# Digital Systems: More sample exam questions (with answers)

Mike Spivey, Trinity Term, 2020

**1** Some 'friends' of yours are starting an indoor gardening project, and you have agreed to build a soil temperature monitor for them in return for a share of the produce. The hardware design is complete, with temperature sensors connected to an analogue-to-digital converter (ADC) in your chosen microcontroller. After configuration, the ADC is started by setting a device register `ADC_START` to 1. Some time later, the ADC sets event register `ADC_DONE` to 1 and causes an interrupt that calls the handler function `adc_handler`. The reading is then available in device register `ADC_DATA`. After `ADC_DONE` is cleared, the device will not request another interrupt before it is started again. The ADC has eight channels, selected by assigning an integer from 0 to 7 to the device register `ADC_CHANNEL`; this must done before starting the ADC.

(a) In the existing design, there is one temperature sensor connected to channel 0 of the ADC, and the control software is single-threaded. Design an interrupt-controlled driver for the ADC, providing a function

```
int adc_reading(void)
```

that takes a reading and returns its value. [8 marks]

(b) Following the success of the first growing season, your friends decide to expand the project, and now need a temperature monitor that supports multiple sensors. You decide to connect the sensors to different channels of the same ADC, and to use a message-based process scheduler to organise the software, including multiple processes that each monitor the temperature of a different sensor. Design a driver process for the ADC, providing a function

```
int adc_reading(int channel)
```

the may safely be called by other processes to read the temperature on a specified channel, ensuring that one reading is finished before another one starts. [12 marks]

**2** Figure 1 shows part of the circuit diagram for a CMOS gate with complementary pull-up and pull-down networks that computes output $z$ from inputs $a$, $b$, $c$, $d$.

Copyright © 2019–20 J. M. Spivey

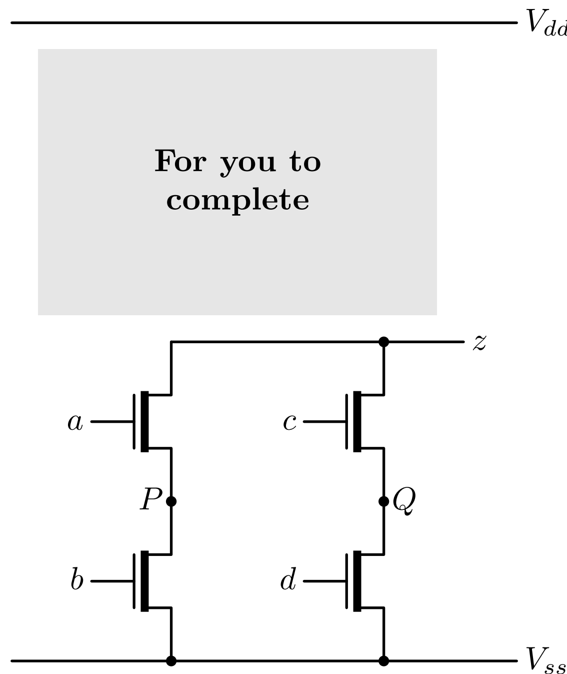


**Figure 1:** *Circuit diagram for Question 2*

(a) Complete the circuit diagram. [3 marks]

(b) Determine the Boolean function that is computed by the gate. [3 marks]

(c) Another CMOS gate has the same pull-down network as in Figure 1, except that the points labelled $P$ and $Q$ are connected to each other. What corresponding change is needed in the rest of the circuit? [2 marks]

(d) Determine the Boolean function that is computed by the second circuit, and find values for the inputs that lead the two circuits to produce different outputs. [3 marks]

(e) Design a fully synchronous sequential circuit that monitors a single input $a$ and has a single output $z$. If $a$ has been high at two or more successive clock edges and is low at the next clock edge, then the circuit should immediately produce an output pulse for one clock cycle, and it should repeat this behaviour indefinitely. [9 marks]

**3** This question concerns implementations of the instruction set of a little-endian RISC-like machine. There are loads and stores with addressing modes where the address is computed either as the sum of two registers, or as the sum of a register and a small constant.

- The instructions `ldr` and `str` permit loads and stores where a 32-bit value is transferred between a register and memory.
- The instructions `ldrb` and `strb` load and store an 8-bit value in the least significant bits of a 32-bit register, with zero extension for the load.

