Exam 1 Notes

Compilation

The following are definitions of common compilation processes.

- **Preprocessing (CPP)**: The preprocessor handles directives for source code file inclusion, macro definitions, and conditional compilation.
- **Compilation (CC)**: The compiler takes the preprocessed source code and converts it into assembly code.
- **Assembly (AS)**: The assembler then takes this assembly code and translates it into machine code, producing object files.
- **Linking (LD)**: Finally, the linker combines these object files into a single executable program.

Memory Hierarchy

At the top, we have registers, which are the fastest type of memory within a computer. Below registers are levels of cache memory (L1, L2, and L3), each slower than the last but still faster than RAM (Random Access Memory). Further down are disks, like HDDs or SSDs, which provide more storage but at slower access speeds. Lastly, remote storage, which can be cloud storage or network-attached storage, offers the most space but has the slowest access speed due to its physical and network distance from the CPU.

Memory Hierarchy

`Registers` → `L1 Cache` → `L2 Cache` → `L3 Cache` → `RAM` → `Disks` → `Remote Storage`

Abstraction

In the context of computer systems, an abstraction is a simplification where complex details are hidden to reduce complexity and increase efficiency. It allows users and programs to interact with systems and devices at a higher level without concern for the underlying implementation details. This concept is central to computer science because it enables the development of complex systems and applications by breaking them down into more manageable parts. Each layer of abstraction provides a set of interfaces for the level above, ensuring that changes in one layer do not necessarily affect others.

- Files abstract the details of I/O devices, allowing users and programs to interact with data storage without needing to understand the specifics of the hardware.
- Virtual memory abstracts the physical memory, giving an application the impression of having a contiguous and large amount of memory while physically it could be fragmented and less than the virtual space.
- Processes abstract the execution of multiple tasks, giving the impression that there is more than one processor executing different tasks simultaneously when, in fact, the CPU switches between tasks to give the illusion of concurrency.

Propositional Logic

Propositional logic is a branch of logic that deals with propositions, which are statements that can be either true or false. It involves logical operations such as:

- **AND (Conjunction)**: A logical operator that results in true if both operands are true.
- **OR (Disjunction)**: A logical operator that results in true if at least one of the operands is true.
- **XOR (Exclusive OR)**: A logical operator that results in true only if one operand is true and the other is false.

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise Operations

Bitwise operators perform operations on binary representations of numbers:

- **& (AND)**: Sets each bit to 1 if both bits are 1.
- **| (OR)**: Sets each bit to 1 if one of the bits is 1.
- **^ (XOR)**: Sets each bit to 1 if only one of the two bits is 1.
- **~ (NOT)**: Inverts all the bits.
- **! (Logical NOT)**: Inverts the truth value (used with Boolean values, not bitwise).
- **» (Right Shift)**: Shifts the bits of a number to the right by a specified number of positions.
- **« (Left Shift)**: Shifts the bits of a number to the left by a specified number of positions.

Decimal To Binary

To convert a decimal number to binary, divide the number by 2 and record the remainder. Repeat this process with the quotient until the quotient is 0. The binary representation is the sequence of remainders read in reverse (from the last remainder obtained to the first).

Decimal To Hexadecimal

To convert a decimal number to hexadecimal:

1. Divide the decimal number by 16.
2. Record the remainder.
3. Continue dividing the quotient by 16 until you get a quotient of zero.
4. The hexadecimal number is the sequence of remainders read in reverse (from the last remainder to the first).
5. Each remainder corresponds to a hexadecimal digit: 0-9 for remainders 0-9 and A-F for remainders 10-15.

Binary To Decimal

To convert binary to decimal, each bit in the binary number is multiplied by the base (2) raised to the power of its position. Starting from the rightmost bit (least significant bit), the position starts at 0 and increases by 1 as you move left. Summing these products gives the decimal equivalent.

Binary To Decimal Formula

Mathematically, if $b_n b_{n-1} \dots b_2 b_1 b_0$ is a binary number, its decimal equivalent D is:

$$D = \sum_{i=0}^{n} b_i \cdot 2^i$$

Here, b_i represents each binary digit (0 or 1), and n is the position of the digit from the right.

Binary To Hexadecimal

To convert binary to hexadecimal:

1. Group the binary number into sets of four digits (bits), starting from the right. If the leftmost group has less than four bits, add zeros to make a group of four.
2. Convert each 4-bit group to its hexadecimal equivalent, using the fact that each group represents a number from 0 to 15.
3. The hexadecimal number is the sequence of these hexadecimal digits read from left to right.

Hexadecimal To Decimal

To convert hexadecimal to decimal:

1. Assign each digit of the hexadecimal number a positional value, starting from 0 on the right.
2. Convert each hexadecimal digit to its decimal equivalent (0-9 stay the same, and A-F correspond to 10-15).
3. Multiply each digit by 16 raised to the power of its positional value.
4. Sum these values.

