TEXAS INSTRUMENTS

Application Report
SPRAAL8A – July 2010

Advanced Profiling with XDS560 Trace

Dan Rinkes, Nipuna Gunasekera *SDS Applications Engineering*

ABSTRACT

While many developers consider XDS560 Trace a valuable tool for debugging difficult scenarios, it also provides quick and effective profiling techniques. With the capability of Trace to capture timestamps on each sample and the flexibility of Advanced Event Triggering to capture data at the appropriate locations, XDS560 Trace provides unique profiling capabilities. This application note focuses on a few such scenarios.

A prerequisite for this work is Code Composer Studio 3.3. The scenarios provided in this document are for ‘C64x and ‘C64x+ devices that support XDS560 Trace.

Contents

1 Introduction 2
2 Devices Supporting AET and Trace 2
3 Advanced Event Triggering Hardware Overview 3
3.1 Triggers and Trigger Types 3
3.2 AET Hardware on ‘C6455 4
4 Advanced Event Triggering Target Library 5
5 Code Composer Studio Configuration 6
6 Statistical Profiling 7
6.1 Target Application Configuration 8
6.2 Delta Value Considerations 9
6.3 Post-Processing 10
6.3.1 Saving the Data in .csv Format 10
6.3.2 Post-Processing the Data with a Script 10
6.3.3 Creating Intermediate Files 12
6.3.4 Processing the Entire Script in a Single Command Line 12
6.4 Results 13
7 Pipeline Stall Analysis 13
7.1 Target Application Configuration 14
7.2 Post-Processing 16
8 Conclusion 18
9 References 18

Tables

Table 1. Devices That Support AET/Trace 2
Table 2. Trigger Builder Detail for ‘C64x and ‘C64x+ Devices 4
Table 3. Statistical Profiling Functions 9
Table 4. Typical Stall Values for Cache Misses 17

Figures

Figure 1. A Simplified View of AET Hardware on the 'C64x+ Family 3
Figure 2. Sample Triggering Logic....4
Figure 3. Trace Setup Menu....6
Figure 4. Starting the Trace Display Capture....7
Figure 5. Function Sampling with Statistical Profiling....7
Figure 6. Sample Trace Statistical Profile....10
Figure 7. Sample Perldoc Output....11
Figure 8. Sample Statistical Profiling Data....13
Figure 9. Event Analysis Plug-in....14
Figure 10. Programming a Start Trace Job....15
Figure 11. Sample Pipeline Stall Analysis Data....15
Figure 12. Sample Pipeline Stall Output ('C64x)....17

1 Introduction

While XDS560 Trace is an effective tool for debugging complex scenarios within DSP code, it also provides unique ways to extend its functionality and generate quick and accurate profiling results. Results that had previously been obtained in a number of hours or days can now be gathered within minutes with more reliable results. You can quickly narrow down "hot spots" in your code where a small amount of optimization will have the greatest impact.

An overview of the AET (Advanced Event Triggering) hardware is provided so the profiling examples presented serve as a basis for creating additional ways to use AET and Trace to suit your own needs.

2 Devices Supporting AET and Trace

Check your device-specific data sheet to determine if it supports AET and Trace. The 17 devices are listed in Table 1.

Table 1. Devices That Support AET/Trace

TMS320C6454	TMS320C6414T	TMS320DM643
TMS320C6455	TMS320C6414	TMS320DM642
TMS320C6418	TMS320C6413	TMS320DM641
TMS320C6416T	TMS320C6412	TMS320DM640
TMS320C6416	TMS320C6411	
TMS320C6415T	TMS320C6410	
TMS320C6415		

There are no devices at this time that support AET but do not support Trace (or vice versa).

3 Advanced Event Triggering Hardware Overview

In order to set up complex triggering, you must have a basic understanding of the counters and trigger builders within the AET hardware.

Figure 1 shows a simple diagram of AET. Each of the input boxes on the left are configurable for a number of different scenarios. It's important to note that *only one* of each square box in the diagram exists in the hardware, while there are multiple copies of the OR gates and trigger builders tied to the outputs of the square boxes.

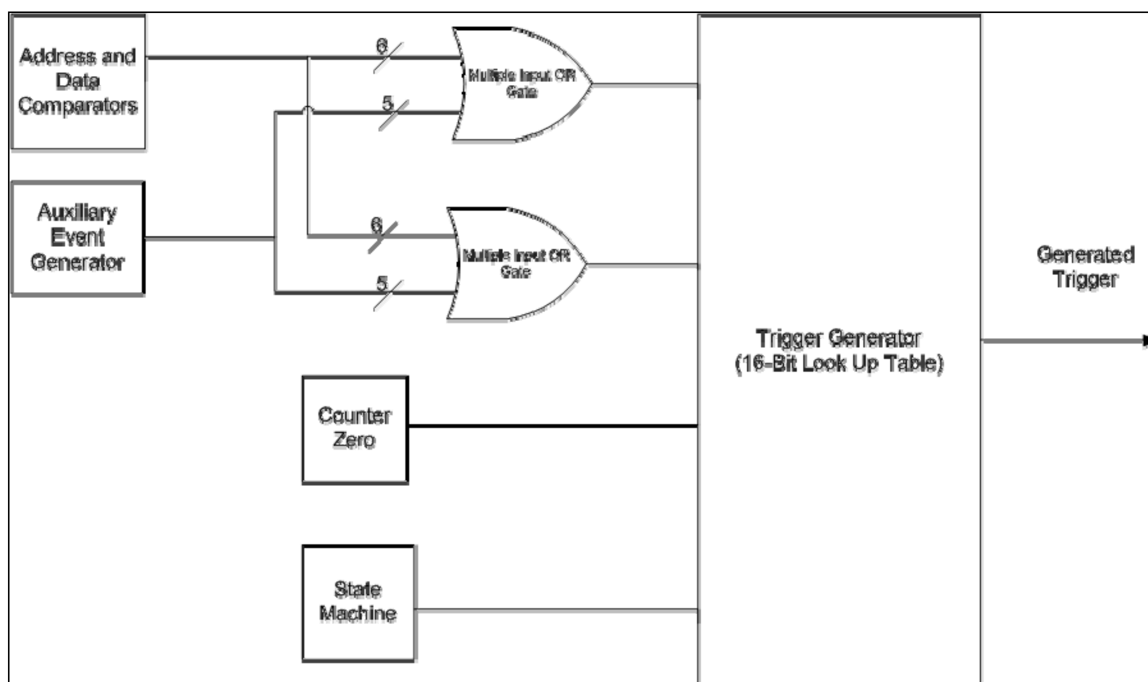


Figure 1. A Simplified View of AET Hardware on the 'C64x+ Family

3.1 Triggers and Trigger Types

Triggers are events that tell the AET hardware to do something. The AET hardware really consists of two parts, a) the hardware that detects events and combinations thereof, and b) the hardware that actually makes the desired events occur. The hardware that does the detection of events is the part you will need to configure.

