ECE 220: Computer Systems & Programming

Lecture 8: Run-time Stack

Midterm 1:

Thursday, 2/15/2024

- Regular Exam: 7:00pm - 8:20pm CT
- ~~Conflict Exam: 5:40pm - 7:00pm CT~~
- **Conflict Exam: Follow the instruction on the course website**
- **Conflict Sign-Up is due by 11:59pm CT on Sunday, 2/11/2024.**

Basics of Function in C

```c
/* generate 5 random number between a and b */
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int rand_gen(int a, int b);


int main()
{
        int a,b,i;
       printf("Enter the rand a and b:  ");
        scanf("%d%d",&a, &b);
        srand(time(0));

        for(i=0; i<5; i++)
                printf("%d ", rand_gen(a,b));
        printf("\n");
        return 0;
}
```

```c
int rand_gen(int a, int b)
{
        return (rand()%(b-a+1)+a);
}
```

Four basic phases in the execution of a function call:

- Argument values from the caller are passed to the callee,
- control is transferred to the callee,
- the callee executes its task, and
- control is passed back to the caller, along with a return value.

Local Variables in Activation Record

- Every *function call* creates an *activation record (or stack frame)* and *pushes* it onto the run-time stack.
- Local variables are one part of the activation record.
- Whenever a function *completes (return)*, the activation record is *popped* off the run-time stack.
- Whenever a function calls another one (*nested*), the run time stack *grows* (push another activation record onto the run-time stack).

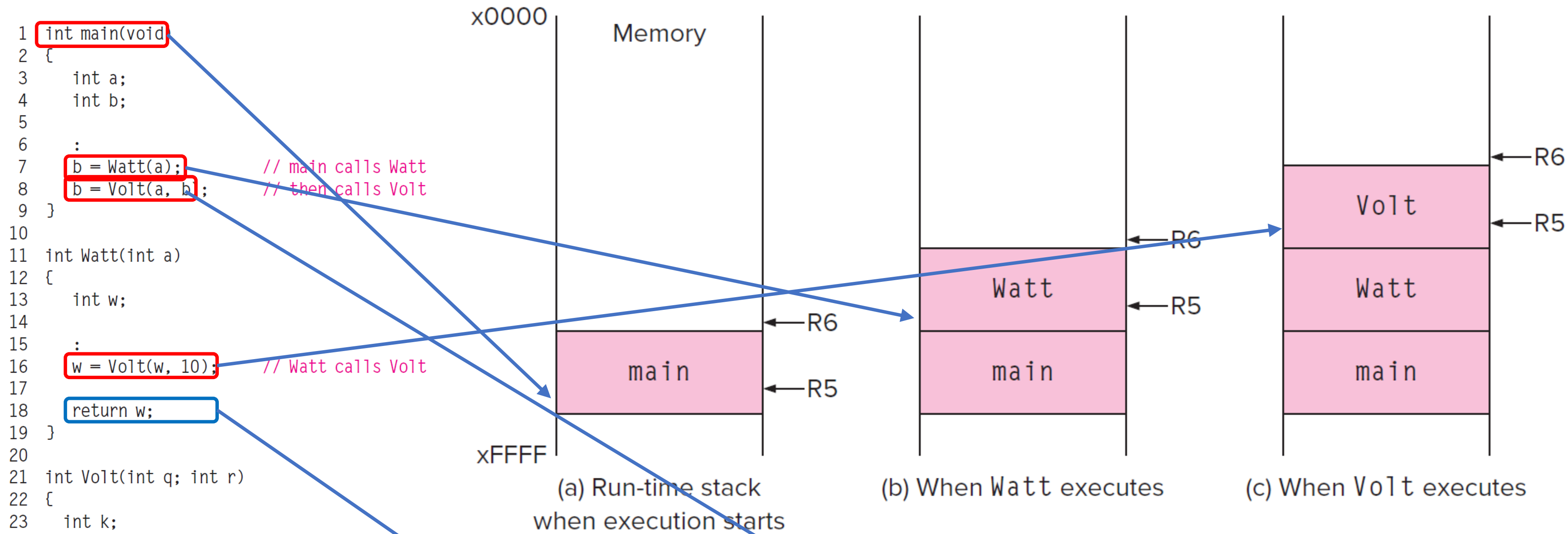
one function call = one activation record

A function could call **itself** (recursion)

Example

```
 1  int main(void)
 2  {
 3     int a;
 4     int b;
 5
 6       :
 7     b = Watt(a);          // main calls Watt first
 8     b = Volt(a, b);       // then calls Volt
 9  }
10
11  int Watt(int a)
12  {
13     int w;
14
15       :
16     w = Volt(w, 10);      // Watt calls Volt
17
18     return w;
19  }
20
21  int Volt(int q; int r)
22  {
23    int k;
24    int m;
25
26     :
27    return k;
28  }
```

Figure 14.4 Code example that demonstrates the stack-like nature of function calls.



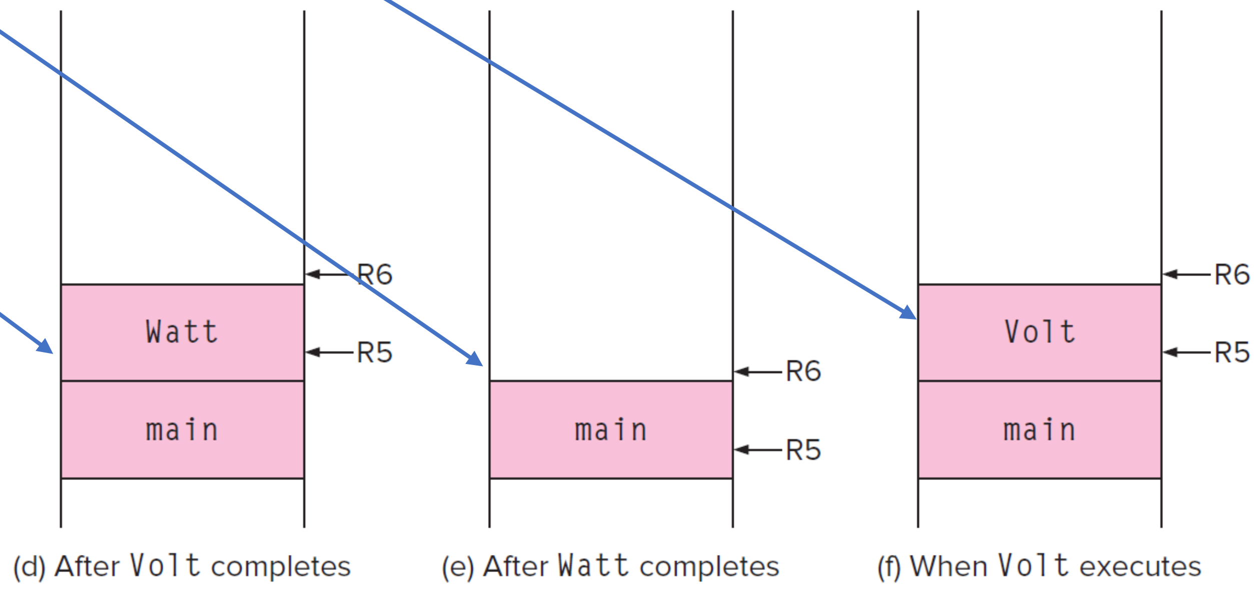


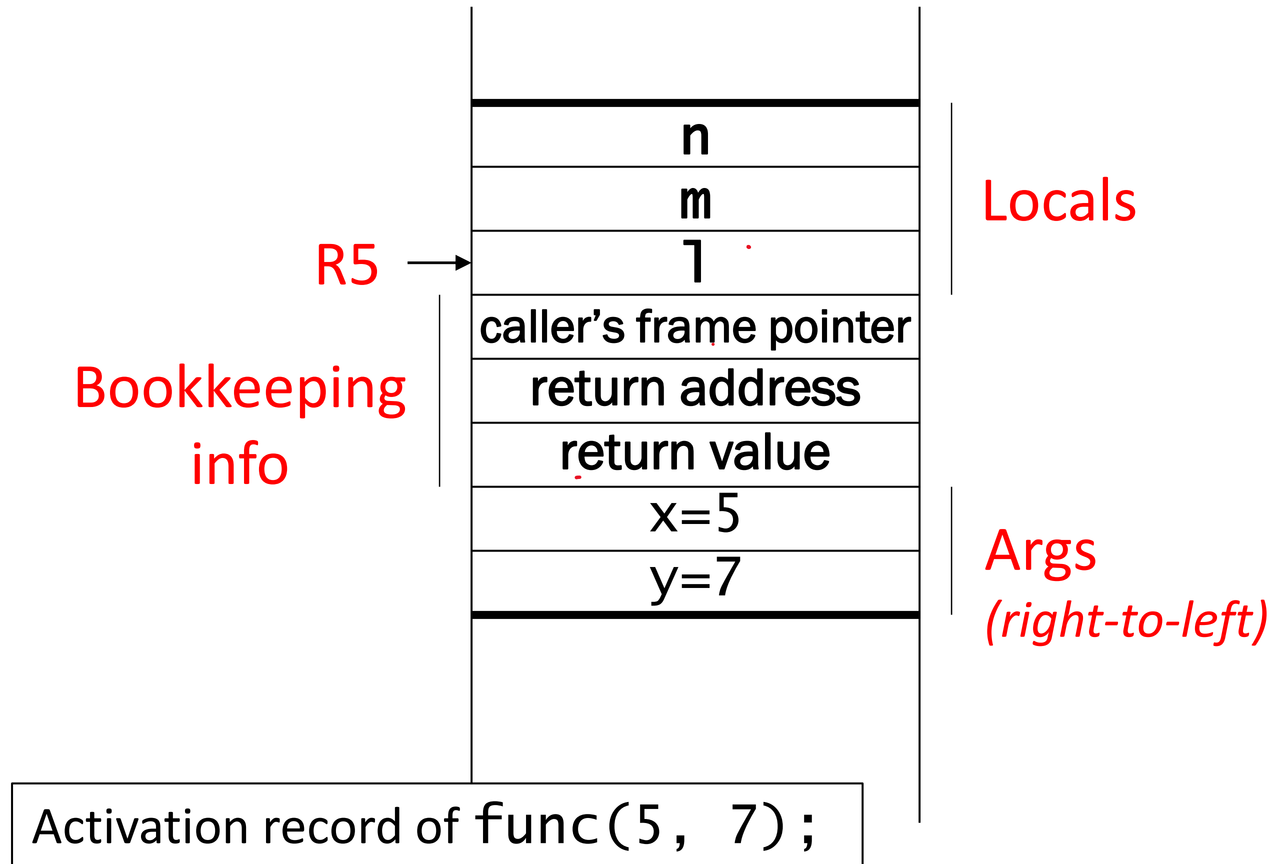
Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.

Building Activation Record

- Information about each function call, including

