CS 33 – Discussion 1A
Computer System Organization

Week 3

Questions before we start?

Logistics

- Lab 1 - Datalab due tonight 11:59pm
- Lab 2 – Bomb lab released tomorrow – Due Friday May 1st 11:59
  - Probably have HW 2 assigned and due before then
- Please submit suggestions on CCLE TA-site on how to improve OH or DISC

Agenda

10am-11am PST:

- x86-64 basics continued
  - Condition codes
  - Conditional Branching
  - Switch statements/Jump tables
  - Structs/Data alignment
  - Register saving conventions
- Recursion/Tail Recursion code example

11am-11:50am PST:

- LA worksheet

Agenda

If time permits:

- Detail Bomblab
- Answer questions on Datalab

x86-64 Basics – Condition codes

- Condition codes are set implicitly by arithmetic instructions
- Explicitly set by setX instructions or cmp/test instruction

Instruction		Description	Page #
`cmp`	S_2, S_1	Set condition codes according to S_1 - S_2	185
`test`	S_2, S_1	Set condition codes according to S_1 & S_2	185

- Can be thought of as single bit registers

x86-64 Basics – Condition codes

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: `addq` *Src,Dest* ↔ `t = a+b`

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow

  ```
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  ```

- **Not set by `leaq` instruction**

x86-64 Basics – Condition codes

Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: `addq` *Src,Dest* ↔ `t = a+b`

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow

  ```
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  ```

- **Not set by `leaq` instruction**

x86-64 Basics – Condition codes

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpq` *Src2, Src1*
  - `cmpq b,a` like computing `a-b` without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if `a == b`
  - **SF set** if `(a-b) < 0` (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

x86-64 Basics – Condition codes

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - `testq` *Src2*, *Src1*
    - `testq b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - **ZF set** when `a&b == 0`
  - **SF set** when `a&b < 0`

x86-64 Basics – Condition codes

- **SetX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

x86-64 Basics – Conditional Branching

3.4.2 Jump Instructions

Instruction		Description	Condition Code	Page #
`jmp`	*Label*	Jump to label		189
`jmp`	**Operand*	Jump to specified location		189
`je / jz`	*Label*	Jump if equal/zero	`ZF`	189
`jne / jnz`	*Label*	Jump if not equal/nonzero	`~ZF`	189
`js`	*Label*	Jump if negative	`SF`	189
`jns`	*Label*	Jump if nonnegative	`~SF`	189
`jg / jnle`	*Label*	Jump if greater (signed)	`~(SF^0F)&~ZF`	189
`jge / jnl`	*Label*	Jump if greater or equal (signed)	`~(SF^0F)`	189
`jl / jnge`	*Label*	Jump if less (signed)	`SF^0F`	189
`jle / jng`	*Label*	Jump if less or equal	`(SF^OF)|ZF`	189
`ja / jnbe`	*Label*	Jump if above (unsigned)	`~CF&~ZF`	189
`jae / jnb`	*Label*	Jump if above or equal (unsigned)	`~CF`	189
`jb / jnae`	*Label*	Jump if below (unsigned)	`CF`	189
`jbe / jna`	*Label*	Jump if below or equal (unsigned)	`CF|ZF`	189

x86-64 Basics – Switch Statement

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

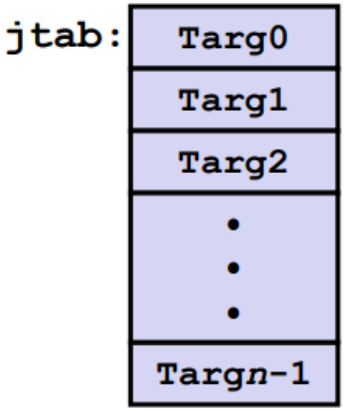
x86-64 Basics – Switch Statement

Jump Table Structure

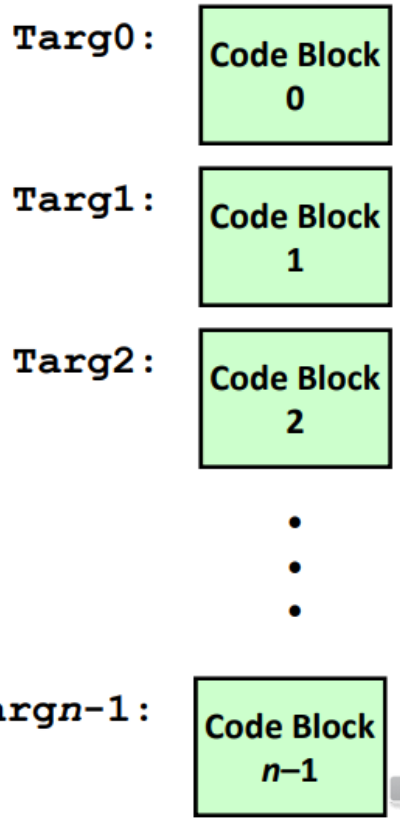
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Translation (Extended C)

```
goto *JTab[x];
```

