
ELEC2041

Microprocessors and Interfacing

Lectures 16 : Functions in C/ Assembly - II

<http://webct.edtec.unsw.edu.au/>

April 2005

Saeid Nooshabadi

saeid@unsw.edu.au

ELEC2041 lec16-function-II.1

Saeid Nooshabadi

Overview

- Resolving Registers Conflicts
- Caller /Callee Responsibilities
- Frame/Stack pointer
- Conclusion

ELEC2041 lec16-function-II.2

Saeid Nooshabadi

Review: Basics of Function Call

Caller

...(use regs)
set up args
jump to function

access result
...(use regs)

Callee

access args
... compute
result ...
...(use regs)
set up return
value
jump back to
caller

ELEC2041 lec16-function-II.3

Saeid Nooshabadi

Review: Function Call Bookkeeping

- Procedure address Registers for functions
- Return address → $lr = r14$
- Arguments → $a1, a2, a3, a4$
- Return value → $a1$
- Local variables → $v1, v2, v3, v4, v5, v6, v8$
- Registers (conflicts)

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

ELEC2041 lec16-function-II.4

Saeid Nooshabadi

Review: Instruction Support for Functions?

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

address

```
1004 mov a1,v1    ; x = a
1008 mov a2, v2   ; y = b
1012 bl sum      ; lr = 1016, goto sum
1016 ...
```

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

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- v7	local variables
r9	sb	static variable base
r10	sl	stack limit
r11	fp/v8	frame pointer/local variables
r12	ip	intra procedure-call scratch pointer
r13	sp	stack pointer
r14	lr	return address
r15	pc	program counter

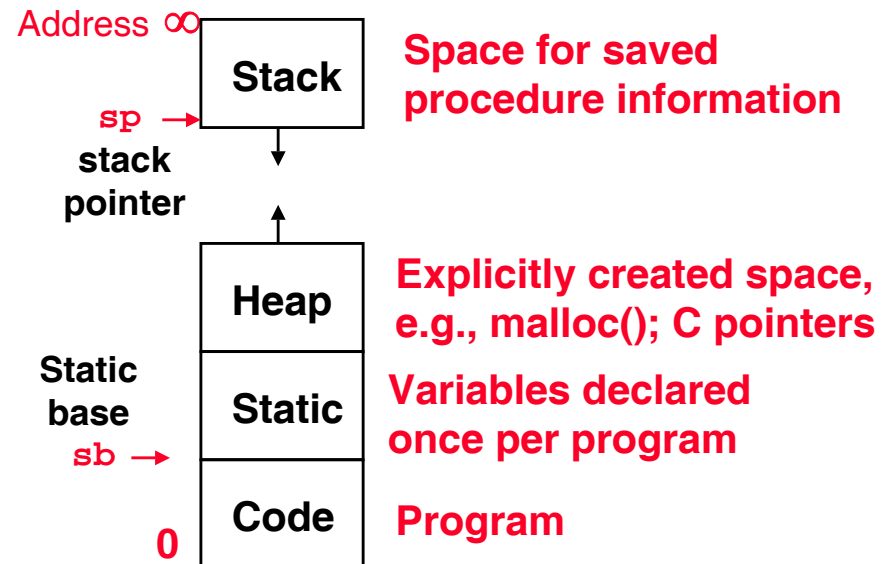
Red are SW conventions for compilation, blue are HW

ARM Procedure Call Standard (APCS)

