Digital Systems: Collected problems

Mike Spivey, Hilary and Trinity Terms, 2023

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 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);
```

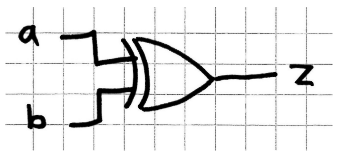
It is useful as a form of 'remote procedure call', where a client process sends a request to a server process and then waits for a reply. Outline, in terms of process states, how this operation can be implemented. What efficiency advantages does it offer, compared with the equivalent `send` followed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

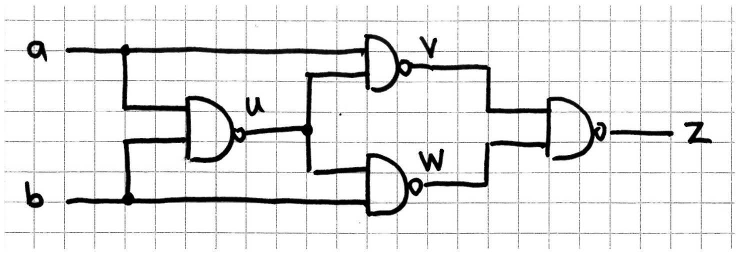


(a) Show that $\oplus$ is associative and commutative. Does it have an identity element?

(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.

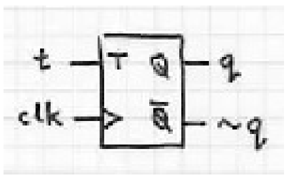
(c) What general principle relates part (a) with the CMOS NAND gate designed in lectures, and part (b) with the AND–OR–NOT gate designed there?

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.)

Answers and notes

1.1 To set `r0` to zero, we can use any of the following, and can replace `r3` in the second instruction by any register `r0` to `r7`.

```
movs r0, #0
subs r0, r3, r3
eors r0, r0
bics r0, r0
lsrs, r0, r3, #32
```

All of these set the conditions codes *NZ* to 01, but they have various effects on the *V* and *C* bits. Note that the `lsrs` instruction has an immediate field in the range 1 to 32, with 32 encoded by setting all five bits of the immediate field to 0.[1]

Copying `r1` to `r0` can use any of the following.

```
lsls r0, r1, #0  @ synonymous with movs r0, r1
adds r0, r1, #0  @ an instance of adds imm3
subs r0, r1, #0
mov  r0, r1      @ an instance of mov hi/lo
```

All but the last of these set the *N* and *Z* according to the value copied, and have various effects of the *V* and *C* bits. We cannot use `lsrs` or `asrs` for this purpose, as there is no encoding for a shift of 0 with these instructions. To add to the confusion, the assembler does accept the form `lsrs r0, r1, #0`, but treats it as another synonym for `lsls r0, r1, #0`.

1.2 Here's the program again, but with each instruction word shown split into binary fields. Each instruction contains an opcode of from 5 to 8 bits, followed by fields containing register numbers (usually 3 bits each) and immediate fields of different widths.

```
          Opcode    Fields
                    rw  rz  ry  rx     imm
                                                 func:
c0: 2200  00100     010                00000000      movs    r2, #0
                                                 loop:
c2: 2900  00101     001                00000000      cmp     r1, #0
c4: d002  11010000                     00000010      beq     cc <done>
c6: 3901  00111     001                00000001      subs    r1, #1
c8: 1812  0001100       000 010 010                  adds    r2, r2, r0
ca: e7fa  11100                     11111111010      b       c2 <loop>
                                                 done:
cc: 0010  00000     00000   010 000                  movs    r0, r2
ce: 4770  010001110        1110 000                  bx      lr
```

Note that:

- The first `movs`, the `cmp`, and the `subs` instructions contain 8-bit immediate fields, with values 0, 0 and 1 respectively.
- The `beq` has an 8-bit displacement field with a value of 2. Multiply this by 2 and add 4 to give an effective displacement of 8, which is the difference between the address of the instruction (c4) and the branch target (cc).
- The order in which the three registers are specified in the `adds` instruction is different from the order they appear in the assembly language source.

[1] Thanks to Cara Higgins, Keble College, for pointing out this subtle feature.

- The b has an 11-bit displacement field with a value of −6. Again double and add 4 to give and effective displacement of −8, which is the difference between the instruction address (0xca) and the branch target (0xc2).
- The second movs instruction is a shorthand for the shift instruction lsrs r1, r2, #0, with the shift amount of 0 specified by a 5-bit immediate field.
- The return instruction bx lr mentions the lr register, number 14. The final three bits have no meaning and are set to zero for tidiness (and maybe compatibility with future extensions to the instruction set).

1.3 The existing routine is as follows.

```
func:
    movs r2, #0             @ z = 0;
loop:
    cmp r1, #0              @ while (y != 0) {
    beq done
    subs r1, r1, #1         @   y = y-1;
    adds r2, r2, r0         @   z = z+x;
    b loop                  @ }
done:
    movs r0, r2             @ return z;
    bx lr
```

An immediate improvement is to put the test at the bottom, so that only one branch instruction appears in the loop.

```
func:
    movs r2, #0             @ z = 0;
    b test
again:
    subs r1, r1, #1         @   y = y-1;
    adds r2, r2, r0         @   z = z+x;
test:
    cmp r1, #0              @ while (y != 0)
    bne again
done:
    movs r0, r2             @ return z;
    bx lr
```

But we can do slightly better still; first, treat zero as a special case that avoids entering the loop at all, and eliminate the label test. Next, swap the subs and adds instructions. Finally, observe that the condition codes set by the subs instruction allow us to detect when r0 has reached zero.

```
func:
    movs r2, #0             @ z = 0;
    cmp r1, #0
    beq done
again:
    adds r2, r2, r0         @   z = z+x;
    subs r1, r1, #1         @   y = y-1;
    bne again               @ while (y != 0)
done:
    movs r0, r2             @ return z;
    bx lr
```

This achieves 5 cycles per iteration. We can do better by unrolling the loop like this:

```
func:
        movs r2, #0
        cmp r1, #0
        beq done
again:
        adds r2, r2, r0
        subs r1, r1, #1
        beq done
        adds r2, r2, r0
        subs r1, r1, #1
        beq done
        adds r2, r2, r0
        subs r1, r1, #1
        beq done
        adds r2, r2, r0
        subs r1, r1, #1
        bne again
done:
        movs r0, r2
        bx lr
```

Now four interations of the loop require 11 instructions plus a taken branch, or 14 cycles, giving 3.5 cycles per iteration. Further unrolling uses more ROM space but brings down the number of cycles per iteration closer to the limit of 3.

1.4 We can replace an instruction `beq lab` that has insufficient range with the two-instruction sequence,

```
    bne stay
    b lab
stay:
```

The original conditional branch would take one cycle if not taken, and three cycles if taken. The replacement takes three cycles if the original `beq` was not taken (because the `bne` will be taken), and four cycles if it was taken (because the `bne` is not taken but the `b` is taken).

We can use `bl` in a similar way, but it trashes the `lr` register; this will not matter in a non-leaf routine where `lr` has been saved on entry.

1.5 If `r0` contains `0xffffffff` and `r1` contains `0x00000001`, then these are interpreted as the signed numbers -1 and 1, and the instruction sequence `cmp r0, r1; blt lab` should result in a taken branch. Viewed as *unsigned* numbers, however, `r1` still contains 1, but `r0` is read as $2^{32} - 1$, a huge positive number. The branch in `cmp r0, r1; blo lab` ought therefore not to be taken, and `blt` and `blo` do different things in this case.

For `blt`, the condition is $N \neq V$. The `cmp` instruction sets the condition codes from the result of a subtraction $a - b$ but throws away the result of the subtraction itself. For inputs of moderate size, the N bit (equal to r_{31}) correctly shows whether $a - b$ is negative, i.e., whether $a < b$. If a and b have the same sign, then overflow is impossible; but if they have opposite signs, then the sign of the result ought to agree with the sign of a. If not, then the overflow bit V is set, and the sense of N must then be reversed to find if the subtraction had a negative result. The formula $N \neq V$ gives the right answer in each case.

For `blo`, the condition is $C = 0$. The examples below are shown with 8 bits instead of 32 for clarity. The left example shows the subtraction $a - b = 9 - 7$, performed by complementing each bit of b, then adding the result and an additional 1 to a. The result is shown with 9 bits in order to make the carry-out (the C bit) explicit. As you will see, the 8 bit result is correctly calculated as 2, and $C = 1$; the result is correct whether we read it as signed or unsigned. In the right-hand example, the subtraction is $a - b = 7 - 9$; this produces the correct result -2 in the signed case, but produces the wrong answer $2^8 - 2$ if read as an unsigned number.

