Pipeline

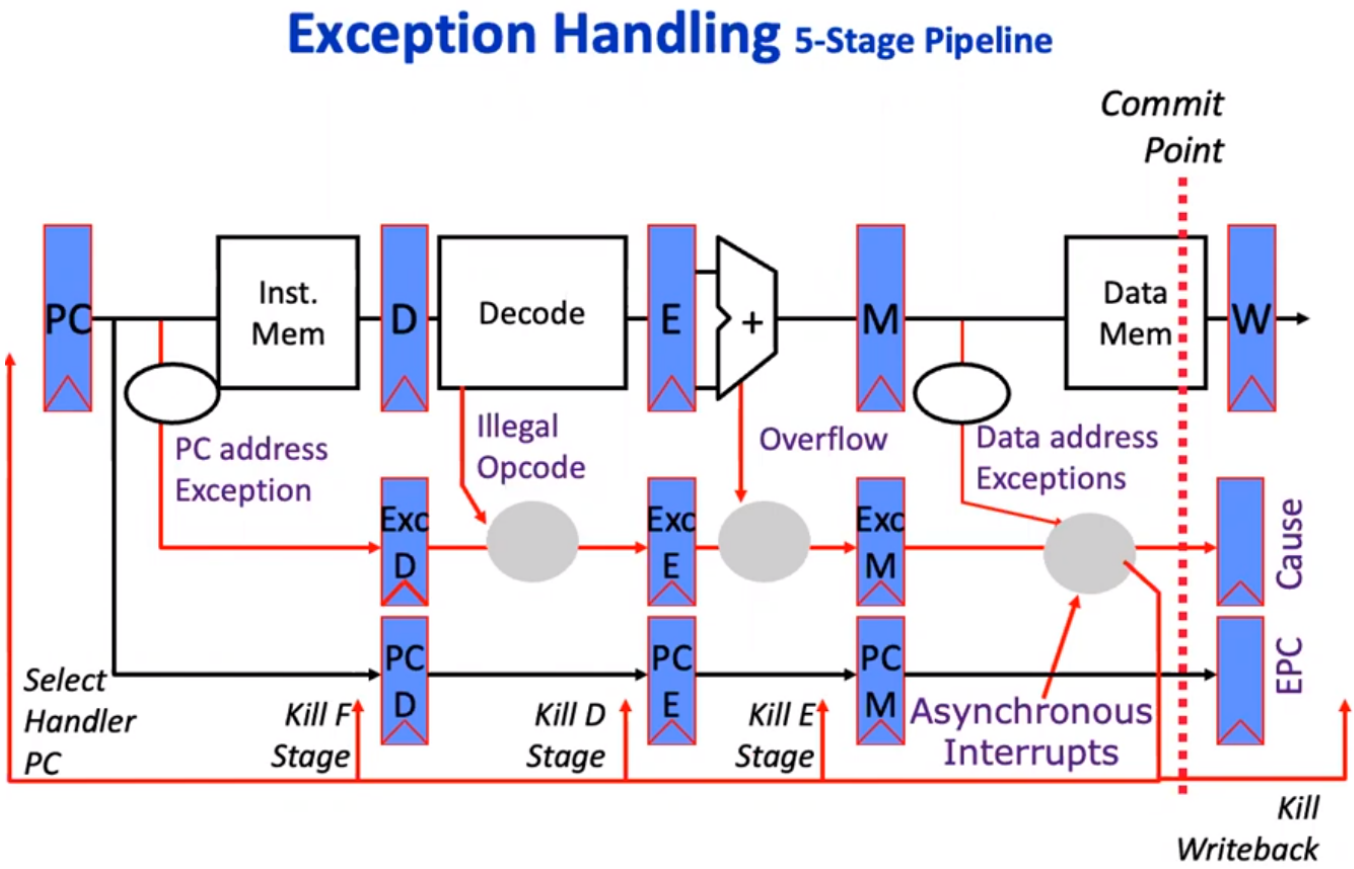
Resources

> HCU: https://www.cse.cuhk.edu.hk/~byu/CENG3420/2016Spring/
>
> UCB: https://people.eecs.berkeley.edu/~pattrsn/252F96/(More advance)
>
> UCB: https://inst.eecs.berkeley.edu/~cs152/sp23/

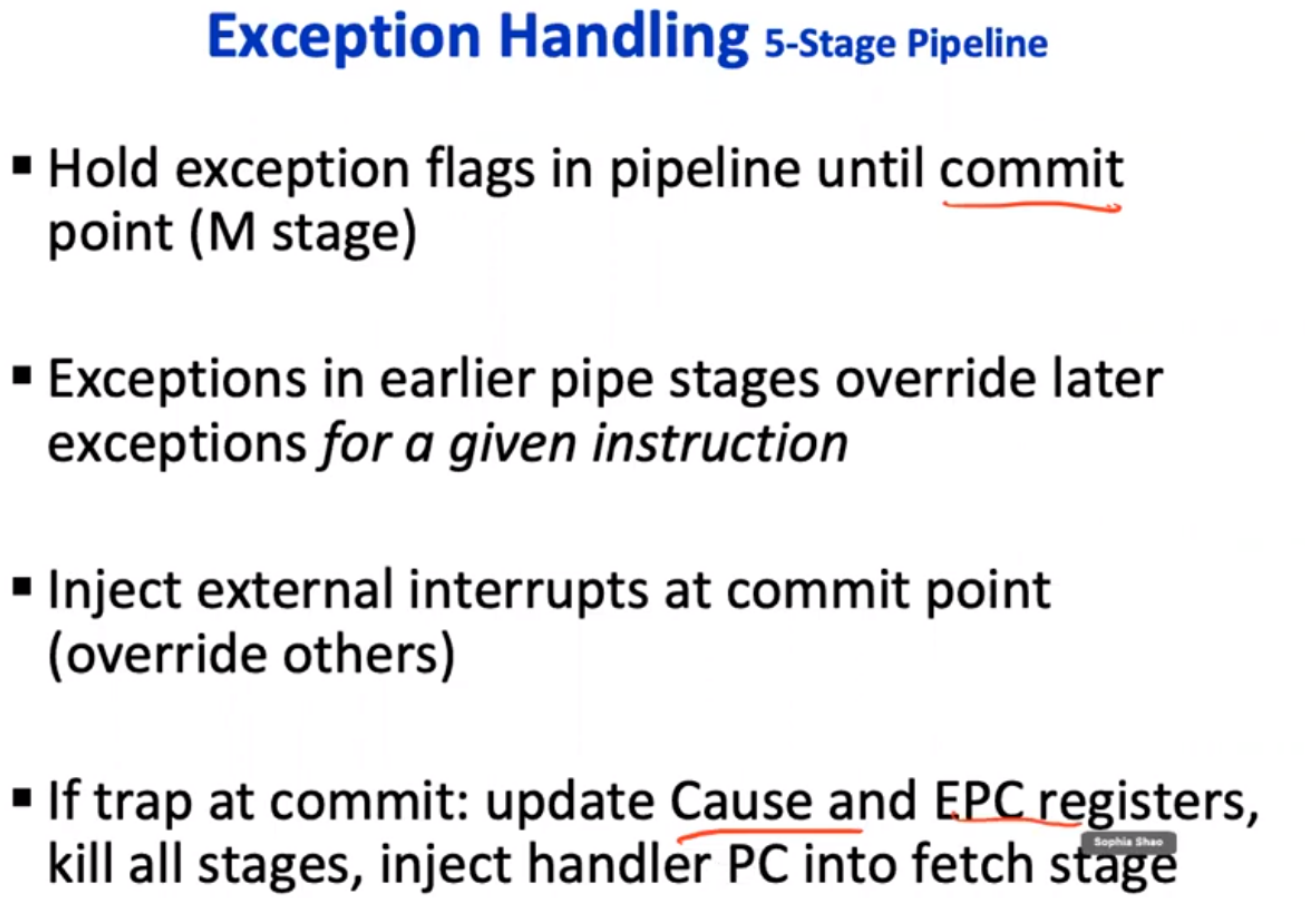
Instruction scheduling

> https://www.cs.cmu.edu/afs/cs/academic/class/15411-f20/www/lec/21-scheduling.pdf

Trap



Handling Principle



The principle of trap handling leverages the idea of speculation. Since the trap seldom occurs, we can reduce the overal latency by always predicting trap not to occur.

Instruction Pipelined Cache

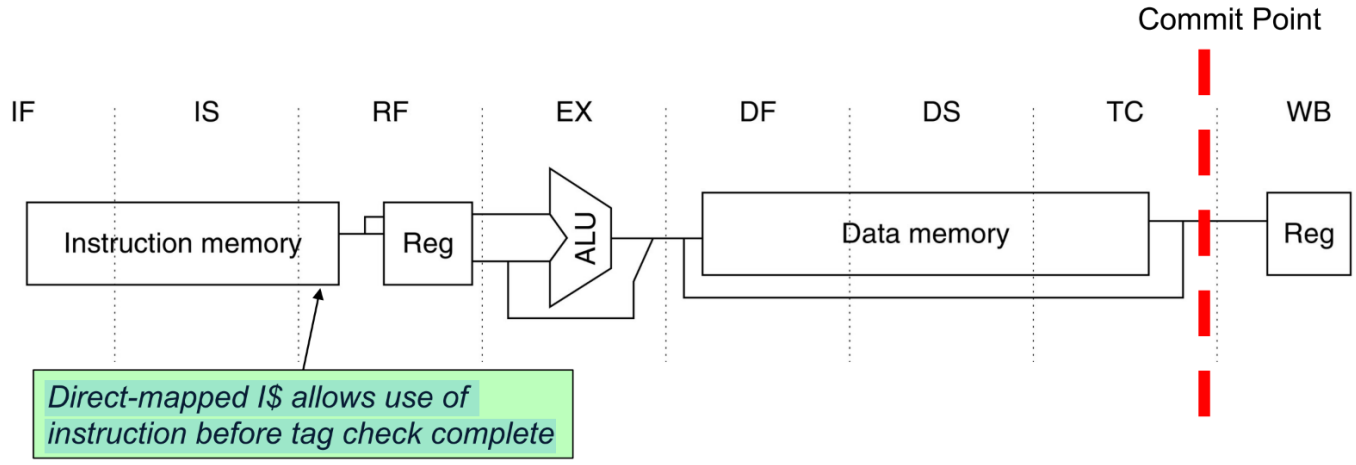


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

10

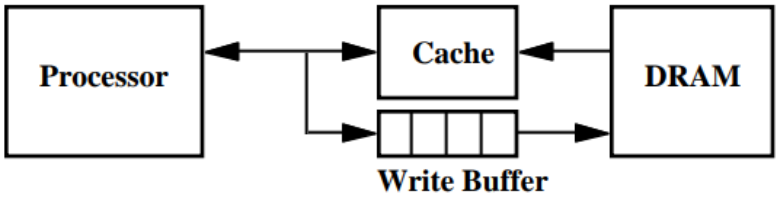
For instruction, we can directly use it before tag check is fully completed(RF stage in the figure), thus reduce the time of instruction fetch. However, if a cache miss occurs, we may need to flush the pipeline and reload the instrcution from the memory.

Branch Delay Slot

Cache Design and Buffered write

> https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture2a.pdf

Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

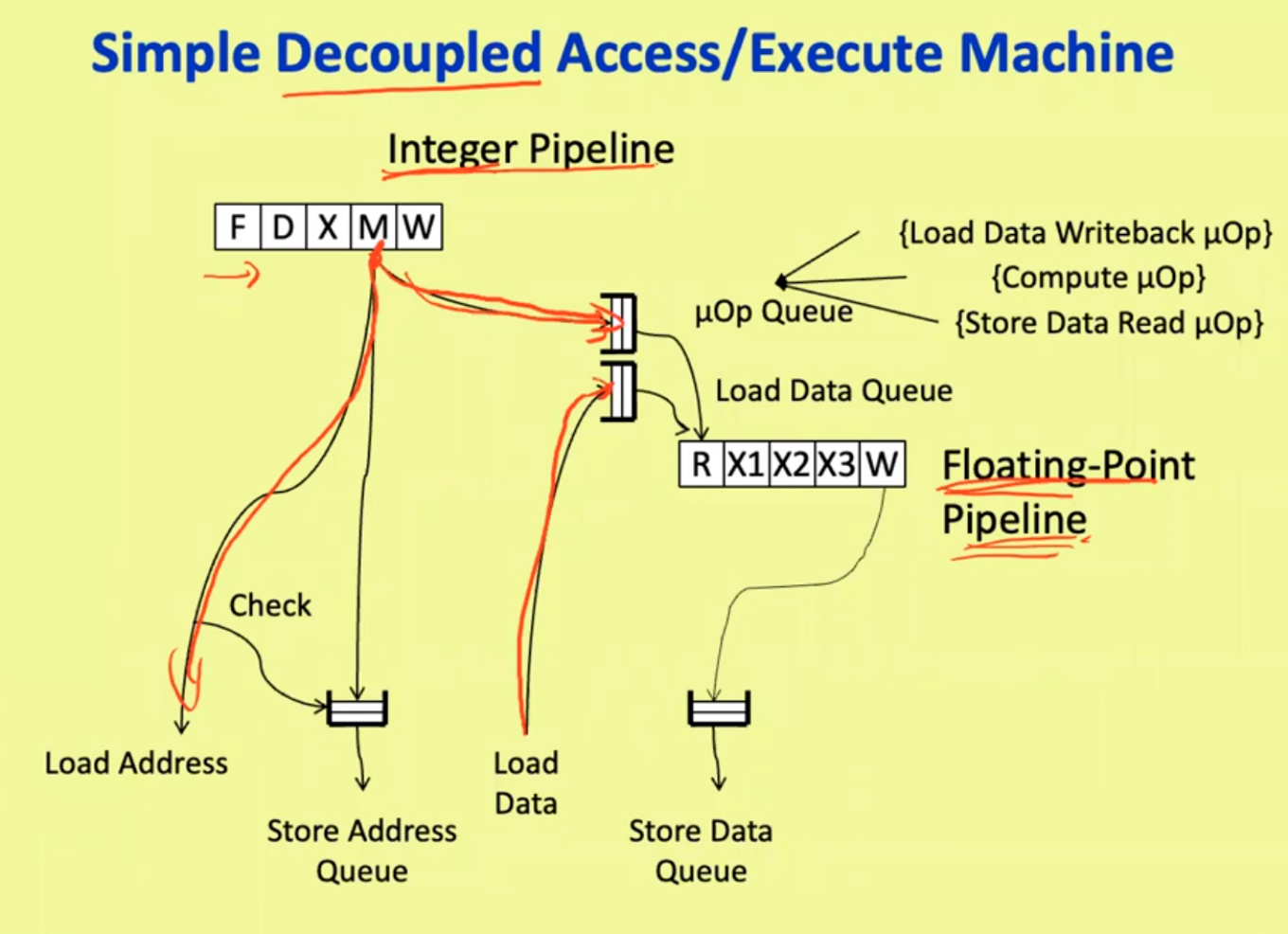
DAP.F96 49

Advance Pipelining

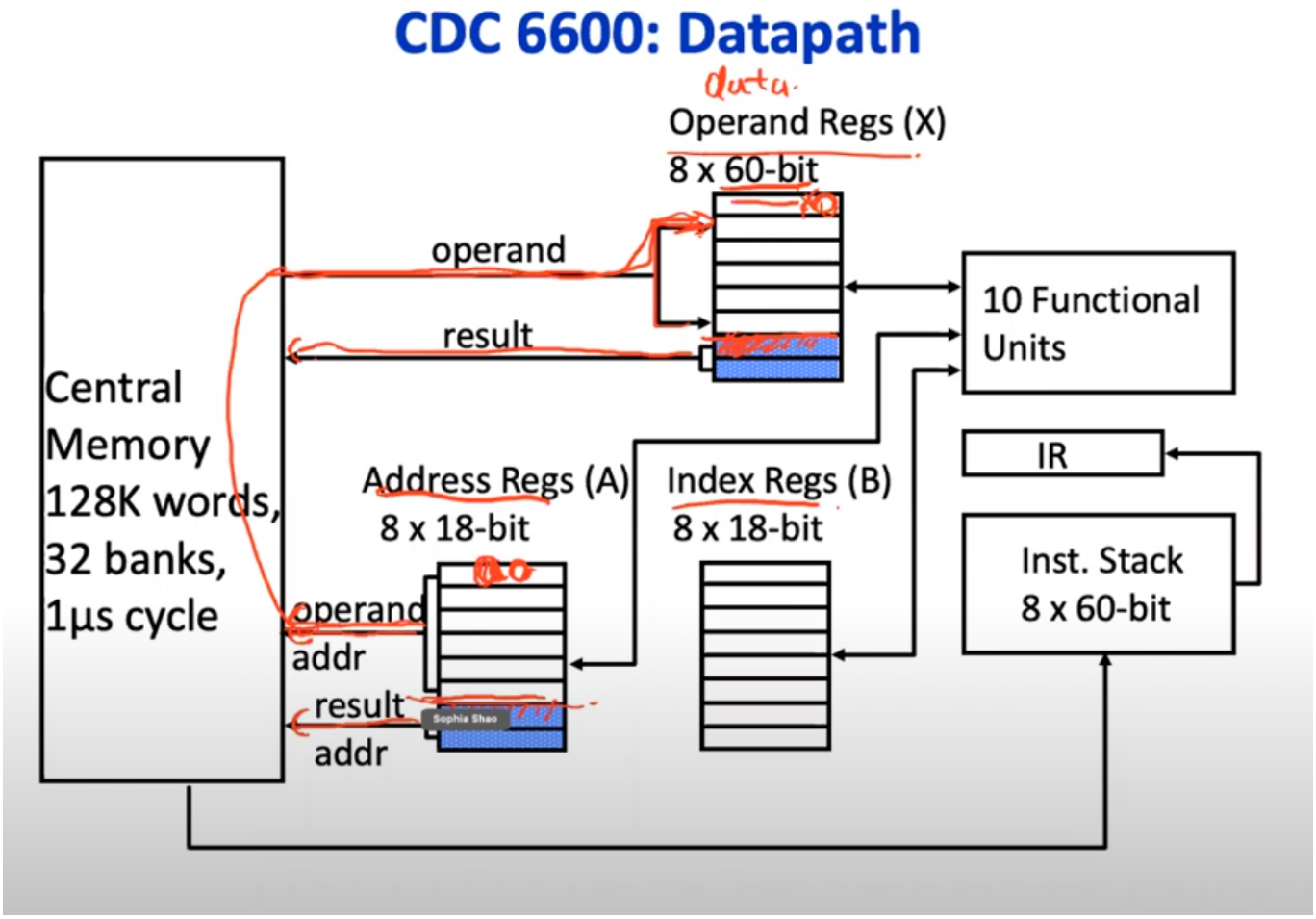
> https://drive.google.com/file/d/1-2SpAD-7hnqqxqLfQL9vjGWPc0okbenI/view

Decoupling in Pipeline

Use queues between "access" and "execute" pipeline to tolerate long memory lentency, thus accelerate the computation.



CDC6000 architecture


CDC 6600: Datapath
Operand Regs (X)
8 x 60-bit
operand
result
Central
Memory
128K words,
32 banks,
1µs cycle
10 Functional
Units
IR
Address Regs (A)
8 x 18-bit
Index Regs (B)
8 x 18-bit
Inst. Stack
8 x 60-bit
operand
addr
result
addr

Advantage

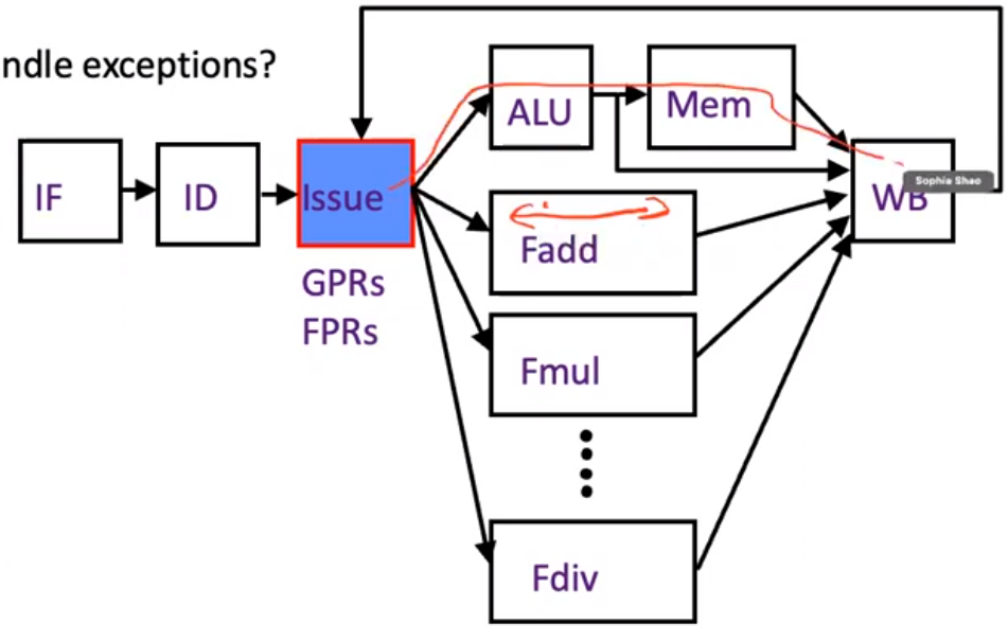
CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
  - Only 3-bit register-specifier fields checked for dependencies
  - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
  - Address update instruction also issues implicit memory operation
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions in between
- CDC6600 has multiple parallel *unpipelined* functional units
  - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs

Issues in Complex Pipeline Control

Issues in Complex Pipeline Control

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle
- Structural conflicts at the write-back stage due to variable latencies of different functional units -> many writes to reg file
- Out-of-order write hazards due to variable latencies of different functional units
- How to handle exceptions?



CDC6600's solution to the complex pipeline control: Scoreboard.

Insturction-Level Parallelism

- **Pipelining: executing multiple instructions in parallel**
- **To increase ILP**
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

Def: Expand each pipeline stage to accommodate multiple instructions.

Two-issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

"Static" typically means let's our compiler to take care of it.

"Dynamic" typically means let's build some hardware to take care of it.

Multiple Issue

Static multiple issue

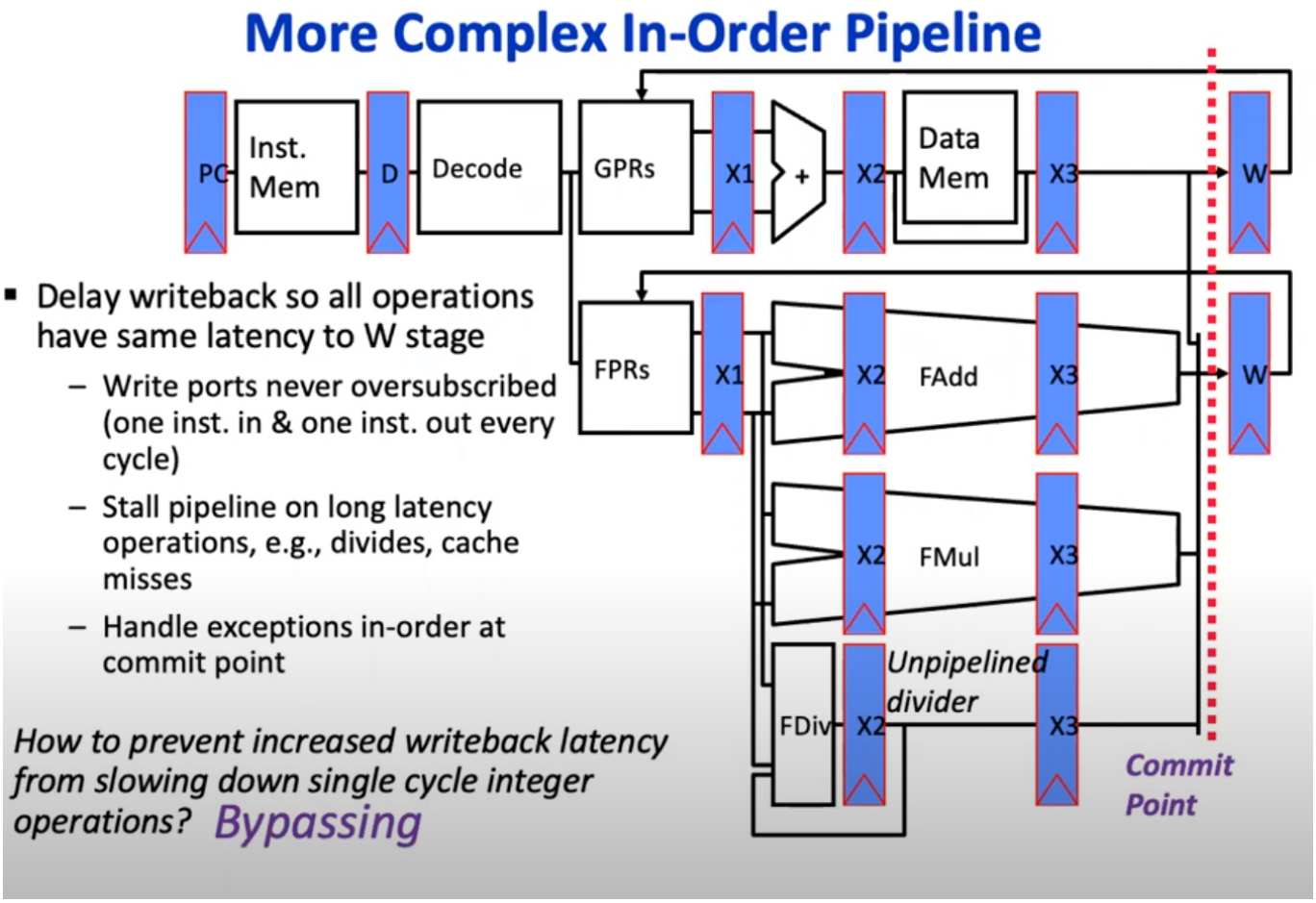
- Compiler groups instructions to be issued together
- Packages them into “issue slots”
- Compiler detects and avoids hazards

Dynamic multiple issue

- CPU examines instruction stream and chooses instructions to issue each cycle
- Compiler can help by reordering instructions
- CPU resolves hazards using advanced techniques at runtime

