MONTANA
STATE UNIVERSITY

x86-64 Assembly Calling Conventions

Writing Code

Review: Registers in x64

- Recall the register setup in x86 64
- Today we are going to write some real code in x86
- To invoke this code we are going to have to learn about *calling conventions*

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

X64 Calling Conventions

- Calling conventions are the mechanism used by a given platform to pass arguments and return values from functions
- Typically Operating System and programming language specific

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

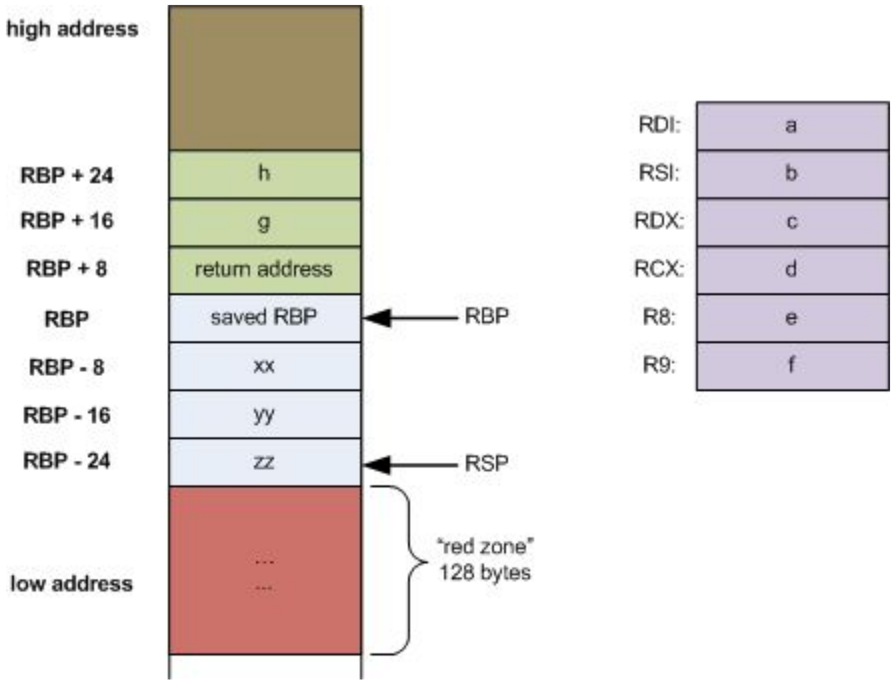
X64 Calling Conventions

- x64 Linux C programs have one set of calling conventions
- x64 Linux C++ programs have another
- x64 Windows C programs have yet another

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

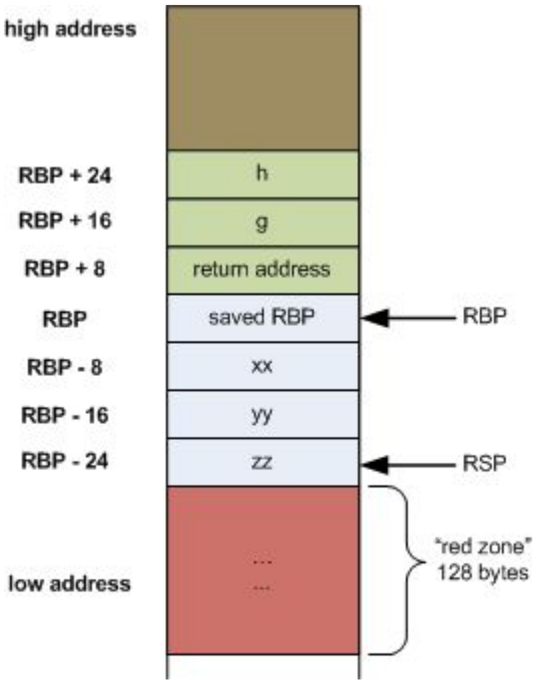
X64 Calling Conventions

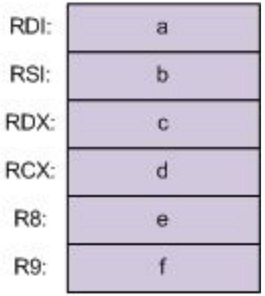
- x64 Linux C programs have one set of calling conventions
- x64 Linux C++ programs have another
- x64 Windows C programs have yet another



