C to ASM pt. 2 (The Stack Cont.)

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
- Check-in 9 “The make-up” check-in to be released Thursday, due Monday @ 4:59 pm before lecture.

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

Mid Semester Feedback

- I am reading these, but there are a lot of responses to read. Expect more responses next week.
  - Lots of discussion on Pace and how/quick/slow explanations are
    - Most people seem to like the pace?
    - Second Place: Go Slower, Third Place: Go Faster
  - Some discussion on what is done in lecture
    - Some people want more interaction (I do too)
    - Some want more demos/higher level perspectives etc.
    - Some people feel lecture is too dense

Lecture Outline

- **Stack Continued**
  - **Prologue**
  - **Epilogue**
  - **Register Spilling**
- Implications of the stack system

Stack is really important for Hw10 & Hw11

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

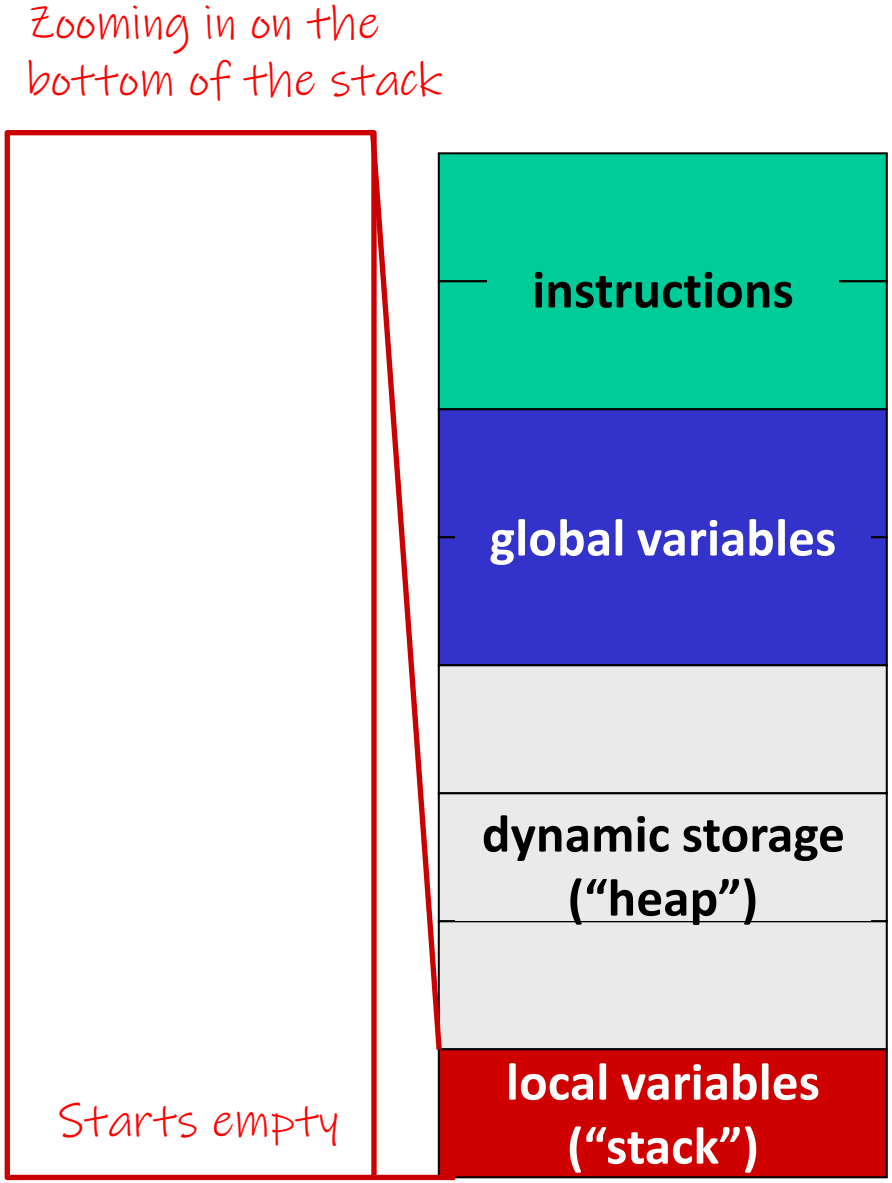
- Local variables are stored in a portion of memory called the "Stack" sometimes called the "Call Stack".
  - Whenever a function is invoked, we "push" a "stack frame" for that function onto the top of the stack.
  - The stack frame contains important information about the execution of the function and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

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;
}
```

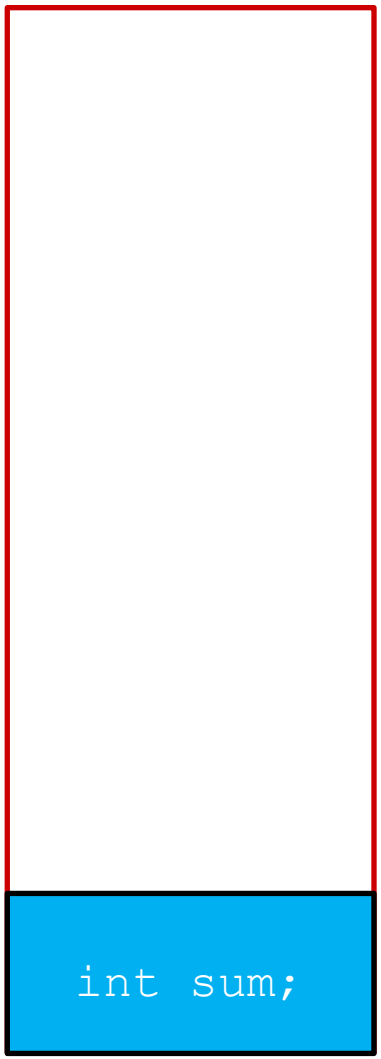


Stack Example:

```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

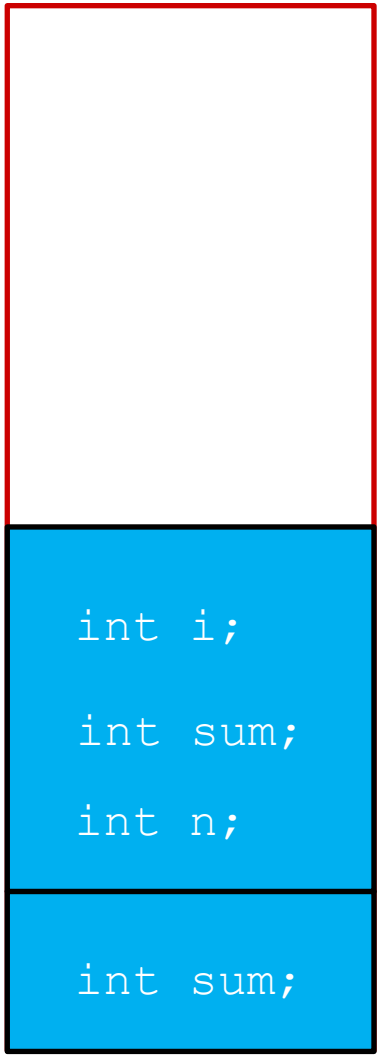
Stack frame for `main()`

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()`

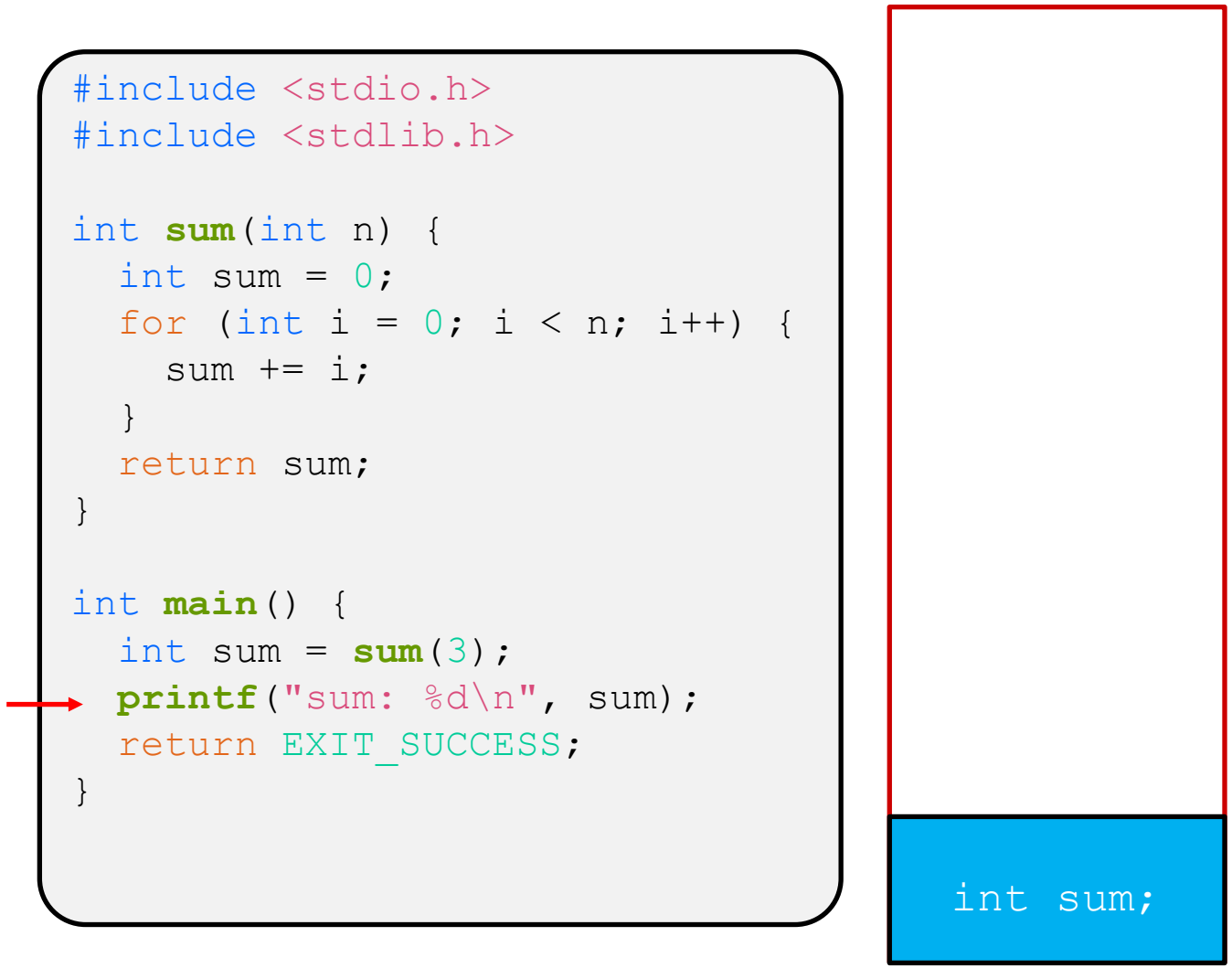
Stack frame for `main()`

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()**

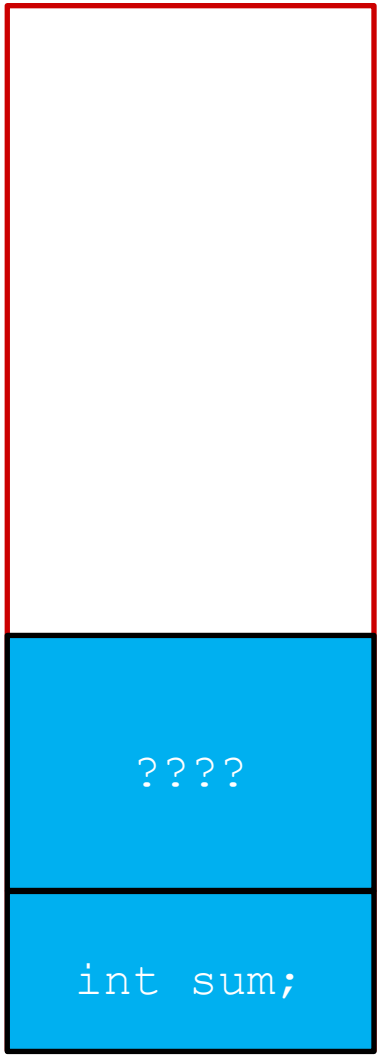
Stack frame for `main()`

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)

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

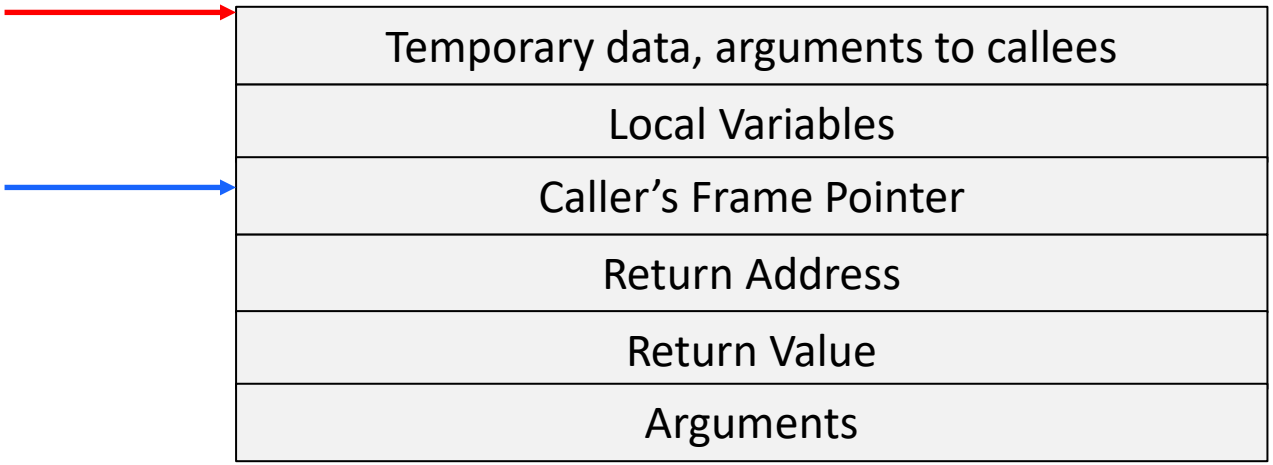
Stack frame in LC4 always follows this layout

- 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)

Temporary data, arguments to callees
Local Variables
Caller's Frame Pointer
Return Address
Return Value
Arguments

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;
}

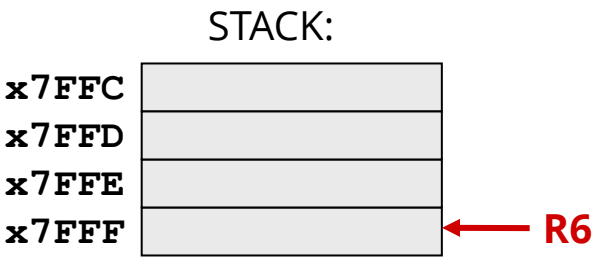
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

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
```

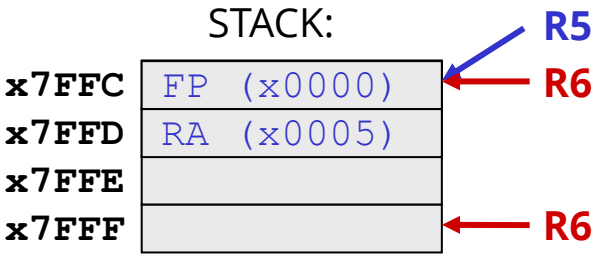


Example Stack Walkthrough

```
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
```

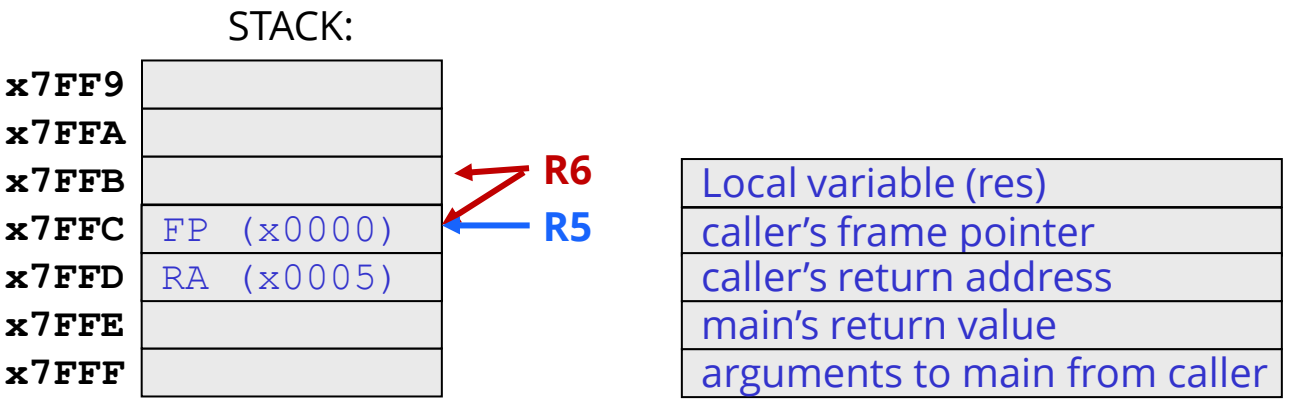


caller's frame pointer
caller's return address
main's return value
arguments to main from caller

Stack Walkthrough Cont.

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

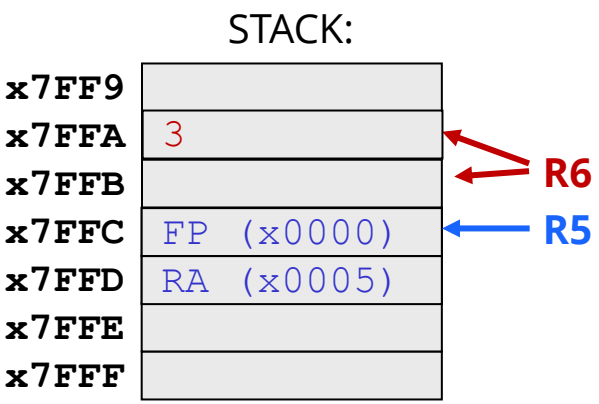
```
        .CODE
        .FALIGN
main
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
        ADD R6, R6, #-1 ;; allocate stack space for local variables
        ;; more later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



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

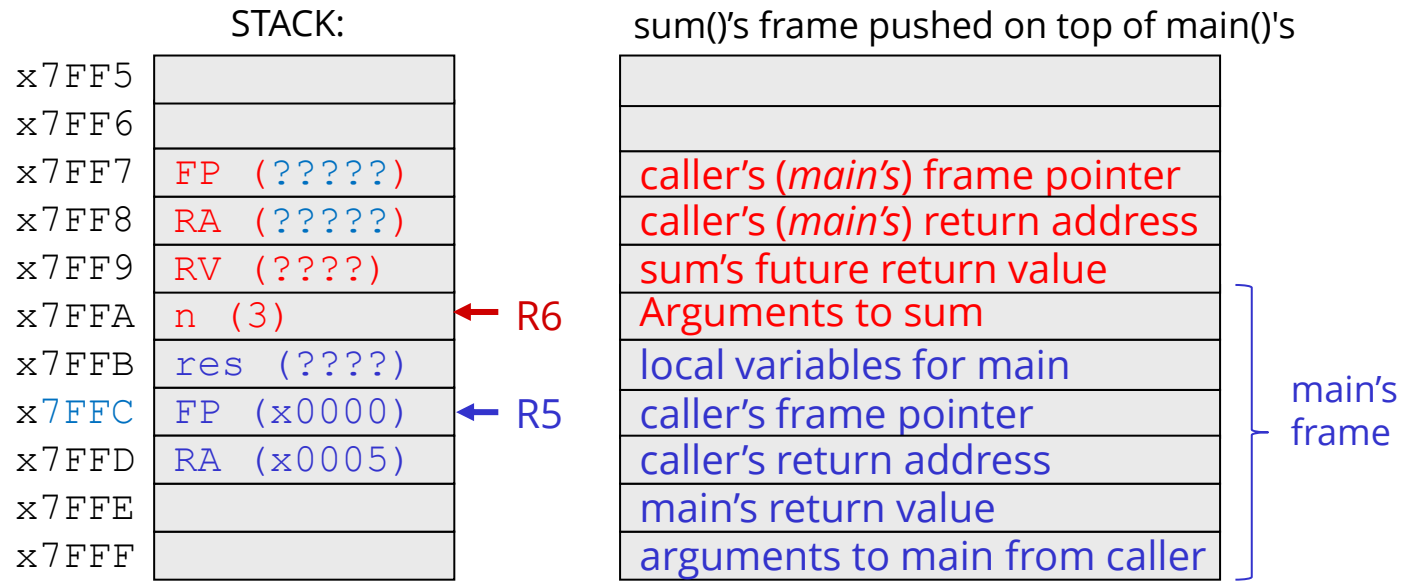
First thing after prologue is to setup local vars

