Computer Organization and Networks
(INB.06000UF, INB.07001UF)

Chapter 12: Building Faster Processors

Winter 2020/2021

Stefan Mangard, www.iaik.tugraz.at

Note on Material

The following parts of the slides of this chapter are based on material from Prof. Onur Mutlu, ETH Zurich:

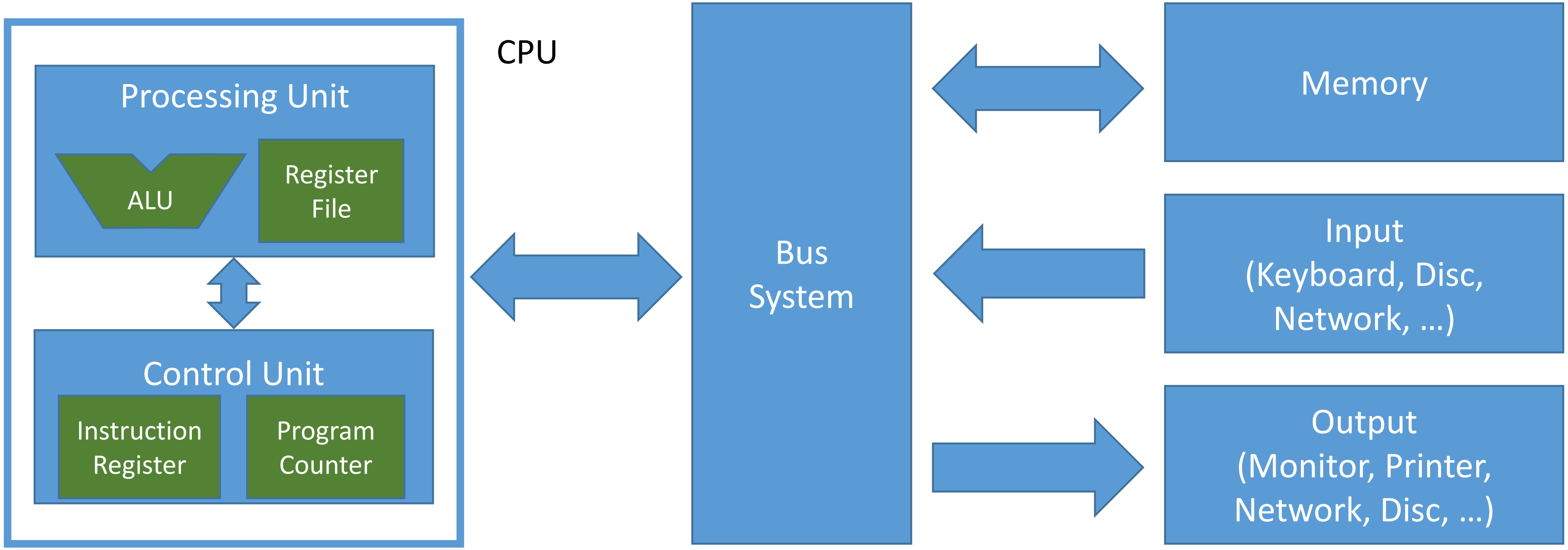
- Multi-Cycle Execution
- Out-of-Order Execution
- Memory Hierarchy and Caches

Changes that have been made:

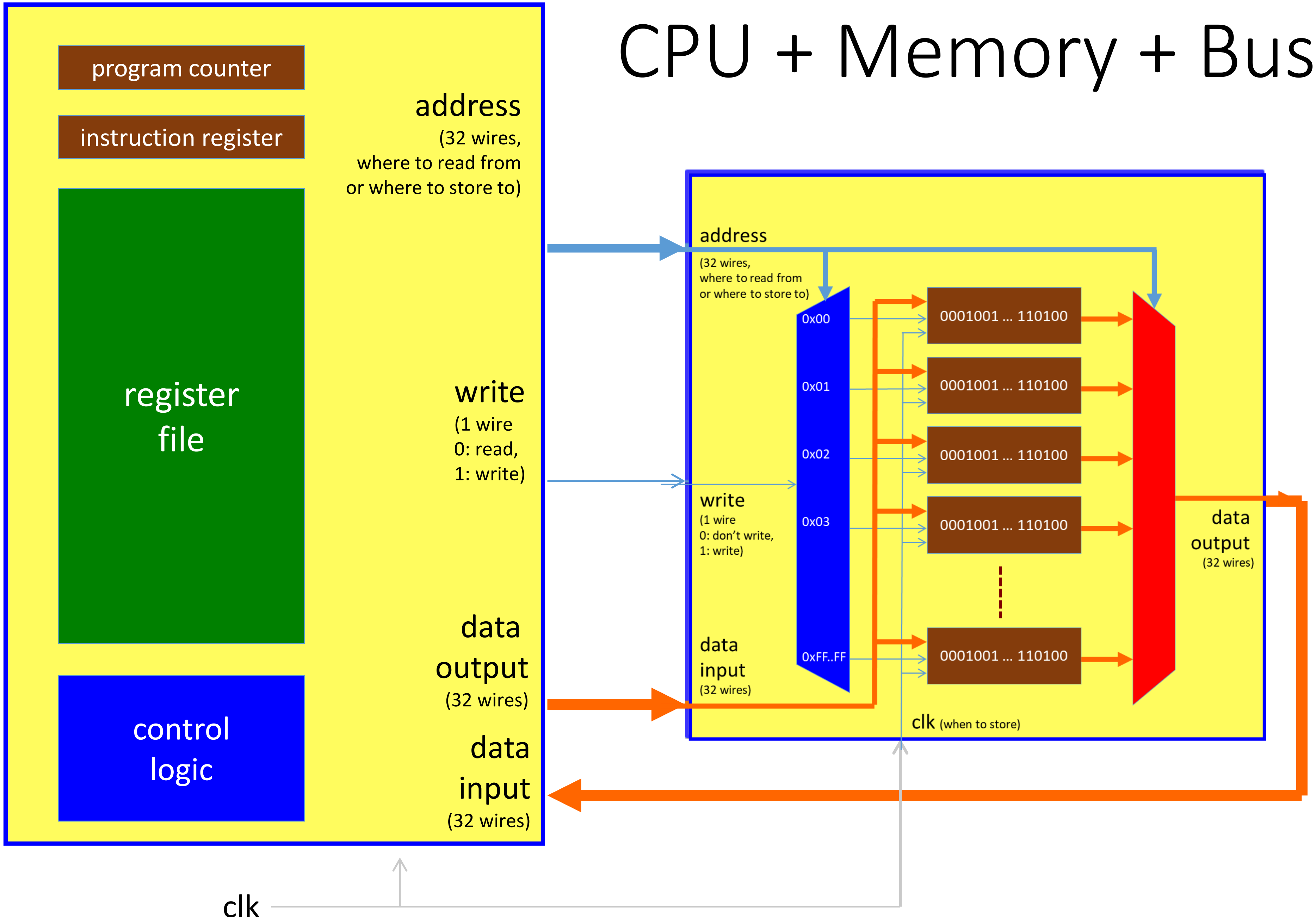
- Textual updates have been performed
- Material been combined from multiple slide decks
- Changes of the sequence and the amount of content has been done

Original source: https://safari.ethz.ch/digitaltechnik/spring2019/doku.php?id=schedule

