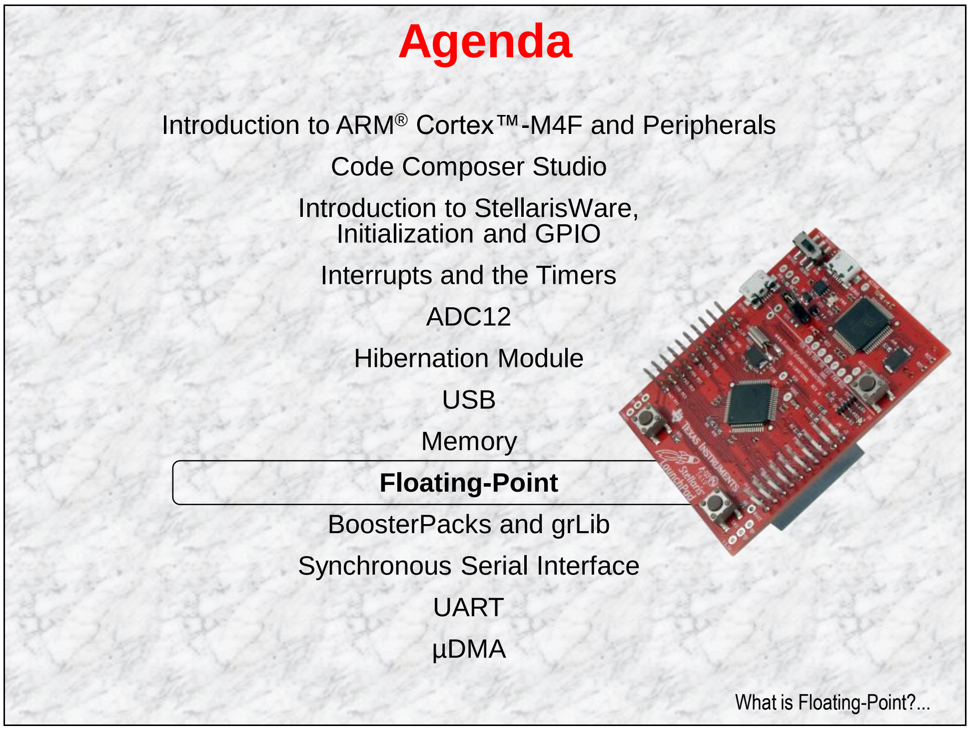
Floating-Point Unit

Introduction

This chapter will introduce you to the Floating-Point Unit (FPU) on the LM4F series devices. In the lab we will implement a floating-point sine wave calculator and profile the code to see how many CPU cycles it takes to execute.



Chapter Topics

Floating-Point Unit..9-1

Chapter Topics..9-2

What is Floating-Point and IEEE-754?..9-3

Floating-Point Unit..9-4

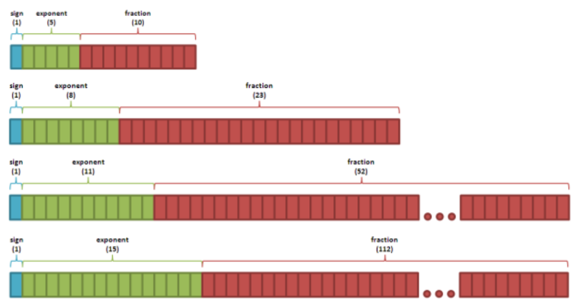
CMSIS DSP Library Performance..9-6

Lab 9: FPU..9-7
Objective..9-7
Procedure..9-8

What is Floating-Point and IEEE-754?

What is Floating-Point?

- Floating-point is a way to represent *real* numbers on computers
- IEEE floating-point formats:
  - Half (16-bit) →
  - Single (32-bit) →
  - Double (64-bit) →
  - Quadruple (128-bit) →



What is IEEE-754?...

What is IEEE-754?

