COMP 3221

Microprocessors and Embedded Systems

Lectures 17 : Functions in C/ Assembly - III

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

September, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au

Overview

° Why Procedure Conventions?

° Basic Structure of a Function

° Example: Recursive Function

° Instruction Support for Function

° Block Store and Load

° Conclusion

Review: 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)

Review: Function Call Bookkeeping

° **Big Ideas:**
- Follow the procedure conventions and nobody gets hurt.
- Data is just 1's and 0's, what it represents depends on what you do with it

- **Function Call Bookkeeping:**
  - Caller Saved Registers are saved by the caller, that is, the function that includes the `bl` instruction
  - Callee Saved Registers are saved by the callee, that is, the function that includes the `mov pc, lr` instruction
  - Some functions are both a caller and a callee

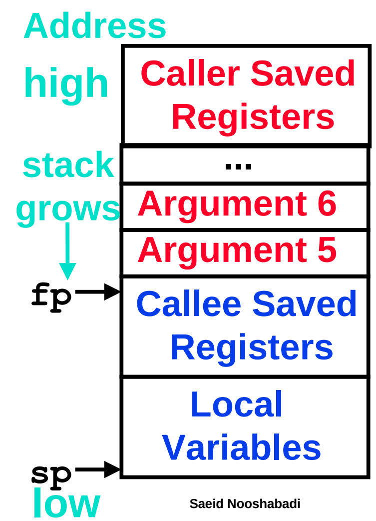
Review: Caller & Callee Saved Registers

- **Caller Saved Registers:**
  - Return address `lr`
  - Arguments `a1, a2, a3, a4`
  - Return value `a1, a2, a3, a4`
- **Callee Saved Registers:**
  - `v` Registers `v1 - v8`

Review: StackMemory Allocation on Call

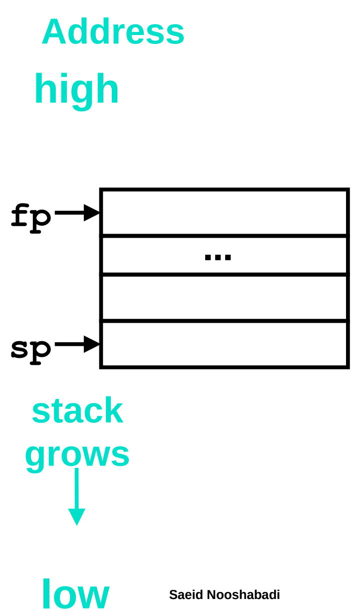
C Procedure Call Frame

- Pass arguments (4 regs)
- If called from another functions, save `lr`
- Save caller-saved regs
- Save additional Arguments
- `bl`
- Save old `sp` and `fp` & set `fp` 1st word of frame (old `sp-4`)
- Save callee-saved regs



Review: Memory Deallocation on Return

- Move return value into `a1`
- Restore callee-saved regs from the stack
- Restore old `sp` and `fp` from stack
- `mov pc , lr`
- Restore caller-saved regs
- If saved `lr`, restore it



Why Procedure Conventions? (#1/2)

- **Think of procedure conventions as a contract between the Caller and the Callee**
  - If both parties abide by a contract, everyone is happy (:))
  - If either party breaks a contract, disaster and litigation result (: O)
- **Similarly, if the Caller and Callee obey the procedure conventions, there are significant benefits. If they don’t, disaster and program crashes result**

Why Procedure Conventions? (#2/2)

° **Benefits of Obeying Procedure Conventions:**

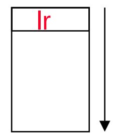
- People who have never seen or even communicated with each other can write functions that work together
- Recursion functions work correctly

Basic Structure of a Function

Prologue

```
entry_label:
    sub sp,sp, #fsize    ; create space on stack
    str lr,[sp, #fsize-4]; save lr
                         ; save other regs
```

Body . . .



Epilogue

```
                          ;restore other regs
    ldr lr, [sp,#fsize-4];restore lr
    add sp, sp,  #fsize  ;reclaim space on stack
    mov pc, lr
```


Example: Compile This (#1/5)

```
main() {
 int i,j,k,m; /* i-m:v1-v4 */

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

return 0

}

int mult (int mcand, int mlier){
 int product;

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

}
```


Example: Compile This (#2/5)

```
__start:

 str lr, [sp,#-4]!; store return
                  ; address
 mov a1,v2        ; arg1 = j
 mov a2,v3        ; arg2 = k
 bl mult          ; call mult
 mov v1, a1       ; i = mult()
 ...
 mov a1, v1       ; arg1 = i
 mov a2, v1       ; arg2 = i
 bl mult          ; call mult
 mov v4, a1       ; m = mult()
 ...

ldr lr, [sp,#4]!  ; restore return address

mov pc, lr
```

Example: Compile This (#3/5)

° **Notes:**

- `main` function returns to O/S, so `mov pc, lr`, so there's need to save `lr` onto stack
- all variables used in `main` function are callee saved registers (`"v"`), so there's no need to save these onto stack

