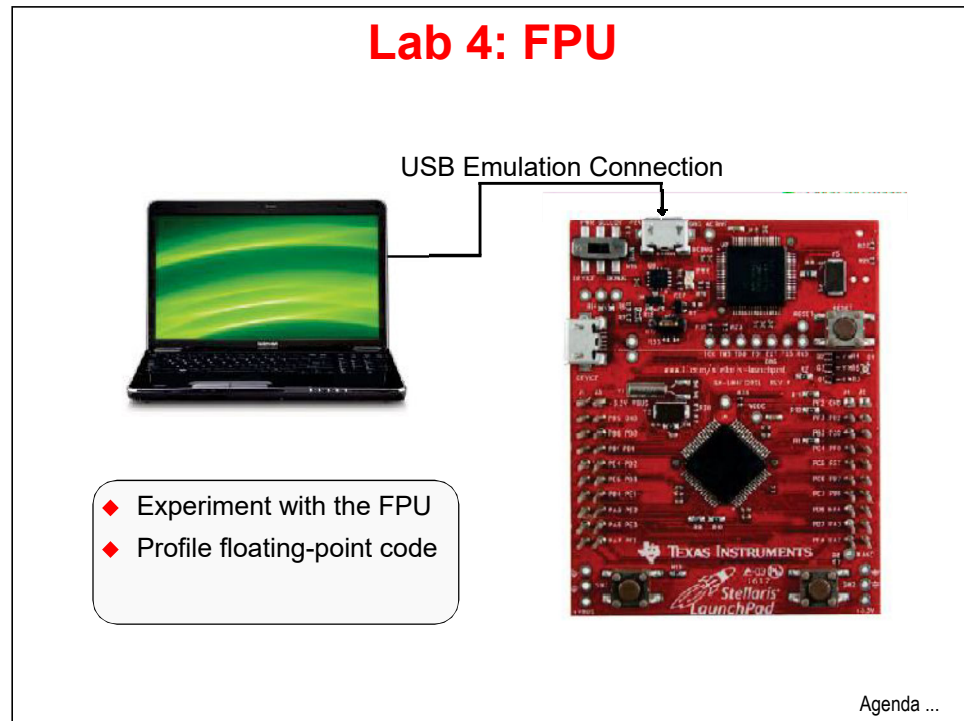


Lab 4: FPU

Objective

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



Procedure

Import Lab9

1. Create the Lab project in `main.c`, a startup file and all necessary project and build options set. →

The code is fairly simple. We'll use the FPU to calculate a full cycle of a sine wave inside a 100-data point 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++;
    }

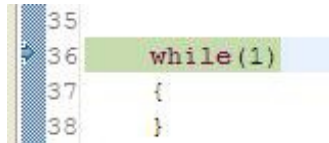
    while(1)
    {
    }
}
```

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 2 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.



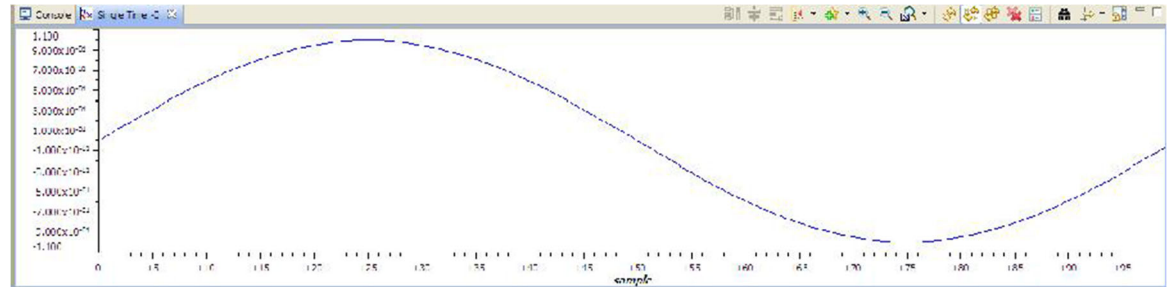
9. If your Memory Browser isn't currently visible, Click View → Memory Browser on the debug mode.

—

Is that a sine wave? It's hard to see from

→ →

10. You will see the graph below at the bottom of your screen (draw it! Or show it on the analyzer!):



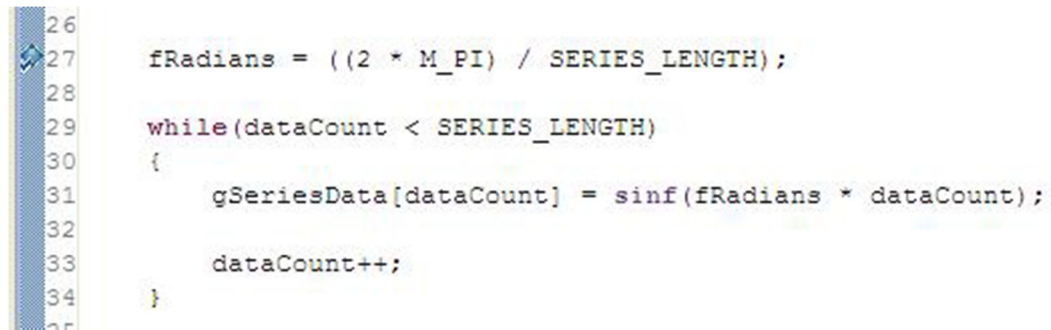
Profiling the Code

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.

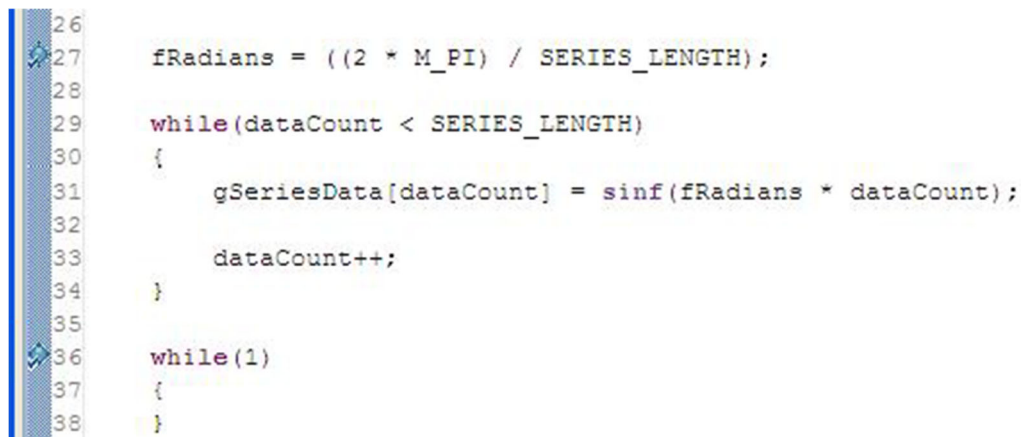
12. Remove any existing breakpoints by clicking Run → Remove All Breakpoints. In the `main.c`, set a breakpoint by double-clicking in the gray area to the left of the line containing:

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

A screenshot of the Code Composer Studio editor showing a C program. Line 27 has a breakpoint set, indicated by a blue diamond icon in the left margin. The code is as follows:

```
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
```

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.

A screenshot of the Code Composer Studio editor showing the same C program as before, but with a second breakpoint set on line 36. The code is as follows:

```
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.

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 $7\mu\text{S}$, and the entire 100 sample loop required about $700\mu\text{S}$.
18. Right-click in the Breakpoints pane and select Remove All, and then click Yes to remove all of your breakpoints.
19. Draw the flowchart!.

You're done.

