

Appendix A

Instruction Timing and Restrictions

This appendix describes the various aspects of execution timing analysis for each instruction mnemonic and for various instruction sequences. The section consists of the following tables and information:

- Tables showing how to calculate DSP56300 core instruction timing for each instruction mnemonic (instruction timing)
- Tables showing the number of instruction program words for each instruction mnemonic (instruction program words)
- Description of various sequences that cause timing delays and stalls in the execution (instruction sequence delays)
- Description of various instruction sequences that are forbidden and cause undefined operation (instruction sequence restrictions)

A.1 Overview

The number of oscillator clock cycles per instruction depends on many factors, including the number of words per instruction, the addressing mode, whether the instruction fetch pipeline is full, the number of external bus accesses, cache hit/miss/burst, and the number of wait states inserted into each external access.

Table A-1 lists instruction timing and is based on the assumption that all instruction cycles are counted in clock cycles and the instruction fetch pipeline is full. The following terms are used inside the table:

- **T**: clock cycles for the normal case:
 - All instructions fetched from the internal program memory
 - No interlocks with previous instructions
 - Addressing mode is the Post-Update mode (post-increment, post-decrement and post offset by N) or the No-Update mode.
- **+ pr**: Pre-update specifies clock cycles added for using the pre-update addressing modes (pre-decrement and offset by N addressing modes).

- **+ lab**: Long absolute specifies clock cycles added for using the Long Absolute Address mode.
- **+ lim**: Long immediate specifies clock cycles added for using the long immediate data addressing mode.

Note: A dash under one or more of the columns **pru**, **lab**, or **lim** indicates that this column is not applicable to the corresponding instruction.

