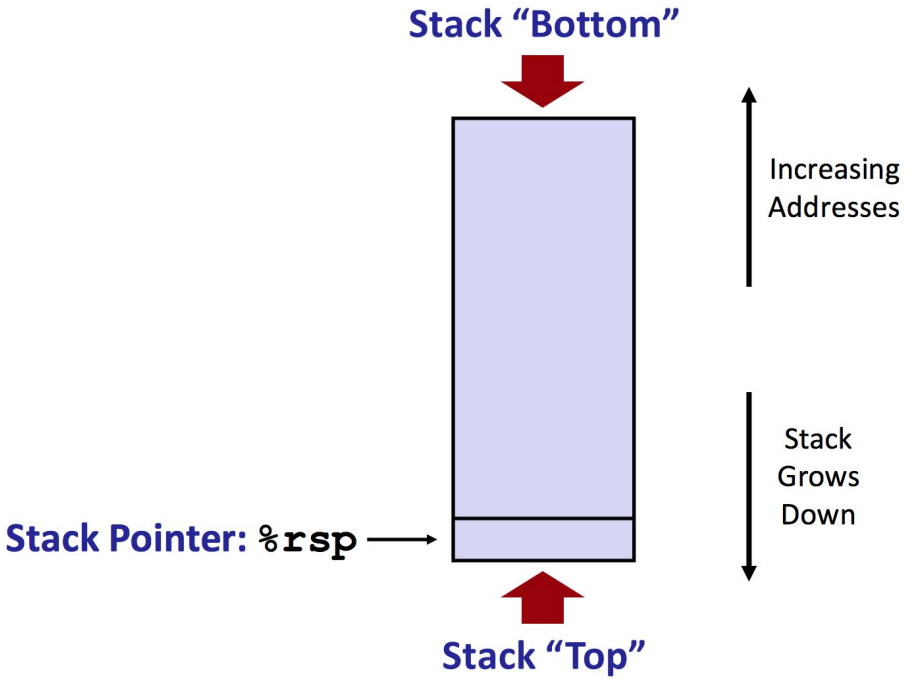
CS 33: Computer Organization

Dis 1B: Week 4 Discussion

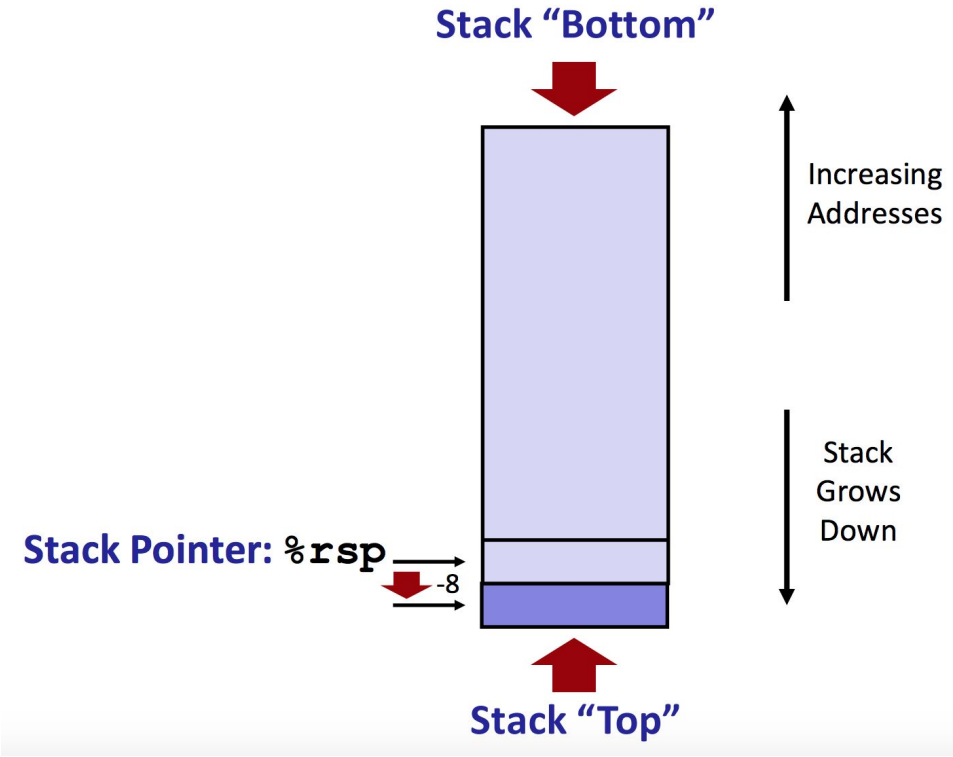
Stack

- Used for static memory allocation
- Local variables, arguments, and return address
- Each function has its own stack frame
- %rsp contains the lowest stack address
- Wait… Where do global variables and dynamically allocated get stored in?



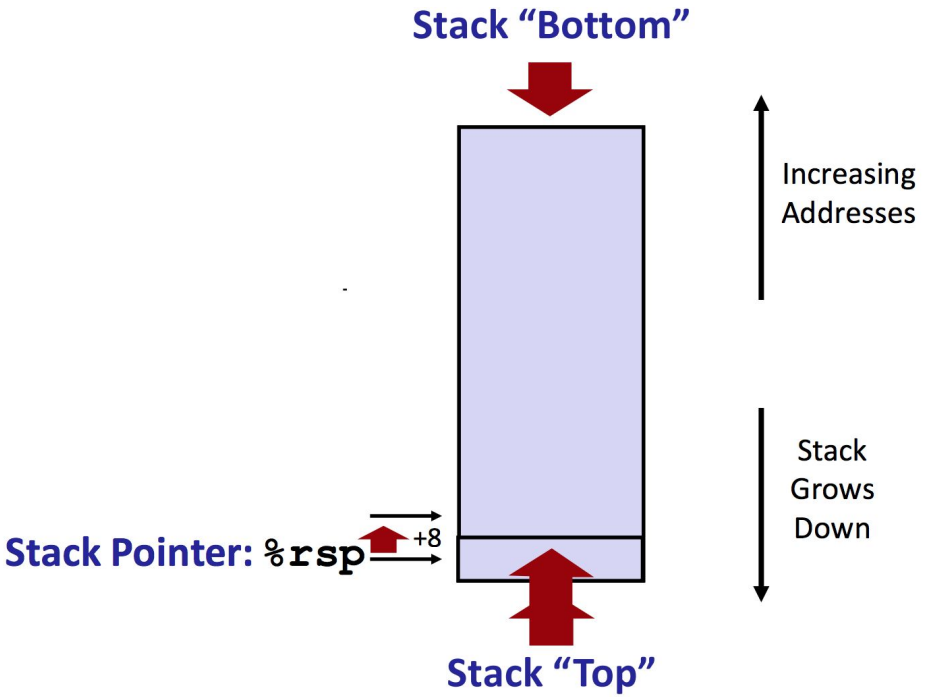
pushq

- pushq Source
- Decrement %rsp by 8
- Pushing an 8 bit value into stack



popq

- popq Dest
- Increment %rsp by 8
- Popping an 8 bit value from stack to destination

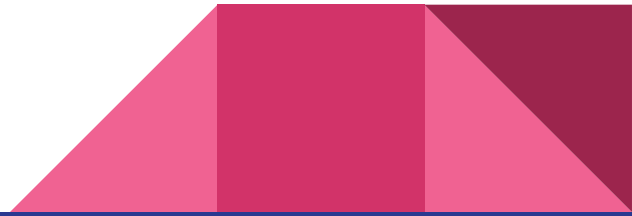


Procedure Control Flow

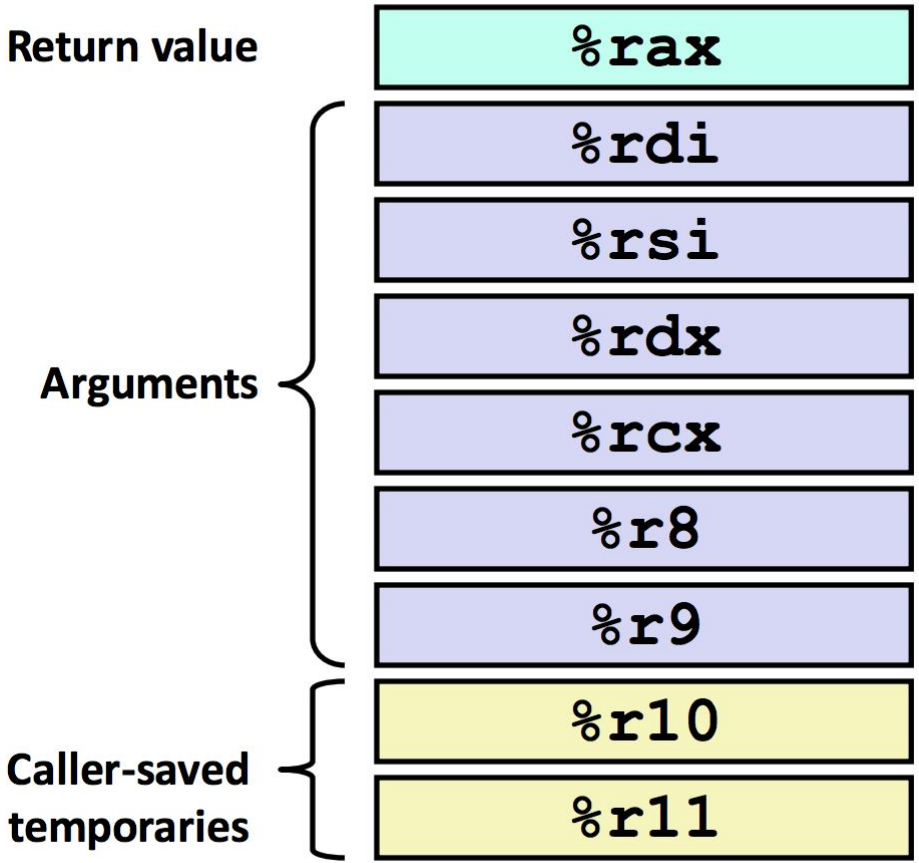
- Stack is used to support procedure call and return
- call label
  - Push return address on stack
  - Jump to label
- ret
  - Pop return address from stack
  - Jump to the return address that we just popped off

Caller vs. Callee

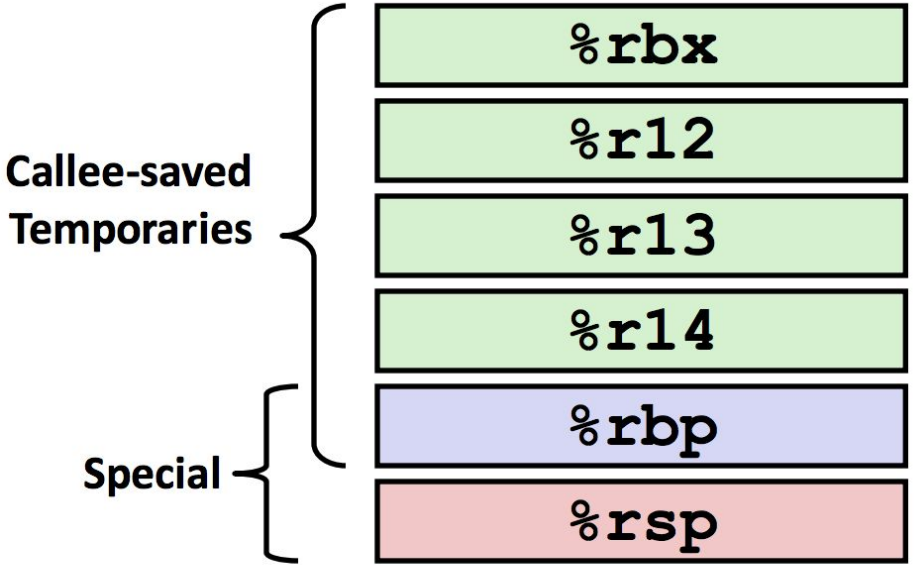
- Some registers are designated for caller and others are designated for callee



- **`%rax`**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **`%rdi`, …, `%r9`**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **`%r10`, `%r11`**
  - Caller-saved
  - Can be modified by procedure

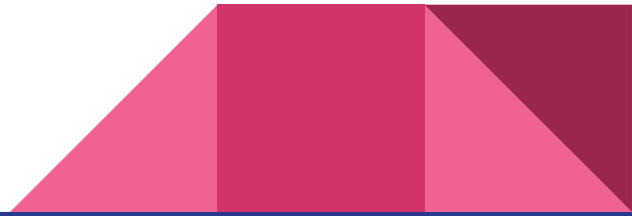


- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Callee-saved
  - Callee must save & restore
- **`%rbp`**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- **`%rsp`**
  - Special form of callee save
  - Restored to original value upon exit from procedure



What does all these mean?

