CSE 2421

X86-64Assembly Language – Part 1:
Stack, registers, assembler directives, and
data movement instructions

Section 1 Programs and Addressing

Assembler directives (“pseudo-ops”)

- .file
  - Allows a name to be assigned to the assembly language source code file.
- .section
  - This makes the specified section the current section.
  - .rodata
    - Specifies that the following data is to be placed in the read only memory portion of the executable
- .string
  - Specifies that the characters enclosed in quotation marks are to be stored in memory, terminated by a null byte
- .data
  - Changes or sets the current section to the data section
- .text
  - Changes or sets the current section to the text (or code) section

Assembler directives (continued)

- .globl
  - A directive needed by the linker for symbol resolution: followed by name of function
- .type
  - Needed by the linker to identify the label as one associated with a function, as opposed to data
- .size
  - Needed by the linker to identify the size of the text for the program

- **Note**: labels (for functions or data) in assembly language source code are followed by a colon.

Data size assembler directives

- .quad *value*
  - Places the given value, (0x prefix for hex, no prefix for decimal) in memory, encoded in 8 bytes
- .long *value*
  - Places the given value, (0x prefix for hex, no prefix for decimal) in memory, encoded in 4 bytes
- .word *value*
    - Places the given value, (0x prefix for hex, no prefix for decimal) in memory, encoded in 2 bytes
- .byte *value*
    - Places the given value, (0x prefix for hex, no prefix for decimal) in memory, encoded in 1 byte

Run X86 program

```
.file “first.s”
.section .rodata
.data
    .align 8
Array:
    .quad 0x6f
    .quad 0x84
```

```
.text
.globl main
    .type main, @function
main:
    pushq %rbp
    movq %rsp, %rbp

    movq $55,%rdx
    movq %rdx, %rbx
    movq $Array, %rax
    movq %rbx,8(%rax)
    movq (%rax),%rcx

    leave
    ret
.size main, .-main
```

Run X86 program

```
.file “second.s”
.section .rodata
.data
.align 8
Array:
.quad 0x6f
.quad 0x84
.quad 0x55
.quad 0x44
.globl main
    .type main, @function
.text
```

```
main:
pushq %rbp
movq %rsp, %rbp

movq $55,%rdx
movq %rdx, %rbx
movq $0x33, %r8
movq $Array, %rax
movq %rbx,8(%rax)
movq %r8, 24(%rax)
movq %rax,(%rax)
movq (%rax),%rcx

leave
ret
.size main, .-main
```

Assembly Syntax

- Immediate values are preceded by $
  $ -> decimal value
  $0x –> hex value

- Registers are prefixed with %

- Moves and ALU operations are source, destination:
  movq $5, %rax
  movq $0x30, %rbx

- Effective address *DISPLACEMENT(BASE)*

movq $0x30, 8(%rbx)

Review

- What is the size of a memory address on stdlinux???
- So what the only suffix should we be using when we are calculating/moving addresses?
- What size registers should we be using when we are calculating addresses?
- Is there ever an exception to this?

Review

- What is the size of a memory address on stdlinux???

  8 bytes = 64 bits

- So what is the only suffix should we be using when we are calculating/moving addresses?

- What size registers should we be using when we are calculating addresses?

- Is there ever an exception to this?

Review

- What is the size of a memory address on stdlinux???

  8 bytes = 64 bits

- So what is the only suffix should we be using when we are calculating/moving addresses?

  q

- What size registers should we be using when we are calculating addresses?

- Is there ever an exception to this?

Review

- What is the size of a memory address on stdlinux???

  8 bytes = 64 bits

- So what is the only suffix should we be using when we are calculating/moving addresses?

  q

- What size registers should we be using when we are calculating addresses?

  %rax,%rbx, %rcx, %rdx, %r12, etc.

- Is there ever an exception to this?

Review

- What is the size of a memory address on stdlinux???

  8 bytes = 64 bits
- So what is the only suffix should we be using when we are calculating addresses?

  q
- What size registers should we be using when we are calculating/moving addresses?

  %rax,%rbx, %rcx, %rdx, %r12, etc.
- Is there ever an exception to this?

  Not ever!

  (as long as we are working on a 64 bit processor.)

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

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

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

    `movq (%rcx),%rax`
  - Are any of these a valid instruction on stdlinux?