x86-64 Basics – Switch Statement

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Indirect jump →

x86-64 Basics – Switch Statement

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```

x86-64 Basics – Switch Statement

Code Blocks (x == 2, x == 3)

```
    long w = 1;
       . . .
    switch(x) {
       . . .
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
       . . .
    }
```

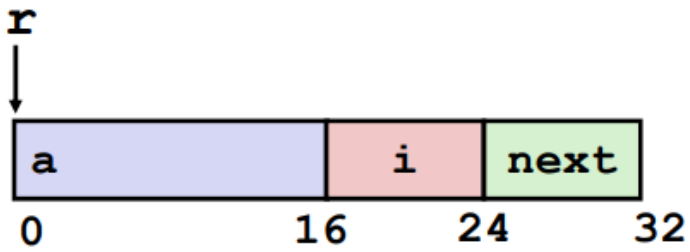
```
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3
   movl    $1, %eax         #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax       #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

x86-64 Basics – Structs/Data align

Structure Representation

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

x86-64 Basics – Structs/Data align

Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

x86-64 Basics – Structs/Data align

Specific Cases of Alignment (x86-64)

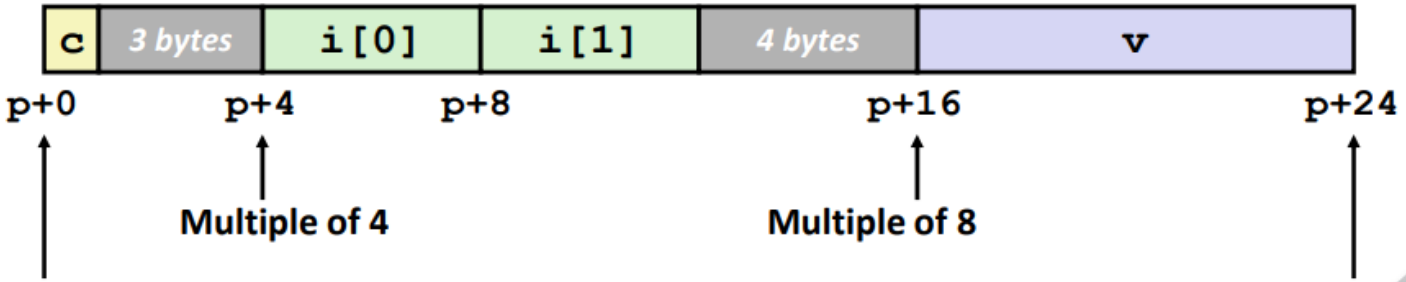
- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2
- **16 bytes: `long double`** (GCC on Linux)
  - lowest 4 bits of address must be 0000_2

x86-64 Basics – Structs/Data align

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to **`double`** element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



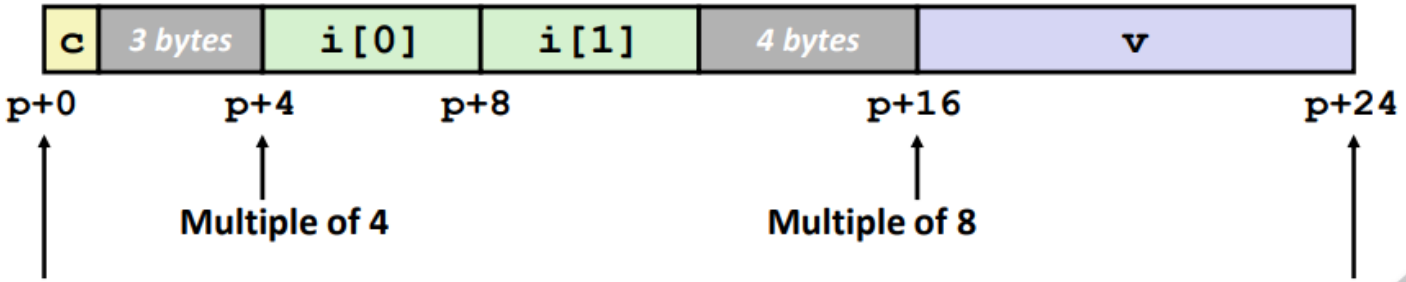
- **For largest alignment requirement K**
- **Overall structure must be multiple of K**

x86-64 Basics – Structs/Data align

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example:**
  - K = 8, due to **`double`** element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



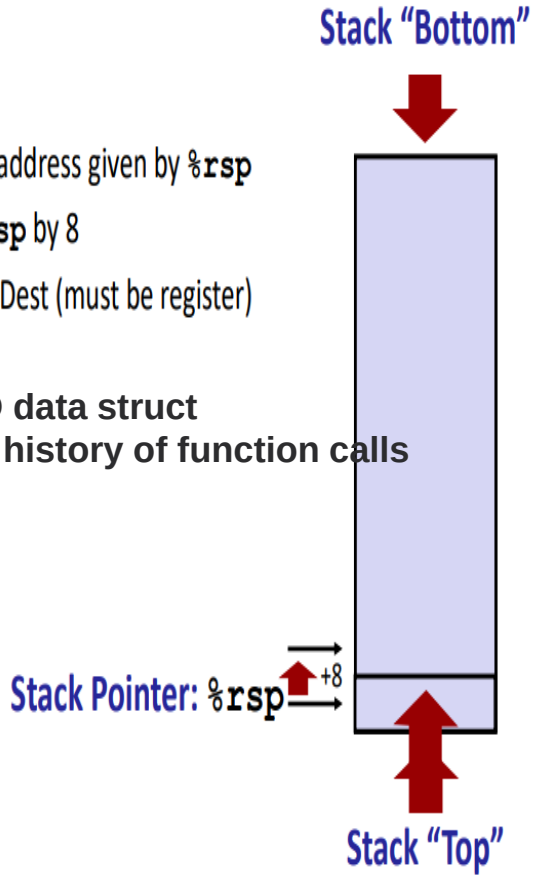
- **For largest alignment requirement K**
- **Overall structure must be multiple of K**