> https://www.cse.cuhk.edu.hk/~byu/CENG3420/2016Spring/L11_MIssue.pdf

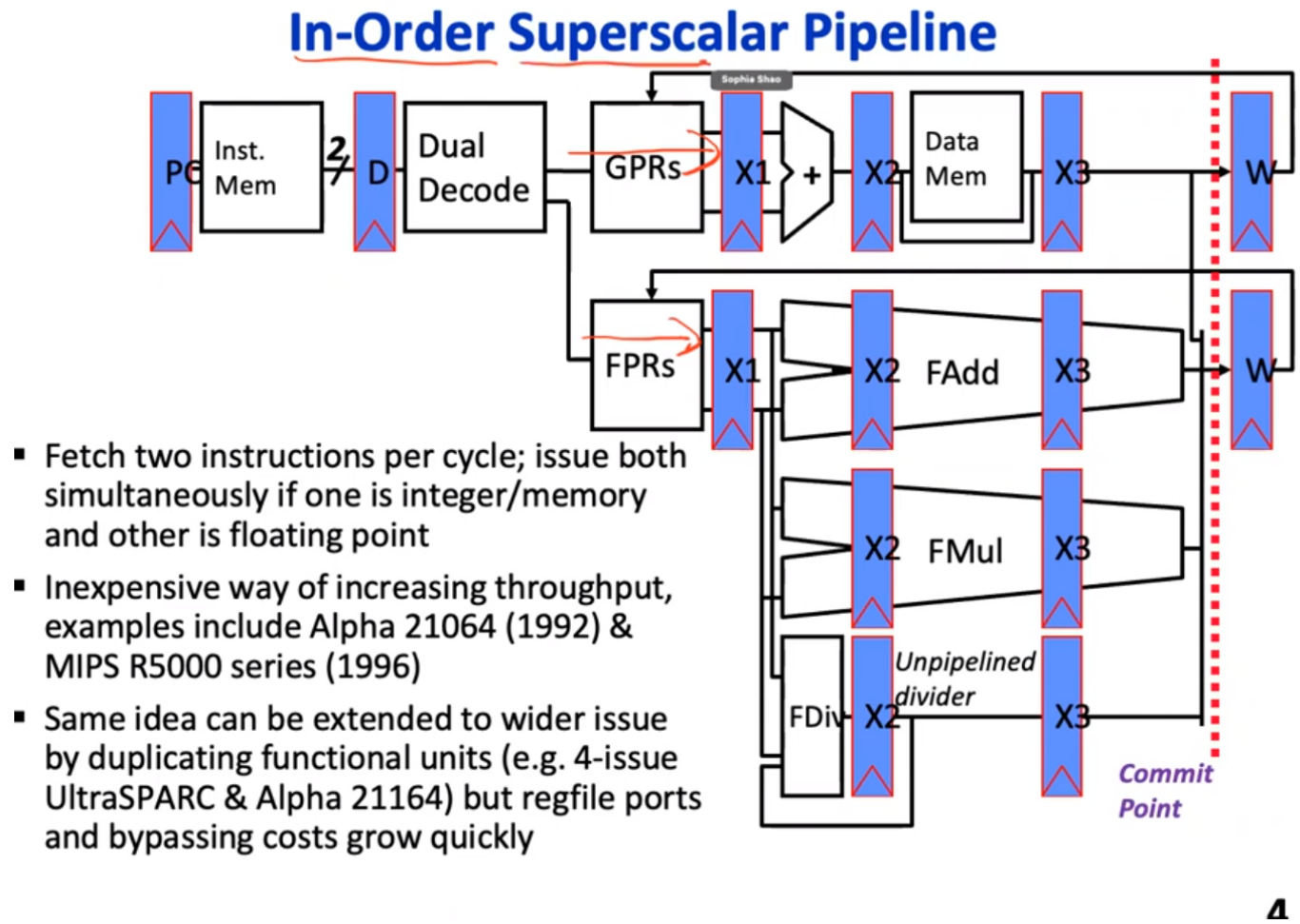
More Complex In-order Pipeline



- By delaying the commit point:

1. Write ports never oversubscribed.
2. We can stall the pipeline on long latency operations like cache miss.
3. Handle exceptions in-order at commit point.

MISD pipeline

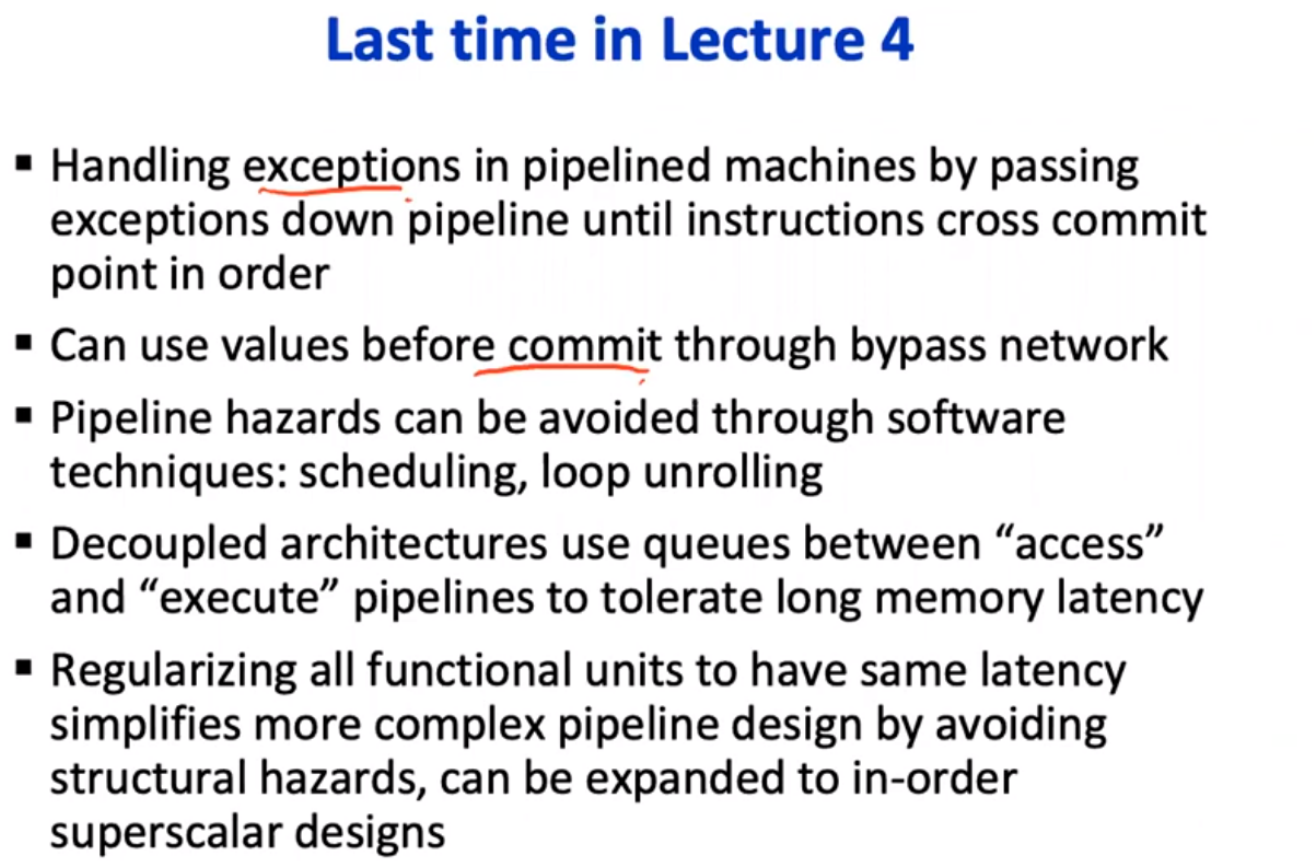


Further exploitation of paralleraism increases throughput. For instance, this architecture can fetches two instruction per cycle and issues multiple functional units, here a integer computation unit and a floating unit, in execution.

Architectural State

> https://en.wikipedia.org/wiki/Architectural_state#:~:text=Architectural%20state%20is%20the%20collection,registers%2C%20and%20the%20program%20counter.

A summary of pipeline



Microcoded vs Pipelined

one way to reduce hazard: Loop unrolling Alternative approach: Decoupling Execution

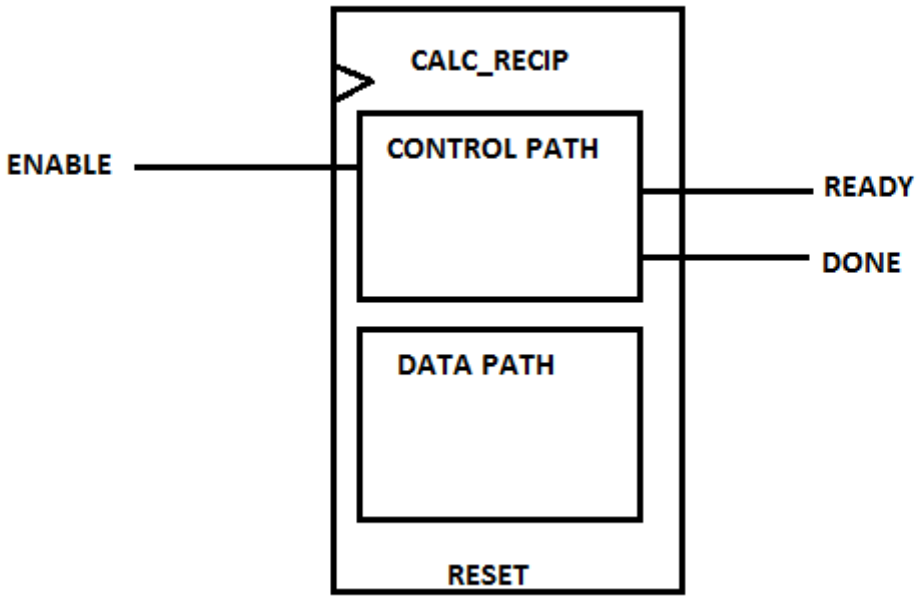
CDC6600 Scoreboard

- read-after-write out of order execution
- write-

Memory System

Prologue - Datapath and Controller

Some random thoughts, just in mind. I always curious about how memory is implemented, even though in the simplest form it just comprises of a pile of registers and maybe a mux for locating the storage unit. But that seems too simplistc, since technology varies, take DRAM as an example, since you have to initiate row access, col access and precharge operations one by one, state element is necessary. Meanwhile, to support a inter-component communication in precessor, like between CPU and main memory which works under different frequency, pure combinational design seems incapable.



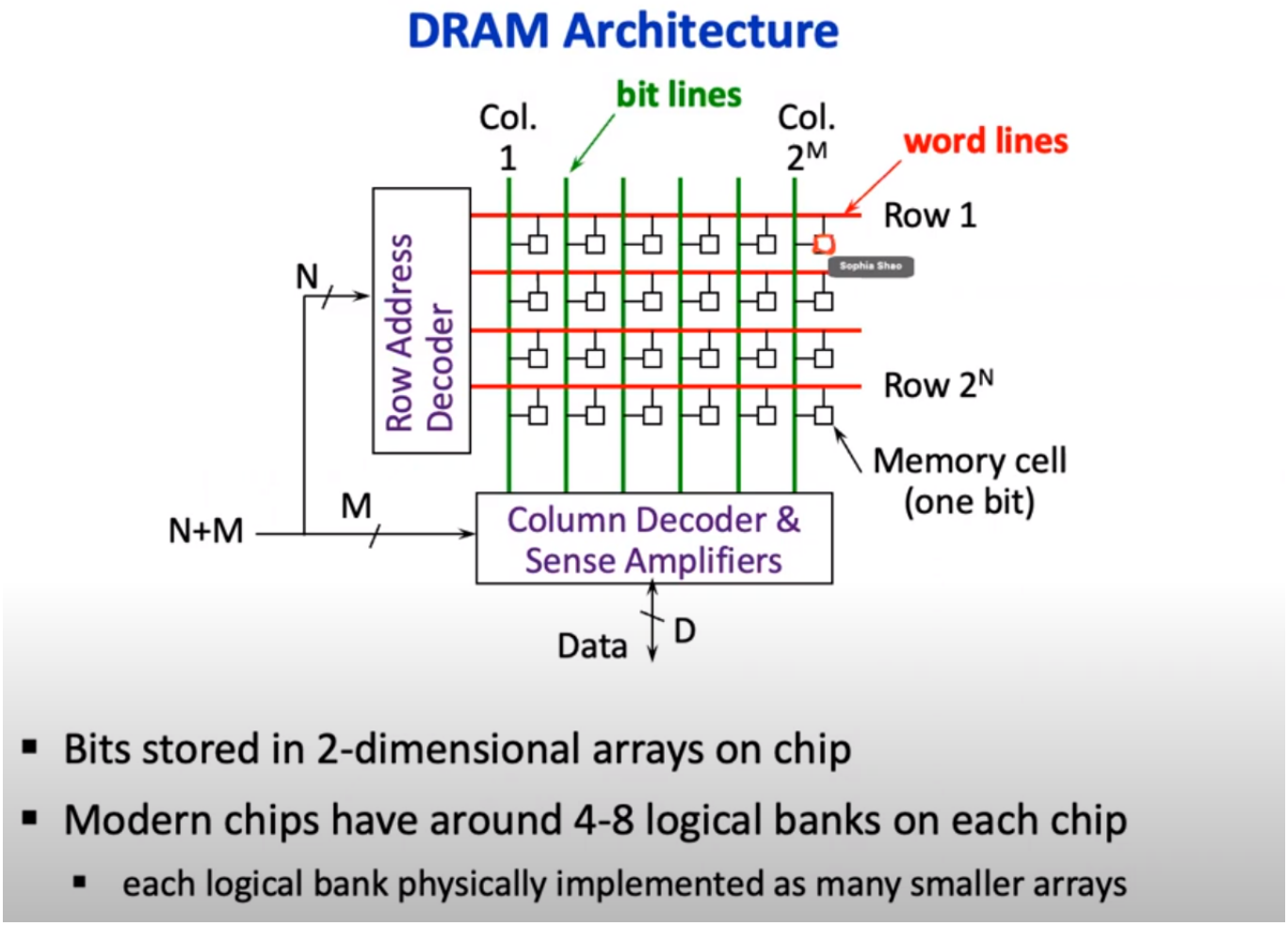
After some study, I realize that many functional modules, capable of executing a certain task and issue inter-module communication, can be conceptually divided into two part.

