CMPT 295

Unit - Machine-Level Programming

Lecture 20 – Assembly language – Array – 1D

Recursive Function – `countOnesR( … )`

```
/* Recursive counter of 1’s */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

What does this function do?

Recursive Function – Example – Base Case

```
/* Recursive counter of 1's */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl     %eax, %eax
  testq    %rdi, %rdi
  je       done
  pushq    %rbx
  movq     %rdi, %rbx
  andl     $1, %ebx
  shrq     %rdi
  call     countOnesR
  addq     %rbx, %rax
  popq     %rbx
done:
  ret
```

Recursive Function – Example - Saving registers

```
/* Recursive counter of 1’s */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl     %eax, %eax
  testq    %rdi, %rdi
  je       done
  pushq    %rbx
  movq     %rdi, %rbx
  andl     $1, %ebx
  shrq     %rdi
  call     countOnesR
  addq     %rbx, %rax
  popq     %rbx
done:
  ret
```

Recursive Function – Example - Call Setup

```
/* Recursive counter of 1’s */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl    %eax, %eax
  testq   %rdi, %rdi
  je      done
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    countOnesR
  addq    %rbx, %rax
  popq    %rbx
done:
  ret
```

Recursive Function – Example – Recursive Call

```
/* Recursive counter of 1’s */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl    %eax, %eax
  testq   %rdi, %rdi
  je      done
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    countOnesR
  addq    %rbx, %rax
  popq    %rbx
done:
  ret
```

Recursive Function – Example – Result

```
/* Recursive counter of 1's */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl     %eax, %eax
  testq    %rdi, %rdi
  je       done
  pushq    %rbx
  movq     %rdi, %rbx
  andl     $1, %ebx
  shrq     %rdi
  call     countOnesR
  addq     %rbx, %rax
  popq     %rbx
done:
  ret
```

Recursive Function – Example – Clean-up and return

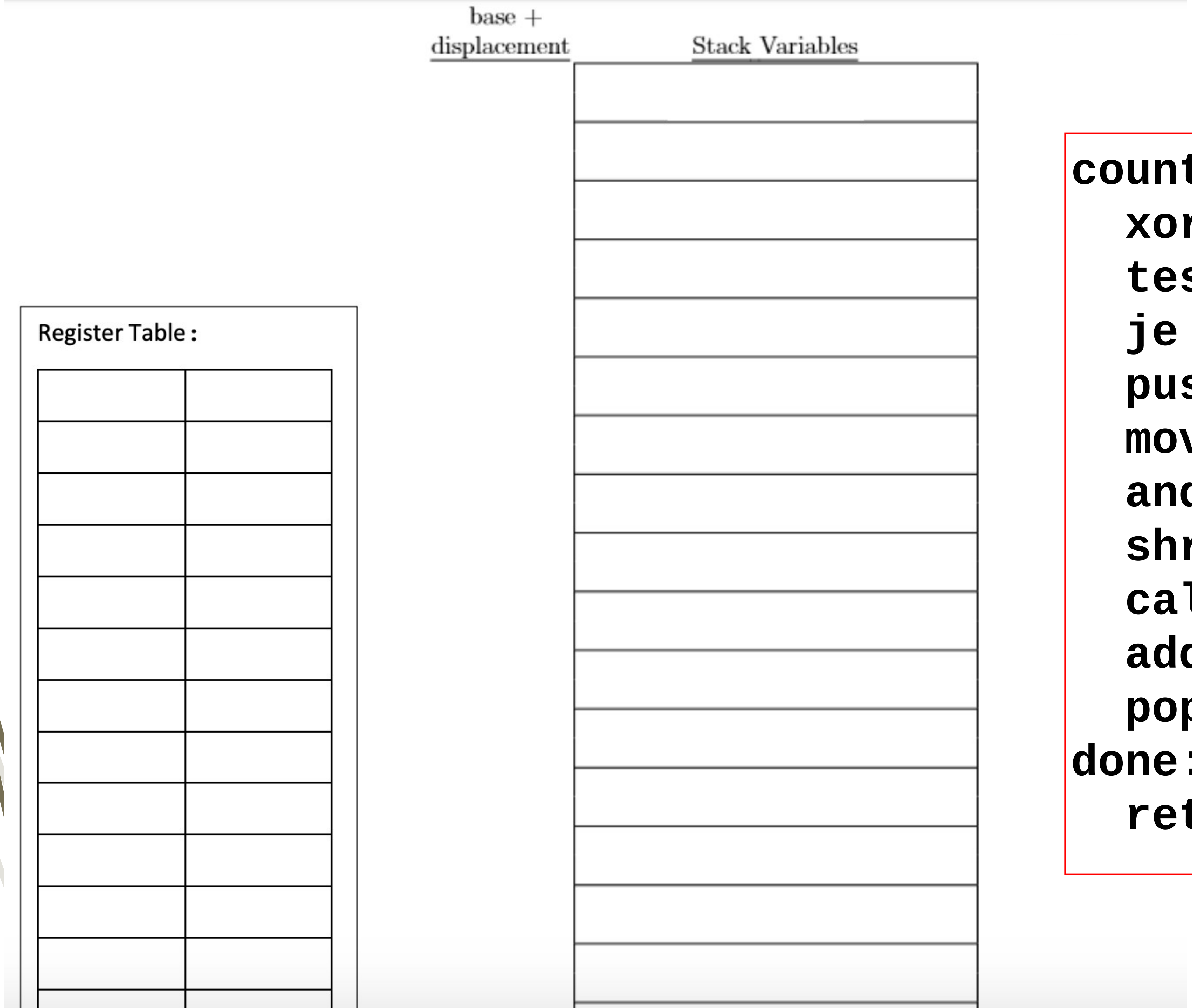
```
/* Recursive counter of 1’s */
long countOnesR(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + countOnesR(x >> 1);
}
```

