Computer Organization I

Subroutine, Trap, Stacks

June 16, 2015

Outline

- Chapter 8.1-8.6,9.1-9.2, 10.1-10.2
- Keyboard and Monitor (skip)
- Polling vs. Interrupt
- Subroutines
  - **JSR(R), RET/JMP**
- LC-3 TRAP Routines
  - **TRAP**
- The Stack Structure
  - **STI**

Trap Codes

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.

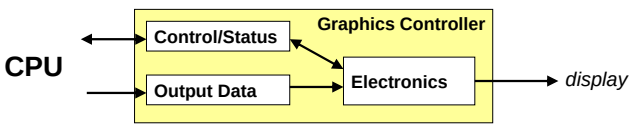
Code	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard. Character stored in R0[7:0].
PUTS	TRAP x22	Write null-terminated string to console. Address of string is in R0.

I/O: Connecting to Outside World

- So far, we've learned how to:
  - Compute with values in registers
  - Load data from memory to registers
  - Store data from registers to memory
- But where does data in memory come from?
  - Set manually
  - Set in the program
- And how does data get out of the system so that humans can use it?

I/O Controller

- Control/Status Registers
  - CPU tells device what to do -- write to control register
  - CPU checks whether task is done -- read status register
- Data Registers
  - CPU transfers data to/from device



- Device electronics
  - Performs actual operation
    - pixels to screen, bits to/from disk, characters from keyboard

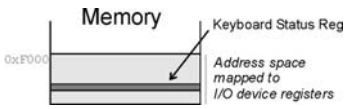
Programming Interface

- How are device registers identified?
  - Memory-mapped vs. special instructions
- How is timing of transfer managed?
  - Asynchronous vs. synchronous
- Who controls transfer?
  - CPU (polling) vs. device (interrupts)

Memory-Mapped vs. I/O Instructions

- *Instructions*
  - Designate opcode(s) for I/O
  - Register and operation encoded in instruction

- Memory-mapped
  - Assign a memory address to each device register
  - Use data movement instructions (LD/ST) for control and data transfer



*Transfer Timing

- I/O events generally happen much slower than CPU cycles.
- Synchronous
  - Data supplied at a fixed, predictable rate
  - CPU reads/writes every X cycles
- *Asynchronous*
  - Data rate less predictable
  - CPU must synchronize with device, so that it doesn't miss data or write too quickly

Transfer Control

- Who determines when the next data transfer occurs?
- Polling
  - CPU keeps checking status register until new data arrives OR device ready for next data
  - "Are we there yet? Are we there yet? Are we there yet?"
- *Interrupts*
  - Device sends a special signal to CPU when new data arrives OR device ready for next data
  - CPU can be performing other tasks instead of polling device.
  - "Wake me when we get there."

LC-3

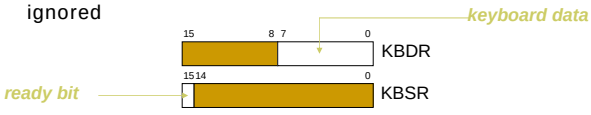
- Memory-mapped I/O (Table A.3)

Location	*I/O Register*	*Function*
xFE00	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
xFE04	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
xFE06	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.

- Asynchronous devices
  - Synchronized through status registers
- Polling and Interrupts
  - The details of interrupts will be discussed in Chapter 10

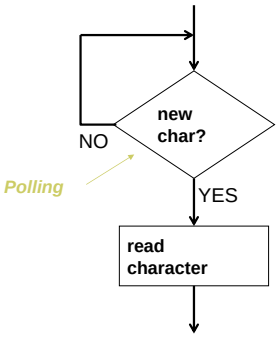
*Input from Keyboard

- When a character is typed:
  - Its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
  - The "ready bit" (KBSR[15]) is set to one
  - Keyboard is disabled -- any typed characters will be ignored