One is the datapath, responsible for performing arithmetic, logic, and data movement operations. It typically consists of registers, multiplexers, arithmetic logic units (ALUs), and other functional units. The datapath operates on data as it flows through the system, executing instructions and processing data according to the program's logic.

While the datapath may involve sequential operations, it is not usually implemented as a state machine. Instead, it consists of combinatorial logic circuits that operate based on the control signals provided by the controller.

The other is controller, which directs the operation of the datapath and manages the sequencing of operations. It generates control signals that determine the operation of the datapath components based on the current state of the system and the instruction being executed. The controller often implements a finite-state machine (FSM) or state machine design, where the system behavior is modeled as a set of states and transitions between states.

Dram



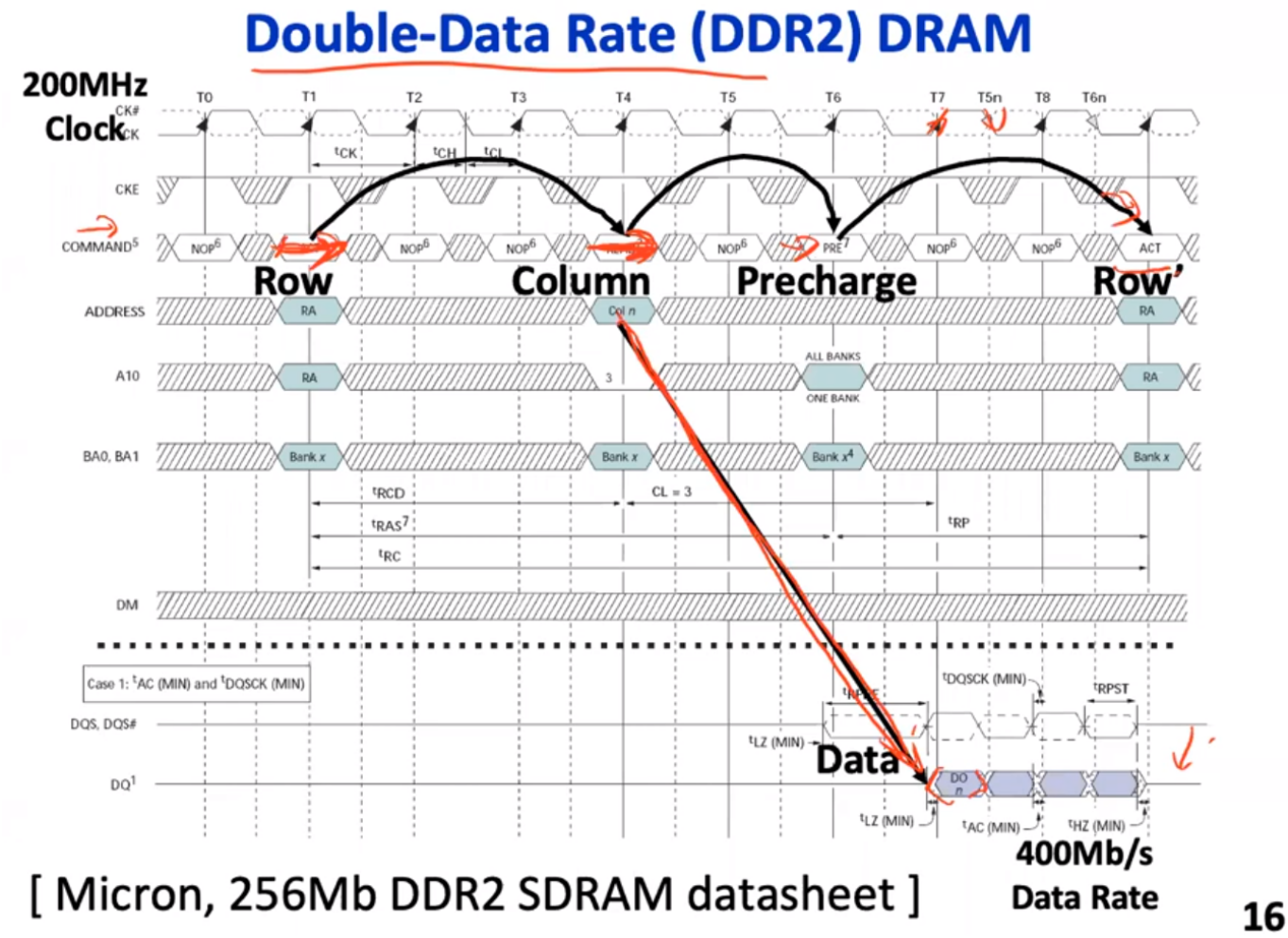
- In a row access, the sense amplifier "sense" the voltage change in each bitline and latch(or we say memorize) whole row of bits. Then a recharge is performed to restore the charge(state) in the storage cells.
- So sense amplifier bookkeeps the state in each storage unit(using some forms of memory element, like latches). In read operation, the selected column sends latched bit(stored in amplifier) out to the chip pins. And on write operation, we simply changes the value in latches to the required value.
- One bank manages one bit of data, and in one chip typically multiple banks are used(increase bandwidth).

Operations

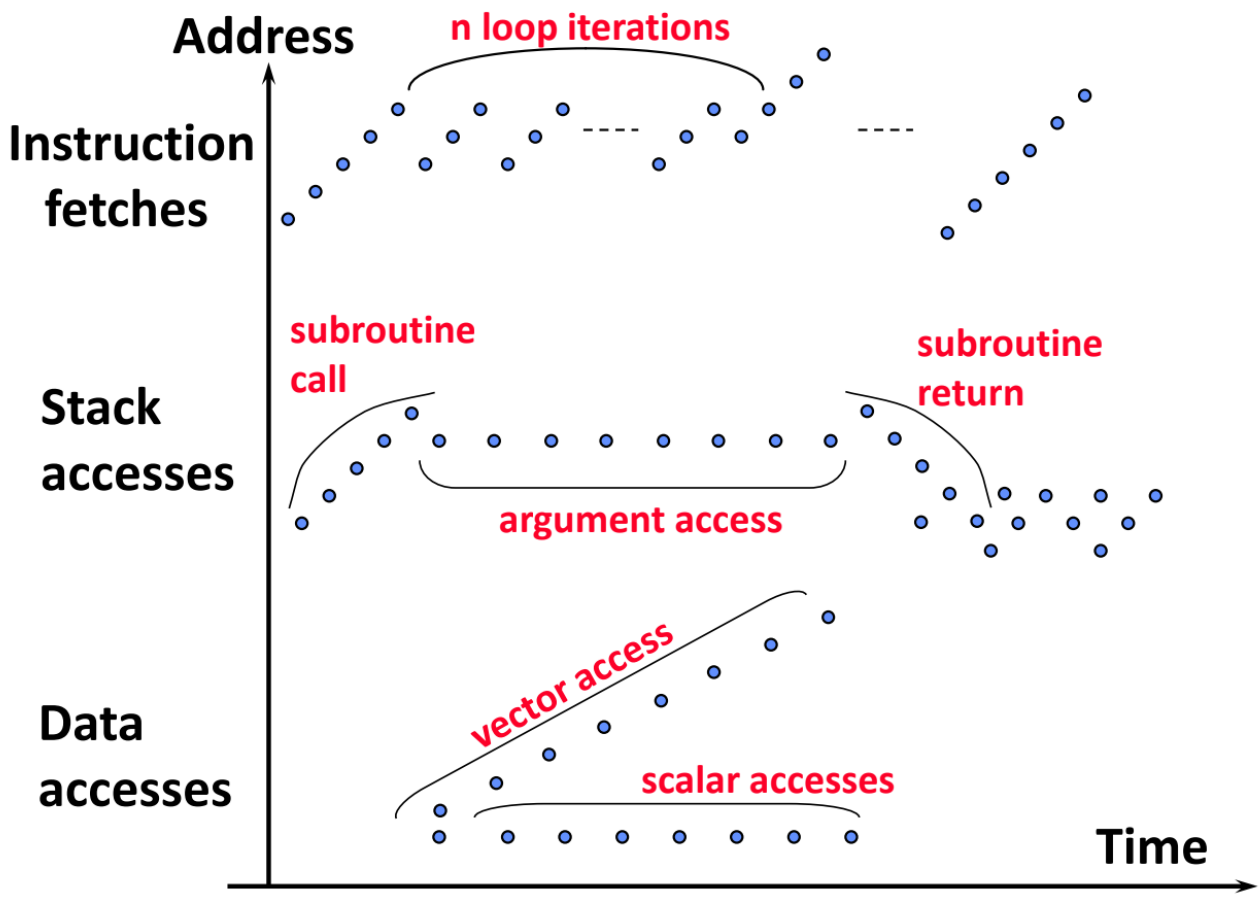
DRAM Operation

- Three steps in read/write access to a given bank
- Row access (RAS)
  - decode row address, enable addressed row (often multiple Kb in row)
  - bitlines share charge with storage cell
  - small change in voltage detected by sense amplifiers which latch whole row of bits
  - sense amplifiers drive bitlines full rail to recharge storage cells
- Column access (CAS)
  - decode column address to select small number of sense amplifier latches (4, 8, 16, or 32 bits depending on DRAM package)
  - on read, send latched bits out to chip pins
  - on write, change sense amplifier latches which then charge storage cells to required value
  - can perform multiple column accesses on same row without another row access (burst mode)
- Precharge
  - charges bit lines to known value, required before next row access
- Each step has a latency of around 15-20ns in modern DRAMs
- Various DRAM standards (DDR, RDRAM) have different ways of encoding the signals for transmission to the DRAM, but all share same core architecture

Working State



Memory Access Partten



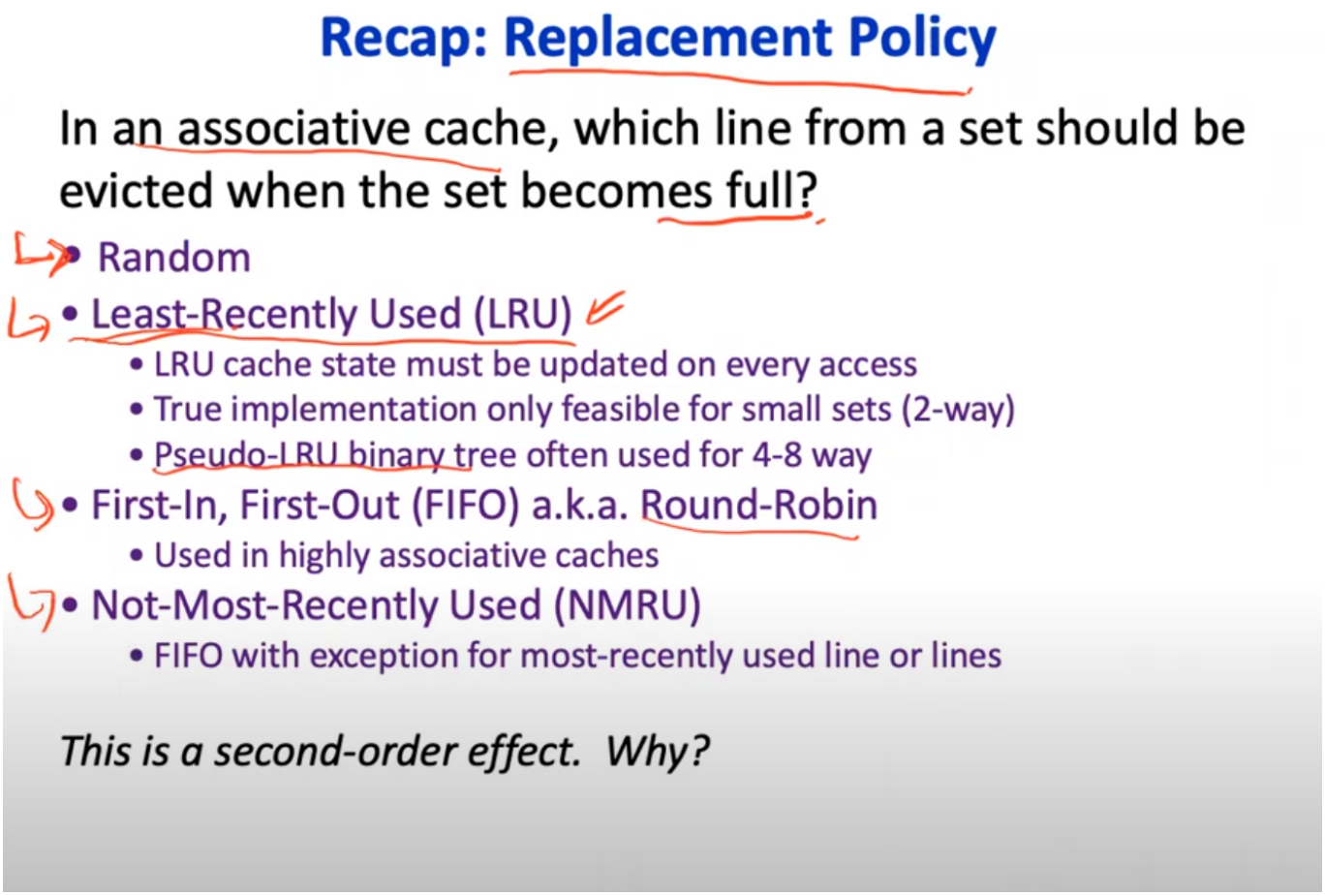
Q:Why DM is faster than FA?

