Appendix B INSTRUCTION EXECUTION TIMING

B-1 INTRODUCTION

This section 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:

1. Tables showing how to calculate DSP56300 Core instruction timing for each instruction mnemonic (instruction timing)
2. Tables showing the number of instruction program words for each instruction mnemonic (instruction program words).
3. Description of various sequences that cause timing delays and stalls in the execution (instruction sequence delays).
4. Description of various instruction sequences that are forbidden and will cause undefined operation (instruction sequence restrictions).

B-2 INSTRUCTION TIMING

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 or not, the number of external bus accesses, cache hit/miss/burst, and the number of wait states inserted in each external access.

The timing table is based on the following assumptions:

1. All instruction cycles are counted in **clock cycles**.
2. The instruction fetch pipeline is **full**.

The following terms are used inside the table:

1. **T** - clock cycles for the normal case:
   - All the instructions are fetched from the Instruction Cache (hit) or from the internal program memory.
   - All accesses to data memory are to the internal X and/or Y internal ROMs or RAMs.
   - The previous instructions access internal data memory only.
   - No interlocks with previous instructions.
   - The stack extension mode is disabled.
   - Addressing mode is the Post-Update mode (post increment, post

decrement and post offset by N) or the No-Update mode.

2. **+ pru** - **PR**e **U**pdate - clocks cycles added for using the pre-update addressing modes (pre decrement & offset by N addressing modes).
3. **+ lab** - **L**ong **AB**solute - clock cycles added for using the long absolute address mode.
4. **+ lim** - **L**ong **IM**mediate - clock cycles added for using the long immediate data addressing mode.

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

Table B-1. Instruction Timing, Word Count and encoding

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
ADD	**ADD #iiiiii,D**	2	-	-	-
	ADD #iii,D	1	-	-	-
AND	**AND #iiiiii,D**	2	-	-	-
	AND #iii,D	1	-	-	-
ANDI	**ANDI EE**	3	-	-	-
ASL	**ASL #ii,S,D**	1	-	-	-
	ASL sss,S,D	1	-	-	-
ASR	**ASR sss,S,D**	1	-	-	-
	ASR #ii,S,D	1	-	-	-
Bcc	**Bcc (PC+Rn)**	4	-	-	-
	Bcc (PC+aaaa)	5	-	-	-
	Bcc (PC+aa)	4	-	-	-
BCHG	**BCHG #bbbbb,S:<aa>**	2	-	-	-
	BCHG #bbbbb,S:<ea>	2	1	1	-
	BCHG #bbbbb,S:<pp>	2	-	-	-
	BCHG #bbbbb,S:<qq>	2	-	-	-
	BCHG #bbbbb,DDDDDD	2	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
BCLR	BCLR #bbbbb,S:<pp>	2	-	-	-
	BCLR #bbbbb,S:<ea>	2	1	1	-
	BCLR #bbbbb,S:<aa>	2	-	-	-
	BCLR #bbbbb,S:<qq>	2	-	-	-
	BCLR #bbbbb,DDDDDD	2	-	-	-
BRA	BRA (PC+Rn)	4	-	-	-
	BRA (PC+aaaa)	5	-	-	-
	BRA (PC+aa)	4	-	-	-
BRCLR	BRCLR #bbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BRCLR #bbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BRCLR #bbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
BRKcc	BRKcc	5	-	-	-
BRSET	BRSET #bbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BRSET #bbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BRSET #bbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BRSET #bbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BRSET #bbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BScc	BScc (PC+aaaa)	5	-	-	-
	BScc (PC+Rn)	4	-	-	-
	BScc (PC+aa)	4	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
BSCLR	BSCLR #bbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BSCLR #bbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BSCLR #bbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BSET	BSET #bbbbb,S:<pp>	2	-	-	-
	BSET #bbbbb,S:<ea>	2	1	1	-
	BSET #bbbbb,S:<aa>	2	-	-	-
	BSET #bbbbb,DDDDDD	2	-	-	-
	BSET #bbbbb,S:<qq>	2	-	-	-
BSR	BSR (PC+Rn)	4	-	-	-
	BSR (PC+aaaa)	5	-	-	-
	BSR (PC+aa)	4	-	-	-
BSSET	BSSET #bbbbb,S:<pp>,(PC+aaaa)	5	-	-	-
	BSSET #bbbbb,S:<ea>,(PC+aaaa)	5	1	-	-
	BSSET #bbbbb,S:<aa>,(PC+aaaa)	5	-	-	-
	BSSET #bbbbb,DDDDDD,(PC+aaaa)	5	-	-	-
	BSSET #bbbbb,S:<qq>,(PC+aaaa)	5	-	-	-
BTST	BTST #bbbbb,S:<pp>	2	-	-	-
	BTST #bbbbb,S:<ea>	2	1	1	-
	BTST #bbbbb,S:<aa>	2	-	-	-
	BTST #bbbbb,DDDDDD	2	-	-	-
	BTST #bbbbb,S:<qq>	2	-	-	-
CLB	CLB S,D	1	-	-	-
CMP	CMP #iiiiii,D	2	-	-	-
	CMP #iii,D	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+ pru	+ lab	+ lim
CMPU	CMPU ggg,D	1	-	-	-
DEBUG/DEBUGcc	DEBUG	1	-	-	-
	DEBUGcc	5	-	-	-
DEC	DEC	1	-	-	-
DIV	DIV	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,(PC+aaaa)	5	-	-	-
	DOR DDDDDD,(PC+aaaa)	5	-	-	-
	DOR S:<ea>,(PC+aaaa)	5	1	-	-
	DOR S:<aa>,(PC+aaaa)	5	-	-	-
DOR FOREVER	DOR FOREVER,(PC+aaaa)	4	-	-	-
ENDDO	ENDDO	1	-	-	-
EOR	EOR #iiiiii,D	2	-	-	-
	EOR #iii,D	1	-	-	-
EXTRACT	EXTRACT SSS,s,D	1	-	-	-
	EXTRACT #iiii,s,D	2	-	-	-
EXTRACTU	EXTRACTU SSS,s,D	1	-	-	-
	EXTRACTU #iiii,s,D	2	-	-	-
IFcc	IFcc(.U)	1	-	-	-
ILLEGAL	ILLEGAL	5	-	-	-
INC	INC	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
INSERT	**INSERT SSS,qqq,D**	1	-	-	-
	INSERT #iiii,qqq,D	2	-	-	-
Jcc	**Jcc aa**	4	-	-	-
	Jcc ea	4	0	0	-
JCLR	**JCLR #bbbbb,S:<ea>,aaaa**	4	1	-	-
	JCLR #bbbbb,S:<pp>,aaaa	4	-	-	-
	JCLR #bbbbb,S:<aa>,aaaa	4	-	-	-
	JCLR #bbbbb,DDDDDD,aaaa	4	-	-	-
	JCLR #bbbbb,S:<qq>,aaaa	4	-	-	-
JMP	**JMP aa**	3	-	-	-
	JMP ea	3	1	1	-
JScc	**JScc aa**	4	-	-	-
	JScc ea	4	0	0	-
JSCLR	**JSCLR #bbbbb,S:<pp>,aaaa**	4	-	-	-
	JSCLR #bbbbb,S:<ea>,aaaa	4	1	-	-
	JSCLR #bbbbb,S:<aa>,aaaa	4	-	-	-
	JSCLR #bbbbb,DDDDDD,aaaa	4	-	-	-
	JSCLR #bbbbb,S:<qq>,aaaa	4	-	-	-
JSET	**JSET #bbbbb,S:<pp>,aaaa**	4	-	-	-
	JSET #bbbbb,S:<ea>,aaaa	4	1	-	-
	JSET #bbbbb,S:<aa>,aaaa	4	-	-	-
	JSET #bbbbb,DDDDDD,aaaa	4	-	-	-
	JSET #bbbbb,S:<qq>,aaaa	4	-	-	-
JSR	**JSR aa**	3	-	-	-
	JSR ea	3	1	1	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
JSSET	JSSET #bbbbb,S:<pp>,aaaa	4	-	-	-
	JSSET #bbbbb,S:<ea>,aaaa	4	1	-	-
	JSSET #bbbbb,S:<aa>,aaaa	4	-	-	-
	JSSET #bbbbb,DDDDDD,aaaa	4	-	-	-
	JSSET #bbbbb,S:<qq>,aaaa	4	-	-	-
LSL	LSL sss,D	1	-	-	-
	LSL #ii,D	1	-	-	-
LSR	LSR #ii,D	1	-	-	-
	LSR sss,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 +/- #iiiiii,QQ,D	2	-	-	-
MAC	MAC +/- 2**s,QQ,d	1	-	-	-
	MAC S1,S2,D (su,uu)	1	-	-	-
MAX	MAX A,B	1	-	-	-
MAXM	MAXM A,B	1	-	-	-
MACRI	MACRI +/- #iiiiii,QQ,D	2	-	-	-
MACR	MACR +/- 2**s,QQ,d	1	-	-	-
MERGE	MERGE SSS,D	1	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
MOVE	No parallel data Move (DALU)	1	-	-	-
	MOVE #xx --> DDDDD	1	-	-	-
	MOVE ddddd --> DDDDD	1	-	-	-
	U move	1	-	-	-
	MOVE S:<ea>,DDDDD	1	1	1	1
	MOVE S:<aa>,DDDDD	1	-	-	-
	MOVE S:<Rn+aa>,DDDD	2	-	-	-
	MOVE S:<Rn+aaaa>,DDDDDD	3	-	-	-
	MOVE d -> X Y:<ea>,YY	1	1	1	1
	MOVE X:<ea>,XX & d ->Y	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	-	-
	MOVE Y0 -> B B Y:<ea>	1	1	-	-
	MOVE L:<ea>,LLL	1	1	1	-
	MOVE L:<aa>,LLL	1	-	-	-
	MOVE X:<ea>,XX & Y:<ea>,YY	1	-	-	-
MOVEC	MOVEC #xx -> 1DDDDD	1	-	-	-
	MOVEC S:<ea>,1DDDDD	1	1	1	1
	MOVEC S:<aa>,1DDDDD	1	-	-	-
	MOVEC DDDDDD,1ddddd	1	-	-	-
MOVEM	MOVEM P:<ea>,DDDDDD	6	1	1	-
	MOVEM P:<aa>,DDDDDD	6	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
MOVEP	**MOVEP S:<pp>,s:<ea>**	2	1	1	0
	MOVEP S:<pp>,P:<ea>	6	1	1	-
	MOVEP S:<pp>,DDDDDD	1	-	-	-
	MOVEP X:<qq>,s:<ea>	2	1	1	0
	MOVEP Y:<qq>,s:<ea>	2	1	1	0
	MOVEP X:<qq>,DDDDDD	1	-	-	-
	MOVEP Y:<qq>,DDDDDD	1	-	-	-
	MOVEP S:<qq>,P:<ea>	6	1	1	-
MPY	**MPY S1,S2,D (su,uu)**	1	-	-	-
	MPY +/- 2s,QQ,d**	1	-	-	-
MPYI	**MPYI +/- #iiiiii,QQ,D**	2	-	-	-
MPYR	**MPYR +/- 2**s,QQ,d**	1	-	-	-
MPYRI	**MPYRI +/- #iiiiii,QQ,D**	2	-	-	-
NOP	**NOP**	1	-	-	-
NORM	**NORM**	5	-	-	-
NORMF	**NORMF SSS,D**	1	-	-	-
OR	**OR #iiiiii,D**	2	-	-	-
	OR #iii,D	1	-	-	-
ORI	**ORI EE**	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	-	-	-

