CMPT 295

Unit - Machine-Level Programming

Lecture 18 – Assembly language – Program Control – Function Call and Stack - Passing Data

Last Lecture

- Function call mechanisms: 1) passing control, 2) passing data, 3) managing local data on memory (stack)
- Memory layout
  - Stack (local variables …)
  - Heap (dynamically allocated data)
  - Data (statically allocated data)
  - Text / Shared Libraries (program code)
- A "stack" is the right data structure for function call / return
  - If `multstore` calls `mult2`, then `mult2` returns before `multstore` returns
- x86-64 stack register and instructions: stack pointer **`%rsp`**, **`push`** and **`pop`**
- x86-64 function call instructions: **`call`** and **`ret`**

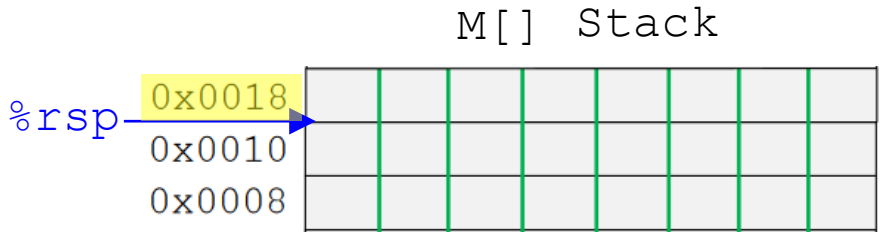
From Lecture 17 – Slide 14

Why 8?

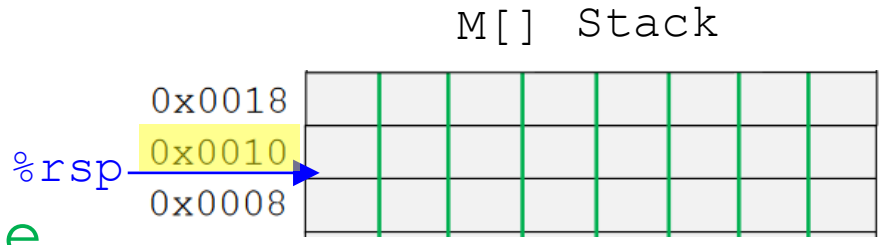
- **`push* Src`**
  1. Get value of operand **`Src`**
  2. Decrement **`%rsp`** by **8**
  3. Store value at **`%rsp`**
- **`pop* Dest`**
  1. Read value at **`%rsp`** and load this value in operand **`Dest`**
  2. Increment **`%rsp`** by **8**

Remember: ***Compressed view of memory***

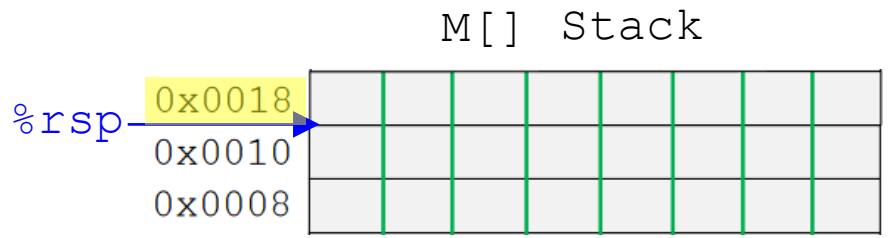
1. `%rsp` contains the memory address `0x0018`



2. `%rsp` contains the memory address `0x0010`



3. `%rsp` contains the memory address `0x0018`



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

2. Passing data mechanism – using stack x86-64 function call convention

1. **Caller** and **callee** functions must obey ***function call convention*** when passing data during function call
   - **Caller**:
     - Before calling the **callee** function, the **caller** must copy the **callee**'s arguments (1 to 6) into specific registers:

If there is a …

     - 1st argument -> **`%rdi`** (or `%edi`, or `%di` or `%dil`)
     - 2nd argument -> **`%rsi`** (or `%esi`, or `%si` or `%sil`)
     - 3rd argument -> **`%rdx`** (or `%edx`, or `%dx` or `%dl`)
     - 4th argument -> **`%rcx`** (or `%ecx`, or `%cx` or `%cl`)
     - 5th argument -> **`%r8`** (or `%r8d`, or `%r8w` or `%r8b`)
     - 6th argument -> **`%r9`** (or `%r9d`, or `%r9w` or `%r9b`)
   - **Callee**:
     - Before returning to **caller**, **callee** must copy returned value into register **`%rax`**

