# Introduction

# About me

- Course Instructor
  - Ms. Ayesha Inam

# Teacher Info.

- Name: Ayesha Inam
- E-mail: ayeshainam91@gmail.com
- Visiting Hours: on appointment

# Course Info.

- Credit Hours: 3+1
- Two lectures per week each of duration 1.5 hour
- Lab class each week

# Text Books

- Text Books
  - Assembly Language for x86 Processors
    - by Kip R. Irvine
    - 6th Edition

# Grading Policy

| Class | |
|---|---|
| Assignments | 5% |
| Quizzes | 15% |
| Midterm Exams | (15+15)% |
| Class Participation | 5% |
| Final Exam | 45% |
| Total | 100% |

# Grading Policy

- All deadlines will be hard
- Re-grading can be requested after grade reporting, within following time limits:
  - Midterm: Same day
  - Assignments: 2 working days
  - Quizzes: 2 working days
  - Everything will be final on 3rd day

# General Guidelines

- Start work on project/assignment right from the first day
- No assignment will be accepted after due date
- Assignments copied from others will be marked zero
- No excuse will be accepted for a missed assignment or quiz
- Unannounced quizzes, so come prepared in the class

# Lecture 01

Week 01

# Chapter Overview

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

# Welcome to Assembly Language

- Some Good Questions to Ask
- Assembly Language Applications

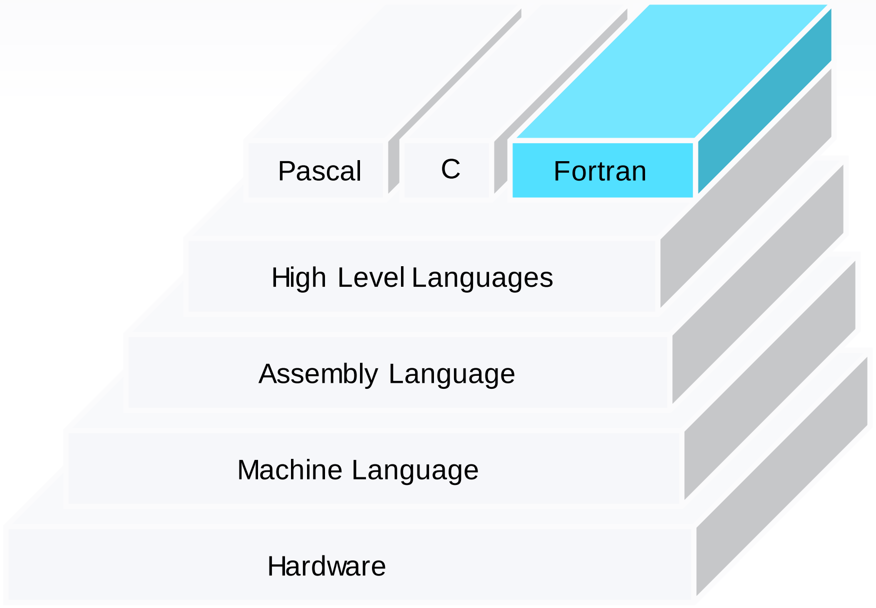
# Questions to Ask

- Why am I learning Assembly Language?
- What background should I have?
- What is an assembler?
- What hardware/software do I need?
- What types of programs will I create?
- What do I get with this book?
- What will I learn?

# Welcome to Assembly Language *(cont)*

- How does assembly language (AL) relate to machine language?
- How do C++ and Java relate to AL?
- Is AL portable?
- Why learn AL?

# Hierarchy of Computer Languages



# Assembly Language Applications

- Some representative types of applications:
  - Business application for single platform
  - Hardware device driver
  - Business application for multiple platforms
  - Embedded systems & computer games

# High Level Language

- Called High Level because closer to human language and farther from machine language
- Independent of a particular type of processor
- Easier to read, write and understand because uses natural language elements
- Hides implementation details
- Must be translated to machine language

# Assembly Language

- Low level programming language
- Used to interact with computer hardware
- Specific to a particular computer architecture
- The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*
- Focuses on programming microprocessors
- Used to program
  - Embedded system
  - Device driver programming
  - Computer viruses and bootloaders