There are three different types of triggers. They can be categorized by their width. The width of a trigger builder is the number of different triggers that it can generate for a single set of inputs. A 1-wide trigger builder can generate only a single trigger for a set of inputs. A 3-wide trigger builder can generate 3 different triggers based on a single set of inputs. This is particularly useful when using a 4-state machine because a single trigger builder can generate any of the signals to go to one of the other states. A 7-wide trigger builder is useful in generating the 7 different types of trace triggers (Program Counter, Read Address, Write Address, Read Data, Write Data, Timing, and PC Tag).

Table 2. Trigger Builder Detail for 'C64x and 'C64x+ Devices

Trigger Builder Width	Total	Typical Uses
1	6	Halt CPU, Interrupt, Start/Stop/Reload Counter
3	6	State Change
7	2	Trace Capture

Any AET job can be declared in words as "If X then do Y". X can be a simple or complex combination of AET signals, and Y is a trigger builder or number of trigger builders (or trigger generators).

3.2 AET Hardware on 'C6455

This section provides a more detailed example of what the Advanced Event Triggering Block looks like on a 'C6455 and how it is used to generate some of the trace scenarios. In the example, we show how AET implements a Trace job. This specific job is programmed to start 3 separate trace streams (PC Tag, Timing, and Write Data) when a specific value is written to a range of memory locations. In this case, the value we have chosen is 0x00001234 and the range of memory locations is 0x10000 – 0x10020. Sample values are written on the buses.

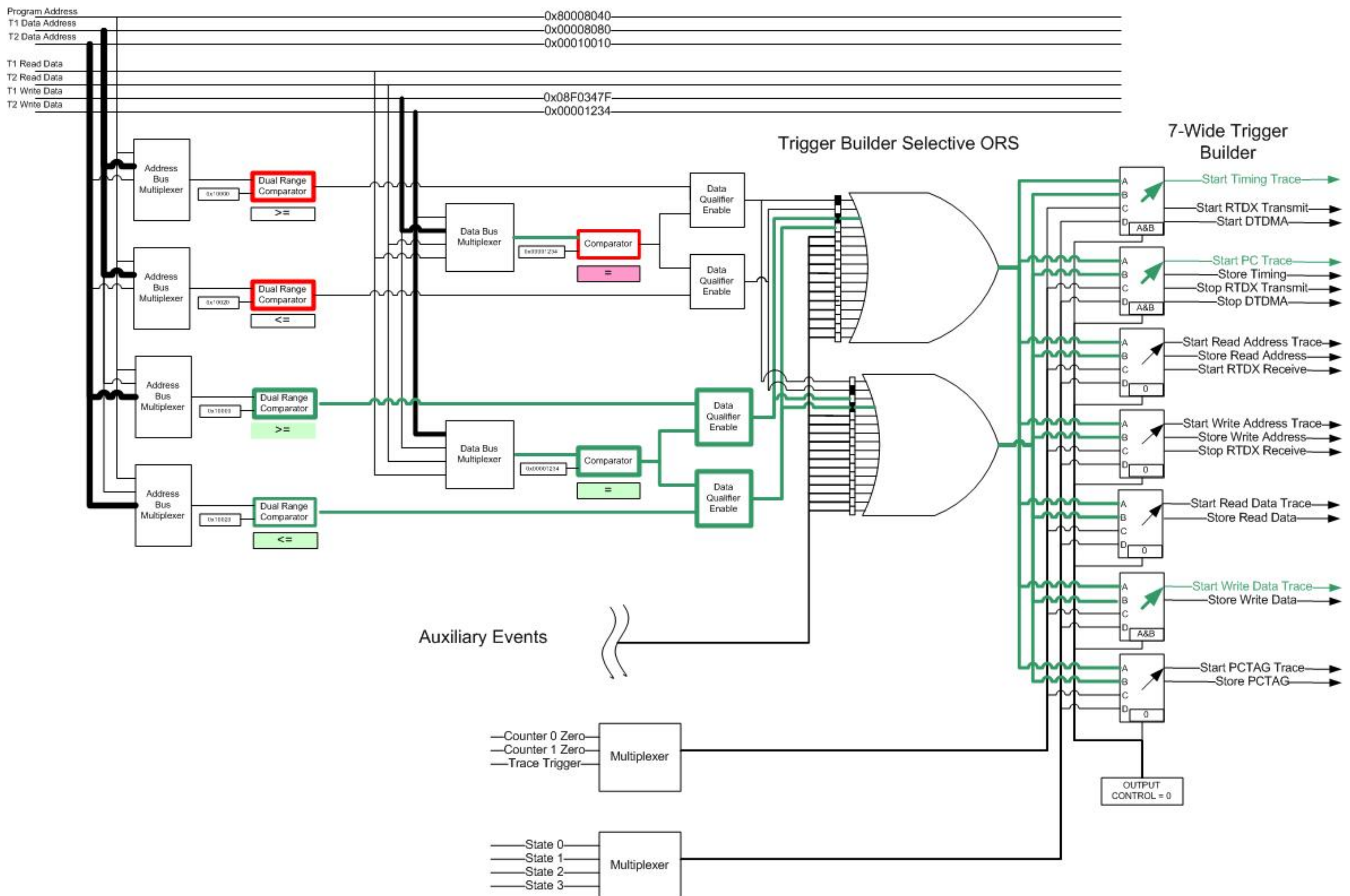


Figure 2. Sample Triggering Logic

Understanding all that is involved here is not completely trivial. To watch for a specific value written to a range of values, we need 4 dual-range comparators plus a data qualifier for each pair. In each pair, one comparator is set up to watch for a write address above the lower limit of the range (0x10000). The second one is set up to watch for a write address below the upper limit of the range (0x10020). The reason we need both pairs is that there are two 32-bit data buses (T1 and T2). The comparators can only watch for an address on one of these. So, in our configuration, the top pair monitors T1 and the bottom pair monitors T2. If we knew that this write would only occur on either T1 or T2, we could use only a single pair of comparators.

Each pair of comparators also has a data qualifier. The data qualifier is multiplexed between the T1 and T2 read and write buses. In this case, we are using the top data qualifier to monitor the T1 write bus and the bottom data qualifier to monitor the T2 write bus. When the comparators output a true value (as the lower set of comparators in this example do) the output is fed to the trigger builder selective OR gates. These OR gates mask off all of the values that are not used in this job and pass through the outputs that we care about. In the trigger builder, a lookup table is used to determine when to generate the triggers. (In this case, it's when the A and B inputs are both true.) The output control determines which set of triggers is used. Each N-wide trigger builder has the same set of inputs for each lookup table, but can use different logic to generate the triggers at different times. In this case, the logic on the Start Timing, Start PC, and Start Write Data trace triggers is programmed to fire whenever A and B are simultaneously true. All other triggers are programmed to never fire.

