Data Hazards.

- Execution time = # instructions · $\frac{\text{cycles}}{\text{instruction}}$ · $\frac{\text{seconds}}{\text{cycles}}$

Pipeline Hazards:

- Structural hazards; attempt to use the same resource by two different instructions at the same
- Data hazards; attempt to use data before its ready
  - An instruction's source operands are produced by prior instruction still in the pipeline
- Control hazards; attempt to make a decision about a program control flow before the condition has been evaluated and the new PC target address calculated.
  - Branch and jump instructions, exceptions
- Can always resolve hazards by waiting
  - Pipeline control must detect the hazard
  - And take action resolve hazards

Resolving control hazards! inserting NOPs or stalling the pipeline or forwarding with look-one
- move branch point
- Delay branching
- predict outcome

BTB = Branch target Buffer
- Cache of target addresses
- Indexed by PC when instruction is fetched
  - If hit and instruction is branch predicted taken, can fetch target immediately.
  - works for jumps (always predicted as taken)

Control Hazards;
- You can stall a pipeline by introducing flush when jumping but could impact performance. Early in the pipeline as possible

Branch-delay slot;
- Branch does not take effect until the 8ycle after its execution
- Compiler can fill one slot 50% of the time

Performance
- Instructions are stalled / caused 15%
  - $CPI = (1+3) \cdot .15 + 0.85 \cdot 1 = 1.45$
- Instruction is stalled
  - $CPI = (1+1) \cdot .15 + .85 = 1.15$

Virtual memory:
- Use main memory as "cache" for secondary (disk) storage
  - managed jointly by CPU hardware and the OS
- Programs share memory
  - Each gets a private virtual access space holding its code and data
  - Frequently used code and data pulled into main memory

Page fault penalty:
- On a page fault, the page must be fetched from disk
  - takes millions of clock cycles
  - handled by OS
- Minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

Page tables:
- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - page table register in CPU points to page table in physical memory.
- If page is present in memory
  - PTE stores the physical page number
  - + status bits
- If not present
  - PTE can refer to location in swap space on disk

Categorizing cache misses:
- Compulsory (initial misses when cache is empty)
- Capacity (cache is full, depends only on cache size)
- Conflicts (two addresses corresponding to the same block, depends on cache size and associativity)

Replacements and writes;
- To reduce page fault rate, prefer least recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - write through is impractical
  - use write-back
  - Dirty bit in PTE set when page is written

Page table size:
Calculate the size of a page table
- 32 bit virtual address
- 4KB pages, and page offset = 12 bits
- 4 byte page table entries (PTEs)

$\#PTEs = \frac{2^{\#\text{ bits for virtual address}}}{2^{\#\text{ bits page offset}}} = \frac{2^{32}}{2^{12}} = 2^{20}$

$PT\ size = \#PTEs \cdot PTE\ size = 2^{20}\ PTEs \cdot 2^2 \frac{\text{bytes}}{PTE} = 4MB$

Fast translation
- Address translation would appear to require extra memory references
  - one to access memory access

TLB misses:
- If page is in memory
  - Load the PTE from memory and retry
  - could be handled in hardware
  - or in software
    - special exception, write handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table

Set Associative Blocks:
- Block offset = memory address mod 2^n
- Block Address = memory address / 2^n
- Set index = Block Address mod 2^s

Tag	Index	
(m-s-n)	s	n ← Block offset

- Larger block sizes take advantage of spatial locality

Block Replacement:
- Any empty block in the correct set may be used for storing data.
- Replacement policy
  - Identify the block used the furthest in the future
  - Impossible to predict
  - Select other feasible approaches.
- No empty blocks → least recently used block.

Write Buffers:
- write caches → slow memory

Split instructions / data caches!
- pro: No structural hazards between IF & MEM stages
- A single ported unified cache stalls fetch during load or store
- con: static partitioning of cache between instruction and data
  - Bad if working sets dynamically unexpected
- Cache Hit
  - tradeoff between access time & hit rate
  - L1 cache can focus on fast access time (with okay hit rate)
  - L2 cache can focus on good hit rate (with okay access time)
- Identify all pipeline stages that really consume values → EX (both A and B ports) and MEM (only data port)
  - These stages are the destinations of a forwarded operand
  - Add multiplexer input for each pair of source/destination stages.

