

ECE 220

Lecture x0008 - 02/12

Slides based on material originally by: Yuting Chen & Thomas Moon

Recap

- Last time
 - Functions in C
 - Prototype vs. definition
 - Examples
 - Implementation in assembly & intro to Run-Time-Stack
 - Let's quickly review!

How do functions work at assembly level?

- When C-compiler compiles a program, it keeps track of variables in a program using a **symbol table**.
- For our purposes, the symbol table contains
 - Identifier
 - type of the variable,
 - memory location allocated (by offset - see next slide) and
 - scope

Getting this to work - example

```
int inGlobal=2;
int outGlobal=3;
int dummy(int in1, int in2);

int main(void){
    int x,y,z;
    ...
}

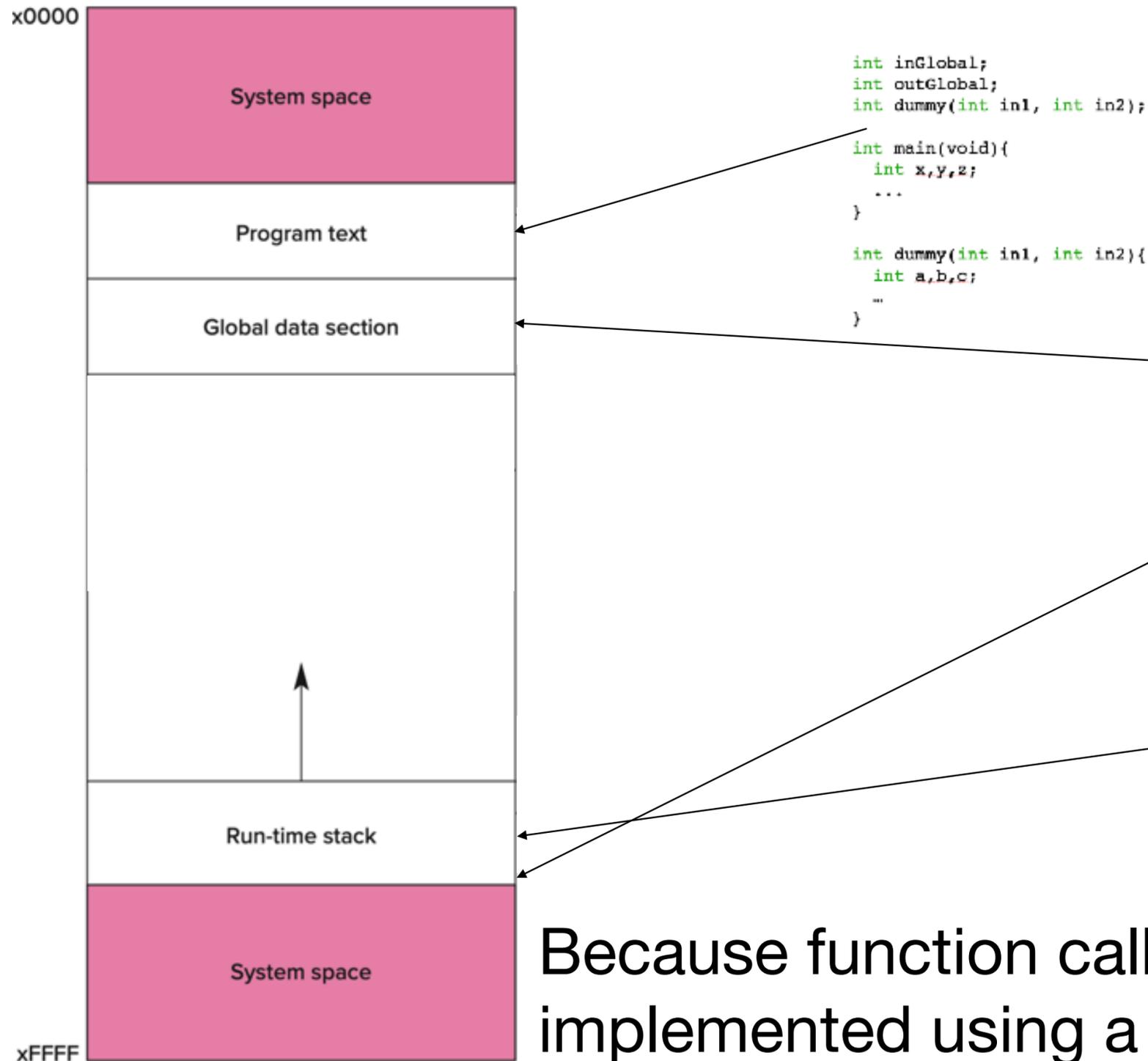
int dummy(int in1, int in2){
    int a,b,c;
    ...
}
```

Symbol table

Name	Type	Location	Scope
inGlobal	int	0	Global
outGlobal	int	1	Global
x	int	0	Main
y	int	-1	Main
z	int	-2	Main
a	int	0	Dummy
b	int	-1	Dummy
c	int	-2	Dummy

Why are some offsets negative and others positive?

Example: In LC3 memory map



Symbol table

Name	Type	Location	Scope
<code>inGlobal</code>	int	0	Global
<code>outGlobal</code>	int	1	Global
<code>x</code>	int	0	Main
<code>y</code>	int	-1	Main
<code>z</code>	int	-2	Main
<code>a</code>	int	0	Dummy
<code>b</code>	int	-1	Dummy
<code>c</code>	int	-2	Dummy

Because function calls are implemented using a stack ADT.

Run-time stack: A place (actually a stack data structure) to hold *activation frames*

Basic idea

Activation record: Parts of a *stack* that holds information about each function call (sometimes called *stack frames*)

- **Every** function *call* creates an **activation record** (or stack frame) and pushes it onto the **run-time stack**.
- Whenever a function *completes* (returns), the activation record is popped off the run-time stack
- Whenever a function calls *another one* (nested, including itself), the run time stack grows (pushes another activation record onto the run-time stack).

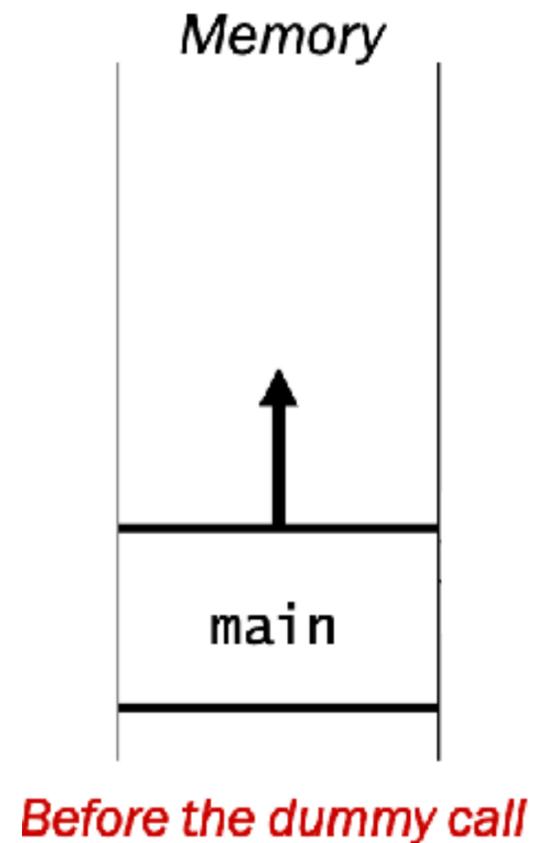
Arguments passed in
Variables defined in function
Bookkeeping information