- When KBDR is read:
  - KBSR[15] is set to zero
  - keyboard is enabled

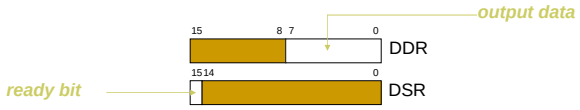
*Basic Input Routine




```
POLL      LDI  R0, KBSRPtr
          BRzp POLL
          LDI  R0, KBDRPtr
          ...
KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02
```

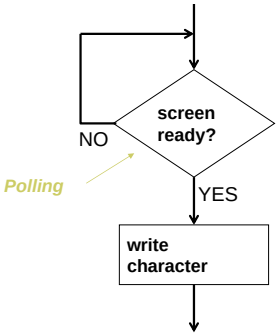
*Output to Monitor

- When Monitor is ready to display another character:
  - The "ready bit" (DSR[15]) is set to one



- When data is written to Display Data Register:
  - DSR[15] is set to zero
  - Character in DDR[7:0] is displayed
  - Any other character data written to DDR is ignored (while DSR[15] is zero)

Basic Output Routine



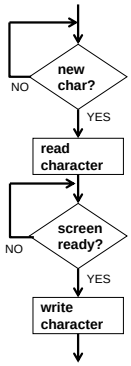

```
POLL     LDI  R1, DSRPtr
         BRzp POLL
         STI  R0, DDRPtr
         ...
DSRPtr   .FILL xFE04
DDRPtr   .FILL xFE06
```

*Keyboard Echo Routine

- Usually, input character is also printed to screen.
  - User gets feedback on character typed and knows its ok to type the next character.

```
POLL1    LDI  R0, KBSRPtr
         BRzp POLL1
         LDI  R0, KBDRPtr
POLL2    LDI  R1, DSRPtr
         BRzp POLL2
         STI  R0, DDRPtr

         ...
KBSRPtr  .FILL xFE00
KBDRPtr  .FILL xFE02
DSRPtr   .FILL xFE04
DDRPtr   .FILL xFE06
```

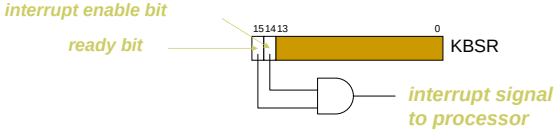


Interrupt-Driven I/O

- External device can:
  - (1) Force currently executing program to stop;
  - (2) Have the processor satisfy the device's needs; and
  - (3) Resume the stopped program as if nothing happened.
- Why?
  - Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
  - Example: Process previous input while collecting current input. (See Example 8.1 in text.)

*Interrupt-Driven I/O

- To implement an interrupt mechanism, we need:
  - A way for the I/O device to signal the CPU that an interesting event has occurred.
  - A way for the CPU to test whether the interrupt signal is set and whether its priority is higher than the current program.
- Generating Signal
  - Software sets "interrupt enable" bit in device register.
  - When ready bit is set and IE bit is set, interrupt is signaled.



TRAP Routines and Their Names

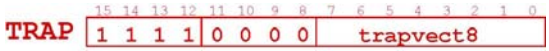
vector	symbol	routine
x20	GETC	read a single character (no echo)
x21	OUT	output a character to the monitor
x22	PUTS	write a string to the console
x23	IN	print prompt to console, read and echo character from keyboard
x25	HALT	halt the program

LC-3 TRAP Mechanism

- 1. A set of service routines.
  - Part of operating system -- routines start at arbitrary addresses (convention is that system code is below x3000)
  - Up to 256 routines
- 2. Table of starting addresses.
  - Stored at x0000 through x00FF in memory
  - Called System Control Block in some architectures
- 3. TRAP instruction.
  - Used by program to transfer control to operating system
  - 8-bit trap vector names one of the 256 service routines
- 4. A linkage back to the user program.
  - Want execution to resume immediately after the TRAP instruction

