Computer Systems

Lecture 12

Procedures and the Call Stack

Dr Lito Michala, Dr José Cano
School of Computing Science
University of Glasgow

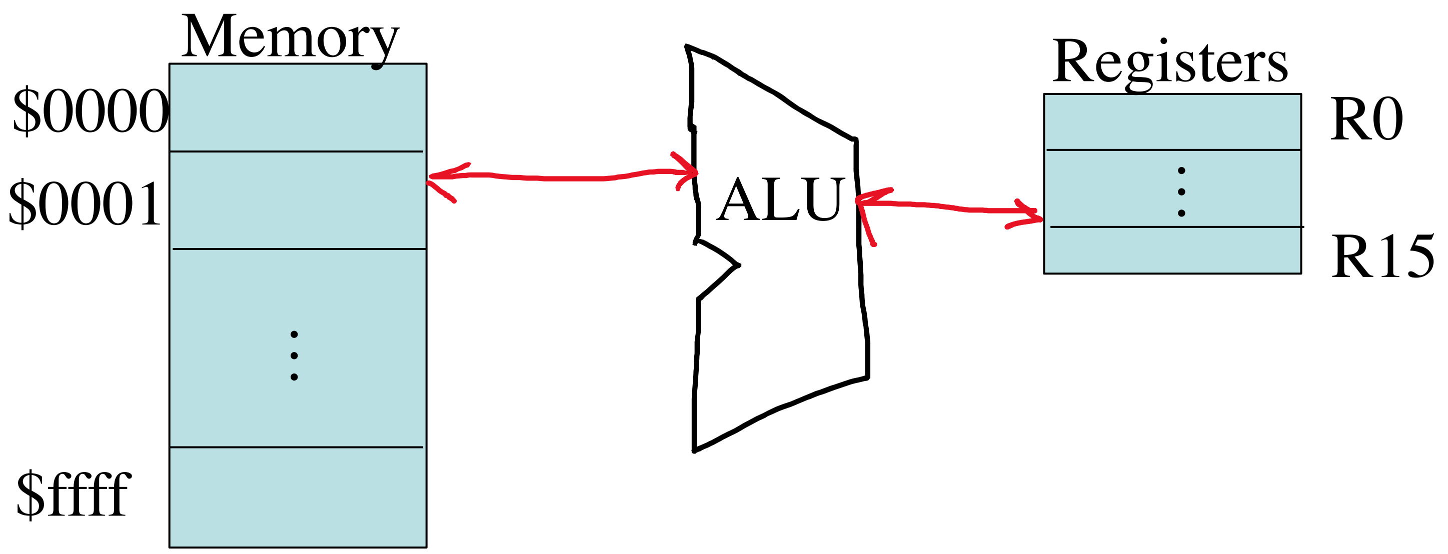
Spring 2020

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

Pointers quick-review

- A pointer is an address
  - much like your home address on google maps
- A value is what the pointer points at
  - much like you, in your home, on google maps
- Pointers and values are both stored in memory



Procedures: reusable code

- Often there is a sequence of instructions that comes up again and again
  - For example: sqrt (square root)
  - It takes a lot of instructions to calculate a square root
  - An application program may need a square root in many different places

- We don't want to keep repeating the code
  - It's tedious
  - It wastes space (all those instructions require memory!)

- The aim: write it once and reuse the same instructions many times

Procedure

- Write the code one time: the block of code is called a procedure (or subroutine, or function)
- Put the instructions by themselves somewhere, not in the main flow of instructions
- Give the block of code a label (e.g. work) that describes what it does
- Every time you need to perform this computation, call it: *goto work*
- When it finishes, the procedure needs to return
  - Go back to **the instruction after** the one that jumped to it

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

Call and return

- One idea is just to use jump instructions for both call and return
- But that isn't actually sufficient
  - Let's look in more detail at what happens

Returning to the instruction after the call

- Suppose a procedure named dowork is used in several places
- Each call jumps to the same place (the address of the first instruction of the procedure
- But the calls come from different places
- Therefore the procedure must finish by returning to different places

