Computer Systems Architecture I

CM10194

Content
History of computing (overview).
Architectures - von Neumann, Non-von Neumann, SIMD to MIMD.
Principles of digital computer operation.
Integer numbers: representations.
Floating point numbers: representations, precision and accuracy.
Introduction to digital logic, simplification of Boolean logic.
Input and output: memory maps, polling, interrupts.
Buses, devices and device addresses.
Aspects of hardware: such as Arduino, tablets and phones.
Introduction to C and developing with SDKs.
Basic assembler concepts: machine codes, instruction execution, addressing modes, CISC v RISC, register use, subroutine calls and the stack.

Ronaldo Butrus

Table of Contents

This document was created with reference to lecture slides by Fabio Nemetz.
All content is taken directly from these slides and the author cannot guarantee the validity of such content.
Completed 26 Dec 2022.

History of Computing

- **1800s: Indian mathematician** Radhanath Sikdar.
- **100BC: Antikyhtera Device.** Calculates astronomical positions.
- **1642: The Pascaline.** Adds and subtracts two numbers, multiplies and divides by repetition.
- **1671: Leibniz' engine.** Multiplying machine.
- **1820s: The Arithmometer by Thomas de Colmar.** First commercially successful calculator.
- **1837: Charles Babbage's Analytical Engine.** First design of a general-purpose programmable computer.
- **1843: Ada Lovelace.** First programmer.
- **1936: Alan Turing.** Designed abstract machine.
- **1949: MONIAC by London LSE.** An analogue computer which used fluidic logic to model economy.
- **1943: The Colossus by Alan Turing.** First programmable digital computer for code breaking, not general purpose.
- **1946: ENIAC by John von Neumann.** First programmable general-purpose computer.
- **1948: Manchester Baby (SSEM).** First programmable general-purpose computer to use Von Neumann architecture. Followed by Manchester Mark 1.
- **1948: EDVAC by University of Pennsylvania** (similar to Manchester Baby).

- Issues with 1st generation digital computers:
  - vacuum tube technology generated a lot of heat
  - consumed a lot of electricity
  - unreliable
  - costly
  - only supported machine language
  - slow I/O devices
  - huge size
  - not portable

- A computer is a data processor. Data can take many kinds, e.g.
  - process control (sensors and controls)
  - data analysis
  - word processing
  - symbolic processing
  - game-playing
  - speech and vision, robotics (sound and images)
  - neural network simulation (cognitive models)

- 2nd generation computers (1950s):
  - transistors to transfer electronic signals across a resistor
    - smaller than vacuum tubes
    - no warm up time
    - consumed less energy
    - generated less heat
    - faster
    - more reliable

- 3rd generation computers (1961):
  - integrated chips (set of electronic circuits on semiconductor material, typically silicon)
  - 10s to 100s of transistors on each chip

- 4th generation computers (1970s):
    - 10 000s of transistors on each chip
    - Large Scale Integration (LSI)
    - Bill Gates and Paul Allen founded Microsoft and wrote a Basic language interpreter for the Altair (minicomputer kit)
    - Steve Wozniak designed and built a better computer and founded Apple with Steve Jobs

- 5th generation computers (1980s):
    - billions of transistors on each chip
    - Very Large Scale Integration (VLSI) / Ultra Large Scale Integration (ULSI)

- The present:
    - main hardware developments are number of transistors per chip, allowing:
        - operating systems
        - networking
        - mobile computing

- **Moore's Law:** computing power (relative to component cost) doubles over a fixed time.
- This was 2 years, and now appears to be 3, but due to the physical properties of silicon there is a finite limit.

Architectures

Abstractions

- Abstraction layers hide the details of a how lower layers work to allow programmers to deal with problems at particular levels.
- Software abstractions:
    - problem
    - algorithm
    - program
- Hardware abstractions (operating system):
    - instruction set architecture (ISA)
    - microarchitecture
    - logic
    - circuit
    - electrons

Von Neumann Architecture

- Most uniprocessor machines use the Von Neumann architecture.
- It uses the common structure of machines, abstracting from their particular details.
- The components are connected by buses (address, control and data buses).

- **Arithmetic and Logic Unit (ALU):**
    - where computational operations are carried out
    - operations require an operation (e.g. add) and an operand (data)

- **Control Unit (CU):**
    - determines operation to be performed
    - selects operands and makes them available
    - supplies correct control data (operation and operands) to ALU
    - determines next operation to be performed
      (thereby renders the processor automatic)

- Data and instructions:
    - stored together in a single memory unit
    - stored and retrieved by the CU
    - if data store and CU work fast enough the ALU can run continuously
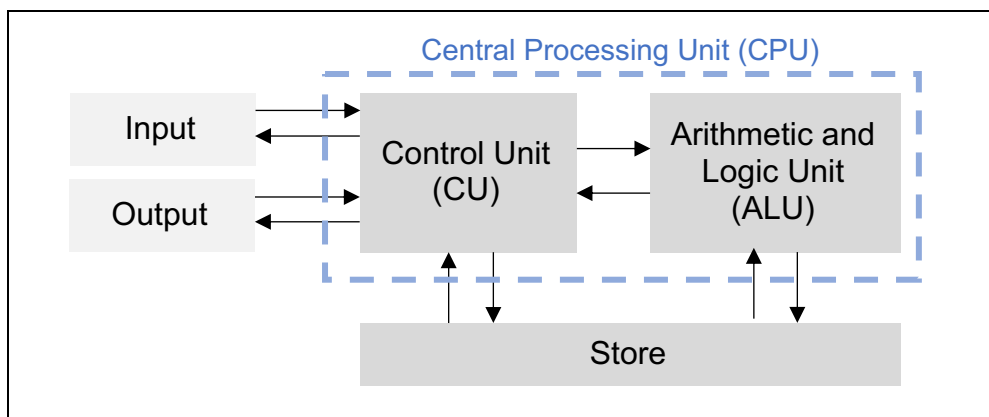
- Von Neumann bottleneck
  - the speed at which data and instructions can be retrieved from memory becomes a limit on the speed at which the CPU can operate
  - because program memory and data memory cannot be accessed at the same time

- Solutions to the Von Neumann bottleneck:
  - alternative architectures (Harvard, distributed storage)
  - caching
  - internal registers

- Caching:
  - **cache:** a data store, duplicating some of main memory, but capable of more rapid access (making frequently used data instantly available to the CPU)
  - **Level 1:** closest to processor, fastest, usually smallest
  - **Level 2:** off-chip, fast, larger

- Internal registers:
  - **register:** a rapid access data store on the CPU to store values used in ongoing computations
  - fastest memory used in a computer
  - modern programming languages do not have direct access to registers (except C)

Harvard Architecture

- Separate stores for data and instructions.
- CPU can access instructions and data simultaneously.
- Harvard Mark 1 was the first computer to use the Harvard architecture, developed by IBM.

