CSCI2467: Systems Programming Concepts

Slideset 5: Examining Programs at the Machine Level

Source: CS:APP Chapter 3, Bryant & O'Hallaron

Course Instructors:

Matthew Toups
Caitlin Boyce

Course Assistants:

Saroj Duwal
David McDonald

Spring 2020

THE UNIVERSITY of NEW ORLEANS

DEPARTMENT OF COMPUTER SCIENCE

Announcements

- copying code or comments *without citation* from websites/classmates/github/stackexchange/etc is plagiarism
- copying with a citation but not explained in your own words will receive no credit, but may save you from disciplinary proceedings

datalab is done!

- Bomblab begins!

Announcements

- Bomblab writeup passed out today (due Thursday September 26)
- don't explode your bomb!
- Scoreboard on AutoLab is automatically updated (no handing in)

Intel x86 Processors

- Dominate laptop/desktop/server market today
  (but not mobile)
- Evolutionary design
- Backwards compatible all the way back to 8086 (1978)
- Added more features over time

Intel x86 Processors

- Complex instruction set computer (CISC)
- Many different instructions with many different formats
- ... but, only small subset encountered with most programs
- Hard to match performance of Reduced Instruction Set Computer (RISC)
- ... but Intel has done just that

  (in terms of speed, less so for low power)

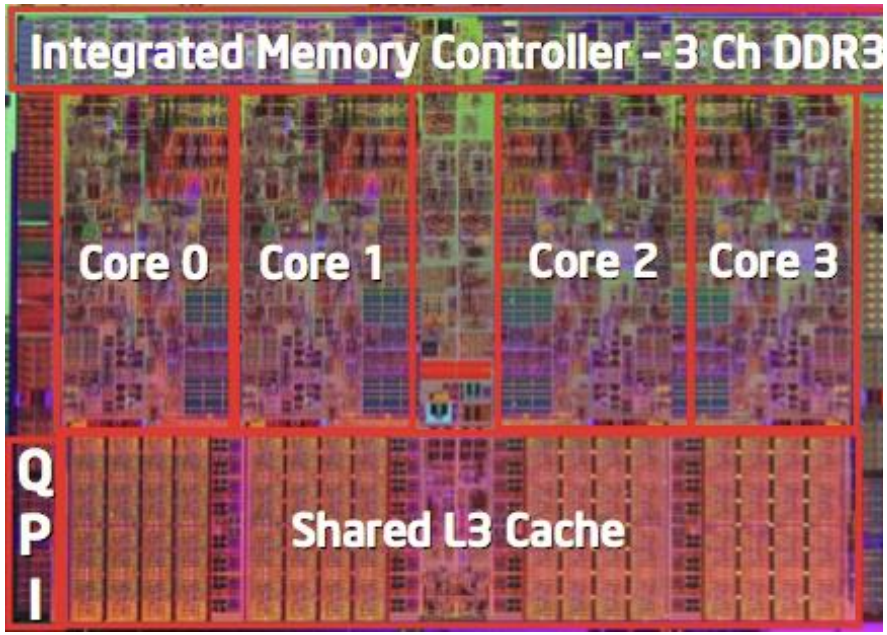
Intel x86 Processors

Name	Date	Transistors	MHz	Notes
8086	1978	29k	5-10	first 16-bit Intel CPU, basis for IBM PC & DOS. 1MB address space
386	1985	275k	16-33	first 32-bit Intel CPU, referred to as IA32. Added "flat addressing" – capable of running Unix OSes
Pentium 4E	2004	125M	2800-3800	First 64-bit Intel x86 CPU (x86-64)
Core 2	2006	291M	1060-3500	First multi-core Intel CPU
Core i7	2008	731M	1700-3900	4 cores per CPU
Xeon E5-2697v2	2013	4.3B	2700	12 cores per CPU
Xeon E5-2699v4	2016	7.2B	2200	22 cores per CPU

Intel x86 processor evolution

- **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	2000	42M
Core 2 Duo	2006	291M
Core i7	2008	731M
Core i7 Skylake	2015	1.9B



