CMPT 295

Unit - Machine-Level Programming

Lecture 14 – Assembly language – Program Control – Function Call and Stack - Passing Control

Demo: alternative way of implementing `if/else` in assembly language

- Lecture 12 – `ifelse.c` and `ifelse.s`

Last Lecture

- In x86-64 assembly, there are no iterative statements
- To alter the execution flow, compiler generates code sequence that implements these iterative statements (`while`, `do-while` and `for` loops) using branching method:
  - **`cmp*`** instruction
  - **`jX`** instructions (jump)
- 2 loop patterns:
  - "*coding the false condition first*" -> `while` loops (hence `for` loops)
  - *"jump-in-middle"* -> `while`, `do-while` (hence `for` loops)

While loop – Question from last lecture
"coding the false condition first"

in C:

```
while (x < y) {
   // stmts
}
```

in assembly: # x in %edi, y in %esi

```
loop:
     cmpl %edi, %esi
     jl endloop
     # stmts
     jmp loop
endloop:
     ret
```

Loop Pattern 1

```
loop:
  if cond false
    goto done:
  stmts
  goto loop:
done:
```

Would this assembly code be the equivalent of our C code?

For loop - Homework

In C:

```
        initialization        increment
for (i = 0; i < n; i++){
              condition
  // stmts     testing
}
return;
```

```
i = 0; // initialization
while (i < n) { //  condition
   // stmts          testing
   i++; // increment
}
return;
```

In Assembly:

```
   xorl %ecx, %ecx   # initialization
loop:                # %ecx (i) <- 0
   cmpl %edi, %ecx   # i-n ? 0 testing
   jge endloop       # i-n >= 0
                     #   false condition
   # stmts
   incl %ecx         # i++ increment
   jmp loop          # loop again
endloop:
    ret
```

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

What happens when a function (*caller*) calls another function (*callee*)?

1. Control is passed (PC is set) …
   - To the beginning of the code in *callee* function
   - Back to where *callee* function was called in *caller* function
2. Data is passed …
   - To *callee* function via function parameter(s)
   - Back to *caller* function via *return value*
3. Memory is …
   - Allocated during *callee* function execution
   - Deallocated upon return to *caller* function

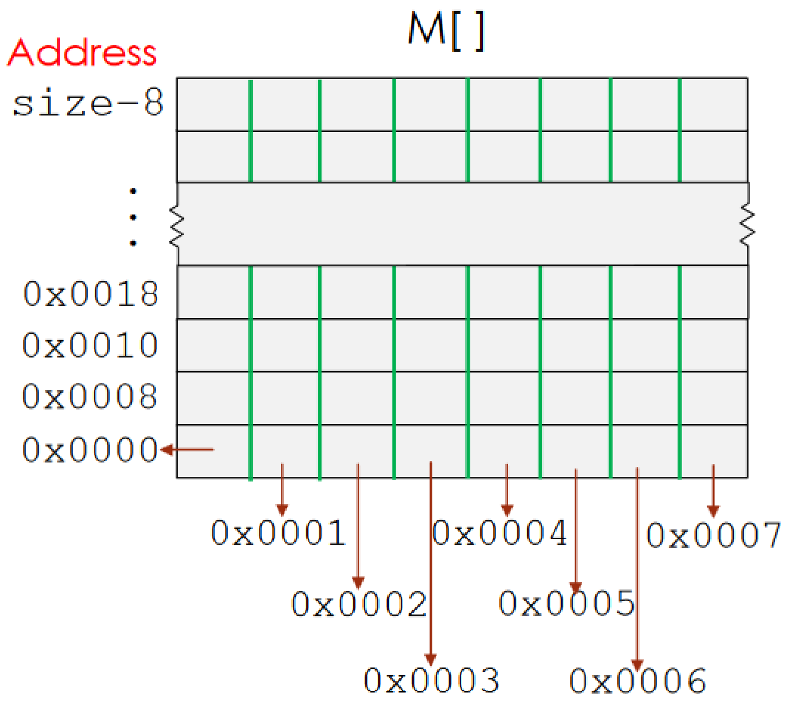
```
void who(…) {
  int sum = 0;
  ...
  y = amI(x);
  sum = x + y;
  return;
}
```

```
int amI(int i)
{
  int t = 3*i;
  int v[10];
  ...
  return v[t];
}
```

- Above mechanisms implemented with machine code instructions and described as a set of conventions (ISA)

