ECE 220: Computer Systems & Programming

Lecture 8: Run-time Stack

Announcements:

Exam	Date & Time	Location	Topic	Practice Questions	Conflict Exam Information
Midterm 1	Thursday 09/25 at 7.00pm - 8.20pm	ECEB	Lecture 1 to Lecture 06 Associated book chapters, labs,and MPs. (Programming & Concept)	Past exams Worksheets LC3 RefSheet	Sign-up Link **Deadline: 09/21**

HKN review session (MT1): Sunday, 9/21, 3-5:30pm, ECEB 1002

Machine Problem	Submission due date	Points
MP 01 - Printing histogram	09-04	100 pts
MP 02 - Stack calculator	09-11	100 pts
MP 03 - Pascal's triangle	09-18	100 pts

Function Call –reimann integral

```
#include<stdio.h>
float reimann_int(int n, float a, float b);
int main()
{
  int n=100;
  float a=-1.0f;
  float b=1.0f;
  float s;
  s=reimann_int(n,a,b);
  printf("The integral of f(x) is %f\n", s);

  return 0;
}
```

```
float reimann_int(int n, float a, float b)
{
    float dx; int i;
    dx=(b-a)/n; float s=0.0;
    float x, y;
    for (i=0;i<n;i++) {
                x=a+dx*i;
                y=x*x+2*x+1;
                s+=y*dx;
}
return s;
}
```

