LC-3
Subroutines

(Textbook Chapter 9)
Subroutines

• Blocks can be encoded as subroutines
• A subroutine is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by a user program
  - returns control to the calling program when finished

• Reasons for subroutines:
  - reuse useful (and debugged!) code without having to keep typing it in
  - divide task among multiple programmers
  - use vendor-supplied library of useful routines
JSR

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.
- saving the return address is called “linking”
- target address is PC-relative (PC + Sext(IR[10:0]))
- bit 11 specifies addressing mode
  • if =1, PC-relative: target address = PC + Sext(IR[10:0])
  • if =0, register: target address = contents of Register[IR[8:6]] (JSRR)
JSR

1. PC

2. Sext

3. Instruction Reg

4. Register File

5. ALU

6. R7
JSRR

• Just like JSR, except Register addressing mode.
  - target address is Base Register
  - bit 11 specifies addressing mode

• What important feature does JSRR provide that JSR does not?
JSRR
Returning from a Subroutine

• RET (JMP R7) gets us back to the calling routine.
  - just like the return to a TRAP (later)
Ex: Negate the value in R0

- 2sComp
  - NOT R0, R0 ; flip bits
  - ADD R0, R0, #1 ; add one
  - RET ; return to caller

- To call from a program (within 1024 instructions):
  - ; need to compute R4 = R1 - R3
  - ADD R0, R3, #0 ; copy R3 to R0
  - JSR 2sComp ; negate
  - ADD R4, R1, R0 ; add to R1
  - ...

Passing Information to/from Subroutines

• Arguments
  - A value **passed in** to a subroutine (or trap) is called an argument.
  - This is a value needed by the subroutine to do its job.
  - Examples:
    • In 2sComp routine, R0 is the number to be negated
    • In PUTS routine, R0 is _address_ of string to be printed.

• Return Values
  - A value **passed out** of a subroutine is called a return value.
  - This is the value that you called the subroutine to compute.
  - Examples:
    • In 2sComp routine, negated value is returned in R0.
    • In GETC service routine, character read from the keyboard is returned in R0.
Using Subroutines

In order to use a subroutine, a programmer must know:

- **its address** (or the label that will be bound to its address)
- **its function** (what does it do?)
  - **NOTE:** The programmer does not need to know how the subroutine works
- **its arguments** (where to pass data in, if any)
- **its return values** (where to get computed data, if any)
- **registers that are changed by Subroutine**
Saving and Restoring Registers

• Must save the value of a register if:
  - Its value will be destroyed by service routine, AND
  - We will need to use the value after that action.

• Who saves?
  - caller of service routine?
    • knows what it needs later, but may not know what gets altered by called routine
  - called service routine?
    • knows what it alters, but does not know what will be needed later by calling routine
Who saves the registers?

• The *Called* routine -- "**callee-save**"
  - Before start, save any registers that will be altered (unless altered value is desired by calling program!)
  - Before return, restore those same registers

• The *Calling* routine -- "**caller-save**"
  - Save all registers that will be needed later before the call
  - Restore after return.
  - In special cases, avoid using those registers altogether

• *Values are saved by storing them in memory.*

• Generally use "callee-save" strategy, except for return values

• Remember to save R7 before any other call (*incl. TRAPs!*) or you won't be able to return
Recommended exercises

• Ex 9.2, 9.5, 9.6, 9.12, 9.13