A picture containing table

Description automatically generatedDiagram

Description automatically generatedVonNeumann Machine ( John von Neumann , 1945)

Table

Description automatically generatedTimeline

Description automatically generated**Memory**: Storage Area (Registers) - All data and instructions are stored registers in before they are processed. **ALU**: Performs arithmetic operations (add, subtract, etc.), Logical operations (AND, OR, NOT, etc.). **Control unit:** Controls operation of Memory Accumulator, ALU Input/output devices. Performs timing and control signals. **Bus**: Data communication medium between components. **Truth table** – all possible outcomes of the proposition. **Boolean algebra**

A picture containing text, document, receipt

Description automatically generatedA picture containing text, shoji

Description automatically generated

**Negation**

(A')' = A – Involution law

Text

Description automatically generatedA • A' = 0

Diagram, schematic

Description automatically generatedA + A' = 1

**Associative**

(A • B) • C = A • (B • C)

(A + B) + C = A + (B + C)

Simplification rules with 1 and 0

A • 0 = 0

Text

Description automatically generatedA • 1 = A

A + 0 = A

A + 1 = 1

(nand=not and;   
nor = not or) **LOGIC GATES:**

A picture containing text, clock, watch

Description automatically generated**NAND** – universal gate, complete (any Boolean function can be generated by a combination of these functions), can be constructed from transistors.

A picture containing text

Description automatically generatedOther gates made of NAND:

**NOT** A picture containing icon

Description automatically generated (idempotent law: A • A = A therefore (A • A)' = A ').

**AND** A picture containing mirror

Description automatically generated (Create an AND gate using the Involution law: (A')' = A)

Diagram

Description automatically generated**OR** Diagram

Description automatically generated (Apply de Morgan's theorem: (A' • B')' = A + B)

Diagram

Description automatically generated**NAND** Diagram, schematic

Description automatically generated I = (B • C)’ X = (A • I)' = (A • (B • C)')'   
Apply de Morgan’s law, we get X = A' + ((B • C)')' = A' + (B • C)

A picture containing shape

Description automatically generatedA picture containing shape

Description automatically generatedA picture containing calendar

Description automatically generatedTable

Description automatically generated

**Basic circuits**: **Half Adder** – performs addition of numbers, used in ALU and processor. Binary output of addition is output in Sum and Carry versions, which are equivalend to truth tables of XOR and AND gates. But they cannot be chained together to add multi-bit numbers, cannot accept a carry bit from the previous stage. Can be cascaded to produce adders of any number of bits by daisy chaining the carry of one output to the input of the next.

Diagram

Description automatically generated**Full Adder** - A picture containing text

Description automatically generated

Diagram, schematic

Description automatically generatedA picture containing text, clock, gauge

Description automatically generated**Ripple-Carry adder**  - consists of several full adders connected in a series so that the carry must propagate through every full adder before the addition is complete. Requires the least amount of hardware of all adders but the slowest.

Table

Description automatically generated**Carry-Lookahead Adder** is faster but more complex. All gates can be built out of NAND and NOR gates!

**Memory: Latches** – building blocks to sequential circuits, can be built from gates. 1 latch can remember 1 bit of information. SR-latch (S=Set, R=Reset). SR-latch truth table:

Diagram

Description automatically generatedInitially the latch S=0, R=0. Then value of S or R can be set to 1 and reset it to 0 to save the state of the latch. The latch will remember the signal. Latches are asynchronous – the output changes very soon after the input changes. **Flip-Flop** is a synchronous version of a latch, where sequential circuits change simultaneously to the rhythm of a global clock signal. **D latch (data latch)** – constructed using the inverted S input as the R input signal. Allows for a single input – no race condition as input is inverted. **Memory**: **Static RAM (SRAM)** – bit-cell is a latch, fast and not very dense, primarily used in cache, consumes less power. **Dynamic RAM (DRAM)** – bit-cell is a transistor and capacitor (which leaks information), so storage must be periodically refreshed. Primarily used in main memory. Cheaper than SRAM.

