**architecture**

5 components of a computer: control, datapath, memory, input, output

datapath: performs arithmetic operations

ALU: arithmetic logical unit

performance

**Amdahl's law**: 1 / (1 - F + F / S); F: portion of speed up part; S: speed up

**AMAT** = average memory access time = time for a hit + miss rate \* miss penalty

**local miss rate** L2$ = $L2 misses / $L1 misses

**CPI** with stalls = CPI ideal + instruction miss rate \* miss penalty + %load and stores \* data miss rate \* miss penalty

**CPU time** = Instruction Count \* Clocks Per Instruction with stall \* Clocks Cycle = IC \* CPI / clock rate

**Power** = Capacitive load \* Voltage^2 \* Frequency switched

service interruption: mean time to repair / **MTTR**

reliability: mean time to failure / **MTTF**

availability = MTTF / (MTTF + MTTR)

mean time between failures = **MTBF** = MTTF + MTTR

annualized failure rate = **AFR**

error detection/correction code: **EDC/ECC**

**Hamming distance** = difference in #bits

Hamming distance 2 => **parity**: simple error detection coding

data value before written: tag with extra bit to force the stored word even parity

only detect odd number of errors

Hamming distance 3 => **Hamming Error Correction Code**

encoding

mark all bit positions that are powers of 2 as parity bits (start numbering bits at 1 at left)

position of parity bit k => checks all data bits with kth least significant bid of address = 1

each data bit covered by 2 or more parity bits

set parity bits to create even parity for each group

**correction**: add up incorrect parity bits => location of bad bit

p: total number of parity bits; d: number of data bits in p+d bit word

p >= log(p+d+1)

Hamming distance 4 => adding extra parity bit p' covering the entire word provides double error detection

H=p'=0 no error; H!=0, p'=1 correctable single error; H!=0, p'=0 double error; H=0, p'=1 single error in p'

**cyclic redundancy check**: for block of k bits, transmitter generates an n-k bit frame check sequence; transmits n bits divisible by some number; receiver divides frame by that number. no remainer => assume no error; disks detect and correct blocks of 512 bytes with **Reed Solomon codes**.

**instruction set architecture**

**big-endian**: use the address of the leftmost byte

two's complement: ~x + 1

int: -2^31 ~ 2^31 - 1

**preserved**: $s0-$s7, $sp, $ra, stack above the stack pointer

not preserved: $t0-$t9, $a0-$a3, $v0-$v1, stack below the stack pointer

**cache**

**temporal locality**

**spacial locality** (a block must be bigger than just one word)

**hit time**: time to access the upper level of the memory hierarchy (includes the time to determine hit or miss)

**miss penalty**: the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor

**direct mapped**

**size of offset** = log (block size in bytes)

**size of index** = log (number of blocks)

**size of tag** = #address - #index - #offset

2^n blocks, block size 2^m words => size of tag = 32 - (n + m + 2)

total number of bits in cache: 2^n \* (block size + tag size + valid bit size)

**LRU**: least recently used

hardware keeps track of access history

replace the entry that has not been used for the longest time

for 2-way set-associative cache, need one bit for LRU replacement

**pseudo LRU**

hardware replacement pointer points to one cache entry

whenever access is made to the entry the pointer points to, move the pointer to the next entry

**write-through**

**write allocate** / no write allocate (whether overwrite appropriate portion of block in cache when miss)

**write buffer**

**write-back**

only write to cache; write to lower level when replaced

include **dirty bit** to see if wrote to block or not

**RAID**

RAID 1: mirroring

RAID 3: parity disk sum of other disks mod 2; subtract to recover

RAID 4: block-interleaved parity; allow independent small reads/writes to different disks

RAID 5: interleave parity bits across all disks

RAID 6: add extra check information to recover from 2 failures

data level parallelism

**false sharing**

different processors read & write data in the same cache block

**data race**: 2 memory accesses from different threads to same location

avoid by synchronizing writing and reading

**multiprocessors (MIMD)**

**openmp**: API for multi-threaded, shared memory parallelism

simple parallelization of for-loop

code

#pragma omp parallel for

for (i = 0; i < max; i++) blablabla;

must have canonical shape, no break / return / exit / goto inside loop

all variables declared outside for loop are shared by default

void omp\_set\_num\_threads(NUM\_THREADS);

int omp\_get\_thread\_num();

reduction (operator:var)

declare variable inside for-loop private by default

hardware

**synchronous digital systems**: all operations coordinated by a central clock; represent all values by 2 discrete values.

symbols: buffer, NOT, AND, NAND, OR, NOR, multiplexer

**register transfer language = RTL**

example: (ADDU) R[rd] <- R[rs] + R[rt]; PC <- PC + 4

instruction level parallelism (ILP)

