Calling Conventions and the Stack

CS2253

Owen Kaser, UNBSJ

Overview

- Stacks and the Load/Store Multiple Instructions
- Subroutines
- ARM Application Procedure Call Standard
  - Code Linkage Mechanism
  - Parameter Passing
  - Caller- and Callee-Save Registers
- See Chapter 13 of textbook

Stack in Memory

- Recall from CS1083 (or CS2383) that one can use an array to store a stack's data. Also need an integer array index, called TOP.
- Push(value) → Data[++TOP] = value;
- Pop → return Data[TOP--];
- TOP was initialized to -1 and always points to the value to be popped. Stack grows up.
- ARM folk call this a "full ascending" stack

Full-Descending Stack

- In low-level programming, a Full-Descending stack is more common. ARM ABI requires it.
- TOS is initialized to max_valid_index +1
- Push(value) → Data[--TOP] = value
- Pop → return Data[TOP++]
- We decrement before on push and increment after on pop.
- DB and IA.

Empty Stacks

- To ARM, an empty stack is where the top-of-stack pointer indicates where the next push will go.
- Push(value) → Data[TOP--]
- Pop → return Data[++TOP]
- Decrement After (DA) and Increment Before(IB) for an “empty descending” stack.
- Empty ascending stacks are also possible.

ARM Assembly Push

- Use a form of the store multiple (STM) instruction for push, and a form of load multiple (LDM) for pop.
- Top-of-Stack is usually the R13 register (alias SP)
- Push(value in R5) → STMDB SP!, {R5}
- Pop (from stack to R5) → LDMIA SP!, {R5}
- This is for a Full-Descending stack, so you can use STMFD and LDMFD if desired (maybe).
- Crossware does not support PUSH and POP (textbook 13.2.2)

An Un-RISCy Quirk

- We'll soon see that programmers often want to push (or pop) a bunch of registers to the stack.
- RISC approach: need 6 instructions for 6 pushes
- ARM: have a single complex instruction STM to push several registers. Requires multiple clock cycles.
- STMDB SP! , {R12, R3-R5, R7, R8}
- Pushed from smallest register to largest, and the ! ensures that SP gets changed.
- LDMIA SP!, {R8, R7, R12, R3-R5} will restore them.

LDM/STM encoding

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	S	W	L	Rn	register list

-
-
- P=1 means Before
- U=1 means Upward
- W means a ! was used
- L=1 means LDM instead of STM
- S=1 means some weird behaviour that depends on processor mode; see technical docs
- The register list is a bit vector; bit k means Rk is included in the set of registers loaded or stored.

Full Descending Stack

```
StackBottom   SPACE 1000
StackTop  ; label to mark address beyond top
…..
                    MOV SP, #StackTop  ; initialized
```

- Now we can push and pop, up to a limit of 250 32-bit values.

Why Push and Pop?

- The stack is convenient for saving registers (push them to save; later pop them to restore).
- Let's suppose you have a some valuable info in R3, R4. You are about to enter into some code that trashes R3 and R4. And after you exit from that code, you have some use for the old R3 and R4.

-
  ```
    STMDB SP!,{R3,R4} ; save R3 and R4
     … code that trashes R3 and R4...
     LDMIA SP!,{R3,R4}  ; restore saved registers
     … code that uses R3 and R4...
  ```

Subroutines

- A subroutine is essentially a method (that doesn't belong to an object).
- You call a subroutine and return from it.
- A reentrant subroutine can be paused while running, and while paused, another copy of it can start running. When finished, the paused subroutine is allowed to continue...and no problems arise.
- Your most familiar reentrant situation is with recursion. (Later: we will see “interrupts”)
- Naive coding of subroutines leads to non-reentrancy. Fancy coding with stack frames will give reentrancy.