**3 Types of memory: CPU** – Registers (Fixed), capacity: <2KB, speed: <1nanosecs, volatility: contents lost. **Main memory** – RAM (Expandable), capacity: 512 MB - 16+ GB, speed: 10-100nanosecs, volatility: contents lost. **Storage Device** – Hard Disk (Expandable), capacity: 250 GB - 8+ TB, speed: 5-10 millisecs, volatility: contents not lost. **Diagram

Description automatically generatedIntegrated Circuits (chips)** – all are made of logic gates. MSI Chip –   
ß **Multiplexer** / MUX (multiple-input, single-output switch). Sel selects which of inputs is mapped to the output. In general, a multiplexer has inputs and n control lines and 1 output. **2to1 multiplexer à**

**ß Demultiplexer** / DEMUX (single-input, multiple-output switch). Usually used in conjunction with MUX.

**Decoder** – (multiple-input, multiple-output logic circuit). Converts coded inputs into coded outputs. Binary decoder has n inputs and outputs. ß Used in e.g. mem address decoding. Only one output is 1 – one selected by the n-bit binary input number. The rest are zero. Diagram

Description automatically generated

**Comparator** – compare two numbers, ex. 1-bit comparison, which gate to use? à Comparator returns 1 is the two n-bit inputs A and B are equal, 0 otherwise.

**Arithmetic Logic Unit (ALU)** – performs arithmetic and logical operations. Building block of a CPU. Can link together 1-bit ALUs to form a multi-bit ALU (bit-slice circuits).

Diagram, schematic

Description automatically generated

**Main memory organisation**

Table

Description automatically generatedBit (0 or 1). Memory cell (location) – can store a piece of information, e.g. 1 byte (8 bits). Byte – smallest addressable unit of memory in most computer archs.

Each mem location is W bits long (byte-multiple). Mem size: R\*W bits. Can read/write entire row or one byte at a time. **Memory addressing**: **word addressing** – addresses entire row. **Byte addressing** – every byte in main memory has an address. Two formats – **Big Endian** – stores **Most Significant Byte** first. **Little Endian** – stores **Least Significant Byte** first. **Byte Ordering** – Multibyte Data Items

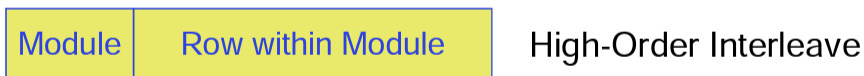
Chart, histogram

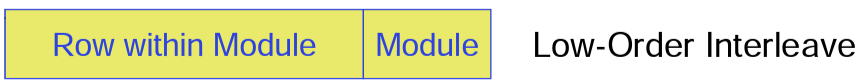
Description automatically generated

KB (kilobyte) = MB (megabyte) = GB (gigabyte) =

TB (terabyte) = PB (petabyte) =

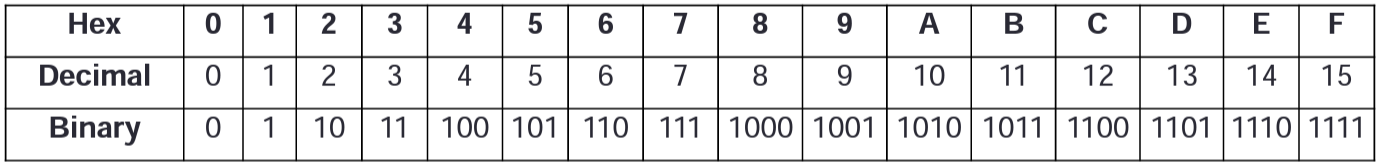
**Memory interleaving** –





*Module* – no of bits needed to select a module, e.g., 2 bits to in a 4-module memory (). *Row within module* – no of bits needed to select a word in a module, e.g., 20 bits in a 1MB module (). **High-order interleave** better is modules are accessed independently by different units for higher performance (parallel operation). **Low-order interleave** better if the CPU can request multiple adjacent memory locations, e.g., accessing elements in an array, or instructions in a program. **Data representation: Binary** (base 2): Divide the number by 2 giving the quotient and the remainder. Repeat step with new quotient until a zero quotient is obtained. Answer - read the remainder column bottom to the top. **Octal** (base 8): Starting from the rightmost end, each group of 3 bits () represents 1 octal digit (called octet). Ex. , or . **Hexadecimal** (base 16): 4 Binary digits represent one hexadecimal digit (bits), starting from the rightmost end, each group of 4 bits repr. 1 hexadec. digit. Ex. , or .

Chart

Description automatically generated with medium confidence

**Representation of natural numbers** – **Sign and Magnitude** – leftmost bit represents the sign of the integer (1 = minus), remaining bits to represent its magnitude. Two representations for zero (.

Range: for n-bits.

Table

Description automatically generated**One’s complement** – negative numbers are the complement of the positive numbers. Less costly to implement (no need for sign comparator).

Range:

**Two’s complement** – negative of an integer is achieved by inverting each of the bits and adding 1 to it. Only one bit pattern for zero (negative).

