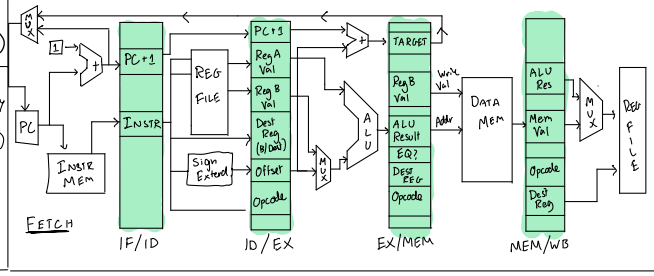
Iron Law:

Exec Time = (# instr)(CPI)(Clock Period)

Pipelining:

Increases throughput without needing to reduce latency for each instruction! (short cycle, small CPI)

- Fetch: Retrieve instruction from memory (++PC)
- Decode: Break instruction into component parts and retrieve register values
- Execute: Do ALU stuff for ADD/NOR/LW/SW/BEQ rA+rB / rA + offset / PC + offset + 1
- Memory: Write to data memory for load/store instr.
- Writeback: Write back to registers



Internal Forwarding:

- Assume register file can return updated value when read from/written to in same cycle

E.g.: ADD on line 1 to R3
ADD on line 4 using R3 is NOT a problem b/c Line 1 is in Decode while Line 4 is in Writeback

ONLY FOR REG FILE

LC2K ISA: 32 bit

0. ADD RA RB R-Dest — R-Dest = RA + RB
1. NOR RA RB R-Dest — R-Dest = !(RA | RB)
2. LW RA RB #offset — RB = RA + offset
3. SW RA RB #offset — [RA + offset] = RB
4. BEQ RA RB # offset — Branch to PC + offset + 1 iff RA == RB (else ++PC)
5. JALR RA RB — 1. RB = PC + 1 2. PC = RA } If A==B, then ++PC.
6. HALT
7. NOOP

- 24-22: OPCODE
- 21-19: REG A
- 18-16: REG B
- 2-0: REG DEST
- 15-0: offset field
- ALL UNUSED BITS: 0 • 0 → 31: Least → Most Sig

Data Hazards:

- Read-after-write dependency: The destination register for add/nor/lw is read from in add/nor/lw/sw/beq
  - register reads happen in Stage 2
  - register writes happen in Stage 5
- At most, next 2 instructions after add/nor/lw can trigger a data hazard in 5 stage-pipeline
- Recall: DestReg is destination for add/nor, RegB is destination for LW } IMPORTANT
- Approaches:
  1. Avoid: Pad instructions with noops (≤ 2) Done by user in code
  2. Detect + Stall: Pad instructions w/ noops, but do it in the hardware. Stall IF, ID stages until front instr. hits WB. Increases CPI - 1-3 cycles for add/nor/lw
  3. Detect + Forward: Once value is computed, forward to decode stage as needed.
     - For ADD/NOR, computed in stage 3, so can be internally-fwded (no stall needed) to 2
     - For LW, computed in stage 4 (MEM), so can be fwded only if instr is 2-before. Still need stall for 1-after instruction.

Cache Vocab:

- Things in cache line: → tag → LRU-tracking → Valid bit → Dirty Bit
- Write-Through: Memory is written to whenever some data is stored.
- Write-Back: Memory is written to only on eviction. This requires a dirty bit.
- Allocate-on-write: If an address used in a store is not in the cache, bring data into cache. → no allocate-on-write would write directly to mem
- Fully-Associative: Addresses can go anywhere in the cache. Same as n-way set associative cache with n cache lines (1 set).
- Direct Mapped: Every address maps to exactly one cache line. Same as 1-way set associative cache (n sets w/ n cache lines).
- Set-Associative: Every address maps to a set of cache lines. n-way set associative → n cache lines in each set.
  → balances latency to get data w/ hit rate

TAG	SET IDX	BLOCK OFFSET
leftover	log(# sets)	log(# addy / bblock)

- Block Size: Amount of data in one line of cache (per tag). Increasing block size reduces overhead b/c more data is stored per line and tag. Also takes advantage of SPATIAL LOCALITY
  → # blocks (# lines) = cache size / block size

- # of ways: def doesn't have to be power of 2
- block size, # sets: would be a pain if not powers of 2, so no

Byte-Addressable vs. Word-Addressable:

64-bit ISA → 64-bit word
32-bit ISA → 32-bit word

Changes block offset size calculation; if block is 256 = 2^8 bytes in byte-addressable ISA, then 80 = 8 bits. If it were a 64-bit word addressable ISA, then 80 = 5 bits b/c 32 words in 256 bytes.

