Machine-Level Programming I: Basics

15-213/14-513/15-513: Introduction to Computer Systems
5th Lecture, September 9, 2021

Today: Machine Programming I: Basics

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

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
    - Now 3 volumes, about 5,000 pages of documentation
- **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**	**1060-3333**

- First multi-core Intel processor

Core i7	**2008**	**731M**	**1600-4400**

- Four cores (our shark machines)

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

Intel x86 Processors, cont.

■ 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
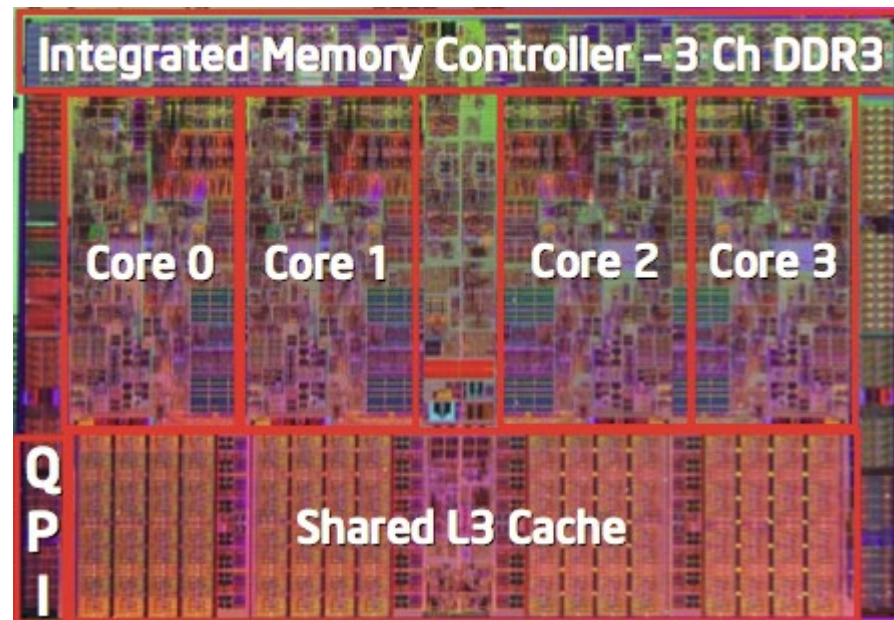
■ Added Features

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

Intel x86 Processors, cont.

- **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
    9. Cannon Lake   2018   10 nm
    10. Ice Lake   2019   10 nm
    11. Tiger Lake   2020   10 nm

