Introduction to Computing Systems
from bits & gates to C & beyond

Introduction to Part II

HLL Structures

- *Variables*
- *Activation records*

Higher Level Languages

- **HLLs allow symbolic naming of variables**
  - After mastering Assemly, you can now appreciate what a luxury this is!
- **HLLs provide expressiveness**
  - Compare an if statement to a conditional Branch instruction!
- **HLLs abstract from the underlying hardware**
  - While each microprocessor has its own ISA, HLL code is usually portable between many different platforms
- **etc., etc.**

Translation

- **HLL code has to be translated into the ISA of each machine it runs on:**
  - **Interpretation:**
    - ***The HLL code is simply text inpuy for a program called an interpreter, which is a virtual machine that translates the HLL code a piece at a time into Machine Language, executes it on the spot, then goes back for another piece.***
    - ***Both the HLL source code and the interpreter are required every time the program is run.***
  - **Compilation**
    - ***The HLL code is translated into an entire ML program (an executable image) by a program called a compiler.***
    - ***Only the executable image is needed to run the program.***
    - ***Most application software is sold in this form.***

Variables

- **Symbolic names for variables**
  - **The compiler creates a symbol table which holds information about each variable:**
    - ***Identifier (name), data type, memory location (as an offset from a "frame pointer"), scope, etc.***
  - **Suppose a variable "seconds" has been assigned an offset of -5 from the frame pointer, which is maintained in R5. We can now access *seconds* by the Base+Offset addressing mode:**
    - ***LDR R0, R5, # -5***
    - ***i.e. each reference to the variable seconds is translated by the compiler to (R5) - 5***
    - ***We have finally found a use for the offset in LDR!***

Activation Records

- **A Record**
  - A collection of contiguous memory locations treated as one entity
  - The *struct* construct in C/C++ describes a record
- **Activation Record**
  - Is allocated for each *function invocation* in memory
  - The area of memory where activation records are allocated is called the *stack memory* or *run-time stack*
  - In LC-3, R6 is the *stack pointer*: it points to the top of the stack (TOS)
  - R5 is the *frame pointer*: it points to the function's first local variable
  - Each element on the run-time stack is an activation record

Function calls

- **When a function call is made:**
  - **The caller saves space on the stack for the return value**
  - **The callee pushes a copy of the return address (in R7) onto the stack**
  - **The callee pushes a copy of the dynamic link (the caller’s frame pointer, in R5) onto the stack**
  - **The callee allocates enough space on the stack for its local variables, and adjusts R5 to point to the first of them, and R6 to point to the top of the stack.**