Remember from Lecture 2: Closer look at memory

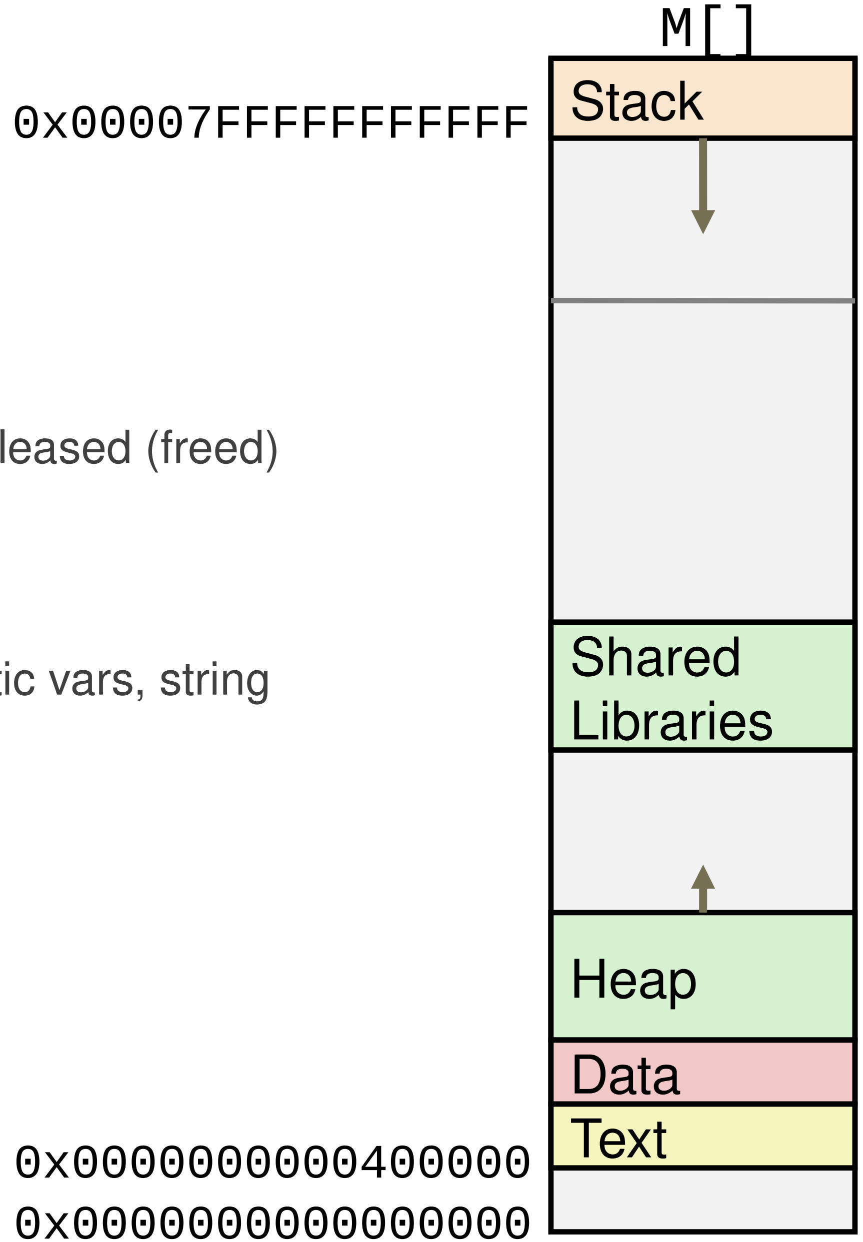
- Seen as a linear array of bytes
- 1 byte (8 bits) smallest addressable unit of memory
  - Byte-addressable
- Each byte has a unique address
- Computer reads a “word size” worth of bits at a time
- Compressed view of memory



Memory Layout

segments

- **Stack**
  - Runtime stack, e. g., local variables
- **Heap**
  - Dynamically allocated as needed, explicitly released (freed)
  - When call malloc(), free(), new(), delete, ...
- **Data**
  - Statically allocated data, e.g., global vars, static vars, string constants
- **Text**
  - Executable machine instructions
  - Read-only
- **Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

Where does everything go?

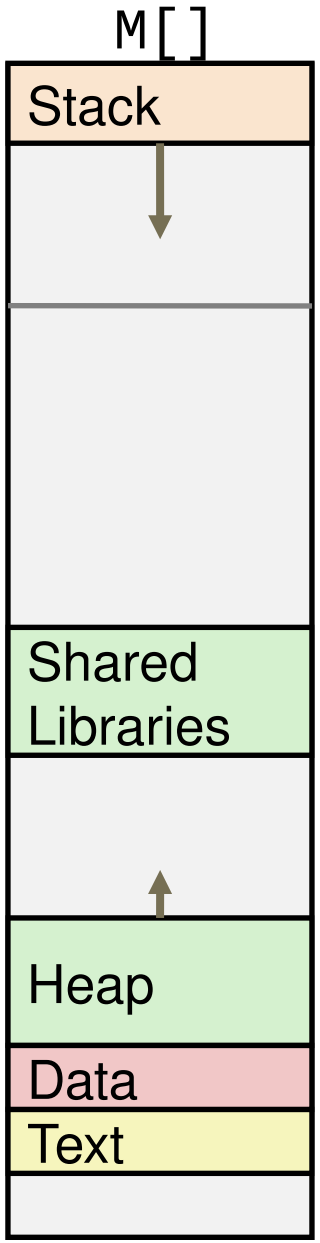
```
#include ...

char hugeArray[1 << 31]; /* 2³¹ = 2GB */
int global = 0;

int useless(){ return 0; }

int main ()
{
    void *ptr1, *ptr2;
    int local = 0;
    ptr1 = malloc(1 << 28); /* 2²⁸ = 256 MB*/
    ptr2 = malloc(1 << 8);  /* 2⁸  = 256 B*/

/* Some print statements ... */
}
```



Closer look at function call pattern

- A function may call a function, which may call a function, which may call a function, …

```
who(…) {
   ...
   ...
   are();
   ...
   ...
}
```

```
are(…) {
   ...
   you();
   ...
   you();
   ...
}
```

```
you(…) {
   ...
   ...
   ...
   ...
   ...
}
```

- When a function (*callee*) terminates and returns, its most recent *caller* resumes which eventually terminates and returns and its most recent *caller* resumes …
- Does this pattern remind you of anything?

Stack

Definition:

A stack is a last-in-first-out (LIFO) data structure with two characteristic operations:

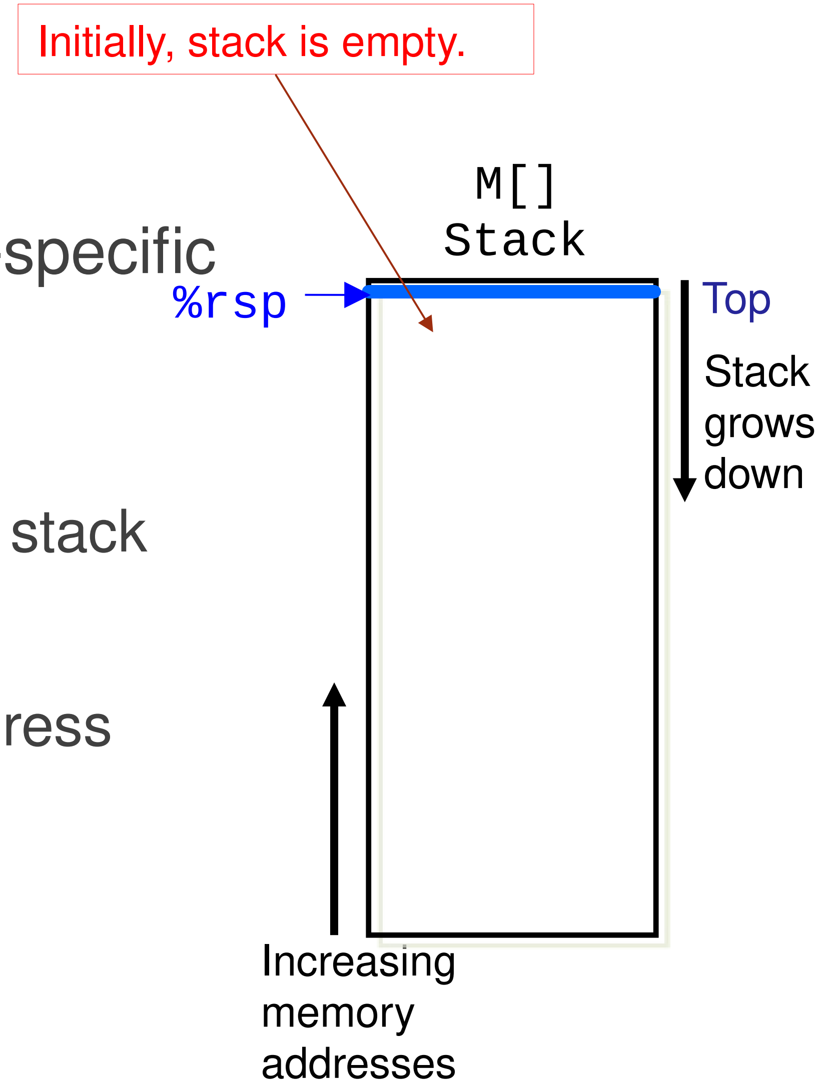
- `push(data)`
- `data = pop( ) or pop(&data)`

Do not have access to anything except what is on (at) top

Source: https://www.thebroad.org/art/robert-therrien/no-title-8

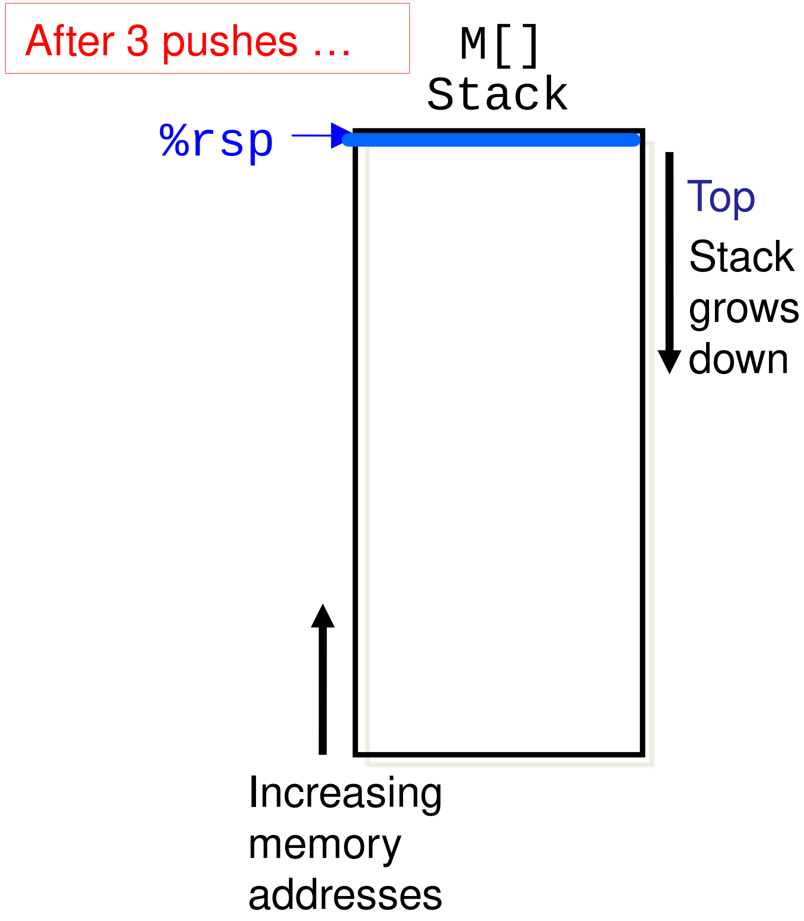
Closer look at stack

- x86-64 assembly language has stack-specific instructions and registers
- `%rsp`
  - Points to address of last used byte on stack
  - Initialized to “top of stack” at startup
  - Stack grows towards low memory address
- `pushq src`
- `popq dest`



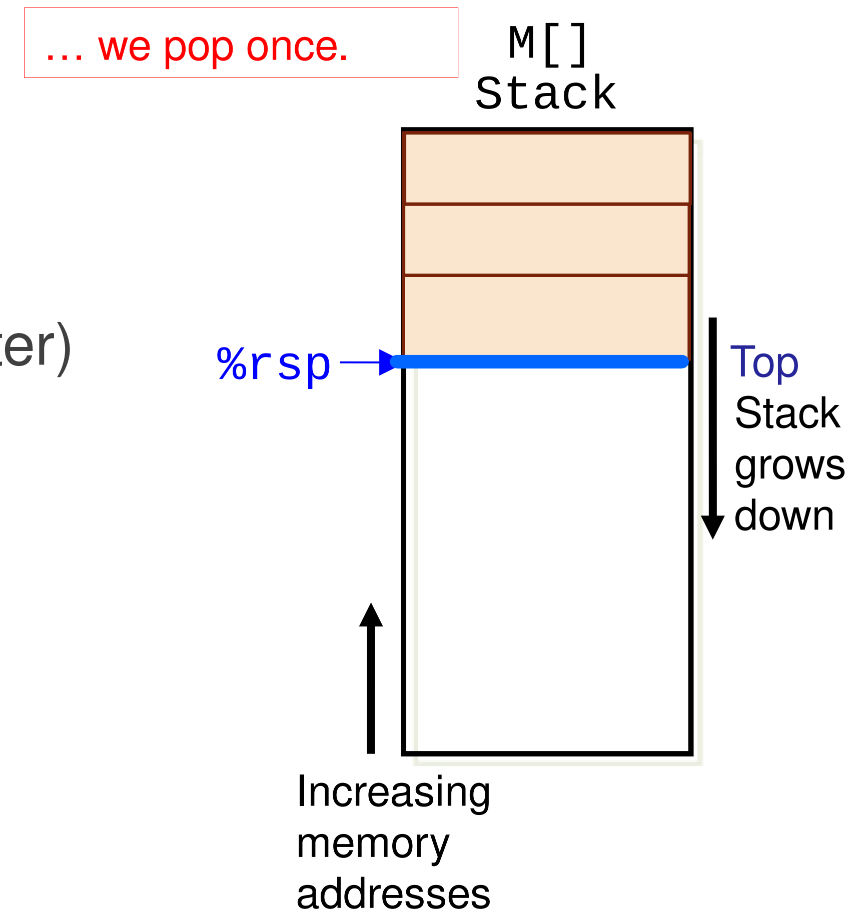
x86-64 stack instruction: push

- `pushq src`
  - Fetch value of operand `src`
  - Decrement `%rsp` by 8
  - Write value at address given by `%rsp`



x86-64 stack instruction: pop

- `popq dest`
  - Read value at %rsp (address) and store it in operand dest (must be register)
  - Increment `%rsp` by 8

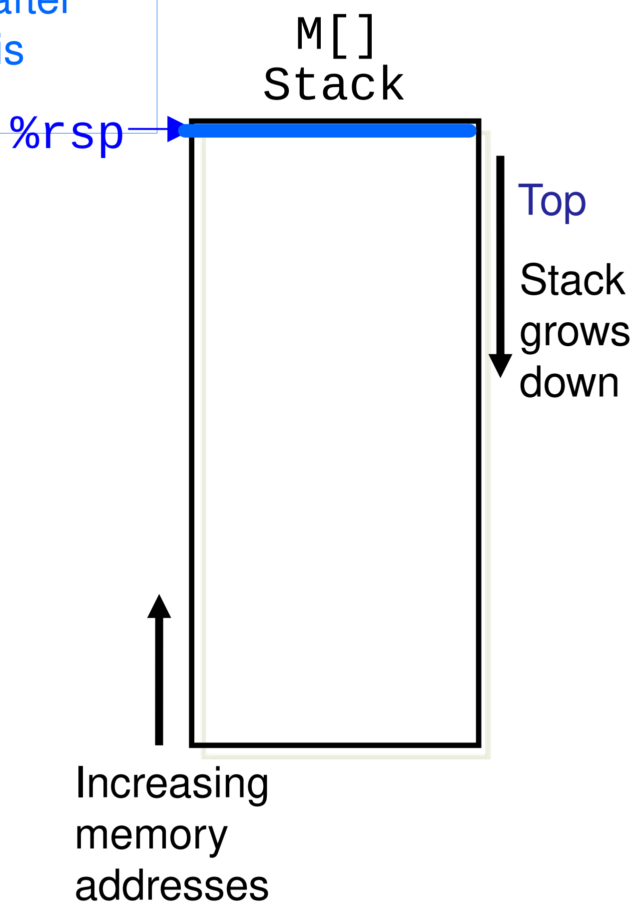


Passing control mechanism
x86-64 instruction: `call` and `ret`

After 1 call …

- `call func`
  - `push PC`
  - `jmp func (set PC to func)`

Effect: return address, i.e., the address of the instruction after `call func` (held in PC) is pushed onto the stack

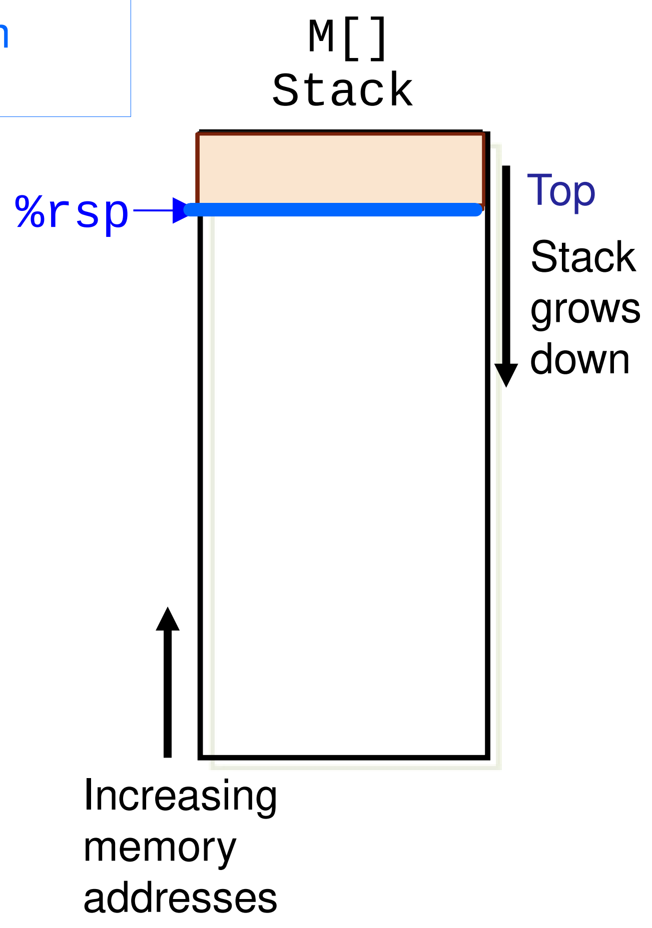


Passing control mechanism
x86-64 instruction: `call` and `ret`

After returning from the call …

Effect: return address, i.e., the address of instruction after `call func`, is pop'ed from the stack and stored in PC

