



## ADSP-BF531/532/533 Blackfin® Processor Multi-cycle Instructions and Latencies

Contributed by Tom L.

September 24, 2003

### Introduction

This document contains a description of the multicycle instructions and latencies specific to the ADSP-BF531/532/533 Blackfin® Processor devices. **Multicycle** instructions take more than one cycle to complete. The specific cycle count cannot be reduced without removing the instruction that caused it. A **latency** condition can occur when two instructions require extra cycles to complete because they are close to each other in the assembly program. The programmer can avoid this cycle penalty by separating the two instructions. Other causes for latencies are memory stalls and store buffer hazards; for these conditions, a discussion of how to improve performance is provided.



The Pipeline Viewer within the VisualDSP++™ simulator provides a way of looking at the way instructions are pushed through the processor's pipeline. While the causes for various conditions like stalls can be discovered interactively, this document contains more detail about the nature of execution latencies.



All of the cycle counts described in this document are based on the assumption that code is executed from L1 memory.

### Multicycle Instructions

This section describes the instructions that take more than one cycle to complete. All instructions not mentioned in this discussion are single-cycle instructions, provided they are executed from L1 memory.

Multicycle instructions include the following categories: Push Multiple/Pop Multiple, 32-bit Multiply, Call, Jump, Conditional Branch, Returns from Events, Core and System Synchronization, Linkage, Interrupts and Emulation, and Testset. In the following examples, the total number of cycles needed to complete a certain instruction is shown next to the corresponding instruction. The full descriptions of each instruction's functionality is provided in the Blackfin Processor Instruction Set Reference.

#### *Push Multiple/Pop Multiple*

The Push Multiple and Pop Multiple instructions take  $n$  core cycles to complete, where  $n$  is the number of registers pushed or popped, assuming the stack is located in L1 data memory.

Example	Number of Cycles
<code>[--SP] = (R7:0, P5:0);</code>	14 cycles
<code>(R7:0, P5:3) = [SP++];</code>	11 cycles

### 32-bit Multiply (modulo $2^{32}$ )

The 32-bit by 32-bit integer multiply instruction always takes 3 cycles to complete.

Example	Number of Cycles
<code>R0 *= R1;</code>	3 cycles

### Call, Jump

All call and jump instructions take 5 cycles to complete, provided the target address is an aligned location (see *Instruction Alignment Unit Empty Latencies* later in this document).

Example	Number of Cycles
<code>CALL 0x22;</code>	5 cycles
<code>CALL (PC + P0);</code>	5 cycles
<code>CALL (P0);</code>	5 cycles
<code>JUMP 0x22;</code>	5 cycles
<code>JUMP (PC + P0);</code>	5 cycles
<code>JUMP(P0);</code>	5 cycles

### Conditional Branch

The number of cycles a branch takes depends on the prediction as well as the actual outcome.

Prediction	Taken		Not taken	
	Taken	Not taken	Taken	Not taken
Outcome				
Number of Cycles	5 cycles	9 cycles	9 cycles	1 cycle

### Returns from Events

Examples	Number of Cycles
<code>RTX; // return from an exception</code>	5 cycles
<code>RTE; // return from emulation</code>	5 cycles
<code>RTN; // return from an NMI</code>	5 cycles

RTI; // return from an interrupt	5 cycles
RTS; // return from a subroutine	5 cycles

#### Core and System Synchronization

Examples	Number of Cycles
CSYNC;	10 cycles
SSYNC;	>10 cycles

#### Linkage

Examples	Number of Cycles
LINK 4;	3 cycles
UNLINK;	2 cycles

#### Interrupts and Emulation

Examples	Number of Cycles
RAISE 10;	3 cycles (if interrupt branch is not taken)
EXCPT 3;	3 cycles (if exception branch is not taken)
STI R4;	3 cycles

#### Testset

The TESTSET instruction is a multicycle instruction that executes in a variable number of cycles. It is dependent on the cycles needed for a read acknowledge from off-core memory. It is also dependent on whether the address being tested is both in the cache and dirty. The number of cycles can be determined as follows,

$$\text{cycles} = 1 \text{ (instruction)} + 1 \text{ (stall)} + x \text{ (read acknowledge)} + y \text{ (cache latency)}$$

## Instruction Latencies

Unlike multicycle instructions, instruction latencies are contingent on the placement of specific instruction pairs relative to one another. They can be avoided by separating them by as many instructions as there are cycles incurred between them. For example, if a pair of instructions incurs a two cycle latency, separating them by two instructions will eliminate that latency.

