Lecture 4: Assembly

Professor G. Sandoval

Some slides adapted by G. Sandoval for CS3224, from Tanenbaum & Bo, Modern Operating Systems:4th ed., (c) 2013 Prentice-Hall, Inc. All rights reserved.
Also some Slides by Brendan Dolan-Gavitt and Bryant and O'Hallaron, Computer Systems: A programmer's Perspective, Third Edition

Today: Machine Programming Basics

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

The PC

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**

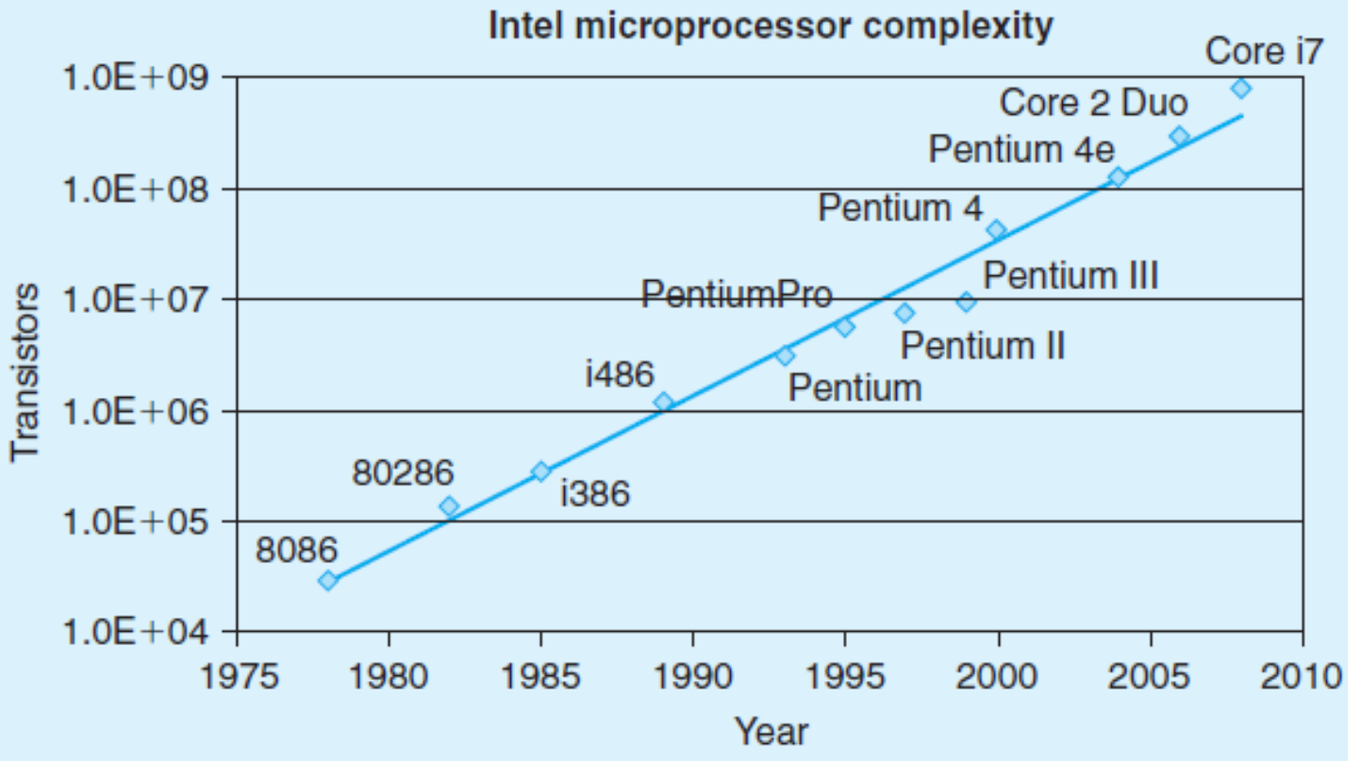
- First multi-core Intel processor

Core i7	**2008**	**731M**	**1700-3900**

- Four cores

Moore’s Law

Aside Moore’s law



If we plot the number of transistors in the different Intel processors versus the year of introduction, and use a logarithmic scale for the y-axis, we can see that the growth has been phenomenal. Fitting a line through the data, we see that the number of transistors increases at an annual rate of approximately 38%, meaning that the number of transistors doubles about every 26 months. This growth has been sustained over the multiple-decade history of x86 microprocessors.

