EE2024 Assignment 1
Report

Name: Anniya Baskaran	**Matric No: A0141812R**
Name: Baghabrah Dana Mohammad A	**Matric No: A0144902L**

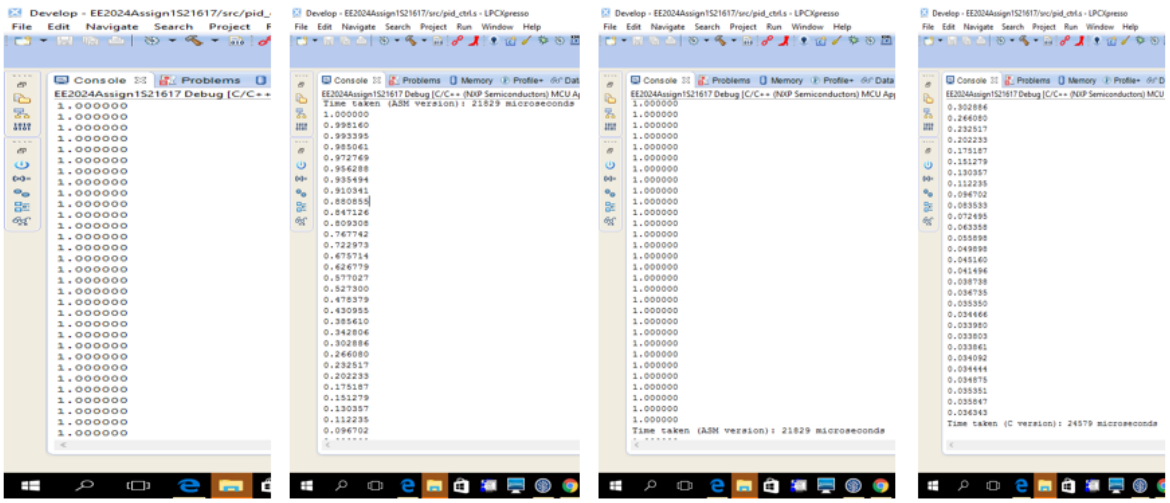
To resolve the issue of ASM function being only able to handle integer values but PID controller required to operate with floating point values, we multiplied the constants and limits of s(n) by a 100. Thus, the ASM program will return an integer value of e(n). Also, In the C program, we had a scale factor of 0.01 for e(n) and 0.0001 for u(n), so as to retain the float value.

```
u = pid_ctrl(e, st);      //Calling the assembly language function
u = u * 0.0001;
e = e * 0.01;
```

The reason for e(n) having a scale factor of 0.0001 is because it contained pairs of multiplications of two scaled values in the ASM as shown in the formula.

$u(n)=Kp\ e(n)+Ki\ s(n)+Kd\ [e(n)-e(n-1)]$

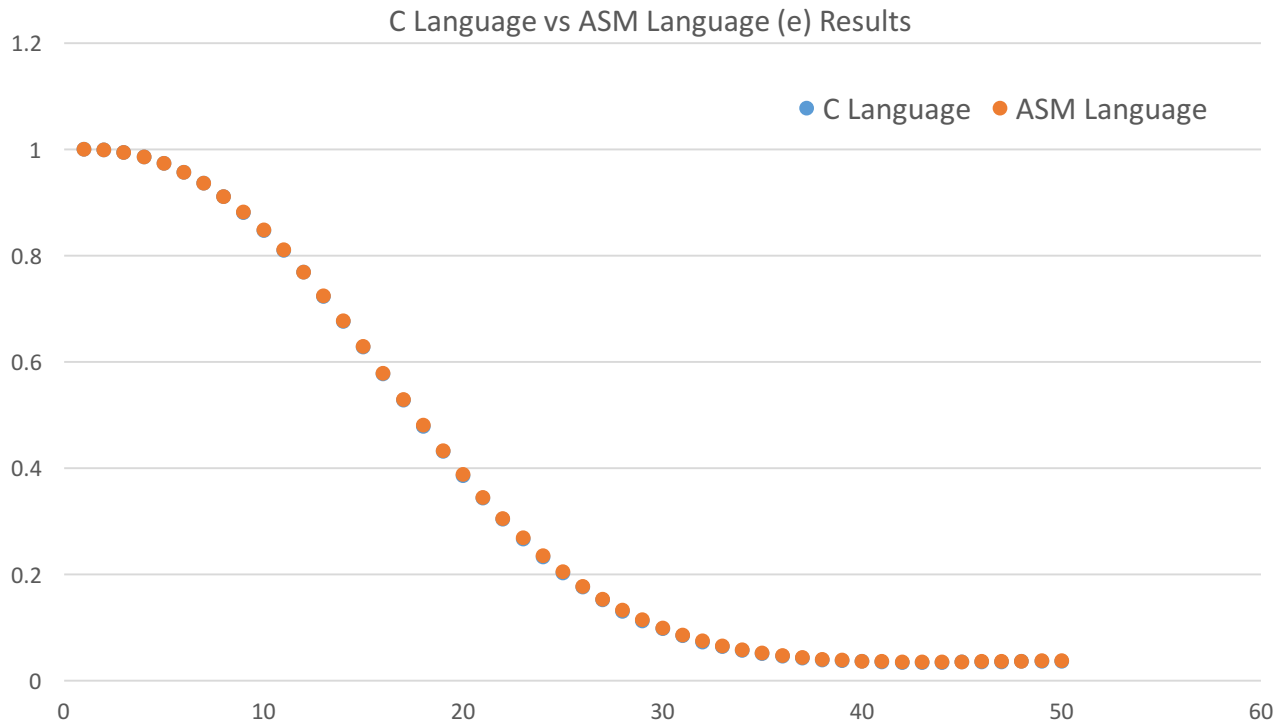
In addition, in the skeleton ASM program, it printed the value of e, which was 1.000000, fifty times. Also, it then stated the time executed by the ASM version, which was 21829 microseconds. Then for the C version, the value of e decreased from 1.000000 to 0.036343 and the time taken for that was 24579 microseconds.



