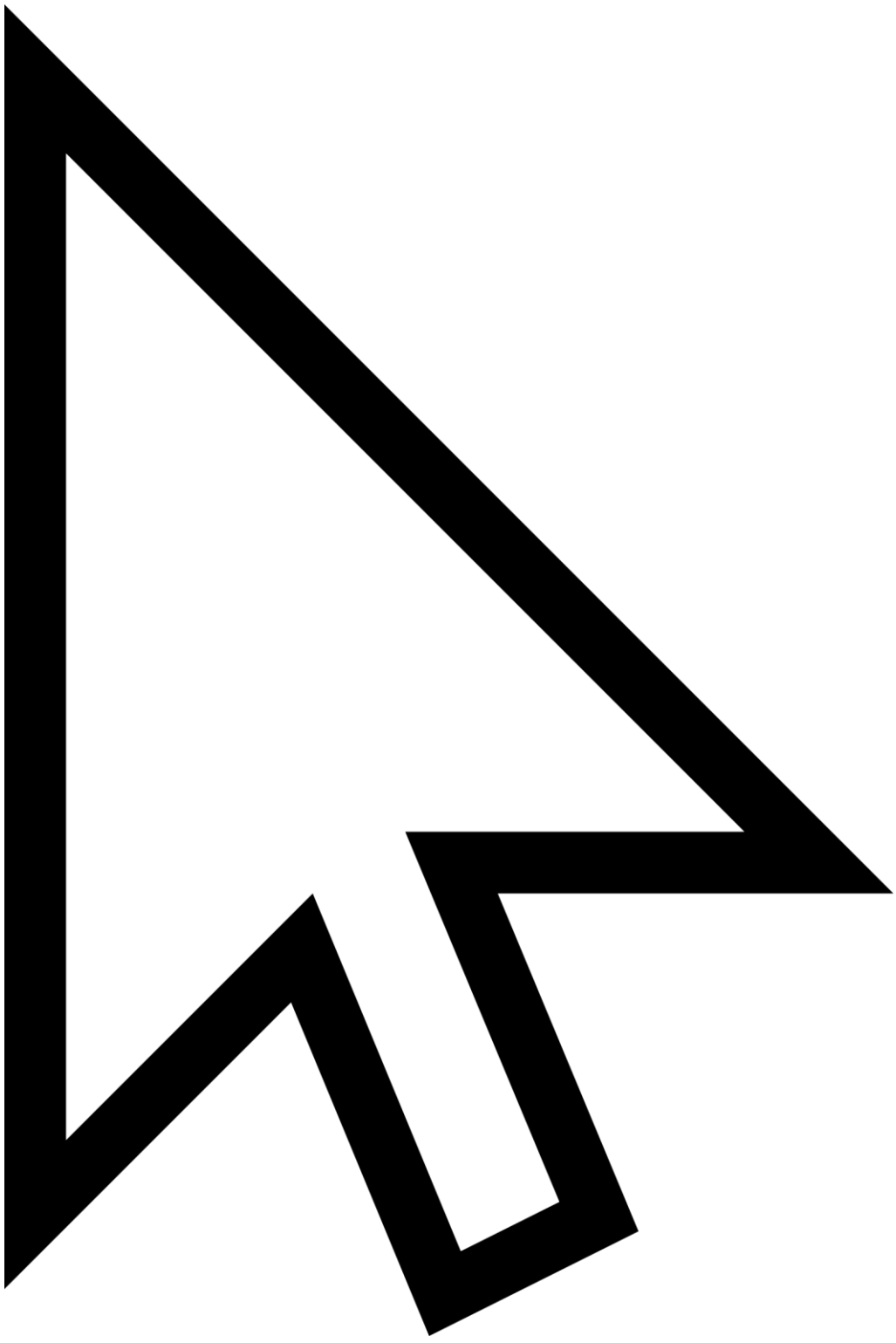


Introduction to Computer Systems

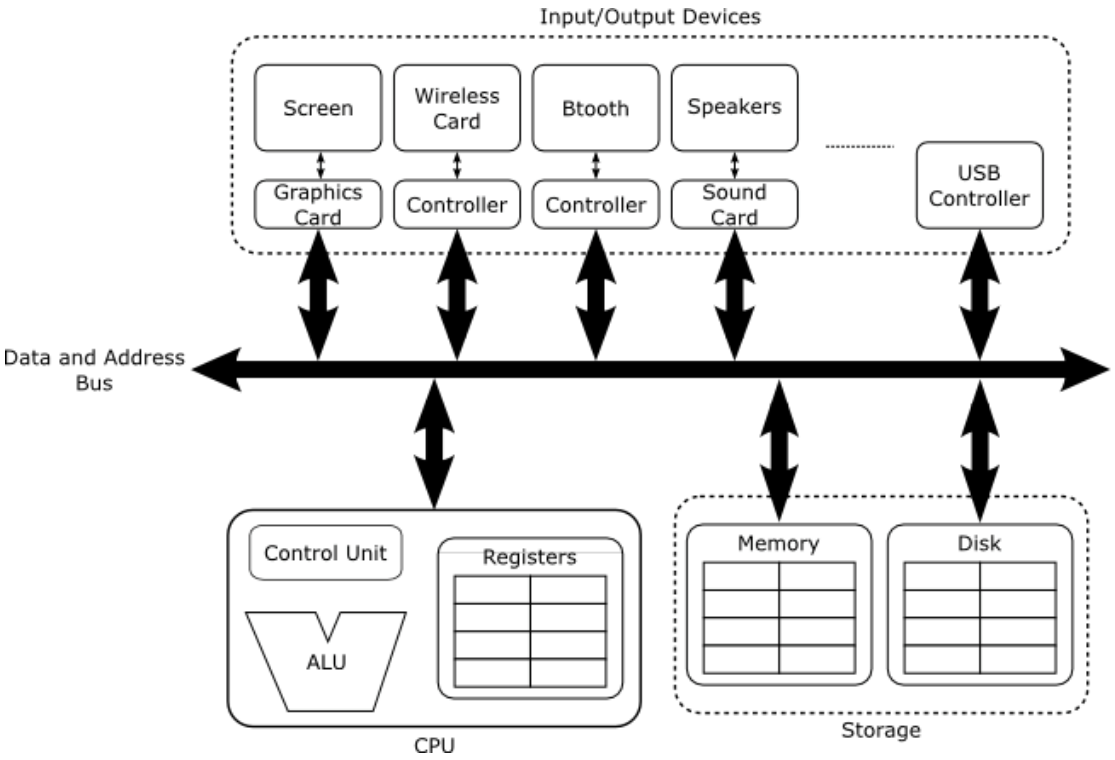
Introduction to Computer Systems Lecture 1:

Computer Systems
Assembly Language

Abstraction:
Abstraction is a process used to simplify a concept by hiding certain aspects to reduce complexity. Different views of the same concept that hide different details can be considered as 'levels of abstraction'. This technique is commonly used to study computer systems.

Computer Systems:
A computer system is analyzed from different levels of abstraction depending on who the user is. A regular user perceives the computer to be a machine that can perform certain tasks through the use of input and output devices. A user with a knowledge of electronics recognizes a computer as a complex system of interconnected circuits that can execute machine instructions to run programs and manipulate data. A programmer perceives a computer as an object that can be given orders through code written in a programming language. From a designer's perspective a computer is a device that directs, controls, and executes machine language instructions through the use of a motherboard.

Structure of a Computer System:
The internal structure of a computer system varies between different computers, especially between different types of computers such as regular PCs and servers. The following is a basic structure of a computer called the 'Von Neumann Architecture'.



Notice how the data and address bus is the medium between the input/output devices and the internal CPU and storage mechanisms.

The Central Processing Unit (CPU) contains different elements such as registers which store data temporarily as it is being processed, an Arithmetic Logic Unit (ALU) which can perform arithmetic functions on the data, and a control unit which processes and executes instructions.

Storage devices are used to store data to be used by the processor. This comes in multiple forms including static RAM, dynamic RAM, hard drives, and more. Note that volatile memory devices lose the information they store when the computer is turned off, while non-volatile memory devices retain the information they store even after powered down.

Input/output devices send and receive data to and from the system. These devices are managed by a circuit called a 'controller' and include the screen, keyboard, mouse, etc.

Programs:
A program is a set of instructions written in a programming language then given to a computer to perform a specific task. Computer systems cannot understand instructions in high-level programming languages like Java or C++. These instructions must first be translated or 'compiled' into low-level machine code to be legible to the computer.

Assembly Language:
Processors are designed to execute a set of instructions called 'machine language'. These instructions are encoded in binary digits, however processors also provide a human-readable format of these instructions called 'assembly language'. This way, rather than writing programs for the processor in binary, we can use the processor's assembly language. Note that assembly languages still require compilation.

High-level programming languages were designed to avoid writing programs in assembly language because it is so complicated, however programming using assembly language assists us in understanding processors, which are used in countless systems, and in turn will help us understand other aspects of computer systems.