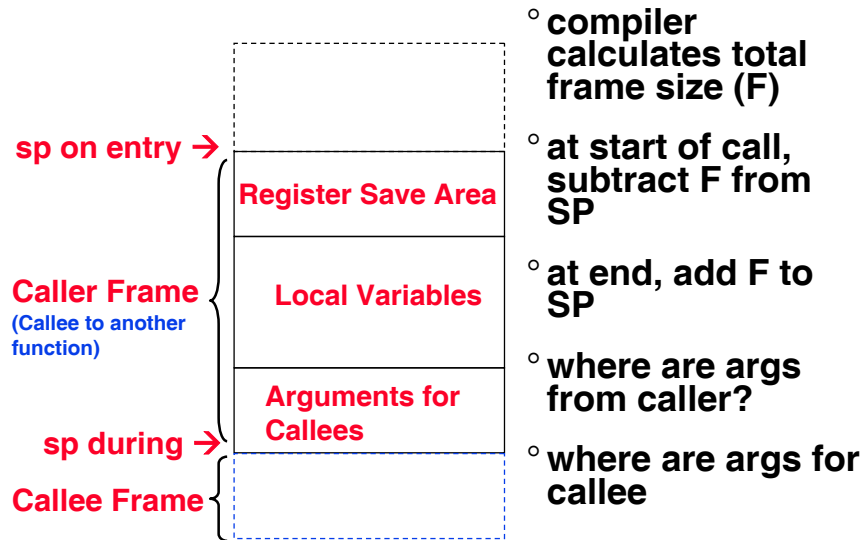
Review: Nested Procedures

- A caller function is itself called from another function.
- We need to store **lr** for the caller before it can call another function.
- In general, may need to save some other info in addition to **lr**.
- But; Where do we save these info?

Review: C memory allocation map



Review: Typical Structure of a Stack Frame



ELEC2041 lec16-function-IL.9

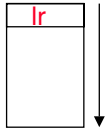
Saeid Nooshabadi

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
```

ELEC2041 lec16-function-IL.10

Saeid Nooshabadi

Compiling nested C func into ARM (#1/2)

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

sumSquare:

```
Prologue  sub sp,sp,#8        ; space on stack
          str lr,[sp,#4]      ; save ret addr
          str a2,[sp,#0]      ; save y
          mov a2,a1           ; mult(x,x)
          bl mult             ; call mult
Body      ldr a2,[sp,#0]      ; restore y
          add a1,a1,a2        ; mult()+y
          ldr lr,[sp,#4]      ; get ret addr
Epilogue  add sp,sp,#8        ; => stack space
          mov pc, lr
```

ELEC2041 lec16-function-IL.11

Saeid Nooshabadi

Compiling nested C func into ARM (#2/2)

sumSquare:

```
sub sp,sp,#8
str lr,[sp,#4]
str a2,[sp,#0]
mov a2,a1
bl mult
ldr a2,[sp,#0]
add a1,a1,a2
ldr lr,[sp,#4]
add sp,sp,#8
mov pc, lr
```

sp before entry to sumSquare →

sumSquare Frame

sp during sumSquare →

mult Frame

ELEC2041 lec16-function-IL.12

Saeid Nooshabadi

Compiling nested C func into ARM (Better way)

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

sumSquare:

A
R
M

```

Prologue str lr,[sp,#-4]!; save ret addr
          str a2,[sp,#-4]!; save y
          mov a2,a1        ; mult(x,x)
          bl mult          ; call mult
Body      ldr a2,[sp], #4   ; restore y
          add a1,a1,a2      ; mult()+y
          ldr lr,[sp], #4   ; get ret addr
          ; => stack space
Epilogue mov pc, lr
    