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

Sum's Prologue

Red arrow is next instruction to execute

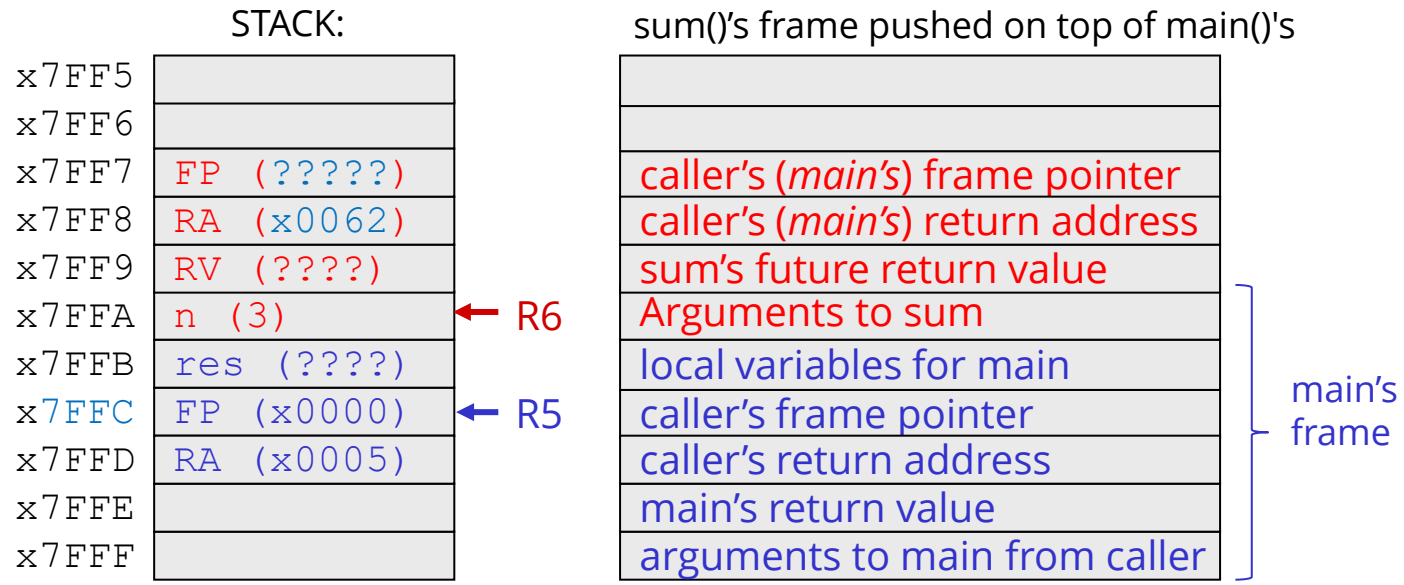
```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
```



Sum's Prologue

Red arrow is next instruction to execute

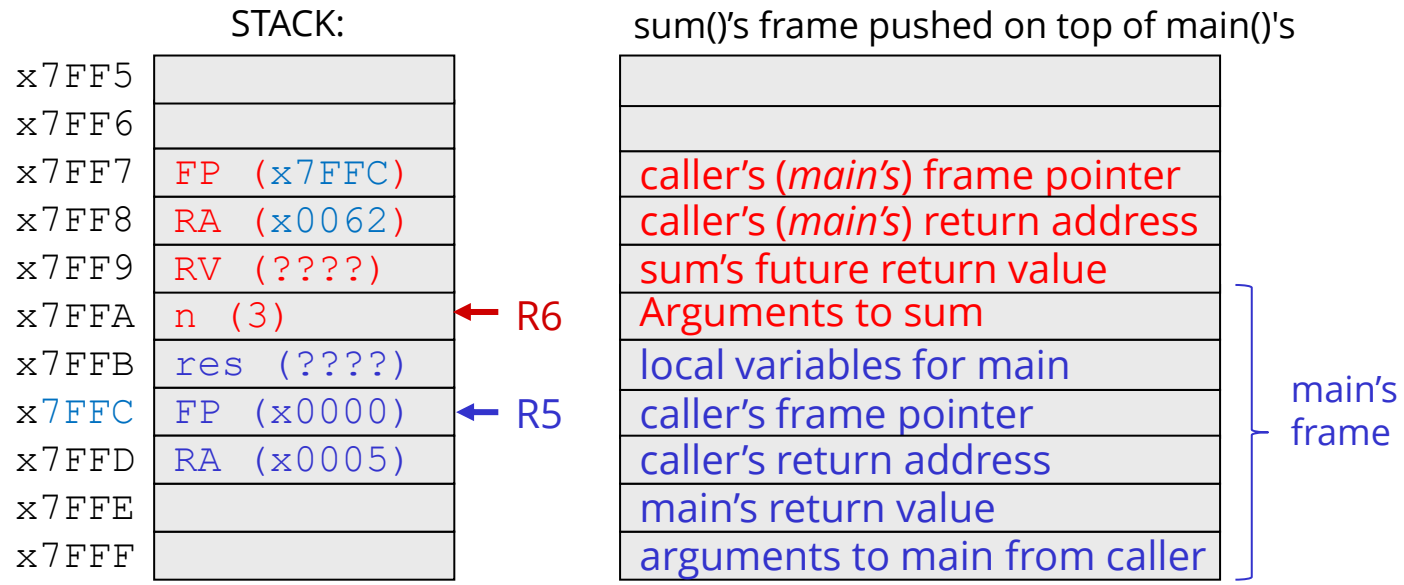
```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
```



Sum's Prologue

Red arrow is next instruction to execute

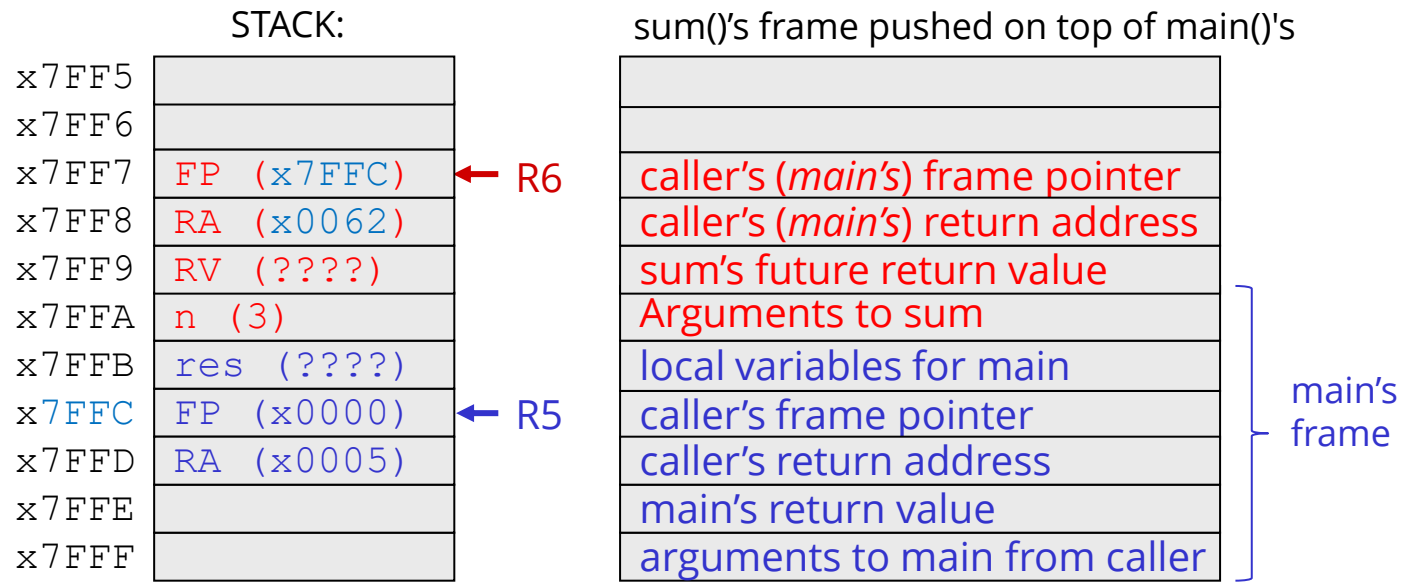
```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
```



Sum's Prologue

Red arrow is next instruction to execute

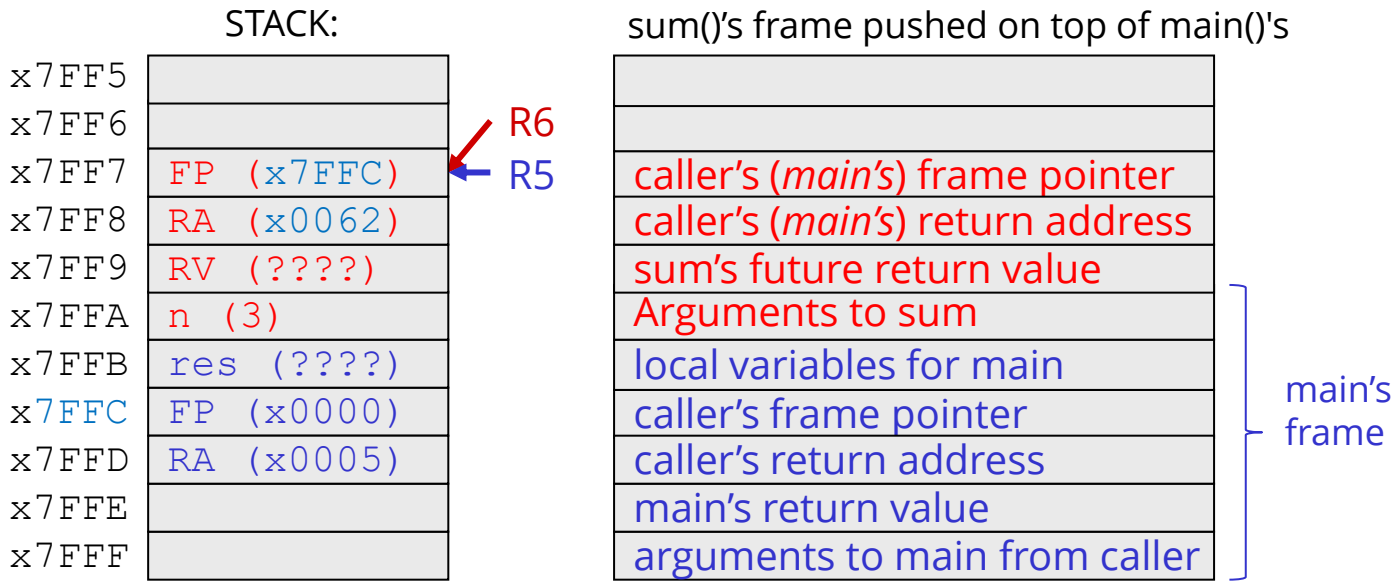
```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
```