Introduction to Computer Systems Lecture 2:

Numbers in Different Bases
Binary, Octal and Hexadecimal Encoding
Encoding Instructions

Numbers in Different Bases:
Natural numbers are usually written in base 10 (decimal), meaning that are written using 10 different digits (0 to 9). Any number written in a base x uses x different digits to write numbers. Also the number x-1 followed by the number 10, and the maximum number possible written with y digits is followed by a 1 with y zeros. Generally, the higher the base, the more compact the number representation. The largest number that can be represented in base b with n digits is $b^n - 1$.

To convert a sequence of digits in another base to base 10, we sum the value of each digit multiplied by the base to the power of the position of the digit.

$1234_7 = (1 \times 7^3) + (2 \times 7^2) + (3 \times 7^1) + (4 \times 7^0) = 466_{10}$

To convert a sequence of digits in base 10 to another base, we divide the sequence by the new base. The remainder is the rightmost digit. If the quotient is less than the new base then it becomes the leftmost digit, otherwise we divide by the new base again and repeat the process.

$47_{10} = (6 \times 7) + 5 = 65_7$

In any base, if we multiply a number by its base, we are simply adding a 0 to the end. Conversely, if we divide a number by its base, we are removing the rightmost digit which is the remainder.

Binary Encoding:
Information that is processed by digital circuits must first be encoded into bits. Bits are a portion of data that exists in one of two states: either a 0 or a 1. Digital circuits are systems that can receive data in bits and perform operations on this data to produce another set of bits. Microprocessors are an example of a complex digital circuit that can execute a previously defined set of instructions that are encoded with bits. This set of instructions is known as 'machine language'.

A single bit can represent only one of two elements: 0 and 1. Two bits can represent four different elements: 00, 01, 10, and 11. In general, n bits can encode 2^n elements. We often have to find the minimum amount of bits n required to encode a certain amount of elements N. To do this, we need to find the smallest value of n that satisfies the inequality $N \leq 2^n$. Alternatively, to encode N elements we need at least $\log_2 N$ bits (rounded up).

In digital systems, natural numbers are encoded into base 2. The methods of conversion to and from base 10 become much simpler when we are only dealing with 0s and 1s. Note that in base 2, if the rightmost bit of a number is 1 then the decimal equivalent is odd.

$1011_2 = 2^3 + 2^1 + 2^0 = 11_{10}$
$11_{10} = (5 \times 2) + 1 = ((2 \times 2)+1) + 1 = (((1 \times 2) + 0) + 1) + 1 = 1011_2$

Octal Encoding:
Encoding in base 8 is also known as 'octal encoding'. Since 8 is a power of 2, we can simply convert between binary and octal without having to first convert to decimal.

To convert from binary to octal, we simply split the binary number up into sets of 3 bits (adding 0s to the front of the number if there are not enough digits). These sets each correspond to one octal digit depending on their binary value.

$100101111_2 = 100 + 101 + 111 = 457_8$

To convert from octal to binary, we simply convert each digit of the number into a 3-digit binary number concatenate them. Octal encoding is often used to more compactly represent long binary numbers. Octal numbers are either denoted with a subscript 8 or prefaced with an '0'.

Hexadecimal Encoding:
Encoding in base 16 is also known as 'hexadecimal encoding'. The extra 6 digits used in base 16 notation are the letters A to F. Since 16 is also a power of 2, we can use the octal methods of conversion to convert between binary and hexadecimal. The only difference is that we split binary numbers up into sets of 4 bits rather than 3 as each hexadecimal digit corresponds to a 4-bit binary number. Hexadecimal numbers are either denoted with a subscript 16 or prefaced with '0x'.

Size of an Encoding:
Digital circuits can only manipulate binary numbers with a finite number of bits. This means that if we are encoding natural numbers using 10 bits, we can only represent the numbers 0 to 1023 ($2^{10} - 1$). If we try to add 1 to 1023, we do not have enough bits to represent this sum. This situation is called 'overflow'. The more bits used in a circuit, the more advanced the circuit has to be and the more complex calculations become. This is why microprocessor designers have to create circuits with a reasonable amount of bits for the type of calculations performed by the circuit.

Binary Encoding Negative Integers:
Negative binary numbers can be represented in two different ways including sign-and-magnitude and 2's complement.

Sign-and-Magnitude Representation:
To represent a positive binary number as a negative binary number in sign-and-magnitude representation we simply add a bit to the beginning of the number to represent the sign. A 0 is used to represent a positive number and a 1 is used to represent a negative number. For example, +5 = 0101 and -5 = 1101. If the leftmost bit is already a 1, we just add a 1 to the beginning of the number. For n bits, this representation can be used to represent the numbers from $-2^{n-1} - 1$ to $2^{n-1} - 1$. We do not tend to use this representation since there are two ways of representing the number zero, thus wasting the opportunity to represent an additional number.

