Chapter 17

Recursion

Implementing a Function Call: Overview

Calling function

- allocate memory for activation record (push)
- copy arguments into stack arguments
- call function
- get result from stack (pop)
- deallocate memory for activation record (pop)

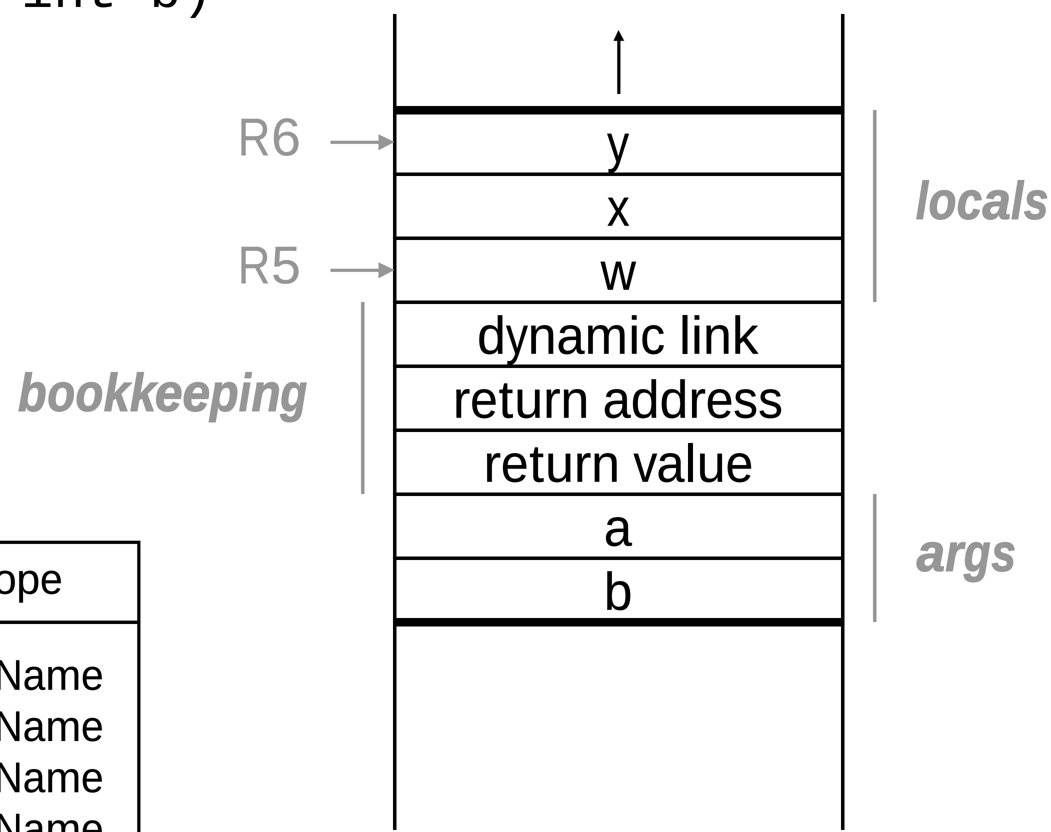
Called function

- allocate space for return value [bookkeeping] (push)
- store mandatory callee save registers [bookkeeping] (push)
- set frame pointer
- allocate local variables (push)
- execute code
- put result in return value space
- deallocate local variables (pop)
- load callee save registers (pop)
- return

Activation Record

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

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



Summary of LC-3 Function Call Implementation

1. Caller pushes arguments (last to first).
2. Caller invokes subroutine (JSR).
3. Callee allocates return value, pushes R7 and R5.
4. Callee allocates space for local variables (first to last).
5. Callee executes function code.
6. Callee stores result into return value slot.
7. Callee pops local vars, pops R5, pops R7.
8. Callee returns (RET/JMP R7).
9. Caller loads return value and pops arguments.
10. Caller resumes computation…