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Symbol	Sign (s)	Exponent (e)								Fraction (f)																						

1 bit — 8 bits — 23 bits

Decimal Value = $(-1)^s\,(1+f)\,2^{e-bias}$

where: $f = \sum[(b_{-i})2^{-i}]\ \forall\ i \in (1,23)$

bias = 127 for single precision floating-point

Symbol	s	e								f																						
Example	0	1	0	0	0	0	1	1	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

sign = $(-1)^0$ = $[1]_{10}$ exponent = $[10000110]_2 = [134]_{10}$ fraction = $[0.11010000100000000000000]_2 = [0.814453]_{10}$

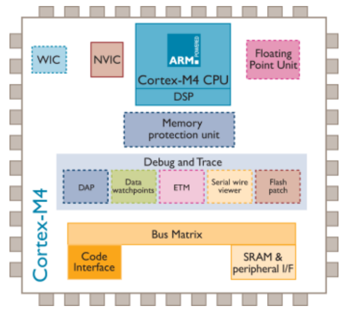
$$\textit{Decimal Value} = (-1)^s \times (1+f) \times 2^{e-bias}$$
$$= [1]_{10} \times ([1]_{10} + [0.814453]_{10}) \times [2^{134-127}]_{10}$$
$$= [1.814453]_{10} \times 128$$
$$= [232.249]_{10}$$

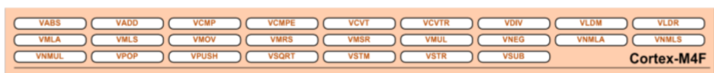
FPU...

Floating-Point Unit

Floating-Point Unit (FPU)

- **The FPU provides floating-point computation functionality that is compliant with the IEEE 754 standard**
- **Enables conversions between fixed-point and floating-point data formats, and floating-point constant instructions**
- **The Cortex-M4F FPU fully supports single-precision:**
  - **Add**
  - **Subtract**
  - **Multiply**
  - **Divide**
  - **Single cycle multiply and accumulate (MAC)**
  - **Square root**





Modes of Operation...

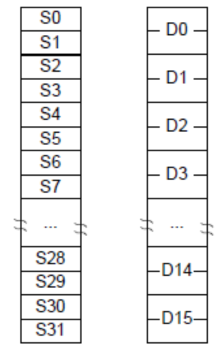
Modes of Operation

- **There are three different modes of operation for the FPU:**
  - **Full-Compliance mode –** In Full-Compliance mode, the FPU processes all operations according to the IEEE 754 standard in hardware. **No support code is required.**
  - **Flush-to-Zero mode –** A result that is very small, as described in the IEEE 754 standard, where the destination precision is smaller in magnitude than the minimum normal value before rounding, is replaced with a zero.
  - **Default NaN (not a number) mode –** In this mode, the result of any arithmetic data processing operation that involves an input NaN, or that generates a NaN result, returns the default NaN. (**0 / 0 = NaN)**

FPU Registers...

FPU Registers

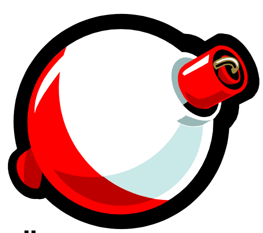
- Sixteen 64-bit double-word registers, D0-D15
- Thirty-two 32-bit single-word registers, S0-S31



Usage...