- I have no idea what’s going on right now
- Let’s just see an example



```
long foo()
{
  long a = 0xfeed;
  long b = 0xface;
  long c = bar(a, b) + 1;
  return c;
}

int main()
{
  foo();
}
```

```
void useless()
{
  int a = 0;
}

long bar(long a, long b)
{
  unsigned long ret =
((unsigned long) (a << 16)) |
((unsigned long) b);
  useless();
  return ret;
}
```

Adapted from CS 33 Discussion Slides by Uen-Tao Wang

```
long foo()
{
  long a = 0xfeed;
  long b = 0xface;
  long c = bar(a, b) + 1;
  return c;
}
```

```
Dump of assembler code for function foo (gcc invoked with
no arguments):
   0x000000000040050c <+0>:     push   %rbp
   0x000000000040050d <+1>:     mov    %rsp,%rbp
   0x0000000000400510 <+4>:     sub    $0x20,%rsp
   0x0000000000400514 <+8>:     movq   $0xfeed,-0x8(%rbp)
   0x000000000040051c <+16>:    movq   $0xface,-0x10(%rbp)
   0x0000000000400524 <+24>:    mov    -0x10(%rbp),%rdx
   0x0000000000400528 <+28>:    mov    -0x8(%rbp),%rax
   0x000000000040052c <+32>:    mov    %rdx,%rsi
   0x000000000040052f <+35>:    mov    %rax,%rdi
   0x0000000000400532 <+38>:    callq  0x4004d6 <bar>
   0x0000000000400537 <+43>:    add    $0x1,%rax
   0x000000000040053b <+47>:    mov    %rax,-0x18(%rbp)
   0x000000000040053f <+51>:    mov    -0x18(%rbp),%rax
   0x0000000000400543 <+55>:    leaveq
   0x0000000000400544 <+56>:    retq
```

Adapted from CS 33 Discussion Slides by Uen-Tao Wang

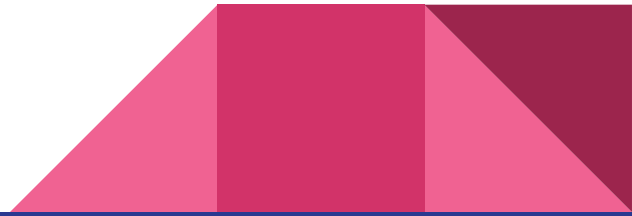
```
long bar(long a, long b)
{
  unsigned long ret =
((unsigned long) (a << 16)) |
((unsigned long) b);
  useless();
  return ret;
}
```

```
Dump of assembler code for function bar:
   0x00000000004004d6 <+0>:     push   %rbp
   0x00000000004004d7 <+1>:     mov    %rsp,%rbp
   0x00000000004004da <+4>:     sub    $0x20,%rsp
   0x00000000004004de <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000004004e2 <+12>:    mov    %rsi,-0x20(%rbp)
   0x00000000004004e6 <+16>:    mov    -0x18(%rbp),%rax
   0x00000000004004ea <+20>:    shl    $0x10,%rax
   0x00000000004004ee <+24>:    mov    %rax,%rdx
   0x00000000004004f1 <+27>:    mov    -0x20(%rbp),%rax
   0x00000000004004f5 <+31>:    or     %rdx,%rax
   0x00000000004004f8 <+34>:    mov    %rax,-0x8(%rbp)
   0x00000000004004fc <+38>:    mov    $0x0,%eax
   0x0000000000400501 <+43>:    callq  0x4004c8 <useless>
   0x0000000000400506 <+48>:    mov    -0x8(%rbp),%rax
   0x000000000040050a <+52>:    leaveq
   0x000000000040050b <+53>:    retq
```

Adapted from CS 33 Discussion Slides by Uen-Tao Wang

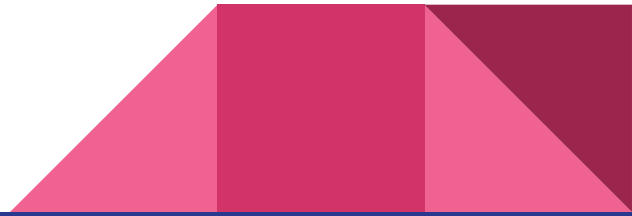
Bomb Lab!

