Pre-MT ch1

C / Memory

*: dereferencer
&: referencer.

- ptr = malloc(size); Allocates uninitialized block, size size @ pointer ptr in memory
- ptr = calloc(num_items, size); allocates num_items blocks of zeroed memory of size size @ pointer ptr.
- free(ptr); free memory allocated at ptr. do this if you'll never use that mem. again.
- realloc(ptr, new_size); Either
  - Expands or contracts memory allocated by ptr to new_size
  - Allocates a new block, copies the memory it can from old block, frees old block.

Alignment

- char: 1 byte, no alignment rules
- short: 2 bytes, aligned by half word
- int: 4 bytes, aligned by word
  void*

Stack vs. Heap Space

- stack space: memory allocated on the call stack
  - most primitive data types
- heap space: memory allocated during execution
  - allocated memory
- static storage: global vars, string literals
- code storage: bytecode that comprises the code being run.

Struct:
- multiple blocks of data in one structure.
- mem is bunched adjacently & spaced according to alignment rules.

Unions:
- multiple data types stored in one block of memory.
- allocated to be the largest data type
- set block to 0 = set everything to 0.

CALL: **Compiler** translates C into a lower-level asm lang. like RISC-V

Assembler translates from Assembly into machine code.
- ↳ reads & uses directives, which give directions to the assembler for how to handle mem, etc.
- passes over twice for PC-relative cases.
- ↳ object file format:
  - header - size & pos of other pieces
  - text segment - code
  - data segment - binary rep of static data in source file

Linker resolves all assembled symbols, possibly w/ libraries, etc.
and, references to static data.
- ↳ stores this info in a relocation table & symbol table.
- ↳ slowest part of CALL

Loader
Running the executable, putting data into memory.
Is the OS

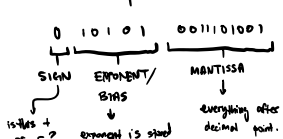
Number Rep:

- **Unsigned**
  - representable range: $[0, 2^n - 1]$
  - just ignores sign, best for nonnegative-only metrics like length.
  - if we wanted signs...
- **Signed Magnitude**
  - Representable Range: $[-2^{n-1}+1, 2^{n-1}-1]$
  - one bit determining sign, rest computing into as normal
- **bias**
  - Representable Range: $[B, 2^n - 1 + B]$, usually $B > 0$.
  - allows us to represent neg. #'s as nonneg.
  - would need to perform bias externally and # would work like unsigned.
- **2's complement**
  - Representable Range: $[-2^{n-1}, 2^{n-1} - 1]$
  - then, to repr. a negative number, flip all bits in original number & add 1 to the result.

Control Logic

- PCSel: If ∃ branch or jump, PC = label, not PC+4.
- ImmSel: diff. instruction types have diff. ways of making immediates
  - ↳ immsel determines which to use.
- RegWEn: do we write to register? No for store & branch
- BSel, ASel: picking what to add to ALU. A is/w PC & rs1, while B btwn. imm & rs2
- ALUSel: which ALU op
- MemRw: write to memory?
- WBSel: What to write to dest. reg? Memory, ALU, or PC+4?

- **Floating Point**
  - four components



Denorm?
- Change behavior of 0b00...0 s.t. range extends to exactly 0.
- ↳ instead of this being $1 \cdot 2^{BIAS}$, make it $0 \cdot 2^{BIAS+1}$ when exponent = 0
- ↳ decreases precision but at least we can represent #'s < 1.

∞
If exponents are all 1's, then our value is ∞ only if mantissa = 0. Otherwise, is NaN.

MTTF / MTTR.

mean time to failure. higher = better
mean time to repair. lower the better.

- ↳ MTBF = MTTF + MTTR, Availability = MTTF / MTBF.
- ↳ if we have smth running for X days out of Y, avail. = X/Y. Use this to find MTTF / MTTR. If we have one, can find the other.

- RAID
  - RAID 0: just put all memory inside one
    - not dependable
  - RAID 1: copy disk n times
    - restricts amt. of storage we can have
  - RAID 4: store info across disks w/ parity bit.
    - can run ecc, and more mem. efficient.
  - RAID 5: same as 4, but stratified.

- ECC
  - → parity = even-odd of sum of the bits found via XOR.
  - → Hamming Distance = # bits different b/w two strings.
  - → Hamming ECC: Use binary value to determine corrupted bit.
    - ↳ Check if parity bits need to be 1 or 0 after adding infill bits & encoding.
    - ↳ if p_i's have errors, find bit overlap of errors & flip the bit.

Pre-Midterm (Asm)

Syntax:

- upper immediate is anything $\geq 2^{12} = 1000$
  - lui is basically adding values above 12th bit to an index.
  - auipc adds upper immediate of an offset to the PC.
- jalr stores pc+4 in a register. commonly use "ra" to find where we jump back
- calling conventions!!
  - values of a, t registers do not carry over.
  - values of s registers & ra must be stored on Stack, done w/ sp.