Example: function call

```
int dummy(int in1, int in2);

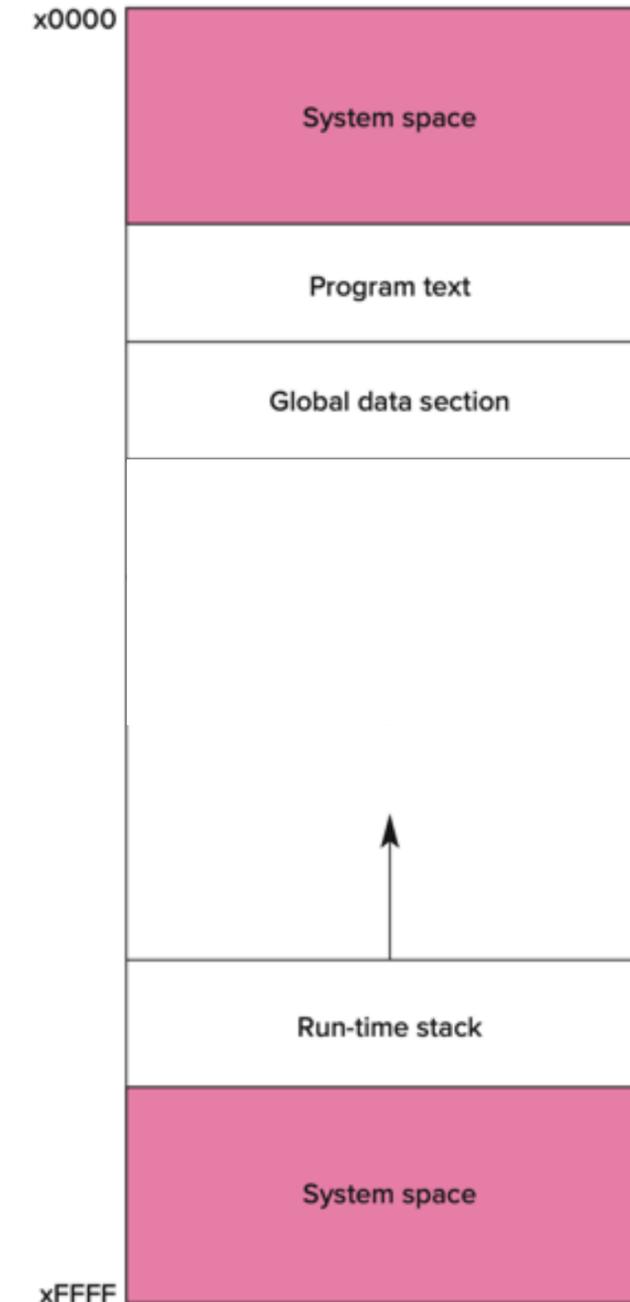
int main(void){
    int x,y,z;
    ...
    z = dummy(x, y);
}

int dummy(int in1, int in2){
    int a,b,c;
    ...
}
```



How to keep track?

- Store pointers:
 - Program counter - **PC**
 - **Global pointer** pointing to first global variable - **R4**
 - Top of stack, called **stack pointer** - **R6**
 - *Current frame pointer* - **R5**
 - Actually points to first local variable of *current* function

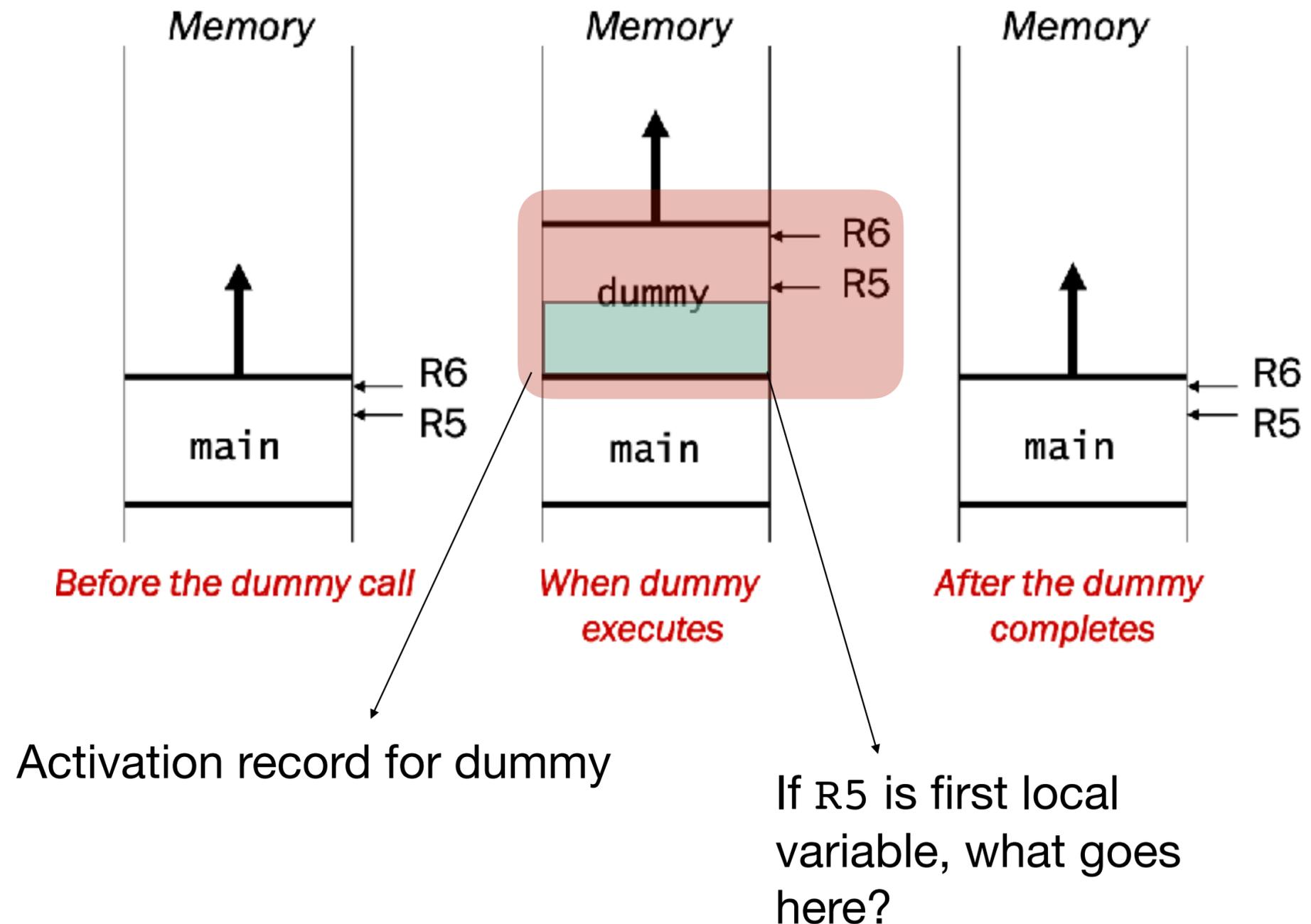


Example: function call

```
int dummy(int in1, int in2);

int main(void){
    int x,y,z;
    ...
    z = dummy(x, y);
}

int dummy(int in1, int in2){
    int a,b,c;
    ...
}
```



Details of a function call

- To *successfully* transfer execution between the caller and callee a few things need to be taken care of:

- Arguments need to be passed around Activation record
- Bookkeeping has to be done:
 - **Return value:** Space for value returned by function according to type has to be allocated
 - **Return address:** Pointer to next instruction has to be saved so caller can resume
 - Caller's frame pointer saved
- Callee local variables have to be stored Pushed before local variables

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack

2. Pass control to callee (JSR/JSRR)

Stack build up

3. *Callee* build-up: (push bookkeeping info and local variables onto stack)

4. Execute function

5. *Callee* tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)

6. Return to caller (RET)

Stack teardown

7. *Caller* tear-down (pop callee's return value and arguments from stack)