For ldr and str, the effective address must be a multiple of 4, but ldrb and strb allow any address to be used.

The instruction set is implemented on a datapath that executes each instruction in a single cycle, and has separate memory interfaces for instructions and data.

(a) Give examples of constructs in a high-level language that can take advantage of the two addressing modes. [2 marks]

(b) Draw in outline and explain the design of a datapath that can implement 32-bit loads and stores using these addressing modes, stating what role is played by each functional unit in executing the instructions. [5 marks]

(c) If the ldrb instruction were not implemented, could the instructions

```
ldrb r0, [r1, #5]
```

and

```
strb r0, [r1, #5]
```

be replaced by an equivalent sequence of instructions that does not use ldrb or strb but only the ldr and str instructions? Assume that plenty of spare registers are available, and that the equivalent sequence may have a different effect on the condition codes. Either show a suitable sequence of instructions, or explain why one does not exist. [6 marks]

(d) A datapath that supports 8-bit loads might use an unchanged interface to data memory, fetch a whole word, but select the appropriate byte to write to a register. Describe a suitable functional unit that might be added to the datapath for this purpose, explaining what control signals it would use and showing how the unit might be constructed from logic gates. [5 marks]

(e) Could an additional modification also support 8-bit stores with the same memory interface, executing each store in a single cycle? Justify your answer. [2 marks]

**4** [2010/4] A certain microprocessor has a 32-bit datapath and uses both unsigned binary and twos-complement signed representations of numbers. There are instructions add and sub that add and subtract the contents of two registers, storing the results in a third register. There also a comparison instruction that subtracts two registers and sets the four flags NZCV from the result, and instructions blt and blo that branch if the flags indicate that the first register is less than the second in value. The blt instruction interprets its inputs as twos-complement signed numbers, and blo interprets them as unsigned numbers.

(a) Define two functions that map sequences of 32 bits to the integers they represent in the twos-complement and in the unsigned representation. [2 marks]

(b) Specify precisely the function of an adder that adds two 32-bit unsigned numbers to obtain a 32-bit result, and show that the same adder can be used with twos-complement numbers. [4 marks]

(c) Show how to combine a 32-bit adder with appropriate additional logic to make a circuit that performs 32-bit subtraction. [4 marks]

(d) Give an example to show that the `blt` and `blo` instructions can give different results. [4 marks]

(e) Give an example to show that the `blt` instruction cannot simply use the sign bit that results from a 32-bit subtraction. Describe appropriate logic to compute the result, and show that it is correct. [4 marks]

(f) Design logic to give the correct result for the `blo` instruction, and show that it is correct. [2 marks]

**5** [2016/3] Consider the following definition of a function `foo`:

```
int foo(int y) {
    if (y < 3)
        return 1;
    else
        return bar(y-2) + bar(y-3);
}
```

Below is an incomplete compilation of the function `foo` into Thumb code.

```
foo:
    push {r4, r5, lr}
    movs r4, r0
    cmp r4, #3
    <missing instruction 1>
    subs r0, r4, #2
    bl bar
    movs r5, r0
    <missing instruction 2>
    bl bar
    <missing instruction 3>
    b foo_end
foo_ret1:
    movs r0, #1
foo_end:
    pop {r4, r5, pc}
```

(a) Outline the calling convention that is exemplified in this code. [4 marks]

(b) Briefly explain the purpose of the first and last instructions in the function, and why the lists of registers are as they appear. [4 marks]

(c) Supply the three instructions that are missing in the code above. [4 marks]

Now consider the definition of the function `bar`. Below is its compilation into Thumb code.

```
bar:
    push {r4, lr}
    movs r4, r0
```

```
    cmp r4, #2
    blt bar_ret1
    adds r0, r4, #1
    bl foo
    subs r0, r4, r0
    b bar_end
bar_ret1:
    movs r0, #1
bar_end:
    pop {r4, pc}
```

(d) Supply suitable code for the two blanks in the following high-level code for bar.

```
int bar(int x) {
    if (<blank 1>)
        return 1;
    else {
        <blank 2>
    }
}
```

[4 marks]

(e) Write a ‘test program’ in Thumb code that calls foo(x) successively for all values of x from 1 to 10, and calls print(y) with the value y returned by each call of foo. [4 marks]

## Model answers

**1** (a) There is only one thread, so concurrent calls to adc_reading are impossible.

```
volatile adc_active = 1;

int adc_reading(void) {
    ADC_CHANNEL = 0;
    adc_active = 1;
    ADC_START = 1;
    while (adc_active) pause();
    return ADC_DATA;
}

void adc_handler(void) {
    if (ADC_DONE) {
        ADC_DONE = 0;
        adc_active = 0;
    }
}
```

Here, pause() halts the processor until the next interrupt. The variable adc_active must be declared volatile to prevent compiler optimisations from concluding that its value will not change asynchronously.