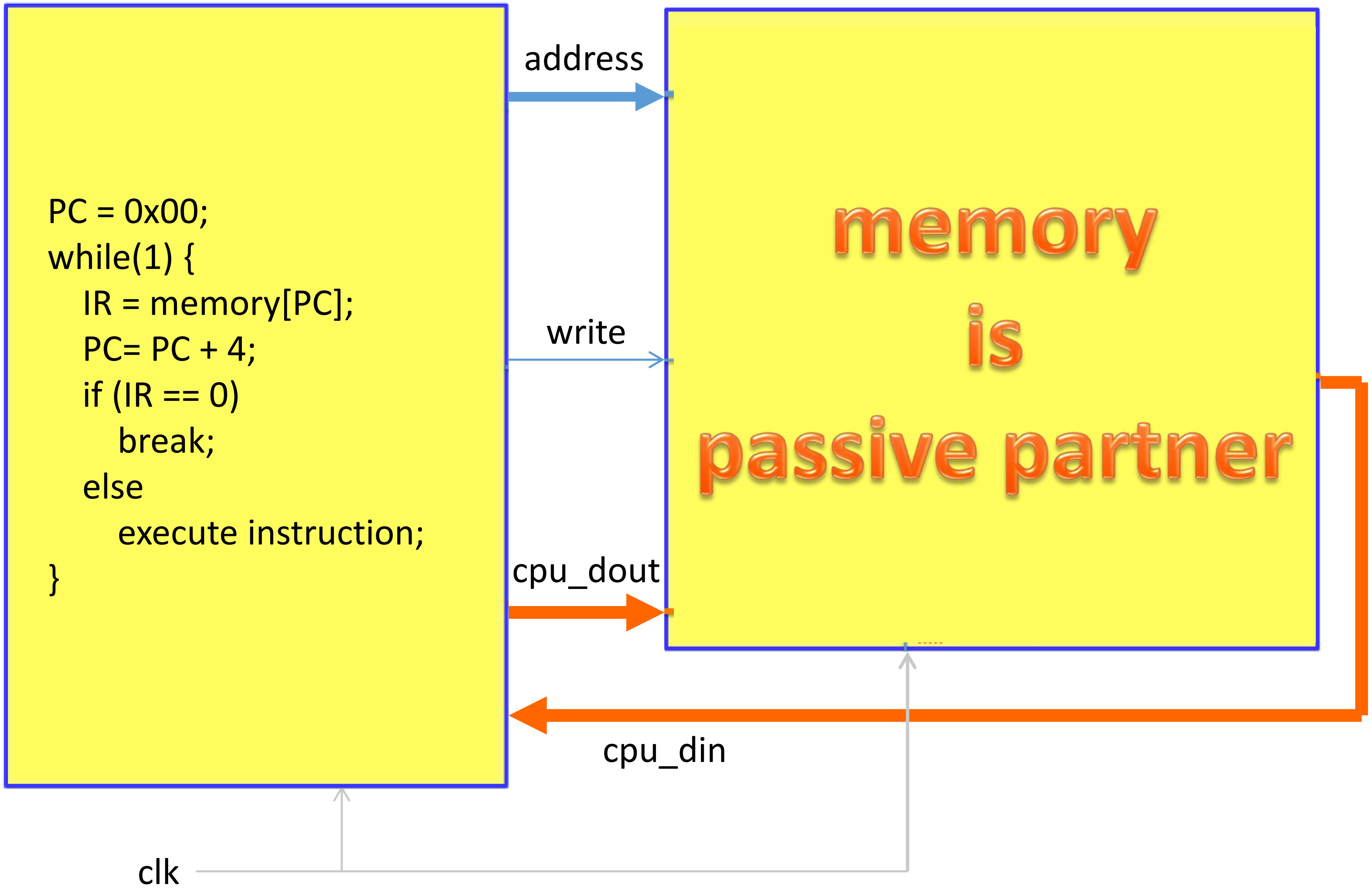
Von Neumann Model



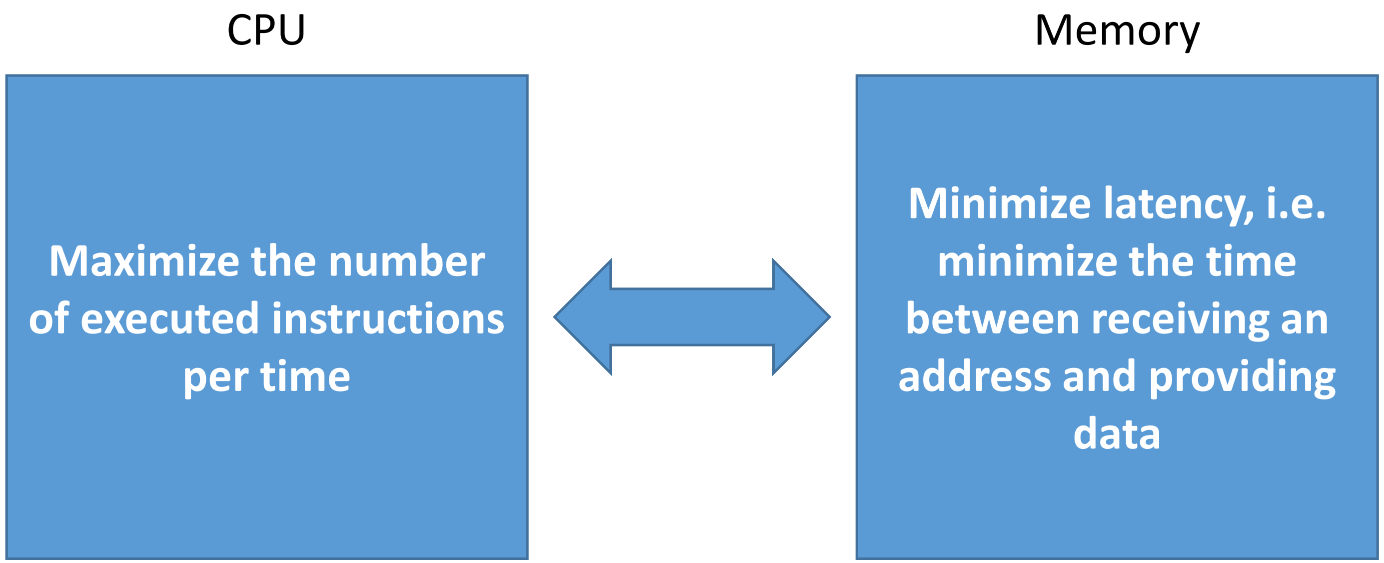
CPU + Memory + Bus



CPU's Job: Fetch, Decode, and Execute

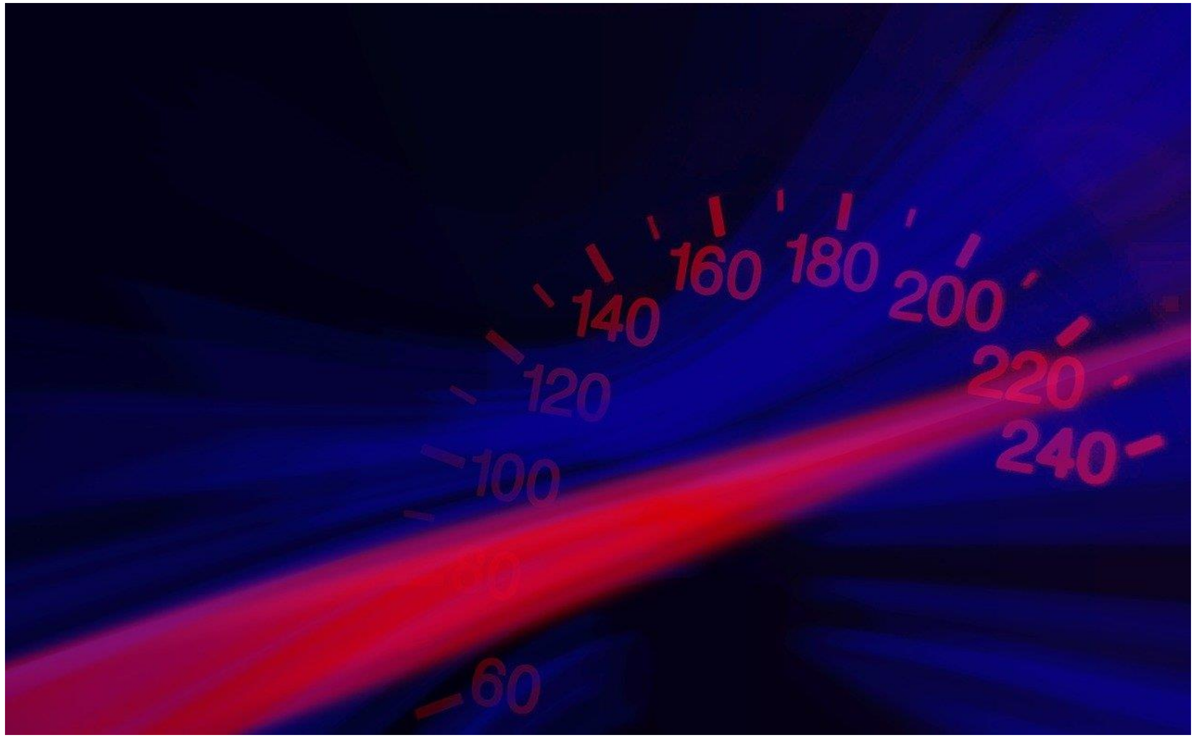


The Goal We Want to Achieve



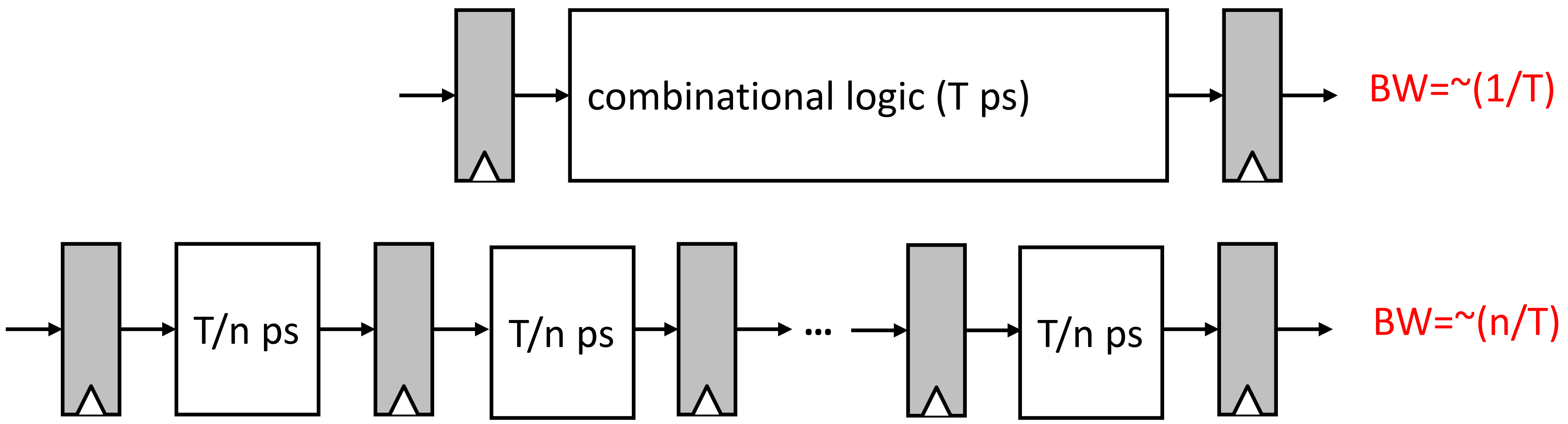
Important Acceleration Techniques for Processors

- **Pipelining**
- **Out-of-Order Execution**
- **Superscalar CPUs**
- **Multiple CPUs**
- **Speculative Execution**