X64 Calling Conventions

- Let's consider integer arguments only (ints, pointers, etc)
  - Linux C conventions for integers & pointers:
    - 1st argument is in RDI
    - 2nd in RSI
    - RDX, RCX, R8, R9
    - Remainder are pushed on the stack in reverse order
    - RAX is used to return integer values





X64 Calling Conventions

- Let's go back to the simple add function that we looked at last time
- Now you can see what the RDI and RSI registers are doing

```
add:
        mov rax, rdi ;; move the first argument into rax
        add rax, rsi ;; add the second argument to that value
                     ;; and store in rax
        ret          ;; return value in rax
```

X64 Calling Conventions

- Invoking an assembly function from C requires that we declare and *extern* function (defined elsewhere)
- The linker then figures out the definition, coming from the assembly file & wires it up properly

```
extern int add(int i, int j);

int result = add( i: 1,  j: 2);
```

Syscall Calling Conventions

- A *syscall* is a call to low level functionality provided by the Operating System
- Requires an *interrupt* to pass control from the current process to the OS
- Slightly different calling convention

```
hello_world:
        mov rax, 1              ;; sys_write -
        mov rdi, 1              ;; stdout file
        mov rsi, message        ;; the message
        mov rdx, 13             ;; print 13 ch
        syscall
        ret
```

Syscall Calling Conventions

- We move the number 1 into rax to signal we want to call *sys_write*
- We move the number 1 into rdi to signal we want to write to standard out
- We move the memory location of a message into rsi
- We move the length of the message into rdx

```
hello_world:
        mov rax, 1              ;; sys_write -
        mov rdi, 1              ;; stdout file
        mov rsi, message        ;; the message
        mov rdx, 13             ;; print 13 ch
        syscall
        ret
```

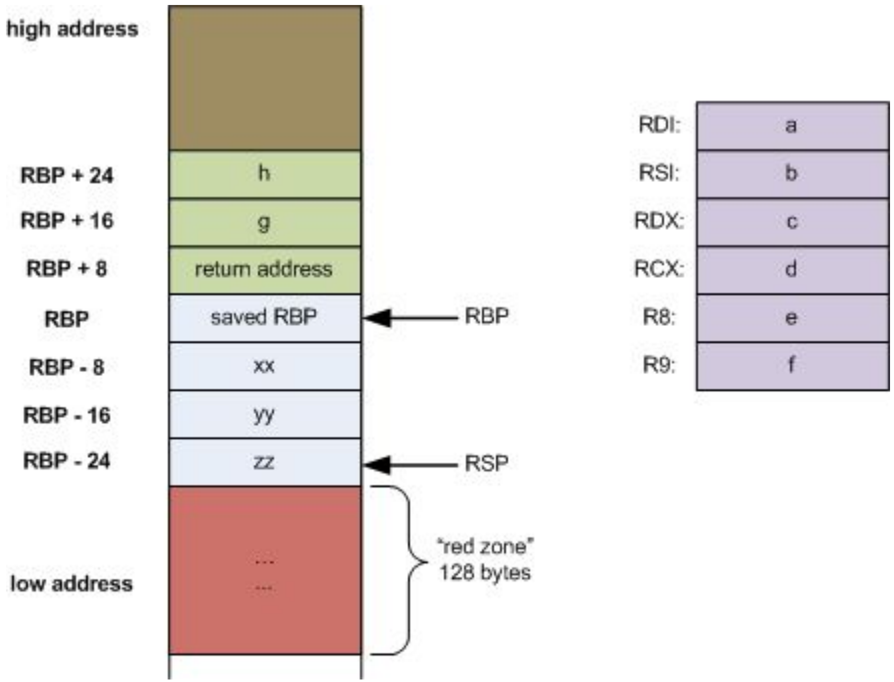
Syscall Calling Conventions

- We then issue the *syscall* instruction, which will interrupt the CPU and pass control to the OS to process
- Using the rax register the OS knows what syscall we want to issue and does its work

```
hello_world:
        mov rax, 1              ;; sys_write -
        mov rdi, 1              ;; stdout file
        mov rsi, message        ;; the message
        mov rdx, 13             ;; print 13 ch
        syscall
        ret
```