The code follows the common pattern of checking the reason for the interrupt (ADC_DONE) in the handler, though in this case there is only one known reason for an interrupt.

(b) Now we use a server process, with selective receive – `receive(HARDWARE, &m)` – to ensure a fresh request is not accepted before the previous request is complete.

```
void adc_driver(int arg) {
    message m;
    int client, data;

    // Initialise the ADC

    // Set up the ADC interrupt
    connect(ADC_IRQ);

    while (1) {
        receive(any, &m);

        switch (m.m_type) {
        case READING:
            client = m.m_sender;
            ADC_CHANNEL = m.m_i1;
            ADC_START = 1;
            receive(HARDWARE, &m);
            assert(m.m_type == INTERRUPT);
            assert(ADC_DONE);
            ADC_DONE = 0;
            data = ADC_DATA;

            m.m_type = DATA;
            m.m_i1 = data;
            send(client, $m);
            break;

        case INTERRUPT:
            panic("Unexpected interrupt");
            break;

        default:
            panic("ADC got bad message");
        }
    }
}
```

The function `adc_reading` constructs and sends a message.

```
int adc_reading(int channel) {
    message m;
    m.m_type = READING;
    m.m_i1 = channel;
    sendrec(ADC, &m);
    assert(m.m_type == DATA);
    return m.m_i1;
}
```

**2** (a) See Figure 2.

(b) The pull-down network drives the output low if either $a$ and $b$ or $c$ and $d$ are high, so the function is

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

(c) Referring to Figure 2, the connection between $R$ and $S$ must be broken to maintain complementarity.

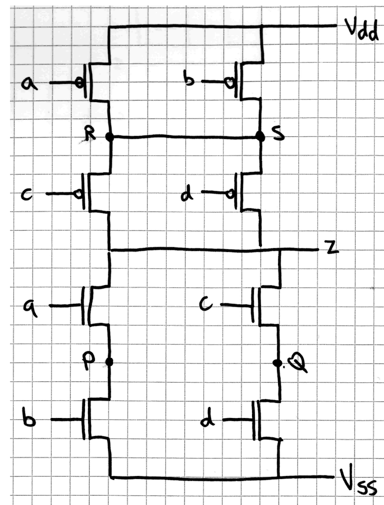


**Figure 2:** *Completed circuit for Question 2*

(d) Now the function computed is

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

The functions differ if $abcd = 1001$ or $0110$, when $z = 1$ but $w = 0$.

(e) The output can be derived from observations of the input at the most recent three clock edges, which must follow the pattern 110 for the output to be high. A solution is therefore a shift register with three stages $u$, $v$, $w$ with the wiring $u' = a$, $v' = u$, $w' = v$ and a 3-input AND gate giving output $z = \neg u \wedge v \wedge w$.

Alternatively, we can design a FSM with four states: $Q_0$ = idle, $Q_1$ = count 1, $Q_2$ = count 2, $Q_3$ = active, and the state transition diagram shown in Figure 3. Sneakily observing that all 1-transitions lead to states $Q_1$ or $Q_2$, we encode the states as $Q_0 = 00$, $Q_1 = 01$, $Q_2 = 11$, $Q_3 = 10$ to obtain the transition table,

| $u$ | $v$ | $a$ | $u'$ | $v'$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1. |

We can read off $v' = a$ as expected, and also $u' = v \wedge (a \vee u)$. The output is given by $z = u \wedge \neg v$. So much grief to save one flip-flop!