FPU Usage

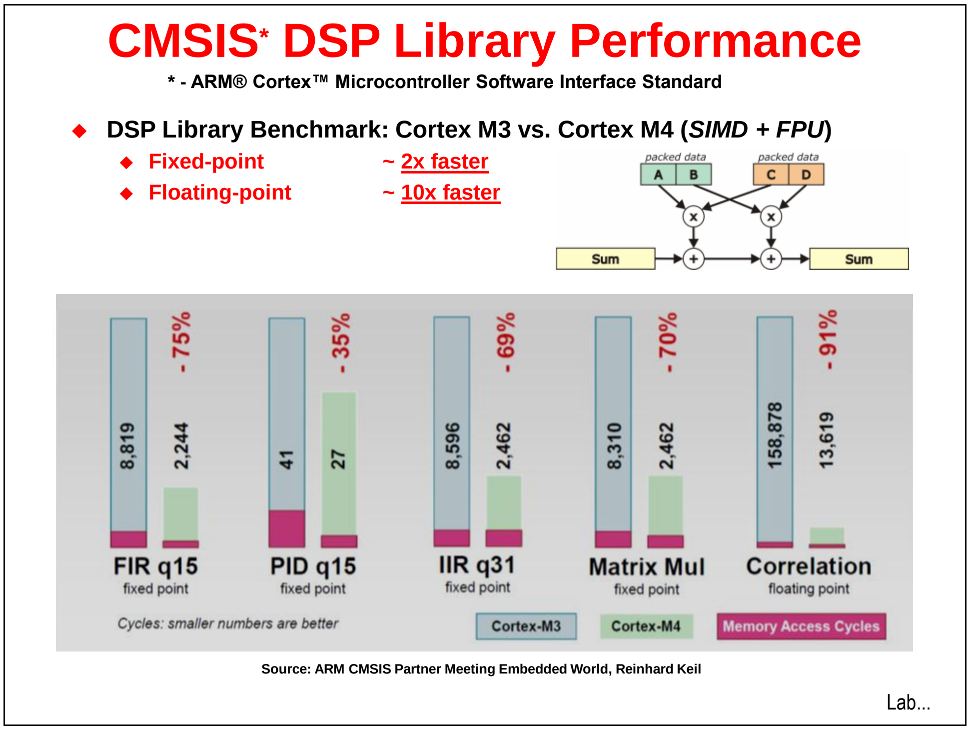
- **The FPU is disabled from reset.** You must **enable it*** before you can use any floating-point instructions. The processor must be in privileged mode to read from and write to the Coprocessor Access Control (CPAC) register.
- **Exceptions:** The FPU sets the cumulative exception status flag in the FPSCR register as required for each instruction. The FPU does not support user-mode traps.
- The processor can reduce the exception latency by using **lazy stacking***. This means that the processor reserves space on the stack for the FPU state, but does not save that state information to the stack.



*** with a StellarisWare API function call**

CMSIS...

