This Week's Action Items

- Read Chapter 3
- Complete Quiz 4
- Start Assignment 2
  - Pre-Assignment Due Date: September 18 at 11:59 pm

1

1

Recap of Last Class: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

2

Floating Point

- Recall scientific notation
  - 65.4 = 6.54×10^1 (normal form – 1 non-zero leading digit)
  - Mantissa = 6.54, exponent = 1, radix (base) = 10
- Mantissa – sign/magnitude representation
- Exponent – biased notation

Single-precision

S|exp+127|significand

1| 8-bits | 23-bits

Double-precision

S|exp+1023|significand

1| 11-bits | 52-bits

3

3

Zero, Infinity, NAN

- Zero – 0|0…0|0…0
- +/- infinity – S|1…1|0…0
- Nan – S|1…1|non-zero

4

4

IEEE 754 Floating Point Rounding

- Four modes:
  - Round to +infinity – add 1 if round or sticky bit is set and result >= 0
  - Round to –infinity – add 1 if round or sticky bit is set and result < 0
  - Round to 0 (truncate)
  - Round to nearest number (default) – round to even when exactly halfway
    - Add 1 if round bit and LSB of result are set, or, if round bit and sticky bit are set
    - Minimizes mean error introduced by rounding

5

5

Closer Look at Nearest-Even

- Default Rounding Mode: Round to nearest, ties to even
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated
- Applying to Other Decimal Places / Bit Positions
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

6

Rounding Binary Numbers

- Binary Fractional Numbers
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

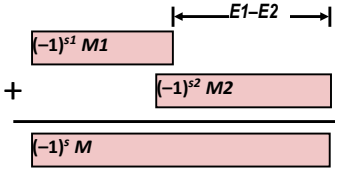
Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

7

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ +\ (-1)^{s2}\ M2\ \ 2^{E2}$
  - Assume $E1 > E2$
- Exact Result: $(-1)^s\ M\ \ 2^E$
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: $E1$
- Fixing
  - If $M \geq 2$, shift M right, increment E
  - if $M < 1$, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit `frac` precision

Get binary points lined up



8

Floating Point Addition

- Align decimal point by matching larger exponent
- Add significands (possibly unnormalized)
- Normalize – shift sum/adjust exponent
- Round number to fit in significand field

Floating Point – Additional bits for accuracy

- Carry/borrow bit – to take into account overflow (carry/borrow) in result
- Guard bit – to take care of normalizing left shift
- Round bit – for correct rounding
- Sticky bit – fine-tune rounding – additional bit to the right of round that is set to 1 if any 1 bit falls off the end of the round digit

FP Multiplication

- $(-1)^{s1}$ ***M1*** 2^{E1} x $(-1)^{s2}$ ***M2*** 2^{E2}
- Exact Result: $(-1)^{s}$ ***M*** 2^{E}
  - Sign *s*: *s1* ^ *s2*
  - Significand *M*: *M1* x *M2*
  - Exponent *E*: *E1* + *E2*
- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit **`frac`** precision

Floating Point Multiplication

- Exponent of product = sum of exponents – bias
- Multiply significands – decimal point location = sum of digits to the right of decimals
- Normalize – check for overflow/underflow
- Round
- Sign – positive if both are the same, negative if otherwise

Exceptions

- IEEE standard specifies defaults and allows traps to permit user to handle exceptions
  - Invalid operation: e.g., sqrt of –ve number, 0/0, inf/inf
    - Result is NaN
  - Overflow: result is +/- inf unless overflow exception is enabled
  - Divide by 0: result is +/- inf if exception not enabled
  - Underflow: non-zero result underflows to 0
  - Inexact: rounded result not the actual result

13

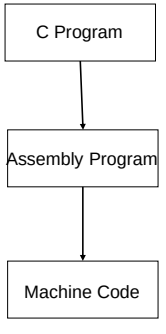
13

CSC 252: Machine-Level Programming: Instruction Set Architectures

14

14

This Module (~4 Lectures)



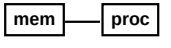
15

15

Abstract Machines

Machine Models

C



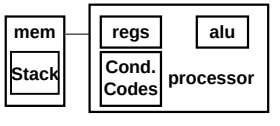
Data

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

Control

1) loops
2) conditionals
3) `switch`
4) Proc. call
5) Proc. return

Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) 8-byte quad word
5) contiguous byte allocation
6) address of initial byte

3) branch/jump
4) `call`
5) `ret`

16

16

Assembly Language Characteristics

- Minimal Data Types
  - "Integer" data of 1, 2, 4, or 8 bytes
    - Addresses (untyped pointers)
    - Data values
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- Primitive Operations
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

Instruction Set Architecture

- There used to be many ISAs
  - Motorola, x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  - More consolidated today: ARM for mobile, x86 for others
- There are even more microarchitectures
  - Apple/Samsung/Qualcomm have their own microarchitecture (implementation) of the ARM ISA
  - Intel and AMD have different microarchitectures for x86

Intel x86 Processors

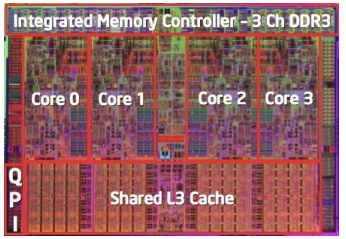
- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
– First 16-bit Intel processor. Basis for IBM PC & DOS			
– 1MB address space			
• 386	1985	275K	16-33
– First 32 bit Intel processor , referred to as IA32			
– Added "flat addressing", capable of running Unix			
• Pentium 4E	2004	125M	2800-3800
– First 64-bit Intel x86 processor, referred to as x86-64			
• Core 2	2006	291M	1060-3500
– First multi-core Intel processor			
• Core i7	2008	731M	1700-3900
– Four cores (our shark machines)			

Intel x86 Processors, cont.

