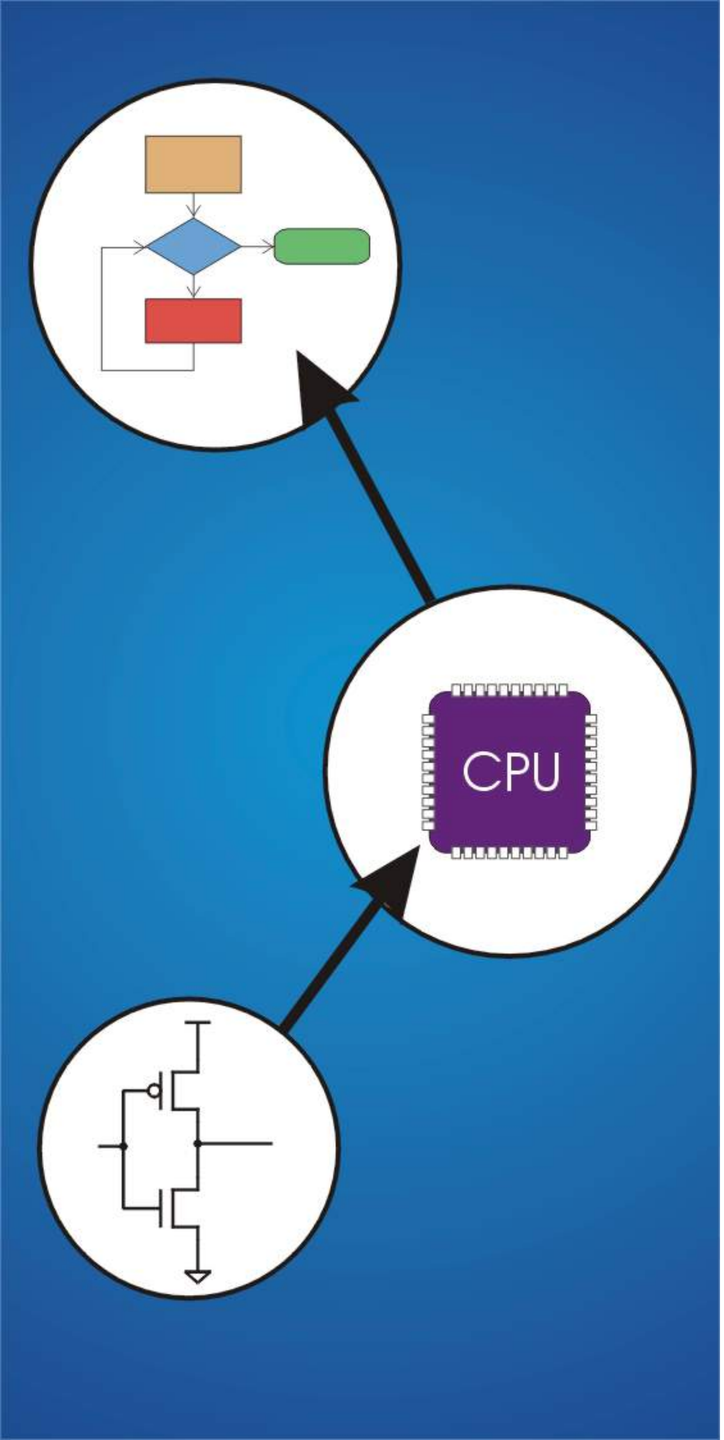
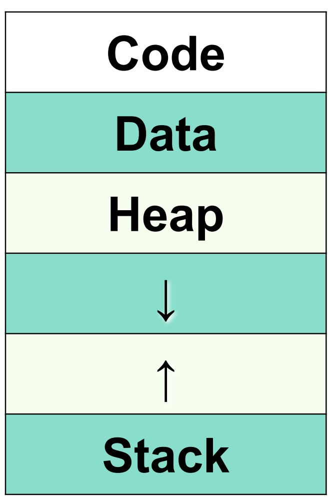


Chapter 10
And, Finally...
The Stack

Memory Usage

- Instructions are stored in code segment
- Global data is stored in data segment
- Local variables, including arryas, uses stack
- Dynamically allocated memory uses heap



- Code segment is write protected
- Initialized and uninitialized globals
- Stack size is usually limited
- Stack generally grows from higher to lower addresses.

