1 Evolution of Computers

Computer Performance

Clock speed (f): cycles per second, measured in Hz.
Average CPI: $\frac{\sum_i \mathrm{CPI}_i \times \mathrm{I}_i}{\sum_i \mathrm{I}_i}$
Process Time (T): $\frac{(\sum_i \mathrm{I}_i)\times CPI}{f}$
MIPS: $\frac{f}{\mathrm{CPI}\times 10^6}$

2 Digital Logic

Boolean Algebra

$A \oplus B = \overline{A}B + A\overline{B}$
$\overline{A \oplus B} = AB + \overline{A}\,\overline{B}$

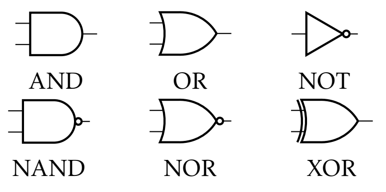
Algebra Laws

$A+0=A$	$A\cdot 1 = A$	Identity Elements
$A+1=1$	$A\cdot 0=0$	Null Law
$A+A=A$	$A\cdot A=A$	Idempotent Law
$A+\overline{A}=1$	$A\cdot\overline{A}=0$	Inverse
$(A+B)+C=A+(B+C)$		Associative (1)
$(A\cdot B)\cdot C=A\cdot(B\cdot C)$		(2)
$A\cdot(B+C)=A\cdot B+A\cdot C$		Distributive (1)
$\boldsymbol{A+(B\cdot C)=(A+B)\cdot(A+C)}$		(2)

De Morgan's Theorem

$\overline{A\cdot B\cdot\cdots\cdot N}=\overline{A}+\overline{B}+\cdots+\overline{N}$
$\overline{A+B+\cdots+N}=\overline{A}\cdot\overline{B}\cdot\cdots\cdot\overline{N}$

Logic Gates



Functional Complete Set

Any boolean function can be implemented by the set.
{AND, OR, NOT} {NAND} {NOR}
{AND, NOT} {OR, NOT}

Implementing Functions

SOP: (1) write 1's as minterms (products of variables), (2) sum minterms.
POS: (1) write 1's as product terms of variables, (2) apply NOT to each term, (3) apply De Morgan's, (4) connect terms with AND.
Karnaugh Map: (1) write map, rows/columns differ by only 1 bit, (2) circle 1's as large, in powers of 2, rectangular, wrap if needed (3) each group is a product, sum groups.

Adders

Half Adder: $S=A\oplus B \quad C=A\cdot B$
Full Adder: $S=A\oplus B\oplus C_{\mathrm{in}}$
$C_{\mathrm{out}}=A\cdot B+C_{\mathrm{in}}\cdot(A\oplus B)$

3 Number Representation

Positional Number System

$\sum_i(a_i r^i)$ Direct ($O(n^2)$)
$r(r(a_n+a_{n-1})+\cdots)+a_0$ Iterative ($O(n)$)

Binary Integers

uint: $\sum_{i=0}^{n-1} 2^i a_i$
Sign-Mag: $(-1)^{a_{n-1}}\sum_{i=0}^{n-2} 2^i a_i$
1's Comp: (if < 0) bit-wise NOT
2's Comp: (if < 0) 1's Comp + 1
$-2^{n-1}+\sum_{i=0}^{n-2} 2^i a_i$
MSB of 1's Comp and 2's Comp is sign bit.

Binary Integer Arithmetics

Negation of 2's Comp

Take 2's Comp of the 2's Comp.

Add/Sub of 2's Comp

Add/Sub directly.
Overflow: Two numbers of same sign added to get oppposite sign.

Multiplication (multiplicand $\times$ multiplier)

+ve×+ve: (1) for each multiplier bit, (2) if 1, shift multiplicand left, add to partial sum, (3) if 0, do nothing, (4) return sum. **Other cases**: (1) for each non-sign multiplier bit, (2) if 1, shift multiplicand left, add to partial sum, (3) if 0, do nothing, (4) for sign bit, if 1, negate multiplicand, left shift, sign extend, add to partial sum, (5) return sum.

