CSCI 6461 Computer Systems Architecture

Functions and the Stack

Labels as functions in ARM32

Call and return

C function invocation example


```
// this is the "callee"
int find_max(int a, int b, int c) {
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}
// this is the "caller"
int main() {
  int max = find_max(3,44,11);
  printf("The maximum is %d\n", max);
}
```

We are all familiar with how function invocations, parameter passing, local variables and return parameters work in C

Let's define the **caller** (`main`) and the **callee** (`find_max`)

Sequence of Events in Call/Return

Call(**caller**):

(1) prepares function arguments

(2) remember where to restart

(3) branch to function

(8) use result

Inside function (**callee**):

(4) use method parameters

(5) perform body of method

(6) set result

(7) branch back to restart point

(1) + (4) : where to put method arguments/parameters ?

(2) + (7) : where to put restart point ?

(5) : how to use registers safely ?

(6) + (8) : where to put result ?

ARM "function" ?

- In ARM assembly there is no such thing as a function declaration.
- We have labels that we branch to, and return points we branch back to
- We will soon see how the caller sends parameters to the callee (label)
- And how the callee label returns results (in non-void cases)

Caller and callee need to agree

Parameters: `r0 → a, r1 → b, r2 → c`

Result → `r0`

Restart point: `r14` (lr – Link Register)

```
adr lr, restart     // (2)
b find_max          // (3)
restart:
    str r0, [r5]    // (8)
```

method ends with:

```
    mov pc, lr      // (7) we will look at this later
```

Replace `adr` and `b` and `mov`

We can replace the `adr` and `b` instructions with one called `bl` (Branch with Link) which branches to an address stored in a register:

```
bl find_max     //use bl not adr
bx lr           //(7) don't use b
str r0, [r5]    //(8)
```

Simplified Example


```
mov r0, #3           // (1)
mov r1, #44          // (1)
mov r2, #11          // (1)
bl find_max      // (2,3)
str r0, [r5]         // (8)
svc #2
find_max:
. . .                // use r0, r1, r2 (4)
. . .                // to calculate (5)
. . .                // result in r0 (6)
bx lr                // (7)
```

Call(**caller**):

(1) process method arguments

(2) remember where to restart

(3) branch to method

(8) use result

Inside method(**callee**):

(4) use method parameters

(5) perform body of method

(6) set result

(7) branch back to restart point

```c
// this is the "callee"
int find_max(int a, int b, int c) {
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return max;
}
// this is the "caller"
int main() {
  int max = find_max(3,44,11);
  printf("The maximum is %d\n", max);
}
```

How to use registers safely

- Save any other registers used by method (e.g. just after start) and restore at end "**callee saved**"

```
find_max:
    str r4, temp  //save r4
    ...
    //method body can use r4
    ...
    ldr r4, temp   // restore
    bx lr
.data:
temp .word 0
```

What if one label invokes another?


```
find_max:
      str lr, temp1
      bl label_abc
      // ...
      ldr pc, temp1

label_abc:
      str lr, temp2
      // ...
      ldr pc, temp2
.data
      temp1: .word 0
      temp2: .word 0
```

The syntax of the `str/ldr` instructions are a syntactical simplification (as mentioned previously)

Does this scale up?

Recursion

- a method can call any method, including itself
- → maybe one "temp" per register won't be enough

Efficiency

- space occupied by all those "temp"s mostly unused at any given moment during run-time
- We need another way to save and restore registers

Nesting Calls

Method calls and returns are nested (like brackets)

- so register saves and restores are also nested
- so what we need is something that works the same way
- Also notice the order in which we save and restore in nested calls
  - `str r0 …`
  - `str r1 …`
  - `ldr r1 …`
  - `ldr r0 …`

Introducing the stack

full descending stack

The Stack

A stack is like a very heavy pile of books:

- you can take anything off the top of the pile (“pop”)
- you can add anything to the top of the pile (“push”)
- you can’t move anything into/out of the pile except at the top!

