Loops

Expressing with Goto Code
- C allows goto statement
- Jump to position designated by label

```c
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

¢ C allows goto statement
¢ Jump to position designated by label

```c
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest)
    goto Else;
    result = x - y;
    goto Done;
    Else:
    result = y - x;
    Done:
    return result;
}
```

¢ Count number of 1’s in argument x ("popcount")
¢ Use conditional branch to either continue looping or to exit loop

Goto Version

```c
long pcount_jtm(unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x)
            goto loop;
    return result;
}
```

Do-While Version

```c
long pcount_dw(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```
Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at \( .L1 \)
- Jumping
  - Direct: \texttt{jmp \_L1}
  - Jump target is denoted by label \( \_L1 \)
  - Indirect: \texttt{jmp *L1 \{rdi, r8\}}
  - Start of jump table: \( \_L0 \)
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address: \( .L0 + n*8 \)
  - Only for 0 ≤ \( n \) ≤ 6

Handling Fall-Through

JMP table

Code Blocks (x == 1)

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

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

x86-64 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Stack "Bottom" contains lowest stack address
- Stack Pointer: rbp
- Address of "top" element
95
x86-64 Stack: Push
- Pushq Src
  - Fetch operand at Src
  - Decrement %rsp by 8
  - Write operand at address given by %rsp

x86-64 Stack: Pop
- Popq Dest
  - Read value at address given by %esp
  - Increment %rsp by 8
  - Store value at Dest (must be register)

Stack "Bottom"
Stack "Top"
Stack Pointer: %esp
Stack Grows Down

Increasing Addresses

96
Code Examples

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

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

Control Flow Example #1
- 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

Control Flow Example #2
- 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

Control Flow Example #3
- 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

Control Flow Example #4
- 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
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
- Stack discipline
  - State for given procedure needed for limited time
  - From when called to when return.
  - Caller returns before caller does
- Stack allocated in Frames
  - state for single procedure instantiation

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
    - "Teardown" code
    - Includes pop by return instruction

Example
```
Example Call Chain Example
```

Call Chain Example
```
Example
Call Chain Example
```

Stack
```
Stack
```

Procedure Stack Example
```
Procedure Stack Example
```

Contents
```
Contents
```

Return Information
```
Return Information
```

Stack Frame
```
Stack Frame
```

Frame Pointer:
```
Frame Pointer:
```

Stack Pointer:
```
Stack Pointer:
```

Stack "Top"
```
Stack "Top"
```

Example
```
Example
```

Example
```
Example
```

Frame Pointer:
```
Frame Pointer:
```

Stack Pointer:
```
Stack Pointer:
```

Stack "Top"
```
Stack "Top"
```

Example
```
Example
```

Example
```
Example
```

Frame Pointer:
```
Frame Pointer:
```

Stack Pointer:
```
Stack Pointer:
```

Stack "Top"
```
Stack "Top"
```

Example
```
Example
```

Example
```
Example
```

Frame Pointer:
```
Frame Pointer:
```

Stack Pointer:
```
Stack Pointer:
```

Stack "Top"
```
Stack "Top"
```
Example: Calling incr #3

Stack Structure

```c
long call_incr() {  
    long v1 = v123;  
    long v2 = incr(v1, 300);  
    return v2;  
}
```

```c
call_incr  
addq $16, trap  
movq $15213, $0(trap)  
movl $0x0, trap  
call incr  
addq $16, trap  
ret
```

Register Saving Conventions

- When procedure `yoo` calls `xoo`:
  - `yoo` is the caller
  - `xoo` is the callee

- Can register be used for temporary storage?
  - `xoo` saves temporary values in its frame before using
  - `xoo` restores them before returning to caller

- Contents of registers overwritten by `xoo`
  - `xoo` could be trouble == something should be done!
  - Need some coordination

x86-64 Linux Register Usage #2

- `rbx`, `rkl2`, `rkl3`, `rkl4`
  - Caller-saved
  - Caller must save & restore
- `rbp`
  - Caller-saved
  - Caller must save & restore
  - May be used as frame pointer
  - Can pass & return
- `rbx`
  - Special form of callee save
  - Restored to original value upon exit from procedure

Callee-Saved Example #1

Initial Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

Resulting Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

Example: Calling incr #4

Stack Structure

```
long call_incr() {  
    long v1 = v123;  
    long v2 = incr(v1, 300);  
    return v2;  
}
```

```c
call_incr  
addq $16, trap  
movq $15213, $0(trap)  
movl $0x0, trap  
call incr  
addq $16, trap  
ret
```

Example: Calling incr #5

Updated Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

Final Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

x86-64 Linux Register Usage #1

- `rbx`, `rkl1`, `rkl9`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `rkl10`
  - Caller-saved
  - Can be modified by procedure

Callee-Saved Example #2

Initial Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

Resulting Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```

Pre-return Stack Structure

```
rbx  
rkl2  
rkl3  
rkl4  
rbp
```
Today

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

Recursive Function

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

Recursive Function Register Save

```c
register:

- edx: x After address
- ebp: saved ebp
```

Recursive Function Call Setup

```c
register:

- eax: x + 1 Register
- ebx: Called-on
```

Recursive Function Result

```c
register:

- ebx: x + 1 Result
```

Recursive Function Completion

```c
register:

- eax: Return value
```

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 10)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Use in findOut
  - Also works for mutual recursion
    - P calls Q, Q calls P
**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 (and 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