Chapter 4
Address Generation Unit

This chapter describes the architecture and the operation of the address generation unit (AGU). The address generation unit is the block where all address calculations are performed. It contains two arithmetic units—a modulo arithmetic unit for complex address calculations and an incrementer/decrementer for simple calculations. The modulo arithmetic unit can be used to calculate addresses in a modulo fashion, automatically wrapping around when necessary. A set of pointer registers, special-purpose registers, and multiple buses within the unit allow up to two address updates or a memory transfer to or from the AGU in a single cycle.

The capabilities of the address generation unit include the following operations:

- Provide one address to X data memory on the XAB1 bus
- Post-update an address after providing the original address value on XAB1 bus
- Calculate an effective address which is then provided on the XAB1 bus
- Provide two addresses to X data memory on the XAB1 and XAB2 buses and post-update both addresses
- Provide one address to program memory for program memory data accesses and post-update the address
- Increment or decrement a counter during normalization operations
- Provide a conditional register move (Tcc instruction)

Note that in the cases where the address generation unit is generating one or two addresses to access X data memory, the program controller generates a second or third address used to concurrently fetch the next instruction.

The AGU provides many different addressing modes, which include the following:

- Indirect addressing with no update
- Indirect addressing with post-increment
- Indirect addressing with post-decrement
- Indirect addressing with post-update by a register
- Indirect addressing with index by a 16-bit offset
- Indirect addressing with index by a 6-bit offset
- Indirect addressing with index by a register
- Immediate data
- Immediate short data
- Absolute addressing
- Absolute short addressing
- Peripheral short addressing
- Register direct
- Implicit

This chapter covers the architecture and programming model of the address generation unit, its addressing modes, and a discussion of the linear and modulo arithmetic capabilities of this unit. It concludes with a discussion of pipeline dependencies related to the address generation unit.

4.1 Architecture and Programming Model

The major components of the address generation unit are as follows:

- Four address registers (R0-R3)
- A stack pointer register (SP)
- An offset register (N)
- A modifier register (M01)
- A modulo arithmetic unit
- An incrementer/decrementer unit

The AGU uses integer arithmetic to perform the effective address calculations necessary to address data operands in memory. The AGU also contains the registers used to generate the addresses. It implements linear and modulo arithmetic and operates in parallel with other chip resources to minimize address-generation overhead.

Two ALUs are present within the AGU: the modulo arithmetic unit and the incrementer/decrementer unit. The two arithmetic units can generate up to two 16-bit addresses and two address updates every instruction cycle: one for XAB1 and one for XAB2 for instructions performing two parallel memory reads. The AGU can directly address 65,536 locations on XAB1 and 65,536 locations on the PAB. The AGU can directly address up to 65,536 locations on XAB2, but can only generate addresses to on-chip memory. The two ALUs work with the data memory to access up to two locations and provide two operands to the data ALU in a single cycle. The primary operand is addressed with the XAB1, and the second operand is addressed with the XAB2. The data memory, in turn, places its data on the core global data bus (CGDB) and the second external data bus (XDB2), respectively (see Figure 4-1 on page 4-3). See Section 6.1, “Introduction to Moves and Parallel Moves,” on page 6-1 for more discussion on parallel memory moves.

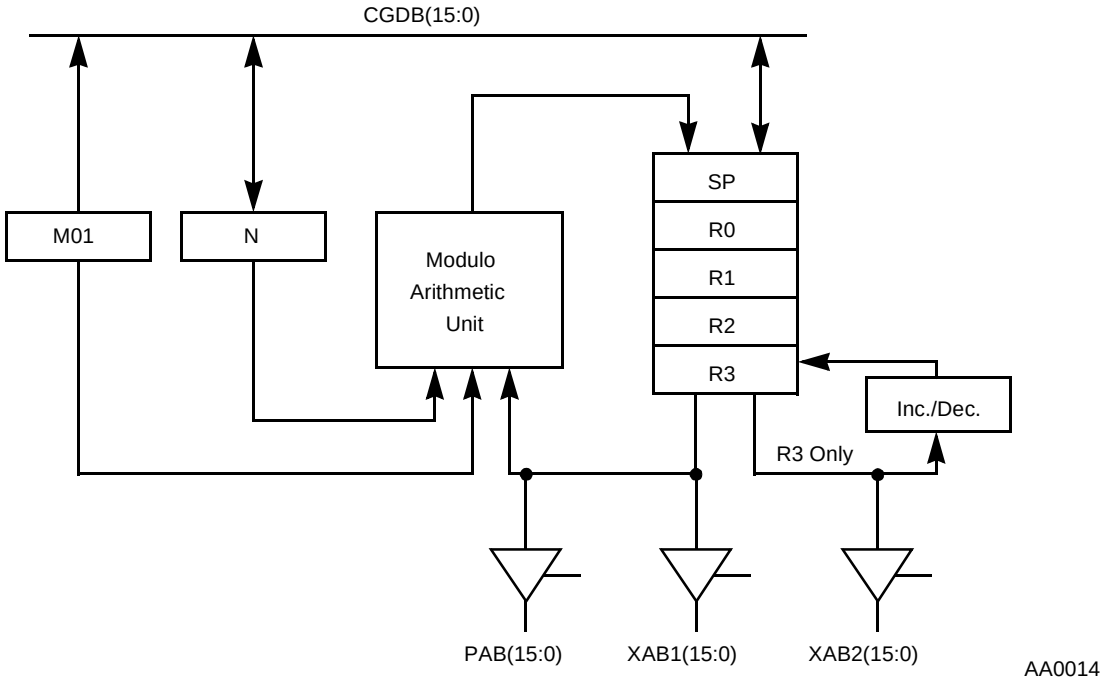


Figure 4-1. Address Generation Unit Block Diagram

All four address pointer registers and the SP are used in generating addresses in the register indirect addressing modes. The offset register can be used by all four address pointer registers and the SP, whereas the modulo register can be used by the R0 or by both the R0 and R1 pointer registers.

Whereas all the address pointer registers and the SP can be used in many addressing modes, there are some instructions that only work with a specific address pointer register. These cases are presented in Table 4-5 on page 4-9.

The address generation unit is connected to four major buses: CGDB, XAB1, XAB2, and PAB. The CGDB is used to read or write any of the address generation unit registers. The XAB1 and XAB2 provide a primary and secondary address, respectively, to the X data memory, and the PAB provides the address when accessing the program memory.

A block diagram of the address generation unit is shown in Figure 4-1, and its corresponding programming model is shown in Figure 4-2. The blocks and registers are explained in the following subsections.

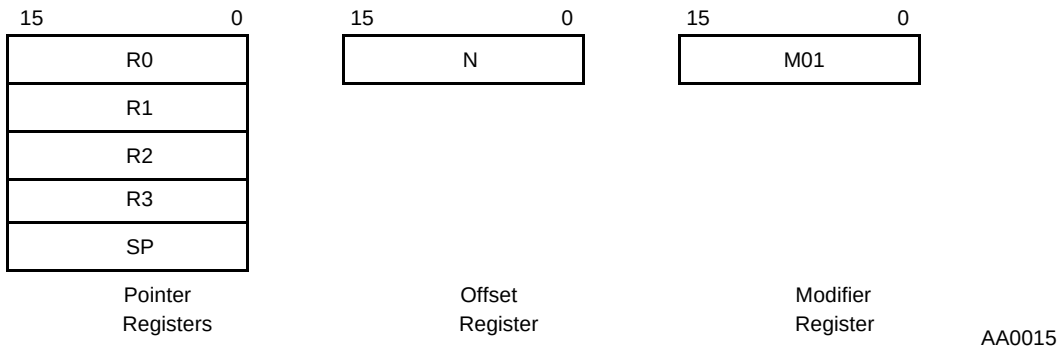


Figure 4-2. Address Generation Unit Programming Model

4.1.1 Address Registers (R0-R3)

The address register file consists of four 16-bit registers R0-R3 (Rn) that usually contain addresses used as pointers to memory. Each register may be read or written by the CGDB. High speed access to the XAB1, XAB2, and PAB buses is required to allow maximum access time for the internal and external X data memory and program memory. Each address register may be used as input for the modulo arithmetic unit for a register update calculation. Each register may be written by the output of the modulo arithmetic unit.