Caller

Callee

Caller

Example function call

```
int main (void){
    int a;
    int b;
    ...
    b = Watt(a);           // main calls Watt first
    b = Volt(a, b);       // then calls Volt
}

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

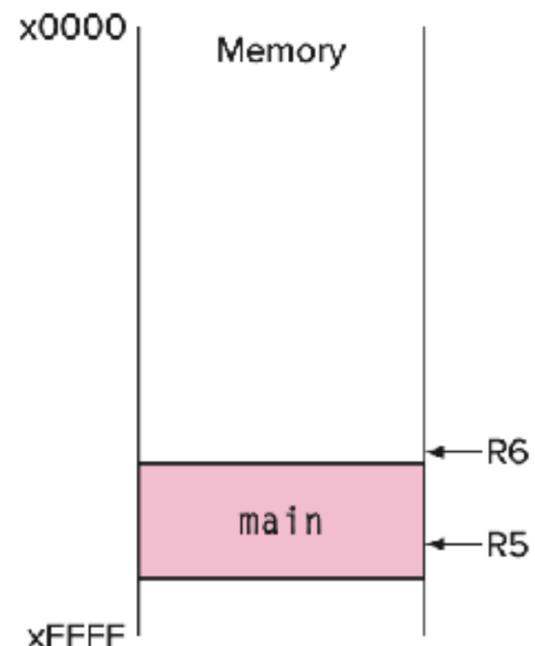
int Watt(int a) {
    int w;
    ...
    w = Volt(w,10);       // Watt also calls Volt
    ...
    return w;
}
```

Run-time stack

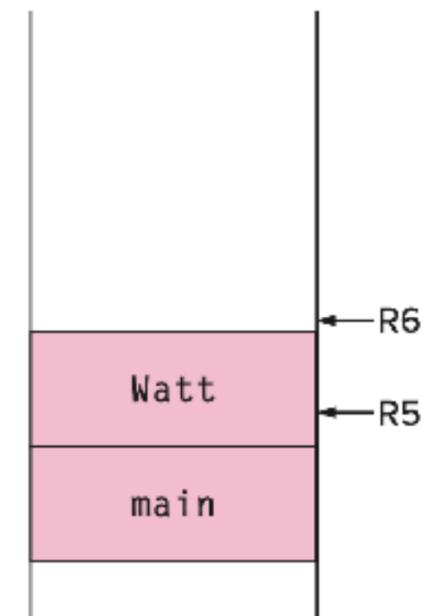
```
int main (void){
    int a;
    int b;
    ...
    b = Watt(a);
    b = Volt(a, b);
}
```

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

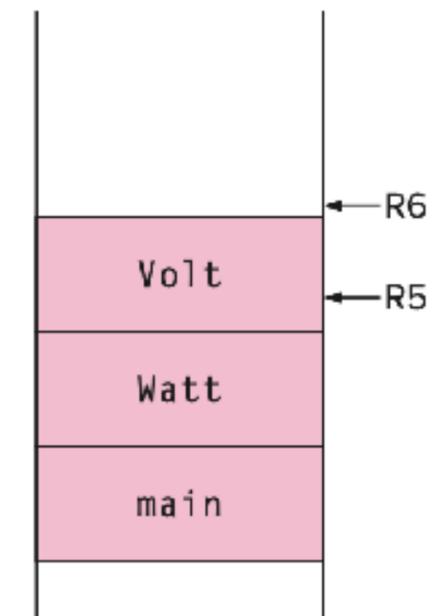
```
int Watt(int a) {
    int w;
    ...
    w = Volt(w, 10);
    ...
    return w;
}
```



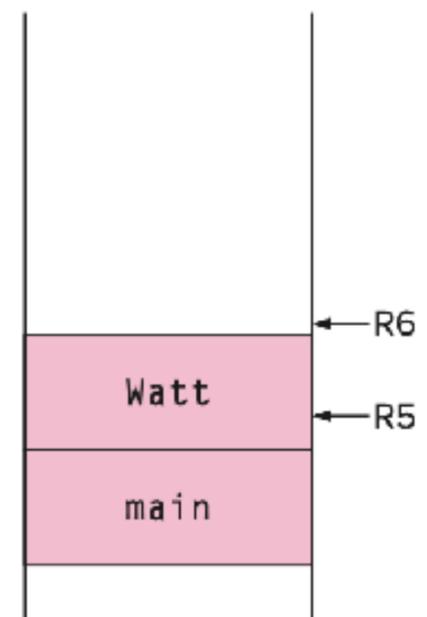
(a) Run-time stack when execution starts



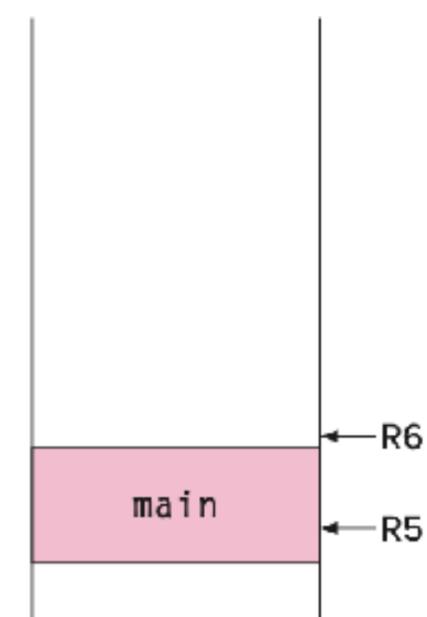
(b) When Watt executes



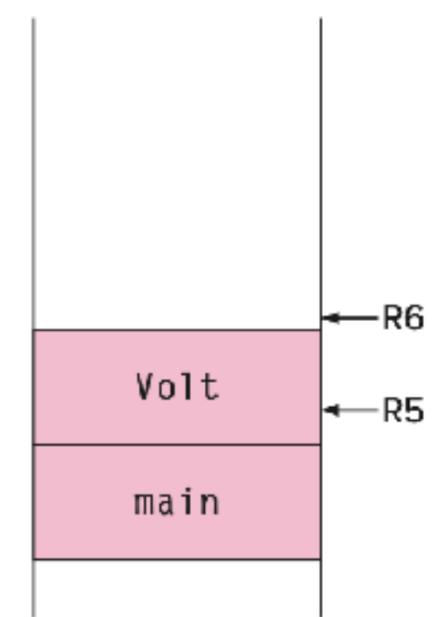
(c) When Volt executes



(d) After Volt completes



(e) After Watt completes



(f) When Volt executes

C Run-time stack protocol

- **STEP 1:** The **caller** function copies arguments for the **callee** onto the run-time stack and passes control to the **callee**.
- **STEP 2:** The **callee** function pushes space for local variables and other information onto the run-time stack, essentially creating its stack frame on top of the stack.
- **STEP 3:** The **callee** executes
- **STEP 4:** Once it is ready to return, the **callee** pops its stack frame off the run-time stack, and gives the *return value* and control to the **caller**.