Calling and returning

- Here is a main program that calls a procedure “dowork” several times
  - It takes the value in R1 and doubles it, and the main program would use the result but we ignore that here

```
        load      R1,x[R0]       ; R1 = x
        “goto”    dowork[R0]     ; call the procedure “dowork”
        load      R1,y[R0]       ; R1 = y
        “goto“    dowork[R0]     ; call dowork
        sub       R5,R6,R7       ; R5 = R6-R7
dowork  add       R1,R1,R1       ; R1 = R1+R1
        “return”                 ; return (go back to the caller)
```

- The return must go to different points in the program: How can it do that?

The jump-and-link instruction: jal

- When the main program calls the subroutine, it needs to remember where the call came from
- This is the purpose of the jal instruction (jump and link)
- jal R5,dowork[R0]
  - A pointer to the next instruction after the jal (the return address) is loaded into the destination register (e.g. R5)
  - Then the machine jumps to the effective address

Jumping

- All jump instructions (jump, jal, jumplt, etc) refer to effective addresses
  - jump loop[R0]

    goto loop
  - jump 0[R14]

    goto instruction whose address is in R14
  - jump const[R2]

    goto instruction whose address is const+R2

Implementing call and return

- To call a procedure dowork: jal R13,dowork[R0]
  - The address of the instruction after the jal is placed in R13
  - The program jumps to the effective address, and the procedure starts executing

- To return when the procedure has finished: jump 0[R13]
  - The effective address is 0 + the address of the instruction after the jal
  - The program jumps there and the main program resumes

Calling with jal and returning with jump

```
          load    R1,x[R0]              ; R1 = x
          jal     R13,dowork[R0]        ; call dowork
          load    R1,y[R0]              ; R1 = y
          jal     R13,dowork[R0]        ; call dowork
          sub     R5,R6,R7              ; R5 = R6-R7
dowork    add     R1,R1,R1              ; R1 = R1+R1
          jump    0[R13]                ; return
```

- **call**: jal puts a pointer to the next instruction into R13
- **return**: follow the pointer in R13

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

Parameter passage

- There are several different conventions for passing argument to the function, and passing the result back

- What is important is that the caller and the procedure agree on how information is passed between them

- If there is a small number of arguments, the caller may put them in registers before calling the procedure

- If there are many arguments, the caller builds an array or vector (sequence of adjacent memory locations), puts the arguments into the vector

- and passes the address of the vector in a register (typically R1)

- A simple convention: the argument and result are passed in R1

Functions

- A function is a procedure that
  - Receives a parameter (a word of data) from the caller
  - Calculates a result
  - Passes the result back to the caller when it returns
- A pure function is a function that doesn't do anything else
  - It doesn't change any global variables, or do any input/output

Example: Passing argument and result in R1

```
; Main program
    load R1,x[R0]          ; arg = x
    jal R13,work[R0]       ; result = work (x)
...
     load R1,y[R0]         ; arg = y
     jal R13,work[R0]      ; result = work (y)
...
; Function work (x) = 1 + 7*x
work lea R2,7[R0]          ; R7 = 2
        lea R3,1[R0]       ; R3 = 1
        mul R1,R1,R2       ; result = arg * 7
        add R1,R3,R1       ; result = 1 + 7*arg
        jump 0[R13]        ; return
```

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

What if a procedure calls another procedure?

