ECE 220

Lecture x0008 - 02/12

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

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Recap

Recap

- Last time

Recap

- Last time
  - Functions in C

Recap

- Last time
  - Functions in C
  - Prototype vs. definition

Recap

- Last time
  - Functions in C
    - Prototype vs. definition
- Examples

Recap

- Last time
  - Functions in C
    - Prototype vs. definition
  - Examples
  - Implementation in assembly & intro to Run-Time-Stack

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?

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**.

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

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

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,

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

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

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;
   …
}
```

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

Getting this to work - example

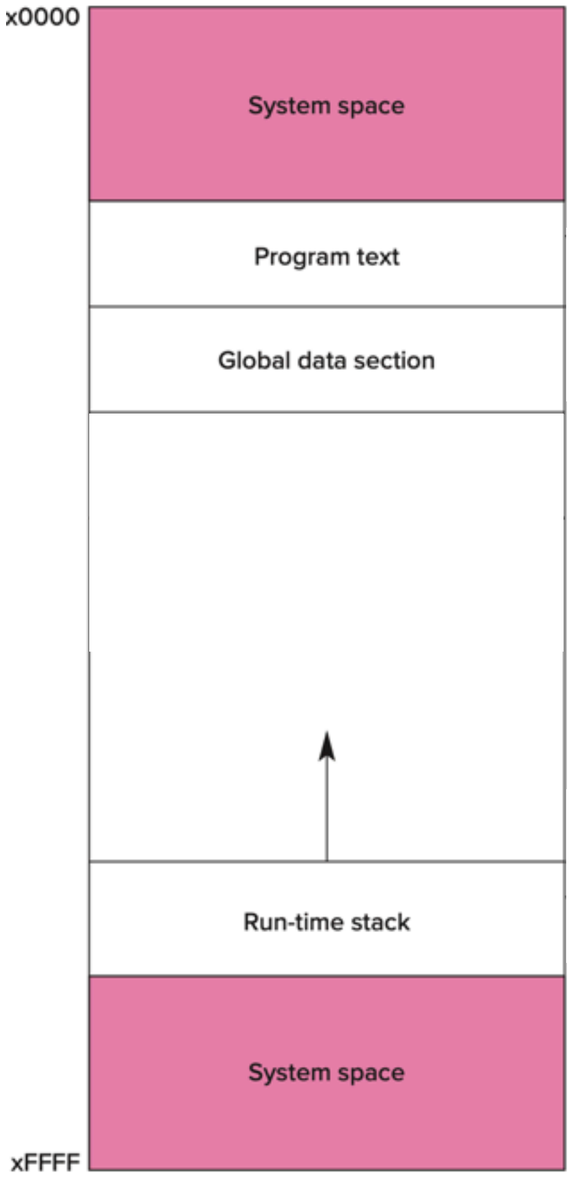
```
int inGlobal=2;
int outGlobal=3;
int dummy(int in1, int in2);

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?

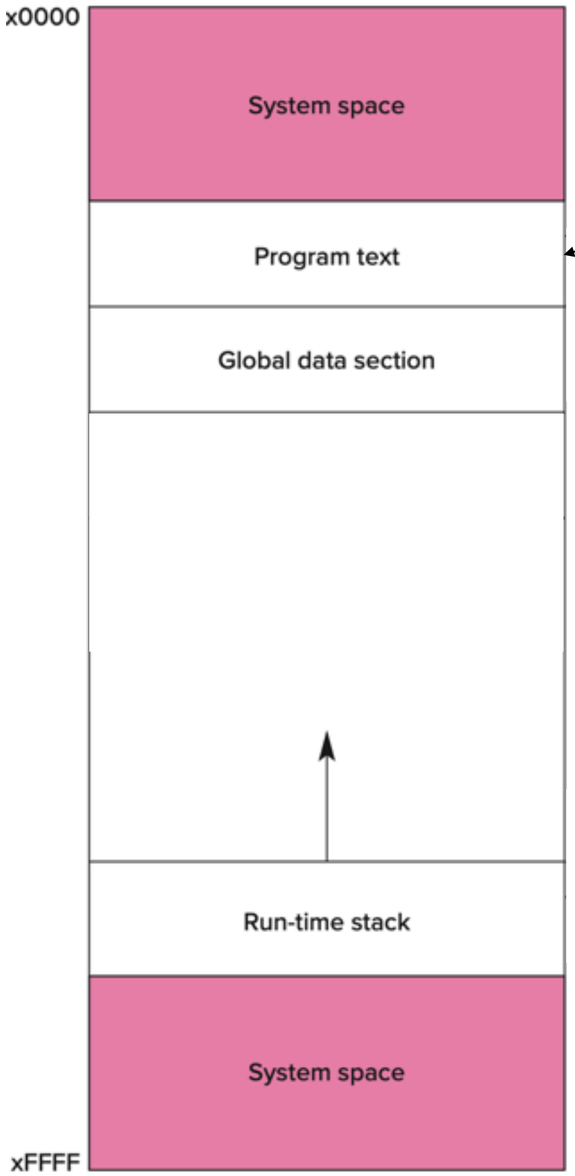
Example: In LC3 memory map



Symbol table

Name	Type	Location	Scope
`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

Example: In LC3 memory map

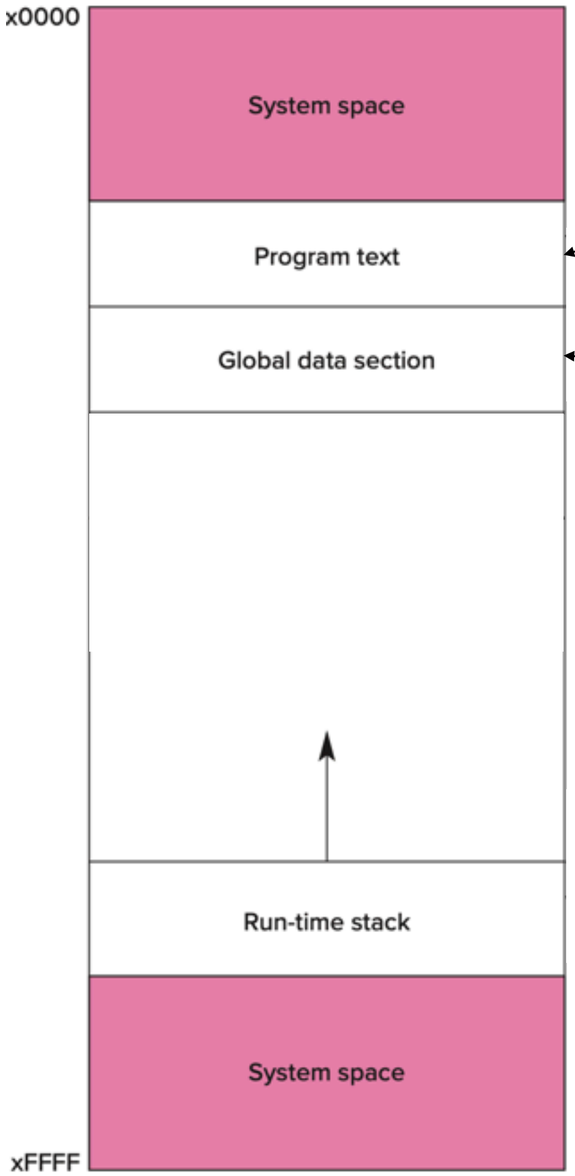



```c
int inGlobal;
int outGlobal;
int dummy(int in1, int in2);

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

Example: In LC3 memory map

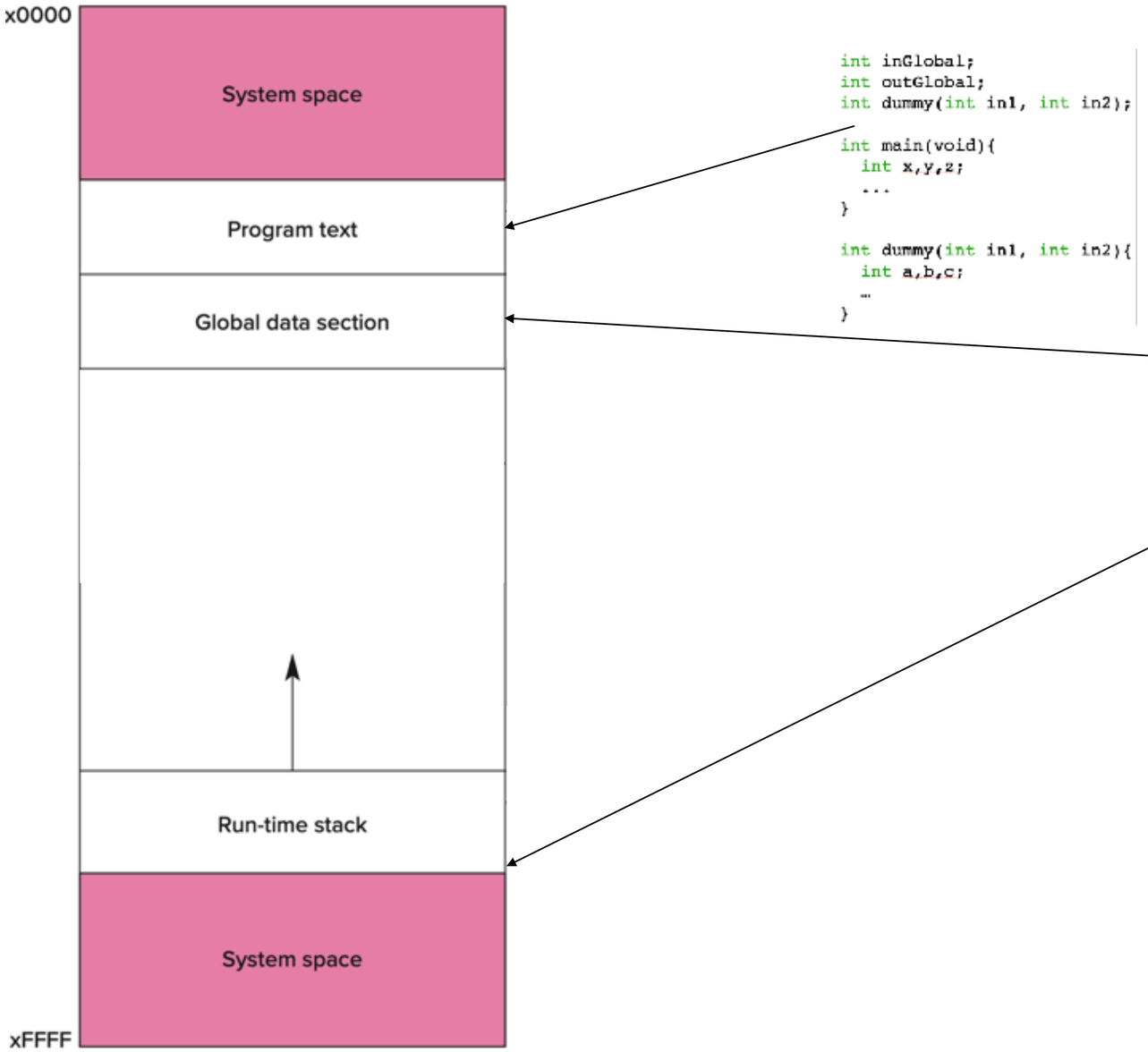



```
int inGlobal;
int outGlobal;
int dummy(int in1, int in2);

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

Example: In LC3 memory map

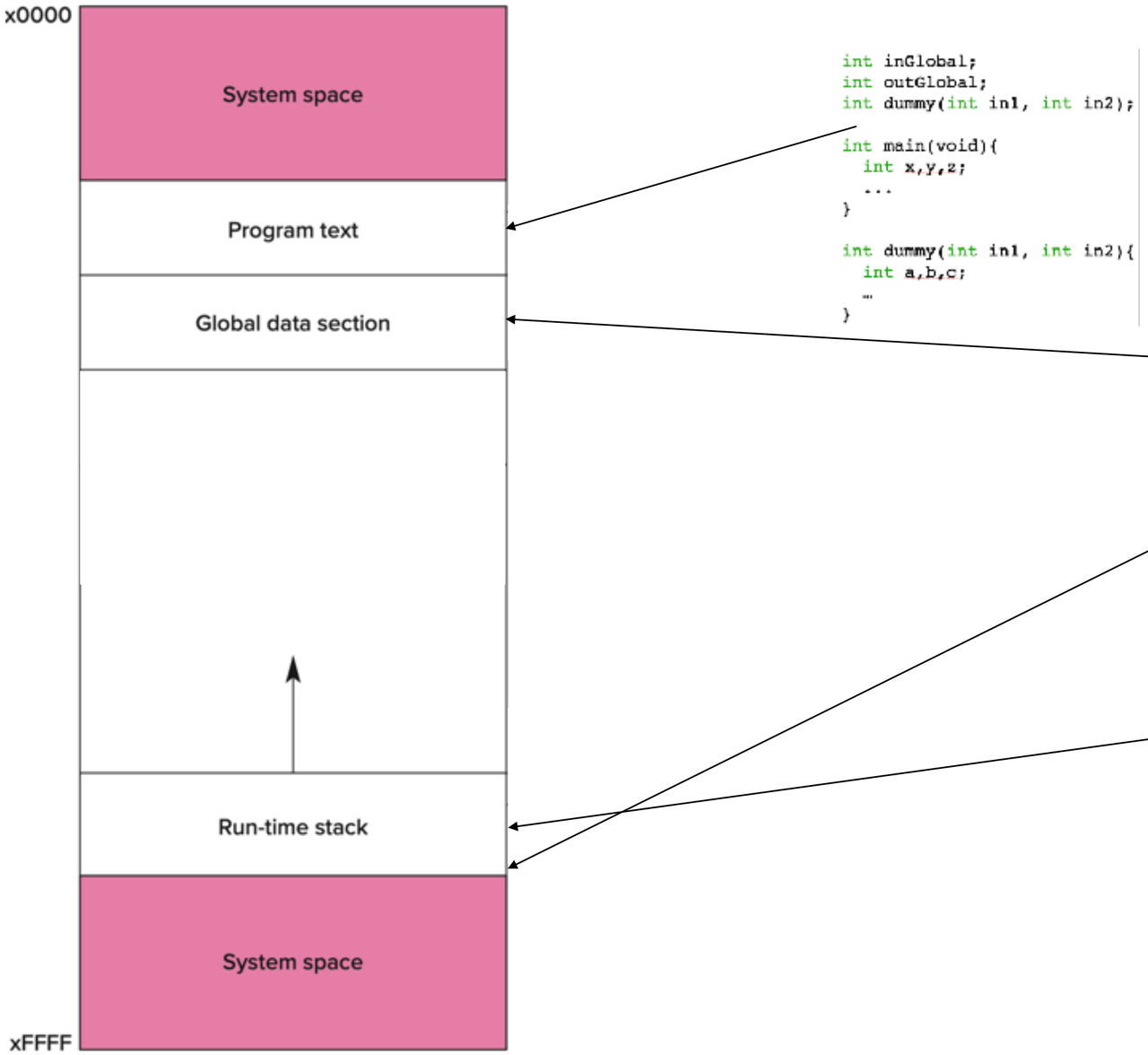



```c
int inGlobal;
int outGlobal;
int dummy(int in1, int in2);

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

Example: In LC3 memory map

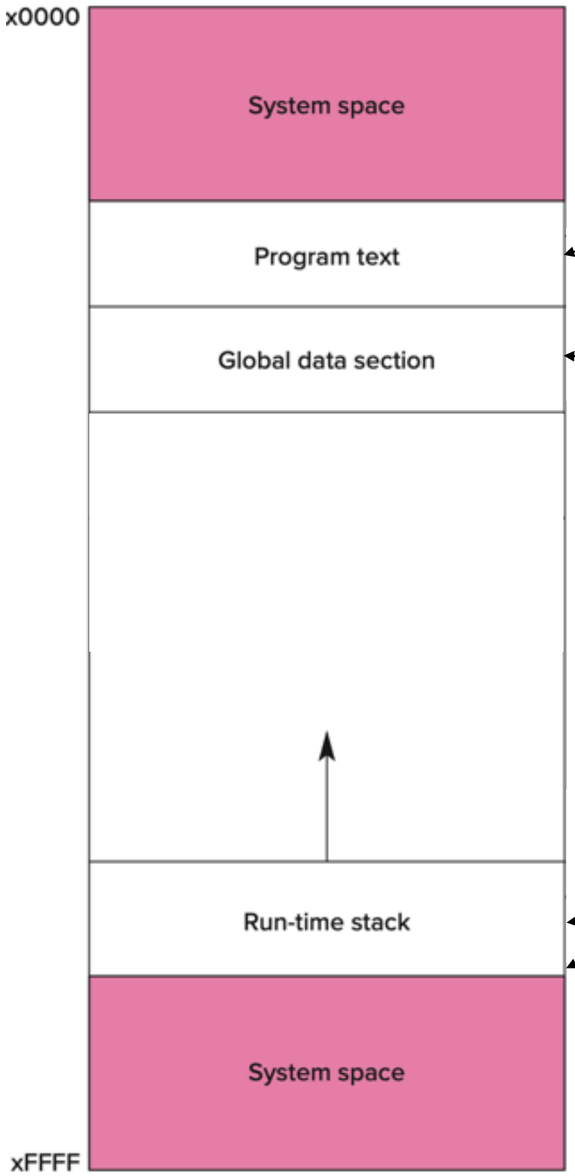



```
int inGlobal;
int outGlobal;
int dummy(int in1, int in2);

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

Example: In LC3 memory map




```c
int inGlobal;
int outGlobal;
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

Because function calls are implemented using a stack ADT.

Basic idea

Basic idea

- ***Every*** function *call* creates an activation record (or stack frame) and pushes it onto the run-time stack.

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.

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.

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.

Arguments passed in
Variables defined in function
Bookkeeping information

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

Arguments passed in
Variables defined in function
Bookkeeping information

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.

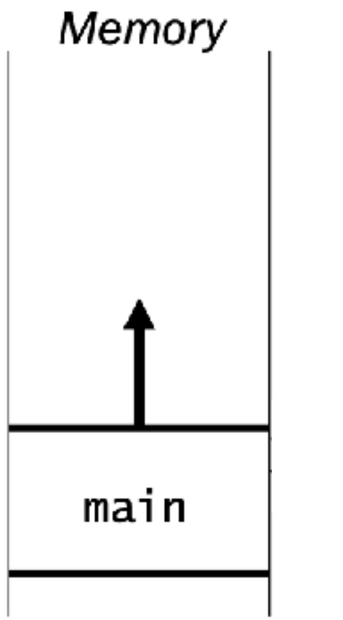
- Whenever a function *completes* (returns), the activation record is popped off the run-time stack

- 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

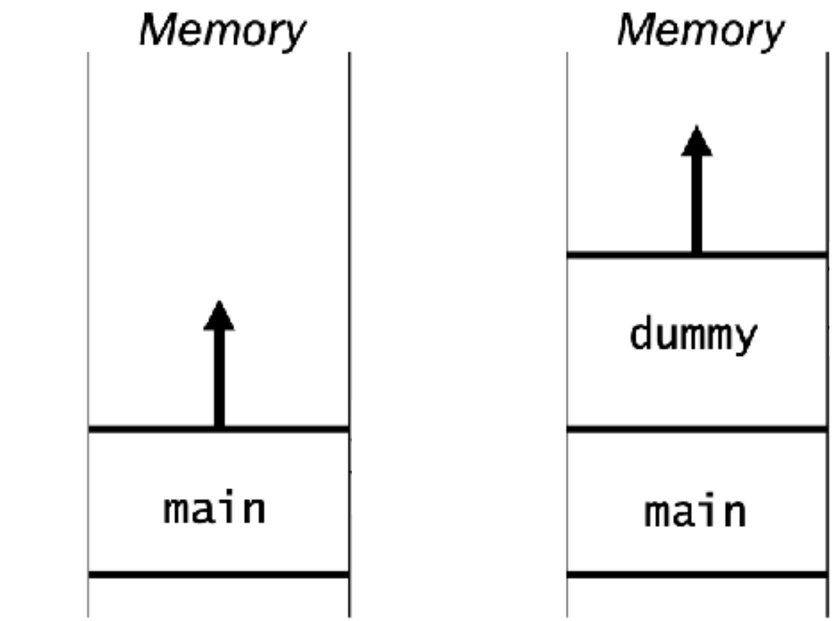
```
int dummy(int in1, int in2);

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

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



Before the dummy call

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;
  …
}
```

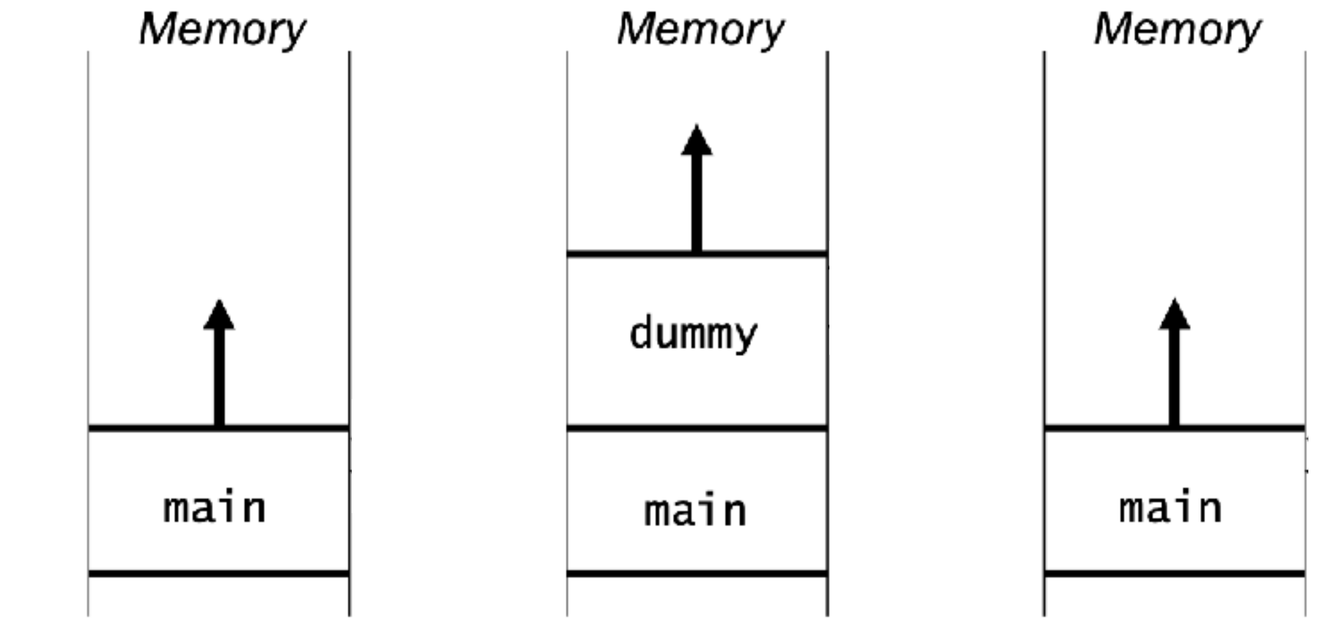


Before the dummy call

When dummy executes

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;
   …
}
```



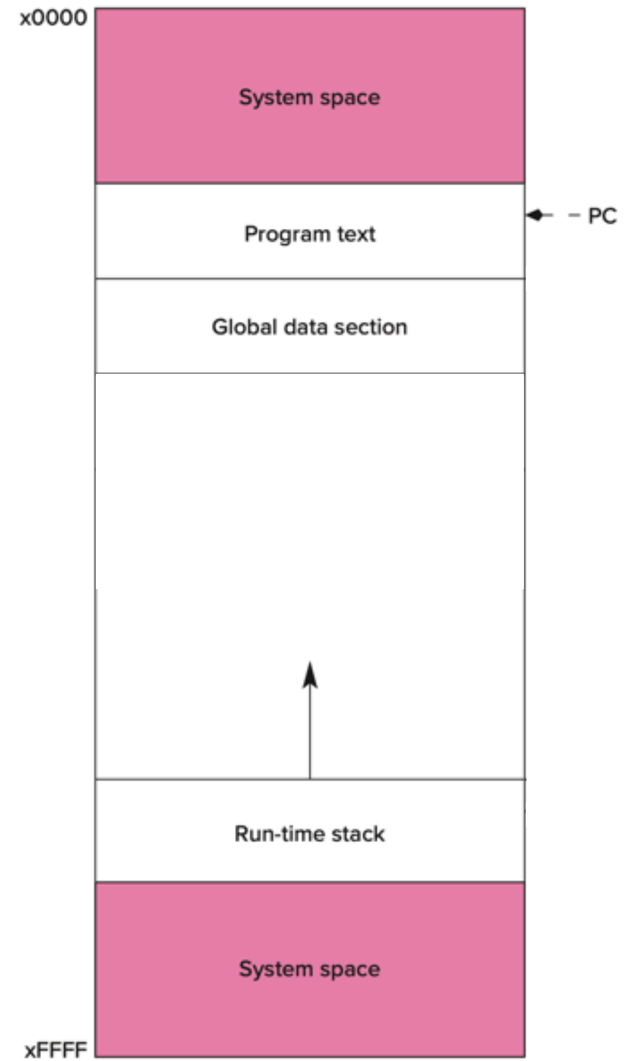
How to keep track?

How to keep track?

- Store pointers:

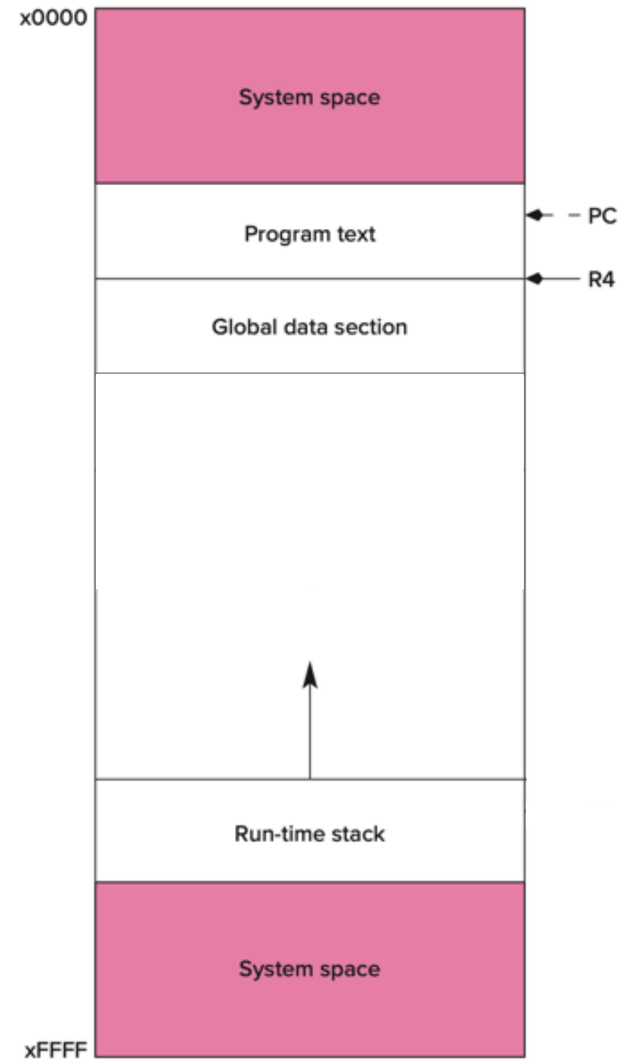
How to keep track?

- Store pointers:
  - Program counter - `PC`



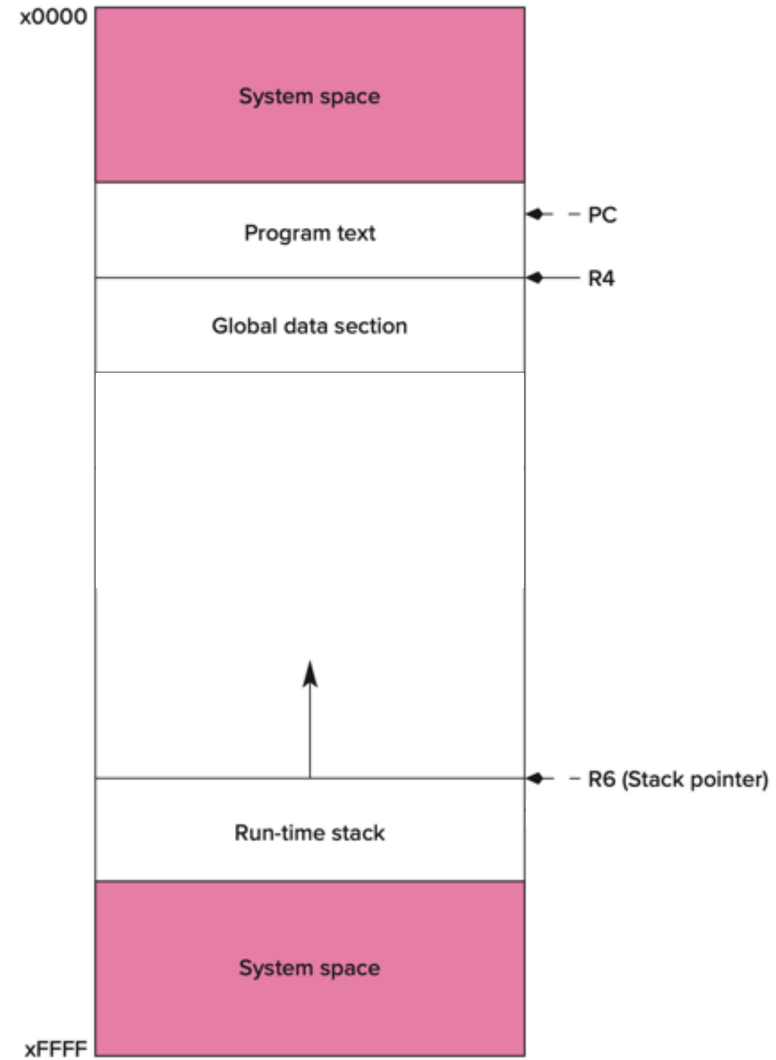
How to keep track?

- Store pointers:
  - Program counter - `PC`
  - **Global pointer** pointing to first global variable - `R4`



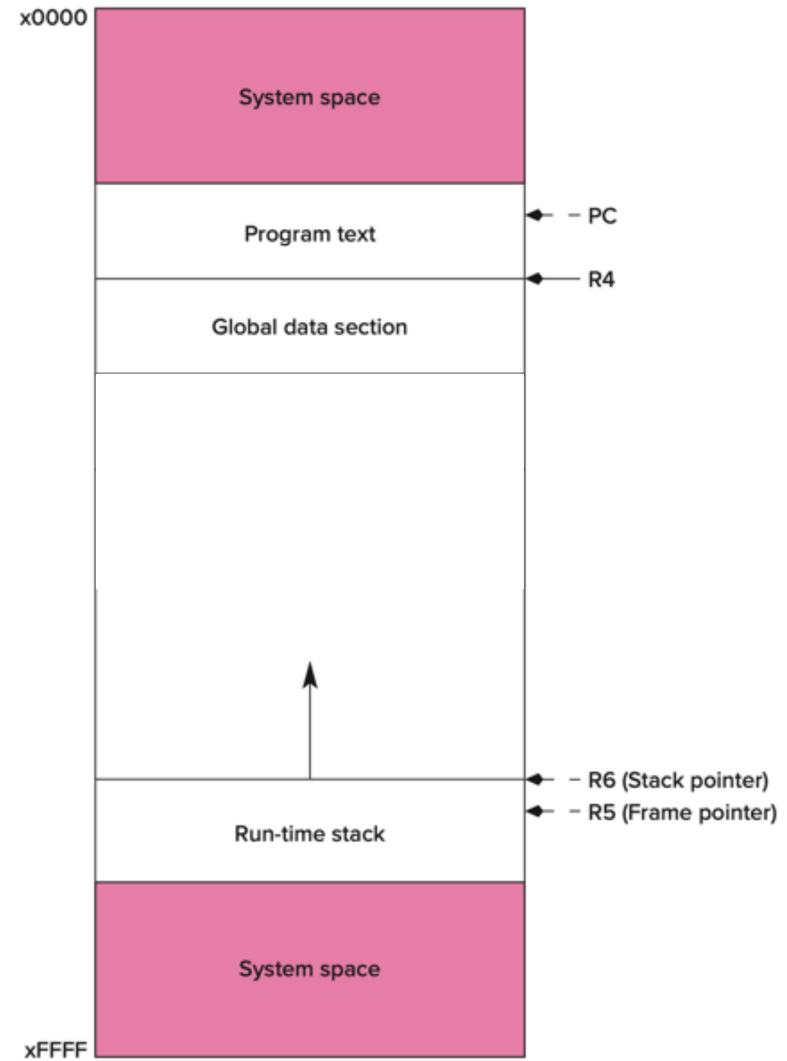
How to keep track?

- Store pointers:
  - Program counter - `PC`
  - **Global pointer** pointing to first global variable - `R4`
  - Top of stack, called **stack pointer** - `R6`



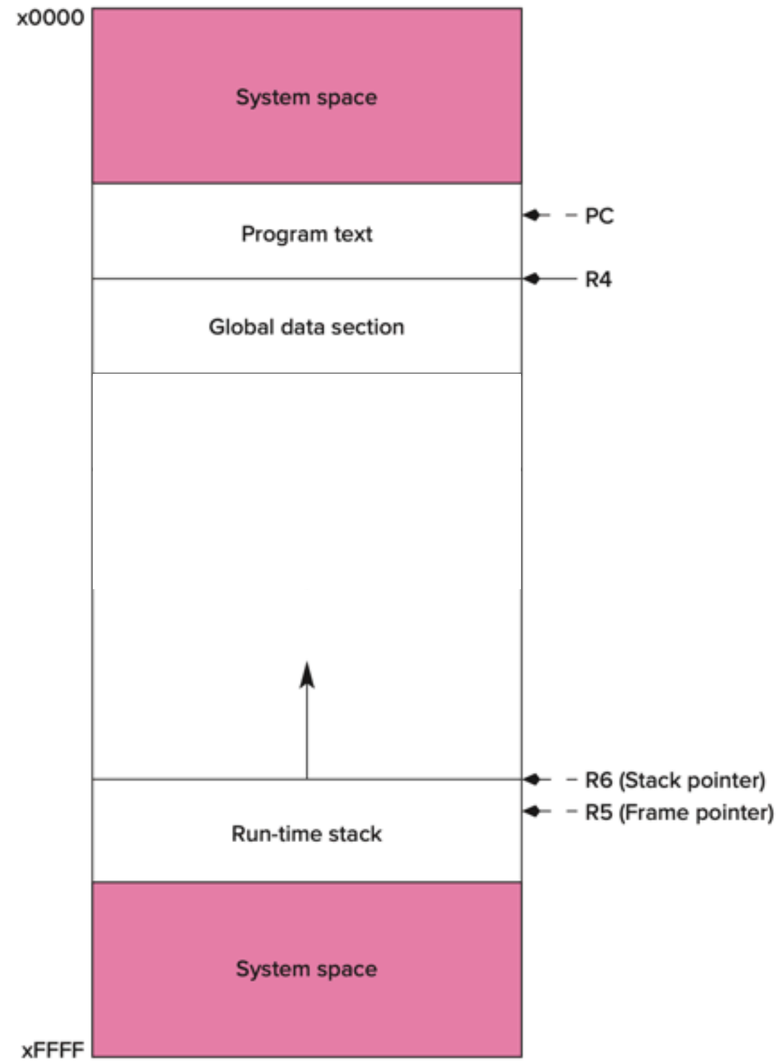
How to keep track?

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



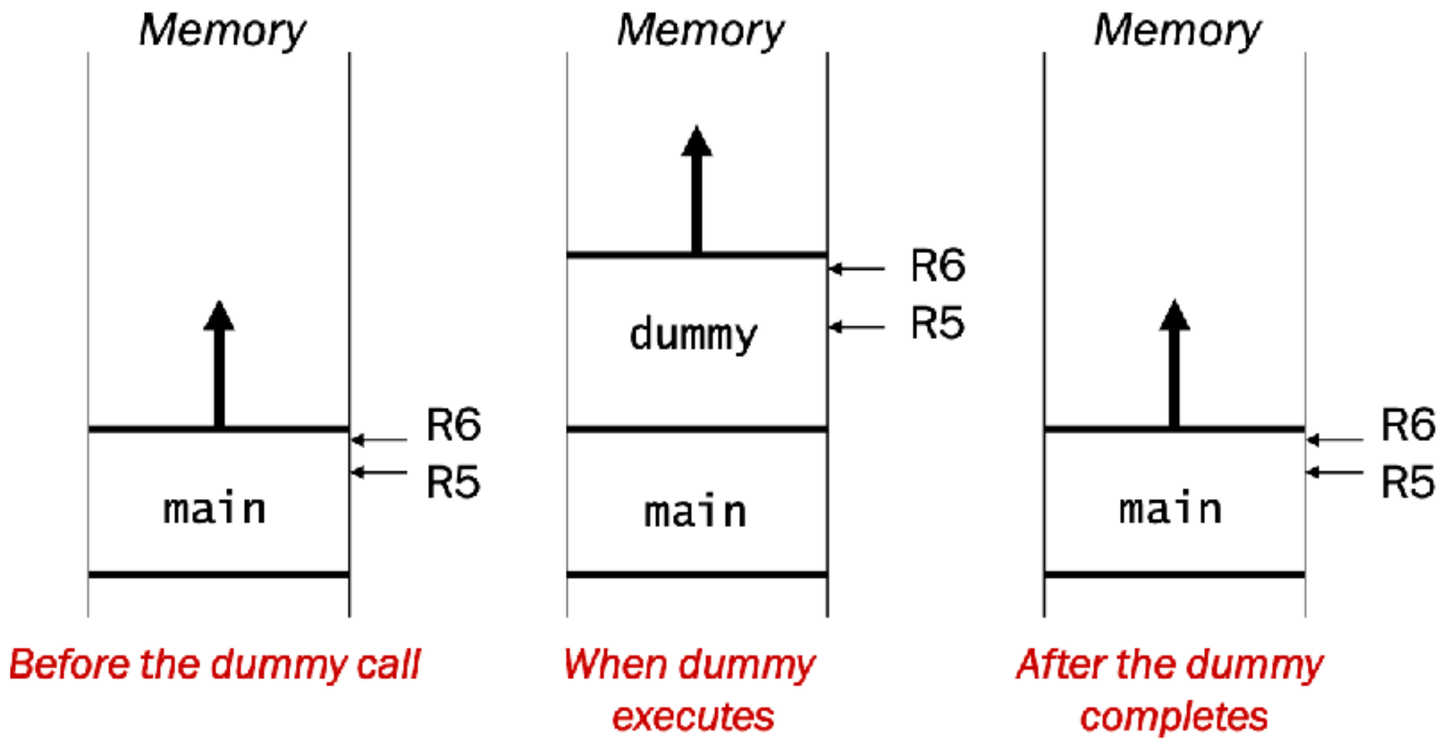
How to keep track?

- 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



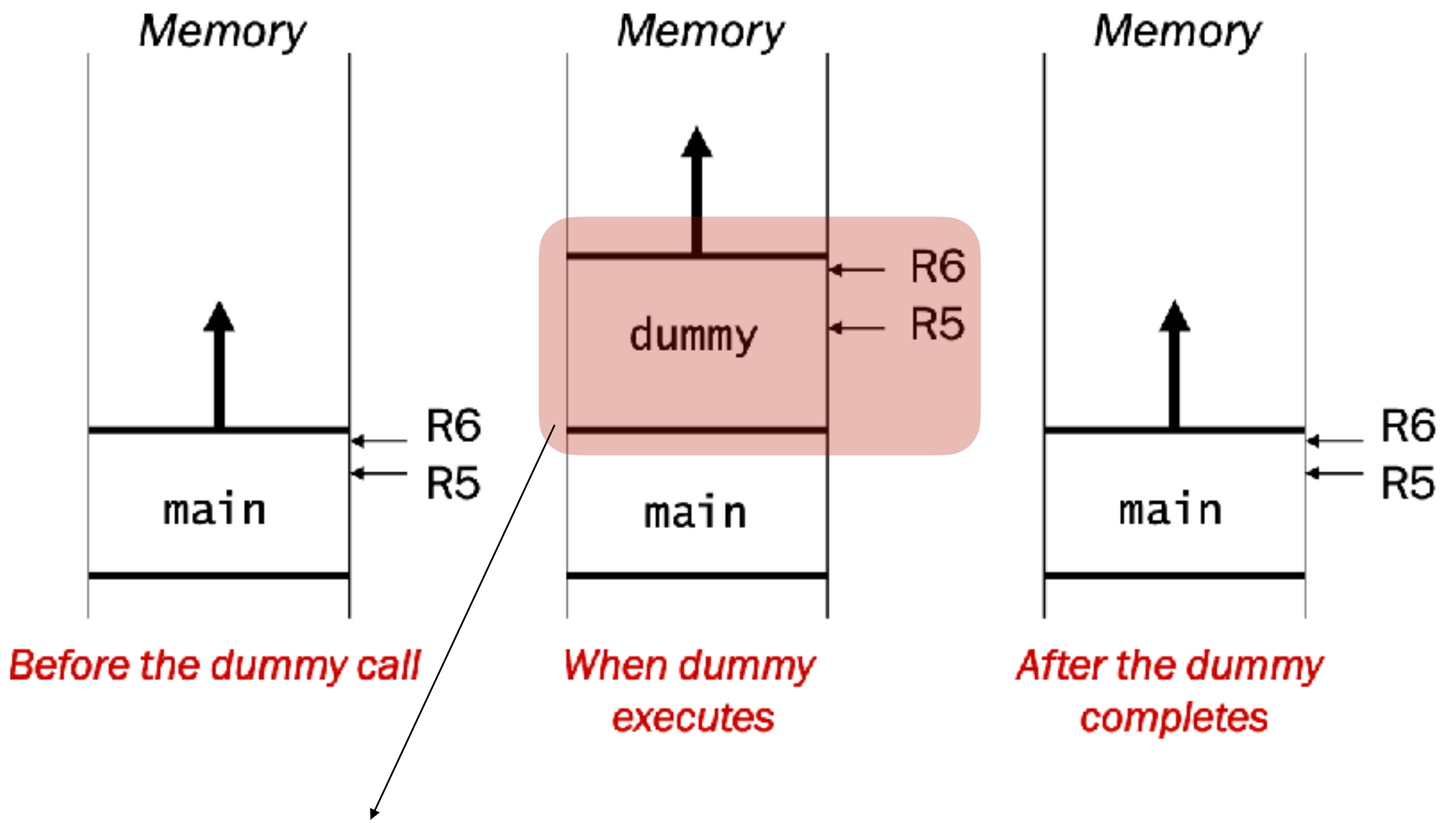
Example: function call

```
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;
  …
}
```



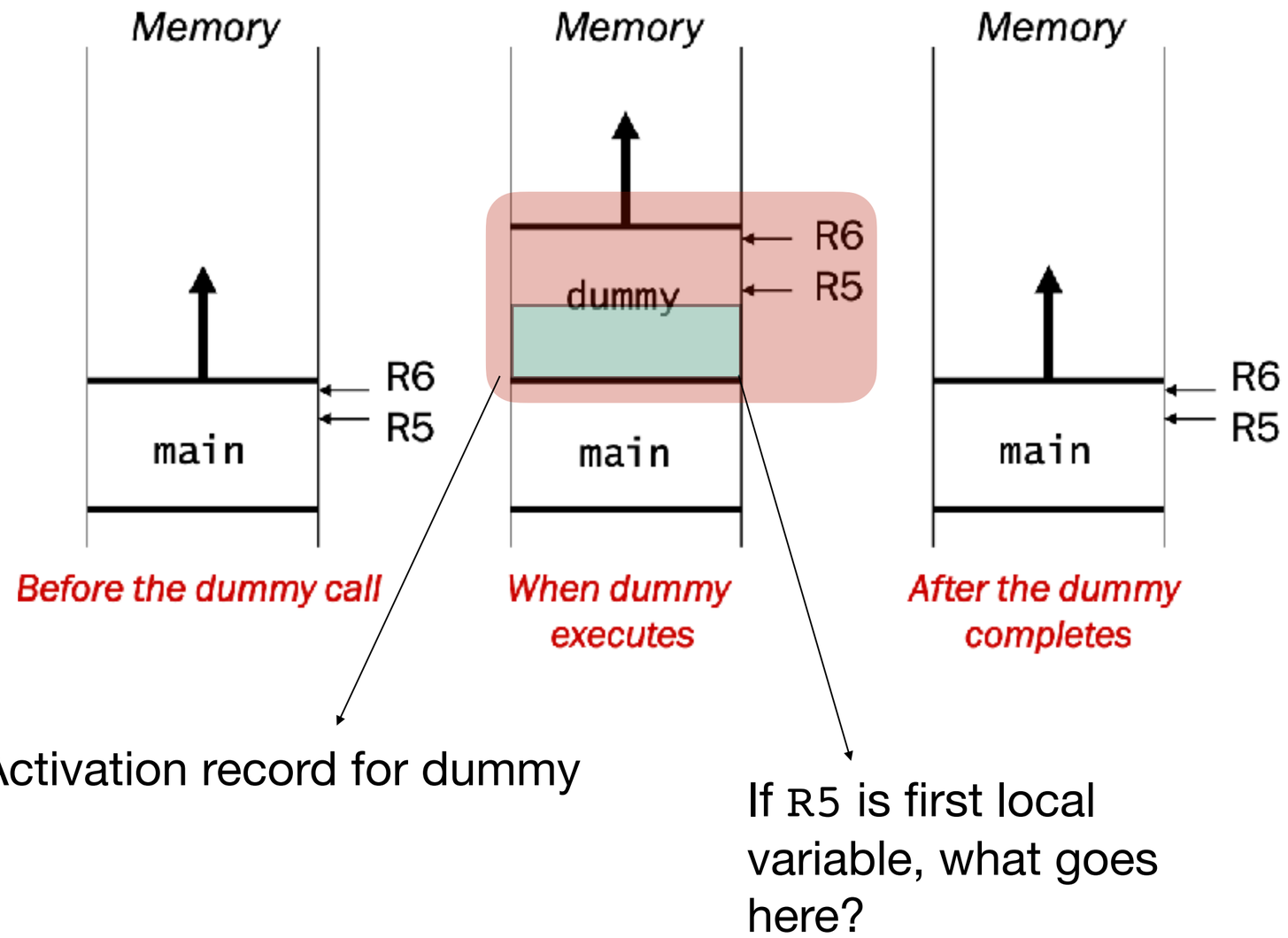
Example: function call

```
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;
  …
}
```



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;
  …
}
```



Activation record for dummy

If `R5` is first local variable, what goes here?

Details of a function call

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

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

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
  - Bookkeeping has to be done:

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
  - Bookkeeping has to be done:
    - **Return value**: Space for value returned by function according to type has to be allocated

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
  - 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

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
  - 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

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
  - 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

Details of a function call

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

Activation record

  - Arguments need to be passed around
  - 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

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
  - 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

Activation record

Pushed before local variables

Generating an activation record

Generating an activation record

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

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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)

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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`)

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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`)

Callee

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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`)

Callee

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

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Caller

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`)

Callee

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

Caller

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR/JSRR`)

Stack build up

Caller

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`)

Callee

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

Caller

Generating an activation record

1. *Caller* build-up: Push callee's arguments onto stack
2. Pass control to callee (`JSR`/`JSRR`)

Stack build up

Caller

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

Callee

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

Caller

Example function call

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;
}
```

Run-time stack

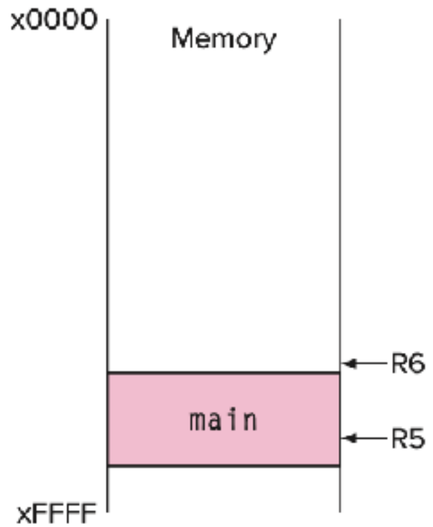
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

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

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;
}
```



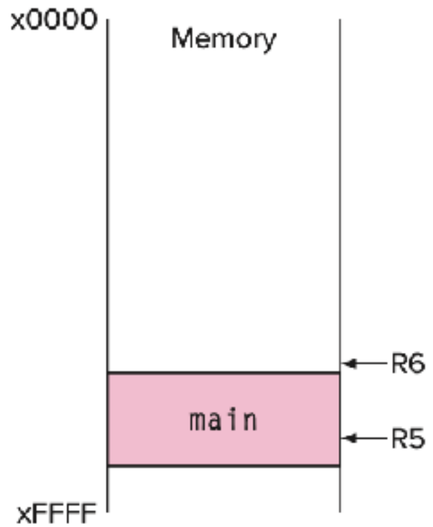
(a) Run-time stack when execution starts

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;
}
```



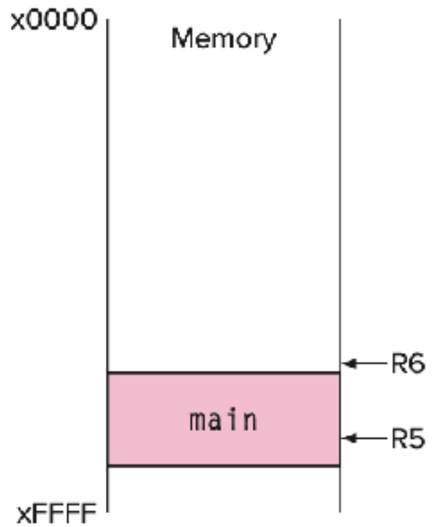
(a) Run-time stack when execution starts

Run-time stack

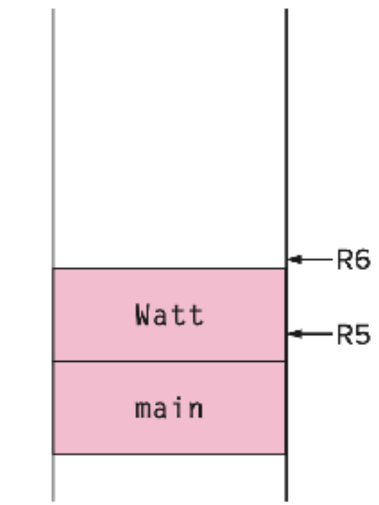
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

```
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



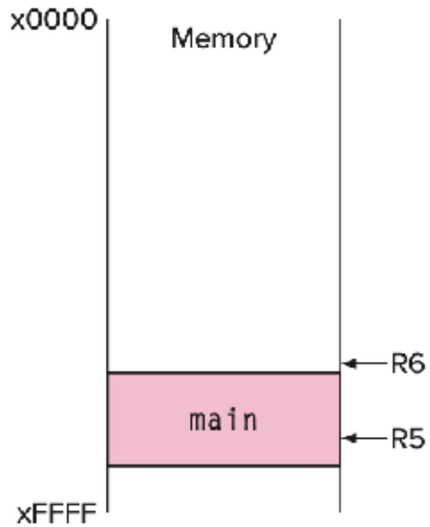
(b) When `Watt` executes

Run-time stack

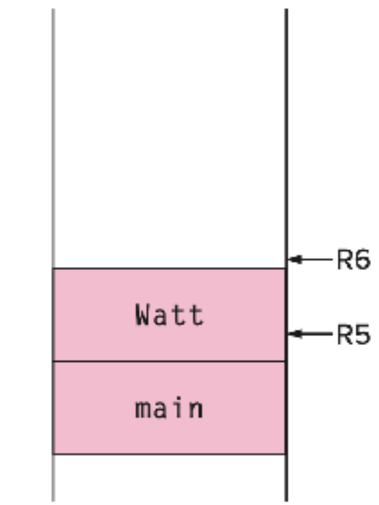
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

```
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



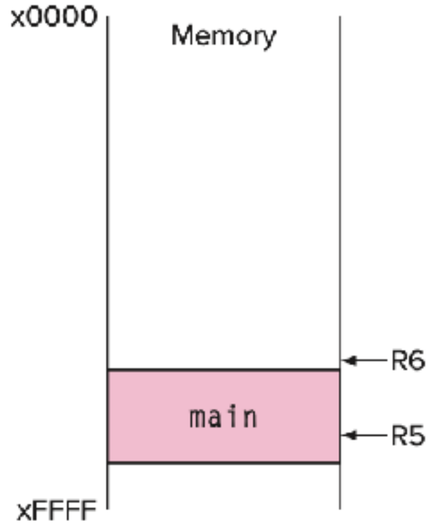
(b) When Watt executes

Run-time stack

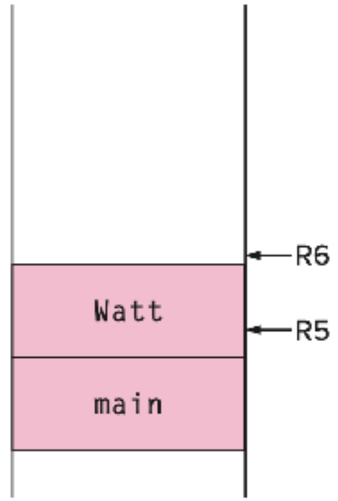
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

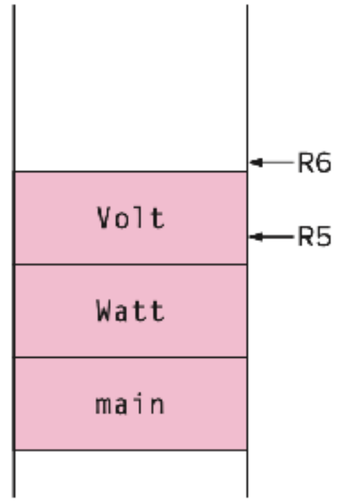
```
int Watt(int a) {
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```



(a) Run-time stack when execution starts



(b) When `Watt` executes



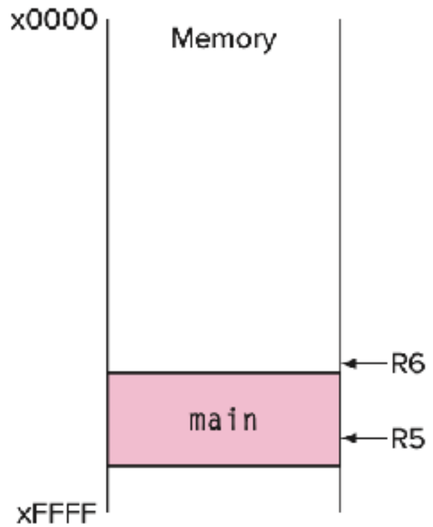
(c) When `Volt` executes

Run-time stack

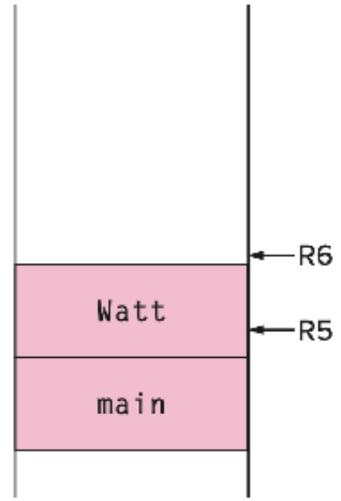
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

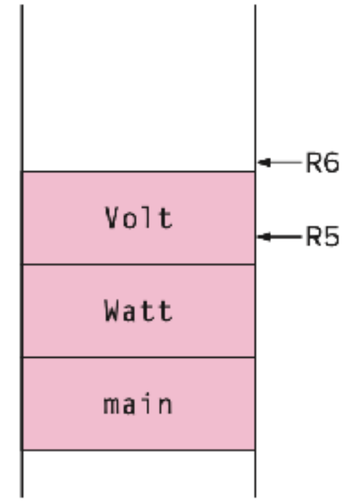
```
int Watt(int a) {
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```



(a) Run-time stack when execution starts



(b) When `Watt` executes



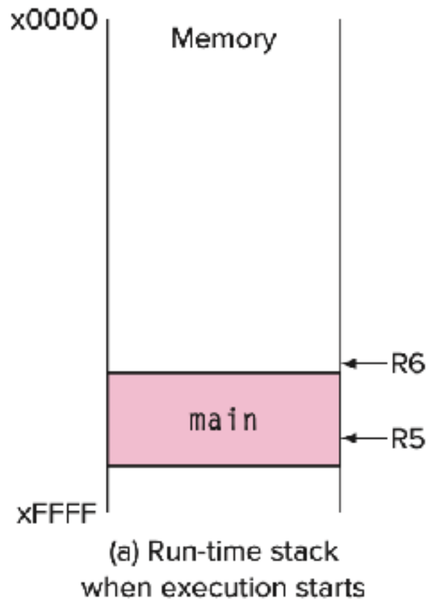
(c) When `Volt` executes

Run-time stack

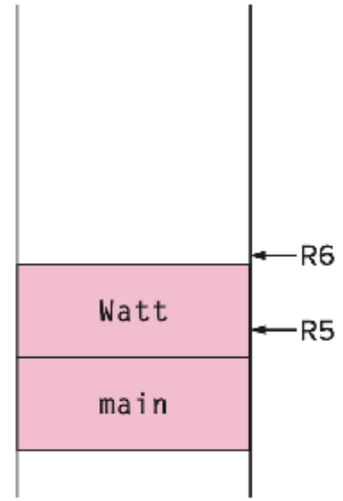
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

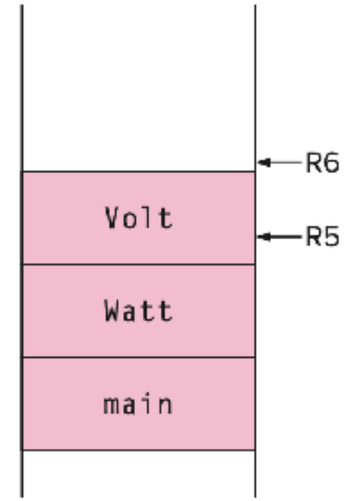
```
int Watt(int a) {
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```



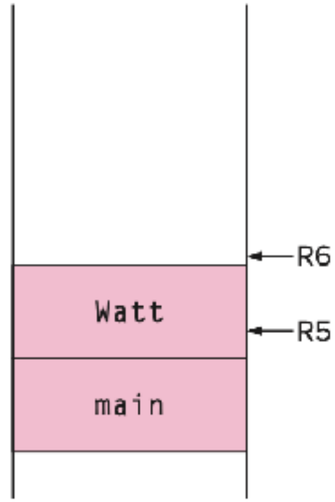
(a) Run-time stack when execution starts



(b) When `Watt` executes



(c) When `Volt` executes

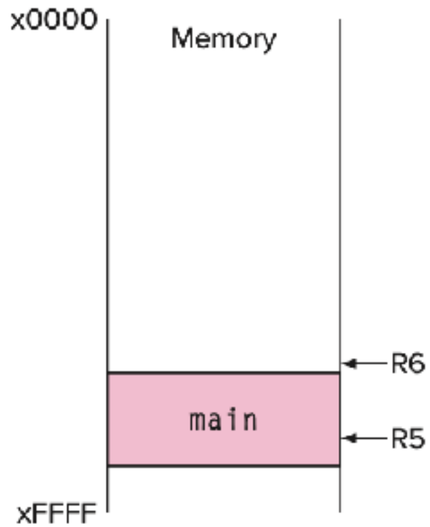


(d) After `Volt` completes

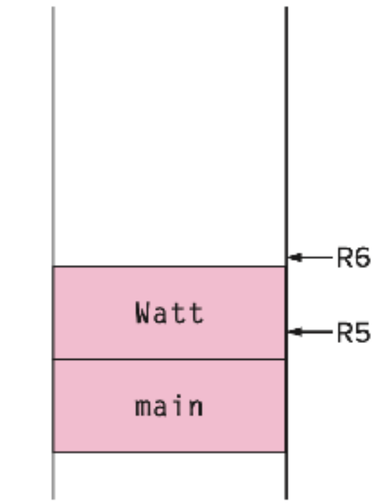
Run-time stack

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

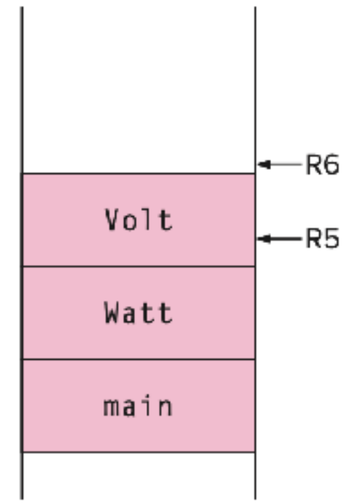
```
int Volt(int q, int r)
{
      int k;
      int m;
      ...
      return k;
}

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



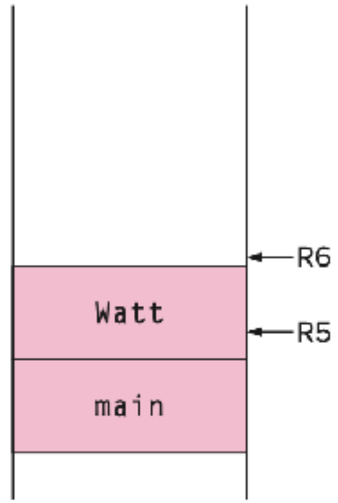
(a) Run-time stack when execution starts