```
 00001001       00000111
 11111000       11110110
        1              1
---------      ---------
100000010      011111110
```

The point being illustrated is that, for an *unsigned* subtraction $a - b$, the C bit is 1 exactly if $a \geq b$, giving a correct unsigned result, and $C = 0$ if $a < b$.

1.6 We assume two arguments $a \geq 0$ and $b > 0$, and compute q and r such that $a = q * b + r$ and $0 \leq r < b$, and return q as the result. A simple loop maintains this as the invariant, except that r may exceed b.

```
q = 0; r = a;

while (r >= b) {
    q = q+1; r = r-b;
}

return q;
```

To code this, it's easiest to keep r in `r0`, so that the initialisation `r = a` disappears. We keep q in `r2`.

```
    movs r2, #0             @ q = 0;
loop:
    cmp r0, r1              @ while (r >= b)
    blo done                @ Note unsigned comparison
    adds r2, r2, #1         @   q = q+1;
    subs r0, r0, r1         @   r = r-b;
    b loop
done:
    movs r0, r2
    bx lr
```

As before, we can squeeze the loop by moving the test to the end. Note that this routine enters an infinite loop in the case $b = 0$. We had better do something about stopping the program with an error message in this case before letting it out into the wild.

1.7 We will use binary long division, divided into two phases. In the first phase, we compute an exponent j such that $a < b * 2^j$, thereby determining an upper bound on the number of digits in the answer. The second phase is the long division itself, using the invariant

$$a = q * b * 2^j + r$$

with $0 \leq r < b * 2^j$. Reducing j to zero while maintaining this invariant gives the desired quotient q.

To simplify the first phase, we ignore a temporarily, and find the smallest j such that $b * 2^j \geq 2^{32}$. Note that the constant 2^{32} is outside the representable range, but we will find a clever implementation of the loop test later.

```
j = 0;
```

```
do { j++; } while (b * 2^j < 2^32);
```

The second phase establishes and maintains the stated invariant. We will use shifts to implement multiplication by powers of 2, but I'll write the powers explicitly for now. Note that the calculation $b * 2^j$ will not overflow once j has been reduced from its initial value.

```
q = 0; r = a;
do {
    j--; q *= 2;
    if (r >= b * 2^j) {
        q += 1; r -= b * 2^j;
    }
} while (j > 0);
```

To implement this in assembly language, let's keep r in `r0`, b in `r1`, q in `r3` and j in `r4`. We will use `r2` as a scratch register. Using `r4` means that we must save its initial value and restore it at the end, even though this routine calls no others.

```
    push {r4, lr}
```

To implement the first phase, we will use that fact that in a left shift instruction, the last bit shifted out of the register is put in the C bit of the condition codes. Thus the `bcc` instruction below continues the loop until the leading 1 bit has shifted out of `r2`.

```
    movs r4, #0             @ j = 0;
    movs r2, r1
double:                     @ while ((b << j) < 2^32) j++;
    adds r4, r4, #1
    lsls r2, r2, #1
    bcc double
```

Now for the second phase. Because initially `b << j` overflows (we've just made sure that's true), it seems simplest to recompute that expression each time in the loop (after decrementing j), rather than saving it from one iteration to the next.

```
    movs r3, #0             @ q = 0;
loop:                       @ do {
    subs r4, r4, #1         @   j--;
    lsls r3, r3, #1         @   q <<= 1;
    movs r2, r1
    lsls r2, r4             @   b << j in r2
    cmp r0, r2              @   if (r >= (b << j)) {
    blo skip
    adds r3, r3, #1         @     q++;
    subs r0, r0, r2         @     r -= (b << j);
skip:                       @   }
    cmp r4, #0              @ } while (j != 0)
    bne loop
```

All that remains is to put the result in `r0` and return.

```
    movs r0, r3             @ return q;
    pop {r4, pc}
```

The `clz` instruction would determine the proper initial value of `j` without the need for a loop. A slightly better implementation would choose j just big enough so that $b * 2^j > a$, and that can be achieved by applying `clz` to both a and b and subtracting, likely treating $a < b$ as a special case.

We can get the same effect without `clz` by modifying the loop for the first phase. It's easiest to require $j \geq 1$ but ensure $b * 2^{j-1}$ does not overflow. In that case, we can use the fact that $a < b * 2^j$ iff $a/2 < b * 2^{j-1}$ iff $\lfloor a/2 \rfloor < b * 2^{j-1}$ to make the test without risk of overflow. That leads to the following loop.

```
j = 1;
while ((b << (j-1)) <= (a >> 1)) j++;
```

It's best to keep `b << (j-1)` and `a >> 1` in registers for the duration of the loop.

```
    movs r4, #1             @ j = 1;
    lsrs r3, r0, #1         @ a >> 1 in r3
    movs r2, r1             @ b << (j-1) in r2
    b dbltest
double:                     @ while ((b << (j-1)) <= (a >> 1))
    adds r4, r4, #1         @   j++;
    lsls r2, r2, #1         @   update r2
dbltest:
    cmp r2, r3
    bls double
```

Since $j \geq 1$ as before, the second phase can be used unmodified.

An alternative approach (`div4.s`) uses two ARM registers to hold a 64-bit integer r, maintaining the invariant,

$$a.2^{32-j} = q.2^{32} * b + r,$$

with $0 \leq r < b.2^{32}$, while reducing j from 32 to 0. The invariant is easily established by setting q to 0 and r to a. When j is decremented, we must double both q and r, and if now $r \geq b.2^{32}$, then increment q and reduce r. At the end, r is a multiple of 2^{32} and dividing through by 2^{32}, we have $a = q * b + r/2^{32}$.

```
func:
@ b = r1, q = r2, r = r3:r0, j = r4

    push {r4}
    movs r2, #0         @ q = 0
    movs r3, #0         @ r = a
    movs r4, #32        @ j = 32
    b loop

next:
    subs r4, #1         @ j--
    beq done            @ done if j = 0

loop:
    adds r2, r2         @ q = 2*q
    adds r0, r0         @ r = 2*r
    adcs r3, r3
    cmp r3, r1          @ if r >= b<<32
    blo next

    adds r2, #1         @ q++
    subs r3, r1         @ r -= b<<32
    b next

done:
    movs r0, r2         @ return q
    pop {r4}
    bx lr
```

Note that we can double the value of r by adding `r0` to itself, then using the `adc` instruction to add `r3` to itself and add in the carry from the doubling of `r0`. The pattern of jumps and labels ensures one taken branch per iteration.

2.1

- `ldr r1, [pc, #n]` is used to represent the pseudo-instruction

  ```
  ldr r1, =const
  ```

  The constant `const` is placed at a convenient location in the code stream, and this form of instruction loads it into `r1` using `pc`-relative addressing.

- `ldr r1, [r2, r3]` is used to index an array. The base address of the array is put in `r2` and the index, multiplied by the element size, is put in `r3`. The instruction adds together the base and offset and fetches the array element into `r1`.

- `ldr r1, [r2, #n]` is useless if `n` is the address of a global, since we cannot expect this address to fit in the available instruction bits. But if `r2` is the address of a record with a fixed layout for its fields, then this instriuction can be used to fetch one of the fields into `r1`.

- `ldr r1, [sp, #n]` is useful for fetching values from the stack frame of the current subroutine, whether they are local variables or parameters beyond the first four.

The `add` instructions listed each compute into a register the addresses that would be used in one of the above load instructions.

- `add r1, pc, #n` computes in `r1` the address of a constant embedded in the code stream. This is useful when the address is in fact the base address of a constant array: for example, it's possible to embed string constants in the code stream, though more usual to collect them together in a specific segement `rodata`. Also, jump tables used in the translation of `switch` or `case` statements can be placed in the code stream in a similar way. The assembler provides a pseudo-instruciton `adr r1, label` that facilitates calculation of the offset `n` as the (suitably adjusted) difference between the label and the address of the current instruction.

- `add r1, sp, #n` computes in `r1` the address of a value stored in the stack frame. It's useful for getting the base address of a local array.

- `add sp, sp, #n` adjusts the stack pointer upwards by `n`. It's used just before a return from a procedure to deallocate the stack space reserved earlier for local variables.

2.2 One of the load instructions (`ldrh`) sets the upper 16 bits of the target register to zero, and the other sets them all to copies of the most significant bit of the 16-bit value from memory. The first results in a 32-bit vector b such that $bin_{32}(b) = bin_{16}(a)$, where a is the 16-bit value loaded from memory, and it is therefore appropriate for loading unsigned numbers, belonging to the C type `unsigned short`. The second form satisfies $twoc_{32}(b) = twoc_{16}(a)$, and is appropriate for use with signed numbers with the C type `(signed) short`. In either case, the proper action on storing is to store a 16-bit value that is congruent to the 32-bit value modulo 2^{16}, and that just means storing the bottom 16 bits of the register unchanged. If the upper bits are not all zeroes or all ones as appropriate, then overflow has invariably occurred.

In the absence of an encoding of the `ldrsh` instruction with the reg+const addressing mode, we can sign extend the result explicitly, replacing `ldrsh r1, [r2, #n]` with the sequence

