Some Formulae:

Power = Capacity · Voltage^2 · Frequency$_{0\to1}$ + Voltage · I_{leakage}

Power = $\frac{\text{Joules}}{\text{Op}} \cdot \frac{\text{Ops}}{\text{Second}}$

Dennard scaling: 0.7x voltage drop

Latency: How long it takes to do a task

Throughput: Total work done per unit of time e.g. queries/sec

Clock period: Duration of a clock cycle

Clock frequency: cycles per second

Response time $T = T_0 \frac{\rho}{1-\rho}$ with ρ = percentage of throughput

Execution Time = Cycles per Program · Clock Cycle Time = $\frac{\text{Cycles Per Program}}{\text{Clock Rate}}$

Clock Cycles = Instruction Count · Cycles per Instruction

CPU Time = Instruction Count · CPI · Clock Cycle Time = $\frac{IC \cdot CPI}{\text{Clock Rate}}$

Exec Time = $\frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock Cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Clock Cycle}}$

Clock Cycles = $\sum_{i=1}^{n} (CPI_i \cdot IC_i)$

Weighted average CPI = $\frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(CPI_i \cdot \frac{IC_i}{IC}\right)$

Performance = $\frac{1}{\text{Exec Time}}$

"X is n faster than Y" means $\frac{Perf_X}{Perf_Y} = \frac{Exec_Y}{Exec_X} = n$

Energy = Average Power x Execution Time

Optimize for Energy per Instruction: $Power = \frac{energy}{second} = \frac{energy}{instruction} \cdot \frac{instructions}{second}$

Amdahl's Law: $Speedup = \frac{CPUTime_{old}}{CPUTime_{new}} = \frac{CPUTime_{old}}{CPUTime_{old}[(1-f_x)+\frac{f_x}{S_x}]} = \frac{1}{(1-f_x)+\frac{f_x}{s_x}}$

Parallel Speedup: $Speedup = \frac{1}{(1-P)+\frac{P}{n}} \to P = \frac{\frac{1}{Speedup}-1}{\frac{1}{n}-1}$

Arithmetic Mean: $\frac{1}{n}\sum_{i=1}^{n} T_i$ used with times, not rates

Harmonic mean: $\frac{n}{\sum_{i=1}^{n}\frac{1}{R_i}}$ used with rates not times

Geometric mean: $\left(\prod_{i=1}^{n}\frac{T_i}{T_{ri}}\right)^{\frac{1}{n}} = \exp\left(\frac{1}{n}\sum_{i=1}^{n}\log\left(\frac{T_i}{T_{ri}}\right)\right)$

Power/performance benchmark: Overall ssj_ops per Watt = $\frac{\left(\sum_{i=0}^{10} ssj\_ops_i\right)}{\left(\sum_{i=0}^{10} Power_i\right)}$

Perf metrics:

`time <application>` measures execution time

`perf record` does low overhead sampling

`perf topdown` uses hardware performance counters

Flip flop setup time: $t_{ck} > t_{pd} + t_s + t_{skew}$

CISC:

Multi-cycle complex instructions; Load/store incorporated in instruction; Small code size; High CPI; Low clock frequency; Variable length instructions

RISC:

Simple (single-clock) Instructions; Register-to-register separate load instructions; Large code size; Low CPI; High clock frequency; Same length instructions; Simple instruction decode;

Call and Return:

Caller:

Save caller-saved registers as needed

Load arguments

Execute `JAL`

Callee setup:

Allocate memory for new frame (`xsp` = `xsp` - frame)

Save callee-saved registers as needed

Set frame pointer (`xfp` = `xsp` + frame size - 4)

Callee return:

Place return values in `x10` and `x11`

Restore any callee-saved registers

Pop stack (`xsp` = `xsp` + frame size)

Return by `jr xra`

Caller:

Restore any caller-saved registers as needed

Pipeline stages:

`IF`: Instruction fetch

`ID`: Instruction decode, register read

`EX`: execute operation or calculate address

`MEM`: Access memory command

`WB`: Write result back to register

Hazards:

Structure Hazards: A required hardware resource is busy

Data hazards: Must wait for previous instructions to produce/consume data

Read after Write: Instruction j tries to read before instruction i tries to write it

Write after Write: Instruction j tries to write an operand before i writes its value

Write after Read: Instruction j tries to write a destination before it is read by i

Control hazards: Next `PC` depends on current instruction result

Forwarding:

Identify *producers*: `EX` and `MEM` stages

All stages after first producer are sources of forwarding data: `MEM` and `WB`

Identify *consumers*: `EX` and `MEM`

These are the destinations of forwarded data

Branch Prediction:

Static: predict not-taken

Dynamic:

Branch History Table: one entry for each branch, taken/not taken record

Branch Target Buffer: One entry for each branch, computes target address

Exceptions and Interrupts:

Switch from 'user' to 'kernel' mode

Exceptions: Arises within CPU. Undefined opcode, overflow, etc

Interrupt: External I/O controller, network card, etc

On exceptions:

Pass to relevant handler

Then return to program using EPC if possible

All previous instructions completed

Faulting instruction not started

No side effects

Nullifying instructions: Converts them to a NOP

Going past the 5-stage pipeline:

Instruction-level Parallelism: Independence among instructions; Fetch multiple instructions per cycle; Evaluate which ones can go down the pipe together

Superscalar: Double up on hardware in order to execute multiple instructions simultaneously

Deeper Pipelines: Increase the number of stages, decrease amount of logic. Higher clock freq.