Sum's Prologue

Red arrow is next instruction to execute

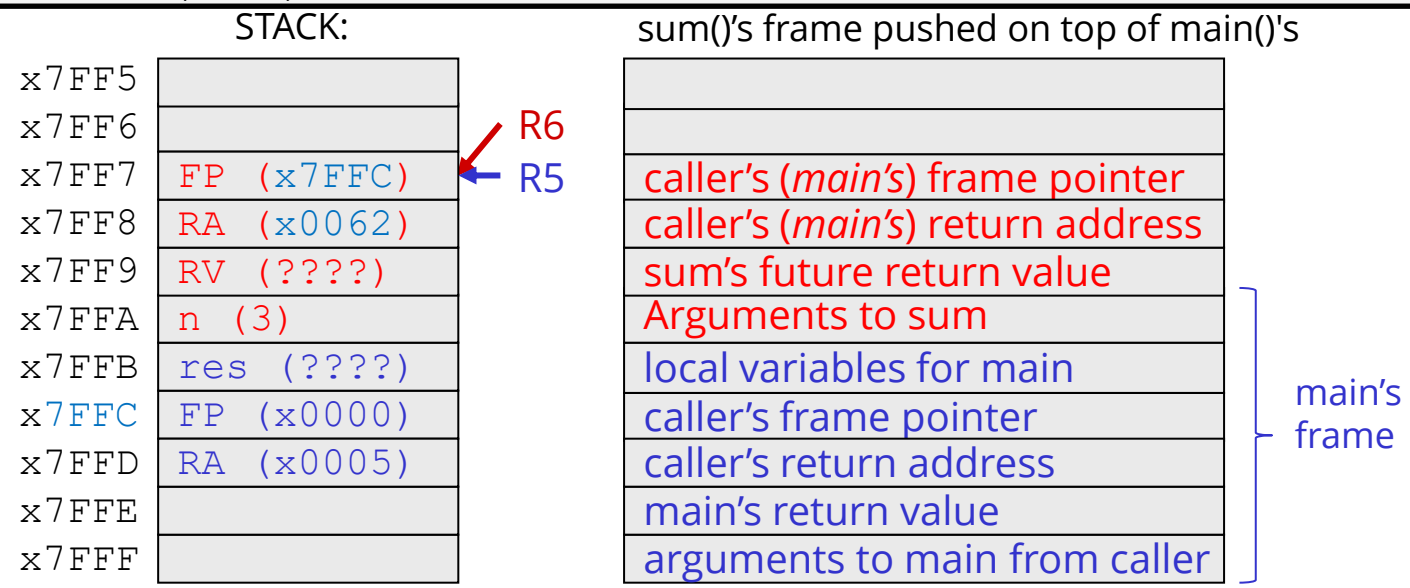
```
        .CODE
        .FALIGN
sum
        ;; prologue
        STR R7, R6, #-2 ;; save return address
        STR R5, R6, #-3 ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0  ;; update fp
```



Sum's Local Variables

Red arrow is next instruction to execute

```
        .CODE
        .FALIGN
sum
        ;; prologue (removed for space)
        ADD R6, R6, #-2 ;; allocate stack space for local variables
        CONST R7, #0
        STR R7, R5, #-2
        CONST R7, #0
        STR R7, R5, #-1
```



Sum's Local Variables

Red arrow is next instruction to execute

```
        .CODE
        .FALIGN
sum
        ;; prologue (removed for space)
        ADD R6, R6, #-2 ;; allocate stack space for local variables
        CONST R7, #0
        STR R7, R5, #-2
        CONST R7, #0
        STR R7, R5, #-1
```

STACK:

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

sum()'s frame pushed on top of main()'s

local variables for sum	
local variables for sum	
caller's (*main's*) frame pointer	
caller's (*main's*) return address	
sum's future return value	
Arguments to sum	main's frame
local variables for main	
caller's frame pointer	
caller's return address	
main's return value	
arguments to main from caller	

Sum's Local Variables

Red arrow is next instruction to execute

```
        .CODE
        .FALIGN
sum
        ;; prologue (removed for space)
        ADD R6, R6, #-2 ;; allocate stack space for local variables
        CONST R7, #0
        STR R7, R5, #-2
        CONST R7, #0
        STR R7, R5, #-1
```

STACK:

x7FF5	sum (0)	← R6
x7FF6	i (????)	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (????)	
x7FFA	n (3)	
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

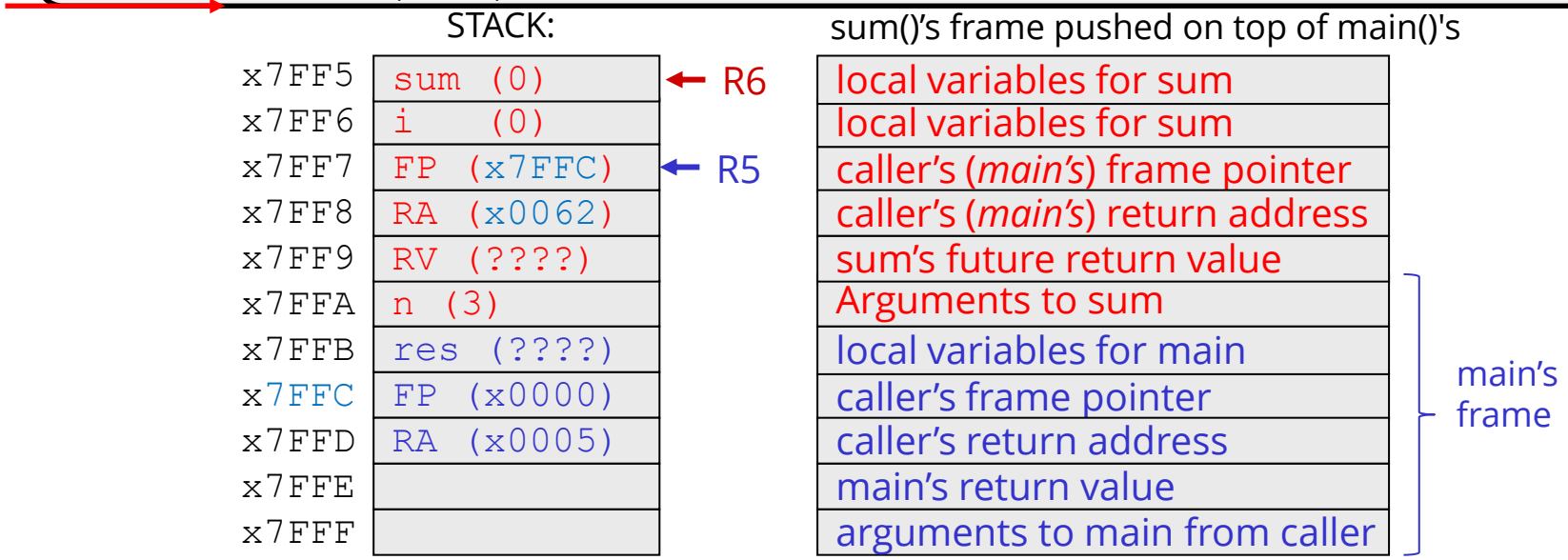
sum()'s frame pushed on top of main()'s

local variables for sum	
local variables for sum	
caller's (*main's*) frame pointer	
caller's (*main's*) return address	
sum's future return value	
Arguments to sum	main's frame
local variables for main	
caller's frame pointer	
caller's return address	
main's return value	
arguments to main from caller	

Sum's Local Variables

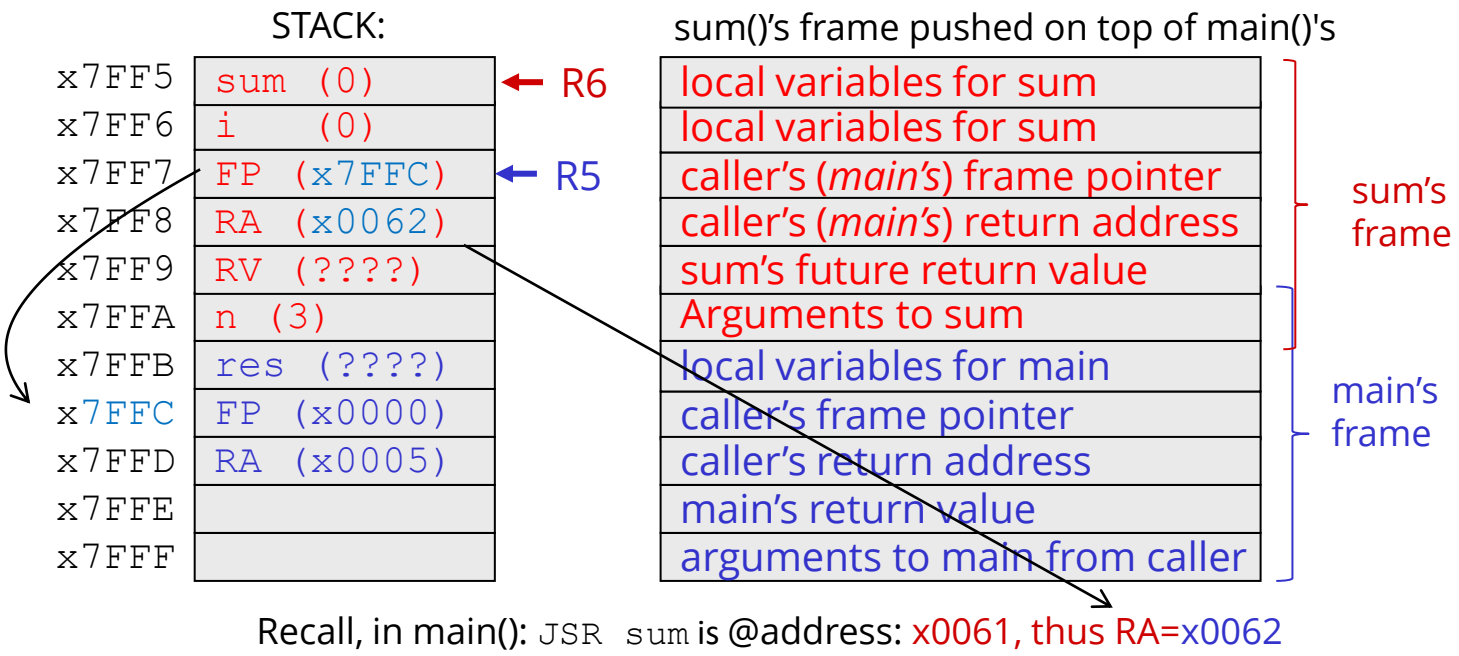
Red arrow is next instruction to execute

```
        .CODE
        .FALIGN
sum
        ;; prologue (removed for space)
        ADD R6, R6, #-2 ;; allocate stack space for local variables
        CONST R7, #0
        STR R7, R5, #-2
        CONST R7, #0
        STR R7, R5, #-1
```