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.

```
1  int main(void)
2  {
3     int a;
4     int b;
5
6     :
7     b = Watt(a);          // main calls Watt
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 s

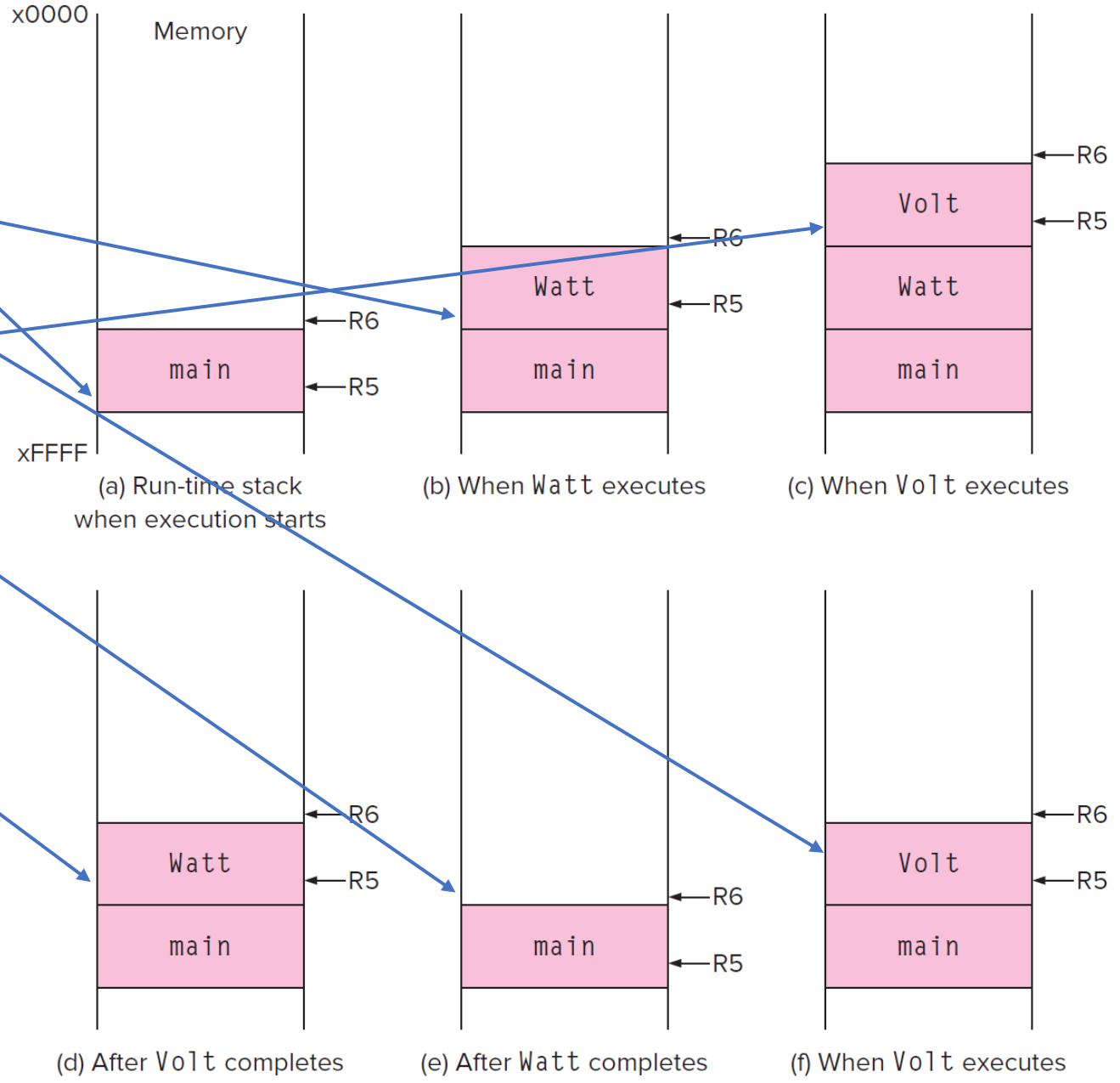


Figure 14.5 Several snapshots of the run-time stack while the program outlined in Figure 14.4 executes.