    `movq (%ecx),%rax`

    `movl (%ecx),%eax`

    `movb (%rax),%al`

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

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

  - Are any of these a valid instruction on stdlinux?

```
movq (%ecx),%rax    #No. must use %rcx
movl (%ecx),%eax
movb (%rax),%al
```

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

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

  - Are any of these a valid instruction on stdlinux?

```
movq (%ecx),%rax     #No. must use %rcx
movl (%ecx),%eax     #No. must use %rcx
                     # l suffix and dest
                     # of %eax is OK

movb (%rax),%al
```

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

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

  - Are any of these a valid instruction on stdlinux?

```
movq (%ecx),%rax     #No. must use %rcx
movl (%ecx),%eax     #No. must use %rcx
                     # l suffix and dest
                     # of %eax is OK
movb (%rax),%al      #Yes! Address is 8
                     #byte reg, suffix          #and dest
are 1 byte
```

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Complete Memory Addressing Modes

See Figure 3.3 page 181

- Most General Form

  Imm(Rb,Ri,S) Mem[Imm+ Reg[Rb]+S*Reg[Ri]]
  - Imm: Constant "displacement"
    - It's often a "displacement" of 1, 2, 4 or 8 bytes, but can be any constant value
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **%rsp**
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
  - This form is seen often when referencing elements of arrays

- Special Cases

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
Imm(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+Imm]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

Complete Memory Addressing Modes

- Examples:

`movq 24(%rax,%rcx,8), %rdx`

means read 8 bytes from this address: (%rax + 8*%rcx + 24)
and store it in %rdx

`movl 24(%rax,%rcx,4), %edx`

means read 4 bytes from this address: (%rax + 4*%rcx + 24)
and store it in %edx

`movw 24(%rax,%rcx,2), %dx`

means read 2 bytes from this address: (%rax + 2*%rcx + 24)
and store it in %dx

`movb 24(%rax,%rcx,1), %dl`

means read 1 byte from this address: (%rax + 1*%rcx + 24)
and store it in %dl

Note that suffix and destination register size match. The change in scale is only so that the example is sensible.

Complete Memory Addressing Modes

- Examples:

`movq %rdx, 24(%rax,%rcx,8)`

means write 8 bytes to this address: (%rax + 8*%rcx + 24) from %rdx

`movl %edx, 24(%rax,%rcx,4)`

means write 4 bytes to this address: (%rax + 4*%rcx + 24) from %edx

`movw %dx, 24(%rax,%rcx,2)`

means write 2 bytes to this address: (%rax + 2*%rcx + 24) from %dx

`movb %dl, 24(%rax,%rcx,1)`

means write 1 byte to this address: (%rax + 1*%rcx + 24) from %dl

Note that suffix and destination register size match. The change in scale is only so that the example is sensible.

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)		
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)		

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Section 2: Computations & the stack

Address Computation Instruction

- **`leaq` *Src*, *Dst***
  - *Load Effective Address*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
  - Doesn't affect condition codes (AGU op instead of an ALU op)
  - http://stackoverflow.com/questions/1658294/whats-the-purpose-of-the-lea-instruction

- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
    - e. g. if %rdx contains a value x, then **leaq 7(%rdx, %rdx,4), %rax** sets %rax to 5x+7
- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

~~Stupid math tricks~~ Computations with lea

- Consider:

```
leaq (%rdi, %rdi,1), %rax    =>   %rdi  + 1*%rdi = 2%rdi
leaq (%rdi, %rdi,2), %rax    =>   %rdi + 2*%rdi = 3%rdi
leaq (%rdi, %rdi,4), %rax    =>   %rdi + 4*%rdi = 5%rdi
leaq (%rdi, %rdi,8), %rax    =>   %rdi + 8*%rdi = 9%rdi
```

- What kind of multiplication problems can you come up with that might make these valuable?

```
leaq(%rdi, %rdi,2), %rax                  # 3%rdi
leaq(%rdi, %rdi,8), %rbx                  # 9%rdi
addq %rbx, %rax                 # 12%rdi
```

Some Arithmetic Operations

- Two Operand Instructions:

Format	*Computation*		
`add`	*Src,Dest*	Dest = Dest + Src	
`sub`	*Src,Dest*	Dest = Dest – Src	
`imul`	*Src,Dest*	Dest = Dest * Src	signed multiply
`mul`	*Src,Dest*	Dest = Dest * Src	unsigned multiply
`idiv`	*Src,Dest*	Dest = Dest / Src	signed divide
`div`	*Src,Dest*	Dest = Dest / Src	unsigned divide
`sal`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sar`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic (fills w/copy of sign bit)***
`shr`	*Src,Dest*	Dest = Dest >> Src	***Logical (fillls with 0s)***
`xor`	*Src,Dest*	Dest = Dest ^ Src	
`and`	*Src,Dest*	Dest = Dest & Src	
`or`	*Src,Dest*	Dest = Dest | Src	

- Watch out for argument order!
- Except for mul and div, no distinction between signed and unsigned int (why?)
- Don’t forget to include a suffix for each of these instructions.

Some Arithmetic Operations

- One Operand Instructions

`inc`	*Dest*	*Dest = Dest + 1*
`dec`	*Dest*	*Dest = Dest – 1*
`neg`	*Dest*	*Dest = – Dest*
`not`	*Dest*	*Dest = ~Dest*

- See book for more instructions (Figure 3.10)

- Obviously, each of these instructions must use the appropriate suffix based on the Destination size

Arithmetic Expression Example

(z+x+y)*((x+4)+(y*48))

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax   # t1 = x+y
  addq    %rdx, %rax          # t2 = z + t1
  leaq    (%rsi,%rsi,2),%rdx  # %rdx = y+2y
  salq    $4, %rdx            # %rdx * 16
  leaq    4(%rdi,%rdx), %rcx  # x + t4 + 4
  imulq   %rcx, %rax          # t2=t2*t5
  ret
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift arithmetic left
- **`imulq`**: signed multiply
  - But, only used once

Understanding Arithmetic Expression Example

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

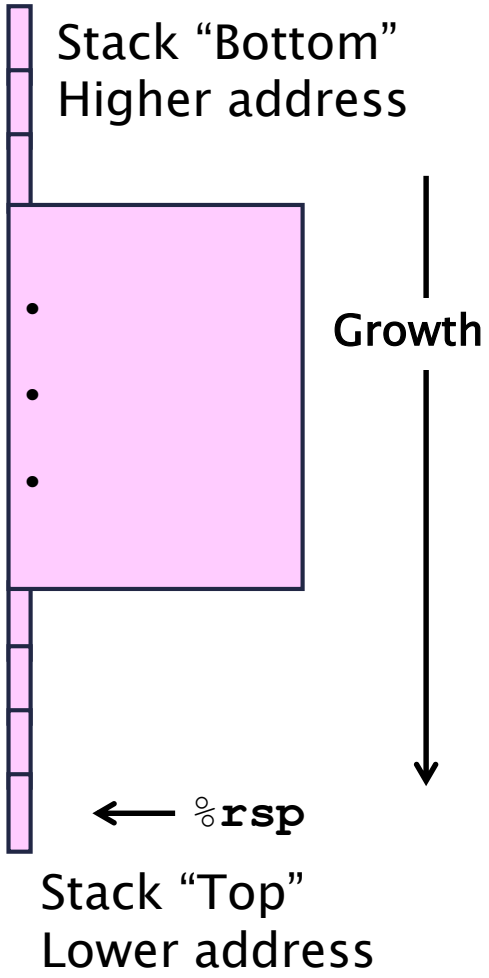
X86 program stack

- The program stack is actually divided conceptually into *frames*.
- Each procedure or function (main and any functions called from main or from another function) has its own part of the stack to use, which is called its frame.
- The frame goes from the stack address pointed to by %rbp in that procedure, this is called the frame (or base) pointer, to %rsp, which points to the top of the stack while the procedure is running.
- This implies that the address pointed to by %rbp is different in different procedures: %rbp must be set when the procedure is entered.

X86 Stack

- Stack top address always held in register %rsp
- Stack grows towards lower addresses

- Where is %rbp???
  - That depends…☺



Use of the stack in X86-64

- To save the caller's %rbp (frame pointer) before setting its own frame pointer;
- To preserve values needed after return before calling another function;
- To pass parameters to another function (if there are more than 6 parameters to pass);
- To store the return address when a call instruction is executed.
- If more data than registers, automatic variables

Procedure calls and returns

- To use procedure calls and returns in our X86 program, we have to manage the program stack and program registers correctly.
- Two different aspects to this:
  - Maintain the stack pointer and associated data in relation to each procedure call and return. (The OS initializes these values upon system start.)
  - Place appropriate values in “some” registers as expected by a calling or caller program. More on this later.

