

Chapter 10 And, Finally... The Stack

Original slides from Gregory Byrd, North
Carolina State University
Modified slides by Chris Wilcox,
Colorado State University

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Stack: An Abstract Data Type

- ◆ An important abstraction that you will encounter in many applications.
- ◆ The fundamental model for execution of C, Java, Fortran, and many other languages.
- ◆ We will describe three uses of the stack:
 - **Interrupt-Driven I/O**
 - The rest of the story...
 - **Evaluating arithmetic expressions**
 - Store intermediate results on stack instead of in registers
 - **Data type conversion**
 - 2's comp binary to ASCII strings

CS 270 - Fall Semester 2014

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Stacks

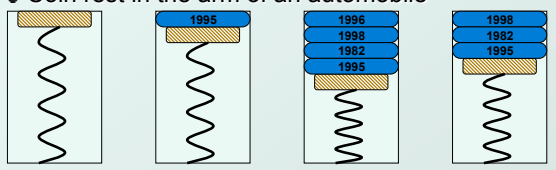
- ◆ A LIFO (last-in first-out) storage structure.
 - The **first** thing you put in is the **last** thing you take out.
 - The **last** thing you put in is the **first** thing you take out.
- ◆ This means of access is what defines a stack, not the specific implementation.
- ◆ Two main operations:
 -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

CS 270 - Fall Semester 2014

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

A Physical Stack

- ◆ Coin rest in the arm of an automobile

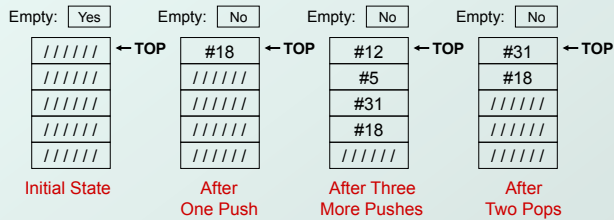


Initial State After One Push After Three More Pushes After One Pop

CS 270 - Fall Semester 2014

A Hardware Implementation

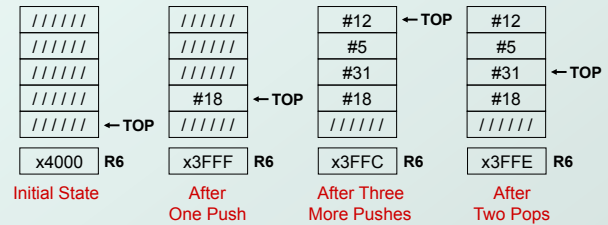
◆ Data items move between registers



CS 270 - Fall Semester 2014

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.

CS 270 - Fall Semester 2014

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 pointer
STR R0, R6, #0  ; store data (R0) to TOS
```

POP

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

CS 270 - Fall Semester 2014

Pop with Underflow Detection

- ◆ If we try to pop too many items off the stack, an **underflow** condition occurs.
 - Check for underflow 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
```