```
countOnesR:
  xorl    %eax, %eax
  testq   %rdi, %rdi
  je      done
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    countOnesR
  addq    %rbx, %rax
  popq    %rbx
done:
  ret
```

Recursive Function – Example – Test Cases

- Test Case 1
  - Input: x = 0
  - Expected result: 0

- Test Case 2
  - Input: x = 7
  - Expected result: 3



base + displacement

Stack Variables

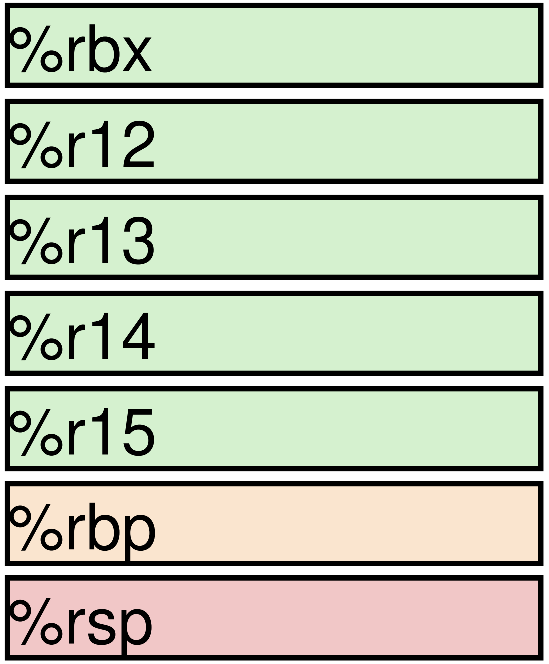
Register Table :