Hexadecimal To Decimal Formula

Mathematically, if $h_n h_{n-1} \dots h_2 h_1 h_0$ is a hexadecimal number, its decimal equivalent D is

$$\boxed{D = \sum_{i=0} h_i \cdot 16^i}$$

Here, h_i represents each hexadecimal digit, and n is the position of the digit from the right.

Hexadecimal To Binary

To convert hexadecimal to binary:

1. Convert each hexadecimal digit to its 4-bit binary equivalent.
2. Each digit in hexadecimal corresponds to four binary digits, as hexadecimal is base-16 and binary is base-2.
3. Concatenate these binary groups to get the final binary number.

Binary, Decimal, Hexadecimal Conversions

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Left Shift

In this operation, the binary digits of a number are moved a certain number of places to the left. For every shift left, a zero is added to the rightmost end, and the leftmost bit is discarded. This operation effectively multiplies the original number by 2 for each shift position (since binary is base 2). For instance, shifting `1011` left by one position results in `0110`, which doubles the original number. The 3-bit left shift multiplies the number by 2^3 or 8.

Logical Right Shift

This operation shifts all the bits to the right by the specified number of positions. For non-negative numbers, logical and arithmetic right shifts yield the same result. It fills in the leftmost bits with zeros. This operation effectively divides a number by 2 raised to the number of shifts.

Arithmetic Right Shift

Used with signed numbers, it shifts values to the right but fills in the new leftmost bit with the sign bit (the original leftmost bit) instead of zeros. This preserves the sign of the number in two's complement form, which is used to represent negative numbers. This operation effectively divides a number by 2 raised to the number of shifts.

C Word Type Declarations

In C, different data types have different sizes when declared. The following is a table of typical data types and the number of bytes that the specific data type takes up in memory.

Data Type	32 Bit Architecture	64 Bit Architecture
`char`	1	1
`short int`	2	2
`int`	4	4
`long int`	4	8
`long long int`	8	8
`char*`	4	8
`float`	4	4
`double`	8	8

Unsigned Integer Representation

In unsigned integers (integers that can only be positive) the number conversion from binary to decimal is in the traditional sense.

Unsigned Integer Examples

Below are some examples of unsigned integers:

$$\texttt{0b00001101} = 13$$
$$\texttt{0b01110010} = 114$$
$$\texttt{0b00101011} = 43$$
$$\texttt{0b10101110} = 174$$

Signed Integer Representation

In signed integers (integers that can be negative) the most significant bit (MSB) is referred to as the 'sign' bit. 0 for positive and 1 for negative. The process of converting a binary number to a decimal is the same, but the MSB is calculated in the number as $-1 \cdot 2^n$ where n is the MSB. The rest of the digits are calculated in the same manner as an unsigned (positive) value.

Signed Integer Examples

Below are some examples of signed integers:

$$\texttt{0b10101001} = -87$$
$$\texttt{0b10101110} = -82$$
$$\texttt{0b10110011} = -77$$
$$\texttt{0b11011101} = -35$$
$$\texttt{0b11111100} = -4$$
$$\texttt{0b00001101} = 13$$
$$\texttt{0b00101011} = 43$$
$$\texttt{0b01110010} = 114$$

Binary Addition

The formula for binary addition is:

1. Start from the least significant bit (rightmost side).
2. Add the bits in each column.
3. If the sum in a column is 2 (10 in binary), write 0 and carry over 1 to the next left column.
4. If the sum is 3 (11 in binary), write 1 and carry over 1.
5. Proceed to the next column to the left, adding the carried over value.

Binary Subtraction

Binary subtraction is similar to decimal subtraction except it follows the rules of base 2. Here's a formulaic approach:

1. Start from the least significant bit (rightmost side).
2. Subtract the second number's bit from the first number's bit in each column.

- $0 - 0 = 0$
- $0 - 1 = 1$ (With a borrow of 1)
- $1 - 0 = 1$
- $1 - 1 = 0$

3. If the bit from the first number is less than the bit from the second number, borrow 1 from the next column to the left (which is equivalent to adding 2 in binary).
4. Continue the process for each column until the subtraction is complete.

Unsigned Integer Overflow

Because there are minimum and maximums for numbers represented in binary for a given number of bits, overflow can occur. The range of values for a binary number b will range from

$$0 \rightarrow 2^n - 1$$

where n is the number of digits in the binary representation. For example, an 8 bit unsigned integer can range from 0 to 255.