The R3 register may be used as input to a separate incrementer/decrementer unit for an independent register update calculation. This unit is used in the case of any instruction that performs two data memory reads in its parallel move field. For instructions where two reads are performed from the X data memory, the second read using the R3 pointer must always access on-chip memory.

NOTE:

Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

4.1.2 Stack Pointer Register (SP)

The stack pointer register (SP) is a single 16-bit register that is used implicitly in all PUSH instruction macros and POP instructions. The SP is used explicitly for memory references when used with the address-register-indirect modes. It is post-decremented on all POPs from the software stack. The SP register may be read or written by the CGDB.

NOTE:

This register must be initialized explicitly by the programmer after coming out of reset.

Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

4.1.3 Offset Register (N)

The offset register (N) usually contains offset values used to update address pointers. This single register can be used to update or index with any of the address registers (R0-R3, SP). This offset register may be read or written by the CGDB. The offset register is used as input to the modulo arithmetic unit. It is often used for array indexing or indexing into a table, as discussed in Section 8.7, “Array Indexes,” on page 8-26.

NOTE:

If the N address register is changed with a MOVE instruction, this register's contents *will* be available for use on the immediately following instruction. In this case the instruction that writes the N address register will be stretched one additional instruction cycle. This is true for the case when the N register is used by the immediately following instruction; if N is not used, then the instruction is not stretched an additional cycle. If the N address register is changed with a bit-field instruction, the new contents *will not* be available for use until the second following instruction.

4.1.4 Modifier Register (M01)

The modifier register (M01) specifies whether linear or modulo arithmetic is used when calculating a new address and may be read or written by the CGDB. This modifier register is automatically read when the R0 address register is used in an address calculation and can optionally be used also when R1 is used. This register has no effect on address calculations done with the R2, R3, or SP registers. It is used as input to the modulo arithmetic unit. This modifier register is preset during a processor reset to $FFFF (linear arithmetic).

NOTE:

Due to pipelining, if an address register (Rn, SP, or M01) is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the following instruction.

4.1.5 Modulo Arithmetic Unit

The modulo arithmetic unit can update one address register or the SP during one instruction cycle. It is capable of performing linear and modulo arithmetic, as described in Section 4.3, "AGU Address Arithmetic." The contents of the modifier register specifies the type of arithmetic to be performed in an address register update calculation. The modifier value is decoded in the modulo arithmetic unit and affects the unit's operation. The modulo arithmetic unit's operation is data-dependent and requires execution cycle decoding of the selected modifier register contents. Note that the modulo capability is only allowed for R0 or R1 updates; it is not allowed for R2, R3, or SP updates.

The modulo arithmetic unit first calculates the result of linear arithmetic (for example, Rn+1, Rn-1, Rn+N) which is selected as the modulo arithmetic unit's output for linear arithmetic. For modulo arithmetic, the modulo arithmetic unit will perform the function (Rn+N) modulo (M01+1), where N can be 1, -1, or the contents of the offset register N. If the modulo operation requires "wraparound" for modulo arithmetic, the summed output of the modulo adder will give the correct, updated address register value; otherwise, if wraparound is not necessary, the linear arithmetic calculation gives the correct result.

4.1.6 Incrementer/Decrementer Unit

The incrementer/decrementer unit is used for address-update calculations during dual data-memory read instructions. It is used either to increment or decrement the R3 register. This adder performs only linear arithmetic; it performs no modulo arithmetic.

4.2 Addressing Modes

The DSP56800 instruction set contains a full set of operand addressing modes, optimized for high-performance signal processing as well as efficient controller code. All address calculations are performed in the address generation unit to minimize execution time.

Addressing modes specify where the operand or operands for an instruction can be found—whether an immediate value, located in a register, or in memory—and provide the exact address of the operand(s).

The addressing modes are grouped into four categories:

- Register direct—directly references the processor registers as operands
- Address register indirect—uses an address register as a pointer to reference a location in memory as an operand
- Immediate—the operand is contained as a value within the instruction itself
- Absolute—uses an address contained within the instruction to reference a location in memory as an operand

An effective address in an instruction will specify an addressing mode (that is, where the operands can be found), and for some addressing modes the effective address will further specify an address register that points to a location in memory, how the address is calculated, and how the register is updated.

These addressing modes are referred to extensively in Section 6.5.2, “LSLL Alias,” on page 6-13.

Several of the examples in the following sections demonstrate the use of assembler forcing operators. These can be used in an instruction to force a desired addressing mode, as shown in Table 4-1.

Table 4-1. Addressing Mode Forcing Operators

Desired Action	Forcing Operator Syntax	Example
Force immediate short data	#<xx	#<$07
Force 16-bit immediate data	#>xxxx	#>$07
Force absolute short address	X:<xx	X:<$02
Force I/O short address	X:<<xx	X:<<$FFE3
Force 16-bit absolute address	X:>xxxx	X:>$02
Force short offset	X:(SP-<xx)	X:(SP-<$02)
Force 16-bit offset	X:(Rn+>xxxx)	X:(R0+>$03)

Other assembler forcing operators are available for jump and branch instructions, as shown in Table 4-2.

Table 4-2. Jump and Branch Forcing Operators

Desired Action	Forcing Operator Syntax	Example
Force 7-bit relative branch offset	<xx	<LABEL1
Force 16-bit absolute jump address	>xxxx	>LABEL5
Force 16-bit absolute loop address	>xxxx	>LABEL4

4.2.1 Register-Direct Modes

The register-direct addressing modes specify that the operand is in one (or more) of the nine data ALU registers, seven address registers, or four control registers. The various options are shown in Table 4-3 on page 4-7.

Table 4-3. Addressing Mode—Register Direct

Addressing Mode: Register Direct	Notation for Register Direct in the Instruction Set Summary[1]	Examples
Any register	DD DDDDD HHH HHHH F F1 F1DD FDD Rj Rn	A, A2, A1, A0 B, B2, B1, B0 Y, Y1, Y0 X0 R0, R1, R2, R3 SP N M01 PC OMR, SR LA, LC HWS

1. The register field notations found in the middle column are explained in more detail in Table 6-16 on page 6-16 and Table 6-15 on page 6-15.

4.2.1.1 Data or Control Register Direct

The operand is in one, two, or three data ALU register(s) as specified in the operands or in a portion of the data bus movement field in the instruction. This addressing mode is also used to specify a control register operand. This reference is classified as a register reference.

4.2.1.2 Address Register Direct

The operand is in one of the seven address registers (R0-R3, N, M01, or SP) specified by an effective address in the instruction. This reference is classified as a register reference.

NOTE:

Due to pipelining, if any address register is changed with a MOVE or bit-field instruction, the new contents *will not* be available for use as a pointer until the second following instruction. If the SP is changed, no LEA or POP instructions are permitted until the second following instruction.