This is simple, and exactly what we need for method calls: **LIFO = last-in, first-out**

Full / Descending

- **Full** : The stack pointer points to the last item pushed onto the stack (not an empty slot).
- **Descending** : The stack grows downward in memory (toward lower addresses).
- Stack and heap grow towards each other
  - This arrangement was devised to simplify memory management
  - It allows room for each of the resources to expand / contract over process running time

Using a Stack on ARM

On most computers:

- stacks start at a large address,
- which decreases as the stack grows

ARM has:

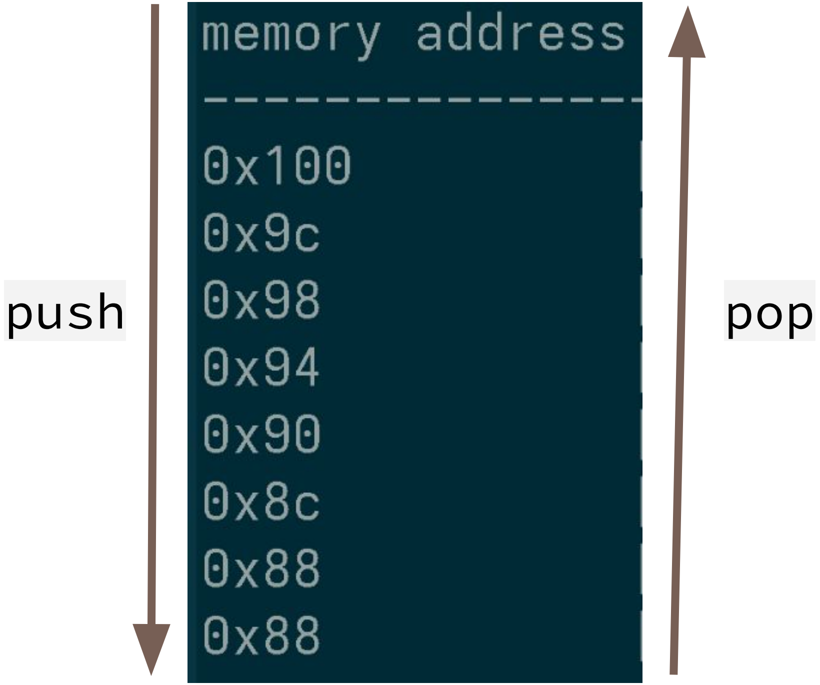
- stack-pointer register (`sp, r13`)
- 4 different ways of using a stack!

We will use commonest version:

- `sp` = address of top word on stack (stack-top)
- decrease `sp` to push, increase to pop

Stack Indexing Modes

- **Push**
  - `str r0, [sp,#-4]!` (pre-indexed)
- **Pop**
  - `ldr r0, [sp],#4` (post-indexed)



Preserved (Callee-Saved) Registers

- Preserved registers (also called callee-saved registers) are those that a label (callee) must preserve for the caller,
  - the function must restore their original values before returning
- `r4-r8, r10, r11`: General-purpose registers that a function must save and restore if it modifies them.
- `sp` (`r13`): The stack pointer must be preserved in the sense that it must be restored to its original value before returning, ensuring the stack is properly aligned and balanced.

Preserving `r0,r1,r2,r3`?

- If the **caller** needs to preserve `r0-r3` it must do so before invoking the label
- The **callee** is not under any obligation to save `r0-r3`
- **Caller**-saved registers before a label call should use the stack too, eg:

```
str r0, [ sp, #-4 ]! //save r0
bl find_max
ldr r0, [ sp ], #4   //restore r0
```

Passing Parameters to a label

- As `r0,r1,r2,r3` are not preserved registers, the caller should not expect them to be preserved
- They should be used to pass parameters to the function

```
mov r0, #3
mov r1, #44
mov r2, #11
bl find_max

find_max:
  // can access / over-write r0,..,r3
```

What if one method calls another? (again)


```
find_max:
  str lr, [sp,#-4]! //"push"
  bl label_abc
  //some code
  ldr lr, [sp],#4   //"pop"
  bx lr

label_abc:
  str lr, [sp,#-4]! //"push"
  bl label_xyz
  //some code
  ldr lr, [sp],#4   //"pop"
  bx lr

label_xyz:
  str lr, [sp,#-4]! //"push"
  // some code
  ldr lr, [sp],#4   //"pop"
  bx lr
```