C Run-time stack protocol

- `volt` called with two arguments
- Value *returned* by `volt` is assigned to local integer variable `w`
- *Arguments* are pushed onto stack from **right to left** in the order in which they appear in the function call

```
int Watt(int a)
{
    int w;
    ...
    w = Volt(w, 10);
    ...
    return w;
}
```

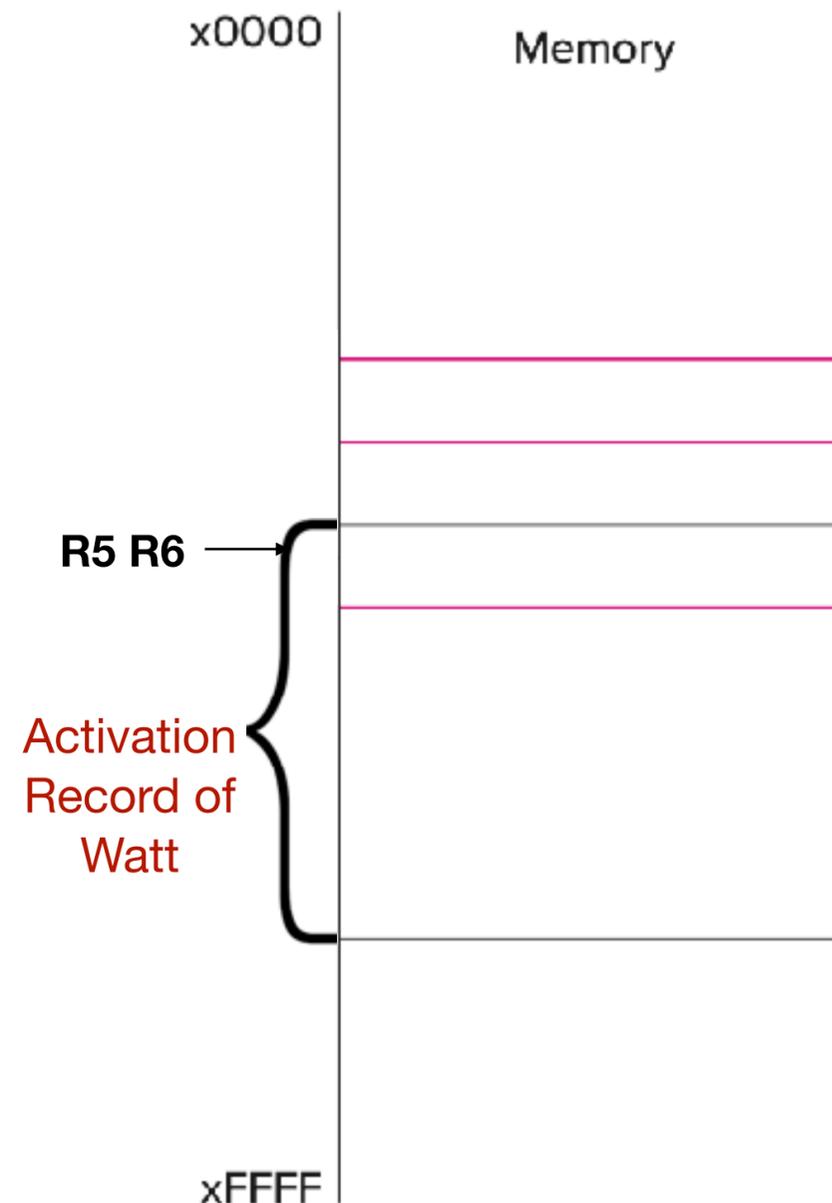
LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)
2. Pass control to callee

```
; push second arg  
AND R0, R0, #0  
ADD R0, R0, #10  
ADD R6, R6, #-1  
STR R0, R6, #0
```

```
; push first arg  
LDR R0, R5, #0 ;R ← w  
ADD R6, R6, #-1  
STR R0, R6, #0
```

```
; call subroutine  
JSR VOLT
```



```
int Watt(int a)  
{  
    int w;  
    ...  
    w = Volt(w, 10);  
    ...  
    return w;  
}
```

LC-3 Implementation

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

3. Callee setup (push bookkeeping info and local variables onto stack)

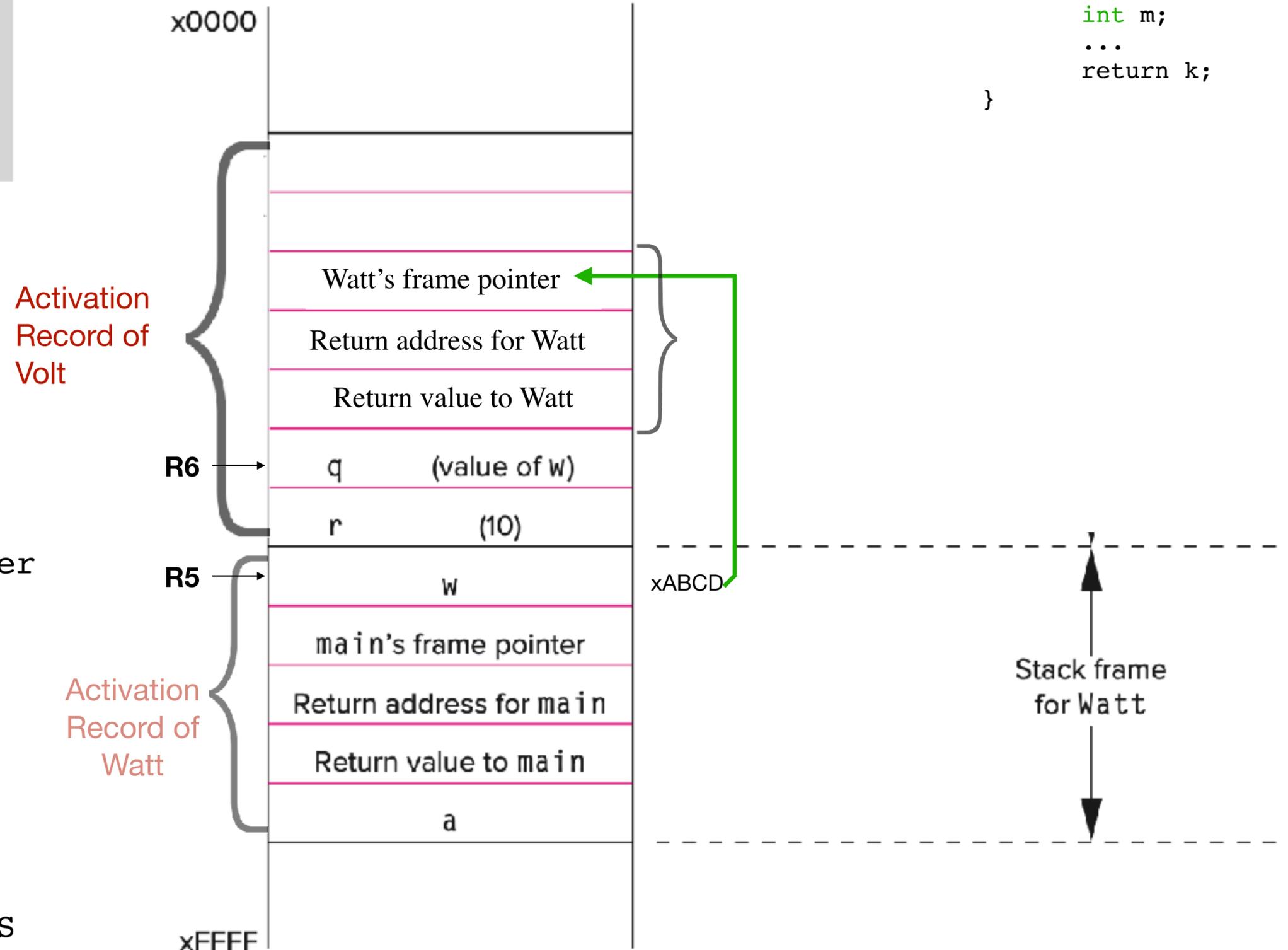
4. Execute function

```
;space for return value
ADD R6, R6, #-1
```

```
;space for return address
ADD R6, R6, #-1
;Push R7 (Return Addr)
STR R7, R6, #0
```

```
;space for Caller Frame Pointer
ADD R6, R6, #-1
;Push R5 (CFP)
STR R5, R6, #0
```

```
;Set frame pointer for Volt
ADD R5, R6, #-1
;
;Space for local variables
ADD R6, R6, #-2 ; updates TOS
```



```

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

```

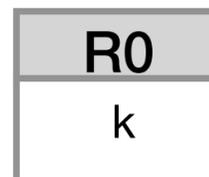
LC-3 Implementation

5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller

```

; copy k into return value(R5+3)
LDR R0, R5, #0
STR R0, R5, #3

