



# Machine-Level Programming I

15-213/14-513/15-513:

Introduction to Computer Systems, Fall 2026

3<sup>rd</sup> Lecture, September 1, 2026

# Today: Machine Programming I: Basics

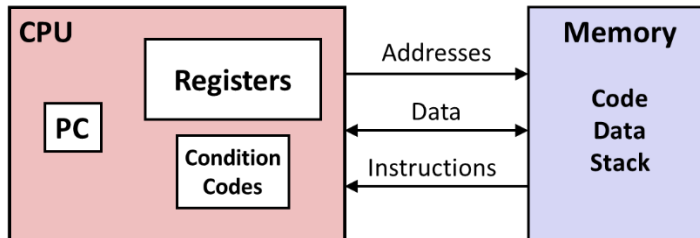
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- Condition codes and jumps
- C, assembly, machine code

# Context: Levels of Abstraction

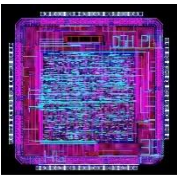
C programmer

```
#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; }
```

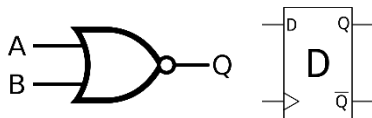
Assembly programmer



Computer Designer



Gates, clocks, circuit layout, ...



# CPU store data in Registers and Memory

- **Registers: Super-fast memory right on the CPU core itself.**
  - About 16 named registers on x86
- **Many instructions operate on registers**
- **Registers are fixed-width, often 32 or 64 bits**
- **“int x = 5; x \*= y; x++” – x is probably stored in a *register* while being operated on frequently.**
- **Registers are accessed by their name**
- **“Memory” means DRAM, typically off-chip storage, bigger and cheaper and slower. When you call “malloc”, you usually get this.**
- **Memory is accessed by (byte) address**

# Introduction to Assembly

## ■ Assembly is a programming language

- It has “types” (sort of)
- It has control flow shape
- It is an intermediate language, has very limited verbs, and operations are need to be explicit

## ■ Assembly is defined by the ISA (instruction set architecture)

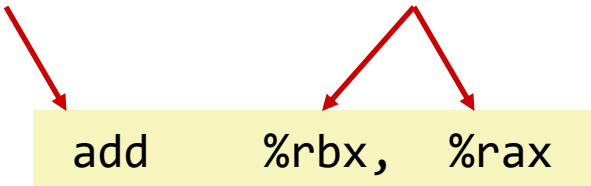
- x86, ARM, RISC-V, etc

## ■ Implemented by the microarchitecture - e.g., a “core i7” x86 processor

# Beginning Grammar

## ■ Say something in assembly:

Operation (op)      Register names



```
add    %rbx, %rax
```

is

```
rax += rbx
```

# Nouns

## ■ Registers

- Special parts of the CPU to hold small amounts of data, like local variables
- Names are limited and very specific:

`%r?x (a,b,c,d)`

`%r?i (d,s)`

`%r?p (b,i,s)`      `// Special meanings`

`%r? (8-15)`

## ■ Memory

- Everything else; accessed by *address*.

# X86 registers – some history

Intel created x86 in the 16-bit microprocessor 8086.

Its registers held 16 bits. So:

“ax”, the “accumulator”, stored 16 bits of data.

Initially the registers had a defined purpose. (They mostly don't now.)

In 1985, Intel introduced the 32-bit 80386, and “e”xtended the registers to 32 bits, creating “eax”

In 2003, AMD introduced x86-64, growing registers to 64 bits and creating “rax”

# x86-64 Integer Registers (**reference**)

|             |             |
|-------------|-------------|
| <b>%rax</b> | <b>%eax</b> |
| <b>%rbx</b> | <b>%ebx</b> |
| <b>%rcx</b> | <b>%ecx</b> |
| <b>%rdx</b> | <b>%edx</b> |
| <b>%rsi</b> | <b>%esi</b> |
| <b>%rdi</b> | <b>%edi</b> |
| <b>%rsp</b> | <b>%esp</b> |
| <b>%rbp</b> | <b>%ebp</b> |

|             |              |
|-------------|--------------|
| <b>%r8</b>  | <b>%r8d</b>  |
| <b>%r9</b>  | <b>%r9d</b>  |
| <b>%r10</b> | <b>%r10d</b> |
| <b>%r11</b> | <b>%r11d</b> |
| <b>%r12</b> | <b>%r12d</b> |
| <b>%r13</b> | <b>%r13d</b> |
| <b>%r14</b> | <b>%r14d</b> |
| <b>%r15</b> | <b>%r15d</b> |

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

# Verbs (i.e., 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

# Verbs have optional suffix

## ■ Operation suffix specifies the size(s) involved

- There are also different register names for the different sizes

| Suffix | Register Name | Size (bytes) |
|--------|---------------|--------------|
| q      | r..           | 8            |
| l      | e.. (usually) | 4            |
| w      | see reference | 2            |
| b      | see reference | 1            |

# Moving Data

■  $x = 5$

`mov $5, %rax`

■  $x = y$

`mov %rcx, %rdx`

■  $x = *p$

`mov (%rsi), %r8`

# Memory

- Parentheses always denote a memory address computation

`(%rdi) → *ptr`

- Most operations will then access memory

# movq Operand Combinations (**reference**)

|      | 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***

# Reading Assembly Example

```
void
```

```
whatAmI (<type> a, <type> b)
```

```
{
```

```
    ????
```

```
}
```

```
whatAmI:
```

```
    movq    (%rdi), %rax
```