4 Advanced Event Triggering Target Library

In some cases, you need to reprogram the Advanced Event Triggering (AET) hardware from within the target application to optimize data gathering. Additionally you can better use the available AET resources by reprogramming the same units on the fly. A good example of this is stack overflow checking. On a ‘C6455 with *n* comparators, you would only be able to trap on *n*/4 DSP/BIOS TSK stacks (4 comparators needed to trap an upper/lower range on T1/T2). But with AETLIB you simply reprogram the same 4 comparators with your operating system's task switch function to trap on a new PC range for each task.

In these cases, you need to link in and use APIs from the Advanced Event Triggering Target Library (AETLIB). The AET Target Library is available from https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm (login required).

Documentation for the AET Target Library is provided in the docs\html and docs\win directories in the release package. See the documentation for additional details and uses of the AET Target Library.

Note: The Advanced Event Triggering Target Library must be used completely independent of the Advanced Event Analysis Code Composer Studio plug-in. All AET jobs must be programmed from either the plug-in OR the application. There is no means for resource sharing between the plug-in and the application.

5 Code Composer Studio Configuration

For each example, Code Composer Studio must be used to program and calibrate the XDS560 Trace receiver. For each of the following scenarios, you must perform the following steps prior to running the application on the target.

1. Open CCStudio 3.3.
2. Connect to the target by choosing **Debug→Connect**.
3. Load the application .out file.
4. Select **Tools→XDS560 Trace→Control**.

   Choose the appropriate Trace Buffer Type (choose "Stop on buffer full" for all of the following examples) and click OK (see Figure 3). There will be a slight delay while the Trace Receiver is programmed and calibrated. The calibration process is shown in the status bar at the bottom of the window.