```
ldrh r1, [r2, #n]
```

```
sxth r1, r1
```

that uses the ‘sign extend halfword’ instruction `sxth`. In the absence of this instruction, we would use first a left shift by 16 bits, then an arithmetic right shift by 16 bits to smear out the sign bit across the upper half of the register.

2.3 The simplest approach is just to develop the offset into a register, so that (for example) `ldr r1, [sp, #2000]` is replaced by

```
ldr r1, =2000
ldr r1, [sp, r1]  @ Wrong!
```

But sadly, the sp+reg addressing mode has no encoding, so we have to copy the `sp` value into a low register first:

```
mov r0, sp
ldr r1, =2000
ldr r1, [r0, r1]
```

That looks very messy, and a better approach is to observe that a large-ish stack frame probably consists of a few scalars and one or more larger arrays. In this case, it is wise to gather all the scalars together, and either reorder the frame so the scalars are near the stack pointer, or dedicate a register (`r7`, say) to point to the block of scalars. Individual local variables can then be loaded with an instruction `ldr r1, [r7, #`*n*`]` with a value of *n* that is moderate in size. This approach is especially helpful if the arrays are variable in size.

2.4 In place of the instruction `push {r4, r5, lr}` we can write an explicit adjustment of the stack pointer followed by some store instructions. Let’s maintain the layout in memory that would be used by the `push` instruction if it existed.

```
sub sp, #12
mov r3, lr
str r3, [sp, #8]
str r5, [sp, #4]
str r4, [sp, #0]
```

On a machine where interrupts are enabled, it’s important to decrement the stack pointer *before* storing the values, lest an interrupt intervene and save its context over the values we have stored. Note that the `str` instruction is incapable of storing from a high register, so we must find a scratch register (`r3` in the code above) and move the return address there before storing it; if the procedure has more than three arguments, then further contortions are necessary, such as storing `r4` first and using that as the scratch register. The stack adjustment can be combined with another immediately following adjustment to allocate space for local variables, provided the offsets in the `str` instructions are adjusted appropriately.

At the end of the procedure, the instruction `pop {r4, r5, lr}` can be replaced by

```
ldr r4, [sp, #0]
ldr r5, [sp, #4]
ldr r3, [sp, #8]
add sp, #12
bx r3
```

Again, the stack adjustment can be combined with a preceding one, given appropriate changes to the offsets. At this point, `r3` is dead according to the calling convention, so it’s safe in every case to use it as a scratch register.[2]

[2] The actual `push` instruction stores the values first, then decrements the stack pointer; but if it is interrupted half way through, it is abandoned and executed again from scratch when the interrupt returns.

2.5 We can begin by computing `4*(i+j)` into a register; for that we will need to get the address of `i` in a register, say `r0`.

```
ldr r0, =i              @ Becomes pc-relative addressing
ldr r0, [r0, #0]        @ Could be written [r0]
ldr r1, [sp, #60]
adds r0, r0, r1
lsls r0, r0, #2
```

Now we want to fetch `b[i+j]` into a register (`r1`); the address of that is the sum of three things: the stack pointer, the offset 4 for the base of `b`, and the offset `4*(i+j)` already computed in `r0`. The reg+reg addressing mode gives us one addition for free, so we must write one add instruction.

```
add r2, sp, #4
ldr r1, [r2, r0]
```

Next, we must multiply the value in `r1` by 3. The `mul` instruction requires its operands both in registers, so we must put 3 in a register with an additional instruction.[3]

```
movs r2, #3
muls r1, r2
```

Now to store the result into `a[i+j]`. For that we need the base address of `a` in a register (`r2`); the offset `4*(i+j)` is still available in `r0`.

```
ldr r2, =a
str r1, [r2, r0]
```

The assembler will convert the `ldr=` pseudo-instructions into `pc`-relative addressing, put the constants in a convenient place in the text segment, and fill in the offsets in the instructions appropriately.

2.6 The fly in the ointment with Mr User's suggestion is that the hidden register would have to be preserved in the case that an interrupt intervened between the first half and the second half of a `bl` instruction. Perhaps there are enough implementation-defined bits in the PSR for them to be used for this purpose, but if not, then an extra word would need to be saved on entry to an exception handler and restored on exit. Alternatively, interrupts could be momentarily disabled between one half of a `bl` instruction and the other, and any interrupts delayed until the time between the `bl` instruction and the first instruction of the subroutine being called.

The solution that uses the `lr` register for this purpose is neat, because the contents of the `lr` register are about to be overwritten with the return address of the call, and they are preserved by the exception mechanism in any case. One slight risk it that an interrupt routine could observe the value momentarily stored there; another is that code could be written that depends on the action of one or other half of the instruction in isolation: one loads a certain pattern of bits into the `lr` register, and the other branches to a subroutine by computing an address from the contents of `lr` and some bits from the instruction. Alternatively, a program might have other operations between the two halves of the instruction. A program that depended on these separate behaviours of the halves of the instruction would fail in an implementation that required them both to appear together.

[3] With the same number of instructions, we could compute `3*r1` using a shift and an add:

```
lsls r2, r1, #1
adds r1, r1, r2
```

That would be better on a machine with a slower multiplier, or none at all.

2.7 We can access characters of the string *either* by keeping a pointer to the start, and using an integer index that increases from 0 to the length of the string, *or* by incrementing a pointer so that it points to successive characters. I've chosen the former, allowing us to keep a pointer to the start of the string, but needing an extra register for the index. The subroutine still uses only `r0` to `r2`, so there is no need to save registers.

There's no need, even in assembly language, to substitute the constants `'a'` = `97`, `'A'` = `65` or `'z'-'a'` = `25` into the program. Let the assembler to the work!

```
    .thumb_func
toupper:
    movs r1, #0              @ Index starts at zero
    b loop

examine:
    subs r2, r2, #'a'        @ Compute c - 'a'
    cmp r2, #('z'-'a')       @ Compare with 'z' - 'a'
    bhi skip                 @ Skip it if not in range

    adds r2, #'A'            @ Compute upper-case equivalent
    strb r2, [r0, r1]        @ Overwrite the character

skip:
    adds r1, r1, #1          @ Increment index

loop:
    ldrb r2, [r0, r1]        @ Fetch next character c
    cmp r2, #0               @ If it is non-zero,
    bne examine              @   go to examine it

    bx lr
```

This program uses a trick for the range test `'a'` $\leq c \leq$ `'z'`: if we compute the quantity $c -$ `'a'` and $c <$ `'a'`, then the result will be negative, but an unsigned comparison will treat it as a large positive number. Comparing $c -$ `'a'` with `'z'` $-$ `'a'` can therefore do the job with a single comparison. This trick is commonly used to implement subscript bounds checking in the code output by compilers.

2.8 For a recursive implementation of `fac(n)`, we will need to save some registers and carefully preserve the value of `n` across the recursive call.

```
facrec:
    push {r4, lr}
    movs r4, #1
    cmp r0, #0
    beq base

    movs r4, r0
    subs r0, r0, #1
    bl facrec
    muls r4, r0

base:
    movs r0, r4
    pop {r4, pc}
```

I've done my best here to make the implementation fast whilst adhering to the convention that there be only one pop instruction at the end of the routine. The result is computed in `r4` and moved into `r0` just before returning; `r4` is set to the default value of 1 before testing whether the argument `n` is non-zero.

An further optimised version avoids saving registers in the base case $n = 0$, deviating slightly from the normal outline of a subroutine.

```
facrec2:
    cmp r0, #0
    beq base

    push {r4, lr}
    movs r4, r0
    subs r0, r0, #1
    bl func
    muls r0, r4
    pop {r4, pc}

base:
    movs r0, #1
    bx lr
```

For comparison, here is an implementation that uses a loop, again tuned for speed, so that the loop is three instructions taking five cycles.