Pipelining

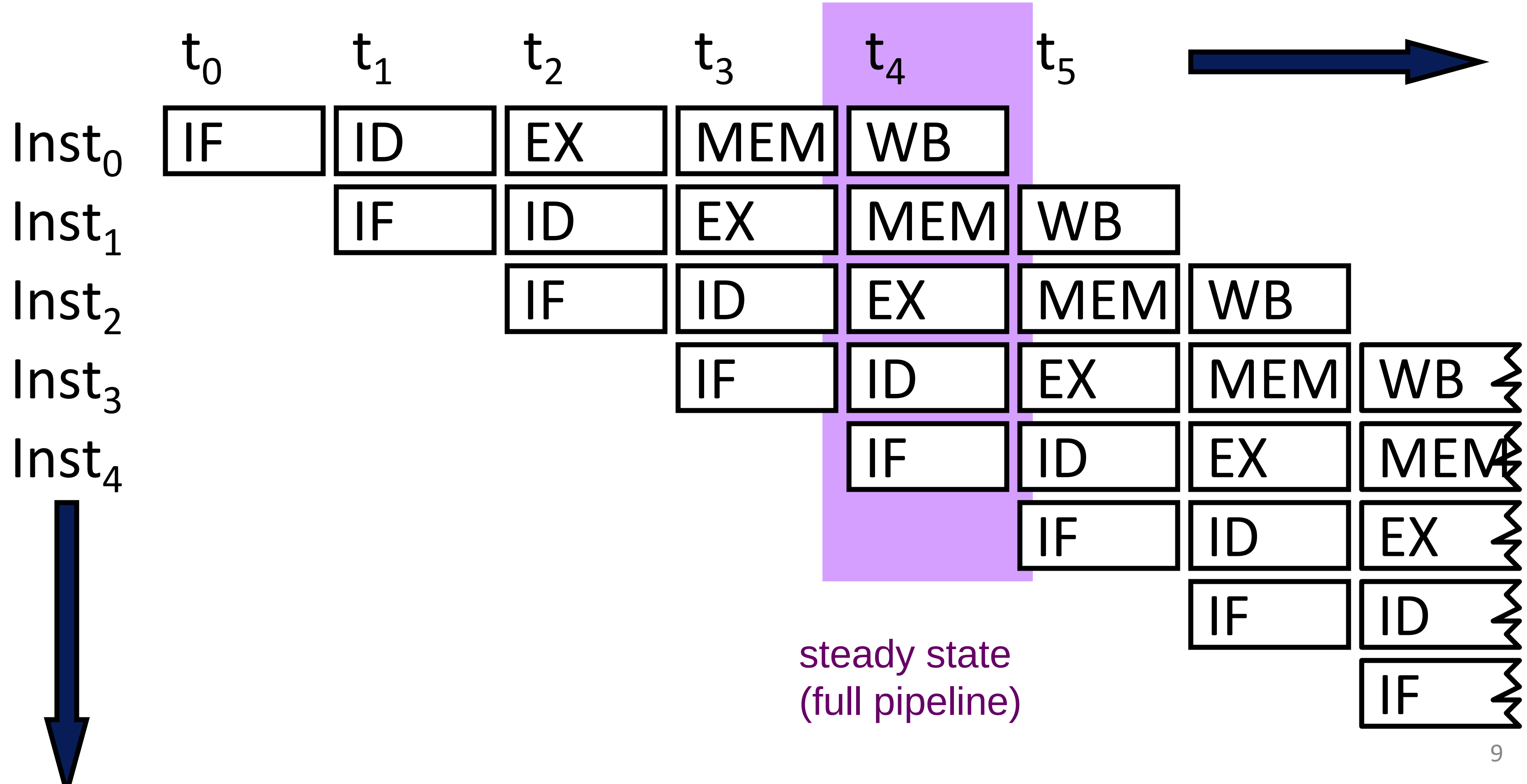
- Idea:
  - Divide the instruction processing into distinct “stages” of processing
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages



The Example Discussed Earlier in the Lecture

5-stage pipeline:

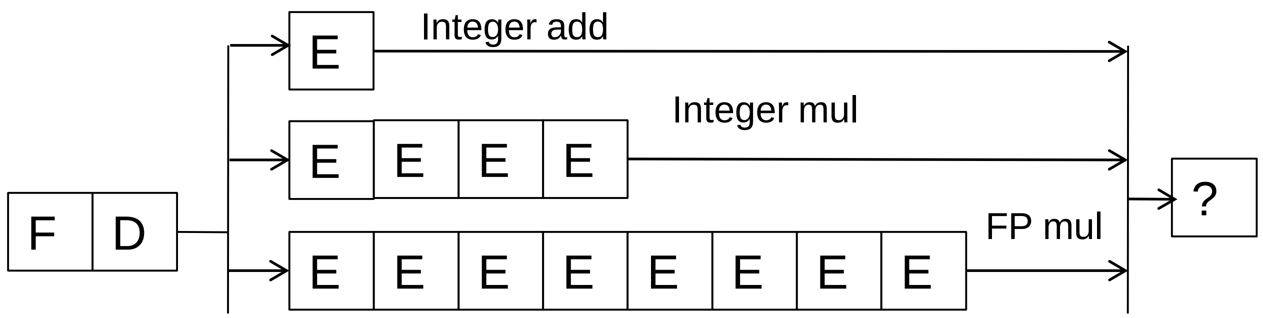
- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execution (EX)
- Memory Access (MEM)
- Write Back (WB)



Pipelining & Multi-Cycle Execution

Multi-Cycle Execution

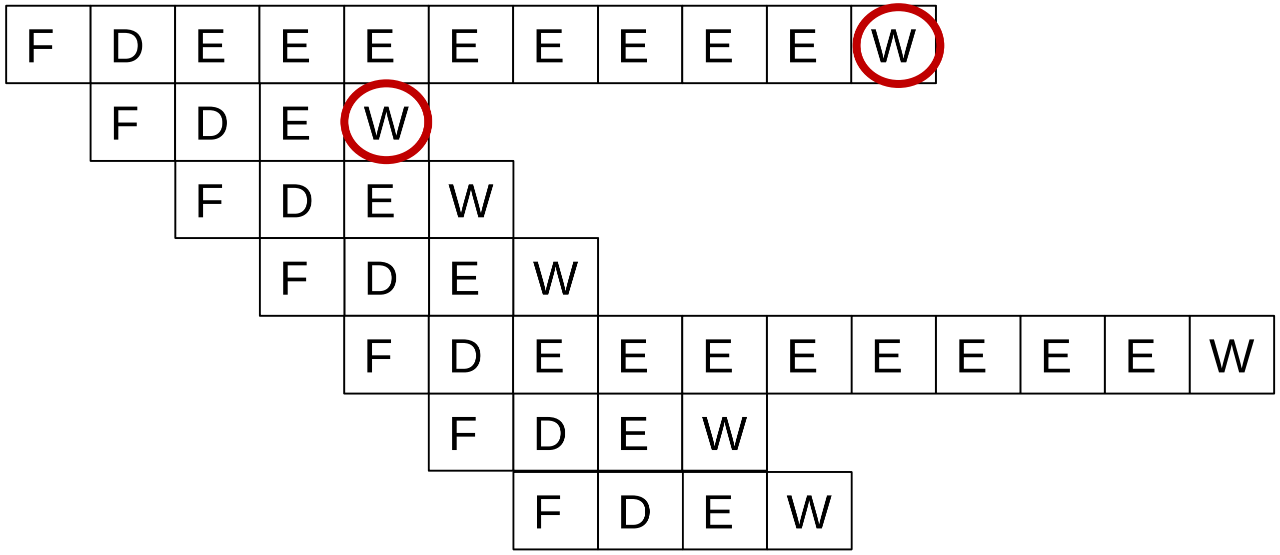
- Not all instructions need the same amount of time for "execution"
- Idea: Have multiple different functional units that take different number of cycles
  - Let independent instructions start execution on a different functional unit before a previous long-latency instruction finishes execution



Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

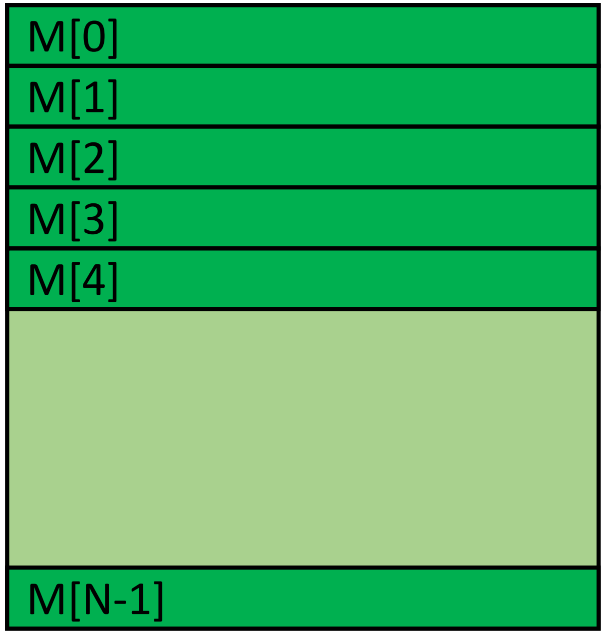
FMUL R4 ← R1, R2
ADD R3 ← R1, R2



FMUL R2 ← R5, R6
ADD R7 ← R5, R6

- What is wrong with this picture in a Von Neumann architecture?
  - If we complete ADD before FMUL, the sequential semantics of the ISA NOT preserved!

Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address

Registers
- given special names in the ISA
  (as opposed to addresses)
- general vs. special purpose

memory address
of the current instruction

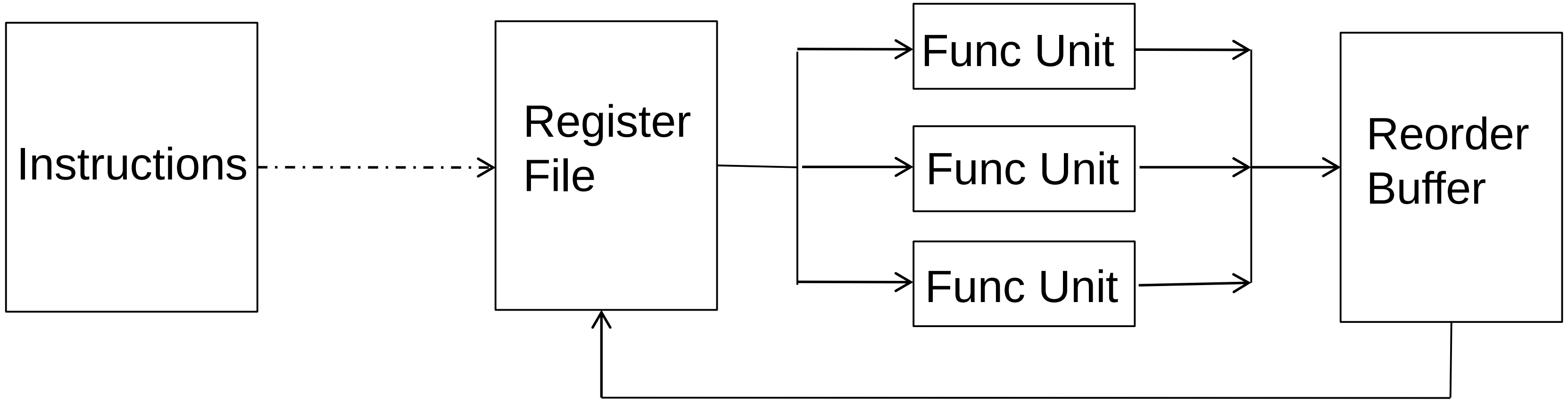
Instructions (and programs) specify how to transform
the values of programmer visible state

The Contract Between the Hardware and the Software

- The software requires that
  - Instructions that have been executed up to the PC (program counter) have been executed in the given order
  - Instructions beyond the current value of the PC do not affect the architectural state of the processor

Reorder Buffer (ROB)

- Idea: **Complete** instructions **out-of-order**, but reorder them before making results visible to architectural state
- When instruction is decoded it reserves the next-sequential entry in the ROB
- When instruction completes, it writes result into ROB entry
- When the oldest instruction in the ROB has completed without exceptions, its result moved to register file or memory



Reorder Buffer

- Buffers information about all instructions that are decoded but not yet retired/committed

- It needs to store all information that is required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction

- Needs valid bits to keep track of readiness of the result(s) and find out if the instruction has completed execution

Reorder Buffer: Independent Operations

- Result first written to ROB on instruction completion
- Result written to register file at commit time

F	D	E	E	E	E	E	E	E	E	R	W						
	F	D	E	R								W					
		F	D	E	R								W				
			F	D	E	R								W			
				F	D	E	E	E	E	E	E	E	E	R	W		
					F	D	E	R								W	
						F	D	E	R								W

- What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation → stall the pipeline
  - Better: Read the value from the reorder buffer.