Memory stall cycles:
(memory accesses) (miss rate) (miss penalty)

CPU time = (CPU execution cycles + memory stall cycles) · cycle time

Average memory access time / AMAT
= Hit time + (miss rate · miss penalty)

15% overall (64 bits direct mapped vs 4 way?)
- CPI = 1
- 1 cycle hits
- 30% memory instruction
- 14 cycle miss penalty

$ET = I \times CPI \times CT$

$CPI = F_{nonmem} \cdot 1 + F_{mem} \cdot (L_{hit} + F_{miss} \cdot L_{miss})$

$CPI_{dm} = (1 - 0.3) + 0.3 \cdot (1 + 0.045 \cdot 14) = 1.1$[illegible]

$CPI_{4way} = (1 - 0.3) + 0.3 \cdot (1 + 0.03 \cdot 14) = 1.1$[illegible]

- $\frac{CPI_{dm}}{CPI_{4w}} = 1.06$ → 6% performance gain

Basic memory Design:
- 1 cycle = sends address to RAM
- 15-cycle = for each RAM access
- 1 cycle = return data from RAM
- If the cache has one-word blocks when filling a block would take 17 cycles
  - 1 + 15 + 1 = 17 cycles
- If it has 4-word blocks then it would take 4 · (1+15+1) = 68 clock cycles miss penalties.

Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better

	Cache 1	Cache 2
Block size	32-byte	64-byte
miss rate	5%	4%

8 byte gets the first memory access take 15 cycles, and the memory bus is 8-bytes wide
a 16-byte memory access takes 18 cycles: 1 + 15 + 2 (two 8 byte transfers)

AMAT overall

8 KB direct-mapped data cache with 64-byte blocks
- offset = $\log_2(64)$ = 6 bits (change base $\frac{\log(64)}{\log(2)}$)
- number of blocks = 8K / 64 = 128
- Index = 7 bits
- Tag = 32 - 13 = 19 bits

Data Forwarding: IF D EX M WB

Forwarding cost: Needs to add stuff to data path and stuff to control
Datapath:
- multiplexers at the input of functional units
- wires from forwarding sources to forwarding destinations
control:
- Additional control signals for multiplexers
Notes:
- Adding these wires increases the critical path of design
- Needs to be designed carefully.

Implementation:
- Identify all pipeline stages that produce new values → EX and stages
- All pipeline registers after the earliest producer can be the source of a forwarded operand → EX/MEM, MEM/WB

Definitions:

DRAM = Dynamic Access Memory
- high density, low power, cheap, but slow
- contents are lost when no power

SRAM = Static Random Access Memory
- Lower Density
  - cost more
- going to hold forever
  - only when power is on

Flash types:

- NOR flash: bit cell like a NOR Gate
  - Random read/write access
  - used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser (bits/area), but block-at-a-time access
  - cheaper per GB
- Flash bits wears out after 1000's of access
  - Not suitable for direct RAM or disk replacement

Caches:

- cache: a small amount of fast programmable memory
  - goes between processor and dynamic main memory
  - keeps a copy of the most recently used data
  - Memory access speed increases overall (because we've made the common case faster
    - Reads and writes to the most frequently used addresses will be serviced by the cache

Locality:

- temporal locality:
  - If program accesses one memory address, there is a good chance that it will access the same address again.
- spatial locality:
  - If a program accesses one memory accesses are memory address, there is a good chance that it will access other nearby addresses
- cache hit: occurs if the cache contains the data that we're looking for. Hits are good, because the cache can return the data much faster than main memory. / handled by cache
- cache miss: occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower memory.

blocks: caches are divided

- Direct-mapped: Cache is the simplest approach. Each main memory address maps exactly to one cache block
- tags: supply the rest of the address bits to let us distinguish between different memory locations that map to the same cache block.
- Valid bit: adding for each cache block
- meta-data: to help us find data within the cache and verify its validity
- cache controller: When the CPU tries to read from memory, the address will be sent here.
- Least recently used: If the cache fills up, older data is less likely to be requested than newer data
- block addresses: If the cache block size is 2^n bytes, we can conceptually split the main memory into 2^n-byte chunks
- block offset: The lowest n bits, that decides which of the 2^n bytes in the cache block will store the data

Cache Design:

