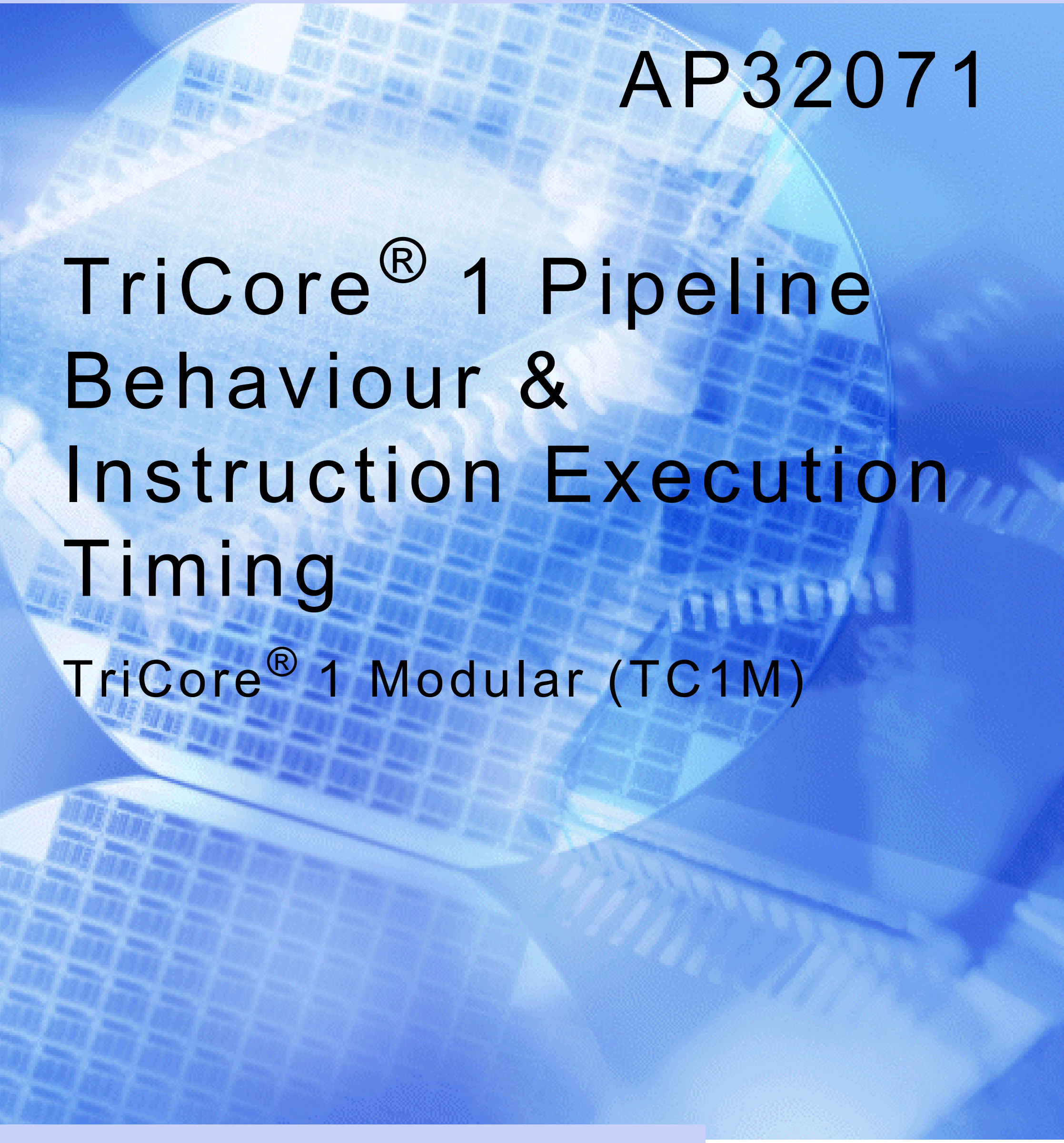
Application Note, V 1.1, June 2004

AP32071

TriCore® 1 Pipeline Behaviour & Instruction Execution Timing

TriCore® 1 Modular (TC1M)



Microcontrollers

Never stop thinking

TriCore® 1 Pipeline Behaviour & Instruction Execution Timing

Revision History: **2004-06** V 1.1

Previous Version:-

Page	Subjects (major changes since last revision)
	Minor changes

1 Introduction

This application note describes pipeline behavior and its effect on TriCore®'s performance. The document gives a description of pipeline operation and then addresses the effect of both hazards and dependencies on the overall performance.