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

2018 State of the Art: Coffee Lake
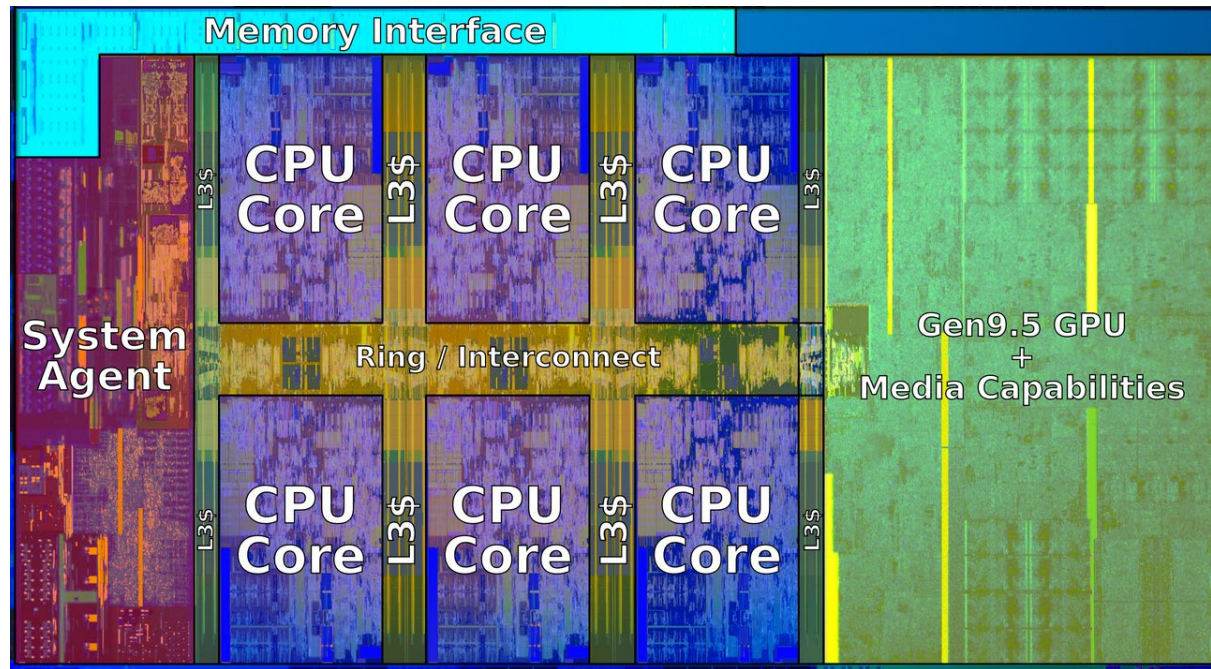
Mobile Model: Core i7

- 2.2-3.2 GHz
- 45 W

Desktop Model: Core i7

- Integrated graphics
- 2.4-4.0 GHz
- 35-95 W

Server Model: Xeon E

- 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 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
    - 1995-2011: Lead semiconductor "fab" in world
    - 2018: #2 largest by $$ (#1 is Samsung)
    - 2019: reclaimed #1
  - AMD fell behind
    - Relies on external semiconductor manufacturer GlobalFoundaries
    - ca. 2019 CPUs (e.g., Ryzen) are competitive again

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

Our Coverage

- **IA32**
  - The traditional x86
  - For 15/18-213: RIP, Summer 2015

- **x86-64**
  - The standard
  - ```shark> gcc hello.c```
  - ```shark> gcc –m64 hello.c```

