Chapter 10
And, Finally...
The Stack

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

Stack: An Abstract Data Type

- 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

Stacks

A Physical Stack

A coin rest in the arm of an automobile

Initial State

After One Push

After Three More Pushes

After One Pop

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

Data items move between registers

<table>
<thead>
<tr>
<th>Initial State</th>
<th>After One Push</th>
<th>After Three More Pushes</th>
<th>After Two Pops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty: Yes</td>
<td>Empty: No</td>
<td>Empty: No</td>
<td>Empty: No</td>
</tr>
</tbody>
</table>

By convention, R6 holds the Top of Stack (TOS) pointer.

A Software Implementation

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

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 ; increment stack pointer

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 x3FF
LDR R0, R6, #0 ; Stack not empty (POP)
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 before adding data.
  - Return status code in R5 (0 for success, 1 for overflow)

```
PUSH R1, MAX ; MAX = -x3F
ADD R2, R6, R1 ; Compare stack pointer
BRz FAIL ; with x3F
ADD R6, R6, #1 ; Stack not full (PUSH)
STR R0, R6, #0
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R6, R6, #-1
RET
MAX .FILL xc005
```

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.
- Small instruction size
- Convenient calling convention for subroutines.
- Algorithm naturally expressed using FIFO data structure.

Example: OpAdd

- POP two values, ADD, then PUSH result.

```
OpAdd JSR POP
ADD R5,R5,#0 ; Get first operand.
BRp Exit ; Check for POP success.
ADD R1,R0,#0 ; Make room for second.
JSR POP ; Get second operand.
ADD R5,R5,#0 ; Check for POP success.
BRp Restore1 ; If error, 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
```

Example: OpAdd
Recall that local variables are stored on the run-time stack in an activation record.

Stack Pointer (R6) is a pointer to the location of the last item on 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.

Example:

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

double ValueInDollars(double amount, double rate)
{
    return amount * rate;
}
```

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
- pop activation record

Called function:
- execute code
- put result in activation record
- return
Activation Record

```c
int NoName(int a, int b)
{
    int w, x, y;
    ...
    return y;
}
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>int</td>
<td>4</td>
<td>NoName</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
<td>5</td>
<td>NoName</td>
</tr>
<tr>
<td>w</td>
<td>int</td>
<td>0</td>
<td>NoName</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>-1</td>
<td>NoName</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>-2</td>
<td>NoName</td>
</tr>
</tbody>
</table>

- **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**

Example Function Call

```c
int Volta(int q, int r)
{
    int k;
    int m;
    ...
    return k;
}
```

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

Calling the Function

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

- **w (param)**
- **q (param)**
- **r (param)**
- **w (local)**
- **dyn link**
- **ret addr**
- **ret val**
- **a (param)**

Note: Caller needs to know number and type of arguments, doesn’t know about local variables for function being called.
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
; allocate space for locals
ADD R5, R6, #-1
; allocate space for locals
ADD R6, R6, #-2

Ending the Callee Function

; return k;
; copy k into return value
return k;
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

Resuming the Caller Function

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

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 sets up new R5; 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.