Code Description

Introduction:

The code performs arithmetic on IEEE-754 Floating point numbers as per the requirement with the main subroutines named as FADD, FSUB, FMUL and FDIV.

Purpose of Registers:

R0 and R1 are used as input registers and after the code executes R0 is the output register. R4 and R6 are used to store the exponents of the two inputs while R5 and R7 store their respective mantissas. Since R0 contains the answer, all operations are moved into R4 and R5 at the end. R13 is used as the stack pointer for a downwards growing stack (ED). R3 is used as the pointer for the input memory block TEMP which also contains two additional values (required for division) and exceptional case outputs such as infinity and NaN. R8 is assigned with the sign of the answer and is used for putting back the answer together as an IEEE 754 number. For the multiply code, R8 contains the most significant 16 bits of the product and R6 contains the lower 32 bits. R9 is used to store the overflow from LSB end while normalising numbers of the form 10.frac and 11.frac and also while denormalising the answer if exponent is less than 0. R10 is used as the loop counter for the division loop. R11 is used as for handling special inputs infinity(+/-) and NaN.

Init:

Init is the first subroutine that the code will enter for all arithmetic. This is the sub-routine that separates the input numbers into exponent and mantissa and also calls the Normalise sub-routine that normalises denormalised inputs. Finally, it also sets R11 according to the condition of inputs. If R0 is infinite, we add 1 to R11 and if it is NaN we add 2 to R11. Similarly for R1 infinite, 4 is added and 8 for NaN. Then, in the code, individual bits of R11 are checked to see which condition is met for special inputs.

Chneg:

The Chneg sub-routine obtains the 2’s complement version of the mantissa if the sign is negative. This is done only for addition and subtraction. For multiplication, the sign is checked in the main sub-routine.

Normalise:

The Normalise sub-routine is used to normalise input and answers by default if of the form 0.frac so that we have nice normalised numbers to manipulate. The number of shifts to normalise is subtracted from the corresponding exponent. The mantissa is shifted by 8 by default as the most significant 8 bits will always be zero once exponent is removed.

Result1:

First, if the mantissa is completely zero then exponent is also set to 0 as that is the only case when mantissa will be zero. Then the sign of the answer is then taken into account by taking the 2's complement form. Then if the answer is less than 1.23 bits frac, it will go to Normalise else if greater it will go to Greater. The Greater branch does a similar function to Normalise without the initial left shift by 8. Moreover, it gets rid of the 1 before the decimal point while shifting. Then on return from Normalise (if sub-routine taken), R10 is assigned with the bits from the overflow R9 that need to be shifted into R5. R9 is also re-assigned with the remaining bits in the original R9. Then checks are performed to see if R9 is greater than 0x80000000. If greater, we add 1 and if equal we round to the nearest even number. However, if the answer is negative we branch to Negs which subtracts one for the greater than case to account for the sign. Then, we enter Result2.

Result2:

The main purpose of the result2 sub-routine is to put together R0 from R4, R5 and R8 (sign). Another function of Result2 is to denormalise R5 if R4 is less than or equal to 0. Moreover, the 1 before the decimal point removed earlier will be restored if the exponent is less than 0. Finally if there is any overflow while denormalising, it will store in R9 and consider that to round off the answer.

ADDH:

In ADDH initially we check for special inputs and handle them accordingly. For instance, if both inputs are infinity of opposite signs, we ORR either with 0xC00000 to make NaN. If either input is NaN then that is moved to the answer. If only one is infinite, then we make that the answer. If the first input is -0 then the second input is moved to R0 and we exit. This was made necessary because the code gives -0 for -0 + 0 when it should have given 0. Finally, another case was added due to necessity when tested. If we have both mantissas and exponents equal with different signs, the answer should be 0. However, one of the later sub-routines messed with the answer so answer is made 0 here itself.

Addition is implemented by finding the difference between exponents R4 and R6. This is stored in R8. If R8 is negative, we branch to Lesser. Otherwise we remain in ADDH. We shift R7 right (ASR) by the value of R8 and then add the two. While shifting R7, the overflow bits are stored as MSBs of R9. Lesser does the same function but shifts R5 instead of R7. Then as usual we branch to Result1 and Result2 and we get the final answer.

FADD:

ADDH was originally implemented in FADD. However to make it APCS compliant, a separate subroutine had to be implemented as Result2, which was the end of ADD was used by division. So, to restore registers, ADDH was separated.

FSUB:

FSUB is implemented by calling FADD with the sign of R1 changed.

FMUL:

FMUL begins with special case handler implemented in a similar way to FADD but with different outputs for each case. For instance if both are infinity then the answer must be NaN. Then R10, R0 and R1 are stored. To conform to APCS R0 and R1 are moved to R10 and R9 respectively and then stored in the stack. R10 is stored since it is used for division (explained later).

Then we move to MUL. This uses MUL32 that we designed earlier in the autumn term. For ease of debugging, I used the version of MUL32 provided to us by Dr Clarke. The multiply that we need to perform is between two numbers of the 1.frac i.e. 24 bit numbers. So, the product will be 48 bits.

The product is stored in R8:R6. The algorithm is simple: first we do a 16x16 bit multiply (R6:=R0[15:0]*R1[15:0]) then two 8x16 bit multiplies (R8[7:0]:R6[31:16]:=R0[23:16]*R1[15:0] and R8[7:0]:R6[31:16]:=R0[15:0]*R1[23:16])and one 8x8 bit multiply (R8:=R0[23:16]*R1[23:16]). R0 and R1 are used for the inputs and R2 for the multiply output of MUL32. R2 is then moved around to R6 and R8. To optimise the code by reducing the number of multiplications, the lower half of MUL32 is called MUL24 and this bit is alone accessed when doing the 8x16 and 8x8 multiplications.