Execution Stack

- What is a stack?
  - First In, Last Out (FILO) data structure
  - PUSH adds data, POP removes data
    - Overflow condition: push when stack full
    - Underflow condition: pop when stack empty
  - Stack grows and shrinks as data is added and removed
  - Stack grows downward from the end of memory space
  - Function calls allocate a stack frame
    - Return cleans up by freeing the stack frame
    - Corresponds nicely to nested function calls
    - Stack Trace shows current execution (Java/Eclipse)

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.

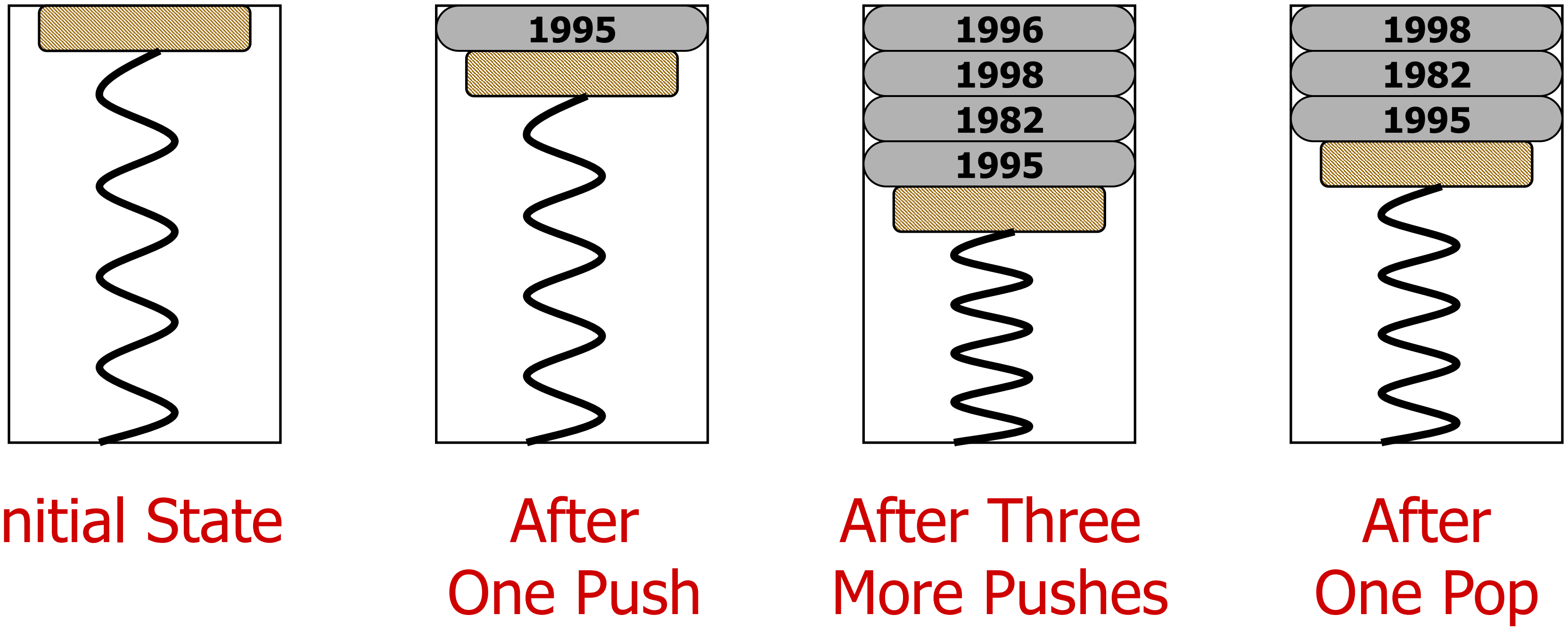
Two main operations:

PUSH: add an item to the stack

POP: remove an item from the stack

A Physical Stack

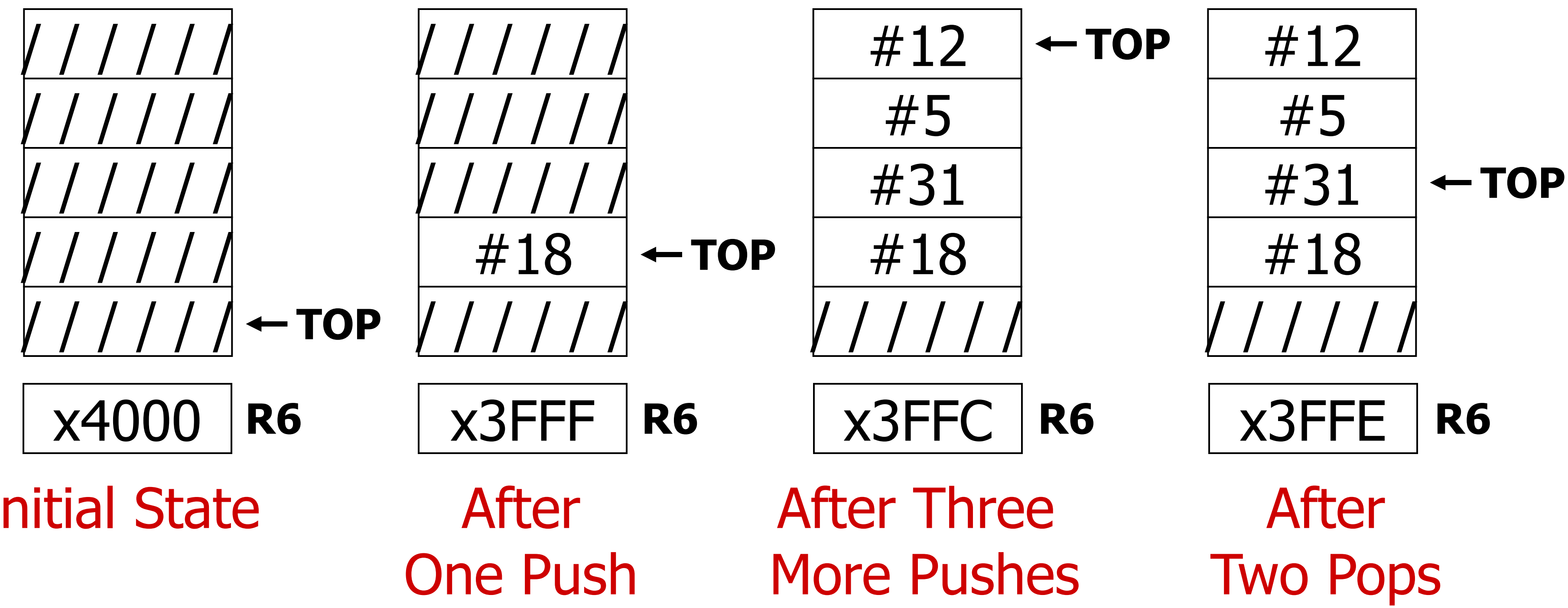
Coin rest in the arm of an automobile



First quarter out is the last quarter in.