- The simplest kind of procedure
  - Call it with jal R13,procname[R0]
  - It returns by executing jump 0[R13]

Limitations of basic call

- If the procedure modifies any registers, it may destroy data belonging to the caller
- If the procedure calls another procedure, it can't use R13 again
  - Each procedure would need a dedicated register for its return address, limiting the program to a small number of procedures
- The basic call mechanism doesn't allow a procedure to call itself
  - This is called recursion

R13 overwritten: proc1 returns to the wrong place!

```
        load    R1,x[R0]          ; R1 := x
        jal     R13,proc1[R0]     ; call proc1
        load    R1,y[R0]          ; R1 := y
                                                    R13

        store   R1,result[R0]     ; result := 2  (2x)2
proc1   lea     R2,1[R0]          ; R2 := 1
proc1   add     R1,R1,R2          ; R1 := R1+1
        jal     R13,proc2[R0]     ; R1 := square (R1)
        add     R1,R1,R2          ; R1 := R1+1
        jump    0[R13]            ; return

square  mul     R1,R1,R1          ; R1 = R1*R1
        jump    0[R13]            ; return
```

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

Saving state

- Calling a procedure creates new information
  - The return address
  - Whatever values the procedure loads into the registers

- But this new information could overwrite essential information belonging to the caller

- We need to save the caller's state so the procedure won't destroy it

The wrong way to save state

- Suppose we just have a variable saveRetAdr
- Store R13 into it in the procedure, load that when we return
- Now it's ok for proc1 to call proc2
- But if proc2 calls proc3 we are back to the same problem: it doesn’t work!
- The solution: a stack

Saving registers

- Most procedures need to use several registers
- It's nearly impossible to do anything without using some registers!
- The first thing a procedure should do is to save the registers it will use by copying them into memory (with store instructions)
- The last thing it should do before returning is to restore the registers by copying their values back from memory (with load instructions)

Where can the registers be saved?

- It won't work to copy data from some of the registers to other registers!
- It's essential to save the data into memory
- Two approaches
  - Allocate fixed variables in memory to save the registers into (simple but doesn't allow recursión)
  - Maintain a stack in memory, and push the data onto the stack (this is the best approach and is used by most programming languages)

Who saves the state: the caller or the procedure?

- Two approaches

- Caller saves (used occasionally)
  - Before calling a procedure, the caller saves the registers, so all its essential data is in memory
  - After the procedure returns, the caller does whatever loads are needed

- Callee saves (usually the preferred solution)
  - The caller keeps data in registers, and assumes that the procedure won't disturb it
  - The first thing the procedure does is to save the registers it needs to use into memory
  - Just before returning, the procedure restores the registers by loading the data from memory

Outline

- Procedures
- Call and return
- Parameter passage
- Procedure calls another procedure
- Saving state
- Stack

Stack of return addresses

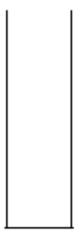
- To allow a large number of procedures, we can't dedicate a specific register to each one for its return address

- Therefore we
  - Always use the same register for the return address in a jal instruction (we will use R13)
  - The first thing a procedure does is to store its return address into memory
  - The last thing the procedure does is to load its return address and jump to it
  - The return addresses are pushed onto a stack, rather than being stored at a fixed address

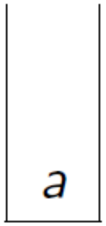
Stacks

- A stack is a container
- Initially it is empty
- You can push a value onto the stack: this is now sitting on the top of the stack
- You can pop the stack: this removes the most recently pushed value and returns it
- A stack allows access only to the top value; you cannot access anything below the top
- We can save procedure return addresses on a stack because return always needs the most recently saved return address

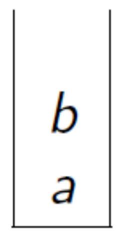
Initially the stack is empty



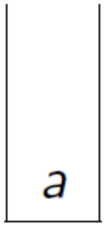
Call procedure, push return address a



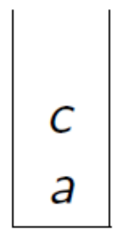
Call another procedure, push return address b



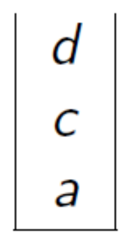
Return: pop produces return address b



Call some procedure, push return address c



Call a procedure, push return address d



The call stack

- Central technique for
  - Preserving data during a procedure call
  - Holding most of your variables

- It goes by several names (these are all the same thing)
  - call stack
  - execution stack
  - “The stack”

- It's important!
  - Most programming languages use it
  - Computers are designed to support it
  - Often referred to (Stack Overflow web site, etc.)

