COS 217: Introduction to Programming Systems

Assembly Language

Local Variables and Function Calls

PRINCETON UNIVERSITY

This Lecture

We've learned about how the following are done in assembly:

- Arithmetic and logic operations
- Data Structures (Arrays)
- Control flow: GOTO a new location and resume executing there

Function calls are more complex:

- Have to go and come back: more complex control flow
- Functions can call other functions before they return: how to manage state

Goal: Learn how function calls are implemented in AARCH64

- Learn how indirection, abstraction, and data structures are used

Problems to Solve in Function Calls

(1) Control flow: 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?

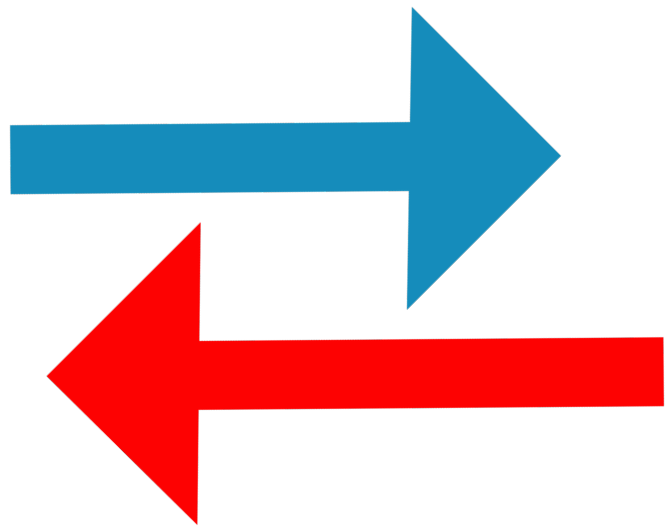
(5) Optimization

- How do caller and callee function minimize memory access?

Running Example

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

`absadd()` calls C `labs()` function, which returns absolute value of given long



CALLING AND RETURNING

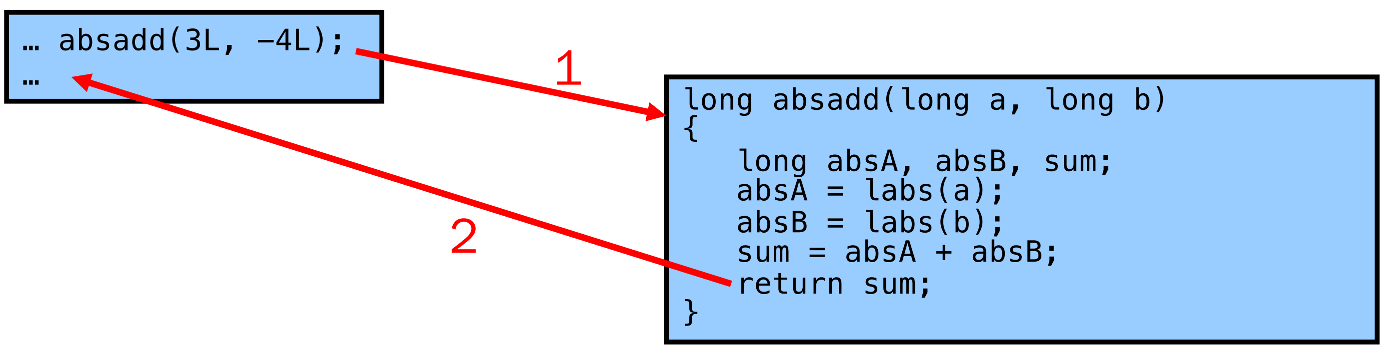
Problem 1: Calling and Returning

How does caller call the callee?

i.e., jump to (goto) the address of the callee’s first instruction

How does callee get back to the right place in the caller?

i.e., jump to the instruction immediately following the most-recently-executed call



iClicker Question

Q: Based on last lecture, what instructions would we use to “jump” into and back out of the callee?

```
… absadd(3L, -4L);
…
```

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

A. 2 conditional branches

B. 1 conditional branch, then 1 unconditional branch

C. 1 unconditional branch, then 1 conditional branch

D. 2 unconditional branches

E. Something more complicated

Attempted Solution: b Instruction

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

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

```
g:
   …
   b fReturnPoint    // Return
```

Attempted Solution: b Instruction

Problem: callee may be called by multiple callers

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

```
g:
   …
   b ???      // Return
```

```
h:
   …
   b g      // Call g
hReturnPoint:
   …
```

