COMP 3221

Microprocessors and Embedded Systems

Lectures 15 : Functions in C/ Assembly - I

http://www.cse.unsw.edu.au/~cs3221

August, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au

Overview

- C functions
- Bookkeeping for function call/return
- Instruction support for functions
- Nested function calls
- C memory allocation: static, heap, stack
- Conclusion

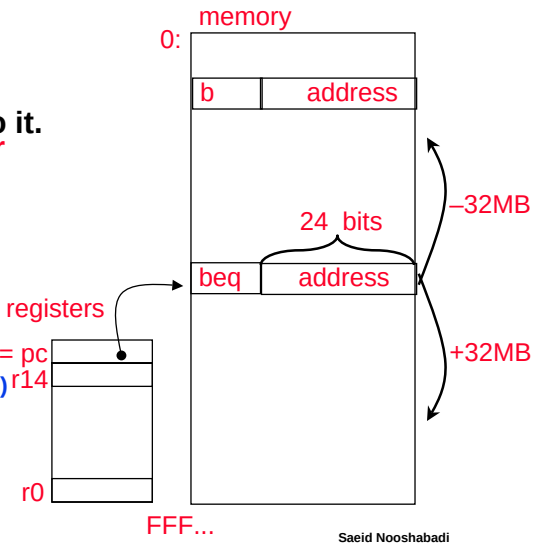
Review

- HLL decisions (if, case) and loops (while, for) use same assembly instructions
  - Flag Setting Instructions: `cmp, cmn, tst, teq` in ARM
  - Data Processing Instructions with Flag setting Feature: `adds, subs, ands`, in ARM
  - Conditional branches: `beq, bne, bgt, blt,` etc in ARM
  - Conditional Instructions: `addeq, ldreq,etc` in ARM
  - Unconditional branches: `b, bal,` and `mv pc, Rn` in ARM
  - Switch/Case: chained if-else or jump table + `ldr pc, [ ]`

  - `ldr pc, [ ]` is VERY POWERFUL!

Review: Branches: PC-relative addressing

- Recall register r15 in the machine also called PC;
- points to the currently executing instruction
- Most instruction add 4 to it. (pc increments by 4 after execution of most instructions)
- Branch changes it to a specific value
- Branch adds to it
  - 24-bit signed value (contained in the instruction)
  - Shifted left by 2 bits
- Labels => addresses

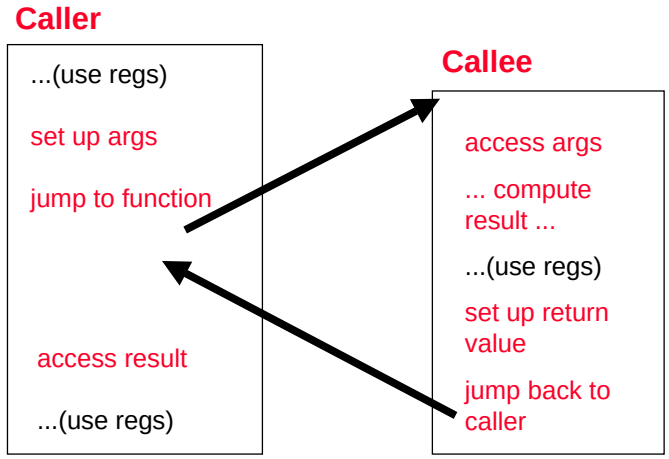


C functions

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

 i = mult(j,k); ... ;
 m = mult(i,i); ...

}

int mult (int mcand, int mlier)

{
 int product = 0;
 while (mlier > 0)  {
   product = product + mcand;
   mlier = mlier -1;

 }
 return product;

