Subroutines

- Blocks of code can be encoded as subroutines
  - One entry point – one exit point
- 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 an unconditional branch), and saves current PC (address of next instruction) in R7
  - Saving the return address is called “linking”
  - Target address is PC-relative ($PC + \text{Sext}(IR[10:0])$)
  - Bit 11 specifies addressing mode
    - If = 1, PC-relative: target address = $PC + \text{Sext}(IR[10:0])$
    - If = 0, register: target address = contents of Register$[IR[8:6]]$ (JSRR)

PC-Relative

![](image1)

![](image2)
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?

RET

![Diagram of PC and Register File with R7 and Base highlighted]
Returning from a Subroutine

- **RET (JMP R7)** gets us back to the calling routine
  - Just like **TRAP**

---

**Ex: Negate the value in R0**

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

JMP R7
```

To call from a program (within 1024 instructions)
- JSR 2sComp
- Caller should save R0 if it will need it later!

```
; need to compute
; R4 = R1 - R3
  ADD R0, R3, #0 ; R0 ← R3
  JSR 2sComp ; negate
  ADD R4, R1, R0; add to R1 ...
```
Passing Information to/from Subroutines

- **Arguments**
  - A value passed in to a subroutine 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, but what changes are visible in the machine’s state after the routine has run.
  - Its arguments (where to pass data in, if any)
  - Its return values (where to get computed data, if any)
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
  - 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 9.8, 9.12, 9.13