MOTOROLA

MCF5102 User's Manual

PREFACE

The *MCF5102 User's Manual* describes the capabilities, operation, and programming of the MCF5102 ColdFire microprocessors. The *M68000 Family Programmer's Reference Manual* contains the complete instruction set.

The organization of this manual is as follows:

Section 1	Introduction
Section 2	Execution Pipelines
Section 3	Access Control Units
Section 4	Instruction and Data Caches
Section 5	Signal Description
Section 6	IEEE 1149.1 Test Access Port (JTAG)
Section 7	Bus Operation
Section 8	Exception Processing
Section 9	Instruction Timings
Section 10	MCF5102 Electrical and Thermal Characteristics
Section 11	Ordering Information and Mechanical Data
Appendix A	Address, $\overline{\text{TIP}}$, and $\overline{\text{LOCKE}}$ Generation
Appendix B	MCF5102 Evaluation Socket
Index	

ELECTRONIC SUPPORT:
The Technical Support BBS, known as AESOP (Application Engineering Support Through On-Line Productivity), can be reach by modem or the internet. AESOP provides commonly asked application questons, latest device errata, device specs, software code, and many other useful support functions.

Modem: Call 1-800-843-3451 (outside US or Canada 512-891-3650) on a modem that runs at 14,400 bps or slower. Set your software to N/8/1/F emulating a vt100.

Internet: This access is provided by telneting to pirs.aus.sps.mot.com [129.38.233.1] or through the World Wide Web at http://pirs.aus.sps.mot.com.

Apps FAX Line: You may FAX questions to 1-800-248-8567.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 6
IEEE 1149.1 Test Access Port (JTAG)

TABLE OF CONTENTS (Continued)

Section 7
Bus Operation

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

SECTION 1
INTRODUCTION

ColdFire™ represents a revolutionary new microprocessor architecture that has been optimized for embedded processing applications. It brings new levels of price and performance to cost-sensitive high-volume markets. Based on the concept of Variable-Length RISC technology, ColdFire combines the architectural simplicity of conventional 32-bit fixed length RISC with a memory-saving variable-length instruction set.

The ColdFire RISC processors are tuned to offer embedded processor designers significant system-level advantages over conventional fixed length RISC architectures. Binary code for ColdFire processors is denser and therefore takes up less program memory than 32-bit fixed-length machines. This improved code density results in systems which require less memory for a given application and also allows the use of slower and less costly memory to achieve a given performance level.

The first ColdFire Family member to be announced, the MCF5102 is fully ColdFire code compatible. As the first chip in the family, it has been designed with special capabilities which allow it to also execute the M68000 code which exists today. These extensions to the ColdFire instruction set allow Motorola customers to utilize the MCF5102 as a bridge to future ColdFire processors for applications requiring the advantages of a variable-length RISC architecture.

By providing compatibility with the M68000 Family this enables designers to take full advantage of industry leading software and hardware tools. Compatibility with existing development tools such as compilers, debuggers, real-time operating systems and adapted hardware tools offers MCF5102 developers access to a broad range of mature tool support; enabling an accelerated product development cycle, lower development costs and critical time-to-market advantages for Motorola customers.

1.1 FEATURES

The primary features of the MCF5102 are as follows:

- Very Low Cost ColdFire Compatible Integer Core
  - — Optimized Variable-Length Instruction Set for Embedded Applications
  - — Extended Instruction Support For M68040 Code Compatibility
  - — 16 User Visible 32 Bit Wide Registers
- High Integer Performance
  - — 1 instruction Per Clock Peak Performance

- On Chip Caches
  - — 2K bytes Instruction Cache, 4-way set associative
  - — 1K bytes Data Cache, 4-way set associative
  - — Data Cache supports bus snooping
- 4 Separate Access Control Registers
- Supervisor / User Modes For System Protection
- Low Interrupt Latency
- Multiplexed 32-Bit Address and 32-Bit Data Bus To Minimize Board Space And Interconnections
- Full Static Design Allows Operation Down To DC To Minimize Power Consumption
- 3.3-Volt Operation
- 5-Volt TTL Compatible, 5-Volt CMOS Tolerant
- JTAG IEEE 1149.1 Test Interface
- Single Bus Clock Input
- Fast Locking Internal PLL

1.2 FUNCTIONAL BLOCKS

Figure 1-1 illustrates a simplified block diagram of the MCF5102. The MCF5102 consists to two tightly coupled multi-stage pipelines. The Instruction Fetch Pipeline (IFP) is responsible for instruction address calculation, instruction prefetch, and instruction decode. The Operand Execution Pipeline (OEP) includes stages for effective address calculation, operand fetch, instruction execute and writeback. Conditional branches are optimized for the more common case of the branch taken, and both execution paths of the branch are fetched and decoded to minimize refilling of the instruction pipeline.

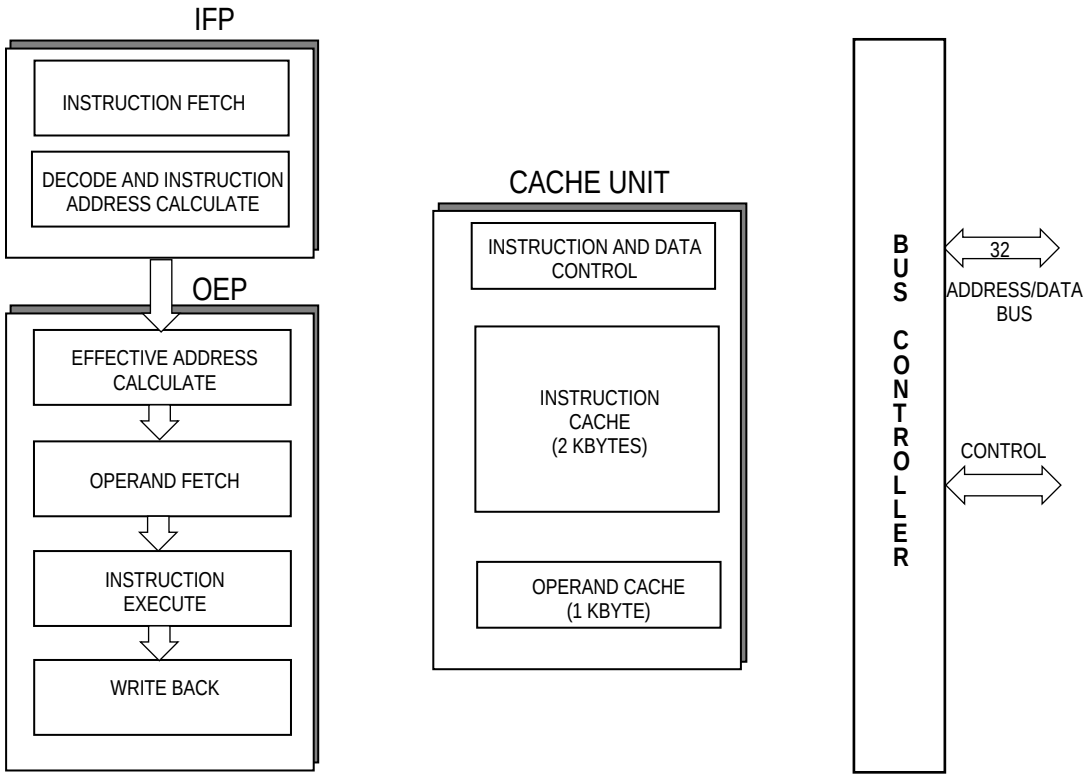


Figure 1-1. Block Diagram

Separate on-chip instruction and data caches operate independently. The caches improve the overall performance of the system by reducing the number of bus transfers required by the processor to fetch information from slower external memory and by increasing the bus bandwidth available for alternate bus masters in the system. Both caches are organized as four-way set associative. Each line contains four long words for a storage capability of 2 Kbytes of instruction and 1 Kbyte for data cache (3 Kbytes total). Each cache is allocated separate internal address and data buses, allowing simultaneous access to both. The data cache provides write-through or copyback write modes. The caches are physically mapped, reducing software support for multitasking operating systems, and support external bus snooping to maintain cache coherency in multimaster systems.

The bus controller executes bus transfers on the external bus and prioritizes external memory requests from each cache. The MCF5102 bus controller supports a high-speed, multiplexed, synchronous, external bus interface supporting burst accesses for both reads and writes to provide high data transfer rates to and from the internal caches. Additional bus signals support bus snooping and external cache tag maintenance.

1.3 PROCESSING STATES

The processor is always in one of four states: normal processing, exception processing, low-power, or halted. It is in the normal processing state when executing instructions, fetching instructions and operands, and storing instruction results.

Exception processing is the transition from program processing to system, interrupt, and exception handling. Exception processing includes fetching the exception vector, stacking operations, and refilling the instruction pipe caused after an exception. The processor enters exception processing when an exceptional internal condition arises such as tracing an instruction, an instruction results in a trap, or executing specific instructions. External conditions, such as interrupts and access errors, also cause exceptions. Exception processing ends when the first instruction of the exception handler begins to execute.

The low-power stop mode is a reduced power mode of operation, that causes the processor to remain quiescent until either a reset or non-masked interrupt occurs. This mode of operation has four phases of operation and is triggered by the low-power stop (LPSTOP) instruction.

The processor halts when it receives an access error or generates an address error while in the exception processing state. For example, if during exception processing of one access error another access error occurs, the MCF5102 is unable to complete the transition to normal processing and cannot save the internal state of the machine. The processor assumes that the system is not operational and halts. Only an external reset can restart a halted processor. Note that when the processor executes a STOP instruction, it is in a special type of normal processing state, one without bus cycles. The processor stops, but it does not halt.

1.4 PROGRAMMING MODEL

The ColdFire programming model is separated into two privilege modes: supervisor and user. The S-bit in the status register (SR) indicates the privilege mode that the processor uses. The processor identifies a logical address by accessing either the supervisor or user address space, maintaining the differentiation between supervisor and user modes.

Programs access registers based on the indicated mode. User programs can only access registers specific to the user mode; whereas, system software executing in the supervisor mode can access all registers, using the control registers to perform supervisory functions. User programs are thus restricted from accessing privileged information, and the operating system performs management and service tasks for the user programs by coordinating their activities. This difference allows the supervisor mode to protect system resources from uncontrolled accesses.

Most instructions execute in either mode, but some instructions that have important system effects are privileged and can only execute in the supervisor mode. For instance, user programs cannot execute the STOP or RESET instructions. To prevent a user program from entering the supervisor mode, except in a controlled manner, instructions that can alter the S-bit in the SR are privileged. The TRAP instructions provide controlled access to operating system services for user programs.

If the S-bit in the SR is set, the processor executes instructions in the supervisor mode. Because the processor performs all exception processing in the supervisor mode, all bus cycles generated during exception processing are supervisor references, and all stack accesses use the active supervisor stack pointer. If the S-bit of the SR is clear, the

processor executes instructions in the user mode. The bus cycles for an instruction executed in the user mode are user references. The values on the transfer modifier pins indicate either supervisor or user accesses.

The processor utilizes the user mode and the user programming model when it is in normal processing. During exception processing, the processor changes from user to supervisor mode. Exception processing saves the current value of the SR on the active supervisor stack and then sets the S-bit, forcing the processor into the supervisor mode. To return to the user mode, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE, which execute in the supervisor mode, modifying the S-bit of the SR. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space.

The registers depicted in the programming model (see Figure 1-2) provide operand storage and control for these three units. The registers are partitioned into two levels of privilege modes: user and supervisor. The user programming model consists of 16, general-purpose, 32-bit registers and two control registers.

Only system programmers can use the supervisor programming model to implement operating system functions, and I/O control. This supervisor/user distinction allows for the coding of application software, which if confined to the ColdFire instruction set rather than the extended instruction set of the MCF5102, will run without modification on any ColdFire family processor. The supervisor programming model contains the control features that system designers would not want user code to erroneously access because this might effect normal operation of the system. Furthermore, the supervisor programming model may need to change slightly from ColdFire generation to generation to add features or improve performance as the architecture evolves.

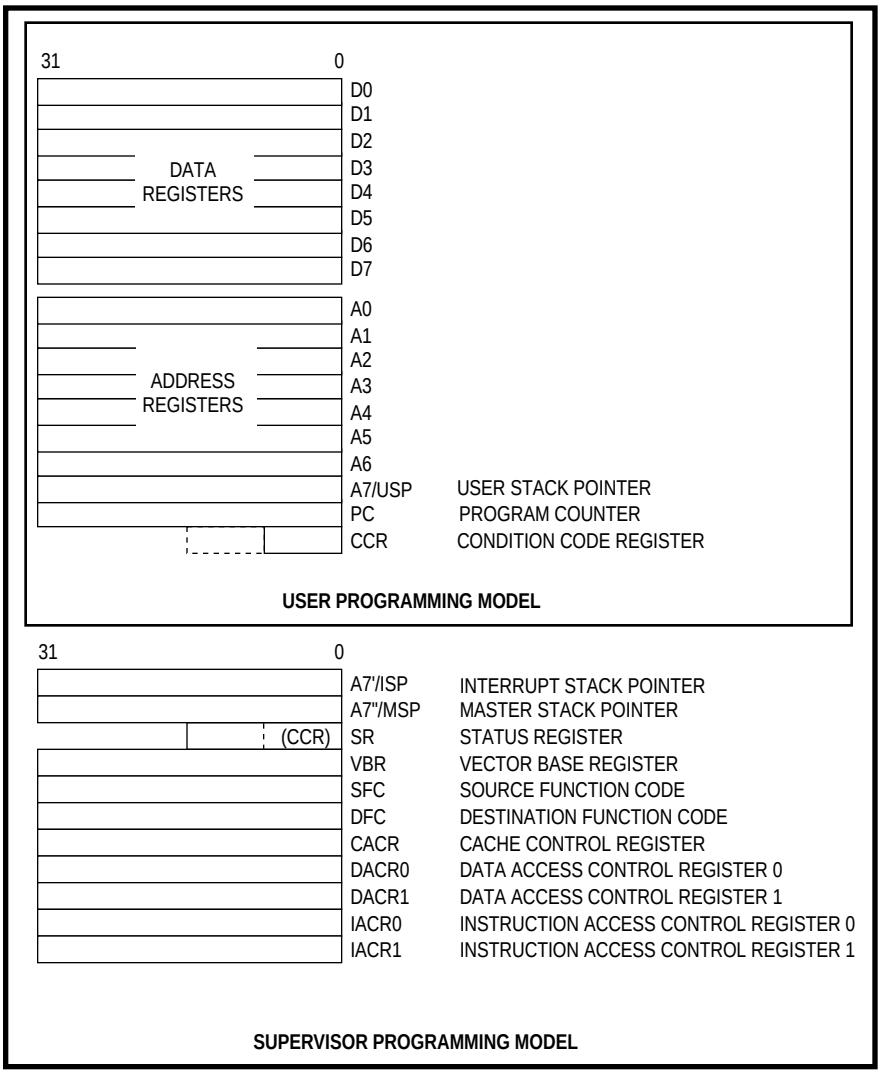


Figure 1-2. Programming Model

The user programming model includes eight data registers, seven address registers, and a stack pointer register. The address registers and stack pointer can be used as base address registers or software stack pointers, and any of the 16 registers can be used as index registers. Two control registers are available in the user mode—the program counter (PC), which usually contains the address of the instruction that the MCF5102 is executing, and the lower byte of the SR, which is accessible as the condition code register (CCR). The CCR contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program.

The supervisor programming model includes the upper byte of the SR, which contains operation control information. The vector base register (VBR) contains the base address of the exception vector table, which is used in exception processing. The source function code (SFC) and destination function code (DFC) registers contain 3-bit function codes. These function codes can be considered extensions to the 32-bit logical address. The processor automatically generates function codes to select address spaces for data and

program accesses in the user and supervisor modes. Some instructions use the alternate function code registers to specify the function codes for various operations.

The cache control register (CACR) controls enabling of the on-chip instruction and data caches of the MCF5102. There are four transparent translation registers, two for instruction accesses and two for data accesses. These registers allow portions of the logical address space to be transparently mapped to specify cachability.

1.5 DATA FORMAT SUMMARY

The MCF5102 provides support for the basic data formats of the ColdFire architecture and also provides extended support for all the data formats of the M68000 family. The instruction set supports operations on other data formats such as memory addresses, bit, bit field, binary-coded decimal (BCD), byte, word, long word, quad word and 16-byte.

The operand data formats supported are standard twos-complement data. Registers, memory, or instructions themselves can contain operands. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Table 1-1 lists a summary of the data formats for the MCF5102 which are ColdFire compatible. Table 1-2 lists the extended data formats supported exclusively by the MCF5102 which provide compatibility for existing M68000 code.

Table 1-1. ColdFire MCF5102 Data Formats

Operand Data Format	Size	Notes
Bit	1 Bit	—
Byte Integer	8 Bits	—
Word Integer	16 Bits	—
Long-Word Integer	32 Bits	—
Quad-Word Integer	64 Bits	Any Two Data Registers

Table 1-2. MCF5102 Extended Data Formats

Operand Data Format	Size	Notes
Bit Field	1–32 Bits	Field of Consecutive Bits
Binary-Coded Decimal (BCD)	8 Bits	Packed: 2 Digits/Byte; Unpacked: 1 Digit/Byte
16-Byte	128 Bits	Memory Only, Aligned to 16-Byte Boundary

1.6 ADDRESSING CAPABILITIES SUMMARY

The MCF5102 supports the basic addressing modes of the ColdFire family and also provides extended support for all the addressing modes of the M68000 family. The register indirect addressing modes support postincrement, predecrement, offset, and

indexing, which are particularly useful for handling data structures common to sophisticated embedded applications and high-level languages. The program counter indirect mode also has indexing and offset capabilities. This addressing mode is typically required to support position-independent software. Besides these addressing modes, the MCF5102 provides index sizing and scaling features.

An instruction's addressing mode can specify the value of an operand, a register containing the operand, or how to derive the effective address of an operand in memory. Each addressing mode has an assembler syntax. Some instructions imply the addressing mode for an operand. These instructions include the appropriate fields for operands that use only one addressing mode. Table 1-3 lists a summary of the effective addressing modes for the MCF5102 which are ColdFire compatible. Table 1-4 lists the extended addressing modes supported exclusively by the MCF5102 which provide compatibility for existing M68000 code.

Table 1-3. ColdFire MCF5102 Effective Addressing Modes

Addressing Modes	Syntax
Register Direct	
Data	Dn
Address	An
Register Indirect	
Address	(An)
Address with Postincrement	(An)+
Address with Predecrement	–(An)
Address with Displacement	(d16,An)
Address Register Indirect with Index	
8-Bit Displacement	(d_8,An,Xn)
Base Displacement	(bd,An,Xn)
Program Counter Indirect	
with Displacement	(d_{16},PC)
Program Counter Indirect with Index	
8-Bit Displacement	(d_8,PC,Xn)
Base Displacement	(bd,PC,Xn)
Absolute Data Addressing	
Short	(xxx).W
Long	(xxx).L
Immediate	#<xxx>

Table 1-4. MCF5102 Extended Effective Addressing Modes

Addressing Modes	Syntax
Memory Indirect	
Postindexed	([bd,An],Xn,od)
Preindexed	([bd,An,Xn],od)
Program Counter Memory Indirect	
Postindexed	([bd,PC],Xn,od)
Preindexed	([bd,PC,Xn],od)

1.7 NOTATIONAL CONVENTIONS

Table 1-5 lists the notation conventions used throughout this manual unless otherwise specified.

Table 1-5. Notational Conventions

Single- And Double-Operand Operations	
+	Arithmetic addition or postincrement indicator.
–	Arithmetic subtraction or predecrement indicator.
¥	Arithmetic multiplication.
÷	Arithmetic division or conjunction symbol.
~	Invert; operand is logically complemented.
L	Logical AND
V	Logical OR
≈	Logical exclusive OR
➧	Source operand is moved to destination operand.
➧➧	Two operands are exchanged.
<op>	Any double-operand operation.
<operand>tested	Operand is compared to zero and the condition codes are set appropriately.
sign-extended	All bits of the upper portion are made equal to the high-order bit of the lower portion.
Other Operations	
TRAP	Equivalent to Format ÷ Offset Word ➧ (SSP); SSP – 2 ➧ SSP; PC ➧ (SSP); SSP – 4 ➧ SSP; SR ➧ (SSP); SSP – 2 ➧ SSP; (Vector) ➧ PC
STOP	Enter the stopped state, waiting for interrupts.
<operand>$_{10}$	The operand is BCD; operations are performed in decimal.
If <condition> then <operations> else <operations>	Test the condition. If true, the operations after “then” are performed. If the condition is false and the optional “else” clause is present, the operations after “else” are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.
Register Specification	
An	Any Address Register n (example: A3 is address register 3)
Ax, Ay	Source and destination address registers, respectively.
BR	Base Register—An, PC, or suppressed.
Dc	Data register D7–D0, used during compare.
Dh, Dl	Data registers high- or low-order 32 bits of product.
Dn	Any Data Register n (example: D5 is data register 5)
Dr, Dq	Data register’s remainder or quotient of divide.
Du	Data register D7–D0, used during update.
Dx, Dy	Source and destination data registers, respectively.
MRn	Any Memory Register n.
Rn	Any Address or Data Register
Rx, Ry	Any source and destination registers, respectively.
Xn	Index Register—An, Dn, or suppressed.

Table 1-5. Notational Conventions (Continued)

Data Format And Type	
+ inf	Positive Infinity
<fmt>	Operand Data Format: Byte (B), Word (W), Long (L), Single (S), Double (D), Extended (X), or Packed (P).
B, W, L	Specifies a signed integer data type (twos complement) of byte, word, or long word.
D	Double-precision real data format (64 bits).
k	A twos complement signed integer (–64 to +17) specifying a number's format to be stored in the packed decimal format.
P	Packed BCD real data format (96 bits, 12 bytes).
S	Single-precision real data format (32 bits).
X	Extended-precision real data format (96 bits, 16 bits unused).
– inf	Negative Infinity
Subfields and Qualifiers	
#<xxx> or #<data>	Immediate data following the instruction word(s).
( )	Identifies an indirect address in a register.
[ ]	Identifies an indirect address in memory.
bd	Base Displacement
ccc	Index into the MC68881/MC68882 Constant ROM
d_n	Displacement Value, n Bits Wide (example: d_{16} is a 16-bit displacement).
LSB	Least Significant Bit
LSW	Least Significant Word
MSB	Most Significant Bit
MSW	Most Significant Word
od	Outer Displacement
SCALE	A scale factor (1, 2, 4, or 8, for no-word, word, long-word, or quad-word scaling, respectively).
SIZE	The index register's size (W for word, L for long word).
{offset:width}	Bit field selection.
Register Names	
CCR	Condition Code Register (lower byte of status register)
DFC	Destination Function Code Register
IC, DC, IC/DC	Instruction, Data, or Both Caches
PC	Program Counter
Rc	Any Non Floating-Point Control Register
SFC	Source Function Code Register
SR	Status Register

Table 1-5. Notational Conventions (Concluded)

Register Codes	
*	General Case.
C	Carry Bit in CCR
cc	Condition Codes from CCR
FC	Function Code
N	Negative Bit in CCR
U	Undefined, Reserved for Motorola Use.
V	Overflow Bit in CCR
X	Extend Bit in CCR
Z	Zero Bit in CCR
—	Not Affected or Applicable.
Stack Pointers	
ISP	Supervisor/Interrupt Stack Pointer
MSP	Supervisor/Master Stack Pointer
SP	Active Stack Pointer
SSP	Supervisor (Master or Interrupt) Stack Pointer
USP	User Stack Pointer
Miscellaneous	
<ea>	Effective Address
<label>	Assemble Program Label
<list>	List of registers, for example D3–D0.
LB	Lower Bound
m	Bit m of an Operand
m–n	Bits m through n of Operand
UB	Upper Bound

1.8 INSTRUCTION SET OVERVIEW

The instruction set is tailored to support high-level languages and is optimized for those instructions most commonly executed by embedded code. Table 1-7 provides an alphabetized listing of the ColdFire instruction set's opcode, operation, and syntax. Table 1-8 lists the extended instructions supported exclusively by the MCF5102 which provide compatibility for existing M68000 code. Refer to Table 1-5 for notations used in Tables 1-7 and 1-8. The left operand in the syntax is always the source operand, and the right operand is the destination operand.

Table 1-7. ColdFire MCF5102 Instruction Set Summary

Opcode	Operation	Syntax
ADD	Source + Destination ➧ Destination	ADD <ea>,Dn ADD Dn,<ea>
ADDA	Source + Destination ➧ Destination	ADDA <ea>,An
ADDI	Immediate Data + Destination ➧ Destination	ADDI #<data>,<ea>
ADDQ	Immediate Data + Destination ➧ Destination	ADDQ #<data>,<ea>
ADDX	Source + Destination + X ➧ Destination	ADDX Dy,Dx ADDX –(Ay),–(Ax)
AND	Source L Destination ➧ Destination	AND <ea>,Dn AND Dn,<ea>
ANDI	Immediate Data L Destination ➧ Destination	ANDI #<data>,<ea>
ANDI to CCR	Source L CCR ➧ CCR	ANDI #<data>,CCR
ANDI to SR	If supervisor state then Source L SR ➧ SR else TRAP	ANDI #<data>,SR
ASL, ASR	Destination Shifted by count ➧ Destination	ASd Dx,Dy[1] ASd #<data>,Dy[1] ASd <ea>[1]
Bcc	If condition true then PC + d_n ➧ PC	Bcc <label>
BCHG	~(bit number of Destination) ➧ Z; ~(bit number of Destination) ➧ (bit number) of Destination	BCHG Dn,<ea> BCHG #<data>,<ea>
BCLR	~(bit number of Destination) ➧ Z; 0 ➧ bit number of Destination	BCLR Dn,<ea> BCLR #<data>,<ea>
BRA	pc + D_N ➧ pc	bra <LABEL>
BSET	~(bit number of Destination) ➧ Z; 1 ➧ bit number of Destination	BSET Dn,<ea> BSET #<data>,<ea>
BSR	SP – 4 ➧ SP; PC ➧ (SP); PC + d_n ➧ PC	BSR <label>
BTST	–(bit number of Destination) ➧ Z;	BTST Dn,<ea> BTST #<data>,<ea>
CLR	0 ➧ Destination	CLR <ea>
CMP	Destination – Source ➧ cc	CMP <ea>,Dn
CMPA	Destination – Source	CMPA <ea>,An
CMPI	Destination – Immediate Data	CMPI #<data>,<ea>
EOR	Source ≈ Destination ➧ Destination	EOR Dn,<ea>

EORI	Immediate Data ≈ Destination ➧ Destination	EORI #<data>,<ea>
EXT EXTB	Destination Sign – Extended ➧ Destination	EXT.W Dn extend byte to word EXT.L L Dn extend word to long word EXTB.L Dn extend byte to long word
JMP	Destination Address ➧ PC	JMP <ea>
JSR	SP – 4 ➧ SP; PC ➧ (SP) Destination Address ➧ PC	JSR <ea>
LEA	<ea> ➧ An	LEA <ea>,An
LINK	SP – 4 ➧ SP; An ➧ (SP) SP ➧ An, SP+d ➧ SP	LINK An,d_n
LSL, LSR	Destination Shifted by count ➧ Destination	LSd Dx,Dy[1] LSd #<data>,Dy[1] LSd <ea>[1]
MOVE	Source ➧ Destination	MOVE <ea>,<ea>
MOVEA	Source ➧ Destination	MOVEA <ea>,An
MOVE from CCR	CCR ➧ Destination	MOVE CCR,<ea>
MOVE to CCR	Source ➧ CCR	MOVE <ea>,CCR
MOVE from SR	If supervisor state then SR ➧ Destination else TRAP	MOVE SR,<ea>
MOVE to SR	If supervisor state then Source ➧ SR else TRAP	MOVE <ea>,SR
MOVEC	If supervisor state then Rc ➧ Rn or Rn ➧ Rc else TRAP	MOVEC Rc,Rn MOVEC Rn,Rc
MOVEM	Registers ➧ Destination Source ➧ Registers	MOVEM <list>,<ea>[2] MOVEM <ea>,<list>[2]
MOVEQ	Immediate Data ➧ Destination	MOVEQ #<data>,Dn
MULS	Source ¥ Destination ➧ Destination	MULS.W <ea>,Dn 16 ¥ 16 ➧ 32 MULS.L <ea>,Dl 32 ¥ 32 ➧ 32
MULU	Source ¥ Destination ➧ Destination	MULU.W <ea>,Dn 16 ¥ 16 ➧ 32 MULU.L <ea>,Dl 32 ¥ 32 ➧ 32
NEG	0 – (Destination) ➧ Destination	NEG <ea>
NEGX	0 – (Destination) – X ➧ Destination	NEGX <ea>
NOP	None	NOP
NOT	~ Destination ➧ Destination	NOT <ea>
OR	Source V Destination ➧ Destination	OR <ea>,Dn OR Dn,<ea>
ORI	Immediate Data V Destination ➧ Destination	ORI #<data>,<ea>
PEA	SP – 4 ➧ SP; <ea> ➧ (SP)	PEA <ea>
RTE	If supervisor state then (SP) ➧ SR; SP + 2 ➧ SP; (SP) ➧ PC; SP + 4 ➧ SP; restore state and deallocate stack according to (SP) else TRAP	RTE
RTS	(SP) ➧ PC; SP + 4 ➧ SP	RTS

Scc	If condition true then 1s ➧ Destination else 0s ➧ Destination	Scc <ea>
STOP	If supervisor state then Immediate Data ➧ SR; STOP else TRAP	STOP #<data>
SUB	Destination – Source ➧ Destination	SUB <ea>,Dn SUB Dn,<ea>
SUBA	Destination – Source ➧ Destination	SUBA <ea>,An
SUBI	Destination – Immediate Data ➧ Destination	SUBI #<data>,<ea>
SUBQ	Destination – Immediate Data ➧ Destination	SUBQ #<data>,<ea>
SUBX	Destination – Source – X ➧ Destination	SUBX Dx,Dy SUBX –(Ax),–(Ay)
SWAP	Register 31–16 ⟷ Register 15–0	SWAP Dn
TRAP	SSP – 2 ➧ SSP; Format ÷ Offset ➧ (SSP); SSP – 4 ➧ SSP; PC ➧ (SSP); SSP – 2 ➧ SSP; SR ➧ (SSP); Vector Address ➧ PC	TRAP #<vector>
TRAPF	If never then TRAP	TRAPF TRAPF.W #<data> TRAPF.L #<data>
TST	Destination Tested ➧ Condition Codes	TST <ea>
UNLK	An ➧ SP; (SP) ➧ An; SP + 4 ➧ SP	UNLK An

NOTES:
1. Where d is direction, left or right.
2. List refers to register.

Table 1-8. MCF5102 Instruction Set Extensions

Opcode	Operation	Syntax
ABCD	BCD Source + BCD Destination + X ➧ Destination	ABCD Dy,Dx ABCD –(Ay),–(Ax)
BFCHG	~(bit field of Destination) ➧ bit field of Destination	BFCHG <ea>{offset:width}
BFCLR	0 ➧ bit field of Destination	BFCLR <ea>{offset:width}
BFEXTS	bit field of Source ➧ Dn	BFEXTS <ea>{offset:width},Dn
BFEXTU	bit offset of Source ➧ Dn	BFEXTU <ea>{offset:width},Dn
BFFFO	bit offset of Source Bit Scan ➧ Dn	BFFFO <ea>{offset:width},Dn
BFINS	Dn ➧ bit field of Destination	BFINS Dn,<ea>{offset:width}
BFSET	1s ➧ bit field of Destination	BFSET <ea>{offset:width}
BFTST	bit field of Destination	BFTST <ea>{offset:width}
BKPT	Run breakpoint acknowledge cycle; TRAP as illegal instruction	BKPT #<data>
CAS	CAS Destination – Compare Operand ➧ cc; if Z, Update Operand ➧ Destination else Destination ➧ Compare Operand	CAS Dc,Du,<ea>
CAS2	CAS2 Destination 1 – Compare 1 ➧ cc; if Z, Destination 2 – Compare ➧ cc; if Z, Update 1 ➧ Destination 1; Update 2 ➧ Destination 2 else Destination 1 ➧ Compare 1; Destination 2 ➧ Compare 2	CAS2 Dc1–Dc2,Du1–Du2,(Rn1)–(Rn2)
CHK	If Dn < 0 or Dn > Source then TRAP	CHK <ea>,Dn
CHK2	If Rn < LB or If Rn > UB then TRAP	CHK2 <ea>,Rn
CINV	If supervisor state then invalidate selected cache lines else TRAP	CINVL <caches>, (An) CINVP <caches>, (An) CINVA <caches>
CMPM	Destination – Source ➧ cc	CMPM (Ay)+,(Ax)+
CMP2	Compare Rn < LB or Rn > UB and Set Condition Codes	CMP2 <ea>,Rn
CPUSH	If supervisor state then if data cache push selected dirty data cache lines; invalidate selected cache lines else TRAP	CPUSHL <caches>, (An) CPUSHP <caches>, (An) CPUSHA <caches>
DBcc	If condition false then (Dn–1 ➧ Dn; If Dn ≠ –1 then PC + d_n ➧ PC)	DBcc Dn,<label>
DIVS, DIVSL	Destination ÷ Source ➧ Destination	DIVS.W <ea>,Dn 32 ÷ 16 ➧ 16r:16q DIVS.L <ea>,Dq 32 ÷ 32 ➧ 32q DIVS.L <ea>,Dr:Dq 64 ÷ 32 ➧ 32r:32q DIVSL.L <ea>,Dr:Dq 32 ÷ 32 ➧ 32r:32q
DIVU, DIVUL	Destination ÷ Source ➧ Destination	DIVU.W <ea>,Dn 32 ÷ 16 ➧ 16r:16q DIVU.L <ea>,Dq 32 ÷ 32 ➧ 32q DIVU.L <ea>,Dr:Dq 64 ÷ 32 ➧ 32r:32q DIVUL.L <ea>,Dr:Dq 32 ÷ 32 ➧ 32r:32q

Table 1-8. MCF5102 Instruction Set Extensions (Continued)

EORI to CCR	Source ≈ CCR ➧ CCR	EORI #<data>,CCR
EORI to SR	If supervisor state then Source ≈ SR ➧ SR else TRAP	EORI #<data>,SR
EXG	Rx ➧ ➧ Ry	EXG Dx,Dy EXG Ax,Ay EXG Dx,Ay EXG Ay,Dx
ILLEGAL	SSP – 2 ➧ SSP; Vector Offset ➧ (SSP); SSP – 4 ➧ SSP; PC ➧ (SSP); SSp – 2 ➧ SSP; SR ➧ (SSP); Illegal Instruction Vector Address ➧ PC	ILLEGAL
LPSTOP	If supervisor state immediate data ➧ SR SR ➧ broadcast cycle STOP else TRAP	LPSTOP #<data>
MOVE USP	If supervisor state then USP ➧ An or An ➧ USP else TRAP	MOVE USP,An MOVE An,USP
MOVE16	Source block ➧ Destination block	MOVE16 (Ax)+, (Ay)+[2] MOVE16 (xxx).L, (An) MOVE16 (An), (xxx).L MOVE16 (An)+, (xxx).L
MOVEP	Source ➧ Destination	MOVEP Dx,(d_n,Ay) MOVEP (d_n,Ay),Dx
MOVES	If supervisor state then Rn ➧ Destination [DFC] or Source [SFC] ➧ Rn else TRAP	MOVES Rn,<ea> MOVES <ea>,Rn
MULS	Source ¥ Destination ➧ Destination	MULS.L <ea>,Dh–Dl 32 ¥ 32 ➧ 64
MULU	Source ¥ Destination ➧ Destination	MULU.L <ea>,Dh–Dl 32 ¥ 32 ➧ 64
NBCD	0 – (Destination$_{10}$) – X ➧ Destination	NBCD <ea>
ORI to CCR	Source V CCR ➧ CCR	ORI #<data>,CCR
ORI to SR	If supervisor state then Source V SR ➧ SR else TRAP	ORI #<data>,SR
PACK	Source (Unpacked BCD) + adjustment ➧ Destination (Packed BCD)	PACK –(Ax),–(Ay),#(adjustment) PACK Dx,Dy,#(adjustment)
RESET	If supervisor state then Assert RSTO Line else TRAP	RESET
ROL, ROR	Destination Rotated by count ➧ Destination	ROd Rx,Dy[1] ROd #<data>,Dy[1]
ROXL, ROXR	Destination Rotated with X by count ➧ Destination	ROXd Dx,Dy[1] ROXd #<data>,Dy[1] ROXd <ea>[1]
RTD	(SP) ➧ PC; SP + 4 + d_n ➧ SP	RTD #(d_n)

Table 1-8. MCF5102 Instruction Set Extensions (Continued)

RTR	(SP) ➧ CCR; SP + 2 ➧ SP; (SP) ➧ PC; SP + 4 ➧ SP	RTR
SBCD	Destination10 – Source$_{10}$ – X ➧ Destination	SBCD Dx,Dy SBCD –(Ax),–(Ay)
TAS	Destination Tested ➧ Condition Codes; 1 ➧ bit 7 of Destination	TAS <ea>
TRAPcc	If cc then TRAP	TRAPcc TRAPcc.W #<data> TRAPcc.L #<data>
TRAPV	If V then TRAP	TRAPV
UNPK	Source (Packed BCD) + adjustment ➧ Destination (Unpacked BCD)	UNPACK –(Ax),–(Ay),#(adjustment) UNPACK Dx,Dy,#(adjustment)

NOTES:
1. Where d is direction, left or right.
2. MOVE16 (ax)+,(ay)+ is functionally the same as MOVE16 (ax),(ay)+ when ax = ay. The address register is only incremented once, and the line is copied over itself rather than to the next line.

SECTION 2
EXECUTION PIPELINES

This section describes the organization of the MCF5102 instruction and operand execution pipelines and a brief description of the associated registers.

2.1 PIPELINES

The MCF5102 is comprised of two tightly coupled execution pipelines. The instruction fetch pipeline (IFP) is a 2-stage pipeline which prefetches and decodes instructions. The decoded instruction stream is then gated into the 4-stage operand execution pipeline (OEP), which calculates any needed effective addresses, fetches the required operands and then executes the required function. Since the IFP and OEP operate semi-autonomously, the IFP is able to prefetch instructions in advance of their actual use by the OEP thereby minimizing time stalled waiting for instructions. Figure 2-1 illustrates the MCF5102 pipeline structure.

The IFP consists of two stages:

- Instruction Fetch
- Instruction Decode and next Instruction Address Calculate

The OEP is implemented using a four-stage pipeline featuring a traditional RISC datapath with a register file feeding an arithmetic/logic unit. In this design, each pipeline stage has a specific function:

- Effective Address Calculate
- Operand Fetch
- Instruction Execute
- Write-back

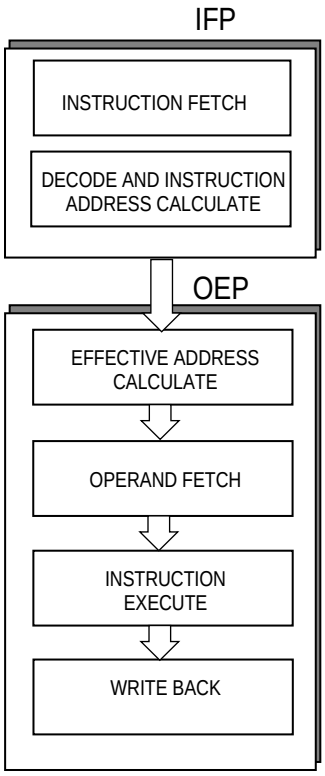


Figure 2-1. Execution Pipeline

The IFP contains special shadow registers that can begin processing future instructions for conditional branches while the OEP is processing current instructions. An instruction stream is fetched from the instruction cache and decoded on an instruction-by-instruction basis in the decode stage. Multiple instructions are fetched to keep the pipeline stages full to minimize pipeline stalls.

The effective address calculate stage eliminates pipeline blockage for instructions with postincrement, postdecrement, or immediate add and load to address register for updates that occur in the effective address calculate stage.

The resulting effective address is passed to the operand fetch stage, which initiates an operand fetch from the data cache if the effective address is for a source operand. The fetched operand is returned to the Instruction execute stage, which completes execution of the instruction and writes any result to either a data register, memory, or back to the effective address calculate stage for storage in an address register. For a memory destination, the operand fetch stage passes the operand to the instruction execute stage.

The previously described sequence of effective address calculation and fetch can occur multiple times for an instruction, depending on the source and/or destination addressing modes. For memory indirect addressing modes, the effective address calculate stage initiates an operand fetch from the intermediate indirect memory address, then calculates the final effective address. Also, some instructions access multiple memory operands and initiate fetches for each operand.

The instruction finishes execution in the instruction execute stage. Instructions with write-back operands to memory generate pending write accesses that are passed to the write-back stage.

The write-back stage holds the operand until an opportune moment when no data fetches are required. The write-back can defer writes indefinitely until either the data cache is free or another write is pending from the execution stage. Holding the data in the write-back stage maximizes system performance by not interrupting the incoming instruction or data stream.

2.2 PROGRAMMING MODEL REGISTERS

The following paragraphs describe the registers in the user and supervisor programming models.

2.2.1 User Programming Model

Figure 2-2 illustrates the user programming model which consist of the following registers.

- 16 General-Purpose 32-Bit Registers (D7–D0, A7–A0)
- 32-Bit Program Counter (PC)
- 8-Bit Condition Code Register (CCR)

2.2.1.1 DATA REGISTERS (D7–D0). These registers are used as data registers for bit and bit field (1 to 32 bits), byte (8 bit), word (16 bit), long-word (32 bit), and quad-word (64 bit) operations. These registers may also be used as index registers.

2.2.1.2 ADDRESS REGISTERS (A6–A0). These registers can be used as software stack pointers, index registers, or base address registers. The address registers may be used for word and long-word operations.

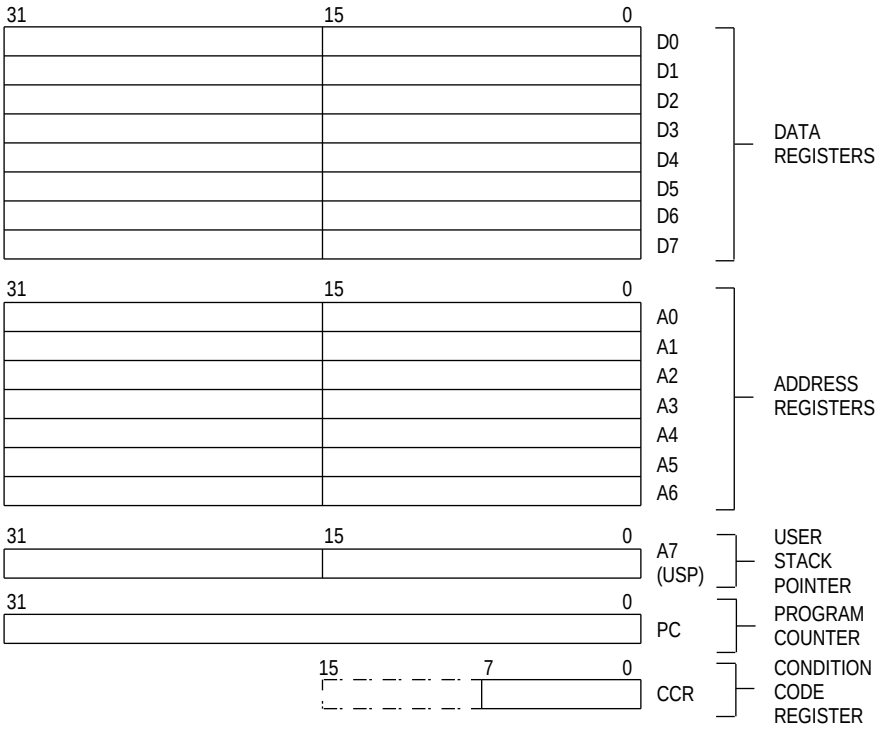


Figure 2-2. User Programming Model

2.2.1.3 SYSTEM STACK POINTER (A7). A7 is used as a hardware stack pointer during stacking for subroutine calls and exception handling. The register designation A7 refers to three different uses of the register: the user stack pointer (USP) (A7) in the user programming model and either the interrupt stack pointer (ISP) or master stack pointer (MSP) (A7' or A7", respectively) in the supervisor programming model. When the S-bit in the status register (SR) is clear, the USP is the active stack pointer. Explicit references to the system stack pointer (SSP) refer to the USP while the processor is operating in the user mode.

A subroutine call saves the program counter (PC) on the active system stack, and the return restores it from the active system stack. Both the PC and the SR are saved on the supervisor stack (either ISP or MSP) during the processing of exceptions and interrupts. Thus, the execution of supervisor level code is independent of user code and condition of the user stack. Conversely, user programs use the USP independently of supervisor stack requirements.

2.2.1.4 PROGRAM COUNTER. The PC contains the address of the currently executing instruction. During instruction execution and exception processing, the processor automatically increments the contents of the PC or places a new value in the PC, as appropriate. For some addressing modes, the PC can be used as a pointer for PC-relative addressing.

2.2.1.5 CONDITION CODE REGISTER. The CCR consists of five bits of the SR least significant byte. The first four bits represent a condition of the result generated by a processor operation. The fifth bit, the extend bit (X-bit), is an operand for multiprecision

computations. The carry bit (C-bit) and the X-bit are separate in the M68000 family to simplify programming techniques that use them.

2.2.2 Supervisor Programming Model

Only system programmers use the supervisor programming model (see Figure 2-3) to implement sensitive operating system functions, and I/O control. All accesses that affect the control features of the MCF5102 are in the supervisor programming model. Thus, all application software is written to run in the user mode and migrates to the MCF5102 from any M68000 platform without modification.

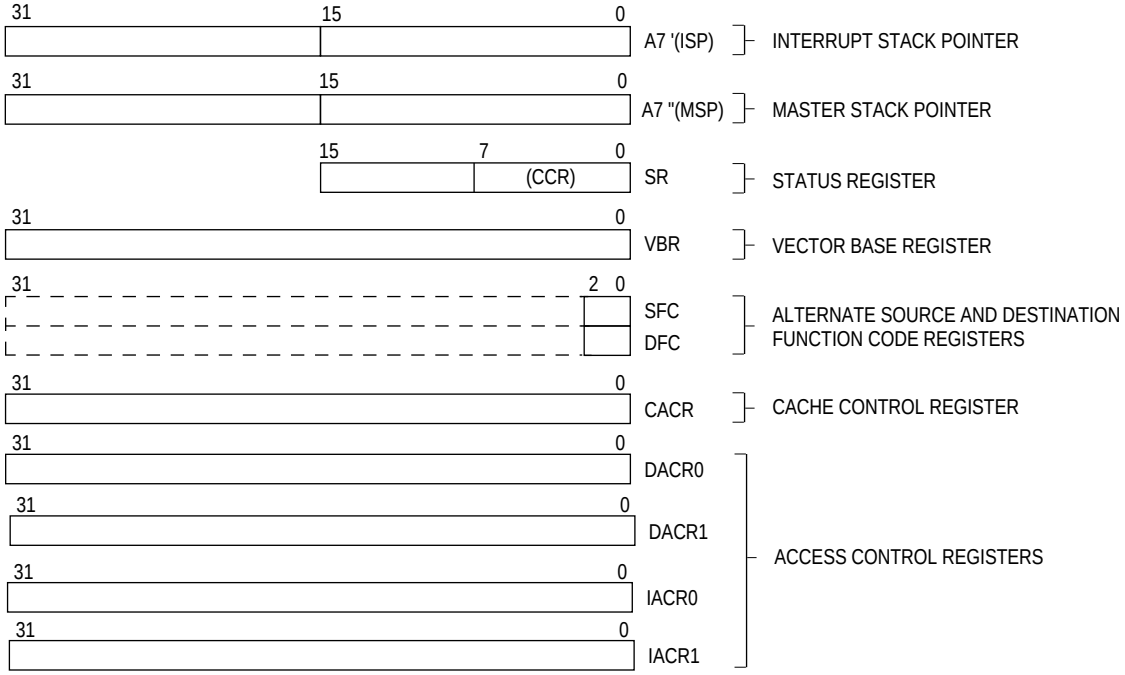


Figure 2-3. Supervisor Programming Model

The supervisor programming model consists of the registers available to the user as well as the following control registers:

- Two 32-Bit Supervisor Stack Pointers (ISP, MSP)
- 16-Bit Status Register (SR)
- 32-Bit Vector Base Register (VBR)
- Two 32-Bit Alternate Function Code Registers: Source Function Code (SFC) and Destination Function Code (DFC)
- 32-Bit Cache Control Register (CACR)
- Four 32-bit Access Control Registers (DACR0, DACR1, IACR0, IACR1)

The following paragraphs describe the supervisor programming model registers. Additional information on the ISP, MSP, SR, and VBR registers can be found in **Section 8 Exception Processing.**

2.2.2.1 INTERRUPT AND MASTER STACK POINTERS. In a multitasking operating system, it is more efficient to have a supervisor stack pointer associated with each user task and a separate stack pointer for interrupt-associated tasks. The MCF5102 provides two supervisor stack pointers, master and interrupt. Explicit references to the SSP refer to either the MSP or ISP while the processor is operating in the supervisor mode. All instructions that use the SSP implicitly reference the active stack pointer. The ISP and MSP are general-purpose registers and can be used as software stack pointers, index registers, or base address registers. The ISP and MSP can be used for word and long-word operations.

The M-bit of the SR selects whether the ISP or MSP is active. SSP references access the ISP when the M-bit is clear, putting the processor into the interrupt mode. If an exception being processed is an interrupt and the M-bit is set, the M-bit is cleared, putting the processor into the interrupt mode. The interrupt mode is the default condition after reset, and all SSP references access the ISP. The ISP can be used for interrupt control information and for workspace area as interrupt exception handling requires.

SSP references access the MSP when the M-bit is set. The operating system uses the MSP for each task pointing to a task-related area of supervisor data space. This procedure separates task-related supervisor activity from asynchronous, I/O-related supervisor tasks that can only be coincidental to the currently executing task. The MSP can separately maintain task control information for each currently executing user task, and the software updates the MSP when a task switch is performed, providing an efficient means for transferring task-related stack items. The value of the M-bit does not affect execution of privileged instructions. Instructions that affect the M-bit are MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, and RTE. The processor automatically saves the M-bit value and clears it in the SR as part of the exception processing for interrupts.

2.2.2.2 STATUS REGISTER. The SR (see Figure 2-4) stores the processor status. In the supervisor mode, software can access the full SR, including the CCR available in user mode (see **2.2.1.5 Condition Code Register**) and the interrupt priority mask and additional control bits available only in the supervisor mode. These bits indicate the following states for the processor: one of two trace modes (T1, T0), supervisor or user mode (S), and master or interrupt mode (M).

The term SSP refers to the ISP and MSP. The M and S bits of the SR decide which SSP to use. When the S-bit is one and the M-bit is zero, the ISP is the active stack pointer; when the S-bit is one and the M-bit is one, the MSP is the active stack pointer. The ISP is the default stack pointer after reset.

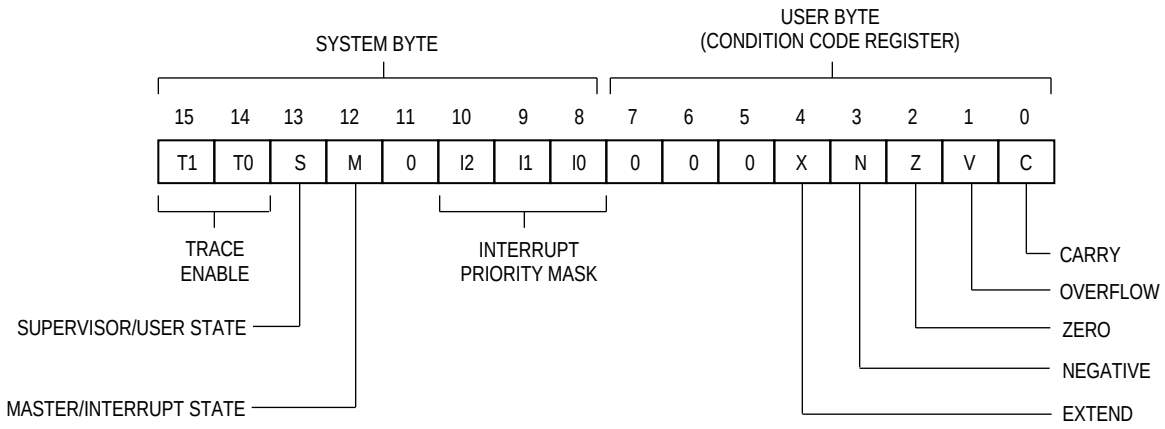


Figure 2-4. Status Register

2.2.2.3 VECTOR BASE REGISTER. The VBR contains the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table. Refer to **Section 8 Exception Processing** for information on exception vectors.

2.2.2.4 ALTERNATE FUNCTION CODE REGISTERS. The alternate function code registers contain 3-bit function codes. Function codes can be considered extensions of the 32-bit logical address that optionally provides as many as eight 4-Gbyte address spaces. The processor automatically generates function codes to select address spaces for data and programs at the user and supervisor modes. Certain instructions use the SFC and DFC registers to specify the function codes for operations.

2.2.2.5 CACHE CONTROL REGISTER. The CACR contains two enable bits that allow the instruction and data caches to be independently enabled or disabled. Setting an enable bit enables the associated cache without affecting the state of any lines within the cache. A hardware reset clears the CACR, disabling both caches.

SECTION 3
ACCESS CONTROL UNITS

The MCF5102 contains two independent ACUs, one for instructions and one for data. Each ACU allows memory selections to be made requiring attributes particular to peripherals, shared memory, or other special memory requirements. The following paragraphs describe the ACUs and the access control registers contained in them.

3.1 ACCESS CONTROL REGISTERS

Each ACU has two independent access control registers (ACRs). The instruction ACU contains the instruction access control registers (IACR0 and IACR1). The data ACU contains the data access control registers (DACR0 and DACR1). Both ACRs provide and control status information for access control of memory in the MCF5102. Only programs that execute in the supervisor mode using the MOVEC instruction can directly access the ACRs.

The 32-bit ACRs each define blocks of address space for access control. These blocks of address space can overlap or be separate, and are a minimum of 16 Mbytes. Three blocks are used with two user-defined attributes, cachability control and optional write protection. The ACRs specify a block of address space as serialized noncachable for peripheral selections and as write-through for shared blocks of address space in multi-processing applications. The ACRs can be configured to support many embedded control applications while maintaining cache integrity. Refer to **Section 4 Instruction and Data Caches** for details concerning cachability. Figure 3-1 illustrates the ACR format.

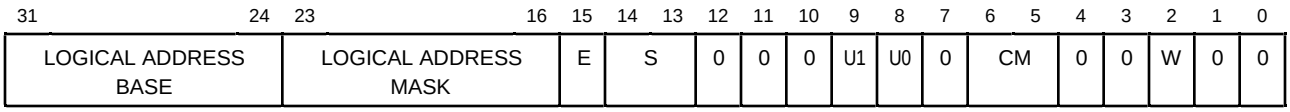


Figure 3-1. Access Control Register Format

ADDRESS BASE

This 8-bit field is compared with physical address bits A31–A24. Addresses that match in this comparison (and are otherwise eligible) are accessible.

ADDRESS MASK

This 8-bit field contains a mask for the ADDRESS BASE field. Setting a bit in the ADDRESS MASK field causes the processor to ignore the corresponding bit in the ADDRESS BASE field. Setting some of the ADDRESS MASK bits to ones obtains blocks of memory larger than 16 Mbytes. The low-order bits of this field are normally set to define contiguous blocks larger than 16 Mbytes, although contiguous blocks are not required.

E—Enable

This bit enables and disables transparent translation of the block defined by this register.

0 = Access control disabled.
1 = Access control enabled.

S—Supervisor/User Mode

This field specifies the way FC2 is used in matching an address:

00 = Match only if FC2 = 0 (user mode access).
01 = Match only if FC2 = 1 (supervisor mode access).
10, 11 = Ignore FC2 when matching.

CM—Cache Mode

This field selects the cache mode and access serialization for a page as follows:

00 = Cachable, Write-through
01 = Cachable, Copyback
10 = Noncachable, Serialized
11 = Noncachable

Detailed information on caching modes is available in **Section 4 Instruction and Data Caches**, and information on serialization is available in **Section 7 Bus Operation**.

W—Write Protect

This bit indicates if the transparent block is write protected. If set, write and read-modify-write accesses to the protected block.

0 = Read and write accesses permitted.
1 = Write accesses not permitted.

3.2 ADDRESS COMPARISON

The following description of address comparison assumes that the ACRs are enabled. Clearing the E-bit in each ACR independently disables access control, causing the processor to ignore it.

When an ACU receives an address, the privilege mode and the eight high-order bits of the address are compared to the block of addresses defined by the two ACRs for the corresponding ACU. Each block of address space for an ACR contains an S-field, a BASE ADDRESS field, and an ADDRESS MASK field. The S-field allows for matching either user or supervisor accesses (or both). Setting a bit in the ADDRESS MASK field causes the corresponding bit of the ADDRESS BASE to be ignored in the address comparison and privilege mode. Setting successively higher order bits in the ADDRESS MASK field increases the size of the block of address space.

The address for the current bus cycle and an ACR address match when the privilege mode and address bits for each (not including the masked bits) are equal. Each ACR specifies write protection for the block of address space. Enabling write protection for a block of address space causes the abortion of write or read-modify-write accesses to the block, and an access error exception occurs.

By appropriately configuring an ACR, flexible mappings can be specified. For example, to control access to the user address space, the S-field equals $0, and the ADDRESS MASK field equals $FF in all four ACRs. To control access to the supervisor address space ($00000000–$0FFFFFFF) with write protection, the BASE ADDRESS field = $0X, the ADDRESS MASK field equals $0F, the W-bit is set to one, and the S-field = $1. The inclusion of independent ACRs in both the instruction ACU (IACU) and data ACU (DACU provides an exception to the merged instruction and data address space, allowing different access control for instruction and operand accesses. Also, since the instruction memory unit is only used for instruction prefetches, different instruction and data ACRs can cause PC relative operand fetches to be controlled differently from instruction prefetches.

3.3 EFFECT OF $\overline{\text{RSTI}}$ ON THE ACU

When the assertion of the reset input ($\overline{\text{RSTI}}$) signal resets the MCF5102, the E-bits of the ACR's are cleared, disabling address access control.

SECTION 4
INSTRUCTION AND DATA CACHES

The MCF5102 contains two independent on-chip caches located in the physical address space. Accessing instruction words and data simultaneously through separate caches increases instruction throughput. The MCF5102 caches improve system performance by providing cached data to the on-chip execution unit with very low latency. Systems with an alternate bus master receive increased bus availability.

Cache coherency in the MCF5102 is optimized for multimaster applications in which the MCF5102 is the caching master sharing memory with one or more noncaching masters (such as DMA controllers). The MCF5102 implements a bus snooper that maintains cache coherency by monitoring an alternate bus master's access and performing cache maintenance operations as requested by the alternate bus master. Matching cache entries can be invalidated during the alternate bus master's access to memory, or memory can be inhibited to allow the MCF5102 to respond to the access as a slave. For an external write operation, the processor can intervene in the access and update its internal caches (sink data). For an external read operation, the processor supplies cached data to the alternate bus muster (source data). This prevents the MCF5102 caches from accumulating old or invalid copies of data (stale data). Alternate bus masters are allowed access to locally modified data within the caches that is no longer consistent with external memory (dirty data). Allowing memory to be specified as write-through instead of copyback also supports cache coherency. When a processor writes to the write-through cache, external memory is always updated through an external bus access after updating the cache, keeping memory and cached data consistent.

4.1 CACHE OPERATION

The instruction cache is four-way set-associative that has 32 sets of four 16-byte lines for a total of 2 Kbytes. The data cache is also four-way set-associative that has 16 sets of four 16-byte lines for a total of 1 Kbyte. There are two formats that define each cache line, an instruction cache line format and a data cache line format. Each format contains an address tag consisting of the upper 23 bits of the physical address for the instruction cache and 24 bits for the physical address data cache., status information, and four long words (128 bits) of data. The status information for the instruction cache line address tag consists of a single valid bit for the entire line. The status information for the data cache line address tag contains a valid bit and four additional bits to indicate dirty status for each long word in the line. Note that only the data cache supports dirty cache lines. Figure 4-1 illustrates the instruction cache line format (a) and the data cache line format (b).

TAG	V	LW3	LW2	LW1	LW0

(a) Instruction Cache Line

TAG	V	LW3	D3	LW2	D2	LW1	D1	LW0	D0

TAG — 23 Bit Physical Address Tag for instruction
24 Bit Physical Address Tag for data
V — Line VALID Bit
LW — Long Word n (32-Bit) Data Entry
Dn — DIRTY Bit for Long Word n

(b) Data Cache Line

Figure 4-1. Cache Line Formats

The cache stores an entire line, providing validity on a line-by-line basis. Only burst mode accesses that successfully read four long words can be cached. Memory devices unable to support bursting can respond to a cache line read or write access by asserting the transfer burst inhibit ($\overline{\text{TBI}}$) signal, forcing the processor to complete the access as a sequence of three long-word accesses. The cache recognizes burst accesses as if the access were never inhibited, detecting no difference.

A cache line is always in one of three states: invalid, valid, or dirty. For invalid lines, the V-bit is clear, causing the cache line to be ignored during lookups. Valid lines have their V-bit set and D-bits cleared, indicating all four long words in the line contain valid data consistent with memory. Dirty cache lines have the V-bit and one or more D-bits set, indicating that the line has valid long-word entries that have not been written to memory (long words whose D-bit is set). A cache line changes from valid to invalid if the execution of the CINV or CPUSH instruction explicitly invalidates the cache line; if a snooped write access hits the cache line and the line is not dirty; or if the SCx signals for a snooped read access invalidates the line. Both caches should be explicitly cleared after a hardware reset of the processor since reset does not invalidate the cache lines.

Figure 4-2 illustrates the general flow of a caching operation. To minimize latency of the requested data the physical address bits and are used to access a set of cache lines. Physical address bits, 8–4 for the instruction cache and 7-4 for the data cache, are used to index into the cache and select one of the sets of four cache lines. The four tags from the selected cache set are compared with the translated physical address bits 31–12 and bits 11 and 10 of the address offset. If any one of the four tags matches and the tag status is either valid or dirty, then the cache has a hit. During read accesses, a half-line (two long words) is accessed at a time, requiring two cache accesses for reads that are greater than a half-line or two long words. Write accesses within a cache line require a single cache access.

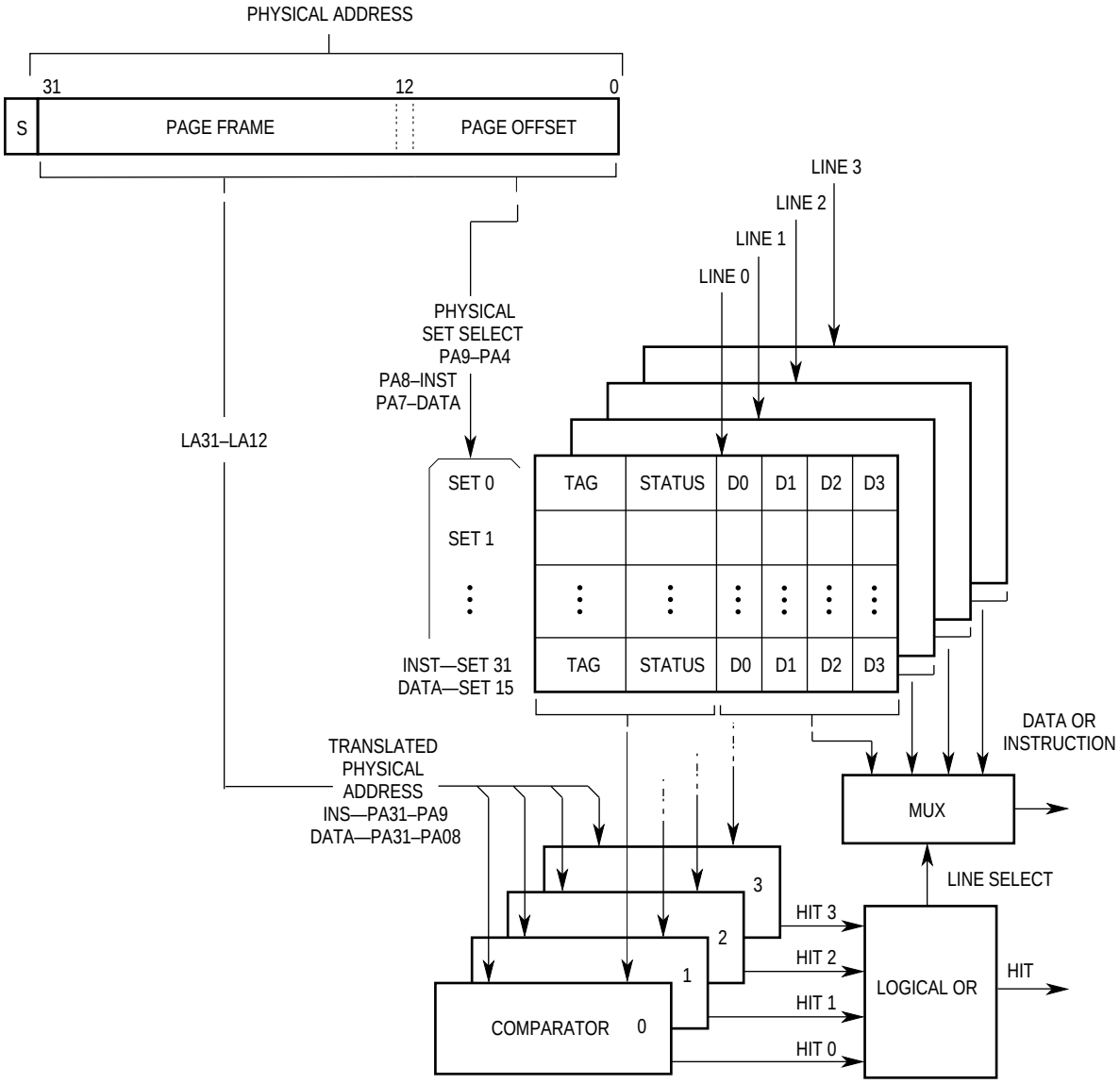
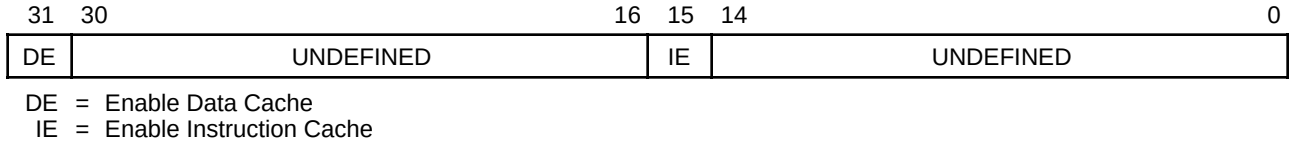


Figure 4-2. Caching Operation

Both caches contain circuitry to automatically determine which cache line in a set to use for a new line. The cache controller locates the first invalid line and uses it; if no invalid lines exist, then a pseudo-random replacement algorithm is used to select a valid line, replacing it with the new line. Each cache contains a 2-bit counter, which is incremented for each access to the cache. The instruction cache counter is incremented for each half-line accessed in the instruction cache. The data cache counter is incremented for each half-line accessed during reads, for each full line accessed during writes in copyback mode, and for each bus transfer resulting from a write in write-through mode. When a miss occurs and all four lines in the set are valid, the line pointed to by the current counter value is replaced, after which the counter is incremented.

