# Machine-Level Programming I: architecture, assembly and object code

**These slides adapted from materials provided by the textbook authors.**

# Machine Programming I: Basics

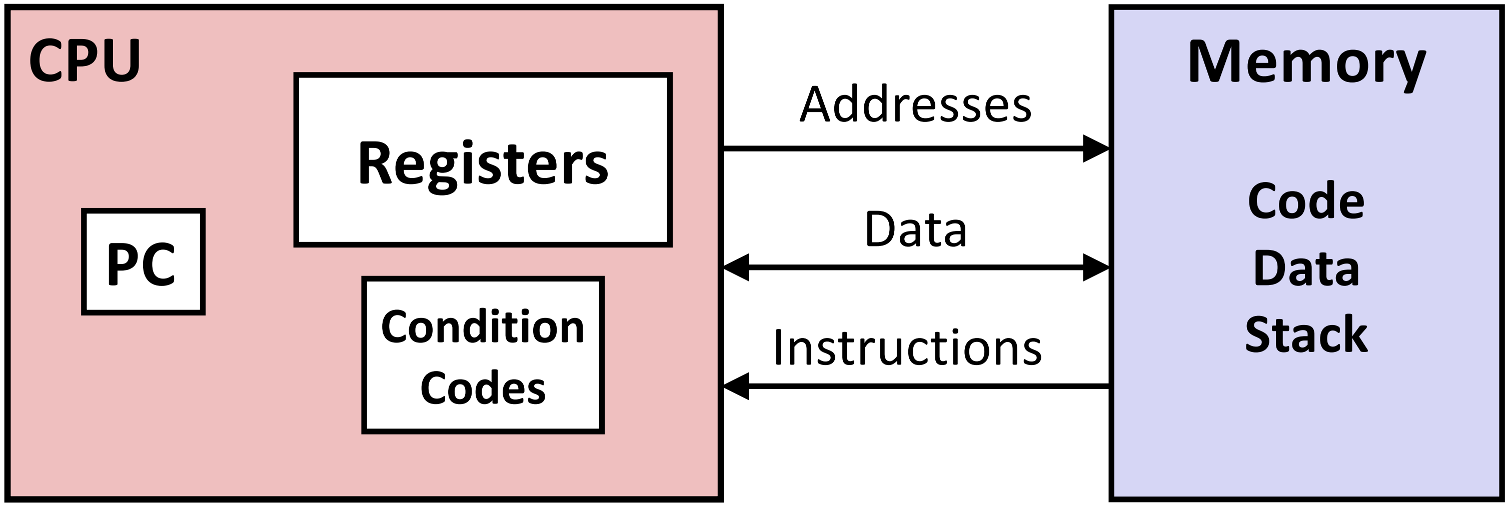
- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