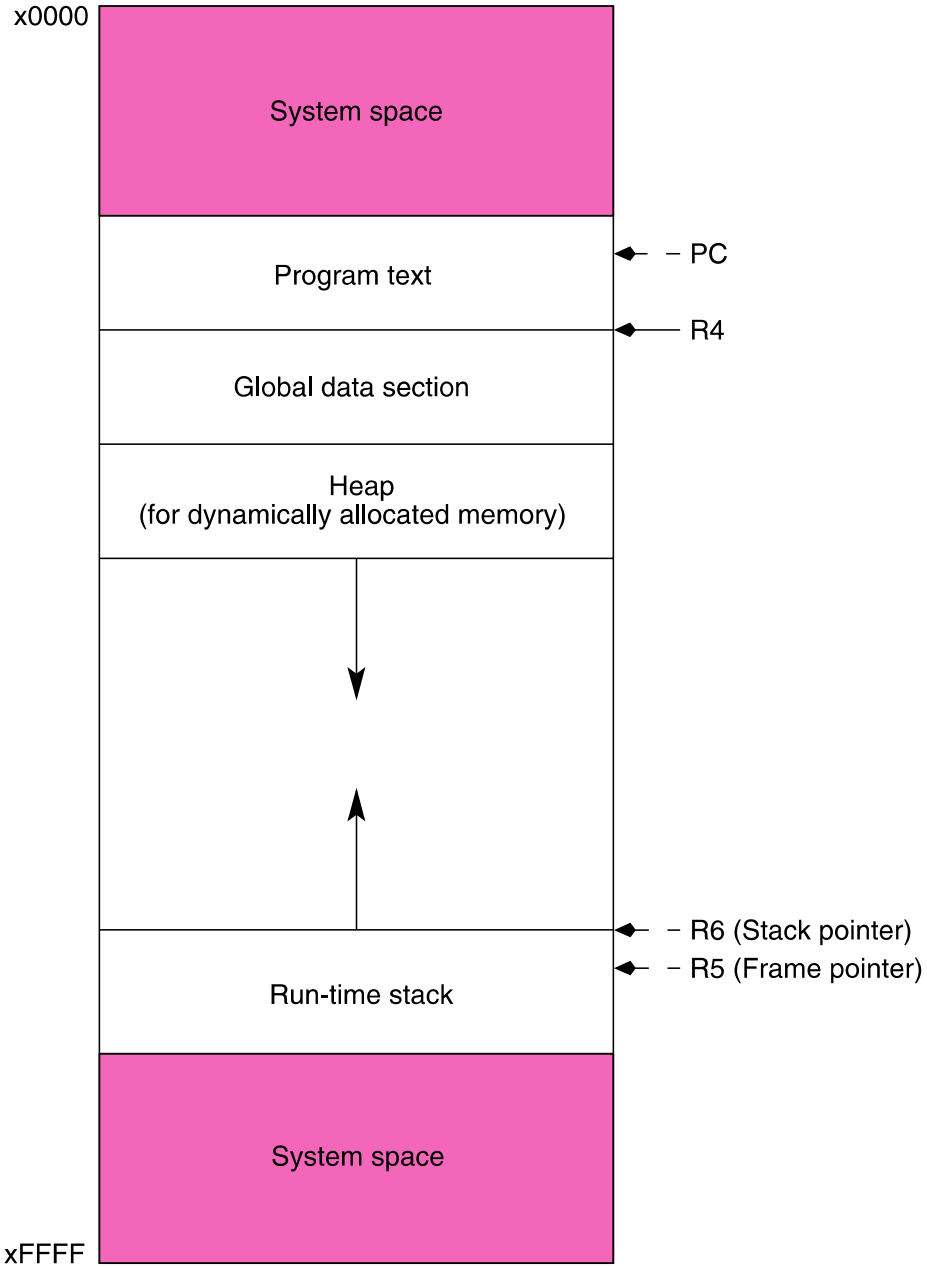
Activation Record

- stored in run-time stack
- *function call = push activation record*
- *function return = pop activation record*

Space for Variables

1. Global data section (global variables)
2. Run-time stack (local variables)

- R4 (global pointer) points the first global variable
