

EE445L – Lab2: Performance Debugging

Harley Ross and Dalton Altstaetter

2/4/14

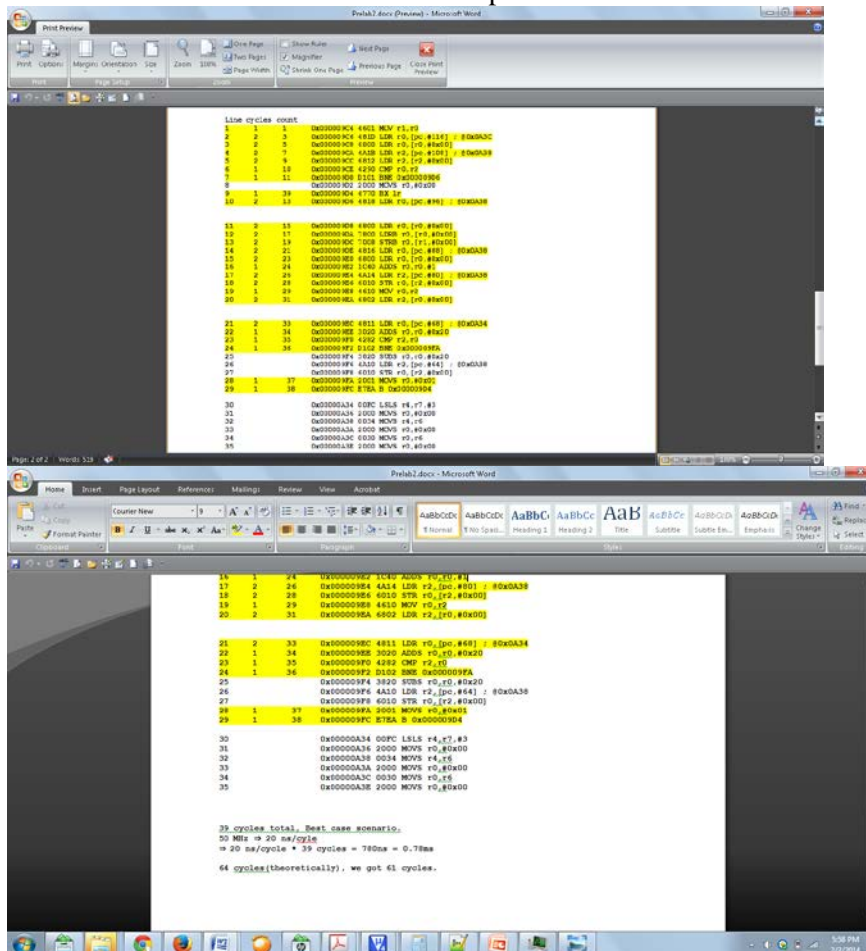
GOALS

The goals of this lab were to develop software debugging techniques for performance and profiling using hardware and software, and then comparing the advantages and disadvantages of each. We also needed to learn how to pass data using the FIFO queue and become familiar with oscilloscope and the logic analyzer. We then needed to observe critical sections and get a head start on the draw functions needed for Lab 3.