- **Presentation**
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

Today: Machine Programming I: Basics

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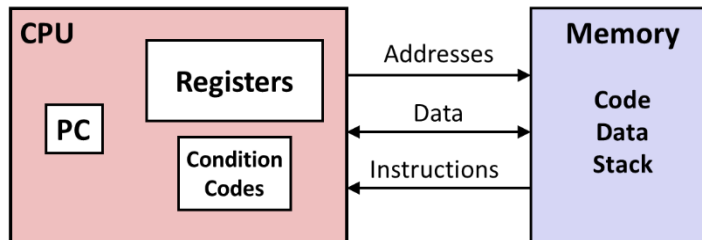
Levels of Abstraction
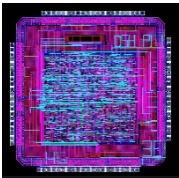
C programmer

```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```
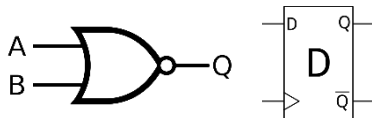
Seems like nice clean layers…

Assembly programmer


Computer Designer

Gates, clocks, circuit layout, …

Definitions

- **Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing 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
  - RISC V: New open-source ISA

Assembly/Machine Code View
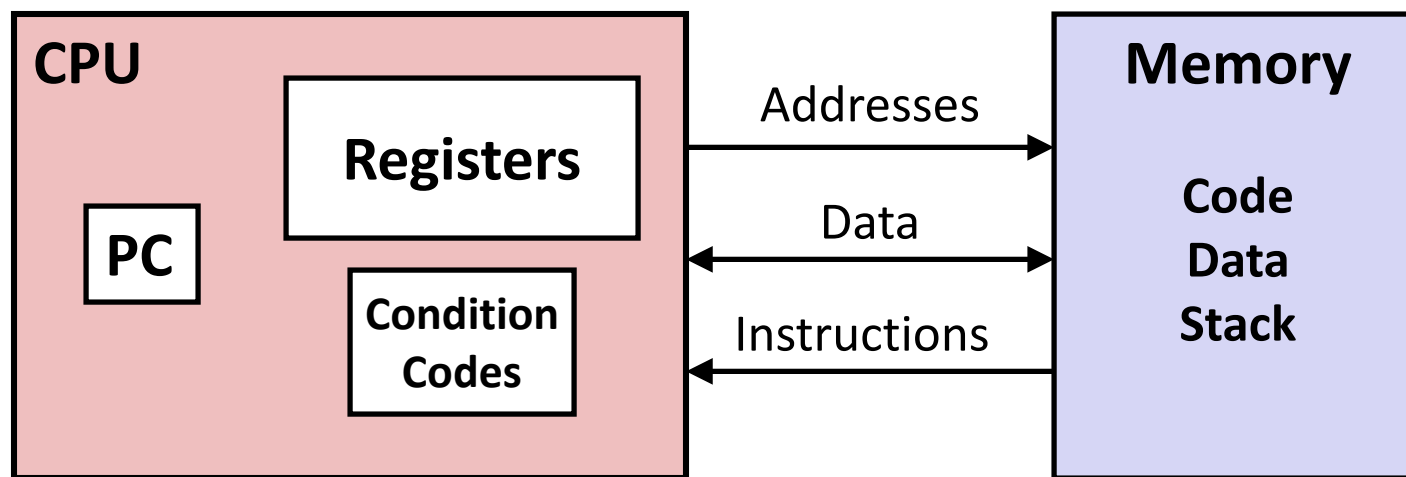
Programmer-Visible State

- **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

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**

- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**

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

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

x86-64 Integer 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)
- Not part of memory (or cache)

Some History: IA32 Registers
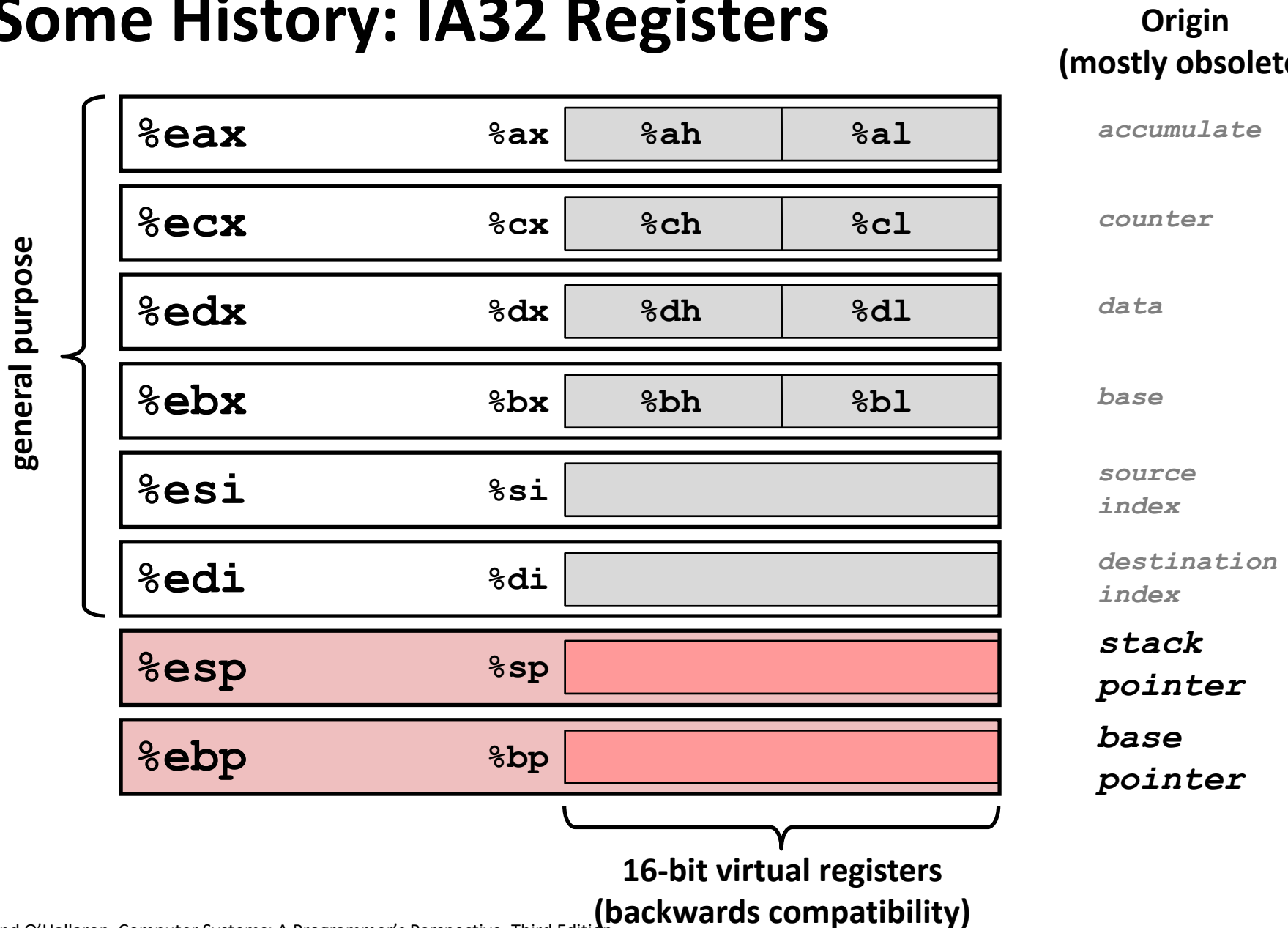
**Origin
(mostly obsolete)**

%eax	**%ax**	**%ah**	**%al**	*accumulate*
%ecx	**%cx**	**%ch**	**%cl**	*counter*
%edx	**%dx**	**%dh**	**%dl**	*data*
%ebx	**%bx**	**%bh**	**%bl**	*base*
%esi	**%si**			*source index*
%edi	**%di**			*destination index*
%esp	**%sp**			*stack pointer*
%ebp	**%bp**			*base pointer*

general purpose

**16-bit virtual registers
(backwards compatibility)**

Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

Moving Data

- **Moving Data**

  `movq` *Source*, *Dest*

- **Operand Types**

  - *Immediate:* Constant integer data
    - Example: `$0x400, $-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - *Register:* One of 16 integer registers
    - Example: `%rax, %r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - *Memory:* 8 consecutive bytes of memory at address given by register
    - Simplest example: `(%rax)`
    - Various other "addressing modes"

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`

`%rN`

Warning: Intel docs use mov *Dest, Source*

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- **Normal**   **(R)**   **Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  ```
  movq (%rcx),%rax
  ```

