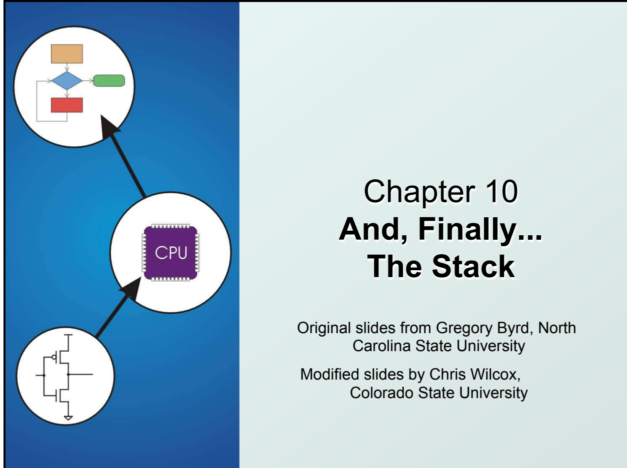


Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

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 two uses of the stack:
  - Evaluating arithmetic expressions
    - Store intermediate results on stack instead of in registers
  - Function calls
    - Store parameters, return values, return address, dynamic link
  - Interrupt-Driven I/O
    - Store processor state for currently executing program

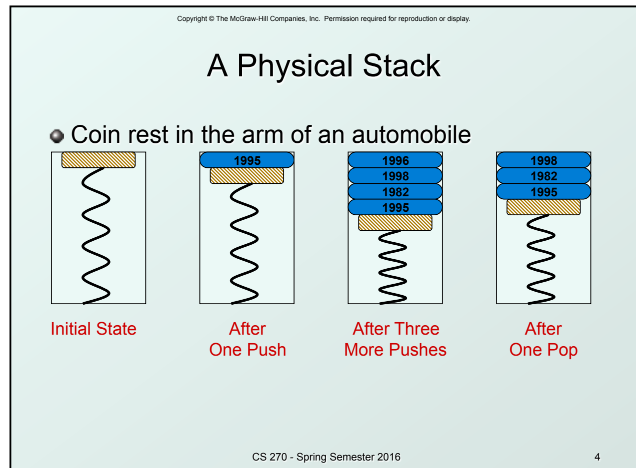
Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

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

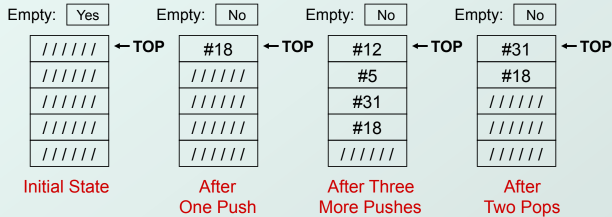
  POP: remove an item from the stack



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

A Hardware Implementation

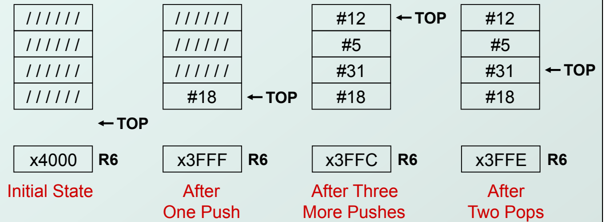
- Data items move between registers



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

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.

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

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
```


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

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
```

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

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
```


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)·(C+D) using a stack:

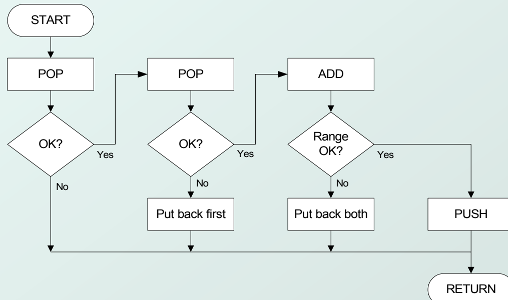
(1) push A
(2) push B
(3) ADD
(4) push C
(5) push D
(6) ADD
(7) MULTIPLY
(8) pop Result

Why use a stack?
- Limited registers.
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using FIFO data structure.

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

Example: OpAdd

- POP two values, ADD, then PUSH result.



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

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
```