# 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

# Assembly/Machine Code View



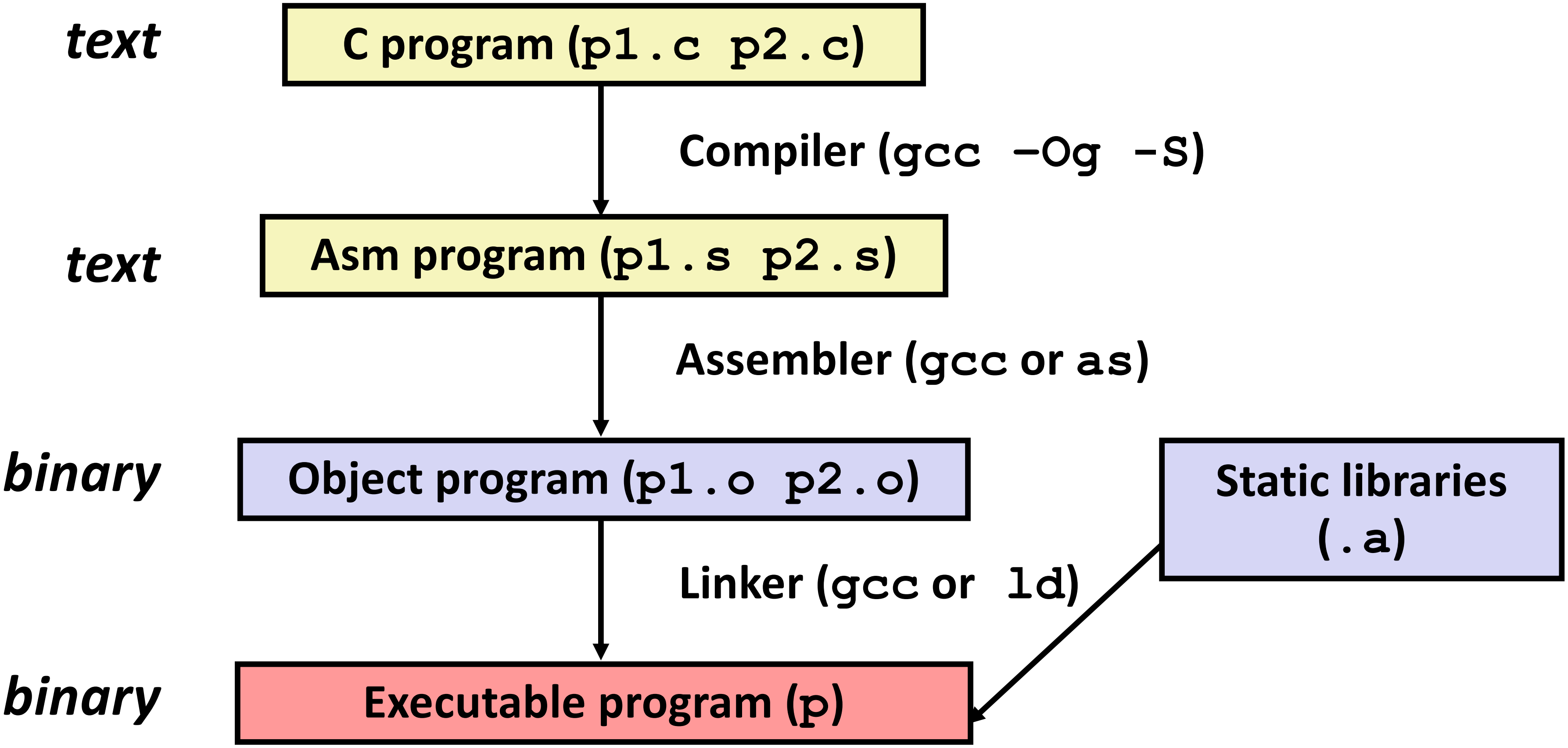
## Programmer-Visible State

- **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

# 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`



# Compiling Into Assembly

## C Code (sum.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 VM) with command

```
gcc –Og –S sum.c
```

Produces file `sum.s`

*Warning*: May get very different results on other machines, even other Linux machines, due to different versions of gcc and different compiler settings.

# Aside: Assembly ‘Syntax’

- Different ways to write down assembly.
- AT&T / GAS Syntax
  - Used by gcc.
  - Used in this course.

```
mnemonic source, destination
```

- Intel / MASM Syntax
  - Might get if you google (try your Textbook instead)
  - Doesn’t use size suffixes (ie, ‘q’)

```
mnemonic destination, source
```

# One more time, just to be clear

## In this course:

```
mnemonic source, destination
```

**So:**

```
addq %rax, %rbx
```

**Is equivalent to:**

```
%rbx = %rbx+%rax
```

# Assembly Characteristics: Data Types

- **“Integer” data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4 or 8 bytes**

- **Code: Byte sequences encoding series of instructions**

- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

# 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

# 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

# Machine Instruction Example

```
*dest = t;
```

- **C Code**
  - Store value **t** where designated by **dest**

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

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:

| | |
|---|---|
| **t**: | Register **%rax** |
| **dest**: | Register **%rbx** |
| ***dest**: | Memory **M[%rbx]** |

