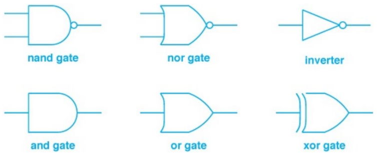
Digital logic



Half adder: takes two input bits, outputs sum (xor) and carry (and).
Full adder: takes two input bits and a carry, outputs sum and carry. Built from two half adders and an or gate.
Latch: uses previous output until the enable line is enabled. The output remains the same (even when the enable line is on) until propagation delay is over.
Propagation delay: the delay between an input change an an output change.

Representations

Unsigned integers: only positive integers from 0 to $2^k - 1$.
Sign-magnitude: MSB is sign, rest is magnitude. From 0 to $2^{k-1} - 1$ and from -0 to $-2^{k-1} - 1$.
One's complement: MSB is sign, rest is magnitude. Flipped for negative. From 0 to $2^{k-1} - 1$ and from -0 to $-2^{k-1} - 1$.
Two's complement: MSB is sign, rest is magnitude. Flipped - 1 for negative. From 0 to $2^{k-1} - 1$ and from -1 to -2^{k-1}.
Floating-point (IEEE): stored as `sign-exponent-mantissa`: $(-1)^s \cdot (1 + m) \cdot 2^{e-b}$ where s is the sign, m is the mantissa, e is the biased exponent and b is the bias. The exponent is biased so it can have negative numbers while being unsigned: $b = 2^{k-1} - 1$ where k is the number of bits the exponent has. So, $e' = e + b$ where e is the unbiased exponent. The mantissa is *normalized* and is assumed to start with 1.

Processors

Von Neumann architecture: single memory to hold both instructions and data. More flexible, but bottleneck between memory to processor. More popular than Harvard.
Harvard architecture: separate memory for instructions and data. Less susceptible to bottlenecks and more secure, but the split between instructions and data memory is rigid.
Processor: a digital device that can perform multi-step computation.
Types:

- **Fixed logic**: function is fixed in hardware.
- **Selectable logic**: can choose one of several fixed functions.
- **Parameterizable logic**: parameters control computation of fixed functions.
- **Programmable logic**: list of instructions provided at runtime.

Categories:

- **Co-processors**: dedicated function; fixed/selectable logic.
- **Microcontrollers**: programmable logic, direct hardware control.
- **Embedded systems**: real-time OS, dedicated hardware.
- **General purpose**: more functionality than embedded systems.

Units:

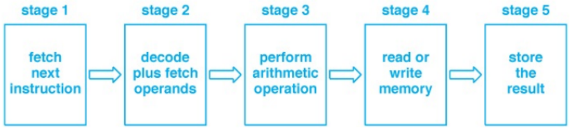
- **Controller**: controls the flow of program execution.
- **Arithmetic logic unit**: performs all computational tasks sequentially according to controller.
- **Local storage (registers)**: holds data values.
- **Internal connections**: Transfers data values between hardware units (between local storage and ALU).
- **External interface**: handles communication between processors and the rest of the computer system.

Fetch-execute cycle

The *instruction pointer* `ip` automatically moves through the program in memory. The cycle *doesn't proceed at a fixed rate* since execution speed depends on the instruction. The cycle *never ends* (unless the system powers down) so there always needs to be a program executing (either the same program or OS).

Instructions

Instruction sets: the *design* of all available instructions for a processor (available instructions, what each instruction does, number of operands, operand encoding, instruction encoding, result storing, etc).
Branching: moving the *instruction pointer* to a different location in the program (e.g. for conditionals, loops, etc.). **Register bank**: each *bank* stores a set of registers. Allows parallel access to different registers within the same clock cycle. Can lead to register bank conflicts if an operation requires multiple registers from the same bank.
Passing arguments to subroutine calls: either stored in *memory* or *registers* through *register windows*.
Register window: exposing only a set of registers at a time (i.e. a window), which moves each time a subroutine is called and moves back when it returns. The window overlaps with what the program sees and what the subroutine sees so *arguments are passed into the overlapping registers*.
Complex instruction set computer (CISC): a kind of ISA where each instruction performs a *complex operation*, instructions take *variable number* of clock cycles, and requires *fewer instruction calls* than a RISC.
Reduced instruction set computer (RISC): a kind of ISA where each instruction performs a *simple operation*, instructions take the *same number* of clock cycles (achieved through the RISC pipeline), and requires *many instruction calls*.
Pipeline: a computational architecture that improves throughput if computation can be divided into *distinct stages*, each stage takes *similar time* to complete, and it's *efficient* to move data between stages.
RISC pipeline: a 5 stage pipeline for RISC that allows one instruction to execute per clock cycle **once the pipeline is filled**.