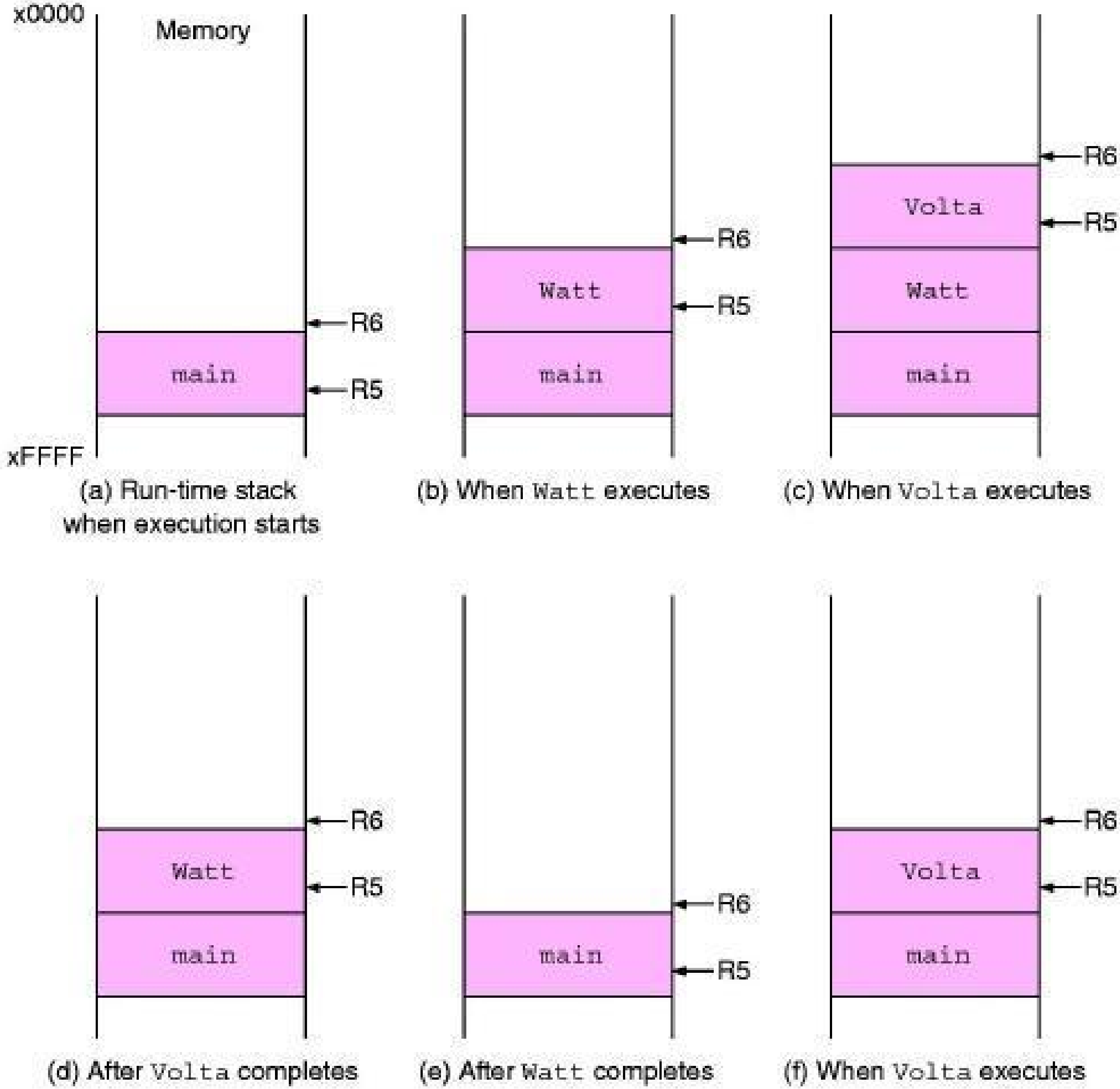
Sample Function calls

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

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