Caches

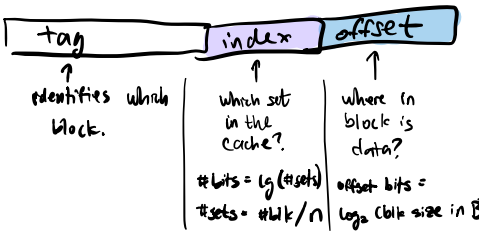
- intermediate btwn. energy cost & time for registers & DMEM.

lg (mem size) = addr. bit width
= |T| + |I| + |O|.

Cache size = (blk size)(# blk)
= $N \cdot 2^{|O|} + |I|$.

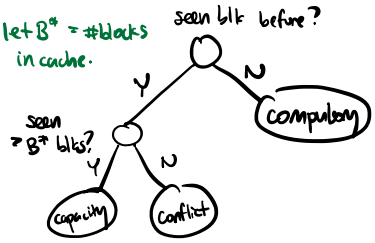
- n-way set associativity: we can put n addresses in a certain index.
  - fully associative: there is no index, we just put addresses in the cache.
  - more associativity: ↓ hit speed, ↑ hit time, ↑ hit rate.
    - however, cache misses are much more costly than cache hits, so a respectable hit rate is crucial.
  - direct-mapped: one block per index.

TIO:



- Cache misses.
  1) compulsory miss – have not yet fetched block.
  2) capacity miss – req'd data evicted because no space left.
  3) conflict miss – need to update cache to fit the tag.

let B* = #blocks in cache.

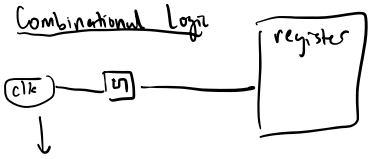


- eviction policy.
  - FIFO: first block added is evicted
  - LRU: least recently used block is evicted
    - b/c of locality, this is better.
- AMAT = avg. memory access time.
  (hit time) + (miss rate) · (miss penalty)
- multi-level caches.
  - ↳ usually increasing in associativity.

  L1 cache → L2 cache → etc.

  AMAT = L1 HT + L1MR · (L2 HT + L2MR(L3 ···))

Combinational Logic



clock repeatedly turns on & off. this is the clock cycle.

Registers only update once per clock cycle, when clk turns on!. however, you must wait until after clk-to-q time to see updates.

What are limits to clock cycle?

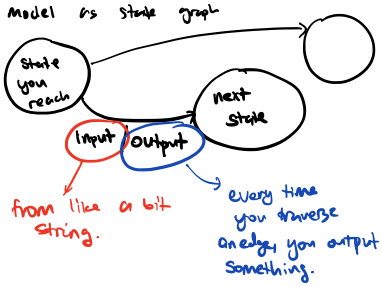
- each component in the circuit has a propagation time.
- setup time: amount of time before a clock interval that input must be stable/constant. } is a constant
- hold time: amt. of time after a clock interval that input must be stable. } depends on components.
- $t_{clk\text{-}to\text{-}q} + t_{shortest\text{-}comb\text{-}path} \geq t_{hold}$
- $t_{clk\text{-}to\text{-}q} + t_{longest\text{-}comb\text{-}path} + t_{setup} \leq t_{clock\text{-}period}$.

SHORTEST comb. path is determined by finding least time btwn. two timed components. (NOT necessarily regs.!)

LONGEST comb. path is determined by finding the longest time btwn. two timed components

Finite States

- detect a bit sequence or smth
- model as state graph



Post - MT:

Pipeline; (data path in ref sheet)

- latency: delay to process operation
- throughput: # of operations done in one op.
- pipelining increases latency, but also throughput.

There are 5 stages:

- IF → instruction fetch. what's the instruction
- ID → instruction decode - find the meaning of instruction
- EX → execute - do stuff w/ branch comp. ALU, etc.
- MEM → memory - put info into memory
- WB → write back - what to reinsert into the reg?

Must put registers btwn. stages

Hazards:

- structural hazard
  - hardware doesn't support running these insts. @ same time,
  - solutions are either stalling or + HW, and cur. regfile implementation seems to work.
- data hazard
  - instructions have data dependency,
  - need to wait for memory to behave as intended before using that block.

1) register access — Same reg. written & read in 1 cycle. Solution: write, then read
2) ALU result — inst. depends on WB's regfile write from prev. inst. Solution: forwarding: grab info from pipeline stage, requiring added hardware.
3) load data hazard — needs stall. need to access memory only changed during end of clock cycle, @ beginning of this cycle.
   ↳ need load delay, perf. can be salvaged by code scheduling.

