Digital Systems: Problem sheet 2

Mike Spivey, Hilary Term, 2019

This sheet again probes details of the encoding of ARM instructions as binary machine code. The details are interesting because they reflect which instructions are most useful in programming. Undergraduates have no need to memorise everything, and in fact suitable compact reference material will be provided in the exam.

1 Thumb provides encodings for the following instructions, with `n` a small constant. In each case, suggest a use for the instruction in implementing the constructs of a high-level language.

```
ldr r1, [pc, #n]        add r1, pc, #n
ldr r1, [r2, r3]        add r1, sp, #n
ldr r1, [r2, #n]        add sp, sp, #n
ldr r1, [sp, #n]
```

2 There are two instructions, `ldrh` and `ldrsh`, that load a 16-bit quantity from memory and put it in a 32-bit register, but only one instruction `strh` that stores into a 16-bit memory location. What is the difference between the two load instructions, and why is only one store instruction needed? Thumb provides no encoding for the form `ldrsh r1, [r2, #n]`; what equivalent sequence of instructions can be used instead?

3 We have used the instruction `ldr r1, [sp, #n]` to access local variables stored in the stack frame of the current procedure. In the Thumb encoding, `n` is constrained to be a multiple of 4 that is less than 1024 bytes. What can be done to address variables in a stack frame that is bigger than this?

4 If the `push` and `pop` instructions did not exist, what sequences of instructions could be used to replace them in the prologue and epilogue of a subroutine?

5 In the program shown in Figure 1, function `baz` has 64 bytes of space for local variables, including an integer `j` at offset 60 from the stack pointer, and an array `b` of 10 integers at offset 4. There is a global integer variable `i`,


```
baz:
    push {r4-r7, lr}
    sub sp, #64
    ...
    add sp, #64
    pop {r4-r7, pc}

    .bss
    .align 2
i:
    .space 4
    ...
    .align 2
a:
    .space 40
```

Figure 1: *Skeleton for Exercise 5*

and a global array `a`, also containing 10 integers. Write assembly language code that might appear in the body of `baz`, equivalent to the C statement

```
a[i+j] = 3 * b[i+j];
```

6 Use the skeleton shown in Figure 2 to write a subroutine `foo(n, k)` that computes the binomial coefficient $\binom{n}{k}$ by calculating successive rows of Pascal's triangle in an array, updating it in place according to the recurrence,

$$\binom{a}{b} = \binom{a-1}{b-1} + \binom{a-1}{b},$$

with suitable boundary conditions. Give an outline of your solution in a high-level language, then (explaining the correspondence between variables in your outline and registers in your code) show how the high-level program can be realised in assembly language. You may assume $0 \le k \le n < 256$.

The last part of the skeleton contains instructions to the assembler to reserve 256 words of RAM as part of the 'uninitialised data' segment of the program. (Actually, it is initialised to zeroes, but will no longer contain zeroes the second time the subroutine is called.) The `ldr` instruction at the top of the subroutine sets `r4` to the address of this block of storage; it works well to leave the address in the register for the entire duration of the subroutine. Also shown are code to initialise *row*[0] to 1 and code to return *row*[*k*] at the end.

7 Describe the changes needed to make each of the following modifications to the program `lab1-asm/catalan.s` (shown also in Figure 3) for computing Catalan numbers C_n.

(a) Instead of storing the values of C_k in a statically allocated array, allocate space for the array in the stack frame of the function.

(b) Instead of an inner loop that counts up from 0 to k, write one that counts down from k to 0; exploit the fact that the loop body always executes

```
    .thumb_func
foo:
    push {r4-r7, lr}        @ Save all registers
    ldr r4, =row            @ Set r4 to base of array

    @ r4 points to the array throughout the subroutine

    movs r3, #1             @ row[0] = 1
    str r3, [r4, #0]

    ...

    lsls r2, r1, #2         @ return row[k]
    ldr r0, [r4, r2]
    pop {r4-r7, pc}         @ Restore regs and return

@ Statically allocate 256 words for the array
    .bss
    .align 2
row:
    .space 1024
```

Figure 2: *Skeleton for Exercise 6*

at least once to move the test to the end of the loop body, and elide the `cmp` instruction.

(c) Instead of storing the values of n, k and j in registers, store $4 * n$, $4 * k$ and $4 * j$ instead, adjusting them by 4 at a time. This should allow you to eliminate almost all of the shift instructions from the program.

(d) Instead of using the `mul` instruction to multiply C_j and C_{k-j}, call a subroutine to do the multiplication. This will make the program slower, but allow it to run on chips with no hardware multiplier.

8 Unusually, the subroutine call instruction `bl` on the Cortex-M0 occupies 32 bits instead of 16, so that it can contain 24 bits of immediate data, enough to address any even address in a range of ±16MB relative the the `pc`. About half the bits of the immediate data are found in the first 16-bit instruction word, and about half in the second word. Ben Lee User (a student) suggests that the `bl` instruction could be implemented by adding a hidden register to the machine. The first word of a `bl` instruction would store half the address bits in the hidden register: then the second word, executed in the next cycle, would perform the jump, combining the hidden register with its own address bits. Would this scheme work on a version of the architecture that supported interrupts? What special provisions would be needed to allow it to work?

Early versions of the Thumb instruction set were in fact implemented in this way, but using `lr` in place of a hidden register. Why was this a better idea than introducing a new register for the purpose? What risks does it entail?

```
    .global foo
    .thumb_func
foo:
@ Compute Catalan(n) from the defining recurrence
@ ... using a static array and loops

    push {r4-r7, lr}        @ Save registers

@@ r0 = n, r3 = t, r4 = row, r5 = k, r6 = j

    movs r5, 0              @ k = 0
    ldr r4, =row
    movs r1, 1
    str r1, [r4]            @ row[0] = 1

outer:
    cmp r5, r0              @ while (k < n)
    bge done

    movs r6, #0             @ j = 0
    movs r3, #0             @ t = 0

inner:
    cmp r6, r5              @ while (j <= k)
    bgt indone

    lsls r1, r6, #2         @ put row[j] in r2
    ldr r2, [r4, r1]
    subs r1, r5, r6         @ put row[k-j] in r1
    lsls r1, r1, #2
    ldr r1, [r4, r1]
    muls r2, r2, r1         @ multiply
    adds r3, r3, r2         @ add to t
    adds r6, r6, #1         @ j++
    b inner

indone:
    adds r5, r5, #1         @ k++
    lsls r1, r5, #2         @ row[k] = t
    str r3, [r4, r1]
    b outer

done:
    lsls r1, r0, 2          @ return row[n]
    ldr r0, [r4, r1]
    pop {r4-r7, pc}         @ restore and return

@ Statically allocate 256 words of storage
    .bss
    .align 2
row:
    .space 1024
```

Figure 3: *Program for Catalan numbers*