Digital Systems: Collected problems

Mike Spivey, Hilary and Trinity Terms, 2020

1 Machine-level programming

Questions on this sheet probe details of ARM Cortex-M0 assembly language, and also of its encoding as binary machine code. It can't be emphasised enough that, though a lot of this detail is needed for the experience of understanding how a particular computer works (which every Computer Science student should enjoy at some stage), it isn't detail that is worth memorising in the medium term. Id est, *it won't be on the exam.*

1.1 Find as many single Thumb instructions as you can that have the effect of setting register r0 to zero, and as many as you can that copy register r1 to register r0. In each case, the instruction may or may not set the condition codes: say which.

To answer this exercise, use the ARM Architecture Reference Manual (linked from the micro:bit page on the wiki) to look up the details of some of the instructions shown in red, orange, yellow or green in tables [A] and [B] of the Rainbow Chart.

1.2 The following listing shows a disassembly of the simple multiplication routine from the lecture.

```
          func:
c0: 2200      movs    r2, #0
          loop:
c2: 2900      cmp     r1, #0
c4: d002      beq     cc <done>
c6: 3901      subs    r1, #1
c8: 1812      adds    r2, r2, r0
ca: e7fa      b       c2 <loop>
          done:
cc: 0010      movs    r0, r2
ce: 4770      bx      lr
```

(I've edited it a bit to remove distracting details.) Decode some of the instructions by hand, and in particular explain the displacements that appear

Copyright © 2018–23 J. M. Spivey

in the `beq` and `b` instructions.

Again look up the instructions in the ARM Architecture Reference Manual, and use the information about encodings given there to decypher some of the hexadecimal numbers shown in the listing.

1.3 As it stands, the multiplication routine shown in Exercise 1.2 has a loop that contains 5 instructions, and takes 7 cycles for each iteration but the last: three cycles for the branch back to the start, and one cycle for each of the other instructions, including the untaken conditional branch. Try to rewrite the loop so that it contains fewer instructions. *A good solution has 3 instructions in the loop, taking 5 cycles per iteration, but it is possible to go further using 'loop unrolling' and make a still faster routine.*

1.4 Note that conditional branch instructions in Thumb code, such as `beq`, have a displacement field of limited size, but the unconditional branch `b` has a bigger displacement field. Show how a two-instruction sequence can be used to simulate conditional branches with a bigger range than can be achieved with a single `beq`. What is the penalty in execution time for doing this? The branch-and-link instruction `bl` has a still larger displacement field. Can it be used in a similar way to simulate even longer conditional branches?

1.5 Show with examples why different instructions `blt` and `blo` are needed following comparison of signed and unsigned numbers. Find out the conditions under which each of these branches is taken, and explain why the result is correct in each case.

1.6 Write (in some high-level language – C, Scala, or Python if you must) a routine for unsigned integer division, using the naïve algorithm based on repeated subtraction. Code the result in assembly language.

To answer this problem well, you must: formulate precisely the problem to be solved; give a clear algorithm for the problem and justify its correctness; code the algorithm in commented assembly language in an efficient way, considering the possibility of overflow; comment on errors that should be detected.

1.7 Repeat the previous exercise using a better algorithm. Fancier versions of the ARM have an instruction `clz r1, r2` that sets `r1` to the number of leading zeroes in the binary number in `r2`; what use is this instruction in writing a division routine?

2 Programming with memory

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.

2.1 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?

2.2 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?

2.3 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?

2.4 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?

2.5 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];
```

2.6 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?

```
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 2.5*

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?

2.7 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'`.

2.8 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?

2.9 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.

3 Interrupts

3.1 Figure 2 shows the interrupt-based driver for serial output that we studied in the lecture. The functions `intr_disable()` and `intr_enable()` use the instructions `cpsid i` and `cpsie i` to change the PRIMASK bit in the CPU that allows interrupts. The function `pause()` uses the `wfe` instruction to halt the CPU until an interrupt occurs.

(a) What relationship is maintained among the values of `bufin`, `bufout` and `bufcnt`? Try to find an alternative implementation with only two integer variables.

(b) Why is it necessary in `serial_putc` to disable interrupts before checking `txidle`?

(c) Why must interrupts be disabled during the command `bufcnt++`? *Hint: consider the assembly-language equivalent of the command.*

(d) Study the difference between the `wfe` and `wfi` instructions, and explain why `wfe` is needed in this program.

(e) If it was important to have interrupts disabled for the shortest possible time, how could the code of `serial_putc` be modified so as to remain safe?

3.2 Program `heart-intr` embeds all the code needed to multiplex the LED display in the handler for the timer interrupt, allowing the 'main program' to be devoted to other tasks – but it can display only a static image. Show how to enhance it to show an animated heartbeat, like the one in Lab 2. What are the limitations of this approach?