LC2K is 16-bit address, word (4 byte) addressable.

CPI = Base + Control Hazard Stalls + Data Hazard Stalls + I-Cache Stalls + D-Cache Stalls

LRU Bits = log(# cache lines)

Often have 3 levels of caches — L1, L2, L3

Control Hazards

- Triggered by BEQ instructions: don't know what to execute next until BEQ executed (S3)
- Approaches:
  1. Avoid: This is dumb and impractical
  2. Detect + Stall: Keep instructions in IF stage until branch is predicted
     - need instr in IF stage
     - get ALU result in EX stage, but no forwarding
     - so, stall for ≤ 3 instructions (IF-MEM align)
  3. Speculate + Squash: Assume taken/not-taken
     - if correct, no change needed!
     - if wrong, "squash" (replace w/ noops) all instr that are in prev pipeline stages, and load in correct PC
       → works b/c no perm changes till WB/MEM
       → send noops to decode/ex/mem (IF/ID, ID/EX, EX/MEM registers)
     - 3 stalls if wrong, 0 if correct

Cache Design Considerations:

- Instruction Cache (IC) vs. Data Cache (DC)
  → can be separate or same
  → Pro: can optimize cache design for the diff use cases; e.g., IC wants higher block size because instructions have more spatial locality
  → Con: can't dynamically tradeoff between data space and instruction space in software; may be worse on some programs.
- Fully-associative caches can be slow b/c parallel tag searches are expensive.

3C Problems

3 types of cache misses exist:

- Compulsory: First reference to an address. These can't be completely avoided, but reduced by increasing block size (spatial locality).
- Capacity: Would've had a hit w/ a large enough cache, reduce by increasing cache size.
- Conflict: Would've had a hit w/ a fully-associative cache; reduce by increasing associativity (larger set size).

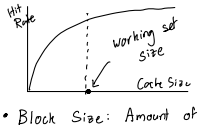
To classify misses, simulate 3 caches:

1. Fully-associative cache of infinite size, but same block size as original.
2. Fully-associative cache w/ all other parameters same as original.
3. Actual intended cache

Miss in ①,②,③ → Compulsory
Miss in ②,③ → Capacity
Miss in ③ → Conflict

Cache Parameters vs. Hit Rate

- Cache Size: Total data capacity. Bigger can take advantage of temporal locality better, but may increase cache lookup latency, degrade critical path



"Working Set": Whole set of data executing references within a time interval

- Block Size: Amount of data associated with a single tag. If too small, don't take advantage of spatial locality and lead to compulsory misses. If too big, transfer a lot of useless data — this both increases latency and kicks out useful data.

- Associativity: How many blocks map to the same set? Larger associativity ⇒ lower miss rate, but diminishing returns. Smaller associativity is lower cost, faster hit time

Branch Prediction:

- Can we improve branch prediction at fetch stage?
  → branch direction (taken/not-taken)
  → branch target address
- Static Prediction: At compile time
  → based on analysis of program
  → common: BTFN (backwards taken, forwards not)
    → Loops are back-branches
    → others are forward
  → Overall: 60-70% of branches are taken
- Dynamic: @ run-time
  - local predictor: for that specific line
  - global predictor: for ALL branches
  - 1-bit counter: predict last time
  - 2-bit: SNT ⟷ WNT ⟷ WT ⟷ ST (NT / T)

Virtual Memory

- Cannot directly use virtual addresses (used by ISA) to get data; 2^{64} bytes of memory is tough, and need to separate processes. Need to translate virtual addresses to physical addresses
- Use disk as "extra" space if main memory exhausted (swap portition).
- Divide memory into pages; size of virtual page is size of physical page. Map pages.
- Page Tables: One per process, maintained by OS
  - In main mem, address already known
  - Indexed by VPN; store PPN, valid bit
    → NOT A TAG; space exists for all VPNs
  - Page fault: Page is on disk or invalid
    → Disk is evicted, invalid is unassigned
  - Page Table Entry contents:
    → PPN, Valid bit, Main Mem or Disk, Access Permissions, Dirty Page, LRU data
    → Sometimes LRU expensive, so OS uses heuristics like "accessed bit", cleared occasionally

