C to ASM pt. 1

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

Poll:

- Are there any topics you would like me to talk about in lecture?

Upcoming Due Dates

- HW08 (Disassembler) Due Friday 11/18 @ 11:59 pm
  - Should have everything you need

- Midterm regrade requests
  - Opens at 12:01 AM on Tuesday(11/15)
  - Close at 11:59 pm the next Tuesday (11/22)
  - Please look at the sample solution before submitting a regrade request

- **Assignments will very likely take increasingly longer to complete. Please try to not let the work accumulate**

Lecture Outline

- **Binary files & Endianness**
- Globals in ASM
- Maintaining the Stack in ASM

Binary files & Serialization

- So far this lecture has implicitly assumed we are working with files that hold text (characters)
- Binary files also exist where data isn't stored as characters. (.obj files are an example)
- Some data/data-structures make more sense to be stored in binary through a process called **serialization**.

Serialization Example:

- Posted on course website
  - read_floats.c
  - write_floats.c

- Notes:
  - Don't have to read/write an array, can read/write only one "element"
  - Trying to open these files in an editor will not be readable

Endianness

- In other architectures, there is one byte at each address location
  - For multi-byte data, how do we order it in memory?
  - Data should be kept together, but what order should it be?
  - Example, store the 4-byte (32-bit) int:
    0x A1 B2 C3 D4

- The order of the bytes in memory is called endianness
  - Big endian vs little endian

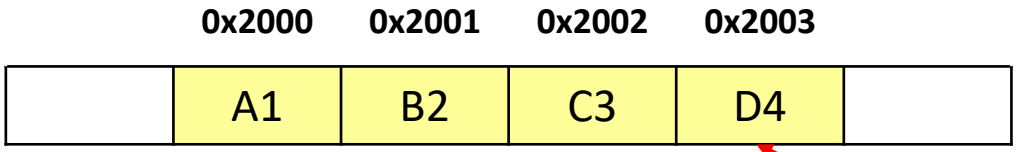
Endianness

- Consider our example 0x A1 B2 C3 D4

Most significant Byte

Least significant Byte

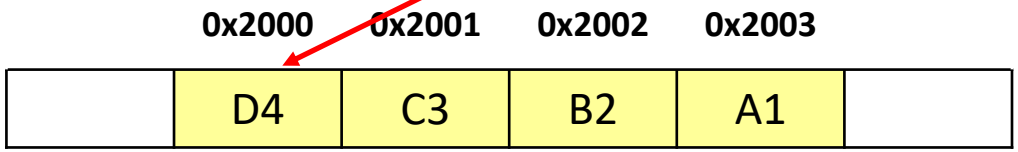
- Big endian
  - Least significant byte has highest address
  - Looks the most like what we would read
  - The standard for storing information on files/the network



- Little Endian
  - Least significant byte has lowest address
  - What your VM probably uses

Least significant Byte

Note how the hex digits within a byte are still in the same order



Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

A.

CA	DE	DA	DA

B.

DA	DA	DE	CA

C.

AC	ED	AD	AD

D.

AD	AD	ED	AC

E. I'm not sure

Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

A.

CA	DE	DA	DA

B.

DA	DA	DE	CA

C.

AC	ED	AD	AD

D.

AD	AD	ED	AC

E. I'm not sure

Endianness: Why it matters

- Since machines may store things in different byte orderings, it causes problems when they share files or communicate over the network.
- A standard ordering is used for storing binary data, big endian (often called Network ordering).
- Need to make sure that we store bytes in network byte ordering when we serialize data

Endianness functions

- There are some functions out there that convert byte orderings
  - `htons()` -> Host to Network short (16 bits)
    - Converts from Host byte ordering to network byte ordering
  - `ntohs()` -> Network to Host short (16 bits)
    - Converts from network byte ordering to host byte ordering

- "Network byte order" is big endian. Your "host" machine is little endian

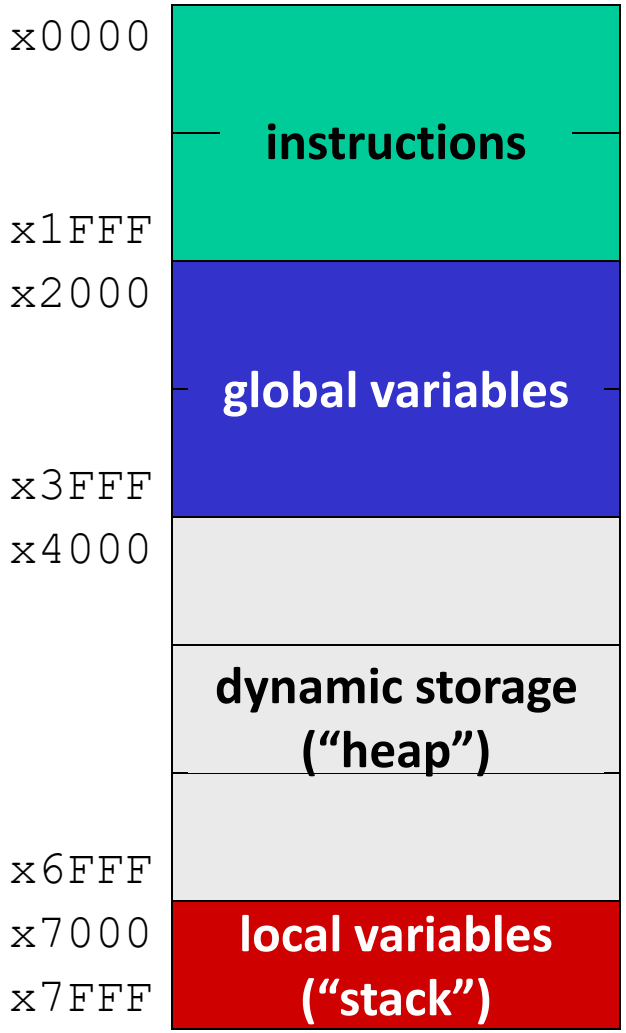
- More info in <arpa/inet.h>
  - Variants also exist for 32 bit and 64 bit conversion

