Computer Organization and Architecture
ARM64 Assembly Language

Dr Russ Ross

Dixie State University—Computing and Design

Spring 2021
To implement functions/procedures, we must be able to:

- Jump to the code that implements the function
- Return to the jump site when it is finished
- Pass arguments to the function
- Get return values back from the function
Call and Return

- Here is code that
  - calls a function with two arguments (x0 and x1)
  - then uses the return value (x0) as the argument to the exit system call
- `bl` is the *branch with link* instruction. It
  - jumps to address 0x40008c (copies that address to the PC)
  - copies the address 0x400084 (the instruction following `bl`, where the function should return to) to the *link register*, a.k.a. x30.
- The `ret` instruction
  - copies the value from x30 back into the PC, effectively
  - jumping back to the the instruction after the `bl`
- Notice the 3 visible in the `bl` instruction. The actual address is computed by skipping forward 3 instructions. Since every instruction is exactly 4 bytes long, it is encoded as the number of instructions to skip forward (or backward) and that number is multiplied by 4 before being added to the PC.

```
400078: d28000a0 mov x0, #0x5
40007c: d28000e1 mov x1, #0x7
400080: 94000003 bl 0x40008c
400084: d2800ba8 mov x8, #0x5d
400088: d4000001 svc #0x0
40008c: 8b010000 add x0, x0, x1
400090: d65f03c0 ret
```
Call and Return

- Here is the original source code
  - _start is the entry point for every program
  - we use a label for the function add2 and let the assembler compute the address
  - we use another label for the system call number
  - Note the differences between the call and the return
    - bl has the (relative) address hard-coded in the instruction
      - it always branches to the same place
      - it encodes a relative address since the branch source and target are fixed
    - the return address is compute on-the-fly by bl and stored in a register for ret to use
      - the function could be called from many different places in the program
    - bl computes the full address 0x400084 and stores it in x30
    - it needs the ability to branch back to a different place each time
  - What happens if add2 needs to call a function?

```assembly
_start:  mov x0, #5
         mov x1, #7
         bl  add2
         mov x8, #sys_exit
         svc #0
add2:   add x0, x0, x1
        ret
```
How do functions pass arguments?

1. On the stack
2. In registers
3. In registers and on the stack

Calling conventions are the rules functions follow to interoperate with each other. A function call requires coordination between the caller (the code initiating the function call) and the callee (the code being called).

Each function allocates a chunk of stack space called a stack frame where it can store private data. Compilers/programmers have quite a bit of freedom in how they manage the stack frame, but it also has some structure that everyone must honor.
Parameters

- Arguments go in the registers x0–x7 in order
- The return value is put in x0
- Note: nothing magic happens when a bl instruction is issued: it is up to the caller to put the arguments in the right place and then the callee trusts that they are there. Same for return value.

Complications:

- What if there are more than 8 arguments?
  - The remaining arguments go on the stack
- What if an argument does not fit in a register (a struct)
  - The too-big argument goes on the stack

Details:

Registers

- x31: The stack point (sp) is a special case. Name it xzr or wzr to use it as the zero register. It provides zero as an input, or you can write to it to throw a value away. Name it sp to use it as the stack pointer.

- x30: The link register (referred to as lr) can be used like a normal register, but it is where return addresses are stored by the bl instruction and we normally do not use it for anything else.

- x29: the frame pointer (referred to as fp) is related to the stack pointer. It points to the top of the current stack frame, and has an important role in debugging.

- x0–x15: scratch registers—also called caller-saved registers—the callee can use these without restriction, so if the caller cares about their contents, the caller must save them before making a function call.

- x19–x28: callee-saved registers—if the callee uses them, then the callee must restore the original values before returning. Standard practice is to store the old values on the stack frame at the beginning of the function, use them, then load the values back from the stack frame to restore them before returning.
Stack frames

Each instance of a function allocates a *stack frame*, which is just a chunk of stack space owned by that instance. Stack frames typically hold:

- The return address (from the link register)
- The old frame pointer
- Parameters that do not fit in the registers
- Local variables
- Copies of callee-saved registers

Imagine a recursive function that calls itself many times. Each stack frame is tied to a single call, so there can be many instances of the function outstanding but their local storage will not overlap or be confused.
### Example stack frame

Consider a typical stack frame and the code that sets it up:

#### myfunc: // function prelude:
- `stp x29, x30, [sp, -16]!`
- `mov x29, sp`
- `sub sp, sp, #32`
- `str x19, [x29, #-32]`
- `str x20, [x29, #-24]`
- `str x21, [x29, #-16]`
- `str x22, [x29, #-8]`

#### // main function code goes here
- `ldr x19, [x29, #-32]`
- `ldr x20, [x29, #-24]`
- `ldr x21, [x29, #-16]`
- `ldr x22, [x29, #-8]`

#### // can use x19, x20, x21, and x22

#### // function postlude:
- `ldr sp, [x29, #32]`
- `ldp x29, x30, [sp], 16`
- `ret`