Source: Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective. Third Edition

2015 State of the Art

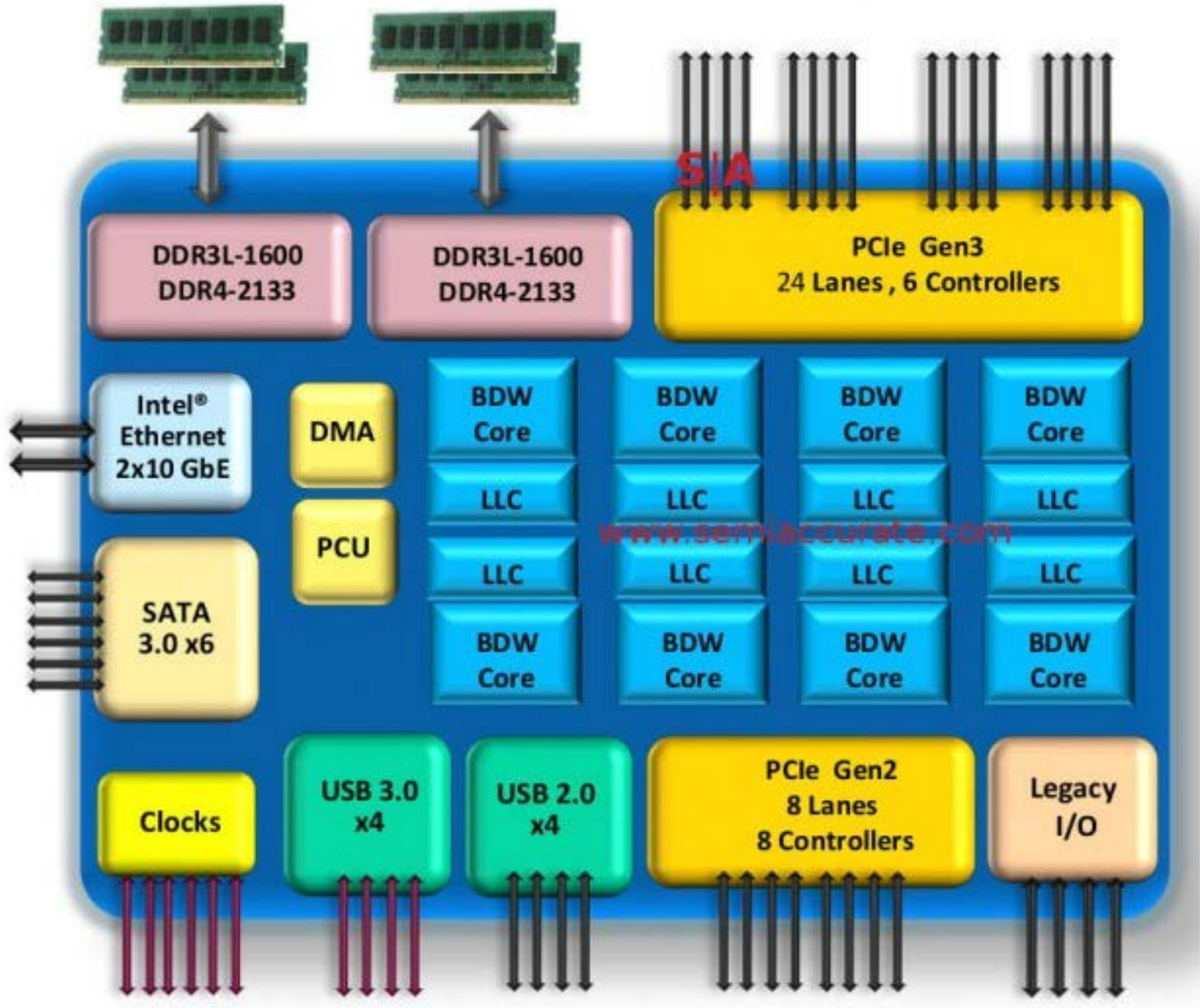
- Core i7 Broadwell 2015

Desktop Model

- 4 cores
- Integrated graphics
- 3.3-3.8 GHz
- 65W

Server Model

- 8 cores
- Integrated I/O