It's my own opinion. In direct-mapped cache, to fetch a certain data, we first decode the address the select the cache line, then do the tag comparison and read the data. Here the tricky stuff is that we can do tag check and read in parallel!

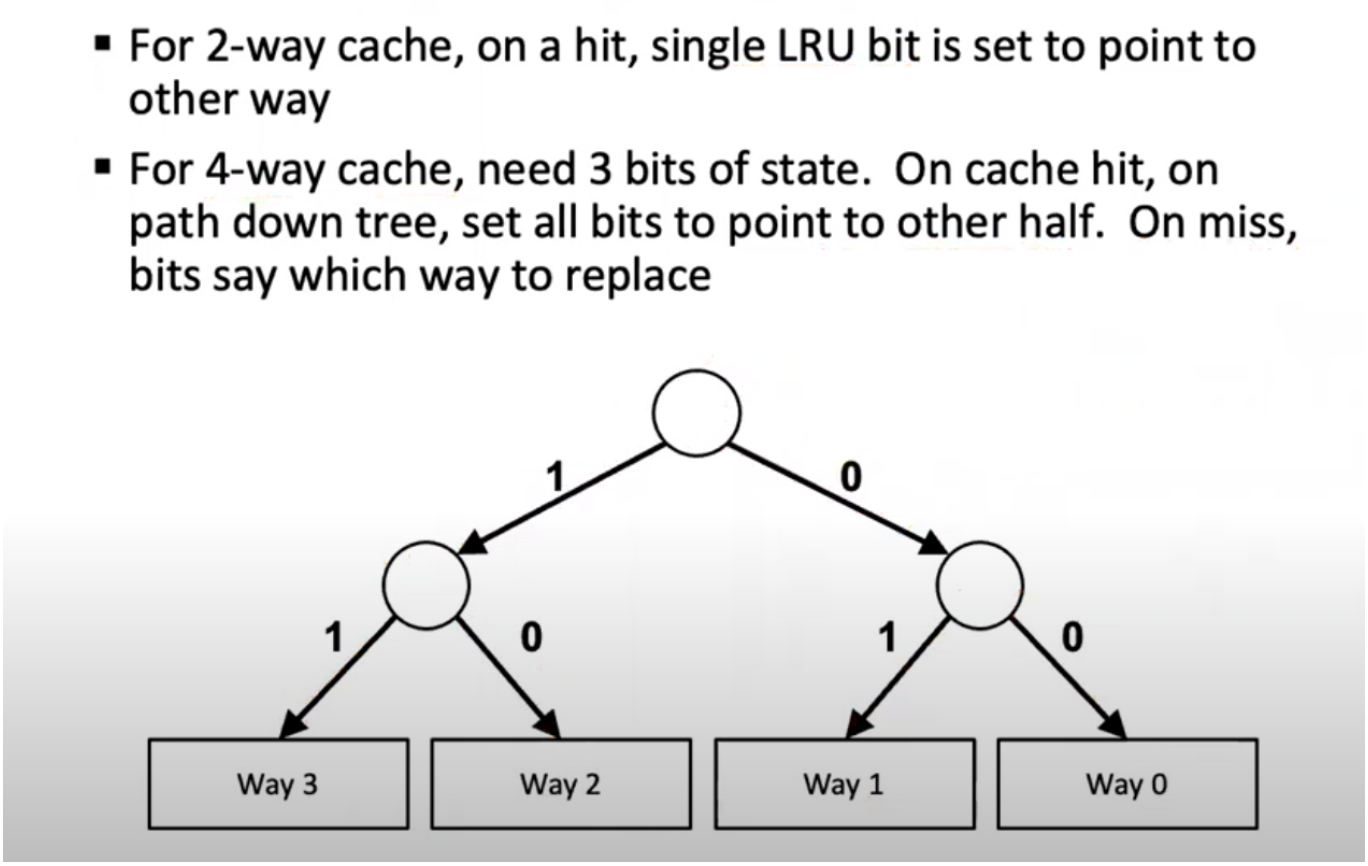
But for fully-associative cache, we don't possess such parallelism, since before tag check we never know where the data is, so we cannot do them in the same time.

Cache Optimization

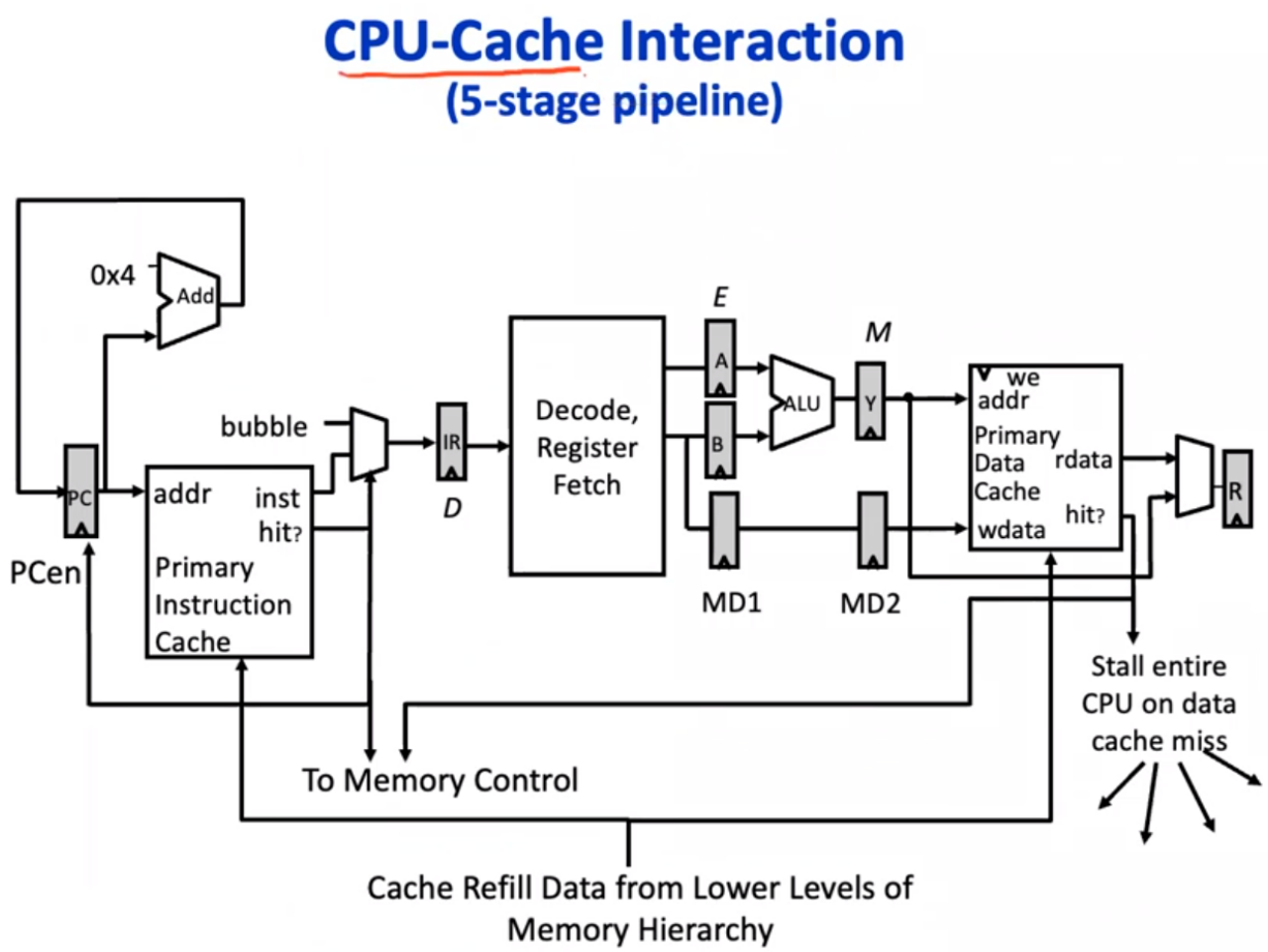
Cache Replacement Policy



- Since LRU needs more hardware to implement, especially for multiple ways senario, we won't use it in highly associative caches, but instead using Round-Robin policy.

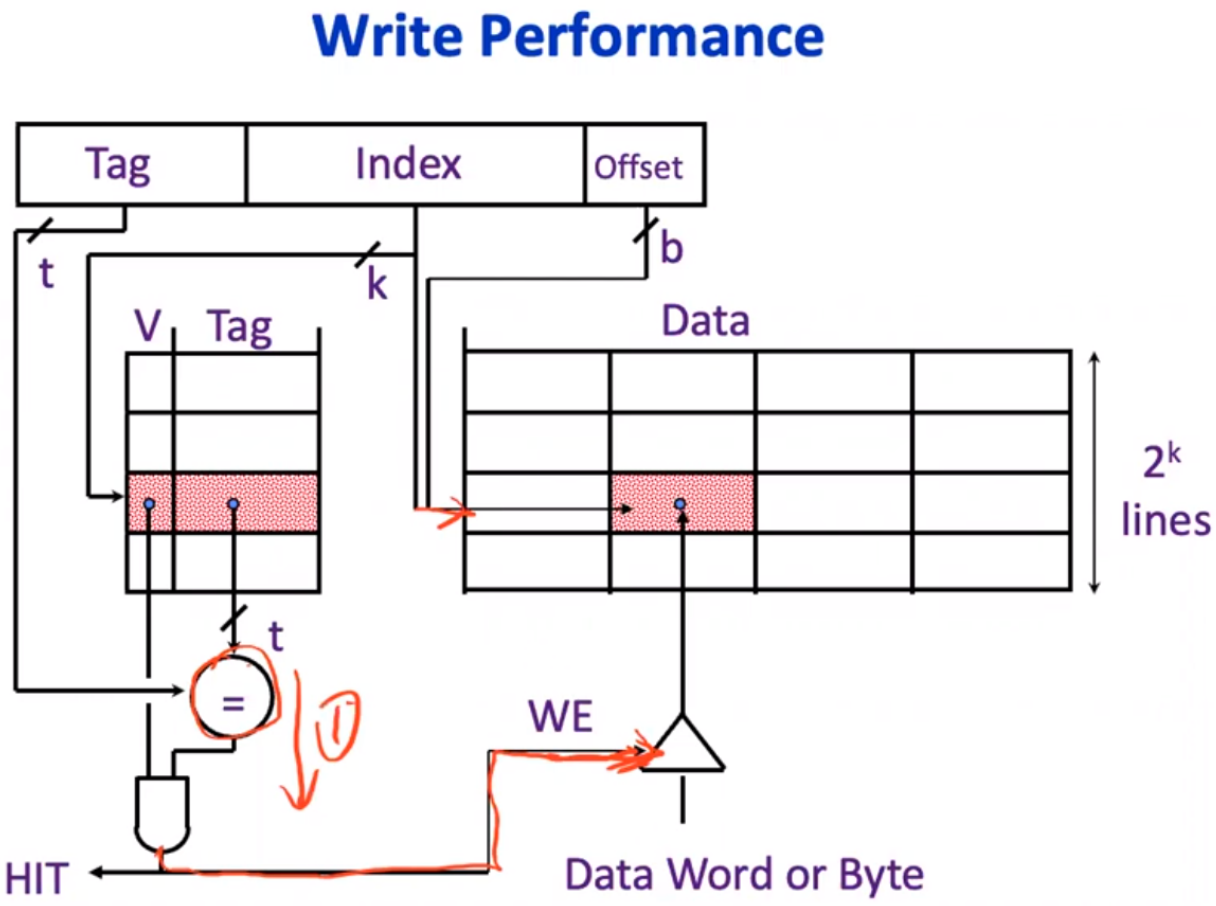


Cache-Pipeline Interaction



> One thing that really troubles me for weeks that I don't really understand the details of cache miss, of its influence on the pipeline execution, of the intricate interaction between pipeline, cache, and memory.
>
> So far we already know that the time for read/ write memory takes much longer time than single CPU cycle, which indicate that when a cache miss happen, we need time for the memory access to finish. But since I don't know the specific implementation, I wonder in real world design whether a pipeline will be stalled or we simply expand the time of each pipeline stage, which now I see as the most inefficient, since it make the cache useless.
>
> Now I still have other questions, like how cache "know" the memory is ready, how data is transferred bewteen cache and memory(may be there are some techniques like buffer) and how cache communicate with the pipeline, since I now think CPU and memory must work under different frequencies, thus we need extra hardware to support asynchronous, inter-commponent commnication. Lucily, in EECS151 I see a promising approach that leverage the handshake protocal, fifo, and state machine, which I might cover in this article somewhere.

Cache Structure and Multicycle Issue



Unlike instruction cache where we only perform read, we cannot write data cache before tag check is done, since once it is written, I think, it will contaminate the cache and it's impossibly hard to recover the state if the tag is finally invalid.