Setting up the program stack

- In X86 programs, you must set up the stack frame in your assembly language source code.
- There are three things to do:
  - At the start of a function:
    - Set %rbp to point to the bottom of the current stack frame.
    - Set %rsp to point to the top of the stack (the same address as the stack bottom initially).
  - At the end of a function:
    - Put them back
- The next slide shows a typical way of doing it.

Setting up the stack

Part 1:

pushq %rbp **# Save caller's base pointer**

movq %rsp, %rbp **# Set my base pointer**

Put these two instructions at the beginning of your function before any other statements!

* Notice that, since %rbp equals %rsp, the stack is empty.

* We are now ready to use the stack!

Part 2:

leave **# set caller's stack frame back up**

Put this statement directly before the **ret** instruction of your program.

Section 3: Functions and Registers

Call instruction

- **call Dest**
- *Dest* will be a label which has been placed in the assembly language source code at the address of the procedure to be called.
- When the **call** instruction is executed, the address of the instruction after the call instruction is pushed onto the stack (that is, the return address is pushed), and the address of Dest is assigned to the PC (%rip).
- This means that, when the called procedure begins execution, the return address is the last thing that has been pushed onto the stack.

How do we determine return address?

- PC typically contains address of the current instruction (intel %rip holds the address of the next instruction)
- A call instruction has a one-byte opcode followed by either
  - An 8-byte address or
  - A 4-byte offset
- So… either 9 or 5 bytes after the call opcode is the address of the next instruction.
- That is the return address to push ☺

Calling Functions

- We've discussed how to separate space on the stack for each function by using **stack frames**
- If main(), or some other function, fills many (all?) of the 16 integer registers with valid data, then calls another function, what happens to that data? What registers can the called function use to perform it's work?

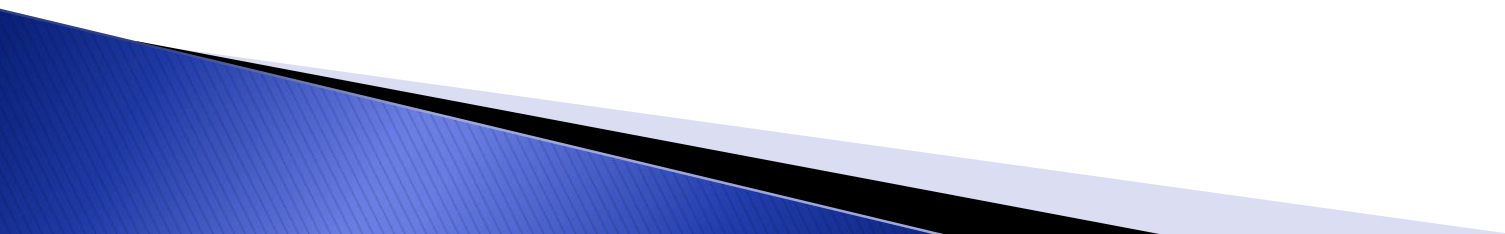
What to do? What to do? ☺

Options:

1. The function that is performing the call has to save ***every, single*** register it's using to the stack prior to making the call, then pop them back into the appropriate registers upon return.
2. The function that is called has to save ***every, single*** register it plans to use to the stack prior to doing any "real" work, then pop the values back into the correct registers before returning to the calling function.

Both seem a little harsh! Can't we both just get along???

How about a little cooperation?



X86 Register Usage Conventions

- Although only one procedure can be active at a given time, the 16 registers are “shared” by all procedures.
- Therefore, we need a way to ensure that when one procedure (the caller) calls another (the callee), values that the caller needs after return will not be overwritten.
- To ensure this, conventions have been adopted as to which procedure, the caller or callee, is responsible for preserving a given register (other than %rsp).
- We will be adopting register save conventions used by X86-64 in a C programming environment.

X86 Register Usage Conventions

- **Caller Saved Registers:** registers that the "Caller" function pushes to the stack prior to calling the "Callee" function, IF the register contains data needed by the "Caller" after the "Callee" function returns. Caller must pop them from the stack after Callee returns.

