- Jump addresses should be forwarded to earlier stage.

- Branch prediction is a complicated art.

- Muxes can be added to each intermediate stage

- Muxes can be added between each stage, and the register file to ALU path.

- Some instructions cannot be resolved with just bypassing. Ex. load instructions: always have to wait for writeback phase. This is when stalling is necessary. Sometimes compilers will try to prevent this, but cannot depend on them too much.

Control Hazards

- Two ways of mitigation:
  - Stalling:
    - Just stall until branch result is known
  - Prediction:
    - Follow one of the branches,
      - If right, continue as is.
      - If missed, flush pipeline and restore corrupted instructions.

ON Reference to Mem[X] : look for Tag(x)

Hit Tag(x) == Tag[i]
READ : return Data[i]
Write : change Data[i] dirty[i] = 1

MISS: Tag(x)
- replace selection
  - select block h to hold Mem[X]
  - If dirty[h] = 1 write to Mem[h], dirty[h]=0
  - Read Mem[x]
  - Write: start Write to Mem[X], dirty = 1

CPU Pipelining

- Just simply pipline 5 stages, DUH!
  - Not so simple.
  - Data Hazards

- How to fix data hazards:
  - Stall, Bypass, or Speculate.
- Stalling make execution kinda slow :(Might diminish pipelining benefits.
- Bypassing uses extra muxes and logic to detect hazards that gives computed values to previous stages when necessary.

Write Policies

Write Through: CPU writes are cached, and then written to memory immediately (CPU stalls)
- Simple, slow AF

Write-behind: CPU writes cached; writes are buffered and executed in the background. This is faster but actually uses a lot of bandwidth and hardware.

Write-back: CPU writes cached, but not written to main memory until block replacements
- Fastest, but leads to stale data in main; a problem for multicore systems. Often can add a dirty bit to each cache line, enabling writes back to main only when bit is set, which only happens after cache line is written to.

N-Way Set Associative Cache

- A middle ground between direct mapping and FA. Can be seen as a set of smaller, parallel DM caches within one. The most significant bits are associative. The least or direct within each array.
  - More ways reduces conflict but increases hit time, also adds more hardware costs.

Cache Replacement Policies: How To Choose a line after Cache Miss

- Locality suggests the LRU strategy:
  - Replaces least recently used:
    - Works in theory but requires $\log_2(N!)$ bits to encode order and a massive amount of hardware to find the largest number, too expensive.
- LRU is still reasonable sometimes.

- Miss penalty increases with block size, miss ratio is optimized between at a certain block size, and so is AMAT (around 64 bytes)

DM Cache Problems

- Conflict misses - when two data addresses that map to same cache line are both needed consistently for programs, making them very likely to cause misses due to alternating presence in the cache.

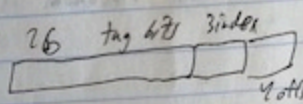
Fully Associative Cache

- No indexes, only pure 32-bit tags, with some large amounts of data per blocks. No set conf
- They have the lower miss rates, but have high hardware cost due to comparing 32 bits at once for each line.

- There are as many index bits as $\log_2$ of available cache lines. ~~offset bits 32 address that provide number.~~
- Index bits are not necessarily the very least LSBs, but rather the next level up from the 2 00, or offset bits.

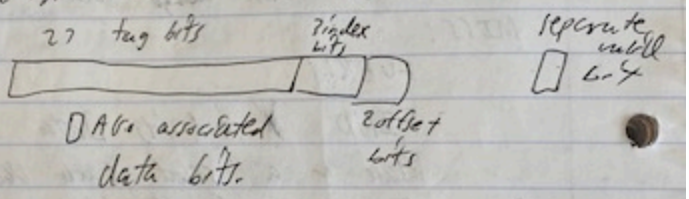
Blocking and Data Size

- Another idea for 32-bit processors: 128 data bits per line. Each line represents 4 words instead of one. Therefore not as many address bits are needed (least significants can be offset for a mux to select specific word instead), and more data stored means a much lower miss rate!
  - Trade off - worse cost for misses, hits do get slightly slower, much more data retrieved.



↑ acts as mux for each word stored within block. More indexes would be used in larger caches.

- Direct Mapping caches also use a valid bit. If the line's memory is just garbage because the cache was just flushed. If the line was not actually retrieved from memory, it is not valid.



- Because of locality principle, a cache is assumed to store memory in the same region, therefore making the addresses only differ in the least significant bits.
- When looking for a particular address, the processor looks in parallel for matching index (least significant) bits. If it finds a match, it compares with rest of tag and valid bit. If correct, cache hit, otherwise nah.