Getting Into/ Out of Subroutines

- If you have some code you wish to call, use a BL (branch-and-link) instruction.
- Like B, it changes the program counter so you jump somewhere (the first instruction of the subroutine.)
- But R14 (alias LR, the link register) is automatically set to the return address of PC-4 (the address of the instruction immediately after the BL instruction)
- at the end of the subroutine, arrange for the return address to go into PC.

Example Subroutine

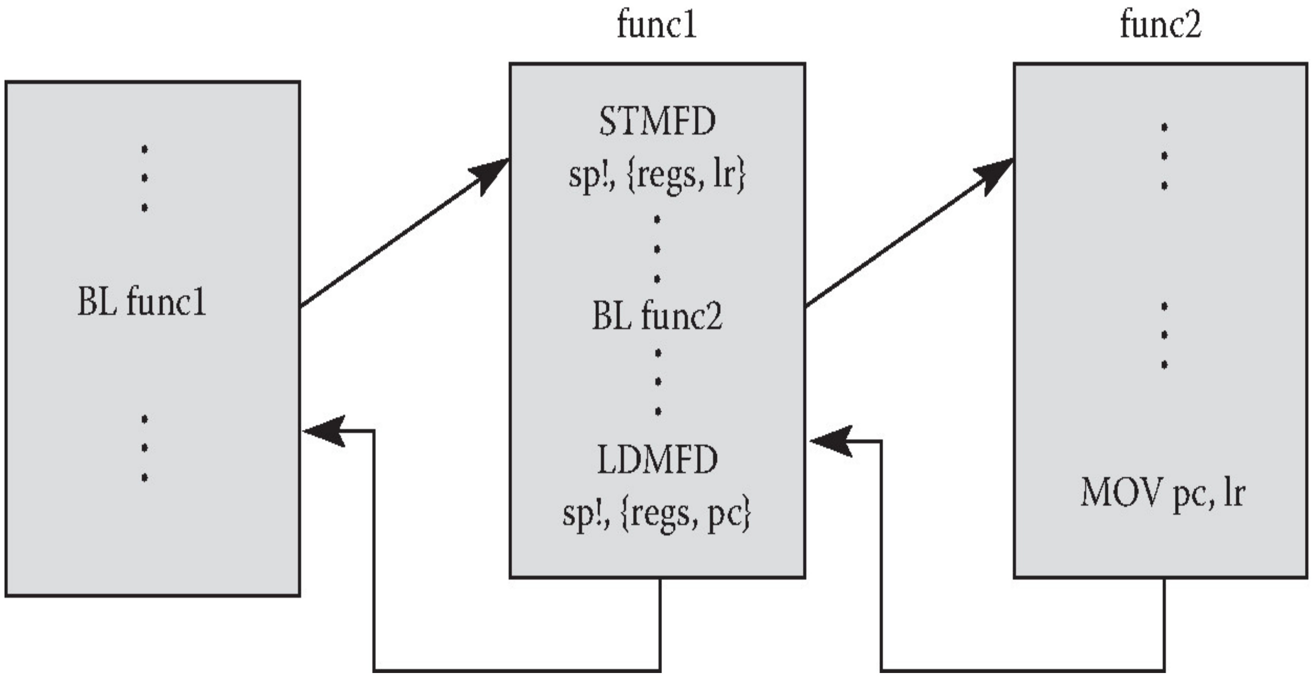
```
; this subroutine multiplies the value in r0
; by 10 and returns result in r1.  Trashes r2.
times10   mov r2, r0, lsl #3  ; r2 = 8*r0
          add  r1, r2, r0, lsl #1 ; r1 = r2 + 2*r0
          mov pc, lr  ; return
…...
           mov r0, #5
           bl times10 ; invoke subroutine
 ….
```

Passing Parameters

- Our example passed an input parameter by using a register.
- And it used a register to pass back a return value.
- This is a common approach.
- Another alternative is to pass parameters on the stack (caller pushes them).
- The callee then accesses them in memory.
- When subroutine is finished, either the caller or callee must pop off the parameters so stack doesn't overflow.

