x86-64

Science

Computer Science

CS 351: Systems Programming

Michael Lee <lee@iit.edu>

ILLINOIS TECH | College of Computing

x86-64 overview

- x86-64 is a 64-bit version of the x86 ISA
- Originally specified in 2000 by AMD as an alternative to IA-64 (“Itanium”)
- CISC ISA, so we have:
    - Memory operands for non-load/store instructions
    - Complex addressing modes
    - Relatively large number of instructions
        - We will only cover most common ones you’ll see

Coverage

- Syntax
- Registers
- Addressing modes
- Instructions
- Functions & Call stack

Syntax / Formatting

- Two common variants: Intel and AT&T syntax
- *Intel syntax* common in Windows world
  - e.g., `mov DWORD PTR [rbp-4], 10 ; format: OP DST, SRC`
- *AT&T syntax* common in UNIX world (default GCC output)
  - e.g., `movl $10, -4(%rbp) # format: OP SRC, DST`
  - We will use this syntax

Registers

- 16 64-bit “general purpose” registers
  - Many have a special purpose (e.g., stack pointer)
  - Each can be accessed as a 64/32/16/8-bit value (typically LSBs)
  - Each register is, by convention, *volatile* or *non-volatile*
    - A *volatile* register may be clobbered by a function call; i.e., its value should be saved — maybe on the stack — if it must be preserved
    - A *non-volatile* register is preserved (by callees) across function calls

Registers

Register(s)	Purpose	Volatile/Non-volatile	Lower 32 / 16 / 8 bits
`%rsp`	Stack pointer	Non-volatile	`%esp / %sp / %spl`
`%rbp`	Frame/Base pointer	Non-volatile	`%ebp / %bp / %bpl`
`%rax`	Return value	Volatile	`%eax / %ax / %ah, %al`
`%rbx`	Local variable	Non-volatile	`%ebx / %bx / %bh, %bl`
`%rcx`	—	Volatile	`%ecx / %cx / %ch, %cl`
`%rdx`	—	Volatile	`%edx / %dx / %dh, %dl`
`%rsi`	Source index (for arrays)	Volatile	`%esi / %si / %sil`
`%rdi`	Destination index (for arrays)	Volatile	`%rdi / %di / %dil`

`%r8-%r11`	—	Volatile	`%rNd / %rNw / %rNb`
			N ∈ {8-15}

`%r12-%r15`	Local variable	Non-volatile	
`%rip`	Program counter	(Cannot modify directly)	–

For function calls, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` are used as arguments 1-6 (before placing on stack)

Instruction operands

Mode	Example(s)	Meaning
Immediate	$0x42, $0xd00d	Literal value
Register	%rax, %rsp	Value found in register
Direct	0x4001000	Value found in address
Indirect	(%rsp)	Value found at address in register
Base-Displacement	8(%rsp), -24(%rbp)	Given D(B), value found at address D+B (i.e., address in base register B + numeric offset D)
Scaled Index	8(%rsp,%rsi,4)	Given D(B,I,S), value found at address D+B+I×S S ∈ {1,2,4,8}; D and I default to 0 if left out, S defaults to 1

Memory references

Instructions

- Instructions have 0-3 operands
  - For many 2 operand instructions, one operand is both read and written
    - e.g., `addl $1, %eax  # %eax = %eax + 1`
- Instruction suffix indicates width of operands (`q`/`l`/`w`/`b` → 64/32/16/8 bits)
- Arithmetic operations populate `FLAGS` register bits, including `ZF` (zero result), `SF` (signed/neg result), `CF` (carry-out of MSB occurred), `OF` (overflow occurred)
  - Used by subsequent conditional instructions (e.g., jump if result = zero)

Arithmetic

Instruction(s)	Description
`{add,sub,imul}` ***`src, dst`***	dst = dst {+,–,×} src
`neg` ***`dst`***	dst = –dst
`{inc,dec}` ***`dst`***	dst = dst {+,–} 1
`{sal,sar,shr}` ***`src, dst`***	dst = dst {<<,>>,>>>} src (arithmetic & logical shifts)
`{and,or,xor}` ***`src, dst`***	dst = dst {&,|,^} src (bitwise)
`not` ***`dst`***	dst = ~dst (bitwise)

`src` can be an immediate, register, or memory operand; ***`dst`*** can be a register or memory operand.
But at most one memory operand!