Lecture Outline

- Binary files & Endianness
- **Globals in ASM**
- Maintaining the Stack in ASM

LC4 User Memory Layout for C

- LC4 User memory has CODE and DATA portions. But the DATA is split into three parts for running C code
- Global Variables
- Dynamic Storage (the heap)
- Local Variables (the stack)



Global Variables in C

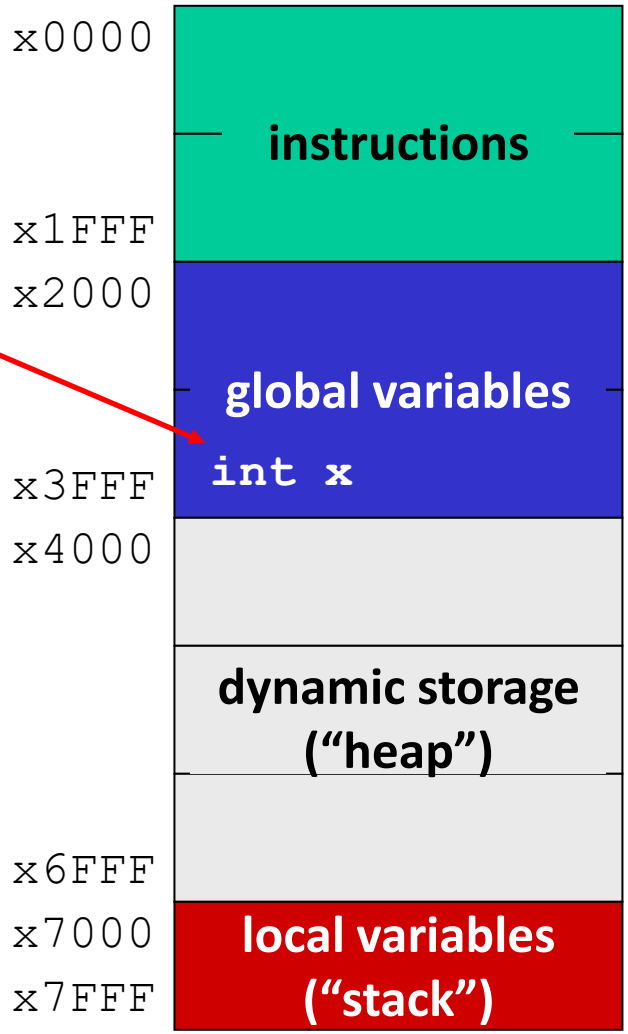
```c
#include <stdio.h>
#include <stdlib.h>

int x = 1;

void incr_globals() {
  x++;
}

int main() {
  printf("x: %d\n", x);   // prints 0
  incr_globals();
  printf("x: %d\n", x);   // prints 1
  return EXIT_SUCCESS;
}
```

Declaring a variable outside of a function makes it "global"

- Global variables exist outside of any function, can be accessed from any function
- Exist throughout the entire lifespan of a program

Global Variables in Memory

- Global variables can be stored at a static (un-changing) address (similar to video memory)
- Reading/writing to that variable just involves going to that static memory location.
- The variable are “allocated as soon as the program is loaded. Program exiting will “de-allocate” t



Global Initialization in ASM

- Global variable would be initialized when program is loaded.
- Can specify initial values of memory with .FILL directive
- Address x220D is arbitrary for this example

```
int global_x = 1;

void incr_global() {
  global_x++;
}
```

```
        .DATA           ; next portion is data
        .ADDR  0x220D   ; start at address 0x220d
global_x                ; label for global
        .FILL x0001     ; directive to initialize this
                        ; memory location to 1 when
                        ; program is loaded
```

Global Read/Write

- Once we have a global, how do we read and/or write to it?
- Global variables are basically constants, we can "lookup" them and then use that address to access it.
- LEA: **L**oad **E**ffective **A**ddress

```
int global_x = 1;

void incr_global() {
  global_x++;
}
```

```
        LEA R4, global_x     ; r4 = &global_x
        LDR R3, R4, #0       ; r3 = *r4
        ADD R3, R3, #1       ; r3 = r3 + 1
        STR R3, R4, #0       ; *r4 = r3
```

Poll Everywhere

pollev.com/tqm

- How many LC4 instructions will be stored in the resulting object file to represent the following LC4 code snippet:

```
        .DATA           ; next portion is data
        .ADDR  0x220D   ; start at address 0x220d
global_x                ; label for global
        .FILL x0001     ; directive to initialize this
                        ; memory location to 1 when
                        ; program is loaded
        .CODE
        ; ...
        ; ...
        LEA R4, global_x    ; r4 = &global_x
        LDR R3, R4, #0      ; r3 = *r4
        ADD R3, R3, #1      ; r3 = r3 + 1
        STR R3, R4, #0      ; *r4 = r3
```

A. 4

B. 5

C. 6

D. 8

E. I'm not sure

Poll Everywhere

pollev.com/tqm

- How many LC4 instructions will be stored in the resulting object file to represent the following LC4 code snippet:

These are all labels/LC4 directives. These just say how to setup memory. Not executed during run-time

```
        .DATA               ; next portion is data
        .ADDR   0x220D      ; start at address 0x220d
global_x                    ; label for global
        .FILL x0001         ; directive to initialize this
                            ; memory location to 1 when
                            ; program is loaded

        .CODE
        ; ...
        ; ...
        LEA R4, global_x    ; r4 = &global_x
        LDR R3, R4, #0      ; r3 = *r4
        ADD R3, R3, #1      ; r3 = r3 + 1
        STR R3, R4, #0      ; *r4 = r3
```

LEA is a pseudo instruction, made of CONST & HICONST

B. 5

Lecture Outline

- Binary files & Endianness
- Globals in ASM
- **Maintaining the Stack in ASM**

Variables in Functions

- Variables declared outside of functions (global variables) exist over the lifetime of the program

