Princeton University
Computer Science 217: Introduction to Programming Systems

Assembly Language: Function Calls

1

1

Goals of this Lecture

Help you learn:

- Function call problems
- AARCH64 solutions
  - Pertinent instructions and conventions

2

2

Function Call Problems

(1) Calling and returning
- How does caller function **jump** to callee function?
- How does callee function **jump back** to the right place in caller function?

(2) Passing arguments
- How does caller function pass **arguments** to callee function?

(3) Storing local variables
- Where does callee function store its **local variables**?

(4) Returning a value
- How does callee function send **return value** back to caller function?
- How does caller function access the **return value**?

(5) Optimization
- How do caller and callee function minimize memory access?

3

3

Running Example

```
long absadd(long a, long b)
{
   long absA, absB, sum;
   absA = labs(a);
   absB = labs(b);
   sum = absA + absB;
   return sum;
}
```

Calls standard C `labs()` function
- Returns absolute value of given `long`

4

4

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Optimization

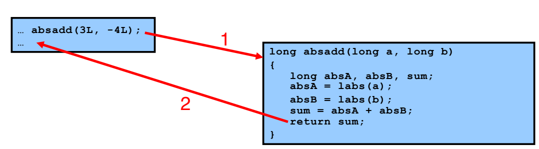
5

5

Problem 1: Calling and Returning

How does caller *jump* to callee?
- i.e., Jump to the address of the callee's first instruction

How does the callee *jump back* to the right place in caller?
- i.e., Jump to the instruction immediately following the most-recently-executed call instruction



6

6

Attempted Solution: b Instruction

Attempted solution: caller and callee use
b (unconditional branch) instruction

```
f:
  …
  b g      # Call g
fReturnPoint:
  …
```

```
g:
  …
  b fReturnPoint     # Return
```

7

7

Attempted Solution: b Instruction

Problem: callee may be called by multiple callers

```
f1:
  …
  b g      # Call g
f1ReturnPoint:
  …
```

```
f2:
  …
  b g      # Call g
f2ReturnPoint:
  …
```

```
g:
  …
  b ???      # Return
```

8

8

Partial Solution: Use Register

bl (branch and link) instruction stores return point in X30
ret (return) instruction returns to address in X30

```
f1:
  bl g        # Call g
f1ReturnPoint:
  …
```

```
f2:
  bl g        # Call g
f2ReturnPoint:
  …
```

```
g:
  …
  ret       # Return
```

Correctly returns
to either f1 or f2

9

9

Partial Solution: Use Register

Problem: Cannot handle nested function calls

```
f:
  bl g          # Call g
  ...
```

Problem if **f()** calls **g()**,
and **g()** calls **h()**

Return address **g()** → **f()**
is lost

```
g:
  bl h          # Call h
  ...
  ret           # Return
```

```
h:
  ...
  ret          # Return
```

10

10

Rest of Solution: Use the Stack

Observations:

- May need to store many return addresses
  - The number of nested function calls is not known in advance
  - A return address must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored return addresses are destroyed in reverse order of creation
  - **f()** calls **g()** ⇒ return addr for **g** is stored
  - **g()** calls **h()** ⇒ return addr for **h** is stored
  - **h()** returns to **g()** ⇒ return addr for **h** is destroyed
  - **g()** returns to **f()** ⇒ return addr for **g** is destroyed
- LIFO data structure (stack) is appropriate

↓
addr for h
addr for g
addr for f

AARCH64 solution:

- Use the STACK section of memory, usually accessed via SP

11

11

Saving Link (Return) Addresses

Push X30 on stack when entering a function
Pop X30 from stack before returning from a function

```
f:
  // Save X30
  ...
  bl g  # Call g
  ...
  // Restore X30
  ret
```

```
g:
  // Save X30
  ...
  bl h  # Call h
  ...
  // Restore X30
  ret
```

```
h:
  ...
  ret
```

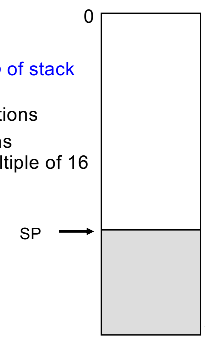
12

12

Stack Operations

SP (stack pointer) register points to *top* of stack

- Can be used in `ldr` and `str` instructions
- Can be used in arithmetic instructions
- AARCH64 requirement: must be multiple of 16



