**Last Time**

- Began exploring x86-64 instruction set architecture
- 16 general-purpose registers
  - All registers are 64 bits wide
  - rax-rdx are general-purpose integer registers
  - rsp is the stack pointer, rbp is the frame pointer (a.k.a. “base pointer”)
  - rsi, rdi are general purpose; also used for string operations
  - r8-r15 are additional general-purpose integer registers
- Also:
  - rip is the instruction pointer
  - rflags contains status flags
x86-64 Instructions

Instructions follow this pattern:
- opcode operand, operand, ...

Examples:
- add $5, %ax
- mov %rcx, %rdx
- push %rbp

Important note!
- Above assembly-code syntax is called AT&T syntax
- GNU assembler uses this syntax by default
- Intel 64 manuals, other assemblers use Intel syntax

Some big differences between the two formats!
- mov %rcx, %rdx # AT&T: Copies rcx to rdx
- mov rdx, rcx # Intel: Copies rcx to rdx
X86-64 INSTRUCTIONS (2)

- Some general categories of instructions:
  - Data movement instructions
  - Arithmetic and logical instructions
  - Flow-control instructions
  - (many others too, e.g. floating point, SIMD, etc.)

- Data movement:
  - `mov` Move data value from source to destination
  - `movs` Move value with sign-extension
  - `movz` Move value with zero-extension
  - `push` Push value onto the stack
  - `pop` Pop value off of the stack
**x86-64 Data Movement Instructions**

- Data movement instructions can specify a suffix to indicate size of operand(s)
  - b = byte, w = word, l = doubleword, q = quadword
- Some instructions work with one data size:
  - `movq %rcx, %rdx`
    - Moves quadword (8 byte) register `rcx` into `rdx`
  - `pushw %ax`
    - Pushes register `ax` (2 bytes) onto stack
- Move with sign/zero extension takes two sizes:
  - `movsbl %al, %edx`
    - Moves byte `al` into doubleword (4 bytes) register `edx`, extending sign-bit of value into remaining bytes
  - `movzwq %cx, %rax`
    - Moves word (2 bytes) `cx` into quadword (8 bytes) register `rax`, zeroing out higher-order bytes in destination
**x86-64 Operand Types**

- Many different operand types and combinations supported by x86-64 instruction set
- Immediate values – numeric constants:
  - Must specify $ prefix to use a numeric constant
  - $5, $−37, $0xF005B411
- Registers:
  - Specify % prefix on register name
  - %rbp, %eax, %rcx
- Example:
  - **movq $42, %rbx**
  - Moves the value $42_{10}$ into rbx register

![Diagram showing the movement of the value 42 into the rbx register]
x86-64 Memory-Reference Operands

- x86-64 has very rich support for memory references
  - Denote memory access as M[Address]

- Absolute memory access
  - Immediate value with no $ prefix
    - \texttt{movq 0xE700, %rdx}
      - Retrieves memory value M[0xE700] into rdx

- Indirect memory access
  - Register name, enclosed with parens: \texttt{(Reg)}
    - \texttt{movw %cx, (%rbx)}
      - Stores word (2 bytes) in %cx into memory location M[%rbx]

- Base + displacement memory access
  - \texttt{Imm(Reg)} accesses M[Imm + Reg]
    - \texttt{movq \text{-8}(%rbp), %rax}
      - Retrieves quadword M[-8 + %rbp] into %rax
      - (Presumably, %rbp – 8 > 0)
X86-64 Memory-Reference Operands (2)

- Indexed memory access
  - \((\text{RegB, RegI})\) accesses \(M[\text{RegB} + \text{RegI}]\)
    - RegB is the base (i.e. starting) address of a memory array
    - RegI is an index into the memory array
  - \((\text{Imm(RegB, RegI)})\) accesses \(M[\text{Imm} + \text{RegB} + \text{RegI}]\)
  - Assumes that array elements are bytes

- Examples:
  - \(%\text{rax} = 150, %\text{rbx} = 400\)
  - \(\text{movq } (\%\text{rax}, \%\text{rbx}), \%\text{rdx}\)
    - Retrieves value at \(M[150 + 400] = M[550]\) into \(\text{rdx}\)
  - \(\text{movq } \%\text{rcx}, -200(\%\text{rbx}, \%\text{rax})\)
    - Stores \(\text{rcx}\) into location \(M[-200 + 400 + 150] = M[350]\)
Scaled indexed memory access