- 2-2.6 GHz
- 45W



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

Today: Machine Programming I: Basics

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

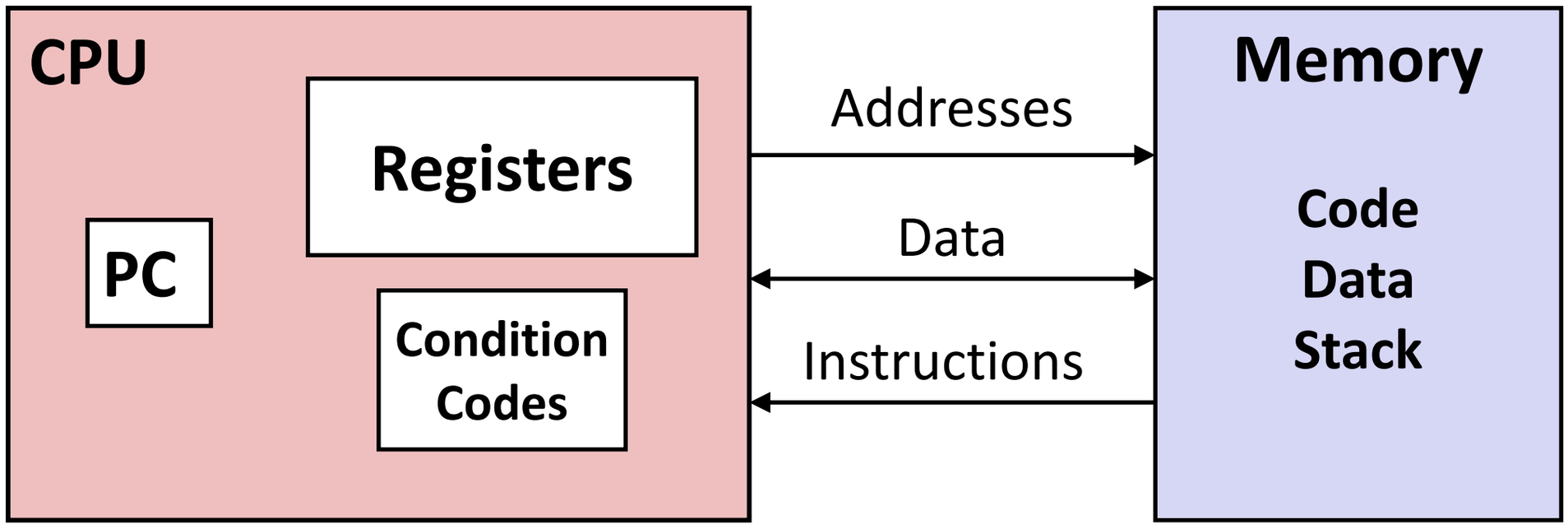
Definitions

- **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.

