Chapter 14
Functions

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Palsetia

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, ...

Functions in C

Definition

```
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);
```


How do C Functions translate to assembly

To see how subroutines are implemented at machine level

- Out target ISA is LC3

Core operations:

1) callers passes parameters to callee and the control is transferred to callee

2) the callee does its task

3) a return value is passed back to caller and control returns to the caller

Passing Arguments & Returns

Using a stack

- **Calling routine pushes arguments on the stack and calls the subroutine**
- **Called routine uses passed arguments from the stack and pushes any return value onto the stack**
- **Finally, the calling routine then retrieves the return value(s) from the stack and continues execution**

Function Call scenario

```
Main    . . .
        JSR     Foo
Next    . . .
     HALT

Foo     ST      R7, SaveR7
        AND     R0, R0, #0
        . . .
        JSR     Foo
After   . . .
        LD      R7, SaveR7
        RET
Save7   .FILL   #0
Counter .FILL   #0

     .END
```

- First call to Foo (SaveR7 contains address of Next)
- Second call to Foo (SaveR7 contains address of After)
- First return from Foo (returns to After)
- Second return from Foo (returns to After again!!!)
- Problem: we cannot call foo() from foo() – this known as recursive call

Solution to Recursion problem

For each subroutine call, need a mechanism to distinguish its invocation

- **This is known as *activation record***

Activation Record contains

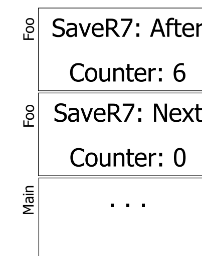
- **Invocation-specific data (*e.g.,* local variables, saved registers, arguments and returns)**
- **Need to store this information per function call and discard when the function call is over**
- **Stack data structure fits this description well**
  - **When function is called we store (push) record on the stack**
  - **When function call is over we discard (pop) record from the stack**

Function foo() and activation record

```
Main    . . .
        JSR     Foo
Next    . . .
     HALT

Foo     ST      R7, SaveR7
        AND     R0, R0, #0
        . . .
        JSR     Foo
After   . . .
        LD      R7, SaveR7
        RET
Save7   .FILL   #0
Counter .FILL   #0

     .END
```



Counter is local variable of foo()

Important

My slides **deviate from books explanation on how function call mechanism works**. Most real world implementation have the mechanism explained in the next couple of slides.

Recap: Compiler's 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

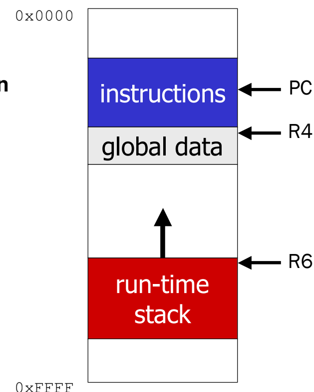
Name (identifier)
Type
Location in memory
Scope

Name	Type	Offset	Scope

Local variable storage

Now we know that each function gets its own **activation record**

Since local variables stored are stored in *activation record* which stored on the stack, implies that **local variables are stored on the stack**



Frame Pointer

- Each function gets a record (a.k.a frame), but we need to somehow track where in the activation record are the local variables, book keeping info, arguments etc.
- This tracking is done by what is called the **frame pointer**. Frame pointer is the address which points **to the top of the record/frame**.
- Once we know the frame pointer, local variables accessed via frame pointer + offset
- By convention in LC3, R5 holds frame pointer (i.e. add of the first local variable). Hence the compiler only keeps track of the offset
- **Every function will have different value frame pointer**
  - We will see why in a bit

Upon every function

```
int main()
{
  double amt = 0;
  int p = 0;
  int t = 0;
  int r = 0;
 //get p, t, r
  amt = p * t * r;
}
```

The stack is adjusted to accommodate the record entries

Therefore R5 = R6 – (#number of entries to be pushed onto stack)

- **Return value**
- **Return Addr & other register**
- **Local variables**
- **Plus one more thing (coming soon)**

So in this example: The number of entries to push on to stack is 4(amt, p, t, r)

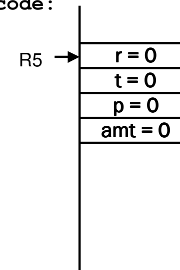
So R5 = R6 – 4;

The variable added last is always on the top of the stack (LIFO)

Example local variables of main()

```
int main()
{
   double amt = 0;
   int p = 0;
   int t = 0;
   int r = 0;
 //get p, t, r
  amt = p * t * r;
}
```

```
LC3 initialization code:
ADD R6, R6, #-4
ADD R5, R6, #0
ADD R0, R0, #0
STR R0, R5, #0
STR R0, R5, #1
STR R0, R5, #2
STR R0, R5, #3
```



Compiler's Symbol Table

Name	Type	Offset	Scope
amt	double	3	main
p	int	2	main
t	int	1	main
r	int	0	main

We know that R5 contains the frame pointer of function main(). Now variables in main can accessed by :

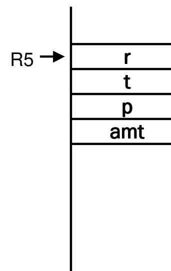
Frame Pointer + Offset

Example local variables of main() (contd..)

To calculate amt, we can now easily access local variables p,t, and r using the frame pointer R5

- **Assigning amt = p * t * r ;**

```
LDR   R1, R5,  #2  ;R1 = p
LDR   R2, R5,  #1  ; R2 = t;
LDR   R3, R5,  #0  ; R3 = r
MUL  R1,  R1, R2  ; R1 = p * t
MUL  R1,  R1, R3  ; R1 = p * t * r
STR  R1,  R5,  #3  ;  Store R1 = amt
```



Compiler's Symbol Table

Name	Type	Offset	Scope
amt	double	3	main
p	int	2	main
t	int	1	main
r	int	0	main

Complete view of an activation record

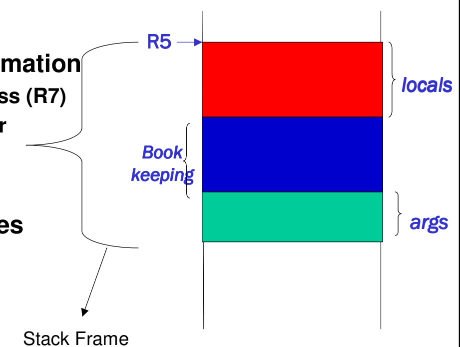
Activation Record of a called function stores:

1. local variables

2. book keeping information

- **Callers Return address (R7)**
- **Callers Frame pointer**

3. input parameters passed & return values



Stack Frame

Activation Record Bookkeeping

Callers Frame Pointer

- Need to save callers frame pointer so that when the control returns to the caller, it will be able to access its own local variable
- If we destroy this value then we have trouble restarting the caller correctly when the callee finishes

Return address

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

Saved Registers

Save all registers that will be used for temporary work in the function

Returns

Every function always allocates a space for return value on the activation record, whether or not it returns anything (i.e. even though the function return is void)

Activation Record Arguments

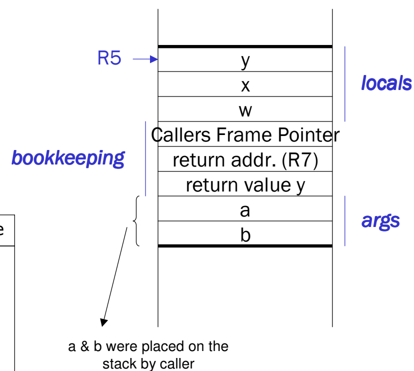
Arguments

- The calling routine places the arguments on top of the stack before calling a function
- The callee will use the arguments from the stack to do its work

Example of a Activation Record of a function

```
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



So why do we need R5 where we could just use R6?

It looks likes R5 is top of the frame as well as top of the stack. So how did we introduce R5 (Frame Pointer)?

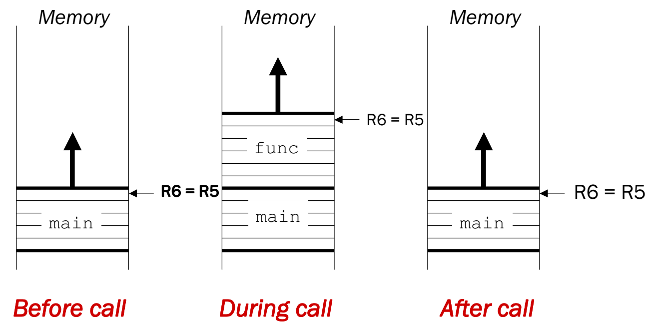
Remember that the stack space can be used to store all immediate data, there no restriction that it is used only for activation records (unless otherwise stated in the ISA manual.

We could be using stack to hold temporary data performed during function call. Example: Reversing a string (midterm problem)

This often involves pushing and popping values onto and off of the stack. If stack pointer (R6) keeps changing, it would be hard to access local variables at a fixed location on the stack (basically compilers offset will be get muddled up if R6 is the reference).

To make things easy for compilers (and for human assembly language programmers) it is convenient to have a frame pointer that does not change its value while a subroutine is active. The variables will always be the same distance from the unchanging frame pointer.

Big Picture

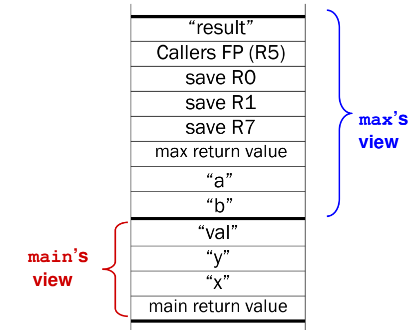


CIT 593 14 - 21

Function Call Example

```
int main()
{
  int x, y, val;
  x = 10;
  y = 11;
  val = max(x + 10, y);
  printf("%d", val);
  return 0;
}

int max(int a, int b)
{
  int result = 0;
  if (b > a) {
    result = b;
  }
  else {
      result = a;
  }
  return result;
}
```



Why haven't I shown save R7 and frame pointer R5 for main() ?

CIT 593 14 - 22

Main Function (1 of 2)