These two registers R6 and R8 are then normalised. Since we will have two 24 bit numbers definitely (denormalised numbers are also normalised) the product can be either 47 bits or 48 bits. So the answer is normalised accordingly by Shifter. Similar to ADD R9 is used to keep overflow and round the answer. After

rounding up if the mantissa is still 0x800000 then we add one to the exponent and call Result2.

Division:

```
Express D as M × 2^e where 1 ≤ M < 2 (standard floating point
representation)
D' := D / 2^(e+1)   // scale between 0.5 and 1, can be performed with bit
shift / exponent subtraction
N' := N / 2^(e+1)
X := 48/17 - 32/17 × D'   // precompute constants with same precision
as D
repeat ⌈log2 (P+1)/(log2 17)⌉ times   // can be precomputed based on fixed P
    X :=  X × (2 - D' × X)
end
return N' × X
```

The above is taken from the wiki link provided. For IEEE 754 single precision the loop has to be repeated 3 times.

Division if implemented using the existing sub-routines FADD, FSUB and FMUL to perform these arithmetic operations. Initially, the inputs are checked for special cases and handled. Then, we set the exponent of R0 as:

R4 := R4 – R6 + 127 – 1 (the last part is to create N’)

To abide with the algorithm, we also set R6 now to 126 (accounting for the fact that we have to subtract one from 127 which is zero exponent). To facilitate the use of the existing arithmetic sub-routines, we need to put the numbers together as IEEE 754 floating point numbers. This is done by using Result2 as mentioned earlier. Upon completing this, the required arithmetic is then carried out. So for the necessary loop we use R10 as the loop counter. When the loop is completed, we multiply the result with N’.

It is convenient that the results of all subroutines are moved to R0. So we just need to store the result of each iteration into the stack and access it repeatedly without changing the pointer till the end of the loop. The constants 48/17 and 32/17 are stored in the memory under TEMP as 0X3FF0F0F1 and 0x4034B4B5. **Hence, it is important that R3 be incremented while loading R0 and R1.**

Coding Trade-offs:

While coding, several trade-offs were made. For instance, to reduce the size of the code in terms of number of instructions, sub-routines can be re-used like functions in a C++ program. However, this comes at the cost of increasing the size of the stack and the number of cycles for branching.

Similarly, when not used too many times, the necessary code can be repeated the required number of times. For instance, the code for rounding-off is repeated as branching would waste a good number of cycles almost equal to the number of cycles to implement the rounding.

Compactness:

To ensure compactness, several trade-offs were made some of which are stated above. As handling special inputs was not required, those bits can be removed. However, I didn't want to remove bits from a working code. However, I will mention **how many instructions can be reduced:**

32 lines from FDIV

29 lines from FMUL

35 lines from FADD

+	-Inf	-1	-0	0	1	Inf	NaN
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	NaN	NaN
-1	-Inf	-2	-1	-1	0	Inf	NaN
-0	-Inf	-1	-0	0	1	Inf	NaN
0	-Inf	-1	0	0	1	Inf	NaN
1	-Inf	0	1	1	2	Inf	NaN
Inf	NaN	Inf	Inf	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

-	-Inf	-1	-0	0	1	Inf	NaN
-Inf	NaN	-Inf	-Inf	-Inf	-Inf	-Inf	NaN
-1	Inf	0	-1	-1	-2	-Inf	NaN
-0	Inf	1	0	-0	-1	-Inf	NaN
0	Inf	1	0	0	-1	-Inf	NaN
1	Inf	2	1	1	0	-Inf	NaN
Inf	Inf	Inf	Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

*	-Inf	-1	-0	0	1	Inf	NaN
-Inf	Inf	Inf	NaN	NaN	-Inf	-Inf	NaN
-1	Inf	1	0	-0	-1	-Inf	NaN
-0	NaN	0	0	-0	-0	NaN	NaN
0	NaN	-0	-0	0	0	NaN	NaN
1	-Inf	-1	-0	0	1	Inf	NaN
Inf	-Inf	-Inf	NaN	NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

/	-Inf	-1	-0	0	1	Inf	NaN
-Inf	NaN	Inf	Inf	-Inf	-Inf	NaN	NaN
-1	0	1	Inf	-Inf	-1	-0	NaN
-0	0	0	NaN	NaN	-0	-0	NaN
0	-0	-0	NaN	NaN	0	0	NaN
1	-0	-1	-Inf	Inf	1	0	NaN
Inf	NaN	-Inf	-Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Removing these would also save a good number of cycles and hence make the corresponding sub-routines faster. The reason divide takes a large number of cycles is due to the repeated saving of registers on the stack for each call of FADD, FSUB and FMUL. This can be avoided but the solution would involve making the code larger and hence this wasn't implemented.

Working:

After successive testing and implementation of the code stage by stage, several versions were made. The final version that is being submitted is v10.

Instructions for Use:

1) Enter inputs in place of the **first two values in TEMP**. *Do not alter the third and fourth numbers as they are important for division*
2) Use R3 for loading the registers R0 to R1
3) While loading the second value use pre-incrementation as this is necessitated by the code. A sample of the MAIN routine controlling the code can be seen at the top of the code in v10:

```
MAIN    ADR     R3, TEMP
        LDR     R0, [R3]
        LDR     R1, [R3, #4]!
        BL      FDIV
        END
```

4) Please do not change the way TEMP is called. Just change the values in first two words of TEMP and the arithmetic sub-routine to be called.