Partial Solution: Indirection

Put return address in a register. Value changes based on where it's called from

`br` (branch register) instruction branches to address in X register operand

```
f1:

   adr x30, f1ReturnPoint
   b g          // Call g

f1ReturnPoint:

   …
```

```
f2:

   adr x30, f2ReturnPoint
   b g          // Call g

f2ReturnPoint:

   …
```

```
g:

   …
  br x30     // Return
```

Correctly returns to either `f1` or `f2`

`adr` instruction before every function call

Label after every function call

Caller and callee must use same register

Partial Solution: Abstraction with Auto-register (X30)

`bl` (branch and link) instruction stores return point in X30
`ret` (return) instruction returns to address in X30, and it's always X30 (in hardware)

```
f1:
   bl g        // Call g
   …
```

```
f2:
   bl g        // Call g
   …
```

```
g:
   …
   ret         // Return
```

Correctly returns to either `f1` or `f2`

Takes care of the three things on previous slide under the hood

Aside: so `ret` is identical to `b x30`, right? Yes and no …
https://www.mattkeeter.com/blog/2023-01-25-branch/

Not quite there yet ...

Problem: Cannot handle nested function calls

```
f:
   bl g        // Call g
   ...         // location 1
```

Problem if `f()` calls `g()`
then `g()` calls `h()`

Return address `g()` → `f()` is lost
`g()` returns to the middle of `g()` ...

```
g:
   bl h        // Call h
   ...         // location 2
   ret         // Return
```

```
h:
   ... ... ...
   ret         // Return
```

How Do We Keep Track of Calls and Returns?

Observations:

- Every function call needs **separate** tracking of a return address
  - We may need to store many return addresses
  - The number of nested function calls is not known in advance
  - Return address must be saved for as long as invocation of that function is live, then discarded
- *Key: Stored return addresses are discarded in reverse order of creation*
  - `f()` calls `g()` ⇒ return addr for `f` is stored
  - `g()` calls `h()` ⇒ return addr for `g` is stored
  - `h()` returns to `g()` ⇒ return addr for `g` is discarded
  - `g()` returns to `f()` ⇒ return addr for `f` is discarded

↓
`addr for h`
`addr for g`
`addr for f`

- *This is true for any call-return pattern*. So LIFO data structure (***stack***) is appropriate

AARCH64 solution:

- Use the STACK section of memory, usually accessed via SP
- We know about stackframes to manage temp space for functions. Put return addr (X30) there too

