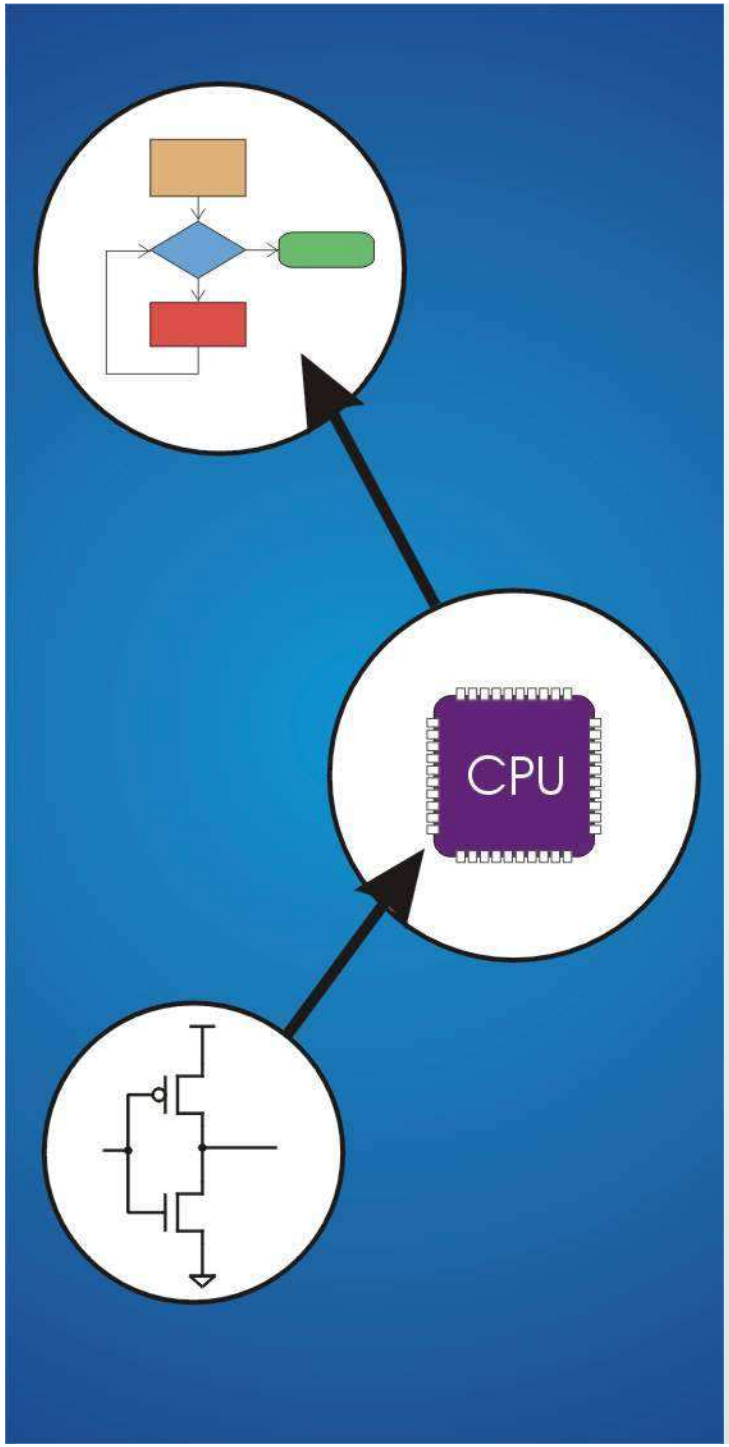


Back to Chapters 13,12

Original slides from Gregory Byrd, North Carolina State University

Modified by Y. Malaiya

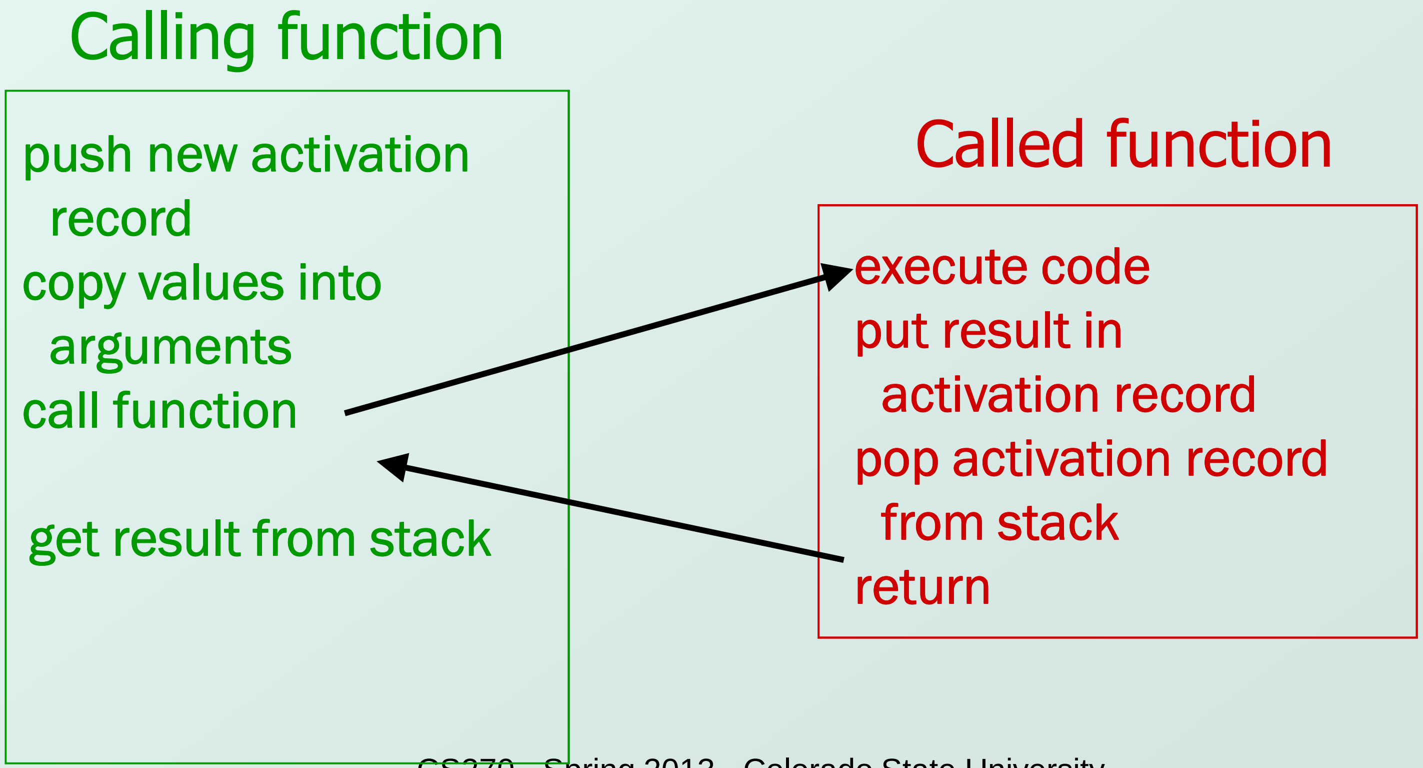
Colorado State University

The Binary Joke

- There are only 10 types of people in the world: those who understand binary, and those who don't.
- The Collegian The Strip Club editor (April 4, 2013) is apparently not among those who understand.

Implementing Functions: Overview

- Activation record (stack frame)
  - information about each function, including arguments and local variables
  - stored on run-time stack



Calling function

push new activation record
copy values into arguments
call function

get result from stack

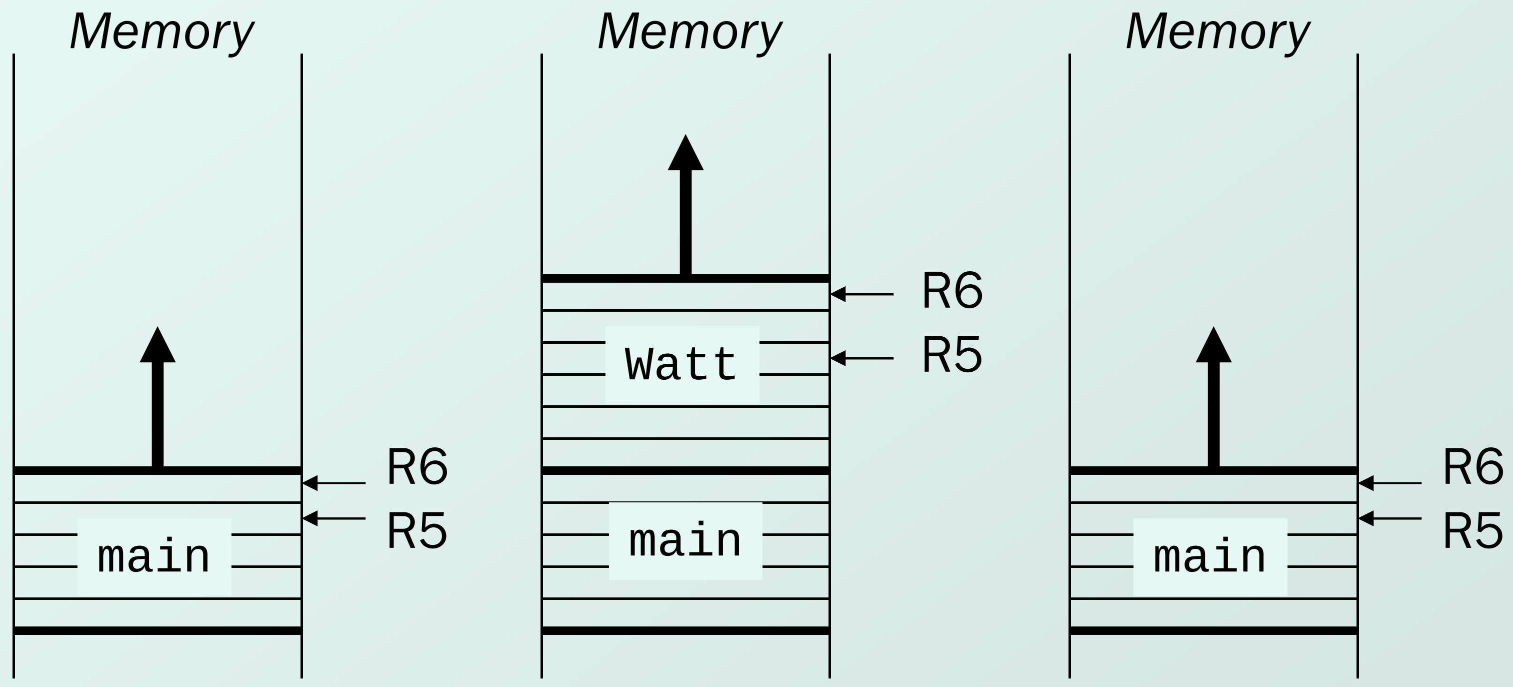
Called function

execute code
put result in activation record
pop activation record from stack
return

Run-Time Stack

- Recall that local variables are stored on the run-time stack in an *activation record*
- Stack Pointer (R6) is a pointer to the next free location in the stack, and is used to push and pop values on and off the stack.
- Frame pointer (R5) is a pointer to the beginning of a region of the activation record that stores local variables for the current function
- When a new function is called, its activation record is pushed on the stack; when it returns, its activation record is popped off of the stack.