Stack at the start of Sum

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



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		

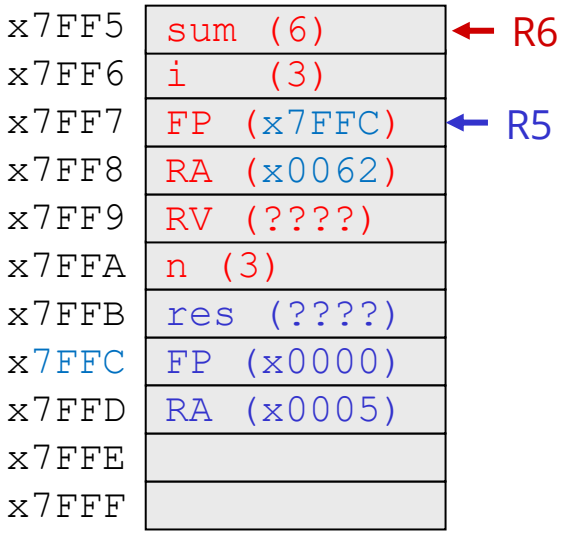
Function Epilogue

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:




```
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
```

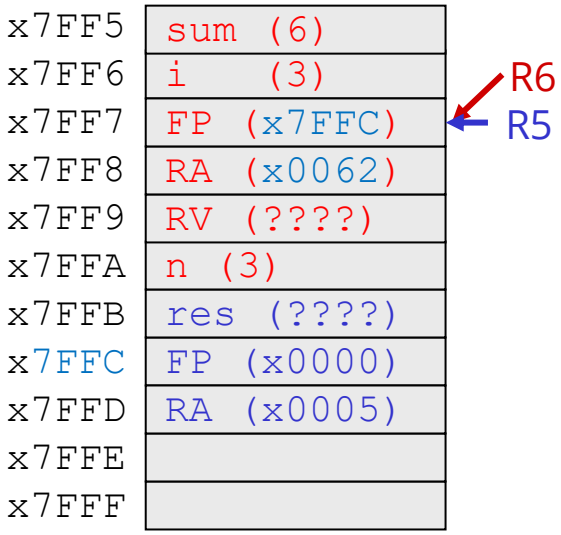
Function Epilogue

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

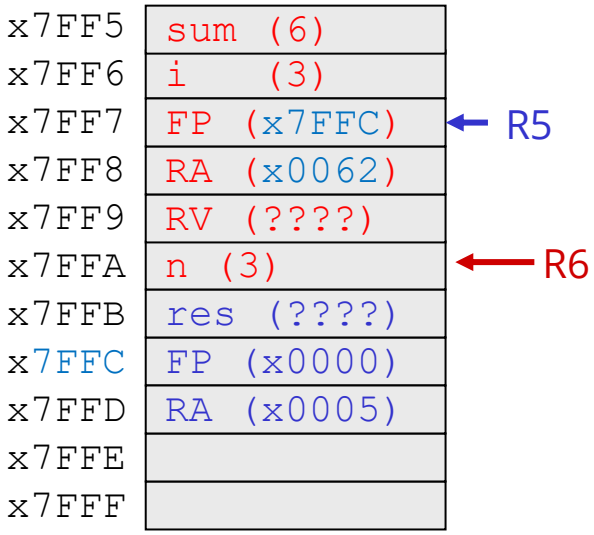
Function Epilogue

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:

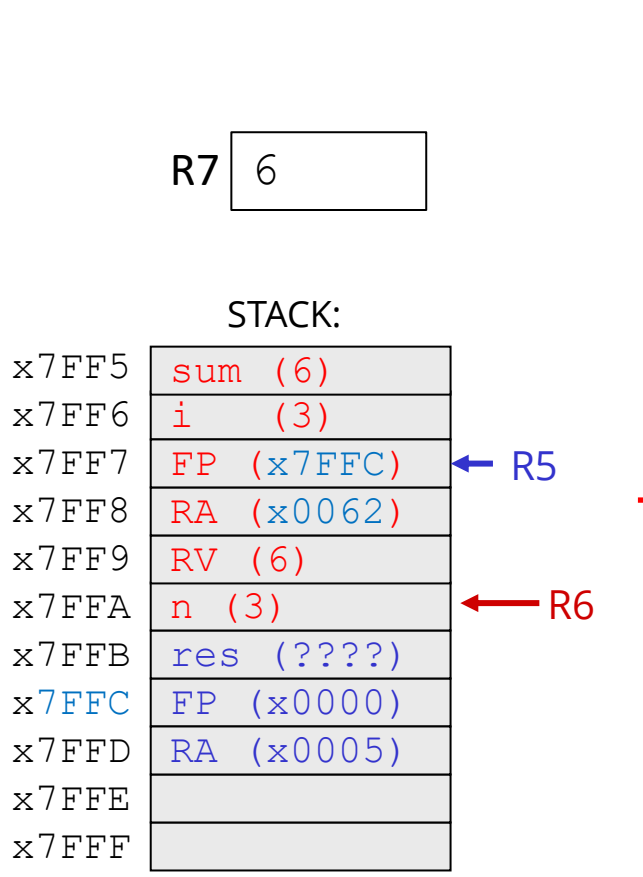



```
sum
        ;; function body (skipped for time)
        ;; epilogue
        LDR R7, R5, #-2 ;; R7 = sum
L1_sum
        ;; epilogue
        ADD R6, R5, #0  ;; pop locals
        ADD R6, R6, #3
        STR R7, R6, #-1 ;; store RV
        LDR R5, R6, #-3 ;; restore FP
        LDR R7, R6, #-2 ;; restore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




```
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
```

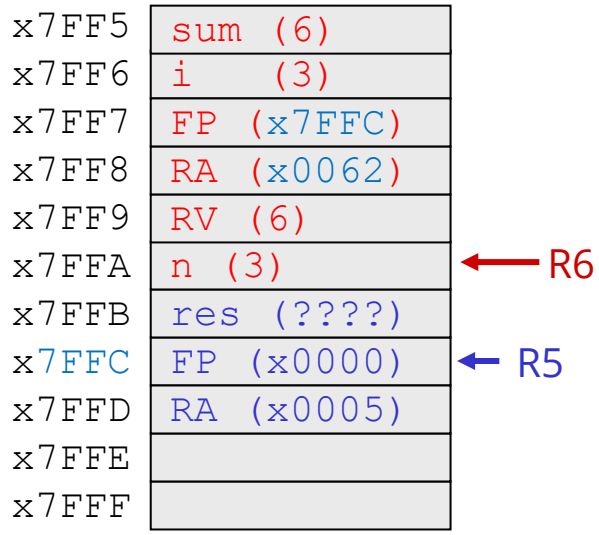
Function Epilogue

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:




```
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:

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

R6 → x7FFA
R5 → x7FFC

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:

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

R6 → x7FFA

R5 → x7FFC

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:

x7FF9	RV (6)
x7FFA	n (3)
x7FFB	res(????) ← R6
x7FFC	FP (x0000) ← R5
x7FFD	RA (x0005)
x7FFE	
x7FFF	

