Optimizing TMS320C55x Assembly Code Using the Pipeline Stall Analyzer Tool

Cesar Iovescu *C5000 Applications*

ABSTRACT

The TMS320C55x digital signal processor (DSP) architecture features a protected pipeline composed of two decoupled segments: a fetch pipeline and an execution pipeline. In this architecture, memory, registers and other resources are accessed simultaneously by the instructions flowing through the pipeline. The pipeline protection unit guarantees that these accesses are performed in a correct order by inserting stall cycles, which will affect the overall performance of the assembly code. In order to optimize assembly code with respect to pipeline stalls, the DSP programmer needs to determine where the stalls occur, what the origin of the stalls is, and how, if possible, to remove the stalls. The pipeline stall analyzer tool was developed to help the TMS320C55x DSP assembly programmers answer these questions and to allow them to efficiently optimize the critical sections of his code.

This application report presents the pipeline stall analyzer tool and provides two examples of assembly code optimization using this tool. Various types of stalls and techniques to remove these stalls are herein described.

Contents

List of Figures

1 Introduction

The constant requirement for more computation power has lead to increasingly higher processor speeds in the DSP applications. The increased clock rate in today's DSPs allow processor architectures with more computational resources and more complex pipelines. The TMS320C55x DSP has a protected pipeline composed of two independent segments: the fetch pipeline and the execution pipeline. The pipeline protection unit manages the accesses to the processor's resources and also inserts stall cycles in the instruction pipeline flow to ensure that the execution flow is correct. These stall cycles may heavily affect the performance of the code if they are present in critical code sections.

In the past, programmers have used different techniques to find out how many pipeline stalls the code had and where the stalls were located. One of the most common techniques was to perform intensive benchmarking by inserting "nop" instruction at locations where a stall was believed to occur. This practice has always been time consuming and tedious, but unfortunately indispensable.

In order to help the DSP programmers, Texas Instruments has developed a pipeline stall analyzer tool integrated in the Code Composer Studio IDE v2.2. This application report gives a brief description of the C55x pipeline before presenting the pipeline stall analyzer tool. Following that disussion is an example of assembly code optimization using this tool, which includes several types of stalls and several techniques to remove these stalls.

2 TMS320C55x Pipeline

The C55x pipeline has two segments: the fetch pipeline and the execution pipeline.

- The fetch pipeline fetches 32-bit instruction packets from memory. The fetch pipeline has 4 stages: pre-fetch1 (PF1), pre-fetch2 (PF2), fetch (F), and pre-decode (PD). It dispatches up to 48-bit instruction packets to the execution pipeline.
- The execution pipeline decodes instructions, and performs memory accesses and data computation. The execution pipeline has 7 stages: Decode (D), Address (AD), Access 1 (AC1), Access 2 (AC2), Read (R), Execute (X), and Write (W).

These two pipeline segments work independently of each other. Each segment can produce stalls, if necessary, to guarantee the correct execution of the program.

Fetch pipeline stalls, also called instruction buffer queue (IBQ) stalls, will generally occur when the processor executes dense code, i.e., code that has many large instructions (5 or 6 bytes). Indeed, since the processor fetches only 4 bytes every cycle, it will not be able dispatch 5- or 6-byte instructions every cycle. The fetch pipeline will eventually stall in order to fetch more 4-byte packets from memory.

Execution pipeline stalls will generally occur when an instruction is trying to use resources that are not available at that time. Take, for example, an instruction that is trying to read a register which has been updated in the previous instruction. If the write to the register happens in a pipeline stage after the read of the register, then the pipeline protection unit will produce a stall so that the final result would be correct.

For more information about the C55x pipeline, refer to the *TMS320C55x DSP Programmer's Guide* (SPRU376).

The impact of the pipeline stalls on the overall performance of a code depends on the architecture of the execution pipeline. The more pipeline stages a processor has, the more its performance will be affected by the pipeline stalls. In order to run at very high clock rates, the DSP architectures are required to use many pipeline stages. The TMS320C55x architecture has a seven-stage execution pipeline. Therefore, the performance of the C55x assembly code can be greatly affected by pipeline stalls. To optimize the assembly code the programmer must answer the following questions:

- How many pipeline stalls do I have in my code?
- Where are the pipeline stalls occurring?
- What causes the pipeline stalls?
- How can I remove these stalls?

In the past, the DSP programmer's tool to analyze the pipeline stalls was the breakpoint and the "nop" instruction. By setting breakpoints at different locations in the code, the programmer was benchmarking the code in order to answer the first question above. The, "nop" instructions were placed in the code and checked to determine if the stall was still present and locate the stalls (this step answered question two). After locating the stalls, the programmer had to determine which resource was causing the stalls and try to remove them. This procedure was tedious and many times did not yield any result.

To help the programmer understand and remove the pipeline stall, Texas Instruments developed a pipeline stall analyzer tool integrated in the Code Composer Studio IDE v2.2. This tool is described in the next section.

3 Pipeline Stall Analyzer Tool

The pipeline stall analyzer tool was developed to give C55x programmers the pipeline stall information they need in order to optimize their assembly code.

The pipeline stall analyzer tool can be instantiated from the Tools menu when the C55x simulator is used from within Code Composer Studio environment.

The pipeline stall analyzer window (Figure 1) opens by selecting Tools\pipeline stall analyzer.

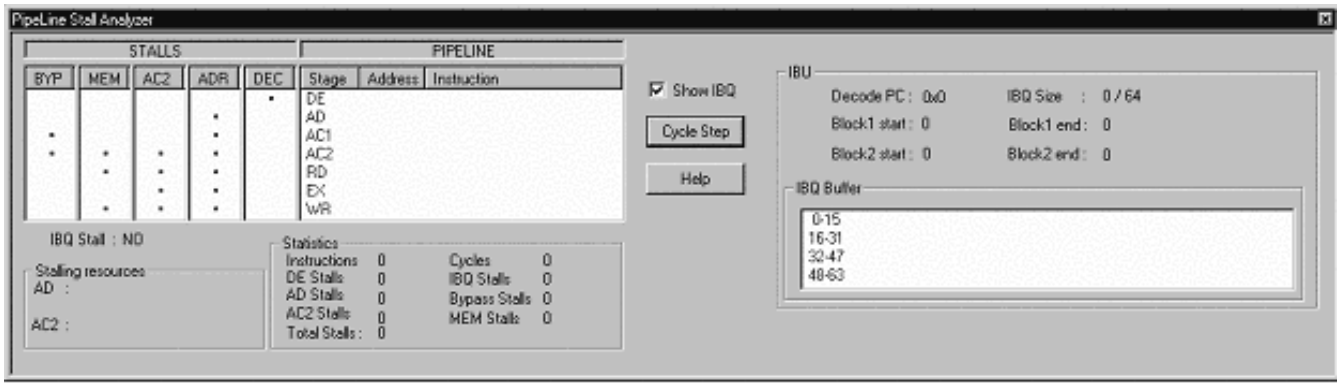

Figure 1. Pipeline Stall Analyzer Window

The window has five sections:

- Pipeline section (Figure 2)

  The Pipeline section displays the C55x DSP pipeline stages and the instructions in each stage.

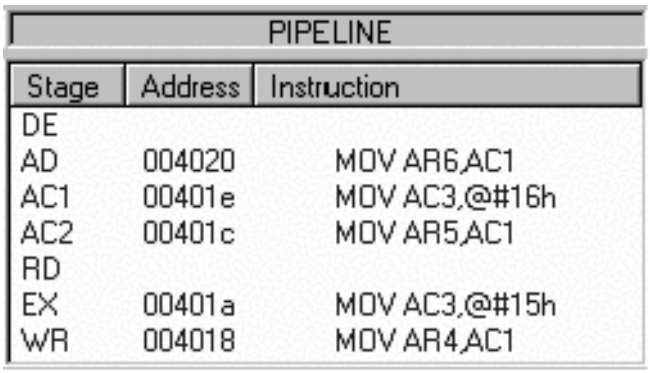

Figure 2. Pipeline Section

- Stalls section (Figure 3)

  The Stalls section shows the five possible types of stalls: bypass (BYP), memory (MEM), access 2 (AC2), address (ADR), and decode (DEC).

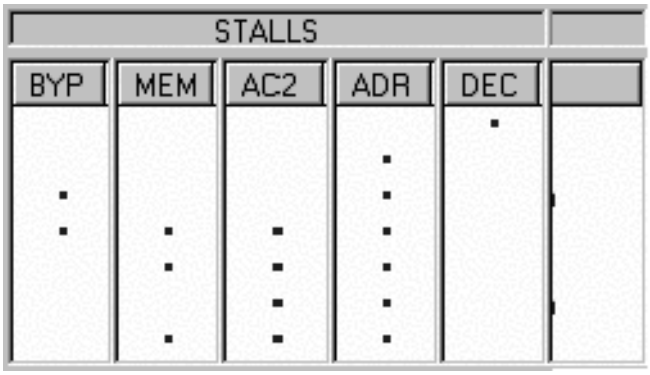

Figure 3. Stalls Section

- Stalling resources section (Figure 4)

  This section shows resources involved in stalls.

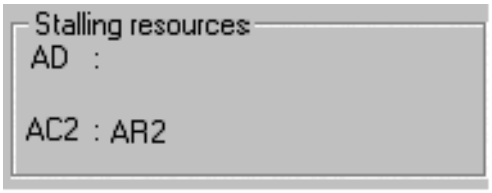

Figure 4. Conflicting Resources Section

- Statistics section (Figure 5)

  This section shows statistics about the stalls, cycle count, and instruction count. This section is useful for benchmarking a larger section of code.

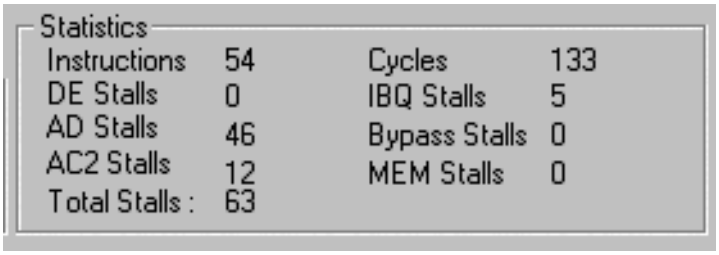

Figure 5. Statistics Section

- IBU section (Figure 6)

  This section is visible only if the Show IBQ check box is checked.

  The IBU section displays information about the instruction buffer queue (IBQ). The content of the IBQ is visible for each cycle.