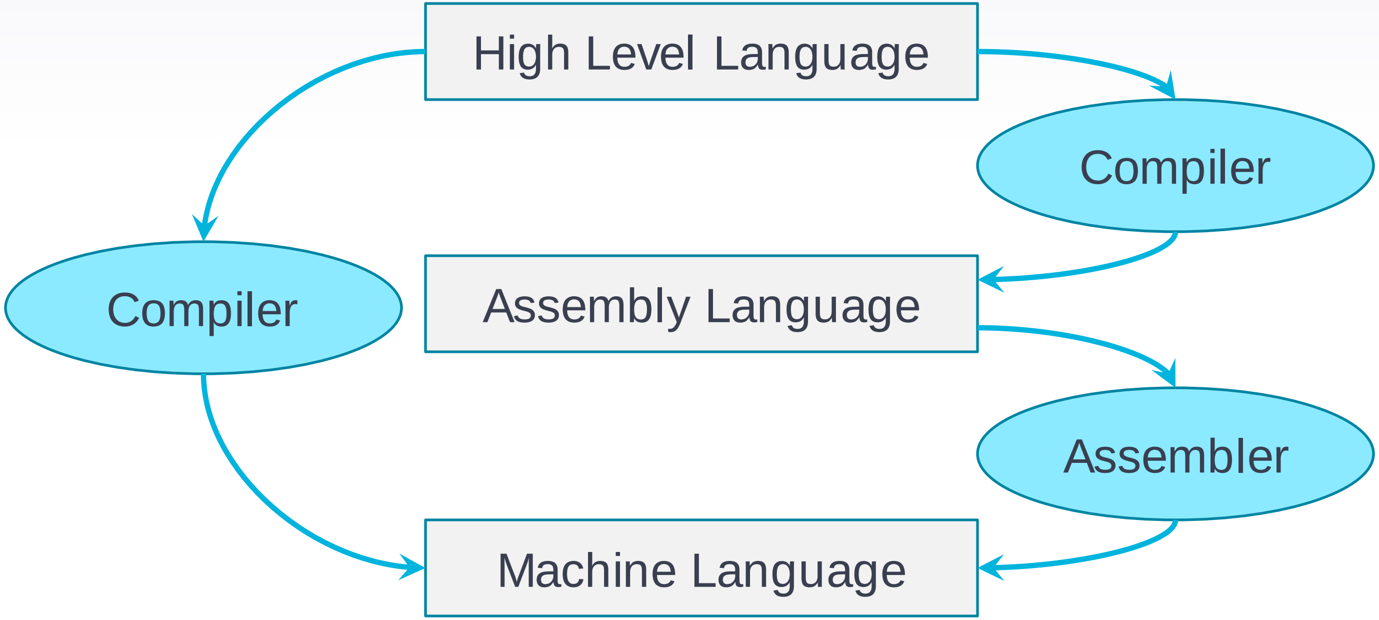
# Machine Language

- Lowest level programming language
- Sequence of 1s and 0s
- Easily understood by computers
- Almost impossible for humans to use
- Each CPU has its own unique machine language

# Conversion from High Level (HL) to Low Level (LL) Language

- From Assembly to Machine Language
  - Assembler is used
- From High Level to Machine Level Language
  - Compiler converts High Level Language to Object Code
  - Assembler is used to convert Assembly Language code to Machine Code

# Compiler and Assembler



# Assembly Language Portability

- Can be compiled and run on a wide variety of computers
- Assembly is designed for a specific processor family
- Motorola 68x00, x86, SUN Sparc, Vax, IBM-370 are different processor architectures

# Conversion from HL to LL Language

Natural Language: Add 5 into 3 and store the result into X

High Level Language: int X = 5 + 3;

**Assembly Language:**

```
mov ax, 5
mov bx, 3
add ax, bx
mov X, ax
```

# Advantages of HL Languages

- Program development is faster
  - High level statements: fewer instructions to code
- Program maintenance is easier
  - For the same above reasons
- Programs are portable
  - Contains less machine dependent details
    - Can be used with little or no modifications on different machines
  - Compiler translates to the target machine language