- Harvard architecture useful in:
  - special-purpose devices such as microcontrollers and signal processors with instructions stored in ROM
  - sophisticated processors to exploit parallel fetching of data and instructions
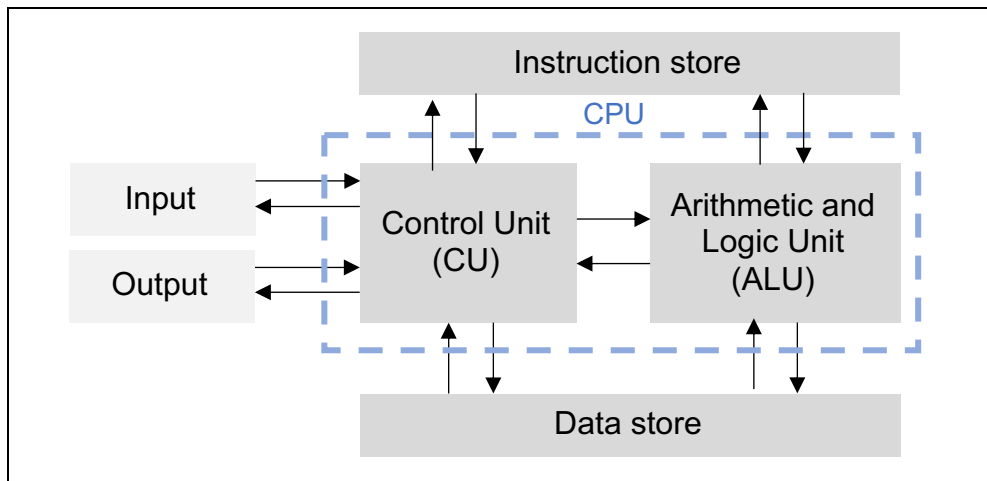
- Modified Harvard architectures:
  - providing a data pathway between the instruction and data stores, so only one loading mechanism from the store to the CPU is required
  - separate caching for code and data but a single store to allow both Harvard and Von Neumann mode

- Harvard architecture has better performance; Von Neumann architecture is more flexible.

Parallel Architectures

- **Parallelism:** carrying out multiple operations simultaneously.
- **Multiprocessor:** computers with multiple ALUs.
- Each ALU may have its own:
    - control unit (task level)
    - data storage (data level)

- Control architectures for multiprocessors:
    - **single instruction stream:** single control unit issues same instruction to each ALU
    - **multiple instruction streams:** each ALU controlled by a separate control unit
- Data architectures for multiprocessors:
    - **single data stream:** all ALUs operate on same data stream
    - **multiple data streams:** each ALU operates on a different data stream (or a vector/array of data)

- **Flynn's Taxonomy:**

DATA LEVEL TASK LEVEL	Single Data Stream	Multiple Data Streams
Single Instruction Streams	SISD	SIMD
Multiple Instruction Streams	MISD	MIMD

- **SISD** (Single Instruction Single Data):
    - uniprocessor architecture (Von Neumann and Harvard)
- **SIMD** (Single Instruction Multiple Data):
    - involves application of basic operation to large dataset (vectors/matrices)
    - used in:
        - supercomputer modelling of physical systems (e.g. weather)
        - signal processing (sound/vision)
        - graphics/audio processing (e.g. adjusting contrast in digital image)
        - video games (e.g. Xbox 360 – Xenon CPU)
- **MISD** (Multiple Instruction Single Data):
    - used in fault-tolerant/safety-critical computing (e.g. Space Shuttle)
- **MIMD** (Multiple Instruction Multiple Data):
    - modern multi-core processors

- Memory architectures:
    - **shared memory:**
        - each processer has access to the same memory space
        - efficient
        - leads to memory-to-CPU bottlenecks
        - cache coherence problems
    - **distributed memory:**
        - each processor has its own data store (memory)
        - easy to scale up
        - inefficient, indirect communication between processors (message passing)
        - distributing and then reassembling data is a significant task

- Compromise solutions:
    - **virtual shared memory:** distributed memory which 'seems' shared to CPUs
    - **non-uniform memory access (NUMA):** shared memory which has parts that are faster to access for each CPU

- Multi-core processors:
  – processors with multiple cores on same chip
  – used for multitasking (running different applications in parallel)
  – software design must take advantage of the parallelism potential

- **Amdahl's Law** (limits of parallelisation):
  – for a given computational task
    suppose a proportion $0 \leq p \leq 1$ can be parallelised
  – a proportion $s = 1 - p$ is sequential
  – the maximum speedup obtainable using N parallel processors is given by:
    $$\frac{1}{(1 - p) + \frac{p}{N}}$$
  – the absolute limit (with infinitely many processors) can be found using $\frac{1}{1-p}$.

Principles of digital computer operation

Analogue vs Digital

- Parameters can be represented by continuously variable analogues such as:
    - rotation of a shaft
    - electrical current or voltage
- Analogues are:
    - precise
    - inaccurate (cannot be exactly measured)

- Parameters can be represented by discretised representations (digital).
- Digital methods are:
    - imprecise
    - accurate

Numeration systems

- Computers are required to represent different forms of data:
    - unsigned integers
    - signed integers
    - alphanumeric characters
    - strings
    - real numbers
    - program instructions
    - addresses of memory locations and external devices

- Electronic computers (except ENIAC) are based on two-state devices (transistors and logic gates).
- The binary system can be used to distinguish between on and off.

- All numeration systems are representational.
- Numeration systems are a form of coding, which should be devised such that:
    - few symbols (easy to remember)
    - unambiguous
    - economical
    - useful quantitative measure
    - easy to manipulate (e.g. to perform arithmetic, unlike Roman system)
- Therefore the elements of a numeration system are that:
    - base determines number of digit symbols needed
    - place values increase from right to left in successive powers of the base
    - addition is used to make up a number from digits
    - multiplication is used to make up a number from its place
    - there is an agreed starting point (the 'unit' place) denoted by a point

- Decimal system:
    - base 10
    - symbols 0-9
    - coincides with number of fingers
    - uses positional notation

- Other numeration systems:
    - binary (base 2)
    - octal (base 8)
    - hexadecimal (base 16)

- Numbers should be followed by a subscript of their base, e.g. $F6_{16}$ indicates hex.