- Machine Evolution
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M

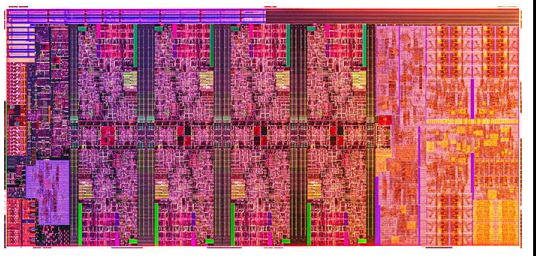


- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits
  - More cores

21

2020 State of the Art

- Tiger Lake (11th generation)
  - 10 nm, 9-15 Watts, 2-4 cores, mobile platforms
- Comet Lake (10th generation)
  - 14 nm, up to 5.3 GHz, 8 cores/16 threads, 45 Watts



https://simplecore.intel.com/newsroom/wp-content/uploads/sites/11/2020/04/Intel-10th-Gen-H-Series-2.jpg

22

x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - Getting very competitive
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
-

23

Intel's 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- All but low-end x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

24

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

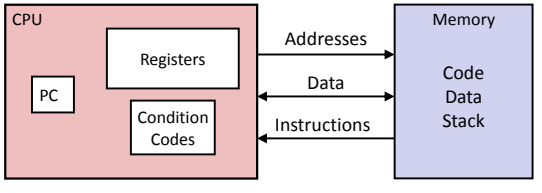
25

Definitions

- Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code
- Example ISAs:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones

26

Assembly/Machine Code View



Programmer-Visible State

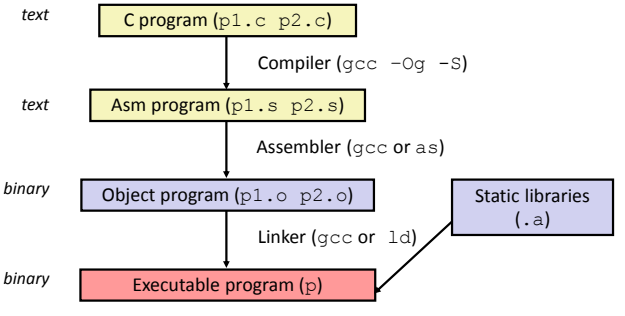
- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

27

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -Og p1.c p2.c -o p`**
  - Use basic optimizations (**`-Og`**) [New to recent versions of GCC]
  - Put resulting binary in file **`p`**

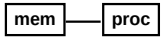


28

Abstract Machines

Machine Models

C



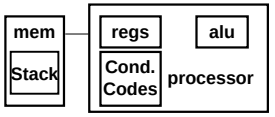
Data

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

Control

1) loops
2) conditionals
3) `switch`
4) Proc. call
5) Proc. return

Assembly



1) byte
2) 2-byte word
3) 4-byte long word
4) 8-byte quad word
5) contiguous byte allocation
6) address of initial byte

3) branch/jump
4) `call`
5) `ret`

29

29

Pointers in C

```
int a = 4;
int b = 3;
int *c;
c = &a;
b += *c;
```

30

30

Compiling Into Assembly

C Code

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
   pushq   %rbx
   movq    %rdx, %rbx
   call    plus
   movq    %rax, (%rbx)
   popq    %rbx
   ret
```

Obtain (on shark machine) with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

31

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- Code: Byte sequences encoding series of instructions
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

32

Assembly Characteristics: Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

33

Object Code

Code for `sumstore`

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address `0x0400595`

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

34

Machine Instruction Example

```
*dest = t;
```

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

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

- C Code
  - Store value **`t`** where designated by **`dest`**
- Assembly
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - **`t:`** Register **`%rax`**
    - **`dest:`** Register **`%rbx`**
    - **`*dest:`** Memory **`M[%rbx]`**
- Object Code
  - 3-byte instruction
  - Stored at address **`0x40059e`**

35

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:
  400595:  53                push   %rbx
  400596:  48 89 d3          mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff    callq  400590 <plus>
  40059e:  48 89 03          mov    %rax,(%rbx)
  4005a1:  5b                pop    %rbx
  4005a2:  c3                retq
```

- Disassembler
  `objdump -d sum`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

36

Alternate Disassembly

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- Within gdb Debugger
  - **`gdb sum`**
  - **`disassemble sumstore`**
  - – Disassemble procedure
  - **`x/14xb sumstore`**
  - – Examine the 14 bytes starting at `sumstore`

37

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

38