Machine Level Programming: Procedures

CSCI 237: Computer Organization
16th Lecture, Wednesday, October 16

Kelly Shaw

1

1

Administrative Details

- Lab #3 due Wednesday today at 11pm
  - Any questions?
- Read CSAPP 3.7-3.8
- Weekly quiz open at 2:30 and is due by Friday at 2:30
- Snack and Gab (4:10-4:30 today)
- Midterm in lab on Wednesday 10/23
  - Closed book and closed notes
  - Review session on Monday 10/21 7-8:30pm in Physics 205
- Apply to be a TA!
- TA feedback forms
- CS Colloquium talk today at 2:35pm in Wege
  - What I did last summer part 2

2

2

Last Time: Machine-Level Programming: Control

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements

3

3

Today: Machine-Level Programming: Procedures

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

4

4

Handling Fall-Through

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
merge:
    w += z;
```


5

Code Blocks (x == 2, x == 3)

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
   . . .
}
```

```
.L5:                    # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx        #  y/z
   jmp     .L6         #  goto merge
.L9:                    # Case 3
   movl    $1, %eax    #  w = 1
.L6:                    # merge:
   addq    %rcx, %rax  #  w += z
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

6

Code Blocks (x == 5, x == 6, default)

```
switch(x) {
   . . .
   case 5:  // .L7
   case 6:  // .L7
       w -= z;
       break;
   default: // .L8
       w = 2;
}
```

```
.L7:                    # Case 5,6
   movl   $1, %eax     #  w = 1
   subq   %rdx, %rax   #  w -= z
   ret
.L8:                    # Default:
   movl   $2, %eax     #  2
   ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

7

Finding Jump Table in Binary

```
00000000004005e0 <switch_eg>:
4005e0:       48 89 d1                mov    %rdx,%rcx
4005e3:       48 83 ff 06             cmp    $0x6,%rdi
4005e7:       77 2b                   ja     400614 <switch_eg+0x34>
4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
4005f0:       48 89 f0                mov    %rsi,%rax
4005f3:       48 0f af c2             imul   %rdx,%rax
4005f7:       c3                      retq
4005f8:       48 89 f0                mov    %rsi,%rax
4005fb:       48 99                   cqto
4005fd:       48 f7 f9                idiv   %rcx
400600:       eb 05                   jmp    400607 <switch_eg+0x27>
400602:       b8 01 00 00 00          mov    $0x1,%eax
400607:       48 01 c8                add    %rcx,%rax
40060a:       c3                      retq
40060b:       b8 01 00 00 00          mov    $0x1,%eax
400610:       48 29 d0                sub    %rdx,%rax
400613:       c3                      retq
400614:       b8 02 00 00 00          mov    $0x2,%eax
400619:       c3                      retq
```


8

Finding Jump Table in Binary (cont.)

```
  00000000004005e0 <switch_eg>:
  . . .
  4005e9:       ff 24 fd f0 07 40 00    jmpq   *0x4007f0(,%rdi,8)
  . . .
```

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
(gdb)
```

9

9

Finding Jump Table in Binary (cont.)

```
 % gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:       0x0000000000400614      0x00000000004005f0
0x400800:       0x00000000004005f8      0x0000000000400602
0x400810:       0x0000000000400614      0x000000000040060b
0x400820:       0x000000000040060b      0x2c646c25203d2078
```

```
  . . .
  4005f0:       48 89 f0                mov    %rsi,%rax
  4005f3:       48 0f af c2             imul   %rdx,%rax
  4005f7:       c3                      retq
  4005f8:       48 89 f0                mov    %rsi,%rax
  4005fb:       48 99                   cqto
  4005fd:       48 f7 f9                idiv   %rcx
  400600:       eb 05                   jmp    400607 <switch_eg+0x27>
  400602:       b8 01 00 00 00          mov    $0x1,%eax
  400607:       48 01 c8                add    %rcx,%rax
  40060a:       c3                      retq
  40060b:       b8 01 00 00 00          mov    $0x1,%eax
  400610:       48 29 d0                sub    %rdx,%rax
  400613:       c3                      retq
  400614:       b8 02 00 00 00          mov    $0x2,%eax
  400619:       c3                      retq
```

10

10

Summarizing Control Instructions

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

11

11

Today: Machine-Level Programming: Procedures

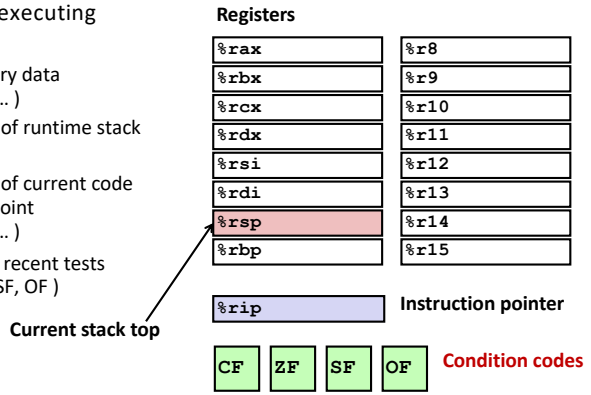
- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

13

13

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)



14

14

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

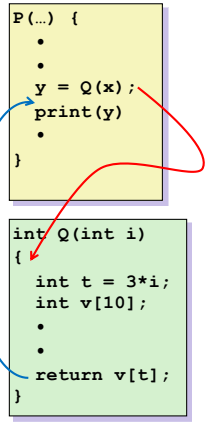
```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

15

15

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

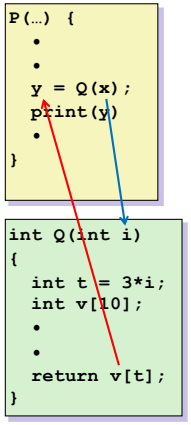


16

16

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required



17

17

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

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

18

18

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

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

19

19

Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of a procedure uses only those mechanisms required

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

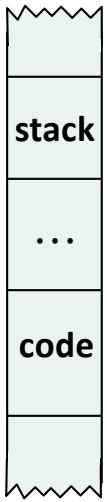
```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

20

20

x86-64 Stack

- Region of memory managed with stack discipline
- Memory viewed as array of bytes
- Different regions have different purposes

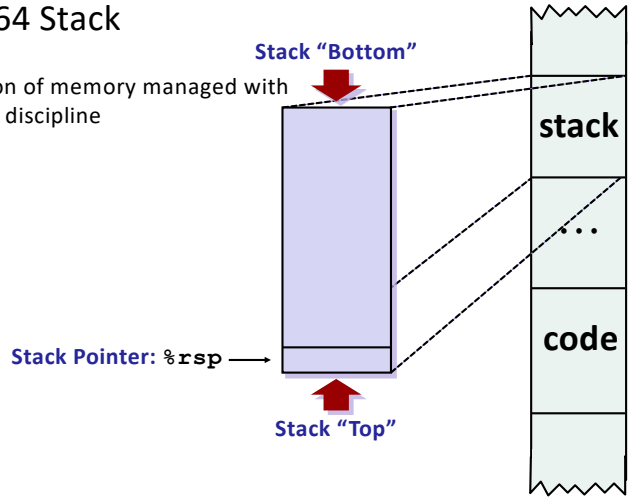


21

21

x86-64 Stack

- Region of memory managed with stack discipline

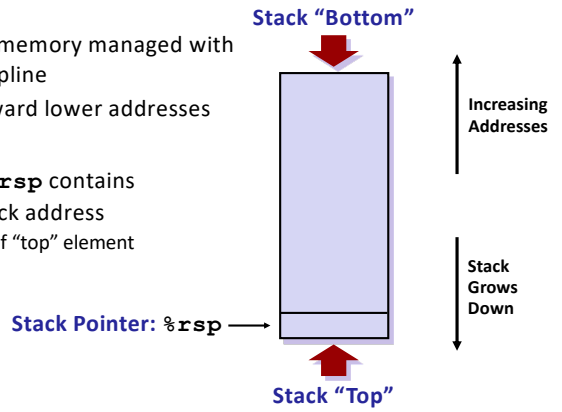


22

22

x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of "top" element

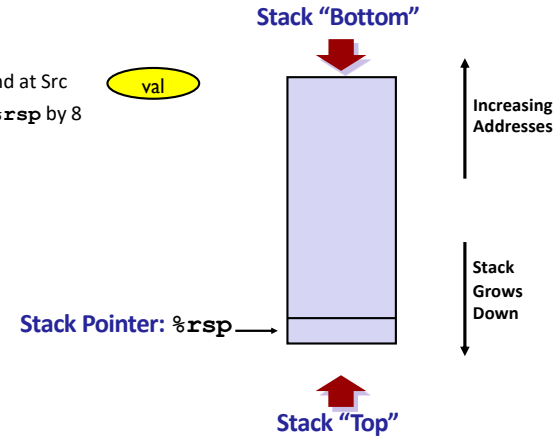


23

23

x86-64 Stack: Push

- **`pushq`** Src
  - Fetch operand at Src
  - Decrement `%rsp` by 8

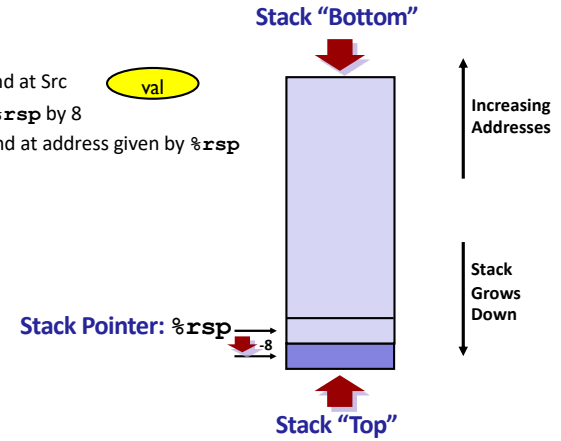


24

24

x86-64 Stack: Push

- **`pushq`** Src
  - Fetch operand at Src
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`

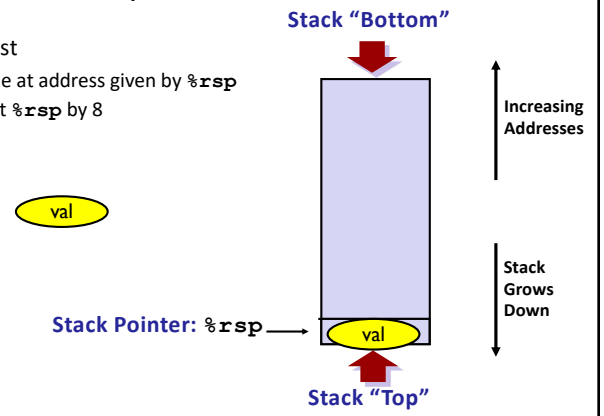


25

25

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8

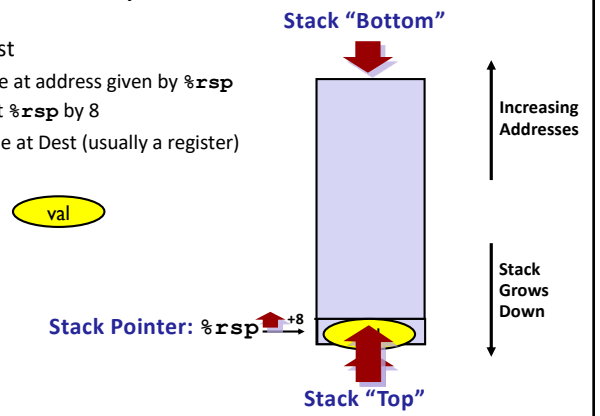


26

26

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

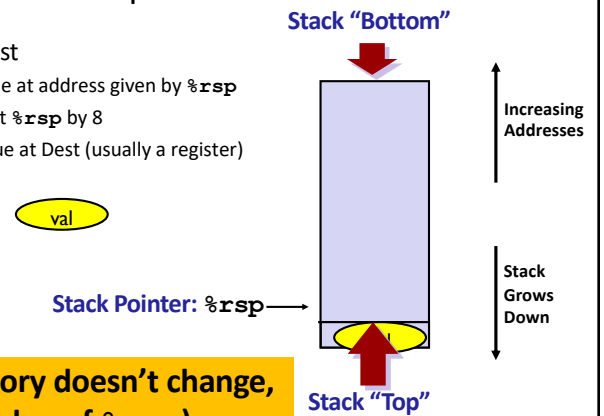


27

27

x86-64 Stack: Pop

- `popq` Dest
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)



(The memory doesn't change, only the value of `%rsp`)

28

28

Code Examples

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