The simple, perspicuous solution simulates the minimal one via the abstraction function $f(uvw)$ given by

$$f(000) = f(001) = f(010) = Q_0,$$
$$f(100) = f(101) = Q_1,$$
$$f(110) = f(111) = Q_2,$$
$$f(011) = Q_3.$$

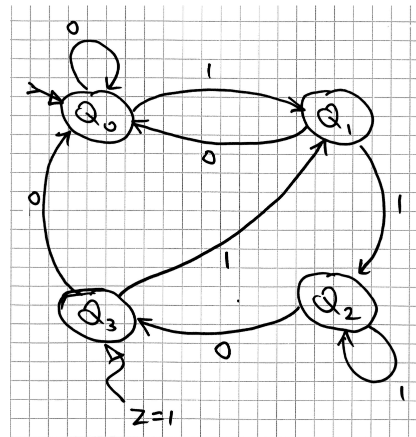


**Figure 3:** *Flying spaghetti monster for Question 2*

**3** (a) The reg+reg addressing mode is useful for array indexing: if `r1` contains the base address of an array and `r2` contains the offset of an element, then `ldr r0, [r1, r2]` will load the element into register `r0`.

The reg+const addressing mode can be used to fetch fields from a record whose address is in a register, with the fields at constant offsets. Additionally, if the register if `sp`, the addressing mode gives access to local variables at fixed offsets in the stack frame.

(b) See the datapath diagram from the course; answers should refer to the functional units listed below, but need not mention the many muxes that feed them.

- One, two, or three register values are output by the register file.
- The offset is scaled if appropriate by the barrel shifter.
- The base address and offset are added by the ALU.
- The data memory performs a read or write cycle.
- If the instruction if `ldr`, the result is written back by the register file.

(c) For clarity, the following instruction sequences use more registers than are really needed. For `ldrb r0, [r1, #5]`, we first compute the effective address, and split it into the word address (a multiple of 4), and the last two bits that give the index of a byte within the word.

```
adds r2, r1, #5    @ Compute effective address
movs r3, r2
movs r4, #3        @ Mask for lower bits
bics r2, r4        @ Word address in r2
ands r3, r4        @ Byte index in r3
lsl r3, r3, #3     @ Multiply byte index by 8
```

Now we fetch the word, shift it to obtain the correct byte in the bottom 8 bits, then zero out the remainder.

```
ldr r0, [r2]       @ Fetch the word
lsr r0, r3         @ Shift desired byte to bottom
uxtb r0, r0        @ Zero out upper 24 bits.
```

For `strb`, we can start with the same address calculation, but then we must fetch the whole word, modify the appropriate byte, and store it back.

```
ldr r5, [r2]       @ Fetch the whole word
```

```
movs r4, #0xff       @ Compute a mask for the unwanted bits
lsls r4, r4, r3
bics r5, r4          @ Clear out existing byte
uxtb r4, r0          @ Compute replacement byte
lsls r4, r4, r3
orrs r5, r4          @ Combine old and new values
str r5, [r2]         @ Store back the word
```

(d) We need an additional barrel shifter connected to the output of the data memory interface, capable of passing its input through unchanged, or shifting it by 0, 8, 16 or 24 bits, selecting the bottom byte, and extending with zeroes. The barrel shifter can have three stages: the first shifting by 0 or 16 bits and producing a 16-bit output, the second shifting by 0 or 8 bits and producing an 8-bit output, and the third selecting either the output of the second stage extended with 24 zeroes, or selecting the original input.

The control inputs to the first two stages come from the bottom two bits of the effective address, which are not used by the word-oriented data memory. The control input of the last stage is derived from the opcode of the instruction, and is active only for the `ldrb` instruction. Each stage is a row of 1-bit, 2-input multiplexers, and can be built from an OR gate, two AND gates and an inverter.

```
memout' = mux_32(cByteLoad, mem2 ++ 0_24, memout)

mem2 = mux_8(aluout[0], mem1[15:8], mem1[7:0])

mem1 = mux_16(aluout[1], memout[31:16], memout[15:0])
```

(e) As the assembly-language equivalent reveals, an 8-bit store would require a read–modify–write operation on the memory, and the needed read and write could not happen in the same cycle.

