Imperial College London

113: Architecture

Spring 2018

Lecture: Machine-Level Programming I

Instructor: Dr. Jana Giceva

Textbooks

- ***"Computer systems: A programmer's perspective"***
  Randal E. Bryant and David O'Hallaron, 2013 (3^{rd} edition)

  Available as e-book in the Imperial library
  Website: http://csapp.cs.cmu.edu

  Very important for the course:
  practice problems relevant to exam questions

- ***Intel 64 and IA-32 manuals*** for extra reference
- Not mandatory: any good C book

Hardware/software interface

- What is software? What is hardware? Why do we need HW/SW interface?
- Why do we need to understand both sides of this interface?

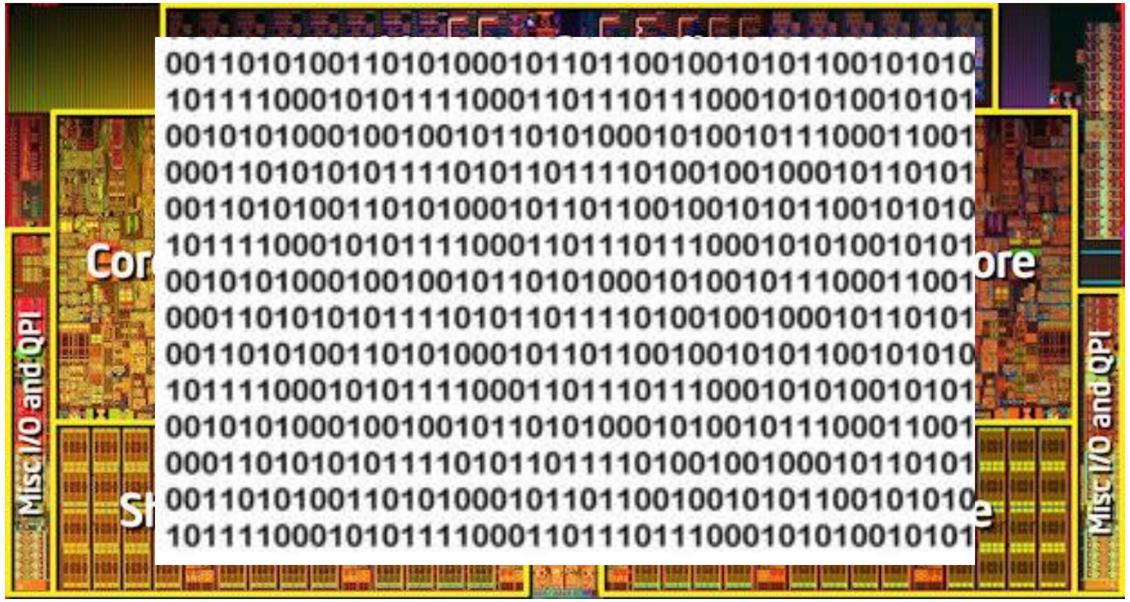
```java
public class DebuggingDemo {

    int counter;

    public void increment()
    {
        int temp = counter;
        temp ++;
        counter = temp;
    }

    public static void main(String[] args) {
        DebuggingDemo d = new DebuggingDemo();
        d.setCounter(1);
        d.increment();
        System.out.println(d.counter);
    }

    public void setCounter(int p_initialValue)
    {
        counter = p_initialValue;
    }
}
```



C/Java, assembly, and machine code

C / Java

```
if  (x  !=  0){
     y  =  (y+z)/x;
}
```

Assembly

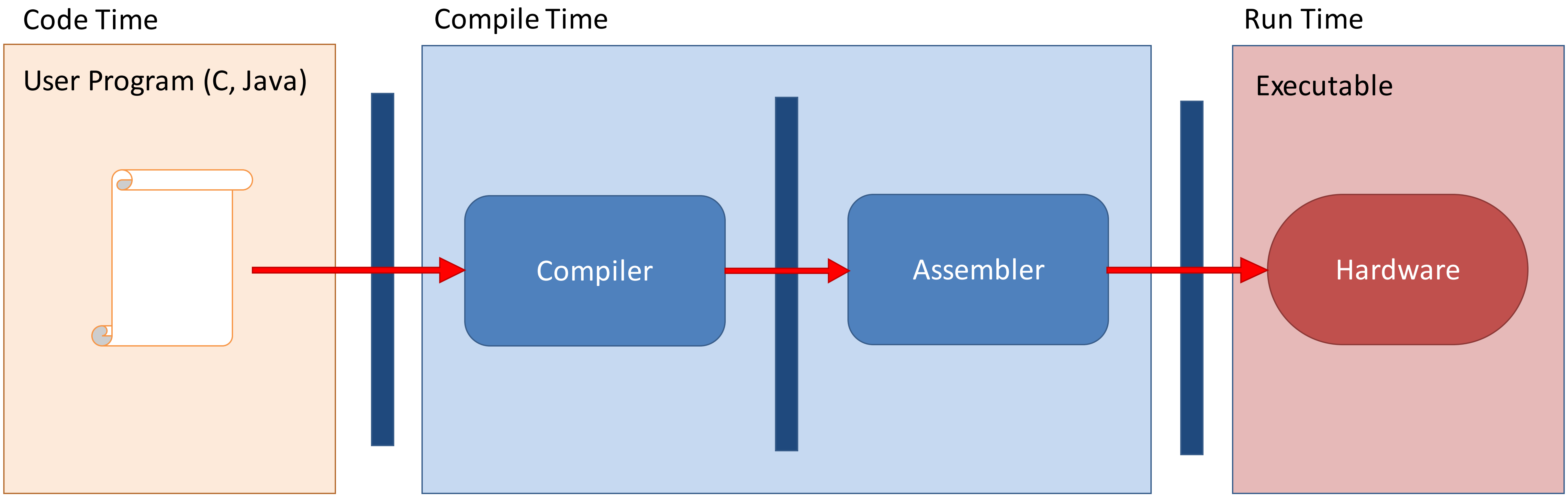
```
     cmpl    $0,-4(%rbp)
     je      .L2
     movl    -8(%rbp), %edx
     movl    -12(%rbp), %eax
     addl    %edx, %eax
     cltd
     idivl   -4(%rbp)
     move    %eax, -8(%rbp)
.L2:
```

Machine code

```
1000001101111100001001000001110000000000
0 0111010000011000
10001011010001000010010000010100
10001011010001100010010100010100
100011010000010000000010
1000100111000010
110000011111101000011111
11110111011111000010010000011100
10001001010001000010010000011000
```

- All program fragments are equivalent
- You'd rather write C / Java (more human friendly)
- Hardware executes strings of bytes

Code / Compiler / Run time



- Note: The compiler and assembler are just programs (developed using the same principle).