2's Complement Representation:
This representation is more commonly used because there is only one representation of the number zero. For n bits, this representation can be used to represent the numbers from -2^{n-1} to $2^{n-1} - 1$. The rightmost bit still represents the sign. To obtain an n-bit negative binary number (K) in 2's complement representation, we subtract the positive version of binary number (P) from 2^n.

$K = 2^n - P$

In other words, all we have to do is invert all the 0s and 1s in the positive binary number and add 1 to the rightmost bit. For example, +5 = 0101 and -5 = 1011. To extend a number in 2's complement representation, we add 1s to the front of the number.

To convert from a 2's complement representation of a negative binary number to a decimal number, we first invert all the 0s and 1s, then we add 1. Now we convert this number to a positive decimal number and add a minus sign.

Binary Encoding Floating-Point Numbers:
Encoding floating-point numbers is more difficult than encoding integers because there are an infinite amount of floating-point numbers in an interval. To combat this, we restrict floating-point numbers to a certain interval and certain values within that interval. Floating-point numbers are represented using only a sign, a mantissa (the part of a number after the floating point) and an exponent. This is called 'floating-point representation' and is similar to scientific notation except we use a mantissa instead of one non-zero digit.

We binary encode a floating-point number the same way we binary encode a natural number, by adding the corresponding powers of two for the number before the floating point, but we also add the negative powers of two for the mantissa.

$2.5_{10} = 2 + 0.5 = 2^1 + 2^{-1} = 10.1_2$

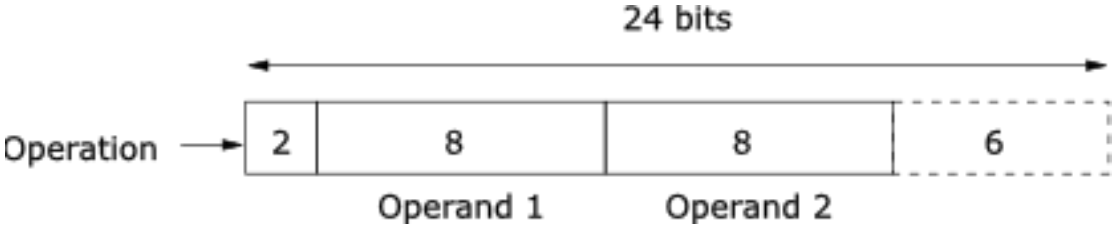
Note that we can move the floating point of a number by multiplying or dividing by the base. To express a binary number in floating-point format we need to know how many bits are used to describe the mantissa and how many are used to describe the exponent. These amounts of bits determine the range and precision of numbers they can represent. The more bits used to represent the exponent, the greater the range interval. The more bits used to represent the mantissa, the greater the amount of numbers that can expressed for each exponent is (the precision). This precision is not to be confused with accuracy, which refers to how close to the real number the binary encoded number is.

When a number to be represented is outside the range, an 'overflow' is produced. When a number cannot be represented because the precision is too low, an 'underflow' is produced.

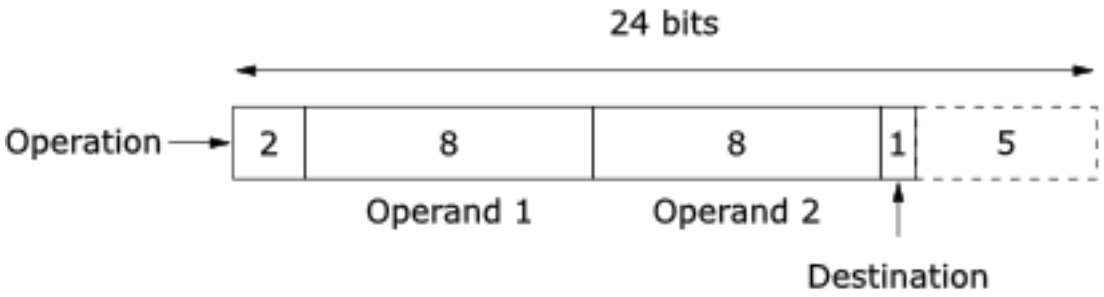
Encoding Other Elements:
Bits can be used to encode more than just numbers. Any set of elements can be encoded by assigning different combinations of bits to each element. The amount of bits must of

course satisfy $N \leq 2^n$ or $n \geq \log_2 N$ where n is the amount of bits, and N is the amount of elements. An example of encoding scheme for a set of characters is ASCII, which applies an 7-bit binary code to each key on a keyboard.