- **Displacement**   **D(R)**   **Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  ```
  movq 8(%rbp),%rdx
  ```

Example of Simple Addressing Modes

```
void
whatAmI(<type> a, <type> b)
{
    ????
}
```

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

%rsi
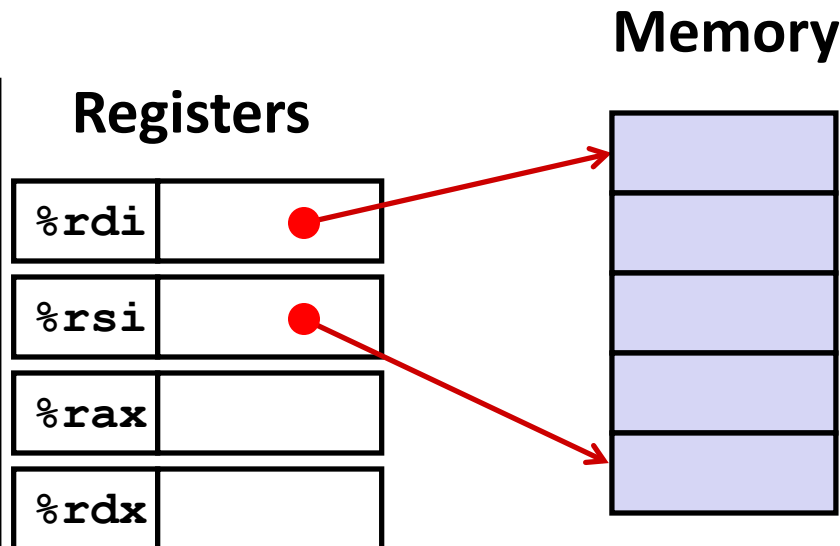
%rdi

Example of Simple Addressing Modes

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

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

Understanding Swap()

Memory

```
void swap
    (long *xp, long *yp)
{

  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Registers

%rdi	
%rsi	
%rax	
%rdx	

Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()

Registers

Memory

%rdi	0x120

%rsi	0x100

%rax	

%rdx	

Address

123	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()