- R5 (frame pointer) points the first local variable
- R6 (stack pointer) points the top of run-time stack

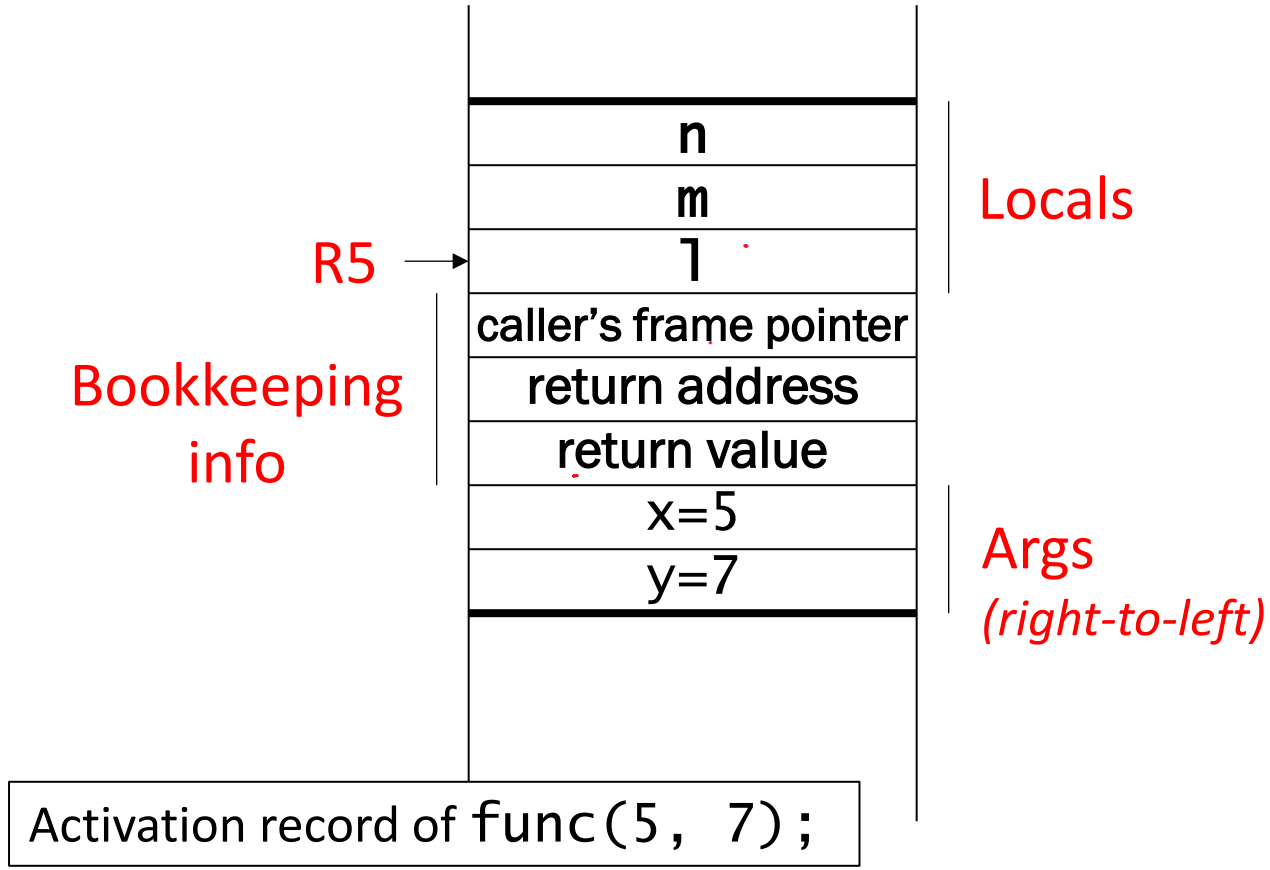


Building Activation Record

- Information about each function call, including

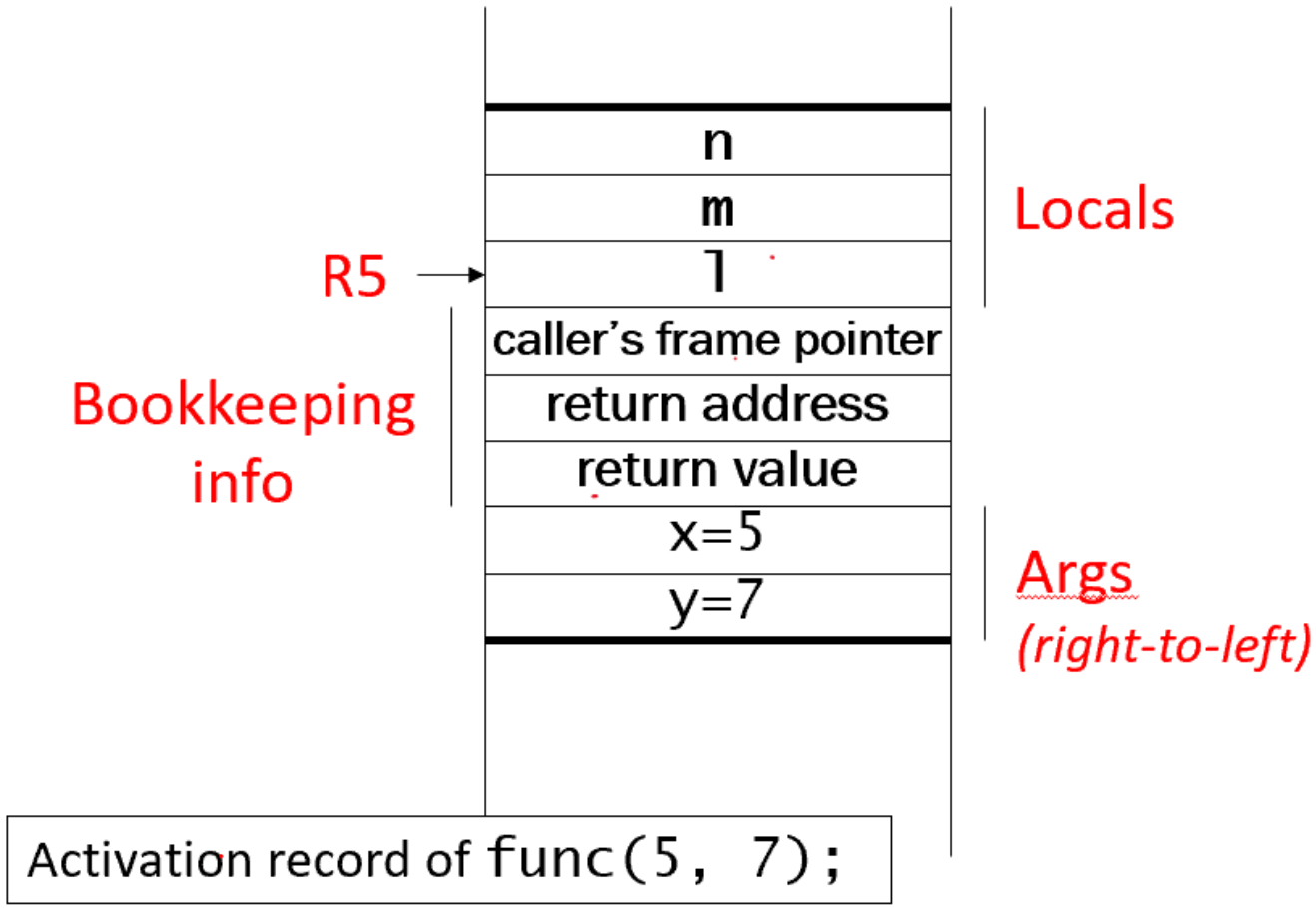
1. Arguments *2. Bookkeeping info* *3. Local variables*