MEASUREMENT DATA

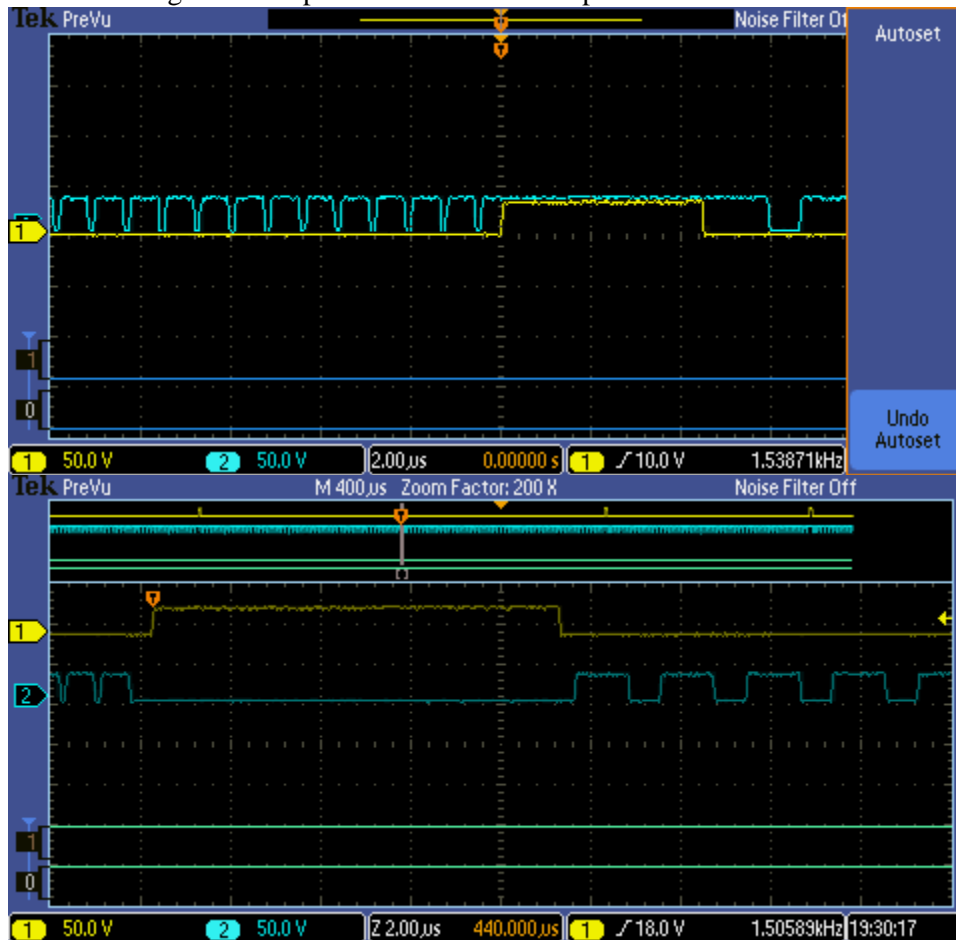
Part 4

The cycle count is 61 and the execution speed is $61 \times 20\text{ns} = 1.28$ micro seconds. These pictures show the instruction used to determine execution speed.



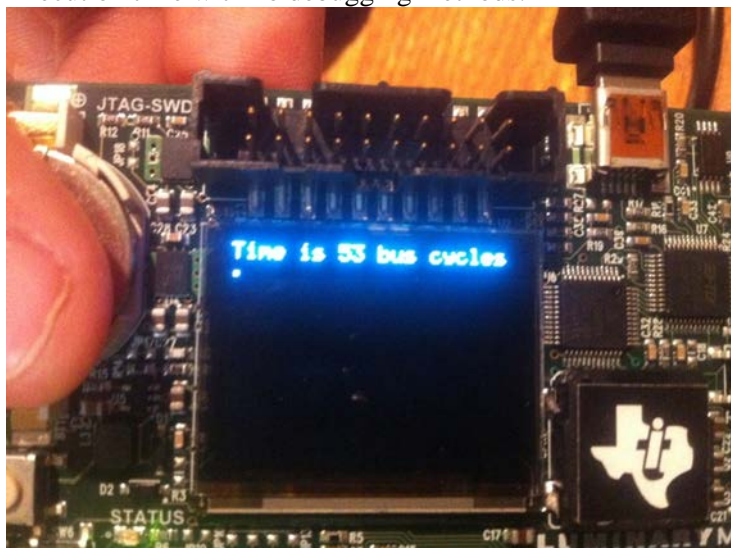
Parts B and C

Profiles during an interrupt where 1 is the interrupt and 2 is normal execution.



Part D

Execution time with no debugging methods.



Execution time using the dump method.

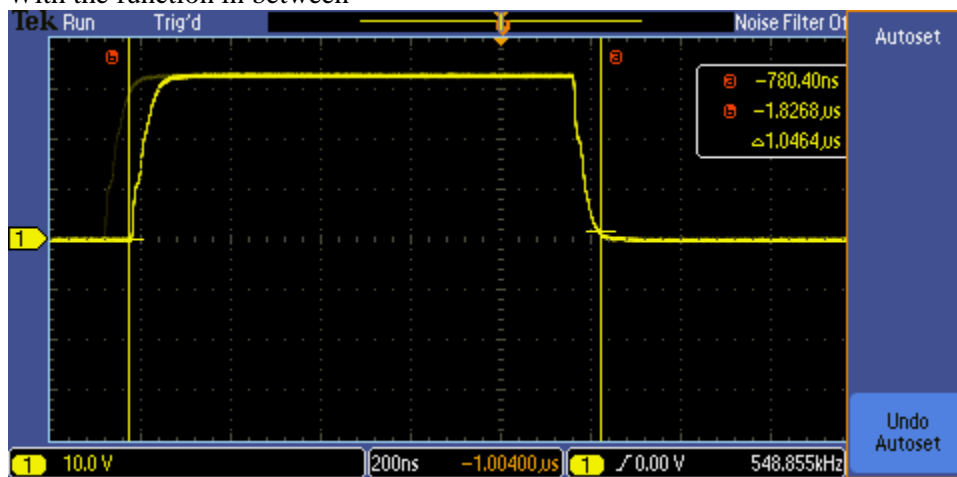


Execution time using the Printf method.

