Debugging TMS320C55x Applications Using Emulation Analysis

Matthias Kassner — *European Sales and Applications*

ABSTRACT

With the increase in complexity in DSP applications, debugging has become a more complex and a more important part of the development process. The need to debug complex applications while running in real-time implies the usage of dedicated hardware.

TI has recognized and actively supported this trend and provides developers with a completely new toolset, the Emulation Analysis Module, which based on dedicated hardware within the TMS320C55x™ family of DSPs. This toolset gives the developer the possibility to use both hardware breakpoints and watchpoints to detect the occurrence of certain hardware events.

This application report includes an overview of the hardware architecture, describes the capabilities available and how they can be used, and illustrates the use of these features in a simple example.

This application report was written for Code Composer Studio version 2.12 and all present C55x DSP core revisions.

Contents

List of Figures

List of Tables

1 Introduction

The emulation analysis module is a dedicated block of hardware integrated into every C55x DSP. This module is controlled by several registers that are located in the DSP peripheral I/O space. Code Composer Studio provides a graphical interface to easily and correctly configure these registers.

1.1 Hardware Architecture – System Level

In order to better understand the emulation analysis features, we will first have a short look at the way it is implemented in a C55x DSP. The emulation analysis module is tightly integrated in the DSP core and communicates with the other units there via a wrapper.

The wrapper essentially handles the communication between the different functional blocks within the DSP core. These blocks are:

- The functional units in the DSP core that execute all DSP operations; for instance multiply-accumulate, load, store, and program control operations.

- Execution control that arbitrates between interrupt, instruction execution, and debug requests for specific hardware resources
- JTAG control that controls the emulation Interface

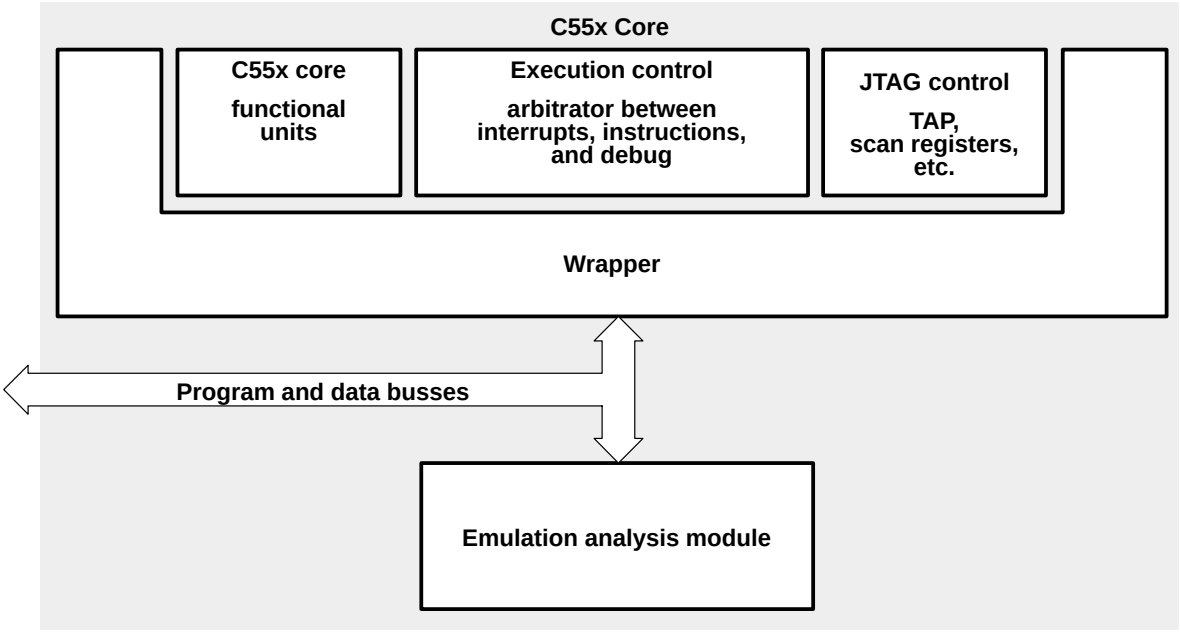

Figure 1. Conceptual DSP System Diagram

1.2 Hardware Architecture – CPU Bus System

In order to make efficient use of the emulation analysis module, some knowledge about the underlying CPU bus structure is required. This will enable you as user to select the correct CPU bus to be monitored.

The C55x CPUs have the following internal busses:

- Data read address busses – B address bus (BAB), C address bus (CAB), and D address bus (DAB)
- Data read data busses – B bus (BB), C bus (CB), and D bus (DB)
- Data write address busses – E address bus (EAB) and F address bus (FAB)
- Data write data busses – E bus (EB) and F bus (FB)
- Program address bus – P bus (PAB). The program data bus (PB) is not used by the emulation analysis module.