3.3 The NRF51822 has a hardware random number generator. When it is appropriately configured, it periodically generates an interrupt that calls `rng_handler`, and an eight-bit random number can then be retrieved from the device register RNG_VALUE before resetting the event flag RNG_VALRDY to zero. Design an interrupt-based driver for the random number generator; provide two functions

```
unsigned randint(void);
unsigned roll(void);
```

such that `randint()` returns a random four-byte value each time it is called, and `roll()` returns a random integer between 1 and 6, with each outcome having precisely equal probability. Arrange a suitable buffering scheme so

```
void uart_handler(void) {
    if (UART_TXDRDY) {
        UART_TXDRDY = 0;
        if (bufcnt == 0)
            txidle = 1;
        else {
            UART_TXD = txbuf[bufout]; bufcnt--;
            bufout = (bufout+1) % NBUF;
        }
    }
}

/* serial_putc -- send output character */
void serial_putc(char ch) {
    while (bufcnt == NBUF) pause();

    intr_disable();
    if (txidle) {
        UART_TXD = ch;
        txidle = 0;
    } else {
        txbuf[bufin] = ch; bufcnt++;
        bufin = (bufin+1) % NBUF;
    }
    intr_enable();
}
```

Figure 2: *Code for interrupt-driven serial output*

that the caller of these functions does not have to wait if it calls them infrequently enough, and introduce a subroutine whenever doing so avoids repeating the same code in more than one place. What should happen if random bytes are generated by the hardware faster than the program is consuming them?

3.4 A cut-down version of the Cortex-M0 saves only the program counter and the processor status register to the stack before invoking the interrupt handler. It does not set `lr` to a magic value, but leaves it unchanged, and returning from the interrupt requires a special instruction `rti` (with encoding 0xbfd0). Design an assembly-language adapter that allows a C function such as `uart_handler` in Exercise 3.1 to be installed as an interrupt handler.

3.5 In the buffer overrun attack of Lecture 7, a long sequence of input was able to overflow the array in the frame of `init()` and replace its return address. This worked because the array was stored at a lower address than the register save area in `init`'s stack frame. Would buffer overrun attacks be prevented if the stack were to grow upwards in memory instead of downwards?

```
#include "microbian.h"

static volatile int r = 0;

void proc1(int n) {
    for (int i = 0; i < 10; i++)
        printf("r = %d\n", r);
}

void proc2(int n) {
    while (r < 100000) r++;
}

void init(void) {
    serial_init();
    start("Proc1", proc1, 0, STACK);
    start("Proc2", proc2, 0, STACK);
}
```

Figure 3: *Program for Exercise 4.4*

4 Operating systems

4.1 In micro:bian, what happens if the function that forms the body of a process returns? What happens if all such functions return?

4.2 In micro:bian, what happens if a process tries to send a message to itself?

4.3 A micro:bian process can have a list of processes waiting to send to it. Imagine a directed graph in which the nodes are processes, and there is an arrow to each process from all the other processes that are waiting to send to it. What happens if there is a cycle in this graph? How could such cycles be detected?

4.4 The program shown in Figure 3 contains two processes that share a variable `r`: one process increments `r` from 0 to 100,000 while the other prints the value of `r` ten times. What might we see as output from this program? Would it make a difference if the calls to `start` in `init` were re-ordered?

4.5 Consider a situation where a process is continuously sending characters to the serial driver. The processor time for a typical context switch to send and receive a message is about 20 μsec.

(a) How many context switches happen for each character sent?

(b) How much can the UART speed be increased before context switching time occupies a substantial fraction of the time that the UART takes to send a character?

(c) Suggest a way of reducing the number of context switches per character output.

```
#include "microbian.h"

void put_string(char *s) {
    for (char *p = s; *p != '\0'; p++)
        serial_putc(*p);
}

static const char *slogan[] = {
    "no deal is better than a bad deal\n",
    "BREXIT MEANS BREXIT!\n"
};

void speaker(int n) {
    while (1)
        put_string(slogan[n]);
}

void init(void) {
    serial_init();
    start("May", speaker, 0, STACK);
    start("Farage", speaker, 1, STACK);
}
```

Figure 4: *Program for Exercise 4.6*

4.6 The program shown in Figure 4 was written to display political slogans, but (un)fortunately its output is garbled. Why? Closer examination reveals that characters from the two slogans alternate in the output: "nBoR EdXeIaTl MiEsA NbSe. . . ". Why does that happen?

Design a modification to the program that (unlike the Today programme on Radio 4) allows each speaker to complete a sentence before the other one intervenes. If your first solution involves the two speakers transmitting their slogans via a coordinating process (the 'presenter'), design another solution where the presenter does not handle the text of each slogan, but only coordinates them by giving them permission to speak, one at a time.