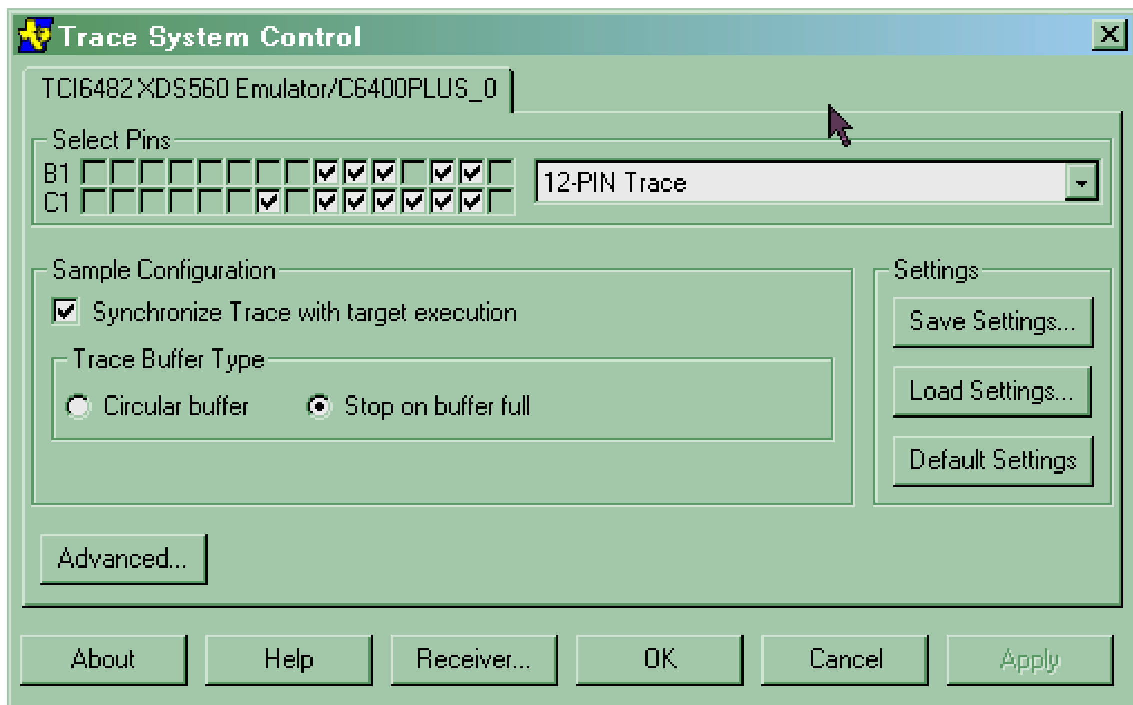


Figure 3. Trace Setup Menu

5. Open the Trace Display by selecting **Tools→Trace→Display**.

6. Ensure that the Trace Display has been started by clicking the Start button on the trace display (see Figure 4).

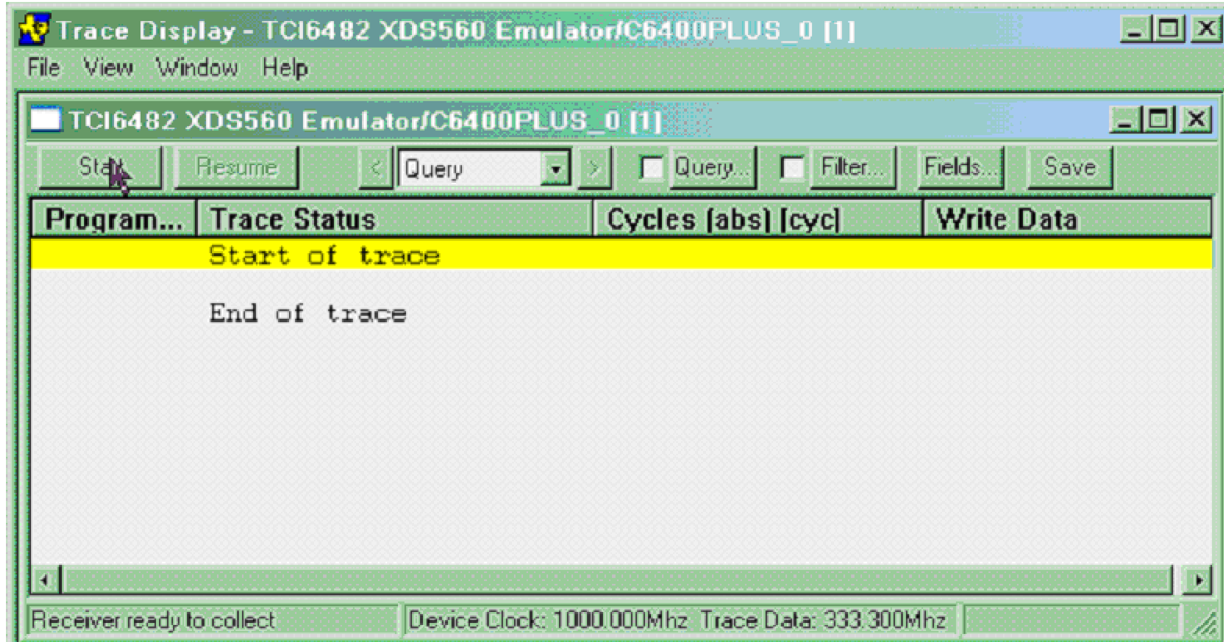


Figure 4. Starting the Trace Display Capture

6 Statistical Profiling

Statistical profiling is a means of determining the execution time of a particular function in an application relative to all other functions in the application. While the application is running, the Program Address is sampled at regular intervals (see Figure 5). Once an adequate distribution has been captured, these addresses are attached to the function that contains them. A function-level profile is then obtained by observing the number of samples attributed to each function.

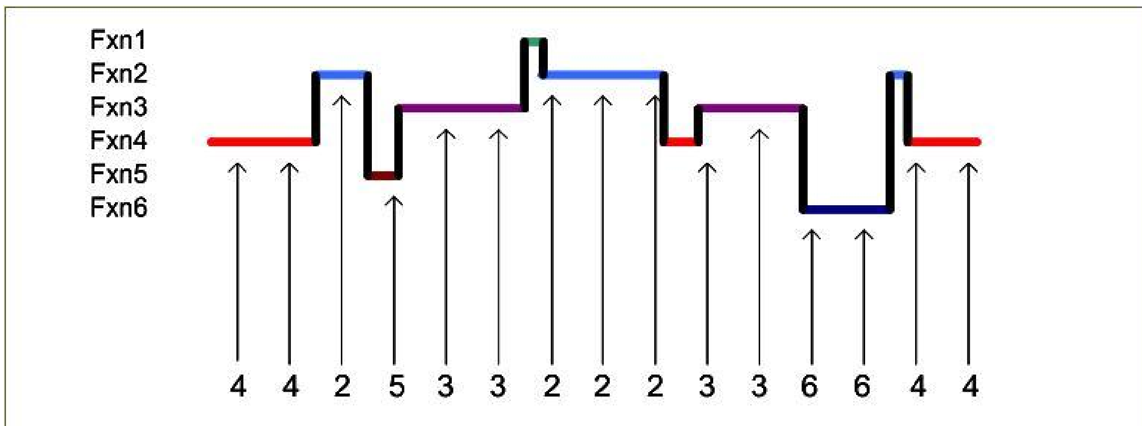


Figure 5. Function Sampling with Statistical Profiling

The benefits of performing statistical profiling over profiling on a simulator are as follows:

- **Quicker access to results.** Statistical profiling results can be generated in a matter of minutes while a simulator might take hours or days to execute a large application.
- **Inherent accuracy of hardware effects (cache, etc.) modeling.** Since we are statistical profiling on hardware, all cache/peripheral/memory considerations are inherently taken into consideration. Note however that statistical profiling itself does not produce an exact cycle count of a given function because it captures a “random” sample of the program counter. However, exact cycle count is not the purpose of statistical profiling.

The following section describes the steps for generating statistical profiling results in an application.

6.1 Target Application Configuration

To perform statistical profiling, it is necessary to use the AET Target Library discussed in Section 3. The Code Composer Studio plug-in does not implement all the features necessary for statistical profiling. In addition, we want to programmatically narrow the capture to the region of interest—only a target library can do this. This means that a small portion of application cycles are will be consumed by the AET target library and some additional memory footprint is needed. However, in comparison to large applications, both of these issues should be negligible.

Statistical Profiling data is captured using an AET Timer/Counter to cause the Trace hardware to capture PC trace data at an interval specified by the user.

There are a few simple steps to statistically profile an application:

1. Add aet_stat_profile.c to the application to be profiled.

   This file is part of the AET Target Library installation. It can be found in the statistical profiling example shipped with the library. It is a wrapper function calling AET APIs to set up statistical profiling. These calls are abstracted into three simple function calls: aetStatProfileStart(), aetStatProfileStop(), and aetStatProfileEnd().

2. Include the aet_stat_profile.h and aet.h header files in any .c files that will call the statistical profiling functions.

3. Link in the appropriate version of the AET Target Library.

4. Place the appropriate calls to the aetStatProfile functions as detailed in Table 3.

   – Place the first instance of aetStatProfileStart() in the code where statistical profiling should begin. In many cases, this might be right before a key algorithm executes. Other cases might use this call at the beginning of main to profile an entire application.

   – Use aetStatProfileStop() to eliminate profiling in uninteresting locations—for example in testbench code or idle time. Use aetStatProfileStart() to later resume statistical profiling.

   – Use aetStatProfileEnd() when the application gets to the end of where tracing is desired.

Table 3. Statistical Profiling Functions

Function Name	Parameters	Details
aetStatProfileStart()	*32-bit unsigned integer (cycleDelta)	Programs AET hardware to start/continue statistical profiling
aetStatProfileStop()	None	Disables capture of statistical profiling data until aetStatProfileStart() is encountered.
aetStatProfileEnd()	None	Ends Statistical Profiling capture and releases AET resources.

*The cycleDelta parameter specifies the number of cycles between trace capture triggers. It is ignored when aetStatProfileStart() is called after aetStatProfileStop(). If you want to change the cycle delta, you must call aetStatProfileEnd() and then call aetStatProfileStart(). Post-processing scripts have no means to account for a change in cycle delta within a single statistical profile run.

6.2 Delta Value Considerations

The advantage of using statistical profiling over a more traditional method of profiling is that the profiling run can be extended. Because we are capturing only a single sample of data for a great number of execution cycles, the XDS560 Trace buffer becomes full less quickly than it would sampling every PC.

Selecting the cycleDelta value to use when calling aetStatProfileStart() for the first time is fairly straightforward. The number specifies how many application cycles are spent between statistical profiling samples. Choosing a value that is too low defeats the purpose of statistical profiling as it consumes the entire trace buffer before it has profiled the area of interest. Conversely, using a value that is too high can be problematic in instances where certain code is executed only once or a few times. A value that is too high makes it possible that the code will execute but be missed by statistical profiling.

The ideal scenario is an application that runs in a loop or executes a great number of times. In these cases, the issues caused by selecting too high of a value for delta are alleviated because if data is missed on the first round through the code, it can be captured during subsequent executions of those functions.