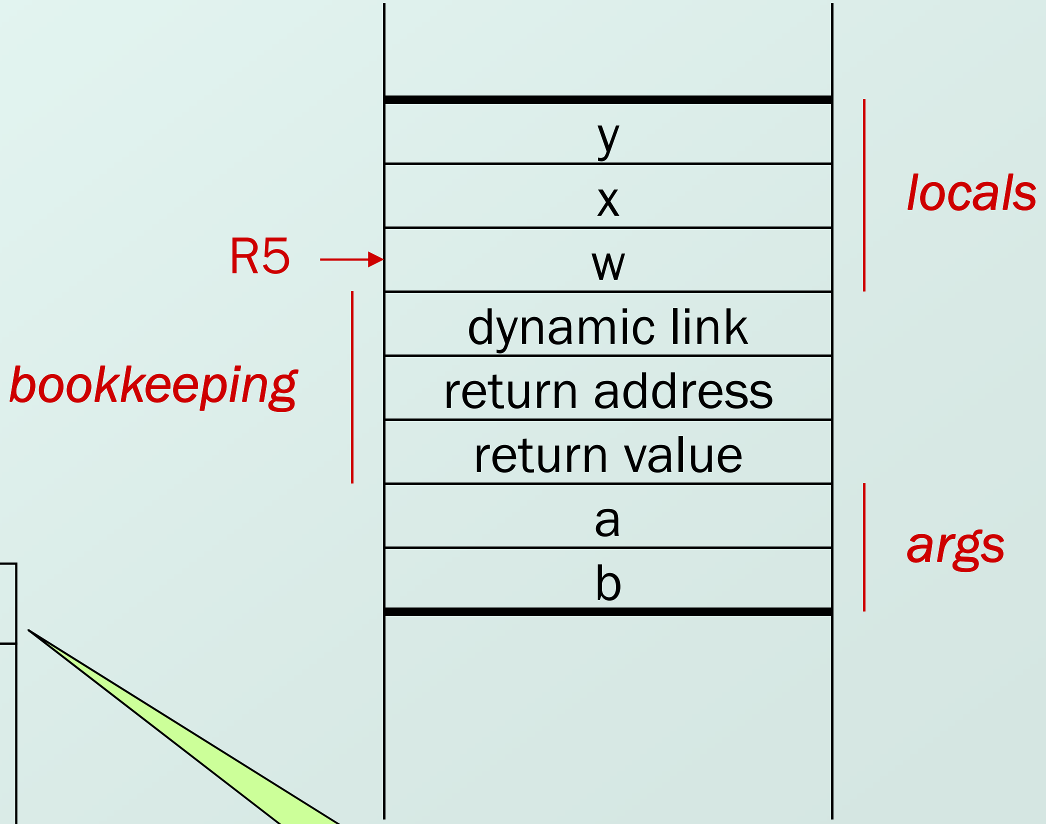
Run-Time Stack



Before call　*During call*　*After call*

Activation Record

```
int NoName(int a, int b)
{
   int w, x, y;
   .
   .
   .
   return y;
}
```



Name	Type	Offset	Scope
a	int	4	NoName
b	int	5	NoName
w	int	0	NoName
x	int	-1	NoName
y	int	-2	NoName