Table 1. Access Type versus Bus Usage

	Data Read Busses						Data Write Busses				Program Address Bus
	Address			Data			Address		Data		
Access and Bus Type	**BAB**	**CAB**	**DAB**	**BB**	**CB**	**DB**	**EAB**	**FAB**	**EB**	**FB**	**PAB**
Single Read			X			X					
Dual Read		X	X		X	X					
Double Read			X		X	X					
Triple Read	X	X	X	X	X	X					
Single Write							X		X		
Dual Write							X	X	X	X	
Double Write							X		X	X	
Program Read											X

It is worth noticing a limitation of the emulation analysis module, which is a direct consequence of the CPU bus usage as shown above. This limitation concerns the case of a double access (read access from or write access to two consecutive locations in memory). Here, only one address is generated internally; the access to the second word is done by toggling the LSB of the data address using some external logic.

This enables a more efficient internal bus design, however, this means also that the data access to the second word within a double access cannot be detected by watching the internal data address busses.

1.3 Hardware Architecture – Emulation Analysis Module

Internally, the emulation analysis module is organized in four different functional units, which are controlled by a set of configuration registers. These registers can be accessed either from the DSP itself using a DSP I/O space access with the `port()` qualifier or — more typically — from Code Composer Studio using the JTAG interface of C55x DSPs.

Input to the module are the various CPU busses as outlined in the previous chapter. Their contents are continuously compared against the user-specified values by the means of hardware comparators. In case of a match, a trigger signal to the CPU is generated, which will lead either to program execution stop or the generation of an operation system (RTOS) interrupt.

This — somewhat simplified — architecture description is illustrated in Figure 2.

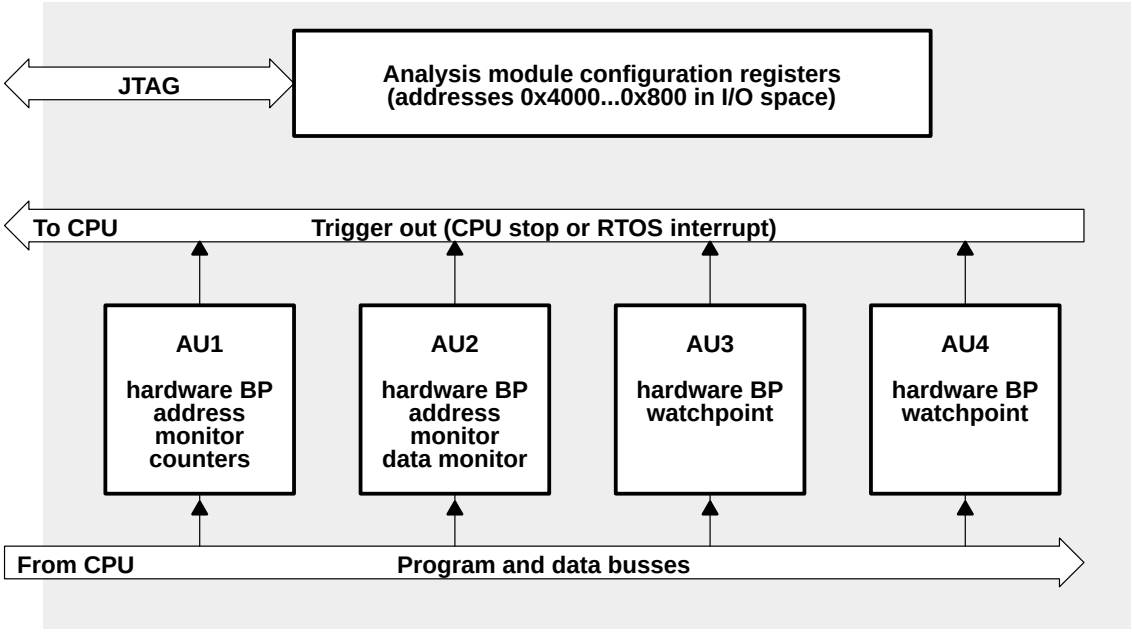

Figure 2. Emulation Analysis Module – Simplified Hardware Structure

1.4 Capabilities of the Emulation Analysis Module

Following is an outline of the capabilities of the emulation analysis module.

First of all, this module provides four *hardware breakpoints*. A hardware breakpoint triggers a debug event (execution stop or RTOS interrupt) when program execution reaches a specified address. Hardware breakpoints work only with *program* addresses and should not be confused with any sort of *data* access to memory.

Breakpoints are required not only for the use as breakpoints in the normal debug environment, they also enable single stepping, the run-to-cursor function, and the automatic execution stop at the end of a C program.

The use of hardware breakpoints (as opposed to software breakpoints) is required only if you are working with read-only memory. In all other cases, you can use software breakpoints, which are basically instructions — `estop_1()` — that are introduced into your code. Each time the CPU encounters such an instruction, it will stop execution. Unlike their hardware counterparts, the number of software breakpoints is practically unlimited.

The emulation analysis module also provides several bus address and bus data monitors. You can use these to watch for specific values on the data busses and specific addresses at the data address busses. Address and data monitors can be combined into *watchpoints*. A watchpoint will be triggered upon the match of a specified data address and/or data value with a predefined mask.