You must save the link register on the callee stack (frame) when the label begins

Brining it together


```
mov r0, #3  // prepare param 1
mov r1, #44 // prepare param 2
mov r2, #11 // prepare param 3
bl find_max // invoke using bl => saves the address of svc 2 into lr
svc #2
find_max:
     str lr, [ sp,#-4 ]! // preserve the link registrer
     str r4, [ sp,#-4 ]! // preserve r4 if we plan to use it
     mov r0, #44         // prepare algorithm return value (cheat!)
     ldr r4, [ sp ], #4  // restore r4
     ldr lr, [ sp ], #4  // restore lr
     bx lr               // return
```

`bx lr` versus `mov pc, lr`

- You might sometimes see `mov pc, lr` used interchangeably or in place of `bx lr`
- They are not the same thing
- `bx` is a branching instruction, *so it can be optimized via branch prediction technology within the microarchitecture*.
- Not so with `mov pc, lr`… so best avoid this, although in practice the performance may be the same.

Recursion

A function that calls itself

Recursion

When a function calls itself this pattern is called **recursion**

- We sometimes use recursive implementations of methods
  - because they can lead to compact, elegant code
  - code that may be easier to understand than a corresponding implementation that does not use recursion
- Recursion tends to appear more from the mathematical / analysis viewpoint rather than in commercial software development
- Anything that can be done using recursion, can be done without using recursion
  - By redesigning the algorithm

Stack Overflow Example

```
// this is a terrible idea
int find_max(int a, int b, int c) {
    int max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    return find_max(a,b,c);
}
```

```
Segmentation fault        (core dumped) ./a.out
```

```
real-time non-blocking time  (microseconds, -R) 200000
core file size              (blocks, -c) unlimited
data seg size               (kbytes, -d) unlimited
scheduling priority                 (-e) 0
file size                   (blocks, -f) unlimited
pending signals                     (-i) 124653
max locked memory           (kbytes, -l) 8192
max memory size             (kbytes, -m) unlimited
open files                          (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues         (bytes, -q) 819200
real-time priority                  (-r) 0
stack size                  (kbytes, -s) 8192
cpu time                   (seconds, -t) unlimited
max user processes                  (-u) 124653
virtual memory              (kbytes, -v) unlimited
file locks                          (-x) unlimited
```

Risks of recursion

- Recursion does not add any computational efficiency or performance advantage to your software
- In fact, it adds **risk** which can destabilise your code
  - As seen above, the runtime does not warn us about the dangers in the code
- Recursive calls have **hard limits** on the stack depth - which is the main constraint that limits its usefulness

Stack Variables

Making and reclaiming space

From the last time

- Why do we use a stack rather than fixed memory locations?
- A stack is implemented using `sp` and pre-/post-indexing instructions.
  - You should be able to give the code for the **push** and **pop** operations
  - and explain the value in `sp` after each operation.

Problems

Last lecture, we said:

e.g. `r0` = 1st parameter, `r1` = 2nd, `r2` = 3rd etc.

- What if we have a lot of parameters?
  - Use the stack if more than four parameters are required (since `r0-r3` can only hold up to four 32-bit values).
- Alternative: push arguments onto stack during call
- What if we want to declare some stack local variables?

Local variables

Local to the stack

Variable Lifetime

Most variables declared inside a function have the same lifetime as the function itself:

- when the function is called, its variables are created
- when the function returns, its variables are destroyed

(there are exceptions, but we will ignore them for now)

We could use the stack for this

The most space-efficient way to do this is:

- at the method start, "**push**" extra space for its variables
- at the method end, "**pop**" the extra space

```
sub sp, sp, #bytes of variables ; create space
...
add sp, sp, #bytes of variables ; destroy space
```

Making space for local variables


```
// lets make space for
// int max = a;

find_max:
  // after saving lr and r4
  str r0, [ sp, #-4 ]!
```

Of course, we need to restore the stack pointer to the right place at the end of `find_max`

Create space for 4 local variables

We can create room for any number of uninitialized variables by just moving the stack pointer down, for example, four local variable `int`'s:

```
find_max:
  // save lr and r4
  sub sp, sp, #16
  // do the algorithm
  // restore the 4 ints
 add sp, sp, #16
```

Reset the `sp` and return

- Ultimately, at the end of every label, the sp must be reset to the value it had before the label began
- This can be done its lots of ways:
  - Pairwise, `str … [sp,#-4], str … [sp,#-4], str … [sp,#-4]`, followed later by `ldr … [sp], #4, ldr … [sp], #4, ldr … [sp], #4`
  - Or, a bulk reclaim at the end: `add sp, sp, #12`
  - Just so long as the `sp` is moved down and up the same number of bytes.

