CSC 252: Computer Organization
Fall 2021: Lecture 5

Assembly Programming: Overview

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

A1 due tomorrow

Remember that you have 3 slip days for the semester. Email Abhishek if you want to apply any to this assignment.

A2 will be released tonight
IEEE 754 Floating Point Standard

- Single precision: 32 bits
  - 1 8-bit
  - 23-bit

- Double precision: 64 bits
  - 1 11-bit
  - 52-bit

- In C language
  - `float` single precision
  - `double` double precision
### Floating Point in C

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Bits</th>
<th>Max Value</th>
<th>Max Value (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>$2^7 - 1$</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>$2^{15} - 1$</td>
<td>32767</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>$2^{31} - 1$</td>
<td>2147483647</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>$2^{63} - 1$</td>
<td>$\approx 9.2 \times 10^{18}$</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>$(2 - 2^{-23}) \times 2^{127}$</td>
<td>$\approx 3.4 \times 10^{38}$</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>$(2 - 2^{-52}) \times 2^{1023}$</td>
<td>$\approx 1.8 \times 10^{308}$</td>
</tr>
</tbody>
</table>

**Fixed point (implicit binary point)**

- SP floating point
- DP floating point
### Floating Point in C

#### Fixed point (implicit binary point)

- **SP floating point**
- **DP floating point**

#### 32-bit Machine

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Bits</th>
<th>Max Value</th>
<th>Max Value (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>(2^7 - 1)</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>(2^{15} - 1)</td>
<td>32767</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>(2^{31} - 1)</td>
<td>(2147483647)</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>(2^{63} - 1)</td>
<td>(~9.2 \times 10^{18})</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>((2 - 2^{-23})\times2^{127})</td>
<td>(~3.4 \times 10^{38})</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>((2 - 2^{-52})\times2^{1023})</td>
<td>(~1.8 \times 10^{308})</td>
</tr>
</tbody>
</table>

- To represent \(2^{31}\) in fixed-point, you need at least 32 bits
  - Because fixed-point is a *weighted positional* representation
- In floating-point, we directly encode the exponent
  - Floating point is based on scientific notation
  - Encoding 31 only needs 7 bits in the \(exp\) field
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN

- **int → float**

```plaintext
s  exp  frac
1  8-bit  23-bit
```
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN

- **int → float**
  - Can’t guarantee exact casting. Will round according to rounding mode

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8-bit</td>
<td>23-bit</td>
</tr>
</tbody>
</table>
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN

- **int → float**
  - Can’t guarantee exact casting. Will round according to rounding mode

- **int → double**
Floating Point Conversions/Casting in C

- **double/float → int**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN

- **int → float**
  - Can’t guarantee exact casting. Will round according to rounding mode

- **int → double**
  - Exact conversion
So far in 252...
So far in 252...

C Program

Machine Code

int, float
if, else
+, -, >>

00001111
01010101
11110000
So far in 252...

C Program

Assembly Program

Machine Code

Compiler

\[\text{int, float} \]

\[\text{if, else} \]

\[+,-, \gg \]

\[\text{ret, call} \]

\[\text{fadd, add} \]

\[\text{jmp, jne} \]

\[00001111 \]

\[01010101 \]

\[11110000 \]
So far in 252...

C Program

Compiler

Assembly Program

Semantically Equivalent

int, float
if, else
+, -, >>

ret, call
fadd, add
jmp, jne

Machine Code

00001111
01010101
11110000
So far in 252...

C Program

Compiler

Assembly Program

Assembler

Machine Code

Semantically Equivalent

int, float
if, else
+, -, >>

ret, call
fadd, add
jmp, jne

00001111
01010101
11110000
So far in 252…

C Program

Compiler

Assembly Program

Assembler

Machine Code

Semantically Equivalent

int, float
if, else
+
-
>
ret,
call
fadd,
add
jmp,
jne

00001111
01010101
11110000
So far in 252…

C Program
\[\text{Compiler}\]
Assembly Program
\[\text{Assembler}\]
Machine Code
\[\text{Processor}\]

\text{int, float}
\text{if, else}
\text{+, -, >>}

\text{ret, call}
\text{fadd, add}
\text{jmp, jne}

\text{00001111}
\text{01010101}
\text{11110000}

\text{Fixed-point adder}
(e.g., ripple carry),
\text{Floating-point adder}
So far in 252…

C Program → Compiler → Assembly Program → Assembler → Machine Code → Processor → Transistor

C Program: int, float
Assembly Program: if, else, +, -, >>
Machine Code: ret, call fadd, add
Processor: jmp, jne

00001111 01010101 11110000

Fixed-point adder (e.g., ripple carry), Floating-point adder

NAND Gate NOR Gate
So far in 252...