Hierarchical Page Tables

- All entries in single-level page table are allocated when process starts; this is very large and problematic — so how do we allocate on demand?
- Have first-level page table point to second-level page table, which is created if needed — idea of a deque
  → generalize to n-level page table
  → more levels mean less memory but more lookup time

FL PN	SL PN	...	Page Offset
VPN			log(PT Size)

Address Translation

- Going thru N-level page table needs N memory accesses, which is very slow
- Solution: Translation Look-aside Buffer (TLB)
  → VPN to PPN, 16-512 entries
  → optimized cache for page lookups: low block size, only caches actual phys addys
- Virtually-addressed cache vs. Physically-addressed cache
  - Virtual: Faster access (no TLB / Page Table lookup in the middle)
  - can skip address translation
  - context-switching (changing processes) is expensive w/ VA cache as cache needs to be cleared/reloaded
  - Typically: L1 is VA; L2, L3 are PA

Pipelining Performance:

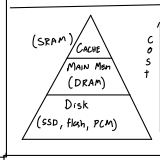
cycles = startup (=4) + # instr + data stalls + control stalls

CPI = 1 + (extra data stalls) + (extra control stalls)
= 1 + P(LW) P(hazard) 1 + P(BEQ) P(Wrong) (3) [detect + fwd / spec + squash]

Know how to do this for deeper pipelines as well:

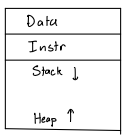
data stalls:
  # add/nor stalls: [EX Finish] - [Reg Read] - 1 (max)
  # LW stalls: [MEM finish] - [Reg Read] - 1 (max) } FOR DETECT + FWD also # of instr after you need to worry abt

control stalls: [Branch Resolve] - 1

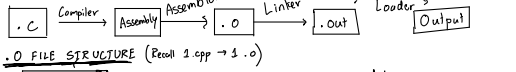


Cache: store data most likely to be used. Reduce Average Memory Access Time (AMAT).
Key-Value structure — search w/ a tag
TEMPORAL LOCALITY: Things recently accessed are likely accessed again → informs LRU policy
AMAT Problems: Check if cache access time is part of the mem latency!

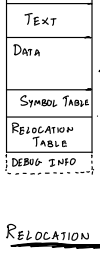
MISC First-Half Content



- CISC: Complex Instruction Set
- RISC: Reduced Instruction Set
- Combinational Logic: No memory (Moore)
- Sequential Logic: Have memory (Mealy)



.o File Structure (Recall 1.cpp → 1.o)



- Header: sizes of other parts ONLY
- Text: Machine code READ-ONLY SEGMENT
- Symbol Table:
  - Map symbols to addresses
  - strings → values (values can be addresses or consts.)
  - lists unresolved labels
  - function calls + globals
  - statics also here for consistency
- Relocation Table:
  - Identifies instructions (by address) that need to have addresses updated

ADDRESS (mult. 4)	INSTR	SYMBOL
0	LDUR	#D

  - All instr. that load/store stuff in data segment (static, global, extern)
  - All things that reference variables/funcs that are undefined in symbol table
  - Refs to locally defined functions DO NOT go here (in curr. .cpp)
  - Types: STUR, LDUR, BL
- Symbol Table: DOES have local functions, but not local vars

- Where data goes:
  → Local variables: stack
  → Arguments: 4 in regs rest on stack
  → Dynamic Memory: heap
  → Globals: Data
  → static vars: Data
  → const. strings: Data
- Data segment DOES NOT CONTAIN Stack / Heap
- #defines do not go anywhere
- In symbol table, anything external is "U"-type (unknown)
- Initializations/declarations are handled @ compile-time and don't need to be in relocation table

- K: 2^{10} / 10^3
- M: 2^{20} / 10^6
- G: 2^{30} / 10^9
- R-type instr: ADD/NOR
- LC2K is not big or little endian, b/c only one thing loaded @ a time
- Recall: FL Page Table is NOT A CACHE — size is determined by amount of of virtual addresses