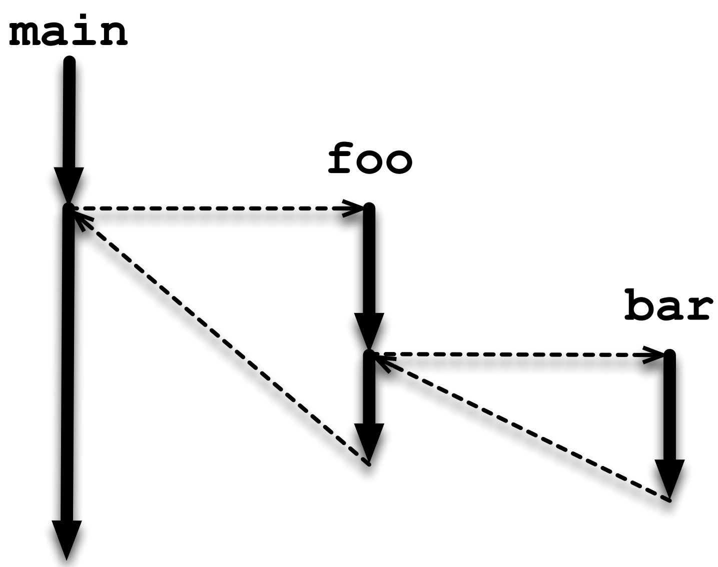
Subroutines

010.133
Digital Computer Concept and Practice
Spring 2013

Lecture 08

Subroutines

- A sequence of instructions to perform a particular task
- Called from multiple places, even from within itself (in which case it is called recursive)
- Need to take care of returning control to the location just after the calling location, usually with the support of call and return instructions at the machine language level



Subroutines (contd.)

- A subroutine may expects to obtain one or more data values from the caller
  - Called parameters or arguments
- A subroutine may return a value (called return value) to its caller
  - May use parameters to return data values to the caller
- For example, calling a subroutine **pow** with arguments 2 and 10 returns 2^{10}

Branch and Link Register Instruction

- Use it to implement a subroutine call
  - Before branching occurs, the return address is stored in LR
- Returning to the caller
  - Set up PC with the return address
  - MOV PC, LR
- For example, when the call to the subroutine foo occurs, LR is set to 0x8204
  - When returning to the caller from foo, PC is set to the value saved in LR

```
                ...
0x8200          BL   foo
0x8204          MOV  R3, R0
                ...
0x8400  foo:    SUB  R3, R3, #1
0x8404          MOV  PC, LR
                ...
```

Calling Convention

- Rules for a caller to pass parameters to a callee and receive a result value back from the callee
  - Where to place parameters and return values
    - In registers
    - On the call stack
    - A mix of both
  - The order in which parameters are passed
  - What registers might be ***overwritten*** by the callee
- Typically, responsibility for setting up and cleaning up a subroutine call is distributed between the caller and callee
- Different calling convention for different languages, different operating systems, and different architectures

Calling Convention for Our VM

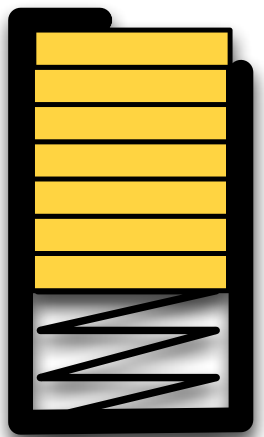
- R0-R3
  - Argument, scratch
  - R0 contains the result of callee when it returns to the caller
  - The caller saves their contents if it needs them again
- R4-R12
  - Variable register
  - The callee saves their contents and restore them before returning to the caller

Calling Convention for Our VM (contd.)

