Functions in C

- Global data pointer: R4
  - Global and static variables
  - Specify positive offsets
- Frame pointer: R5
  - Points to current code block
  - Negative offset
- Stack Pointer: R6
  - Top of stack
- Return address: R7

C to LC3 Code generation
- Complete and submit C to LC3 code generation
  - Linked from Lectures webpage

Memory Allocation in C

- Global data pointer: R4
- Frame pointer: R5
- Stack Pointer: R6
- Return address: R7

Next.
- How to handle function calls?
  - Caller save and callee save concepts again!
  - Where to store the data?
- Example C code translated to LC3
- Finally: what can go wrong?
Implementing Functions

- Functions in C
- Implementing Functions in C
  - Run-time Stack, Getting It All to Work, Tying it All Together
- Recursion – how?

Functions in C

- Declaration (also called prototype)
  - int Factorial(int n);
- Function call -- used in expression
  - int Factorial(f + g);

Function Definition

- State type, name, types of arguments
  - must match function declaration
  - give name to each argument (doesn’t have to match declaration)
- int Factorial(int n)
  - int i;
  - int result = 1;
  - for (i = 1; i < n; i++)
    - result *= i;
  - return result;

Functions calling functions...

- int mult(int a, int b) {
  int c = 0;
  while (b > 0) {
    c = c + a;
    b = b - 1;
  }
  return c;
}
- int pow(int a, int p) {
  int c = 1;
  for (c = 1; p > 0; p--)
    c = mult(c, a); // performs: c = c * a
  return c;
}
- int main() {
  int a=2, b=3, c=0;
  c = pow(a, b); // performs: c = a^b
}

We’ve written our own power and “multiplication” functions
We’ll trace these through the stack in this lecture.
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
        c=c+a ;
        b=b-1 ;
    }
    return c ;
}

int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--) 
        c = mult(c, a) ;
    return c ;
}

int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
Passing Parameters "By Value"

```c
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
        c=c+a ;
        b=b-1 ;
    }
    return c ;
}
```

```c
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
        c = mult(c, a) ;
    return c ;
}
```

```c
int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

How do we keep locals vars, parameters, arguments, RVs straight?

```
Passing Parameters "By Value"
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
        c=c+a ;
        b=b-1 ;
    }
    return c ;
}
```

```c
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
        c = mult(c, a) ;
    return c ;
}
```

```c
int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

Locals, Parameters, Arguments, and Return Values

```
int mult(int a, int b) {
    int c=0 ;
    while (b > 0) {
        c=c+a ;
        b=b-1 ;
    }
    return c ;
}
```

```
int pow(int a, int p) {
    int c ;
    for (c = 1; p > 0; p--)
        c = mult(c, a) ;
    return c ;
}
```

```c
int main() {
    int a=2,b=3,c=0;
    c = pow (a, b) ; // performs: 2^3
}
```

Function calls.. What needs to be done?

- Caller can pass parameters to the function
- Function returns a value
- Function needs to return to caller
  - PC needs to be stored
  - "pointer" to variables used by caller needs to be restored
- Function uses local variables, so allocate space for these variables
  - New scope (i.e., new frame pointer)
- Function can be called from another function...

- model this behaviour and capture all this information in an Activation Record
Activation Record/Stack Frame

- Function call results in activation record pushed on stack
- Function return results in activation record popped off stack
- Allows for recursion
- Place to keep
  - Parameters
  - Local (auto) variables
  - Register spillage
  - Return address
  - Return value

Run-Time Stack

- Recall that local variables are stored on the run-time stack in an activation record
- Frame pointer (R5) points to the beginning of a region of 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.

Run-Time Stack

- Before call
- During call
- After call

Stack Frames (Activation Records)

- Stack generally managed in function-sized chunks
  - Called frames or activation records
  - Function call "pushes" frame of called function onto stack
  - Function return "pops" frame off of stack
  - This all happens in run-time

In C, top-level user function is always 'main'.
Frame Pointer (R5) and Stack Pointer (R6)

- LC3 uses two more registers as part of calling convention
  - R6 is the stack pointer (SP), “points to” current “top” of stack
  - R5 is the frame pointer (FP), “points to” bottom of current frame
    - Sometimes called base pointer (BP)

```
main
R5
pow
R6
```

Finally ... Stack Frame Layout (Activation Records)

- In caller’s stack frame: addresses > R5
  - Caller’s saved frame pointer
  - return address/value
  - arguments
- In running function’s stack frame: addresses <= R5
  - Local variables
  - temporaries
  - arguments to running function’s callees

```
temperaries, arguments to callees
R6
local variables
R5
caller’s frame pointer
return address
return value
arguments
```

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
- Dynamic link
  - caller’s frame pointer
  - used to pop this activation record from stack

```
R5
```

Activation Record

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

```
Name     Type Offset Scope
a        int   4    NoName
b        int   5    NoName
w        int   0    NoName
x        int   -1   NoName
y        int  -2    NoName
```
Calling Convention

- Compilers typically compile functions separately
  - Generate assembly for main(), mult(), and pow() independently
  - Why? They may be in different files, mult may be in a library, etc.