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
```

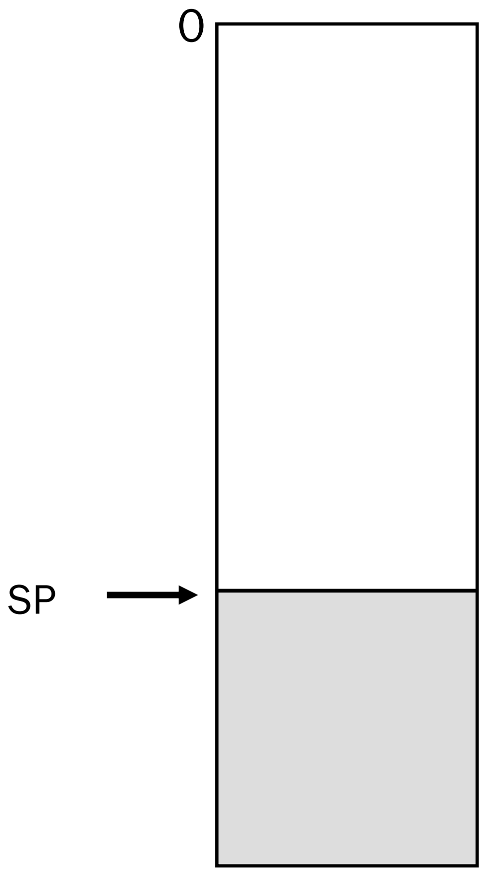
```
h:
   ...
   ret
```

Stack Operations

SP (stack pointer) register points to top of stack

- Points to beginning of active stack frame, i.e. stack frame for currently active function
- If active function calls a function, latter's stack frame will be pushed above this point and SP moved to its top
- Etc.

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

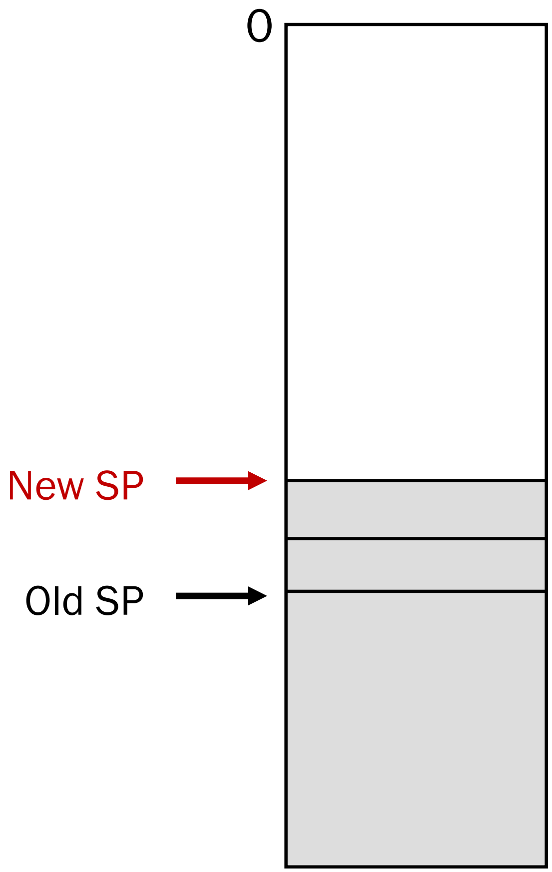


Stack Operations

To create (push) a new stack frame:

- Decrement `sp`
  `sub sp, sp, 16`

- 16 is the minimum size allowed, even for just the return value of 8 bytes

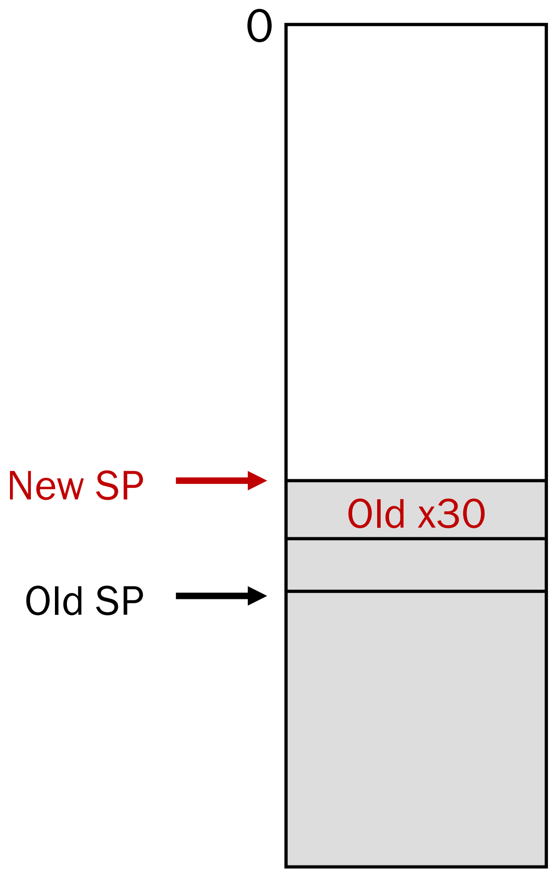


Stack Operations

Stack is in memory, so can load/store information from and to it:

- Load/store X30 value at or offset from `sp`

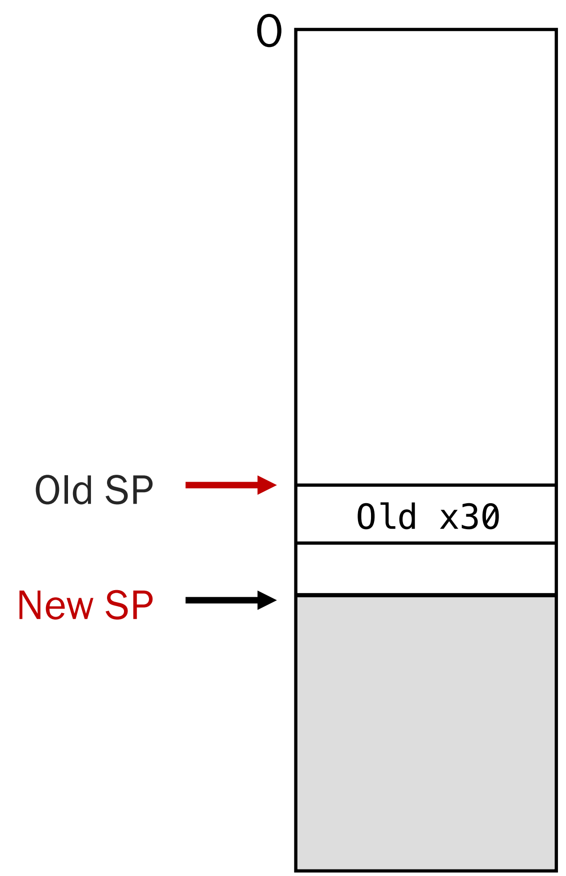
```
str x30, [sp]
...
ldr x30, [sp]
```



Stack Operations

To delete (pop) the current active stack frame:

- Increment `sp`
  ```
  add sp, sp, 16
  ```
- Assumes size of frame is 16 bytes
- Need to restore value of X30 from the stack to the X30 register, since that may be used for function calls and value may have changed, so it's not lost when stack frame goes away
- (Note: data above New SP are not erased, just stay there. But not program data anymore.)



Function Call Prolog and Epilog

Prolog: Make room on stack; Push X30 on stack. Then execute function code
Epilog: Pop X30 from stack; tear down stack frame. Then returning from 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
```