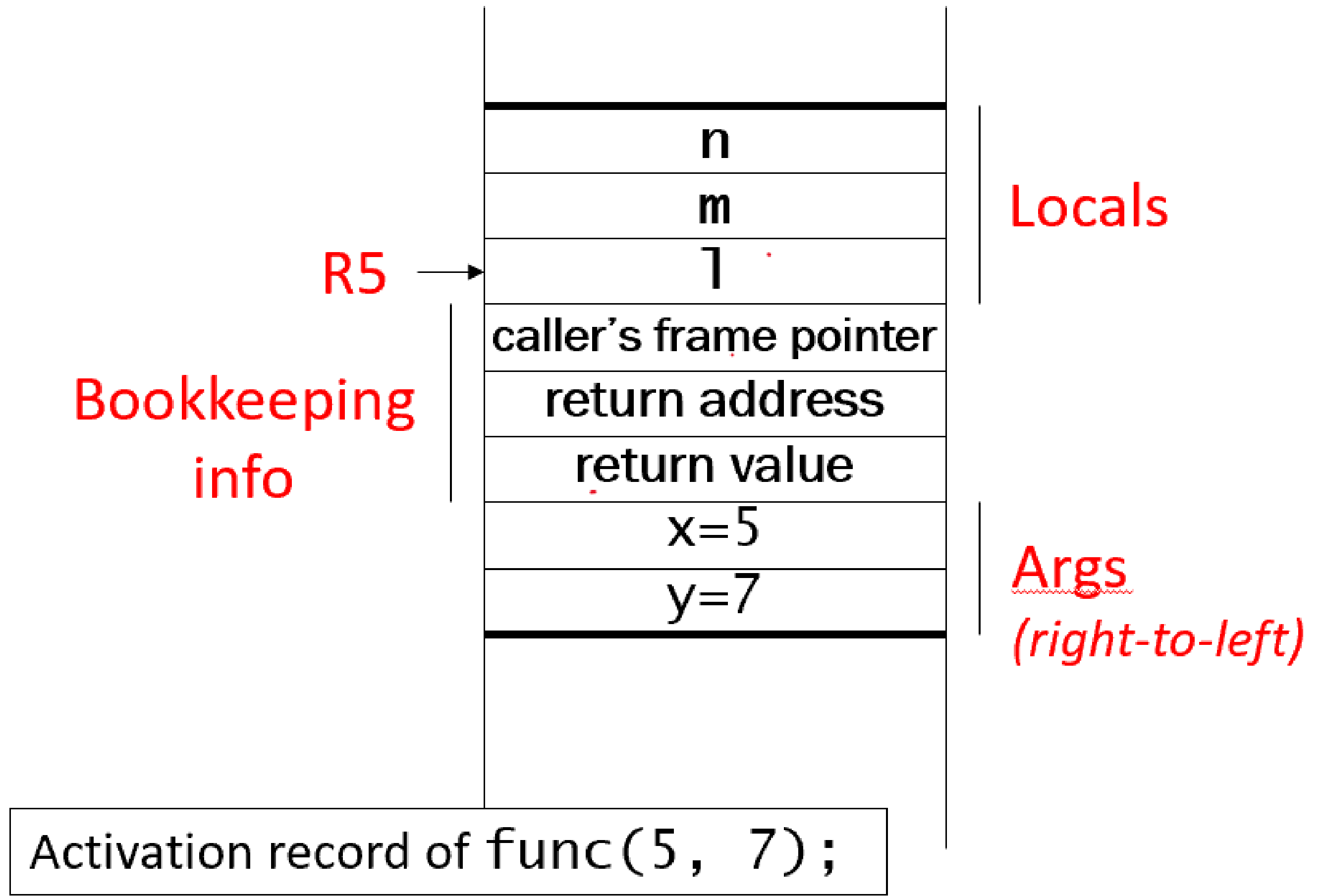
1. Arguments *2. Bookkeeping info* *3. Local variables*

```
int main()
{
    int x,y,z;
    x = func(5, 7);

}
int func(int x, int y)
{
    int l,m,n;
    .
    .
    .
    return l;
}
```