Symbol table

Activation Record Bookkeeping

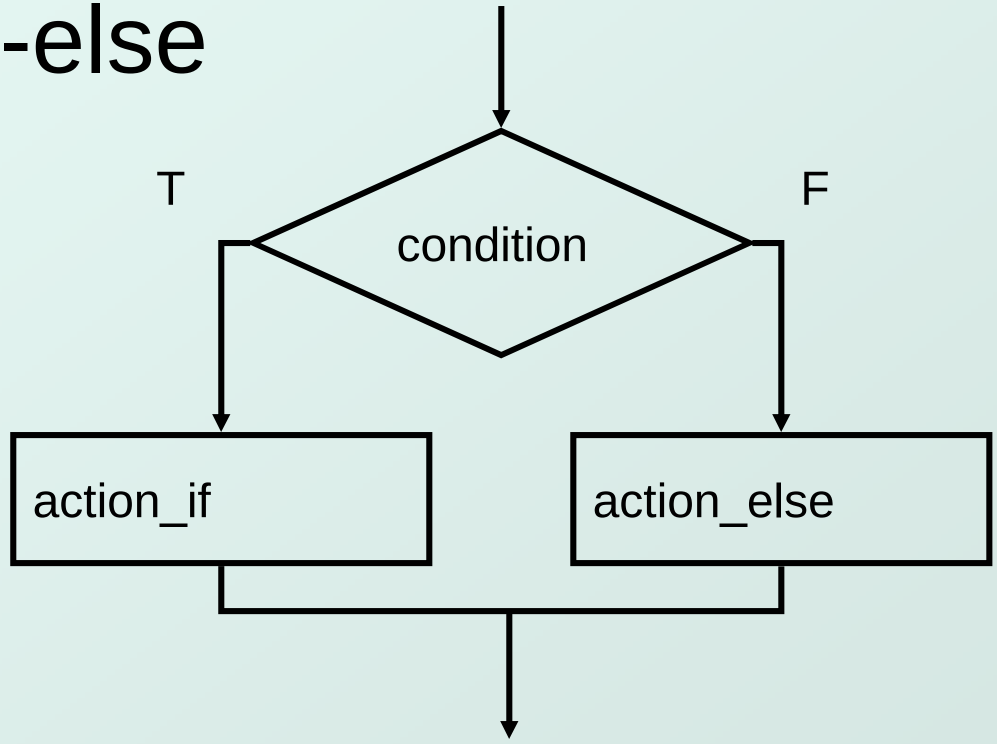
- Return value
  - space for value returned by function
  - allocated even if function does not return a value
- Return address
  - save pointer to next instruction in calling function
  - convenient location to store R7 in case another function (JSR) is called
- Dynamic link
  - caller’s frame pointer
  - used to pop this activation record from stack

Back to Chap 13

- Let’s see that again in LC-3 ..

If-else

-
  ```
  if (condition)
    action_if;
  else
    action_else;
  ```



Else allows choice between two mutually exclusive actions without re-testing condition.

Generating Code for If-Else

-
  ```
  if (x)
  {
     y++;
     z--;
  }
  ```

-
  ```
  else {
     y--;
     z++;
  ```

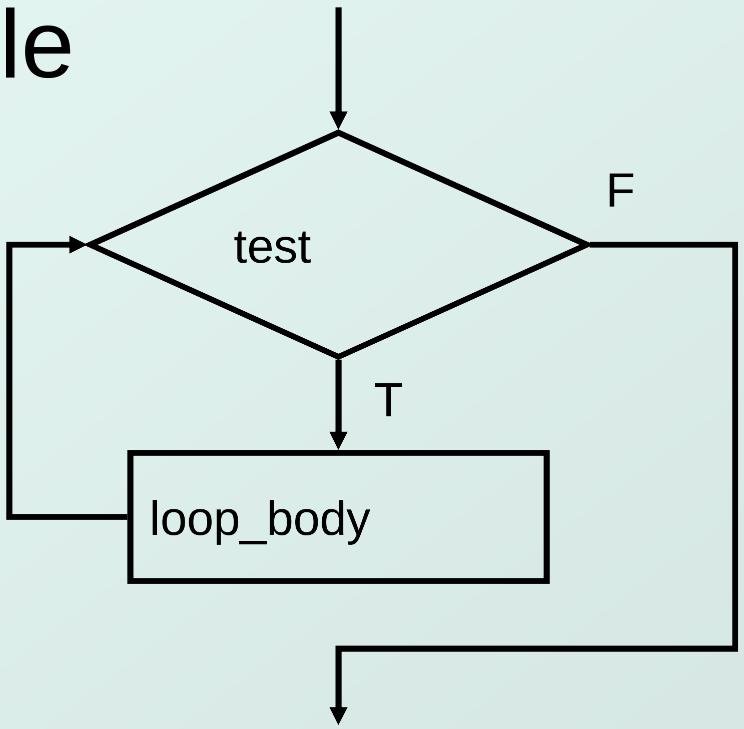
Symbol table

Name	Type	Offset	
x	int	0	main
y	int	-1	main
z	int	-2	main

```
         LDR   R0, R5, #0
        BRz   ELSE
        ; x is not zero
        LDR   R1, R5, #-1   ; incr y
        ADD   R1, R1, #1
        STR   R1, R5, #-1
        LDR   R1, R5, #-2   ; decr z
        ADD   R1, R1, #-1
        STR   R1, R5, #-2
        JMP   DONE    ; skip else code
        ; x is zero
ELSE    LDR   R1, R5, #-1   ; decr y
        ADD   R1, R1, #-1
        STR   R1, R5, #-1
        LDR   R1, R5, #-2   ; incr z
        ADD   R1, R1, #1
        STR   R1, R5, #-2
DONE    ...   ; next statement
```