To use this capability, you need to know if the access will be a read or a write access, the access type (short or long), and in the case of a read access, if it will be on the B bus or not. The latter is usually easy to decide, as B bus accesses occur only in the case of the constant coefficient in Dual-MAC types of operations.

Note that it is not possible to set a watchpoint on a memory-mapped register (MMR) access. These transactions are not visible on the main address and data busses.

Finally, a 32-bit *counter* is provided, which can be split into two 16-bit counters. These counters can be used to count the occurrence of all sorts of events; for instance execution cycles, interrupts, cache misses, and others. A trigger threshold can be specified, upon which a trigger signal to the CPU will be generated.

Also notice that the use of emulation analysis capabilities requires the hardware presence of a C55x CPU; these features presently do not work with the simulator.

2 Using Emulation Analysis

After looking into the basic architecture of the emulation analysis module, we will now describe how to use its various capabilities in conjunction with Code Composer Studio. Code Composer Studio provides access to the various configuration registers by means of a graphical interface.

This facilitates the use of emulation analysis to a great extent as compared to programming all these registers yourself. This plug-in is the first step towards a more comprehensive debugging toolset and will be continuously improved.

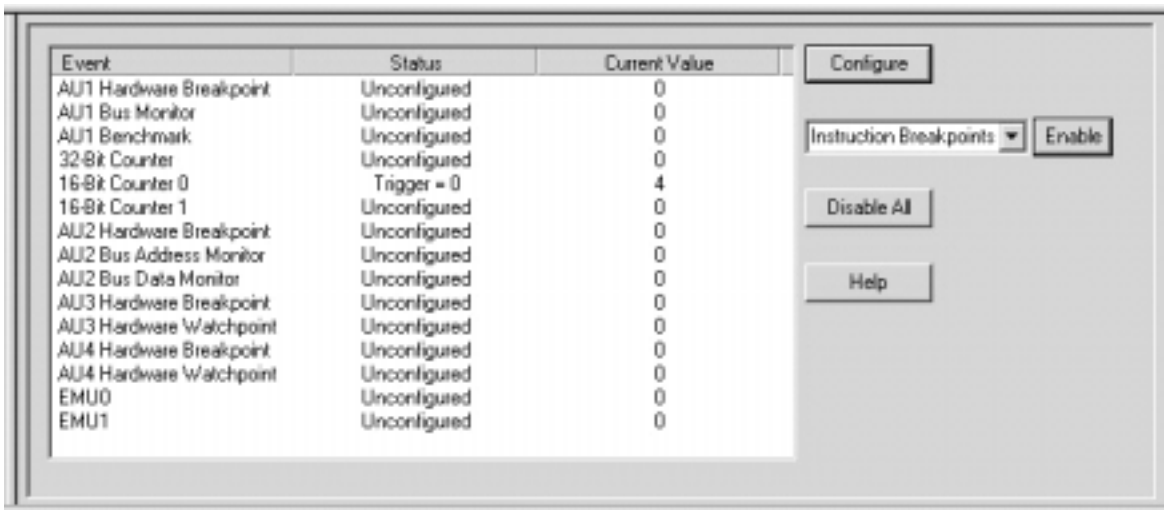

Figure 3. Emulation Analysis Plug-In interface

Notice the partitioning of the window — in the first column you find the available hardware resources, in the second one their present state (unconfigured, configured, or triggered), and in the third one their current value.

You can configure each of the available hardware resources by double-clicking on the respective item in the first column. This will invoke a dialog to configure it.

The general process of using this module is such that first you load your program, then you configure the emulation analysis module, and finally run the program. A trigger signal (CPU stop or RTOS interrupt) will be generated if a breakpoint/watchpoint/counter value match occurs. If you are using the counters, the number of occurrences of a certain event will be displayed in the third column.

2.1 Hardware Breakpoints

As said before, hardware breakpoints monitor program execution and generate a debug event upon a match with a predefined value. The emulation analysis module provides four hardware breakpoints, which can be used to set breakpoints in non-writable memory.

2.1.1 Using Hardware Breakpoints

For using hardware breakpoints, start Code Composer Studio, load your program and open the emulator analysis plug-in as described above. We will use as an example the hardware breakpoint in AU1; so double click on it in the plug-in window. The dialog shown in Figure 4 will open.