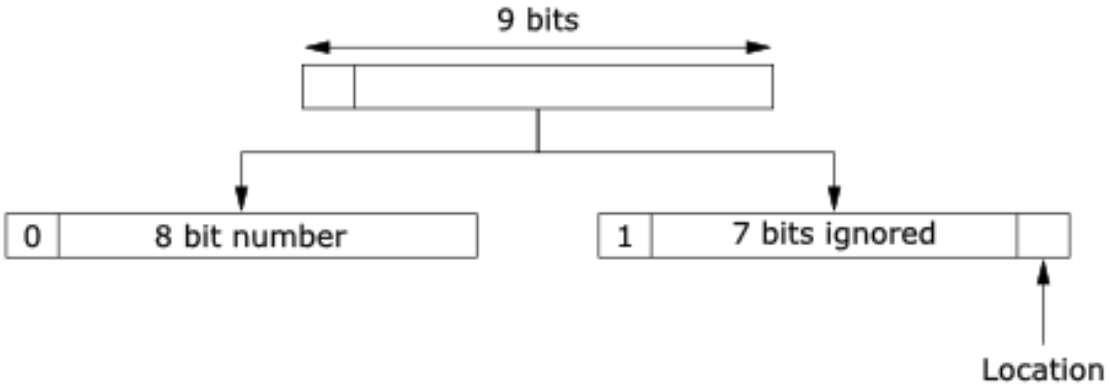
Encoding Instructions:
Instructions are encoded is many different ways depending on their function. An example of an encoding scheme for a set of instructions is ual-1. In ual-1, instead of enumerating every possible instruction, we divide each instruction into a set of bit and encode them separately. For example, for instructions that add, subtract, multiply, or divide two operands, two bits would be used for the instruction, and an additional 8-bits might be used to represent each operand depending on their size. We may have to add some extra filler bits on the end so the whole instruction can be hexadecimal encoded.



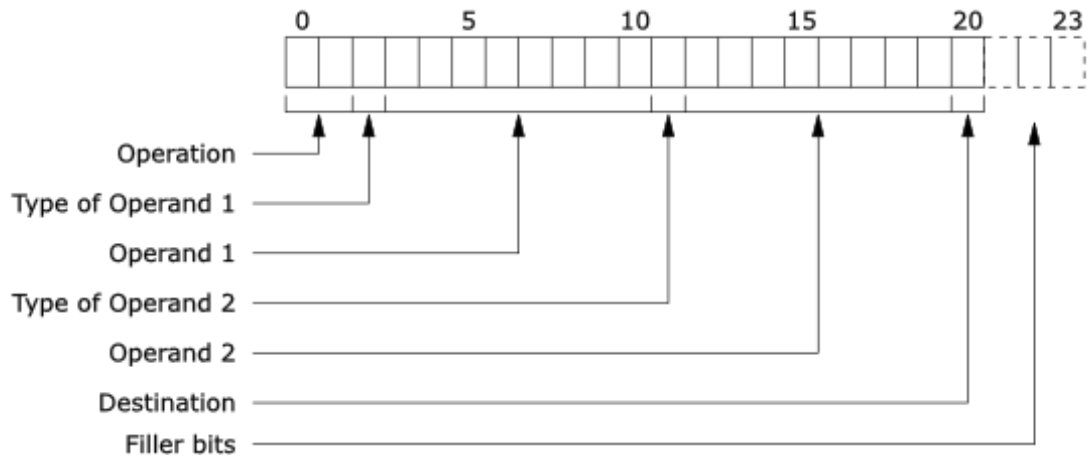
Another encoding scheme is ual-2, which is the same as ual-1 except we have an additional set of bits to denote the location where the result of the instruction will be stored.



The ual-3 encoding scheme allows us to perform operations on the values stored in the locations from ual-2. We use 9-bits to encode each operand. If the leftmost bit is 0 then the remaining 8 represent the operand. If the leftmost bit is 1 then the rightmost bit determines the location of the operand.



The result of this calculation must also be stored in a location, and so we might need 24 bits to represent an instruction in ual-3.



Considering that some bits may be ignored when using the ual-3 encoding scheme, we can encode some instructions using less bits than others. We can also create special programs that detect when an instruction will require or contain less bits. These programs are said to decode 'incrementally' and these instructions are said to be of 'variable format' rather than of 'fixed format'.

The encoded instructions of a processor are called its 'machine language'. Each processor has its own machine language that defines its executable instructions and their binary encoding.