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

Intel x86 processors, process technology

- **Past Generations**

		Process technology
1st Pentium Pro	1995	600 nm
1st Pentium III	1999	250 nm
1st Pentium 4	2000	180 nm
1st Core 2 Duo	2006	65 nm

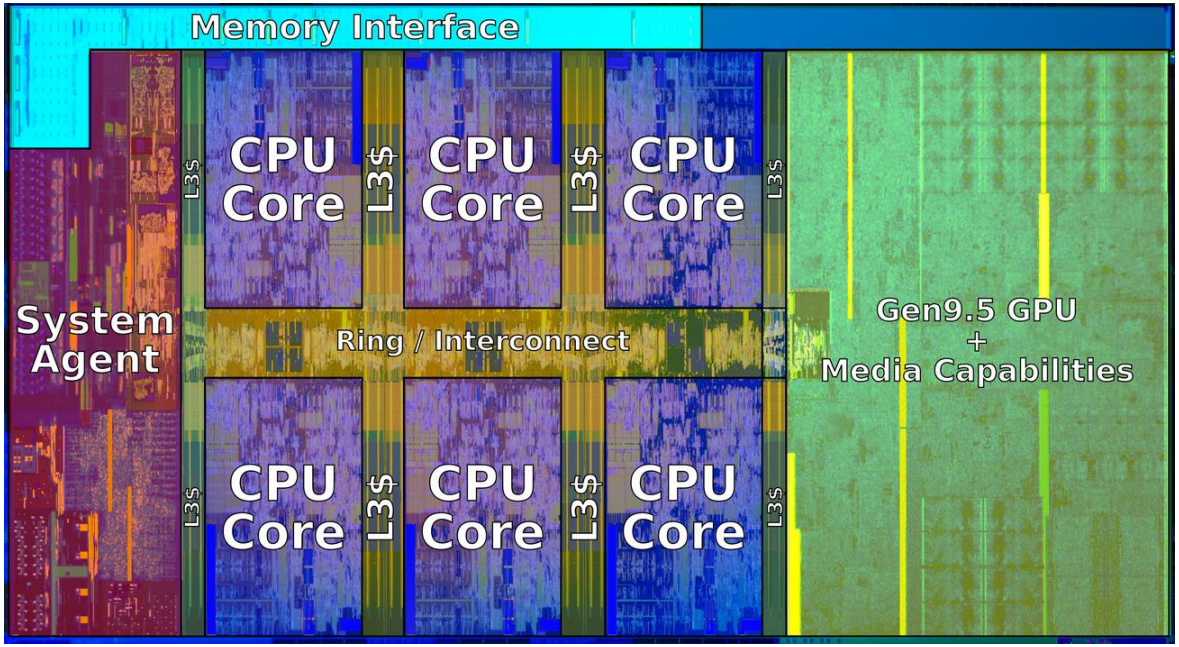
- **Recent & Upcoming Generations**

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
-	Cannon Lake	2019?	10 nm

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

2018 CPU State of the Art

Intel "Coffee Lake" microarchitecture



- **Mobile Model: Core i7**
  - 2.2-3.2 GHz
  - 45 W
- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W
- **Server Model: Xeon**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

x86 clones: Advanced Micro Devices (AMD)

- Historically AMD has followed just behind Intel
  - a little slower, a lot cheaper
- Then in early 2000s ...
  - AMD recruited top designers from Digital Equipment Corp and other defunct CPU makers
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64 extension (64 bit x86)
- In recent Years...
  - Intel got its act together, retook the lead
  - AMD returned to 2nd place . . . until recently?

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 (also known as 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 amd64!
- **Since then**: all but low-end x86 CPUs support x86-64
  - but lots of code still runs in 32-bit mode
  - 32-bit CPUs still very widely used (embedded, mobile)

Coverage in CSCI2467

- x86-64 is now standard
- CS:APP 3rd edition focuses on x86-64
- (web asides on IA32 available)
- We will only cover x86-64

  (extension of x86, will be easy for 2450 students to pick up)

Some definitions

- Architecture (also ISA: instruction set architecture)

  The parts of a processor design that one needs to understand to write assembly/machine code.

  - Examples: instruction set specification, registers

- Microarchitecture

  Implementation of the architecture.