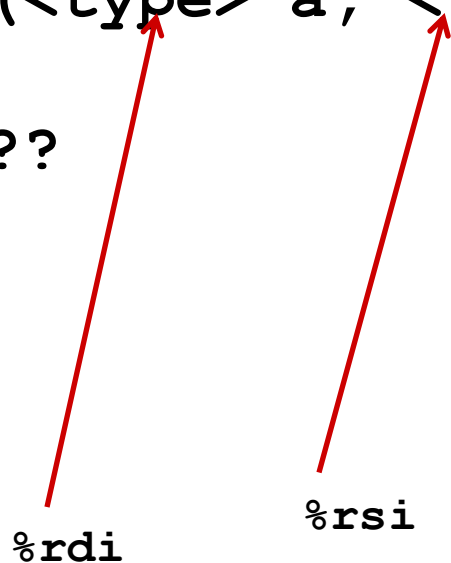
```
    movq    (%rsi), %rdx
```

```
    movq    %rdx, (%rdi)
```

```
    movq    %rax, (%rsi)
```

```
    ret
```

`%rdi`



`%rsi`

What is <type>?

- Find `%rdi` in the assembly.
- How is it used?

# Reading Assembly Example

```
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
```

We have now written the “pseudo-C” for the assembly, using the types and replacing each register with a “variable name”.

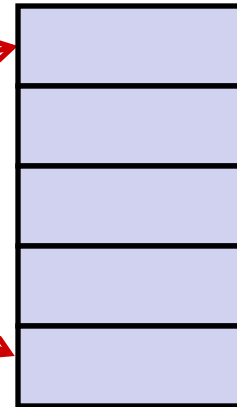
# Understanding Swap()

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

## Registers

|      |  |
|------|--|
| %rdi |  |
| %rsi |  |
| %rax |  |
| %rdx |  |

## Memory



| 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

|                   |                    |
|-------------------|--------------------|
| <code>%rdi</code> | <code>0x120</code> |
| <code>%rsi</code> | <code>0x100</code> |
| <code>%rax</code> |                    |
| <code>%rdx</code> |                    |

## Memory

| Address            |     |
|--------------------|-----|
| <code>0x120</code> | 123 |
| <code>0x118</code> |     |
| <code>0x110</code> |     |
| <code>0x108</code> |     |
| <code>0x100</code> | 456 |

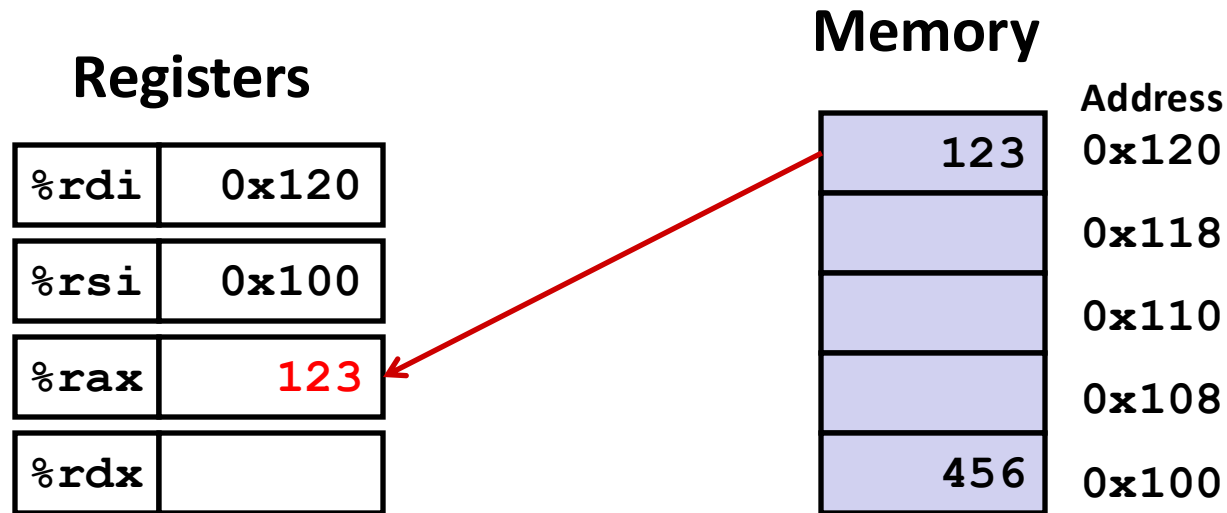
**swap:**

```

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

