Chapter 14

Functions & Activation Records
Function

- Smaller, simpler, subcomponent of program
- Provides abstraction
  - hide low-level details
  - give high-level structure to program,
    easier to understand overall program flow
  - enables separable, independent development
- C functions
  - zero or multiple arguments passed in
  - single result returned (optional)
  - return value is always the same type
  - by convention, only one function named main (this defines the initial PC)
- In other languages, called procedures, subroutines, ...
Functions in C

- A function consists of a
  - Declaration (a.k.a prototype)
    - includes return value, function name, and the order and data type of all arguments – names of argument are optional
  - Calls
  - Definition
    - Names of variables do not need to match prototype, but must match order/type
    - Defines functionality (source code) and returns control (and a value) to caller
    - May produce side-effects

- Declaration
  ```c
  int Factorial (int n);
  ```

- Call
  ```c
  a = x + Factorial (f + g);
  ```

- Definition
  ```c
  int Factorial(int n){
    int i;
    int result = 1;
    for (i = 1; i <= n; i++)
      result *= i;
    return result;
  }
  ```
Why Declaration?

● Since function definition also includes return and argument types, why is declaration needed?

● Use might be seen before definition.
  – Compiler needs to know return and arg types and number of arguments.

● Definition might be in a different file, written by a different programmer.
  – include a "header" file with function declarations only
  – compile separately, link together to make executable

● What if we need to pass more parameters than we have registers?!?

● All the compiler needs to know
  – (1) what symbol to call the function (its name),
  – (2) the type of the return value (for proper usage/intermediate storage)
  – (3) how to set up the stack to call the function (arguments)

● This allows the compiler to inserted the necessary code for function activation when it is called
Implementing a Function Call: Overview

- Making a function call involves three basic steps
  - The parameters of the call are passed from the caller to the callee
  - The callee does its task
  - A return value is returned to the caller

- Functions in C are caller-independent
  - Every function must be callable from any other function

- Activation record
  - information about each function, including arguments and local variables
  - also includes bookkeeping information
    - values that must always be saved by the caller before doing the JSR
  - arguments and bookeeping info pushed on run-time stack by caller
  - callee responsible for other uses of stack (such as local variables)
  - popped of the stack by caller after getting return value

- R5 (the frame pointer) is the “bottom” of the stack as far as the callee function definition is aware.
  - It should not modify anything existing on the stack before its execution!
Implementing a Function Call: Overview

### Calling function
- Allocate memory for activation record (push)
- Copy arguments into stack
- Call function
- Get result from stack (pop)
- Deallocate memory for activation record (pop)

### Called function
- Allocate space for return value
- [Bookkeeping] (push)
- Store mandatory callee save registers [bookkeeping] (push)
- Set frame pointer
- Allocate local variables (push)
- Execute code
- Put result in return value space
- Deallocate local variables (pop)
- Load callee save registers (pop)
- Return
Activation Record

• int funName(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>funName</td>
</tr>
<tr>
<td>b</td>
<td>int</td>
<td>5</td>
<td>funName</td>
</tr>
<tr>
<td>w</td>
<td>int</td>
<td>0</td>
<td>funName</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>-1</td>
<td>funName</td>
</tr>
<tr>
<td>y</td>
<td>int</td>
<td>-2</td>
<td>funName</td>
</tr>
</tbody>
</table>
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 to protect it from change in case another function (JSR) is called

- **Dynamic link**
  - caller’s frame pointer (basically a caller save of its own frame pointer R5)
  - used to “pop” this activation record from stack

- In the LC-3 the format for an activation record includes only (and always) these three words pushed in that order!
Run-Time Stack