Additionally, the use of aetStatProfileStart() and aetStatProfileStop() calls to bracket the specific areas of interest in the application further increases the precision of the results we obtain through the statistical profile.

In order to choose an appropriate starting value for the cycle delta, we can use a simple equation. Typically, statistical profiling generates on the order of 5000 samples before the trace buffer gets filled, assuming a Trace 1.0 buffer size of 224 KB. So, if we estimate the number of cycles it takes to execute the entire area of interest, and divide that value by 5000, we should get a decent value for the cycle delta. Again, running the application in a loop and turning profiling off in locations that are uninteresting makes the choice of the cycle delta easier. If results show that a number of functions never get executed or aren't executed as often as expected, the cycle delta should be tweaked.

6.3 Post-Processing

Once data is captured in the Trace Display, it is displayed in a raw data format as shown in Figure 6. Not much information can be obtained from the data in this format, so it is necessary to post-process the data with a script to convert it to a more useful form.

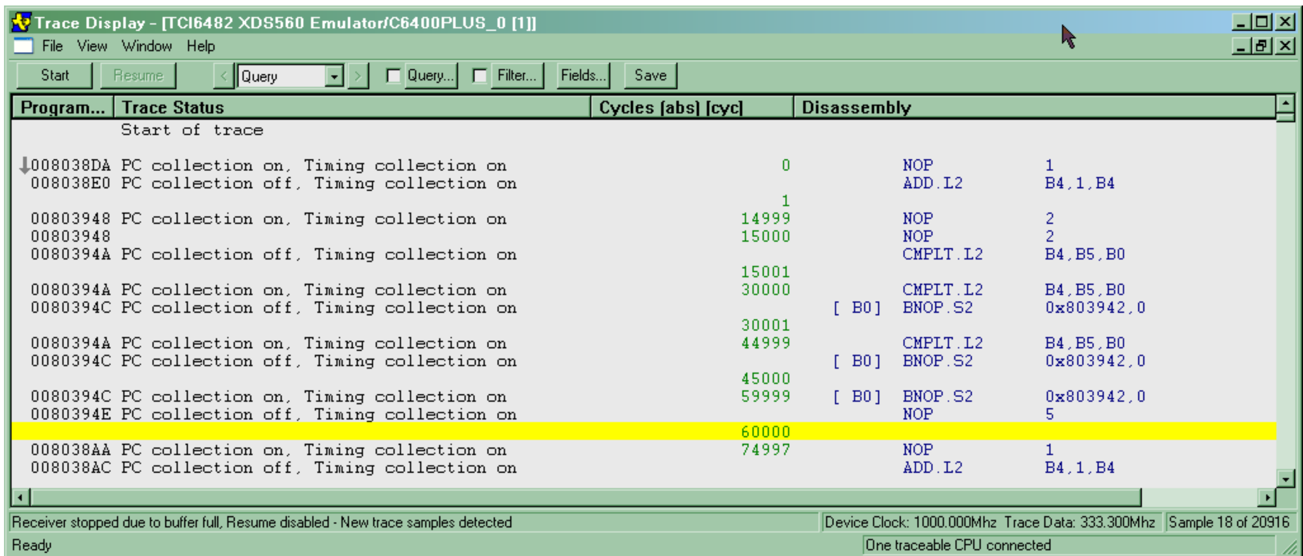


Figure 6. Sample Trace Statistical Profile

6.3.1 Saving the Data in .csv Format

To post-process the data into a more useful form, we first save the data to a comma-separated value (.csv) file. To do this, from within the trace display, choose **File→Save As**. In the Save As dialog, choose “Text Export to CSV” and click Save. Type a name for the file and give it the .csv extension. The fields currently displayed are the only fields that will be saved in the .csv format. See the documentation for the specific script to find out which fields are required. In general, saving only the required fields allows the script to process the data more quickly.

6.3.2 Post-Processing the Data with a Script

Once this file is saved, we can then post-process it with a Perl script. Our script needs 3 fields: Program Address, Trace Status, and Cycles. The Perl script used to process this data is called trace_stat_profile.pl and is included with the Trace CSV Script Release package available at https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm (login required).

Note: All scripts mentioned in this app-note were tested against ActivePerl 5.8.3. They should work fine in other environments (cygwin, etc.) but you may need to install some dependent modules as indicated in this FAQ.

Each script has documentation about the details and switches that can be used. PowerPoint information is provided in the distribution. In addition, documentation is contained within the script and can be viewed by issuing the command "`perldoc script_name.pl`". The output of the perldoc command looks similar to Figure 7, with specifics of what the script is used for, how to invoke it from the command line, and a detailed list of options.

```
C:\WINDOWS\system32\cmd.exe
NAME
    trace_stat_profile.pl

SYNOPSIS
    This script is provided to generate statistical profiling data from a
    Trace generated CSV file created by using the trigger_on_timer API of
    AET target library. It generates function execution data by sampling the
    PC at a specified interval. The script then takes that data and converts
    it into function related statistics. It has been proven that this is an
    effective method fore

LIMITATIONS
    Requires that the output of the func_info.pl script be generated with
    the --no_header option.

    The following columns MUST be included in the Trace .csv file. - Program
    Address - Cycles - Trace Status (Must be saved in textual form)

CONSIDERATIONS
    There are some issues to consider when using statistical profiling. One
    of the limitations of the trace hardware is that it cannot process a
    trace trigger while in the middle of an SPLOOP (on 64x+) and during
    delay slots of a branch. Triggers generated in when these conditions
    occur are pushed off until the condition goes away. If we are performing
    statistical profiling and one of the conditions occurrs for a duration
    of one or more sampling periods, some of the samples will be missed. The
    script can account for these samples, and accomodate for them if the
    timestamp is included in the trace output and the --delta parameter is
    provided.

    However, if AET_statProfileStart and AET_statProfileStop are used
    multiple times in order to focus on a single function and a single task,
    the --delta parameter should no longer be used as it will skew the data.
    When AET_statProfileStop is used to turn off profiling, and then

USAGE
     perl trace_stat_profile [options] --input=<csv file> --function_csv=<function csv file>
                       ... OR ...
     ofdXX -x -g <.out file> | perl func_info.pl --no_header | perl trace_function_vs_counts.pl --input

OPTIONS
     -h                                  : Dump usage and quit
     --input <input filename>            : Specify input .CSV file generated by trace
     --func_input <function csv filename> : Specify input function .csv file if not piped in
     --delta <number>                    : Specify delta cycle value used for statistical profiling
     --noheader                          : Do not print a header for the .csv data
```

Figure 7. Sample Perldoc Output

One other utility is needed to post-process the date. This is **ofd6x.exe**, which is released with each version of the Code Generation Tools. This file is located in <cgtool dir>\bin. You need a version of this tool that supports the –func_info switch. If your version does not support this switch, you can get an updated version at: https://www-a.ti.com/downloads/sds_support/applications_packages/cg_xml/cg_xml_v0_90_00/ofd_alpha_610a07115.zip (login required).