- **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

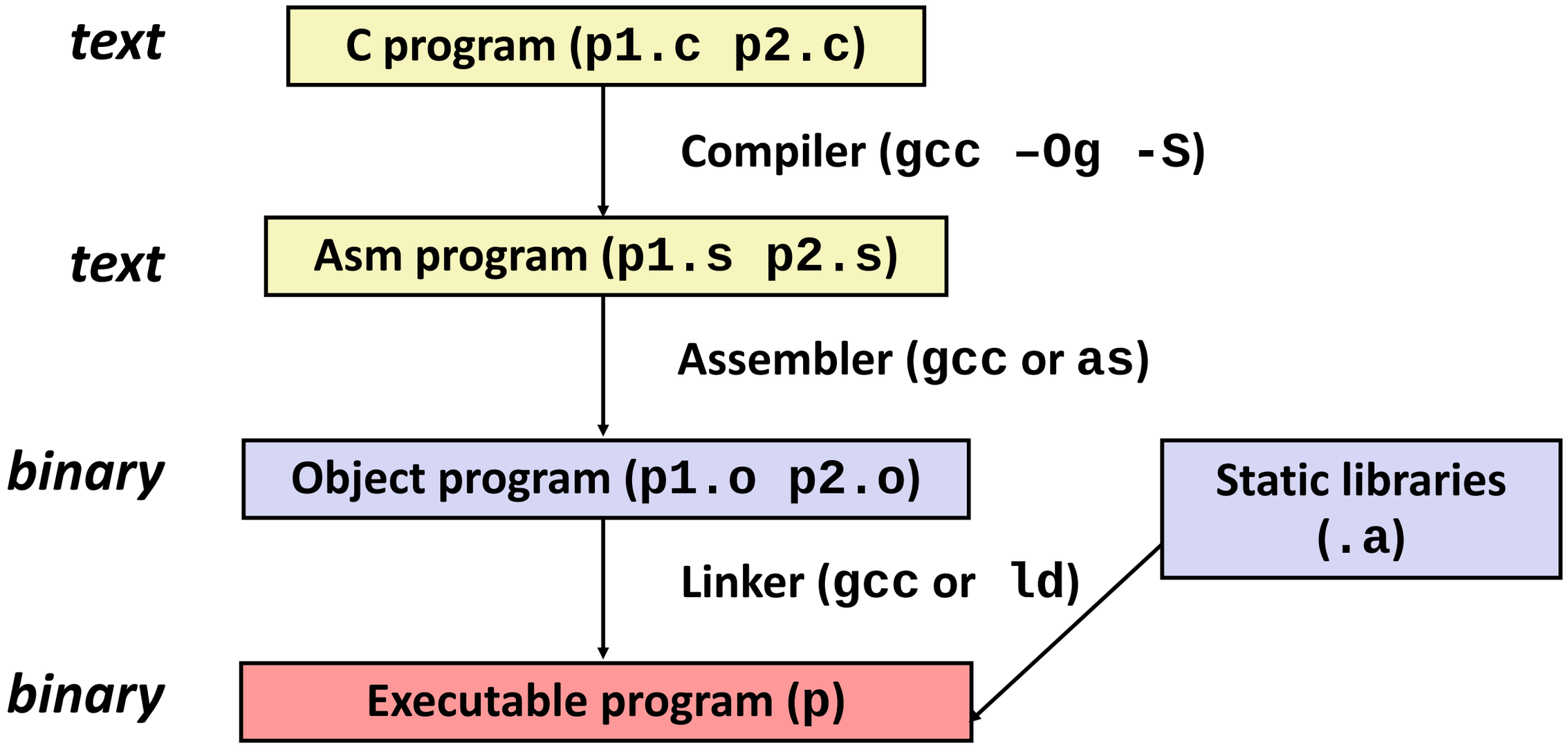


Programmer-Visible State

- **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 windows machine) with command

```
gcc –Og -fno-asynchronous-unwind-tables –S
sum.c
```