- `ret`
  - `popq PC`
  - `jmp PC`



Example

```
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
    return;
}
```

```
long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

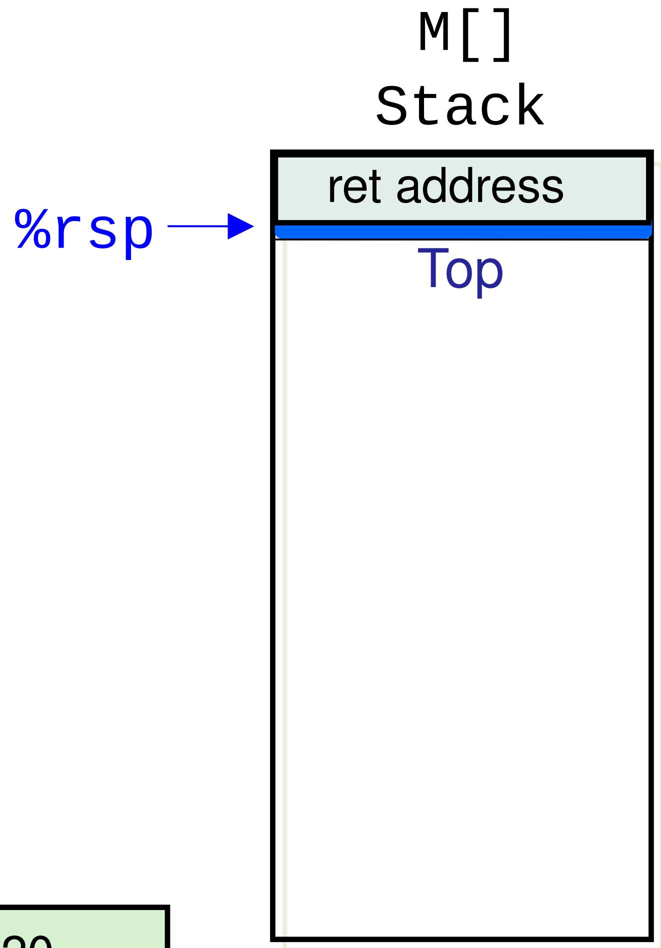
```
0000000000400540 <multstore>:
  400540: push   %rbx                # Save %rbx
  400541: mov    %rdx,%rbx           # Save dest
  400544: callq  400550 <mult2>      # mult2(x,y)
  400549: mov    %rax,(%rbx)         # Save at dest
  40054c: pop    %rbx                # Restore %rbx
  40054d: retq                       # Return
```

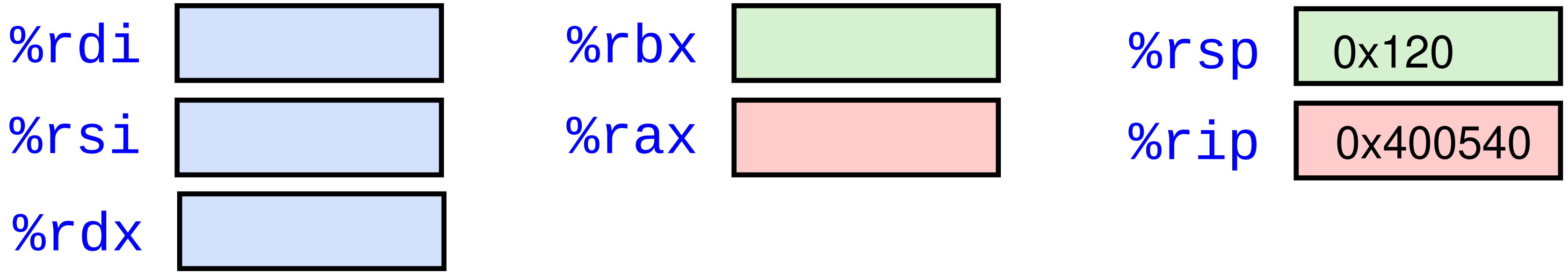
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

Example – Steps 1 and 2

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:   mov     %rdi,%rax    # a
  400553:   imul    %rsi,%rax    # a * b
  400557:   retq                 # Return
```

