Module III
Machine Language

Dr. Arijit Roy
Computer Science and Engineering Group
Indian Institute of Information Technology Sri City

The concepts are based on Carnegie Mellon University course and the text-book, authored by Patterson and Hennesy

Today: Machine Programming I: Basics

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

Intel x86 Processors

- x86 is a family of instruction set architectures initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variant.

- The 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor, with memory segmentation as a solution for addressing more memory than can be covered by a plain 16-bit address.

- The term "x86" came into being because the names of several successors to Intel's 8086 processor end in "86", including the 80186, 80286, 80386 and 80486 processors.

- Dominate laptop/desktop/server market

Intel x86 Processors

- 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.

- RISC
  - Typically, a RISC-based machine executes one instruction per clock cycle.
  - RAM is used heavily
  - Simple and standardized instructions
- CISC
  - In a CISC-based machine, each instruction performs so many actions that it takes several clock cycles.
  - RAM is used efficiently
  - Complex, and large number of instructions

Intel x86 Evolution: Milestones

Name	*Year*	*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

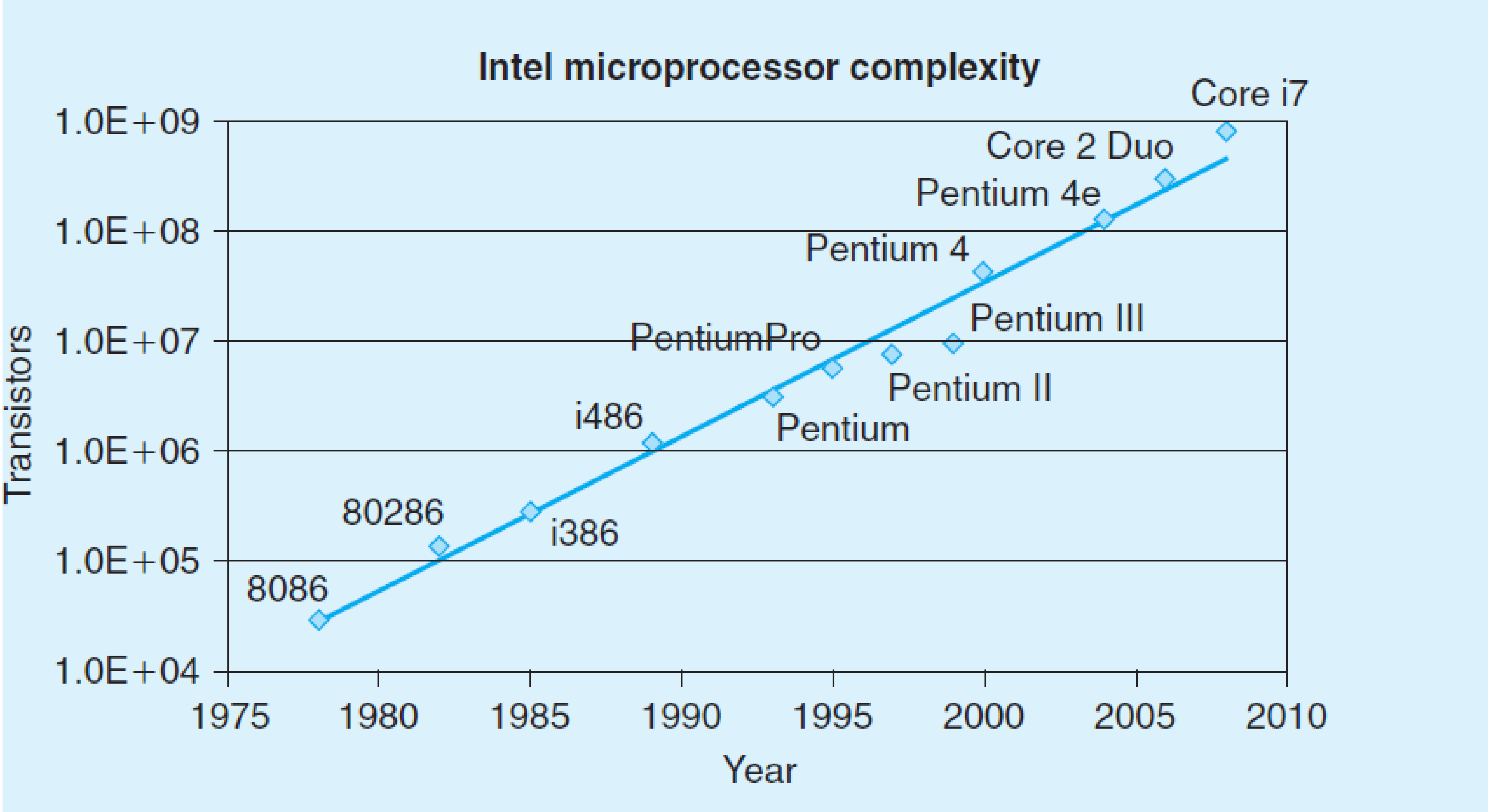
- First 64-bit Intel x86 processor, referred to as x86-64