Stall: disrupts the flow of a pipeline. Creates a *bubble* that floats until the pipeline restarts. Also called a *hazard*.
Bubble: a delay in the pipeline of one or more clock cycles due to a stall.
Data hazard: a stall from an instruction *waiting for data* from an earlier instruction. Solved by *forwarding* between stages or *rearranging instructions* so instructions requiring data from earlier instructions occur later.
Forwarding: pass results from earlier instructions into later instructions that require it *right after it's calculated* rather than once it's done being stored.
Control hazard: a stall from an *incorrect instruction* in the pipeline, causing *branching*. Solved by predicting conditional branching and flushing the pipeline if the prediction is wrong.
Structural hazard: a stall from *resource conflicts* where a required resource is needed simultaneously by more than one instruction. Solved by loading data in parallel like in *register banks*.

Operands

The arguments of instructions. Can either be `src` (value, register, address) or `dst` (register, address).
There can be *fixed* (bit fields have same semantics) and *variable* (more memory efficient) number of operands.

1. **0 operands**: stack architecture.
2. **1 operands**: implicit destination.
3. **2 operands**: specified destination, given operand is destination.
4. **3 operands**: specified destination, can have separate destination from inputs.

Implicit encoding: opcodes define what type of operands we need.
Explicit encoding: operand fields define what type of operands.
Different ways operands can be accessed:

1. *Immediate value* in instruction.
2. Value in register, *register id* in instruction.
3. Value in memory, *memory address* in instruction.
4. Value in memory, *memory address in register*, register id in instruction. I.e. pointer.
5. Value in memory, memory address in *different memory location* encoded in instruction. I.e. double pointer.

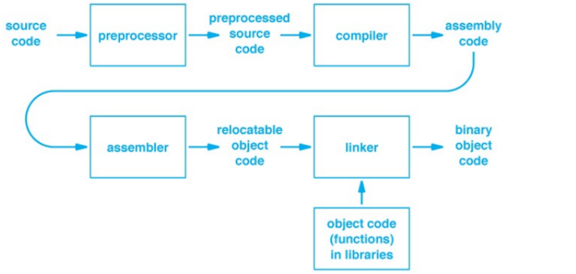
Microcode

CPU and programmer views the macro instruction set, which is implemented with microcode that is hidden to the programmer. Micro instruction set is implemented with digital logic and is processed by the microcontroller.
Pros: less prone to errors compared to hardware design; easier to implement; ease of extension and modification; offers another level of abstraction.
Cons: more overhead than dedicated hardware design; variable cost for macro instructions (depends on number of micro instructions); microcontroller needs to run very fast.
Vertical microcode: sequential coding where you control one functional unit at a time. Microcontroller runs the instructions in sequence. More intuitive.
Horizontal microcode: each instruction is a bit string that controls multiple functional units simultaneously. Some operations can take multiple cycles and you have to tell the ALU to `continue`. As long as resources aren't blocked, parallel execution is possible.
Lookahead execution: microcontroller can look ahead to next instruction and perform optimization at the micro instruction level, but need to preserve computation semantics (result of computation stays the same). Lookahead for conditional branching does branch prediction or execution depending on hardware.

Compilation

High-level languages are *compiled* into binary *object code* for the processor to compute through the fetch-execute cycle.

1. **Preprocessor**: expands macros.
2. **Compiler**: translates to assembly language.
3. **Assembler**: translates to relocatable object code (contains references to external library functions).
4. **Linker**: replaces external function references with its code.