Example: Compile This (#4/5)

```
mult:
    mov a3, #0          ; prod=0

Loop:
    cmp a2,#0           ; mlier > 0?
    beq Fin             ; no=>Fin
    add a3,a3,a1        ; prod+=mcand
    sub a2,a2,#1        ; mlier-=1
    b    Loop           ; goto Loop

Fin:
    mov  a1,a3          ; a1=prod
    mov  pc, lr         ; return
```


Example: Compile This (#5/5)

° **Notes:**

- no `bl` calls are made from `mult` and we don't use any callee saved (`"v"`) registers, so we don't need to save anything onto stack
- Scratch registers `a1 – a3` are used for intermediate calculations
- `a2` is modified directly (instead of copying into a another scratch register) since we are free to change it
- result is put into `a1` before returning

Fibonacci Rabbits

° **Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on.**

° **How many pairs will there be in one year?**

Fibonacci's Puzzle

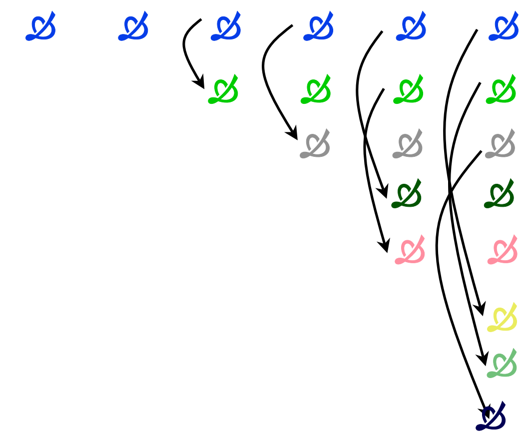
Italian, mathematician Leonardo of Pisa (also known as Fibonacci) 1202.

Fibonacci Rabbits (Solution)

1. At the end of the first month, they mate, but there is still one only 1 pair.
2. At the end of the second month the female produces a new pair, so now there are 2 pairs of rabbits in the field.
3. At the end of the third month, the original female produces a second pair, making 3 pairs in all in the field.
4. At the end of the fourth month, the original female has produced yet another new pair, the female born two months ago produces her first pair also, making 5 pairs.

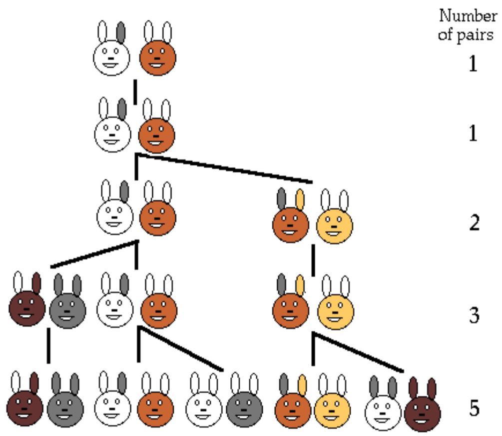
Fibonacci Rabbits (Solution Animated)

Month →	0	1	2	3	4	5



Rabbit Numbers	0	1	1	2	3	5	8

Fibonacci Rabbits (Solution in Picture)



The number of pairs of rabbits in the field at the start of each month is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Example: Fibonacci Numbers (#1/6)

- The Fibonacci numbers are defined as follows:
- F(n) = F(n – 1) + F(n – 2)

  F(0) and F(1) are defined to be 1
- In C, this could be written:

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

}
```

Example: Fibonacci Numbers (#2/6)

- Now, let's translate this to ARM!
- You will need space for three words on the stack
- The function will use `v1`
- Write the Prologue:

```
fib:
str lr, [sp,#-4]!       ; Save the return address
str v1, [sp,#-4]!       ; Save v1 & Push the
                        ; stack frame
```


Example: Fibonacci Numbers (#3/6)

- Now write the Epilogue:

```
fin:
ldr v1, [sp,#4]!        ; Restore v1
ldr lr, [sp,#4]!        ; Restore return address
                        ; Pop the stack frame
mov pc,lr               ; Return to caller
```


Example: Fibonacci Numbers (#4/6)

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
 if(n == 0) { return 1; } /*Translate Me!*/
 if(n == 1) { return 1; } /*Translate Me!*/
 return (fib(n - 1) + fib(n - 2));
}
```

```
  cmp     a1, #0          ; if (n == 0). . .
  cmpne   a1, #1          ; if (n == 1). . .
  moveq, a1, #1           ; . . .
  beq   fin               ; return 1
```

Continued on next slide. . .