- R13
  - Stack pointer (SP)
  - The value of SP at the entry of the callee must be equal to the value at the exit of the callee
- R14
  - Link register (LR)
  - If the return address is saved in the memory, this register can be used freely
- R15
  - Program counter (pc)
  - Cannot be used for other purposes

Coin Dispenser

- Only the top coin is visible and accessible
  - As new coins are added, each new coin becomes the top of the stack, hiding each coin below, pushing the stack of coins down
  - As the top coin is removed from the stack, the second coin becomes the top of the
- Last-In, First-Out (LIFO)

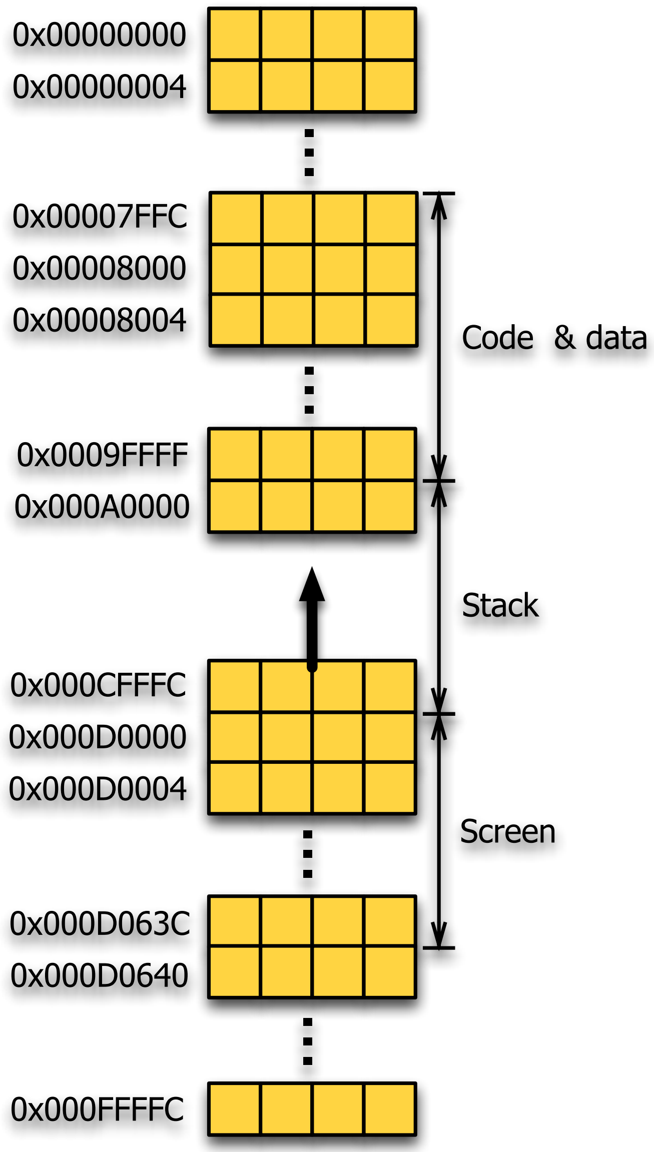


Stacks

- A stack is a data structure based on the principle of LIFO
- Data structure is a way of storing data in a computer
  - Any method of organizing a collection of data to manipulate it effectively and efficiently
- Two basic operations of a stack
  - Push
    - Adds a given element to the top of the stack leaving previous elements below
  - Pop
    - Removes the current top element of the stack.
- Most CPUs include support for stacks in their ISA
  - Stack pointer

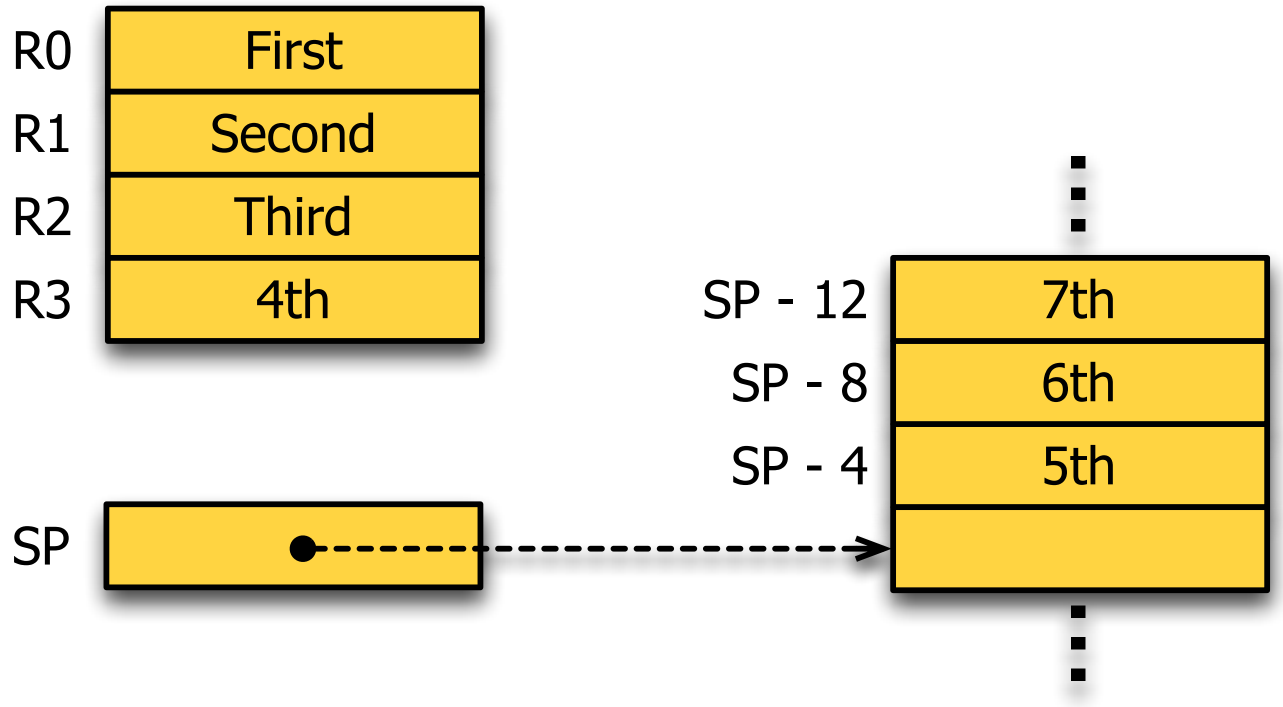
The Stack in Our VM

- Full descending stack
  - A full stack is where the stack pointer points to the last data item pushed on the stack
  - A descending stack grows downwards in memory (i.e., grows to the direction of decreasing addresses
- SP (stack pointer) contains the address at the top of the run-time stack



Parameter Passing for Our VM

- Parameter passing
  - Through r0, r1, r3, r4, and the stack (excessive parameters)



Stack Frame

- The block of information stored on the stack to implement a subroutine call and return
  - Parameters to the subroutine
  - Saved registers
  - Local variables (in C)
  - Return address
- Allocated on the stack when a subroutine is called
  - The programmer or a compiler must allocate it
- Removed upon return from the subroutine
- In general, the stack frame for a subroutine contains all necessary information to save and restore the state of a subroutine

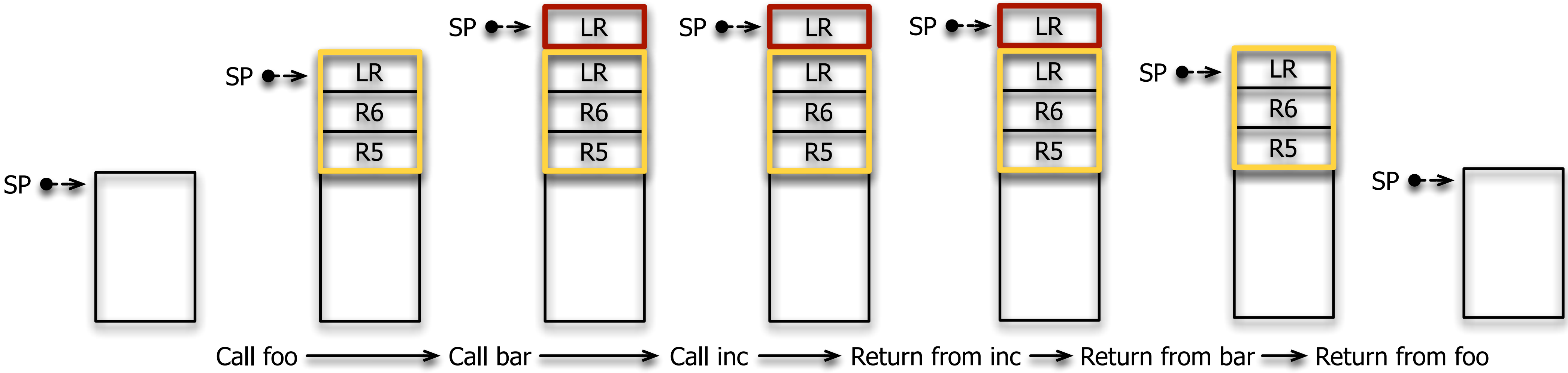
Stack Frames (contd.)

