1. In binary, \( i = 10100 \), \( j = 01100 \) (I’m omitting the 27 leading zeros). Then:

\[
\begin{align*}
i \mid j &= 10100 \mid 01100 = 11100 = 28 \text{ (base 10)} \\
i \& j &= 10100 \& 01100 = 00100 = 4 \text{ (base 10)}
\end{align*}
\]

2. We do the whole number part separately—6\(icom\) = 110\(itwo\). For the rest, repeatedly multiply the fractional part by 2, using the whole number part of the product as the next bit in the fraction:

\[
\begin{align*}
.3 \times 2 &= 0.6 \\
.6 \times 2 &= 1.2 \\
.2 \times 2 &= 0.4 \\
.4 \times 2 &= 0.8 \\
.8 \times 2 &= 1.6 \\
.6 \times 2 &= 1.2 \\
\ldots &= \ldots \text{(repeats)}
\end{align*}
\]

The final answer is therefore 110.010011001\ldots

3. Initially, \( a_0 = 10 \), \( a_1 = 20 \). Executing \texttt{jal f} takes us to the function \( f \), where the instructions do the following:

\[
\begin{align*}
\text{add} & \quad a_0, a_0, a_1 \quad \# a_0 = a_0 + a_1 = 10 + 20 = 30 \\
\text{sub} & \quad a_1, a_0, a_1 \quad \# a_1 = a_0 - a_1 = 30 - 20 = 10 \\
\text{sub} & \quad a_0, a_0, a_1 \quad \# a_0 = a_0 - a_1 = 30 - 10 = 20
\end{align*}
\]

Then the function returns to the statement just after the \texttt{jal}, at which point the program terminates with a \texttt{syscall} of 10. Thus, the final values are \( a_0 = 20 \), \( a_1 = 10 \).

4. Here’s the entire program (with function \( g \) highlighted):

```c
#include <stdio.h>

/* Function prototype for \( g \): */
double g(double a[], int n);

double main() {
    double x[4] = {1.2, 2.2, 3.4, 2.1};
    double y[8] = {3, 4, 5, 6, 7, 8, 9, 10};
```
printf("g(x,4)=%f
",g(x,4));
printf("g(y,8)=%f
",g(y,8));
}

double g(double a[], int n) {
    return(a[0]+a[n-1])/2.0;
}

5. 1 01111111 011000000000000000000000
The sign bit is "1", so we know the number is negative. The next 8 bits hold the power of 2 plus 127; subtracting 127 we get 0, so now we know that the number is -1...something...x2^0. (The leading "1" is assumed to be there always—this is the "hidden bit.") The "...something..." is the fraction .0110..., and from the table of simple fractions we see that corresponds to 3/8, or .375 in decimal. Therefore, the answer is -1.375.

6. addi $sp,$sp,-12 # make room for three words (also subi $sp,$sp,12)
sw $s0,0($sp) # store register $s0 on the stack
sw $s1,4($sp) # store register $s1 on the stack
sw $ra,8($sp) # store register $ra on the stack

7. lw $ra,8($sp) # load register $ra from the stack
lw $s1,4($sp) # load register $s1 from the stack
lw $s0,0($sp) # load register $s0 from the stack
addi $sp,$sp,12 # remove three words (also addi $sp,$sp,-12)
jr $ra # return

8. Since a is a pointer to the first element in the array, "*a" is the same thing as "a[0]". The pointers "a+1", "a+2", "a+3", etc., point to the second, third, fourth, ... elements, so "*(a+1)", "*(a+2)", "*(a+3)" mean the same as "a[1]", "a[2]", "a[3]", etc. Thus, the output is:

   10 50 30

9. ... assume $t0 contains n ...
loop: beq $zero,$t0,done # check for loop termination
move $a0,$t0 # getting ready to print
li $v0,1
syscall
addi $t0,$t0,-1 # update loop counter
loop # do it again

There is another way to test for the end of the loop:

loop: slt $t1,$zero,$t0 # if 0 < loop counter, set t1 = 1
beq $t1,$zero,done # if t1 ==0, loop counter must be <= 0
10.  

```
prod = 0;
for (i = 0; i < 32; i++) {
    if (a & 1) { /* see if we should add multiplicand */
        prod = prod + b;
    }
    a = a >> 1; /* shift multiplier one place to the right */
    b = b << 1; /* shift multiplicand one place to the left */
}
```

Note that “a & 1” is equal to 1 if and only if the rightmost bit of a is 1; otherwise it is 0. Since 0 means false, the addition is not performed when the rightmost bit of a is zero.

11.  

```
... assume $s1 contains multiplier, $s2 contains multiplicand ...

li $t0,32 # initialize counter ($t0 counts down to zero)
li $s0,0 # s0 = product
loop: beq $t0,$zero,done # all finished?
    andi $t1,$s1,1 # check multiplier bit to see if it’s 1
    beq $zero,$t1,skip # if not, don’t add
    add $s0,$s0,$s2 # add multiplicand to product
skip: sll $s2,$s2,1 # shift multiplicand left 1 bit
    srl $s1,$s1,1 # shift multiplier right 1 bit
    addi $t0,$t0,-1 # subtract 1 from loop counter
    j loop
done: .... not shown ...
```

This was perhaps a bit hard—you had to figure out (from the “beq” statement) that register $t1 contained the result of the check. Also, I used an “andi” rather than an ordinary “and” so that I wouldn’t need an extra register to hold the constant “1”.

The above is not a full review!