Before call

During call

After call

Caller

Callee

Memory

R6

R5
Example Function Call

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

int Caller (int a) {
    int w = 25;
    w = Callee (w,10);
    return w;
}
```
The Function Call: Preparation to Jump

- int Callee (int q, int r) ...
- int Caller (int a) ...
- ... w = Callee(w, 10);...
- ; push second arg
  AND R0, R0, #0
  ADD R0, R0, #10
  ADD R6, R6, #-1
  STR R0, R6, #0
- ; push first argument
  LDR R0, R5, #0
  ADD R6, R6, #-1
  STR R0, R6, #0
- JSR CALLEE ; Jump!

Note: Caller needs to know number and type of arguments, doesn't know about local variables.
Starting the Callee Function

- ; push space for return value
  ADD  R6, R6, #1
- ; push return address
  ADD  R6, R6, #1
  STR  R7, R6, #0
- ; push dyn link (caller’s R5)
  ADD  R6, R6, #1
  STR  R5, R6, #0
- ; set new frame pointer
  ADD  R5, R6, #1
- ; allocate space for locals
  ADD  R6, R6, #1
- ... execute body of function

Note: Callee makes room for bookkeeping variables (standard format) and its local variables (as necessary)
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, #1
  ; pop dynamic link (into R5)
  LDR R5, R6, #0
  ADD R6, R6, #1
  ; pop return addr (into R7)
  LDR R7, R6, #0
  ADD R6, R6, #1
  ; return control to caller
  RET`

```
R6 → -43
R5 → 217
new R6 → 217
new R5 → 25
xFD00
m  k
dyn link ret addr
ret val
q  r
w  10
dyn link ret addr
ret val
```

```
x3100
```
Resuming the Caller Function

- \( w = \text{Callee}(w,10); \)

- ; JSR Callee    ; LAST COMMAND

  ; load return value (top of stack)
  LDR   R0, R6, #0

  ; perform assignment of return value
  STR   R0, R5, #0

  ; pop return value and arguments
  ADD   R6, R6, #3

  ...

Continue caller code
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 (first to last).
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 (RET/JMP R7).
9. **Caller** loads return value and pops arguments.
10. **Caller** resumes computation...
In Summary: The Stack

- Since our program usually starts at a low memory address and grows upward, we start the stack at a high memory address and work downward.

- Purposes
  - Temporary storage of variables
  - Temporary storage of program addresses
  - Communication with subroutines
    - Push variables on stack
    - Jump to subroutine
    - Clean stack
    - Return
Parameter passing on the stack

- If we use registers to pass our parameters:
  - Limit of 8 parameters to/from any subroutine.
  - We use up registers so they are not available to our program.
- So, instead we push the parameters onto the stack.
  - Parameters are passed on the stack
  - Return values can be provided in registers (such as R0) or on the stack.
  - Generally, only R6 should be changed by a subroutine.
    - Other registers that are changed should must be callee saved/restore.
    - Subroutines should be transparent
- Both the subroutine and the main program must know how many parameters are being passed!
  - In C we would use a prototype: `int power (int number, int exponent);`
- In assembly, you must take care of this yourself.
- After you return from a subroutine, you must also clear the stack.
  - Clean up your mess!
Characteristics of good subroutines

- **Readability** – well documented.
- **Generality** – can be easily reused elsewhere
  - Passing arguments on the stack does this.
- **Transparency** – you have to leave the registers like you found them, except R6.
  - Registers must be callee saved.
- **Re-entrant** – subroutine can call itself if necessary
  - Store all information relevant to specific execution to non-fixed memory locations
    - The stack!
  - This includes temporary callee storage of register values!
- **Secure** – no undocumented side-effects on the stack / memory.
Know how to...

- Push parameters onto the stack
- Access parameters on the stack using base + offset addressing mode
- Draw the stack to keep track of subroutine execution
  - Parameters
  - Return address
- Clean the stack after a subroutine call
Practice problems

- 14.2, 14.4, 14.9, 14.10, 14.15 (good!)
- The convention in LC-3 assembly is that all registers are callee-saved except for R5 (the frame pointer) R6 (the stack pointer) and R7 (the return link).
  - Why is R5 not callee-saved?
  - Why is R6 not callee-saved?
  - Why is R7 not callee-saved?
- Is it true that any problem that can be solved recursively can be solved iteratively using a stack data structure? Why or why not?
main() {
    int i, j, k;

    i = 5;
    j = 3;
    ...
    k = sub1(i, j);
    ...
}

int sub1(a, b) {
    int x, y;
    ...
    x = a;
    y = sub2(x, 3);
    return y;
}

int sub2(var1, var2) {
    int temp;
    ...
    temp = var1 - var2;
    return temp;
}