Range:

Diagram, schematic

Description automatically generated**Calendar

Description automatically generated**Table

Description automatically generated**Excess-n (Bias-n)** – if we want to represent neg numbers but want to keep the same ordering where 000 repr the smallest value and 111 repr the largest v in 3-bits. **Binary Coded Decimal (BCD)** – each decimal digit is repr by a fixed num of bits, usually four or eight. Takes up more space.**ASCII character mapping** – uses 7-bits (128 bit patterns). 26 lowercase and upper letters, 10 digits, 32 punctuation marks and others. **Unicode** – 120000 characters defined and going. **Binary operations.** Subtrahend – what is being subtracted; minuend – subtracted from what. **Binary addition (unsigned)** – Ex. 111011 + 101010 = 1100101 **Binary subtraction (unsigned)** – Ex. 1010101 – 11100 = 0111001 **Binary multiplication (unsigned)** – Ex. 11101 x 111 = 11001011 à Binary arithmetic on signed numbers, using two’s compliment:

Table

Description automatically generatedText

Description automatically generatedTable

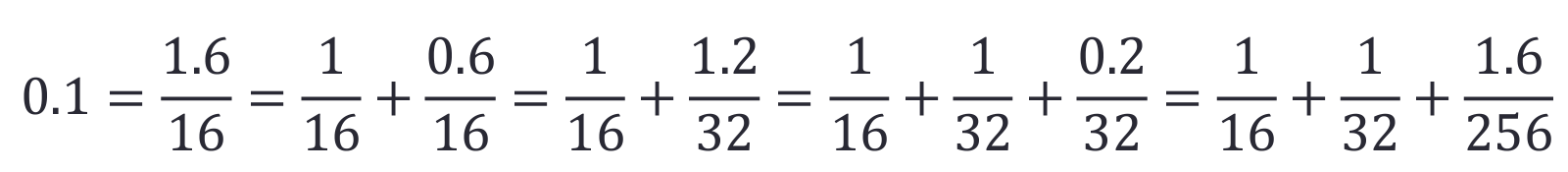
Description automatically generatedAddition and subtraction works without having a separate sign bit. Overflow! – occurs if 2 Two’s complement no’s are added and both have the same sign, and the result has the opposite sign. **Addition** – add the values and discard any carry-out bit.**Subtraction** – negate the subtrahend and add it to the minuend. Any carry-out bit is discarded. Overflow! Occurs iff 2 two’s complement no’s are subtracted and their signs are different, and the result has the same sign as the subtrahend. Multiplication and division cannot be done with two’s complement using the std technique. **Floating point numbers** –  **Binary to decimal: Table

Description automatically generated with medium confidenceText

Description automatically generated**

**Decimal to binary: Text

Description automatically generated with medium confidence**

** Normalised floating-point multiplication**: we multiply the coefficients and add the exponents. Then, normalise the result. To shorten the number, truncation (biased error) or rounding (unbiased error) is performed. **A floating-point addition** - Exponents need to be the same. Otherwise, we need to align them first. To align, choose the number with the smaller exponent and shift its coefficient the corresponding number of digits to the right. **Exponent overflow** occurs when the result is exponent > maximum exponent. To handle overflow, set value as infinity or raise an exception. **IEEE floating point standard – single-precision format (32-bit):** The normal bit(the 1.) is omitted from the significand field ➔a hidden bit. Exponents are stored as excess values. Single precision yields 24 bits (approx. 7 decimal digits of precision). Normalised ranges in decimal appx: . **Double-precision format (64-bit)**: A picture containing chart

Description automatically generated

Table

Description automatically generated Double precision yields 53 bits (approx. 16 decimal digits of precision). Normalised ranges in decimal appx: . **Conversion from decimal to binary IEEE single-precision**: 1) Convert to binary number, eg. 42.6875=10 1010.1011, 2) Normalise, eg. 3) Significand field, eg. 0101 0101 1000 0000 0000 000. 4) Exponent field in excess-127, eg. (5+127)=132: 1000 0100. Result: **0 1000 0100** **0101 0101 1000 0000 0000 000**, HEX: 422A C000. **Conversion from IEEE format**: Ex. BEC0 0000 to decimal. 1) Convert to binary: **1** **0111 1101** **1000 0000 0000 0000 0000 000**. 2) Exponent field: 0111 1101 à 125. 3) True binary exponent (in excess-127): 125-127=-2. 4) Significand field + hidden bit: 1.1000 0000 0000 0000 0000 000. 5) Unsigned value: . 6) Add sign bit: -0.375. When performing arithmetic operations on IEEE numbers, remember to adjust the exponent so that it’s the same! Zero exponent used to represent denormalised numbers or zero! An all one exponent used to represent infinities and NaNs. Special operations: CPU organisation: Fetch-Execute cycle: Fetch the ***Instruction*,** Increment the ***Program Counter*,** Decode the ***Instruction*,** Fetch the ***Operands*,** Perform the ***Operation*,** Store the ***Results*, Repeat** Forever. **Performing multiplication:**

Shape, rectangle

Description automatically generated

A picture containing diagram

Description automatically generated Ex. 

