Anthony Debruyn
Quentin Delhaye
Alexis Lefebvre
Aurélien Plisnier

Labo 6
dsPIC33

Dragomir Milojevic & Yannick Allard ELEC-H-473

1. Sequentially.

2.

Operation	Correct	Error	Result
`c=a+b`	yes		
`g=a+b`	yes		
`g=e+f` (l.21)	yes		
`g=e-f`	no	Trying to put a negative number in an unsigned variable	65280
`g=e+f` (l.26)	no	Overflow. The bit `SRBits.C` is then set to 1, the test passes and sets `g` to `0xFFFF` (the maximum allowed by its type).	3399
`s3=s1+s2` (l.32)	no	overflow	-120
`s3=s1+s2` (l.39)	no	overflow	120
`c=a*b`	no	overflow	0
`g=a*b`	yes		
`g=e*f`	no	overflow	42496
`h=e*f`	no	overflow due to the operation made on 16 unsigned bits.	42496
`h=(INT32U)e*(INT32U)f`	yes		
`j=h*i`	no	overflow due to the operation made on 32 unsigned bits.	1510719488
`j=(INT64U)h*(INT64U)i`	yes		

3.

Multiplication	Instruction cycles	Correct
`c=a*b`	6	5
`g=a*b`	6	6
`g=e*f`	4	4
`h=e*f`	6	6
`h=(INT32U)e*(INT32U)f`	15	15
`j=h*i`	22	19
`j=(INT64U)h*(INT64U)i`	110	15

4. We see that the various manner of computing $a * b/c$ corresponds to the different associations possible among the operands.

Different ways	Precision
d=(a*b)/c;	Since the result is 442.2 (floored to 442), we see that the amount of bits needed to represent the number in the memory is not sufficient. The value written in memory is $0xBA$ (186).
d=(a/c)*b;	Idem. The value written in the memory is even further from the correct value: $0xB8$ (184).
f=a*b/c;	The result written in the memory is correct ($0x1BA$, 442), since the number of bits given to the variable to be represented in memory is sufficient (16 bits).
f=(a*b)/c;	Idem.
f=(a/c)*b;	There is a problem here. The result written in memory is 440 ($\neq$ 442). This is due to the floor operation after each mathematical operation on the operands. Since the results are stored as integers, the decimal part is always erased. We need to minimize the error caused by this, this is why the multiplication should always be first.
f=a*(b/c);	The outcome is even worse for this association. This is again due to the floor operation on the result after each mathematical operation. The value written is 402 ($\neq$ 442).
f=(a*b)/c;	The result written is correct (80).
d=(a/c)*b;	Since the intermediate error is too small, the final result is not altered this time, even with the division first. We get 80 in the memory.

5. The reasons for these differences come from the fact that either the result variable is too small in bits to store the value, or the intermediate results are floored to the nearest $\leq$ integer.

   For the first reason, as the result is finally stored in a 8bits variable, the 8 MSB are not taken into account. The result which was $0b1.1011.1010$ in binary was written $0b1011.1010$.

   For the second reason, we must remember that the results are stored as integers in the memory/registers. Even the intermediate ones. So, if all the intermediate results are floored, we add an error to the value at each mathematical operation. As an example, let's consider the association (f=(a/c)*b;):
   At first, a is divided by c: temp = a/c = 201/5 = 40.2 $\rightarrow$ 40.
   Then: f = $temp * b$ = 440 $\neq$ 442.

   The computation without any parenthesis is calculated in the right order. One conclusion could be that we don't need to deal with the order ourselves, the machine decides for us what is the best one.

6. If we look in the documentation "Programmer's Reference Manual", we can see that multiplication takes only 1 cycle, while division takes 18 cycles. So division is the longest operation.

Operation	Cycles	μs
d=(a*b)/c;	28	0.466667
d=(a/c)*b;	29	0.483333
f=a*b/c;	28	0.466667
f=(a*b)/c;	28	0.466667
f=(a/c)*b;	30	0.5
f=a*(b/c);	30	0.5
f=(a*b)/c;	28	0.466667
d=(a/c)*b;	29	0.483333

   We can tell by looking at the table that the longest operations are the one where the result is put in the 16bits variable, and the division is made first.

7.

Method	Cycles	μs	Result
1st method (classical)	253	4.216667	0x39
2nd method	13	0.216667	0x38
3rd method	14	0.233333	0x39

8. In the 3rd method, the "d + 128" is used to correct the rounding. For example, 2.999 + 0.5 (the "+ 0.5" coming from the addition of 128) gives 3.499, which allows a better rounding. Indeed, with the truncation we have 3, not 2.

9. By multiplying every operand by 16, being on 8 bits encoding, the risk of overflow is really high.

10. We have b = 432, which cannot be stored anymore on 8 bits. Therefore we need 16 bits to code b.

11. In the stack frame created at the begining of the `main` function.

12. In the first working registers (`w0`, `w1` and `w2`).

13. At the begining of the stack frame created for the function `add3()`.

14. In the stack frame created at the begining of the function `add3()`.

15. They both use the working registers to compute the intermediate results of their computation and store the final result in `w0`.

16. This parameter is obviously passed as a reference (the name of the section was very helpful). Anyway, vectors/arrays are always passed as reference in C.

17. The "cells" of the array are set to given values, obtained from operations on a and b.

18. **&a:** The & operator returns the address in memory of the following variable.
    ***i:** The * operator is used to get the value at address i in memory. In French, we speak about "déréférencer".

19. "*The swapnum function is used to swap numbers.*" - Captain Obvious, 2014.
    The 2 variables are passed by reference (their addresses). We access to their values by the addresses (with the * operator). We swap the 2 values by using a temporary variable. We put the first value (*i) in it. Then we take the value from the second variable and put it in the memory at the address of the first variable (i). We eventually put the temporary value at the address of the second variable (j).

20. It is viewed as a normal area, only it's using the PSV address. It is so beceause that area of the program memory is mapped to the upper half of the data memory.

21. Using a 16 bits index requires less instructions since there no address extension needed as it's the case for a 8 bits index.