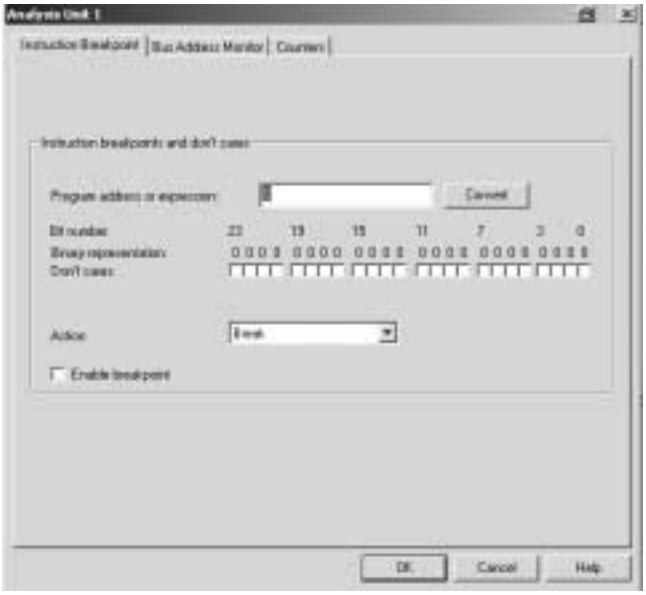

Figure 4. Hardware Breakpoint Dialog

In the Program address or expression field you have to enter the program address or the expression where you want the hardware breakpoint to be set. This can be an absolute address, any C expression, the name of a C function, or an assembly language label.

Make sure you press Enter or click Convert after specifying the address. You should then see the binary representation of your address above the “Don’t cares” checkmarks. If you want to specify a hex address, be sure to prefix the address number with 0x; otherwise, the debugger treats the number as a decimal address. You have the option to specify don’t cares under the binary address representation.

Using don’t cares, you can specify an address range rather than just a single address. If your address for instance is 0x1000 and you specify the last 4 bits as don’t cares, then any address access within the range of 0x1000…0x100F will trigger this breakpoint. Even though it enables only certain address ranges, this feature can be helpful in detecting an access to a memory region.

Finally, don’t forget to check the Enable breakpoint field to enable the breakpoint. Then click OK to accept your changes and to dismiss the dialog box. You can now run your program. Program execution will stop at the specified program address.

2.2 Watchpoints

Watchpoints are triggered on a combination of a certain address and a certain value used in memory accesses. This enables you to trigger, for instance, on the occurrence of a memory access writing the value 2 to variable y. You do not have to specify both an address and a certain value. You can also trigger using either an address or a certain data value only.

Note, that the caveat described in section 1.2 applies here as well — you cannot trigger on either a data address or a data value that is accessed as the second word of a double access.

2.2.1 Using Watchpoints

In order to define a watchpoint, you can either use the address monitor of AU1 in conjunction with the data monitor of AU2, or — somewhat easier — select the dedicated hardware watchpoints in AU3 or AU4.

As an example, we will use the watchpoint provided by AU4, so double-click on the caption which reads AU4 Hardware Watchpoint. The configuration dialog shown below will appear.

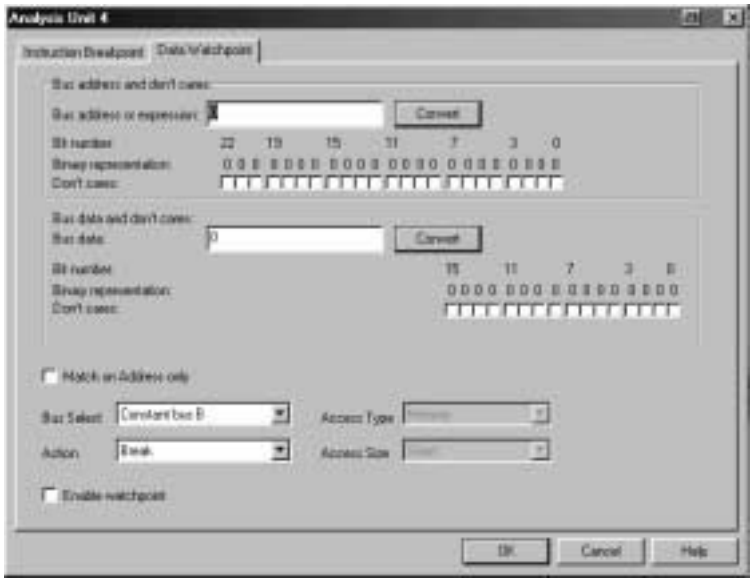

Figure 5. Watchpoint Configuration Dialog

Notice that you have two input fields here — one for the address and one for the data value. Check the Match on Address only checkmark if you want to use the address only as watchpoint criterion. If you want to use only the data value as trigger, then check all don't cares under the binary address representation.

After you have entered your address and/or data value, you need to specify the bus and the data size you want to monitor. The next chapter gives you a summary about the different options. Finally, enable the watchpoint and run your program.

2.2.2 Configuring Watchpoints

As outlined in the previous section, it is important to know the correct settings for bus and access size to get watchpoints triggered correctly. Table 2 summarizes these watchpoints.