Conditions and Branches

Instruction(s)	Description
cmp *src, dst*	dst – src (discard result but set flags)
test *src, dst*	dst & src (discard result but set flags)
jmp *target*	Unconditionally jump to target (change `%rip`)
{je,jne} *target*	Jump to target if dst equal/not equal src (ZF=1 / ZF=0)
{jl,jle} *target*	Jump to target if dst </≤ src (SF≠OF / ZF=1 or SF≠OF)
{jg,jge} *target*	Jump to target if dst >/≥ src (ZF=0 and SF=OF / SF=OF)
{ja,jb} *target*	Jump to target if dst above/below src (CF=0 and ZF=0 / CF=1)

conditional jump often follows `cmp` (or `test`)

target is usually an address encoded as an immediate operand (e.g., `jmp $0x4001000`), but addresses may be stored in a register or memory, in which case *indirect addressing* is required, which uses the `*` symbol. E.g., `jmp *%rax` (jump to address in `%rax`), `jmp *0x4001000` (jump to address found at address 0x4001000)

Basic control structures

```
if (cond) {
    // if-clause
} else {
    // else-clause
}
...
```

```
    testq %rax, %rax   # %rax = cond
    je ELSE
    # if-clause
    jmp ENDIF
ELSE:
    # else-clause
ENDIF:
    # ...
```

```
while (cond) {
    // loop-body
}
...
```

```
    testq %rax, %rax   # %rax = cond
    je ENDLOOP
LOOP:
    # loop-body
    testq %rax, %rax
    jne LOOP
ENDLOOP:
    # ...
```

Data movement

Instruction(s)	Description
`mov src, dst`	Copy data from src to dst (memory→memory moves not possible)
`movzbq src, dst`	Copy 8-bit value to 64-bit target (& other variants), using zero-fill
`movsbq src, dst`	Copy 8-bit value to 64-bit target (& other variants), using sign-extension
`{cmove/ne} src, dst`	Move data from src to dst if ZF=1 / ZF=0
`{cmovg/ge/l/le/a/b/…}`	Conditionally move data from src to dst (per jump naming conventions)

Address computation

`lea address, dst`	dst = address (no memory access! just computes value of address)

Functions and Call stack

Instruction(s)	Description
push *src*	Push src onto stack
pop *dst*	Pop top of stack into dst
call *target*	Push current `%rip` (address of instruction after call) onto stack, jump to target
leave	Restore frame pointer (`%rbp`) and clears stack frame
ret	Pop top of stack into `%rip`

All instructions above implicitly adjust `%rsp` and access the stack.

target may use *indirect addressing* as well, e.g., `call *%rax` (call function whose address is in `%rax`)

Function calls

- Functions make extensive use of the call stack — leads to convention-driven *prologue* and *epilogue* blocks in assembly code
- Typical function prologue:
    - Save old frame pointer and establish new frame pointer
    - Save non-volatile register values we might clobber (“callee-saved”)
    - Load needed parameters from prior stack frame
    - Allocate stack space for any local data

