The following is a slightly modified solution set from a student in the class. My comments are in bold. I also created a table in problem 1.3(a) , so that students can better understand how an array is laid out in memory. –NR

1.1

a)

	Char	Short	Int	Long long	Float	Double
Add (+)	5 (1)	5 (1)	5 (1)	12 (2.4)	68 (13.6)	102 (20.4)
Sub (-)	5 (1)	5 (1)	5 (1)	10 (2)	79 (15.8)	123 (24.6)
Mul (*)	7 (1.4)	7 (1.4)	7 (1.4)	23 (4.6)	56 (11.2)	106 (21.2)
Div (/)	16 (3.2)	16 (3.2)	16 (3.2)	119 (23.8)	149 (29.8)	314 (62.8)

b) Add two ints: 4
Sub two long longs: 10

c)

sinf()	263 (52.6)
sin()	647 (129.4)
sqrtf()	293 (58.6)
sqrt()	603 (120.6)

d)

	Int	Long long
1 bit	5 (1)	17 (3.4)
10 bit	5 (1)	17 (3.4)
31 bit	5 (1)	17 (3.4)

Bit shifting by any number takes the same number of operations (confirmed by looking at the disassembly). If an int is bit shifted by 32, the result will be zero, as zeroes are shifted in from the left.

1.2

a) It takes 0 time.
There is no assembly code for the arithmetical operations in the disassembly.

b) I get a value of 29. The value is what I expect.

1.3

a)
Refer to Table 1 for a mental picture of how an array is setup in RAM. The formula for

calculating the address of the last element is:

last element address = start address + size of the array in bytes – size of data type in bytes.

Memory Address	RAM	Element in Array
0xAA0000238	*Byte 0*	Element 1
. . .	*Byte 1*	
	Byte 2	
	Byte 3	
0xAA000023C	*Byte 4*	
0xAA000023D	*Byte 5*	
. . .	*Byte 6*	
	Byte 7	
0xAA0000240	*Byte 0*	Element 2
. . .	*Byte 1*	
	Byte 2	
	Byte 3	
	Byte 4	
	Byte 5	
	Byte 6	
0xAA0000247	*Byte 7*	

Table 1: Memory layout of first two elements of an array of any 8-byte data type (i.e., double or long long).

A valid answer to 1.3(a):

	Bytes	**First element**	**Last element**
char	**100 (0x64)**	**0xAA00001D0**	**0xAA0000233**
short	**200 (0xC8)**	**0xAA0000B98**	**0xAA0000C5E**
int	**400 (0x190)**	**0xAA0000558**	**0xAA00006E4**
long long	**800 (0x320)**	**0xAA0000878**	**0xAA0000B90**
float	**400 (0x190)**	**0xAA00006E8**	**0xAA0000874**
double	**800 (0x320)**	**0xAA0000238**	**0xAA0000550**

b) double pointer array: 0x190 (400 bytes)
int pointer array: 0x190 (400 bytes)

The double pointer array is smaller than the double array because each entry in the double pointer array is an address, not an actual double precision variable. Therefore, all that needs to be allocated is enough space for the 32 bit address that indicates where a double variable will be.

c) **A short data type was used.**
memory: 0x3118 **(= sizeof(short)*6284 elements = 12,568 bytes)**

Init code:

```
short LookUpTable[6284];
int ii;
for (ii=0;ii<6284;ii++) {
        LookUpTable[ii] = (short)(10000*sin(ii/1000.0));
}
```

resolution: 0.001 radians

(The cast to a short inside the for-loop is one of the key parts to this problem!)

d) 647/9