Text, timeline

Description automatically generated**Pentium architecture – Registers & addressing** – **eax, ebx, ecb, edx**, **esi** (source index reg), **edi** (destination index reg), **esp** (stack pointer reg), **ebp** (base pointer reg). In 32-bit arch, these registers take up entire words. In 16-bit arch, the least significant 16-bits of registers have an additional register name that can be used to access those 16 bits. In 8-bit arch, the 2 least significant bytes of reg eax, ebc, ecx, edx also have register names. **eip**- instruction pointer register, holds the address of the next instruction to be executed;. **eflags**- register that holds info about the current state of the CPU, mostly used by OS, but some relevant for conditional branch instructions**: Zero Flag** (bit 6) (if result is zero), **Sign Flag** (bit 7) (set to MS-bit of the result), **Overflow Flag** (Bit 11) (if result too large/too small), **Carry Flag** (bit 0) (if carry or borrow out of MS-bit), **Parity Flag** (bit 2) (if LS-byte of result contains even number of 1-bits). Basic data types: byte (8bits), word (16bits), doubleword (32bits), quadword (64 bits). **Pentium instruction format:** Label: opcode Destination, Source ; OR ; Label: opcode Operand ; OR ; Label: opcode , Where label is an optional user-defined identifier. Intel syntax, **netwide assembler (nasm). Data declaration directives** – assembler commands that allow global variables to be declared, by mapping global data to fixed mem locations and can be accessed by using the name of the variable. **Initialised data** directives db (byte), dw (word), dd (doubleword) e.g., users db 3 ; byte with value 3

total dd 999 ; doubleword with value 999 ; sequence dw 1, 2, 3 ; 3-words with values 1, 2, 3. **Uninitialised data** can be reserved with resb (byte), resw (word), resd (doubleword), e.g., tiny resb 10 ; reserve 10 bytes. **Named constants** can be defined with a directive: equ, e.g., dozen equ 12. **Operands (addressing modes): Register operands** (e.g., eax, edx, eal, esi, ebp) – found in the specified register. A register operand can be in any of the general purpose registers. Some instructions implicitly use operands contained in a pair of registers. For most 2-operand instructions, destination and source operands must be of the same size!; **Immediate operands**, constants (e.g., 23, 67H, ‘R’, ‘ON’) – operand is an immediate value. Not normally applicable for destination operands. If a data variable is used as an immediate operand, then its address is used; **Memory operands** (e.g., [24], [bp], [esi+2]) – specify an address using expressions in the form: [BaseRegister + Scale\*IndexRegister + Displacement]. The size of the operand is usually inferred from the instruction or register operand. If ambiguous, prefix the operand with byte, word, or dword to specify the size. **Displacement** (direct addressing), e.g., [list+22]. **Base** (register indirect) – contents of specified base reg gives address, e.g. mov ax,[bx]. **Base+Displacement** (Register relative) – can be used to access object fields (position of field within object), or array elements, or parameters and local variables. E.g., mov ax,[bx+4]. **Base+Index** (Based index) – can be used to access array elements where start of array is dynamically determined at run-time. E.g., mov ax,[bx+di]. **Base+Index+Displacement** (Base Relative Index) – Can be used to access arrays of objects, arrays within objects, and arrays on the stack. E.g., mov ax,[bp+di+10]. **(Scale\*Index)+Displacement** (Scaled Index) – using constant scaling factor 2, 4, or 8. Supports efficient access to array elements when the element size is 2, 4, or 8 bytes. **Base + (Scale\*Index) + Displacement** – supports efficient access to arrays within objects and on the stack when the element size is 2, 4, or 8 bytes.

**Graphical user interface

Description automatically generated with low confidence**Note: **The src and dest operands CANT both be memory operands.**

Graphical user interface, application

Description automatically generated

A picture containing table

Description automatically generatedTimeline

Description automatically generated