Excess-K

Values range: $[0-K, 2^n-1-K]$
K is typically chosen to be $2^{n-1}-1$.

Floating Point Numbers

$\pm\text{Significand}\times 2^{\pm\text{(Biased) Exponent}}$
Single: 32 bits, 8 exp, 23 sig.
Double: 64 bits, 11 exp, 52 sig.
Extended: 80 bits, 15 exp, 112 sig.

Special Values (used only when specified)

0: exp = 0, sig = 0.
Subnormalized: exp = 0, sig ≠ 0.
∞: exp = all 1, sig = 0.
NaN: exp = all 1, sig ≠ 0.

Properties

# of representable numbers same as int.
Not uniformly distributed.
Arithmetic laws not always hold.

Floating Point Arithmetics

Add/Sub

(1) Check 0. (2) Align significand (smaller exp shift right). (3) Add/Sub significands. (4) Normalise. (5) Round.

Multiplication/Division

(1) Check 0. (2) Multiply/Divide significands. (3) Multiplication: add exponents, sub K; Division: sub exponents, add K. (4) Determine sign. (5) Normalise. (6) Round.

Rounding Methods

Round to nearest even.
Round towards zero.
Round towards $\pm\infty$.

4 Instruction Execution Cycle

(1) Instruction address calculation; (2) Instruction fetch; (3) Instruction decode; (4) Operand address calculation; (5) Operand fetch (one or more); (6) Data operation; (7) Operand address calculation; (8) Operand store; (9) Interrupt check.

Operation Format

One word: `[opcode, src1, src2, dest] (register)`
Two word: `[opcode, src, address model, dest], [address (mem)]`

Instruction Fetch

(1) $\mathrm{MAR}\leftarrow\mathrm{PC}$; (2) $\mathrm{PC}\leftarrow\mathrm{PC}+1$;
(3) $\mathrm{MDR}\leftarrow\mathrm{Mem[MAR]}$; PC increment is implied, will change if branch.

Operand Fetch

Operands in registers: $\mathrm{ALU}\leftarrow\mathrm{Reg}$
Operands in memory: (1) $\mathrm{MAR}\leftarrow\mathrm{MBR}$; (2) $\mathrm{MBR}\leftarrow\mathrm{Mem[MAR]}$;

Interrupt Handling

Reasons: (1) Improve efficiency; (2) Prevent data loss (e.g. from network); (3) Other programs need to run (e.g. time-sharing).
Information saved: (1) PC; (2) Modified registers; (3) Flags; (4) Current instruction address.

5 Memory

Memory Hierarchy: Inbound (Registers, on-chip cache, cache, main mem) → Outbound (disk, SSD, DVD) → Off-line (magnetic tape)
Trends (top to bottom): Capacity ↑; Cost per bit ↓; Access time ↑; Frequency of access ↓.
Principle of Locality: **Temporal** (recently accessed likely to be accessed again, e.g. `sum`) and **Spatial** (items with nearby addresses likely to be accessed soon, e.g. `arr[]`).
Memory Organisation: **Big Endian** (left-to-right) and **Little Endian** (right-to-left).
Access Modes: Sequential, Random, Associative.

Internal Memory

ROM: Read-only; Non-volatile; Written by masks; No erasure.
PROM: Read-only; Non-volatile; Written electrically; No erasure.
EPROM: Read-mostly; Non-volatile; Written electrically; Erased by UV light.
EEPROM: Read-mostly; Non-volatile; Written electrically; Erased electrically (byte-wise).
Flash: Read-mostly; Non-volatile; Written electrically; Erased electrically (block-wise); limited write cycles.
DRAM: Read-write; Volatile; Use transistors; Refresh needed; Slow; Cheaper.
SRAM: Read-write; Volatile; Use logic gates; No refresh; Fast; Expensive.