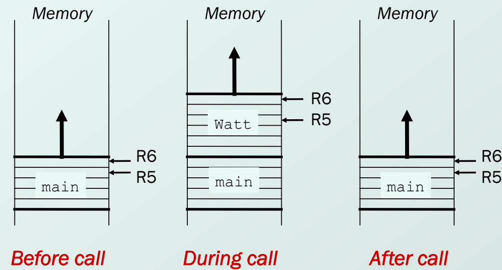
Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

Run-Time Stack

- Recall that local variables are stored on the run-time stack in an ***activation record***
- **Stack Pointer (R6)** is a pointer to the next free location in the stack, and is used to push and pop values on and off the stack.
- **Frame pointer (R5)** is a pointer to the beginning of a region of the activation record that stores local variables for the current function
- When a new function is **called**, its activation record is **pushed** on the stack; when it **returns**, its activation record is **popped** off of the stack.

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

Run-Time Stack



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

Example

```
double ValueInDollars(double amount, double rate);
int main()
{
    ...
    dollars = ValueInDollars(francs,
                             DOLLARS_PER_FRANC);
    printf("%f francs equals %f dollars.\n",
           francs, dollars);
    ...
}
double ValueInDollars(double amount, double rate)
{
  return amount * rate;
}
```

function declaration (prototype)

function call (invocation)

function definition (code)

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

Implementing Functions: Overview

- Activation record (stack frame)
  - information about each function, including arguments and local variables
  - stored on run-time stack

Calling function

- push new activation record
- copy values into arguments
- call function
- get result from stack

Called function

- execute code
- put result in activation record
- pop activation record from stack
- return

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

Activation Record

- `int NoName(int a, int b)`

```
{
  int w, x, y;
  .
  .
  .
  return y;
}
```

Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName

locals: y, x, w
R5 → w
bookkeeping: dynamic link, return address, return value
args: a, b

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

Activation Record Bookkeeping

- **Return value**
  - space for value returned by function
  - allocated even if function does not return a value
- **Return address**
  - save pointer to next instruction in calling function
  - convenient location to store R7 in case another function (JSR) is called
- **Dynamic link**
  - caller's frame pointer
  - used to pop this activation record from stack

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

Example Function Call

- `int Volta(int q, int r)`

```
{
  int k;
  int m;
  ...
  return k;
}

int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```


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

Calling the Function

- `w = Volta(w, 10);`

```
; push second arg
AND  R0, R0, #0
ADD  R0, R0, #10
PUSH R0
; push first argument
LDR  R0, R5, #0
PUSH R0
; call subroutine
JSR  Volta
```

new R6 → 25 q (param)
10 r (param)
old R6 → 25 w (local)
dyn link
ret addr
ret val
a (param)
x4000

Note: Caller needs to know number and type of arguments, doesn't know about local variables for function being called.

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

Starting the Callee Function

```
; leave space for return value
ADD  R6, R6, #-1
; push return address
PUSH R7
; push caller's frame ptr
PUSH R5
; set new frame pointer
ADD  R5, R6, #-1
; allocate space for locals
ADD  R6, R6, #-2
```

	Stack	
new R6 →		m
new R5 →		k
	x3FFB	dyn link
	x3100	ret addr
		ret val
R6 →	25	q
	10	r
R5 →	25	w
		dyn link
		ret addr
		ret val
		a
x4000		

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

Ending the Callee Function