}
```

What information must compiler/program keep track of?

Basics of Function Call



Function Call Bookkeeping

- Procedure address
- Return address
- Arguments
- Return value
- Local variables
- Registers (conflicts)

Registers for functions

→ `lr = r14`

→ `a1, a2, a3, a4`

→ `a1`

→ `v1, v2, v3, v4, v5, v6, v7`

=>ARM Procedure Call Standards (APCS) conventions for use of registers simplify bookkeeping

APCS Register Convention: Summary

register name	software name	use and linkage
r0 – r3	a1 – a4	first 4 integer args
		scratch registers
		integer function results
r4 – r11	v1- v8	local variables
r9	sb	static variable base
r10	sl	stack limit
r11	fp	frame pointer
r12	ip	intra procedure-call scratch pointer
r13	sp	stack pointer
r14	lr	return address
r15	pc	program counter

Red are SW conventions for compilation, blue are HW

ARM Procedure Call Standard (APCS)

Instruction Support for Function Call?

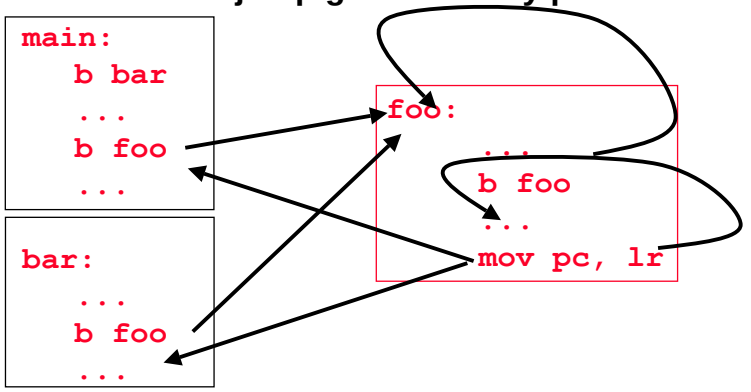
C
```
... sum(a,b);... /* a,b:v1,v2 */
}
int sum(int x, int y) {
    return x+y;
}
```

ARM
```
address
1000 mov a1,v1     ; x = a
1004 mov  a2, v2   ; y = b
1008 mov lr,#1016  ; lr = 1016
1012 b sum         ; jump to sum
1016 ...

2000 sum: add a1,a1,a2
2004 mov pc, lr     ; b 1016
```

Why `mov pc, lr` *vs.* `b 1016` *to return?*

`b` vs `mov pc, lr` : why jump register?

- Consider a function `foo` that is called from several different places in your program
  - each call is performed by a `b` instruction
    - `b foo`
  - but the return jump goes to many places



A Level of Indirection

- Solves many problems in CS!
- How do you make a jump instruction behave differently each time it is executed?
  - indirection
  - `mov pc, lr` returns control to last caller
    - recorded in a registers
- How do you make an instruction fetch a different element of an array each time though a loop?
  - indirection
  - update index address register of `ldr`, and `str`

Accessing array elements => indirection

```
int sumarray(int arr[]) {
  int i, sum;

  for(i=0;i<100;i=i+1)
      sum = sum + arr[i];

}
```

```
      mov   v1, #0          ; clear v0
      add   a2,a1,#400       ; beyond end of arr[]
Loop: cmp   a1,a2

      bge   Exit
      ldr   a3, [a1], #4    ; a3=arr[i], a1++
      add   v1,v1,a3        ; v1= v1+ arr[i]
      b     Loop
Exit: mov lr, pc
```

Instruction Support for Functions?

- Single instruction to branch and save return address: branch and link (`bl`):
- Before:

```
1008 mov lr, #1016 ;lr = 1016
1012 b sum         ;goto sum
```

- After:

```
1012 bl sum  ; lr = 1016,goto sum
```

- Why `bl`? Make the common case fast
  - and elegance

Nested Procedures (#1/2)

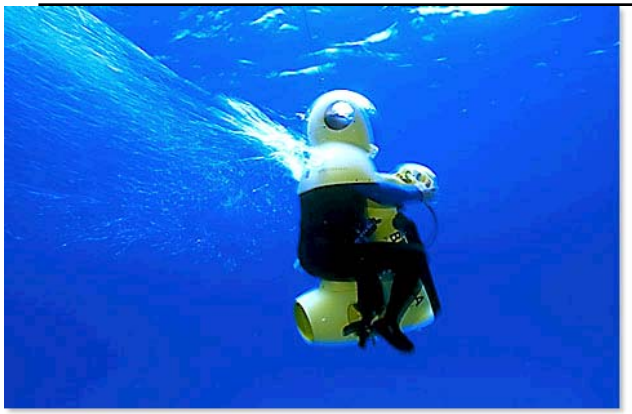
```
...sumSquare(a,b)...