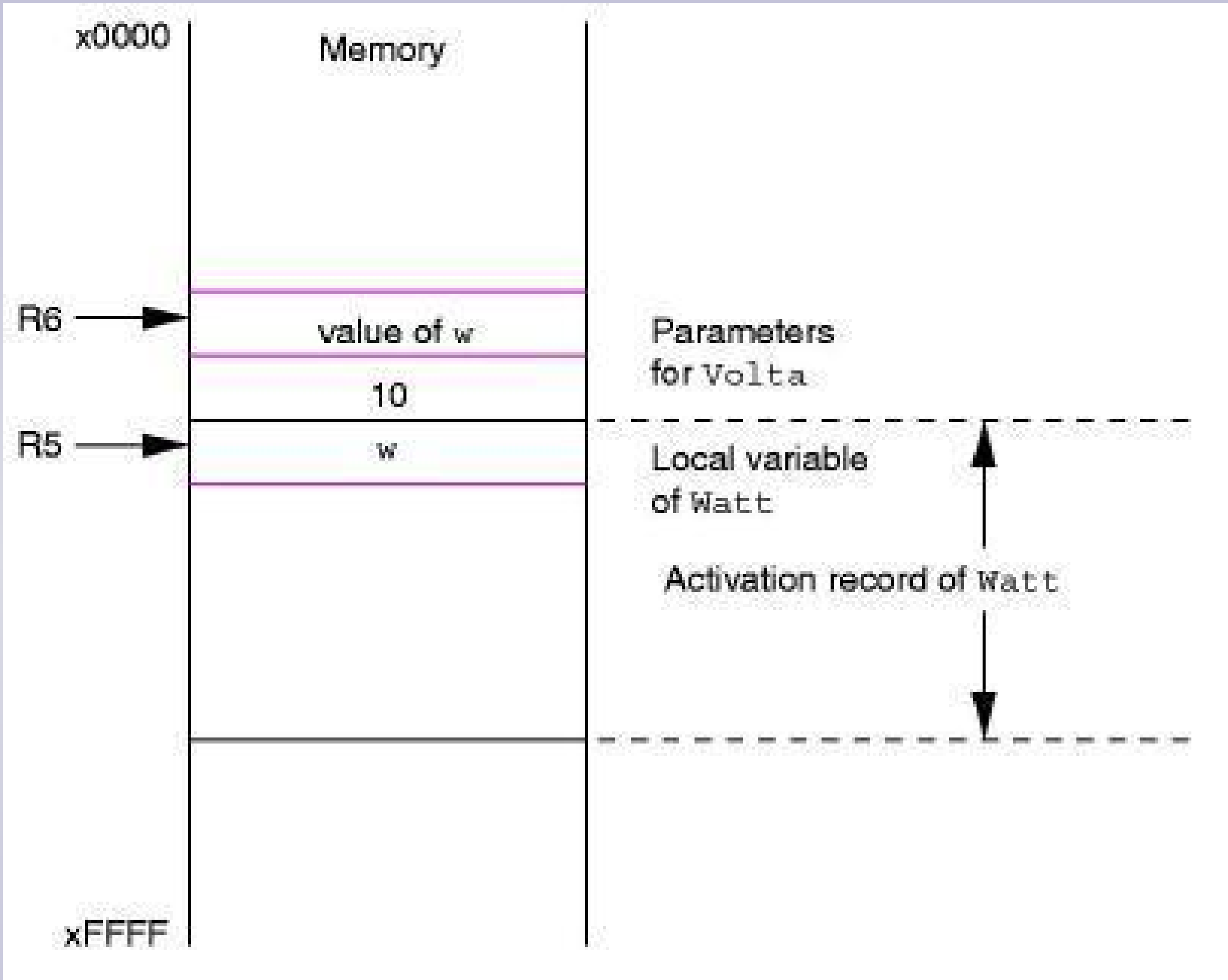
```
int Volta( int q, int r)
{
        int k, m;
        …
        return k;
}
```

Main calls Watt, which calls Volta . . .



Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Watt prepares to call Volta



LC-3 code for passing arguments

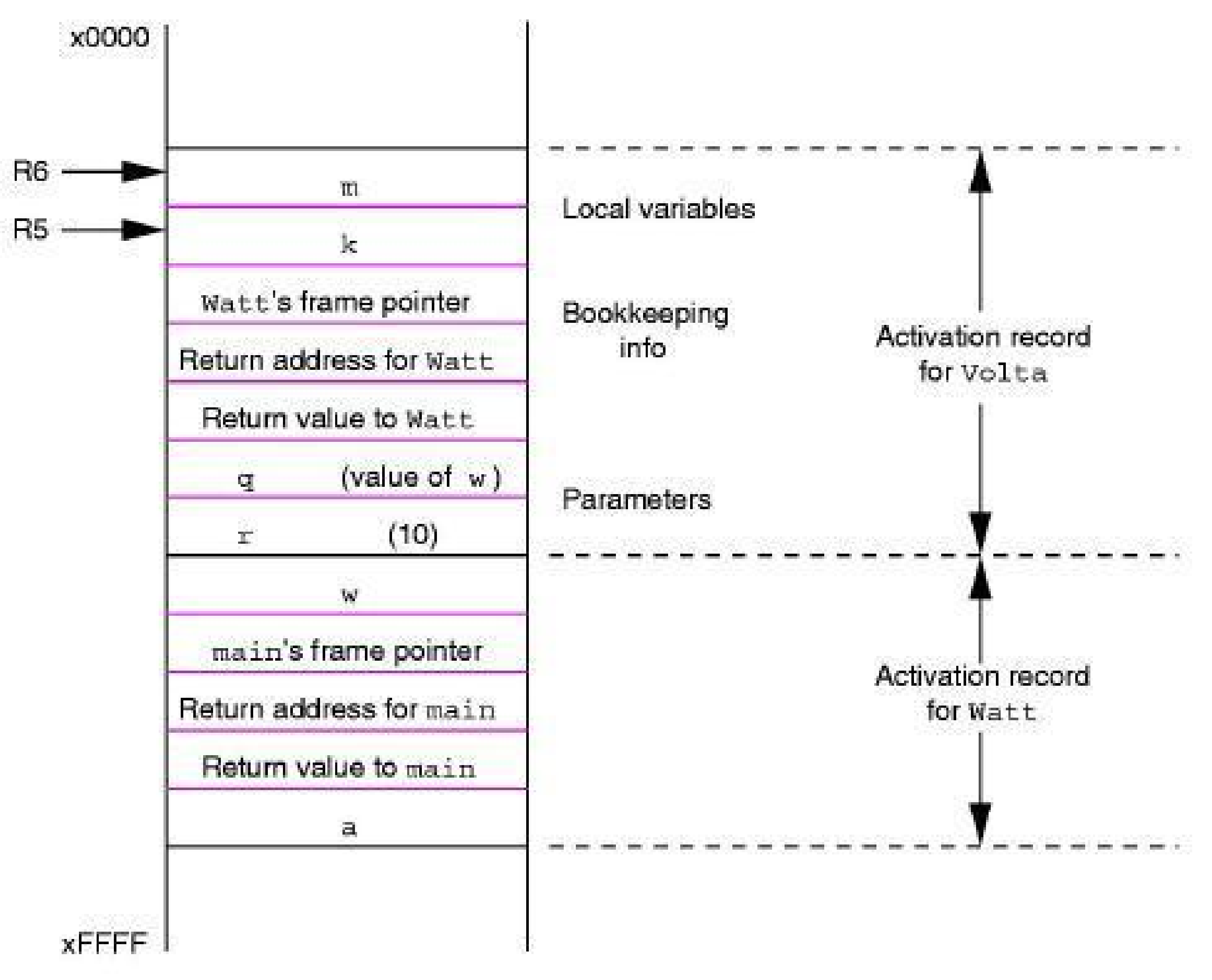
Watt prepares to call Volta:

```
AND   R0, R0, #0
ADD   R0, R0, #10   ; R0 <= 10
ADD   R6, R6, #-1
STR   R0, R6, #0    ; Push 10 onto stack

LDR   R0, R5, #0    ; Get value of w
ADD   R6, R6, #-1
STR   R0, R6, #0    ; Push value of w onto stack

JSRVolta
```

Volta is ready to run



LC-3 code for starting the function

Volta:

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

ADD    R6, R6, #-1
STR    R7, R6, #0    ; Push R7 (return address)

ADD    R6, R6, #-1   ; Push R5 (caller's frame pointer,
STR    R5, R6, #0    ; aka dynamic link)

ADD    R5, R6, #-1   ; Set new frame pointer
ADD    R6, R6, #-2   ; Allocate memory for Volta's
               ; local variables
```


Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Returning from a call

- **Ending the callee function:**
  - If there is a return value, it is written into the return value slot of the activation record
  - The local variables are popped off the stack
  - The dynamic link (the caller's frame pointer) is restored to R5
  - The return address is restored to R7
  - The RET instruction returns control to the caller function
- **Returning to the caller function:**
  - The return value is popped off the stack
  - The arguments are popped off the stack

LC-3 code for returning from function

End Volta:

```
LDR   R0, R5, #0   ; Load local variable k,
STR   R0, R5, #3   ; and write it to return value slot

ADD   R6, R5, #1   ; pop local variables

LDR   R5, R6, #0   ; pop the dynamic link
ADD   R6, R6, #1

LDR   R7, R6, #0   ; pop the return address
ADD   R6, R6, #1

RET
```

Continued …

Watt resumes after Volta returns:

```
LDR     R0, R6, #0    ; Load the return value
                      ; at the top of the stack
STR     R0, R5, #0    ; i.e. w = Volta( w, 10 )
ADD     R6, R6, #1    ; pop the return value

ADD     R6, R6, #2    ; pop arguments

…                     ; code for Watt.
```

ISN’T THAT CLEVER??