13

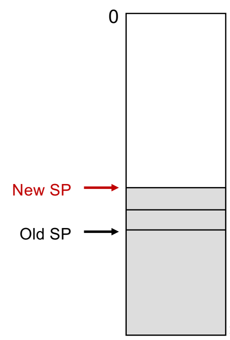
13

Stack Operations

To create a new *stack frame*:

- Decrement `sp`

```
sub sp, sp, 16
```



14

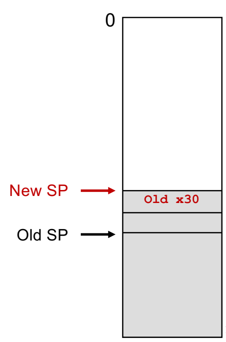
14

Stack Operations

To use the *stack frame*:

- Load/store *at* or *offset from* `sp`

```
str x30, [sp]
…
ldr x30, [sp]
```



15

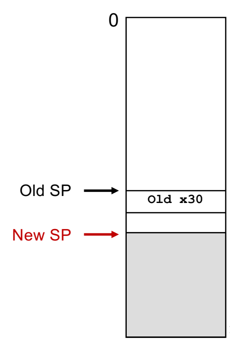
15

Stack Operations

To delete the *stack frame*:

- Increment `sp`

```
add sp, sp, 16
```



16

16

Saving Link (Return) Addresses

Push X30 on stack when entering a function
Pop X30 from stack before returning from a function

```
f:
    // Save X30
    sub sp, sp, 16
    str x30, [sp]
    ...
    bl g  # Call g
    ...
    // Restore X30
    ldr x30, [sp]
    add sp, sp, 16
    ret
```

```
g:
    // Save X30
    sub sp, sp, 16
    str x30, [sp]
    ...
    bl h  # Call h
    ...
    // Restore X30
    ldr x30, [sp]
    add sp, sp, 16
    ret
```

```
h:
    ...
    ret
```

17

17

Running Example

```
// long absadd(long a, long b)
absadd:
    sub sp, sp, 16
    str x30, [sp]
    // long absA, absB, sum
    ...
    // absA = labs(a)
    ...
    bl labs
    ...
    // absB = labs(b)
    ...
    bl labs
    ...
    // sum = absA + absB
    ...
    // return sum
    ...
    ldr x30, [sp]
    add sp, sp, 16
    ret
```

18

18

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Optimization

19

19

Problem 2: Passing Arguments

Problem:

- How does caller pass *arguments* to callee?
- How does callee accept *parameters* from caller?

```
long absadd(long a, long b)
{
   long absA, absB, sum;
   absA = labs(a);
   absB = labs(b);
   sum = absA + absB;
   return sum;
}
```

20

20

ARM Solution 1: Use the Stack

Observations (déjà vu):

- May need to store many arg sets
  - The number of arg sets is not known in advance
  - If this function calls any others, arg set *must be saved* for as long as the invocation of this function is live, and discarded thereafter
- Stored arg sets are destroyed in reverse order of creation
- LIFO data structure (stack) is appropriate

21

21

ARM Solution 2: Use Registers

AARCH64 solution:

- Pass first 8 (integer or address) arguments in registers for efficiency
  - X0..X7 and/or W0..W7
- More than 8 arguments ⇒
  - Pass arguments 9, 10, … on the stack
  - (Beyond scope of COS 217)
- Arguments are structures ⇒
  - Pass arguments on the stack
  - (Beyond scope of COS 217)

Callee function then saves arguments to stack

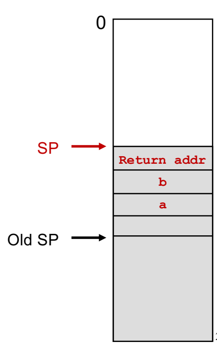
- Or maybe not!
  - See "optimization" later this lecture
- Callee accesses arguments as positive offsets vs. SP

22

22

Running Example

```
// long absadd(long a, long b)
absadd:
   sub sp, sp, 32
   str x30, [sp]         // Save x30
   str x0, [sp, 16]      // Save a
   str x1, [sp, 8]       // Save b
   // long absA, absB, sum
   ...
   // absA = labs(a)
   ldr x0, [sp, 16]      // Load a
   bl labs
   ...
   // absB = labs(b)
   ldr x0, [sp, 8]       // Load b
   bl labs
   ...
   // sum = absA + absB
   ...
   // return sum
   ...
   ldr x30, [sp]         // Restore x30
   add sp, sp, 32
   ret
```



23

23

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Optimization

24

24

Problem 3: Storing Local Variables

Where does callee function store its *local variables?*

```
long absadd(long a, long b)
{
   long absA, absB, sum;
   absA = labs(a);
   absB = labs(b);
   sum = absA + absB;
   return sum;
}
```

25

25

ARM Solution: Use the Stack

Observations (déjà vu again!):
- May need to store many local var sets
  - The number of local var sets is not known in advance
  - Local var set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored local var sets are destroyed in reverse order of creation
- LIFO data structure (stack) is appropriate

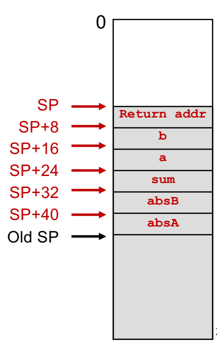
AARCH64 solution:
- Use the STACK section of memory
- Or maybe not!
  - See later this lecture

26

26

Running Example

```
// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, 48
   str x30, [sp]      // Save x30
   str x0, [sp, 16]   // Save a
   str x1, [sp, 8]    // Save b
   // absA = labs(a)
   ldr x0, [sp, 16]   // Load a
   bl labs
   ...
   // absB = labs(b)
   ldr x0, [sp, 8]    // Load b
   bl labs
   ...
   // sum = absA + absB
   ldr x0, [sp, 40]   // Load absA
   ldr x1, [sp, 32]   // Load absB
   add x0, x0, x1
   str x0, [sp, 24]   // Store sum
   // return sum
   ...
   ldr x30, [sp]      // Restore x30
   add sp, sp, 48
   ret
```



27

27

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Optimization

28

28

Problem 4: Return Values

Problem:
- How does callee function send return value back to caller function?
- How does caller function access return value?

```
long absadd(long a, long b)
{
   long absA, absB, sum;
   absA = labs(a);
   absB = labs(b);
   sum = absA + absB;
   return sum;
}
```

29

29

ARM Solution: Use X0 / W0

In principle
- Store return value in stack frame of caller

Or, for efficiency
- Known small size ⇒ store return value in register
- Other ⇒ store return value in stack

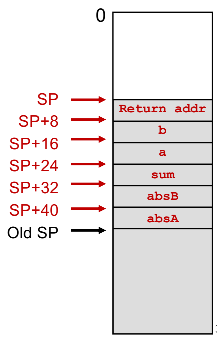
AARCH64 convention
- Integer or address:
  - Store return value in X0 / W0
- Floating-point number:
  - Store return value in floating-point register
  - (Beyond scope of COS 217)
- Structure:
  - Store return value in memory pointed to by X8
  - (Beyond scope of COS 217)

30

30

Running Example

```
// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, 48
   str x30, [sp]      // Save x30
   str x0, [sp, 16]   // Save a
   str x1, [sp, 8]    // Save b
   // absA = labs(a)
   ldr x0, [sp, 16]   // Load a
   bl labs
   str x0, [sp, 40]   // Store absA
   // absB = labs(b)
   ldr x0, [sp, 8]    // Load b
   bl labs
   str x0, [sp, 32]   // Store absB
   // sum = absA + absB
   ldr x0, [sp, 40]   // Load absA
   ldr x1, [sp, 32]   // Load absB
   add x0, x0, x1
   str x0, [sp, 24]   // Store sum
   // return sum
   ldr x0, [sp, 24]   // Load sum
   ldr x30, [sp]      // Restore x30
   add sp, sp, 48
   ret
```



31

31

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Optimization

32

32

Problem 5: Optimization

Observation: Accessing memory is expensive
- More expensive than accessing registers
- For efficiency, want to store parameters and local variables in registers (and not in memory) when possible

Observation: Registers are a finite resource
- In principle: Each function should have its own registers
- In reality: All functions share same small set of registers

Problem: How do caller and callee use same set of registers without interference?
- Callee may use register that the caller also is using
- When callee returns control to caller, old register contents may have been lost
- Caller function cannot continue where it left off