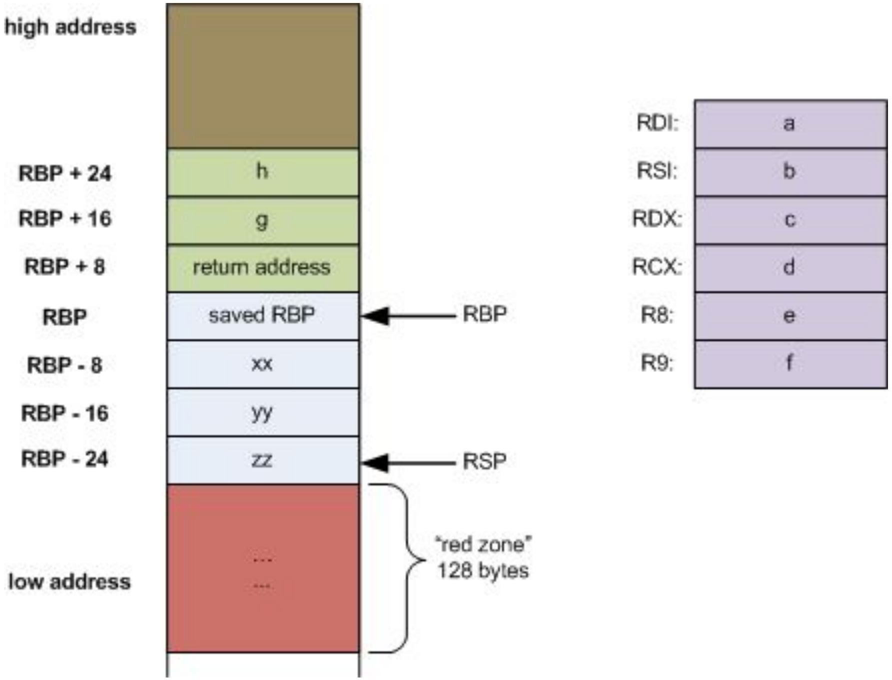
Register Preservation

- What if we call a function that calls another function
- What happens to registers?
- *It Depends™*
- Some registers are *caller saved*
  - Must be saved by the caller if they are in use
- Others are *callee saved*
  - Must be saved by the callee



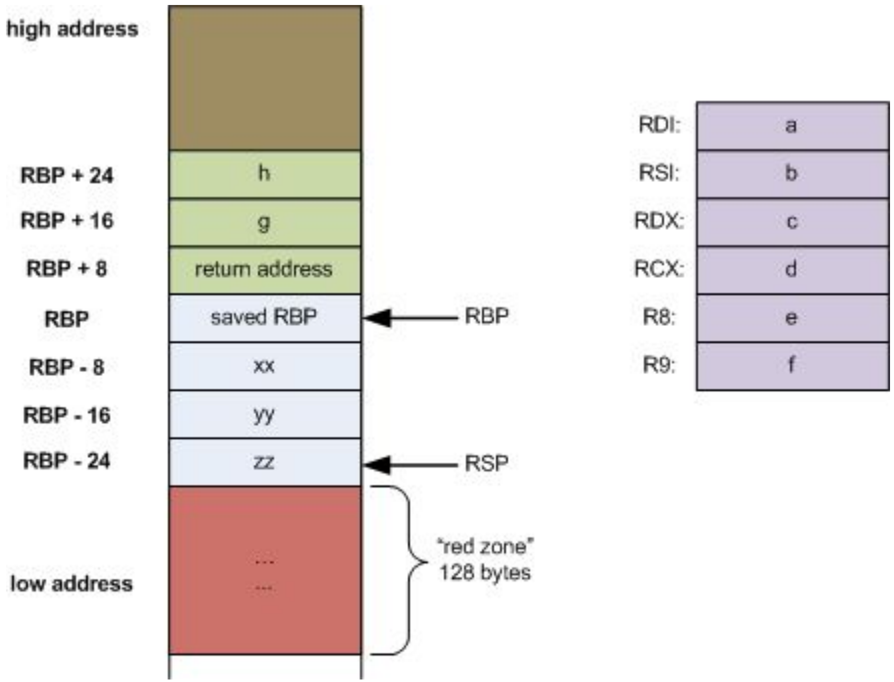
Register Preservation

- Caller saved registers
  - Any registers used to pass parameters (obviously)
  - R10 and R11
- Typically the caller would not store many registers



Register Preservation

- Callee saved registers
  - RBX, RBP (base pointer), R12, R13, R14, R15
  - So if a function is going to use any of these registers, it needs to save the value at the start and restore it before you return
  - The RBP, in particular, is typically modified, leading to something called the *function epilog*