```
countOnesR:
   xorl     %eax, %eax
   testq    %rdi, %rdi
   je       done
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     countOnesR
   addq     %rbx, %rax
   popq     %rbx
done:
   ret
```

Last Lecture - x86-64 “register saving” convention

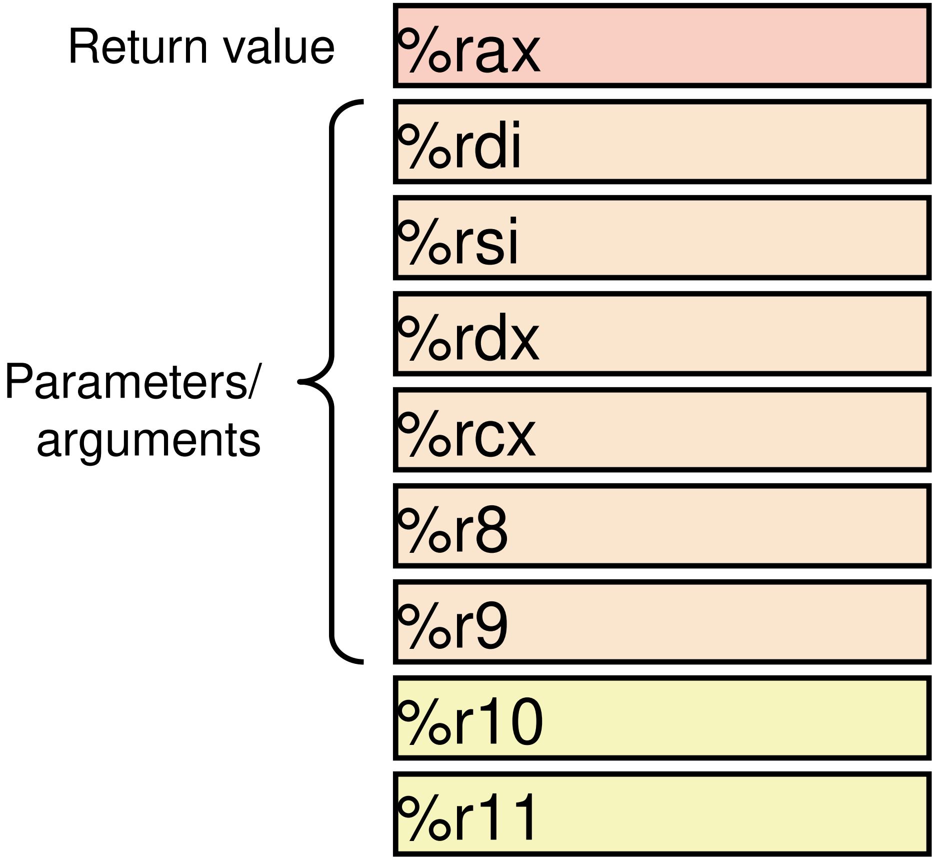
callee saved registers:

- **Callee** must save & restore

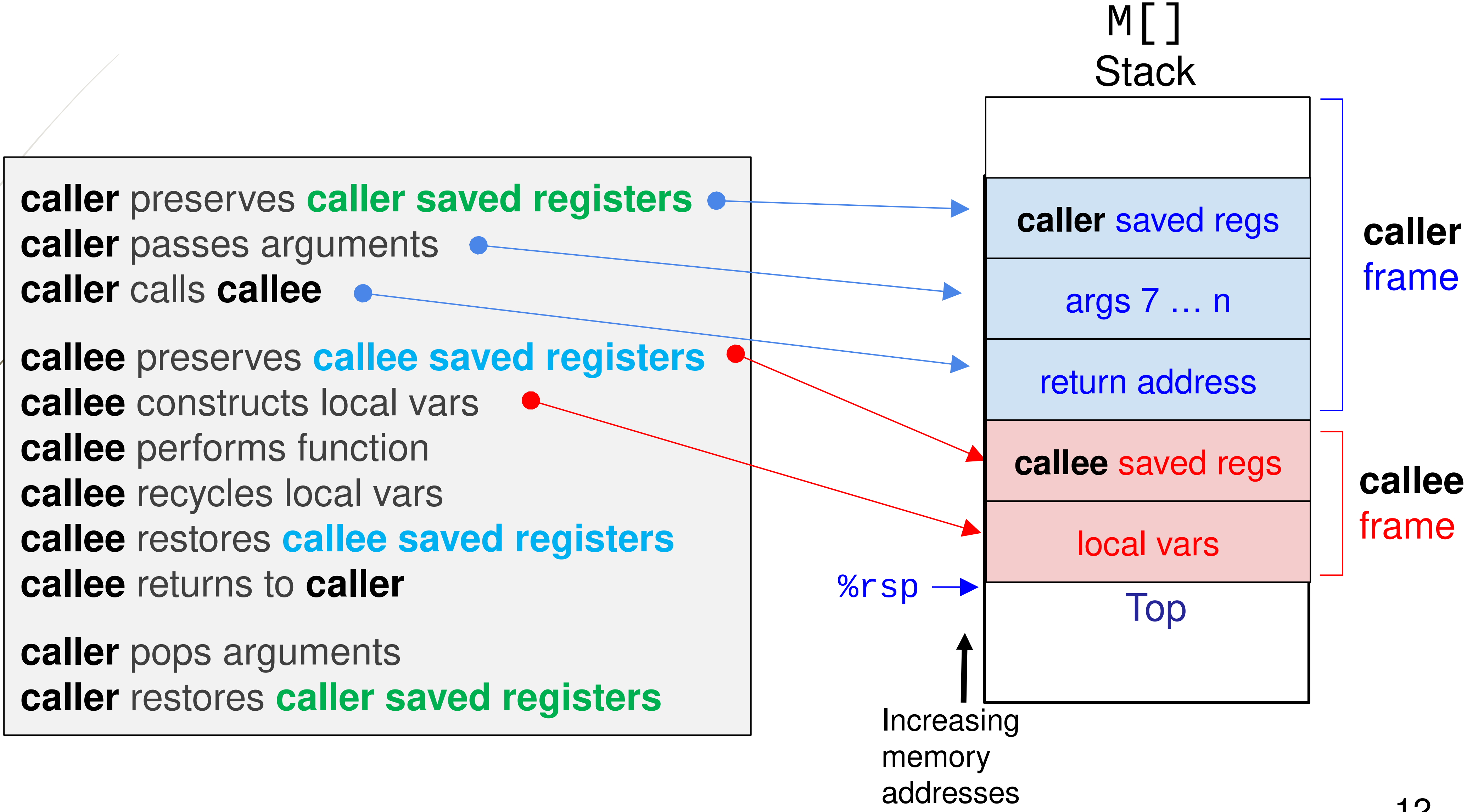


caller saved registers:

- **Caller** must save & restore
- Can be modified by **callee**



Last Lecture - x86-64 conventions and stack frame



caller preserves **caller saved registers**
caller passes arguments
caller calls **callee**

callee preserves **callee saved registers**
callee constructs local vars
