EE 675: Microprocessor Applications in Power Electronics - Assignment 5

Jayesh Choudhary 170070038

1 MAIN

We first setup the CLA registers by appropriate reset and enabling. Then we set the CLA VECT1 for Task 1 (which has highest priority) in LS5 mempry block. (Its program memory starts from 0xa800 and data memory from 0xa000). Now, we enable Task 1 by moving 1 in MIER. Now, we map the CLA data RAM to the CLA space. Also, we map CLA program RAM to the CLA space.
Now we put the input into the cla_in in CLA data memory. We set 0-th bit of c-status which is used to find if the CLA task is completed or not. We enable 0th bit of IACK to start Task 1 in CLA.
If c-status has its 0-th bit as zero, then the CLA task has completed and we take the output from CLA space in the variable named retval. We then transfer the value of retval to the CPU data memory. If not 0, then we keep checking this condition of c-status.
(*It is same for both the parts of the assignment*)

2 inv_170070038

$$\frac{1}{1-x} = \Sigma x^n \tag{1}$$

We see that the series can easily be implemented by using for loop. But in CLA, on checking the condition for flags and conditional branching, the next 3 instructions are always executed making it more time consuming or less efficient. Since we have to write the program only for the first 5 terms, unrolling the loop is much more easier owing to the fact that we do not need to change the counter value and also we do not need to do the comparisons. So here, repetitive addition and multiplication instructions are written using MADDF32 and MMPYF32 instructions. First, MR0 and MR1 are initialized by cla_in. Then, MR2 stores the addition of 1 and MR1. We then keep on multiplying MR1 with MR0 and adding the changed MR1 to MR2. Finally after 3 such repetitive instructions, we get the final result in MR2. We move it to retval. After completion of CLA code, we set the 0th bit of c-status as 0 to indicate that the CLA task has been completed and then further instruction in the MAIN can be executed. If we use the loop, there are total 11 instructions with 8 of them being in the for loop which are repeated 4 times due to the counter being decreased from 4 to 0. If we unroll the loop, then we can implement the same question using only 13 sequential instruction and so in the final submission I have not used the loops but instead the unrolling method.

3 exp_170070038

$$e^x = \Sigma \frac{x^n}{n!} \tag{2}$$

For exponential, apart from the iterative update of x, we also need to keep track of the factorial which is in the denominator. Also in CLA, we have only 4 MRx. Thus implementing the logic in loop using branching condition would not only require extra NOP instructions, but also the use of other variables which would also require some instructions if we are moving them to MRx and then updating them and sending them back to CLA data memory. So we directly use the unrolling method without even trying out the loop method. It would be inefficient and would require a lot of cycles to execute one complete exponential calculation. We take in input from cla_in to MR0 and MR1. MR2 stores the addition of MR0 and 1. This takes care of the first two terms. Now we multiply MR0 to MR1, store the product of MR1 and (1/factorial) in MR3 and add MR3 to MR2. We repeat these 3 instructions 3 times to get the final answer in MR2 (1st two terms were already added in MR2). After completion of CLA code, we set the 0th bit of c-status as 0 to indicate that the CLA task has been completed and then further instruction in the MAIN can be executed.

4 Results and Simulation

First five terms means n goes from 0 to 4
Values are being read from 8a02 using the instruction (rf 8a02) which reads 32 bits.

Input	Output	Sum of 5 terms of series
-0.2	0.83359987 (0x3f5566cd)	0.8336
0.2	1.2495997 (0x3f9ff2e2)	1.2496
0.5	1.9375000 (0x3ff80000)	1.9375
0.7	2.7730997 (0x40317a77)	2.7731
-0.8	0.73759997 (0x3f3cd35a)	0.7376
0	1 (0x3f800000)	1

Table 1: Inverse function (with value of x less than 1 and greater than -1)

Input	Output	Sum of 5 terms of series
6	114.64844 (0x42e54c00)	115
-5	13.687988 (0x415b0200)	13.70833
-0.4	0.67043740 (0x3f2ba1c9)	0.6704
0	1 (0x3f800000)	1
0.8	2.2219996 (0x400e353e)	2.2224
9	443.83252 (0x43ddea90)	445.375

Table 2: Exponential function

Here we observe that for inverse function there is not much deviation in the obtained result. But for exponential function, especially for large values, deviation is there due to precision error of multiplication and addition which adds up over each operation and propagated further.