- A subroutine foo gets two arguments in R0 and R1 from the caller and returns the result in R0
  - $foo(a, b) = bar(a^b)$
  - foo calls bar
- A subroutine bar gets one argument in R0 from the caller and returns the result in R0
  - $bar(a) = inc(a \times a)$
  - bar calls inc
- A subroutine inc gets one argument in R0 from the caller and returns the result in R0
  - $inc(a) = a + 1$

```
@====================
@ inc(x)
@====================
inc:  ADD  R0, R0, #1  @ Increment R0, R0 contains the
                       @    return value
      MOV  PC, LR      @ Return to the caller
```

```
@====================
@ bar(x)
@====================
bar:  SUB  SP, SP, #4  @ Decrement SP by 4
      STR  LR, [SP]    @ Save LR on the stack
@====================
      MUL  R1, R0, R0
      MOV  R0, R1      @ The argument for inc is in R0
      BL   inc         @ Call inc. When returning from
                       @    inc, R0 contains the result
@====================
      LDR  LR, [SP]    @ Restore LR from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      MOV  PC, LR      @ Return to the caller
```

```
@====================
@ foo(x, y)
@====================
foo:  SUB  SP, SP, #4  @ Decrement SP by 4
      STR  R5, [SP]    @ Save R5 on the stack
      SUB  SP, SP, #4  @ Decrement SP by 4
      STR  R6, [SP]    @ Save R6 on the stack
      SUB  SP, SP, #4  @ Decrement SP by 4
      STR  LR, [SP]    @ Save LR on the stack
@====================
      MOV  R5, #1
L1:   CMP  R1, #0
      BEQ  DONE
      MUL  R6, R5, R0
      MOV  R5, R6
      SUB  R1, R1, #1
      B    L1
DONE: MOV  R0, R5      @ The argument for bar is in R0
      BL   bar         @ Call bar. When returning from
                       @   bar, R0 contains the result
@====================
      LDR  LR, [SP]    @ Restore LR from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      LDR  R6, [SP]    @ Restore R6 from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      LDR  R5, [SP]    @ Restore R5from the stack
      ADD  SP, SP, #4  @ Increment SP by 4
      MOV  PC, LR      @ Return to the caller
```


SP
SP
LR
R6
R5
SP
LR
LR
R6
R5
SP
LR
LR
R6
R5
SP
LR
LR
R6
R5
SP
LR
R6
R5
SP
Call foo
Call bar
Call inc
Return from inc
Return from bar
Return from foo

Converting an ASCII String to an Integer

- Assume we have two subroutines **`printCh`** and **`printInt`** for printing a character and an integer on the screen, respectively
- **`printCh`**
  - Prints a character on the screen
  - Before you call printCh, you need to set up the arguments in r0 and r1
    - r0 - the address A in the screen section of the memory to which the character is printed
    - r1 - the character in ASCII
  - On return, r0 contains (A + 1)
    - The next address on the screen for printing

Converting an ASCII String to an Integer (contd.)

- `printInt`
  - Before you call printInt, you need to set up r0 and r1
    - r0 - the address A in the screen section of the memory to which the integer is printed
    - r1 - the integer
  - On return, r0 contains (A + the number of characters printed)
    - The next address on the screen for printing

Converting an ASCII String to an Integer (contd.)