- **Callee Saved Registers**: registers that the "Callee" function must push to the stack if the "Callee" function wishes to use the register. "Callee" **must** assume there is data in each of these registers that is important to the "Caller" function. Callee function must pop these register back prior to returning to the "Caller".

- The first 6 parameters are passed from the caller to the callee in registers %rdi, %rsi, %rdx, %rcx, %r8 and %r9, respectively.

- If the callee returns a value to the caller, it is returned in register %rax.
- Confused? Check out Figure 3.2, p 180 of Bryant/O'Halloran

Register Conventions

%rax	Return value
%rbx	Callee Saved
%rcx	4th parameter
%rdx	3rd parameter
%rsi	2nd parameter
%rdi	1st parameter
%rsp	Stack Pointer
%rbp	Callee Saved

%r8	5th parameter
%r9	6th parameter
%r10	Caller Saved
%r11	Caller Saved
%r12	Callee Saved
%r13	Callee Saved
%r14	Callee Saved
%r15	Callee Saved

What did all that register stuff really mean?

Register Allocation

- What can be used?
- When do you save?

- Linux uses what is call System V ABI to define this.

System V ABI – 64 bit processors

- What is System V ABI? – a “bible” of sorts with respect to how to interface to C standard libraries when not using C code….X86-64, for example.
- Here is the link to a reasonable “draft” copy from 2013:

https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

A bug seems to be that if you just try to click on this link, it doesn’t work, but if you copy/paste the link in a browser window it comes up just fine.

There are many interface standards within the ABI, register usage and caller/callee parameters a just a couple…

System V ABI says

- This first six integer or pointer arguments are passed in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9 – ***in this order.***
- %rax is used for return values
- %rsp must be restored when control is returned to the caller function
- If the callee wishes to use registers %rbx, %rbp or %r12-%r15, it must save/then restore their original values before returning control to the caller.
- All other registers must be saved by the caller if it wishes to preserve their values

So which registers do we use?

- It depends!
  - On what our function is passed
  - On what registers our function wants to use
  - On what other functions our function might call
  - How we can minimize save/restore activity
    - Efficiency is why we got here in the first place, remember?

All functions

- Have to save and restore any of these registers it plans to use:
  - %rbp: probably using as part of the stack frame and planned to restore it anyway
  - %rsp: if we don't restore it, the program will probably crash. Assume all functions deal with %rsp correctly.
  - %rbx and %r12-%r15: Have to save these before we use them

Leaf Functions

- Leaf functions make no calls to any other function
- Can freely use %rdi, %rsi, %rdx, %rcx, %r8 and %r9 even when passed fewer than 6 parameters.
- Can freely use %r10 and %r11 since these are caller-saved registers
- Can freely use %rax as long as function fills it with the return value prior to returning to the caller.

Functions that call other functions

- Trade-offs to be made:
  - If we make many calls, we have to save and restore any of the parameter registers as well as %r10/%r11 before and after **each** call if we still want to use the values they had prior to the call.
  - If we use %rbx, %rbp, %r12-%r15, we only have to save them one time (at the beginning) and restore them one time (at the end).

Recursive procedure calls

- Because the stack frame for the procedure is set up at the beginning of its code, a procedure which calls itself recursively will get **a new stack frame** each time it is called.
- The stack frame for the second call of the procedure will be above the stack frame for the first (i.e., closer to the top of the stack, but at a lower-numbered address), and so on.
- When each call returns, the frame pointer of the previous call will be restored, and at that point, what is at the top of the stack will be the return address from the previous call.
- Therefore, when the ret instruction is executed at the end of the recursive function's assembly code, execution will return to the point in the code of the function from which the call was made.
- So, just how deep of a recursive procedure do you want to have in your code given all of the resources each call is going to use? Hmmm?

Linux/Unix C-Library function calls

- We can call any C-Library function that we used in our C-language programs from any x86-64 program that we write
- Parameters must be passed using the caller/callee/return value paradigm described above
- Assume that all caller saved registers will be trashed after return and plan accordingly ☺

System V ABI – 64 bit processors

- From section 3.5.7 Variable Argument Lists:

  Some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that all arguments are passed on the stack, and arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many implementations. However, they do not work on the AMD64 architecture because some arguments are passed in registers. Portable C programs must use the header file in order to handle variable argument lists. **When a function taking variable-arguments is called, %rax must be set to the total number of floating point parameters passed to the function in vector registers.**

- Since we won't be passing any floating point parameters (we're only using integers), we will **always** have to set %rax to zero before calling a function that allows a variable argument list.