```
MAIN    ;R6 is initialized STACKBASE by OS code in LC3
        ADD R6, R6, #-1     ; Space for return value

        AND R0, R0, #0      ; x = 10
        ADD R0, R0, #10
        ADD R6, R6, #-1
        STR R0, R6, #0

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

        ADD R6, R6, #-1 ;   space for variable "val"
      ;Now we have push the record on the stack
        ADD R5, R6, #0     ; R5 = Main's Frame Pointer

        LDR R0, R5, #1      ; load y into R0
        ADD R6, R6, #-1
        STR R0, R6, #0      ;arg for max() is put on stack

        LDR R1, R5, #2     ; load x into R1
        ADD R1, R1, #10     ; R1 = x + 10
        ADD R6, R6, #-1
        STR R1, R6, #0      ;arg for max() is put on stack

        JSR MAX             ; call max function

        … ; more here (Main Function (2 of 2)
```

Local variables of main() (brace: x = 10 and y = 11 blocks)

Args for callee (i.e. MAX()) (brace: argument push blocks)

CIT 593 14 - 23

Max Function (1 of 3)

```
MAX     ADD R6, R6, #-1 ;allocate spot for return var

        ADD R6, R6, #-1    ;
        STR R7, R6, #0     ; save R7 (link register)

        ;we will be using R1 and R0
        ;within func so we must save them
        ADD R6, R6, #-1
        STR R1, R6, #0     ; save R1
        ADD R6, R6, #-1
        STR R0, R6, #0     ; save R0

        ADD R6, R6, #-1
        STR R5, R6, #0     ; Save Main's Frame Pointer

        AND R0, R0, #0
        ADD R6, R6, #-1
        STR R0, R6, #0     ;Max's local var (result = 0)

        ADD R5, R6, #0    ; Max's Frame Pointer

        LDR R0, R5, #6     ; load "a"

        LDR R1, R5, #7     ; load "b"
```

CIT 593 14 - 24

Max Function (2 of 3)

```
        NOT R0, R0        ; calculate -a
        ADD R0, R0, #1

        ADD R0, R1, R0    ; compare (i.e. b - a > 0)
        BRp BGRTA

        LDR R0, R5, #6    ; load "a"
        STR R0, R5, #0    ; store "a" into "result"

BGRTA   STR R1, R5, #0    ; store "b" into "result"

        LDR R1, R5, #0    ;load "result"
        STR R1, R5, #5    ;store "result" into return
                          ;value

        ;pop of local variable of max()
        ADD R6, R6, #1
```


Max Function (3 of 3)

```
;restore registers
LDR R5, R6, #0    ; restore R5 (main's frame pointer)
ADD R6, R6, #1    ; pop the SAVE R5 (Mains' FP)

LDR R0, R6, #0    ; restore R0
ADD R6, R6, #1    ; pop SAVE R0

LDR R1, R6, #0    ; restore R1
ADD R6, R6, #1    ; pop SAVE R1

LDR R7, R6, #0    ; restore R7 (link register)
ADD R6, R6, #1    ;pop the return address (SAVE R7)

RET
```


Main Function (2 of 2)

```
; previous code here
JSR MAX             ; call max function

LDR R0, R6, #0    ; read return value of max
STR R0, R5, #0    ; put value into local "val"

;Do code for printf()

AND R0, R0, #0
STR R0, R5, #3    ;"return 0" for main's exit

ADD R6, R5, #3    ; pop of main's local variables

HALT
```


Summary of LC-3 Function Call Implementation

1. Caller places arguments on stack
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

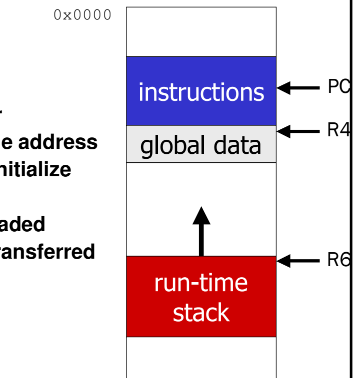
Callee saved registers
- In our examples, the callee saved and restored registers
- Saves/restores any registers it modifies

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

Allocating Space for Global Variables

Global data section
- All global variables stored here (actually all static variables)
- The memory address where global data section starts is kept a register
  - In LC3, we assume R4 holds the address
  - So programmer access R4 to initialize and assign global variables
  - The initial reference of R4 is loaded by the OS before the control is transferred to user code



Note:

R4 is just a convention in LC3, could be different for different ISAs

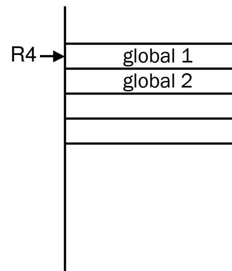
Example of Global Variable

- Example of C program:

```
int global1 = 0;
int global2 = 0;
int main(){…}
```

- LC3 Equivalent code

```
AND   R0, R0, #0;
LDR   R0, R4, #0;
LDR   R0, R4, #1;
```



Compiler's Symbol table

Identifier	Type	Location (offset)	Scope	
global1	global	0	global	
global2	global	1	global	