```
facloop:
    movs r1, #1        @ Set f to 1
    cmp r0, #0         @ If n = 0
    beq done           @   finished

again:
    muls r1, r0        @ Multiply f by n
    subs r0, #1        @ Decrease n by 1
    bne again          @ If still not zero, repeat

done:
    movs r0, r1        @ Put result in r0
    bx lr              @   and return
```

Measurements made on a genuine, vintage V1 micro:bit show that the first recursive version takes $20n + 16$ cycles to compute $n!$, and the optimised version takes $18n + 9$ cycles, but the iterative version takes only $5n + 6$ cycles for $n > 0$, a clear win.[4] In addition, of course, the iterative version runs in constant space, whereas the recursive version requires an amount of stack space proportional to n.

A good example of recursion? Answers on a postcard please! I would incline towards something that naturally involves a tree structure. Perhaps Quicksort is the best.

2.9 `GPIO` expands into a cast that interprets the constant `0x50000500` as a pointer to a `_gpio` structure, with a field `IN` at offset 16. `BUTTON_A` and `BUTTON_B` are the constants 17 and 11 on the V1 micro:bit, and `GET_BIT(x, n)` is an abbreviation for `((x >> n) & 0x1)`, and expression that first shifts `x` right by `n` bits, then isolates the LSB of the result by ANDing with a mask. So the test is synonymous with

```
if (((x >> 17) & 0x1) == 0 || ((x >> 11) & 0x1) == 0)) ...
```

Looking at the object code, however, we see the following (at least with the version of `gcc` on my machine).

```
1a4:  4b06      ldr   r3, [pc, #24]   ; (1c0 <init+0xb4>)
1a6:  691b      ldr   r3, [r3, #16]
1a8:  4a08      ldr   r2, [pc, #32]   ; (1cc <init+0xc0>)
```

[4] On the V2, the corresponding figures are $18n + 15$ and $16n + 7$ for the two recursive versions, and $4n + 6$ for the iterative version.

```
1aa:  4013        ands   r3, r2
1ac:  4293        cmp    r3, r2
1ae:  d1e2        beq.n  176 <init+0x6a>
...
1c0:  50000500       .word   0x50000500
...
1cc:  04020000       .word   0x04020000
```

This renders the test as

```
if ((x & 0x04020000) != 0x04020000) ...
```

This uses a constant `0x04020000` in which bits 11 and 17 are set, corresponding to the two bits associated with the buttons. So all the shifting and almost all of the masking has been optimised away. Modern compilers have extensive optimisation passes that are capable of simplifications like this, freeing us from the need to worry too much about how our code looks in low-level detail, even in cases where micro-optimisations like this make a discernable difference to the bigger picture. In such cases, the example also underlines the value of being able to read the compiler output and check it is reasonable. Next year's compilers course will not amass a large enough collection of rules to cover cases like this, but the methods used there allow simplifications like this to be expressed.

3.1 (a) We always have $0 \leq bufin < NBUF$ and $0 \leq bufout < NBUF$, so that both variables are valid indices into the buffer array. What's more, $bufin \equiv bufout + bufcnt \pmod{NBUF}$, so the value of *bufin* could be deduced from *bufout* and *bufcnt*.

A better approach is to store *bufin* and *bufout* without reducing them modulo *NBUF*, and make *bufnct* be the difference between them. Now *bufin* is modified only by `serial_putc` and *bufout* only by the interrupt handler, so we don't need to disable interrupts to prevent interference from concurrent modification. A subtle point: even if we let arithmetic on the two variables overflow, the subtraction $bufin - bufout$ still gives the correct result. Note that we still need to disable interrupts for another reason outlined in part (b).

(b) If the main program finds that *txidle* is false, then it assumes the UART is busy, and the waiting character must be put in the buffer. Unless we disable interrupts, however, it is possible for the UART handler to be invoked between testing *txidle* and incrementing *bufcnt*, to find the buffer empty, and to go to sleep by setting *txidle* to true. This leads to a situation where the UART is idle but the buffer is non-empty; the situation will resolve itself next time `serial_putc` is called, but the characters will come out in the wrong order.

(c) The variable *bufcnt* is modified by both `serial_putc` and the interrupt handler. Since both `bufcnt++` and `bufcnt--` are implemented by separate read, modify and write instructions, it's possible for both processes to read the variable, both to modify their copies in different ways, then one to write it back before the other. Even if memory reads and writes are atomic, this will mean one of the updates will be lost, and the count of how many characters are in the buffer will ever afterwards be wrong.

(d) This is very subtle, but important: `wfi` waits the an interrupt to happen, while `wfe` waits for the event flag to be set. Any interrupt sets the event flag, and it is only cleared when a `wfe` instruction is executed. (The event flag is also set in multi-processor systems when any processor executes a `sev` instruction, but that is irrelevant in a single-processor system like the micro:bit.)

To see why a `wfi` instruction is wrong, consider the scenario where `serial_putc` finds the buffer full, and enters the body of its wait loop. If a UART interrupt intervenes between the decision to enter the loop body and the wfi instruction, then it can make space in the buffer, but the instruction will still block. This need not be serious (provided the buffer is bigger than one element) because a subsequent UART interrupt will unblock it. But the problem is avoided by using wfe instead: the first time wfe is executed, it will clear the event flag and continue; the second time, it will wait, unless an interrupt has intervened and potentially made space in the buffer.

(e) Provided there is at least one space in the buffer, the assignment txbuf[bufin] = ch is benign, and can be executed anyway. Likewise, the assignment bufin = (bufin+1) % NBUF affects only a variable that is not shared with the interrupt handler. So interrupts need to be disabled only for the brief interval between checking txidle and incrementing bufcnt.

```
void serial_putc(char ch) {
    while (bufcnt == NBUF) pause();
    txbuf[bufin] = ch;

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

A minor point: after restarting the UART, about 1 msec will elapse before there is another interrupt, so it's probably safe to leave the assignment to txidle after the assignment that puts a character in UART_TXD. If we do not wish to depend on the timing in this way, we can swap the two assignments as shown above, and make the program safe for (a) machines that have a hardware transmit buffer and interrupt immediately for another character, and (b) circumstances where the machine is nearly swamped by other, higher-priority interrupts.

No doubt trickier schemes can be devised that avoid disabling interrupts at all, but the point here is really that this kind of high-octane reasoning is best confined to very small contexts. We can write safe and efficient programs and stay sane by confining interrupts to the very lowest levels in a design: this is part of what an operating system can do for us.

3.2 The chief limitation of shoe-horning all the logic into the interrupt handler is that the handler is just a subroutine, and can have no control state that persists from one invocation to the next. In short, the triply-nested loop – a cycle of four images, each consisting of several frames, and each frame containing three rows – must be unrolled into a single loop.

It helps greatly to introduce tables that make the data uniform, so that our position in the performance can be identified by three coordinates (i, j, k), where $0 \le i < 4$ and $0 \le j < dur[i]$ and $0 \le k < 3$. Each iteration then increments this coordinate triple like a multi-digit counter.

```
static const unsigned heart[] = {
    0x28f0, 0x5e00, 0x8060
};
```

```
static const unsigned small[] = {
    0x2df0, 0x5fb0, 0x8af0
};

static const unsigned *frame[] = {
    heart, small, heart, small
};

static const int dur[] = {      /* Duration of each frame */
    70, 10, 10, 10
};

void advance(void) {
    static unsigned i = 0;      /* Index in frame */
    static unsigned j = 0;      /* Range [0..dur[i]) */
    static unsigned k = 0;      /* Index in frame[i] */

    k++;
    if (k == 3) {
        k = 0; j++;
        if (j == dur[i]) {
            j = 0; i++;
            if (i == 4) i = 0;
        }
    }

    GPIO_OUT = frame[i][k];
}
```

Now we just need to call `advance()` from the interrupt handler for the timer.

```
void timer1_handler(void) {
    if (TIMER1_COMPARE[0]) {
        advance();
        TIMER1_COMPARE[0] = 0;
    }
}
```

This solution was feasible because the timings all fall into a regular, predictable pattern, and the loop structure of the original program was also simple and regular. Anything more complex would quickly become overwhelming.

[The code above applies to the V1 micro:bit where all LED bits appear in a single GPIO register. For the V2 board, each assignment to `GPIO` needs to be adjusted.]

3.3 A good design is to have an interrupt handler looking after the random number generator, with some kind of buffer between it and the main program. The interrupt handler can put random bytes into the buffer, throwing bytes away when the buffer is full. We can safely say that no information is lost in doing this, though there is a loss of eight bits of entropy.

For the sake of variety, we can make the buffer a stack rather than a queue, using a single integer variable to keep track of the number of samples stored.

```
#define NRAND 64

static volatile unsigned char randoms[NRAND];
static volatile unsigned nrand = 0;
```

The interrupt handler expects to be called only on the event `RNG_VALRDY`, but we check for this as a matter of routine.

```
void rng_handler(void) {
        unsigned char val;

        if (RNG_VALRDY) {
            val = RNG_VALUE;
            RNG_VALRDY = 0;
            if (nrand < NRAND) {
                randoms[nrand] = val;
                nrand++;
            }
        }
    }
```

Clients of the random number generator can share a common subroutine that delivers a single random byte. We must pause if there is no byte in the buffer, and then (with interrupts disabled) pop one byte from the stack and return it.

```
    unsigned char randbyte(void) {
        unsigned char val;

        while (nrand == 0) pause();

        intr_disable();
        nrand--;
        val = randoms[nrand];
        intr_enable();

        return val;
    }
```

We can use the one-liner `val = randoms[--nrand]` here, just as we could use `randoms[nrand++] = val` above.

