Exercise 1 – Expanding Opcodes

Instruction 1: C – 2 operands

This instruction does not have the first 12 bits as all "1"s, so it isn't a no-operand instruction.
It doesn't have the first 7 bits as all "1"s, so it isn't a 1-operand instruction
But it does have the first 4 bits as all "1"s, therefore it must be a 2-operand instruction.

Instruction 2: B – 1 operand

This instruction does not have the first 12 bits as all "1"s, so it isn't a no-operand instruction.
But it does have the first 7 bits as all "1"s, therefore it must be a 1-operand instruction

Instruction 3: D – 3 operands

This instruction does not have the first 12 bits as all "1"s, so it isn't a no-operand instruction.
It doesn't have the first 7 bits as all "1"s, so it isn't a 1-operand instruction
And it doesn't have the first 4 bits as all "1"s, so it's not a 2-operand instruction either.
Therefore it must be a 3-operand instruction.

Exercise 2 – Addressing Modes

1. A – 16

This is Register Indirect addressing mode – the value pushed onto the stack is found in memory at the address contained in register R0. R0 contains 155, so the value is stored in the word at memory address 155 – this word's value is 16.

2. C – 155

The "#" shows this to be an Immediate-Mode instruction, so the value used is the one that's part of the instruction itself: 155.

3. D – 159

This is Register addressing mode, so the value used is the contents of the R2 register: 159

4. F – 2478

This is Direct addressing mode – the instruction contains the memory address of the value to be used. The address is 158, and the value of the word at that address is 2478.

5. B – 42

This is Indexed addressing mode – the value is found at the address contained in R0 plus 2. R0 contains 155, so the value is read from address 157 (155+2), which is 42.

6. E – 1053

This is Based Indexed addressing mode – the value is found at an address computed by adding the contents of R1, R3 and "58". The address is -2 (R1) plus 100 (R3) plus 58, which gives 156. The value of the memory word at address 156 is 1053.

Exercise 3 – Relative Addressing

F – B and C above

Answer A is false because relative addresses often contain too few bits to refer to any memory location. For example a relative jump with an 8-bit offset can only jump 128 bytes back or 127 bytes forward in the program.

Exercise 4 – Monadic Instructions

1. A – TRUE

As an example, it takes fewer bytes to put a zero into register R0 with a monadic instruction with an opcode and one operand such as:

```
CLEAR  R0
```

…than using a MOVE instruction which requires an opcode and two operands, such as:

```
MOVE  #0,R0
```

Since part of the execution time include fetching the instructions themselves, a shorter instruction will take slightly less time to fetch than a longer instruction will, therefore it's fair to say that the instruction will execute slightly faster.

2. A – TRUE

For example, if you have an instruction set that uses 8-bit opcodes, that gives you 256 different instructions. Having both a `CLEAR` and a `MOVE` instruction uses up two of those opcodes, leaving 254 available for other instructions – whereas making do with just a `MOVE` instruction uses up only one and leaves 255 for other instructions.

Exercise 5 – Conditional Branch

1. A – TRUE

As an example, a combined instruction to branch if R0 is zero might look like this:

```
BZ  R0,A10
```

…which takes up one opcode and two operands. The alternative:

```
CMP  R0,0
JZ   A10
```

…uses two opcodes and three operands. As before, fewer bytes in the instructions means faster fetching from memory and therefore faster execution.

2. B – FALSE

The target of the jump is given by the operand of the instruction. The opcodes only specifies when (i.e., under what condition) to jump, not where.

Exercise 6 – Signed Comparisons

1. **B** – **Branch**

   When the "-4" value is loaded into the register, it will be "`1111 1100`" binary (using 8-bit registers for this example), while the "5" value will be "`0000 0101`". When these are compared using an unsigned comparison, the "-4" value will be considered to be larger.

2. **B** – **Branch**

   As signed values, the "4" will be considered to be larger than "-5".

Exercise 7 – Loop Control

1. **B** – **Test at the bottom**

   The control variable in this loop from 100 down until it reaches 1. When the compiler creates the code it knows that the loop will always execute 100 times, so it can use the more efficient test at the bottom of the loop.

2. **A** – **Test at the top**

   Although the end value of the control variable is known at compile time, the start value is not. If the start value is 100 or more, the statements in the loop should never be executed. Therefore the compiler must test the end condition at the top of the loop before the statements in it's body are executed.

Exercise 8 – I/O

1. **D** – **Asynchronous I/O**

   Asynchronous I/O means that the program instructions continue to execute while data is being read or written.

2. **A** – **Polling**

   Programs that poll keep the CPU busy while waiting for I/O, therefore they are only used in embedded or dedicated systems where the CPU would have no other work that needs to be done.

3. **B** – **Programmed I/O**

   Issuing program instructions for slow-speed devices like the mouse or keyboard is not a problem, but for high-speed devices like disk drives it can seriously slow down the transfer rate. For this reason most high speed devices use DMA (Direct Memory Access). These are sometimes known as "Bus Mastering" devices.

Exercise 9 – Recursive Procedure

A – **There's no way for the procedure to stop calling itself**

The procedure calls itself whether n is greater than one or not, so it will rapidly run out of stack space to hold copies of parameters and local variables for all of it's incarnations.

Exercise 10 – Traps and Interrupts

C – **They both cause a hardware procedure call to a service routine**

Traps are caused synchronously by the action of the program itself, while interrupts occur as a result of events that are asynchronous to the program. Note that traps often, but not always, are an indication of an error condition. Interrupts are usually not caused by an error.