CMPT 295

Unit - Machine-Level Programming

Lecture 8 – Introduction

Compilation process: C -> assembly code -> machine level code

Last Lecture

- Most fractional decimal numbers cannot be exactly encoded using IEEE floating point representation -> rounding
- Denormalized values
  - Condition: **`exp = 0000…0`**
  - 0 <= denormalized values < 1, equidistant because all have same 2^E
- Special values
  - Condition: **`exp`** = **`1111…1`**
    - Case 1: **`frac`** = **`000…0 ->`** ∞ (infinity)
    - Case 2: **`frac`** ≠ **`000…0 ->`** NaN
- Impact on C
  - Conversion/casting, rounding
  - Arithmetic operators:
    - Behaviour not the same as for *real* arithmetic => violates associativity

Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point data & operations

What could these 32 bits represent? What kind of information could they encode?

$00011000\ 11101100\ 10000011\ 01001000_2$

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 instructions of our C program are in main memory together (but in different locations)
- So, these bits could represent code, for example:
  - Assembly code: `sub $0x18, %rsp`
  - Machine code: `48 83 ec 18`

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 I’s, now …

1. Question: How does our C program end up being represented as a series of 0’s and 1’s (i.e., as machine code)?
2. Question: Then, how does our C program (once it is represented as a series of 0’s and 1’s) end up being stored in memory?
3. Question: Then, how does our C program (once it is represented as a series of 0’s and 1’s and it is stored in memory) end up being executed by the microprocessor (CPU)?

Demo – C program: sum_store.c

1. Question: How does our C program end up being represented as a series of 0’s and 1’s (i.e., as machine code)?

   Let’s answer these questions with a demo

Turning C into machine code - gcc

The Big Picture

C program (.c) → `sum_store.c`

Preprocessed Source → `sum_store.i`

C Preprocessor `gcc -E sum_store.c > sum_store.i` (*text* → *text*)

C Compiler `gcc -Og -S sum_store.i`
OR `gcc -Og -S sum_store.c`

Assembly program (.s) → `sum_store.s`

Assembler `gcc -g -c main.s sum_store.s` (*text*, *text*)

Object (.o) → `sum_store.o`

Executable → `ss`

Linker
`gcc -o ss main.o sum_store.o` (*binary*, *binary*)

Loader
`./ss 5 6`

optional

Disassembled object file
text & binary

Disassembler
`objdump -d ss`
gdb/ddd debugger

ISA - Instruction Set Architecture

Computer executes it

CPU

Memory

Snapshot of compiled code

```
*dest = sum;
```

```
movq %rax, (%rbx)
```

Memory address of this byte is **0x40059f**

```
0x40059e:   48 89 03
```

Memory address of this byte

Memory address of this byte is ____________

- C code
  - Store **sum** in memory designated by pointer **dest**
- Assembly code
  - Move an 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

sum:	Register	**%rax**
dest:	Register	**%rbx**
***dest:**	Memory	**M[%rbx]**

- Machine code (0's and 1's)
  - 3-byte instruction
  - Stored at address **0x40059e**

Fetch-Execute Cycle

PC: program counter

Def^n: register containing **address** of instruction of **ss** that is currently executing

IR: instruction register

Def^n: register containing **copy** of instruction of **ss** that is currently executing

- Question: How does our C program (once it is represented as a series of 0's and 1's and it is stored in memory) end up being executed by the microprocessor (CPU)?

- Answer: The microprocessor executes the machine code version of our C program by executing the following simple loop:

  DO FOREVER:

  fetch next instruction from memory into CPU

  update the program counter

  decode the instruction

  execute the instruction

Summary

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

1. Question: How does our C program end up being represented as a series of 0's and 1's (i.e., as machine code)?
   - Compiler: C program -> assembly code -> machine level code
   - gcc: 1) C preprocessor, 2) C compiler, 3) assembler, 4) linker
2. Question: How does our C program (once it is represented as a series of 0's and 1's) end up being stored in memory?
   - When C program is executed (e.g. from our demo: `./ss 5 6`)
3. Question: How does our C program (once it is represented as a series of 0's and 1's and it is stored in memory) end up being executed by the microprocessor (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
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmov*`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations