Recap

- 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
    - Allows k-way branch in O(1) operations
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)

115

Conditions from Condition Codes

- When is condition ge true (jge, setge)?
  - Overflow: a >= 0, b < 0; SF = 1
  - No overflow:
    - a >= 0, b < 0; SF = 0
    - a >= 0, b >= 0; SF = 0
    - a < 0, b < 0; SF = 0
  - OF.SF + ~OF. ~SF = ~(OF^SF)

116

116

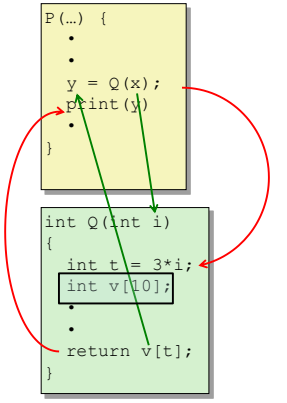
Today

- Procedures
  - **Stack Structure**
  - **Calling Conventions**
    - **Passing control**
    - **Passing data**
    - **Managing local data**
  - **Illustration of Recursion**

117

Mechanisms in Procedures

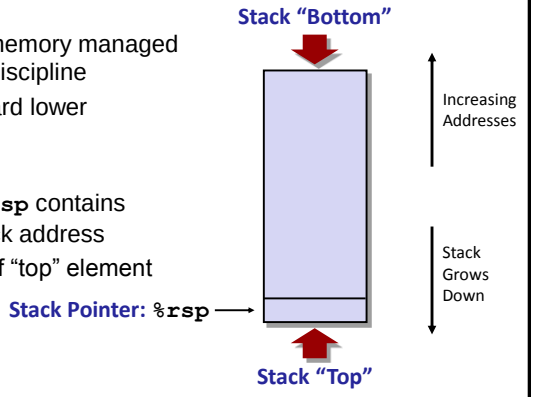
- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- Mechanisms all implemented with machine instructions
- x86-64 implementation of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



118

x86-64 Stack

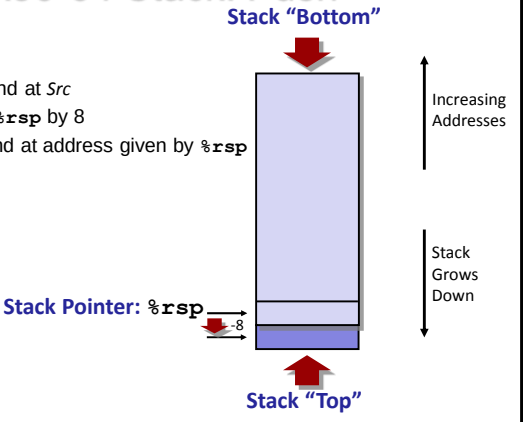
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
  - address of "top" element



119

x86-64 Stack: Push

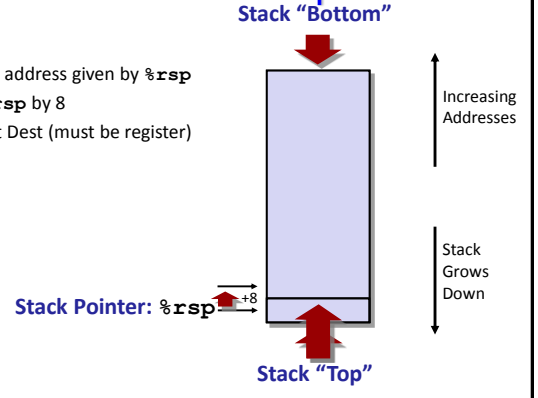
- `pushq` *Src*
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`



120

x86-64 Stack: Pop

- `popq` ***Dest***
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (must be register)



121

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - **Passing control**
    - Passing data
    - Managing local data
  - Illustration of Recursion

