# Machine-Level Programming I: Basics

CSE 238/2038/2138: Systems Programming

**Instructor:**

Fatma CORUT ERGİN

*Slides adapted from Bryant & O'Hallaron's slides*

## 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

| *Name* | *Date* | *Transistors* | *MHz* |
|---|---|---|---|
| **386** | **1985** | **275K** | **16-33** |

- First 32 bit Intel processor , referred to as IA32
- Added “flat addressing”, capable of running Unix

| *Name* | *Date* | *Transistors* | *MHz* |
|---|---|---|---|
| **Pentium 4E** | **2004** | **125M** | **2800-3800** |

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

| *Name* | *Date* | *Transistors* | *MHz* |
|---|---|---|---|
| **Core 2** | **2006** | **291M** | **1060-3500** |

- First multi-core Intel processor

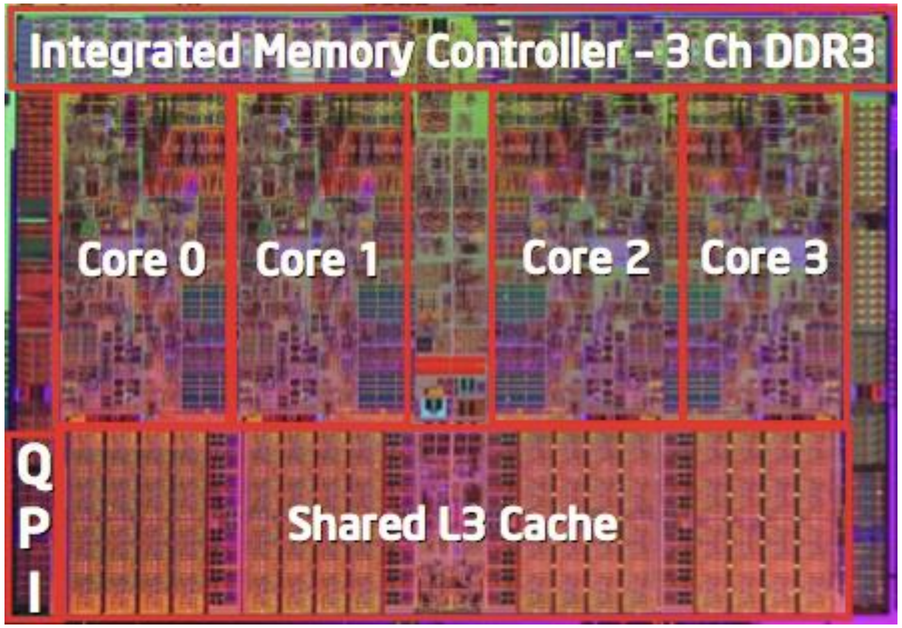
| *Name* | *Date* | *Transistors* | *MHz* |
|---|---|---|---|
| **Core i7** | **2008** | **731M** | **1700-3900** |

- Four cores

# 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 | 2001 | 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

## 2017 State of the Art

- **Mobile Model: Core i7**
  - 2.6-2.9 GHz
  - 45 W
- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.8-4.0 GHz
  - 35-91 W
- **Server Model: Xeon**
  - Integrated graphics
  - Multi-socket enabled
  - 2-3.7 GHz
  - 25-80 W