Bookkeeping info part:

```
int main()
{
    int x,y,z;
    x = func(5, 7);

}
int func(int x, int y)
{
    int l,m,n;
    .
    .
    .
    return l;
}
```



Stack Build-up and Tear-down

Caller function

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

Callee function

3. Callee setup (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)

Caller function

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

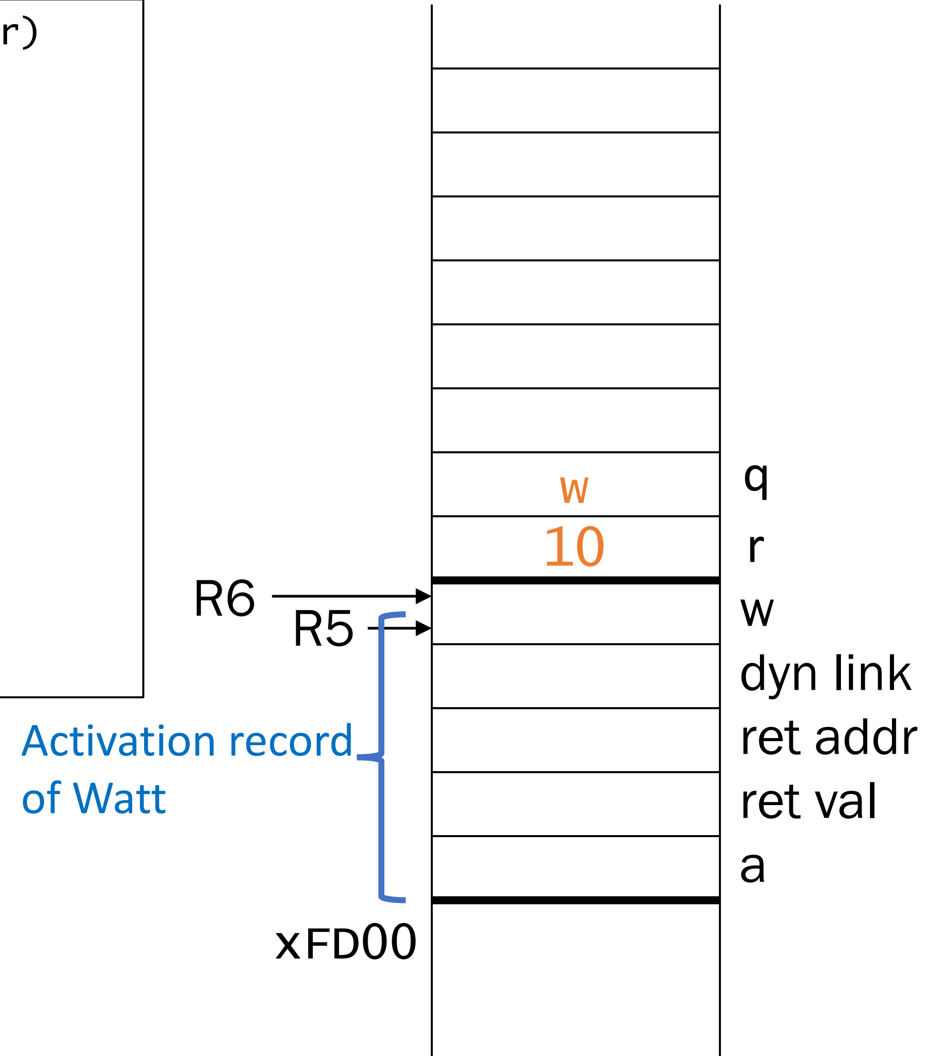
Example Function Call