Table 2. Configuring Watchpoints

Access Type	Example	Configure as…
Single or dual-read, with or without data	Trigger on read-access to address of x (optional: only if the value of x is 5)	Read on C or D
Double-read with or without data	Trigger on double-read to x and x+1 (optional: only if the value is 0x12345678)	Single read on C Access type: long
Read on B-bus	Trigger on read to the constant coefficient in a dual-MAC operation	Constant bus B
I/O port read	Trigger on read-access to Timer1 count register	Read on D Access type: I/O
I/O port write	Trigger on write-access to McBSP1 TX register	Write on E Access type: I/O
Single or dual-write, with or without data	Trigger on write-access to address of y (optional: only if the value of y is 5)	Write on E or F
Double-write with or without data	Trigger on double-write to y and y+1 (optional: only if the value is 0x12345678)	Single write on E Access type: long

An important thing to keep in mind when setting a watchpoint on a double-access is the endianess of the DSP. All C55x DSPs are big endian. This means that when you want to set a watchpoint on a double-access to address 0x1000 and the data at address 0x1000 is 0x1111 and the data at address 0x1001 is 0x2222, then the resulting data word in 32-bit format is 0x11112222, not the way around.

2.2.3 Chaining Watchpoints

The emulation analysis module provides the option to chain the two watchpoints provided by AU3 and AU4 together. Chaining means that a trigger signal will be generated only after both watchpoints have been triggered. You have to setup the watchpoints such that the watchpoint in AU3 will be triggered first, then the one in AU4.

For making use of this feature, you must first configure the watchpoints in AU3 and AU4. Then check the Match on AU3 and AU4 and the Sequential checkmarks in the configuration dialog for the AU3 watchpoint. The procedure of chaining watchpoints is outlined in detail in Appendix A3.5.

2.2.4 Watchpoint Trigger Latencies

In order to correctly understand the process of triggering a watchpoint, we need to discuss another piece of C55x hardware — the pipeline. You are probably familiar with the basic idea of having a pipeline to make use of all functional units at the same time.

The C55x DSP family employs a seven-stage pipeline as shown in Figure 6.

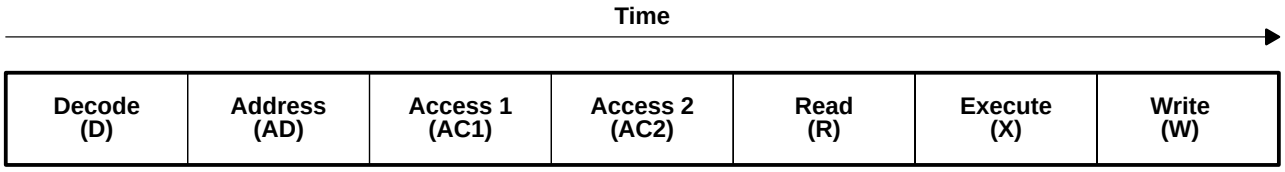

Figure 6. C55x Execution Pipeline

When working with C55x hardware, the execution arrow in Code Composer Studio corresponds to the instruction that is just being decoded. This differs from the simulator, where this arrow corresponds to the execution (X) phase of the pipeline.

What does this mean for using watchpoints? It means that you will see latency between where the program execution arrow is located after stopping due to a watchpoint, and where the instruction is located that caused this watchpoint to be triggered. It is important to be aware of this latency in order to correctly identify the instruction that has triggered the watchpoint.

The latencies versus the different access types are outlined in Table 3. Note that this table represents average values only. Some factors that can affect these latencies are:

- Accesses to slow external memory
- Pipeline protection unit actions
- Bus resource contention (e.g., between CPU and DMA)

Table 3. Latencies versus Watchpoint Type

Access Type	Watchpoint on…	Latency
Read access	Address only	5 cycles
Read access	Address and data	8 cycles
Write access	Address only	8 cycles
Write access	Address and data	10 cycles

If you have problems identifying the instruction that caused the access (for instance due to branches occurring within the latency period) you can use the C55x trace module to trace back the respective amount of instructions.

2.2.5 Watchpoint Limitations

The present implementation is the first step on the roadmap to extensive debugging capabilities within the C55x family of DSPs. It still has some limitations as can be seen from the previous sections.

First, you need to know if the access to a memory location is a single (short) or a double (long) access. Second, if it is a read-access you need to know if it will occur on the B-bus or not. Third, you cannot trigger on the address of the second word in a double access, as this address is generated by external logic by just toggling the LSB of the address.

Finally, you cannot trigger a watchpoint on an access to the memory mapped register region (word addresses 0x0…0x60) as this region is accessed using special address generation logic.

2.3 Counters

The analysis module has internal counters that count bus events and detect other internal events. The counters keep track of how many times an event occurs. The counter configuration page of the Analysis Unit 1 dialog box allows you to configure each counter to count one out of several types of events.

You can use either the 32-bit counter or split it into two 16-bit counters. You can count a single event on any of the counters, or you can count two events by setting up one on each of the two 16-bit counters.

Events that can be counted include CPU clock cycles or instructions, interrupts taken, cache misses, idle clock cycles and pipeline protection, and instruction fetch or data fetch wait cycles.

2.3.1 Using Counters

Click on any of the counters in the first column of the emulation analysis plug-in. You will see the configuration dialog as shown below in Figure 7.