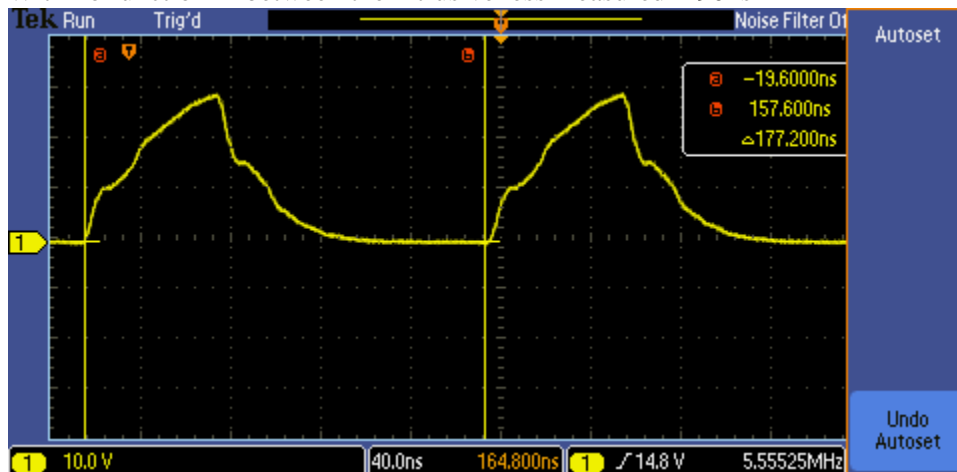


Part E

With the function in between



With no function in between the intrusiveness measured ~90ns



Part F and H

This screenshot shows there are no errors so the values are not being changed during the interrupt.

The screenshot shows the Keil uVision4 IDE with the disassembly of the main function. The registers window on the left shows the core registers. The disassembly window in the center shows the code for the main function, which includes a while loop that calls the RxFifo_Get function. The watch window on the right shows the values of the variables in the main function, including the LineHistogram, Errors, EnteredCount, and LineHistogramAddress. The command window at the bottom shows the command 'WS 1, 'result' WS 1, 'letter'.

Name	Location/Value	Type
main	0x00000514	int
i	0x00000000	aut
returnaddr...	<not in scope>	aut

Name	Value	Type
LineHistogram	0x200000A0	LineHistogram
Errors	0	unsigned short
EnteredCount	2227	unsigned long
LineHistogramAddress	0x00000000	int
result	0x00000000	int
letter	0x00	char

In this screenshot, store and move instructions have no interrupts when executing and these are the critical sections.

The screenshot shows the uVision4 IDE with the project 'FIFO_1968'. The registers window displays the state of the processor registers, with R0-R15 showing various values. The watch window shows a list of variables, including 'EnteredCount' with a value of 156304. The code window shows the implementation of the FIFO test, which includes a critical section for updating the FIFO pointer.

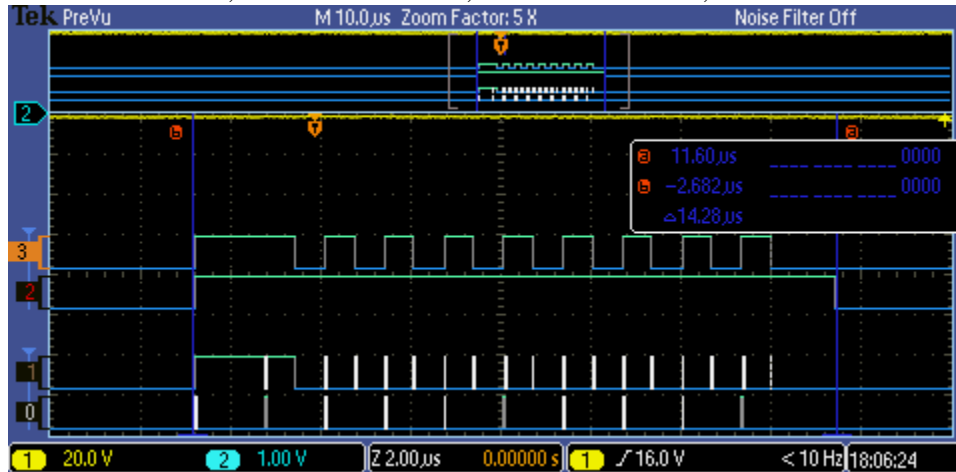
Part G

In this screenshot, there are many errors because the critical sections are being interrupted.