callee performs function
callee recycles local vars
callee restores **callee saved registers**
callee returns to **caller**

caller pops arguments
caller restores **caller saved registers**

Last Lecture

- Recursion
  - Handled without special consideration using …
    - Stack frames
    - x86-64 Function call and Register saving conventions

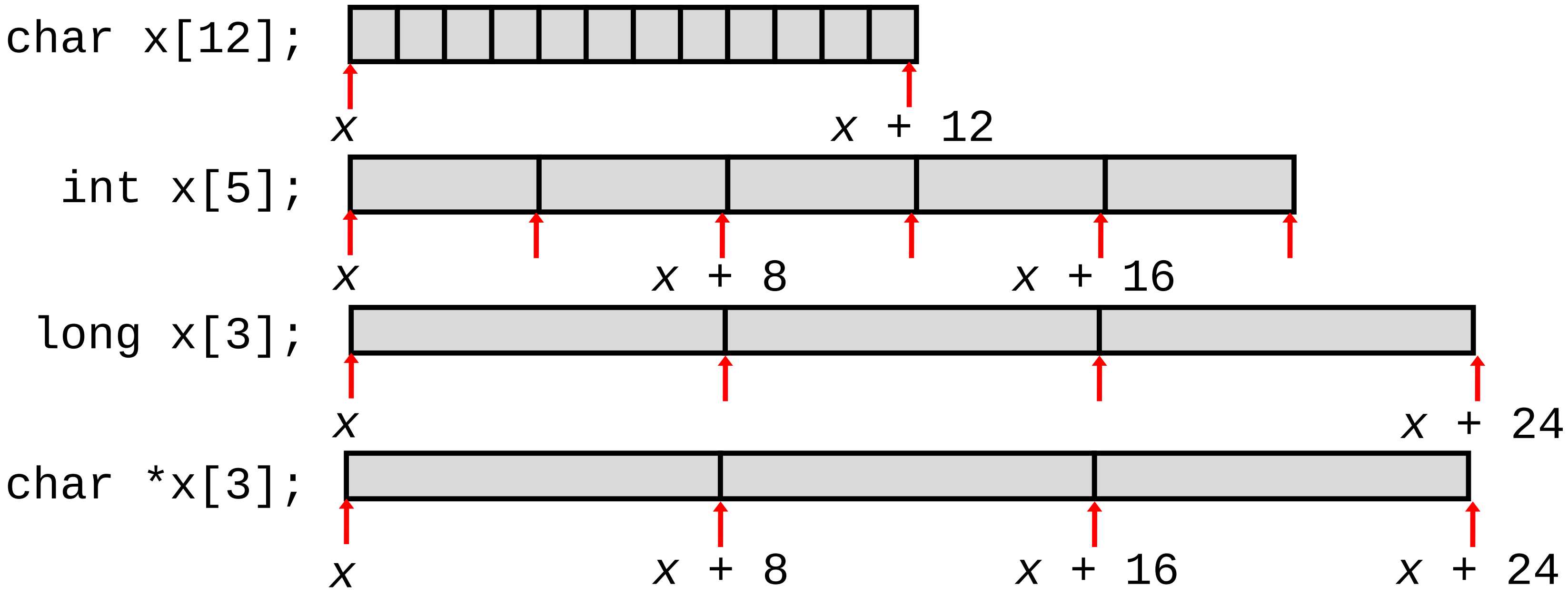
Today’s Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations

1D Array

T **A[*n*];**

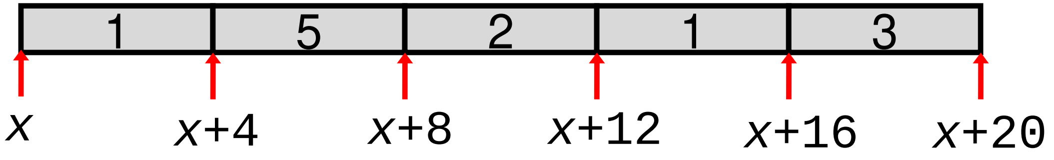
- Array of data type T and length n
- Contiguously allocated region of n * L bytes in memory where L = **sizeof**(*T*)



Accessing 1D Array

- Address of **A**`[i]` = base address + `i` * *L*
- **A** can be used as a pointer to array element 0
- Can increment a pointer **A** by adding *L* to the address

`int x[5];`



C	**Type**	**Value**
`x[4]`	`int`	
`x`	`int *`	
`x + 1`	`int *`	
`&x[2]`	`int *`	
`x[5]`	`int`	
`*(x+1)`	`int`	
`x + i`	`int *`	

Manipulating 1D array – Example - `main.c`

This program defines 4 arrays:

- an array of char’s,
- an array of short’s,
- an array of int’s,
- an array of long’s

then it calls the appropriate `sum` function, i.e., the one that sums elements of the correct data type.

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

char sumChar(char *, int);
short sumShort(short *, int);
int sumInt(int *, int);
long sumLong(long *, int);

#define N 6
```

