

Chapter 10

The Stack and More...

Based on slides © McGraw-Hill
 Additional material © 2004/2005 Lewis/Martin
 Modified by Diana Palsetia

Stacks

Concept

A last-in first-out (LIFO) 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
- Not like an array, where you can access any item based on an index

Two main operations

PUSH: add an item to the stack

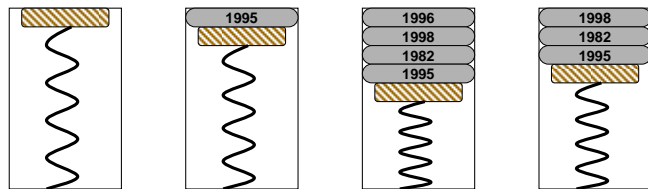
POP: remove an item from the stack

CIT 593

10-2

A Physical Stack

Coin holder



Initial State

After One Push

After Three More Pushes

After One Pop

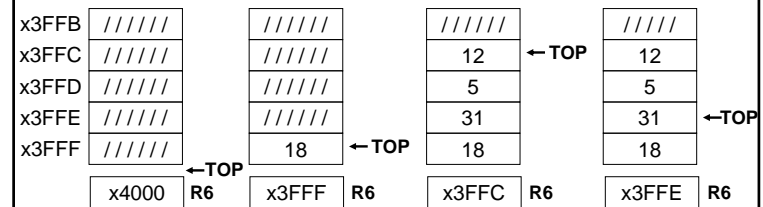
Last quarter in is the first quarter out (LIFO)

CIT 593

10-3

A Software Stack implemented in Memory

- Assume section x3FFFB to x3FFF in mem used for stack
- Data items don't move in memory, just our idea about where TOP of the stack is (a register is used to keep track of the location)



Initial State

After One Push

After Three More Pushes

After Two Pops

By convention, in LC3 R6 holds the Top of Stack (TOS) pointer

CIT 593

10-4

Basic Push and Pop Code

Push

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

Pop

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

Note

- Stacks can grow in either direction (toward higher address or toward lower addresses)

CIT 593

10-5

What happens when we run out of space?

Overflow

- When we have no more locations and we try to push some data on the stack

Underflow

- The stack is empty and we try to pop the data

Some how the programmer needs to learn about the overflow/underflow problem

CIT 593

10-6

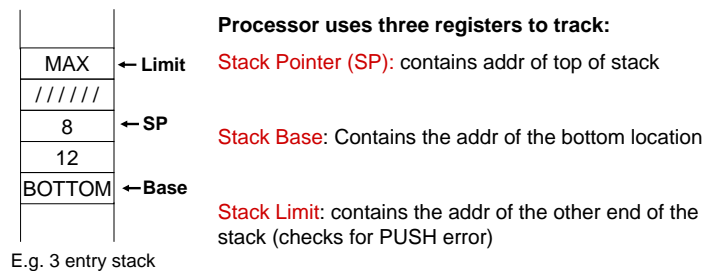
POP routine w/ Underflow Detection

```
POP LD R1, EMPTY ;R1 = -x4000
ADD R2, R6, R1
BRz Failure ;check whether R6 = x4000
LDR R0, R6, #0 ;POP value of the stack
ADD R6, R6, #1 ;update the stack pointer
RET
Failure AND R5, R5, #0
ADD R5, R5, #1 ;R5 = 1 indicates that the POP was
;not successful
RET
Empty .FILL xC000 ; - x4000
```

CIT 593

10-7

General Implementation



CIT 593

10-8

Uses of Stack

- Initiate and service interrupts (Interrupt Driven I/O)
 - Use as Temporary storage
 - Computers without registers store temporary values during computation on a stack
 - Passing arguments & routines for subroutine calls
 - Instead passing them through registers
- More on this when we learn C (chapter 12 & 14)
- Subroutines with nested subroutine a.k.a recursive subroutines
 - Allocating space for local variables inside a subroutine

CIT 593

10-9

Use 1: Interrupt-Driven I/O

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
- 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*

CIT 593

10-10

How is Interrupt Signaled?