```

# Understanding Swap()



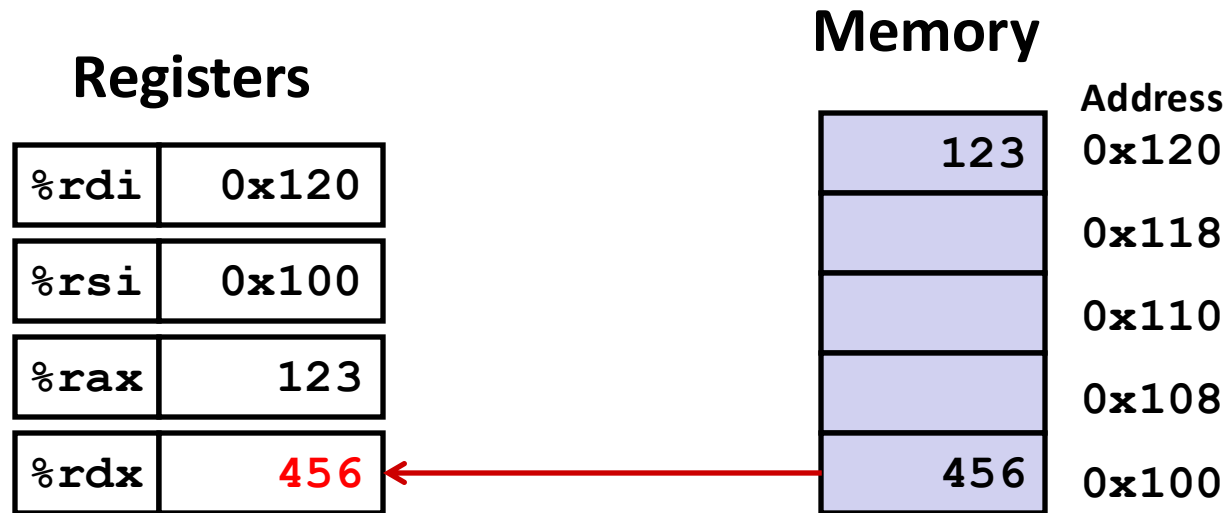
swap:

```

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

```

# Understanding Swap()



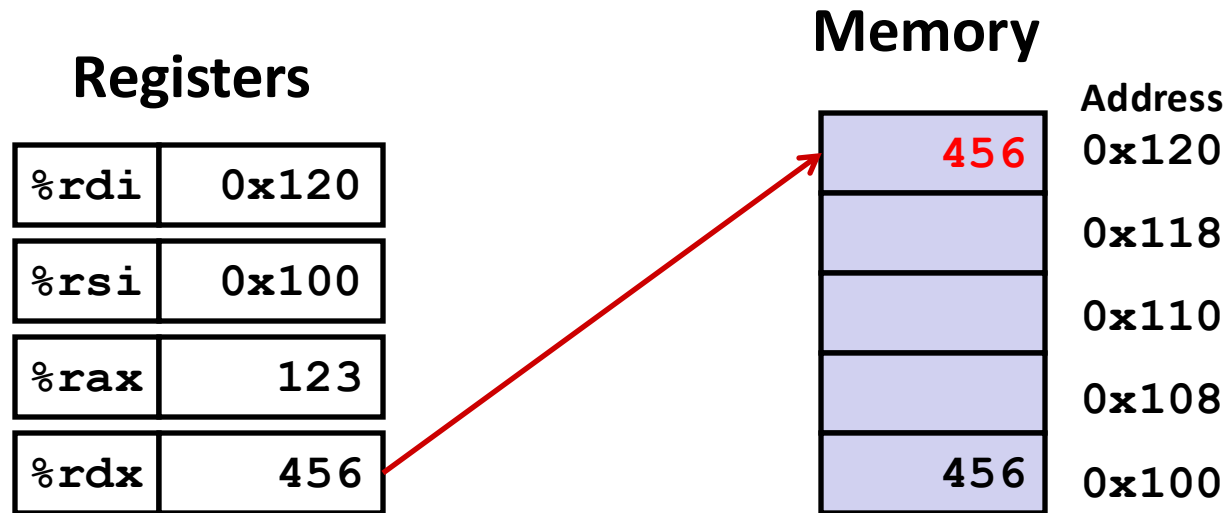
swap:

```

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

```

# Understanding Swap()



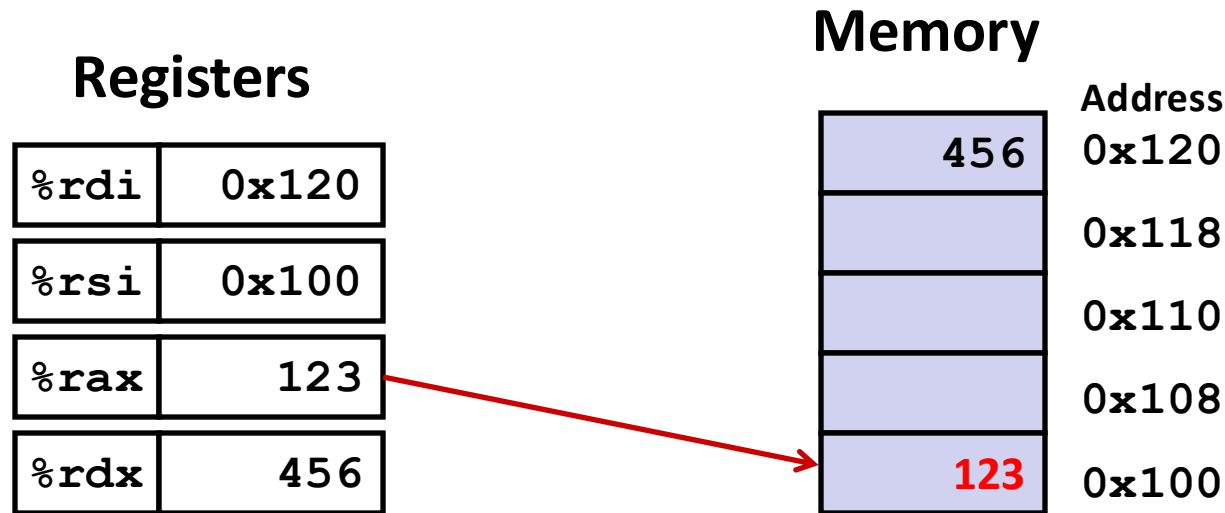
swap:

```

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