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

Returning to the caller

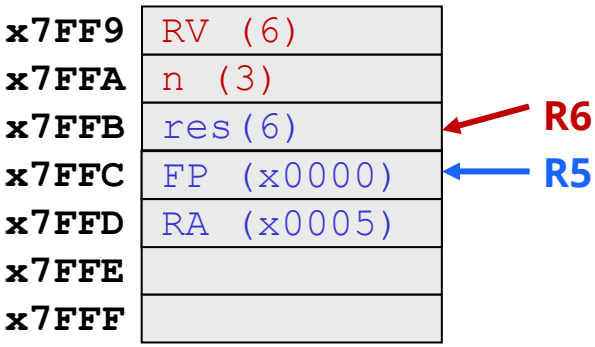
```
int main() {
  int res;
  res = sum(3);
  return 0;
}
```

```
        .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:



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

Using Variables

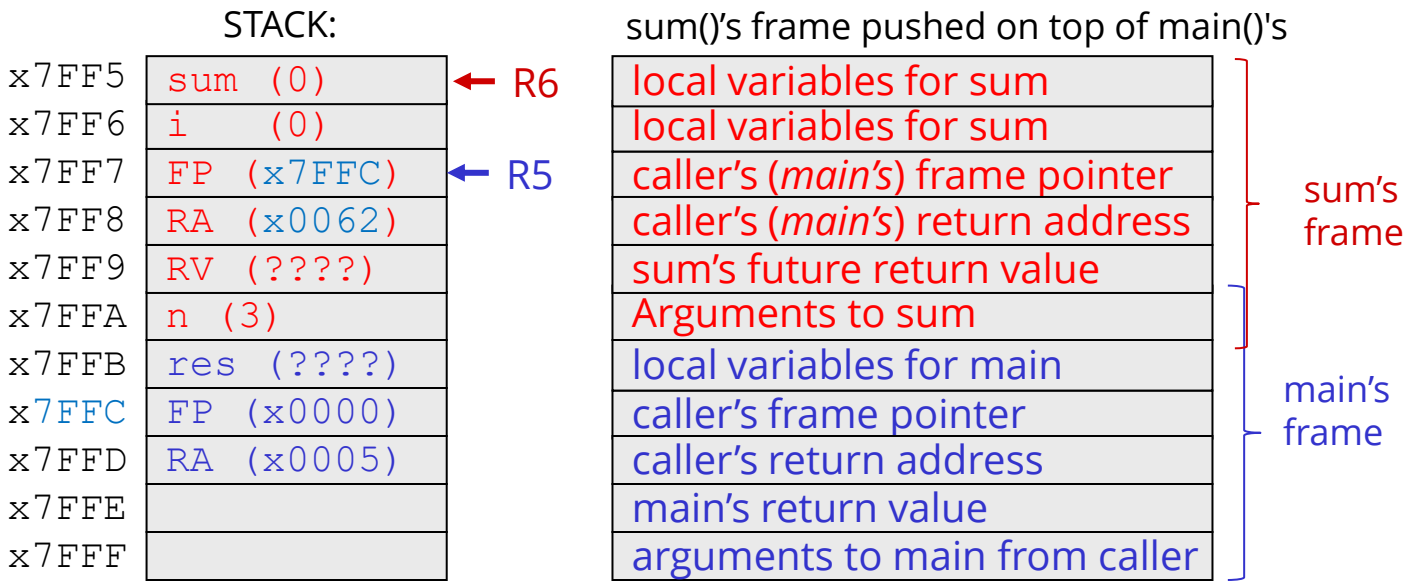
- With C, you can declare variables with names and use those names to use the variable.
  - In LC4 assembly, we don't have this leisure

- In LC4 assembly, the programmer (compiler in our case) needs to be careful about what registers and what portions of memory contain values that are being used.

Example: Variables in Sum()

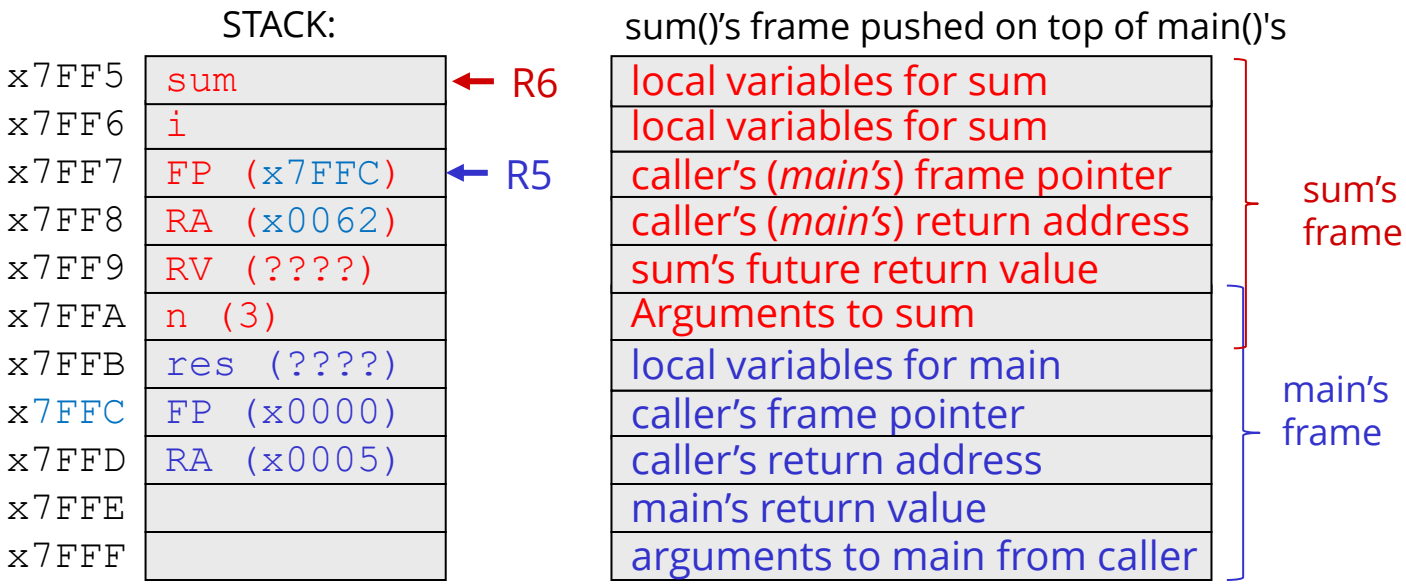
- The local variables in the sum() example are stored on the stack.
- If we want to access the variable 'i' at address R5 - 1

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



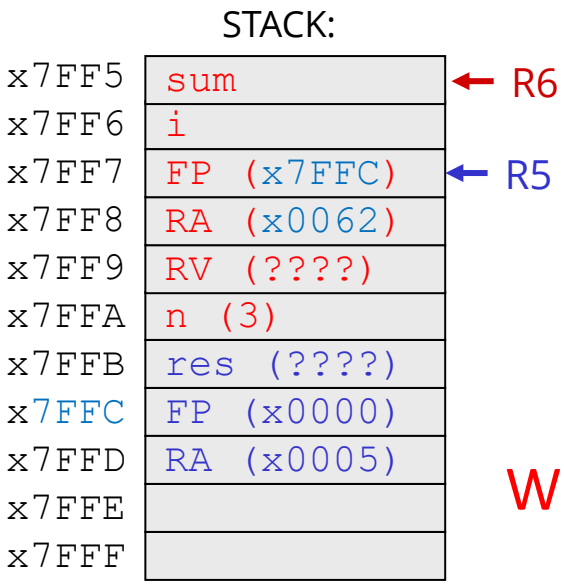
Expressions Involving Variables

- In our sum example, how would we write:
  - sum += i;
  - How many registers needed to evaluate this statement?
  - How many instructions?



Expressions Involving Variables

- In our sum example, how would we write:
  - **`sum += i;`**
  - how many registers needed to evaluate this statement? 2 (not counting r5 and r6)
  - How many instructions? 4




```
LDR R0, R5, #-1
LDR R1, R5, #-2
ADD R1, R1, R0
STR R1 R5, #-2
```

Why did this code use R5? Why didn't it use R6?

Temporary Values & Register "Spilling

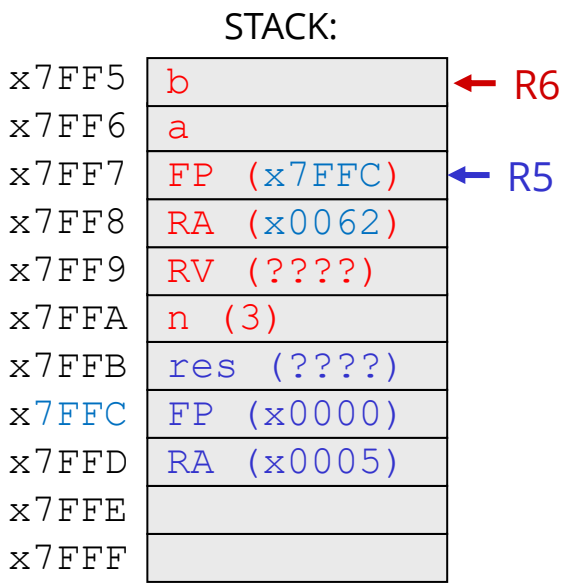
- What if instead we wanted to calculate:
  - ```
    a = (a + b) * (a - b);
    // assume 'a' and 'b' are local variables
    // a is at R5 - 1, B is at R5 - 2
    ```
  - How many register would this need at minimum? 3
  - What if we only had 2 "free" registers?

```
LDR R0, R5, #-1
LDR R1, R5, #-2
ADD R2, R1, R0
SUB R0, R0, R1
MUL R0, R0, R2
STR R0, R5, #-1
```

- What if an operation required more registers than we had available?
  - Answer: Can "free" registers mid-evaluation by storing temporary data on the stack
```

Spilling registers

- We want to calculate:
  - **`a = (a + b) * (a - b);`**
    `// assume 'a' and 'b' are local variables`
    `// a is at R5 - 1, B is at R5 - 2`
  - With only using two registers (R0, R1) by spilling to the stack?

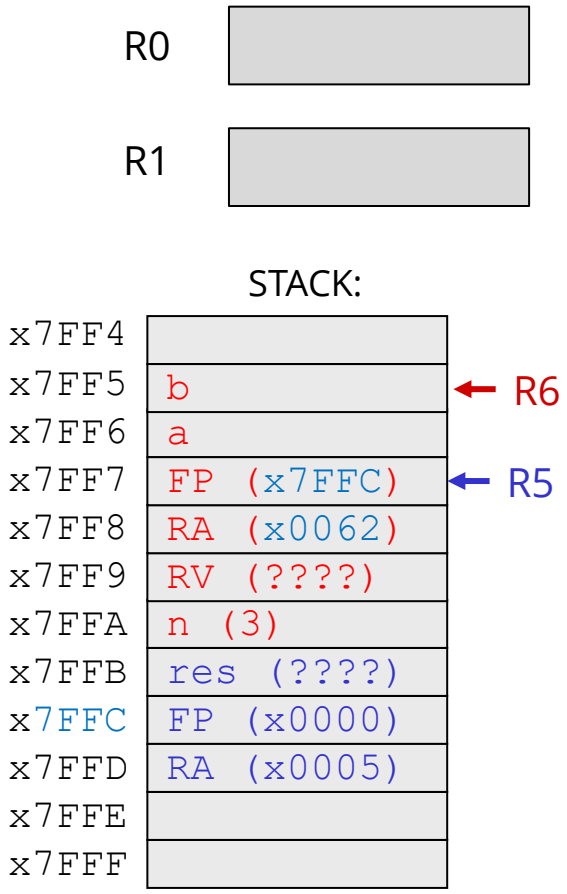



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

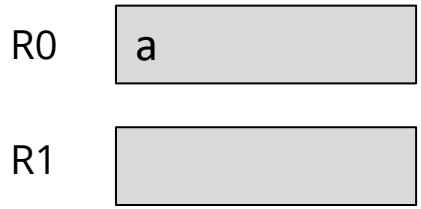


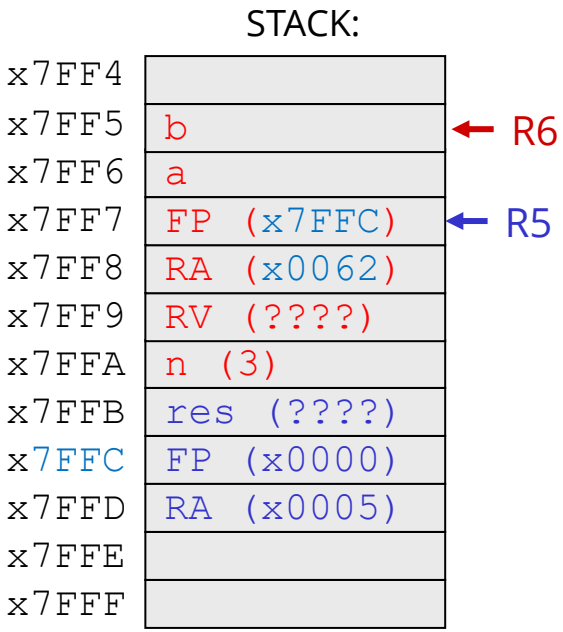