```

ELEC2041 lec16-function-IL.13

Saeid Nooshabadi

Function Call Bookkeeping: thus far

- Procedure address **x**
- Return address **x** **lr**
- Arguments **x** **a1 – a4**
- Return values **x** **a1 – a4**
- Local variables **x** **v1 – v8**
- Registers
 - what if they are reused?
 - what if there aren't enough?

ELEC2041 lec16-function-IL.14

Saeid Nooshabadi

ELEC2041 Readings for Week #6

- Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6. **Chapters 6**
- **Experiment 3 Documentation**
- **/lectures/stack.html & /lectures/stack example.html on Webct. They are notes on stacks**

ELEC2041 lec16-function-IL.15

Saeid Nooshabadi

Exceeding limits of registers

- Recall: assembly language has fixed number of operands, HLL doesn't
- Local variables: v1, ..., v8
 - What if more than 8 words of local variables?
- Arguments; a1, ..., a4
 - What if more than 4 words of arguments?
- Place extra variables and extra arguments onto stack (**sp**)
- Use scratch registers and data transfers to access these variables

ELEC2041 lec16-function-IL.16

Saeid Nooshabadi

Register Conflicts

- Procedure A calls Procedure B
 - A referred to as is “**calling procedure**” or “**caller**”
 - B referred to as is “**called procedure**” or “**callee**”
- Both A and B want to use the 15 registers
=> must cooperate

Register Conflicts: 2 options (A calls B)

- 1) **Called procedure/callee (B) leaves registers the way it found them** (except **lr**); its B's job to save it before using it and then restore it: “**callee saves**”
 - Since B only saves what it changes, more accurate is “**callee saves (what it uses)**”
- 2) **B can use any register it wants**;
Calling procedure/caller A must save any register it wants to use after call of B: “**caller saves**”
 - Since A knows what it needs after call, more accurate is “**caller saves (if it wants to)**”
- Is either optimal?

ARM Solution to Register Conflicts

- Divide registers into groups
 - Local variables / Callee Saves registers (**v1** – **v8**)
 - Scratch registers / Argument / Caller Save registers (**a1** – **a4**)
 - Some caller saves (if wants) and some callee saves (if used)
- Caller (**A**) save/restore scratch / argument (**a1** – **a4**) if needs them after the call; also **lr** → callee can use (**a1** – **a4**) and **lr**
- Callee (**B**) must save/restore local variables / callee saves registers (**v1** – **v8**) if it uses them → caller can leave them unsaved
 - Procedure that doesn't call another tries to use only scratch / argument registers (**a1** – **a4**)

Register Conventions (#1/5)

- Caller: the calling function
- Callee: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- **Register Conventions**: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**b1**) and which may be changed.

Register Conventions (#2/5)

- ° Three views of registers **a1 - a4**
- ° **a1 - a4** : **Change**. These are expected to contain new return values.

Or

- ° **a1 - a4** : **Change**. These are volatile argument registers.

Or

- ° **a1 - a4** : **Change**. They're called scratch: any procedure may change them at any time.

Register Conventions (#3/5)

- ° **v1 - v8** : **No Change**. Very important, that's why they're called callee saves registers / local variable. If the callee changes these in any way, it must restore the original values before returning.
- ° **sp**: **No Change**. The stack pointer must point to the same place before and after the **bl** call, or else the caller won't be able to restore values from the stack.
 - Grows downward, **sp** points to last full location
- ° **lr**: **Change**. The **bl** call itself will change this register.
- ° **ip**: **Change**. In most variants of APCS **ip** is an scratch register.

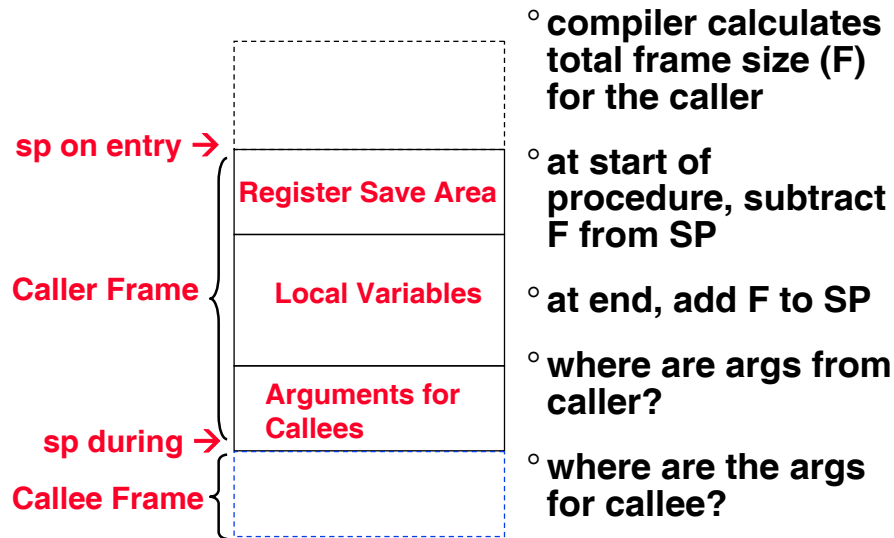
Register Conventions (#4/5)

- ° What do these conventions mean?
 - If function A calls function B, then function A must save any scratch registers **a1 - a4** that it may be using onto the stack before making a **bl** call.
 - Remember: Caller needs to save only registers it is using, not all scratch registers.
 - It also needs to store **lr** if A is in turn is called by another function.

Register Conventions (#5/5)

- ° Note that, even though the callee may not return with different values in the callee saves registers **v1 - v8**, it can use them by :
 - save **v1 - v8** on the stack
 - use these eight registers
 - restore **v1 - v8** from the stack
- ° The difference is that, with the scratch registers **a1 - a4**, the callee doesn't need to save them onto the stack.

Recall: Typical Structure of a Stack Frame



ELEC2041 lec16-function-IL.25

Saeid Nooshabadi

Callees' & Callers' Rights (Summary)

- **Callees' Right**
 - Right to use **a1 - a4** registers freely
 - Right to assume args are passed correctly in **a1 - a4**
- **Callers' Rights**
 - Right to use **v1 - v8** registers without fear of being overwritten by Callee
 - Right to assume return values will be returned correctly in **a1 - a4**

Keep this slide in mind for quiz and exam

ELEC2041 lec16-function-IL.26

Saeid Nooshabadi

Callee's Responsibilities (Summary)

1. If using **v1 - v8** or big local structs, slide **sp** down to reserve memory:
e.g. `sub sp, sp, #32`
2. If using **v1 - v8**, save before using:
e.g. `str v1, [sp, #28]`
3. Receive args in **a1 - a4**, additional args on stack
4. Run the procedure body
5. If not `void`, put return values in **a1 - a4**
6. If applicable, undo steps 2-1
e.g. `ldr v1, [sp, #28]`
`add sp, sp, #32`
7. `mov pc, lr`

Keep this slide in mind for quiz and exam

ELEC2041 lec16-function-IL.27

Saeid Nooshabadi

Caller's Responsibilities (Summary)

1. Slide **sp** down to reserve memory:
e.g. `sub sp, sp, #28`
2. Save **lr** on stack before **bl** to callee clobbers it:
e.g. `str lr, [sp, #24]`
3. If you'll still need their values after the function call, save **a1 - a4** on stack or copy to **v1 - v8** registers. Callee can overwrite "a" registers, but not "v" registers. e.g. `str a1, [sp, #20]`
4. Put first 4 words of args in **a1 - a4**, at most 1 arg per word, additional args go on stack: "arg 5" is `[sp, #0]`, "arg 6" is `[sp, #4]`
5. **bl** to the desired function
6. Receive return values in **a1 - a4**
7. Undo steps 3-1: e.g. `ldr a1, [sp, #20]`,
`ldr lr, [sp, #24]`, `add sp, sp, #28`

Keep this slide in mind for quiz and exam

ELEC2041 lec16-function-IL.28

Saeid Nooshabadi

Compile using pencil and paper Example 1 (#1/2)