Example: Fibonacci Numbers (#5/6)

- Almost there, but be careful, this part is tricky!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
str a1,  [sp, #-4]!     ; Need a1 after bl
sub a1, a1, #1          ; a1 = n - 1
bl fib                  ; fib(n - 1)
mov v1, a1              ; Save return value
ldr a1, [sp, #4]!       ; Restore a1
sub a1, a1, #2          ; a1 = n - 2
```

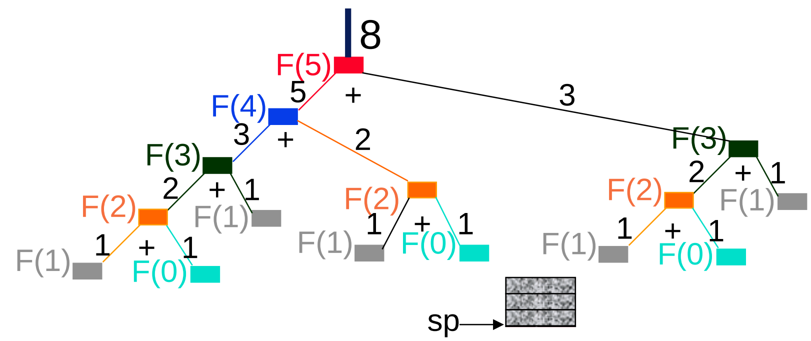
Continued on next slide. . .

Example: Fibonacci Numbers (#6/6)

° Remember that is `v1` Callee Save and `a1` is caller saved!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
bl fib              ; fib(n-2)
add a1, a1, v1      ; a1 = fib(n-1) + fib(n-2)
;To the epilogue and beyond. . .
```


Stack Growth and Shrinkage



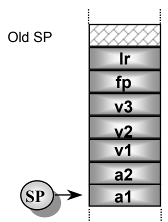

```
int fib(int n) {
if(n == 0) { return 1; }
if(n == 1) { return 1; }
return (fib(n - 1) + fib(n - 2);
}
```


Instruction Support for Stack

° Consider the following code:

Prologue

```
str lr, [sp,#-4]!
str fp, [sp,#-4]!
str v3, [sp,#-4]!
str v2, [sp,#-4]!
str v1, [sp,#-4]!
str a2, [sp,#-4]!
str a1, [sp,#-4]!
```



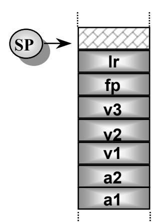
Store Multiple
Full Descending

Body ...

```
stmfd sp!, {a1-a2,v1-v3,fp,lr}
ldmfd sp!, {a1-a2,v1-v3,fp,lr}
```

Epilogue

```
ldr a1, [sp,#4]!
ldr a2, [sp,#4]!
ldr v1, [sp,#4]!
ldr v2, [sp,#4]!
ldr v3, [sp,#4]!
ldr fp, [sp,#4]!
ldr lr, [sp,#4]!
```



Load Multiple
Full Descending

Block Copy via Stack Operation

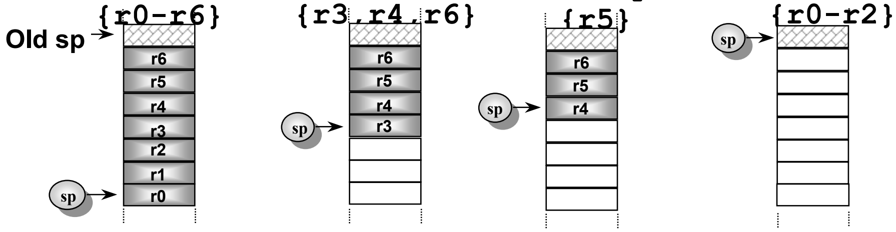
° The contents of registers `r0` to `r6` need to be swapped around thus:

- `r0` moved into `r3`
- `r1` moved into `r4`
- `r2` moved into `r6`
- `r3` moved into `r5`
- `r4` moved into `r0`
- `r5` moved into `r1`
- `r6` moved into `r2`

° Write a segment of code that uses full descending stack operations to carry this out, and hence requires no use of any other registers for temporary storage.

Block Copy Sample Solution

`stmfd sp!, {r0-r6}`   `ldmfd sp!, {r3,r4,r6}`   `ldmfd sp!, {r5}`   `ldmfd sp!, {r0-r2}`




```
r3 = r0      r5 = r3      r0 = r4
r4 = r1                   r1 = r5
r6 = r2                   r2 = r6
```


Direct functionality of Block Data Transfer

- When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:
  - i.e. specify whether to increment / decrement the base pointer, before or after the memory access.
- In order to do this, LDM / STM support a further syntax in addition to the stack one:
  - STMIA / LDMIA : Increment After
  - STMIB / LDMIB : Increment Before
  - STMDA / LDMDA : Decrement After
  - STMDB / LDMDB : Decrement Before

For details See Chapter 3, page 61 – 62
Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6.

“And in Conclusion …”

- ARM SW convention divides registers into those calling procedure save/restore and those called procedure save/restore
  - Assigns registers to arguments, return address, return value, stack pointer
- Optional Frame pointer `fp` reduces bookkeeping on procedure call
- Use Stack Block copy Instructions `stmfd` & `ldmfd` to store and retrieve multiple registers to/from stack.