High-level language: one line to many instruction translation; hardware independent (can be compiled in either x86-64 or ARM); usability: application oriented, general purpose, and powerful abstractions.
Low-level language: one line to one instruction translation; hardware specific (code is specific to x86-64 or ARM); usability: systems programming oriented, special purpose, and few abstractions.
Two-stage assembler: a kind of assembler for assembly where step 0 is running the preprocessor; step 1 is mapping each instruction to a memory address and building a label symbol table; and step 2 is translating instructions into binary and substituting labels with their location in the symbol table.

Physical memory

Memory: any physical storage device that can *store and retrieve* information. CPU doesn't get memory directly, it asks the memory controller (a microcontroller).
Volatile: when power is off, values are gone (e.g. RAM).
Persistent: maintain values when power is off (e.g. SSD, HDD).
Random access: cost of memory access is the same regardless of the location in memory (e.g. RAM).
Sequential: order matters in cost of memory access (e.g. CD). Includes a location pointer, which requires time to go from A to B.
Access rights: read-write vs. read-only.
Primary: computer's main memory and stores data temporarily (e.g. RAM and registers).
Secondary: external memory and saves data permanently (e.g. SSD, HDD).
Density: how much data can we store per unit chip area?
Latency: time to complete a single operation.
Cycle time: average time per operation.
Clock frequency: memory is managed by a microcontroller at a certain clock frequency. **Synchronous**: same frequency as CPU (results in better cycle times).
Asynchronous: different frequency from CPU.

Memory hierarchy

Lowers latency, increases capacity, and lowers cost and power consumption. Arguments (inputs) go from storage ↔ memory ↔ registers ↔ ALU; results (outputs) go opposite direction.

RAM

SRAM (static): high-power consumption (constantly supply voltage since it doesn't have a refresh) but lower latency and higher access speed.
DRAM (dynamic): low-power consumption (set the charge once and refresh supplies charge again), but higher latency and slower access speed (when input and refresh are different values, write enable has to prevent refresh from executing and synchronization).

Memory organization

A memory bus acts as a direct line between the CPU and microcontroller; a parallel interface with lines dependent on the interface width (word size). Memory is word addressable.

Byte addressing

Virtual addresses at processor level are byte addresses, but actually accessing it returns the word address. So, it goes inside the word to grab the byte. If byte address is b and word size is N (in bytes), word address $= b$ div N and offset $=$ b mod N.

Memory alignment

A data type stored in memory is aligned when it takes up only the required bytes. E.g. an integer stored in one byte is aligned, but if its upper bits and lower bits are split between two bytes, then it's non-aligned.

Memory address space

The amount of physical memory we can have in an architecture depends on the size of addresses and the word-size of the addresses. KBit $= 10^3$ bits, MBit $=$ 10^6 bits, GBit $= 10^9$ bits, byte $= 2^3$ bits, Kb $= 2^{10}$ bytes $= 2^{13}$ bits, Mb $= 2^{20}$ bytes, Gb $= 2^{30}$ bytes. If a word is 32 bits, then there's 2^{32} unique addresses (words) and 4 bytes in each word so $2^4 \times 2^{30}$ memory size, so 16 Gb of memory.

Memory management unit (MMU)

Multiple physical memory chips (modules) can connect to one common interface, the MMU. MMU provides a common address space to the processor and implements virtual memory and caching.
Sequential mode: each module is combined vertically so 0 to N on module 1, $N + 1$ to $2N$ on module 2, ...
Interleaved mode: each module is combined horizontally so 0 is on module 1, 1 on module 2, 2 on module 3, ...

Caching

Used when data is read/written repeatedly (between registers and memory) to speed up read/write access. Helps with Von Neumann architecture's bottleneck. A system that acts as an intermediary between a data producer and a data consumer. A cache intercept requests and handles them on behalf of the data producer. The cache is located closer to the data producer than the data consumer for faster access. Cache is also volatile memory. Cache is a small intermediary storage so it can't store as much as the data consumer.
Small: size of cache is much smaller than the size of data at producer.
Active: contains an algorithm that makes decisions and handles requests.
Transparent: producer and consumer are unaware of cache's existence.
Automatic: not controlled by an external mechanism.
Caches can be highly parallelized and are arranged in a hierarchy. For each CPU core, they each have their own L1 (level 1) cache, then a shared L2 cache in the CPU and then an L3 cache outside of the CPU.