x86-64 Basics – Calling Convention

x86-64 Stack: Pop

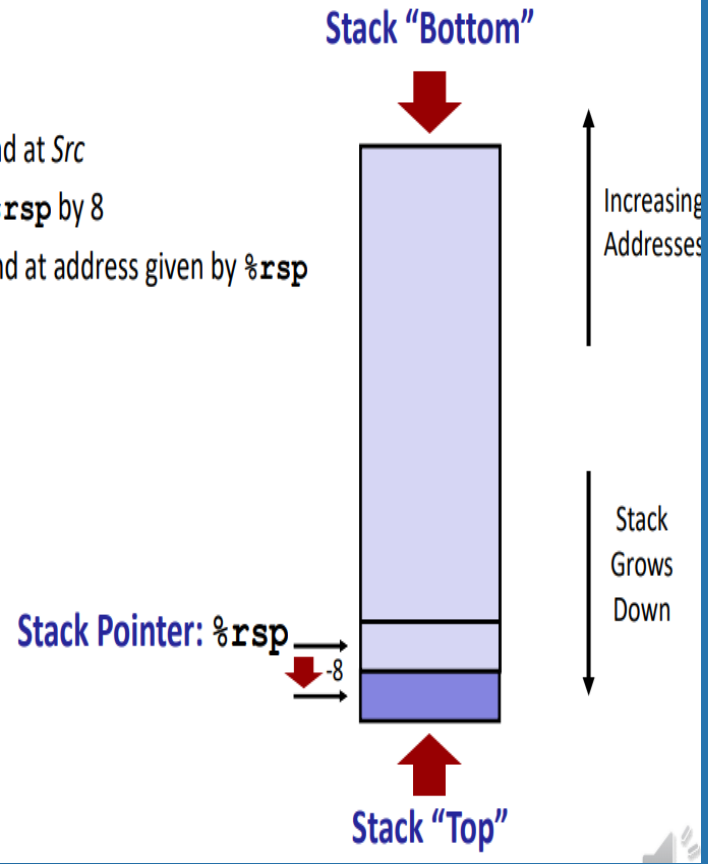
- popq *Dest*
  - Read value at address given by %rsp
  - Increment %rsp by 8
  - Store value at Dest (must be register)

Stack is LIFO data struct
Contains the history of function calls



x86-64 Stack: Push

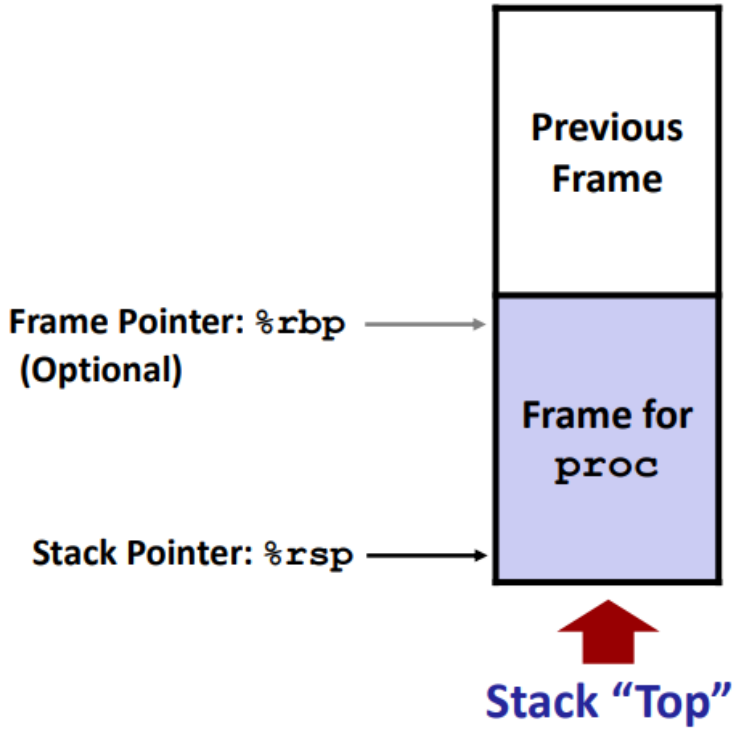
- pushq *Src*
  - Fetch operand at *Src*
  - Decrement %rsp by 8
  - Write operand at address given by %rsp



x86-64 Basics – Calling Convention

Stack Frames

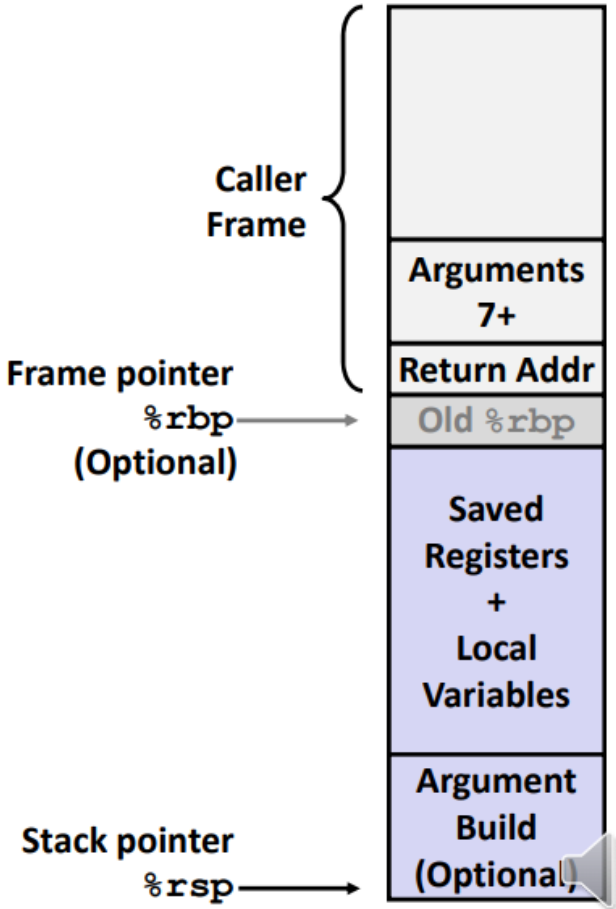
- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- **Management**
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by **`call`** instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by **`ret`** instruction



x86-64 Basics – Calling Convention

x86-64/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