Table A-1. Instruction Timing, Word Count, and Encoding

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
ADD	ADD #xxxxxx,D	2	—	—	—
	ADD #xx,D	1	—	—	—
AND	AND #xxxxxx,D	2	—	—	—
	AND #xx,D	1	—	—	—
ANDI	ANDI D	3	—	—	—
ASL	ASL #ii,S2,D	1	—	—	—
	ASL S1, S2,D	1	—	—	—
ASR	ASR S1, S2, D	1	—	—	—
	ASR #ii,S2,D	1	—	—	—
Bcc	Bcc Rn	4	—	—	—
	Bcc xxxx	5	—	—	—
	Bcc xxx	4	—	—	—
BCHG	BCHG #n, [x or y]:aa	2	—	—	—
	BCHG #n, [x or y]:ea	2	1	1	—
	BCHG ##n, [x or y]:pp	2	—	—	—
	BCHG ##n, [x or y]:qq	2	—	—	—
	BCHG #n, D	2	—	—	—
BCLR	BCLR #n, [x or y]:pp	2	—	—	—
	BCLR #n, [x or y]:ea	2	1	1	—
	BCLR #n, [x or y]:aa	2	—	—	—
	BCLR #n, [x or y]: qq	2	—	—	—
	BCLR #n, D	2	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
BRA	BRA (PC + Rn)	4	—	—	—
	BRA (PC + aa)	4	—	—	—
	BRA (PC+aa)	4	—	—	—
BRKcc	BRKcc	5	—	—	—
BRSET	BRSET #bbbb, S:pp, (PC+aaaa)	5	—	—	—
	BRSET #bbbb, S:qq, (PC+aaaa)	5	1	—	—
	BRSET #bbbb, S:ea, (PC+aaaa)	5	—	—	—
	BRSET #bbbb, S:aa, (PC+aaaa)	5	—	—	—
	BRSET #bbbb, DDDDDD, (PC+aaaa)	5	—	—	—
BScC	BScC (PC + Rn)	4	—	—	—
	BScC (PC + aa)	4	—	—	—
BSCLR	BSCLR #bbbb,S:ea,(PC+aaaa)	5	1	—	—
	BSCLR #bbbb,S:aa,(PC+aaaa)	5	—	—	—
	BSCLR #bbbb,S:pp,(PC+aaaa)	5	—	—	—
	BSCLR #bbbb,S:DDDDDD,(PC+aaaa)	5	—	—	—
	BSCLR #bbbb,S:qq,(PC+aaaa)	5	—	—	—
BSET	BSET #n,[x or y]:pp	2	—	—	—
	BSET ##n,[x or y]:ea	2	1	1	—
	BSET ##n,[x or y]:aa	2	—	—	—
	BSET ##n,D	2	—	—	—
	BSET ##n,[x or y]:qq	2	—	—	—
BSR	BSR (PC + Rn)	4	—	—	—
	BSR (PC+aaaa)	5	—	—	—
	BSR (PC + aa)	4	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
BSSET	BSSET #bbbb,S:pp,(PC+aaaa)	5	—	—	—
	BSSET #bbbb,S:ea,(PC+aaaa)	5	1	—	—
	BSSET #bbbb,S:aa,(PC+aaaa)	5	—	—	—
	BSSET #bbbb,S:DDDDDD,(PC+aaaa)	5	—	—	—
	BSSET #bbbb,S:qq,(PC+aaaa)	5	—	—	—
BTST	BTST #n,[x or y]:pp	2	—	—	—
	BTST #n,[x or y]:ea	2	1	1	—
	BTST #n,[x or y]:aa	2	—	—	—
	BTST #n,D	2	—	—	—
	BTST #n,[x or y]:qq	2	—	—	—
CLB	CLB S,D	1	—	—	—
CMP	CMP #iiii,D	2	—	—	—
	CMP #iii,D	1	—	—	—
CMPU	CMPU S1, S2	1	—	—	—
DEBUG/ DEBUGcc	DEBUG	1	—	—	—
	DEBUGcc	5	—	—	—
DEC	DEC D	1	—	—	—
DIV	DIV S, D	1	—	—	—
DMAC	DMAC S1,S2,D (ss,su,uu)	1	—	—	—
DO	DO #xxx,aaaa	5	—	—	—
	DO DDDDDD,aaaa	5	—	—	—
	DO S:<ea>,aaaa	5	1	—	—
	DO S:<aa>,aaaa	5	—	—	—
DO FOREVER	DO FOREVER ,(aaaa)	4	—	—	—
DOR	DOR #xxx,(PX+aaaa)	5	—	—	—
	DOR DDDDDD,(PC+aaaa)	5	—	—	—
	DOR S:ea,(PC+aaaa)	5	1	—	—
	DOR S:aa,(PC+aaaa)	5	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
DOR FOREVER	DOR FOREVER,(PC+aaaa)				
ENDDO	ENDDO	1	—	—	—
EOR	EOR #xx,D	2	—	—	—
	EOR #iii,D	1	—	—	—
EXTRACT	EXTRACT S1,S2,D	1	—	—	—
	EXTRACT #iii,s,D	2	—	—	—
EXTRACTU	EXTRACTU S1,S2,D	1	—	—	—
	EXTRACTU #iii,s,D	2	—	—	—
IFcc	IFcc	1	—	—	—
ILLEGAL	ILLEGAL	5	—	—	—
INC	INC D	1	—	—	—
INSERT	INSERT S1,S2,D	1	—	—	—
	INSERT #iii,qqq,D	2	—	—	—
Jcc	Jcc xxx	4	—	—	—
	Jcc ea	4	0	0	—
JCLR	JCLR #n,[x or y]:ea,xxxx	4	1	—	—
	JCLR #n,[x or y]:pp,xxxx	4	—	—	—
	JCLR #n,[x or y]:aa,xxxx	4	—	—	—
	JCLR #n,S,xxxx	4	—	—	—
	JCLR #n,[x or y]:qq,xxxx	4	—	—	—
JMP	JMP aa	3	—	—	—
	JMP ea	3	1	1	—
JScC	JScC aa	4	—	—	—
	JScC ea	4	0	0	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
JSCLR	JSCLR #n,[x or y]:pp,xxxx	4	—	—	—
	JSCLR #n,[x or y]:ea,xxxx	4	1	—	—
	JSCLR #n,[x or y]:aa,xxxx	4	—	—	—
	JSCLR #n,S,xxxx	4	—	—	—