# Comparing ASM to High-Level Languages

| Type of Application | High-Level Languages | Assembly Language |
|---|---|---|
| Business application software, written for single platform, medium to large size. | Formal structures make it easy to organize and maintain large sections of code. | Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code. |
| Hardware device driver. | Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties. | Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented. |
| Business application written for multiple platforms (different operating systems). | Usually very portable. The source code can be recompiled on each target operating system with minimal changes. | Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain. |
| Embedded systems and computer games requiring direct hardware access. | Produces too much executable code, and may not run efficiently. | Ideal, because the executable code is small and runs quickly. |

# What's Next

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

# Virtual Machine Concept

- Virtual Machines
- Specific Machine Levels

# Virtual Machines

- Tanenbaum: Virtual machine concept
- Programming Language analogy:
  - Each computer has a native machine language (language L0) that runs directly on its hardware
  - A more human-friendly language is usually constructed above machine language, called Language L1
    - Programs written in L1 can run two different ways:
      - Interpretation – L0 program interprets and executes L1 instructions one by one
      - Translation – L1 program is completely translated into an L0 program, which then runs on the computer hardware

# Translating Languages

English: Display the sum of A times B plus C.

C++:  cout << (A * B + C);

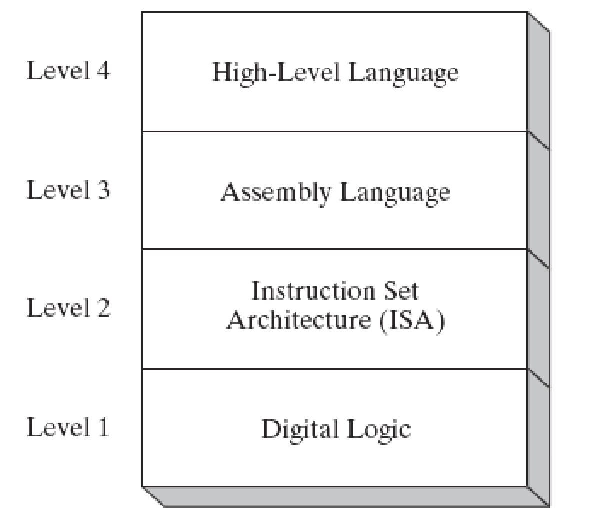
Assembly Language:

```
mov eax,A
mul B
add eax,C
call WriteInt
```