122

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
```

123

Procedure Control Flow

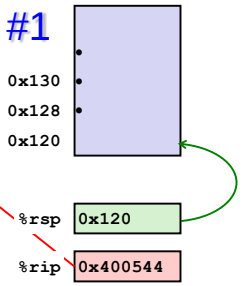
- Use stack to support procedure call and return
- Procedure call: **`call label`**
  - Push return address on stack
  - Jump to ***label***
- Return address:
  - Address of the next instruction right after call
  - Example from disassembly
- Procedure return: **`ret`**
  - Pop address from stack
  - Jump to address

124

Control Flow Example #1

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

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

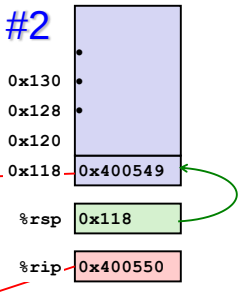


125

Control Flow Example #2

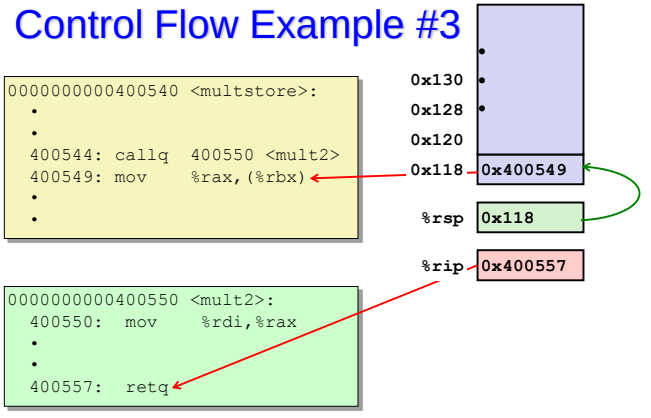
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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



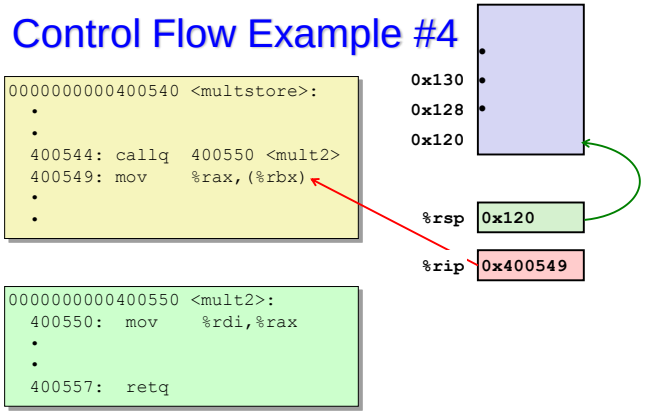
126

Control Flow Example #3



127

Control Flow Example #4



128

Today

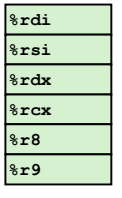
- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - **Passing data**
    - Managing local data
  - Illustrations of Recursion & Pointers

129

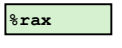
Procedure Data Flow

Registers

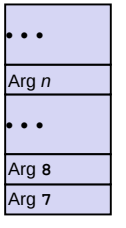
- First 6 arguments



- Return value



Stack



- Only allocate stack space when needed

130

Data Flow Examples

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  •••
```

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

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

131

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Managing local data**
  - Illustration of Recursion

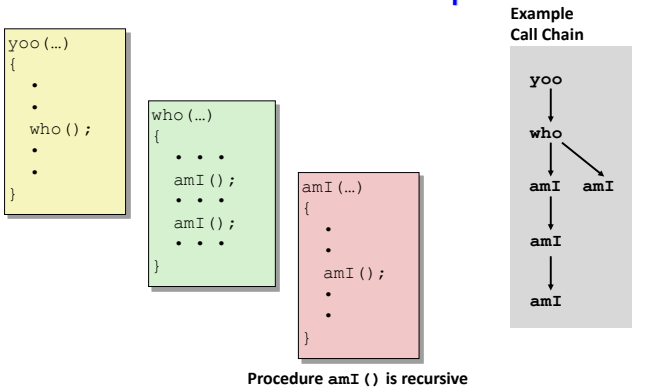
132

Stack-Based Languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "*Reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in ***Frames***
  - state for single procedure instantiation

133

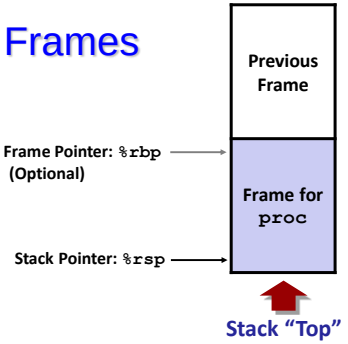
Call Chain Example



134

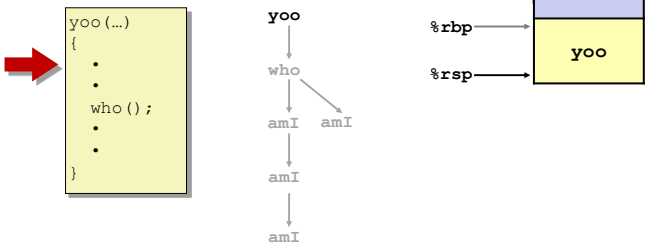
Stack Frames

- Contents
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space allocated when enter procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code
    - Includes pop by `ret` instruction



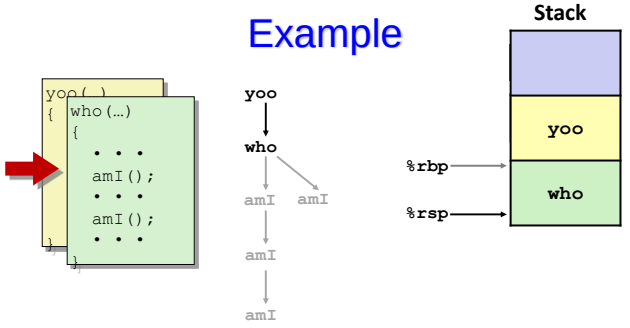
135

Example



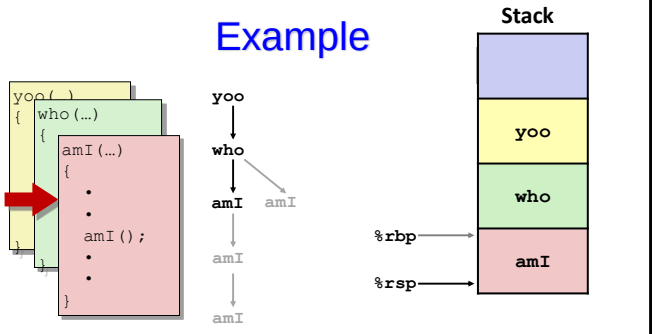
136

Example

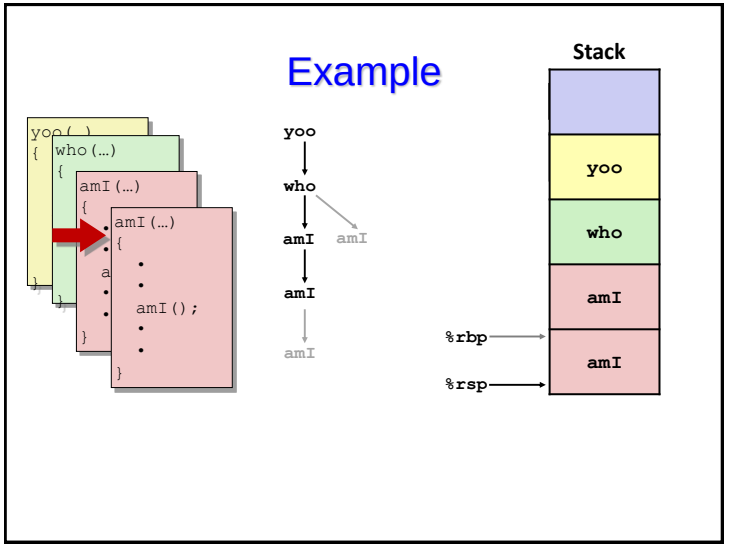


137

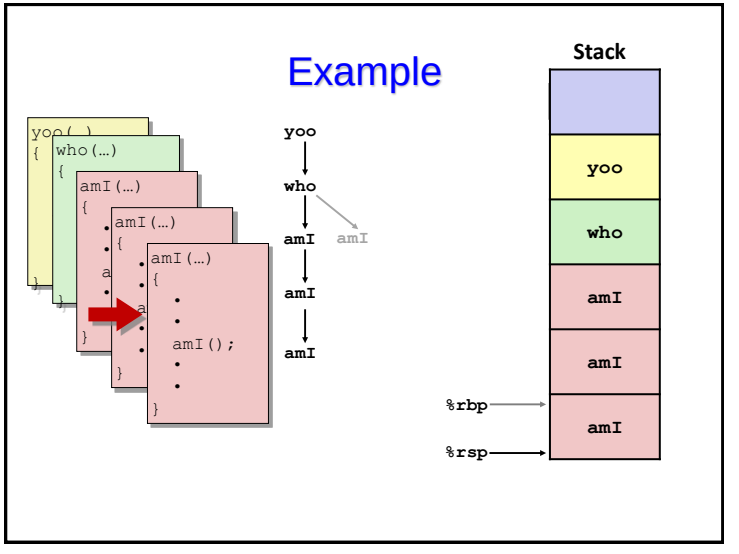
Example



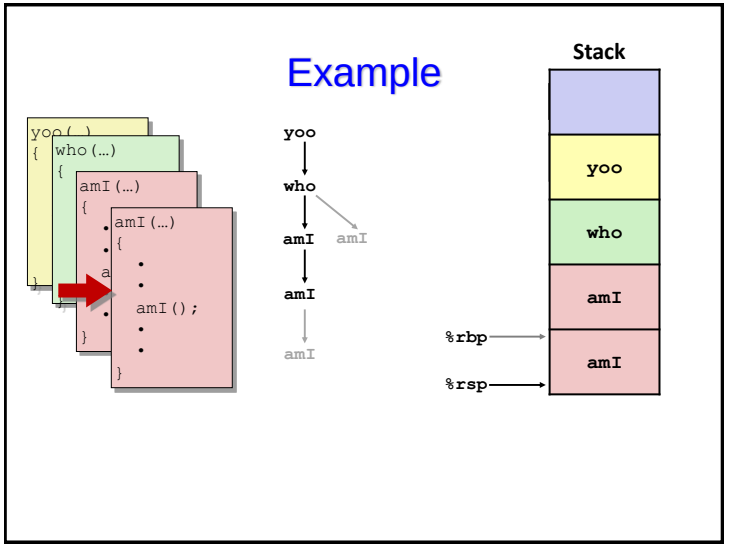
138



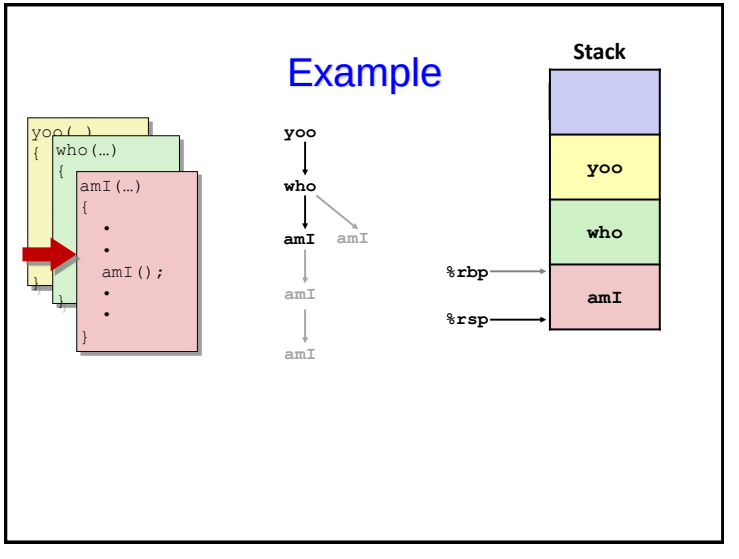
139