4.2.2 Address-Register-Indirect Modes

When an address register is used to point to a memory location, the addressing mode is called address register indirect. The term *indirect* is used because the operand is not the address register itself, but the contents of the memory location pointed to by the address register. The effective address in the instruction specifies the address register Rn or SP and the address calculation to be performed. These addressing

modes specify that the operand is (or operands are) in memory and provide the specific address(es) of the operand(s). A portion of the data bus movement field in the instruction specifies the memory reference to be performed. The type of address arithmetic used is specified by the address modifier register.

Table 4-4. Addressing Mode—Address Register Indirect

Addressing Mode: Address Register Indirect	Notation in the Instruction Set Summary[1]	Examples
Accessing Program (P) Memory		
Post-increment	P:(Rj)+	P:(R0)+
Post-update by offset N	P:(Rj)+N	P:(R3)+N
Accessing Data (X) Memory		
No update	X:(Rn)	X:(R3) X:(N) X:(SP)
Post-increment	X:(Rn)+	X:(R1)+ X:(SP)+
Post-decrement	X:(Rn)-	X:(R3)- X:(N)-
Post-update by offset N or N3 available for word accesses only	X:(Rn)+N	X:(R1)+N
Indexed by offset N	X:(Rn+N)	X:(R2+N) X:(SP+N)
Indexed by 6-bit displacement R2 and SP registers only	X:(R2+xx) X:(SP-xx)	X:(R2+15) X:(SP-$1E)
Indexed by 16-bit displacement	X:(Rn+xxxx)	X:(R0-97) X:(N+1234) X:(SP+$03F7)

1. Rj represents one of the four pointer registers R0-R3; Rn is any of the AGU address registers R0-R3 or SP.

Address-register-indirect modes may require an offset and a modifier register for use in address calculations. The address register (Rn or SP) is used as the address register, the shared offset register is used to specify an optional offset from this pointer, and the modifier register is used to specify the type of arithmetic performed.

Some addressing modes are only available with certain address registers (Rn). For example, although all address registers support the “indexed by long displacement” addressing mode, only the R2 address register supports the “indexed by short displacement” addressing mode. For instructions where two reads are performed from the X data memory, the second read using the R3 pointer must always be from on-chip memory. The addressed register sets are summarized in Table 4-5.

Table 4-5. Address-Register-Indirect Addressing Modes Available

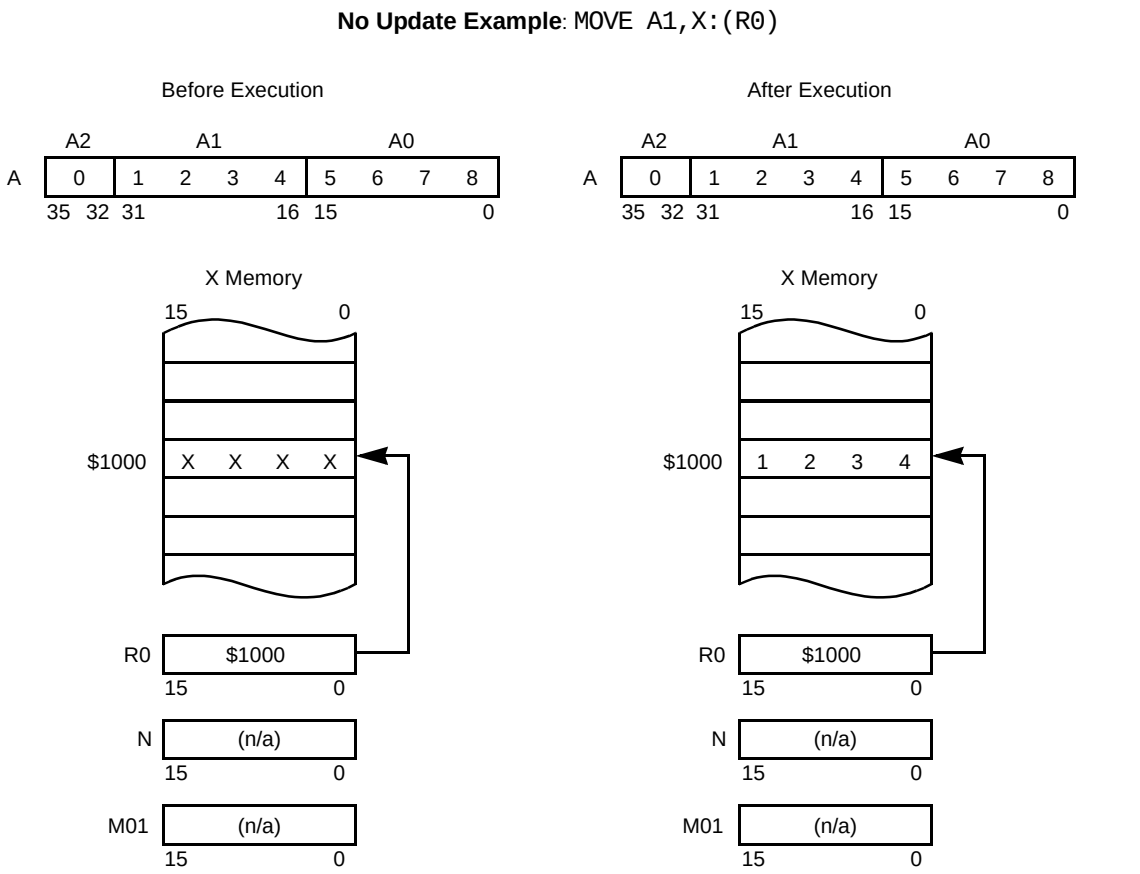
Register Set	Arithmetic Types	Addressing Modes Allowed	Notes
R0/M01/N	Linear or modulo	(R0) (R0)+ (R0)- (R0)+N (R0+N) (R0+xxxx)	R0 *always* uses the M01 register to specify modulo or linear arithmetic. R0 can optionally be used as a source register for the Tcc instruction. R0 is the only register allowed as a counter for the NORM instruction.
R1/M01/N	Linear or modulo	(R1) (R1)+ (R1)- (R1)+N (R1+N) (R1+xxxx)	R1 *optionally* uses the M01 register to specify modulo or linear arithmetic. R1 can optionally be used as a destination register for the Tcc instruction.
R2/N	Linear	(R2) (R2)+ (R2)- (R2)+N (R2+N) (R2+xx) (R2+xxxx)	R2 supports a one-word indexed addressing mode. R2 is not allowed as either pointer for instructions that perform two reads from X data memory. No modulo arithmetic is allowed.
R3/N	Linear	(R3) (R3)+ (R3)- (R3)+N (R3+N) (R3+xxxx)	R3 provides a second address for instructions with two reads from data memory. This second address can only access internal memory. It can also be used for instructions that perform one access to data memory. No modulo arithmetic is allowed.
SP/N	Linear	(SP) (SP)- (SP)+ (SP)+N (SP+N) (SP-xx) (SP+xxxx)	The SP supports a one-word indexed addressing mode, which is useful for accessing local variables and passed parameters. No modulo arithmetic is allowed.

The type of arithmetic to be performed is not encoded in the instruction, but it is specified by the address modifier register (M01 for the DSP56800 core). It indicates whether linear or modulo arithmetic is performed when doing address calculations. In the case where there is not a modifier register for a particular register set (R2 or R3), linear addressing is always performed. For address calculations using R0, the modifier register is always used; for calculations using R1, the modifier register is optionally used.

Each address-register-indirect addressing mode is illustrated in the following subsections.

4.2.2.1 No Update: (Rn), (SP)

The address of the operand is in the address register Rn or SP. The contents of the Rn register are unchanged. The M01 and N registers are ignored. This reference is classified as a memory reference. See Figure 4-3.