A cache holds many blocks of data with a tag (address), and its data.

Look for Mem[X]

HIT: X = TAG
Read data
Write and change data

MISS:
FUCK!!
FIND X and it's Data
Place in a select cache line
Read or write to it.

How to quickly Read Through Caches

Direct Mapping
- Each memory address in CPU address space maps to a specific cache address.
- This is very simple.
- Because there is a much larger space than cache memory slots, the lines of cache have many addresses linked to each one.

- Handshaking protocols involving valid/ready signals can help prevent deadlocks or loops if conditionals are used correctly. This is much less stringent than strictly-timed pipelines. Making them asynchronous or even purely combinational is another hot topic that could speed things up.

Memory Technologies and Control

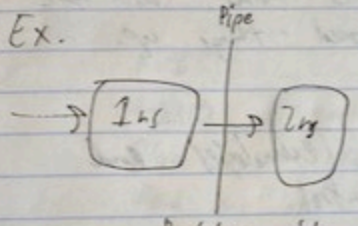
- Registers are fast, but not very spacious, so we need more distant memory to work with like SRAM and DRAM.

- Caching:
  - If processor needs data:
    - In cache:
      - Cache hit, yay!
    - Otherwise, cache miss
      - Minimize these please, now we have to access main memory and find the address and then store on cache.

Goal, achieve max throughput with fewest registers.

What if one stage cannot be broken up and I still want to speed up pipeline?

Ex.

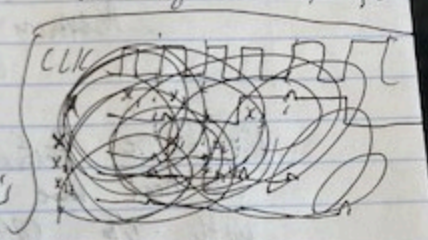


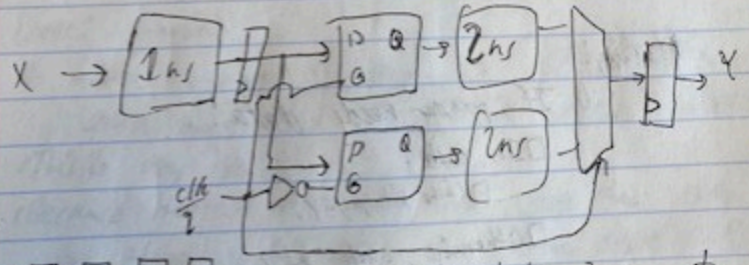
latency: 4ns
throughput: $\frac{1}{2}$/ns

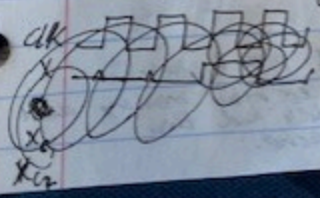
not pipelined
latency: 3ns
throughput: $\frac{1}{3}$ ns

Better idea

Now try this



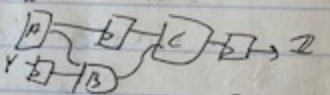




still takes 3ns to get original signal, but having parallel duplicates doubles the speed of that stage. ~~throughput~~ throughput now 1/ns.

Even though option 1 as higher throughput, it takes one input 120 ns to complete vs option 2's 60 ns. This demonstrates the tradeoff.

- Balancing a pipeline could mitigate this, as a pipeline is only as good as it's longest delay, but no matter what, latency is always lower due to FF delays that would otherwise not be present.

- Successive Inputs should NEVER get mixed. Ex. 

Pipelining Method

- Draw line that crosses every output in the circuit. Make sure every connection crosses each line in the same direction
- Place registers around the longest combinational components.

MIT OCW Computational structures

7, 14, 15

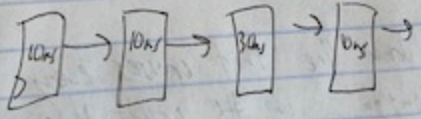
Chapter 7 : Pipelined Performance

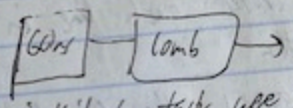
- Latency is a measure of time to complete a single task.
- Throughput is a measure of the number of tasks that can be completed at once.

Pipeline stages

- More pipeline stages increases throughput, but decreases latency, not ideal for smaller loads, or real time loads.

Ex.

a.) 

b.) 

If individual tasks were broken into smaller ones to be solved in sequence (pipelining), example a we would have a throughput of 1 output per 30ns. Example b would not be high