```



```

; pop local variables
ADD R6, R5, #1

```

```

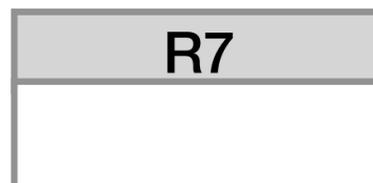
; pop Watt's frame pointer (to R5)
LDR R5, R6, #0
ADD R6, R6, #1

```

```

; pop return addr (to R7)
LDR R7, R6, #0
ADD R6, R6, #1

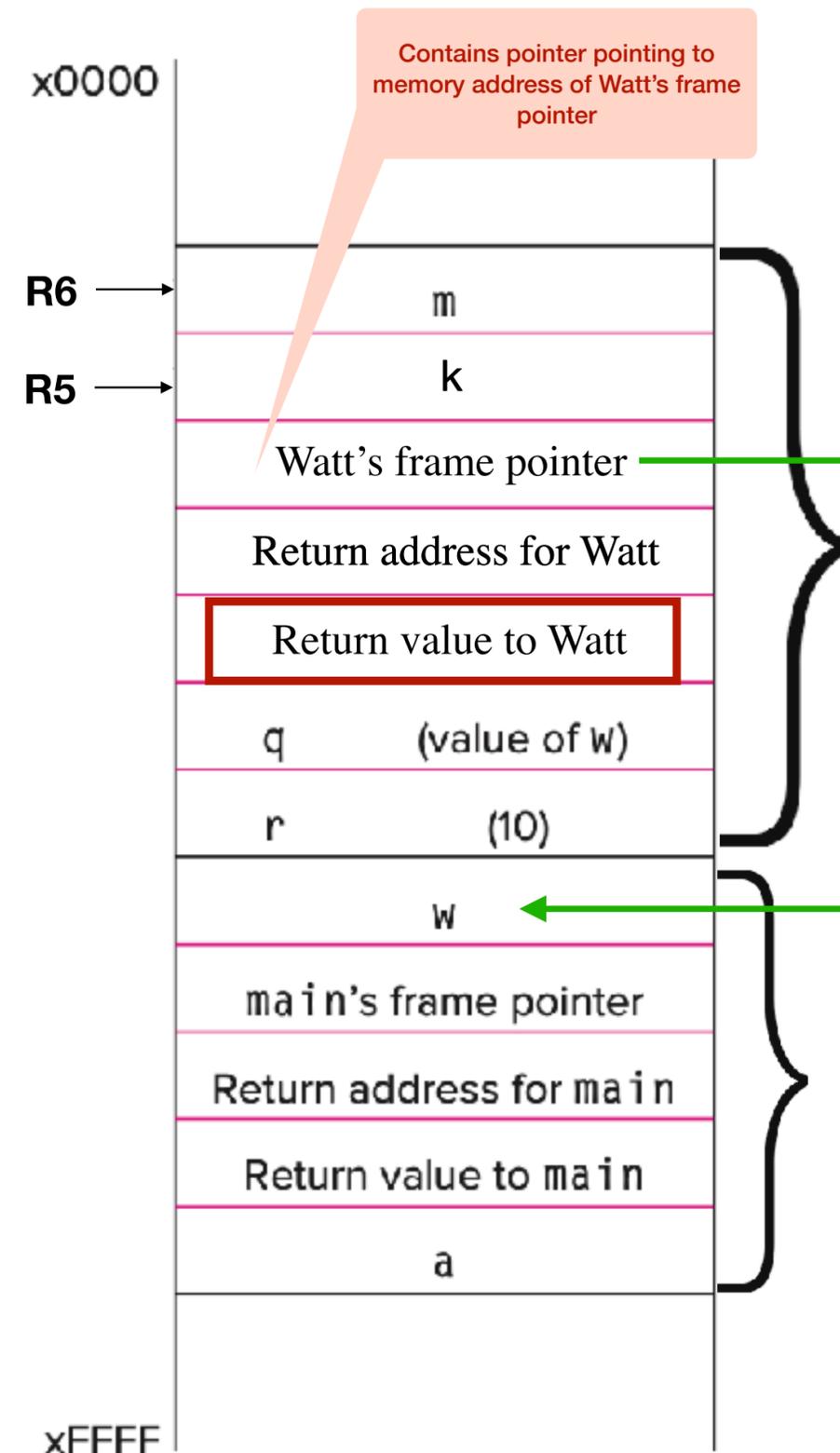
```



```

; return control to caller
RET

```



Note :
 Even though the stack frame for Volt is popped off the stack, its values remain in memory until they are explicitly overwritten

Activation Record of Volt

Activation Record of Watt

LC-3 Implementation

```
int Watt(int a)
{
    int w;
    ...
    w = Volt(w, 10);
    ...
    return w;
}
```