A Software Implementation

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



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
```

Rest of the slides skipped for now

Skip to discussion on Activation Records.

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]**

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

- **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!

Use a stack!

- Location of stack "hard-wired".
- Push state to save, pop to restore.

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.

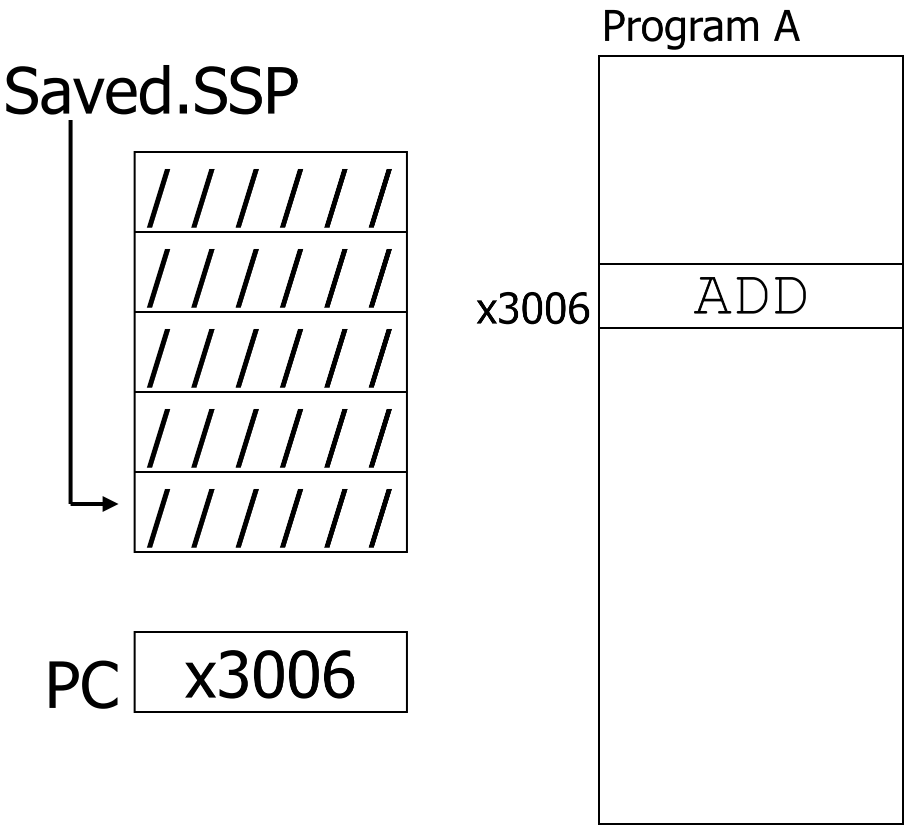