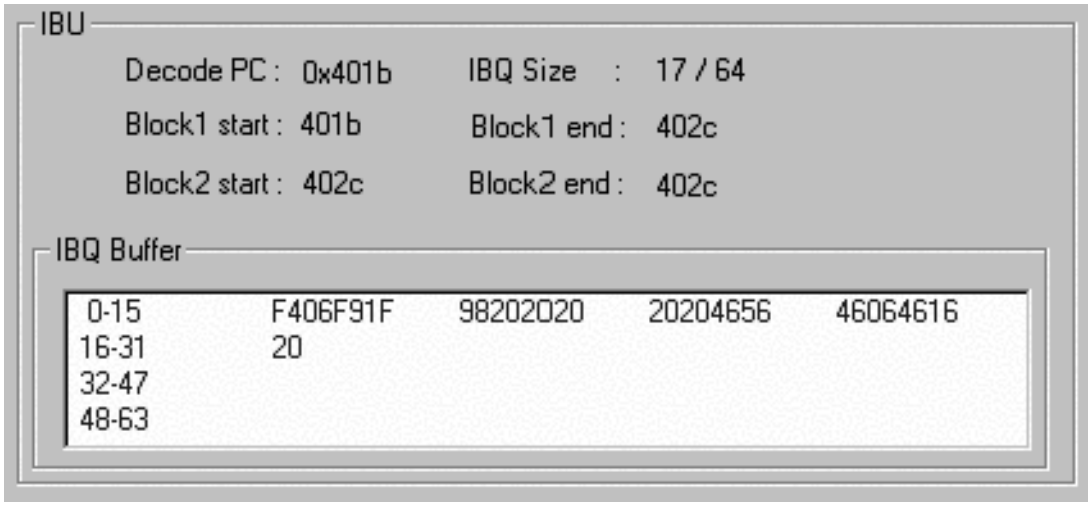

Figure 6. IBU Section

For more information about the pipeline stall analyzer window please read the Help document.

The information provided in the sections presented above help the programmers to quickly answer the questions mentioned in section 2:

- How many pipeline stalls do I have in my code?

  The Statistics section will show how many stalls and what type of stalls are present in the code.

- Where are the pipeline stalls occurring?

  The Pipeline and Stall Sections will help to locate the stalls. When the user “cycle steps” through the code, a red sign indicates the instruction that stalls.

- What causes these pipelines to stall?

  The Conflicting Resources and IBU sections will help understand the causes of the stalls.

- How can I remove these stalls?

  After understanding the nature of the stalls, the programmer can find tips about removing the stalls in the *TMS320C55x DSP Programmer’s Guide* (SPRU376).

4 Using the Pipeline Stall Analyzer to Optimize a Radix-4 FFT Kernel

This example describes how to use the pipeline stall analyzer to optimize a “simple” radix-4 FFT kernel. We use the name “simple” for this radix-4 kernel because this kernel is usually implemented at the beginning of a decimation in time (DIT) FFT processing or at the end of a decimation in frequency (DIF) FFT processing. It results from the combination of two radix stages with twiddle factors one and zero. For this reason we use the qualifier “simple.” For more information about the FFT implementation, please read the *TMS320C55x DSP Library Programmer’s Reference* (SPRU422).

The equations of this kernel are the following:

```
; ------------------------
; In-place radix-4 stage
;
;         ----------
; a -----|   R    |----- a'
;            A
; b -----|   D    |----- b'
;            I
; c -----|   X    |----- c'
;
; d -----|   4    |----- d'
;         ---------
;
; ar' =  (ar + cr) + (br + dr)
```

```
        ; ai' =  (ai + ci) + (bi + di)
        ;
        ; br' =  (ar + cr) – (br + dr)
        ; bi' =  (ai + ci) – (bi + di)
        ;
        ; cr' =  (ar – cr) + (bi – di)
        ; ci' =  (ai – ci) – (br – dr)
        ;
        ; dr' =  (ar – cr) – (bi – di)
        ; di' =  (ai – ci) + (br – dr)
```

These equations were implemented in the following assembly code:

```
  RPTB r4_loop
  MOV  AC3, dbl(*AR2+)              ; out(cr',ci')
||MOV  dbl(*AR0), AC3               ; in(ar,ai)

  ADD  dual(*AR1), AC3, AC2         ; (ar+br)/(ai+bi)
||MOV  pair(HI(AC0)), dbl(*AR3+)    ; out(dr',di')
  MOV  pair(LO(AC0)), dbl(*AR5+)    ; out(br', bi')
||MOV  dbl(*AR2), AC1               ; in(cr,ci)

  SUB  dual(*AR1+), AC3            ;(ar–br)/(ai–bi)
||MOV  AC2, dbl(*AR6)              ; out(ar+br),(ai+bi)

  ADD  dual(*AR3), AC1, AC3        ; (cr+dr)/(ci+di)
||MOV  AC3, dbl(*AR7(T0))          ; (ar–br),(ai–bi) (unaligned)

  SUB  dual(*AR3), AC1             ; (cr–dr)/(ci–di)
  MOV  AC2, dbl(*AR0+)
||ADD  dual(*AR6), AC3, AC2        ;(ar+br)+(cr+dr) = ar'
                                   ; (ai+bi)+(ci+di) = ai'
                                   ; (ar+br)–( cr+dr) = di'
  SUB AC3, dual(*AR6), AC3         ; (ai+bi)–( ci+di) = ci'
||MOV AC1, T2                      ; move (ci–di)
                                   ; (ar–br)–(ci–di) = dr'
  SUBADD  T2, *AR7(T0), AC0        ; (ar–br)+(ci–di) = br'
||MOV  HI(AC1), T3                 ; move (cr–dr)
r4_loop:
  ADDSUB  T3, *AR7, AC1            ; (ai–bi)+(cr–dr) = di'
                                   ; (ai–bi)–(cr–dr) = bi'
```