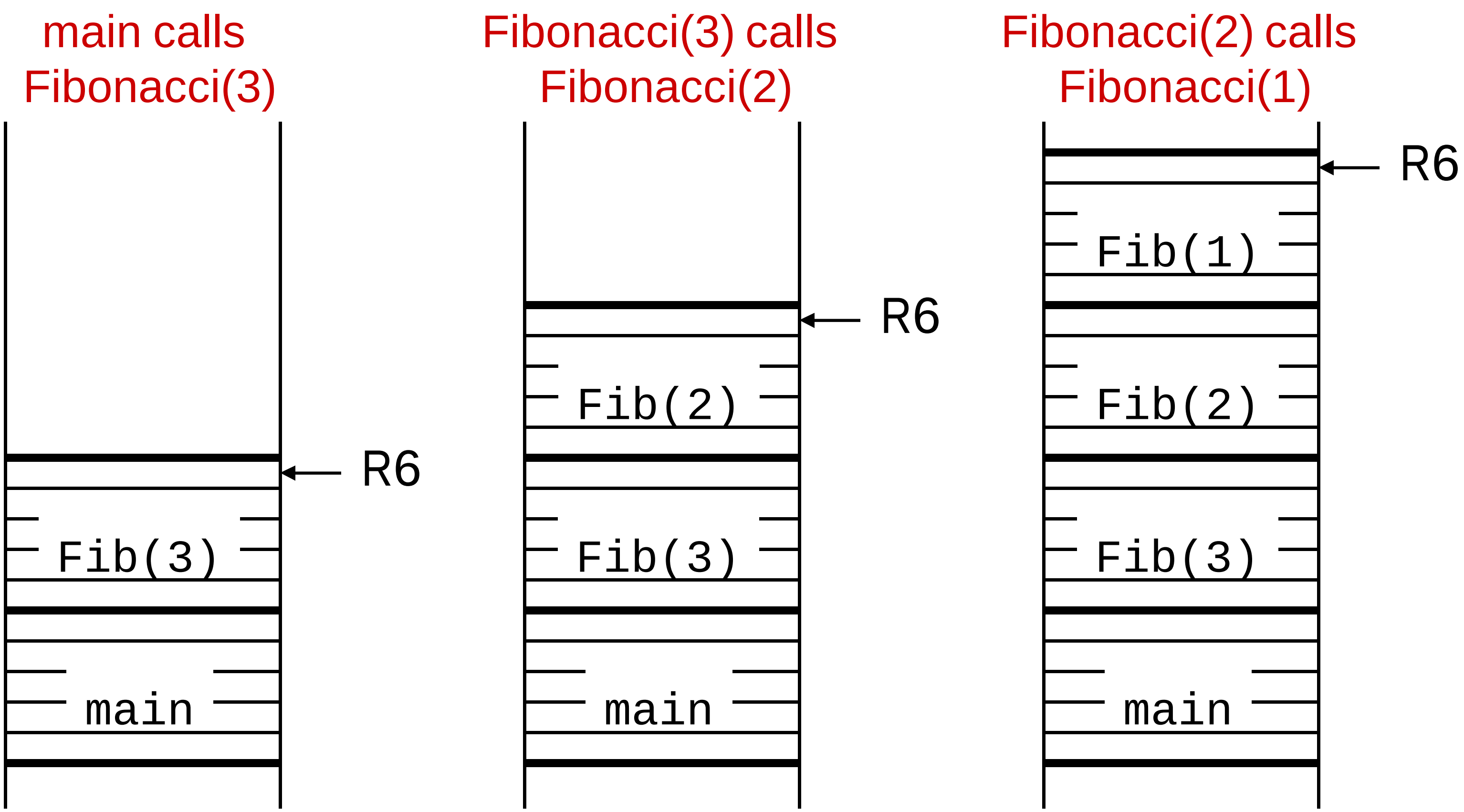
What is Recursion?