Register Preservation

- Recall the assembly code from godbolt for our square function
    - Note the rbp is pushed on the stack to begin
    - Next the current stack pointer is moved into the base pointer
    - *Some logic happens*
    - Then the base pointer is restored

```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
cube:
```

Register Preservation

- This push and mov are called the *function prolog*
  - They don't have anything to do with the logic
- The pop is called the *function epilog*
- Together they are sometimes called the *function perilog*

```
square:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
        pop     rbp
        ret
cube:
```

Register Preservation

- Let's take a look at the generated assembly for the cube function, which invokes the square function
- Note that it looks a little different
  - Stores the base pointer and moves the current stack pointer into the base pointer
  - Then subtracts 8 bytes
  - Why?

```
cube:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 8
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        imul    eax, DWORD PTR [rbp-4]
        leave
        ret
```

Register Preservation

- Note that the function stores the edi register onto the stack, at the location of the base pointer - 4
- The sub instruction bumps the stack pointer so that the square function doesn't stomp on our values!

```
cube:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 8
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        imul    eax, DWORD PTR [rbp-4]
        leave
        ret
```

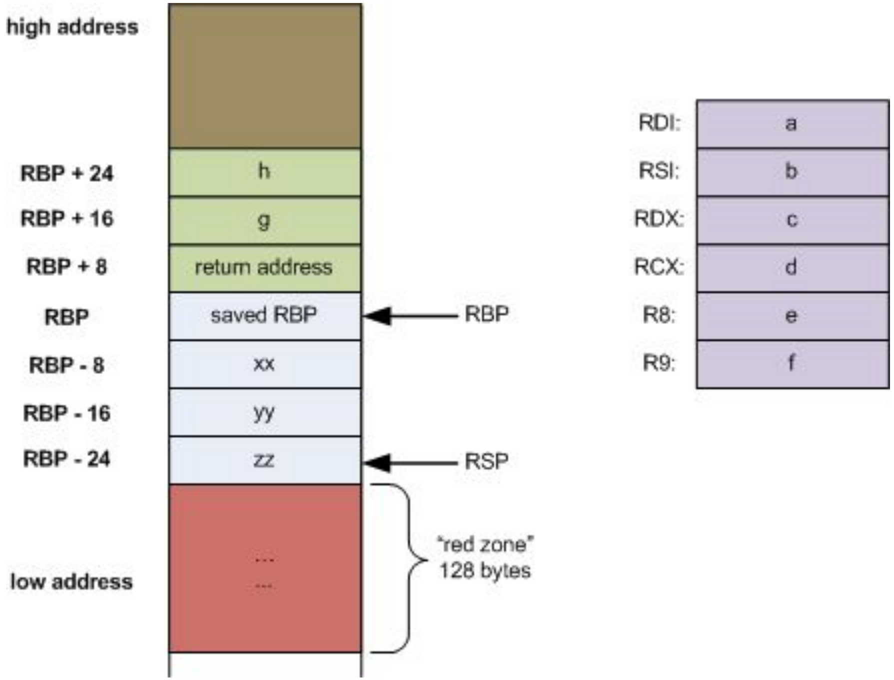
Register Preservation

- Here we see the stack growing down!
  - Pretty cool!
- But wait… why didn’t the square function do the same thing and bump the stack pointer?

```
cube:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 8
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        imul    eax, DWORD PTR [rbp-4]
        leave
        ret
```