```
int Doh(int i, int j, int k, int l){  
    return i+j+l;  
}
```

Doh:

add a1,

Compile using pencil and paper Example 1 (#2/2)

```
int Doh(int i, int j, int k, int l){  
    return i+j+l;  
}
```

Doh: add a1, a2, a1

"a"
Regs
Safe
For
Callee

add a1, a4 a1

mov pc lr

Compile using pencil and paper Example 2 (#1/2)

```
int Doh(int i, int j, int k, int m,  
char c, int n){  
    return i+j+n;  
}
```

Doh:

add a1,

Compile using pencil and paper Example 2 (#2/2)

```
int Doh(int i, int j, int k, int m,  
char c, int n){  
    return i+j+n;  
}
```

Doh: ldr ip, [sp, #4]

6th argument

add a1, a1, ip

add a1, a1, a2

mov pc lr

"a" &
ip Regs
Safe
For
Callee

```
int Doh(int i, int j, int k, int m, char c,  
        int n){return mult(i,j)+n;}
```

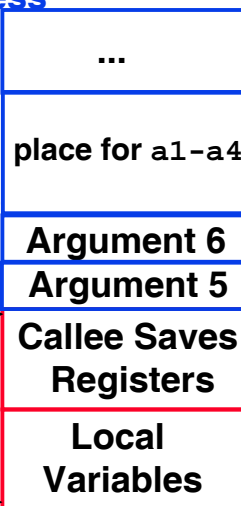
Doh: _____	add _____
_____	_____
_____	_____
_____	_____
bl _____	_____

```
int Doh(int i, int j, int k, int m, char c,  
        int n){ return mult(i,j)+n;}
```

Doh: <u>sub sp, sp, #4</u>	add <u>a1, a1, v1</u>
We need <u>str lr, [sp]</u>	
to save	
and	<u>ldr lr, [sp]</u>
restore	
v1 as well <u>ldr v1, [sp, #8]</u>	<u>add sp, sp, #4</u>
<u>bl mult</u>	<u>mov pc, lr</u>

3. Need n after funcall => v1

sp—
low



- Optional Frame pointer \mathbf{fp} reduces bookkeeping on procedure call