Roadmap – Overview of material

- Memory and data
- Machine code and x86 assembly language
- Procedures and stacks
- Arrays, structs and objects
- Floating point numbers and representation
- Memory architecture and caches
- Processes, interrupts and exceptions

Writing assembly code? In 2018??

- You are probably never going to write a program in assembly
  - compilers are much better and more patient
  - unless you write a delicate, “special” code
- But, understanding assembly is ***key*** to the machine level execution model
  - behaviour of programs in the presence of ***bugs***
    - High-level language model breaks down
  - tuning program ***performance***
    - understand the optimisations done / not done by the compiler
    - understanding sources of program inefficiency
  - implementing ***system software*** (e.g., Operating Systems, Compilers)
  - creating / fighting ***malware***

Memory matters!

- RAM is an unrealistic abstraction:
  - Memory is not unbounded
  - Memory performance is not uniform (caches, virtual memory, etc.)
  - Memory referencing bugs are especially tricky (working with pointers in C).

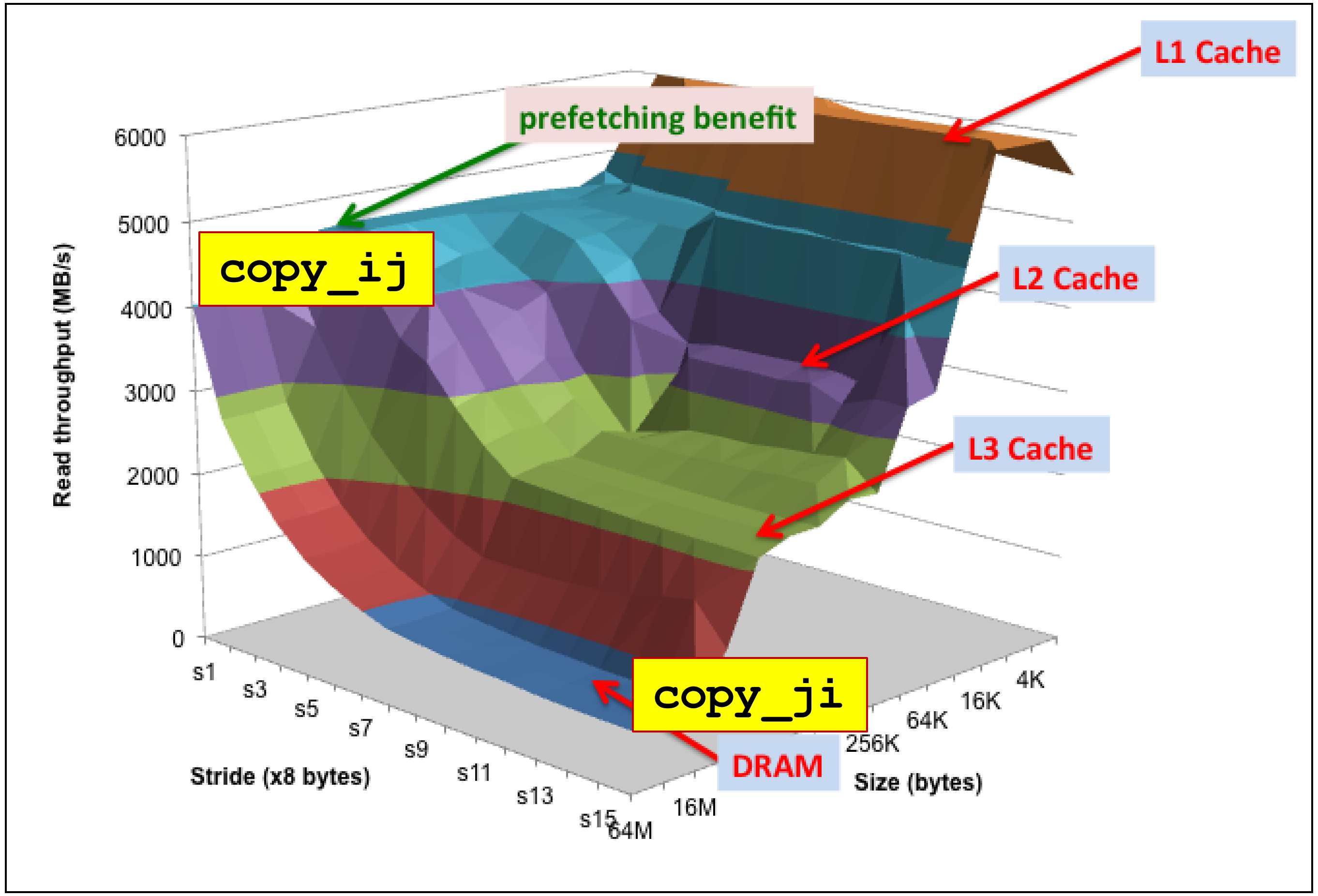
```
void copy_ij(int src[2048][2048],
             int dst[2048][2048]) {
    int i, j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

5.2 ms

```
void copy_ji(int src[2048][2048],
             int dst[2048][2048]) {
    int i, j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

162 ms

The memory mountain



Today: Machine Programming – Basics

- **What is an ISA?**
- History of x86
- The assembler language in context
- Introduction to assembly: Registers, operands
- Instruction format
- Memory addressing modes

Instruction Set Architecture (ISA)

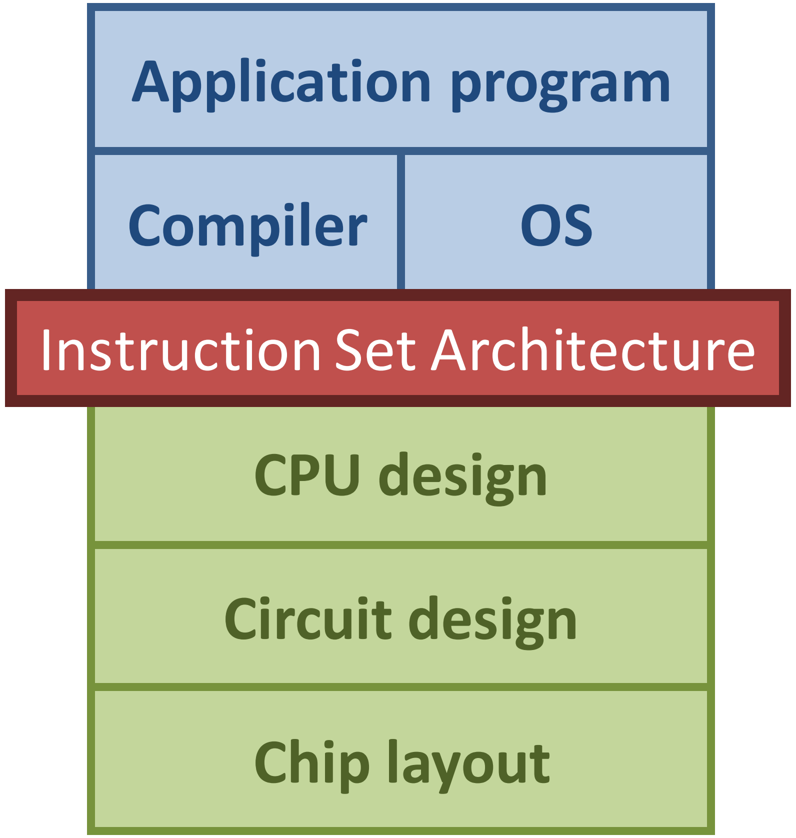
- ***Definition 1***
  Architecture (Instruction Set Architecture (ISA)) the parts of a processor design that one needs to understand to write assembly.

- ***Definition 2***
  Microarchitecture is the implementation of the architecture.

- Example ISAs: x86, MIPS, ia64, VAX, Alpha, ARM, etc.

Instruction Set Architecture – big picture

- **Assembly language view**
  - Processor state
    - Registers, memory, etc.
  - Instructions
    - **`addl`**, **`movq`**, **`leal`**, etc.
    - How instructions are encoded as bytes
- **Layer of Abstraction**
  - Above: how to program a machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use various tricks to make it run fast



Today: Machine Programming – Basics

- What is an ISA?
- **Why x86?**
- The assembler language in context
- Introduction to assembly: Registers, operands
- Instruction format
- Memory addressing modes

Intel x86 Processors

- **The x86 architecture dominates the laptop/desktop/server market**

- **Evolutionary design**
  - Backwards compatible up until 8086 (introduced in 1978)
  - Added more features as time goes on

- **Complex instruction set computer (CISC)**
  - Many different instructions with many different formats
  - Hard to match performance of RISC
  - But, Intel has done just that!

Intel x86 Evolution: Key Milestones

Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**

- First 16-bit processor. Basis for IBM PC & DOS
- 1 MB address space

386	**1985**	**275K**	**16-33**

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

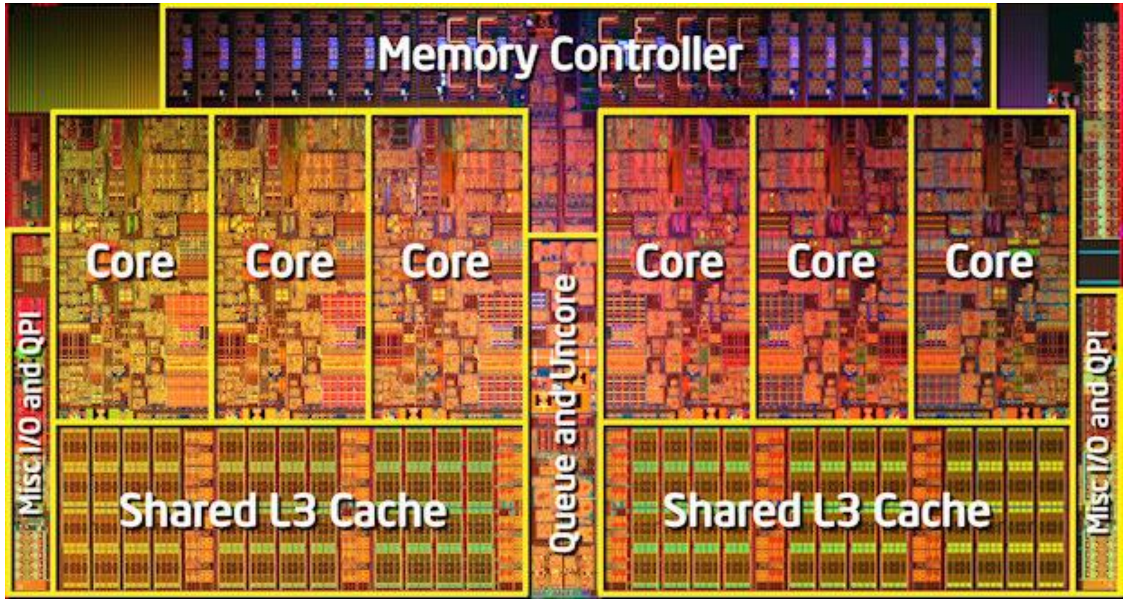
Pentium 4E	**2004**	**125M**	**2800-3800**

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

Intel x86 Processors: Overview

- **Machine evolution, examples:**

486	1989	1.9M
Pentium	1993	3.1M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium IV	2001	42M
Core 2 Duo	2006	291M
Xeon 7400	2008	1.9B
Xeon i7	2012	4.3B



- **Added features:**
  - transition from 32 bits to 64 bits
  - more cores
  - instructions to support multimedia operations, more efficient conditional ops

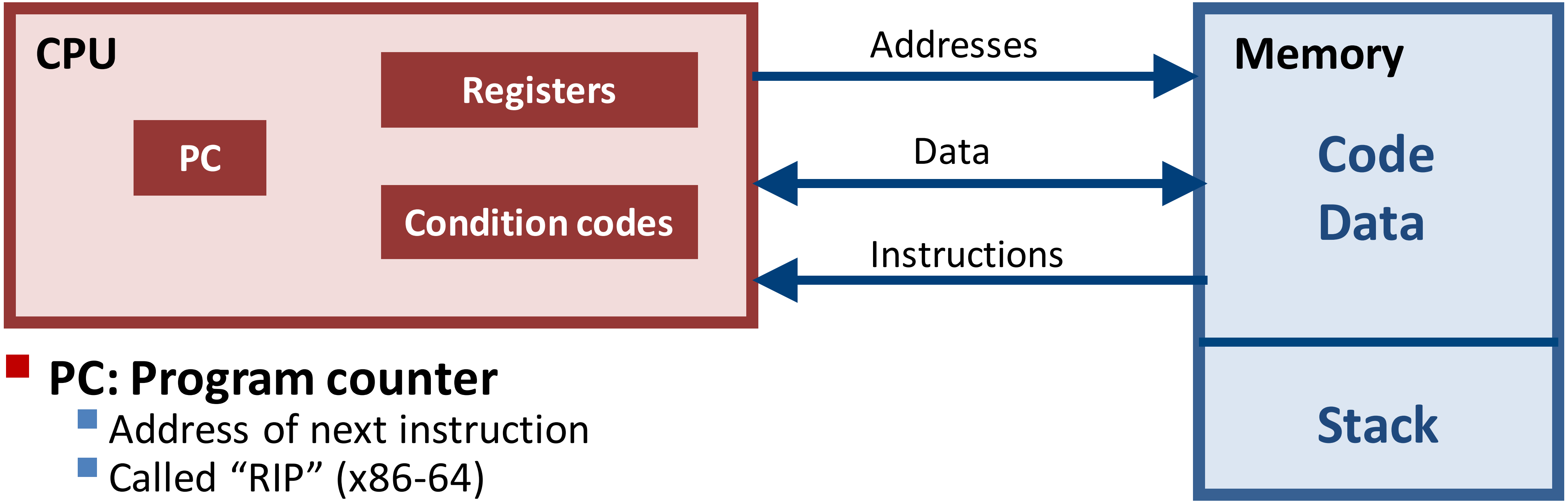
A quick note on syntax

- There are two common ways to write x86 assembler code:
  - AT&T syntax
    - We are going to use it in this course
    - Common on Unix
  - Intel syntax
    - Generally used for Windows machines

Today: Machine Programming – Basics

- What is an ISA?
- History of x86
- **The assembler language in context**
- Introduction to assembly: Registers, operands
- Instruction format
- Memory addressing modes

Assembly/Machine code view



- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Frequently used program data
- **Condition codes**
  - Status information about recent arithmetic or logical operations
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - code and user data
  - stack to support procedures

Compiling into Assembly

C code

```
int sum(int x, int y) {
    int t = x + y;
    return t;
}
```

Generated x86 assembly

```
sum:
        pushq     %rbp
        movq      %rsp, %rbp
        movl      %edi, -20(%rbp)
        movl      %esi, -24(%rbp)
        movl      -24(%rbp), %eax
        movl      -20(%rbp), %edx
        addl      %edx, %eax
        movl      %eax, -4(%rbp)
        movl      -4(%rbp), %eax
        popq      %rbp
        ret
```

- Running the command:

```
gcc –O –S code.c
```

- Produces file: `code.s`

Assembly characteristics: *Data Types* and *Instructions*

- **Data Types:**
  - Integers (1, 2, 4, 8 byte), Floating point (later in the course), no arrays/structs
- **Perform arithmetic function on register or memory data**
- **Transfer data between memory and register**
  - Load data from memory into a register
  - Store register data into memory
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches

Machine instruction example

```
int t = x + y;
```

```
addl 8(%rbp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *rbp;
eax += rbp[2]
```

```
0x401046:   03 45 08
```

- **C code**
  - Add two integers
- **Assembly**
  - Add two 4-byte integers
    - “long” words in GCC parlance
  - Operands:
    - `x`: register `%eax`
    - `y`: memory `M[%rbp+8]`
    - `t`: register `%eax`
  - Return function value in `%eax`
- **Object code**
  - 3-byte instruction
  - stored at address `0x401046`

Today: Machine Programming – Basics

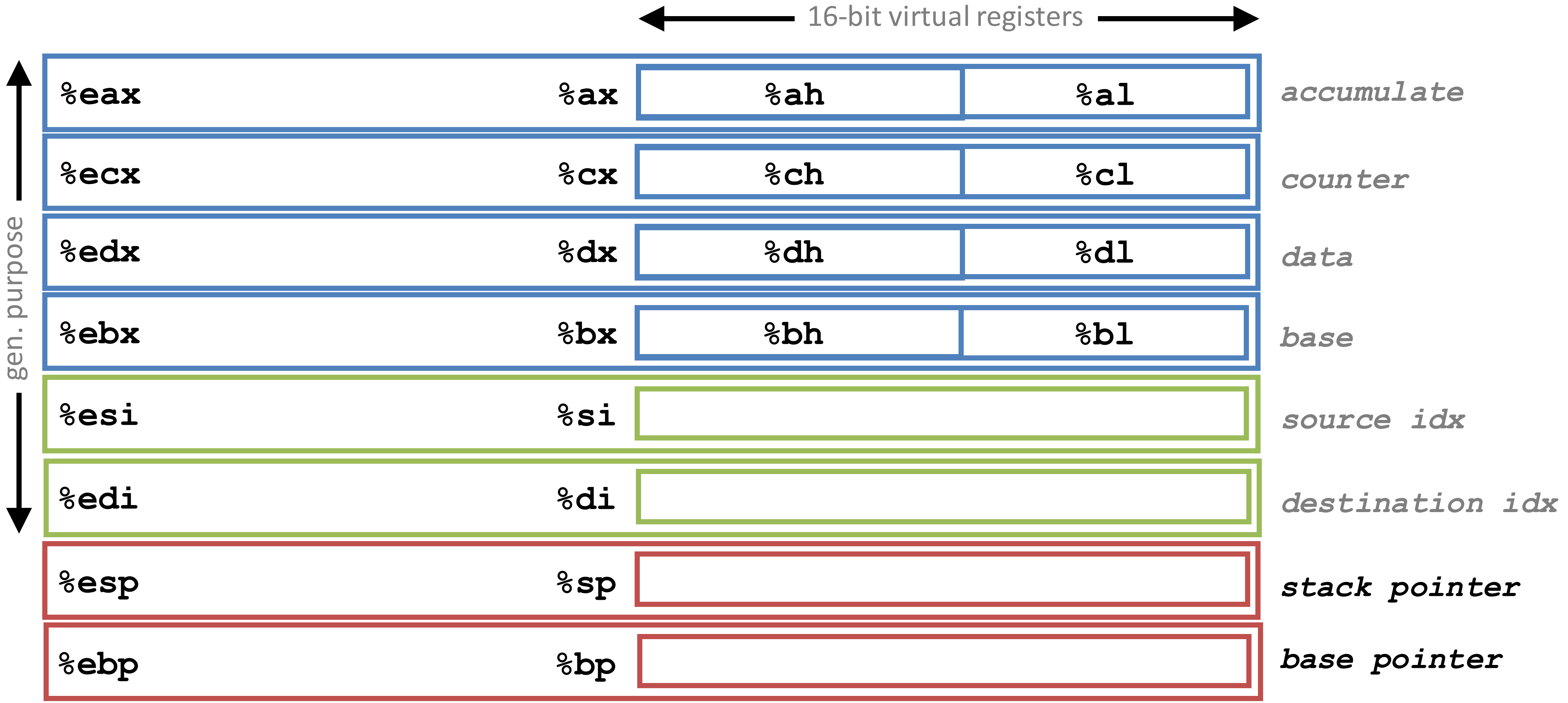
- What is an ISA?
- History of x86
- The assembler language in context
- **Introduction to assembly: registers, data formats, operands**
- Instruction format
- Memory addressing modes

What is a register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle).

- Registers are at the heart of assembly programming.
  - They are a precious commodity in all architectures, but especially x86.

History: IA32 Integer Registers



x86-64 Integer Registers

general purpose

%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

Data Formats

- Due to its origins as a 16-bit architecture, Intel uses ***word*** to refer to a 16-bit data type.
- 32-bit quantities are ***double words***, and 64-bit are ***quad words***.

C declaration	Intel data type	Assembly code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long int`	Double word	`l`	4
`char*`	Double word	`l`	4
`float`	Single precision	`s`	4
`double`	Double precision	`q`	8
`long double`	Extended precision	`t`	10/12

Instruction format

- Most Intel assembly ***instructions*** have at least one ***operand***

```
label:    opcode    source, destination    ;comments
label:    opcode    operand                ;comments
```

- ***label*** is an optional user-defined identifier for the address of the instruction or data item which follows.

- The ***operands*** specify the source to reference in performing the operation and the destination location where to place the result.

Operand Types

1. ***Immediate*** for constant values

Format: $\$Imm$ Example: `$-536, $0x1F`

2. ***Register*** for the contents of one of the registers

Format: register E_a, referenced value is then $\mathtt{R}[E_a]$ Example: `%rax,%eax`

3. ***Memory reference*** to access a memory location based on a computed address

Format: memory address $Addr$, referenced value is then $\mathtt{M}[Addr]$

Moving data

- **Moving Data**
  `movx` ***Source, Destination***

- **Operand Types**
  - ***Immediate:*** constant integer data
    - Example: **`$0x400, $-526`**
    - Encoded with 1,2, or 4 bytes
  - ***Register:*** one of 16 integer registers
    - Example: **`%rax, %r13`**
    - Note that **`%rsp`** is reserved for special use
  - ***Memory:*** 8 consecutive bytes of memory at address given by register

%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

`movl` operand combinations

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

Cannot do memory-to-memory transfer with a single instruction

Summary: Machine Programming – Basics

- What is an ISA?
- History of x86
- The assembler language in context
- Introduction to assembly: Registers, operands
- Instruction format
- **Memory addressing modes**

Memory operands:
Simple memory addressing modes

- **The operand is the value at the specified address**
- **Normal addressing mode**
  - An Immediate **(Imm)** or a Register **(R[E_a])** specifies the memory address

  Mem[Imm], Mem[R[E_a]]

  Example: **`movq (%rcx), %rax`**

- **Displacement**
  - An immediate **(Imm)** or a Register **(R[E_a])** specifies the start of a memory region
  - A constant displacement **D** specifies the offset

  Mem[Imm+D], Mem[R[E_a]+D]

  Example: **`movq 8(%rbp),%rdx`**

Example of Simple Addressing Mode

C code

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

Compiled to assembly (with optimization)

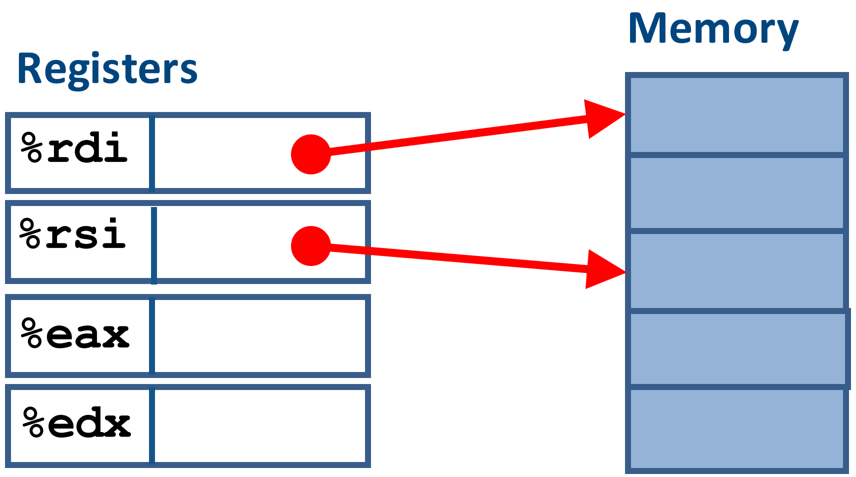
```
swap:
    movl (%rdi), %eax
    movl (%rsi), %edx
    movl %edx, (%rdi)
    movl %eax, (%rsi)
    ret
```

```
swap:
    pushq %rbp
    movq %rsp, %rbp
    movq %rdi, -24(%rbp)
    movq %rsi, -32(%rbp)
    movq -24(%rbp), %rax
    movl (%rax), %eax
    movl %eax, -4(%rbp)
    movq -32(%rbp), %rax
    movl (%rax), %eax
    movl %eax, -8(%rbp)
    movq -24(%rbp), %rax
    movl -8(%rbp), %edx
    movl %edx, (%rax)
    movq -32(%rbp), %rax
    movl -4(%rbp), %edx
    movl %edx, (%rax)
    nop
    popq %rbp
    ret
```

Example of Simple Addressing Mode

C code

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



Compiled to assembly (with optimization)

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

Example of Simple Addressing Mode

C code

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

Registers

%rdi	0x120
%rsi	0x110
%eax	
%edx	

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

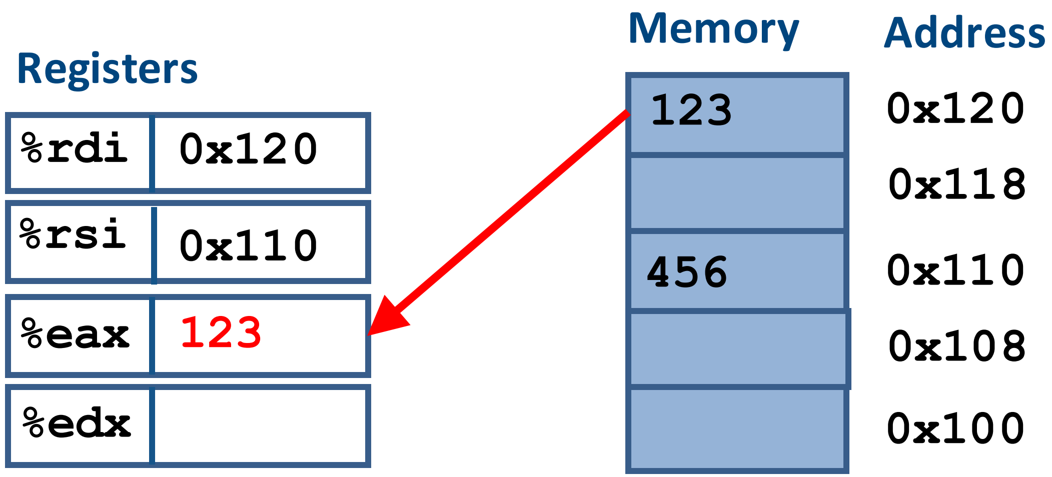
Compiled to assembly (with optimization)

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

Example of Simple Addressing Mode

C code

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



Compiled to assembly (with optimization)

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

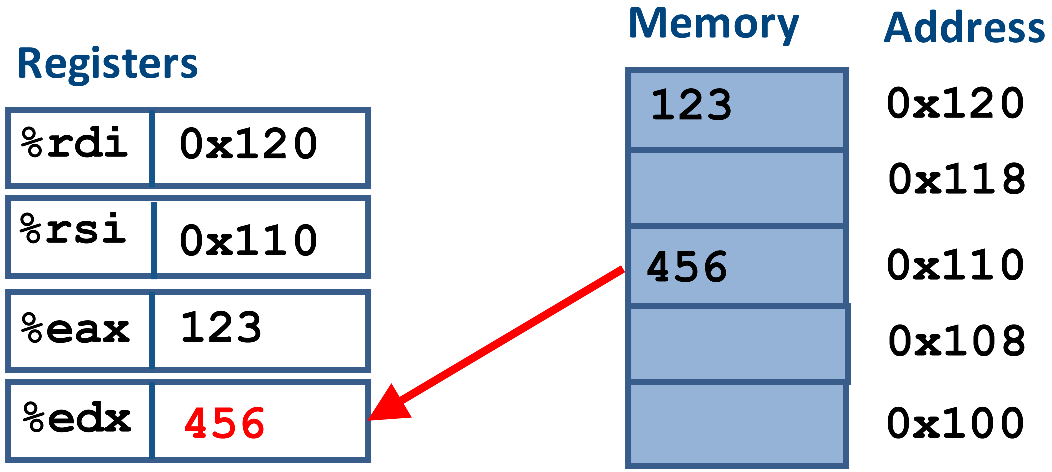
Example of Simple Addressing Mode

C code

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

Compiled to assembly (with optimization)

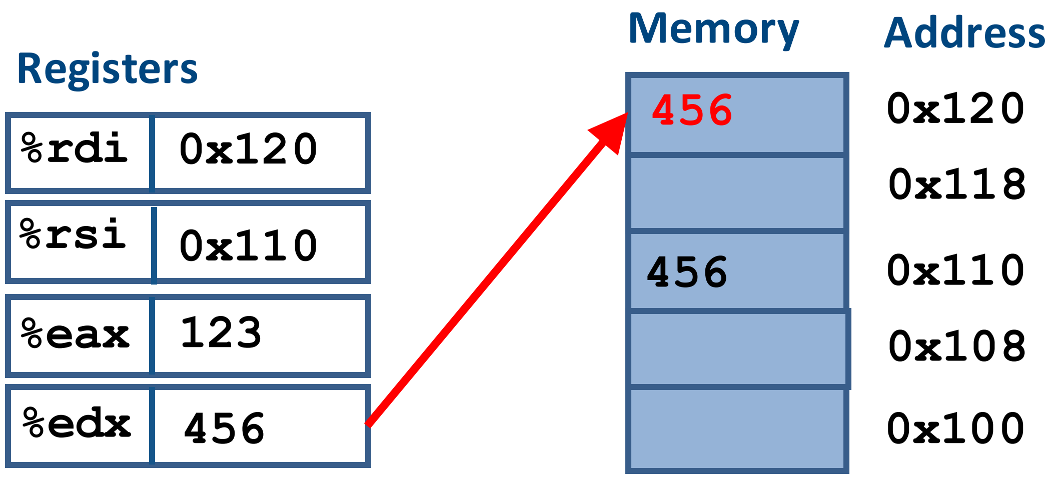
```
swap:
    movl (%rdi), %eax
    movl (%rsi), %edx
    movl %edx, (%rdi)
    movl %eax, (%rsi)
    ret
```



Example of Simple Addressing Mode

C code

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



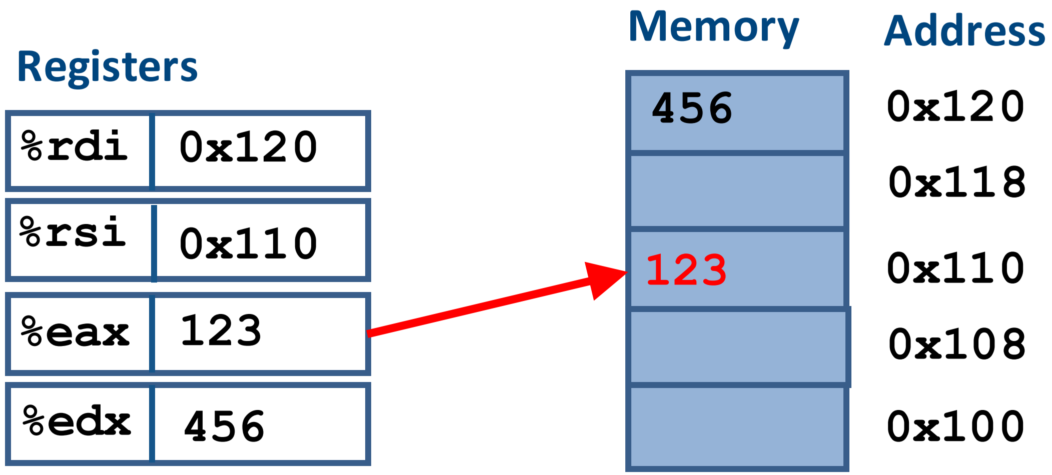
Compiled to assembly (with optimization)

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

Example of Simple Addressing Mode

C code

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



Compiled to assembly (with optimization)

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

Memory operands:
Complete memory addressing modes

- **Most General Form – D(Rb,Ri,S)**

  $\texttt{Mem[R[}E_b\texttt{]+s}\cdot\texttt{R[}E_i\texttt{]+ D]}$

where

- $\texttt{R[}E_b\texttt{]}$: is the base register: any of the 16 integer registers
- $\texttt{R[}E_i\texttt{]}$: index register: any register, except for **`%rsp`**
- **D** : constant displacement 1, 2, or 4 bytes (*why not 8?*)
- **S** : scale: 1, 2, 4, or 8 (*why these numbers?*)

Memory operands:

Special cases memory addressing modes

- **Base register + index register**
  - Can be used to access **array elements** when **start of array is dynamically determined**

Mem[R[E_b] + R[E_i]]

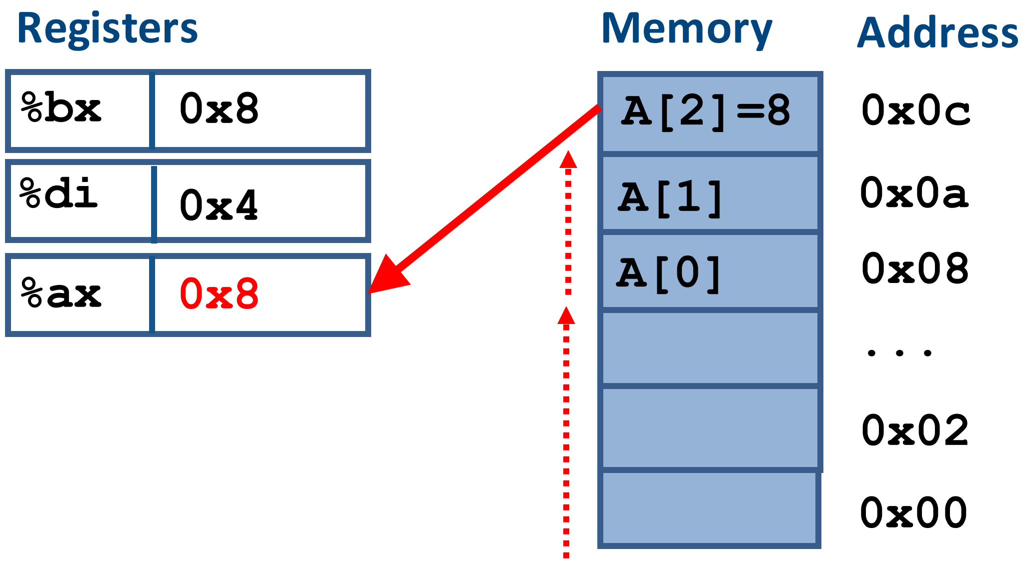
where

R[E_b] = the start of the array, and

R[E_i] = byte index of array element

Example assembly:

```
mov (%bx,%di), %ax
```



Memory operands:
Special cases memory addressing modes

- **Relative based index = base register + index register + disp**
  - Can be used to access **arrays of objects, arrays within objects,** and **arrays on the stack**

$$\mathtt{Mem[R[}E_b\mathtt{]\ +\ R[}E_i\mathtt{]\ +\ D]}$$

where

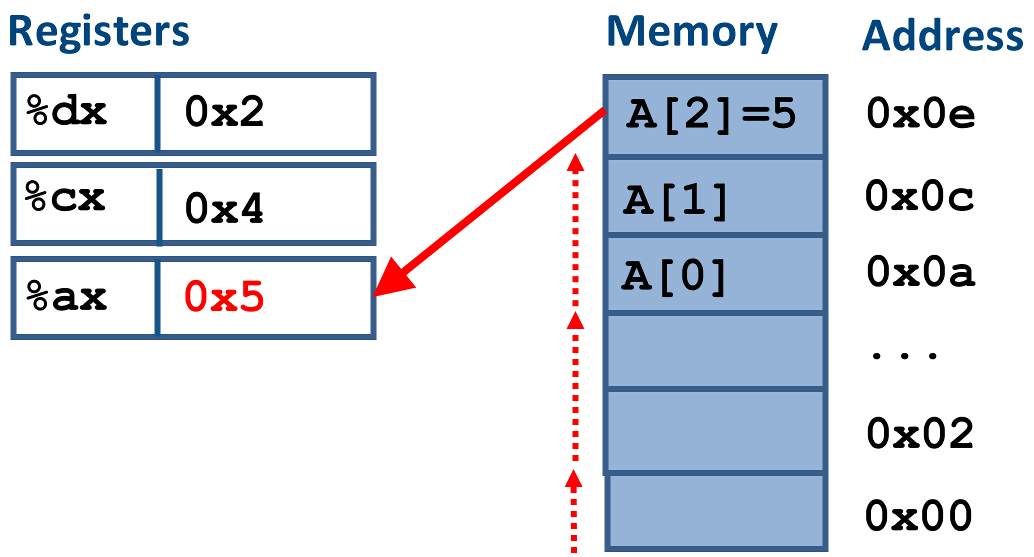
$\mathtt{R[}E_b\mathtt{]}$ = the start of the object

D = array field within the object

$\mathtt{R[}E_i\mathtt{]}$ = byte offset of array element

Example assembly:

```
mov 0x8(%dx,%cx), %ax
```



Memory operands:
Special cases memory addressing modes

- **Base register + (scale * index register) + displacement**
  - efficient access to **arrays elements within objects** and **on the stack** when the element size is 1, 2, 4 or 8 bytes

$$\mathtt{Mem}[\mathtt{R}[E_b] + \mathtt{s} \cdot \mathtt{R}[E_i] + \mathtt{D}]$$

where

$\mathtt{R}[E_b]$ = the start of the object

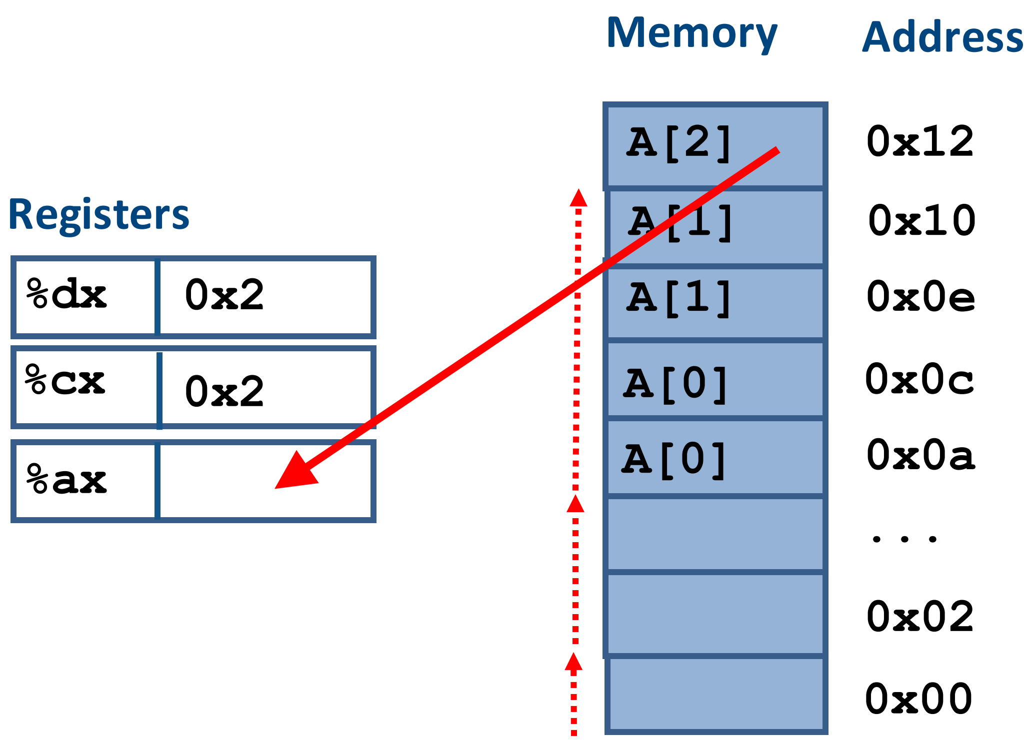
$\mathtt{D}$ = array field within the object

$\mathtt{R}[E_i]$ = "index" of array element

$\mathtt{s}$ = element size

Example assembly:

```
mov 0x8(%dx,%cx,4), %ax
```



Summary: Machine Programming – Basics

- What is an ISA?
- History of x86
- The assembler language in context
- Introduction to assembly: Registers, operands
- Instruction format
- Memory addressing modes

References

- **The slides are heavily influenced by:**
  - course “CS 15-213” at CMU (R. Bryant, D. O’Halloran, G. Kesden and M. Puschel)
  - course “Computer Architecture and System Programming” at ETH Zurich (T. Roscoe)
  - course “The Hardware/Software Interface” at UW (L. Ceze and G. Borriello)
  - previous instalments of C113 (H. Wiklicky, A. Gopalan, R. Hayden and N. Dulay).

- **Additional information online:**
  - book “Computer Systems: A Programmer’s Perspective”, 3rd edition, Prentice Hall 2013
  - online resources (related to the book, and else) http://csapp.cs.cmu.edu
  - Intel manuals