

# x86-64 Programming I

CSE 351 Spring 2021

## Instructor:

Ruth Anderson

## Teaching Assistants:

Allen Aby

Joy Dang

Alena Dickmann

Catherine Guevara

Corinne Herzog

Ian Hsiao

Diya Joy

Jim Limprasert

Armin Magness

Aman Mohammed

Monty Nitschke

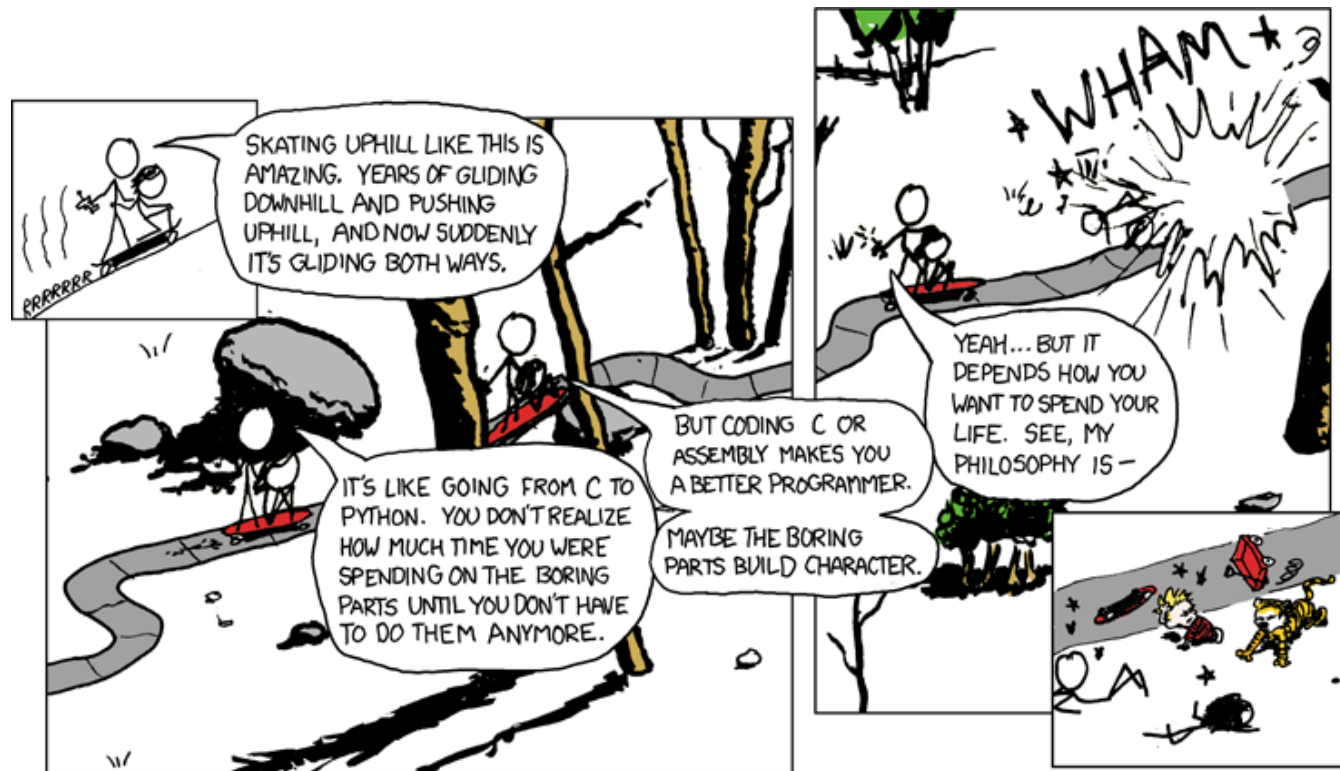
Allie Pflieger

Neil Ryan

Alex Saveau

Sanjana Sridhar

Amy Xu



<http://xkcd.com/409/>

# Administrivia

- ❖ hw6 due TONIGHT (4/14) @ 11:59 pm
- ❖ Lab 1a closes TONIGHT (4/12) @ 11:59 pm
  - Submit `pointer.c` and `lab1Areflect.txt`
  - Make sure you check the Gradescope autograder output!
  - Can use late day tokens to submit up until Wed 11:59 pm
- ❖ Lab 1b, due 4/19
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Breflect.txt`
- ❖ **Questions Docs:** Use @uw google account to access!!
  - <https://tinyurl.com/CSE351-21sp-Questions>

# Reading Review

## ❖ Terminology:

- Instruction Set Architecture (ISA): CISC vs. RISC
- Instructions: data transfer, arithmetic/logical, control flow
  - Size specifiers: b, w,  $\uparrow$ , q
- Operands: immediates, registers, memory
  - Memory operand: displacement, base register, index register, scale factor

# Review Questions

- ❖ Assume that the register `%rax` currently holds the value `0x 01 02 03 04 05 06 07 08`
- ❖ Answer the questions on Ed Lessons about the following instruction (`<instr> <src> <dst>`):

```
xorw $-1, %ax
```

- Operation type:
- Operand types:
- Operation width:
- Result in `%rax`:

# Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- Integers & floats
- x86 assembly**
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

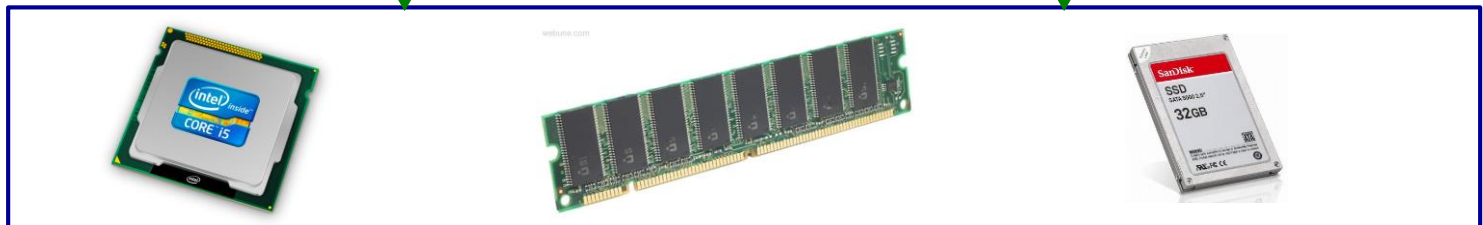
Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

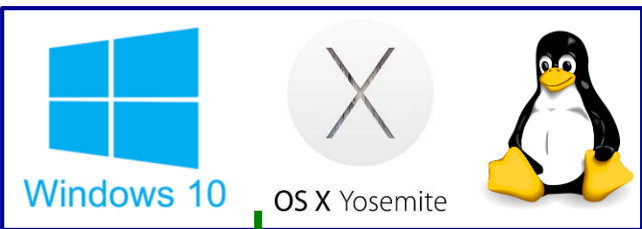
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



OS:

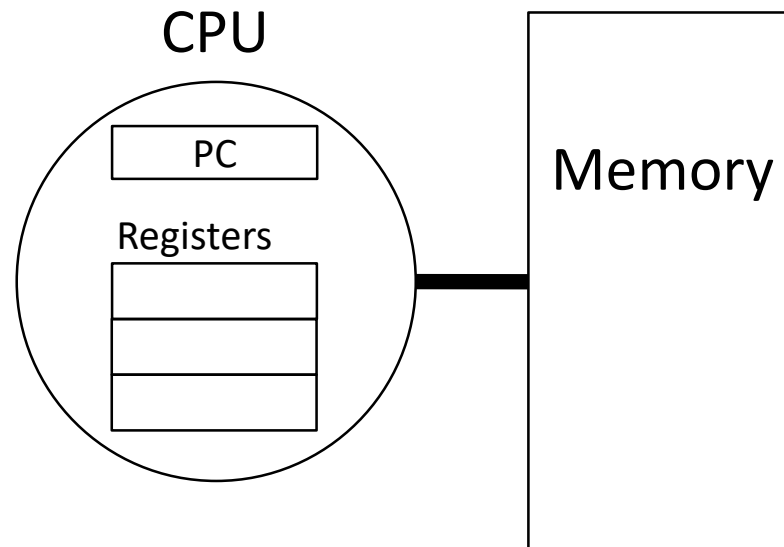


# Definitions

- ❖ **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”
- ❖ **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469

# Instruction Set Architectures

- ❖ The ISA defines:
  - The system's **state** (e.g. registers, memory, program counter)
  - The **instructions** the CPU can execute
  - The **effect** that each of these instructions will have on the system state



# General ISA Design Decisions

- ❖ Instructions
  - What instructions are available? What do they do?
  - How are they encoded?
  
- ❖ Registers
  - How many registers are there?
  - How wide are they?
  
- ❖ Memory
  - How do you specify a memory location?



# Instruction Set Philosophies

- ❖ *Complex Instruction Set Computing (CISC)*: Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- ❖ *Reduced Instruction Set Computing (RISC)*: Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

# Mainstream ISAs



## x86

<b>Designer</b>	Intel, AMD
<b>Bits</b>	16-bit, 32-bit and 64-bit
<b>Introduced</b>	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
<b>Design</b>	CISC
<b>Type</b>	Register-memory
<b>Encoding</b>	Variable (1 to 15 bytes)
<b>Branching</b>	Condition code
<b>Endianness</b>	Little

Macbooks & PCs  
(Core i3, i5, i7, M)  
[x86-64 Instruction Set](#)



## ARM

<b>Designer</b>	Arm Holdings
<b>Bits</b>	32-bit, 64-bit
<b>Introduced</b>	1985
<b>Design</b>	RISC
<b>Type</b>	Register-Register
<b>Encoding</b>	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions; ARMv7 user-space compatibility. <sup>[1]</sup>
<b>Branching</b>	Condition code, compare and branch
<b>Endianness</b>	Bi (little as default)

Smartphone-like devices  
(iPhone, iPad, Raspberry Pi)  
[ARM Instruction Set](#)



## MIPS

<b>Designer</b>	MIPS Technologies, Imagination Technologies
<b>Bits</b>	64-bit (32 → 64)
<b>Introduced</b>	1985
<b>Version</b>	MIPS32/64 Release 6 (2014)
<b>Design</b>	RISC
<b>Type</b>	Register-Register
<b>Encoding</b>	Fixed
<b>Branching</b>	Compare and branch
<b>Endianness</b>	Bi

Digital home & networking equipment  
(Blu-ray, PlayStation 2)  
[MIPS Instruction Set](#)

# Architecture Sits at the Hardware Interface

## Source code

Different applications  
or algorithms

## Compiler

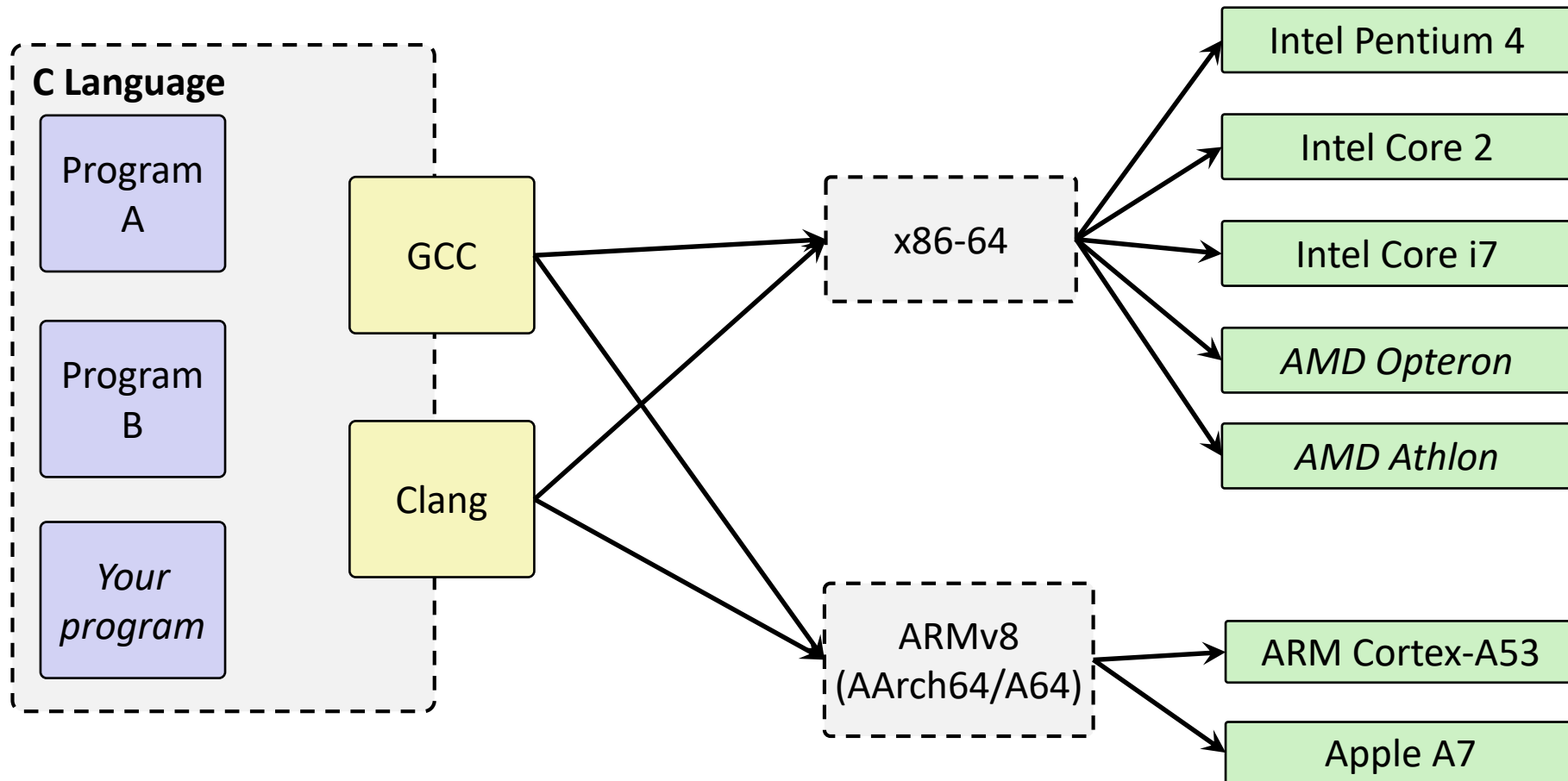
Perform optimizations,  
generate instructions

## Architecture

Instruction set

## Hardware

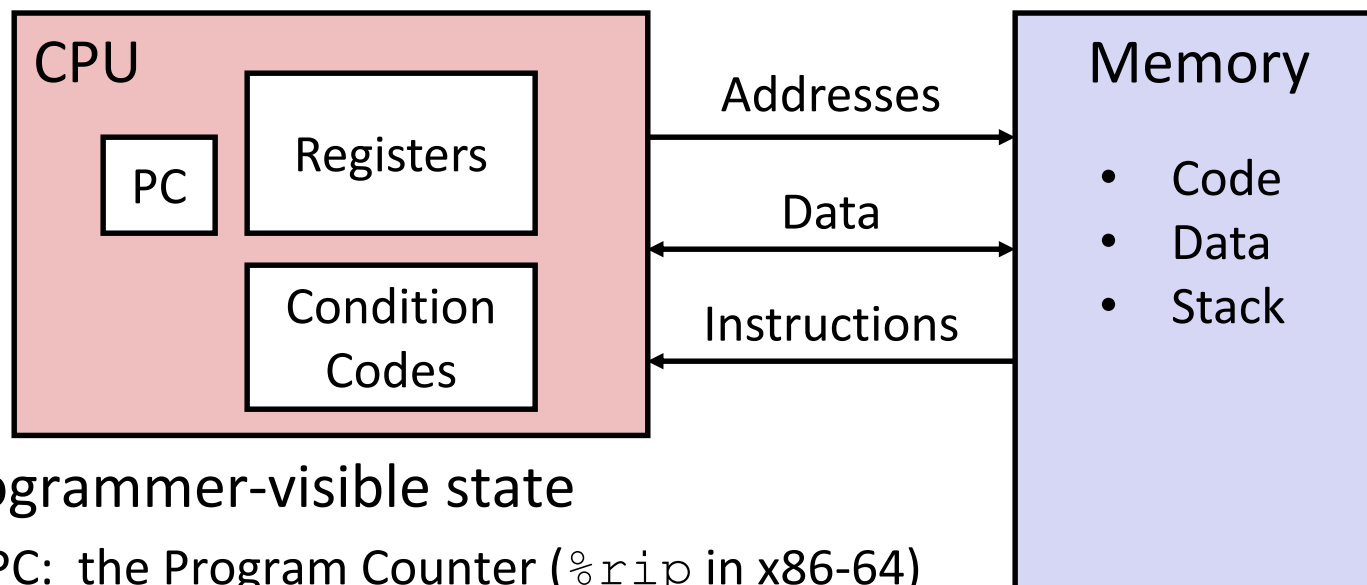
Different  
implementations



# Writing Assembly Code? In 2021???

- ❖ Chances are, you'll never write a program in assembly, but understanding assembly is the key to the machine-level execution model:
  - Behavior of programs in the presence of bugs
    - When high-level language model breaks down
  - Tuning program performance
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing systems software
    - What are the “states” of processes that the OS must manage
    - Using special units (timers, I/O co-processors, etc.) inside processor!
  - Fighting malicious software
    - Distributed software is in binary form

# Assembly Programmer's View



## ❖ Programmer-visible state

- PC: the Program Counter (`%rip` in x86-64)
  - Address of next instruction
- Named registers
  - Together in “register file”
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

## ❖ Memory

- Byte-addressable array
- Code and user data
- Includes *the Stack* (for supporting procedures)

# x86-64 Assembly “Data Types”

- ❖ Integral data of 1, 2, 4, or 8 bytes
    - Data values
    - Addresses
  - ❖ Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
    - Different registers for those (e.g. `%xmm1`, `%ymm2`)
    - Come from *extensions to x86* (SSE, AVX, ...)
  - ❖ No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
  - ❖ Two common syntaxes
    - “AT&T”: used by our course, slides, textbook, gnu tools, ...
    - “Intel”: used by Intel documentation, Intel tools, ...
    - Must know which you’re reading
- } Not covered  
In 351

# 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 have *names*, not *addresses*
  - In assembly, they start with `%` (e.g. `%rsi`)
- ❖ Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86

# x86-64 Integer Registers – 64 bits wide

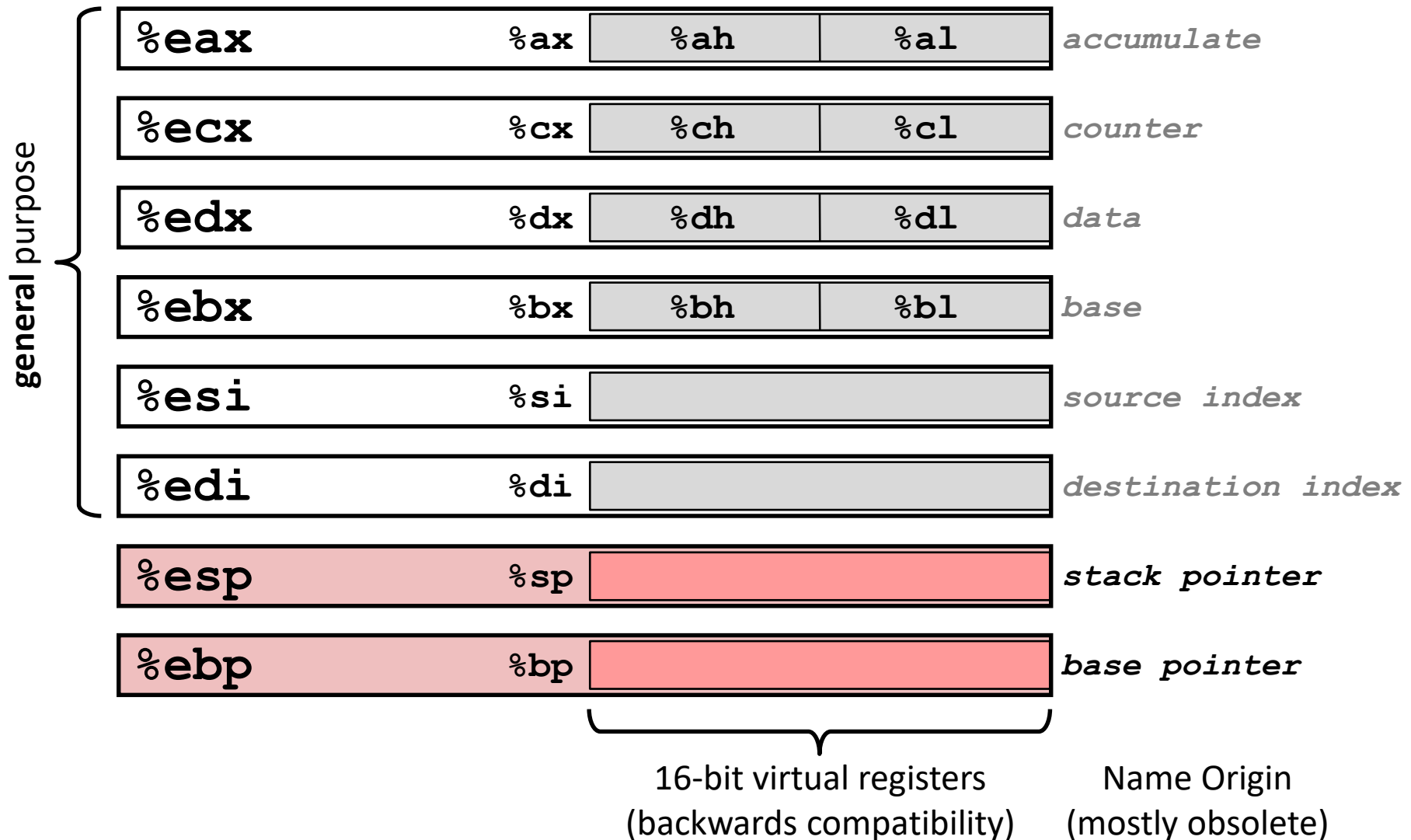
<code>%rax</code>	<code>%eax</code>
<code>%rbx</code>	<code>%ebx</code>
<code>%rcx</code>	<code>%ecx</code>
<code>%rdx</code>	<code>%edx</code>
<code>%rsi</code>	<code>%esi</code>
<code>%rdi</code>	<code>%edi</code>
<code>%rsp</code>	<code>%esp</code>
<code>%rbp</code>	<code>%ebp</code>

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

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



# Some History: IA32 Registers – 32 bits wide



# Memory

- ❖ Addresses
  - `0x7FFFD024C3DC`
- ❖ Big
  - `~ 8 GiB`
- ❖ Slow
  - `~50-100 ns`
- ❖ Dynamic
  - Can “grow” as needed while program runs

# vs. Registers

- vs. Names
  - `%rdi`
- vs. Small
  - `(16 x 8 B) = 128 B`
- vs. Fast
  - sub-nanosecond timescale
- vs. Static
  - fixed number in hardware

# Three Basic Kinds of Instructions

## 1) Transfer data between memory and register

- *Load* data from memory into register
  - `%reg = Mem[address]`
- *Store* register data into memory
  - `Mem[address] = %reg`

**Remember:** Memory is indexed just like an array of bytes!

## 2) Perform arithmetic operation on register or memory data

- `c = a + b;`      `z = x << y;`      `i = h & g;`

## 3) Control flow: what instruction to execute next

- Unconditional jumps to/from procedures
- Conditional branches

# Instruction Sizes and Operands

## ❖ Size specifiers

- b = 1-byte “byte”, w = 2-byte “word”,  
l = 4-byte “long word”, q = 8-byte “quad word”
- Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names

## ❖ Operand types

- **Immediate:** Constant integer data (\$)
- **Register:** 1 of 16 integer registers (%)
- **Memory:** Consecutive bytes of memory at a computed address (C)

# x86-64 Introduction

- ❖ Data transfer instruction (mov)
- ❖ Arithmetic operations
- ❖ Memory addressing modes
  - swap example

# Moving Data

- ❖ General form: `MOV_ source, destination`
  - Really more of a “copy” than a “move”
  - Like all instructions, missing letter (`_`) is the size specifier
  - Lots of these in typical code

# Operand Combinations

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	var_a = 0x4;
		Mem	movq \$-147, (%rax)	*p_a = -147;
	Reg	Reg	movq %rax, %rdx	var_d = var_a;
		Mem	movq %rax, (%rdx)	*p_d = var_a;
	Mem	Reg	movq (%rax), %rdx	var_d = *p_a;

❖ *Cannot do memory-memory transfer with a single instruction*

- How would you do it?

# Some Arithmetic Operations

## ❖ Binary (two-operand) Instructions:

- **Maximum of one memory operand**
- Beware argument order!
- No distinction between signed and unsigned
  - Only arithmetic vs. logical shifts

Format	Computation	
<code>addq src, dst</code>	$dst = dst + src$	( <code>dst += src</code> )
<code>subq src, dst</code>	$dst = dst - src$	
<code>imulq src, dst</code>	$dst = dst * src$	signed mult
<code>sarq src, dst</code>	$dst = dst \gg src$	Arithmetic
<code>shrq src, dst</code>	$dst = dst \gg src$	Logical
<code>shlq src, dst</code>	$dst = dst \ll src$	(same as <code>salq</code> )
<code>xorq src, dst</code>	$dst = dst \wedge src$	
<code>andq src, dst</code>	$dst = dst \& src$	
<code>orq src, dst</code>	$dst = dst   src$	

↑ operand size specifier



# Practice Question

❖ Which of the following are valid implementations of  $rcx = rax + rbx$ ?

- `addq %rax, %rcx`  
`addq %rbx, %rcx`

- `movq %rax, %rcx`  
`addq %rbx, %rcx`

- `movq $0, %rcx`  
`addq %rbx, %rcx`  
`addq %rax, %rcx`

- `xorq %rax, %rax`  
`addq %rax, %rcx`  
`addq %rbx, %rcx`

# Arithmetic Example

```
long simple_arith(long x, long y)
{
    long t1 = x + y;
    long t2 = t1 * 3;
    return t2;
}
```

Register	Use(s)
%rdi	1 <sup>st</sup> argument (x)
%rsi	2 <sup>nd</sup> argument (y)
%rax	return value

```
y += x;
y *= 3;
long r = y;
return r;
```

```
simple_arith:
    addq    %rdi, %rsi
    imulq   $3, %rsi
    movq    %rsi, %rax
    ret
```

# Example of Basic 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
```

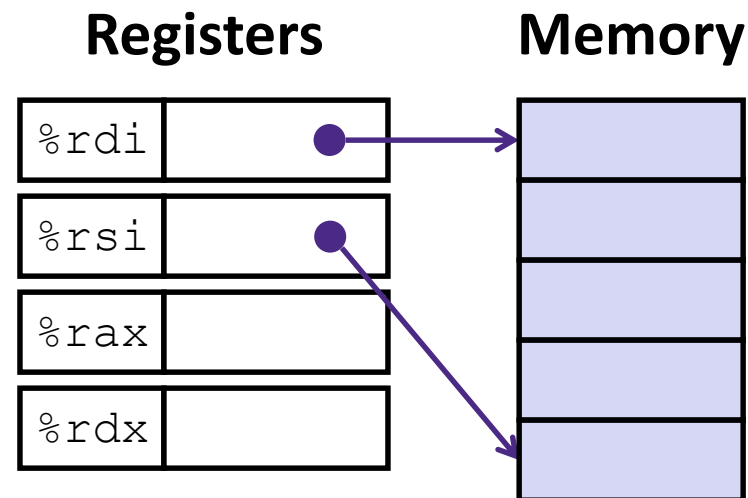
Compiler Explorer:

<https://godbolt.org/z/zc4Pcq>

# Understanding swap ()

```
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
```



<u>Register</u>		<u>Variable</u>
%rdi	↔	xp
%rsi	↔	yp
%rax	↔	t0
%rdx	↔	t1

# Understanding swap ()

## Registers

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

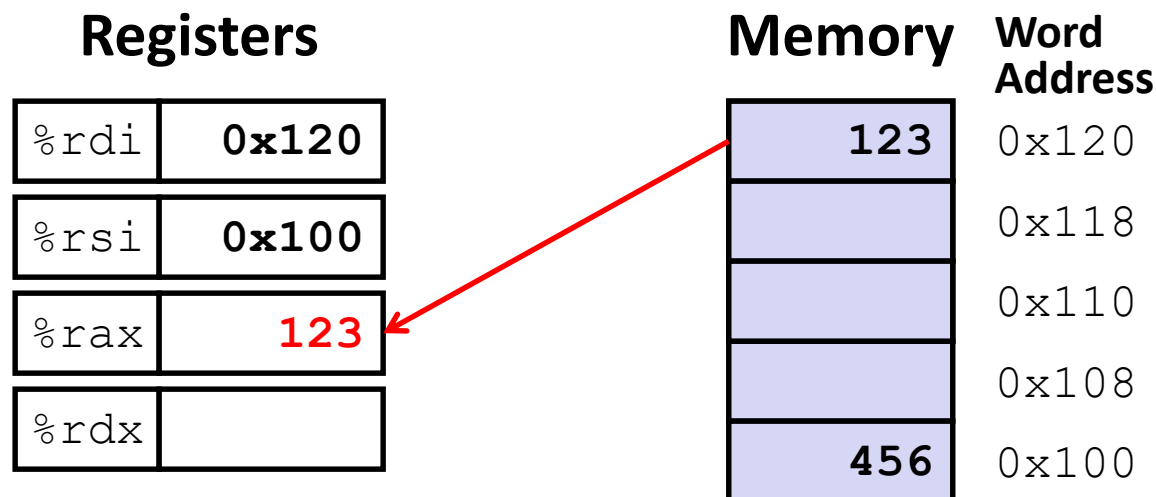
## Memory Word Address

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 ()



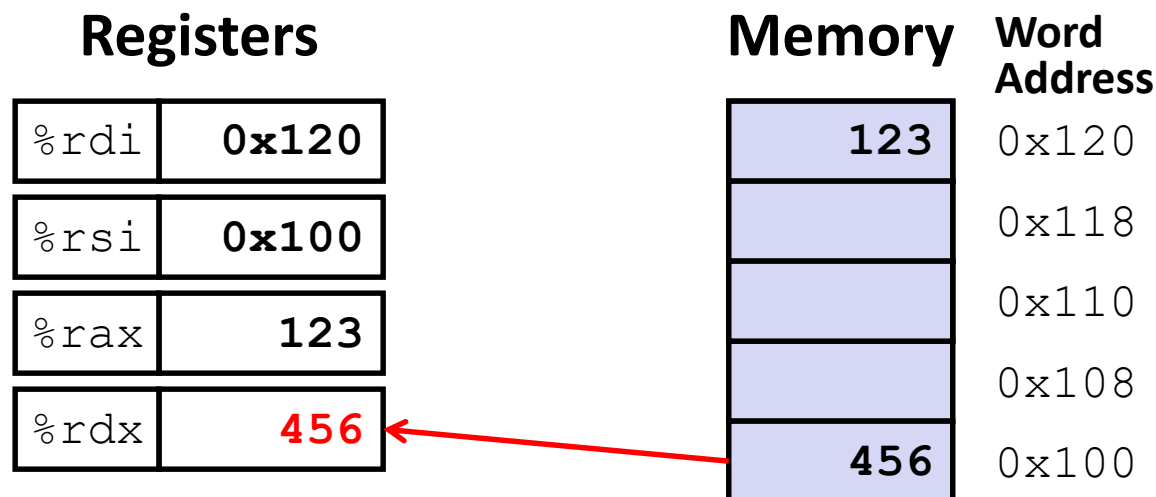
```
swap:
```

```

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

```

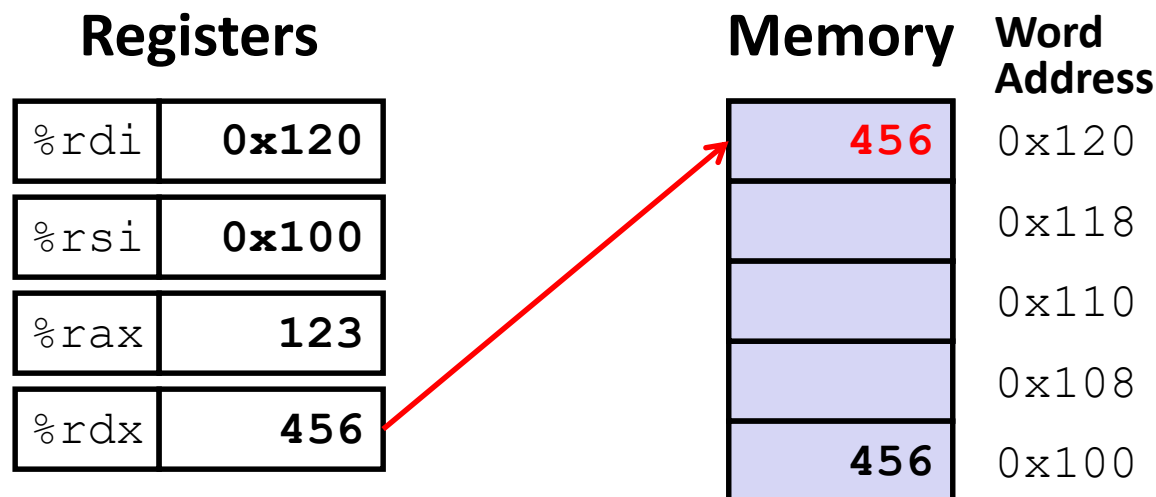
# Understanding swap ()



```
swap:
```

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

# Understanding swap ()

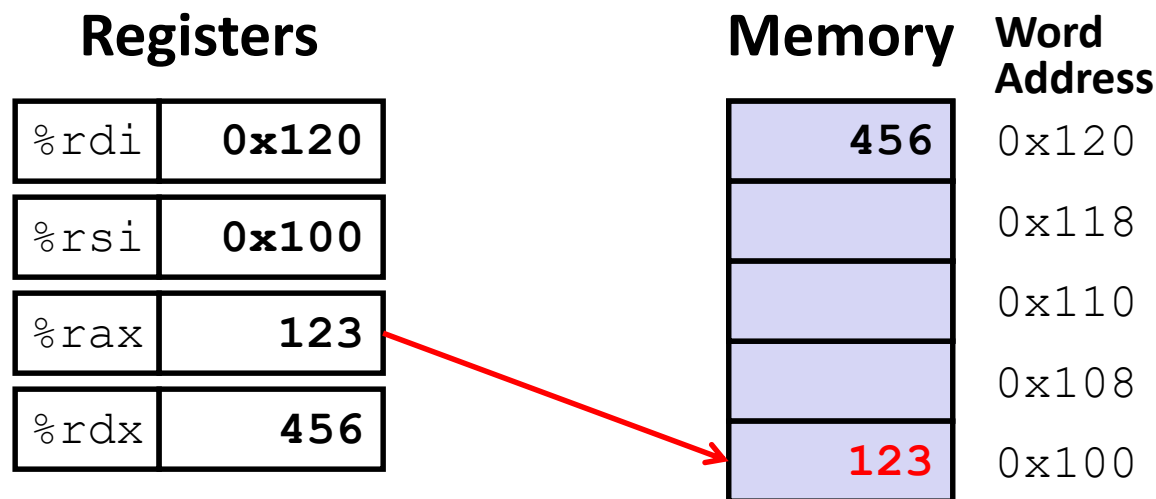


```
swap:
```

```
    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

```

# Memory Addressing Modes: Basic

❖ **Indirect:**  $(R)$   $\text{Mem}[\text{Reg}[R]]$

- Data in register  $R$  specifies the memory address
- Like pointer dereference in C
- Example: `movq (%rcx), %rax`

❖ **Displacement:**  $D (R)$   $\text{Mem}[\text{Reg}[R]+D]$

- Data in register  $R$  specifies the *start* of some memory region
- Constant displacement  $D$  specifies the offset from that address
- Example: `movq 8(%rbp), %rdx`

# Complete Memory Addressing Modes

## ❖ General:

- $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$ 
  - Rb: Base register (any register)
  - Ri: Index register (any register except `%rsp`)
  - S: Scale factor (1, 2, 4, 8) – *why these numbers?*
  - D: Constant displacement value (a.k.a. immediate)

## ❖ Special cases (see CSPP Figure 3.3 on p.181)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

# Address Computation Examples

<code>%rdx</code>	<b>0xf000</b>
<code>%rcx</code>	<b>0x0100</b>

$$D(Rb, Ri, S) \rightarrow$$

$$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$$

Expression	Address Computation	Address
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

# Summary

- ❖ x86-64 is a complex instruction set computing (CISC) architecture
  - There are 3 types of operands in x86-64
    - Immediate, Register, Memory
  - There are 3 types of instructions in x86-64
    - Data transfer, Arithmetic, Control Flow
- ❖ **Memory Addressing Modes:** The addresses used for accessing memory in `MOV` (and other) instructions can be computed in several different ways
  - *Base register, index register, scale factor, and displacement* map well to pointer arithmetic operations