- PLEASE READ INSTRUCTIONS CAREFULLY
- Except for this line
  - linux> ./bomb psol.txt
- As you can tell, this command will explode your bomb



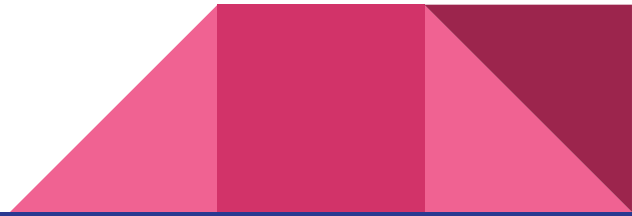
Bomb Lab!

- How to use a debugger
- Learn how to understand assembly language



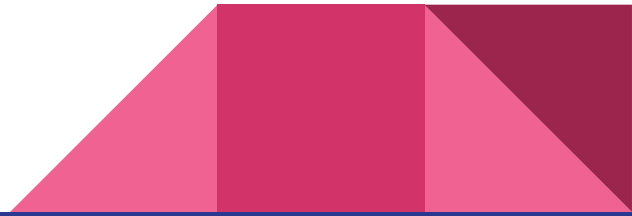
Bomb Lab!

- Download the source code from http://lnxsrv04.seas.ucla.edu:15213/
- Make sure you are on UCLA network or VPN



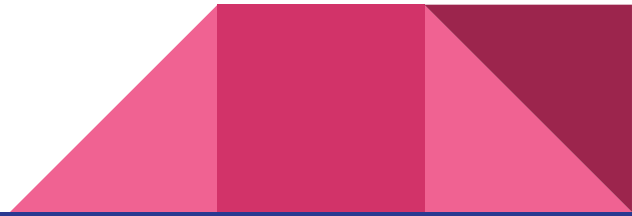
Bomb Lab!

- objdump -d bomb > disassemble.txt
- psol.txt
  - If you save your solutions in psol.txt, your solutions will be automatically applied



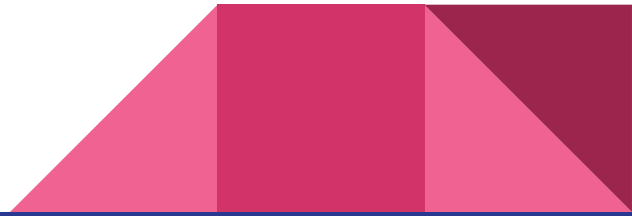
Bomb Lab!

- Always use gdb
  - If you just run ./bomb… Boom!
- gdb bomb



gdb

- Useful instructions
  - break <function_name>
    - Sets breakpoint right after entering the function
  - break *0x80483c3
    - Sets breakpoint at address 0x80483c3
  - continue
    - I’m smarter than breakpoint: continue executing until program terminates or hit another breakpoint

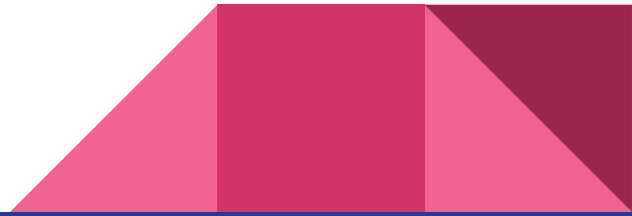


gdb

- print /x $rsp
  - Print contents of $rsp in hexadecimal
- x/*n*b $rsp
  - Examine n-bytes starting at address in $rsp
  - Shows the most significant bit first. Think about how different variables are stored!
    - Endianness

Tips!

- Draw a stack
  - Visualization makes everything so much easier
- Take your time
  - Brute force takes a lot longer than simply solving the problem
- Try it before Tuesday
  - This is a great potential midterm problem



Midterm

- C Puzzles (int, float)
- x86-64
- Data in memory (pointers, arrays, structs, unions, etc.)