	JSCLR #n,[x or y]:qq,xxxx	4	—	—	—
JSET	JSET #n,[x or y]:pp,xxxx	4	—	—	—
	JSET #n,[x or y]:ea,xxxx	4	1	—	—
	JSET #n,[x or y]:aa,xxxx	4	—	—	—
	JSET #n,S,xxxx	4	—	—	—
	JSET #n,[x or y]:qq,xxxx	4	—	—	—
JSR	JSR aa	3	—	—	—
	JSR ea	3	1	1	—
JSSET	JSSET #n,[x or y]:pp,xxxx	4	—	—	—
	JSSET #n,[x or y]:ea,xxxx	4	1	—	—
	JSSET #n,[x or y]:aa,xxxx	4	—	—	—
	JSSET #n,S,xxxx	4	—	—	—
	JSSET #n,[x or y]:qq,xxxx	4	—	—	—
LSL	LSL S,D	1	—	—	—
	LSL #ii,D	1	—	—	—
LSR	LSR #ii,D	1	—	—	—
	LSR S,D	1	—	—	—
LRA	LRA (PC + Rn) → 0DDDDD	3	—	—	—
	LRA (PC + aaaa) → 0DDDDD	3	—	—	—
LUA, LEA	LUA ea → 0DDDDD	3	—	—	—
	LUA (Rn + aa) → 01DDDD	3	—	—	—
MACI	MACI ± #xxxxxx,S,D	2	—	—	—
MAC	MAC ± 2**s,QQ,d	1	—	—	—
	MAC S1,S2,D (su,uu)	1	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
MACRI	MACRI \pm #iiii,QQ,D	2	—	—	—
MACR	MACR \pm 2**s,QQ,d	1	—	—	—
MAX	MAX A,B	1	—	—	—
MAXM	MAXM A,B	1	—	—	—
MERGE	MERGE S,D	1	—	—	—
MOVE	No parallel data Move (DALU)	1	—	—	—
	MOVE #xx,D	1	—	—	—
	MOVE S,D	1	—	—	—
	MOVE ea (U move, address register update)	1	—	—	—
	MOVE [x or y]:ea,D	1	1	1	1
	MOVE S,[x or y]:ea	1	1	1	1
	MOVE #xxxxx,D	1	1	1	1
	MOVE [x or y]:aa,D	1	—	—	—
	MOVE [x or y]aa	2	—	—	—
	MOVE [x or y]:(Rn+xxx),D	2	—	—	—
	MOVE S,[x or y]:(Rn+xxx)	2	—	—	—
	MOVE [x or y]:(Rn+xxxx),D	3	—	—	—
	MOVE S,[x or y]:(Rn+xxxx)	3	—	—	—
	MOVE X:ea,D1,S2,D2	1	1	1	1
	MOVE S1,S:ea S2,D2	1	1	1	1
	MOVE #xxxxx,D1 S2,D2	1	1	1	1
	MOVE S1,D1 Y:ea,D2	1	1	1	1
	MOVE S1,D1 S2,Y:ea	1	1	1	1
	MOVE S1,D1 #xxxxx,D2	1	1	1	1
	MOVE A,X:ea X0,A	1	1	—	—
MOVE B,X:ea X0,B	1	1	—	—	
	MOVE Y0 A,A,Y:ea	1	1	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
MOVE cont.	MOVE Y0 B,B,Y:ea	1	1	—	—
	MOVE L:ea,D MOVE S,L:ea	1	1	1	—
	MOVE X:eax,D1 Y:ey,D2	1	—	—	—
	MOVE X:eax,D1 S2,Y:ey	1	—	—	—
	MOVE S1,X:eax Y:ey,D2	1	—	—	—
	MOVE S1,X:eax S2,Y:ey	1	—	—	—
MOVEC	MOVEC #xx,D1	1	—	—	—
	MOVEC [x or y]:ea,D1	1	1	1	1
	MOVEC S1,[x or y]:ea	1	1	1	1
	MOVEC #xxxxxx,D1	1	1	1	1
	MOVEC [x or y]:aa,D1	1	—	—	—
	MOVEC S1,[x or y]:aa	1	—	—	—
	MOVEC S1,D2	1	—	—	—
	MOVEC S2,D1	1	—	—	—
MOVEM	MOVEM S,P:ea	6	1	1	—
	MOVEM P:ea,D	6	1	1	—
	MOVEM S,P:aa	6	—	—	—
	MOVEM P:aa,D	6	—	—	—
MOVEP	MOVEP [x or y]:pp,[x or y]:ea	2	1	1	0
	MOVEP [x or y]:ea,[x or y]:pp	2	1	1	0
	MOVEP [x or y]:qq,[x or y]:ea	2	1	1	0
	MOVEP [x or y]:ea,[x or y]:qq	2	1	1	0
	MOVEP [x or y]:pp,P:ea	6	1	1	—
	MOVEP P:ea,[x or y]:pp	6	1	1	—
	MOVEP [x or y]:qq,P:ea	6	1	1	—
	MOVEP P:ea,[x or y]:qq	6	1	1	—
	MOVEP [x or y]:pp,D	1	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
MOVEP cont.	MOVEP S,[x or y]:pp	1	—	—	—
	MOVEP [x or y]:qq,D	1	—	—	—
	MOVEP S,[x or y]:qq	1	—	—	—
MPY	MPY S1,S2,D (su,uu)	1	—	—	—
	MPY $\pm 2^{**}s$,QQ,d	1	—	—	—
MPYI	MPYI (I)#xxxxxx,S,D	2	—	—	—
MPYR	MPYR $\pm 2^{**}s$,QQ,d	1	—	—	—
MPYRI	MPYRI $\pm \#iiii$,QQ,D	2	—	—	—
NOP	NOP	1	—	—	—
NORM	NORM	5	—	—	—
NORMF	NORMF S,D	1	—	—	—
OR	OR #xx,D	2	—	—	—
	OR #iii,D	1	—	—	—
ORI	OR(I) D	3	—	—	—
PFLUSH	PFLUSH	1	—	—	—
PFLUSHUN	PFLUSHUN	1	—	—	—
PFREE	PFREE	1	—	—	—
PLOCK	PLOCK ea	2	1	1	—
PLOCKR	PLOCKR (PC+aaaa)	4	—	—	—
PUNLOCK	PUNLOCK ea	2	1	1	—
PUNLOCKR	PUNLOCKR (PC+aaaa)	4	—	—	—
REP	REP #xxx	5	—	—	—
	REP S	5	—	—	—
	REP [x or y]:ea	5	1	—	—
	REP [x or y]:aa	5	—	—	—
RESET	RESET	7	—	—	—
RTI/RTS	RTI	3	—	—	—
	RTS	3	—	—	—