Bulk Register Load and Store

ARM has “Store/Load Multiple” instructions to move several register contents to/from memory with a single instruction

- May apply to any subset of 1-16 registers
- More efficient than using many instructions
- Limited in addressing options: only a single base register

Store Multiple Full Descending


```
// push several registers
stmfd sp!, { r0, r1, r3-r6 }
```

Store Multiple, Full Descending: STMFD (Store Multiple, Full Descending): stores multiple registers to the stack in a full descending stack model, where `sp` points to the last item pushed and the stack grows downward (toward lower addresses).

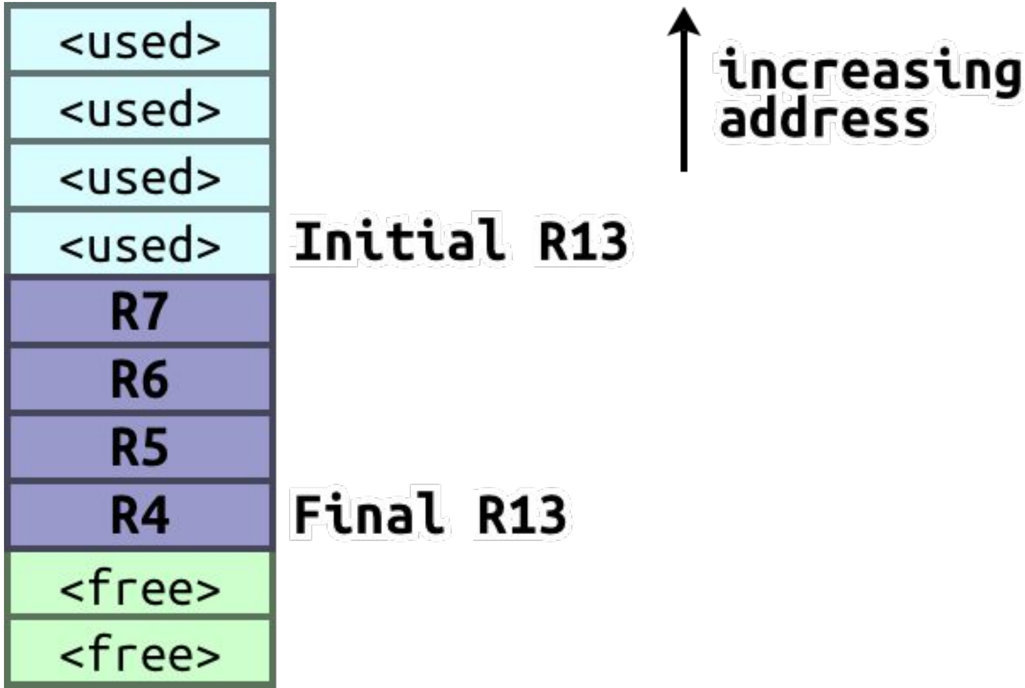
Load Multiple Full Descending


```
// pop several registers
ldmfd sp!, { r0, r1, r3-r6 }
```

LDMFD (Load Multiple, Full Descending) loads multiple registers from the stack in a full descending stack model, where the stack pointer (`sp`) points to the last item pushed and the stack grows downward.