CS 270 - Fall Semester 2014

Push with Overflow Detection

- ◆ If we try to push too many items onto the stack, an **overflow** condition occurs.
 - Check for underflow 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

```

CS 270 - Fall Semester 2014

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...

CS 270 - Fall Semester 2014

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]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P							PL						N	Z	P

- ◆ **Program Counter**
 - Pointer to next instruction to be executed.
- ◆ **Registers**
 - Temporary process state that's not stored in memory.

CS 270 - Fall Semester 2014

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!
- ◆ **Use a stack!**
 - Location of stack “hard-wired”.
 - Push state to save, pop to restore.

CS 270 - Fall Semester 2014

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.

CS 270 - Fall Semester 2014

Invoking the Service Routine (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.**

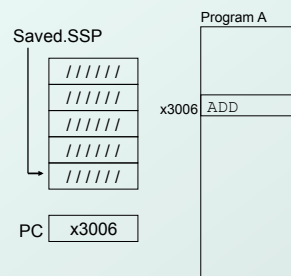
CS 270 - Fall Semester 2014

Returning from Interrupt

- ◆ Special instruction – RTI – that restores state.
- | | | | | | | | | | | | | | | | | |
|------------|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| RTI | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
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 going back to user mode, need to restore User Stack Pointer:
(if PSR[15] = 1, R6 = Saved.USP)
- ◆ RTI is a privileged instruction.
 - Can only be executed in Supervisor Mode.
 - If executed in User Mode, causes an exception. (More about that later.)

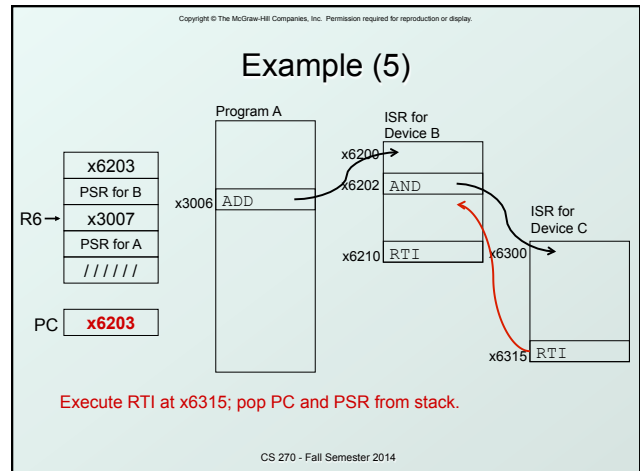
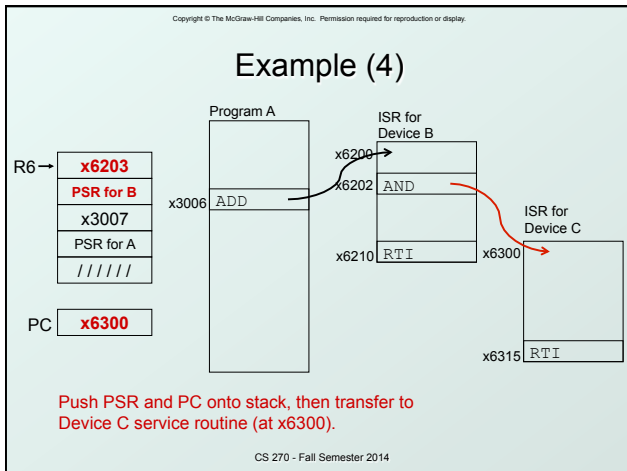
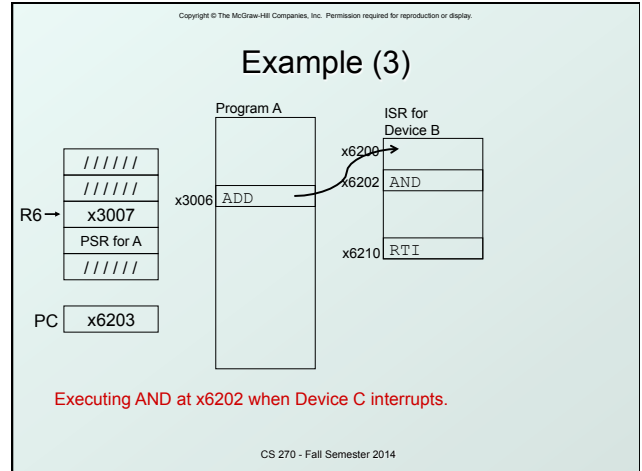
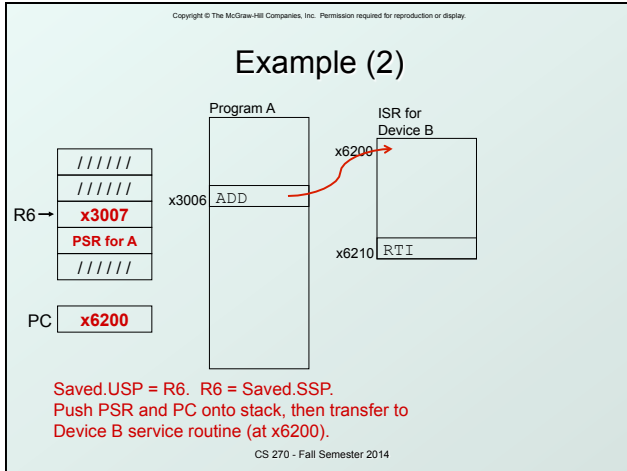
CS 270 - Fall Semester 2014

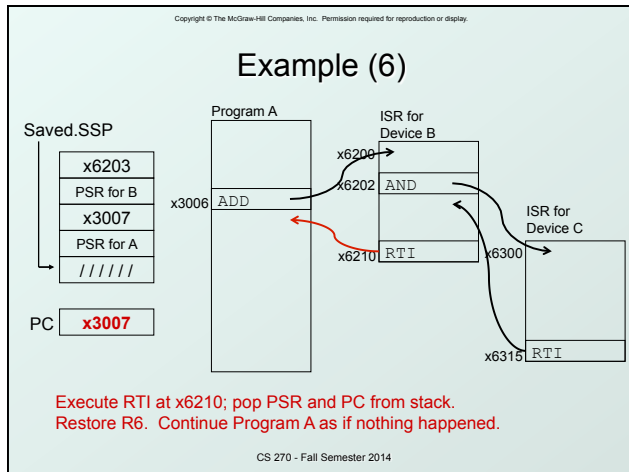
Example (1)



Executing ADD at location x3006 when Device B interrupts.

CS 270 - Fall Semester 2014





Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Exception: Internal Interrupt

- When something unexpected happens inside the processor, it may cause an exception.
- Examples:
 - Privileged operation (e.g., RTI in user mode)
 - Executing an illegal opcode
 - Divide by zero
 - Accessing an illegal address (e.g., protected system memory)
- Handled just like an interrupt
 - Vector is determined internally by type of exception
 - Priority is the same as running program

CS 270 - Fall Semester 2014

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Arithmetic Using a Stack

- Instead of registers, some ISA's use a stack for source/destination ops (**zero-address** machine).
 - Example: ADD instruction pops two numbers from the stack, adds them, and pushes the result to the stack.

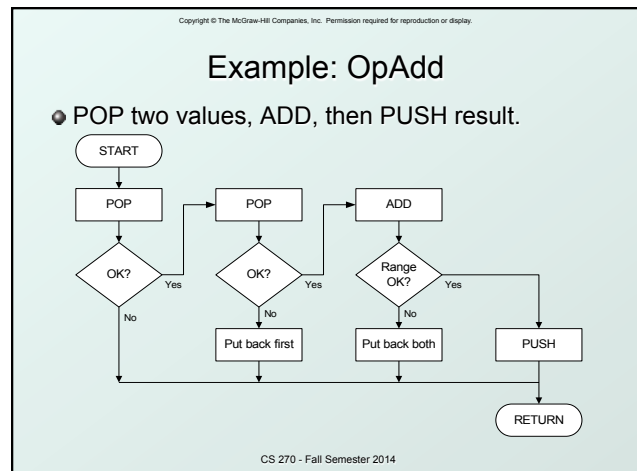
Evaluating $(A+B) \cdot (C+D)$ using a stack:

- push A
- push B
- ADD
- push C
- push D
- ADD
- MULTIPLY
- pop Result

Why use a stack?

- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using FIFO data structure.

CS 270 - Fall Semester 2014



Example: OpAdd