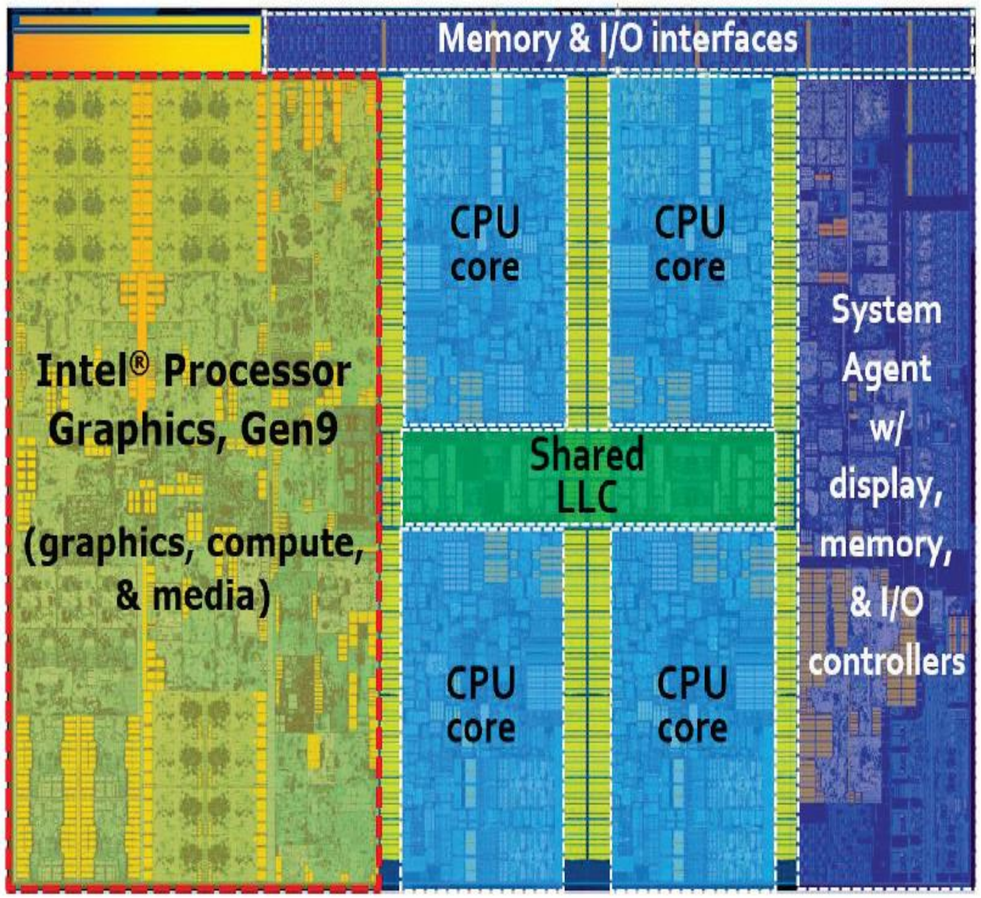

Memory & I/O interfaces
Intel® Processor Graphics, Gen9 (graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

*Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.*

# 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,

## 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

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

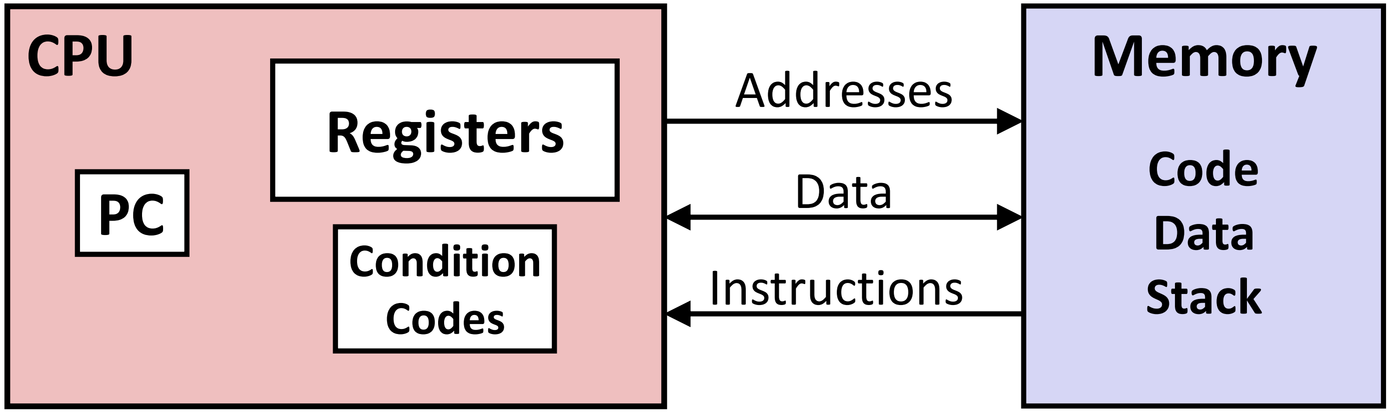
## Today: Machine Programming I: Basics

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

## 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
  - 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

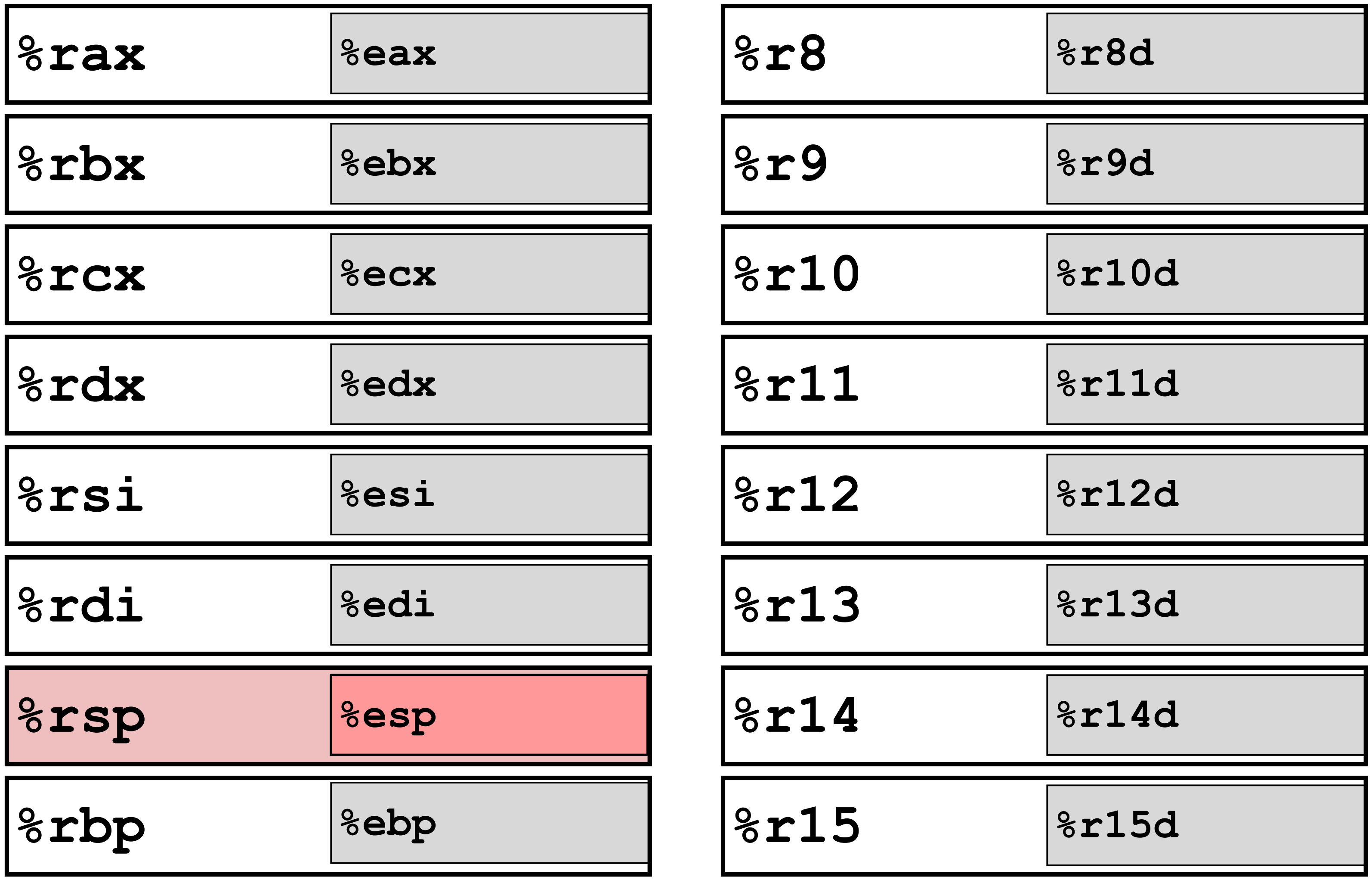
- **“Integer” data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)

- **Floating point data of 4, 8, or 10 bytes**

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

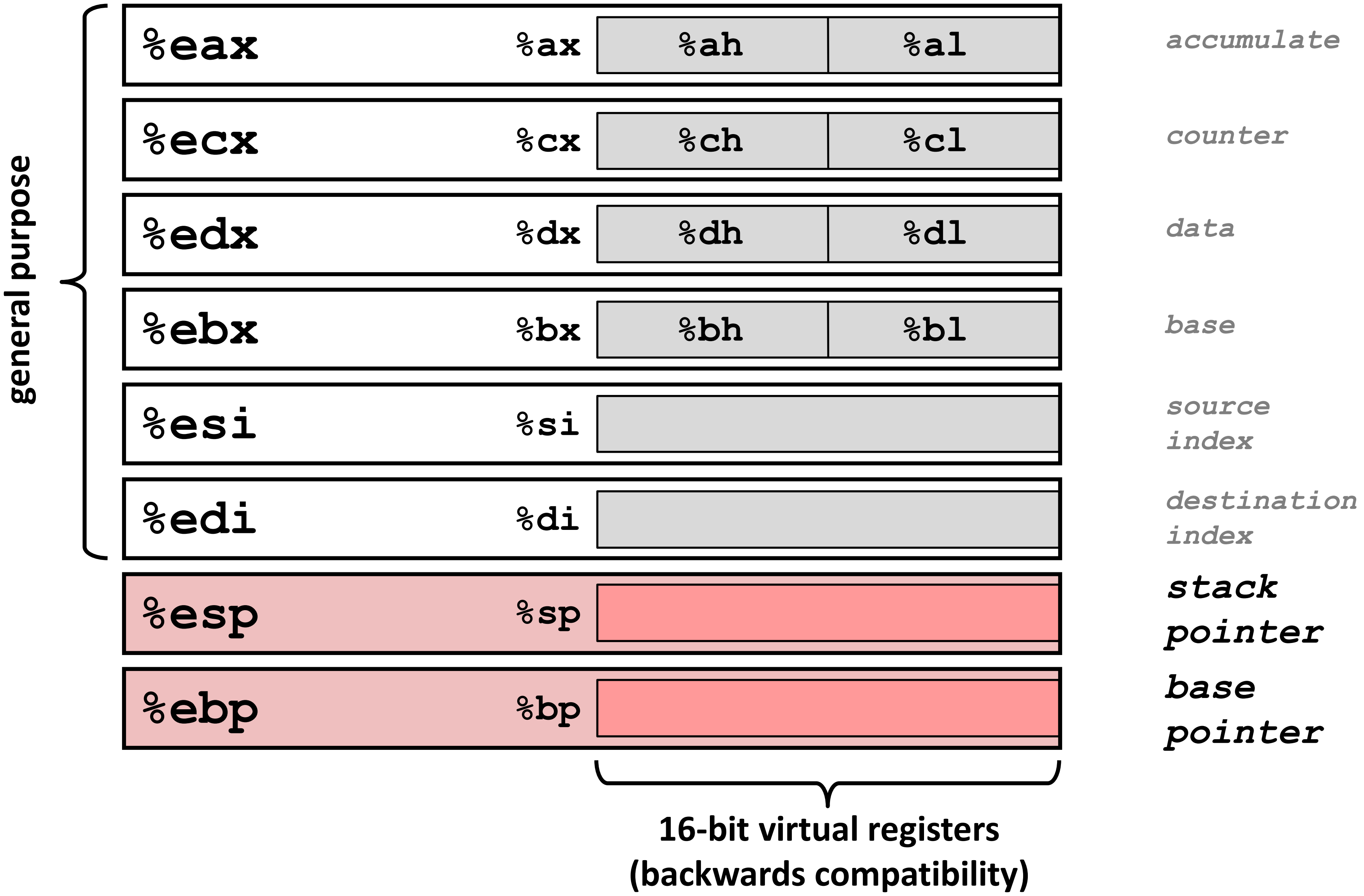
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