  - Examples: cache sizes and core frequency

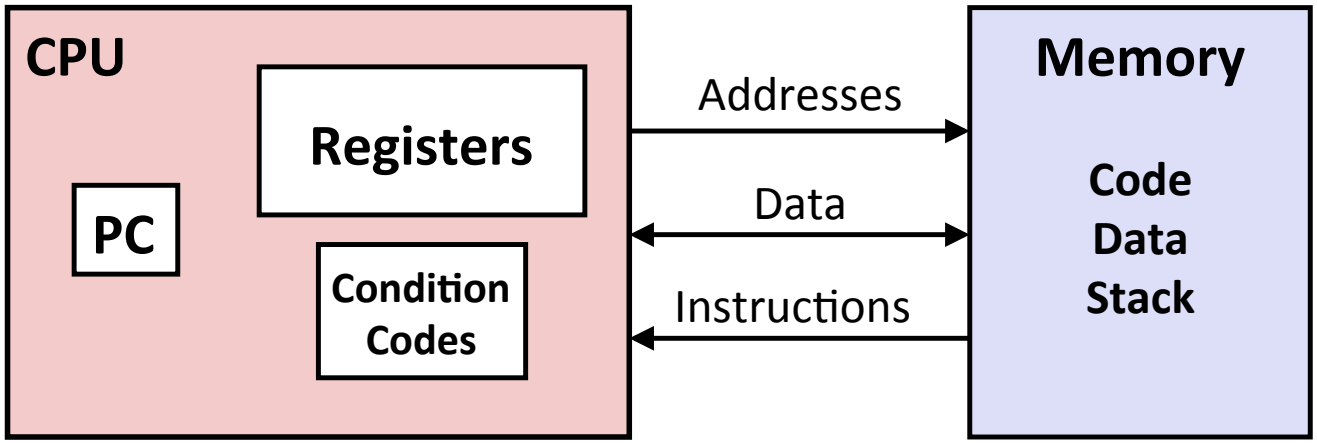
- Code forms:

  - Machine code: byte-level programs that a processor executes
  - Assembly code: a text representation of machine code

Some definitions

- Example Instruction Set Architectures (ISAs):
- Intel:
  x86 (IA32)
  Itanium (64-bit, never mass produced)
  x86-64 (64-bit, created by AMD and copied by Intel)
- ARM (Acorn RISC Machine):
  used in almost all mobile phones
  designed for low power consumption
- RISC V (origins: UC Berkeley)
  New open-source ISA