Table A-1. Instruction Timing, Word Count, and Encoding (Continued)

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
STOP	STOP	10	—	—	—
SUB	SUB #xx,D	2	—	—	—
	SUB #iii,D	1	—	—	—
Tcc	Tcc S1,D1,S2,D2	1	—	—	—
	Tcc S1,D1	1	—	—	—
	Tcc S2,D2	1	—	—	—
TRAP/ TRAPcc	TRAP	9	—	—	—
	TRAPcc	9	—	—	—
VSL	VSL S,i,L:ea	1	1	1	—
WAIT	WAIT	10	—	—	—

A.2 Instruction Sequence Delays

Because of pipelining in the DSP56300 core, certain instruction sequences can cause a delay in the execution of instructions. Most of these sequences are caused by a source-destination conflict or by the need to access the external bus. There are six types of sequence delays:

- External bus wait states
- Instruction fetch delays
- Data ALU interlocks
- Address register interlocks
- Stack extension delays
- Pipeline interlocks

A.2.1 External Bus Wait States

An external bus wait state is caused by an instruction accessing the external bus for data read or write. The execution time of the instruction is increased by the number of clock cycles equal to the number of wait states programmed for that external data access. The exact number of wait states depends on the type of memory accessed.

A.2.2 Instruction Fetch Delays

At an external instruction fetch, the effective number of stall states in the pipeline is the number specified in the Bus Control Register (BCR).

A.2.3 Data ALU Interlock

A Data ALU interlock is caused by one of the following sequences:

- *Arithmetic stall*: Occurs when an instruction uses one of the Data ALU registers (A0, A1, A2, B0, B1, or B2) or accumulators (A or B) as a source register for the move portion of the instruction when the preceding instruction is an arithmetic instruction¹ that uses the same accumulator as its destination. Delays execution of the initiating instruction by one clock cycle.
- *Transfer stall*: Occurs when an instruction uses one of the Data ALU registers (A0, A1, A2, B0, B1, or B2) or accumulators (A or B) as a source register for the move portion of the instruction when the preceding instruction uses the corresponding accumulator or one of the Data ALU registers that comprise the accumulator as its destination register in the move portion of that instruction. Delays execution of the initiating instruction by one instruction cycle.
- *Status stall*: Occurs when an instruction reads the contents of the Status Register (SR) for either a move operation or bit testing and the preceding or the second preceding instruction is an arithmetic instruction. Delays execution of the initiating instruction by two instruction cycles for a move operation or one instruction cycle for bit testing.

A.2.4 Address Register Interlocks

An address register interlock is caused by one of the following sequences:

- *Conditional Transfer Interlock*: Occurs when a Transfer On-Condition (Tcc) instruction is followed by an instruction that explicitly specifies one of the address generation registers (R0–R7) as its source operand. Delays execution of the second instruction by one instruction cycle.
- *Address Generation Interlock*: Occurs when the move portion of an instruction uses one of the AGU registers R0–R7 for address generation or for address calculation, while one of the three preceding instruction cycles uses one of the register set (Ri, Ni or Mi) members as a destination register in its move portion. Consider **Example A-1**.

1. An arithmetic instruction uses the internal Data ALU data paths.

Example A-1. Address Generation Interlock

```

I1 MOVE #$addr,R0
I2 NOP
I3 NOP
I4 NOP
I5 MOVE #$offset,N0
I6 MOVE X:(R0)+,Y1

```

In this example, instruction I6 causes an address generation interlock because it uses R0 as the source for address generation on the X Address Bus while the preceding instruction, I5, uses N0 as its destination.

Three types of address generation interlock exist: Type0, Type1, and Type2. These types depend on the clock cycle distance between the instruction causing the interlock and the preceding instruction that uses the AGU register as a destination. **Figure A-1** gives an example of each interlock type:

Type0 Interlock	Type1 Interlock	Type2 Interlock
I1 MOVE #\$addr,R0	I1 MOVE #\$addr,R0	I1 MOVE #\$addr,R0
I2 MOVE X:(R0)+,Y1	I2 CLR A	I2 CLR A
	I3 MOVE X:(R0)+,Y1	I3 INC B
		I4 MOVE X:(R0)+,Y1
Three NOP instructions are inserted	Two NOP instructions are inserted	One NOP instruction is inserted

Figure A-1. Types of Address Generation Interlock

When a Type0 address generation interlock is detected (during the decoding of I2 in the example), three NOP clock cycles are automatically inserted before execution of the instruction starts. When a Type1 interlock is detected (during the decoding of I3 in the example), two NOP clock cycles are automatically inserted before the execution of the instruction starts. When a Type2 interlock is detected (during the decoding of I4 in the example), one NOP clock cycle is inserted before execution of the instruction starts.

Note: Only clock cycles are counted to determine when interlock cycles should be inserted.

When an instruction using one of the AGU registers as an address generation enters the decoding stage of the DSP56300 core, the distance from that instruction to the preceding instruction using the register as destination is measured in clock cycles to determine the existence and type of address generation interlock. Once an address generation interlock is detected, the appropriate number of NOP clock cycles is inserted. The following instructions take these additional cycles into account for detecting a possible new address generation interlock. **Example A-2** demonstrates this feature.

Example A-2. Detection of Address Generation Interlock

```
I1 MOVE #addr,R0
I2 CLR A
I3 MOVE X:(R0)+,Y1
I4 MOVE X:(R0)+,Y0
```

In this example, a Type1 interlock is detected during the decoding phase of I3 and two NOP cycles are inserted before that instruction executes. During the decoding of I4, no address generation interlock is detected, so no NOP cycles are inserted. However, if I3 were an instruction that did not use R0, a Type2 address generation interlock would be detected during the decoding phase of I4, and one NOP cycle would be inserted before the instruction executes.