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


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

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

```
Watt
...
; 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   ; R0<- w
ADD   R6, R6, #-1
STR   R0, R6, #0

; call subroutine
JSR   Volta
```

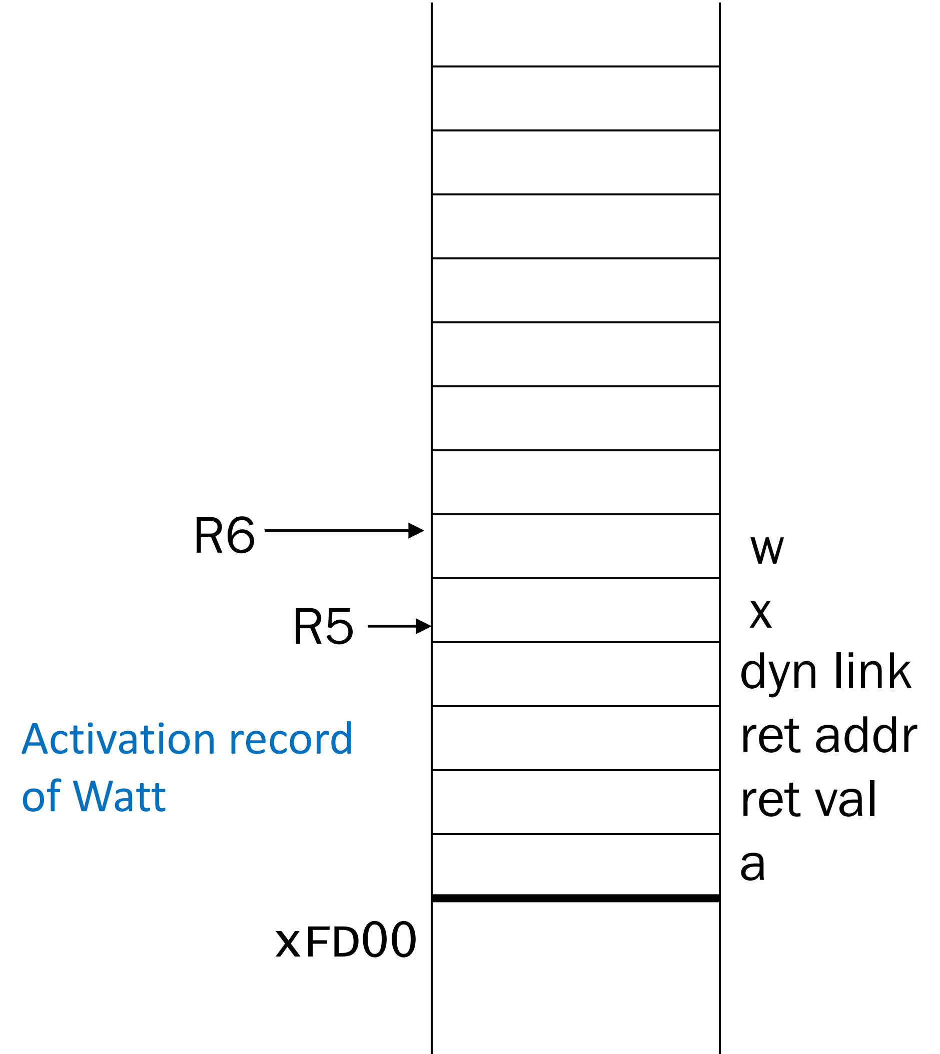
```
int Volta(int q, int r)
{
   int k;
   int m;
   ...
   return k;
}
int Watt(int a)
{
   int w;
   ...
   w = Volta(w,10);
   ...
   return w;
}
```



