Subroutines and TRAP Routines in LC3

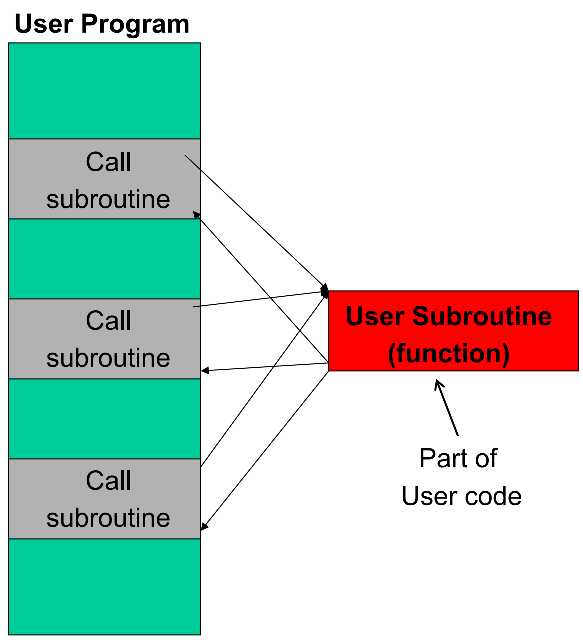
Based on slides © McGraw-Hill

1

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

2



3

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

4

LC3 Call/Return Mechanism

JSR	0100	1	PCoffset11		
JSRR	0100	0	00	BaseR	000000

They differ in how the address of the subroutine is obtained

RET	1100	000	111	000000

JSR Instruction

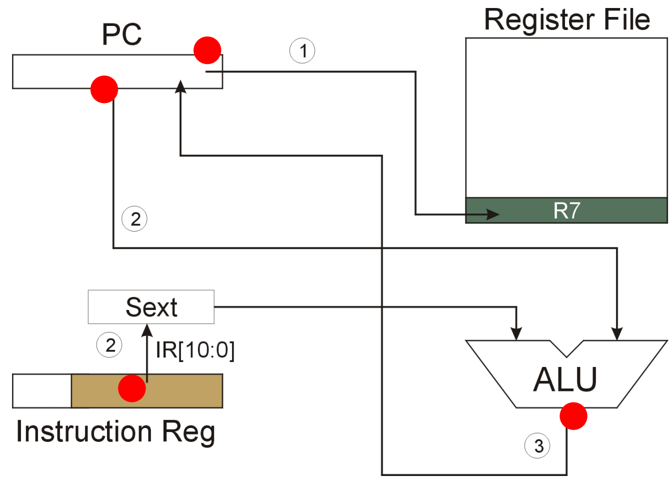
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSR	0	1	0	0	1	PCoffset11										

•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
  - -2^{10} to $2^{10}-1$

JSR



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

7

JSRR Instruction

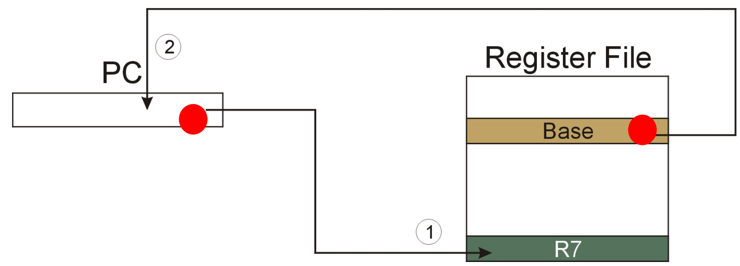
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSRR	0	1	0	0	0	0	0	Base			0	0	0	0	0	0

- 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

- What important feature does JSRR provide that JSR does not?

8

JSRR



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

9

Returning from a Subroutine

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

10

Example: Subtraction

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

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

11

Changing Subtraction code to a Subroutine

- need to be able to call and return from SUB subroutine
- inputs are in R0,R1
- Output is in R3= R1-R0

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

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

12

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 R0,R1 and output in R2

13

Concept of Scope in High level languages

```
int sub(x,y){
        int z
….}

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

These two are
Different variables in C
BUT
Same (registers) in assembly!

14

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

•Values are saved by storing them in memory.

15

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)
  - its return values (where to get computed data, if any)
- 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 R0, R1
  - Output is in R3

16

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 R0, R1.
  - Call SUB and store result in memory location result

```
            .ORIG x3000
            LD R0, number1
            LD R1, number2
            ; now call SUB – use JSRR if SUB is within 11 bit offset
            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

17

```
; what if address of SUB is not within 11 bit offset?
.ORIG x3000
LoopLD R0, number1 ; load number1 into R0
            LDR R1, number2 ; load number2 into R1
            ST R2, SaveR2     ; save register R2
            LD R5, goSUB      ; load address of SUB into R4
            JSRR R5           ; go to subroutine whose address in R5
            STR  R3, result   ; store result
            LD R2, SaveR2               ; restore old value R2
            HALT
number1     .FILL #10
number2     .FILL # -8
goSUB       .FILL  SUB ; initialize goSUB to address of SUB
SaveR2      .BLKW 1; reserve space SaveR2 and SaveR3
result      .BLKW #1
SUB         NOT R1, R1
            ADD R2, R1, #1
            ADD R3, R0, R2
            RET
            .END
```

18

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
- Role of the O/S
  - Enforce Isolation
  - Privilege level

19

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"

20

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

21

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

22

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!

23

Next…..

•Stack in assembly

- Used in ASCII to Binary etc.
- Interrupt processing
- And………need it to support high level languages
  - ex: C to Assembly

24