```
return k;
; copy k into return value
LDR  R0, R5, #0
STR  R0, R5, #3
; pop local variables
ADD  R6, R5, #2
; pop dynamic link (into R5)
POP R5
; pop return addr (into R7)
POP R7
; return control to caller
RET
```

	Stack	
R6 →	-43	m
R5 →	217	k
	x3FFB	dyn link
	x3100	ret addr
new R6 →	217	ret val
	25	q
	10	r
new R5 →	25	w
		dyn link
		ret addr
		ret val
		a
x4000		

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

Resuming the Caller Function

```
w = Volta(w,10);
JSR  Volta
; load return value
; from top of stack
LDR  R0, R6, #0
; perform assignment
STR  R0, R5, #0
; pop return value
ADD  R6, R6, #1
; pop arguments
ADD  R6, R6, #2
```

	Stack	
R6 →	217	ret val
	25	q
new R6 →	10	r
R5 →	217	w
		dyn link
		ret addr
		ret val
		a
x4000		

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

Summary of LC-3 Function Call Implementation

1. Caller pushes arguments (last to first).
2. Caller invokes subroutine (JSR).
3. Callee allocates return value, pushes R7 and R5.
4. Callee allocates space for local variables.
5. Callee executes function code.
6. Callee stores result into return value slot.
7. Callee pops local vars, pops R5, pops R7.
8. Callee returns (JMP R7).
9. Caller loads return value and pops arguments.
10. Caller resumes computation

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

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

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…

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

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.

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

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.

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

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.

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

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.

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

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