Instruction Mnemonic	Instruction Format	T	+pru	+lab	+lim
REP	REP #xxx	5	-	-	-
	REP DDDDDD	5	-	-	-
	REP S:<ea>	5	1	-	-
	REP S:<aa>	5	-	-	-
RESET	RESET	7	-	-	-
RTI/RTS	RTI	3	-	-	-
	RTS	3	-	-	-
STOP	STOP	10	-	-	-
SUB	SUB #iiiiii,D	2	-	-	-
	SUB #iii,D	1	-	-	-
Tcc	Tcc JJJ -> D ttt TTT	1	-	-	-
	Tcc JJJ -> D	1	-	-	-
	Tcc ttt -> TTT	1	-	-	-
TRAP/TRAPcc	TRAP	9	-	-	-
	TRAPcc	9	-	-	-
WAIT	WAIT	10	-	-	-

B-3 INSTRUCTION SEQUENCE DELAYS

Due to the pipeline nature of the DSP56300 Core, there are certain instruction sequences that cause a delay in the execution of instructions involved in that sequences. 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:

1. External Bus Wait States.
2. External Bus Contention.
3. Instruction fetch delays.
4. Data ALU Interlock.
5. Address Generation Interlock.
6. Stack Extension delays.
7. Pipeline interlocks.