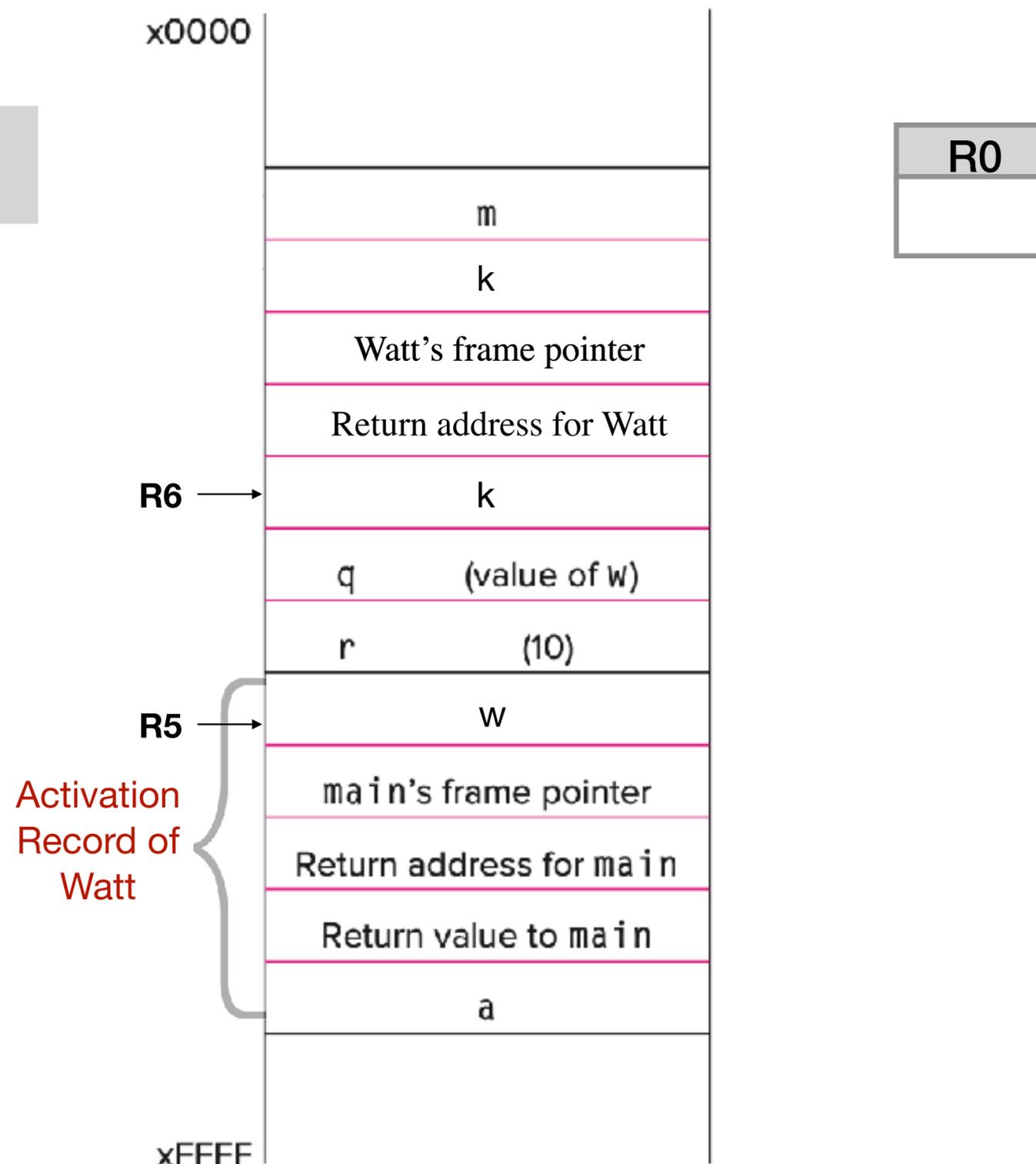
7. Caller tear-down (pop callee's return value and arguments from stack)

```
JSR VOLT
; load return value (top of stack)
LDR R0, R6, #0

; perform assignment
STR R0, R5, #0

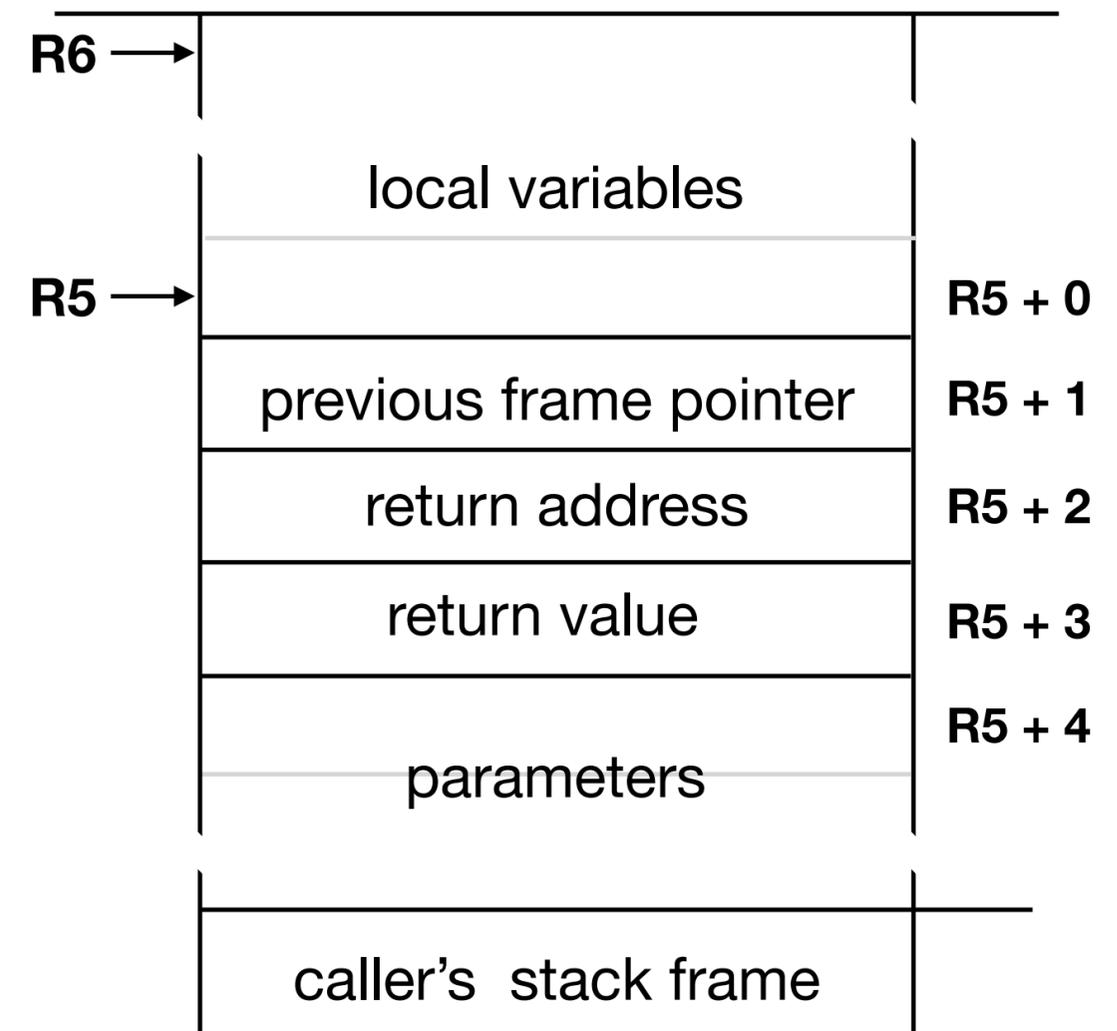
; pop return value
ADD R6, R6, #1

; pop arguments
ADD R6, R6, #2
```



General principles

- **R4** points first global variable
- **R5** points to first local variable
- **R6** is top of stack
- **R7** is reserved for RET
- **R0–R3** are caller saved