For completeness, here is an initialisation function that sets up the RNG and enables the appropriate interrupt. The handler `rng_handler` is identified by name: defining a function with that name links it into the vector table in place of the default handler that would otherwise occupy the slot.

```
    void rng_init(void) {
        SET_BIT(RNG_CONFIG, RNG_DERCEN); // Correct for bias
        RNG_VALRDY = 0;
        RNG_START = 1;

        RNG_INTENSET = BIT(RNG_INT_VALRDY);
        set_priority(RNG_IRQ, 3);
        enable_irq(RNG_IRQ);
    }
```

The bias correction uses an algorithm to ensure that 0 and 1 are equally likely in each bit position of the output: the procedure is to generate pairs of bits repeatedly, setting an output bit to 0 for 01 and 1 for 10, and trying again if the bits are 00 or 11.

To generate a random four-byte integer, we just fetch four random bytes and combine them:

```
    unsigned randint(void) {
        int k
        unsigned v = 0;

        for (k = 0; k < 4; k++)
            v = (v << 8) + randbyte();

        return v;
    }
```

To generate perfectly random dice rolls, we generate a random byte and assign 1 to the scores 0–41, and 2 to the scores 42–83, and so on up to 6 for the scores 210–251. If the byte is 252 or more, then we go round again, in an algorithm that (like the bias-removal algorithm in the hardware) terminates with probability 1.

```
#define QUANT (256/6)

unsigned roll(void) {
    unsigned char val;

    do {
        val = random_byte();
    } while (val >= 6*QUANT);

    return val/QUANT+1;
}
```

It's fun to make a main program that generates and prints a histogram of the random values.

3.4 We must assume that link register, like all the others, is unchanged by the interrupt, so that we can save it on the stack. For tidiness, let's use the same exception frame layout as the real machine. Note, however, that we can't store `r12` directly, nor directly reload `r12` or `lr`, so we must carefully save the low registers before using one of them as a staging post for saving `r12`, and use them again for the same purpose before restoring them.

```
.thumb_func
uart_handler:
    push {lr}
    sub sp, #4
    push {r0-r3}
    mov r0, r12
    str r0, [sp, #16]
    bl real_uart_handler
    ldr r0, [sp, #16]
    ldr r1, [sp, #20]
    mov r12, r0
    mov lr, r1
    pop {r0-r3}
    add sp, #8
    rti
```

(Sorry, that was more messy than I intended!)

We can save a bit of the mess (and make better use of the push instruction) by changing the layout a bit.

```
.thumb_func
uart_handler:
    push {r0-r3}
    mov r0, r12
    push {r0, lr}
    bl real_uart_handler
    pop {r0, r1}
    mov r12, r0
    mov lr, r1
    pop {r0-r3}
    rti
```

Note that in neither case do we need to save registers r4-r7, because we can assume that the C function real_uart_handler preserves them.

3.5 The suggestion would prevent the return address of a function being overwritten by overrunning a buffer within its own stack frame, but it would not prevent the return address of a different function being overwritten. For example, suppose a function getnum() is written as follows. (Code taken from the lecture.)

```
/* getnum -- read a line of input and convert to a number */
int getnum(void)
{
   char buf[64];
   getline(buf);
   return atoi(buf);
}
```

With a stack that grew upwards, the stack frame for getline() would appear just above that for getnum(), like this:

```
    +--------------------------------+
sp: |                                |
    | Locals of getline              |
    |                                |
    +--------------------------------+
    | Return addr of getline         |
    +--------------------------------+
    |                                |
    | 8 words for array buf          |
    |                                |
    +--------------------------------+
    | Return addr of getnum          |
    +--------------------------------+
```

(with perhaps some padding for alignment.) The getline() function does not know the size of buffer it has been passed, so cannot prevent overflow of the buffer. Overrunning the buffer could replace the return address of getline with an address in the buffer, so getline would return not to getnum but to our malicious code. The situation is no better.

4.1 When a fake exception frame for a process is created by start, the return address is set to the address of the function exit. Thus, if the process body returns, exit runs and the operating system marks the process as DEAD, never to be scheduled again.

If all the processes in the system do this, then the only process left will be the idle process, which never leaves state IDLING. So the idle process will run and execute pause(), equivalent to the wfe instruction that halts the processor until the next interrupt. At this point, very little more can happen before the processor goes to sleep forever. A few more interrupts can occur once each, with each interrupt setting the p_pending flag on its now-dead handler process before being disabled: being already dead, the handler never gets to run. But once that is over, the last wfe sleeps the big sleep.

4.2 The running process is always in state READY. In the code for mini_send, we see that if the destination of the message is not in state RECEIVING, then the state of the sender is set to SENDING and it is hung on the end of the destination's queue, and next the scheduler is called to pick a new process to run. The original process is left in a state where it is sleeping, and the only process that can wake it is the process itself. In short, the only message a process can send to itself is a suicide note. This is a very special case of the kind of deadlock that occurs when there is a ring of processes, each waiting to send to the next: see Exercise 4.3.

4.3 If there is a cycle in the graph, then each process in the cycle is waiting to send to the next one, and none of the processes are prepared to receive a message. The result is deadlock: none of the processes involved can make any further progress.

We could detect this kind of deadlock by augmenting the `send` operation: whenever a process *A* tries to send to a process *B*, we can see whether *B* is already waiting to send to another process *C*, and follow the chain to see if it leads back to *A*. This will not prevent the deadlock, but will at least allow us to be informed of it. The detection process could be aided by recording in the process table for processes waiting to send which process they are sending to – the inverse of the relation recorded in the queue of each receiving process.

Deadlock is also possible because of selective `receive`: for example, if *A* waiting to send a message of type t_1 to *B*, but *B* is waiting for a message of type t_2 that no process other than *A* could send, then the two processes are also deadlocked.

4.4 The results depend on the order in which the three processes get to run – the two shown, plus the driver process for the UART. If process 2 gets to run before process 1, then the non-preemptive nature of the scheduling means that it will run to completion before anything else happens, and the program will display 100,000 ten times.

If process 1 gets to run first, the story is a bit more complicated: it will certainly capture the value 0 and start to print it, soon sending a message to the UART driver to print the first part of the output, and waiting for the result. At this point, the UART driver runs because it has higher priority, and when it has sent a character to the UART, it suspends, waiting to receive another message. At this point, the scheduler picks process 2 to run, and it starts to increment `r`. Now that all three processes have started, the scheduler will switch between them, stimulated by the fact that the UART driver will run whenever there is an interrupt, or whenever process 1 blocks waiting for a reply from it; and when the driver blocks, the scheduler will pick processes 1 and 2 more-or-less in alternation. In practice, the values printed go up in increments of 15,000 or so, with the last few equal to 1,000,000.

After all ten values have been printed, both processes exit, and the program ends up running the idle process, allowing the last few characters to drain from the UART buffer, before going into an everlasting sleep.

The point of the exercise is that (the above reasoning aside) the results are hard to predict and dependent on details of scheduler behaviour and rates of process execution.

4.5 (a) If a process is continuously outputting, we can assume the output buffer is constantly full, the process is waiting to send a character to the serial driver, and the serial driver has accepted a previous `PUTC` request and is waiting for an interrupt message. The processor is waiting in the idle process.

(1) When the interrupt arrives, there is a context switch from the idle process to the serial driver. The driver then restarts the UART, freeing a character slot in the buffer, which it immediately fills with a character.

(2) Next, the driver accepts the waiting `PUTC` request, making the outputting process ready again, and the driver continues to run. I'm counting this as a context switch because the system must enter the kernel to satisfy the `receive` call.

(3) The driver blocks for another interrupt. Now the outputting process can run until it blocks with another `PUTC` request.

(4) The context switches to the idle process again.

That makes four context switches per character output.

(b) If each context switch takes 20 μsec, that's 80 μsec in all. At 9600 baud, transmission of a character takes 1 msec, so that's an 8% overhead for the context switches. At 38400 baud, the overhead would rise to 32%, which would be noticeable.

(c) The number of context switches can be practically halved by introducing a local buffer specific to the output process in addition to the shared buffer managed by the driver process. The outputting process can add to its local buffer without context switches; when it is full, the local buffer can be passed as a unit to the serial driver, which can then copy the characters into its shared buffer, again without context switches for each character. What remains is the two context switches to and from the driver process in response to each interrupt from the UART. The current implementation of `printf` supports this improvement, with an underlying interface

```
extern void print_buf(char *buf, int n);
```

that allows `printf` to pass many characters at once to the device driver. Micro:bian's serial driver comes with an implementation of `print_buf` that hands the whole buffer to the driver process in one message, then waits for a reply that indicates it has all been copied into the driver's own internal buffer.