B-3.1 External Bus Wait States

An External Bus Wait State is caused by an instruction accessing the external bus for data read or write. In this case, the execution time of the instruction is increased by the number of clock cycles equal to the number of wait states that is programmed for that external data access. The exact number of wait states depends on the type of memory accessed, as described in Chapter 2 of this document.

B-3.2 External Bus Contention

An External Bus Contention is caused by an attempt to simultaneously access the external bus with more than one source (REFRESH request from the internal DRAM controller, X memory space, Y memory space, P memory space or a DMA channel). In this case, the execution time of the instructions that reside in the pipeline at that time is lengthen by a number of clock cycles that is equal to the number of simultaneous requests minus 1. For every request, additional wait states will be added according to the memory speed, as described in Section B-3.1 above. If one of these requests is a REFRESH request, than this request will be the first to receive mastership over the external bus. If one of these requests is the DMA, then the following cases should be distinguished:

1. The DMA has higher priority than the CORE. In this case, the DMA will receive full control over the external bus and will hold that control for all its transfers, provided that they are all external. After the DMA finished its transfers, the bus will be given to the memory space in the order of P (first), X (second) and Y (last).
2. The DMA has a priority equal to the CORE. In this case, the bus will be given to the memory space in the order of P (first), X (second), Y (third) and DMA (last).
3. The DMA has lower priority than the CORE. In this case, the DMA will wait

for a free external bus slot and the bus will be given to the memory space in the order of P (first), X (second) and Y (last).

B-3.3 Instruction Fetch delays

An external Instruction Fetch is caused by one of the following two cases:

- Instruction Cache is disabled and a fetch to an external address is initiated. In this case, an external fetch will be initiated.
- Instruction Cache is enabled and a program fetch to an instruction that does not exist in the instruction cache is initiated. This produces a miss indication from the instruction cache control unit, and an external fetch will be initiated.

In both cases, if the external memory is an SSRAM (Synchronous Static RAM), one cycle delay is inserted after the external access. The effective number of stall states in the pipeline will be the number specified in the Bus Control Register (BCR) + 1. If the external memory is either SRAM or DRAM, this one cycle delay will not be inserted.

During the operation of the Instruction Cache Controller, the following special cases should be distinguished:

1. When two identical locations are fetched one after another, and the first one is detected as miss, the second one will also be detected as miss although it was written to the cache memory. The number of wait states added will be the same as the general miss case.
2. When the Burst Mode is enabled, than upon detection of miss, up to 4 fetch requests will be initiated by the core. The exact number of fetch requests depends on the two least significant bits of the address of the initiating fetch that was detected as miss -

2 List Significant bits	Number of generated fetches	Number of clock cycles added
11	1	0, as if Burst is Disabled
10	2	2
01	3	3
00	4	4

All these requests will be considered as one for the detection of contention states.

B-3.4 Data ALU Interlock

A Data ALU Interlock may be caused by one of the following sequences:

B-3.4.1 Arithmetic Stall

This interlock is caused by an instruction that uses one of the data ALU accumulators or accumulator-parts (A0, A1, A2, B0, B1, B2) as a source register to the move portion of that instruction, while the preceding instruction was an arithmetic instruction (i.e. an instruction that uses the internal Data-ALU data paths) that used the same accumulator as its destination. The execution of the initiating instruction will be delayed by **one** clock cycle.

B-3.4.2 Transfer Stall

This interlock is caused by an instruction that uses one of the data ALU registers (A0, A1, A2, B0, B1 or B2) or accumulators (A or B) as a source register to the move portion of that instruction, while the preceding instruction used the corresponding accumulator (A or B) or one of the data ALU registers (A0, A1, A2, B0, B1 or B2) that comprise this accumulator as its destination register in the move portion of that instruction. The execution of the initiating instruction will be delayed by **one** instruction cycle.

B-3.4.3 Status Stall

This interlock is caused by an instruction that reads the contents of the Status Register (SR) for either move operation or bit testing, while the **preceding or the second preceding** instruction was an arithmetic instruction (i.e. an instruction that uses the internal Data-ALU data paths). The execution of the initiating instruction will be delayed by **two or one** (respectively) instruction cycles.

B-3.5 Address Registers Interlocks

B-3.5.1 Conditional Transfer Interlock

This interlock is caused by a Transfer On-Condition (Tcc) instruction followed by an instruction that explicitly specifies one of the address generation registers: R0..R7 as its source operand. The execution of the second instruction will be delayed by **one** instruction cycle.