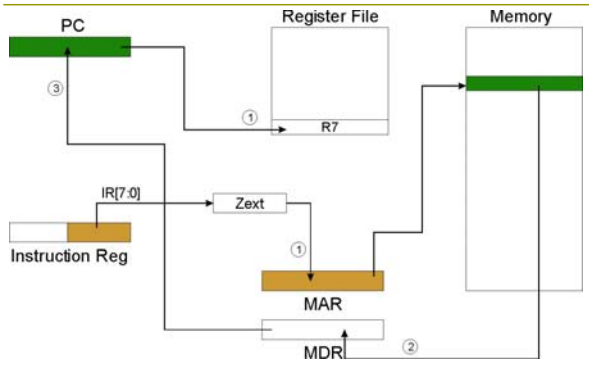
TRAP Instruction

- Trap vector
  - Identifies which system call to invoke
  - 8-bit index into table of service routine addresses
    - in LC-3, this table is stored in memory at 0x0000 – 0x00FF
    - 8-bit trap vector is zero-extended into 16-bit memory address



- Where to go
  - Lookup starting address from table; place in PC
- How to get back
  - Save address of next instruction (current PC) in R7

TRAP

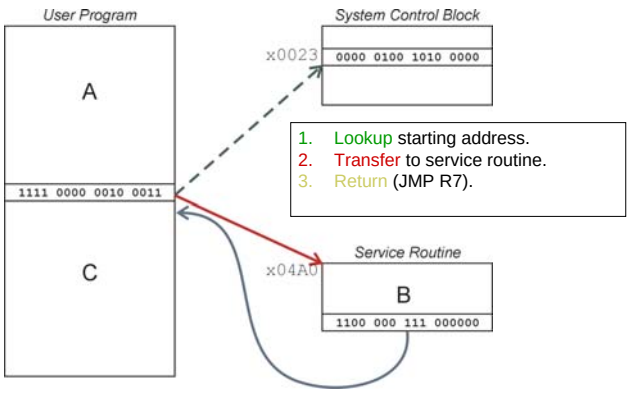


NOTE: PC has already been incremented during instruction fetch stage.

RET (JMP R7)

- How do we transfer control back to instruction following the TRAP?
- We saved old PC in R7.
  - JMP R7 gets us back to the user program at the right spot.
  - LC-3 assembly language lets us use RET (return) in place of "JMP R7".
- Must make sure that service routine does not change R7, or we won't know where to return.

TRAP Mechanism Operation



TRAP Example:

```
        .ORIG x3000
        LD   R2, TERM    ; Load negative ASCII '7'
        LD   R3, ASCII   ; Load ASCII difference
AGAIN   TRAP  x23        ; input character
        ADD  R1, R2, R0  ; Test for terminate
        BRz  EXIT        ; Exit if done
        ADD  R0, R0, R3  ; Change to lowercase
        TRAP  x21        ; Output to monitor...
        BRnzp  AGAIN     ; ... again and again...
TERM    .FILL xFFC9      ; -'7'
ASCII   .FILL x0020      ; lowercase bit
EXIT    TRAP  x25        ; halt
        .END
```

Saving and Restoring Registers

- Must save the value of a register if:
  - Its value will be destroyed by service routine, and
  - We will need to use the value after that action.
- Who saves?
  - Caller of service routine?
    - knows what it needs later, but may not know what gets altered by called routine
  - Called service routine?
    - knows what it alters, but does not know what will be needed later by calling routine

Example

```
          LEA   R3, Binary
          LD    R6, ASCII     ; char->digit template
          LD    R7, COUNT     ; initialize to 10
AGAIN     TRAP  x23           ; Get char
          ADD   R0, R0, R6    ; convert to number
          STR   R0, R3, #0    ; store number
          ADD   R3, R3, #1    ; incr pointer
          ADD   R7, R7, -1    ; decr counter
          BRp   AGAIN         ; more?
          BRnzp NEXT
ASCII     .FILL  xFFD0
COUNT     .FILL  #10
Binary    .BLKW #10
```

What's wrong with this routine?
What happens to R7?

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.

What about User Code?