Although several cases are considered, covering both superscalar and non-superscalar effects, this application note does not attempt to exhaustively cover all possible scenarios. Instead it aims to help educate the TriCore 1 user so that they can develop highly optimized code for superior performance.

2 Pipeline description

TriCore 1 uses a superscalar implementation, incorporating two major pipelines and one minor pipeline.

The two major pipelines are each 4 stages deep and consist of a shared Fetch stage, followed by Decode, Execute and Writeback stages. One pipeline, called the Integer pipeline (I-pipe), mainly handles the data arithmetic instructions, including the data conditional jumps. The other pipeline, called the Load Store pipeline (L/S-pipe), mainly handles load/store memory accesses, address arithmetic, unconditional jumps, calls and context switching operations.

The third pipeline, the Loop pipeline, is used mainly for the loop instruction. Its purpose is to provide zero-overhead loops.

TriCore can issue one instruction per cycle into each of the major pipelines, with the following constraints:

1. Back to back data arithmetic instructions may only be issued on separate cycles.
2. Instructions to the L/S-pipe can be issued either:
   a) On their own (single)
   b) Paired with a data arithmetic instruction, provided they're the second of the pair

Pairing-up instructions would provide the highest performance, which in the ideal case will result in 0.5 clock cycles (cycles – for short) per instruction or 2 instructions per cycle. However this assumes that the program is written in such a way that this policy is possible. Although the optimizing compilers will do most of the work, it is up to the programmer to further optimize the code for the highest performance possible.

It is important to remember that if one of the major pipelines is stalled, the other one will automatically be stalled for the same amount of time. The exception is the issue of an unpaired load/store instruction, where the load/store instruction is issued to the L/S-pipe and the I-pipe is stalled.

An illustration of pipeline operation is shown in the table below for the following code example:

```
add     d1, d2, d3      ; i1
add     d4, d5, d6      ; i2
ld      a3, [a2]0       ; l1
ld      a4, [a5]0       ; l2
sub     d7, d8, d9      ; i3
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	i1	i2	-	i3				
	Execute		i1	i2	-	i3			
	Writeback			i1	i2	-	i3		
Load / Store Pipeline	Decode	-	l1	l2	-				
	Execute		-	l1	l2	-			
	Writeback			-	l1	l2	-		

Note: ‘ - ’ denotes a stall cycle (NOP) when seen in any table in this document

As shown in the table above, i2 and l1 are issued in parallel, so the five instructions will take only 4 cycles to complete. l2 can only be issued on its own (according to the rules given above) and the result is a stall cycle in the I-pipe. However, by swapping the order of l2 and i3, these two instructions can be issued in parallel and the 5 instructions will be executed in only 3 cycles.

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	i1	i2	i3					
	Execute		i1	i2	i3				
	Writeback			i1	i2	i3			
Load / Store Pipeline	Decode	-	l1	l2					
	Execute		-	l1	l2				
	Writeback			-	l1	l2			

3 General Case

The General Case covers the situation in which instructions are independently issued; i.e. No data dependencies or hazards are taken into account.

TriCore has been designed with high performance in mind and therefore, with very few exceptions, ***ALL*** instructions take only one clock cycle to complete (the clock referred to here is the pipeline clock). The main exceptions which take more than one cycle are the multiplication and branch instructions, although there are also a small number of other seldom-used instructions which take more than one cycle.

Like all pipeline-based systems, TriCore exhibits structural, control and data dependencies (hazards). This section covers Structural and Control hazards, while the data dependencies will be discussed in sections which follow.

3.1 Structural Hazards

Structural hazards are the result of a hardware resource that can not be accessed, in any given stage of the pipeline, by more than one instruction.

An example of a Structural hazard is illustrated here:

```
add     d0, d1, d2      ; A
ld      d0, [a0]0       ; L
```

Both the add and the load instruction, issued in parallel, target the same destination register:

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	A							
	Execute		A						
	Writeback			**A**					
Load / Store Pipeline	Decode	L							
	Execute		L						
	Writeback			**L**					

The Write After Write (WAW) hazard is detected in the decode stage and is removed by stalling the pipeline for one cycle (*see the table which follows*).

Structural Hazards - continued

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	A	-						
	Execute		A	-					
	Writeback			A	-				
Load / Store Pipeline	Decode	L	L						
	Execute		-	L					
	Writeback			-	L				

Note: ' - ' denotes a stall cycle (nop)

Store Followed By A Dependent Load

Another example of a Structural hazard is a store followed by a dependent load. This is illustrated here:

```
add       d0, d2, d3      ; A1
st.w      [a0]0, d1       ; S1
add       d4, d5, d3      ; A2
ld.w      d4, [a0]0       ; L1
add       d7, d6, d3      ; A3
add.a     a5, a8, a9      ; AA
```

If the store is followed by an independent load (i.e. there is no overlap between the data being stored and the data to be loaded), then the program shown above will be executed in 3 cycles (AA uses the L/S pipe since it performs address manipulations).

A1//S1 (A1 in parallel with S1) executes just before A2//L1 and both S1and L1 try to access the same memory location in the same cycle.

See the diagram which follows.

		1	2	3	4	5	6	7	8
	Decode	A1	A2	**A3**	A3	A3			
Integer Pipeline	Execute		A1	**A2**	-	-	A3		
	Writeback			**A1**	A2	-	-	A3	
	Decode	S1	L1	**AA**	AA	AA			
Load / Store Pipeline	Execute		S1	**L1**	L1	L1	AA		
	Writeback			**S1**	-	-	L1	AA	
Store Buffer				**S1**	S1				

The dependency is detected during cycle 3. At that point the load is aborted, the pipelines are stalled and the store is allowed to commit to memory on the following cycle. Once the store has been committed to memory, the load is allowed to proceed.

3.2 Control Hazards

Control Hazards are related to the execution of the jump instructions, both **conditional** and **unconditional**.

Unconditional Jumps

Because the behavior of the pipeline is similar for all 3 types of unconditional jumps (relative, absolute and indirect), only the relative case is described here in detail.

Unconditional jumps are issued to the L/S pipe and are detected at the decode stage. In the decode stage the PC (Program Counter) and displacement calculation is performed and the new PC passed to the fetch unit. Consider the following example:

```
        ld.w      d3, [a2]0       ; L
        add       d4, d6, d2      ; A
        j         target          ; J
        add       d2, d9, 1       ; N

target:
        add       d4, d5, d9      ; T
        sh        d4, d2, 1       ; T+1
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Fetch	-	A	*N*	T	T+1	🟦	🟦	🟦
	Decode	🟦	-	A	-	T	T+1	🟦	🟦
	Execute	🟦	🟦	-	A	-	T	T+1	🟦
	Writeback	🟦	🟦	🟦	-	A	-	T	T+1
Load / Store Pipeline	Fetch	L	J	-	-	-	🟦	🟦	🟦
	Decode	🟦	L	J	-	-	-	🟦	🟦
	Execute	🟦	-	L	J	-	-	-	🟦
	Writeback	🟦	🟦	-	L	J	-	-	-

The jump is detected in the L/S pipe's decode stage (cycle 3) and the target of the jump is fetched in cycle 4, introducing a single stall cycle.

Conditional Jumps

Conditional jumps depend on either data or address register values. Depending on which type of register is checked, the conditional jump instructions are detected in the decode stage of either the I-pipe or L/S pipe and the condition is resolved in the execute stage.