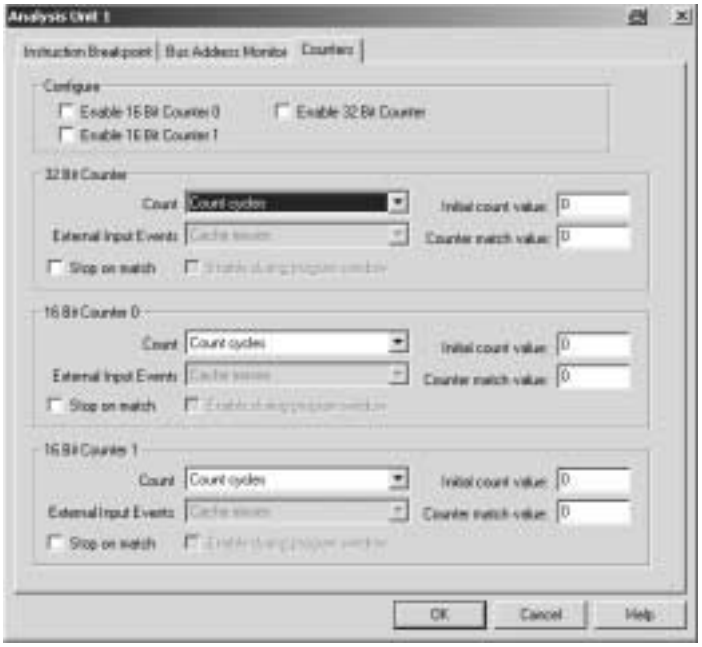

Figure 7. Counter Configuration Dialog

You must first select if you want to use the 32-bit counter or the 16-bit counter(s) by enabling it. After that, select what you want to count, in the Count field of the respective counter. The standard setting is cycles. If you want to count other events, scroll down in the Count list and select Count external input 0 for Counter 0 or Count external input 1 in the case of Counter 1. Now you can select which event you want to count, in the External input events field.

After that, you can run you program and check for the results. You can also specify a counter match value upon which program execution will stop. In order to reset the counters, open the configuration dialog, set the initial counter value there to zero and click OK. Updates do not take effect until one cycle (single-step) later.

3 Getting Started with Emulation Analysis

Appendix A includes a very simple test program, together with step-by-step instructions, that you can use to familiarize yourself with the capabilities of the emulation analysis module.

3.1 Demonstration Program

A complete source code listing is provided that can be used to test various capabilities of the emulation analysis module. Specifically, you can test the following:

- Hardware breakpoints on labels
- Hardware breakpoints on absolute addresses
- Watchpoints on:
  - Single read and write with and without data
  - Dual read and write and without data
  - Long read and write and without data
  - I/O read and write and without data
- Counters counting:
  - Cycles
  - Pipeline stalls

Detailed description about how to test these capabilities is provided as well.

3.2 System Requirements

The use of the emulation analysis module requires Code Composer Studio version 2.1 or higher connected to a C55x DSP hardware target.

4 Conclusion

The emulation analysis module is a useful tool in various different debug situations. It enables the user to track the occurrence of both data and address patterns on certain busses. Simple examples have been presented to demonstrate this functionality.

Texas Instruments is working to continuously improve this tool and to extend its functionality, to make it both an easier and a more powerful tool to use.

Appendix A Test Code and Procedure

A.1 Source Code of Test.asm

```
      .def start
output .usect ”output”, 6, 1, 1
      .sect “input”
            .even
input   .int 1, 2, 3, 4, 5, 6
            .sect “code”
start:
            ; Single Access Read and Write
      XAR0 = #input
            XAR1 = #output
            *AR1 = *AR0       ;0
            NOP               ;1 – The NOP’s are introduced to count latency
cycles
            NOP               ;2
            NOP               ;3
            NOP               ;4
            NOP               ;5
            NOP               ;6
            NOP               ;7
            NOP               ;8
            NOP               ;9
            NOP               ;10
            ; Double Access Read and Write, label to test Hardware Breakpoints
label:      XAR0 = #(input+2)
            XAR1 = #(output+2)
   dbl(*AR1) = dbl(*AR0);0
      NOP               ;1
            NOP               ;2
            NOP               ;3
            NOP               ;4
            NOP               ;5
            NOP               ;6
            NOP               ;7
            NOP               ;8
            NOP               ;9
            NOP               ;10
            ; Dual Read Access
            XAR0 = #(input+1)
            XAR1 = #(input+5)
              T0 = *AR0
            ||T1 = *AR1       ;0
            NOP               ;1
            NOP               ;2
            NOP               ;3
            NOP               ;4
            NOP               ;5
            NOP               ;6
            NOP               ;7
            NOP               ;8
            NOP               ;9
```

```
              ; Dual Write Access
              XAR0 = #(output+1)
              XAR1 = #(output+5)
              *AR0 = T0
        ||*AR1 = T1       ;0
              NOP               ;1
              NOP               ;2
              NOP               ;3
              NOP               ;4
        NOP               ;5
              NOP               ;6
              NOP               ;7
              NOP               ;8
              NOP               ;9
              NOP               ;10

; I/O Write Access to DMA Channel 0 Configuration Register
              *port(#0xC00) = #0  ;0
              NOP               ;1
              NOP               ;2
              NOP               ;3
              NOP               ;4
              NOP               ;5
              NOP               ;6
              NOP               ;7
              NOP               ;8
              ; I/O Read Access to DMA Channel 0 Configuration Register
              AR0 = *port(#0xC00) ;0
              NOP               ;1
              NOP               ;2
              NOP               ;3
              NOP               ;4
        NOP               ;5
              NOP               ;6
              NOP               ;7
              NOP               ;8
              estop_1()         ;Software Breakpoint – program execution will
stop here
here:  goto here
```