- This necessitates use of calling convention
  - Some standard format for arguments and return values
  - Allows separately compiled functions to call each other properly
  - In LC-3, we’ve seen part of its calling convention:
    - R7 = return address

- Calling convention is function of HLL, ISA, and compiler
  - Why code compiled by different compilers may not inter-operate

Caller and Callee: Who does what?

- Caller
  - Puts arguments onto stack (R→L)
  - Does a JSR (or JSRR) to function

- Callee
  - Makes space for Return Value and Return Address (and saves Return address)
  - makes space for and saves old FP (Frame Pointer)
    - Why?
  - Makes FP point to next space
  - Moves SP enough for all local variables
  - Starts execution of "work" of function

Who does what?

- Caller (continued)
  - As registers are needed their current contents can be spilled onto stack
  - When computation done...
    - Bring SP back to base
    - Restore FP (adjust SP)
    - Restore RA (adjust SP)
    - Leave SP pointing at return value
    - RET

- Callee
  - Grabs return value and does the right thing with it!
Implementing Functions: Overview

- **Activation record**
  - information about each function, including arguments and local variables
  - stored on run-time stack

**Calling function**
- push new activation record
- copy values into arguments and call function
- get result from stack

**Called function**
- Push return address and Dynamic link, return val
- execute code
- put result in activation record
- pop activation record
- return address
- return

**Example: Function Call**

```c
int main
{
    int a, b;
    ...
    b = foo(a);
    b = bar(a, b);
}

int bar(int q, int r)
{
    int k;
    ...
    return k;
}

int foo(int a)
{
    int w, x, y;
    ...
    w = bar(w, 10);
    ...
    return w;
}
```

**Example: Addresses on stack**

- At each “time”
- frame pointer for main = 3000

**Example: Calling the Function**

```c
int main
{
    int a, b;
    ...
    b = foo(a);
    b = bar(a, b);
}

int bar(int q, int r)
{
    int k;
    ...
    return k;
}

int foo(int a)
{
    int w, x, y;
    ...
    w = bar(w, y);
    ...
    return w;
}
```
CS 2461

Step 2: Calling the Function - LC3

```
● \( \text{w} = \text{bar}(x, y); \)
● ; push arguments
  JSR bar
```

Note: Caller needs to know number and type of arguments, doesn't know about local variables. It needs to push the arguments and then JSR.

Next steps: starting Callee function

- Create space for return value
- Store return address
- Store frame pointer
- Set new frame pointer
- Set space for local variables

CS 2461

Calling the Function

```
● \( \text{w} = \text{bar}(\text{w}, 10); \)
● ; 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
  ; call subroutine
  JSR \text{bar}
```

Note: Caller needs to know number and type of arguments, doesn't know about local variables. It needs to push the arguments and then JSR.

Question 4: Starting the Callee Function

```
; leave space for return value
new R6
new R5

; push return address
... ...

; push dyn link (caller's frame ptr) R6 ...

; set new frame pointer ...

; allocate space for locals ...

int bar(int q, int r) {
  int k;
  int m;
  return k;
}
```

CS 2461
**Starting the Callee Function**

- ; leave space for return value
  ADD R6, R6, #-1
- ; push return address
  ADD R6, R6, #1
  STR R7, R6, #0
- ; push dyn link (caller’s frame ptr)
  ADD R5, R5, #0
- ; set new frame pointer
  ADD R5, R6, #1
- ; allocate space for locals
  ADD R5, R6, #2

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

**Returning from function...steps**

- Write return value
- return address into R7
- Restore old frame pointer
- Pop local variables
- Where should top of stack point to after RET?

**Ending the Callee Function**

- return k;
- ; copy k into return value
  LDR R0, R5, #0
  STR R0, R6, #3
- ; pop return addr (into R7)
  ADD R6, R5, #1
- ; return control to caller
  RET
- Now we understand why swap Did not work!
  In C arguments are passed by value

**Ending the Callee Function**

- return k;
- ; copy k into return value
  LDR R0, R5, #0
  STR R0, R6, #3
- ; pop local variables
  ADD R6, R5, #1
- ; pop dynamic link (into R5)
  ADD R6, R6, #0
- ; pop return addr (into R7)
  ADD R6, R6, #1
- ; return control to caller
  RET
Back to caller...steps

- What should caller do now?
- Get return value
- Clear arguments

Resuming the Caller Function

- w = bar(w,10);
- JSR bar

  | Load return value (top of stack) |
  | LDR R0, R6, #0                  |
  | Perform assignment              |
  | STR R0, R5, #0                  |
  | Pop return value                |
  | ADD R6, R6, #1                  |
  | ADD R6, R6, #2                  |

And We’re Done...