# x86-64 Integer Registers



- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

# Some History: IA32 Registers



## Assembly Characteristics: Operations

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

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

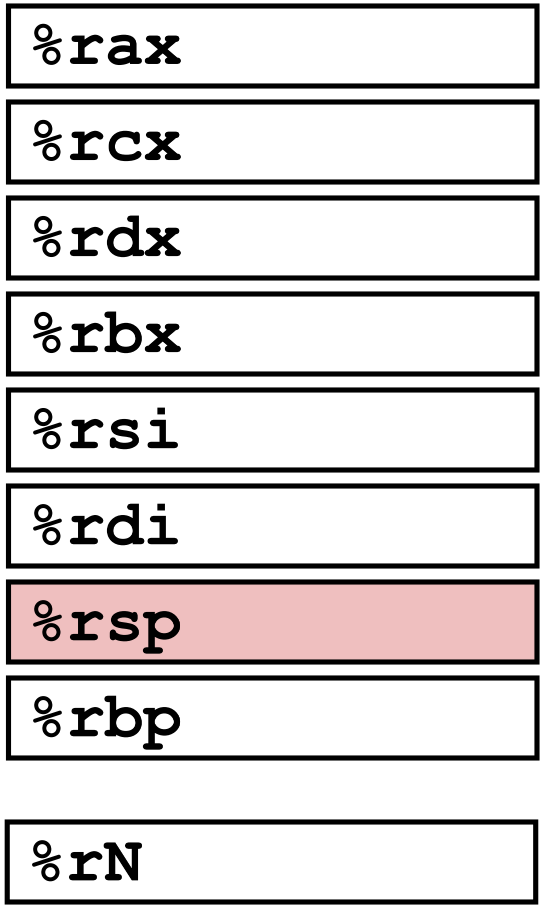
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