- What about variables in functions?
  - Function parameters, local variables, return values etc.
  - Exist only for the lifetime of an instance of execution of a function
  - There may be multiple instances of a function at a time, needing multiple (but separate) sets of variables (e.g. recursion)
  - **Where do these exist in memory?**

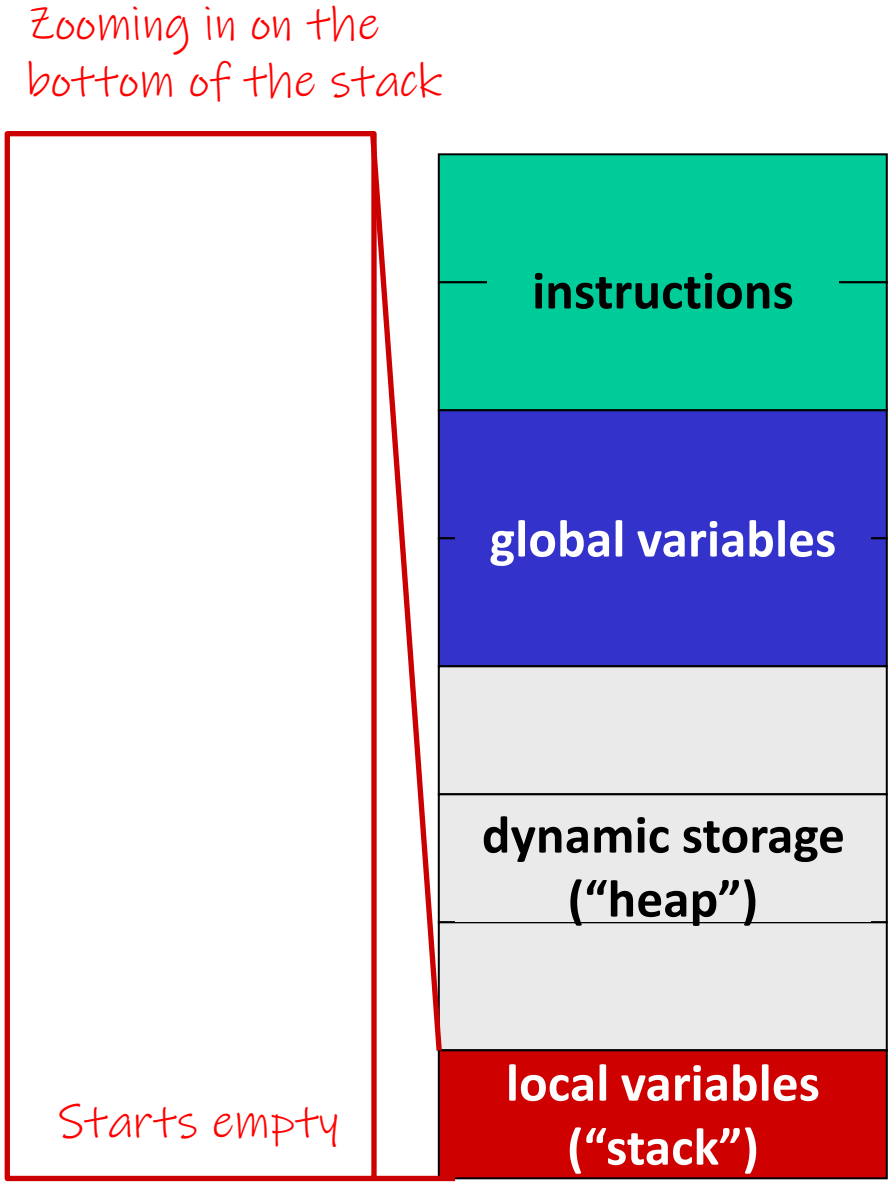
The Stack – short version

- Local variables are stored in a portion of memory called the "Stack" sometimes called the "Call Stack".
  - Whenever a function is invoked, we "push" a "stack frame" for that function onto the top of the stack.
  - The stack frame contains important information about the execution of the function and has space for every local variable
  - When a function exits, its stack frame is "popped" and the local variables are "deallocated"

More details on how the stack works in THIS lecture

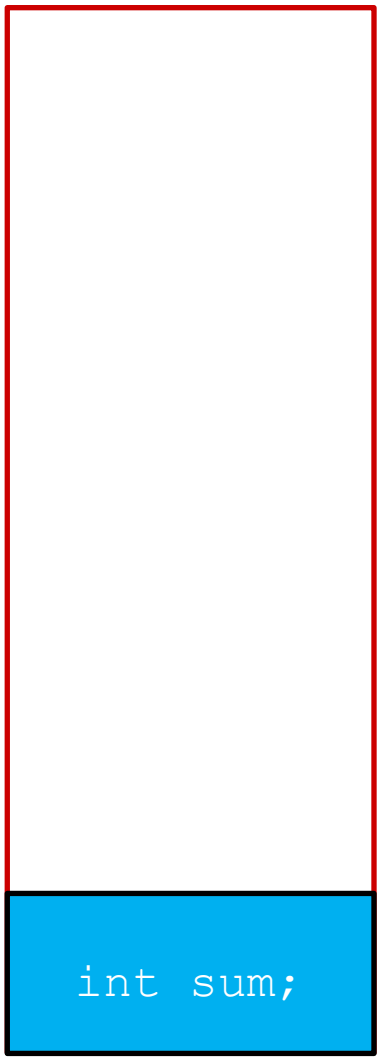
Stack Example:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack Example:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