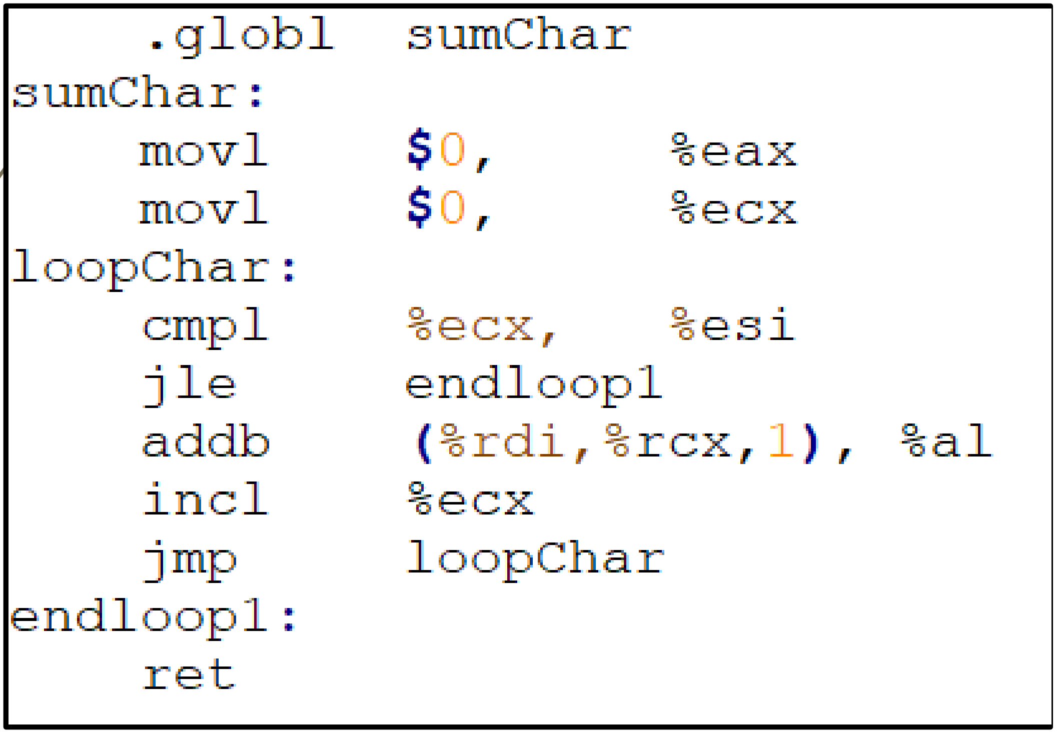
Test cases

Expected results

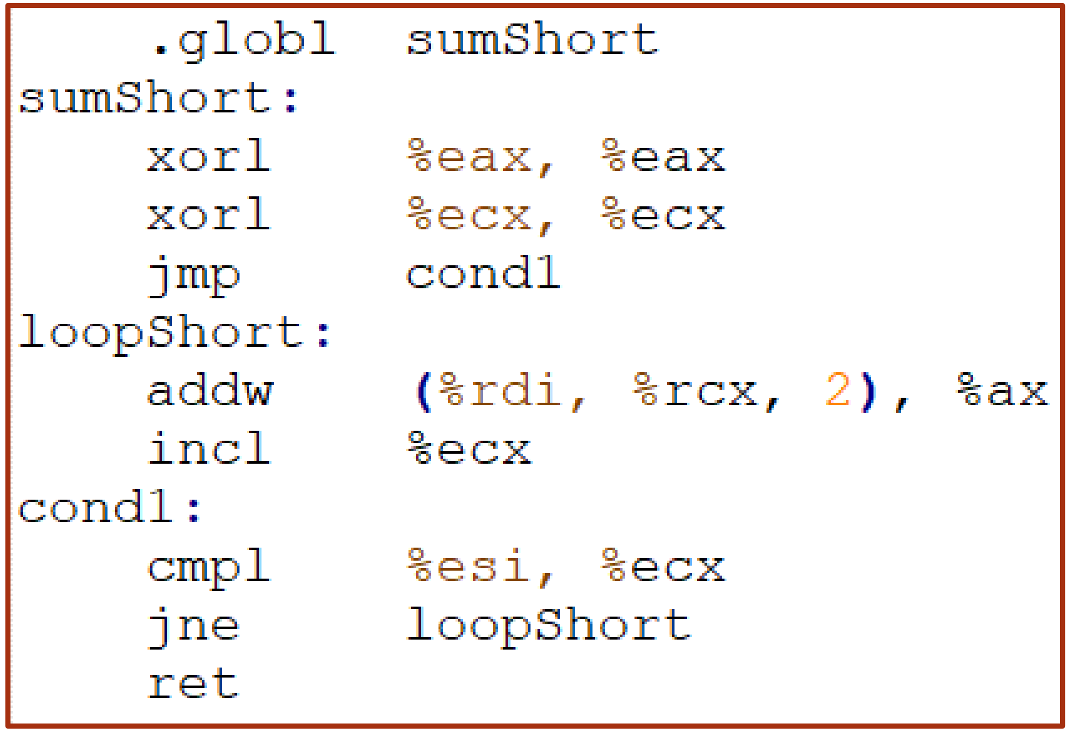
```c
char AC[N] = {-58, 22, 101, -15, 72, 27};             // Expected results: sum = -107
short AS[N] = {-58, 22, 101, -15, 72, 27};            // Expected results: sum = 149
int AI[N] = {258, 522, 1010, -15, -3372, 27};         // Expected results: sum = -1570
long AL[N] = {258, 522, 1010, -15, -3372, 27};        // Expected results: sum = -1570


void main () {

    printf("The total of AC is %hhi.\n", sumChar(AC, N));
    printf("The total of AS is %hi.\n", sumShort(AS, N));
    printf("The total of AI is %d.\n", sumInt(AI, N));
    printf("The total of AL is %ld.\n", sumLong(AL, N));
    return;
}
```

Manipulating 1D array – Example - `sums.s` - Part 1

- Register %rdi contains starting address of array (base address of array)
- Register %esi contains size of array (N)
- Register %ecx contains array index
- Register %al or %ax contains the running sum



```
    .globl   sumChar
sumChar:
    movl     $0,      %eax
    movl     $0,      %ecx
loopChar:
    cmpl     %ecx,    %esi
    jle      endloop1
    addb     (%rdi,%rcx,1), %al
    incl     %ecx
    jmp      loopChar
endloop1:
    ret
```



```
    .globl   sumShort
sumShort:
    xorl     %eax, %eax