# 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 "address modes"



**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
  - 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 aFunc(<type> a, <type> b){
     ????
}
```

`%rdi` `%rsi`

# Example of Simple Addressing Modes

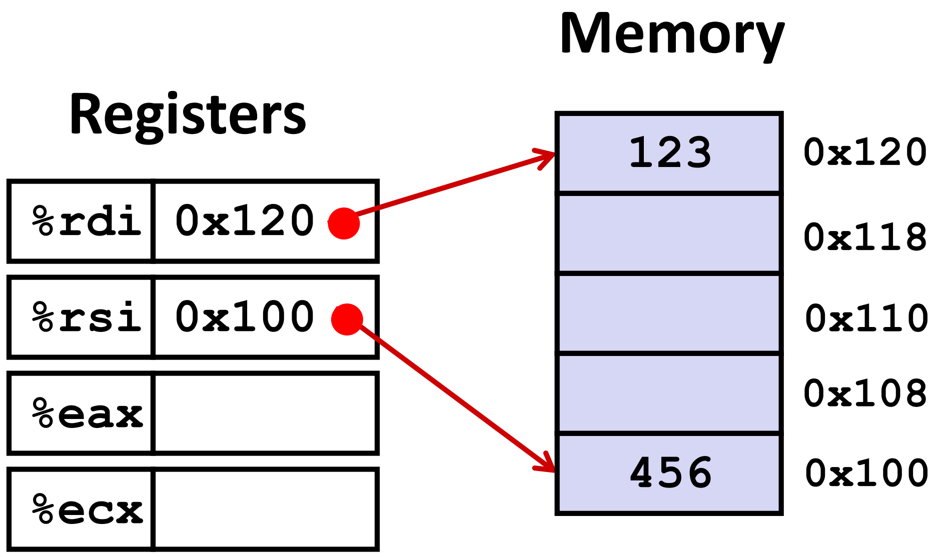
```
void swap (int *xp, int *yp){
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movl    (%rdi), %eax
   movl    (%rsi), %ecx
   movl    %ecx, (%rdi)
   movl    %eax, (%rsi)
   ret
```

## Understanding Swap()

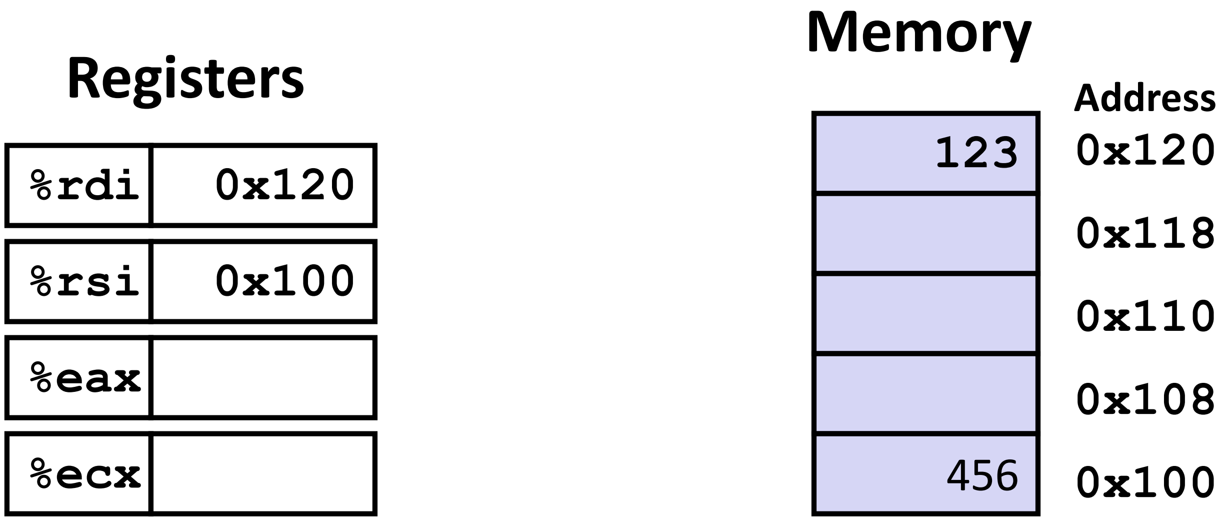
```
void swap (int *xp, int *yp){
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

| Register | Value |
|---|---|
| %rdi | xp |
| %rsi | yp |
| %eax | t0 |
| %ecx | t1 |



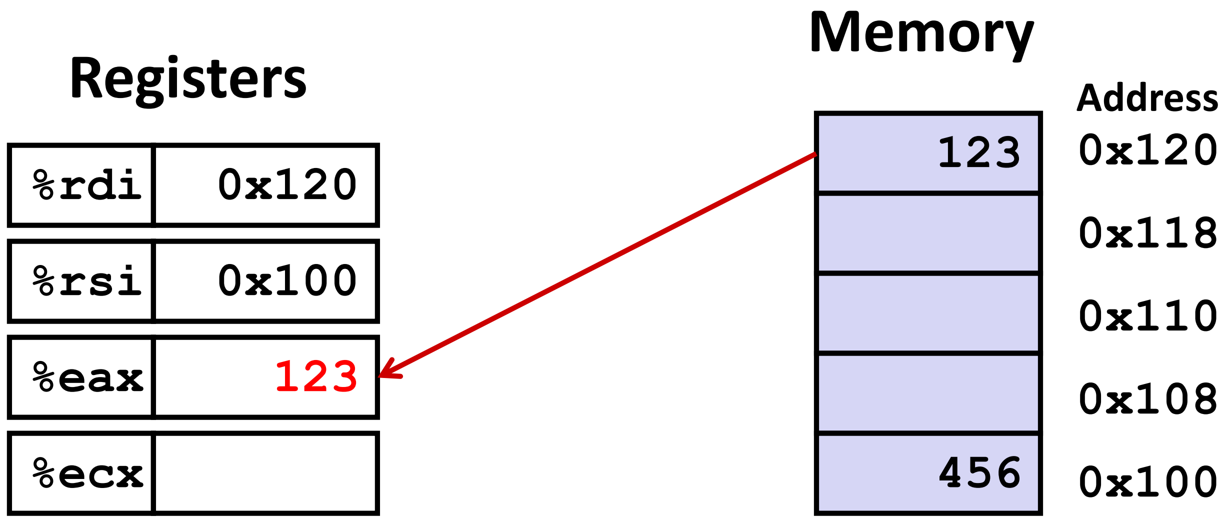
```
swap:
   movl    (%rdi), %eax       # t0 = *xp
   movl    (%rsi), %ecx       # t1 = *yp
   movl    %ecx, (%rdi)       # *xp = t1
   movl    %eax, (%rsi)       # *yp = t0
   ret
```

