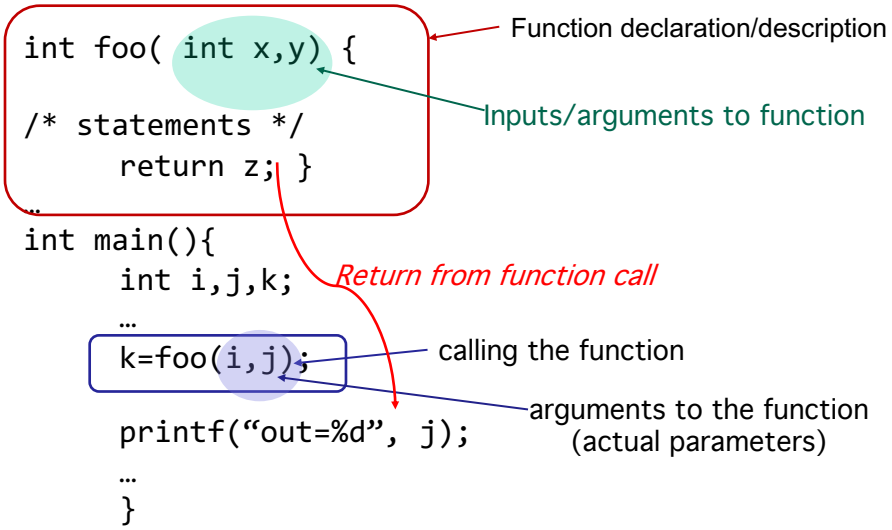
# LC3 Assembly Programming: Subroutines



1

## Function Calls….




2

2

## Function call and return: To do

- Define body of function
- Specify input parameters
- Mechanism to call the function – *need assembly instruction*
- Pass the arguments to the function
- Return from the function…to instruction immediately following the function call – *need assembly instruction*

- Question: In Assembly (LC3), how can we pass the arguments to the function ?
  - Note that function parameters are "fixed"

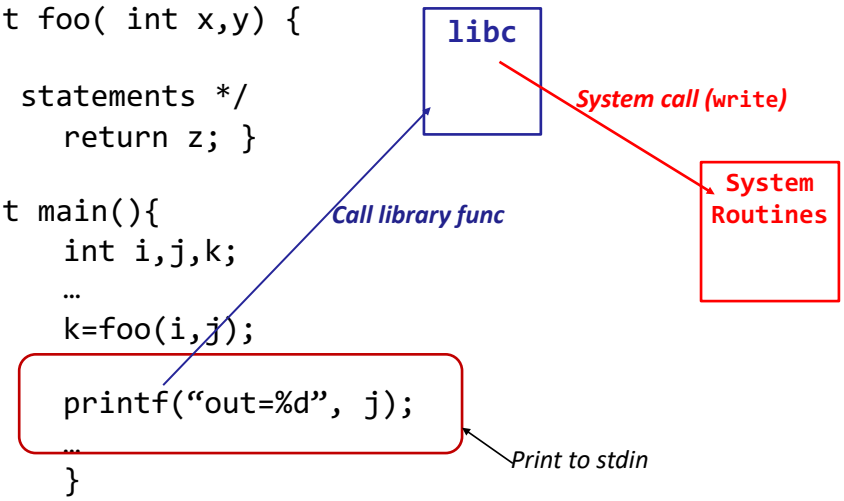
Registers!

3

3

## Input/Output….

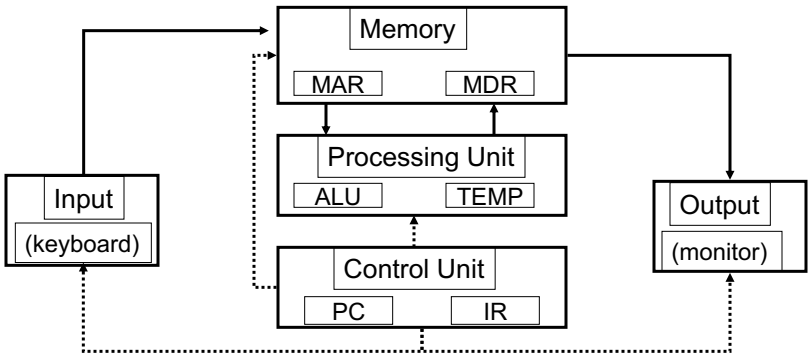
```
int foo( int x,y) {

/* statements */
      return z; }
…
int main(){
      int i,j,k;
      …
      k=foo(i,j);

      printf("out=%d", j);
      …
      }
```