Efficient Reorder Buffer Access

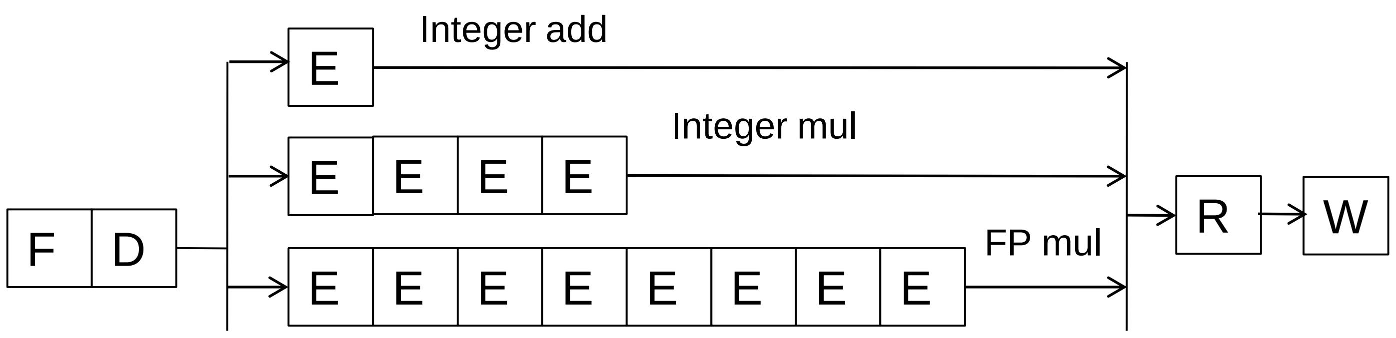
- Access register file first (check if the register is valid)
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register

  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next

Out-of-Order Execution

An In-order Pipeline



- Dispatch: Act of sending an instruction to a functional unit
- Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

Can We Do Better?

- What do the following two pieces of code have in common?

```
MUL   R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
MUL   R5 ← R6, R8
ADD   R7 ← R9, R9
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R4 ← R6, R7
MUL   R5 ← R6, R8
ADD   R7 ← R9, R9
```

- Answer: First ADD stalls the whole pipeline!
  - The MUL and the LD instruction take many cycles to execute
  - ADD cannot dispatch because its source registers unavailable
  - Later **independent** instructions cannot get executed

Preventing Dispatch Stalls

- Problem: **in-order** dispatch (scheduling, or execution)
- Solution: **out-of-order** dispatch (scheduling, or execution)
- Basic idea: "fire" an instruction when its inputs are ready

Out-of-order Execution (Dynamic Scheduling)

- Idea: Move the dependent instructions out of the way of independent ones (such that independent ones can execute)
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source "values" of each instruction in the resting area

- When all source "values" of an instruction are available, "fire" (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- Benefit:
  - Latency tolerance: Allows independent instructions to execute and complete in the presence of a long-latency operation

In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:

F	D	E	E	E	E	R	W								
	F	D		STALL		E	R	W							
		F		STALL		D	E	R	W						
						F	D	E	E	E	E	E	R	W	
							F	D			STALL		E	R	W

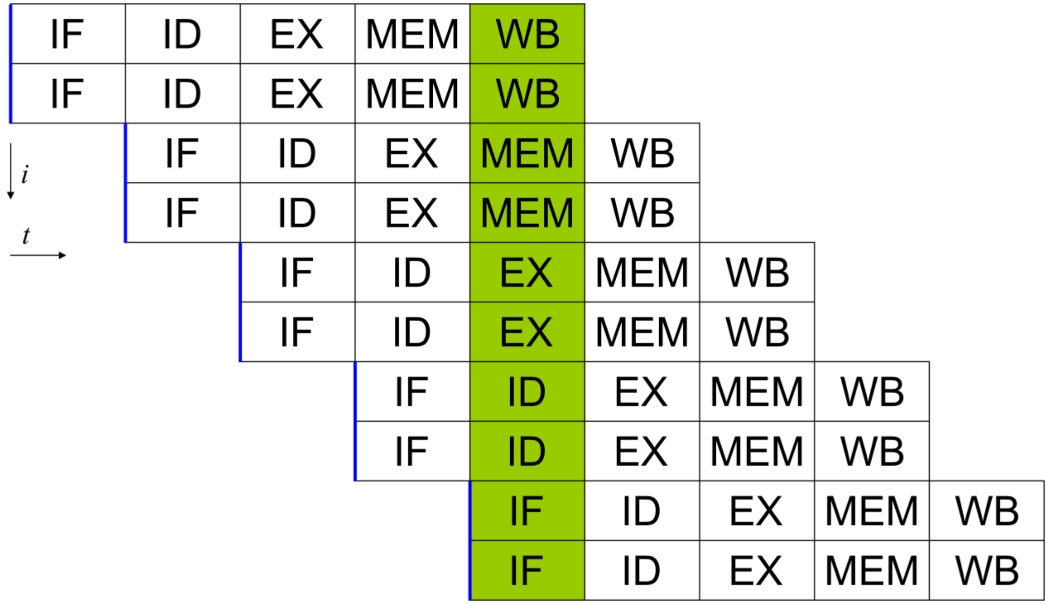
MUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
MUL R5 ← R6, R8
ADD R7 ← R3, R5

- Out-of-order dispatch + precise exceptions:

F	D	E	E	E	E	R	W				
	F	D		WAIT		E	R	W			
		F	D	E	R				W		
			F	D	E	E	E	E	R	W	
				F	D		WAIT		E	R	W

- 16 vs. 12 cycles

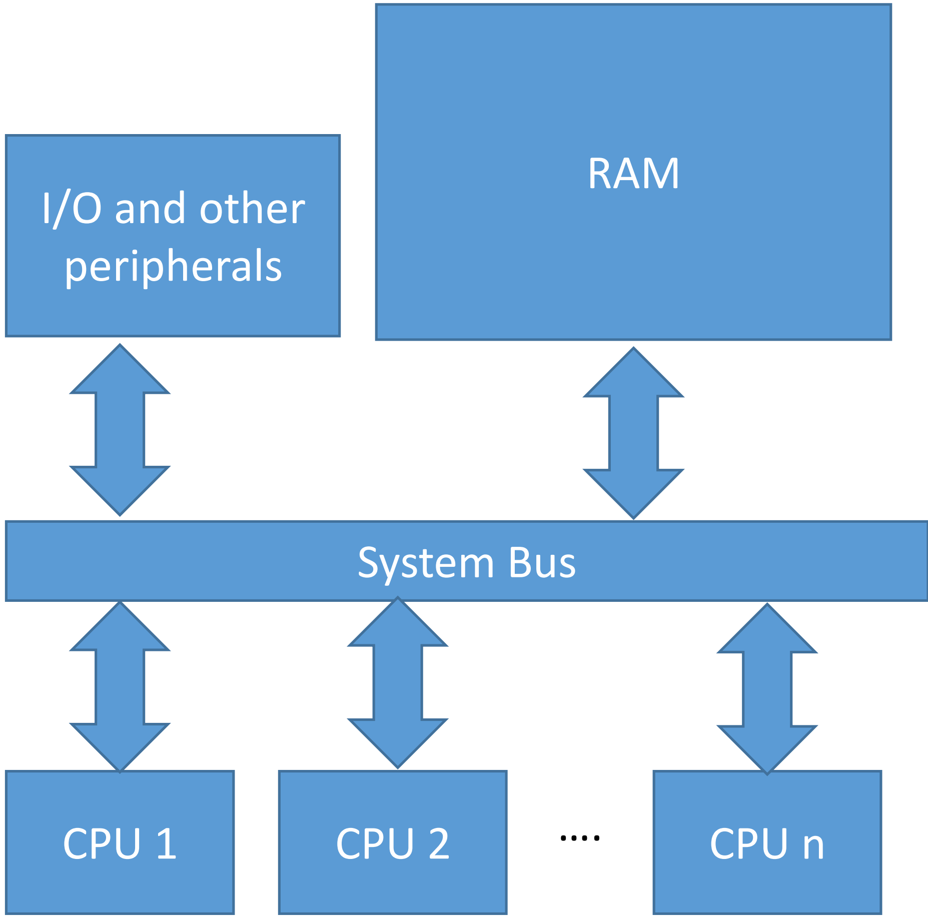
Superscalar CPUs



- **Basic Idea**
  - Add hardware to be able to handle multiple instructions in each pipeline stage (e.g. fetch two instructions at the same time, execute two instructions at the same time, …)
  - The width can be varied for each stage

Multiple CPUs

- **Basic Idea**
  - Put multiple CPU cores on one chip
  - Typical setup is symmetric: all CPUs are equal
  - All are connected to a shared memory
- **Important Topics**
  - Scheduling of processes on the different CPUs
  - Arbitration of shared resources
  - Security



What About the Memory?

- We build CPUs that can execute more and more instructions per time and we instantiate more and more CPUs?

Is the memory fast enough to deliver all the instructions and data to the CPUs?

Slow Memory Accesses

- In general memory accesses are slow
- In the worst case a single access can take the same time as hundreds instructions on the CPU
- Caches (later in this lecture) are a technique to decrease the access time to memory.
- However, slow accesses happen → In order to not lose performance, CPUs use speculative execution

Speculative Execution / Branch Prediction

- **Motivation**
  - If there is a conditional branch and it is not clear if the branch will be taken or not, the CPU can't fetch any more instructions
- **Basic Idea**
  - Instead of waiting for a branch condition (e.g. because it depends on a memory access), speculate on the outcome and continue execution storing the results in the reorder buffer
  - Trash the result in case the speculation was incorrect, make the execution architecturally visible, if it was correct
- **Implementations**
  - Significant effort is spent by CPUs on learning to predict the branches correctly in an executed program
    - Branch prediction on is done based on execution history: if a branch was taken before, it is likely to be taken again (think of loops!)