The script requires two different inputs in order to generate its output. The first input is the .csv trace data file that you created. For this example, we call the file stat_profile_data.csv. The second is a .csv file generated with the --func_info switch of the ofd6x.exe script that specifies the start and end address of every function. For this example, we call this file statprof_func_info.csv.

Execution of the script can happen in two ways. We can create intermediate files (described in Section 6.3.3), or we can pipe the output of one utility to the input of the next in a single command (described in Section 6.3.4).

6.3.3 Creating Intermediate Files

If you want to generate intermediate files (faster on Windows than piping), use the following commands to create the intermediate files and generate the appropriate output:

```
[>] ofd6x –func_info statprof.out > statprof_func_info.csv

[>] perl trace_stat_profile.pl -n –i=statprof_data.csv –f=statprof_func_info.csv
–d=nnnn > results.csv
```

The value *nnnn* should match the cycle value that was used in capturing the trace data.

These commands cause the following actions to occur:

1. Extract symbol information from the .out file into an .xml file
2. Post-process the XML information generated in step 1, and create a table that contains each function's name, filename, start address, and end address.
3. Execute the Trace Perl script, which combines the data captured in the trace data file with its function/file information and performs a statistical breakdown of function execution percentage.

One thing to note in this example is that as long as the .out file does not change, the output of the first two steps will not change. So, once you have created the func_info.csv file, you can simply execute the third instruction again and again for multiple iterations of trace data capture.

The meanings of all Perl script switches are outlined in the perldoc.

6.3.4 Processing the Entire Script in a Single Command Line

If you want to perform all of this processing in a single step, the scripts are configured to accept piped output from another script. The disadvantage to doing it this way on a Windows machine is that the pipe operator can degrade the speed of the script significantly. The advantage is that the entire script command line is executed in single command, and there's no (large) intermediate XML file.

```
[>] ofd6x –xg statprof.out | perl func_info.pl -n | perl trace_stat_profile.pl
–i=statprof_data.csv –d=nnnn > results.csv
```

6.4 Results

You can open the resulting file, results.csv, in Microsoft Excel to view the results. The results should look similar to those in Figure 8.

	A	B	C	D
1	Function name	File name	Times Encountered	Percentage
2	H264MPVDEC_TI_residual_block_cavld	h264vdec_ti_cavldmb_p.sa	845	10.95%
3	H264MPVDEC_TI_decode_nmb_P	h264d_decodenmb_p.c	405	5.25%
4	H264MPVDEC_TI_QDMA_wait	h264vdec_qdma.c	381	4.94%
5	H264MPVDEC_TI_mb_header	h264d_mbheader.c	329	4.26%
6	H264MPVDEC_TI_aso_calc_mvp	h264d_aso_mvp.c	296	3.83%
7	H264MPVDEC_TI_recon_nmb_P	h264d_reconnmb_p.c	290	3.76%
8	H264MPVDEC_TI_filterEdgeVertLuma_Str3210	h264d_edgeloop_genericLumaV_p.sa	285	3.69%
9	H264MPVDEC_TI_sparse_idct4x4_addpred_mbaff	h264vdec_ti_sparse_idct4x4_addpred_mbaff_p.sa	231	2.99%
10	H264MPVDEC_TI_cavldMB	h264vdec_ti_cavldmb_i.c	222	2.88%
11	H264MPVDEC_TI_interpHqphp_N8x8	h264vdec_ti_interphqphp_n8x8_p.sa	208	2.69%
12	H264MPVDEC_TI_filterEdgeHorzLuma_Str3210	h264d_edgeloop_genericLumaH_p.sa	201	2.60%
13	H264MPVDEC_TI_aso_read_coeff	h264vdec_ti_cavldmb_i.c	195	2.53%
14	H264MPVDEC_TI_idct4x4	h264d_idct4x4_p.sa	191	2.47%
15	H264VDEC_TI_calc_ext_edge_sths	h264d_loopfilter.c	179	2.32%
16	H264MPVDEC_TI_slice_data_I	h264d_slice.c	174	2.25%
17	H264MPVDEC_TI_slice_data_P	h264d_slice_p.c	168	2.18%
18	H264MPVDEC_TI_bbox_calc_1bb	h264d_mc.c	157	2.03%
19	H264MPVDEC_TI_deblock_nmb_frm	h264d_loopfilter.c	151	1.96%
20	H264MPVDEC_TI_interpVqpHhp_N8x8	h264vdec_ti_interpvqphhp_n8x8_p.sa	149	1.93%
21	H264MPVDEC_TI_yuvtobt835_c	yuvtobt835_i.c	139	1.80%
22	H264MPVDEC_TI_filterEdgeVertChroma_Str3210	h264d_edgeloop_genericChromaV_p.sa	138	1.79%
23	H264MPVDEC_TI_interpChroma_N4x4	h264vdec_ti_interpchroma_n4x4_p.sa	138	1.79%
24	H264MPVDEC_TI_residual_block_cavld_Chroma	h264vdec_ti_cavldmb_chroma_p.sa	132	1.71%

Figure 8. Sample Statistical Profiling Data

7 Pipeline Stall Analysis

Embedded DSPs can perform calculations at high speed for a number of reasons. Some of the most significant reasons are fast internal memory, the instruction pipeline, and sophisticated memory caches. The instruction pipeline allows up to 8 instructions to be executed in a single cycle, while memory caches can allow external memory to be accessed at speeds comparable to internal memory.

Instruction pipeline stalls occur when the CPU has to wait for data and are essentially wasted cycles. Identifying where pipeline stalls are occurring and taking action on those that have the greatest effect on an application is a good way to optimize an application.

7.1 Target Application Configuration

Unlike statistical profiling, pipeline stall analysis can be done on an unmodified application. There is no need to add the AET Target Library.

The pipeline stall analysis uses a full PC and Timing Trace. Essentially Trace is started, and every sample is captured along with a timestamp.

The simplest way to perform pipeline stall analysis is to create a “Start Trace Job”. To do this, select **Tools→Advanced Event Triggering→Event Analysis** in CCStudio. You will see the plug-in shown in Figure 9.