4.2 CACHE MANAGEMENT

Using the MOVEC instruction, the caches are individually enabled to access the 32-bit cache control register (CACR) illustrated in Figure 4-3. The CACR contains two enable bits that allow the instruction and data caches to be independently enabled or disabled. Setting one of these bits enables the associated cache without affecting the state of any lines within the cache. A hardware reset clears the CACR, disabling both caches; however, reset does not affect the tags, state information, and data within the caches. The CINV instruction must clear the caches before enabling them.



DE = Enable Data Cache
IE = Enable Instruction Cache

Figure 4-3. Cache Control Register

System hardware can assert the cache disable ($\overline{\text{CDIS}}$) signal to dynamically disable both caches, regardless of the state of the enable bits in the CACR. The caches are disabled immediately after the current access completes. If $\overline{\text{CDIS}}$ is asserted during the access for the first half of a misaligned operand spanning two cache lines, the data cache is disabled for the second half of the operand. Accesses by the execution units bypass the caches while they are disabled and do not affect their contents (with the exception of CINV and CPUSH instructions). Disabling the caches with $\overline{\text{CDIS}}$ does not affect snoop operations. $\overline{\text{CDIS}}$ is intended primarily for use by in-circuit emulators to allow swapping between the tags and emulator memories.

Even if the instruction cache is disabled, the MCF5102 can cache instructions because of an internal cache line register. This happens for instruction loops that are completely resident within the first six bytes of a half-line. Thus, the cache line holding register can operate as a small cache. If a loop fits anywhere within the first three words of a half-line, then it becomes cached.

The CINV and CPUSH instructions support cache management in the supervisor mode. CINV allows selective invalidation of cache entries. CPUSH performs two operations: 1) any selected data cache lines containing dirty data are pushed to memory; 2) all selected cache lines are invalidated.

4.3 CACHING MODES

Every IU access to the cache has an associated caching mode that determines how the cache handles the access. An access can be cachable in either the write-through or copyback modes, or it can be cache inhibited in nonserialized or serialized modes. The CM field in the Access Control register, ACR, corresponding to the logical address of the access normally specifies one of these caching modes. The default memory access caching mode is nonserialized. When the cache is enabled the default caching mode is write-through. The ACR registers allow the defaults to be overridden. In addition, some instructions and IU operations perform data accesses that have an implicit caching mode

associated with them. The following paragraphs discuss the different caching accesses and their related cache modes.

4.3.1 Cachable Accesses

If the CM field indicates write-through or copyback, then the access is cachable. A read access to a write-through or copyback is read from the cache if matching data is found. Otherwise, the data is read from memory and used to update the cache. Since instruction cache accesses are always reads, the selection of write-through or copyback modes do not affected them. The following paragraphs describe the write-through and copyback modes in detail.

4.3.1.1 WRITE-THROUGH MODE. Accesses to memory specified as write-through are always written to the external address, although the cycle can be buffered, keeping memory and cache data consistent. Writes in write-through mode are handled with a no-write-allocate policy—i.e., writes that miss in a data cache are written to memory but do not cause the corresponding line in memory to be loaded into the cache. Write accesses always write through to memory and update matching cache lines. Specifying write-through mode for the shared pages maintains cache coherency for shared memory areas in a multiprocessing environment. The cache supplies data to instruction or data read accesses that hit in the appropriate cache; misses cause a new cache line to be loaded into the cache, replacing a valid cache line if there are no invalid lines.

4.3.1.2 COPYBACK MODE. Copyback memory is typically used for local data structures or stacks to minimize external bus usage and reduce write access latency. Write accesses specified as copyback that hit in the data cache update the cache line and set the corresponding D-bits without an external bus access. The dirty cached data is only written to memory if 1) the line is replaced due to a miss, 2) a cache inhibited access matches the line, or 3) the CPUSH instruction explicitly pushes the line.

4.3.2 Cache-Inhibited Accesses

Address space regions containing targets such as I/O devices and shared data structures in multiprocessing systems can be designated cache inhibited. If the ACR's CM field indicates nonserialized or serialized, then the access is cache inhibited. The caching operation is identical for both cache-inhibited modes. If the CM field of a matching address indicates either nonserialized or serialized modes, the cache controller bypasses the cache and performs an external bus transfer. The data associated with the access is not cached internally, and the cache inhibited out ($\overline{\text{CIOUT}}$) signal is asserted during the bus transfer to indicate to external memory that the access should not be cached. If the data cache line is already resident in an internal cache, then the data cache line is pushed from the cache if it is dirty or the data cache line is invalidated if it is valid.

If the CM field indicates serialized, then the sequence of read and write accesses to memory is guaranteed to match the sequence of the instruction order. Without serialization, the IU pipeline allows read accesses to occur before completion of a write-back for a previous instruction. Serialization forces operand read accesses for an instruction to occur only once by preventing the instruction from being interrupted after the operand fetch stage. Otherwise, the instruction is aborted, and the operand is accessed

when the instruction is restarted. These guarantees apply only when the CM field indicates the serialized mode and the accesses are aligned. Regardless of the selected cache mode, locked accesses are implicitly serialized. The TAS, CAS, and CAS2 instructions use locked accesses for operands in memory and for updating translation table entries during table search operations.

4.3.3 Special Accesses

Several other processor operations result in accesses that have special caching characteristics besides those with an implied cache-inhibited access in the serialized mode. Exception stack accesses, and exception vector fetches, that miss in the cache do not allocate cache lines in the data cache, preventing replacement of a cache line. Cache hits by these accesses are handled in the normal manner according to the caching mode specified for the accessed address.

Accesses by the MOVE16 instruction also do not allocate cache lines in the data cache for either read or write misses. Read hits on either valid or dirty cache lines are read from the cache. Write hits invalidate a matching line and perform an external access. Interacting with the cache in this manner prevents a large block move or block initialization implemented with a MOVE16 from being cached, since the data may not be needed immediately.

If the data cache is re-enabled after a locked access has hit and the data cache was disabled, the next non-locked access that results in a data cache miss will not be cached.

4.4 CACHE PROTOCOL

The cache protocol for processor and snooped accesses is described in the following paragraphs. In all cases, an external bus transfer will cause a cache line state to change only if the bus transfer is marked as snoopable on the bus. The protocols described in the following paragraphs assume that the data is cachable (i.e., write-through and copyback).

4.4.1 Read Miss

A processor read that misses in the cache causes the cache controller to request a bus transaction that reads the needed line from memory and supplies the required data to the IU. The line is placed in the cache in the valid state. Snooped external reads that miss in the cache have no affect on the cache.

4.4.2 Write Miss

The cache controller handles processor writes that miss in the cache differently for write-through and copyback. Write misses to copyback memory cause the processor to perform a bus transaction that writes the needed cache line into its cache from memory in the same manner as for a read miss. The new cache line is then updated with the write data, and the D-bits are set for each long word that has been modified, leaving the cache line in the dirty state. Write misses to write-through memory write directly to memory without

loading the corresponding cache line in the cache. Snooped external writes that miss in the cache have no affect on the cache.

4.4.3 Read Hit

The cache controller handles processor reads that hit in the cache differently for write-through and copyback modes. No bus transaction is performed, and the state of the cache line does not change. Physical address bit 3 selects either the upper or lower half-line containing the required operand. This half-line is driven onto the internal bus. If the required data is allocated entirely within the half-line, only one access into the cache is required. Because the organization of the cache does not allow selection of more than one half-line at a time, misalignment across a half-line boundary requires two accesses into the cache.

A snooped external read that hits in the cache is ignored if the cache line is valid. If the snooped access hits a dirty line, memory is inhibited from responding, and the data is sourced from the cache directly to the alternate bus master. A snooped read hit does not change the state of the cache line unless the snooped access also indicates mark invalid, which causes the line to be invalidated after the access, even if it is dirty. Alternate bus masters should indicate mark invalid only for line reads to ensure the entire line is transferred before invalidating.

4.4.4 Write Hit

The cache controller handles processor writes that hit in the cache differently for write-through and copyback modes. For write-through accesses, a processor write hit causes the cache controller to update the affected long-word entries in the cache line and to request an external memory write transfer to update memory. The cache line state does not change. A write-through access to a line containing dirty data constitutes a system programming error even if the D-bits for the line are unchanged. This situation can be avoided by pushing cache lines when a ACR is changed and ensuring that alternate bus masters indicate the appropriate snoop operation for writes to corresponding memory (i.e., mark invalid for write-through and sink data for copyback). If the access is copyback, the cache controller updates the cache line and sets the D-bit for of the appropriate long words in the cache line. An external write is not performed, and the cache line state changes to, or remains in, the dirty state.

An alternate bus master can drive the SCx signals for a write access with an encoding that indicates to the MCF5102 that it should sink the data, inhibit memory, and respond as a slave if the access hits in the cache. The cache operation depends on the access size and current line state. A snooped line write that hits a valid line always causes the corresponding cache line to be invalidated. For snooped writes of byte, word, or long-word size that hit a dirty line, the processor inhibits memory and responds to the alternate bus master as a slave, sinking the data. Data received from the alternate bus master is written to the appropriate long word in the cache line, and the D-bit is set for that entry. The cache controller invalidates a cache line if the snoop control pins have indicated that a matching cache line is marked invalid for a snoop write.

4.5 CACHE COHERENCY

The MCF5102 provides several different mechanisms to assist in maintaining cache coherency in multimaster systems. Both write-through and copyback memory update techniques are supported to maintain coherency between the data cache and memory.

Alternate bus master accesses can reference data that the MCF5102 caches, causing coherency problems if the accesses are not handled properly. The MCF5102 snoops the bus during alternate bus master transfers. If a write access hits in the cache, the MCF5102 can update its internal caches, or if a read access hits, it can intervene in the access to supply dirty data. Caches can be snooped even if they are disabled. The alternate bus master controls snooping through the snoop control signals, indicating which access can be snooped and the required operation for snoop hits. Table 4-1 lists the requested snoop operation for each encoding of the snoop control signals. Since the processor and the bus snooper must both access the caches, the snoop controller has priority over the processor for snoopable accesses to maintain cache coherency.

Table 4-1. Snoop Control Encoding

		Requested Snoop Operation	
SC1	**SC0**	**Alternate Bus Master Read Access**	**Alternate Bus Master Write Access**
0	0	Inhibit Snooping	Inhibit Snooping
0	1	Supply Dirty Data and Leave Dirty Data	Sink Byte/Word/Long/Long Word
1	0	Supply Dirty Data and Mark Line Invalid	Invalidate Line
1	1	Reserved (Snoop Inhibited)	Reserved (Snoop Inhibited)

The snooping protocol and caching mechanism supported by the MCF5102 are optimized to support multimaster systems with the MCF5102 as the single caching master. In systems implementing multiple MC68040s as bus masters, shared data should be stored in write-through memory. This procedure allows each processor to cache shared data for read access while forcing a processor write to shared data to appear as an external write to memory, which the other processors can snoop.

If shared data is stored in copyback memory, only one processor at a time can cache the data since writes to copyback memory do not access the external bus. If a processor accesses shared data cached by another processor, the slave can source the data to the master without invalidating its own copy only if the transfer to the master is cache inhibited. For the master processor to cache the data, it must force invalidation of the slave processor’s copy of the data (by specifying mark invalid for the snoop operation), and the ACR must monitor the data transfer between the processors and update memory with the transferred data. The memory update is required since the master processor is unaware of the sourced data (valid data from memory or dirty data from a snooping processor) and initially creates a valid cache line, losing dirty status if a snooping processor supplies the data.

Coherency between the instruction cache and the data cache must be maintained in software since the instruction cache does not monitor data accesses. Processor writes

that modify code segments access memory through the ACR. Because the instruction cache does not monitor these data accesses, stale data occurs in the instruction cache if the corresponding data in memory is modified. Invalidating instruction cache lines before writing to the corresponding memory lines can prevent this coherency problem, but only if the data cache line is in write-through mode and is marked serialized. A cache coherency problem could arise if the data cache line is configured as copyback and no serialization is done.

To fully support self-modifying code in any situation, it is imperative that a CPUSHA instruction be executed before the execution of the first self-modified instruction. The CPUSHA instruction has the effect of ensuring that there is no stale data in memory, the pipeline is flushed, and instruction prefetches are repeated and taken from external memory.

4.6 MEMORY ACCESSES FOR CACHE MAINTENANCE

The cache controller in each ACR performs all maintenance activities that supply data from the cache to the execution units. The activities include requesting accesses to the bus interface unit for reading new cache lines and writing dirty cache lines to memory. The following paragraphs describe the memory accesses resulting from cache fill operations (by both caches) and push operations (by the data cache). Refer to **Section 7 Bus Operation** for detailed information about the bus cycles required.

4.6.1 Cache Filling

When a new cache line is required, the cache controller requests a line read from the bus controller. The bus controller requests a burst read transfer by indicating a line access with the size signals (SIZ1, SIZ0) and indicates which line in the set is being loaded with the transfer line number signals (TLN1, TLN0). TLN1 and TLN0 are undefined for the instruction cache. These pins indicate the appropriate line numbers for data cache transfers only. Table 4-2 lists the definition of the TLNx encoding.

Table 4-2. TLNx Encoding

TLN1	TLN0	Line
0	0	Zero
0	1	One
1	0	Two
1	1	Three

The responding device sequentially supplies four long words of data and can assert the transfer cache inhibit signal ($\overline{\text{TCI}}$) if the line is not cachable. If the responding device does not support the burst mode, it should assert the $\overline{\text{TBI}}$ signal for the first long word of the line access. The bus controller responds by terminating the line access and completes the remainder of the line read as three, sequential, long-word reads.

Bus controller line accesses implicitly request burst mode operations from external memory. To operate in the burst mode, the device or external hardware must be able to increment the low-order address bits as described in **Section 7 Bus Operation**. The device indicates its ability to support the burst access by acknowledging the initial long-word transfer with transfer acknowledge ($\overline{TA}$) asserted and $\overline{TBI}$ negated. This procedure causes the processor to continue to drive the address and bus control signals and to latch a new data value for the cache line at the completion of each subsequent cycle (as defined by $\overline{TA}$) for a total of four cycles. The bursting mechanism requires addresses to wrap around so that the entire four long words in the cache line are filled in a single operation.

When a cache line read is initiated, the first cycle attempts to load the line entry corresponding to the instruction half-line or data item requested by the IU. Subsequent transfers are for the remaining entries in the cache line. In the case of a misaligned access in which the operand spans two line entries, the first cycle corresponds to the line entry containing the portion of the operand at the lower address.

The cache controller temporarily stores the data from each cycle in a line read buffer, where it is immediately available to the IU. If a misaligned access spans two entries in the line, the second portion of the operand is available to the IU as soon as the second memory cycle completes. A new IU access that hits the cache line being filled is also supplied data as soon as the required long word has been received from the bus controller. During the period required to fill the buffer, other IU accesses that hit in the cache are supplied data. This is vertical for a short cache-inhibited code loop that is less than eight bytes in length. Subsequent interactions of the loop hit in the buffer, but appear to hit in the cache since there is no external bus activity associated with the reads.

The assertion of $\overline{TCI}$ during the first cycle of a burst read operation inhibits loading of the buffered line into the cache, but it does not cause the burst transfer (or pseudo-burst transfer if $\overline{TBI}$ is asserted with $\overline{TCI}$) to be terminated early. The data placed in the buffer is accessible by the IU until the last long word of the burst is transferred from the bus controller, after which the contents of the buffer are invalidated without being copied into the cache. The assertion of $\overline{TCI}$ is ignored during the second, third, or fourth cycle of a burst operation and is ignored for write operations.

A bus error occurring during a burst operation causes the burst operation to abort. If the bus error occurs during the first cycle of a burst, the data from the bus is ignored. If the access is a data cycle, exception processing proceeds immediately. If the cycle is for an instruction prefetch, a bus error exception is pending. The bus error is processed only if the IU attempts to use either instruction word. Refer to **Section 7 Bus Operation** for more information about pipeline operation.

For either cache, when a bus error occurs on the second cycle or later, the burst operation is aborted and the line buffer is invalidated. The processor may or may not take an exception, depending on the status of the pending data request. If the bus error cycle contains a portion of a data operand that the processor is specifically waiting for (e.g., the second half of a misaligned operand), the processor immediately takes an exception. Otherwise, no exception occurs, and the cache line fill is repeated the next time data

within the line is required. In the case of an instruction cache line fill, the data from the aborted cycle is completely ignored.

On the initial access of a line read, a retry (indicated by the assertion of $\overline{TA}$ and $\overline{TEA}$) causes the bus controller to retry the bus cycle. However, a retry signaled during the remaining cycles of the line access (either burst or pseudo-burst) is recognized as a bus error, and the processor handles it as described in the previous paragraphs.

A cache inhibit or bus error on a line read can change the state of the line being replaced, even though the new line is not copied into the cache. Before loading a new line, the cache line being replaced is copied to the push buffer; if it is dirty, the cache line is invalidated. If a cache inhibit or bus error occurs on a replacement line read, a dirty line is restored to the cache from the push buffer. However, the line being replaced is not restored in the cache if it was originally valid and the cache line remains invalid. If the line read resulting from a write miss in copyback mode is cache inhibited, the write access misses in the cache and writes through to memory.

4.6.2 Cache Pushes

When the cache controller selects a dirty data cache line for replacement, memory must be updated with the dirty data before the line is replaced. This occurs when a CPUSH instruction execution explicitly selects the cache and when a cache inhibit access hits in the cache. To reduce the requested data's latency in the new line, the dirty line being replaced is temporarily placed in a push buffer while the new line is fetched from memory. When a line is allocated to the push buffer, an alternate bus master can snoop it, but the execution units cannot access it. After the bus transfer for the new line successfully completes, the dirty cache line is copied back to memory, and the push buffer is invalidated. If the operation to access the replacement line is abnormally terminated or signaled as cache inhibited, the line in the push buffer is copied back into its original position in the cache, and the processor continues operation as described in the previous paragraphs.

The number of dirty long words in the line to be pushed determines the size of the push transfer on the bus, minimizing bus bandwidth required for the push. A single long word is written to memory using a long-word push transfer if it is dirty. A push transfer is distinguished from a normal write transfer by an encoding of 000 on the transfer modifier signals (TM2–TM0) for the push. Asserting $\overline{TA}$ and $\overline{TEA}$ retries the transfer; a bus-error-asserted $\overline{TEA}$ terminates it. If a bus error terminates a push transfer, the processor immediately takes an exception.

A line containing two or more dirty long words is copied back to memory, using a line push transfer. For a line push, the bus controller requests a burst write transfer by indicating a line access with SIZ1 and SIZ0. The responding device sequentially accepts four long words of data. If the responding device does not support the burst mode, it should assert $\overline{TBI}$ for the first long word of the line access. The bus controller responds by terminating the line access and completes the remainder of the line push as three, sequential, long-word writes. The first cycle of the burst can be retried, but the bus controller interprets a

retry for any of the three remaining cycles as a bus error. If a bus error occurs in any cycle in the line push transfer, the processor immediately takes an exception.

A dirty cache line hit by a cache-inhibited access is pushed before the external bus access occurs. If the access is part of a locked transfer sequence for TAS, CAS, or CAS2 operand accesses or translation table updates, the $\overline{\text{LOCK}}$ signal is also asserted for the push access.

4.7 CACHE OPERATION SUMMARY

The instruction and data caches function independently when servicing access requests from the IU. The following paragraphs discuss the operational details for the caches and present state diagrams depicting the cache line state transitions.

4.7.1 Instruction Cache

The IU uses the instruction cache to store instruction prefetches as it requests them. Instruction prefetches are normally requested from sequential memory locations except when a change of program flow occurs (e.g., a branch taken) or when an instruction that can modify the status register (SR) is executed, in which case the instruction pipe is automatically flushed and refilled. The instruction cache supports a line-based protocol that allows individual cache lines to be in either the invalid or valid states.

For instruction prefetch requests that hit in the cache, the half-line selected by physical address bit 3 is multiplexed onto the internal instruction data bus. When an access misses in the cache, the cache controller requests the line containing the required data from memory and places it in the cache. If available, an invalid line is selected and updated with the tag and data from memory. The line state then changes from invalid to valid by setting the V-bit. If all lines in the set are already valid, a pseudo-random replacement algorithm is used to select one of the four cache lines replacing the tag and data contents of the line with the new line information. Figure 4-4 illustrates the instruction-cache line state transitions resulting from processor and snoop controller accesses. Transitions are labeled with a capital letter, indicating the previous state, followed by a number indicating the specific case listed in Table 4-3.

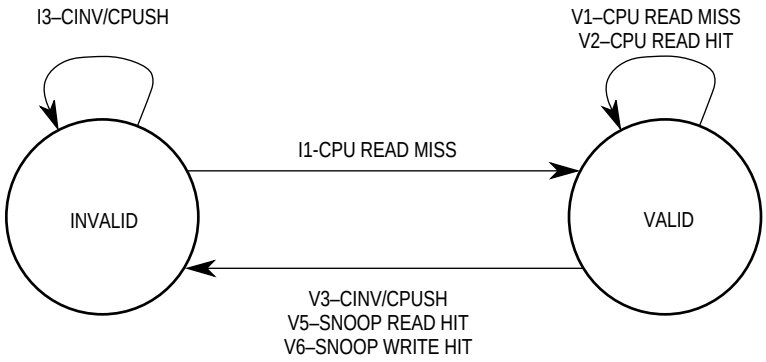


Figure 4-4. Instruction-Cache Line State Diagram

Table 4-3. Instruction-Cache Line State Transitions

Cache Operation	Current State			
	Invalid Cases		Valid Cases	
CPU Read Miss	I1	Read line from memory; supply data to CPU and update cache; go to valid state.	V1	Read line from memory; supply data to CPU and update cache (replacing old line); remain in current state.
CPU Read Hit	I2	Not Possible	V2	Supply data to CPU; remain in current state.
Cache Invalidate or Push (CINV or CPUSH)	I3	No action; remain in current state.	V3	No action; go to invalid state.
Alternate Master Read Hit (Snoop Control = 01 — Leave Dirty)	I4	Not possible; not snooped.	V4	Not possible; not snooped.
Alternate Master Read Hit (Snoop Control = 10 — Invalidate)	I5	Not Possible	V5	No action; go to invalid state.
Alternate Master Write Hit (Snoop Control = 01 — Leave Dirty or Snoop Control = 10 — Invalidate)	I6	Not Possible	V6	No action; go to invalid state.

4.7.2 Data Cache

The IU uses the data cache to store operand data as it generates the data. The data cache supports a line-based protocol allowing individual cache lines to be in one of three states: invalid, valid, or dirty. To maintain coherency with memory, the data cache supports both write-through and copyback modes, specified by the CM field.

Read misses and write misses to copyback memory cause the cache controller to read a new cache line from memory into the cache. If available, an invalid line in the selected set is updated with the tag and data from memory. The line state then changes from invalid to valid by setting the V-bit for the line. If all lines in the set are already valid or dirty, the pseudo-random replacement algorithm is used to select one of the four lines and replace the tag and data contents of the line with the new line information. Before replacement, dirty lines are temporarily buffered and later copied back to memory after the new line has been read from memory. If a snoop access occurs before the buffered line is written to memory, the snoop controller snoops the buffer and the caches. Figure 4-5 illustrates the three possible states for a data cache line, with the possible transitions caused by either the processor or snooped accesses. Transitions are labeled with a capital letter, indicating the previous state, followed by a number indicating the specific case listed in Table 4-4.

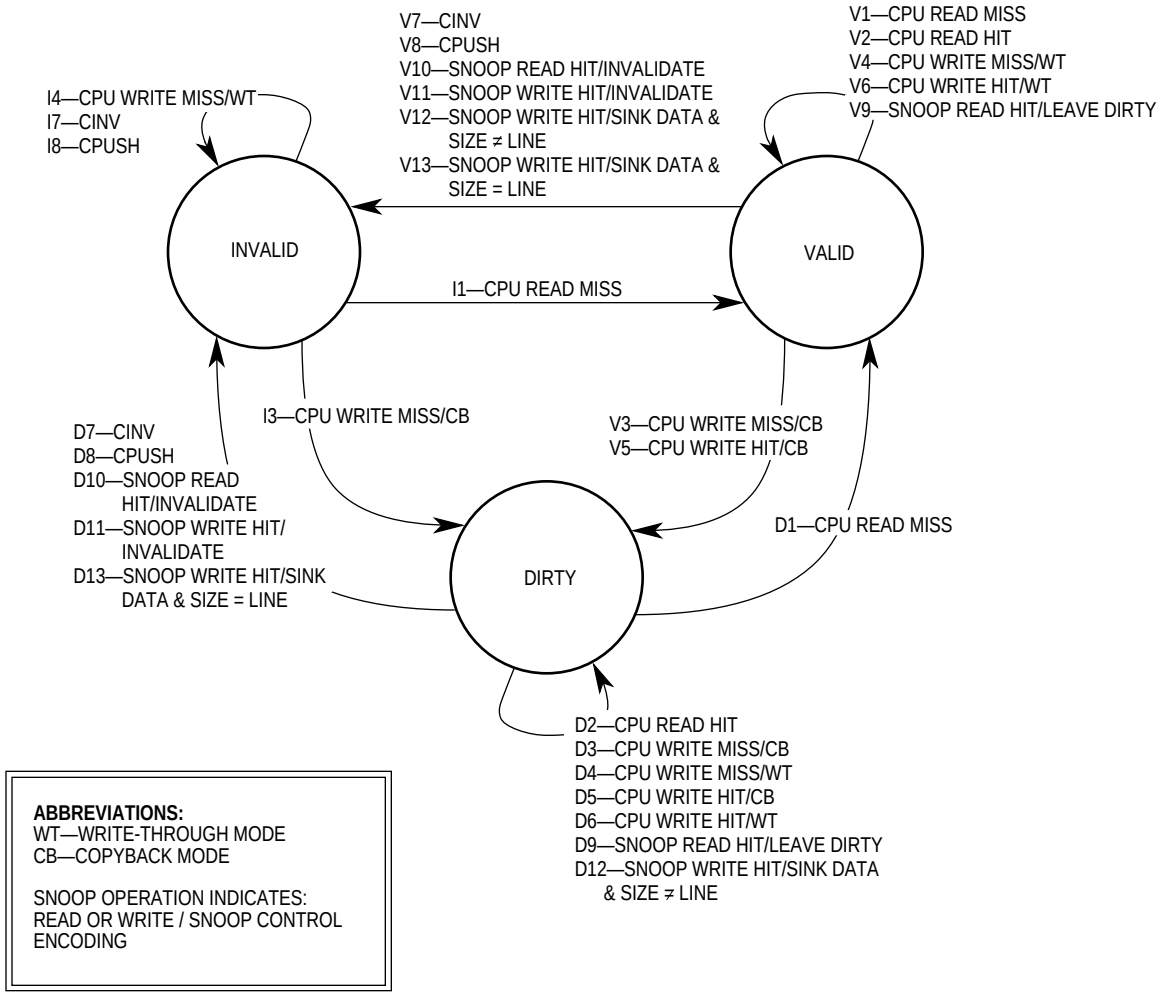


Figure 4-5. Data-Cache Line State Diagram

Table 4-4. Data-Cache Line State Transitions

Cache Operation	Current State					
	Invalid Cases		Valid Cases		Dirty Cases	
CPU Read Miss	I1	Read line from memory; supply data to CPU and update cache; go to valid state.	V1	Read line from memory; supply data to CPU and update cache (replacing old line); remain in current state.	D1	Buffer dirty cache line; read new line from memory; supply data to CPU and update cache; write buffered dirty data to memory; go to valid state.
CPU Read Hit	I2	Not Possible	V2	Supply data to CPU; remain in current state.	D2	Supply data to CPU; remain in current state.
CPU Write Miss (Copyback)	I3	Read line from memory into cache; write data to cache; set Dn bits of modified long words; go to dirty state.	V3	Read line from memory into cache (replacing old line); write data to cache and set Dn bits; go to dirty state.	D3	Buffer dirty cache line; read new line from memory; write data to cache and set Dn bits; write buffered dirty data to memory; remain in current state.
CPU Write Miss (Write-through)	I4	Write data to memory; remain in current state.	V4	Write data to memory; remain in current state.	D4	Write data to memory; remain in current state (see note).
CPU Write Hit (Copyback)	I5	Not Possible	V5	Write data into cache; set Dn bits of modified long words; go to dirty state.	D5	Write data in cache; set Dn bits of modified long words; remain in current state.
CPU Write Hit (Write-through)	I6	Not Possible	V6	Write data to cache; write data to memory; remain in current state.	D6	Write data into cache (no change to Dn bits); write data to memory; remain in current state (see note).
Cache Invalidate (CINV)	I7	No action; remain in current state.	V7	No action; go to invalid state.	D7	No action (dirty data lost); go to invalid state.
Cache Push (CPUSH)	I8	No action; remain in current state.	V8	No action; go to invalid state.	D8	Write dirty data to memory; go to invalid state.
Alternate Master Read Hit (Snoop Control = 01 — Leave Dirty)	I9	Not Possible	V9	No action; remain in current state.	D9	Inhibit memory and source data; remain in current state.

NOTE: Dirty state transitions D4 and D6 are the result of a system programming error and should be avoided even though they are technically valid.

Table 4-4. Data-Cache Line State Transitions (Continued)

Cache Operation	Current State					
	Invalid Cases		Valid Cases		Dirty Cases	
Alternate Master Read Hit (Snoop Control = 10 — Invalidate)	I10	Not Possible	V10	No action; go to invalid state.	D10	Inhibit memory and source data; go to invalid state
Alternate Master Write Hit (Snoop Control = 10 —Invalidate)	I11	Not Possible	V11	No action; go to invalid state.	D11	No action; go to invalid state.
Alternate Master Write Hit (Snoop Control = 01 — Sink Data and Size ≠ Line)	I12	Not Possible	V12	No action; go to invalid state.	D12	Inhibit memory and sink data; set Dn bits of modified long words; remain in current state.
Alternate Master Write Hit (Snoop Control = 01 — Sink Data and Size = Line)	I13	Not Possible	V13	No action; go to invalid state.	D13	No action; go to invalid state.

SECTION 5
SIGNAL DESCRIPTION

This section contains brief descriptions of the input and output signals in their functional groups (see Figure 5-1). Each signal's function is briefly explained, referencing other sections that contain detailed information about the signal and related operations. Table 5-1 lists the signal names, mnemonics, and functional descriptions of the input and output signals for the MCF5102. Timing specifications for these signals can be found in **Section 10 MCF5102 Electrical and Thermal Characteristics**.

NOTES

Assertion and *negation* are used to specify forcing a signal to a particular state. *Assertion* and *assert* refer to a signal that is active or true. *Negation* and *negate* refer to a signal that is inactive or false. These terms are used independent of the voltage level (high or low) that they represent.

Table 5-1. Signal Index

Signal Name	Mnemonic	Function
Address/Data Bus	A31/D31–A0/D0	32-bit address/data bus used to address 4-Gbytes of memory and 32-bit data bus used to transfer up to 32 bits of data per bus transfer.
Transfer Type	TT1,TT0	Indicates the general transfer type: normal, MOVE16, alternate logical function code, and acknowledge.
Transfer Modifier	TM2–TM0	Indicates supplemental information about the access.
Transfer Line Number	TLN1,TLN0	Indicates which cache line in a set is being pushed or loaded by the current line transfer.
Read/Write	R/$\overline{W}$	Identifies the transfer as a read or write.
Transfer Size	SIZ1,SIZ0	Indicates the data transfer size. These signals, together with A0 and A1, define the active sections of the data bus.
Bus Lock	$\overline{LOCK}$	Indicates a bus transfer is part of a read-modify-write operation, and the sequence of transfers should not be interrupted.
Cache Inhibit Out	$\overline{CIOUT}$	Indicates the processor will not cache the current bus transfer.
Transfer Start	$\overline{TS}$	Indicates the beginning of a bus transfer.
Transfer Acknowledge	$\overline{TA}$	Asserted to acknowledge a bus transfer.
Transfer Error Acknowledge	$\overline{TEA}$	Indicates an error condition exists for a bus transfer.
Transfer Cache Inhibit	$\overline{TCI}$	Indicates the current bus transfer should not be cached.
Transfer Burst Inhibit	$\overline{TBI}$	Indicates the slave cannot handle a line burst access.
Snoop Control	SC1,SC0	Indicates the snooping operation required during an alternate master access.
Memory Inhibit	$\overline{MI}$	Inhibits memory devices from responding to an alternate master access during snooping operations.
Bus Request	$\overline{BR}$	Asserted by the processor to request bus mastership.
Bus Grant	$\overline{BG}$	Asserted by an arbiter to grant bus mastership to the processor.
Bus Busy	$\overline{BB}$	Asserted by the current bus master to indicate it has assumed ownership of the bus.
Cache Disable	$\overline{CDIS}$	Dynamically disables the internal caches to assist emulator support.
Reset In	$\overline{RSTI}$	Processor reset.
Reset Out	$\overline{RSTO}$	Asserted during execution of a RESET instruction to reset external devices.
Interrupt Priority Level	$\overline{IPL2}$–$\overline{IPL0}$	Provides an encoded interrupt level to the processor.
Interrupt Pending	$\overline{IPEND}$	Indicates an interrupt is pending.
Autovector	$\overline{AVEC}$	Used during an interrupt acknowledge transfer to request internal generation of the vector number.
Processor Status	PST3–PST0	Indicates internal processor status.
Bus Clock	BCLK	Clock input used to derive all bus signal timing.

Table 5-1. Signal Index (Continued)

Signal Name	Mnemonic	Function
Test Clock	TCK	Clock signal for the IEEE 1149.1 Test Access Port (TAP).
Test Mode Select	TMS	Selects the principle operations of the test-support circuitry.
Test Data Input	TDI	Serial data input for the TAP.
Test Data Output	TDO	Serial data output for the TAP.
Wait State Pin	WAITER	Delays Transfer Start by One BCLK after read bus cycle.
Three-State Control	$\overline{Z}$	Three-State Control Pin Three-States all Signal Pins.
System CLK Disable	$\overline{SCD}$	System Clock Disable
Power Supply	V_{CC}	Power supply.
Ground	GND	Ground connection.

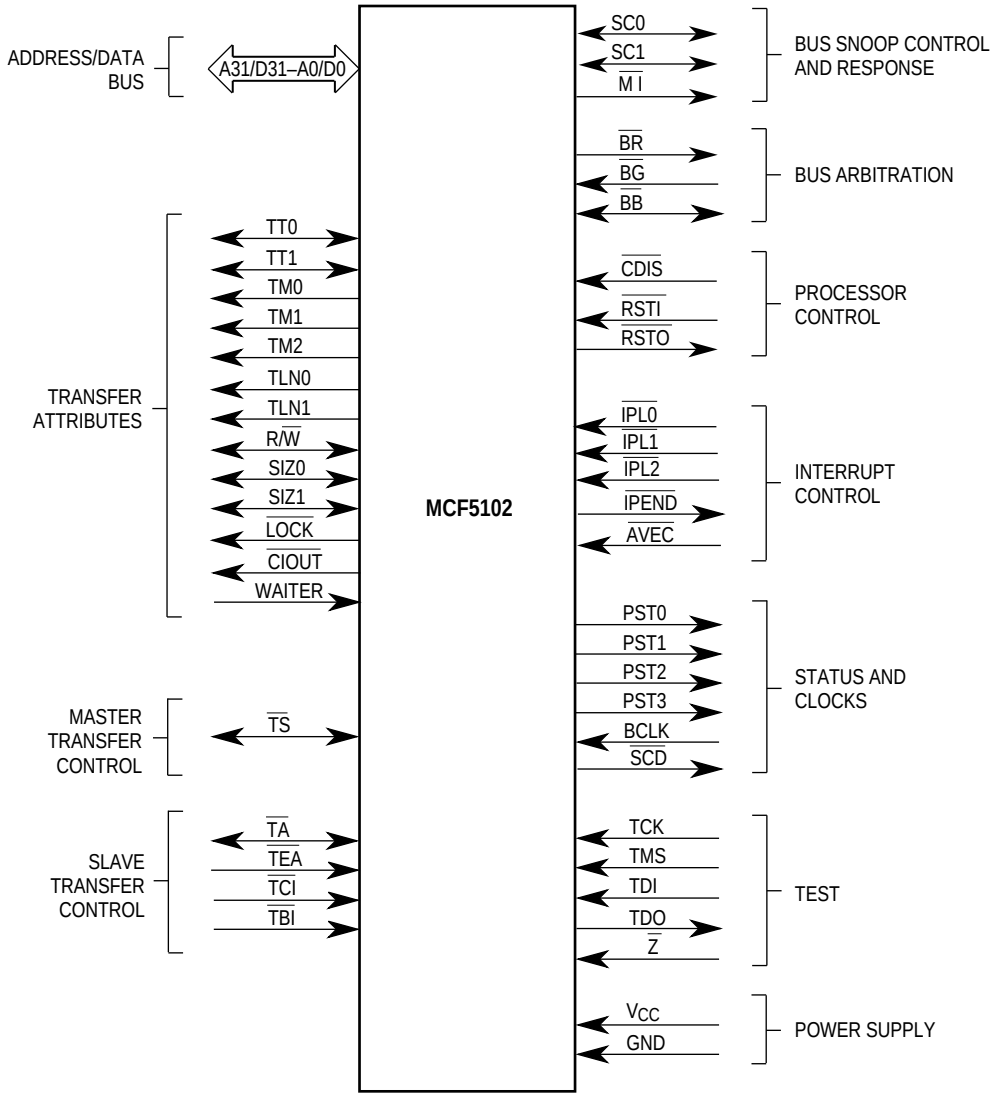


Figure 5-1. Functional Signal Groups

5.1 ADDRESS/DATA BUS (A31/D31–A0/D0)

These multiplexed three-state bidirectional signals provide the address of the first item of a bus transfer (except for acknowledge transfers) when the MCF5102 is the bus master. When an alternate bus master is controlling the bus, the processor examines (snoops) these signals to determine whether the processor should intervene in the access to maintain cache coherency. These signals also provide the general-purpose data path between the MCF5102 and all other devices. The data bus can transfer 8, 16, or 32 bits of data per bus transfer. During a burst transfer, the data lines are time-multiplexed to carry all 128 bits of the burst request using four 32-bit transfers.

5.2 TRANSFER ATTRIBUTE SIGNALS

The following paragraphs describe the transfer attribute signals, which provide additional information about the bus transfer.

5.2.1 Transfer Type (TT1, TT0)

The processor drives these three-state bidirectional signals to indicate the type of access for the current bus transfer. During bus transfers by an alternate bus master, the processor samples these signals to determine if it should snoop the transfer; only normal and MOVE16 accesses can be snooped. Table 5-2 lists the definition of the transfer-type encoding. The acknowledge access (TT1 = 1 and TT0 = 1) is used for both interrupt and breakpoint acknowledge transfers, and for LPSTOP broadcast cycles.

Table 5-2. Transfer-Type Encoding

TT1	TT0	Transfer Type
0	0	Normal Access
0	1	MOVE16 Access
1	0	Alternate Logical Function Code Access
1	1	Acknowledge Access

5.2.2 Transfer Modifier (TM2–TM0)

These three-state outputs provide supplemental information for each transfer type. Table 5-3 lists the encoding for normal and MOVE16 transfers, and Table 5-4 lists the encoding for alternate access transfers. For interrupt acknowledge transfers, the TMx signals carry the interrupt level being acknowledged; for breakpoint acknowledge transfers and LPSTOP broadcast cycles, the TMx signals are low. When the MCF5102 is not the bus master, the TMx signals are set to a high-impedance state.

Table 5-3. Normal and MOVE16 Access Transfer Modifier Encoding

TM2	TM1	TM0	Transfer Modifier
0	0	0	Data Cache Push Access
0	0	1	User Data Access*
0	1	0	User Code Access
0	1	1	Reserved
1	0	0	Reserved
1	0	1	Supervisor Data Access*
1	1	0	Supervisor Code Access
1	1	1	Reserved

* MOVE16 accesses use only these encodings.

Table 5-4. Alternate Access Transfer Modifier Encoding

TM2	TM1	TM0	Transfer Modifier
0	0	0	Logical Function Code 0
0	0	1	Reserved
0	1	0	Reserved
0	1	1	Logical Function Code 3
1	0	0	Logical Function Code 4
1	0	1	Reserved
1	1	0	Reserved
1	1	1	Logical Function Code 7

5.2.3 Transfer Line Number (TLN1, TLN0)

These three-state outputs indicate which line in the set of four data cache lines is being accessed for normal push and line data read accesses. TLNx signals are undefined for all other accesses to instruction space and are placed in a high-impedance state when the processor relinquishes the bus.

The TLNx signals can be used in high-performance systems to build an external snoop filter with a duplicate set of cache tags. The TLNx signals and address bus provide a direct indication of the state of the data caches and can be used to help maintain the duplicate tag store. The TLNx pins do not indicate the correct TLN number when an instruction cache burst fill occurs. Refer to **Section 4 Instruction and Data Caches** Table 4-2.

5.2.4 Read/Write (R/$\overline{W}$)

This bidirectional three-state signal defines the data transfer direction for the current bus cycle. A high level indicates a read cycle, and a low level indicates a write cycle. The bus snoop controller examines this signal when the processor is not the bus master.

5.2.5 Transfer Size (SIZ1, SIZ0)

These bidirectional three-state signals indicate the data size for the bus transfer. The bus snoop controller examines this signal when the processor is not the bus master. Refer to **Section 5 Signal Description** for settings.

5.2.6 Lock ($\overline{\text{LOCK}}$)

This three-state output indicates that the current transfer is part of a sequence of locked transfers for a read-modify-write operation. The external arbiter can use $\overline{\text{LOCK}}$ to prevent an alternate bus master from gaining control of the bus and accessing the same operand between processor accesses for the locked sequence of transfers. Although $\overline{\text{LOCK}}$ indicates that the processor requests the bus be locked, the processor will give up the bus if the external arbiter negates the $\overline{\text{BG}}$ signal. When the MCF5102 is not the bus master, the $\overline{\text{LOCK}}$ signal is set to a high-impedance state. $\overline{\text{LOCK}}$ drives high before three-stating.

5.2.7 Cache Inhibit Out ($\overline{\text{CIOUT}}$)

This three-state output reflects the state of the cache mode field in one of the ACR's and is asserted for accesses to noncachable memory to indicate that an external cache should ignore the bus transfer. When the MCF5102 is not the bus master, the $\overline{\text{CIOUT}}$ signal is set to a high impedance state.

5.3 BUS TRANSFER CONTROL SIGNALS

The following signals provide control functions for bus transfers.

5.3.1 Transfer Start ($\overline{\text{TS}}$)

The processor asserts this three-state bidirectional signal for one clock period to indicate the start of each transfer. During alternate bus master accesses, the processor monitors this signal to detect the start of each transfer to be snooped.

5.3.2 Transfer Acknowledge ($\overline{\text{TA}}$)

This three-state bidirectional signal indicates the completion of a requested data transfer operation. During transfers by the MCF5102, $\overline{\text{TA}}$ is an input signal from the referenced slave device indicating completion of the transfer. During alternate bus master accesses, $\overline{\text{TA}}$ is normally three-stated to allow the referenced slave device to respond, and the MCF5102 samples it to detect the completion of each bus transfer. The MCF5102 can inhibit memory and intervene in the access to source or sink data in its internal caches by asserting $\overline{\text{TA}}$ to acknowledge the data transfer. This capability applies to alternate bus master accesses that reference modified (dirty) data in the MCF5102 caches.

5.3.3 Transfer Error Acknowledge ($\overline{\text{TEA}}$)

The current slave asserts this input signal to indicate an error condition for the bus transaction. When asserted with $\overline{\text{TA}}$, this signal indicates that the processor should retry

the access. During alternate bus master accesses, the MCF5102 samples $\overline{TEA}$ to detect completion of each bus transfer.

5.3.4 Transfer Cache Inhibit ($\overline{TCI}$)

This input signal inhibits read data from being loaded into the MCF5102 instruction or data caches. $\overline{TCI}$ is ignored during all writes and after the first data transfer for both burst line reads and burst-inhibited line reads. $\overline{TCI}$ is also ignored during all alternate bus master transfers.

5.3.5 Transfer Burst Inhibit ($\overline{TBI}$)

This input signal indicates to the processor that the accessed device cannot support burst mode accesses and that the requested line transfer should be divided into individual long-word transfers. Asserting $\overline{TBI}$ with $\overline{TA}$ terminates the first data transfer of a line access, which causes the processor to terminate the burst and access the remaining data for the line as three successive long-word transfers. During alternate bus master accesses, the MCF5102 samples the $\overline{TBI}$ to detect completion of each bus transfer.

5.4 SNOOP CONTROL SIGNALS

The following signals control the operation of the MCF5102 on-chip snoop logic. **Section 4 Instruction and Data Caches** provides information about the relationship of the snoop control signals to the caches.

5.4.1 Snoop Control (SC1, SC0)

These input signals specify the snoop operation to be performed by the MCF5102 for an alternate bus master transfer. If the MCF5102 is allowed to snoop an alternate bus master read transfer, it can intervene in the access to supply data from its data cache when the memory copy is stale, ensuring that the alternate bus master receives valid data. Writes by an alternate bus master can also be snooped to either update the MCF5102 internal data cache with the new data or invalidate the matching cache lines, ensuring that subsequent MCF5102 reads access valid data. These signals are ignored when the processor is the bus master.

5.4.2 Memory Inhibit ($\overline{MI}$)

This output signal prevents an alternate bus master from accessing possibly stale data in memory while the MCF5102 is unable to respond. $\overline{MI}$ is asserted during reset preventing external memory from responding. When the SCx signals indicate an access should be snooped, the MCF5102 keeps $\overline{MI}$ asserted until it determines if intervention in the access is required. If no intervention is required, $\overline{MI}$ is negated and memory is allowed to respond to complete the access. Otherwise, $\overline{MI}$ remains asserted and the MCF5102 completes the transfer as a slave. It updates its caches on a write or supplies data to the alternate bus master on a read. $\overline{MI}$ is negated when the MCF5102 is the bus master. During a snoop cycle, the MCF5102 ignores all $\overline{TA}$ and $\overline{TEA}$ assertions while $\overline{MI}$ is asserted; when $\overline{RSTI}$ is asserted, $\overline{MI}$ is asserted.

5.5 ARBITRATION SIGNALS

The following control signals support requests to an external arbiter to become the bus master.

5.5.1 Bus Request ($\overline{BR}$)

This output signal indicates to the external arbiter that the processor needs to become bus master for one or more bus transfers. $\overline{BR}$ is negated when the MCF5102 begins an access to the external bus with no other accesses pending, and $\overline{BR}$ remains negated until another access is required. There are some situations in which the MCF5102 asserts $\overline{BR}$ and then negates it without having run bus transfers; this is a disregard request condition.

5.5.2 Bus Grant ($\overline{BG}$)

This input signal from an external arbiter indicates that the bus is available to the MCF5102 as soon as the current bus access completes. $\overline{BG}$ must be asserted and $\overline{BB}$ must be negated (indicating the bus is free) before the MCF5102 assumes ownership of the bus.

5.5.3 Bus Busy ($\overline{BB}$)

This three-state bidirectional signal indicates that the bus is currently owned. $\overline{BB}$ is monitored as a processor input to determine when a alternate bus master has released control of the bus. $\overline{BG}$ must be asserted and $\overline{BB}$ must be negated (indicating the bus is free) before the MCF5102 asserts $\overline{BB}$ as an output to assume ownership of the bus. The processor keeps $\overline{BB}$ asserted until the external arbiter negates $\overline{BG}$ and the processor completes the bus transfer in progress. When releasing the bus, the processor negates $\overline{BB}$, then sets it to a high-impedance state for use again as an input.

5.6 PROCESSOR CONTROL SIGNALS

The following signals control disabling caches and access control units (ACUs) and support processor and external device initialization.

5.6.1 Cache Disable ($\overline{CDIS}$)

$\overline{CDIS}$ dynamically disables the on-chip caches on the next internal cache access boundary. $\overline{CDIS}$ does not flush the data and instruction caches; entries remain unaltered and become available after $\overline{CDIS}$ is negated. The assertion of $\overline{CDIS}$ does not affect snooping. During a processor reset, the level on $\overline{CDIS}$ is latched and used to select the normal bus mode ($\overline{CDIS}$ high) or multiplexed bus mode ($\overline{CDIS}$ low). Refer to **Section 4 Instruction and Data Caches** for information about the caches and to **Section 7 Bus Operation** for information about the multiplexed bus mode.

5.6.2 Reset In ($\overline{RSTI}$)

This input signal causes the MCF5102 to enter reset exception processing. The $\overline{RSTI}$ signal is an asynchronous input that is internally synchronized to the next rising edge of

the BCLK signal. All three-state signals are set to the high-impedance state, and all outputs, except $\overline{MI}$, are negated when $\overline{RSTI}$ is recognized. The assertion of $\overline{RSTI}$ does not affect the test pins. Refer to **Section 7 Bus Operation** for a description of reset operation and to **Section 8 Exception Processing** for information about the reset exception.

5.6.3 Reset Out ($\overline{RSTO}$)

The MCF5102 asserts this output during execution of the RESET instruction to initialize external devices. Refer to **Section 7 Bus Operation** for a description of reset out bus operation.

5.7 INTERRUPT CONTROL SIGNALS

The following signals control the interrupt functions.

5.7.1 Interrupt Priority Level ($\overline{IPL2}$–$\overline{IPL0}$)

These input signals provide an indication of an interrupt condition and the encoding of the interrupt level from a peripheral or external prioritizing circuitry. $\overline{IPL2}$ is the most significant bit of the level number.

5.7.2 Interrupt Pending Status ($\overline{IPEND}$)

This output signal indicates that an interrupt request has been recognized internally and exceeds the current interrupt priority mask in the status register (SR). External devices (other bus masters) can use $\overline{IPEND}$ to predict processor operation on the next instruction boundaries. $\overline{IPEND}$ is not intended for use as an interrupt acknowledge to external peripheral devices.

5.7.3 Autovector ($\overline{AVEC}$)

This input signal is asserted with $\overline{TA}$ during an interrupt acknowledge transfer to request internal generation of the vector number.

5.8 STATUS AND CLOCK SIGNALS

The following paragraphs explain the signals that provide timing, test control, and the internal processor status.

5.8.1 Processor Status (PST3–PST0)

These outputs indicate the internal execution unit's status. The timing is synchronous with BCLK, and the status may have nothing to do with the current bus transfer. The PSTx signal is updated depending on the type of PSTx encoding. There are two classes of PSTx encodings. The first class is associated with instruction boundaries, and the second class indicates the processor's present status. Table 5-6 lists the definition of the encodings.

The encodings 0, 8, 4, 5, C, D, E, and F indicate the present status and do not reflect a specific stage of the pipe. These encodings persist as long as the processor stays in the indicated state. The default encoding 0 (user) or 8 (supervisor) is indicated if none of the above conditions apply. The encodings 1, 2, 3, 9, A, and B belong to the first class of PSTx encoding. This class indicates that the instruction is in its last instruction execution stage. These encodings exist for only one BCLK period per instruction and are mutually exclusive.

Table 5-5. Processor Status Encoding

Hex	PST3	PST2	PST1	PST0	Internal Status
0	0	0	0	0	User, Start/Continue Current Instruction
1	0	0	0	1	User, End Current Instruction
2	0	0	1	0	User, Branch Not Taken/End Current Instruction
3	0	0	1	1	User, Branch Taken/End Current Instruction
4	0	1	0	0	User, Table Search
5	0	1	0	1	Halted State (Double Bus Fault)
6	0	1	1	0	Low-Power Stop Mode (Supervisor Instruction)
7	0	1	1	1	Reserved
8	1	0	0	0	Supervisor, Start/Continue Current Instruction
9	1	0	0	1	Supervisor, End Current Instruction
A	1	0	1	0	Supervisor, Branch Not Taken/End Current Instruction
B	1	0	1	1	Supervisor, Branch Taken/End Current Instruction
C	1	1	0	0	Supervisor, Table Search
D	1	1	0	1	Stopped State (Supervisor Instruction)
E	1	1	1	0	RTE Executing
F	1	1	1	1	Exception Stacking

When a ‘branch taken/end current instruction’ is indicated, it means that a change of instruction flow is pending. Along with the following instructions, an exception stacking (encoding F) sequence is ended with the ‘supervisor, branch taken/end current instruction’ encoding as though it were a virtual JMP instruction. This includes all the possible exceptions listed in the processor’s vector table. Instructions that cause a ‘branch taken/end current instruction’ encoding when they are executed are as follows:

ANDI to SR	DBcc (Taken)	ORI to SR
Bcc (Taken)	JMP	RTD
BRA	JSR	RTE
BSR	MOVE to SR	RTR
CAS	MOVE USP	RTS
CAS2	MOVEC	STOP
CINV	MOVES	TAS
CPUSH	NOP	

The Bcc (not taken) and DBcc (not taken) are the only instructions that cause a ‘branch not taken/end current instruction’ encoding. All instructions and conditions end with the ‘end current instruction’ encoding. For instance, if the processor is running back-to-back single clock instructions, the encoding ‘end current instruction’ remains asserted for as many clock cycles as instructions.

The following examples are for PSTx encodings:

1. An access error terminates an instruction such that the instruction execution stage is not reached. In this case, an ‘end current instruction’ is not indicated. Exception processing starts, the exception stacking status is indicated, and then the virtual JMP causes the ‘supervisor, branch taken/end current instruction’ encoding.
2. Two simultaneous interrupt exception processing sequences follow an ADD instruction. The ADD instruction ends with ‘end current instruction’, followed by exception stacking, followed by ‘branch taken/end current instruction’, followed by exception stacking, followed by ‘branch taken/end current instruction’.
3. An RTE instruction follows an ADD instruction. The ‘end current instruction’ is followed by RTE executing followed by a branch taken/end current instruction.

5.8.2 Bus Clock (BCLK)

This input signal is used as a reference for all bus timing. It is a TTL-compatible signal and cannot be gated off.

5.9 TEST SIGNALS

The MCF5102 includes dedicated user-accessible test logic that is fully compatible with the IEEE 1149.1 *Standard Test Access Port and Boundary Scan Architecture*. Problems associated with testing high-density circuit boards have led to the development of this standard under the IEEE Test Technology Committee and Joint Test Action Group (JTAG) sponsorship. The MCF5102 implementation supports circuit board test strategies based on this standard. However, the JTAG interface is not intended to provide an in-circuit test to verify MCF5102 operations; therefore, it is impossible to test MCF5102 operations using this interface. **Section 6 IEEE 1149.1 Test Access Port (JTAG)** describes the MCF5102 implementation of the IEEE 1149.1 and is intended to be used with the supporting IEEE document.

5.9.1 Test Clock (TCK)

This input signal is used as a dedicated clock for the test logic. Since clocking of the test logic is independent of the normal operation of the MCF5102, several other components on a board can share a common test clock with the processor even though each component may operate from a different system clock. The design of the test logic allows the test clock to run at low frequencies, or to be gated off entirely as required for test purposes.

5.9.2 Test Mode Select (TMS)

This input signal is decoded by the TAP controller and distinguishes the principle operationas of the test support circuitry.

5.9.3 Test Data In (TDI)

This input signal provides a serial data input to the TAP.

5.9.4 Test Data Out (TDO)

This three-state output signal provides a serial data output from the TAP. The TDO output can be placed in a high-impedance mode to allow parallel connection of board-level test data paths.

5.9.5 Wait State Pin (WAITER)

While the WAITER pin is asserted (active high), transfer starts and the address phase of the multiplexed bus cycle is delayed one BCLK after a read bus cycle. This provides the user additional setup time to decode this information (see timing specifications). In this mode all transfers following read bus cycles will have a minimum of one idle clock added.

5.9.6 System Clock Disable ($\overline{SCD}$) Signal

When system clock disable is asserted this output signal indicates that the BCLK input can be disabled or changed in frequency. $\overline{SCD}$ is asserted upon termination of the LPSTOP broadcast cycle. BCLK levels and timing must be within specification when $\overline{SCD}$ is negated. $\overline{SCD}$ is negated with a valid interrupt or reset.

5.9.7 $\overline{Z}$ Signal

$\overline{Z}$ (Active Low) three-state control pin. When asserted, all input/outputs pins will be three-stated uncondition

5.10 POWER SUPPLY CONNECTIONS

The MCF5102 requires connection to a V_{CC} power supply, positive with respect to ground. The V_{CC} and ground connections are grouped to supply adequate current to the various sections of the processor.

5.11 SIGNAL SUMMARY

Table 5-7 provides a summary of the electrical characteristics of the signals discussed in this section.

Table 5-6. Signal Summary

Signal Name	Mnemonic	Type	Active	Three-State
Address/Data Bus	A31/D31–A0/D0	Input/Output	High	Yes
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Bus Busy	$\overline{\text{BB}}$	Input/Output	Low	Yes
Bus Clock	BCLK	Input	—	—
Bus Grant	$\overline{\text{BG}}$	Input	Low	—
Bus Request	$\overline{\text{BR}}$	Output	Low	No
Cache Disable	$\overline{\text{CDIS}}$	Input	Low	—
Cache Inhibit Out	$\overline{\text{CIOUT}}$	Output	Low	Yes
Interrupt Pending	$\overline{\text{IPEND}}$	Output	Low	No
Interrupt Priority Level	$\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$	Input	Low	—
Bus Lock	$\overline{\text{LOCK}}$	Output	Low	Yes
Memory Inhibit	$\overline{\text{MI}}$	Output	Low	No
Processor Status	PST3–PST0	Output	High	No
Read/Write	R/$\overline{\text{W}}$	Input/Output	High/Low	Yes
Reset In	$\overline{\text{RSTI}}$	Input	Low	—
Reset Out	$\overline{\text{RSTO}}$	Output	Low	No
Snoop Control	SC1, SC0	Input	High	—
Transfer Acknowledge	$\overline{\text{TA}}$	Input/Output	Low	Yes
Transfer Burst Inhibit	$\overline{\text{TBI}}$	Input	Low	—
Transfer Cache Inhibit	$\overline{\text{TCI}}$	Input	Low	—
Transfer Error Acknowledge	$\overline{\text{TEA}}$	Input	Low	—
Transfer Line Number	TLN1, TLN0	Output	High	Yes
Transfer Modifier	TM2–TM0	Output	High	Yes
Transfer Size	SIZ1, SIZ0	Input/Output	High	Yes
Transfer Start	$\overline{\text{TS}}$	Input/Output	Low	Yes
Transfer Type	TT1, TT0	Input/Output	High	Yes
Test Clock	TCK	Input	—	—
Test Data Input	TDI	Input	High	—
Test Data Output	TDO	Output	High	Yes
Test Mode Select	TMS	Input	High	—
Three-State Control Pin	$\overline{\text{Z}}$	Input	Low	—
Wait State Pin	WAITER	Input	High	—
Syatem Clock Disable	$\overline{\text{SCD}}$	Output	Low	—
Ground	GND	Ground	—	—
Power Supply	V_{CC}	Power	—	—

SECTION 6
IEEE 1149.1A TEST ACCESS PORT (JTAG)

The MCF5102 includes dedicated user-accessible test logic that is fully compatible with the IEEE standard 1149.1A *Standard Test Access Port and Boundary Scan Architecture*. Problems associated with testing high-density circuit boards have led to the standard's development under the sponsorship of the IEEE Test Technology Committee and the Joint Test Action Group (JTAG).

The following paragraphs are to be used in conjunction with the supporting IEEE document and includes those chip-specific items that the IEEE standard requires to be defined and additional information specific to the MCF5102 implementations. For details and application information regarding the standard, refer to the IEEE standard 1149.1A document.

The MCF5102 implementations support circuit board test based on the standard. The test logic utilizes static logic design and is system logic independent of the device. The MCF5102 implementations provide capabilities to:

a. Perform boundary scan operations to test circuit board electrical continuity,
b. Bypass the MCF5102 by reducing the shift register path to a single cell,
c. Sample the MCF5102 system pins during operation and transparently shift out the result,
d. Disable the output drive to output-only pins during circuit board testing.

NOTE

The IEEE standard 1149.1A test logic cannot be considered completely benign to those planning not to use this capability. Certain precautions must be observed to ensure that this logic does not interfere with system operation. Refer to **6.4 Disabling The IEEE Standard 1149.1A Operation**.

Figure 6-1 illustrates a block diagram of the MCF5102 implementations of IEEE standard 1149.1A. The test logic includes a 16-state dedicated TAP controller. These 16 controller states are defined in detail in the IEEE standard 1149.1A, but only 8 are included in this section.

Test-Logic-Reset	Run-Test/Idle
Capture-IR	Capture-DR
Update-IR	Update-DR
Shift-IR	Shift-DR

Four dedicated signal pins provides access to the TAP controller:

TCK—A test clock input that synchronizes the test logic.

TMS—A test mode select input with an internal pullup resistor sampled on the rising edge of TCK to sequence the TAP controller.

TDI—A test data input with an internal pullup resistor sampled on the rising edge of TCK.

TDO—A three-state test data output actively driven only in the shift-IR and shift-DR controller states that changes on the falling edge of TCK.

The test logic also includes an instruction shift register and two test data registers, a boundary scan register and a bypass register. The boundary scan register links all device signal pins into a chain that can be controlled by the 3-bit instruction shift register.

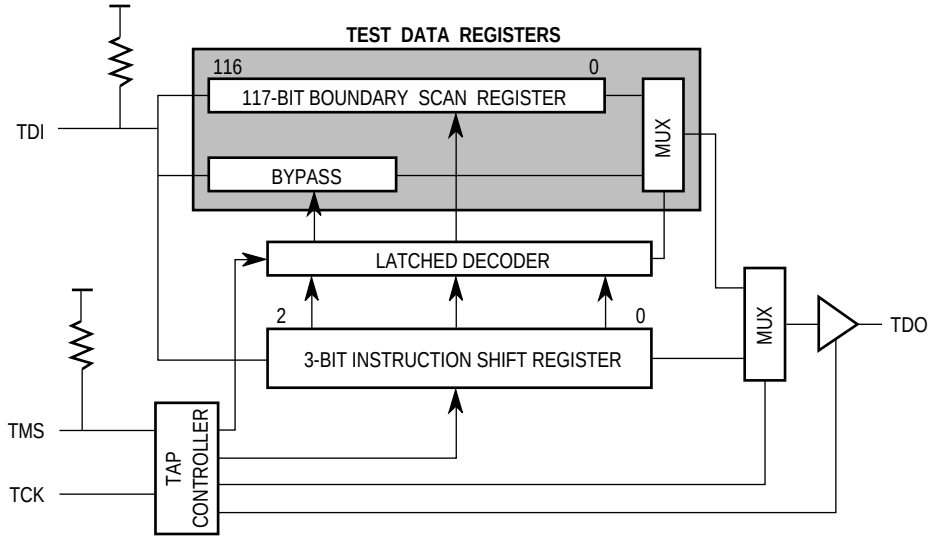


Figure 6-1. MCF5102 Test Logic Block Diagram

6.1 INSTRUCTION SHIFT REGISTER

The MCF5102 IEEE standard 1149.1A implementations include a 3-bit instruction shift register without parity. The register shifts one of six instructions, which can either select the test to be performed or access a test data register, or both. Data is transferred from the instruction shift register to latched decoded outputs during the update-IR state. The instruction shift register is reset to all ones in the TAP controller test-logic-reset state, which is equivalent to selecting the BYPASS instruction. During the capture-IR state, the binary value 001 is loaded into the parallel inputs of the instruction shift register.

The MCF5102 IEEE standard 1149.1A implementations include three mandatory standard public instructions (BYPASS, SAMPLE/PRELOAD, and EXTEST), two optional public standard instructions, and one manufacturer's private instruction. The five public instructions provide the capability to disable all device output drivers, operate the device in a BYPASS configuration, and conduct boundary scan test operations. Table 6-1 lists the three bits used in the instruction shift register to decode the instructions and

their related encodings. Note that the least significant bit of the instruction (bit 0) is the first bit to be shifted into the instruction shift register.

Table 6-1. IEEE Standard 1149.1A Instructions

Bit 2	Bit 1	Bit 0	Instruction Selected	Test Data Register Accessed
0	0	0	EXTEST	Boundary Scan
0	0	1	HIGHZ	Bypass
0	1	0	SAMPLE/PRELOAD	Boundary Scan
1	0	0	CLAMP	Bypass
1	1	0	PRIVATE	—
1	1	1	BYPASS	Bypass

6.1.1 EXTEST.

The external test instruction (EXTEST) selects the boundary scan register. This instruction also activates one internal function that is intended to protect the device from potential damage while performing boundary scan operations. EXTEST asserts internal reset for the MCF5102 system logic to force a predictable benign internal state.