- control Hazard
  - how to update the pipeline after branch is taken?
  ↳ must kill all processes and start @ label if branch taken.
  Sol. to speedup is to take a branch predictor.
  Loss of 0 cc if correct, 3cc if wrong.

Virtual Memory

Purpose: illusion of isolated processes, important for security.

↳ Memory is stored at fixed-size pages.
↳ can spoof addr. of physical memory w/ virtual memory.
↳ can have arbitrary amounts of physical & virtual memory.
  ↳ each index can be represented as [P/V]PN + offset.
↳ memory manager translates info.

TLB: translation lookaside buffer.
↳ cache for VPN-PPN mappings.

Parallelism. Can use the fact that CPU has multiple cores to do multiple things @ once

SIMD: does vector addition, in order to save time for many additions. should have a speed increase proportional to vector size.
from Intel AVX library.

} data-level parallelism

Amdahl's Law: speedup bound by ~~not~~ parallelizable.

$$\frac{1}{(1-F) + \frac{F}{S}}$$, where F is % code sped up. S = speed factor.

OpenMP:

- split a looped procedure into a series of threads.
- #pragma omp parallel
  ↳ runs in threads in following block
- #pragma omp for
  ↳ splits for loop automatically
- #pragma critical
  ↳ prevents data races.

Multiple threads receiving + modifying same data @ unsynch'd times

Whenever there's C code:
for example, x = x+1
this is a load, alu, and store. All 3 actions can happen at different times, with any # of others in between.

- # pragma omp reduction (+:sum) → compute the aggregate of some value in parallel

Multiprocessing

- run multiple processes = multiple independent instances.
- workers & manager processes.

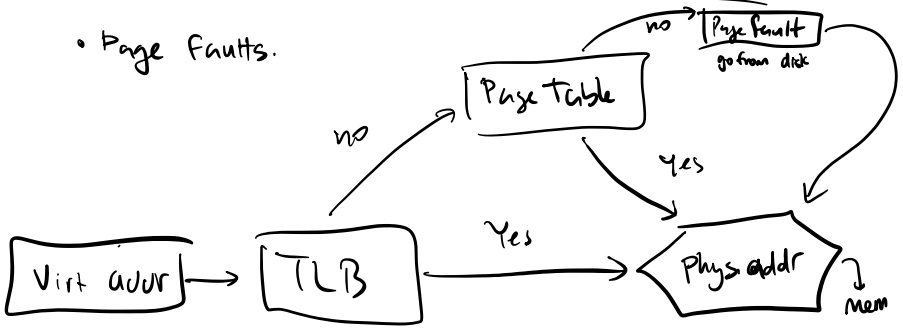
manager: assign work & inform user of progress
workers: performs task from manager.

Needs:

- code that sends tasks w/ procID = 0.
- code that recv's tasks. every other process.

↳ Want to minimize size of message assigning new work.

- Page faults.



MAP REDUCE & SPARK

- Many workers do small parts of large task.

map(in_key, in_value)

↳ (int_key, int_value)

reduce(int_key, int_value)

↳ (out_value)

OpenMPI

↳ continuation of multiprocessing section above.

- initialize multiprocessing: MPI_Init(argc, argv)
- get # of nodes / proc ID (MPI_Comm_size(comm, *size); MPI_Comm_rank(comm, *rank))
  - functions taken in pointers & values written to pointers.
  - returns # the process who successful.
  - Can use MPI_COMM_WORLD for comm value.
- run acc. to worker/manager model?
  - MPI_Recv(*buf, count, datatype, source, tag, comm, *status)
  - MPI_Send(*buf, count, datatype, dest, tag, comm)
    - buf: array of data to be sent / buffer to receive data
    - datatype: constants used to specify input type
    - count: # elements of datatype to receive
    - dest: for send, the procID of intended msg recipient
    - source: for recv, the procID of expected msg sender.
    - tag: for when you want to further classify msgs.
      - send/recv only matches when tags identical
    - comm: communication group; for this class set to MPI_COMM_WORLD
    - status: for recv, contains the source, tag, error msgs of communication.
- clean up w/ MPI_Finalize()

How much faster?

		Pros?	Cons?
reg/func inlining:	< 2x	• easy change • reduces # accesses	• minimal effect • compiler might alr do
loop unrolling:	< 2x	• reduce branching	• minimal effect • penalty to maintainability
Cache opt:	~ 10x	• surprisingly good	• reg's algo changes
SIMD:	~ 8x	• fairly applicable • minimal overhead	• Amdahl • restricted by HW.
open MP.	# cores/node	• more flexible than SIMD or MPI	• concurrency issues • high overhead
open MPI:	# cores.	• can be extended arbitrarily large	• expensive comm. • high overhead.