Assembly/Machine Code view



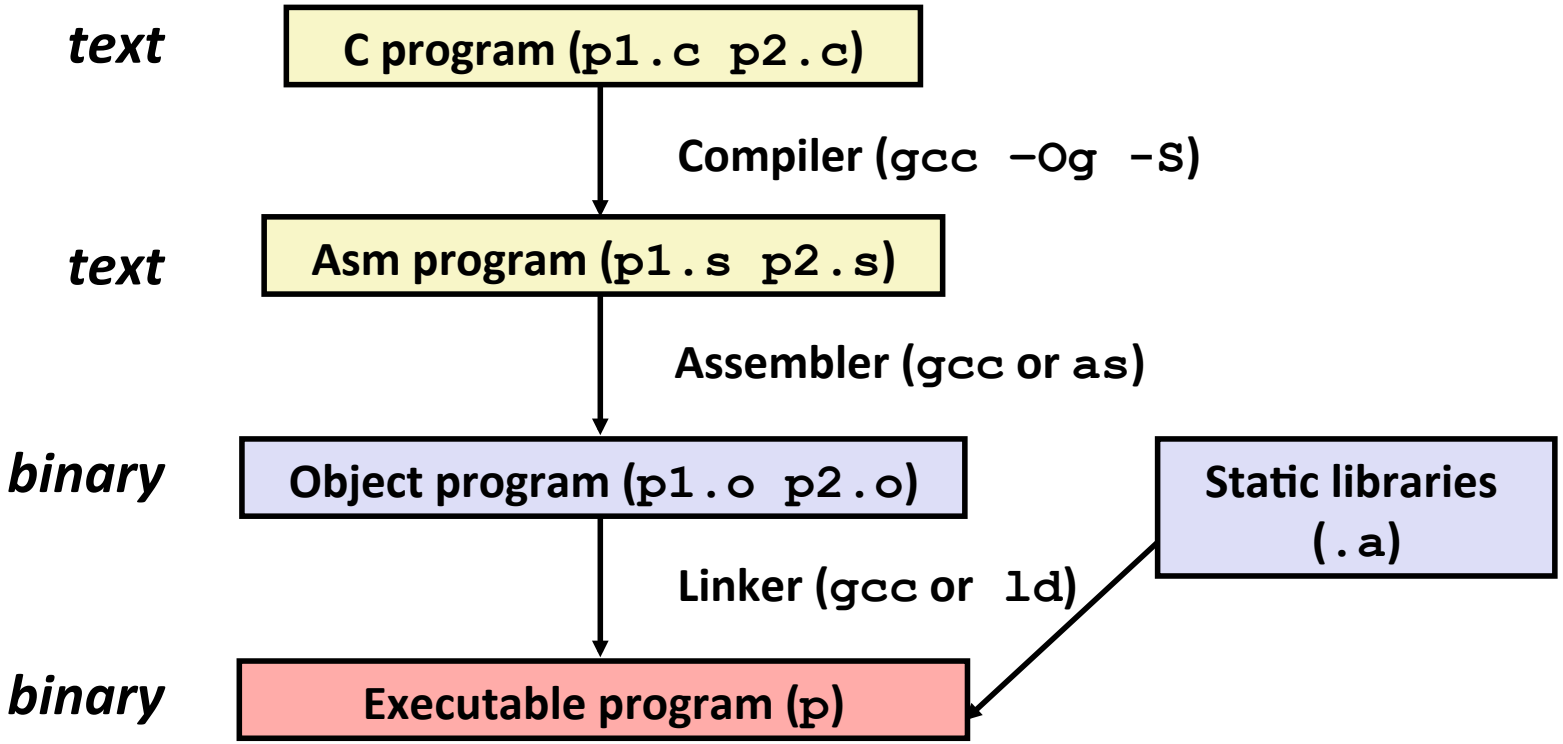
Programmer-Visible State

- **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

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

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

```
sumstore:
   push    rbx
   mov     rbx, rdx
   call    plus
   mov  QWORD PTR [rbx], rax
   pop     rbx
   ret
```

Using:

`gcc -Og -S sum.c -masm=intel`

Will get very different results on other systems: Mac OS X, Windows, other compilers, even gcc with other flags

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

Operations are assembly instructions, which can:

- 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 (binary)

```
400532:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- Assembler:
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly-complete image of executable program
  - missing linkages between code in different files
- Linker:
  - resolves references between files
  - combined with static run-time libraries (e.g. `printf`)
  - some libraries are *dynamically linked* (linking occurs when program begins execution)

Disassembling object code

Disassembler: `objdump -M intel -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

```
0000000000400532 <sumstore>:
  400532:   53                      push   rbx
  400533:   48 89 d3                mov    rbx,rdx
  400536:   e8 f2 ff ff ff          call   40052d <plus>
  40053b:   48 89 03                mov    QWORD PTR [rbx],rax
  40053e:   5b                      pop    rbx
  40053f:   c3                      ret
```

Machine instruction example

`*dest = t;`	C code: store value `t` where designated by `dest`

`mov QWORD PTR [rbx],rax`	Assembly:
	• move 8-byte value to memory ("quad word")
	• Operands:
	`t`: register `rax`
	`dest`: register `rbx`
	`*dest`: memory at [rbx]

`40053b: 48 89 03`	Object code: 3-byte instruction stored at address 0x40053b

Disassembly with gdb

gdb commands:

```
(gdb) disassemble sumstore
```

```
Dump of assembler code for function sumstore:
 0x0000000000400532 <+0>:     push   rbx
 0x0000000000400533 <+1>:     mov    rbx,rdx
 0x0000000000400536 <+4>:     call   0x40052d <plus>
 0x000000000040053b <+9>:     mov    QWORD PTR [rbx],rax
 0x000000000040053e <+12>:    pop    rbx
 0x000000000040053f <+13>:    ret
```

Examine the 14 bytes starting at location `sumstore`:

```
(gdb) x/14xb sumstore
```

```
0x400532 <sumstore>:    0x53    0x48    0x89    0xd3    0xe8    0xf2    0xff    0xff
0x40053a <sumstore+8>:  0xff    0x48    0x89    0x03    0x5b    0xc3
```

GDB

Crucial tool for our labs! See CSAPP3e for examples (p.280 Figure 3.39)

Command	Effect
Starting and Stopping	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
Breakpoints	
`break sum`	Set breakpoint at entry to function `sum`
`break *0x80483c3`	Set breakpoint at address `0x80483c3`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas sum`	Disassemble function sum
`disas 0x80483b7`	Disassemble function around address `0x80483b7`
`disas 0x80483b7 0x80483c7`	Disassemble code within specified address range
`print /x $eip`	Print program counter in hex
Examining data	
`print $eax`	Print contents of `%eax` in decimal
`print /x $eax`	Print contents of `%eax` in hex
`print /t $eax`	Print contents of `%eax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of `%ebp` plus 8 in hex
`print *(int *) 0xbffff890`	Print integer at address `0xbffff890`
`print *(int *) ($ebp+8)`	Print integer at address `%ebp` + 8
`x/2w 0xbffff890`	Examine two (4-byte) words starting at address `0xbffff890`
`x/20b sum`	Examine first 20 bytes of function `sum`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.26: **Example GDB Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

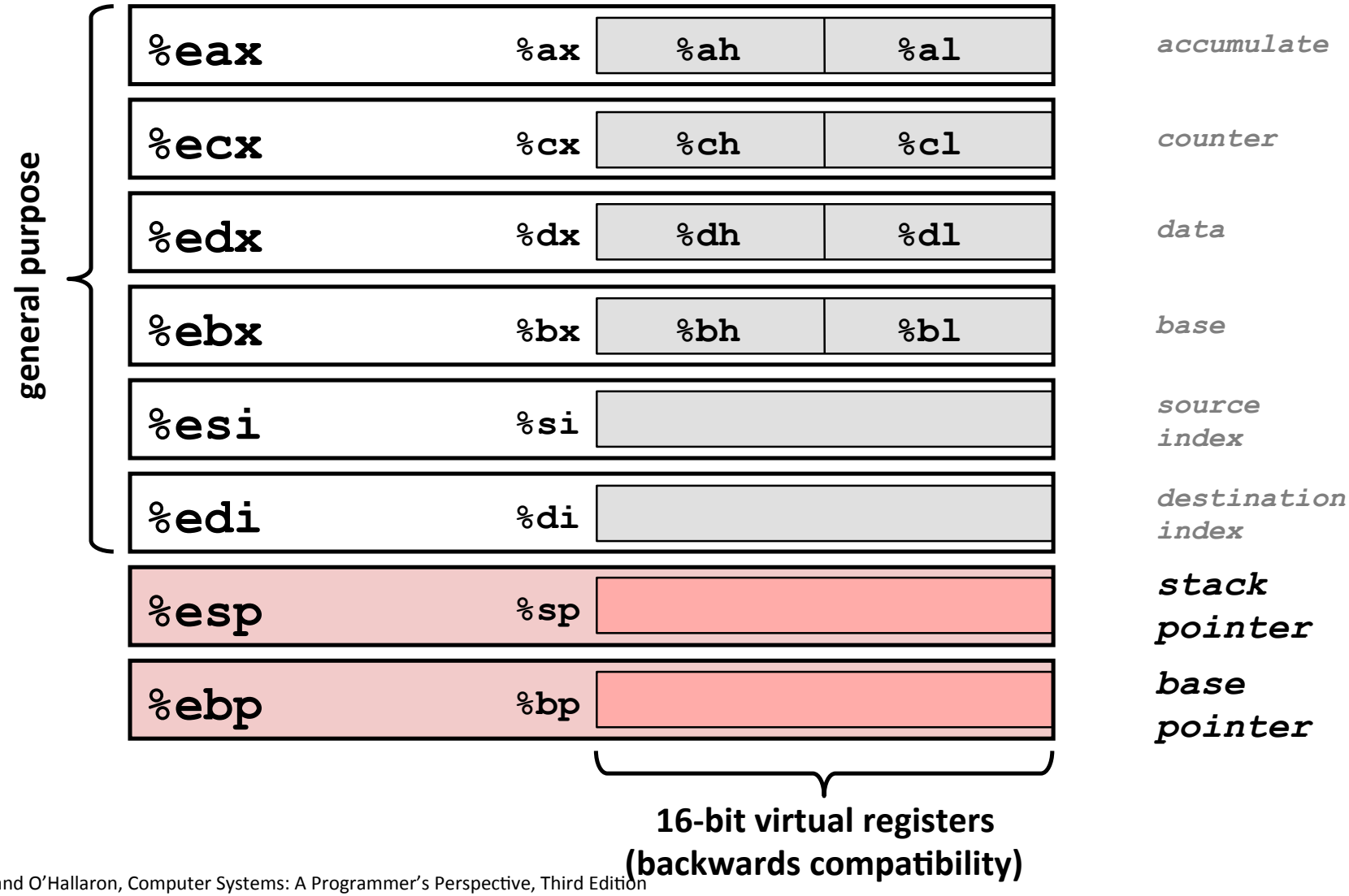
What can be disassembled?

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