Bench-marking Memory Performance

Access Time: Time to read/write data.
Bandwidth/Transfer Rate: Rate at which data can be read/written.
Memory Cycle Time: Access time + Transfer time.

Cache Memory

A unit-addressable main memory with n-bit addresses, a block size of 2^k units, has $M=2^{n-k}$ blocks. The cache has m blocks (lines), $m \ll M$.

Address Mapping

Direct Mapping: 1-to-1 mapping.
(Cache line) = (Main mem block) % m.
Fields: Tag (remaining bits), Line (r bits, corresponds to 2^r lines), Offset (k bits, corresponds to line size 2^k addressable units)
Pros: (1) Simple circuitry. (2) Fast.
Cons: (1) High miss rate.
Fully Associative: 1-to-all mapping.
Fields: Tag (remaining bits), Offset (k bits, corresponds to line size 2^k addressable units)
Pros: (1) Low miss rate. (2) Flexible use of cache.
Cons: (1) Need to search all lines. (2) Complex circuitry.
Set Associative: 1-to-some mapping.
m (# of lines) $= v$ sets $\times k$ lines/set.
i (Set #) $= j$ (Main mem block) % v.
Implementation: (1) v associative caches. (high associativity) (2) k direct cache. (k-way set associative, low associativity)
Fields: Tag (remaining bits), Set (s bits, corresponds to $v=2^s$ sets), Offset (k bits, corresponds to line size 2^k addressable units)
Pros: (1) Low miss rate.
Cons: (1) Complex circuitry.

Replacement Algorithms

Random: Randomly choose a line to replace. (Not used)
FIFO: Replace the line that has been in the cache the longest.
LRU: Replace the line that has been least recently used.
LFU: Replace the line that has been least frequently used.
Not applicable to direct mapping.

Write Policies

Write-through: Write every time cache is changed.
Write-back: Write only when line is replaced.

Performance

Average Access Time = Hit time + Miss rate × Miss penalty.

Unified/Split Cache

Unified: Instructions and data share the same cache. Auto balanced. Memory contention problem on pipeline and parallel executions, causes bottoleneck.
Split: Instructions and data have fixed-size separate caches. Better performance. Main trend.

Virtual Memory

Physical vs Logical Address: Physical for addressing actual memory, space smaller; logical visible to the program, space may be larger.
Memory Management Unit (MMU): Maps between logical and physical addresses.

Paging

Page vs Frame: Page is a fixed-size block of logical memory, frame is a fixed-size block of physical memory.
Demand Paging: Pages are loaded into memory only when needed.
Page Fault: Occurs when a page is not in memory.
Pros: fast response; less memory usage;
Cons: page faults until stable set of pages loaded.
Page Table: One page table per process, maps logical pages to physical frames.
PTE: (**VPDF**) – **V**alid bit (whether page in memory), **P**rotection bits (manages access rights), **D**irty bit (whether page modified), **F**rame number (physical frame #).
Translation Lookaside Buffer (TLB): Like a cache, stores some valid PTEs. TLB consulted first, if not found, page table is consulted.

External Memory

Hard Disk Drive (HDD)

Components: **Platter** (disk), **Track** (concentric circle), **Sector** (segment of track, 512 bytes), **Cylinder** (set of same tracks vertically).
Sector Format: *e.g.* **Gap 1** (separate sectors) - **ID Field** (synch, track, head, sector #, CRC) - **Gap 2** (separate ID & data) - **Data Field** (data, CRC) - **Gap 3**
Disk Layout Methods: (1) **C**onstant **A**ngular **V**elocity - easy read/write, density decreases towards the rim, wastes space; (2) **M**ultiple **Z**one **R**ecording - zones with different # of sectors, maximise storage capacity, density similar but **NOT** uniform
Data Access Time: (1) **Seek time** - move read/write head to cylinder, distance dependent, 5 - 15 ms startup, 0.2 - 1 ms consecutive; (2) **Rotational latency** - average is half a revolution; (3) **Transfer time** - $t_T \ll \text{seek}+\text{latency}$

$t_T = \frac{\text{bytes to transfer}}{\text{bytes per track}} \times \frac{1}{\text{rotation speed (rps)}}$

Redundant Array of Independent Disks (RAID)

0 (non-redundant): data stored in round-robin fashion, efficient for accessing one block of data, no failure tolerance.