In order to analyze the performance of this kernel, the pipeline stall analyzer can be used to see if the kernel is affected by any stalls. The following steps are performed:

1. Open the pipeline stall analyzer in the Code Composer Studio Tools Menu.
2. Set a breakpoint at the beginning of the assembly code section to analyze.
3. Set a breakpoint at the end of the assembly code section to analyze.
4. Reload the Program and Run to the breakpoint.
5. Reset Analyzer Counters: Right click cursor in the plug-in window; select Reset Counters.
6. Run to the breakpoint.

4.1 Analyze the Results in the Analyzer's Statistics Window

The pipeline stall analyzer Statistics window shows that the code has 499 total stalls (Figure 7). There are 436 IBQ stalls and 63 AC2 stalls. If we divide these numbers of stalls by the number of iterations in the loop, which is 63 (one iteration is unrolled, thus loop iterations = 64–1) we notice that we have 1 AC stall and 7 IBQ stalls per iteration. The impact of these stalls is very high since we have 10 instructions in the loop. Therefore, we need to further study these stalls in order to remove them.

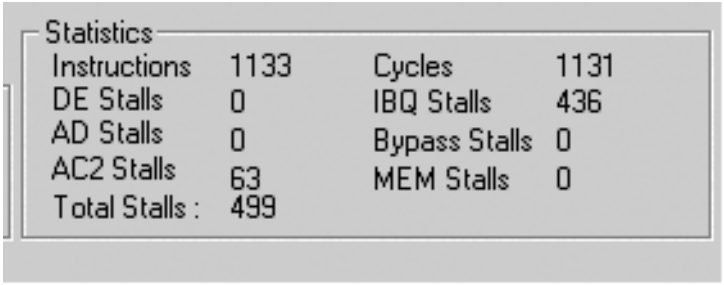

Figure 7. First Implementation – Statistics

4.2 IBQ Stalls

IBQ stalls are produced by the fetch pipeline and most of the time they are related to PC discontinuities. They can be encountered with BLOCKREPEAT, BRANCH, and CALL instructions. Some of these IBQ stalls may be resolved by properly aligning the target address. Refer to the *TMS320C55x DSP Programmer's Guide* (SPRU376) for more information about how to solve IBQ stalls.

In this code example we will first try to align the target address (loop starting address) on a 32-bit boundary by adding nop instructions before the beginning of the loop. After benchmarking the code, four IBQ stalls have been removed.

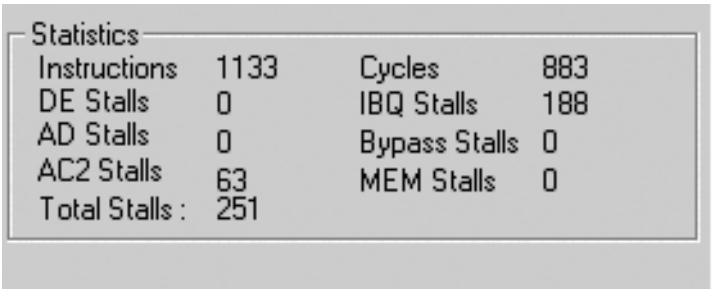

Figure 8. 32-Bit Loop Aligning – Statistics

After aligning the first instruction of the loop on a 32-bit boundary, we still have 3 IBQ stalls in the loop. In order to better understand why these stalls happen, we can use the IBQ view option of the plug-in. Enabling the Show IBQ option will provide visibility on every cycle into the contents of the IBQ (Figure 9). This information will help you to understand why there are IBQ stalls in dense code after a PC discontinuity. You must perform the following steps:

1. Debug\Reset CPU in Code Composer Studio menu.
2. Set a breakpoint at the beginning of the assembly code section to analyze.
3. Reload the Program and Run to the breakpoint.
4. Reset Analyzer Counters: Right click cursor in the plug-in window; select Reset Counters.
5. Open the IBQ window by checking the “Show IBQ” box in the Pipeline Analyzer window.
6. Cycle step through the code.

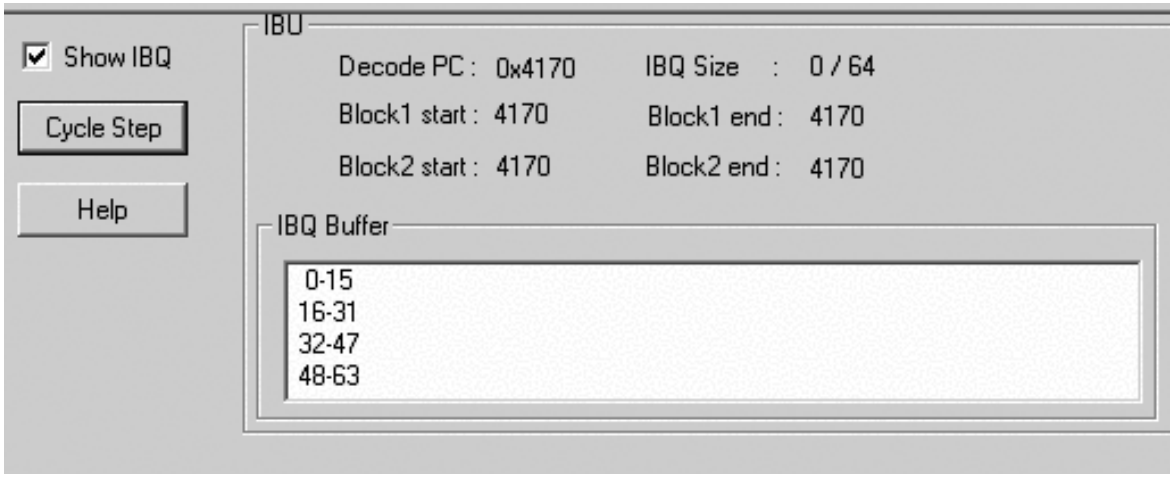

Figure 9. Empty IBQ: IBQ Stall

After understanding what causes the IBQ stalls in a piece of code, you may or may not be able to further optimize the code. Fortunately, the size of the loop is smaller than 55 bytes, which tells us that we will be able to use a LOCALREPEAT loop instead of the BLOCKREPEAT loop. The content of a LOCALREPEAT loop is fully contained in the IBQ; therefore, LOCALREPEAT loops are totally immune to IBQ stalls and there is no special memory alignment required.

In this case, we totally remove the IBQ stalls by using a LOCALREPEAT loop (Figure 10).