```
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
```

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

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

29

29

Procedure Control Flow

- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - (you will see `callq label` in gdb output)
  - Does two things:
    - Pushes return address on stack
    - Jumps to label
- Return address:
  - Address of the next instruction right after call
  - (Example on next slide)
- Procedure return: **`ret`**
  - Does two things (inverse of call):
    - Pop address from stack
    - Jump to address

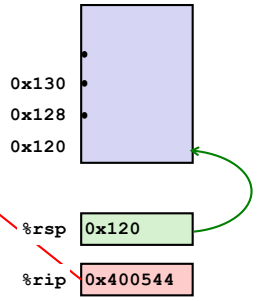
30

30

Control Flow Example #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



`callq 400550`: Push return addr onto stack, jump to label <mult2>.

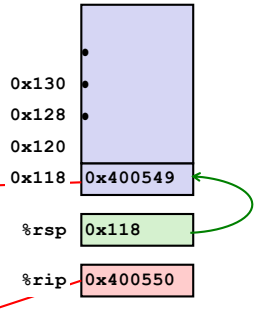
31

31

Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



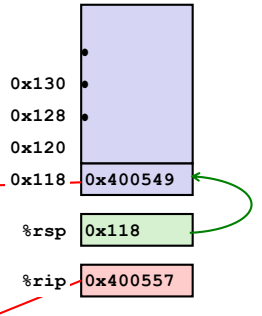
32

32

Control Flow Example #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



`retq`: Pop return addr from stack, jump to addr.

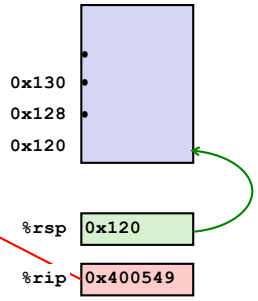
33

33

Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



34

34

Today: Machine-Level Programming: Procedures

- Intro to data-dependent control
  - Condition codes
  - Conditional branches
  - Conditional data
  - Loops
  - Switch Statements
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

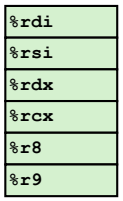
35

35

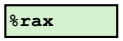
Procedure Data Flow

Registers

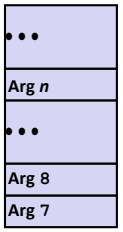
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed (e.g., more than 6 integral arguments)!

36

36