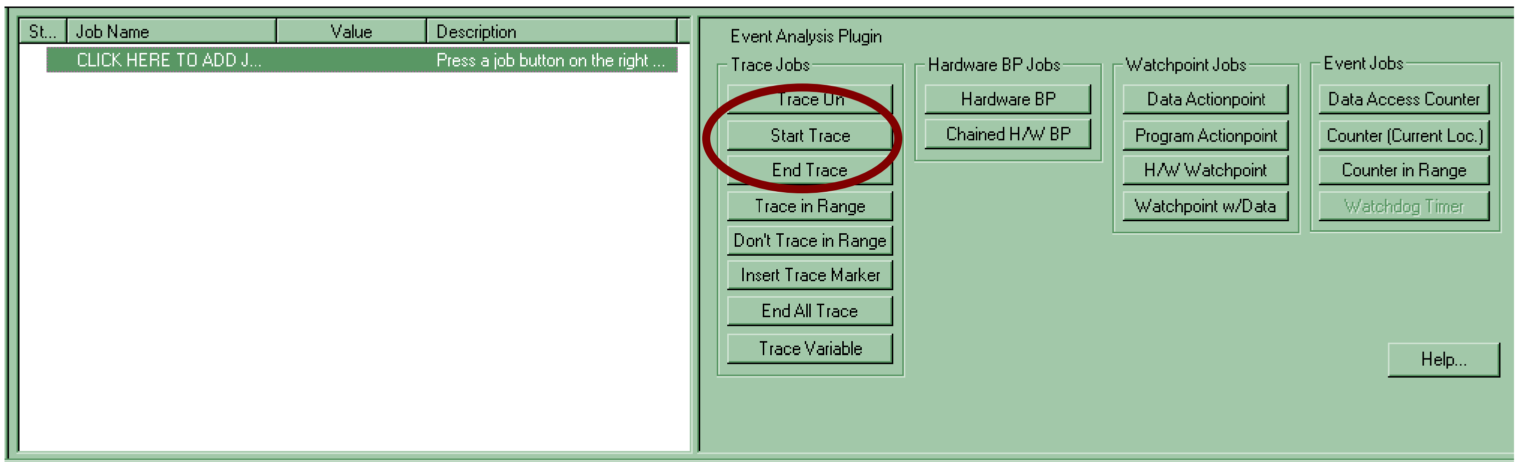


Figure 9. Event Analysis Plug-in

To capture the data that is most useful to us, we want to delay the capture of trace until we encounter an interesting location in the application. Finding pipeline stalls in boot code or other initialization code isn’t very useful, since it executes only once. So, the appropriate job to choose in this case is a “Start Trace” job, which starts trace capture at a specified program addresses. In some cases this could be the beginning of main. In most cases, it will be the beginning of processing an algorithm.

To create the job, click the Start Trace button in the AET plug-in. Set the Start Address at the location of your choice. You can enter the hex address directly, or select a line from a .c file and drag it into the text box. Choose Program Address and Time Stamp to trace and click Apply. See Figure 10 for an example.

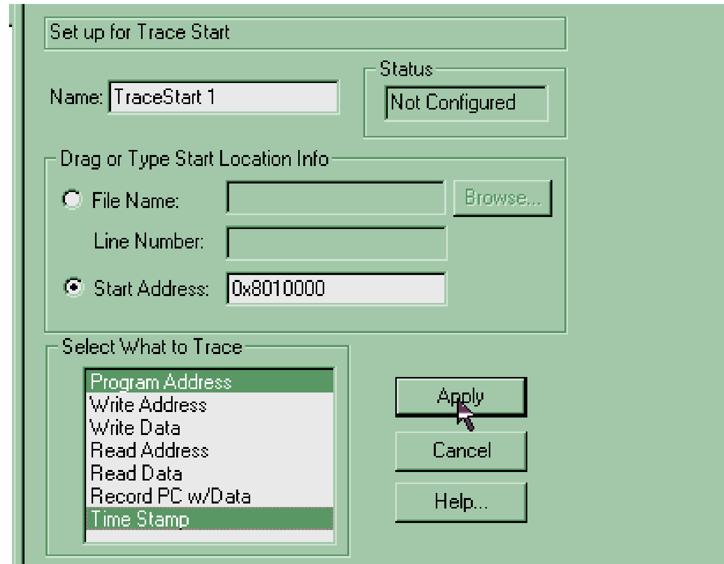


Figure 10. Programming a Start Trace Job

Once trace has been programmed, run the application. When the trace buffer is completely filled, the Trace Display decodes all the information and displays it. The resulting data should resemble that in Figure 11.

Trace Display - [C64xx XDS560 Emulator/CPU_1[1]]

File View Window Help

Start Resume Query Query... Filter... Fields... Save

Program Address	Cycles [dt] [cyc]	Trace Status	Disassembly
		Start of trace	
00014780	+1	PC collection on, Ti...	STW.D2T2 ...
00014784	+1		NOP 2
00014784	+9	Pipeline stall	NOP 2
00014784	+1		NOP 2
00014788	+1		B.S1 ...
0001478C	+1		MVK.S2 ...
00014790	+1		MVKH.S2 ...
00014794	+1		OR.L2 ...
00014798	+1		OR.L1X ...
0001479C	+1		ADDKPC.S2 ...
000147A0			|| OR.L1X ...
000146C0	+1		SUB.D2 ...
000146C4	+1		STW.D2T2 ...
000146C8	+1		STW.D2T1 ...
000146CC	+1		STW.D2T2 ...
000146D0	+1		STW.D2T1 ...
000146D4	+1		NOP 2
000146D4	+1		NOP 2
000146D8	+1		OR.L2X ...
000146DC	+1		STW.D2T2 ...

Figure 11. Sample Pipeline Stall Analysis Data

7.2 Post-Processing

Again, the data captured in the Trace Display is not that useful in its raw format, so we need to post-process it to efficiently analyze it. We must first save it as a comma-separated value (.csv) file. See Section 6.3.1 for instructions on how to save the data as a comma-separated value file.

The script used to process this data is trace_pipeline_stall_analysis.pl, which is included with the Trace CSV Script Release package that can be downloaded from https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm (login required). Each script has documentation within the script that can be viewed by issuing the command `perldoc 'script_name.pl'`.

The trace_pipeline_stall_analysis.pl script also requires the ofd6x.exe utility. See Section 6.3.2 for additional information on where to get the latest version.

This script needs two different inputs in order to generate its output. The first input is the .csv trace data file you created. For this example, we call this file pipe_stall_analysis_data.csv.

You can execute the script can happen in two of ways: using intermediate files or by piping the output of one utility to the input of the next. For more information see Section 6.3. The sample command lines for both methods are shown below.

Assuming that the name of the file gathered from trace is pipestall_data.csv and the name of the .out file is pipestall.out, the following commands create the intermediate files and generate the appropriate output. The value *nnnn* is the lower threshold for average pipeline stalls. That is, –p=10 shows only instructions that have an average stall cycle value greater than or equal to 10.