- A recursive function is one that solves its task by calling itself on smaller pieces of data.
  - Similar to recurrence function in mathematics.
  - Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.
- Standard example: Fibonacci numbers
  - The n-th Fibonacci number is the sum of the previous two Fibonacci numbers.
  - $F(n) = F(n-1) + F(n-2)$ where $F(1) = F(0) = 1$

```
int Fibonacci(int n){
    if ((n == 0) || (n == 1))
            return 1;
    else
            return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Activation Records

- Whenever Fibonacci is invoked, a new activation record is pushed onto the stack.

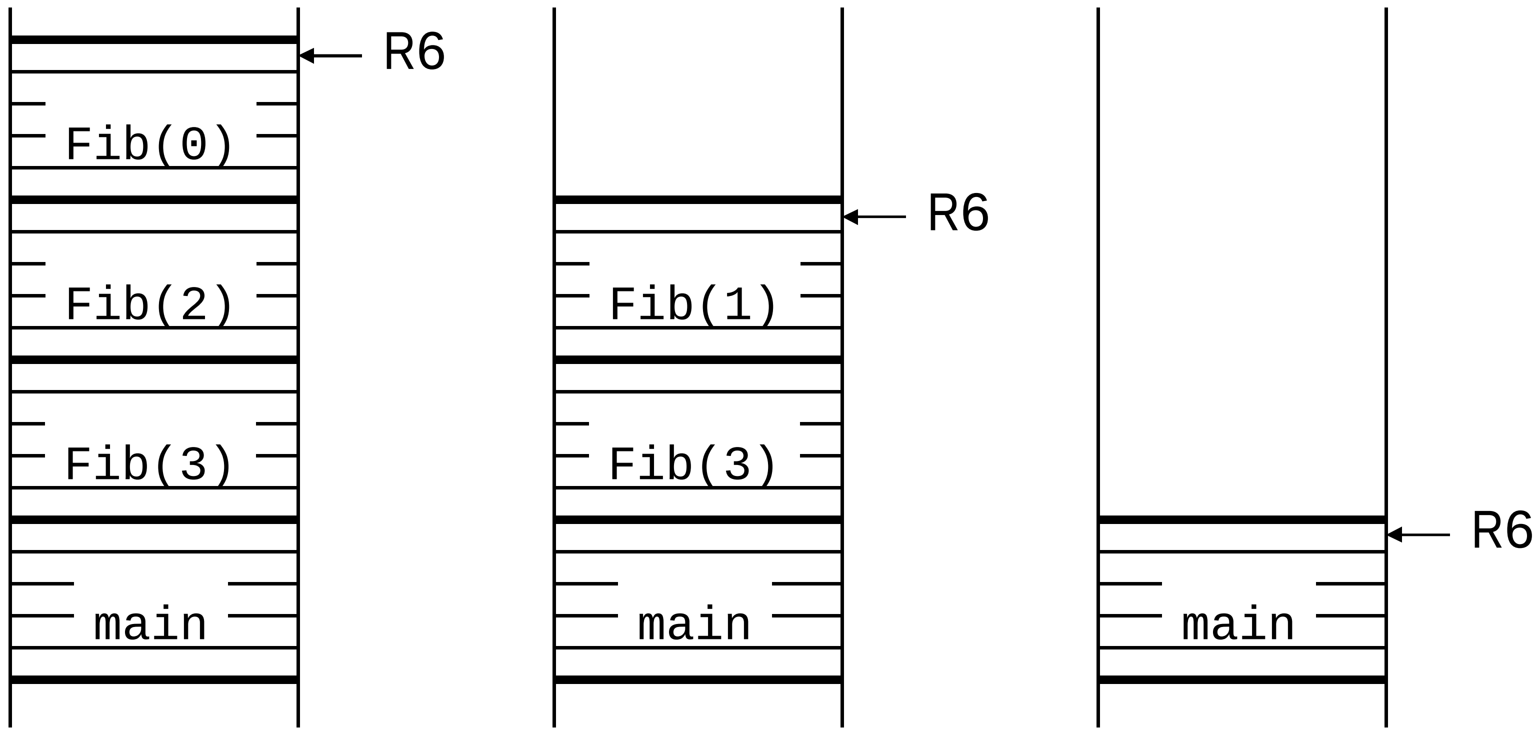


Activation Records (cont.)

Fibonacci(1) returns, Fibonacci(2) calls Fibonacci(0)

Fibonacci(2) returns, Fibonacci(3) calls Fibonacci(1)

Fibonacci(3) returns



Tracing the Function Calls

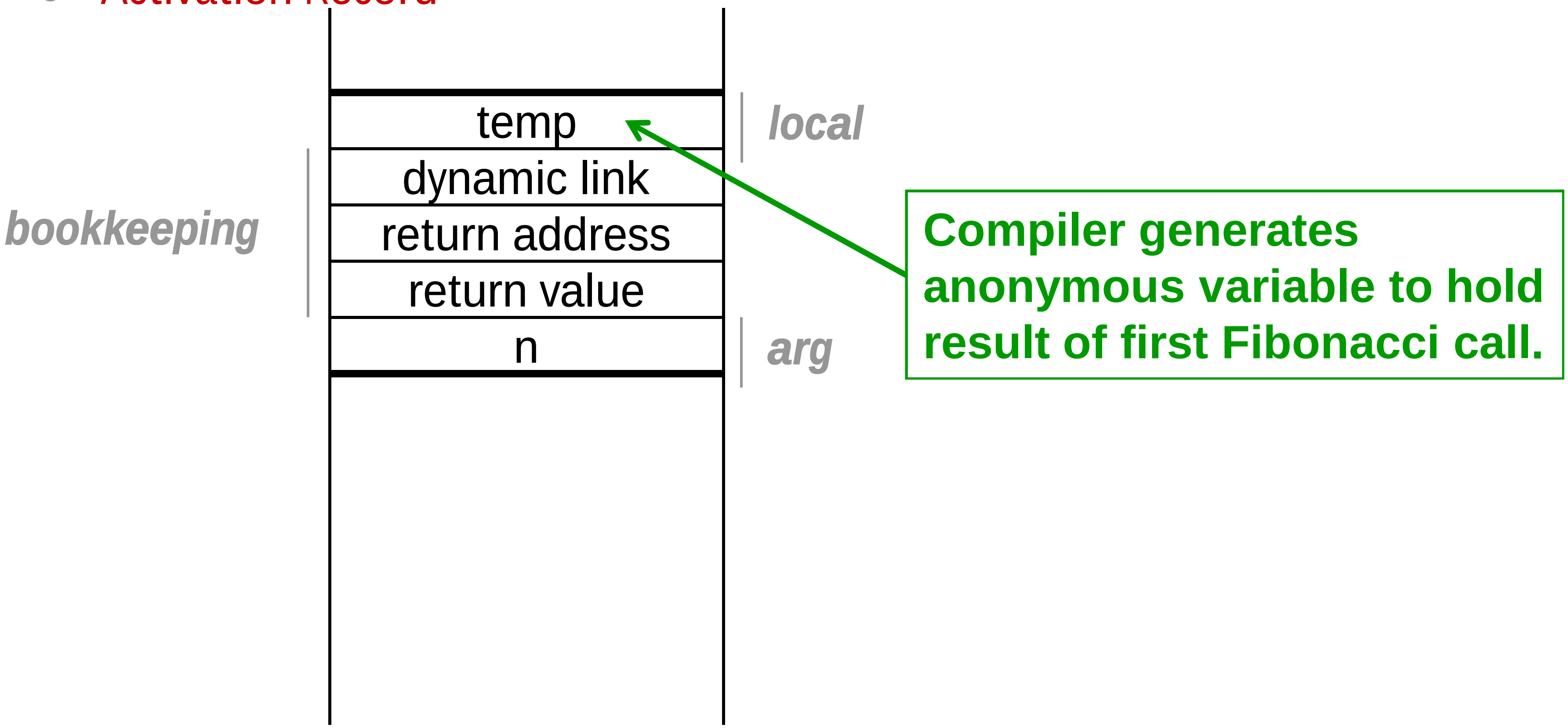
- If we are debugging this program,
  we might want to trace all the calls of Fibonacci.
  - Note: A trace will also contain the arguments passed into the function.

- For Fibonacci(3), a trace looks like:
- ```
  Fibonacci(3)
   Fibonacci(2)
    Fibonacci(1)
    Fibonacci(0)
   Fibonacci(1)
  ```

- What would trace of Fibonacci(4) look like?
  ```