Memory

Registers

%rdi	0x120

%rsi	0x100

%rax	123

%rdx	

Address

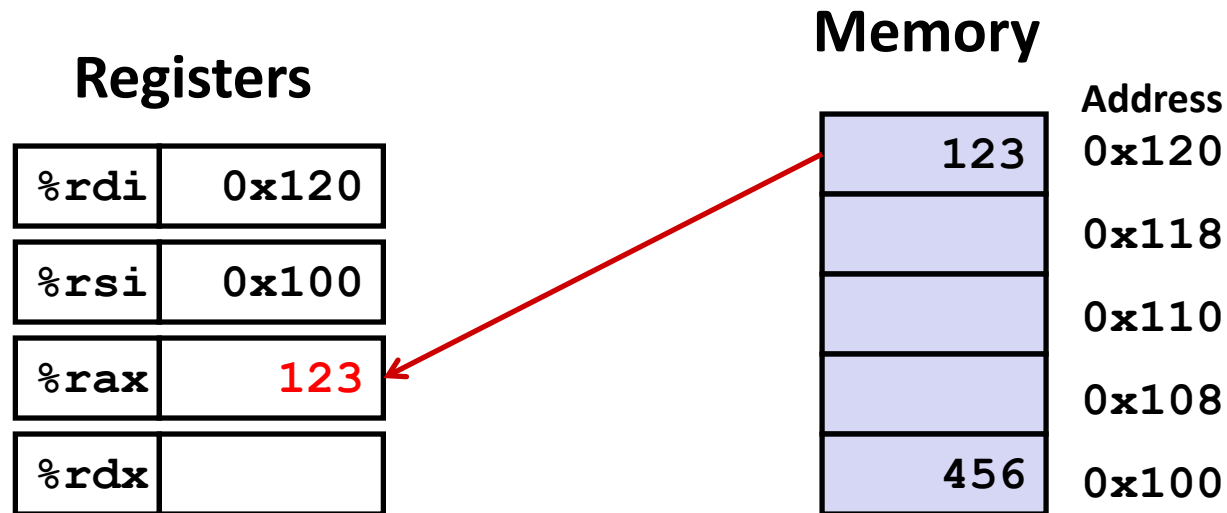|	
123	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
    movq    (%rdi), %rax   # t0 = *xp
    movq    (%rsi), %rdx   # t1 = *yp
    movq    %rdx, (%rdi)   # *xp = t1
    movq    %rax, (%rsi)   # *yp = t0
    ret
```

Understanding Swap()

Memory

Registers

%rdi	**0x120**
%rsi	**0x100**
%rax	**123**
%rdx	**456**

Address

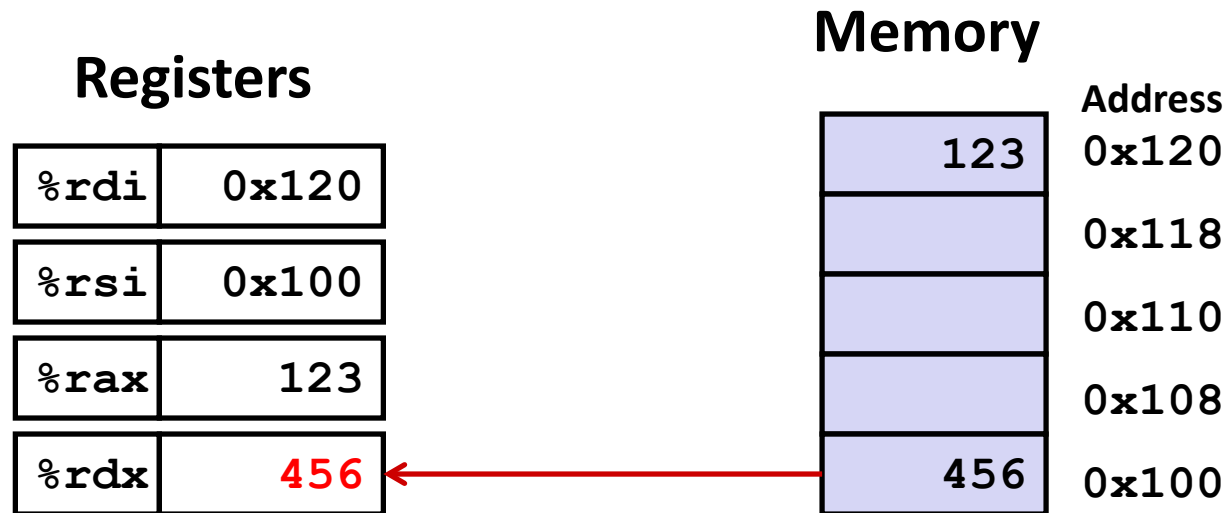|	
123	**0x120**
	0x118
	0x110
	0x108
456	**0x100**

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()

Memory

Registers

Address

%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

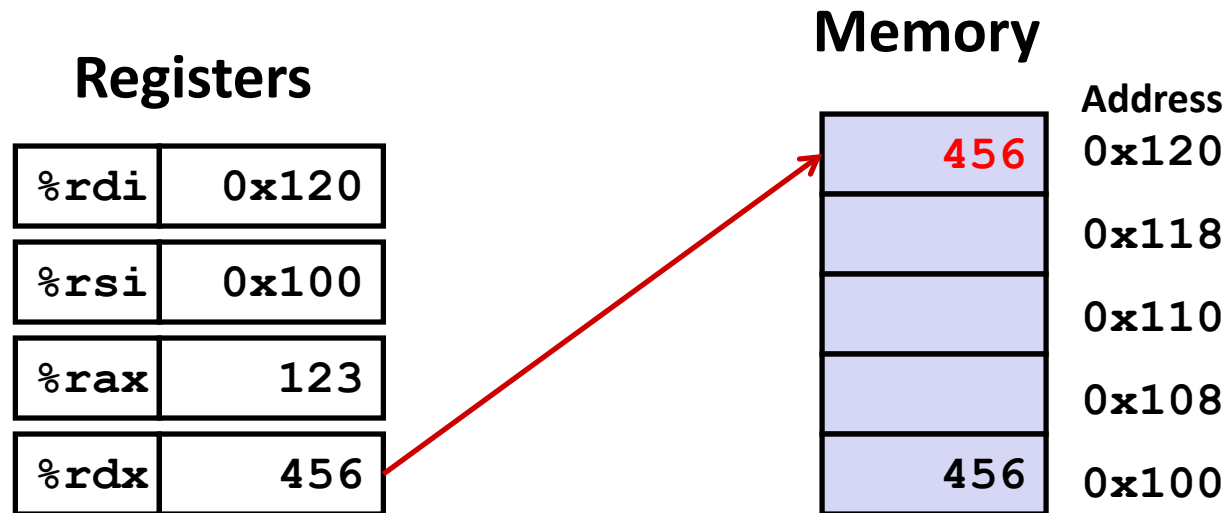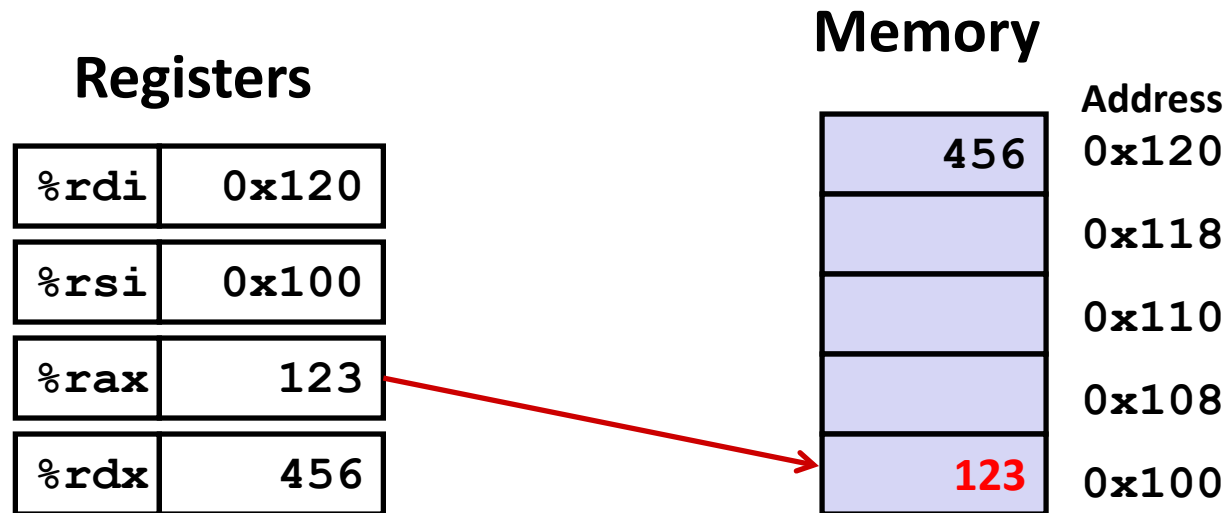|	
456	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()

Registers

Memory

%rdi	**0x120**
%rsi	**0x100**
%rax	**123**
%rdx	**456**

Address

456	**0x120**
	0x118
	0x110
	0x108
123	**0x100**

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Simple Memory Addressing Modes

■ **Normal**   **(R)**   **Mem[Reg[R]]**

  ■ Register R specifies memory address

  ■ Aha! Pointer dereferencing in C

     `movq (%rcx),%rax`

■ **Displacement**   **D(R)**   **Mem[Reg[R]+D]**

  ■ Register R specifies start of memory region

  ■ Constant displacement D specifies offset

     `movq 8(%rbp),%rdx`

Complete Memory Addressing Modes

■ Most General Form

D(Rb,Ri,S)   **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- ▪ D: Constant "displacement" 1, 2, or 4 bytes
- ▪ Rb: Base register: Any of 16 integer registers
- ▪ Ri: Index register: Any, except for `%rsp`
- ▪ S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ Special Cases

(Rb,Ri)   **Mem[Reg[Rb]+Reg[Ri]]**

D(Rb,Ri)   **Mem[Reg[Rb]+Reg[Ri]+D]**

(Rb,Ri,S)   **Mem[Reg[Rb]+S*Reg[Ri]]**

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

D(Rb,Ri,S)   **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Expression	Address Computation	Address
0x8(%rdx)		
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Today: Machine Programming I: Basics

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

Address Computation Instruction

- **leaq** *Src, Dst*
  - *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

- **Example**

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

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax   # t = x+2*x
salq $2, %rax              # return t<<2
```