x86-64 Basics – Calling Convention

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination

x86-64 Basics – Calling Convention

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- Contents of register `%rdx` overwritten by `who`
- This could be trouble → something should be done!
  - Need some coordination

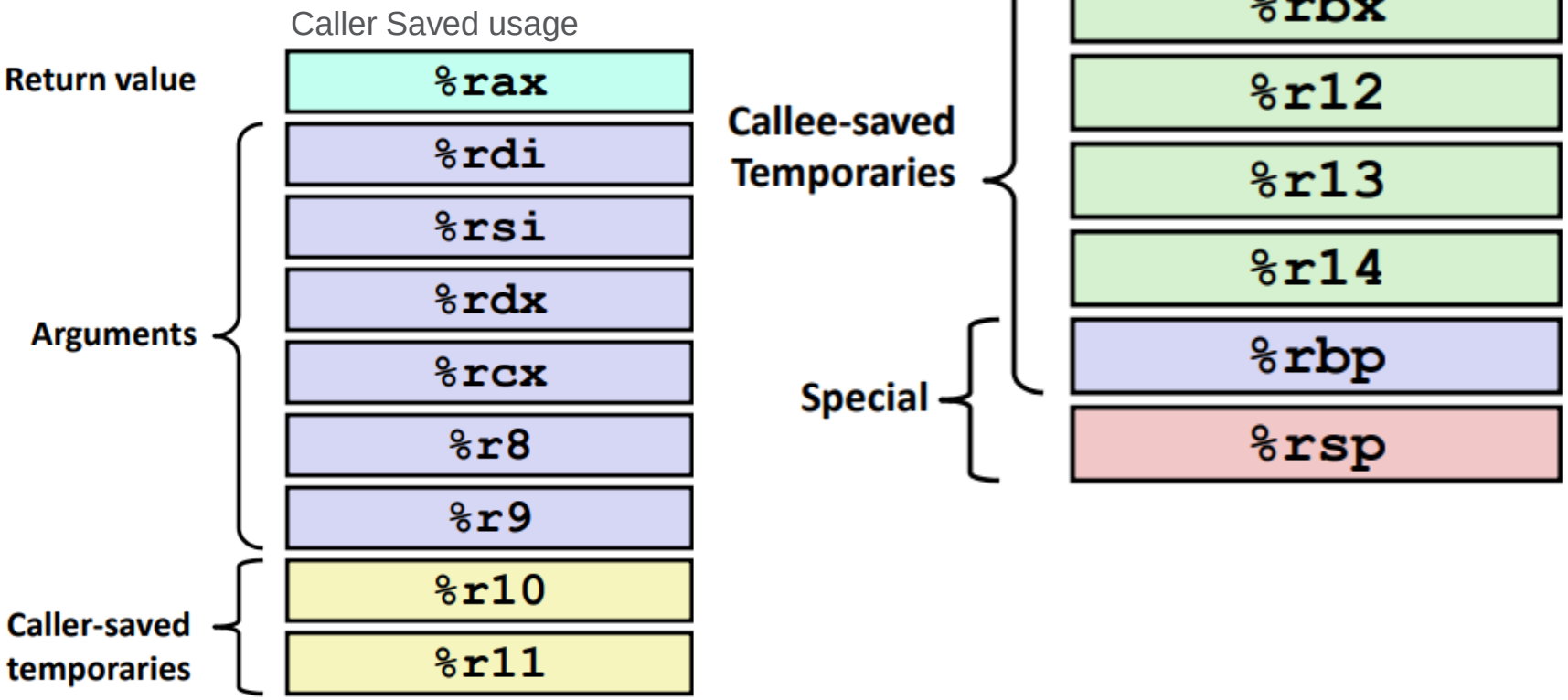
x86-64 Basics – Calling Convention

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*
  - `who` is the *callee*
- Can register be used for temporary storage?
- Conventions
  - *"Caller Saved"*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved"*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

x86-64 Basics – Calling Convention

Usage #1



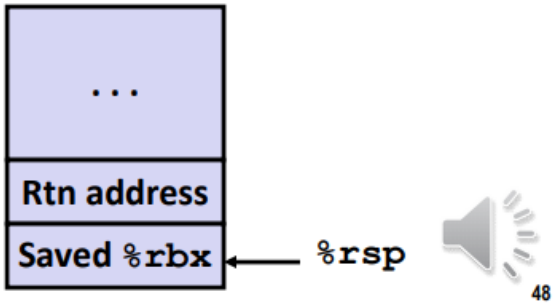
x86-64 Basics – Recursion

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  ret
```