Cache performance

Given a cache C, let C_h be the cost of cache hit and C_m be the cost of a cache miss.
Locality of reference: the same element being requested frequently.
Caching works better when locality of reference is high since when it's low, C_m is high. Let r, the hit ratio be $r = N_{hit}/(N_{hit} + N_{miss})$ then the expected cost of a lookup is $C_{cache} = r \times C_h + (1 - r) \times C_m$ and the cost without cache is C_m so the benefit of using a cache is, on average, $C_m - C_{cache} = r \times (C_m - C_h) > 0$. When $C_m > C_h$ and $r > 0$ a cache always improves performance. Better performance with higher r (higher locality of reference) and lower C_h (closer to requester = smaller storage); competing objectives since higher r requires larger storage. Cache performance improvement through sequential access of allocated memory, aligned memory allocations and group access operations; last two are usually handled by the compiler. Multiple caches always improve performance as long as for each C_h, $C_h < C_m$ and for each r, $r > 0$.

Cache replacement policy

The cache stores items that we expect to be hit frequently according to the cache replacement policies. When the cache is full, either replace the least *recently* used item or the least *frequently* used item.

Cache maintenance

Write-through: as soon as a value is modified, update back along the chain. If L1 cache changes a value, update L2, then L3, then memory.
Write-back: set a *dirty bit* when a value is modified; only write (update down the cache levels) when the value is replaced in the cache.
Both strategies have a **cache coherence protocol** (since L1's will be different). When multiple processors are accessing and modifying the same value, L1's need to be the same. All caches need to see write (store) operations in the same order. All caches need to be informed immediately when a value is changed. Implement a connection between L1 caches.
Common directory: whenever one cache wants to change a value, it goes into the directory and notifies all caches in the same directory.
Snooping: all of the caches monitor a sequence of write operations being done by their counterparts.
In some cases, a cache may need to be flushed.

Direct mapped memory cache (DMMC)

Maintain several cache lines containing words from memory.
Tag: a section of contiguous blocks (e.g. 4 blocks).
Block: a section of contiguous addresses (e.g. 8 sequential word addresses).
Cache line: a sequence of words in a block in a tag. E.g. 64-bit architecture with DMMC of 128 lines where each line is 16 words long. 127 lines = 2^7 blocks, so 7 bits for block id; 16 words = 2^4, so 4 bits for offset; tag id = 64 - 7 - 4 = 53 bits. 2^7 lines, each line 2^4 words, each word 2^3 bytes, so overall 2^{14} bytes of data storage or 16 Kb plus storage for tags and value bits.

Set associative memory cache

Look through K DMMCs in parallel. Store multiple lines with same block ID but different tag ID. When a request is received, loop through the K DMMCs and if we have a hit, return it and we can avoid looking at the remaining DMMCs.
Fully associative cache: each DMMC holds a single line. Helps with implementing least recently used replacement since we can pass data into the next cache.

Virtual memory

Addresses between CPU and MMU are in a virtual address space (byte or word addressable memory). MMU talks to each physical memory in physical address space (word addressable memory).
Pros: homogeneous integration of hardware (no need to change software when changing hardware); programming convenience (can refer to relative addresses rather than absolute); support for multi-programming (don't want multiple programs to access the same memory; MMU translates multiple virtual spaces to multiple physical spaces).
Caching: caches hold virtual addresses with a process ID. When switching processes, the cache is usually flushed. `[address used by cache] = [ID (process ID][virtual address (program memory)]`.

Base-bound registers

MMU keeps track of mapping between virtual memory space i and a pair $(base, bound)$ of registers on the MMU. $base$ points to a memory address that maps to virtual address 0. $bound$ can be scaled dynamically and points to the end of the memory space.
Cons: doesn't allow allocation of more virtual memory than available physical memory; don't know where to place this memory; only works for a very limited number of virtual spaces.