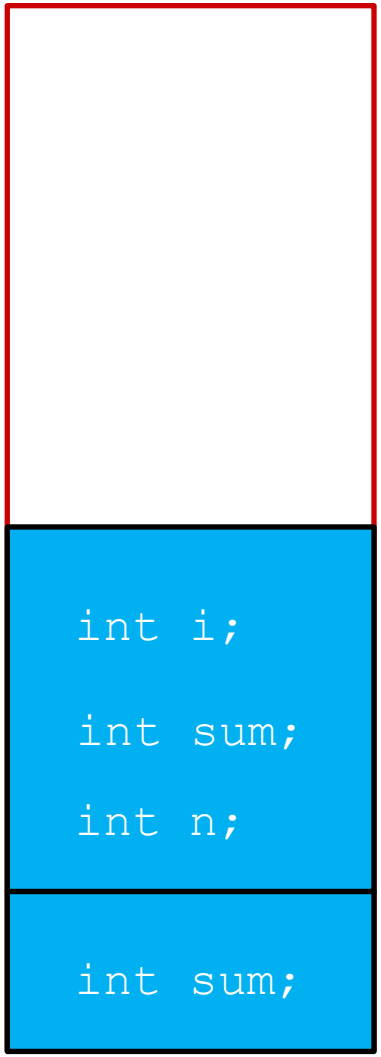


Stack frame for main is created when CPU starts executing it

Stack frame for `main()`

Stack Example:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

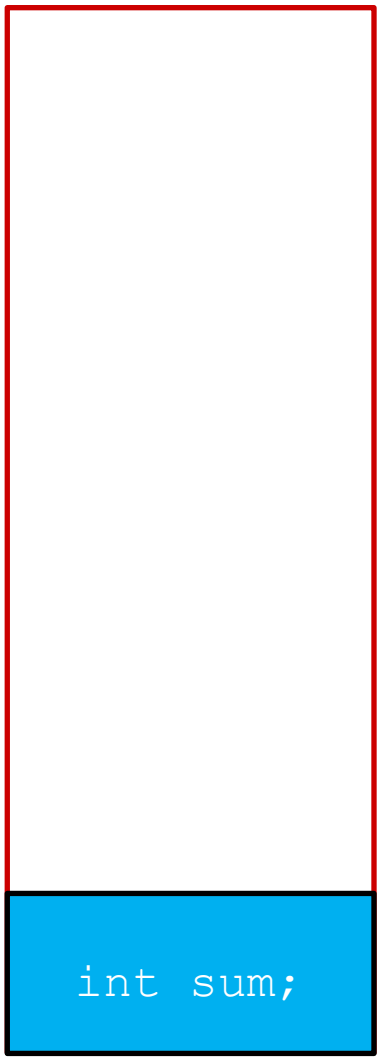


Stack frame for `sum()`

Stack frame for `main()`

Stack Example:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



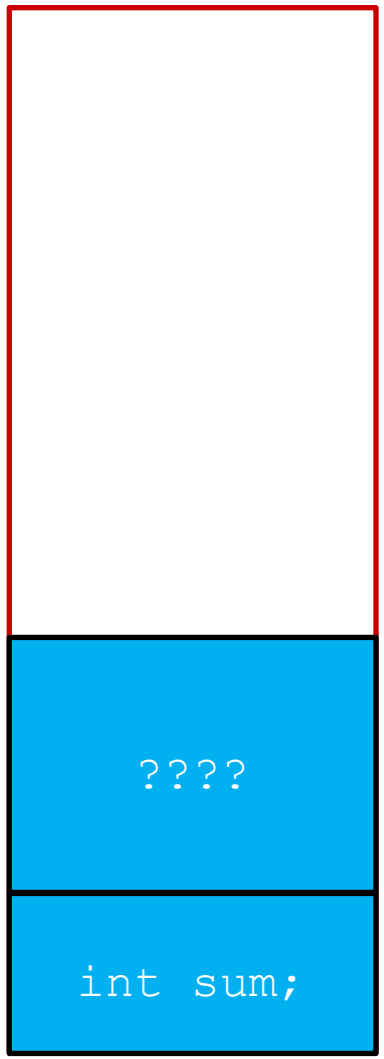
sum()'s stack frame goes away after **sum()** returns.

main()'s stack frame is now top of the stack and we keep executing **main()**

Stack frame for `main()`

Stack Example:

```c
#include <stdio.h>
#include <stdlib.h>

int sum(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int sum = sum(3);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```



Stack frame for `printf()`

Stack frame for `main()`

Creating Functions in LC4

- We have something close to a function call in LC4 already, we can use this as a starting point for LC4 functions:
  - JSR (Jump Subroutine)

Creating a Subroutine:

- Consider the multiply program from 3 lectures ago:
- How do we make this a subroutine?
  - Add a RET pseudo-instruction wherever we are "done" with the subroutine
  - Add the .FALIGN directive before the first label/instruction
    - .FALIGN makes sure the code starts at an address that is a multiple of 16.
    - This is needed since JSR stores a IMM11 that is then shifted to the left by 4
    - (x << 4) == x * 16

```
;; Multiplication program
;; C = A*B
;; R0 = A, R1 = B, R2 = C
        .CODE
        .FALIGN
MULT
        CONST R2, #0
LOOP
        CMPI R1, #0
        BRnz END
        ADD R2, R2, R0