A.2.5 Stack Extension Delays

Some instructions access the System Stack (SS) as part of their normal activity. When the SS is either completely full or empty, the special stack extension mechanism is engaged and the access completes only after an access to data memory is automatically performed. This delays the decoding and the execution phases of that instruction. A stack-full or a stack-empty state is defined by the contents of the Stack Counter (SC) register. When the stack counter equals 14, the on-chip hardware stack contains fourteen words (a stack word is a 48-bit long word combined from the low and the high portions of the stack). The stack is declared as stack-full, and any additional push operation activates the stack extension mechanism. When the stack counter equals 2, the on-chip hardware stack contains only two words. The stack is declared as stack-empty, and any additional pop operations activate the stack extension mechanism. The instructions/cases listed in **Table A-2** cause an access to the system stack and may engage the stack extension mechanism.

Table A-2. Instructions That Access the System Stack

Instruction	Description
JSR, Jcc	All the conditional and unconditional Jump to Subroutine instructions (e.g., JSR, JSSET, and so on). These instructions perform a stack PUSH operation that stores the PC and the SR on top of the stack for the use of the 'Return from Subroutine' instruction that terminates the subroutine execution.
RET	The two Return from Subroutine instructions, RTS and RTI. These instructions perform a stack POP operations that pulls the PC and (optionally) the SR out from the top of stack in order to return to the calling procedure and restore the status bits and loop flag state.
END-OF-DO	A condition of the hardware inside the Program Control Unit. This hardware detects a fetch from the last address of a loop initiated when the Loop Counter equals 1. This condition defines the end of the loop, thus performing a stack POP operation. This POP operation restores the loop flag, purges the top of stack (PC:SR), and pulls LA and LC from the new top of stack.
LOOP	All the hardware-loop initiating instructions (e.g., DO) with all their options. These instructions perform a stack double-PUSH operation that first stores the previous values of LA and LC on top of the stack. Then the DO instruction stores the contents of SR and PC on the new top of stack. This PC value is used every loop iteration to return to the top of loop location and start fetch from there. DO performs two accesses to the stack instead of the normal single access done by most stack operations.
ENDDO	A special instruction that forces an end-of-do condition during a hardware loop. Like END-OF-DO, ENDDO performs two accesses to the stack instead of the normal single access done by most stack operations.
SSHWR	All the explicit stack PUSH instructions that use SSH as their destination (e.g., the MOVE R0,SSH instruction).
SSHRD	All the explicit stack POP instructions that use SSH as their source (e.g., the MOVE SSH,Y1 instruction).

Table A-3 shows how many clock cycles are added in the various instructions/cases described.

Table A-3. Stack Extension Delays

CASE	Stack Full Condition (+ clock cycles)	Stack Empty Condition (+ clock cycles)
JSR, Jcc	2	—
RET	—	3
END-OF-DO	—	5
DO	4	—
ENDDO	—	5
SSHWR	2	—
SSHRD	—	3

A.2.6 Program Flow Control Delays

When flow-control instructions execute, some boundary cases exist and introduce interlocks into the program flow. These interlocks lengthen the decoding phase of the instructions, thus delaying execution. The following sequences represent unusual operations that will probably never be used. The detection of these cases and the generation of interlocks is done to maintain object code compatibility between the DSP56300 core and the 56000 family of DSPs. The following terms are used in this discussion:

- I1: An address of an instruction, where I2, I3, and I4 indicate the next instructions in the program flow
- MOVE: any type of MOVE, MOVEM, MOVEP, MOVEC, BSET, BCHG, BCLR, and BTST
- LA: the last address of a DO LOOP
- (LA – 1): the address of an instruction word located at LA – 1
- CR: Control Register, every one of the registers LA, LC, SR, SP, SSH, SSL, and OMR

A.2.6.1 JMP to LA or to LA – 1

When I1 is any type of JMP with its target address equal to LA, the decoding phase of the instruction following the instruction at LA is delayed by 2 clock cycles. When I1 is any type of JMP with its target address equal to LA – 1, the decoding phase of the instruction following the instruction at LA is delayed by one clock cycle.

A.2.6.2 RTI to LA or to LA – 1

When I1 is an RTI instruction whose return address is LA, the decoding phase of the instruction following the instruction at LA is delayed by 2 clock cycles. When I1 is an RTI instruction whose return address is LA – 1, the decoding phase of the instruction following the instruction at LA is delayed by one clock cycle.

A.2.6.3 Conditional Instructions

When I1 is a conditional change of flow instruction (such as Jcc) and the condition is false, the decoding phase of I2 is delayed by one clock cycle.

A.2.6.4 Interrupt Abort

When I1 is an instruction with a decoding phase that is longer than one cycle, it may be aborted by the Interrupt Control Unit. In this case, a 1 clock cycle “hole” is inserted into the pipeline, after which the instruction at the interrupt vector is decoded.

A.2.6.5 Degenerated DO loop

When I1 is a DO loop but the loop contains only one instruction, the decoding phase of I1 is lengthened by one clock cycle.

A.2.6.6 Annulled REP and DO

If the repeat count of a REP instruction is zero, the decoding phase of the REP instruction is lengthened by one clock cycle. If the repeat count of a DO instruction is zero, the decoding phase of the DO instruction is lengthened by three clock cycles.