6.1.2 HIGHZ.

The HIGHZ instruction is an optional instruction provided as a Motorola public instruction to anticipate the need to backdrive output pins during circuit board testing. The HIGHZ instruction asserts internal system reset, selects the bypass register, and forces all output and bidirectional pins to the high-impedance state.

Holding TMS high and clocking TCK for at least five rising edges causes the TAP controller to enter the test-logic-reset state. Using only the TMS and TCK pins and the capture-IR and update-IR states invokes the HIGHZ instruction. This scheme works because the value captured by the instruction shift register during the capture-IR state is identical to the HIGHZ opcode.

6.1.3 Sample/Preload.

The SAMPLE/PRELOAD instruction provides two separate functions. First, it provides a means to obtain a sample system data and control signal. Sampling occurs on the rising edge of TCK in the capture-DR state. The user can observe the data by shifting it through the boundary scan register to output TDO using the shift-DR state. Both the data capture and the shift operations are transparent to system operation. The user must provide some form of external synchronization to achieve meaningful results since there is no internal synchronization between TCK and BCLK.

The second function of the SAMPLE/PRELOAD instruction is to initialize the boundary scan register output cells before selecting EXTEST or CLAMP, which is accomplished by ignoring data being shifted out of TDO while shifting in initialization data. The update-DR state can then be used to initialize the boundary scan register and ensure that known data

and output state will occur on the outputs after entering the EXTEST or CLAMP instruction.

6.1.4 CLAMP.

The CLAMP instruction allows the state of the signals driven from the MCF5102 pins to be determined from the boundary scan register, while the bypass register is selected as the serial path between TDI and TDO. The signals driven from the MCF5102 pins do not change while the CLAMP instruction is selected.

6.1.5 BYPASS

The BYPASS instruction selects the single-bit bypass register, creating a single-bit shift-register path from TDI to the bypass register to TDO. The instruction enhances test efficiency when a component other than the MCF5102 becomes the device under test. When the bypass register is initially selected, the instruction shift register stage is set to a logic zero on the rising edge of TCK following entry into the capture-DR state. Therefore, the first bit to be shifted out after selecting the bypass register is always a logic zero. Figure 6-2 illustrates the bypass register.

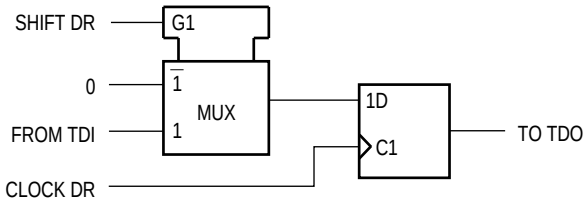


Figure 6-2. Bypass Register

6.2 BOUNDARY SCAN REGISTER

The 117-bit boundary scan register uses the TAP controller to scan user-defined values into the output buffers, capture values presented to input pins, and control the direction of bidirectional pins. The instruction shift register cell nearest TDO (i.e., first to be shifted out) is defined as bit zero. The last bit to be shifted out is bit 116. This register includes cells for all device signal pins and clock pins along with associated control signals.

The MCF5102 boundary scan register consists of three cell structure types, O.Latch, I.Pin, and IO.Ctl, that are associated with a boundary scan register bit. All boundary scan output cells capture the logic level of the device output latch during the capture-DR state. Figures 6-3 through 6-6 illustrate these three cell types. Figure 6-3 illustrates the general arrangement of these cells.

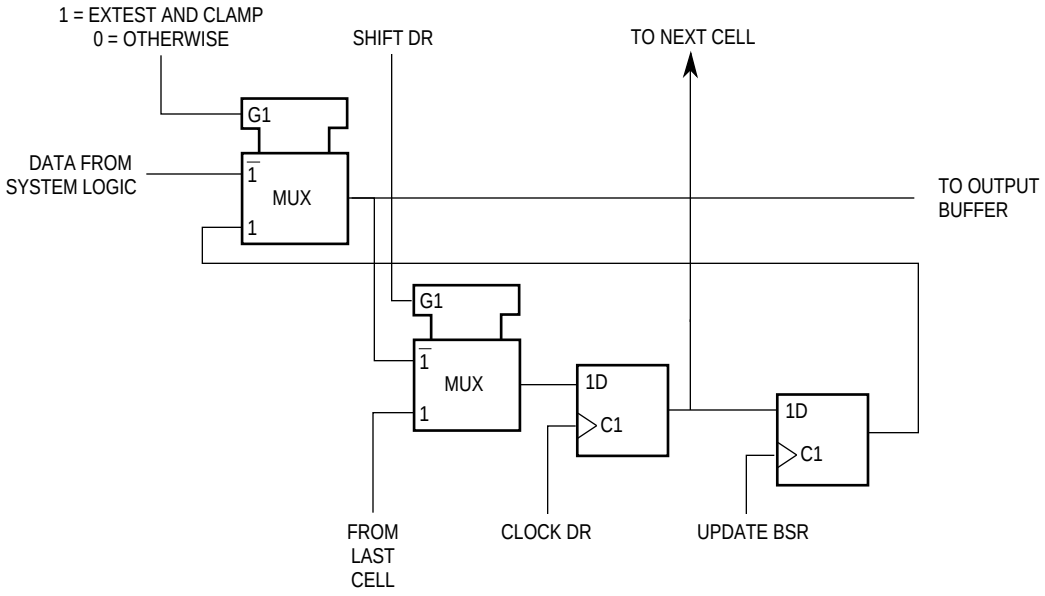


Figure 6-3. Output Latch Cell (O.Latch)

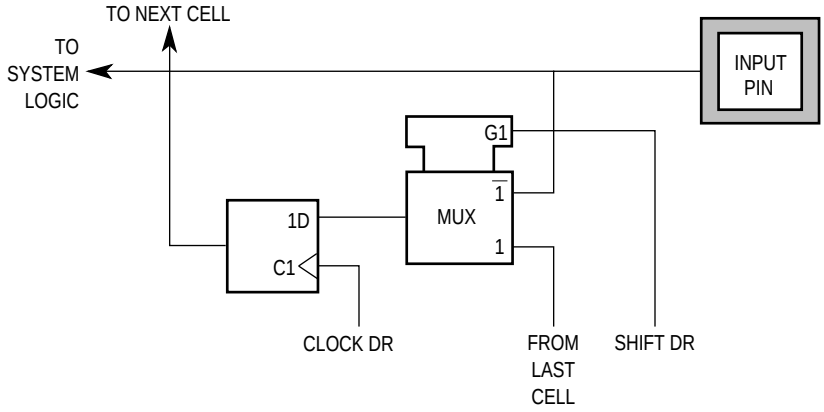


Figure 6-4. Input Pin Cell (I.Pin)

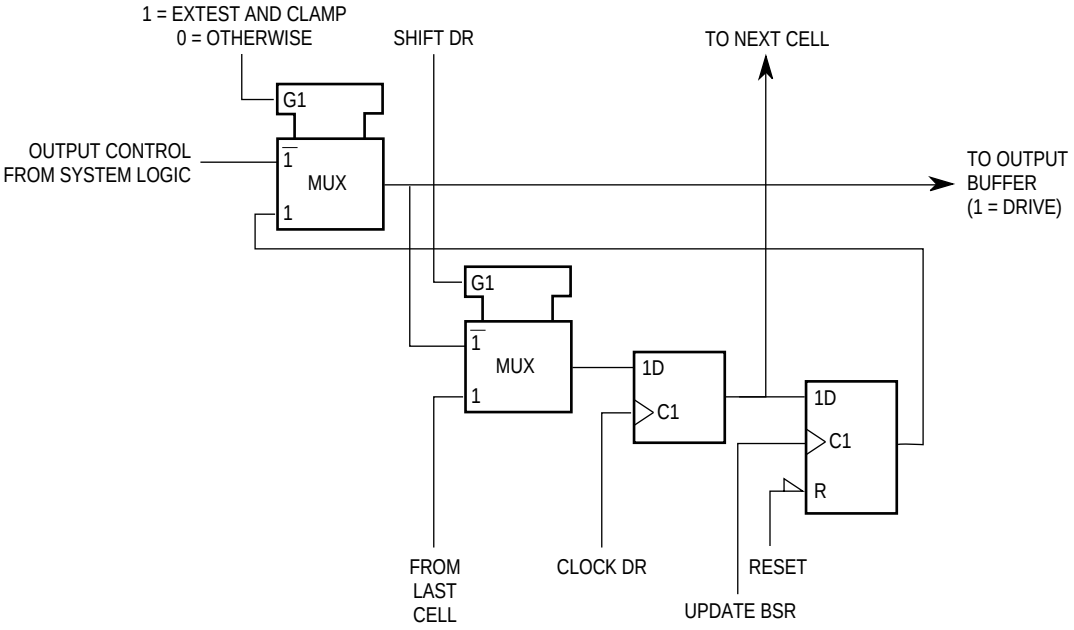


Figure 6-5. Output Control Cells (IO.Ctl)

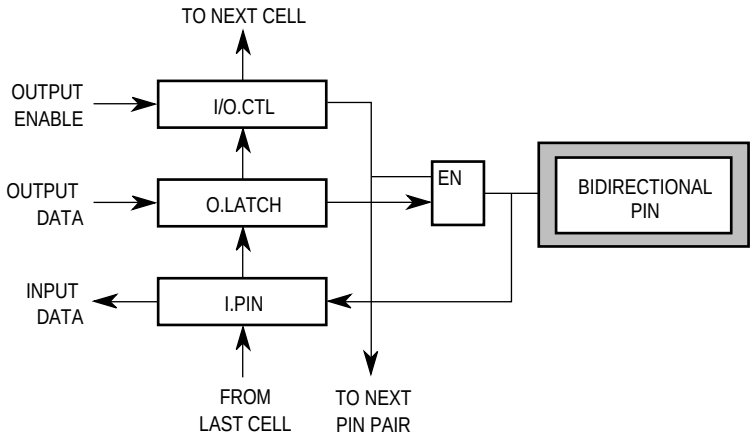


Figure 6-6. General Arrangement of Bidirectional Pins

All MCF5102 bidirectional pins include two boundary scan data cells, an input, and an output. One of five associated boundary scan control cells controls each bidirectional pin. If these cells contain a logic one, the associated bidirectional or three-state pin will be configured as an output and enabled. The cell captures the current value during the capture-DR state.

6.3 RESTRICTIONS

Control over the output enable signals using the boundary scan register and the EXTEST and HIGHZ instructions requires a compatible circuit-board test environment to avoid destructive configurations. The user is responsible for avoiding situations in which the MCF5102 output drivers are enabled into actively driven networks.

The MCF5102 include on-chip circuitry to detect the initial application of power to the device. Power-on reset (POR, which is an internal signal), the output of this circuitry, is used to reset both the system and the IEEE 1149.1A logic. The purpose of applying POR to the IEEE 1149.1A circuitry is to avoid the possibility of bus contention during power-on. The time required to complete device power-on is power supply dependent. However, the TAP controller remains in the test-logic-reset state while POR is asserted. The TAP controller does not respond to user commands until POR is negated.

The following restrictions apply:

1. Leaving the TAP controller test-logic-reset state negates the ability to achieve the lowest power consumption during the LPSTOP instruction, but does not otherwise affect device functionality.
2. The TCK input is not blocked in LPSTOP mode. To consume minimal power, the TCK input should be externally connected to V_{CC} .
3. The TDI and TMS pins have on-chip pull-up resisters. To achive minimal power consumption in LPSTOP mode, these pins should be connected to Vcc.
4. The external system must assert $\overline{\text{RSTI}}$ within eight bus clocks of exiting from the EXTEST $\overline{\text{JTAG}}$ instruction or else on the tenth bus clock, the MCF5102 will begin normal reset processing.
5. Pins JTAG and $\overline{Z}$ are defined to be compliance-enable inputs per section 3.8 of the IEEE standard 1149.1a-1993. Subordination of this standard within a higher level test strategy. The compliance-enable pattern is (0,1) respectively.
6. Pins TYLO1-TYLO3 are used for Factory test and require a connection to ground for proper system operation. Boundary-scan cells are provided for these pins to facilitate diagnostics of PCB defects.

6.4 DISABLING THE IEEE STANDARD 1149.1A OPERATION

There are two considerations for non-IEEE standard 1149.1A operation. First, TCK does not include an internal pullup resistor and should not be left unconnected to preclude mid-level inputs. The second consideration is to ensure that the IEEE standard 1149.1A test logic remains transparent to the system logic by providing the ability to force the test-logic-

reset state. Figure 6-7 illustrates a circuit to disable the IEEE standard 1149.1A test logic for the MCF5102.

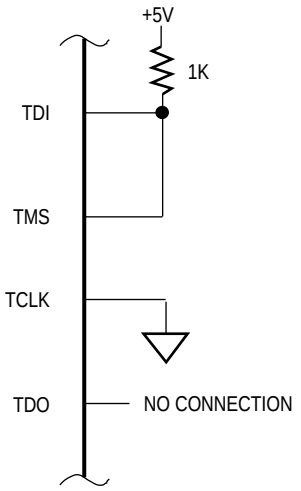


Figure 6-7. Circuit Disabling IEEE Standard 1149.1A

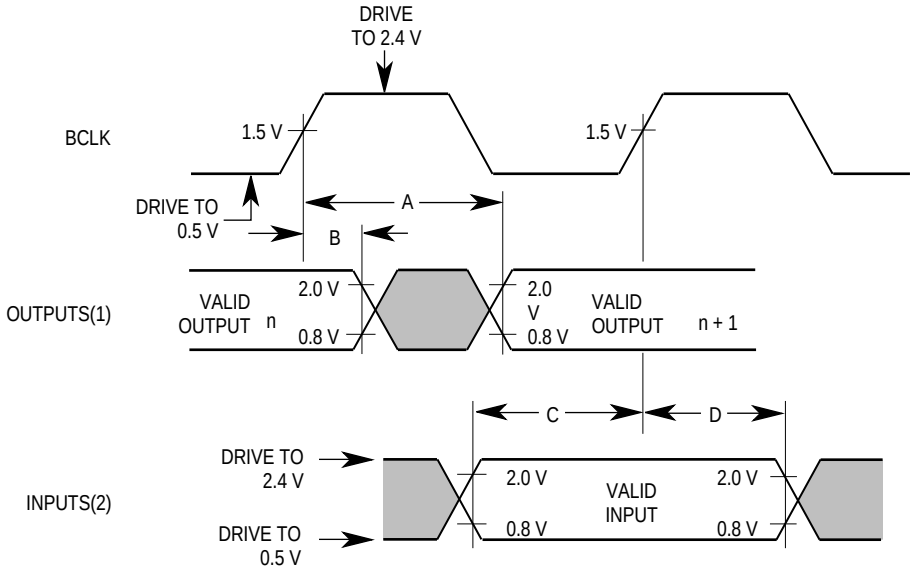
6.5 MCF5102 JTAG ELECTRICAL CHARACTERISTICS

The following paragraphs provide information on JTAG electrical and timing specifications This section is subject to change. For the most recent specifications, contact a Motorola sales office or complete the registration card at the beginning of this manual. Table 6-2 and FIgure 6-8 provide the JTAG DC electrical spectifications.

Table 6-2. JTAG DC Electrical Specifications

Characteristic	Symbol	Min	Max	Unit
Input High Voltage	V_{IH}	2	5.5	V
Input Low Voltage	V_{IL}	GND	0.8	V
Overshoot	—	—	TBD	V
TCK Input Leakage Current @ 0.5–2.4 V	I_{in}	TBD	TBD	mA
TDO Hi-Z (Off-State) Leakage Current @ 0.5–2.4 V	I_{TST}	TBD	TBD	mA
Signal Low Input Current, V_{IL} = 0.8 V TMS, TDI	I_L	TBD	TBD	mA
Signal High Input Current, V_{IH} = 2.0 V TMS, TDI	I_H	TBD	TBD	mA
TDO Output High Voltage I_{OH} = 5ma	V_{OH}	2.4	—	V
TDO Output Low Voltage I_{OL} = 5ma	V_{OL}	—	0.5	V
Capacitance*, V_{in} = 0 V, f = 1 MHz	C_{in}	—	TBD	pF

*Capacitance is periodically sampled rather than 100% tested.



NOTES:

1. This output timing is applicable to all parameters specified relative to the rising edge of the clock.
2. This input timing is applicable to all parameters specified relative to the rising edge of the clock.

LEGEND:

A. Maximum output delay specification.
B. Minimum output hold time.
C. Minimum input setup time specification.
D. Minimum input hold time specification.

Figure 6-8. Drive Levels and Test Points for AC Specifications

6.6 JTAG PINOUT

Figure 6-9 shows the pinout for JTAG. The JTAG pin are show in bold face.

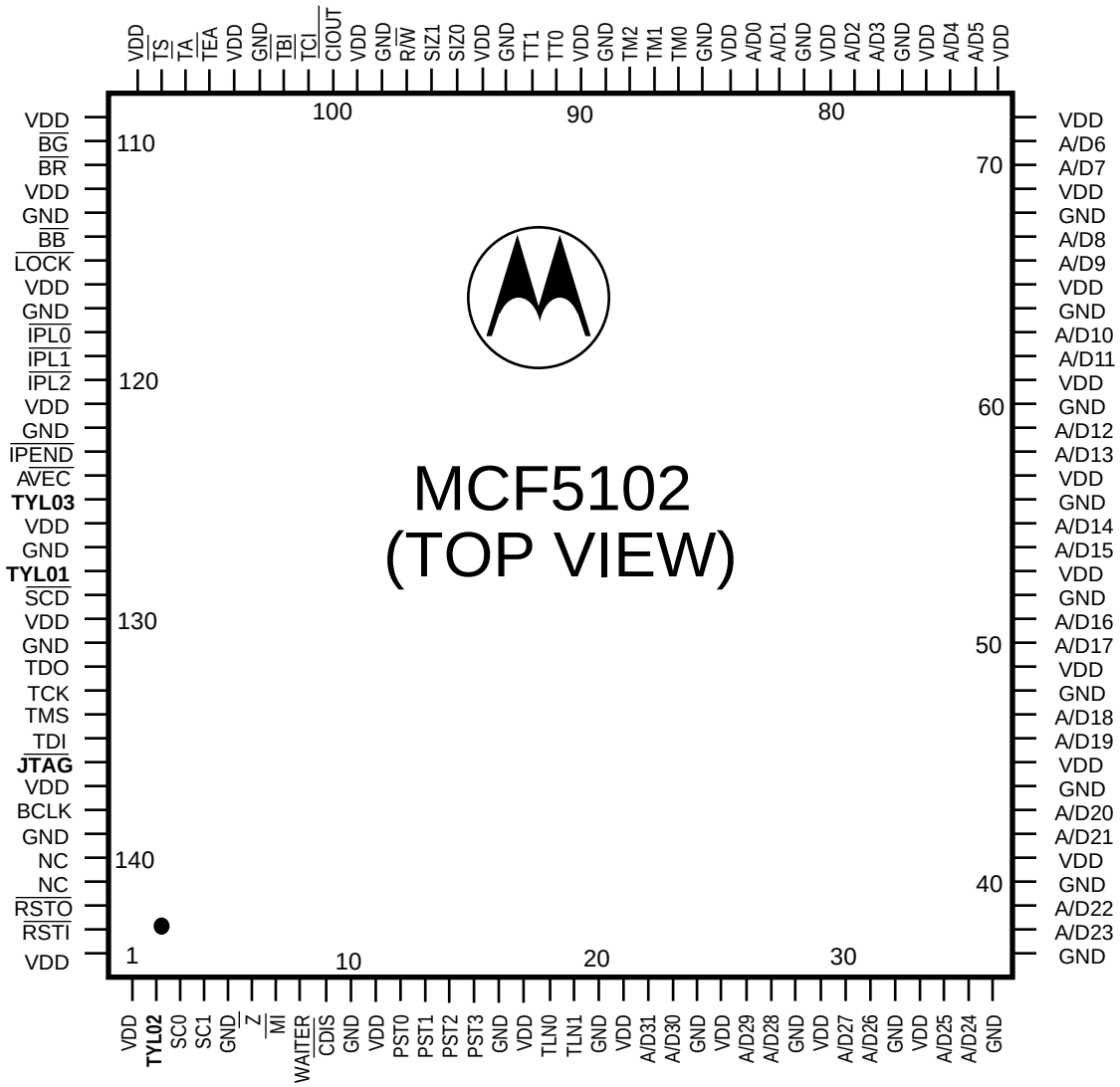


Figure 6-9. JTAG Pinout

6.7 BSDL DESCRIPTION.

The coding for BSDL can be downloaded from AESOP, our on-line BBS and Internet Server. Refer to the **Preface** for information on accessing AESOP.

SECTION 7
BUS OPERATION

The MCF5102 bus interface supports synchronous data transfers between the processor and other devices in the system. This section provides a functional description of the bus, the signals that control the bus, and the bus cycles provided for data transfer operations. Operation of the bus is defined for transfers initiated by the processor as a bus master and for transfers initiated by an alternate bus master, which the processor snoops as a slave device. Descriptions of the error and halt conditions, bus arbitration, and the reset operation are also included. For timing specifications, refer to **Section 10 MCF5102 Electrical and Thermal Characteristics.**

7.1 BUS CHARACTERISTICS

The MCF5102 uses a multiplexed address and data bus (A31/D31–A0/D0) to specify the address and the data for a bus cycle on a time-multiplexed basis. Transfer attribute signals indicate the type of bus cycle, and the Transfer Start ($\overline{TS}$) signal indicate the beginning of a bus cycle. The system then controls the length of the cycle by terminating it using the termination signals $\overline{TA}$ and/or $\overline{TEA}$. The rising edge of BCLK is used as the reference point all timing specifications.

For the sake of brevity, the following paragraphs contain several references to the address bus or the data bus as though they were non-multiplexed. This is done to separate the physical implementation of the bus architecture from the abstract concept of "the address" and "the data" when describing a bus cycle.

7.2 DATA TRANSFER MECHANISM

Figure 7-1 illustrates how the bus designates operands for transfers on a byte boundary system. These designations are used in the figures and descriptions that follow.

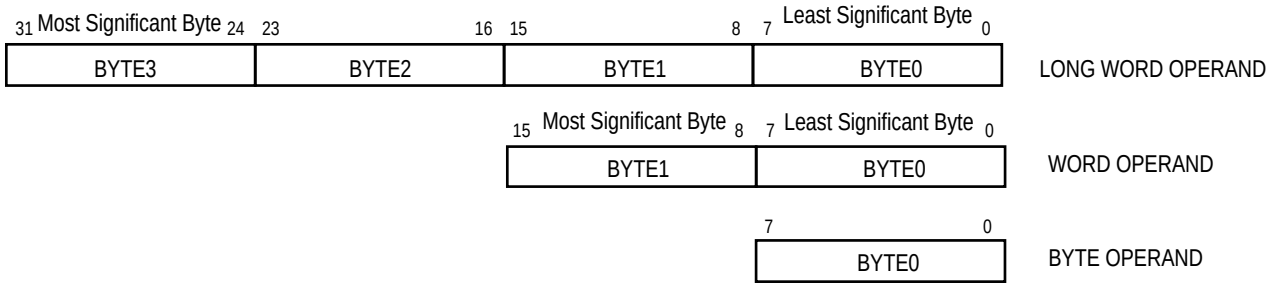


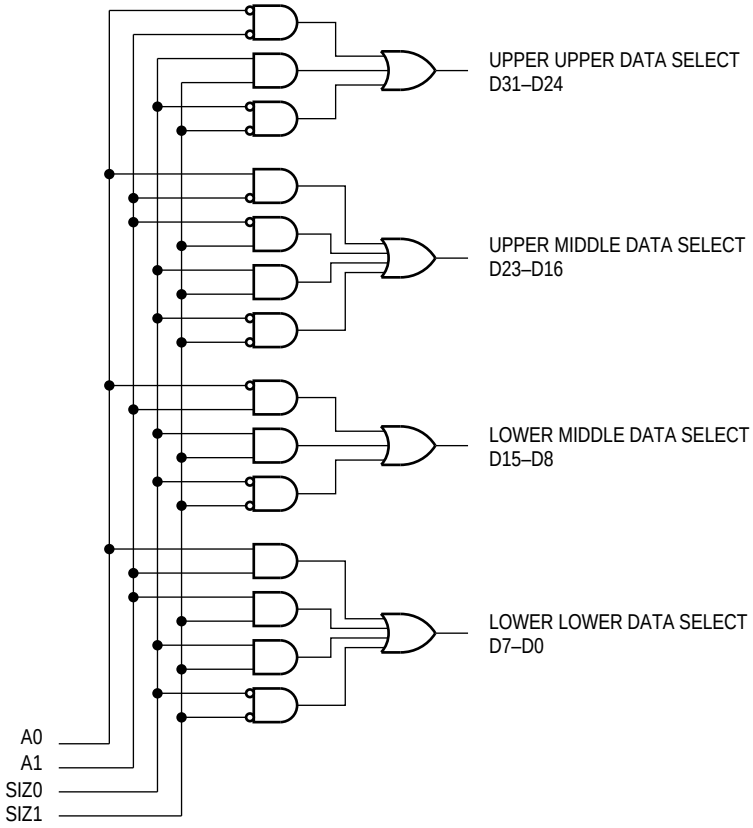
Figure 7-1. Internal Operand Representation

The MCF5102 does not support dynamic bus sizing and expects the referenced device to accept the requested access width. Hence, memory devices must be 32 bits wide. I/O devices that can supply their own vectors must drive the least significant byte (D7-D0). Therefore, it is recommended that byte- and word-sized I/O ports must be mapped into the low-order 8 or 16 bits, respectively, of the data bus.

Table 7-1 lists the combinations of the SIZx, A1, and A0 signals, collectively called byte enable signals. In the table, BYTEn indicates the data bus section that is active, the portion of the requested operand that is read or written during that bus transfer. For line transfers, all bytes are valid as listed and can correspond to portions of the requested operand or to data required to fill the remainder of the cache line. The bytes labeled with a dash are not required; they are ignored on read transfers and driven with undefined data on write transfers. Not selecting these bytes prevents incorrect accesses in sensitive areas such as I/O devices. Figure 7-2 illustrates a logic diagram for one method for generating byte enable signals from the SIZx, A1, and A0 and the associated PAL equation. These byte enable signals can be combined with the address decode logic.

Table 7-1. Data Bus Requirements for Read and Write Cycles

Transfer Size	Signal Encodings				Active Data Bus Sections			
	SIZ1	SIZ0	A1	A0	D31–D24	D23–D16	D15–D8	D7–D0
Byte	0	1	0	0	BYTE0	—	—	—
	0	1	0	1	—	BYTE0	—	—
	0	1	1	0	—	—	BYTE0	—
	0	1	1	1	—	—	—	BYTE0
Word	1	0	0	0	BYTE1	BYTE0	—	—
	1	0	1	0	—	—	BYTE1	BYTE0
Long Word	0	0	X	X	BYTE3	BYTE2	BYTE1	BYTE0
Line	1	1	X	X	BYTE3	BYTE2	BYTE1	BYTE0




```
PAL16L8
U1
MCF5102 Byte Data Select Generation.
Motorola Worldwide Marketing Training Organization
A0 A1 SIZ0 SIZ1 NC NC NC NC NC GND NC UUD UMD LMD LLD
NC NC NC NC VCC

/UUD =   /A0 * /A1                ; directly addressed, any size
       + /SIZ1 * /SIZ0            ; enable every byte for long word size
       + SIZ1 * SIZ0              ; enable every byte for line size
/UMD =   A0 * /A1                 ; directly addressed, any size
       + /A1 * /SIZ1              ; word aligned, size is word or line
       + SIZ1 * SIZ0              ; enable every byte for long word size
       + /SIZ1 * /SIZ0            ; enable every byte for line size
/LMD =   /A0 * /A1                ; directly addressed, any size
       + /SIZ1 * /SIZ0            ; enable every byte for long word size
       + SIZ1 * SIZ0              ; enable every byte for line size
/LLD =    A0 * /A1                ; directly addressed, any size
       + /A1 * /SIZ1              ; word aligned, word or line size
       + SIZ1 * SIZ0              ; enable every byte for long word size
       + /SIZ1 * /SIZ0            ; enable every byte for line size
```

Figure 7-2. Byte Enable Signal Generation and PAL Equation

A brief summary of the bus signal encodings for each access type is listed in Table 7-2. Additional information on the encodings for the MCF5102 signals can be found in **Section 5 Signal Description.**

Table 7-2. Summary of Access Types versus Bus Signal Encodings

Bus Signal	Data Cache Push Access	Normal Data/Code Access	MOVE16 Access	Alternate Access	Interrupt Acknowledge	Breakpoint Acknowledge
A31–A0	Access Address	Access Address	Access Address	Access Address	$FFFFFFFF	$00000000
SIZ1, SIZ0	L/Line	B/W/L/Line	Line	B/W/L	Byte	Byte
TT1, TT0	$0	$0	$1	$2	$3	$3
TM4–TM2	$0	$1,2,5, or 6	$1 or 5	Function Code	Int. Level $1–7	$0
TLN1, TLN0	Cache Set Entry	Undefined[2]	Undefined	Undefined	Undefined	Undefined
R/W	Write	Read/Write	Read/Write	Read/Write	Read	Read
LOCK	Negated	Asserted/ Negated[3]	Negated	Negated	Negated	Negated
$\overline{\text{CIOUT}}$	Negated	MMU Source[1]	MMU Source[1]	Asserted	Negated	Negated

NOTES
1. $\overline{\text{CIOUT}}$ signals are determined by the U1, U0 data and CM bit fields, respectively, corresponding to the access address.
2. The TLNx signals are defined only for normal push accesses and normal data line read accesses.
3. The LOCK signal is asserted during TAS, CAS, and CAS2 operand accesses .
4. Refer to **Section 5 Signal Description** for definitions of the TMx signal encodings for normal, MOVE16, and alternate accesses.

7.3 MISALIGNED OPERANDS

All MCF5102 data formats can be located in memory on any byte boundary. A byte operand is properly aligned at any address; a word operand is misaligned at an odd address; and a long word is misaligned at an address that is not evenly divisible by 4. However, since operands can reside at any byte boundary, they can be misaligned. Although the MCF5102 does not enforce any alignment restrictions for data operands (including PC relative data addressing), some performance degradation occurs when additional bus cycles are required for long-word or word operands that are misaligned. For maximum performance, data items should be aligned on their natural boundaries. All instruction words and extension words must reside on word boundaries. Attempting to prefetch an instruction word at an odd address causes an address error exception.

The MCF5102 data ACU converts misaligned operand accesses that are noncachable to a sequence of aligned accesses. These aligned accesses are then sent to the bus controller for completion, always resulting in aligned bus transfers. Misaligned operand accesses that miss in the data cache are cachable and are not aligned before line filling.

Figure 7-3 illustrates the transfer of a long-word operand from an odd address requiring more than one bus cycle. For the first transfer or bus cycle, the SIZx signals specify a byte transfer, and the byte offset is $1. The slave device supplies the byte and acknowledges the data transfer. When the processor starts the second cycle, the SIZx signals specify a word transfer with a byte offset of $2. The next two bytes are transferred during this cycle. The processor then initiates the third cycle, with the SIZEx signals indicating a byte transfer. The byte offset is now $0; the port supplies the final byte and the operation is

complete. This example is similar to the one illustrated in Figure 7-4 except that the operand is word sized and the transfer requires only two bus cycles.

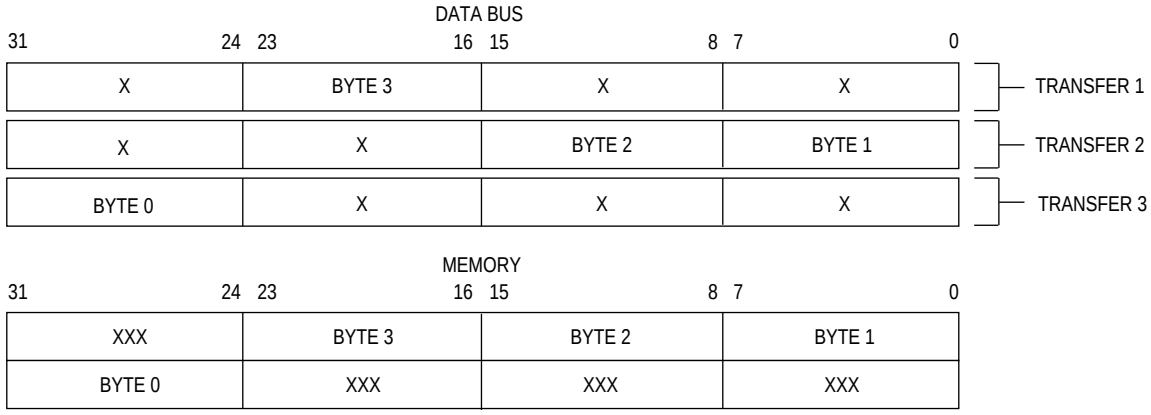


Figure 7-3. Example of a Misaligned Long-Word Transfer

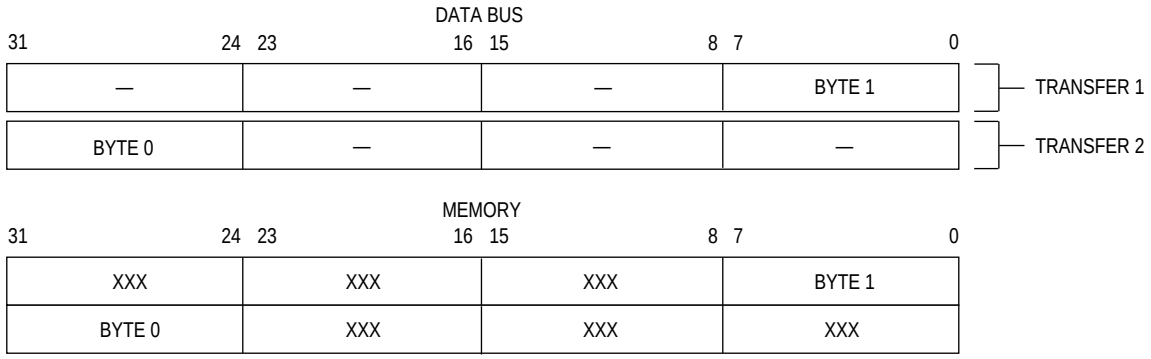


Figure 7-4. Example of a Misaligned Word Transfer

The combination of operand size and alignment determines the number of bus cycles required to perform a particular memory access. Table 7-3 lists the number of bus cycles required for different operand sizes with all possible alignment conditions for read and write cycles. The table confirms that alignment significantly affects bus cycle throughput for noncachable accesses. For example, in Figure 7-3 the misaligned long-word operand took three bus cycles because the byte offset = $1. If the byte offset = $0, then it would have taken one bus cycle. The MCF5102 system designer and programmer should account for these effects, particularly in time-critical applications.

Table 7-3. Memory Alignment Influence on Noncachable and Write-Through Bus Cycles

Transfer Size	Number of Bus Cycles			
	$0*	$1*	$2*	$3*
Instruction	1	N/A	N/A	N/A
Byte Operand	1	1	1	1
Word Operand	1	2	1	2
Long-Word Operand	1	3	2	3

*Where the byte offset (A1 and A0) equals this encoding.

The processor always prefetches instructions by reading a long word from a half-line address (A2–A0 = $0), regardless of alignment. When the required instruction begins at the second long word, the processor attempts to fetch the entire half-line (two long words) although the second long word contains the required instruction.

7.4 PROCESSOR DATA TRANSFERS

The transfer of data between the processor and other devices involves the mulitplexed address/bus, and control signals. The address/data buses are parallel, multiplexed buses, supporting byte, word, long-word, and line (16-byte) bus cycles. Line transfers are normally performed using an efficient burst transfer, which provides an initial address and time-multiplexes the data bus to transfer four long words of information to or from the slave device. Slave devices that do not support bursting can burst-inhibit the first long word of a line transfer, forcing the bus master to complete the access using three additional long-word bus cycles. All bus input and output signals are synchronous to the rising edge of the BCLK signal. The MCF5102 moves data on the bus by issuing control signals and using a handshake protocol to ensure correct data movement. The following paragraphs describe the bus cycles for byte, word, long-word, and line read, write, and read-modify-write transfers.

7.4.1 Byte, Word, and Long-Word Read Transfers

During a read transfer, the processor receives data from a memory or peripheral device. Since the data read for a byte, word, or long-word access is not placed in either of the internal caches by definition, the processor ignores the level on the transfer cache inhibit ($\overline{TCI}$) signal when latching the data. The bus controller performs byte, word, and long-word read transfers for the following cases:

- Accesses to a disabled cache.
- Accesses to memory that is specified noncachable.
- Accesses that are implicitly noncachable (read-modify-write accesses and accesses to an alternate logical address space via the MOVES instruction).
- Accesses that do not allocate in the data cache on a read miss (table searches, exception vector fetches, and exception stack deallocation for an RTE instruction).

- The first transfer of a line read is terminated with transfer burst inhibit ($\overline{TBI}$), forcing completion of the line access using three additional long-word read transfers.

Figure 7-5 is a functional timing diagram for byte, word, and long-word read transfers.

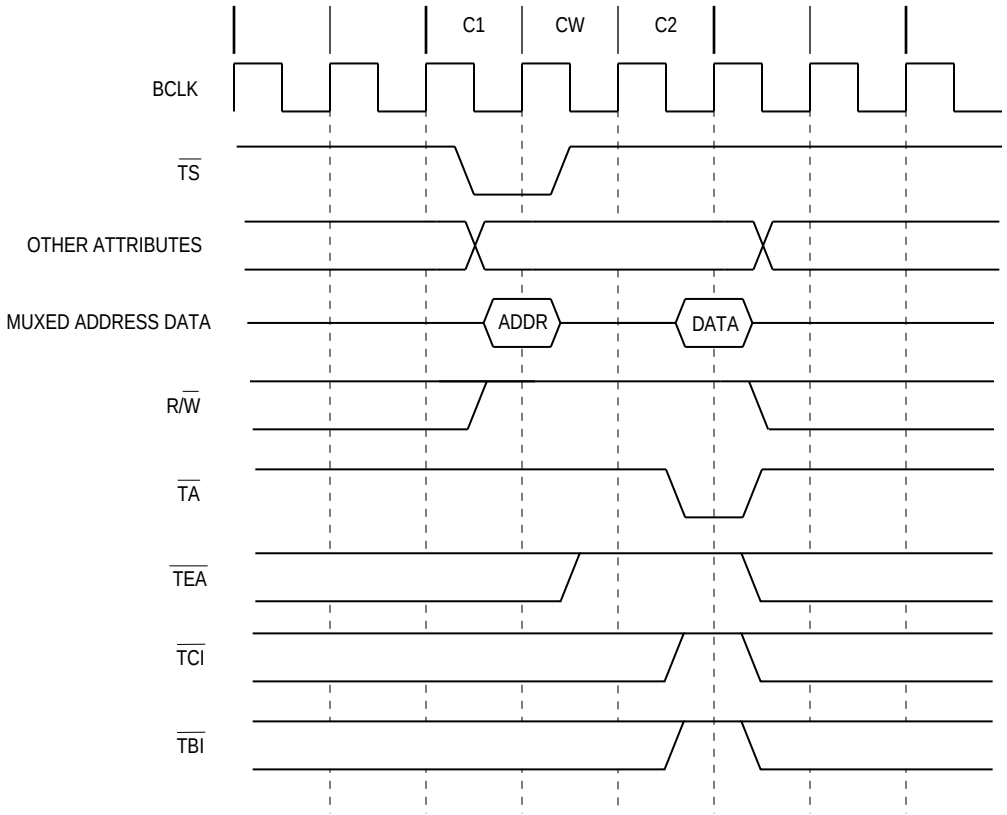


Figure 7-5. Byte, Word, and Long-Word Read Transfer Timing

Clock 1 (C1)

The read cycle starts in C1. During C1, the processor drives the address on the multiplexed bus. It also drives the appropriate values on the transfer attributes.

The processor asserts transfer start ($\overline{TS}$) during C1 to indicate the beginning of a bus cycle.

After the C1 state, the processor negates $\overline{TS}$, and the multiplexed bus no longer drives the address. If the next cycle is a wait state, proceed to CW, otherwise, proceed to C2.

Clock W (CW)

On a wait state, the system does not assert $\overline{TA}$. The processor ignores any data on the multiplexed bus and the processor continues to sample $\overline{TA}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized asserted. If the next cycle is not a wait state, proceed to C2.

Clock 2 (C2)

The system asserts the transfer acknowledge ($\overline{TA}$) signal and latches the current value on the data bus; the bus cycle terminates.

7.4.2 Line Read Transfer

The processor uses line read transfers to access a 16-byte operand for a MOVE16 instruction and to support cache line filling. A line read accesses a block of four long words, aligned to a 16-byte memory boundary, by supplying a starting address that points to one of the long words and requiring the memory device to sequentially drive each long word on the data bus. The system must internally increment A3 and A2 of the supplied address for each transfer, causing the address to wrap around at the end of the block. The system terminates each long word transfer by driving the long word on the data bus and asserting $\overline{TA}$. A line transfer performed in this manner with a single address is also referred to as a line burst transfer.

The MCF5102 also supports burst-inhibited line transfers for memory devices that are unable to support bursting. For this type of bus cycle, the system supplies the first long word pointed to by the processor address and asserts transfer burst inhibit ($\overline{TBI}$) with $\overline{TA}$ for the first transfer of the line access. The processor responds by terminating the line burst transfer and accessing the remainder of the line, using three individual, separate long-word read bus cycles. Although the line transfer results in four, independent, long-word bus cycles, the processor still handles these four transfers as a single line transfer and does not allow bus arbitration to intervene between the transfers. $\overline{TBI}$ is ignored after the first long-word transfer.

Line reads to support cache line filling can be cache inhibited by asserting transfer cache inhibit ($\overline{TCI}$) with $\overline{TA}$ for the first long-word transfer of the line. The assertion of $\overline{TCI}$ does not affect completion of the line transfer, but the bus controller latches and passes it to the ACU for use. $\overline{TCI}$ is ignored after the first long-word transfer of a line burst transfer and during the three long-word bus cycles for a burst-inhibited line transfer.

The system should ignore A1 and A0 for long-word and line read transfers.

The address of an instruction fetch will always be aligned to a half-line boundary ($XXXXXXX0 or $XXXXXXX8); therefore, compilers should attempt to locate branch targets on half-line boundaries to minimize branch stalls. For example, if the target of a branch is a two-word instruction located at $1000000C, the following burst sequence will occur upon a cache miss: $10000008, $1000000C, $10000000, then $10000004. The internal pipeline of the MCF5102 stalls until the second access of the burst (the address of the instruction to be executed) has completed. Figures 7-6 illustrates a functional timing diagram for a line read bus transfer.

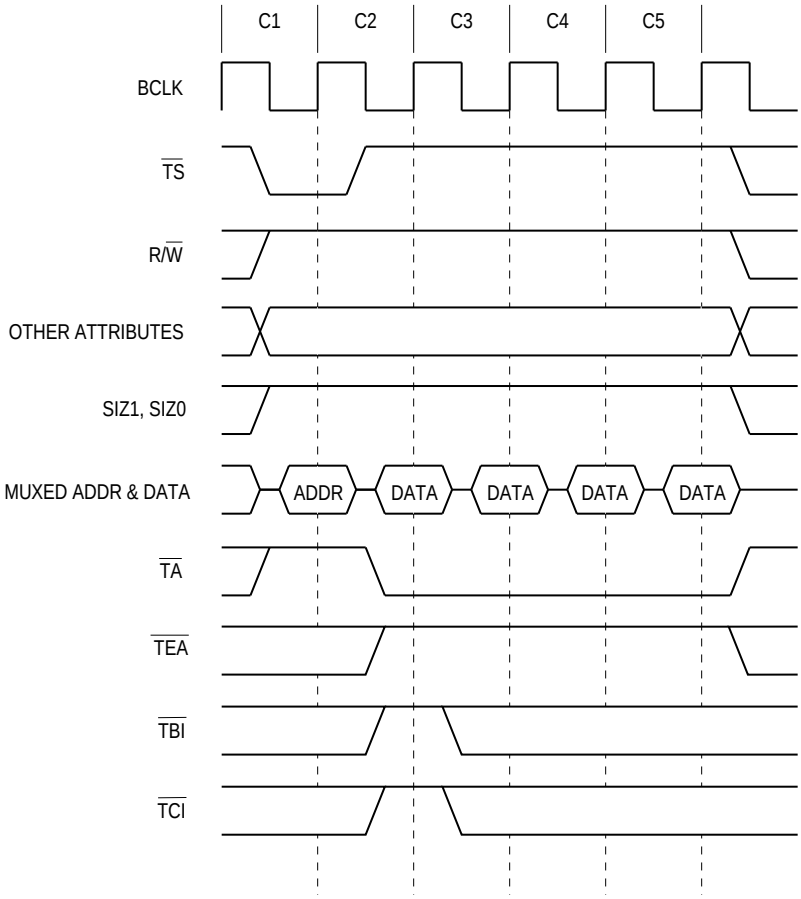


Figure 7-6. Line Read Transfer Timing

Clock 1 (C1)

The line read cycle starts in C1. During C1, the processor places valid values on the transfer attributes and drives the address onto the multiplexed bus. The size signals (SIZx) indicate line size.

After C1, the processor negates $\overline{TS}$ and the multiplexed bus no longer drives the address. The system is responsible for driving data onto the multiplexed bus. The first transfer must supply the long word at the corresponding long-word boundary.

Clock 2 (C2)

At C2, the processor samples the level of $\overline{TA}$, $\overline{TBI}$, and $\overline{TCI}$ and latches the current value on the data bus. If $\overline{TA}$ is asserted, the transfer terminates and the data is passed to the appropriate ACU. Otherwise, if $\overline{TA}$ is negated, the processor ignores the data and inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$, $\overline{TBI}$, and $\overline{TCI}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized asserted. The latched data and the level on $\overline{TCI}$ are then passed to the appropriate ACU.

If $\overline{TBI}$ is sampled negated with $\overline{TA}$, the processor continues the cycle with C3. Otherwise, if $\overline{TBI}$ is asserted, the line transfer is burst inhibited, and the processor reads the remaining three long words using long-word read bus cycles. The processor increments A3 and A2 for each read, and the new address is placed on the address bus for each bus cycle. Refer to **7.4.1 Byte, Word, and Long-Word Read Transfers** for information on long-word reads.

Clock 3 (C3)

The processor holds the transfer attribute signals constant during C3. The system must increment A3 and A2 to reference the next long word to transfer, place the data on the multiplexed bus, and assert $\overline{TA}$. At the end of C3, the processor samples the level of $\overline{TA}$ and latches the current value on the data bus. If $\overline{TA}$ is asserted, the transfer terminates, and the second long word of data is taken by the processor. If $\overline{TA}$ is not recognized as asserted, the processor ignores the latched data and inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$ on successive rising edges of BCLK until it is recognized. The latched data is then passed to the processor.

Clock 4 (C4)

This clock is identical to C3 except that once $\overline{TA}$ is recognized as asserted, the latched value corresponds to the third long word of data for the burst.

Clock 5 (C5)

This clock is identical to C3 except that once $\overline{TA}$ is recognized, the latched value corresponds to the third long word of data for the burst. After the processor recognizes the last $\overline{TA}$ assertion and terminates the line read bus cycle.

Figures 7-7 illustrate a functional timing diagram for a burst-inhibited line read bus cycle.

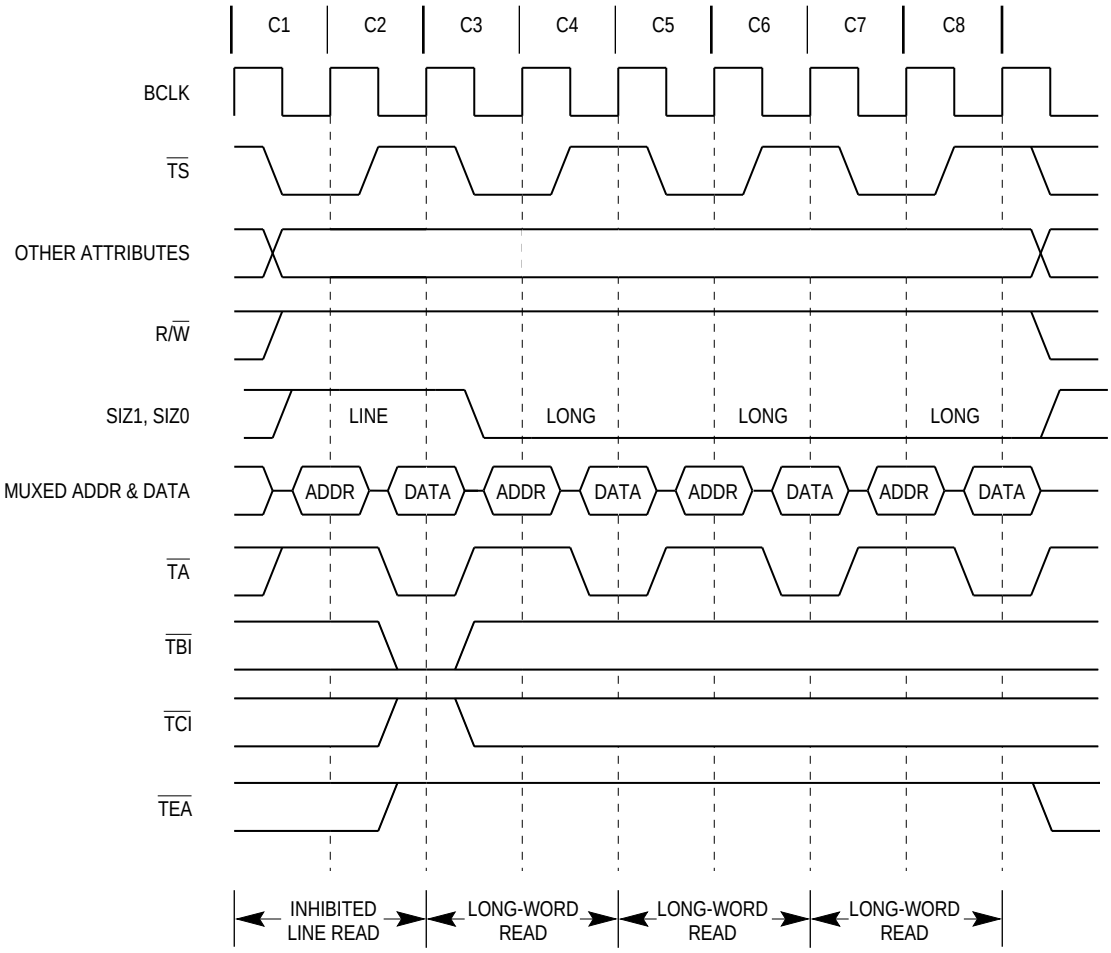


Figure 7-7. Burst-Inhibited Line Read Transfer Timing

7.4.3 Byte, Word, and Long-Word Write Transfers

During a write transfer, the processor transfers data to a memory or peripheral device. The level on the $\overline{\text{TCI}}$ signal is ignored by the processor during all write cycles. The bus controller performs byte, word, and long-word write transfers for the following cases:

- Accesses to a disabled cache.
- Accesses to memory that is specified noncachable.
- Accesses that are implicitly noncachable (read-modify-write accesses and accesses to an alternate logical address space via the MOVES instruction).
- Writes to write-through memory.
- Accesses that do not allocate in the data cache on a write miss (exception stacking).
- The first transfer of a line write is terminated with $\overline{\text{TBI}}$, forcing completion of the line access using three additional long-word write transfers.
- Cache line pushes for lines containing a single dirty long word.

Figures 7-8 illustrates a functional timing diagram for byte, word, and long-word write bus transfers.

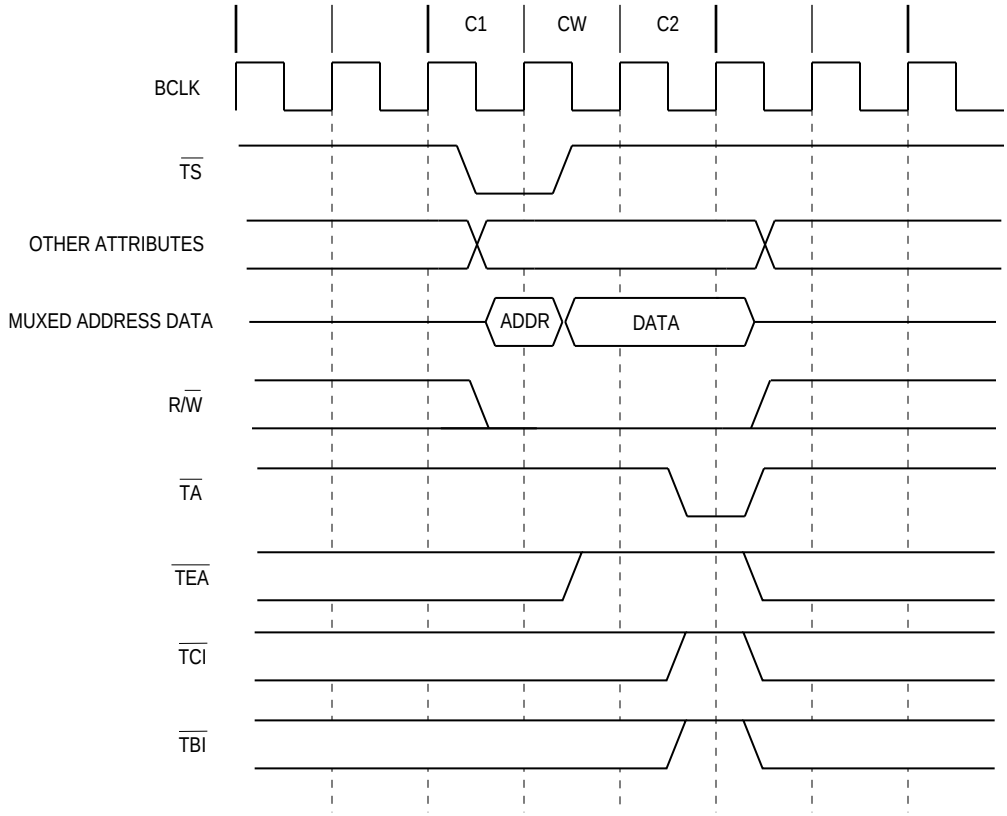


Figure 7-8. Byte, Word, and Long-Word Write Transfer Timing

Clock 1 (C1)

The write cycle starts in C1. During this time, the processor places valid values on the transfer attributes and drives the address onto the multiplexed bus. The processor asserts $\overline{TS}$ during C1 to indicate the beginning of a bus cycle.

After C1, the processor negates $\overline{TS}$, drives the appropriate bytes of the data bus with the data to be written on the multiplexed bus. All other byte lanes of the multiplexed bus are driven with undefined values.

Clock 2 (C2)

The system uses R/$\overline{W}$, SIZ1, SIZ0, A1, A0, and $\overline{CIOUT}$ to identify the valid byte lanes on the multiplexed bus. If C2 is not a wait state, then the system asserts the $\overline{TA}$ signal.

At the end of C2, the processor samples the level of $\overline{TA}$, terminating the bus cycle if $\overline{TA}$ is asserted. If $\overline{TA}$ is not recognized as asserted at the end of the clock cycle, the processor inserts a wait state instead of terminating the transfer. The processor continues to sample $\overline{TA}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized as asserted. The multiplexed bus is then tri-stated and the bus cycle ends.

7.4.4 Line Write Transfers

The processor uses line write bus cycles to access a 16-byte operand for a MOVE16 instruction and to support cache line pushes. Both burst and burst-inhibited transfers are supported. Figures 7-9 illustrates a functional timing diagram for a line write bus cycle.

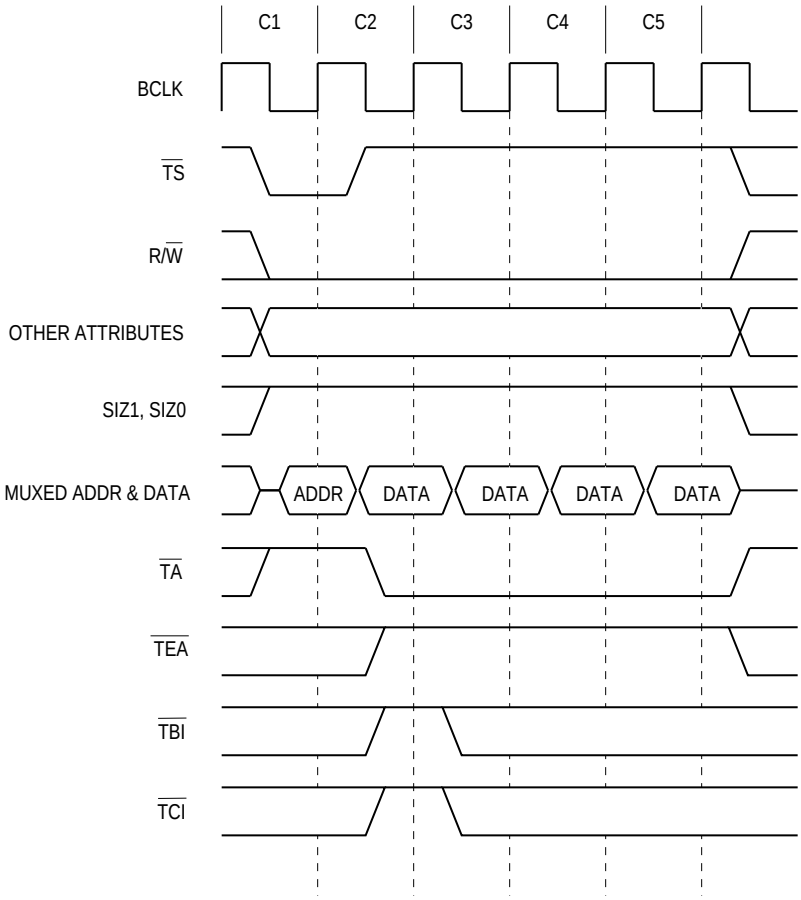


Figure 7-9. Line Write Transfer Timing

Clock 1 (C1)

The line write cycle starts in C1. During C1, the processor places valid values on the transfer attributes and drives the multiplexed bus with the address. The SIZ1 and SIZ0 indicate line size. The processor asserts $\overline{TS}$ to indicate the beginning of the bus cycle.

After C1, the processor negates $\overline{TS}$ and drives the multiplexed bus with the data to be written. All byte lanes are valid.

Clock 2 (C2)

During C2, the system asserts $\overline{TA}$ and either negates or asserts $\overline{TBI}$ to indicate it can or cannot support a burst transfer. At the end of C2, the processor samples the level of $\overline{TA}$ and $\overline{TBI}$. If $\overline{TA}$ is asserted, the transfer terminates. If $\overline{TA}$ is not recognized asserted, the

processor inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$ and $\overline{TBI}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized asserted.

If $\overline{TBI}$ is negated with $\overline{TA}$, the processor continues the cycle with C3. Otherwise, if $\overline{TBI}$ is asserted, the line transfer is burst inhibited, and the processor writes the remaining three long words using long-word write bus cycles. Only in this case does the processor increment A3 and A2 for each write, and the new address is placed on the address bus for each bus cycle. Refer to **7.4.3 Byte, Word, and Long-Word Write Transfers** for information on long-word writes.

Clock 3 (C3)

The processor drives the second long word of data on the multiplexed bus and holds the transfer attribute signals constant during C3. The system references the next long word, latches this data from the data bus, and asserts $\overline{TA}$. At the end of C3, the processor samples the level of $\overline{TA}$; if $\overline{TA}$ is asserted, the transfer terminates. If $\overline{TA}$ is not recognized asserted at the end of C3, the processor inserts wait states instead of terminating the transfer. The processor continues to sample $\overline{TA}$ on successive rising edges of BCLK until $\overline{TA}$ is recognized as asserted.

Clock 4 (C4)

This clock is identical to C3 except that the value driven on the data bus corresponds to the third long word of data for the burst.

Clock 5 (C5)

This clock is identical to C3 except that the value driven on the data bus corresponds to the fourth long word of data for the burst.

7.4.5 Read-Modify-Write Transfers (Locked Transfers)

The read-modify-write transfer performs a read, conditionally modifies the data in the processor, and writes the data out to memory. In the MCF5102, this operation can be indivisible, providing semaphore capabilities for multiprocessor systems. During the entire read-modify-write sequence, the MCF5102 asserts the $\overline{LOCK}$ signal to indicate that an indivisible operation is occurring. The external arbiter can use the $\overline{LOCK}$ signals to prevent arbitration of the bus during locked processor sequences. A read-modify-write operation is treated as noncachable. If the access hits in the data cache, it invalidates a matching valid entry and pushes a matching dirty entry. The read-modify-write transfer begins after the line push (if required) is complete; however, $\overline{LOCK}$ may assert during the line push bus cycle.

The TAS, CAS, and CAS2 instructions are the only instructions that utilize read-modify-write transfers.

7.5 ACKNOWLEDGE BUS CYCLES

Bus transfers with transfer type signals TT1 and TT0 = $3 are classified as acknowledge bus cycles. The following paragraphs describe interrupt acknowledge and breakpoint acknowledge bus cycles that use this encoding.

7.5.1 Interrupt Acknowledge Bus Cycles

When a peripheral device requires the services of the MCF5102 or is ready to send information that the processor requires, it can signal the processor to take an interrupt exception. The interrupt exception transfers control to a routine that responds appropriately. The peripheral device uses the active-low interrupt priority level signals ($\overline{IPL2}$–$\overline{IPL0}$) to signal an interrupt condition to the processor and to specify the priority level for the condition.

The status register (SR) contains an interrupt priority mask (I2–I0 bits). The value in the interrupt mask is the highest priority level that the processor ignores. When an interrupt request has a priority higher than the value in the mask, the processor makes the request a pending interrupt. $\overline{IPL2}$–$\overline{IPL0}$ must maintain the interrupt request level until the MCF5102 acknowledges the interrupt to guarantee that the interrupt is recognized. The MCF5102 continuously samples $\overline{IPL2}$–$\overline{IPL0}$ on consecutive rising edges of BCLK to synchronize and debounce these signals. An interrupt request that is held asserted for as little as two consecutive clock periods may signal an interrupt, although the protocol requires that the request remain until the processor runs an interrupt acknowledge cycle for that interrupt value.

The MCF5102 asserts $\overline{IPEND}$ when an interrupt request is pending. $\overline{IPEND}$ indicates that an interrupt exception will be taken at an upcoming instruction boundary (following any higher priority exception). However, $\overline{IPEND}$ must not be used instead of the interrupt acknowledge cycle.

Table 7-4 provides a summary of the possible interrupt acknowledge terminations and the exception processing results.

Table 7-4. Interrupt Acknowledge Termination Summary

$\overline{TA}$	$\overline{TEA}$	$\overline{AVEC}$	Termination Condition
High	High	Don’t Care	Insert Waits
High	Low	Don’t Care	Take Spurious Interrupt Exception
Low	High	High	Latch Vector Number on D7–D0 and Take Interrupt Exception
Low	High	Low	Take Autovectored Interrupt Exception
Low	Low	Don’t Care	Retry Interrupt Acknowledge Cycle

7.5.1.1 INTERRUPT ACKNOWLEDGE BUS CYCLE (TERMINATED NORMALLY). When the MCF5102 processes an interrupt exception, it performs an interrupt acknowledge bus cycle to obtain the vector number that contains the starting location of

the interrupt exception handler. Some interrupting devices have programmable vector registers that contain the interrupt vectors for the exception handlers they use. Other interrupting conditions or devices cannot supply a vector number and use the autovector bus cycle described in **7.5.1.2 Autovector Interrupt Acknowledge Bus Cycle**.

The interrupt acknowledge bus cycle is a read bus cycle. It differs from a normal read cycle in the following respects:

1. TT1 and TT0 = $3 to indicate an acknowledged bus cycle.
2. The Address bus is set to all ones ($FFFFFFFF).
3. TM2–TM0 are set to the interrupt request level (the inverted values of $\overline{IPL2}$–$\overline{IPL0}$).

The system must place the vector number on the multiplexed bus (D7-D0 during the data phase) when the interrupt acknowledge cycle is terminated normally with $\overline{TA}$.

7.5.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE BUS CYCLE. When the system chooses not to supply a vector number, it may request an automatically generated vector (autovector) instead of placing a vector number on the data bus. This is done by and asserting the $\overline{AVEC}$ signal with $\overline{TA}$ to terminate the cycle. $\overline{AVEC}$ is significant with $\overline{TA}$ asserted only during interrupt acknowledge cycles. $\overline{AVEC}$ can be grounded if all interrupt requests are autovectored.

The vector number supplied in an autovector operation is derived from the interrupt priority level of the current interrupt. When the $\overline{AVEC}$ signal is asserted with $\overline{TA}$ during an interrupt acknowledge bus cycle, the MCF5102 ignores the data on the multiplexed bus and internally generates the vector number. The vector used is the sum of the interrupt priority level plus 24 ($18). There are seven distinct autovectors that can be used, corresponding to the seven levels of interrupts available with $\overline{IPL2}$–$\overline{IPL0}$ signals.

7.5.1.3 SPURIOUS INTERRUPT ACKNOWLEDGE BUS CYCLE. When an interrupt acknowledge bus cycle is terminated with a bus error ($\overline{TEA}$ asserted without $\overline{TA}$, $\overline{AVEC}$ is insignificant), the MCF5102 automatically generates the spurious interrupt vector number 24 ($18) instead of the interrupt vector number or autovector. If $\overline{TA}$ and $\overline{TEA}$ are both asserted, the processor retries the cycle.