(b) When `Watt` executes



(c) When `Volt` executes

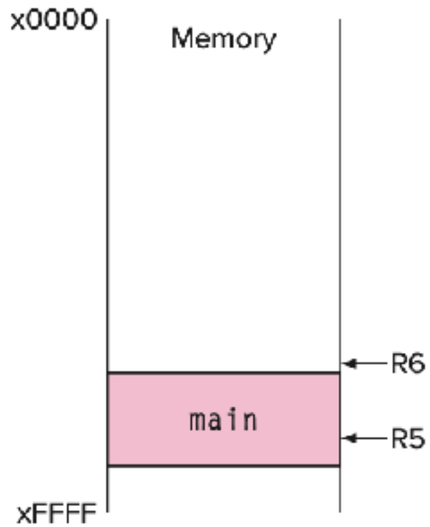


(d) After `Volt` completes

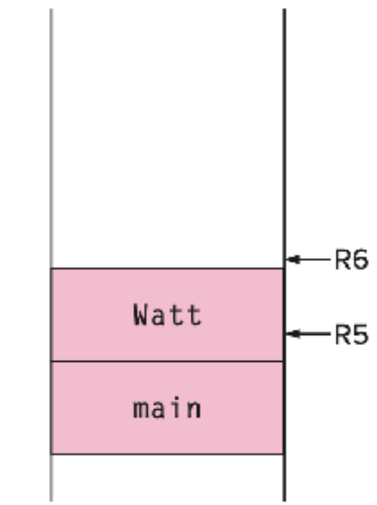
Run-time stack

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

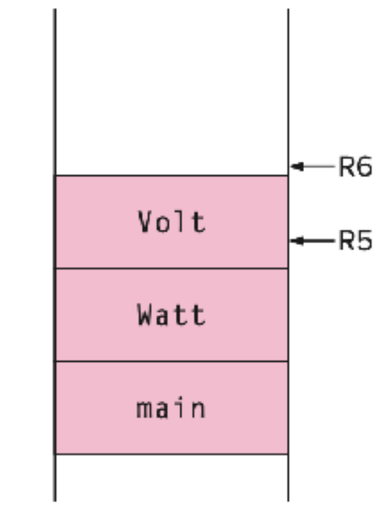
```
int Volt(int q, int r)
{
      int k;
      int m;
      ...
      return k;
}

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



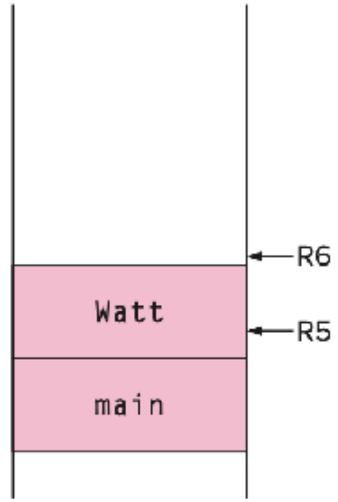
(a) Run-time stack when execution starts



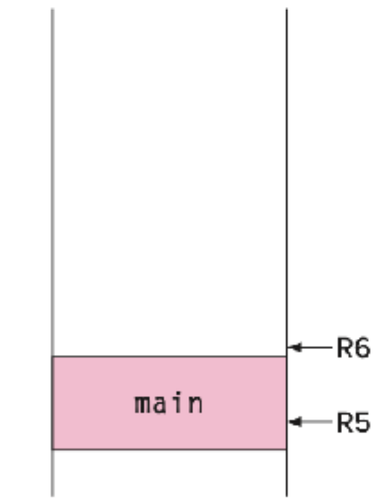
(b) When `Watt` executes



(c) When `Volt` executes



(d) After `Volt` completes



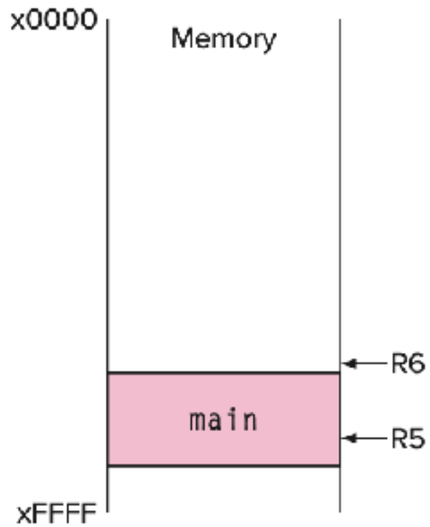
(e) After `Watt` completes

Run-time stack

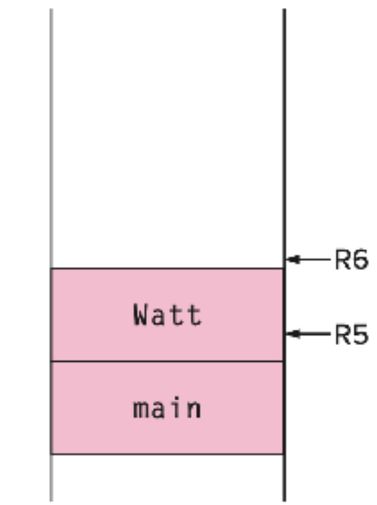
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

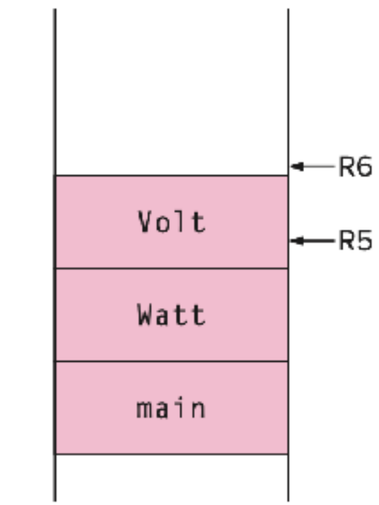
```
int Watt(int a) {
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```



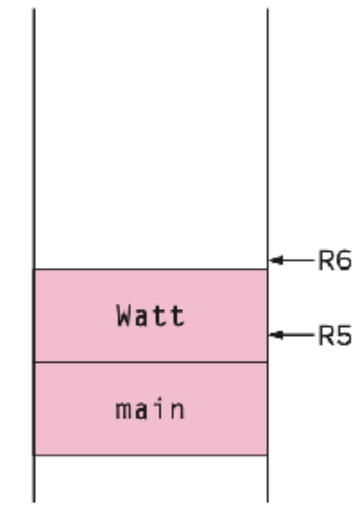
(a) Run-time stack when execution starts



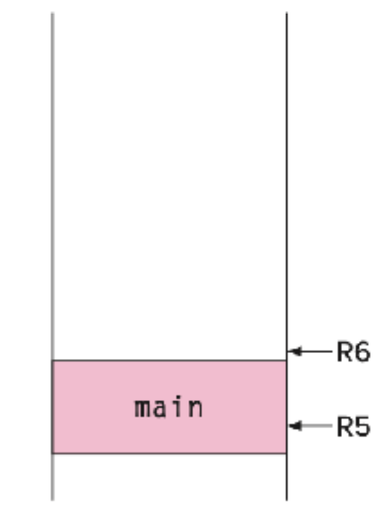
(b) When `Watt` executes



(c) When `Volt` executes



(d) After `Volt` completes



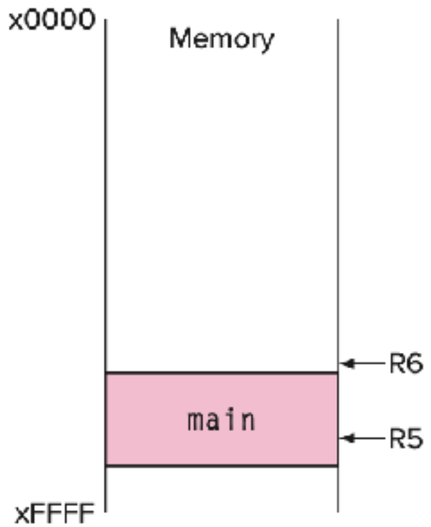
(e) After `Watt` completes

Run-time stack

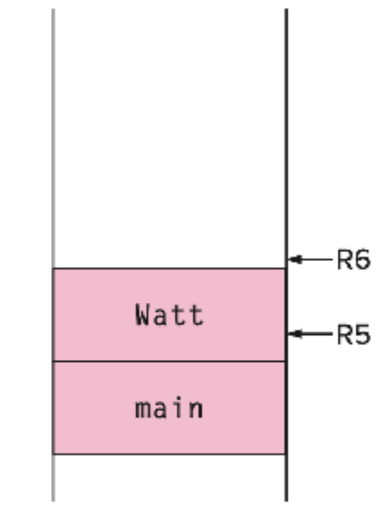
```
int main (void){
      int a;
      int b;
      …
      b = Watt(a);
      b = Volt(a, b);
}
```

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

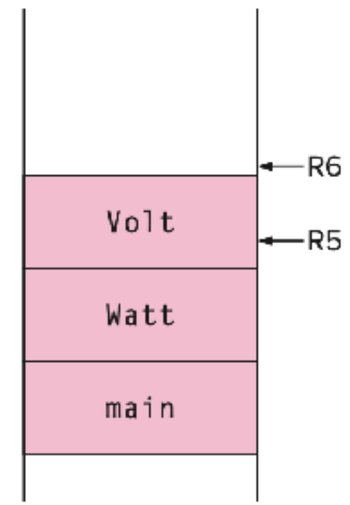
```
int Watt(int a) {
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```



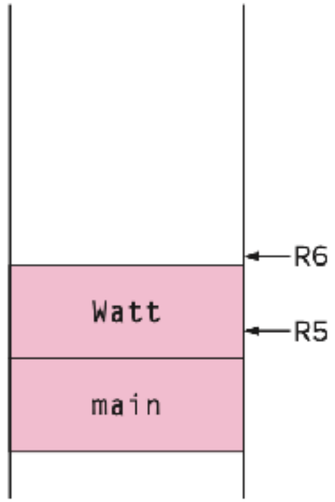
(a) Run-time stack when execution starts



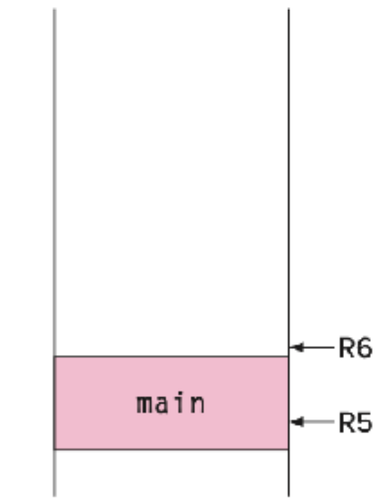
(b) When `Watt` executes



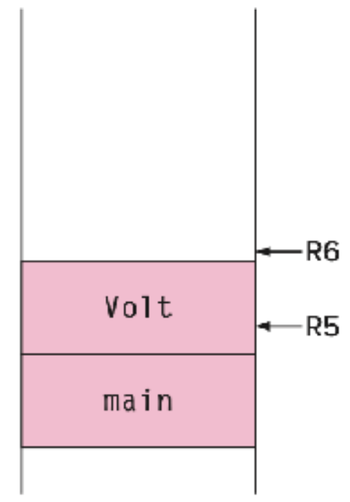
(c) When `Volt` executes



(d) After `Volt` completes



(e) After `Watt` completes



(f) When `Volt` executes

C Run-time stack protocol

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.

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.

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

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

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

C Run-time stack protocol

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

C Run-time stack protocol

- `Volt` called with two arguments

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

C Run-time stack protocol

- `Volt` called with two arguments
- Value *returned* by `Volt` is assigned to local integer variable `w`

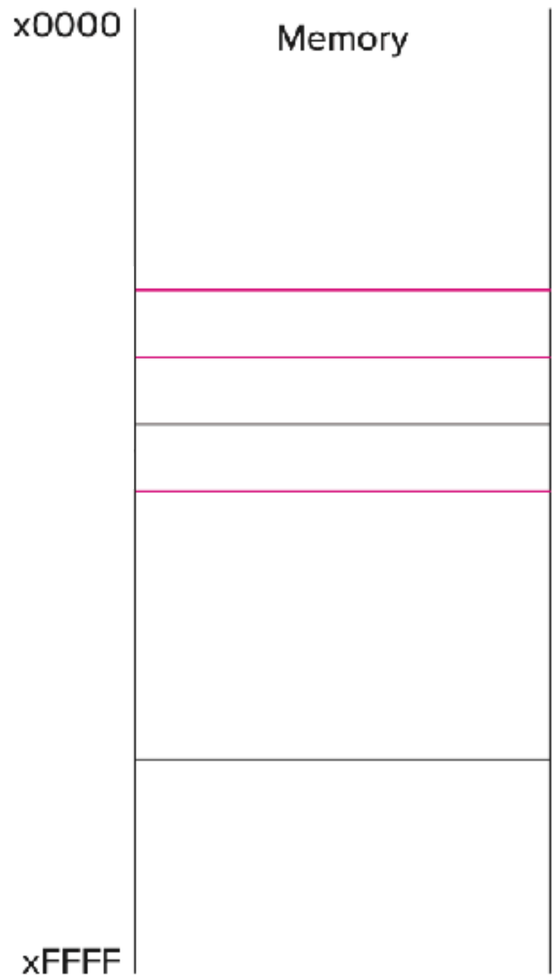
```
int Watt(int a)
{
     int w;
     ...
     w = Volt(w,10);
     …
     return w;
}
```

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

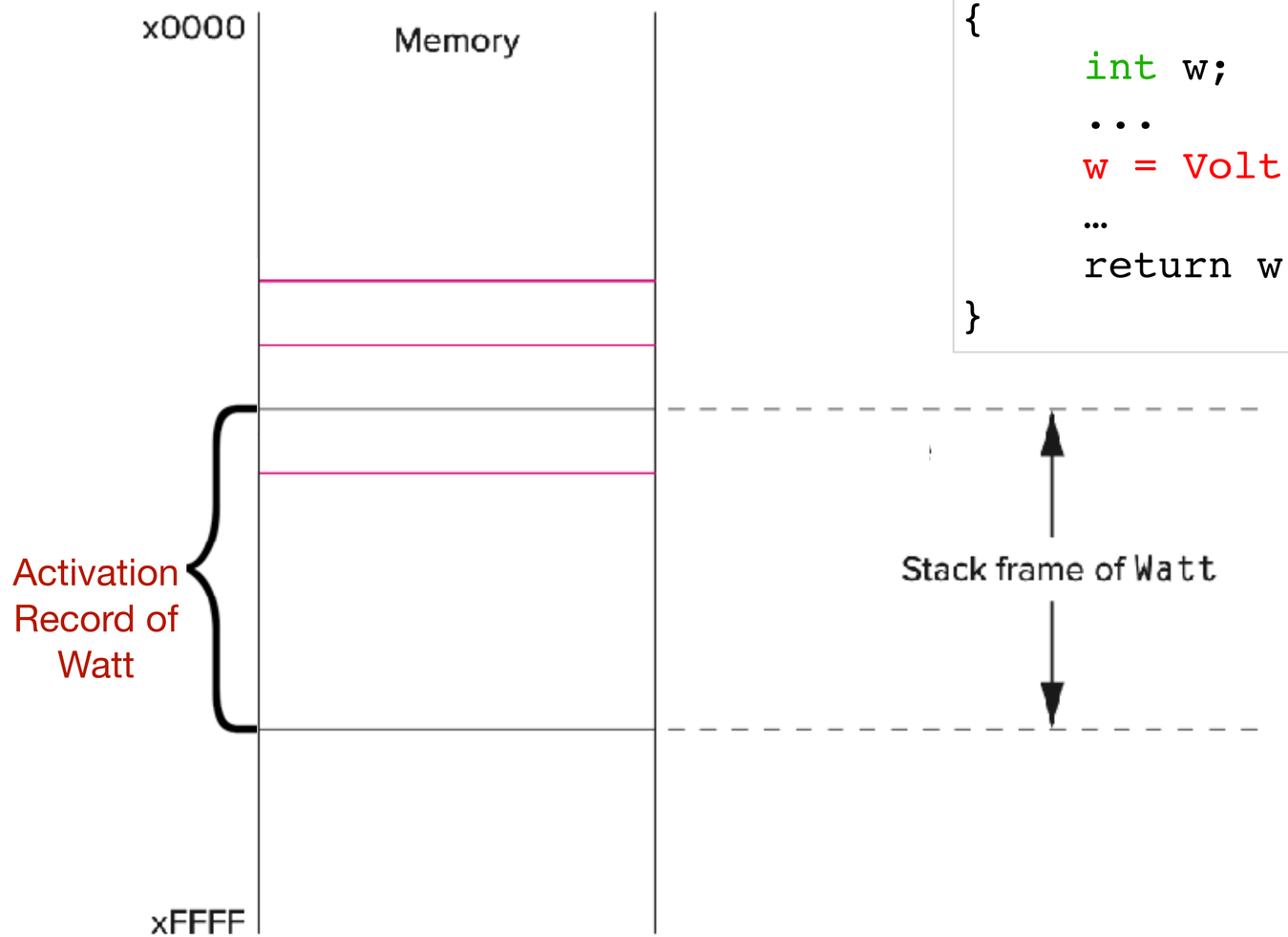
```
int Watt(int a)
{
     int w;
     ...
     w = Volt(w,10);
     …
     return w;
}
```

LC-3 Implementation



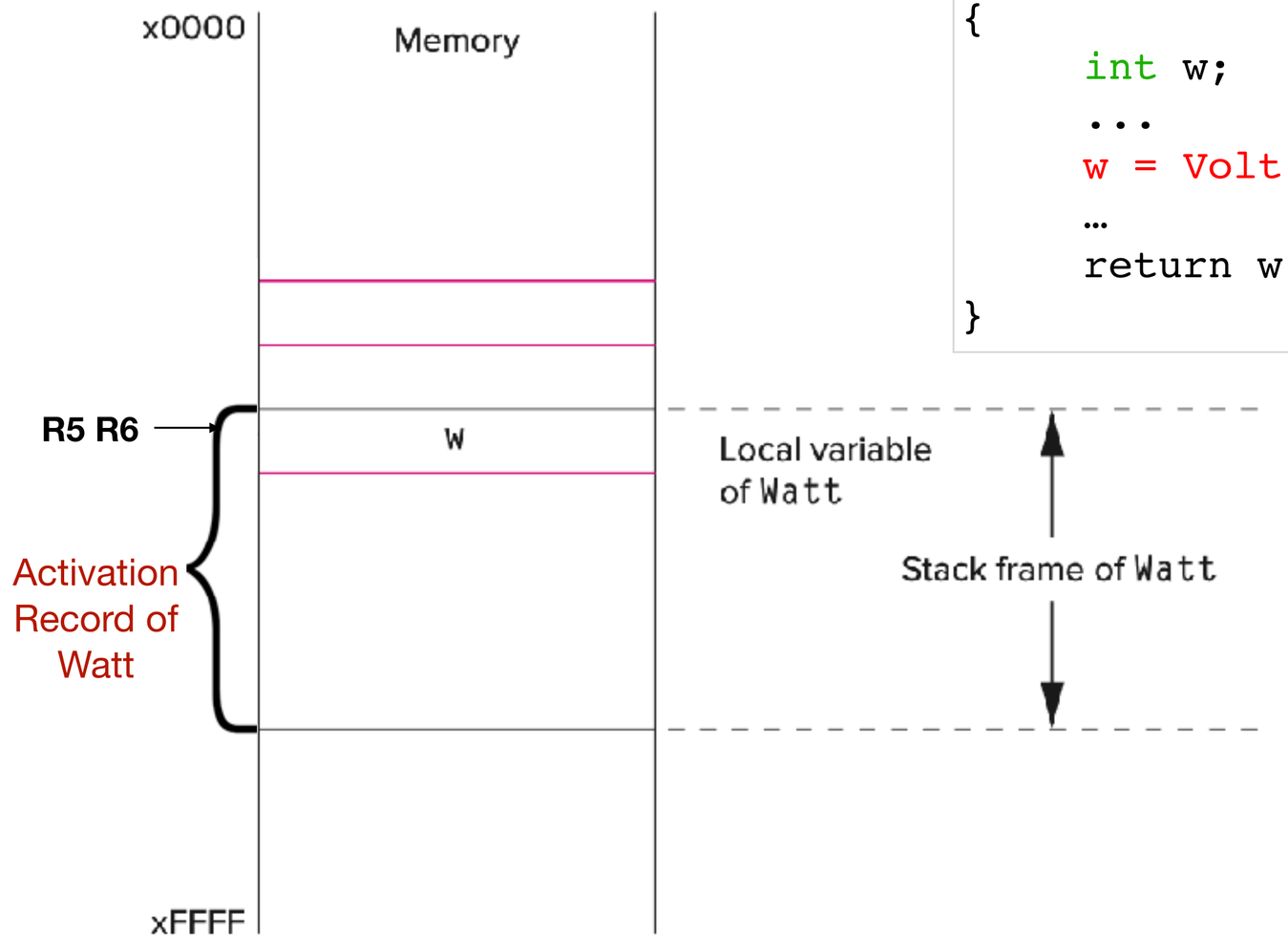

```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation




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

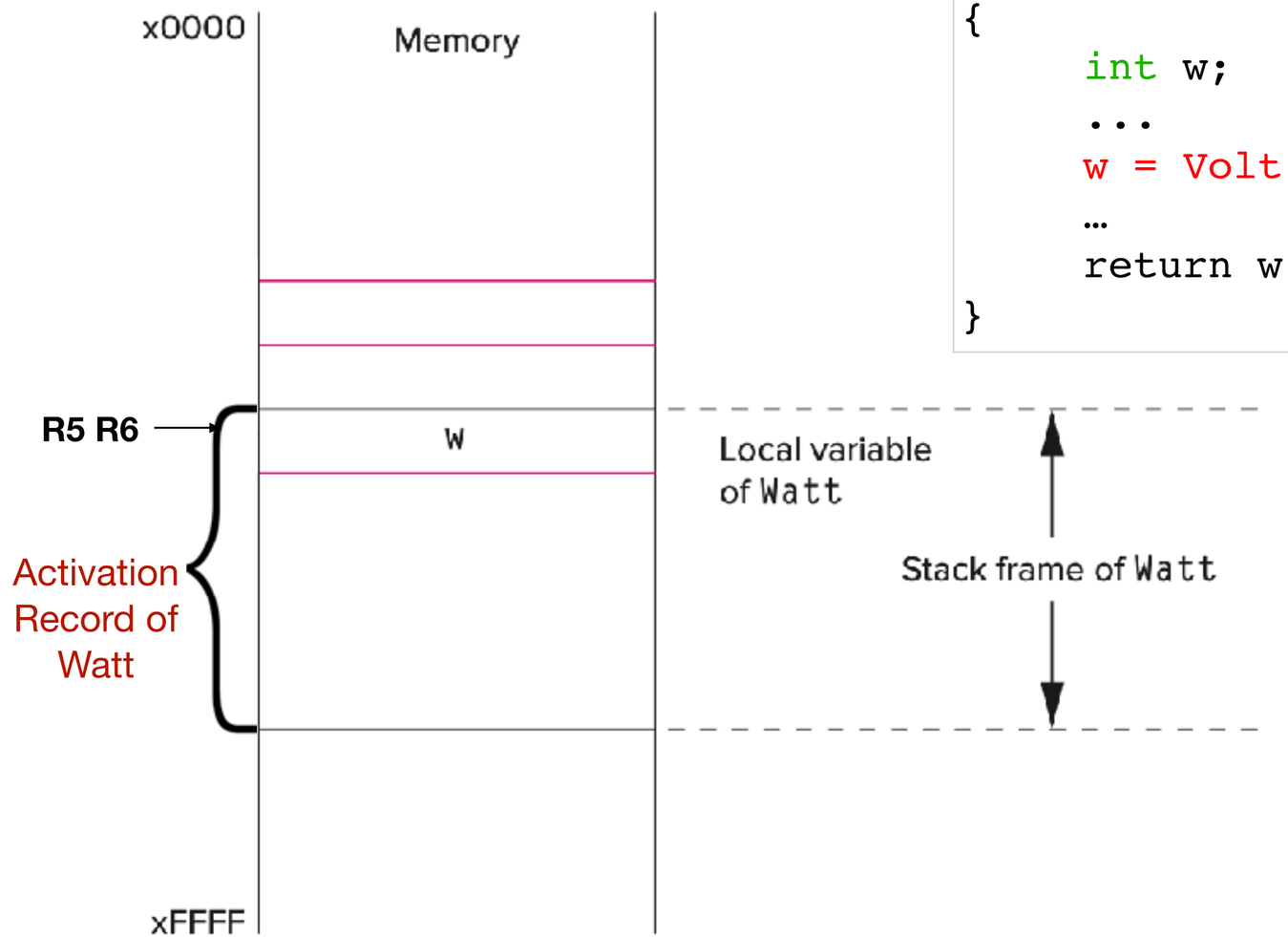
LC-3 Implementation




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

LC-3 Implementation

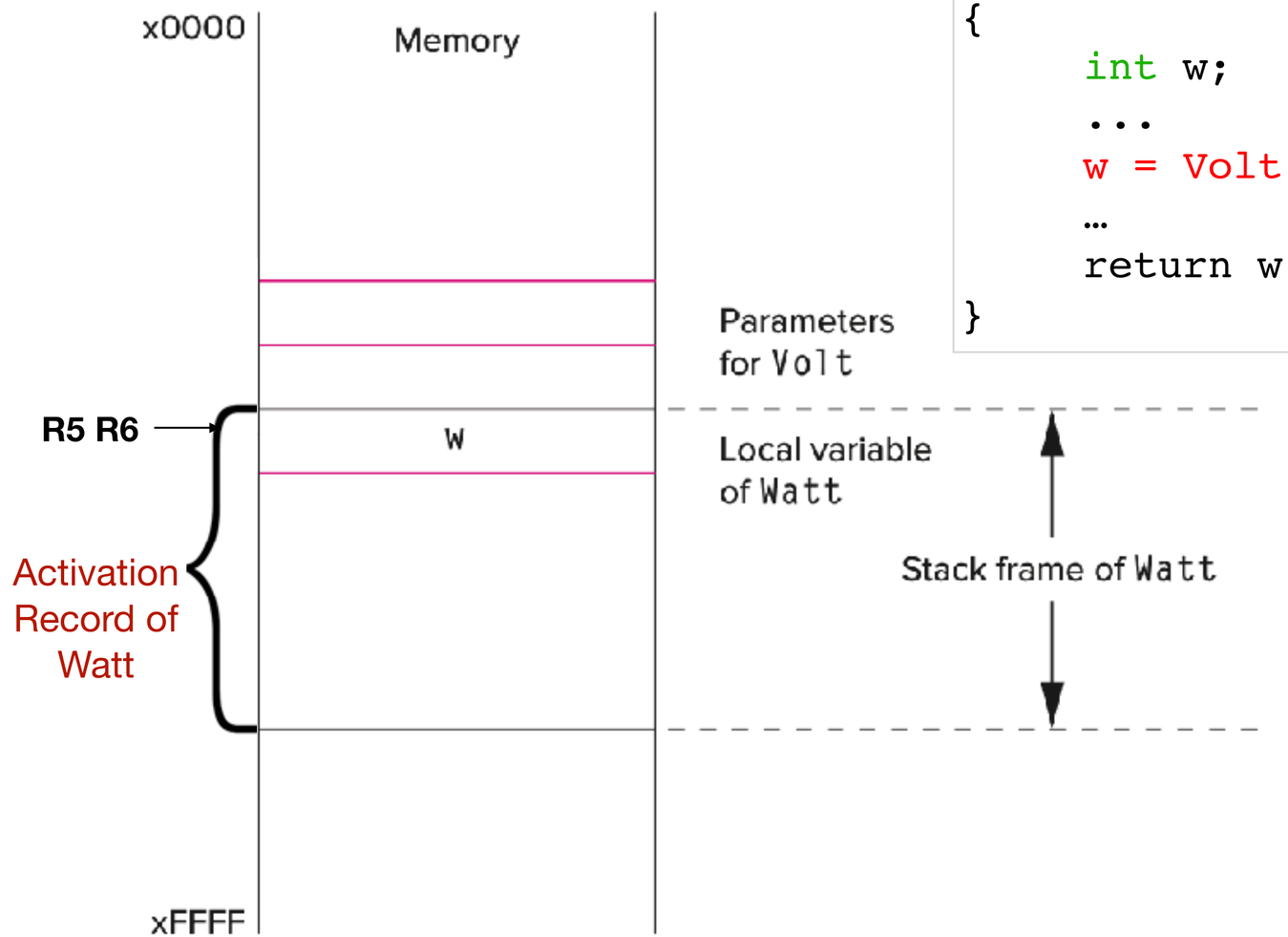
1. Caller setup (push callee's arguments onto stack)




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

LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)

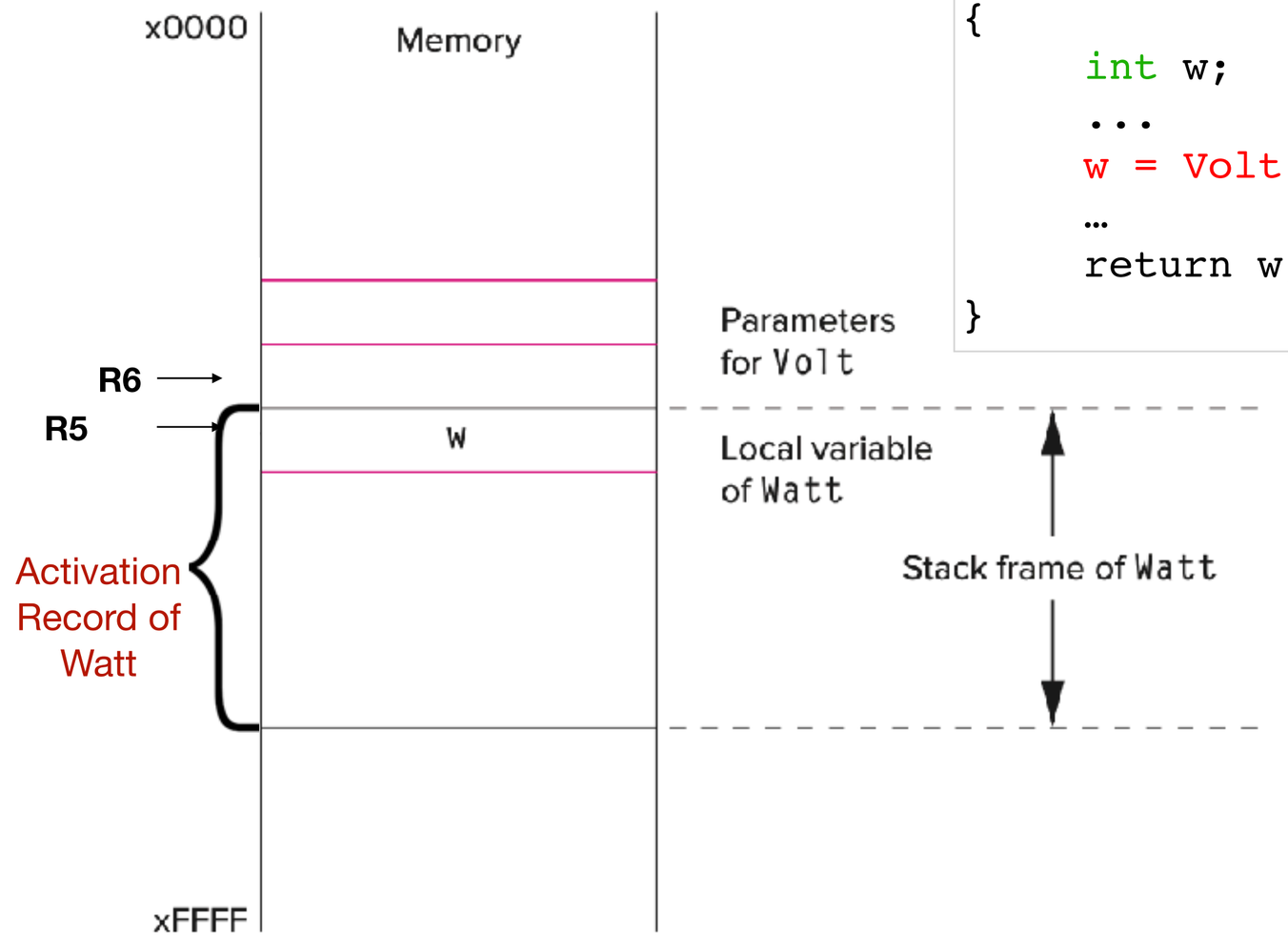



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)

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

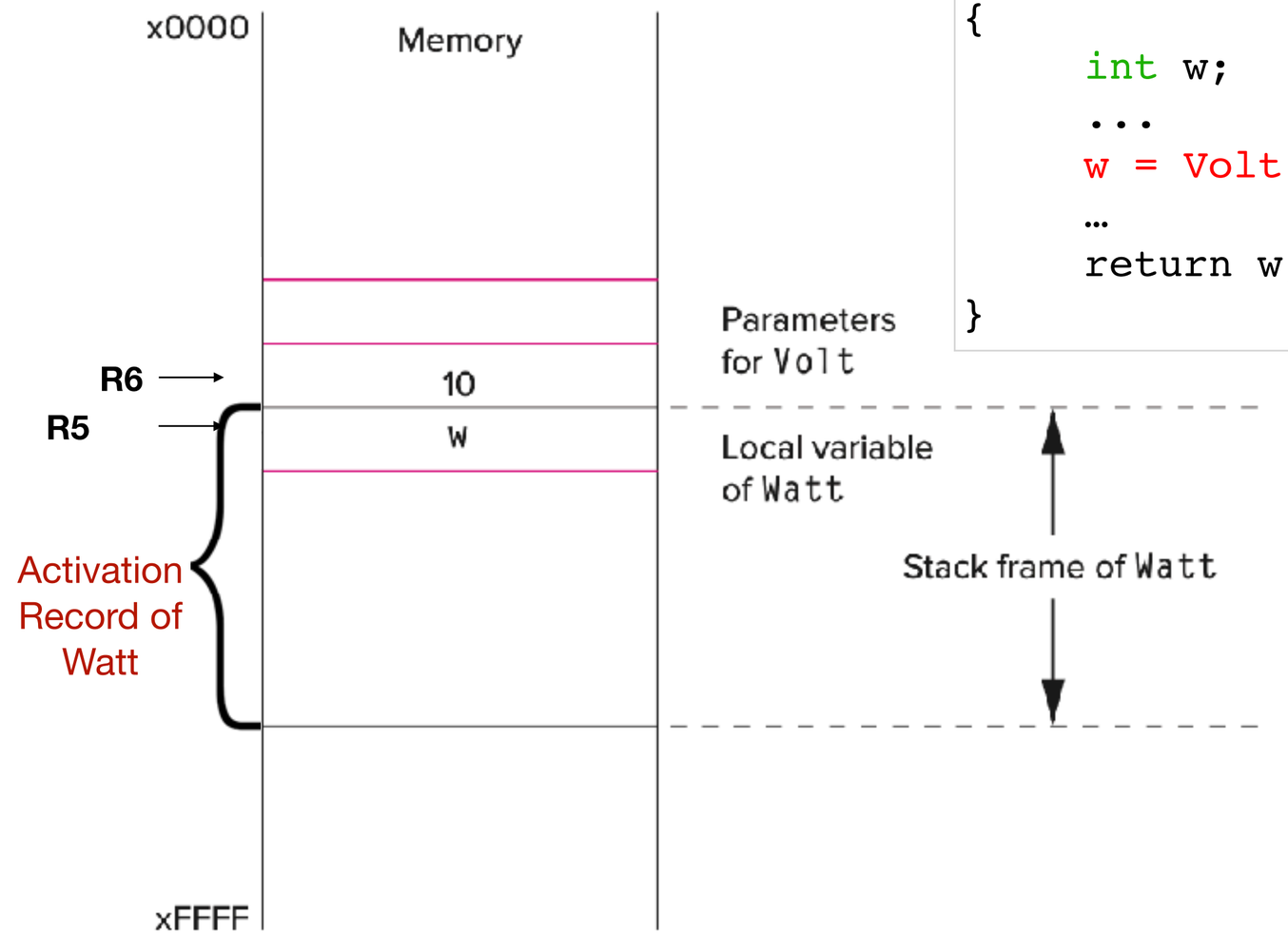



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)

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

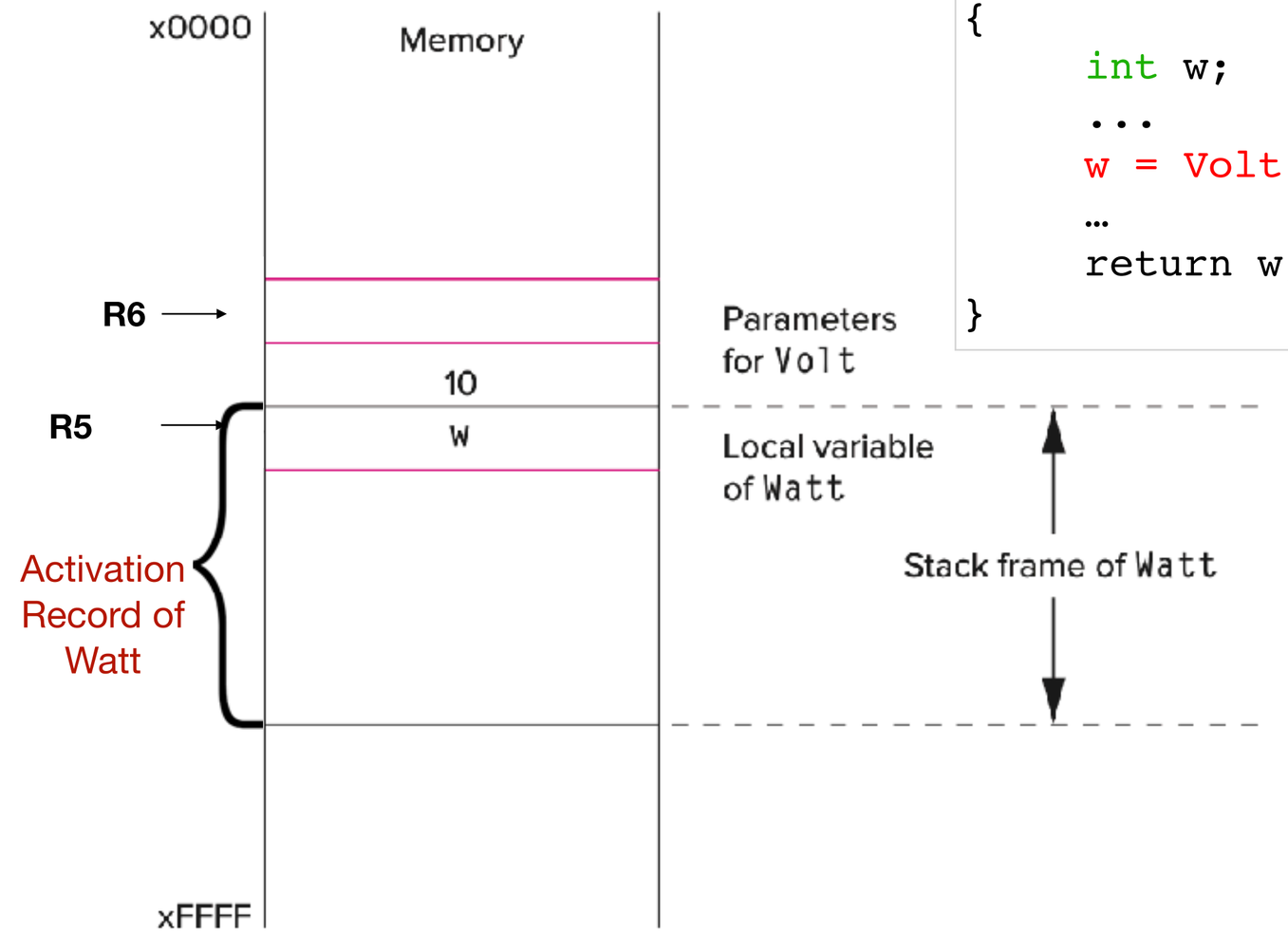



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)

```
; 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
```

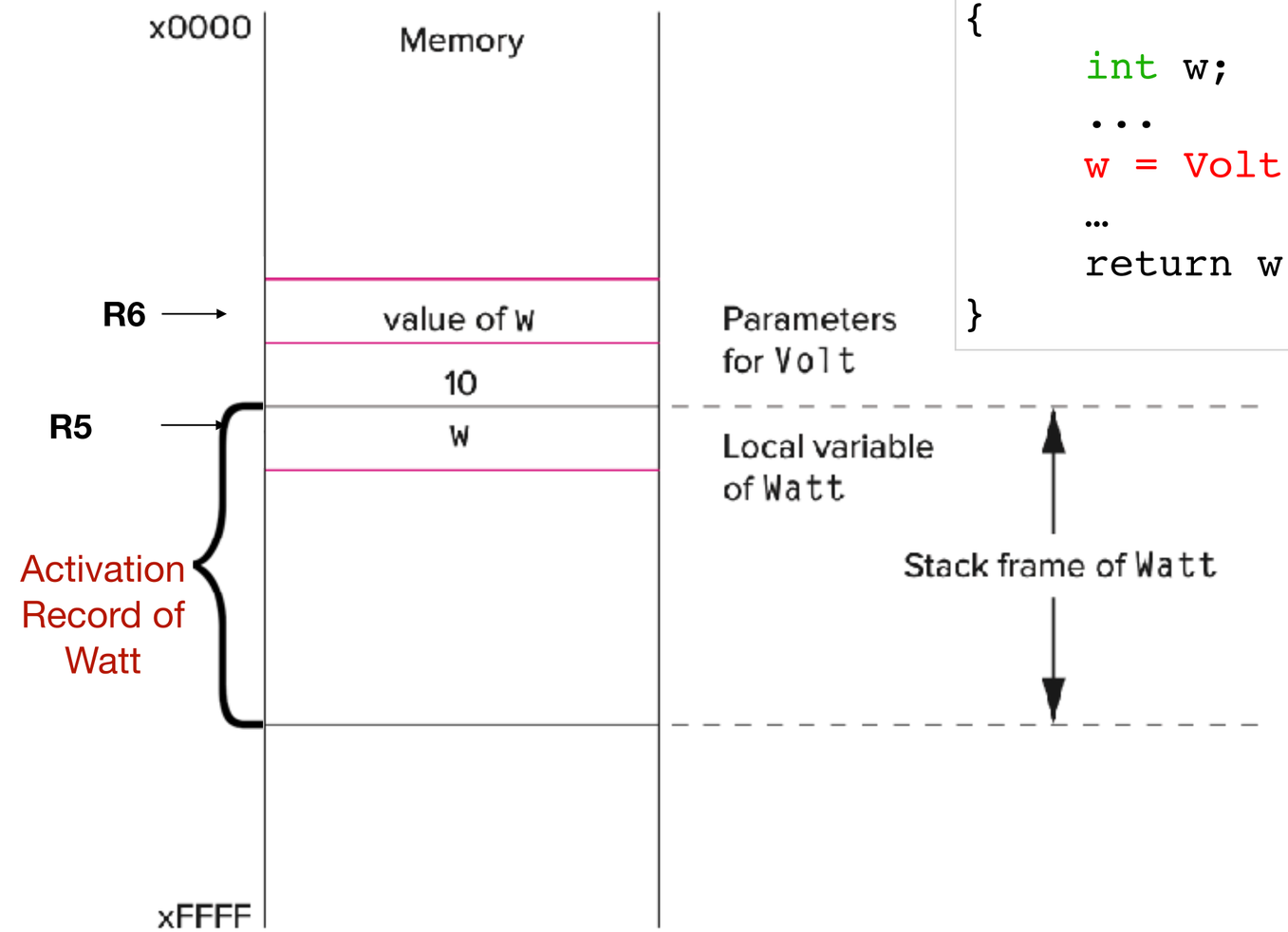



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation

1. Caller setup (push callee's arguments onto stack)

```
; 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
```

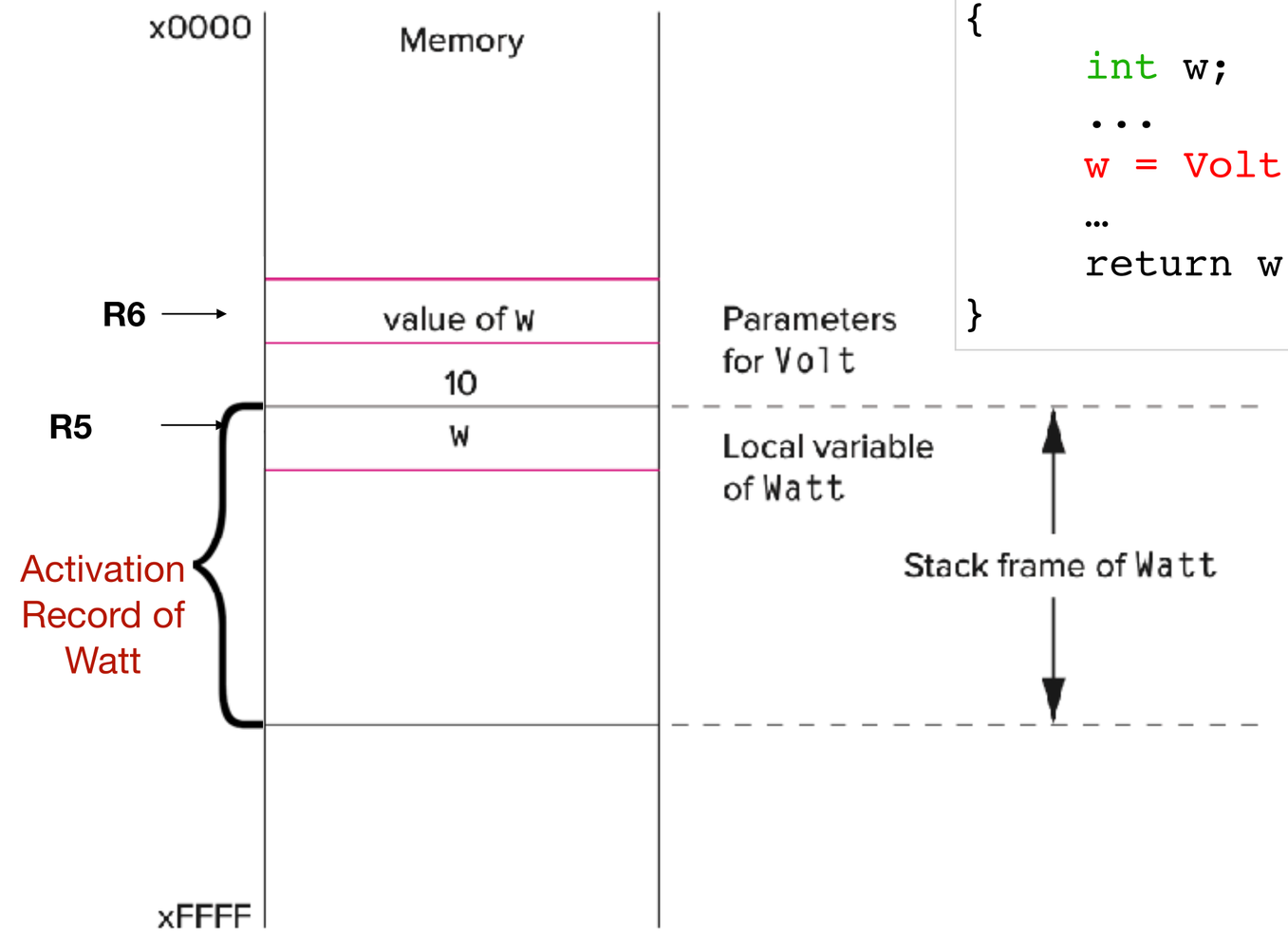



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

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
```

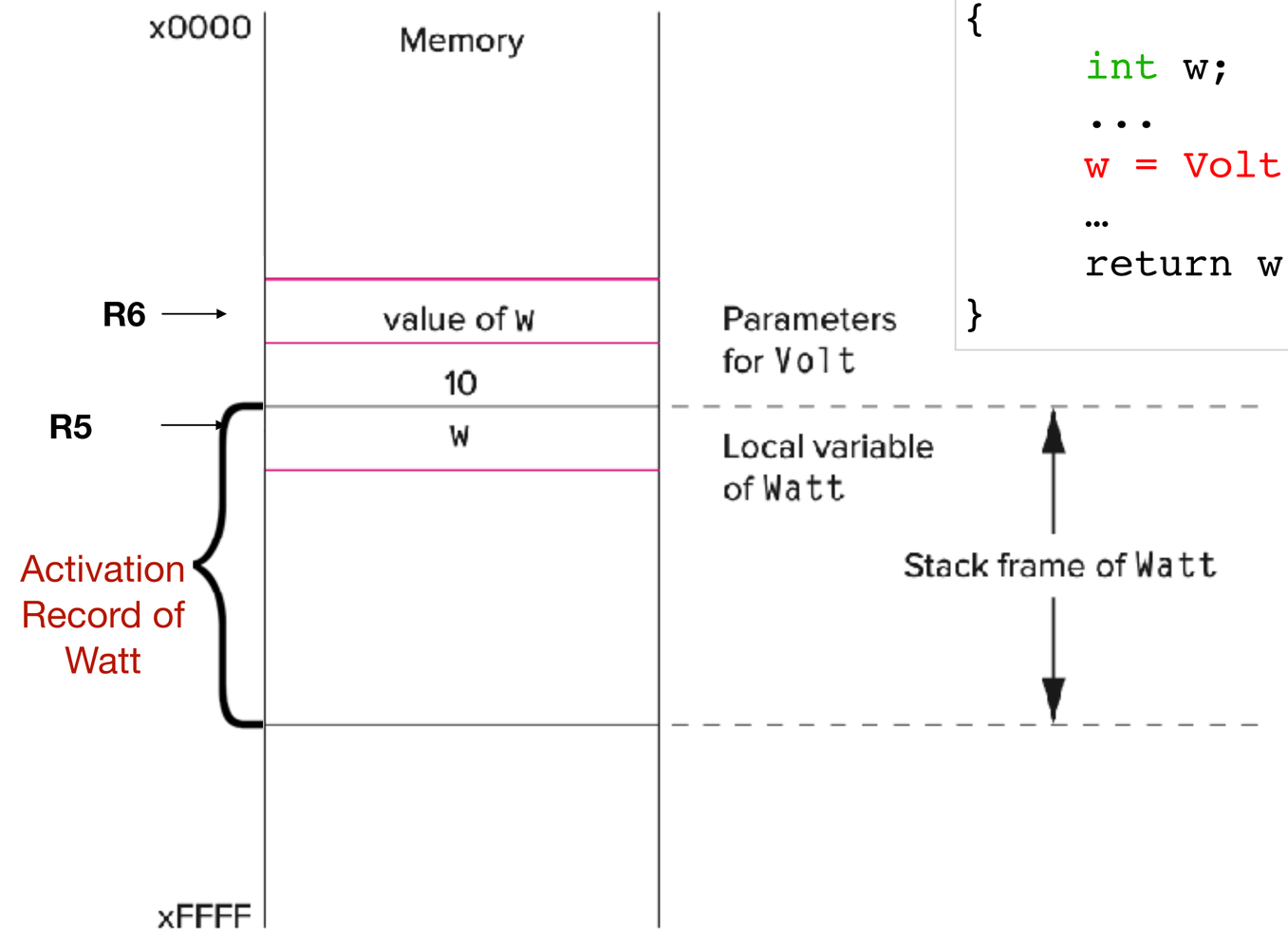



```
int Watt(int a)
{
      int w;
      ...
      w = Volt(w,10);
      …
      return w;
}
```

LC-3 Implementation

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

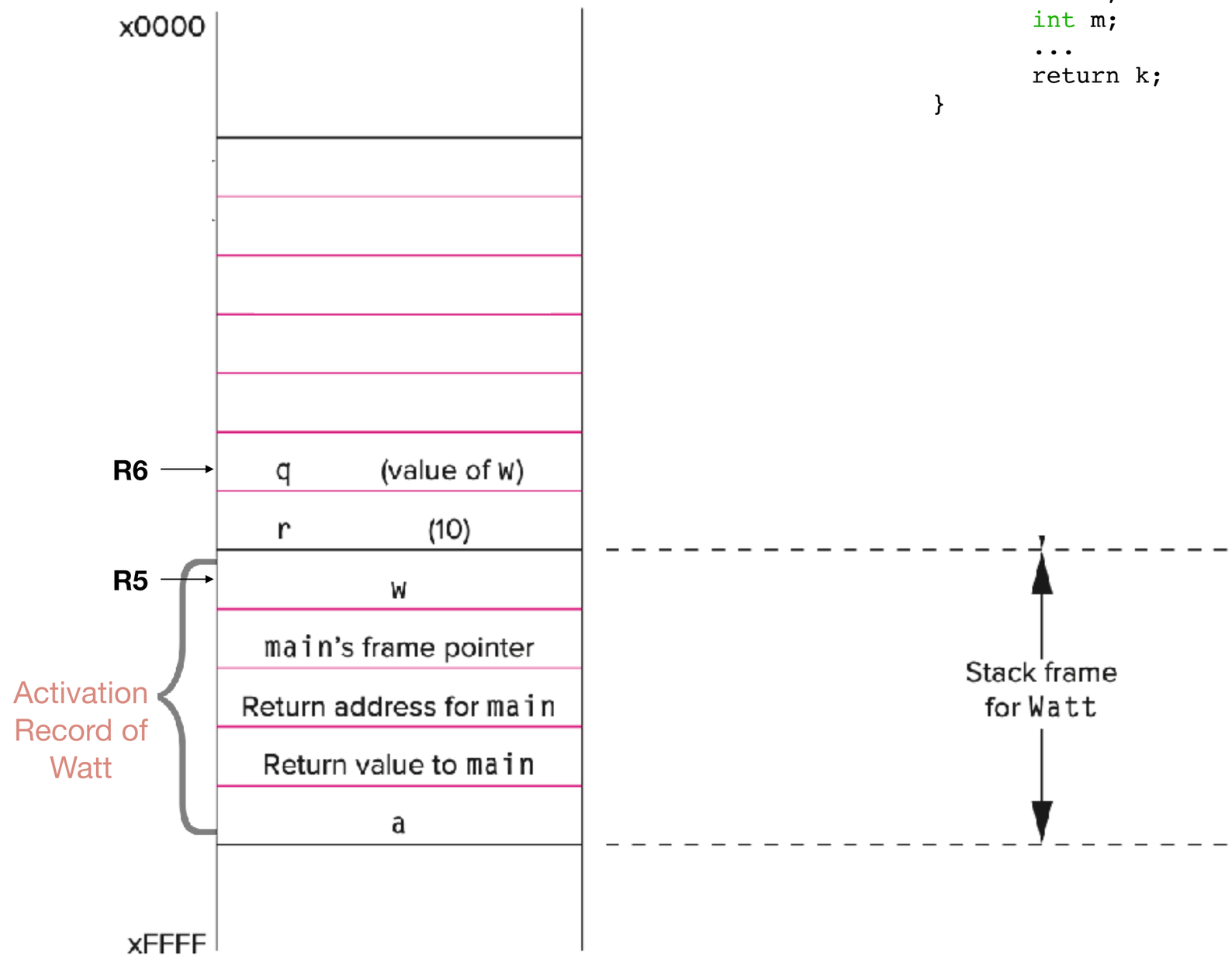
```
; push second arg
AND R0, R0, #0
ADD R0, R0, #10
ADD R6, R6, #-1
STR R0, R6, #0