        ADD R1, R1, #-1
        BRnzp LOOP
END
        RET
```

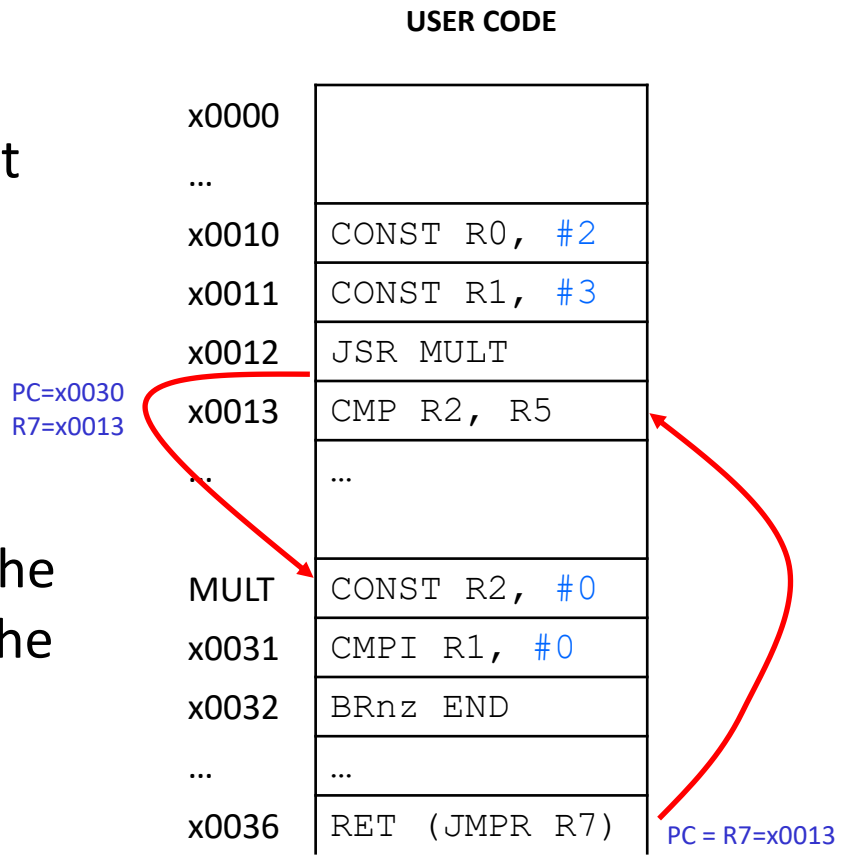
Calling a Subroutine:

- If we wanted to call a subroutine from other LC4 Code

```
.CODE
.ADDR  0x0000
CONST R0, #5  ; Initialize input "parameters"
CONST R1, #6

JSR MULT      ; call the subroutine

; resume execution here after MULT returns
```

Subroutine Walkthrough

- When a JSR is executed:
  - Stores PC + 1 in R7
  - PC jumps to the address of the start of the subroutine (which must be a multiple of 16).
- During Subroutine:
  - R0-R7 are possibly modified
  - R7 should have the same value at the end of the subroutine. It contains the address needed to return to Caller
- After Subroutine is complete:
  - Returns using RET (which is JMPR R7)
  - R7 should contain the return address



Subroutine vs Functions

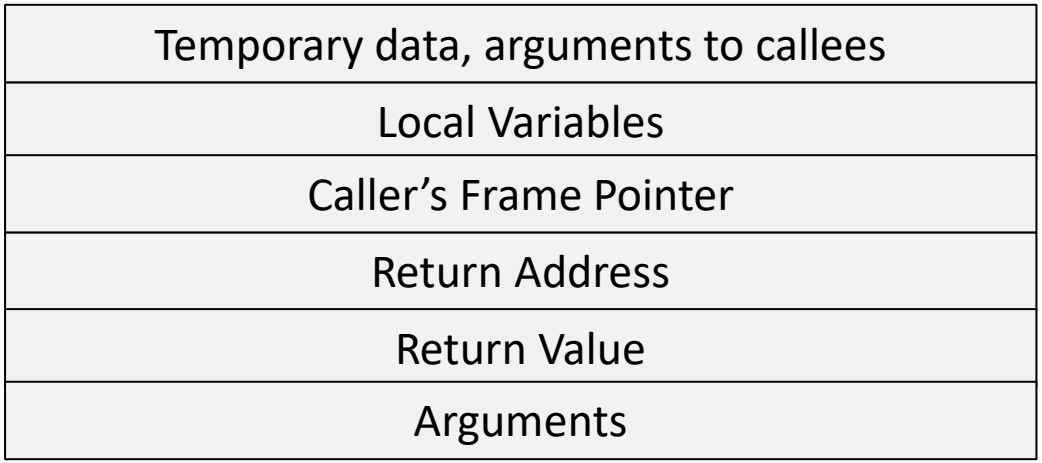
- Calling: Subroutines can be invoked and returned from similar to functions
- "Parameters": Subroutines can designate some registers to contain “inputs” that are set by the caller.
  - What if there are more than 7 parameters??
- "Return Values": Subroutines can designate a register to store their "result" in (if there is one)
- "Variables"
  - The same registers R0-R7 are used inside and outside a subroutine and could be modified
  - What if there are more than 7 variables??
  - Where would we be able to store variables without overwriting other data?

Stack Frames

- We need to be able to allocate space for variables local to a function
  - Local variables, parameters, return values, etc
- Space to hold local variables is the point of the Stack!
- For each function call, we need to maintain a **Stack Frame:**
  - Portion of stack dedicated to that function's local variables
  - Also stores return address so that we can call functions and still be able to return the caller
  - Stores/maintains pointers to keep track of the stack frame
    - Frame Pointer
    - Stack Pointer (top of the current frame)

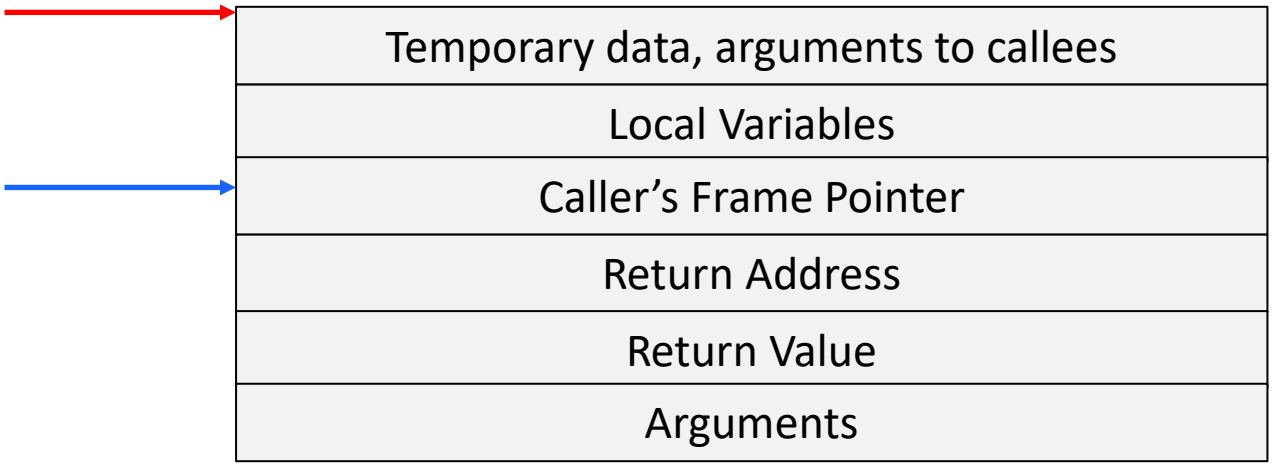
LC4 Stack Frame Layout

- A Frame holds a few things
  - Local variables, return value, arguments
  - Return address (where to return to after this function)
  - A copy of the previous frame pointer (so we can restore it after this function finishes)
  - Temporary Data
  - Arguments to other functions we call from this function (callees)



LC4 Stack Frame Management

- Use two pointers to keep track of the current Stack frame
  - R5: Frame Pointer. Points to the previous frame pointer. Stays constant while executing this function. Useful reference point for getting arguments from caller, local variables setting up, and returning from function
  - R6: Stack Pointer. Points to the top of the Stack (which is the top of the currently executing function's frame). Grows as we add new data to the stack (arguments to callees, temp data, etc).



Example Stack Walkthrough

- Lets manually compile this code into LC4:

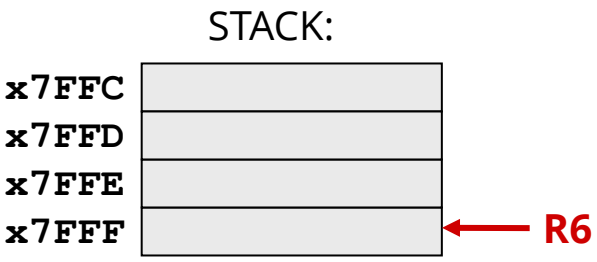
```
int sum(int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