4

4

## The von Neumann Model




- **Memory: holds both data and instructions**
- **Processing Unit: carries out the instructions**
- **Control Unit: sequences and interprets instructions**
- **Input: external information into the memory**
- **Output: produces results for the user**

5

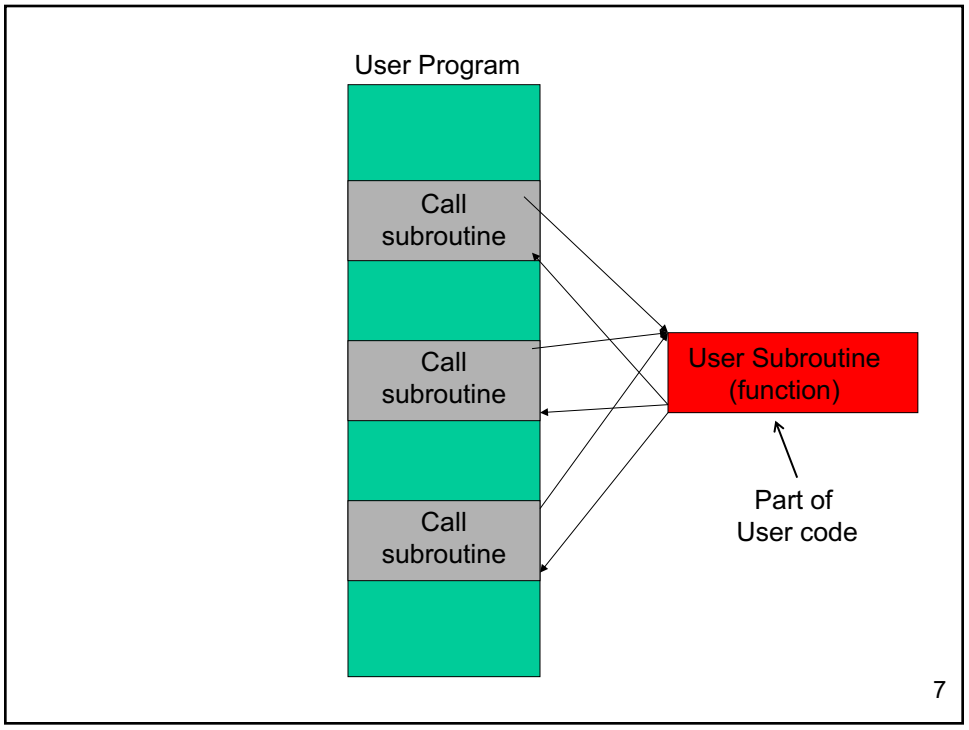
5

## Subroutines in LC3

- we covered TRAP routines
  - System calls to process I/O (or other system tasks)
  - Written by system, called by user
    - Resides as part of system code
  - Steps: Call, Process, Return
- Subroutines – i.e., functions
  - Written by user
  - Called by user program
  - Steps: Call, Process, Return

6

6




7

7

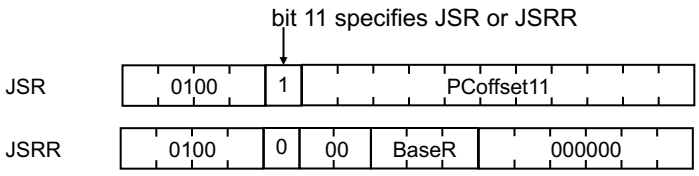
## In Assembly: Subroutines

- A subroutine is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by 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
  - Written by user
- To call a subroutine (i.e, function) in assembly, we have to "go" to a specific address and execute instructions starting at that address
  - Therefore the Call mechanism has to specify the address of the subroutine