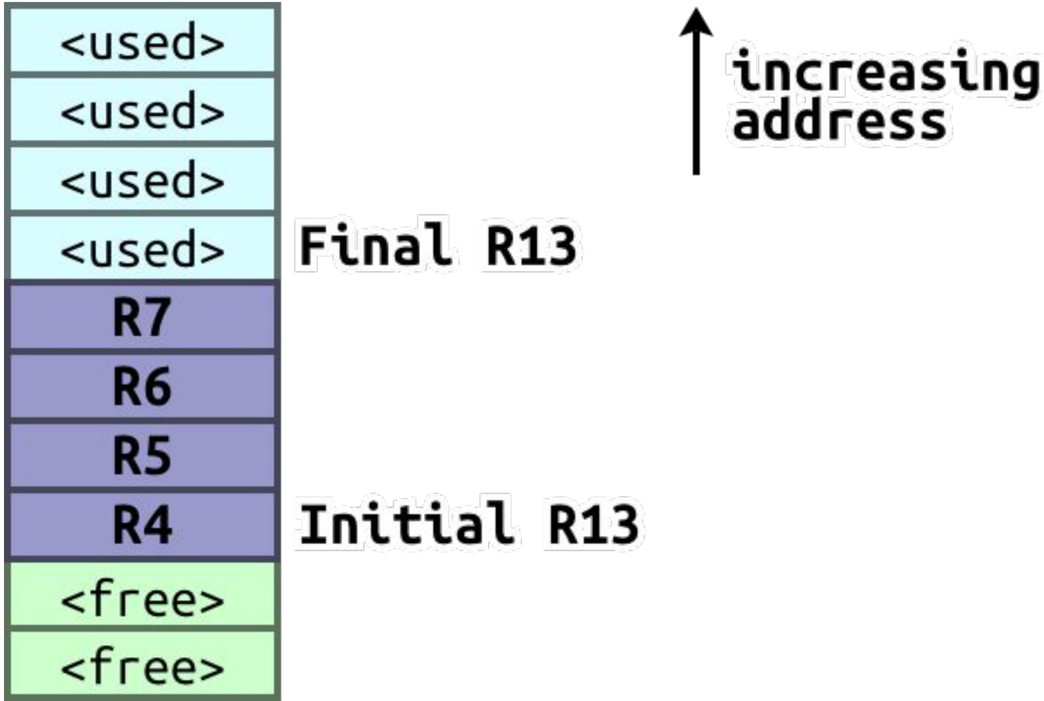
Example Stack Entry

`stmfd r13!, {r4-r7}` – pushes `r4,r5,r6,r7` onto the stack.



Example Stack Exit

`ldmfd r13!, {r4-r7}` – pops `r4,r5,r6,r7` from the stack.



Uses

Two main uses:

- Saving/restoring registers to create working space
- Copying blocks of memory around

(We are only interested in the first of these.)

Note, The `sp` and `pc` can be in the list in ARM instructions, but not in Thumb instructions.

Saving Registers

The assembler has instructions which save typing:

`push { r0, r1, r3-r6 }` // push several registers

`pop  { r0, r1, r3-r6 }` // pop several registers

Which builds a stack in the ‘traditional’ way.

- The listed register contents are moved (6 in this case)
- The memory used is consecutive; the lowest numbered register always corresponds to the lowest address. (i.e. the top of the stack)

Save and Restore

- If an expression is really complicated, we may need to save/restore registers to evaluate it.
- Can save registers at start of method – "callee saved" (as in previous lecture) or before call – "caller saved"

Normal Pattern :

- Caller saves when "sending" parameters
- Callee saves when preparing the return branch point `lr` register

ARM Procedure Call Standard (APCS)

- A standard on how to use registers in real programs
- `r0-r3`: parameter/result passing (extra arguments are stacked) anyone can use, but not saved across call (caller saved)
- `r4-r8, r10, r11`: temporaries/locals (callee saved)
- `r9, r12`: temporaries (caller saved)
- `sp, lr, pc`: special purpose registers

Stack Frame Example


```
memory address | content             | description
---------------|---------------------|------------------------------
0x100         | (unused)            | initial sp, not used after push
0x9c          | parameter           | 5th parameter (pushed by caller)
0x98          | lr                  | saved return address
0x94          | r5                  | callee-saved register
0x90          | r4                  | callee-saved register
0x8c          | local variable 1    | function-local data
0x88          | local variable 2    | function-local data
0x88          | [sp points here]    | final sp after prologue
```