# Understanding Swap()



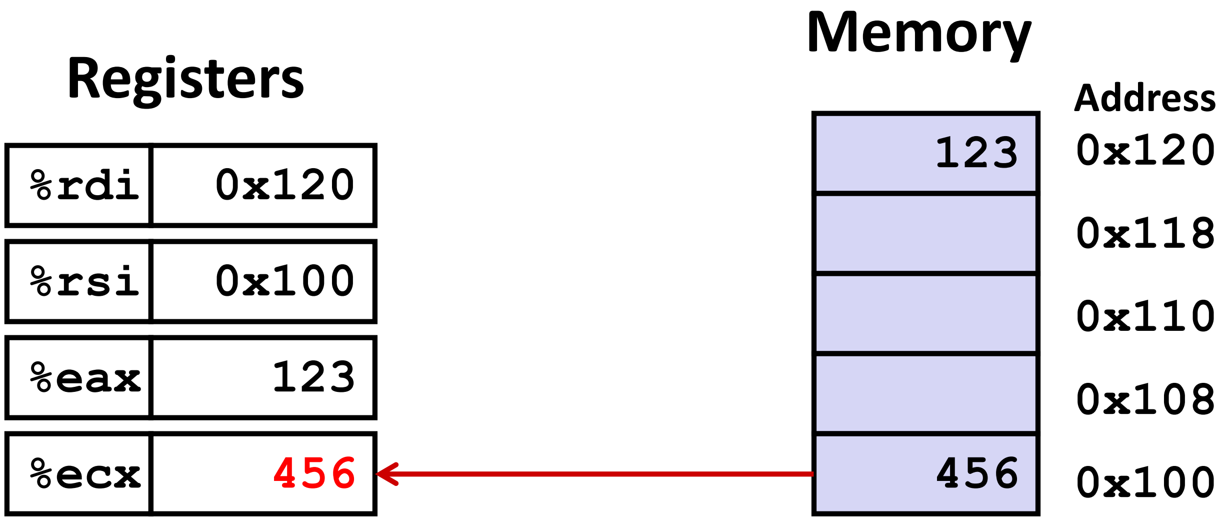
```
swap:
   movl    (%rdi), %eax       # t0 = *xp
   movl    (%rsi), %ecx       # t1 = *yp
   movl    %ecx, (%rdi)       # *xp = t1
   movl    %eax, (%rsi)       # *yp = t0
   ret
```

# Understanding Swap()



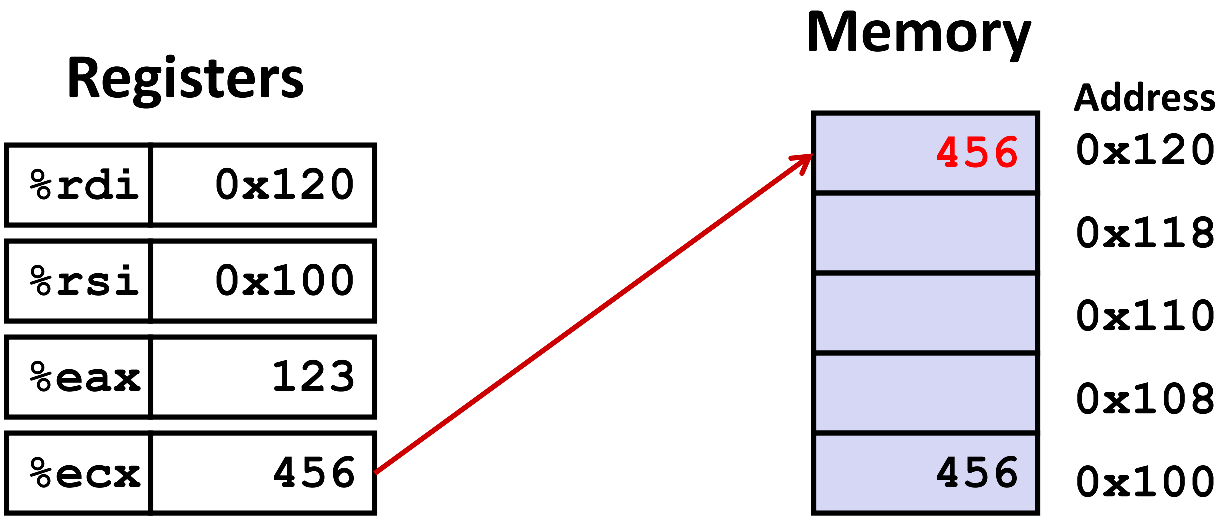
```
swap:
   movl    (%rdi), %eax      # t0 = *xp
   movl    (%rsi), %ecx      # t1 = *yp
   movl    %ecx, (%rdi)      # *xp = t1
   movl    %eax, (%rsi)      # *yp = t0
   ret
```