1 (mirroring): multi-disk failure tolerance, either copy can be used $\rightarrow$ reduce seek time, 1 logical write = 2 physical writes.

2 (hamming code): not used, expensive, $\#\text{ redundant disks} \approx \log_2(\#\text{ data disks})$, efficient for parallel with small strip size, universally controlled spindles.

3 (bit-interleaved parity): 1 disk for parity, can recover from 1 disk failure ($p = b_{\text{lost}} \oplus b_2 \oplus b_3 \Leftrightarrow b_{\text{lost}} = b_2 \oplus b_3 \oplus p$), fast read/write, low ECC:Data ratio.

4 (block-level parity): not used, write penalty = 2 reads + 2 writes, methods for writing: (1) write data, recalculate parity, write parity; (2) write data, compare old data with new data, add difference to parity;, individual spindle control, fast read, slow write, low ECC:Data.

5 (block-level distributed parity): 1 disk for parity, parity distributed across all disks, can endure 1 disk failure, common for Network Attached Storage, fastest read/write, low ECC:Data

6 (dual redundancy): 2 disks for parity, can endure 2 disk failures, use two different parity methods, distributed across all disks.

6 Input & Output

I/O Module: interface between processor and memory via system bus or central switch; interface to peripheral devices through dedicated data links.

Model Requirements: (1) asynchronous timing (2) command decoding (*e.g.* SEEK) (3) data exchange (4) status reporting (*e.g.* ready, busy, etc.) (5) address recognition (6) data buffering (speed up transactions) (7) error detection & correction

I/O Register Mapping

Memory-mapped: registers mapped into main memory address space; accessed as if memory locations;

I/O-mapped: mapped into separate address space; accessed via special instructions.

I/O Techniques

Programmed I/O: Not using interrupts. CPU waits for I/O device to complete operation. CPU accesses device via Control and Status Registers (CSR). Wastes CPU time.

Interrupt-driven I/O: (Memory $\leftrightarrow$ CPU $\leftrightarrow$ I/O) CPU executes other instructions after sending I/O command. I/O interrupts CPU when complete. CPU sends acknowledgment (`INTA`) to I/O. Interruptions handled **between** instruction cycles.

Direct Memory Access (DMA): (Memory $\leftrightarrow$ I/O) Use Input-Output Processor (IOP). IOP steals cycles from CPU. CPU sees elongated cycle and wait until cycle is over. Interruptions handled **within** one instruction cycle.

7 Instruction Sets

Arithmetic Operations: treat operands as numbers; consider signs; (*e.g.* arithmetic shift = multiplication/division by 2, sign bit preserved).

Logical Operations: treat operands as bit patterns; discard bits shifted out; replenish new bits with 0.

Rotate Operations: put bits shifted out back into the other end of the number; are logical operations.

Instruction Operands

Op#	Symbolic	Interpretation
3	`OP A, B, C`	`A`←`B OP C`
2	`OP A, B`	`A`←`A OP B`
1	`OP A`	`AC`←`AC OP A`
0	`OP`	`T`←`(T-1) OP T`

Registers

General Purpose Registers: can be used for whatever reason

Dedicated Purpose Registers: have a specific purpose (*e.g.* PC, IR, SP, processor status word - PSW, flag)

Data Types

Basic Data Types

Typical lengths: 8, 16, 32, 64 bits

Numeric: integer, floating point;

Non-numeric: character, binary data;

MIPS Architecture

(family of RISC, not ARM nor x86)

9 basic types: (1) (un)/signed bytes; (2) (un)/signed half-words; (3) (un)/signed words; (4) double-words; (5) single-precision floating point (32 bits); (6) double-precision floating point (64 bits);

ARM Architecture

Supported lengths: (1) byte (8 bits); (2) half-word (16 bits); (3) word (32 bits); Only unsigned integers, nonnegative integers, and 2's comp integers.