- fully-associative cache: permits data to be stored in any cache block, instead of forcing each memory address into one particular block.
  - can be placed in any unused block of the cache
  - avoid a conflict with two or more addresses that map to a single cache block
  - least recently used
  - Allows any memory blocks to map to any block within the cache opposed to direct mapping
- Set associativity: each memory address maps to exactly one set in the cache, but data may be placed in any block within that set
- Sets: the cache is divided into groups of blocks
  - If each set has 2^x blocks, the cache is a 2^x-way associative cache
- set index: to select a set within the cache instead of an individual block
- Associative cache: assign each memory address to a particular set within the cache, but not to any specific block within that set
- write-through cache: solves the inconsistency problem by forcing all writes to update both the cache and the main memory
  - keeps the cache and memory consistent
- write-buffer: queues writes to main memory and permits the CPU to continue.
  - commonly used when two devices run at different speeds)
  - If a producer temporarily generates bursts of data too quickly for a consumer to handle, the extra data is stored in a buffer and the producer can continue on the other tasks, without waiting for the consumer
- write-back cache: the memory is not updated until the cache block needs to be replaced
- dirty bit: to indicate whether or not it must be saved to main memory before being replaced - otherwise perform unnecessary write backs
- write-miss: trying to write to an address that is not already contained in the cache
- write around policy: the write operation goes directly to main memory without affecting the cache.
- Allocate on write: load the newly written data into the cache.

Performance:

- interleave: the memory are split into banks that can be accessed individually

virtual memory:

- Translation Look-aside Buffer (TLB): fast cache of PTEs within the CPU

Quiz Recalling,

What is the purpose of TLB?

It is used to speed up the virtual to physical page translation. It holds the PTEs.

Quiz 7

Calculate the required size of a naive page table for 64-bit virtual address space that uses 4KB pages and has 16 byte page table entries (PTE's)?

$$\frac{2^{64}}{2^{12}} = 2^{52} \text{ entries}$$

$2^{52} \cdot 2^4 = 2^{56}$ bytes in total $= 64$ PB

What would the minimum page size needs to be for the following virtually indexed, physical-tagged cache not to have aliasing? Assume byte-addressable memory space.

Block size: 32 B
Total data size: 64 KB
Associativity: 8-way

5 bits for block offset

$$\# \text{ of sets} = \frac{2^{16}}{(2^3 \cdot 2^5)} = 256 \text{ sets} \rightarrow 8 \text{ bits}$$

page offset $= 5 + 8 = 13$ bits
$= 2^{13}$
$= 8$ KB minimum page size

Quiz 5

What is cache?

It's a memory type that is used to store most frequently used data and program memory blocks to speed up the program. Cache accesses are much faster than DRAM accesses.

Consider two cache structures, both initially empty...

Index	Valid	Tag	Data

Index	Valid bit	tag	Data

For the following memory accesses, what are the hit and miss results, for two different caches?

mem[2], mem[5], mem[6], mem[10], mem[5], mem[2], mem[1], mem[4], mem[5], mem[6]

Small cache → miss, miss, miss, hit, hit, miss, miss, miss, miss, miss
→ 1/10

Large cache → miss, miss, hits, miss, hit, miss, miss, miss, hit, hit

Quiz 6

For byte addressable machine with a 16-bit address space, how many bits of data does the cache hold? (Do not count meta-data bits)

Cache parameters: Direct mapped, block size is one byte, cache index is the four least significant bits of the address

16 blocks, 8 bits = 128 bits

For a byte addressable machine with 16 bit address space, how many bits of meta-data does the following cache hold?

Cache parameters: block size is one byte, Direct mapped, cache index is the four least significant bits of the address

16 blocks. (12 + 1 bits) = 208 bits

Tag = 12 bits. Valid bit = 1 bit

Note: Index bits are not stored in the cache

Fully associative vs. set-associative cache: For the following memory accesses, what are the hit/miss results for a 4 sets associative cache and a 2-way, 2 sets set associative cache?

Assumed the caches are initially empty, Least recently used policy and each cache block is one byte

	M[0]	M[1]	M[1?]	M[?]	M[4]	M[0]	M[0]	M[4]	M[?]	M[?]	M[?]	M[?]
fully	M	M	M	M	M	M	H	H	M	H	H	M
4-sets	M	M	M	M	M	M	H	M	M	M	M	H
2-sets	M	M	M	M	M	M	M	M	M	M	H	H