Interrupt-Driven I/O (Reprise)

Timing of I/O controlled by *device*

- Tells processor when something interesting happens
  - Example: when character is entered on keyboard
  - Example: when monitor is ready for next character
  - Example: when block has been transferred from disk to memory
- Processor ***interrupts*** its normal instruction processing and executes a service routine (like a TRAP)
  - Figure out what device is causing the interrupt
  - Execute routine to deal with event
  - Resume execution
- No need for processor to poll device
  - Can perform other useful work

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

How is Interrupt Signaled?

External interrupt signal: INT

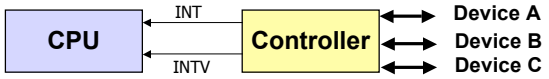
- Device sets INT=1 when it wants to cause an interrupt

Interrupt vector: INTV

- 8-bit signal for device to identify itself
- Also used as entry into Interrupt Vector Table, which gives starting address of *Interrupt Service Routine (ISR)*
  - Just like Trap Vector Table and Trap Service Routine
  - TVT: x0000 to x00FF
  - IVT: x0100 to x01FF

What if more than one device wants to interrupt?

- External logic controls which one gets to drive signals



How Does Processor Handle It?

Examines INT signal just before starting FETCH phase

- If INT=1, don't fetch next instruction
- Instead
  - Save state (PC, PSR (privilege and CCs)) on stack
  - Update PSR (set privilege bit)
  - Index INTV into IVT to get start address of ISR (put in PC)

After service routine

- RTI instruction restores PSR and PC from stack
- Need a different return instruction, because
  - RET gets PC from R7 and doesn't update PSR
  - RTT gets PC from R7 and always clears privilege bit in PSR

Processor only checks between STORE and FETCH phases -- Why?

Supervisor Mode and the Stack

Problem

- PC and PSR shouldn't be saved on user stack
- What if R6 is uninitialized?
- What if user has set R6 to refer to OS memory?
- User could see OS data (when trap returns)

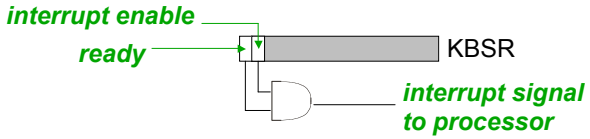
Solution

- Create two versions of R6 (stack pointer) in register file
  - One is *user stack pointer* (what we've been using all along)
  - The other is *supervisor stack pointer*
- Extra register file logic selects the appropriate register based on privilege bit in current PSR
- Bottom line: OS code always uses its own stack

More Control

At the device

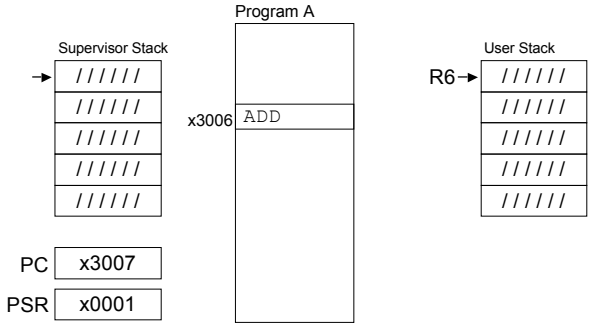
- Control register has "Interrupt Enable" bit
- Must be set for interrupt to be generated



At the processor

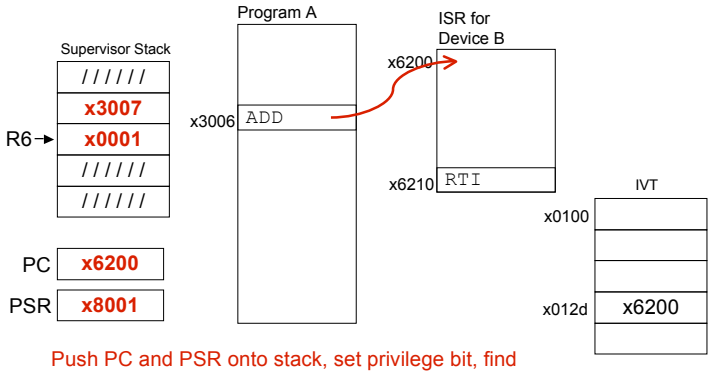
- Sometimes have "Interrupt Mask" register (LC-3 does not)
- When set, processor ignores INT signal
- Why?
  - Example: may not want to be interrupted while in ISR

Example (1)



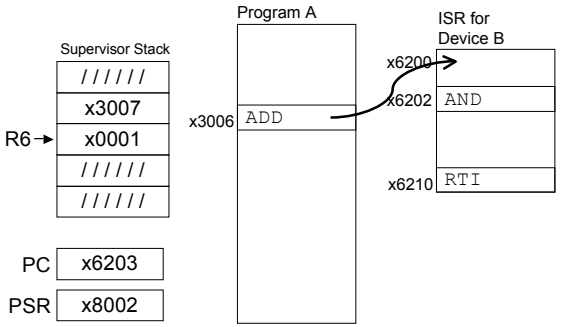
Executing ADD at location x3006 when Device B interrupts (INTV = x1d)