*the section above references AMD64 architecture, but x86-64 is equivalent

Variable Argument Lists

- What functions did we use in C that had variable argument lists?
  - printf() family
  - scanf() family
- You have make a point to set %rax to zero prior to calling printf() or scanf(), because if you do not, expect your program to seg fault.
- No only that, but **fully expect** the information in all “caller saved registers” to be **totally trashed** upon return

Section 4: A Program Example

Simple C program example

- Consider the following simple C program, with two functions. It illustrates the X86 conventions for parameter passing, return value, and use of caller and callee save registers.
- First, main:

```
long sum(long count, long *array);
int main() {
    static long array[4] = {10, 12, 15, 19};
    long count= 4;                  /* number of array elements  */
    long result;
    result = sum(count, array);
    printf("The sum of the array is %i\n", result);
}
```

Function sum()

- Now, sum():

```
long sum(long count, long *array) {
    long result = 0;
     long i;
    for (i = 0; i < count; i++) {
        result = result + array[i];
    }
    return(result);
}
```

Now, the assembly language . . .

The next slide shows X86 assembler directives to set up space in memory for:

1. the static array,
2. output, and
3. the stack

Now, the assembly language . . .

```
.file “sumprog.s”
# Assembler directives to allocate storage for static array
.section
.rodata
printf_line:
.string “The sum of the array is %i\n”
.data
.align 8            # insure that we are starting on an 8-byte boundary
array:              # this is a LABEL
    .quad 10
    .quad 12
    .quad 15
    .quad 19
.globl main
        .type main, @function
```

Now, main()

```
.text
main:
    pushq %rbp              # save caller's %rbp
    movq  %rsp, %rbp                # copy %rsp to %rbp so our stack frame  is ready to use

    movq  $array, %rsi              # set %rsi (2nd parameter) to point to start of array
    movq $4, %rdi                   # set %rdi (1st parameter) to count = 4
                            # (i.e. caller saved registers) since we aren't using %rsi or %rdi
                            # values or the value in any other caller saved registers,
                            # we don't have to push them
    call sum

    movq %rax, %rsi                 # Write return value to 2nd parameter
    movq $printf_line, %rdi         # Write string literal to 1st parameter

    movq $0, %rax
    call printf
    leave
    ret
.size main, .-main
```

Finally, sum()

```
.globl sum
        .type sum, @function
sum:
   pushq    %rbp            #save caller's rbp
   movq %rsp, %rbp          #set function's frame pointer
                            # register %rdi contains count (1st parameter)
                            # register %rsi contains address to array (2nd parameter)
   movq $0, %rax            # initialize sum to 0, by putting 0 in %rax,
                            # it's where return value
                            # needs to be when we return
loop:                       # loop to sum values in array
   decq   %rdi              # decrement number of remaining elements by 1
   jl exit                          # jump out of loop if no elements remaining
   addq (%rsi,%rdi,8), %rax # add element to sum
jmp     loop                # jump to top of loop
exit:                       # sum already in register %rax so ready to return
   leave
   ret                      #return to caller's code at return address
.size sum, .-sum
```

Finally, sum() (modified)

