Chapter 14
Functions

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 (or parameters) passed in**
- **Single result returned (optional)**
- **Return value is always a particular type**

In other languages, called procedures, subroutines, ...

Example of High-Level Structure

```
main()
{
  setup_board();  /* place pieces on board */

  determine_sides(); /* choose black/white */

  /* Play game */
  while (no_outcome_yet()){
    whites_turn();
    blacks_turn();
  }
}
```

Structure of program is evident, even without knowing implementation.

Functions in C

Definition

type of return value

name of function

types of all arguments

exits function with specified return value

```
int factorial(int n)
{
  int i;
  int result = 1;
  for (i = 1; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

Function call -- used in expression

```
a = x + factorial(f + g);
```

1. evaluate arguments

2. execute function

3. use return value in expression

Implementing Functions and Variables in LC-3

We've talked about…

- Variables
  - Local
  - Global
- Functions
  - Parameter passing
  - Return values

What does the assembly code look like for these idioms?

Important notes

- Different compilers for different ISAs do things differently
- As long as a compiler is consistent
- We're straying from the book's version to simplify things
  - Leaving out the R5 "frame pointer"

Allocating Space for Variables

Global data section

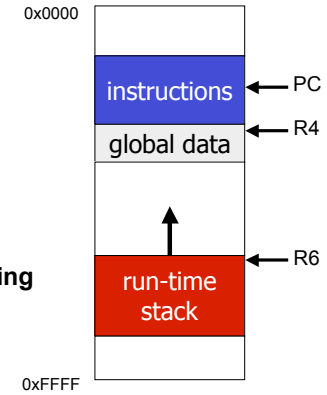
- All global variables stored here (actually all static variables)
- R4 points to beginning

Run-time stack

- Used for local variables
- R6 points to top of stack
- New frame for each block (goes away when block exited)

Offset = distance from beginning of storage area

- Global: `LDR R1, R4, #4`
- Local: `LDR R2, R6, #3`

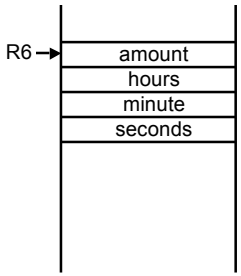


Local Variable Storage

Local variables stored in *activation record* (*stack frame*)

Symbol table "offset" gives the distance from the base of the frame

- A new frame is pushed on the run-time stack each time block is entered
- R6 is the stack pointer – holds address of current top of run-time stack
- Because stack grows downward, stack pointer is the smallest address of the frame, and variable offsets are >= 0.



Symbol Table

Compiler tracks each symbol (identifiers) and its location

- In assembler, all identifiers were labels
- In compiler, identifiers are variables

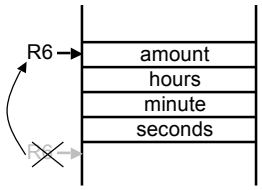
Compiler keeps more information

- Name (identifier)
- Type
- Location in memory
- Scope

Name	Type	Offset	Scope
amount	double	0	main
hours	int	1	main
minutes	int	2	main
seconds	int	3	main

Symbol Table Example

```
int main()
{
   int seconds;
   int minutes;
   int hours;
   double amount;
   ...
}
```



Name	Type	Offset	Scope
amount	double	0	main
hours	int	1	main
minutes	int	2	main
seconds	int	3	main

Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal & ~inGlobal;
  outLocalB = (inLocal + inGlobal) + outLocalB;

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n",
         outLocalA, outLocalB);
}
```

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	2	main
outLocalA	int	1	main
outLocalB	int	0	main

Example: Code Generation

```
; main
; inLocal = 5
     AND R0, R0, #0
     ADD R0, R0, #5   ; inLocal = 5
     STR R0, R6, #2   ; (offset = 2)

; inGlobal = 3
     AND R0, R0, #0
     ADD R0, R0, #3   ; inGlobal = 3
     STR R0, R4, #0   ; (offset = 0)
```


Example (continued)

```
; first statement:
; outLocalA = inLocal & ~inGlobal;

   LDR R0, R6, #2   ; get inLocal(offset = 2)
   LDR R1, R4, #0   ; get inGlobal
   NOT R1, R1       ; ~inGlobal
   AND R2, R0, R1   ; inLocal & ~inGlobal
   STR R2, R6, #1   ; store in outLocalA
                    ; (offset = 1)
```

Example (continued)

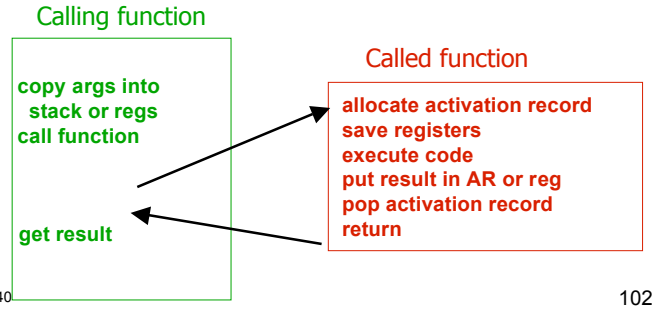
R0

```
;outLocalB = (inLocal + inGlobal) + outLocalA;

      LDR R0, R6, #2   ; inLocal
      LDR R1, R4, #0   ; inGlobal
      ADD R0, R0, R1   ; R0 is sum

      LDR R1, R6, #1   ; outLocalA
      ADD R2, R0, R1   ; R2 is sum
      STR R2, R6, #0   ; outLocalB (offset = 0)
```