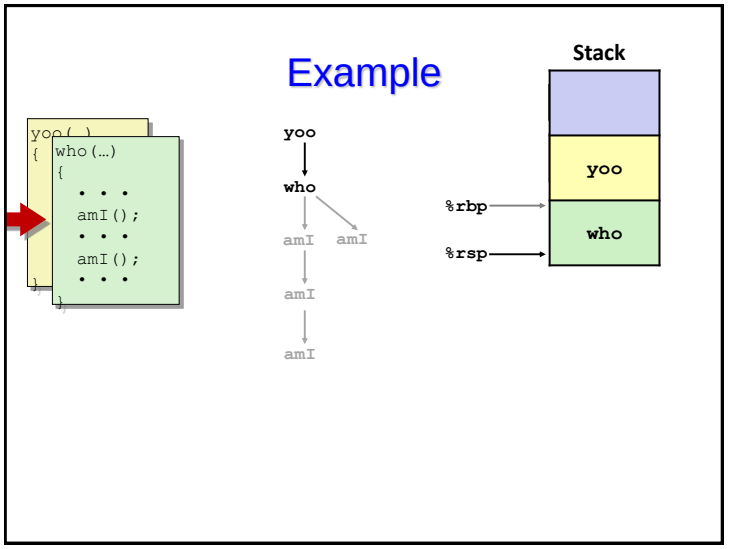
140



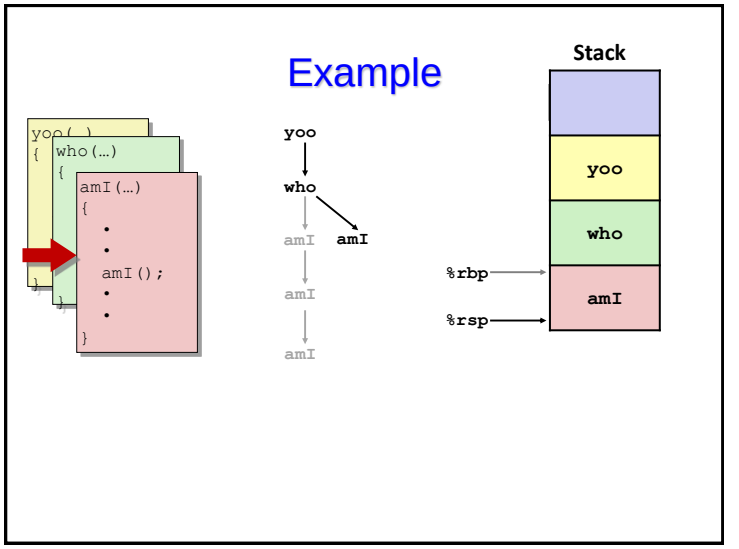
141



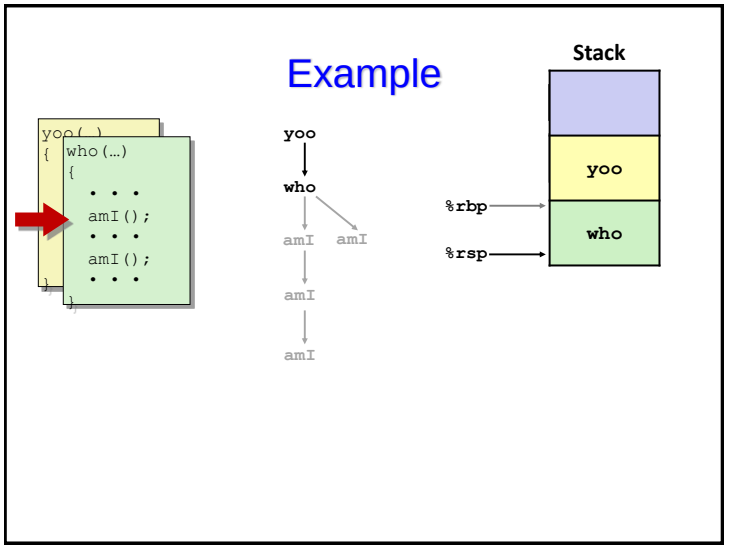
142



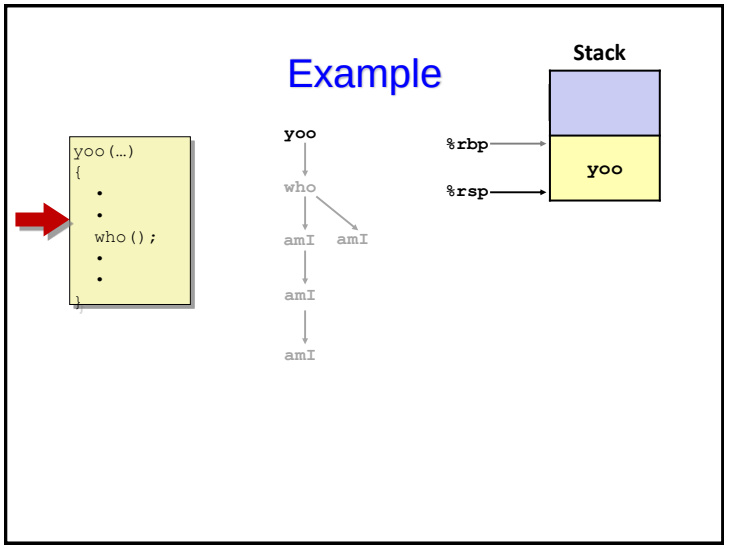
143



144



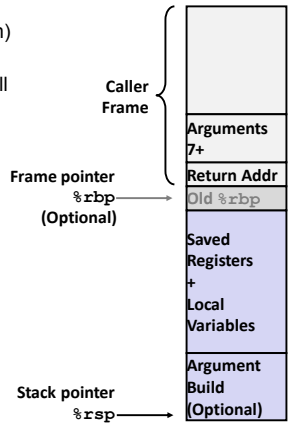
145



146

x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer (optional)
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



147

Example: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, Return value

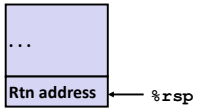
148

Example: Calling `incr` #1

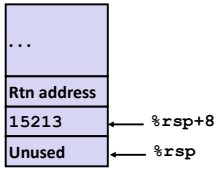
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure



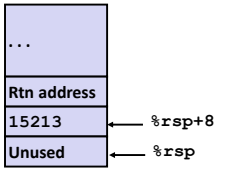
149

Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	3000

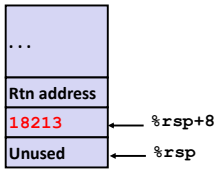
150

Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rdi	&v1
%rsi	3000

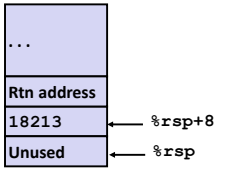
151

Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

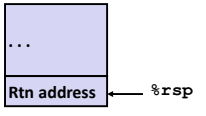
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
%rax	Return value

Updated Stack Structure



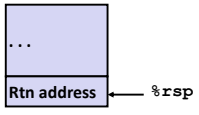
152

Example: Calling `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

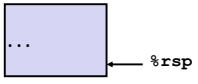
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



153

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

  - Contents of register `%rdx` overwritten by `who`
  - This could be trouble → something should be done!
    - Need some coordination

154

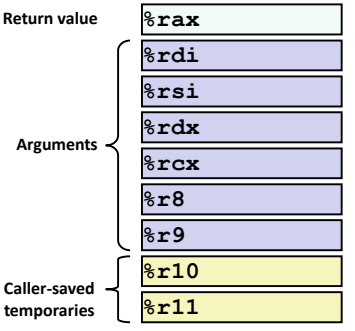
Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the ***caller***
  - **who** is the ***callee***
- Can register be used for temporary storage?
- Conventions
  - ***"Caller Saved"***
    - Caller saves temporary values in its frame before the call
  - ***"Callee Saved"***
    - Callee saves temporary values in its frame before using