Example (2)



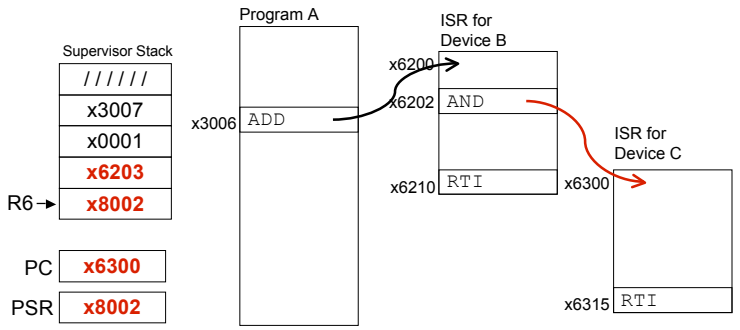
Push PC and PSR onto stack, set privilege bit, find Device B (INTV=x2d) service routine address in IVT, and transfer control

Example (3)



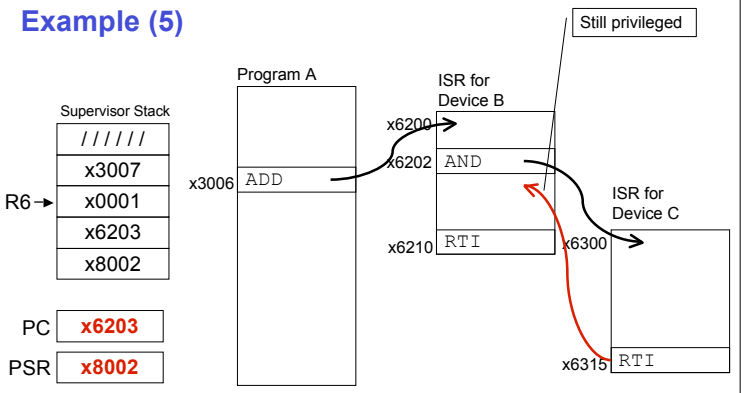
Executing AND at x6202 when Device C interrupts

Example (4)



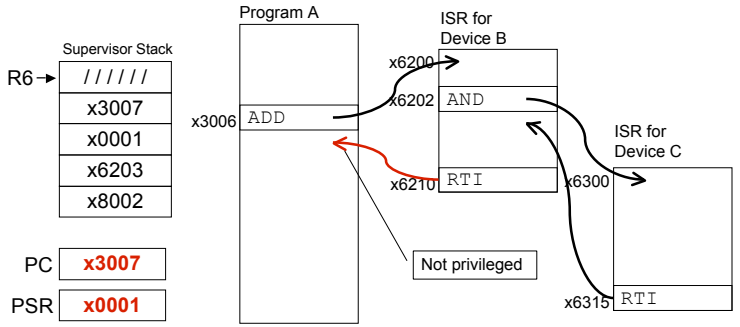
Push PC and PSR onto stack, set privilege bit, then transfer to Device C service routine (at x6300)

Example (5)



Execute RTI at x6315; pop PSR and PC from stack

Example (6)



Execute RTI at x6210; pop PSR and PC from stack; continue Program A as if nothing happened!

Questions

What do condition codes look like after interrupt is serviced? PSR?

Who saves registers during the servicing of interrupt? Where are they saved?

What does R7 look like after interrupt is serviced?

On interrupt, why doesn't the processor save return address in R7?

Do interrupt service routines have return values?

Bottom line: program can't tell when it is interrupted!

Data Type Conversion

I/O

- **Keyboard input routines read ASCII characters (not binary values)**
- **Console output routines write ASCII ('s' not "x73")**

Consider this program

```
TRAP  x23          ; input from keyboard
ADD   R1, R0, #0   ; move to R1
TRAP  x23          ; input from keyboard
ADD   R0, R1, R0   ; add two inputs
TRAP  x21          ; display result
TRAP  x25          ; HALT
```

Input: '2' and '3'
Output: 'e'

Why?

- **ASCII '2' (x32) + ASCII '3' (x33) = ASCII 'e' (x65)**

ASCII to Binary

Single digit numbers are trivial (subtract x30)

- E.g., '7' is ASCII x37, x37 - x30 = x7

Input

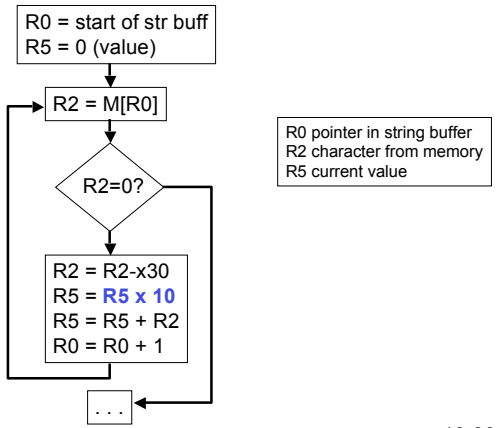
- **Assume we've read three ASCII digits (*e.g.*, "259") into a memory buffer**

x32	'2'
x35	'5'
x39	'9'
x0	

How do we convert this to a *number* we can use?