Register	Use(s)	Type
%rdi	x	Argument



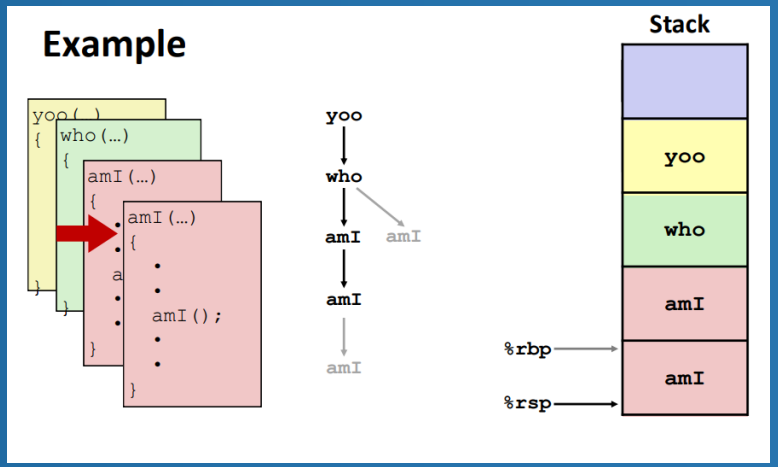
Notes adapted from Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

48

x86-64 Basics – Recursion

- A potential problem: stack overflow
- To reduce the chance of stack overflow:
- Tail-recursion: make the last instruction in the function be the recursive call (last instruction is the function calling itself)
- Tail recursive function calls don't need to return to caller, since there's nothing left in the caller to be handled.
  - No need to store the caller's address
  - No need to store different stack frames for each invocation
  - Saves tons of space on stack

Example of stack with regular function calls, can easily overload with stack frames :



x86-64 Basics – Recursion

- How do we make this function tail recursive?
- What does the stack look like in regular / tail recursive example?
- Lets investigate!
- Compiler should optimize recursion into loop

```
00000000004004c4 <factorial>:
  4004c4:       53                      push   %rbx
  4004c5:       89 fb                   mov    %edi,%ebx
  4004c7:       b8 00 00 00 00          mov    $0x0,%eax
  4004cc:       85 ff                   test   %edi,%edi
  4004ce:       74 12                   je     4004e2 <factorial+0x1e>
  4004d0:       b0 01                   mov    $0x1,%al
  4004d2:       83 ff 01                cmp    $0x1,%edi
  4004d5:       74 0b                   je     4004e2 <factorial+0x1e>
  4004d7:       8d 7b ff                lea    -0x1(%rbx),%edi
  4004da:       e8 e5 ff ff ff          callq  4004c4 <factorial>
  4004df:       0f af c3                imul   %ebx,%eax
  4004e2:       5b                      pop    %rbx
  4004e3:       c3                      retq

00000000004004e4 <main>:
  4004e4:       48 83 ec 08             sub    $0x8,%rsp
  4004e8:       bf 03 00 00 00          mov    $0x3,%edi
  4004ed:       e8 d2 ff ff ff          callq  4004c4 <factorial>
  4004f2:       89 c6                   mov    %eax,%esi
  4004f4:       bf 08 06 40 00          mov    $0x400608,%edi
  4004f9:       b8 00 00 00 00          mov    $0x0,%eax
  4004fe:       e8 b5 fe ff ff          callq  4003b8 <printf@plt>
  400503:       b8 01 00 00 00          mov    $0x1,%eax
  400508:       48 83 c4 08             add    $0x8,%rsp
  40050c:       c3                      retq
  40050d:       90                      nop
  40050e:       90                      nop
  40050f:       90                      nop
```

```c
#include <stdlib.h>
#include <stdio.h>

unsigned int factorial (unsigned int x)
{
  if (x==0)
    return 0;
  else
    if (x==1)
      return 1;
    else
      return x*factorial(x-1);
}

int main( int argc, const char* argv[] )
{

  printf("%d\n", factorial(3));

  return 1;

}
```

Questions before we start worksheet?

Resources used

- https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
- https://cs.stackexchange.com/questions/76871/how-are-variables-stored-in-and-retrieved-from-the-program-stack
- https://www.geeksforgeeks.org/memory-layout-of-c-program/
- https://vim.rtorr.com/
- https://www.tenouk.com/ModuleW.html
- Professor Reinman's slides (CCLE)
- **Credit for slides: Attiano Purpura-Potoniere**