(For authenticity, the two speakers in this simulation repeat the same, tired phrases over and over again, but your solution should also accommodate a more fruitful debate, where the two speakers concoct a series of new lies, using some method that cannot be delegated to the presenter.)

4.7 micro:bian provides an operation

```
sendrec(dest, type, &m);
```

that is equivalent to the two calls,

```
send(dest, type, &m);
receive(REPLY, &m);
```

It is useful as a form of 'remote procedure call', where a client process sends a request to a server process and then waits for a reply. Outline, in terms of process states, how this operation can be implemented. What efficiency advantages does it offer, compared with the equivalent `send` followed by `receive`? How does using `sendrec` help to ensure process priorities are

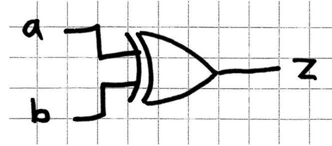
respected in a situation where a low-priority client sends a request to a high-priority server process?

4.8 The interface of `receive` requires that a process be prepared to accept a message either of a specific type, or any message at all. Suggest changes to the interface and the implementation of micro:bian that would allow any set of acceptable message types to be specified.

5 Digital logic

5.1 An XOR gate $z = a \oplus b$ has the following truth table:

a	b	z
0	0	0
0	1	1
1	0	1
1	1	0

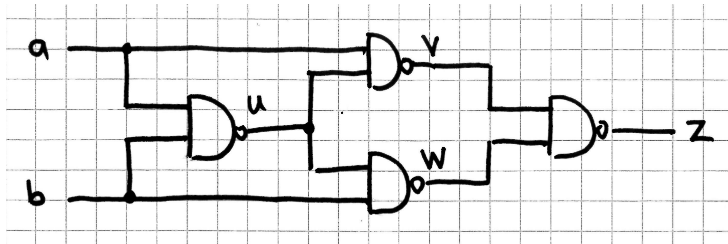


(a) Show that $\oplus$ is associative and commutative. Does it have an identity element?

(b) Show how to build an XOR gate from a 2-input OR gate, two 2-input AND gates and two inverters.

(c) Can you still build an XOR gate if one of the two AND gates is replaced by an OR gate?

(d) Show that the following circuit of four NAND gates also computes $z = a \oplus b$.



5.2 (a) Design a CMOS implementation of a NOR gate, with the following truth table.

a	b	z
0	0	1
0	1	0
1	0	0
1	1	0

(b) In the lecture, we designed a CMOS gate that computed the function

$$z = \neg((a \wedge b) \vee c).$$

Design a gate that computes

$$w = \neg((a \vee b) \wedge c)$$

instead.

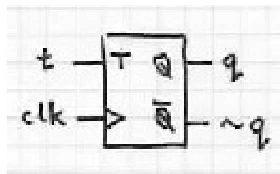
(c) What general principle relates part (a) with the CMOS NAND gate designed in lectures, and part (b) with the AND–OR–NOT gate designed there?

5.3 (a) Design a *clocked set/reset latch* with the following behaviour. There are two inputs a and b; if $a = 1$ at a clock edge, then the output z goes from 0 to 1. The output then remains at 1 until $b = 1$ at a clock edge, and then returns to 0. The behaviour if $a = b = 1$ at any clock edge can be whatever is easiest to implement.

(b) Enhance your design to produce an additional output w that receives a pulse for exactly one clock cycle whenever the circuit is triggered by an event with $a = 1$, but does not receive another pulse until the circuit has been reset by setting $b = 1$ at a clock edge.

5.4 A T-type flip-flop has a control input t, in addition to an edge-triggered clock input. If $t = 1$ at a clock edge, then the flip-flop changes state; otherwise it remains in the same state.

q_t	t	q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0



(a) Show how to construct a T-type flip-flop from a D-type flip-flop and an XOR gate.

(b) Show how to construct a synchronous binary counter from a row of T-type flip-flops and a row of AND gates. The counter should satisfy the specification $bin(a_{t+1}) \equiv bin(a_t) + 1 \pmod{2^n}$.

(c) Show how to construct a synchronous binary counter from a row of D-type flip-flops and a row of half-adders.

(d) Use your answer to part (a) to explain the connection between the circuit in parts (b) and (c).

5.5 Tests with an actual pull-cord light switch installed at the lecturer's home reveal that the light does not go on until the cord is released, but goes off as soon as it is pulled a second time. Modify the bathroom light-switch circuit to reproduce this behaviour.

5.6 In the lecture, it was shown that the set of connectives $\{\wedge, \vee, \neg\}$ is adequate to express any Boolean function, as is the singleton set {NAND}.