- **Convert first character to digit (subtract x30) and multiply by 100**
- **Convert second character to digit and multiply by 10**
- **Convert third character to digit**
- **Add the three digits together**

ASCII to Binary Conversion Algorithm



Multiplication

How can we multiply a number by 10?

Approach 0

- **Use the MUL instruction**

Approach 1

- **Add <number> to itself 10 times**

Approach 2

- **Add 10 to itself <number> times (better if number < 10)**

Approach 3

- **Look it up! Only practical if number of multiplicands is small**

Lookup table in memory

#0	0x10
#10	1x10
#20	2x10
#30	3x10
#40	4x10
#50	5x10
. . .	

Code for Lookup Table

```
; mult. R0 by 10, using lookup table;(product in R0)
          LEA  R1, Lookup10    ; R1 = table base
          ADD  R1, R1, R0      ; add index (R0)
          LDR  R0, R1, #0      ; load from M[R1]
          ...
Lookup10  .FILL #0    ; entry 0
          .FILL #10 ; entry 1
          .FILL #20 ; entry 2
          .FILL #30 ; entry 3
          .FILL #40 ; entry 4
          .FILL #50 ; entry 5
          .FILL #60 ; entry 6
          .FILL #70 ; entry 7
          .FILL #80 ; entry 8
          .FILL #90 ; entry 9
```


ASCII to Binary Conversion

R0 pointer in string buffer
R2 character from memory
R5 current value

```
ASCIItoBin
        LEA     R0, BUF          ; R0 = string pointer
        AND     R5, R5, #0       ; R5 = 0 (current value)
L2
        LDR     R2, R0, #0       ; get character from buffer
        BRz     DONE             ; done, if end-of-string
        LD      R4, NegASCIIOffset ; R4 = -x30
        ADD     R2, R2, R4       ; convert char to number
        MUL     R5, R5, #10      ; mult current value by 10
        ADD     R5, R5, R2       ; add number to cur. val.
        ADD     R0, R0, #1       ; decrement pointer
        BR      L2               ; loop
DONE
        HALT


BUF             .STRINGZ   "32767"
NegASCIIOffset  .FILL      xFFD0
```


Binary to ASCII Conversion

Converting a register value to ASCII string

Instead of multiplying, we need to divide by 10

- Why wouldn't we use a lookup table for this problem?
- Instead: subtract 100 repeatedly from number to divide

To simplify

- Check whether number is negative
- Write sign character (+ or -) to buffer and take abs. val.
- Now we only have to deal with positive values

Binary to ASCII Conversion Code (part 1 of 3)

```
; R0 is between -999 and +999.
; Put sign character in ASCIIBUF, followed by three
; ASCII digit characters.
BinaryToASCII  LEA R1, ASCIIBUF  ; pt to result string
               ADD R0, R0, #0    ; test sign of value
               BRn NegSign
               LD  R2, ASCIIplus ; store '+'
               STR R2, R1, #0
               BR  Begin100
NegSign        LD  R2, ASCIIneg  ; store '-'
               STR R2, R1, #0
               NOT R0, R0        ; convert value to pos
               ADD R0, R0, #1
```

Conversion (2 of 3)

```
Begin100        LD  R2, ASCIIoffset
                LD  R3, Neg100
Loop100         ADD R0, R0, R3
                BRn End100
                ADD R2, R2, #1   ; add one to digit
                BR  Loop100
End100          STR R2, R1, #1   ; store ASCII 100's digit
                LD  R3, Pos100
                ADD R0, R0, R3   ; restore last subtract
;
                LD  R2, ASCIIoffset
                LD  R3, Neg10
Loop10          ADD R0, R0, R3
                BRn End10
                ADD R2, R2, #1   ; add one to digit
                BR  Loop10
```


Conversion Code (3 of 3)

```
End10           STR R2, R1, #2   ; store ASCII 10's digit
                ADD R0, R0, #10 ; restore last subtract
;
                LD  R2, ASCIIoffset
                ADD R2, R2, R0   ; convert one's digit
                STR R2, R1, #3   ; store one's digit
                RET

;
ASCIIplus    .FILL x2B    ; plus sign
ASCIIneg     .FILL x2D    ; neg sign
ASCIIoffset  .FILL x30    ; zero
Neg100       .FILL xFF9C ; -100
Pos100       .FILL 100
Neg10        .FILL xFFF6 ; -10
```