```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`



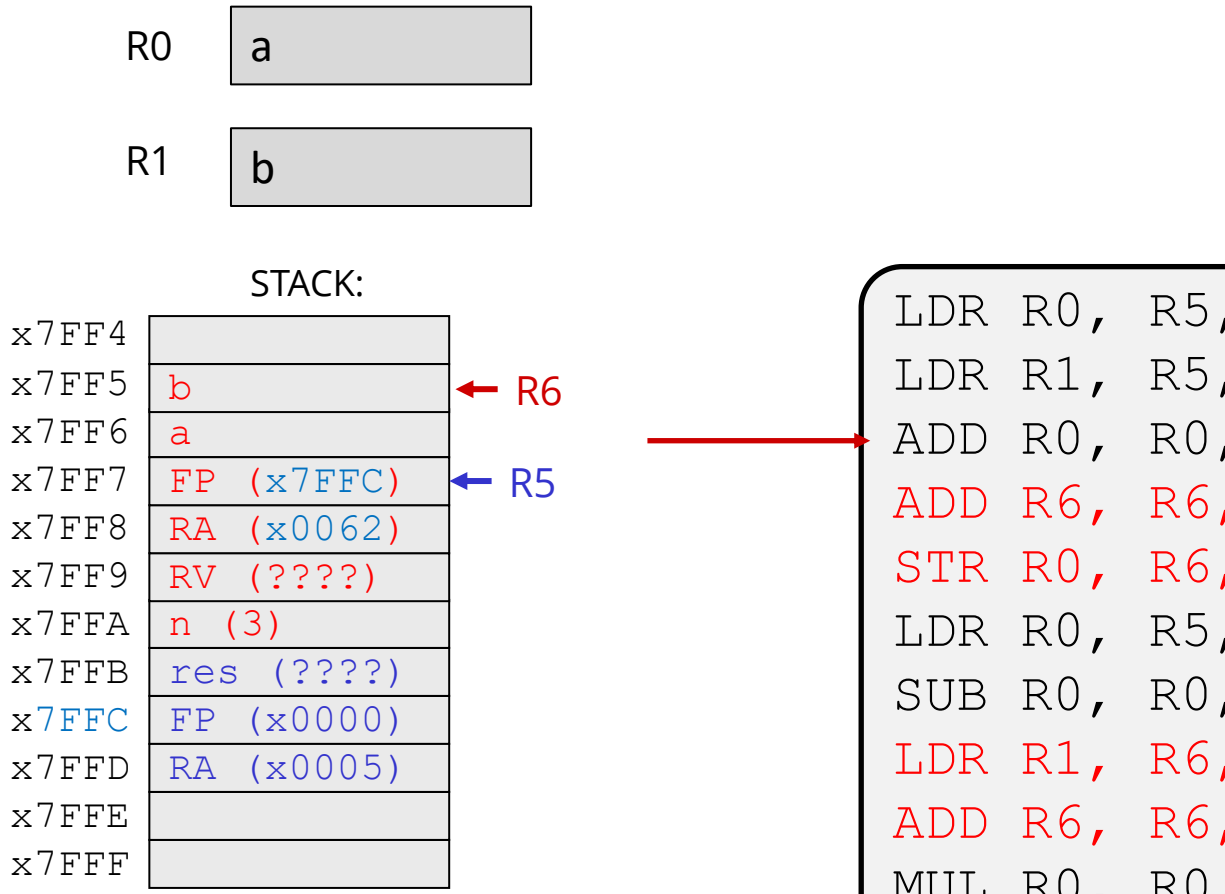



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

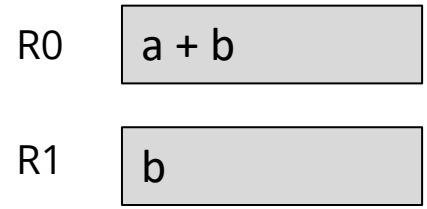


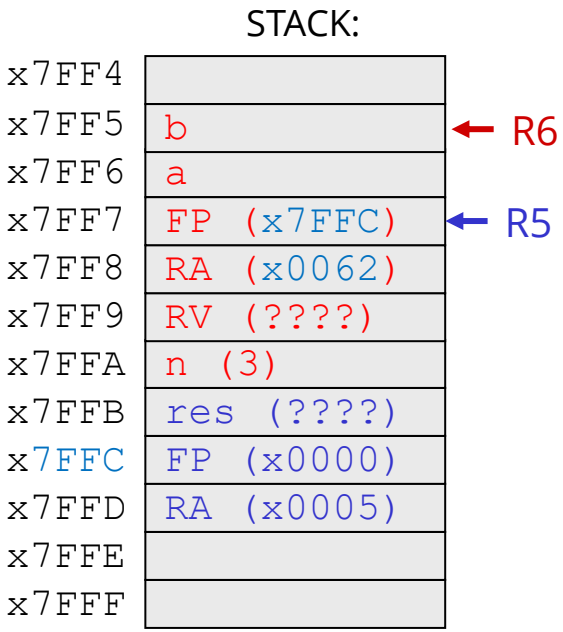

```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`



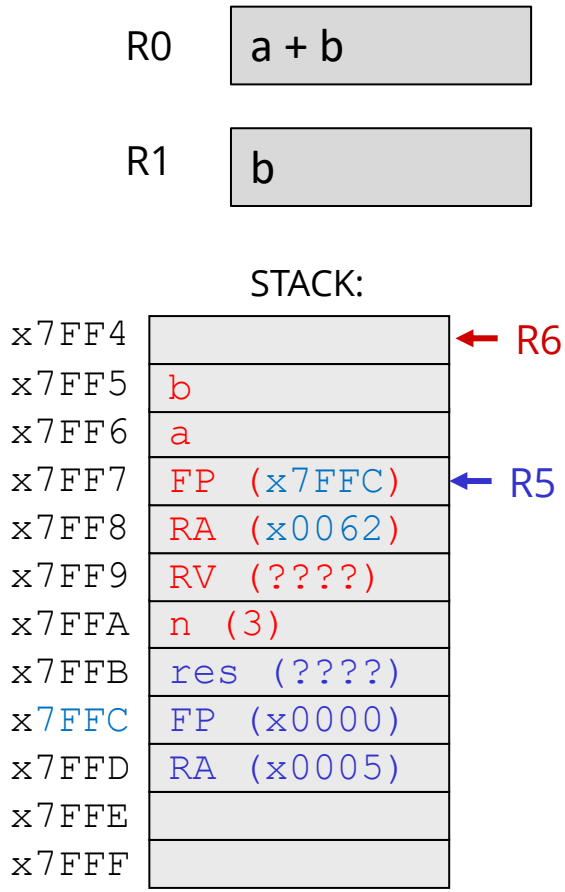



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

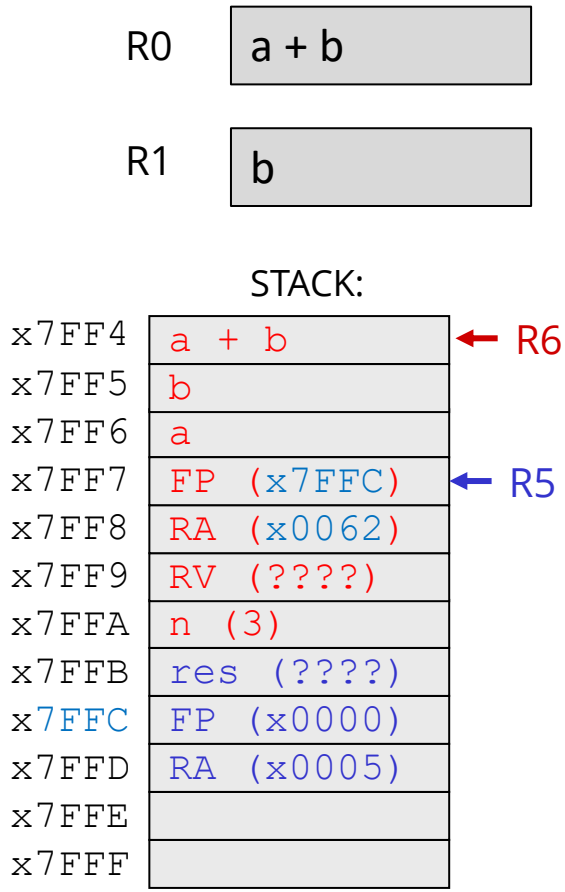



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

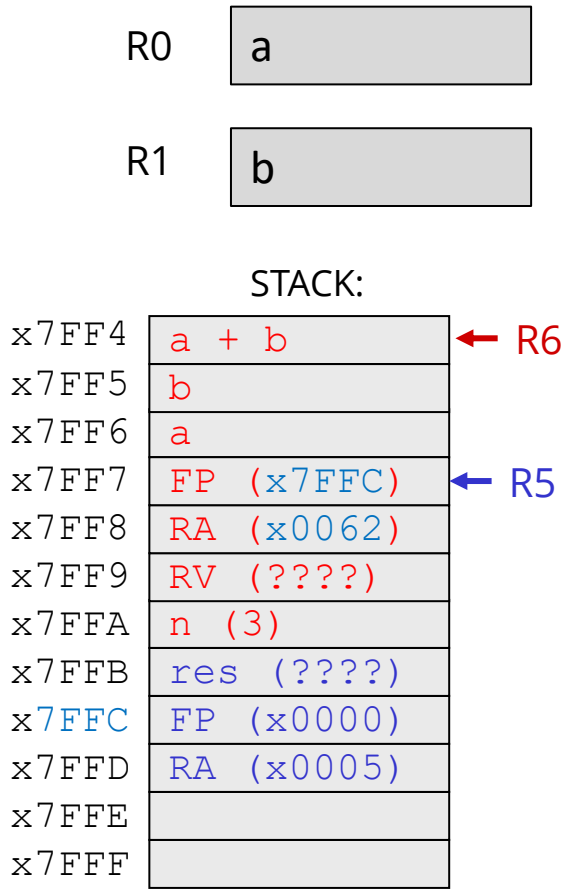



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

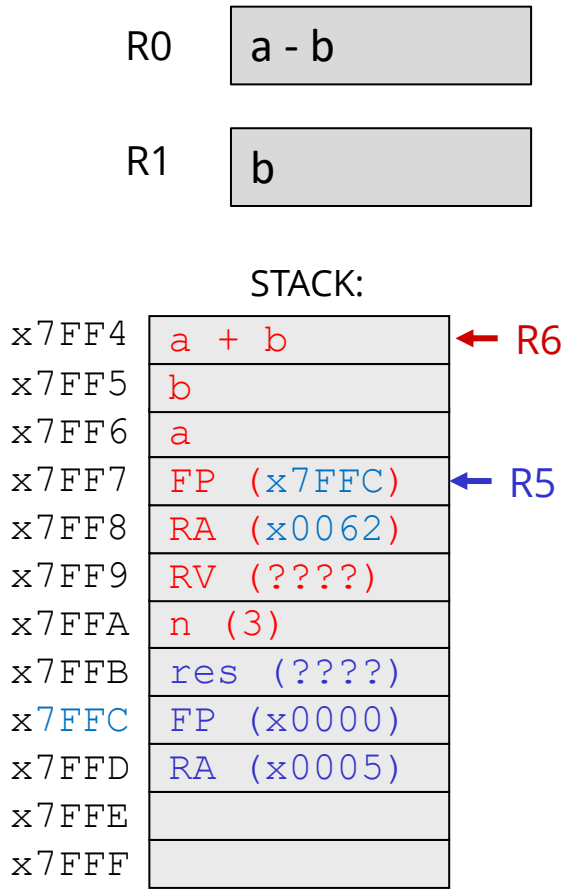



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

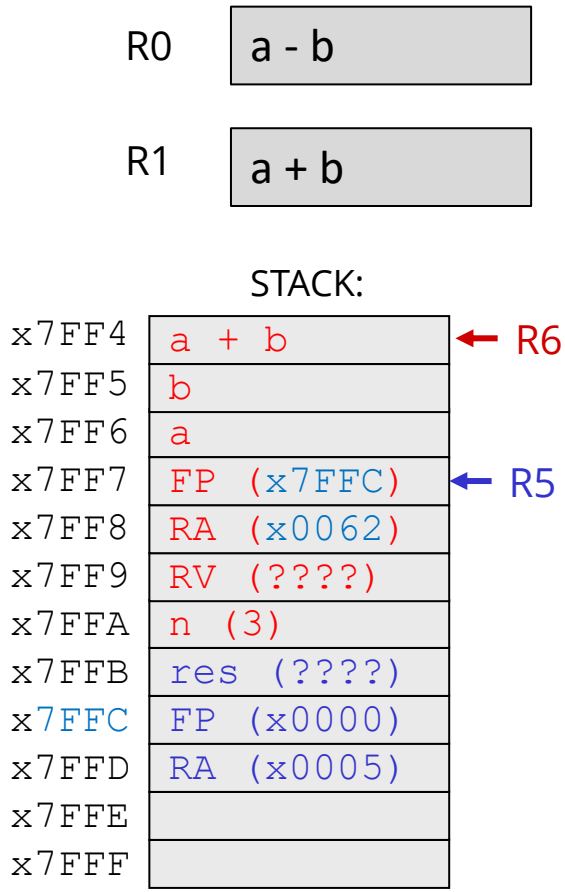



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

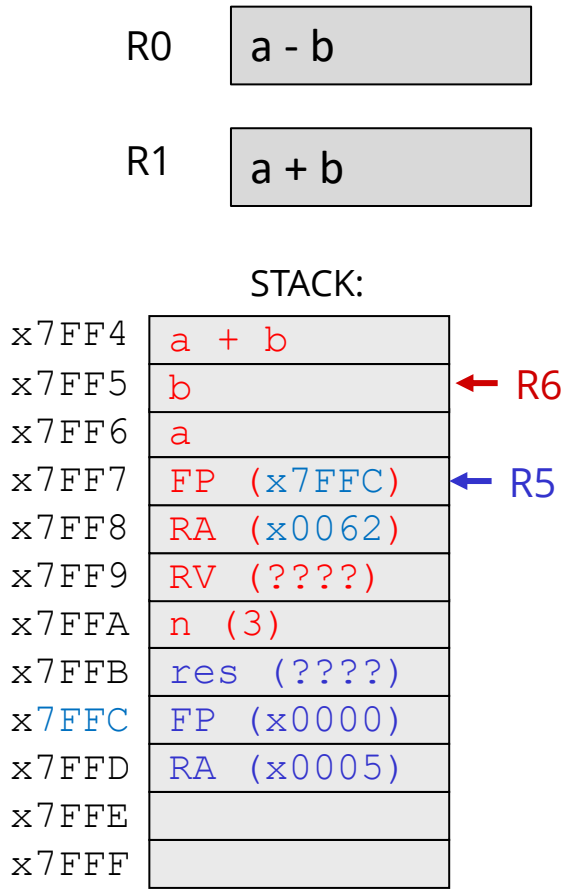