**Blue** type is used to identify register dependencies within the instruction pairs. A dependency occurs if a register is accessed in the instruction immediately following an instruction that modifies the register. The lack of the color blue in a entry indicates that the latency condition will occur regardless of what registers are used. *Red* type is used to highlight the stall consequences.

Instruction latencies are separated into these groups: Accumulator to Data Register Latencies, Register Move Latencies, Move Conditional and Move CC Latencies, Loop Setup Latencies, Hardware Loop Latencies, Instruction Alignment Unit Latencies, and Miscellaneous Latencies. The total cycle time of each entry can be calculated by adding the cycles taken by each instruction to the number of stall cycles for the instruction pair.



Refer to the Appendix for abbreviations, instruction group descriptions, as well as register groupings.

#### Accumulator to Data Register Latencies

Description	Example	<Cycles + <i>Stalls</i> >
- <b>dreg</b> = Areg2Dreg op	<b>R1</b> = R6.L * R4.H (IS);	<1>
- video op using <b>dreg</b> as src	R5 = BYTEOP1P (R3:2, <b>R1</b> :0);	<1+ <i>I</i> >

#### Register Move Latencies

In each of the following cases, the stall condition occurs when the same register is used in both instructions.

Description	Example	<Cycles + <i>Stalls</i> >
- <b>dreg</b> = sysreg	<b>R0</b> = LC0;	<1>
- multiply/video op with <b>dreg</b> as src	R2.H = R1.L * <b>R0</b> .H;	<1+ <i>I</i> >
- <b>preg</b> = dreg	<b>P0</b> = R3;	<1>
- any op using <b>preg</b>	R0 = <b>P0</b> ;	<1+4>
- <b>dagreg</b> = dreg	<b>I3</b> = R3;	<1>
- any op using <b>dagreg</b>	R0 = <b>I3</b> ;	<1+4>
- POP to <b>dagreg</b>	<b>I3</b> = [SP++];	<1>
- any op using <b>dagreg</b>	R0 = <b>I3</b> ;	<1+3>
- LOAD/POP to <b>preg</b>	<b>P3</b> = [SP++];	<1>
- any op using <b>preg</b>	R0 = <b>P3</b> ;	<1+3>
- <b>dreg</b> = seqreg	<b>R0</b> = RETS;	<1>
- any ALU op using <b>dreg</b>	R1 = <b>R0</b> + R3;	<1+ <i>I</i> >
- <b>dreg</b> = MMR register	<b>R3</b> = [P0]; // P0 points to an MMR	<1>
- any ALU op using <b>dreg</b>	R0 = <b>R3</b> – R0;	<1+ <i>I</i> >

#### Move Conditional and Move CC Latencies

In each of the following cases, the stall condition occurs when the same register is used in both instructions.

Description	Example	<Cycles + <i>Stalls</i> >
- if CC <b>dreg</b> = dreg - multiply/video op using <b>dreg</b> as src	if CC <b>R0</b> = R1; R2.H = R1.L * <b>R0.H</b> ; ----- if CC <b>R1</b> = R3; SAA (R3:2, <b>R1</b> :0);	<1> <1+ <i>I</i> > ----- <1> <1+ <i>I</i> >
- if CC <b>preg</b> = dpreg - any op using <b>preg</b>	if CC <b>P0</b> = R1; R4 = <b>P0</b> ;	<1> <1+ <i>4</i> >

### Loop Setup Latencies

The following are latencies specific to the configuration of the zero-overhead looping mechanism.

Description	Example	<Cycles + <i>Stalls</i> >
- loop setup - loop setup with same LC	LSETUP (top1, bottom1) LC0 = P0; LSETUP (top2, bottom2) LC0 = P1;	<1> <1+ <i>6</i> > (3 additional cycles are required for the case “LSETUP is not followed by the first instruction of the loop” – see below)
- modification of LT or LB - loop setup with same loop registers	LT0 = [SP++]; LSETUP (top, bottom) LC0 = P0;	<1> <1+ <i>2</i> >
(if LC0 > 1) - LC0/LC1 reg written to - any op	LC0 = R0; NOP;	<1> <1+ <i>9</i> >
- LT0/LB0 written to and LC0 != 0 - any op	LT0 = [SP++]; NOP;	<1> <1+ <i>9</i> >
- LT1/LB1 written to and LC1 != 0 - any op	LB1 = P0; NOP;	<1> <1+ <i>9</i> >

### Hardware Loop Latencies

Instruction	Number of Stalls
LSETUP is not followed by the first instruction of the loop	<3> starting at the second iteration

### Instruction Alignment Unit Empty Latencies

If the instruction alignment unit (IAU) is empty of the next instruction, that next instruction will incur a stall while the IAU is being filled. The following conditions will result in an IAU empty stall.

