

Diamonds of Computer Science (202500333)

Binary logic and computer architecture

Test of September 30, 2025

Answers

1. Binary numbers

6 pt (a) -22
One can calculate this using the weights of the positions, with the left-most bit having negative weight: $-1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^0 + 0 \cdot 2^0 = -22$.

5 pt (b) 101010111100
Conversion from hexadecimal to binary is easy as each hexadecimal digit corresponds to a group of four bits (because $16 = 2^4$). Convert each hexadecimal digit to binary separately: A(hex) = 10(dec) = 1010(bin); B(hex) = 11(dec) = 1011(bin); and C(hex) = 12(dec) = 1100(bin). Then concatenate them to get the result.

6 pt (c) 520
Although 208 may look decimal, it is given that it is hexadecimal. So the weights of the digits are 16^2 , 16^1 and 16^0 , giving the final result $2 \cdot 16^2 + 0 \cdot 16^1 + 8 \cdot 16^0 = 520$ decimal.

4 pt (d) • (C) When a number is written in binary according to this system and then interpreted as if it is a normal 2-complement number, only the plus/minus sign is flipped.
Note that the weights are the same as in the normal 2-complement system, except that all the signs are flipped. So also the result is the same except for the sign.

4 pt (e) • (B) No, the resulting number will be 1 too high.
The easiest way to figure this out is by trying a few numbers and see what happens. However, it can also be reasoned out formally, as follows.

First consider all bits except the left-most two bits: they all move to a position where their weight is doubled.

Next, consider the left-most bit. Since the original number is given to be negative, we know that this bit must be 1, and its contribution -2^7 . After shifting, this bit is moved in at the right, where its contribution is only 2^0 .

Thirdly, consider the second bit from the left. After shifting, it becomes the left-most bit, and thus determines the sign. It is given that the original number was such that after doubling, it can still be represented correctly in 8 bits 2-complement notation. That means that after doubling the number is ≥ -128 , so before doubling it must be ≥ -64 , implying that this second bit from the left was originally 1. Originally it contributed 2^6 , and after shifting it contributes -2^7 .

Taking the latter two together we see that their total contribution changes from $-2^7 + 2^6 = -64$ to $-2^7 + 2^0 = -127$; that's a doubling followed by an increase of 1.

Now combining everything, we see that the total number is indeed doubled and then increased by 1.

Note: option (F) was an editing mistake; the question clearly needed a yes/no answer, which this option did not provide.

Continued on next page...

2. Boolean logic

6 pt

(a)

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

8 pt

(b) From top to bottom:
 distributive (F)
 wrong (J)
 DeMorgan (I)
 idempotence (M)
 complement (G)

7 pt

(c) The most straightforward is $\overline{ABC \cdot \overline{B + C}}$, following straight from the diagram: a NAND taking as its inputs the results of an AND and a NOR.

However, it can be simplified further, by applying DeMorgan's law twice:

$$\overline{ABC \cdot \overline{B + C}} = \overline{ABC} + B + C = \overline{A} + \overline{B} + \overline{C} + B + C = \overline{A} + B + \overline{B} + C + \overline{C} = \overline{A} + 1 + 1 = 1$$

So, this entire arrangement of gates will always output 1, regardless of the inputs!

4 pt

(d) • (D) It's a flipflop that remembers which of the inputs was most recently '1', rather than '0' as in the original NAND flip-flop.

Consider what happens if one input of the circuit is 1 and the other is 0. The NOR with a 1 on its input is guaranteed to make its output a 0, regardless of what the gate's other input is. And this in turn forces the other NOR gate's output to be a 1 (as both its inputs are 0). If subsequently both circuit inputs become 0, this state is preserved: when one input of a NOR gate is 0, its output is the inverse of its other input.

Continued on next page...

12 pt

3. Problem 3

One example of a correct solution:

	read address 1 / write address	read address 2	instruction	explanation
Timeslot 0	0	0	1	calculate $x \cdot x$ and store this in R0
Timeslot 1	0	1	1	calculate R0·R1, which gives $x^2 \cdot y$, and store this in R0
Timeslot 2	1	1	1	calculate y^2 and store this in R1
Timeslot 3	1	2	1	calculate R1·R2, which is y^2z and store in R1
Timeslot 4	1	0	0	calculate R1+R0, which is the final result and store in R1

There are several other correct answers. The most frequently made error is prematurely overwriting a register whose (original) contents you still need further on. For example, if you would start by multiplying R1 by itself (to get y^2), you'd no longer have access to the original value of y for later multiplying it with x^2 .

Continued on next page...

4. Problem 4

18 pt

(a) In hexadecimal:

line	programcounter	R17	R18	R19	Z	branch/comment
1	0000	D8				
2	0001	D8	08			
3	0002	D8	08	10		
4	0003	E0	08	10	0	
5	0004	F0	08	10	0	
6	0005	F0	08	08	0	
7	0006	F0	08	08	0	branch to 0004 (ADD...)
8	0004	F8	08	08	0	
9	0005	F8	08	00	1	
10	0006	F8	08	00	1	branch not taken
11	0007	00	08	00	1	
12	0008	00	00	00	1	
13	0009	00	00	00	1	branch to 0006
14	0006	00	00	00	1	branch not taken
15	0007	00	00	00	1	
16	0008	00	F8	00	0	
17	0009	00	F8	00	0	branch not taken

