Programming Studio #6

ECE 190

Programming Studio #6

- Concepts this week
  - TRAP instruction and trap service routines
  - Subroutines
  - Stack

Announcements

- MP2 Checkpoint 1 due Weds. 3/3 at 5pm
  - Read handout carefully to not lose points
- MP2 Checkpoint 2 due Weds. 3/10 at 5pm
- Note: MP2 requires use of TRAPs and subroutines covered today!
  - OUT, PUTS

TRAP Service Routines

- TRAP Routines are provided by the Operating System and called by a User program to perform a specific task
- Uses the TRAP instruction to call a TRAP Routine

```
    TRAP  1111  0000  trapvect8
```

- trapvect8 is an 8-bit offset
  - Used to determine the starting address of a TRAP routine
  - Trap vector table is in memory locations x0000 through x00FF and contains starting addresses of the TRAP routines
  - How many possible traps are there?
- When a TRAP instruction is used, R7 is loaded with the current contents of PC, and PC is loaded with the address of the TRAP routine
  - Why?
  - When the TRAP routine is completed, PC is loaded with the value stored in R7 and control is returned to the user program
  - What does this mean about using R7 as a general purpose register now?

TRAP Routines

Trap vector	Assembler Name	Description
x20	GETC	Read one character from keyboard into R0 ANDed with 0x00FF; doesn't echo
x21	OUT	Write R0[7:0] to display
x22	PUTS	Display string from subsequent locations starting at mem[R0] until x0000 (null character) in a location
x23	IN	Read a single character from keyboard; echo character; R0 <- character & x00FF
x24	PUTSP	Display string from subsequent locations starting at mem[R0], with 2-chars per location, bits [7:0] first, then [15:8] until x0000 in a location; [15:8] = x00 if odd length
x25	HALT	Halt execution and print message

TRAP Example: echo again

```
        .ORIG x3000
        LEA R0, MSG
        TRAP x22 ; PUTS
LOOP    TRAP x20 ; GETC
        TRAP x21 ; OUT
        BRnzp LOOP
MSG     .STRINGZ "User Input: "
        .END
```

Subroutines

- Subroutines allow us to write a piece of code once, and execute it several time throughout a program
- Use JSR(R) instruction to jump to a subroutine

```
JSR  0100 1 PCOffset11
JSRR 0100 0 00   baseReg   000000
```

- When a JSR(R) instruction is executed the return address is stored in R7, and PC is loaded with the address of the subroutine
  - Bit 11 of JSR(R) determines the addressing mode
  - PC-relative or Base Register
- Use RET (JMP R7) instruction to return to caller function

Subroutine Example: Multiply

```
        .ORIG x3100
        ; R0 <- R1 * R2
MULT    ST R2, SaveR2
        AND R0, R0, 0
MLOOP   ADD R0, R0, R1
        ADD R2, R2, -1
        BRp MLOOP
        LD R2, SaveR2
        RET
SaveR2  .BLKW 1
        .END
```

- **IMPORTANT**: Subroutines should not *clobber* registers!
- Save/Restore any registers used in the subroutine besides the register used to hold a return value
- Callee vs. caller save

Full Example

- To demonstrate how to use TRAP routines and Subroutines in a program, we have provide a Multiplier program (**multiplier.asm**)
- The program retrieves two digits from the user (ranging from 1 to 9), multiples them together, and displays the answer
- Several TRAP routines are used to get input and display output. Also, MULT subroutine is used to multiple the two numbers, and B2A (Binary2ASCII) converts the result into a data type that can be displayed on the screen
- Note: If a subroutine calls a TRAP function you ***must save R7 and restore it*** before you return!

Stack

- A ***stack*** is an *abstract data type*
  - An abstract data type is a storage mechanism that is defined by the operations performed on it
- With a stack the last data you stored in it is the first data you remove from it
  - **Last In, First Out (LIFO)**
- Inserting an element onto the stack is called a ***Push***
- Removing an element from the stack is called a ***Pop***
- Questions:
  - Can a stack be empty (no elements)? What does pop do?
  - Can a stack ever be full (unable to insert more elements) with this definition?
  - Does the LC-3 have finite memory? What does push do?

Implementing the Stack

- Sequence of memory locations along with a ***stack pointer*** *(R6)* that keeps track of the top of the stack
  - R6 = location of most recent element pushed
- Push puts a value onto the stack
  - Stack pointer decremented and the value is stored at mem[R6]

```
PUSH  ADD R6, R6, -1
      STR R0, R6, 0
```

- Pop takes a value from the stack
  - Value loaded from mem[R6] and stack pointer incremented

```
POP   LDR R0, R6, 0
      ADD R6, R6, 1
```

- **Is order important? What does R6 start at? What if stack empty? What if it's full?**

Exercise: Palindrome Check

- Create a program that checks if a string is a ***palindrome***
  - Implementation should use a Stack
  - PUSH and POP Subroutines are provided for you (**stack.asm**)
- Examples:
  - racecar
  - otto
  - hannah
  - 12321