A.3 Instruction Sequence Restrictions

Because of the pipelining in the DSP56300 core central processor, certain instruction sequences are forbidden. Use of these sequences causes undefined operation. Most of these restricted sequences cause contention for an internal resource, such as the Stack Register. The DSP Assembler flags these as assembly errors. The following terms are used in this discussion:

- **MOVE**: any type of MOVE, MOVEM, MOVEP, MOVEC
- **MOVEM**: any type of MOVE to/from the Program space
- **LA**: the last address of a DO LOOP
- **Two-words <inst>**: a double-word instruction in which the second word is used as an immediate data or absolute address
- **Single-word <inst>**: an instruction with an addressing mode that does not need a second word extension

A.3.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed for an instruction sequence similar to one of the following sequences.

- At LA – 5: The following instructions should not start at address LA – 5:
 - Single-word or two-word MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- At LA – 4: The following instructions should not start at address LA – 4:
 - Single-word or two-word MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
- At LA – 3: The following instructions should not start at address LA – 3:
 - BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - MOVE from SSH, SSL
 - Two-word JMP, Jcc, JSR, JScC
 - JSET, JCLR, JSSET, JSCLR
 - Two-word MOVEM
- At LA – 2: The following instructions should not start at address LA – 2:
 - DO, DOR, DO FOREVER
 - MOVE to/from {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - BCHG, BSET, BCLR, BTST on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - JMP, Jcc, JSR, JScC, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BScC
 - MOVEM
 - ANDI, ORI on MR
 - BRKcc, ENDDO, REP
 - STOP, WAIT, DEBUG, DEBUGcc, TRAP, TRAPcc, ILLEGAL
- At LA – 1: The following instructions should not start at address LA – 1:
 - DO, DOR, DO FOREVER
 - MOVE to/from {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - BCHG, BSET, BCLR, BTST on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - JMP, Jcc, JSR, JScC, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BScC
 - MOVEM

- ANDI, ORI on MR
- BRK_{cc}, ENDDO, REP
- STOP, WAIT, DEBUG, DEBUG_{cc}, TRAP, TRAP_{cc}, ILLEGAL

Note: A one-word conditional branch instruction at LA-1 is not allowed.

When two consecutive LAs have a conditional branch instruction at LA-1 of the internal loop, the device does not operate properly. For example, the following sequence may generate incorrect results:

```

        DO #5, LABEL1
        NOP
        DO #4, LABEL2
        NOP
        MOVE (R0) +
        BSCC _DEST          ; conditional branch at LA-1 of internal loop
        NOP                ; internal LA
LABEL2
        NOP                ; external LA
LABEL1
        NOP
        NOP
_DEST  NOP
        NOP
        RTS

```

Workaround: Put an additional NOP between LABEL2 and LABEL1.

- At LA: The following instructions should not start at address LA:
 - Any two-word instruction
 - MOVE to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - MOVE from SSH, SSL
 - BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}
 - BTST on SSH
 - JMP, JSR, BRA, BSR, J_{cc}, JS_{cc}, B_{cc}, BS_{cc}
 - MOVE to/from Program space {MOVEM, MOVEP (only the P space options)}.
 - RESET
 - RTI, RTS
 - ANDI, ORI on MR
 - BRK_{cc}, ENDDO, REP
 - STOP, WAIT, DEBUG, DEBUG_{cc}, TRAP, TRAP_{cc}, ILLEGAL

A.3.2 General DO Restrictions

The general restrictions on DO instructions are as follows:

- A DO loop should be initialized and aborted using only the following instructions: DO, DOR, DO FOREVER, ENDDO, and BRKcc.
- The LF and the FV bits in the Status Register (SR) should not be explicitly changed using the MOVE, BCHG, BSET, BCLR, ANDI, or ORI instructions.
- Proper DO loop operation is not guaranteed if an instruction sequence similar to one of the following sequences is used.
 - SSH cannot be used as the source for the Loop-Count for a DO, DOR, or a DO FOREVER instruction.
 - The following instructions should not appear within four words before a DO, DOR, or DO FOREVER:
 - BCHG, BCLR, BSET, MOVE on/to SSH,SSL
 - BCHG, BCLR, BSET, MOVE on/to SP, SC
 - The following instructions should not appear immediately before a DO, DOR, or DO FOREVER:
 - MOVE from SSH
 - BTST on SSH
 - BCHG, BCLR, BSET, MOVE to/on {LA, LC, SP, SC, SSH, SSL}
 - JSR, JScc, JSSET, JSCLR to LA whenever LF is set
 - BSR, BSc, to LA whenever LF is set
 - The following instructions should not appear in a DO, DOR, or DO FOREVER loop:
 - {JMP, Jcc, JSR, JScc, JSET, JCLR, JSSET, JSCLR, BRA, Bcc, BSR, BSc}

When Stack Extension mode is enabled, use of the BRKcc or ENDDO instructions inside DO loops may cause an improper operation. If the loop is not nested and has no nested loop inside it, this restriction is relevant only if LA or LC values are in use outside the loop. If Stack Extension is used, emulate the BRKcc or ENDDO as shown in the following examples in which there is a split between two cases, finite DO loops and DO FOREVER loops.