Intel Machine Language:

```
A1 00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

# Specific Machine Levels




(descriptions of individual levels follow . . . )

# High-Level Language

- Level 4
- Application-oriented languages
  - C++, Java, Pascal, Visual Basic . . .
- Programs compile into assembly language (Level 4)

# Assembly Language

- Level 3
- Instruction mnemonics that have a one-to-one correspondence to machine language
- Programs are translated into Instruction Set Architecture Level - machine language (Level 2)

# Instruction Set Architecture (ISA)

- Level 2
- Also known as conventional machine language
- Executed by Level 1 (Digital Logic)

# Digital Logic

- Level 1
- CPU, constructed from digital logic gates
- System bus
- Memory
- Implemented using bipolar transistors

next: Data Representation

# What's Next

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

# Data Representation

- Binary Numbers
  - Translating between binary and decimal
- Binary Addition
- Integer Storage Sizes
- Hexadecimal Integers
  - Translating between decimal and hexadecimal
  - Hexadecimal subtraction
- Signed Integers
  - Binary subtraction
- Character Storage

# Data Representation

- Four basic data representation techniques
  - Binary(base 2)
  - Octal(base 8)
  - Decimal(base 10)
  - Hexadecimal(base 16)

| System | Base | Possible Digits |
|---|---|---|
| Binary | 2 | 0 1 |
| Octal | 8 | 0 1 2 3 4 5 6 7 |
| Decimal | 10 | 0 1 2 3 4 5 6 7 8 9 |
| Hexadecimal | 16 | 0 1 2 3 4 5 6 7 8 9 A B C D E F |

# Translating Binary to Decimal

Weighted positional notation shows how to calculate the decimal value of each binary bit:

$$\boldsymbol{dec = (D_{n-1} \times 2^{n-1}) + (D_{n-2} \times 2^{n-2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)}$$

D = binary digit

binary 00001001 = decimal 9:

$$(1 \times 2^3) + (1 \times 2^0) = 9$$

# Binary to Decimal (1/2)

$$\text{Dec} = (D_{n\text{-}1} \text{ x } 2^{n\text{-}1}) + (D_{n\text{-}2} \text{ x } 2^{n\text{-}2}) + \ldots + (D_1 \text{ x } 2^1) + (D_0 \text{ x } 2^0)$$

- Weighted Positional Notation method
  - $D$ = binary digit
  - $n$ = bit position number in binary number

# Binary to Decimal (2/2)

$0\,1\,1\,1_2$ → 4-bit number so $n = 4$

$D_0 = 1$

$D_1 = 1$

$D_2 = 1$

$D_3 = 0$

$$\text{Dec} = (D_{n\text{-}1} \times 2^{n\text{-}1}) + (D_{n\text{-}2} \times 2^{n\text{-}2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)$$

$$= (D_{4\text{-}1} \times 2^{4\text{-}1}) + (D_{4\text{-}2} \times 2^{4\text{-}2}) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)$$

$$= (D_3 \times 2^3) + (D_2 \times 2^2) + \ldots + (D_1 \times 2^1) + (D_0 \times 2^0)$$

$$= (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

$$= \; 7$$

# Translating Unsigned Decimal to Binary

- Repeatedly divide the decimal integer by 2. Each remainder is a binary digit in the translated value:

| Division | Quotient | Remainder |
|---|---|---|
| 37 / 2 | 18 | 1 |
