What about User Code?

Service routines provide three main functions

1. **Protect system resources from malicious/clumsy programmers**
2. **Shield programmers from system-specific details**
3. **Write frequently-used code just once**

Do these benefits apply to application code, too?

Subroutines

A subroutine is a program fragment that. . .

- **Resides in user space (*i.e,* not in OS)**
- **Performs a well-defined task**
- **Is invoked (called) by a user program**
- **Returns control to the calling program when finished**

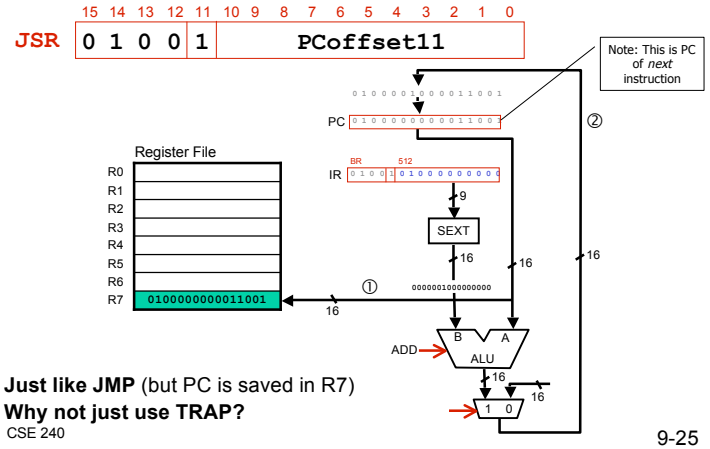
Like a service routine, but not part of the OS

- **Not concerned with protecting hardware resources**
- **No special privilege required**

Virtues

- **Reuse useful code without having to keep typing it in (and debugging it!)**
- **Divide task among multiple programmers**
- **Use vendor-supplied *library* of useful routines**

JSR

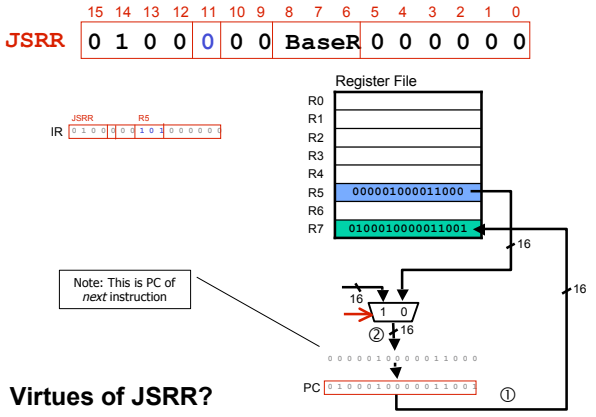


Just like JMP (but PC is saved in R7)

Why not just use TRAP?

JSRR

this zero means "register mode"



Virtues of JSRR?

Returning From a Subroutine

The RET instruction

- Just a special case of JMP
  RET == JMP R7
- Same idea as returning from TRAPs

Note

- If we use JMP to call subroutine, we can't use RET!
- Why not?

Example: 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

```
; need to compute R4 = R1 - R3
         ADD  R0, R3, #0  ; copy R3 to R0
         JSR  2sComp      ; negate
         ADD  R4, R1, R0  ; add to R1
         ...
```

Note: Caller should save R0 if we'll need it later!

Passing Information To Subroutines

Argument(s)

- Value passed in to a subroutine is called an argument
- This is a value needed by the subroutine to do its job
- Examples
  - 2sComp: R0 is number to be negated
  - OUT: R0 is character to be printed
  - PUTS: R0 is ***address*** of string to be printed

How?

- In registers (simple, fast, but limited number)
- In memory (many, but awkward, expensive)
- Both

Getting Values From Subroutines

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
  - 2sComp: negated value is returned in R0
  - GETC: character read from the keyboard is returned in R0

How?

- Registers, memory, or both
- Single return value in register most common

Calling Conventions

Caller/Callee must agree on argument/ret-val location

Approach 1

- Every subroutine does what it likes
- Program needs to look at documentation for each one

Approach 2

- Define a consistent *calling convention*

LC-3

- First 4 arguments passed in R0, R1, R2, R3
- Subsequent arguments passed in memory (more on this later)
- Single value returned in R5

Using Subroutines

Programmer must know

- **Address:** or at least a label that will be bound to its address
- **Function**: what it does
  - 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
- **Arguments**: what they are and where they are placed
- **Return values:** what they are and where they are placed

Saving and Restore Registers

Like service routines, must save and restore registers

- Who saves what is part of the calling convention

Generally use "callee-save" strategy, except for ret vals

- Same as trap service routines
- Save anything that subroutine alters internally that shouldn't be visible when the subroutine returns
- Restore incoming arguments to original values (unless overwritten by return value)

Remember

- You MUST save R7 if you call any other subroutine or trap
- Otherwise, you won't be able to return!