Decimal	Binary	Octal	Hex	Decimal	Binary	Octal	Hex
1	1	1	1	9	1001	11	9
2	10	2	2	10	1010	12	A
3	11	3	3	11	1011	13	B
4	100	4	4	12	1100	14	C
5	101	5	5	13	1101	15	D
6	110	6	6	14	1110	16	E
7	111	7	7	15	1111	17	F
8	1000	10	8	16	10000	20	10

Data storage

- Binary:
  - a bit is a binary digit
  - can be encoded by two-state devices
  - a cell is a string of bits
  - a byte is a cell of 8 bits
  - a word is often a cell of 16 bits
  - a long word is often a cell of 32/64 bits

- Cells:
  - a store (memory) consists of a collection of cells
  - a cell is the smallest addressable unit of a store
  - the states of the bits of a cell make up its contents
  - each cell has a unique address numbered 0 to m–1 for a store with m cells
  - once a cell has been located at an address, its contents can be examined/changed
  - contents of a cell can be instructions, data or even addresses, and it is impossible to tell by looking

- Types of memory access:
  - **Sequential Access Memory:** time to access cell with given address depends on address of cell just accessed
  - **Random Access Memory:** time to access cell independent of its address
    - **Dynamic RAM (DRAM):** used for main memory, continuously refreshed
    - **Static RAM (SRAM):** used for cache, can retain data bits with power supply

- Address space:
  - a range of addresses associated with a physical or virtual location (cells, I/O ports, network IP addresses, etc.)
  - same address may refer to different locations in different spaces
  - length of address determines size of space (e.g. 32-bit processor can address 2^{32} memory addresses)
  - processor, operating system and application need to match number of bits per address

- Representing text characters:
  - each character is assigned a unique bit pattern
  - a string of characters can be stored in a succession of bytes
  - American Standard Code for Information Interchange (ASCII) is a 7-bit representation
  - UNICODE is a 32-bit interpretation which can encode emojis, symbols and foreign alphabets

Integer numbers

Unsigned integers

E.g. 10111001_2 represents $128 + 32 + 16 + 8 + 1 = 185$:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	1	0	0	1

- A byte can represent 2^8 different numbers (0 to 255_{10}).
- When adding binary numbers using column addition, $1 + 1 = 10$ so the leftmost bit should be 'carried over' to the next addition.

- Other representations of unsigned integers:
  - **Binary-coded decimal (BCD):** 4 bits (used in early decimal computers, e.g. ENIAC)
  - **Binary-coded sexagesimal:** 6 bits (time and angles)
  - **Gray codes:** reflected binary code, where successive values differ by a single digit (digital TV)

Signed integers

- **Sign and Magnitude:**
  - can represent integers from $-(2^{n-1} - 1)$ to $(2^{n-1} - 1)$ in a cell of n bits
  - first bit represents sign (0 for positive, 1 for negative)
  - remaining $n-1$ bits represent magnitude
  - **advantages:** easy to read, symmetric about zero
  - **disadvantages:** two representations of zero, only 255 values represented in a cell, arithmetic becomes complicated

- **1s Complement:**
  - leading 0 for positive integers
  - swap 1s and 0s for negative integers
  - **advantages:** simplified arithmetic, symmetric about zero
  - **disadvantages:** two representations of zero

- **2s Complement:**
  - leading 0 for positive integers
  - swap 1s and 0s for negative integers and add 1
  - **advantages:** one representation of zero, simplified arithmetic
  - **disadvantages:** not symmetric about zero

Integers in programming languages

- Integers in C:
    - unsigned int n
    - signed int n
    - int n   (same as signed)

    - char is 8 bits
    - short is often 16 bits
    - int is often 32 bits
    - long is often 64 bits

- Integers in Arduino UNO:
    - int is 16 bits (signed used 2s complement)
    - word is unsigned int
    - long is 32 bits
    - short is 16 bits
    - byte is unsigned 8-bit   (B is binary formatter, e.g. byte b = B1010)

- Integers in Java: (all signed)
    - byte is 8 bits
    - short is 16 bits
    - int is 32 bits
    - long int is often 64 bits

Floating point numbers

- Place values can be extended to the right of the point.
- E.g. 179.32 can be represented as:
  $(1 \times 10^2) + (7 \times 10^1) + (9 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$

- When storing floating point real numbers there are floating point errors because there are more real numbers between any two points than there are integers.

- Scientific notation: mantissa x base^{exponent}
  - **mantissa (or significand):** normalised as $1 \leq |mantissa| < base$
  - **base (or radix):** in binary this is 2
  - **exponent:** can be positive or negative
  - given a fixed mantissa size we can only approximate numbers with a greater number of significant digits
  - **advantages:**
    - simple to generate, understand and manipulate
    - can compactly represent numbers of varying magnitudes
    - accurate to precision of representation
  - **disadvantages:**
    - gives approximation of numbers which could be represented exactly
    - rounding errors may increase beyond control in complex calculations

- IEEE 754 Single Precision (32 bits):
  - **sign:** 1 bit (0 for positive, 1 for negative)
  - **exponent:** 8 bits (0-255) (subtract bias of 2^7-1 to 'split' into positive and negative)
  - **mantissa:** 23 bits (24 bit precision as significand of normalised binary floating point number is always 1)

  - converting 13.1875 to single precision floating point number:
    - **sign is 0**
    - $13 = 1101_2$
    - keep multiplying 0.1875 by 2 until zero is reached or repetition
      0.1875 x 2 = 0.375
      0.375 x 2 = 0.75
      0.75 x 2 = 1.5
      0.5 x 2 = 1.0
    - **mantissa is 1101.0011**
    - normalised mantissa is 1.1010011 x 2^3
    - **ignore preceding 1, hence use 1010011**
    - **exponent is 3 + 127 = 130 = 10000010**
    - **result is 0100000101010011000000000000000**

  - arithmetic:
    - addition is done by de-normalising, adding mantissas and renormalising
    - multiplication is done by multiplying mantissas, adding exponents and renormalising

  - special cases:
    - an exponent of 0 represents a zero-valued floating point number
    - an exponent of 255 represents infinity with mantissa of zero
    - exponent values of 1 to 254 represent binary exponents 2^{-126} to 2^{127}
    - very large numbers (e.g. near x=0 on a graph of y=1/x) cause rounding errors
    - double precision uses 11 exponent bits and 52 significand bits

Introduction to digital logic, simplification of Boolean logic

Digital logic

- **Combinational** logic circuits are:
    - **static:** do not change
    - **deterministic:** for each combination of inputs there is a single output
    - they are used to implement 'timeless' operations such as **logic** and **arithmetic**
- **Sequential** logic circuits are:
    - **dynamic:** change over time
    - **non-deterministic:** for each combination of inputs there may be multiple possible outputs
    - they incorporate **feedback** from their outputs
    - they are used to implement 'stateful' operations such as **control circuits** and **memory cells**
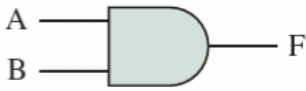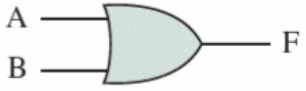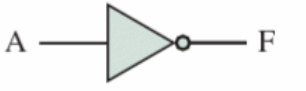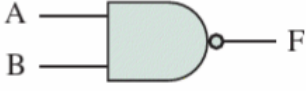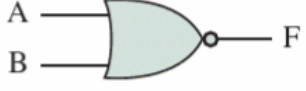
- Boolean algebra is based upon the three fundamental logic operations: AND, OR and NOT.
- It is a mathematical description of systems which can have one of two states:
    - TRUE (voltage / 1) or
    - FALSE (no voltage / 0).   (as observed by Claude Shannon in 1937)