| 18 / 2 | 9 | 0 |
| 9 / 2 | 4 | 1 |
| 4 / 2 | 2 | 0 |
| 2 / 2 | 1 | 0 |
| 1 / 2 | 0 | 1 |

37 = 100101

# Decimal to Binary (2/2)

- Convert $25_{10}$ into binary

| Division | Quotient | Remainder |
|---|---|---|
| 25 / 2 | 12 | **1** |
| 12 / 2 | 6 | **0** |
| 6 / 2 | 3 | **0** |
| 3 / 2 | 1 | **1** |
| 1 / 2 | 0 | **1** |

First remainder goes to LSB position

$1\,1\,0\,0\,1_2$

When quotient is 0, remainder goes at MSB position

- Final result is **0001 1001**

# Hexadecimal Integers

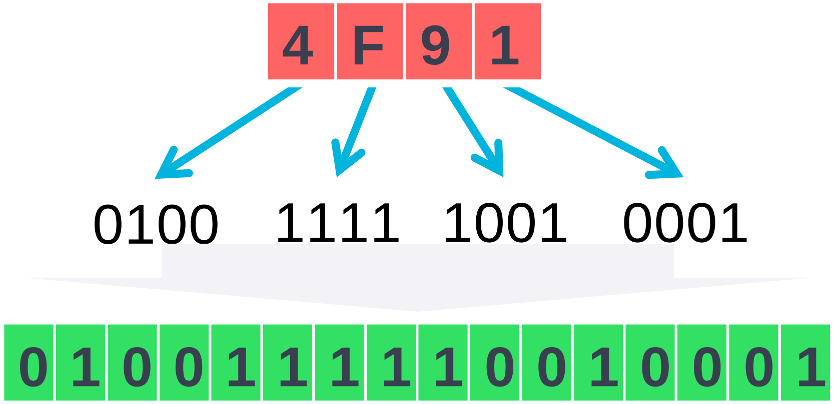
Binary values are represented in hexadecimal.

**Table 1-5** Binary, Decimal, and Hexadecimal Equivalents.

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

# Hexadecimal to Binary

- Each hexadecimal integer corresponds to 4 binary bits
- Convert each hexadecimal number to corresponding binary number



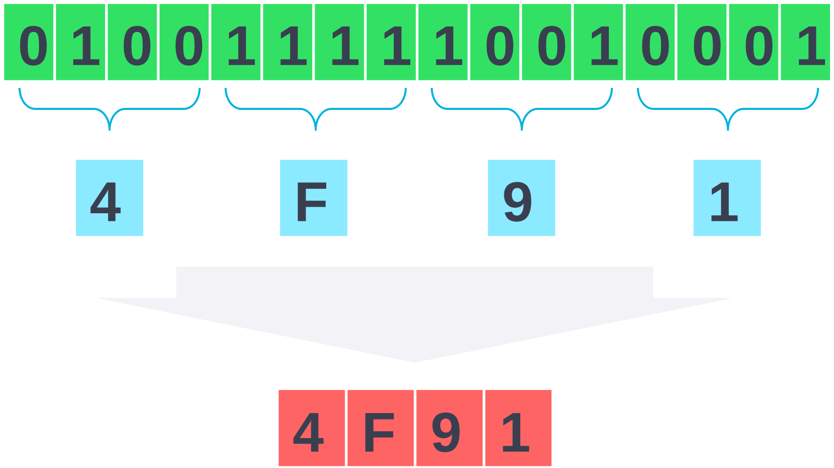
# Translating Binary to Hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer 000101101010011110010100 to hexadecimal:

| | | | | | |
|---|---|---|---|---|---|
| 1 | 6 | A | 7 | 9 | 4 |
| 0001 | 0110 | 1010 | 0111 | 1001 | 0100 |

# Binary to Hexadecimal

- Convert each 4 bits of binary into its corresponding hexadecimal



# Lecture 02

Week 01

# Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16:

  $$dec = (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

- Hex 1234 equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal 4,660.

- Hex 3BA4 equals $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal 15,268.

# Hexadecimal to Decimal (2/2)

3 B A 4 → 4-digit number so $n = 4$

$D_0 = 4$

$D_1 = A$

$D_2 = B$

$D_3 = 3$

$$= (D_{4\text{-}1} \times 16^{4\text{-}1}) + (D_{4\text{-}2} \times 16^{4\text{-}2}) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

$$= (D_3 \times 16^3) + (D_2 \times 16^2) + (D_1 \times 16^1) + (D_0 \times 16^0)$$

$$= (3 \times 4096) + (11 \times 256) + (10 \times 16) + (4 \times 1)$$

$$= (12288 + 2816 + 160 + 4) \quad = \quad \mathbf{15268}$$

# Decimal to Hexadecimal (1/2)

- Repeatedly divide the decimal integer by 16 until last quotient is 0
- Each remainder is a hex digit
- First remainder goes at least significant position and last remainder goes at most significant position

# Powers of 16

Used when calculating hexadecimal values up to 8 digits long:

| $16^n$ | Decimal Value | $16^n$ | Decimal Value |
|---|---|---|---|
| $16^0$ | 1 | $16^4$ | 65,536 |
| $16^1$ | 16 | $16^5$ | 1,048,576 |
| $16^2$ | 256 | $16^6$ | 16,777,216 |
| $16^3$ | 4096 | $16^7$ | 268,435,456 |

# Converting Decimal to Hexadecimal

| Division | Quotient | Remainder |
|---|---|---|
| 422 / 16 | 26 | 6 |
| 26 / 16 | 1 | A |
| 1 / 16 | 0 | 1 |

decimal 422 = 1A6 hexadecimal

# Decimal to Hexadecimal (2/2)

- Convert $2895_{10}$ into hexadecimal

| Division | Quotient | Remainder |
|---|---|---|
| 2895 / 16 | 180 | F |
| 180 / 16 | 11 | 4 |
| 11 / 16 | 0 | B |

First remainder goes to LS position

When quotient is 0, remainder goes at MS position