No floating point by hardware, must be emulated.

Addressing Modes

Immediate (`OP = A`): operand is value; **Pros**: no memory reference; **Cons**: small operand magnitude

Direct (`EA = A`): operand is address; **Pros**: fast & increased magnitude; **Cons**: limited address space

Indirect (`EA = (A)`): operand is address of address; **Pros**: large address space; **Cons**: multiple memory references

Register (`EA = R`): operand points to register; **Pros**: fast; **Cons**: limited # of registers (*e.g.* 32 in MIPS)

Register Indirect (`EA = (R)`): operand points to register, register has address; **Pros & Cons**: same as indirect

Displacement (`EA = A + (R)`): address is base address + offset; **Pros**: flexible; **Cons**: complex; Usage: local vars, arrays; Registers: PC, SP, base pointer register

Stack (`EA = Top of Stack`): implicit; **Pros**: no memory references; **Cons**: limited applicability; Usage: `PUSH` and `POP`; Register: SP

Assembly Language

Syntax

```
[LABEL:] OP_NAME [OP_1, OP_2,
...] [# COMMENT]
```

Assembler Directives

`.data`	data segment
`.text`	program segment
`.global NAME`	introduce to other files
`.reserve EXPR`	reserve space with 0
`.word VAL1[, ...]`	write to memory

OS Support

OS Services: (1) Program creation (compilers), execution (loading, managing resources), (2) I/O access (provide uniform interface, implementation hidden), (3) File system management, (4) System access (user/kernel mode), (5) Error detection & response (BSOD), (6) Accounting (collect usage stats)

System Calls: special entry points to execute OS functions via kernel model (executed by OS on behalf of user program)

Scheduling & Time Sharing: CPU time divided into time slices, each process gets a slice, when used up, process is suspended and another process is scheduled. OS maintains a priority queue, depends on waiting time, system load, CPU-bound etc.

Processor Pipelining

Ideal throughput = 1 CPI.

Pipeline Hazards

Resource: multiple instructions need the same resource (PC, ALU, registers). **Solution**: add more resources

Data: instruction depends on result of previous instruction – (1) **RAW**: occurs if read happens before write when it should be RAW; (2) **WAR**: opposite of RAW, not occurs in pipeline but in parallel systems; (3) **WAW**: occurs if 2nd write happens before 1st, not in pipeline but in parallel systems. **Solutions**: (1) **Stalling**: wait; (2) **Data forwarding**: use result of previous instruction; (3) **Rearrange instructions**: separate dependent instructions, not always possible;

Control: branch not resolved. **Unsolvable**. Mitigated by branch prediction.

Branch Prediction

Static: always predict taken/not taken. **Pros**: simple, 50% accuracy, higher in `for` loops. **Cons**: possible high page faults.

Dynamic: Use history to help predict. (1) **1-bit**: If wrong prediction, predict opposite; Problem: high misprediction rate at the end of loops; (2) **2-bit**: If two consecutive wrong predictions, predict opposite.

RISC Architecture

Characteristics: (1) Load/store architecture: only load/store instructions access memory; (2) Fixed-length, simple, fixed-format instructions $\Rightarrow$ high clock rate, low clock cycle time; (3) Fewer addressing modes $\Rightarrow$ low CPI; (4) More instructions (reduction in CPI is more significant than increase in instruction count); (5) Extensive soft/hardware pipelining; (6) Relies on compiler optimisation.