Administtrivia

❖ Study Guide 1 due tonight

❖ Lab 2 due next Friday (2/5)

❖ hw10 due Monday, hw11 due Wednesday
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
Switch Statement

Example

- Multiple case labels
  - Here: 5 & 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

- Implemented with:
  - Jump table
  - Indirect jump instruction

```c
long switch_ex (long x, long y, long z)
{
    long w = 1;
    switch (x) {
        case 1:
            w = y*z; break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z; break;
        case 5:
        case 6:
            w -= z; break;
        case 7:
            w = y%z; break;
        default:
            w = 2;
    }
    return w;
}
```
Jump Table Structure

Switch Form

```java
switch (x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...
    case val_n-1:
        Block n-1
}
```

Approximate Translation

```java
target = JTab[x];
goto target;
```

Jump Table

- JTab:
  - Targ0
  - Targ1
  - Targ2
  - Targn-1

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targn-1: Code Block n-1
Jump Table Structure

C code:

```c
switch (x) {
    case 1: <code> break;
    case 2: <code>
    case 3: <code> break;
    case 5:
    case 6: <code> break;
    case 7: <code> break;
    default: <code>
}
```

Use the jump table when $x \leq 7$:

```c
if (x <= 7)
    target = JTab[x];
goto target;
else
    goto default;
```
Switch Statement Example

```c
long switch_ex(long x, long y, long z) {
    long w = 1;
    switch (x) {
        . . .
    }
    return w;
}
```

Note compiler chose to not initialize `w`

Take a look!
https://godbolt.org/z/Y9Kerb

Jump above – unsigned > catches negative default cases
Switch Statement Example

```c
long switch_ex(long x, long y, long z)
{
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

Jump table

```
.section .rodata
.align 8
.L4:
    .quad .L9  # x = 0
    .quad .L8  # x = 1
    .quad .L7  # x = 2
    .quad .L10 # x = 3
    .quad .L9  # x = 4
    .quad .L5  # x = 5
    .quad .L5  # x = 6
    .quad .L3  # x = 7
```

```
switch_ex:
    movq  %rdx, %rcx
    cmpq  $7, %rdi    # x:7
    ja    .L9         # default
    jmp   * .L4(,%rdi,8)  # jump table
```

Indirect jump

\[ D(R_s, R_s, s) = ((L_4 + x) * 8) \]
Assembly Setup Explanation

❖ Table Structure
  ▪ Each target requires 8 bytes (address)
  ▪ Base address at .L4

❖ Direct jump: jmp .L9
  ▪ Jump target is denoted by label .L9

❖ Indirect jump: jmp *.L4(%rdi,8)
  ▪ Start of jump table: .L4
  ▪ Must scale by factor of 8 (addresses are 8 bytes)
  ▪ Fetch target from effective address .L4 + x*8
    • Only for 0 ≤ x ≤ 7

Jump table

```
.section .rodata
.align 8
.L4:
quadr   .L9  # x = 0
.quadr   .L8  # x = 1
.quadr   .L7  # x = 2
.quadr   .L10 # x = 3
.quadr   .L9  # x = 4
.quadr   .L5  # x = 5
.quadr   .L5  # x = 6
.quadr   .L3  # x = 7
```
Roadmap

C:
```c
#include <malloc.h>

struct car {
    int miles;
    int gals;
};

int get_mpg(struct car *c) {
    // Implementation
}

int main() {
    struct car *c = malloc(sizeof(struct car));
    c->miles = 100;
    c->gals = 17;
    float mpg = get_mpg(c);
    free(c);
    return 0;
}
```

Java:
```java
public class Car {
    private int miles;
    private int gals;

    public Car() {
    }

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        // Implementation
        return 0;
    }
}
```

Assembly language:
```
get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret
```

Machine code:
```
011101000011000
100011010000010
1000100111000010
110000011111101
```

Computer system:
- Intel Core i5
- RAM
- SSD

Operating system:
- Windows 10
- OS X Yosemite

Memory & data
Integers & floats
x86 assembly

Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:
Reading Review

❖ Terminology:
  - Stack, Heap, Static Data, Literals, Code
  - Stack pointer (%rsp), push, pop
  - Caller, callee, return address, call, ret
    - Return value: %rax
    - Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Stack frames and stack discipline

❖ Questions from the Reading?
  - remember to post to Ed!
Review Questions

❖ How does the stack change after executing the following instructions?
  pushq %rbp
  subq $0x18, %rsp

❖ For the following function, which registers do we know must be used?
  void* memset(void* ptr, int value, size_t num);

rax ➔ execute instrs.
%rip ➔ procedure call
Mechanisms required for *procedures*

1) **Passing control**
   - To beginning of procedure code
   - Back to return point

2) **Passing data**
   - Procedure arguments
   - Return value

3) **Memory management**
   - Allocate during procedure execution
   - Deallocate upon return

❖ All implemented with machine instructions!
- An x86-64 procedure uses only those mechanisms required for that procedure

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

P(...) {
    
    y = Q(x);
    print(y)
    
}
```
Procedures