B 4 F $_{16}$

- So $2895_{10}$ = **B 4 F**$_{16}$

# Hexadecimal Addition

- Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

| | | | |
|---|---|---|---|
| | | 1 | 1 |
| 36 | 28 | 28 | 6A |
| 42 | 45 | 58 | 4B |
| 78 | 6D | 80 | B5 |

21 / 16 = 1, rem 5

Important skill: Programmers frequently add and subtract the addresses of variables and instructions.

# Hexadecimal Subtraction

- When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:

$\boxed{16 + 5 = 21}$

–1

| | |
|---|---|
| C6 | 75 |
| A2 | 47 |
| 24 | 2E. |

Practice: The address of var1 is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

# Signed Integers

- Signed integers are either positive or negative
- Not possible to stick negative sign to a number in binary numbers
- When explicitly mentioned as signed integer, then MSB decides the +ve and –ve sign
- In signed binary/octal/hex integers
  - MSB = 1 → integers is negative
  - MSB = 0 → integers is positive
- Negative integers are represented using 2's complement notation

# Forming the Two's Complement

- Negative numbers are stored in two's complement notation
- Represents the additive Inverse

| | |
|---|---|
| Starting value | 00000001 |
| Step 1: reverse the bits | 11111110 |
| Step 2: add 1 to the value from Step 1 | 11111110<br>+00000001 |
| Sum: two's complement representation | 11111111 |

Note that 00000001 + 11111111 = 00000000

# Binary Subtraction

- When subtracting A – B, convert B to its two's complement
- Add A to (–B)

$$\begin{array}{r} 00001100 \\ -\ 00000011 \\ \hline \end{array} \longrightarrow \begin{array}{r} 00001100 \\ 11111101 \\ \hline 00001001 \end{array}$$

Practice: Subtract 0101 from 1001.

# Learn How To Do the Following:

- Form the two's complement of a hexadecimal integer
- Convert signed binary to decimal
- Convert signed decimal to binary
- Convert signed decimal to hexadecimal
- Convert signed hexadecimal to decimal

# Range of Signed Numbers

- A certain number of bits can store only a fixed number of signed integers

| Bits | Range | Total Numbers |
|---|---|---|
| 8 | -128 to +127 | 256 |
| 16 | -32768 to +32767 | 65,536 |
| 32 | -2,147,483,648 to +2,147,483,647 | 4,294,967,296 |
| 64 | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | 18,446,744,073,709,551,616 |

# Range of Unsigned Numbers

- Total numbers in signed integers is exactly equal to the total numbers in unsigned integers in the same size of bits

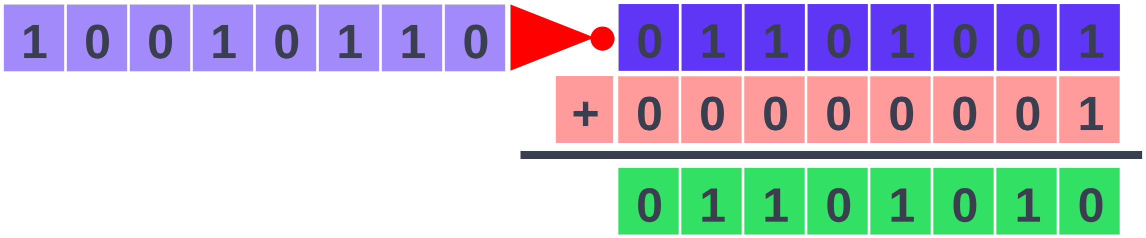
| Bits | Range | Total Unsigned Numbers |
|---|---|---|
| 8 | 0 to 255 | 256 |
| 16 | 0 to 65,535 | 65,536 |
| 32 | 0 to 4,294,967,295 | 4,294,967,296 |
| 64 | 0 to 18,446,744,073,709,551,615 | 18,446,744,073,709,551,616 |

# 2's Complement Notation

- Useful for processors to perform subtraction with addition operation
- A fixed number of bits are used to represent the numbers
- The leftmost bit is called **sign bit**
- 2's complement notation is used to represent both +ve and –ve numbers