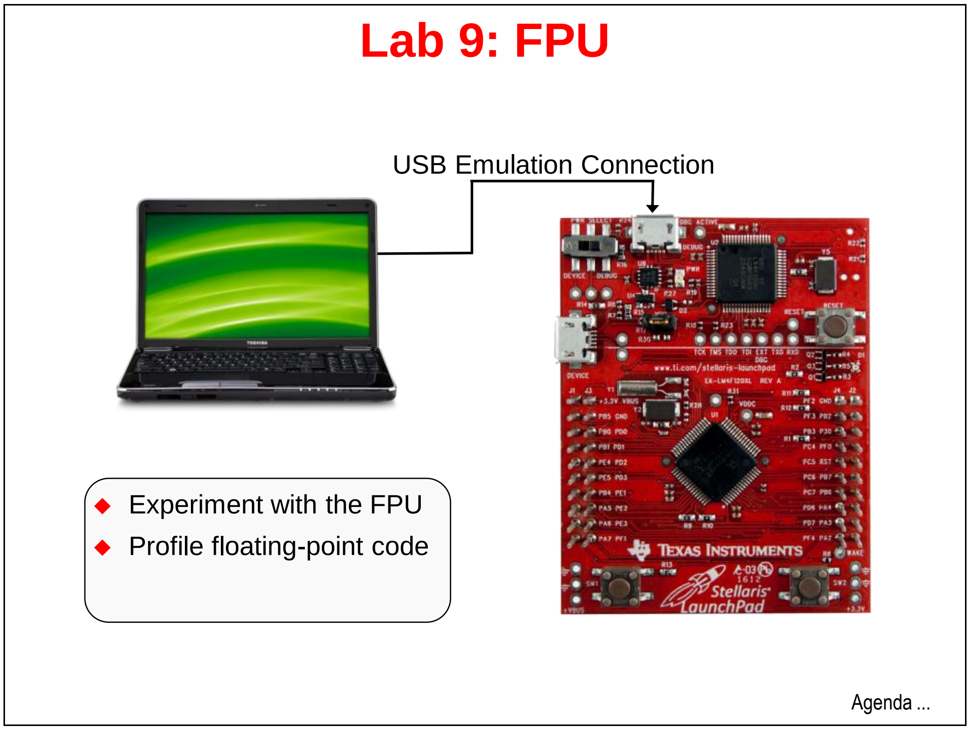
CMSIS DSP Library Performance



Lab 9: FPU

Objective

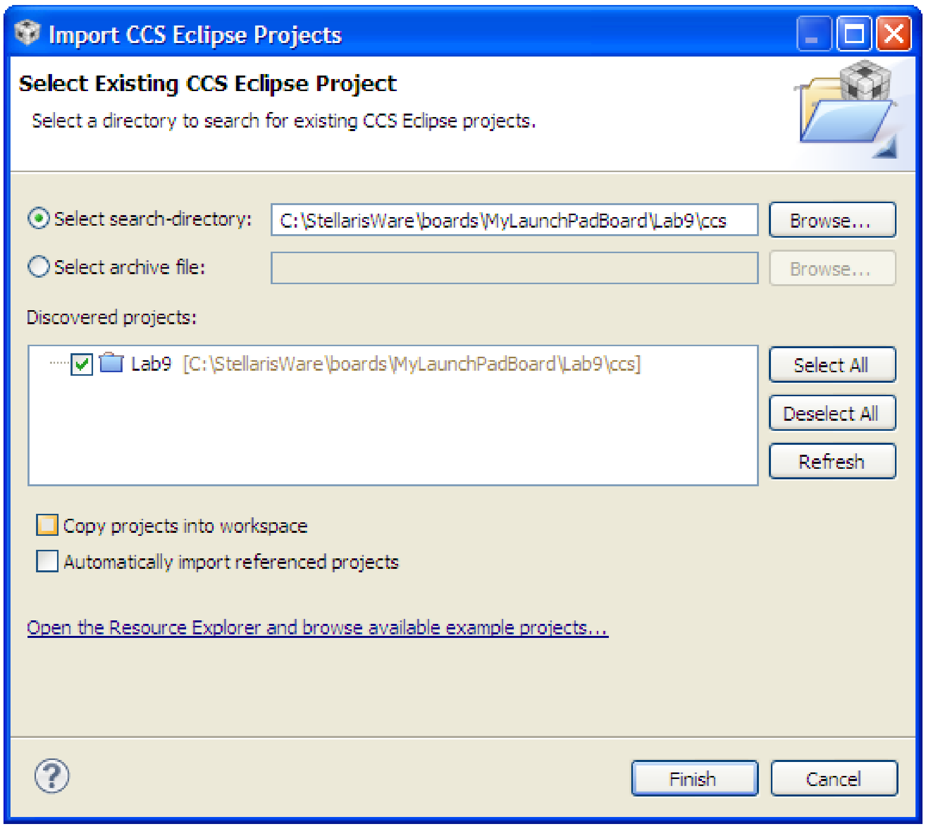
In this lab you will enable the FPU to run and profile floating-point code.



Procedure

Import Lab9

1. We have already created the Lab9 project for you with `main.c`, a startup file and all necessary project and build options set. Maximize Code Composer and click Project → Import Existing CCS Eclipse Project. Make the settings shown below and click Finish. Make sure that the "Copy projects into workspace" checkbox is **unchecked**.



The code is fairly simple. We'll use the FPU to calculate a full cycle of a sine wave inside a 100 datapoint long array.

Browse the Code

2. In order to save some time, we're going to browse existing code rather than enter it line by line. Open `main.c` in the editor pane and copy/paste the code below into it.

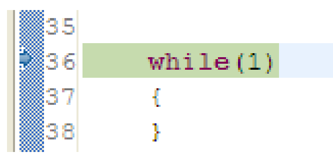
```
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/fpu.h"
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define SERIES_LENGTH 100

float gSeriesData[SERIES_LENGTH];

int dataCount = 0;

int main(void)
{
    float fRadians;

    ROM_FPULazyStackingEnable();
    ROM_FPUEnable();

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

    fRadians = ((2 * M_PI) / SERIES_LENGTH);

    while(dataCount < SERIES_LENGTH)
    {
        gSeriesData[dataCount] = sinf(fRadians * dataCount);

        dataCount++;
    }

    while(1)
    {
    }
}
```