Core 2	2006	291M	1060-3500

- First multi-core Intel processor

Core i7	2008	731M	1700-3900

- Four cores (our shark machines)



- Moore’s Law: In 1965, Gordon Moore, a founder of Intel Corporation, extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. Over more than 45 years, the semiconductor industry has been able to double transistor counts on average every 18 months.

x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - 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
- Recent Years
  - Intel got its act together
    - Leads the world in semiconductor technology
  - AMD has fallen behind
    - Relies on external semiconductor manufacturer

Opteron: AMD's x86 former server and workstation processor line

Update!!!
The scenario has changed yet again

Our Primary Focus

- The processor (CPU)…
    - datapath
    - control
- …implemented using millions of transistors
- …impossible to understand by looking at individual transistors
- we need...

Abstraction

- Delving into the depths reveals more information, but…
- **An abstraction omits “unneeded” detail, helps us cope with complexity**

- *From the figure on the right, how does abstraction help the programmer and how does she avoid too much detail?*

Hides the details of an implementation through the use of simpler abstract model

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

C compiler

Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

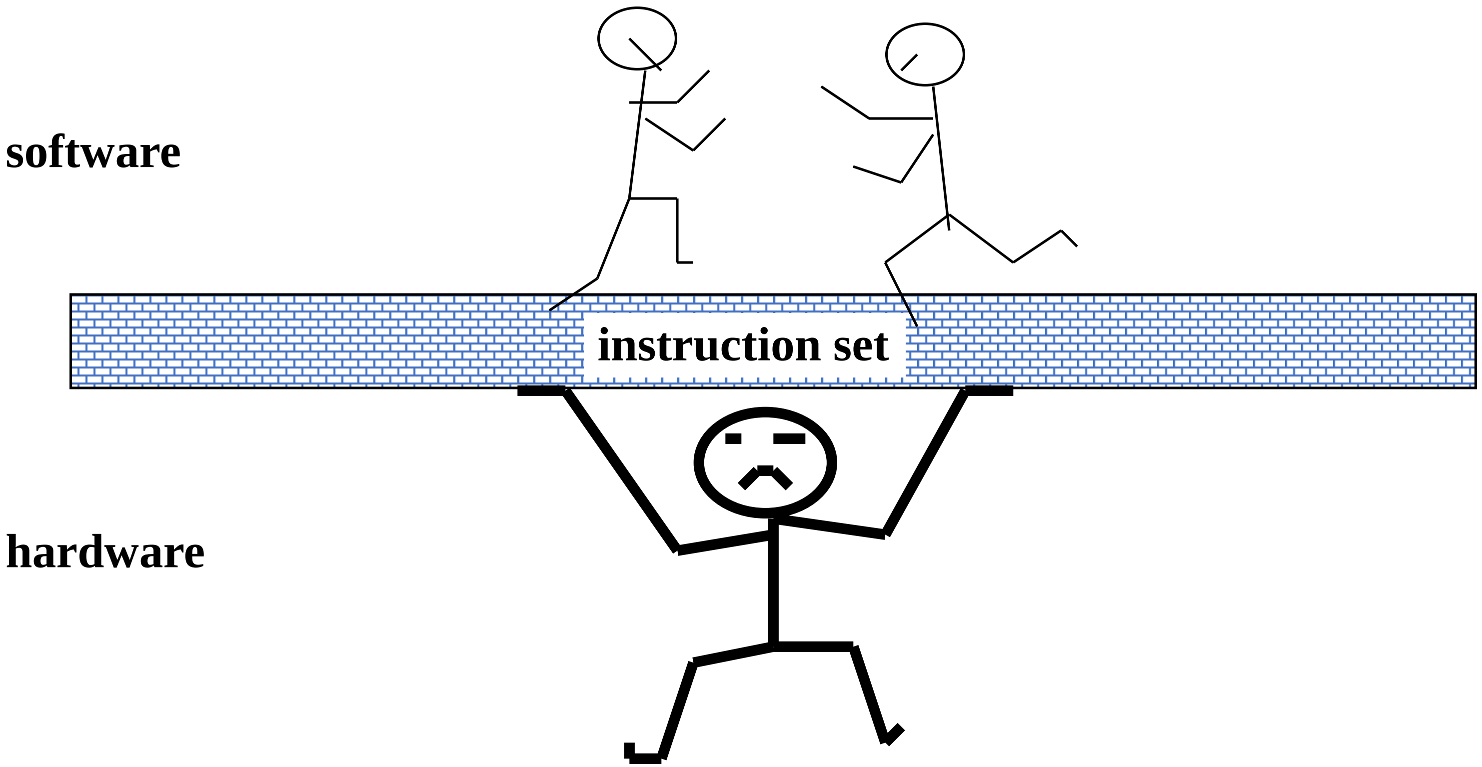
Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