Event	Number of Stalls
instruction cache miss or SRAM fetch miss	<1>
change of flow to an instruction address aligned across a 64-bit boundary	<1>

In the example below, a branch to instruction B at address 0xFFA0 0104 would result in a one cycle stall, because this 64-bit instruction requires two fetches by the instruction alignment unit.

Address	Data Fetched by the IAU	
0xFFA0 0100	A (32-bit instruction)	B (64-bit instruction)
0xFFA0 0108	B (64-bit instruction, continued)	C (32-bit instruction)
0xFFA0 0110	D (32-bit instruction)	E (32-bit instruction)

In order to eliminate this stall, instruction B should be moved to a 64-bit aligned location (like 0xFFA0 0110, shown below).

Address	Data Fetched by the IAU	
0xFFA0 0100	A (32-bit instruction)	MNOP
0xFFA0 0108	B (64-bit instruction)	
0xFFA0 0110	C (32-bit instruction)	D (32-bit instruction)

### Miscellaneous Latencies

The following latencies do not fall into any of the above categories.

Description	Example	<Cycles + Stalls>
- move register or POP to I0 or I1 - SAA,BYTEOP2P,BYTEOP3P	I1 = [SP++]; R0 = BYTEOP3P (R1:0, R1:0) (HI);	<1> <1+3>
- move register or POP to I0 or I1 - BYTEOP1P/16P/16M, BYTEUNPACK	I0 = R0; R3 = BYTEOP1P (R3:2, R1:0);	<1> <1+4>
- write to return register	RETI = P0;	<1>

- return op	RTI;  ----- RETS = P3; RTS;	<1+4> (number of cycles for the branch is not included)  ----- <1> <1+4> (number of cycles for the branch is not included)
- math op - video op with RAW data dependency	<b>R3</b> = R2 + R4; SAA ( <b>R3</b> :2, R1:0);	<1> <1+1>
- <b>dreg</b> = search - math op using <b>dreg</b>	(R3, <b>R0</b> ) = search R1 (LE); R2.H = R1.L * <b>R0</b> .H;	<1> <1+2>
- core and system MMR access	R0 = [P0]; // P0 = MMR address	<1+2> (for system MMR accesses, the latency due to the system acknowledge signaling will increase the total cycle count)
- L0/B0 = dreg - IO modulo update (similarly for the corresponding L1/B0 and I1 registers)	<b>L0</b> = R0; R1 = [ <b>I0</b> ++]; ----- <b>B1</b> = R2; <b>I1</b> += 4; ----- <b>L1</b> = R3; R4 = [ <b>I1</b> ++M2]; ----- <b>B0</b> = R5; <b>I0</b> += M2;	<1> <1+4> ----- <1> <1+4> ----- <1> <1+4> ----- <1> <1+4>

## L1 Data Memory Stalls

L1 data memory (DM) stalls can be incurred by accessing L1 data memory. Accesses can either be explicit (if the data memory is configured as SRAM) or implicit (if the data memory is configured as cache). Some of these stalls are multicycle instruction conditions, and some are latency conditions. The specifics are described in each entry.

**Bold blue** type is used to highlight the causal factors in offending instructions. *Italicized red* type is used to highlight the stall consequences.

## Sub-bank Access Collision

### SRAM Access (1 cycle stall)

This stall can only occur when an instruction accesses memory configured as SRAM. A major difference between the devices in the ADSP-BF531/2/3 class is memory size and configuration. The ADSP-BF531 Blackfin Processor has one data bank (data bank A), which is divided into four contiguous 4096 byte (4 KB) sub-banks. The ADSP-BF532 Blackfin Processor has two data banks (data bank A and data bank B), each of which is divided into four contiguous 4096 byte (4 KB) sub-banks. The ADSP-BF533 Blackfin Processor has two data banks (data bank A and data bank B), each of which is divided into eight contiguous 4096 byte (4 KB) sub-banks. The following table shows the memory ranges for each of the sub-banks.