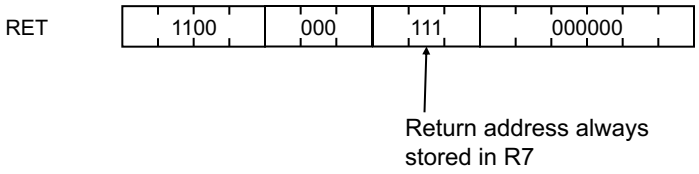
8

8

## LC3 Call/Return Mechanism




They differ in how the address of the subroutine is obtained




9

9

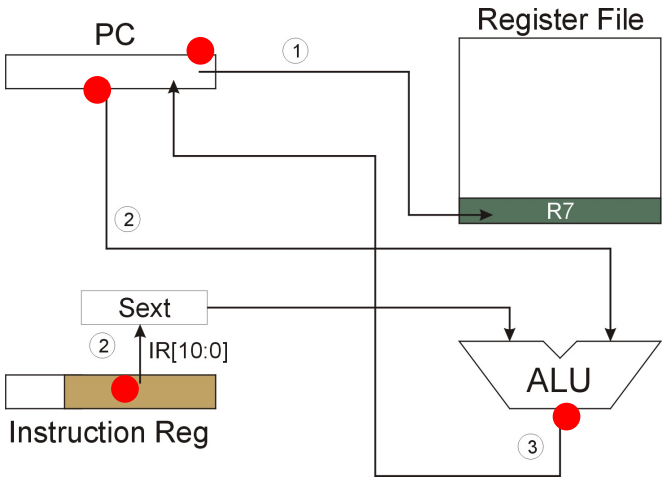
## JSR Instruction


- 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]
- JSR can be used to call a subroutine that is at an address within the 11 bit offset
  - *Can call subroutine within range of - $2^{10}$ to $2^{10}$ -1 addresses from current instruction*

10

10

## JSR




NOTE: PC has already been incremented during instruction fetch stage.

11

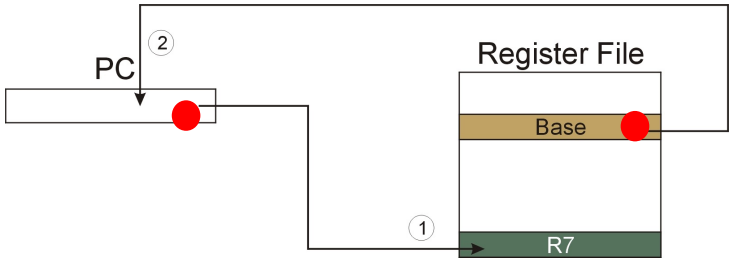
## JSRR Instruction


- Just like JSR, except Register addressing mode.
  - target address is in Base Register
  - bit 11 specifies addressing mode
- JSRR R4 ; calls subroutine whose address is in R4
  - *R4 should have been loaded with address of subroutine before the JSRR instruction*
    - LD R4, example
    - example .FILL x1234
    - gotoSUB .FILL Sub ; can use .FILL <label>
- What important feature does JSRR provide that JSR does not?

12

12

## JSRR




NOTE: PC has already been incremented during instruction fetch stage.

13

## Returning from a Subroutine

- RET (JMP R7) gets us back to the calling routine.
  - just like TRAP

14

14

## Example: Subtraction

- LC3 does not have SUB instruction…
- To do subtraction we write set of instructions:

```
      .ORIG x3000   ; subtract R1 from R2
SUB    NOT R0, R1  ; complement R1 and add 1 to get
      ADD R0, R0, #1      ; 2's complement, R2 = -R1
      ADD R3, R0, R2 ; R3= R2 + R0 = R2 – R1
      HALT
      .END
```

This code keeps original value in R1 unchanged

15



## 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.
- 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.

- In assembly – how to pass arguments and return values ?
- Registers:
  - In GETC service routine, character read from the keyboard is returned in R0.
  - In OUT service routine, R0 is the character to be printed.
  - In PUTS routine, R0 is ***address*** of string to be printed.
  - **In SUB: inputs in R1,R2 and output in R3**