A.2 Source Code of Test.cmd

```
MEMORY
{
  SARAM: org = 0010000h,  len = 08000h
  DARAM: org = 0002000h,  len = 01000h
}

SECTIONS
{
  output  :> DARAM
  input   :> SARAM
  code    :> SARAM
}
```

A.3 Test Procedure

In order to use the code provided above, you can follow the procedure outlined now. You are of course free to use this on your own account.

A.3.1 Setup of Code Composer Studio and the Test Project

1. Configure Code Composer Studio according to your hardware using Code Composer Studio Setup. Open Code Composer Studio.
2. Select Project → New and select a project name. Remember the location of your project directory.
3. Use File → New → Source File to open a new source file. Copy and paste the content of Test.asm from the listing above into your new source file. Select File → Save as 0 and save it in the project directory you created in step 2 under the name Test.asm. You need to change the file type first to “Assembly Source Files *.asm”.
4. Select Project → Add Files to Project and browse for Test.asm that you have saved. Click OK. Test.asm will now appear in the Project Manager tree view.
5. Use File → New → Source File to open a new source file. Copy and paste the content of Test.cmd from the listing above into your new source file. Select File → Save as 0 and save it in the project directory you created in step 2 under the name Test.cmd. You need to change the file type first to “TI Command Language Files *.cmd”.
6. Select Project → Add Files to Project and browse for Test.cmd that you have saved. Click OK. Test.cmd will now also appear in the Project Manager tree view.
7. Go to Project → Build Options0 and select the Compiler tab. Go to Category Advanced. Check the Algebraic Assembly checkbox. Type 5510:2 into the Processor Version field.
8. Select the Linker tab and then Category Basic. Choose Absolute executable as the Output Module and type Test.out in the Output Filename field . Type Start in the Code Entry Point field . Click OK to close the Build Option dialog.
9. Build the project and load the program. There should not be errors. Open the emulation analysis plug-in using Tools → C55x Emulator Analysis. Resize the window so that you see the full content of this plug-in.

A.3.2 Testing Hardware Breakpoints

10. Double-click on AU1 Hardware Breakpoint. Type Label into the Program address or expression field. Check the Enable Breakpoint field. Click OK to close the window. Run the program — it should stop at label.
11. Change your display to mixed mode by selecting View → Mixed Mode/ASM. You see now in your assembly source file Test.asm, the generated assembly code interlisted with the original source code. In the first column you find the program memory addresses.
12. Select one of these program addresses, open the Breakpoint configuration dialog again and enter this address into the Program address or expression field. Do not forget to start the address with the hexadecimal prefix 0x.
13. Close the dialog, restart your program, and run. Check that execution did indeed stop at the specified location.
14. Once you are finished with testing hardware breakpoints, make sure you disable them again by unchecking the Enable Breakpoint field. This is to prevent them from interfering with further tests.