Function calls

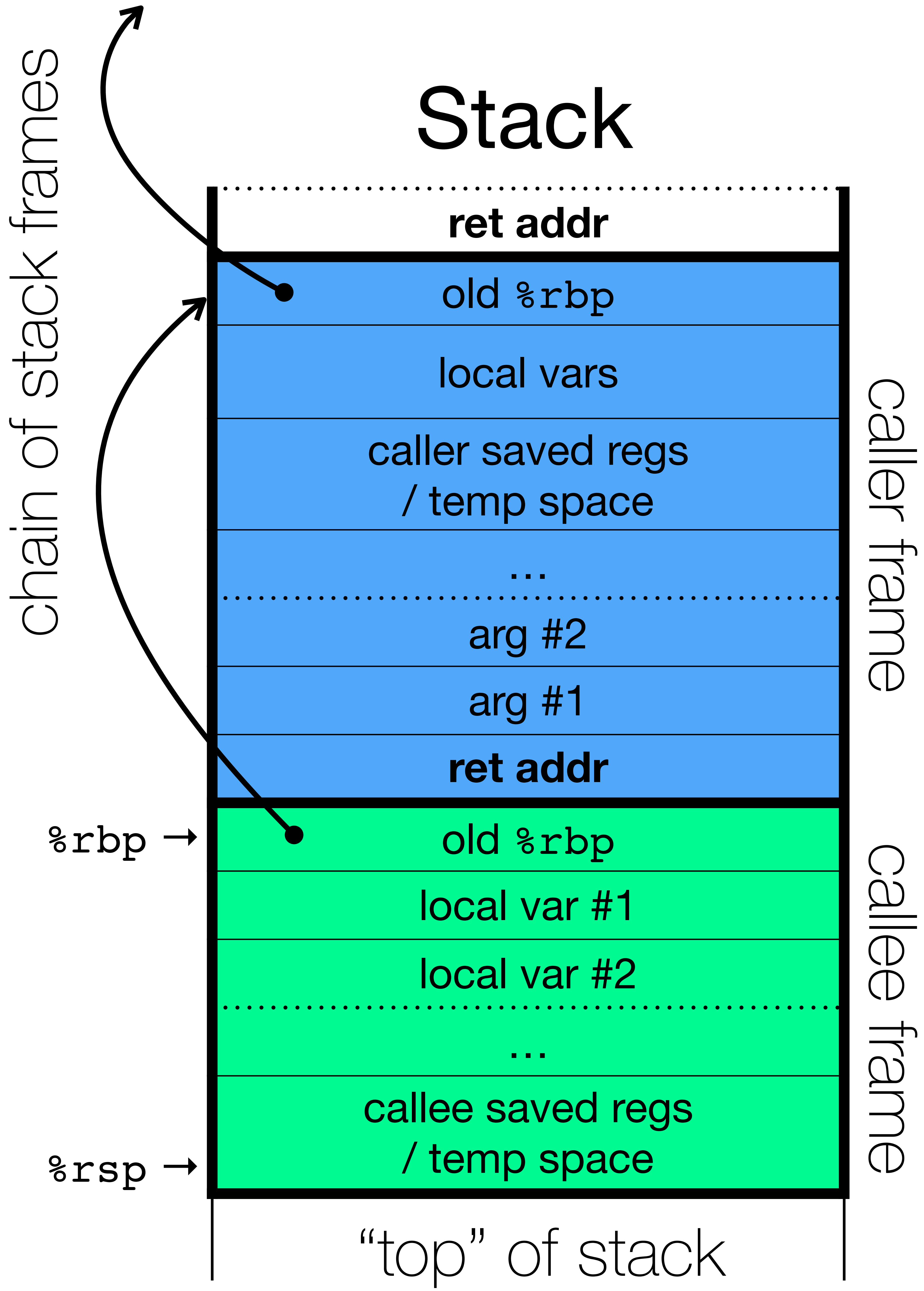
- Typical function epilogue:
  - Place return value in `%rax`
  - Deallocate any space used for local data
  - Restore/Pop any clobbered non-volatile register values
  - Restore/Pop old frame pointer
  - Return

Function calls (Optimization)

- Many of these steps may be optimized (simplified or neglected altogether) by the compiler!
  - Prefer registers to stack-based args or local vars (regs vs. memory)
  - `%rsp` doesn't always reflect the top of the stack (only need to do this if calling another function)
  - `lea` often used in surprising ways (addressing modes as arithmetic)

Call Stack

- Maintains dynamic state and context of executing program
- Saved frame pointers (previous values of `%rbp`) create a chain of stack frames
  - Useful to navigate for debugging and tracing! (e.g., gdb “backtrace”)



Function calls

```
int main() {
    int x=10, y=20;
    sum(x, y);
    return 0;
}

int sum(int a, int b) {
    int ret = a + b;
    return ret;
}
```

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $10, -4(%rbp)
    movl    $20, -8(%rbp)
    movl    -4(%rbp), %edi
    movl    -8(%rbp), %esi
    callq   sum
    movl    $0, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
```

```
sum: # unoptimized
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    -8(%rbp), %eax
    movl    %eax, -12(%rbp)
    movl    -12(%rbp), %eax
    popq    %rbp
    retq

sum: # optimized
    leal    (%rdi,%rsi), %eax
    retq
```