- Subroutine **`asciiStr2Int`**
  - Converts a null-terminated string to an integer
    - "3456" → 3456
  - When called, r0 contains the address of the null-terminated string
  - On return r0 contains the converted integer

```
@====================
@ asciiStr2Int
@====================
asciiStr2Int:
     ldr  r12, r0          @ R12 contains the address
                           @   of the string
     eor  r0, r0, r0       @ Set r0 to 0
     mov  r10, #10         @ Set r10 to 10
     ldrb r11, [r12]       @ Load the 1st digit to r11
LOOP:
     cmp  r11, #0          @ Does r11 contains '\0' (NULL)?
     beq  DONE             @ If so, done
     mul  r1, r0, r10      @
     sub  r11, r11, #0x30  @ Get the value of the digit,
                           @   '0' is 0x30 in ASCII
     add  r0, r1, r11      @
     add  r12, r12, #1     @
     ldrb r11, [r12]       @ Get the next digit
     b    LOOP             @
DONE:
     mov  pc, lr           @
```

Converting an ASCII String to an Integer (contd.)

- After returning from asciiStr2Int, r0 contains the integer

```
@====================
@ main
@====================
main:
      ldr  r0, addr_str    @ R0 contains the address
                           @   of the string
      bl   asciiStr2Int    @ Call asciiStr2Int
      mov  r1, r0          @ R1 contains the integer
      ldr  r0, addr_screen @ R0 contains the location
                           @   on the screen
      bl   printInt        @ Call printInt
      b    halt            @
addr_str
      .word 0x9000
addr_screen
      .word 0xD0000
```

C Functions

- C functions are subroutines
- A function definition consists of the type of its return value, its name, a list of arguments expected when it is called, and the block of code it executes
- **`main`** function
  - A special C function where the program execution begins

```c
#include <stdio.h>

int area(int w, int h) {
    return w * h;
}

int main(void) {
    int width = 5, height;

    height = 10;
    printf("width: %d, height: %d, area: %d\n",
           width, height, area(width, height));

    return 0;
}
```

Polish Notation

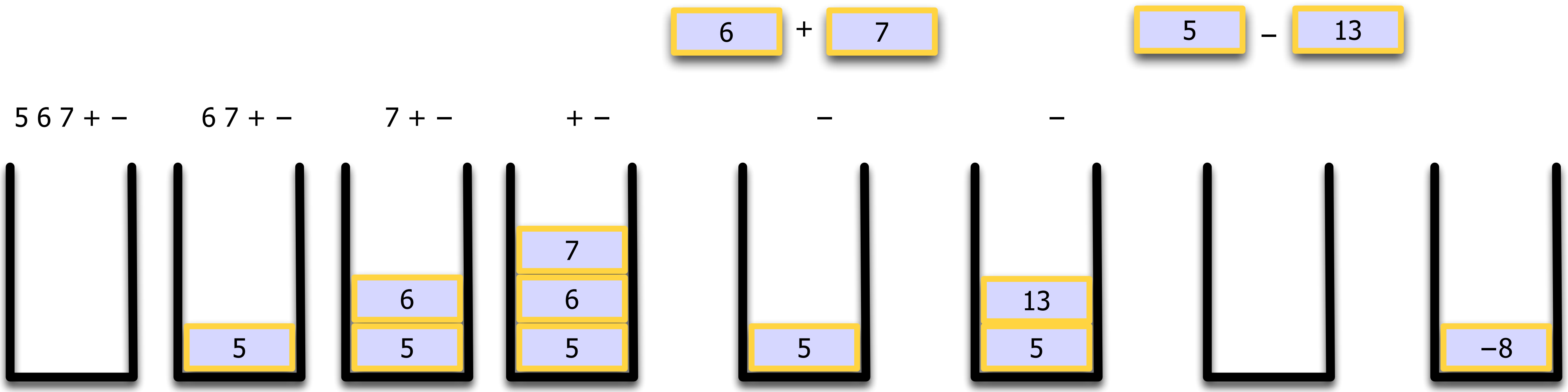
- Prefix notation
  - Places operators to the left of their operands
    - The operator precedes all its operands
  - No parentheses needed
    - No ambiguity
- 5 – (6 + 7) vs. 5 – 6 + 7
  - Infix notation
- – 5 + 6 7 vs. + – 5 6 7
  - Prefix notation

Reverse Polish Notation (RPN)

- Postfix notation
  - The operators follow their operands
    - The operator is preceded by all its operands
  - For binary operators
  - No parentheses needed
    - No ambiguity
- 5 – (6 + 7) vs. 5 – 6 + 7
  - Infix notation
- 5 6 7 + – vs. 5 6 – 7 +
  - postfix notation