A.3.3 Testing Hardware Data Watchpoints

15. Look at the assembly program first. It contains a variable input, which holds the values 1, 2, ..., 6 at addresses `input`, `input+1`, ..., `input+5`. There is also a variable output defined, which will be used for write accesses. The program executes a single read/write to address input/output. Then it performs a double read/write to addresses `input+2` and `input+3/output+2` and `output+3`. Finally, a dual access to addresses `input+1` and `input+5/output+1` and `output+5` is executed.
16. We start with a simple read watchpoint on a read access to address `input`. Click on AU3 Hardware Watchpoint and make sure the Data Watchpoint tab is selected. In the configuration window, type input into the Bus address or expression field. Press Convert. Check the Match on Address only field . Select Read on C or D in the Bus Select field. Select Break as action and check the Enable Watchpoint field. Click OK.
17. Restart the program and run. The execution should stop 5 cycles after the read instruction (see Table 3).
18. Now we will specify a value as well, so open the configuration window again and uncheck the Match on Address only field. In the Bus data field enter 1 and press Convert. Click OK. Restart, run, and check the latency. It should be 8 cycles now.
19. We continue with a simple write watchpoint on a read access to address `output`. Type output into the Bus address or expression field. Press Convert. Check the Match on Address only field. Select Write on E or F in the Bus select field. Make sure that the selected action is Break and that you have checked the Enable Watchpoint field. Click OK.
20. Restart the program and run. The execution should stop 8 cycles after the write instruction.
21. Now we will specify a value as well. Open the configuration window again and uncheck the Match on Address only field. In the Bus data field enter 1 and press Convert. Click OK. Restart, run, and check the latency. It should be 10 cycles now.
22. You can use the same approach as outlined in steps 16 to 21 to trigger on a dual-read access to either `input+1` or `input+5` or a dual-write to `output+1` or `output+5`. The values are 2 for `input+1/output+1` and 6 for `input+5/output+5`. The latencies are the same.
23. Finally, we will test a watchpoint on a double access. We start with a read access. Type input+2 into the address field and click Convert. Then select Single Read address D as bus and long as access size. Check the Match on Address only and Enable Watchpoint checkboxes. Click OK. Restart and run. Program execution should stop 5 cycles after the line containing the access.
24. To use a watchpoint on both the address and the data content of a long access, we have to keep in mind that the DSP is big endian. This means, that the data in the memory location with the lower address is the most significant word, i.e. the upper 16 bits of a 32-bit value. Uncheck the Match on Address only checkbox, type 0x00030004 into the bus data field, and click Convert. Click OK.
25. Restart the program and run. Latency should be 8 cycles.
26. In order to test a write watchpoint, type output+2 into the address field and click Convert. Then select Single Write address E as bus and long as access size. Check the Match on Address only and Enable Watchpoint checkboxes.

27. Restart and run. Program execution should stop 8 cycles after the line containing the access.
28. To use a watchpoint on both the address and the data of a long access, uncheck the Match on Address only checkbox, type 0x00030004 into the data field, and click Convert. Click OK. Restart the program and run. Latency should be 10 cycles now.

A.3.4 Testing Hardware Watchpoints on I/O Accesses

29. Of course you can also use watchpoints on I/O accesses. The demonstration program contains both a write access to I/O address 0xC00 (typically DMA configuration resister) and a read-access from there.
30. In order to test the write access, type 0xC00 into the address field and click Convert. Type 0 into the bus data field and click Convert. Select Single Write address E as bus and I/O as access type. Check the Enable Watchpoint checkbox.
31. Restart and run. Program execution should stop 8 cycles after the line containing the I/O read access.
32. To test a read access, again type 0xC00 into the address field and click Convert. Type 0 into the bus data field and click Convert. Select Single Read address D as bus and I/O as access type. Check the Enable Watchpoint checkbox.
33. Restart and run. Program execution should stop 8 cycles after the line containing the I/O write access.

A.3.5 Chaining Watchpoints

34. We will now chain two watchpoints together, meaning that a trigger signal will only be generated after both watchpoints have occurred. We want to trigger on the following sequence:
    a. Read access to `input` with value `1`
    b. Write access to `output+5` with value 6
35. Open the dialog for the AU3 watchpoint. Type input in the address field. Press Convert. Type 1 in the data field. Press Convert. Select Read on C or D. Check Match on AU3 and AU4. Check Sequential. Click OK.
36. Open the dialog for the AU4 watchpoint. Type output+5 in the address field. Press Convert. Type 6 in the data field. Press Convert. Select Write on E or F. Make sure the Enable Watchpoint checkbox is checked. Press OK.
37. Restart your program and run. It should stop 9 cycles after the write access to `output+6`.

A.3.6 Testing Counters

38. Finally, we will have a quick look onto the counters and their capabilities. For doing so, disable the watchpoint from the previous section. Then double-click one of the 16-bit counters. The counter configuration dialog will appear. There, check the Enable 16-Bit counter 0 field. Select Count Cycles for Counter 0. Click OK.
39. Restart and run the program. The program should execute in about 84 cycles. This way of using a hardware counter for counting cycles is a very convenient way to profile certain sections of a program while running it in real time.

40. We cannot only count cycles, but also all sorts of events as for instance cache misses and pipeline stalls. We want to check if there is a data pipeline stall in our program. This is a wait state incurred in the course of a data access to memory. Look on the program and on the linker command file. Where could such a stall possibly occur?
41. This program is arranged as such that all data and program components reside in internal memory, so we don't get stall cycles from slow memory response times. We can, however, get stalls, for example, when more than one access to a memory location in SARAM is done at the same time. This is the case for the dual-read operation — both operands reside in SARAM and are accessed at the same time.
42. In order to detect such a stall, open 16-bit Counter 1. Check the Enable 16-Bit counter 1 checkbox and select Count external input 1 as count event. Then, select Data fetch NULL as external input event. Type 0 in the Initial count value field for Counter 1 and type 1 in the Counter match value field for Counter 1. Check the Stop on match checkbox. Click OK to close the window.
43. Restart your program and run. The program should stop after about 38 cycles with an 8-cycle latency to the dual-read instruction. Note that here the latency depends on the event you are counting. For instance, the occurrence of a stall in the program pipeline is detected earlier than a stall in the data pipeline.

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 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.

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265