16

## Using Subroutines

- In order to use a subroutine, a programmer must know:
  - its address (or at least a 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) = which registers?
  - its return values (where to get computed data, if any)= in which register?
- User code must save registers used to pass arguments
  - If subroutine uses other registers, then save them before use and restore before returning
- Example: SUB
  - Inputs are in registers R1, R2
  - Output is in R3

17

17

## Changing Subtraction code to a Subroutine

- need to be able to call and return from SUB subroutine
- inputs are in R1,R2 and Output is in R3= R2 – R1

give label to first line in the code…this is address of subroutine SUB…To call, the user program needs to set PC to this address

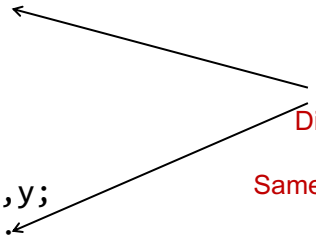
```
SUB    NOT R0, R1  ; complement R1 and add 1 to get
       ADD R0, R0, #1      ; 2's complement R2 = -R1
       ADD R3, R0, R2 ;  R3= R0+R2 = R2 – R1
       RET    ; replace HALT by RET to return to caller
```

18

18

## Observation:
## Concept of Scope in High level languages

```
int sub(x,y){
      int z
….}

int main{
      int x,y;
      int z;
      …
      z= sub(x,y);
…}
```




19

19

## Using SUB from 'main'

- main code: subtract two numbers in memory and write back difference.
  - Read two numbers from memory locations number1, number2 and store into registers R1, R2.
  - Call SUB and store result in memory location result

```
      .ORIG x3000
      LD R0, number1
      LD R1, number2
      JSR SUB ; call SUB (JSRR if SUB not within 2^10)
      ST R3, result; store result returned in R3 into memory
      HALT
Number1    .FILL x000A
Number2    .FILL #8
Result     .BLKW #1      ;reserve space for result
```

*If R2 is used in main then need to save them into memory*

20

20

```
; what if address of SUB is not within 11 bit offset?
.ORIG x3000
Loop        LD R1, number1 ; load number1 into R1
            LD R2, number2 ; load number2 into R2
            ST R3, SaveR3  ; save register R3
            LD R5, goSUB   ; load address of SUB into R5
            JSRR R5 ; go to subroutine whose address in R5
            STR  R3, result
            LD R3, SaveR3          ; restore old value R3
            HALT
number1     .FILL #10
number2     .FILL # -8

goSUB       .FILL  SUB ; initialize goSUB to address of SUB

SaveR3      .BLKW 1 ; reserve space for SaveR3
result      .BLKW #1
SUB         NOT R0, R1
            ADD R0, R0, #1 ; R0 = -R1
            ADD R3, R0, R2
            RET
            .END
```

21

21

## Saving and Restoring Registers

- What if the same registers are used in the "main" and in the subroutine ?
  - Need to save the registers so their value is not overwritten
- 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
- Calling routine -- *"caller-save"*
  - Save registers destroyed by own instructions or by called routines (if known), if values needed later
    - ex: save R0 before TRAP x23 (input character)
    - ex: save R7 before calling routine
  - Or avoid using those registers altogether
- Pick a convention and use it consistently in your ASM code
- *Values are saved by storing them in memory.*

22

22

## Example: subroutine1.asm

- Replace each element A[i] in an array with A[i]-X until A[i] is <=0
  - Call subroutine SUB to compute subtraction A[i] –X
  - Assume X is stored at some memory location
- Open subroutine1.asm in LC3

```
i=0;
while ( A[i] >= 0) {
      A[i] = X - A[i];
      i=i+1; }
```

A[i] = SUB(A[i], X);

23



## Multiplication using repeated addition

- No Multiplication operation in LC3….implement MULT using repeated addition

```
;assume non-zero positive x,y
while (x>0) {
       mult = mult + y;
       x= x-1;
```