High-Level Language

C Program

Assembly Program

Machine Code

Processor

Transistor
So far in 252...

**High-Level Language**

- C Program
- Assembly Program
- Machine Code
- Processor
- Transistor

**Instruction Set Architecture (ISA)**

- ISA: Software programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor
So far in 252...

High-Level Language

Instruction Set Architecture (ISA)

- **ISA**: Software programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor

- Processors execute machine code (binary). Assembly program is merely a text representation of machine code
So far in 252...

High-Level Language

Instruction Set Architecture (ISA)

Microarchitecture

Circuit

- **ISA**: Software programmers’ view of a computer
  - Provide all info for someone wants to write assembly/machine code
  - “Contract” between assembly/machine code and processor

- Processors execute machine code (binary). Assembly program is merely a text representation of machine code

- **Microarchitecture**: Hardware implementation of the ISA (with the help of circuit technologies)
This Module (4 Lectures)

High-Level Language

- Assembly Programming
  - Explain how various C constructs are implemented in assembly code
  - Effectively translating from C to assembly program manually
  - Helps us understand how compilers work
  - Helps us understand how assemblers work

Instruction Set Architecture (ISA)

- Microarchitecture is the topic of the next module

Microarchitecture

Circuit

- C Program
- Assembly Program
- Machine Code
- Processor
- Transistor
Today: Assembly Programming I: Basics

• Different ISAs and history behind them
• C, assembly, machine code
• Move operations (and addressing modes)
Instruction Set Architecture
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very consolidated today: ARM for mobile, x86 for others
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very 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
Instruction Set Architecture

• There used to be many ISAs
  • x86, ARM, Power/PowerPC, Sparc, MIPS, IA64, z
  • Very 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

• ISA is lucrative business: ARM’s Business Model
  • Patent the ISA, and then license the ISA
  • Every implementer pays a royalty to ARM
  • Apple/Samsung pays ARM whenever they sell a smartphone

Intel x86 ISA

- Dominate laptop/desktop/cloud market
Intel x86 ISA

• Dominate laptop/desktop/cloud market
Intel x86 ISA

- Dominate laptop/desktop/cloud market
Intel x86 ISA Evolution (Milestones)

• Evolutionary design: Added more features as time goes on
## Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on.

<table>
<thead>
<tr>
<th>Date</th>
<th>Feature</th>
<th>Notable Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>8-bit ISA</td>
<td>8080</td>
</tr>
<tr>
<td>1978</td>
<td>16-bit ISA (Basis for IBM PC &amp; DOS)</td>
<td>8086</td>
</tr>
<tr>
<td>1980</td>
<td>Add Floating Point instructions</td>
<td>8087</td>
</tr>
<tr>
<td>1985</td>
<td>32-bit ISA (Refer to as IA32)</td>
<td>386</td>
</tr>
<tr>
<td>1997</td>
<td>Add Multi-Media eXtension (MMX)</td>
<td>Pentium/MMX</td>
</tr>
<tr>
<td>1999</td>
<td>Add Streaming SIMD Extension (SSE)</td>
<td>Pentium III</td>
</tr>
<tr>
<td>2001</td>
<td>Intel’s first attempt at 64-bit ISA (IA64, failed)</td>
<td>Itanium</td>
</tr>
<tr>
<td>2004</td>
<td>Implement AMD’s 64-bit ISA (x86-64, AMD64)</td>
<td>Pentium 4E</td>
</tr>
<tr>
<td>2008</td>
<td>Add Advanced Vector Extension (AVE)</td>
<td>Core i7 Sandy Bridge</td>
</tr>
</tbody>
</table>
Intel x86 ISA Evolution (Milestones)

- Evolutionary design: Added more features as time goes on.

<table>
<thead>
<tr>
<th>Date</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>8-bit ISA</td>
</tr>
<tr>
<td>1978</td>
<td>16-bit ISA (Basis for IBM PC &amp; DOS)</td>
</tr>
<tr>
<td>1980</td>
<td>Add Floating Point instructions</td>
</tr>
<tr>
<td>1985</td>
<td>32-bit ISA (Refer to as IA32)</td>
</tr>
<tr>
<td>1997</td>
<td>Add Multi-Media eXtension (MMX)</td>
</tr>
<tr>
<td>1999</td>
<td>Add Streaming SIMD Extension (SSE)</td>
</tr>
<tr>
<td>2001</td>
<td>Intel’s first attempt at 64-bit ISA (IA64, failed)</td>
</tr>
<tr>
<td>2004</td>
<td>Implement AMD’s 64-bit ISA (x86-64, AMD64)</td>
</tr>
<tr>
<td>2008</td>
<td>Add Advanced Vector Extension (AVE)</td>
</tr>
</tbody>
</table>
Backward Compatibility