Quiz Time!

Check out:

https://canvas.cmu.edu/courses/24383/quizzes/67227

Some Arithmetic Operations

- **Two Operand Instructions:**

Format   *Computation*

addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest − Src	
imulq	Src,Dest	Dest = Dest * Src	
salq	Src,Dest	Dest = Dest << Src	*Also called shlq*
sarq	Src,Dest	Dest = Dest >> Src	*Arithmetic*
shrq	Src,Dest	Dest = Dest >> Src	*Logical*
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

- **Watch out for argument order! *Src,Dest*
(Warning: Intel docs use "op *Dest,Src*")**

- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One Operand Instructions**

  incq   *Dest*   *Dest = Dest + 1*

  decq   *Dest*   *Dest = Dest − 1*

  negq   *Dest*   *Dest = − Dest*

  notq   *Dest*   *Dest = ~Dest*

- **See book for more instructions**

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:
    leaq    (%rdi,%rsi), %rax
    addq    %rdx, %rax
    leaq    (%rsi,%rsi,2), %rdx
    salq    $4, %rdx
    leaq    4(%rdi,%rdx), %rcx
    imulq   %rcx, %rax
    ret
```

Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

Understanding Arithmetic Expression Example

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

```
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;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

Today: Machine Programming I: Basics

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

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`

text   **C program (`p1.c p2.c`)**

        Compiler (`gcc –Og -S`)

text   **Asm program (`p1.s p2.s`)**

        Assembler (`gcc –c` or `as`)

binary   **Object program (`p1.o p2.o`)**   **Static libraries (`.a`)**

        Linker (`gcc` or `ld`)