; 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;
}
```

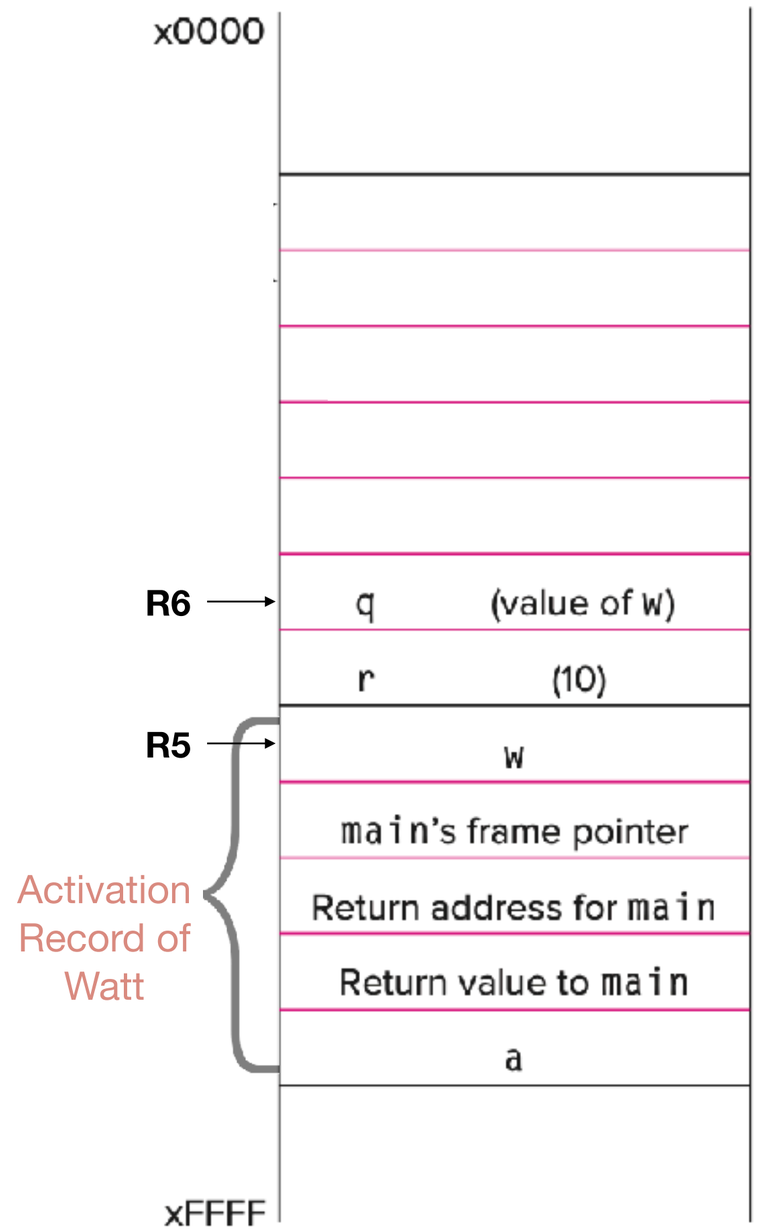
LC-3 Implementation

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



LC-3 Implementation

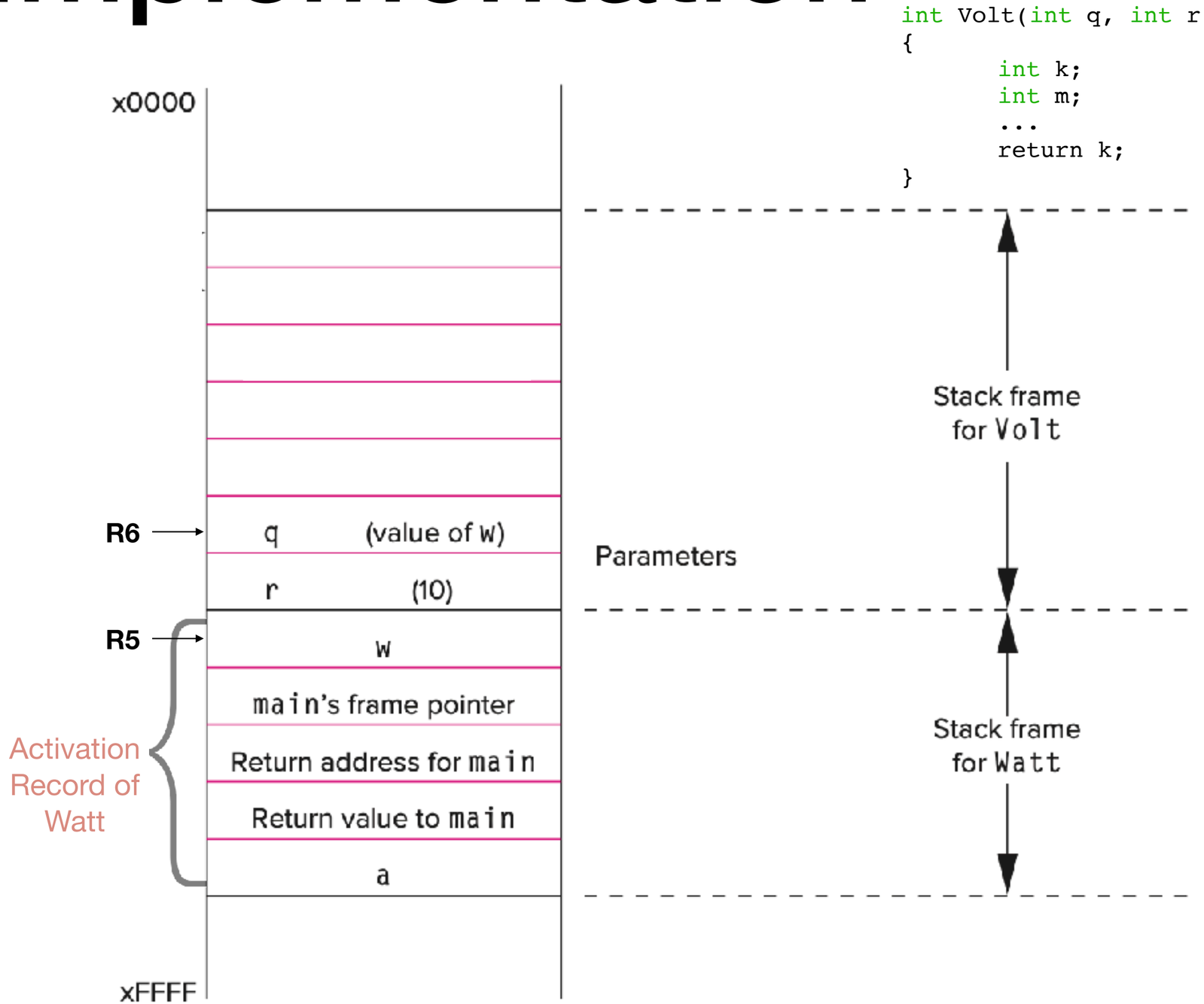
3. Callee setup (push bookkeeping info and local variables onto stack)




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

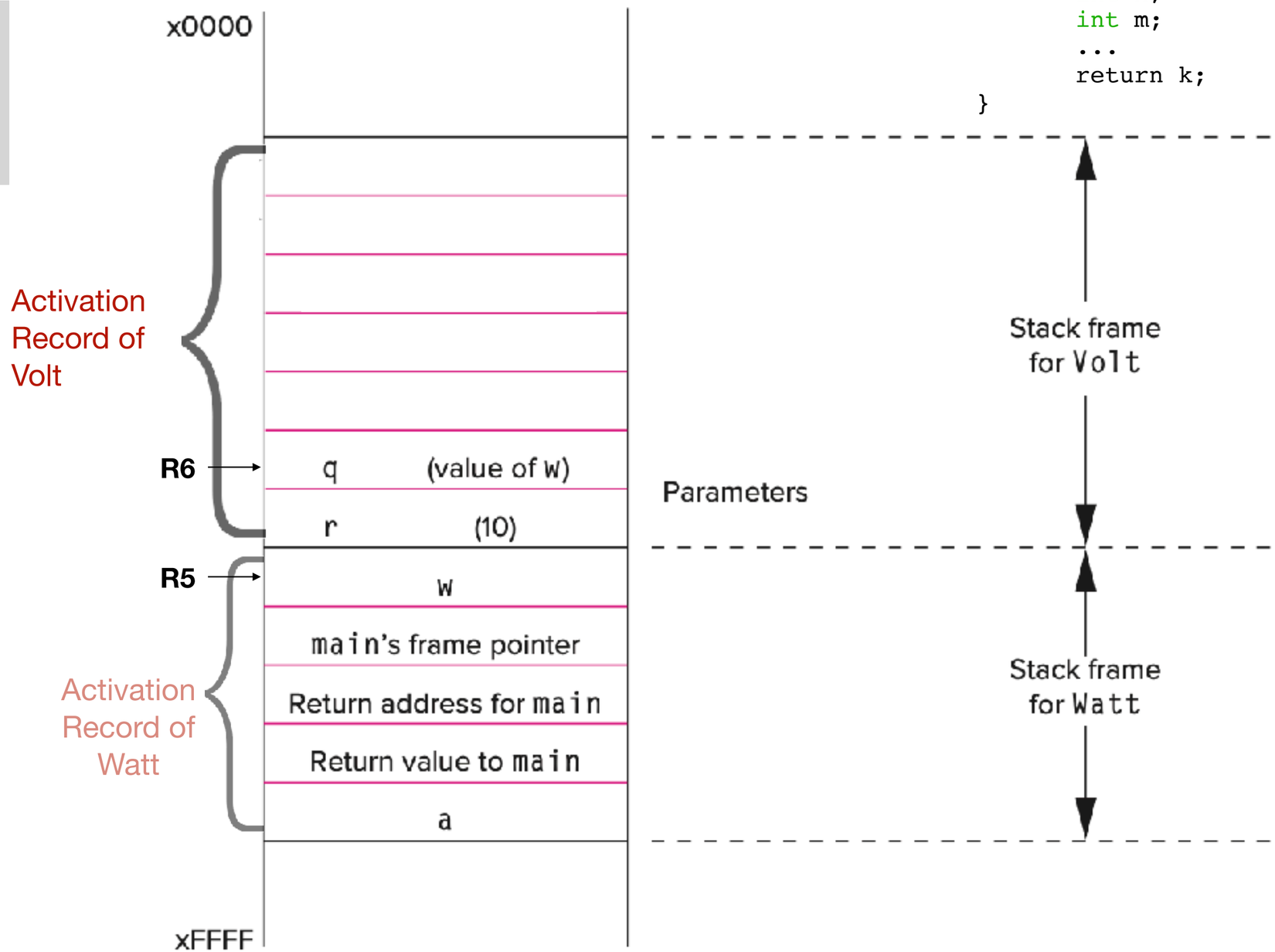
LC-3 Implementation

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



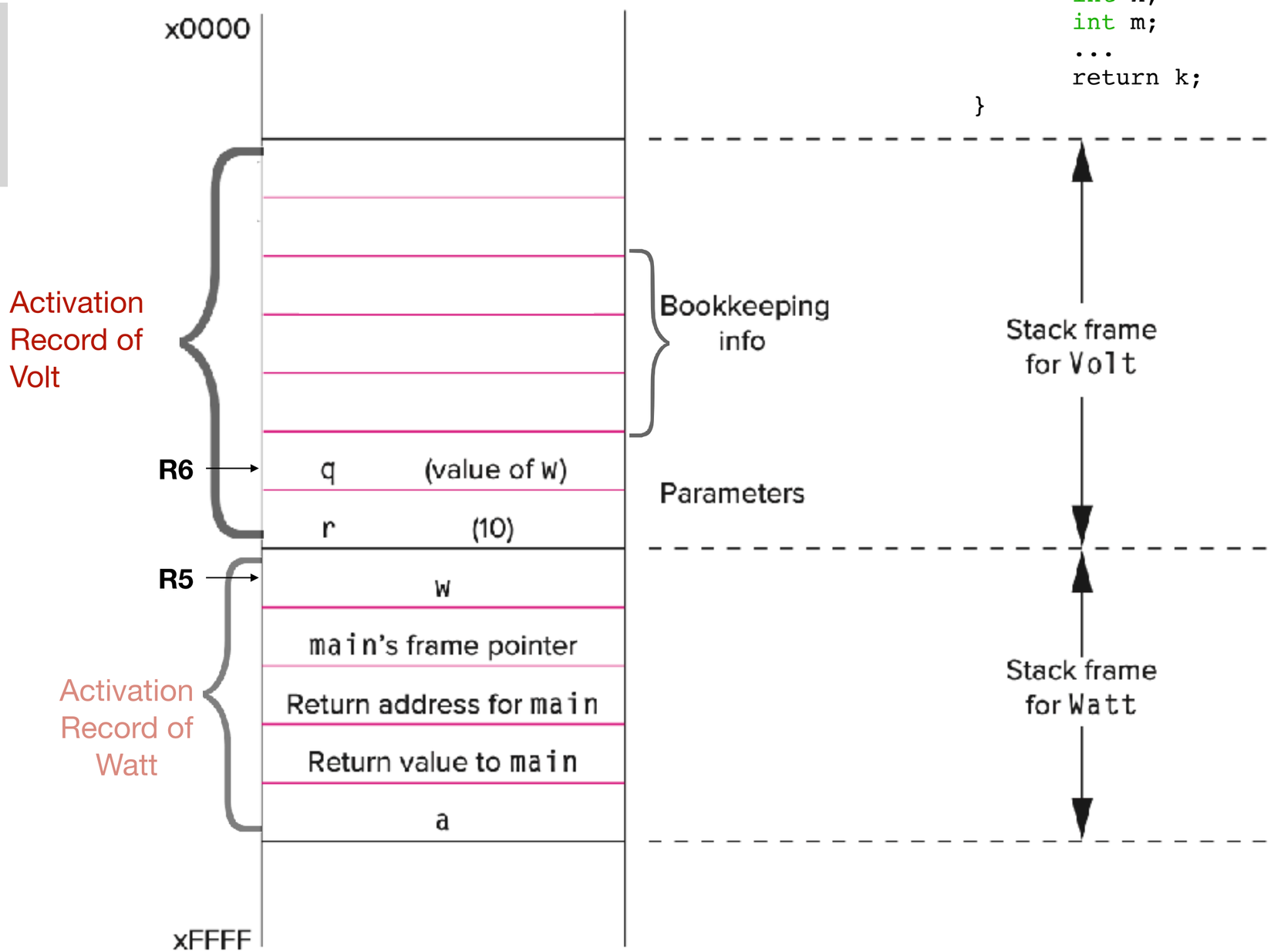
LC-3 Implementation

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



LC-3 Implementation

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

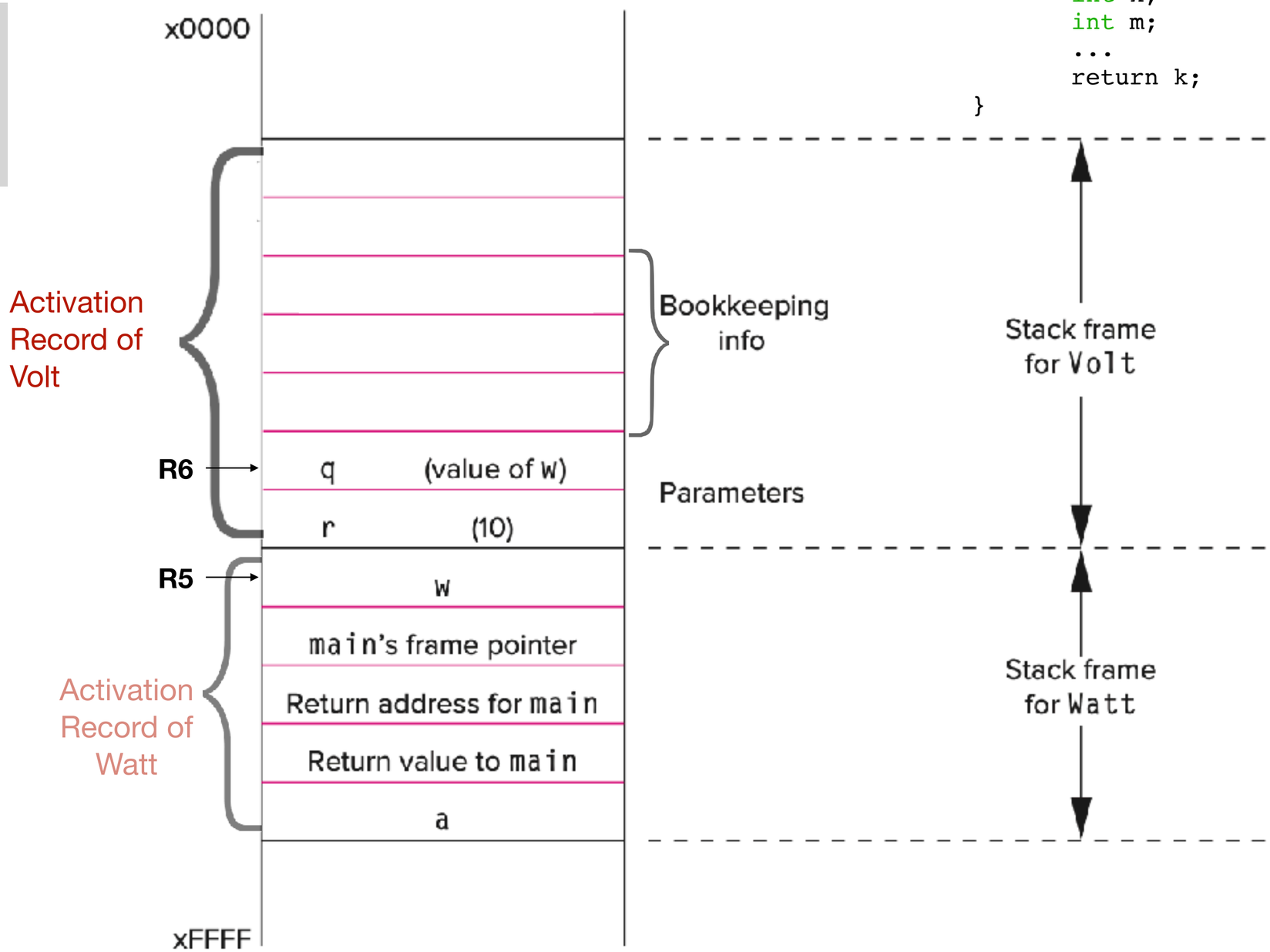


LC-3 Implementation

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

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

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

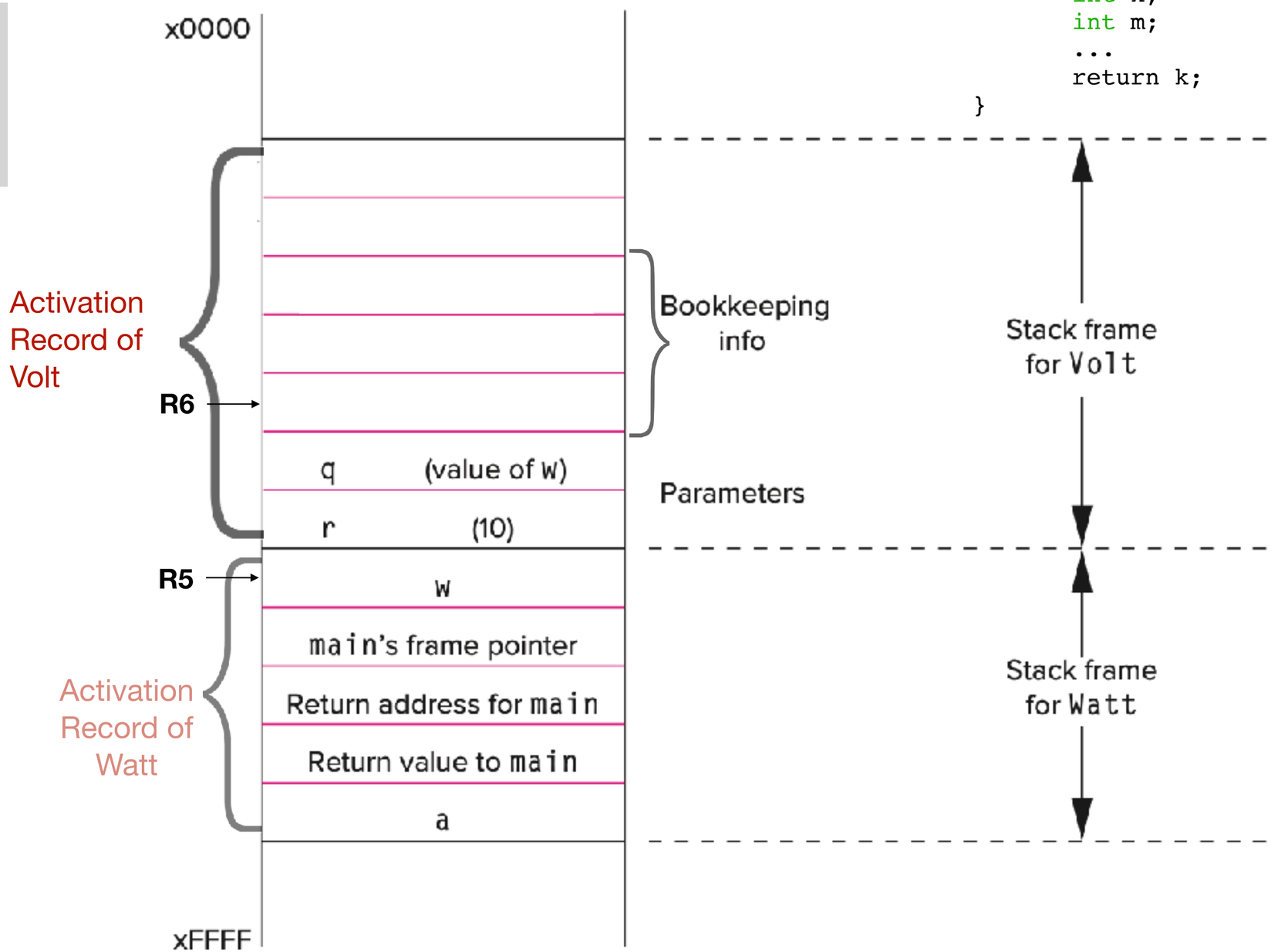


LC-3 Implementation

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

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

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

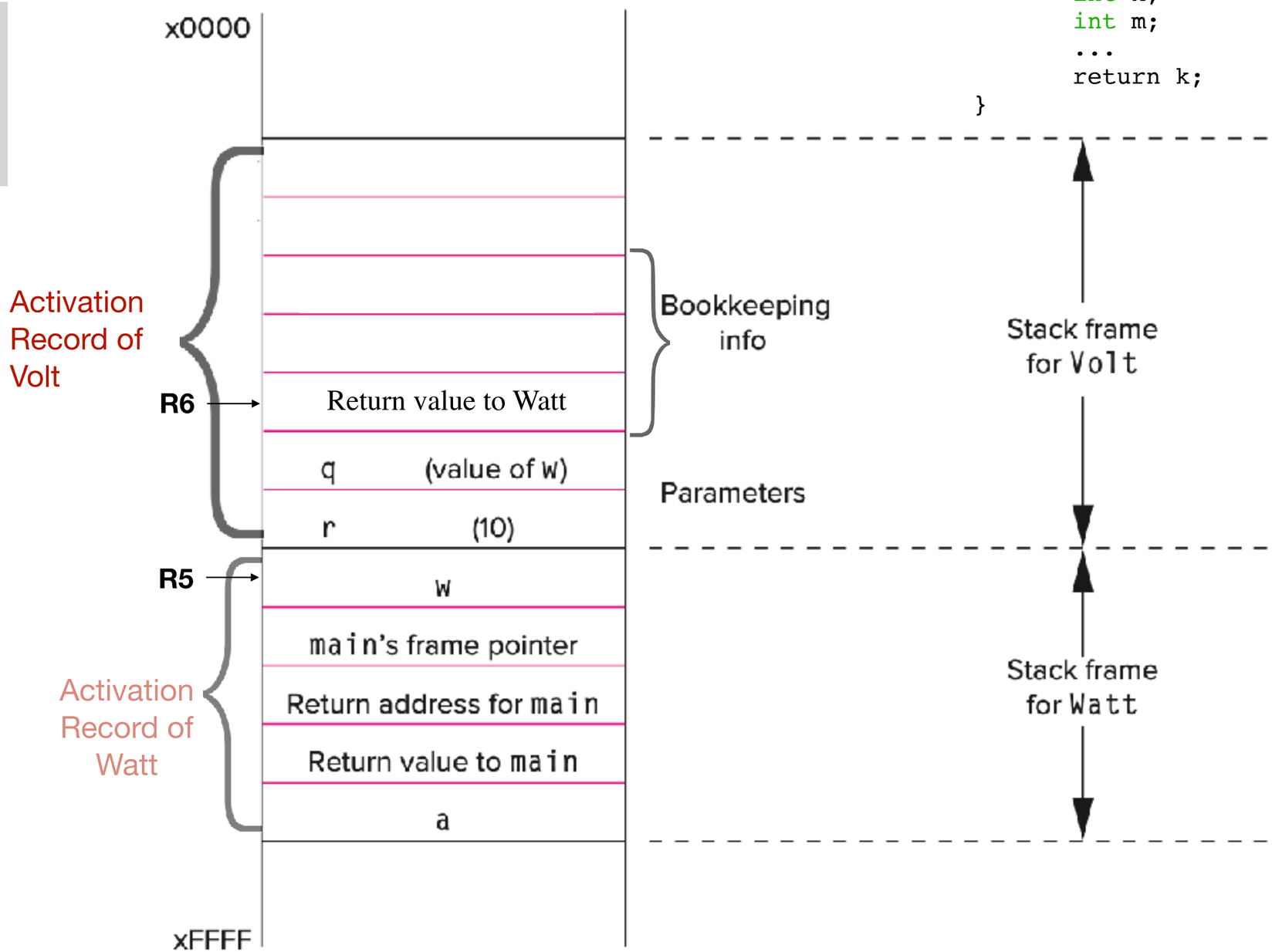


LC-3 Implementation

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

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

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

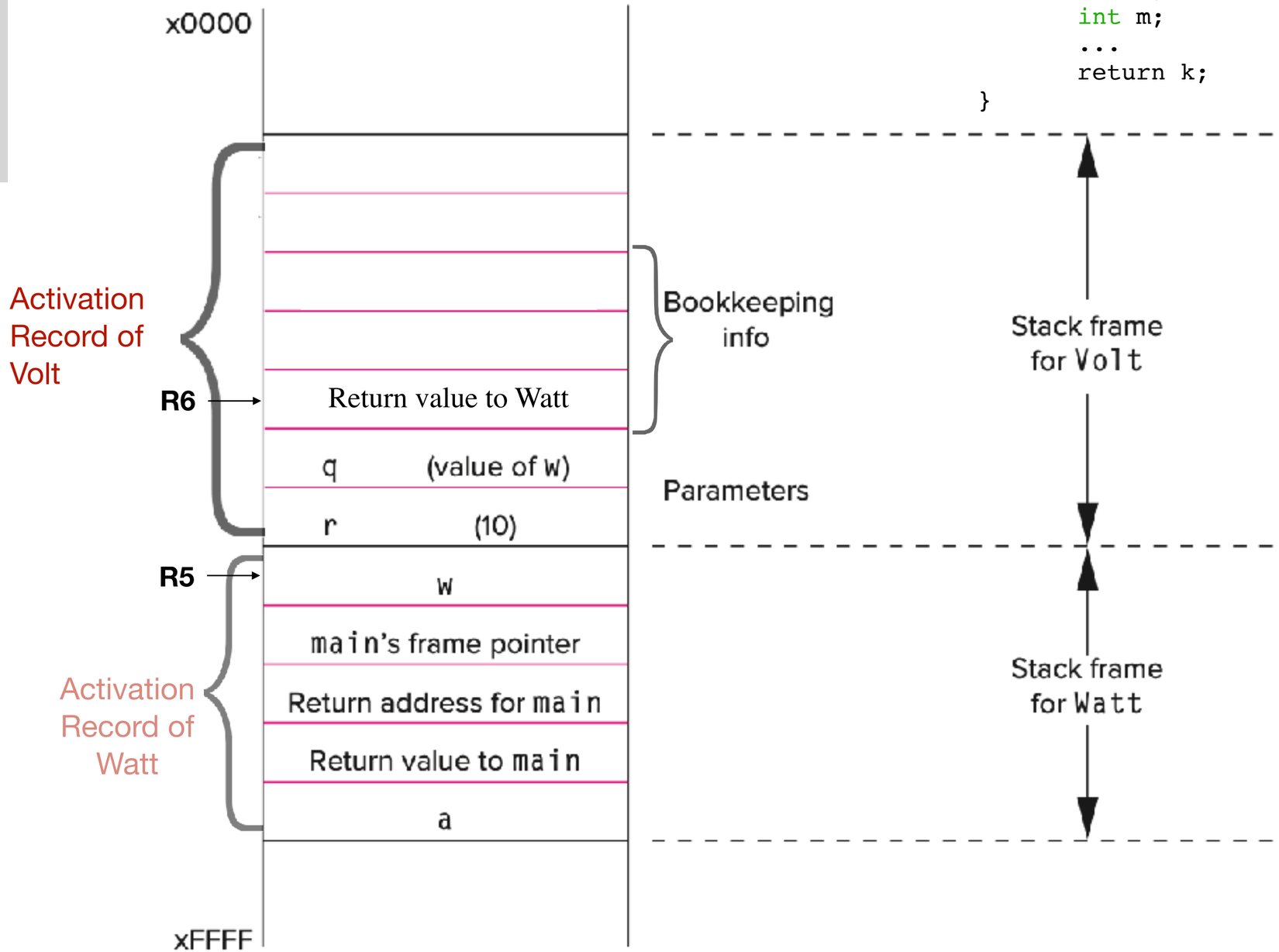


LC-3 Implementation

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

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

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

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

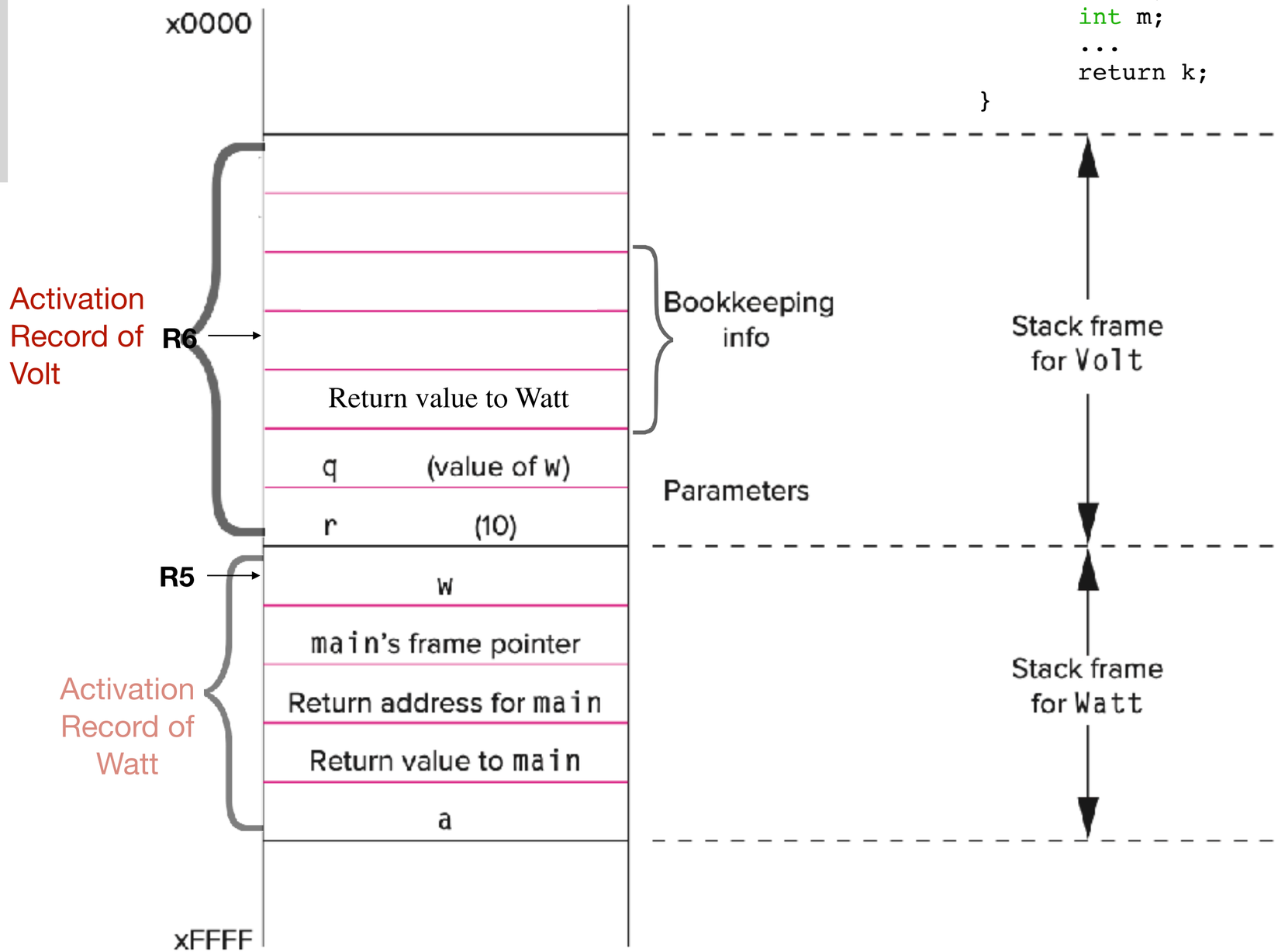


LC-3 Implementation

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

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

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

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

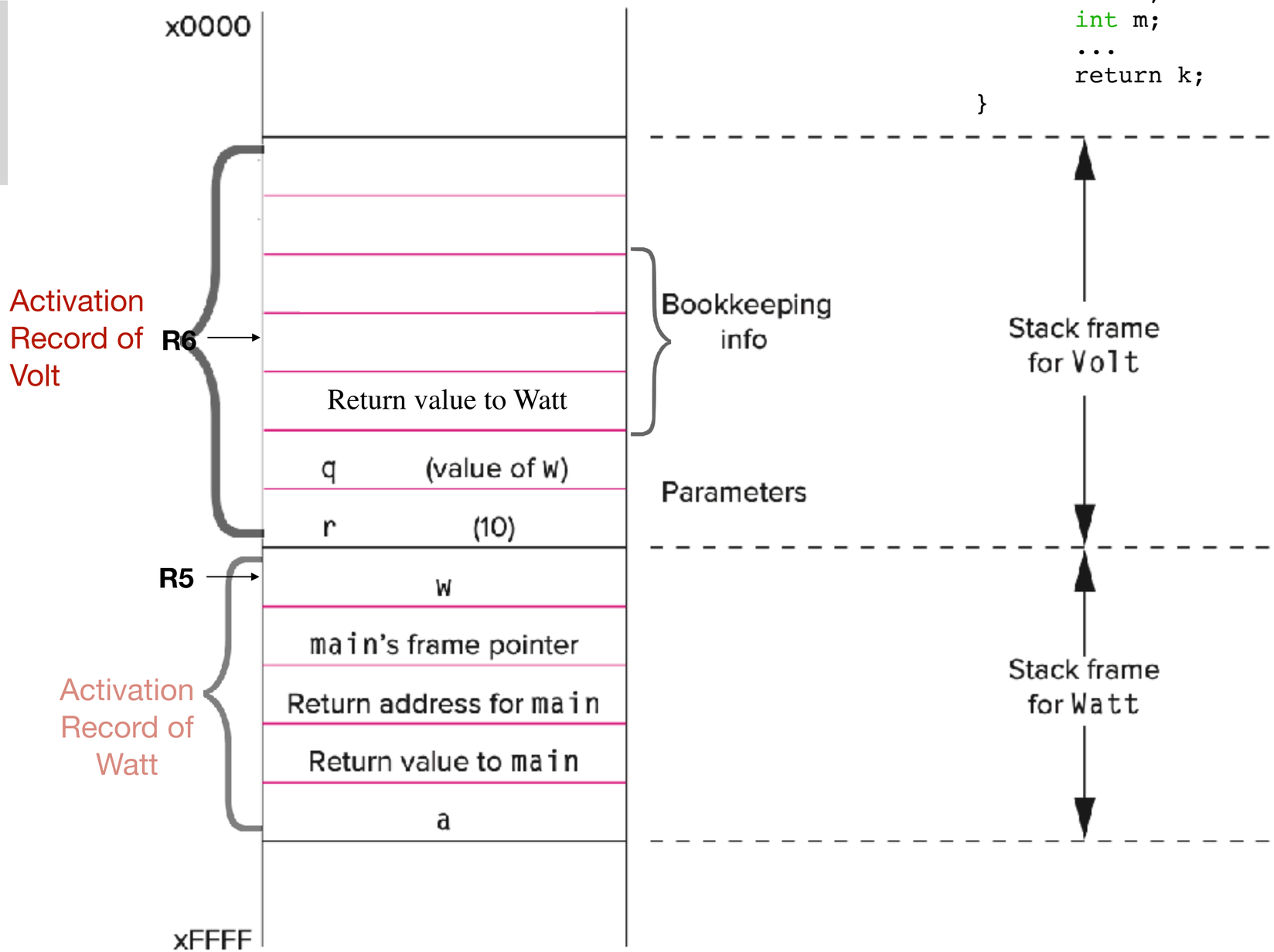


LC-3 Implementation

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

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

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

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

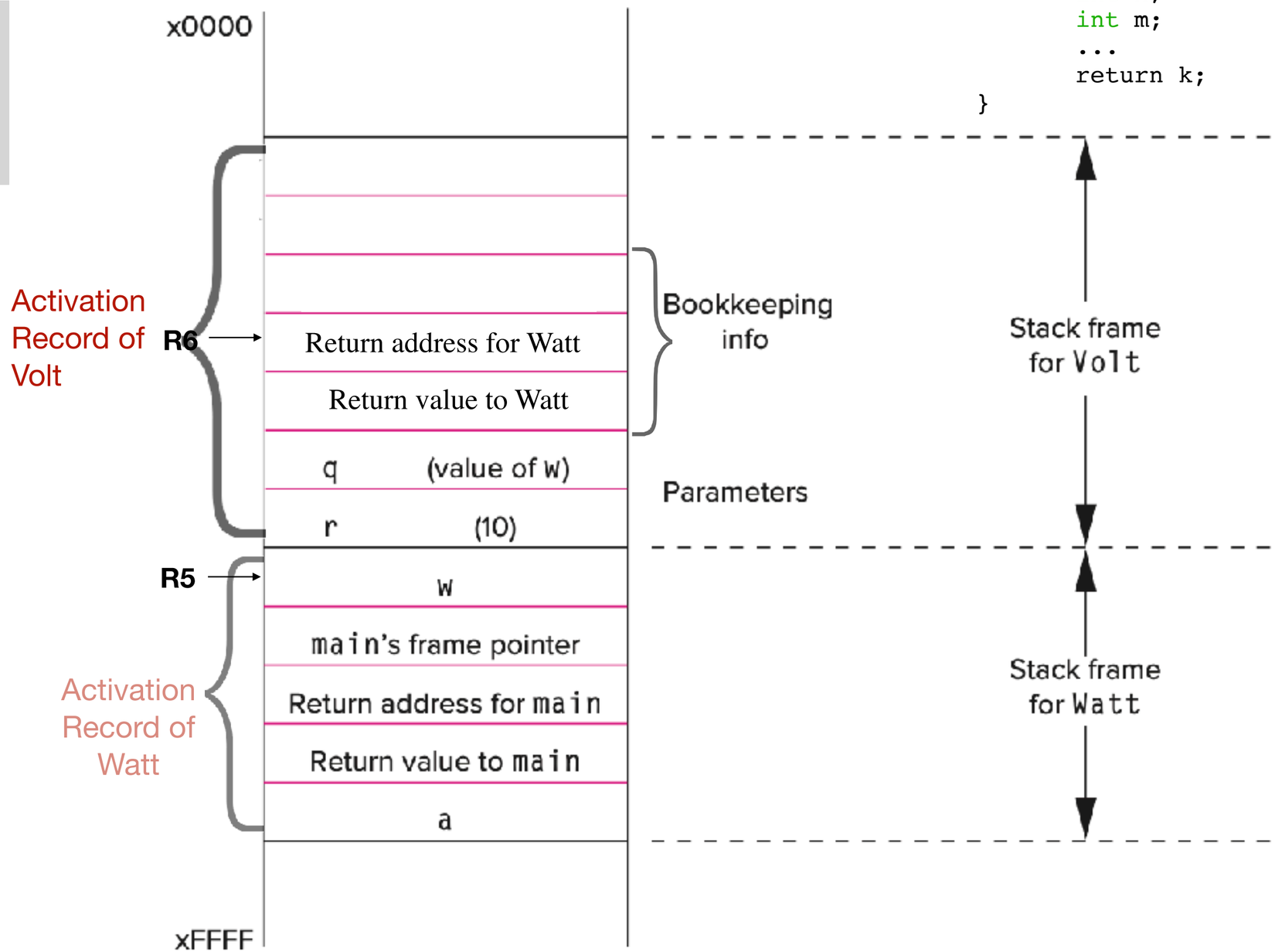


LC-3 Implementation

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

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

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

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

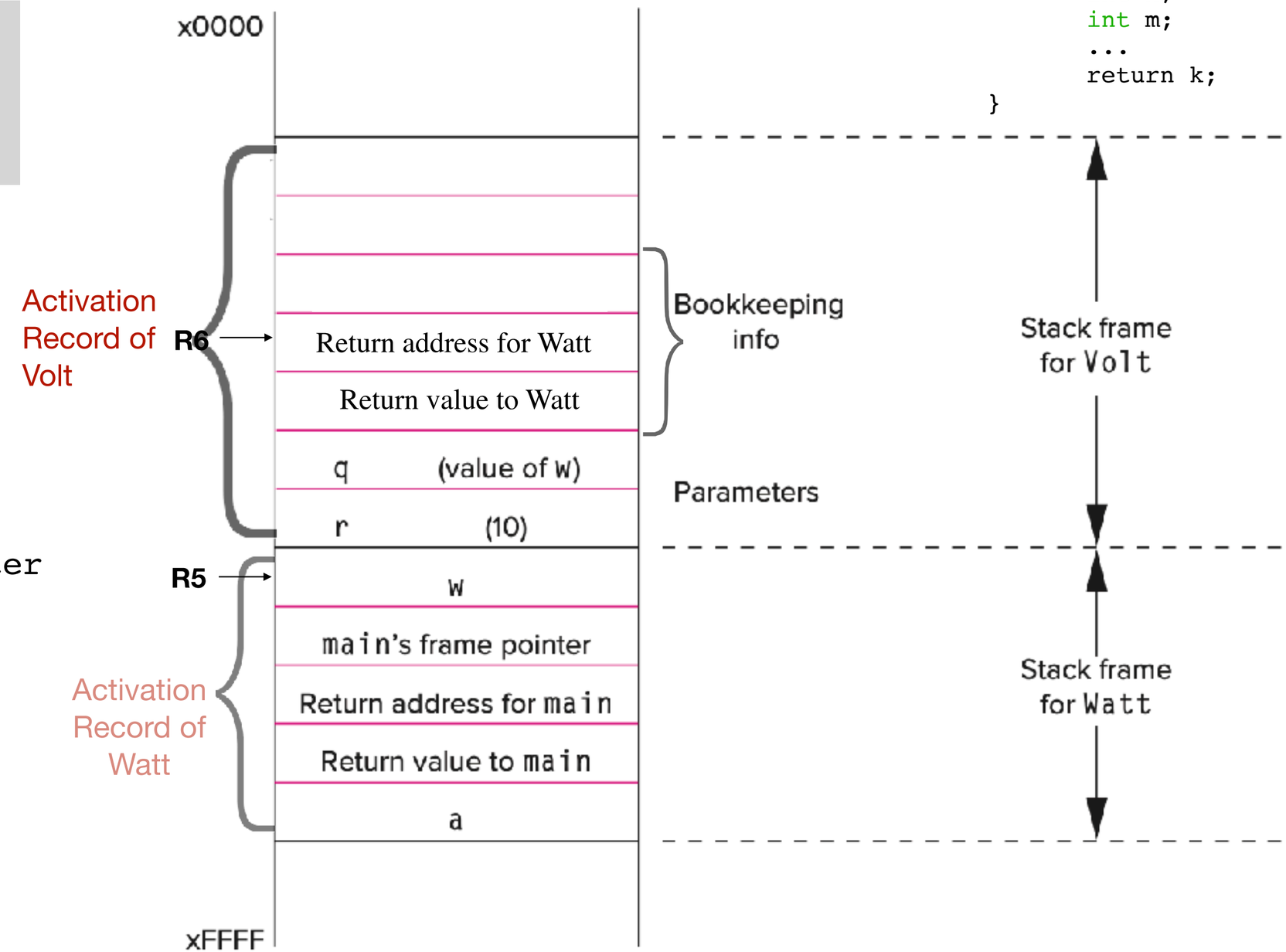


LC-3 Implementation

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

```
;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
```

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

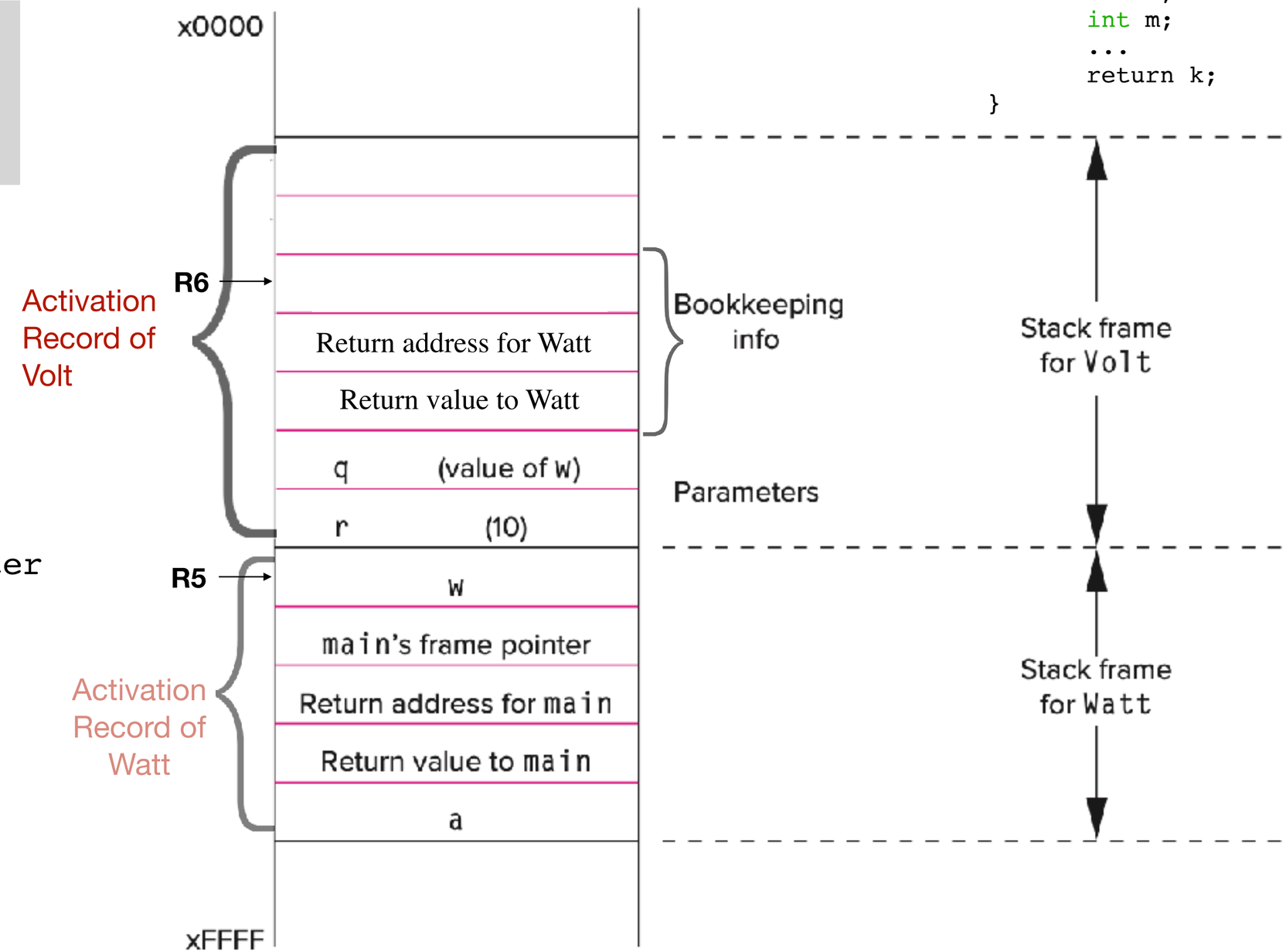


LC-3 Implementation

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

```
;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
```

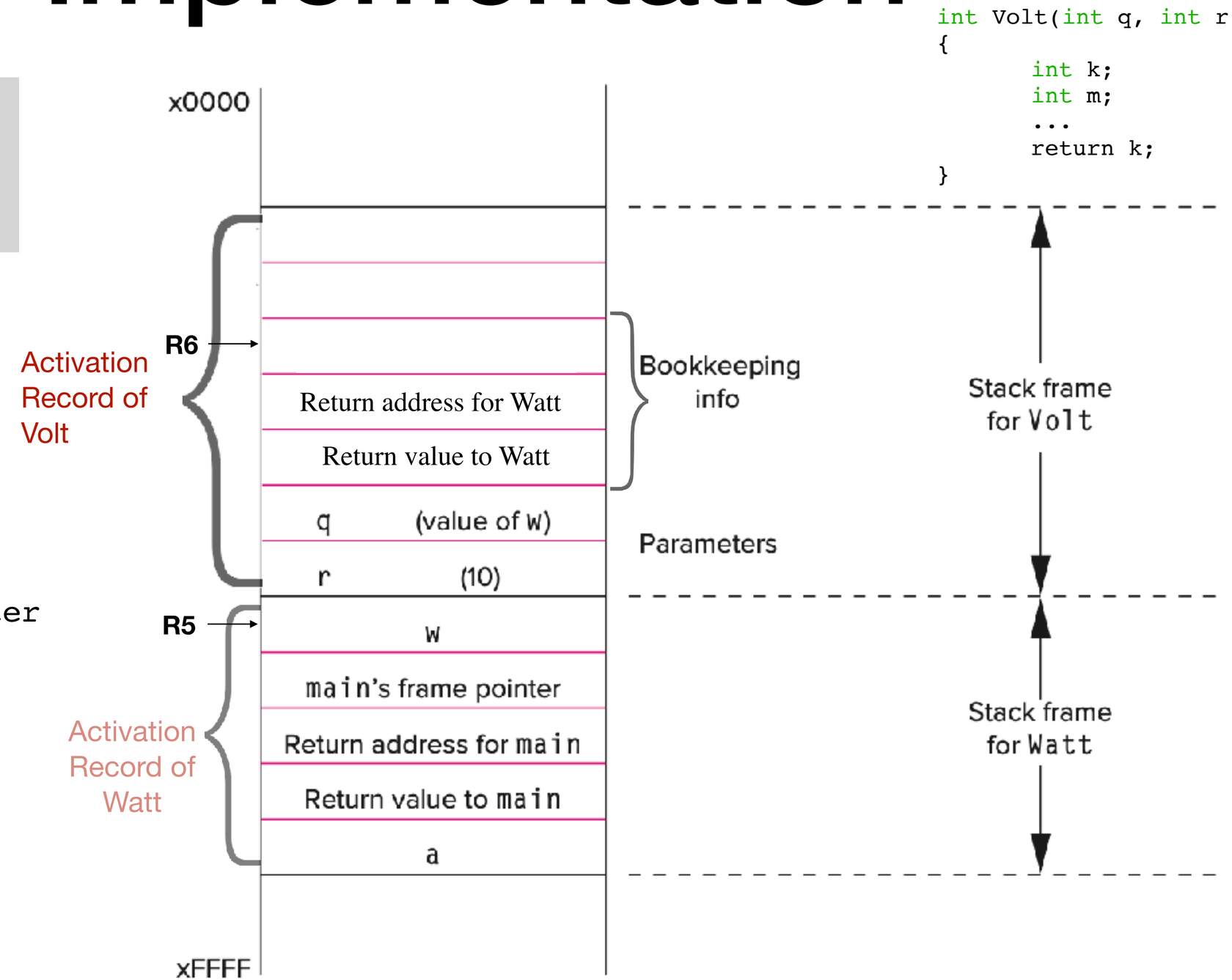
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```



LC-3 Implementation

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

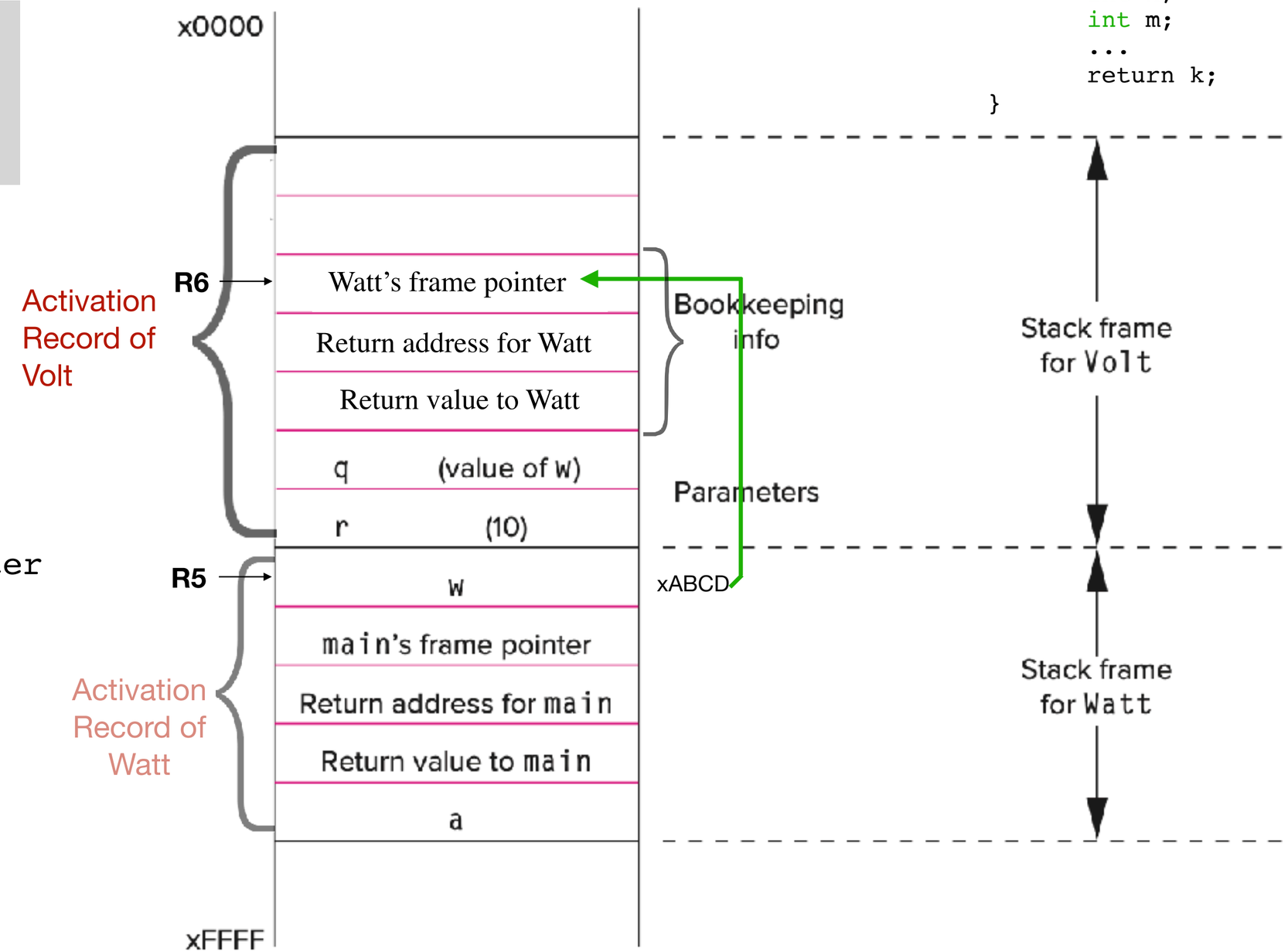
```
;space for return value
ADD R6, R6, #-1

;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)
```



LC-3 Implementation

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

```
;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
```

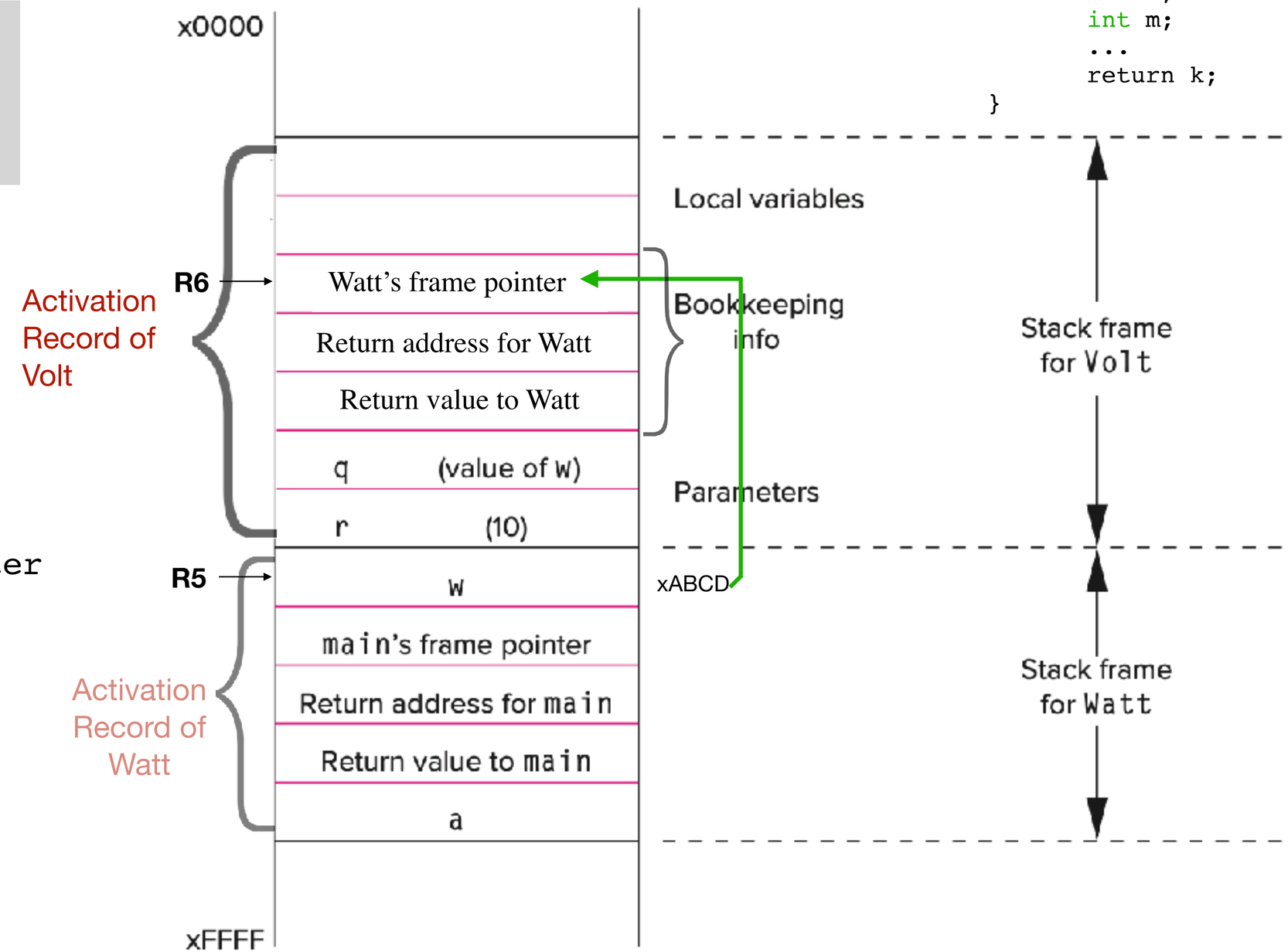


LC-3 Implementation

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

```
;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
```

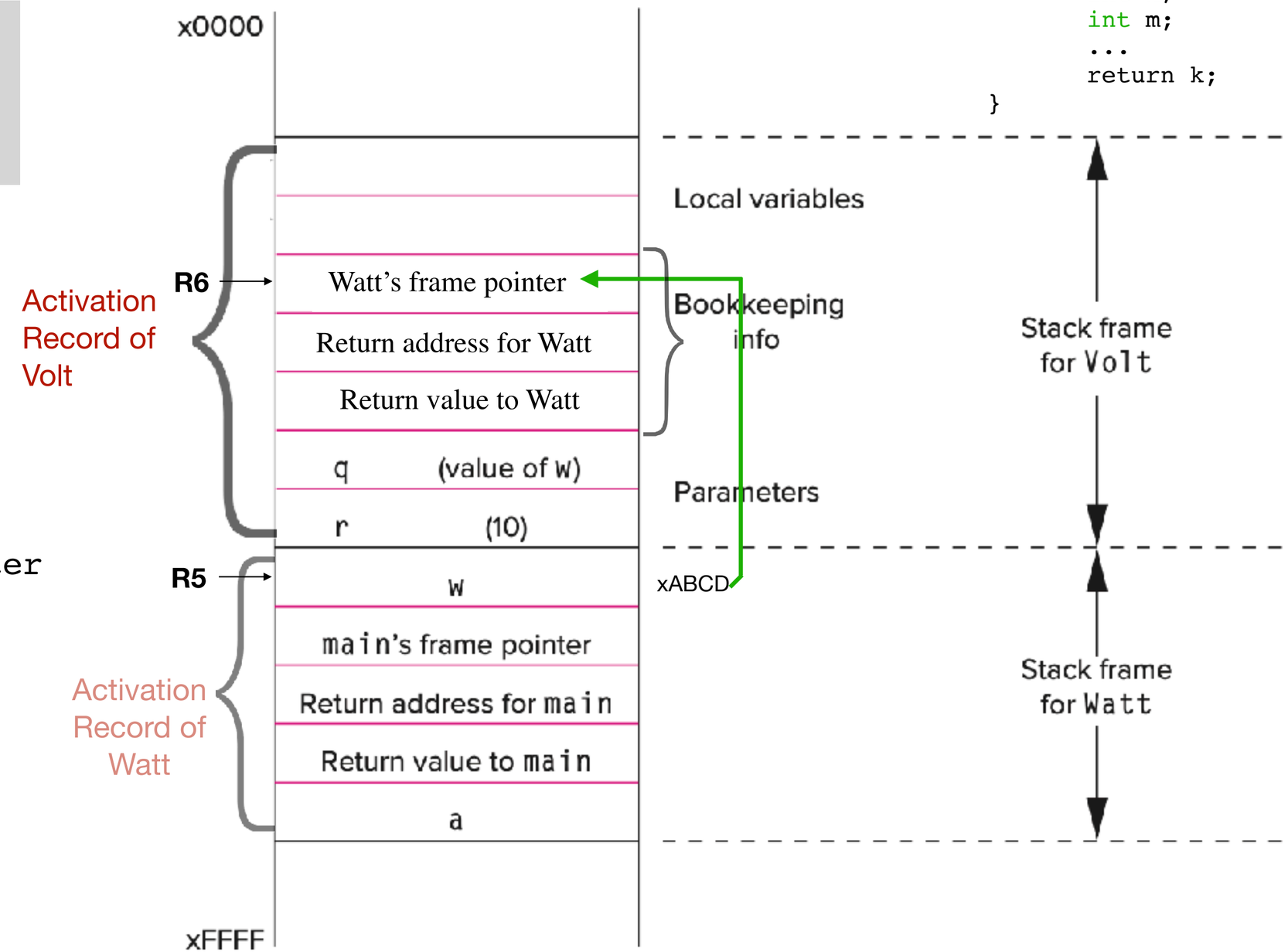
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```



LC-3 Implementation

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

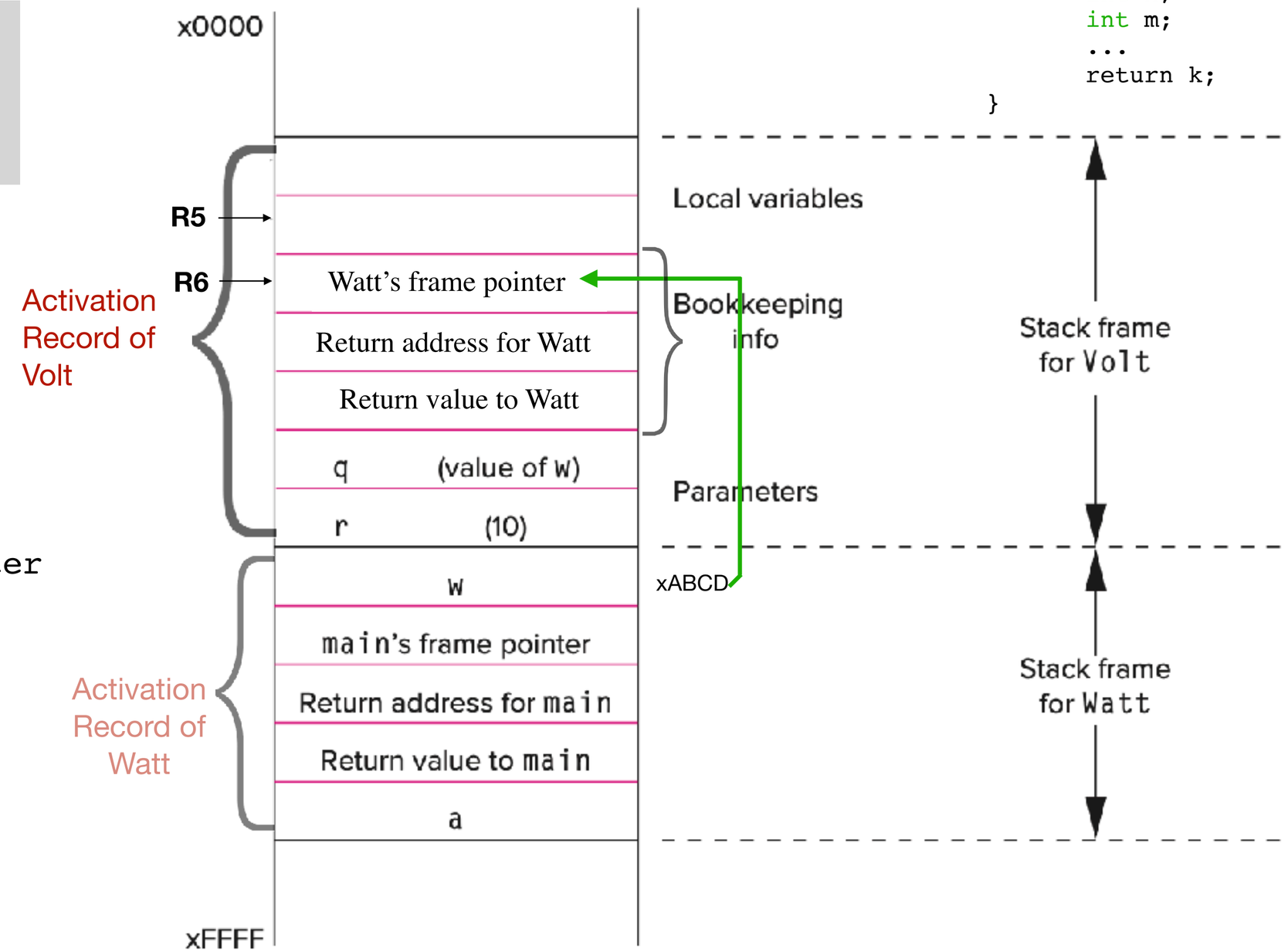
```
;space for return value
ADD R6, R6, #-1

;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
```



LC-3 Implementation

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

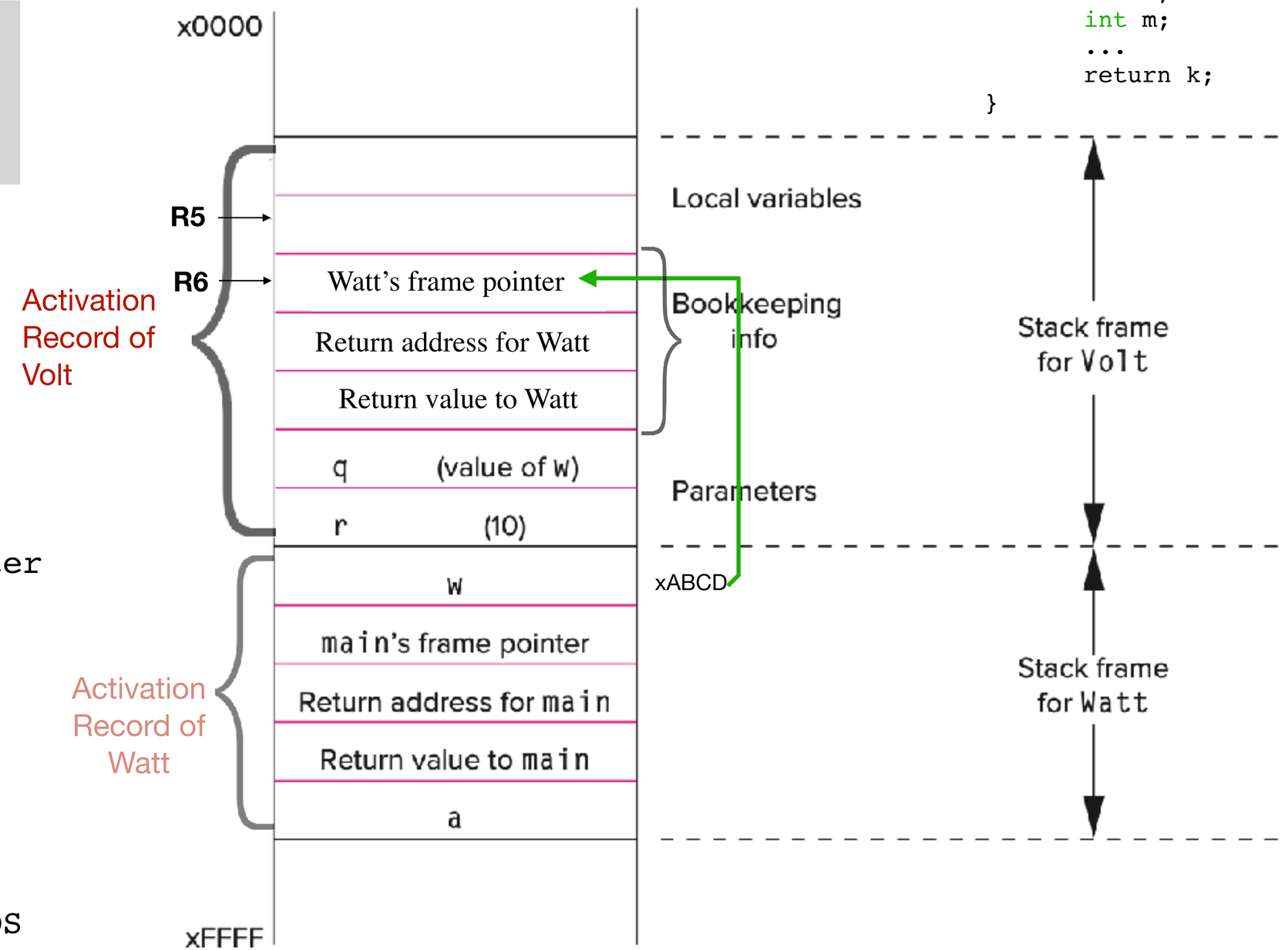
```
;space for return value
ADD R6, R6, #-1