Exercise: build the activation frame

```
void Swap(int first, int second);

int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(valueA, valueB);
}

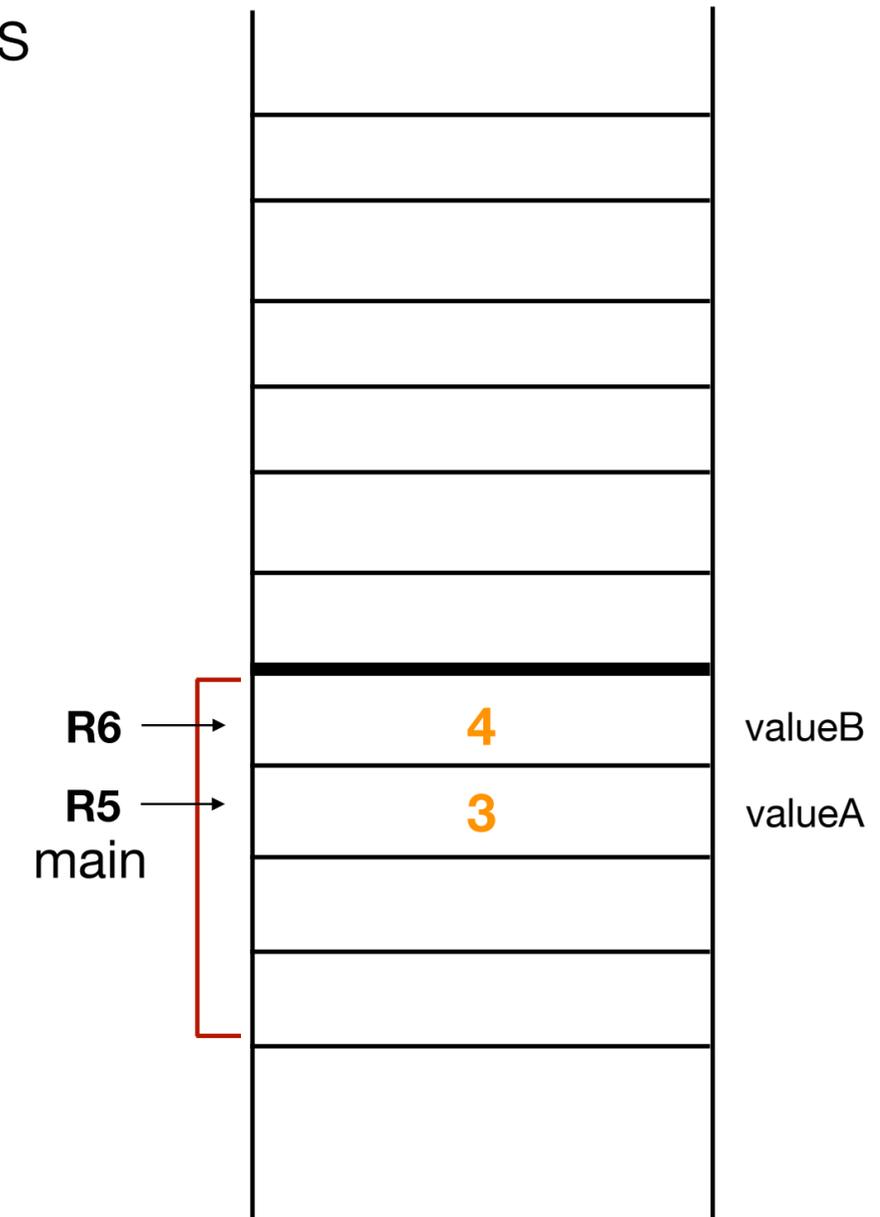
void Swap(int first, int second){
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

Goal:

Swap valueA and valueB in main.

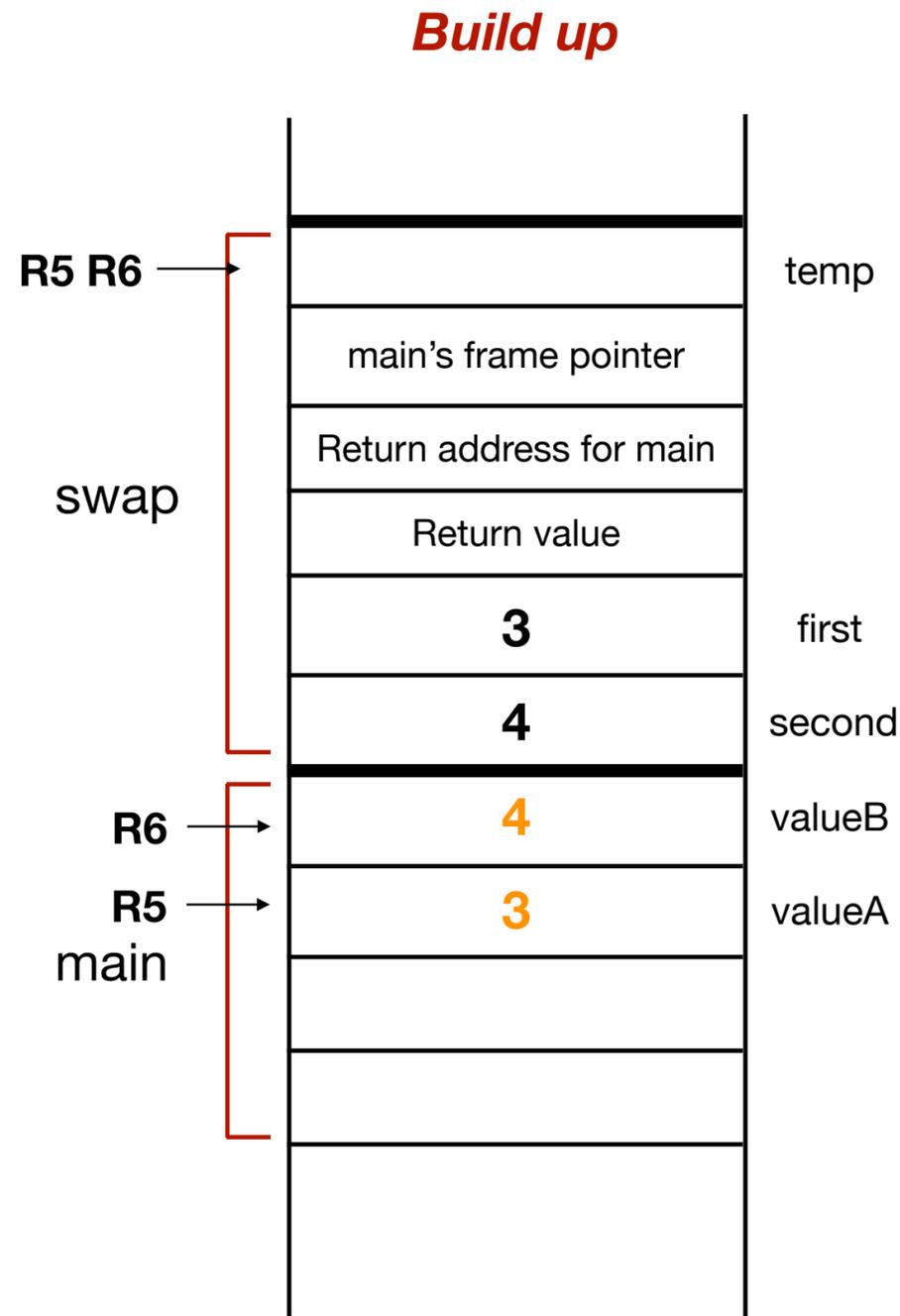
1. Push arguments (R-to-L) onto RTS
2. JSR
3. Callee build up
 - A. Return value
 - B. Return address
 - C. Caller frame pointer (CFP)
 - D. Push local variables
4. Execute
5. Callee tear down
 - E. Update return value
 - F. Pop local variables
 - G. Pop CFP
 - H. Pop return address
6. RET
7. Caller tear down
 - I. Pop return value
 - J. Pop arguments

Before call



swap function - build up

1. Push arguments (R-to-L) onto RTS
2. JSR
3. Callee build up (push onto RTS)
 - A. Return value (allocate)
 - B. Return address (from R7)
 - C. Caller frame pointer (CFP)
 - D. Local variables
4. Execute
5. Callee tear down
 - E. Update return value
 - F. Pop local variables
 - G. Pop CFP (into R5)
 - H. Pop return address (into R7)
6. RET
7. Caller tear down
 - I. Pop return value
 - J. Pop arguments



```
void Swap(int first, int second);

int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(valueA, valueB);
}

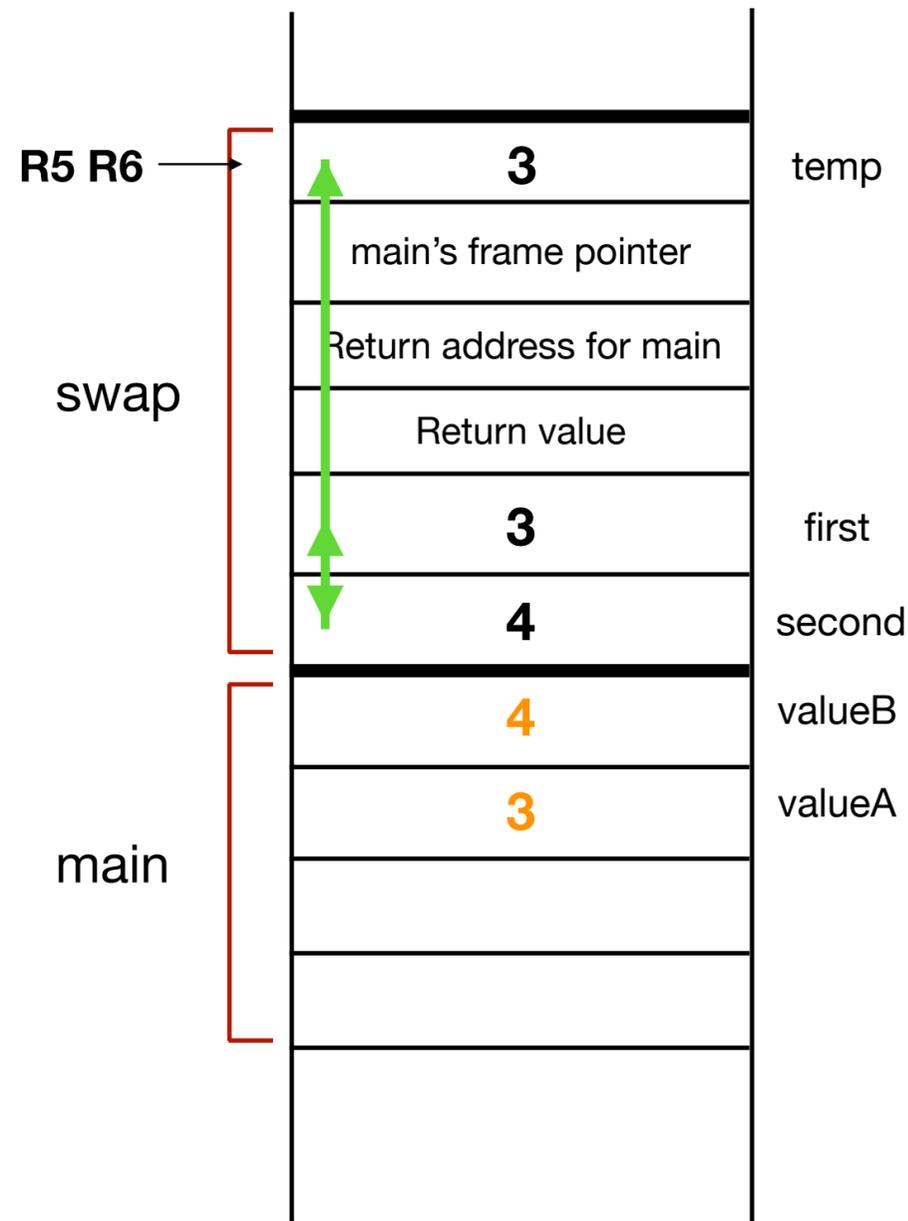
void Swap(int first, int second){
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

```
ADD R6, R6, #-1
ADD R5, R6, #01
```