Example A-3. Finite DO Loops

BRKcc

Original code:

```

do #N,label1
    .....
    .....
        do #M,label2
            .....
            .....
            BRKcc
            .....
            .....
label2
    .....
    .....
label1

```

Will be replaced by:

```

do #N, label1
    .....
    .....
        do #M, label2
            .....
            .....
            Jcc    fix_brk_routine
            .....
            .....

nop_before_label2
    nop        ; This instruction must be NOP.
label2
    .....
    .....
label1
....
....

fix_brk_routine
    move #1,lc
    jmp  nop_before_label2

ENDDO
-----

```

Original code:

```

do #M,label1
    .....
    .....
        do #N,label2
            .....
            .....
            ENDDO
            .....
            .....

```

```

label2
    .....
    .....
label1
Will be replaced by:
    do #M, label1
    .....
    .....
        do #N, label2
        .....
        .....
        JMP      fix_enddo_routine

nop_after_jump
    NOP ; This instruction must be NOP.
    .....
    .....

label2
    .....
    .....

label1
    ....
    ....

fix_enddo_routine
    move #1,lc
    move #nop_after_jump,la
    jmp  nop_after_jump

```

Example A-4. DO FOREVER Loops

```

BRKcc
-----
Original code:
    do #M,label1
    .....
    .....
        do forever,label2
        .....
        .....
        BRKcc
        .....
        .....

label2
    .....
    .....

label1
Will be replaced by:
    do #M,label1
    .....
    .....
        do forever,label2

```

```

        .....
        .....
        JScc    fix_brk_forever_routine  ; <---
note: JScc and not Jcc
        .....
        .....

nop_before_label2
        nop    ; This instruction must be NOP.
label2
        .....
        .....
label1
.....
.....

fix_brk_forever_routine
        move ssh,x:<..>  ; <..> is some reserved not used
address (for temporary data)
        move #nop_before_label2,ssh
        bclr #16,ssl    ;
        move #1,lc
        rti            ; <----- note: "rti" and not "rts" !

ENDDO
-----
Original code:
do #M,label1
.....
.....

do forever,label2
        .....
        .....
        ENDDO
        .....
        .....

label2
        .....
        .....

label1

Will be replaced by:
        do #M,label1
        .....
        .....
        do forever,label2
        .....
        .....
        JSR    fix_enddo_routine    ; <--- note:
JSR and not JMP
nop_after_jump
        NOP    ; This instruction should be NOP
        .....
        .....

```

```

label2
    .....
    .....
label1
    ....
    ....
fix_enddo_routine
    nop
    move #1,lc
    bclr #16,ssl
    move #nop_after_jump,la
    rti          ; <--- note: "rti" and not "rts"

```

A.3.3 ENDDO Restrictions

The instructions in the following list should not appear within four words before an ENDDO instruction:

- BCHG, BCLR, BSET, MOVE on/to SSH,SSL
- BCHG, BCLR, BSET, MOVE on/to SP, SC

The instructions in the following list should not appear immediately before an ENDDO instruction:

- ANDI, ORI on MR
- MOVE from SSH
- BTST on SSH
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

A.3.4 BRKcc Restrictions

The instructions in the following list should not appear immediately before a BRKcc instruction:

- Every arithmetic instruction
- IFcc, Tcc
- BCHG, BCLR, BSET, MOVE on/to {LA, LC, SP, SC, SSH, SSL, SZ, VBA, OMR}

A.3.5 RTI and RTS Restrictions

The instructions in the following list should not appear within four words before an RTI or RTS instruction:

- BCHG, BCLR, BSET, MOVE on/to SSH,SSL
- BCHG, BCLR, BSET, MOVE on/to SP, SC

The instructions in the following list should not appear immediately before an RTI instruction:

- MOVE, BCHG, BCLR, BSET on {SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH
- ANDI, ORI on {MR, CCR}
- ENDDO

The instructions in the following list should not appear immediately before an RTS instruction:

- MOVE, BCHG, BCLR, BSET on {SSH, SSL, SP, SC}
- MOVE, BTST from/on SSH
- ENDDO

A.3.6 SR Manipulation Restrictions

Changing values of bits in the Status Register (SR) should not be done explicitly using one of the MOVE, BCHG, BSET, BCLR instructions, but only using the ANDI or ORI instructions with the appropriate 8-bit portion on the SR (MR, EMR, CCR).

A.3.7 SP/SC and SSH/SSL Manipulation Restrictions

The instructions in List A should not be executed within four instructions before executing any of the instructions in List B.

List A

- MOVE to (SP, SC)
- BCHG, BSET, BCLR on (SP, SC)

List B

- MOVE to/from {SSH,SSL}

- BTST, BCHG, BSET, BCLR on {SSH,SSL}
- JSET, JCLR, JSSET, JSCLR on {SSH,SSL}

A.3.8 Fast Interrupt Routines

The following instructions cannot be used in a fast interrupt routine:

- DO, DO FOREVER, REP
- ENDDO, BRKcc
- RTI, RTS
- STOP, WAIT
- TRAP, TRAPcc
- ANDI, ORI on {MR, CCR}
- MOVE from SSH
- BTST on SSH
- MOVE to {LA, LC, SP, SC, SSH, SSL}
- BCHG, BSET, BCLR on {LA, LC, SP, SC, SSH, SSL}

A.3.9 REP Restrictions

The REP instruction can repeat any single-word instruction except the REP instruction itself and any instruction that changes program flow. The following instructions are not allowed to follow a REP instruction (cannot be repeated):