```
% 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

Legacy: IA32 (x86) registers



and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

x86-64 integer registers

introducing: 64-bits wide, and 8 additional general purpose registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Arithemetic operations
Two operand instructions

- Pay attention to order of operands
- No distinction between signed & unsigned. (why not?)

Format	Operands	Computation	
`add`	dest,src	dest = dest + src	
`sub`	dest,src	dest = dest - src	
`imul`	dest,src	dest = dest * src	
`sal`	dest,src	dest = dest << src	(also `shl`)
`sar`	dest,src	dest = dest >> src	(arithmetic)
`shr`	dest,src	dest = dest >> src	(logical)
`xor`	dest,src	dest = dest ^ src	
`and`	dest,src	dest = dest & src	
`or`	dest,src	dest = dest | src	

Arithemetic operations
One operand instructions

Format	Operand	Computation
`inc`	dest	dest = dest + 1
`dec`	dest	dest = dest - 1
`neg`	dest	dest = – dest
`not`	dest	dest = ~dest

- See CSAPP3e for more on these operations.

An arithmetic example

How would we write this in x86-64 assembly?

Assume x is stored in register `rdi`:

```
long m12(long x)
{
  return x*12;
}
```

Perhaps this?

```
imul rax, rdi, 12
```

Nope: This is not how a compiler "thinks"!

Address computation instruction

- `lea dst src`
  - `src` is address mode expression
  - set `dst` to address denoted by expression
- Uses:
  - Computing addresses without a memory reference
    (e.g. translation of p = & x [i];)
  - Computing arithmetic expressions of the form $x + k * y$
    k = 1, 2, 4, or 8

```
lea     rax, [rdi+rdi*2]      # rax <- x+x*2
sal     rax, 2                # rax <- rax<<2
```

Address computation instruction

- Compilers *love* lea instruction
- Fast way to compute $x + k * y$ and similar

```
long m12(long x)
{
  return x*12;
}
```

Compiles to:

```
m12:
  lea     rax, [rdi+rdi*2]      # rax <- x+x*2
  sal     rax, 2                # rax <- rax<<2
```

Arithmetic expression example

```
long arith
(long x,long y,long z)
{
   long t1 = x+y;
   long t2 = z+t1;
   long t3 = x+4;
   long t4 = y * 48;
   long t5 = t3 + t4;
   long rval = t2 * t5;
   return rval;
}
```

```
arith:
 lea    rax, [rdi+rsi]
 add    rax, rdx
 lea    rcx, [rsi+rsi*2]
 sal    rcx, 4
 lea    rcx, [rdi+4+rcx]
 imul   rax, rcx
 ret
```

Interesting instructions:

- lea: address computation
- sal: shift left
- imul: multiplication

  (only used once!)

```
arith:
 lea    rax, [rdi+rsi]   # t1
```

Machine Programming I: Summary

- History of Intel processors and architectures
- Evolutionary design leads to many quirks and artifacts
- C, assembly, machine instructions
- New forms of visible state: program counter, registers, ...
- Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly basics: registers, operands, move
- Arithmetic
- C compiler will figure out different instruction combinations to carry out computation