33

33

ARM Solution: Register Conventions

Callee-save registers
- X19..X29 (or W19..W29)
- Callee function ***must preserve*** contents
- If necessary…
  - Callee saves to stack near beginning
  - Callee restores from stack near end

Caller-save registers
- X8..X18 (or W8..W18) – plus parameters in X0..X7
- Callee function ***can change*** contents
- If necessary…
  - Caller saves to stack before call
  - Caller restores from stack after call

34

34

Running Example

Parameter handling in *unoptimized* version:
- **absadd()** accepts parameters (**a** and **b**) in X0 and X1
- At beginning**,** **absadd()** copies contents of X0 and X1 to stack
- Body of **absadd()** uses stack
- At end, **absadd()** pops parameters from stack

Parameter handling in *optimized* version:
- **absadd()** accepts parameters (**a** and **b**) in X0 and X1
- At beginning**,** copies contents of X0 and X1 to X19 and X20
- Body of **absadd()** uses X19 and X20
- Must be careful:
  - **absadd()** cannot change contents of X19 and X20
  - So **absadd()** must save X19 and X20 near beginning, and restore near end

35

35

Running Example

Local variable handling in *unoptimized* version:
- At beginning, **absadd()** allocates space for local variables (**absA**, **absB**, **sum**) on stack
- Body of **absadd()** uses stack
- At end, **absadd()** pops local variables from stack

Local variable handling in *optimized* version:
- **absadd()** keeps local variables in X21, X22, X23
- Body of **absadd()** uses X21, X22, X23
- Must be careful:
  - **absadd()** cannot change contents of X21, X22, or X23
  - So **absadd()** must save X21, X22, and X23 near beginning, and restore near end

36

36

Running Example

```
// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, 48
   str x30, [sp]      // Save x30
   str x19, [sp, 8]   // Save x19, use for a
   str x20, [sp, 16]  // Save x20, use for b
   str x21, [sp, 24]  // Save x21, use for absA
   str x22, [sp, 32]  // Save x22, use for absB
   str x23, [sp, 40]  // Save x23, use for sum
   mov x19, x0        // Store a in x19
   mov x20, x1        // Store b in x20
   // absA = labs(a)
   mov x0, x19        // Load a
   bl labs
   mov x21, x0        // Save absA
   // absB = labs(b)
   mov x0, x20        // Load b
   bl labs
   mov x22, x0        // Store absB
   // sum = absA + absB
   add x23, x21, x22
   // return sum
   mov x0, x23        // Load sum
   ldr x30, [sp]      // Restore x30
   ldr x19, [sp, 8]   // Restore x19
   ldr x20, [sp, 16]  // Restore x20
   ldr x21, [sp, 24]  // Restore x21
   ldr x22, [sp, 32]  // Restore x22
   ldr x23, [sp, 40]  // Restore x22
   add sp, sp, 48
   ret
```

`absadd()` stores parameters and local vars in X19..X23, not in memory

`absadd()` cannot destroy contents of X19..X23

So `absadd()` must save X19..X23 near beginning and restore near end

37

37

Eliminating Redundant Copies

```
// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, 32
   str x30, [sp]      // Save x30
   str x19, [sp, 8]   // Save x19, use for b
   str x20, [sp, 16]  // Save x20, use for absA
   mov x19, x1        // Store b in x19
   // absA = labs(a)
   bl labs            // a already in x0
   mov x20, x0        // Save absA
   // absB = labs(b)
   mov x0, x19        // Load b
   bl labs
   // sum = absA + absB
   add x0, x20, x0    // x0 held absB, now holds sum
   // return sum - already in x0
   ldr x30, [sp]      // Restore x30
   ldr x19, [sp, 8]   // Restore x19
   ldr x20, [sp, 16]  // Restore x20
   add sp, sp, 32
   ret
```

Further optimization: remove redundant moves between registers

- "Hybrid" pattern that uses both caller- and callee-saved registers
- Can be confusing: no longer systematic mapping between variables and registers
- Attempt only *after* you have working code!
- Save working versions for easy comparison!