External interrupt signal: INT

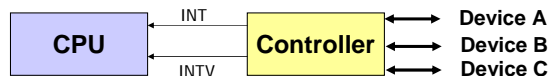
- Device sets INT=1 when it wants to cause an interrupt

Interrupt vector: INTV

- 8-bit signal for device to identify itself
 - Also used as index into Interrupt Vector Table, which gives starting address of *Interrupt Service Routine (ISR)*
 - Just like Trap Vector Table and Trap Service Routine
- In LC3
- Interrupt Vector Table (IVT): x0100 to x01FF

What if more than one device wants to interrupt?

- External logic controls which one gets to drive signals



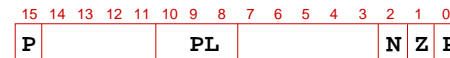
CIT 593

10-11

How Does Processor Handle an Interrupt?

Examines INT (interrupt) signal just before starting FETCH phase

- If INT=1, don't fetch next instruction
- Instead
 - Save state (PC, PSR (privilege and CCs)) on *Supervisor stack*
 - Update PSR (set privilege bit i.e. PSR[15] = 1)



- Index INTV into IVT to get start address of ISR (put in PC)

After service routine

- RTI (Return from Interrupt) instruction restores PSR and PC from *Supervisor stack*

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

CIT 593

10-12

Supervisor Stack

Why have a separate supervisor stack to save PC and PSR instead of user stack?

- Basically to provide protection & isolation from users accessing operating system data

Implementation

- Have 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

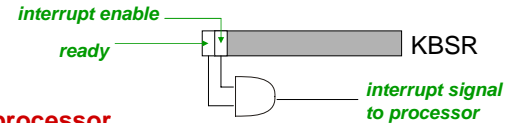
CIT 593

10-13

More Control

At the device

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



At the processor

- Processor can choose to ignore Interrupt (INT) signal
- Why? Example: may not want to be interrupted while in ISR

Achieved by

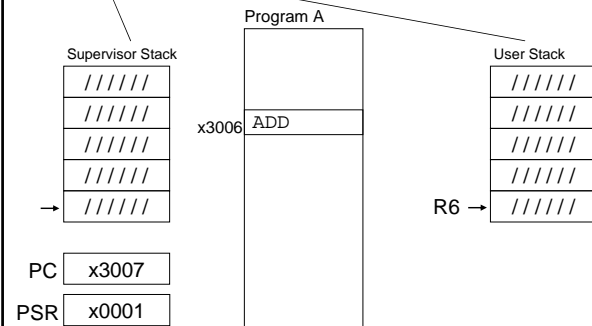
- "Interrupt enable bit" in device register is set by processor
 - Depending on whether processor wants to give I/O device service

CIT 593

10-14

Example (1)

Different sections in memory

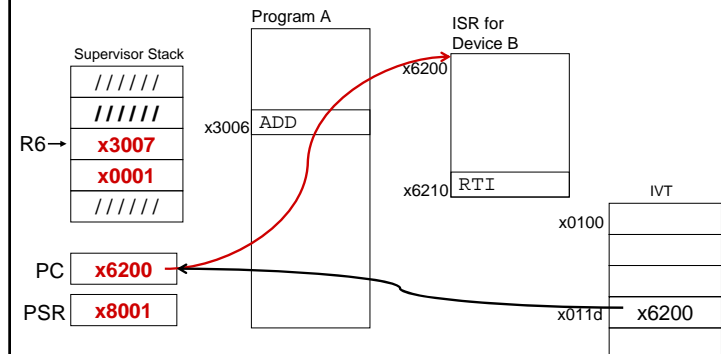


Executing ADD at location x3006 when Device B interrupts (INTV = x1d)

CIT 593

10-15

Example (2)

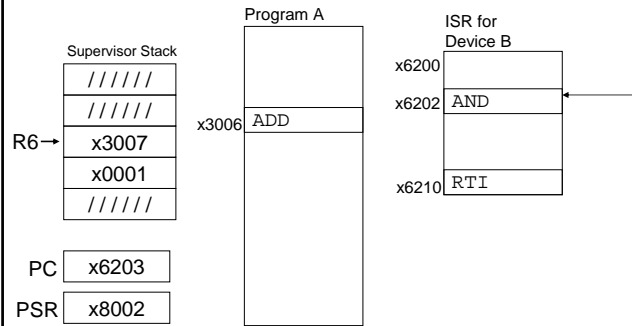


Push PC and PSR onto stack, set privilege bit, find Device B (INTV=x1d) service routine address in IVT, and transfer control