Signed Integer Overflow

Overflow applies the same for signed integers as it does for unsigned integers, but just slightly different. Because signed integers can represent negative values, the range of values for a binary number b will range from

$$-1 \cdot 2^{n-1} \rightarrow 2^n - 1$$

where n again is the number of digits in the binary representation. For example, an 8 bit signed integer can range from -128 to 127.

Floating Point Representation

For a floating point value, there are two main parts:

- **Integer Values**: These bits are in front of the decimal point - `XX.YYY ...`
- **Fractional Values**: These bits are after the decimal point = `...XX.YYY ...`

Fractional Binary To Decimal

To convert from a binary fractional number we add up the integer values like regular binary to decimal conversions. For the fractional bits, we add up the fractional bits with negative exponents indexing from -1 to n. For n fractional bits b_f, the fractional representation F is calculated with

$$\boxed{F = \sum_{f=1}^{n} b_f \cdot 2^{-f}.}$$

Take for example the following example.

Fractional Binary Example

Take for example the fractional binary number `11.0001` is going to be

$$\text{Integer Values} = 1 \cdot 2^1 + 1 \cdot 2^0 = 2 + 1 = 3$$
$$\text{Fractional Values} = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} = \frac{1}{16} = 0.0625.$$

The final decimal value D is then

$$\boxed{D = 3.0625_{10}.}$$

IEEE Floating Point Representation

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established by the Institute of Electrical and Electronics Engineers (IEEE). The standard defines formats for representing floating-point numbers (including negative zero and special "Not a Number" (NaN) values) and establishes guidelines for floating-point arithmetic in computer systems.

The key components of IEEE floating point representation are:

- **Sign Bit**: A single bit is used to denote the sign of the number. A 0 represents a positive number, and a 1 represents a negative number.
- **Exponent**: A set of bits that follow the sign bit, used to represent the exponent for the number. The exponent is stored in a biased form, meaning that a fixed value (the bias) is subtracted from the actual exponent to get the stored exponent. The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field. The exponent in IEEE 754 is used to represent both very large and very small numbers.
- **Mantissa (Significand)**: This is the fraction part of the floating-point number, representing the significant digits of the number. The binary point is assumed to be just to the right of an implicit leading bit (which is typically a 1, except for denormal numbers). The mantissa is normalized, meaning that it is scaled to be just less than 1 (for normalized numbers), which is represented by the implicit leading bit.
- **Special Values**: IEEE floating-point format can represent special values such as infinity (both positive and negative) and NaN (Not a Number), which are used to denote results of certain operations that do not yield a numerical value.

Denormalized And Normalized Values

Normalized and denormalized values play a crucial role in the IEEE floating-point representation. In the context of IEEE 754 standard, normalized values are represented with an implicit leading bit equal to 1, allowing for a higher precision and a wider range of representable numbers. On the other hand, denormalized values, also known as subnormal numbers, lack the implicit leading bit and are used to represent numbers close to zero, which fall below the normal range of the floating-point format. These denormalized numbers enable a graceful underflow, ensuring that very small numbers can still be represented with reduced precision.

Denormalized numbers can be summed up with the following:

- Denormalized numbers are used to represent values that are too small to be normalized (those that are closer to zero than what can be represented by a normalized value). These numbers do not have an implicit leading 1.
- In denormalized form, the exponent is all zeros, and the mantissa is allowed to begin with a series of zeros. This allows for representation of numbers closer to zero than is possible with normalized form, albeit with less precision.
- For example, a binary number `0.00101` (in denormalized form) cannot assume an implicit leading 1 and must store all bits explicitly.

Normalized numbers can be summed up with the following:

- In normalized representation, the floating-point number is scaled such that the leading digit of the mantissa is always a 1 (except for 0). Because this leading digit is always 1, it doesn't need to be stored, which effectively gives one more bit of precision. This is known as an "implicit leading bit."

- For a binary floating-point system, this means the mantissa (or significand) is always in the range of [1,2) (including 1 but excluding 2).
- For example, a binary number `1.101` (in normalized form) is represented with an implicit 1, so only `.101` needs to be stored.
- The exponent is adjusted accordingly to represent the correct scale of the number and is not all zeros or all ones.

IEEE To Decimal

When converting from IEEE floating point binary to decimal, there are specific values that we need to calculate to in order to convert the number to decimal representation. Here is the recipe for doing so:

- **Bias**: $b = 2^{k-1} - 1$ where k is the number of bits in the exponent.
- **Exponent**: $e =$ value of exponent in decimal.
- **Exponent With Bias**: $\mathbf{E} = \mathbf{e} - \mathbf{b}$ (Normalized) i.e. $e \neq 0$, $\mathbf{E} = \mathbf{1} - \mathbf{b}$ (Denormalized) i.e. $e = 0$.
- **Mantissa**: $\mathbf{M} = \mathbf{1} + \mathbf{f}$ (Normalized) i.e. $e \neq 0$, $\mathbf{M} = \mathbf{f}$ (Denormalized) i.e. $e = 0$.
- **Sign**: s: 1 if sign bit is 0, -1 if sign bit is 1.