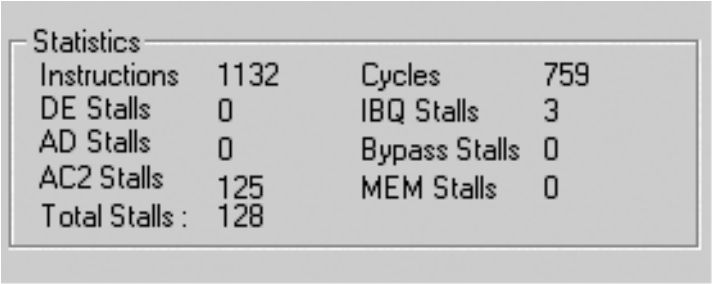

Figure 10. LOCALREPEAT Loop: No IBQ Stalls

4.3 AC2 Stalls

In order to understand the AC2 stalls we will follow the procedure described in the first example and “cycle step” through the loop code. This helps us to realize that the AC2 stalls are produced by the presence of T2 and T3 in the following instruction sequence:

```
                              ; (ar+br)–( cr+dr) = cr’
  SUB  AC3, dual(*AR6), AC3   ; (ai+bi)–( ci+di) = ci’
||MOV  AC1, T2                ; move (ci–di)
                              ; (ar–br)–(ci–di) = dr’
  SUBADD  T2, *AR7(T0), AC0   ; (ar–br)+(ci–di) = br’
||MOV  HI(AC1), T3            ; move (cr–dr)
  ADDSUB  T3, *AR7, AC1       ; (ai–bi)+(cr–dr) = di’
                              ; (ai–bi)–(cr–dr) = bi’
```

The reason for this stall is that the instruction “MOV ACx, Tn” updates the register Tn in the EXECUTE phase of the pipeline, whereas the instruction “SUBADD Tn, *AR7(T0), AC0” reads the Tn register in the READ phase of the pipeline. For more information about this type of stall, please read *The TMS320C55x DSP Programmer’s Guide* (SPRU376).