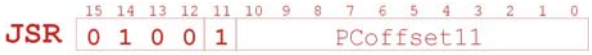
- Service routines provide three main functions:
  1. Shield programmers from system-specific details.
  2. Write frequently-used code just once.
  3. Protect system resources from malicious/clumsy programmers.
- Are there any reasons to provide the same functions for non-system (user) code?

Subroutines

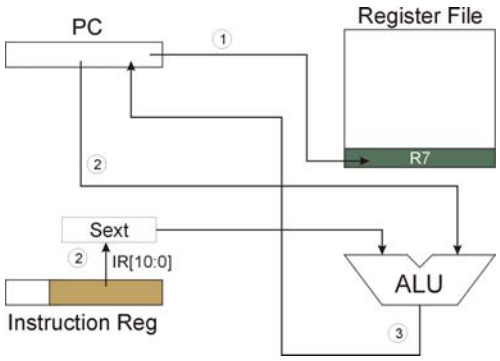
- A subroutine is a program fragment that:
  - lives in user space
  - performs a well-defined task
  - is invoked (called) by another user program
  - returns control to the calling program when finished
- Like a service routine, but not part of the OS
  - not concerned with protecting hardware resources
  - no special privilege required
- Reasons for subroutines:
  - reuse useful (and debugged!) code without having to keep typing it in
  - divide task among multiple programmers
  - use vendor-supplied *library* of useful routines

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]



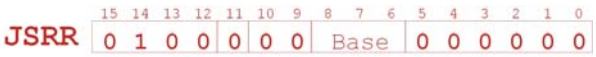
JSR



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

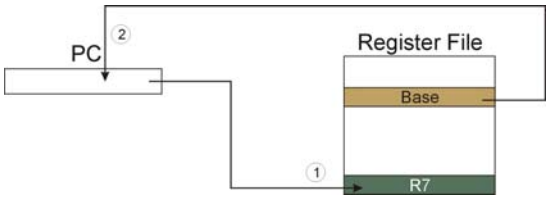
JSRR Instruction

- Just like JSR, except Register addressing mode.
  - Target address is Base Register
  - Bit 11 specifies addressing mode



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

JSRR



Returning from a Subroutine
RET (JMP R7) gets us back to the calling routine. just like TRAP