Data Memory Sub-Bank	Address Range
data bank A, sub-bank 0 (ADSP-BF533)	0xFF80 0000 - 0xFF80 0FFF
data bank A, sub-bank 1 (ADSP-BF533)	0xFF80 1000 - 0xFF80 1FFF
data bank A, sub-bank 2 (ADSP-BF533)	0xFF80 2000 - 0xFF80 2FFF
data bank A, sub-bank 3 (ADSP-BF533)	0xFF80 3000 - 0xFF80 3FFF
data bank A, sub-bank 4 (ADSP-BF531/2/3)	0xFF80 4000 - 0xFF80 4FFF
data bank A, sub-bank 5 (ADSP-BF531/2/3)	0xFF80 5000 - 0xFF80 5FFF
data bank A, sub-bank 6 (ADSP-BF531/2/3)	0xFF80 6000 - 0xFF80 6FFF
data bank A, sub-bank 7 (ADSP-BF531/2/3)	0xFF80 7000 - 0xFF80 7FFF
data bank B, sub-bank 0 (ADSP-BF533)	0xFF90 0000 - 0xFF90 0FFF
data bank B, sub bank 1 (ADSP-BF533)	0xFF90 1000 - 0xFF90 1FFF
data bank B, sub-bank 2 (ADSP-BF533)	0xFF90 2000 - 0xFF90 2FFF
data bank B, sub-bank 3 (ADSP-BF533)	0xFF90 3000 - 0xFF90 3FFF
data bank B, sub-bank 4 (ADSP-BF532/3)	0xFF90 4000 - 0xFF90 4FFF
data bank B, sub bank 5 (ADSP-BF532/3)	0xFF90 5000 - 0xFF90 5FFF
data bank B, sub-bank 6 (ADSP-BF532/3)	0xFF90 6000 - 0xFF90 6FFF
data bank B, sub-bank 7 (ADSP-BF532/3)	0xFF90 7000 - 0xFF90 7FFF

A one cycle stall is incurred during a collision of simultaneous accesses only if the accesses are to the same 32-bit word polarity (address bits 2 match), the same 4 KB sub-bank (address bits 13 and 12 match), the same 16 KB half-bank (address bits 16 match), and the same bank (address bits 21 and 20 match).

Example	<Cycles + Stalls>
R1 = R4.L * R5.H (IS)    R3 = [I0++]    R4 = [I1++]; (I0 is address 0xFF80 1004, I1 is address 0xFF80 1244)	<1+I> (stall is due to a collision in the data bank A, sub-bank 1)

When an address collision occurs, the DAGs get top priority, followed by the store buffer, and finally by the DMA and cache fill/victim accesses.



If the DMA transfer has been blocked for more than 16 sequential core clock cycles, then this DMA transfer will be given top priority to ensure a level of DMA bandwidth.

### Cache Access

This stall can only occur when one or both banks are configured as cache. A one cycle stall is incurred during a collision of simultaneous accesses only if the accesses are to same 4 KB sub-bank (address bits 13 and 12 match), the same 16 KB half-bank (address bits 16 match), and the same bank (address bits 21 and 20 match).

#### Cache Access When One Data Bank is Configured as Cache (1 cycle stall)

When only one data bank is configured as cache, data memory accesses will always be cached to the same data bank. Therefore, it is necessary only to determine the cache sub-bank.

The ADSP-BF532/3 Blackfin Processor devices have four 4 KB cache sub-banks within each data bank (data bank A and data bank B). The ADSP-BF531 Blackfin Processor devices have four 4 KB cache sub-banks within data bank A (data bank B is not available). Because only one data bank is configured as cache, address bits 16, 20, and 21 will match. Therefore, bits 13 and 12 of the data address determine which sub-bank the data will be cached into. In the following example, data bank A is configured as cache:

Data Address[13:12]	Sub-bank Selected (data bank A is Cache)
00	sub-bank 4 (0xFF80 4000 - 0xFF80 4FFF)
01	sub-bank 5 (0xFF80 5000 - 0xFF80 5FFF)
10	sub-bank 6 (0xFF80 6000 - 0xFF80 6FFF)
11	sub-bank 7 (0xFF80 7000 - 0xFF80 7FFF)

If the addresses in a dual memory access (multi-issue) instruction cache to the same sub-bank, a 1 cycle stall will be incurred.