While

-
  ```
  while (test)
    loop_body;
  ```



Executes loop body as long as test evaluates to TRUE (non-zero).

*Note: Test is evaluated **before** executing loop body.*

Generating Code for While

```
x = 0;
while (x < 10) {
  printf(“%d ”, x);
  x = x + 1;
}
```

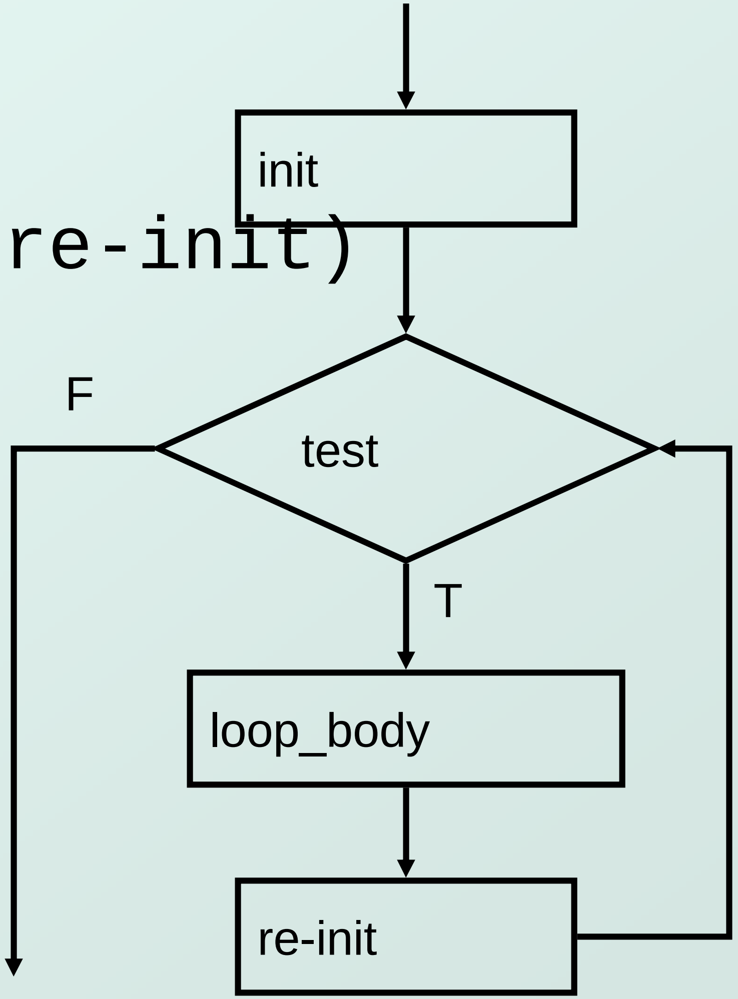
Symbol table			
Name	Type	Offset	
x	int	0	main
y	int	-1	main
z	int	-2	main

```
        AND  R0, R0, #0
        STR  R0, R5, #0 ; x = 0
        ; test
LOOP    LDR  R0, R5, #0 ; load x
        ADD  R0, R0, #-10
        BRzp DONE
        ; loop body
        LDR  R0, R5, #0 ; load x
        ...
        <printf>
        ...
        ADD  R0, R0, #1 ; incr x
        STR  R0, R5, #0
        JMP  LOOP     ; test again

DONE    ; next statement
```

For

- `for (init; end-test; re-init)`
  `statement`



Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code includedin loop statement.