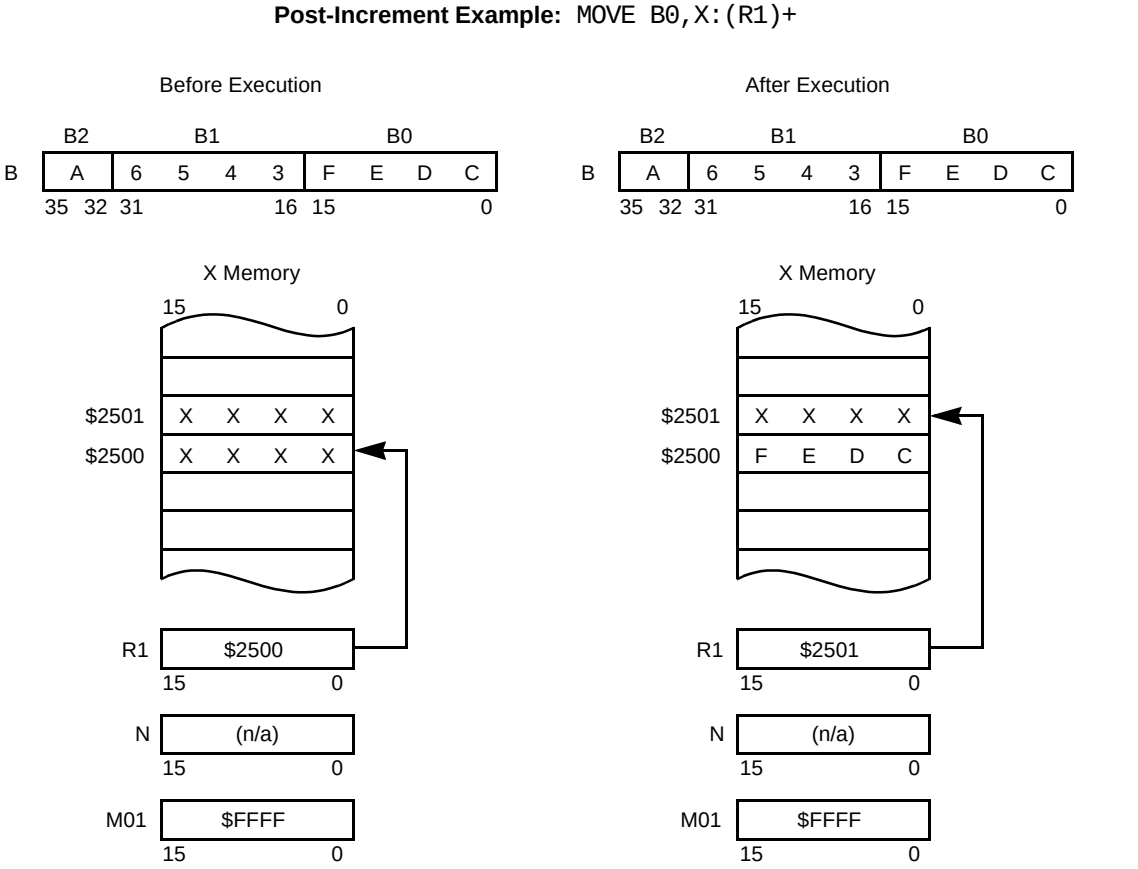
Assembler syntax: X:(Rn), X:(SP)
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0016

Figure 4-3. Address Register Indirect: No Update

4.2.2.2 Post-Increment by 1: (Rn)+, (SP)+

The address of the operand is in the address register Rn or SP. After the operand address is used, it is incremented by one and stored in the same address register. The type of arithmetic (linear or modulo) used to increment Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. The N register is ignored. This reference is classified as a memory reference. See Figure 4-4.



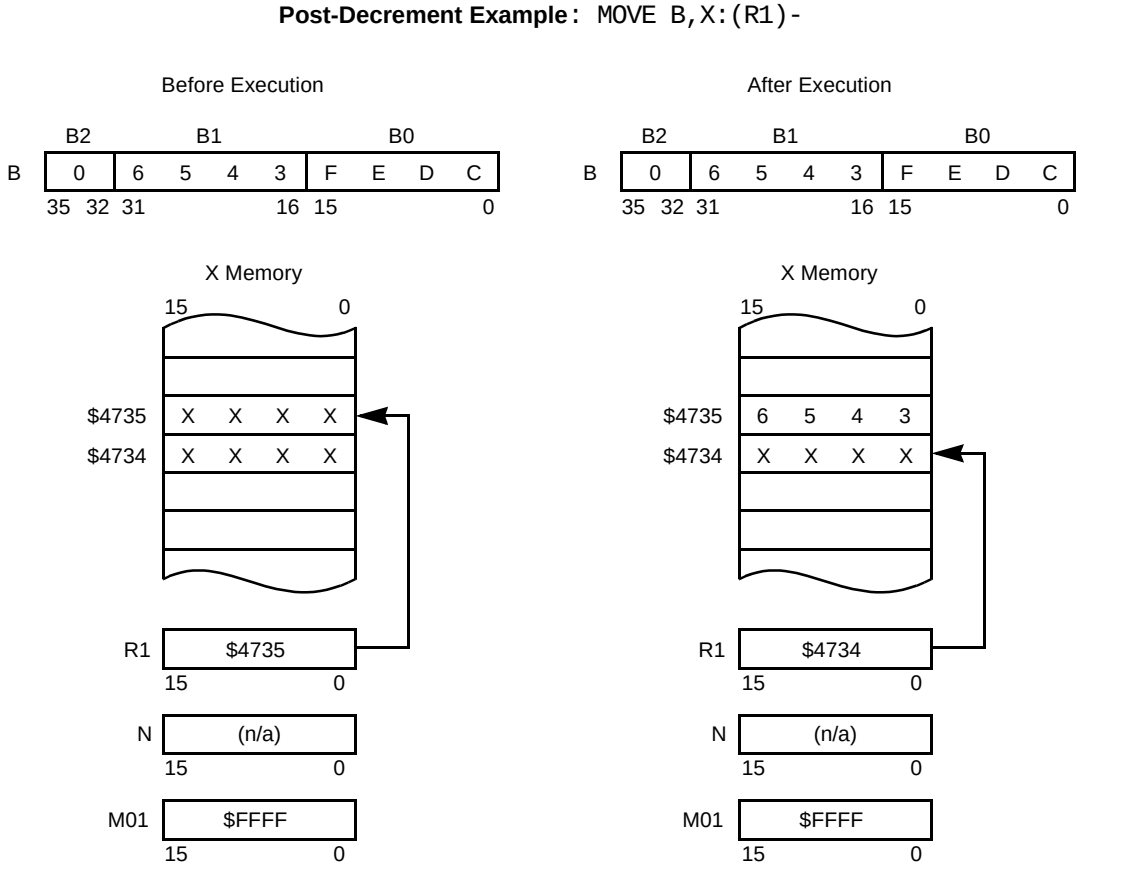
Assembler syntax: X:(Rn)+, X:(SP)+, P:(Rn)+
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0017

Figure 4-4. Address Register Indirect: Post-Increment

4.2.2.3 Post-Decrement by 1: (Rn)-, (SP)-

The address of the operand is in the address register Rn or SP. After the operand address is used, it is decremented by one and stored in the same address register. The type of arithmetic (linear or modulo) used to increment Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. The N register is ignored. This reference is classified as a memory reference. See Figure 4-5.