The Instruction Set: a Critical Interface

How a CPU is controlled by the software



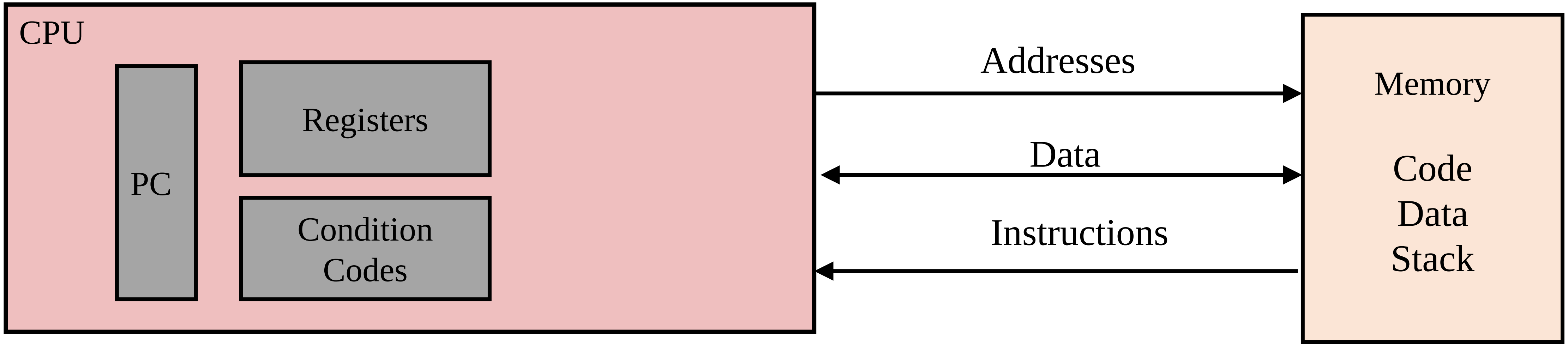
Instruction Set Architecture

- A very important abstraction:
  - *interface* between hardware and low-level software
  - *standardizes* instructions, machine language bit patterns, etc.
  - advantage: *allows different implementations of the same architecture*
- Modern instruction set architectures:
  - x86, IA32, Itanium, x86-64/Pentium/K6, PowerPC, DEC Alpha, MIPS, SPARC, HP, ARM

Format and behavior of a machine-level program is defined by instruction set architecture

Microarchitecture: Implementation of the architecture.
Examples: cache sizes and core frequency.