```
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)

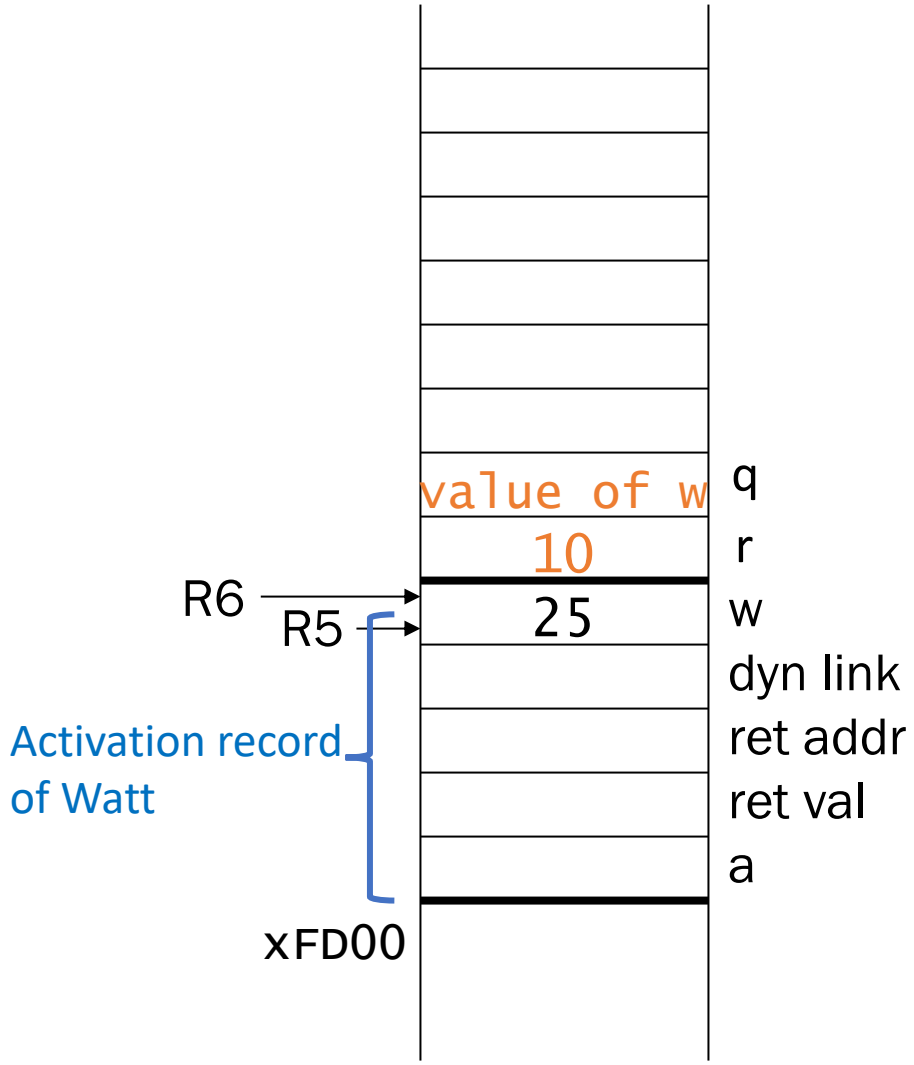
Example Function Call

```
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
ADD  R6, R6, #-1
STR  R0, R6, #0

; call subroutine
JSR  Volta
```