# How to calculate 2's complement

- How to get 2's complement of a binary number?
  - Take 1's complement of that number(invert all its bits)
  - Add 1 into the inverted binary number
  - … and the result is 2's complement of that number



# 2's Complement of Hexadecimal

- Invert all bits of hex number
- All bits of hex numbers can be inverted simply by subtracting the number from $F_{16}$
- Add 1 into the inverted hex number and the result is the 2's complement
- Calculate 2's complement of $(B\ 4\ F)_{16}$

$$\begin{array}{r} F\ F\ F \\ -\ B\ 4\ F \\ \hline 4\ B\ 0 \end{array} \quad \Rightarrow \quad 4\ B\ 0 \quad \Rightarrow \quad \begin{array}{r} 4\ B\ 0 \\ +\quad\ \ 1 \\ \hline \color{red}{\mathbf{4\ B\ 1}} \end{array}$$

# Converting Signed Binary to Decimal

- If MSB is 0, then number is +ve and convert it into decimal in usual way
- If MSB is 1, then the number is in 2's complement notation and follow these steps
  - Calculate its 2's complement again
  - Convert this new number into decimal and add a –ve sign with it

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

2's comp.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

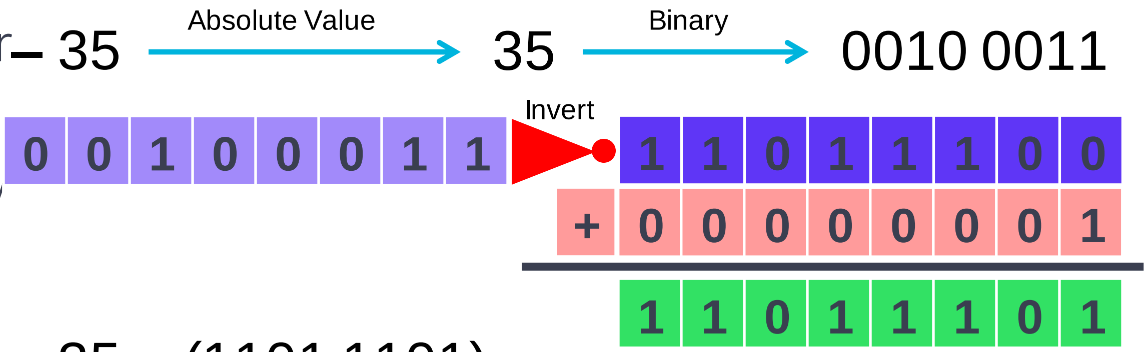
Decimal

1 0 6

- As the number was negative
  - So in decimal it is **– 1 0 6**

# Converting Signed Decimal to Binary

- Convert absolute value of decimal into binary
- If original decimal number is –ve, calculate 2's complement of the binary number
- Convert -35 to binary




$$-35 = (1101\ 1101)_2$$

# Convert Signed Decimal to Hexadecimal

- Convert absolute value of decimal to hex
- If decimal integer is –ve, create 2's complement of hexadecimal integer
- Convert -2895 to hexadecimal

$$-2895 \xrightarrow{\text{Absolute Value}} 2895 \xrightarrow{\text{Hexadecimal}} \text{B 4 F}$$

$$\text{B 4 F} \underbrace{\xrightarrow{\text{Bit Inversion}} \text{4 B 0} \xrightarrow{\text{Add 1}}}_{\text{2's Complement}} \text{4 B 1}$$

# Converting Signed Hex to Decimal (1/3)

- In signed hex number, if MSB=1, the number is –ve
- To convert it into decimal, follow these steps
  - Create its 2's complement
  - Convert the 2's complemented hex to decimal
  - Attach –ve sign to the decimal number

# Converting Signed Hex to Decimal (2/3)

- Determine if Signed $8C_{16}$ is +ve or –ve
- By converting into binary
  - If MSB = 1, then number is –ve
  - $8C_{16} = (1000\ 1100)_2$
  - Since MSB = 1, so $8C_{16}$ is –ve
- Another method
  - If leftmost digit > 7, then number is –ve
  - Since leftmost digit i.e. 8 > 7
  - $8C_{16}$ is –ve

# Converting Signed Hex to Decimal (3/3)

- Convert Signed $A3_{16}$ into decimal

A 3

A > 7 => A3 is –ve

2's complement of A3 = 5D

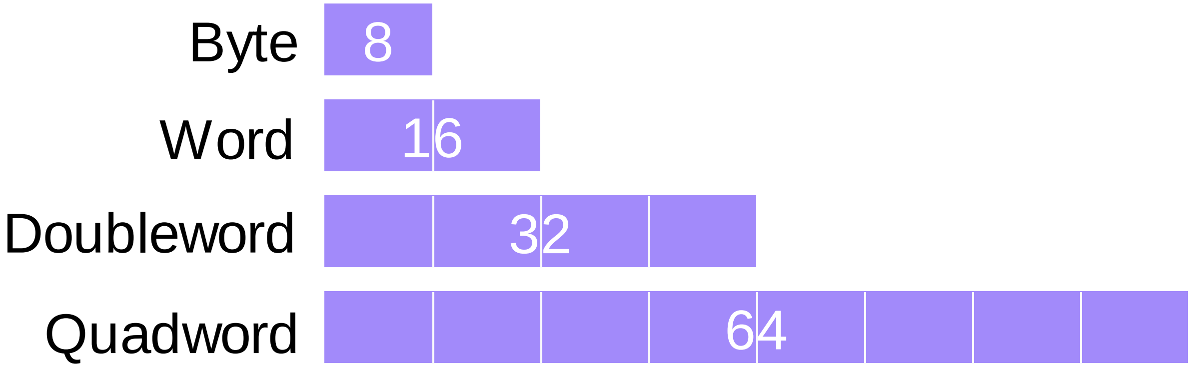
# Binary Subtraction

- Big advantage of signed number is to use same circuit for addition and subtraction
- To perform A – B
  - Calculate –B by taking 2's complement of B
  - Perform A+(-B)