int main() {
  int res;
  res = sum(3);
  return 0;
}
```

Example Stack Walkthrough

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

- Let's start with main():
  - Prologue is where a function begins to construct its frame
  - main() is called using JSR; let's assume R7=x0005 for this example
  - before main() was called, assume R6 = x7FFF (start of the stack), R5=x0000 (no frames)

```
        .CODE
        .FALIGN
main
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
        ;; more later
```

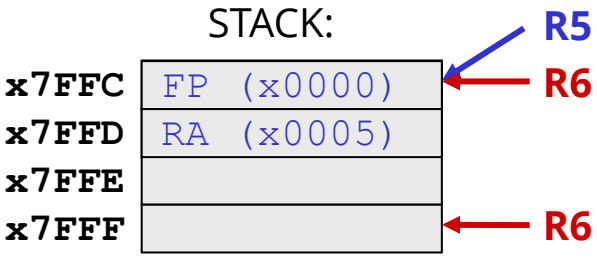


Example Stack Walkthrough

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

- Let's start with main():
  - Prologue is where a function begins to construct its frame
  - main() is called using JSR; let's assume R7=x0005 for this example
  - before main() was called, assume R6 = x7FFF (start of the stack), R5=x0000 (no frames)

```
        .CODE
        .FALIGN
main
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
        ;; more later
```

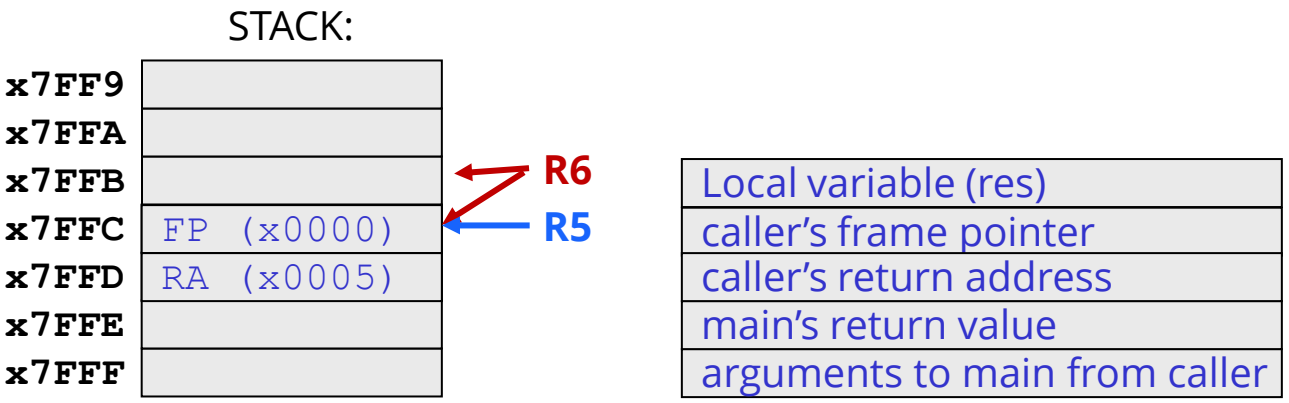


caller's frame pointer
caller's return address
main's return value
arguments to main from caller

Stack Walkthrough Cont.

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

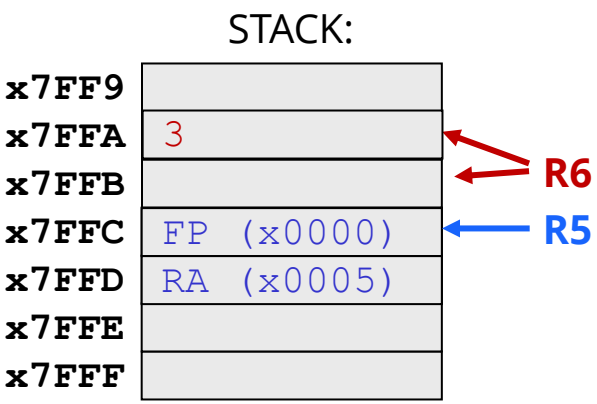
```
        .CODE
        .FALIGN
main
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
        ADD R6, R6, #-1 ;; allocate stack space for local variables
        ;; more later
```



Calling a Function

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .CODE
        .FALIGN
main
        ;; prologue (removed for space)
        ;; function body
        CONST R7, #3
        ADD R6, R6, #-1
        STR R7, R6, #0
        JSR sum  ; Jumps to the Sum function, need to see what it does to the stack
        ;; more later
```