A different scenario

```
Watt:
...
; 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, #-1
ADD   R6, R6, #-1
STR   R0, R6, #0

; call subroutine
JSR   Volta
```

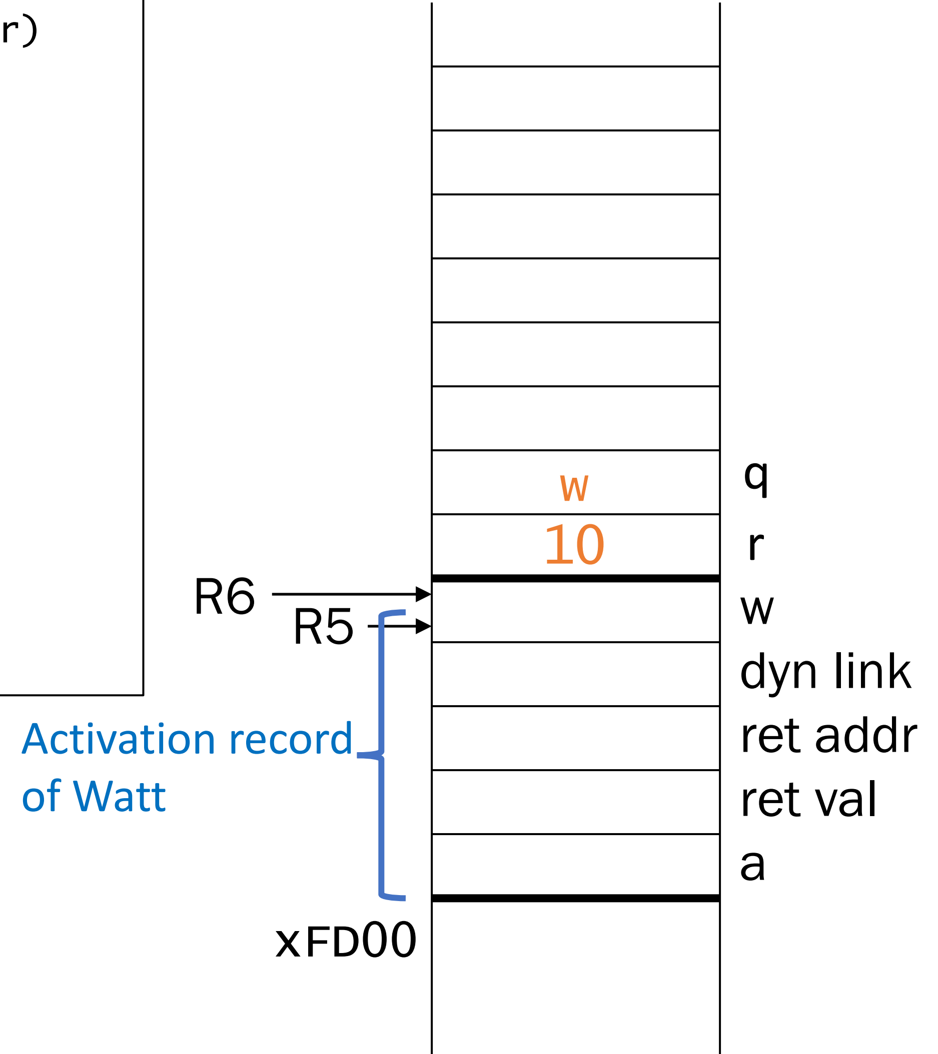
```
int Volta(int q, int r)
{
   int k;
   int m;
   ...
   return k;
}
int Watt(int a)
{
   int x, w;
   ...
   w = Volta(w,10);
   ...
   return w;
}
```



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

```
Watt
...
; 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   ; R0<- w
ADD   R6, R6, #-1
STR   R0, R6, #0

; call subroutine
JSR   Volta
```

```
int Volta(int q, int r)
{
   int k;
   int m;
   ...
   return k;
}
int Watt(int a)
{
   int w;
   ...
   w = Volta(w,10);
   ...
   return w;
}
```