- With scale factor $s = 1, 2, 4, 8$:
  - $(, \text{Reg}, s)$ \( M[\text{Reg} \times s] \)
  - \(\text{Imm}(, \text{Reg}, s)\) \( M[\text{Imm} + \text{Reg} \times s] \)
  - $(\text{RegB}, \text{RegI}, s)$ \( M[\text{RegB} + \text{RegI} \times s] \)
  - \(\text{Imm}(\text{RegB}, \text{RegI}, s)\) \( M[\text{Imm} + \text{RegB} + \text{RegI} \times s] \)
- For arrays with elements that are 1/2/4/8 bytes

Examples:

- $\%\text{rax} = 150$, $\%\text{rbx} = 400$
- \text{movq} $(, \%\text{rax}, 4), \%\text{rdx}$
  - Retrieves value at \( M[150 \times 4] = M[600] \) into $\%\text{rdx}$
- \text{movq} $\%\text{rcx}, 350(\%\text{rbx}, \%\text{rax}, 2)$
  - Stores $\%\text{rcx}$ into \( M[350 + 400 + 150 \times 2] = M[1050] \)
x86-64 Memory-Reference Summary

Summary chart, from Intel 64 manual:

<table>
<thead>
<tr>
<th>Base</th>
<th>+</th>
<th>Index</th>
<th>×</th>
<th>Scale</th>
<th>+</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td></td>
<td>rax</td>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td>rbx</td>
<td></td>
<td>rbx</td>
<td></td>
<td></td>
<td></td>
<td>8-bit</td>
</tr>
<tr>
<td>rcx</td>
<td></td>
<td>rcx</td>
<td></td>
<td></td>
<td></td>
<td>16-bit</td>
</tr>
<tr>
<td>rdx</td>
<td></td>
<td>rdx</td>
<td></td>
<td></td>
<td></td>
<td>32-bit</td>
</tr>
<tr>
<td>rsp</td>
<td>(not rsp!)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rbp</td>
<td></td>
<td>rbp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rsi</td>
<td></td>
<td>rsi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rdi</td>
<td></td>
<td>rdi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Base, Index, Displacement are all optional
- Scale is only allowed when Index is specified
- Note that \texttt{rsp} can only be used as a base value, but never as an index value
**X86-64 Operand Combinations**

- Important constraints on combinations of operand types
- Source argument can be:
  - Immediate, Register, Memory (direct or indirect)
- Destination argument can be:
  - Register, Memory (direct or indirect)
- Both arguments cannot be memory references
  - To move data from one memory location to another, must move from Mem1 → Register, then from Register → Mem2

- These constraints apply to data movement instructions, and most other instructions too
x86-64 Arithmetic/Logical Operations

- **Unary arithmetic/logical operations:**
  - `inc Dst`  
    \[ Dst = Dst + 1 \]
  - `dec Dst`  
    \[ Dst = Dst - 1 \]
  - `neg Dst`  
    \[ Dst = -Dst \]
  - `not Dst`  
    \[ Dst = \neg Dst \]

- **Binary arithmetic/logical operations:**
  - `add Src, Dst`  
    \[ Dst = Dst + Src \]
  - `sub Src, Dst`  
    \[ Dst = Dst - Src \]
  - `xor Src, Dst`  
    \[ Dst = Dst \oplus Src \]
  - `or Src, Dst`  
    \[ Dst = Dst \lor Src \]
  - `and Src, Dst`  
    \[ Dst = Dst \land Src \]

- Specify byte-width of operands via suffixes, as usual
  - `decb %cl`
    - Decrements the 1-byte value in cl register
  - `addl 16(%rbp), %eax`
    - Adds double-word at M[16 + rbp] to contents of eax

**X86-64 Shift Operations**

- **Shift operations:**
  - `shl k, Dst`  \(\text{Dst} = \text{Dst} \ll k\)
  - `shr k, Dst`  \(\text{Dst} = \text{Dst} \gg k\) (logical)
  - `sal k, Dst`  \(\text{Dst} = \text{Dst} \ll k\)
  - `sar k, Dst`  \(\text{Dst} = \text{Dst} \gg k\) (arithmetic)
  - `shl`, `sal` are identical
  - `k` is a constant, or `%cl` register
  - On 64 bit, can only shift up to 63 bits
  - (On 32 bit, can only shift up to 31 bits)

- **Also rotate operations**
  - See docs for `rol`, `ror`, `rcl`, `rcr`
  - Similar form as shift operations
**X86-64 Multiply, Divide Operations**

- Multiplication and division are more challenging
  - 64-bit value × 64-bit value = 128-bit value
  - 128-bit value ÷ 64-bit value = 64-bit quotient (hopefully), 64-bit remainder