One thing to be mention is that the introduction of cache has implicitly change the write stage into two cycle , one for tag check, and one for data write if hit, which might prolong the execution time for single instruction. To solve this issue, we have several techniques:

Reducing Write Hit Time

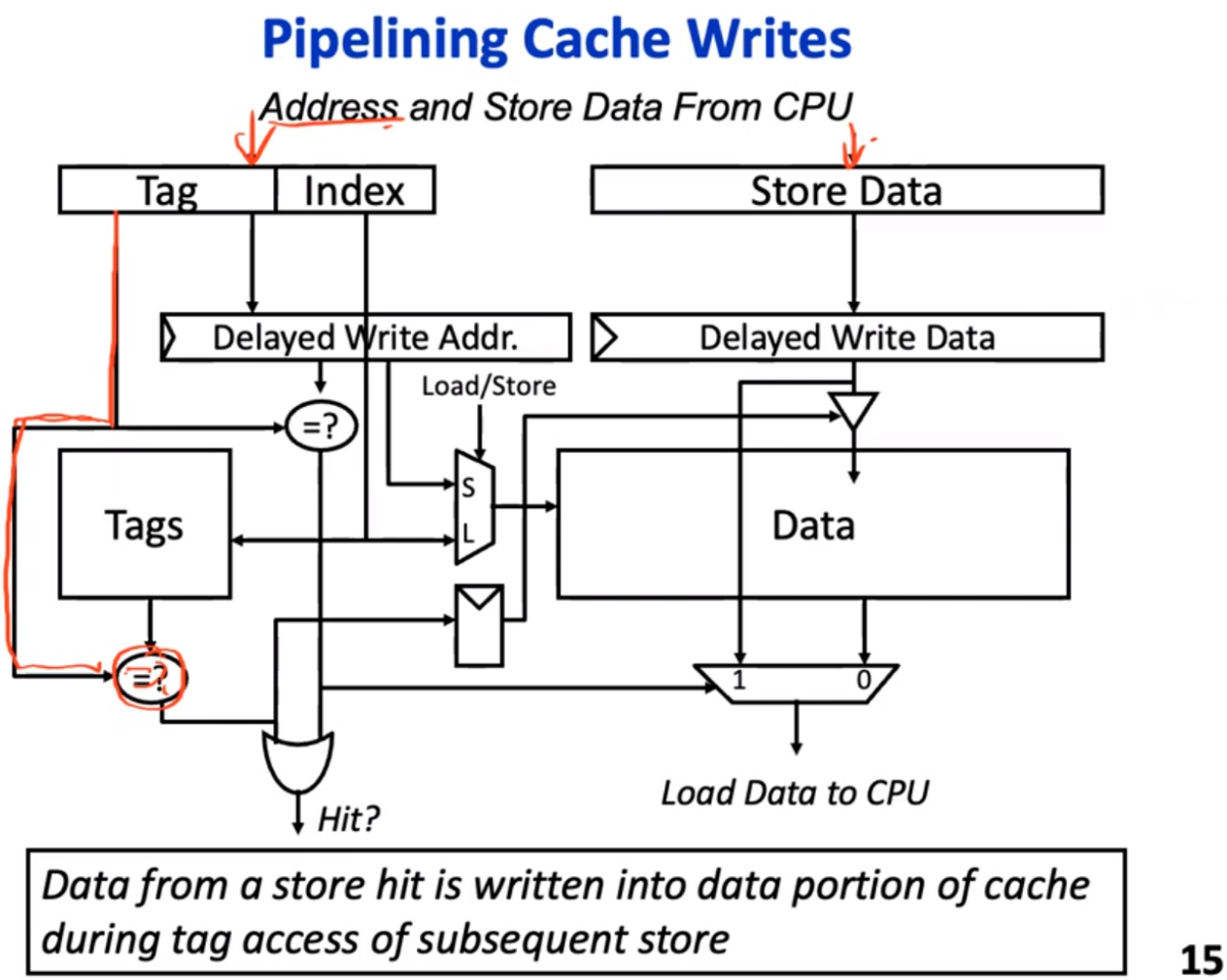
Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

Solutions:

- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store's tag check
- Fully-associative (CAM Tag) caches: Word line only enabled if hit

Pipelining Cache Write

Since it takes two consecutive cycles for a write operation, we can somehow pipeline the cache to optimize the throughput.



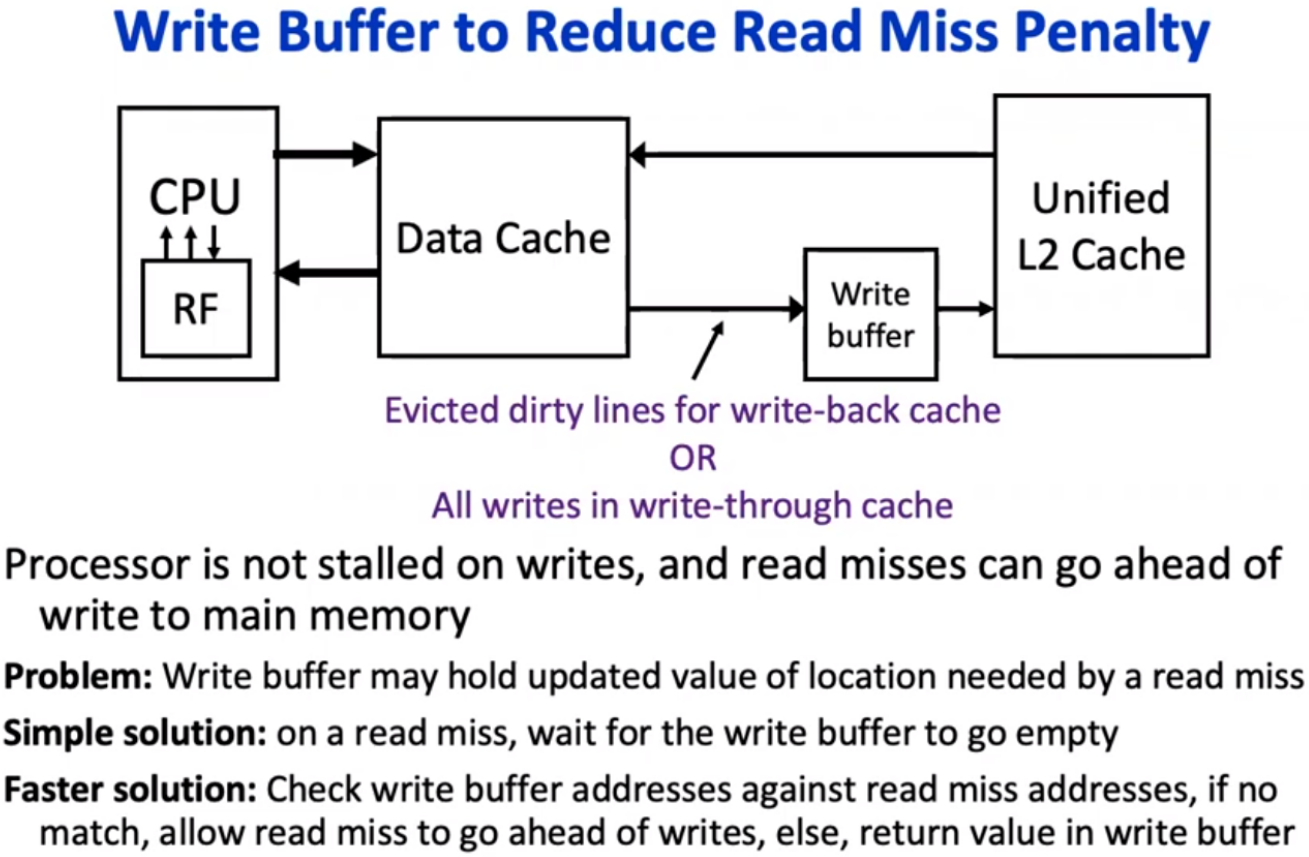
The figure above shows the pipelined cache implementation. In the first stage, we compare

the tag and load data & address into a buffer (here a register to delay operation into the next cycle). In the next cycle, we write the data if the tag is valid.

Dependency Issue: Read after Load

The introduction of pipelining adds to the complexity of the cache, which is likely to cause some harzard. For example, in the pipelined cache shown above.

Now let's assuming executing a store and a load to the same address subsequently. Here we see the stored data is directed into the delaying buffer, then when the read is performed, in the second cycle, the correct data, I mean the data at that address after the execution of the load operation (we always assume that the former instruction has been finished), however, is still in the buffer, not in the memory.

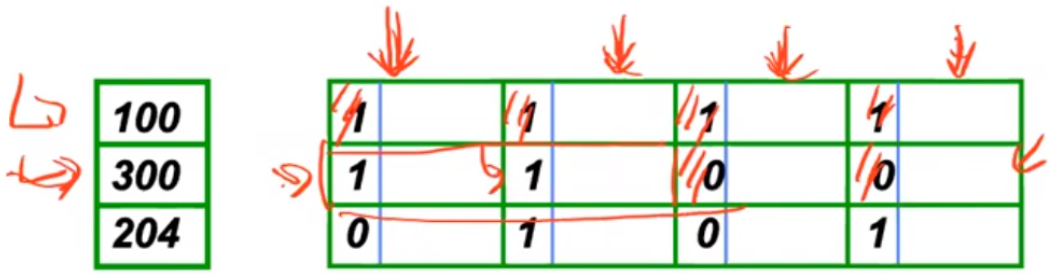


Therefore, remember for a read operation we can fetch the data before the tag check is done(in case of tag invalid we simple restore the state somehow), we shall look up both the memory and buffer for the data. If it is in the buffer, we can fetch it simply and nothing new happens (maybe, I don't know now).

Reduce Tag overhead

Reducing Tag Overhead with Sub-Blocks

- **Problem**: Tags are too large, i.e., too much overhead
  - Simple solution: Larger lines, but miss penalty could be large.
- **Solution**: Sub-block placement (aka sector cache)
  - A valid bit added to units smaller than full line, called sub-blocks
  - Only read a sub-block on a miss
  - *If a tag matches, is the word in the cache?*

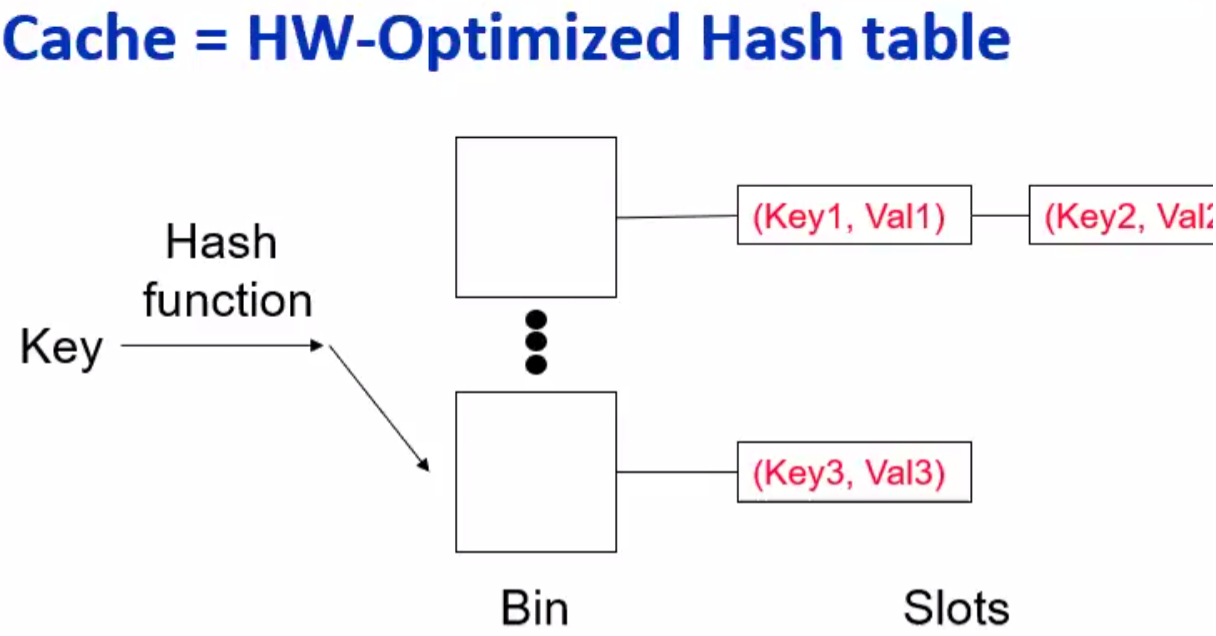


If tag is too large, it may takes much time for tag check. To reduce the tag overhead, we can simply make our cache line holds more data, thus shorten the tag. However, it introduces another problem, since read a whole bunch of cache line from memory is painful if cache line is large.

Therefore, we employ the idea of sub-blocks. Here on each cache miss, we no longer load the whole line of data from the memory, instead just a sub-block, thus decrease the miss penalty.

We shall also add an extra valid bit to those sub-blocks to indicate whether the sub-block is valid or not.