7.5.2 Breakpoint Interrupt Acknowledge Bus Cycle

The execution of a breakpoint instruction (BKPT) generates the breakpoint interrupt acknowledge bus cycle. An acknowledged access is a read bus cycle, although no data is required, and is indicated with TT1 and TT0 = $3. The breakpoint interrupt acknowledge bus cycle differs from the interrupt acknowledge cycle in that the the address bus indicates $00000000, and TM2–TM0 = $0. When this bus cycle is terminated with either $\overline{TA}$ or $\overline{TEA}$ (as long as it is not a retry termination), the processor takes an illegal instruction exception.

7.6 BUS EXCEPTION CONTROL CYCLES

The MCF5102 bus architecture requires assertion of $\overline{TA}$ from an external device to signal that a bus cycle is complete. $\overline{TA}$ is not asserted in the following cases:

- The external device does not respond.
- No interrupt vector is provided.
- Various other application-dependent errors occur.

External circuitry can provide $\overline{TEA}$ when no device responds by asserting $\overline{TA}$ within an appropriate period of time after the processor begins the bus cycle. This allows the cycle to terminate and the processor to enter exception processing for the error condition. $\overline{TEA}$ can also be asserted in combination with $\overline{TA}$ to cause a retry of a bus cycle in error.

Table 7-5 lists the control signal combinations and the resulting bus cycle terminations. Bus error and retry terminations during burst cycles operate as described in **7.4.2 Line Read Transfers** and **7.4.4 Line Write Transfers**.

Table 7-5. $\overline{TA}$ and $\overline{TEA}$ Assertion Results

Case No.	$\overline{TA}$	$\overline{TEA}$	Result
1	High	Low	Bus Error—Terminate and Take Bus Error Exception, Possibly Deferred
2	Low	Low	Retry Operation—Terminate and Retry
3	Low	High	Normal Cycle Terminate and Continue
4	High	High	Insert Wait States

7.6.1 Bus Errors

The system hardware can use the $\overline{TEA}$ signal to abort the current bus cycle when a fault is detected. A bus error is recognized during a bus cycle when $\overline{TA}$ is negated and $\overline{TEA}$ is asserted. When the processor recognizes a bus error condition for an access, the access is terminated immediately. A line access that has $\overline{TEA}$ asserted for one of the four long-word transfers aborts without completing the remaining transfers, regardless of whether the line transfer uses a burst or burst-inhibited access.

When $\overline{TEA}$ is asserted to terminate a bus cycle, the MCF5102 can enter access error exception processing immediately following the bus cycle, or it can defer processing the exception. The instruction prefetch mechanism requests instruction words from the instruction ACU before it is ready to execute them. If a bus error occurs on an instruction fetch, the processor does not take the exception until it attempts to use the instruction. Should an intervening instruction cause a branch or should a task switch occur, the access error exception for the unused access does not occur. Similarly, if a bus error is detected on the second, third, or fourth long-word transfer for a line read access, an access error exception is taken only if the execution unit is specifically requesting that long word. Otherwise, the line is not placed in the cache, and the processor repeats the line access when another access references the line. If a misaligned operand spans two long

words in a line, a bus error on either the first or second transfer for the line causes exception processing to begin immediately. A bus error termination for any write accesses or for read accesses that reference data specifically requested by the execution unit causes the processor to begin exception processing immediately. Refer to **Section 8 Exception Processing** for details of access error exception processing.

When a bus error terminates an access, the contents of the corresponding cache can be affected in different ways, depending on the type of access. For a cache line read to replace a valid instruction or data cache line, the cache line being filled is invalidated before the bus cycle begins and remains invalid if the replacement line access is terminated with a bus error. If a dirty data cache line is being replaced and a bus error occurs during the replacement line read, the dirty line is restored from an internal push buffer into the cache to eliminate an unnecessary push access. If a bus error occurs during a data cache push, the corresponding cache line remains valid (with the new line data) if the line push follows a replacement line read, or is invalidated if a CPUSH instruction explicitly forces the push. Write accesses to memory specified as write-through by the data ACU update the corresponding cache line before accessing memory. If a bus error occurs during a memory access, the cache line remains valid with the new data.

7.6.2 Retry Operation

When an external device asserts both the $\overline{TA}$ and $\overline{TEA}$ signals during a bus cycle, the processor enters the retry sequence. The processor terminates the bus cycle and immediately retries the cycle using the same access information (address and transfer attributes). However, if the bus cycle was a cache push operation, the bus is arbitrated away from the MCF5102 before the retry operation, and a snoop during the arbitration invalidates the cache push, then the processor does not use the same access information.

The processor retries any read or write cycles of a read-modify-write transfer separately; $\overline{LOCK}$ remains asserted during the entire retry sequence.

On the first longword of a line transfer, a retry causes the processor to retry the bus cycle. However, the processor recognizes a retry signaled during the second, third, or fourth cycle of a line as a bus error and causes the processor to abort the line transfer. A burst-inhibited line transfer can only be retried on the initial transfer. A burst-inhibited line transfer aborts if a retry is signaled for any of the three long-word transfers used to complete the line transfer. Negating the bus grant ($\overline{BG}$) signal on the MCF5102 while asserting both $\overline{TA}$ and $\overline{TEA}$ provides a relinquish and retry operation for any bus cycle that can be retried.

7.6.3 Double Bus Fault

A double bus fault occurs when an access or address error occurs during the exception processing sequence—e.g., the processor attempts to stack several words containing information about the state of the machine while processing an access error exception. If a bus error occurs during the stacking operation, the second error is considered a double bus fault.

The MCF5102 indicates a double bus fault condition by continuously driving PST3–PST0 with an encoded value of $5 until the processor is reset. Only an external reset operation can restart a halted processor. While the processor is halted, negating $\overline{BG}$ and forcing all outputs to a high-impedance state releases the external bus.

A second access or address error that occurs during execution of an exception handler or later, does not cause a double bus fault. A bus cycle that is retried does not constitute a bus error or contribute to a double bus fault. The processor continues to retry the same bus cycle as long as external hardware requests it.

7.7 BUS SYNCHRONIZATION

The integer unit generates access requests to the instruction and data ACU's to support integer operations. Both the <ea> fetch and write-back stages of the integer unit pipeline perform accesses to the data ACU with effective address fetches assigned a higher priority. This priority allows data read and write accesses to occur out of order, with a memory write access potentially delayed for many clocks while allowing read accesses generated by later instructions to complete. The processor detects a read access that references earlier data waiting to be written (address collisions) and allows the corresponding write access to complete. A given sequence of read accesses or write accesses is completed in order, and reordering only occurs with writes relative to reads.

Besides address collisions, the instruction restart model used for exception processing causes another potential problem. After the operand fetch for an instruction, an exception that causes the instruction to be aborted can occur, resulting in another access for the operand after the instruction restarts. For example, an exception could occur after a read access of an I/O device’s status register. The exception causes the instruction to be aborted and the register to be read again. If the first read accesses clears the status bits, the status information is lost, and the instruction obtains incorrect data.

Designating the memory containing the address of the device as serialized noncachable prevents multiple out-of-order accesses to devices sensitive to such accesses. When the data ACU detects an attempt to read an operand from memory designated as serialized noncachable, it allows all pending write accesses to complete before beginning the external read access. The definition of memory as noncachable versus serialized noncachable only affects read accesses. When a write operation reaches the integer unit’s write-back stage, all previous instructions have completed. When a read access to a serialized noncachable memory begins, only a bus error exception on the operand read itself can cause the instruction to be aborted, preventing multiple reads.

Since write cycles can be deferred indefinitely, many subsequent instructions can be executed, resulting in seemingly nonsequential instruction execution. When this action is not desired and the system depends on sequential execution following bus activity, the NOP instruction can be used. The NOP instruction forces instruction and bus synchronization because it freezes instruction execution until all pending bus cycles have completed.

A write operation of control information to an external register in which the external hardware attempts to control program execution based on the data that is written with the conditional assertion of $\overline{\text{TEA}}$ is one situation where the NOP instruction can be used to prevent multiple executions. If the data cache is enabled and the write cycle results in a hit in the data cache, the cache is updated. That data, in turn, may be used in a subsequent instruction before the external write cycle completes. Since the MCF5102 cannot process the bus error until the end of the bus cycle, the external hardware cannot successfully interrupt program execution. To prevent a subsequent instruction from executing until the external cycle completes, the NOP instruction can be inserted after the instruction causing the write. In this case, access error exception processing proceeds immediately after the write before subsequent instructions are executed. This is an irregular situation, and the use of the NOP instruction for this purpose is not required by most systems.

Note that the NOP instruction can also be used to force access serialization by placing NOP before the instruction that reads an I/O device. This practice eliminates the need to specify memory as serialized noncachable but does not prevent the instruction from being aborted by an exception condition.

7.8 BUS ARBITRATION

The bus design of the MCF5102 provides for one bus master at a time, either the MCF5102 or an external device. More than one device having the capability to control the bus can be attached to the bus. An external arbiter prioritizes requests and determines which device is granted access to the bus. Bus arbitration is the protocol by which the processor or an external device becomes the bus master. When the MCF5102 is the bus master, it uses the bus to read instructions and data not contained in its internal caches from memory and to write data to memory. When an alternate bus master owns the bus, the MCF5102 is able to monitor the alternate bus master’s transfer and intervene when necessary to maintain cache coherency.

The MCF5102 implements an arbitration method in which an external arbiter controls bus arbitration and the processor acts as a slave device requesting ownership of the bus from the arbiter. Since the user defines the functionality of the external arbiter, it can be configured to support any desired priority scheme. For systems in which the processor is the only possible bus master, the bus can be continuously granted to the processor, and no arbiter is needed. Systems that include several devices that can become bus masters require an arbiter to assign priorities to these devices so that, when two or more devices simultaneously attempt to become the bus master, the one having the highest priority becomes the bus master first.

The MCF5102 bus controller generates bus requests to the external arbiter in response to internal requests from the instruction and data ACU's. The MCF5102 performs bus arbitration using the bus request ($\overline{\text{BR}}$), bus grant ($\overline{\text{BG}}$), and bus busy ($\overline{\text{BB}}$) signals. The arbitration protocol, which allows arbitration to overlap with bus activity, requires a single idle clock to prevent bus contention when transferring bus ownership between bus masters. The bus arbitration unit in the MCF5102 operates synchronously and transitions between states on the rising edge of BLCK.

The MCF5102 requests the bus from the external bus arbiter by asserting $\overline{BR}$ whenever an internal bus request is pending. The processor continues to assert $\overline{BR}$ for as long as it requires the bus. The processor negates $\overline{BR}$ at any time without regard to the status of $\overline{BG}$ and $\overline{BB}$. If the bus is granted to the processor when an internal bus request is generated, $\overline{BR}$ is asserted simultaneously with transfer start ($\overline{TS}$), allowing the access to begin immediately. The processor always drives $\overline{BR}$, and $\overline{BR}$ cannot be wire-ORed with other devices.

The external arbiter asserts $\overline{BG}$ to indicate to the processor that it has been granted the bus. If $\overline{BG}$ is negated while a bus cycle is in progress, the processor relinquishes the bus at the completion of the bus cycle. To guarantee that the bus is relinquished, $\overline{BG}$ must be negated prior to the rising edge of the BCLK in which the last $\overline{TA}$ or $\overline{TEA}$ is asserted. Note that the bus controller considers the four bus transfers for a burst-inhibited line transfer to be a single bus cycle and does not relinquish the bus until completion of the fourth transfer. The read and write portions of a locked read-modify-write sequence are divisible in the MCF5102, allowing the bus to be arbitrated away during the locked sequence. For system applications that do not allow locked sequences to be broken, the arbiter can use $\overline{LOCK}$ to detect locked accesses and prevent the negation of $\overline{BG}$ to the processor during these sequences.

When the bus has been granted to the processor in response to the assertion of $\overline{BR}$, one of two situations can occur. In the first situation, the processor monitors $\overline{BB}$ to determine when the bus cycle of the alternate bus master is complete. After the alternate bus master negates $\overline{BB}$, the processor asserts $\overline{BB}$ to indicate explicit bus ownership and begins the bus cycle by asserting $\overline{TS}$. The processor continues to assert $\overline{BB}$ until the external arbiter negates $\overline{BG}$, after which $\overline{BB}$ is first negated at the completion of the bus cycle, then forced to a high-impedance state. As long as $\overline{BG}$ is asserted, $\overline{BB}$ remains asserted to indicate the bus is owned, and the processor continuously drives the bus signals. The processor negates $\overline{BR}$ when there are no pending accesses to allow the external arbiter to grant the bus to the alternate bus master if necessary.

In the second situation, the processor samples $\overline{BB}$ until the external bus arbiter negates $\overline{BB}$. The processor drives its output pins with undetermined values and three-states $\overline{BB}$, but does not perform a bus cycle. This procedure, called implicit ownership of the bus, occurs when the processor is granted the bus but there are no pending bus cycles. If an internal access request is generated, the processor assumes explicit ownership of the bus and immediately begins an access, simultaneously asserting $\overline{BB}$, $\overline{BR}$, and $\overline{TS}$. If the external arbiter keeps $\overline{BG}$ asserted after completion of the bus cycle, the processor keeps $\overline{BB}$ asserted and drives the bus with undefined values, causing the processor to park. In this case, because $\overline{BB}$ remains asserted until the external arbiter negates $\overline{BG}$, the processor must assert $\overline{BR}$, and $\overline{TS}$ simultaneously to enter an active bus cycle. When it completes the active bus cycle and the external arbiter has not negated $\overline{BG}$, the processor goes back into park, negating $\overline{BR}$, and $\overline{TS}$. As long as $\overline{BG}$ is asserted, the processor oscillates between park and active bus cycles.

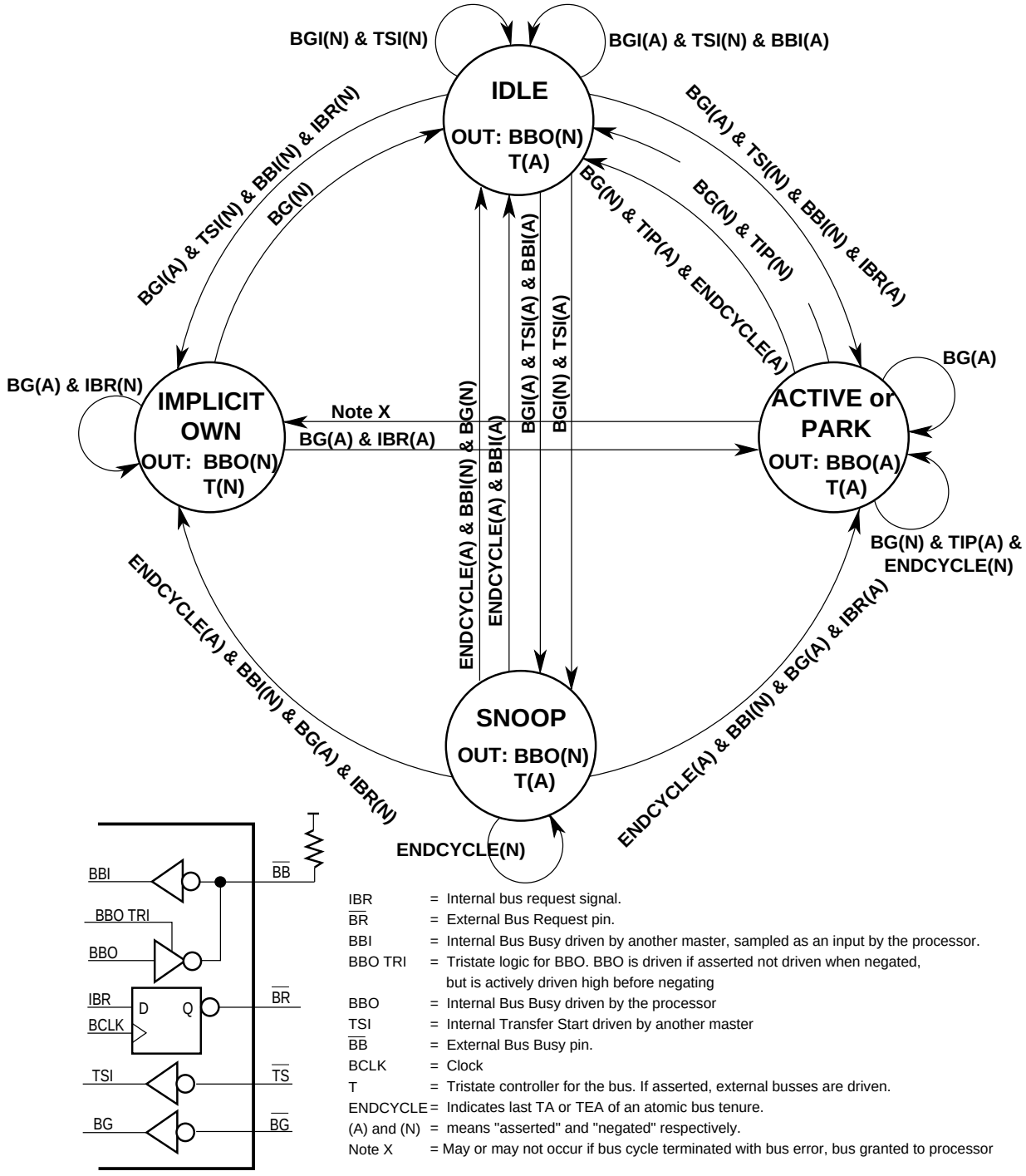
The MCF5102 can be in any one of five bus arbitration states during bus operation: idle, snoop, implicit ownership, park, and active bus cycle. There are two characteristics that determine these five states: whether the three-state logic determines if the MCF5102

drives the bus and how the MCF5102 drives $\overline{BB}$. If neither the processor nor the external bus arbiter asserts $\overline{BB}$, then an external pullup resistor drives $\overline{BB}$ high to negate it. Note that the relationship between the internal $\overline{BR}$ and the external $\overline{BR}$ is best described as a synchronous delay off BCLK.

The idle state occurs when the MCF5102 does not have ownership of the bus and is not in the process of snooping an access. In the idle state, $\overline{BB}$ is negated and the MCF5102 does not drive the bus. The snoop state is similar to the idle state in that the MCF5102 does not have ownership of the bus. The snoop state differs from the idle state in that the MCF5102 is ready to service snooped transfers. Otherwise, the status of $\overline{BB}$ and the bus is identical.

The implicit ownership state indicates that the MCF5102 owns the bus. The MCF5102 explicitly owns the bus when it runs a bus cycle immediately after being granted the bus. If the processor has completed at least one bus cycle and no internal transfers are pending, the processor drives the bus with undefined values, entering the park state. In either case, $\overline{BG}$ remains asserted. The simultaneous assertion of $\overline{BR}$, and $\overline{TS}$ allows the processor to leave the park state and enter the active bus cycle state.

Figure 7-10 is a bus arbitration state diagram illustrating the relationship of these five states with an example of an external bus arbiter circuit. Table 7-6 lists the five states and the conditions that indicate them.



IBR = Internal bus request signal.
$\overline{BR}$ = External Bus Request pin.
BBI = Internal Bus Busy driven by another master, sampled as an input by the processor.
BBO TRI = Tristate logic for BBO. BBO is driven if asserted not driven when negated, but is actively driven high before negating
BBO = Internal Bus Busy driven by the processor
TSI = Internal Transfer Start driven by another master
$\overline{BB}$ = External Bus Busy pin.
BCLK = Clock
T = Tristate controller for the bus. If asserted, external busses are driven.
ENDCYCLE = Indicates last TA or TEA of an atomic bus tenure.
(A) and (N) = means "asserted" and "negated" respectively.
Note X = May or may not occur if bus cycle terminated with bus error, bus granted to processor

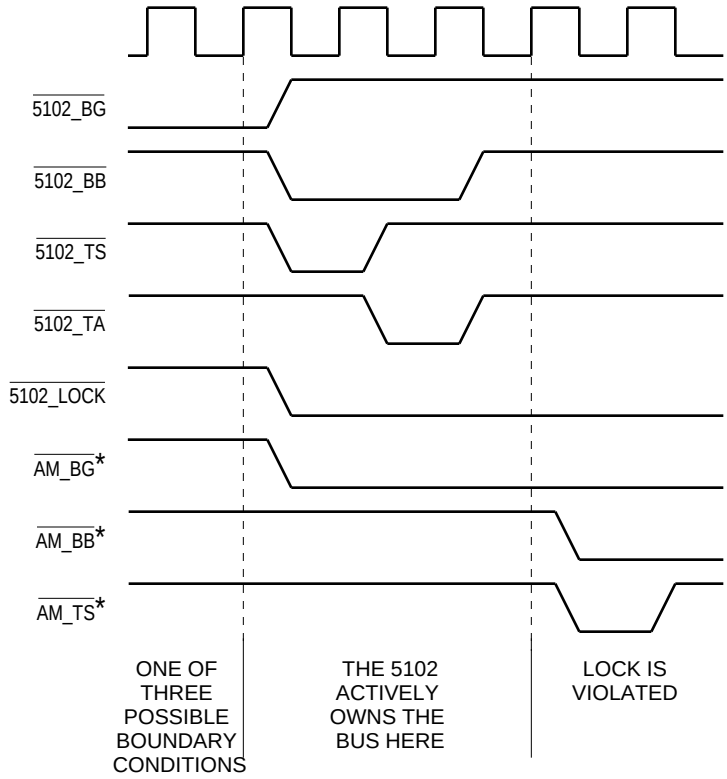
Figure 7-10. MCF5102 Internal Interpretation State Diagram and External Bus Arbiter Circuit

Table 7-6. MCF5102 Bus Arbitration States

State	Conditions
Idle	MCF5102 three-states $\overline{BB}$; arbiter negates $\overline{BG}$; bus is not driven.
Implicit Ownership	MCF5102 three-states $\overline{BB}$; arbiter asserts $\overline{BG}$; bus is driven with undefined values.
Active Bus Cycle or Park	MCF5102 asserts $\overline{BB}$; arbiter asserts $\overline{BG}$; bus is driven with defined values
Park	MCF5102 asserts $\overline{BB}$; arbiter asserts $\overline{BG}$; bus is driven with undefined values
Alternate Bus Master Ownership and Snooped	MCF5102 three-states $\overline{BB}$; arbiter asserts $\overline{BG}$; MCF5102 does not drive the bus.

The MCF5102 can be in the active bus cycle, park, or implicit ownership states when $\overline{BG}$ is negated. Depending on the state the processor is in when $\overline{BG}$ is negated, uncertain conditions can occur. The only guaranteed time that the processor relinquishes the bus is when $\overline{BG}$ is negated prior to the rising edge of BCLK in which the last $\overline{TA}$ or $\overline{TEA}$ is asserted and the processor is in the active bus cycle state. However, if the processor is in either the active bus cycle, park, or implicit ownership states and $\overline{BG}$ is negated at the same time or after the last $\overline{TA}$ or $\overline{TEA}$ is asserted, then from the standpoint of the external bus arbiter, the next action that the processor takes is undetermined because the processor can internally decide to perform another active bus cycle (indeterminate condition).

External bus arbiters must consider this indeterminate condition when negating $\overline{BG}$ and must be designed to examine the state of $\overline{BB}$ immediately after negating $\overline{BG}$ to determine whether or not the processor will run another bus cycle. A somewhat dangerous situation exists when the processor begins a locked transfer after the bus has been granted to the alternate bus master, causing the alternate bus master to perform a bus transfer during a locked sequence. To correct this situation, the external bus arbiter must be able to recognize the possible indeterminate condition and reassert $\overline{BG}$ to the processor when the processor begins a locked sequence. The indeterminate condition is most significant when dealing with systems that cannot allow locked transfers to be broken. Figure 7-11 illustrates an example of an error condition that is a consequence of the interaction between the indeterminate condition and a locked transfer. External bus arbiters must be designed so that all bus grants to all bus masters be nagated for at least one rising edge of BCLK between bus tenures; preventing bus conflicts resulting from the above conditions.



*AM indicates the alternate bus master.

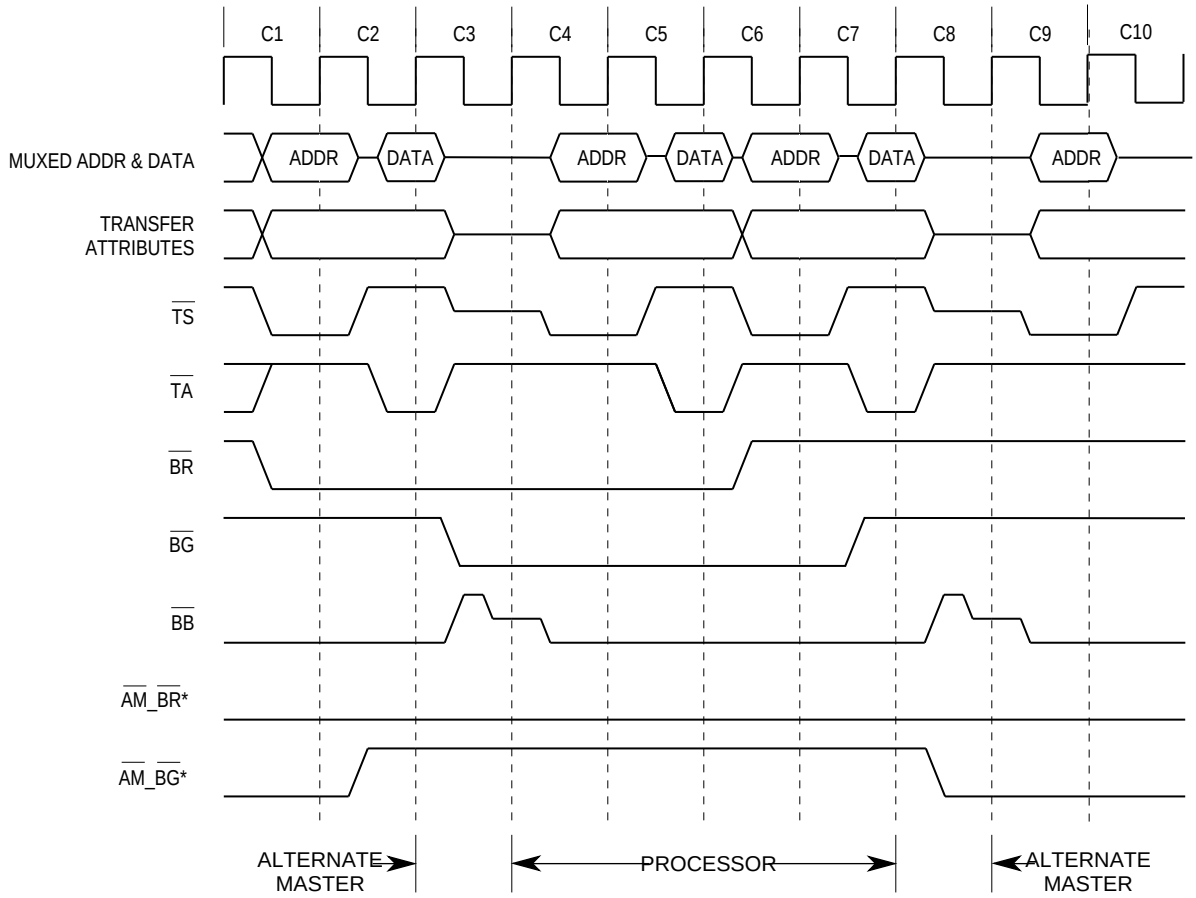
Figure 7-11. Lock Violation Example

In addition to the indeterminate condition, the external arbiter's design needs to include the function of $\overline{BR}$. For example, in certain cases associated with conditional branches, the MCF5102 can assert $\overline{BR}$ to request the bus from an alternate bus master, then negate $\overline{BR}$ without using the bus, regardless of whether or not the external arbiter eventually asserts $\overline{BG}$. This situation happens when the MCF5102 attempts to prefetch an instruction for a conditional branch. To achieve maximum performance, the processor prefetches the instructions of both paths for a conditional branch. If the conditional branch results in a branch-not-taken, the previously issued branch-taken prefetch is then terminated since the prefetch is no longer needed. In an attempt to save time, the MCF5102 negates $\overline{BR}$. If $\overline{BG}$ takes too long to assert, the MCF5102 enters a disregard request condition.

The $\overline{BR}$ signal can be reasserted immediately for a different pending bus request, or it can stay negated indefinitely. If an external bus arbiter is designed to wait for the MCF5102 to assert $\overline{BB}$ before proceeding, then the system experiences an extended period of time in which bus arbitration is locked. Motorola recommends that an external bus arbiter not assume that there is a direct relationship between $\overline{BR}$ and $\overline{BB}$ or $\overline{BR}$ and $\overline{BG}$ signals.

Figure 7-12 illustrates an example of the processor requesting the bus from the external bus arbiter. During C1, the MCF5102 asserts $\overline{BR}$ to request the bus from the arbiter, which negates the alternate bus master's $\overline{BG}$ signal and grants the bus to the processor by asserting $\overline{BG}$ during C3. During C3, the alternate bus master completes its current access and relinquishes the bus by three-stating all bus signals. Typically, the $\overline{BB}$ signal require a pullup resistor to maintain a logic-one level between bus master tenures. The alternate

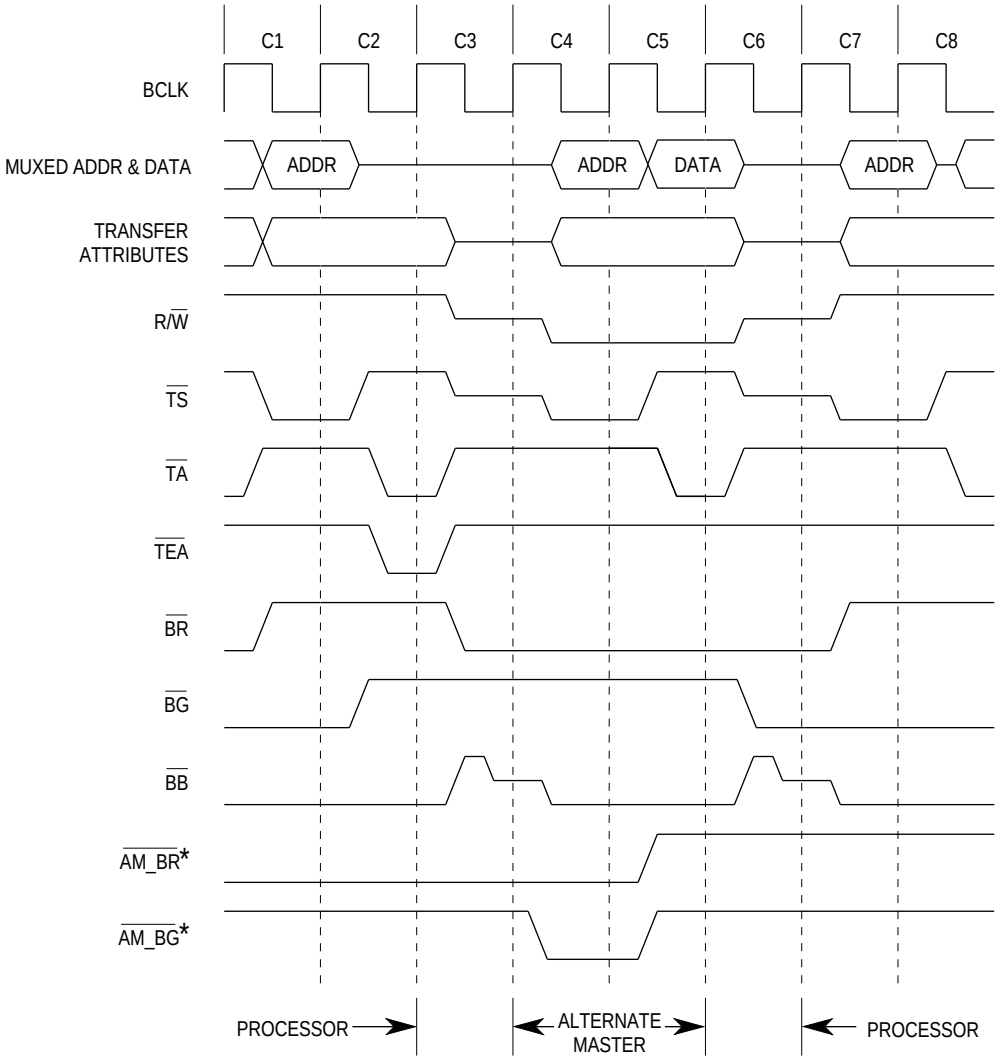
bus master should negate these signals before three-stating to minimize rise time of the signals and ensure that the processor recognizes the correct level on the next BCLK rising edge. At the end of C3, the processor recognizes the bus grant and bus idle conditions ($\overline{BG}$ asserted and $\overline{BB}$ negated) and assumes ownership of the bus by asserting $\overline{BB}$ and immediately beginning a bus cycle during C4. During C6, the processor begins the second bus cycle for the misaligned operand and negates $\overline{BR}$ since no other accesses are pending. During C7, the external bus arbiter grants the bus back to the alternate bus master that is waiting for the processor to relinquish the bus. The processor negates $\overline{BB}$ before three-stating these and all other bus signals during C8. Finally, the alternate bus master recognizes the bus grant and idle conditions at the end of C8 and is able to resume bus activity during C9.



*AM indicates the alternate bus master.

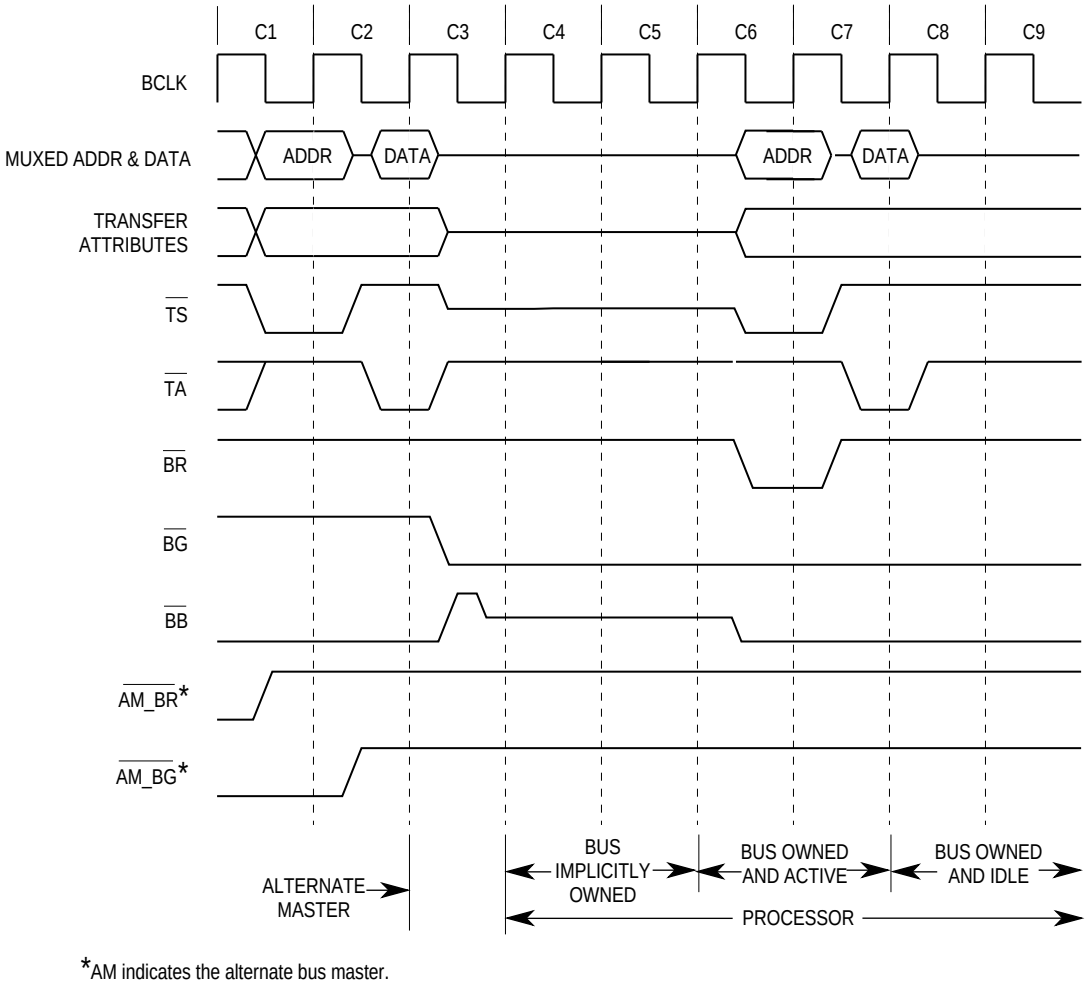
Figure 7-12. Processor Bus Request Timing

Figure 7-13 illustrates a functional timing diagram for an arbitration of a relinquish and retry operation. Figure 7-14 is a functional timing diagram for implicit ownership of the bus. In Figure 7-13, the processor read access that begins in C1 is terminated at the end of C2 with a retry request and $\overline{BG}$ negated, forcing the processor to relinquish the bus and allow the alternate master to access the bus. Note that the processor reasserts $\overline{BR}$ during C3 since the original access is pending again. After alternate bus master ownership, the bus is granted to the processor to allow it to retry the access beginning in C7.



*AM indicates the alternate bus master.

Figure 7-13. Arbitration During Relinquish and Retry Timing



*AM indicates the alternate bus master.

Figure 7-14. Implicit Bus Ownership Arbitration Timing

7.9 BUS SNOOPING OPERATION

When required, the MCF5102 can monitor alternate bus master transfers and intervene in the access to maintain cache coherency. The encoding of the SCx signals generated by the alternate bus master for each bus cycle controls the process of bus monitoring and intervention called snooping. Only byte, word, long-word, and line bus transfers can be snooped. Refer to **Section 4 Instruction and Data Caches** for SCx encodings.

When the MCF5102 recognizes that an alternate bus master has asserted $\overline{TS}$, the processor latches the level on the byte offset, SIZx, TMx, and R/W signals during the rising edge of BCLK for which $\overline{TS}$ is first asserted. The processor then evaluates the SCx and TTx signals to determine the type of access (TTx = $0 or $1), if it is snoopable, and, if so, how it should be snooped. If snooping is enabled for the access, the processor inhibits memory from responding by continuing to assert the memory inhibit signal ($\overline{MI}$) while checking the internal caches for matching lines. During the snooped bus cycle, the MCF5102 ignores all $\overline{TA}$ assertions while $\overline{MI}$ is asserted. Unless the data cache contains a dirty line corresponding to the access and the requested snoop operation indicates sink data for a write or source data for a read, $\overline{MI}$ is negated, and memory is allowed to respond and complete the access. Otherwise, the processor continues to intervene in the

access by keeping $\overline{MI}$ asserted and responding to the alternate bus master as a slave device. The processor monitors the levels of $\overline{TA}$, $\overline{TEA}$, and $\overline{TBI}$ to detect normal, bus error, retry, and burst-inhibited terminations. Note that for alternate bus master burst-inhibited line transfers, the MCF5102 snoops each of the four resulting long-word transfers. If snooping is disabled, $\overline{MI}$ is negated, and the MCF5102 counts the appropriate number of $\overline{TA}$ or $\overline{TEA}$ assertions before proceeding. For example, if the SIZx signals are pulled high, the MCF5102 requires four $\overline{TA}$ assertions, one $\overline{TEA}$ assertion, or one retry termination before proceeding.

In a system with multiple bus masters, the memory unit must wait for each snooping bus master to negate $\overline{MI}$ before responding to an access. A termination signal asserted before the negation of $\overline{MI}$ leads to undefined operation and must be avoided at all costs. Also, if the system contains multiple caching masters, then each master must access shared data using write-through memory that allow writes to the data to be snooped by other masters. The copyback caching mode is typically used for data local to a processor because in a multimaster caching system only one master at a time can access a given memory section of copyback data. The copyback caching mode also prevents multiple snooping processors from intervening in a specific access.

7.9.1 Snoop-Inhibited Cycle

For alternate bus master accesses in which the SCx signal encodings indicate that snooping is inhibited (SCx = $0), the MCF5102 immediately negates $\overline{MI}$ and allows memory to respond to the access. Snoop-inhibited alternate bus master accesses do not affect performance of the processor since no cache lookups are required. Figure 7-15 illustrates an example of a snoop-inhibited operation in which an alternate bus master is granted the bus for an access. No matter what the values are on the SCx and TTx signals, $\overline{MI}$ is asserted between bus cycles. Because $\overline{MI}$ is asserted while a cache lookup is performed, snooping inherently degrades system performance.

$\overline{MI}$ is asserted from the last $\overline{TA}$ of the current bus cycle if the MCF5102 owns the bus and loses it (see Figure 7-15). If an alternate bus master has the bus and loses it, there are two different resulting cases. Usually, an idle clock occurs between the alternate bus master's cycle and the MCF5102's cycle. If so, $\overline{MI}$ is asserted during the idle clock and negated from the same edge that the MCF5102 asserts the $\overline{TS}$ signal (see Figure 7-15). If there is no idle clock, $\overline{MI}$ is not asserted. $\overline{MI}$ is asserted during and after reset until the first bus cycle of the MCF5102. Even though snoop is inhibited, all $\overline{TA}$ or $\overline{TEA}$ assertions while $\overline{MI}$ is asserted are ignored. If a line snoop is started, the MCF5102 still requires four $\overline{TA}$ assertions.

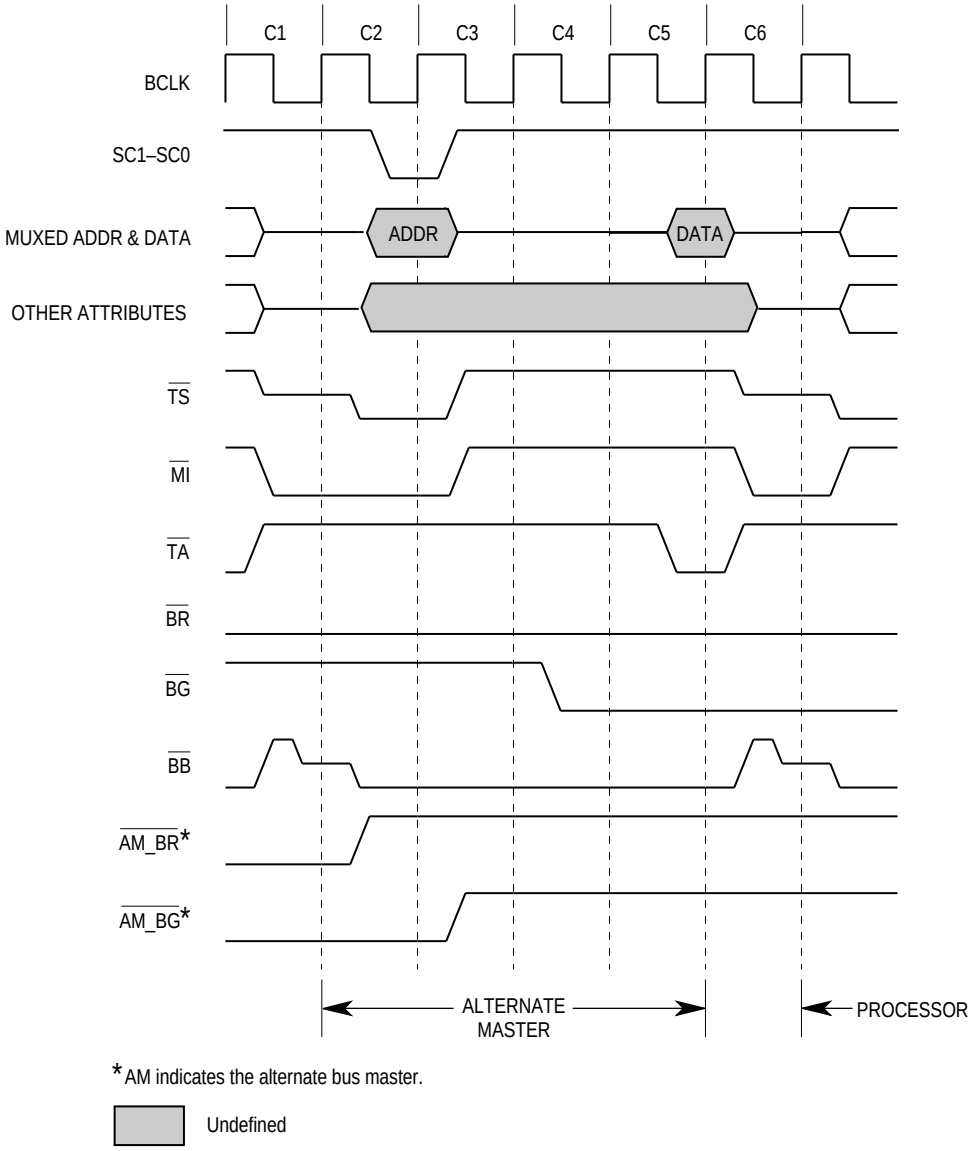


Figure 7-15. Snoop-Inhibited Bus Cycle

7.9.2 Snoop-Enabled Cycle (No Intervention Required)

For alternate bus master accesses in which SCx = $1 or $2, indicating that snooping is enabled, the MCF5102 continues to assert $\overline{MI}$ while checking for a matching cache line. If intervention in the alternate bus master access is not required, $\overline{MI}$ is then negated, and memory is allowed to respond and complete the access. Figure 7-16 illustrates an example of snooping in which memory is allowed to respond. Best-case timing is illustrated, which results in a memory access having the equivalent of two wait states. Variations in the timing required by snooping logic to access the caches can delay the negation of $\overline{MI}$ by up to two additional clocks. External logic must ensure that the termination signals as negated at all rising BCLK edges in which $\overline{MI}$ is asserted. Otherwise, if one of the termination signals is asserted, either the MCF5102 ignores all termination signals, reading them as negated, or the MCF5102 exhibits improper operation.

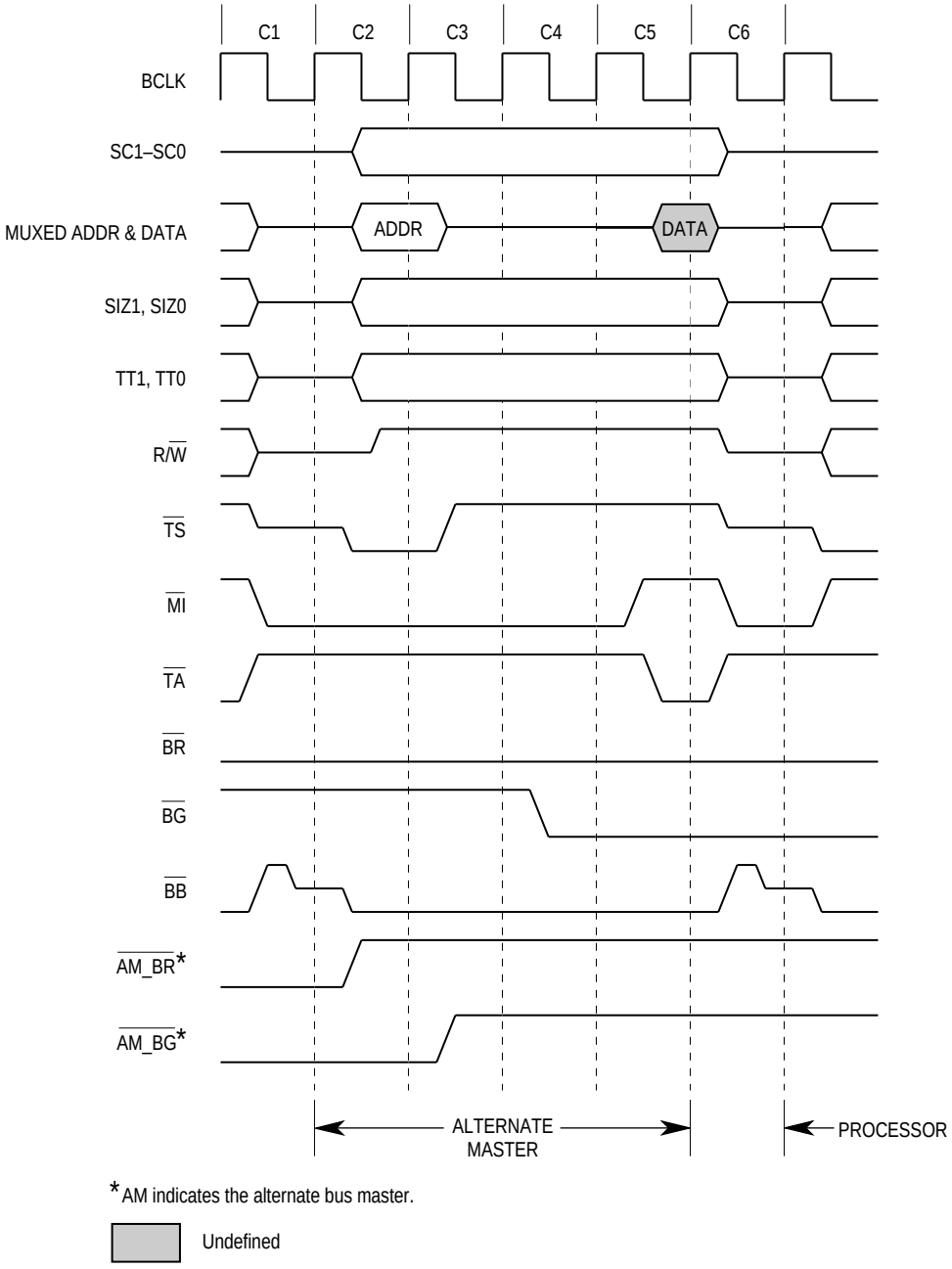


Figure 7-16. Snoop Access with Memory Response

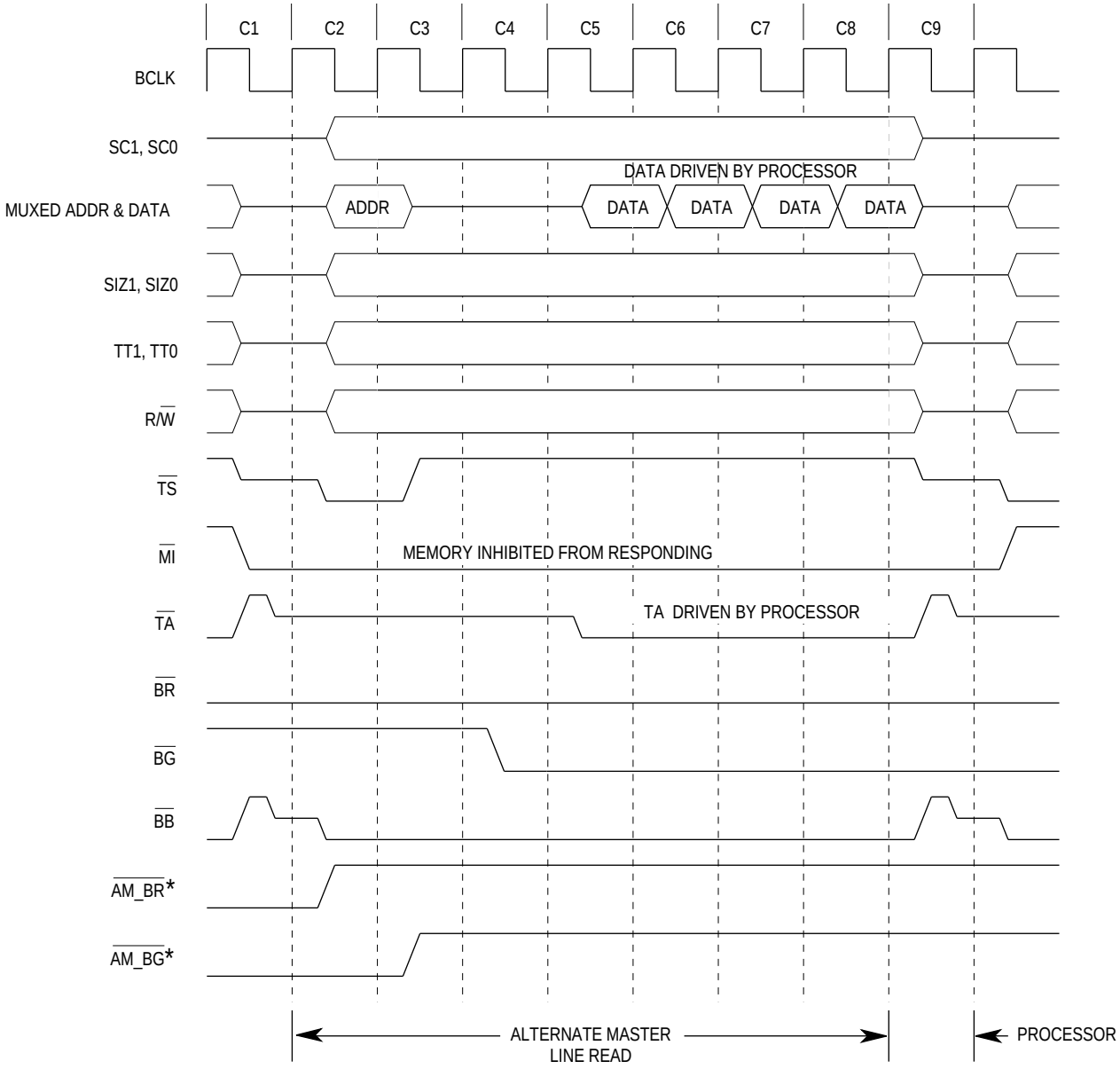
7.9.3 Snoop Read Cycle (Intervention Required)

If snooping is enabled for a read access and the corresponding data cache line contains dirty data, the MCF5102 inhibits memory and responds to the access as a slave device to supply the requested read data. Intervention in a byte, word, or long-word access is independent of which long-word entry in the cache line is dirty. Figure 7-17 illustrates an alternate bus master line read that hits a dirty line in the MCF5102 data cache. The processor asserts $\overline{TA}$ to acknowledge the transfer of data to the alternate bus master, and the data bus is driven with the four long words of data for the line. The timing illustrated is

for a best-case response time. Variations in the timing required by snooping logic to access the caches can delay the assertion of $\overline{TA}$ by up to two additional clocks.

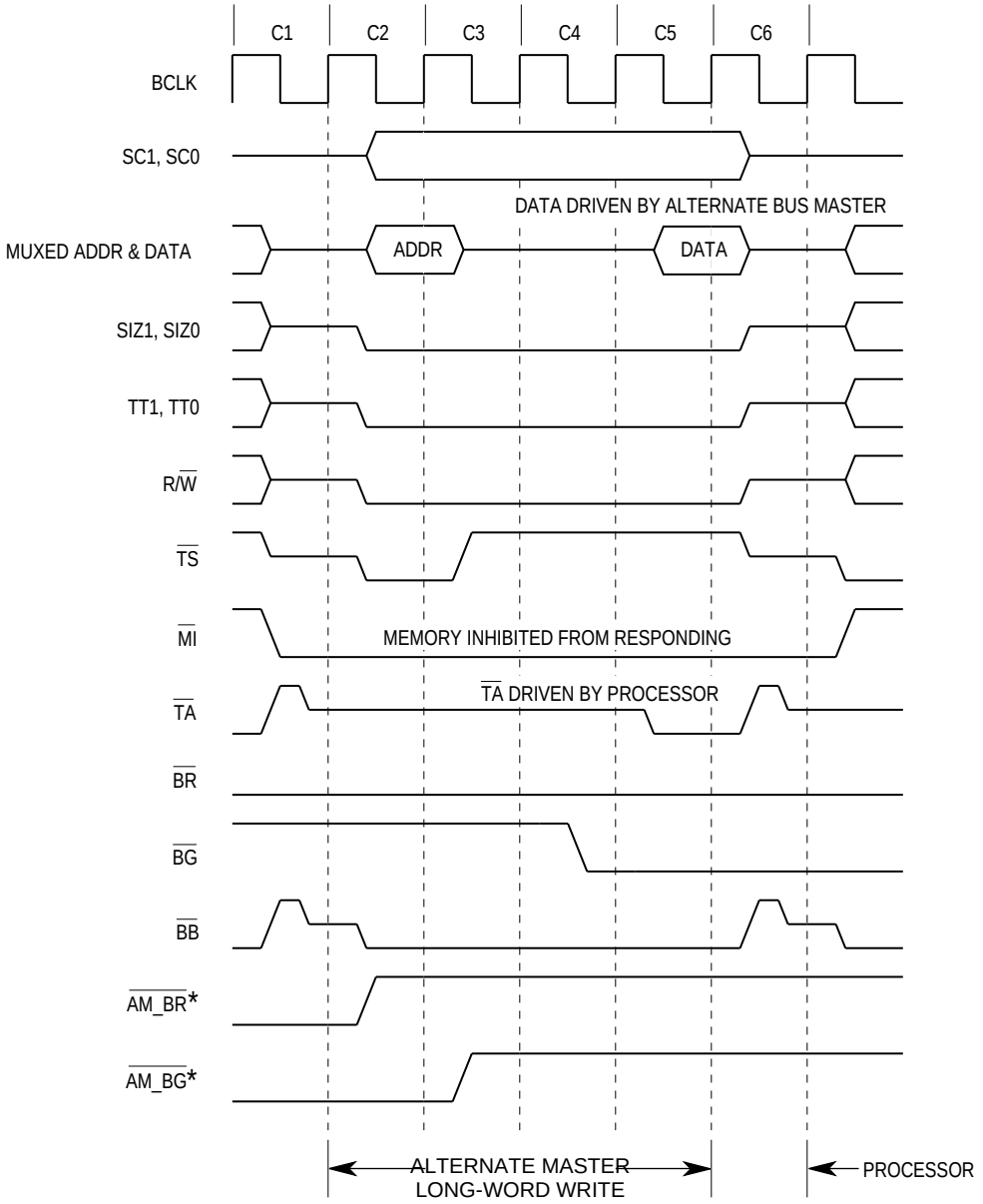
7.9.4 Snoop Write Cycle (Intervention Required)

If snooping with sink data is enabled for a byte, word, or long-word write access and the corresponding data cache line contains dirty data, the MCF5102 inhibits memory and responds to the access as a slave device to read the data from the bus and update the data cache line. The dirty bit is set for the long word changed in the cache line. Figure 7-18 illustrates a long-word write by an alternate bus master that hits a dirty line in the MCF5102 data cache. The processor asserts $\overline{TA}$ to acknowledge the transfer of data from the alternate master, and the processor reads the value on the data bus. The timing illustrated is for a best-case response time. Variations in the timing required by snooping logic to access the caches can delay the assertion of $\overline{TA}$ by up to two additional clocks.



*AM indicates the alternate bus master.

Figure 7-17. Snooped Line Read, Memory Inhibited



*AM indicates the alternate bus master.

Figure 7-18. Snooped Long-Word Write, Memory Inhibited

7.10 RESET OPERATION

An external device asserts the reset input signal ($\overline{RSTI}$) to reset the processor. When power is applied to the system, external circuitry should assert $\overline{RSTI}$ for a minimum of 10 BCLK cycles after V_{CC} is within tolerance. Figure 7-19 is a functional timing diagram of the power-on reset operation, illustrating the relationships among V_{CC}, $\overline{RSTI}$, mode selects, and bus signals. The BCLK clock signal is required to be stable by the time V_{CC} reaches the minimum operating specification. The V_{IH} levels of the clocks should not exceed V_{CC} while it is ramping up. $\overline{RSTI}$ is internally synchronized for two BCLKS before being used and must meet the specified setup and hold times to BCLK (specifications #51

and #52 in **Section 1o Electrical and Thermal Characteristics**) only if recognition by a specific BCLK rising edge is required. $\overline{MI}$ is asserted while the MCF5102 is in reset.

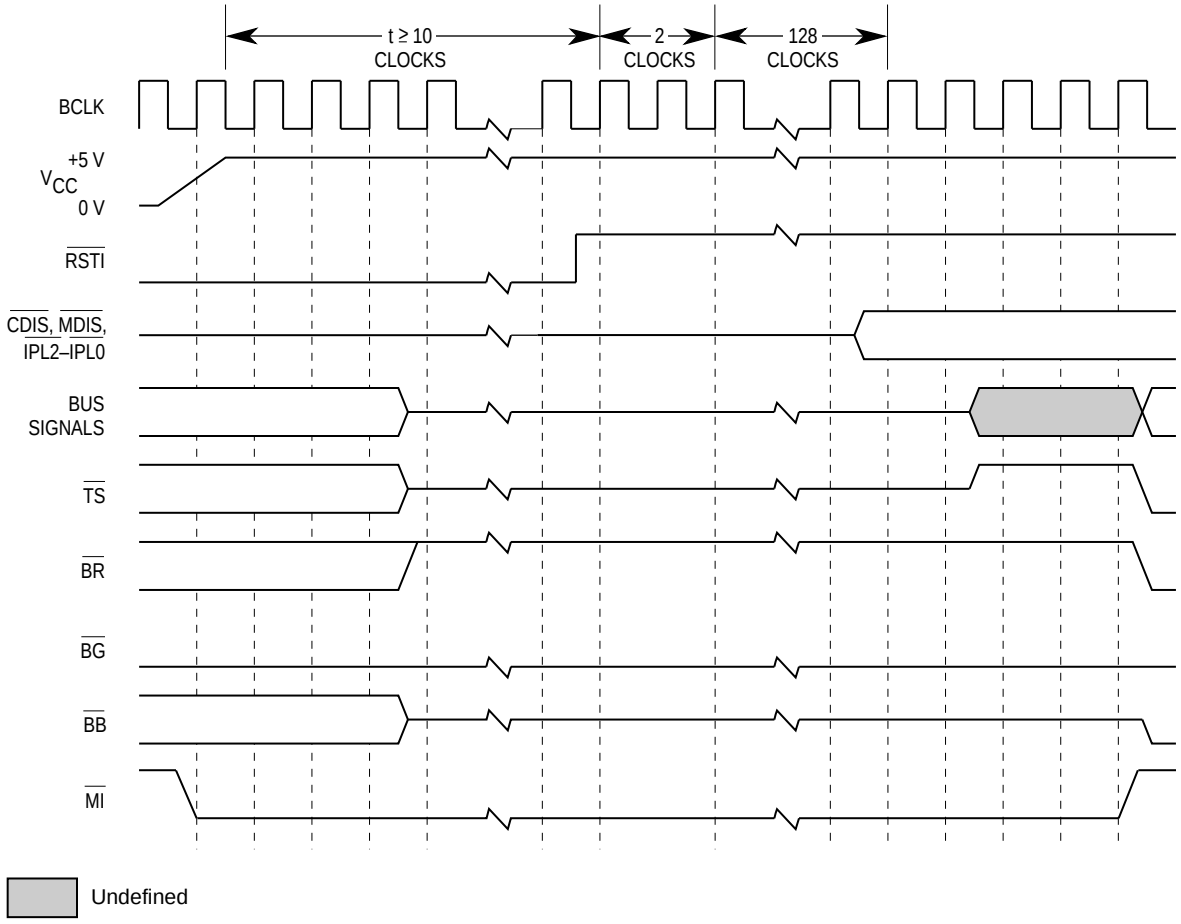


Figure 7-19. Initial Power-On Reset Timing

Once $\overline{RSTI}$ negates, the processor is internally held in reset for another 128 clock cycles. During the reset period, all signals that can be, are three-stated, and the rest are driven to their inactive state. Once the internal reset signal negates, all bus signals continue to remain in a high-impedance state until the processor is granted the bus. Afterwards, the first bus cycle for reset exception processing begins. In Figure 7-19 the processor assumes implicit bus ownership before the first bus cycle begins.

For processor resets after the initial power-on reset, $\overline{RSTI}$ should be asserted for at least 10 clock periods. Figure 7-20 illustrates timings associated with a reset when the processor is executing bus cycles. Note that $\overline{BB}$ (and $\overline{TA}$ if driven during a snooped access) are negated before transitioning to a three-state level.

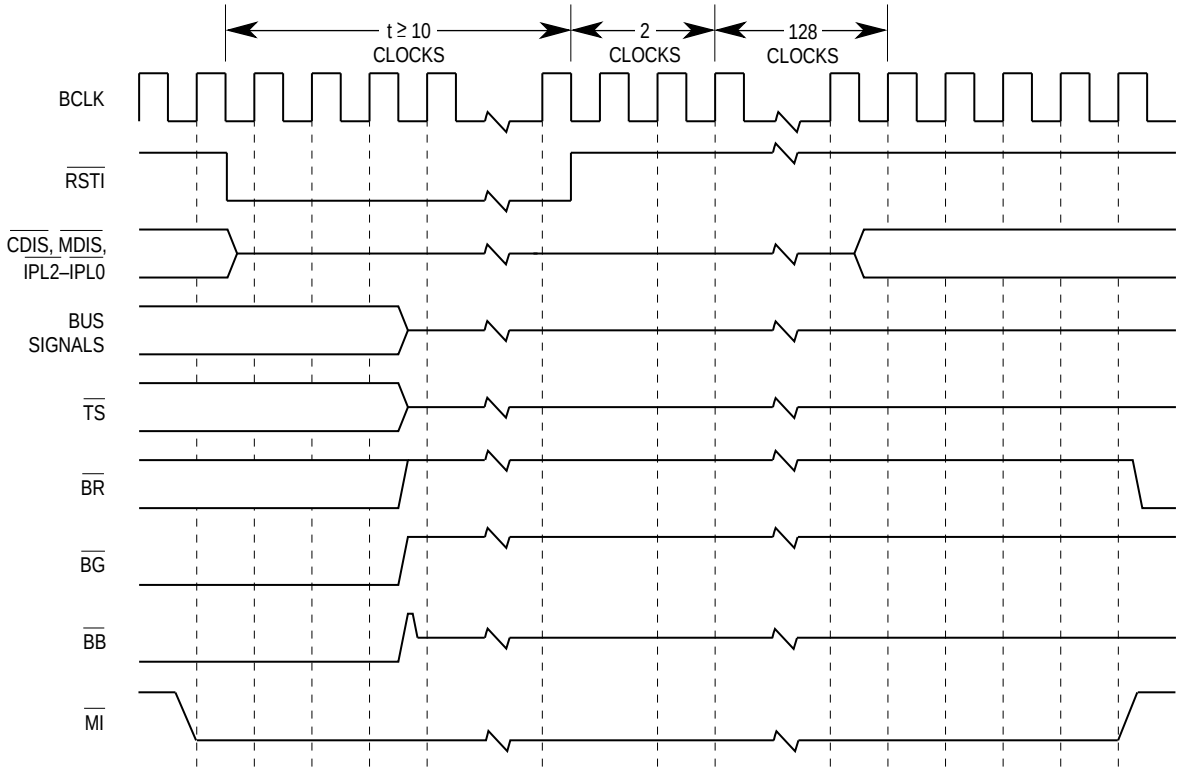


Figure 7-20. Normal Reset Timing

Resetting the processor causes any bus cycle in progress to terminate as if TA or TEA had been asserted. In addition, the processor initializes registers appropriately for a reset exception. **Section 8 Exception Processing** describes exception processing. When a RESET instruction is executed, the processor drives the reset out (RSTO) signal for 512 BCLK cycles. In this case, the processor resets the external devices of the system, and the internal registers of the processor are unaffected. The external devices connected to the RSTO signal are reset at the completion of the RESET instruction. An RSTI signal that is asserted to the processor during execution of a RESET instruction immediately resets the processor and causes the RSTO signal to negate. RSTO can be logically ANDed with the external signal driving RSTI to derive a system reset signal that is asserted for both an external processor reset and execution of a RESET instruction.