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

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 (within 1024 instructions):

```
; 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/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.
  - Examples:
    - In 2sComp routine, R0 is the number to be negated
    - In OUT service routine, R0 is the character to be printed.
    - In PUTS routine, R0 is address of string to be printed.
- 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:
    - In 2sComp routine, negated value is returned in R0.
    - In GETC service routine, character read from the keyboard is returned in R0.

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)

Saving and Restore Registers

- Since subroutines are just like service routines, we also need to save and restore registers, if needed.
- Generally use "callee-save" strategy, except for return values.
  - Save anything that the subroutine will alter internally that shouldn't be visible when the subroutine returns.
  - It's good practice to restore incoming arguments to their original values (unless overwritten by return value).
- Remember: You MUST save R7 if you call any other subroutine or service routine (TRAP).
  - Otherwise, you won't be able to return to caller.

User Defined Traps

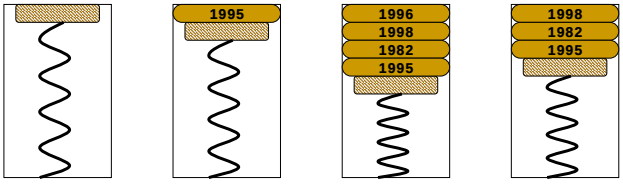
- Traps vs. Subroutines
  - Similarity
    - They are all functions
    - They have the same mechanism – call/ret (R7)
  - Difference
    - TRAP vs. JSR(R)
    - Stored at x0000 through x00FF in memory vs. Assigned
    - 'Global' vs. Close (JSR)
- Implementation of user defined traps (easy)
  1. Assign subroutine address to Trap table
  2. Use TRAP instruction
  3. DONE

Stacks

- A LIFO (last-in first-out) storage structure.
  - The first thing you put in is the last thing you take out.
  - The last thing you put in is the first thing you take out.
- This means of access is what defines a stack,not the specific implementation.
- Two main operations:
  1. PUSH: add an item to the stack
  2. POP: remove an item from the stack

A Physical Stack

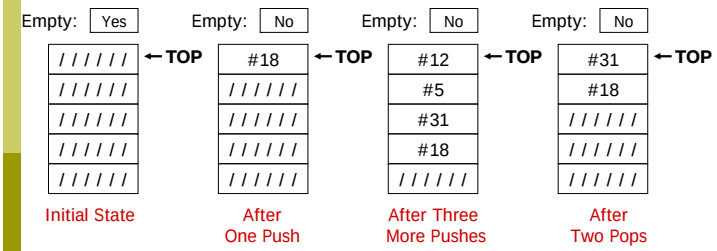
- Coin rest in the arm of an automobile



- First quarter out is the last quarter in.

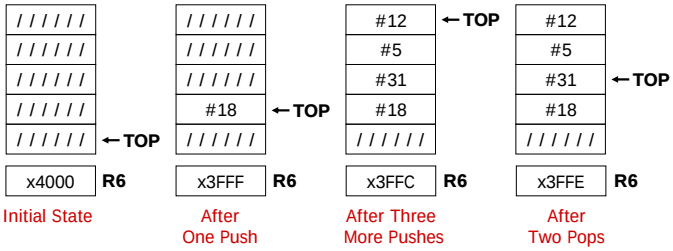
A Hardware Implementation

- Data items move between registers



A Software Implementation

- Data items don't move in memory, just our idea about there the TOP of the stack is.



By convention, R6 holds the Top of Stack (TOS) pointer.

Basic Push and Pop Code

- For our implementation, stack grows downward (when item added, TOS moves closer to 0)
  - Push

```
ADD  R6, R6, #-1 ; decrement stack ptr
STR  R0, R6, #0  ; store data (R0)
```

  - Pop

```
LDR  R0, R6, #0  ; load data from TOS
ADD  R6, R6, #1  ; decrement stack ptr
```

Pop with Underflow Detection

- If we try to pop too many items off the stack, an underflow condition occurs.
  - Check for underflow by checking TOS before removing data.
  - Return status code in R5 (0 for success, 1 for underflow)

```
POP     LD  R1, EMPTY  ; EMPTY = -x4000
        ADD R2, R6, R1 ; Compare stack pointer
        BRz FAIL       ; with x3FFF
        LDR R0, R6, #0
        ADD R6, R6, #1
        AND R5, R5, #0 ; SUCCESS: R5 = 0
        RET
FAIL    AND R5, R5, #0 ; FAIL: R5 = 1
        ADD R5, R5, #1
        RET
EMPTY .FILL xC000
```

Push with Overflow Detection

- If we try to push too many items onto the stack, an overflow condition occurs.
  - Check for underflow by checking TOS before adding data.
  - Return status code in R5 (0 for success, 1 for overflow)