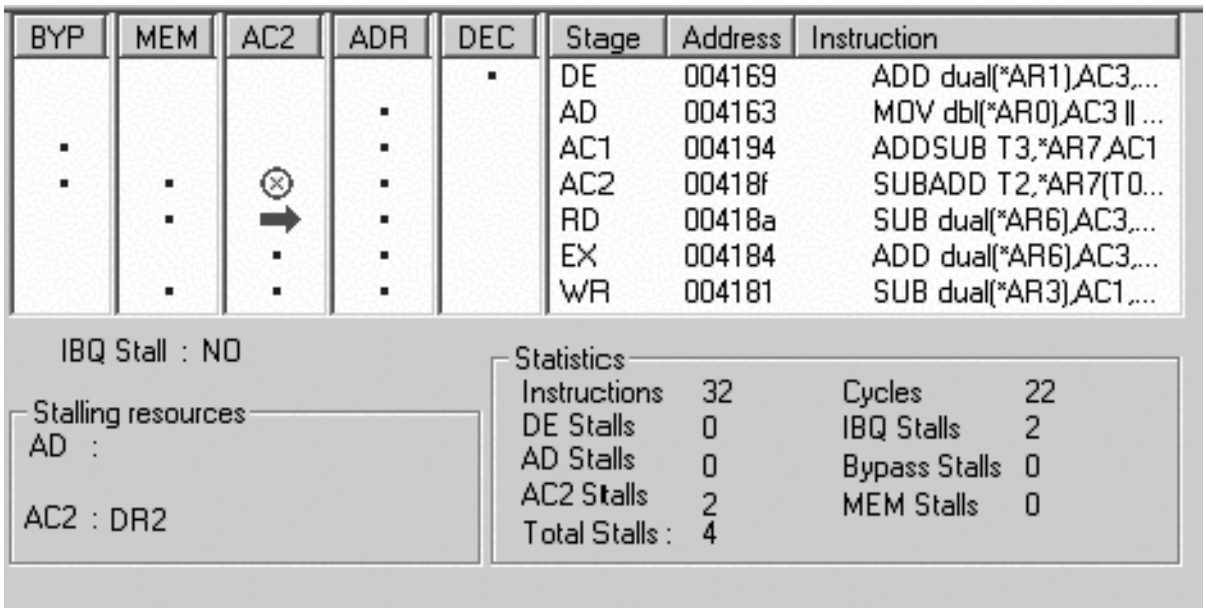

Figure 11. AC2 Stalls

The technique to remove this type of stall is to change the order of the instructions in order to separate the instructions causing the stall. The efficiency of this technique will depend on the particular implementation you are trying to optimize. For the implementation we are analyzing, we were able to rearrange the instructions and remove the stalls:

```
      RPTBLOCAL r4_loop

  MOV  AC3, dbl(*AR2+) = AC3       ; out(br', bi')
||MOV dbl(*AR0), AC3               ; in (ar,ai)

  MOV pair(HI(AC0)), dbl(*AR3+)    ; out(dr',di')
||ADD dual(*AR2), AC3, AC2         ; (ar+cr)/ (ai+ci)

  MOV dbl(*AR1), AC1               ; in (br,bi)
||MOV pair(LO(AC0)), dbl(*AR5+)    ; out (cr', ci')

  SUB dual(*AR3), AC1, AC2         ; (br–dr)/ (bi–di)
||MOV AC2,dbl(*AR6)                ; temp (ar+cr),(ai+ci)

  SUB dual(*AR3), AC3, AC0         ; (ar–cr)/ (ai–ci)
||MOV Hi(AC2),T3                   ; move(br–dr)

  ADD dual(*AR3), AC1, AC3         ; (br+dr)/ (bi+di)
```

```
||MOV AC2,T2                        ; move (bi–di)

                                    ; (ar+cr) + (dr+br) = ar’
  ADD dual(*AR6), AC3, AC0          ; (ai+ci) + (di+bi) = ai’
||MOV AC0, dbl(*AR7(T0))            ; (ar–cr),(ai–ci) unaligned

                                    ; (ar+cr) – (dr+br) = br’
  SUB dual(*AR6), AC3               ; (ai+ci) – (di+bi) = bi’
||MOV AC0, dbl(*AR0+)

  SUBADD T2, *AR7(T0), AC0          ; (ar–cr) – (bi–di) = dr’
                                    ; (ar–cr) + (bi–di) = cr’
r4_loop:
  ADDSUB T3, *AR7, AC1              ; (ai–ci) + (br–dr) = di’
                                    ; (ai–ci) – (br–dr) = ci’
```

We notice that the sequence of conflicting instructions was modified and the sources of the stalls removed:

```
                                    ; (ar+cr) – (dr+br) = br’
  SUB dual(*AR6), AC3               ; (ai+ci) – (di+bi) = bi’
||MOV AC0, dbl(*AR0+)

  SUBADD T2, *AR7(T0), AC0          ; (ar–cr) – (bi–di) = dr’
                                    ; (ar–cr) + (bi–di) = cr’
r4_loop:
  ADDSUB T3, *AR7, AC1              ; (ai–ci) + (br–dr)  = di’
                                    ; (ai–ci) – (br–dr) = ci’
```

We benchmarked this code and we show that we removed the access stalls.

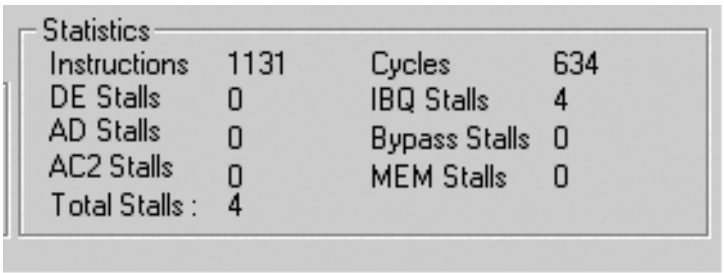

Figure 12. Final Optimized Code – Statistics

5 Conclusion

The pipeline stall analyzer tool provides useful information that helps C55x assembly programmers optimize code by understanding and removing the pipeline stalls. The information provided by the Statistics window enables programmers to evaluate the impact of stalls on their code and to focus on the critical sections. The Pipeline Stalls and Conflicting Resources windows present the detailed information that will help the programmers optimize these sections.

In the Radix-4 FFT kernel example, the optimization of the code using the pipeline stall analyzer, improved the performance by 20%. In further tests of optimizing other kernels, 20–30% improvements were common when using the pipeline stall analyzer.

6 References

1. *TMS320C55x DSP Programmer's Guide* (SPRU376)
2. *TMS320C55x DSP Library Programmer's Reference* (SPRU422)

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