Subroutines Calling Subroutines

- Commonly, one method calls another.
- Say main() calls func1(), which then calls func2(). Don't lose the original return address!



Saving/Restoring Registers: 1

- Suppose you are going to call a routine that's documented to trash R4 and R5. Currently, you have something valuable in R4 that you will need later. But the value in R5 doesn't matter anymore to you.
- Caller-save scheme:

    push just the value in R4

    call the routine, using BL

    pop into R4

Saving/Restoring Registers: 2

- Suppose you're writing a subroutine. You've been told you're not allowed to trash R4, but you are allowed to trash R5.
- You really need to use both R4 and R5
- Callee-save scheme:

```
mysub  STMDB SP!,{R4}  ; save R4
       ...code trashing R4 and R5
       LDMIA SP!,{R4}   ; restore R4
       MOV  PC, LR  ; return
```

Interoperability

- I want to be able to call your subroutines, and you want to be able to call mine.
- So I need to know how you expect parameters passed in, and how return values should be handled.
- I'd like to know whether I can trust you not to trash my registers' values.
- I'd like to know if there are any registers that I can trash, without having to save/restore them.
- We need some rules...

AAPCS (Textbook 13.5)

- The ARM Application Procedure Call Standard is our binding contract. If we both follow it, our code will interoperate smoothly.
- It's an example of a calling convention.
- 3 parts to the contract
  - obligations of caller to set things up for callee
  - obligations of callee not to (permanently) trash parts of the state of the caller
  - rights of the callee to trash other parts of the caller's state.

AAPCS: Parameter Passing

- R0 to R3 have the parameter values. Caller places them there.
- Callee places return value(s) in R0 and R1
- Caller is otherwise free to trash R0 to R3 (so they are essentially “caller-save registers”).
- Subroutines with more than 4 parameters will have the caller push the extra parameter values on the stack. [not sure who is responsible for their cleanup: guess it is the caller, who made the mess.]

AAPCS:Callee-save

- The callee must preserve R4-R11,SP (so they are “callee-save registers”)
- R4 to R11 typically contain local variables of the caller. And the caller expects the SP to come back unchanged.
- It is very normal to do all the callee-save pushing as a single STMDB instruction at the very start of the subroutine. This said to create a stack frame.
- And the corresponding popping is a single LDMIA instruction that is at the end of the subroutine.

AAPCS: Status flags

- You cannot expect the status flags to be preserved during a call.
- So essentially, they would be caller-save.
- Except that it's rare to need an earlier-computed status flag after the subroutine returns.

AAPCS: Link register

- The caller will have given us an R14 value that tells us where to return to when finished.
- Any BL that we do will destroy it. Very bad.
  - Case 1: we are a “leaf procedure” (ie, make no calls). Just don't use R14 for anything. Finish with

    ```
    MOV PC, R14
    ```
  - Case 2: we are a non-leaf procedure (and can potentially make a call). Push R14 with the callee-save registers at the start of the subroutine.

    And pop it (into PC) with the LDMIA instruction when finishing.
  - This works because of the order in which LDM/STM stores regs.

AAPCS: R12

- Textbook indicates that R12 is somehow used by the linker at the point when the caller invokes the callee. Mysterious.
- You are allowed to trash it, but you must expect the calling process to potentially trash it.
- I.e., R12 is a caller-save register.

Warning

- Deviate from AAPCS at your own risk.
- First, your code won't otherwise interoperate.
- Second, DIY approaches to these things tend to fail in horrible and utterly puzzling ways, especially if recursion is involved. I don't want to be responsible for your soul-destroying all-night debugging session that makes you switch to a BBA degree. (The world needs you in CS: follow the AAPCS...)

Let's Code the Recursive Fibonacci

-
  ```
  int fib(int n) {
       int temp, temp2;
       if (n < 2) return n;
       else  {
          temp = fib(n-1);
          temp2 = fib(n-2);
          return temp+temp2;
                 }
  }
  ```