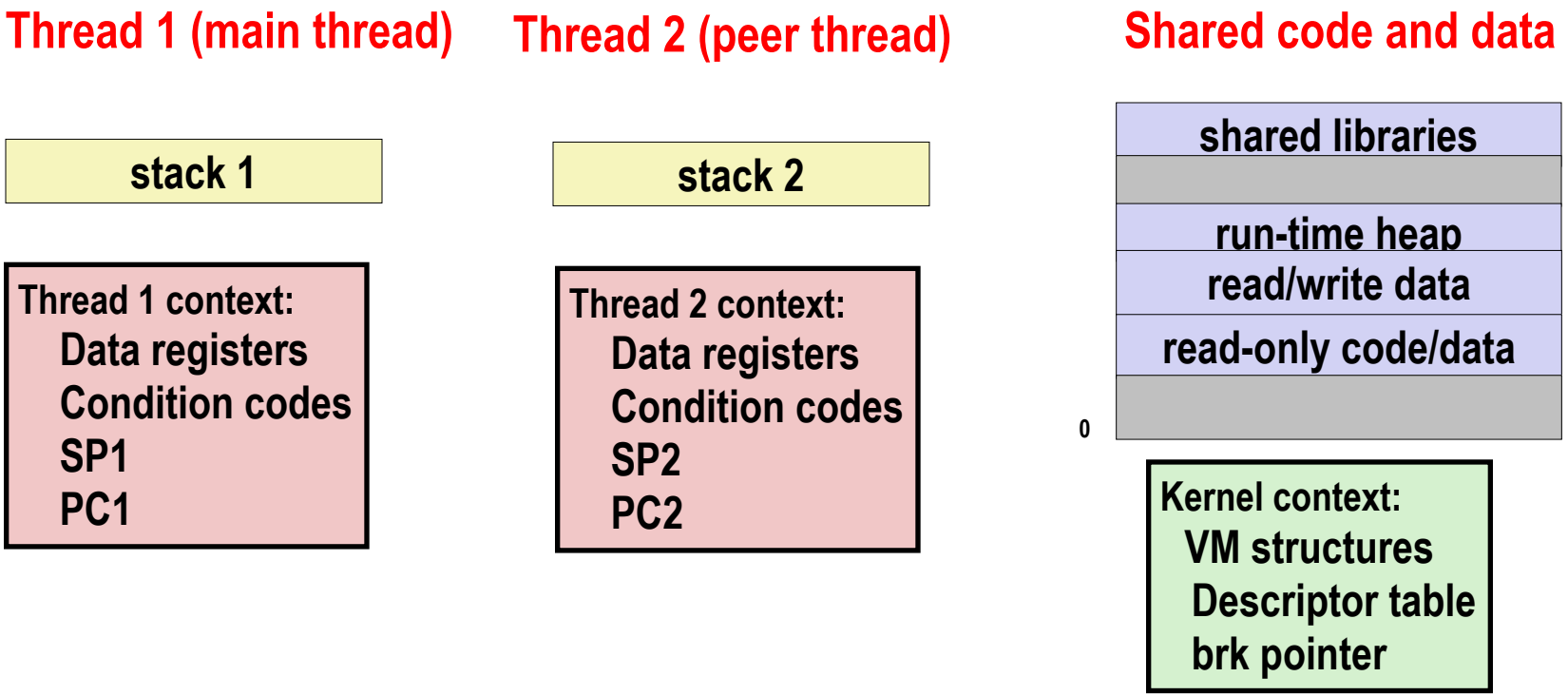
- Garbage Collection
  - Mark and Sweep
  - Mark Sweep Compact
  - How does GC affect performance?

Multi-Threading

- General idea
- Pthread library
- Thread synchronization

A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)



The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main()
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

Thread arguments (void *p)

Return value (void **p)

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate each shared variable (or related set of shared variables) with a unique variable, called **semaphore**, initially 1.
  - Every time a thread tries to enter the critical section, it first checks the semaphore value. If it's still 1, the thread decrements the mutex value to 0 (through a **P operation**) and enters the critical section. If it's 0, wait.
  - Every time a thread exits the critical section, it increments the semaphore value to 1 (through a **V operation**) so that other threads are now allowed to enter the critical section.
  - No more than one thread can be in the critical section at a time.
- Terminology
  - **Binary semaphore** is also called **mutex** (i.e., the semaphore value could only be 0 or 1)
  - Think of P operation as "**locking**", and V as "**unlocking**".

Deadlock

- Def: A process/thread is *deadlocked* if and only if it is waiting for a condition that will never be true
- General to concurrent/parallel programming (threads, processes)
- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!