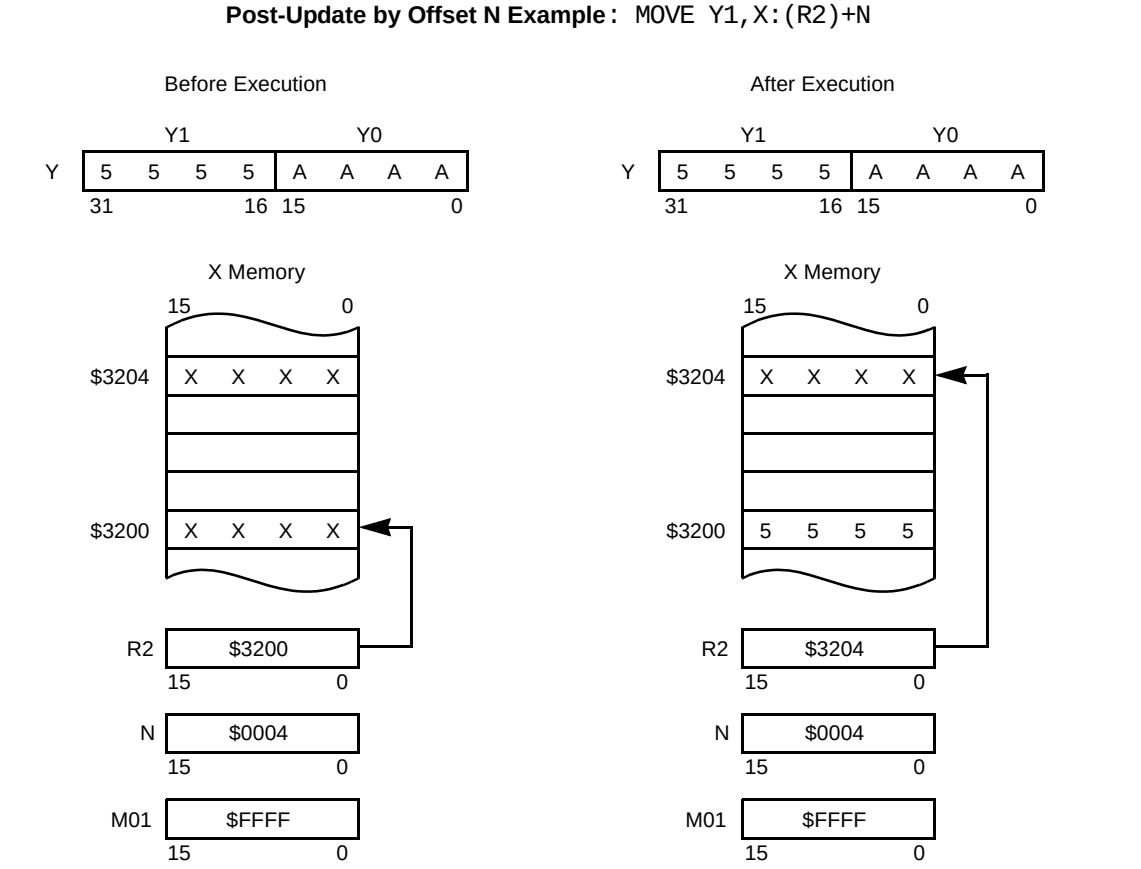
Assembler syntax: X:(Rn)-, X:(SP)-
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0018

Figure 4-5. Address Register Indirect: Post-Decrement

4.2.2.4 Post-Update by Offset N: (Rn)+N, (SP)+N

The address of the operand is in the address register Rn or SP. After the operand address is used, the contents of the N register are added to Rn and stored in the same address register. The content of N is treated as a two's-complement signed number. The contents of the N register are unchanged. The type of arithmetic (linear or modulo) used to update Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This reference is classified as a memory reference. See Figure 4-6.



Assembler syntax: X:(Rn)+N, X:(SP)+N, P:(Rn)+N
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0019

Figure 4-6. Address Register Indirect: Post-Update by Offset N

4.2.2.5 Index by Offset N: (Rn+N), (SP+N)

The address of the operand is the sum of the contents of the address register Rn or SP and the contents of the address offset register N. This addition occurs before the operand can be accessed and, therefore, inserts an extra instruction cycle. The content of N is treated as a two's-complement signed number. The contents of the Rn and N registers are unchanged by this addressing mode. The type of arithmetic (linear or modulo) used to add N to Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This reference is classified as a memory reference. See Figure 4-7.

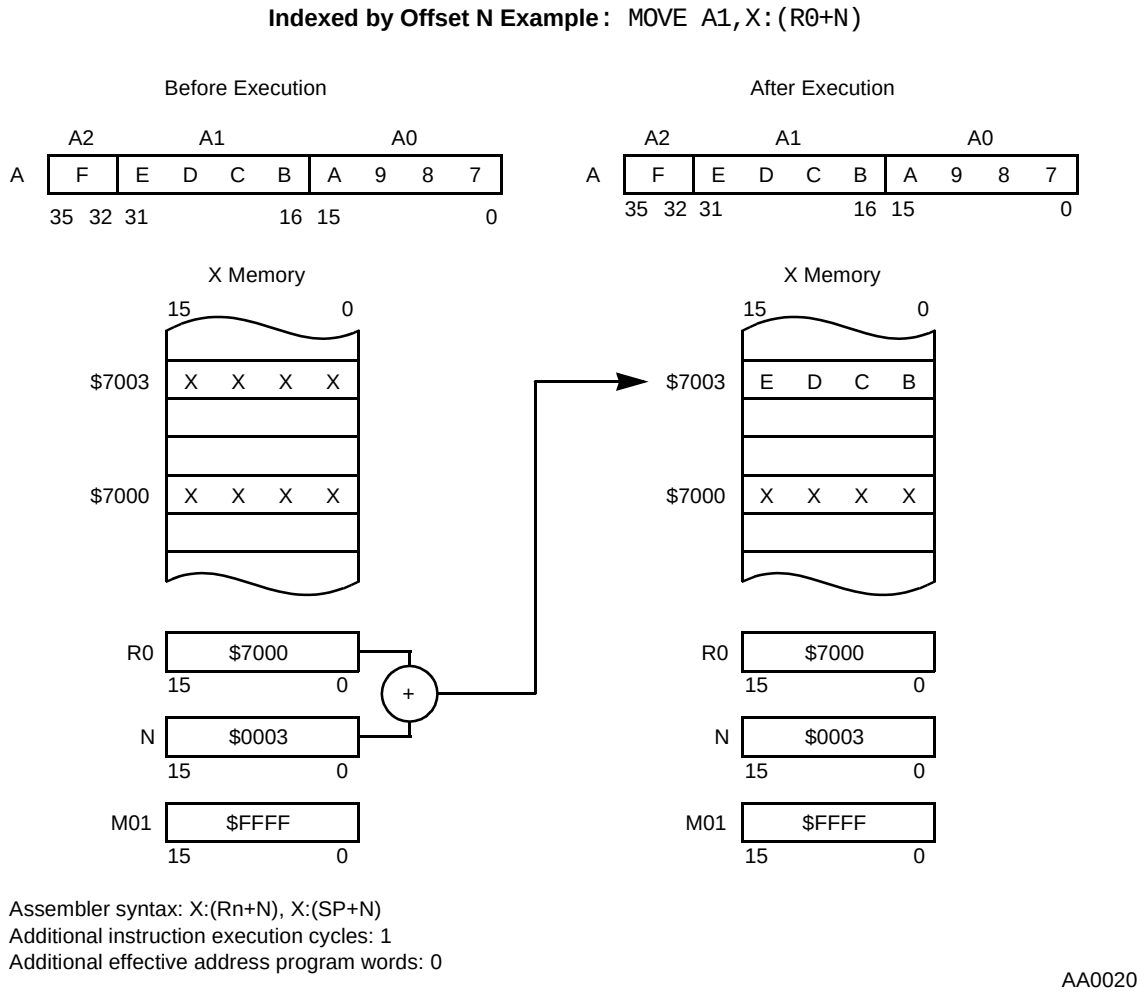


Figure 4-7. Address Register Indirect: Indexed by Offset N

4.2.2.6 Index by Short Displacement: (SP-xx), (R2+xx)

This addressing mode contains the 6-bit short immediate index within the instruction word. This field is always one-extended to form a negative offset when the SP register is used and is always zero-extended to form a positive offset when the R2 register is used. The type of arithmetic used to add the short displacement to R2 or SP is always linear; modulo arithmetic is not allowed. This addressing mode requires an extra instruction cycle. This reference is classified as an X memory reference. See Figure 4-8.