Compiling this to calculate the decimal representation D we have

$$\boxed{D = s \cdot 2^E \cdot M.}$$

IEEE To Decimal Example

Consider the IEEE floating point binary number `0b101001100000` that has 1 sign bit, 5 exponent bits, and 6 fractional bits.

$$\begin{aligned}
\textbf{Sign}\text{: } s &: \text{Sign bit is a 1} \;\; \therefore \;\; s = -1 \\
\textbf{Bias}\text{: } b &: b_n = 5 \;\; \therefore \;\; b = 2^{5-1} - 1 = 2^4 - 1 = 16 - 1 = 15 \\
\textbf{Exponent}\text{: } e &: e = 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 9 \\
\textbf{Biased Exponent}\text{: } E &: e \neq 0 \;\; \therefore \;\; \text{Norm.} \;\; \therefore \;\; E = e - b = 9 - 15 = -6 \\
\textbf{Fraction}\text{: } f &: e \neq 0 \;\; \therefore \;\; \text{Norm.} \;\; \therefore \;\; f = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + \cdots + 0 \cdot 2^{-6} = 1/2 \\
\textbf{Mantissa}\text{: } M &: e \neq 0 \;\; \therefore \;\; \text{Norm.} \;\; \therefore \;\; M = 1 + 1/2 = 3/2 = \texttt{0b1.1} \\
\textbf{Decimal Value}\text{: } D &= s \cdot 2^E \cdot M = -1 \cdot 2^{-6} \cdot 3/2 = -1 \cdot \frac{1}{64} \cdot \frac{3}{2} = -\frac{3}{128} = -0.0234375.
\end{aligned}$$

Therefore the decimal value is then

$$\boxed{\texttt{101001100000}_2 = -\frac{3}{128} = -0.0234375_{10}.}$$

Decimal To IEEE

We can convert decimal values to IEEE as well. Below is a recipe for doing so.

- Determine the sign bit (s) based on the sign of the decimal value. If the value is positive, the sign bit is 0; if the value is negative, the sign bit is 1.
- Convert the absolute value of the decimal number into binary form.
- Calculate the bias (b) for the number of exponents.

- Calculate the maximum number of shifts (α) when writing the binary number in scientific notation.
  - $\alpha = 1 - b$
- Normalize the absolute value of the binary number in the form $1.xxx \cdot 2^y$
- If $y > \alpha$
  - $E = y$, value is normalized.
- If $y < \alpha$
  - Normalize the absolute value of the binary number in the form $0.xxx \cdot 2^\alpha$. $E = \alpha$, value is denormalized.
- Calculate the exponent of the binary number and represent it in binary.
  - If the value is **normalized**:
    - $e = E + b$
  - If the value is **denormalized**:
    - $e = 0$
- Grab the mantissa (the digits after the decimal place in normalized binary number), these are the fraction bits (f).
- Stitch the results together: `s` + `e` + `f`

Decimal To IEEE Example

Consider the decimal number $d = -0.046875$, approximate this value to floating point IEEE with 4 exponent bits and 5 fractional bits.

$$\begin{aligned}
\textbf{Sign}&: \text{This number is negative therefore } s = 1\\
\textbf{Bias}&: b = 2^{k-1} - 1 = 2^{4-1} - 1 = 7 \;\; (k \text{ is the number of exponent bits})\\
\textbf{Alpha}&: \alpha = 1 - b = 1 - 7 = -6 \;\; (\text{Max number of shifts is } -6)\\
\textbf{Absolute Value}&: |d| = 0.046875 = 3/64\\
\textbf{Abs. Value In Binary}&: b_d = 1/32 + 1/64 = 1 \cdot 2^{-5} + 1 \cdot 2^{-6} = \texttt{0b0.000011}\\
\textbf{Normalized Binary}&: \hat{b}_d = 1.1 \cdot 10^{-5} \;\; (-5 > \alpha \therefore E = -5)\\
\textbf{Normalized Exponent}&: e = E + b = -5 + 7 = 2 \;\; (e > 0 \text{ therefore Norm.})\\
\textbf{Normalized Exponent In Binary}&: e = \texttt{0010}\\
\textbf{Mantissa}&: M = \texttt{1000}, \;\text{Values to right of normalized binary value}
\end{aligned}$$

The finally binary representation is then

$$\boxed{-0.046857_{10} = \texttt{0b1001010000}_2.}$$

Rounding

Rounding a value to IEEE is performed the same as rounding with decimals. Round the decimal value to the corresponding decimal value, and then convert to binary representation.