SECTION 8
EXCEPTION PROCESSING

Exception processing is the activity performed by the processor in preparing to execute a special routine for any condition that causes an exception. In particular, exception processing does not include execution of the routine itself. This section describes the processing for each type of integer unit exception, exception priorities, the return from an exception, and bus fault recovery. This section also describes the formats of the exception stack frames.

8.1 EXCEPTION PROCESSING OVERVIEW

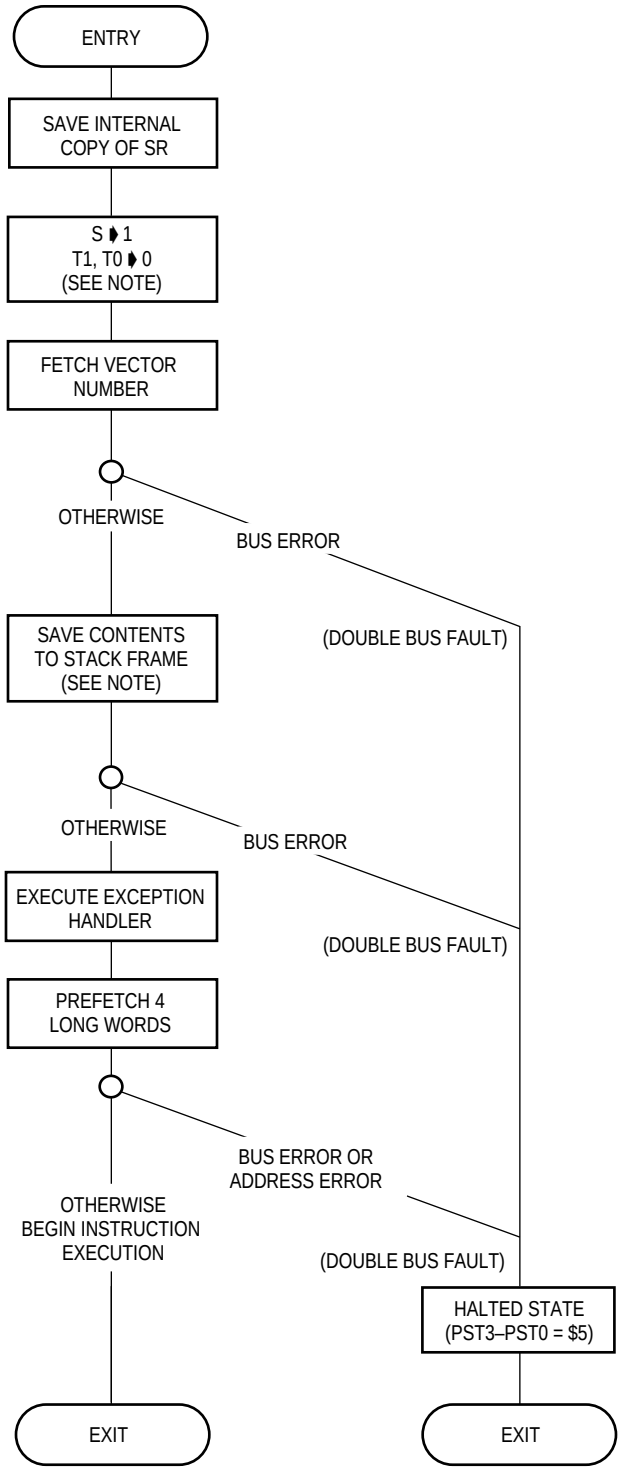
Exception processing is the transition from the normal processing of a program to the processing required for any special internal or external condition that preempts normal processing. External conditions that cause exceptions are interrupts from external devices, bus errors, and resets. Internal conditions that cause exceptions are instructions, address errors, and tracing. For example, the TRAP, TRAPcc, CHK, RTE, and DIV instructions can generate exceptions as part of their normal execution. In addition, illegal instructions and data types, and privilege violations cause exceptions. Exception processing uses an exception vector table and an exception stack frame. The following paragraphs describe the vector table and a generalized exception stack frame.

The MCF5102 uses a restart exception processing model to minimize interrupt and instruction latency and to reduce the size of the stack frame (compared to the frame required for a continuation model). Exceptions are recognized at each instruction boundary in the execute stage of the integer pipeline and force later instructions that have not yet reached the execute stage to be aborted. Instructions that cannot be interrupted, such as those that generate locked bus transfers or access serialized memory, are allowed to complete before exception processing begins.

Exception processing occurs in four functional steps. However, all individual bus cycles associated with exception processing (vector acquisition, stacking, etc.) are not guaranteed to occur in the order in which they are described in this section. Figure 8-1 illustrates a general flowchart for the steps taken by the processor during exception processing.

During the first step, the processor makes an internal copy of the status register (SR). Then the processor changes to the supervisor mode by setting the S-bit and inhibits tracing of the exception handler by clearing the trace enable (T1 and T0) bits in the SR. For the reset and interrupt exceptions, the processor also updates the interrupt priority mask in the SR.

During the second step, the processor determines the vector number for the exception. For interrupts, the processor performs an interrupt acknowledge bus cycle to obtain the vector number. For all other exceptions, internal logic provides the vector number. This vector number is used in the last step to calculate the address of the exception vector. Throughout this section, vector numbers are given in decimal notation.



NOTE: These blocks vary for reset and interrupt exceptions.

Figure 8-1. General Exception Processing Flowchart

The third step is to save the current processor contents for all exceptions other than reset. The processor creates one of five exception stack frame formats on the active supervisor stack and fills it with information appropriate for the type of exception. Other information can also be stacked, depending on which exception is being processed and the state of the processor prior to the exception. If the exception is an interrupt and the M-bit of the SR is set, the processor clears the M-bit and builds a second stack frame on the interrupt stack. Figure 8-2 illustrates the general form of the exception stack frame.

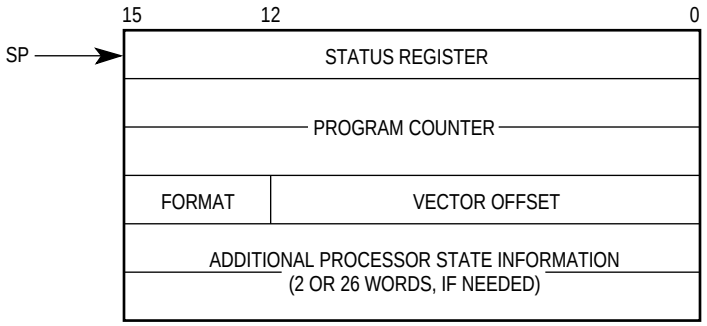


Figure 8-2. General Form of Exception Stack Frame

The last step initiates execution of the exception handler. The processor multiplies the vector number by four to determine the exception vector offset. It adds the offset to the value stored in the vector base register (VBR) to obtain the memory address of the exception vector. Next, the processor loads the program counter (PC) (and the interrupt stack pointer (ISP) for the reset exception) from the exception vector table entry. After prefetching the first four long words to fill the instruction pipe, the processor resumes normal processing at the address in the PC. When the processor executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires.

All exception vectors are located in the supervisor address space and are accessed using data references. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the exception vector table, the exception vector table can be located anywhere in memory it can even be dynamically relocated for each task that an operating system executes.

The MCF5102 supports a 1024-byte vector table containing 256 exception vectors (see Table 8-1). Motorola defines the first 64 vectors and reserves the other 192 vectors for user-defined interrupt vectors. External devices can use vectors reserved for internal purposes at the discretion of the system designer. External devices can also supply vector numbers for some exceptions. External devices that cannot supply vector numbers use the autovector capability, which allows the MCF5102 to automatically generate a vector number.

Table 8-1. Exception Vector Assignments

Vector Number(s)	Vector Offset (Hex)	Assignment
0	000	Reset Initial Interrupt Stack Pointer
1	004	Reset Initial Program Counter
2	008	Access Fault
3	00C	Address Error
4	010	Illegal Instruction
5	014	Integer Divide by Zero
6	018	CHK, CHK2 Instruction
7	01C	TRAPcc, TRAPV Instructions
8	020	Privilege Violation
9	024	Trace
10	028	Line 1010 Emulator (Unimplemented A-Line Opcode)
11	02C	Line 1111 Emulator (Unimplemented F-Line Opcode)
12	030	(Unassigned, Reserved)
13	034	Unassigned, Reserved)
14	038	Format Error
15	03C	Uninitialized Interrupt
16–23	040–05C	(Unassigned, Reserved)
24	060	Spurious Interrupt
25	064	Level 1 Interrupt Autovector
26	068	Level 2 Interrupt Autovector
27	06C	Level 3 Interrupt Autovector
28	070	Level 4 Interrupt Autovector
29	074	Level 5 Interrupt Autovector
30	078	Level 6 Interrupt Autovector
31	07C	Level 7 Interrupt Autovector
32–47	080–0BC	TRAP #0–15 Instruction Vectors
48–55	0C0–0DC	(Unassigned, Reserved)
56	0E0	(Unassigned, Reserved)
57	0E4	(Unassigned, Reserved)
58	0E8	(Unassigned, Reserved)
59–63	0EC–0FC	(Unassigned, Reserved)
64–255	100–3FC	User Defined Vectors (192)

8.2 INTEGER UNIT EXCEPTIONS

The following paragraphs describe the external interrupt exceptions and the different types of exceptions generated internally by the MCF5102 integer unit. The following exceptions are discussed:

- Access Fault
- Address Error
- Instruction Trap
- Illegal and Unimplemented Instructions
- Privilege Violation (PV)

- Trace
- Format Error
- Breakpoint Instruction
- Interrupt
- Reset

8.2.1 Access Fault Exception

An access fault exception occurs when a data or instruction prefetch access faults due to either an external bus error or an internal access fault. Both types of access faults are treated identically and the access fault exception handler or a status bit in the access fault stack frame distinguishes them. An access fault exception may or may not be taken immediately, depending on whether the faulted access specifically references data required by the execution unit or whether there are any other exceptions that can occur, allowing the execution pipeline to idle.

An external access fault (bus error) occurs when external logic aborts a bus cycle and asserts the $\overline{\text{TEA}}$ input signal. A bus error on a data write access always results in an access fault exception, causing the processor to begin exception processing immediately. A bus error on a data read also causes exception processing to begin immediately if the access is a byte, word, or long-word access or if the bus error occurs on the first transfer of a line read. Bus errors on the second, third, or fourth transfers for a data line read cause the transfer to be aborted, but result in a bus error only if the execution unit is specifically requesting the long word being transferred. For example, if a misaligned operand spans the first two long words in the line being read, a bus error on the second transfer causes an exception, but a bus error on the third or last transfer does not, unless the execution unit has generated another operand access that references data in these transfers.

Bus errors that occur during instruction prefetches are deferred until the processor attempts to use the information. For instance, if a bus error occurs while prefetching other instructions after a change-of-flow instruction (BRA, JMP, JSR, TRAP#n, etc.), BRA, JMP, JSR, TRAP#n execution of the new instruction flow clears the exception condition. This also applies to the not-taken branch for a conditional branch instruction, even though both sides of the branch are decoded.

Processor accesses for either data or instructions can result in internal access faults. Internal access faults must be corrected to complete execution of the current context. Four types of internal access faults can occur:

1. Push transfer faults occur when the execution unit is idle, the integer unit pipeline is frozen, the instruction and data cache requests are canceled (however, writes are not lost), and pending writes are stacked.
2. Data access faults occur when the bus controller and the execution unit are idle. A data access fault freezes the pipeline and cancels any pending instruction cache accesses. Pending writes are stacked because the data cache is deadlocked until stacking transfers are initiated.

3. Instruction access faults occur when the PC section is deadlocked because of the faulted data or another prefetch is required, the copyback stage is empty, and the data cache and bus controller are idle. Since instruction access faults are reset, they can be ignored.

When an exception is detected, all parts of the execution unit either remain or are forced to idle, at which time the highest priority exception is taken. Restarting the instruction or a user-defined supervisor cleanup exception handler routine regenerates lower priority exceptions on the return from exception handling. Internal access faults and bus errors are reported after all other pending integer instructions complete execution. If an exception is generated during completion of the earlier instructions, the pending instruction fault is cleared, and the new exception is serviced first. The processor restarts the pending prefetch after completing exception handling for the earlier instructions and takes a bus error exception if the access faults again. For data access faults, the processor aborts current instruction execution. If a data access fault is detected, the processor waits for the current instruction prefetch bus cycle to complete, then begins exception processing immediately.

As illustrated in Figure 8-1, the processor begins exception processing for an access fault by making an internal copy of the current SR. The processor then enters the supervisor mode and clears T1 and T0. The processor generates exception vector number 2 for the access fault vector. It saves the vector offset, PC, and internal copy of the SR on the stack. The saved PC value is the address of the instruction executing at the time the fault was detected. This instruction is not necessarily the one that initiated the bus cycle since the processor overlaps execution of instructions. It also saves information to allow continuation after a fault during a MOVEM instruction and to support other pending exceptions. The faulted address and pending write-back information is saved. The information saved on the stack is sufficient to identify the cause of the bus error, complete pending write-backs, and recover from the error. The exception handler must complete the pending write-backs. Up to three write-backs can be pending for push errors and data access errors.

If a bus error occurs during the exception processing for an access fault, address error, or reset or while the processor is loading internal state information from the stack during the execution of an RTE instruction, a double bus fault occurs, and the processor enters the halted state as indicated by the PST3–PST0 encoding $5. In this case, the processor does not attempt to alter the current state of memory. Only an external reset can restart a processor halted by a double bus fault.

The supervisor stack has special requirements to ensure that exceptions can be stacked. The stack must be resident with correct protection in the direction of growth to ensure that exception stacking never has a bus error or internal access fault. Memory allocated to the stack that are higher in memory than the current stack pointer can be nonresident since an RTE instruction can check for residency and trap before restoring the state.

8.2.2 Address Error Exception

An address error exception occurs when the processor attempts to prefetch an instruction from an odd address. This includes the case of a conditional branch instruction with an

odd branch offset that is not taken. A prefetch bus cycle is not executed, and the processor begins exception processing after the currently executing instructions have completed. If the completion of these instructions generates another exception, the address error exception is deferred, and the new exception is serviced. After exception processing for the address error exception commences, the sequence is the same as an access fault exception, except that the vector number is 3 and the vector offset in the stack frame refers to the address error vector. The stack frame is generated containing the address of the instruction that caused the address error and the address itself (A0 is cleared). If an address error occurs during the exception processing for a bus error, address error, or reset, a double bus fault occurs.

8.2.3 Instruction Trap Exception

Certain instructions are used to explicitly cause trap exceptions. The TRAP#n instruction always forces an exception and is useful for implementing system calls in user programs. The TRAPcc, TRAPV, CHK, and CHK2 instructions force exceptions if the user program detects an error, which can be an arithmetic overflow or a subscript value that is out of bounds. The DIVS and DIVU instructions force exceptions if a division operation is attempted with a divisor of zero.

As illustrated in Figure 8-1, when a trap exception occurs, the processor internally copies the SR, enters the supervisor mode, and clears T1 and T0. The processor generates a vector number according to the instruction being executed. Vector 5 is for DIVx, vector 6 is for CHK and CHK2, and vector 7 is for TRAPcc, and TRAPV instructions. For the TRAP#n instruction, the vector number is 32 plus n. The stack frame saves the trap vector offset, the PC, and the internal copy of the SR on the supervisor stack. The saved value of the PC is the address of the instruction following the instruction that caused the trap. For all instruction traps other than TRAP#n, a pointer to the instruction that caused the trap is also saved. Instruction execution resumes at the address in the exception vector after the required instruction is prefetched.

8.2.4 Illegal Instruction and Unimplemented Instruction

An illegal instruction exception corresponds to vector number 4, and occurs when the processor attempts to execute an illegal instruction. An illegal instruction is an instruction that contains any bit pattern that does not correspond to the bit pattern of a valid MCF5102 instruction. An illegal instruction exception is also taken after a breakpoint acknowledge bus cycle is terminated, either by the assertion of the transfer acknowledge ($\overline{TA}$) or the transfer error acknowledge ($\overline{TEA}$) signal. An illegal instruction exception can also be a MOVEC instruction with an undefined register specification field in the first extension word.

Instruction word patterns with bits 15–12 equal to $A do not correspond to legal instructions for the MCF5102 and are treated as unimplemented instructions. $A word patterns are referred to as an unimplemented instruction with A-line opcodes. When the processor attempts to execute an unimplemented instruction with an A-line opcode, an exception is generated with vector number 10, permitting efficient emulation of unimplemented instructions.

Exception processing for illegal and unimplemented instructions is similar to that for instruction traps. When the processor has identified an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. The processor copies the SR, enters the supervisor mode, and clears T1 and T0, disabling further tracing. The processor generates the vector number, either 4 or 10, according to the exception type. The illegal or unimplemented instruction vector offset, current PC, and copy of the SR are saved on the supervisor stack, with the saved value of the PC being the address of the illegal or unimplemented instruction. Instruction execution resumes at the address contained in the exception vector. It is the responsibility of the exception handling routine to adjust the stacked PC if the instruction is emulated in software or is to be skipped on return from the exception handler.

8.2.5 Privilege Violation Exception

To provide system security, some instructions are privileged. An attempt to execute one of the following privileged instructions while in the user mode causes a privilege violation exception:

ANDI to SR	MOVE from SR	MOVES	RESET
CINV	MOVE to SR	ORI to SR	RTE
CPUSH	MOVE USP	PFLUSH	STOP
EORI to SR	MOVEC	PTEST	

Exception processing for privilege violations is similar to that for illegal instructions. When the processor identifies a privilege violation, it begins exception processing before executing the instruction. As illustrated in Figure 8-1, the processor copies the SR, enters the supervisor mode, and clears the trace bits. The processor generates vector number 8, saves the privilege violation vector offset, the current PC value, and the internal copy of the SR on the supervisor stack. The saved value of the PC is the address of the first word of the instruction that caused the privilege violation. Instruction execution resumes after the required prefetches from the address in the privilege violation exception vector.

8.2.6 Trace Exception

The MCF5102 can be programmed to trace all instructions or only instructions that change program flow. In the trace mode, an instruction generates a trace exception after the instruction completes execution, allowing a debugging program to monitor execution of a program.

In general terms, a trace exception is an extension to the function of any traced instruction. The execution of a traced instruction is not complete until trace exception processing is complete. If an instruction does not complete due to an access fault or address error exception, trace exception processing is deferred until after execution of the suspended instruction is resumed. If an interrupt is pending at the completion of an instruction, trace exception processing occurs before interrupt exception processing starts. If an instruction forces an exception as part of its normal execution, the forced exception processing occurs before the trace exception is processed.

The T1 and T0 bits in the supervisor portion of the SR control tracing. The state of these bits when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. T1 and T0 bit = $1 causes an instruction that forces a change of flow to take a trace exception. The following instructions cause a trace exception to be taken when trace on change of flow is enabled.

ANDI to SR	CAS2	JMP	MOVE USP	RTE
Bcc (Taken)	CINV	JSR	NOP	RTS
BRA	CPUSH	MOVEC	ORI to SR	STOP
BSR	DBcc (Taken)	MOVES	RTR	
CAS	EORI to SR	MOVE to SR	RTD	

Instructions that increment the PC normally do not take the trace exception. This mode also includes SR manipulations because the processor must prefetch instruction words again to fill the pipeline any time an instruction that modifies the SR is executed. Table 8-2 lists the different trace modes.

Table 8-2. Tracing Control

T1	T0	Tracing Function
0	0	No Tracing
0	1	Trace on Change of Flow
1	0	Trace on Instruction Execution (Any Instruction)
1	1	Undefined, Reserved

When the processor is in the trace mode and attempts to execute an illegal or unimplemented instruction, that instruction does not cause a trace exception since the instruction is not executed. This is of particular importance to an instruction emulation routine that performs the instruction function, adjusts the stacked PC to skip the unimplemented instruction, and returns. Before returning, the trace bits of the SR on the stack should be checked. If tracing is enabled, the trace exception processing should also be emulated for the trace exception handler to account for the emulated instruction.

Trace exception processing starts at the end of normal processing for the traced instruction and before the start of the next instruction. As illustrated in Figure 8-1, the processor makes an internal copy of the SR, and enters the supervisor mode. It also clears the T1 and T0 bits of the SR, disabling further tracing. The processor supplies vector number 9 for the trace exception and saves the trace exception vector offset, PC value, and the internal copy of the SR on the supervisor stack. The saved value of the PC is the address of the next instruction to be executed. Instruction execution resumes after the required prefetches from the address in the trace exception vector.

When the STOP instruction is traced, the processor never enters the stopped condition. A STOP instruction that begins execution with the trace bits equal to $3 forces a trace exception after it loads the SR. Upon return from the trace exception handler, execution continues with the instruction following the STOP instruction, and the processor never enters the stopped condition.

8.2.7 Format Error Exception

Just as the processor checks for valid prefetched instructions, it also performs some checks of data values for control operations. The RTE instruction checks the validity of the stack format code. If any of these checks determine that the format of the data is improper, the instruction generates a format error exception. This exception saves a stack frame, generates exception vector number 14, and continues execution at the address in the format exception vector. The stacked PC value is the address of the instruction that detected the format.

8.2.8 Breakpoint Instruction Exception

To use the MCF5102 in a hardware emulator, the processor must provide a means of inserting breakpoints in the emulator code and performing appropriate operations at each breakpoint. Inserting an illegal instruction at the breakpoint and detecting the illegal instruction exception from its vector location can achieve this. However, since the VBR allows arbitrary relocation of exception vectors, the exception address cannot reliably identify a breakpoint. Consequently, the processor provides a breakpoint capability with a set of breakpoint exceptions, $4848–$484F.

When the MCF5102 executes a breakpoint instruction, it performs a breakpoint acknowledge cycle (read cycle) with an acknowledge transfer type and transfer modifier value of $0. Refer to **Section 7 Bus Operation** for a description of the breakpoint acknowledge cycle. After external hardware terminates the bus cycle with either $\overline{\text{TA}}$ or $\overline{\text{TEA}}$, the processor performs illegal instruction exception processing.

8.2.9 Interrupt Exception

When a peripheral device requires the services of the MCF5102 or is ready to send information that the processor requires, it can signal the processor to take an interrupt exception using the active-low $\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$ signals. The three signals encode a value of 0–7 ($\overline{\text{IPL0}}$ is the least significant bit). High levels on all three signals correspond to no interrupt requested (level 0). Values 1–7 specify one of seven levels of interrupts, with level 7 having the highest priority. Table 8-3 lists the interrupt levels, the states of $\overline{\text{IPL2}}$–$\overline{\text{IPL0}}$ that define each level, and the SR interrupt mask value that allows an interrupt at each level.

Table 8-3. Interrupt Levels and Mask Values

Requested Interrupt Level	Control Line Status			Interrupt Mask Level Required for Recognition
	$\overline{IPL2}$	$\overline{IPL1}$	$\overline{IPL0}$	
0	High	High	High	No Interrupt Requested
1	High	High	Low	0
2	High	Low	High	0–1
3	High	Low	Low	0–2
4	Low	High	High	0–3
5	Low	High	Low	0–4
6	Low	Low	High	0–5
7	Low	Low	Low	0–7

When an interrupt request has a priority higher than the value in the interrupt priority mask of the SR (bits 10–8), the processor makes the request a pending interrupt. Priority level 7, the nonmaskable interrupt, is a special case. Level 7 interrupts cannot be masked by the interrupt priority mask, and they are transition sensitive. The processor recognizes an interrupt request each time the external interrupt request level changes from some lower level to level 7, regardless of the value in the mask. Figure 8-3 shows two examples of interrupt recognition, one for level 6 and one for level 7. When the MCF5102 processes a level 6 interrupt, the SR mask is automatically updated with a value of 6 before entering the handler routine so that subsequent level 6 interrupts and lower level interrupts are masked. Provided no instruction that lowers the mask value is executed, the external request can be lowered to level 3 and then raised back to level 6 and a second level 6 interrupt is not processed. However, if the MCF5102 is handling a level 7 interrupt (SR mask set to level 7) and the external request is lowered to level 3 and than raised back to level 7, a second level 7 interrupt is processed. The second level 7 interrupt is processed because the level 7 interrupt is transition sensitive. A level comparison also generates a level 7 interrupt if the request level and mask level are at 7 and the priority mask is then set to a lower level (with the MOVE to SR or RTE instruction, for example). The level 6 interrupt request and mask level example in Figure 8-3 is the same as for all interrupt levels except 7.

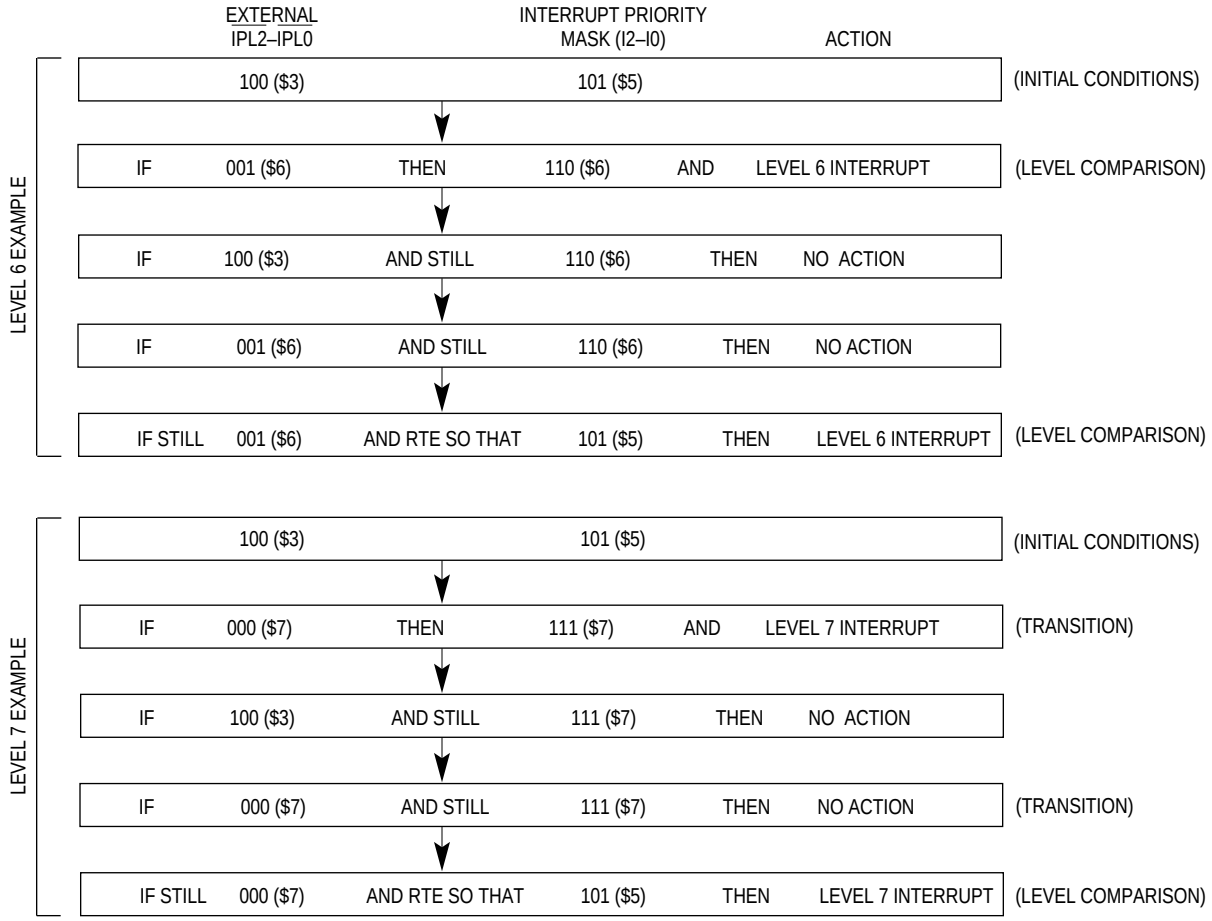


Figure 8-3. Interrupt Recognition Examples

Note that a mask value of 6 and a mask value of 7 both inhibit request levels of 1–6 from being recognized. In addition, neither masks a transition to an interrupt request level of 7. The only difference between mask values of 6 and 7 occurs when the interrupt request level is 7 and the mask value is 7. If the mask value is lowered to 6, a second level 7 interrupt is recognized.

External circuitry can chain or otherwise merge signals from devices at each level, allowing an unlimited number of devices to interrupt the processor. When several devices are connected to the same interrupt level, each device should hold its interrupt priority level constant until its corresponding interrupt acknowledge bus cycle ensures that all requests are processed. Refer to **Section 7 Bus Operation** for details on the interrupt acknowledge cycle.

Figure 8-4 illustrates a flowchart for interrupt exception processing. When processing an interrupt exception, the processor first makes an internal copy of the SR, sets the mode to supervisor, suppresses tracing, and sets the processor interrupt mask level to the level of the interrupt being serviced. The processor attempts to obtain a vector number from the interrupting device using an interrupt acknowledge bus cycle with the interrupt level number output on the transfer modifier signals. For a device that cannot supply an interrupt vector, the autovector signal ($\overline{AVEC}$) must be asserted. In this case, the MCF5102 uses an internally generated autovector, which is one of vector numbers 25–31, that corresponds to the interrupt level number (see Table 8-1). If external logic indicates a bus error during the interrupt acknowledge cycle, the interrupt is considered spurious, and the processor generates the spurious interrupt vector number, 24.

Once the vector number is obtained, the processor saves the exception, PC value, and the internal copy of the SR on the active supervisor stack. The saved value of the PC is the address of the instruction that would have been executed had the interrupt not occurred.

If the M-bit of the SR is set, the processor clears the M-bit and creates a throwaway exception stack frame on top of the interrupt stack as part of interrupt exception processing. This second frame contains the same PC value and vector offset as the frame created on top of the master stack, but has a format number of $1. The copy of the SR saved on the throwaway frame has the S-bit set, the M-bit clear, and the interrupt mask level set to the new interrupt level. It may or may not be set in the copy saved on the master stack. The resulting SR (after exception processing) has the S-bit set and the M-bit cleared. The processor loads the address in the exception vector into the PC, and normal instruction execution resumes after the required prefetches for the interrupt handler routine.

Most M68000 family peripherals use programmable interrupt vector numbers as part of the interrupt acknowledge operation for the system. If this vector number is not initialized after reset and the peripheral must acknowledge an interrupt request, the peripheral usually returns the vector number for the Uninitialized interrupt vector, 15.

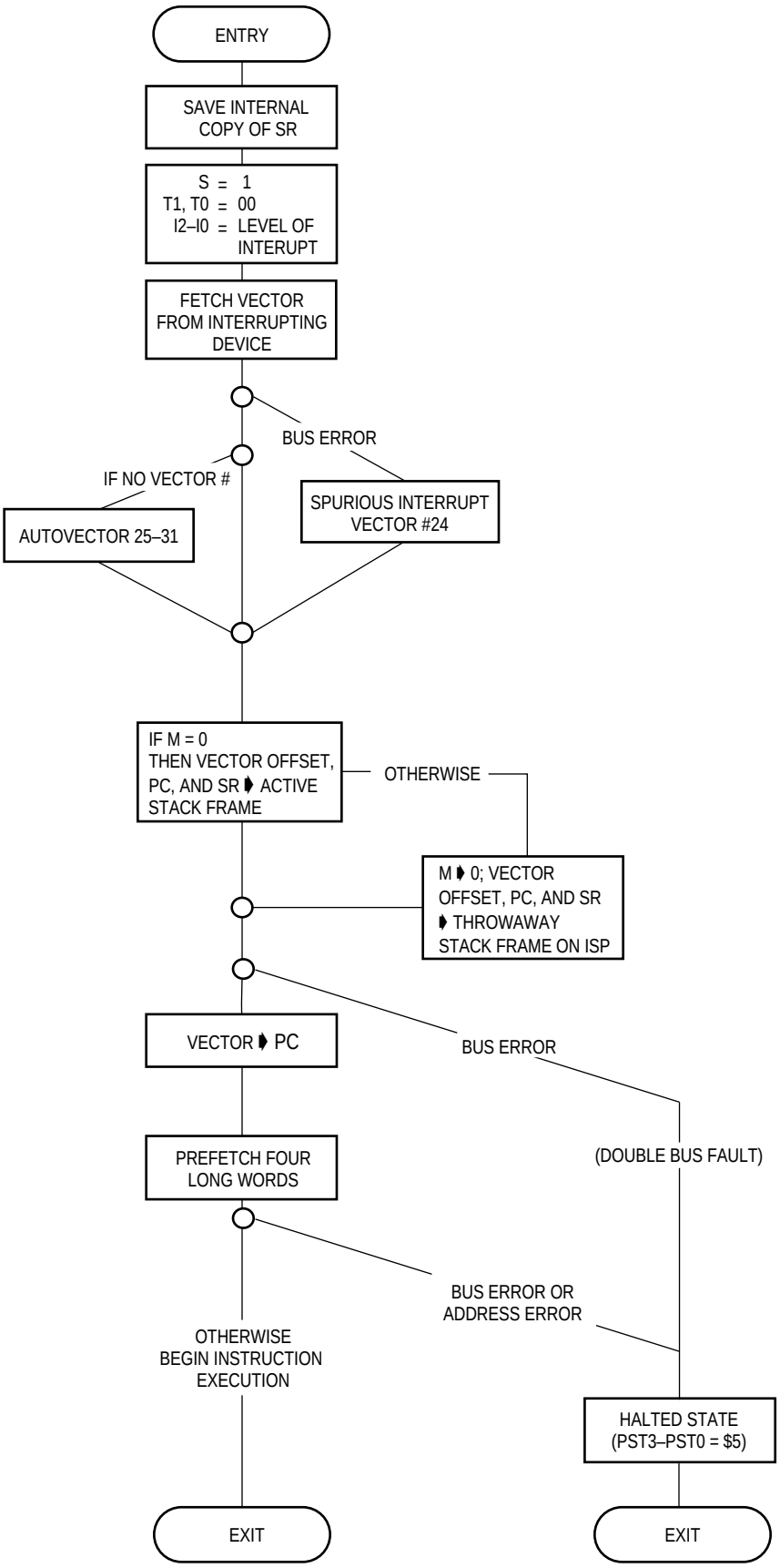


Figure 8-4. Interrupt Exception Processing Flowchart

8.2.10 Reset Exception

Asserting the reset in input signal causes a reset exception. The reset exception has the highest priority of any exception it provides for system initialization and recovery from catastrophic failure. Reset also aborts any processing in progress when $\overline{\text{RSTI}}$ is recognized processing cannot be recovered. Figure 8-5 is a flowchart of the reset exception processing.

The reset exception places the processor in the interrupt mode of the supervisor privilege mode by setting the S-bit and clearing the M-bit and disables tracing by clearing the T1 and T0 bits in the SR. This exception also sets the processor's interrupt priority mask in the SR to the highest level, level 7. Next the VBR is initialized to zero ($00000000), and the enable bits in the cache control register (CACR) for the on-chip caches are cleared. The reset exception also clears the enable bit in the .access control registers. An interrupt acknowledge bus cycle is begun to generate a vector number. This vector number references the reset exception vector (two long words, vector numbers 0 and 1) at offset zero in the supervisor address space. The first long word is loaded into the interrupt stack pointer, and the second long word is loaded into the PC. Reset exception processing concludes with the prefetch of the first four long words beginning at the memory location pointed to by the PC.

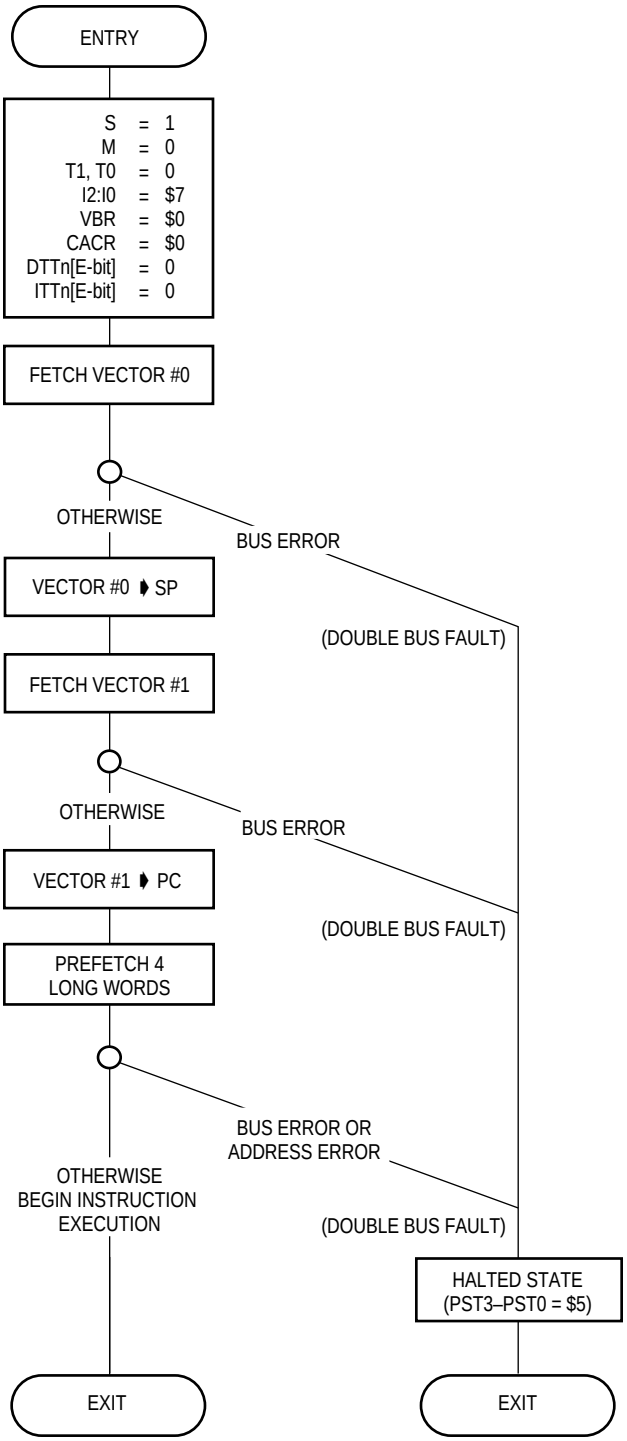


Figure 8-5. Reset Exception Processing Flowchart

After the initial instruction is prefetched, program execution begins at the address in the PC. The reset exception does not flush the PVs or invalidate entries in the instruction or data caches it does not save the value of either the PC or the SR. If an access fault or address error occurs during the exception processing sequence for a reset, a double bus fault is generated. The processor halts, and the processor status (PST3–PST0) signals indicate $5. Execution of the reset instruction does not cause a reset exception, or affect

any internal registers, but it does cause the MCF5102 to assert the reset out ($\overline{\text{RSTO}}$) signal, resetting all external devices.

8.3 EXCEPTION PRIORITIES

When several exceptions occur simultaneously, they are processed according to a fixed priority. Table 8-4 lists the exceptions, grouped by characteristics. Each group has a priority, from 0–6, with 0 as the highest priority.

Table 8-4. Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristics
0	Reset	Aborts all processing (instruction or exception) and does not save old context.
1	Data Access Error (PV Fault or Bus Error)	Aborts current instructions
2	BKPT #n, CHK, CHK2, Divide by Zero, RTE, TRAP#n, TRAPV Illegal Instruction, Unimplemented A- Line, Privilege Violation	Exception processing is part of instruction execution. Exception processing begins before instruction is executed.
3	Address Error	Reported after all previous instructions and associated exceptions have completed.
4	Trace	Exception processing begins when current instruction or previous exception processing has completed.
5	Instruction Access Error (PV Fault or Bus Error)	Reported after all previous instructions and associated exceptions have completed.
6	Interrupt	Exception processing begins when current instruction or previous exception processing has completed.

The method used to process exceptions in the MCF5102 is significantly different from that used in earlier members of the M68000 processor family due to the restart exception model. In general, when multiple exceptions are pending, the exception with the highest priority is processed first, and the remaining exceptions are regenerated when the current instruction restarts. Note that the reset operation clears all other exceptions except in the following circumstances:

- As soon as the MCF5102 has completed exception processing for a condition when an interrupt exception is pending, it begins exception processing for the interrupt exception instead of executing the exception handler for the original exception condition. For example, if simultaneous interrupt and trap exceptions are pending, the exception processing for the trap exception occurs first, followed immediately by exception processing for the interrupt. When the processor resumes normal instruction execution, it is in the interrupt handler, which returns to the trap exception handler.

- Exception processing for access error exceptions creates a format $7 stack frame that contains status information that can indicate a pending trace. The RTE instruction used to return from the access error exception handler checks the status bits for one of these pending exceptions. If one is indicated, the RTE changes the access error stack frame to match the pending exception and fetches the vector for the exception. Instruction execution then resumes in the new exception handler.
- If a trace exception is pending at the same time an exception priority level 2, the trace exception is not reported, and the exception handler for the other exception condition must check for the trace condition.

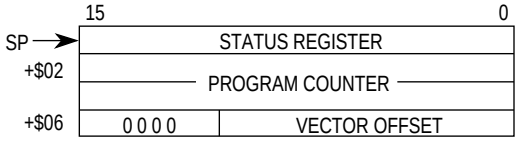
8.4 RETURN FROM EXCEPTIONS

After the processor has completed executing the exception handlers for all pending exceptions, the processor resumes normal instruction execution at the address in the processor's vector table for the last exception processed. Once the exception handler has completed execution, if possible the processor must return the system context as it was prior to the exception using the RTE instruction. (If the internal data of the exception stack frames are manipulated, MCF5102 may enter into an undefined state this applies specifically to the SSW on the access error stack frame.)

When the processor executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires. If during restoration, a stack frame has an odd address PC and an SR that indicates user trace mode enabled, then an address error is taken. The SR stacked for the address error has the SR S-bit set. When the MCF5102 writes or reads a stack frame, it uses long-word operand transfers wherever possible. Using a long-word-aligned stack pointer greatly enhances exception processing performance. The processor does not necessarily read or write the stack frame data in sequential order. The system software should not depend on a particular exception generating a particular stack frame. For compatibility with future devices, the software should be able to handle any format of stack frame for any type of exception. The following paragraphs discuss in detail each stack frame format.

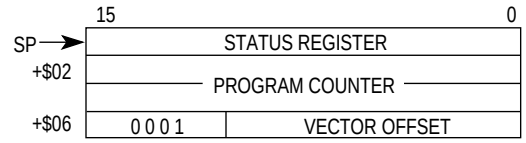
8.4.1 Four-Word Stack Frame (Format $0)

If a four-word stack frame is on the active stack and an RTE instruction is encountered, the processor updates the SR and PC with the data read from the stack, increments the stack pointer by eight, and resumes normal instruction execution.

Stack Frames	Exception Types	Stacked PC Points To
 FOUR-WORD STACK FRAME–FORMAT $0	• Interrupt • Format Error • TRAP #N • Illegal Instruction • A-Line Instruction • Privilege Violation	• Next Instruction • RTE or RESTORE Instruction • Next Instruction • Illegal Instruction • A-Line Instruction • First Word of Instruction Causing Privilege Violation

8.4.2 Four-Word Throwaway Stack Frame (Format $1)

If a four-word throwaway stack frame is on the active stack and an RTE instruction is encountered, the processor increments the active stack pointer by eight, updates the SR with the value read from the stack, and then begins RTE processing again, as illustrated in Figure 8-6. The processor reads a new format word from the stack frame on top of the active stack (which may or may not be the same stack used for the previous operation) and performs the proper operations corresponding to that format. In most cases, the throwaway frame is on the interrupt stack, and when the SR value is read from the stack, the S-bit and M-bit are set. In that case, there is a normal four-word frame on the master stack. However, the second frame can be any format (even another throwaway frame) and can reside on any of the three system stacks.

Stack Frames	Exception Types	Stacked PC Points To
 THROWAWAY FOUR-WORD STACK FRAME–FORMAT $1	• Created on interrupt stack during interrupt exception processing when transition from master state to interrupt state occurs.	• Next Instruction: same as on master stack.

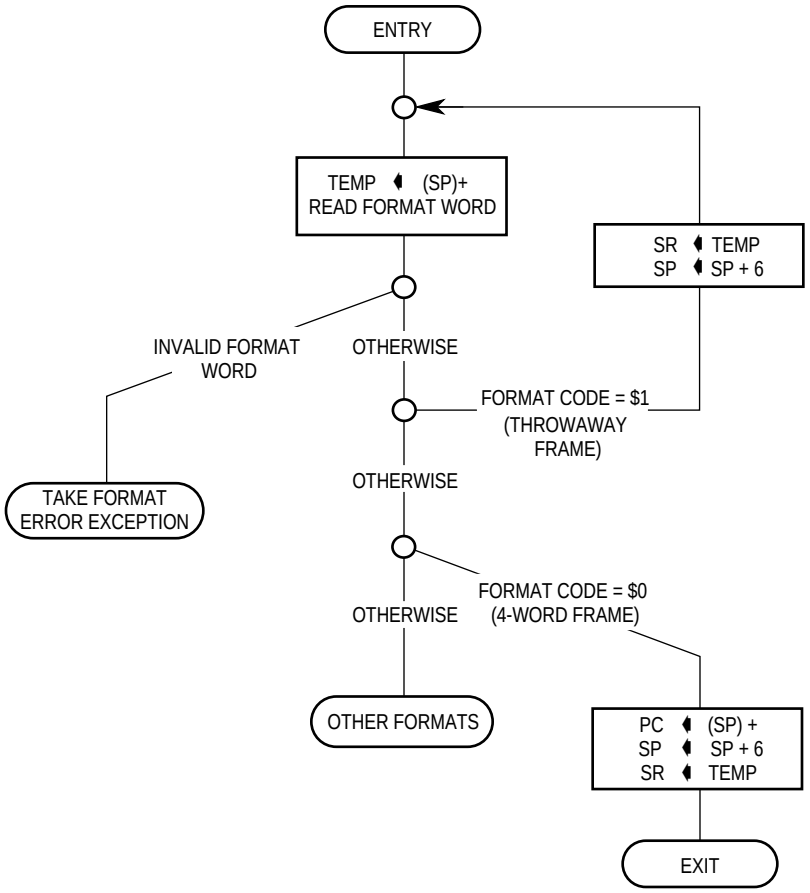
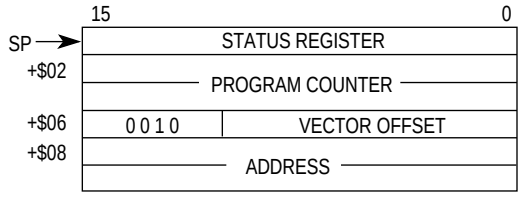


Figure 8-6. Flowchart of RTE Instruction for Throwaway Four-Word Frame

8.4.3 Six-Word Stack Frame (Format $2)

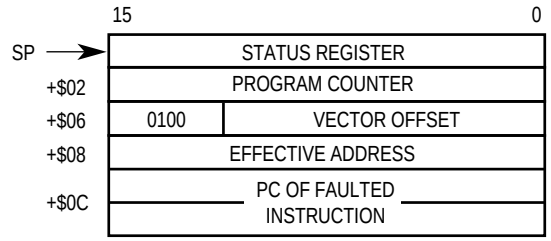
If a six-word throwaway stack frame is on the active stack and an RTE instruction is encountered, the processor restores the SR and PC values from the stack, increments the active supervisor stack pointer by $C, and resumes normal instruction execution.

Stack Frames	Exception Types	Stacked PC Points To
 SIX-WORD STACK FRAME–FORMAT $2	• CHK, CHK2, TRAPcc, TRAPV, Trace, or Zero Divide • Address Error	• Next Instruction: address is the address of the instruction that caused the exception. • Instruction that caused the address error, address is the reference address – 1.

8.4.4 Eight-Word Stack Frames

If an eight-word throwaway stack frame is on the active stack and an RTE is encountered, the processor restores the SR and PC values from the stack, increments the active supervisor stack pointer by $10, and resumes normal execution.

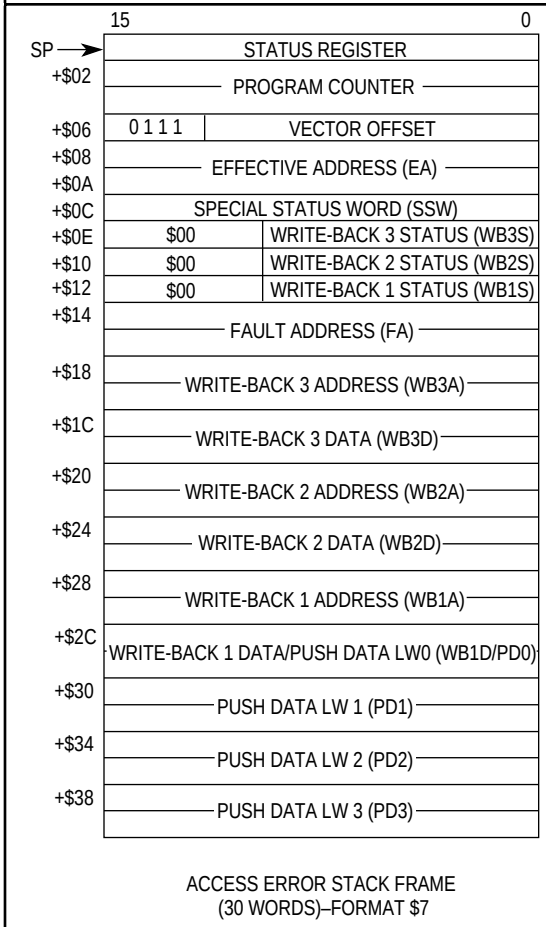
Eight-Word Stack Frame (Format $4)

Stack Frames	Exception Types	Stacked PC Points To
	• Unimplemented floating-point instruction	• Effective address field is the address of the faulted instruction operand.

When the MCF5102 writes or reads a stack frame, it uses long-word operand transfers wherever possible. Using a long-word-aligned stack pointer greatly enhances exception processing performance. The processor does not necessarily read or write the stack frame data in sequential order. The system software should not depend on a particular exception generating a particular stack frame.

8.4.5 Access Error Stack Frame (Format $7)

A 30-word access error stack frame is created for data and instruction access faults other than instruction address errors. In addition to information about the current processor status and the faulted access, the stack frame also contains pending write-backs that the access error exception handler must complete. The following paragraphs describe in detail the format for this frame and how the processor uses it when returning from exception processing.

Stack Frames	Exception Types	Stacked PC Points To
	• Data or Instruction Access Fault (PV Fault or Bus Error)	• Next Instruction

8.4.5.1 Effective Address. The effective address contains address information when one of the continuation flags CM, CT, CU, or CP in the SSW is set.

8.4.5.2 Special Status Word (SSW). The SSW information indicates whether an access to the instruction stream or the data stream (or both) caused the fault and contains status information for the faulted access. Figure 8-7 illustrates the SSW format.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	CT	CM	MA	PV	LK	RW	X	SIZE		TT		TM		

Figure 8-7. Special Status Word Format

CT—Continuation of Trace Exception Pending

CT is set for an access error with a pending trace exception. When RTE is executed with CT set, the MCF5102 will move the words on the stack an offset of $00–$0B from the current SP to offset $30–$3B, adjusting the stack pointer by +$30. The MCF5102

changes the stack frame format to $2 before fetching the trace exception vector and jumping directly to trace exception handling. This stack adjustment creates the stack frame that normally would have been created for the trace exception had the pending access not encountered a bus error.

CM—Continuation of MOVEM Instruction Execution Pending

CM is set if a data access encounters a bus error for a MOVEM. Since the MOVEM operation can write over the memory location or registers used to calculate the effective address, the MCF5102 internally saves the effective address after calculation. When MOVEM encounters a bus error, a stack frame is created with CM set, and the effective address field contains the calculated effective address for the instruction. When RTE is executed, MOVEM restarts using the effective address on the stack (instead of repeating the effective address calculate operation) if the address mode is PC relative (mode = 111, register = 010 or 011) or indirect with index (mode = 110).

MA—Misaligned Access

MA is set if an Privilege violation occurs for second-ACR access that crosses two ACR's in memory.

PV—Privileged Violation Fault

This bit is set. WHEN A WRITE OR SUPERVISOR PRIVILEGE VIOLATION IS REPORTED BY THE ACR. It is cleared for a bus-error instruction, data, or cache line-push access.

LK—Locked Transfer (Read-Modify-Write)

This bit is set if a fault occurred on a locked transfer it is cleared otherwise.

RW—Read/Write

This bit is set if a fault occurred on a read transfer it is cleared otherwise.

X—Undefined

SIZE—Transfer Size

The SIZE field corresponds to the original access size. If a data cache line read results from a read miss and the line read encounters a bus error, the SIZE field in the resulting stack frame indicates the size of the original read generated by the execution unit.

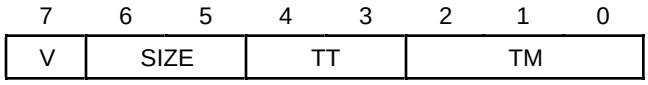
TT—Transfer Type

This field defines the TT1–TT0 signal encodings for the faulted transfer.

TM—Transfer Modifier

This field defines the TM2–TM0 signal encodings for the faulted transfer.

8.4.5.3 Write-Back Status. These fields contain status information for the three possible write-backs that could be pending after the faulted access (see Figure 8-8). For a data cache line-push fault or a MOVE16 write fault, WB1S is zero (invalid).



TM—Transfer Modifier
TT—Transfer Type
SIZE—Transfer Size
V—Valid Write (write-back pending if set)

Figure 8-8. Write-Back Status Format

8.4.5.4 Fault Address. The fault address (FA) is the initial address for the access that faulted. For a misaligned access that faults, the FA field contains the address of the first byte of the transfer, regardless of which of the two or three bus transfers for the misaligned access was faulted.

8.4.5.5 Write-Back Address and Write-Back Data. Write-back addresses (WB3A, WB2A, and WB1A) are memory pointers that indicate where to place the write-back data (WB3D, WB2D, and WB1D). WB3A and WB3D correspond to the temporary holding register in the integer unit (WB3). WB2A and WB2D correspond to the temporary holding register in the data ACU (WB2) prior to address translation. WB1A and WB1D correspond to the temporary holding register in the bus controller (WB1), which determines the external address and data bus bit patterns. Refer to Section 2 Integer Unit for details on the operation of the integer unit pipeline.

The write-back data in WB3D and WB2D is register aligned with byte and word data contained in the least significant byte and word, respectively, of the field. Write-back data in WB1D is memory aligned and resides in the byte positions corresponding to the data bus lanes used in writing each byte to memory. Table 8-5 lists the data alignment for each combination of data format and A1 and A0.

Table 8-5. Write-Back Data Alignment

Data Format	Address		Data Alignment	
	A1	A0	WB1D	WB2D, WB3D
Byte	0	0	31–24	7–0
	0	1	23–16	7–0
	1	0	15–8	7–0
	1	1	7–0	7–0
Word	0	0	31–16	15–0
	0	1	23–8	15–0
	1	0	15–0	15–0
	1	1	7–0, 31–24	15–0
Long Word	0	0	31–0	31–0
	0	1	23–0, 31–24	31–0
	1	0	15–0, 31–16	31–0
	1	1	7–0, 31–8	31–0

NOTE: For a line transfer fault, the four long words of data in PD3–PD0 are already aligned with memory. Bits 31–0 of each field correspond to bits 31–0 of the memory location to be written to, regardless of the value of the address bits A1 and A0 for the write-back address.

8.4.5.6 Push Data. The push data field contains an image of the cache line that needs to be pushed to memory.

8.4.5.7 Access Error Stack Frame Return From Exception. For the access error stack frame (format $7), the processor restores the SR and PC values from the stack and checks the four continuation status bits in the SSW on the stack. If these bits are not set, the processor increments the active supervisor stack pointer by 30 words and resumes normal instruction execution. If the MOVEM continuation bit is set, the processor restores the calculated effective address from the stack frame, increments the active supervisor stack pointer by 30 words, and restarts the MOVEM instruction at a point after the effective address calculation. All operand accesses for the MOVEM that occurred before the faulted access are repeated. If a continuation bit is set for a pending trace, unimplemented floating-point instruction, or floating-point post-instruction exception, the processor restores the calculated effective address from the stack frame, increments the active supervisor stack pointer by 30 words, and immediately begins exception processing for the pending exception. The processor sets only one of the continuation bits when the access error stack frame is created. If the access error exception handler sets multiple bits, operation of the RTE instruction is undefined.

If the frame format field in the stack frame contains an illegal format code, a format exception occurs. If a format error or access fault exception occurs during the frame validation sequence of the RTE instruction, the processor creates a normal four-word or an access error stack frame below the frame that it was attempting to use. The illegal stack frame remains intact, so that the exception handler can examine or repair the illegal frame. In a multiprocessor system, the illegal frame can be left so that, when appropriate, another processor of a different type can use it.

The bus error exception handler can identify bus error exceptions due to instruction faults by examining the TM field in the SSW of the access error stack frame. For user and supervisor instruction faults, the TM field contains $2 and $6, respectively (see Figure 8-7). Since the processor allows all pending accesses to complete before reporting an instruction fault, the stack frame for an instruction fault will not contain any pending write-backs. The PV bit of the SSW is used to distinguish between PV faults and bus errors, and the FA field contains the address of the instruction prefetch.

For an address error fault, the processor saves a format $2 exception stack frame on the stack. This stack frame contains the PC pointing to the instruction that caused the address error as well as the actual address referenced by the instruction. Note that bit 0 of the referenced address is cleared on the stack frame. Address error faults must be repaired in software.

For a fault due to a Privileged violation or bus error, pending write-backs are also saved on the access error stack frame and must be completed by the exception handler. For the faulted access, the fault address in the FA field combined with the transfer attribute information from the SSW can be used to identify the cause of the fault. In identifying the fault, the system programmer should be aware that the ACU considers the read portion of read-modify-write transfers

All accesses other than instruction prefetches go through the data ACU, and the MCF5102 treats the instruction and data address spaces as a single merged address space (the exception is the presence of separate transparent translation registers). The function codes for accesses such as PC relative operand addressing and MOVES transfers to function codes $2 and $6 are converted to data references to go through the ACU, and appear in the TM field of the access error stack frame as data references.

After the fault is corrected, any pending write-backs on the stack frame must be completed. The write-back status fields should be checked for possible write-backs, which the exception handler should complete in the following order: write-back 1, write-back 2, and write-back 3. For a push fault, the push must be completed first, followed by two potential write-backs. Completion of write-back 1 should not generate another access error since this write-back corresponds to the faulted access that has been corrected by the handler. However, write-backs 2 and 3 can cause another bus error exception when the handler attempts to write to memory and should be checked before attempting the write to prevent nesting of exceptions if required by the operating system. The following general bus fault examples indicate the resulting contents of the access error stack frame fields:

1. All Read Access Errors (SSW–RW = $1, TT = $0, TM = $1 or $5)—The FA field contains the address of the fault. The WB1S and WB2S fields are zero, and only WB3S can indicate an additional write-back.
2. Cache Push Bus Error (SSW–RW = $0, TT = $0, TM = $0)—The assertion of $\overline{\text{TEA}}$ causes this error when a cache push bus cycle is in progress. The FA field contains the address of the fault, and the WB1S field is ignored. All four long words of the data for a push are contained in LW3–LW0 regardless of the size of the transfer. The size of the transfer is indicated in the SIZE field of the SSW and can be either a line or long word. If a line is indicated, all four long words need to be pushed out. If a

long word is indicated, all four long words can be written out, or bits 3 and 2 of the FA field can be evaluated to indicate which long words need to be written out to memory ($3, $2, $1, and $0 indicate LW3, LW2, LW1, and LW0, respectively). The WB2S and WB3S fields indicate up to two additional write-backs. If WB2S is valid and if it indicates a MOVE16 instruction, no data should be written out for that write-back slot.

3. Normal Write Bus Error (SSW–RW = $0, TT = $0, TM = $1 or $5)—The assertion of $\overline{\text{TEA}}$ causes this error when a normal write bus cycle is in progress. The FA field contains the address of the fault, and the WB1S field indicates that it is valid. The FA and WB1A are equivalent. The WB2S and WB3S fields indicate up to two additional write-backs.
4. MOVE16 Write Bus Error (SSW–RW = $0, TT = $1)—The assertion of $\overline{\text{TEA}}$ causes this error during the write portion of a MOVE16 instruction. The FA field contains the address of the fault, and the WB1S field indicates that it is valid. All four long words are contained in LW3–LW0 and must be written out before using FA. Software must ensure that address bits 1 and 0 are both clear if regular move instruction are to be used to write out to the destination.
5. ACR Privileged Violation(SSW–RW = $0, WB1S–V = $0)—The FA field contains the address of the faulted instruction, WB1S = 0, and WB2S indicates that it is valid. Only WB3S can indicate an additional write-back. If WB2S indicates a MOVE16 instruction and if the MOVE16 instruction is used to read from a peripheral that cannot tolerate double reads, then software must write the data contained in PD3–PD0 out to memory and increment the stacked PC to take it beyond the MOVE16 instruction that caused the fault. Otherwise, if the MOVE16 instruction is allowed to be restarted, another read from the peripheral would occur. If double reads can be tolerated, simply do no write-backs and allow instruction to restart. This is the only case in which the action to be taken depends on whether or not a double read can be tolerated.

Table 8-6 lists the possible combinations of write-backs and the proper way to handle them. The SSW_RW column indicates a read or write cycle; the SSW_PUSH column indicates whether the fault is for a push (TT = 00 and TM = 000). The WB1S, WB2S, and WB3S columns list the respective field's V-bit and indicate a MOVE16 transfer type (TT = 01). The easy cleanup data written column lists the stack's field to be written out to memory if the user is not concerned with retouching peripherals. The hard cleanup action column lists the action to be taken if the peripherals cannot be retouched by MOVE16 (if different from easy cleanup). Note that if a push access error is reported and the size is long word, all four long words, PD0–PD3, are still valid for the line. The exception handler can either write PD0–PD3 using the fault address with bits 3–0 cleared or write the PD corresponding to bits 3–2 of the address (e.g., address $0000000C corresponds to PD3). Note that a MOVE16 is never reported in the WB3S. The SIZE field of WB3S is never a line.

After the bus error exception handler completes all pending operations and executes an RTE to return, the RTE reads only the stack information from offset $0–$D in the access error stack frame. For a pending trace exception, the RTE adjusts the stack to match the pending exception and immediately begins exception processing, without requiring the exception to reoccur.

Table 8-6. Access Error Stack Frame Combinations

Main Case	SSW_RW	SSW_PUSH	WB1S		WB2S		WB3S	Easy Cleanup Data Written	Hard Cleanup Action
			1V	1M16	2V	2M16	3V		
All Read Access Errors	1a	No	0	X	0	X	0	None	
	1a	No	0	X	0	X	1	WB3D	
	All other read cases are not possible.								
Cache Push Bus Errorc	0	Yes	0	X	0	X	0	PD3–0	
	0	Yes	0	X	0	X	1	PD3–0, WB3D	
	0	Yes	0	X	1	0	0	PD3–0, WB2D	
	0	Yes	0	X	1	0	1	PD3–0, WB2D, WB3D	
	0	Yes	0	X	1	1	0	PD3–0, ~WB2D[d] (Note b)	(Note b)
Normal Write bus Error	0	No	1	0	0	X	0	WB1D	
	0	No	1	0	0	X	1	WB1D, WB3D	
	0	No	1	0	1	0	0	WB1D, WB2D	
	0	No	1	0	1	0	1	WB1D, WB2D, WB3D	
	0	No	1	0	1	1	0	WB1D, ~WB2D[d] (Note b)	(Note b)
MOVE16 Write Bus Error	0	No	1	1	0	X	1	PD3–0, WB3D	
	0	No	1	1	0	X	0	PD3–0	
	0	No	1	1	1	0	0	PD3–0, WB2D	
	0	No	1	1	1	0	1	PD3–0, WB2D, WB3D	
	0	No	1	1	1	1	0	PD3–0, ~WB2D[d] (Note b)	(Note b)
Write Fault	0	No	0	X	1	0	0	WB2D	
	0	No	0	X	1	0	1	WB2D, WB3D	Write PD3–0 and skipe.
	0	No	0	X	1	1	0	~WB2D[d]	
Impossible Write Cases	0	Yes	1	X	X	X	X	(Note f)	—
	0	Don't Care	X	X	X	1	1	(Note g)	

NOTES:

a. The data ACU stage is tied up until the bus controller passes the read back through the data memory unit and to the execution stage in the integer unit. Therefore, no pending write is possible in WB1 or WB2. WB3 could hold a pending write that was deferred due to operand read or was generated after the read.

b. If any kind of access error is reported and if a MOVE16 write is pending in the WB2 stage, then that MOVE16 read must hit in the cache so the MOVE16 can be safely restarted since it has not caused bus cycles that could retouch peripherals.

c. A cache push bus error is normally considered a fatal error.

d. Indicates that the data should not be written even though the V-bit for it is set (WB2 corresponds to a MOVE16 write).

e. The exception handler must alter the stacked PC to point past the MOVE16 and predecrement and postincrement address registers.

f. 1V must be 0 for push exceptions.

g. The execution stage does not post a write until the MOVE16 is in the integer unit.

