CS 2461: Computer Architecture I

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~bhagiweb/cs2461/

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

Operating Systems (OSes)

First job of an OS:
- Handle I/O … 2nd job of OS … well you know
- 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”
  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
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

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!

Note: LC3 simulator does not implement this….

Functions

- Procedural programming languages
  - Abstraction and functions/procedures key part of language
- Readable code
- Code reuse
- development and debugging
- Aids in problem decomposition/algorithm design

... x = FindGPA(studentID);
printf("Gpa is %d", x);
In Assembly: Subroutines

- A subroutine is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another 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

LC3 Call/Return Mechanism

<table>
<thead>
<tr>
<th>JSR</th>
<th>0100</th>
<th>1</th>
<th>PCoffset11</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>000</td>
<td>111</td>
</tr>
</tbody>
</table>

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]

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

- Just like JSR, except Register addressing mode.
  - target address is 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
  - example: LD R4, example FILL x1234

What important feature does JSRR provide that JSR does not?

Returning from a Subroutine

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

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.
Saving and Restoring Registers

- **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
  - save R7 before TRAP
  - save R0 before TRAP x23 (input character)
  - Or avoid using those registers altogether

- Values are saved by storing them in memory.

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

Subroutines

- Calling subroutine:
  - JSR – if subroutine is “close” to current PC
  - JSRR Reg – to call subroutine located anywhere

- To do:
  - Save registers
  - Determine how parameters are passed
  - Use specific registers
  - Return
    - Restore registers
    - Return value is in a specific register

TRAPs vs Subroutines

- TRAPS are system code the user can call repeatedly
- Subroutines are for user code
  - Function calls
TRAP vs JSR(R)

- **TRAP**
  - Uses trap vector table
  - (Can get to from anywhere)
  - Normally do system functions
    - DO
  - Written very carefully!
  - Typically tied into some sort of system protection mechanism

- **JSR(R)**
  - Local (JSR)
  - Anywhere (JSRR)
    - with some work
  - Routine abstraction
  - Code reuse/libraries
  - Written
  - No protection mechanism

Library Routines

- Vendor may provide object files containing useful subroutines
  - don’t want to provide source code -- intellectual property
  - assembler/linker must support EXTERNAL symbols
    - (or starting address of routine must be supplied to user)
  - . . .
    - . . . .EXTERNAL SQRT
  - . . .
    - LD R2, SQAddr ; load SQRT addr
    - JSRR R2
    - . . .
    - SQAddr .FILL SQRT

- Using JSRR, because we don’t know whether SQRT is within 1024 instructions.

Linking

- Libraries provide set of subroutines/functions
  - Pseudo-op .EXTERNAL specifies it is an external subroutine
    - Written by someone else/provided by vendor

- Create one executable image at link time
  - Combine multiple modules at link time to produce one executable image
    - Static linking