155

x86-64 Linux Register Usage #1

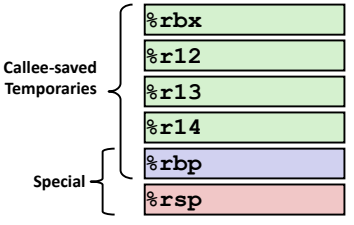
- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi`, ..., `%r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10`, `%r11`
  - Caller-saved
  - Can be modified by procedure



156

x86-64 Linux Register Usage #2

- `%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



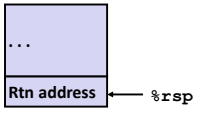
157

Callee-Saved Example #1

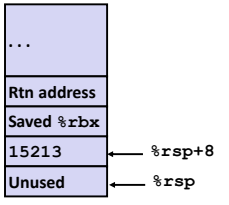
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure



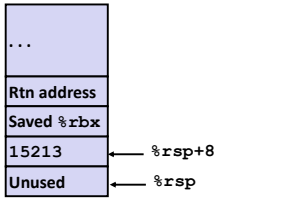
158

Callee-Saved Example #2

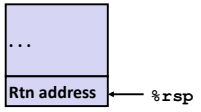
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



159

Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - **Illustration of Recursion**

160

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

161

Recursive Function Terminal Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

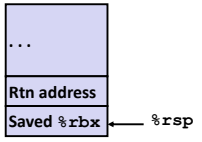
162

Recursive Function Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi # (by 1)
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rdi	x	Argument



163

Recursive Function Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi # (by 1)
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

164

Recursive Function Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi # (by 1)
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

165

Recursive Function Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi # (by 1)
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

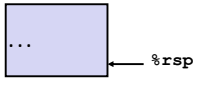
166

Recursive Function Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi # (by 1)
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



167

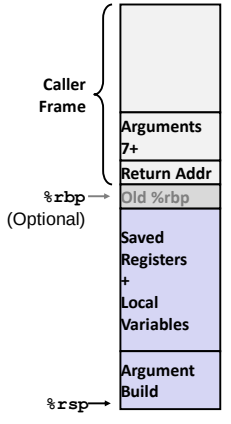
Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

168

x86-64 Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in %rax
- Pointers are addresses of values
  - On stack or global

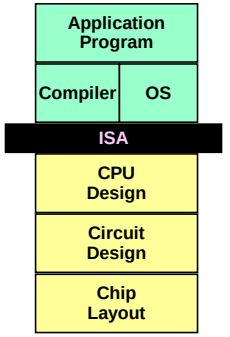


169

170

Instruction Set Architecture

- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Basic Issues in Instruction Set Design

Goal: find a language that makes it easy to build both the hardware and the compiler while maximizing performance and minimizing cost

- What operations (and how many) should be provided
- How (and how many) operands are specified
- What data types and sizes
- How to encode these into consistent instruction formats

Instruction Format

- Can be
  - fixed length – simpler decoding
  - variable length – higher code density/smaller program
  - Large instruction word (encoding multiple instructions and dependences among those instructions)
- If we have many memory operands per instruction and many addressing modes, we need an address specifier per operand/result
- If we have a load-store architecture with 1 address per instruction and one or two addressing modes, we can encode the addressing mode in the opcode

CISC Instruction Sets

- Complex Instruction Set Computer
- Dominant style through mid-80's

- Philosophy
  - Add instructions to perform "typical" programming tasks
- Variable length encodings
- Variable execution times
- Multiple formats for specifying operands
- Implementation artifacts hidden from machine-level programs
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Any instruction can access memory
  - `addl %eax, 12(%ebx,%ecx,4)`
    - requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions

RISC Instruction Sets

- Reduced Instruction Set Computer
- Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- Fixed length encoding
- Simple addressing formats (typically just base+displacement)
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
- No Condition codes - test instructions return 0/1 in register
- Implementation artifacts exposed to machine-level programs

CISC vs. RISC

- Original Debate
  - Strong opinions!
  - CISC proponents---easy for compiler, fewer code bytes
  - RISC proponents---better for optimizing compilers, can make run fast with simple chip design
- Current Status
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power