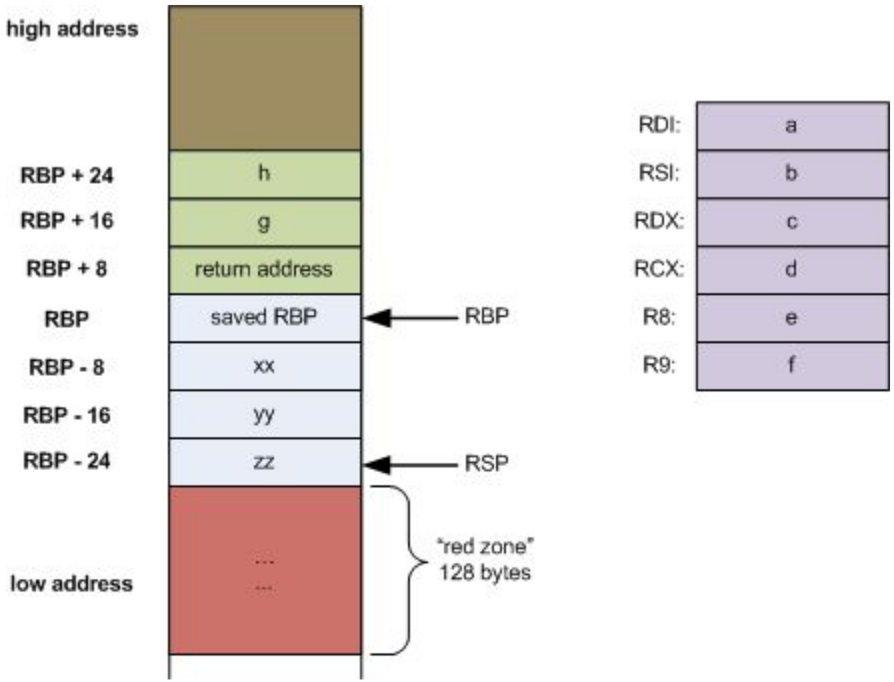
Register Preservation

- Note the *“Red Zone”* area
- The x64 architecture guarantees that the 128 bytes past the current stack pointer will not be updated by the OS or anyone else
- So a leaf function, which calls no other functions, may use this as a scratch area without bumping the stack pointer



Register Preservation

- Isn’t x64 fun?!



Register Preservation

- What about that leave instruction?
- It turns out that restoring the base pointer and decrementing the stack pointer is so common that these two operations were combined into one instruction on x86

```
cube:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 8
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        imul    eax, DWORD PTR [rbp-4]
        leave
        ret
```

Register Preservation

- Good example of the CISC vs RISC mindset!
- A very complex and specialized instruction
- MIPS omits this it in favor of multiple, simpler instructions

```
cube:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 8
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        mov     edi, eax
        call    square
        imul    eax, DWORD PTR [rbp-4]
        leave
        ret
```

Pointers

- We have been working with integers so far, what about pointers?
- Pointers *are* integers
- Are passed and returned in the same registers

```
nth_char:
    mov rax, rdi
    add rax, rsi
    mov rax, [rax]
    ret
```

Pointers

- Here is a function that returns the nth char in a string
- Arguments are a char * pointer and an int
- Move the pointer (rdi) into rax
- Increment it by rsi
- Dereference the pointer into memory

```
nth_char:
    mov rax, rdi
    add rax, rsi
    mov rax, [rax]
    ret
```

Pointers

- Bracket Syntax
  - Brackets around a register mean “the memory pointed to by this register”
  - Again, this is another example of CISC
  - No LOAD/STORE instructions, just MOV with different types of arguments
  - You will need to use this syntax for homework 3

```
nth_char:
    mov rax, rdi
    add rax, rsi
    mov rax, [rax]
    ret
```

Pointers

- What does the godbolt code look like for this?
- Much more complex!
    - Stores parameters to the stack… why?
        - Probably for debugging
    - Obviously the intel syntax is a little rougher
    - Unnecessary perilog code...

```
nth_char:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     DWORD PTR [rbp-12], esi
        mov     eax, DWORD PTR [rbp-12]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-8]
        add     rax, rdx
        movzx   eax, BYTE PTR [rax]
        pop     rbp
        ret
```

Pointers

- This is why people will sometimes write assembly by hand
  - More efficient than compiler generated code
- On the other hand, this will make almost no difference in most situations

```
nth_char:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     DWORD PTR [rbp-12], esi
        mov     eax, DWORD PTR [rbp-12]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-8]
        add     rax, rdx
        movzx   eax, BYTE PTR [rax]
        pop     rbp
        ret
```

Pointers

- Raise your hand if you want to write x86 assembly for a career...

```
nth_char:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     DWORD PTR [rbp-12], esi
        mov     eax, DWORD PTR [rbp-12]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-8]
        add     rax, rdx
        movzx   eax, BYTE PTR [rax]
        pop     rbp
        ret
```

x64 Assembly

- Today we looked at the calling conventions for x86 assembly on linux
- We discussed registers and how they are used
- We discussed caller vs callee saved registers
- We took a look at how to implement some practical functions in assembly and compared with the gcc compiler output
- *IT'S JUST CODE*

MONTANA
STATE UNIVERSITY