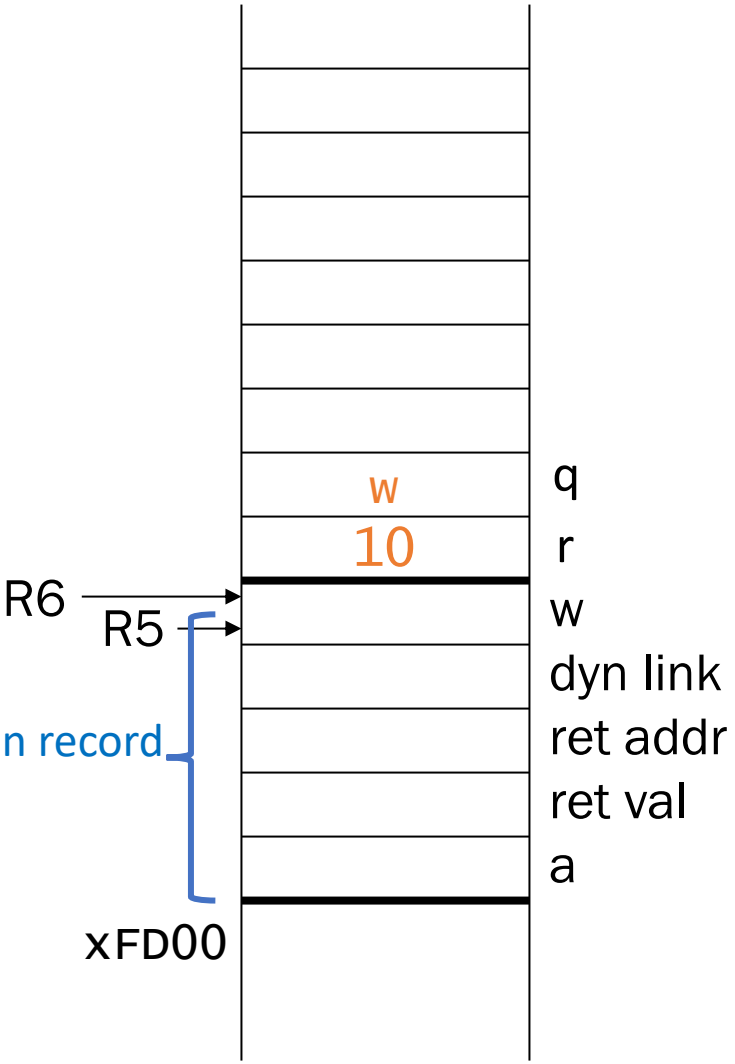
```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```



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
```