Making it a different color since this is "shared" with the sum function

Argument to sum (3)
Local variable (res)
caller's frame pointer
caller's return address
main's return value
arguments to main from caller

Creating sum's stack frame

```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp

        ADD R6, R6, #-2 ;; allocate stack space for local variables
        CONST R7, #0
        STR R7, R5, #-2
        CONST R7, #0
        STR R7, R5, #-1
```

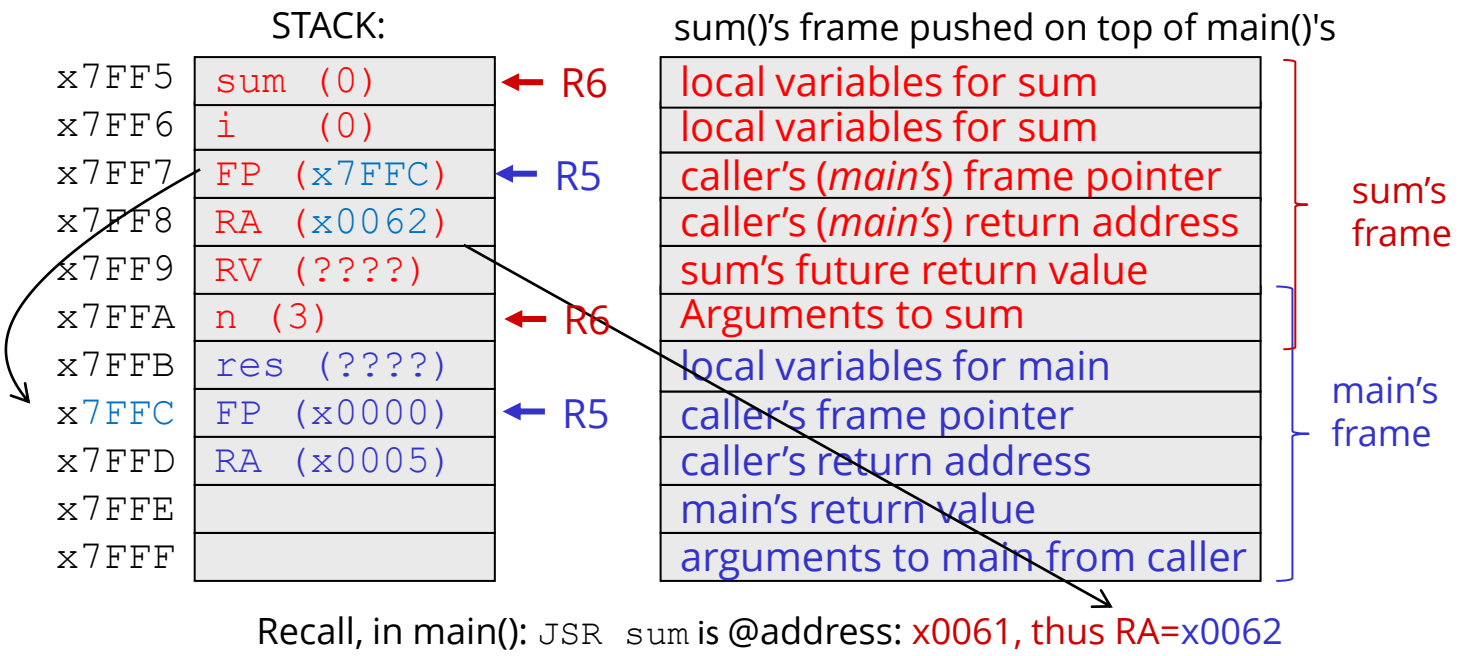
Prologue always the same for lcc

First thing after prologue is to setup local vars

```
int sum(int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
```

Stack at the start of Sum

- After creating the local variables for Sum, our stack looks like:



Poll Everywhere

pollev.com/tqm

- If I wanted to load argument N into R7, which instruction would most reliably do that?

A. LDR R7, R5, #(n + 3)

B. LDR R7, R5, #-(n +3)

C. LDR R7, R6, #(n + 3)

D. LDR R7, R6, #-(n + 3)

E. I'm not sure

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

STACK:

x7FF5	sum (6)	← R6
x7FF6	i (3)	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (????)	
x7FFA	n (3)	
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 6

STACK:

x7FF5	sum (6)	← R6
x7FF6	i (3)	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (????)	
x7FFA	n (3)	
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

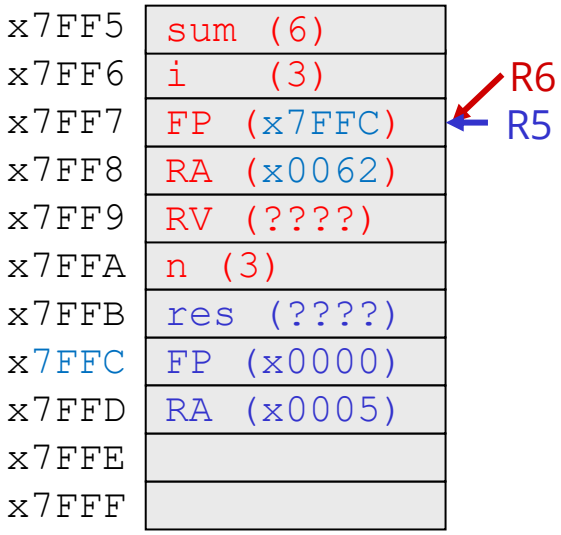
Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 6