;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
;
```



LC-3 Implementation

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

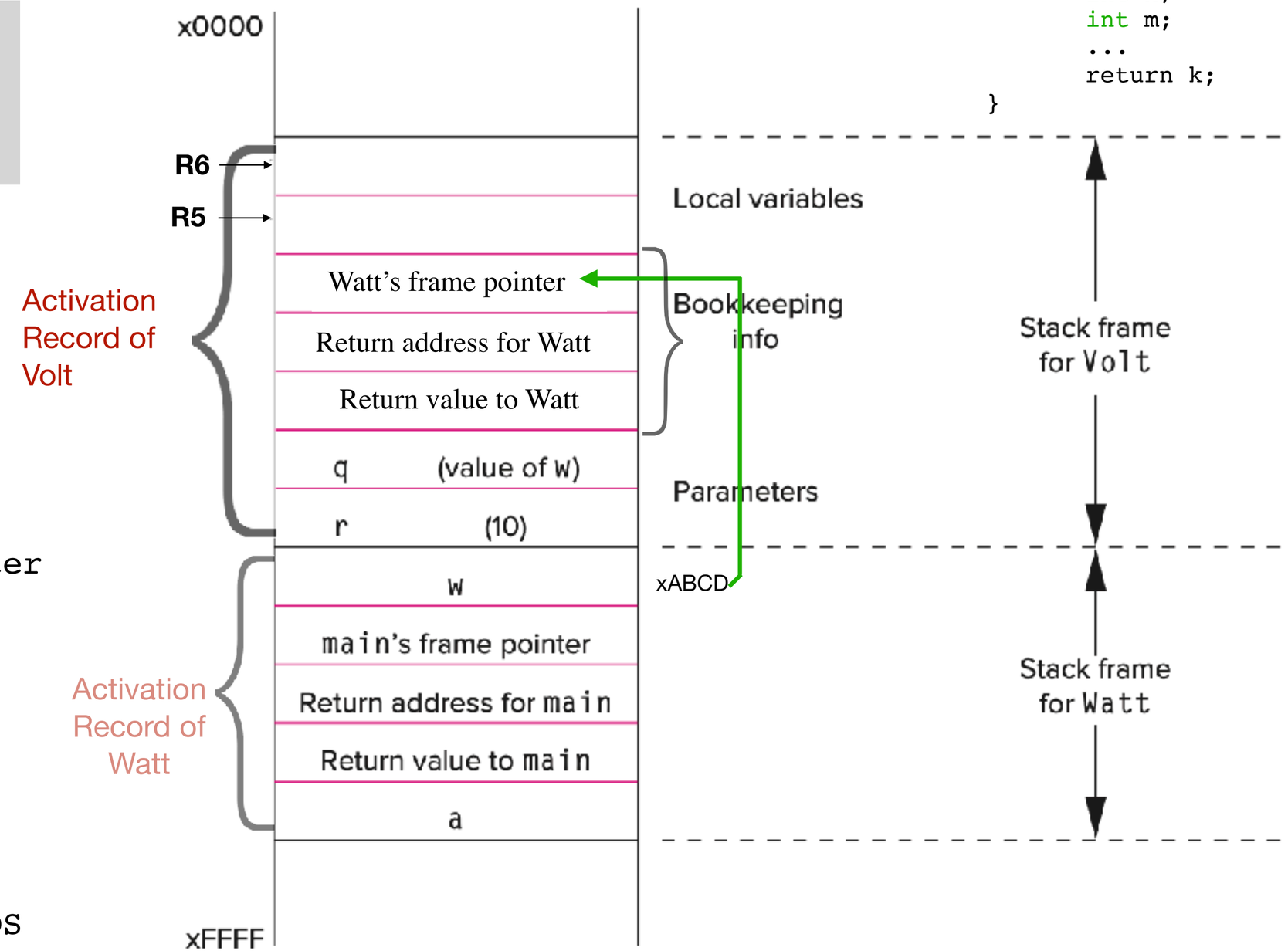
```
;space for return value
ADD R6, R6, #-1

;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
```



LC-3 Implementation

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

```
;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
```

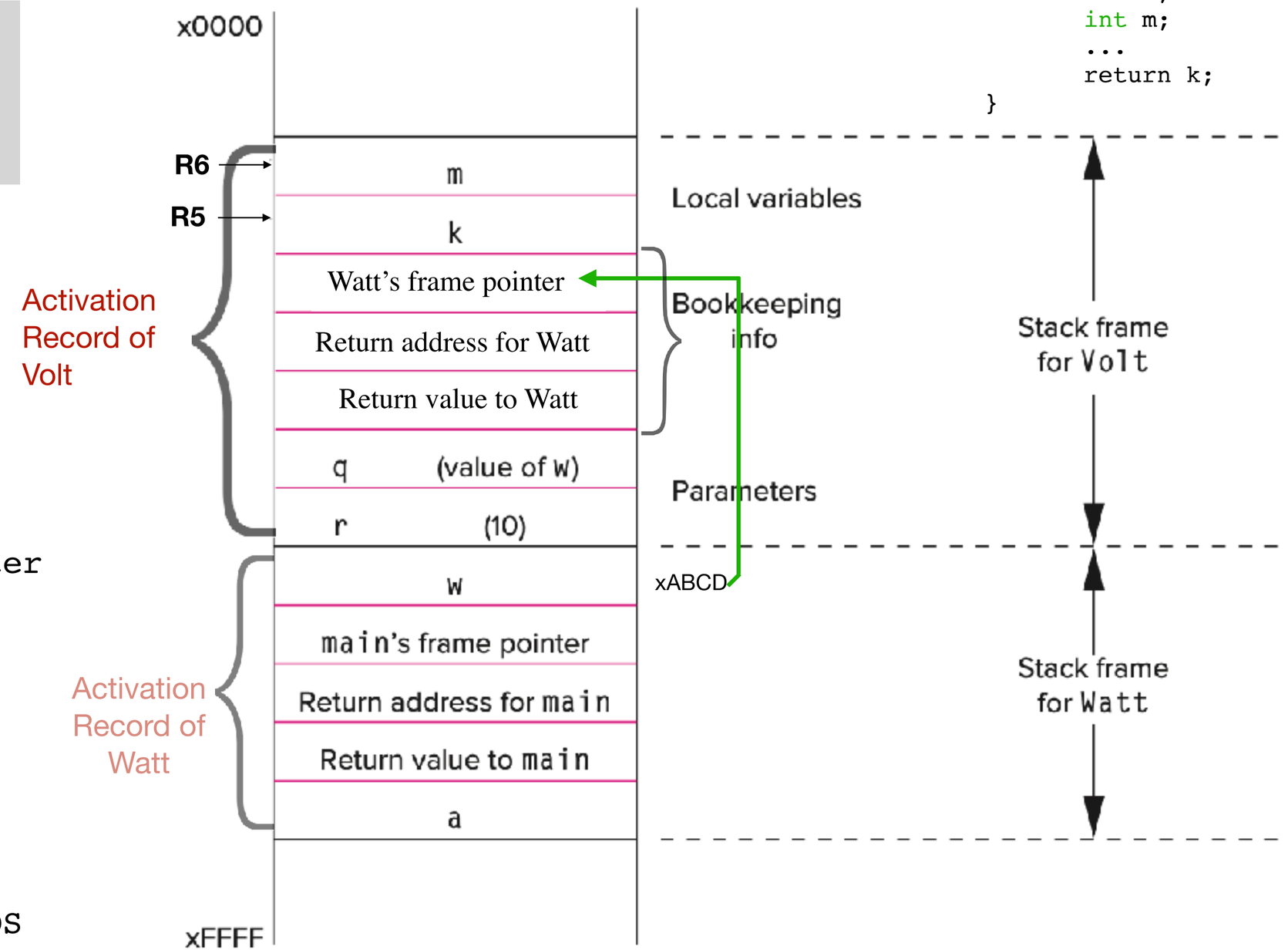



```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

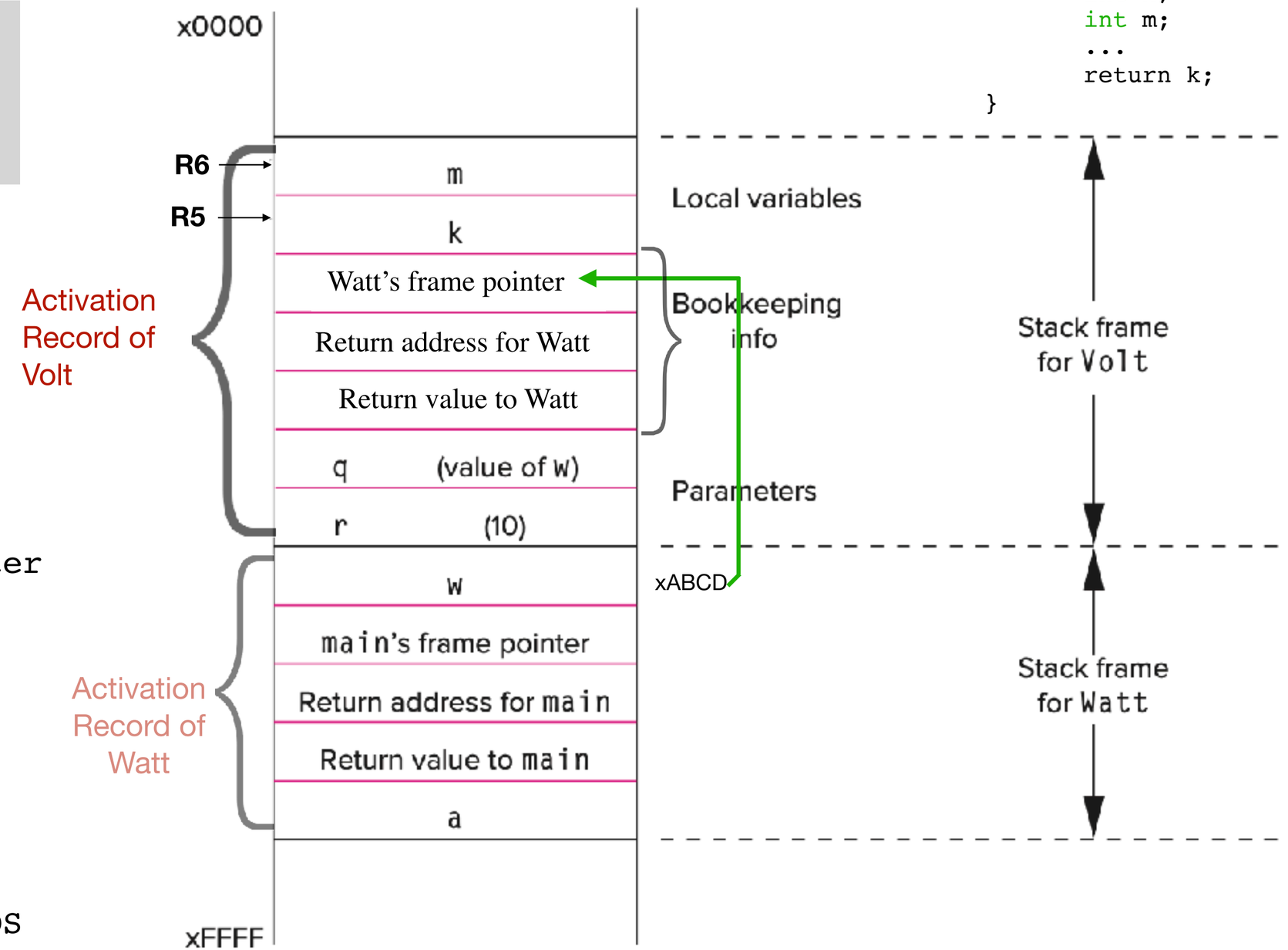
```
;space for return value
ADD R6, R6, #-1

;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
```



LC-3 Implementation

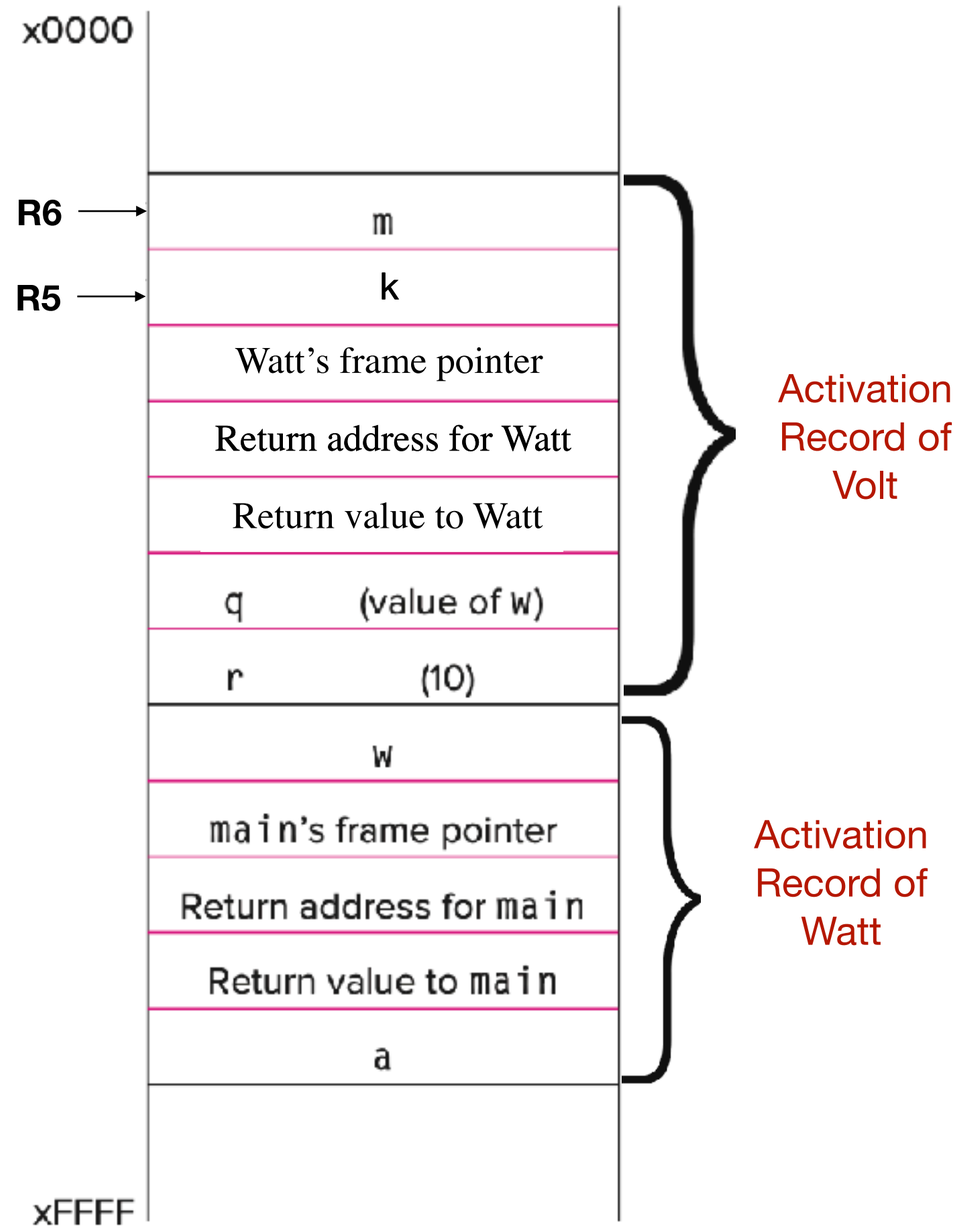
3. Callee setup (push bookkeeping info and local variables onto stack)
4. Execute function

```
;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;
}
```

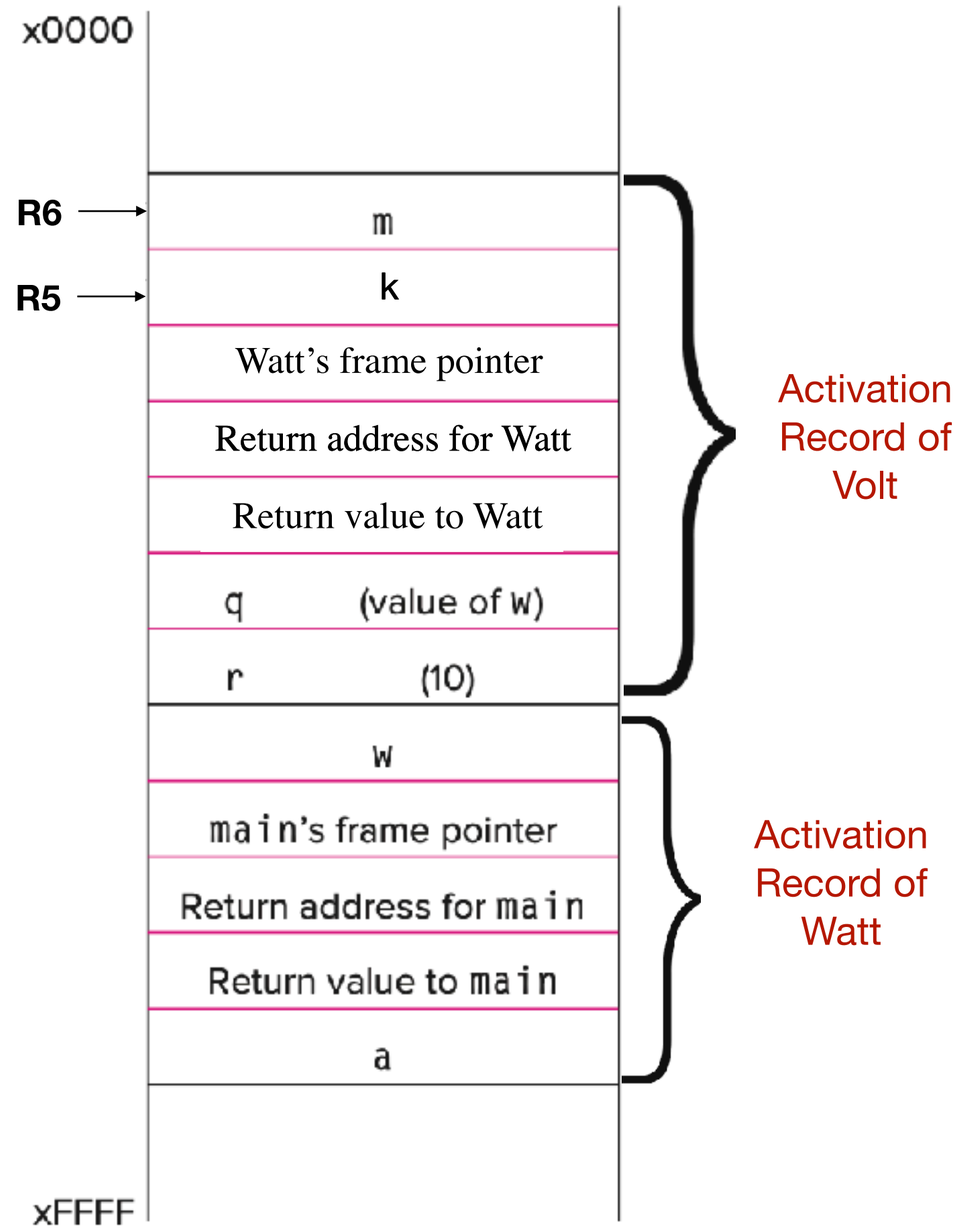
LC-3 Implementation



```
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)

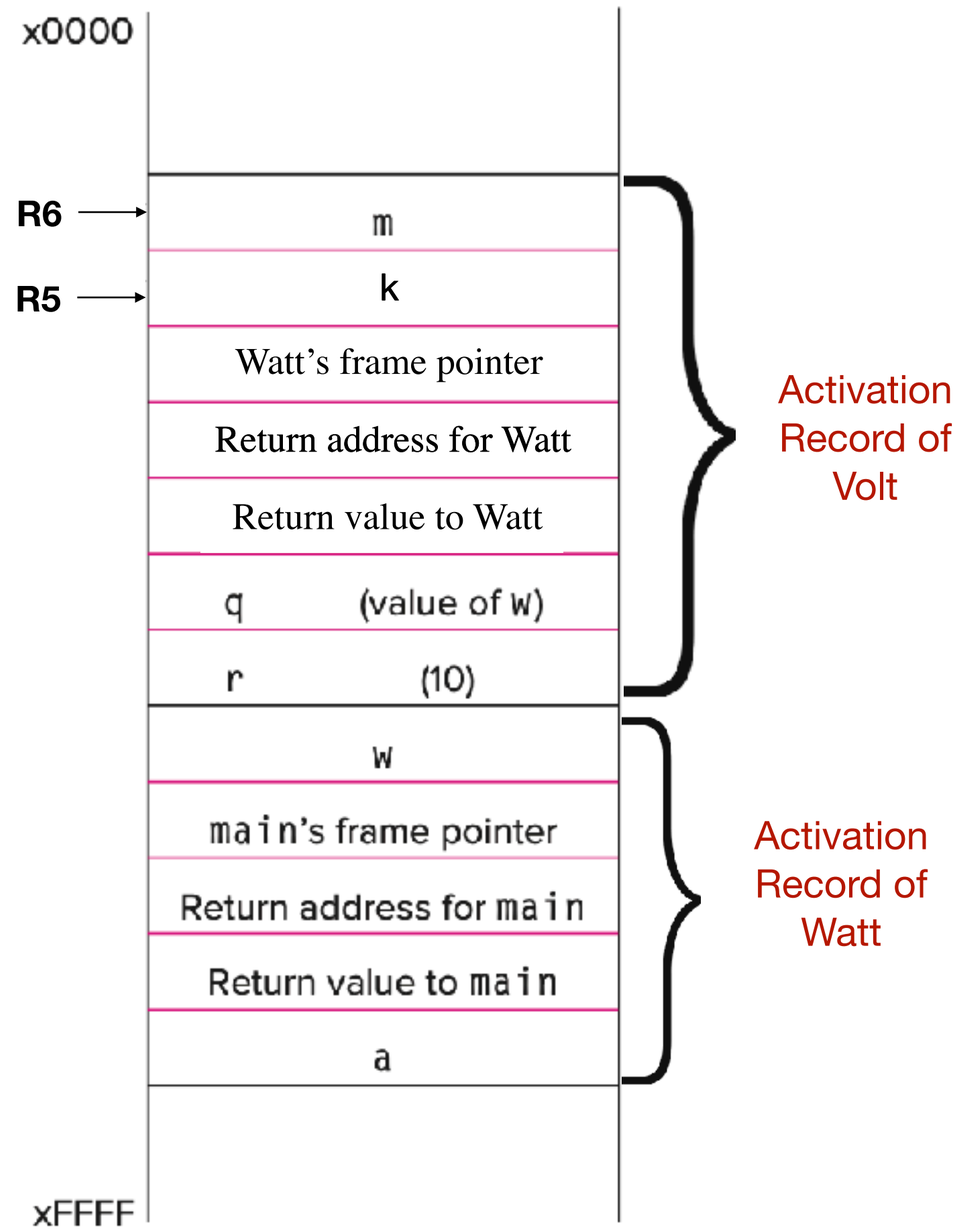


```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; copy k into return value(R5+3)
```

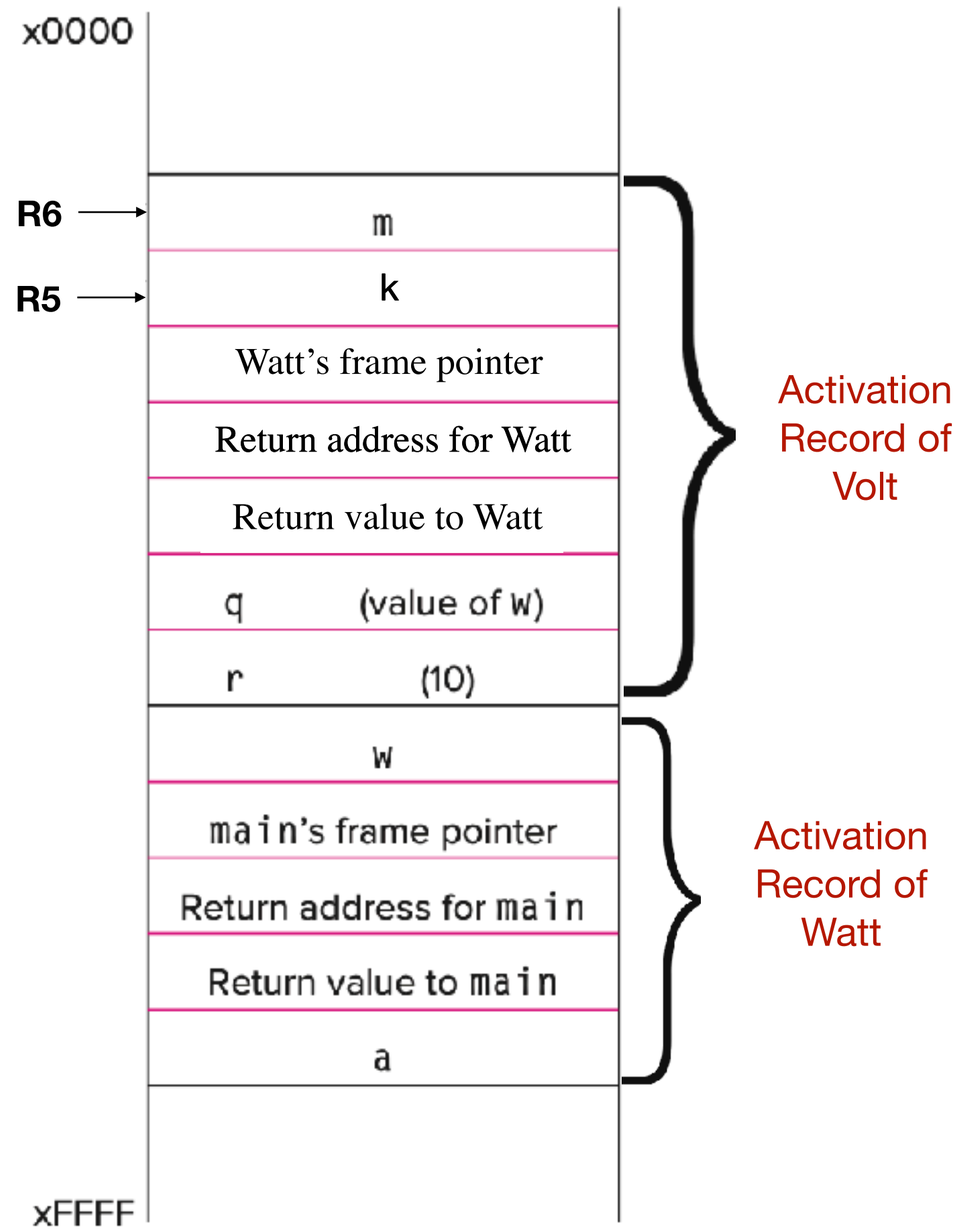


```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

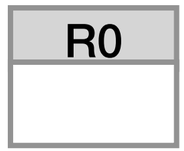


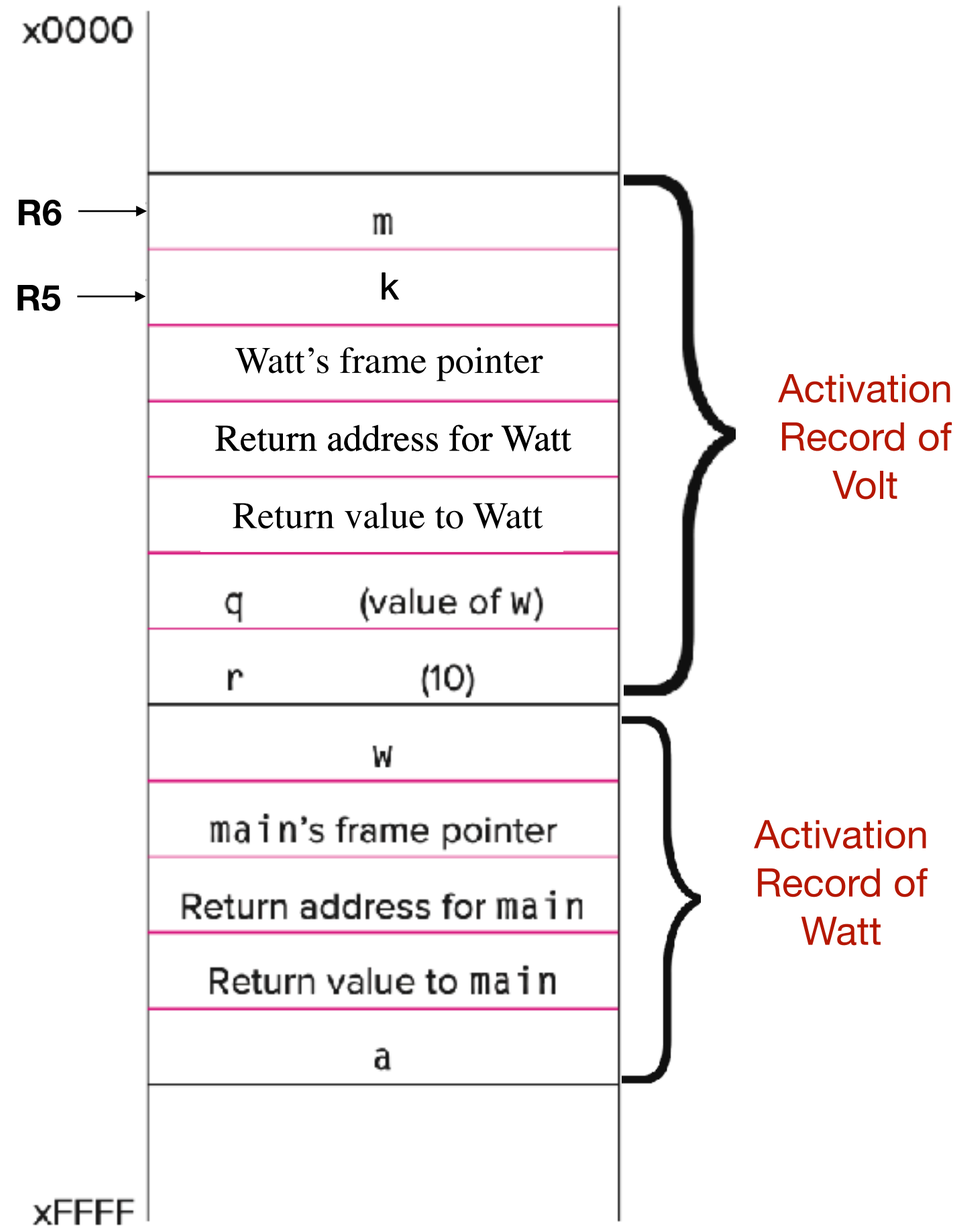
```c
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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





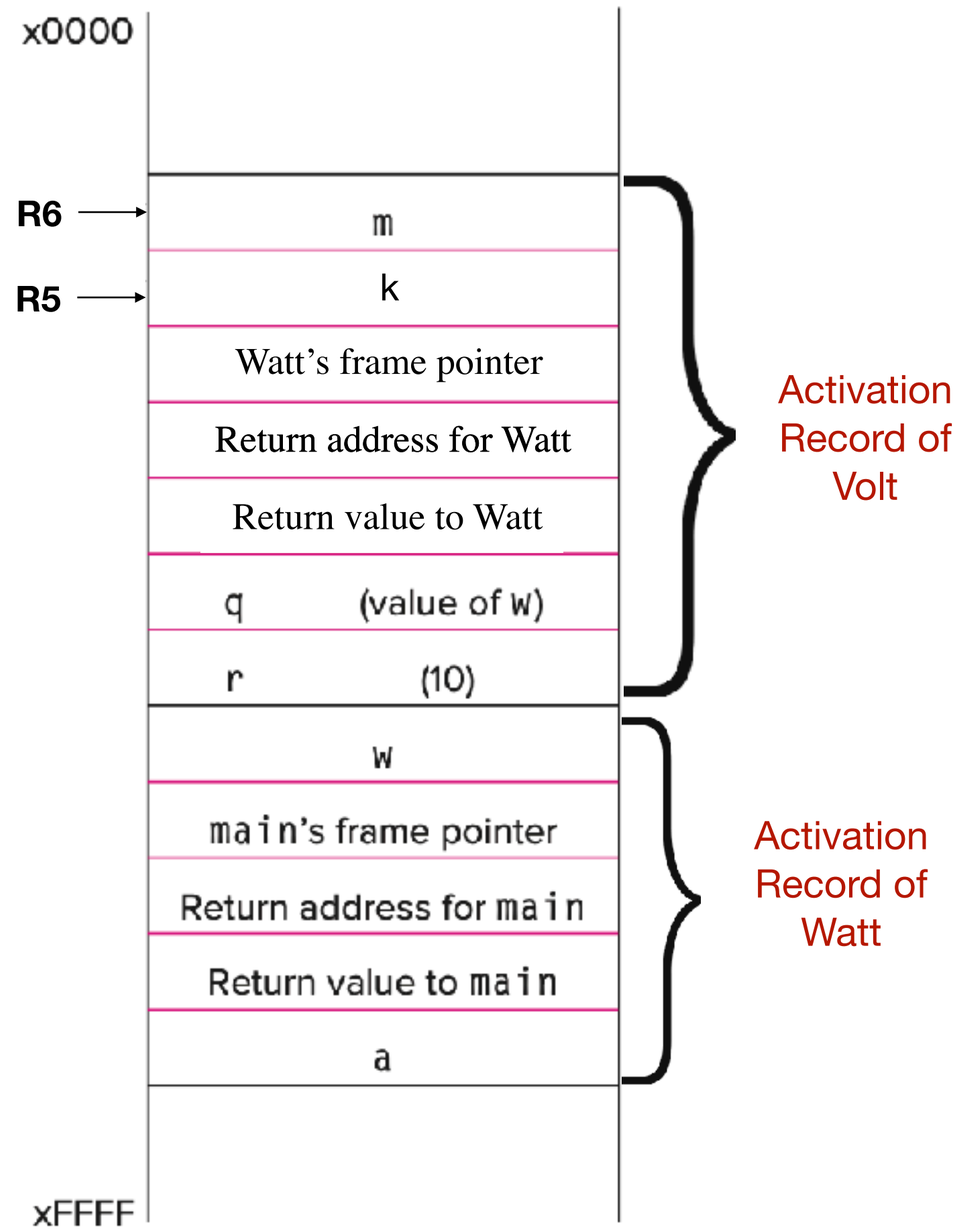
```c
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

R0
k



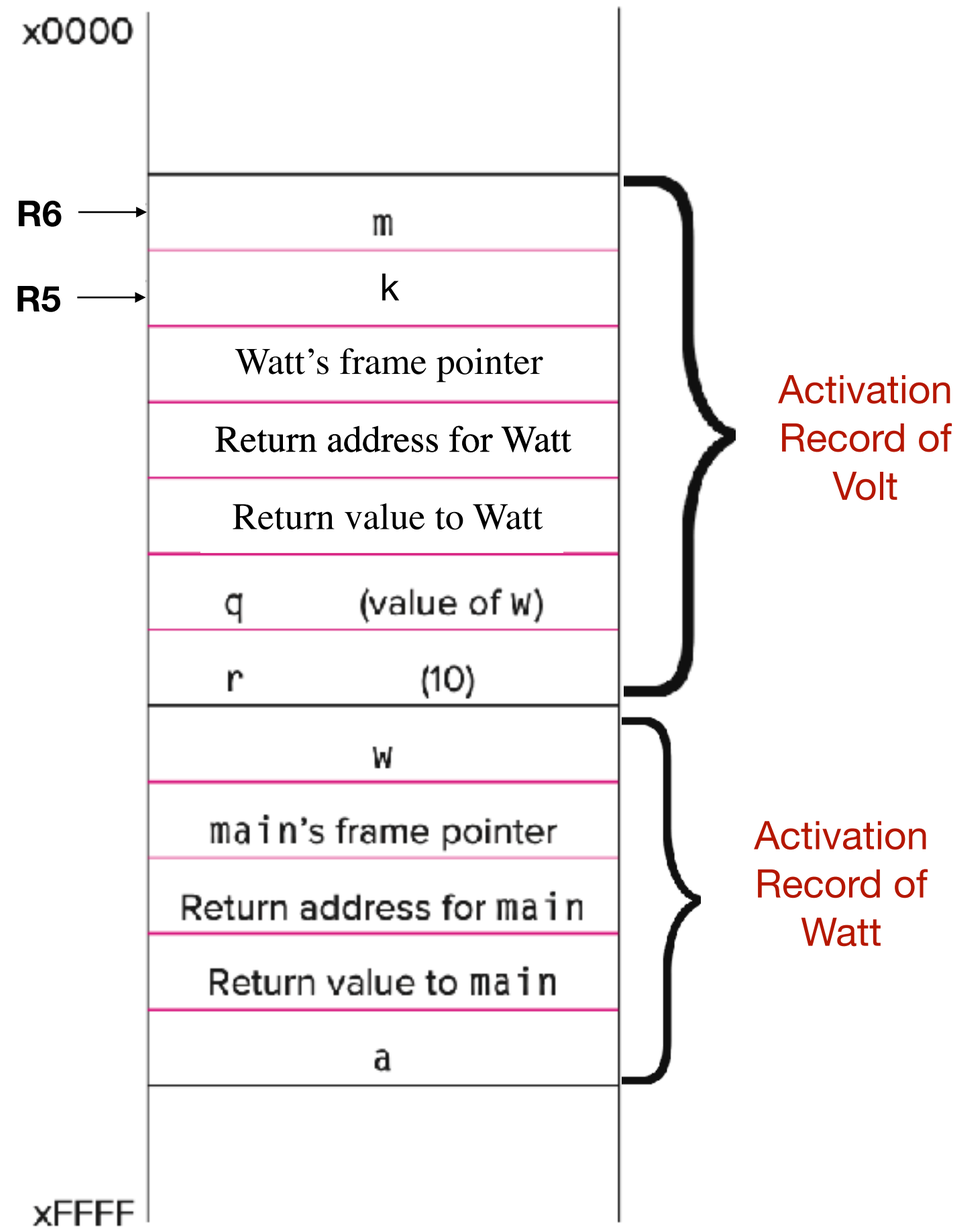
```c
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

R0
k



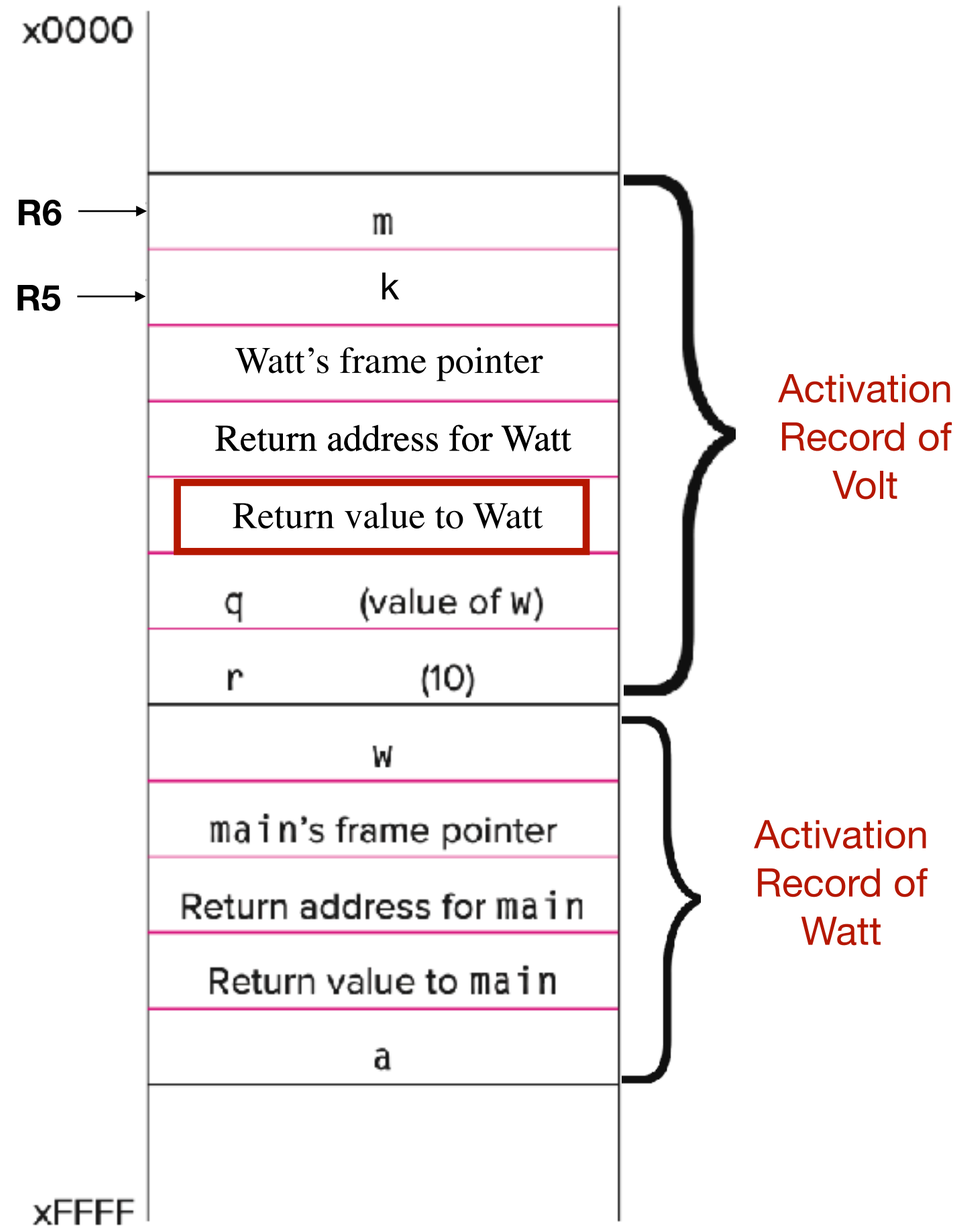
```c
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

R0
k



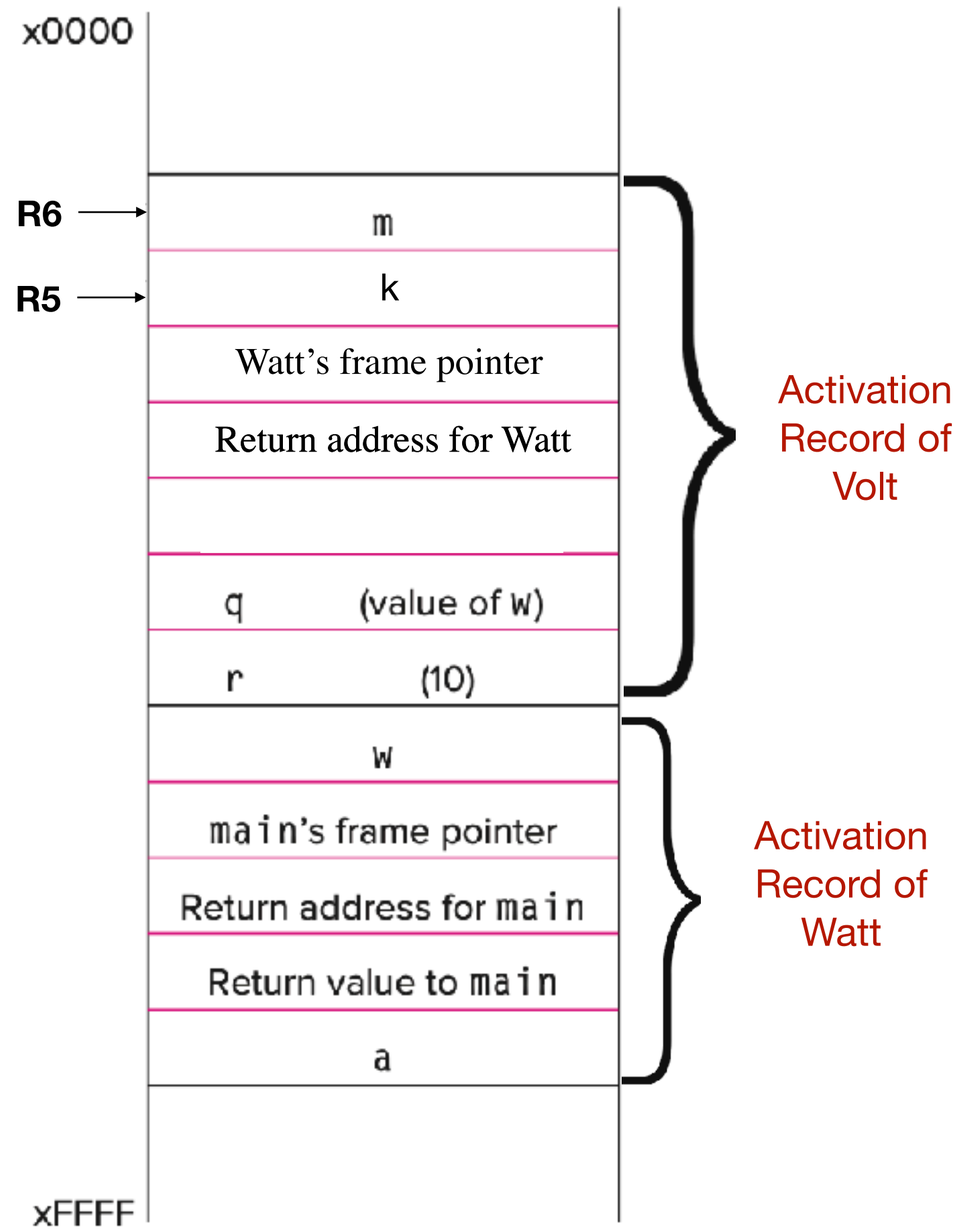
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

R0
k



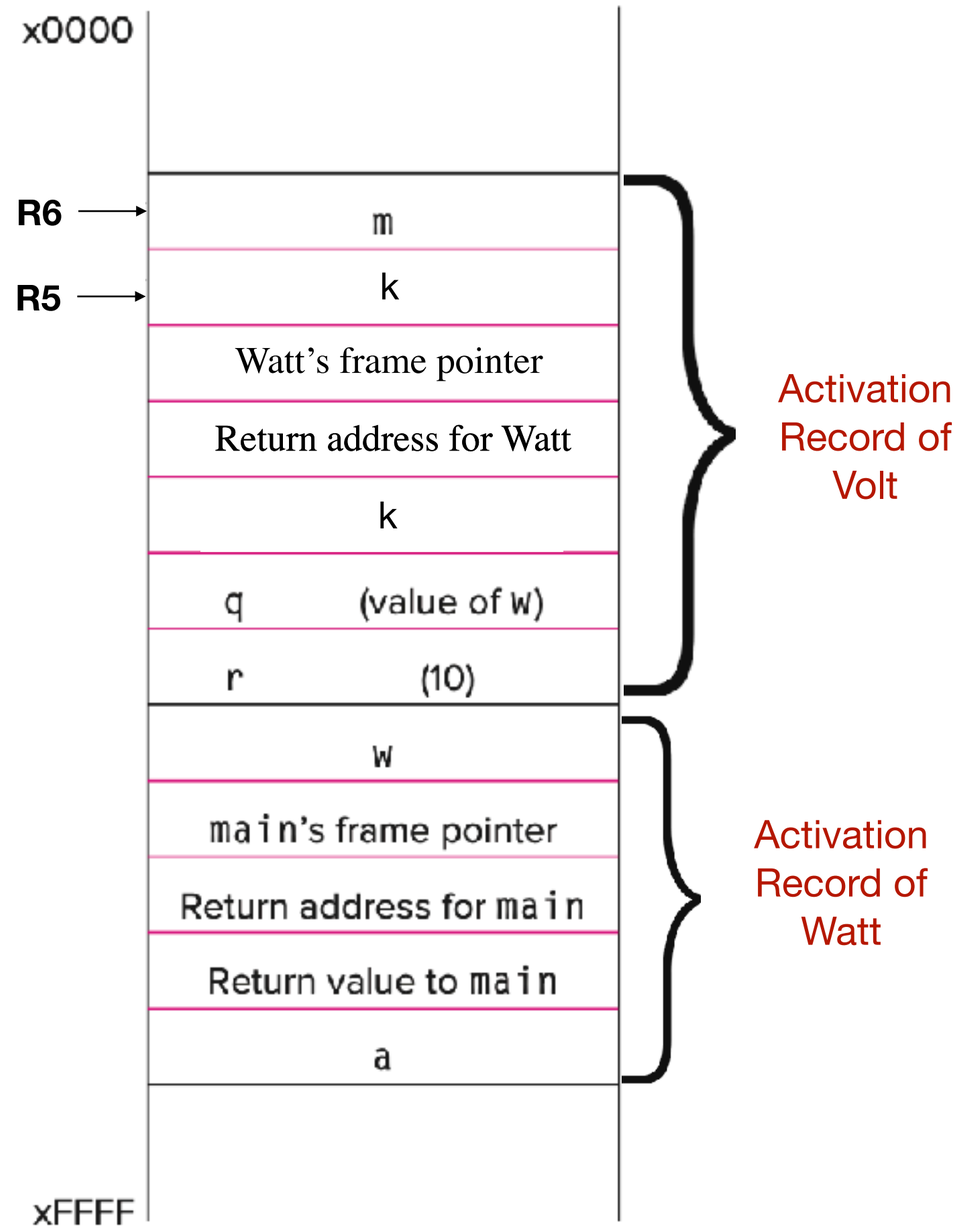
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

R0
k



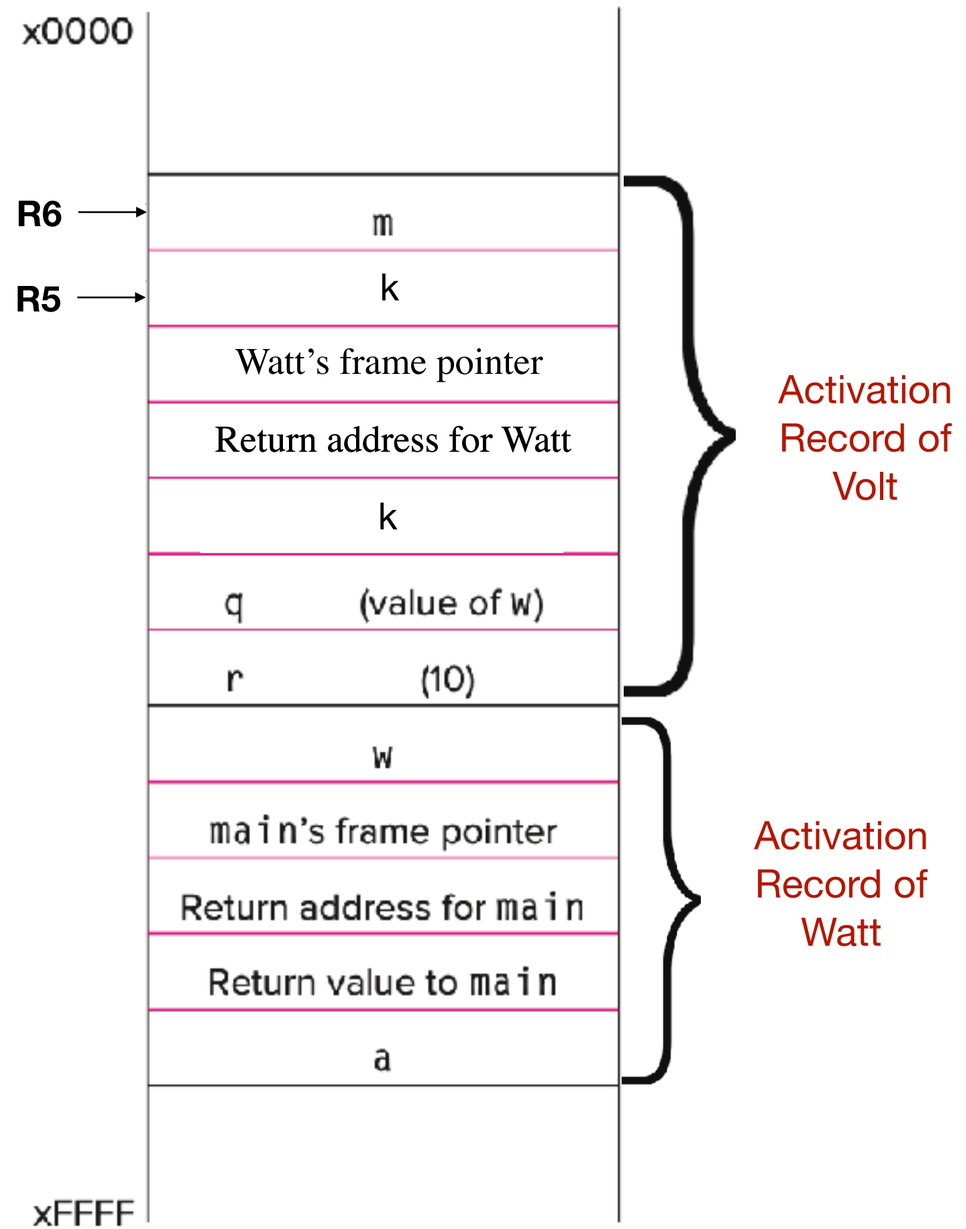
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

; pop local variables
```

R0
k



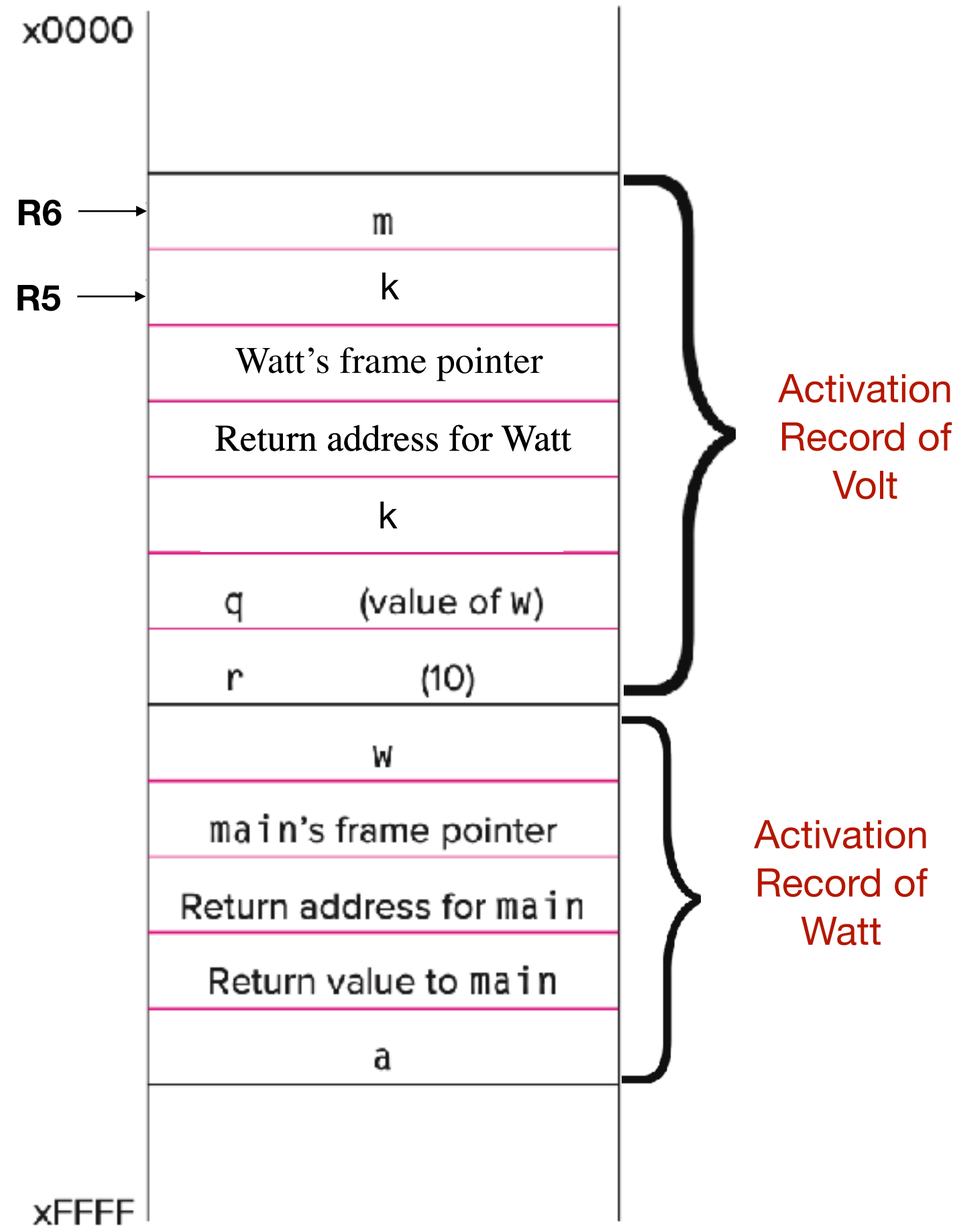
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

; pop local variables
ADD R6, R5, #1
```

R0
k



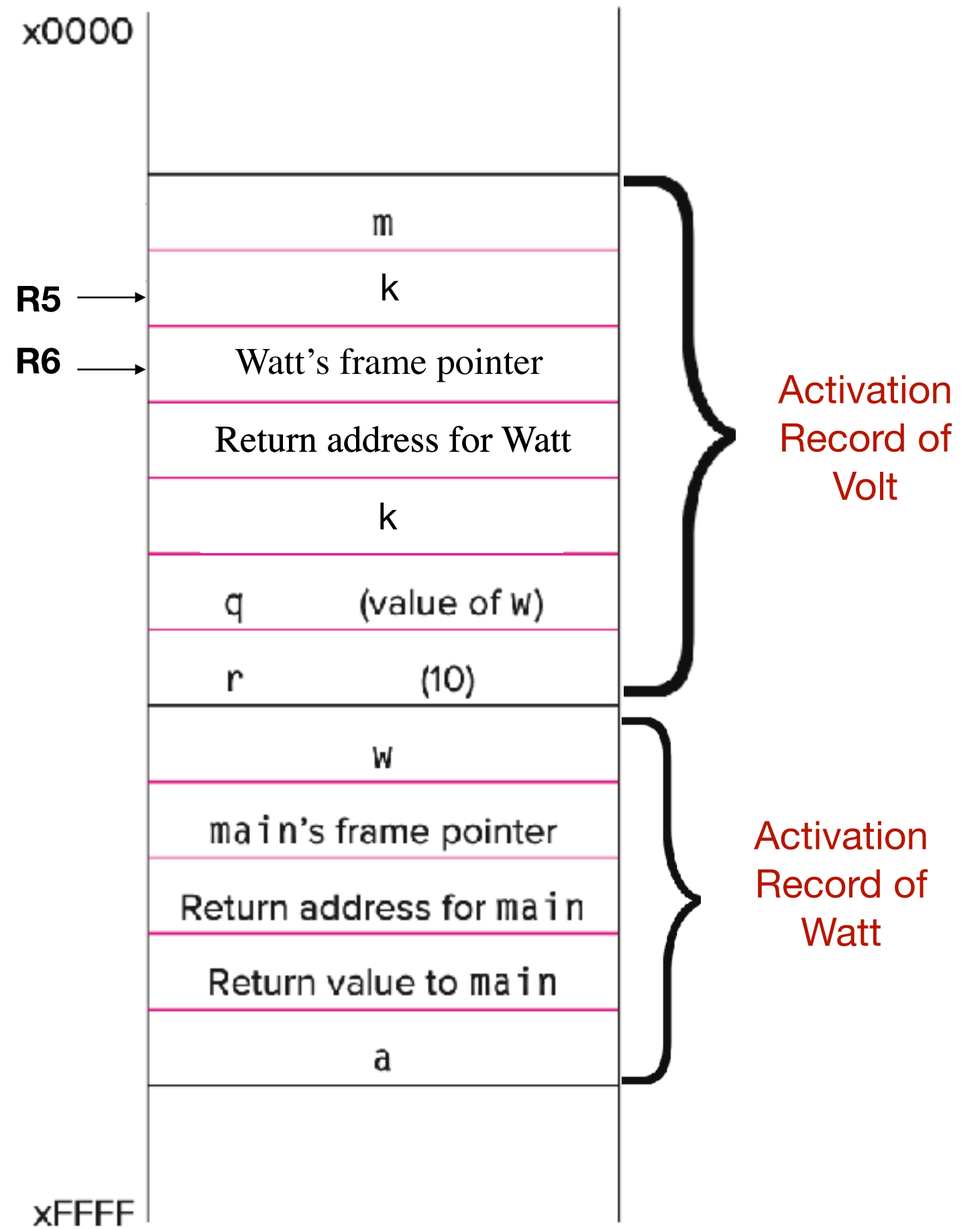
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