CIT 593

10-16

Example (3)

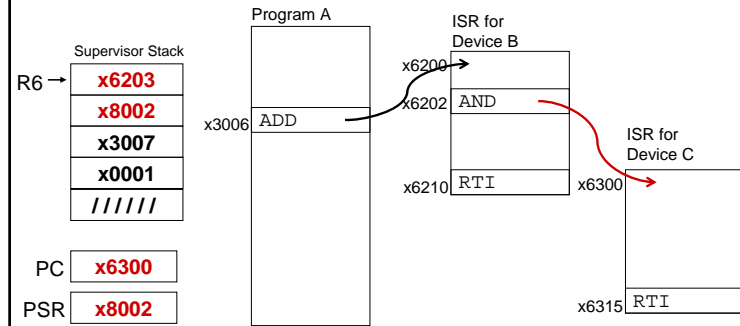


Executing AND instruction of Interrupt Routine for device B at x6202 when Device C interrupts

CIT 593

10-17

Example (4)

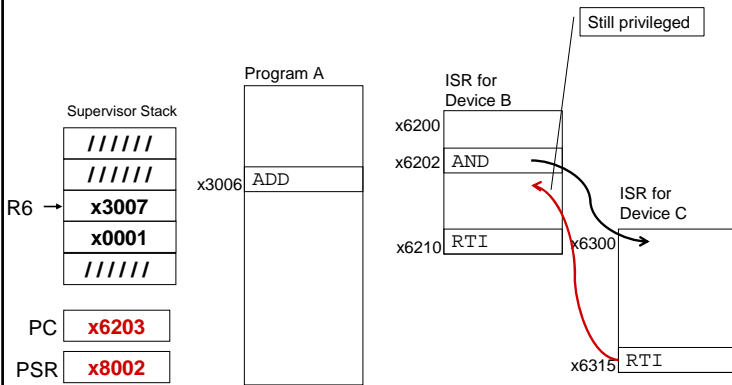


Push PC and PSR onto stack, set privilege bit, then transfer to Device C service routine (at x6300)

CIT 593

10-18

Example (5)

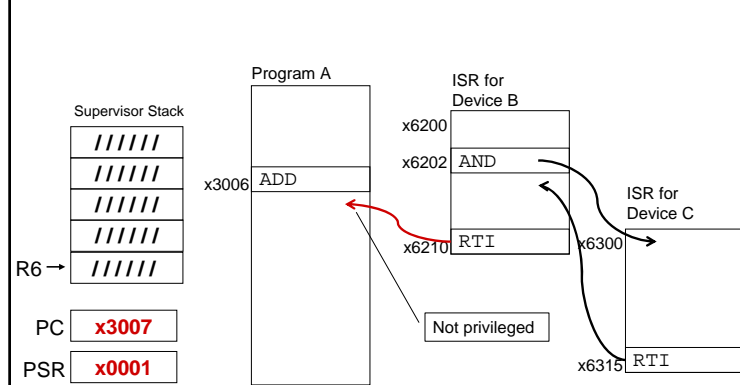


Execute RTI at x6315; pop PSR and PC from stack

CIT 593

10-19

Example (6)



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

CIT 593

10-20

Questions

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

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

Why do we need to save the condition codes before an interrupt is serviced?

Remember: program can't tell when it is interrupted!