- Binary executable generated for an older processor can execute on a newer processor
- Allows legacy code to be executed on newer machines
  - Buy new machines without changing the software
- x86 is backward compatible up until 8086 (16-bit ISA)
  - i.e., an 8086 binary executable can be executed on any of today’s x86 machines
- Great for users, nasty for processor implementers
  - Every instruction you put into the ISA, you are stuck with it FOREVER
x86 Clones: Advanced Micro Devices (AMD)

• Historically
  • AMD build processors for x86 ISA
  • A little bit slower, a lot cheaper

• Then
  • Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  • Developed x86-64, their own 64-bit x86 extension to IA32
  • Built first 1 GHz CPU

• Intel felt hard to admit mistake or that AMD was better

• 2004: Intel Announces EM64T extension to IA32
  • Almost identical to x86-64!
  • Today’s 64-bit x86 ISA is basically AMD’s original proposal
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
x86 Clones: Advanced Micro Devices (AMD)

• Today: Holding up not too badly
Our Coverage

• IA32
  • The traditional x86
  • 2nd edition of the textbook

• x86-64
  • The standard
  • CSUG machine
  • 3rd edition of the textbook
  • Our focus
Moore’s Law

- More instructions typically require more transistors to implement
Moore’s Law

- More instructions typically require more transistors to implement
Moore’s Law

• More instructions typically require more transistors to implement
Moore’s Law

- More instructions require more transistors to implement
Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
Moore’s Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
Moore’s Law

• More instructions require more transistors to implement
• Gordon Moore in 1965 predicted that the number of transistors doubles every year
• In 1975 he revised the prediction to doubling every 2 years
Moore’s Law

- More instructions require more transistors to implement
- Gordon Moore in 1965 predicted that the number of transistors doubles every year
- In 1975 he revised the prediction to doubling every 2 years
- Today’s widely-known Moore’s Law: number of transistors double about every 18 months
  - Moore never used the number 18…
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • $\sim 1.4x$ smaller each dimension ($1.4^2 \sim 2$)
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension($1.4^2 \sim 2$)

• Moore’s Law is:
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • \(~1.4\times\) smaller each dimension\((1.4^2 \sim 2)\)

• Moore’s Law is:
  • A law of physics?
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics? $\textcolor{red}{No}$
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension ($1.4^2 \sim 2$)
• Moore’s Law is:
  • A law of physics? \textit{No}
  • A law of circuits?
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension\((1.4^2 \sim 2)\)

• Moore’s Law is:
  • A law of physics? \textit{No}
  • A law of circuits? \textit{No}
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension ($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics? $\text{No}$
  • A law of circuits? $\text{No}$
  • A law of economy?
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension\((1.4^2 \sim 2)\)

• Moore’s Law is:
  • A law of physics? \(\text{No}\)
  • A law of circuits? \(\text{No}\)
  • A law of economy? \(\text{Yes}\)
Moore’s Law

Transistors will stop shrinking in 2021, but Moore’s law will live on

Final semiconductor industry roadmap says the future is 3D packaging and cooling.

The first problem has been known about for a long while. Basically, starting at around the 65nm node in 2006, the economic gains from moving to smaller transistors have been slowly dribbling away. Previously, moving to a smaller node meant you could cram tons more chips onto a single silicon wafer, at a reasonably small price increase. With recent nodes like 22 or 14nm, though, there are so many additional steps required that it costs a lot more to manufacture a completed wafer—not to mention additional costs for things like package-on-package (PoP) and through-silicon vias (TSV) packaging.
Transistors will stop shrinking in 2021, but Moore’s law will live on

Final semiconductor industry roadmap says the future is 3D packaging and cooling.

