PA4
The Hardware Stack

CSE 30: Computer Organization and Systems Programming

Diba Mirza
Dept. of Computer Science and Engineering
University of California, San Diego
Implement: is_substring

/* Check if s1 is a substring of s2 */
/* Return 1 if s1 is a substring of s2 */
/* Return 0 otherwise */

int is_substring (char *s1, char *s2) {
    int i = 0, j = 0;
    if (s1 == NULL || s2 == NULL)
        return 0;
    if (s1[0] == '\0')
        return 1;
    while (s2[i] != 0) {
        if (s2[i] == s1[0]) {
            j = 0;
            while (s1[j] != '\0' && s2[i + j] != '\0' && s1[j] == s2[i + j])
                j++;
            if (s1[j] == '\0')
                return 1;
        }
        i++;
    }
    return 0;
}
Stacks

- A stack follows the last in first out (LIFO) principle
  - Operations: push, pop
- Software stacks: Program with a LIFO interface. Implementations are based on:
  - Arrays
  - Linked-lists
- Hardware stacks: Area of memory that grows and shrinks according to the LIFO principle
  - Pointers define stack limits:
    - Base pointer: points to the “bottom” of the stack
    - Stack pointer: points to the “top” of the stack
  - Fixed width
Hardware Stack Models

- There are a few different models for implementing hardware stacks, different architectures follow different models.

- ARM follows a Full Descending Model.

- Why do I need to know these models?
  - Creating local variables
  - Understanding weird program behavior
  - Implement your own!
Hardware Stack Models

In general the register ‘sp’ holds the address of the ‘top’ of the stack.

1. Ascending/Descending

Ascending: Stack grows in the direction of increasing memory locations.
Descending: Stack grows in the direction of decreasing memory locations.

2. Full/Empty

Full: Stack pointer points to the last entry on the stack.
Empty: Stack pointer points to the next available (free) spot on the stack.
Match the diagram to stack model

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FA</td>
<td>EA</td>
<td>FD</td>
<td>ED</td>
</tr>
<tr>
<td>B</td>
<td>FD</td>
<td>ED</td>
<td>FA</td>
<td>EA</td>
</tr>
<tr>
<td>C</td>
<td>ED</td>
<td>FD</td>
<td>EA</td>
<td>FA</td>
</tr>
<tr>
<td>D</td>
<td>EA</td>
<td>FA</td>
<td>ED</td>
<td>FD</td>
</tr>
</tbody>
</table>

- (1) FA: Full Ascending
- (2) FD: Full Descending
- (3) EA: Empty Ascending
- (4) ED: Empty Descending

0x418
Push r5, r3, r1

0x400
Old SP

0x3FC
SP

0x3E8
SP

4 bytes
ARM Procedure call standard

The AAPCS specifies a
- Full descending stack
- Stack width is 8 bytes

```
push {r0, r1, r3-r5}
```

```
LDR r6, [sp, #12]
! Read the value of r4 in stack!
```

```
0x400
```

```
0x418
```

```
0x3e8
```

```
sp should always be 8byte aligned
```

```
0x418
```

```
0x400
```

```
Old SP
```

```
r5
```

```
r4
```

```
r3
```

```
r2
```

```
r1
```

```
r0
```

```
sp
```

```
0x400
```

```
or
```

```
rs
```

```
empty
```

```
0x3e8
```

```
sp
```

```
new
```

```
To read this value use LDR
```

```
int i = 10;

Which of the following ARM statements stores the local variable i on the stack according to the ARM procedure call standard?

A. MOV r0, #10
   SUB sp, sp, #8
   LDR r0, [sp]  

B. MOV r0, #10
   SUB sp, sp, #8
   STR r0, [sp]  

C. MOV r0, #10
   SUB sp, sp, #4
   LDR r0, [sp]  

D. MOV r0, #10
   SUB sp, sp, #4
   STR r0, [sp]
int i = 10, j = 20;

Which of the following ARM statements stores the local variables i and j on the stack according to the ARM procedure call standard?

A. MOV r0, #10
   MOV r1, #20
   SUB sp, sp, #8
   STR r0, [sp]
   STR r1, [sp, #4]

B. MOV r0, #10
   MOV r1, #20
   SUB sp, sp, #8
   STR r0, [sp]
   STR r1, [sp]

C. Both A and B follow the standard but A is more optimized
STMDB sp!, {r0 – r1}

MOV r0, #10
MOV r1, #20
SUB sp, sp, #8
STR r0, [sp]
STR r1, [sp, #4]
New instruction STM

STMDB sp!, {r0 – r1}

Are the two given ARM codes equivalent?
A. Yes
B. No

foo:
MOV r0, #10
MOV r1, #20
SUB sp, sp, #8
STR r0, [sp]
STR r1, [sp, #4]
BX lr

foo:
MOV r0, #10
MOV r1, #20
STMDB sp!, {r0, r1}
BX lr
C local variables

Is the given C and ARM code equivalent?

A. Yes
B. No

foo:

MOV r0, #10
MOV r1, #20
STMDB sp!, {r0, r1}
ADD r0, r0, r1
BX lr

int foo()
{
    int i=10, j=20;
    return i+j;
}
int read_and_sum() {
    int l1, l2;
    scanf("%d%d", &l1, &l2);
    return l1 + l2;
}

read_int:
    push {r4-r11, ip, lr}
    sub sp, sp, #8  
        @ reserve 8 bytes for the 2 local integers

    ldr r0, =scan_ints  
        @ format string
    mov r1, sp          
        @ address of l1
    add, r2, sp, #4     
        @ address of l2
    bl scanf

    ldr r0, [sp]        
        @ read l1
    ldr r1, [sp, #4]    
        @ read l2
    add r0, r0, r1

    add sp, sp, #8      
        @ de-allocate 8 bytes, restore sp to
        @ value after push
    pop {r4-r11, ip, lr}
    bx lr