Side Effects of Speculation

- **Side Effects**
  - Speculative execution does cause side effects on current CPUs; e.g. instructions that are executed speculatively and trashed affect the timing of actual instructions that are executed later on
  - Timing differences can be exploited in order to make trashed results visible

- **More on this**
  - https://spectreattack.com
  - Bachelor course "Information Security"
  - Master course "Side-Channel Security"

- Our institute was part of the team finding these completely new attacks and of many follow-up works

Memory Hierarchy and Caches

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

The Problem

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec

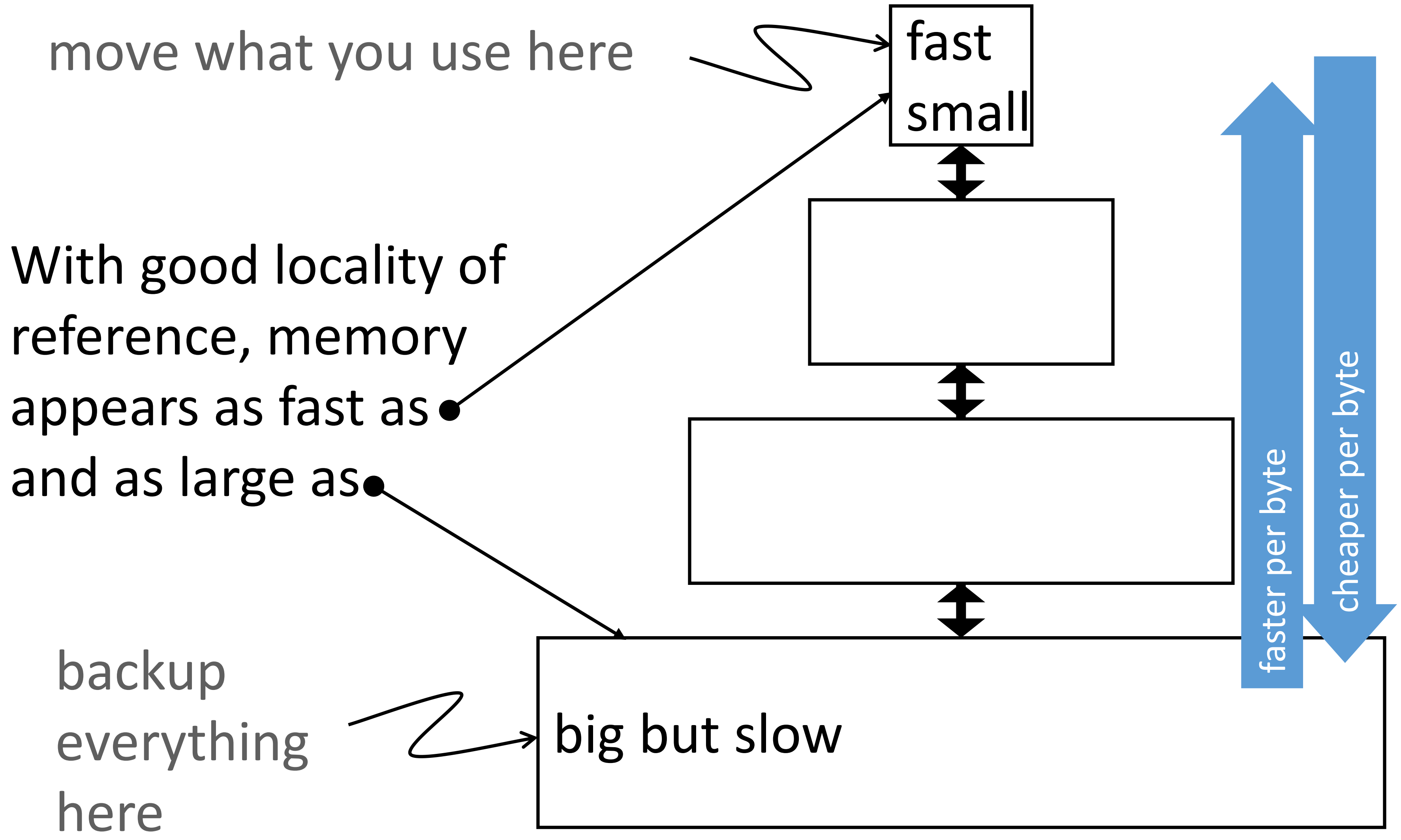
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time

- Other technologies have their place as well
  - Flash memory, MRAM, RRAM, …

Why Memory Hierarchy?

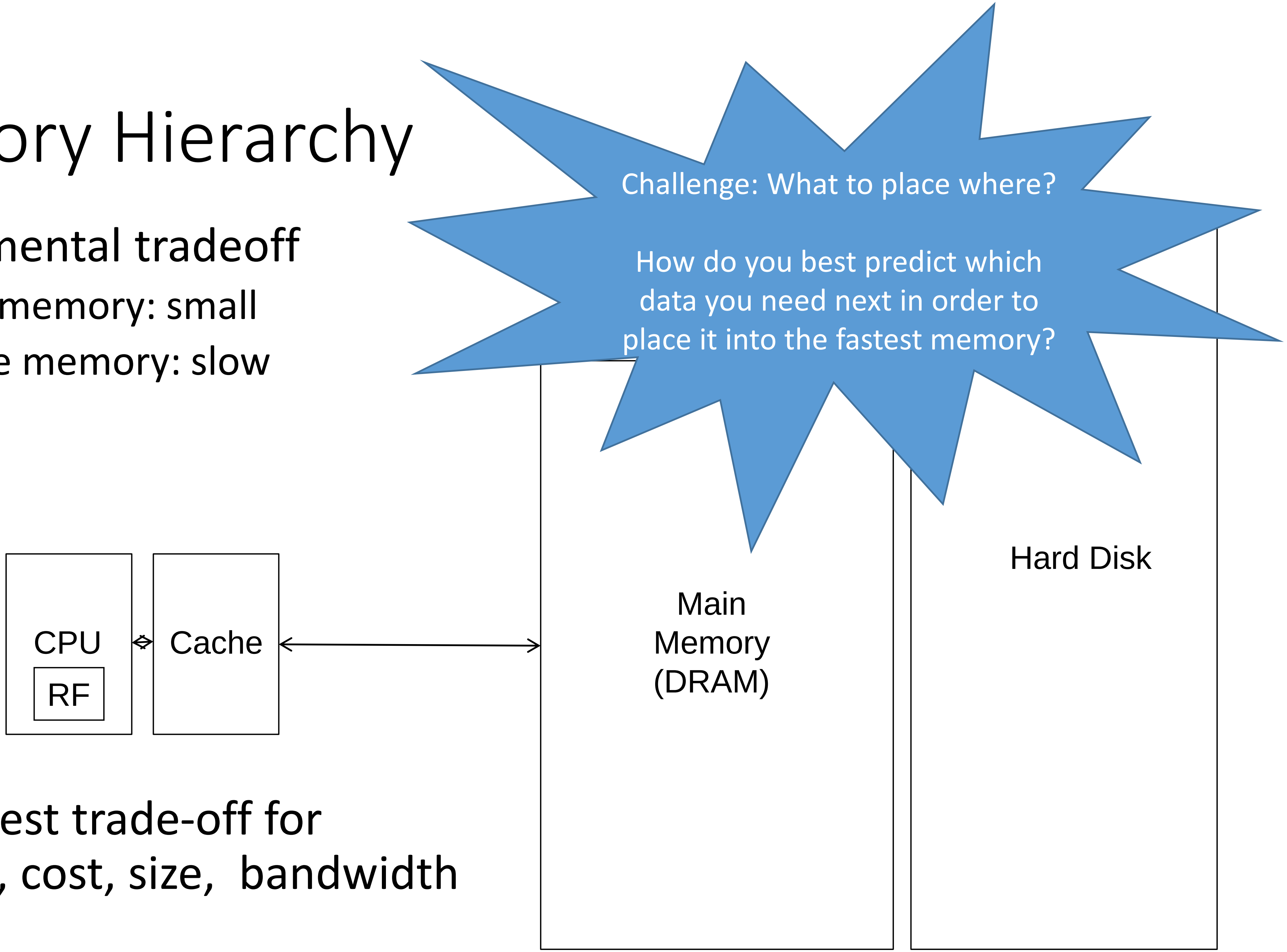
- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

The Memory Hierarchy



Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow



- Goal: Best trade-off for latency, cost, size, bandwidth

Locality

- One's recent past is a very good **predictor** of his/her near future.

- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon

- Temporal locality principle
  - Recently accessed data will be again accessed in the near future

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal

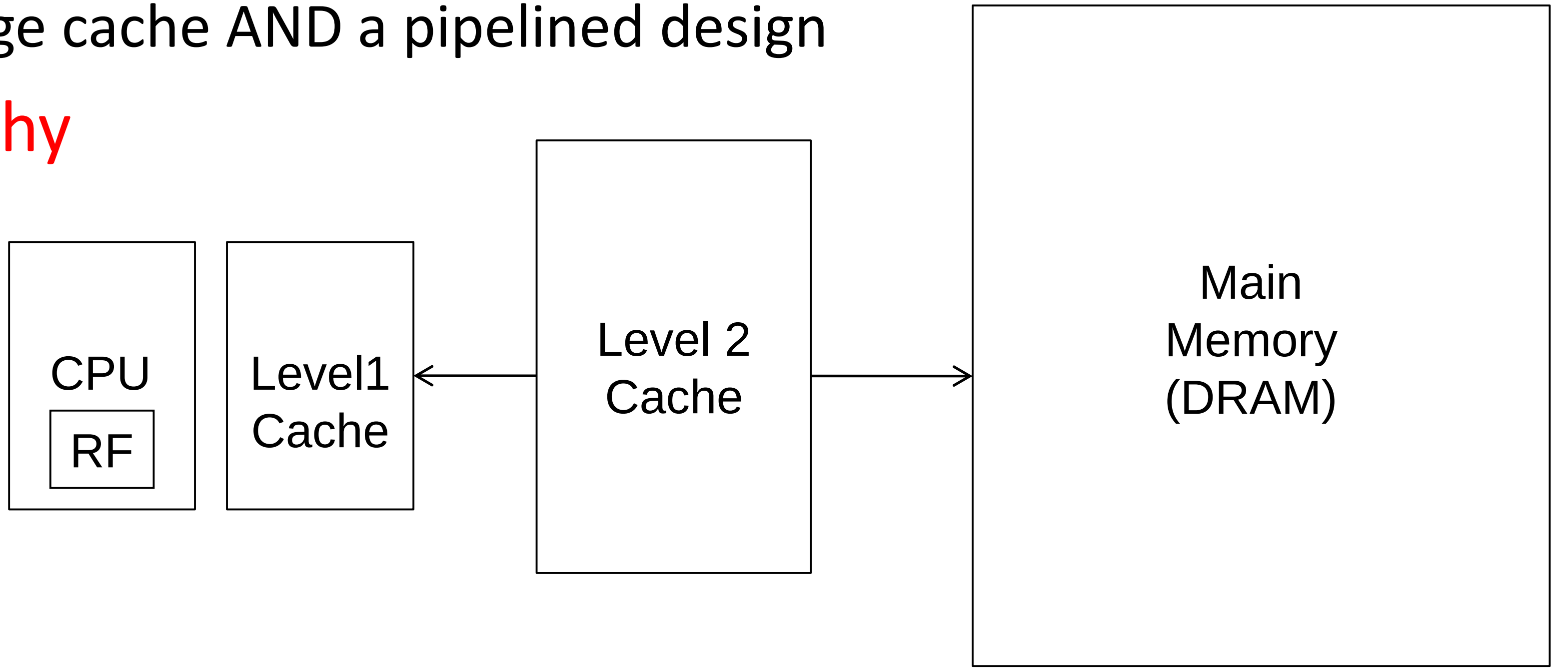
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that load-dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy



A Note on Manual vs. Automatic Management

- Manual: Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

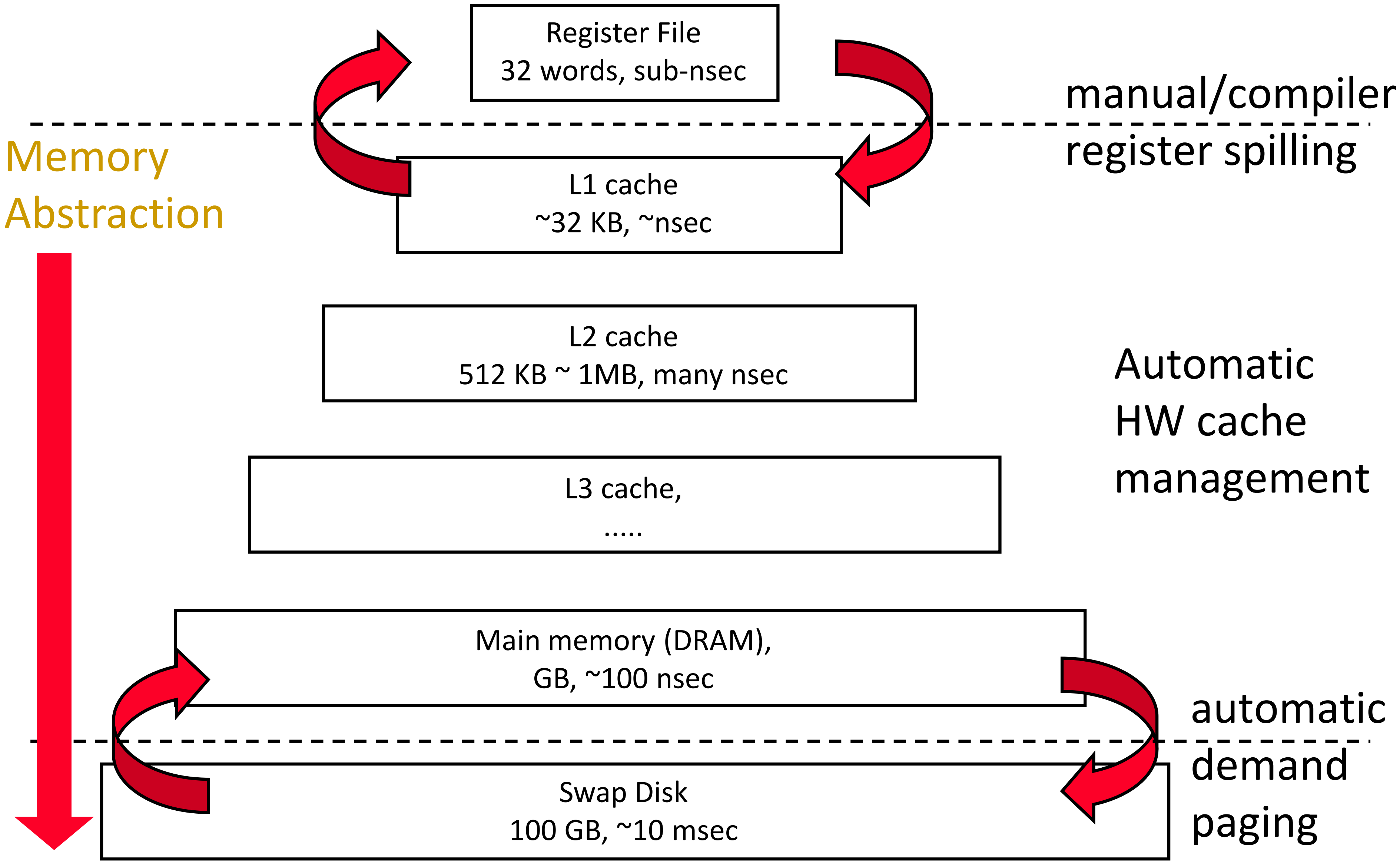
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache) and GPUs (called "shared memory")

- Automatic: Hardware manages data movement across levels, transparently to the programmer

  ++ programmer's life is easier

  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

A Modern Memory Hierarchy



Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of $t_{i,}$ The perceived access time T_i is longer than t_i
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate h_i) you "hit" and access time is t_i
  - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

$$T_i = t_i + m_i \cdot T_{i+1}$$

h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost
- $T_i \approx t_i$ is desirable

- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter cache management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)

- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Caches

Cache

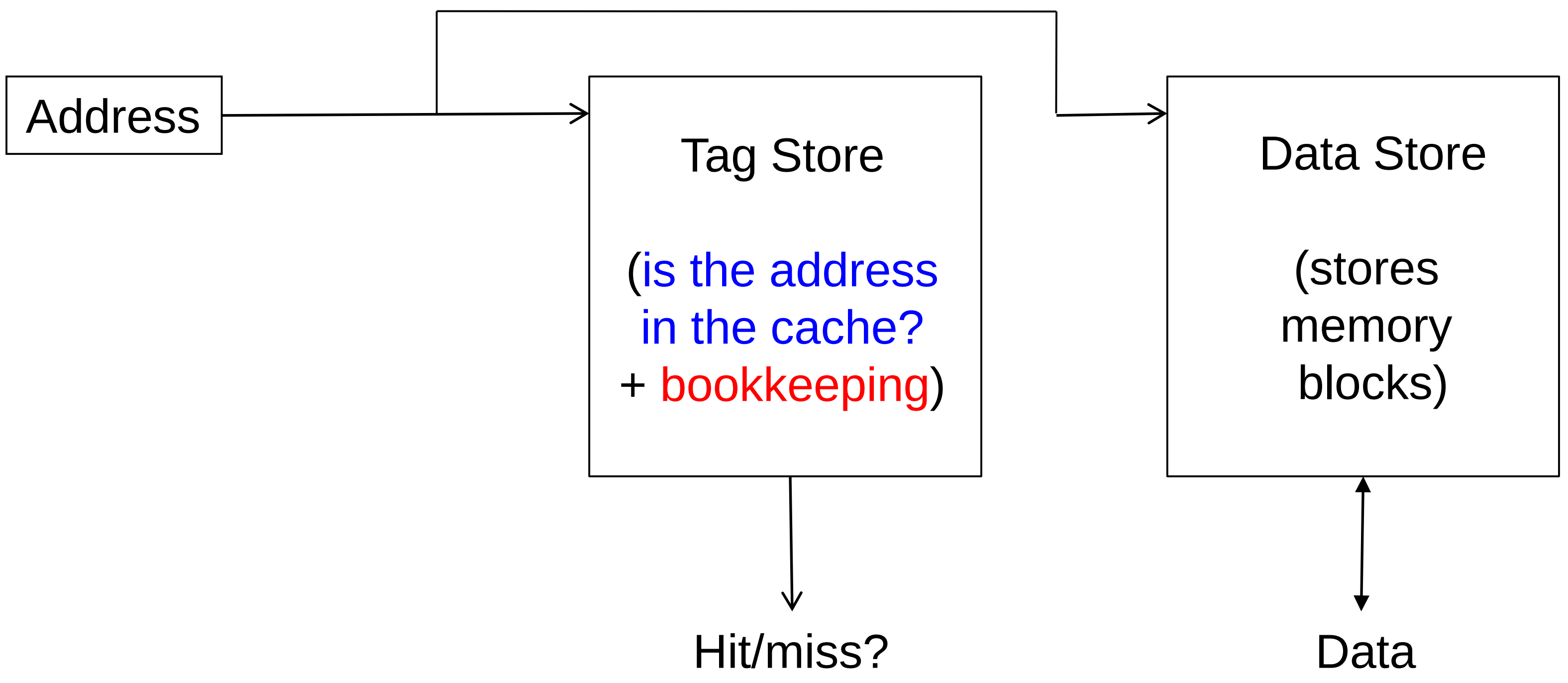
- Generically, any structure that "memorizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly in the processor design context: an automatically-managed memory structure based on SRAM
  - memorize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Caching Basics

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache
- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large or small blocks? Subblocks?
  - Write policy: what do we do about writes?
  - Instructions/data: do we treat them separately?