❖ **Stack Structure**

❖ **Calling Conventions**
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ **Register Saving Conventions**

❖ **Illustration of Recursion**
Simplified Memory Layout

Address Space:

- **Stack**: Local variables and procedure context
- **Dynamic Data (Heap)**: Variables allocated with `new` or `malloc`
- **Static Data**: Static variables (including global variables)
- **Literals**: Large literals/ constants (e.g., "example")
- **Instructions**: Program code

High Addresses

Low Addresses

Memory Addresses
Memory Management

Address Space:

- Stack
  - Managed “automatically” (by compiler/assembly)

- Dynamic Data (Heap)
  - Managed “dynamically” (by programmer)

- Static Data
  - Managed “statically” (initialized when process starts)

- Literals
  - Managed “statically” (initialized when process starts)

- Instructions
  - Managed “statically” (initialized when process starts)

Who’s Responsible:

- High Addresses
- Memory Addresses
- Low Addresses

0xF...F

0x0...0
Memory Permissions

- **Segmentation fault**: impermissible memory access

- **Address Space**:
  - **Stack**: writable; not executable
  - **Dynamic Data (Heap)**: writable; not executable
  - **Static Data**: writable; not executable
  - **Literals**: read-only; not executable
  - **Instructions**: read-only; executable
x86-64 Stack

- Region of memory managed with stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”
  - LIFO (Last In, First Out)

- Register %rsp contains lowest stack address
  - %rsp = address of top element, the most-recently-pushed item that is not-yet-popped

![Stack Diagram]

- Stack Pointer: %rsp
  - Stack “Bottom”
  - Stack “Top”
  - High Addresses: Increasing Addresses
  - Stack Grows Down
  - Low Addresses: 0x00...00
x86-64 Stack: Push

- pushq src
  - Fetch operand at src
    - Src can be reg, memory, immediate
  - **Decrement** %rsp by 8
  - Store value at address given by %rsp

**Example:**
- pushq %rcx
  - Adjust %rsp and store contents of %rcx on the stack

**Stack Pointer:** %rsp

- Stack “Bottom”
- Stack “Top”
- High Addresses
  - Increasing Addresses
  - Stack Grows Down
  - Low Addresses
    - 0x00...00
x86-64 Stack: Pop

- `popq dst`
  - Load value at address given by `%rsp`
  - Store value at `dst`
  - **Increment** `%rsp` by 8

**Example:**
- `popq %rcx`
  - Stores contents of top of stack into `%rcx` and adjust `%rsp`

Those bits are still there; we’re just not using them.
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Procedure Call Overview

- **Callee** must know where to find args
- **Callee** must know where to find *return address*
- **Caller** must know where to find *return value*
- **Caller** and **Callee** run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (*e.g.*, no arguments)
Procedure Call Overview

- The convention of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn’t follow these conventions?
Code Example (Preview)

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

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

Compiler Explorer:
https://godbolt.org/z/ndro9E

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

```
0000000000400550 <mult2>:
400550: movq %rdi,%rax  # a
400553: imulq %rsi,%rax  # a * b
400557: ret  # Return
```
Procedure Control Flow

- Use stack to support procedure call and return

Procedure call: `[call] label`

1) Push return address on stack *(why? which address?)*
2) Jump to *label*
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call:** `call label`
  1) Push return address on stack (*why? which address?*)
  2) Jump to `label`
- **Return address:**
  - Address of instruction immediately after `call` instruction
  - Example from disassembly:
    
    ```
    400544: call 400550 <mult2>
    400549: movq %rax,(%rbx)
    ```
    
    Return address = `0x400549`

- **Procedure return:** `ret`
  1) Pop return address from stack
  2) Jump to address
Procedure Call Example (step 1)

0000000000400540 <multstore>:
  
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)

0000000000400550 <mult2>:
  
  400550: movq %rdi, %rax
  
  400557: ret
Procedure Call Example (step 2)

0000000000400540 <multstore>:
  .
  .
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  .
  .