```

# Understanding Swap()



swap:

```

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

```

# More memory grammar

```
movq    (%rdi) ,           %rax  
    x = *rdi
```

```
movq    (%rdi, %rsi) ,     %rax  
    x = *(rdi + rsi)
```

```
movq    (%rdi, %rsi, 4) ,   %rax  
    x = *(rdi + 4 * rsi)
```

```
movq    0x80(%rdi, %rsi, 4) , %rax  
    x = *(rdi + 4 * rsi + 0x80)
```

# The memory ops make arrays easy

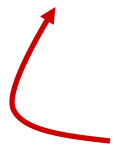
```
char *d;  
int y;  
char a = '\\0';  
d[y] = a;
```

```
movb    %rax, (%rdi, %rsi)
```

# When array type is larger than a byte

```
int *d;  
int s;  
char a = 67;  
d[y] = a;
```

```
movl    %eax, (%rdi, %rsi, 4)
```

  
sizeof(int) == 4

# Generalized memory grammar

**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

**If a value is “missing”, it is the identity element**

- +0 or \*1

# Address Computation Examples

|                   |                     |
|-------------------|---------------------|
| <code>%rdx</code> | <code>0xf000</code> |
| <code>%rcx</code> | <code>0x0100</code> |

**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 |
|----------------------------|---------------------|---------|
| <code>0x8(%rdx)</code>     |                     |         |
| <code>(%rdx,%rcx)</code>   |                     |         |
| <code>(%rdx,%rcx,4)</code> |                     |         |
| <code>0x80(,%rdx,2)</code> |                     |         |

# More Verbs

## ■ Two operands (think datalab)

### *Format*

### *Computation*

|      |                  |                                            |
|------|------------------|--------------------------------------------|
| addq | <i>Src, Dest</i> | $\text{Dest} = \text{Dest} + \text{Src}$   |
| salq | <i>Src, Dest</i> | $\text{Dest} = \text{Dest} \ll \text{Src}$ |
| sarq | <i>Src, Dest</i> | $\text{Dest} = \text{Dest} \gg \text{Src}$ |
| shrq | <i>Src, Dest</i> | $\text{Dest} = \text{Dest} \gg \text{Src}$ |

...

*Also called shlq*

*Arithmetic*

*Logical*

## ■ One operand (also datalab)

|      |             |                                  |
|------|-------------|----------------------------------|
| incq | <i>Dest</i> | $\text{Dest} = \text{Dest} + 1$  |
| decq | <i>Dest</i> | $\text{Dest} = \text{Dest} - 1$  |
| negq | <i>Dest</i> | $\text{Dest} = -\text{Dest}$     |
| notq | <i>Dest</i> | $\text{Dest} = \sim \text{Dest}$ |

# 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  $x + k * y$ 
  - $k = 1, 2, 4, \text{ or } 8$

## ■ 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
```

# Registers have conventions

## ■ Architectural requirements:

- **%rip** – program counter / instruction pointer
  - Only points to the next instruction to execute
- **%rsp** – stack pointer
  - **%rbp** – sometimes also stack related

## ■ “Calling Convention” (must be agreed upon for interop!)

- **%rax** – return value
- **%rdi** – first argument
- **%rsi** – second argument
- **%rdx** – third argument
- ...

# Putting it Together

## ■ Arguments?

- How many?
- What type(s)?

## ■ Return value?

## ■ Local variables?

```
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

# 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 <b>x</b>         |
| %rsi     | Argument <b>y</b>         |
| %rdx     | Argument <b>z</b> ,<br>t4 |
| %rax     | t1, t2, rval              |
| %rcx     | t5                        |

# Quiz Time!

Check out:

<https://canvas.cmu.edu/courses/55964/quizzes/169245>

# Expressing Inequality

- **We want to express numeric relations in assembly**
  - Equals, not equals
  - Greater, less than

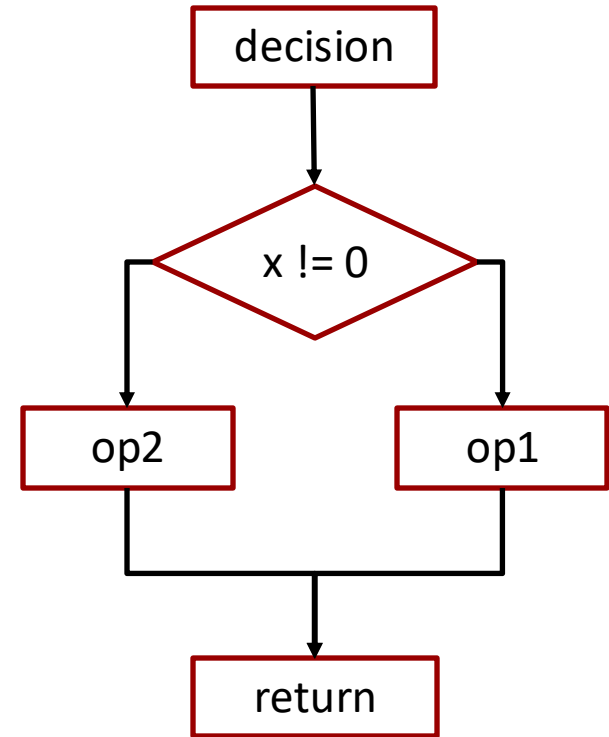
# Control Flow

- **We also need assembly to make decisions based on these (in)equalities**
  - We call the sequence of instructions executed, the control flow

# Control flow

```
extern void op1(void);  
extern void op2(void);
```

```
void decision(int x) {  
    if (x) {  
        op1();  
    } else {  
        op2();  
    }  
}
```



# Control flow in assembly language

```
extern void op1(void);  
extern void op2(void);  
  
void decision(int x) {  
    if (x) {  
        op1();  
    } else {  
        op2();  
    }  
}
```

```
decision:  
    testl    %edi, %edi  
    je       .L2  
    call     op1  
    jmp      .L1  
    .L2:  
    call     op2  
    .L1:  
    ret
```



It's all done with  
GOTO!