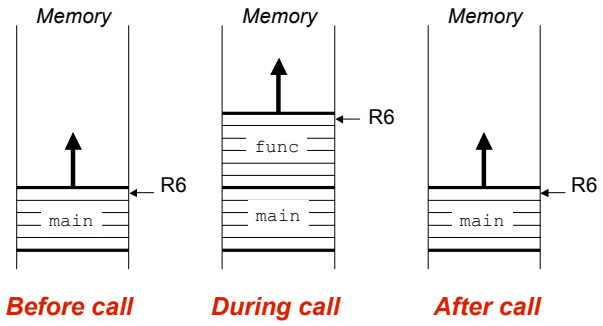

Implementing Functions

Activation record

- Information about each function, including arguments and local variables
- Also stored on run-time stack



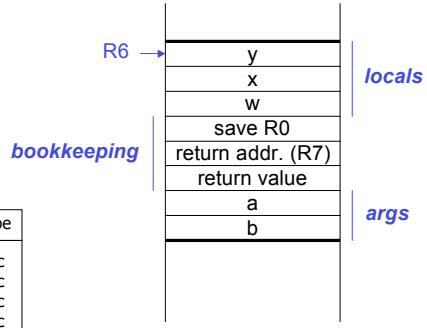
Run-Time Stack for Functions



Activation Record

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

Name	Type	Offset	Scope
b	int	7	func
a	int	6	func
"ret. value"	int	5	func
w	int	2	func
x	int	1	func
y	int	0	func



Activation Record Bookkeeping

Return value

- Space for value returned by function
- Allocated even if function does not return a value

Return address

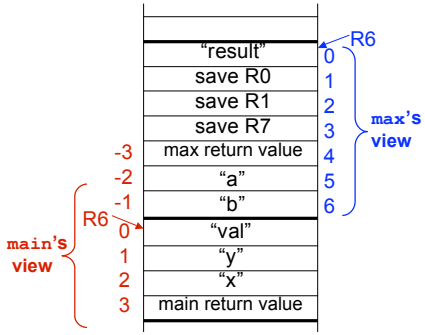
- Save pointer to next instruction in calling function
- Convenient location to store R7
  - ➢ in case another function (JSR) is called

Save registers

- Save all other registers used (but not R6, and often not R4)

Function Call Example

```
int main()
{
  int x, y, val;
  x = 10;
  y = 11;
  val = max(x + 10, y);
  return val;
}

int max(int a, int b)
{
  int result;
  result = a;
  if (b > a) {
    result = b;
  }
  return result;
}
```



Main Function (1 of 2)

```
MAIN    ADD R6, R6, #-4     ; allocate frame
        AND R0, R0, #0      ; x = 10
        ADD R0, R0, #10
        STR R0, R6, #2

        AND R0, R0, #0      ; y = 11
        ADD R0, R0, #11
        STR R0, R6, #1

        LDR R0, R6, #1      ; load y into R0
        STR R0, R6, #-1     ; 2nd argument

        LDR R1, R6, #2      ; load x into R1
        ADD R1, R1, #10     ; R1 = x + 10
        STR R1, R6, #-2     ; 1st argument

        JSR MAX             ; call max function

        … ; more here
```


Max Function

```
MAX     ADD R6, R6, #-7     ; allocate frame
        STR R7, R6, #3      ; save R7 (link register)
        STR R1, R6, #2      ; save R1
        STR R0, R6, #1      ; save R0

        LDR R0, R6, #5      ; load "a"
        STR R0, R6, #0      ; store "a" into "result"

        LDR R1, R6, #6      ; load "b"
        NOT R1, R1          ; calculate -b
        ADD R1, R1, 1
        LDR R0, R6, #5      ; load "a"
        ADD R0, R1, R0      ; compare

        BRp AFTER
        LDR R0, R6, #6      ; load "b"
        STR R0, R6, #0      ; store "b" into "result"

AFTER   LDR R0, R6, #0      ; load "result"
        STR R0, R6, #4      ; store "result" into return value

        LDR R0, R6, #1      ; restore R0
        LDR R1, R6, #2      ; restore R1
        LDR R7, R6, #3      ; restore R7 (link register)
        ADD R6, R6, #7      ; pop stack
        RET
```

Main Function (2 of 2)

```
; previous code here

JSR MAX            ; call max function


LDR R0, R6, #-3    ; read return value of max
STR R0, R6, #0     ; put value into local "val"

LDR R0, R6, #0     ; load "val"
STR R0, R6, #3     ; put "val" into main's
                   ; return value

ADD R6, R6, #4     ; pop stack
RET
```


Summary of LC-3 Function Call Implementation

1. Caller places arguments on stack (last to first)
2. Caller invokes subroutine (JSR)
3. Callee allocates frame
4. Callee saves R7 and other registers
5. Callee executes function code
6. Callee stores result into return value slot
7. Callee restores registers
8. Callee deallocates frame (local vars, other registers)
9. Callee returns (RET or JMP R7)
10. Caller loads return value
11. Caller resumes computation…

Callee versus Caller Saved Registers

Callee saved registers

- In our examples, the callee saved and restored registers
- Saves/restores any registers it modifies

What if a you wants R7 to be preserved across a call?

- Before call: caller saves it on the stack
- After call: caller restores it from the stack

Caller saved registers

- R7 is an example of a caller saved register
- Value assumed destroyed across calls
- Only needs to save R7 when it's in use

Which is better? Callee or Caller saved registers?

- Neither: many ISA calling conventions specify some of each

Compilers are Smart(er)

In our examples, variables always stored in memory

- Read from stack, written to stack

Compiler will perform code optimizations

- Keeps many variables in registers
- Avoids many save/restores of registers
- Why?

Passing parameter values in registers

- First few parameters in registers
- Return value in register
- Like in your homework projects
- Again, why?