Passing data mechanism – Example of passing arguments in registers and returning return value

```c
long plus(long x, long y) {
    return x + y;
}

void sum_store(long x, long y, long *dest)
{
    long sum = plus(x, y);
    *dest = sum;
}

int main(int argc, char *argv[]) {
  if ( argc == 3 ) {
    long x = atoi(argv[1]);
    long y = atoi(argv[2]);
    long result;
    sum_store(x, y, &result);
    printf("%ld + %ld --> %ld\n", x, y, result);
  }
  else printf("2 numbers required!\n");
  return 0;
}
```

```
./ss 5 6
```

```
sum_store:
.LFB40:
    .cfi_startproc
    endbr64
    addq    %rsi, %rdi
    movq    %rdi, (%rdx)
    ret
```

```
main:
    pushq   %r13
    pushq   %r12
    pushq   %rbx
    subq    $16, %rsp
    movq    %fs:40, %rax
    movq    %rax, 8(%rsp)
    xorl    %eax, %eax
    cmpl    $3, %edi
    je  .L7
    leaq    .LC1(%rip), %rdi
    call    puts@PLT
.L3:
    movq    8(%rsp), %rax
    xorq    %fs:40, %rax
    jne .L8
    addq    $16, %rsp
    xorl    %eax, %eax
    popq    %rbx
    popq    %r12
    popq    %r13
    ret
```

```
.L7:
    movq    8(%rsi), %rdi
    movq    %rsi, %rbx
    movl    $10, %edx
    xorl    %esi, %esi
    call    strtol@PLT
    movq    16(%rbx), %rdi
    xorl    %esi, %esi
    movl    $10, %edx
    movslq  %eax, %r12
    call    strtol@PLT
    movq    %rsp, %rdx
    movq    %r12, %rdi
    movslq  %eax, %r13
    movq    %r13, %rsi
    call    sum_store@PLT
    movq    (%rsp), %r8
    movq    %r13, %rcx
    movq    %r12, %rdx
    leaq    .LC0(%rip), %rsi
    movl    $1, %edi
    xorl    %eax, %eax
    call    __printf_chk@PLT
    jmp .L3
.L8:
    call    __stack_chk_fail@PLT
```

What if the **callee** function has more than 6 arguments?

1. **Caller** and **callee** functions must obey ***function call convention*** when passing data during function call
   - **Caller**:
     - Before calling the **callee** function, the **caller** must copy the **callee**'s arguments (1 to 6) into specific registers: …
     - Then must push the rest of the arguments on the stack in reverse order
   - **Callee**:
     - Before returning to **caller**, **callee** must copy returned value into register `%rax`

If a callee function has more than 6 arguments …

2. Passing data mechanism – using stack x86-64 function call convention

2. When passing data that is a memory address (i.e., a pointer) during function call

- **Caller**:
  - Must make use of the stack in order to create such memory address

Passing data mechanism – Examples of local variables, arguments and pointers on the stack