Evaluation of Expressions in RPN using a Stack

- Operands are pushed onto a stack
- When you see an operator, its operands are popped from the stack, and the operation is performed with the operands
- The result pushed back on the stack
- 5 - (6 + 7) in RPN
  - 5 6 7 + −



A Simple Calculator

- The null-terminated input string that contains an arithmetic expression in the RPN is stored in a memory location whose address is stored in the location labeled "input_str_addr"
- The numbers and operators are delimited by a space character (0x20 in ASCII)
  - A buffer is used to store a string that represents a number
  - This will be converted to an integer
- We want to know the value of the input expression
  - Use a user defined stack (not a call stack)
    - An empty descending stack
    - The stack bottom is located at "stack_bottom_addr"

A Simple Calculator (contd.)

```
@====================
@ Simple Calculator
@====================
    ldr  r5, stack_bottom_addr   @ R5 is a pointer to the top of the user stack
    ldr  r7, input_str_addr      @ R7 contains the address of the input string
    sub  r7, r7, #1
LOOP1:
    add  r7, r7, #1              @ Increment r7 by one
    ldrb r6, [r7]                @ Load the next char in r6
LOOP1_NUMBER:
    cmp  r6, #0                  @ Check if the char is '\0' (NULL)?
    beq  ALL_DONE                @ If so, done!
    cmp  r6, #0x2b               @ Check if the char is '+'
    beq  ADD                     @ If so, perform addition
    cmp  r6, #0x2d               @ Check if the char is '-'
    beq  SUBTRACT                @ If so, perform subtraction
    cmp  r6, #0x20               @ Check if the char is a space (' ')
    beq  LOOP1                   @ If so, skip to the next char
```

A Simple Calculator (contd.)

```
NUMBER:                              @ Otherwise, it is the first char of a number
    ldr   r8, =buffer                @ R8 contains the address of the buffer
LOOP2:
    strb r6, [r8]                    @ Put the char in to the buffer
    add  r8, r8, #1                  @ Increment the buffer pointer
    add  r7, r7, #1                  @ The address of the next char
    ldrb r6, [r7]                    @ Load the next char in r6
    cmp  r6, #0x20                   @ Check if the char is a space (' ')
    beq  CONVERT_STR_TO_INT:         @ If so, a complete string for a
                                     @   number is recognized
    cmp  r6, #0                      @ Check if the char is '\0'
    bne  LOOP2                       @ If not, we are in the middle of a number
CONVERT_STR_TO_INT:                  @ Convert the string to an integer
    mov  r9, #0
    strb r9, [r8]                    @ Put '\0' at the end of the buffer
    ldr  r0, =buffer                 @ The first argument of asciiStr2Int
    bl   asciiStr2Int                @ Call asciiStr2Int
    sub  r5, r5, #4                  @ Decrement the user stack pointer by four
    str  r0, [r5]                    @ Push the integer onto the user stack
    b    LOOP1_NUMBER
```

A Simple Calculator (contd.)

```
ADD:
    ldr  r4, [r5]              @ Pop the second operand of +
    add  r5, r5, #4            @ Increment the user stack pointer by four
    ldr  r3, [r5]              @ Pop the first operand of +
    add  r3, r3, r4            @ Perform addition
    str  r3, [r5]              @ Push the result on to the user stack
    b    LOOP1
SUBTRACT:
    ldr  r4, [r5]              @ Pop the second operand of -
    add  r5, r5, #4            @ Increment the user stack pointer by four
    ldr  r3, [r5]              @ Pop the first operand of -
    sub  r3, r3, r4            @ Perform subtraction
    str  r3, [r5]              @ Push the result on to the user stack
    b    LOOP1
ALL_DONE:
    ldr  r0, screen_addr       @ The first argument of printInt
    ldr  r1, [r5]              @ The second argument of printInt
    bl   printInt              @ Call printInt
    b    halt
```