# Understanding Swap()



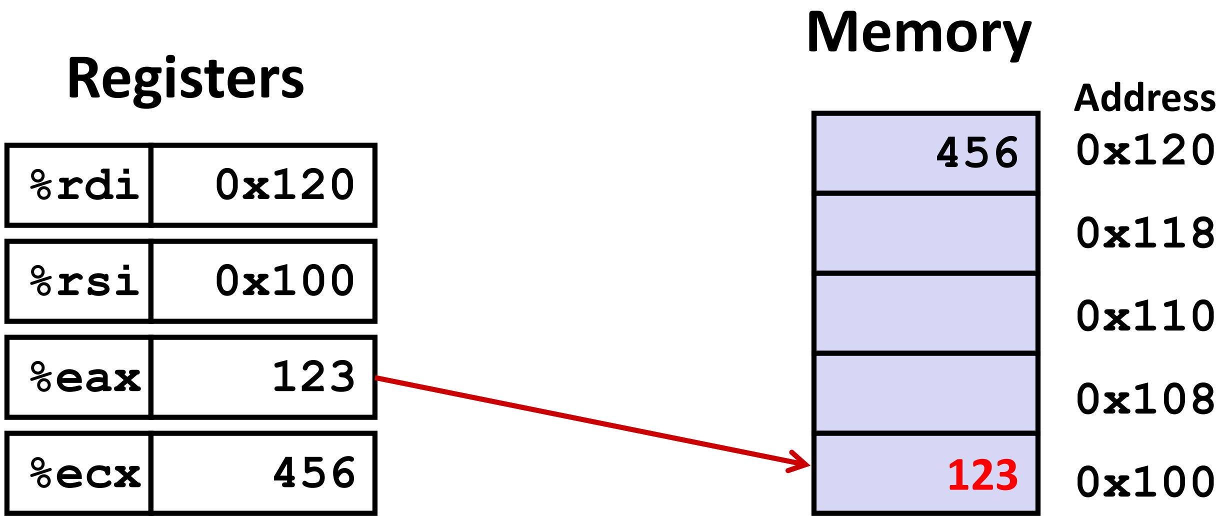
```
swap:
   movl    (%rdi), %eax       # t0 = *xp
   movl    (%rsi), %ecx       # t1 = *yp
   movl    %ecx, (%rdi)       # *xp = t1
   movl    %eax, (%rsi)       # *yp = t0
   ret
```

# Understanding Swap()



```
swap:
   movl    (%rdi), %eax       # t0 = *xp
   movl    (%rsi), %ecx       # t1 = *yp
   movl    %ecx, (%rdi)       # *xp = t1
   movl    %eax, (%rsi)       # *yp = t0
   ret
```

# Understanding Swap()



```
swap:
   movl    (%rdi), %eax        # t0 = *xp
   movl    (%rsi), %ecx        # t1 = *yp
   movl    %ecx, (%rdi)        # *xp = t1
   movl    %eax, (%rsi)        # *yp = t0
   ret
```

# 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)` | `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+x*2
salq $2, %rax            # return t<<2
```

# 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 right shift* |
| `shrq` | `Src,Dest` | Dest = Dest >> Src | *Logical right shift* |
| `xorq` | `Src,Dest` | Dest = Dest ^ Src | |
| `andq` | `Src,Dest` | Dest = Dest & Src | |
| `orq` | `Src,Dest` | Dest = Dest \| Src | |

- **Watch out for argument order!**

  **(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