CIT 593

10-21

Use 2: Temporary Storage

LC3 three address machine because it specifies all 3 locations (i.e. 1 Dest and 2 Src). E.g. **ADD R1, R2, R1**

Some ISAs, are **zero address machines or stack machine**

- They use **stack** for Dst and Src operands and the instruction **does not explicitly** specify them
- In an all memory machine, some location in memory holds the address of the top of the stack. The address stored is called **Stack Pointer (SP)**

CIT 593

10-22

Example: Zero Address ISA

PUSH Instruction

PUSH address

- adjusts the stack pointer (internally done, just like how TRAP instruction in LC3 saves PC in R7) and pushes value onto the stack
- Programmer can only read the stack pointer (just like in LC3 the programmer can read the condition code registers).
- **POP** is similar to PUSH, except that value is removed from top of the stack and put back into memory location other than stack space. (the location is specified by the address field)

ADD Instruction

ADD 0.....0

- Takes the two values out the stack (2 POP instructions) and then gives that to ALU to compute the result

CIT 593

10-23

Stack Machine: Multiply Add Example

Want to compute $E = (A + B).(C + D)$. Let say $A = 25$, $B = 17$, $C = 3$ and $D = 2$

Stack Implementation

Push A (take a data from memory and push it on to stack)
Push B
Add (2 POP's, so that operands can be given to ALU, and then PUSHed back)
Push C
Push D
Add
Mult
POP E (pop it off stack and store somewhere in memory)

Register Implementation

```
LD R0, A
LD R1, B
ADD R0, R0, R1
LD R2, C
LD R3, D
ADD R2, R2, R3
MUL R0, R0, R2
ST R0, E
```

Does everyone see the usefulness of register file ??

CIT 593

10-24

Use 3: Passing arguments & returns

Use the stack for **arguments** & **return** values instead of registers

- Calling routine pushes arguments on the stack and calls the subroutine
- Called routine pops the passed arguments from the stack and pushes any return value onto the stack
- Finally, the calling routine then retrieves the return value(s) from the stack and continues execution

Why would this be helpful?

- This is especially helpful if there are more than 4 arguments and two returns

CIT 593

10-25

Example: Passing Arguments w/ Stack

;Calling a drawRect(w, l) subroutine
;will need to save restore registers according to call
;convention (not shown here)

```

PUSH ← {
        ADD R0, R0, #5    ;R0 = width
        STR, R0, R6, #0  ;Push arg width onto stack (R6 = SP)
        ADD R6, R6, #-1  ;adjust stack pointer
        ADD R1, R1, #10  ;R1 = length
        STR, R1, R6, #0  ;Push arg length onto stack
        ADD R6, R6, #-1  ;adjust stack pointer
        JSR drawRect
    
```

```

drawRect POP          ;POPs the arg length
        ADD R3, R0, #0 ;copy it some temp reg
        POP           ;PoPs the arg width
        ADD R4, R0, #0 ;Copy it to some temp reg
    
```

CIT 593

10-26

Subroutines with Recursion

A subroutine that calls itself is known as **recursive subroutine**

```

Main   . . .
      JSR   Foo
Next   . . .
      HALT

Foo    ST    R7, SaveR7
      AND  R0, R0, #0
      . . .
      JSR   Foo
After  . . .
      LD   R7, SaveR7
      RET

Save7  .FILL #0
Counter .FILL #0

      .END
    
```

Problem:

- First call to Foo (SaveR7 contains address of Next)
- Second call to Foo (SaveR7 contains address of After)
- First return from Foo (returns to After)
- Second return from Foo (returns to After again!!!)

CIT 593

10-27

Solution

For each subroutine call, need a mechanism to distinguish its invocation

- This is known as *activation record*

Approach

- Allocate new *activation record* on a **stack** whenever a subroutine is called
- Subroutine uses its own activation record to hold invocation-specific data (e.g., local variables, saved registers)

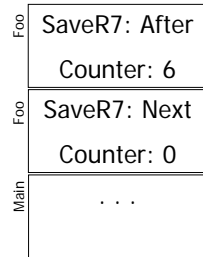
CIT 593

10-28

Big Picture

Each subroutine invocation gets its own activation record
...details in chapter 12 &14

```
Main . . .  
      JSR   Foo  
Next . . .  
      HALT  
  
Foo   ST     R7, SaveR7  
      AND   R0, R0, #0  
      ST     R0, Counter  
      . . .  
      JSR   Foo  
After . . .  
      LD     R7, SaveR7  
      RET  
SaveR7 .FILL #0  
Counter .FILL #0  
  
      .END
```



Counter is local variable of foo()

CIT 593

10-29

Next Time

Thursday 19th Oct, Midterm Exam

Next half of the semester: C language (4 weeks)
Operating System

CIT 593

10-30