B-3.5.2 Address Generation Interlock

An Address Generation Interlock is caused by a move portion of an instruction that uses one of the AGU registers R0-R7 for address generation or for address calculation, while one of the three preceding instruction cycles used one of the register-set (Ri, Ni or Mi) members as a destination register in its move portion. For example, consider the following code:

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

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

Three types of Address Generation Interlock exist - type0, type1 and type2, depending on the distance, in term of clock cycles, between the instruction causing the interlock and the preceding instruction that used the AGU register as a destination. The following figure describes an example to each of the types:

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`

When an Address Generation Interlock of **Type0** is detected (during the decoding of I2 in the example), **three** nop clock cycles will be automatically inserted before the execution of the instruction starts. When an Address Generation Interlock of **Type1** is detected (during the decoding of I3 in the example), **two** nop clock cycles will be automatically inserted before the execution of the instruction starts. When an Address Generation Interlock of **Type2** is detected (during the decoding of I4 in the example), **one** nop clock cycle will be automatically inserted before the execution of the instruction starts.

Note that only **clock cycles** are counted to determine when interlock cycles should be inserted. Whenever 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 that used the register as destination is measured in term of 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 the detection of a possible new Address Generation Interlock. The following example demonstrates this feature.

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

In this example, a type1 Address Generation Interlock is detected during the decoding phase of I3 and two nop cycles are inserted before the execution of that instruction. During the decoding of I4, no Address Generation Interlock is detected - no nop cycles are inserted! If, however, I3 would have been an instruction that does not use R0, a type2 Address Generation Interlock would have been detected during the decoding phase of I4 and one nop cycle would have been inserted before the execution of that instruction.

B-3.6 Stack Extension Delays

Some instructions access the System Stack as part of their normal activity. If, however, the stack is full, or if it is empty, the special stack extension mechanism is engaged and the access will be completed only after an access to data memory is automatically performed. This will delay the decoding and the execution phases of that instruction. A stack-full or stack-empty states are defined by the contents of the SC (Stack Counter) register. When the stack counter equals 14, it means that the on-chip hardware stack has 14 words (a stack word is a 48-bit long word combined from the low and the high portions of the stack) inside. The stack is declared as stack-full, and any additional push operation will activate the stack extension mechanism. When the stack counter equals 2, it means that the on-chip hardware stack has only 2 words inside. The stack is declared as stack-empty, and any additional pop operation will activate the stack extension mechanism.

The following instructions/cases causes an access to the system stack and may engage the stack extension mechanism:

SUBcc This denotes all the conditional and unconditional ‘Jump to Subroutine’ instructions e.g. JSR, JSSET, BRCLR etc. 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 will terminate the subroutine execution.

RET This denotes 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 back to the calling procedure and to restore the status bits and loop flag state.

END-OF-DO This is a condition achieved by the internal hardware inside the Program Control Unit. This hardware detects the case where a fetch from the last address of a loop is initiated when the Loop Counter equals 1. This condition defines the end of the loop, thus performs 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 This denotes all the hardware-loop initiating instructions e.g. DO, DOR 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 in order to go back 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 This is 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 This denotes all the explicit stack PUSH instructions that uses SSH as their destination, e.g. the instruction MOVE R0,SSH.

SSHRD This denotes all the explicit stack POP instructions that uses SSH as their source, e.g. the instruction MOVE SSH,Y1.

The following table describes how many clock cycles are added in the various instructions/ cases described above:

Table B-2. Stack Extension Delays

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

B-3.7 Program Flow-Control delays

During the execution of flow-control instructions, some boundary cases exist and introduce interlocks to the program flow. These interlocks lengthen the decoding phase of the instructions thus delays the execution of them.

Legend:

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

Note: The sequences described in this section represent very unusual operations which probably would never be used. The detection of these cases and hence the generation of interlocks is done in order to maintain an object code compatibility between the DSP56300 Core and the 56k Family of Digital Signal Processors.

B-3.7.1 MOVE to CR

Whenever I1 is a MOVE to CR and it is located at (LA-3), (LA-4) or (LA-5) then the decoding phase of I3 is delayed by 3 clock cycles. The decoding of the instruction following (LA) will be also delayed by an additional 1 clock cycle.

B-3.7.2 MOVE from CR

Whenever I1 is a MOVE from CR and it is located at (LA-2) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 1 clock cycle.