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

- **Object Code**
  - 3-byte instruction
  - Stored at address **0x40059e**

# Disassembling Object Code

## Disassembled

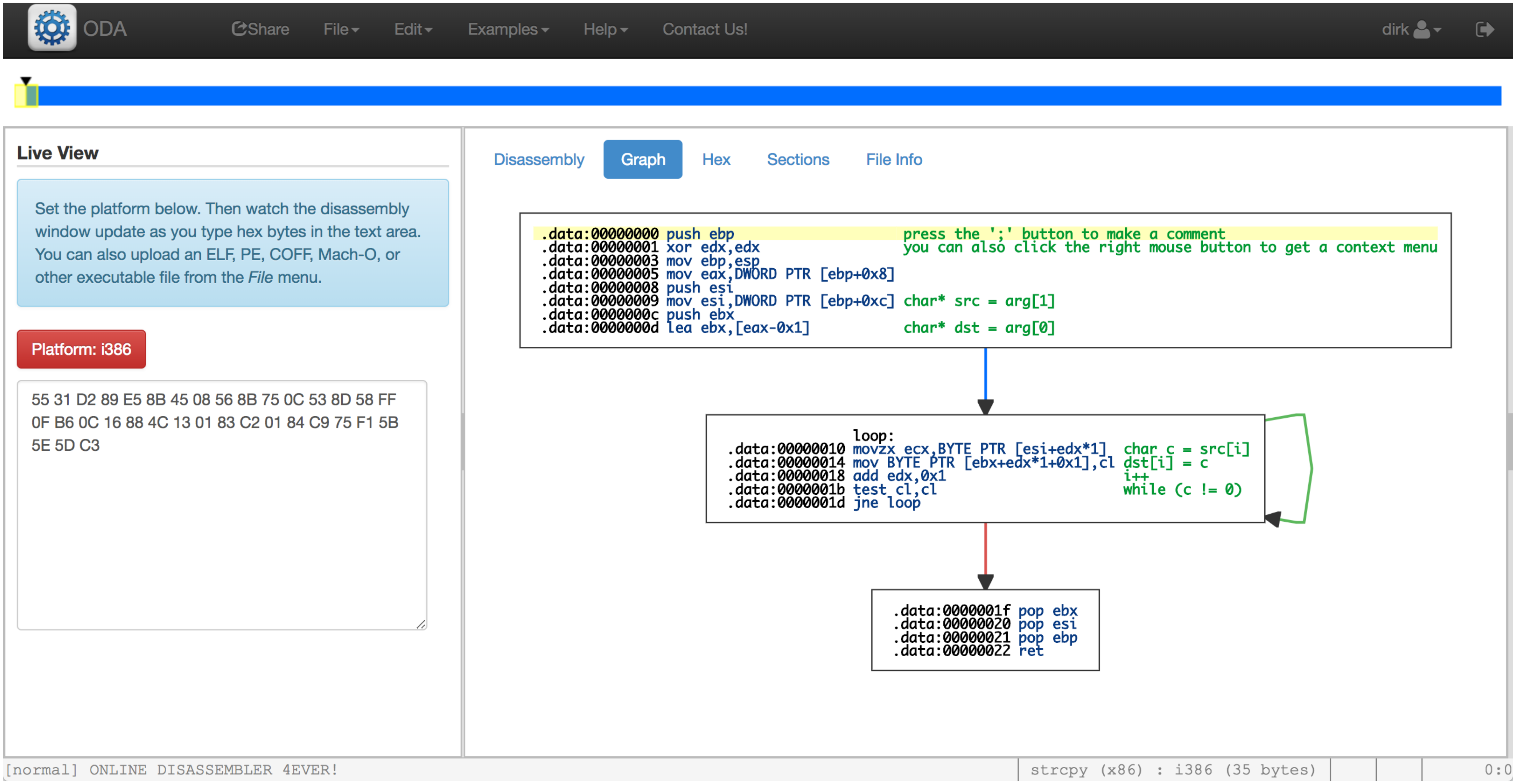
```
0000000000400595 <sumstore>:
  400595:   53                push   %rbx
  400596:   48 89 d3          mov    %rdx,%rbx
  400599:   e8 f2 ff ff ff    callq  400590 <plus>
  40059e:   48 89 03          mov    %rax,(%rbx)
  4005a1:   5b                pop    %rbx
  4005a2:   c3                retq
```

- **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

# Web-Based
# https://onlinedisassembler.com



**Live View**

Set the platform below. Then watch the disassembly window update as you type hex bytes in the text area. You can also upload an ELF, PE, COFF, Mach-O, or other executable file from the *File* menu.

```
55 31 D2 89 E5 8B 45 08 56 8B 75 0C 53 8D 58 FF
0F B6 0C 16 88 4C 13 01 83 C2 01 84 C9 75 F1 5B
5E 5D C3
```

```
.data:00000000 push ebp                          press the ';' button to make a comment
.data:00000001 xor edx,edx                       you can also click the right mouse button to get a context menu
.data:00000003 mov ebp,esp
.data:00000005 mov eax,DWORD PTR [ebp+0x8]
.data:00000008 push esi
.data:00000009 mov esi,DWORD PTR [ebp+0xc] char* src = arg[1]
.data:0000000c push ebx
.data:0000000d lea ebx,[eax-0x1]           char* dst = arg[0]
```

```
               loop:
.data:00000010 movzx ecx,BYTE PTR [esi+edx*1]  char c = src[i]
.data:00000014 mov BYTE PTR [ebx+edx*1+0x1],cl dst[i] = c
.data:00000018 add edx,0x1                     i++
.data:0000001b test cl,cl                      while (c != 0)
.data:0000001d jne loop
```

```
.data:0000001f pop ebx
.data:00000020 pop esi
.data:00000021 pop ebp
.data:00000022 ret
```

[normal] ONLINE DISASSEMBLER 4EVER!    strcpy (x86) : i386 (35 bytes)    0:0

# 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`

# 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**