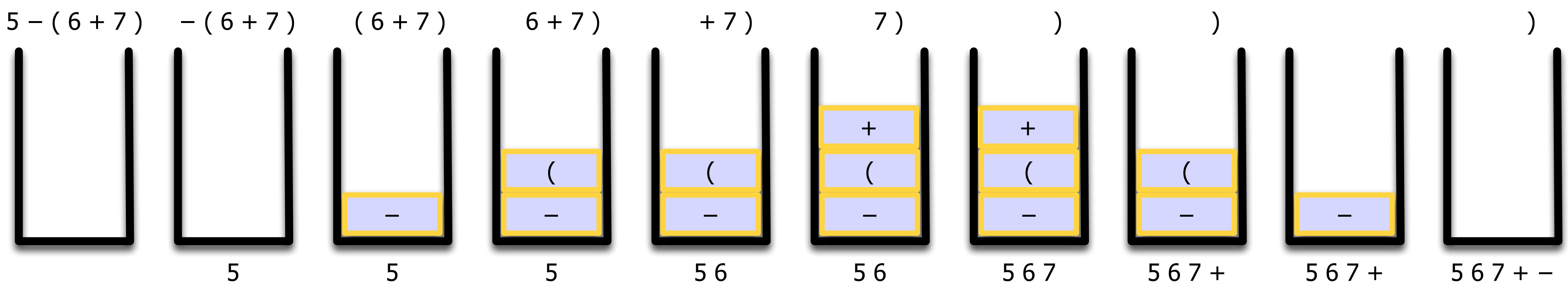
A Simple Calculator (contd.)

```
screen_addr:
    .word 0xD0000
input_str_addr:
    .word 0x9000
stack_bottom_addr:
    .word 0xa000
buffer:
    .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

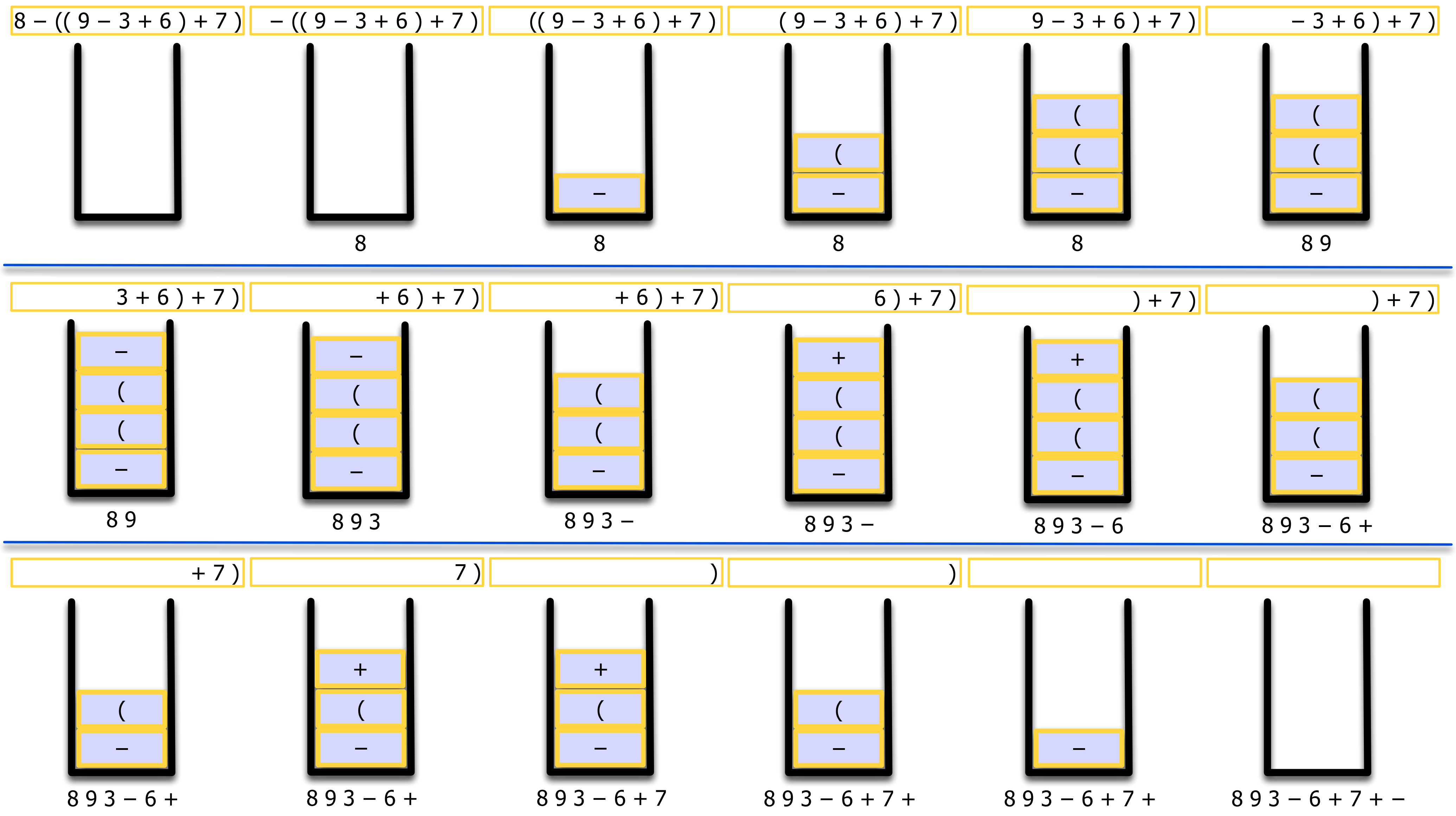
Infix to Postfix Conversion

- A token is a categorized block of text
  - Number, +, –, (,)
- Read a token from the input
  - If the token is a number, then copy it to the output
  - If the token is a left parenthesis, then push it onto the stack
  - If the token is a right parenthesis, pop the element onto the output until the topmost element of the stack is a left parenthesis
    - Discard both parentheses
  - If the token is an operator,
    - If the top element of the stack is an operator, pop it off the stack and copy it the output
    - Push the token onto the stack
- When there is no more tokens to read,
  - While there are still operators in the stack, pop them on to the output from the top

Infix to Postfix Conversion (contd.)



Infix to Postfix Conversion (contd.)



Infix to Postfix Conversion (contd.)

- Can you write an assembly routine that performs the conversion?
  - String to string conversion
  - For example, a string "5 - (6 + 7)" is converted to "5 6 7 + -"

Iteration vs. Recursion

- Factorial function
  - An recursive definition

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