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

; pop local variables
ADD R6, R5, #1
```

R0
k



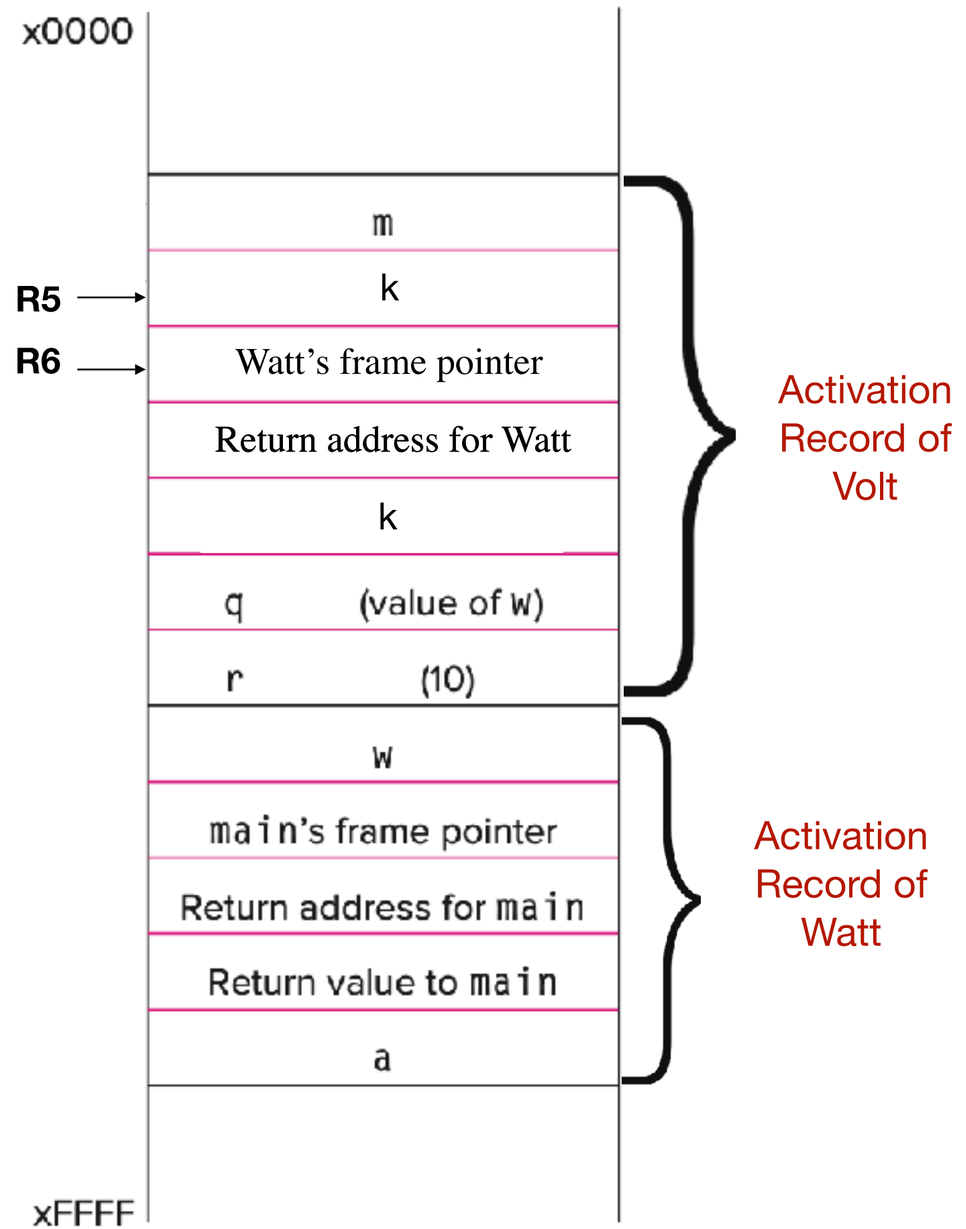
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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)
```

R0
k



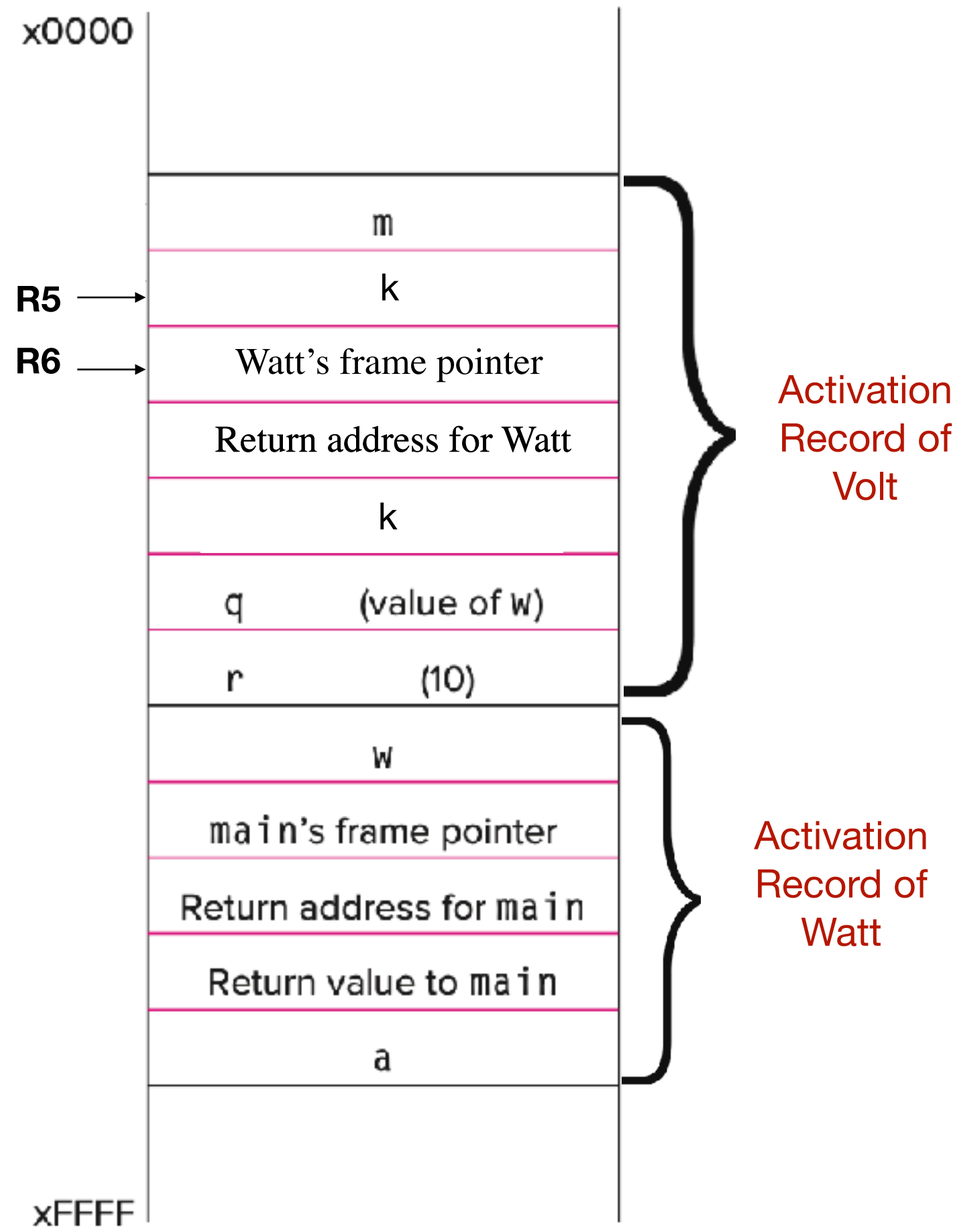
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k



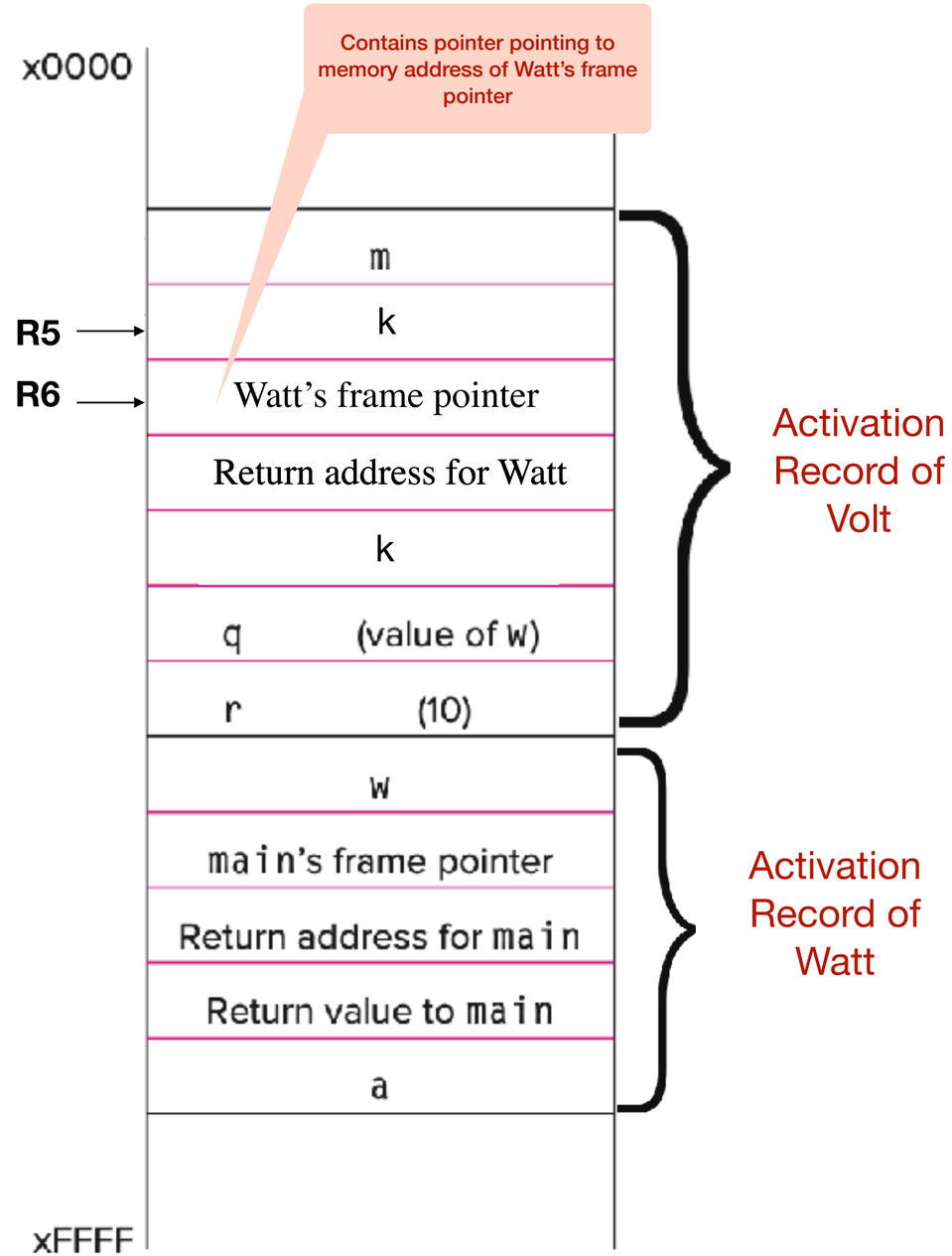
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k



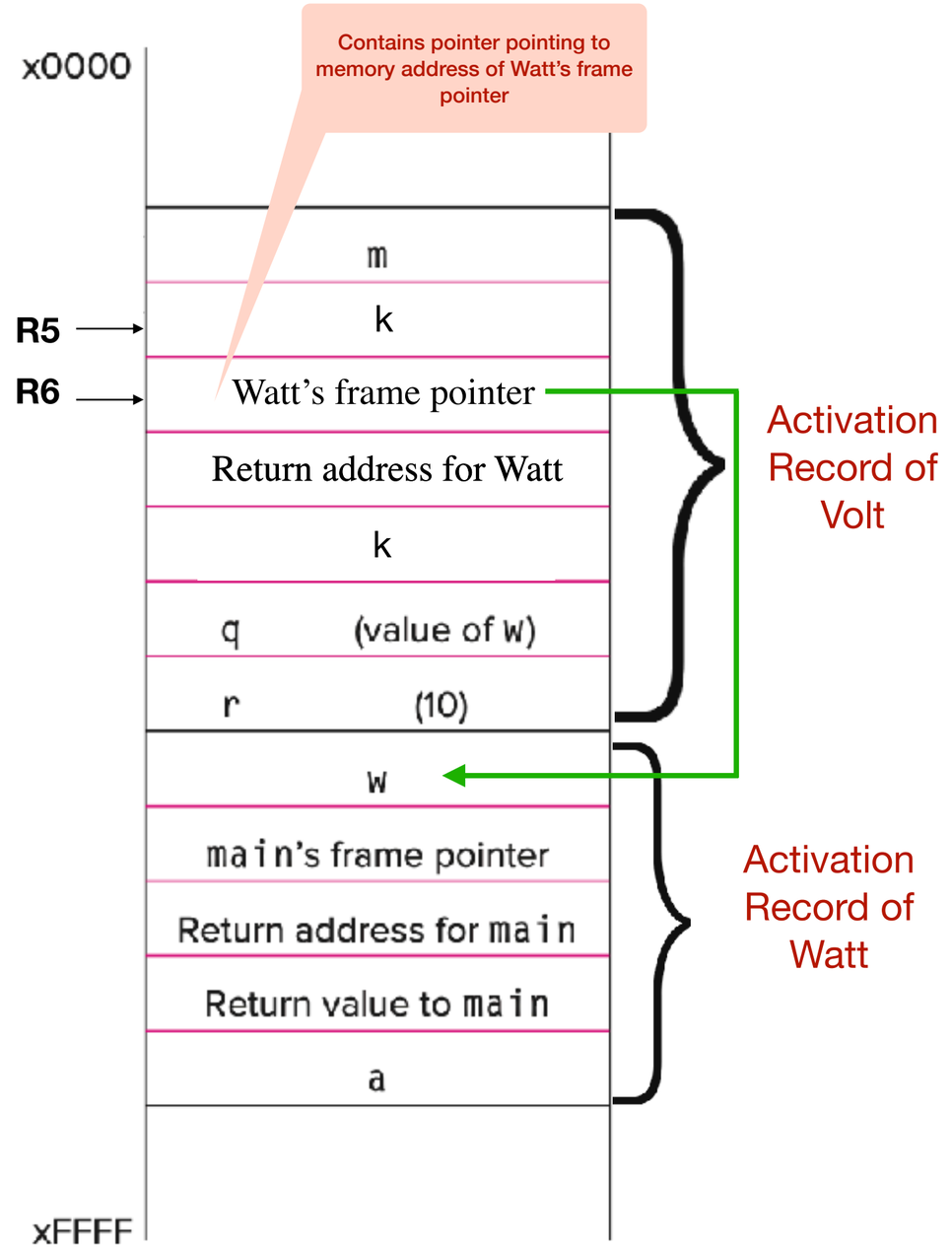
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k



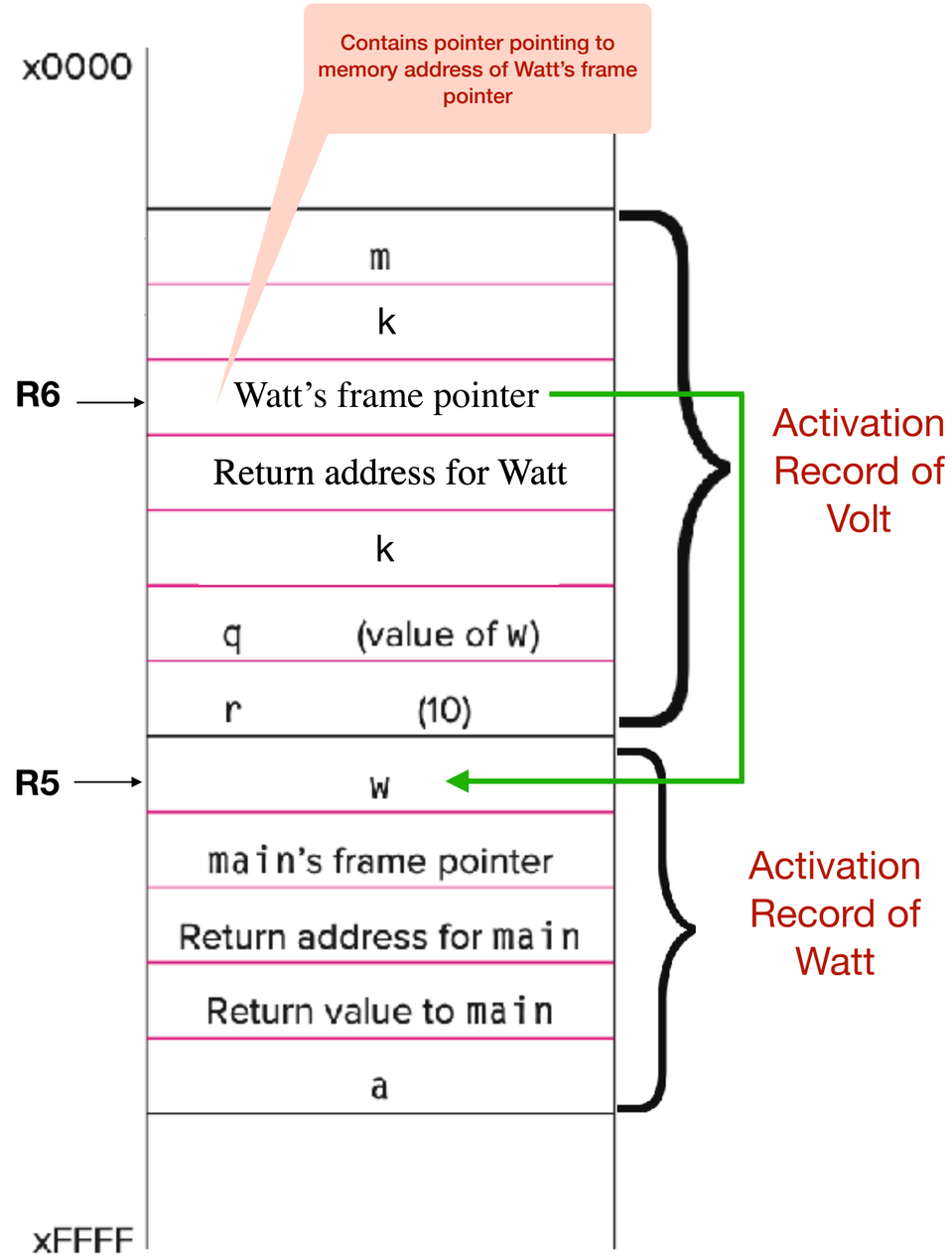
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k



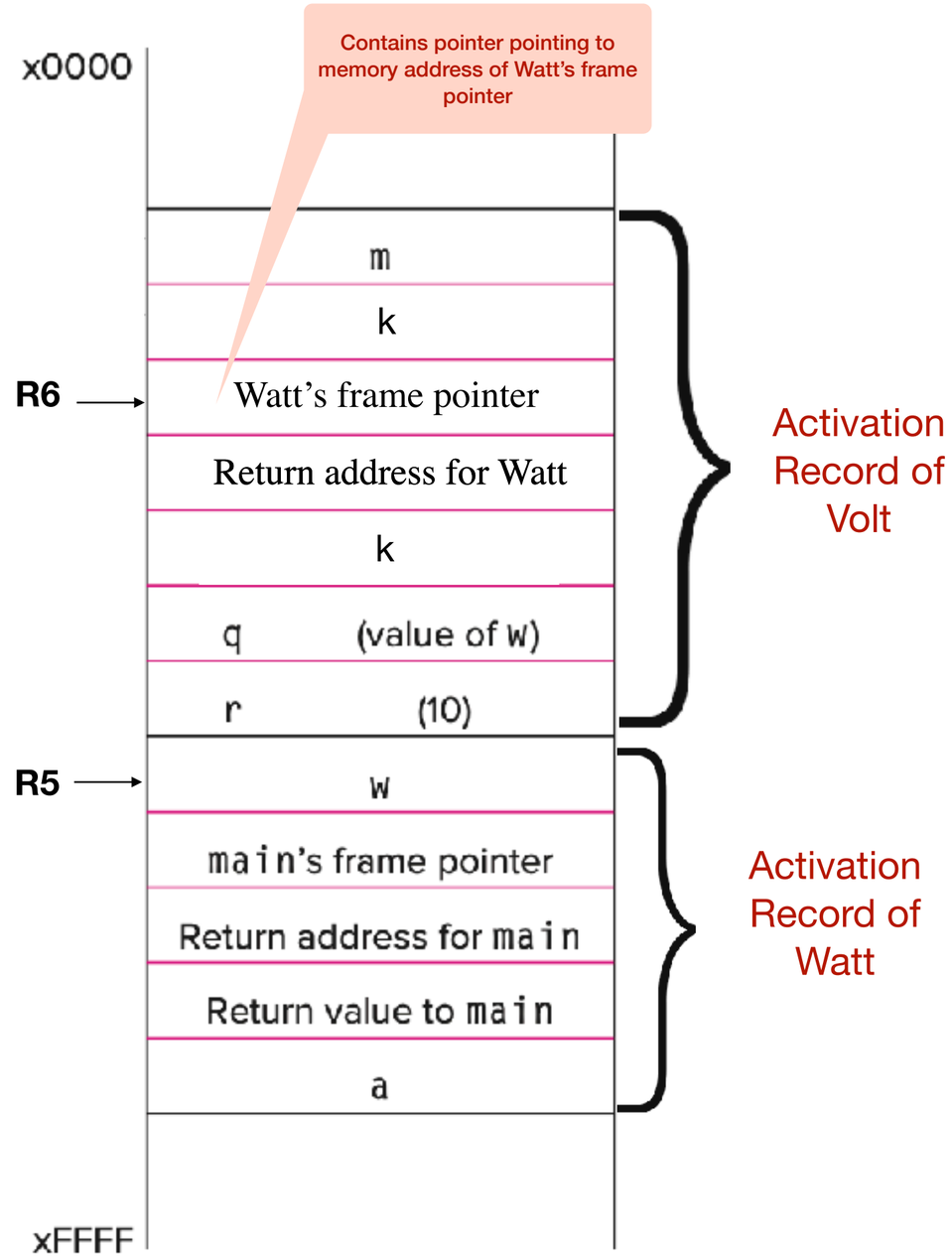
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k

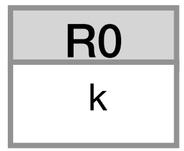


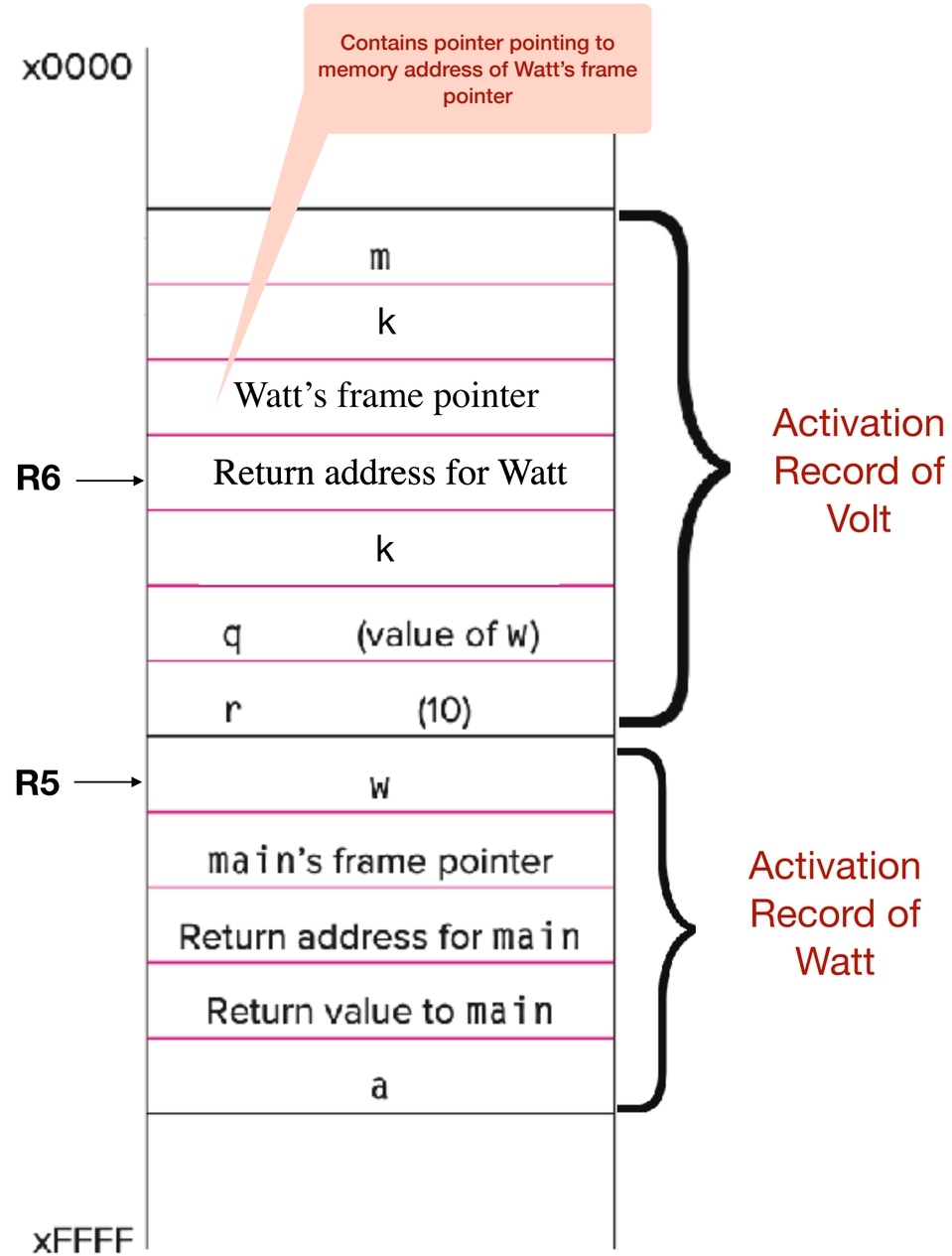
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```



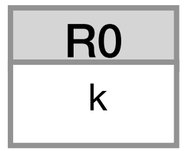


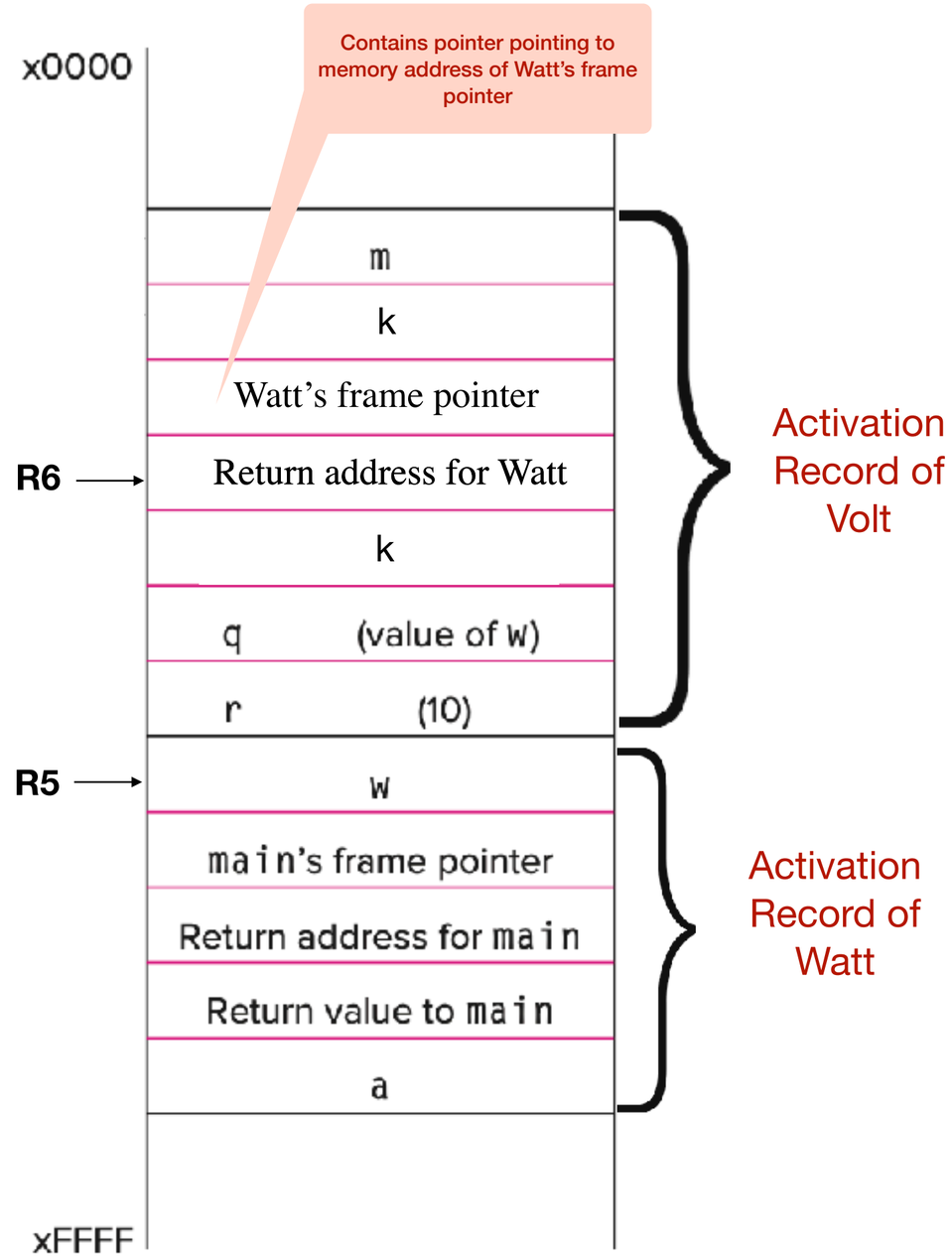
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```





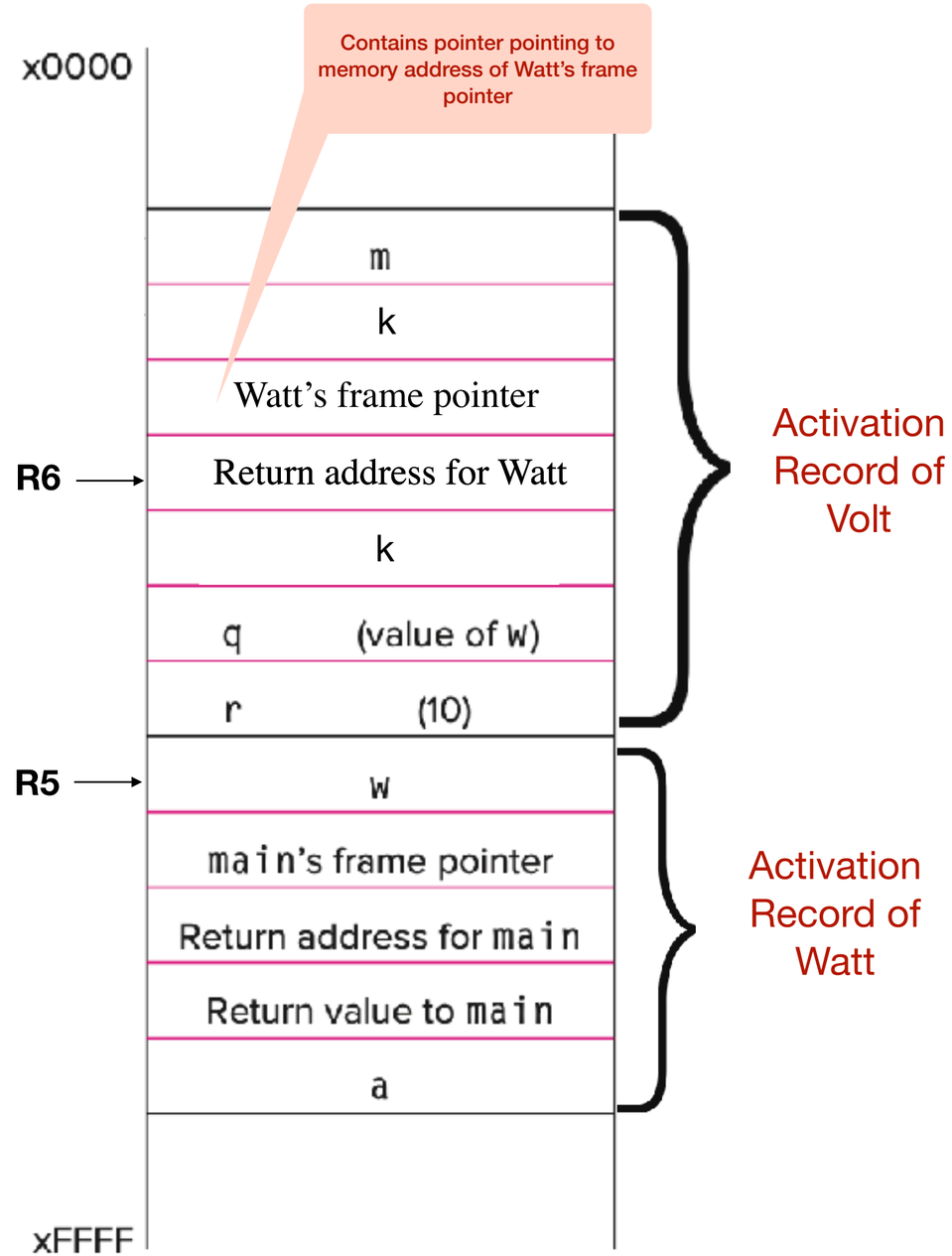
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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)
```

R0
k



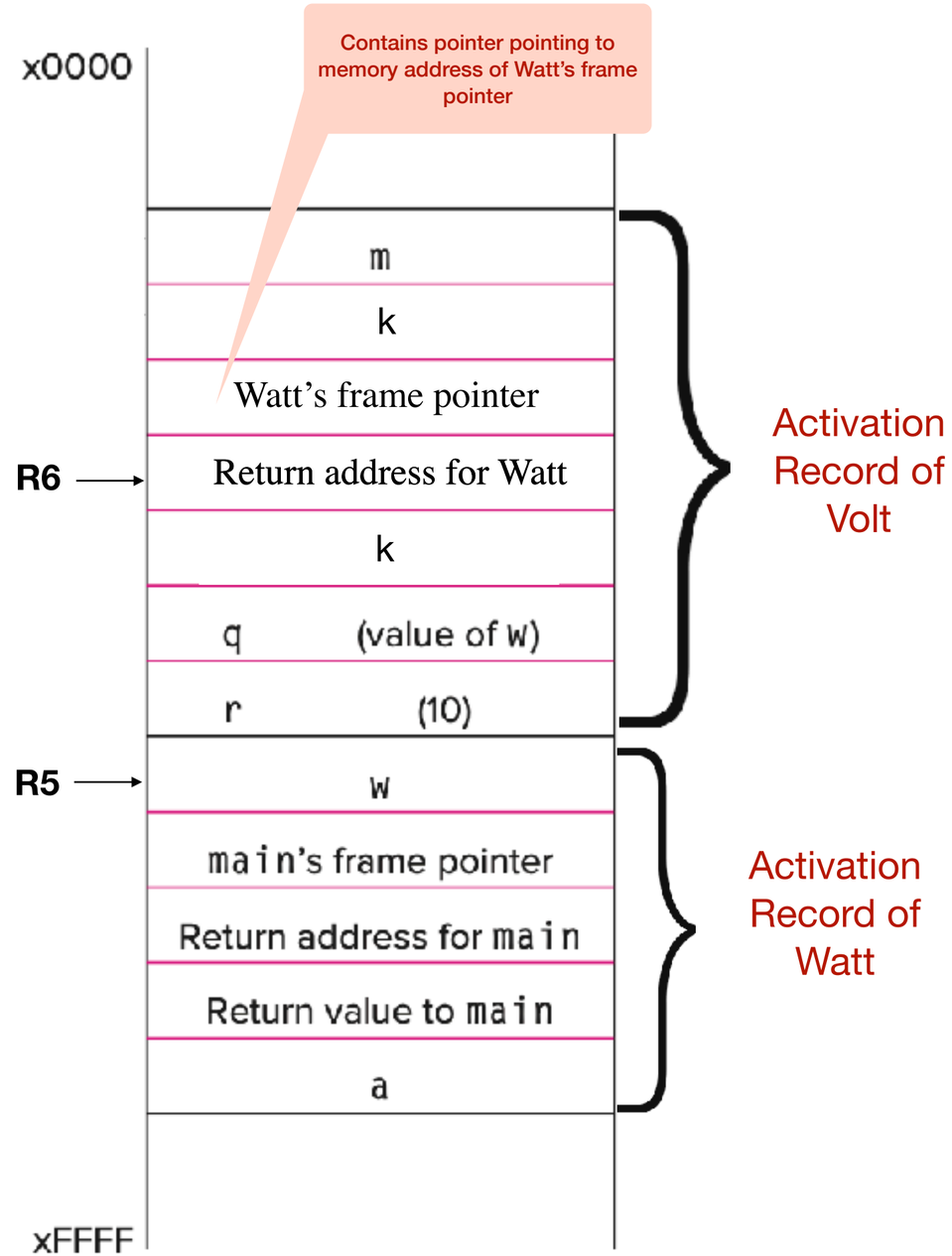
```
int Volt(int q, int r)
{
        int k;
        int m;
        ...
        return k;
}
```

LC-3 Implementation

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

```
; 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
```

R0
k



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

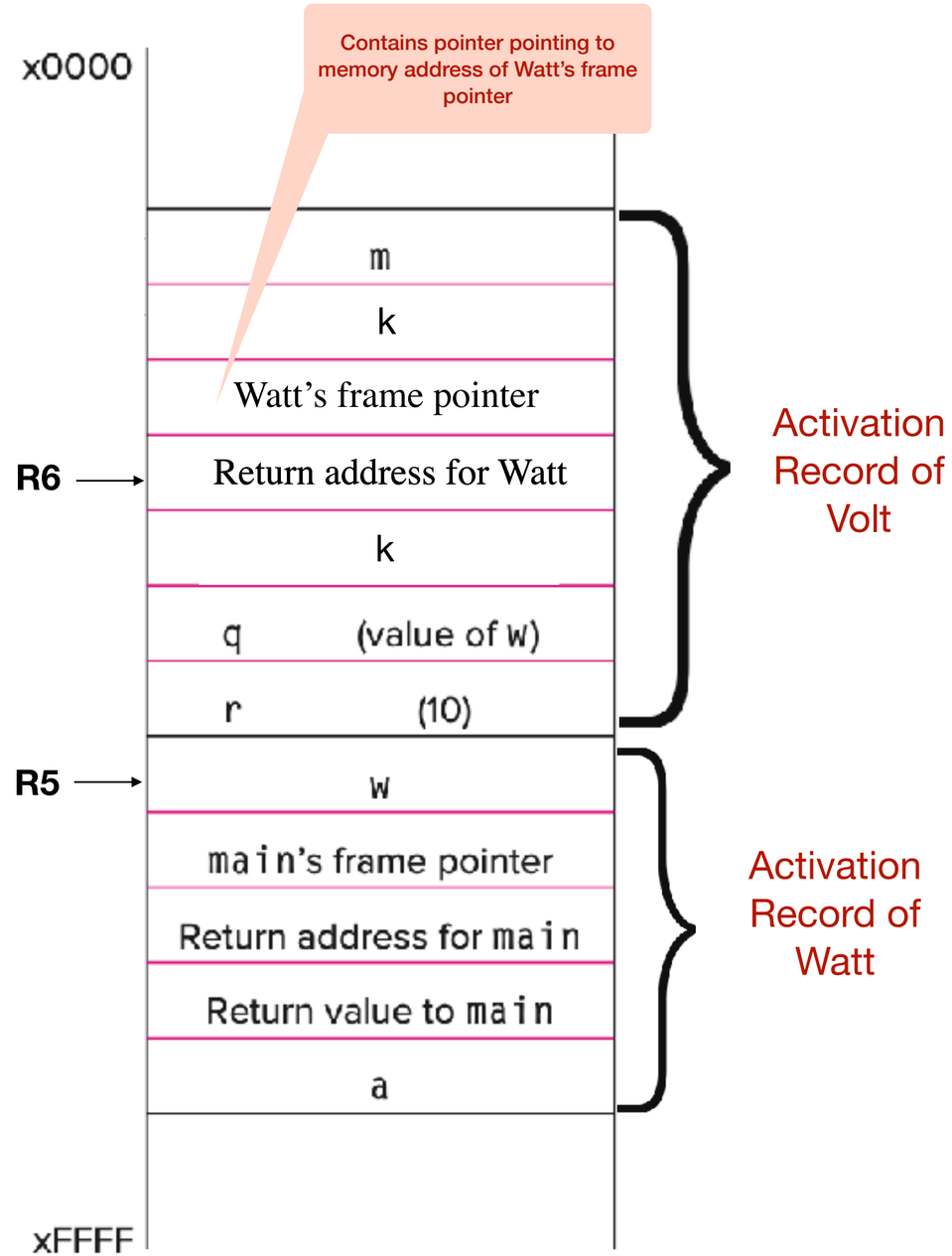
LC-3 Implementation

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

```
; 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
```

R0
k

R7



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

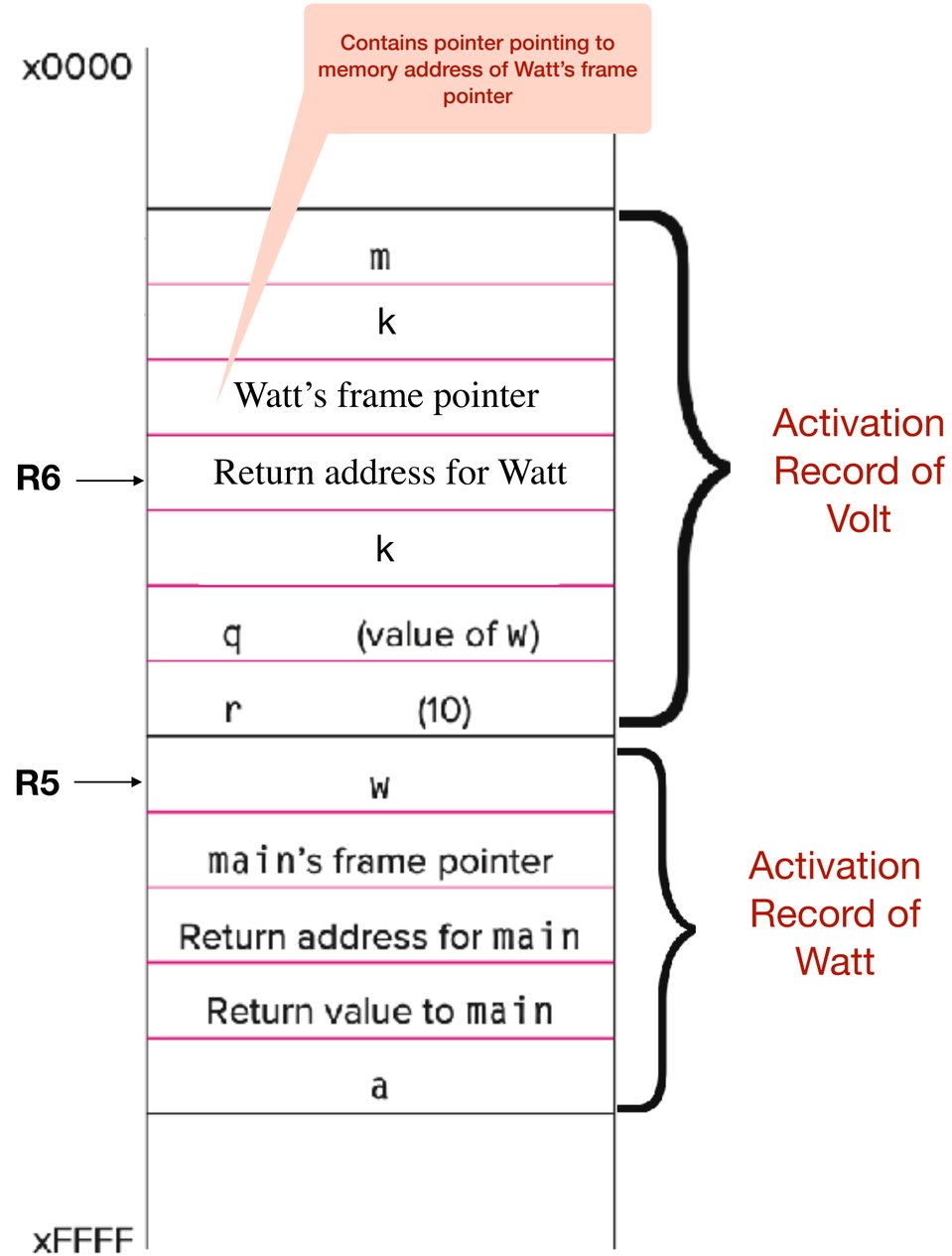
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

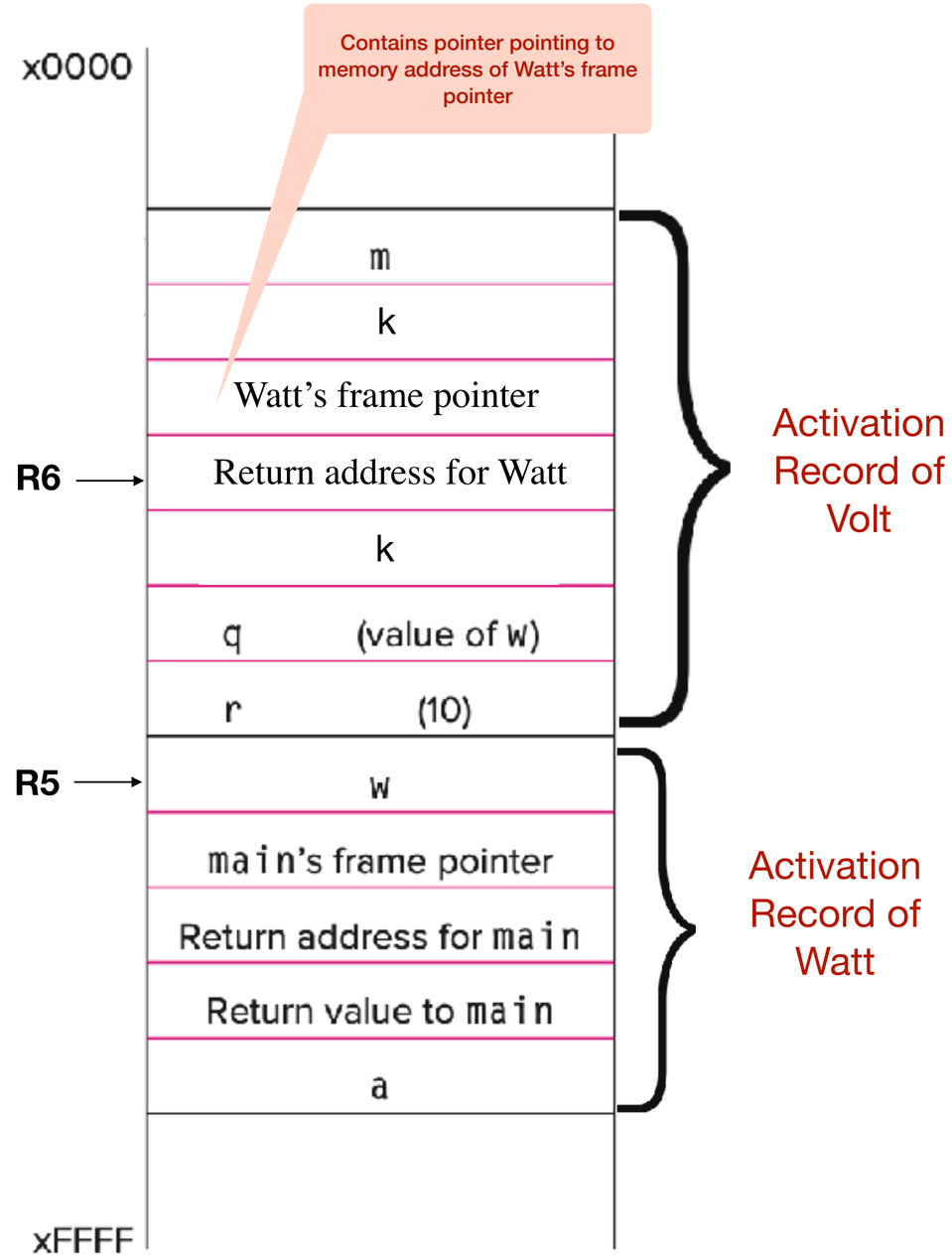
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

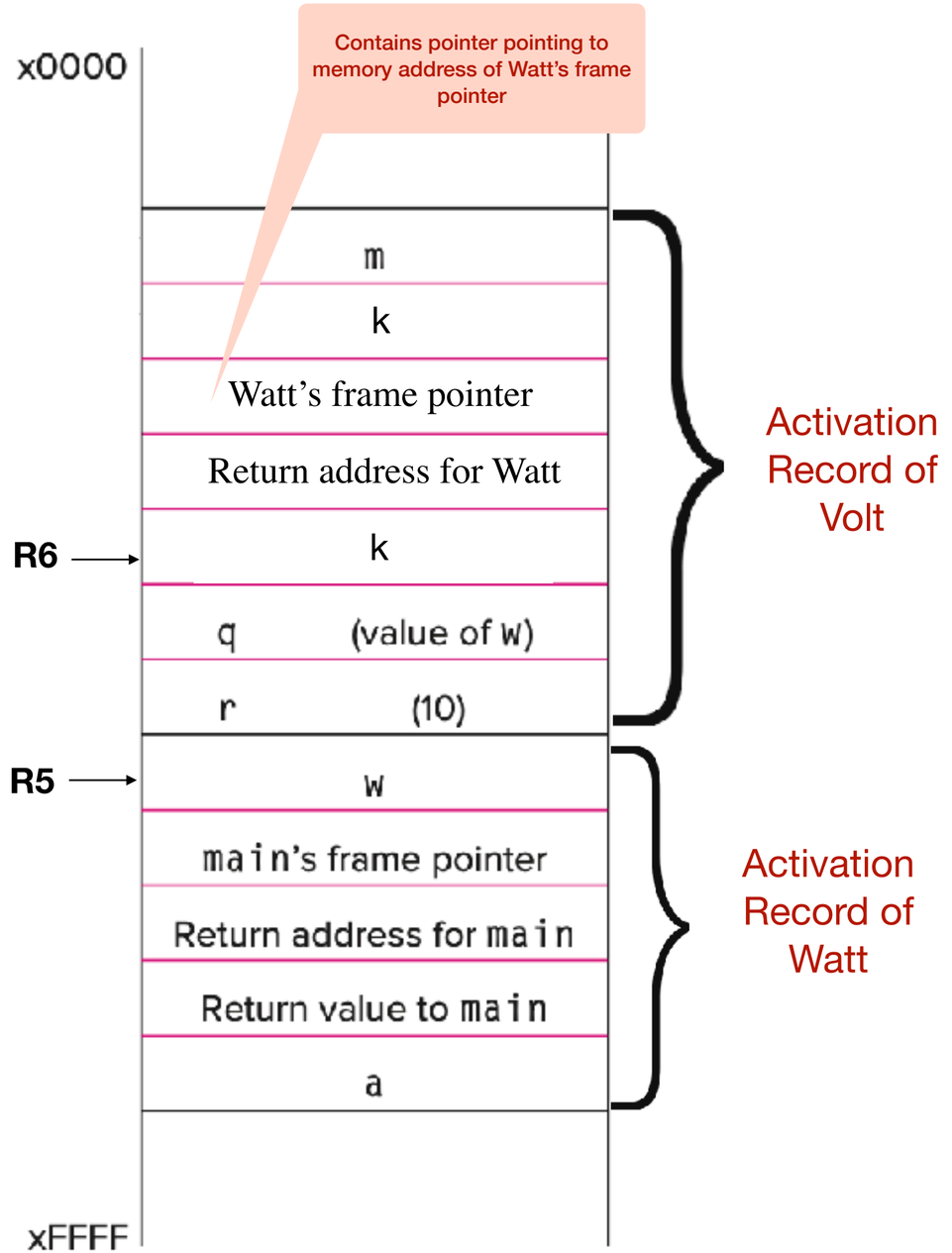
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

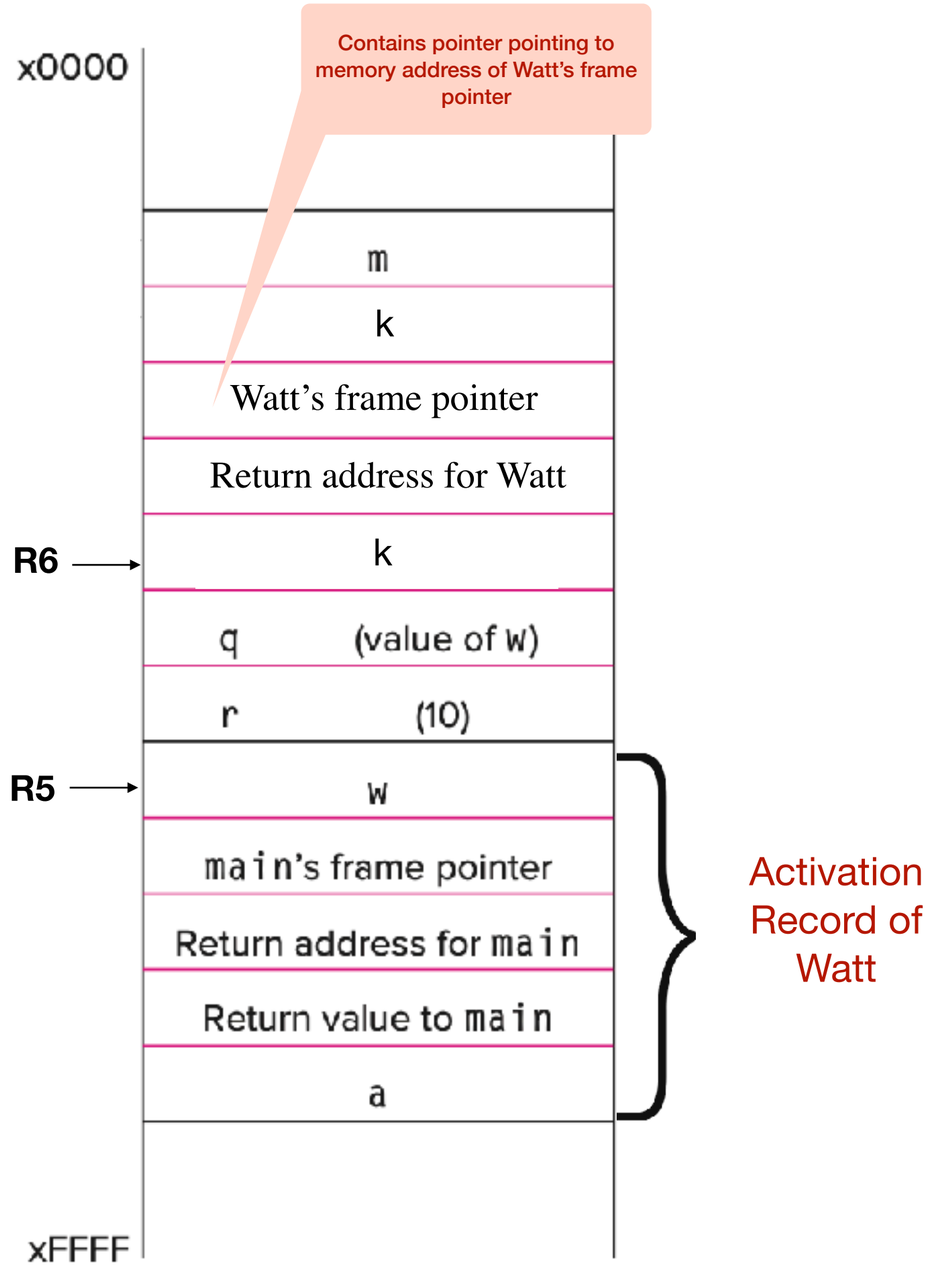
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

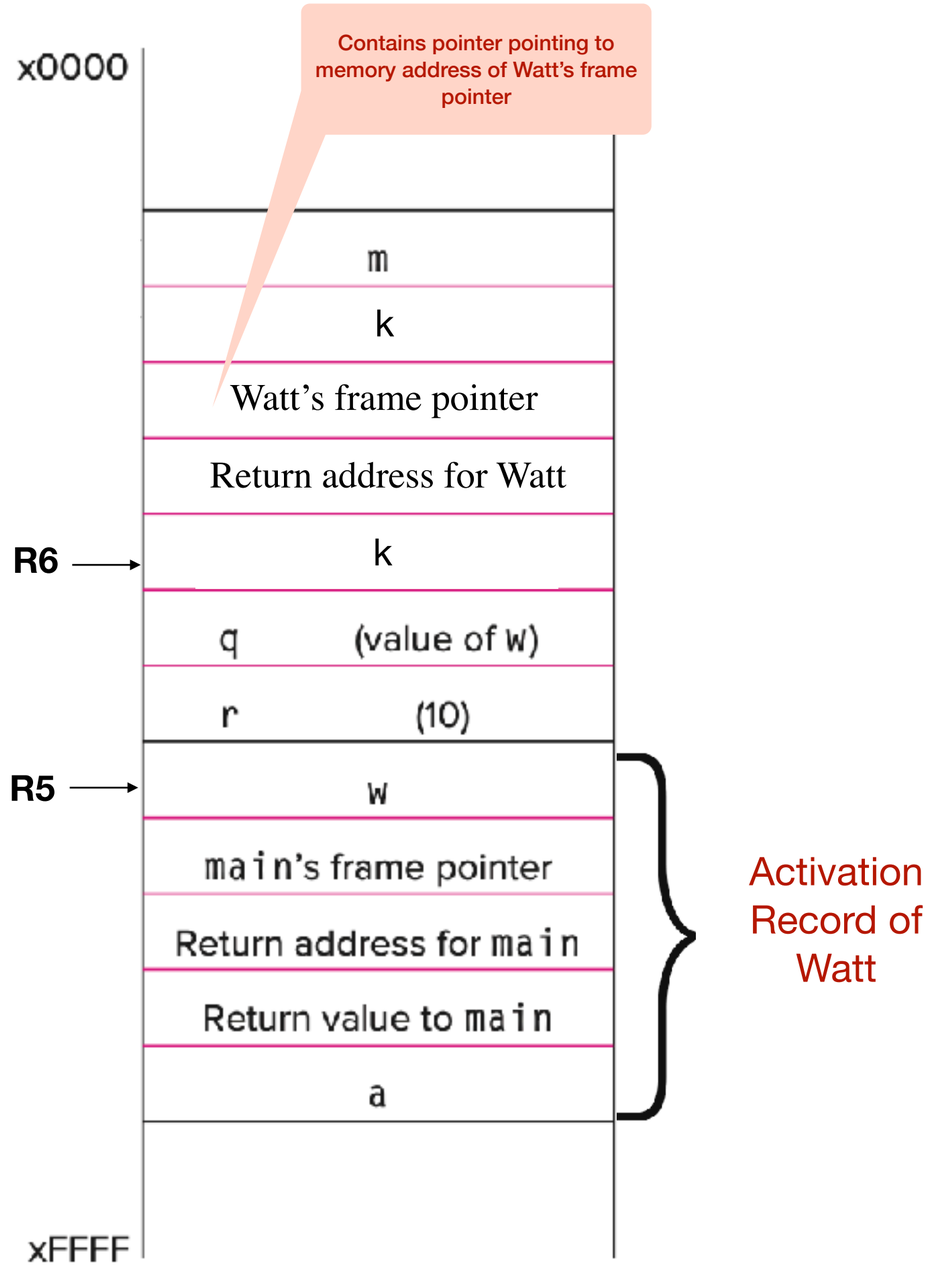
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

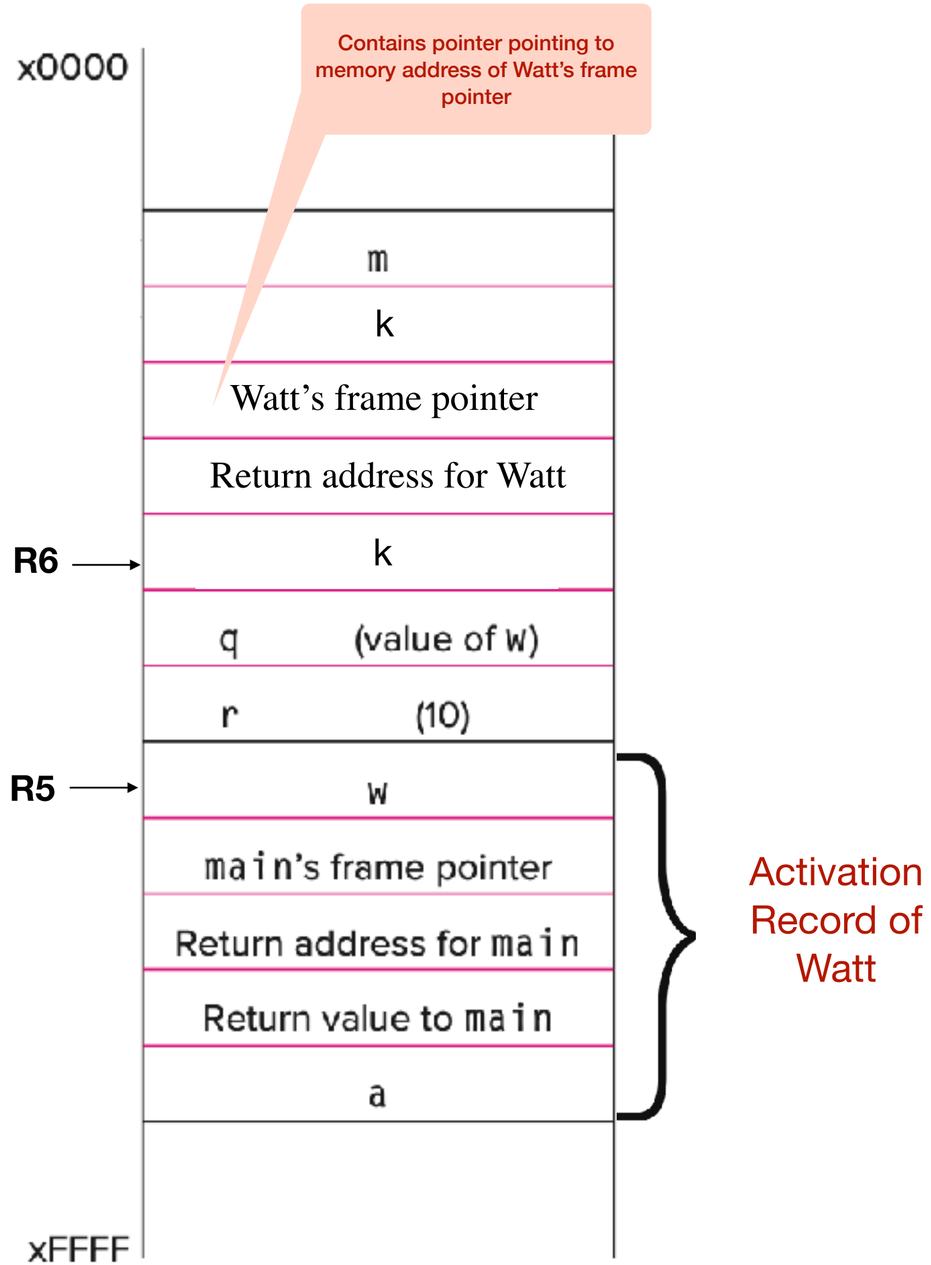
LC-3 Implementation

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