    xorl     %ecx, %ecx
    jmp      cond1
loopShort:
    addw     (%rdi, %rcx, 2), %ax
    incl     %ecx
cond1:
    cmpl     %esi, %ecx
    jne      loopShort
    ret
```

Manipulating 1D array – Example - `sums.s` - Part 2

- Register %rdi contains starting address of array (base address of array)
- Register %esi contains size of array (N)
- Register %ecx contains array index
- Register %eax or %rax contains the running sum

```
    .globl  sumInt
sumInt:
    xorl    %eax, %eax
    xorl    %ecx, %ecx
    jmp     cond2
loopInt:
    addl    (%rdi, %rcx, 4), %eax
    incl    %ecx
cond2:
    cmpl    %esi, %ecx
    jne     loopInt
    ret
```

```
    .globl  sumLong
sumLong:
    movq    $0,     %rax
    movl    $0,     %ecx
loopLong:
    cmpl    %ecx,   %esi
    jle     endloop
    addq    (%rdi,%rcx,8), %rax
    incl    %ecx
    jmp     loopLong
endloop:
    ret
```

Warning!

- Forgetting that the memory addresses of contiguous array cells differ by the size of the content of these cells instead of by 1 is a common mistake in assembly language programming

Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
  - 1D array: address of **A**`[i]` = **A** + `i` * L

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
  - Recursion
- Array
- Buffer Overflow
- Floating-point operations