Digital Systems: Problem sheet 2

Mike Spivey, Hilary Term, 2022

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]
```

In native code, these instructions have a uniform encoding that allows any register to be used for any purpose. In Thumb code, only some forms of the instructions have been given an encoding, and the designers have deliberately chosen to spend encoding space on instructions that are useful for some purpose: but what?

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*`]`, where *n* is a numeric offset, 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?


```
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*

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`, 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 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?

7 Write in assembly language an implementation of the function `toupper()` with heading

```
void toupper(char *s);
```

It should modify the null-terminated ASCII string `s` in place, changing each lower-case letter into the corresponding upper-case letter, and leaving other characters unchanged.

According to C conventions, the parameter `s` *that is passed to the* `toupper` *function (and arrives in register* `r0`*) is the address of the first character of the string. Subsequent characters are found at addresses* `s+1`, `s+2`, `...`, *up to a terminating character with the numeric value zero, written* `'\0'` *in C. Note that this null character is different from the digit* `'0'`.

8 The factorial function has long been used as an example of recursive programming: unwisely so, because it can be more simply computed with a loop. Assuming a machine that has a multiply instruction, implement a recursive version of factorial in assembly language, and compare it for speed with a version implemented with a loop.

What programming example does make a good, simple example of the power of recursion?

9 The lecture notes suggest the following code for testing whether a button on the micro:bit is pressed.

```
x = GPIO_IN;
if (GET_BIT(x, BUTTON_A) == 0 || GET_BIT(x, BUTTON_B) == 0) {
   // A button is pressed
}
```

Then names `GET_BIT`, `BUTTON_A`, etc., are defined as macros in `hardware.h`. Find out the definitions of these macros, and write a C expression that more directly represents the operations requested for the test. Then use a disassembler to look at the corresponding object code produced by the C compiler, and write C source that represents the operations actually performed at runtime. Comment whether it is necessary to focus much on the low-level efficiency of the code we write.