B-3.7.3 MOVE to SP/SC

Whenever I1 is a MOVE to SP or to SC then the decoding phase of I3 will be delayed by up to 3 clock cycles.

B-3.7.4 MOVE to LA register

Whenever I1 is a MOVE to the LA register and the preceding instruction was a MOVE to SR then the decoding phase of I3 will be delayed by 3 clock cycles.

B-3.7.5 MOVE to SR

Whenever I1 is a MOVE to SR then the decoding phase of I2 will be delayed by 1 clock cycle.

B-3.7.6 MOVE to SSH/SSL

Whenever I1 is a MOVE to SSH or to SSL and I3 is any one of the instructions DO, DOR, RTI, RTS, ENDDO or BRKcc then the decoding phase of I3 will be delayed by 3 clock cycles.

B-3.7.7 JMP to (LA) or to (LA-1)

Whenever I1 is any type of JMP with the target address equals to (LA) or to (LA-1) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 2 or 1 clock cycles respectively.

B-3.7.8 RTI to (LA) or to (LA-1)

Whenever I1 is an RTI instruction whose return address is (LA) or (LA-1) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 2 or 1 clock cycles respectively.

B-3.7.9 MOVE from SSH

Whenever I1 is a MOVE from SSH and it is located at (LA-2) then the decoding phase of the instruction following the instruction at (LA) will be delayed by 1 clock cycle.

B-3.7.10 Conditional Instructions

Whenever I1 is a conditional change of flow instruction e.g. Jcc and the condition is false then the decoding phase of I2 will be delayed by 1 clock cycle.

B-3.7.11 Interrupt Abort

Whenever I1 is an instruction which its decoding phase is longer than 1 cycle then it may be aborted by the interrupt control unit. In this case, 1 clock cycle “hole” will be inserted to the pipeline after which the instruction at the interrupt vector will be decoded.

B-3.7.12 Degenerated DO loop

Whenever I1 is a DO loop but the loop contains only one instruction then the decoding phase of I1 is lengthen by 1 clock cycle.

B-3.7.13 Annulled REP and DO

If the repeat count of a REP or DO instruction is 0 then the decoding phase of the REP or the DO instruction is lengthen by 1 or 3 clock cycles respectively.

Note: Annulled REP or DO can be executed only when the Sixteen-Bit compatibility mode in the Status Register (SR[13]) is cleared. When this bit is set, a annulled REP will execute 2**16 times.

B-4 INSTRUCTION SEQUENCE RESTRICTIONS

Due to the pipelined nature of the DSP56300 Core central processor, there are certain instruction sequences that are forbidden and will cause undefined operation. Most of these restricted sequences would cause contention for an internal resource, such as the stack register. The DSP assembler will flag these as assembly errors.

Most of the following restrictions represent very unusual operations which probably would never be used but are listed only for completeness.

Legend:

- MOVE - any type of MOVE, MOVEM, MOVEP, MOVEC.
- LA - the last address of a DO LOOP
- Two-words <inst> - a double-word instruction in which the 2nd word is used as an immediate data or absolute address
- Single-word <inst> - an instruction with an addressing mode that does not need a 2nd word extension

B-4.1 Restrictions Near the End of DO Loops

Proper DO loop operation is not guaranteed if an instruction sequence similar to one of the sequences described below is used.

B-4.1.1 At LA-3

The following instructions should not start at address LA-3:

- MOVE to {LA}
- BCHG, BSET, BCLR on {LA}
- Two-words MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- Two-words MOVE from SSH
- Two-words PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.2 At LA-2

The following instructions should not start at address LA-2:

- DO, DOR, DOFOREVER
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH
- BCHG, BSET, BCLR on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- REP on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BTST on SSH
- JCLR, JSET, JSCLR, JSSET, BRCLR, BRSET, BSCLR, BSSET on SSH
- Two-words MOVE from {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

- ANDI, ORI on MR
- BRKcc
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.3 At LA-1

The following instructions should not start at address LA-1:

- DO, DOR, DOFOREVER
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- REP on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BCHG, BSET, BCLR, BTST on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- JCLR, JSET, JSCLR, JSSET on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BRCLR, BRSET, BSCLR, BSSET on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- ANDI, ORI on MR
- BRKcc
- ENDDO
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.1.4 At LA

The following instructions should not start at address LA:

- Any Two-word instruction
- MOVE to {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- MOVE from SSH
- BCHG, BSET, BCLR on {LA, LC, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}
- BTST on SSH
- ANDI, ORI on MR
- BRKcc
- JMP, JSR, BRA, BSR, Jcc, JScc, Bcc, BScc
- REP
- RESET, STOP, WAIT
- RTI, RTS
- ENDDO
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.2 General DO Restrictions

A DO loop should be initialized and aborted by using only the following instructions: DO, DOR, ENDDO and BREAKcc. The LF and the FV bits in the Status Register (SR) should not be explicitly changed by using one of 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 sequences described below is used.