```
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    # 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
```

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

**Compiler optimization:**

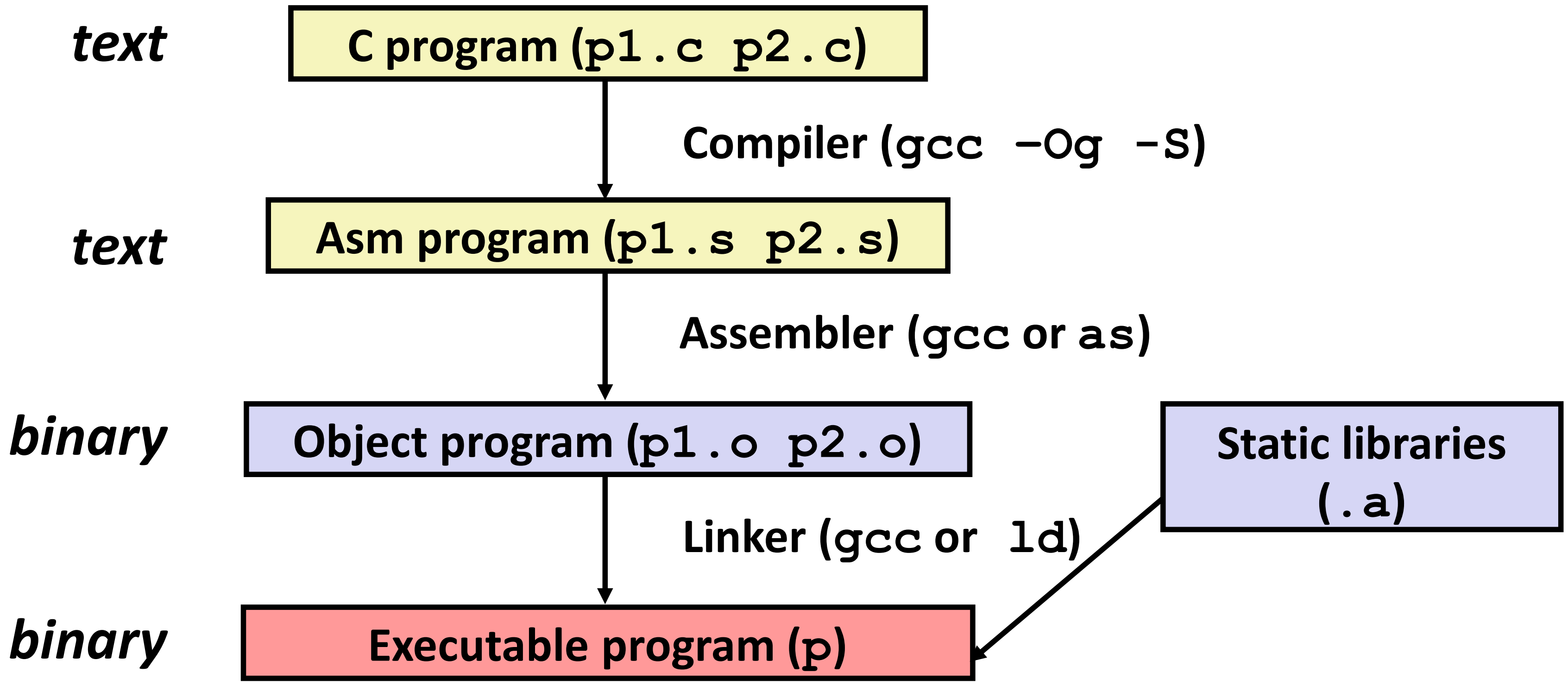
- Reuse of registers
- Substitution (copy propagation)
- Strength reduction

## 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`



# Compiling Into Assembly

**C Code (sum.c)**

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

**Generated x86-64 Assembly**

```
sumstore:
   pushq   %rbp
   movq    %rsp, %rbp
   addl    %esi, %edi
   movl    %edi, (%rdx)
   popq    %rbp
   ret
```

**Obtain (on Mac OS) with command**

```
gcc –O –S sum.c
```

**Produces file `sum.s`**

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

## What it Really Looks Like

```
        .globl  _sumstore
        .align  4, 0x90
_sumstore:
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp3:
        .cfi_def_cfa_offset 16
Ltmp4:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp5:
        .cfi_def_cfa_register %rbp
        addl    %esi, %edi
        movl    %edi, (%rdx)
        popq    %rbp
        retq
        .cfi_endproc
```

**Things preceding with a "." are generally directives**

```
sumstore:
   pushq    %rbp
   movq     %rsp, %rbp
   addl     %esi, %edi
   movl     %edi, (%rdx)
   popq     %rbp
   ret
```

# Object Code

## Code for `sumstore`

```
0x100000f20:
    0x55
    0x48
    0x89
    0xe5
    0x01
    0xf7
    0x89
    0x3a
    0x5d
    0xc3
```

- **Total of 10 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x100000f20`**

- **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`**

```
movl      %edi, (%rdx)
```

- **Assembly**
  - Move 4-byte value to memory
  - Operands:

| | |
|---|---|
| **`t:`** | Register **`%edi`** |
| **`dest:`** | Register **`%rdx`** |
| **`*dest:`** | Memory **M[`%rdx`]** |

```
0x100000f26:  89 3a
```

- **Object Code**
  - 2-byte instruction
  - Stored at address **`0x100000f26`**

# Disassembling Object Code

- **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

## Disassembled

```
<sumstore>:
  100000f20:  55                 pushq  %rbp
  100000f21:  48 89 e5           movq   %rsp, %rbp
  100000f24:  01 f7              addl   %esi, %edi
  100000f26:  89 3a              movl   %edi, (%rdx)
  100000f28:  5d                 popq   %rbp
  100000f29:  c3                 retq
```

## Alternate Disassembly

**Object**

```
0x100000f20:
    0x55
    0x48
    0x89
    0xe5
    0x01
    0xf7
    0x89
    0x3a
    0x5d
    0xc3
```

**Disassembled**

```
Dump of assembler code for function sumstore:
 0x0000000100000f20 <+0>: pushq   %rbp
 0x0000000100000f21 <+1>: movq    %rsp,%rbp
 0x0000000100000f24 <+4>: addl    %esi, %edi
 0x0000000100000f26 <+6>: movl    %edi, (%rdx)
 0x0000000100000f28 <+8>: popq    %rbp
 0x0000000100000f29 <+9>: retq
```

- **Within gdb Debugger**
  - `gdb sum`
  - `disassemble sumstore`
    - Disassemble procedure
  - `x/10xb sumstore`
    - Examine the 10 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**

# 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