Note: Test is evaluated ***<u>before</u>*** *executing loop body.*

Generating Code for For

```
for (i = 0; i < 10; i++)
  printf("%d ", i);
```

This is the same
as the while example!

Symbol table			
Name	Type	Offset	
i	int	0	main
y	int	-1	main
z	int	-2	main

```
          ; init
          AND  R0, R0, #0
          STR  R0, R5, #0  ; i = 0
          ; test
LOOP      LDR  R0, R5, #0  ; load i
          ADD  R0, R0, #-10
          BRzp DONE
          ; loop body
          LDR  R0, R5, #0  ; load i
          ...
          <printf>
          ...
          ; re-init
          ADD  R0, R0, #1  ; incr i
          STR  R0, R5, #0
          JMP  LOOP        ; test again

DONE      ; next statement
```

Back to Chap 12

- Let’s see that again in LC-3 ..

Symbol Table

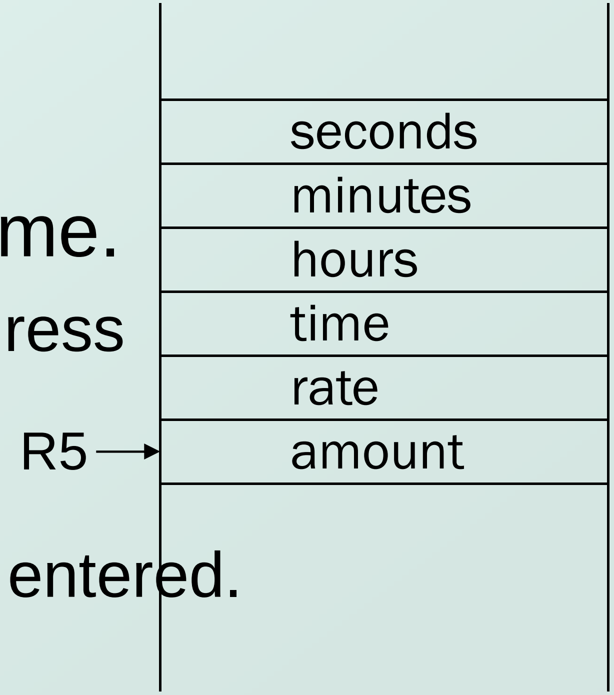
- Like assembler, compiler needs to know information associated with identifiers
  - in assembler, all identifiers were labels and information is address

- Compiler keeps more information
- Name (identifier)
- Type
- Location in memory
- Scope

Name	Type	Offset	Scope
amount	int	0	main
hours	int	-3	main
minutes	int	-4	main
rate	int	-1	main
seconds	int	-5	main
time	int	-2	main