Segmentation

Split up program memory space into variable size chunks. Chunks form a virtual memory space which gets mapped onto the physical memory space. Load each chunk when it's being used according to the program counter location and unload when it's not being used.
Cons: how to choose optimal segment splits (where and how many); how to avoid memory fragmentation (enough free memory but not large enough to fit a segment).

Demand paging

Split up program memory space into fixed size chunks called pages. Requires cooperation between software (OS; configure hardware; monitor page use; move pages between memory and hardware [swapping]) and hardware (MMU; handle address mapping from virtual to physical; record when a page is used; detect access attempts to a missing page).
Frame: a corresponding slot in physical memory of the same fixed size as a page.
Load: move contents of a page from hard drive into a frame in memory.
Swap: exchange a resident page for another page from storage.
Resident page: a page currently in memory.
Resident set: set of all resident pages.
Page table: an array that holds pointers mapping from page number to physical address of the frame where page resides.
Page replacement policy: a decision policy for choosing what page to swap.
Page table: holds pointers to frames. Allocated for all pages in a program's memory space. Each entry also maintains several flags. **Presence bit**: set if page is resident. **Use bit**: set every time an address on the page has been fetched, re-set if page not used for a fixed number of cycles. **Modified bit**: set if a memory address on the page has been written to. If the modified bit isn't set, we don't need to write it onto the disk. If modified is set and the page on the frame needs to be swapped, it should be written onto the disk. When memory is full (no more pages can be loaded onto the frames), the OS starts swapping.
Address translation: to access address a where each page is $P = 2^n$ bytes long, calculate the page number $N = a \operatorname{div} P$ (higher n order bits of a), fetch the frame offset $F = P_t[N]$, and calculate the offset $O = a \bmod P$. The physical address of a is at $F + O$ and the virtual address of a is at $N + O$.
Page table storage: multiple page tables need to be maintained since multiple virtual spaces are running simultaneously. Need dynamic memory allocation since memory space of the program needs to be able to grow. Page table sizes can grow quickly. So usually stored in memory `OS - page tables - frame storage`.
Soft/hardware synchronization: MMU attempts to load address a on page P. MMU checks page table $P_t[P]$ and checks if presence bit is set. If not, MMU raises page fault (interrupt to OS), OS handles page fault by loading P from disk and updating P_t, if no space left in memory, OS swaps a page P_r with a use bit set to 0, if modified bit is set, page needs to be saved to disk, otherwise discarded. If presence bit is set, MMU looks up address and updates use bit and modified bit.

Translation lookaside buffer (TLB)

MMU needs to query page table P_t from memory, so lookup into page table, get F, get offset and lookup into memory, but each address lookup costs 2 memory lookups so TLB is a sort of cache that keeps records of $F = P_t[N]$. If page N is looked up a second time, return a local copy of F. Supports parallel lookup through TLB and request to memory. If page table exists on MMU, you don't need a TLB since it doesn't lookup twice.

I/O devices

An external circuit connected to the CPU via digital signals. Primary function is data transfer. Follows the fetch/store data paradigm.

Data transfer

Performance: measured by latency (travel time of a data item) and throughput = n (from n-bit parallel or 1 for serial) * latency (in Hz).
Parallel interface: multiple data lines (may have more for control and clock). The number of lines = interface width (usually word size of architecture). Parallel interface transfers multiple bits simultaneously. Has a lower latency and potentially higher throughput but can have issues with interference requiring data items be spread apart.
Serial interface: single data line (may have more for control and clock). Transfers data in series, one bit at a time. Has a higher latency, but possible higher throughput.
Synchronization: to disambiguate between bits, a clock signal is required. Can be explicitly transmitted on a dedicated line. E.g. interpret the bit at every rising edge (but depends on protocol). Interfaces can be clockless through an implied clock where sender and receiver need to agree on a frequency (e.g. expecting certain bits per second like 9600 bps or 9.6 kHz).

Communication channel directionality

Single direction: data only flows from A to B, never B to A.
Half-duplex: allows either A to B or B to A, but not both (e.g. single wire to transfer data).
Full-duplex: allows A to B and B to A (both send and receive).

Multiplexing

Allows I/O devices to transfer larger data items over a narrow channel. Breaks data into chunks and sends through a multiplexor, which gets reassembled by the demultiplexor in the receiver.