Cache - Hashmap Analogy



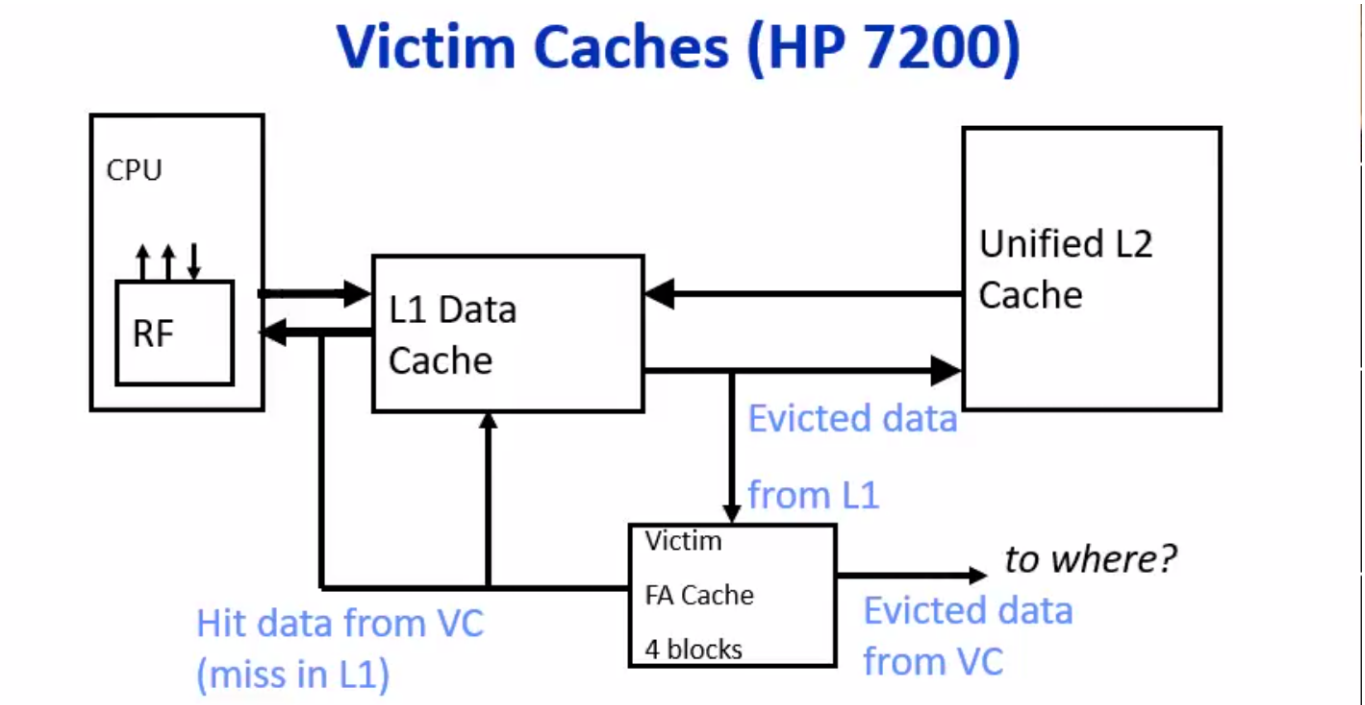
- Bin == Set, Slots == Ways
- 1 Bin → Fully associative; 1 Slot / Bin → direct mapped
- M Slots / N Bins where M, N > 1 → set associative

- Hash function: take bits of the address (“Index bits”)
- Fixed # Slots per Bin; Slots read out in parallel
- Key/Value pairs in Bins stored separately (tag + data array)

Victim Cache

Direct-mapped caches have faster access time than set-associative caches. However, in direct-mapped caches, when multiple cache blocks in memory map to the same cache line, they end up evicting each other whenever one of them is accessed. This issue, known as the cache-conflict problem, arises due to the limited associativity of the cache.

The introduction of victim cache exploits the both advantages of DM and FA, decrease the hit time while effectively reduce the miss penalty.



Victim cache is a small associative backup cache, added to a direct-mapped cache, which holds recently evicted lines

- First look up in direct-mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim->?

Fast hit time of direct mapped but with reduced conflict misses

> Wikipedia: https://en.wikipedia.org/wiki/Victim_cache#:~:text=A%20victim%20cache%20is%20a,3%20or%20Level%204%20caches.

Due to the problem of cache-conflict problem, we introduce the victim cache, which is placed between a direct-mapped cache(L1 here) and a highly associative (also larger) cache. When the line in the direct-mapped cache is evicted, instead put it in the the lower level of cache immediately, we can temporaily place them in a victim cache.

Let's look at an example, when a line has just been evicted is accessed. Since without victim cache we need to come into the next level of cache, the miss penalty is multiple cycle. But with a victim cache, we can simply look at victim cache immediately after cache miss, which takes only one cycle. We can also view a set in the victim cache serves as a backup set in the L1 cache, seemingly adding its associativity.

Besides, time to loop up the victim cache is relatively short, hence we can consult the victim cache before look up in the L2 cache.

Phrased Cache and Way Prediction

> https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=799456

> The paper above proposes a new approach using way prediction for achieving high performance and low energy consumption of set-associative caches. By accessing only a single cache way predicted, instead of accessing all the ways in a set, the energy consumption can be reduced.

Phrased Cache

The energy consumption of set-associative cache tends to be higher than that of direct-mapped cache, because all the ways in a set are accessed in parallel although at most only one way has the desired data. To solve the energy issue, Hasegwa et al. proposed a low-power set-associative cache architecture [5], which is referred to as phased cache in this paper. As shown in Figure l(b), the phased cache divides the cache-access process into the following two phases:

- First, all the tags in the set are examined in parallel, and no data accesses occur during this phase.
- Next, if there is a hit, then a data access is performed for the hit way.

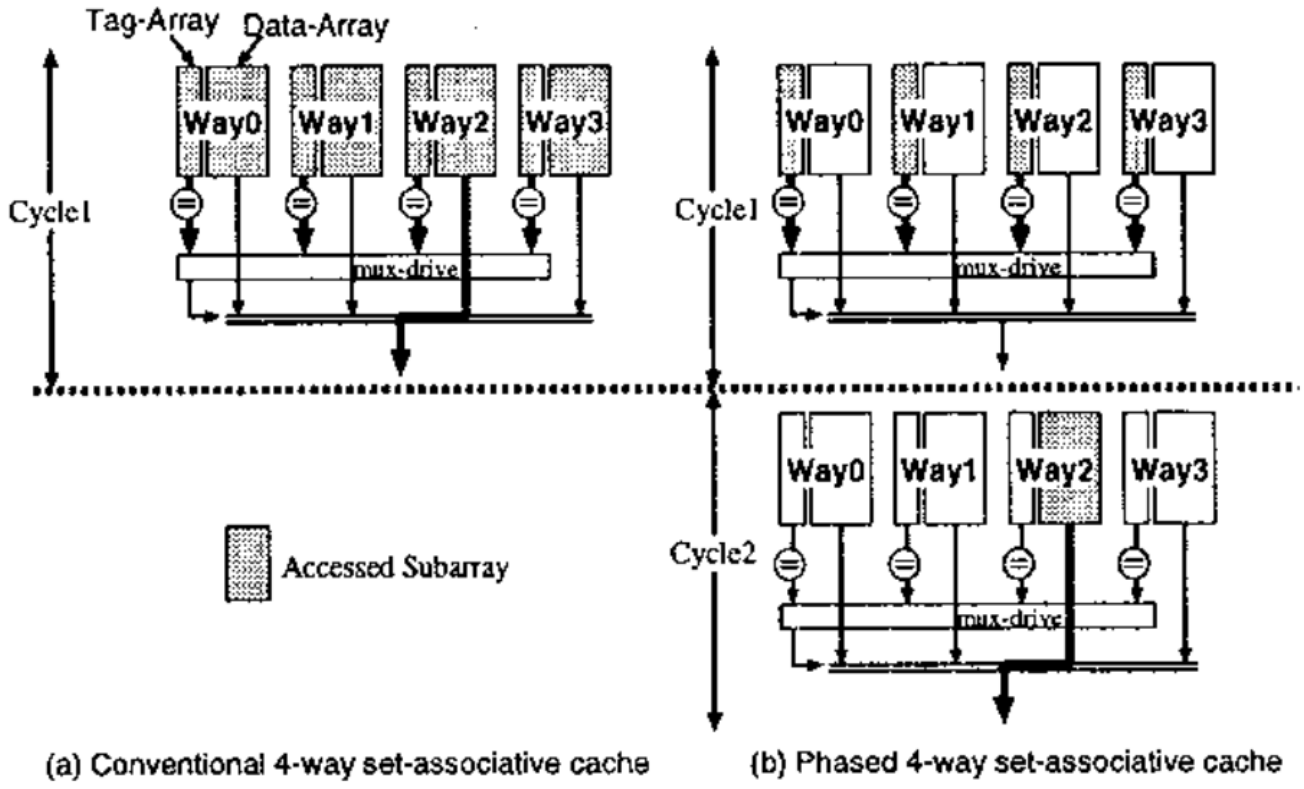


Figure 1: Phased Set-Associative Cache

Way Prediction

The way-predicting cache speculatively chooses one way before starting the normal cache-access process, and then accesses the predicted way as shown in Figure 2(a).

- If the prediction is correct, the cache access has been completed successfully.

- Otherwise, the cache then searches the other remaining ways as shown in Figure 2(b).

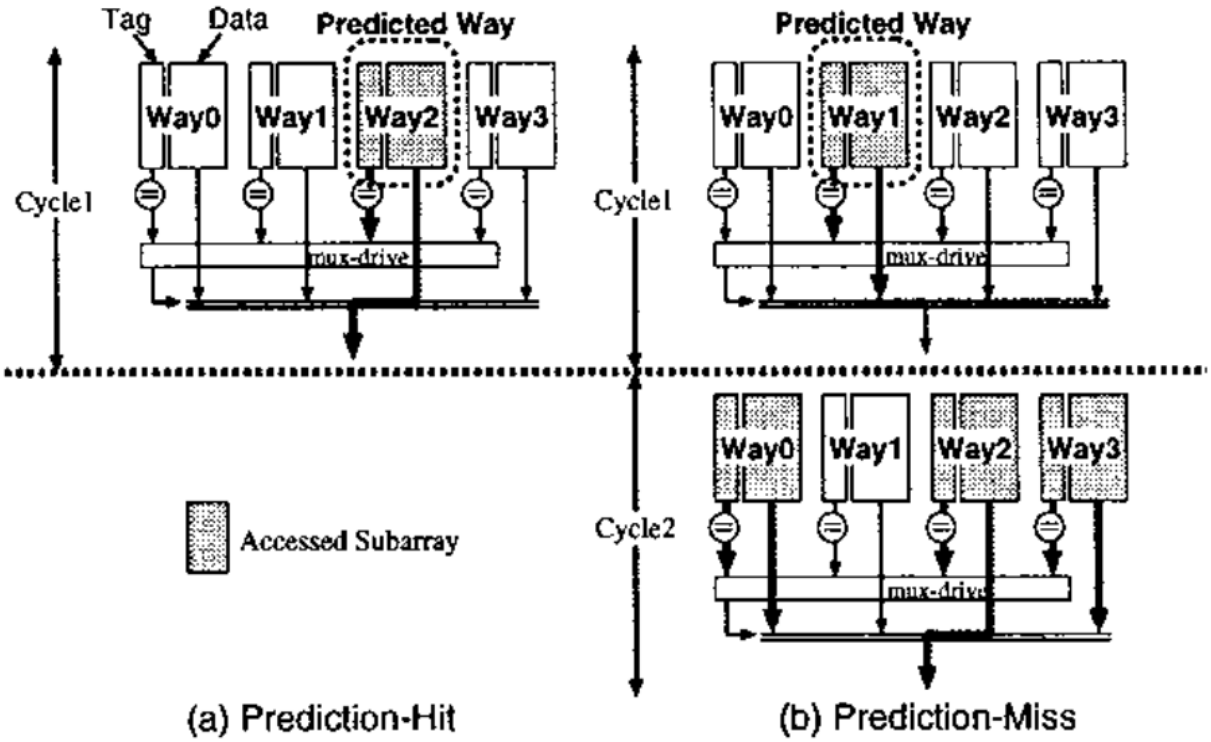


Figure 2: Way-Predicting Set-Associative Cache

> Ref: In this paper, we have employed a MRU (Most Recently Used) algorithm for the way prediction. In case of a 16 KB four-way set-associative cache, the MRU region is only 4 KB. The MRU information for each set, which is a two-bit flag, is used to speculatively choose one way from the corresponding set. These two-bit flags are stored in a table accessed by the set-index address. Reading the MKU information before starting the cache access might make cache access time longer. However, it can be hidden by calculating the set-index address at an earlier pipe-line stage [l]. In addition, way prediction helps reduce cache access-time due to eliminating of a delay for way selection.

Crtical Word First & Early Start

When a miss occurs in L1 cache,the pipeline will stall while we fetch the cache line from the next level of cache(in case is the L2). The efficiency of this process rely heavily on the width of the bus connecting L1 and L2 cache, as it directly affects the speed of data transfer.

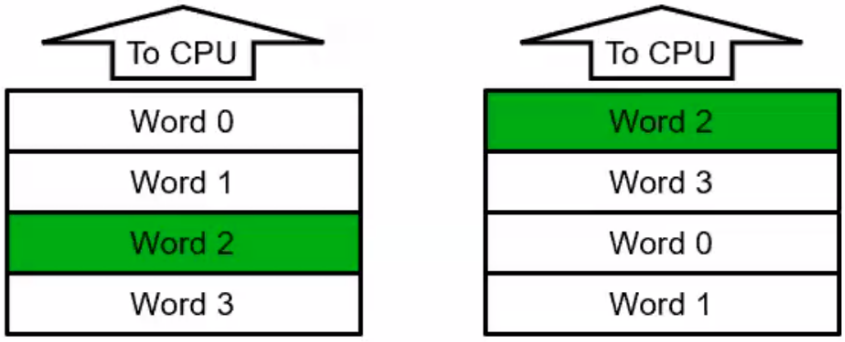
To determine its width, we essentially need to do some trade-off.

- **small width** -> more cycles required for data transfer.
- **large width** -> consume more space and energy.