```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`




```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

R0 | (a - b) * (a + b)

R1 | a + b

STACK:

x7FF4	a + b	
x7FF5	b	← R6
x7FF6	a	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (????)	
x7FFA	n (3)	
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling Registers Walkthrough

Red arrow is next instruction to execute

- `a = (a + b) * (a - b);`

R0 (a - b) * (a + b)

R1 a + b

STACK:

x7FF4	a + b	
x7FF5	b	← R6
x7FF6	(a+b)*(a-b)	
x7FF7	FP (x7FFC)	← R5
x7FF8	RA (x0062)	
x7FF9	RV (????)	
x7FFA	n (3)	
x7FFB	res (????)	
x7FFC	FP (x0000)	
x7FFD	RA (x0005)	
x7FFE		
x7FFF		

```
LDR R0, R5, #-1   ; r0 = a
LDR R1, R5, #-2   ; r1 = b
ADD R0, R0, R1
ADD R6, R6, #-1
STR R0, R6, #0
LDR R0, R5, #-1   ; r0 = a
SUB R0, R0, R1
LDR R1, R6, #0
ADD R6, R6, #1
MUL R0, R0, R1
STR R0, R5, #-1
```

Spilling In Real Life

- x86 and ARM ISA's include have finite registers. both also have registers used for frame pointer and base pointer

- To compile many expressions, compiler will sometimes have to spill operands onto the call stack like we just di

Spilling: Another Use Case

- Given an arbitrary arithmetic expression in C:
  - A = (B + C)*(D/E – F/G) + ((G*E – H*F)*(S-T)/A)

- We can write this like we did with RPN, constantly pushing/popping values onto the stack. Only using 2 registers (aside from stack pointer and frame pointer)
  - B C + D E / F G / - * G E * H F * - S T – A / * +

Push 'B' onto the stack

Push 'C' onto the stack

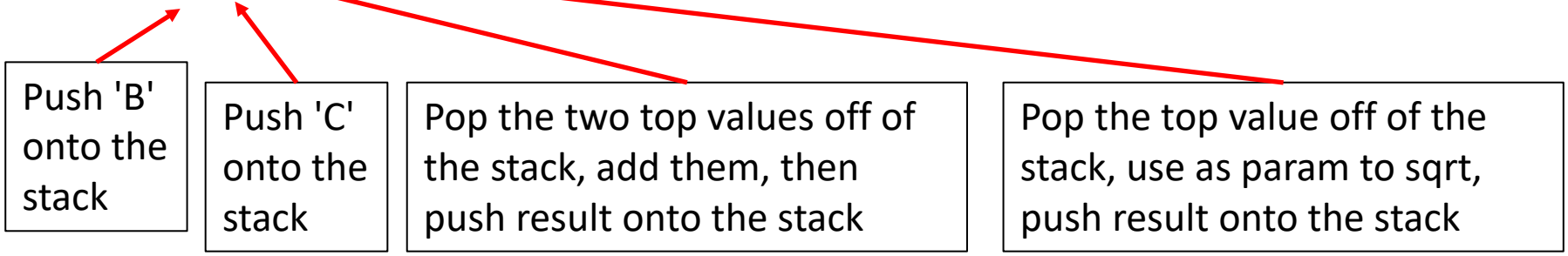
Pop the two top values off of the stack, add them, then push result onto the stack

This RPN-like example is something you will do for HW10 & HW11

Spilling: Another Use Case

- This can also be used for expressions with function calls
  - A = (sqrt(B + C) / sin(D + E*F)) – mod(B,F)

- Becomes:
  - **B C + SQRT D E F * + SIN / B F MOD -**



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

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

When to use R5 vs R6?

- R5 (frame pointer) stays relatively constant over function life-time. Make it easier for compiler to get:
  - Arguments from caller
  - Local Variables

- R6 (stack pointer) grows as we need to push arguments to callees and/or "spill" temporary values.
  - R6 is easier to use for getting/setting these spilt values and arguments

- ***Could*** use either for either cases, but it is more work for the compiler to keep track of.

Lecture Outline

- Stack Continued
  - Prologue
  - Epilogue
  - Register Spilling
- **Implications of the stack system**

Stack Mechanism

- The stack as a mechanism makes it easy to have functions that call other functions
- A function can even call itself (recursive functions) with the stack system.
- We will explore some of these implications with this example:

```
int fact(int n) {
  if(n <= 0) {
    return 1;
  }
  return n * fact(n - 1);
}
```

Implication 1

- What would happen if I were to run **fact**(x7FFF)? (maximum positive integer)

```
int fact(int n) {
  if(n <= 0) {
    return 1;
  }
  return n * fact(n - 1);
}
```

Implication 1

- What would happen if I were to tweak the code and call fact(2) ?

```
int fact(int n) {
  if(n <= 0) {
    return 1;
  }
  return n * fact(n);
}
```

Implication 1: Stack Overflow

- **The stack is finite**, if we allocate too many local variables or go "too deep" on function calls, we can run out of space and "overflow" the stack
  - (Hence the term Stack Overflow)
  - Doing this on modern machines almost always cause a crash (even in Java)

- Even if the code doesn't go "infinite" with broken recursion, we can still run out of space
  - Since the heap is larger, sometimes it is preferred to store larger data structures on the heap (or even global data)

Implication 1: Another Example

- A single stack frame could be big enough to cause a segmentation fault

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

// SIZE_MAX is 2^64
#define len (SIZE_MAX / 1000000000000)

int main(int argc, char** argv){
  printf("I ran\n");
  printf("buf size of: %ld\n", len);
  uint8_t buf[len];
  return EXIT_SUCCESS;
}
```

Implication 2:

- Here we have an iterative solution and a recursive solution. Which one requires more instructions to execute?

```
int fact(int n) {
  if(n <= 0) {
    return 1;
  }
  return n * fact(n - 1);
}
```

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

Implication 2: Performance

- Recursive solutions are typically slower than iterative solutions at run-time.
  - Recursive functions need to setup and destroy a new stack frame for each invocation -> which requires more instructions -> which requires more clock cycles.

- There are optimizations like tail-recursion that can be done, to make recursive examples about the same as iterative ones

- Though, sometimes recursive functions are easier to read/maintain.
- Sometimes the performance gain is not worth it

Next Time:

- More C -> ASM
- Ideology of Performance & C