4.6 The initial program alternates on a character-by-character basis because both speakers spend almost all their time queueing to send a character to the serial driver task. When the process at the head of the queue succeeds in sending a message, it runs for only a brief moment before joining the queue again at the back.

We can introduce a multiplexer process that accepts slogans from the two speakers and prints each one fully before accepting another slogan.

```
static int MUX;

#define PUTS 99

void mux(int n) {
    message m;

    while (1) {
        receive(PUTS, &m);
        put_string(m.ptr1);
    }
}
```

The `speaker` function then becomes the following.

```
void process(int n) {
    message m;

    while (1) {
        m.ptr1 = slogan[n];
        send(MUX, PUTS, &m);
    }
}
```

Then we need only start the multiplexer with the speakers during startup.

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

```
        start("Farage", speaker, 1, STACK);
        MUX = start("Kearney", mux, 0, STACK);
    }
```

Now the two speakers spend almost all their time queueing to pass a note to the presenter for her to read out, so they alternate line-by-line instead of character-by-character.

This solution is safe only because the speakers are repeatedly spouting fixed slogans that never change. If the slogans were computed dynamically –

```
    sprintf(buf, "%d Million a week for the NHS!\n", lie++);
```

– then we would have to be sure the presenter had finished reading out one slogan from `buf` before overwriting it with the next. This can be ensured by having the presenter send a reply back to the speaker after reading the slogan. Though the variable `buf` might be declared locally to the speaker, ownership of the storage space effectively passes back and forth between the presenter and the speaker.

An alternative solution does not require all remarks to be passed as notes to the presenter, but requires each speaker to get permission before speaking, and to tell the presenter when they have finished.

```
    #define REQUEST 42
    #define RELEASE 43

    void presenter(int n) {
        message m;
        while (1) {
            receive(REQUEST, &m);
            speaker = m.sender;
            receive(RELEASE, &m);
            assert(m.sender == speaker);
        }
    }
```

Then replace the `put_string` call in the `speaker` function with

```
    send(PRESENTER, REQUEST, NULL);
    put_string(slogan[n]);
    send(PRESENTER, RELEASE, NULL);
```

Note that in both cases, the presenter process holds waiting speakers on its implicit queue of processes waiting to send, and doesn't need to manage an explicit queue; that works so long as the presenter is content with a first-come-first-served rule. As soon as she wants to control who speaks next by some other rule, the process will need to accept requests, store them in some explicit queue, and in the second solution ask speakers not to start speaking until she has sent them a `GRANT` message.

4.7 If the destination is waiting to receive (and the source is acceptable to it), then the message can be transferred immediately, the destination becomes ready, and the source process can be put in state `RECEIVING` to await the reply.

If the destination is not in state `RECEIVING` (or if the message is not acceptable), then the source process must be put on its queue, and must enter a new state that we can call `BOTH`. When the destination eventually accepts the message, the source process (instead of entering state `READY`) is put in state `RECEIVING` in order await the reply.

With a normal `send`/`receive` pair, the sending process must run again in order to reach the `receive` call, before the process that received the original request can send the reply. The saving in `SENDREC` is that the sending process is immediately waiting for the reply, and does not need to run again first. In addition, `sendrec` avoids the possibility of *priority inversion*: if the server process has a higher priority than the client, then separate calls to send() and receive() could create a situation where the server cannot run until a lower priority process (the client) has been scheduled, so that it can reach the receive() call and accept the reply from the server.

4.8 Let's assume that message types are small integers in the range $\{0..31\}$. We can then represent sets of types by single-word bitmaps, and implement the test $k \in S$ by

```
#define member(k, S) ((S) & BIT(k) != 0)
```

At present, if a process in in state RECEIVING, the acceptable senders are recorded in its p_accept field, which contains either the pid of a specific sender or the magic value ANY $= -1$. We could replace the p_accept field by a set-valued field p_oktypes represented as above.

There are several places where the operating system tests whether a message of type k is acceptable to a process p, using a test that invariably has the form

```
p->p_accept == ANY || p->p_accept == k
```

Such test occur in mini_receive (to see whether an acceptable sender is already waiting) and in mini_send (to see whether the receiving process will accept a message from the caller), as well as mini_sendrec and interrupt. In each case, the test can be replaced with

```
member(k, p->p_oktypes)
```

We can enhance the receive call to take a set-valued parameter senders:

```
void receive_set(unsigned typeset, message *msg);
```

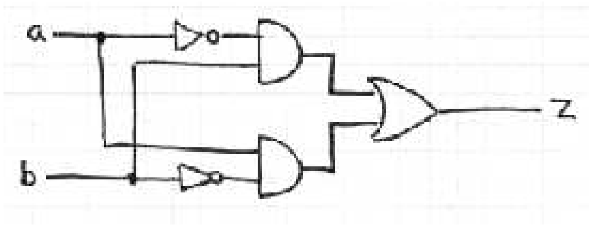
Then the existing interface for receive can be re-implemented in terms of the enhanced one:

```
void receive(int type, message *msg) {
    unsigned typeset =
        (type == ANY ? (unsigned) (-1) : BIT(type));
    receive_set(typeset, msg);
}
```

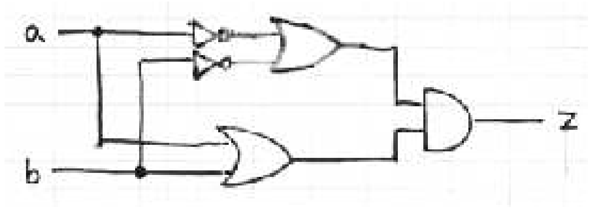
It would be an amateurish mistake to keep the name receive for the changed interface, because that would break all existing client programs.

5.1 (a) Use truth tables.

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



(c) Use de Morgan or truth tables to obtain $z = (a \vee b) \wedge (\neg a \vee \neg b)$.



(d) Use a truth table again, or algebraic reasoning:

$$\begin{aligned} v &= \neg(a \wedge \neg(a \wedge b)) = \neg(a \wedge (\neg a \vee \neg b)) \\ &= \neg((a \wedge \neg a) \vee (a \wedge \neg b)) = \neg(a \wedge \neg b) \end{aligned}$$

and similarly $w = \neg(b \wedge \neg a)$. So

$$z = \neg(v \wedge w) = \neg v \vee \neg w = (a \wedge \neg b) \vee (b \wedge \neg a) = a \oplus b.$$

5.2 (a) A NAND gate turned upside down!

(b) The gate from the lecture turned upside down, because

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

so z and w are dual.

(c) If two functions f and g are dual, in the sense that $f(\neg a, \neg b) = \neg g(a, b)$, then a circuit for f can be obtained by inverting (drawing upside-down) a circuit for g. Each p-type in the inverted circuit conducts exactly when the corresponding n-type in the original circuit does not conduct, and so the pull-up network as a whole pulls the output of the inverted circuit up to 1 exactly when the inputs have the opposite values from those that would pull the output down to 0 in the original circuit. Since the n-type and p-type networks in the original circuit are complementary, so are those in the inverted circuit.

5.3 (a) We need a single bit x of state that agrees with the output.

```
x = 0;
forever {
    z = x;
    pause;
    x = (x || a) && !b;
}
```

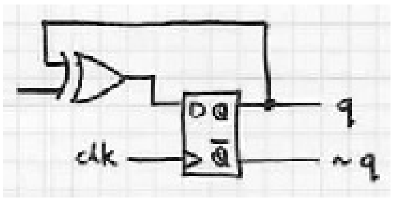
Since $a = b = 1$ is disallowed, we could write `(x && !b) || a` instead.

(b) For the enhanced circuit, we should record the previous value of x as a state variable y; then the output w is given by the expression `x && !y`.

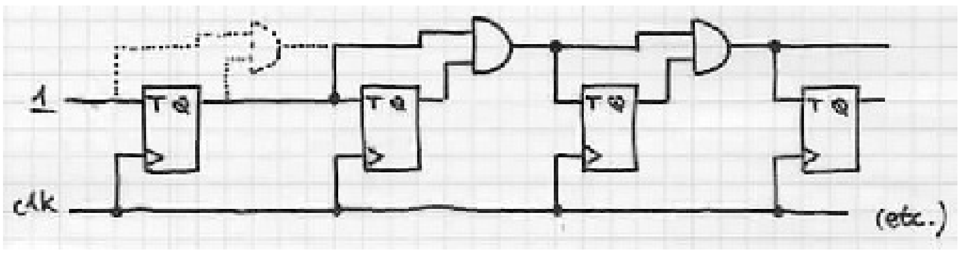
```
x = y = 0;
forever {
    w = x && !y;
    pause;
    y = x; x = (x || a) && !b;
}
```

Both outputs are computed by the following circuit. [Picture omitted].