```
; 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
```

R0
k

R7
Return address for Watt



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

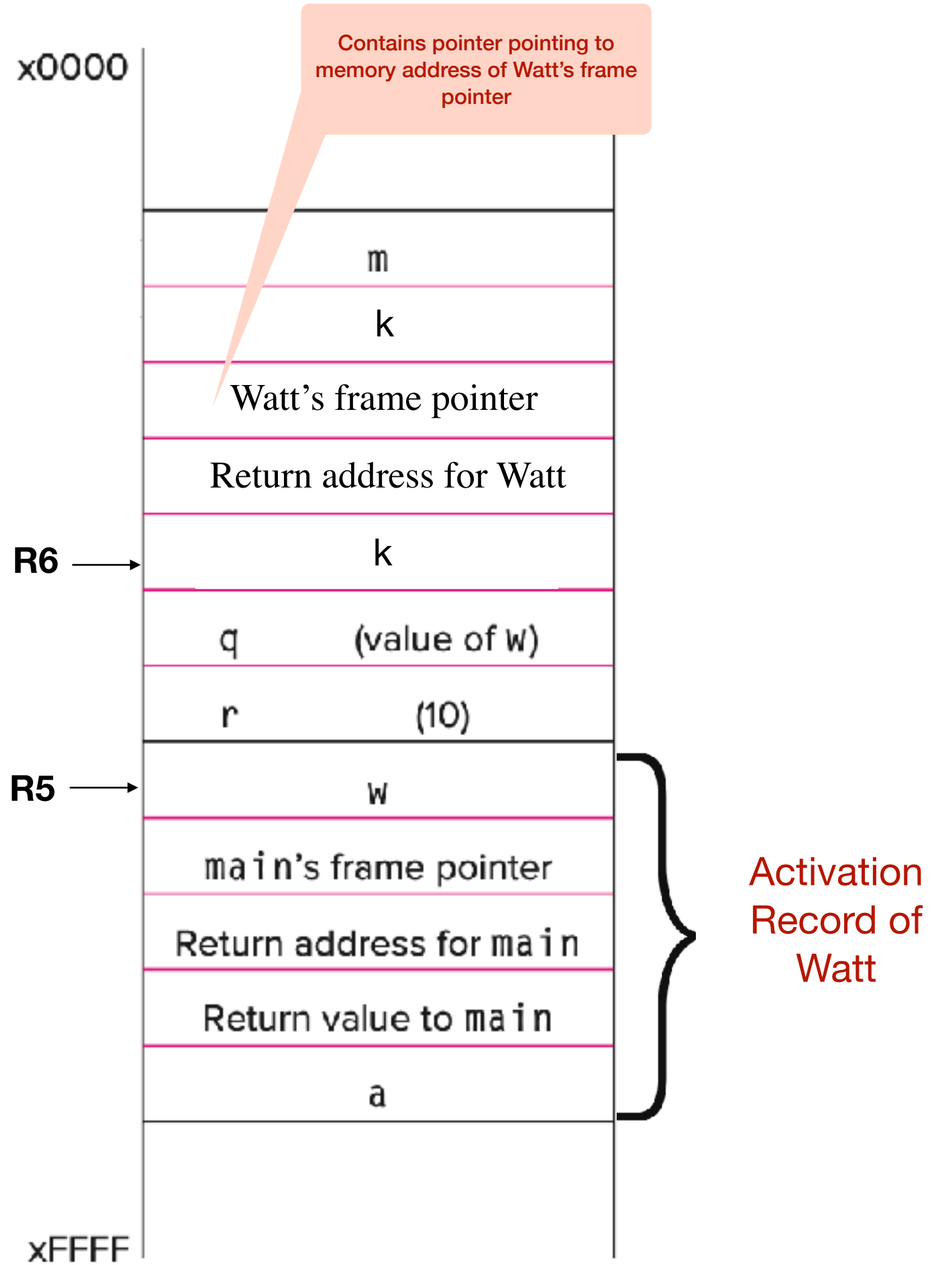
LC-3 Implementation

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
```

R0
k

R7
Return address for Watt



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

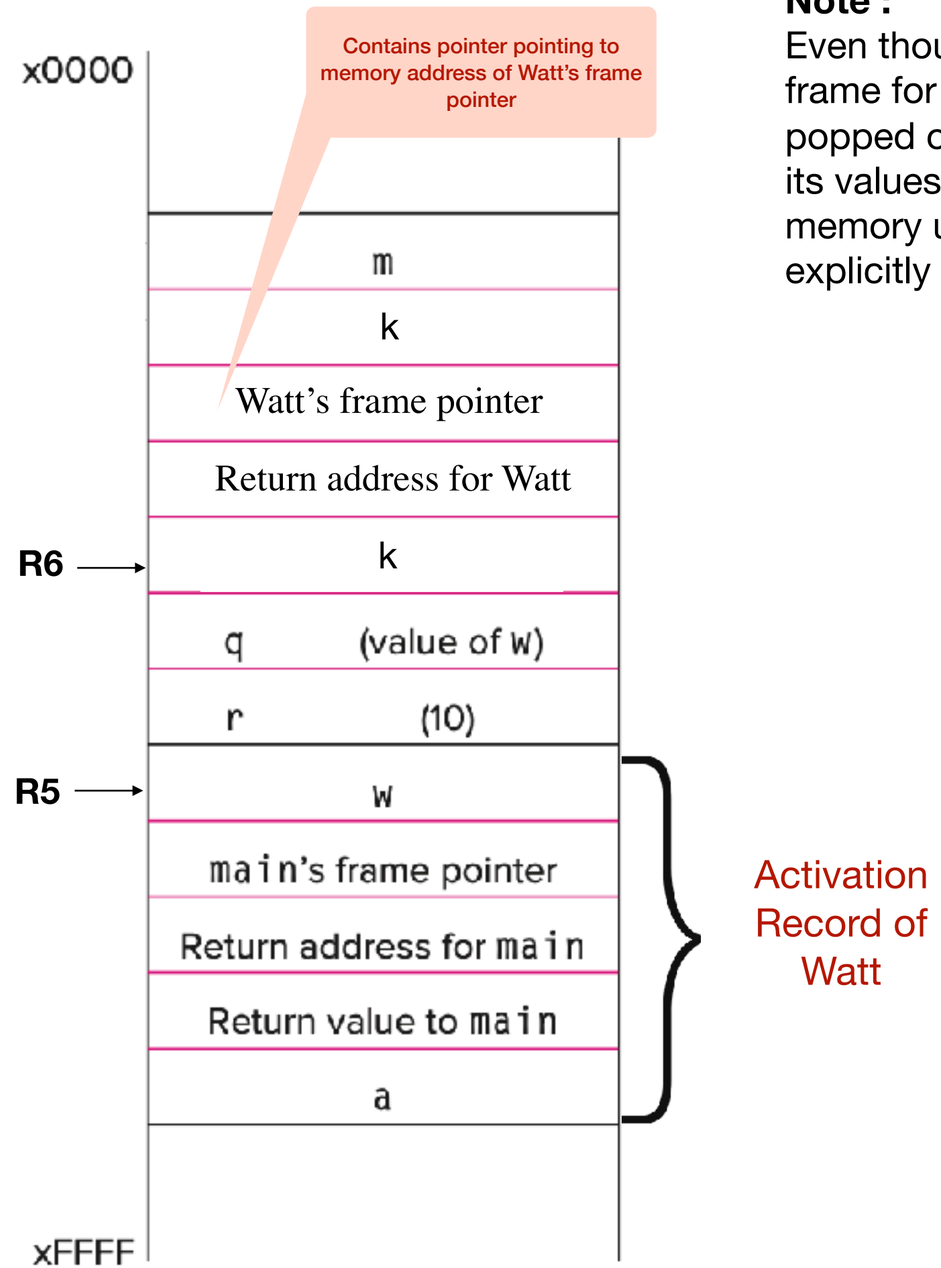
LC-3 Implementation

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
```

R0
k

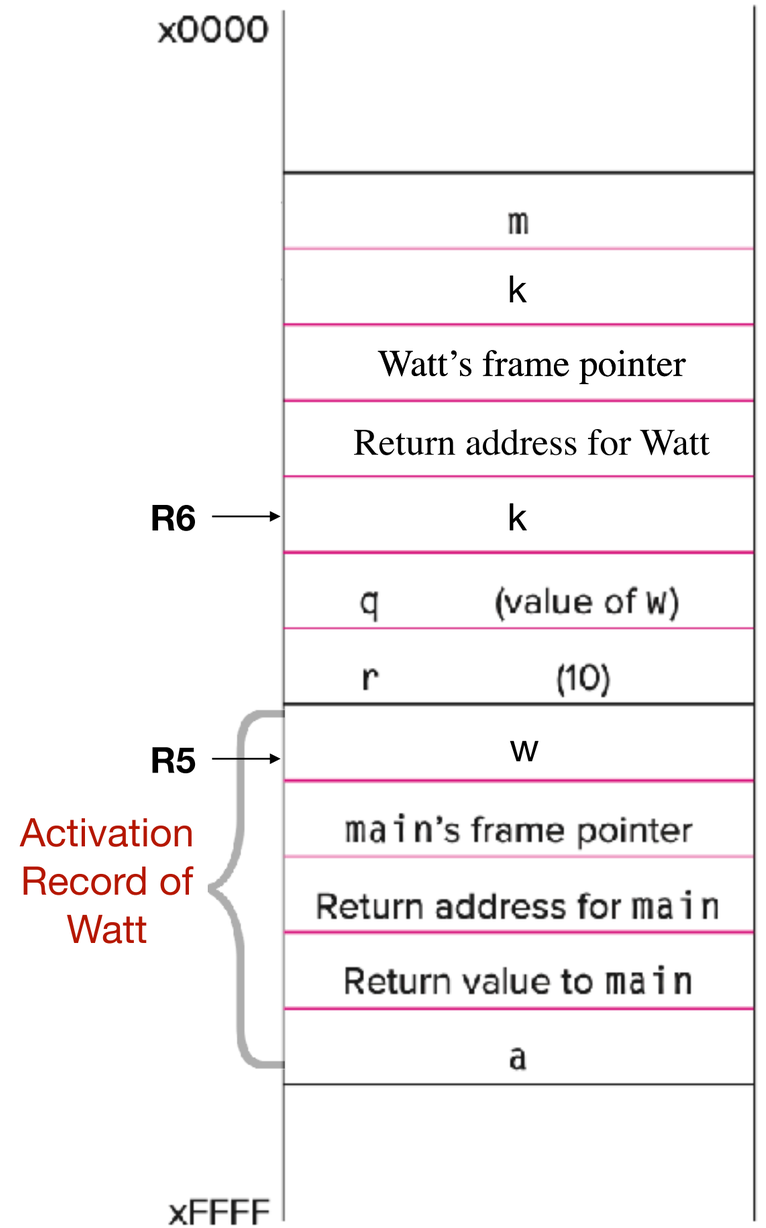
R7
Return address for Watt



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

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

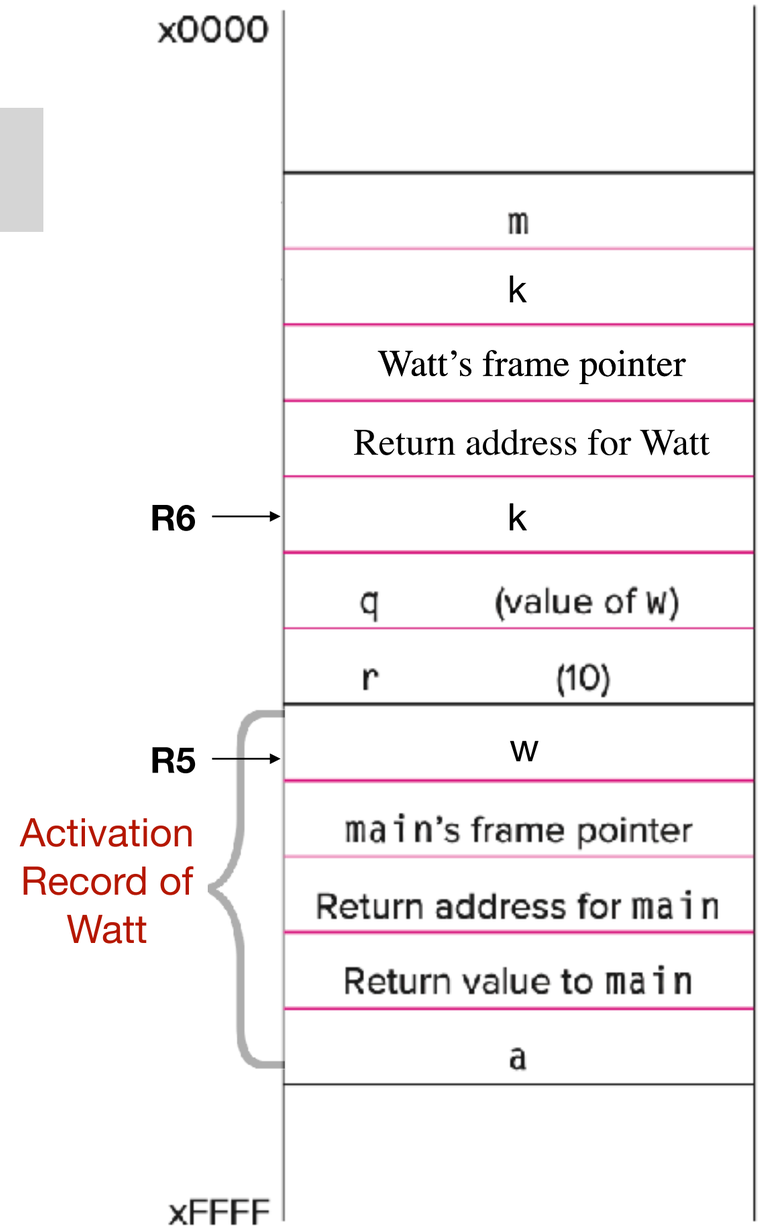
LC-3 Implementation



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

LC-3 Implementation

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

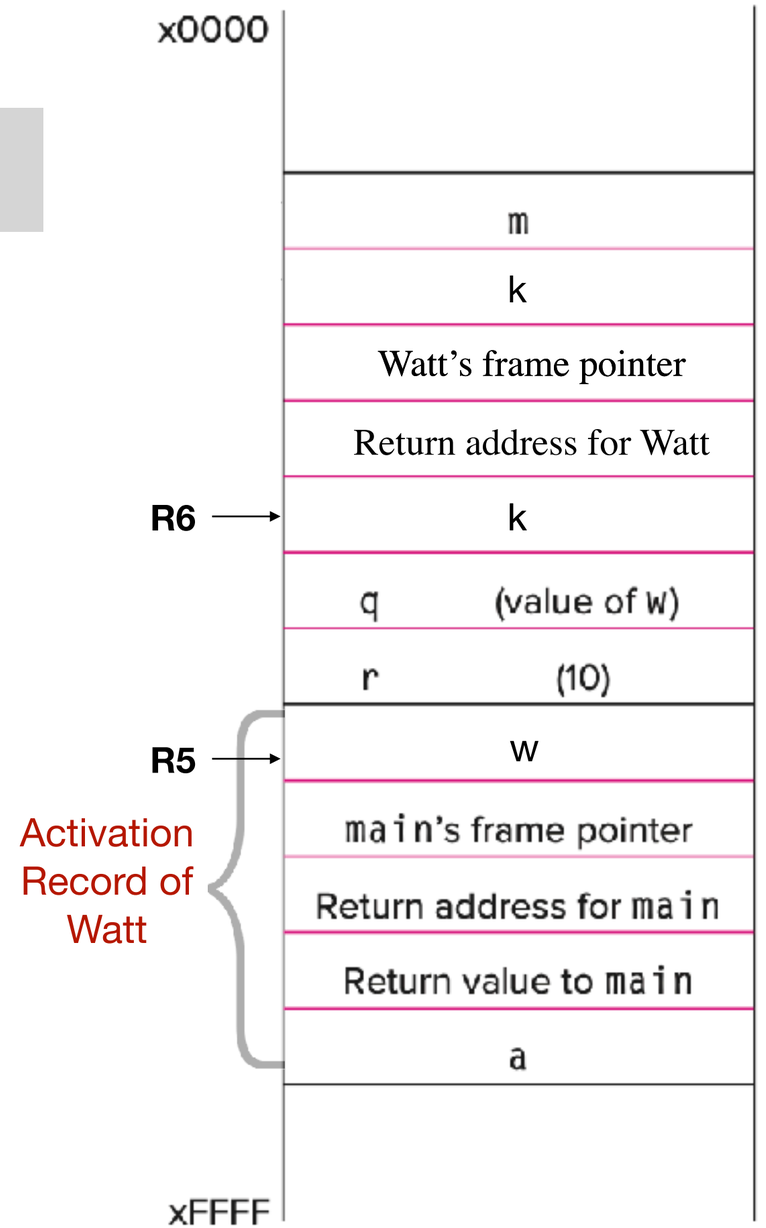


```
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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

```
JSR VOLT
```

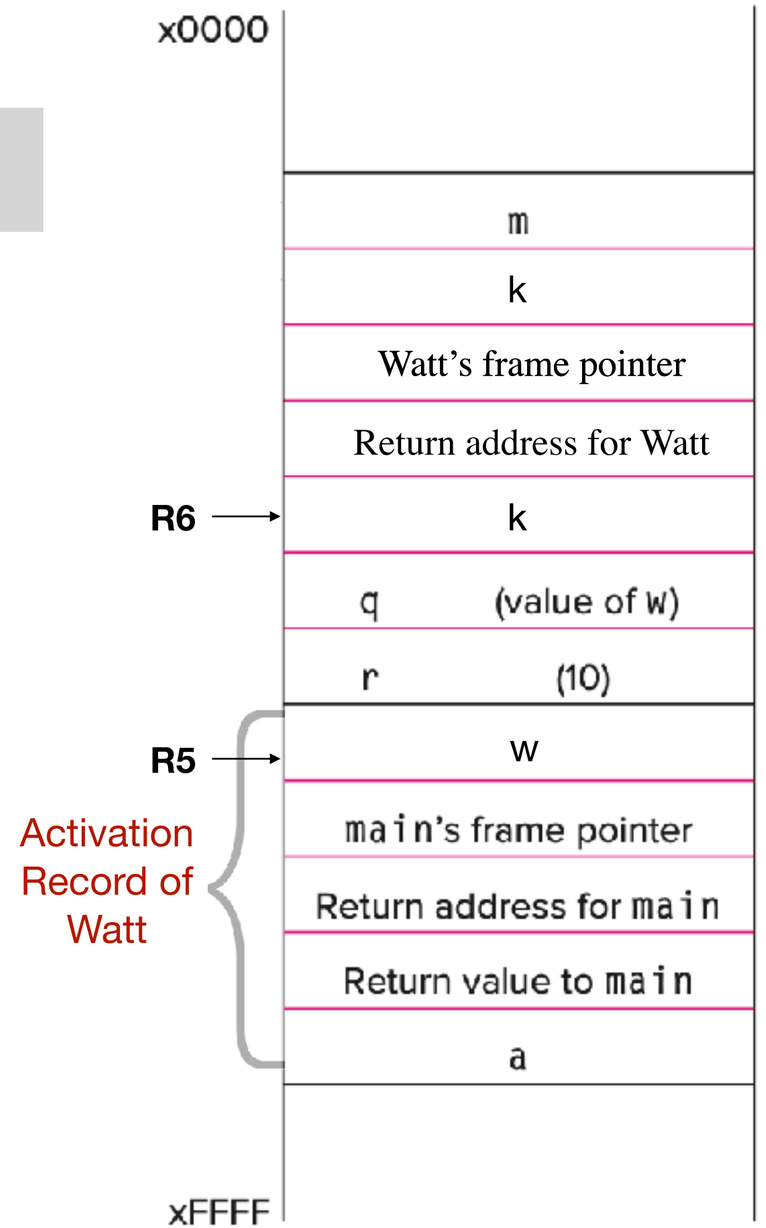


```
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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

```
JSR VOLT
; load return value (top of stack)
```

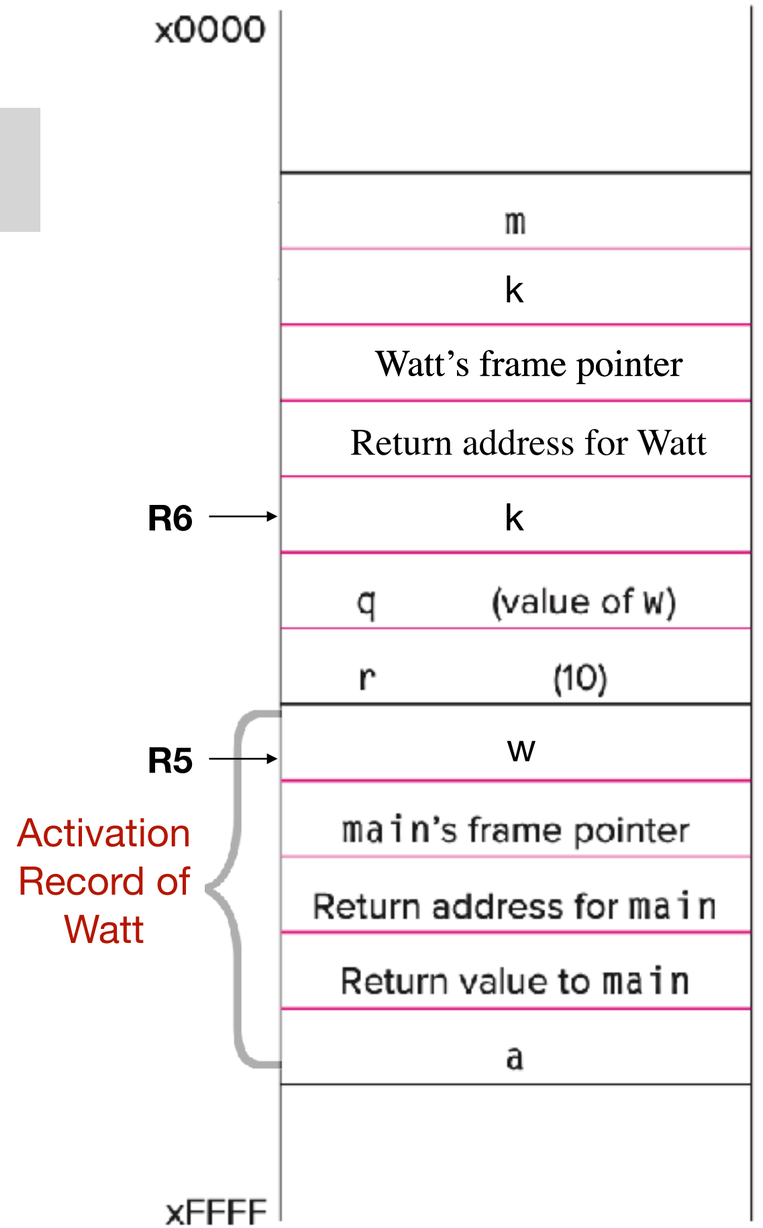


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

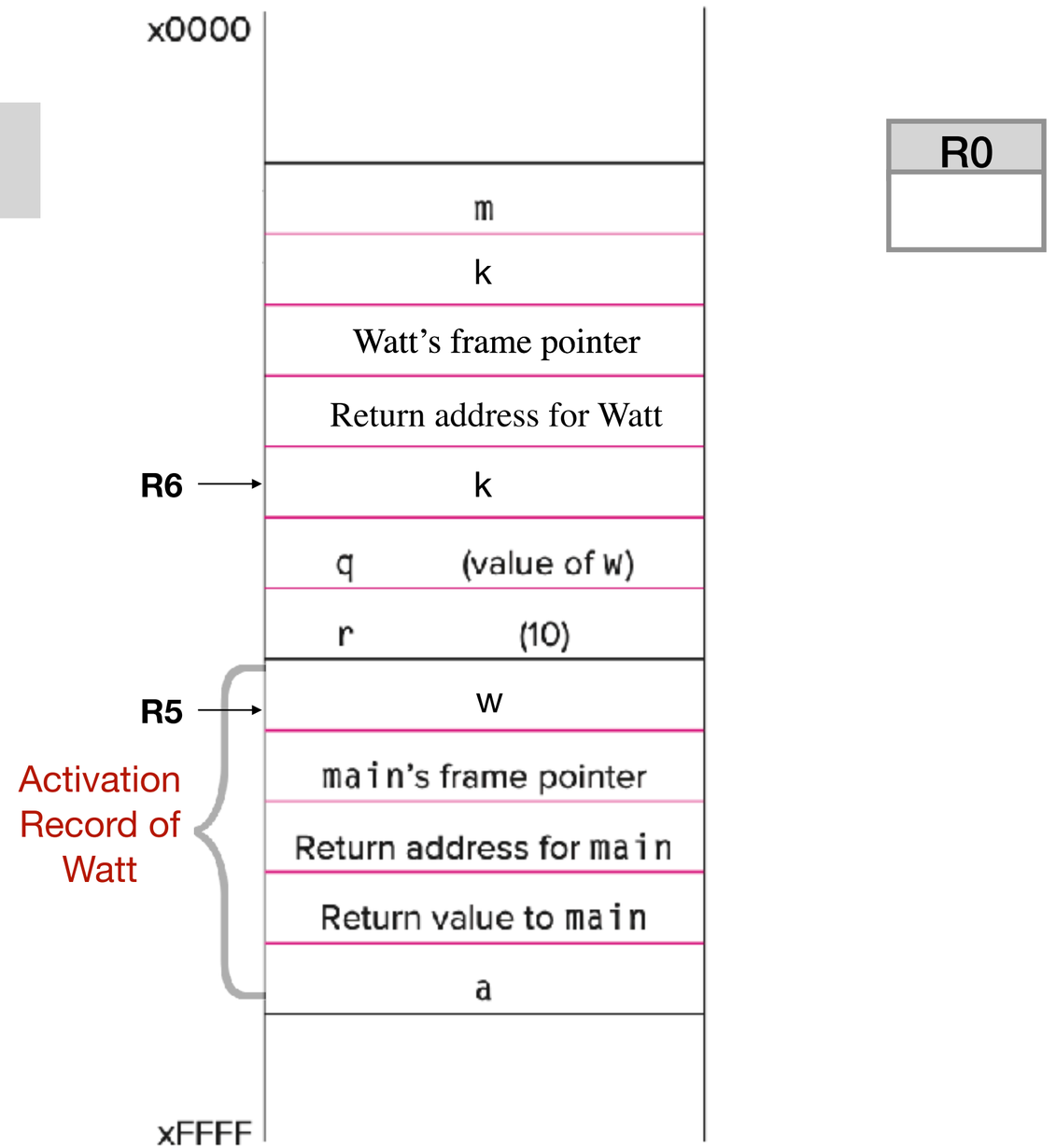


```
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

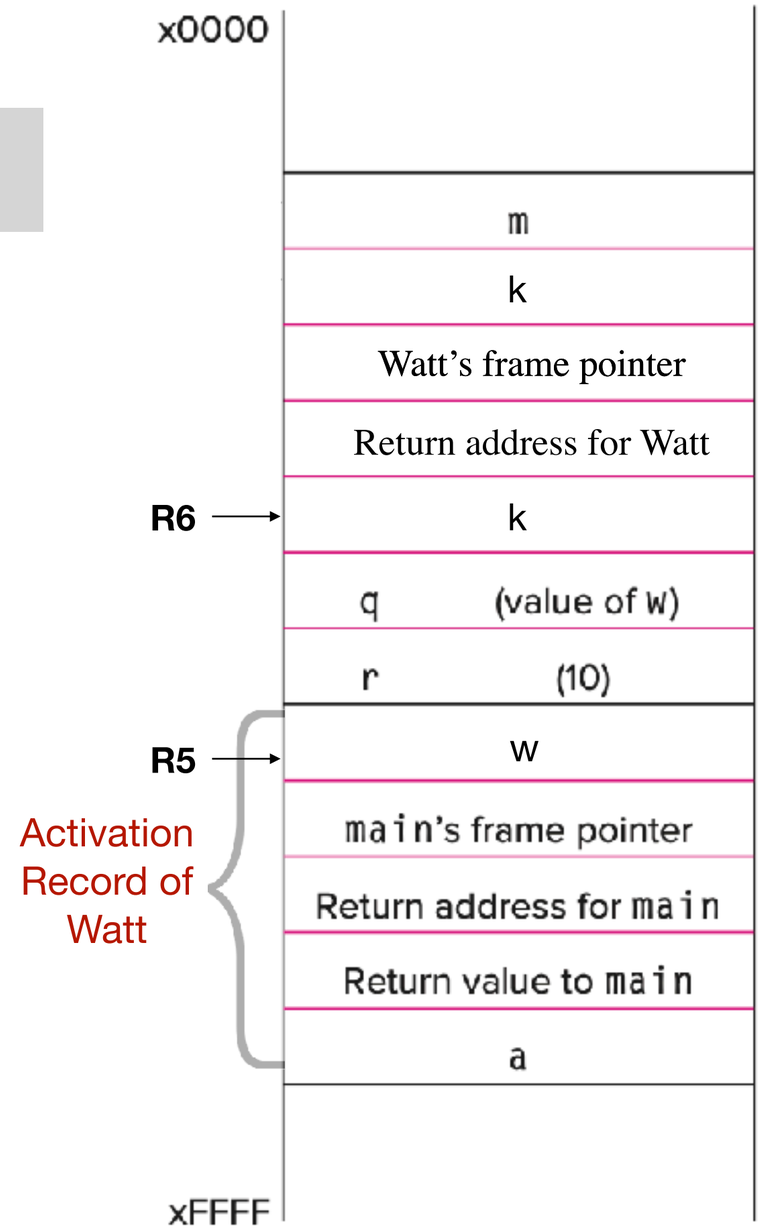


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```



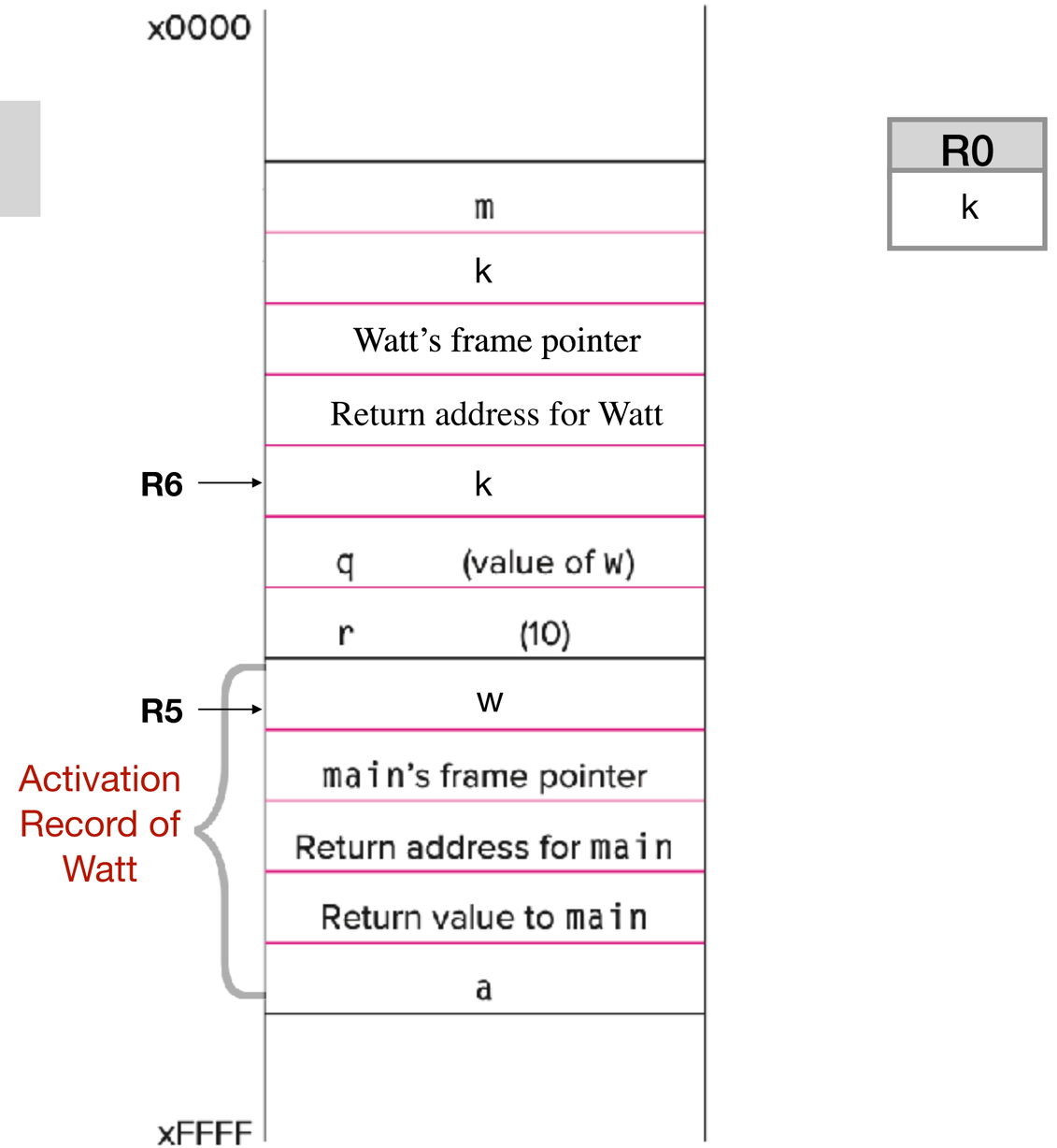
R0
k

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

LC-3 Implementation

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
```

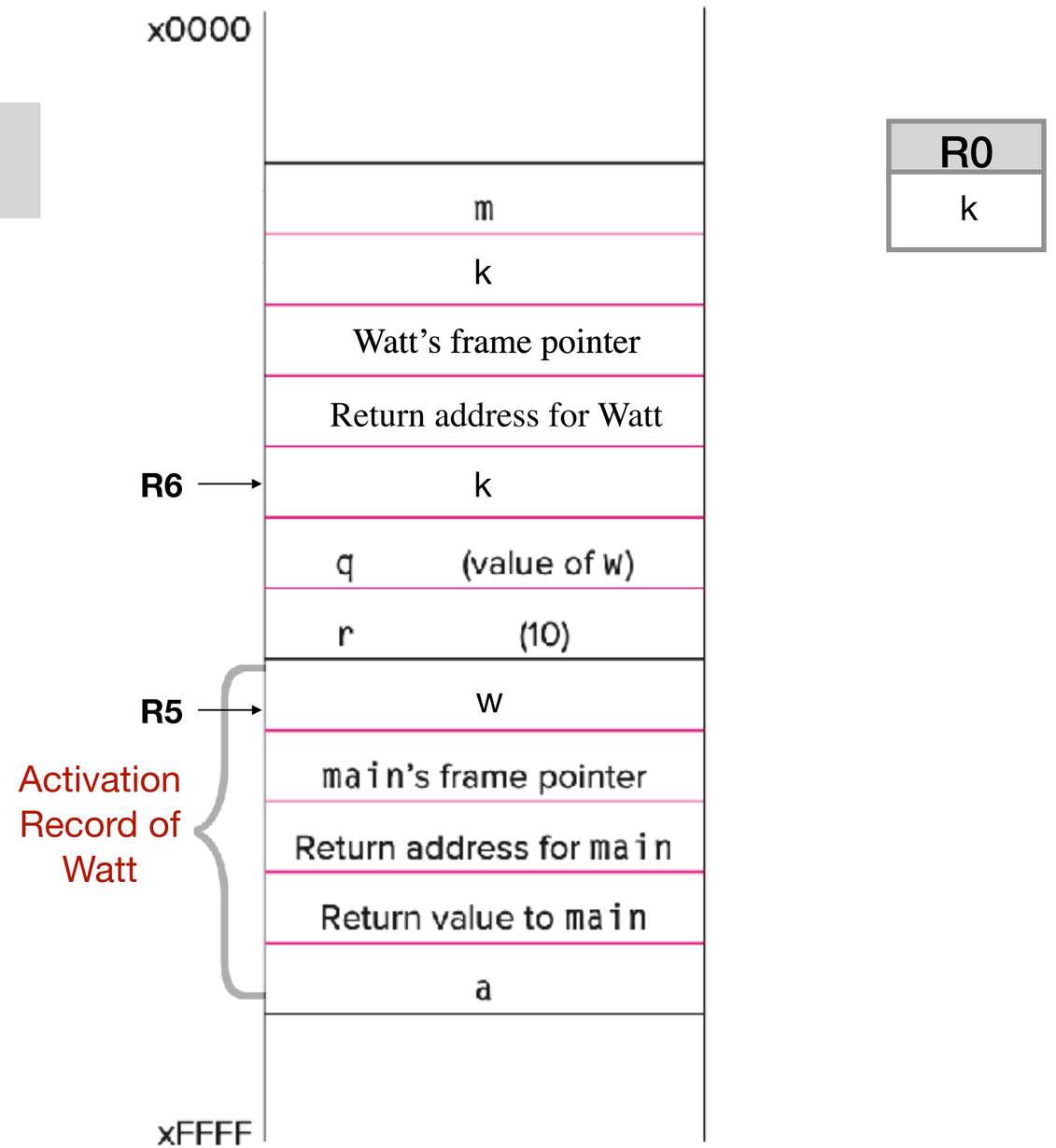


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

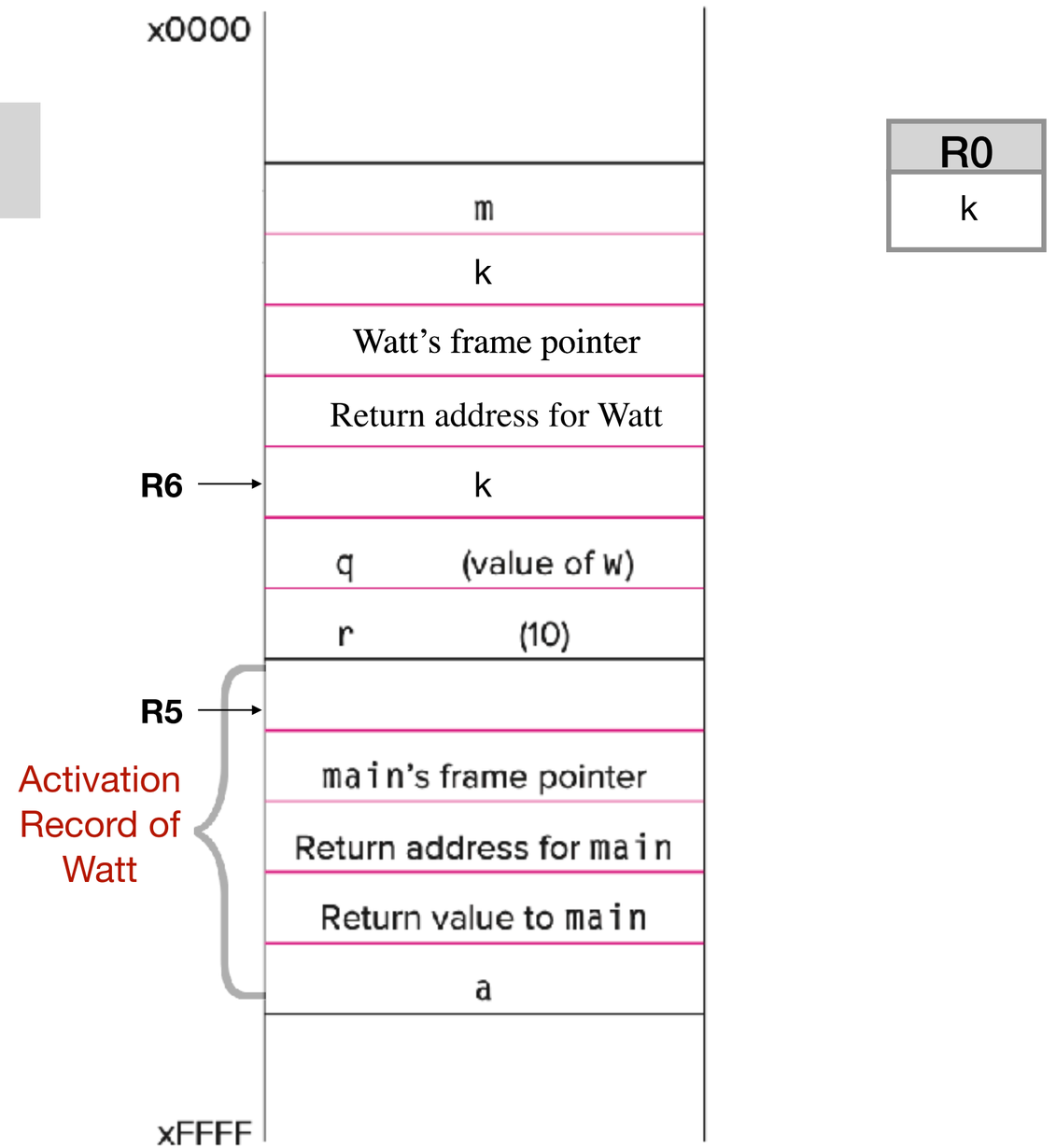


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

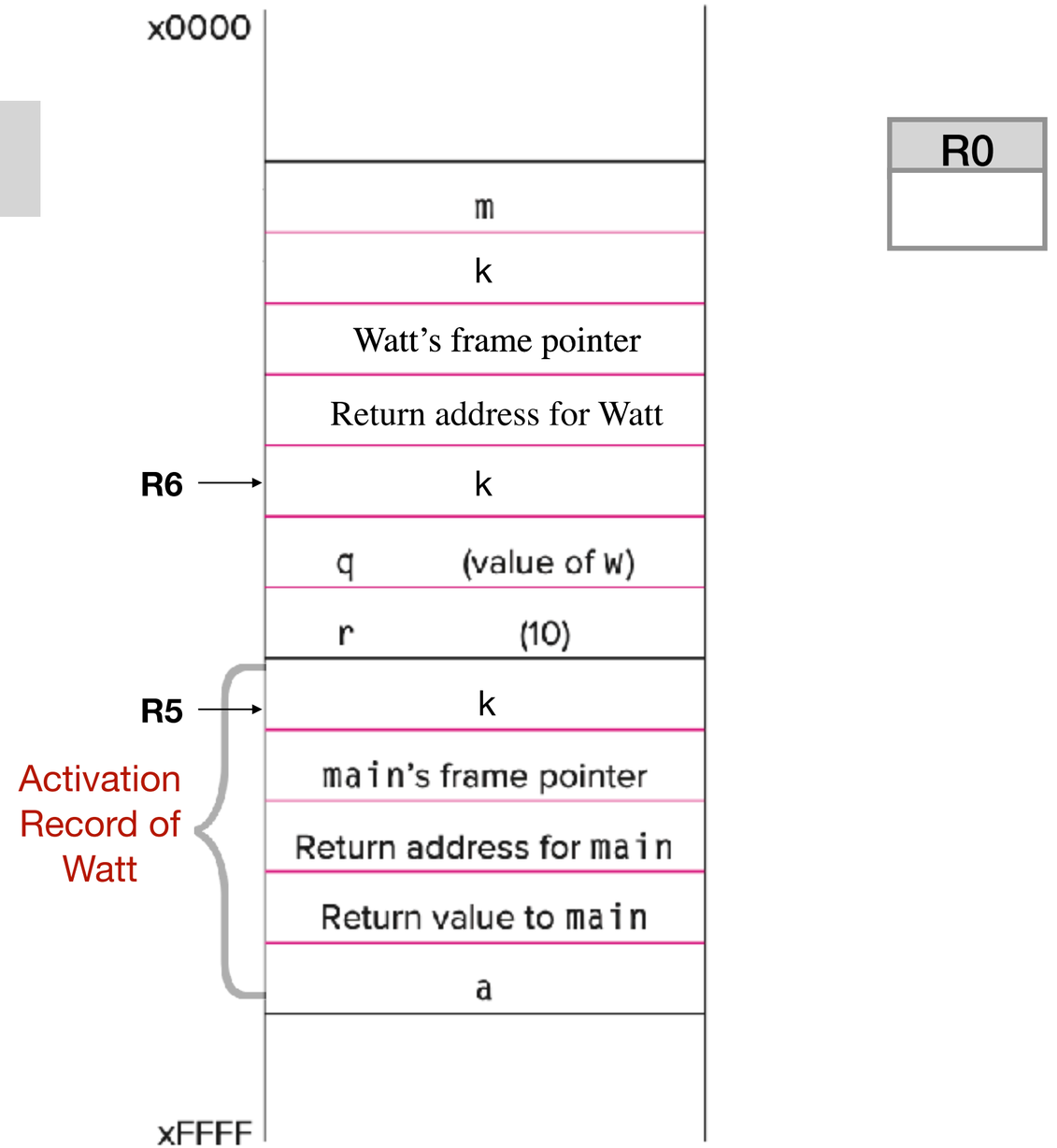


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

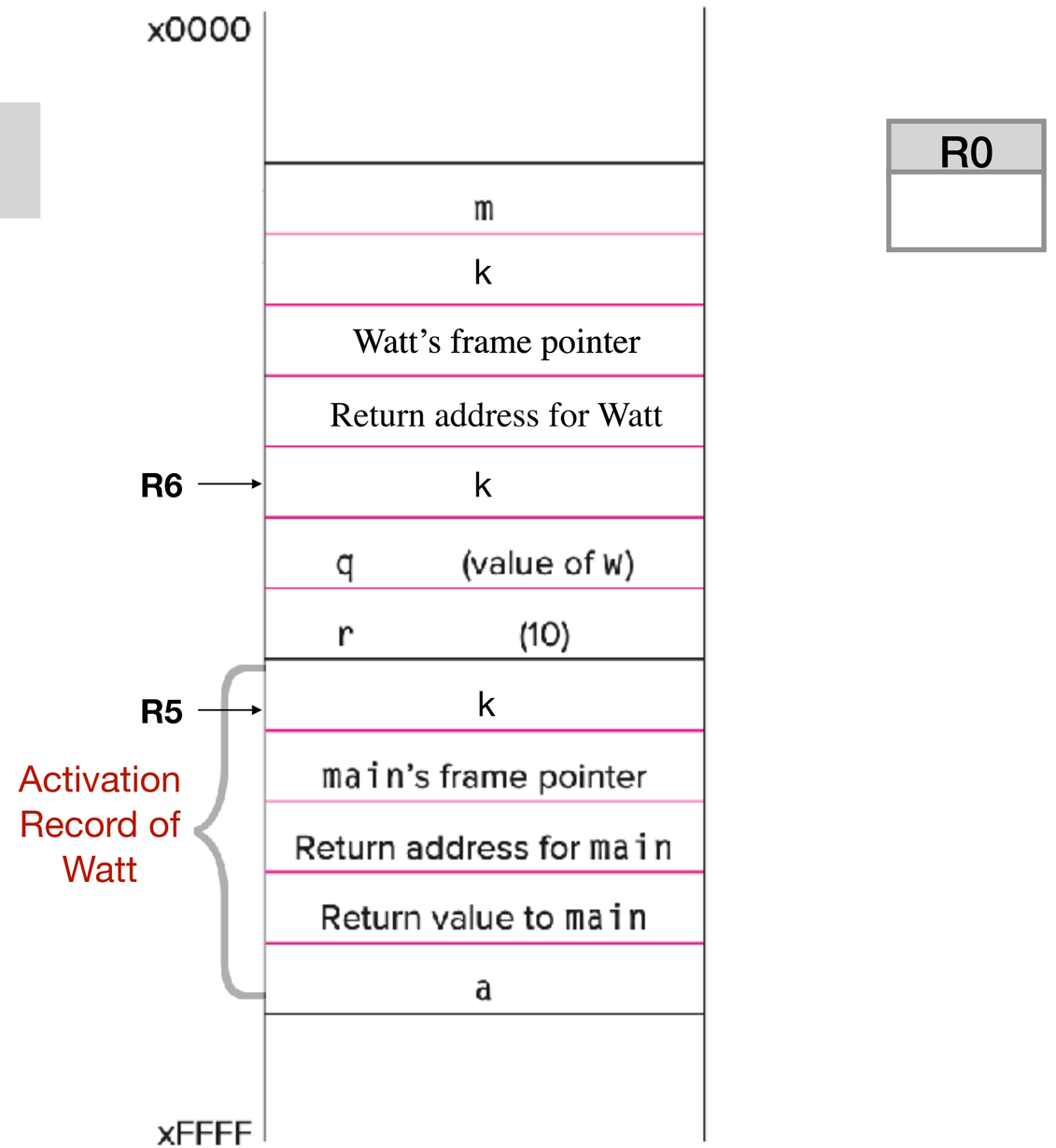


```
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

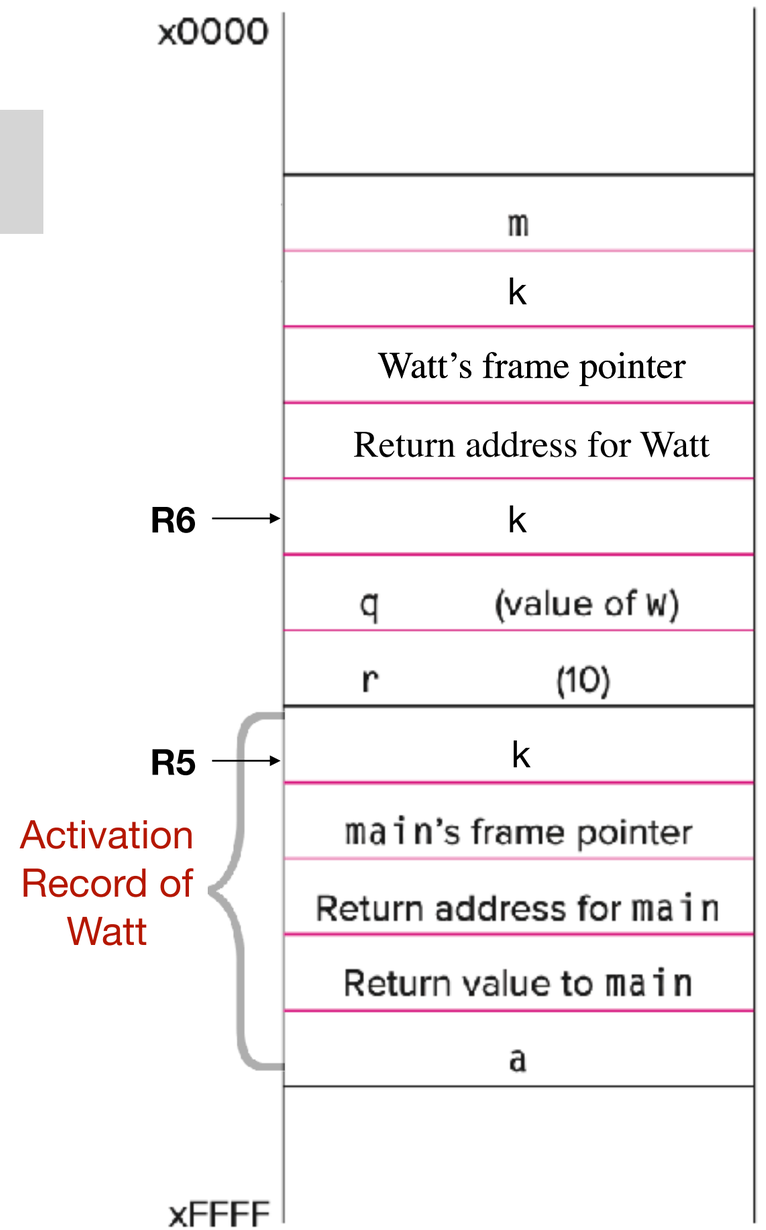


```
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

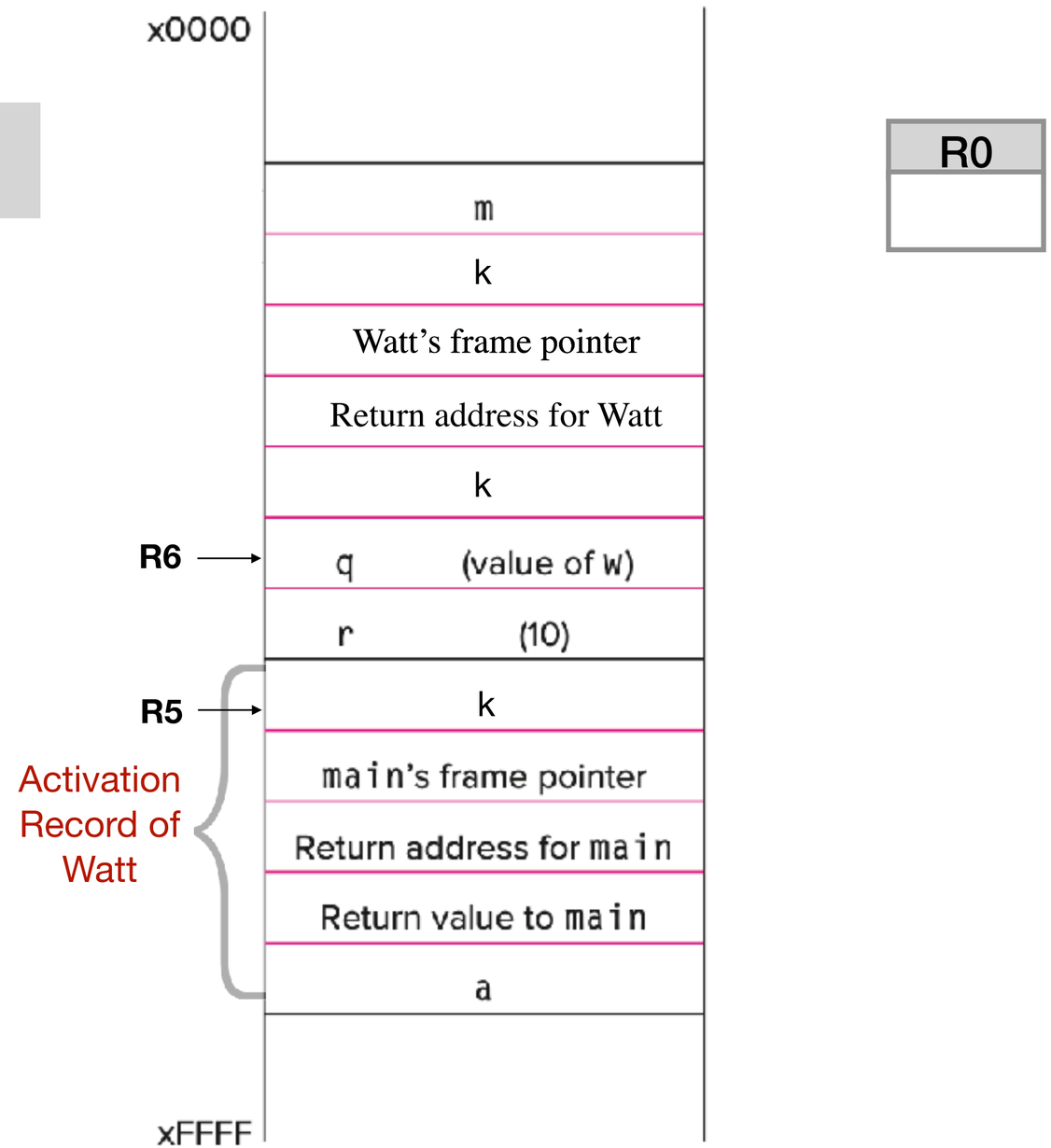


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

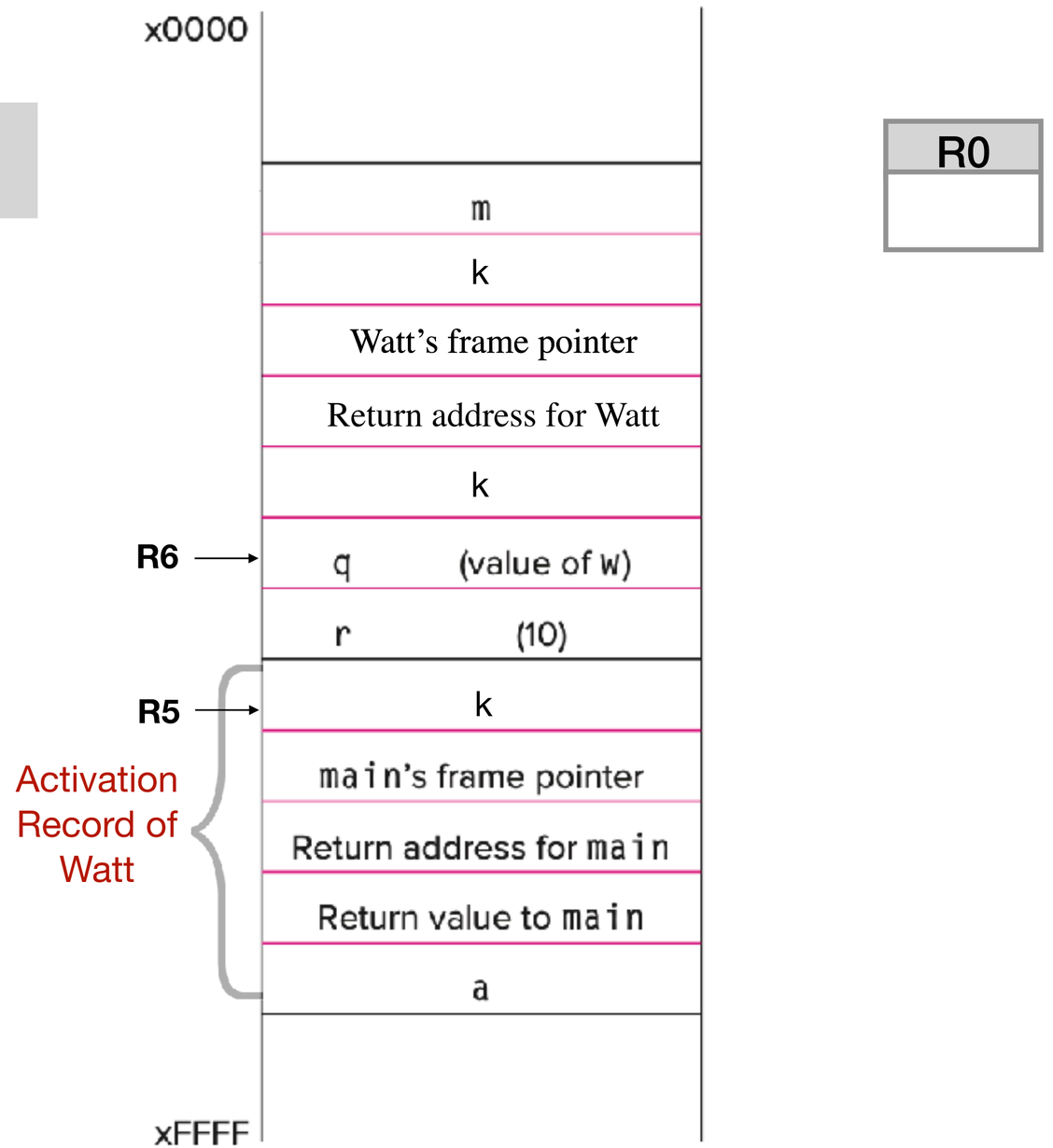


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

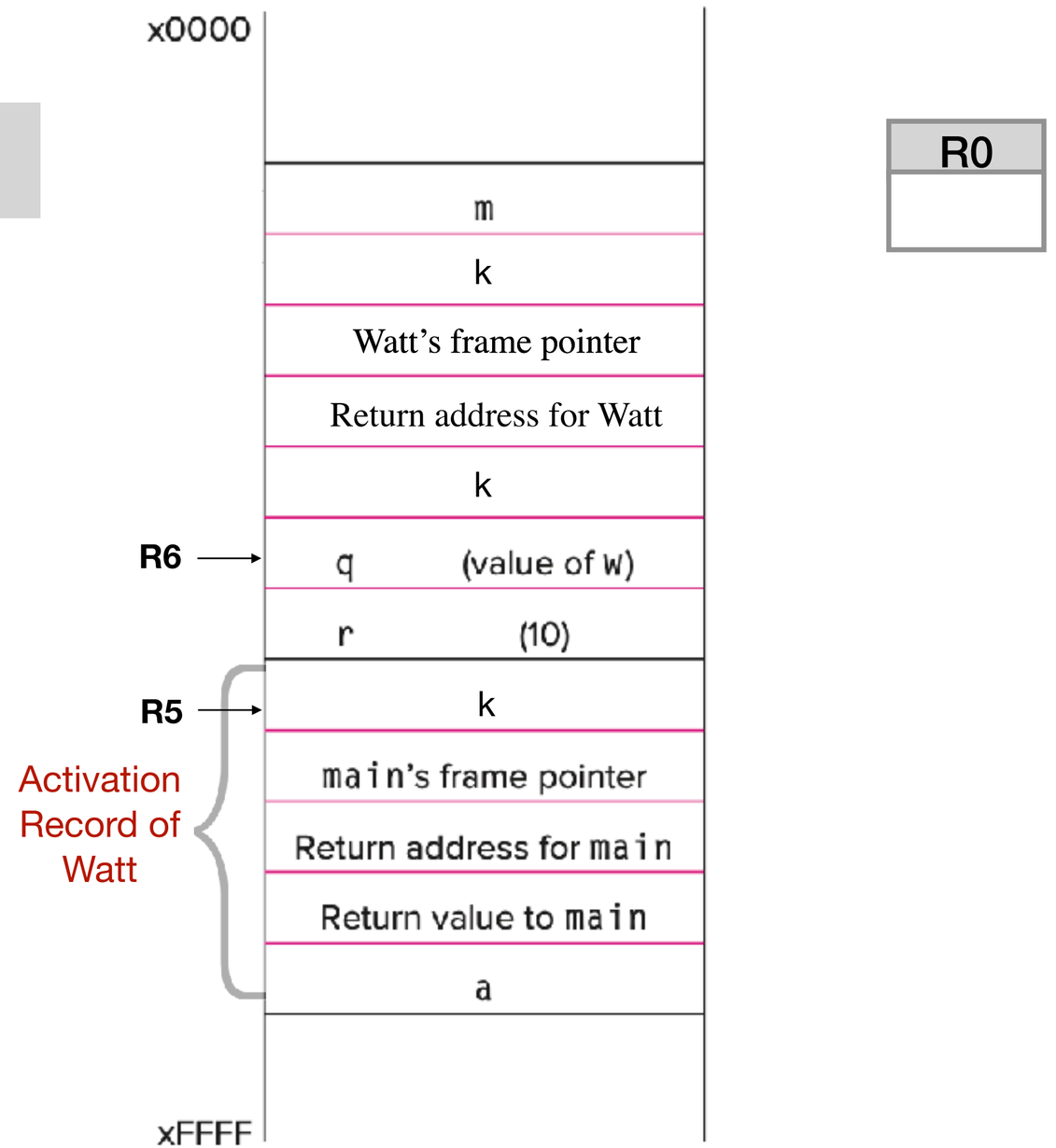


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

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
```