Fibonacci: LC-3 Code

- Activation Record



In Summary: The Stack

- Since our program usually starts at a low memory address and grows upward, we start the stack at a high memory address and work downward.
- Purposes
  - Temporary storage of variables
  - Temporary storage of program addresses
  - Communication with *subroutines*
    - Push variables on stack
    - Jump to subroutine
    - Clean stack
    - Return

Parameter passing on the stack

- If we use registers to pass our parameters:
  - Limit of 8 parameters to/from any subroutine.
  - We use up registers so they are not available to our program.
- So, instead we push the parameters onto the stack.
  - Parameters are passed on the stack
  - Return values can be provided in registers (such as R0) or on the stack.
  - Generally, only R6 should be changed by a subroutine.
    - Other registers that are changed should must be callee saved/restored.
    - Subroutines should be *transparent*
- Both the subroutine and the main program must know how many parameters are being passed!
  - In C we would use a prototype: `int power (int number, int exponent);`
- In assembly, you must take care of this yourself.
- After you return from a subroutine, you must also *clear the stack*.
  - Clean up your mess!

Characteristics of good subroutines

- **Readability** – well documented.
- **Generality** – can be easily reused elsewhere
  - Passing arguments on the stack does this.
- **Transparency** – you have to leave the registers like you found them, except R6.
  - Registers must be callee saved.
- **Re-entrant** – subroutine can call itself if necessary
  - Store all information relevant to specific execution to non-fixed memory locations
    - The stack!
  - This includes temporary callee storage of register values!
- **Secure** – No unexpected side effects on the stack / memory.

Know how to…

- Push parameters onto the stack
- Access parameters on the stack using base + offset addressing mode
- Draw the stack to keep track of subroutine execution
  - Parameters
  - Return address
- Clean the stack after a subroutine call

Practice problems

- 14.2, 14.4, 14.9, 14.10, 14.15 (good!)
- The convention in LC-3 assembly is that all registers are callee-saved except for R5 (the frame pointer) R6 (the stack pointer) and R7 (the return link).
  - Why is R5 not callee-saved?
  - Why is R6 not callee-saved?
  - Why is R7 not callee-saved?
- Is it true that any problem that can be solved recursively can be solved iteratively using a stack data structure? Why or why not?

```
main() {
  int i, j, k;

  i = 5;
  j = 3;
    …
  k = sub1(i, j);
    …
}


int sub1(a, b) {
  int x, y;
    …
  x = a;
  y = sub2 (x, 3);
  return y;
}


int sub2(var1, var2) {
  int temp;
    …
  temp = var1 – var2;
  return temp;
}
```