- SSH can not be used as the source for the Loop-Count for a DO or DOR instruction
- The following instructions should not appear immediately before a DO, DOR or DOFOREVER:
    - MOVE from SSH
    - BTST on SSH
    - BCHG, BCLR, BSET, MOVE to/on {LA, LC, SR, SP, SC, SSH, SSL}
    - JSR, JScc, JSSET, JSCLR to LA whenever LF is set
    - BSR, BScc, BSSET, BSCLR, to LA whenever LF is set

B-4.3 ENDDO Restrictions

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, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

B-4.4 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, SR, SP, SC, SSH, SSL, SZ, VBA, OMR}

B-4.5 RTI and RTS Restrictions

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

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

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

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

B-4.6 SP, SC and SSH/SSL Manipulation Restrictions

The instructions in the following list #a should not appear immediately before any of the instructions from the following list #b.

List #a:

- MOVE to {SP,SC}

- BCHG, BSET, BCLR on {SP,SC}

List #b:

- MOVE from {SSH,SSL}
- BTST, BCHG, BSET, BCLR on {SSH,SSL}
- JSET, JCLR, JSSET, JSCLR, BRSET, BRCLR, BSSET, BSCLR on {SSH,SSL}

B-4.7 Fast Interrupt Routines

The following instructions may not be used in a fast interrupt routine:

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

B-4.8 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 (can not be repeated):

- REP, DO, DOR, DOFOREVER
- ENDDO, BRKcc
- JMP, Jcc, JCLR, JSET
- JSR, JScc, JSCLR, JSSET
- BRA, Bcc, BRSET, BRCLR
- BSR, BScc, BSSET, BSCLR
- RTS, RTI
- TRAP, TRAPcc
- WAIT, STOP
- PLOCK, PUNLOCK, PLOCKR, PUNLOCKR

B-4.9 Stack Extension Restrictions

The following instructions, related to the operation of the on-chip hardware stack extension, may not 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

B-4.10 Instruction Cache General Restrictions

The following instructions may not be used whenever the Instruction Cache is disabled. Using these instructions when the Instruction Cache is disabled will generate an illegal interrupt.

- PLOCK, PLOCKR
- PUNLOCK, PUNLOCKR
- PFREE, PFLUSH, PFLUSHUN

B-5 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 have been kept to a minimum such as, in fact, these effects are transparent to the user. Design techniques, such as forwarding and interlocking, alleviate the need for the user to have a thorough knowledge of the machine's pipeline in order to avoid data dependencies. This knowledge becomes relevant only when further optimization of the code is pursued. Therefore the pipeline is hidden from the user for the vast majority of the application development stage.

There is, however, an aspect of the machine's pipeline that is exposed to the user and this is the area of peripheral activity. This section describes the cases in which the user must take precautions in order to achieve the desired functionality.

B-5.1 Polling a peripheral device for write

When writing data to a peripheral device there is a two cycle pipeline delay until any status bits affected by this operation are updated. For example, the operates a peripheral port using the polling technique. The user will look for the data empty flag set. After this status bit is set, the user will write new data to the transmit data register. If the user attempts to read the status bit within the next two cycles, due to the pipeline delays associated with the peripheral operations, the user will mistakenly read the flag as set. Therefor the user will assume that the transmit data register is empty and will write a new data word that will in fact overwrite the previously written data. In order to achieve the correct functionality the user must wait (at least) two cycles before attempting to read the status register following a write to the transmit data register. Following is an example of the correct sequence for transmit operations:

```
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
```

B-5.2 Writing to a read-only register

Writing to a read-only register is an operation that basically 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 will be the value of the data that was written instead of the unchanged data of the read-only register. In order to ensure that the correct data is read after the write operation, the user should wait (at least) two cycles before performing the read.