```
OpAdd JSR POP      ; Get first operand.
      ADD R5,R5,#0 ; Check for POP success.
      BRp Exit    ; If error, bail.
      ADD R1,R0,#0 ; Make room for second.
      JSR POP      ; Get second operand.
      ADD R5,R5,#0 ; Check for POP success.
      BRp Restore1 ; If err, restore & bail.
      ADD R0,R0,R1 ; Compute sum.
      JSR RangeCheck ; Check size.
      BRp Restore2 ; If err, restore & bail.
      JSR PUSH     ; Push sum onto stack.
      RET
Restore2 ADD R6,R6,#-1 ; undo first POP
Restore1 ADD R6,R6,#-1 ; undo second POP
Exit RET
```

CS 270 - Fall Semester 2014

Data Type Conversion

- Keyboard input routines read ASCII characters, not binary values, output routines write ASCII.

- Consider this program:

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

- User inputs **2** and **3** -- what happens?
 - Result displayed: **e**
 - Why? ASCII '2' (**x32**) + ASCII '3' (**x33**) = ASCII 'e' (**x65**)

CS 270 - Fall Semester 2014

ASCII to Binary

- Useful to deal with multi-digit decimal numbers
- Assume we've read three ASCII digits (e.g., "259") into memory.

x32	'2'
x35	'5'
x39	'9'
- How do we convert this to a number we can use?
 - Convert first character to digit and multiply by 100.
 - Convert second character to digit and multiply by 10.
 - Convert third character to digit.
 - Add the three digits together.

CS 270 - Fall Semester 2014

Multiplication via a Lookup Table

- How can we multiply a number by 100?**
 - One approach: Add number to itself 100 times.
 - Another approach: Add 100 to itself <number> times. (Better if number < 100.)
- Since we have a small range of numbers (0-9), use number as an index into a lookup table.**
 - Entry 0: 0 x 100 = **0**
 - Entry 1: 1 x 100 = **100**
 - Entry 2: 2 x 100 = **200**
 - Entry 3: 3 x 100 = **300**
 - etc.

CS 270 - Fall Semester 2014

Code for Lookup Table