I/O bus

A digital communication mechanism that allows two or more I/O devices to exchange data. **Open**: everyone can use/modify. **Proprietary**: if you want to make a device that uses this bus, you have to pay a license to the company that owns the bus line.
Every device connected to the bus has an *interface controller* that implements bus access protocols.
Different lines from the device to the bus connection.
Control lines: for devices to take control of the bus for transfer requests, setting signals and interrupts.
Address lines: transmit the address in a fetch/store operation.
Data lines: transmit data.
When data is wider than interface width, use data multiplexing, which joins address and data lines. Requires implementing a protocol that allows transferring address and data sequentially over shared lines. USes available lines more efficiently. Worst case: no gain. Best case: more lines for data.

Bus address space

All devices connected to the bus see the data/address/control bits sent. Only devices with matching address respond. Each device on the bus have a unique set of addresses assigned. Each address corresponds to a particular device function. Each socket on the bus has a range of unique addresses pre-assigned.

Unified address space

Device addresses are in the same space as memory addresses. Fetch/store for devices is the same as fetch/store for memory. MMU manages address allocation and raises faults when accessing a hole (not available memory) in address space.

I/O bridge

A modern architecture has one primary bus and several auxiliary buses. An I/O bridge is a device that connects two buses; provides address translation in the unified address space (addresses on the main bus can be kept constant and mapped differently on auxiliary buses); and relays data. Typically has two interface controllers, one for each bus.

Programmed I/O

CPU takes control of all low-level operations on devices. Devices are exposed as addresses and CPU directly performs fetch/store operations on them.
Pros: devices can be very simple, fixed logic circuits.
Cons: synchronization issues since CPU runs at a high clock rate, but device operations are much slower so the CPU has to wait; only single devices can be handled at a time; devices can block program execution until their operations are completed. CPU has to periodically check if operations are complete through *polling*.
Control and status registers: information exchange through special registers on the device. The CPU reads status registers and sets control registers.

Interrupt-driven I/O

Allow devices to work asynchronously and interrupt the processor when their operations are done. Requires devices to have their own processor and raise interrupts. The bus needs support for interrupt signals. Processor requires context switching (switching between running programs and interrupts). The OS needs to support interrupts.
Interrupt vector: entry in a table which contains a pointer to a function which should be executed when a particular interrupt happens. Every time a device triggers an interrupt signal, the OS needs to respond.
If interrupted, CPU branches to interrupt handler, Once interrupt is handler, clear signals and restore the state. When interrupt handler is interrupted, interrupts are queued. Not usually a problem devices are slow and interrupt handler is run on the CPU.
At boot time, the OS fills table with interrupt handlers per device. Device IDs can be decided either by plug-in slot or set by BIOS. Hot-plug devices like USBs trigger a new device interrupt and the handler for that interrupt allocates a device id and interrupt handler for the new device. Handling devices are usually through device drivers.

Device driver

Code that provides an upper-half (provides OS interface to user-space programs [system calls], which takes instructions and puts it in a queue [shared variables] for the device to do) and a lower-half (runs asynchronously from the upper-half; runs at a different frequency and takes care of the queue). Communication between application programs and device hardware through shared variables, buffers, and mutually exclusive execution (mutex: cannot be run at the same time by two different actors; makes sure execution happens in the order specified)

System calls

Provide access to devices, but this access is raw. Library functions provide wrappers (high-level). High-level access typically provide buffers to optimize how you do system calls since they're expensive due to having to take control of shared variables.

Direct memory access

Minimize the number of context switches by letting devices fetch/store from memory directly. Instead of the processor feeding data one by one to a device, put the data in a buffer and only interrupt when the buffer is full. Possible to chain buffers for better performance. Can also chain buffers and operations.

Buffering

System calls are expensive since they require synchronization.
Buffering calls: collect several calls and perform them simultaneously, making overhead constant.
On average, expect that a single call takes M cycles for operations and N cycles for overhead. With K calls, it costs $K \times (M + N)$, but if we buffer with a buffer of size K, then the calls cost $K \times M + N$. Sometimes the buffer needs to be flushed before it's full to avoid losing bytes.