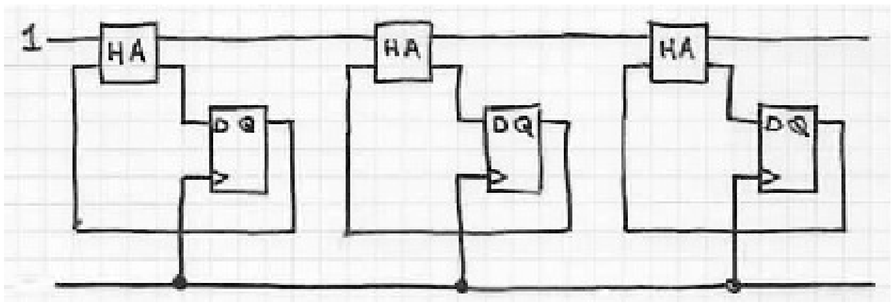
5.4 (a) The T-type satisfies $\hat{q} = t \oplus q$:



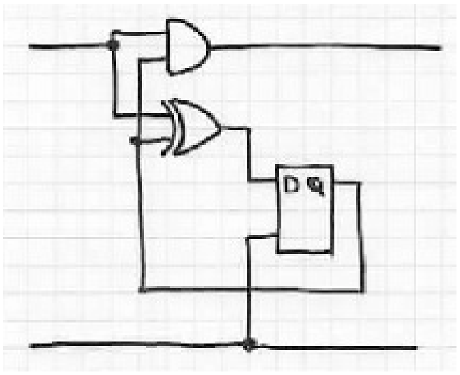
(b) Stage k should change state exactly if stages 0 up to $k - 1$ are all 1:



(c) Each half-adder combines the present value of stage k with the carry from stage $k - 1$ to obtain the new value of stage k and a carry that goes to stage $k + 1$:



(d) Each stage in the circuit based on half-adders can be viewed as:



The XOR gate is connected in the same way as in our implementation on the T-type flip-flop, and the AND gate is as in the counter built from T-types. Just untangle the wiring!

5.5 The existing design implements the program

```
x = y = 0;
forever {
    z = y;
    pause;
    if (a) y = !x; else x = y;
}
```

In this program, y becomes 1 as soon as the input a goes high, and x becomes 1 then a goes low again. Symmetrically, y becomes 0 again as soon as a goes high a second time. If we want the output z to be high as described, it's enough to set $z = x \wedge y$, replacing the direct connection between y and z with an AND gate.

5.6

(a) Writing $\tilde{\vee}$ for NOR, we have $\neg x = x \mathbin{\tilde{\vee}} x$ and $x \vee y = \neg(x \mathbin{\tilde{\vee}} y)$ and $x \wedge y = \neg x \mathbin{\tilde{\vee}} \neg y$. So adequacy of $\{\tilde{\vee}\}$ follows from that of $\{\wedge, \vee, \neg\}$.

(b) Let's write *abcd* for the Boolean function $f(x, y)$ such that $f(0, 0) = a$, $f(0, 1) = b$, $f(1, 0) = c$ and $f(1, 1) = d$. Consider the set that contains $0 = 0000$, $1 = 1111$, $x = 0011$ and $y = 0101$, together with $\neg x = 1100$, $\neg y = 1010$, $x \oplus y = 0110$ and $\neg(x \oplus y) = 1001$. Negating any one of these, or taking the XOR of any two, gives a function also in the set. Consquently, the Boolean function $x \wedge y = 0001$ is not expressible.

5.7 (a) Popcount is the identity function on individual bits. We can combine the results for the bits in pairs using one-bit adders to obtain $n/2$ two-bit numbers, then combine these in pairs to obtain $n/4$ numbers with three bits each, and so on, finally obtaining the answer for the whole input at the root of the tree.

(b) We get $S(n) = 2S(n/2) + O(\log n)$ and $T(n) = T(n/2) + O(\log n)$. For $S(n)$, the Master Theorem[5] tells us that the cost in dominated by the leaves of the recursion tree, and $S(n) = O(n)$. For $T(n)$, replacing $O(\log n)$ by $c \log n$ and unfolding, we get

$$T(n) \le c(1 + 2 + \ldots + \log n) = O(\log n)^2.$$

To prove directly the bound on $S(n)$, let $s_m = S(2^m)$, and for simplicity set the hidden constant in $O(\log n)$ to 1. From the recurrence $s_m = m + 2s_{m-1}$ with $s_0 = 0$ we can prove by induction that $s_m = 2^{m+1} - m - 2$, and so $S(n) = O(n)$. The constants $A = 2$, $B = -1$, $C = -2$ in the conjecture $s_m = A2^m + Bm + C$ can be discovered by considering small examples with, say, $m = 0, 1, 2$.

Alternatively, unfold to get

$$s_m = m + 2.(m - 1) + 4.(m - 2) + \cdots + 2^{m-2}.2 + 2^{m-1}.$$

Now write out the terms in the sum $2s_m$, and underneath write the terms for s_m, shifting the terms by one place.

$$\begin{aligned} 2s_m &= \phantom{m + {}} 2m \phantom{{}-1)} + 4(m-1) + 8(m-2) + \cdots + 2^{m-1}.2 + 2^m \\ s_m &= m + 2(m-1) + 4(m-2) + 8(m-3) + \cdots + 2^{m-1} \end{aligned}$$

Subtracting gives

$$s_m = -m + 2 + 4 + 8 + \cdots + 2^{m-1} + 2^m = 2^{m+1} - m - 2.$$

As another alternative, write

$$t_m(x) = x^m + x^{m-1} + \cdots + 1 = \frac{1 - x^{m+1}}{1 - x},$$

and set

$$u_m(x) = \frac{d}{dx} t_m(x) = mx^{m-1} + (m-1)x^{m-2} + \cdots + 1,$$

so that

$$u_m(x) = \frac{d}{dx}\frac{1 - x^{m+1}}{1 - x} = \frac{1 - (m+1)x^m + mx^{m+1}}{(1 - x)^2}.$$

Now notice that $s_m = 2^{m-1} u_m(1/2)$, so

$$s_m = \frac{2^{m-1}\,(1 - (m+1)/2^m + m/2^{m+1})}{1/4} = 2^{m+1} - 2(m+1) + m.$$

This and similar sums frequently crop up in the analysis of algorithms, for example, in calculating the expected number of comparisons in binary search.

[5] To apply the Master Theorem, we note that $S(n) = aS(n/b) + f(n)$ where $a = 2$, $b = 2$, and $f(n) = O(\log n)$. The critical growth factor is $C = \log_b a = 1$, and we can conclude that $S(n) = O(n^C) = O(n)$ provided $f(n) = O(n^c)$ for some $c < C$. In fact, $f(n) = O(\log n) = O(n^\epsilon)$ for any $\epsilon > 0$, so the job is done.

(c) In truth, the combinational paths from full adder to full adder in the circuit move only upwards towards the root of the tree and leftwards towards more significant bits, so the longest path has length about $2 \log n = O(\log n)$, and this is the delay of the circuit. Indeed, if we take the constant of proportionality for the delays mentioned in the question to be 1, then output bit i of each of the adders in level j of the circuit is ready at time $i + j$, and consequently all bits up to the most significant bit of the last level are ready by time $2 \log n \pm 1$.

5.8 (a) If $a_i \neq b_i$, then comparing them determines the next output; otherwise $a[0\,..\,i+1)$ compares with $b[0\,..\,i+1)$ in the same way $a[0\,..\,i)$ compares with $b[0\,..\,i)$. This leads the formulae,

$$L_{i+1} = (\neg a_i \wedge b_i) \vee ((a_i \equiv b_i) \wedge L_i),$$

and

$$G_{i+1} = (a_i \wedge \neg b_i) \vee ((a_i \equiv b_i) \wedge G_i)$$

These simplify to

$$L_{i+1} = (\neg a_i \wedge b_i) \vee (\neg a_i \wedge L_i) \vee (b_i \wedge L_i) = maj(\neg a_i, b_i, L_i),$$

and

$$G_{i+1} = (a_i \wedge \neg b_i) \vee (a_i \wedge G_i) \vee (\neg b_i \wedge G_i) = maj(a_i, \neg b_i, G_i).$$

(b) The formulae above suggest the sequential circuit (with inputs a and b and outputs L and G) that we can write,

```
L, G = 0, 0;
forever {
    pause;
    L, G = maj(!a, b, L), mag(a, !b, G);
}
```

(c) If the inputs come MSB-first, then a conclusion about which is greater can be made as soon as the two inputs differ. It should then stick forever after, giving the equations,

$$L_{i+1} = L_i \vee (\neg a_i \wedge b_i \wedge \neg G_i),$$

and

$$G_{i+1} = G_i \vee (a_i \wedge \neg b_i \wedge \neg L_i).$$

6.1 (a) If *cRegSelC* = *Rx* always, then the register that is written by most instructions (and the register providing the data for a store instruction) will always be selected by bits *instr*⟨2:0⟩. For many instructions, this is already the case, but others use a different instruction field or a fixed register. For example, the form add i8 uses the field *instr*⟨10:8⟩ selected by *Rw*, and branch instructions use the PC; these instructions will no longer work. If *cRegSelC* = *Rw* always, then most instructions that write a register will no longer work, but adds i8 will work still.