- Logical AND:
    - TRUE only if both A and B are true. Otherwise FALSE.
    - **Associative:** (A.B).C = A.(B.C)
    - **Commutative:** A.B = B.A
    - **Identity is 1:** A.1 = A
    - **Absorptive:** A.A = A
    - **Annihilator is 0:** A.0 = 0
    - **Distributive:** A.(B+C) = A.B + A.C

- Logical OR:
    - TRUE if either A or B is true. Otherwise FALSE.
    - **Associative:** (A+B)+C = A+(B+C)
    - **Commutative:** A+B = B+A
    - **Identity is 0:** A+0 = A
    - **Absorptive:** A+A = A
    - **Annihilator is 1:** A.1 = 1
    - **Distributive:** A + (B.C) = (A+B).(A+C)

- Logical NOT:
    - TRUE if A is false. Otherwise FALSE.
    - **Involutivity:** A'' = A (double negation)

- De Morgan's Laws:
    (A.B)' = A' + B'
    (A+B)' = A'.B'

Name	Graphical Symbol	Algebraic Function	Truth Table
AND	A ─┐ [gate] ── F B ─┘	$F = A \bullet B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR	A ─┐ [gate] ── F B ─┘	$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT	A ──[gate]o── F	$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND	A ─┐ [gate]o── F B ─┘	$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR	A ─┐ [gate]o── F B ─┘	$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR	A ─┐ [gate] ── F B ─┘	$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

- Logic gates can be connected to build circuits taking multiple inputs and performing complex logical operations.
- Boolean variables can be combined to make more complex formulae.
- Circuits can be used to implement Boolean logic, and Boolean logic can be used to represent and reason about circuits.
- Boolean variables are inputs, formula values are outputs.

- Determining if two formulae are equivalent by:
  - **equational reasoning:** using laws or axioms to get from one formula to another through algebraic manipulation
  - **semantic reasoning:** verifying that both formulae produce the same output value for each possible combination of inputs (e.g. using a truth table)

- Proving A.(A+B) = A by algebra:
  A.(A + B)
  = A.A + B.B
  = A + A.B
  = A.(1 + B)
  = A.1
  = A

- Proving A.(A+B) = A by truth table:

A	B	A+B	A.(A+B)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

- A **standard sum of products** of a Boolean formula is a series of terms joined by OR operations, where each term contains a combination of all variables joined by AND operations (i.e. a disjunction of conjunctions).
- Every Boolean expression over n variables is equivalent to a standard sum of products.

- E.g. finding the standard sum of products for A+B.C by algebra:

  A + B.C
  = A.(B + B') + B.C
  = A.B + A.B' + B.C
  = A.B.(C + C') + A.B' + B.C
  = A.B.C + A.B.C' + A.B'.(C + C') + B.C
  = A.B.C + A.B.C' + A.B'.C + A.B'.C' + B.C.(A+A')
  = ***A.B.C*** + A.B.C' + A.B'.C + A.B'.C' + ***A.B.C*** + A'.B.C
  = A.B.C + A.B.C' + A.B'.C + A.B'.C' + A'.B.C

- E.g. finding the standard sum of products for A+B.C by truth table:

A	B	C	B.C	A+(B.C)	Conjunction for this row
0	0	0	0	0	
0	0	1	0	0	
0	1	0	0	0	
0	1	1	1	1	A'.B.C
1	0	0	0	1	A.B'.C'
1	0	1	0	1	A.B'.C
1	1	0	0	1	A.B.C'
1	1	1	1	1	A.B.C

Then summing to get A'.B.C + A.B'.C' + A.B'.C + A.B.C' + A.B.C

Simplification of Boolean logic

- To implement a Boolean formula as a circuit, we need it in its simplest form:
  - convert it to the standard sum of products, then
  - find the simplest equivalent sum of products expression using Karnaugh maps.

- Karnaugh maps:
  - a method to simplify Boolean expressions
  - reduces the need for extensive calculations by using pattern-recognition
  - works well for up to 4 variables

- Any function of two Boolean variables, A and B, can be plotted on a map with four regions:

  E.g. F = AB' + A'B   (i.e. A XOR B as a standard sum of products)

	AB		
00	01	11	10
	1		1

- Any function of three Boolean variables, A, B and C, can be plotted on a map with eight regions:

  E.g. F = A'BC' + A'BC + ABC'

		BC			
		00	01	11	10
A	0			1	1
	1				1

  C can be eliminated:
  A'BC' + A'BC becomes A'B

- Any function of four Boolean variables, A, B, C and D, can be plotted on a map with sixteen regions:

  E.g. F = A'B'CD + AB'C'D + ABC'D'

		CD			
		00	01	11	10
	00			1	
	01				
AB	11	1			
	10		1		

- Once the map is created, note the arrangement of 1s:
  - **any two adjacent 1s:** product terms differ in only one variable, one can be eliminated (including around edges)
  - **inputs we don't need to consider can be treated as a 1 or 0:** this allows for simpler expressions

- Seven segment display:
  - Driven by **binary coded decimal (BCD).**
  - 4 bits to represent one decimal digit.
  - Combinational logic turns on or off each segment to create the digit display.
  - Each of the 7 outputs (a, b, c, d, e, f, g) will need a Karnaugh map using binary digits A, B, C and D from the binary representation of the digit to display.
  - E.g. for segment a: (on for 0, 2, 3, 5, 6, 7, 8, 9)

		CD			
		00	01	11	10
	00	1	0	1	1
AB	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

BD

		CD			
		00	01	11	10
	00	1	0	1	1
AB	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

A

		CD			
		00	01	11	10
	00	1	0	1	1
AB	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

C

		CD			
		00	01	11	10
	00	1	0	1	1
AB	01	0	1	1	1
	11	x	x	x	x
	10	1	1	x	x

B'D'

  - This leads to A + C + BD + B'D'.