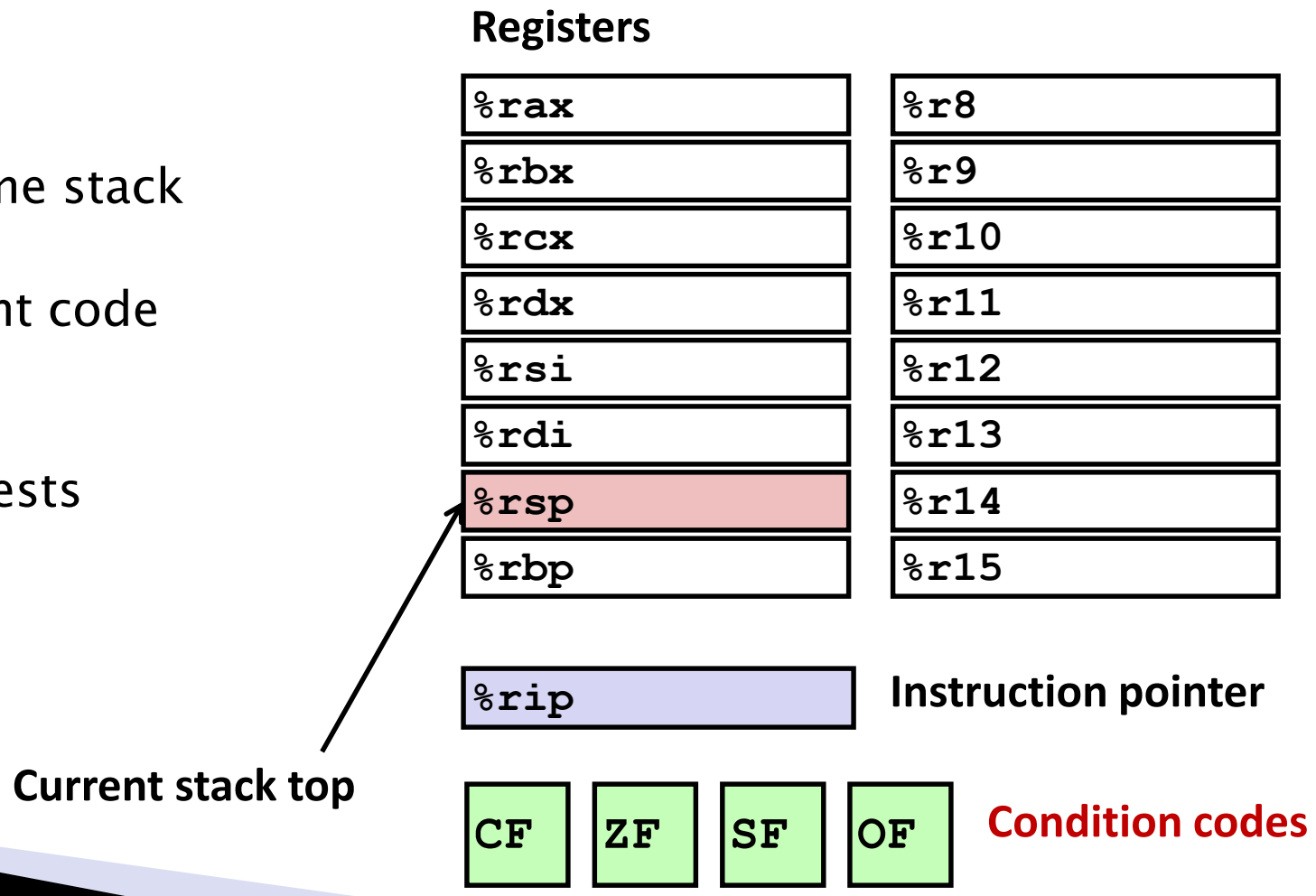
```
sum:
    pushq    %rbp                  #save caller's rbp
    movq %rsp, %rbp                #set function's frame pointer
                                   # register %rdi contains count (1st parameter)
                                   # register %rsi contains address to array (2nd parameter)
    movq $0, %rax                  # initialize sum to 0, by putting 0 in %rax,
                                   # it's where return value
                                   # needs to be when we return
    pushq %rax
    pushq %rdi                     #so what happens if I decide to add a printf call
    pushq %rsi                     # in the middle of this code?
    movq $printf_literal1, %rdi    # It changes a fundamental assumption about registers.
    movq $0, %rax                  # There is a better way to do this.
    call printf                    # the performance hit of this code if it was in the loop
    popq %rsi                      # would be bad – and there is a better way to do it.
    popq %rdi                      # what is the better way?
    popq %rax
loop:                              # loop to sum values in array
    decq   %rdi                    # decrement number of remaining elements by 1
    jl exit                        # jump out of loop if no elements remaining
    addq (%rsi,%rdi,8), %rax       # add element to sum
jmp     loop                       # jump to top of loop
exit:                              # sum already in register %rax so ready to return
    leave
    ret                            #return to caller's code at return address
.size sum, .-sum
```

Today

- Control: Condition codes
- Conditional branches
- Loops

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



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 – AGU doesn't set codes
- Not set by mov – the data doesn't go through the ALU