Produces file `sum.s`

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

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

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;
```

```
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:
30001001:
30001003:
30001005:
3000100a:
```

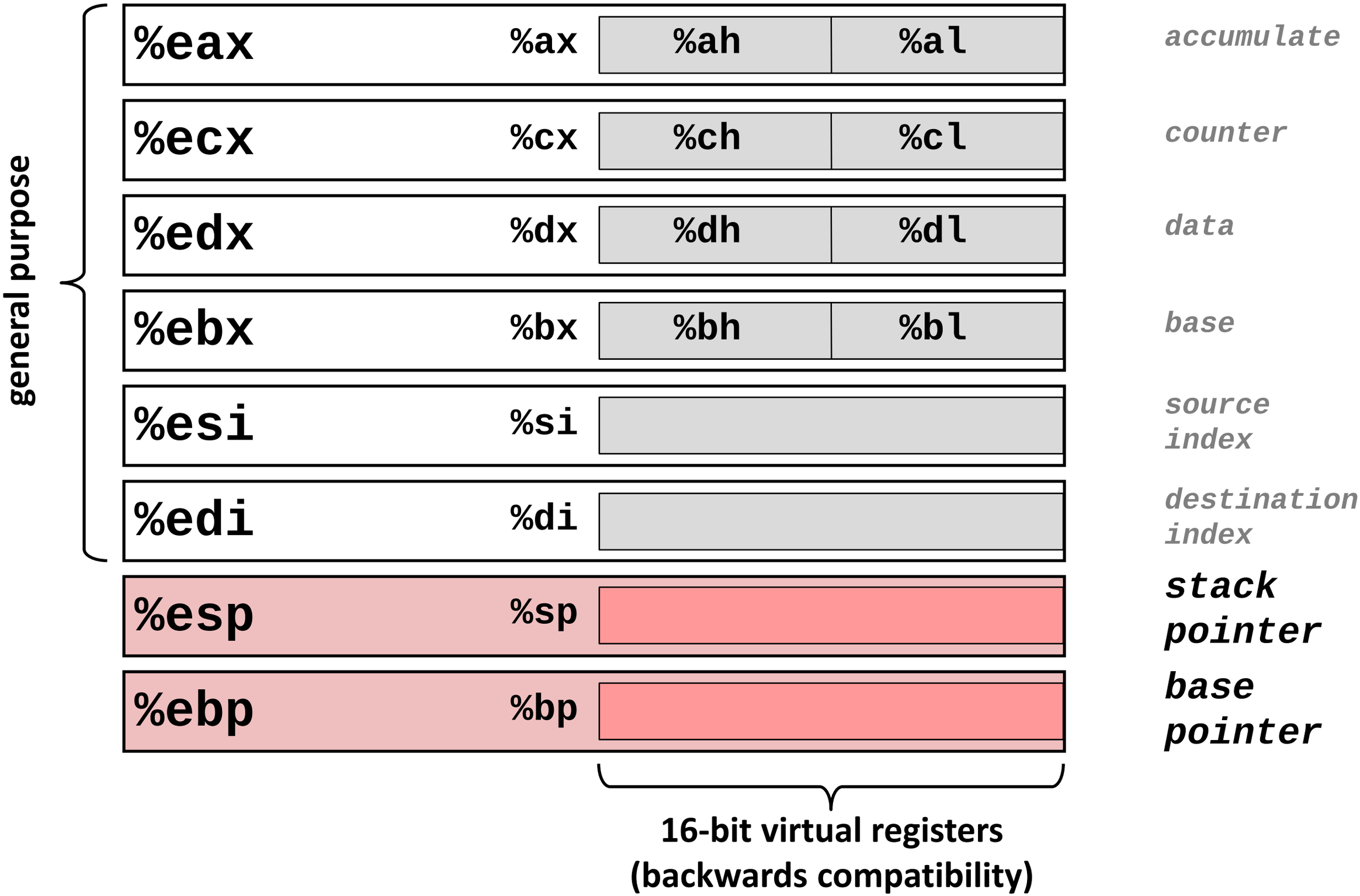
Reverse engineering forbidden by Microsoft End User License Agreement

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

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**

Some History: IA32 Registers



x86 (32-bit)

- General purpose registers:
  EAX, EBX, ECX, EDX, ESI, EDI
  (though some have historical uses: ECX is used for counters, ESI & EDI are source and destination for string operations)

- Special-purpose registers:
  EBP (base pointer), ESP (stack pointer)

- Flags register: EFLAGS

- Segment registers: CS, SS, DS, ES, FS, GS

- Instruction pointer/program counter: EIP

- Control registers: CR0-CR4

x86 Instructions

- **Instructions are *variable length***
- **Conventions:**
  - regN means a register of N bits
  - mem means memory
  - constN means a constant of N bits

Note: x86 is *BIG*

- When they stopped printing paper manuals (c. 2010), occupied Intel manual occupied 5 volumes
- Two volumes (1300 pages as of June 2015) dedicated just to describing instructions
- We will only cover a minimal subset of x86 here; if you have questions about a particular instruction the Intel manuals are actually pretty good


Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3B: System Programming Guide Part 2
3B
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 3A: System Programming Guide Part 1
3A
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 2B: Instruction Set Reference, N-Z
2B
Intel® 64 and IA-32 Architectures Software Developer 's Manual