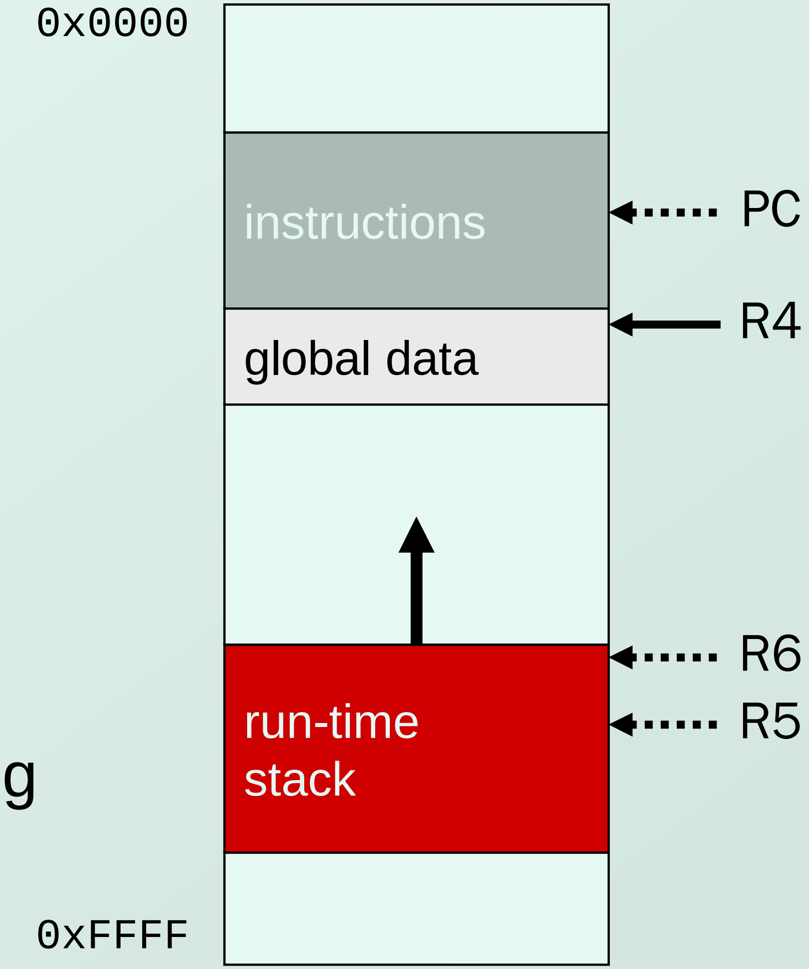
Local Variable Storage

- Local variables are stored in an *activation record*, also known as a *stack frame*.
- Symbol table "offset" gives the distance from the base of the frame.
  - R5 is the frame pointer – holds address of the base of the current frame.
  - A new frame is pushed on the run-time stack each time a block is entered.
  - Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.



Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
  - R4 points to beginning
- Run-time stack
  - Used for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - New frame for each block (goes away when block exited)
- Offset = distance from beginning of storage area
  - Global: `LDR R1, R4, #4`
  - Local: `LDR R2, R5, #-3`



Variables and Memory Locations

- In our examples,
  a variable is always stored in memory.

- When assigning to a variable,
  must <u>store</u> to memory location.

- A real compiler would perform code optimizations
  that try to keep variables allocated in registers.
- Why?

Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main()
{
  int inLocal;   /* local to main */
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n",
         outLocalA, outLocalB);
}
```

Example: Symbol Table

Name	Type	Offset	Scope
`inGlobal`	`int`	`0`	`global`
`inLocal`	`int`	`0`	`main`
`outLocalA`	`int`	`-1`	`main`
`outLocalB`	`int`	`-2`	`main`

Example: Code Generation

- ; main
- ; initialize variables
-
  ```
      AND R0, R0, #0
      ADD R0, R0, #5   ; inLocal = 5
      STR R0, R5, #0   ; (offset = 0)

      AND R0, R0, #0
      ADD R0, R0, #3   ; inGlobal = 3
      STR R0, R4, #0   ; (offset = 0)
  ```

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

Examp

- ; first statement:
- ; outLocalA = inLocal++ & ~inGlobal;
-
```
        LDR R0, R5, #0   ; get inLocal
        ADD R1, R0, #1   ; increment
        STR R1, R5, #0   ; store

        LDR R1, R4, #0   ; get inGlobal
        NOT R1, R1       ; ~inGlobal
        AND R2, R0, R1   ; inLocal & ~inGlobal
        STR R2, R5, #-1  ; store in outLocalA
                         ; (offset = -1)
```

Example (continued)

-
  ```
  ; next statement:
  ```

-
  ```
  ; outLocalB = (inLocal + inGlobal)
  ;             - (inLocal - inGlobal);
  ```

-
  ```
        LDR R0, R5, #0  ; inLocal
        LDR R1, R4, #0  ; inGlobal
        ADD R0, R0, R1  ; R0 is sum
        LDR R2, R5, #0  ; inLocal
        LDR R3, R5, #0  ; inGlobal
        NOT R3, R3
        ADD R3, R3, #1
        ADD R2, R2, R3  ; R2 is difference
        NOT R2, R2      ; negate
        ADD R2, R2, #1
        ADD R0, R0, R2  ; R0 = R0 - R2
        STR R0, R5, #-2 ; outLocalB (offset = -2)
  ```

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main