Cache Abstraction and Metrics



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)

  = (hit-rate * hit-latency) + (miss-rate * miss-latency)

A Basic Hardware Cache Design

- We will start with a basic hardware cache design
- Then, we will examine a multitude of ideas to make it better

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks

- Each block maps to a location in the cache, determined by the index bits in the address
  - used to index into the tag and data stores

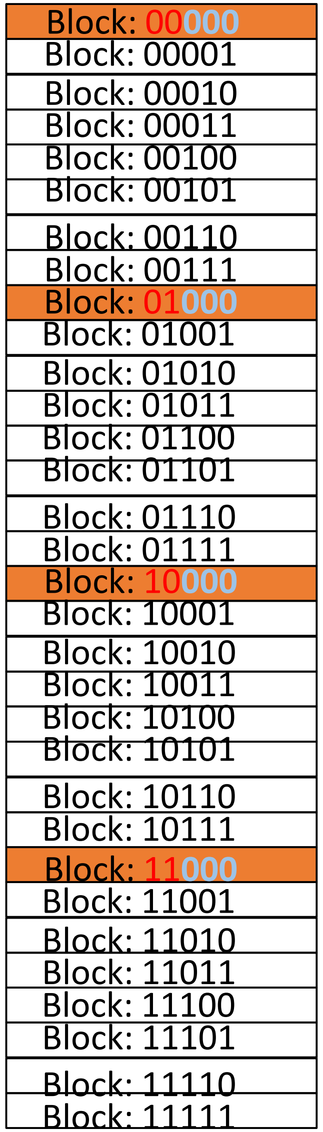
tag	index	byte in block
2b	3 bits	3 bits

8-bit address

- Cache access:
  1) index into the tag and data stores with index bits in address
  2) check valid bit in tag store
  3) compare tag bits in address with the stored tag in tag store

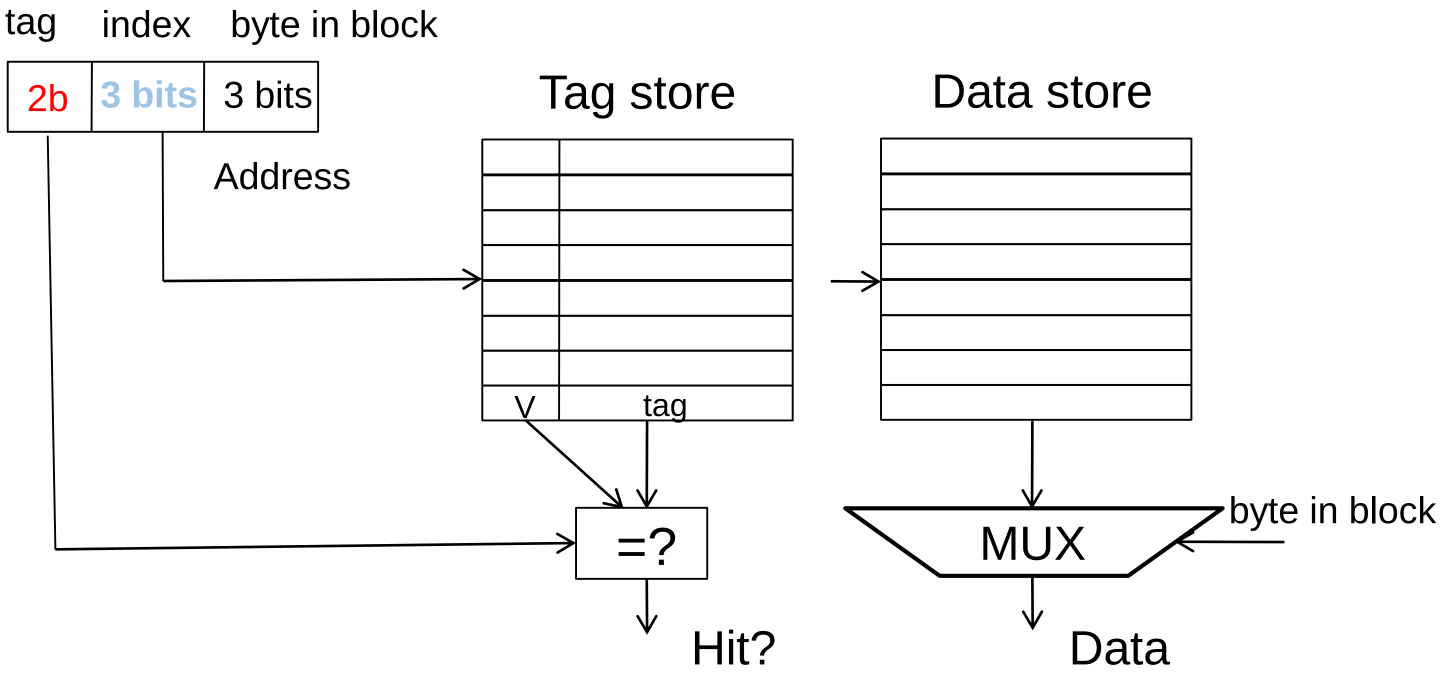
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache: Placement and Access



Main memory

- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location



- Addresses with same index contend for the same location
  - Cause conflict misses

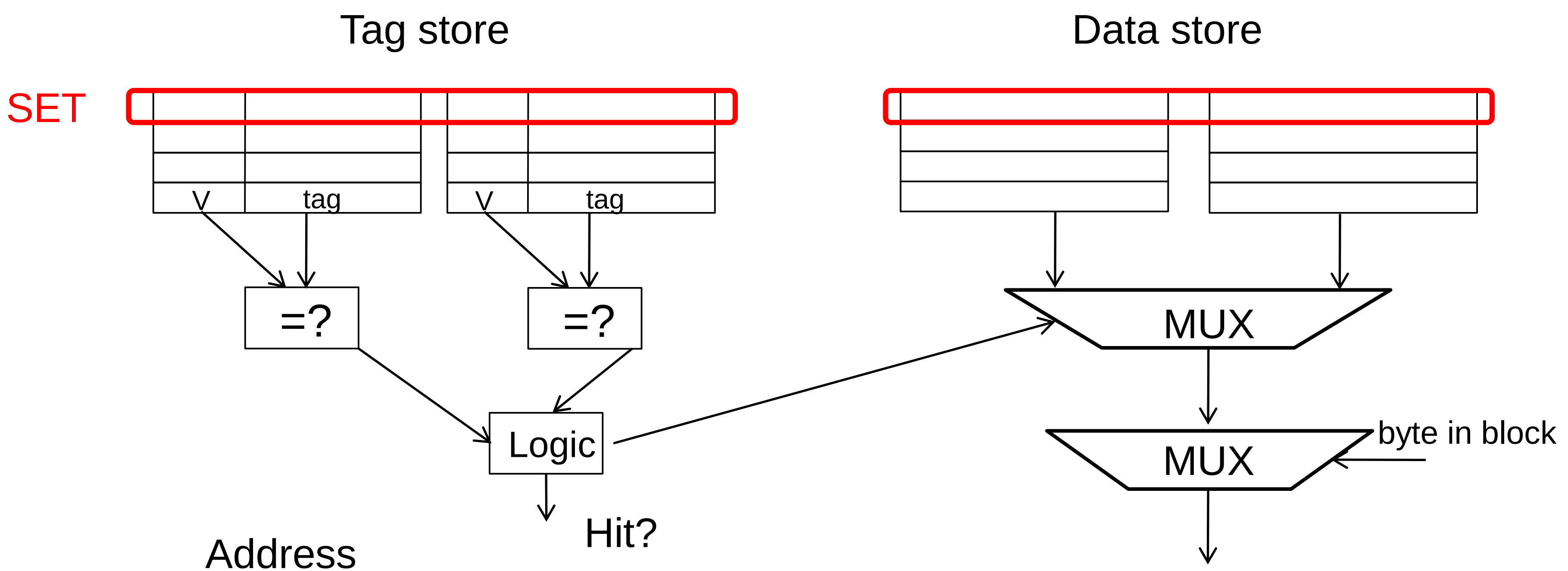
Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Address

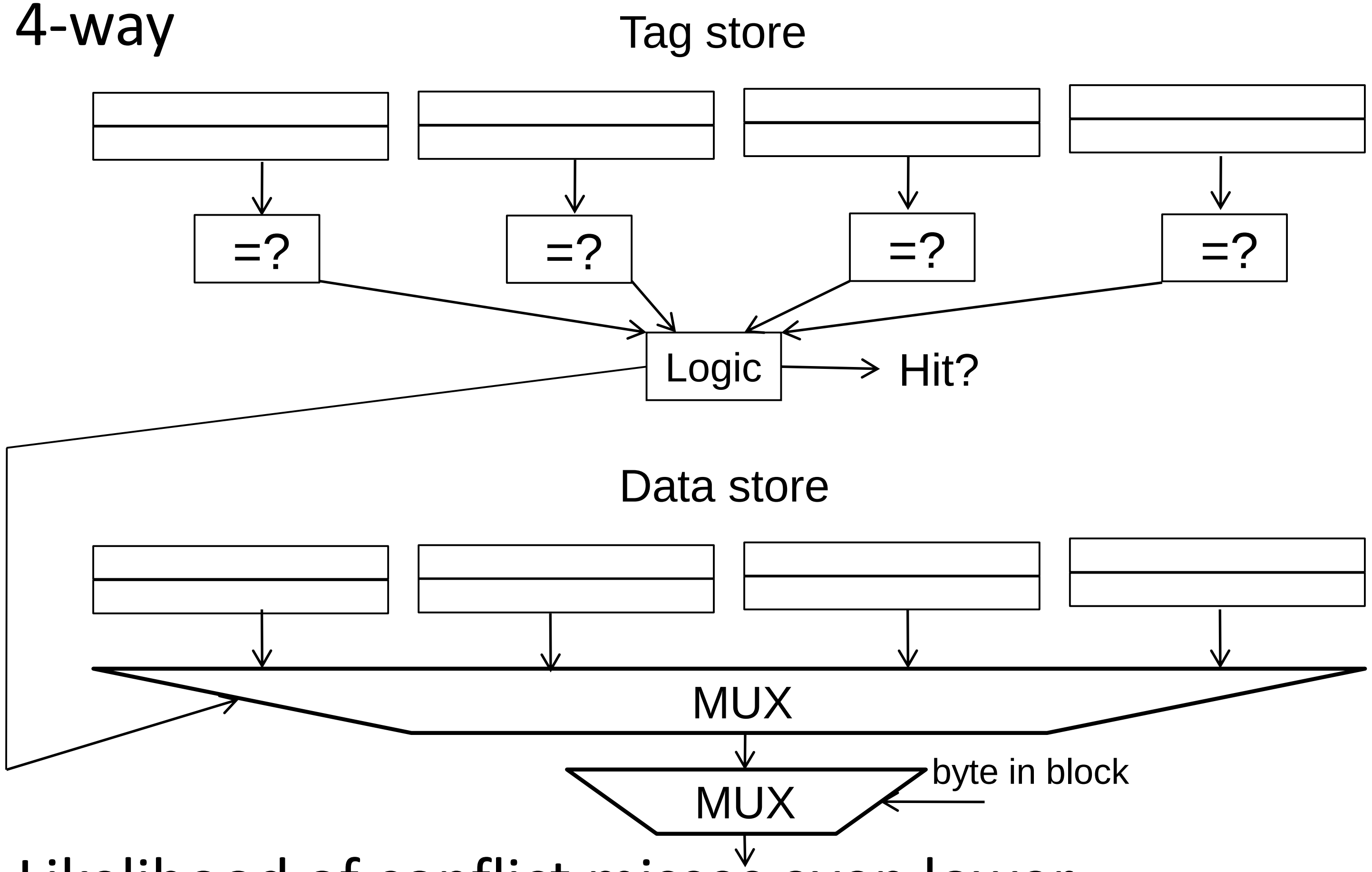
tag	index	byte in block
3b	2 bits	3 bits