```
PUSH  LD  R1, MAX    ; MAX = -x3FFB
      ADD R2, R6, R1 ; Compare stack pointer
      BRz FAIL       ; with x3FFF
      ADD R6, R6, #-1
      STR R0, R6, #0
      AND R5, R5, #0 ; SUCCESS: R5 = 0
      RET
FAIL  AND R5, R5, #0 ; FAIL: R5 = 1
      ADD R5, R5, #1
      RET
MAX   .FILL xC005
```

Interrupt-Driven I/O (Part 2)

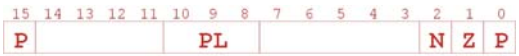
- Interrupts were introduced in Chapter 8.
  1. External device signals need to be serviced.
  2. Processor saves state and starts service routine.
  3. When finished, processor restores state and resumes program.

*Interrupt is an **unscripted subroutine call**, triggered by an external event.*

- Chapter 8 didn't explain how (2) and (3) occur, because it involves a stack.
- Now, we're ready…

Processor State

- What state is needed to completely capture the state of a running process?
- Processor Status Register
  - Privilege [15], Priority Level [10:8], Condition Codes [2:0]



- Program Counter
  - Pointer to next instruction to be executed.
- Registers
  - All temporary state of the process that's not stored in memory.

Where to Save Processor State?

- Can't use registers.
  - Programmer doesn't know when interrupt might occur, so she can't prepare by saving critical registers.
  - When resuming, need to restore state exactly as it was.
- Memory allocated by service routine?
  - Must save state before invoking routine, so we wouldn't know where.
  - Also, interrupts may be nested – that is, an interrupt service routine might also get interrupted!

*Supervisor Stack

- A special region of memory used as the stack for interrupt service routines.
  - Initial Supervisor Stack Pointer (SSP) stored in Saved.SSP.
  - Another register for storing User Stack Pointer (USP): Saved.USP.
- Want to use R6 as stack pointer.
  - So that our PUSH/POP routines still work.
- When switching from User mode to Supervisor mode (as result of interrupt), save R6 to Saved.USP.

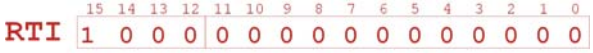
*Invoking the Service Routine – The Details

1. If Priv = 1 (user), Saved.USP = R6, then R6 = Saved.SSP.
2. Push PSR and PC to Supervisor Stack.
3. Set PSR[15] = 0 (supervisor mode).
4. Set PSR[10:8] = priority of interrupt being serviced.
5. Set PSR[2:0] = 0.