Stack frames

- There is a call stack or execution stack that maintains complete information about all procedure calls and returns

- Every “activation” of a procedure pushes a stack frame

- When the procedure returns, its stack frame is popped (removed) from the stack

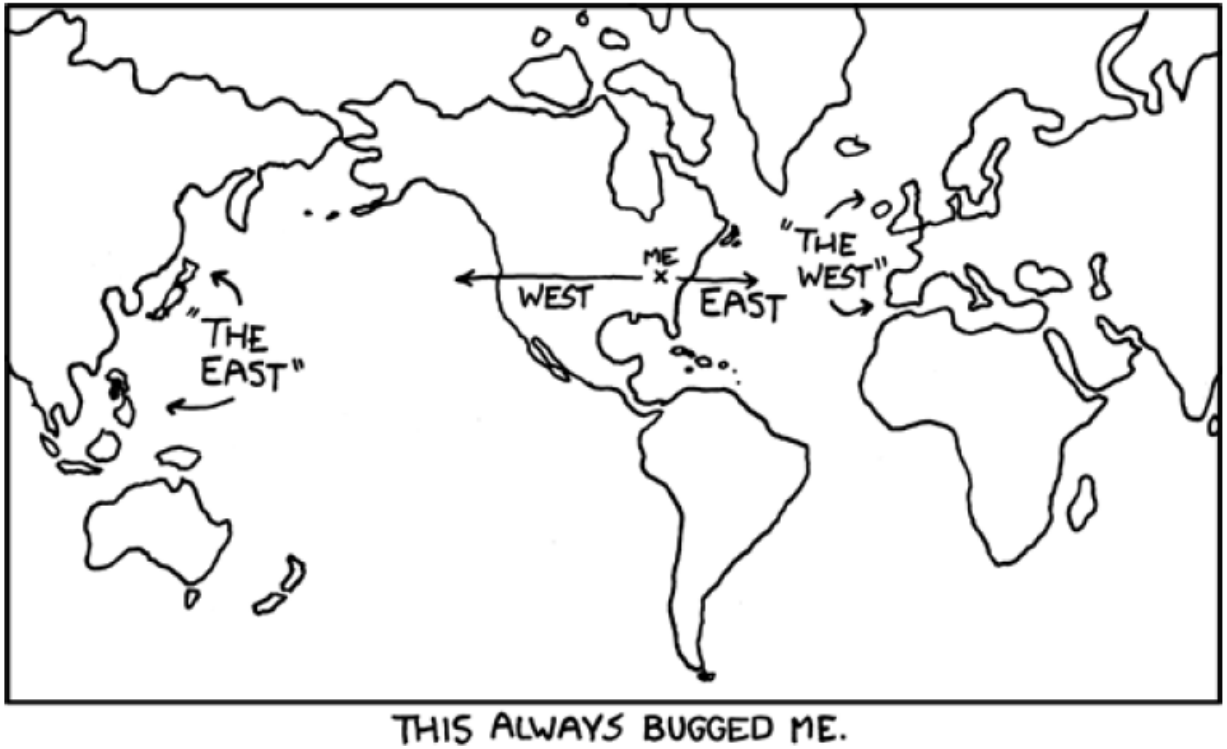
- R14 contains the address of the current (top) stack frame

- The stack frame contains
  - A pointer to the previous stack frame (this is required to make the pop work)
  - The return address (saved value of R13)
  - The saved registers (so the procedure can use the registers without destroying information)
  - Local variables (so the procedure can have some memory of its own to use)

Implementing the call stack

- Dedicate R14 to the stack pointer
- This is a programming convention, not a hardware feature
- When the program is started, R14 will be set to point to an empty stack
- When a procedure is called, the saved state will be pushed onto the stack: store a word at 0[R14] and add 1
- When a procedure returns, it pops the stack and restores the state: subtract 1, load from 0[R14]
- The program should never modify R14 apart from the push and pop

terminology



https://xkcd.com/503/