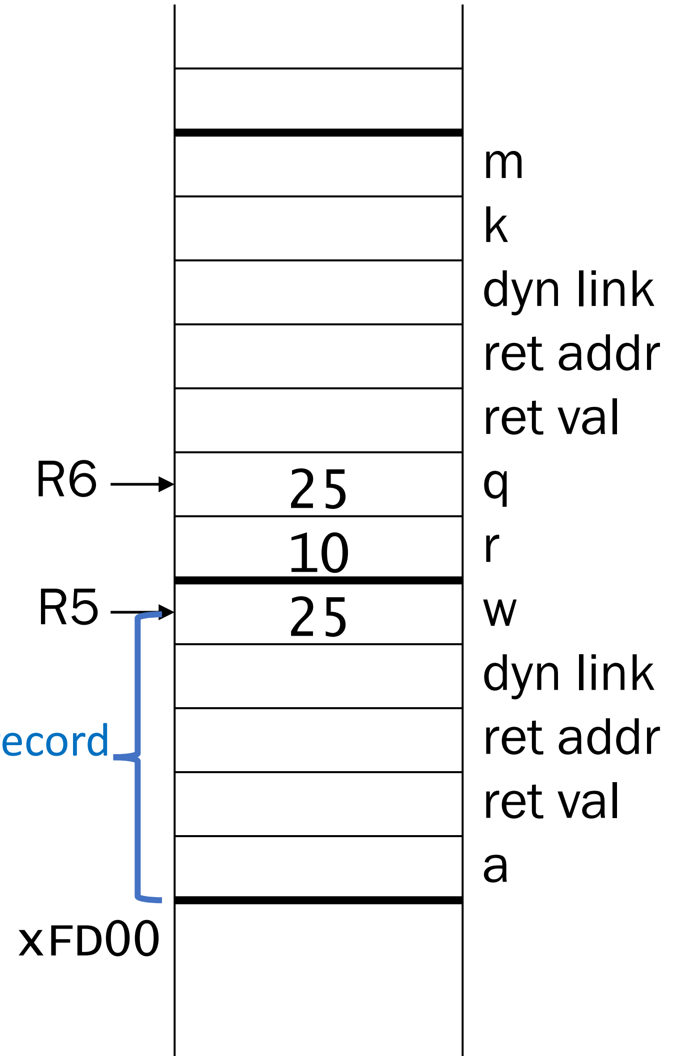


3. Callee setup (push bookkeeping info and local variables onto stack)
4. Execute function

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

depends on # local var

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



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

Credit: Prof. Moon

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

depends on # local var

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

		m
		k
	xFCFB	dyn link
	x3100	ret addr
		ret val
R6 →	25	q
	10	r
R5 →	25	w
		dyn link
		ret addr
		ret val
		a
xFD00		

Activation record of Volta (m through r)

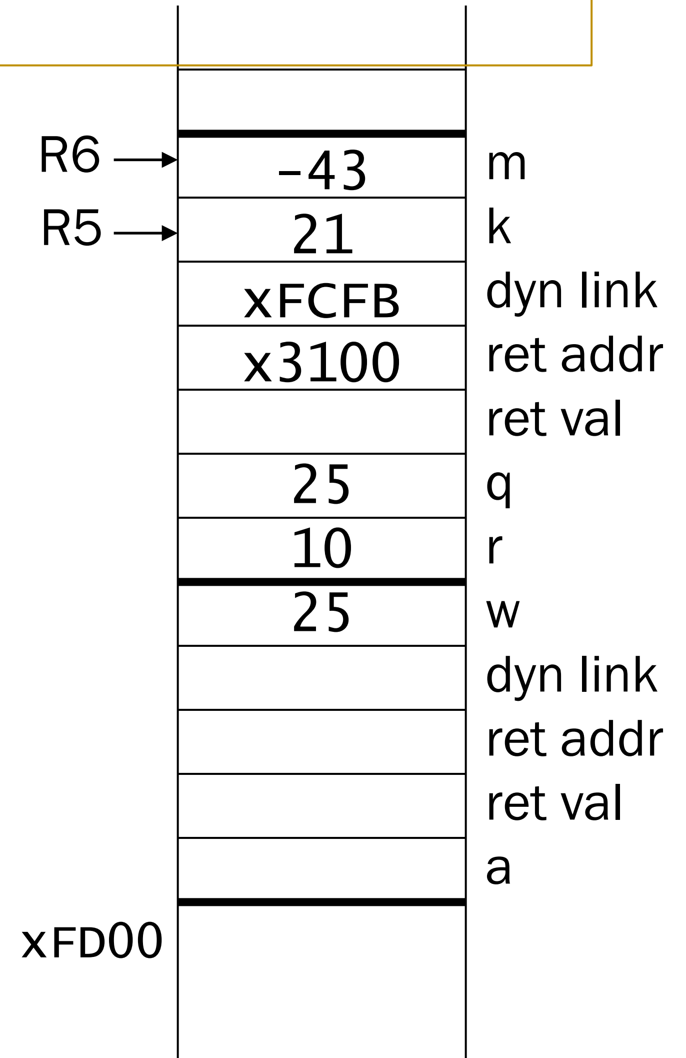
Activation record of Watt (w through a)