3. At the top of `main.c`, look first at the includes, because there are a couple of new ones:
   - **`math.h`** – the code uses the `sinf()` function prototyped by this header file
   - **`fpu.h`** – support for Floating Point Unit
4. Next is an `ifndef` construct. Just in case `M_PI` is not already defined, this code will do that for us.
5. Types and defines are next:
   - **`SERIES_LENGTH`** – this is the depth of our data buffer
   - **`float gSeriesData[SERIES_LENGTH]`** – an array of floats `SERIES_LENGTH` long
   - dataCount – a counter for our computation loop

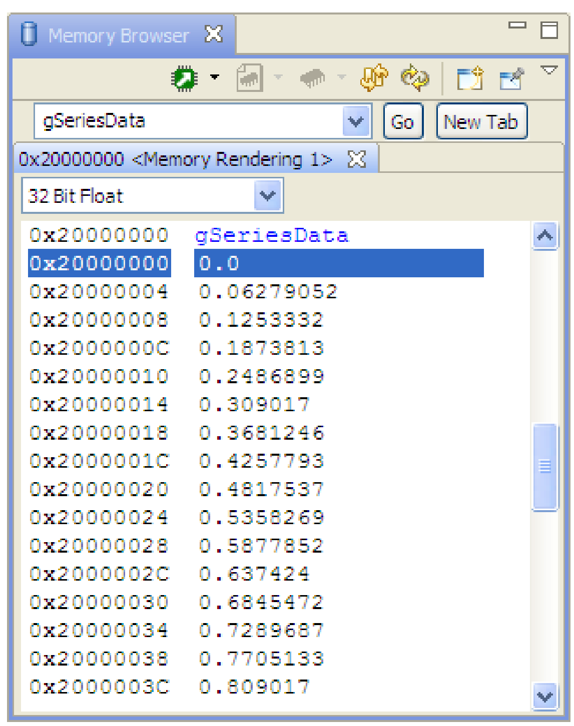
6. Now we've reached main():
   - We'll need a variable of type float called `fRadians` to calculate sine
   - Turn on Lazy Stacking (as covered in the presentation)
   - Turn on the FPU (remember that from reset it is off)
   - Set up the system clock for 50MHz
   - A full sine wave cycle is 2π radians. Divide 2π by the depth of the array.
   - The `while()` loop will calculate the sine value for each of the 100 values of the angle and place them in our data array
   - An endless loop at the end

Build, Download and Run the Code

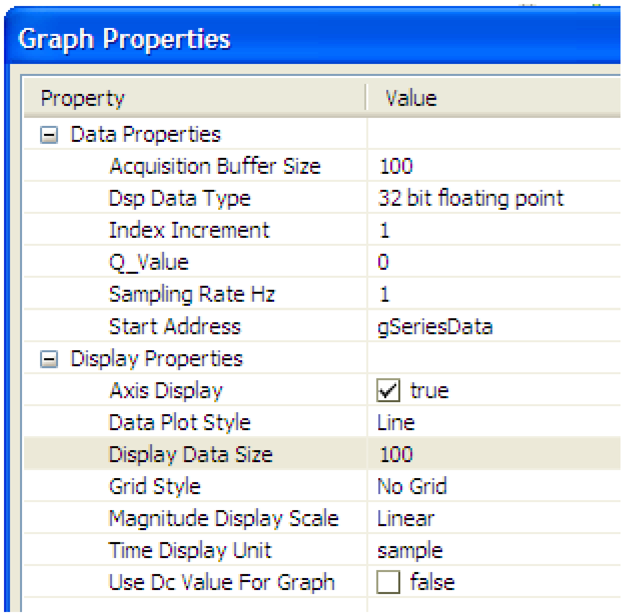
7. Click the Debug button to build and download the code to the LM4F120H5QR flash memory. When the process completes, click the Resume button to run the code.
8. Click the Suspend button to halt code execution. Note that execution was trapped in the `while(1)` loop.