Pointers in C

C Code

```
int x;
int *p;

x = 99;    //holds a value
p = &x;    //holds an address of a value
```

Operator	Function
&	"address of"

Pointers in C

C Code

```
int x;
int *p;

x = 99;    //holds a value
p = &x;    //holds an address of a value
```


Memory

C pointer syntax

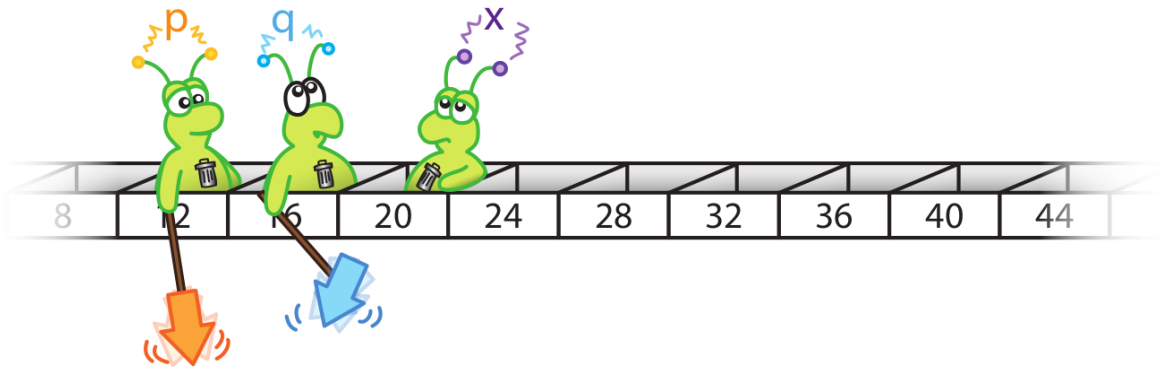
Operator	Function
*	pointer / dereference
&	"address of"

```
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */

ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
ip = &z[0]; /* ip now points to z[0] */
```

Source: K&R Chapter 5

Pointers illustrated

Imagine memory as a long block of boxes that store data. Each box is labeled with an **address**. A **pointer** is a variable that holds a particular address. An **array** is a group of contiguous boxes that can be accessed by their index values.




```
int *p, *q, x;
```

Here we declare p and q as pointers that will hold the *addresses* of `int` variables, and `x` as an ordinary `int` variable.

C pointer syntax

Operator	Function
*	pointer / dereference
&	"address of"

```
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */

ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
ip = &z[0]; /* ip now points to z[0] */
```

Source: K&R Chapter 5

C pointer syntax

C uses "call-by-value" semantics for function calls

```
void swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
int a=123,b=456;

swap(a, b);
```

This function won't swap a and b, only *copies* of the values.

Source: K&R Section 5.2

Addressing example

```
void swap (long *xp,
           long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Called with:

```
long a=123,b=456;

swap(&a, &b);
```

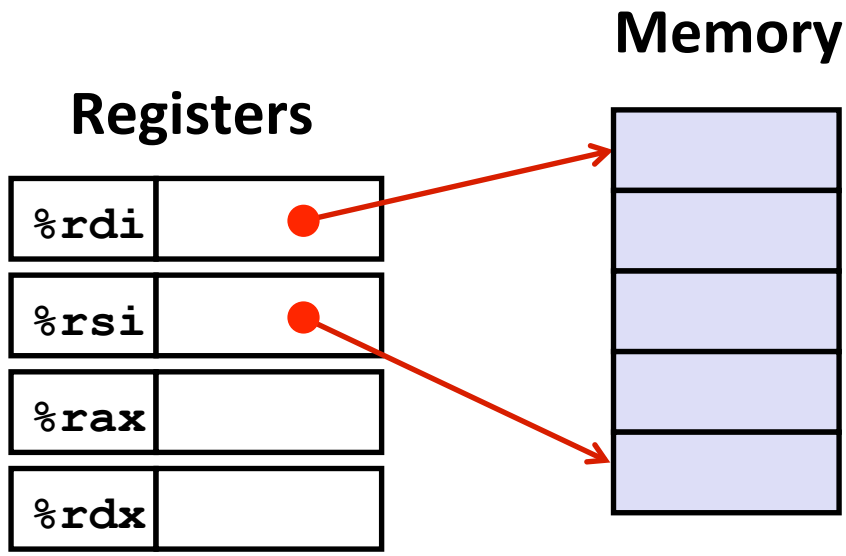
```
swap:
  mov   rax, QWORD PTR [rdi]
  mov   rdx, QWORD PTR [rsi]
  mov   QWORD PTR [rdi], rdx
  mov   QWORD PTR [rsi], rax
```

(or in the other "flavor" asm)