Or in decimal:

line	programcounter	R17	R18	R19	Z	branch/comment
1	0000	216				
2	0001	216	8			
3	0002	216	8	16		
4	0003	224	8	16	0	
5	0004	240	8	16	0	
6	0005	240	8	8	0	
7	0006	240	8	8	0	branch to 0004 (ADD...)
8	0004	248	8	8	0	
9	0005	248	8	0	1	
10	0006	248	8	0	1	branch not taken
11	0007	0	8	0	1	
12	0008	0	0	0	1	
13	0009	0	0	0	1	branch to 0006
14	0006	0	0	0	1	branch not taken
15	0007	0	0	0	1	
16	0008	0	248 (or -8)	0	0	
17	0009	0	248 (or -8)	0	0	branch not taken

There are many points of attention:

- The program counter (called PC in the instruction set document) is the register containing the address of the instruction currently being executed. This was discussed in the lecture, but indeed never came back in the practicals, and it was clear from the results that many/most students didn't know this, so we did not take it into account for grading. (Also, strictly speaking we asked for the value *after* the instruction had been performed, which would be the address of the *next* instruction; but for clarity in the tables above we wrote the address of the *current*

instruction.)

Note that writing down the program counter is useful to keep track of where you are in the program, so you don't accidentally skip an instruction.

- The Z flag is set by arithmetic instructions (set to 1 if the arithmetic result is 0, and to 0 otherwise). Other instructions (in particular, the branch instructions) leave it unchanged. Like the PC, for many students this column was apparently unclear/unexpected, and we didn't take it into account for grading (this time).
- It's enough to only write down the registers that are changed by an instruction (black in the above tables), as the others remain unchanged (gray in the tables).
- Be consistent and careful about using hexadecimal or decimal notation. This holds particularly with numbers like the 10 that is loaded into R19 initially, which is hexadecimal, but looks like a valid decimal number too.
- Be careful about where the branches jump to. It's easiest to remember that RJMP -1 gives an infinite loop (as we saw during the practicals), so -1 results in a jump to itself, -2 to the instruction just above, -3 to the one before that, etc.
- Be careful about the meaning of BRNE and BREQ and their difference: BRNE jumps if the Z flag is *not* set (i.e., is 0), while BREQ jumps if the Z flag *is* set (i.e., is 1). These instructions do *not* do a comparison *themselves*, despite the name 'branch if (not) equal': they rely on a previous instruction having (un)set the flag.
- Although we didn't subtract points for it (this time), the exam said you should use one line per instruction; so also use a line for each branch instruction, whether they result in a jump or not. This is also very helpful in the next question, where you're asked to count the number of clock cycles.
- In line 12, we add 8 to R17 which is already at F8 (hex.) or 248 (dec.), resulting in 100 (hex.) or 256 (dec.). However, this doesn't fit in 8 bits (the size of the Arduino registers), so the overflow is lost, resulting in 0.
- Similarly, in line 16, we subtract 8 from R17 which is already at 00, resulting in F8 (hex.) or -8 (dec.) or 248 (dec.). The reason both -8 and 248 are allowed in decimal is because they are two different interpretations of the *same* bits in the register, namely as a 2-complement signed number, or as an unsigned number, respectively. The processor doesn't know or care about this difference.
- In line 13, we jump (BREQ) to address 0006, which contains another conditional jump, namely BRNE. The flag registers do not change here, so the Z flag at this moment is still set (otherwise the BREQ would not have jumped), so the BRNE will *not* jump at this point.

5 pt

(b) 19 cycles.

Explanation: 17 instructions are executed, as can be seen in the table. 2 of these are BRNEs and BREQs that do jump and thus each take 2 cycles rather than 1, bringing the total to 19 cycles.

For grading this question, we checked whether your answer matched what you wrote in question (a).

Note that a BRNE or BREQ that does not result in a jump because its condition is not satisfied, still takes 1 cycle to process.

Continued on next page...

15 pt

5. Problem 5

This calculates $\lfloor \log_2(X) \rfloor$, i.e., the base-2 logarithm of X , rounded down to an integer.

Looking at the code, we see a loop (between the `repeat` label and the `RJMP` at the end). The only way to get out of this loop is via the `BRCS`, which jumps if the carry flag was set by the `SUB`traction, i.e., if at this point `R17` (which gets initialized as X and never changes) is less than `R18`.

If that condition is not satisfied, then the `SUB` is undone by the next `ADD` instruction. What remains in the loop is doubling `R18` (by adding `R18` to itself), and incrementing `R19`. So basically, we keep doubling `R18` until it is bigger than X , and in `R19` we count how many times `R18` was doubled.

In other words, we're calculating powers of 2 in `R18`, and stop when this power of 2 is bigger than X . And we count how many times we can double `R18`. Thus, we find to what exponent 2 needs to be raised to become bigger than X . That sounds like a logarithm!

Now, to get to a precise conclusion (about rounding and a possible offset), we have to carefully check the numbers. At the start of the loop, `R18=1=20` and `R19=-1` (FF in hexadecimal, using 2-complement arithmetic). So we have `R18 = 2R19+1` there. And as we saw before, `R18` is doubled and `R19` is incremented every iteration, from which it follows that everytime we are at the `repeat` label we have `R18 = 2R19+1`. (B.t.w., such a thing that is true everytime you go through the loop, is called a *loop invariant*. Finding them can be very useful for reasoning about what a program does.)

Thus, when we leave the loop, we have that $2^{R19+1} > X$. Besides that, we also know that $2^{R19} \leq X$ as otherwise we'd have left the loop the previous time. Thus, we see that indeed `R19` is the 2-logarithm of X , rounded down.