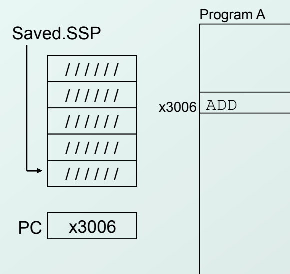
1. Pop PC from supervisor stack:
   (PC = M[R6]; R6 = R6 + 1)
2. Pop PSR from supervisor stack:
   (PSR = M[R6]; R6 = R6 + 1)
3. If going back to user mode, need to restore User Stack Pointer:
   (if PSR[15] = 1, R6 = Saved.USP)

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

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

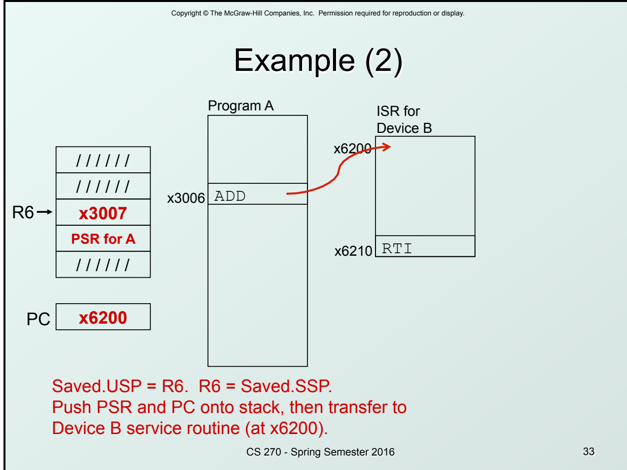
Example (1)

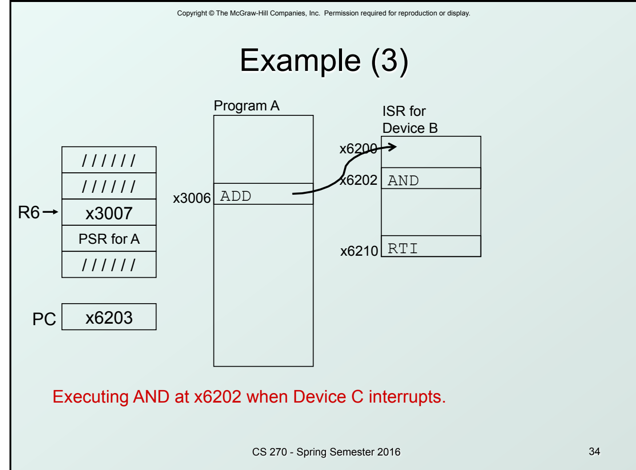
Saved.SSP

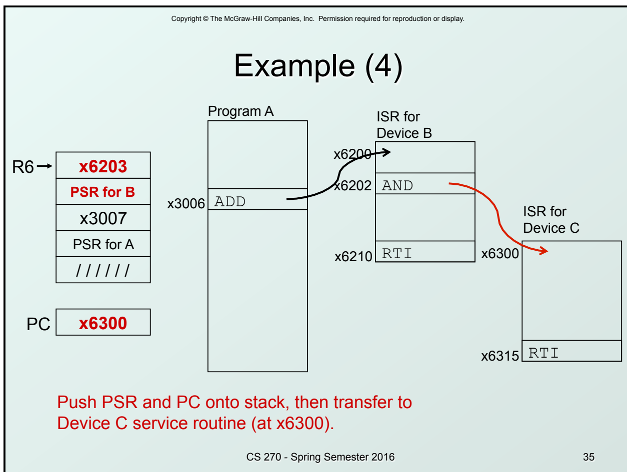


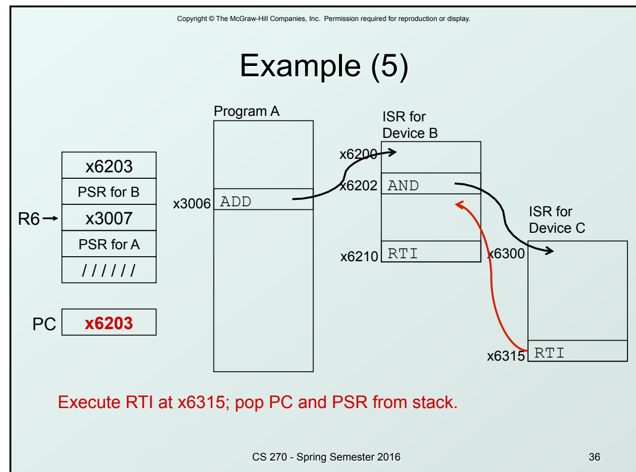
PC x3006

Executing ADD at location x3006 when Device B interrupts.


Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Example (2)
Program A
ISR for
Device B
//////
//////
R6
x3007
PSR for A
//////
x3006
ADD
x6200
x6210
RTI
PC
x6200
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to
Device B service routine (at x6200).
CS 270 - Spring Semester 2016
33

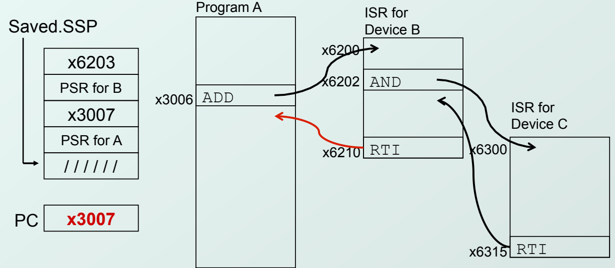

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Example (3)
Program A
ISR for
Device B
//////
//////
R6
x3007
PSR for A
//////
x3006
ADD
x6200
x6202
AND
x6210
RTI
PC
x6203
Executing AND at x6202 when Device C interrupts.
CS 270 - Spring Semester 2016
34


Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Example (4)
Program A
ISR for
Device B
R6
x6203
PSR for B
x3007
PSR for A
//////
x3006
ADD
x6200
x6202
AND
ISR for
Device C
x6210
RTI
x6300
x6315
RTI
PC
x6300
Push PSR and PC onto stack, then transfer to
Device C service routine (at x6300).
CS 270 - Spring Semester 2016
35


Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Example (5)
Program A
ISR for
Device B
x6203
PSR for B
R6
x3007
PSR for A
//////
x3006
ADD
x6200
x6202
AND
ISR for
Device C
x6210
RTI
x6300
x6315
RTI
PC
x6203
Execute RTI at x6315; pop PC and PSR from stack.
CS 270 - Spring Semester 2016
36

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

Example (6)



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