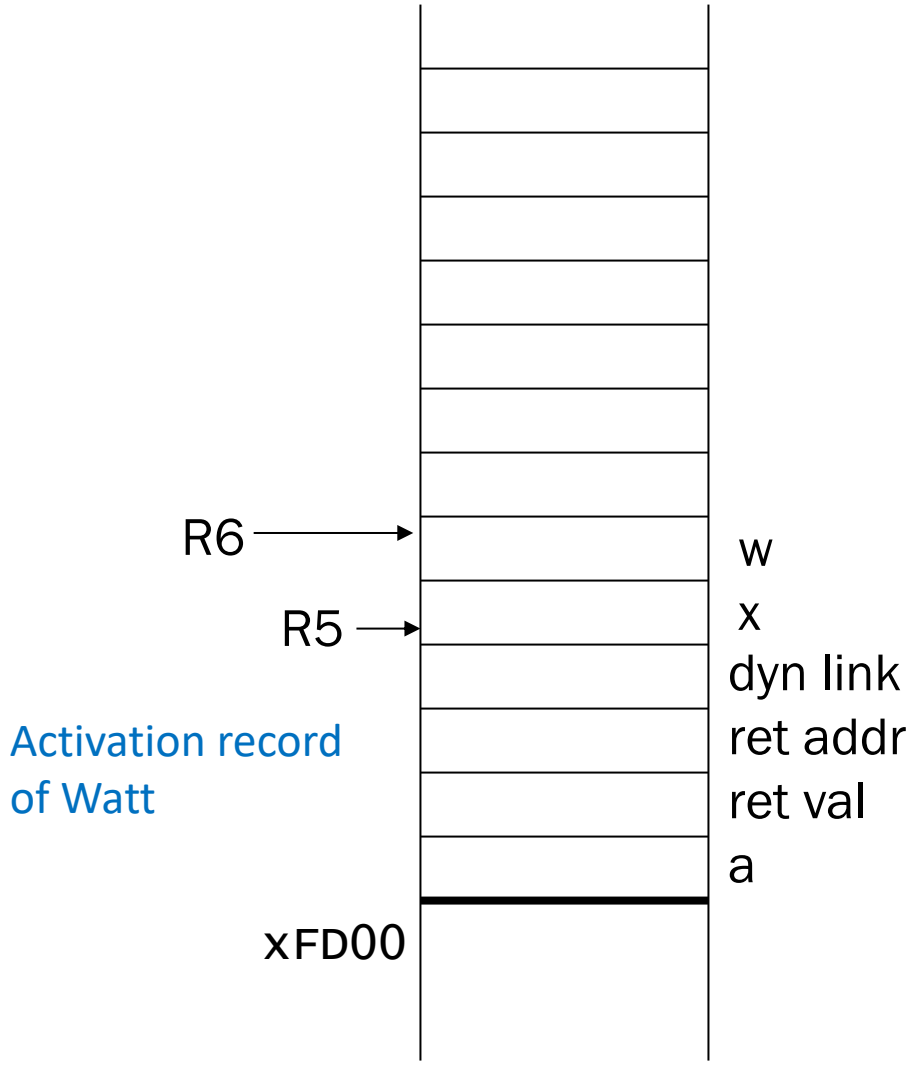
```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
int Watt(int a)
{
  int w;
  ...
  w = Volta(w,10);
  ...
  return w;
}
```



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
```

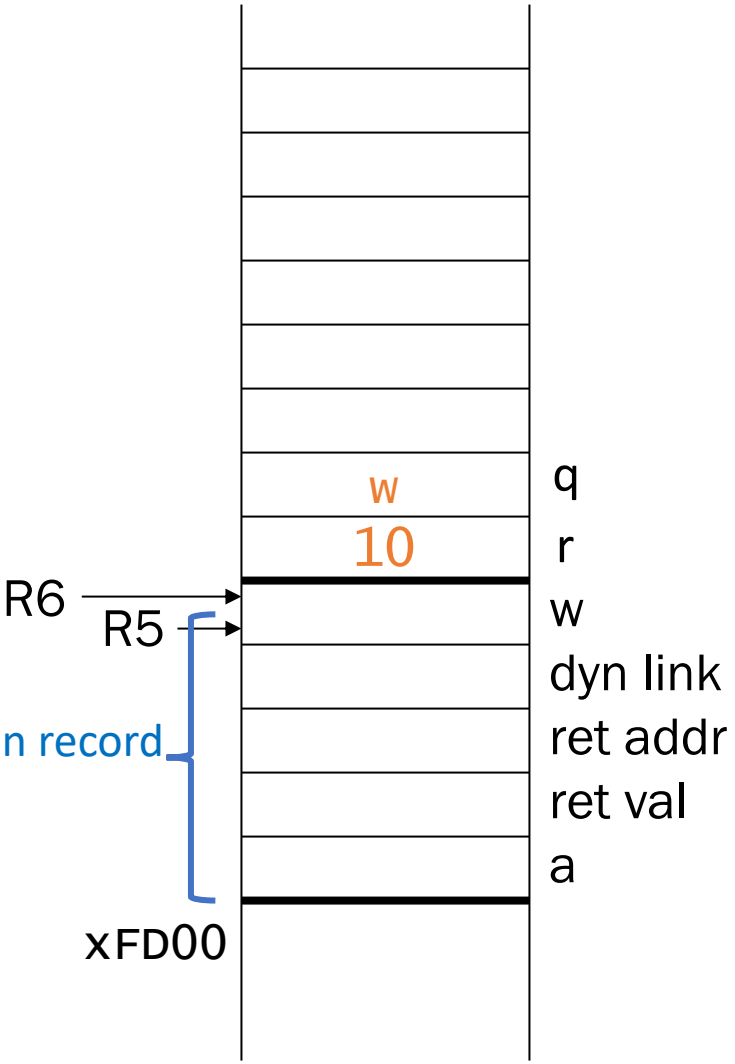
```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
int Watt(int a)
{
  int x, w;
  ...
  w = Volta(W,10);
  ...
  return w;
}
```



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;
}
```