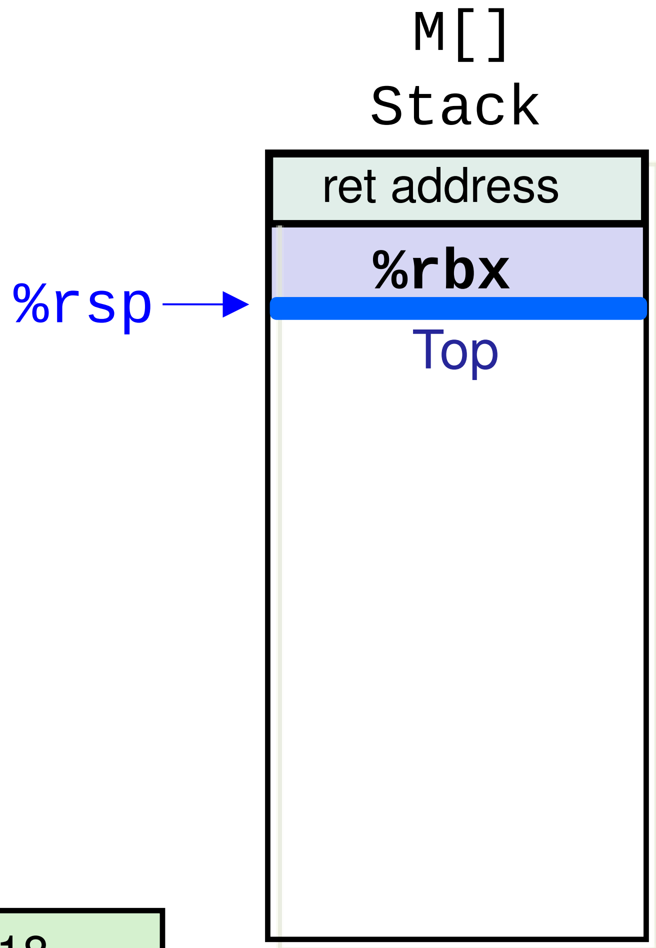


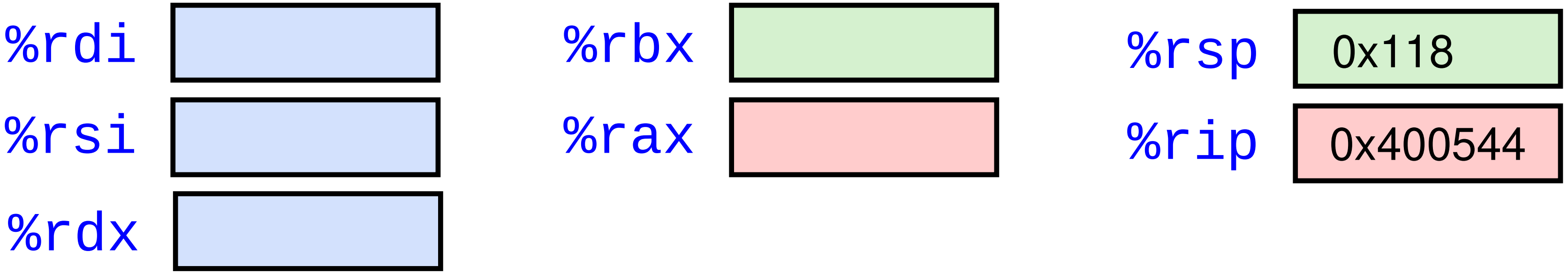


Example – Steps 3 and 4

```
0000000000400540 <multstore>:
  400540: push   %rbx             # Save %rbx
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: mov    %rax,(%rbx)      # Save at dest
  40054c: pop    %rbx             # Restore %rbx
  40054d: retq                    # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

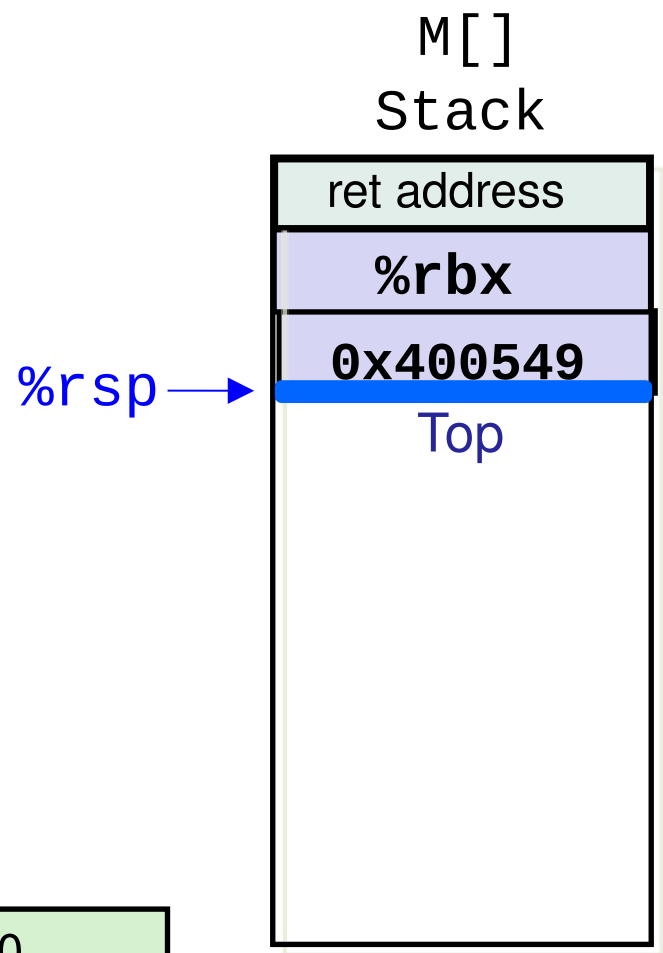




Example – Steps 5 and 6

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Save %rbx
  400541: mov    %rdx,%rbx         # Save dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Save at dest
  40054c: pop    %rbx              # Restore %rbx
  40054d: retq                     # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq                # Return
```