swap function - execute

1. Push arguments (R-to-L) onto RTS
2. JSR
3. Callee build up (push onto RTS)
 - A. Return value (allocate)
 - B. Return address (from R7)
 - C. Caller frame pointer (CFP)
 - D. Local variables
4. Execute
5. Callee tear down
 - E. Update return value
 - F. Pop local variables
 - G. Pop CFP (into R5)
 - H. Pop return address (into R7)
6. RET
7. Caller tear down
 - I. Pop return value
 - J. Pop arguments

Execution



```

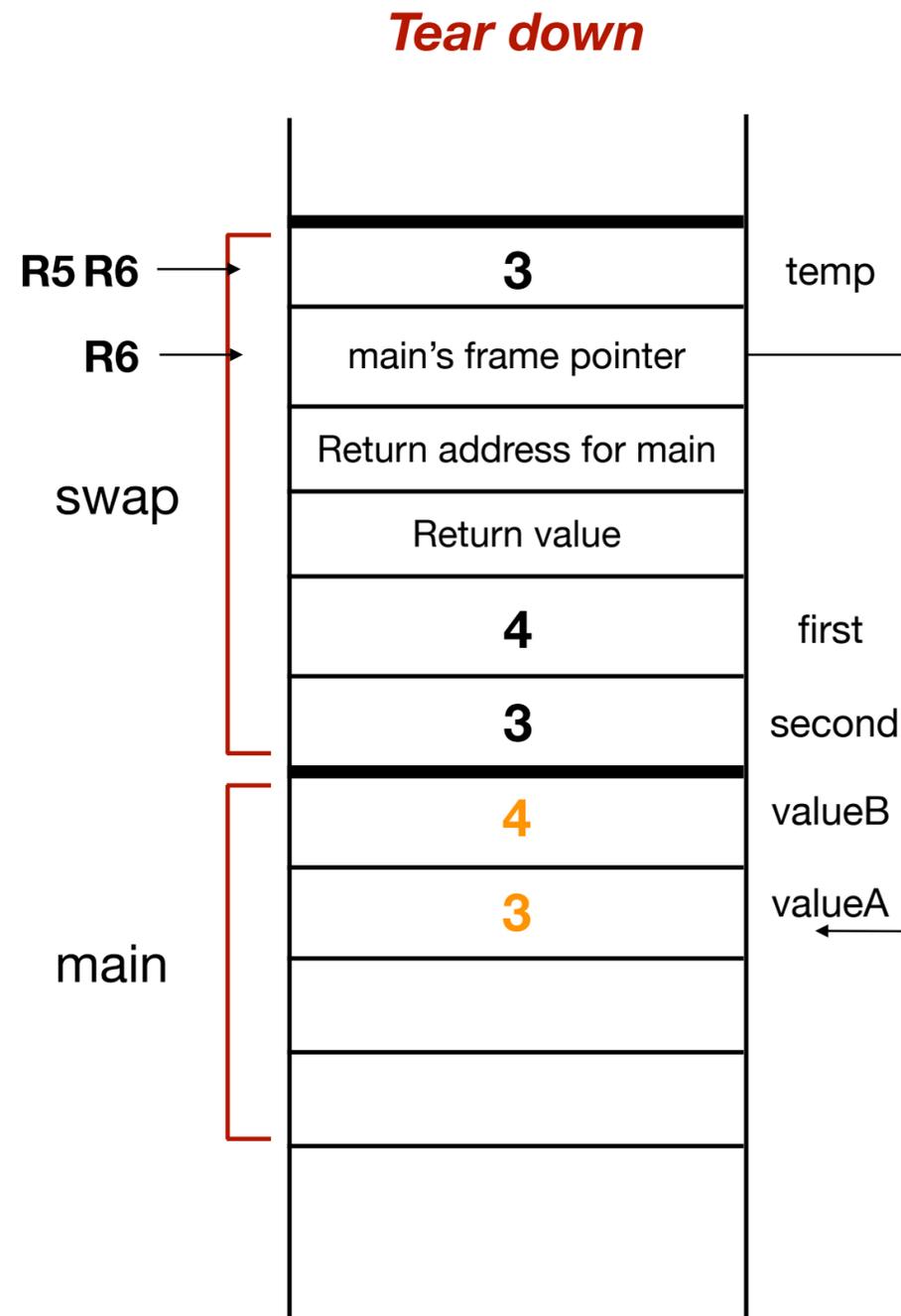
void Swap(int first, int second);

int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(valueA, valueB);
}

void Swap(int first, int second){
    int temp;
    temp = first;
    first = second;
    second = temp;
}
    
```

swap function - tear down

1. Push arguments (R-to-L) onto RTS
2. JSR
3. Callee build up (push onto RTS)
 - A. Return value (allocate)
 - B. Return address (from R7)
 - C. Caller frame pointer (CFP)
 - D. Local variables
4. Execute
5. Callee tear down
 - E. Update return value
 - F. Pop local variables
 - G. Pop CFP (into R5)
 - H. Pop return address (into R7)
6. RET
7. Caller tear down
 - I. Pop return value
 - J. Pop arguments



R7
Return address for main

```
void Swap(int first, int second);

int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(valueA, valueB);
}

void Swap(int first, int second){
    int temp;
    temp = first;
    first = second;
    second = temp;
}
```

LC3 commands left as an exercise

Argument passing

- Argument passing in C is what we call **pass-by-value**:
 - The functions get their own copies of the arguments
 - Changes made to these local copies are not reflected back
- Contrast with **pass-by-reference**.
- What needs to be changed for the `swap` function to work?
 - Somehow the `swap` function needs to know the *memory locations* of the variables that `main` needs swapped
 - Enter **pointers**.

Introduction to pointers

Working version

```
#include <stdio.h>
void Swap(int *first, int *second);

int main(){
    int valueA = 3;
    int valueB = 4;
    Swap(&valueA, &valueB);
}

void Swap(int *first, int *second){
    int temp;
    temp = *first;
    *first = *second;
    *second = temp;
}
```

Recall from scanf:
what does `&var` do to
`var`?

How do we tell the compiler
some variables are
supposed to hold memory
addresses a.k.a *pointers* and
not usual values?

If you have pointer, how do you tell the
compiler you want to refer to its contents?

Too fast?

Don't miss next class! We will cover it again.