int sumSquare(int x, int y) {
    return mult(x,x)+ y;
 }
```

- Need to save `sumSquare` return address saved in `lr` by `bl sumSquare` instruction, before call to `mult`
  - Otherwise `bl mult` overwrites `lr`
- One word per procedure in memory ?
  - e.g., `str lr, [sp,sumSquareRA]  ; sp = r13`
- Recursive procedures could overwrite saved area => need safe area per function invocation => stack

Nested Procedures (#2/2)

- In general, may need to save some other info in addition to `lr`.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes
  - Heap: Variables declared dynamically (such as counters in `for` loops, or by function `malloc()`)
  - Stack: Space to be used by procedure during execution; this is where we can save register values

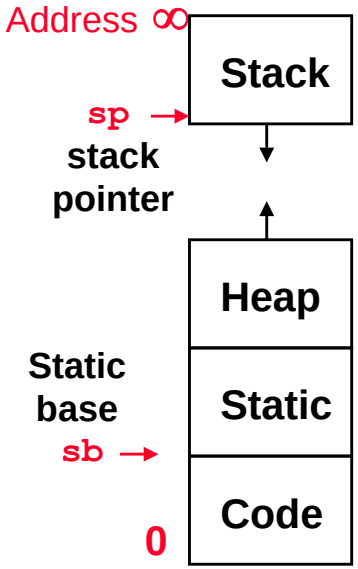
"What's This Stuff Good For?"



Breathing Observation Bubble: BOB pipes air from a tank under the handlebars into an acrylic dome, replacing a diver's face mask and breathing apparatus. Wireless technology lets riders talk to other BOBsters darting through the water nearby, as well as to armchair divers above in a boat or back on shore. Saving energy from not having to kick, divers can stay submerged almost an hour with the BOB. Like most modern scuba gear, the BOB features a computer that tells riders when to come up and calculates decompression times for a safe return to the surface. *One Digital Day*, 1998 www.intel.com/onedigitalday

What do applications ("apps") like these mean for reliability requirements of our technology?

C memory allocation seen by the Program



Space for saved procedure information

Explicitly created space, e.g., malloc(); C pointers

Variables declared once per program

Program

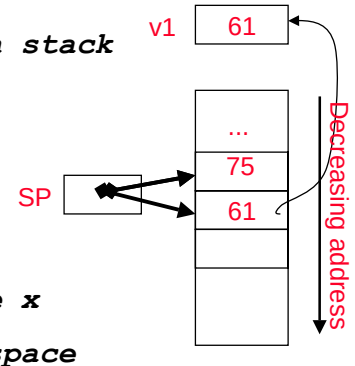
Stack Operations

° PUSH v1

```
sub sp,sp, #4    ; space on stack
str v1, [sp,#0]  ; save x
```

° POP v1

```
ldr v1, [sp,#0]  ; restore x
add sp, sp,#4 ; => stack space
```

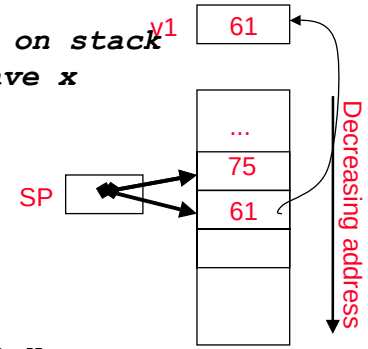


What does sp point to?

Stack Operations (Better Way)

° PUSH v1

```
str v1, [sp,#-4]!  ; space on stack
                   ; and save x
```

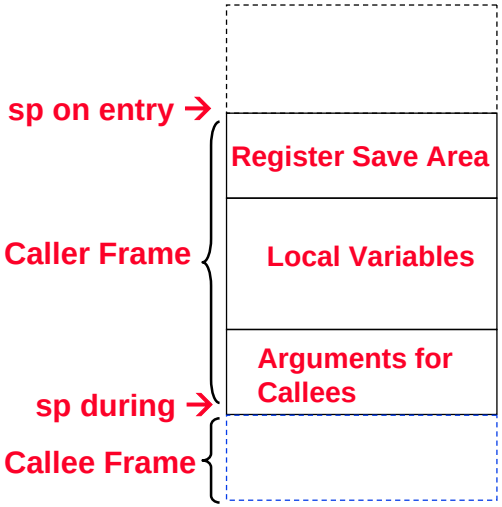


° POP v1

```
ldr v1, [sp], #4 ; restore x
                 ; and reclaim stack space
```

What does sp point to?

Typical Structure of a Stack Frame



- compiler calculates total frame size (F)
- at start of call, subtract F from SP
- at end, add F to SP
- where are args from caller?
- where are args for callee

“And in Conclusion …” (#1/2)

- ARM Assembly language instructions
  - Unconditional branches: `b`, `mov pc, rx` , `bl`
- Operands
  - Registers (word = 32 bits); `a1 - a3`, `v1 – v8`, `ls, sb, fp, sp, lr`

“And in Conclusion …” (#2/2)

- Functions, procedures one of main ways to give a program structure, reuse code
- `mov pc, Rn` required instruction; most add `bl` (or equivalent) to make common case fast
- Registers make programs fast, but make procedure/function call/return tricky
- ARM SW convention divides registers for passing arguments, return address, return value, stack pointer