# Conditional “Goto”

- How did the example work?
- The first jump instruction is conditional
  - It uses processor state set by the **test** instruction
  - Processor has special registers to hold specific state
- We call this specific state, “condition codes”

# Expressing Inequality

- x86 has two instructions to set the specific state
  - `cmp`
  - `test`
- Instructions other than `leal` also implicitly set the state on x86

# Compare Instruction

- `cmp a, b`
  - Computes  $b - a$  (just like **sub**)
  - Sets condition codes based on result, but...
  - **Does not change  $b$**
- Used for **if** (**a < b**) { ... }  
whenever  $b - a$  isn't needed for anything else

# Test Instruction

## ■ `test a, b`

- Computes  $b \& a$  (just like **and**)
- Sets condition codes (only SF and ZF) based on result, but...
- **Does not change  $b$**
- Most common use: `test %rX, %rX`  
to compare `%rX` to zero
- Second most common use: `test %rX, %rY`  
tests if any of the 1-bits in `%rY` are also 1 in `%rX` (or vice versa)

# Jumping

## ■ jX Instructions

- Jump to different part of code depending on condition codes
- jmp – unconditional
- Z
  - ZERO
- GE
  - Greater than or equal

# Jumping (reference)

## ■ jX Instructions

- Jump to different part of code depending on condition codes

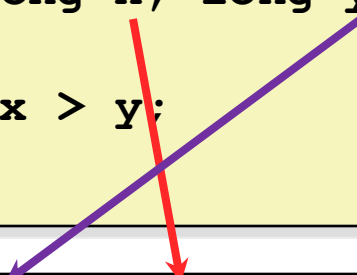
| jX  | Condition                            | Description               |
|-----|--------------------------------------|---------------------------|
| jmp | 1                                    | Unconditional             |
| je  | ZF                                   | Equal / Zero              |
| jne | $\sim ZF$                            | Not Equal / Not Zero      |
| js  | SF                                   | Negative                  |
| jns | $\sim SF$                            | Nonnegative               |
| jg  | $\sim (SF \wedge OF) \ \& \ \sim ZF$ | Greater (Signed)          |
| jge | $\sim (SF \wedge OF)$                | Greater or Equal (Signed) |
| jl  | $(SF \wedge OF)$                     | Less (Signed)             |
| jle | $(SF \wedge OF) \   \ ZF$            | Less or Equal (Signed)    |
| ja  | $\sim CF \ \& \ \sim ZF$             | Above (unsigned)          |
| jb  | CF                                   | Below (unsigned)          |

# Returning Condition Codes

## ■ setX will

- Set low-order byte of destination to 0 or 1 based on *combinations* of condition codes
- Does not alter remaining 7 bytes

```
int gt (long x, long y)
{
    return x > y;
}
```



```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al           # Set when >
movzbl   %al, %eax     # Zero rest of %rax
ret
```

# movXYZ – Moving to larger bit widths

## ■ mov with three suffixes will move to larger bit widths

- X – {s,z}
  - Sign extend
  - Zero extend
- Y – source bit width
- Z – destination bit width

## ■ `movzbl %al, %eax`

- z – zero
- b – source is 1 byte
- l – destination is 4 bytes

# Summary

## ■ Nouns

- Registers

## ■ Verbs

- Instructions or operations

## ■ Suffixes and Annotations

- Specify the size (usually optional)
- Memory addressing mode

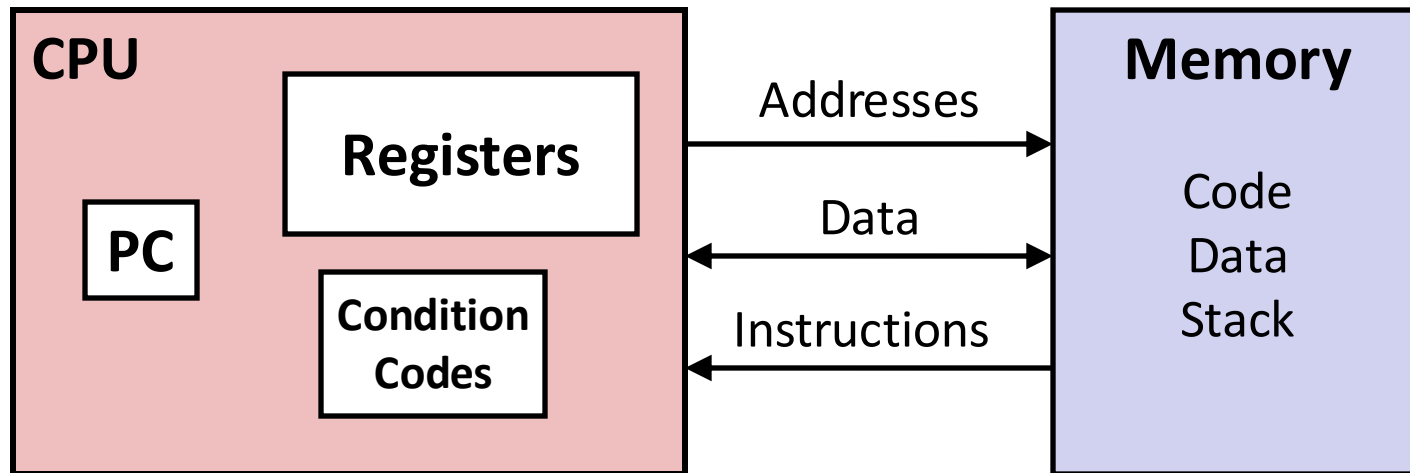
## ■ Relations

- Relies on condition codes and conditional jump (i.e., goto)