- Two-argument multiplication:
  - `imul Src, Dst`
  - Use width modifier, as usual: `imulq (%rbx), %rcx`
  - For Src, Dst of bit-width \( w \), produces result also of width \( w \)
  - `Dst = (Src \times Dst) \mod 2^w`

- Also three-argument multiplication:
  - `imul Src1, Src2, Dst`
  - `Dst = (Src1 \times Src2) \mod 2^w`
x86-64 Multiply, Divide Operations (2)

- One-argument multiplication:
  - `imulq Src` – 64-bit signed multiplication
    - `rdx:rax = Src \times rax`
    - `rdx` is top 8 bytes of result, `rax` is bottom 8 bytes of result
  - `mulq Src` – 64-bit unsigned multiplication

- One-argument division:
  - `idivq Src` – 64-bit signed division
    - `rax = rdx:rax \text{ div} \ Src`
    - `rdx = rdx:rax \text{ mod} \ Src`
  - `divq Src` – 64-bit unsigned division

- Can use `cqto` to set up `rdx:rax` for division
  - `cqto` – “convert quad-word to oct-word”
    - Sign-extends `rax` into `rdx`, creating `rdx:rax`
  - Note: in Intel manual, this instruction is called `cqo`
x86-64 Multiply, Divide Operations (3)

- Can perform multiplication and division on varying input widths, too
- Examples:
  - `imuld Src` – 32-bit signed multiplication
    - `edx:eax = Src × eax`
  - `idivb Src` – 8-bit signed division
    - `al = ax div Src`
    - `ah = ax mod Src`
x86-64 Conversion Instructions

- Also variants of cqto to set up for division on different input widths
  - cwtd – sign-extends ax into dx:ax
    - “Convert word to double-word”
  - cltd - sign-extends eax into edx:eax
    - “Convert long to double-long”

- Finally, can also sign-extend portions of rax as needed
  - cbtw – sign-extends al into ax
  - cwtl – sign-extends ax into eax
  - cltq – sign-extends eax into rax
**x86-64 Multiply/Divide Examples**

- **Values:**
  - x at location `16(%rbp)`, y at location `24(%rbp)`
  - Both signed values, quad-words (8 bytes)

- **Compute signed product of x and y**
  ```
  movq 16(%rbp), %rax  # rax = x
  imulq 24(%rbp)       # rdx:rax = x * y
  pushq %rdx            # Save 128-bit result
  pushq %rax            # onto stack.
  ```

- **Compute signed quotient and remainder of x ÷ y**
  ```
  movq 16(%rbp), %rax  # rax = x
  cqto                  # rdx:rax = x
  idivq 24(%rbp)        # Compute x / y
  pushq %rax            # rax = quotient
  pushq %rdx            # rdx = remainder
  ```
X86-64 Flow-Control Instructions

- Many different instructions for branching
- Simplest version: unconditional jump
  - `jmp Label`  — Direct jump to address
  - `jmp *Operand`  — Indirect jump to address
    - `jmp *%rax`  — jumps to address stored in `rax`
    - `jmp *((%rax)`  — jumps to address stored at `M[rax]`
  - Can use indirect addressing with unconditional jumps! Very useful in many situations:
    - Implementing switch statements: jump tables
    - Object oriented programming: virtual function ptr tables
    - Other flow-control mechanisms in high-level languages

- Other jumps are conditional jumps
  - Jump occurs based on flags in `rflags` status register
BRANCH LOGIC, CONDITIONAL JUMPS

- Previous branching logic:
  - Fed output of A to Branching Logic
    - Have one opcode: BRZ
  - Not very extensible to general branching tests
Branch Logic, Conditional Jumps (2)

- More powerful branching logic:
  - ALU generates status flags; feed to Branch Logic
  - Conditional jumps use status flags to drive branching
x86-64 Conditional Operations

- Status bits in `rflags` register:
  - CF = carry flag (1 indicates unsigned overflow)
  - SF = sign flag (1 = result is negative)
  - OF = overflow flag (1 indicates signed overflow)
  - ZF = zero flag (1 = result is zero)

- Conditional jump instructions use these flags to control program flow.

- All arithmetic and logical operations set these flags:
  - Good to know how these instructions affect above flags!