38

38

Non-Optimized vs. Optimized Patterns

Unoptimized pattern

- Parameters and local variables strictly in memory (stack) during function execution
- **Pro**: Always possible
- **Con**: Inefficient
- gcc compiler uses when invoked *without* –O option

Optimized pattern

- Parameters and local variables mostly in registers during function execution
- **Pro**: Efficient
- **Con**: Sometimes impossible
  - Too many local variables
  - Local variable is a structure or array
  - Function computes address of parameter or local variable
- gcc compiler uses when invoked *with* –O option, when it can!

39

39

Writing Readable Code

```
// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, 48
   str x30, [sp]
   str x19, [sp, 8]
   str x20, [sp, 16]
   str x21, [sp, 24]
   str x22, [sp, 32]
   str x23, [sp, 40]
   mov x19, x0
   mov x20, x1
   // absA = labs(a)
   mov x0, x19        // Load a
   bl labs
   mov x21, x0        // Save absA
   // absB = labs(b)
   mov x0, x20        // Load b
   bl labs
   mov x22, x0        // Store absB
   // sum = absA + absB
   add x23, x21, x22
   // return sum
   mov x0, x23        // Load sum
   ldr x30, [sp]      // Restore x30
   ldr x19, [sp, 8]   // Restore x19
   ldr x20, [sp, 16]  // Restore x20
   ldr x21, [sp, 24]  // Restore x21
   ldr x22, [sp, 32]  // Restore x22
   ldr x23, [sp, 40]  // Restore x22
   add sp, sp, 48
   ret
```

Problem

- Hardcoded sizes, offsets, registers are difficult to read, understand, debug

40

40

Writing Readable Code

```
   // Stack frame size in bytes
   .equ STACKSIZE, 48
   // Registers for parameters
   a    .req x19
   b    .req x20
   // Registers for local variables
   absA .req x21
   absB .req x22
   sum  .req x23

// long absadd(long a, long b)
absadd:
   // long absA, absB, sum
   sub sp, sp, STACKSIZE
   str x30, [sp]      // Save x30
   str x19, [sp, 8]   // Save x19
   str x20, [sp, 16]  // Save x20
   str x21, [sp, 24]  // Save x21
   str x22, [sp, 32]  // Save x22
   str x23, [sp, 40]  // Save x23
   mov a, x0          // Store a in x19
   mov b, x1          // Store b in x20

   ...
```

Problem

- Hardcoded sizes, offsets, registers are difficult to read, understand, debug

Using .equ and .req

- To define a symbolic name for a constant:
  `.equ SOMENAME, nnn`
- To define a symbolic name for a register (e.g. what variable it holds):
  `SOMENAME .req Xnn`

41

41

Writing Readable Code

```
   ...

   // absA = labs(a)
   mov x0, a
   bl labs
   mov absA, x0
   // absB = labs(b)
   mov x0, b
   bl labs
   mov absB, x0
   // sum = absA + absB
   add sum, absA, absB
   // return sum
   mov x0, sum
   ldr x30, [sp]      // Restore x30
   ldr x19, [sp, 8]   // Restore x19
   ldr x20, [sp, 16]  // Restore x20
   ldr x21, [sp, 24]  // Restore x21
   ldr x22, [sp, 32]  // Restore x22
   ldr x23, [sp, 40]  // Restore x23
   add sp, sp, STACKSIZE
   ret
```

Problem

- Hardcoded sizes, offsets, registers are difficult to read, understand, debug

Using .equ and .req

- To define a symbolic name for a constant:
  `.equ SOMENAME, nnn`
- To define a symbolic name for a register (e.g. what variable it holds):
  `SOMENAME .req Xnn`

42

42

Summary

Function calls in AARCH64 assembly language

Calling and returning

- `bl` instruction saves return address in X30 and jumps
- `ret` instruction jumps back to address in X30

Passing arguments

- Caller copies args to caller-saved registers (in prescribed order)
- Unoptimized pattern:
  - Callee pushes args to stack
  - Callee uses args as positive offsets from SP
  - Callee pops args from stack
- Optimized pattern:
  - Callee keeps args in caller-saved registers
  - Be careful!

43

43

Summary (cont.)

Storing local variables

- Unoptimized pattern:
  - Callee pushes local vars onto stack
  - Callee uses local vars as positive offsets from SP
  - Callee pops local vars from stack
- Optimized pattern:
  - Callee keeps local vars in callee-saved registers

Returning values

- Callee places return value in X0
- Caller accesses return value in X0

44

44