+ Accommodates conflicts better (fewer conflict misses)

-- More complex, slower access, larger tag store

Higher Associativity

- 4-way

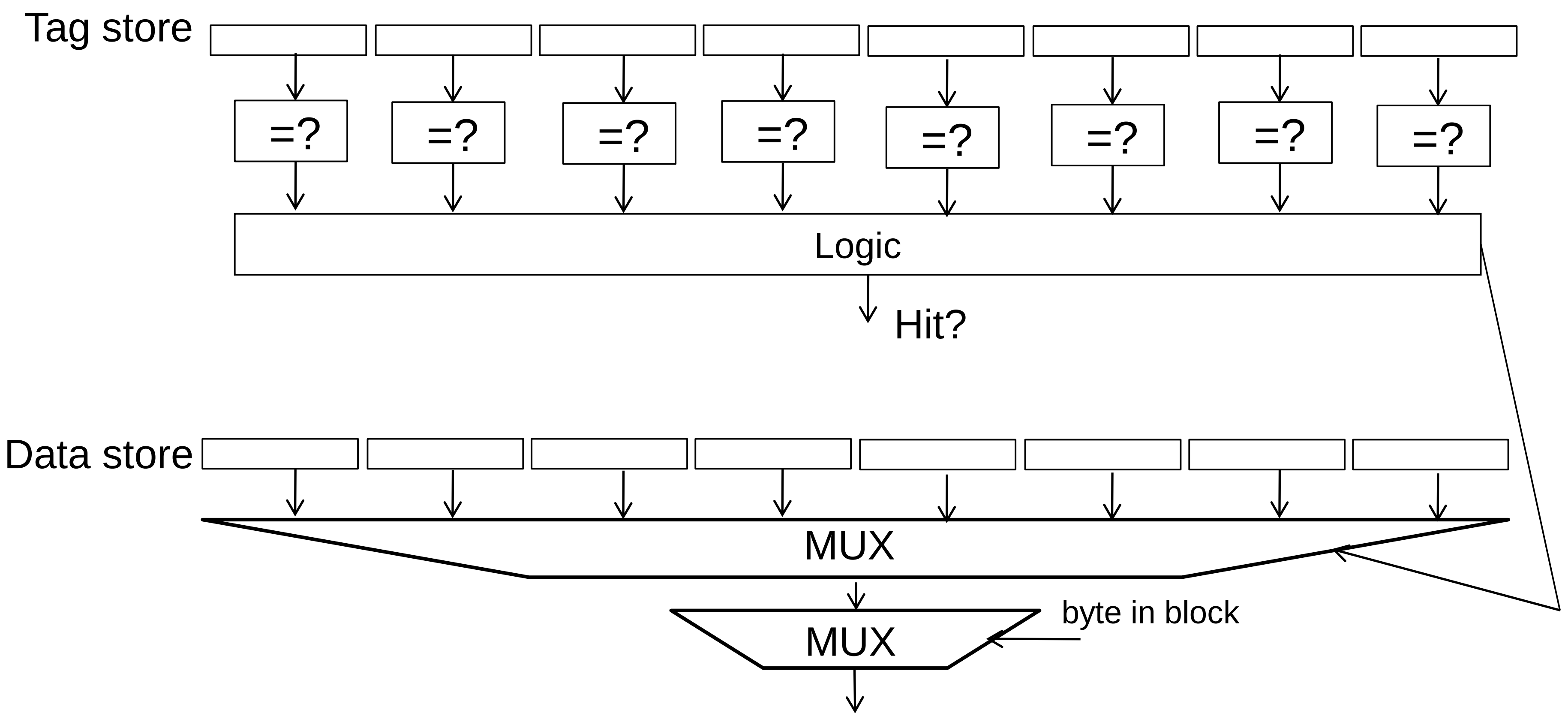


+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location

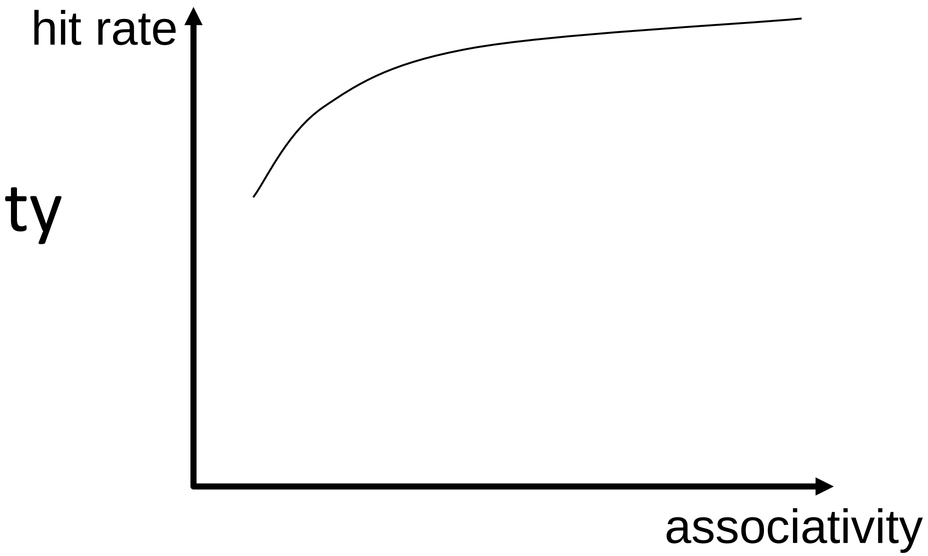


Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

- Higher associativity
  - ++ Higher hit rate
  - -- Slower cache access time (hit latency and data access latency)
  - -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Hybrid replacement policies
    - Optimal replacement policy?

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?

Approximations of LRU

- Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

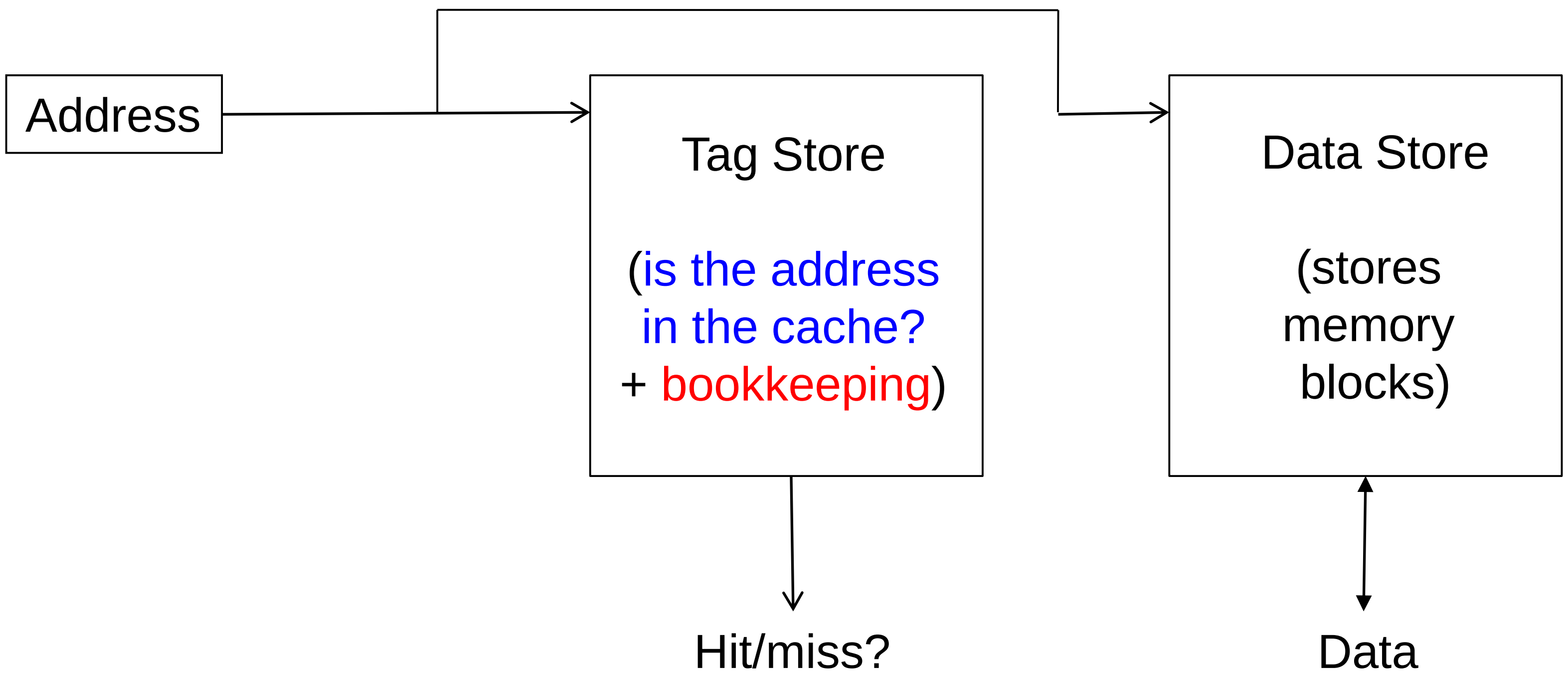
- Example:
  - Not MRU (not most recently used)

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

Handling Write Opertions

Recall: Cache Structure



What's In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back

  + Can combine multiple writes to the same block before eviction

    - Potentially saves bandwidth between cache levels + saves energy

  -- Need a bit in the tag store indicating the block is “dirty/modified”

- Write-through

  + Simpler

  + All levels are up to date

  -- More bandwidth intensive; no combining of writes

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss
  + Can combine writes instead of writing each of them individually to next level
  + Simpler because write misses can be treated the same way as read misses
  -- Requires transfer of the whole cache block

- No-allocate
  + Conserves cache space if locality of writes is low (potentially better cache hit rate)

Instruction vs. Data Caches

- Separate or Unified?

- Pros and Cons of Unified:

    + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)

    -- Instructions and data can thrash each other (i.e., no guaranteed space for either)

- First level caches are almost always split

- Higher level caches are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store accessed in parallel

- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially