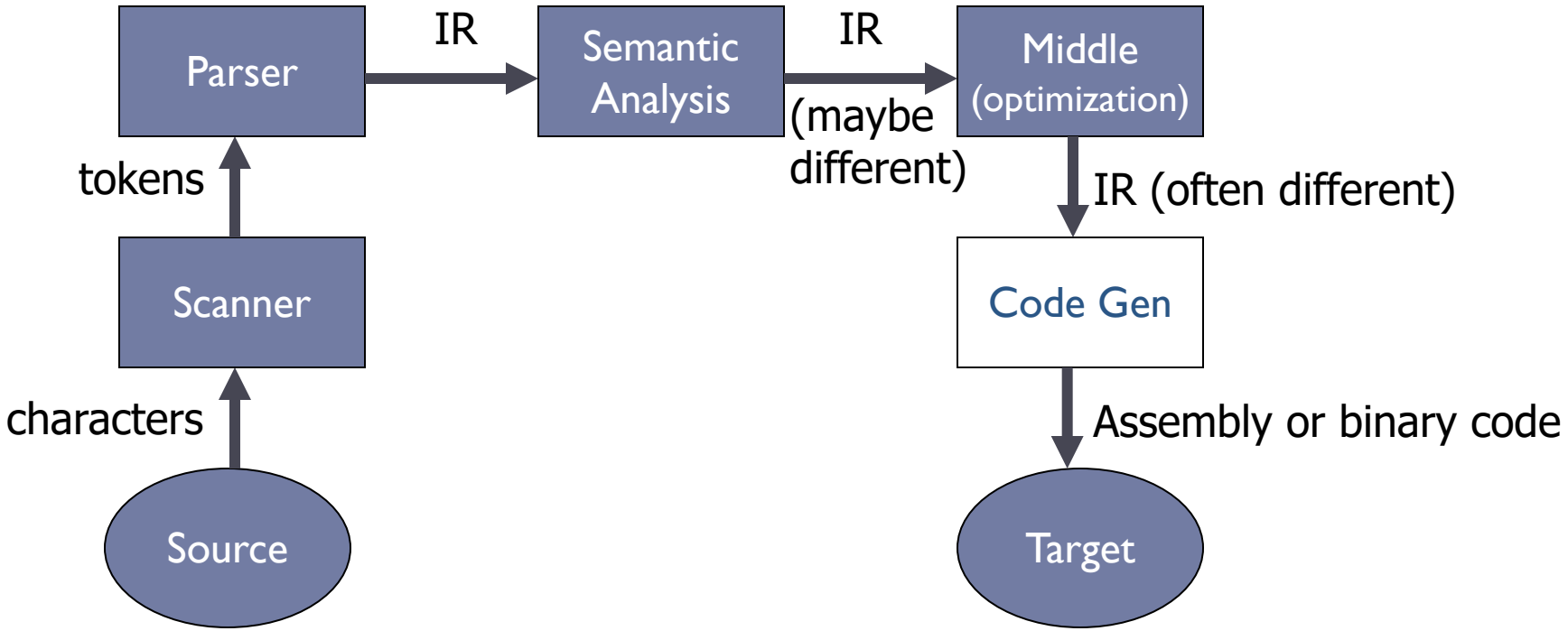
CSC 348 – Intro to Compilers
Lecture 6

Dr. David Zaretsky
david.zaretsky@depaul.edu

Agenda

- Overview of x86-64 architecture
- Programming Assignment 4

Compiler Structure (review)

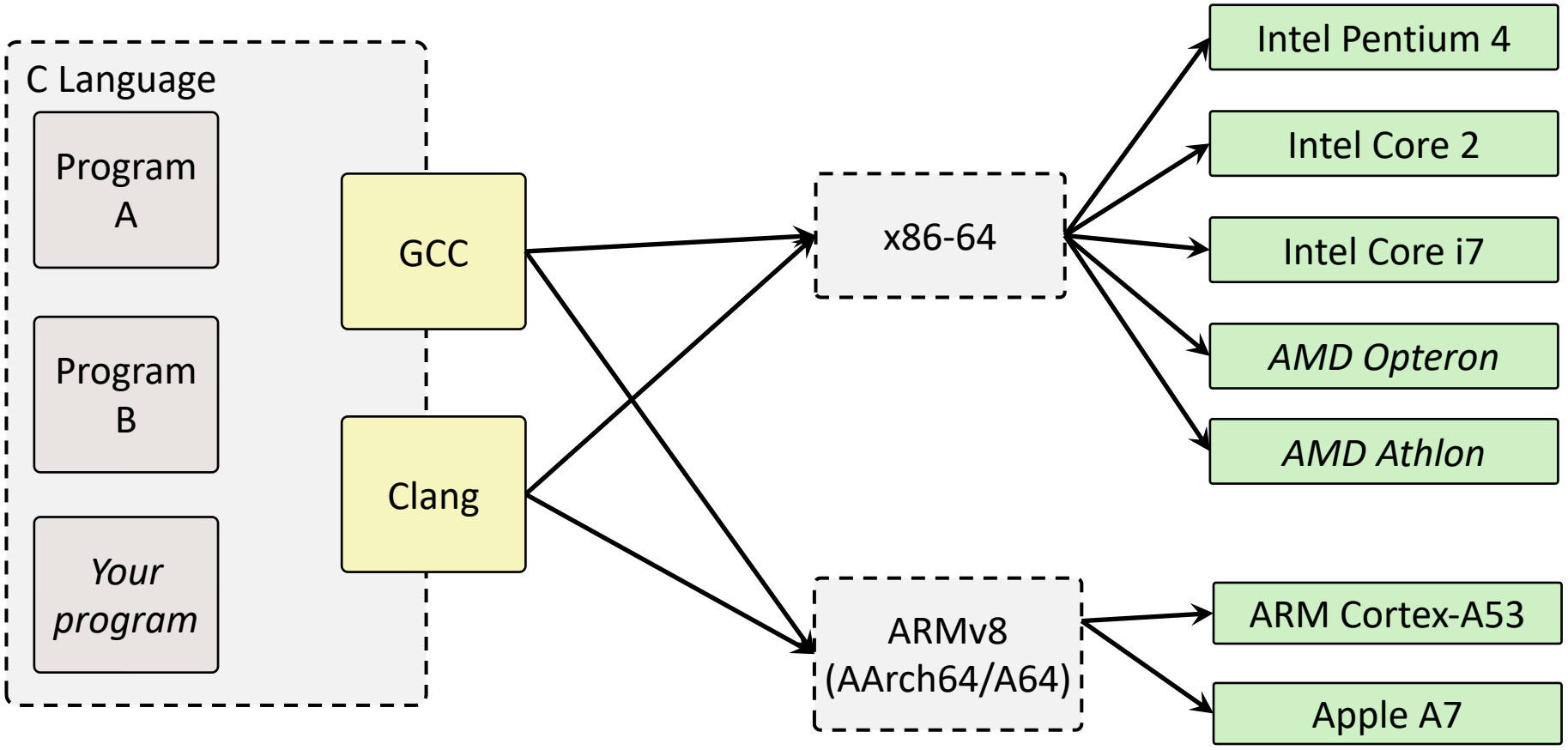


Code Translation

- Code translation can vary based on:
  - Different programs have different algorithms & code structures
  - Different architectures expose different interfaces
  - Hardware implementations have different performance
- Compilers figure out what instructions to generate from the code
  - Employ various optimizations
  - Use combination of instructions to implement high-level operations

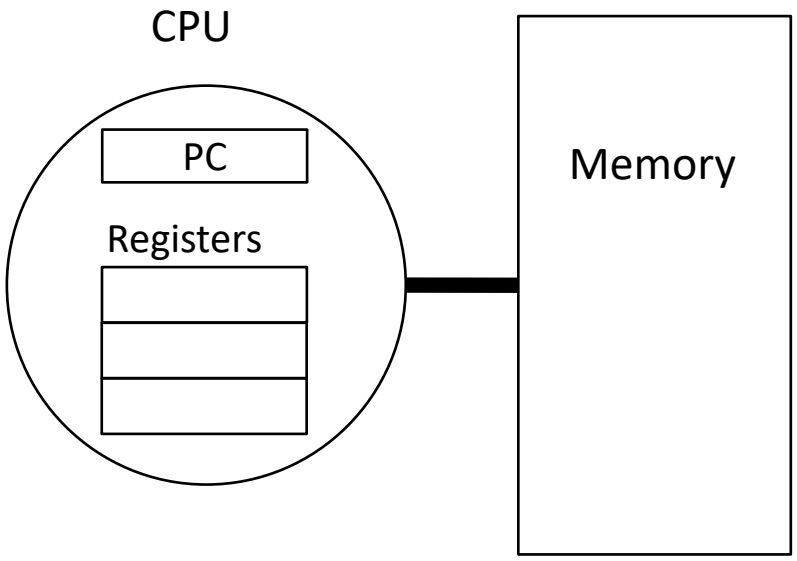
HW Interface Affects Performance

Source code	Compiler	Architecture	Hardware
Different applications or algorithms	Perform optimizations, generate instructions	Instruction set	Different implementations



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



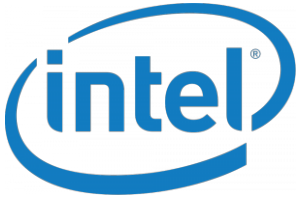
Architecture - CISC vs. RISC

- 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
  - Many very special purpose instructions that you will never see, and a given compiler may never use - just need to know how to use the manual
  - Variable-length instructions, between 1 and 16(?) bytes long.
    - 16 is max len in theory, I don't know if it can happen in practice
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs
- Reduced Instruction Set Computing (RISC):
  - Typically more registers, less and fixed-size instructions
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones
  - Other major architectures are typically RISC
  - Examples: PowerPC, ARM, SPARC, MIPS

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?
  - How do you read / write to a memory location?

Mainstream ISAs



x86

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

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

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

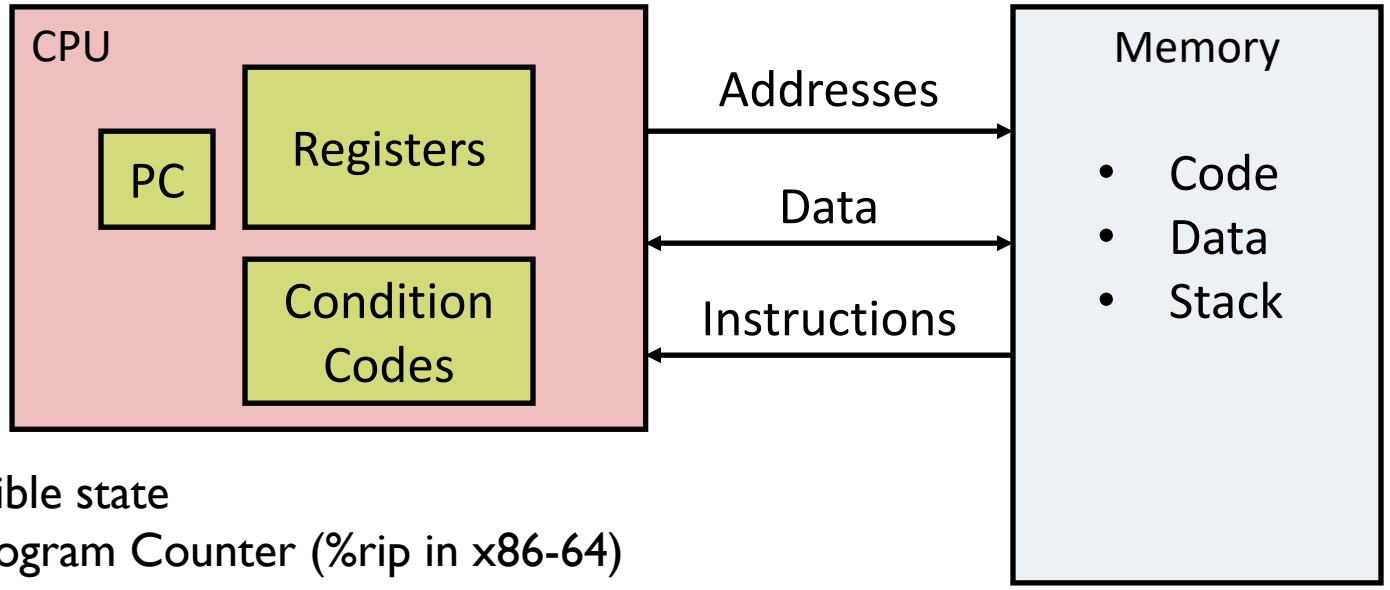
Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

MIPS

Designer	MIPS Technologies, Inc.
Bits	64-bit (32→64)
Introduced	1981; 35 years ago
Design	RISC
Type	Register-Register
Encoding	Fixed
Endianness	Bi

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

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 References

- x86-64 Instructions and ABI
  - Handout for University of Chicago CMSC 22620 by John Reppy
- x86-64 Machine Level Programming
- Quick guide to GDB
- Intel processor documentation
- GNU assembler documentation (x86 and x86-64 details, AT&T vs Intel syntax)

x86-64 Main features

- 16 64-bit general registers
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for floating point, SIMD
- Register-based function call conventions
- Additional addressing modes (pc relative)
- 32-bit legacy mode
- Some pruning of old features

x86-64 Assembler Language

- Target for our compiler project
- But, the nice thing about standards…
- Two main assembler languages for x86-64
  - Intel/Microsoft version – what's in the Intel docs
  - AT&T/GNU assembler – what we're generating and what's in the linked references
    - Use gcc –S to generate asm code from C/C++ code
- Slides use gcc/AT&T/GNU syntax

Intel vs. GNU Assembler

- Main differences between Intel docs and gcc assembler

	Intel/Microsoft	AT&T/GNU as
Operand order: op a,b	a = a op b (dst first)	b = a op b (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, …	movq, addq, pushq [explicit operand size added to end]
Register names	rax, rbx, rbp, rsp, …	%rax, %rbx, %rbp, %rsp, …
Constants	17, 42	$17, $42
Comments	; to end of line	# to end of line or /* … */

- Intel docs include many complex, historical instructions and artifacts that aren't commonly used by modern compilers – and we won't use them either

x86-64 Assembly "Data Types"

- Integral data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)
- 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 CSC-348

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

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)
- One is particularly important to not misuse: **%rsp – Stack Pointer**

%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

x86-64 registers

- 16 64-bit general registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %rbp, %rsp, %r8-%r15
- Registers can be used as 64-bit integers or pointers, or as 32-bit ints
  - Also possible to reference low-order 16- and 8-bit chunks – we won't for most part
- To simplify our project we'll use only 64-bit data (ints, pointers, even booleans!)

X86-64 Registers

Register	Callee Save	Description
`%rax`		result register; also used in `idiv` and `imul` instructions.
`%rbx`	yes	miscellaneous register
`%rcx`		fourth argument register
`%rdx`		third argument register; also used in `idiv` and `imul` instructions.
`%rsp`		stack pointer
`%rbp`	yes	frame pointer
`%rsi`		second argument register
`%rdi`		first argument register
`%r8`		fifth argument register
`%r9`		sixth argument register
`%r10`		miscellaneous register
`%r11`		miscellaneous register
`%r12-%r15`	yes	miscellaneous registers

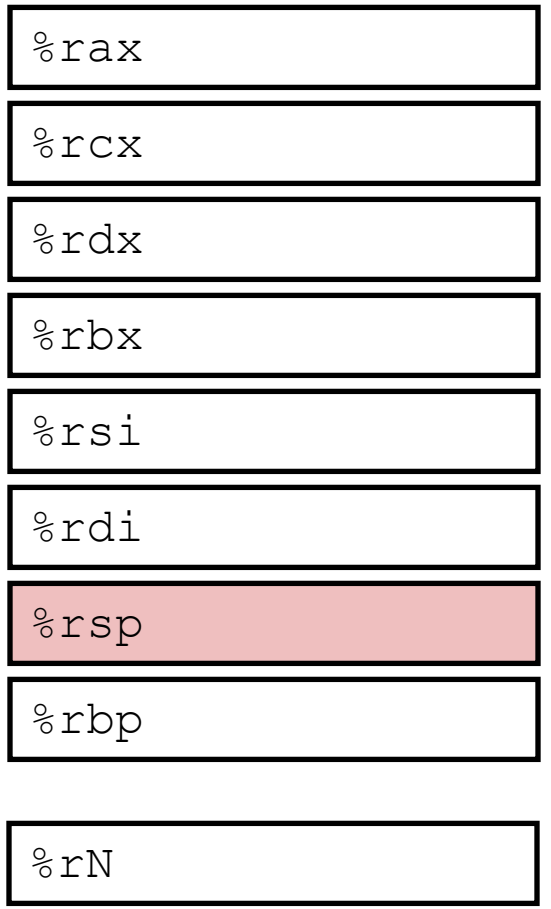
Three Basic Kinds of Instructions

- Transfer data between memory and register
  - Load data from memory into register
    - %reg = Mem[address]
  - Store register data into memory
    - Mem[address] = %reg
- Perform arithmetic operation on register or memory data
  - c = a + b; z = x << y; i = h & g;
- Control flow: what instruction to execute next
  - Unconditional jumps to/from procedures
  - Conditional branches

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

Operand types

- Immediate: Constant integer data
  - Examples: $0x400, $-533
  - Like C literal, but prefixed with '$'
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction
- Register: 1 of 16 integer registers
  - Examples: %rax, %r13
  - But %rsp reserved for special use
  - Others have special uses for particular instructions
- Memory: Consecutive bytes of memory at a computed address
  - Simplest example: (%rax)
  - Various other "address modes"



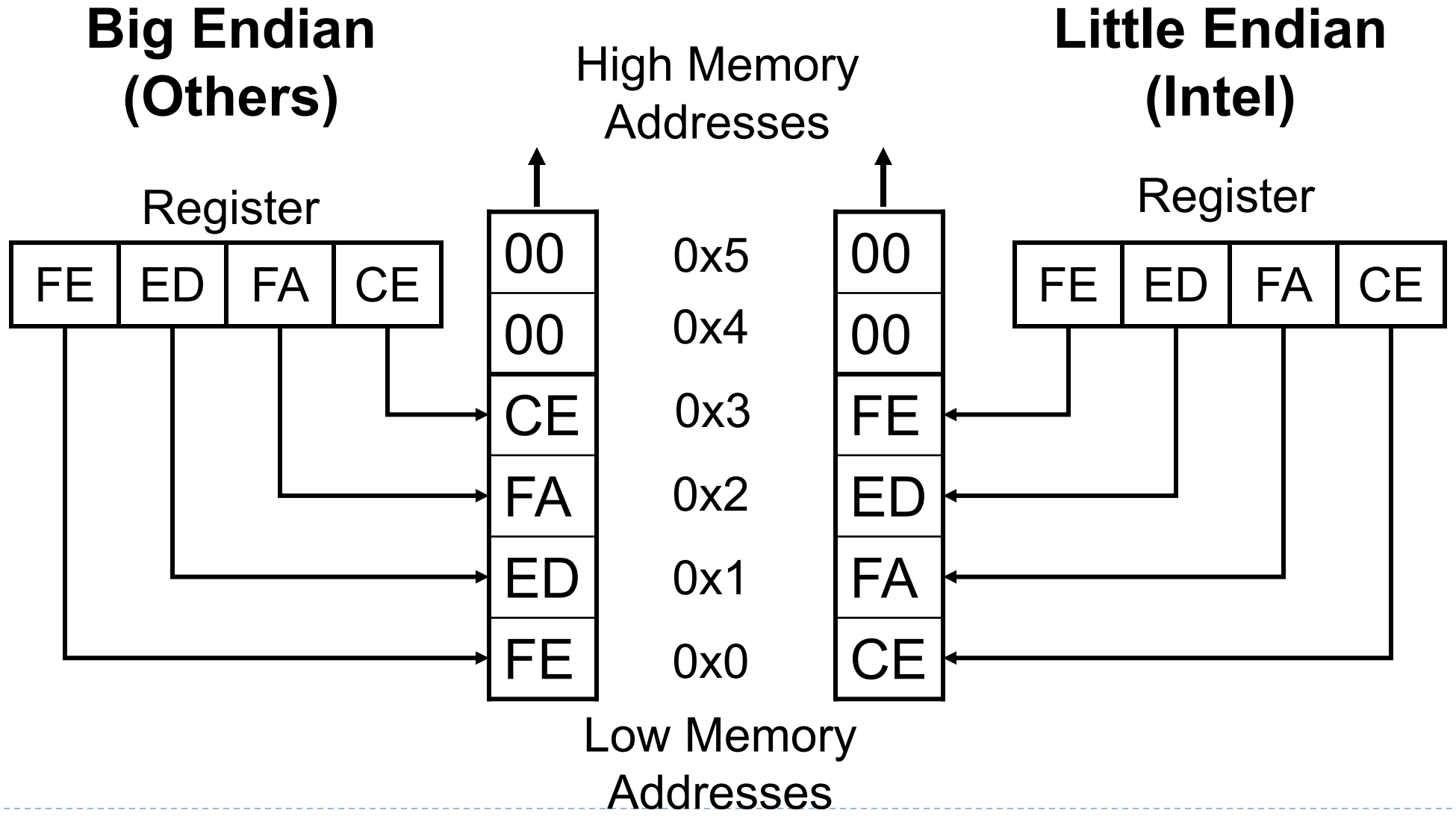
x86-64 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words, double words and quad words (Intel terminology)
  - That's why the 'q' in 64-bit instructions like movq, addq, etc.
- Data should usually be aligned on "natural" boundaries for performance, although unaligned accesses are generally supported – but with a big performance penalty on many machines
- Little-endian – address of a multi-byte integer is address of low-order byte

Architecture - Endian

- Endianness comes from Jonathan Swift's Gulliver's Travels. It doesn't matter which way you eat your eggs :)
- Little Endian - 0x12345678 stored in RAM "little end" first. The least significant byte of a word or larger is stored in the lowest address. E.g. 0x78563412
  - Intel is Little Endian
- Big Endian - 0x12345678 stored as is.
  - Network traffic is Big Endian
  - Most everyone else you've heard of (PowerPC, ARM, SPARC, MIPS) is either Big Endian by default or can be configured as either (Bi-Endian)

Endianess pictures



Instruction Format

- Typical data manipulation instruction

  opcode src,dst # comment

- Meaning is

  dst ⟵ dst op src

- Normally, one operand is a register, the other is a register, memory location, or integer constant
  - Can't have both operands in memory – can't encode two memory addresses in a single instruction (e.g., cmp, mov)
- Language is free-form, comments and labels may appear on lines by themselves (and can have multiple labels per line of code)

Moving Data

- General form: mov_ source, destination
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), "word" means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code

- movb src, dst
  - Move 1-byte "byte"
- movw src, dst
  - Move 2-byte "word"

- `movl src, dst`
  - Move 4-byte "long word"
- `movq src, dst`
  - Move 8-byte "quad word"

movq Operand Combinations

	Source	Dest	Src, Dest	C
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?

Basic Data Movement and Arithmetic Instructions

movq src,dst

dst ⟵ src

addq src,dst

dst ⟵ dst + src

subq src,dst

dst ⟵ dst – src

incq dst

dst ⟵ dst + 1

decq dst

dst ⟵ dst - 1

negq dst

dst ⟵ - dst

(2's complement arithmetic negation)

x86-64 Memory Stack

- Register %rsp points to the "top" of stack
  - Dedicated for this use; don't use otherwise
  - Points to the last 64-bit quadword pushed onto the stack (not next "free" quadword)
  - Should always be quadword (8-byte) aligned
    - It will start out this way, and will stay aligned unless your code does something bad
    - Should be 16-byte aligned on most function calls
  - Stack grows down

Stack Instructions

pushq src

%rsp ⟵ %rsp – 8;

memory[%rsp] ⟵ src

(e.g., push src onto the stack)

popq dst

dst ⟵ memory[%rsp];

%rsp ⟵ %rsp + 8

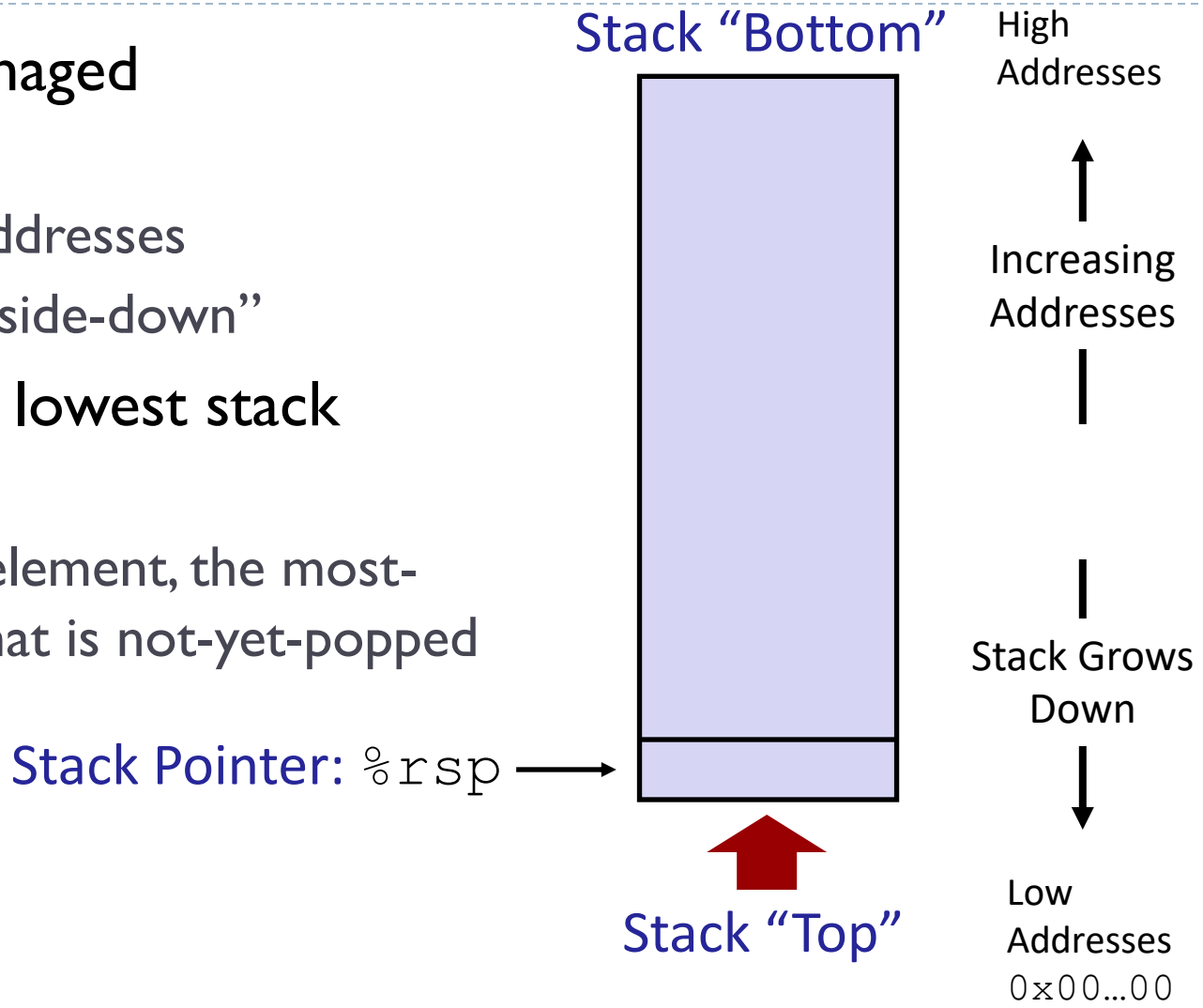
(e.g., pop top of stack into dst and logically remove it from the stack)

Stack Frames

- When a method is called, a stack frame is traditionally allocated on the top of the stack to hold its local variables
- Frame is popped on method return
- By convention, %rbp (base pointer) points to a known offset into the stack frame
  - Local variables referenced relative to %rbp
  - Base pointer common in 32-bit x86 code; less so in x86-64 code where push/pop used less & stack frame normally has fixed size so locals can be referenced from %rsp easily
  - We will use %rbp in our project – simplifies addressing of local variables and compiler bookkeeping

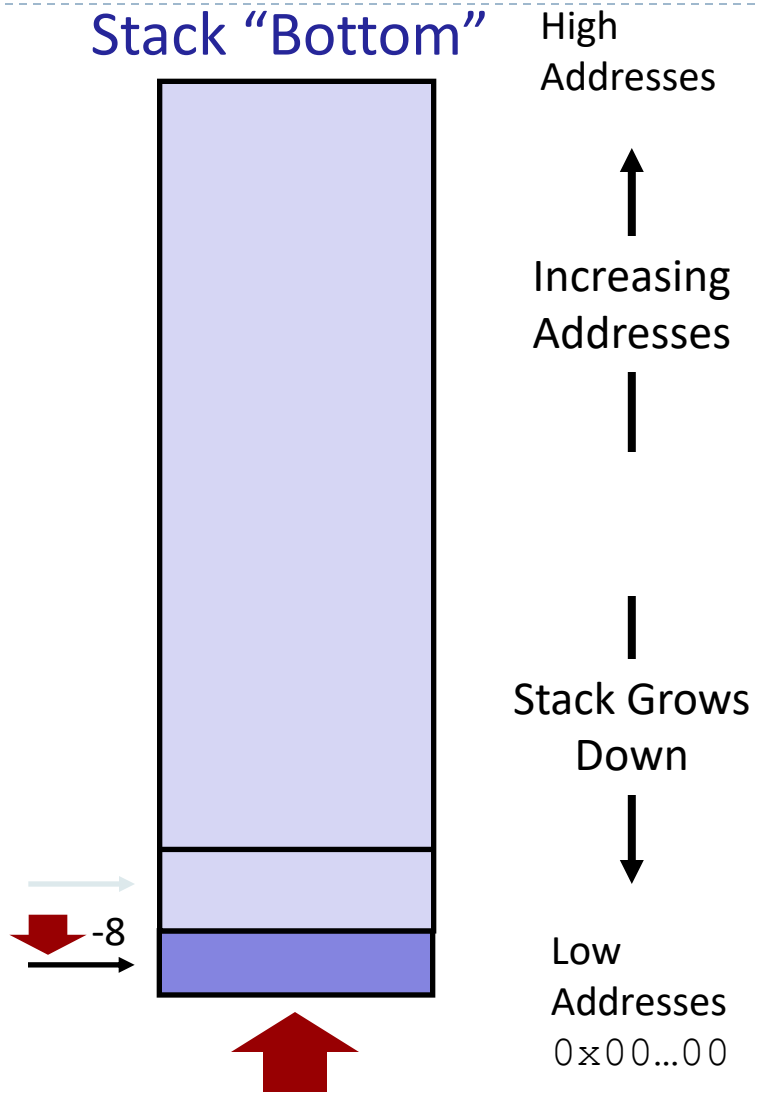
x86-64 Stack

- Region of memory managed with stack "discipline"
  - Grows toward lower addresses
  - Customarily shown "upside-down"
- Register %rsp contains lowest stack address
  - %rsp = address of top element, the most-recently-pushed item that is not-yet-popped



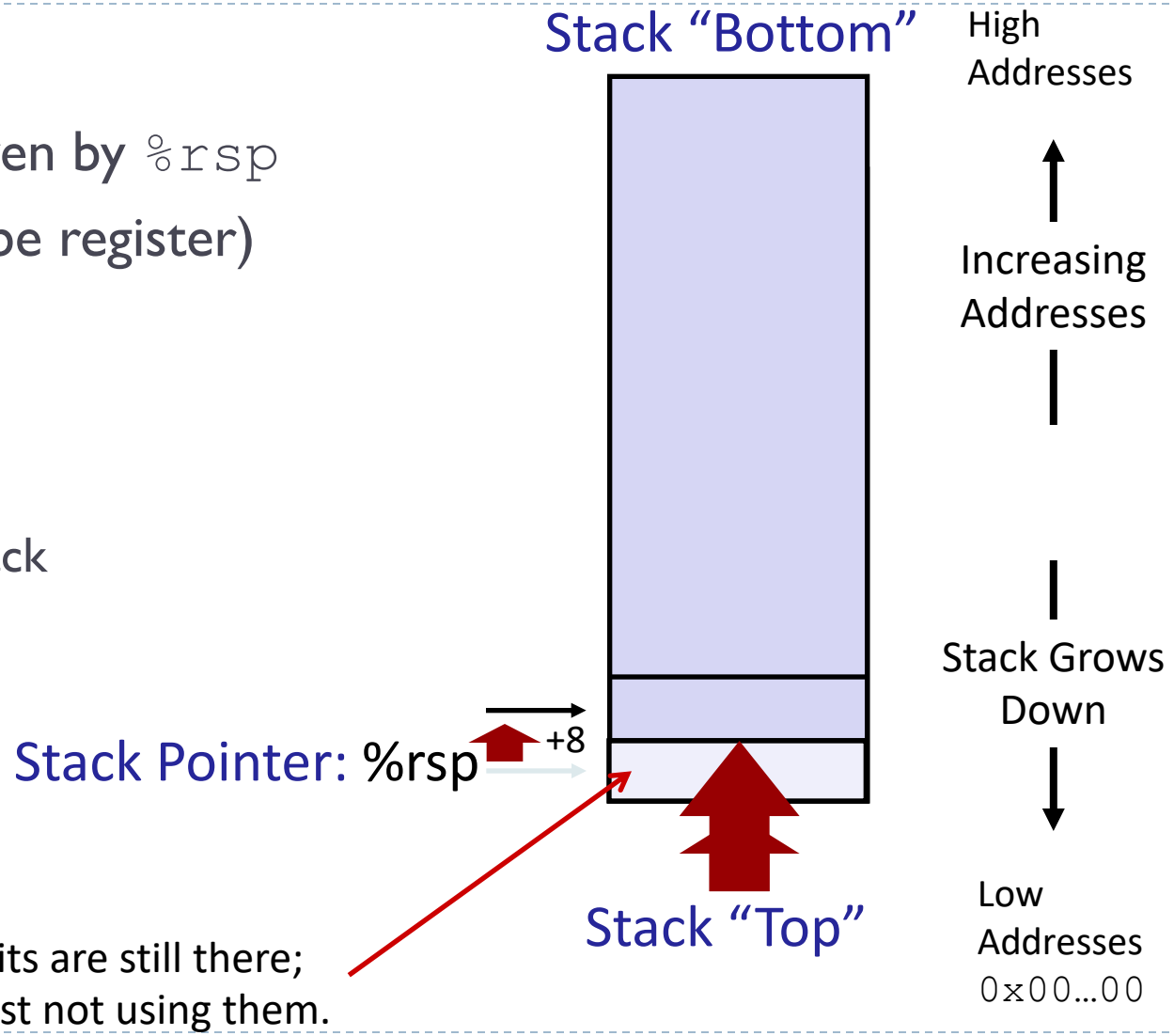
x86-64 Stack: Push

- `pushq` ***src***
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - ***Decrement*** `%rsp` by 8
  - Store value at address given by `%rsp`
- Example:
  - **`pushq %rcx`**
  - Adjust `%rsp` and store contents of `%rcx` on the stack



x86-64 Stack: Pop

- `popq` ***dst***
  - Load value at address given by `%rsp`
  - Store value at *dst* (must be register)
  - ***Increment*** `%rsp` by 8
- Example:
  - **`popq %rcx`**
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`



Operand Address Modes (1)

- These should cover most of what we'll need

```
movq  $17,%rax          # store 17 in %rax
movq  %rcx,%rax         # copy %rcx to %rax
movq  16(%rbp),%rax     # copy memory to %rax
movq  %rax,-24(%rbp)    # copy %rax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus an offset
- Remember: can't have two memory addresses in a single instruction

Operand Address Modes (2)

- A memory address can combine the contents of two registers (with one optionally multiplied by 2, 4, or 8) plus a constant:

  basereg + indexreg*scale + constant

- Main use of general form is for array subscripting or small computations - if the compiler is clever
- Example: suppose we have an array of 8-byte ints with address of the array A in %rcx and subscript i in %rax. Code to store %rbx in A[i]:

```
movq   %rbx,(%rcx,%rax,8)
```

Integer Multiply and Divide

imulq src,dst

dst ⟵ dst * src

dst must be a register

cqto

%rdx:%rax ⟵ 128-bit sign extended copy of %rax

(why??? To prep numerator for idivq!)

idivq src

Divide %rdx:%rax by src (%rdx:%rax holds sign-extended 128-bit value; cannot use other registers for division)

%rax ⟵ quotient

%rdx ⟵ remainder

(no division in MiniJava!)

Bitwise Operations

andq src,dst

dst ⟵ dst & src

orq src,dst

dst ⟵ dst | src

xorq src,dst

dst ⟵ dst ^ src

notq dst

dst ⟵ ~ dst

(logical or 1's complement)

Shifts and Rotates

shlq dst,count

dst ⟵ dst shifted left count bits

shrq dst,count

dst ⟵ dst shifted right count bits (0 fill)

sarq dst,count

dst ⟵ dst shifted right count bits (sign bit fill)

rolq dst,count

dst ⟵ dst rotated left count bits

rorq dst,count

dst ⟵ dst rotated right count bits

Uses for Shifts and Rotates

- Can often be used to optimize multiplication and division by small constants
  - But be careful – be sure semantics are OK
  - Example: right shift is not the same as integer divide for negative numbers (why?)
- There are additional instructions that:
  - shift and rotate double words
  - use a calculated shift amount instead of a constant
  - etc.

Load Effective Address

- The unary **&** operator in C/C++

```
leaq  src,dst          # dst ⟵ address of src
```

  - dst must be a register
  - Address of src includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.
  - Also useful for computing arithmetic expressions that match r1+scale*r2+const

Control Flow - GOTO

- At this level, all we have is Goto and Conditional Goto
- Loops and conditional statements are synthesized from these
- Note on Goto:
  - random jumps play havoc with pipeline efficiency
  - much work is done in modern compilers and processors to minimize this impact

Unconditional Jumps

jmp dst

%rip ⟵ address of dst

- dst is usually a label in the code (which can be on a line by itself)
- dst address can also be indirect using the address in a register or memory location (*reg or *(reg)) – use for method calls, switch

Conditional Jumps

- Most arithmetic instructions set "condition code" bits to record information about the result (zero, non-zero, >0, etc.)
  - Sets condition codes: addq, subq, andq, orq;
  - Does not set conditions codes: imulq, idivq, leaq
- Other instructions that set condition codes

```
cmpq src,dst        # compare dst to src (e.g., dst-src)
testq src,dst       # calculate dst & src (logical and)
```

  - These do not alter src or dst

Conditional Jumps Following Arithmetic Operations

- jz label # jump if result == 0
- jnz label # jump if result != 0
- jg label # jump if result > 0
- jng label # jump if result <= 0
- jge label # jump if result >= 0
- jnge label # jump if result < 0
- jl label # jump if result < 0
- jnl label # jump if result >= 0
- jle label # jump if result <= 0
- jnle label # jump if result > 0
- Obviously, the assembler is providing multiple opcode mnemonics for several actual instructions

Compare and Jump Conditionally

- We would like to be able to compare two operands and jump if a relationship holds between them
- Would like to do this

  jmp_{cond} op1, op2, label

- but can't, because 3-operand instructions can't be encoded in x86-64
- Many other architectures share this same limitation

cmp and jcc

- Instead, we use a 2-instruction sequence

```
cmpq    op1,op2
jcc     label
```

- where j_{cc} is a conditional jump that is taken if the result of the comparison matches the condition cc
  - je label # jump if op1 == op2
  - jne label # jump if op1 != op2
  - jg label # jump if op1 > op2
  - jng label # jump if op1 <= op2
  - jge label # jump if op1 >= op2
  - jl label # jump if op1 < op2
  - jle label # jump if op1 <= op2

Function Call and Return

- The x86-64 instruction set itself only provides for transfer of control (jump) and return
- Stack is used to capture return address and recover it
- Everything else – parameter passing, stack frame organization, register usage – is a matter of convention and not defined by the hardware

call and ret Instructions

DEPAUL

call label

- Push address of next instruction and jump
- %rsp ⟵ %rsp – 8
- memory[%rsp] ⟵ %rip
- %rip ⟵ address of label
- Address can also be in a register or memory as with jmp – we'll use these for dynamic dispatch of method calls (more later)

ret

- Pop address from top of stack and jump
- %rip ⟵ memory[%rsp];
- %rsp ⟵ %rsp + 8
- **WARNING!** The word on the top of the stack had better be an address and not some leftover data

enter and leave

- Complex instructions for languages with nested procedures
  - **enter** can be slow on current processors – best avoided – i.e., don't use it in your project
  - **leave** is equivalent to

```
mov %rsp,%rbp
pop %rbp
```

  and is generated by many compilers. Fits in 1 byte, saves space. Not clear if it's any faster.

X86-64-Register Usage for Functions

- %rax – function result
- Arguments 1-6 passed in these registers in order
  - %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - For Java/C++ "this" pointer is first argument, in %rdi
- %rsp – stack pointer
  - value must be 8-byte aligned always and 16-byte aligned when calling a function
- %rbp – frame pointer (optional use)
  - We'll use it

x86-64 Register Save Conventions

- A called function must preserve these registers (or save/restore them if it wants to use them)
  - %rbx, %rbp, %r12 - %r15
- %rsp isn't on the "callee save list", but needs to be properly restored for return
- All other registers can change across a function call
  - Debugging/correctness note: always assume every called function will change all registers it is allowed to

x86-64 Function Call

- Caller places up to 6 arguments in registers, rest on stack, then executes call instruction (which pushes 8-byte return address)
- On entry, called function prologue sets up the stack frame:

```
pushq    %rbp                # save old frame ptr
movq   %rsp,%rbp             # new frame ptr is top of stack after ret addr and
                             #      old rbp pushed
subq     $framesize,%rsp     # allocate stack frame
```

x86-64 Function Return

- Called function puts result (if any) in %rax and restores any callee-save registers if needed
- Called function returns with:

```
movq    %rbp,%rsp          #  or use leave instead
popq    %rbp               #       of movq/popq
ret
```

- If caller allocated space for arguments it deallocates as needed

Caller Example

- n = sumOf(17,42)

```
        movq    $17, %rdi               # load arguments
        movq    $42, %rsi
        call    sumOf                   # jump & push ret addr
        movq    %rax, offset_n(%rbp)    # store result
```

Example Function

- Source code

```
int sumOf(int x, int y) {
        int a, int b;
        a = x;
        b = a + y;
        return b;
}
```

Assembly Language Version

```
# int sumOf(int x, int y) {
# int a, int b;
sumOf:
    pushq       %rbp            # prologue
    movq        %rsp, %rbp
    subq        $16, %rsp

# a = x;
    movq        %rdi, -8(%rbp)  # %rdi = x

# b = a + y;
    movq        -8(%rbp), %rax
    addq        %rsi, %rax      # %rsi = y
    movq        %rax, -16(%rbp)

# return b;
    movq        -16(%rbp), %rax # %rax = result
    movq        %rbp,%rsp       # epilogue
    popq        %rbp
    ret
# }
```

The Nice Thing About Standards…

- The above is the System V/AMD64 ABI convention (used by Linux, OS X)
- Microsoft's x64 calling conventions are slightly different (sigh…)
  - First four parameters in registers %rcx, %rdx, %r8, %r9; rest on the stack
  - Stack frame needs to include empty space for called function to save values passed in parameter registers if desired
- Not relevant for us, but worth being aware of it

Next…

- Now that we've got a basic idea of the x86-64 instruction set, we need to map language constructs to x86-64
- Next class we will talk about Code Shape
- Then need to figure out how to get compiler to generate this and how to bootstrap things to run the compiled programs
- Programming Assignment 4: Semantic Analysis