A static branch prediction algorithm is used to improve the performance of the conditional jump instructions. The algorithm used is:

- 16 bit conditional jump instruction, predicted taken
- 32 bit conditional bit instruction, backward displacement, predicted taken
- 32 bit conditional jump instruction, forward displacement, predicted not taken

The penalty (in extra cycles) for wrongly predicting the outcome of the conditional jump is as follows:

	Predicted	
	Taken	**Not Taken**
Taken	1	2
Not Taken	2	0

When the jump is resolved in the execute stage of the pipeline, the actions taken are independent of whether the branch was single or dual issued.

The following example illustrates the pipeline behavior when the conditional data jump is single issued:

```
        add     d4, d6, d2      ; A
        jeq     d0, d1, target  ; J
        add     d0, d0, 1       ; N
target:
        add     d4, d5, 0       ; T
```

Pipeline examples for resolved / unresolved jeq, follow.

If **jeq** is resolved, then the pipeline will appear as follows.

		1	2	3	4	5	6	7	8
Integer Pipeline	Fetch	A	J	*N*	*N+1*	T			
	Decode		A	J	*N*	-	T		
	Execute			A	**J**	-	-	T	
	Writeback				A	J	-	-	T
Load / Store Pipeline	Fetch								
	Decode								
	Execute								
	Writeback								

If **jeq** is resolved not taken, the pipeline will appear as follows:

		1	2	3	4	5	6	7	8
Integer Pipeline	Fetch	A	J	N					
	Decode		A	J	N				
	Execute			A	**J**	N			
	Writeback				A	J	N		
Load / Store Pipeline	Fetch								
	Decode								
	Execute								
	Writeback								

The other two cases can be similarly analyzed.

4 Hazards & Dependencies (No Superscalar Effects)

This section covers the effect of data dependencies on various instructions, but without taking into consideration whether the dependency is due to the fact that the two instructions are issued in parallel.

TriCore significantly limits the impact of data dependencies by using extensive 'forwarding' paths within the CPU pipelines. These forwarding paths allow the result of one instruction to be passed to the inputs of a following instruction without having to wait for the write and subsequent read from the register file. The two examples which follow illustrate the improvement in performance from the concept of forwarding.

Example 1

```
add     d0, d1, d2     ; A
sub     d4, d3, d0     ; S
```

In this example register d0 is updated by the add instruction in cycle 3, therefore the sub instruction has to wait two cycles (data is read in the decode stage of the pipeline) until data is available in cycle 4.

Without Forwarding

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	A	S	S	S	🟦	🟦	🟦	🟦
	Execute	🟦	A	-	-	S	🟦	🟦	🟦
	Writeback	🟦	🟦	A	-	-	S	🟦	🟦
Load / Store Pipeline	Decode	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	Execute	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	Writeback	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦

With Forwarding

Integer Pipeline	Decode	A	S	🟦	🟦	🟦	🟦	🟦	🟦
	Execute	🟦	A	S		🟦	🟦	🟦	🟦
	Writeback	🟦	🟦	A	S	🟦	🟦	🟦	🟦
Load / Store Pipeline	Decode	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	Execute	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	Writeback	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦

Example 2

```
add     d0, d1, d2      ; A
or      d7, d8, d9      ; O
sub     d4, d3, d0      ; S
```

Without Forwarding

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	A	O	S	S				
	Execute		A	O	-	S			
	Writeback			A	O	-	S		
Load / Store Pipeline	Decode								
	Execute								
	Writeback								

With Forwarding

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	A	O	S					
	Execute		A	O	S				
	Writeback			A	O	S			
Load / Store Pipeline	Decode								
	Execute								
	Writeback								

5 Hazards & Dependencies (With Superscalar Effects)

When considering superscalar effects, the potential data dependencies would mostly concern the MAC (Multiply & Accumulate) instruction. This is because the MAC instruction takes an extra cycle to complete, as it involves two execute stages: EX1 and EX2. Two examples follow, illustrating some of the potential data dependencies with Superscaler effects:

Example 1 - Interaction Between MAC Instructions & Loads

```
madd.q      d0, d0, d1, d2      ; M1
ld          d0, [a0] 0          ; L1
ld          d8, [a1] 0          ; L2
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	M1							
	Execute 1		M1						
	Execute 2			M1					
	Writeback				**M1**				
Load / Store Pipeline	Decode	L1	L2						
	Execute		L1	**L2**					
	Writeback			**L1**	L2				

This example illustrates the potential WAW hazard (Write After Write), caused by the fact that L1 enters the writeback stage a cycle earlier than M1 (L1 should be the one to make the last update to d0). In order to remove this potential hazard, L1 is stalled for two cycles:

Integer Pipeline	Decode	M1							
	Execute 1		M1						
	Execute 2			M1					
	Writeback				M1				
Load / Store Pipeline	Decode	L1	L1	L1	L2				
	Execute		-	-	L1	L2			
	Writeback			-	-	L1	L2		

The same holds true if the dependency is between the MAC and the second load:

```
madd.q      d0, d0, d1, d2     ; M1
ld          d5, [a0] 0         ; L1
ld          d0, [a1] 0         ; L2
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	M1							
	Execute 1		M1						
	Execute 2			M1					
	Writeback				**M1**				
Load / Store Pipeline	Decode	L1	L2						
	Execute		L1	**L2**					
	Writeback			**L1**	**L2**				

When this occurrence is detected, the pipeline is stalled for one cycle:

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	M1	-						
	Execute 1		M1	-					
	Execute 2			M1	-				
	Writeback				**M1**	-			
Load / Store Pipeline	Decode	L1	L2	L2					
	Execute		L1	-	L2				
	Writeback			**L1**	-	L2			

Note: ' - ' denotes a stall cycle (nop)

Example 2 – Interaction Between MAC & Context Operations

Context operations are usually performed as the result of an interrupt, a trap or a call instruction. A context operation cannot be issued when there is an instruction in the integer decode stage or when there is a MAC instruction in the execute 1 stage.

```
madd.q      d0, d0, d1, d2      ; M1
call        func_a              ; C
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	M1							
	Execute 1		M1						
	Execute 2			M1					
	Writeback				**M1**				
Load / Store Pipeline	Decode	C	C	C					
	Execute		-	-	C				
	Writeback			-	-	C			

6 Miscellaneous

All the examples in this application note are based on data that is available in the cache (or internal SRAM), and therefore can be accessed in one clock cycle.

If the data (or instruction) is not in the cache, then the pipeline will be stalled until data does become available. Depending on the implementation, this can add several cycles to the execution of each instruction.

Consider the following example, where the load misses in the cache. The cache refill is performed in a burst of 16 bytes from local SRAM memory (no wait states).

```
ld.w        d0, [a0] 0        ; L
add         d2, d0, d1        ; A
```

		1	2	3	4	5	6	7	8
Integer Pipeline	Decode	-	A	A	A	A			
	Execute		-	-	-	-	A		
	Writeback			-	-	-	-	A	
Load / Store Pipeline	Decode	L							
	Execute		**L**	**L**	**L**	**L**			
	Writeback						L		

Two points to remember when considering the effect of the cache refill on the performance are:

1. If the access spans a cache line then a multi access load will be performed, thereby introducing extra clock cycles. For the example above, another 4 cycles will be needed before the add can execute.

2. If the refill is performed from SDRAM and it occurs at the same time as the refresh should be performed, the refresh will take precedence, thereby introducing extra clock cycles to each instruction timing.

7 Conclusion

Understanding the system interactions in an embedded system based on superscalar processors with caches will help in achieving the maximum level of performance.

Recommended reading

The reader can find a detailed guide to enhanced performance in the following Infineon document: **TriCore Optimization Guide**

Contact your nearest **Infineon** sales office for a copy

You can find your nearest **Infineon** sales office at:

http://www.infineon.com/business/offices

The **TriCore** Home page can be found at:

http://www.infineon.com/tricore

Glossary

Reference	Definition
SDRAM	Synchronous Dynamic Random Access Memory
CPU	Central Processing Unit
I-pipe	Integer Pipeline
L/S-pipe	Load / Store Pipeline
PC	Program Counter
SRAM	Static Random Access Memory
TC	TriCore
WAW	Write After Write
MAC	Multiply & Accumulate
EX	Execute