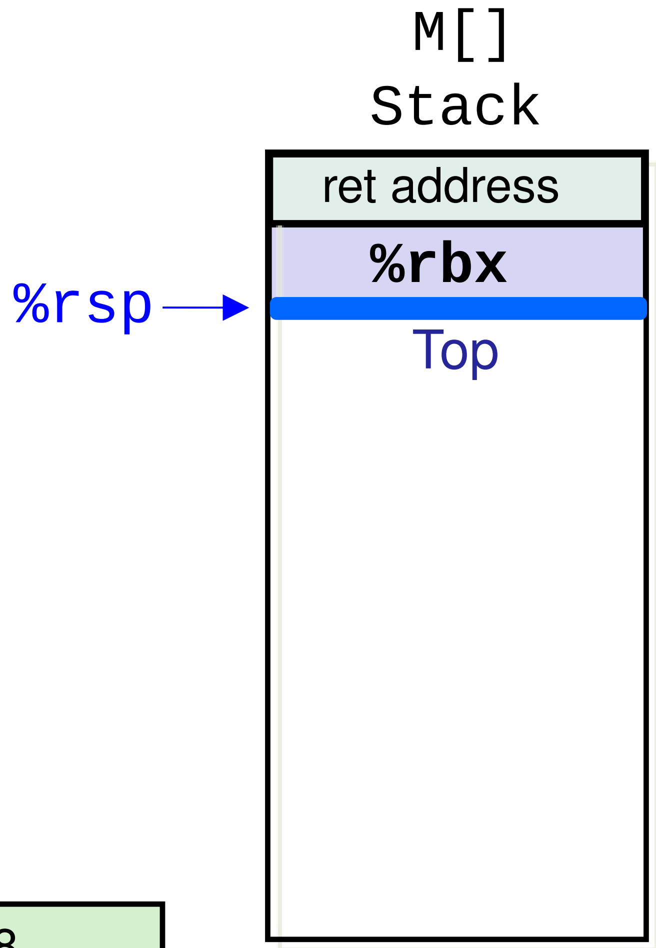


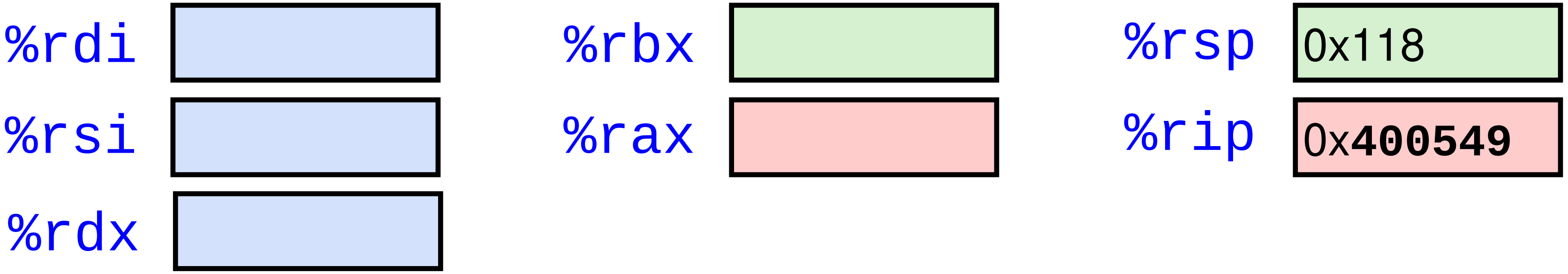
%rdi
%rsi
%rdx
%rbx
%rax
%rsp 0x110
%rip 0x400553

Example – Steps 7, 8 and 9

```
0000000000400540 <multstore>:
  400540: push   %rbx             # Save %rbx
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: mov    %rax,(%rbx)      # Save at dest
  40054c: pop    %rbx             # Restore %rbx
  40054d: retq                    # Return
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax   # a
  400553:  imul   %rsi,%rax   # a * b
  400557:  retq               # Return
```





Summary

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

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