Exercise 1 – Direct-Mapped Cache

1. **D – 1M bytes**

   The entire address is 20 bits long, therefore the system can access 2^{20} = 1M bytes

2. **E – 16 bytes**

   The "Byte" field of the address is 4 bits long, therefore there must be 2^4 = 16 bytes in each line.

3. **D – 256 lines**

   The "Line" field of the address is 8 bits long, therefore there must be 2^8 = 256 lines in the cache.

4. **C – 4K bytes**

   There are 256 lines with 16 bytes each so the total capacity is 256 x 16 = 4096, or 4K bytes.

   Another way to look at this is that there are a total of 12 bits in the "Line" and "Byte" fields of the address, so the capacity must be 2^{12} = 4K bytes.

Exercise 2 – Set-Associative Cache

1. **D – 1M bytes**

   The entire address is 20 bits long, therefore the system can access 2^{20} = 1M bytes

2. **E – 16 bytes**

   The "Byte" field of the address is 4 bits long, therefore there must be 2^4 = 16 bytes in each entry.

3. **D – 256 lines**

   The "Line" field of the address is 8 bits long, therefore there must be 2^8 = 256 lines in the cache.

4. **E – 16K bytes**

   There are 256 lines, and since this is a 4-way set associative cache this means there are four 16-byte entries for each line. This means the total capacity is 256 x 4 x 16 = 16384, or 16K bytes.

   Answer "C" would be correct for a direct-mapped cache, but this is a 4-way set associative cache so each line has four entries and the total capacity is 4 times larger than a direct-maped cache would be.

Notice that the memory address is treated in exactly the same manner for either a direct-mapped or a set-associative cache. The only difference is that a set-associative cache has multiple entries for each cache line.

Exercise 3 – Cache

1. **C**

   The system can access 1M bytes – since 2^{20} = 1M this means the address must be 20 bits long.
   There are 256 cache lines and 2^8 = 256, so the "Line" field must be 8 bits long.
   There are 16 bytes per line and 2^4 = 16, so the "Byte" field must be 4 bits long.

2. **C – 4K Bytes**

   There are 256 lines with 16 bytes each so the total capacity is 256 x 16 = 4096, or 4K bytes.

   Another way to look at this is that there are a total of 12 bits in the "Line" and "Byte" fields of the address, so the capacity must be 2^{12} = 4K bytes.

3. **B – line 2**

   In the hex address "`0502F`", the "`05`" is the tag (because from question 1 the tag is 8 bits long and 2 hex digits = 8 bits), "`02`" is the line number (for the same reason) and "`F`" is the byte number (because the byte number is 4 bits and 1 hex digit = 4 bits)

Exercise 4 – Cache

B – The data being requested is not in one of the cache entries. The cache reads the data from memory and places it into entry 2.

In the hex address "`073FF402`", the "`073`" is the tag (because the tag is 12 bits long and 3 hex digits = 12 bits), "`FF40`" is the line number (line number is 16 bits long and 4 hex digits = 4 bits) and "`2`" is the byte number (because the byte number is 4 bits and 1 hex digit = 4 bits)

The cache looks to see if the given line has an entry with a tag of "`073`" – but there is no such tag. So the cache will request the information from memory.

Since all four entries already contain valid copies of data from memory, one of them will have to be replaced by the data being read for the current read request. The cache will use the Least Recently Used entry, which is number 2.

Exercise 5 – Branch Prediction

1. **B – it avoids having to flush and re-fill the pipeline**

   Since the system sends subsequent instructions into the pipeline before the conditional branch executes, the work done in those stages is wasted if the branch ends up jumping to a different set of instructions. In a pipeline with a lot of stages this can mean a lot of wasted clock cycles.

2. **F – all of the above**

   Static prediction = assuming (for example) that all "backward" jumps will jump and all "forward" jumps won't.

   Prediction hints = different opcodes for "branch (probably)" vs. "(unlikely) branch".

   Dynamic branch prediction = track actual branch results for each instruction as it executes and the next time the instruction is executed try to guess which way it will go based on previous activity.

Exercise 6 – Detecting Dependencies

1. **C** – **Write After Read (WAR) dependency**

   This instruction (`R0=R2+R6`) will write a new value into register R0, but the scoreboard shows that a prior instruction is still using (reading) the old R0 value.

2. **B** – **Write After Write (WAW) dependency**

   This instruction (`R3=R7-R0`) will write a new value into register R3, but the scoreboard shows that a prior instruction is still writing a different value into R3.

3. **D** – **No dependency**

   This instruction (`R6=R4*R2`) has no conflicts with prior instructions. R6 is not being read or written by any instructions, and R2/R4 are not being written by any instructions.

   Note that both R2 and R4 are being used (read) by previous instructions, but it's perfectly OK for multiple instructions to read values from the same registers since the values aren't affected by this.

4. **A** – **Read After Write (RAW) dependency**

   This instruction (`R6=R5/R6`) is using (reading) the value in R5, but a prior instruction is still writing a new value into R5.

   This instruction reads and writes register R6, but this is not a problem since dependencies can only exist between different instructions.

Exercise 7 – Register Renaming

E – **A and B above**

Register renaming is used to eliminate "write-after" dependencies. If an instruction needs to write into a register that is still in use by a prior instruction, register renaming allows the instruction to write into a different register and therefore avoid the dependency.

Exercise 8 – Registers

1. **C** – **Status or Flags Register**

   This register holds flags such as "C" (Carry), "N" (Negative), "V" (Overflow), "Z" (Zero), etc which record the status of the last arithmetic or logical operation.

2. **E** – **Program Counter or IP register**

   The program counter holds the address of the next instruction. This is called the "IP" (Instruction Pointer) register on Intel systems.

3. **B** – **Frame Pointer, BP or LV register**

   This register is called by various names of different systems, but has the same usage on all.

Note that most systems have a Microinstruction register, but it's part of the implementation of the processor and not part of it's Instruction Set Architecture.

Exercise 9 – Instruction Set Architecture

F – **A, B and C above**

The clock speed and the number of pipeline stages are part of the implementation of a processor, not part of it's Instruction Set Architecture. For example, an Intel Pentium and an AMD Athlon have different clock speeds and pipelines, yet they can execute the same IA-32 instructions.