- REP, DO, DO FOREVER
- ENDDO, BRKcc
- JMP, Jcc, JCLR, JSET
- JSR, JScc, JSCLR, JSSET
- BRA, Bcc
- BSR, BSc
- RTS, RTI
- TRAP, TRAPcc
- WAIT, STOP

A.3.10 Stack Extension Restrictions

The following instructions, related to the operation of the on-chip hardware stack extension, cannot be used whenever the stack extension is enabled:

- MOVE to EP
- BCHG, BSET, BCLR on EP
- MOVE to SC with a value greater than 15

The following instructions, related to the operation of the on-chip hardware stack extension, cannot be placed in the stack error vector locations whenever the stack extension is enabled:

- JSR, JScc, JSCLR, JSSET
- BSR, BSc

A.3.11 Stack Extension Enable Restrictions

When stack extension is enabled, the read result from stack may be improper if two previous executed instructions cause sequential read and write operations with SSH. Two cases are possible:

- Case 1:
 - For the first executed instruction: move from SSH or bit manipulation on SSH (i.e., JCLR, BRCLR, JSET, BRSET, BTST, BSSET, JSSET, BSCLR, JSCLR).
 - For the second executed instruction: move to SSH or bit manipulation on SSH (i.e., JSR, BSR, JScc, BSc).
 - For the third executed instruction: an SSL or SSH read from the stack result may be improper. Move from SSH or SSL or bit manipulation on SSH or SSL (i.e., BSET, BCLR, BCHG, JCLR, BRCLR, JSET, BRSET, BTST, BSSET, JSSET, BSCLR, JSCLR).

Workaround: Add two NOP instructions before the third executed instruction.

- Case 2:
 - For the first executed instruction: bit manipulation on SSH (i.e., BSET, BCLR, BCJG).
 - For the second executed instruction: an SSL or SSH read from the stack result may be improper. Move from SSH or SSL or bit manipulation on SSH or SSL

(i.e., BSET, BCLR, BCHG, JCLR, BRCLR, JSET, BRSET, BTST, BSSET, JSSET, BSCLR, JSCLR).

Workaround: Add two NOP instructions before the second executed instruction.

A.4 Peripheral Pipeline Restrictions

The DSP56300 core is based on a highly optimized pipeline engine. Despite the relatively deep pipeline (seven stages), the latency effects normally associated with long pipelines are minimal because most of these effects are transparent to the user. Such design techniques as forwarding and interlocking alleviate the need for a thorough knowledge of the machine's pipeline in order to avoid data dependencies. This knowledge becomes necessary only when you are further optimizing the code. The assembler detects when transparency does not exist (e.g., pointer restrictions) and generates an appropriate warning message. However, the pipeline is exposed to the user during peripheral activity. This section describes the cases in which you must take precautions in order to achieve the desired functionality.

A.4.1 Polling a Peripheral Device for Write

When data is written to a peripheral device, there is a two-cycle pipeline delay until any status bits affected by this operation are updated. For example, you operate a peripheral port using the polling technique. You look for the Data Empty flag to be set, and when it is set, you write new data to the Transmit Data Register. If you try to read the status bit within the next two cycles, the flag is mistakenly read as set due to the pipeline delays associated with the peripheral operations. Therefore, if you assume that the Transmit Data Register is empty and write a new data word, this data word overwrites the previously written data. To achieve the correct functionality, you must wait at least two cycles before attempting to read the Status Register after a write to the Transmit Data register. **Example A-5** shows the correct sequence for transmit operations.

Example A-5. Providing a Wait for Proper Data Writes

```

send
    movep  x:(r0)+,x:STX      ; send new data
    nop                    ; pipeline delay
    nop                    ; pipeline delay

poll
    jclr   #TDE,x:SCSR,poll  ; wait for data empty
    jmp    send              ; go to send data

```

A.4.2 Writing to a Read-Only Register

Writing to a read-only register is an operation that normally has no effect, but if a read operation from the same register is attempted within the following two cycles, the value of the read data is the value of the data that was written instead of the unchanged data of the read-only register. To ensure that the correct data is read after the write operation, you must wait at least two cycles before performing the read.

A.4.3 XY Memory Data Move

An XY memory data move does not work properly in either of the following situations:

- The X-memory move destination is internal I/O and the Y-memory move source is a register used as destination in the previous adjacent move from non Y-memory.
- The Y-memory move destination is a register used as source in the next adjacent move to non Y-memory.

Here are examples cases (where x:(r1) is a peripheral):

Example 1:

```
move  #12,y0
move  x0,x:(r7) y0,y:(r3) (while x:(r7) is a peripheral).
```

Example 2:

```
mac    x1,y0,a x1,x:(r1)+      y:(r6)+,y0
move   y0,y1
```

To address this problem, use one of the following alternatives:

- Separate these two consecutive moves by any other instruction.
- Split the XY Data Move to two moves.

A.5 Sixteen-Bit Compatibility Mode Restrictions

When there is a return from a long interrupt (by the RTI instruction), and the first instruction after the RTI is a move to a DALU register (A, B, X, Y), the move may not be correct if the 16-bit arithmetic mode bit (bit 17 of SR) is changed due to restoring SR after RTI. To address this problem, replace the RTI with the following sequence:

```
movec  ssl,sr
nop
rti
```