**structural hazards**: required resource is busy

single memory

**pipeline bubble**: instruction fetch has to stall for load/store instruction

solution: provide separate instruction / data memories

register

write during first half and read during second half

or build regfile with independent read and write ports

**data hazards**: need to wait for previous instruction to complete its data read/write

read/write in same cycle can be resolved assuming write in first half & read in second half

**forwarding / bypassing**: use result when it is computed. requires extra connections in the datapath

load-use data hazard: cannot always avoid stalls -> no-op

**control hazard**: deciding on control action depends on previous instruction

simple solution: stall on every branch until new PC value (2 stalls per branch)

optimization #1: insert special branch comparator in stage 2, set new PC as soon as instruction decoded (1 stalls per branch)

optimization #2: predict outcome (simplest to predict NOT taken)

optimization #3: **branch delay slot** (always execute instruction after branch. used by MIPS)

put no-op in the branch slot or place some instruction preceding the branch in the slot

**dynamic branch prediction**

branch prediction buffer / branch history table

indexed by recent branch instruction address

1-bit predictor shortcoming: inner / outer branch

use 2-bit predictor: only change prediction on two successive misprediction

greater instruction level parallelism

**IPC: instructions per cycle**

**VLIW: very long instruction word**

deeper pipeline => less work per stage => shorter clock cycle

**multiple issue** superscalar

replicate pipeline stages => multiple pipelines

multiple instructions per clock cycle

static multiple issue: compiler group instructions that can be issued in a single cycle

compiler must remove some/all hazards

MIPS + static dual issue

1 ALU/branch + 1 load/store

pad unused instruction with noop

**loop unrolling**

replicate loop body to expose more parallelism, reduces loop control overhead

**register renaming**

use different registers per replication

avoid loop-carried anti-dependencies

**dynamic multiple issue**: CPU examines instruction stream and chooses instructions to issue each cycle

**OOO** / out-of-order execution: unroll loops in hardware

in-order fetch instructions / issue -> out-of-order execution -> in-order commit

predict branches as taken / untaken

rename registers using a set of internal registers to avoid hazards

buffer results of executed instructions until predicted branches are resolved in reorder buffer

if predicted branch incorrectly, discard all dependent results and start with correct PC

(system) **virtual machines**: provide interface identical to underlying bare hardware

**virtual machine monitor** **/ hypermonitor / VMM**

maps virtual resources to physical resources: memory, i/o devices, processors

handles real i/o devices, emulates generic virtual i/o devices for guest OS

must intercept attempts by guest OS to access real i/o devices, allocate resources

**virtual memory**: paging. divide memory into equal sized blocks called pages, typically 4KB or 8KB

need special registers to check all addresses to make sure they stay within restricted segment

need mode in processor that says OK to change address limit registers or not OK

typically 2 modes to support VM

problem: results fragmentation of memory (enough memory but not contiguous)

**page table**: maps virtual address to physical address

size: 1~2 MB; keep page tables in main memory; host OS must know where they are and protect them

**page fault**

input transfer into a free page is initiated

if no free page available, a page is selected to be replaced

replaced page is written on the disk (select first empty page on the disk to minimize latency)

page table is updated to point to the new location of the page on the disk

OS must reserve swap space on disk for each virtual machine

to grow a virtual machine, ask OS

if unused pages available, OS uses them first

if not, OS swaps some old pages to disk

Least Recently Used to pick pages to swap

address translation and protection

**virtual page number / VPN** | offset --address translation--> **physical page number / PPN** | offset

protection via page table

access rights: read (only), read/write, execute (only)

valid: invalid means it's on the disk, not in physical memory

VPN -> find page table entry = valid | Access Rights | Physical Page Address + offset -> physical memory address

**page table address translation cache / translation lookaside buffer / TLB**

TLB hit -> single cycle translation; miss -> access page-table to refill

usually fully associative, typically 32-128 entries

each entry maps a large page, so less spatial locality across pages

also page dirty bit, set when any data in page is written

when TLB entry replaced, corresponding page dirty bit is set in page table entry

**comparison with data cache**

**valid bit**: same. if valid = 0, must miss in both

**dirty bit**: different. for cache, means data block has been changed; for TLB, means page corresponding to this TLB entry has been changed

**memory manage unit / MMU**: hardware that walks the page tables and reloads the TLB

**instruction set support**

user and system modes (some instruction only runs in system mode)

page table base register / PTBR: page table addr

privileged instructions only available in system mode (trap to system if executed in user mode)

all physical resources only accessible using privileged instructions (interrupt controls, i/o)

no impact on computation bound OS, big impact on i/o intensive

host OS can also limit amount of time a VM uses a processor, at cost of swapping registers