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | | | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| - | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | ⇒ | + | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| | | | | | | | | | | | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

# Integer Storage System (1/2)

- **Byte** is the basic storage unit in x86 architecture
- Byte is composed of **8 bits**



# Integer Storage System

- Some larger measurements units
  - One kilobyte = $2^{10}$ bytes = 1024 bytes
  - One megabyte = $2^{20}$ bytes = 1,048,576 bytes
  - One gigabyte = $2^{30}$ bytes = 1,073,741,824 bytes
  - One terabyte = $2^{40}$ bytes = 1,099,511,627,776 bytes
  - One petabyte = $2^{50}$ bytes = $2^{40}$ kilobytes
  - One exabyte = $2^{60}$ bytes = $2^{10}$ petabytes
  - One zettabyte = $2^{70}$ bytes = $2^{30}$ terabytes
  - One yottabyte = $2^{80}$ bytes = $2^{20}$ exabytes

# Character Storage

- Character sets
  - Standard ASCII (0 – 127)
  - Extended ASCII (0 – 255)
  - ANSI (0 – 255)
  - Unicode (0 – 65,535)
- Null-terminated String
  - Array of characters followed by a *null byte*
- Using the ASCII table
  - back inside cover of book

# Numeric Data Representation

- pure binary
  - can be calculated directly
- ASCII binary
  - string of digits: "01010101"
- ASCII decimal
  - string of digits: "65"
- ASCII hexadecimal
  - string of digits: "9C"

next: Boolean Operations

# What's Next

- Welcome to Assembly Language
- Virtual Machine Concept
- Data Representation
- Boolean Operations

# Boolean Operations

- NOT
- AND
- OR
- Operator Precedence
- Truth Tables

# Boolean Algebra

- Based on symbolic logic, designed by George Boole
- Boolean expressions created from:
  - NOT, AND, OR

| Expression | Description |
|---|---|
| $\neg X$ | NOT X |
| $X \wedge Y$ | X AND Y |
| $X \vee Y$ | X OR Y |
| $\neg X \vee Y$ | ( NOT X ) OR Y |
| $\neg(X \wedge Y)$ | NOT ( X AND Y ) |
| $X \wedge \neg Y$ | X AND ( NOT Y ) |

# Operator Precedence

- Examples showing the order of operations:

| Expression | Order of Operations |
|---|---|
| $\neg X \vee Y$ | NOT, then OR |
| $\neg (X \vee Y)$ | OR, then NOT |
| $X \vee (Y \wedge Z)$ | AND, then OR |

# Truth Tables (1 of 3)

- A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg X \vee Y$

| **X** | **¬X** | **Y** | **¬X ∨ Y** |
|---|---|---|---|
| F | T | F | T |
| F | T | T | T |
| T | F | F | F |
| T | F | T | T |

# Truth Tables (2 of 3)

- Example: $X \wedge \neg Y$

| X | Y | $\neg$Y | X $\wedge$ $\neg$Y |
|---|---|---|---|
| F | F | T | F |
| F | T | F | F |
| T | F | T | T |
| T | T | F | F |

# Summary

- Assembly language helps you learn how software is constructed at the lowest levels
- Assembly language has a one-to-one relationship with machine language
- Each layer in a computer's architecture is an abstraction of a machine
  - layers can be hardware or software
- Boolean expressions are essential to the design of computer hardware and software