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)
```

AT&T Syntax, Compare, & Conditionals

- AT&T Syntax is “backwards” to intel syntax
- This is confusing when using the compare instruction and conditional operations:

```
                                # %rdi holds x, %rsi holds y
                                # jump if x > y
    cmpq        %rsi, %rdi      # compare x and y
    jg          some_label          # jump when %rdi > %rsi
```

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 for:
    - repeating the operand to determine if value is negative, zero or positive (e.g. testq %rax %rax)
    - to have one of the operands be a mask to test individual bits (e.g. testq %rax, 0x0100)
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Reading Condition Codes

- SetX Instructions (Figure 3.14 in Bryant/O'Hallaron)
  - Set low-order byte of destination(low order single-byte register or a single byte memory location) 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)

Why? So that you can store a condition longer than one instruction

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbq`** to finish job
    - (Figure 3.5 & last 4 paragraphs of 3.4.2)

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
    cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when >
    movzbq  %al, %rax     # Zero rest of %rax
    ret
```

In a nutshell

- Aritmetic
  - cmp <src1> <src2>
    - Computes <src2> – <src1>, but does not save result anywhere
    - Condition codes are set based on the computation
    - src1, and src2 must be of the same size
  - cmpb, cmpw, cmpl or cmpq
- Logical
  - Test <src1> <src2>
    - Computes <src2> & <src1>, but does not save result anywhere
    - Condition codes are set based on the computation
    - src1, and src2 must be of the same size
  - testb, testw, testl, testq

Conditional Moves

- cmovX Instructions
  - Move a value (or not) depending on condition codes

cmovX	Condition	Description
`cmove`	`ZF`	Equal / Zero
`cmovne`	`~ZF`	Not Equal / Not Zero
`cmovs`	`SF`	Negative
`cmovns`	`~SF`	Nonnegative
`cmovg`	`~(SF^OF)&~ZF`	Greater (Signed)
`cmovge`	`~(SF^OF)`	Greater or Equal (Signed)
`cmovl`	`(SF^OF)`	Less (Signed)
`cmovle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`cmova`	`~CF&~ZF`	Above (unsigned)
`cmovb`	`CF`	Below (unsigned)

Appendix: Looking at a simple C Program

- We'll see this again as our first assembler programming homework

Simple C program

- The simple C program below will be translated to assembly language in the following slides:

```
#include <stdio.h>

long x;            /* file scope variable – stored on the heap */

int main () {
    printf("Please enter an integer on the next line, followed by enter:\n“);
    scanf("%i", &x);     /* Get a value from the user */
    x = x + 5;           /* add 5 to the input value */
    printf("The value of x after adding 5 is: %i\n", x);
    return 0;
}
```

x86-64 program

```
.file	"scanPrint.s“		#optional directive
.section		.rodata			#required directives for rodata
.LC0:
		.string	"Please enter an integer on the next line, followed by enter:\n"
.LC1:
		.string	"%i"
.LC2:
		.string	"The value of x after adding 5 is: %i\n"

.data				#required for file scope data: read-write program data
		#of static storage class
x:
		.quad 0

.globl main			#required directive for every function
		.type	main, @function		#required directive
```

Code for main

```
.text                                   #required directive
main:
        pushq   %rbp                    #stack housekeeping #1
        movq    %rsp, %rbp                      #stack housekeeping #2
        movq    $.LC0, %rdi                     #address of string “Please enter…:\n“ to %rdi
                                        # %rdi is location of 1st parameter not pushing any caller saved
                                        # registers because there is no valuable data there
        movq    $0, %rax                # C library ABI says %rax should be zero b4 call to printf
        call    printf
        movq    $x, %rsi                #mov the address of x to %rsi (2nd parameter)
        movq    $.LC1, %rdi                     #address of string “%i” in %rdi (1st parameter)
        movq    $0, %rax                # to keep ABI happy
        call    scanf
        addq    $5, x                   #add the constant 5 to what is stored in variable x
        movq    x, %rsi                 #value of x to %rsi (2nd parameter)
        movq    $.LC2, %rdi                     #address of string “The value of…” to %rdi (1st param)
        movq    $0, %rax                # keep ABI happy
        call    printf
        movq    $0, %rax                #set return value to 0
        leave
        ret
        .size   main, .-main            #required directive
```