Implementing arithmetic

- Combinational Boolean logic circuits can be used to efficiently implement:
  - **arithmetic operations:** addition, subtraction and multiplication
  - **binary representations of:** unsigned integers, signed integers and floating point numbers

- 1-bit half adder:
  - inputs A and B are addition operands
  - outputs are S (sum) and C (carry)
  - S = A XOR B
  - C = A AND B
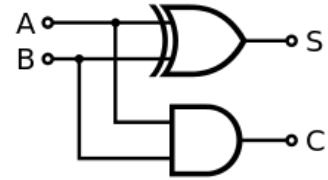
- 4-bit adder (four half-bit adders connected):
  - each full adder has three inputs (A, B and C) and two outputs (S and C)
  - S = A XOR B XOR C
  - C = (A AND B) OR (C AND (A XOR B))

  - $S = A \oplus B \oplus C$
  - $C = A.B + C.(A \oplus B)$
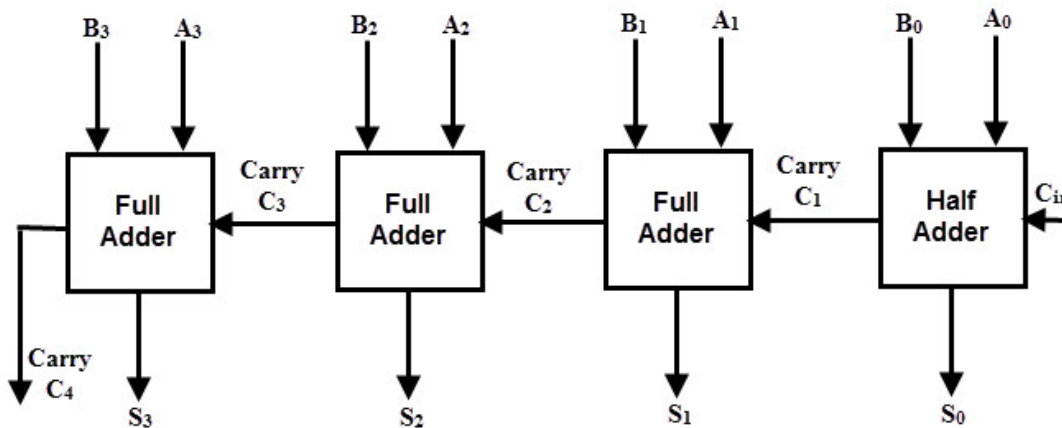
- These additions work for signed integers (2s complement) given no overflow.

- Detecting overflow:
  - **guard bit** is added to most significant end of representation, copied from the most significant bit
  - arithmetic is internally performed on an (n+1) bit operand (for an n-bit operand)
  - if the guard bit in the (n+1) bit result is different from the most significant bit of the n-bit representation, then there is an **arithmetic overflow**

- Multiplication (positive integers) is performed either by:
  - **adding repeatedly**
  - **long multiplication**

- Binary multiplication by powers of two:
  - shift bit pattern to the left

- Binary long multiplication:
    - split one operand (e.g. 1011) into multiple operands with single 1s (e.g. 1000, 10, 1)
    - multiply other operand by each of these
    - sum the results
    - multiplying 2 n-bit integers can produce a result of up to 2n-bits in length

- Shift operations:
    - **logical shifts** on unsigned binary**:**
        - **n-bit left-shift** corresponds to multiplying by 2^n
        - **n-bit right-shift** corresponds to dividing by 2^n with truncation
        - bits moved out are lost and zeroes fill vacated positions
    - **arithmetic shifts:**
        - **left:** same as logical shift
        - **right:** logical shift but copies of the sign bit are propagated
    - **circular shifts:**
        - bits moved out of one end are moved in at the opposite end of the register

- Bitwise logical operations:
    - typically parallel operations, thus fast
    - e.g.   A =   0110 1010
              B =   1101 0000

          A AND B: 0100 0000
          A OR B: 1111 1010
          A XOR B: 1011 1010
          NOT A: 1001 0101

Sequential logic

- Combinational logic system outputs can be used as inputs to a gate 'further down' a system.
- A sequential logic system has a **state**.

- **SR latch** (set-reset):
    - two NOR gates connected in a feedback loop
    - when S = R = 0 (and Q = 0)
    - system is consistent and stable as long as S and R aren't both 1
    - when S = R = 1, system is inconsistent (Q = Q' = 0)
    - when S changes to 1, Q = 1
    - when S changes to 0, Q remains 1
    - when R changes to 1, Q = 0
    - when R changes to 0, Q remains 0
    - as there is a delay before a logic gate responds to an input, two separate gates cannot switch simultaneously

input	output
00	1
01	0
10	0
11	0
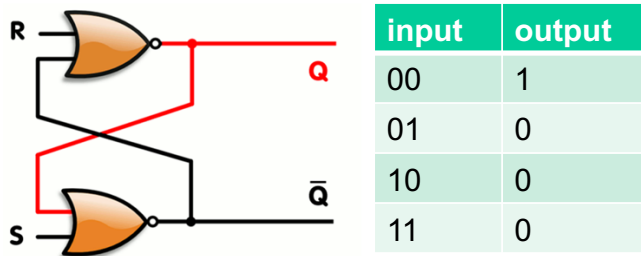
(a) Characteristic Table

Current Inputs	Current State	Next State
SR	Q_n	Q_{n+1}
00	0	0
00	1	1
01	0	0
01	1	0
10	0	1
10	1	1
11	0	—
11	1	—

(b) Simplified Characteristic Table

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	—

Figure 20.23   NOR S–R Latch timing Diagram
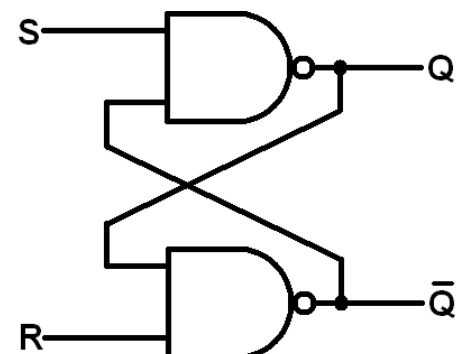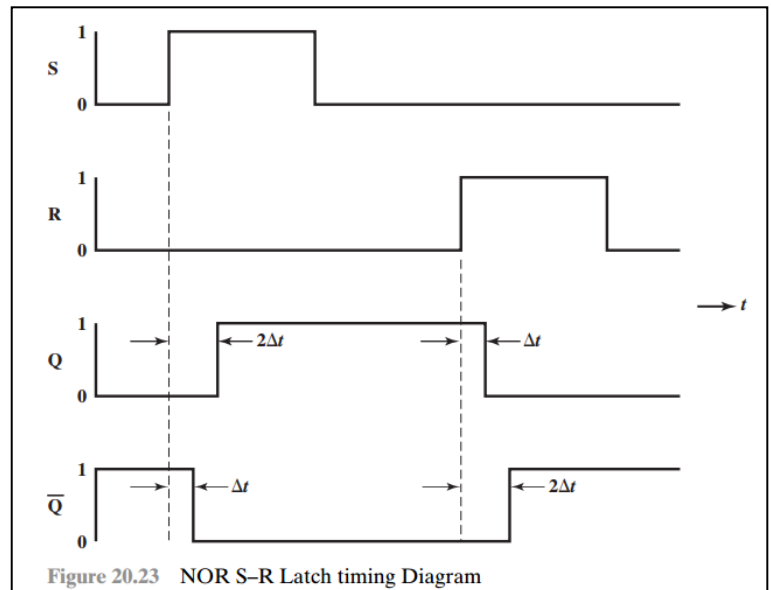
- **Inverted SR latch:**
    - sets when S = 0, resets when R = 0
    - implemented using NAND gates
    - cannot have S = R = 0