6. Set MAR = x01vv, where vv = 8-bit interrupt vector provided by interrupting device (e.g., keyboard = x80).
7. Load memory location (M[x01vv]) into MDR.
8. Set PC = MDR; now first instruction of ISR will be fetched.

Note: This all happens between the STORE RESULT of the last user instruction and the FETCH of the first ISR instruction.

*Returning from Interrupt

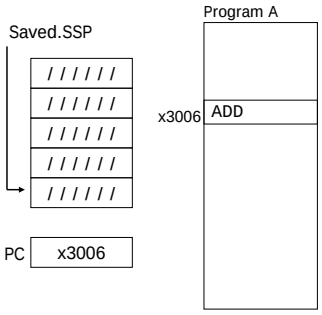
- Special instruction – RTI – that restores state.



1. Pop PC from supervisor stack. (PC = M[R6]; R6 = R6 + 1)
2. Pop PSR from supervisor stack. (PSR = M[R6]; R6 = R6 + 1)
3. If PSR[15] = 1, R6 = Saved.USP. (If going back to user mode, need to restore User Stack Pointer.)

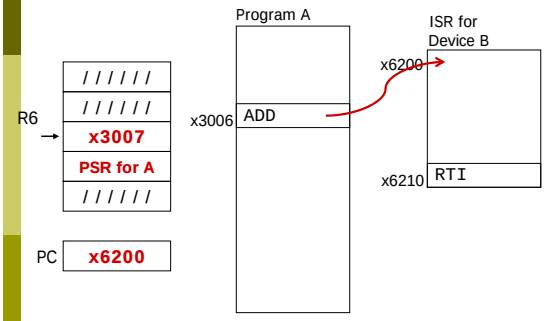
- RTI is a privileged instruction.
  - Can only be executed in Supervisor Mode.
  - If executed in User Mode, causes an exception. (More about that later.)

Example (1)



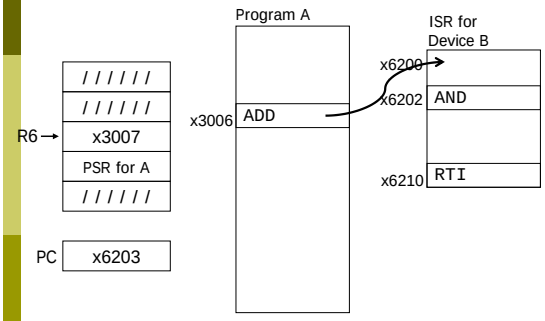
Executing ADD at location x3006 when Device B interrupts.

Example (2)



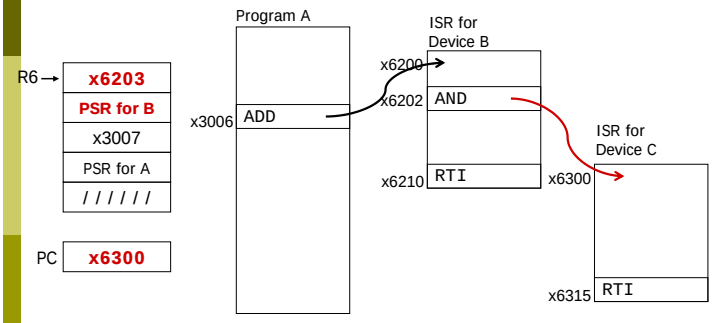
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to Device B service routine (at x6200).

Example (3)



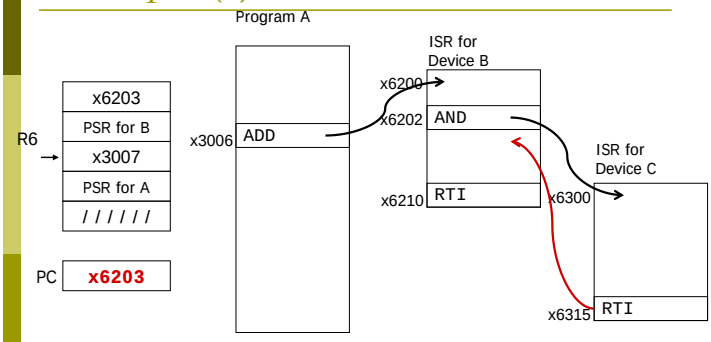
Executing AND at x6202 when Device C interrupts.

Example (4)



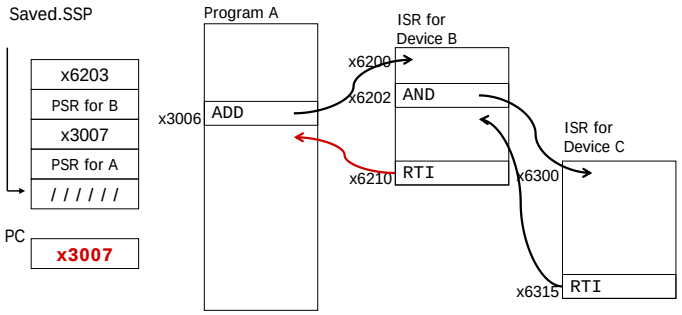
Push PSR and PC onto stack, then transfer to Device C service routine (at x6300).

Example (5)



Execute RTI at x6315; pop PC and PSR from stack.

Example (6)



Execute RTI at x6210; pop PSR and PC from stack.
Restore R6. Continue Program A as if nothing happened.

LC-3 Summary

- Compare to TinyCPU
  - 16bit, 15 instructions, 8 registers
- Instructions
  - **ADD/AND** with 2 styles (register and number)
  - Complete logic with **NOT**
  - **BR** and **BRz** (skip BRn or BRp)
- Assembly Programming
  - Identical Logic Approach – conditional jump
  - Identical 3 constructs
  - .ORIG, .END, .FILL, .BLKW, .STRINGZ

LC-3 Summary (cont)

- Two More Memory Mode
  - Offset <256 bytes – **LD/ST, LEA**
  - Offset > 256 buyes – **LDI/STI, LDR/STR**
- Subroutine and Trap
  - Return address in R7 – **JSR(R), RET/JMP**
  - Global, defined in trap vectors – **TRAP**
- Interrupt and Stack
  - Polling vs. Interrupt
  - Push/Pop, Under/Overflow
  - **RTI**