- ...at least with that call/return sequence
  - Stack frame back to where it was
- A lot of extra instructions involved in function call
  - Many compilers try to inline functions
  - Expand function body at call site
  - Removes most of call overhead
  - Introduces other overheads
    - Multiple static copies of same function

Prologue, Body, Epilogue

- Steps at start of function that we saw are called function prologue
  - Setup code compiler generates automatically
  - One of the (few) abstractions C provides over assembly
  - More sophisticated compilers can generate tighter prologues
- Code that follows is translation of function body
  - lcc does this statement-by-statement
  - Results in many inefficiencies
  - More sophisticated compilers view entire function (at least)
- When explicit body finishes, need function epilogue
  - Cleanup code compiler generates automatically
  - Epilogue (unwinding/pop of the stack)
Things to notice

1) Arguments are pushed onto stack right-to-left
   - So that first argument from left is closest to callee
   - This is called C convention (left-to-right is called PASCAL)
   - Needed for functions with variable argument counts (e.g., printf)
2) C is pass-by-value (not pass-by-reference)
   - Functions receive “copies” of local variables
     - Recall, arguments to functions were copies of local vars
     - Protects local variables from being modified accidentally
3) We see why variables must be declared at start of function
   - Size of static/automatic variables are known at compile time:
     - ADD R6, R6, #-1 ; allocate space for local vars
     - Also, compiler may compile line-by-line, hence right up front!

Caller and Callee Saved Registers

- R5, R6, and R7 actively participate in call/return sequence
  - What happens to R0–R4 across call?
    - “Callee saved”: callee saves/restores if it wants to use them
    - “Caller saved”: caller saves/restores if it cares about values

- Turns out... LCC doesn’t have a convention for these???
  - Doesn’t have to (because it compiles statement-by-statement)
  - At the end of every statement, all local variables are on stack
  - R0-R4 are used just as “temporary” storage within expressions
    - Highly inefficient
  - Register allocation: assign locals to registers too
    - Avoid many unnecessary loads and stores to stack
      - All real compilers do this

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.
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...

For details ..

- Read Chapter 14 – section 14.3, Figure 14.8 for full implementation of the function call process
- Check out the lcc cross compiler
Question...What can go wrong?

- What if the return address was overwritten
  - Where does the program return to?
- Recall ‘privilege’ level – what if program was operating as root?
  - Will level change?
- Buffer overflow attack/stack smashing attack

What is Recursion?

- A recursive function is one that solves its task by calling itself on smaller pieces of data.
  - Similar to recurrence function in mathematics.
  - Question: how do you prove correctness of recurrence function??
  - Like iteration – can be used interchangeably; sometimes recursion results in a simpler solution.
- Example: Running sum

Mathematical Definition:
RunningSum(1) = 1
RunningSum(n) = n + RunningSum(n-1)

Recursive Function:
int RunningSum(int n) {
    if (n == 1)
        return 1;
    else
        return n + RunningSum(n-1);
}

Executing RunningSum

res = RunningSum(4);
return value = 10  RunningSum(4)
return 4 + RunningSum(3);
return value = 6  RunningSum(3)
return 3 + RunningSum(2);
return value = 3  RunningSum(2)
return 2 + RunningSum(1);
return value = 1  RunningSum(1)
return 1;

Example: Fibonacci Numbers

- Mathematical Definition:
  \[ f(n) = f(n-1) + f(n-2) \]
  \[ f(1) = 1 \]
  \[ f(0) = 1 \]

- In other words, the n-th Fibonacci number is the sum of the previous two Fibonacci numbers.
  - To compute value of \( F(n) \) we need values of \( F(n-1) \) and \( F(n-2) \)
How is recursion implemented?

- Do we need to do anything different from how we handled function calls?
  - No!
    - Activation record for each instance/call of Fibonacci!

Fibonacci: C Code

```c
int Fibonacci(int n)
{
    if ((n == 0) || (n == 1))
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Activation Records

- Whenever Fibonacci is invoked, a new activation record is pushed onto the stack.

Activation Records (cont.)

- Fibonacci(1) returns, Fibonacci(2) calls Fibonacci(0)
- Fibonacci(2) returns, Fibonacci(3) calls Fibonacci(1)
- Fibonacci(3) returns
Tracing the Function Calls

- If we are debugging this program, we might want to trace all the calls of Fibonacci.
  - Note: A trace will also contain the arguments passed into the function.

- For Fibonacci(3), a trace looks like:
  - Fibonacci(3)
  - Fibonacci(2)
  - Fibonacci(1)
  - Fibonacci(0)
  - Fibonacci(1)

- What would trace of Fibonacci(4) look like?

---

Function Calls -- Summary

- Activation records keep track of caller and callee variables
  - Stack structure
- What happens if we “accidentally” overwrite the return address?

- Next: Pointers, Arrays, Dynamic data structures and the heap

---

Fibonacci: LC-3 Code

- Activation Record
- read Chapter 17 Fibonacci example

- Compiler generates temporary variable to hold result of first Fibonacci call.