22

- Problems with simple SR latches:
  - **asynchronous transitions:** use clock to enable/disable transitions (flip-flop)
  - **non-deterministic transitions:** avoid disallowed states by adding control gates to inputs

- **S-R flip-flop:**
  - adds clock signal to latch
  - R and S inputs are passed to NOR gates only when clock is high
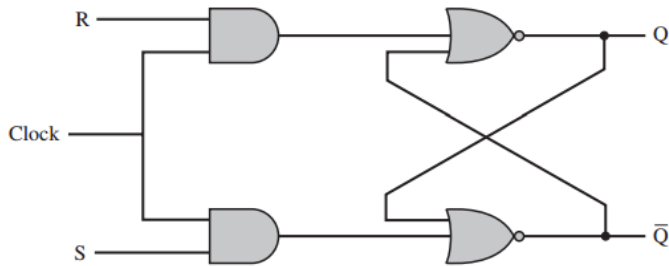
- **D-type flip-flop:**
  - one input instead of two
  - inverter provides 'second' input
  - eliminates S = R = 1
  - referred to as **data** flip-flop because it is effectively storage for one bit of data
  - referred to as **delay** flip-flop as it delays an input for one clock pulse
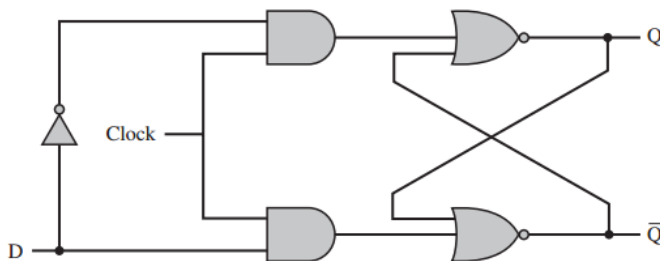
D	Q_{n+1}
0	0
1	1

- **J-K flip-flop:**
  - restricted combination (S = R = 1) used to toggle output state
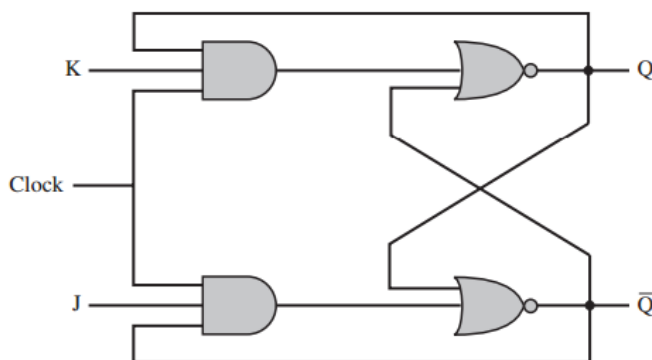
Name	Truth table			Name	Truth table		
S-R	S	R	Q_{n+1}	J-K	J	K	Q_{n+1}
	0	0	Q_n		0	0	Q_n
	0	1	0		0	1	0
	1	0	1		1	0	1
	1	1	-		1	1	$(Q_n)'$
				D	D	Q_{n+1}	
					0	0	
					1	1	

- A **register** is a digital circuit used in a CPU to store one or more bits of data:
  - **parallel register:**
    - consists of 1-bit memories that can be read or written to simultaneously
    - clock used to synchronise simultaneous writing
  - **shift register:**
    - accepts or transfers data serially
    - e.g.   a 5-bit shift register constructed from D flip-flops:
      data is shifted right by one position and rightmost bit transferred out on each clock pulse
    - uses serial I/O devices, ALU logical shift and rotate operations
  - **counters:**
    - a register that can be incremented by 1
    - resets to 0 once maximum is reached
    - register of n flip-flops can count up to $2^n - 1$
    - can be **synchronous** or **asynchronous**
    - **ripple counter** (asynchronous)**:**
      - change that occurs to increment the counter starts at one end and ripples through to the other end
      - delay is proportional to length of counter
      - e.g. 4-bit up counter using edge-triggered J-K flip-flops

Q_3	Q_2	Q_1	Q_o
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
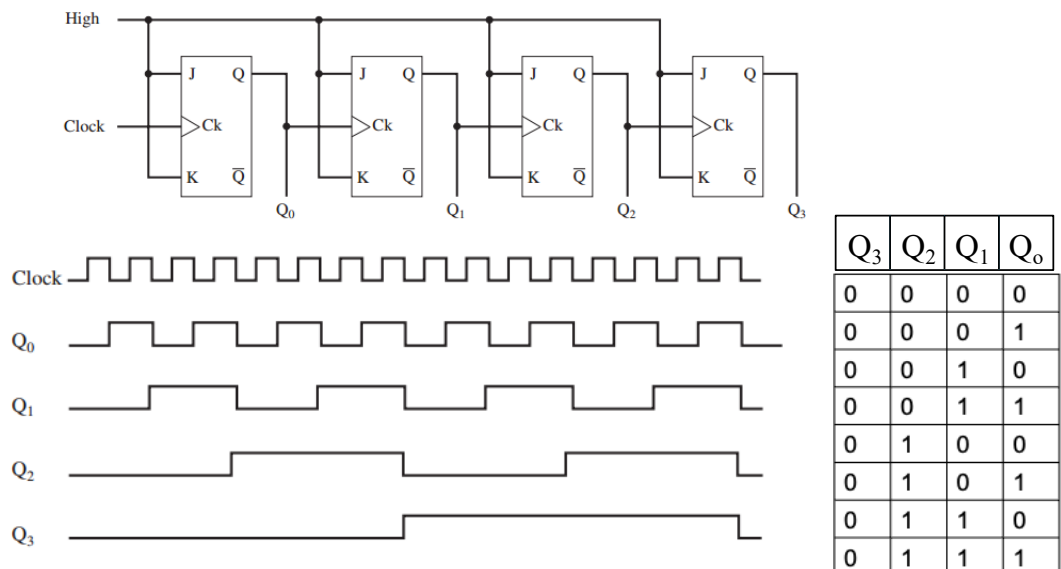
    - **synchronous counter:**
      - uses Boolean expressions derived from Karnaugh maps to change all flip-flops at the same time

Input and output

- Input/Output is any transfer of information beyond the CPU and main memory.
- I/O devices:
  - deliver different amounts of data
  - have different speeds
  - have different formats