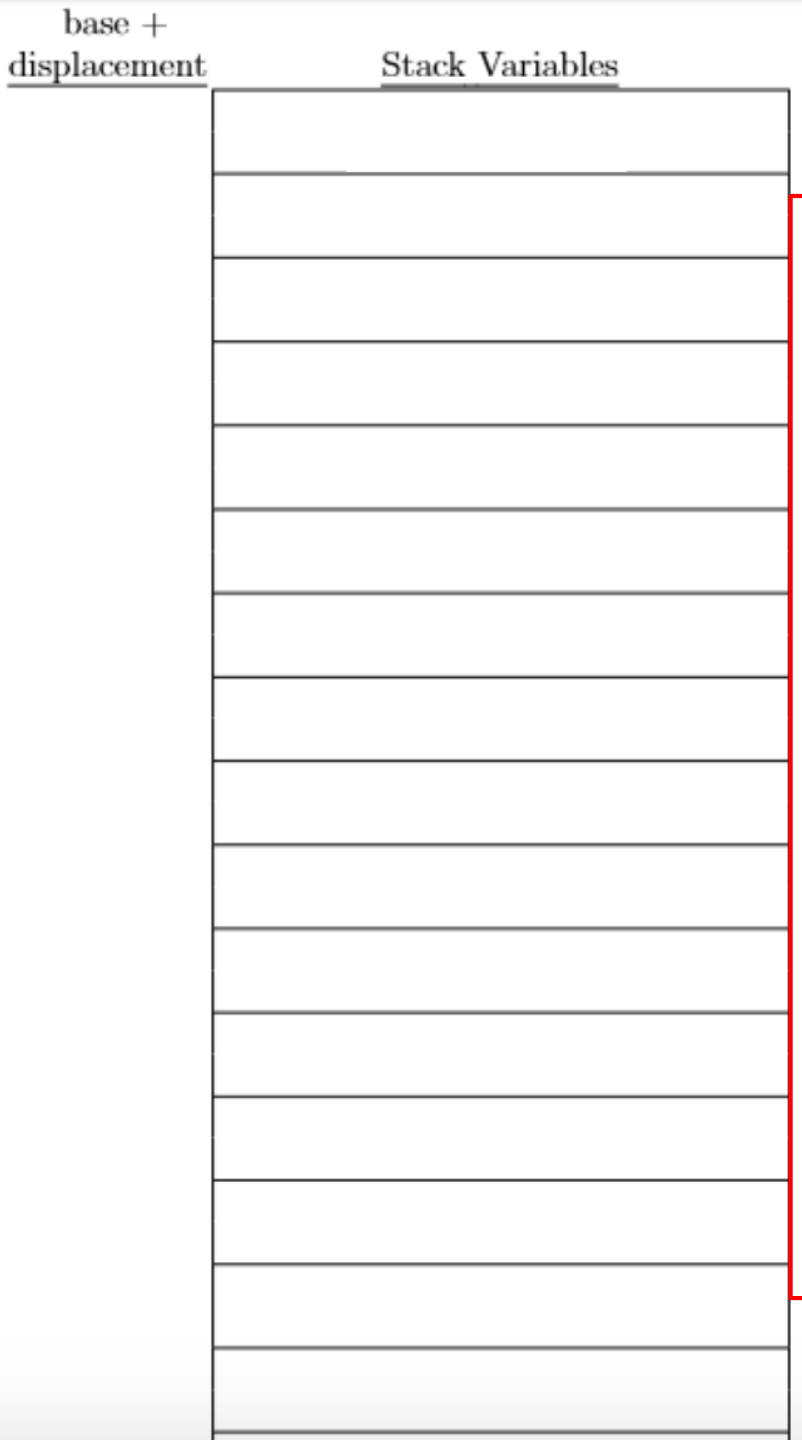
```
long call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);

    return (x1+x2)*(x3-x4);
}
```

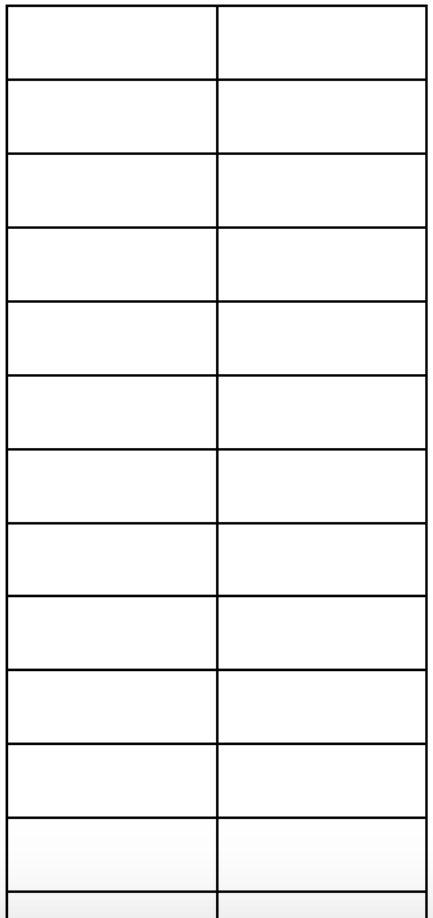
How to push x4 and &x4 onto stack?

```
call_proc:
    subq   $40, %rsp
    movq   $1, 32(%rsp)
    movl   $2, 28(%rsp)
    movw   $3, 26(%rsp)
    movb   $4, 25(%rsp)
    movq   32(%rsp), %rdi
    movl   28(%rsp), %edx
    leaq   25(%rsp), %rax
    movq   %rax, 8(%rsp)
    movl   $4, (%rsp)
    leaq   32(%rsp), %rsi
    leaq   28(%rsp), %rcx
    leaq   26(%rsp), %r9
    movl   $3, %r8d
    callq proc
    ...
```