binary   **Executable program (`p`)**
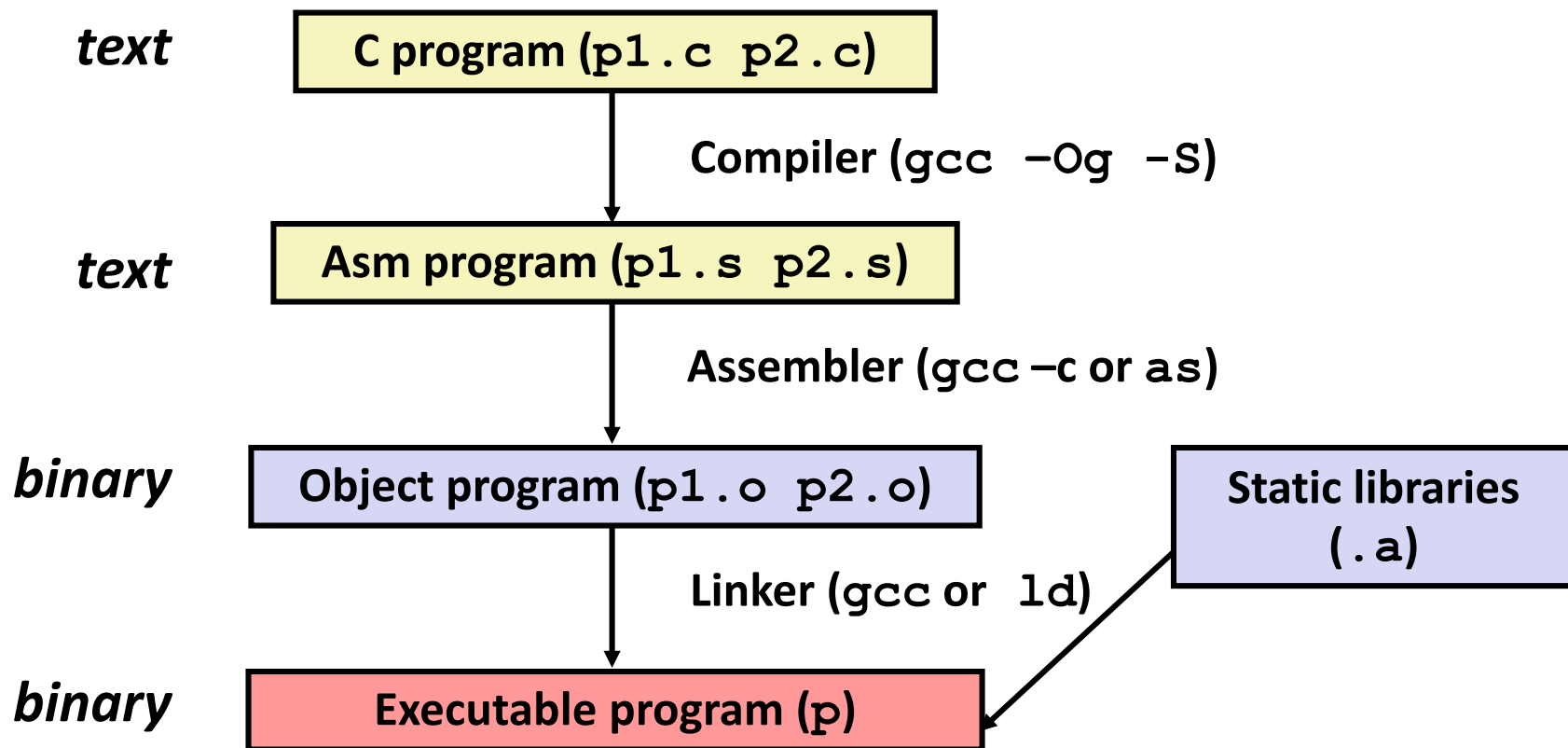
Compiling Into Assembly

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 shark machine) with command

> `gcc –Og –S sum.c`

Produces file `sum.s`

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

What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

What it really looks like

Things that look weird
and are preceded by a '.'
are generally directives.

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

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

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**

- **(SIMD vector data types of 8, 16, 32 or 64 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

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory

- **Perform arithmetic function on register or memory data**

- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branch

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

Alternate Disassembly

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**
  - Disassemble procedure

  `gdb sum`

  `disassemble sumstore`

Alternate Disassembly

Disassembled

Object Code

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

```
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**

  - Disassemble procedure

  gdb sum

  disassemble sumstore

  - Examine the 14 bytes starting at `sumstore`

  x/14xb 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:        Reverse engineering forbidden by
30001005:      Microsoft End User License Agreement
3000100a:
```

- **Anything that can be interpreted as executable code**

- **Disassembler examines bytes and reconstructs assembly source**

Machine Programming I: Summary

- **History of Intel processors and architectures**
  - Evolutionary design leads to many quirks and artifacts

- **C, assembly, machine code**
  - New forms of visible state: program counter, registers, …
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

- **Assembly Basics: Registers, operands, move**
  - The x86-64 move instructions cover wide range of data movement forms

- **Arithmetic**
  - C compiler will figure out different instruction combinations to carry out computation