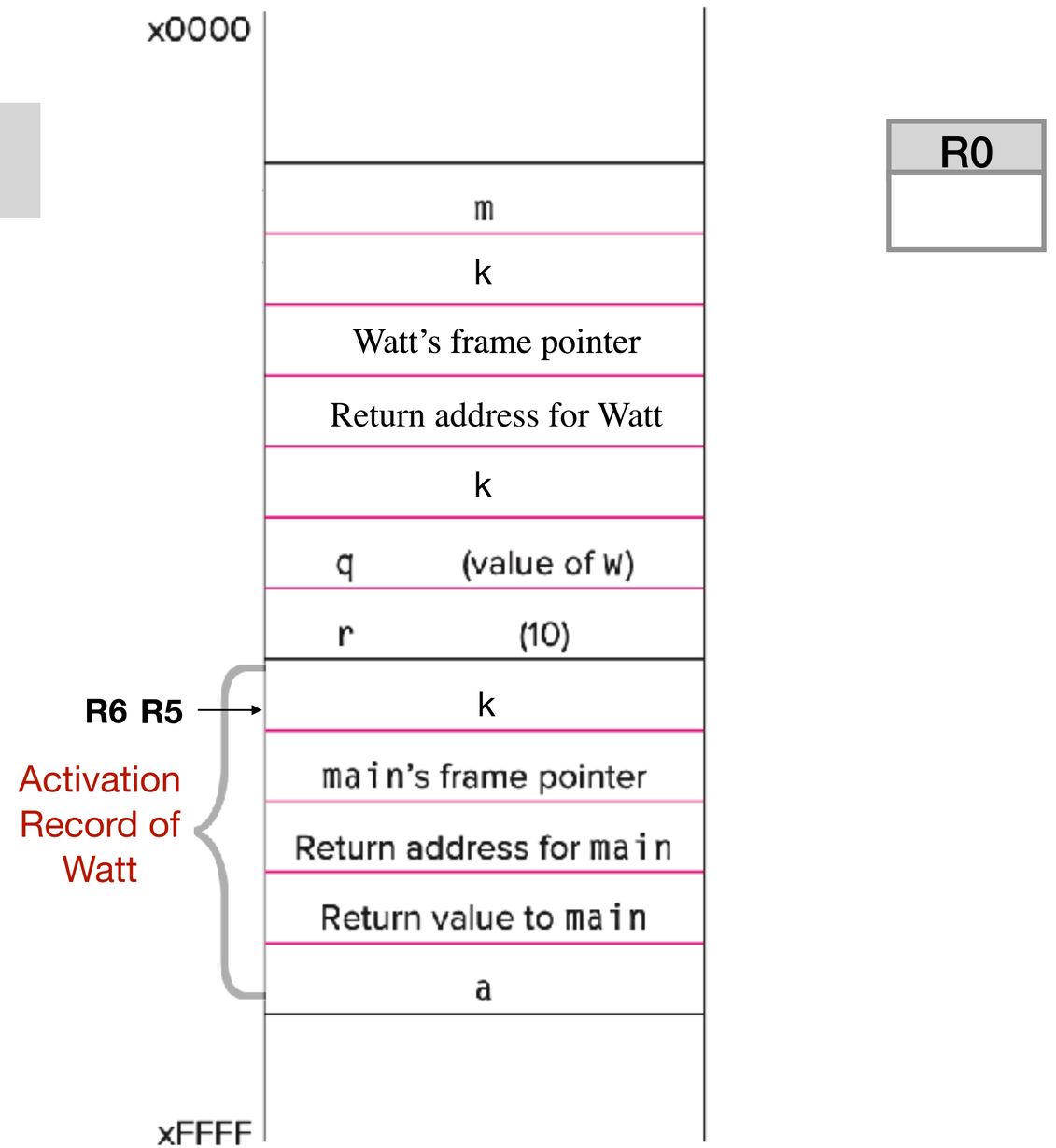


```c
int Watt(int a)
{
        int w;
        ...
        w = Volt(w,10);
        …
        return w;
}
```

LC-3 Implementation

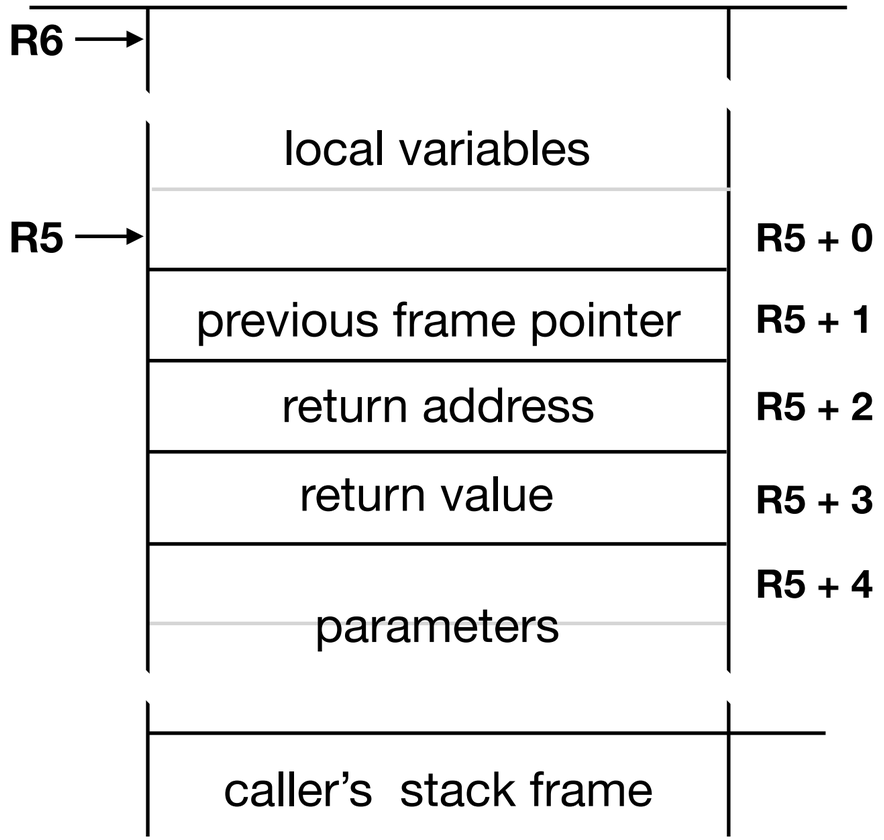
7. Caller tear-down (pop callee's return value and arguments from stack)

```
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:

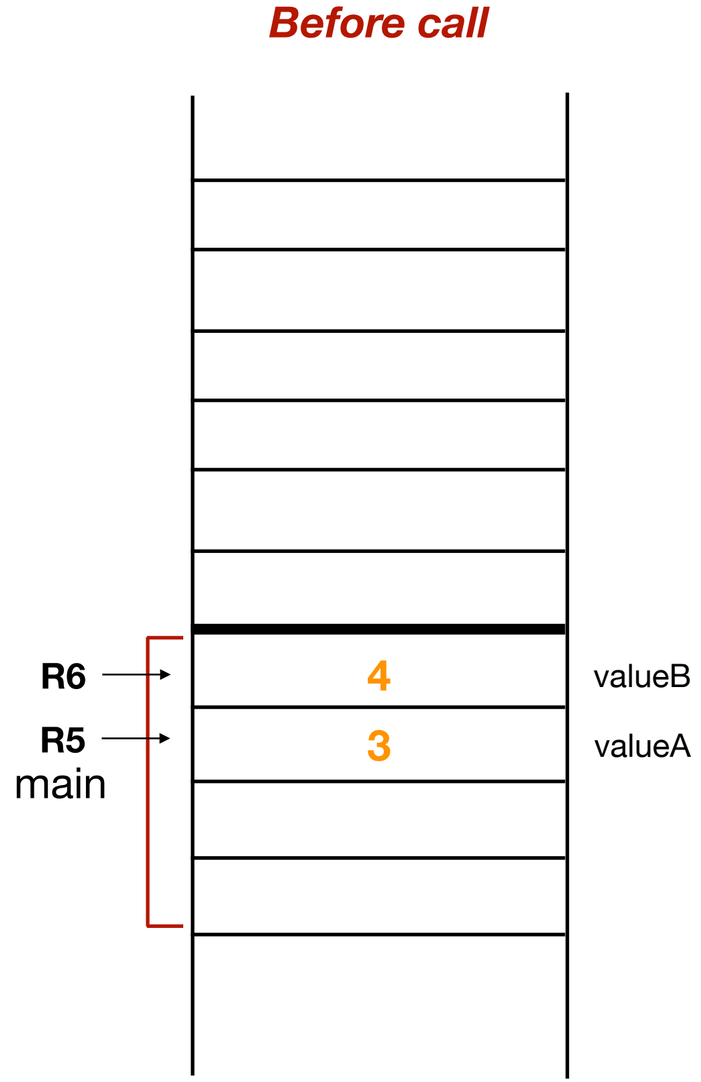
Swap valueA and valueB in main.

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:

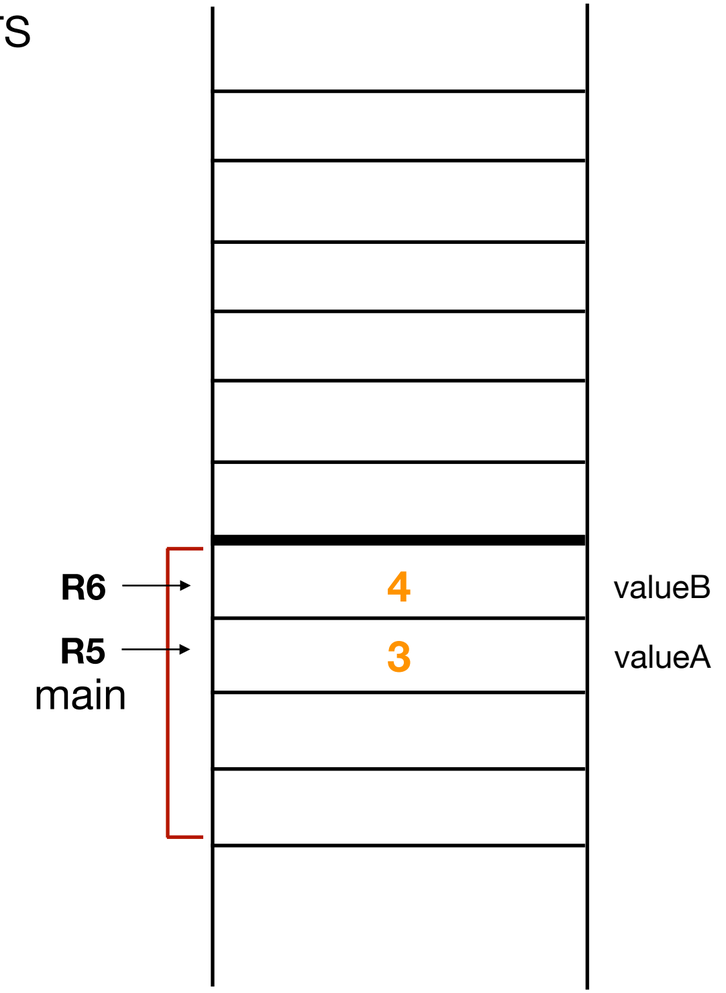
Swap valueA and valueB in main.



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;
}
```

1. Push arguments (R-to-L) onto RTS



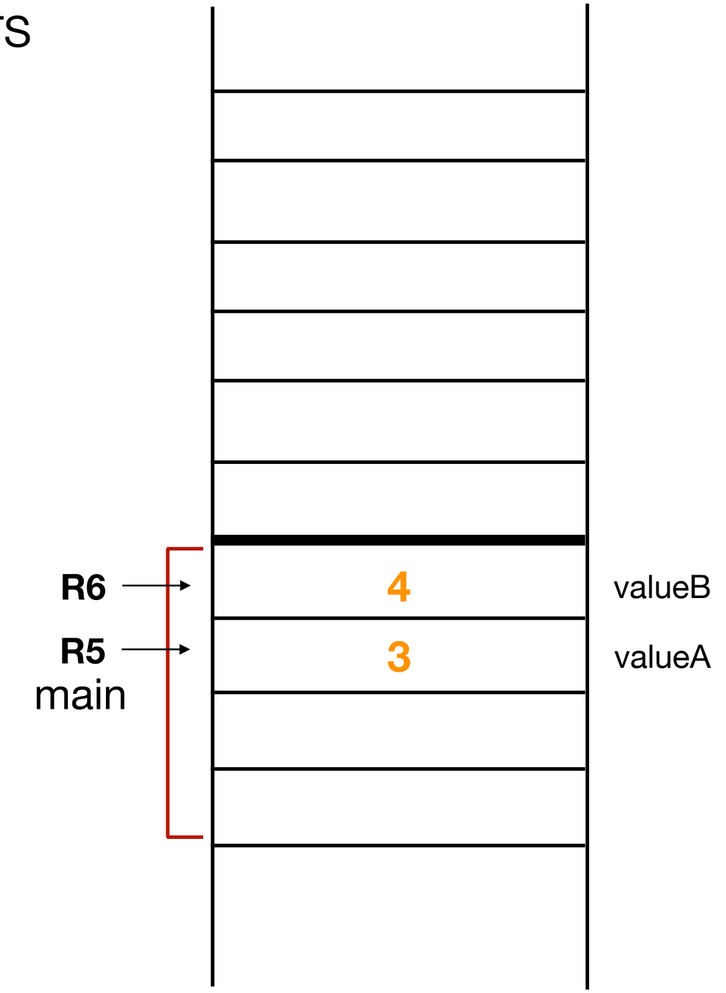
Goal:

Swap valueA and valueB in main.

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;
}
```

1. Push arguments (R-to-L) onto RTS
2. JSR



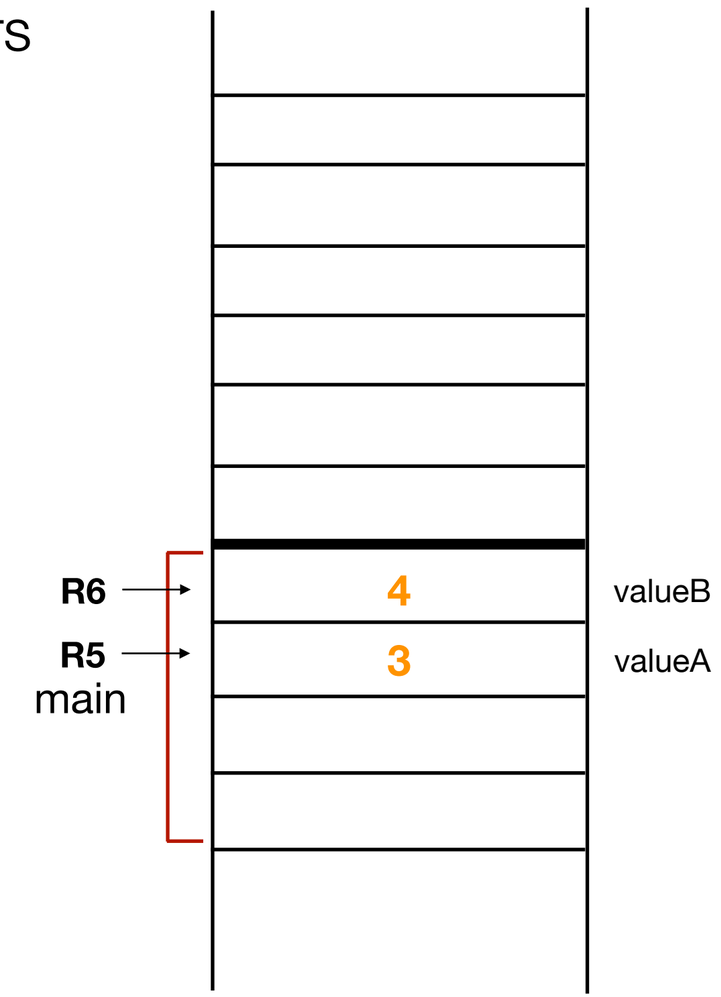
Goal:

Swap valueA and valueB in main.

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;
}
```

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



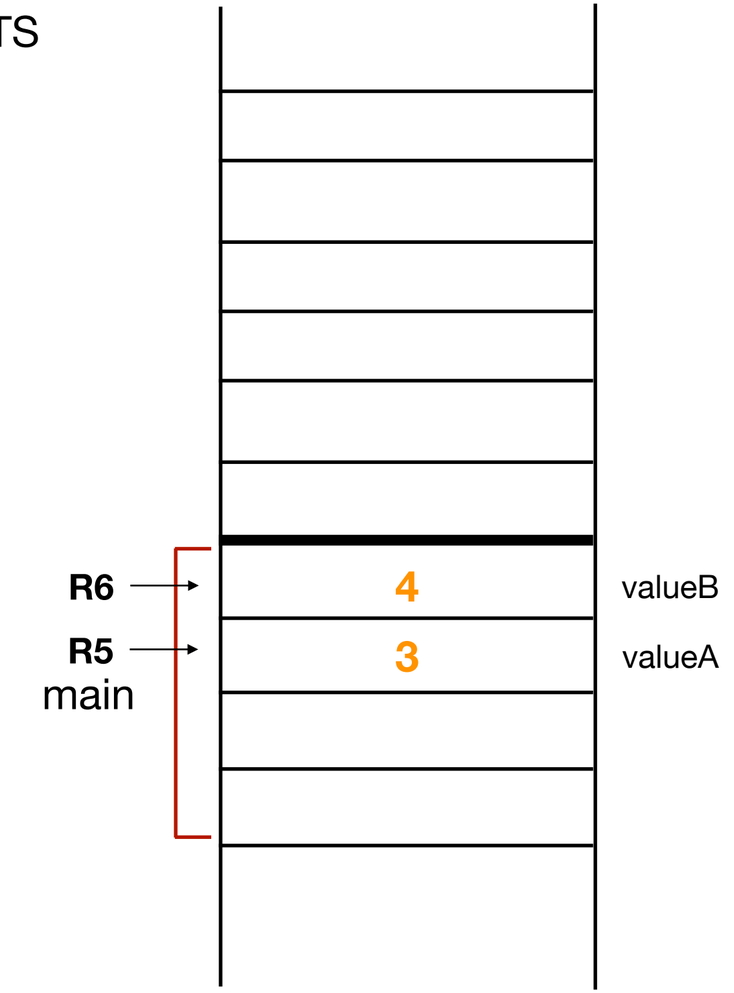
Goal:

Swap valueA and valueB in main.

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;
}
```

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



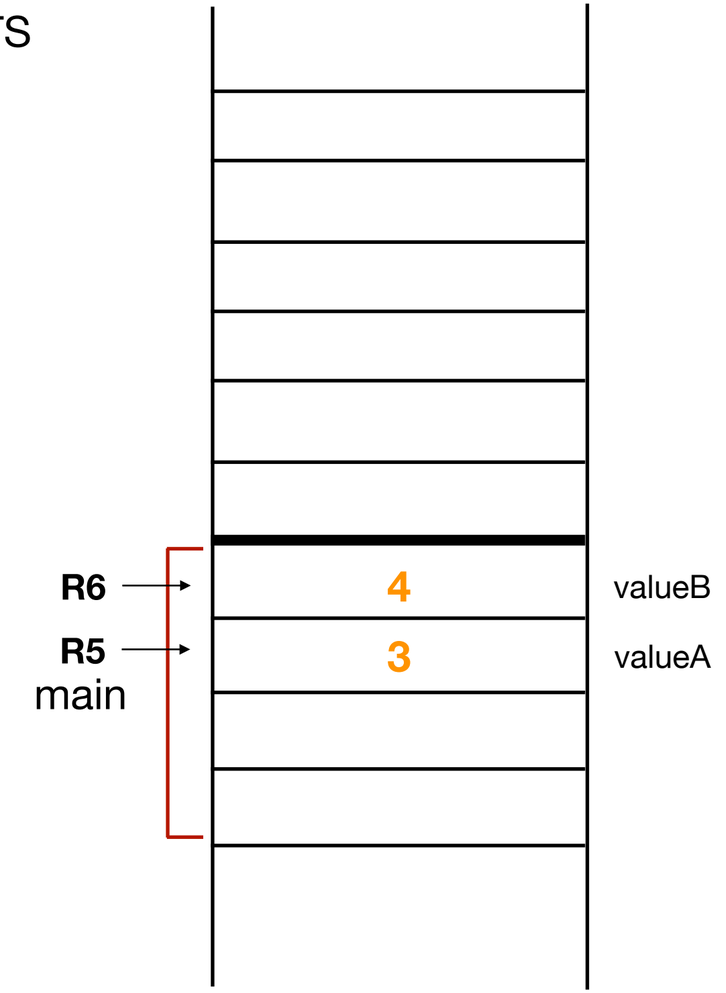
Goal:

Swap valueA and valueB in main.

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;
}
```

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



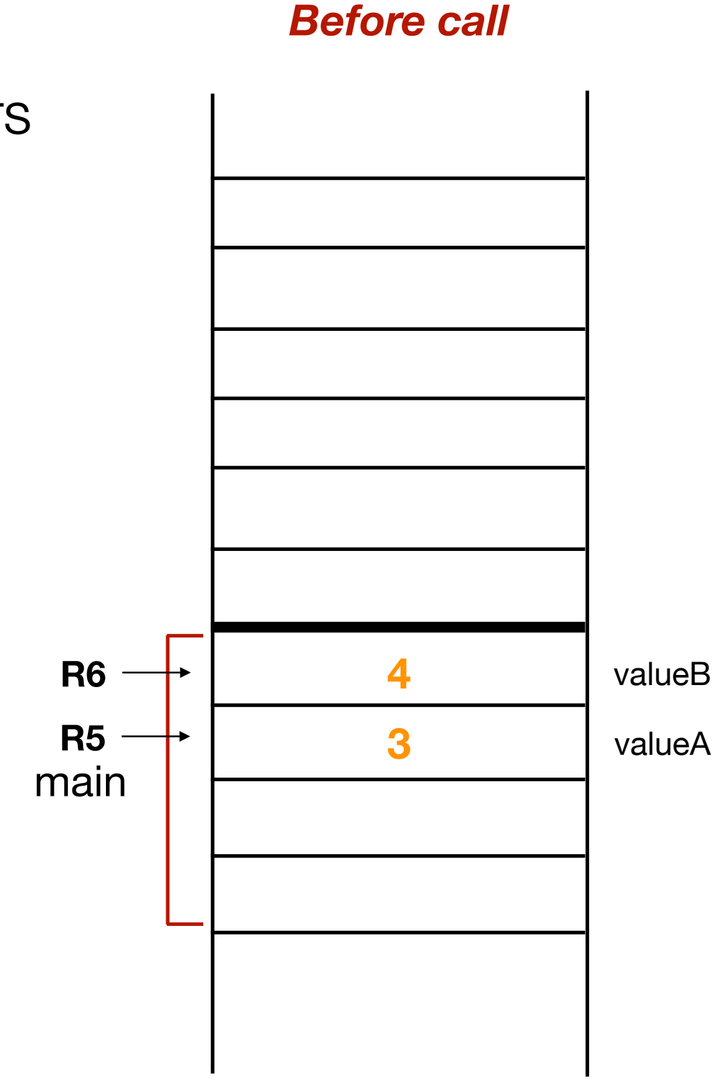
Goal:

Swap valueA and valueB in main.

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;
}
```

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



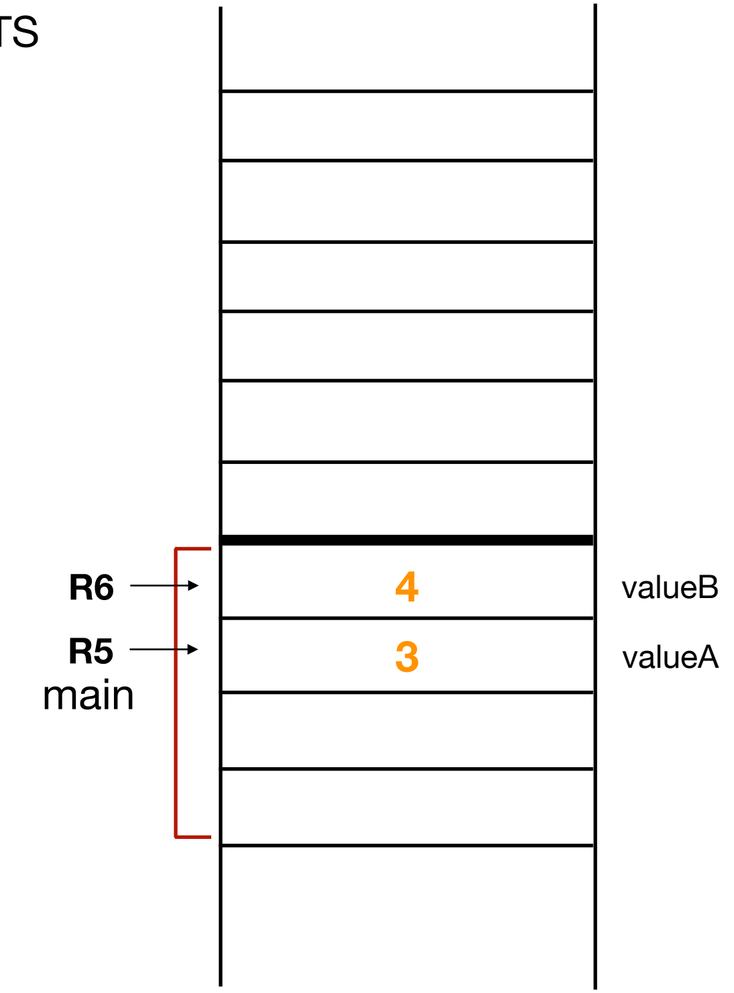
Goal:

Swap valueA and valueB in main.

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;
}
```

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



Goal:

Swap valueA and valueB in main.

`swap` function - build up

Build up

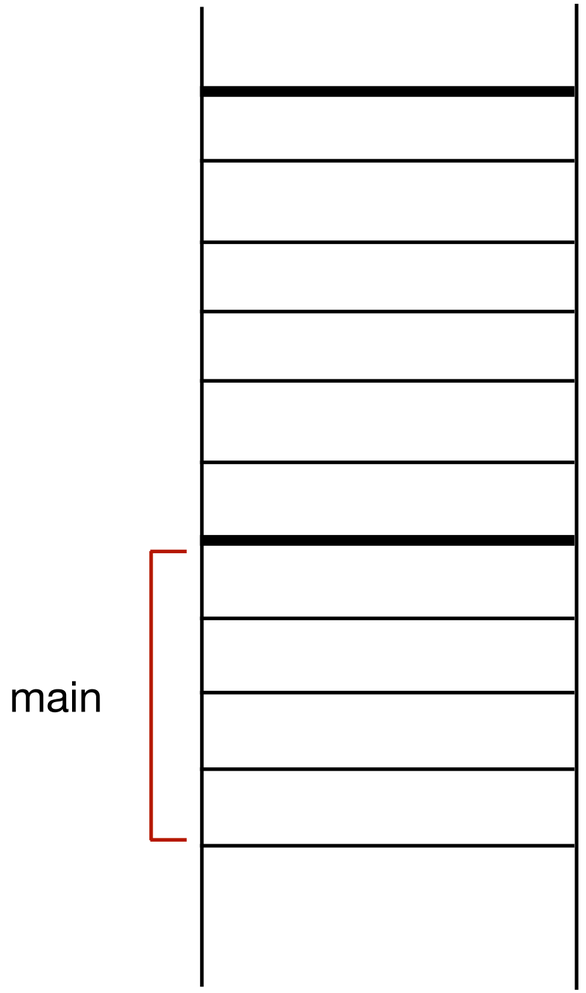
1. Push arguments (R-to-L) onto RTS
2. JSR
3. Callee build up (push onto RTS)
    A. Return value (allocate)
    B. Return address (from R7)
    C. Caller frame pointer (CFP)
    D. Local variables
4. Execute
5. Callee tear down
    E. Update return value
    F. Pop local variables
    G. Pop CFP (into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

```c
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 - 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

Build up

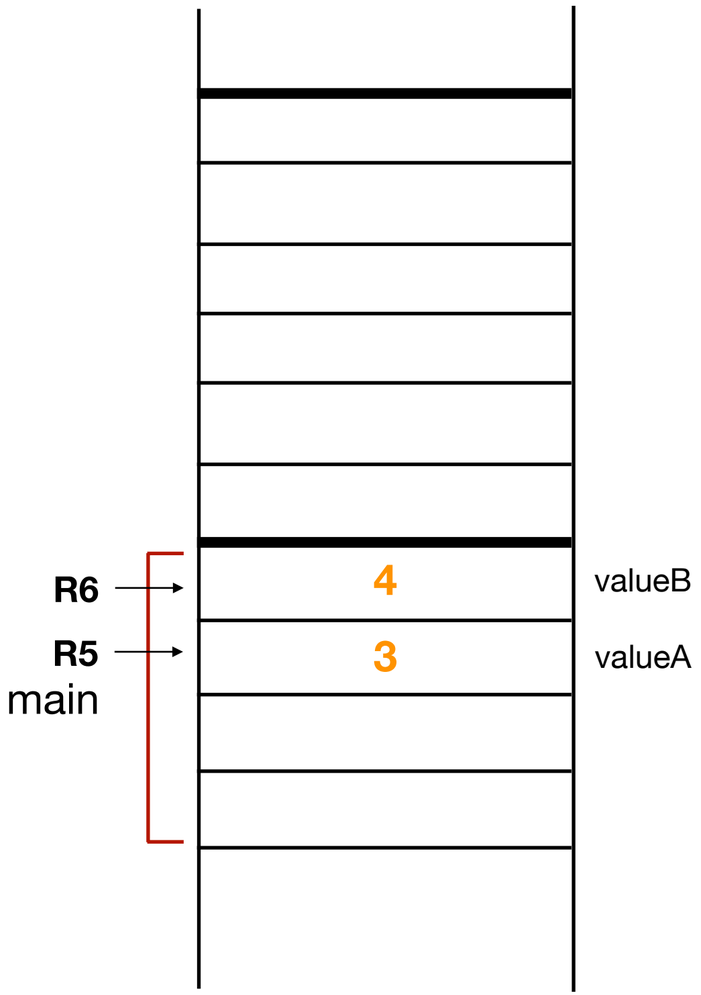



```c
void Swap(int first, int second);

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 - 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

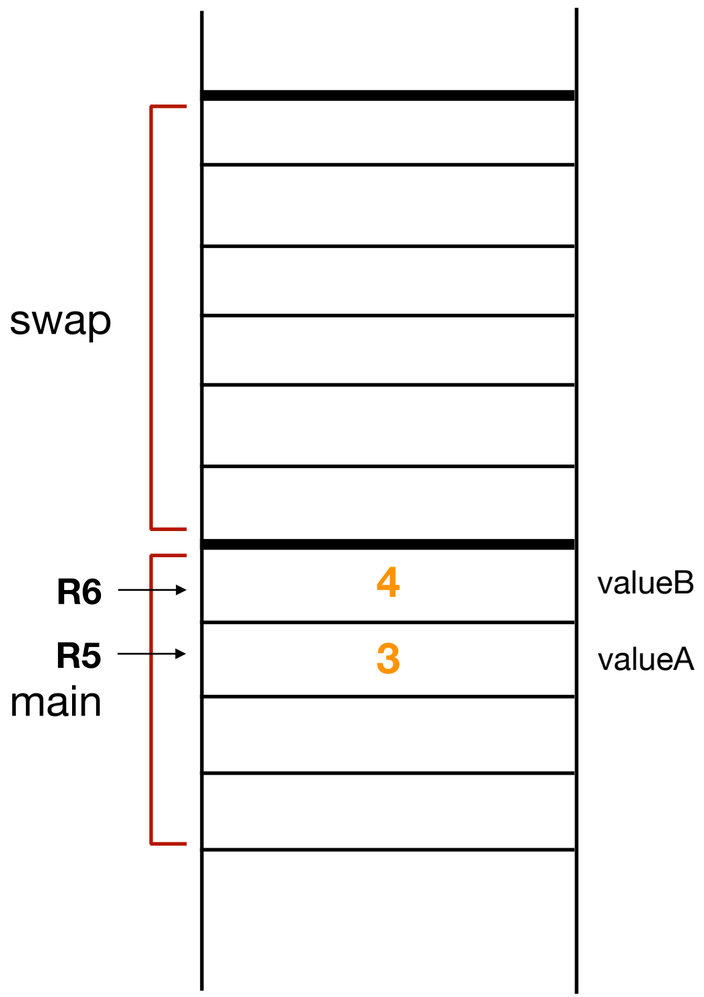
Build up




```c
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 - 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

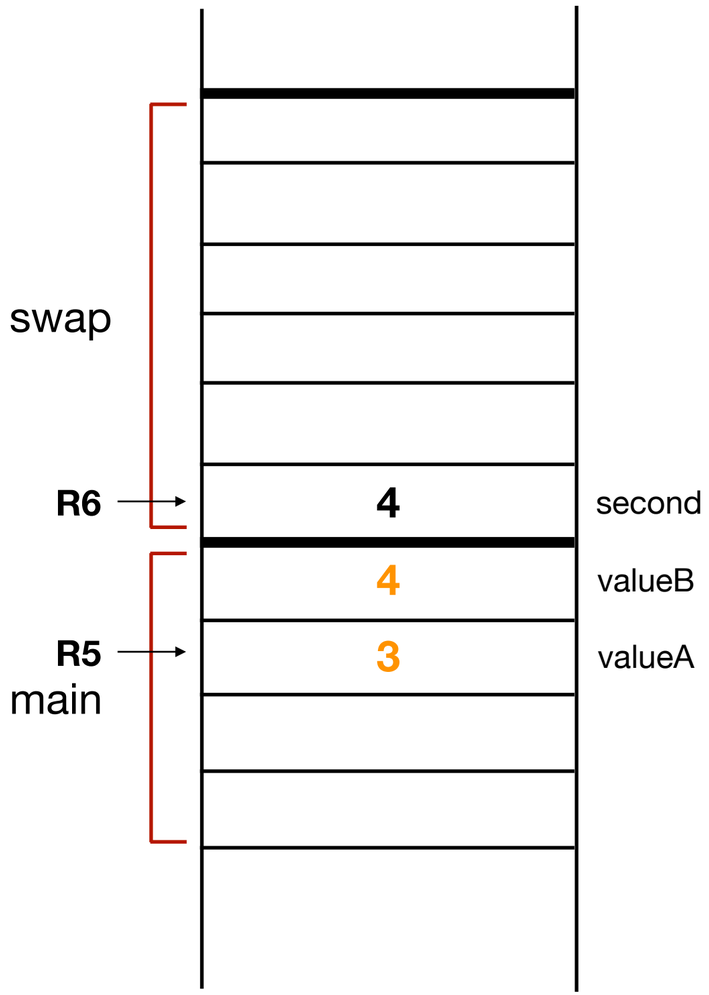



```c
void Swap(int first, int second);

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 - 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

Build up



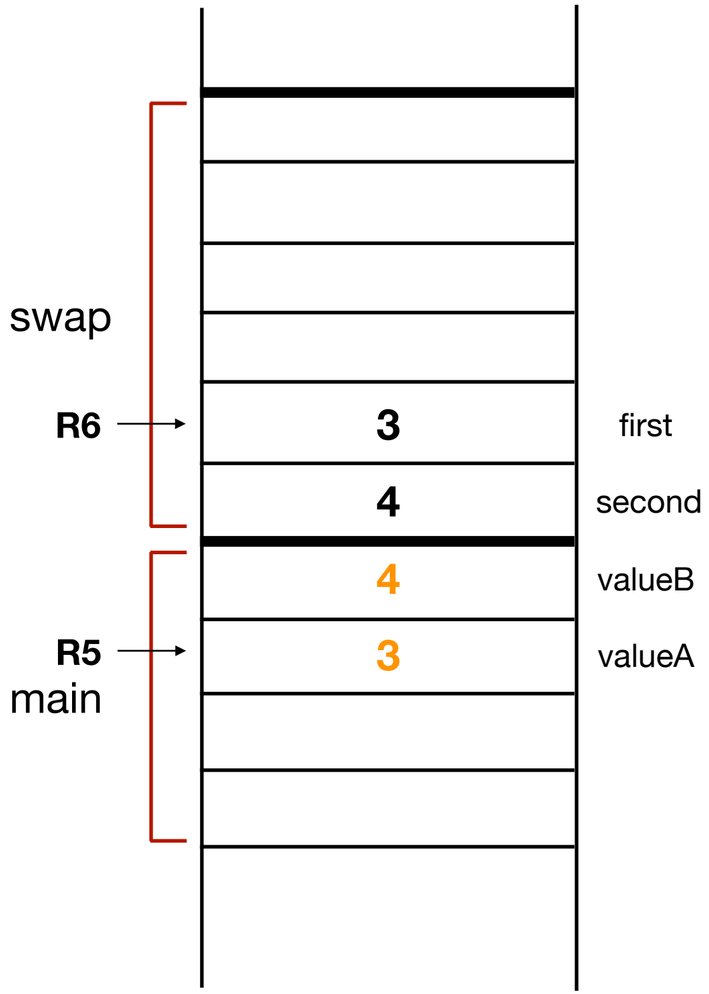

```c
void Swap(int first, int second);

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
LDR R6, R5, #-1
```

`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

Build up



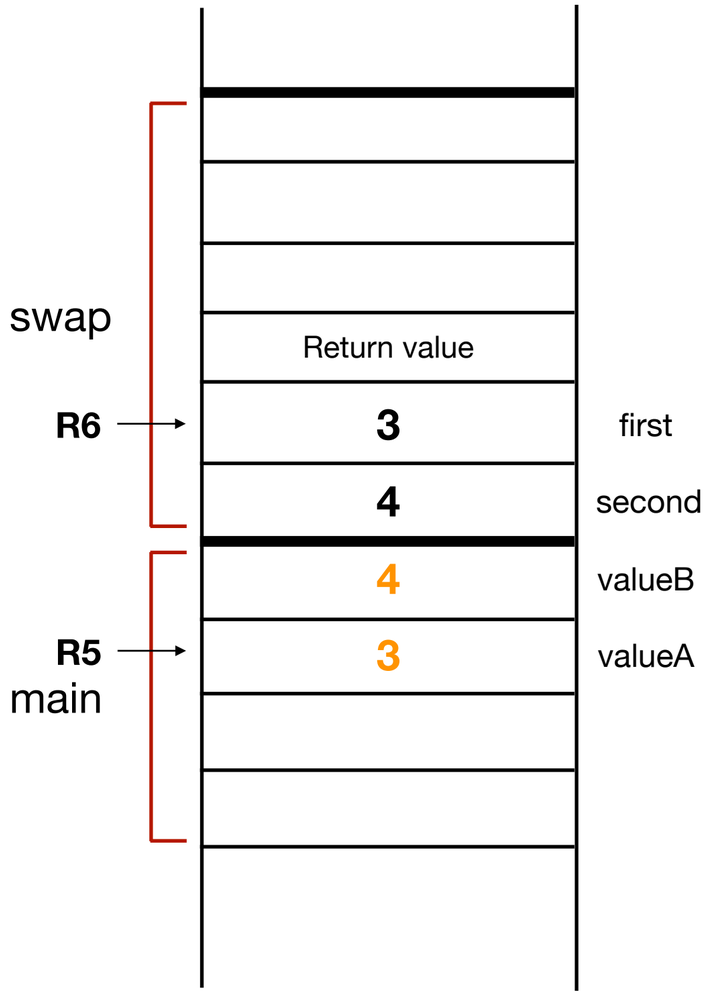

```c
void Swap(int first, int second);

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
LDR R6, R5, #0
```

`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

Build up

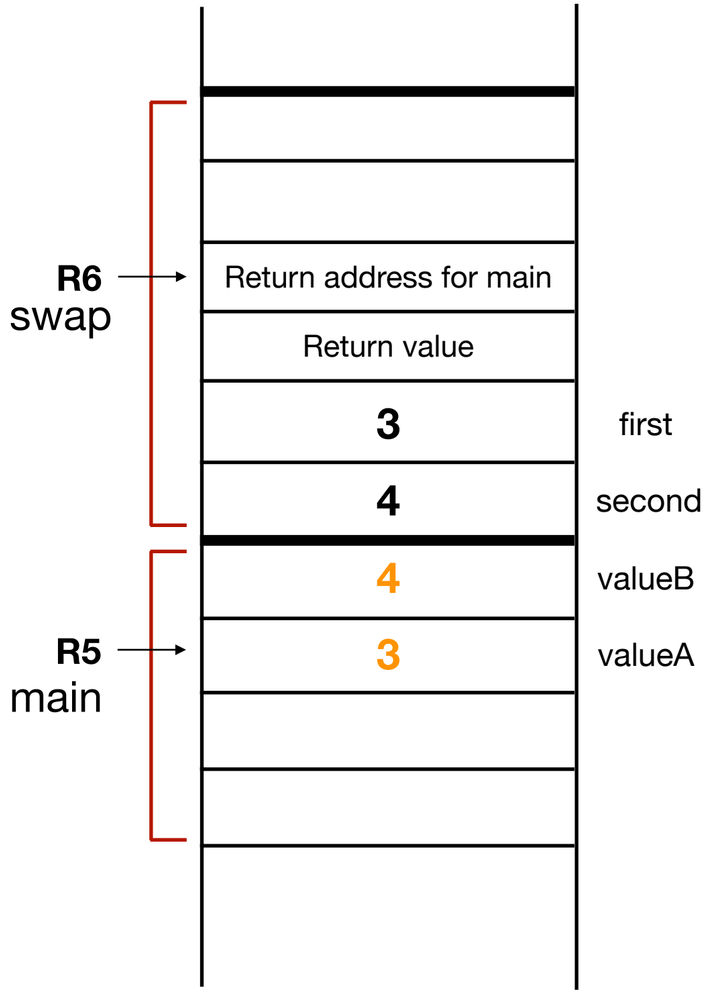



```c
void Swap(int first, int second);

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 - 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

Build up



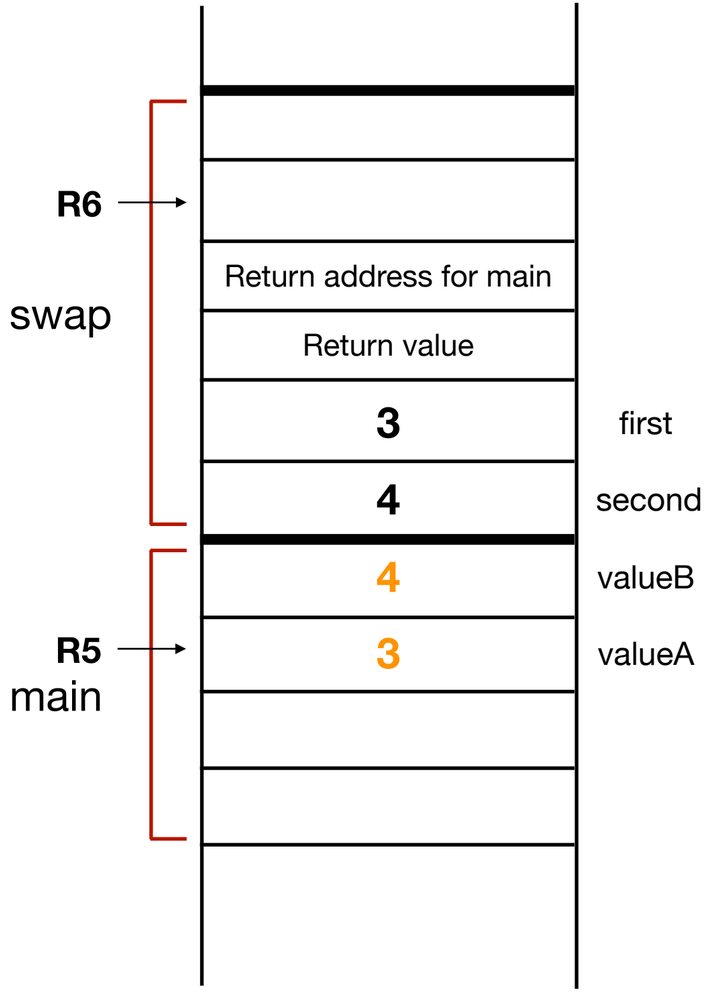

```c
void Swap(int first, int second);

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, #-2
STR R7, R6, #0
```

`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

Build up



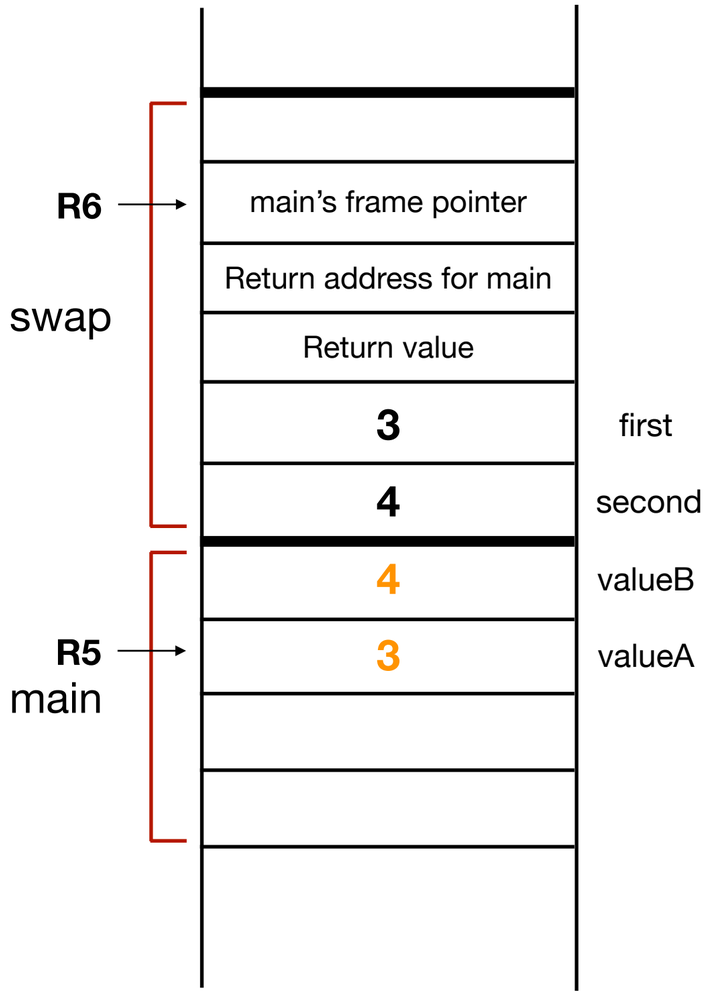

```
void Swap(int first, int second);

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
STR R5, R6, #0
```

`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

Build up



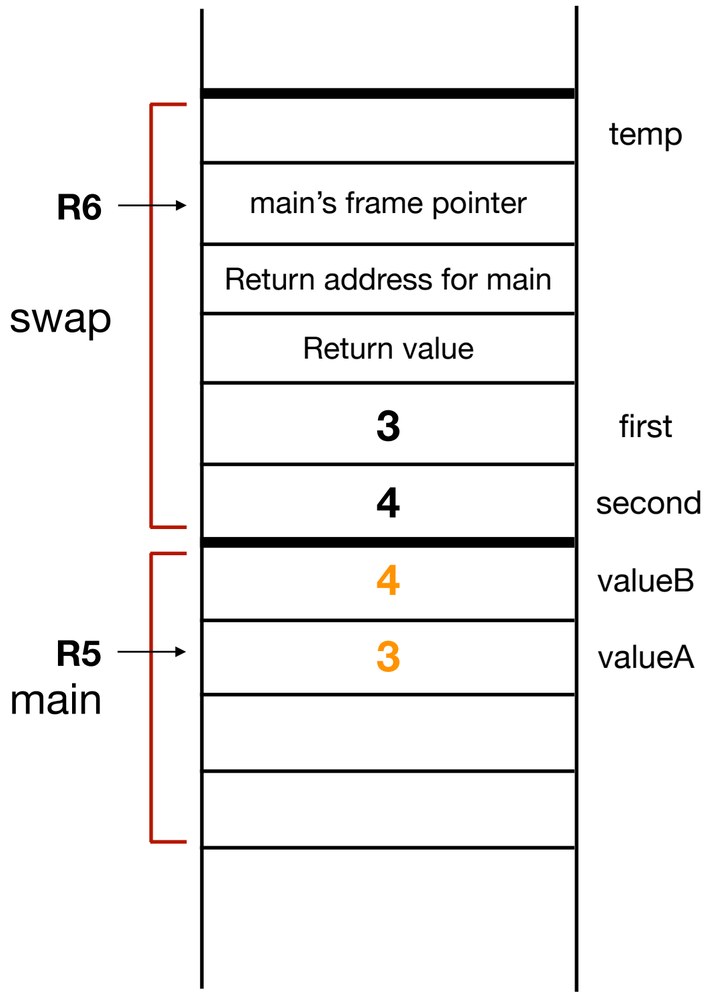

```
void Swap(int first, int second);

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
STR R5, R6, #0
```

`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

Build up

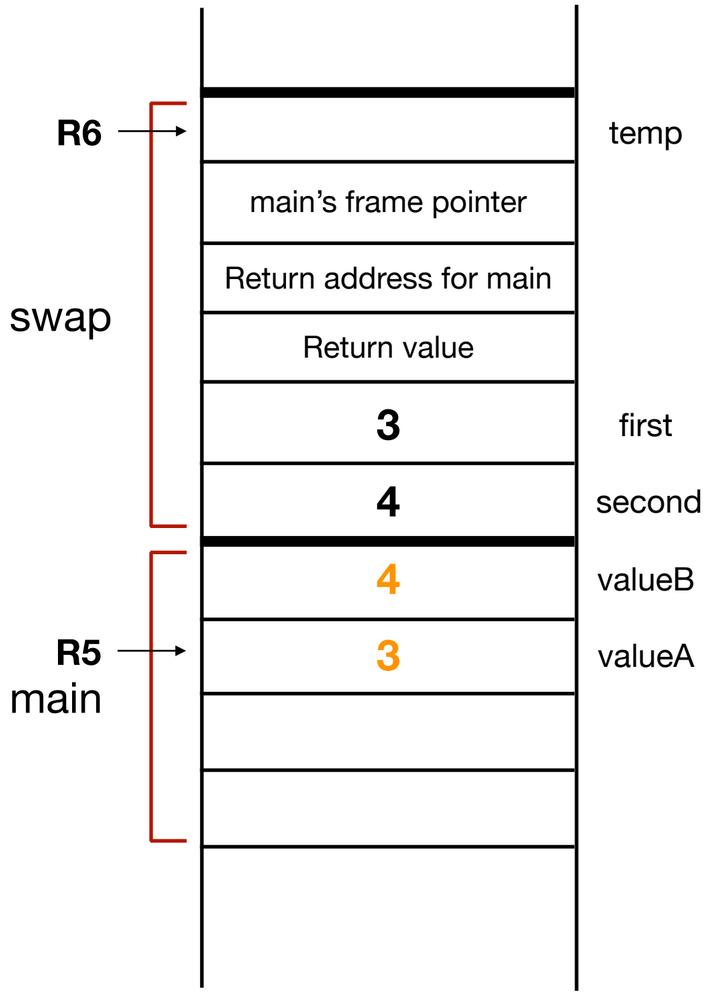



```c
void Swap(int first, int second);

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 - 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

Build up



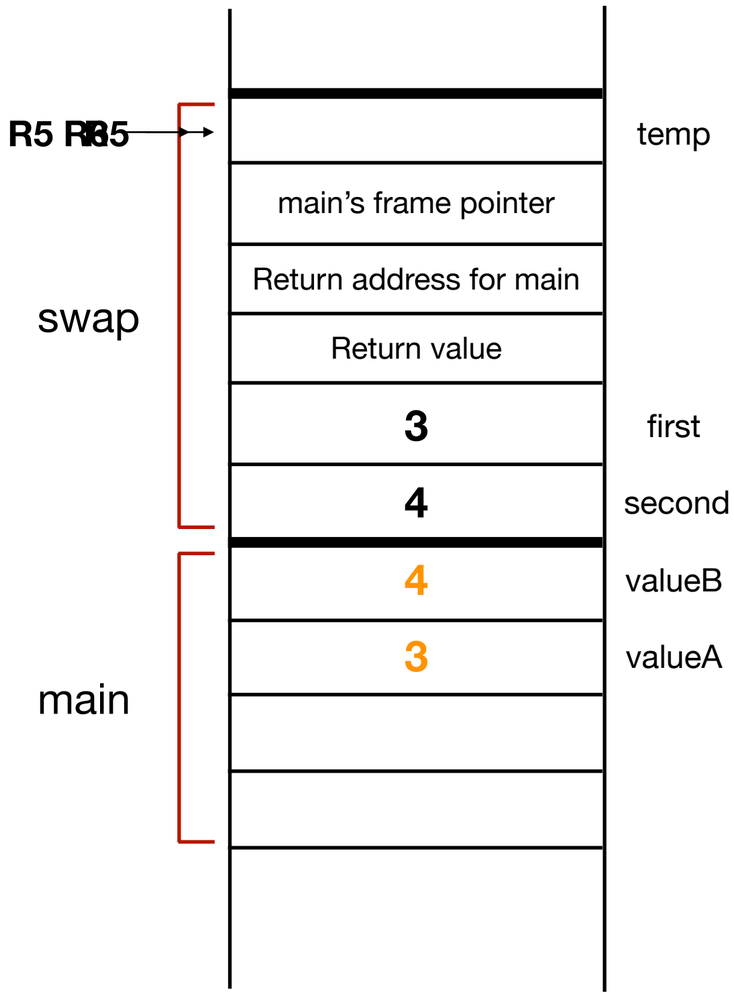

```
void Swap(int first, int second);

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
```

`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

Build up



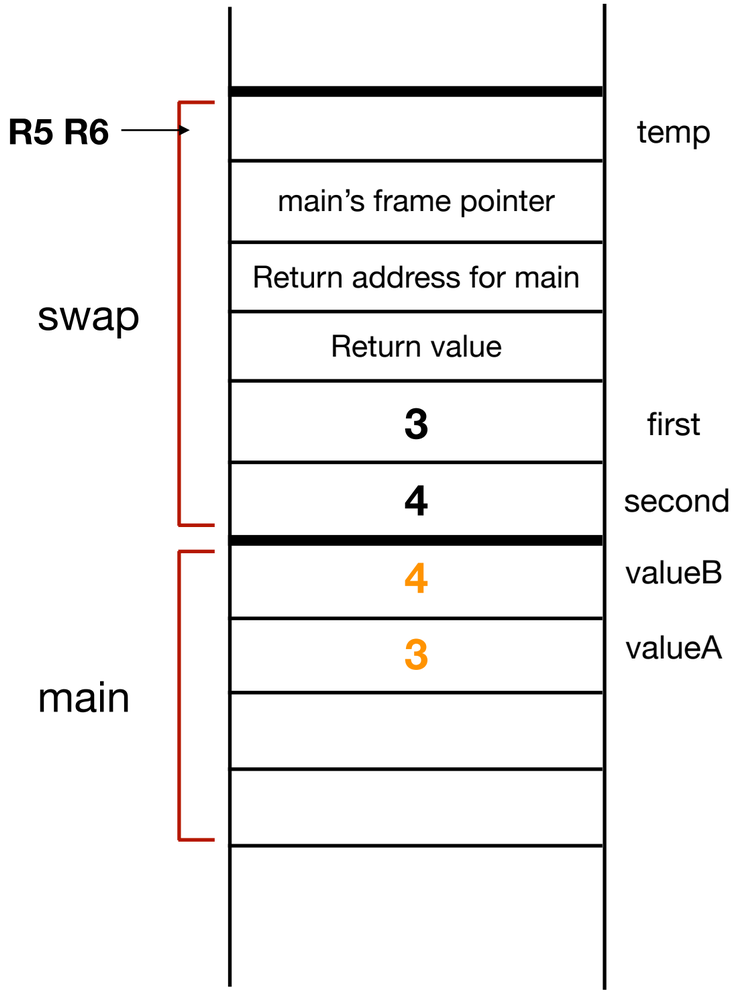

```c
void Swap(int first, int second);

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, #0
```

`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

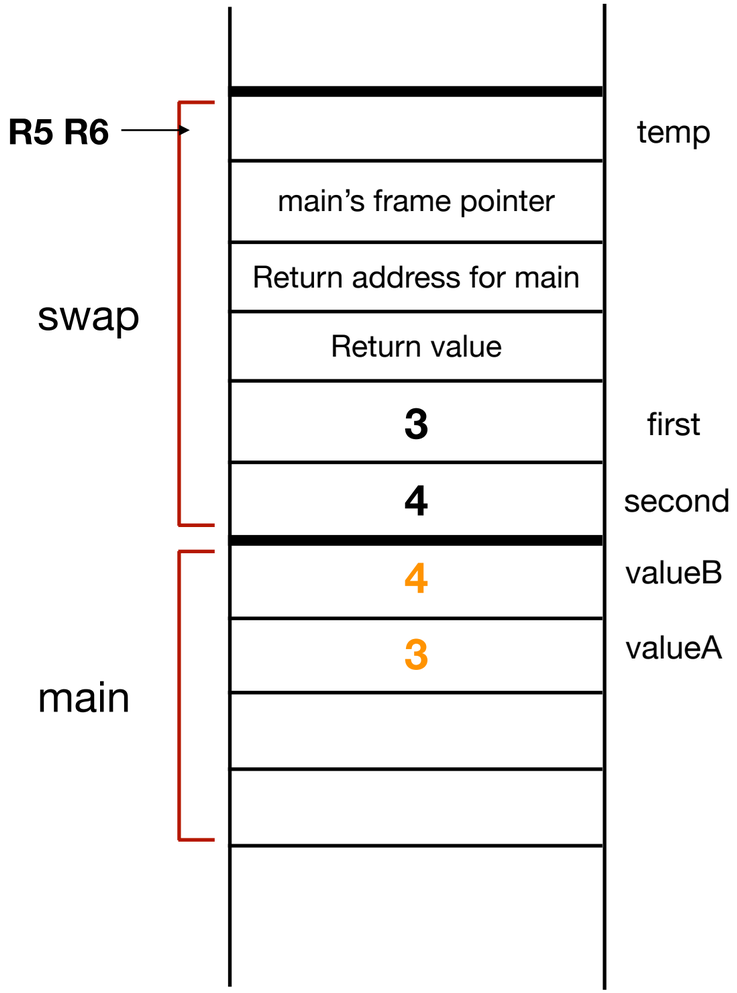



```
void Swap(int first, int second);

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 - 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

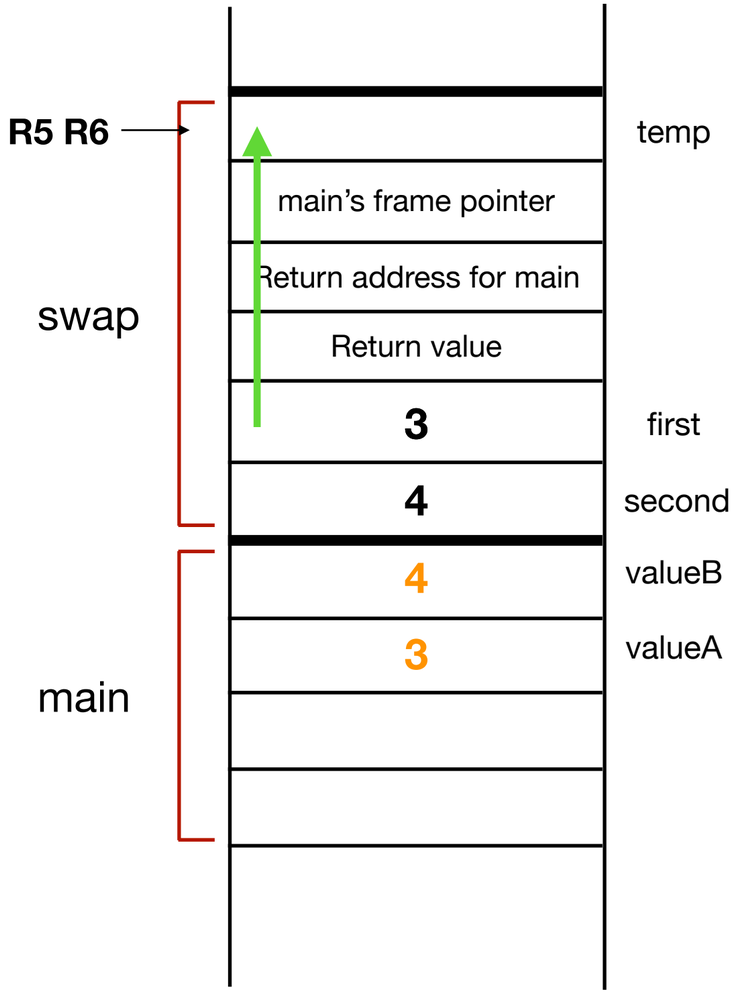