- External devices provide a means of exchanging data between the computer and its external environment.
- External devices can be:
  - **human readable:** suitable for communicating with the user, e.g. printers, video display terminals (VDTs)
  - **machine readable:** suitable for communicating with equipment, e.g. sensors, actuators, disk and tape systems
  - **communication:** suitable for communicating with remote devices, e.g. terminal, machine readable device or another computer

- An I/O device can be attached to a computer by a link to an I/O module which exchanged control, status and data signals between the I/O module and its peripheral.
- An I/O module has the following major functions:
  - **control and timing:** coordinates flow of traffic between internal and external devices
  - **processor communication:** command decoding, data status reporting, addressing
  - **device communication:** commands, status information and data
  - **data buffering:** buffering operations to balance device and memory speeds
  - **error detection:** detects and reports transmission errors

- Techniques for I/O operations:
  - **programmed I/O:**
    - data transfer initiated by instructions stored in computer program
    - uses input instructions to transfer data from device to CPU and store instructions to transfer data from CPU to store
    - CPU continuously monitors I/O module to know if it is ready for transfer
    - usually involves electro-mechanical process, which is much slower than electronic transfer of data within the CPU or between the CPU and store
  - **interrupt-driven I/O:**
    - processor issues I/O command, continues to execute other instructions
    - interrupted by I/O module when I/O module is ready for transfer
  - **direct memory access (DMA):**
    - I/O module and main memory exchange data directly without direct processor involvement
    - CPU starts the transfer but can perform other tasks while the transfer occurs
    - I/O module interrupts CPU when transfer is completed
    - useful when CPU cannot keep up with rate of data transfer between media

- Controlling I/O devices:
  - a stored instruction is fetched to the IR and decoded to be an I/O operation
  - the CU sends an instruction to the I/O CU
  - the I/O CU sends signals to the device to set it into action
  - data is transferred to the device (write) or from the device (read) using a data register in the ALU

- Controlling multiple I/O devices:
  - each I/O instruction must be coded as a bit pattern to the I/O CU, containing:
    - I/O **operation code**
    - **direction** of transfer
    - **address** of device (so data can be retrieved from common I/O data bus)
  - address information is placed on a common I/O address bus
  - advantages of shared I/O bus:
    - simplicity
    - flexibility
  - disadvantages of shared I/O bus:
    - cost of building shared bus
    - contention (conflict) for shared resources (data) on bus

- **Memory-mapped I/O**:
  - The CPU is connected to the store by a single bus.
  - Number of store locations that can be accessed is determined by the number of bits in the address lines of the bus, e.g. 24 lines means 2^{24} locations (address space is number of locations).
  - Address space contains store locations and device addresses.
  - I/O data transmitted on data lines, addresses along address lines and read/write information on control lines.
  - I/O devices can be operated by loading and storing I/O data in the same way as loading and storing data in the store unit.
  - Advantages:
    - simplified I/O control
    - frees up CPU space for other uses
  - Disadvantages:
    - uses address space
    - I/O devices are in contention with the store for the same bus
    - programs may be harder to understand

- Coordinating I/O devices and CPU:
  - **Device flags:**
    - each device has a one-bit register acting as the device status flag
    - when an I/O transfer is started, the device stratus flag is unset (0) to indicate the device is busy
    - when the transfer is complete the device sets its device status flag (1) to indicate is idle and ready to perform another I/O transfer
    - device operation can be controlled by testing the device status flag before starting a transfer
  - **Busy waiting:**
    - if device status flag is idle then reset it and transfer data, otherwise repeat
  - **Polling:**
    - if device status flag is idle then reset it and transfer data, otherwise perform another task for a fixed time and then repeat
    - disadvantages: unresponsiveness, multiple devices have different speeds, complexity

- Interrupts:
  - **Interrupt system:**
    - a device that is ready to handle an I/O transfer sends an interrupt request (IRQ) to the CPU
    - execution of the current instruction is completed
    - the device sending the interrupt request is identified
    - the contents of the PC are saved
    - context is saved (including all processor registers)
    - the contents of the PC are replaced with the address of the procedure for handling or servicing the interrupting I/O device
    - the I/O is executed
    - context is restored (including PC and other registers)
    - the next background program instruction is executed
  - **Device priorities:**
    - each I/O device is given a priority
    - when a device sends an interrupt request to the CPU, the priority is compared with the current operating priority
    - an interrupt request with a higher priority is accepted first
    - a stack is used to process nested interrupts
  - Advantages:
    - less wasteful of CPU time
    - more responsive
    - good for controlling multiple devices (priorities)
  - Disadvantages:
    - need special hardware for fast interrupt system (programmable interrupt controller (PIC) combines interrupt sources into one or more CPU lines)
    - context-switching is costly in terms of CPU time

- Evolution of the I/O function:
  - The CPU directly controls a peripheral device.
  - The CPU uses programmed I/O without interrupts.
  - The CPU uses programmed I/O with interrupts.
  - The I/O module is given direct memory access.
  - The I/O module is given a specialised instruction set for I/O.
  - The I/O module is given its own local memory, limited CPU involvement.

Buses, devices and device addresses

[Not covered]

Aspects of hardware

- Arduino UNO:
    - ATmega 328 microcontroller
    - AVR CPU
    - 32 general purpose registers

- Microcontrollers are designed for embedded applications.
- An embedded processor typically has a well-defined task that it performs reliably and efficiently.
- Many embedded applications therefore use the Harvard architecture.

[Lab work]

Introduction to C and developing with SDKs

Symbol	Operator
&	bitwise AND
|	bitwise inclusive OR
^	bitwise exclusive OR
<<	left shift
>>	right shift
~	bitwise NOT (one's complement) (unary)

[Lab work]

Symbol	Operator
&	bitwise AND

Basic assembler concepts

Instructions

- An instruction requires the:
  - **operation** to be executed
  - **operand(s)** to perform the operation on
    - typically two (addition)
    - sometimes one (negation)
    - occasionally none (e.g. HALT)

- Expressive power is the breadth of ideas that can be represented and communicated.
- Instructions are compactly encoded to maximise expressive power.

- Instructions usually occupy more than one byte of storage.
- The number of bytes varies.
- Typically:
  - operation specification occupies 2 bytes (65 536 different operations)
  - operand occupies 2 or 4 bytes (usually an address)

Registers

- Registers consist of arrays of bits.
- The ALU contains registers for holding data while it is processed (data registers).
- The CU contains registers for storing program information.