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
```

PASSING ARGUMENTS

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;
}
```

ARM Solution 1: Use the Stack

A realization of our understanding from C that these go on the stack

Observations (déjà vu):

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

ARM Solution 2: Use Registers

AARCH64 solution:

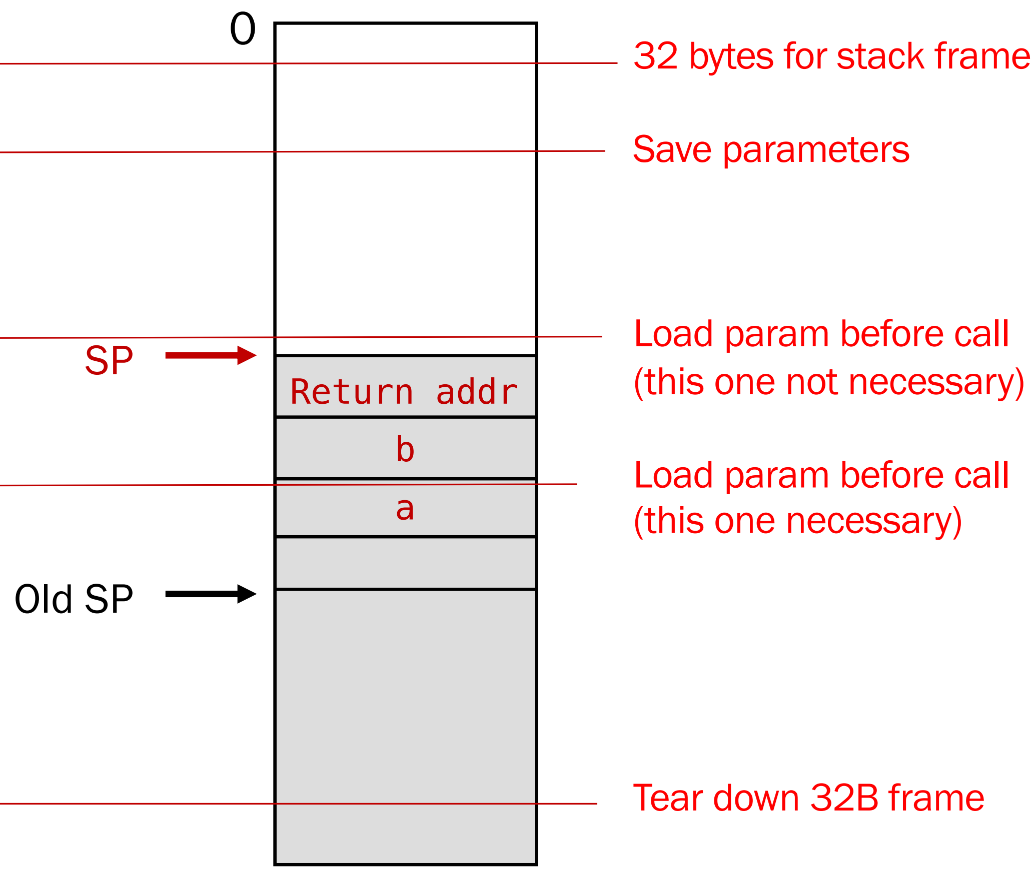
- Pass first 8 (integer or address) arguments in registers for efficiency
  - X0..X7 and/or W0..W7
- More than 8 arguments ⇒ Pass arguments 9, 10, … on the stack
  - (Beyond scope of COS 217)
- Arguments are structures ⇒ Pass arguments on the stack
  - (Beyond scope of COS 217)