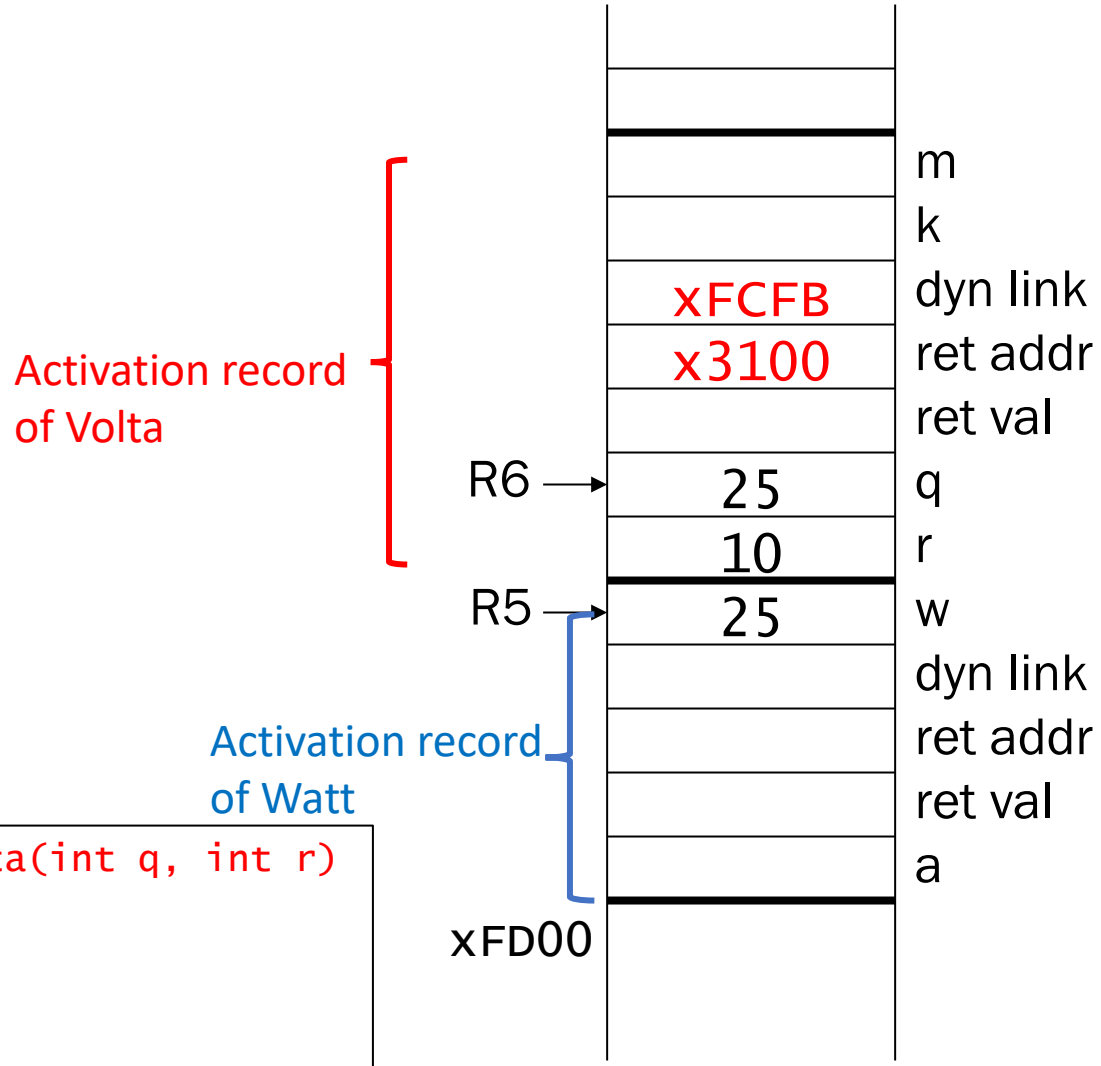


3. Callee setup (push bookkeeping info and local variables onto stack)
4. Execute function

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



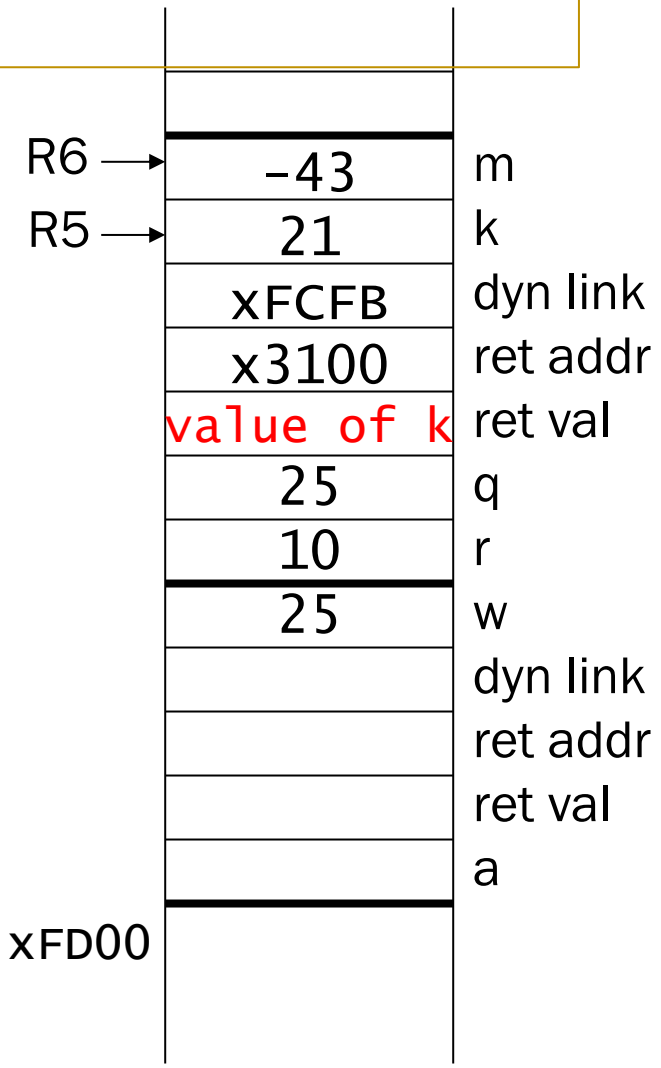

```
int Volta(int q, int r)
{
  int k;
  int m;
  ...
  return k;
}
```

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;
}
```

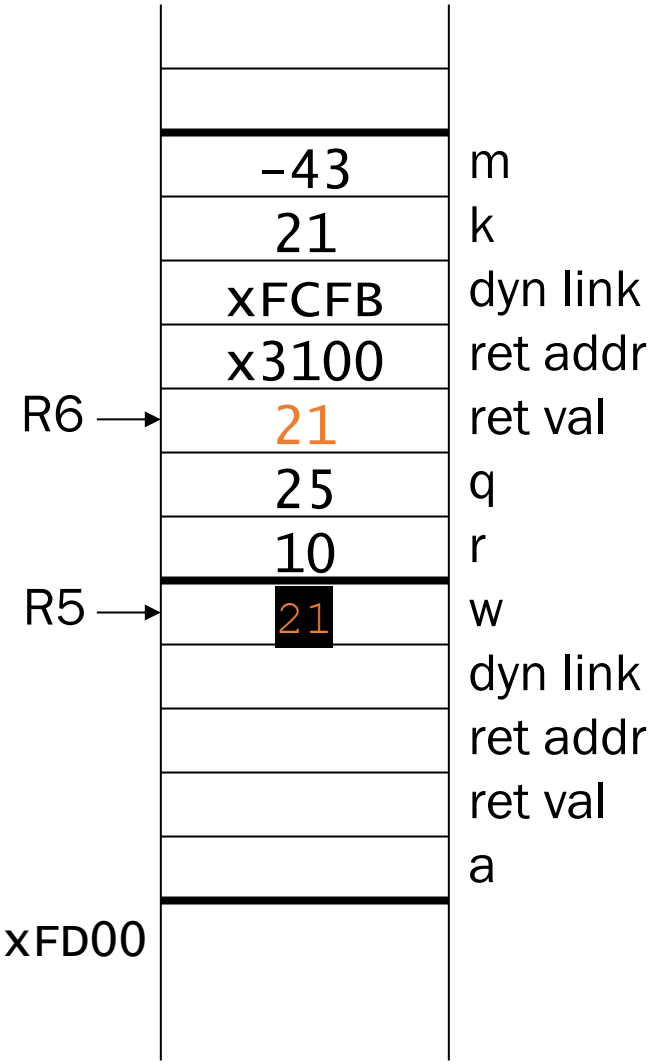
7. Caller tear-down (pop callee's return value and arguments from stack)

```
JSR  Volta
; load return value (top of
stack)
LDR  R0, R6, #0
; perform assignment
STR  R0, R5, #0
; pop return value
ADD  R6, R6, #1

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

depends on which variable

depends on # variables

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



Why Run-time Stack?

- Option1: Assign each activation record at fixed memory location

  Problem: What happen function A calls itself?

  Not memory efficient.

- Option2: Use Run-time Stack

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

  -> Space Allocated for activation records (i.e. run-time stack) becomes available for other function once the callee-function is done and the control is back to caller.