Some architectures choose a 32-bit bus width for their L1 cache, with each cache line spanning 512 bits. As a result, filling the L1 cache may take as long as 16 cycles, posing a significant performance bottleneck. How do architects mitigate this challenge

Reduce Miss Penalty of Long Blocks: Early Restart and Critical Word First

- Don’t wait for full block before restarting CPU
- *Early restart*—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- *Critical Word First*—Request the missed word first from memor and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
  - Long blocks more popular today ⇒ Critical Word 1st Widely used



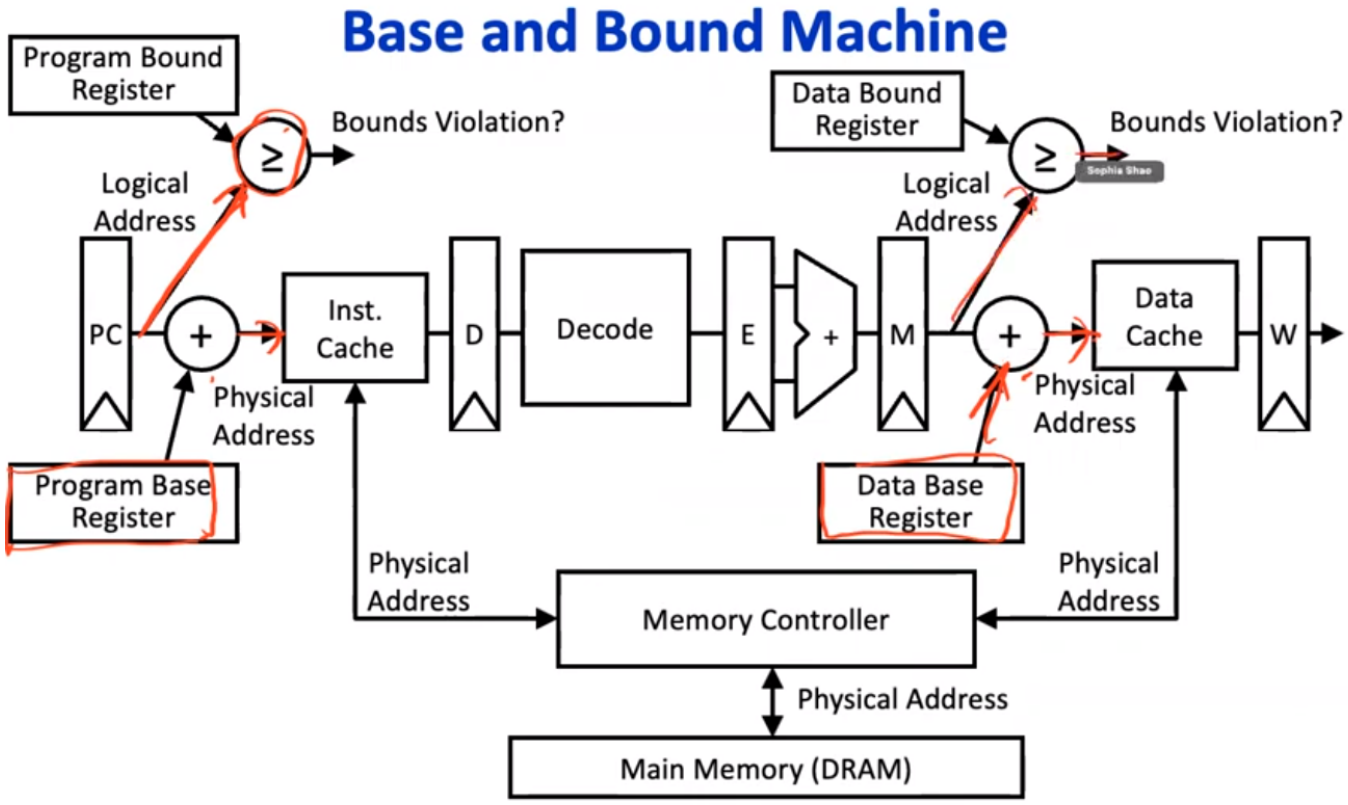
> This technique is based on the observation that the processor normally needs just one word of the block at a time. This strategy is impatience: don’t wait for the full.

Case Study: THE MIPS RIO000 SUPERSCALAR MICROPROCE

> https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=491460

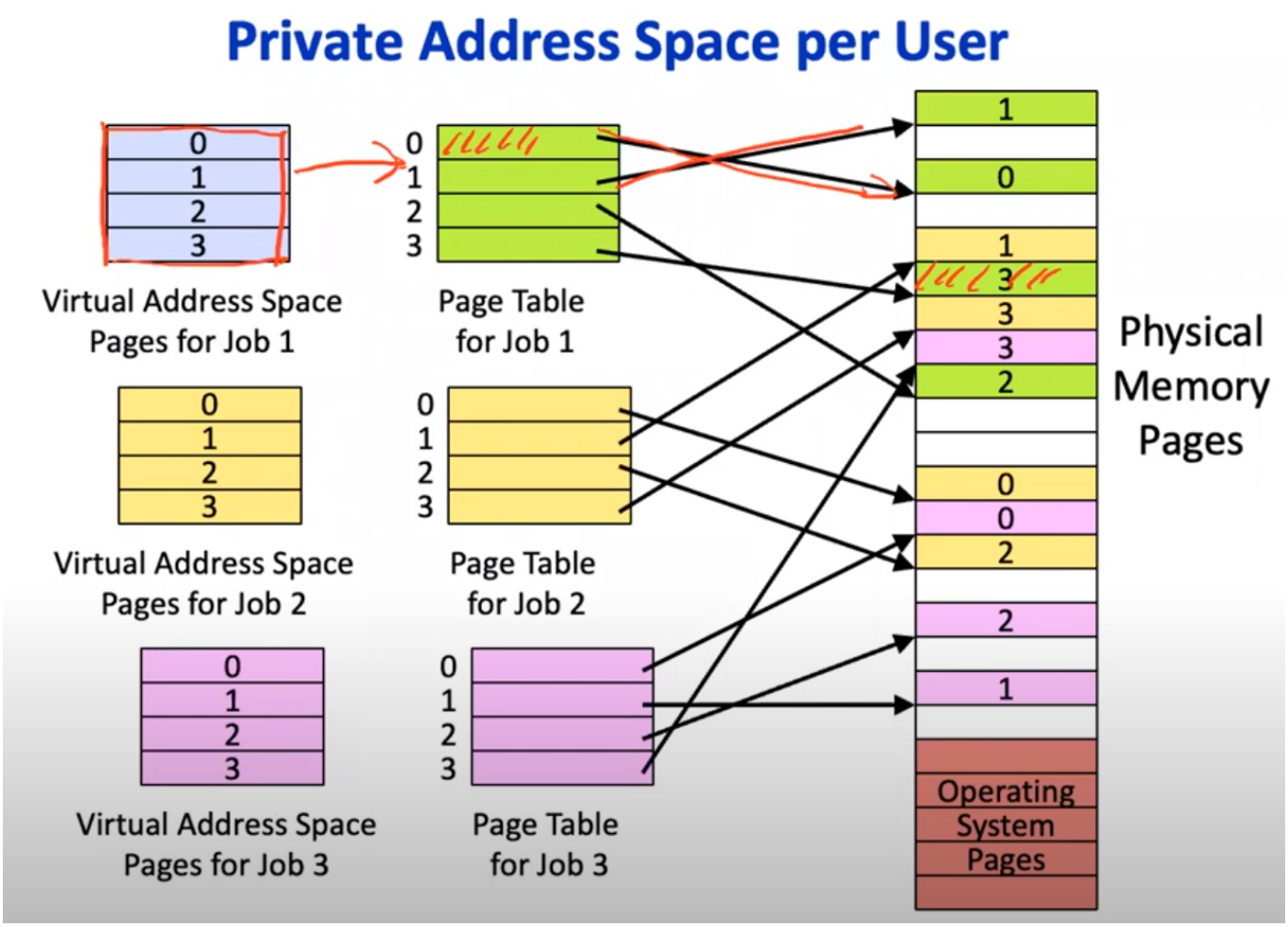
Virtual Memory - Manage Memory at Page Granularity

Base Design

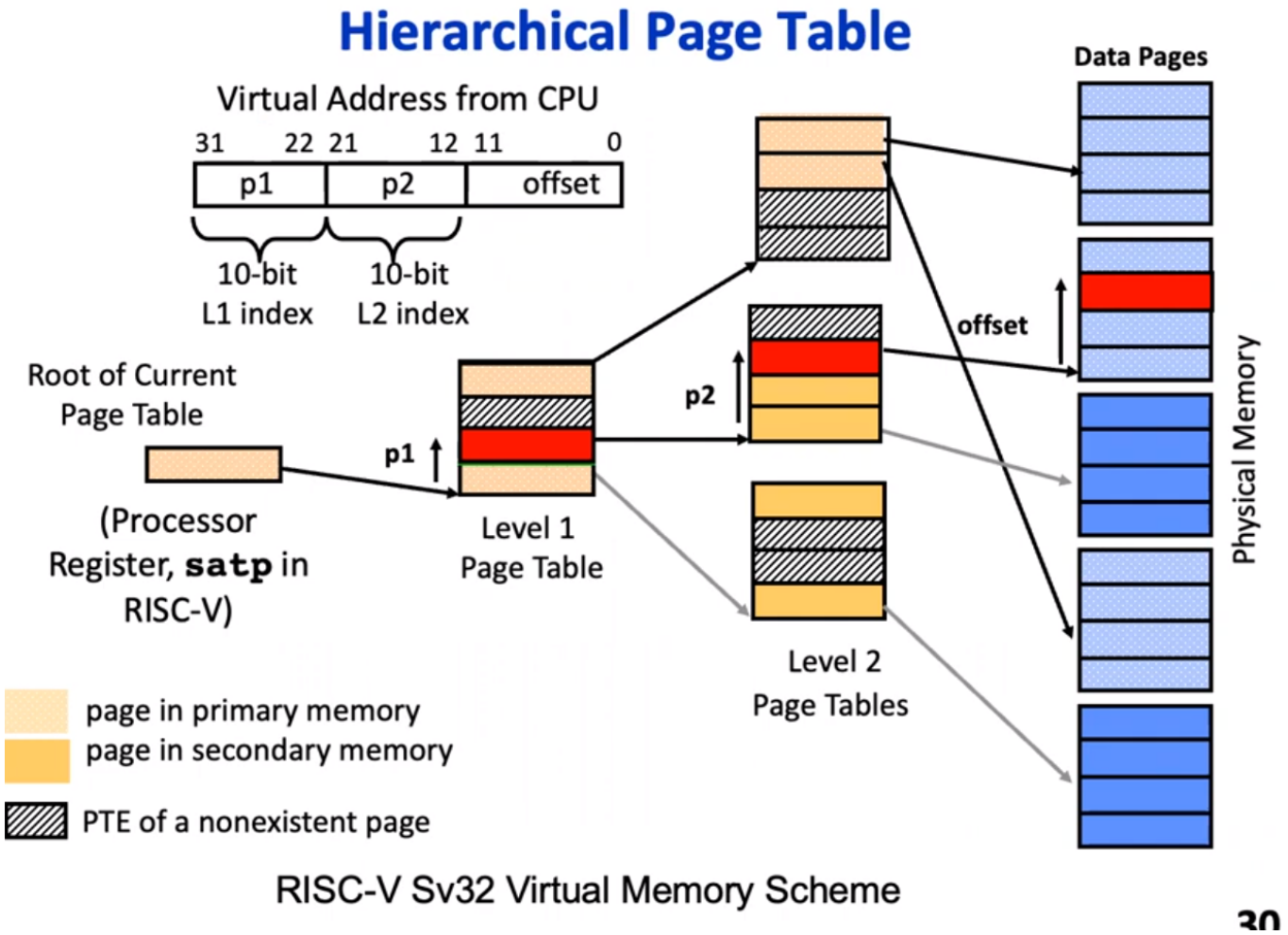


Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)

Page Translation Offers Protection



Hierarchical Page Table



Translation Lookaside Buffer

Translation-Lookaside Buffers (TLB)

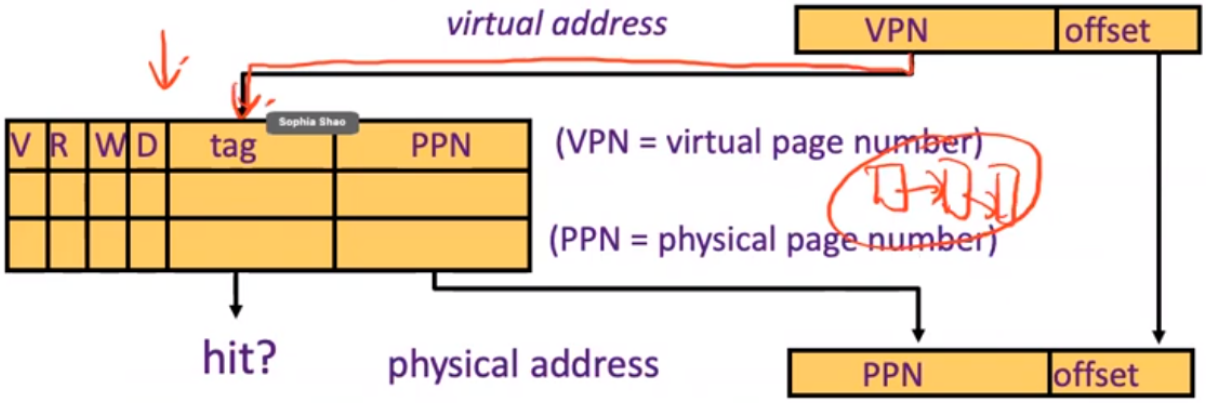
Address translation is very expensive!

In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

TLB hit ⇒ *Single-Cycle Translation*

TLB miss ⇒ *Page-Table Walk to refill*



Design Principle

TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages ➔ more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs
- Random or FIFO replacement policy
- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB
  - Example: 64 TLB entries, 4KB pages, one page per entry
  - TLB Reach = ?

TLB miss

Handling a TLB Miss

- Software (*MIPS, Alpha*)
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged "untranslated" addressing mode used for walk.
  - Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.
- Hardware (*SPARC v8, x86, PowerPC, RISC-V*)
  - A memory management unit (MMU) walks the page tables and reloads the TLB.
  - If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page Fault exception for the original instruction.
- NOTE: A given ISA can use either TLB miss strategy

36