Callee function then may save arguments to stack

- E.g. function calls it makes will use X0..X7 the same way and destroy values; or its code may
- Then, we can reference arguments as positive offsets from stack, like with return value at `sp`
- Or maybe not (See “optimization” later this lecture)

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
```



STORING LOCAL VARIABLES

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;
}
```

In C, we say they're on the stack. But where are they really in ARM?

ARM Solution: Use the Stack

Observations (this is getting repetitive ...):

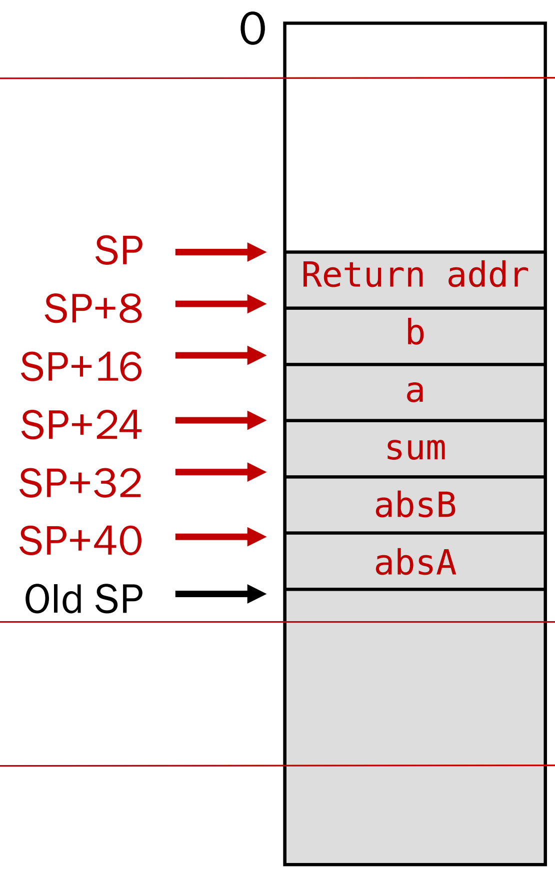
- May need to store many local variable sets
  - The number of local variable sets is not known in advance
  - Local variable sets must be saved
    for as long as the invocation of this function is live, and discarded thereafter
- Stored local variable sets are destroyed in reverse order of creation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

AARCH64 solution:

- Use the STACK section of memory
- Or maybe not (see later this lecture)

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
```



48 bytes for stack frame

Load local variables
Do arithmetic operation
Store result back

Tear down 48B frame

RETURNING A VALUE

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;
}
```

ARM Solution: Use X0 / W0

In principle

- Like return address, store return value in stack frame of caller (so caller always has it)

Or, for efficiency, recognize return value is set just before return and used just after

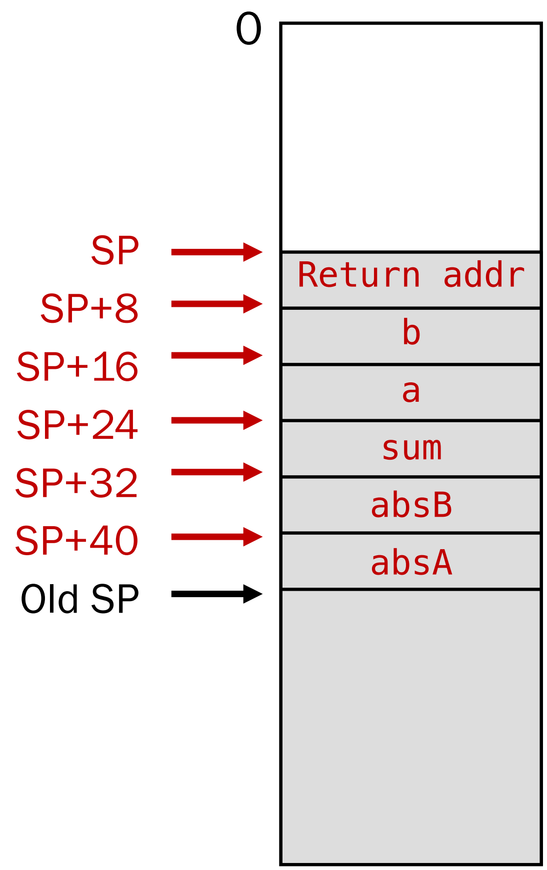
- Known small size ⇒ store return value in register
- For unknown or large size ⇒ still store return value in stack frame of caller