Intermediate File Method

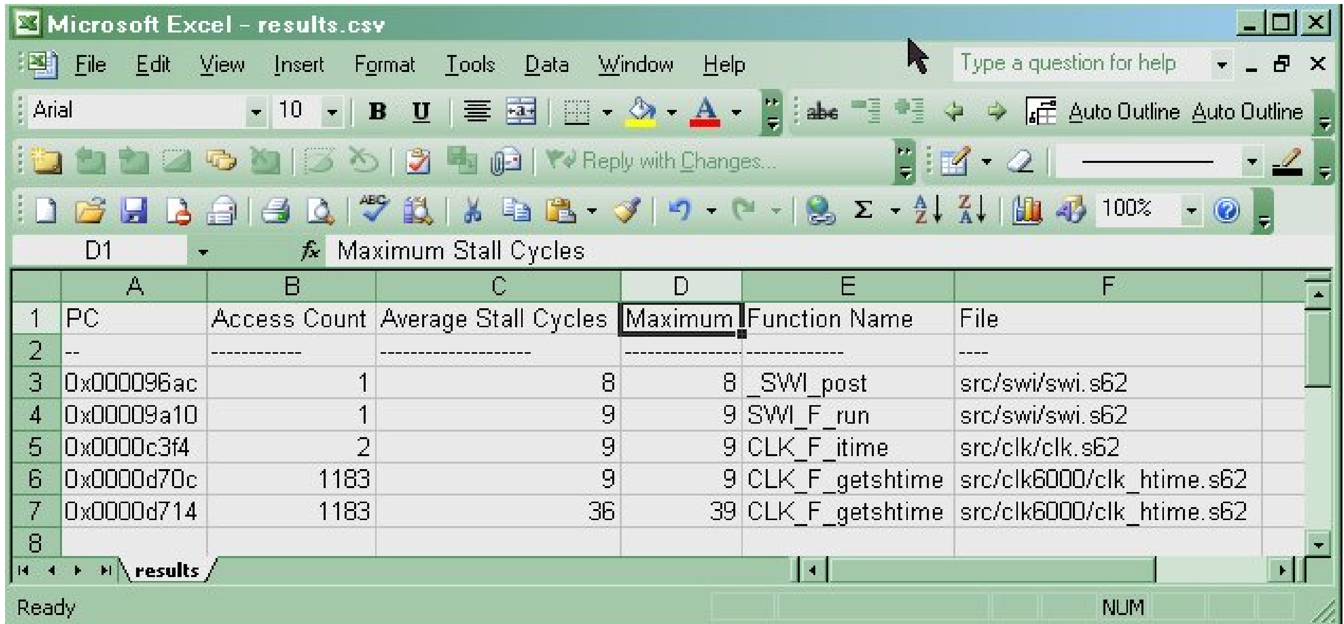
```
ofd6x -func_info pipestall.out > pipestall_func_info.csv

perl  trace_pipeline_stall_analysis.pl -n  -f=pipestall_func_info.csv
-t=pipestall_data.csv -p=nnnn > results.csv
```

Piped Command Line Method

```
ofd6x --func_info  pipestall.out | perl trace_stat_profile.pl -t=pipestall_data.csv
-p=nnnn > results.csv
```

Once the data is gathered, you can open the generated file in Microsoft Excel. See Figure 12 for an example of the generated output.



PC	Access Count	Average Stall Cycles	Maximum	Function Name	File
--	------------	--------------------	--------------	--------------	----
0x000096ac	1	8	8	_SWI_post	src/swi/swi.s62
0x00009a10	1	9	9	SWI_F_run	src/swi/swi.s62
0x0000c3f4	2	9	9	CLK_F_itime	src/clk/clk.s62
0x0000d70c	1183	9	9	CLK_F_getshtime	src/clk6000/clk_htime.s62
0x0000d714	1183	36	39	CLK_F_getshtime	src/clk6000/clk_htime.s62

Figure 12. Sample Pipeline Stall Output (‘C64x)

Table 4 shows typical cycle penalties for L1D and L2 cache misses on the ‘C6416 and ‘C6455 DSKs. These values can affect the threshold value used for capturing pipeline stall information. For example, if we are only interested on L2 misses on a ‘C6455, we could set our threshold value to 20.

Table 4. Typical Stall Values for Cache Misses

‘C6416 DSK	‘C6455 DSK
L1D miss ~ 6 cycles	L1D miss ~ 12 cycles
L2 miss ~ 50-400 cycles	L2 miss ~ 20-200 cycles

Once the locations and causes of pipeline stalls are found, you can then tune the application to remedy some of the situations. In many cases, this might involve rearranging the location of some code (for example, moving some different algorithms into internal memory) or additional considerations in order to minimize the stalls.

In the Figure 12 example, we encounter another interesting issue. We see a 36 cycle stall average in the CLK_F_getshtime function. This is too large of a stall to be a L1D miss, but too small to be an L2 miss. By studying the instruction carefully, we find that this instruction reads the on-chip timer. Because of the way the on-chip timer needs to access the peripheral bus on the ‘C64x devices, this is unfortunately a stall that cannot be avoided. However, this script will allow you to analyze which stalls you have control over, and which ones you don’t so that optimization efforts are focused on areas where it is feasible.

8 Conclusion

This document has described a number of ways to use XDS560 Trace and Advanced Event Triggering (AET) on ‘C64x and ‘C64x+ devices. It provided procedures for gathering and processing data concerning both statistical profiling and pipeline stalls. Using these procedures, you can optimize your applications.

9 References

- AET Target Library: https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm (login required)
- Trace CSV Script Release package available at https://www-a.ti.com/downloads/sds_support/applications_packages/index.htm (login required) includes PowerPoint and Perldoc documentation.
- FAQ on Running Perl XML Scripts Under UNIX/Linux
- OFD Utility: https://www-a.ti.com/downloads/sds_support/applications_packages/cg_xml/cg_xml_v0_90_00/ofd_alpha_610a07115.zip (login required).

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Communications and Telecom	www.ti.com/communications
DSP	dsp.ti.com	Computers and Peripherals	www.ti.com/computers
Clocks and Timers	www.ti.com/clocks	Consumer Electronics	www.ti.com/consumer-apps
Interface	interface.ti.com	Energy	www.ti.com/energy
Logic	logic.ti.com	Industrial	www.ti.com/industrial
Power Mgmt	power.ti.com	Medical	www.ti.com/medical
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Space, Avionics & Defense	www.ti.com/space-avionics-defense
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video and Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless-apps

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2010, Texas Instruments Incorporated