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTI	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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 PSR[15] = 1, R6 = Saved.USP.**
   (If going back to user mode, need to restore User Stack Pointer.)

RTI is a privileged instruction.

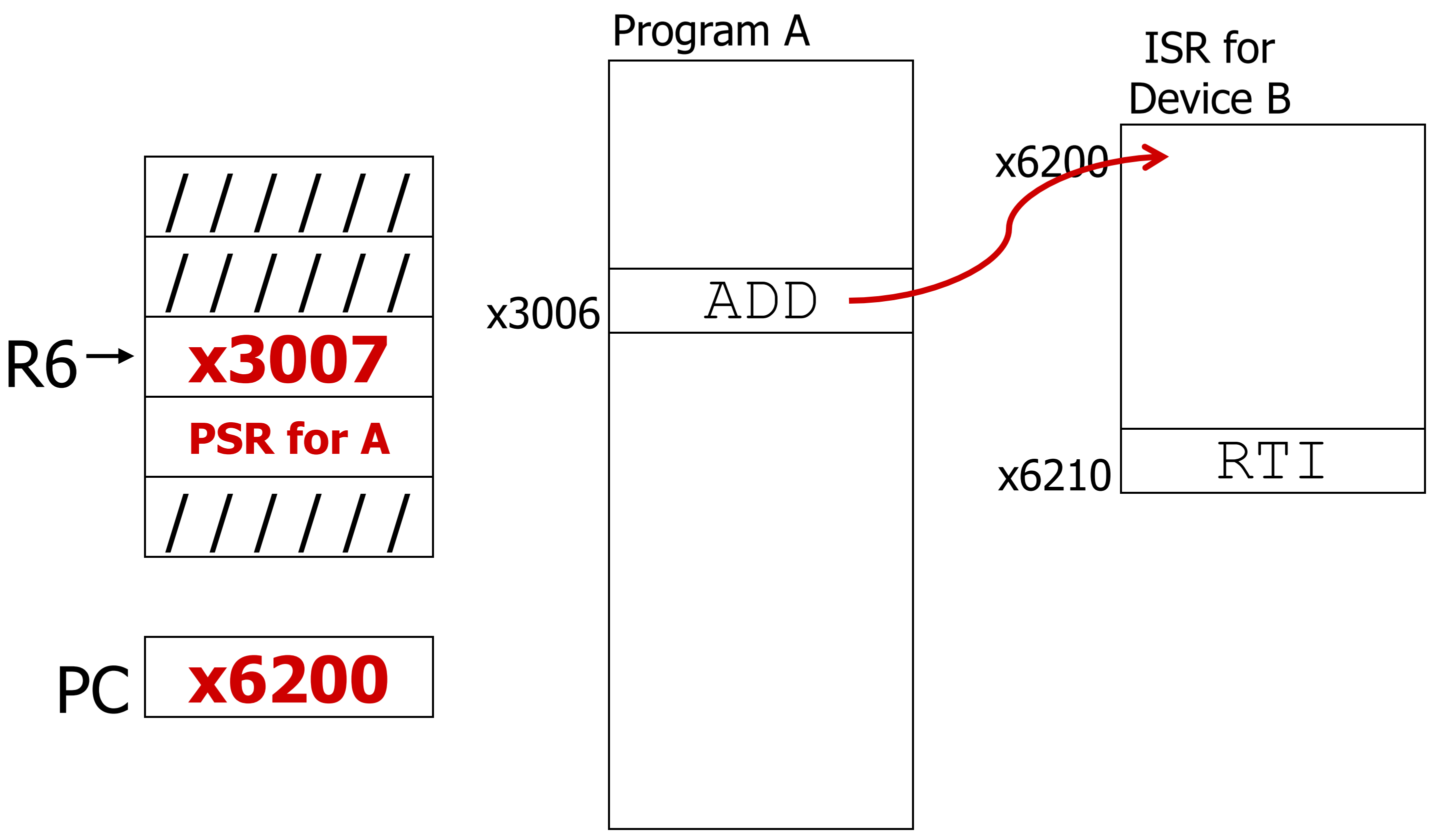
- **Can only be executed in Supervisor Mode.**
- **If executed in User Mode, causes an exception.**
  (More about that later.)

Example (1)



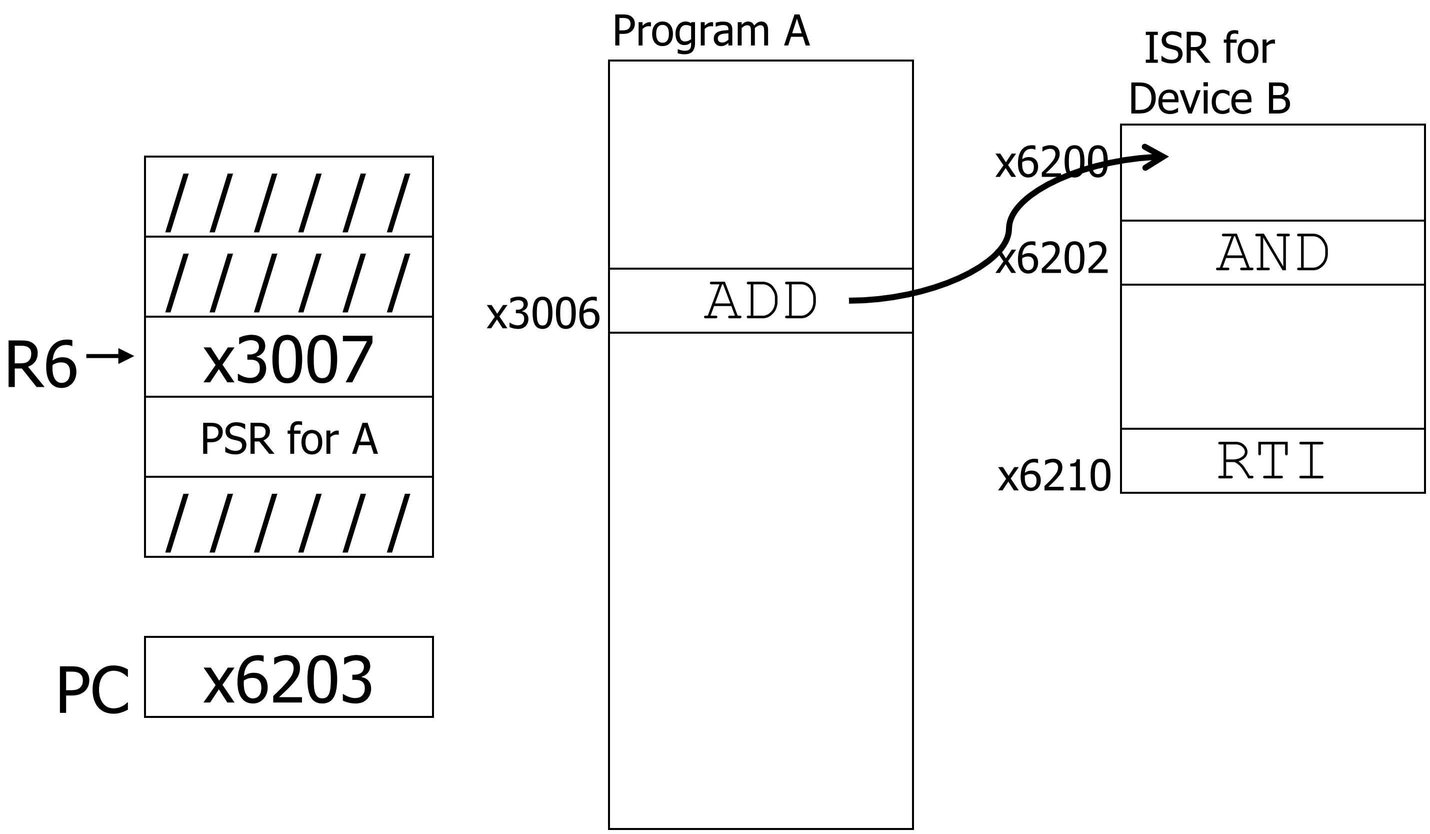
Executing ADD at location x3006 when Device B interrupts.

Example (2)



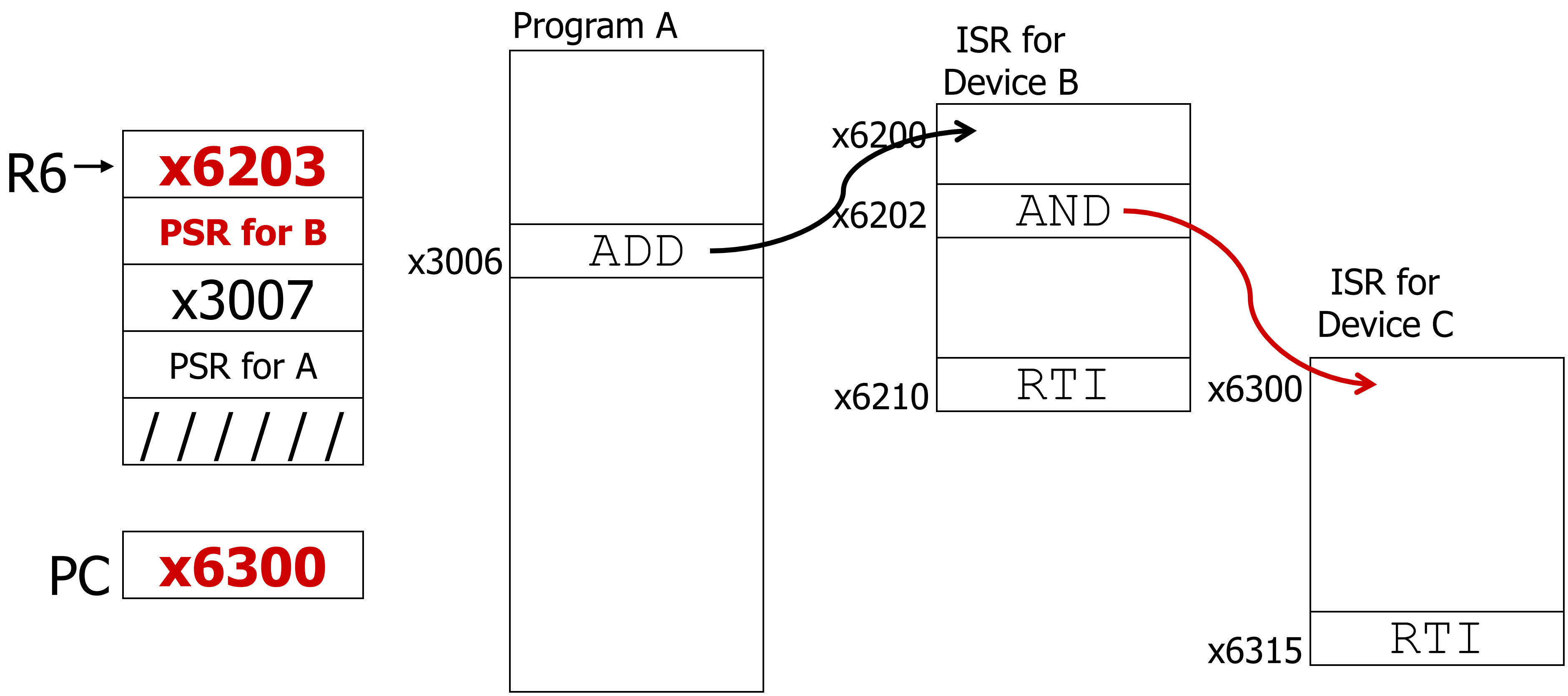
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to Device B service routine (at x6200).