VOLUME 2A: Instruction Set Reference, A-M
2A
Intel® 64 and IA-32 Architectures Software Developer's Manual
VOLUME 1: Basic Architecture
1

x86 Assembly Syntax

- **Two "flavors":**
  - Intel syntax (what's in the processor manual)
  - AT&T syntax (what GNU as uses)
- **The difference between them:**
  - Intel: mov eax, DWORD PTR [ebx]
  - AT&T: movl (%ebx), %eax
- **GNU as can be made to use Intel syntax instead with .intel_syntax noprefix**

Data Movement

- mov <reg>,<reg>
- mov <reg>,<mem>
- mov <mem>,<reg>
- mov <reg>,<const>
- mov <mem>,<const>

Data Movement

- **Examples (AT&T syntax)**
  - Source, dest
  - movl %eax, %edx => edx = eax
  - movl $0x123, %ebx => ebx = 0x123
  - movl 0x123, %ebx => ebx = *(int *)0x123
  - movl (%ebx), %edx => edx = *(int *)ebx
  - movl 4(%ebx), %edx => edx = *(int *)(ebx + 4)

Segmentation

- Original 8086 processor had 20 *address lines* (i.e., bits for specifying a memory address)
- But only had 16-bit registers
- Segment registers provide a way to get the extra 4 bits
- Segments also have permissions associated with them (not discussed here)
- Modern OSes make almost no use of segments (and x86_64 gets rid of them in 64-bit mode)

Segmentation

- **cs: *code segment* (modifies code fetch)**
- **ds: *data segment* (modifies data access)**
- **ss: *stack segment* (modifies ESP & EBP)**
- **es: *extra segment* (modifies string ops)**
- **So:**
  - movl (%ebp), %ebx implicitly uses the 4 bits in the ss as a prefix to %ebp

Push/Pop

- Recall: ESP is stack pointer
- On x86 the stack grows *downward* – putting something on the stack decrements the stack pointer
- Push subtracts 4 from ESP, then writes to the memory at address ESP

Arithmetic

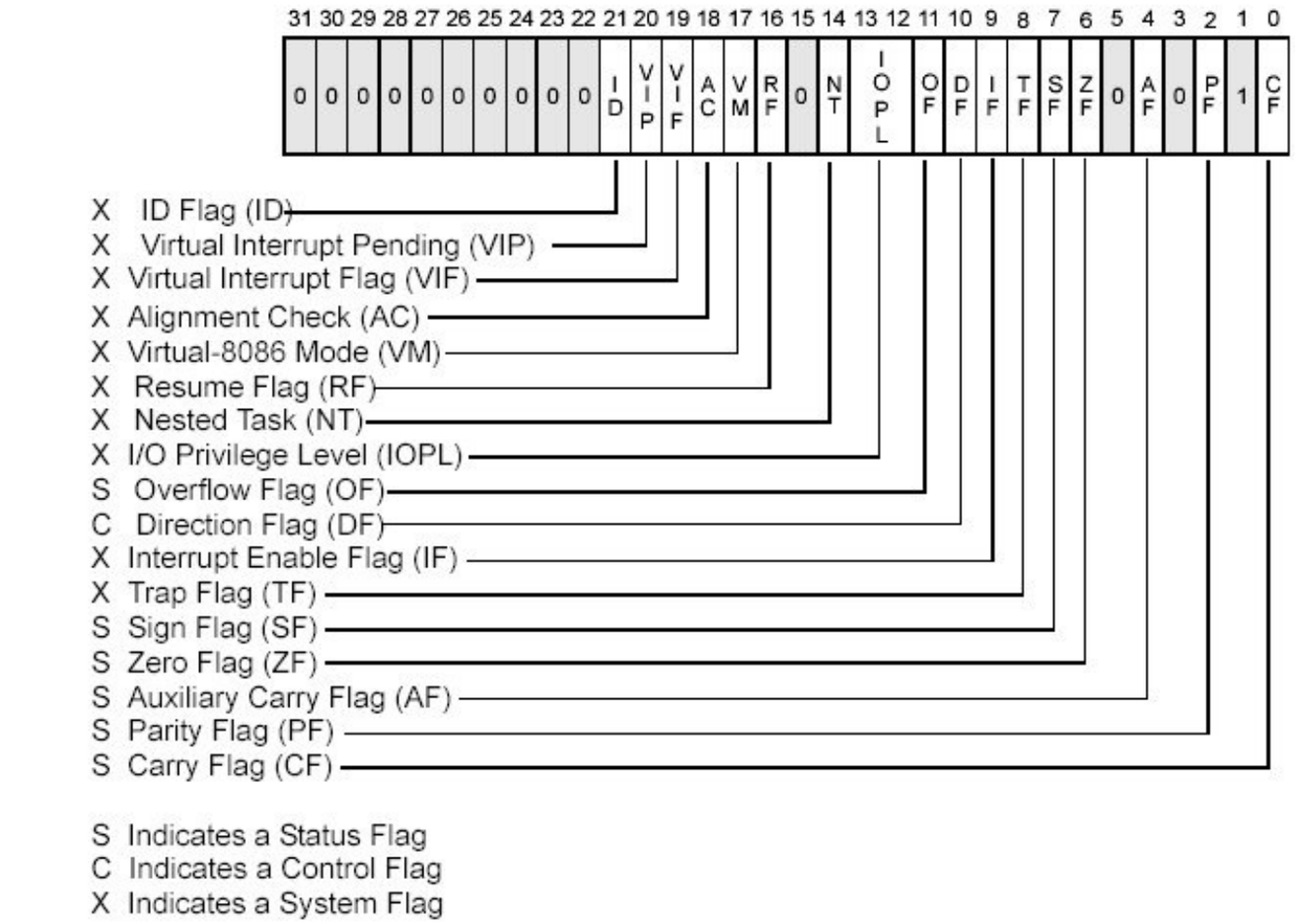
- add, sub, inc, dec
- imul, idiv