The first problem has been known about for a long while. Basically, starting at around the 65nm node in 2006, the economic gains from moving to smaller transistors have been slowly dribbling away. Previously, moving to a smaller node meant you could cram tons more chips onto a single silicon wafer, at a reasonably small price increase. With recent nodes like 22 or 14nm, though, there are so many additional steps required that it costs a lot more to manufacture a completed wafer—not to mention additional costs for things like package-on-package (PoP) and through-silicon vias (TSV) packaging.
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension ($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics?  
    No
  • A law of circuits?  
    No
  • A law of economy?  
    Yes
  • A law of psychology?
Moore’s Law

• Question: why is transistor count increasing but computers are becoming smaller?
  • Because transistors are becoming smaller
  • ~1.4x smaller each dimension($1.4^2 \sim 2$)

• Moore’s Law is:
  • A law of physics? No
  • A law of circuits? No
  • A law of economy? Yes
  • A law of psychology? Yes
Today: Assembly Programming I: Basics

- Different ISAs and history behind them
- Memory, C, assembly, machine code
- Move operations (and addressing modes)
Byte-Oriented Memory Organization

- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes: byte-addressable
  - An address is like an index into that array
    - and, a pointer variable stores an address
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```
# How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>0x16</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
</tbody>
</table>

...
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
</tbody>
</table>

Memory Address:

- ... 0x16
- ...
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>random</td>
<td>0x16</td>
</tr>
</tbody>
</table>

...
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>random</td>
<td>0x16</td>
</tr>
</tbody>
</table>
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The ‘&’ operator (address-of operator) returns the memory address of a variable.
How Does Pointer Work in C???

- The content of a pointer variable is memory address.
- The `&` operator (address-of operator) returns the memory address of a variable.

<table>
<thead>
<tr>
<th>C Variable</th>
<th>Memory Content</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>0x11</td>
</tr>
<tr>
<td>c</td>
<td>0x10</td>
<td>0x16</td>
</tr>
</tbody>
</table>
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The ‘&’ operator (address-of operator) returns the memory address of a variable.
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The ‘&’ operator (address-of operator) returns the memory address of a variable.
- The ‘*’ operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)
How Does Pointer Work in C???

```c
char a = 4;
char b = 3;
char* c;
c = &a;
b += (*c);
```

- The content of a pointer variable is memory address.
- The ‘&’ operator (address-of operator) returns the memory address of a variable.
- The ‘*’ operator returns the content stored at the memory location pointed by the pointer variable (dereferencing)
Assembly Code’s View of Computer: ISA
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

CPU

Memory
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

CPU

Memory
- Code
- Data
- Stack
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

CPU

Memory
  - Code
  - Data
  - Stack
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

CPU

Memory

Code
Data
Stack

Data

0x53
0x48
0x89
0xd3
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

Instruction is the fundamental unit of work. All instructions are encoded as bits (just like data!)
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
<table>
<thead>
<tr>
<th>x86-64 Integer Register File</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
</tbody>
</table>
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 1 bytes)
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 1 bytes)
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 byte)
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 bytes)
x86-64 Integer Register File

• Lower-half of each register can be independently addressed (until 1 byte)
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer</td>
<td>8</td>
</tr>
</tbody>
</table>
x86-64 Integer Register File

- Lower-half of each register can be independently addressed (until 1 bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>Pointer</td>
<td>8</td>
</tr>
</tbody>
</table>

Floating point data is stored in a separate set of register file
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- (Byte Addressable) Memory
  - Code: instructions
  - Data
  - Stack to support function call

- Register file
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **PC: Program counter**
  - A special register containing address of next instruction
  - Called “RIP” in x86-64

---

-memory

-code

-data

-stack

-program

-counter

-address

-register

-file
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **PC: Program counter**
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **PC: Program counter**
  - A special register containing address of next instruction
  - Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

• (Byte Addressable) Memory
  • Code: instructions
  • Data
  • Stack to support function call

• Register file
  • Faster memory (e.g., 0.5 ns vs. 15 ns)
  • Small memory (e.g., 128 B vs. 16 GB)
  • Heavily used program data

• PC: Program counter
  • A special register containing address of next instruction
  • Called “RIP” in x86-64
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **CPU**
  - PC: Program counter
    - A special register containing address of next instruction
    - Called “RIP” in x86-64
  - ALU: Arithmetic logic unit (ALU)
    - Where computation happens

- **Memory**
  - Code
  - Data
  - Stack

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call
Assembly Code’s View of Computer: ISA

Assembly Programmer’s Perspective of a Computer

- **(Byte Addressable) Memory**
  - Code: instructions
  - Data
  - Stack to support function call

- **Register file**
  - Faster memory (e.g., 0.5 ns vs. 15 ns)
  - Small memory (e.g., 128 B vs. 16 GB)
  - Heavily used program data

- **PC: Program counter**
  - A special register containing address of next instruction
  - Called “RIP” in x86-64

- **Arithmetic logic unit (ALU)**
  - Where computation happens

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branch
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

