## Exercise 1 – Pointers

1. **G – 63**

   We find constant number 9 by adding “9” to the value in the CPP register: 3370 + 9 = 3379. This tells us that the value for constant number 9 can be found at address 3379. This address contains the value “63”.

2. **G – 63**

   We find local variable number 5 by adding “5” to the value in the LV register: 3381 + 5 = 3386. This tells us that the value for local variable number 5 can be found at address 3386. This address contains the value “63”.

## Exercise 2 – The Stack

1. **A – 142**

   Each “push” increases the stack pointer by one, and each “pop” decreases it by 1. So the original value (143) plus two pushes (+2) less three pops (-3) = 142.

2. **F –1**

   After the pushes and pops the SP register contains 142 – this is the address of the new word at the top of the stack. The value at this address is “0001”, or “1”.

## Exercise 3 – Execution

**G – C and D above are both true**

The four instructions push two words onto the stack (two ILOADs), replace the two words with their sum (IADD), then remo ve a word from the stack (ISTORE). The SP register will change during the instructions, but once they’re completed it will have the same value it started with. Therefore answer “A” is not correct.

None of the instructions change the LV register (although the ILOAD and ISTORE instructions use it to find the local variable), so answer “B” is not correct.

The PC register is updated after each instruction to point to the next instruction to be executed, so after executing these four instructions it will contain “0159”, the address of the whatever instruction follows these. So this register DOES change.

The ISTORE instruction places a new value into local variable number 5 (it’s an “ISTORE 5” instruction). Local variable number 5 is at memory address 780 (LV register) + 5 = 0785. So the word in memory will change (it receives the sum of local variables 4 and 8, which is 129). So this word DOES change.

Therefore, the correct answer is G – the PC register and the memory word will be different.

## Exercise 4 – Conditional Branch

**E – 2507**

The instruction will branch to target address if the top two words on the stack are true. The target address is found by adding the address of the instruction (2105) to the operand (0402). Note that operands in the Java Virtual Machine are stored in Big Endian format, so "0402" is the correct way to write the number (it would be "0204 if it were in Little Endian format). 2105 + 0402 = 2507, which is the address the next instruction would be fetched from.

## Exercise 5 – BIPUSH

**A – 8**

The operand of a BIPUSH instruction is the value that is used – it's not a local variable or constant number. So a "BIPUSH 8" pushes the value "8" onto the stack.

## Exercise 6 – The Wide Prefix

**B – The WIDE prefix allows a Java program to have more than 256 local variables**

The value of a variable is stored in a memory word which is 32 bits long – this means that variables can have values as large as about 2 billion ($2^{31}$), since all variables are signed and therefore the sign bit is not part of the value. This is true for all instructions whether there's a wide prefix or not, therefore answer "A" is incorrect.

The WIDE prefix also has nothing to do with the stack or with the number of instructions in the program, so answers "C" and "D" are incorrect.

The WIDE prefix expands the operand of ISTORE and ILOAD from 8 bits to 16 bits. This operand holds the local variable number – so without WIDE it means local variable numbers can range from 0 to only 255. So using the WIDE prefix allows more than 256 local variables.

## Exercise 7 – Compiling Java Source Code

**A** – The variables are loaded onto the stack using ILOAD, and the value "6" is loaded using BIPUSH.
Answer "B" tries to load the "j" value using a BIPUSH, which is incorrect.
Answer "C" tries to load the "6" value using an ILOAD, which is incorrect
Answer "D" makes both these errors.

You may wonder why there's a Constant Pool if you can use BIPUSH to load constant values like the "6" used here. The answer is that BIPUSH only has a signed 8-bit operand so it can only handle constants from -128 to 127. Larger constants are put into the constant pool and loaded using the LDC or LDC_W (Load Constant or Load Constant Wide) instructions.

## Exercise 8 – Data Path Control Signals

**B – Add the values in the "TOS" and "H" registers together and store the result in the MDR register**

The ALU control signals tell it to add it's A and B inputs together. The A input comes from the H register, since that's how the input is wired. The B input comes from the TOS register, because the control signal connects TOS's output to the B bus. The ALU result ("H + TOS") travels up the C bus, where control signal causes it to be loaded into the MDR register.

## Exercise 9 – Memory Access

**1. A – Read the value at the top of the stack from memory**

The data path sends the value in the SP register (which is the address of the top of the stack) down the B bus into the ALU's B input. The ALU function code tells it to pass that value through to the C bus without changing it. And the MAR control signal tells it to load the result. So the result of the data path operation is to put the address of the top of the stack into the MAR register.

The memory control unit's "RD" signal is activated, so the unit will read from the memory address specified by the MAR register (the top of stack).

**2. B – The "MDR" register**

Values read from memory via the "RD" signal are placed into the MDR register. Note, however, that the value doesn't arrive right away – it can't be used until the 2nd clock cycle following the read.

The MBR register receives values read from the Method Area via the "Fetch" signal.

The SP register itself isn't used for memory reads or writes (although it's value is often copied to the MAR register so that the stack can be read from or written to).

## Exercise 10 – MBR

**1. A – FF**

"Load as Unsigned" puts the MBR register bits onto the low-order portion of the B bus and sets all of the high-order bits to zero. So in fact the value that's loaded is "00000000FF" – which is equivalent to "FF" (the same way that 00000123 = 123).

**2. D – FFFFFFFF**

"Load as signed" puts the MBR register bits onto the low-order portion of the B bus and sets all of the high-order bits to the same value as MBR's sign bit (bit 7, the high-order bit). The MBR register contains "FF" which is "1111 1111" in binary – so it's sign bit is "1". This means that all of the high-order bits are set to "1"s as well. Since there are 32 bits on the B bus, the value that goes into the ALU is "FFFFFFFF" (eight "F"s since each "F" is "1111", and eight of them give a total of 32 "1" bits).