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
LDR   R0, R5, #0
STR   R0, R5, #3
; pop local variables
ADD   R6, R5, #1
; pop dynamic link (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
```

depends on which variable

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



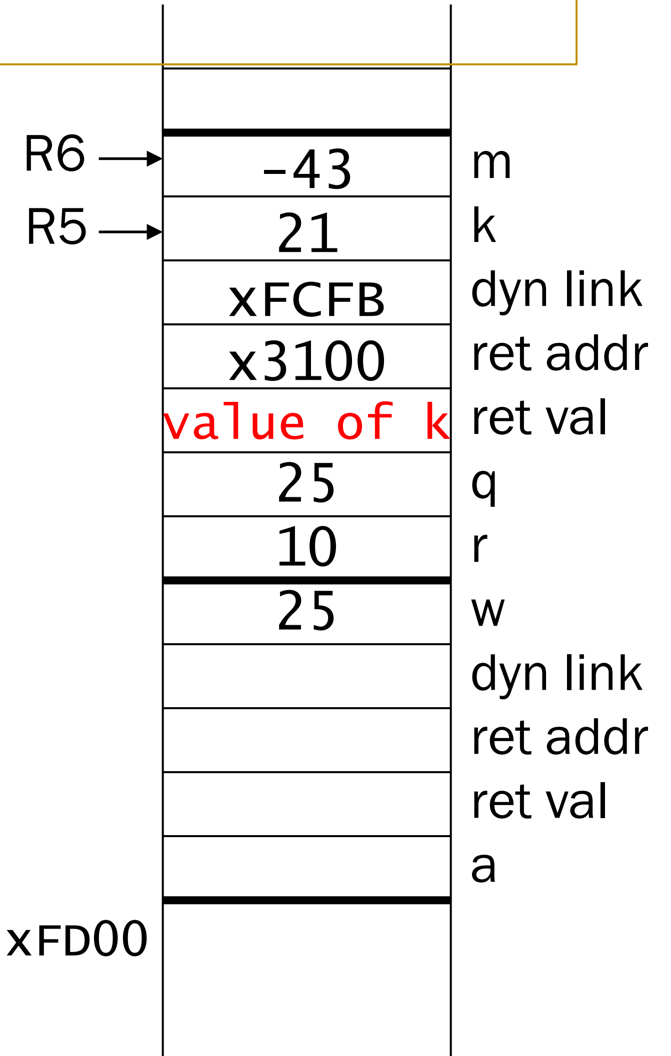
5. Callee tear-down (update return value, pop local variables, caller's frame pointer and return address from stack)
6. Return to caller

Credit: Prof. Moon

```
; copy k into return value
LDR   R0, R5, #0
STR   R0, R5, #3
; pop local variables
ADD   R6, R5, #1
; pop dynamic link (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
```

depends on which variable

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



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

```
JSR   Volta
; load return value (top of
stack)
LDR   R0, R6, #0
; perform assignment
STR   R0, R5, #0
```
depends on which variable

```
; pop return value
ADD   R6, R6, #1

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

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

Value	Label
-43	m
21	k
xFCFB	dyn link
x3100	ret addr
21 (← R6)	ret val
25	q
10	r
25 (← R5)	w
	dyn link
	ret addr
	ret val
	a
xFD00	

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

```
JSR   Volta
; load return value (top of stack)
LDR   R0, R6, #0
; perform assignment
STR   R0, R5, #0
```

depends on which variable

```
; pop return value
ADD   R6, R6, #1

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

depends on # variables

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

Pointer	Value	Label
	-43	m
	21	k
	xFCFB	dyn link
	x3100	ret addr
R6 →	21	ret val
	25	q
	10	r
R5 →	21	w
		dyn link
		ret addr
		ret val
		a
xFD00		

Why Run-time Stack?

- Option1: Assign each activation record at fixed memory location

  Problem: What happen function A calls itself?

- Option2: Use Run-time Stack

  Each invocation of a function gets its own space in memory
  -> Permits functions to be ***recursive***!