# 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

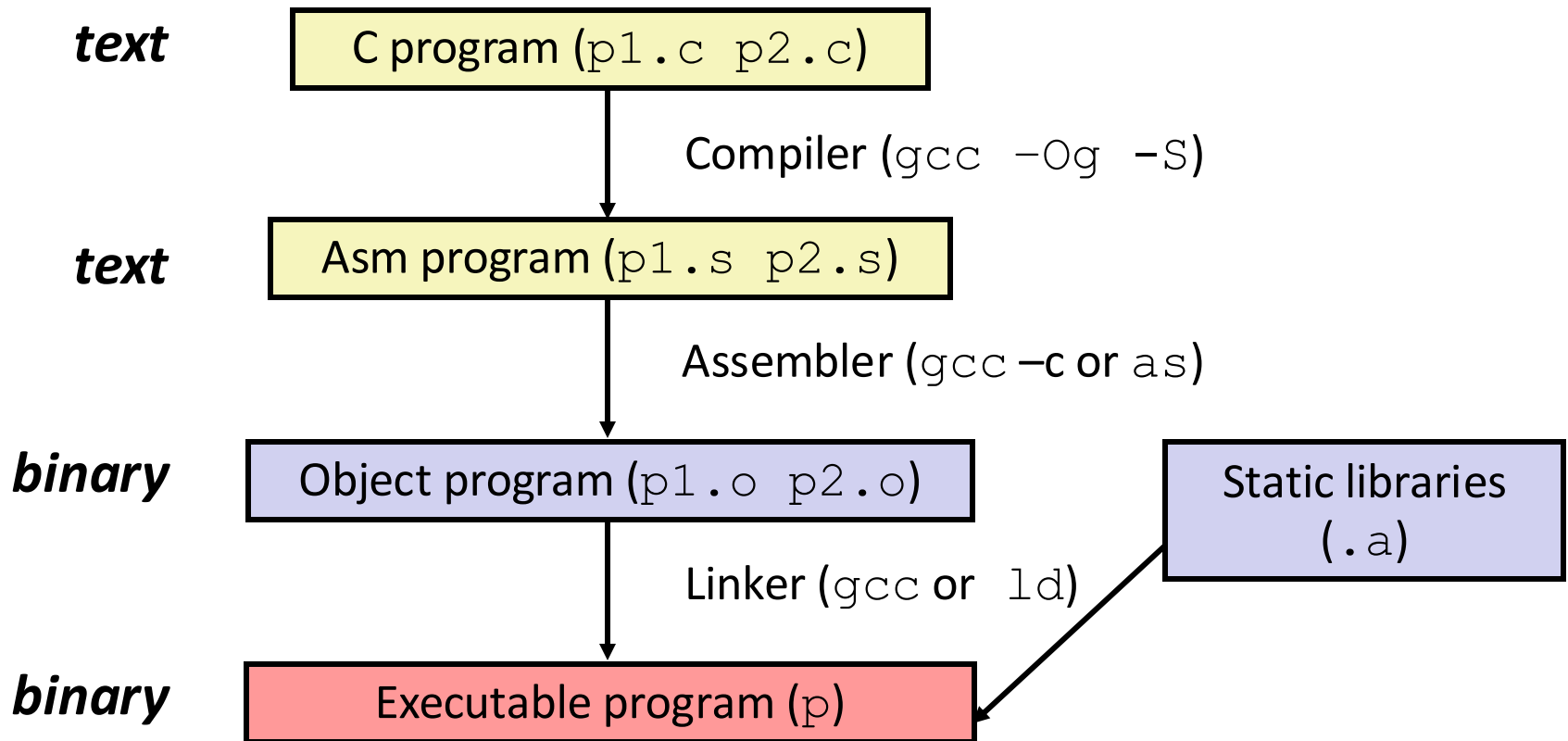
- Byte addressable array
- Code and user data
- Stack to support procedures

# Where does it come from?

- **Assembly is usually generated by the compiler**
  - We can ask the compiler to show us the assembly
  - We can also generate assembly from machine code
- **The compiler runs a separate tool that generates machine code**
  - Machine code is just bytes in memory
- **Execution gives bytes their “types”**

# Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use debugging-friendly optimizations (`-Og`)
  - 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 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

```

        .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

```

Things that look weird  
and are preceded by a ‘  
are generally directives

```

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

```

# 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

## 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

## Object Code

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

## Disassembled

Dump of assembler code for function sumstore:

```
0x00000000000400595 <+0>: push    %rbx
0x00000000000400596 <+1>: mov     %rdx,%rbx
0x00000000000400599 <+4>: callq  0x400590 <plus>
0x0000000000040059e <+9>: mov     %rax, (%rbx)
0x000000000004005a1 <+12>: pop     %rbx
0x000000000004005a2 <+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:
```

```
30001005:
```

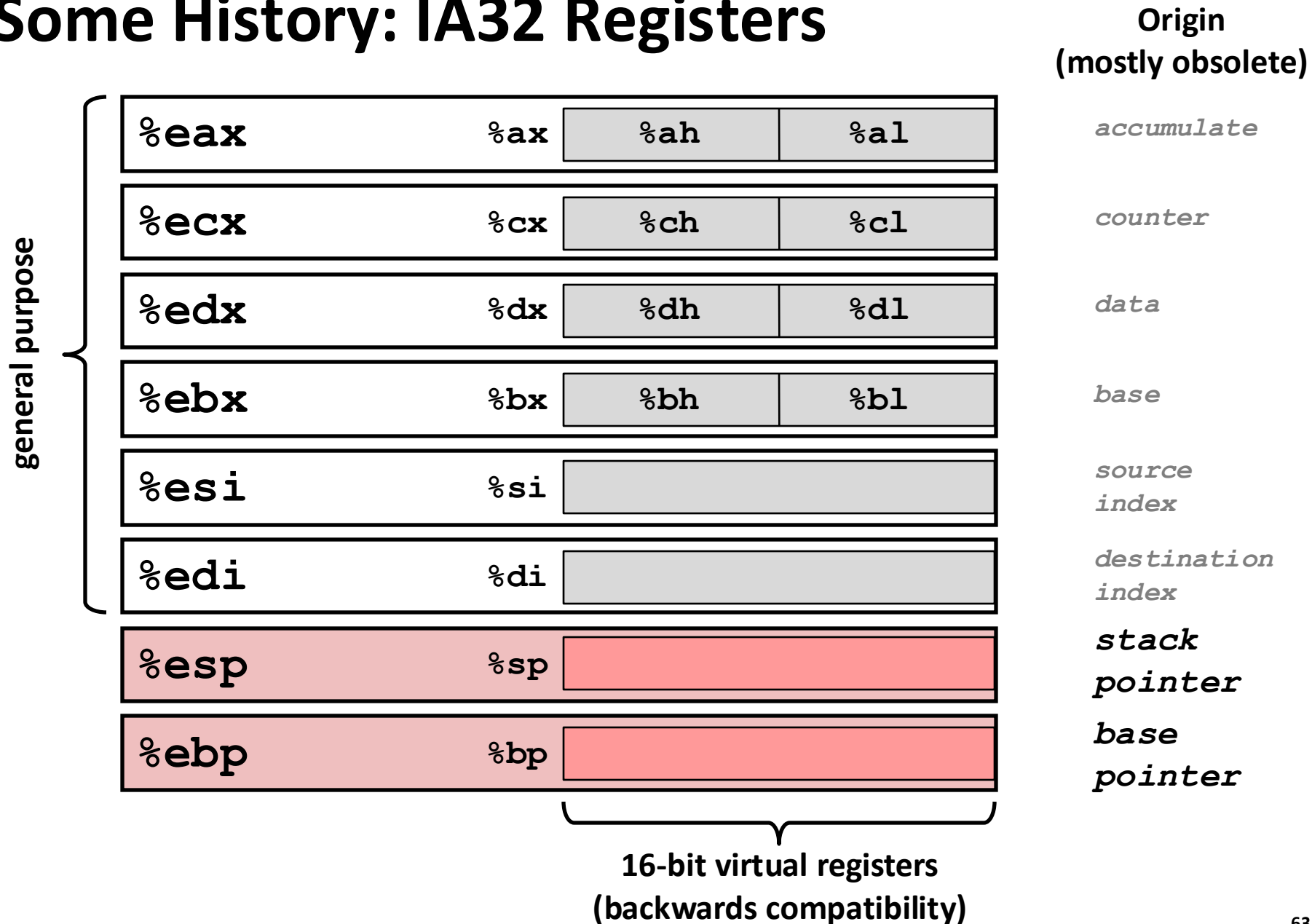
```
3000100a:
```

**Reverse engineering forbidden by  
Microsoft End User License Agreement**

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

**ADDITIONAL SLIDES**

# Some History: IA32 Registers



# 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

# History: 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.

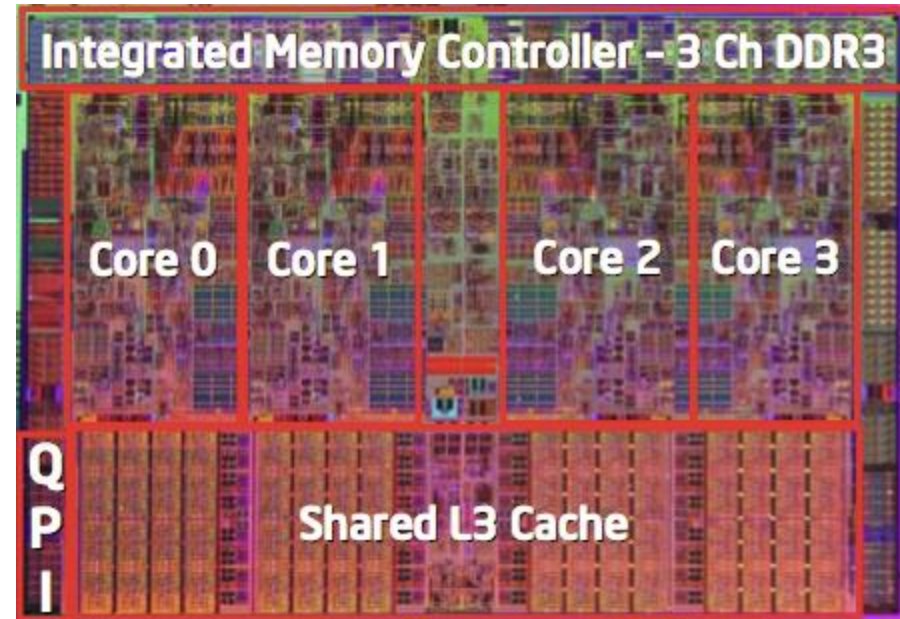
# Intel x86 Evolution: Milestones

| <i><b>Name</b></i>                                                                                                                                             | <i><b>Date</b></i> | <i><b>Transistors</b></i> | <i><b>MHz</b></i> |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|---------------------------|-------------------|
| ■ <b>8086</b>                                                                                                                                                  | <b>1978</b>        | <b>29K</b>                | <b>5-10</b>       |
| <ul style="list-style-type: none"> <li>First 16-bit Intel processor. Basis for IBM PC &amp; DOS</li> <li>1MB address space</li> </ul>                          |                    |                           |                   |
| ■ <b>386</b>                                                                                                                                                   | <b>1985</b>        | <b>275K</b>               | <b>16-33</b>      |
| <ul style="list-style-type: none"> <li>First 32 bit Intel processor , referred to as IA32</li> <li>Added “flat addressing”, capable of running Unix</li> </ul> |                    |                           |                   |
| ■ <b>Pentium 4E</b>                                                                                                                                            | <b>2004</b>        | <b>125M</b>               | <b>2800-3800</b>  |
| <ul style="list-style-type: none"> <li>First 64-bit Intel x86 processor, referred to as x86-64</li> </ul>                                                      |                    |                           |                   |