SECTION 9
INSTRUCTION TIMINGS

This section summarizes instruction timings for the MCF5102. Table 9-1 alphabetically lists instruction timings and their location in this section.

Table 9-1. Instruction Timing Index

Table 9-1. Instruction Timing Index (Continued)

Instruction	Page	Instruction	Page	Instruction	Page
MOVEM <list>,<ea>	9-23	ORI #<xxx>,SR	9-11	RTS	9-11
MOVEM.L <ea>,<list>	9-23	PACK	9-11	SBCD	9-11
MOVEP	9-11	PEA	9-26	Scc	9-27
MOVEQ	9-11	PFLUSH	9-11	SUB	9-13
MOVES <ea>,An	9-24	PFLUSHA	9-11	SUBA	9-27
MOVES <ea>,Dn	9-24	PFLUSHAN	9-11	SUBI	9-13
MOVES Rn,<ea>	9-24	PFLUSHN (An)	9-11	SUBQ	9-14
MULS.W/L	9-25	PTESTR, PTESTW	9-11	SUBX	9-11
MULU.W/L	9-25	RESET	9-11	SWAP	9-11
NBCD	9-25	ROL	9-26	TAS	9-28
NEG	9-26	ROR	9-26	TRAP#	9-11
NEGX	9-26	ROXL	9-27	TRAPcc	9-11
NOP	9-11	ROXR	9-27	TRAPV	9-11
NOT	9-26	RTD	9-11	TST	9-13
OR	9-13	RTE	9-11	UNLK	9-11
ORI	9-13	RTR	9-11	UNPK	9-11
ORI #<xxx>,CCR	9-11				

9.1 OVERVIEW

Refer to **Section 2 Integer Unit** for information on the integer unit pipeline. The <ea> fetch timing is not listed in the following tables because most instructions require one clock in the <ea> fetch stage for each memory access to obtain an operand. An instruction requires one clock to pass through the <ea> fetch stage even if no operand is fetched. Table 9-2 summarizes the number of memory fetches required to access an operand using each addressing mode for long-word aligned accesses. The user must perform his own calculations for <ea> fetch timing for misaligned accesses.

Table 9-2. Number of Memory Accesses

Addressing Mode	Evaluate <ea> And Fetch Operand	Evaluate <ea> And Send To Execution Stage
Dn	0	0
An	0	0
(An)	1	0
(An)+	1	0
–(An)	1	0
(d_{16},An)	1	0
(d_{16},PC)	1	0
(xxx).W, (xxx).L	1	0
#<xxx>	0	0
(d_8,An,Xn)	1	0
(d_8,PC,Xn)	1	0
(BR,Xn)	1	0
(bd,BR,Xn)	1	0
([bd,BR,Xn])	2	1
([bd,BR,Xn],od)	2	1
([bd,BR],Xn)	2	1
([bd,BR],Xn,od)	2	1

In the instruction timing tables, the <ea> calculate column lists the number of clocks required for the instruction to execute in the <ea> calculate stage of the integer unit pipeline. Dual effective address instructions such as ABCD –(Ay),–(Ax) require two calculations in the <ea> calculate stage and two memory fetches. Due to pipelining, the fetch of the first operand occurs in the same clock as the <ea> calculation for the second operand.

The execute column lists the number of clocks required for the instruction to execute in the execute stage of the integer unit pipeline. This number is presented as a lead time and a base time. The lead time is the number of clocks the instruction can stall when entering the execution stage without delaying the instruction execution. If the previous instruction is still executing in the execution stage when the current instruction is ready to move from the <ea> fetch stage, the current instruction stalls until the previous one completes. For

example, if an execution time is listed as 2_L + 1, the lead time is two clocks and the base time is one for a total execution time of three. The instruction can stall for two clocks without delaying the instruction execution time.

The <ea> calculate and execute stages operate in an interlocked manner for all instructions using the brief and full extension word formats. If an instruction using one of these formats is stalled for more than N_L clocks waiting to begin execution in the execute stage, a similar increase in the <ea> calculate time will result. For example, if the execution time listed is 2_L + 1 and the instruction stalls for three clocks, then the <ea> calculate time increases by one clock ($3 – 1 = 2_L$). Write-back times are not listed because they are system dependent and do not affect either <ea> calculate or execute stages of the pipeline.

Not all addressing modes listed in the following tables for an instruction are valid for all variations of the instruction. For example, the table for the integer ADD instruction lists times for both ADD <ea>,Dn and ADD Dn,<ea>. All addressing modes listed are valid for ADD <ea>,Dn. For ADD Dn,<ea> the following invalid modes should be ignored: An, (d_{16},PC), #<xxx>, (d_8,PC,Xn), and modes with BR = PC. Refer to the M68000PM/AD, *M68000 Family Programmer's Reference Manual* for a complete summary of valid instruction and addressing mode combinations. The instruction timings are based on the following suppositions unless otherwise noted:

1. All timings are related to BCLK cycles and are for BR = An or suppressed. For BR = PC, 1 and 1_L clocks to the <ea> calculate and execution times unless otherwise noted. For memory indirect postindexed with suppressed index — ([bd,BR],Xn) or ([bd,BR],Xn,od) with Xn suppressed — times are the same as for memory indirect preindexed with suppressed index — ([bd,BR,Xn]) or ([bd,BR,Xn],od) with Xn suppressed.
2. All memory accesses hit in the caches. It is assumed that external memory has a zero-wait state in this case and that the bus is granted to the MCF5102.

   The result increases access time equal to the number of clocks for the memory access (first bus cycle if the operand access results in a line memory access) if aligned accesses miss in the data cache. As an approximation, this time should be added to the execution time for each operand fetch generated by the instruction.
3. All accesses are aligned to a byte boundary that is a multiple of the operand size. For instance, the timing for all long-word accesses assumes that the operands are on long-word boundaries.

9.2 INSTRUCTION TIMING EXAMPLES

The following examples utilize the instruction timing information given in this section. Figure 9-1 illustrates the integer unit pipeline flow for the simple code sequence listed. The three instructions in the code sequence require only a single clock in each pipeline stage. The TRAPF instructions are also single-clock instructions that function as nonsynchronizing NOPs.

LABEL	INSTRUCTION	<ea> CALCULATE	EXECUTE
P1	TRAPF	1	1
A	MOVE.L $1000,D0	1	1
B	ADDQ.L #1,D0	1	1
C	MOVE.L D0,$1000	1	1
N1	TRAPF	1	1
N2	TRAPF	1	1

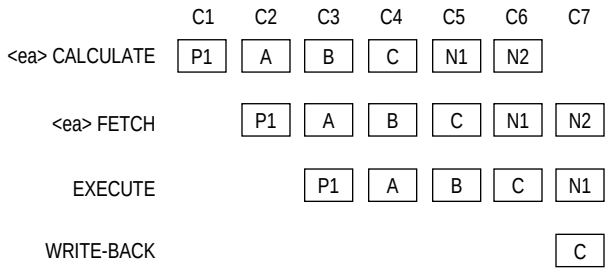


Figure 9-1. Simple Instruction Timing Example

C1 The previous instruction (P1) finishes in the <ea> calculate.

C2 MOVE.L (A) starts in the <ea> calculate and requests an immediate extension word for its effective address.

C3 MOVE.L (A) starts in the <ea> fetch, which fetches the operand at $1000. ADDQ.L (B) starts in the <ea> calculate stage with the operand encoded in the instruction.

C4 MOVE.L (A) executes in the execute stage, storing the fetched operand in register D0. ADDQ.L (B) starts in the <ea> fetch with no operation performed. MOVE.L (C) starts in the <ea> calculate requesting an immediate extension word for its effective address.

C5 ADDQ.L (B) executes in the execution stage, incrementing D0 by 1. MOVE.L (C) passes through the <ea> fetch with no operation performed. The next instruction starts in the <ea> calculate stage.

C6 MOVE.L (C) executes in the execution stage generating a write of D0 to the effective address.

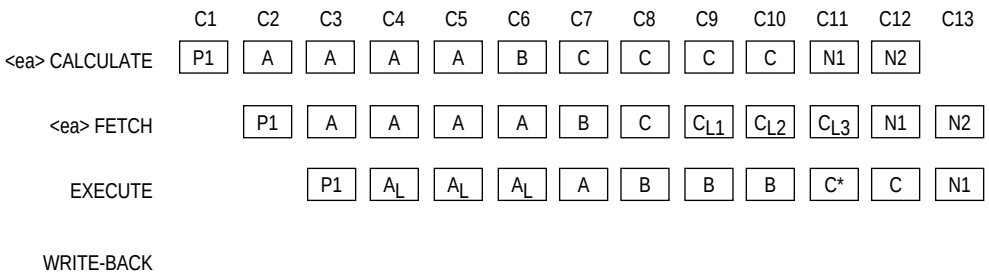
C7 The write to memory by MOVE.L (C) occurs to the data memory unit if it is not busy. If the second TRAPF instruction (N2) in the <ea> fetch stage requires an operand fetch, the write-back for MOVE.L (C) stalls in the write-back stage since it is a lower priority.

The separation of calculation and execution in the <ea> calculate and execute stages allows instruction reordering during compile time to take advantage of potential instruction overlap. Figure 9-2 illustrates this overlap for an instruction requiring multiple clocks in the execute stage and with an instruction with a long lead time. The execution time for LEA (3_L + 1) indicates that the instruction can be stalled three clocks without affecting execution.

When the LEA (A) instruction precedes the ABCD (B) instruction, the execution stalls during C4–C6 (equivalent to the LEA lead time) while the instruction completes in the <ea> calculate and <ea> fetch stages. The resulting execution time for the LEA (A) and ABCD (B) sequence is eight clocks.

However, if the LEA (C) instruction follows the ABCD (B) instruction, the LEA stalls in the <ea> fetch instead, during C9–C11. The LEA then executes in a single clock in the execution stage. The resulting execution time for the LEA (C) and ABCD (B) sequence is five clocks.

LABEL	INSTRUCTION		<ea> CALCULATE	EXECUTE
P1	TRAPF		1	1
A	LEA	$24(PC),A1	4	3_L + 1
B	ABCD	D0,D1	1	3
C	LEA	$24(PC),A1	4	3_L + 1
N1	TRAPF		1	1
N2	TRAPF		1	1

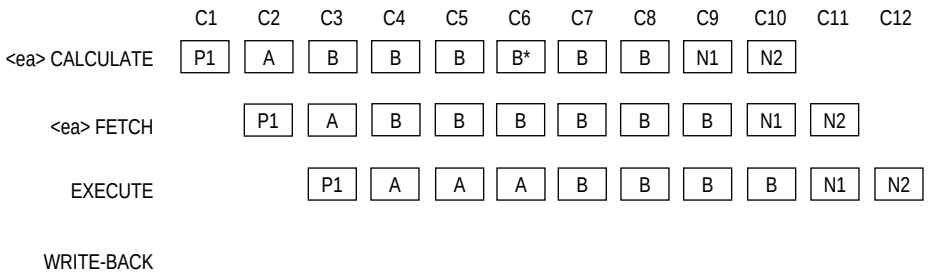


NOTE: *Possible stalls in this stage.

Figure 9-2. Instruction Overlap with Multiple Clocks

Instructions using the brief and full extension word format addressing modes cause the <ea> calculate and execute stages to operate in an interlocked manner. When these instructions wait to begin execution in the execution stage, there is a similar increase in the <ea> calculate time. Figure 9-3 illustrates this effect for an ADD instruction using a brief format extension word. The ADD instruction stalls for two clocks waiting to enter the execution stage. Since this time exceeds by one clock the ADD lead time, the ADD instruction remains in the <ea> calculate stage for one additional clock. If the ADD instruction was in the execution stage for two clocks, the ABCD instruction would not have stalled in the <ea> calculate stage.

LABEL	INSTRUCTION		<ea> CALCULATE	EXECUTE
P1	TRAPF		1	1
A	ABCD	D0,D1	3	3
B	ADD.L	4(A0,D3),D2	5	$1_L + 4$
N1	TRAPF		1	1
N2	TRAPF		1	1



NOTE: *Possible stalls in this stage.

Figure 9-3. Interlocked Stages

9.3 CINV AND CPUSH INSTRUCTION TIMING

The following details the execution time for the CINV and CPUSH instructions used to perform maintenance of the instruction and data caches. These two instructions sample interrupt request ($\overline{IPL}$x) signals on every clock instead of at instruction boundaries. While performing the actual cache invalidate operation, the execution unit stalls to allow previous write-backs and any pending instruction prefetches to complete. The total time required to execute a cache invalidate instruction is dependent on the previous instruction stream. Execution time for this instruction is independent of the selected cache combination. The CINV instructions interlock operation of the <ea> calculate and execution stages to prevent a previous instruction from accessing the caches until the invalidate operation is complete. Idle refers to the number of clocks required for all pending writes and instruction prefetches to complete. Table 9-3 list the CINV timings.

Table 9-3. CINV Timing

Instruction	Execution Time
CINVL	9 + Idle
CINVP	266 + Idle
CINVA	9 + Idle

Execution time for the CPUSH instruction is dependent on several factors, such as the number of dirty cache lines and the size of the resulting push (either long-word or line); the overlapping operations within the data cache and the bus controller; the distribution of dirty cache lines; and the number of wait states in the push access on the bus. The interaction of these factors determines the total time required to execute a CPUSH instruction.

Since the distribution of dirty data within the cache is entirely dependent on the nature of the user's code, it is impossible to provide an equation for execution time that works for all code sequences. Table 9-4 provides baseline information indicating best and worst case execution times for the three CPUSH instruction variants. Best case corresponds to a cache containing no dirty entries, while the worst case corresponds to all lines dirty and requiring line pushes. In Table 9-4, line refers to the number of clocks required in the user's system for a line transfer. Idle refers to the number of clocks required for all pending writes and instruction prefetches to complete.

Table 9-4. CPUSH Best and Worst Case Timing

Instruction	Execution Time	
	Best Case	Worst Case
CPUSHL	6	6 + Line + Idle
CPUSHP CPUSHA	267	11 + 256 ¥ Line + Idle

9.4 MOVE INSTRUCTION TIMING

SOURCE	DESTINATION					
	Dn		(An)		(An)+	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	1	1	1	1
(An)	1	1	1	1	2	1_L + 1
(An)+	1	1	2	1_L + 1	2	1_L + 1
–(An)	1	1	2	1_L + 1	2	1_L + 1
(d_{16},An)	1	1	2	1_L + 1	2	1_L + 1
(d_{16},PC)	3	2_L + 1	3	2_L + 1	3	2_L + 1
(xxx).W, (xxx).L	1	1	1	1	2	1_L + 1
#<xxx>	1	1	1	1	2	1_L + 1
(d_8,An,Xn)	3	3	4	4	5	5
(d_8,PC,Xn)	5	1_L + 4	5	1_L + 4	6	1_L + 5
(b_{16},BR,Xn)	7	1_L + 6	7	1_L + 6	8	1_L + 7
([bd,BR,Xn])	10	1_L + 9	10	1_L + 9	11	1_L + 10
([bd,BR,Xn],od)	11	1_L + 10	11	1_L + 10	12	1_L + 11
([bd,BR],Xn)	11	3_L + 8	11	3_L + 8	12	3_L + 9
([bd,BR],Xn,od)	12	3_L + 9	12	3_L + 9	13	3_L + 10
	–(An)		(d_{16},An)		(xxx).W, (xxx).L	
Dn	1	1	1	1	1	1
(An)	2	1_L + 1	2	1_L + 1	1	1
(An)+	2	1_L + 1	2	1_L + 1	2	1_L + 1
–(An)	2	1_L + 1	2	1_L + 1	2	1_L + 1
(d_{16},An)	2	1_L + 1	2	1_L + 1	2	1_L + 1
(d_{16},PC)	3	2_L + 1	4	3_L + 1	4	3_L + 1
(xxx).W, (xxx).L	2	1_L + 1	2	1_L + 1	2	1_L + 1
#<xxx>	2	1_L + 1	2	1_L + 1	2	1_L + 1
(d_8,An,Xn)	5	5	5	5	5	5
(d_8,PC,Xn)	6	1_L + 5	6	1_L + 5	6	1_L + 5
(b_{16},BR,Xn)	8	1_L + 7	8	1_L + 7	8	1_L + 7
([bd,BR,Xn])	11	1_L + 10	11	1_L + 10	11	1_L + 10
([bd,BR,Xn],od)	12	1_L + 11	12	1_L + 11	12	1_L + 11
([bd,BR],Xn)	12	3_L + 9	12	3_L + 9	12	3_L + 9
([bd,BR],Xn,od)	13	3_L + 10	13	3_L + 10	13	3_L + 10

9.4 MOVE INSTRUCTION TIMING (Continued)

SOURCE	DESTINATION					
	(d_8,An,Xn)		(b_{16},An,Xn)		([bd,An,Xn])	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	3	3	7	1_L + 6	10	1_L + 9
(An)	4	4	7	1_L + 6	10	1_L + 9
(An)+	4	4	7	1_L + 6	10	1_L + 9
–(An)	4	4	7	1_L + 6	10	1_L + 9
(d_{16},An)	4	4	7	1_L+ 6	10	1_L + 9
(d_{16},PC)	8	4_L + 4	10	4_L + 6	13	4_L + 9
(xxx).W, (xxx).L	4	4	7	1_L + 6	10	1_L + 9
#<xxx>	3	3	7	1_L + 6	10	1_L + 9
(d_8,An,Xn)	8	8	10	10	13	13
(d_8,PC,Xn)	9	1_L + 8	11	1_L + 10	14	1_L + 13
(b_{16},BR,Xn)	11	1_L + 10	13	1_L + 12	16	1_L + 15
([bd,BR,Xn])	14	1_L + 13	16	1_L + 15	19	1_L + 18
([bd,BR,Xn],od)	15	1_L + 14	17	1_L + 16	20	1_L + 19
([bd,BR],Xn)	15	3_L + 12	17	3_L + 14	20	3_L + 17
([bd,BR],Xn,od)	16	3_L + 13	18	3_L + 15	21	3_L + 18
	([bd,An,Xn],od)		([bd,An],Xn)		([bd,An],Xn,od)	
Dn	11	1_L + 10	11	3_L + 8	12	3_L + 9
(An)	11	1_L + 10	11	3_L + 8	12	3_L + 9
(An)+	11	1_L + 10	11	3_L + 8	12	3_L + 9
–(An)	11	1_L + 10	11	3_L + 8	12	3_L + 9
(d_{16},An)	11	1_L + 10	11	3_L + 8	12	3_L + 9
(d_{16},PC)	14	4_L + 10	14	6_L + 8	15	6_L + 9
(xxx).W, (xxx).L	11	1_L + 10	11	3_L + 8	12	3_L + 9
#<xxx>	11	1_L + 10	11	3_L + 8	12	3_L + 9
(d_8,An,Xn)	14	14	14	14	15	15
(d_8,PC,Xn)	15	1_L + 14	15	1_L + 14	16	1_L + 15
(b_{16},BR,Xn)	17	1_L + 16	17	1_L + 16	18	1_L + 17
([bd,BR,Xn])	20	1_L + 19	20	1_L + 19	21	1_L + 20
([bd,BR,Xn],od)	21	1_L + 20	21	1_L + 20	22	1_L + 21
([bd,BR],Xn)	21	3_L + 18	21	3_L + 18	22	3_L + 19
([bd,BR],Xn,od)	22	3_L + 19	22	3_L + 19	23	3_L + 20

9.5 MISCELLANEOUS INTEGER UNIT INSTRUCTION TIMINGS

Instruction	Condition	<ea> Calculate	Execute
ABCD	Dy,Dx –(Ay),–(Ax)	1 3	3 1_L + 3
ADDX	Dy,Dx –(Ay),–(Ax)	1 3	1 1_L + 2
ANDI #<xxx>,CCR	—	1	4
ANDI #<xxx>,SR[a]	—	9	1_L + 8
Bcc	Branch Taken Branch Not Taken	2 3	2 3
BRA	Branch Taken Branch Not Taken	2 3	2 3
BSR <offset>	—	2	1_L + 1
CAS2[b]	True False	56 51	6_L + 49 6_L + 44
CMPM	—	3	1_L + 2
DBcc[c]	False, Count > –1 False, Count = –1 True	3 4 4	3 4 4
EORI #<xxx>,CCR	—	1	4
EORI #<xxx>,SR[a]	—	9	1_L + 8
EXG	Dy,Dx Ay,Ax Dy,Ax	1 2 1	1 1_L + 1 1
EXT	Word Long Word	1 1	2 1
EXTB	Long Word	1	1
ILLEGAL[a]	A-Line Unimplemented F-Line Unimplemented	16 16	16 16
LINK	—	3	2_L + 1
MOVE USP	USP,An An,USP[a]	3 7	2_L + 1 1_L + 6
MOVE16[c,d]	(Ax)+,(Ay)+ xxx.L,(An) xxx.L,(An)+ (An),xxx.L (An)+,xxx.L	6 4 5 4 4	1_L+ 7 7 8 7 7
MOVEC[b]	Rn,Rc Rc,Rn	7 11	1_L + 6 1_L + 10
MOVEP[c]	MOVEP.W Dn,d16(An) MOVEP.L Dn,d16(An) MOVEP.W d16(An),Dn MOVEP.L d16(An),Dn	11 13 4 8	2_L + 9 2_L + 11 2_L + 5 2_L + 8
MOVEQ	—	1	1
NOP[a]	—	8	1_L + 7

9.5 MISCELLANEOUS INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Instruction	Condition	<ea> Calculate	Execute
ORI #<xxx>,CCR	—	1	4
ORI #<xxx>,SR[a]	—	9	1_L + 8
PACK	Dx,Dy,#<xxx> –(Ay),–(Ax),#<xxx>	1 3	3 2_L + 3
PFLUSH[b]	—	11	1_L + 10
PFLUSHA[b]	—	11	1_L + 10
PFLUSHAN[b]	—	27	1_L + 26
PFLUSHN (An)[b]	—	11	1_L + 10
PTESTR, PTESTW[e]	—	25	11_L + 14
RESET[a]	—	521	521
RTD[c]	—	6	1_L + 5
RTE[a]	Stack Format $0 Stack Format $1 Stack Format $2 Stack Format $3 Stack Format $4 Stack Format $7	2 4 2 3 2 4	13 23 14 20 15 23
RTR[c]	—	7	1_L + 6
RTS[c]	—	5	5
SBCD	Dy,Dx –(Ay),–(Ax)	1 3	3 1_L + 3
SUBX	Dy,Dx –(Ay),–(Ax)	1 3	1 1_L + 2
SWAP	—	1	2
TRAP#[a]	—	16	16
TRAPcc[f]	Taken Not Taken	19 5	19 5
TRAPV[f]	Taken Not Taken	19 5	19 5
UNLK	—	2	1_L + 1
UNPK	Dx,Dy,# –(Ay),–(Ax),#	1 3	4 2_L + 4

NOTES:

a. Times listed are minimum. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.
b. Times listed are typical. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.
c. This instruction interlocks the <ea> calculate and execute stages.
d. Successive in-line MOVE16 instructions each add eight clocks to the <ea> calculate and execute times.
e. Typical measurement for three-level table search with no descriptor writes, no entries cached, and four-clock memory access times.
f. This instruction interlocks the <ea> calculate and execute stages. For the exception taken, this instruction also synchronizes some portions of the processor before execution; times listed are minimum in this case.

9.6 INTEGER UNIT INSTRUCTION TIMINGS

Addressing Mode	ADD, AND, EOR, OR, SUB, TST		ADDA		ADDI, ANDI, EORI, ORI, SUBI	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	1	2	1	1
An	1	1	1	1	—	—
(An)	1	1	1	2	1	1
(An)+	1	1	2	1_L + 2	2	1_L + 1
–(An)	1	1	2	1_L + 2	2	1_L + 1
(d_{16},An)	1	1	2	1_L + 2	2	1_L + 1
(d_{16},PC)	3	2_L + 1	3	2_L + 2	—	—
(xxx).W, (xxx).L	1	1	1	2	2	1_L + 1
#<xxx>	1	1	1	1	—	—
(d_8,An,Xn)	3	3	4	5	3	3
(d_8,PC,Xn)	5	1_L + 4	5	1_L + 5	—	—
(BR,Xn)	6	1_L + 5	6	1_L + 6	7	1_L + 6
(bd,BR,Xn)	7	1_L + 6	7	1_L + 7	8	1_L + 7
([bd,BR,Xn])	10	1_L + 9	10	1_L + 10	10	1_L + 10
([bd,BR,Xn],od)	11	1_L + 11	11	1_L + 12	11	1_L + 11
([bd,BR],Xn)	11	3_L + 8	11	3_L + 9	11	3_L + 9
([bd,BR],Xn,od)	12	3_L + 10	12	3_L + 11	12	3_L + 10

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	ADDQ, SUBQ		ASL		ASR, LSL, LSR	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	1	3/4*	1	2/3*
An	1	1	—	—	—	—
(An)	1	1	1	3	1	2
(An)+	2	1_L + 1	1	3	1	2
–(An)	2	1_L + 1	1	3	1	2
(d_{16},An)	2	1_L + 1	1	3	1	2
(d_{16},PC)	—	—	—	—	—	—
(xxx).W, (xxx).L	1	1	1	3	1	2
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	3	3	3	5	3	4
(d_8,PC,Xn)	—	—	—	—	—	—
(BR,Xn)	7	1_L + 6	7	1_L + 8	7	1_L + 7
(bd,BR,Xn)	8	1_L + 7	8	1_L + 9	8	1_L + 8
([bd,BR,Xn])	10	1_L + 9	10	1_L + 11	10	1_L + 10
([bd,BR,Xn],od)	11	1_L + 11	11	1_L + 12	11	1_L + 11
([bd,BR],Xn)	11	3_L + 8	11	3_L + 10	11	3_L + 9
([bd,BR],Xn,od)	12	3_L + 10	12	3_L + 11	12	3_L + 10

*Immediate count specified for shift count/shift count specified in register, respectively.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	BCHG, BCLR, BSET[a]		BFCHG, BFCLR, BFSET[b,c]		BFEXTS, BFEXTU[b,d]	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	3/4	3/4[e]	6/7[e]	1/2[e]	4/5[e]
An	—	—	—	—	—	—
(An)	1	3/4	9	2_L + 8	9	2_L + 7
(An)+	1	3/4	—	—	—	—
–(An)	1	3/4	—	—	—	—
(d_{16},An)	2/1	1_L + 3/4	9	2_L + 8	9	2_L + 7
(d_{16},PC)	—	—	—	—	10	3_L + 7
(xxx).W, (xxx).L	2/1	1_L + 3/4	9	2_L + 8	9	2_L + 7
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	3	5/6	10	11	10	10
(d_8,PC,Xn)	—	—	—	—	11	1_L + 10
(BR,Xn)	7	1_L + 8/1_L + 9	13	1_L + 13	13	1_L + 12
(bd,BR,Xn)	8	1_L + 9/1_L + 10	14	1_L + 14	14	1_L + 13
([bd,BR,Xn])	10	1_L + 11/1_L + 12	16	1_L + 16	16	1_L + 15
([bd,BR,Xn],od)	11	1_L + 12/1_L + 13	17	1_L + 17	17	1_L + 16
([bd,BR],Xn)	11	3_L + 10/3_L + 11	17	3_L + 15	17	3_L + 14
([bd,BR],Xn,od)	12	3_L + 11/3_L + 12	18	3_L + 16	18	3_L + 15

NOTES:
a. Bit instruction <ea> calculate and execute times T1/T2 apply to #<xxx>/Dn bit numbers.
b. This instruction interlocks the <ea> calculate and execute stages.
c. If the bit field spans a long-word boundary, add ten and nine clocks to the <ea> calculate and execute times, respectively. Two memory addresses are accessed in this case.
d. If the bit field spans a long-word boundary, add two clocks to the execute time. Two memory addresses are accessed in this case.
e. Immediate count specified for both width and offset and width and/or offset specified in register, respectively.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	BFFFO[a,b]		BFINS[a,c]		BFTST[a]	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	3/4[d]	6/7[d]	2/3[d]	5/6[d]	1/2[d]	3/4[d]
An	—	—	—	—	—	—
(An)	9	2_L + 9	9	2_L + 7	9	2_L + 7
(An)+	—	—	—	—	—	—
–(An)	—	—	—	—	—	—
(d_{16},An)	9	2_L + 9	9	2_L + 7	9	2_L + 7
(d_{16},PC)	10	3_L + 9	—	—	10	3_L + 7
(xxx).W, (xxx).L	9	2_L + 9	9	2_L + 7	9	2_L + 7
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	10	12	10	10	10	10
(d_8,PC,Xn)	11	1_L + 12	—	—	11	1_L + 10
(BR,Xn)	13	1_L + 14	13	1_L + 12	13	1_L + 12
(bd,BR,Xn)	14	1_L + 15	14	1_L + 13	14	1_L + 13
([bd,BR,Xn])	16	1_L + 17	16	1_L + 15	16	1_L + 15
([bd,BR,Xn],od)	17	1_L + 18	17	1_L + 16	17	1_L + 16
([bd,BR],Xn)	17	3_L + 16	17	3_L + 14	17	3_L + 14
([bd,BR],Xn,od)	18	3_L + 17	18	3_L + 15	18	3_L + 15

NOTES:

a. This instruction interlocks the <ea> calculate and execute stages.
b. If the bit field spans a long-word boundary, add two clocks to the execute time. Two memory addresses are accessed in this case.
c. If the bit field spans a long-word boundary, add seven clocks to both the <ea> calculate and execute times. Two memory addresses are accessed in this case.
d. If the bit field spans a long-word boundary, add ten and nine clocks to both the <ea> calculate and execute times, respectively. Two memory addresses are accessed in this case.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	BTST		CAS[b]		CHK[c,d] (<ea>, Dn)	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1/2[a]	—	—	8	1_L + 7
An	—	—	—	—	—	—
(An)	1	1/2	36	6_L + 31	9	2_L + 7
(An)+	1	1/2	37	5_L + 31	9	2_L + 7
–(An)	1	1/2	37	5_L + 31	9	2_L + 7
(d_{16},An)	2/1	1_L + 1/2	37	5_L + 31	9	2_L + 7
(d_{16},PC)	3	2_L + 1/2_L + 2	—	—	10	3_L + 7
(xxx).W, (xxx).L	2/1	1_L + 1/2	36	5_L + 31	9	2_L + 7
#<xxx>	—	—	—	—	8	1_L + 7
(d_8,An,Xn)	3	3/4	36	36	10	10
(d_8,PC,Xn)	5	1_L + 4/1_L + 5	—	—	11	1_L + 10
(BR,Xn)	7/6	1_L + 6/1_L + 7	36	1_L + 35	12	1_L + 11
(bd,BR,Xn)	8/7	1_L + 7/1_L + 8	37	1_L + 36	13	1_L + 12
([bd,BR,Xn])	10/9	1_L + 9/1_L + 10	42	40	16	1_L + 15
([bd,BR,Xn],od)	11/10	1_L + 10/1_L + 11	42	1_L + 41	17	1_L + 16
([bd,BR],Xn)	11/10	3_L + 8/3_L + 9	42	3_L + 38	17	3_L + 14
([bd,BR],Xn,od)	12/11	3_L + 9/3_L + 10	42	3_L + 39	18	3_L + 15

NOTES:
a. Bit instruction <ea> calculate and execute times T1/T2 apply to #<xxx>/Dn bit numbers.
b. Times listed are typical. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.
c. This instruction interlocks the <ea> calculate and execute stages.
d. Times listed are for Dn within bounds. This instruction interlocks the <ea> calculate and execute stages.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	CHK2* (<ea>, Rn)		CLR		CMP	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	—	—	1	1	1	1
An	—	—	—	—	1	1
(An)	11	2_L + 9	1	1	1	1
(An)+	—	—	1	1	1	1
–(An)	—	—	1	1	1	1
(d_{16},An)	11	2_L + 9	1	1	1	1
(d_{16},PC)	12	3_L + 9	—	—	3	2_L + 1
(xxx).W, (xxx).L	11	2_L + 9	1	1	1	1
#<xxx>	—	—	—	—	1	1
(d_8,An,Xn)	13	1_L + 12	3	3	3	3
(d_8,PC,Xn)	14	2_L + 12	—	—	5	1_L + 4
(BR,Xn)	15	2_L + 13	6	1_L + 5	6	1_L + 5
(bd,BR,Xn)	16	2_L + 14	7	1_L + 6	7	1_L + 6
([bd,BR,Xn])	19	2_L + 17	9	1_L + 8	9	1_L + 8
([bd,BR,Xn],od)	20	2_L + 18	10	1_L + 9	10	1_L + 9
([bd,BR],Xn)	20	4_L + 16	10	3_L + 7	10	3_L + 7
([bd,BR],Xn,od)	21	4_L + 17	11	3_L + 8	11	3_L + 8

*This instruction interlocks the <ea> calculate and execute stages. Timing for Dn within bounds, UB > LB. For UB < LB, add three clocks to <ea> calculate and execute times. For Rn = An, add one clock to <ea> calculate and execute times.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	CMPA.L		CMPI		CMP2*	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	1	1	—	—
An	1	1	—	—	—	—
(An)	1	1	1	1	13	2_L + 11
(An)+	2	1_L + 1	2	1_L + 1	0	0
–(An)	2	1_L + 1	2	1_L + 1	0	0
(d_{16},An)	2	1_L + 1	2	1_L + 1	13	2_L + 11
(d_{16},PC)	3	2_L + 1	3	2_L + 1	14	3_L + 11
(xxx).W, (xxx).L	1	1	2	1_L + 1	13	2_L + 11
#<xxx>	1	1	—	—	—	—
(d_8,An,Xn)	3	3	3	3	15	1_L + 14
(d_8,PC,Xn)	5	1_L + 4	5	2_L + 4	16	2_L + 14
(BR,Xn)	6	1_L + 5	6	2_L + 5	17	2_L + 15
(bd,BR,Xn)	7	1_L + 6	7	2_L + 6	18	2_L + 16
([bd,BR,Xn])	9	1_L + 8	9	2_L + 8	21	2_L + 19
([bd,BR,Xn],od)	10	1_L + 9	10	2_L + 9	22	2_L + 20
([bd,BR],Xn)	10	3_L + 7	10	4_L + 7	22	4_L + 18
([bd,BR],Xn,od)	11	3_L + 8	11	4_L + 8	23	4_L + 19

*Times listed are typical.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	DIVS.W, DIVU.W*		DIVS.L, DIVU.L, DIVSL.L, DIVUL.L*		JMP	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	8	27	9	44	—	—
An	—	—	—	—	—	—
(An)	8	27	9	44	3	2_L + 1
(An)+	8	27	9	44	—	—
–(An)	8	27	9	44	—	—
(d_{16},An)	8	27	11	2_L + 44	4	3_L + 1
(d_{16},PC)	11	3_L + 27	12	3_L + 44	6	5_L + 1
(xxx).W, (xxx).L	8	27	11	2_L + 44	3	2_L + 1
#<xxx>	8	27	10	1_L + 44	—	—
(d_8,An,Xn)	11	30	12	47	6	6
(d_8,PC,Xn)	12	1_L + 30	13	1_L + 47	7	1_L + 6
(BR,Xn)	13	1_L + 31	14	1_L + 48	8	1_L + 7
(bd,BR,Xn)	14	1_L + 32	15	1_L + 49	9	1_L + 8
([bd,BR,Xn])	17	1_L + 35	18	1_L + 52	12	1_L + 11
([bd,BR,Xn],od)	18	1_L + 36	19	1_L + 53	12	1_L + 11
([bd,BR],Xn)	18	3_L + 34	19	3_L + 51	13	3_L + 10
([bd,BR],Xn,od)	19	3_L + 35	20	3_L + 52	14	3_L + 11

*This instruction interlocks the <ea> calculate and execute stages. Execution time for a DIV/0 exception taken and exception processing is approximately 16 + <ea> calculate clocks. For example, DIV.W #0,Dn takes approximately 24 clocks in both the <ea> calculate and execute times to execute the divide instruction, perform exception stacking, fetch the exception vector, and prefetch the next instruction.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	JSR		LEA		MOVE from CCR	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	—	—	—	—	1	2
An	—	—	—	—	—	—
(An)	3	2_L + 1	1	1	1	2
(An)+	—	—	—	—	1	2
–(An)	—	—	—	—	1	2
(d_{16},An)	4	3_L + 1	2	1_L + 1	1	2
(d_{16},PC)	6	5_L + 1	4	3_L + 1	—	—
(xxx).W, (xxx).L	3	2_L + 1	1	1	1	2
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	6	6	4	4	3	4
(d_8,PC,Xn)	7	1_L + 6	5	1_L + 4	—	—
(BR,Xn)	8	1_L + 7	6	1_L + 5	6	1_L + 6
(bd,BR,Xn)	9	1_L + 8	7	1_L + 6	7	1_L + 7
([bd,BR,Xn])	12	1_L + 11	9	1_L + 8	10	1_L + 10
([bd,BR,Xn],od)	13	1_L + 12	10	1_L + 9	11	1_L + 11
([bd,BR],Xn)	13	3_L + 10	10	3_L + 7	11	3_L + 9
([bd,BR],Xn,od)	14	3_L + 11	11	3_L + 8	12	3_L + 10

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	MOVE to CCR		MOVE from SR[a]		MOVE to SR[b]	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	2	2	1_L + 2	9	1_L + 8
An	—	—	—	—	—	—
(An)	1	2	2	1_L + 2	10	2_L + 8
(An)+	1	2	2	1_L + 2	10	2_L + 8
–(An)	1	2	2	1_L + 2	10	2_L + 8
(d_{16},An)	1	2	2	1_L + 2	10	2_L + 8
(d_{16},PC)	3	2_L + 2	—	—	11	3_L + 8
(xxx).W, (xxx).L	1	2	2	1_L + 2	10	2_L + 8
#<xxx>	1	2	—	—	9	1_L + 8
(d_8,An,Xn)	3	4	4	5	11	11
(d_8,PC,Xn)	4	1_L + 4	—	—	12	1_L + 11
(BR,Xn)	6	1_L + 6	6	1_L + 6	—	—
(bd,BR,Xn)	7	1_L + 7	7	1_L + 7	14	1_L + 13
([bd,BR,Xn])	10	1_L + 10	10	1_L + 10	17	1_L + 16
([bd,BR,Xn],od)	11	1_L + 11	11	1_L + 11	18	1_L + 17
([bd,BR],Xn)	11	3_L + 9	11	3_L + 9	18	3_L + 15
([bd,BR],Xn,od)	12	3_L + 10	12	3_L + 10	19	3_L + 16

NOTES:
a. This instruction interlocks the <ea> calculate and execute stages.
b. Times listed are minimum. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	MOVEA.L[a]		MOVEM <list>,<ea>[b,c]		MOVEM.L <ea>,<list>[b,c]	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	—	—	—	—
An	1	1	—	—	—	—
(An)	1	1	2 + D' + A'	1_L + 1 + D' + A'	3 + D' + A	1_L + 2 + D' + A'
(An)+	1	1	—	—	3 + D' + A	1_L + 2 + D' + A'
–(An)	1	1	2 + D' + A'	1_L + 1 + D' + A'	—	—
(d_{16},An)	1	1	2 + D' + A'	1_L + 1 + D' + A'	3 + D' + A	1_L + 2 + D' + A'
(d_{16},PC)	3	2_L + 1	—	—	4 + D' + A	2_L + 2 + D' + A'
(xxx).W, (xxx).L	1	1	2 + D' + A'	1_L + 1 + D' + A'	3 + D' + A	1_L + 2 + D' + A'
#<xxx>	1	1	—	—	—	—
(d_8,An,Xn)	4	4	9 + D' + A'	2_L + 7 + D' + A'	10 + D' + A	2_L + 8 + D' + A'
(d_8,PC,Xn)	5	1_L + 4	—	—	11 + D' + A	3_L + 8 + D' + A'
(BR,Xn)	6	1_L + 5	11 + D' + A'	3_L + 8 + D' + A'	12 + D' + A	3_L + 9 + D' + A'
(bd,BR,Xn)	7	1_L + 6	12 + D' + A'	3_L + 9 + D' + A'	13 + D' + A	3_L + 10 + D' + A'
([bd,BR,Xn])	10	1_L + 9	15 + D' + A'	3_L + 12 + D' + A'	16 + D' + A	3_L + 13 + D' + A'
([bd,BR,Xn],od)	11	1_L + 10	16 + D' + A'	3_L + 13 + D' + A'	17 + D' + A	3_L + 14 + D' + A'
([bd,BR],Xn)	11	3_L + 8	16 + D' + A'	5_L + 11 + D' + A'	17 + D' + A	5_L + 12 + D' + A'
([bd,BR],Xn,od)	12	3_L + 9	17 + D' + A'	5_L + 12 + D' + A'	18 + D' + A	5_L + 13 + D' + A'

NOTES:

a. Except for Dn and #<xxx> cases, add one clock to execute times for MOEA.W.
b. This instruction interlocks the <ea> calculate and execute stages.
c. D' and A' indicate the number of data and address registers, respectively (if no data registers specified the number one). For MOVEM.W <ea>,<list>, add N – 2 and N clocks to <ea> calculate and execute times, respectively, for N address registers specified.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	MOVES <ea>,An*		MOVES <ea>,Dn*		MOVES Rn,<ea>*	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	—	—	—	—	—	—
An	—	—	—	—	—	—
(An)	28	4_L + 24	20	4_L + 19	13	4_L + 9
(An)+	28	4_L + 24	20	4_L + 19	13	4_L + 9
–(An)	17	2_L + 15	11	12	11	2_L + 9
(d_{16},An)	29	4_L + 24	21	4_L + 19	14	4_L + 9
(d_{16},PC)	—	—	—	—	—	—
(xxx).W, (xxx).L	17	2_L + 15	11	4_L + 10	11	2_L + 9
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	29	1_L + 27	21	1_L + 22	14	1_L + 12
(d_8,PC,Xn)	—	—	—	—	—	—
(BR,Xn)	21	2_L + 19	15	2_L + 14	15	2_L + 13
(bd,BR,Xn)	22	2_L + 20	16	2_L + 15	16	2_L + 14
([bd,BR,Xn])	35	2_L + 32	26	2_L + 27	21	2_L + 17
([bd,BR,Xn],od)	31	2_L + 29	23	2_L + 24	20	2_L + 18
([bd,BR],Xn)	36	4_L + 31	27	4_L + 26	21	4_L + 16
([bd,BR],Xn,od)	32	4_L + 28	24	4_L + 23	21	4_L + 17

*Times listed are typical. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	MULS.W/L*		MULU.W/L*		NBCD	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	16/20	1	14/20	1	3
An	—	—	—	—	—	—
(An)	1	16/20	1	14/20	1	2
(An)+	1	16/20	1	14/20	1	2
–(An)	1	16/20	1	14/20	1	2
(d_{16},An)	1/2	16/20	1/2	14/20	1	2
(d_{16},PC)	3	2_L + 16/2_L + 20	3	14/20	—	—
(xxx).W, (xxx).L	1/2	16/20	1/2	14/20	1	2
#<xxx>	1	16/20	1	14/20	—	—
(d_8,An,Xn)	3	18/22	3	16/22	3	4
(d_8,PC,Xn)	5	1_L + 19/1_L + 23	5	1_L + 17/1_L + 23	—	—
(BR,Xn)	6	1_L + 20/1_L + 24	6	1_L + 18/1_L + 24	6	1_L + 6
(bd,BR,Xn)	7	1_L + 21/1_L + 25	7	1_L + 19/1_L + 25	7	1_L + 7
([bd,BR,Xn])	9	1_L + 23/1_L + 27	9	1_L + 21/1_L + 27	9	1_L + 9
([bd,BR,Xn],od)	10	1_L + 24/1_L + 28	10	1_L + 22/1_L + 28	10	1_L + 10
([bd,BR],Xn)	10	3_L + 22/3_L + 26	10	3_L + 20/3_L + 26	10	3_L + 8
([bd,BR],Xn,od)	11	3_L + 23/3_L + 27	11	3_L + 21/3_L + 27	11	3_L + 9

*Multiply <ea> calculate and execute times; T1/T2 apply to word/long-word operand size.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	NEG, NEGX, NOT		PEA		ROL, ROR	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	1	—	—	1	3/4*
An	—	—	—	—	—	—
(An)	1	1	2	1_L + 1	1	3
(An)+	1	1	—	—	1	3
–(An)	1	1	—	—	1	3
(d_{16},An)	1	1	2	1_L + 1	1	3
(d_{16},PC)	—	—	4	3_L + 1	—	—
(xxx).W, (xxx).L	1	1	2	1_L + 1	1	3
#<xxx>	—	—	—	—	—	—
(d_8,An,Xn)	3	3	4	1_L + 3	3	5
(d_8,PC,Xn)	—	—	6	2_L + 4	—	—
(BR,Xn)	6	1_L + 5	7	2_L + 5	6	1_L + 7
(bd,BR,Xn)	7	1_L + 6	8	2_L + 6	7	1_L + 8
([bd,BR,Xn])	9	1_L + 8	10	2_L + 8	9	1_L + 10
([bd,BR,Xn],od)	10	1_L + 9	11	2_L + 9	10	1_L + 11
([bd,BR],Xn)	10	3_L + 7	11	4_L + 7	10	3_L + 9
([bd,BR],Xn,od)	11	3_L + 8	12	4_L + 8	11	3_L + 10

*Immediate count specified for shift count/shift count specified in register, respectively.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Continued)

Addressing Mode	ROXL, ROXR		Scc		SUBA	
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	5/6*	1	2	1	1
An	—	—	—	—	1	2
(An)	1	2	1	2	1	2
(An)+	1	2	1	2	2	1_L + 2
–(An)	1	2	1	2	2	1_L + 2
(d_{16},An)	1	2	1	2	2	1_L + 2
(d_{16},PC)	—	—	—	—	3	2_L + 2
(xxx).W, (xxx).L	1	2	1	2	1	2
#<xxx>	—	—	—	—	1	2
(d_8,An,Xn)	3	4	4	5	4	5
(d_8,PC,Xn)	—	—	—	—	5	1_L + 5
(BR,Xn)	6	1_L + 6	6	1_L + 6	6	1_L + 6
(bd,BR,Xn)	7	1_L + 7	7	1_L + 7	7	1_L + 7
([bd,BR,Xn])	9	1_L + 9	10	1_L + 10	9	1_L + 9
([bd,BR,Xn],od)	10	1_L + 10	11	1_L + 11	10	1_L + 10
([bd,BR],Xn)	10	3_L + 8	11	3_L + 9	10	3_L + 8
([bd,BR],Xn,od)	11	3_L + 9	12	3_L + 10	11	3_L + 9

*Immediate count specified for shift count/shift count specified in register, respectively.

9.6 INTEGER UNIT INSTRUCTION TIMINGS (Concluded)

Addressing Mode	TAS*					
	<ea> Calculate	Execute	<ea> Calculate	Execute	<ea> Calculate	Execute
Dn	1	2				
An	—	—				
(An)	26	2_L + 24				
(An)+	26	2_L + 24				
–(An)	26	2_L + 24				
(d_{16},An)	26	2_L + 24				
(d_{16},PC)	—	—				
(xxx).W, (xxx).L	26	2_L + 24				
#<xxx>	—	—				
(d_8,An,Xn)	27	27				
(d_8,PC,Xn)	—	—				
(BR,Xn)	30	1_L + 28				
(bd,BR,Xn)	31	1_L + 29				
([bd,BR,Xn])	33	33				
([bd,BR,Xn],od)	35	34				
([bd,BR],Xn)	34	3_L + 31				
([bd,BR],Xn,od)	36	3_L + 32				

*Times listed are typical. This instruction interlocks the <ea> calculate and execute stages and synchronizes some portions of the processor before execution.

SECTION 10
ELECTRICAL AND THERMAL CHARACTERISTICS

The following paragraphs provide information on the maximum rating for the MCF5102.

10.1 MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	–0.3 to +4.0	V
Maximum Operating Voltage Minimum Operating Voltage	V_{CC}	+3.6 3.0	V V
Input Voltage	V_{in}	-0.5 to Vcc +3	°C
Recommended Operating Temperature	T_A	0 to 70	°C
Storage Temperature Range	T_{stg}	–55 to 150	°C

This device contains protective circuitry against damage due to high static voltages or electrical fields; however, it is advised that normal precautions be taken to avoid application of any voltages higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either GND or V_{CC}).

10.2 THERMAL CHARACTERISTICS

Characteristic	Symbol	Value	Rating
Thermal Resistance, Junction to Ambient— Thin Plastic Surface Mount Package PV	θ_{JA}	46	°C/W

10.3 DC ELECTRICAL SPECIFICATIONS (V_{CC} = 3.3 vdc ± 0.3 vdc)

Characteristic	Symbol	Min	Max	Unit
Input High Voltage	V_{IH}	2	5.5	V
Input Low Voltage	V_{IL}	GND	0.8	V
Input Low Voltage (BCLK)	V_{IL}	-0.5	0.8	V
Overshoot	See Figure 10-1			V
Input Leakage Current @ 0.5/2.4 V During Normal Operation Only $\overline{AVEC}$, BCLK, $\overline{BG}$, $\overline{CDIS}$, $\overline{IPL}$x, $\overline{RSTI}$, SCx, $\overline{TBI}$, TLNx, $\overline{TCI}$, TCK, $\overline{TEA}$	I_{in}	—	3	µA
Hi-Z (Off-State) Leakage Current @ 0.5/2.4 V During Normal Operation A/Dn, BB, $\overline{CIOUT}$, $\overline{LOCK}$, R/$\overline{W}$, SIZx, $\overline{TA}$, TDO, TMx, TLNx, $\overline{TS}$, TTx,	I_{TSI}	—	3	µA
Signal Low Input Current, (estimated) V_{IL} = 0.8 V TMS, TDI	I_{IL}	0	1	mA
Signal High Input Current, (estimated) V_{IH} = 2.0 V TMS, TDI	I_{IH}	0	1	mA
Output High Voltage $I_{OH = 5ma}$	V_{OH}	2.4	—	V
Output Low Voltage $I_{OL = 5ma}$	V_{OL}	—	0.5	V
Capacitance[1], V_{in} = 0 V, f = 1 MHz	C_{in}	—	15	pF

NOTES:
1. Capacitance is periodically sampled rather than 100% tested.

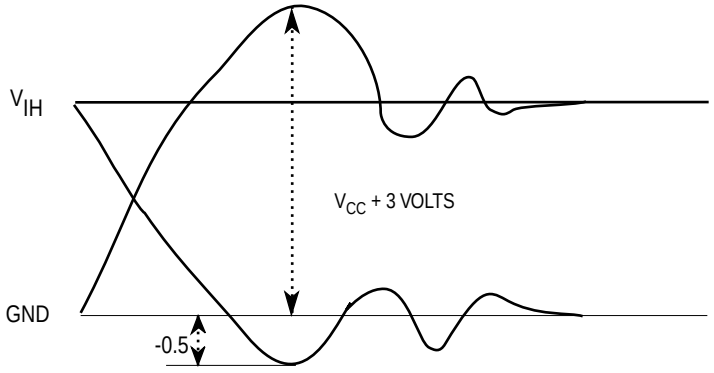


Figure 10-1. Overshoot Diagram

10.4 POWER DISSIPATION

	20 MHz	25 MHz	33 MHz	40 MHz
Worst Case (V_{CC} = 3.6 V , Ta = 0°C)				
MCF5102	.75 W	.95 W	1.25 W	1.5 W
LPSTOP Mode - No output loads, not driving bus				
MCF5102 **V_{CC} = 3.60V, @ TJ = 70° C**	10 mW	10 mW	10 mW	10 mW
Typical Values (V_{CC} = 3.3 V, TJ = 25°C)* - Normal Operation				
MCF5102	.6 W	.75 W	1.0 W	1.2 W

*This information is for system reliability purposes.

10.5 CLOCK AC TIMING SPECIFICATIONS (See Figure 10-2)

		20 MHz		25 MHz		33 MHz		40 MHz		
Num	**Characteristic**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	**Unit**
	Frequency of Operation	0	20	0	25	0	33	0	40	MHz
5	BCLK Cycle Time	50	—	40	—	30	—	25	—	ns
6,7[1]	BCLK Rise and Fall Time	—	2	—	2	—	2	—	2	ns
8	BCLK Duty Cycle Measured at 1.5 V	40	60	40	60	40	60	40	60	%
8a	BCLK Pulse Width High Measured at 1.5 V	20	30	16	24	11	19	10	15	ns
8b	BCLK Pulse Width Low Measured at 1.5 V	20	30	16	24	11	19	10	15	ns
9	BCLK edge to edge jitter	—	125	—	125	—	125	—	125	ps

NOTES:
1. Rising and falling edges of BCLK must be monotonic.

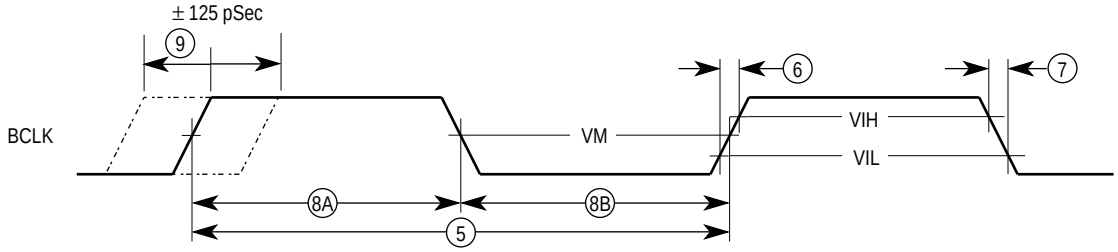


Figure 10-2. CLock Input Timing Diagram

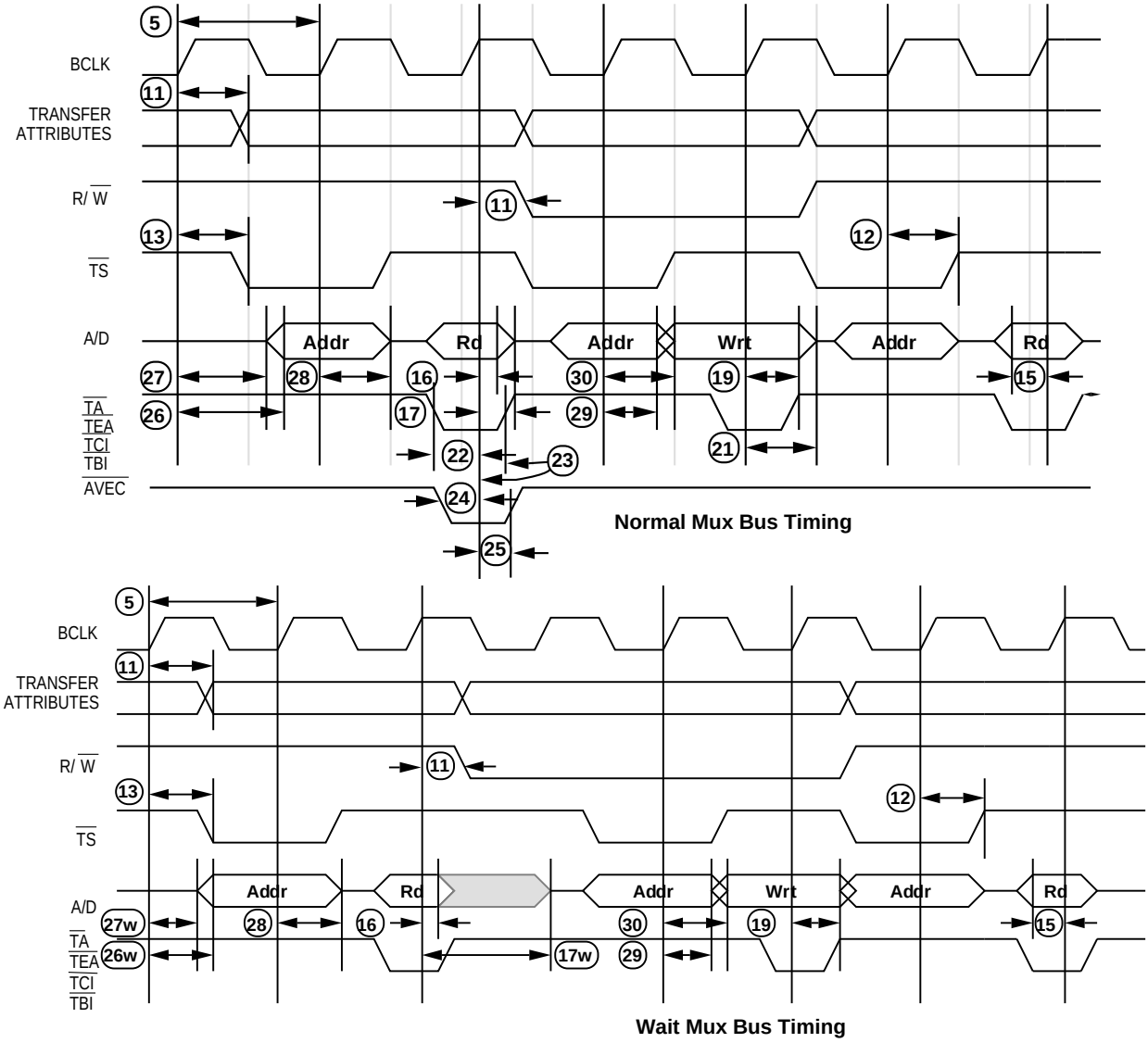
10.6 MULTIPLEXED TIMING SPECIFICATIONS (see Figure 10-3)

Num	**Characteristic**	**20MHz**		**25 MHz**		**33 MHz**		**40 MHz**		**Unit**
		Min	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**	
12	BCLK to Output Invalid (Output Hold)	6.5	—	6.5	—	6.5	—	5.25	—	ns
13	BCLK to $\overline{TS}$ Valid	6.5	35	6.5	25	6.5	20	5.25	18	ns
15	Data-In Valid to BCLK (Setup)	5	—	5	—	4	—	3	—	ns
16	BCLK to Data-In Invalid (Hold)	5	—	4	—	4	—	3	—	ns
17	BCLK to Data-In High Impedance	5	19	4	19	4	13	3	12	ns
17W[2]	BCLK to Data-In High Impedance	5	62.5	4	50	4	36	3	31	ns
19	BCLK to Data-Out Invalid (Output Hold)	6.5	—	6.5	—	6.5	—	5.25	—	ns
21[1]	BCLK to Data-Out High Impedance	6.5	25	6.5	20	6.5	17	5.25	16	ns
26	BCLK to Multiplexed Address Valid	14	45	14	35	14	33	13	30	ns

Num	Characteristic	20MHz		25 MHz		33 MHz		40 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
26W[2]	BCLK to Multiplexed Address Valid	7.5	35	7.5	25	7.5	20	5.25	18	ns
27	BCLK to Multiplexed Address Driven	14	—	14	—	14	—	13	—	ns
27W[2]	BCLK to Multiplexed Address Driven	6.5	—	6.5	—	6.5	—	5.25	—	ns
28	BCLK to Multiplexed Address High Impedance	6.5	25	6.5	20	6.5	15	5.25	14	ns
29	BCLK to Multiplexed Data Driven	6.5	—	6.5	—	6.5	—	5.25	—	ns
30	BCLK to Multiplexed Data-Out Valid	6.5	37	6.5	27	6.5	22	5.25	18	ns

NOTES:

1. Following a loss of bus ownership.
2. WAITER pin asserted.
3. The output timing specifications for the MCF5102 pertain to 3.3V systems only. If the MCF5102 is used in a system containing 5V devices, then the appropriate specifications should be derated by 2ns. This decrease is necessary because of the additional time required to drive signals to ground from 5V as opposed to 3.3V. An example of this would be a MCF5102 used in a system that contains 5V code ROMs but no alternate bus masters. Because the ROMs can drive the A/D bus to 5V during a read cycle, specification 26 (or 26W), “BCLK to Multiplexed Address Valid” should be derated by 2ns. Since the MCF5102 will be the only device driving the TS* signal, this signal voltage will never exceed 3.3V. Therefore, specification 13, “BCLK to TS* Valid” should not be decreased.



Note: Transfer attribute signals = SIZx, TTx, TMx, TLNx,R/W, LOCK, CLOUT

Table 10-3. Read/Write Timing

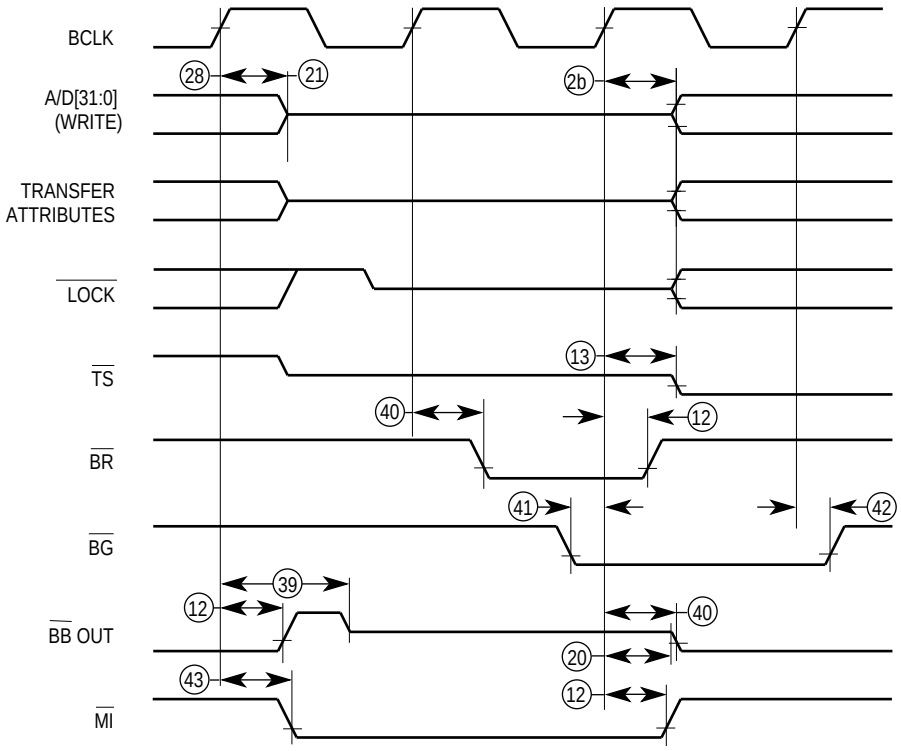
10.7 OUTPUT AC TIMING SPECIFICATIONS (see Figures 10-4 to 10-11)

Num	Characteristic	20 MHz		25 MHz		33 MHz		40 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
11	BCLK to $\overline{\text{CIOUT}}$, $\overline{\text{LOCK}}$, PSTx, R/$\overline{\text{W}}$, SIZx, TLNx, TMx, TTx, Valid	6.5	35	6.5	25	6.5	20	5.25	18	ns
19	BCLK to Data-Out Invalid (Output Hold)	6.5	—	6.5	—	6.5	—	5.25	—	ns
38	BCLK to $\overline{\text{CIOUT}}$, LOCK, R/$\overline{\text{W}}$, SIZx, $\overline{\text{TS}}$, TMx, TTx, High Impedance	6.5	23	6.5	18	6.5	15	5.25	14	ns
39	BCLK to $\overline{\text{BB}}$, $\overline{\text{TA}}$, High Impedance	23	33	19	28	14	25	12	22	ns
40	BCLK to $\overline{\text{BR}}$, $\overline{\text{BB}}$ Valid	6.5	35	6.5	30	6.5	23	5.25	20	ns
43	BCLK to $\overline{\text{MI}}$ Valid	6.5	35	6.5	30	6.5	25	5.25	20	ns
48	BCLK to $\overline{\text{TA}}$ Valid	6.5	35	6.5	30	6.5	25	5.25	20	ns
50	BCLK to $\overline{\text{IPEND}}$, PSTx, $\overline{\text{RSTO}}$, $\overline{\text{SCD}}$ Valid	6.5	35	6.5	30	6.5	25	5.25	20	ns
W	$\overline{\text{RSTI}}$ active to $\overline{\text{SCD}}$ inactive.	8	100	8	100	8	100	3	100	ns
A	$\overline{\text{IPLx}}$ to $\overline{\text{SCD}}$ invalid	8	100	8	100	8	100	3	100	ns

NOTE: Output timing is specified for a valid signal measured at the pin. Timing is specified driving a 50pf capactive load.