local variables



Register Table :

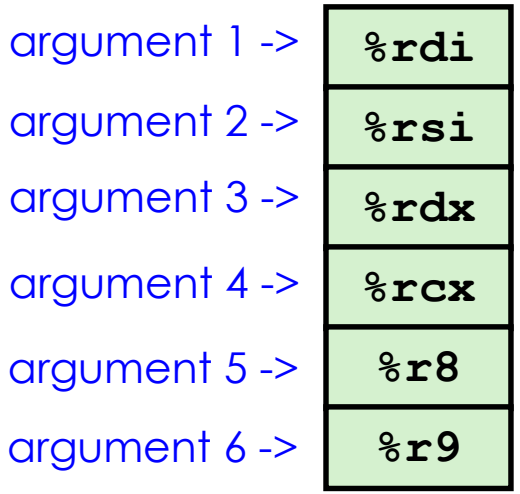


```
call_proc:
    subq   $40, %rsp
    movq   $1, 32(%rsp)
    movl   $2, 28(%rsp)
    movw   $3, 26(%rsp)
    movb   $4, 25(%rsp)
    movq   32(%rsp), %rdi
    movl   28(%rsp), %edx
    leaq   25(%rsp), %rax
    movq   %rax, 8(%rsp)
    movl   $4, (%rsp)
    leaq   32(%rsp), %rsi
    leaq   28(%rsp), %rcx
    leaq   26(%rsp), %r9
    movl   $3, %r8d
    callq proc
    ...
```

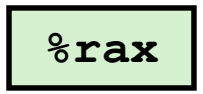
Summary

- Passing data mechanism
  - x86-64 function call convention:

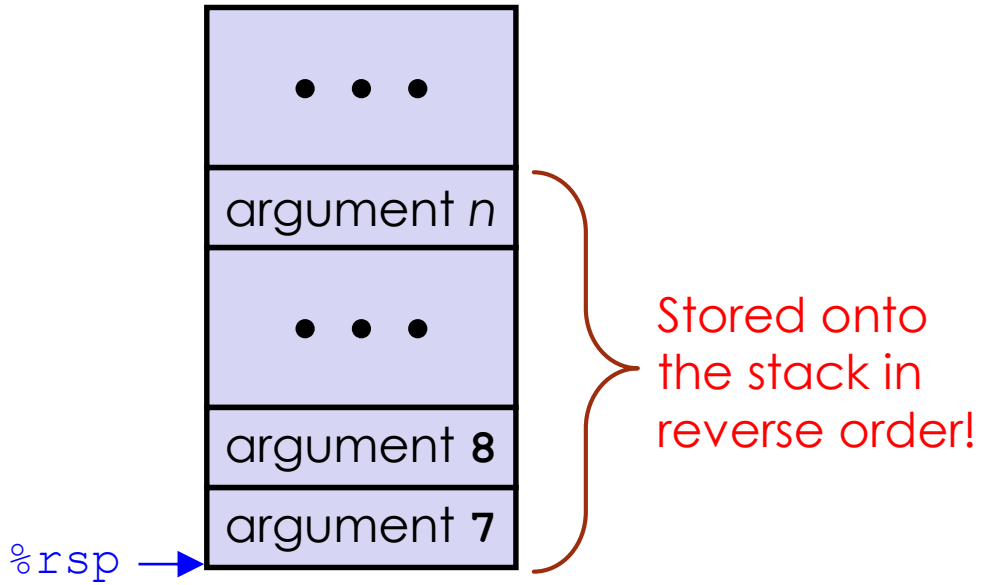
First 6 arguments



return value



Stack



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