```

; multiply R0 by 100, using lookup table
;
    LEA R1, Lookup100 ; R1 = table base
    ADD R1, R1, R0     ; add index (R0)
    LDR R0, R1, #0     ; load from M[R1]

Lookup100 .FILL 0 ; entry 0
          .FILL 100 ; entry 1
          .FILL 200 ; entry 2
          .FILL 300 ; entry 3
          .FILL 400 ; entry 4
          .FILL 500 ; entry 5
          .FILL 600 ; entry 6
          .FILL 700 ; entry 7
          .FILL 800 ; entry 8
          .FILL 900 ; entry 9

```

CS 270 - Fall Semester 2014

Complete Conversion Routine (1 of 3)

```

; Three-digit buffer at ASCIIIBUF.
; R1 tells how many digits to convert.
; Put resulting decimal number in R0.

```

ASCIItoBinary

```

    AND R0, R0, #0 ; clear result
    ADD R1, R1, #0 ; test # digits
    BRz DoneAtoB ; done if no digits

    LD R3, NegZero ; R3 = -x30
    LEA R2, ASCIIIBUF
    ADD R2, R2, R1
    ADD R2, R2, #-1 ; points to ones digit

    LDR R4, R2, #0 ; load digit
    ADD R4, R4, R3 ; convert to number
    ADD R0, R0, R4 ; add 1's

```

CS 270 - Fall Semester 2014

Conversion Routine (2 of 3)

```

    ADD R1, R1, #-1 ; one less digit
    BRz DoneAtoB ; done if zero
    ADD R2, R2, #-1 ; points to tens digit

    LDR R4, R2, #0 ; load digit
    ADD R4, R4, R3 ; convert to number
    LEA R5, Lookup10 ; multiply by 10
    ADD R5, R5, R4
    LDR R4, R5, #0
    ADD R0, R0, R4 ; adds 10's
    ADD R1, R1, #-1 ; one less digit
    BRz DoneAtoB ; done if zero
    ADD R2, R2, #-1 ; points to hundreds digit

```

CS 270 - Fall Semester 2014

Conversion Routine (3 of 3)

```

    LDR R4, R2, #0 ; load digit
    ADD R4, R4, R3 ; convert to number
    LEA R5, Lookup100 ; multiply by 100
    ADD R5, R5, R4
    LDR R4, R5, #0
    ADD R0, R0, R4 ; adds 100's

Done    RET

NegZero .FILL xFFD0 ; -0x30
ASCIIIBUF .BLKW 4
Lookup10 .FILL 0
          .FILL 10
...
Lookup100 .FILL 0
           .FILL 100
...

```

CS 270 - Fall Semester 2014

Binary to ASCII Conversion

- ◆ Converting a 2's complement binary value to a three-digit decimal number
 - Resulting characters can be output using OUT
- ◆ Instead of multiplying, we need to **divide by 100** to get hundreds digit.
 - Why wouldn't we use a lookup table for this problem?
 - Subtract 100 repeatedly from number to divide.
- ◆ First, check whether number is negative.
 - Write sign character (+ or -) to buffer and make positive.