The screenshot shows the uVision4 IDE with the project 'FIFO_1968'. The registers window displays the state of the processor registers, with R0-R15 showing various values. The watch window shows a list of variables, including 'Errors' with a value of 13047. The code window shows the implementation of the FIFO test, which includes a critical section for updating the FIFO pointer. The 'Find in Files' window shows the location of the critical section in the code.

Part H

F3 = TXFIFOPUT, F2 = INTERRUPT, F1 = RXFIFOPUT, F0 = TXFIFOGET



ANALYSIS AND DISCUSSION

1. We did not get the same results when measuring the execution speed of RxFifo_Get. We got different results because the dump (87 cycles) was much less intrusive than the Printf (385,079 cycles)
2. We would use the system clock to measure execution speed because we can store the values and operate on them to determine the minimum, maximum, and average speed. The scope can't be operated on inside the software.
3. We would use the Printf method to measure execution speed because it would be negligible compared to the 20 seconds and it's easier for the user and doesn't require extra equipment (scope).
4. Minimally intrusive means the debugging method has a negligible effect on the system being debugged.
5. The two necessary components collected during a profile are turning the bit on and off.
6. The store and the move functions are the critical sections. You save the current enabled interrupts, then all disable interrupts during those instructions so that their values cannot be changed, then restore the enabled interrupts after the critical section has ended.

SOURCE CODE

```
#define SCALE 4500
#define NUMPIXELS 300
#define POSITIONS 60
#define MINUTEHANDLENGTH 33 // length of the minute hand on OLED
#define HOURHANDLENGTH 24 // length of the hour hand on OLED
#define NULL 0
#define XPIVOT 55
#define YPIVOT 47
```

```
void RIT128x96x4_Line(int x1, int y1, int x2, int y2, unsigned char color)
{
```

```
    int i;
```

```

    int deltaX;
    int deltaY;
    int width;
    int height;
    int tempX;
    int tempY;

    // used to highlight a particular pixel of the OLED
    const unsigned char dot[] = {0xFF};
    width = x2-x1;
    height = y2-y1;

    // need to find the spacing between the two coordinates
    // (deltaX,deltaY) and scale the input by 2500 so that we
    // don't have dropout from integer division by dividing
    // the interval into NUMPIXELS equal intervals
    deltaX = (width*SCALE)/NUMPIXELS;
    deltaY = (height*SCALE)/NUMPIXELS;

    Output_Color(15);
    for(i = 0; i < NUMPIXELS; i++)
    {
        // find the new (X,Y) scaled coordinates
        tempX = x1*SCALE + i*deltaX;
        tempY = y1*SCALE + i*deltaY;

        // convert coordinates back in OLED range and print to the OLED
        RIT128x96x4ImageDraw(&dot[0], tempX/SCALE, tempY/SCALE, 2, 1);
    }
}

void
RIT128x96x4Clear(void)
{
    static const unsigned char pucCommand1[] = { 0x15, 0, 63 };
    static const unsigned char pucCommand2[] = { 0x75, 0, 127 };
    unsigned long ulRow, ulColumn;

    //
    // Clear out the buffer used for sending bytes to the display.
    //
    *(unsigned long *)&g_pucBuffer[0] = 0;
    *(unsigned long *)&g_pucBuffer[4] = 0;

    //
    // Set the window to fill the entire display.
    //
    RITWriteCommand(pucCommand1, sizeof(pucCommand1));
    RITWriteCommand(pucCommand2, sizeof(pucCommand2));
    RITWriteCommand(g_pucRIT128x96x4HorizontalInc,

```

```

        sizeof(g_pucRIT128x96x4HorizontalInc));

//
// Loop through the rows
//
for(ulRow = 0; ulRow < 96; ulRow++)
{
    //
    // Loop through the columns. Each byte is two pixels,
    // and the buffer hold 8 bytes, so 16 pixels are cleared
    // at a time.
    //
    for(ulColumn = 0; ulColumn < 128; ulColumn += 8 * 2)
    {
        //
        // Write 8 clearing bytes to the display, which will
        // clear 16 pixels across.
        //
        RITWriteData(g_pucBuffer, sizeof(g_pucBuffer));
    }
}
}

```