Example (3)



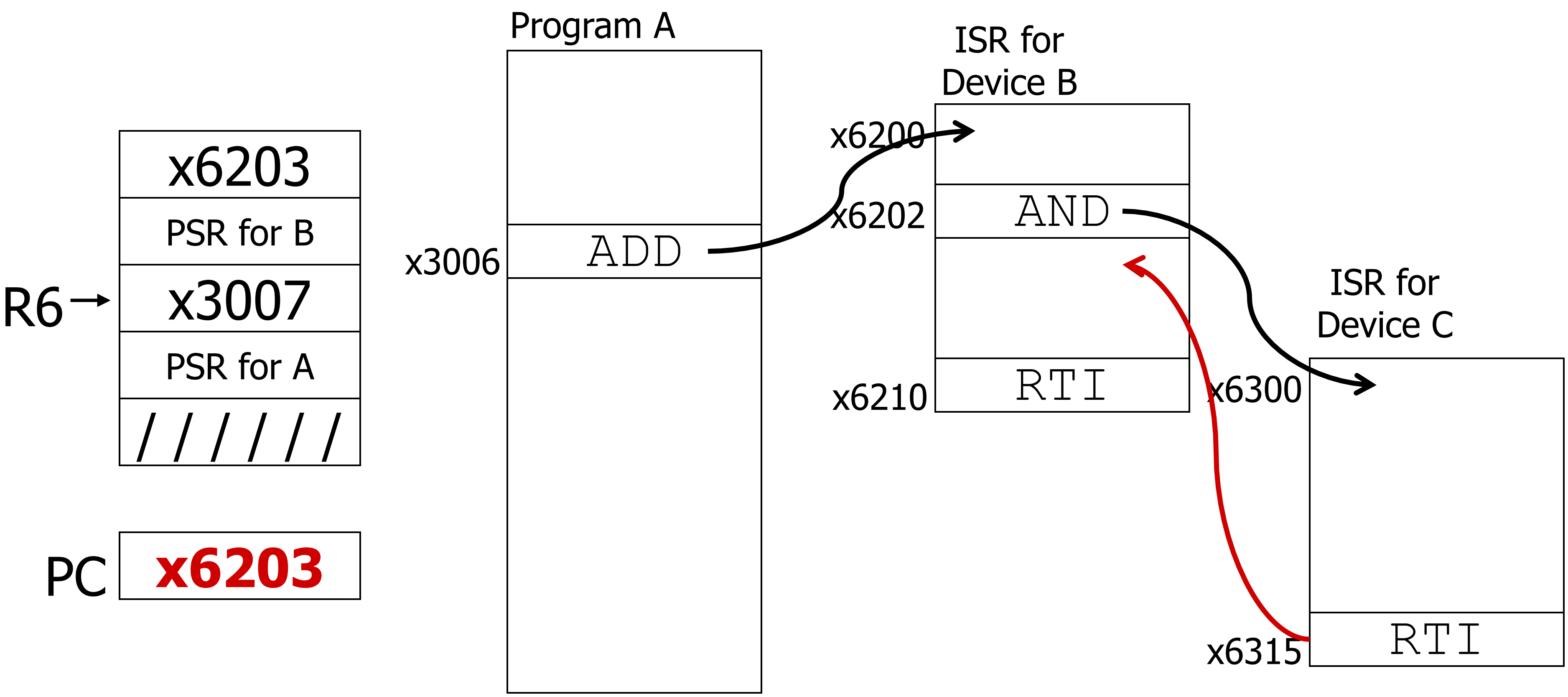
Executing AND at x6202 when Device C interrupts.

Example (4)



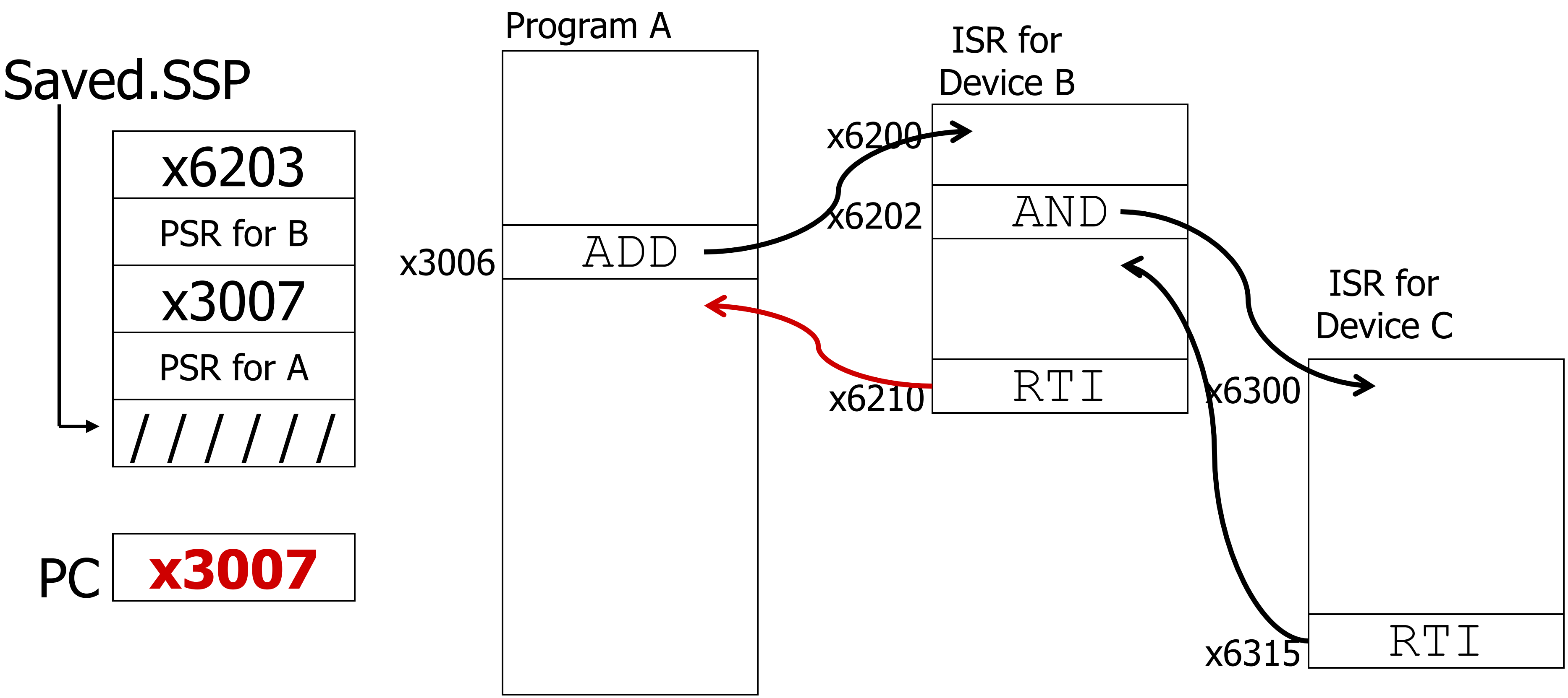
Push PSR and PC onto stack, then transfer to Device C service routine (at x6300).

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.

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**

Data Type Conversion

These routines in the following slides might be useful.

Data Type Conversion

Keyboard input routines read ASCII characters, not binary values.

Similarly, 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)

ASCII to Binary

Useful to deal with mult-digit decimal numbers

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

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

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.

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.

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
```

Complete Conversion Routine (1 of 3)

```
; Three-digit buffer at ASCIIBUF.
; 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, ASCIIBUF
               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 ones contrib
```

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

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 contrib
;
DoneAtoB        RET
NegZero         .FILL xFFD0   ; -x30
ASCIIBUF        .BLKW 4
Lookup10        .FILL 0
                .FILL 10
                .FILL 20
...
Lookup100       .FILL 0
                .FILL 100
...
```

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.**

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
               BRnzp  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
                BRnzp  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
Loop100         ADD R0, R0, R3
                BRn End10
                ADD R2, R2, #1   ; add one to digit
                BRnzp  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
```