```
swap:
  movq    (%rdi), %rax
  movq    (%rsi), %rdx
  movq    %rdx, (%rdi)
  movq    %rax, (%rsi)
  ret
```

Understanding swap()

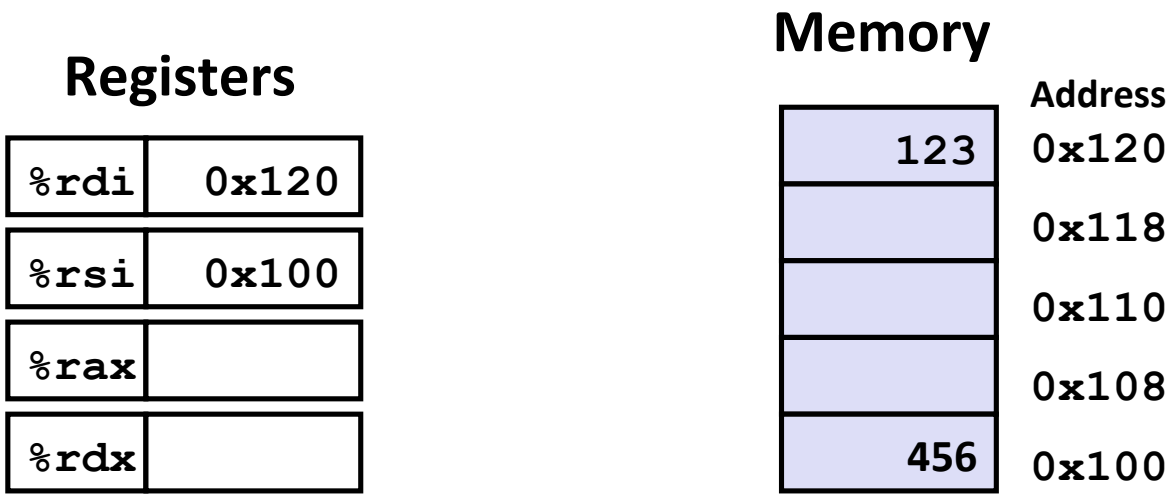
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
rdi	xp
rsi	yp
rax	t0
rdx	t1

```
mov   rax, QWORD PTR [rdi]
mov   rdx, QWORD PTR [rsi]
mov   QWORD PTR [rdi], rdx
mov   QWORD PTR [rsi], rax
```

Understanding swap()




```
swap:
   mov   rax, QWORD PTR [rdi]
   mov   rdx, QWORD PTR [rsi]
   mov   QWORD PTR [rdi], rdx
   mov   QWORD PTR [rsi], rax
```

```
swap:
   mov   rax, QWORD PTR [rdi]
   mov   rdx, QWORD PTR [rsi]
```

lab warmup

- get `sum.c` from 2467 course page
- (go to schedule, link to example code for today)
- compile with `gcc sum.c`
- Then run: `objdump -d a.out -M intel`
- ignore noise at beginning, look for `sumstore` and `plus`
- compile with optimizations such as `gcc -Og` and `-O3`, compare
- launch GNU debugger: `gdb ./a.out`
- breakpoint on `sumstore`...

  commands to try: `run`, `disassemble`, `x`, `nexti`

bomblab

- download bomblab from on campus into homedir on `systems-lab`

  either use classroom terminals, or `systems-lab-web`
- untar, examine with `objdump`
- open with `gdb`