- **Compute Instruction**: Perform arithmetics on register or memory data
  - `addq %eax, %ebx`
  - C constructs: +, -, >>, etc.
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

- **Compute Instruction**: Perform arithmetics on register or memory data
  - `addq %eax, %ebx`
  - C constructs: +, -, >>, etc.
- **Data Movement Instruction**: Transfer data between memory and register
  - `movq %eax, (%ebx)`

```
• Compute Instruction: Perform arithmetics on register or memory data
  • addq %eax, %ebx
  • C constructs: +, -, >>, etc.
• Data Movement Instruction: Transfer data between memory and register
  • movq %eax, (%ebx)
```
Assembly Program Instructions

Assembly Programmer’s Perspective of a Computer

• **Compute Instruction**: Perform arithmetics on register or memory data
  • `addq %eax, %ebx`
  • C constructs: +, -, >>, etc.

• **Data Movement Instruction**: Transfer data between memory and register
  • `movq %eax, (%ebx)`

• **Control Instruction**: Alter the sequence of instructions (by changing PC)
  • `jmp, call`
  • C constructs: *if-else, do-while*, function call, etc.
Turning C into Object Code

C Code (sum.c)

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

void sumstore(long x, long y,  
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```
Turning C into Object Code

C Code (sum.c)

```c
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

```assembly
sumstore:
    pushq %rbx
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    ret
```
Turning C into Object Code

C Code (sum.c)

```c
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 CSUG machine) with command

```
gcc -Og -S sum.c -o sum.s
```
Turning C into Object Code

Generated x86-64 Assembly

```assembly
sumstore:
pushq  %rbx
movq   %rdx, %rbx
call   plus
movq   %rax, (%rbx)
popq   %rbx
ret
```
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for sumstore

Memory

0x53
0x48
0x89
0xd3
0xe8
0xe8
0xf2
0xff
0xff
0xff
0x48
0x89
0x03
0x5b
0xc3
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td>0x53</td>
</tr>
<tr>
<td></td>
<td>0x48</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0xd3</td>
</tr>
<tr>
<td></td>
<td>0xe8</td>
</tr>
<tr>
<td></td>
<td>0xf2</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0xff</td>
</tr>
<tr>
<td></td>
<td>0x48</td>
</tr>
<tr>
<td></td>
<td>0x89</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>0x5b</td>
</tr>
<tr>
<td></td>
<td>0xc3</td>
</tr>
</tbody>
</table>
Turning C into Object Code

Generated x86-64 Assembly

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

Binary Code for `sumstore`

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td></td>
</tr>
<tr>
<td>0x53</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xd3</td>
<td></td>
</tr>
<tr>
<td>0xe8</td>
<td></td>
</tr>
<tr>
<td>0xf2</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0xff</td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x5b</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
</tbody>
</table>

Obtain (on CSUG machine) with command

```
gcc -c sum.s -o sum.o
```
Turning C into Object Code

Generated x86-64 Assembly

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

Obtain (on CSUG machine) with command

gcc –c sum.s -o sum.o

• Total of 14 bytes
• Instructions have variable lengths: e.g., 1, 3, or 5 bytes
• Code starts at memory address 0x0400595

Binary Code for sumstore

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0400595</td>
<td>0x53</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x48</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x89</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xd3</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xe8</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xf2</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xff</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xff</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x48</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x89</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x03</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0x5b</td>
</tr>
<tr>
<td>0x0400595</td>
<td>0xc3</td>
</tr>
</tbody>
</table>
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

Fetch Instruction (According to PC)
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC)

0x4801d8
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
  - Register File
  - Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction

addq %rax, (%rbx)
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU

Register File
Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction

Update Condition Codes
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
  - Register File
  - Condition Codes

Memory
- Code
- Data
- Stack

Addresses
- Data
- Instructions

fetch Instruction (According to PC) ➔ Decode Instruction ➔ Fetch Operands ➔ Execute Instruction ➔ Store Results

Update Condition Codes
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
  - Register File
  - Condition Codes

Memory
- Code
- Data
- Stack

Addresses
- Data
- Instructions

Fetch Instruction (According to PC) ➔ Decode Instruction ➔ Fetch Operands ➔ Execute Instruction ➔ Store Results

Update Condition Codes ➔ Adjust PC
Instruction Processing Sequence

Assembly Programmer’s Perspective of a Computer

CPU
- PC
- ALU
- Register File
- Condition Codes

Memory
- Code
- Data
- Stack

Fetch Instruction (According to PC) → Decode Instruction → Fetch Operands → Execute Instruction → Store Results

Adjust PC

Update Condition Codes