- and, or, xor
- not
- neg
- shl, shr

Logical

- test %x, %y – equivalent to and %x, %y but does not modify %y
- cmp %x, %y – equivalent to sub %x, %y but does not modify %y
- So if they don't modify the destination operand, what do they do?

EFLAGS

- **Arithmetic and logical instructions set bits in EFLAGS as a side effect**



EFLAGS Register

EFLAGS

- Conditional branches depend on the values in EFLAGS
- Also some specific instructions, e.g. cmov, will be executed only when flags are satisfied
- Other architectures take this even further: on ARM, almost *any* instruction can be made conditional

Unconditional Branch

- **jmp: unconditional jump**
  - jmp <const> – jump to constant address
  - jmp <mem> – dereference mem and jump
  - jmp <reg> - jump to address in register

Conditional Branches

Instruction	Description	signed-ness	Flags
JO	Jump if overflow		OF = 1
JNO	Jump if not overflow		OF = 0
JS	Jump if sign		SF = 1
JNS	Jump if not sign		SF = 0
JE **JZ**	Jump if equal Jump if zero		ZF = 1
JNE **JNZ**	Jump if not equal Jump if not zero		ZF = 0
JB **JNAE** **JC**	Jump if below Jump if not above or equal Jump if carry	unsigned	CF = 1
JNB **JAE** **JNC**	Jump if not below Jump if above or equal Jump if not carry	unsigned	CF = 0
JBE **JNA**	Jump if below or equal Jump if not above	unsigned	CF = 1 or ZF = 1
JA **JNBE**	Jump if above Jump if not below or equal	unsigned	CF = 0 and ZF = 0
JL **JNGE**	Jump if less	signed	SF <> OF

System

- Many instructions control system behavior
- int – executes a *software interrupt* (traps into the kernel)
- iret – return from an interrupt
- sysenter/sysret – newer way to enter/leave kernel mode

Fill in the blanks

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Operand	Value	Comment
%rax		
0x104		
$0x108		
(%rax)		
4(%rax)		