**4** (a) For the unsigned representation we use $bin : \mathbf{B}_{32} \to [0, 2^{32})$ with $bin(a) = \sum_{0 \le i < 32} a_i \cdot 2^i$ and for twos-complement $twoc : \mathbf{B}_{32} \to [-2^{31}, 2^{31})$ with $twoc(a) = \sum_{0 \le i < 31} a_i \cdot 2^i - a_{31} \cdot 2^{31}$ or in terms of $bin(a)$: $twoc(a) = bin(a) - a_{31} \cdot 2^{32}$

(b) As we are only specifying a 32-bit result, it will only be correct modulo $2^{32}$ (possible overflow). $bin(s_0, \ldots, s_{n-1}) \equiv bin(a_0, \ldots, a_{n-1}) + bin(b_0, \ldots, b_{n-1}) \pmod{2^{32}}$ Because of

$$twoc(a) = bin(a) - a_{31} \cdot 2^{32},$$

we have

$$twoc(a) \equiv bin(a) \pmod{2^{32}}.$$

Therefore, if

$$bin(s) \equiv bin(a) + bin(b) \pmod{2^{32}},$$

then also

$$\begin{aligned} twoc(s) &\equiv bin(s) \pmod{2^{32}} \\ &\equiv bin(a) + bin(b) \pmod{2^{32}} \\ &\equiv twoc(a) + twoc(b) \pmod{2^{32}} \end{aligned}$$

Hence a 32-bit adder that gives the correct result $\pmod{2^{32}}$ for unsigned operands will also give the correct result for signed operands in two's complement representation.

(c) $-twoc(b) = twoc(\overline{b} + 1)$, hence inverting all bits of $b$ and setting the carry-in to 1 will perform subtraction instead of addition, $twoc(s) \equiv twoc(a) - twoc(b)$

(mod $2^{32}$). By using an additional control signal ⟨sub⟩ and 32 xor-gates in front of the second input to the adder, we can use the same adder for addition and subtraction.

(d) Let $a = 000 \cdots 0$ and $b = 1111 \cdots 1$ then `blo` will branch while `blt` will not, because $twoc(b) = -1$.

(e) Any example that produces an "overflow" when performing the subtraction, eg. $a = 0111 \cdots 1$ and $b = 1000 \cdots 0$ then the result $d$ of the subtraction will be $d = 111 \cdots 1$. Hence the sign bit would give 1 but $twoc(a) > twoc(b)$.

Overflow can only happen if $a$ and $b$ have different sign bits, and is detected when the sign of $d$ does not agree with the sign of $a$. Let $a_s$, $b_s$ and $N = d_s$ be the sign bits of $a$, $b$ and $d$ respectively, and let $V = (a_s \equiv b_s) \wedge (d_s \not\equiv a_s)$. Then the correct result is given by $N \not\equiv V$.

(f) Let $C$ be the carry-out from the subtraction: it indicates whether $bin(a) + (2^{32} - bin(b)) \geq 2^{32}$ or equivalently whether $bin(a) - bin(b) \geq 0$. The correct result is to branch if $\neg C$.

**5** (a) Arguments are passed in `r0`–`r3`, with the first argument (the only one in this case) in `r0`. The subroutine may trash these registers, and it returns its result in `r0`. Registers `r4`–`r7` must be preserved by a subroutine, so must be saved and restored if the subroutine overwrites them. The call instruction saves the return address in `lr`, and the subroutine returns by branching to this address. The subroutine may use stack space for local storage, but must restore the stack pointer to the value it had at the call.

(b) Here's the code again, but with the missing instuctions added, enclosed in `<...>`.

```
foo:
    push {r4, r5, lr}
    movs r4, r0
    cmp r4, #3
    <blt foo_ret1>
    subs r0, r4, #2
    bl bar
    movs r5, r0
    <subs r0, r4, #3>
    bl bar
    <adds r0, r5, r0>
    b foo_end
foo_ret1:
    movs r0, #1
foo_end:
    pop {r4, r5, pc}
```

(c) The same trick, with the missing parts shown in `<...>`.

```
int bar(int x) {
    if (<x < 2>)
        return 1;
    else {
        <return x - foo(x+1);>
    }
}
```

(d) Here's possible C code for the test program:

```
void test(void) {
```

```
    for (int n = 1; n <= 10; n++)
        print(foo(n));
}
```

Translation into Thumb code:

```
test:
    push {r4, lr}
    mov r4, #1
test_loop:
    cmp r4, #10
    bgt test_done
    movs r0, r4
    bl foo
    bl print
    adds r4, r4, #1
    b test_loop
test_done:
    pop {r4, pc}
```