(b) *cRand2* = *RegB* always, then those instructions that contain an immediate field as the second operand will no longer work. That includes arithmetic instructions, but also loads and stores containing a numeric offset, and branches that use the ALU to add a displacement to the PC. These instructions will read a nonsensical register and use its value in place of the immediate constant. If *cRand2* = *Imm8*, then any instruction that does not use an 8-bit immediate will wrongly use bits *instr*⟨7:0⟩ as the operand, creating nonsense.

(c) If *cMemRd* = *F* always, it is load instructions that will not work: instead of fetching a word from memory and writing it to a register, they will instead write the address of the word. If *cMemRd* = *T* always, then every instruction will use the ALU output as an address and try to load from it, and only actual load instructions will work correctly.

(d) If *cWReg* = *N* then no instruction will write its result to a register; if it is *Y*, then every instruction will do so. In the latter case, branches will work (they write the PC), but conditional branches will become unconditional. Store instructions will work, but will overwrite the value stored with the computed address. Compare instructions will write the condition codes, but also overwrite one of their operands with the result of the subtraction.

(e) If *cWFlags* = *F* always, then no instruction will write the flags. If it is *T* always, then every instruction will write the flags, but not always with a sensible value: for loads and stores, for example, the flags will reflect attributes of the computed address, not of the value read from or written to memory.

6.2 If the displacement is negative, say −8, then this is represented with the bit-pattern *a* = 111 1111 1111 in the `bl1` instruction, and *b* = 111 111 1100 in the `bl2` instruction. A modern implementation would concatenate these, shift left by one place, and sign extend from 23 to 32 bits. To take the two half-words one at a time, we can shift left *a* by 12 bits and sign extend it, shift left *b*, and add both on to the starting place for the branch. For example, starting from address 32:

```
0000 0000 0000 0000 0000 0000 0010 0000  Base = 32
1111 1111 1111 1111 1111 0000 0000 0000  signext(a << 12)
                         1111 1111 1000  b << 1
---------------------------------------
0000 0000 0000 0000 0000 0000 0001 1000  Result = 24
```

Sign extending both halves would just give the wrong answer.

6.3 The instruction writes the PC conditionally, allowing it to increment as normal if the branch is not taken. But two factors make it significantly different from other conditional branches. First, the condition is based on a comparison performed in the instruction itself, not on the condition codes established by an earlier instruction. Second, the instruction needs both to make a comparison and to compute the branch target address in the same clock cycle, and we have been using the ALU for both purposes.

Taking the second problem first, we could alter the decision to compute branch targets using the ALU, and install a separate adder instead, perhaps combining it with the “+2” adder that is normally used to increment the PC, or anyway putting it after that adder. We will still need to be able to write the PC from another register in order to implement the instructions `bx reg` and `blx reg`, but may give up on implementing the (deprecated) instruction `mov pc, r3`.

With a separate adder to compute branch targets, we can use the ALU to perform the subtraction `r2 - 0` and feed the resulting Z flag (part of `newflags`) to a mux that determines the next PC value. Control circuitry will be needed to allow branches to be taken unconditionally, conditionally on the `enable` signal from the Condx unit, or conditionally on the ALU result. If we give up on conditional moves and similar instructions, then the `enable` signal need no longer be fed to a `regwrite` mux, but whether to write an ordinary register can be a control signal decoded from the instruction.

6.4 (a) We can get the maximum of `r0` and `r1` into `r0` by using the two instructions,

```
cmp r0, r1
```

```
        movlt r0, r1
```

This executes is 2 cycles, whereas the following code takes 3 cycles in the case where the mov happens, and 4 in the case where it does not, because of the 2-cycle penalty for a taken branch.

```
        cmp r0, r1
        bgt skip
        mov r0, r1
    skip:
```

(On the single-cycle datapath, there is no branch penalty, so the advantage of the conditional move is lessened. Conversely, on processors with a deeper pipeline, conditional moves will offer more of an advantage.)

(b) It's convenient to embed the condition in the same place it appears in conditional branch instructions, because the datapath already has wiring that will then feed it to the condition unit. The first four bits of the instruction can then contain the opcode – 12, if we follow the suggestion in the instruction; then the first five bits of the instruction will be either 24 or 25, and like conditional branch instructions, conditional moves will occupy two of the 32 slots in the main decoding table. We have 8 bits left to encode the source and destination registers. At no extra cost, we can allow high registers here, following the same encoding used in the high-register add/cmp/mov instructions. The instruction format is therefore as follows.

mov⟨c⟩ ⟨Rxx⟩,⟨Ryy⟩

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	cond				X	Ryy				Rx		

where cond is the condition, Ryy is the source register and Rxx = X:Rx is the destination register. This format allows high registers to be used for both Rxx and Ryy. The entry in the decoding table can agree precisely with that for the high-register mov instruction, except that write-back is conditional.

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cWFlags*	*cWReg*	*cWLink*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*			
movc	–	*Ryy*	*Rxx*	*RegB*	*Lsl*	*S0*	*Mov*	*F*	*F*	*F*	*C*	*N*
mov hi	–	*Ryy*	*Rxx*	*RegB*	*Lsl*	*S0*	*Mov*	*F*	*F*	*F*	*Y*	*N*

For comparison, the entry for the move instruction is also given. This conditional move instruction can also be used as a conditional branch with the target taken from a register, simply by specifying pc as the destination of the move.

(c) Since the write-back part of an instruction can be made conditional in the datapath, any instruction whose only effect is to set a register can be executed conditionally if an encoding is provided. This would include conditional arithmetic operations of all kinds, with a few caveats:

- If made conditional, operations that set the condition codes would set them whether the operation was executed or not. Note that the conditional move described above does not set the condition codes. Likewise, conditional branch-and-link instructions would trash the `lr` whether executed or not.

- Conditional stores are not supported, since the cMemWr signal is not dependent on the condition.

- Arguably, conditional loads will not work either. Certainly, the write-back of the result can be made conditional, but the memory cycle will happen anyway, leading potentially to a needless cache miss or addressing fault.

6.5 (a) In the example instruction given, `r2` might contain the base address of an array of (say) 4-byte integers, and `r3` an index into the array. It's necessary to multiply the index by 4 in order to obtain the offset of the desired element from the base of the array, and common to perform this multiplication with a left-shift instruction immediately before a load or store. The new addressing mode could save up to one instruction per array access.

(b) In accordance with the established pattern, we'll use opcode 14 for `str` and 15 for `ldr`. The format agrees with the existing variants `ldr` and `str` with a reg+reg addressing mode. Here is the format for `ldr`:

ldr ⟨Rx⟩,[⟨Ry⟩,⟨Rz⟩,LSL #2]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	0	0	Rx			Ry			Rz		

The format for `str` is similar, but with `01110` in place of `01111`. It's important in implementing these instructions, unlike the existing `ldr` and `str`, to get the two input registers the right way round. Given the inconsistency in the use of Rn and Rm in ARM documentation, we are probably in a state where this can be achieved only by trial and error.

Here are decoding rules for the two instructions, with the existing rules for `ldr` and `str` alongside for comparison.

Instruction	*cRegSel*			*cRand2*	*cShift*		*cAluSel*	*cMem*		*cWFlags*	*cWReg*	*cWLink*
	A	*B*	*C*		*Op*	*Amt*		*Rd*	*Wr*			
str rx	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*S2*	*Add*	*F*	*T*	*F*	*N*	*N*
ldr rx	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*S2*	*Add*	*T*	*F*	*F*	*Y*	*N*
str r	*Ry*	*Rz*	*Rx*	*RegB*	*Lsl*	*S0*	*Add*	*F*	*T*	*F*	*N*	*N*
ldr r	*Ry*	*Rx*	*Rx*	*RegB*	*Lsl*	*S0*	*Add*	*T*	*F*	*F*	*Y*	*N*

The only difference is the use of the constant 2 (S2) as the shift amount in place of the constant 0 (S0).