- Can also update these flags with `cmp`, `test`:
  - `cmp Src2, Src1`
    - Updates flags as for Src1 – Src2 (i.e. `sub Src2, Src1`)
  - `test Src2, Src1`
    - Updates flags as for Src1 & Src2 (i.e. `and Src2, Src1`)
  - Src1, Src2 are unchanged by comparison/test operation
  - Can specify size prefixes, as usual: `cmpq %rcx, $0`
x86-64 Conditional Jumps

- Conditional jumps can only use a label
  - Can’t specify an indirect conditional jump
- Some operations:
  - `je Label` “Jump if equal” (ZF = 1)
  - `jne Label` “Jump if not equal” (ZF = 0)
    - `sub Src2, Src1` produces zero result if Src1 == Src2
    - `cmp Src2, Src1` sets zero-flag in this case
  - `js Label` “Jump if sign” (SF = 1)
  - `jns Label` “Jump if not sign” (SF = 0)
    - Jump if answer is negative (SF = 1) or nonnegative (SF = 0)
  - `jc/jnc Label` “Jump if [not] carry”
    - Unsigned overflow tests
  - `jo/jno Label` “Jump if [not] overflow”
    - Signed overflow tests
x86-64 Signed Conditional Jumps

- **jg** Label  "Jump if greater" (signed >)
  - **jnle** is synonym – “Jump if not less or equal”
  - All comparison opcodes have synonyms like this

- Also:
  - **jge** Jump if greater or equal
  - **jl** Jump if less
  - **jle** Jump if less or equal

- These look at sign flag, overflow flag, zero flag
  - Remember: OF = signed overflow, CF = unsigned overflow
  - ZF indicates if Src1 == Src2 (Src2 – Src1 == 0)
  - SF + OF indicate whether Src2 – Src1 > 0 or < 0 (when nonzero)
    - Logic is slightly involved; see CS:APP §3.6.1-2 for details
x86-64 Unsigned Conditional-Jumps

- Unsigned comparisons are similar:
  - **ja Label** “Jump if above” (unsigned >)
    - **jnbe** is synonym – “Jump if not below or equal”
  - Also:
    - **jae** Jump if above or equal
    - **jb** Jump if below
    - **jbe** Jump if below or equal
  - These look at carry flag and zero flag
    - CF indicates whether unsigned overflow occurred from Src2 – Src1
    - If Src2 – Src1 generates unsigned overflow, (CF = 1) then Src2 < Src1
    - Again, ZF indicates if Src1 == Src2
x86-64 Conditional-Set Instructions

Also a variety of conditional-set instructions

Examples:

- **sete Dst**  “Set if equal”
  - Stores ZF into 8-bit target Dst
  - Result is 0 or 1
  - Synonym: **setz** “Set if zero”

- Others:
  - **sets/setns Dst**  “Set if sign” / “Set if not sign”
  - **setg Dst**  “Set if greater” (signed >)
  - **setl Dst**  “Set if less” (signed <)
  - **seta Dst**  “Set if above” (unsigned >)
  - **setb Dst**  “Set if below” (unsigned <)
  - etc. (same as for conditional-jump instructions)

- All instructions modify a single 8-bit destination
But Wait, There’s More!

- Really only scratched the surface of x86-64
  - Covered a lot of what you will see in CS24...
  - ...but there’s a lot more where that came from!

- The book reading for Week 2 covers several more instructions, and goes into greater detail
  - Chapter 3 – 3.7

- If you see an instruction you don’t recognize, look it up in Intel 64 manuals (provided on Moodle)
  - If it still doesn’t make sense, ask Donnie or a TA 😊
More Advanced Language Features

- Last time, introduced higher-level abstractions
  - Subroutines, the stack, stack frames, frame pointers
- Many different languages, calling conventions, computational models to choose from!
  - e.g. Functional languages allow functions to be created and passed around dynamically
  - e.g. When a value is no longer used, it is garbage collected automatically

- Start with a simple abstraction: C functions
  - Relatively simple computational model
- Pretty easy to implement with x86-64 assembly
C FUNCTIONS

To implement subroutines (tasks from last time):

- Need a way to pass arguments and return values between caller and subroutine
- Need a way to transfer control from caller to subroutine, then return back to caller
- Need to isolate subroutine’s state from caller’s state
x86-64 Subroutine Calls

- x86-64 provides specific features for subroutines
- Registers:
  - `rsp` = stack pointer
    - Stack grows “downward” in memory
    - `push` decrements `rsp`, then stores value to `(%rsp)`
    - `pop` retrieves value at `(%rsp)`, then increments `rsp`
  - `rbp` = base pointer
    - x86-64 name for frame pointer
- Instructions:
  - `call Addr`
    - Pushes `rip` onto stack (`rip` references next instruction)
    - Sets `rip` = `Addr`
  - `ret`
    - Pops top of stack into `rip`
Many different ways to organize stack frames!