| ■ <b>Core 2</b>                                                                                                                                                | <b>2006</b>        | <b>291M</b>               | <b>1060-3333</b>  |
| <ul style="list-style-type: none"> <li>First multi-core Intel processor</li> </ul>                                                                             |                    |                           |                   |
| ■ <b>Core i7</b>                                                                                                                                               | <b>2008</b>        | <b>731M</b>               | <b>1600-4400</b>  |
| <ul style="list-style-type: none"> <li>Four cores (our shark machines)</li> </ul>                                                                              |                    |                           |                   |

# 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 |
|-------------------------------|------|--------------------|
| ■ 1 <sup>st</sup> Pentium Pro | 1995 | 600 nm             |
| ■ 1 <sup>st</sup> Pentium III | 1999 | 250 nm             |
| ■ 1 <sup>st</sup> Pentium 4   | 2000 | 180 nm             |
| ■ 1 <sup>st</sup> Core 2 Duo  | 2006 | 65 nm              |

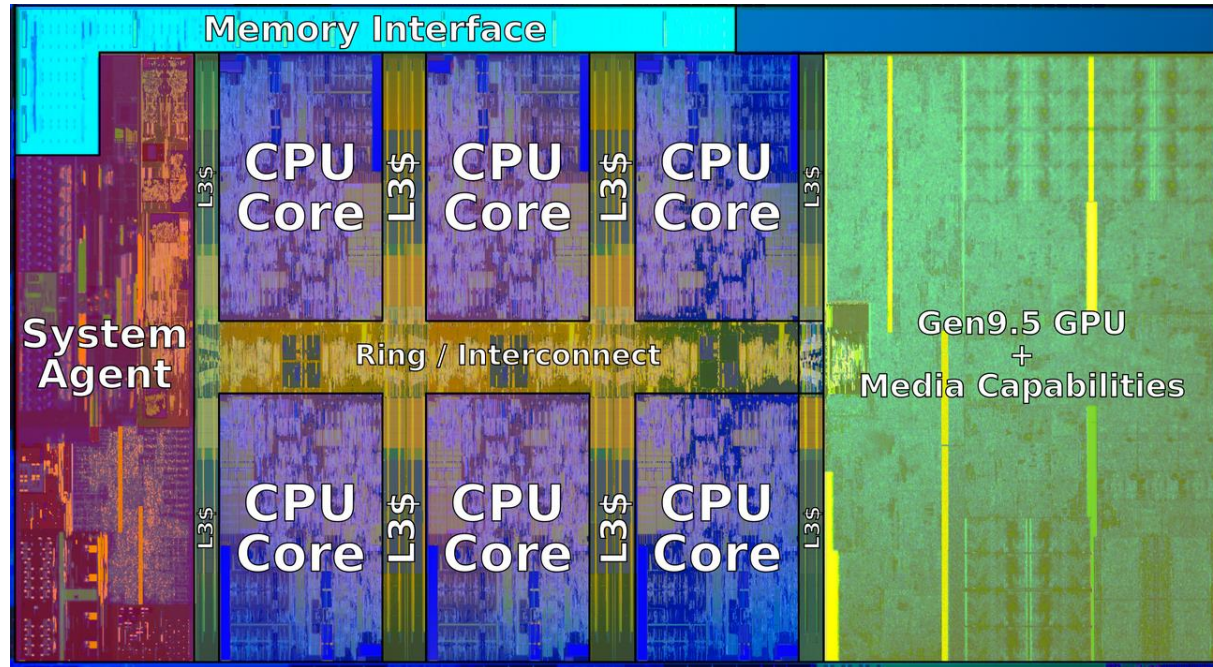
## ■ Recent & Upcoming Generations

|     |              |      |                     |
|-----|--------------|------|---------------------|
| 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               |
| 12. | Alder Lake   | 2022 | “intel 7” (10nm+++) |

Process technology dimension  
= width of narrowest wires  
(10 nm  $\approx$  100 atoms wide)

(But this is changing now.)

# 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: Spun off GlobalFoundries
- 2019-20: Pulled ahead! Used TSMC for part of fab
- 2022: Intel re-took the lead

# 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