10.8 INPUT AC TIMING SPECIFICATIONS (see Figures 10-4 to 10-11)

Num	Characteristic	20 MHz		25 MHz		33 MHz		40 MHz		Unit
		Min	Max	Min	Max	Min	Max	Min	Max	
22a	$\overline{TA}$ Valid to BCLK (Setup)	12.5	—	10	—	10	—	9	—	ns
22b	$\overline{TEA}$ Valid to BCLK (Setup)	12.5	—	10	—	10	—	9	—	ns
22c	$\overline{TCI}$ Valid to BCLK (Setup)	12.5	—	10	—	10	—	9	—	ns
22d	$\overline{TBI}$ Valid to BCLK (Setup)	14	—	11	—	10	—	9	—	ns
23	BCLK to $\overline{TA}$, $\overline{TEA}$, $\overline{TCI}$, $\overline{TBI}$ Invalid (Hold)	2.5	—	2	—	2	—	2	—	ns
24	$\overline{AVEC}$ Valid to BCLK (Setup)	6	—	5	—	5	—	5	—	ns
25	BCLK to $\overline{AVEC}$ Invalid (Hold)	2.5	—	2	—	2	—	2	—	ns
41a	$\overline{BB}$ Valid to BCLK (Setup)	8	—	7	—	7	—	7	—	ns
41b	$\overline{BG}$ Valid to BCLK (Setup)	10	—	8	—	7	—	7	—	ns
41c	$\overline{CDIS}$ Valid to BCLK (Setup)	12.5	—	10	—	8	—	8	—	ns
41d	$\overline{IPLx}$ Valid to BCLK (Setup)	5	—	4	—	3	—	3	—	ns
42	BCLK to $\overline{BB}$, $\overline{BG}$, $\overline{CDIS}$, $\overline{IPLx}$ Invalid (Hold)	2.5	—	2	—	2	—	2	—	ns
44a	Address Valid to BCLK (Setup)	10	—	8	—	7	—	6	—	ns
44b	SIZxValid to BCLK (Setup)	15	—	12	—	8	—	8	—	ns
44c	TTx Valid to BCLK (Setup)	7.5	—	6	—	8.5	—	8.5	—	ns
44d	R/$\overline{W}$ Valid to BCLK (Setup)	7.7	—	6	—	5	—	5	—	ns
44e	SCx Valid to BCLK (Setup)	12.5	—	10	—	11	—	8	—	ns
45	BCLK to Address SIZx ,TTx, R/$\overline{W}$, SCx Invalid (Hold)	2.5	—	2	—	2	—	2	—	ns
46	$\overline{TS}$ Valid to BCLK (Setup)	6	—	5	—	4	—	4	—	ns
47	BCLK to $\overline{TS}$ Invalid (Hold)	2.5	—	2	—	2	—	2	—	ns
49	BCLK to $\overline{BB}$ High Impedance (Processor Assumes Bus Mastership)	—	11	—	9	—	9	—	—	ns
51	$\overline{RSTI}$ Asserted to BCLK (Setup)	6	—	5	—	4	—	4	—	ns
52	BCLK to $\overline{RSTI}$ Negated (Hold)	2.5	—	2	—	2	—	2	—	ns
D	$\overline{IPEND}$ valid to $\overline{IPLx}$ invalid (Hold)	0	—	0	—	0	—	0	—	ns
V	$\overline{RSTI}$ pulse width, leaving LPSTOP mode	10	—	10	—	10	—	10	—	ns



NOTE: Transfer Attribute Signals = SIZx, TTx, TMx, TLNx, R/$\overline{W}$, $\overline{CIOUT}$

Figure 10-4. Bus Arbitration Timing

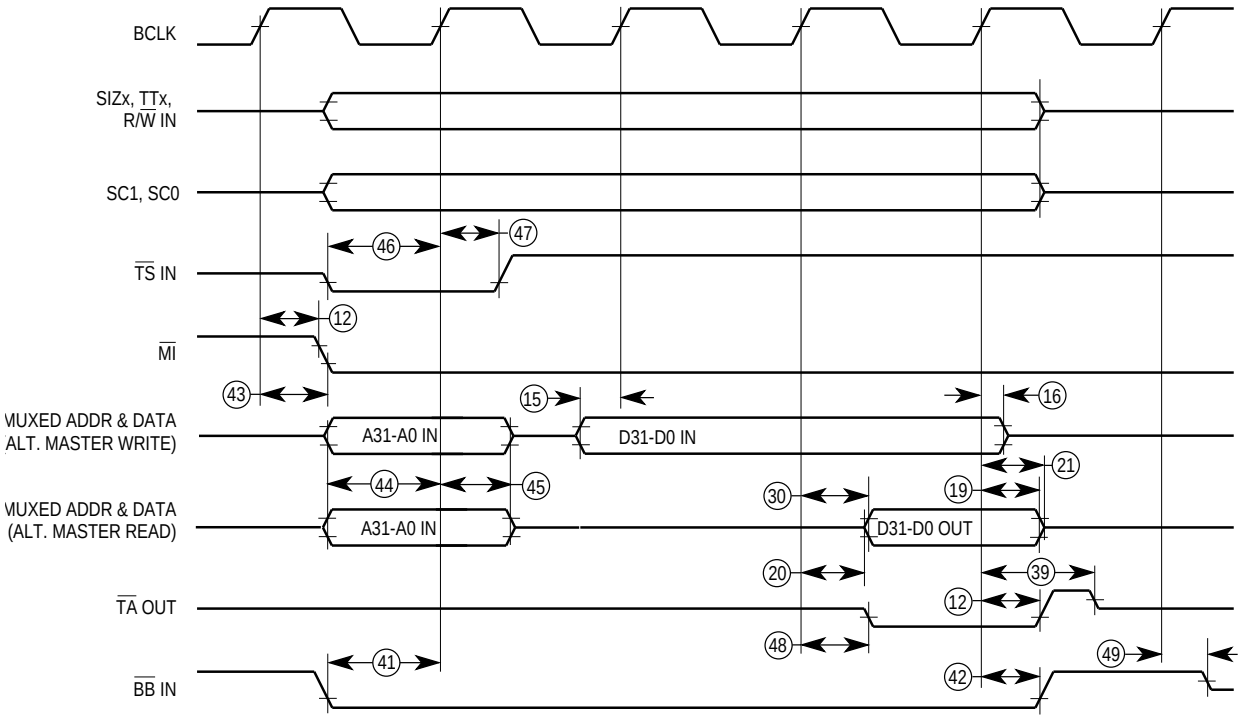


Figure 10-5. Bus Snoop Hit Timing

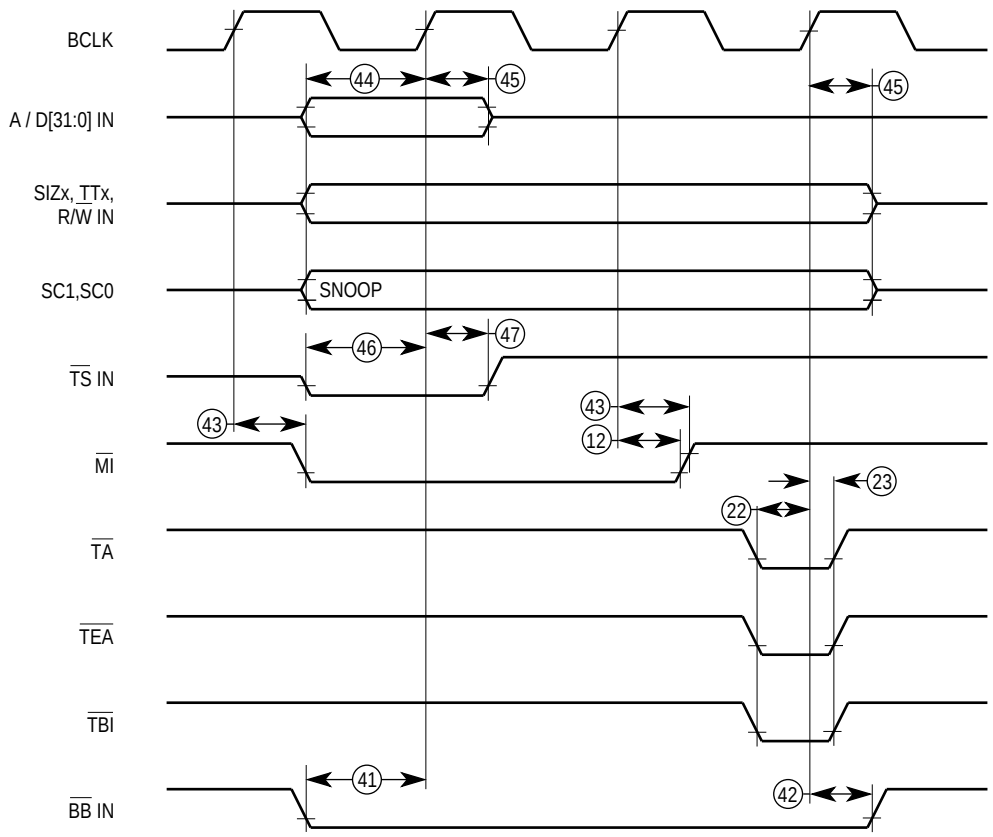


Figure 10-6. Snoop Miss Timing

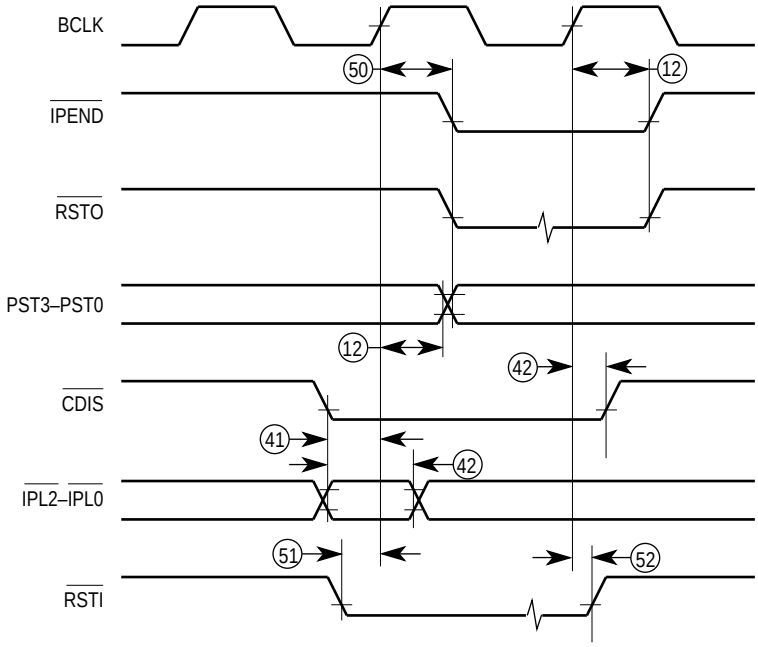


Figure 10-7. Other Signal Timing

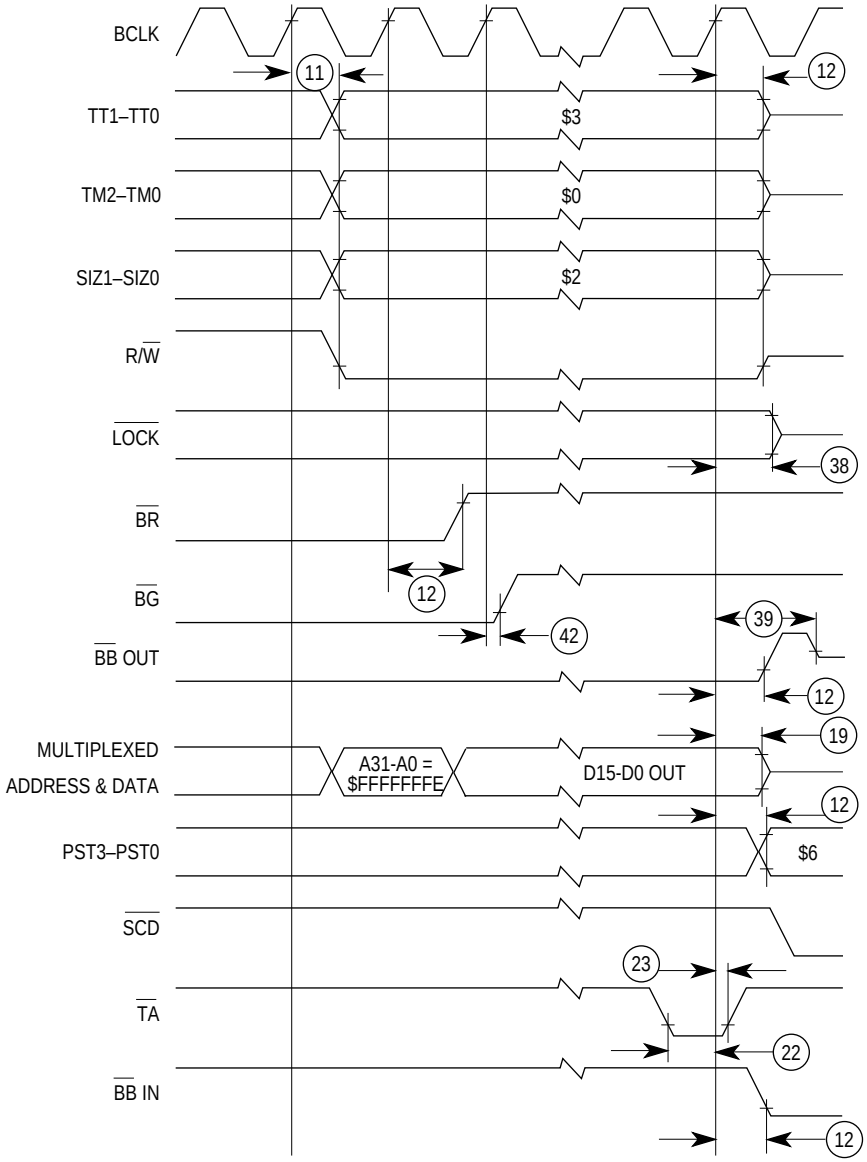


Figure 10-8. Going into LPSTOP with Arbitration

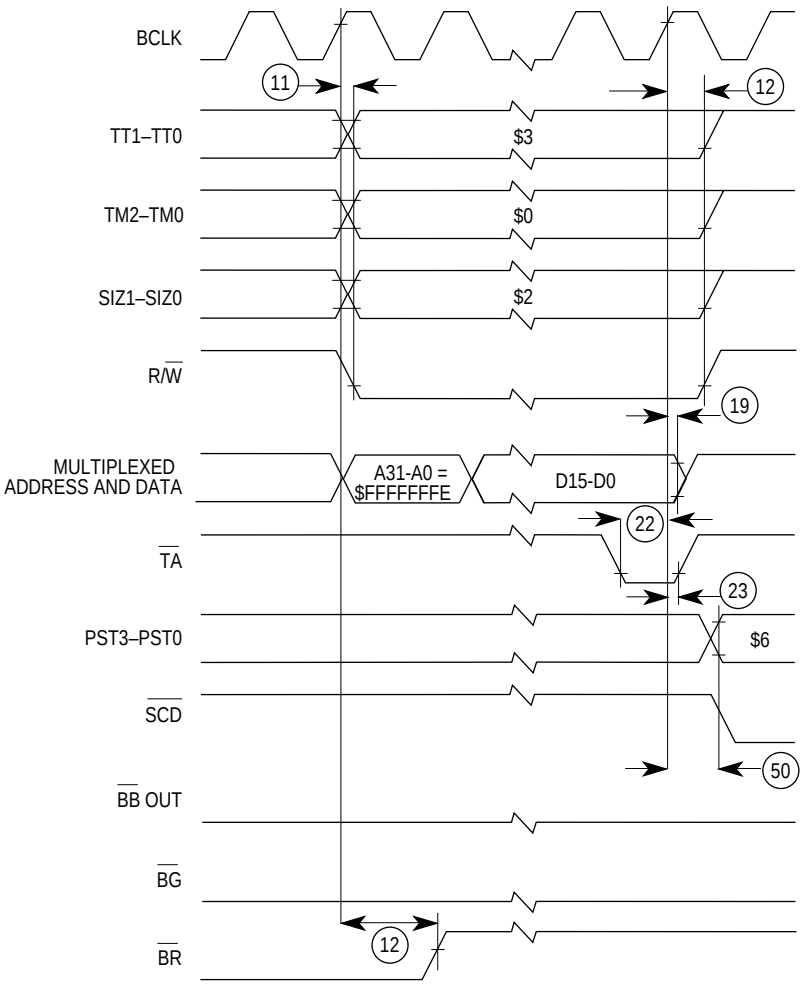


Figure 10-9. LPSTOP no Arbitration, CPU is Master

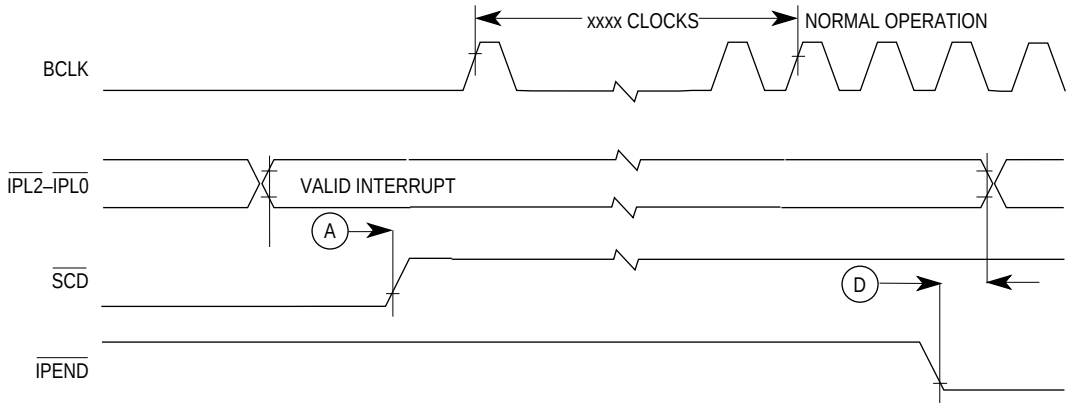


Figure 10-10. Exiting LPSTOP with Interrupt

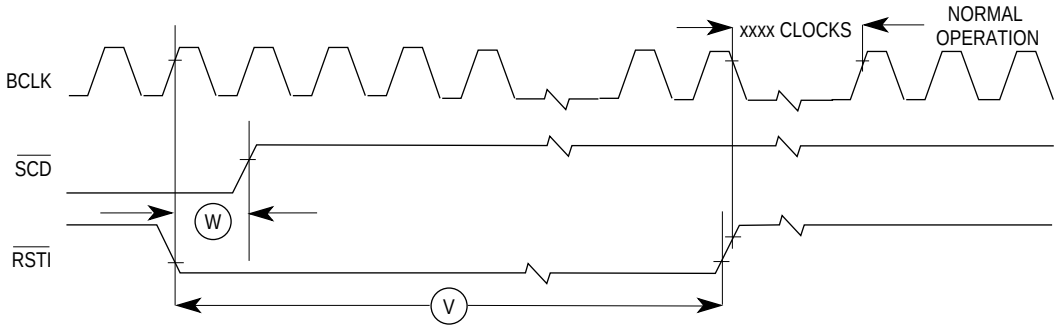


Figure 10-11. Exiting of LPSTOP with RESET

SECTION 11
ORDERING INFORMATION AND MECHANICAL DATA

This section contains the ordering information, pin assignments, and package dimensions FOR the MCF5102

11.1 ORDERING INFORMATION

The following table provides ordering information pertaining to package types, frequencies, temperatures, and Motorola order numbers.

Package Type	Frequency	Maximum Junction Temperature	Minimum Ambient Temperature	Order Number
Thin Plastic Quad-Flat Pack PV Suffix	16.67 MHz 20 MHz 25MHZ	110 °C	0 °C	XCF5102PV16A XCF5102PV20A XCF5102PV25A

11.2 PIN ASSIGNMENTS

Figure 11-1 shows the pin assignments for the MCF5102.

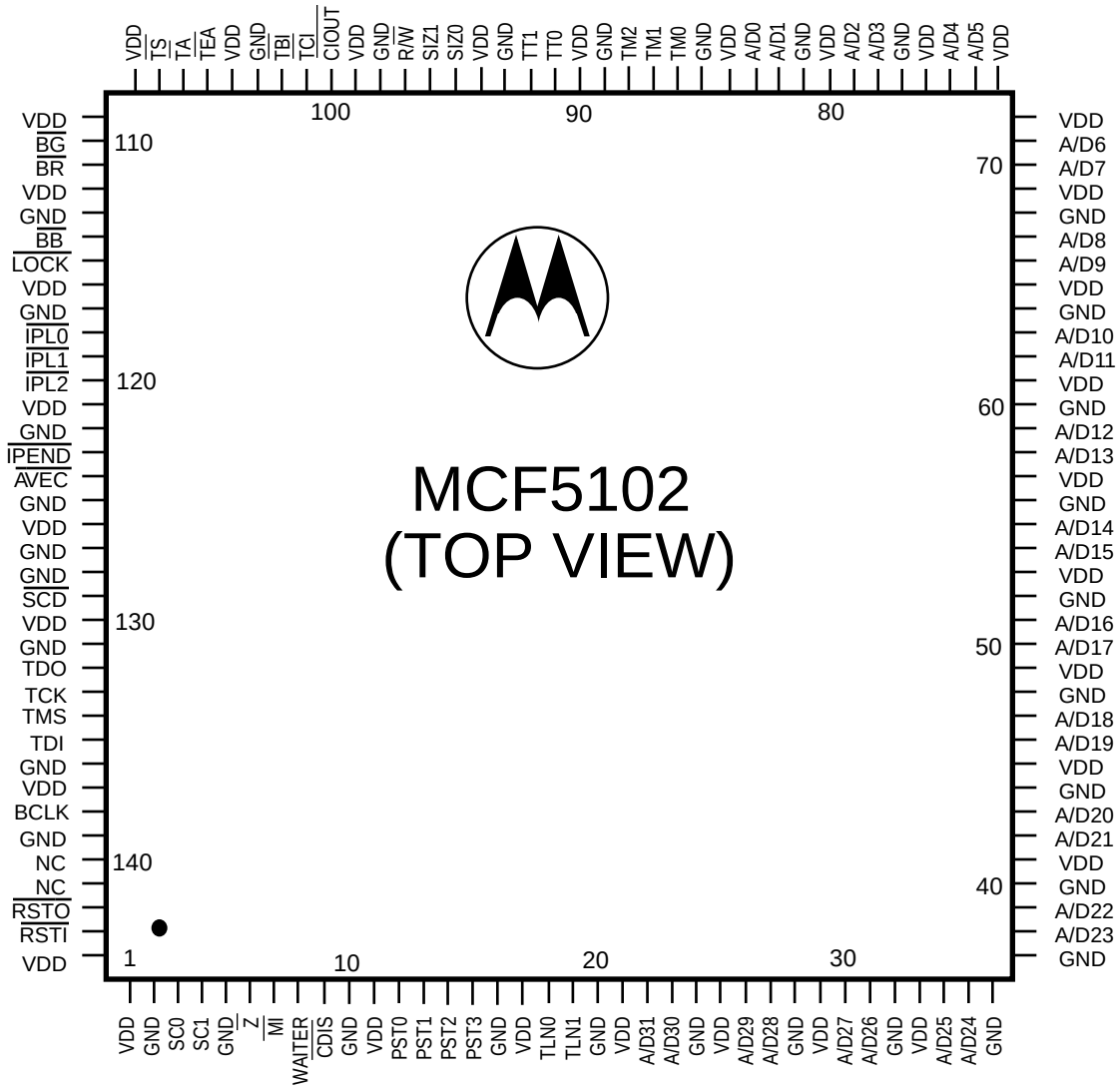
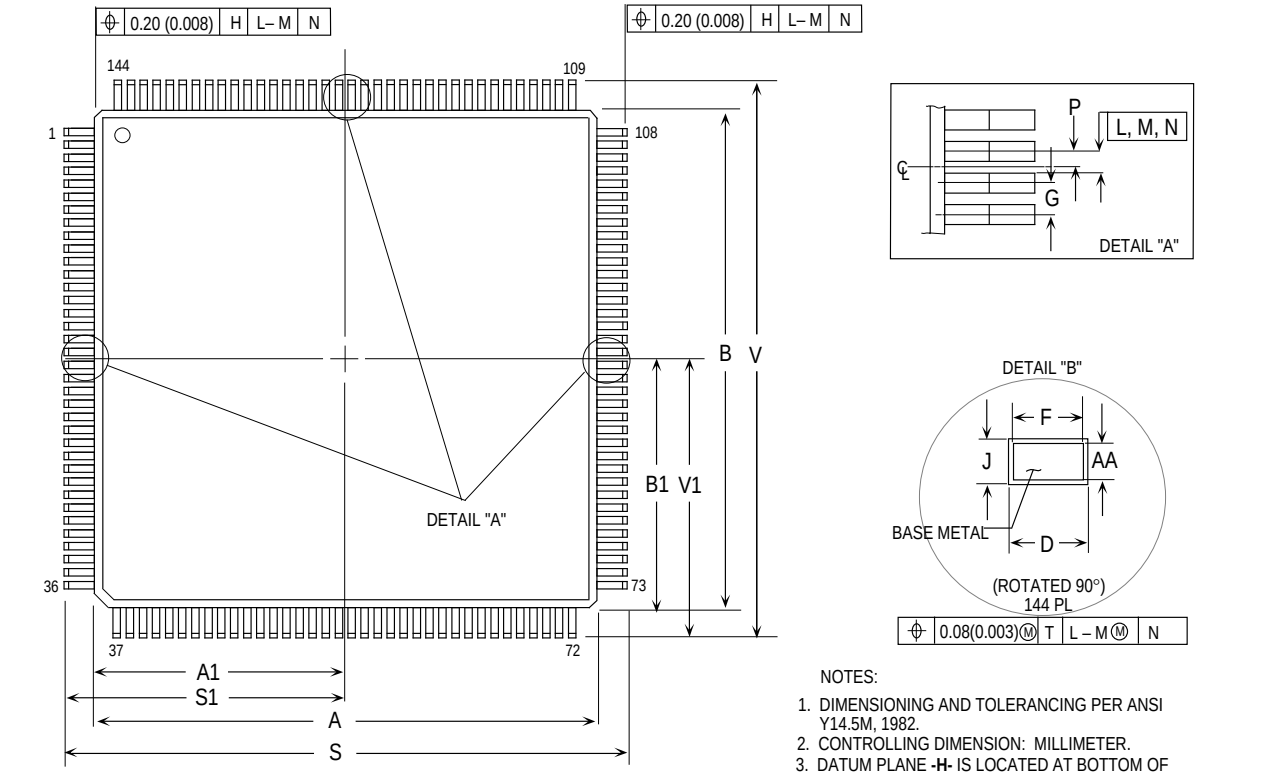


Figure 11-1. MCF5102 Pinout

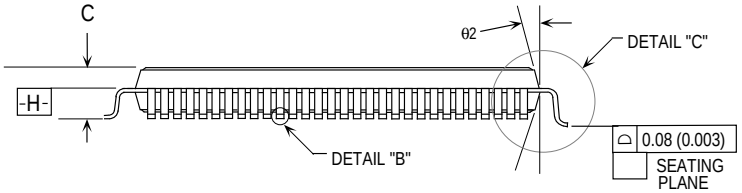
11.3 MECHANICAL DATA

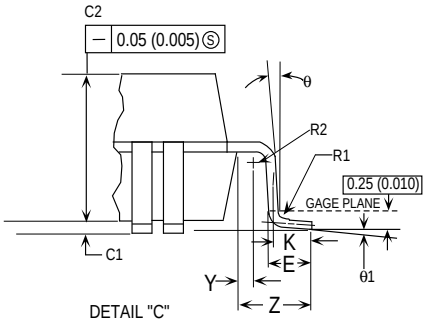
Figure 11-2 illustrates the MCF5102 TQFP package dimensions.



NOTES:

1. DIMENSIONING AND TOLERANCING PER ANSI Y14.5M, 1982.
2. CONTROLLING DIMENSION: MILLIMETER.
3. DATUM PLANE **-H-** IS LOCATED AT BOTTOM OF LEAD AND COINCIDENT WITH THE LEAD WHERE THE LEAD EXITS THE PLASTIC BODY AT THE BOTTOM OF THE PARTING LINE.
4. DATUMS **-L-**, **-M-**, AND **-N-** TO BE DETERMINED AT DATUM PLANE **-H-**.
5. DIMENSIONS S AND V TO BE DETERMINED AT SEATING PLANE **-T-**.
6. DIMENSIONS A AND B DO NOT INCLUDE MOLD PROTRUSION. ALLOWABLEPROTRUSION IS 0.25 (0.010 PER SIDE. DIMENSIONS A AND B DO NOT INCLUDE MOLD MISMATCH AND ARE DETERMINED AT DATUM LINE **-H-**.
7. DIMENSION D DOES NOT INCLUDE DAMBAR PROTRUSION. ALLOWABLEDAMBAR PROTRUSION SHALL NOT CAUSE THE D DIMENSION TO EXCEED 0.35 (0.014).





	MILLIMETERS		INCHES	
DIM	MIN	MAX	MIN	MAX
A	20.00 BSC		0.790 BSC	
A1	10.00 BSC		0.394 BSC	
B	20.00 BSC		0.790 BSC	
B1	10.00 BSC		0.394 BSC	
C	1.40	1.60	0.055	0.063
C1	0.05	0.15	0.002	0.006
C2	1.35	1.45	0.053	0.057
D	0.17	0.27	0.007	0.011
E	0.45	0.75	0.018	0.030
F	0.17	0.23	0.007	0.009
G	0.50 BSC.		0.20 BSC.	
J	0.09	0.20	0.004	0.008
K	0.50 REF		0.020 REF	
P	0.25 BSC		0.010 BSC	
R1	0.13	0.20	0.005	0.008
R2	0.13	0.20	0.005	0.008
S	22.00 BSC		0.866 BSC	
S1	11.00 BSC		0.433 BSC	
V	22.00 BSC		0.866 BSC	
V1	11.00 BSC		0.433 BSC	
Y	0.25 REF		0.010 REF	
Z	1.00 REF		0.039 REF	
AA	0.09	0.16	0.004	0.006
θ	0°		0°	
θ1	0°	7°	0°	7°
θ2	11°	13°	11°	13°

Figure 11-2. TQFP Package Dimensions

APPENDIX A
ADDRESS, $\overline{\text{TIP}}$, AND $\overline{\text{LOCKE}}$ GENERATION

The MCF5102's bus is similar with the MC68040's bus in multiplexed-bus mode. That bus definition includes $\overline{\text{TIP}}$ (transfer in progress) and $\overline{\text{LOCKE}}$ (locked-sequence ending) signals, which are useful in some applications. The MCF5102's bus, while similar in other respects, omits these signals. This application presents considerations for generating these signals externally, and for latching address from the A/D pins.

$\overline{\text{TIP}}$ and $\overline{\text{LOCKE}}$ are easier to generate with a wait-state inserted into the address phase of every bus transaction. In that case, these signals can be generated from a synchronous output of a PAL. Without injecting the wait-state, it must be generated asynchronously.

Sample PAL programs are included for synchronous and asynchronous implementations.

A.1 DEFINITIONS

BCLK Cycle: The period of time between two adjacent rising edges of the BCLK signal.

$\overline{\text{BERR}}$: Abbreviation for "bus error".

Bus Transaction: Any action over the MCF5102's bus-interface signals carried out according to the databook specifications, notably: any data or instruction read or write, an interrupt or breakpoint acknowledge, a snooped bus action, or an LPSTOP broadcast.

Locked Sequence: A sequence of bus transactions during which the LOCK signal continuously is asserted to prevent an alternate master from disturbing the data being manipulated.

Address Phase: The first BCLK cycle of a bus transaction, during which time the address is presented on the A/D pins.

Data Phase: The BCLK cycle(s) following the address phase of bus transaction, including any wait-states. A single data phase nominally results in a single data transfer, but could also end in a $\overline{\text{BERR}}$ or retry.

Wait-State: A data-phase BCLK cycle during which neither $\overline{TA}$ nor $\overline{TEA}$ are asserted.

Data Transfer: A single data movement over the A/D pins.

Burst Transaction: A bus transaction consisting of an address phase followed directly by four data phases (provided that no bus errors nor retries occur). A $\overline{BERR}$ or retry aborts a bus transaction. An inhibited burst (SIZ=11 and TBI=0) produces nonburst cycles, normally a total of four.

Nonburst Transaction: A bus transaction consisting of an address phase followed by exactly one data phase.

Open vs. Closed latch: A transparent latch is “closed” when its latch enable pin is low - when its outputs reflect stored data. It is “open” when it is transparent or acting like a buffer.

A.2 ADDRESS LATCHING

A.2.1 Using Clock-Enabled Flip-Flops

Using clock-enabled flip-flops provides a clean way to demultiplex address from the A/D pins. See Figure A-1.

This provides usable timing for a single-bus-master, 20MHz system with a wait-state injected into to the address phase of every bus transaction. Because of the wait-state injected in the $\overline{TS}$ path, $\overline{TIP}$ can be generated from a simple synchronous state machine implemented in an inexpensive PAL. (The cache hit rates are high enough to make the impact of this additional wait-state small.)

Latching addresses with 74F377s has several disadvantages. One is that setup time on the 74F377s is difficult to satisfy. Because many memory devices have slow output-disable times, the MCF5102 drives address onto the A/D pins late in the address phase of the bus transaction (if WAITER is negated). Operating at 20MHz, spec#26 guarantees only 5ns of setup time to the next BCLK, and at 25MHz no setup time at all is guaranteed.

Additionally, the clock-enabled flip-flops add additional BCLK loads, and may limit choices of available parts in a multiple-master system. The 74F377 does not provide an output disable, a multimaster system will require additional buffers or a more specialized part type that provides both clock enable and output enable.

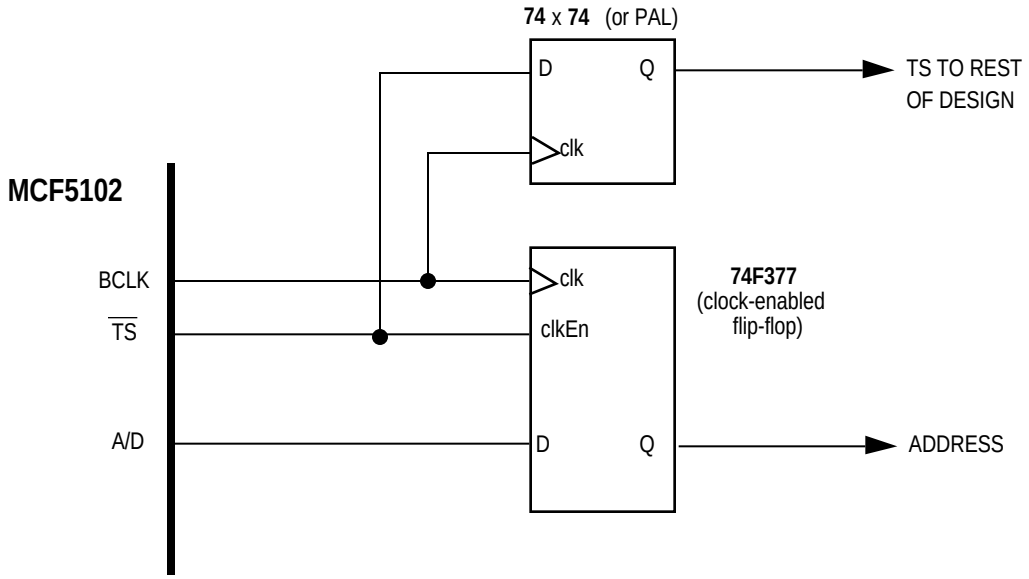


Figure A-1. Clock -Enabled Flip-Flop Address Latching

A.2.2 Using Transparent Latches

Transparent latches are the most efficient way to demultiplex addresses in a performance-sensitive environment, but doing so presents some timing considerations. With latches address setup compared with address hold requirements are critical. This affects whether or not to hold the latches open or closed between bus transactions, and whether to assert the WAITER pin.

With the address latches opened at the end of each bus transaction, the next bus cycle's address will come valid only one D-to-Q propagation delay time after they come valid on the address/data (A/D) pins. With WAITER asserted, address will be valid during the entire address phase, giving ample setup time for address decoders and memory devices. With WAITER negated, however, address is guaranteed valid, at the earliest, only a few nanoseconds before the beginning of the first data phase.

Address access time is often the critical timing path for memory accesses. If satisfying these critical paths requires adding a wait-state, then use WAITER to insert one wait-state. Asserting waiter improves other bus timings as well as setup.

Although opening the latch at the end of each bus transaction (with WAITER asserted) provides ample address setup time to the next bus transaction, it provides very short address hold times. Spec #12w guarantees only 5ns of hold time after BCLK on a 25MHz part. To drive the latch enable low and satisfy the latch's hold time within 5ns, requires a 5ns PAL and fast latches. Designing to the (maximum) specifications of such fast devices in order to satisfy this very short hold time, also guarantees an even shorter (closer to typical) hold time after the bus transaction ends. If this hold time is not sufficient, then consider raising latch enable only during the BCLK-low time when which $\overline{TS}$ goes active (see figure 3 below):

This circuit provides ample address hold time at the end of a bus transaction, and with WAITER asserted, provides address setup times roughly comparable with MC68040 interfaces. With WAITER negated, it provides the best address setup times possible.

This circuit has a glitch hazard on the latch-enable under worst-case conditions.

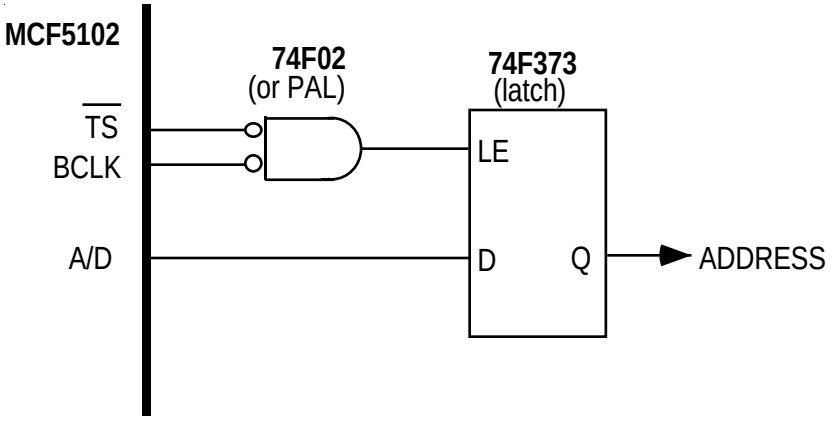


Figure A-2. Normally-Closed Latch-Enable Circuit

Systems that require a $\overline{TIP}$ signal will need a state-machine to count data transfers of a burst cycle. That state-machine will know when the address phase has ended, and can provide an additional qualifier to mask this potential glitch. Alternatively, TS passed through a 1-clock delay flip-flop can provide this qualifier.

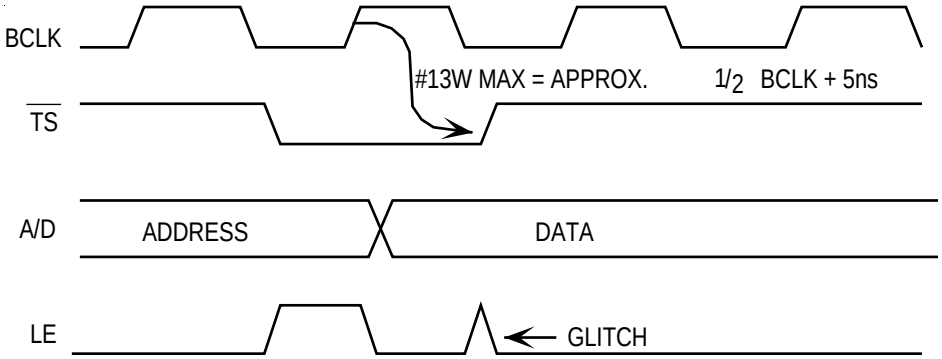


Figure A-3. Timing Hazard in Normally-Closed Latch-Enable Circuit

A.2.3 Using PCLK with Transparent Latches

With WAITER asserted, there will always be at least one bus-inactive BCLK cycle between adjacent bus transactions. In this scenario, opening the latches half a BCLK after the end of the bus transaction.

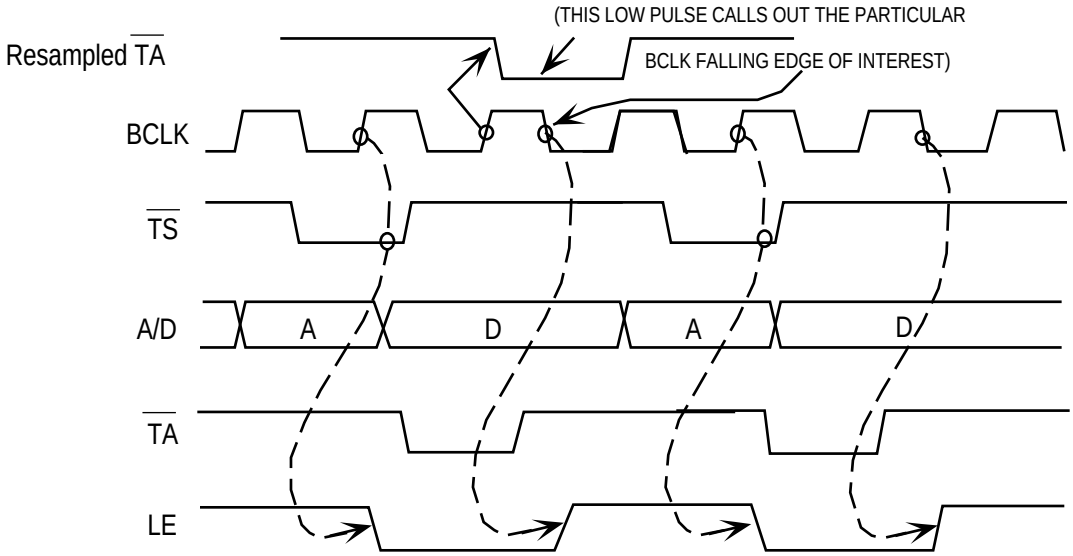


Figure A-4. Negating Latch-Enable Half-BCLK After End of Bus Transaction

If the system has access to a clock running at twice BCLK frequency (like the 68040's PCLK) then this effect can be obtained by:

1. Clocking the PAL on PCLK, and...
2. Qualifying all signal value changes that occur on the rising edge of BCLK, on BCLK being low. Since (unlike the rest) the rising edge of LE needs to occur on a BCLK falling edge, then that transition should be qualified on BCLK being high.

In this case, $\overline{TA}$ may or may not have negated on the following BCLK falling edge. Therefore, resample $\overline{TA}$ on the BCLK rising edge, then raise LE when that resampled $\overline{TA}$ is low.

There are two important caveats to this:

1. BCLK must setup to PCLK; BCLK transitions must occur a few nanoseconds after PCLK rising edges. The MCF5102 timing specifications require only that BCLK transitions occur within 9ns, before or after PCLK rising edges.
2. To avoid a metastable condition, all signals used in the PAL must transition within a half-BCLK-time period after the rising edge (minus skew), or be clearly gated on BCLK being low in the reduced PAL equations.

A.3 PROPERTIES OF $\overline{TIP}$ SIGNAL

1. $\overline{TIP}$ asserts approximately simultaneously with $\overline{TS}$, and remains asserted throughout the bus transaction.
2. $\overline{TIP}$ does not necessarily negate after the completion of a bus transaction. It will negate during any bus-inactive time between the end of that transaction and the beginning of the next. If the next bus transaction directly follows the current one, then there will be no intervening bus-inactive time, so it will stay asserted throughout both bus transactions.

3. Asserting WAITER ensures that there will always be at least one bus-inactive BCLK cycle between adjacent bus transactions.
4. $\overline{\text{TIP}}$ stays asserted throughout all data phases of a burst transaction.
5. As with the MC68040, the MCF5102 initiates a burst transaction with the intention of transferring always exactly four longwords. A burst transfer will never transfer more than four longwords, and only externally visible exceptional circumstances ($\overline{\text{TEA}}$, TBI, or $\overline{\text{RSTI}}$) can cause it to transfer fewer than four.
6. $\overline{\text{TIP}}$ must go high impedance when the processor relinquishes the bus.
7. A retry on the second, third, or last transfers of a burst transaction are treated as though they were bus errors.
8. The state of the WAITER pin slightly changes the nanosecond-level timing of $\overline{\text{TS}}$; this could affect the timing of $\overline{\text{TIP}}$ if generated asynchronously from it.

A.4 GENERATING $\overline{\text{TIP}}$

There are two ways generate $\overline{\text{TIP}}$:

Synchronously: As a synchronous state-machine output. This requires injecting a wait-state into the addressing phase of every bus transaction.

Asynchronously: As an asynchronous state-machine output. This asynchronous state machine is much more complex, but does not force a wait-state.

Note that this wait-state is not related to the wait-state that the WAITER pin injects. Using the synchronous approach with WAITER asserted injects two wait-states in the addressing phase of each bus transaction.

A.4.1 Synchronous $\overline{\text{TIP}}$ Generation

Using the synchronous approach, $\overline{\text{TS}}$ from the MCF5102 must also be delayed by one BCLK to match the one-BCLK delay in generating $\overline{\text{TIP}}$. This delay provides greater address setup time, and allows a synchronous state machine (clocked on BCLK) to drive $\overline{\text{TIP}}$. In the processor's view, this delay looks like a wait-state, but to the rest of the board this delay looks like a bus-inactive BCLK cycle between bus transactions.

Here is the state-machine to produce $\overline{\text{TIP}}$ synchronously.

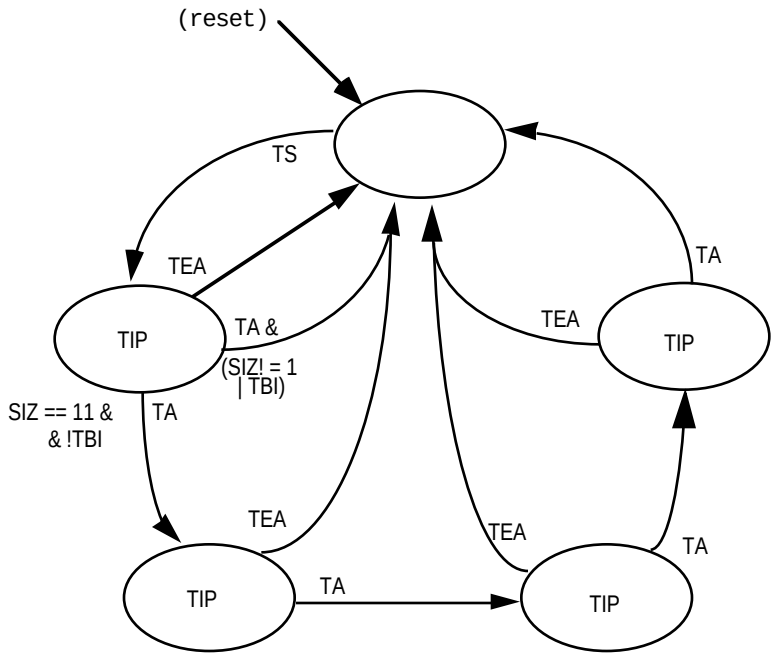


Figure A-5. Synchronous $\overline{\text{TIP}}$ Generation State Diagram

A more complex state machine may be needed if $\overline{\text{LOCKE}}$ is to be generated as well. (Note that the names inside the state ovals in this diagram are outputs instead of state names. In Figure A-6 they are state names.)

A.4.2 Asynchronous $\overline{\text{TIP}}$ Generation

If injecting a wait-state in every address phase is not acceptable, then $\overline{\text{TIP}}$ must be generated asynchronously.

1. The assertion of $\overline{\text{TS}}$ itself must cause $\overline{\text{TIP}}$ to assert.
2. A state machine is still needed to track each of the four data phases of a burst transaction. Minimally, $\overline{\text{TIP}}$ must be asserted when $\overline{\text{TS}}$ is asserted or when that state machine indicates that the bus transaction is in a data phase.
3. Depending on clock skews, loading, and PAL speeds, $\overline{\text{TS}}$ could negate before your state machine transitions to a data-phase state. To make this unproblematic, create a signal that asserts when $\overline{\text{TS}}$ asserts, and negates when $\overline{\text{TA}}$ (or $\overline{\text{TEA}}$) asserts:

```
tHold =   !rsti  "force inactive during reset
        & (   ts & bg  "assert with TS
            # tHold & !ta & !tea       "retain until TA (or TEA)
  asserts
          );
```

4. If all circuitry examines $\overline{\text{TIP}}$ only on BCLK edges, then this becomes a lesser concern.
5. $\overline{\text{TIP}}$ must go high-impedance when the processor relinquishes control of the bus.

A.5 GENERATING $\overline{\text{LOCKE}}$

The MCF5102 does not generate this signal, and it turns out to be virtually impossible to generate it for all possible cases solely from externally-visible signals.

A fair means of detecting the last bus transaction in a locked sequence is to count the total number of reads that occur in the locked sequence, and assert $\overline{\text{LOCKE}}$ during the last of the same number of writes. Counting reads is important, since operands can aligned or misaligned, and of various data sizes. The PAL listings in Appendices A and B, use this technique to generate $\overline{\text{LOCKE}}$. It will not operate correctly:

1. For a CAS2 instruction: If CAS2 does not update its destination operands, it rewrites unchanged data only to the first of them. Since it does not write to both, the write-count will mismatch the read-count.
2. For a CAS instruction where the memory operand is cached in copyback mode. This is not an interesting case anyway, since it is not meaningful unless that operand were snooped, and this PAL does not support snooping. The case where this does not work is actually more specific: The operand must be cacheable, cached in copyback mode, nonbyte, misaligned, straddling between two cache lines, and the second of those two lines must actually be dirty. This design will work if the first line only is dirty.

The asynchronous PAL state-machine has three types of states:

Address-Phase States: The machine is in these states when it is waiting for, or during, an address phase. “reset” (waiting for the address phase of the first bus transaction), and the address phases of the locked bus transactions are of this type. Other than “reset”, these states and the nonburst data-phase states are named in the form. See

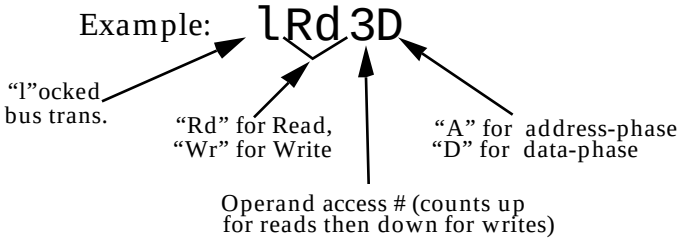


Figure A-6. Address Phase States

(These namings only apply to the asynchronous PAL.)

Nonburst Data-Phase States: The machine is in these states when it is waiting for the data phase to complete (see the naming convention above).

Burst Data-Phase States: The machine sequences through these states to keep track of the four data transfers of a burst cycle. They are named cycD (which also forms the data-phase state of a non-burst, non-locked bus transaction), cycD1, cycD2, and cycD3, (which track each of the additional three data transfers of a burst bus transaction).

Disregarding provisions for $\overline{\text{BERR}}$, retry, and loss of LOCK, the state-diagram simplifies as stated in Figure A-7.

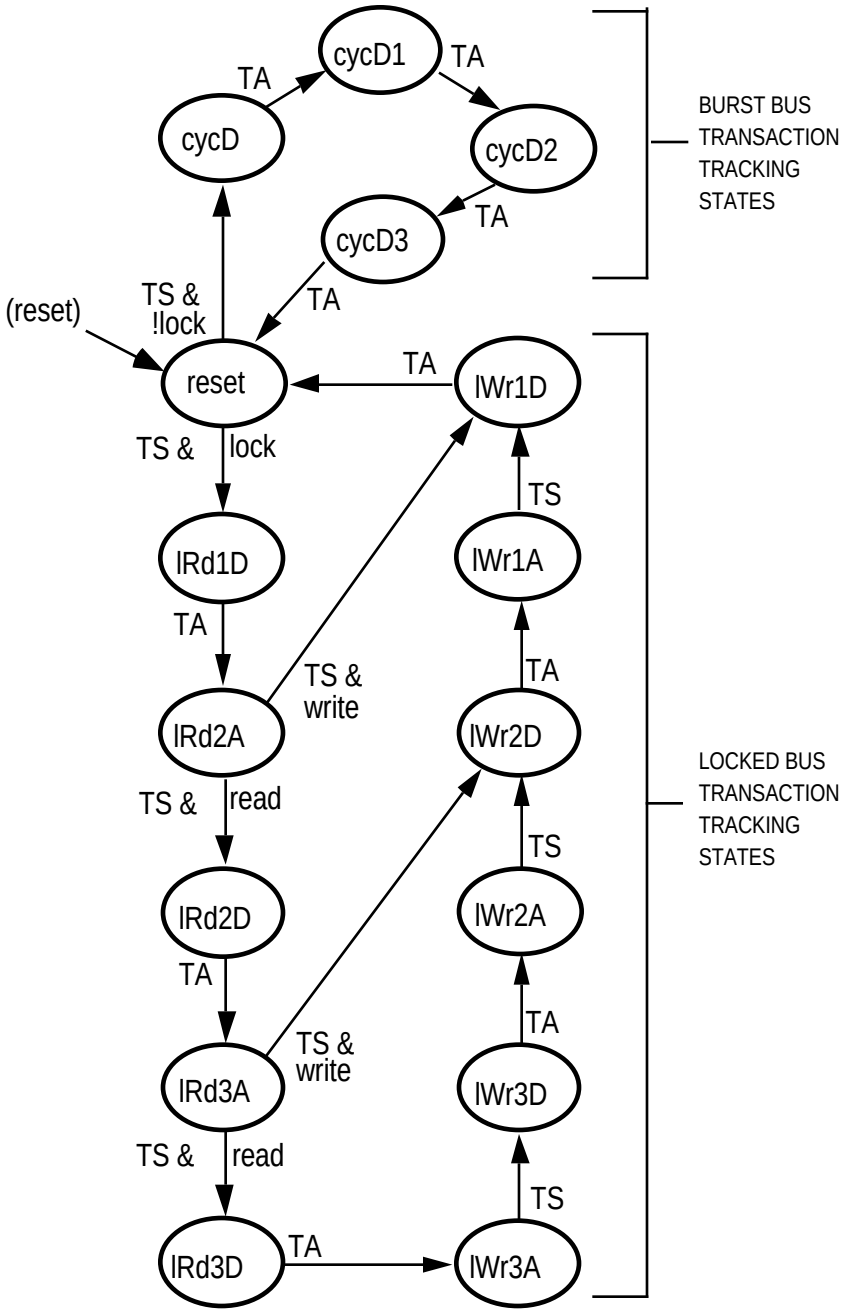


Figure A-7. Asynchronous $\overline{\text{TIP}}$ and $\overline{\text{LOCKE}}$ State Diagram

Note that the names inside the state ovals as shown in Figure A-7 are state names rather than output names as they are in Figure A-5.

A.6 SYNCHRONOUS $\overline{\text{TIP}}$ GENERATION

```
ABEL Source for Synchronous TIP Generation and Other Functions
module letip1ws
leTip1ws device ‘P22V10C’;

“ ‘letip1ws’, with the aid of a 32-bit-wide tristate latch (e.g., 74xx373),
“ converts MCF5102 bus protocol to regular ‘040 bus protocol.  This
       involves
“ four underlying tasks:
“ 1.  Recreate the address bus by telling the latches to hold contents of
“     the A/D bus right after the rising clock edge during which TS is
       asserted.
“ 2.  Tristate off the address lines when the processor loses grant and
“     completes any cycles that might be in progress at the time it loses
       grant
“     (including locked sequences).
“ 3.  Regenerate TIP.  This version of this PAL forces a dead-clock between
“     bus cycles as seen from the test card, or an extra wait-state as seen
“     from the MCF5102.  In this scenario, it also regenerates TS one clock
       later
“     than it comes out of the MCF5102 itself.  TIP and TS are generated as
       regis-
“     tered outputs from this PAL.
“ 4.  Regenerate LOCKE.  As with TIP, this version of this PAL can generate
“     LOCKE as a registered output.
“
“ Timing considerations:
“ TIP:  This PAL uses TIPx bidirectionally, other sources driving TIP must
       meet
“       PAL setups to the rising edge of BCLK.
“ General:  Because this version injects a dead-clock between bus cycles,
       it
“       operates as a synchronous machine, with clocked outputs.  Since the
       clock
“       to output on most PALs is faster than on most ‘040s, these
“
“ Note that this PAL program makes very little provison for bed-of-nails
       testing.
“ Note also that this version does not take inhibited bursts into account

“Inputs:  Note: active-high vs. active-low orientation is defined here -
       equations use assertion and negation
-------
bclk pin 2;    “‘040 BCLK
!tea pin 3;    “‘040 TEAx
!ta pin 4;     “‘040 TAx
!bg pin 5;     “(take a guess)
!rsti pin 7;
!tsIn pin 9;   “TS from the chip
!wr pin 10;    “‘040 R/W line (i.e. negative-logic write signal)
siz1 pin 11;
siz0 pin 12;
!lock pin 13;
“!tip pin 25;  TIP is used bidirectionally
```

```
“Outputs:
“-------
!tsOut pin 27 istype ‘neg,reg’;  “TS to the outside world
!locke pin 26 istype ‘neg,reg’; “regenerated ‘lock end’ signal
!tip pin 25 istype ‘neg,reg’;  “regenerated ‘Transfer In Progress’ signal
!aoe pin 18 istype ‘neg,reg’;   “output enable for address latches
ale pin 17 istype ‘buffer’;    “latch enable for the address latches

“Internally Used Pins:
“--------------------
sv3 pin 23 istype ‘pos,reg’;    “high-order internal state variable
sv2 pin 21 istype ‘pos,reg’;
sv1 pin 20 istype ‘pos,reg’;
sv0 pin 19 istype ‘pos,reg’;    “low-order internal state variable

“Unused:
“------
nc1 pin 6;
nc2 pin 16;
nc3 pin 24;

“Shorthand:
“---------
sv = [sv3, sv2, sv1, sv0]; “ D-inputs
svp = [sv3.fb, sv2.fb, sv1.fb, sv0.fb];  “ Q-outputs
siz = [siz1, siz0];
c, x, z = .C., .X., .Z.;

“Signal Value Names for Test Vectors (easier to read vectors):
“-----------------------------------
“Inputs--------------
“bclk - just use 0, 1, and c.
“!tea”   ber      = 0;
“!ta”    ack      = 0;
“!bg”    grt      = 0;
“!rsti”  rst, run = 0, 1;
“!ts”    tGo      = 0;
“!wr”    red, wrt = 1, 0;
“!lock”  lck      = 0;
“Outputs--------------
“!tip”   iPg, ded = 0, 1;
“!locke” lkE      = 0;
“!aoe”   dvA      = 0;
“ale”    tra, lch = 1, 0;
“tHold”  hld      = 1;

“State Names:
“-----------
reset = ^b0000;      “inactive bus
lkRd0 = ^b0001;      “locked read cycle 1
lkRd1 = ^b0011;      “locked read cycle 2
lkRd2 = ^b0010;      “locked read cycle 3
lkRd3 = ^b0110;      “locked read cycle 4
lkRd4 = ^b0111;      “locked read cycle 5
```

```
lkRd5 = ^b0101;      “locked write cycle 5
lkWr5 = ^b0100;      “locked write cycle 4
lkWr4 = ^b1100;      “locked write cycle 3
lkWr3 = ^b1101;      “locked write cycle 2
lkWr2 = ^b1111;      “locked write cycle 1
lkWr1 = ^b1110;      “locked write cycle 0
brst0 = ^b1010;      “burst read or write cycle, L.W. addr. 1
brst1 = ^b1011;      “burst read or write cycle, L.W. addr. 2
brst2 = ^b1001;      “burst read or write cycle, L.W. addr. 3
“ This state machine makes state transitions only on cycle ends.  During a
      CAS with
“ aligned operands, it will be in ‘reset’ during the read, and ‘lkRd1’
      during the
“ write cycle.  At the end of the write cycle, it will transition back to
      reset.
“ Always being in the state named after what the previous bus cycle was
      about sounds
“ a odd at first, but it’s the best approach since:
“ 1.  This PAL chooses behavior based on the previous and current bus cycle
      types and
“ 2.  The processor’s cycle-status lines tell the type of the current
      cycle.
“ 3.  It makes the subsequent data transfers of burst transactions easier
      to work with,
“     since they have a cycle end but no cycle start.
“ This makes provision for 6 reads and writes, but CAS2 on odd address
      won’t always work anyway.

“Transfer Size Codes:
lword = ^b00;
byte  = ^b01;
word  = ^b10;
burst = ^b11;  isBurst = (siz == burst);


Equations “Note that equations are written in terms of assertion and
      negation
“          See pin definitions for active level definitions.

sv3.oe = 1;     sv2.oe = 1;     sv1.oe = 1;     sv0.oe = 1;
sv3.ar = rsti;  sv2.ar = rsti;  sv1.ar = rsti;  sv0.ar = rsti;
sv3.clk = bclk; sv2.clk = bclk; sv1.clk = bclk; sv0.clk = bclk;

tsOut.oe = !rsti & aoe;  “enable TS when we’re not reset and address is
      enabled
tsOut.clk = bclk;
tsOut := !rsti & tsIn & bg;  “need one-clock delay to allow address to
      setup to next BCLK edge

tip.oe = !rsti & aoe;  “enable TIP when we’re not reset and address is
      enabled
tip.clk = bclk;
tip := !rsti & (  tsIn & bg  “assert on TS
               # (siz==burst) & (svp!=brst2) & ta & !tea
                 “hold active through state-transition to ‘brst0’
```

```
                # (svp==brst0) # (svp==brst1) “hold through second two bus
        cycles of a burst
                # (svp==brst2) & !ta & !tea “hold through all but final
        clock of final bus cycle
                # (!tsIn & !ta & !tea) & tip “hold when no cycle start or
        end
               );

locke.oe = !rsti & aoe;  “same idea as TIP
locke.clk = bclk;
locke :=   !rsti & lock
         & (  (svp==lkRd0) & lock & wr & tsIn
                    “assert after a locked read if this cycle is write
            # (svp==lkWr1) & lock & wr & tsIn “assert after next-to-last
        locked write
             # (!tsIn & !ta & !tea) & locke & ((svp==lkRd0) # (svp==lkWr1))
                    “maintain previous state when no cycle start or end
            );

ale.oe = 1;  “always drive address latch enable and address output enable
ale = !rsti & tsIn & bg & !bclk;
      “this equation tells when the latch is transparent, not when it is
       latching

aoe.oe = 1;
aoe.clk = bclk;
aoe := !rsti & (  !tip & bg  “reflect state of BG if no cycle in progress
                # tip & aoe  “maintain previous state while cycle in
        progress
               );


state_diagram sv  “(See note after state definitions)

state reset:  “inactive bus or completing first cycle
        case
           (!ta & tea):              reset;  “BERR - abort
           (ta & tea):               reset;  “retry - stay put
           (!ta & !tea):             reset;  “keep waiting for a cycle to
        complete
           (ta & !tea & lock & !wr): lkRd0;  “remember that we just did a
        locked read
           (ta & !tea & lock & wr):  reset;  “can’t theoretically happen
           (ta & !tea & !lock & isBurst):  brst0;  “just did first cycle of
        burst
           (ta & !tea & !lock & !isBurst): reset;  “just completed an
        individual bus cycle
        endcase;
state lkRd0:  “completed locked read cycle 0
        case
           (!ta & tea):              reset;  “BERR - abort
           (ta & tea):               lkRd0;  “retry - stay put
           (!ta & !tea):             lkRd0;  “keep waiting for cycle to
        complete
```

```
          (ta & !tea & !lock):      reset;  “processor aborted locked
       transaction
          (ta & !tea & lock & !wr): lkRd1;  “rack up another read
          (ta & !tea & lock & wr):  reset;  “locked sequence done
       endcase;
    state lkRd1:  “completed locked read cycle 1
       case
          (!ta & tea):              reset;  “BERR - abort
          (ta & tea):               lkRd1;  “retry - stay put
          (!ta & !tea):             lkRd1;  “keep waiting for cycle to
       complete
          (ta & !tea & !lock):      reset;  “processor aborted locked
       transaction
          (ta & !tea & lock & !wr): lkRd2;  “rack up another read
          (ta & !tea & lock & wr):  lkWr1;  “pretend like we just finished
       write #1
       endcase;
    state lkRd2:  “completed locked read cycle 2
       case
          (!ta & tea):              reset;  “BERR - abort
          (ta & tea):               lkRd2;  “retry - stay put
          (!ta & !tea):             lkRd2;  “keep waiting for cycle to
       complete
          (ta & !tea & !lock):      reset;  “processor aborted locked
       transaction
          (ta & !tea & lock & !wr): lkRd3;  “rack up another read
          (ta & !tea & lock & wr):  lkWr2;  “pretend like we just finished
       write #2
       endcase;
    state lkRd3:  “completed locked read cycle 3
       case
          (!ta & tea):              reset;  “BERR - abort
          (ta & tea):               lkRd3;  “retry - stay put
          (!ta & !tea):             lkRd3;  “keep waiting for cycle to
       complete
          (ta & !tea & !lock):      reset;  “processor aborted locked
       transaction
          (ta & !tea & lock & !wr): lkRd4;  “rack up another read
          (ta & !tea & lock & wr):  lkWr3;  “pretend like we just finished
       write #3
       endcase;
    state lkRd4:  “completed locked read cycle 4
       case
          (!ta & tea):              reset;  “BERR - abort
          (ta & tea):               lkRd4;  “retry - stay put
          (!ta & !tea):             lkRd4;  “keep waiting for cycle to
       complete
          (ta & !tea & !lock):      reset;  “processor aborted locked
       transaction
          (ta & !tea & lock & !wr): lkRd5;  “rack up another read
          (ta & !tea & lock & wr):  lkWr4;  “pretend like we just finished
       write #4
       endcase;
    state lkRd5:  “completed locked read cycle 5
       case
```

```
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkRd5;  “retry - stay put
            (!ta & !tea):             lkRd5;  “keep waiting for cycle to
         complete
            (ta & !tea & !lock):      reset;  “processor aborted locked
         transaction
            (ta & !tea & lock & !wr): reset;  “can’t theoretically happen
            (ta & !tea & lock & wr):  lkWr5;  “note that we’re done with all
         reads
         endcase;
   state lkWr5:  “completed locked write cycle 5
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkWr5;  “retry - stay put
            (!ta & !tea):             lkWr5;  “keep waiting for cycle to
         complete
            (ta & !tea & !wr):        reset;  “processor is confused - start
         over
            (ta & !tea & wr & !lock): reset;  “processor aborted locked
         transaction
            (ta & !tea & wr & lock):  lkWr4;  “count down another write
         endcase;
   state lkWr4:  “completed locked write cycle 4
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkWr4;  “retry - stay put
            (!ta & !tea):             lkWr4;  “keep waiting for cycle to
         complete
            (ta & !tea & !wr):        reset;  “processor is confused - start
         over
            (ta & !tea & wr & !lock): reset;  “processor aborted locked
         transaction
            (ta & !tea & wr & lock):  lkWr3;  “count down another write
         endcase;
   state lkWr3:  “completed locked write cycle 3
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkWr3;  “retry - stay put
            (!ta & !tea):             lkWr3;  “keep waiting for cycle to
         complete
            (ta & !tea & !wr):        reset;  “processor is confused - start
         over
            (ta & !tea & wr & !lock): reset;  “processor aborted locked
         transaction
            (ta & !tea & wr & lock):  lkWr2;  “count down another write
         endcase;
   state lkWr2:  “completed locked write cycle 2
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkWr2;  “retry - stay put
            (!ta & !tea):             lkWr2;  “keep waiting for cycle to
         complete
            (ta & !tea & !wr):        reset;  “processor is confused - start
         over
```

```
            (ta & !tea & wr & !lock): reset;  “processor aborted locked
         transaction
            (ta & !tea & wr & lock):  lkWr1;  “count down another write
         endcase;
   state lkWr1:  “completed locked write cycle 1
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               lkWr1;  “retry - stay put
            (!ta & !tea):             lkWr1;  “keep waiting for cycle to
         complete
            (ta & !tea & !wr):        reset;  “processor is confused - start
         over
            (ta & !tea & wr & !lock): reset;  “processor aborted locked
         transaction
            (ta & !tea & wr & lock):  reset;  “locked transaction completed
         endcase;
   state brst0:  “completed burst read or write cycle, L.W. addr. 0
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               reset;  “retry - abort
            (!ta & !tea):             brst0;  “keep waiting for cycle to
         complete
            (ta & !tea):              brst1;  “move on to next cycle in burst
         endcase;
   state brst1:  “completed burst read or write cycle, L.W. addr. 1
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               reset;  “retry - abort
            (!ta & !tea):             brst1;  “keep waiting for cycle to
         complete
            (ta & !tea):              brst2;  “move on to next cycle in burst
         endcase;
   state brst2:  “completed burst read or write cycle, L.W. addr. 2
         case
            (!ta & tea):              reset;  “BERR - abort
            (ta & tea):               reset;  “retry - abort
            (!ta & !tea):             brst2;  “keep waiting for cycle to
         complete
            (ta & !tea):              reset;  “burst complete
         endcase;


   test_vectors
   ([bclk, !rsti, !tsIn, !tip, !wr, siz, !lock, !tea, !ta, !bg]->
    [!tsOut, !tip, !locke, !aoe, ale, sv])

   “ Make sure that it stays reset:
   [c,rst, x,  1,  x,  x,    x,    x,   x,   x  ]->[z,   z,   z,  !dvA,x,
         reset]; “1
   [c,rst, x,  1,  x,  x,    x,    x,   x,   x  ]->[z,   z,   z,  !dvA,x,
         reset]; “2
   [c,run, tGo,1,  x,  x,    x,    x,   x,  !grt]->[z,   z,   z,  !dvA,x,
         reset]; “3
   [c,run,!tGo,1,  x,  x,    x,    x,   x,   grt]->[!tGo,ded,!lkE, dvA,x,
         reset]; “4
```

```
" Take bus away in middle of a single cycle:
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "5
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "6
[0,run, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,tra,reset]; "7
[1,run, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,lch,reset]; "8
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "9
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "10
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "11
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "12
[0,run,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "13
[c,run,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "14
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; "15

" Take bus away in middle of a burst cycle:
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "16
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "17
[c,run, tGo,z,  red,burst,!lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,reset]; "18
[c,run,!tGo,z,  red,burst,!lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; "19
[c,run,!tGo,z,  red,burst,!lck,!ber, ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst0]; "20
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst0]; "21
[c,run,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst1]; "22
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst1]; "23
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst1]; "24
[c,run,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst2]; "25
[c,run,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[!tGo,iPg,!lkE,
       dvA,lch,brst2]; "26
[c,run,!tGo,iPg,red,lword,!lck,!ber, ack,!grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; "27
[c,run,!tGo,ded,red,burst,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; "28

" Alternate master runs single cycle; grant comes active during cycle:
```

```
[c,run,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “29
[c,run,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “30
[0,run, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “31
[c,run, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “32
[c,run,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “33
[c,run,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,  z,
       !dvA,lch,reset]; “34
“ (Vectors 35-37 fail JEDSIM because they don’t see the pull-up on TIP)
[c,run,!tGo,iPg,red,lword,!lck,!ber,!ack, grt]->[z,   z,  z,
       !dvA,lch,reset]; “35
[0,run,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,  z,
       !dvA,lch,reset]; “36
[c,run,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,  z,
       !dvA,lch,reset]; “37

“ Locked sequence for TAS with aligned operand - one read then one write:
[c,run,!tGo,ded,red,lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “38
[c,run,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “39
[c,run, tGo,z,  red,lword, lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,reset]; “40
[c,run,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; “41
[c,run,!tGo,z,  red,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd0]; “42
[c,run,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd0]; “43
[0,run, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,tra,lkRd0]; “44
[c,run, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[ tGo,iPg, lkE,
       dvA,tra,lkRd0]; “45
[c,run,!tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[!tGo,iPg, lkE,
       dvA,lch,lkRd0]; “46
[c,run,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “47

“ Locked sequence for CAS2 with aligned operands - two reads then two
       writes:
[c,run,!tGo,ded,red,lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “48
[c,run,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “49
[c,run, tGo,z,  red,lword, lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,reset]; “50
[c,run,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; “51
[c,run,!tGo,z,  red,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd0]; “52
```

```
[c,run, tGo,z,  red,lword, lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd0]; “53
[c,run,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd0]; “54
[c,run,!tGo,z,  red,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd1]; “55
[c,run, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd1]; “56
[c,run,!tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd1]; “57
[c,run,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr1]; “58
[c,run, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[ tGo,iPg, lkE,
       dvA,tra,lkWr1]; “59
[c,run,!tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[!tGo,iPg, lkE,
       dvA,lch,lkWr1]; “60
[c,run,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “61

“ Locked sequence for CAS2 with addr-1-aligned LW data - six reads then six
       writes:
[c,run,!tGo,ded,red,x,     lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “62
[c,run,!tGo,z,  red,x,     lck,!ber,!ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “63
[c,run, tGo,z,  red,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,reset]; “64
[c,run,!tGo,z,  red,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,reset]; “65
[c,run,!tGo,z,  red,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd0]; “66
[c,run, tGo,z,  red,word,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd0]; “67
[c,run,!tGo,z,  red,word,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd0]; “68
[c,run,!tGo,z,  red,word,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd1]; “69
[c,run, tGo,z,  red,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd1]; “70
[c,run,!tGo,z,  red,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd1]; “71
[c,run,!tGo,z,  red,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd2]; “72
[c,run, tGo,z,  red,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd2]; “73
[c,run,!tGo,z,  red,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd2]; “74
[c,run,!tGo,z,  red,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd3]; “75
[c,run, tGo,z,  red,word,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd3]; “76
[c,run,!tGo,z,  red,word,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd3]; “77
[c,run,!tGo,z,  red,word,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd4]; “78
```

```
[c,run, tGo,z,  red,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd4]; “79
[c,run,!tGo,z,  red,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd4]; “80
[c,run,!tGo,z,  red,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkRd5]; “81
[c,run, tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkRd5]; “82
[c,run,!tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkRd5]; “83
[c,run,!tGo,z,  wrt,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr5]; “84
[c,run, tGo,z,  wrt,word,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkWr5]; “85
[c,run,!tGo,z,  wrt,word,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkWr5]; “86
[c,run,!tGo,z,  wrt,word,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr4]; “87
[c,run, tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkWr4]; “88
[c,run,!tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkWr4]; “89
[c,run,!tGo,z,  wrt,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr3]; “90
[c,run, tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkWr3]; “91
[c,run,!tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkWr3]; “92
[c,run,!tGo,z,  wrt,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr2]; “93
[c,run, tGo,z,  wrt,word,  lck,!ber,!ack, grt]->[ tGo,iPg,!lkE,
       dvA,tra,lkWr2]; “94
[c,run,!tGo,z,  wrt,word,  lck,!ber,!ack, grt]->[!tGo,iPg,!lkE,
       dvA,lch,lkWr2]; “95
[c,run,!tGo,z,  wrt,word,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,lkWr1]; “96
[c,run, tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[ tGo,iPg, lkE,
       dvA,tra,lkWr1]; “97
[c,run,!tGo,z,  wrt,byte,  lck,!ber,!ack, grt]->[!tGo,iPg, lkE,
       dvA,lch,lkWr1]; “98
[c,run,!tGo,z,  wrt,byte,  lck,!ber, ack, grt]->[!tGo,ded,!lkE,
       dvA,lch,reset]; “99

end letip1ws;
```