The assembly language function pid_ctrl(int en, int st)

```
.syntax unified
  .cpu cortex-m3
  .thumb
  .align 2
  .global pid_ctrl
  .thumb_func
```

```
@  EE2024 Assignment 1: pid_ctrl(int en, int st) assembly language function
@  CK Tham, ECE, NUS, 2017
@ Dana Baghabrah A0144902L
@ Anniya Baskaran A0141812R
pid_ctrl:
@ PUSH the registers you modify, e.g. R2, R3, R4 and R5*, to the stack
PUSH {R2-R12}

@Loading static varibles
LDR R4, =sn
LDR R12, =enOld
LDR R2, [R4]    @Loads: R2 = sn
LDR R3, [R12]   @Loads: R3 = enOld

@Condition (start)
CMP R1, #1     @CMP compares R1 and 1 and updates flags
ITT EQ         @ If R1 == 1, it will go to the next two instructions below
MOVEQ R2, #0   @R2 = 0
MOVEQ R3, #0   @R3 = 0

@s(n) = s(n) + e(n)
ADD R2, R0     @R2 = R2 + R0
STR R2, [R4]   @Stores value into R2

@Checking limits
MOVW R6, #950 @MAX R6 = 950
MOV R7, #0
SUB R7, R6    @MIN R7 = -950

CMP R2, R6
IT GT         @if R2 > 950
MOVGT R2, R6  @R2 = 950

CMP R2, R7
IT LT         @if R2 < -950
MOVLT R2, R7  @R2 = -950

@un = Kp*en + Ki*sn + Kd*(en-enOld);
MOV R9, #25    @R9 = Kp
MOV R10, #10   @R10 = Ki
MOV R11, #80   @R11 = Kd

MUL R4, R0, R9       @R4 = Kp*en
MLA R5, R10, R2, R4  @R5 = Ki*sn + R4
SUB R4, R0, R3       @R4 = en - enOld
MLA R8, R11, R4, R5  @R8 = Kd*R4 + R5

MOV R3, R0    @enOld = en;
STR R3, [R12] @Stores value into R3

MOV R0, R8    @R0 will be returned to C program

@Store result
.lcomm sn 4
.lcomm enOld 4

POP {R2-R12}
BX LR
```



(i): `Time taken (ASM version): 2393 microseconds`

(ii): `Time taken (C version): 2809 microseconds`

The ASM language is typically faster than C language given that the time taken by ASM is less than that by C. This is because the ASM has a direct mapping between assembly language code statements and machine language instructions and allows access to machine instructions unlike C. Thus, the compiler, which is the translator for C, will involve extra code to translate to machine language to handle the generality. This will lower the speed of the C program.

Moreover, in order to increase the speed of the ASM, we can firstly use IT instead of branches. This is because for branching, after fetching a branch instruction, the processor needs to fetch the next instruction and it is possible that there are “next” instructions. It will not be certain which instruction is the next one until the branching instruction makes it to the end of the loop. Moreover, the processor will not be sure of which path will be taken beforehand. As a result, the processor has to stall until it knows the path to be taken. Not using the branches will thus optimize the performance.