STACK:




```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

Local variables are not really “deallocated”, we just treat anything past the stack pointer (R6) as garbage data. Accessing that data from the user perspective is undefined behaviour

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 6

STACK:

x7FF5	sum (6)
x7FF6	i (3)
x7FF7	FP (x7FFC) ← R5
x7FF8	RA (x0062)
x7FF9	RV (????)
x7FFA	n (3) ← R6
x7FFB	res (????)
x7FFC	FP (x0000)
x7FFD	RA (x0005)
x7FFE	
x7FFF	

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 6

STACK:

x7FF5	sum (6)	
x7FF6	i (3)	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (6)	
x7FFA	n (3)	← R6
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 6

STACK:

x7FF5	sum (6)
x7FF6	i (3)
x7FF7	FP (x7FFC)
x7FF8	RA (x0062)
x7FF9	RV (6)
x7FFA	n (3) ← R6
x7FFB	res (????)
x7FFC	FP (x0000) ← R5
x7FFD	RA (x0005)
x7FFE	
x7FFF	

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

Function Epilogue

Red arrow is next instruction to execute

- Once a function is done, it needs to store the return value in R7 and execute the epilogue:

R7 | 0x0062

STACK:

x7FF5	sum (6)
x7FF6	i (3)
x7FF7	FP (x7FFC)
x7FF8	RA (x0062)
x7FF9	RV (6)
x7FFA	n (3) ← R6
x7FFB	res (????)
x7FFC	FP (x0000) ← R5
x7FFD	RA (x0005)
x7FFE	
x7FFF	

```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore RA
        RET
```

Ret will return us to main()'s code to keep executing

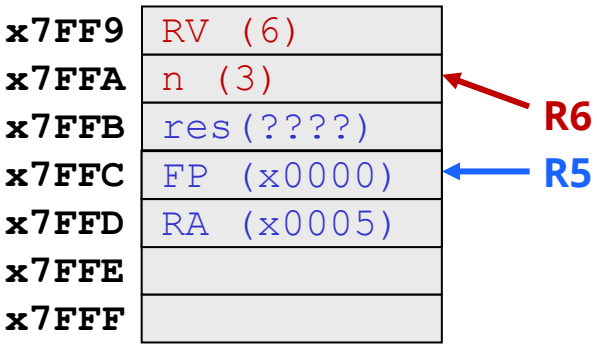
Returning to the caller

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .CODE
        .FALIGN
main
        ;; prologue (removed for space)
        ;; function body
        JSR sum
        LDR R7, R6, #-1 ; grab return value
        ADD R6, R6, #1  ; free space for args
        STR R7, R5, #-1 ; store return value in res local var
        ;; R7 = 0 and epilouge (removed for space)
```

Return value placed just above the stack.
Sometimes we call functions without using the return value

STACK:



Return value from sum (3)
Argument to sum (3)
Local variable (res)
caller's frame pointer
caller's return address
main's return value
arguments to main from caller

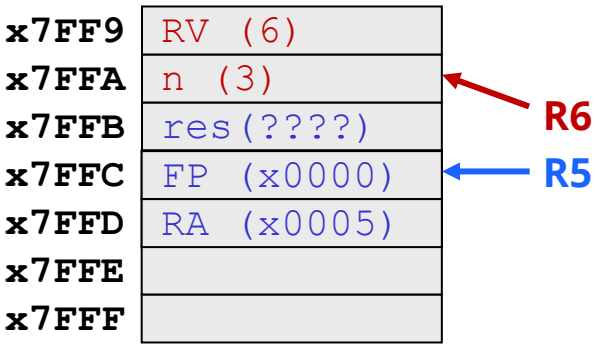
Returning to the caller

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .CODE
        .FALIGN
main
        ;; prologue (removed for space)
        ;; function body
        JSR sum
        LDR R7, R6, #-1 ; grab return value
        ADD R6, R6, #1  ; free space for args
        STR R7, R5, #-1 ; store return value in res local var
        ;; R7 = 0 and epilouge (removed for space)
```

R7 | 6

STACK:



Return value from sum (3)
Argument to sum (3)
Local variable (res)
caller's frame pointer
caller's return address
main's return value
arguments to main from caller

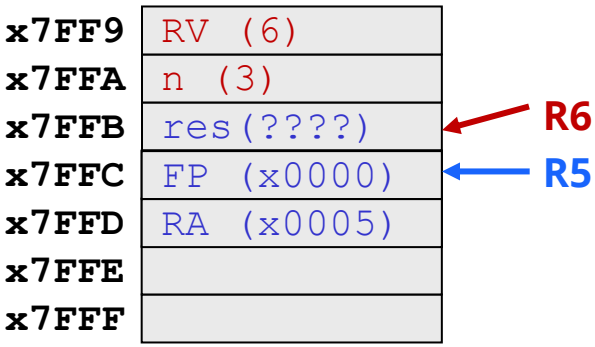
Returning to the caller

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .CODE
        .FALIGN
main
        ;; prologue (removed for space)
        ;; function body
        JSR sum
        LDR R7, R6, #-1 ; grab return value
        ADD R6, R6, #1  ; free space for args
        STR R7, R5, #-1 ; store return value in res local var
        ;; epilouge (removed for space)
```

R7 | 6

STACK:



Local variable (res)
caller's frame pointer
caller's return address
main's return value
arguments to main from caller

Returning to the caller

```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .CODE
        .FALIGN
main
        ;; prologue (removed for space)
        ;; function body
        JSR sum
        LDR R7, R6, #-1 ; grab return value
        ADD R6, R6, #1  ; free space for args
        STR R7, R5, #-1 ; store return value in res local var
        ;; R7 = 0 and epilouge (removed for space)
```

R7 | 6

main()'s epilogue will be the same as sum()'s. The only difference is how we store the return value into R7 before the epilogue.

STACK:

x7FF9	RV (6)
x7FFA	n (3)
x7FFB	res(6)
x7FFC	FP (x0000)
x7FFD	RA (x0005)
x7FFE	
x7FFF	

R6 → x7FFB

R5 → x7FFC

Local variable (res)
caller's frame pointer
caller's return address
main's return value
arguments to main from caller

Next Time:

- More practice with this
- More C -> ASM
- Implications of the stack system