A.7 ASYNCHRONOUS $\overline{\text{TIP}}$ GENERATION

```
ABEL Source for Asynchronous TIP Generation and Other Functions
module mod flag '-r3' "(ABEL 3 won't let you name module same as device)

retro0ws device 'P22V10';

" 'retro0ws', with the aid of a 32-bit-wide tristate latch (e.g., 74xx373),
" converts MCF5102 bus protocol to regular '040 bus protocol.  This
      involves
" four underlying tasks:
" 1.  Recreate the address bus by telling the latches to hold contents of
"     the A/D bus right after the rising clock edge during which TS is
      asserted.
" 2.  Tristate off the address lines when the processor loses grant and
"     completes any cycles that might be in progress at the time it loses
      grant
"     (including locked sequences).
" 3.  Regenerate TIP.  This signal has to be generated combinationally from
"     TS, so it inherently incurs a combinational propagation delay after
      TS
"     which the real '040 does not have.
" 4.  Regenerate LOCKE.  As with TIP, this signal must be generated
      combina-
"     tionally, triggered by LOCK and several other indicators.  The real
      '040
"     asserts LOCKE in the same timing spec as LOCK and R/W.
" Note that this part generates TIP and LOCKE asynchronously based partly
      on
" the state of a state machine.  To prevent glitches or unexpected
      deassertions,
" certain state transitions must be kept unidistant (only a single state
      variable
" changing during the state transition).  These are called out below the
      state-
" definitions; use caution in changing the state assignment.
" (Because the '040 bus is synchronous, TIP and LOCKE should only be
      evaluated on
" rising BCLK edges.  Nevertheless, preventing them from glitching is
      desirable.)
"
" Timing considerations:
" TIP:  'retro0ws' uses TIPx bidirectionally, other sources driving TIP
      must meet
"      PAL setups to the rising edge of BCLK.  When the PAL drives TIP,
      here are
"      its timings:
"      Assertion:  TIPx asserts 1 PAL tpd after TS asserts.  This
      increases
"                  spec.#14 from 35ns max to 40ns max with a 5ns PAL.
      That
"                  leaves 10ns setup time at 20MHz and 5ns setup time at
      25MHz.
"      Negation:   TIPx negates 1 clock-to-feedback plus one tpd after the
      rising
```

```
“                   BCLK edge on which TA or TEA is recognized.  This
      decreases
“                   spec.#12 from 11.5ns min to approximately 3ns min.
“ Address:  Addresses are latched in external latches, such as 74FCT573DT.
“           this part adds an additional 4ns max, 1.5ns min, to spec.#26.
      They
“           therefore go valid, at the latest, 1ns before the BCLK rising
      edge
“           that terminates the address phase.  This is 14ns max in
      addition to
“           the EC/LC spec.#11.  This is obviously a big board-design-level
“           difference, over 2/3s of which is due to how the MCF5102 works.
      Note
“           that the address latches are transparent only during the second
      half
“           (BCLK low) of the clock period during which TS is active.  This
      mimics
“           the 68K family’s behavior of holding the previous address on
      the bus
“           during periods of bus inactivity.
“ LOCKE:  As with TIP, LOCKE asserts 1 PAL tpd after TS asserts.  That
      increases
“         spec.#11 (for that signal) to 40ns max with a 5ns PAL.  That
      leaves
“         10ns setup time at 20MHz and 5ns setup time at 25MHz.  It negates
      1
“         clock-to-feedback plus one tpd after the rising BCLK edge in
      which
“         TA or TEA is asserted.  For a 5ns PAL, that produces a negation
      time
“         of 9ns max and approximately 4ns min.  The EC/LC spec.#11
      guarantees a
“         larger minimum hold time of 11.5ns.
“
“ Note that this PAL program makes very little provison for bed-of-nails
      testing.
“ This is ABEL 3 code.  ABEL 4 could not fit this PAL into 22V10.  This is
      a very dense PAL

“Inputs:  Note: active-high vs. active-low orientation is defined here -
      equations use assertion and negation
“------
“2”  bclk pin 1;    “‘040 BCLK
“3”  !tea pin 2;    “‘040 TEAx
“4”  !ta pin 3;     “‘040 TAx
“5”  !bg pin 4;     “(take a guess)
“6”  !bb pin 5;
“7”  !rsti pin 6;
“9”  !ts pin 7;
“10” !wr pin 8;      “MCF5102 R/W line (i.e. negative-logic write signal)
“11” siz1 pin 9;
“12” siz0 pin 10;
“13” !lock pin 11;
“16” !tbi pin 13;
“25” !tipIn pin 21;  “TIP is used bidirectionally in this PAL
```

```
     tipIn istype ‘feed_pin’;  “use pin value (that’s all a 22V10 will do)
“ tipIn is same as tipOut - two names are used to get around an ABEL quirk

“Outputs:
“-------
“26” !locke pin 22; “regenerated ‘lock end’ signal
“25” !tipOut pin 21;  “regenerated ‘Transfer In Progress’ signal
“18” ale pin 15;    “latch enable for the address latches
“17” !aoe pin 14;   “output enable for address latches
     tipOut istype ‘neg’;  locke istype ‘neg’;
     aoe istype ‘neg’;  ale istype ‘pos’;

“Internally Used Pins:
“--------------------
“27” tHold pin 23;  “holds TIP active from negation of TS to PAL state-
       transition
“24” sv4 pin 20;    “high-order internal state variable
“23” sv2 pin 19;
“21” sv1 pin 18;
“20” sv0 pin 17;    “low-order internal state variable
“19” sv3 pin 16;
     sv0 istype ‘pos,reg’;  sv1 istype ‘pos,reg’;  sv2 istype ‘pos,reg’;
     sv3 istype ‘pos,reg’;  sv4 istype ‘pos,reg’;

“Unused:  (none)
“------

“Shorthand:
“---------
sv = [sv4, sv3, sv2, sv1, sv0];  “ D-inputs
siz = [siz1, siz0, !tbi];  “ Q-outputs
c, x, z = .C., .X., .Z.;

“Signal Value Names for Test Vectors (easier to read without signal names):
“-----------------------------------
“Inputs--------------
“bclk - just use 0, 1, and c.
“!tea”   ber      = 0;
“!ta”    ack      = 0;
“!bg”    grt      = 0;
“!bb”    bsy      = 0;
“!rsti”  rst, run = 0, 1;
“!ts”    tGo      = 0;
“!wr”    red, wrt = 1, 0;
“!lock”  lck      = 0;
“!tbi”   bst, inh = 1, 0;
“Outputs--------------
“!tip”   iPg, ded = 0, 1;
“!locke” lkE      = 0;
“!aoe”   dvA      = 0;
“ale”    tra, lch = 1, 0;
“tHold”  hld      = 1;

“State Names:
“-----------
```

```
reset = ^b00000;      “have bus;  waiting for a bus cycle to begin
lRd1D = ^b00001;      “locked read cycle 1, data phase (addr phase occurs
       in reset)
lRd2A = ^b00010;      “lock still active, so read cycle 2, address phase
lRd2D = ^b01001;      “locked read cycle 2, data phase
lRd3A = ^b01000;      “lock still active, so read cycle 3, address phase
lRd3D = ^b01011;      “locked read cycle 3, data phase
lWr3A = ^b11000;      “locked write cycle 3, address phase
lWr3D = ^b10001;      “locked write cycle 3, data phase
lWr2A = ^b10000;      “locked write cycle 2, address phase
lWr2D = ^b00101;      “locked write cycle 2, data phase
lWr1A = ^b00111;      “locked write cycle 1, address phase
lWr1D = ^b00011;      “locked write cycle 1, data phase
cycD  = ^b10100;      “unlocked read or write cycle, data phase
cycD1 = ^b10101;      “burst read or write cycle, data phase L.W. addr. 1
cycD2 = ^b10111;      “burst read or write cycle, data phase L.W. addr. 2
cycD3 = ^b10110;      “burst read or write cycle, data phase L.W. addr. 3

“ The following state transitions must be unidistant:
“ lRd2A -> lWr1D:  LOCKE asserts on falling edge of R/W during lRd2A;
       should
“                  stay asserted through transition.
“ lWr1A -> lWr1D:  LOCKE is asserted by detection of these two states;
       should
“                  stay asserted through transition.
“ cycD -> cycD1 -> cycD2 -> cycD3:  tHold prevents TIP from negating
       between
“                  any address phase and the following data phase, but not
“                  during transitions between data phases.
“ lWr2D -> lWr1A:  Desirable but not required:  LOCKE is asserted by
       detection
“                  of the above-mentioned states;  should prevent any
       momentary
“                  false assertions before it asserts for real.
“ See also the state-assignment dependency described under the equation for
       TIPx.

“Transfer Size Codes:
lword = ^b001;
byte  = ^b011;
word  = ^b101;
burst = ^b111;
bsInh = ^b110;

Equations “All equations are written in terms of assertion and negation;
“          see pin definitions for active-levels.

sv4.oe = 1;     sv3.oe = 1;     sv2.oe = 1;     sv1.oe = 1;     sv0.oe = 1;
sv4.ar = rsti;  sv3.ar = rsti;  sv2.ar = rsti;  sv1.ar = rsti;  sv0.ar =
       rsti;
“ Note that 22V10 has only 1 .AR for all output macrocells, but it’s worth
       specifying
“ all just in case this ever migrates to a different PLD type.
```

```
tipOut.oe = !rsti & aoe;  “enable TIP when we’re not reset and address is
       enabled
tipOut =   !rsti                          “force inactive during reset
         & (   ts & aoe                    “assert with ts
             # tHold                       “hold up to data phase
             # sv0 & (sv!=lWr1A)           “hold through any locked data
       phase
             # (sv==cycD) # (sv==cycD1)    “or through burst cycle data
       phase
             # (sv==cycD2) # (sv==cycD3)
           );  “TIPx will negate in clk->out + internal feedback time
“ This equation, notably the term ‘sv0 & (sv!=lWr1A)’, is state-assignment-
“ dependent.  Its purpose is to call out all of the data-phase states.
“ Unfortunately, ORing up all of them blew ABEL’s mind (too complex an
“ expression).  There are some complex interdependencies between tipOut.oe
       and aoe,
“ including the requirement of an external PU.

tHold.oe = 1;
tHold =   !rsti  “force inactive during reset
        & (   ts & aoe  “assert with TS
            # tHold & !ta & !tea       “retain until TA asserts and
       renegates
          );

locke.oe = !rsti & aoe;  “same idea as TIP
locke =   !rsti   “deactivate on reset
        & (   (sv==lRd2A) & (tHold # ts) & wr
            # (sv==lWr1A) & (tHold # ts)
            # (sv==lWr1D)
          );

ale.oe = 1;  “always drive address latch enable and address output enable
ale =    ts & aoe & !bclk
        & (   (sv==reset) # (sv==lRd2A) # (sv==lRd3A)
            # (sv==lWr3A) # (sv==lWr2A) # (sv==lWr1A)
          );

“Note:  this equation tells when the latch is transparent, not when it is
       latching

aoe.oe = 1;
aoe :=   !bb & bg  “reflect state of BG while bus not in use
       # bb & aoe;  “maintain previous state while bus in use



state_diagram sv

state reset:   “waiting for first TSx - initial address phase
       if !(ts & aoe) then
          reset            “dead-time between cycles
       else if (lock & !wr & (siz!=burst)) then
          lRd1D            “proceed to locked sequence handling - rack up
       one read
```

```
       else cycD;          “single memory cycle, burst memory cycle, or CPU
       space access

state lRd1D:   “locked read cycle 1, data phase (addr phase occurs in
       reset)
       case
          (ta & tea):   reset;  “retry
          (ta & !tea):  lRd2A;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lRd1D;  “wait-state
       endcase;

state lRd2A:   “lock still active, so read cycle 2, address phase
       if !ts then
          lRd2A                “dead-time between cycles
       else if wr then
          lWr1D                “count down 1 write
       else lRd2D;             “rack up a 2nd locked read
       “Note: if wr asserts during this state, LOCKE will assert
       combinationally

state lRd2D:   “locked read cycle 2, data phase
       case
          (ta & tea):   lRd2A;  “retry
          (ta & !tea):  lRd3A;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lRd2D;  “wait-state
       endcase;

state lRd3A:   “lock still active, so read cycle 3, address phase
       if !ts then
          lRd3A                “dead-time between cycles
       else if wr then
          lWr2D                “count down 2 writes
       else lRd3D;             “rack up a 3rd locked read

state lRd3D:   “locked read cycle 3, data phase
       case
          (ta & tea):   lRd3A;  “retry
          (ta & !tea):  lWr3A;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lRd3D;  “wait-state
       endcase;
state lWr3A:   “locked write cycle 3, address phase
       if !ts then
          lWr3A                “dead-time between cycles
       else if !wr then
          reset                “still trying to read;  possibly misaligned
       CAS2
                               “from reset it will run as if no lock
       else
          lWr3D;               “go to data phase of write cycle 3

state lWr3D:   “locked write cycle 3, data phase
       case
```

```
          (ta & tea):   lWr3A;  “retry
          (ta & !tea):  lWr2A;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lWr3D;  “wait-state
       endcase;

state lWr2A:   “locked write cycle 2, address phase
       if !ts then
          lWr2A               “dead-time between cycles
       else lWr2D;            “go to data phase of write cycle 2

state lWr2D:   “locked write cycle 2, data phase
       case
          (ta & tea):   lWr2A;  “retry
          (ta & !tea):  lWr1A;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lWr2D;  “wait-state
       endcase;

state lWr1A:   “locked write cycle 1, address phase
       if !ts then
          lWr1A               “dead-time between cycles
       else lWr1D;            “go to data phase of write cycle 1

state lWr1D:   “locked write cycle 1, data phase
       case
          (ta & tea):   lWr1A;  “retry
          (ta & !tea):  reset;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): lWr1D;  “wait-state
       endcase;

state cycD:    “unlocked read or write cycle, data phase
       case
          (ta & tea):       reset;  “retry
          (   ta & !tea
             & siz==burst ): cycD1;  “proceed to next transfer of burst
          (   ta & !tea
             & siz!=burst ): reset;  “cycle done
          (!ta & tea):      reset;  “bus error
          (!ta & !tea):     cycD;   “wait-state
       endcase;

state cycD1:   “burst read or write cycle, data phase L.W. addr. 1
       case
          (ta & tea):   reset;  “retry
          (ta & !tea):  cycD2;  “proceed to next address phase
          (!ta & tea):  reset;  “bus error
          (!ta & !tea): cycD1;  “wait-state
       endcase;

state cycD2:   “burst read or write cycle, data phase L.W. addr. 2
       case
          (ta & tea):   reset;  “retry
          (ta & !tea):  cycD3;  “proceed to next address phase
```

```
         (!ta & tea):  reset;  “bus error
         (!ta & !tea): cycD2;  “wait-state
      endcase;

state cycD3:   “burst read or write cycle, data phase L.W. addr. 3
      if ta
         then reset  “retry or operation all done - back to reset either
      case
         else if tea
            then reset  “bus error
            else cycD3;  “wait-state


test_vectors
([bclk,!rsti,!bb,!ts,!tipIn,!wr,siz,!lock,!tea,!ta,!bg]->
 [!tipOut,!locke,!aoe,ale,tHold,sv])

“ Make sure that it stays reset:
[c,rst, x,   x,  x,  x,  x,    x,    x,   x,   x  ]->[z,   z,  !dvA,x,
      !hld,reset]; “1
[c,rst,!bsy, x,  x,  x,  x,    x,    x,   x,   x  ]->[z,   z,  !dvA,x,
      !hld,reset]; “2
[c,run,!bsy, tGo,x,  x,  x,    x,    x,   x,  !grt]->[z,   z,  !dvA,x,
      !hld,reset]; “3
[c,run,!bsy,!tGo,z,  x,  x,    x,    x,   x,   grt]->[ded,!lkE, dvA,x,
      !hld,reset]; “4

“ Take bus away in middle of a single cycle:
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
      dvA,lch,!hld,reset]; “5
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
      dvA,lch,!hld,reset]; “6
[0,run, bsy, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,tra,
      hld,reset]; “7
[1,run, bsy, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
      hld,cycD ]; “8
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
      hld,cycD ]; “9
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
      hld,cycD ]; “10
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[iPg,!lkE, dvA,lch,
      hld,cycD ]; “11
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[iPg,!lkE, dvA,lch,
      hld,cycD ]; “12
[0,run, bsy,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[iPg,!lkE,
      dvA,lch,!hld,cycD ]; “13
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[ded,!lkE,
      dvA,lch,!hld,reset]; “14
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[z,   z,
      !dvA,lch,!hld,reset]; “15

“ Take bus away in middle of a burst cycle:
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
      dvA,lch,!hld,reset]; “16
```

```
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “17
[c,run, bsy, tGo,z,  red,burst,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “18
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “19
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack, grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD1]; “20
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD1]; “21
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD2]; “22
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD2]; “23
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD2]; “24
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD3]; “25
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
       dvA,lch,!hld,cycD3]; “26
[c,run, bsy,!tGo,iPg,red,burst,!lck,!ber, ack,!grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “27
[c,run,!bsy,!tGo,iPg,red,burst,!lck,!ber,!ack,!grt]->[z,  z,
       !dvA,lch,!hld,reset]; “28

“ Alternate master runs single cycle; grant comes active during cycle:
[c,run,!bsy,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “29
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “30
[0,run, bsy, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “31
[c,run, bsy, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “32
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “33
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
       !dvA,lch,!hld,reset]; “34
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack, grt]->[z,   z,
       !dvA,lch,!hld,reset]; “35
[0,run, bsy,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,
       !dvA,lch,!hld,reset]; “36
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,
       !dvA,lch,!hld,reset]; “37

“ Locked sequence for TAS or CAS - one read then one write:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[z,   z,
       !dvA,lch,!hld,reset]; “38
[c,run,!bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “39
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd1D]; “40
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd1D]; “41
```

```
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “42
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “43
[0,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,tra,
       hld,lRd2A]; “44
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
       hld,lWr1D]; “45
[c,run,!bsy,!tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
       hld,lWr1D]; “46
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “47

“ Locked sequence for TAS or CAS: 2-aligned operand:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “48
[c,run,!bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “49
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd1D]; “50
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd1D]; “51
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “52
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “53
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd2D]; “54
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd2D]; “55
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd3A]; “56
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd3A]; “57
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lWr2D]; “58
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lWr1A]; “59
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
       hld,lWr1D]; “60
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “61

“ Locked sequence for TAS or CAS: 1-aligned operand:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “62
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd1D]; “63
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “64
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd2A]; “65
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd2D]; “66
```

```
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd3A]; “67
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lRd3A]; “68
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lRd3D]; “69
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lWr3A]; “70
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lWr3A]; “71
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lWr3D]; “72
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lWr2A]; “73
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,lWr2D]; “74
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,lWr1A]; “75
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
       hld,lWr1D]; “76
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “78

“ Make sure that PAL treats burst-inhibited transaction as four separate
       cycles:
[c,run,!bsy,!tGo,ded,x,  bsInh,!lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “79
[c,run,!bsy,!tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “80
[c,run, bsy, tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “81
[c,run, bsy,!tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “82
[c,run, bsy,!tGo,z,  x,  bsInh,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “83
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “84
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “85
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “86
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “87
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “88
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “89
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “90
[c,run,!bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
       hld,cycD ]; “91
[c,run,!bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
       dvA,lch,!hld,reset]; “92

end mod;
```

APPENDIX B
MCF5102 EVALUATION SOCKET

The MCF5102 Evaluation Socket is designed as a drop-in replacement for a MC68EC040 running at 20/25 MHz for evaluation of the MCF5102 in specific applications. Designers may plug the Evaluation Socket directly into a existing MC68EC040 PGA socket of a target system or Motorola's MC68EC040 Integrated Development Platform.

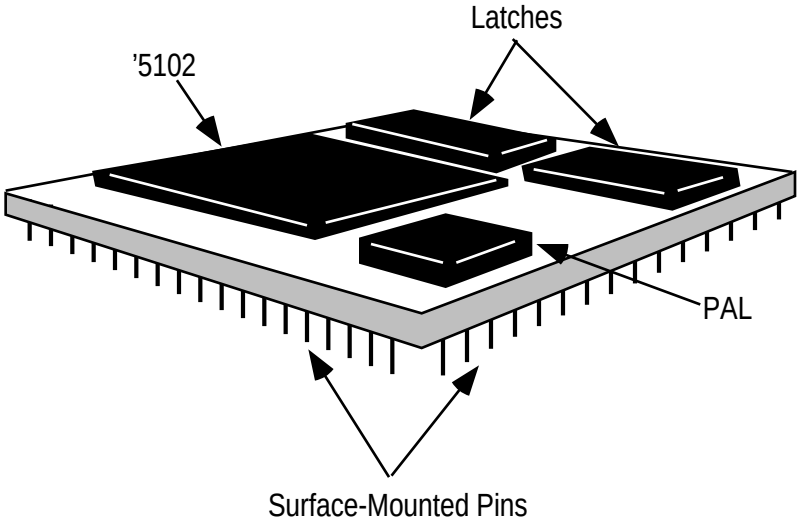


Figure B-1. MCF5102 Evaluation Socket

B.1 SCOPE

The following paragraphs describes the differences between the MCF5102 elavuation socket and the M68EC040, important considerations, hardware timing specifications, PAL equations to generate M68040 bus protocol, $\overline{\text{TIP}}$ and $\overline{\text{LOCKE}}$.

The Evaluation Socket demultiplexes the address and data bus in addition to generating $\overline{\text{TIP}}$ and $\overline{\text{LOCKE}}$. All componets, including the MCF5102, are directly soldered to a printed circuit board that maps the MCF5102 directly to an M68EC040 PGA footprint. A linear regulator generates the 3.3 volts required by the MCF5102.

NOTE

Evaluation Sockets are functional, however they have not been tested over temperature or other stress conditions. The evaluation socket is not intended to be used in end products. The voltage regulator supplies only the MCF5102 and its decoupling capacitors. The remaining circuitry operates at 5V as supplied at the pins of the PGA footprint.

B.2 DOCUMENTATION

M68040 User's Manual M68040UM/AD
M68000 Family Programmers Reference Manual M68000PM/AD
The 68k Source, BR729/D
3.3 Volt Logic And Interface Circuits BR1335/D

B.3 IMPORTANT CONSIDERATIONS:

1. It is recommend plugging the evaluation socket board into a zero-insertion-force (ZIF) socket, but most industry-standard sockets are acceptable. It is not recommend soldering it directly to your board.
2. If you need to extract the evaluation socket from a non-ZIF socket we recommend using an extraction tool specifically designed for removing pin-grid-array (PGA) components. In this scenario, be certain to pull from the white pin-carrier board, which is designed to withstand demating force. Pulling on the upper (green) board will place undue stress on the solder joints by which the pin-carrier board is attached.
3. Frequency of operation is less than or equal to 25MHz.
4. The evaluation socket does not support the following MC68EC040 features:
   a. Bus snooping. Ignores SC1 and SC0 from the target board. MI* is driven from the MCF5102's MI line to mimic its normally-asserted behavior while it is not in control of its bus.
   b. UPA signals. No-connects.
   c. The CAS2 instruction. CAS2 does not execute correctly, but CAS and TAS instructions behave as specified, with one exception: Do not cache the destination (memory) operand of a CAS instruction.
   d. JTAG.
5. MC68EC040 characteristics that are slightly different.
   a. The evaluation socket assumes that the target board removes read data from the data bus in response to the end of the read cycle. The target should not drive read data between bus cycles or, at the beginning of the following bus cycle. The MCF5102's address/data lines are connected to the target board's data lines.
   b. The evaluation socket inserts a dead-clock ($\overline{\text{TIP}}$ high) between every bus cycle.
   c. Bus timings are slightly changed.

d. The evaluation socket places 10K ohm pull-ups to 5V on the $\overline{\text{LOCKE}}$, $\overline{\text{TIP}}$, and $\overline{\text{TS}}$ signals.

e. Specified V_{OL} and V_{OH} values are the same, but because most outputs are driven from a 3.3V part, typical levels will be lower. This may have an effect on noise margin.

f. $\overline{\text{LOCKE}}$ and $\overline{\text{TIP}}$ have about 3 times the MC68EC040's 5mA DC drive strength, and a correspondingly lower output impedance. This is important with regard to transmission-line-, and undershoot-related signal quality issues on the target board.

g. Address lines have between 10 and 36 times the drive strength of the MC68EC040's specified 5mA DC drive strength. They are driven through balanced drivers with edge-rate control.

h. BCLK trace short were kept a a minimum, however it could not be kept as short as that inside the package of the MC68EC040 it replaces. The evaluation socket hardware was implementated by placeing a stub on that transmission line approximately 0.9 inches in length, including the length of that PGA pin, but excluding lengths of bond-wires inside the ICs. The target board's termination scheme may need to be adjusted to minimize this effect. The evaluation socket has pads for pull-up and pull-down termination.

i. Power consumption will differ.

6. $\overline{\text{SCD}}$ (System Clock Disable) signal is not accessible however the LPSTOP instruction can be executed.

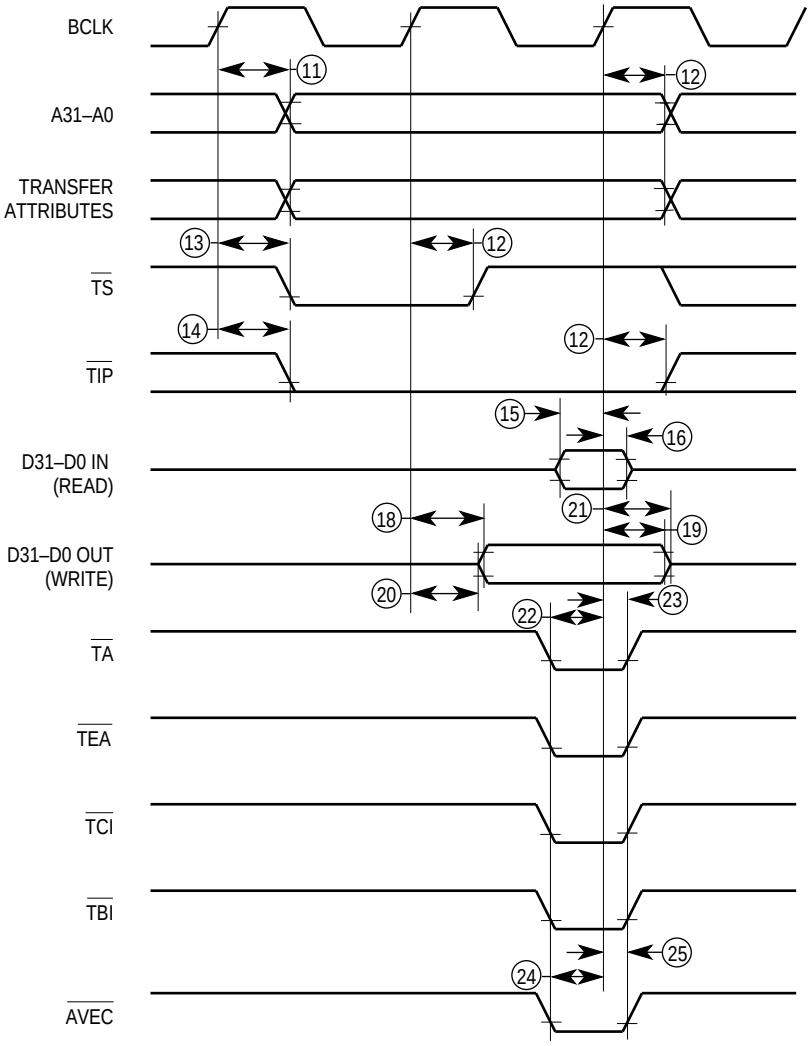
B.4 OUTPUT AC TIMING SPECIFICATIONS (See Figures B-2 to B-4)

Num	Characteristic	20 MHz		25 MHz		Unit
		Min	Max	Min	Max	
11[1]	BCLK to Address, $\overline{\text{CIOUT}}$, $\overline{\text{LOCK}}$, PSTx, R/$\overline{\text{W}}$, SIZx, TLNx,TMx, TTx Valid	11.5	35	9	30	ns
11a[2]	BCLK to Address	23	40.5	19	34.5	ns
12[2]	BCLK to Output Invalid (Output Hold)	11.5	—	9	—	ns
13	BCLK to $\overline{\text{TS}}$ Valid	11.5	35	9	30	ns
14	BCLK to $\overline{\text{TIP}}$ Valid	11.5	35	9	30	ns
18	BCLK to Data-Out Valid	11.5	37	9	32	ns
19	BCLK to Data-Out Invalid (Output Hold)	11.5	—	9	—	ns
20[2]	BCLK to Output Low Impedance	11.5	—	9	—	ns
21	BCLK to Data-Out High Impedance	11.5	25	9	20	ns
38	BCLK to Address, $\overline{\text{CIOUT}}$, $\overline{\text{LOCK}}$, R/$\overline{\text{W}}$, SIZx, $\overline{\text{TS}}$, TLNx, TMx, TTx High Impedance	11.5	23	9	18	ns
39	BCLK to $\overline{\text{BB}}$, $\overline{\text{TA}}$, $\overline{\text{TIP}}$ High Impedance	23	33	19	28	ns
40	BCLK to $\overline{\text{BR}}$, $\overline{\text{BB}}$ Valid	11.5	35	9	30	ns
43	BCLK to $\overline{\text{MI}}$ Valid	11.5	35	9	30	ns
48	BCLK to $\overline{\text{TA}}$ Valid	11.5	35	9	30	ns
50	BCLK to $\overline{\text{IPEND}}$, PSTx, $\overline{\text{RSTO}}$ Valid	11.5	35	9	30	ns

Notes 1. These signals are generated in response to the previous rising edge of BCLK, and are therefore valid the specified time before the indicated BCLK rising edge.
2. This timing is really given by 3ns min, 10.5ns max, from the falling edge of BCLK following the rising edge shown in the timing diagram.

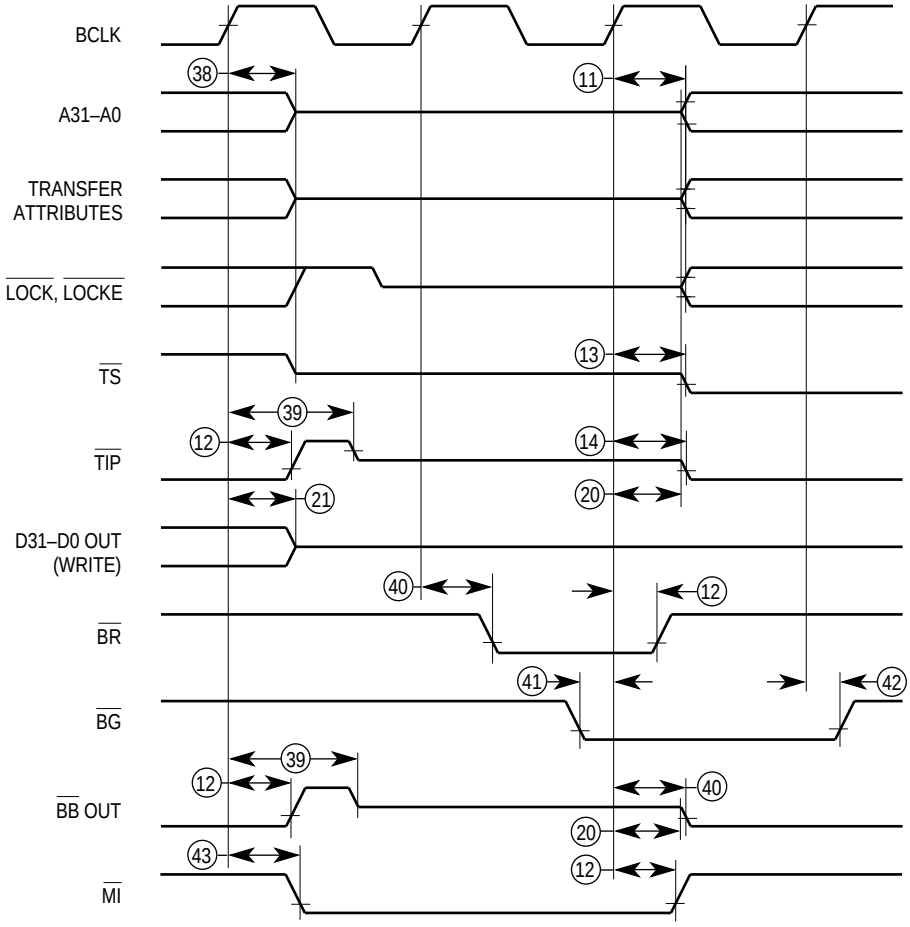
Num	Characteristic	20 MHz		25 MHz		Unit
		Min	Max	Min	Max	
15	Data-In Valid to BCLK (Setup)	6	—	5	—	ns
16[3]	BCLK to Data-In Invalid (Hold)	5	—	4	—	ns
17[4,5]	BCLK to Data-In High Impedance (Read Followed by Write)	—	61	—	49	ns
22a	$\overline{TA}$ Valid to BCLK (Setup)	12.5	—	10	—	ns
22b	$\overline{TEA}$ Valid to BCLK (Setup)	12.5	—	10	—	ns
22c	$\overline{TCI}$ Valid to BCLK (Setup)	12.5	—	10	—	ns
22d	$\overline{TBI}$ Valid to BCLK (Setup)	14	—	11	—	ns
23	BCLK to $\overline{TA}$, $\overline{TEA}$, $\overline{TCI}$, $\overline{TBI}$ Invalid (Hold)	2.5	—	2	—	ns
24	$\overline{AVEC}$ Valid to BCLK (Setup)	6	—	5	—	ns
25	BCLK to $\overline{AVEC}$ Invalid (Hold)	2.5	—	2	—	ns
41a	$\overline{BB}$ Valid to BCLK (Setup)	8	—	7	—	ns
41b	$\overline{BG}$ Valid to BCLK (Setup)	10	—	8	—	ns
41c	$\overline{CDIS}$ Valid to BCLK (Setup)	12.5	—	10	—	ns
41d	$\overline{IPLx}$ Valid to BCLK (Setup)	5	—	4	—	ns
42	BCLK to $\overline{BB}$, $\overline{BG}$, $\overline{CDIS}$, $\overline{IPLx}$ Invalid (Hold)	2.5	—	2	—	ns
44a[6]	Address Valid to BCLK (Setup)	10	—	8	—	ns
44b[6]	SIZx Valid to BCLK (Setup)	15	—	12	—	ns
44c[6]	TTx Valid to BCLK (Setup)	7.5	—	6	—	ns
44d[6]	R/$\overline{W}$ Valid to BCLK (Setup)	7.7	—	6	—	ns
44e[6]	SCx Valid to BCLK (Setup)	12.5	—	10	—	ns
45[6]	BCLK to Address SIZx, TTx, R/$\overline{W}$, SCx Invalid (Hold)	2.5	—	2	—	ns
46[6]	$\overline{TS}$ Valid to BCLK (Setup)	6	—	5	—	ns
47[6]	BCLK to $\overline{TS}$ Invalid (Hold)	2.5	—	2	—	ns
49[6]	BCLK to $\overline{BB}$ High Impedance (MCF5102 Assumes Bus Mastership)	—	11	—	9	ns
51	$\overline{RSTI}$ Valid to BCLK	6	—	5	—	ns
52	BCLK to $\overline{RSTI}$ Invalid	2.5	—	2	—	ns

Notes
3. The evaluation socket assumes that read data will be removed from the data bus in response to the end of the read cycle. The target board must not drive a default bus slave between cycles, or any device during the addressing phase.
4. The MCF5102 drives address onto the data pins during the first BCLK cycle of the bus cycle. On a write cycle, the data pins will already be low impedance.
5. The MCf5102 drives address onto the data bus as early as spec 16 (max) when any bus cycle (not just writes) directly follows a read (or write).
6. The evaluation socket does not support bus snooping.



NOTE: Transfer Attribute Signals = UPAx, SIZx, TTx, TMx, TLNx, R/$\overline{W}$, $\overline{LOCK}$, $\overline{LOCKE}$, $\overline{CIOUT}$

Figure B-2. Read/Write Timing



NOTE: Transfer Attribute Signals = UPAx, SIZx, TTx, TMx, TLNx, R/$\overline{W}$, $\overline{CIOUT}$

Figure B-3. Arbitration Timing

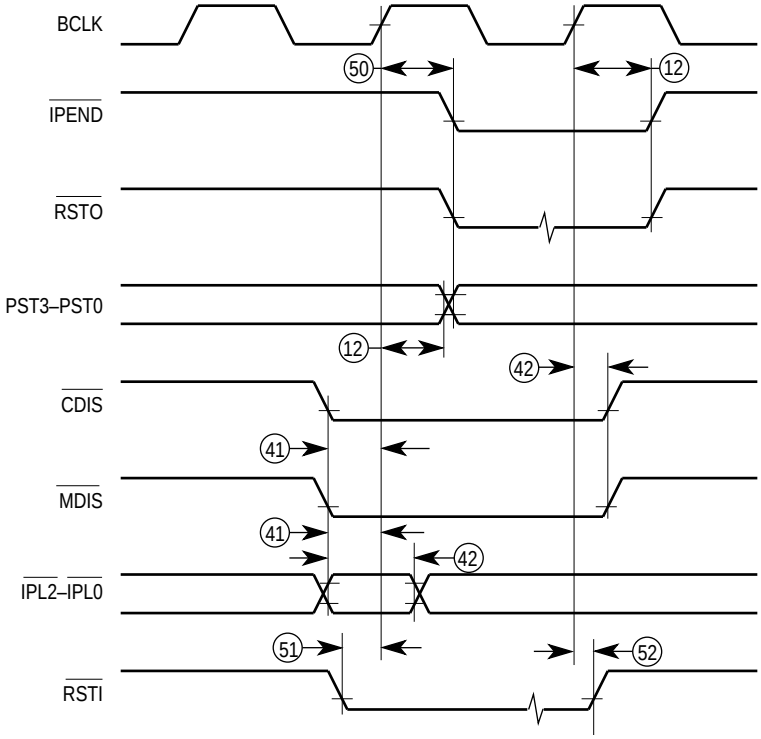


Figure B-4. Other Timing

B.5 PAL CODING

```
module mod flag '-r3'

retro0ws device 'P22V10';

" 'retro0ws', with the aid of a 32-bit-wide tristate latch (e.g., 74xx373),
       " converts MCF5102 bus protocol to regular 68EC040 bus protocol.
       This involves " four tasks:
" 1.  Recreate the address bus by telling the latches to hold contents of
"     the A/D bus right after the rising clock edge during which TS is
       asserted.
" 2.  Tristate off the address lines when the processor loses grant and
"     completes any cycles that might be in progress at the time it loses
       grant
"     (including locked sequences).
" 3.  Regenerate TIP.  This signal has to be generated combinationally from
"     TS, so it inherently incurs a combinational propagation delay after TS
       "which the 68EC040 does not have.
" 4.  Regenerate LOCKE.  This signal must be generated combina-
"     tionally, triggered by LOCK and several indicators.  The 68EC040
"     asserts LOCKE in the same timing spec as LOCK and R/W.
" Note that this part generates TIP and LOCKE asynchronously based partly on
       the state of a state machine. To prevent glitches or unexpected
       deassertions,certain state transitions must be kept unidistant (only a
       single state variable changing during the state transition).  These
       are called out below the state-definitions; use caution in changing
       the state assignment.(Because the 68EC040 bus is synchronous, TIP and
       LOCKE should only be evaluated on rising BCLK edges.  Nevertheless,
       preventing them from glitching is desirable.)
"
" Timing considerations:
" TIP:  'retro0ws' uses TIPx bidirectionally, other sources driving TIP must
       meet PAL setups to the rising edge of BCLK.  When the PAL drives TIP,
       here are its timings:
" Assertion:  TIPx asserts 1 PAL tpd after TS asserts.  This increases
" spec.#14 from 35ns max to 40ns max with a 5ns PAL.  That
" leaves 10ns setup time at 20MHz and 5ns setup time at 25MHz.
"       Negation:   TIPx negates 1 clock-to-feedback plus one tpd after the
 rising
"               BCLK edge on which TA or TEA is recognized.  This decreases
"                   spec.#12 from 11.5ns min to approximately 3ns min.
" Address:  Addresses are latched in external latches, specifically
       74FCT573DT.
"          this part adds an additional 4ns max, 1.5ns min, to spec.#26.
"     They therefore go valid, at the latest, 1ns before the BCLK rising
       edge that terminates the address phase.  This is 14ns max in addition
       to
"          the EC/LC spec.#11.  This is a board-design-level
"          difference, over 2/3s of which is due to how the MCF5102 works.
       Note
"       that the address latches are transparent only during the second half
"     (BCLK low) of the clock period during which TS is active.  This mimics
"       the 68K family's behavior of holding the previous address on the bus
"          during periods of bus inactivity.
```

```
" LOCKE:  As with TIP, LOCKE asserts 1 PAL tpd after TS asserts.  That
 increases
"         spec.#11 (for that signal) to 40ns max with a 5ns PAL.  That leaves
"         10ns setup time at 20MHz and 5ns setup time at 25MHz.  It negates 1
"         clock-to-feedback plus one tpd after the rising BCLK edge in which
"         TA or TEA is asserted.  For a 5ns PAL, that produces a negation
       time
"         of 9ns max and approximately 4ns min.  The EC/LC spec.#11
       guarantees
 a
"         larger minimum hold time of 11.5ns.

"Inputs:
"------
"2"  bclk pin 1;    "68EC040 BCLK
"3"  !tea pin 2;    "68EC040 TEAx
"4"  !ta pin 3;     "68EC040 TAx
"5"  !bg pin 4;     "(take a guess)
"6"  !bb pin 5;
"7"  !rsti pin 6;
"9"  !ts pin 7;
"10" !wr pin 8;      "68EC040 R/W line (i.e. negative-logic write signal)
"11" siz1 pin 9;
"12" siz0 pin 10;
"13" !lock pin 11;
"16" !tbi pin 13;
"25" !tipIn pin 21;  "TIP is used bidirectionally in this PAL
     tipIn istype 'feed_pin';  "use pin value (that's all a 22V10 will do)

"Outputs:
"-------
"26" !locke pin 22; "regenerated 'lock end' signal
"25" !tipOut pin 21;  "regenerated 'Transfer In Progress' signal
"18" ale pin 15;    "latch enable for the address latches
"17" !aoe pin 14;   "output enable for address latches
     tipOut istype 'neg';  locke istype 'neg';
     aoe istype 'neg';  ale istype 'pos';

"Internally Used Pins:
"--------------------
"27" tHold pin 23;  "holds TIP active from negation of TS to PAL
 state-transition
"24" sv4 pin 20;    "high-order internal state variable
"23" sv2 pin 19;
"21" sv1 pin 18;
"20" sv0 pin 17;    "low-order internal state variable
"19" sv3 pin 16;
     sv0 istype 'pos,reg';  sv1 istype 'pos,reg';  sv2 istype 'pos,reg';
     sv3 istype 'pos,reg';  sv4 istype 'pos,reg';

"Unused:  (none)
"------

"Shorthand:
"---------
```

```
sv = [sv4, sv3, sv2, sv1, sv0];
siz = [siz1, siz0, !tbi];
c, x, z = .C., .X., .Z.;

"Signal Value Names for Test Vectors:
"-----------------------------------
"Inputs--------------
"bclk - just use 0, 1, and c.
"!tea"   ber      = 0;
"!ta"    ack      = 0;
"!bg"    grt      = 0;
"!bb"    bsy      = 0;
"!rsti"  rst, run = 0, 1;
"!ts"    tGo      = 0;
"!wr"    red, wrt = 1, 0;
"!lock"  lck      = 0;
"!tbi"   bst, inh = 1, 0;
"Outputs-------------
"!tip"   iPg, ded = 0, 1;
"!locke" lkE      = 0;
"!aoe"   dvA      = 0;
"ale"    tra, lch = 1, 0;
"tHold"  hld      = 1;

"State Names:
"-----------
reset = ^b00000;      "have bus;  waiting for a bus cycle to begin
lRd1D = ^b00001;      "locked read cycle 1, data phase (addr phase occurs
       in
 reset)
lRd2A = ^b00010;      "lock still active, so read cycle 2, address phase
lRd2D = ^b01001;      "locked read cycle 2, data phase
lRd3A = ^b01000;      "lock still active, so read cycle 3, address phase
lRd3D = ^b01011;      "locked read cycle 3, data phase
lWr3A = ^b11000;      "locked write cycle 3, address phase
lWr3D = ^b10001;      "locked write cycle 3, data phase
lWr2A = ^b10000;      "locked write cycle 2, address phase
lWr2D = ^b00101;      "locked write cycle 2, data phase
lWr1A = ^b00111;      "locked write cycle 1, address phase
lWr1D = ^b00011;      "locked write cycle 1, data phase
cycD  = ^b10100;      "unlocked read or write cycle, data phase
cycD1 = ^b10101;      "burst read or write cycle, data phase L.W. addr. 1
cycD2 = ^b10111;      "burst read or write cycle, data phase L.W. addr. 2
cycD3 = ^b10110;      "burst read or write cycle, data phase L.W. addr. 3

" The following state transitions must be unidistant:
" lRd2A -> lWr1D:  LOCKE asserts on falling edge of R/W during lRd2A;
       should
"                  stay asserted through transition.
" lWr1A -> lWr1D:  LOCKE is asserted by detection of these two states;
       should
"                  stay asserted through transition.
" cycD -> cycD1 -> cycD2 -> cycD3:  tHold prevents TIP from negating
       between
"                  any address phase and the following data phase, but not
```

```
"                  during transitions between data phases.
" lWr2D -> lWr1A:  Desirable but not required:  LOCKE is asserted by
       detection
"                  of the above-mentioned states;  should prevent any
       momentary

"                  false assertions before it asserts for real.
" See also the state-assignment dependency described under the equation for
 TIPx.

"Transfer Size Codes:
lword = ^b001;
byte  = ^b011;
word  = ^b101;
burst = ^b111;
bsInh = ^b110;

Equations

sv4.oe = 1;     sv3.oe = 1;     sv2.oe = 1;     sv1.oe = 1;     sv0.oe = 1;
sv4.ar = rsti;  sv3.ar = rsti;  sv2.ar = rsti;  sv1.ar = rsti;  sv0.ar =
       rsti;

tipOut.oe = !rsti & aoe;  "enable TIP when we're not reset and address is
 enabled
tipOut =   !rsti                          "force inactive during reset
         & (   ts & aoe                    "assert with ts
             # tHold                       "hold up to data phase
             # sv0 & (sv!=lWr1A)           "hold through any locked data
       phase
             # (sv==cycD) # (sv==cycD1)    "or through burst cycle data
       phase
             # (sv==cycD2) # (sv==cycD3)
           );  "TIPx will negate in clk->out + internal feedback time
" This equation, notably the term 'sv0 & (sv!=lWr1A)', is state-assignment-
" dependent.  Its purpose is to call out all of the data-phase states.
" Unfortunately, ORing up all of them blew ABEL's mind (too complex an
" expression).

tHold.oe = 1;  "enable TIP when we're not reset and address is enabled
tHold =   !rsti  "force inactive during reset
        & (   ts & aoe  "assert with TS
            # tHold & !ta & !tea       "retain until TA asserts and
       renegates
          );

locke.oe = !rsti & aoe;  "same idea as TIP
locke =   !rsti   "deactivate on reset
        & (   (sv==lRd2A) & (tHold # ts) & wr
            # (sv==lWr1A) & (tHold # ts)
            # (sv==lWr1D)
          );

ale.oe = 1;  "always drive address latch enable and address output enable
ale.ar = rsti;  "reset it to latched (doesn't really matter, but make it
```

```
 predictable)
ale =    ts & aoe & !bclk
        & (   (sv==reset) # (sv==lRd2A) # (sv==lRd3A)
            # (sv==lWr3A) # (sv==lWr2A) # (sv==lWr1A)
          );

"Note:  this equation tells when the latch is transparent, not when it is
 latching

aoe.oe = 1;
aoe :=   !bb & bg  "reflect state of BG while bus not in use
       # bb & aoe;  "maintain previous state while bus in use



state_diagram sv

state reset:   "waiting for first TSx - initial address phase
       if !(ts & aoe) then
          reset            "dead-time between cycles
       else if (lock & !wr & (siz!=burst)) then
          lRd1D            "proceed to locked sequence handling - rack up
       one read
       else cycD;          "single memory cycle, burst memory cycle, or CPU
       space
 access

state lRd1D:   "locked read cycle 1, data phase (addr phase occurs in
       reset)
       case
          (ta & tea):   reset;  "retry
          (ta & !tea):  lRd2A;  "proceed to next address phase
          (!ta & tea):  reset;  "bus error
          (!ta & !tea): lRd1D;  "wait-state
       endcase;

state lRd2A:   "lock still active, so read cycle 2, address phase
       if !ts then
          lRd2A                "dead-time between cycles
       else if wr then
          lWr1D                "count down 1 write
       else lRd2D;             "rack up a 2nd locked read
       "Note: if wr asserts during this state, LOCKE will assert
       combinationally

state lRd2D:   "locked read cycle 2, data phase
       case
          (ta & tea):   lRd2A;  "retry
          (ta & !tea):  lRd3A;  "proceed to next address phase
          (!ta & tea):  reset;  "bus error
          (!ta & !tea): lRd2D;  "wait-state
       endcase;

state lRd3A:   "lock still active, so read cycle 3, address phase
       if !ts then
```

```
            lRd3A                "dead-time between cycles
         else if wr then
            lWr2D                "count down 2 writes
         else lRd3D;             "rack up a 3rd locked read

   state lRd3D:   "locked read cycle 3, data phase
         case
            (ta & tea):   lRd3A;  "retry
            (ta & !tea):  lWr3A;  "proceed to next address phase
            (!ta & tea):  reset;  "bus error
            (!ta & !tea): lRd3D;  "wait-state
         endcase;
   state lWr3A:   "locked write cycle 3, address phase
         if !ts then
            lWr3A                "dead-time between cycles
         else if !wr then
            reset                "still trying to read;  possibly misaligned
         CAS2
                                 "from reset it will run as if no lock
         else
            lWr3D;               "go to data phase of write cycle 3

   state lWr3D:   "locked write cycle 3, data phase
         case
            (ta & tea):   lWr3A;  "retry
            (ta & !tea):  lWr2A;  "proceed to next address phase
            (!ta & tea):  reset;  "bus error
            (!ta & !tea): lWr3D;  "wait-state
         endcase;

   state lWr2A:   "locked write cycle 2, address phase
         if !ts then
            lWr2A                "dead-time between cycles
         else lWr2D;             "go to data phase of write cycle 2

   state lWr2D:   "locked write cycle 2, data phase
         case
            (ta & tea):   lWr2A;  "retry
            (ta & !tea):  lWr1A;  "proceed to next address phase
            (!ta & tea):  reset;  "bus error
            (!ta & !tea): lWr2D;  "wait-state
         endcase;

   state lWr1A:   "locked write cycle 1, address phase
         if !ts then
            lWr1A                "dead-time between cycles
         else lWr1D;             "go to data phase of write cycle 1

   state lWr1D:   "locked write cycle 1, data phase
         case
            (ta & tea):   lWr1A;  "retry
            (ta & !tea):  reset;  "proceed to next address phase
            (!ta & tea):  reset;  "bus error
            (!ta & !tea): lWr1D;  "wait-state
         endcase;
```

```
state cycD:    "unlocked read or write cycle, data phase
       case
          (ta & tea):       reset;  "retry
          (   ta & !tea
             & siz==burst ): cycD1;  "proceed to next transfer of burst
          (   ta & !tea
             & siz!=burst ): reset;  "cycle done
          (!ta & tea):      reset;  "bus error
          (!ta & !tea):     cycD;   "wait-state
       endcase;

state cycD1:   "burst read or write cycle, data phase L.W. addr. 1
       case
          (ta & tea):   reset;  "retry
          (ta & !tea):  cycD2;  "proceed to next address phase
          (!ta & tea):  reset;  "bus error
          (!ta & !tea): cycD1;  "wait-state
       endcase;

state cycD2:   "burst read or write cycle, data phase L.W. addr. 2
       case
          (ta & tea):   reset;  "retry
          (ta & !tea):  cycD3;  "proceed to next address phase
          (!ta & tea):  reset;  "bus error
          (!ta & !tea): cycD2;  "wait-state
       endcase;

state cycD3:   "burst read or write cycle, data phase L.W. addr. 3
       if ta
          then reset  "retry or operation all done - back to reset either
       case
          else if tea
             then reset  "bus error
             else cycD3;  "wait-state


test_vectors
([bclk,!rsti,!bb,!ts,!tipIn,!wr,siz,!lock,!tea,!ta,!bg]->
 [!tipOut,!locke,!aoe,ale,tHold,sv])

" Make sure that it stays reset:
[c,rst, x,   x,  x,  x,  x,    x,    x,   x,   x  ]->[z,   z,  !dvA,x,
 !hld,reset]; "1
[c,rst,!bsy, x,  x,  x,  x,    x,    x,   x,   x  ]->[z,   z,  !dvA,x,
 !hld,reset]; "2
[c,run,!bsy, tGo,x,  x,  x,    x,    x,   x,  !grt]->[z,   z,  !dvA,x,
 !hld,reset]; "3
[c,run,!bsy,!tGo,z,  x,  x,    x,    x,   x,   grt]->[ded,!lkE, dvA,x,
 !hld,reset]; "4

" Take bus away in middle of a single cycle:
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "5
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
```

```
 dvA,lch,!hld,reset]; "6
[0,run, bsy, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,tra,
 hld,reset]; "7
[1,run, bsy, tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "8
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "9
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "10
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "11
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "12
[0,run, bsy,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD ]; "13
[c,run, bsy,!tGo,z,  red,lword,!lck,!ber, ack,!grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "14
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "15

" Take bus away in middle of a burst cycle:
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "16
[c,run,!bsy,!tGo,z,  red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "17
[c,run, bsy, tGo,z,  red,burst,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "18
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "19
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack, grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD1]; "20
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD1]; "21
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD2]; "22
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD2]; "23
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD2]; "24
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber, ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD3]; "25
[c,run, bsy,!tGo,z,  red,burst,!lck,!ber,!ack,!grt]->[iPg,!lkE,
 dvA,lch,!hld,cycD3]; "26
[c,run, bsy,!tGo,iPg,red,burst,!lck,!ber, ack,!grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "27
[c,run,!bsy,!tGo,iPg,red,burst,!lck,!ber,!ack,!grt]->[z,  z,
 !dvA,lch,!hld,reset]; "28

" Alternate master runs single cycle; grant comes active during cycle:
[c,run,!bsy,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "29
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "30
[0,run, bsy, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "31
```

```
[c,run, bsy, tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "32
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "33
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack,!grt]->[z,   z,
 !dvA,lch,!hld,reset]; "34
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber,!ack, grt]->[z,   z,
 !dvA,lch,!hld,reset]; "35
[0,run, bsy,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,
 !dvA,lch,!hld,reset]; "36
[c,run, bsy,!tGo,iPg,red,lword,!lck,!ber, ack, grt]->[z,   z,
 !dvA,lch,!hld,reset]; "37

" Locked sequence for TAS or CAS - one read then one write:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[z,   z,
 !dvA,lch,!hld,reset]; "38
[c,run,!bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "39
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd1D]; "40
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd1D]; "41
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "42
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "43
[0,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,tra,
 hld,lRd2A]; "44
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
 hld,lWr1D]; "45
[c,run,!bsy,!tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
 hld,lWr1D]; "46
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "47

" Locked sequence for TAS or CAS: 2-aligned operand:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "48
[c,run,!bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "49
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd1D]; "50
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd1D]; "51
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "52
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "53
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd2D]; "54
[c,run, bsy,!tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd2D]; "55
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd3A]; "56
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
```

```
 dvA,lch,!hld,lRd3A]; "57
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lWr2D]; "58
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lWr1A]; "59
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
 hld,lWr1D]; "60
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "61

" Locked sequence for TAS or CAS: 1-aligned operand:
[c,run, bsy,!tGo,ded,red,lword,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "62
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd1D]; "63
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "64
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd2A]; "65
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd2D]; "66
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd3A]; "67
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lRd3A]; "68
[c,run, bsy, tGo,z,  red,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lRd3D]; "69
[c,run, bsy,!tGo,z,  red,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lWr3A]; "70
[c,run, bsy,!tGo,z,  x,  lword, lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lWr3A]; "71
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lWr3D]; "72
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lWr2A]; "73
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,lWr2D]; "74
[c,run, bsy,!tGo,z,  wrt,lword, lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,lWr1A]; "75
[c,run, bsy, tGo,z,  wrt,lword, lck,!ber,!ack, grt]->[iPg, lkE, dvA,lch,
 hld,lWr1D]; "76
[c,run,!bsy,!tGo,z,  wrt,lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "78

" Make sure that PAL treats burst-inhibited transaction as four separate
 cycles:
[c,run,!bsy,!tGo,ded,x,  bsInh,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "79
[c,run,!bsy,!tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "80
[c,run, bsy, tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "81
[c,run, bsy,!tGo,z,  x,  bsInh,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "82
[c,run, bsy,!tGo,z,  x,  bsInh,!lck,!ber, ack, grt]->[ded,!lkE,
```

```
 dvA,lch,!hld,reset]; "83
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "84
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "85
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "86
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "87
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "88
[c,run, bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "89
[c,run, bsy, tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "90
[c,run,!bsy,!tGo,z,  x,  lword,!lck,!ber,!ack, grt]->[iPg,!lkE, dvA,lch,
 hld,cycD ]; "91
[c,run,!bsy,!tGo,z,  x,  lword,!lck,!ber, ack, grt]->[ded,!lkE,
 dvA,lch,!hld,reset]; "92

end mod;
```

INDEX

— **W** —

— **X** —