```
@===========================
@ Fact (iterative version)
@===========================
fact:
    mov   r1, #1
LOOP:
    cmp   r0, #0         @ Check if r0 is 0
    beq   DONE           @ If so, done
    mul   r1, r1, r0
    sub   r0, r0, #1     @ Decrement r0 by one
    b     LOOP
DONE:
    mov   r0, r1         @ The return value is in r0
    mov   pc, lr         @ Return to the caller
```

```
//============================
// Fact (iterative version)
//============================
int fact(int n)
{
    int m = 1, i;
    for (i = 1; i <= n; i++)
        m = m * i;

    return m;
}
```

Iteration vs. Recursion (contd.)

```
@==========================
@ Fact (recursive version)
@==========================
fact:
    sub  sp, sp, #4
    str  r4, [sp]
    sub  sp, sp, #4
    str  lr, [sp]
@==========================
    cmp  r0, #0        @ Check if r0 (n) is 0
    beq  BASE          @ If so, done
    mov  r4, r0        @ R4 contains n
    sub  r0, r0, #1    @ R0 contains n-1
    bl   fact          @ Call fact(n-1)
    mul  r0, r4, r0    @ n × fact(n-1)
    b    DONE
BASE:
    mov  r0, #1        @ The return value 1 is in r0
@==========================
DONE:
    ldr  lr, [sp]      @ Restore lr from the stack
    add  sp, sp, #4
    ldr  r4, [sp]      @ Restore lr from the stack
    add  sp, sp, #4
    mov  pc, lr        @ Return to the caller
```

```
//==========================
// Fact (recursive version)
//==========================
int fact(int n)
{
    if (n == 0)
        return 1;
    else
        return n*fact(n-1);
}
```

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fact(n-1) & \text{if } n > 0 \end{cases}$$

A subroutine (function) is recursive if it calls itself either directly or indirectly

Fibonacci Numbers

- Fibonacci numbers are defined by

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- Can you write an assembly subroutine and a C function that takes an integer n as an argument and returns the n^{th} Fibonacci number?
  - Both iterative version and recursive version