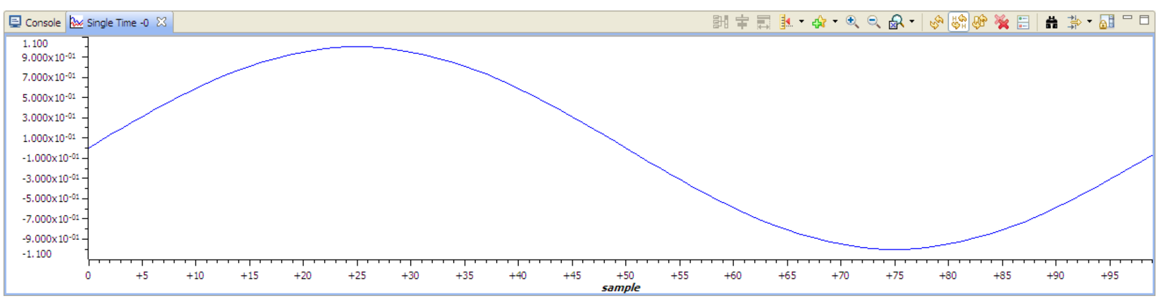
9. If your Memory Browser isn't currently visible, Click View → Memory Browser on the CCS menu bar. Enter `gSeriesData` in the address box and click Go. In the box that says Hex 32 Bit – TI Style, click the down arrow and select 32 Bit Float. You will see the sine wave data in memory like the screen capture below:



10. Is that a sine wave? It's hard to see from numbers alone. We can fix that. On the CCS menu bar, click Tools → Graph → Single Time. When the Graph Properties dialog appears, make the selections show below and click OK.



You will see the graph below at the bottom of your screen:



Profiling the Code

11. An interesting thing to know would be the amount of time it takes to calculate those 100 sine values.

    On the CCS menu bar, click View → Breakpoints. Look in the upper right area of the CCS display for the Breakpoints tab.

12. Remove any existing breakpoints by clicking Run → Remove All Breakpoints. In the `main.c`, set a breakpoint by double-clicking in the gray area to the left of the line containing:

```
fRadians = ((2 * M_PI) / SERIES_LENGTH);
```

```
26
27      fRadians = ((2 * M_PI) / SERIES_LENGTH);
28
29      while(dataCount < SERIES_LENGTH)
30      {
31          gSeriesData[dataCount] = sinf(fRadians * dataCount);
32
33          dataCount++;
34      }
```

13. Click the Restart button  to restart the code from `main()`, and then click the Resume button to run to the breakpoint.
14. Right-click in the Breakpoints pane and Select Breakpoint (Code Composer Studio) → Count event. Leave the Event to Count as Clock Cycles in the next dialog and click OK.
15. Set another Breakpoint on the line containing `while(1)` at the end of the code. This will allow us to measure the number of clock cycles that occur between the two breakpoints.

```
26
27      fRadians = ((2 * M_PI) / SERIES_LENGTH);
28
29      while(dataCount < SERIES_LENGTH)
30      {
31          gSeriesData[dataCount] = sinf(fRadians * dataCount);
32
33          dataCount++;
34      }
35
36      while(1)
37      {
38      }
```

16. Note that the count is now 0 in the Breakpoints pane. Click the Resume button to run to the second breakpoint. When code execution reaches the breakpoint, execution will stop and the cycle count will be updated. Our result is show below:

Variables | Expressions | Registers | Breakpoints

Identity	Name	Condition	Count	Action
Count Event	Count Event		34996	
main.c, line 27 (	Breakpoint		0 (0)	Remain Halted
main.c, line 36 (	Breakpoint		0 (0)	Remain Halted

17. A cycle count of 34996 means that it took about 350 clock cycles to run each calculation and update the dataCount variable (plus some looping overhead). Since the System Clock is running at 50Mhz, each loop took about 7µS, and the entire 100 sample loop required about 700 µS.
18. Right-click in the Breakpoints pane and select Remove All, and then click Yes to remove all of your breakpoints.
19. Click the Terminate button to return to the CCS Edit perspective.
20. Right-click on Lab9 in the Project Explorer pane and close the project.
21. Minimize Code Composer Studio.

You're done.