CMPT 295
Machine-Level Programming
Lecture 8 – Introduction

C program -> assembly code -> machine level code
IEEE Floating Point Representation

- Denormalized values
  - Condition: \( \text{exp} = 0000...0 \)
  - \( 0 \leq \text{denormalized values} < 1 \), equidistant because all have same \( 2^E \)

- Special values
  - Condition: \( \text{exp} = 1111...1 \)
    - Case 1: \( \text{frac} = 000...0 \) -> \( \infty \) (infinity)
    - Case 2: \( \text{frac} \neq 000...0 \) -> NaN

- Interesting numbers:
  - Smallest and largest denormalized/normalized, 0, 1 ...

- Impact on C
  - Conversion/casting, rounding
  - Not the same as real arithmetic: violates associativity
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
  - Assembly language basics: data, move operation
  - Operations -> Arithmetic & logical operations
  - Conditional Statement
  - Loop
  - Function/procedure call – Stack – Recursion
  - Array
  - Floating-point operations
What could these bits represent? What info do they encode?

\[ 01000010\ 01101001\ 01110100\ 01110011 \]

Answer:

- Aside from characters, integers or floating point numbers, etc...
- Review: We saw that all modern computers, designed based on the von Neumann architecture, store their programs in memory
  - Data and code of our C program are in main memory together (but in different locations)
- So, these bits could represent code, for example:
  
  \[\begin{align*}
  0:48 & \quad 83 & \quad \text{ec} & \quad 18 & \quad \text{sub} & \quad 0x18, & \quad \%rsp
  \end{align*}\]
C program in memory?

We have just spent a few lectures looking at how our data can be represented as a series of 0's and 1's, now ...

- **Question**: How does our C program end up being represented as a series of 0's and 1's?

- **Question**: Then, how does our C program (now represented as a series of 0's and 1's) end up in memory?

- **Question**: Then, how is our C program (now represented as a series of 0's and 1's stored in memory) executed by the CPU?
Turning C into machine level code - gcc

1. **C program** *(sum_store.c)*
   - C Preprocessor: `gcc -E sum_store.c > sum_store.i`

2. **Preprocessed Source** *(sum_store.i)*
   - C Compiler: `gcc -Og -S sum_store.i`
     `gcc -Og -S sum_store.c`

3. **Assembly** *(sum_store.s)*
   - Assembler: `gcc -g -c main.s sum_store.s`

4. **Object file** *(sum_store.o)*
   - Linker: `gcc -o ss main.o sum_store.o`

5. **Executable program** *(ss)*

Optional steps:
- **Disassembler**
- **gdb/ddd debugger**
Snapshot of compiled code

- **C code**
  - `*dest = sum;`
  - Store `sum` in memory designated by pointer `dest`

- **Assembly code**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - `sum:` Register `%rax`
    - `dest:` Register `%rbx`
    - `*dest:` Memory `M[%rbx]`

- **Object code (0's and 1's)**
  - 3-byte instruction
  - Stored at address `0x40059e`
**Question:** How is our C program (once represented as a series of 0’s and 1’s stored in memory) executed by the CPU?

**Answer:** The CPU executes the following simple loop

```
DO FOREVER:
    fetch next instruction from memory into CPU
    decode the instruction
    update the program counter
    execute the instruction
```
Summary

➢ Review: von Neumann architecture
  ➢ Data and code are both stored in memory during program execution

➢ Question: How does our C program end up being represented as a series of 0's and 1's?
  ➢ C program -> assembly code -> machine level code
  ➢ gcc: C preprocessor, C compiler, assembler, linker
  ➢ Snapshot of compiled code
    ➢ Example of a C statement compiled into assembly code which is then assembled into a machine level instruction

➢ Question: How does our C program (now represented as a series of 0's and 1's) end up in memory?
  ➢ When C program is executed

➢ Question: How is our C program (now represented as a series of 0's and 1's stored in memory) executed by the CPU?
  ➢ CPU executes C program by looping through the fetch-execute cycle
Next Lecture

- Introduction
  - C program -> assembly code -> machine level code

- Assembly language basics: data, move operation

- Operations -> Arithmetic & logical operations

- Conditional Statement

- Loop

- Function/procedure call – Stack – Recursion

- Array

- Floating-point operations