(a) Show that the singleton {NOR} is also adequate.

(b) Show that the set $\{$XOR$, \neg\}$ is *not* adequate. *Hint: find a proper subset of the set of all Boolean functions of two variables x and y that contains x, y and the two Boolean constants and is closed under* XOR *and* $\neg$.

5.7 A *popcount circuit* has n Boolean inputs, and computes a binary number (with $\lfloor \log n \rfloor + 1$ bits) that counts the number of 1 bits among the inputs.

(a) Show how to construct a popcount circuit from a balanced tree of adders so that the combinational path from each input bit passes through $O(\log n)$ adders before reaching the output.

(b) If we use ripple-carry adders to implement the circuit, a k-bit adder has both size and worst-case delay that are linear in k. Use these facts to estimate the size and propagation delay of the popcount circuit.

(c) In fact, some of the delays in ripple-carry adders are smaller than the estimate $O(k)$, because for $i \leq j$, the combinational path from the i'th pair of inputs to the j'th output has length proportional to $j - i + 1$. Use this fact to refine your estimate of the delay of the popcount circuit.

5.8 A bit-serial comparator has two inputs a and b. Successive binary digits of two numbers are presented at the two inputs on successive clock cycles, least significant bit first, and the circuit has two outputs L and G that indicate whether the number presented so far at a is less than or greater than the number presented at b (up to the preceding clock cycle); both outputs are zero if the numbers are equal so far. Thus, if the inputs at a and 0, 1, 1, 0 and those at b are 1, 0, 1, 1, then after 4 clock pulses the outputs are $L = 1$ and $G = 0$ because $6 = 0110_2$ is less than $13 = 1101_2$.

(a) If the current outputs are L_i and G_i and the current input bits are a_i and b_i, show how to compute the next outputs L_{i+1} and G_{i+1}.

(b) Use the previous part to give the design for a sequential circuit that inputs the numbers a and b and outputs L and G as described.

(c) What would change if the numbers a and b were presented with their most significant bit first?

6 Microprocessor architecture

6.1 [Hennessy & Patterson exx. 5.1–2, translated] A common fault in chip manufacture is that signals become stuck at logic 0 or logic 1. For each of the following stuck-at faults in the processor design shown in the handout, describe the effect on the function of the processor. Which instructions would still work correctly?

(a) *cRegSelC* = *Rx* or *Rw*.

(b) *cRand2* = *RegB* or *Imm8*.

(c) *cMemRd* = *F* or *T*.

(d) *cWReg* = *N* or *Y*.

(e) *cWFlags* = *F* or *T*.

6.2 When the `bl` instruction is split into two halves, the decoding table reveals that the displacement in the first half is sign extended, but that in the second half is not. Why is this correct?

6.3 Later versions of the ARM chip have compare-and-branch-if-zero instructions like

```
cbz r2, lab
```

which tests register `r2` and branches to a label (represented by a displacement) if the register contains zero, without affecting the condition flags. There is also a compare-and-branch-if-nonzero instruction `cbnz` that works in a similar way. What factors make these instruction difficult to implement on our datapath design? Describe in outline the changes that would be needed to implement them.

6.4 In native ARM code, not just branches, but almost any operation can be made conditional on the flags. One of the most useful such operations is a conditional move: the sequence

```
cmp r0, #0
moveq r0, r1
```

checks to see if `r0` contains zero, and if so, replaces its contents with the contents of `r1`. Similar instructions exist for all 14 conditions that are supported for conditional branches, and they all execute in one clock cycle, whether the move happens or not.

(a) Show how a conditional move instruction can be used to compute the maximum of two values in registers without using any branches. Compared with the branching code, how much time would be saved in the Cortex-M0 implementation?

(b) Devise an encoding for conditional moves as Thumb code, and write an entry for the decoding table that implements them in the single-cycle architecture. (If you want to add the instructions to the simulator, you can use the opcodes 24 and 25 that are so far unimplemented.)

(c) Leaving aside the issue of finding suitable encodings for the instructions, what other operations could be made conditional and implemented with the existing single-cycle datapath? What operations could not be made conditional without changing the datapath, and why?

6.5 Native ARM code provides an addressing mode where the values of two registers are added to form the address, but the value of one of the registers is first shifted left. For example, the instruction

```
ldr r0, [r2, r3, LSL #2]
```

loads `r0` with the 4-byte value stored at address `r2 + r3*4`. Native code may shift the register by any amount, multiplying by any power of two, but we consider here only scaling by 4 as shown.

(a) Explain why this addressing mode is particularly useful in programs that contain a lot of array indexing.

(b) Write decoding rules for versions of the `ldr` and `str` instructions that implement this addressing mode. (If you want to add them to the simulator, you could use opcodes 14 and 15.)