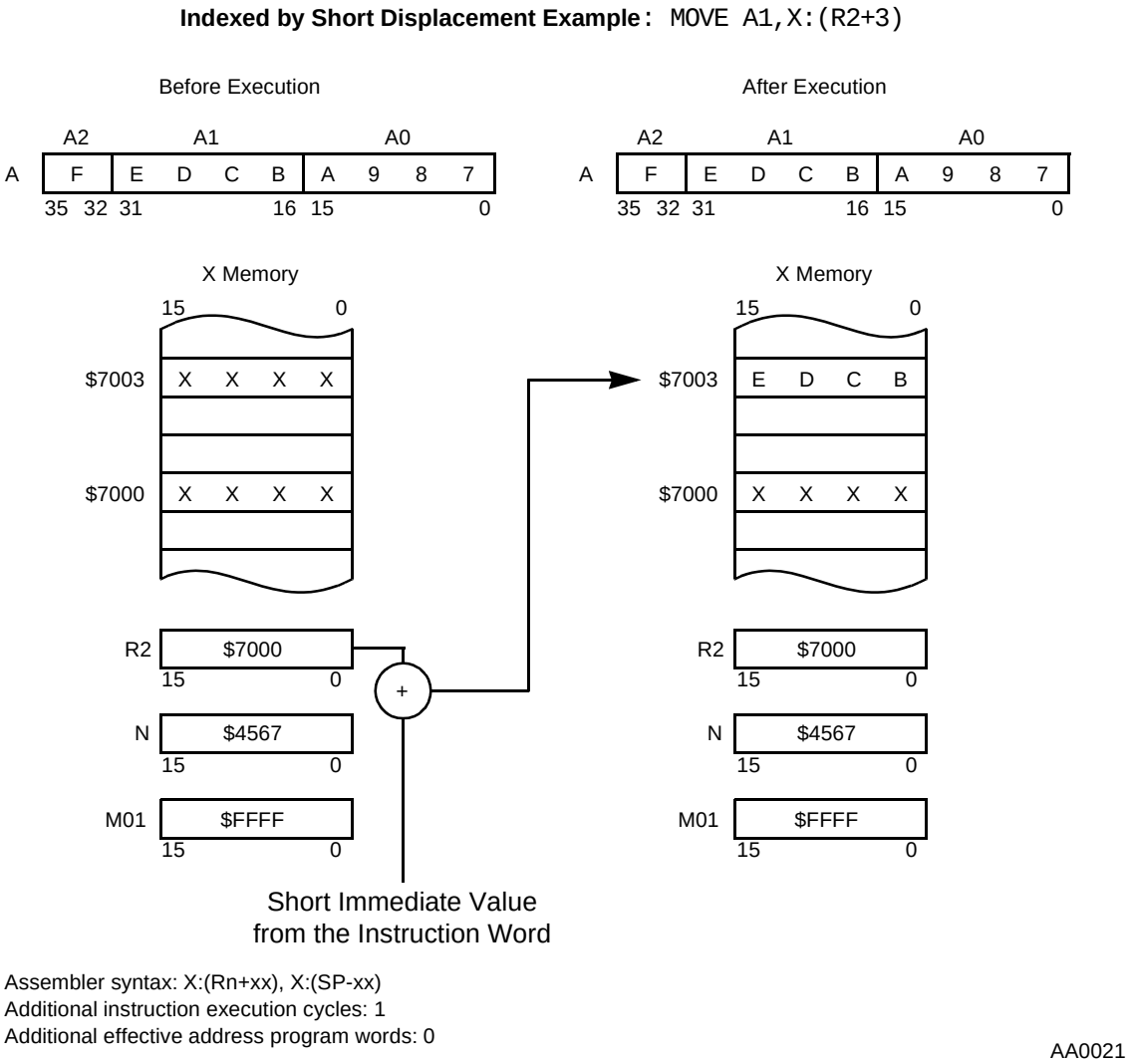
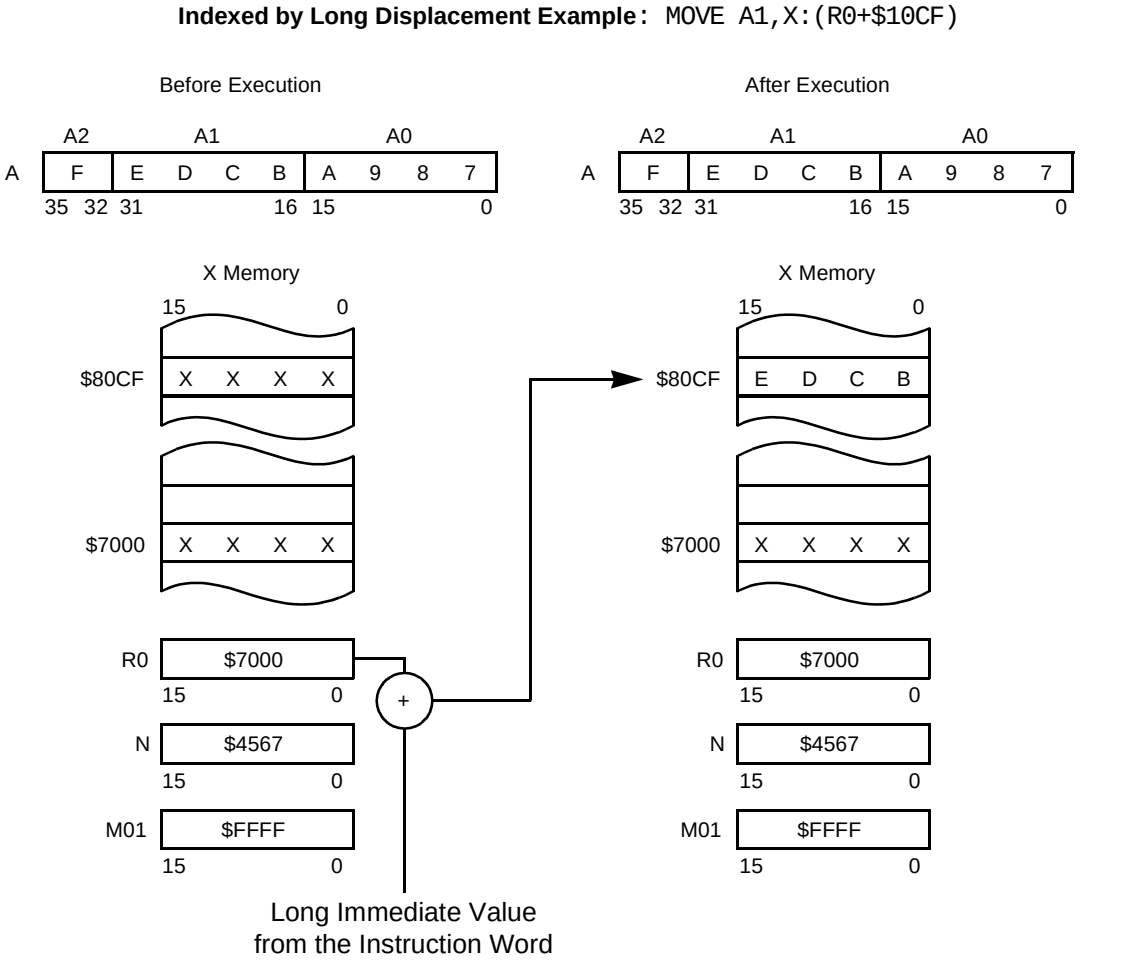