LC3 code outline

Multiply the values in registers R1, R2 and return in R3

```
; initially clear R3 and check if either X or Y is zero
ADD R0, R2, #0  ; copy R2 to R0
loop   BRnz done
       ADD R3, R3, R1 ; add to product
       ADD R0, R0, # -1 ; decrement X
       BRnzp loop
done   HALT   ; program halts with product in R3
```

24

## Exercise -Multiplication

Write LC3 code to compute product of array element with X

```
i=0;
while ( A[i] >= 0) {
      A[i] = MULT(X,A[i];
      i=i+1; }
```

Write MULT subroutine
Assume for now that X,Y are positive…
but work on implementing general version later
…need it for HW4/project!

LC3 code outline for multiplication
Multiply the values in registers R1, R2 and return in R3

```
; initially clear R3 and check if either X or Y is zero
ADD R0, R2, #0  ; copy R2 to R0
loop   BRnz done
       ADD R3, R3, R1 ; add to product
       ADD R0, R0, # -1 ; decrement X
       BRnzp loop
done   HALT   ; program halts with product in R3
```

25

25

## Exercises……

- 1. Download ques3.asm – assemble and run (don't forget breakpoint)
  - This is same algorithm as subroutine1.asm
  - Run the code…… explain what happens
- 2. Download 3 files: ques4.asm, data1.obj, and data2.obj
  - Note: ques4.asm is identical code to last week addition of two input digits – ques2.asm
  - Assemble ques2.asm and then
    1. First load data1.obj
    2. Next load data2.obj
    3. Next load ques2.obj and set PC to x3000 (and set breakpoint)
  - Run code….explain what happens
- 3. Implement multiplication

26

26

## Protecting System space

- System calls go to specific locations in memory
  - We don't want users overwriting these
  - Write protect these locations
  - Halt a program that tries to enter unauthorized space/memory

27

27

## Operating Systems (OSes)

First job of an OS:

- Handle I/O …2nd job of OS …
- OSes virtualize the hardware for user applications

In real systems, only the operating system (OS) does I/O

- "User" programs *ask* OS to perform I/O on their behalf
- Three reasons for this setup:

1) Abstraction/Standardization

- I/O device interfaces are nasty, and there are many of them
- Think of disk interfaces: S-ATA, iSCSI, IDE
- User programs shouldn't have to deal with these interfaces
  - In fact, even OS doesn't have to deal with most of them
  - Most are buried in "device drivers"

28

28

## Operating Systems (OSes)

- 2) Raise the level of abstraction
  - Wrap nasty physical interfaces with nice logical ones
    - Wrap disk layout in file system interface
- 3) Enforce isolation (usually with help from hardware)
  - Each user program thinks it has the hardware to itself
    - User programs unaware of other programs or (mostly) OS
  - Makes programs much easier to write
  - Makes the whole system more stable and secure
    - A can't mess with B if it doesn't even know B exists

29

29

## Implementing an OS: Privilege

OS isolates user programs from each other and itself

- Requires restricted access to certain parts of hardware to do this
- Restricted access should be enforced by hardware
- Acquisition of restricted access should be possible, but restricted

Restricted access mechanism is called privilege

- Hardware supports two privilege levels

"Supervisor" or "privileged" mode

- Processor can execute any code, read/write any data

"User" or "unprivileged" mode

- Processor may not execute some code, read/write some memory
  - E.g., cannot read/write video memory or device registers

30

30

## Privilege in LC3

PSR (Processor Status Register)?

- PSR[15] is the privilege bit
- If PSR[15] == 1, current code is "privileged", i.e., the OS

instruction and data memories split into two- example:

- x0000-x7FFF: user segment
- x8000-xFFFF: OS segment
  - Video memory (xC000-xFDFF) is in OS segment
  - I/O device registers (xFE00-xFFFF) are too

If PSR[15]==0 and current program tries to …

- … execute an instruction with PC[15] == 1
- … or read/write data with address[15] == 1
- … "hardware" kills it!

31

31