```c
void Swap(int first, int second);

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 - 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

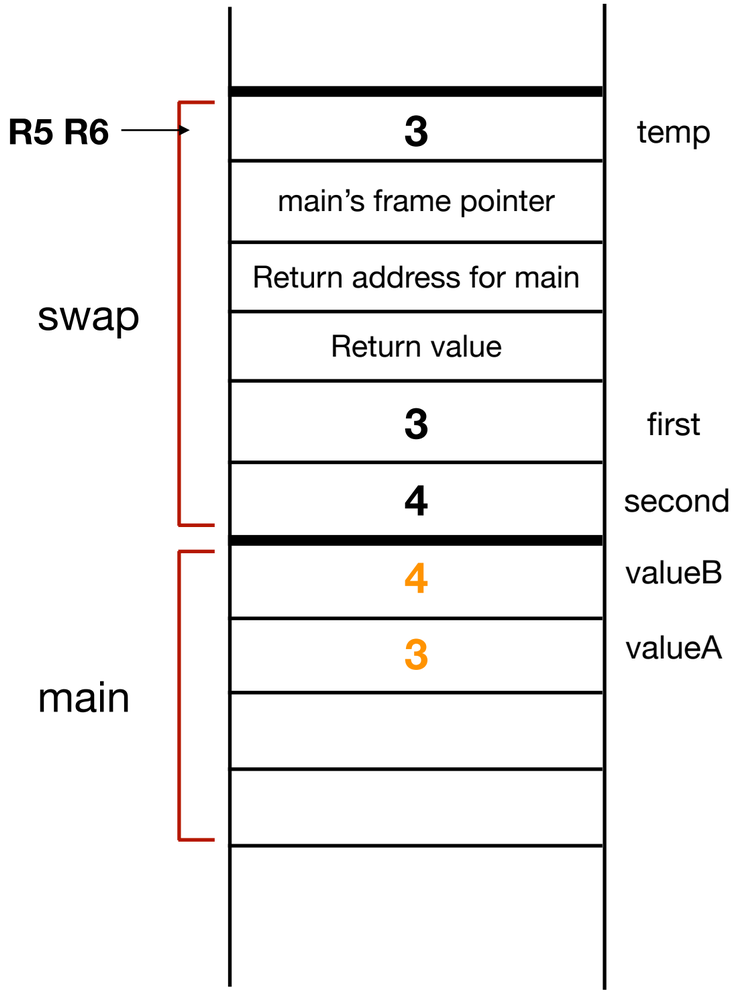



```c
void Swap(int first, int second);

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 - 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

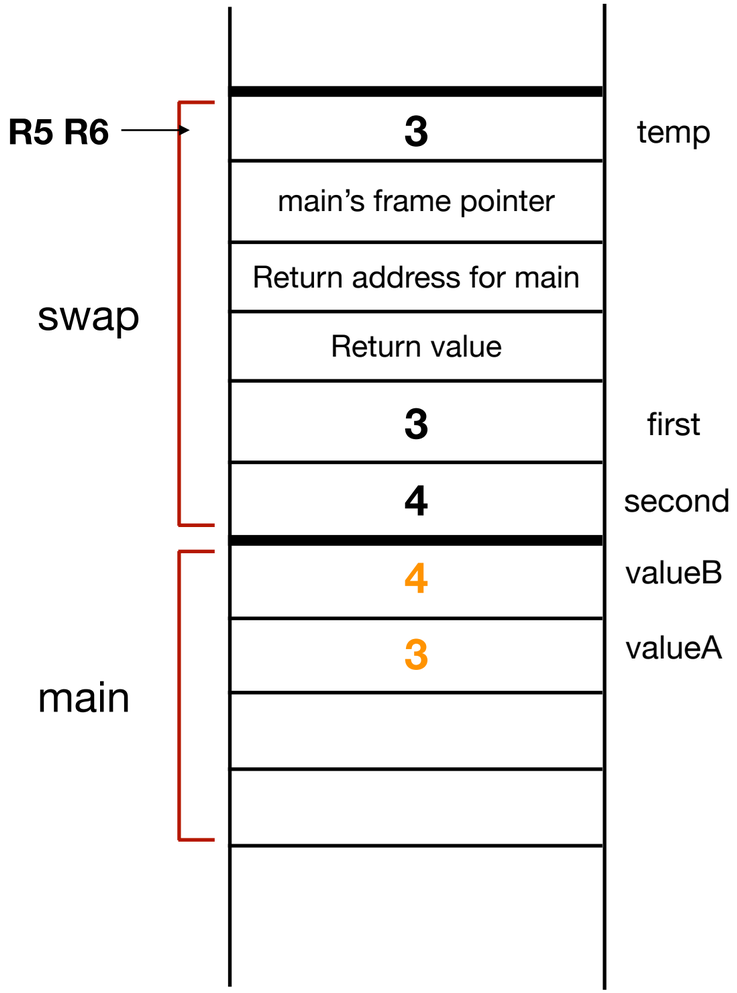



```c
void Swap(int first, int second);

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 - 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

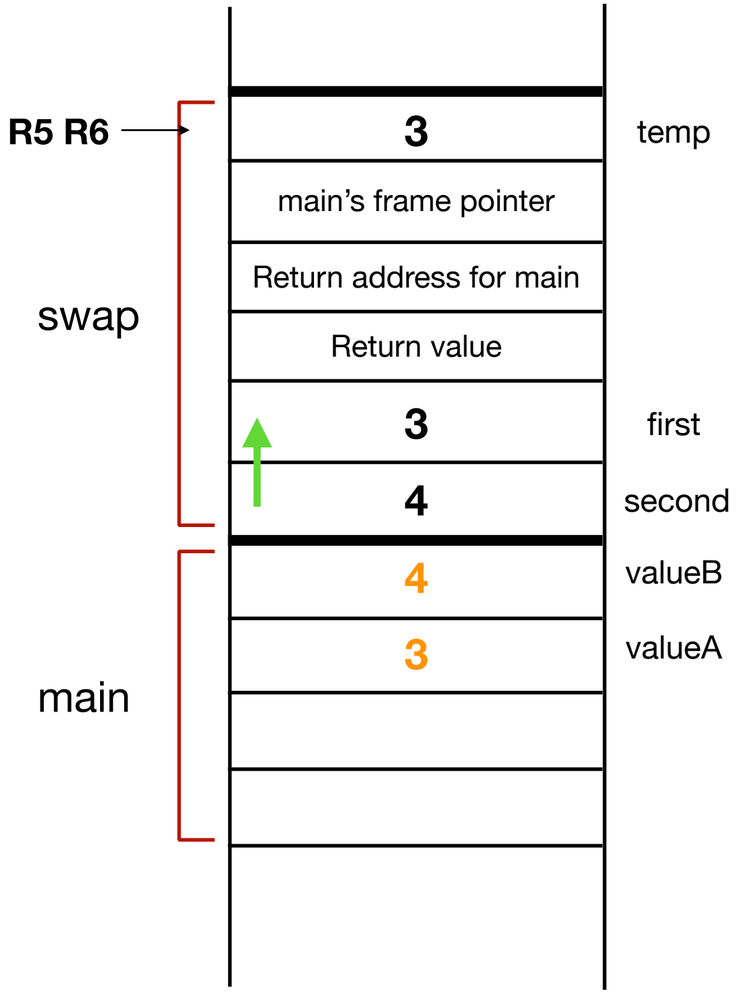



```c
void Swap(int first, int second);

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 - 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

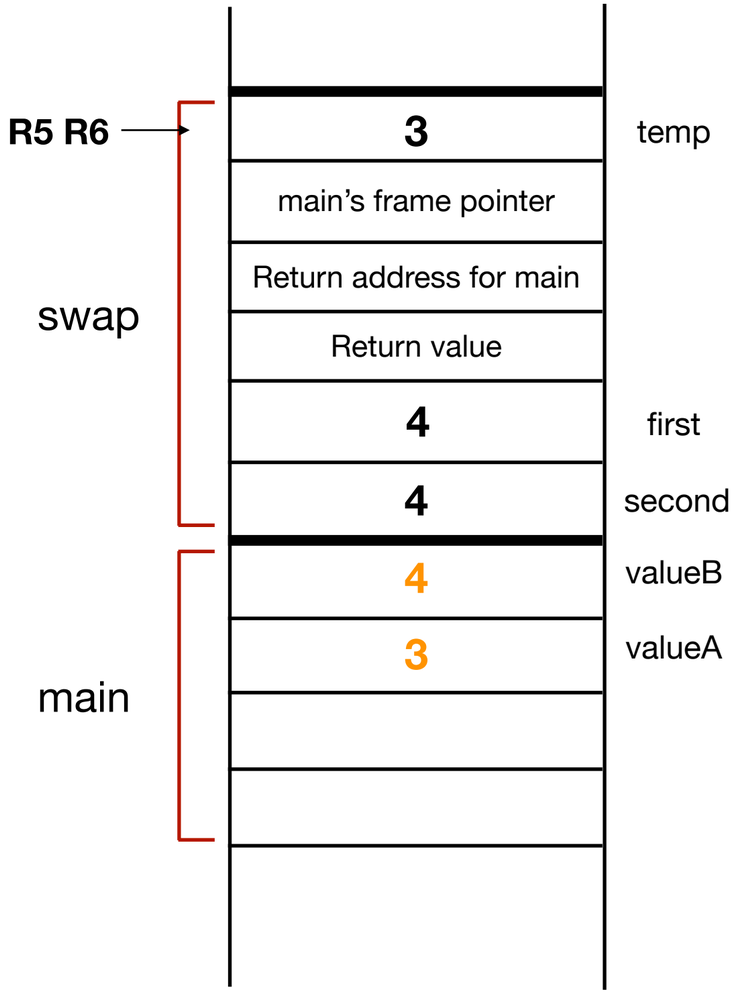



```
void Swap(int first, int second);

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 - 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

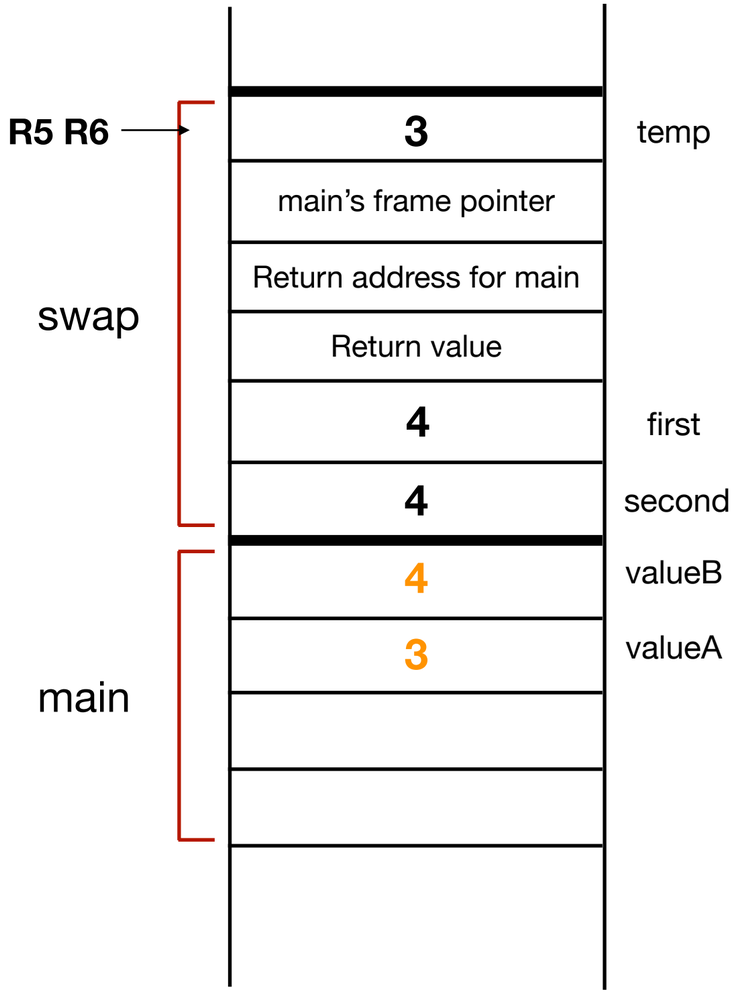



```
void Swap(int first, int second);

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 - 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

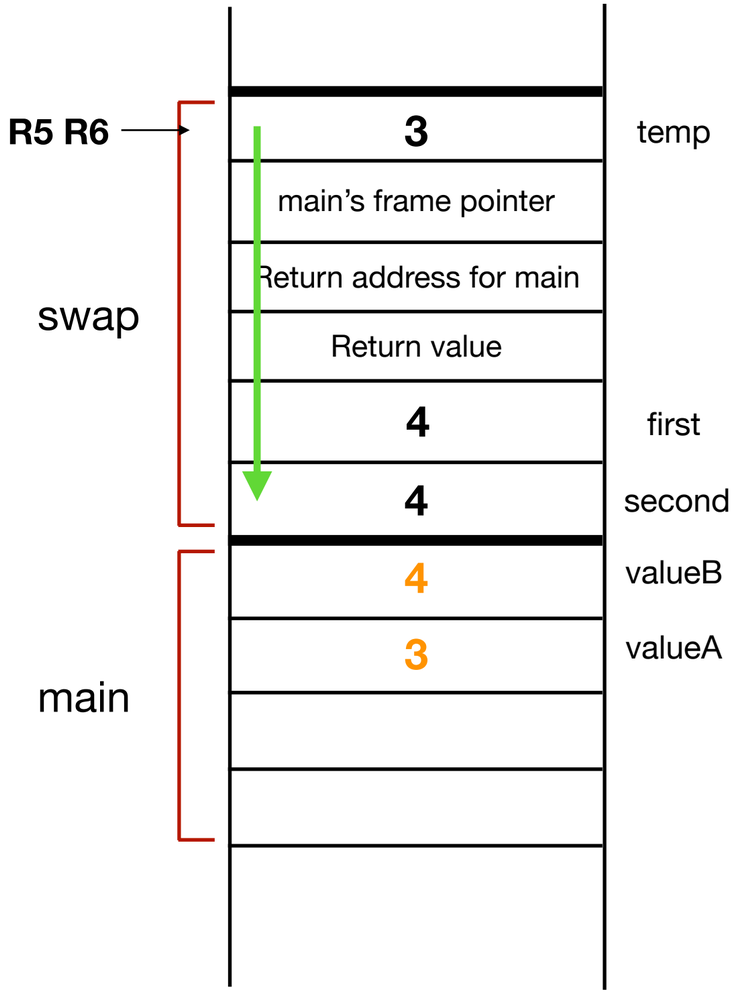



```
void Swap(int first, int second);

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 - 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

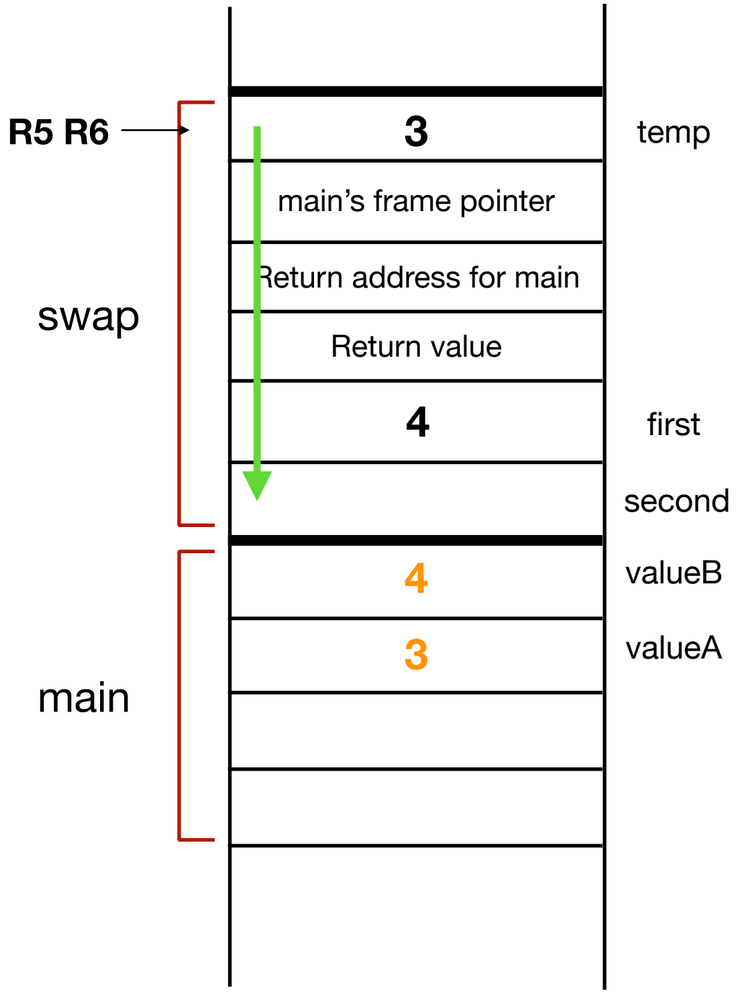



```
void Swap(int first, int second);

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 - 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

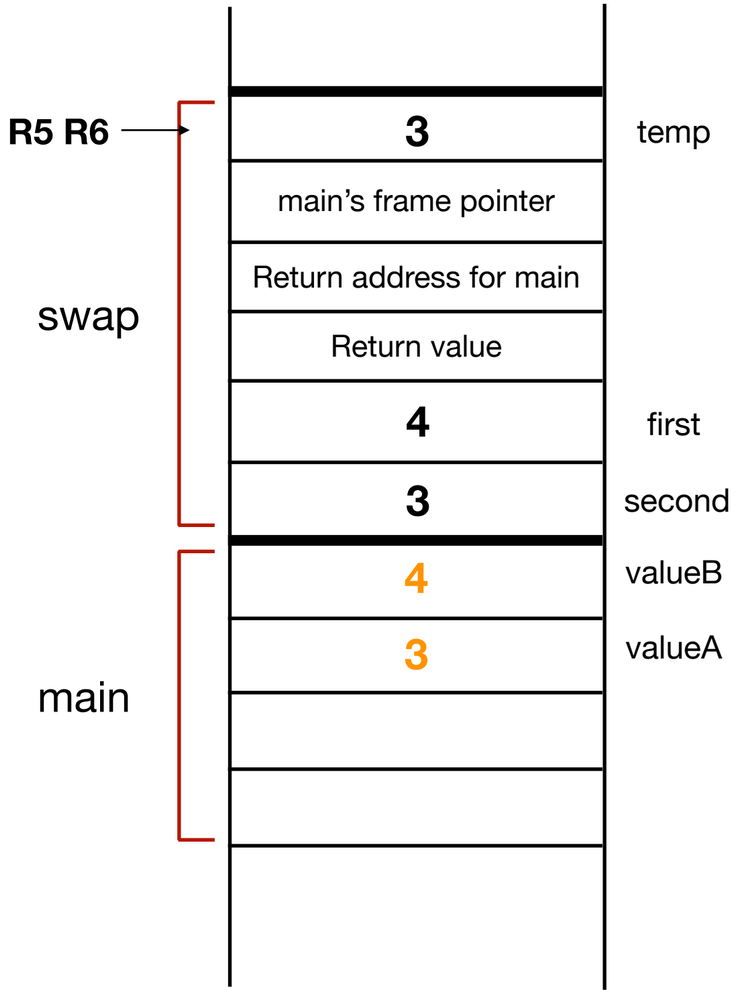



```
void Swap(int first, int second);

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 - 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

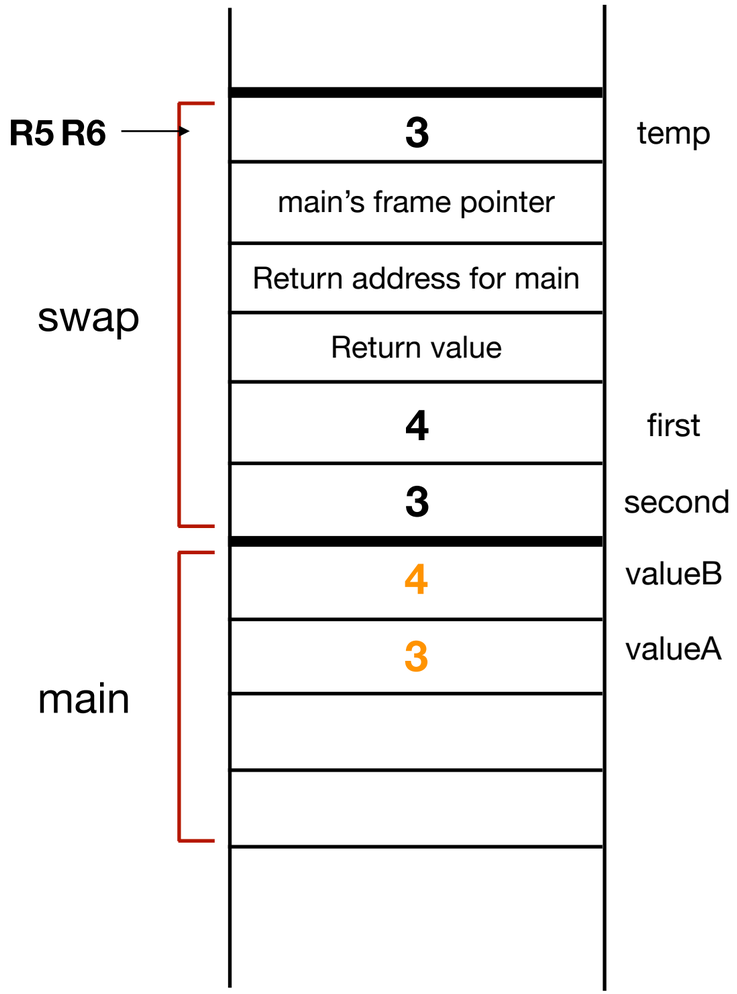
Execution




```c
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

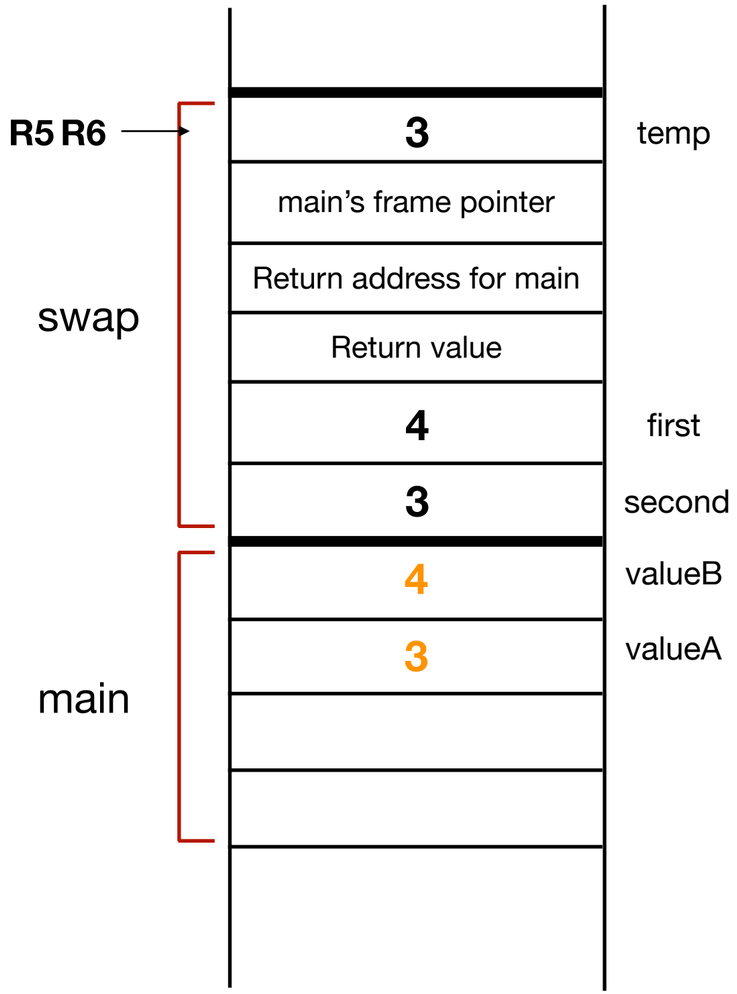



```c
void Swap(int first, int second);

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

`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

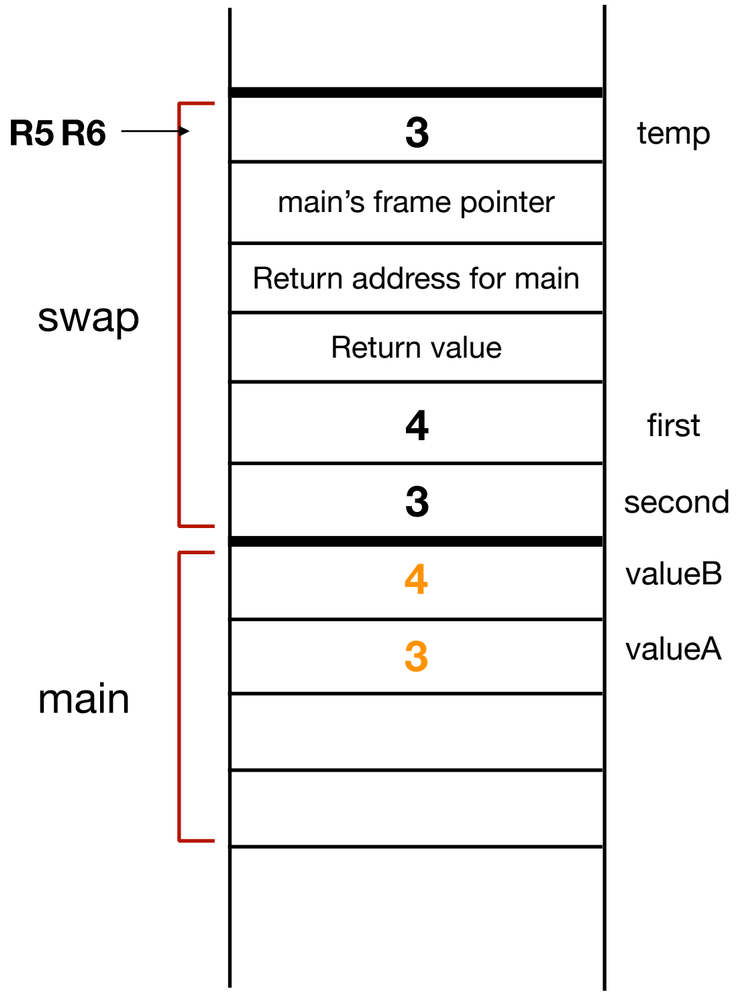



```c
void Swap(int first, int second);

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

`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



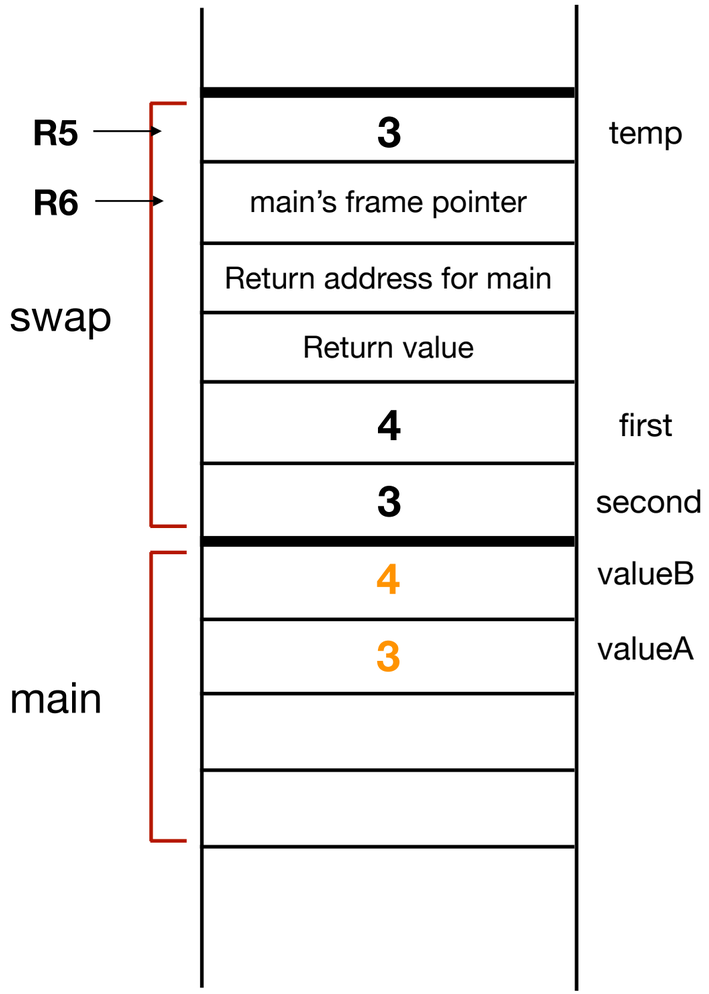

```c
void Swap(int first, int second);

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

`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

Tear down



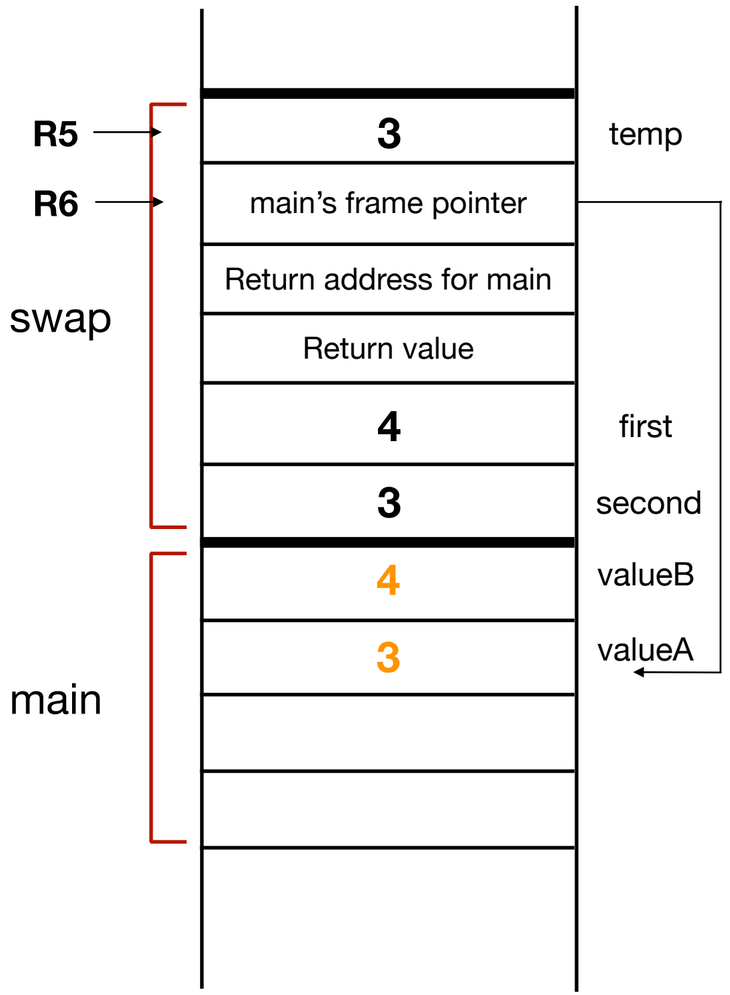

```
void Swap(int first, int second);

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

`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

Tear down



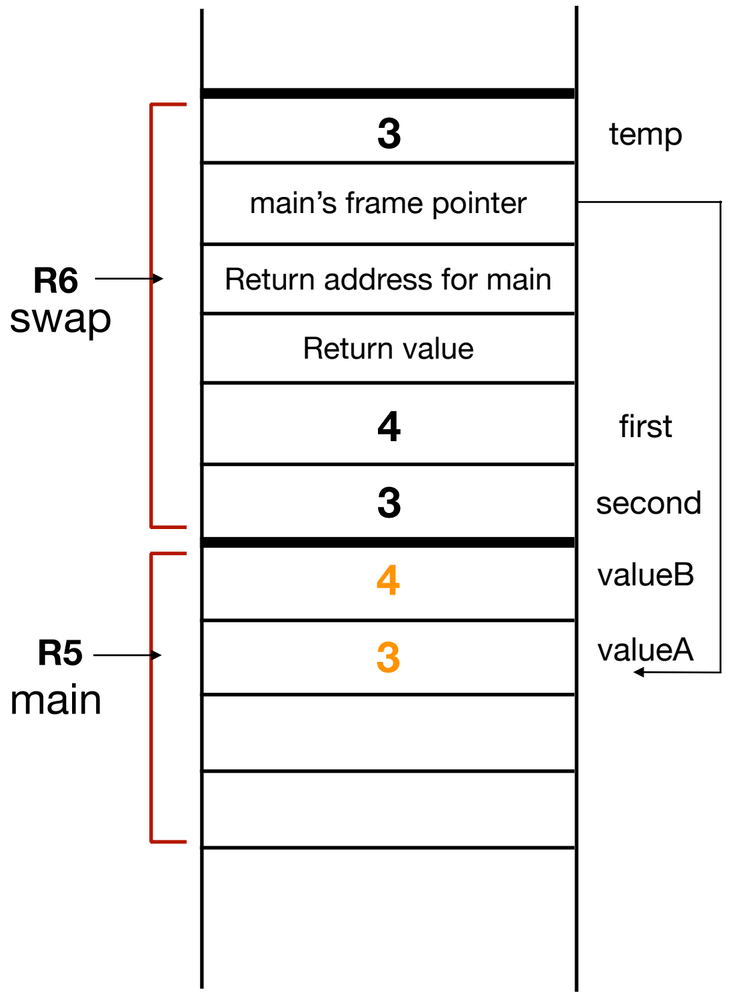

```
void Swap(int first, int second);

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

`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

Tear down

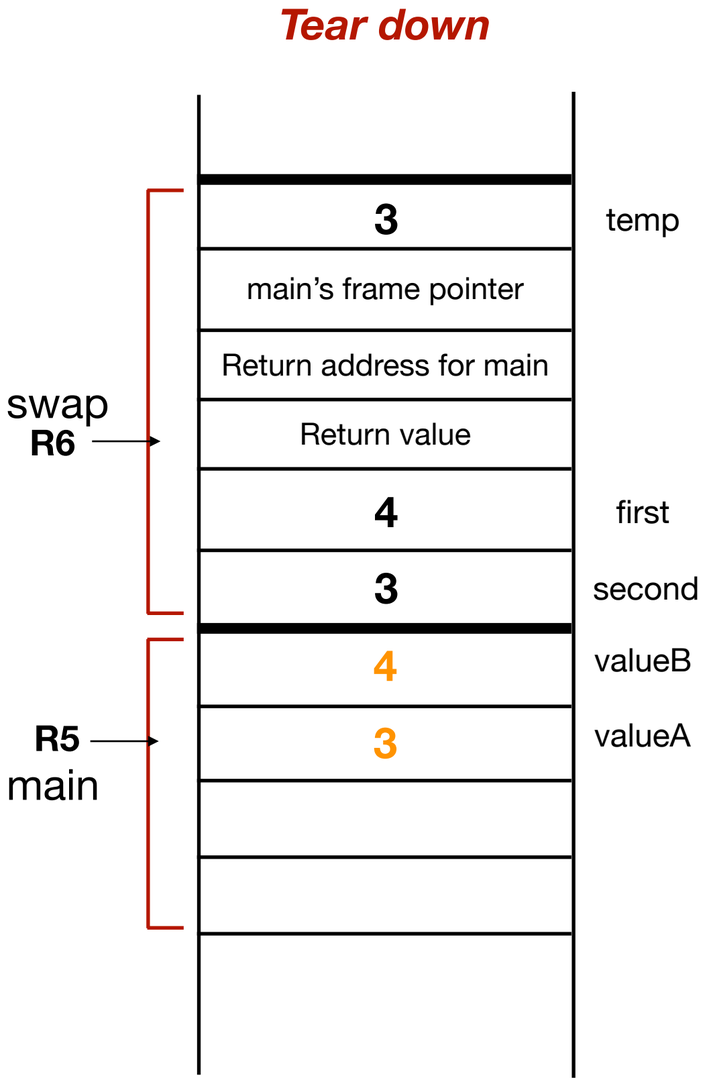


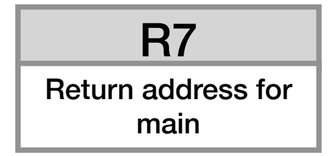

```
void Swap(int first, int second);

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`

`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



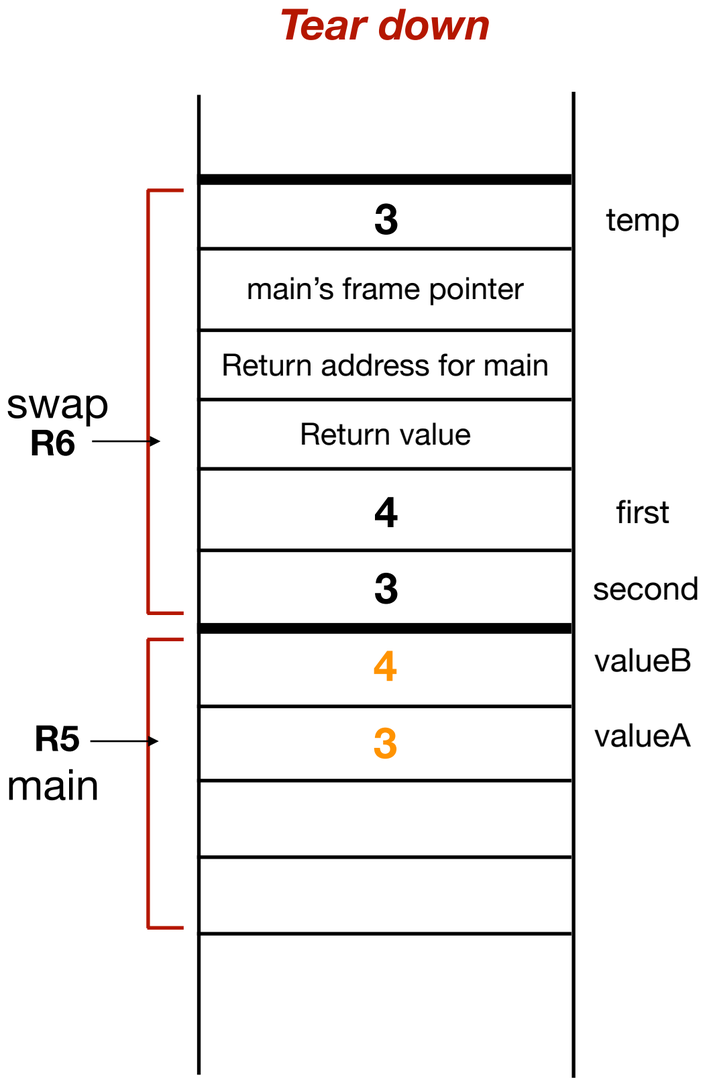


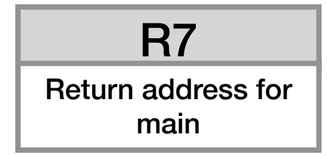

```c
void Swap(int first, int second);

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

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





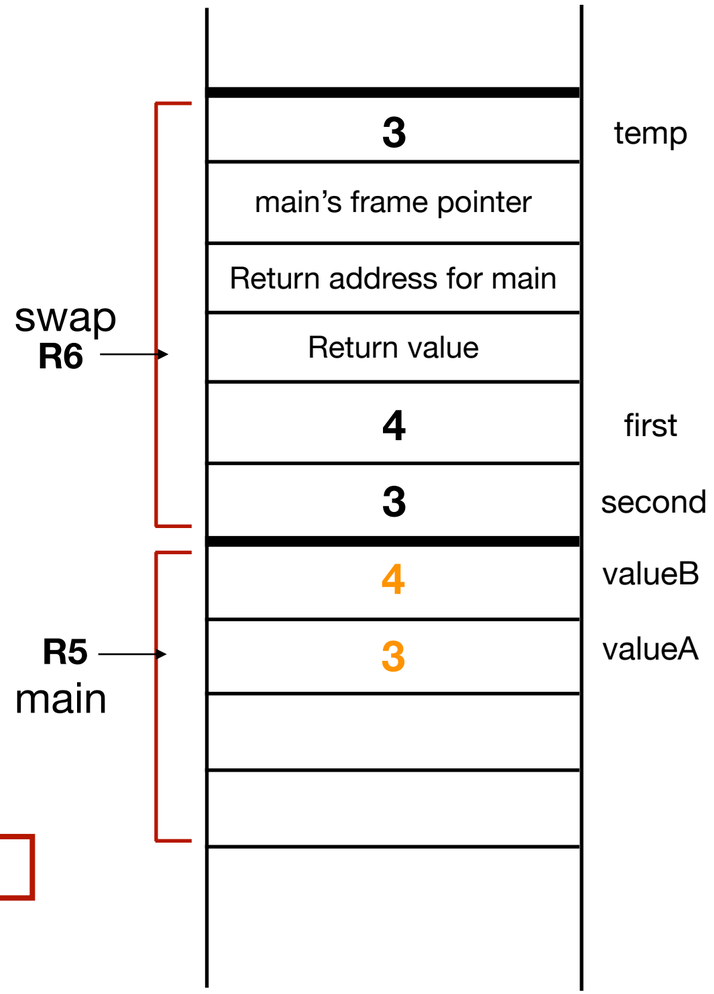

```c
void Swap(int first, int second);

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

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

Tear down



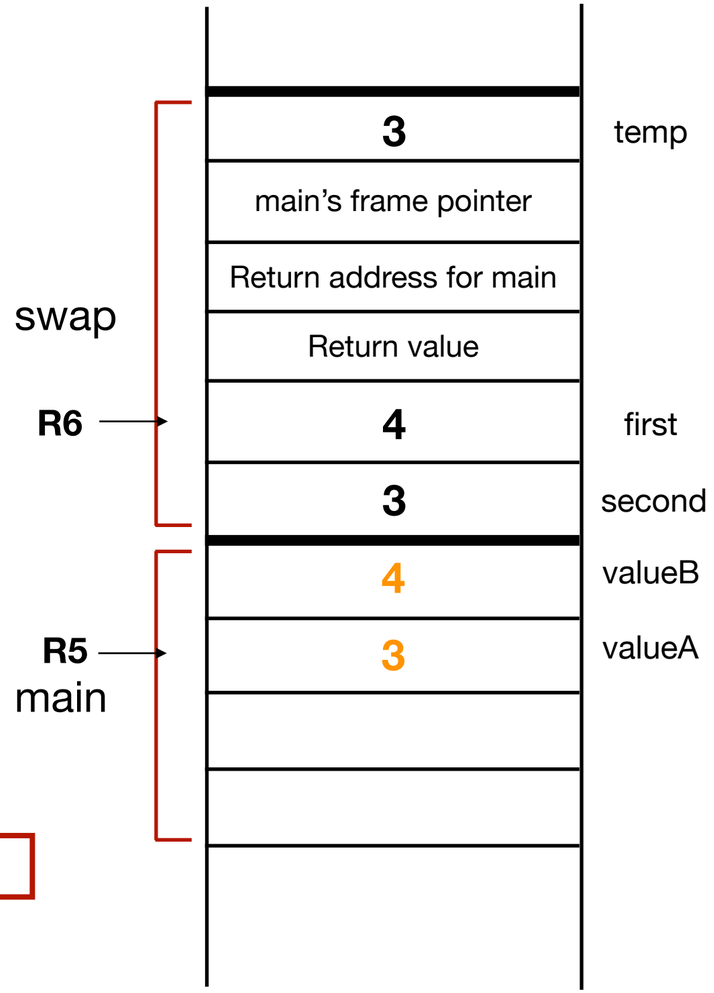

```
void Swap(int first, int second);

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

`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

Tear down



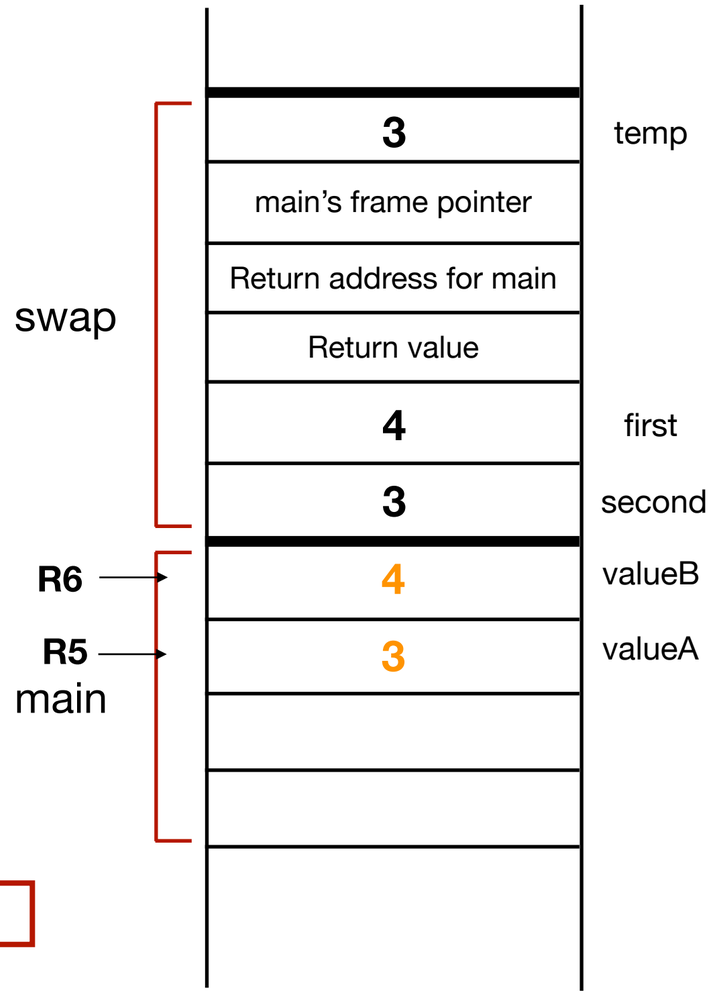

```
void Swap(int first, int second);

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

`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

Tear down

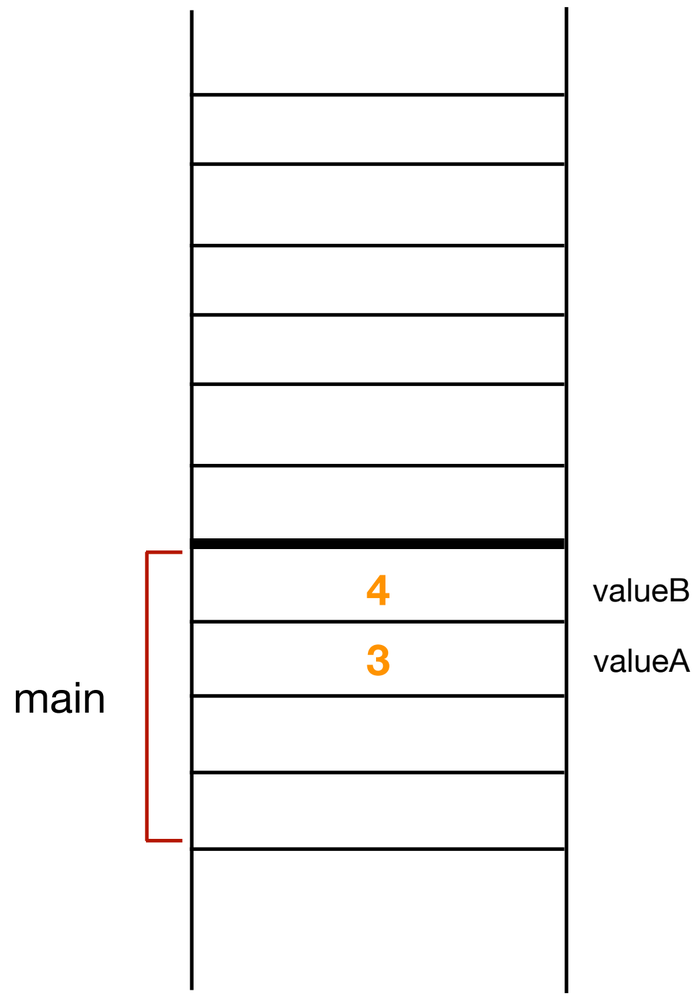



```
void Swap(int first, int second);

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;
}
```

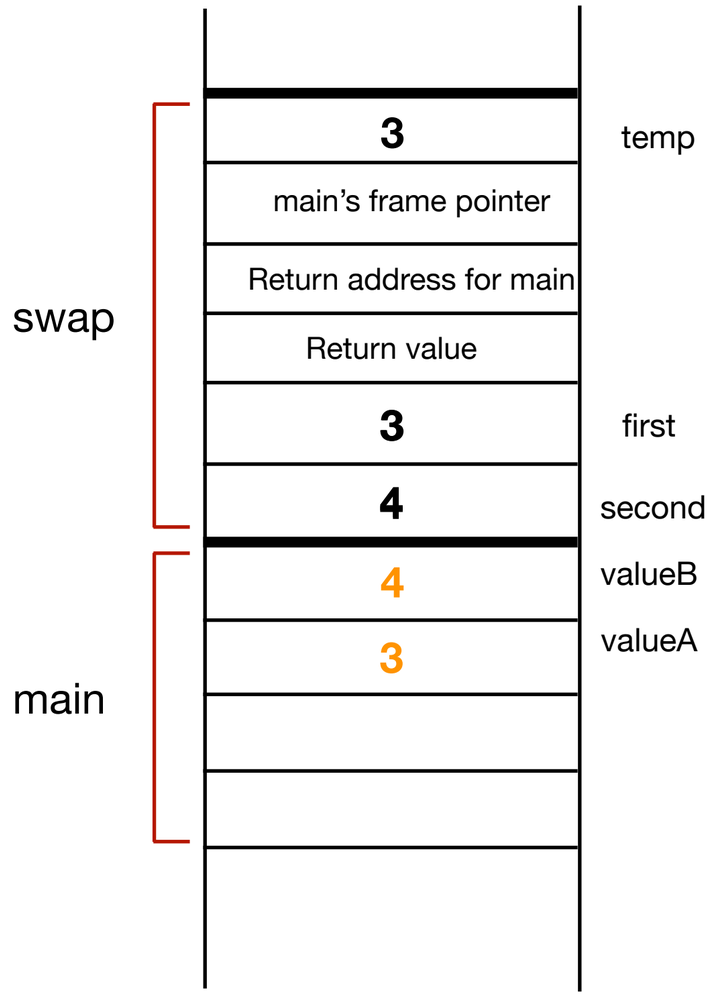
LC3 commands left as an exercise

Swap function - did it work?

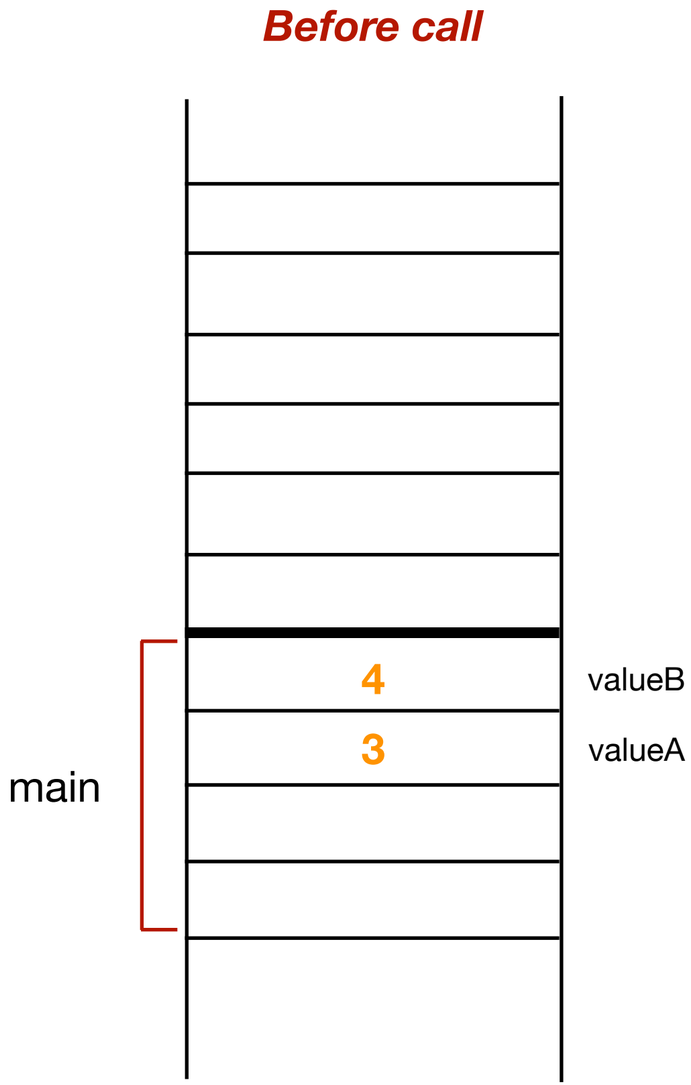
Before call

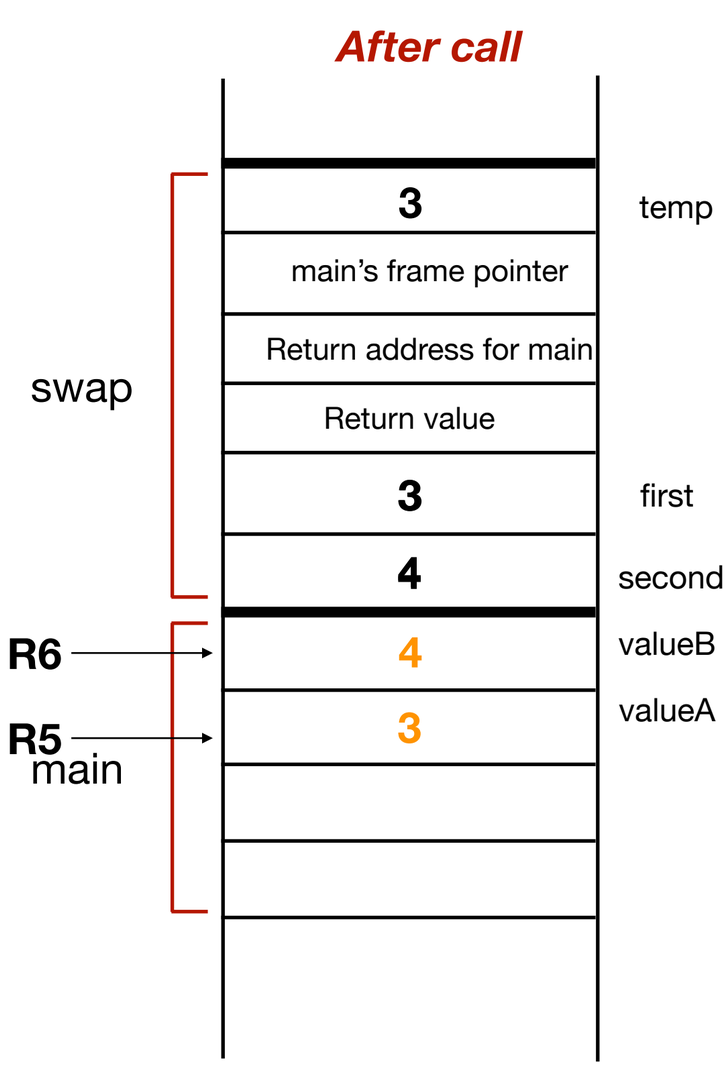


After call



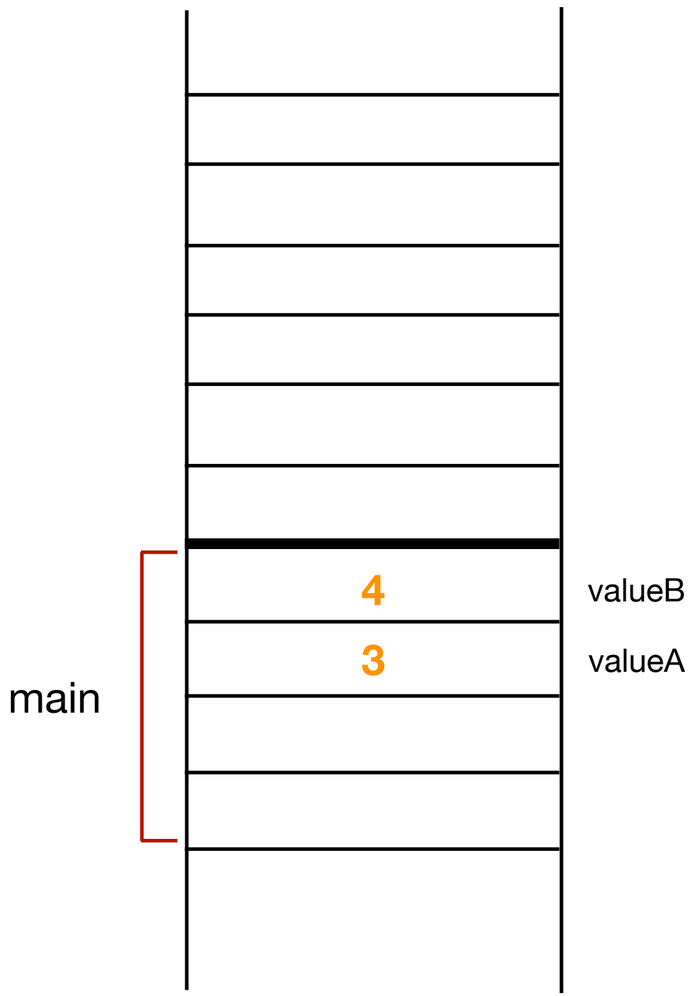
Swap function - did it work?



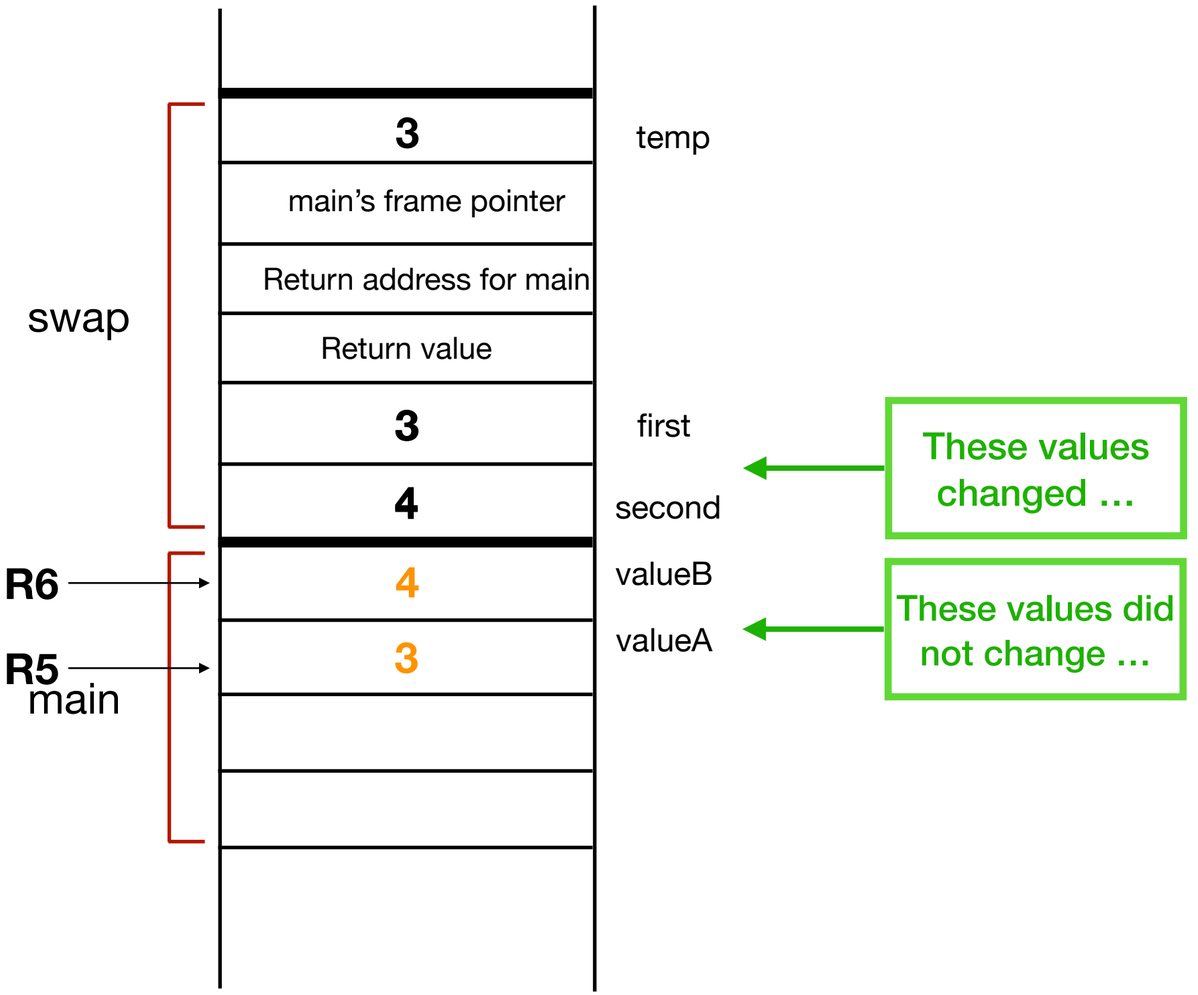


Swap function - did it work?

Before call



After call



Argument passing

Argument passing

- Argument passing in C is what we call **pass-by-value**:

Argument passing

- Argument passing in C is what we call **pass-by-value**:
    - The functions get their own copies of the arguments

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

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**.

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?

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

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

```
#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;
}
```

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;
}
```

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`?

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.ka *pointers* and not usual values?

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.ka *pointers* and not usual values?

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.ka *pointers* and not usual values?

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

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.ka *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.