Figure 4-8. Address Register Indirect: Indexed by Short Displacement

4.2.2.7 Index by Long Displacement: (Rn+xxxx), (SP+xxxx)

This addressing mode contains the 16-bit long immediate index within the instruction word. This second word is treated as a signed two's-complement value. The type of arithmetic (linear or modulo) used to add the long displacement to Rn is determined by M01 for R0 and R1 and is always linear for R2, R3, and SP. This addressing mode requires two extra instruction cycles. This addressing mode is available for MOVEC instructions. This reference is classified as an X memory reference. See Figure 4-9.



Assembler syntax: X:(Rn+xxxx), X:(SP+xxxx)
Additional instruction execution cycles: 2
Additional effective address program words: 1

AA0022

Figure 4-9. Address Register Indirect: Indexed by Long Displacement

4.2.3 Immediate Data Modes

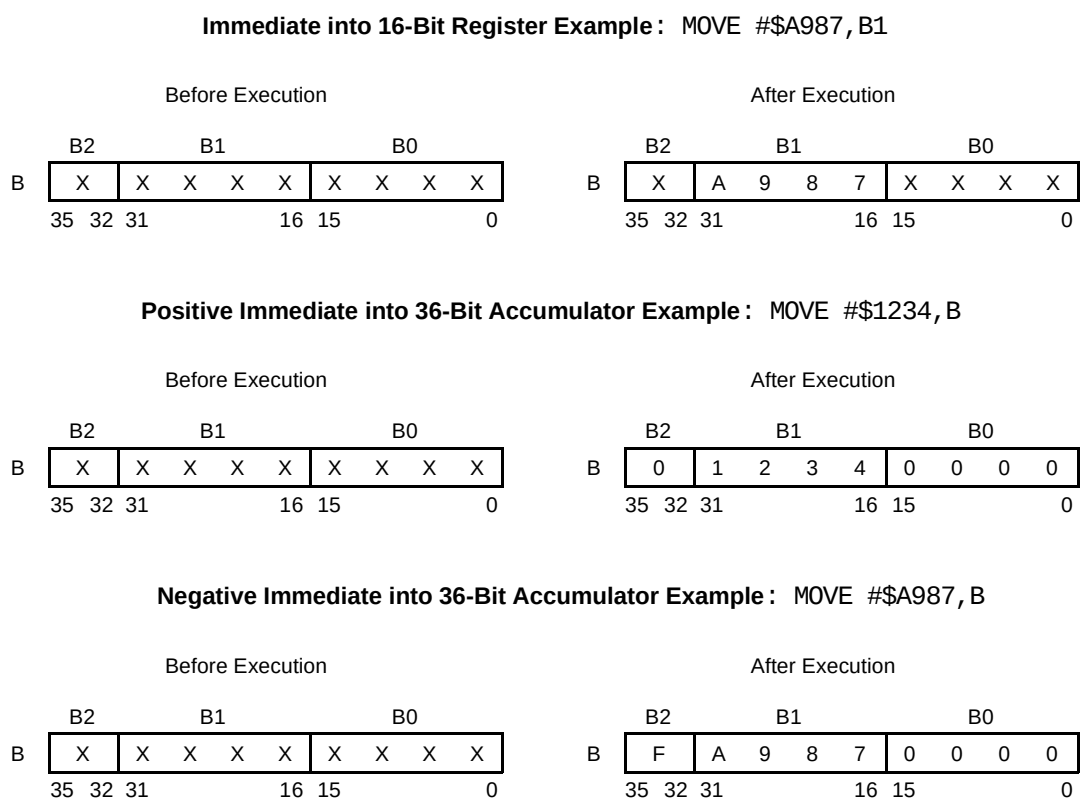
The immediate data modes specify the operand directly in a field of the instruction. That is, the operand value to be used is contained within the instruction word itself (or words themselves). There are two types of immediate data modes: immediate data, which uses an extension word to contain the operand, and immediate short data, where the operand is contained within the instruction word. Table 4-6 summarizes these two modes.

Table 4-6. Addressing Mode—Immediate

Addressing Mode: Immediate	Notation in the Instruction Set Summary	Examples
Immediate short data—5, 6, 7-bit (unsigned and signed)	#xx	#14 #<3
Immediate data—16-bit (unsigned and signed)	#xxxx	#$369C #>1234

4.2.3.1 Immediate Data: #xxxx

This addressing mode requires one word of instruction extension. This additional word contains the 16-bit immediate data used by the instruction. This reference is classified as a program reference. Examples of the use and effects of immediate-data mode are shown in Figure 4-10 on page 4-18.



Assembler syntax: #xxxx
Additional instruction execution cycles: 1
Additional effective address program words: 1

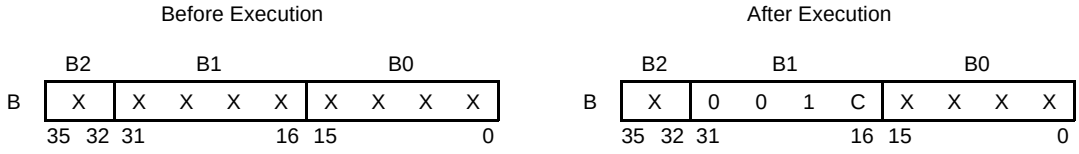
AA0023

Figure 4-10. Special Addressing: Immediate Data

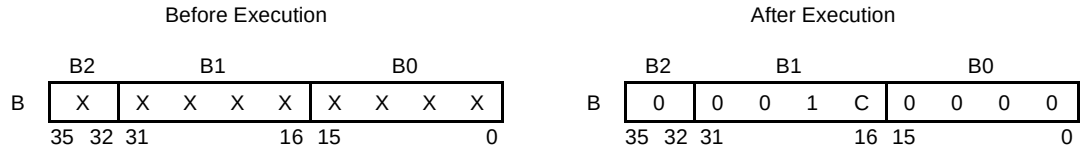
Immediate Short into 16-Bit Address Register Example: MOVE #$0027,N

Immediate Short into 16-Bit Data Register Example: MOVE #$FFC6,X0

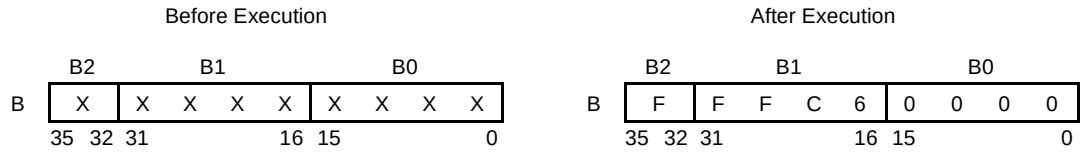
Immediate Short into 16-Bit Accumulator Register Example: MOVE #$001C,B1



Positive Immediate Short into 36-Bit Accumulator Example: MOVE #$001C,B



Negative Immediate Short into 36-Bit Accumulator Example: MOVE #$FFC6,B



Assembler syntax: #xx
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0024

Figure 4-11. Special Addressing: Immediate Short Data

4.2.3.2 Immediate Short Data: #xx

The immediate-short-data operand is located within the instruction operation word. A 6-bit unsigned positive operand is used for DO and REP instructions, and a 7-bit signed operand is used for an immediate move to an on-core register instruction. This reference is classified as a program reference. See Figure 4-11 on page 4-19.