Assembly/Machine Code View

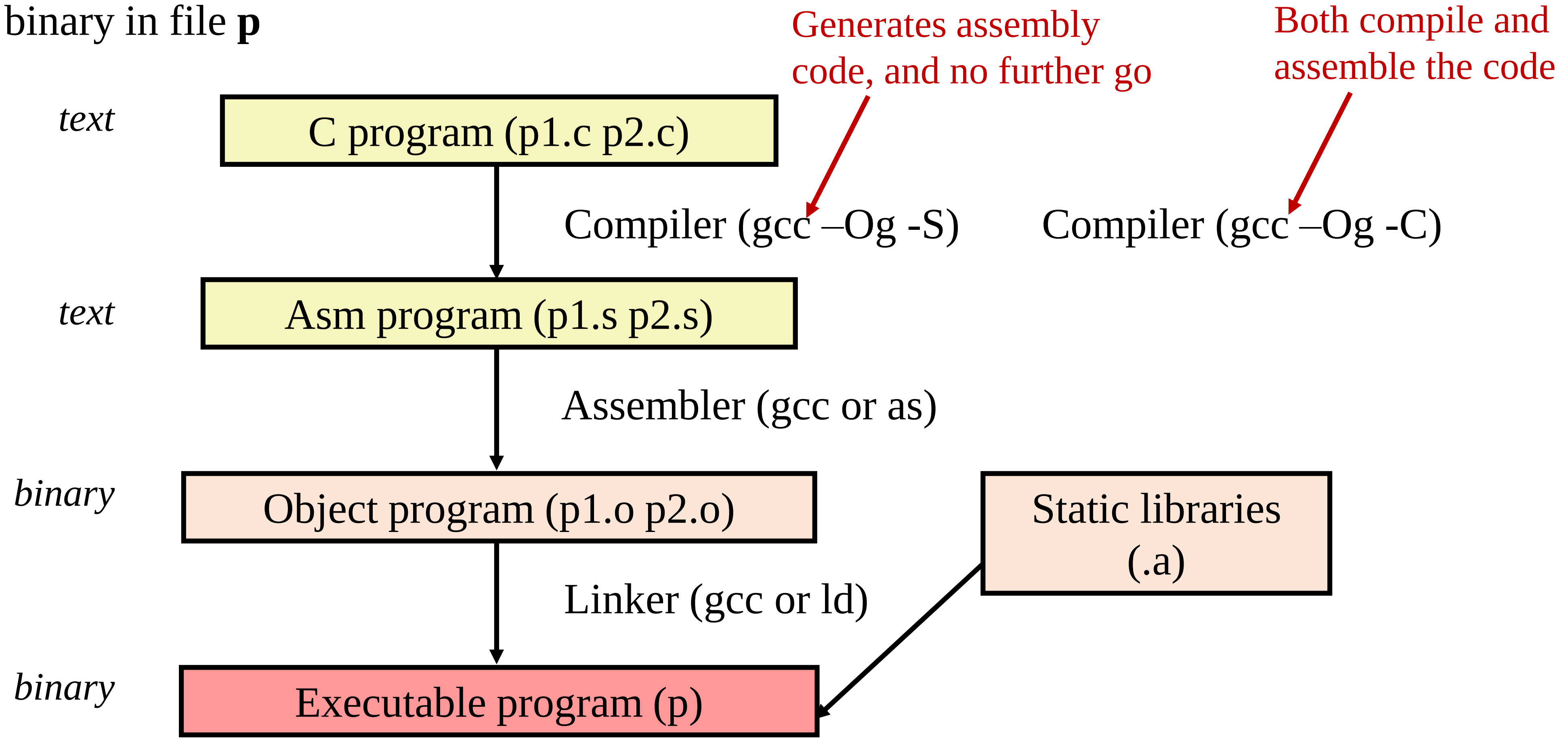


Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called %rip - “RIP” (x86-64)
- **Register file**
  - Heavily used program data 16 x 64 bits
  - Hold addresses or integer data
  - Some registers are used for holding the critical part of the program state, while others are used to hold temporary data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Vector registers**
  - Can Hold one or more integer or floating point values
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
  - No distinguishing between different datatypes, int, pointers, arrays etc.

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 with command

gcc –Og –S sum.c

Produces file sum.s

Warning: Will get very different results on different machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

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;
```

```
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**

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

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:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30 push   $0x30001090
3000100a:  68 91 dc 4c 30 push   $0x304cdc91
```

Reverse engineering forbidden by Microsoft End User License Agreement

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