Example	<Cycles + Stalls>
(I0 is address 0x2000 2348, I1 is address 0x2000 2994) R1 = R4.L * R5.H (IS)    <b>R2 = [I0++]    R3 = [I1++]</b> ;	<1+1> (stall is due to a collision in sub-bank 6)

#### Cache Access When Both Data Banks are Configured as Cache (1 cycle stall)

This configuration is possible only on the ADSP-BF532/3 Blackfin Processor devices, since the ADSP-BF531 Blackfin Processor devices do not have data bank B memory. If both data banks are cacheable,

one must also determine which data bank the accesses are cached to (in addition to sub-bank) to determine if there is a stall. This data bank selection depends on the value of the DCBS bit of the DMEM\_CONTROL MMR. If DCBS is 1, address bit 23 is used as data bank select. If DCBS is 0, address bit 14 is used as data bank select. The example in the following table assumes DCBS is 0:

Addr[14:12]	Sub-bank Selected
000	bank A, sub-bank 4 (0xFF80 4000 - 0xFF80 4FFF)
001	bank A, sub-bank 5 (0xFF80 5000 - 0xFF80 5FFF)
010	bank A, sub-bank 6 (0xFF80 6000 - 0xFF80 6FFF)
011	bank A, sub-bank 7 (0xFF80 7000 - 0xFF80 7FFF)
100	bank B, sub-bank 4 (0xFF90 4000 - 0xFF90 4FFF)
101	bank B, sub-bank 5 (0xFF90 5000 - 0xFF90 5FFF)
110	bank B, sub-bank 6 (0xFF90 6000 - 0xFF90 6FFF)
111	bank B, sub-bank 7 (0xFF90 7000 - 0xFF90 7FFF)

If the addresses in a dual memory access (multi-issue) instruction cache to the same data bank and sub-bank, a 1 cycle stall will be incurred.

Example	<Cycles + <i>Stalls</i> >
(I0 is address 0x2000 2348, I1 is address 0x2000 2994) R1 = R4.L * R5.H (IS)    R2 = [I0++]    R3 = [I1++];	<1+1> (stall is due to a collision in data bank A, sub-bank 6)

### Store Buffer Load Collision

This section describes cases where a load access collides with a pending store access in the store buffer. This happens when the load and store are to the same address.

#### Store Data Not Ready

The data portion of a store does not necessarily have to be ready when it is entered into the store buffer. Store data are delayed by three cycles.

Example	< Cycles, <i>Stalls</i> >
W[P0] = R0;	<1 cycle>
R1 = W[P0];	<3 stalls>
[P0] = P3;	<1 cycle>

R1 = [P0];	<3 stalls> <1 cycle>
------------	-------------------------

## Appendix

This appendix is a reference for abbreviations and mnemonics used in the main document. It consists of a glossary, instruction group descriptions, and register group descriptions.

### Glossary

MMR = memory-mapped register

RAW = read-after-write hazard

src = source

### Instruction Groups

All instruction group members conform to naming conventions used in the Blackfin Processor Instruction Set Reference. Descriptions of the instructions can be found in the chapters indicated with parentheses. Note that instruction groups described are not necessarily mutually exclusive; that is, the same instruction can belong to multiple groups.

math ops		
video ops	mult ops	ALU ops
Video Pixel Operations (13)	Vector Multiply (14.12)	Logical Operations (7)
	32-bit Multiply (10.10)	Bit Operations (8)
	Vector MAC (14.3-5)	Shift/Rotate Operations (9)
		Arithmetic Operations except Multiply (10 except 10.10)
		Vector Operations except Multiply/MAC (14 except 14.3-14.5, 14.12)

areg2dreg ops
MAC to Half-Register (14.4)
MAC to Data Register (14.5)
Vector Multiply (14.12)
Round – 12 bit (10.13)
Round – 20 bit (10.14)
Add on Sign (14.1)
Modify - Increment, only this case: [dreg dreg_hi dreg_lo] = (A0 += A1); (10.9)

## Register Groups

allreg (all registers)						
dreg	preg	sysreg (astat and seqreg)		dagreg	statbits	accreg
R0	P0	astat	seqreg	I0	ASTAT [0]: AZ	A0
R1	P1	ASTAT	LC0	I1	ASTAT [1]: AN	A0.x
R2	P2		LT0	I2	ASTAT [2]: AC	A0.w
R3	P3		LB0	I3	ASTAT [3]: AV0	A1
R4	P4		LC1	M0	ASTAT [4]: AV1	A1.x
R5	P5		LT1	M1	ASTAT [5]: CC	A1.w
R6	FP		LB1	M2	ASTAT [6]: AQ	
R7	SP		CYCLES	M3		
			CYCLES2	L0		
			SEQSTAT	L1		
			SYSCFG	L2		
			RETS	L3		
			RETX	B0		
			RETI	B1		
			RETN	B2		
			RETE	B3		

## Document History

Version	Description
September 24, 2003 by Tom L.	Initial release