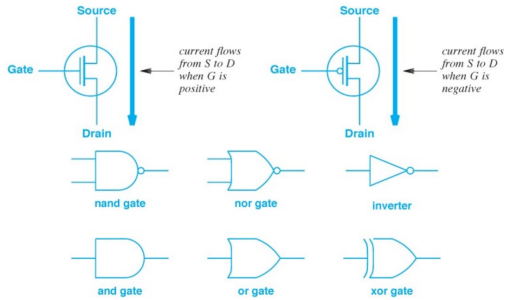
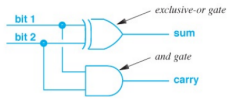
COMPSCI 2GA3 - Geon

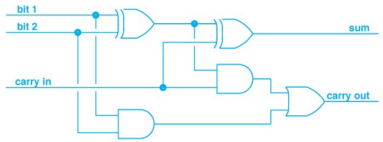
Digital logic



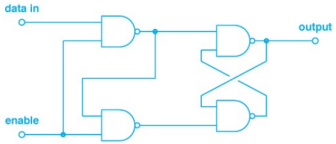
Half adder: takes two input bits, outputs sum (xor) and carry (and).



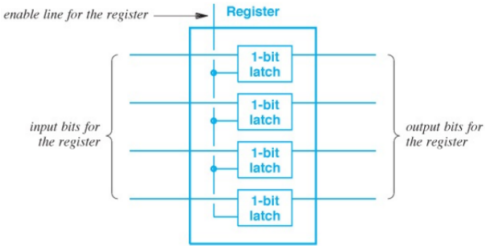
Full adder: takes two input bits and a carry, outputs sum and carry. Built from two half adders and an or gate.



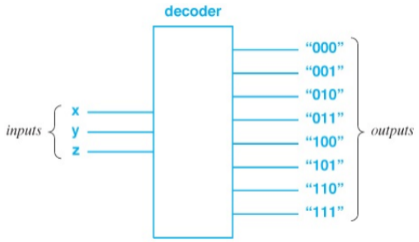
Latch: uses previous output until the enable line is enabled. The output remains the same (even when the enable line is on) until propagation delay is over.



Propagation delay: the delay between an input change an an output change.
Register: stores bits through a series of latches. Different types: general-purpose, floating point, instruction pointer, comparison operation. Use to load data from memory to register, perform ALU operation, and store result from register to memory.



Binary counter: counts in binary. Takes in 1 input, and with n outputs, it can count from 0 to $2^n - 1$. When the input *goes from 0 to 1*, the counter increments by 1. It can have more inputs for *reset* and *pause*. It can have more outputs for *overflow*.
Clock: controls a logic circuit, allowing components to operate without requiring input to change. Oscillates at a regular rate between 1 and 0. *Clocked* components can be synchronized and work as a function of its inputs *and time*.
Decoder: takes in a n-bit binary value and enables one of 2^n outputs.



Demultiplexor: a decoder that takes an extra input which gets passed to the selected output.

Representations

Bit: 0 or 1.
Byte: group of bits (usually 4).
Word: group of bytes (usually 2).
Big endian: MSB to LSB (i.e. how we usually read numbers).
Little endian: LSB to MSB.
Integers:

1. **Unsigned**: only positive integers from 0 to $2^k - 1$.
2. **Sign-magnitude**: MSB is sign, rest is magnitude. From 0 to $2^{k-1} - 1$ and from -0 to $-2^{k-1} - 1$.
3. **One's complement**: MSB is sign, rest is magnitude. Flipped for negative. From 0 to $2^{k-1} - 1$ and from -0 to $-2^{k-1} - 1$.
4. **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:

1. **IEEE**: stored as `sign-exponent-mantissa` (32-bit: 1-8-23, 64-bit: 1-11-52):

$$(-1)^s \cdot (1 + m) \cdot 2^{e-b}$$

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,

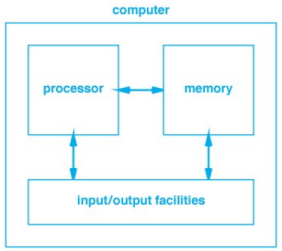
$$e' = e + b$$

where e is the unbiased exponent. The mantissa is *normalized* and is assumed to start with 1.

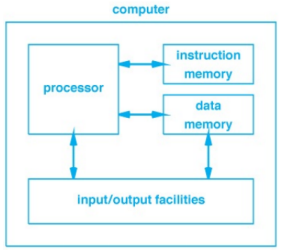
2. **Binary coded decimal**: encode every decimal digit in a byte (4 bits). For fractions, either include an explicit decimal point in the string or specify its location. For signs, a `0xD` is put at the end to indicate it's negative.

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.



Processors: a digital device that can perform multi-step computation.
Types:

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

Categories:

1. **Co-processors**: dedicated function; fixed/selectable logic.
2. **Microcontrollers**: programmable logic, direct hardware control.
3. **Embedded systems**: real-time OS, dedicated hardware.

4. **General purpose**: more functionality than embedded systems.

Units:

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

Fetch-execute cycle

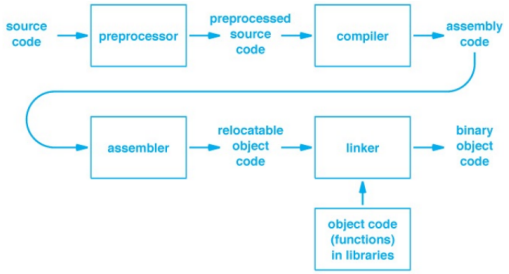
The *instruction pointer* `ip` automatically moves through the program in memory. The cycle *doesn't proceed at a fixed rate* since execution speed depends on the instruction. The cycle *never ends* (unless the system powers down) so there always needs to be a program executing (either the same program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

```
ip = start of program
repeat forever
    instruction = fetch(memory[ip++])
    execute instruction
```

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.



Instructions

Usually encoded as `opcode-operands-offset`.
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.).

```
ip = start of program
repeat forever
    instruction = fetch(memory[ip])
    tmp = ip + 1
    execute instruction
    ip = tmp
```

Absolute branching: `tmp = absolute address`
Relative branching: `tmp = ip + offset`
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.
Subroutine: another word for branching.
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*.

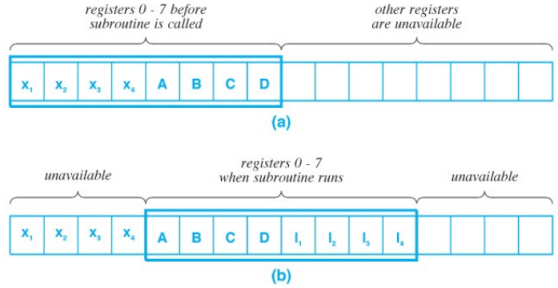


Figure 5.8 Illustration of a register window (a) before a subroutine call, and (b) during the call. Values *A*, *B*, *C*, and *D* correspond to arguments that are passed.

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*.
Throughput: the amount of work done per unit time.
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.