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)

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
```



OPTIMIZATION

(More to come on this general topic in a later lecture)

Optimization: Using Registers More Than the Stack

Observation: Accessing memory is expensive

- Orders of magnitude more expensive than accessing registers. Stack is in main memory
- For efficiency, want to store and reuse parameters and local variables in registers if possible
- So far: `ldr` from stack, use once, `str` to stack. Want to reduce use of stack, use registers instead

Observation: Registers are a finite resource

- Desired Abstraction: Every function has its own registers
- In reality: All functions share same small set of registers

Problem: How do caller and callee use the 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

Solution: carve off some registers for which you're guaranteed any function you call won't corrupt them; for others, don't assume that

ARM Solution: Register Conventions

Callee-saved registers (callee will ensure values aren't corrupted)

- X19..X29 (or W19..W29)
- Callee function must preserve contents or restore them before returning
- Safe places for callee to put values, since it's callees will also preserve them
- If necessary (e.g. if callee wants to modify these registers) ...
  - Callee saves to stack near beginning
  - Callee restores from stack near end. Caller's register values are restored

ARM Solution: Register Conventions

Callee-saved registers (callee must ensure values aren't corrupted)

- X19..X29 (or W19..W29)
- Callee function must preserve contents or restore them before returning
- Safe places for callee to put values, since its callees will also preserve them
- If necessary (e.g. if callee wants to modify these registers) ...
  - Callee saves to stack near beginning
  - Callee restores from stack near end. Callers register values are restored

Caller-saved registers (caller's job to ensure values aren't corrupted)

- X8..X18 (or W8..W18) – plus parameters in X0..X7
- Callee function can change contents and doesn't have to restore them
- If necessary...
  - Caller saves to stack before call
  - Caller restores from stack after call

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, X1 to callee-saved registers X19, X20, which will remain safe
- Body of `absadd()` uses X19 and X20
- Must be careful:
  - `absadd()` itself cannot corrupt contents of X19 and X20
  - So `absadd()` must save X19 and X20 near beginning, and restore near end
  - But it knows every other function it calls will also ensure the safety of X19 and X20

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 callee-saved registers 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

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        // Save a in x19
   mov x20, x1        // Save b in x20
   // absA = labs(a)
   mov x0, x19        // Prepare to pass a
   bl labs
   mov x21, x0        // Save absA
   // absB = labs(b)
   mov x0, x20        // Prepare to pass b
   bl labs
   mov x22, x0        // Save absB
   // sum = absA + absB
   add x23, x21, x22
   // return sum
   mov x0, x23        // Prepare to return 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, 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

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        // Save 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 'mov' instructions 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

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 this pattern 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 this pattern when invoked with –O option, when it can

WRITING READABLE CODE

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
    bl labs
    mov x21, x0
    // absB = labs(b)
    mov x0, x20
    bl labs
    mov x22, x0
    // sum = absA + absB
    add x23, x21, x22
    // return sum
    mov x0, x23
    ldr x30, [sp]
    ldr x19, [sp, 8]
    ldr x20, [sp, 16]
    ldr x21, [sp, 24]
    ldr x22, [sp, 32]
    ldr x23, [sp, 40]
    add sp, sp, 48
    ret
```

Problem

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

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         // Save a (in x19)
    mov b, x1         // Save 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`

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`

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
    // Offset sizes to save registers, in bytes
    .equ oldX19, 8
    .equ oldX20, 16
    ...

// long absadd(long a, long b)
absadd:
    // long absA, absB, sum
    sub sp, sp, STACKSIZE
    str x30, [sp]
    str x19, [sp, oldX19]
    str x20, [sp, oldX20]
    str x21, [sp, oldX21]
    str x22, [sp, oldX22]
    str x23, [sp, oldX23]
    mov a, x0
    mov b, x1
    ...
```

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`

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]
ldr x19, [sp, oldX19]
ldr x20, [sp, oldX20]
ldr x21, [sp, oldX21]
ldr x22, [sp, oldX22]
ldr x23, [sp, oldX23]
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`

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 callee-saved registers
  - Be careful ...

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