- **Zero address codes:** instructions with no arguments:
  - e.g. INC Rn increments whatever is in register n (13 bits for operation, 3 for data register)
  - e.g. HALT stops the computer

- **One address codes:** if a data register is always used as one operand of a two-operand instruction, then this operand does not need to be explicitly specified in the instruction.
  - e.g. addition becomes a sequence of instructions (LOAD, ADD then STORE) without specifying register to load into

- **One and a half address codes:** if there is more than one data register, only part of the instruction word is required to code it:
  - e.g. 8 registers requires 3 bits to specify which register to use
  - if using 2 bytes for operation and 2 bytes for operand, we can use 13 bits (operation), 3 bits (data register) and 16 bits (operand)

- **Two address codes:** instructions with two operands:
  - e.g. ADD D1, D2

Sequences of instructions

- A **program** is an ordered sequence of instructions.
- Successive instructions can be stored in successively higher-addressed store locations to be executed by stepping through the store addresses in order.
- Registers in the control unit are used to keep track of program information, and these cannot be directly accessed by the programmer.

- Registers in the control unit (CU):
  - **Program Counter (PC):** holds address of next instruction to be executed
  - **Instruction Register (IR):** holds copy of current instruction while it is executing
  - **Memory Address (MA):** holds address being accessed
  - **Memory Buffer (MB):** holds contents of location in MA while it is being written to or read from the store

Fetch-execute cycle

- To carry out an instruction:
  - **fetch** instruction from store to CU:
    - contents of the PC placed in the MA
    - contents of cell at address stored in MA placed in the MB
    - contents of the MB placed in the IR
    - increment the PC
  - **interpret** instruction
  - **fetch** operands from store to ALU
    - this can involve fetching an operand from another address using the same registers MA and MB
  - **execute** instruction using control signals from the CU

Motorola MC68000 architecture

- The allocation of bits in an instruction word does not change the nature of the execution of an instruction.
- A word (natural chunk of data) is 16 bits for this processor.
- E.g. the addition operation adds the 16-bit contents of a specified location to one of the 8 data registers:
  - ADD $1202, D5 is represented by 4 bytes   ($ used for hexadecimal)
  - operation specification occupies 4 bits
  - register specification occupies 3 bits
  - opmode specification occupies 3 bits
  - effective address specification occupies 6 bits
  - operand address specification occupies 16 bits

- Store and I/O devices are connected to the CPU by a single bus containing lines for addresses, data and control signals.
- Registers in the 68000 CPU:
  - one 16-bit IR
  - four inaccessible temporary registers T1, T2, T3 and T4
  - eight data registers D0 – D7

- Executing ADD $1202, D5:
  - **fetch:**
    - contents of PC placed on address lines of bus
    - read signal placed on control lines of bus
    - begin read from store, placing data on data lines of bus
    - place data from bus in IR
    - decode instruction in IR
    - increment PC (by 2 as word is 16 bits)
  - **execute:**
    - repeat same fetch for operand
    - place data from bus in T1
    - place data from T1 on address lines of bus
    - read signal placed on control lines of bus
    - begin read from store, placing data on data lines of bus
    - add contents of data lines to lowest 16 bits in D5

- Operand addressing modes:
  - **immediate:** given value is used
  - **direct:** value at given address is used
  - **indirect:** value at address stored at the given address is used
  - other addressing modes: indirect with displacement, indirect with post-increment, indirect with pre-decrement, indirect with index, PC with placement, PC with index…

- Common instructions:
  - LOAD: loads data into register
  - ADD: adds two operands and stores in register
  - STORE or MOVE: stores value from register in memory
  - COMPARE: compares two values
  - JUMP: jump to another address in memory (branch)
  - JUMP IF condition: JUMP if condition is true (conditional branch)

Instruction sets

CISC	RISC
more powerful instructions	simpler instructions
more complicated to decode	simple decoding
arithmetic instructions also perform memory accesses (register memory architecture)	arithmetic instructions only use registers (load/store architecture)
less registers	more registers
popular in laptops, PCs	popular in mobile devices

Complex Instruction Set Computer (CISC)

- CISC involves richer instruction sets with more complex instructions, designed to:
    - simplify compilers
    - improve performance
- Smaller programs:
    - take up less memory
    - improve performance (fewer instructions so fewer fetches and more instructions in cache)
- However CISC requires more transistors for decoding instructions, so less general-purpose registers can be fitted to the processor.

Reduced Instruction Set Computer (RISC)

- RISC involves smaller instruction sets with simpler instructions.
    - each instruction takes one clock cycle to execute, so instructions executed uniformly
    - more memory is needed to store assembly level instructions
    - compiler needs to perform more work to convert high-level code into RISC instructions

Control instructions

- Control instructions can alter the contents of the PC in order to:
    - be able to execute each instruction more than once
    - implement decision-making
    - execute modularised code
- The main control instructions are branching and procedure calls.

- Branching:
    - **conditional branch:** branch is made only if a condition is met
    - **unconditional branch:** branch is always made
    - branches can be either forward (higher address) or backward (lower address)
    - one-bit flags (condition codes) are used to denote the state of aspects of the ALU
      e.g. carry flag (CF) set on bit carry
      e.g. overflow flag (OF) set if result is too large positive or too large negative
    - branching allows programming control structures such as **decisions** and **loops**.

- Subroutines:
  - a block of instructions for performing a task is placed in memory and the subroutine is called by branching to the starting location of the block
  - after the block of instructions is completed, the subroutine returns to the point at which it was called
  - a stack (last in first out – LIFO) is used to implement nested subroutine calls
  - the stack stores the return address of the subroutines being executed
  - stacks start at high memory addresses and lower
  - typically the stack stores a stack frame for each call containing the return address, argument variables, local variables and saved copies of modified registers that need to be restored.

- Implementing stacks:
  - a block of successive memory locations store the contents of the stack
  - the stack pointer (SP) register stores the address at the top of the stack
  - **push** decrements SP and stores the added value at the new address in SP
  - **pop** dereferences the address stored in SP and increments SP
  - to **call** a subroutine the contents of the PC are pushed onto the stack and the PC is reset to the starting address of the subroutine
  - to **return** the PC is reset to the value popped from the stack

Instruction level parallelism

- **Latency** is the amount of time taken between the start of an action and its completion.
- **Throughput** is the total number of actions per unit time.

- **Pipelining** is a technique where multiple instructions are simultaneously executed.
- Considering a complete cycle Fetch, Decode, Fetch (operands), Execute, Store:

F	D	F	E	S		
	F	D	F	E	S	
		F	D	F	E	S
			F	D	F	E	S

- Problems with pipelining:
  - **branching:** avoid branches, predicting decoding, multiple pipelines
  - **instruction conflicts:** dependencies break the pipeline
  - **load delays:** fetch is slower than decode or execute, so pipeline stalls
  - **unequal stage times:** ideally stages should take roughly the same amount of time

- **Superscalar** is a technique where instructions are executed in parallel if they are recognised by the CPU as not data dependent.
- Redundant functional units on the processor such as the ALU, bit shifter or multiplier can be used as an execution resource.

- Architectures can be pipelined, superscalar, both or neither.
- Pipelining and superscalar improve throughput.