Register Renaming: Map architectural registers to physical registers in decode stage to get rid of false dependencies. Need more physical registers than architectural ones.

Data flow graph:

Scoreboard: bit array, 1-bit for each GPR

If the bit is not set, the reg has valid data

If the bit is set, the reg has stale data

Dispatch in order: RD ← Fn(RS,RT)

If `SB[RS]` or `SB[RT]` is set → RAW, stall

If `SB[RD]` is set → WAW, stall

Else dispatch to functional unit, set `SB[RD]`

Complete out-of-order

Update `GPR[RD]`, clear `SB[RD]`

Caching:

AMAT: Access Time = hit time + miss rate · miss penalty

Three C's of misses:

Compulsory: First time you've accessed this item

Capacity: Not enough room in the cache to hold item

Conflict: Item was replaced because of a conflict in its set

Direct-mapped cache:

2^n bytes total with 2^m byte blocks

Byte select: lower m bits

Cache index: lower $(n-m)$ bits of the memory address

Cache tag: upper $32-n$ bits of the memory address

N-way set associative:

Each memory block can go to one of N entries in the cache

2^n-byte cache, 2^m-byte blocks, 2^a set-associative:

Cache contains $2^n/2^m = 2^{n-m}$ blocks

Each cache way contains $2^{n-m}/2^a = 2^{n-m-a}$ blocks

Byte offset: lowest m bits

Cache index: next $n-a$ bits

Associative caches might use Least Recently Used table for evictions

For N-way cache, $N!$ orderings

Write-through:

Main memory updated each cache write

Replacing a cache entry just overwrites new block

Memory write may cause pipeline stalls

Misses are simpler and cheaper

Uses a write buffer — FIFO queue

Write-back:

Only the cache entry is updated on each cache write

Cache and memory entries are inconsistent

Add 'dirty' bit to indicate whether memory needs to be updated

Write new value to memory on evictions

Writes are super fast

Write miss options: Do you allocate for space in the cache on a miss?

Do you fetch the rest of the block contents from memory?

For no-fetch-on-miss must use fine-grained valid bits

Write-back: typically write-allocate, fetch-on-miss

Multilevel caches:

Primary L1 caches attached to CPU: small, fast, focuses on hit time

Secondary L2 caches service misses from L1: larger, slower, still faster than DRAM

Cache Coherency:

P writes X, P reads X → read returns written value

P_1 writes X, P_2 reads X later → read returns written value

P_1 writes X, P_2 writes X → all processors see writes in the same order

Single-Writer, Multiple-Read Invariant: For any memory location A, at any given epoch, there exists only one CPU that may write to A or some number of CPUs that may only read A

Data-Value Invariant: The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch

CPU uses snooping protocols to ensure this:

2-state: very simple hardware and protocol. Write-through, all writes go on interconnect bus.

3-state MSI: Modified (one cache has valid copy), Shared (one or more have read-only copy), Invalid (invalidated so one copy can go to modify state)

DRAM:

SRAM is 6 transistors, static, doesn't need refresh, used for caches. DRAM is 1 transistor + 1 capacitor, needs refresh every 10ms, much higher capacity. DRAM outputs data in bursts (I guess x8 at a time?). DRAM accesses:

`ACTIVATE` to open a row

`READ` to read a column

`WRITE` to write to a column

`PRECHARGE` to close a row

`REFRESH` on timed interval

In order to access a 'closed' row, must first `PRECHARGE` the row that is open, then `ACTIVATE` the row to be read, then `READ/WRITE` to access row buffer. Modern DRAM has BANKS which operate independently of each other but use the same access hardware. Each bank can have a different row active. The `ACTIVATE` and `PRECHARGE` latencies can overlap. Physical constraints:

`tRCD` = Row to Column command delay (how long it takes row to get to sense amp)

`tCAS` = Time between column command and data out

`tCCD` = time between column commands (rate that these can be pipelined)

`tRP` = time to precharge DRAM array

`tRAS` = time between row activation and data restoration (minimum time row must be open)

`tRC = tRAS + tRP` = row 'cycle' time

Timing constraints:

CPU → memory controller transfer time

controller latency

DRAM bank latency:

`tCAS` if row is open

`tRCD + tCAS` if array is precharged

`tRP + tRCD + tCAS` worst case

DRAM transfer time = BurstLen/(MT/s)

Memory controller → CPU transfer time

Virtual Memory:

OS maps virtual addresses to physical memory using a page table. One page table per process. If a page isn't mapped it generates a page fault, which traps to OS, to map a new page. Translation Lookaside Buffer `TLB` is a hardware cache for page tables. Each `TLB` entry stores a page table entry `PTE`. `TLB` is like a cache and uses tag, valid, dirty bits. Page size depends on application, OS and kernel use large pages but user applications frequently use smaller. Page table size is `(num pages) * (page entry width)`. Hierarchical page tables: First level stores directory, second level stores actual page entries. Only top level needs to be resident in memory. Abbreviations:

`MMU`: Memory Management Unit controls `TLB`

Components of the virtual address `VA`:

`TLBI`: TLB index

`TLBT`: TLB tag

`VPO`: Virtual page offset

`VPN`: virtual page number

Components of the physical address `PA`:

`PPO`: phys. page offset

`PPN`: phys. page number

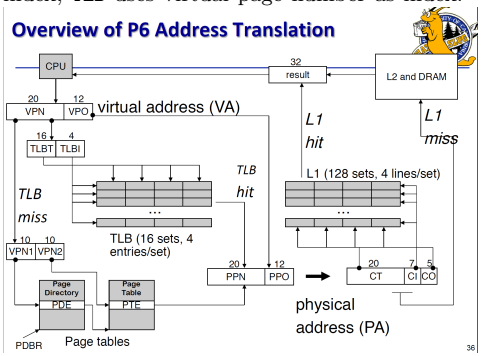
`CO`: byte offset within cache line

`CI`: cache index

`CT`: cache tag

VIPT:

`VIPT` uses virtual address bits for indexing and physical address bits for tag. The cache uses the physical address as the tag. The `TLB` uses the virtual address as the index and has the tag stored as its data. An access means indexing in the `TLB`, getting the `TLB` data, indexing in the cache, getting the cache tag, and comparing the two. Cache uses the page offset as index, `TLB` uses virtual page number as index.



I/O

I/O is measured by `throughput` (amount of data/time) and `latency` (response time to a single I/O operation). Common approach to I/O is use memory mapping. Devices separated into 'block devices' and 'stream devices'. Use `lseek(fp, offset, reference)`, `read(fp, buf, nbytes)`, `write(fp, buf, nbytes)`. Stream devices don't use addressing. Device drivers are low-level code to hide the interface. Hard drive accesses:

Queue delay if other requests pending

Seek: move the heads (time given)

Rotational latency ($\frac{1}{2}$/ rotational speed)

Data transfer (given)

Controller overheads (given)

A bus is a shared communication link between devices with wires all connected in parallel. Can be synchronous (uses a clock) or not. Can increase transmission bandwidth using separate control/data lines, wider lines, block transfers. An initiator acquires the bus (may require arbiter), starts the transaction, specifies command, specifies address, specifies outbound data for write. Target responds to the action with data as necessary. Processory-memory bus (or processor-processor) is short, high-speed, designed to match memory system. I/O bus is usually long and slow.

PCI

`CLK` clock cycle

`FRAME` is asserted by the device controlling the bus

`AD` is the address of the target device and then holds responding data

`C/BE` is the command, then the ID of the byte needed

`IRDY` says the initiator is ready to receive data

`TRDY` says the target is ready to transmit data

`DEVSEL` says the target is connected

DMA

Direct Memory Access is a custom engine for data interface. Uses custom registers `from`, `to`, and `length`. Also has registers for status and control. Typically notifies processor operation is complete via interrupt. DMA engine will typically chain multiple operations such that ≤ 1 page is loaded at a time. DMA will use a `TLB` with virtual addresses.

Vectors

Implement data-level parallelism using vector instructions. Operate on vector registers. Perform single operations on multiple data elements. Amortizes instruction cost. Vector ALUs will be run in parallel in order to increase bandwidth. Domain specific hardware implements convolution engine to perform specific set of calculations really quickly. Uses a DMA to load/store data.

FPGA

Field-programmable Gate Array is a 2-dimensional array of reconfigurable logic elements which implements Look-up tables, flip-flops, adders, Block RAM, and DSP blocks. Uses switch programmable interconnect to connect between array elements. Has I/O interface at chip boundary.

Multicore:

Use shared memory accessed through loads/-stores. Synchronization through atomic instructions `lr.w` and `st.w` which provide locking information. Multithreading executes more threads than cores. When 1 stalls, switch to another. Fine-grain MT switches threads each cycle. Coarse-grain MT switches threads on a long stall. Simultaneous MT uses multiple-issue dynamically scheduled pipeline hardware to exploit thread-level parallelism so it'll issue them as fast as it can.