0000000000400550 <mult2>:
  400550: movq %rdi, %rax
  .
  .
  400557: ret
Procedure Return Example (step 1)

00000000000400540 <multstore>:
  •
  •
  400544: call 400550 <mult2>
  400549: movq %rax, (%rbx)
  •
  •

00000000000400550 <mult2>:
  400550: movq %rdi, %rax
  •
  •
  400557: ret

%rip 0x400557
%rsp 0x118

0x130
0x128
0x120
0x118
0x400549
Procedure Return Example (step 2)

0000000000400540 <multstore>:
  •
  400544: call 400550 <mult2>
  400549: movq %rax,(%rbx)
  •
  •

0000000000400550 <mult2>:
  400550: movq %rdi,%rax
  •
  •
  400557: ret

%rsp 0x120
%rip 0x400549
0x130
0x128
0x120
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Procedure Data Flow

Registers (NOT in Memory)

- First 6 arguments
  - %rdi
  - %rsi
  - %rdx
  - %rcx
  - %r8
  - %r9

- Return value
  - %rax

Stack (Memory)

- Only allocate stack space when needed

Diane’s Silk Dress Costs $89
x86-64 Return Values

- By convention, values returned by procedures are placed in %rax
  - Choice of %rax is arbitrary

1) Caller must make sure to save the contents of %rax before calling a callee that returns a value
  - Part of register-saving convention

2) Callee places return value into %rax
  - Any type that can fit in 8 bytes – integer, float, pointer, etc.
  - For return values greater than 8 bytes, best to return a pointer to them

3) Upon return, caller finds the return value in %rax
Data Flow Examples

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

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

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

```assembly
0000000000400550 <mult2>:  
# a in %rdi, b in %rsi
400550: movq %rdi,%rax       # a
400553: imulq %rsi,%rax      # a * b
# s in %rax
400557: ret                 # Return
```
Procedures

❖ Stack Structure

❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data

❖ Register Saving Conventions

❖ Illustration of Recursion
Stack-Based Languages

❖ Languages that support recursion
  ▪ *e.g.*, C, Java, most modern languages
  ▪ Code must be *re-entrant*
    • Multiple simultaneous instantiations of single procedure
  ▪ Need some place to store *state* of each instantiation
    • Arguments, local variables, return address

❖ Stack allocated in *frames*
  ▪ State for a single procedure instantiation

❖ Stack discipline
  ▪ State for a given procedure needed for a limited time
    • Starting from when it is called to when it returns
  ▪ Callee always returns before caller does
Call Chain Example

Procedure `amI` is recursive (calls itself)
1) Call to `who`

```
whoa(...) {
  •
  •
  who();
  •
}
```

Stack

```
whoa %rbp
who %rsp
ami ami
ami
ami
```
2) Call to who

```c
whoa(…)
{
    who(…)
    {
        •
        amI();
        •
        amI();
    }
}
```

Stack

```
whoa
%rbp

who
%rsp

amI

amI
```
3) Call to amI (1)

```c
whoa(...)
{
    who(...)
    {
        amI(...)
        {
            .
            if(){
                amI()}
        }
        .
    }
}
```

Stack

```
whoa

who

amI

amI

%rbp

%rsp

amI (1)
```
### 4) Recursive call to `amI (2)`

```plaintext
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        if()
        {
          amI()
        }
        amI()
      }
    }
  }
}
```

![Diagram of recursive calls and stack movement]

- `whoa` function calls `who` function, which in turn calls `amI` function recursively.
- The stack frame for each function call is shown with `%rbp` and `%rsp` indicating the stack pointer and base pointer, respectively.
- The stack grows downward, with each recursive call pushing new frames onto the stack.

The stack frame details include:
- `whoa` function at the top.
- `who` function in the middle.
- `amI` function frames below, showing the recursive calls.

The stack frame for `amI` shows:
- `%rbp` and `%rsp` indicating the current stack frame.
- Each recursive call adds a new frame to the stack, with the parameters and local variables for each function.

This diagram illustrates the recursive nature of the `amI` function and how the stack is used to manage the function calls and return values.
5) (another) **Recursive call to amI (3)**

```plaintext
whoa(…)
{
  who(…)
  {
    amI(…)
    {
      amI(…)
      {
        amI(…)
        {
          amI(…)
          {
            amI(…)
            {
              amI(…)
              {
                if(){
                  amI()
                  amI()
                  amI()
                }
              }
            }
          }
        }
      }
    }
  }
}
```

**Stack**

- whoa
- who
- amI
- amI
- amI
- amI
- amI
- %rbp
- %rsp
6) Return from (another) recursive call to amI

Stack

whoa

who

amI

amI

%rbp

%rsp
7) Return from recursive call to \texttt{amI}

[Diagram showing recursive calls and stack frames]
8) Return from call to `amI`

```c
whoa(...) {
  who(...) {
    • amI();
    • amI();
  }
}
```

The stack diagram shows the activation records for the calls to `whoa` and `amI`. The stack frame for `amI` is popped from the stack as the return from the call to `amI` is made. The stack is then ready for the next call or return.
9) (second) Call to `amI (4)`
10) Return from (second) call to amI
11) Return from call to `who`

```c
whoa(...) {
    ...
    who();
    ...
}
```

Stack Diagram:
- `whoa`
- `who`
- `amI` (Repeats)
- `%rbp`
- `%rsp`
- `amI_4`
- `amI_2`
- `amI_3`