A calling convention is a particular way of passing information to/from a subroutine

The main calling convention for *NIX platforms on x86-64 is called the System V AMD64 ABI

Both the procedure caller and the callee have to coordinate the operation!
- Shared resources: the stack, the register file

Calling convention specifies:
- Who sets up which parts of the call
- What needs to be saved, and by whom
- How to return values back to the caller
- Who cleans up which parts of the call
**x86-64: Passing Arguments**

- Caller is responsible for setting up arguments (of course)
- Since x86-64 has 16 registers, first six arguments are passed in registers
  - `rdi` = Arg 1
  - `rsi` = Arg 2
  - `rdx` = Arg 3
  - `rcx` = Arg 4
  - `r8` = Arg 5
  - `r9` = Arg 6
- Any additional arguments are pushed onto the stack, in reverse order
  - Arg N is pushed first, then N-1, then ..., then Arg 7
x86-64: Passing Arguments (2)

- On x86-64, primitives may be double-word (**int**, **float**) or quad-word (**long**, **double**, pointer)
  - Simplify for now: All values are quad-words
- Note: Stack addresses are in hexadecimal
  - Each entry on the stack is 8 bytes
- Two benefits of pushing arguments in reverse order:
  - Later arguments have a larger offset added to the frame pointer
  - If procedure is passed more arguments than it expects, it doesn’t break the procedure’s code
x86-64: Invoking the Procedure

- Caller uses `call` to invoke the procedure
  - Pushes `rip` of `next` instruction onto the stack
- If the callee needs a frame pointer:
  - Set up frame pointer for this function call
  - `rbp` is used for the frame pointer on x86-64
- Must preserve caller’s frame pointer!
- Typical code:
  ```
  pushq %rbp       # Save caller rbp
  movq %rsp, %rbp  # Set up frame ptr
  ```
- Now:
  - `8 (%rbp)` = Return address
  - `16 (%rbp)` = Arg 7 value
  - `24 (%rbp)` = Arg 8 value
x86-64: Saving Registers

- “Callee must save rbp before it modifies it”
- A general issue:
  - The register file is a shared resource
  - Calling convention must specify how registers are managed
- Callee-save registers:
  - When callee returns to caller, register values must be same as when subroutine was invoked
  - rbp, rbx, and r12 through r15 are callee-save registers
- Caller-save registers:
  - Callee may change these registers without saving them!
  - The caller must save these registers before the call, if the old values need to be preserved
  - All other registers (except rsp) are caller-save registers
  - Note: even argument-passing registers may be changed
x86-64: Returning Results

- For now, only consider simple results
  - e.g. integer value up to 64 bits in size, or a pointer
- In these cases, callee returns the result in rax
  - Set rax to result, restore rbp, then return to caller

Who removes the arguments from the stack??

- In the System V AMD64 ABI calling convention, the caller cleans up stack
  - e.g. can add a constant to rsp to remove arguments
- Having the caller clean up the stack is the safest approach...
  - The caller knows how many arguments they actually pushed onto the stack
x86-64: Local Variables

- Procedures sometimes need space for local variables
  - Compiler figures out how much space, from the source code
  - (Sometimes the compiler allocates more space than is strictly required, for various reasons)
- Note: With 16 registers, a function may be able to use registers instead of stack space
- If needed, local variables reside just below the frame pointer
  - Accessed via \(-\text{off}(\%\text{rbp})\)
- Common pattern:
  - Allocate \(n\) bytes on stack for local vars
  - \texttt{subq}\ $n, \%\text{rsp} \quad \text{(or } \texttt{addl}\ $-n, \%\text{rsp})
**x86-64: Local Variables (2)**

- **Example:** Need space for two 64-bit local vars
  - `subq $16, %rsp` (or `addl $-16, %rsp`)
- **Local variables are at these offsets:**
  - `-8 (%rbp) = Local Var 1`
  - `-16 (%rbp) = Local Var 2`
- **Note:** These memory locations are *not* initialized!!
  - Contains whatever values were in that memory before the call...
  - Must initialize them before reading
- **When function ends, can clean up stack easily:**
  - `movq %rbp, %rsp`  # Discard local vars
  - `popq %rbp`        # Restore caller rbp
x86-64: Stack Frame?

- In x86-64, functions often don’t need to set up a frame pointer.
- Frame pointer is most useful when:
  - A function has more than 6 arguments (not particularly common)
  - A function must store local variables on the stack (with 16 registers, not likely)
- If function doesn’t require either of these things, a frame pointer is unnecessary.
  - Less code = faster program 🤗