4.2.4 Absolute Addressing Modes

Similar to the direct addressing modes, the absolute addressing modes specify the operand value within the instruction or instruction-extension words. Unlike the direct modes, these values are not used as the operands themselves, but are interpreted as absolute data memory addresses for the operand values. The different absolute addressing modes are shown in Table 4-7.

Table 4-7. Addressing Mode—Absolute

Addressing Mode: Absolute	Notation in the Instruction Set Summary	Examples
Absolute short address—6 bit (direct addressing)	X:aa	X:$0002 X:<$02
I/O short address—6 bit (direct addressing)	X:pp	X:$00FFE3 X:<<$FFE3
Absolute address—16-bit (extended addressing)	X:xxxx	X:$00F001 X:>$C002

4.2.4.1 Absolute Address (Extended Addressing): xxxx

This addressing mode requires one word of instruction extension, which contains the 16-bit absolute address of the operand. No registers are used to form the address of the operand. Absolute address instructions are used with the bit-manipulation and move instructions. This reference is classified as a memory reference and a program reference. See Figure 4-12.

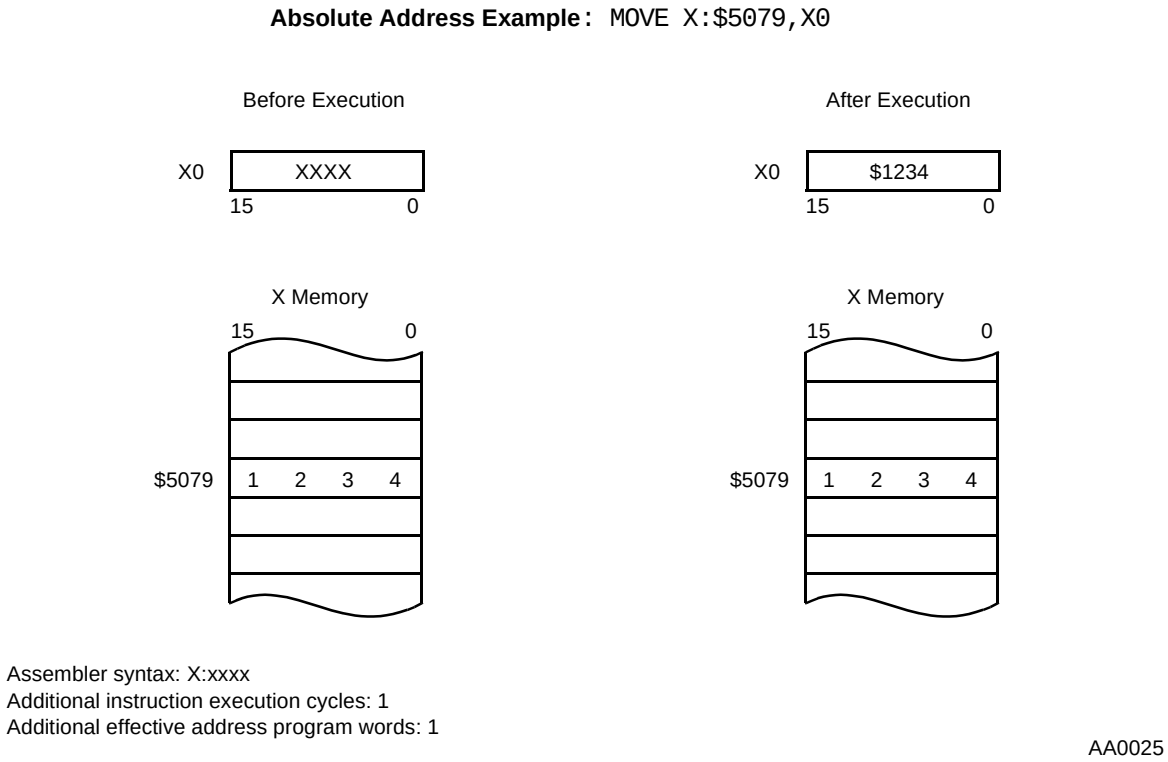


Figure 4-12. Special Addressing: Absolute Address

4.2.4.2 Absolute Short Address (Direct Addressing): <aa>

For the absolute short addressing mode, the address of the operand occupies 6 bits in the instruction operation word and is zero-extended. This allows direct access to the first 64 locations in X memory. No registers are used to form the address of the operand. Absolute short instructions are used with the bit-field manipulation and move instructions. See Figure 4-13.

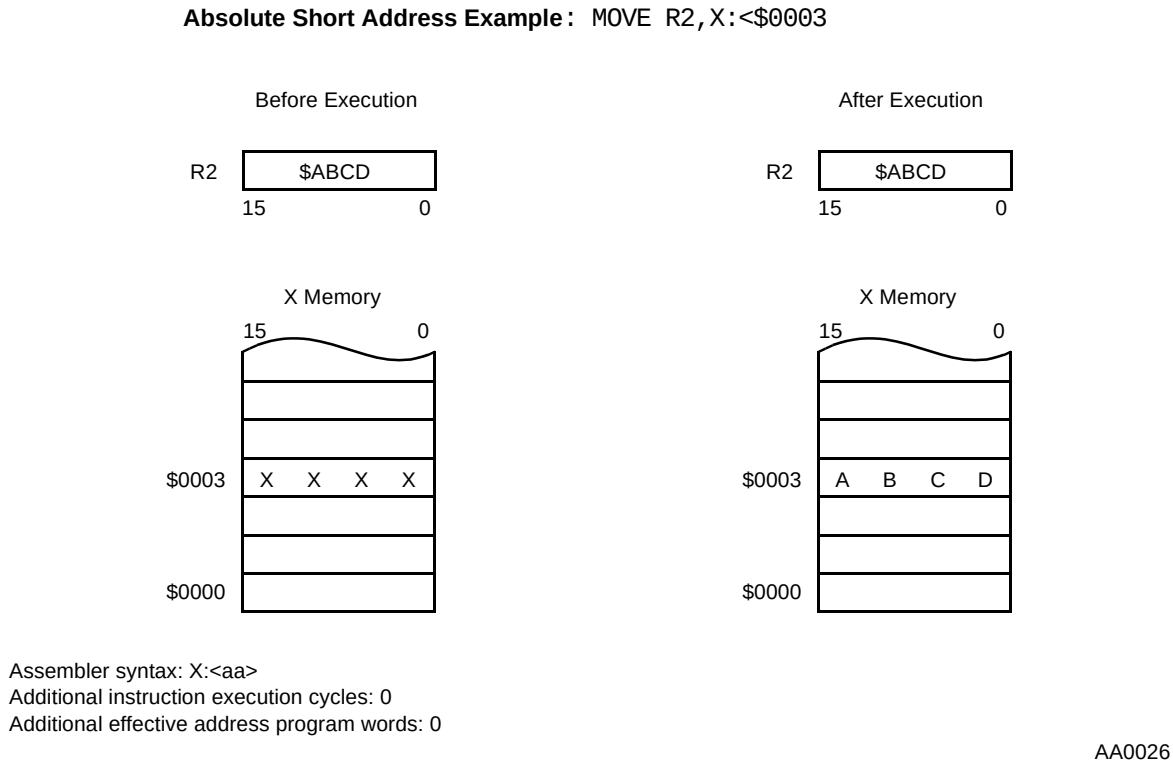
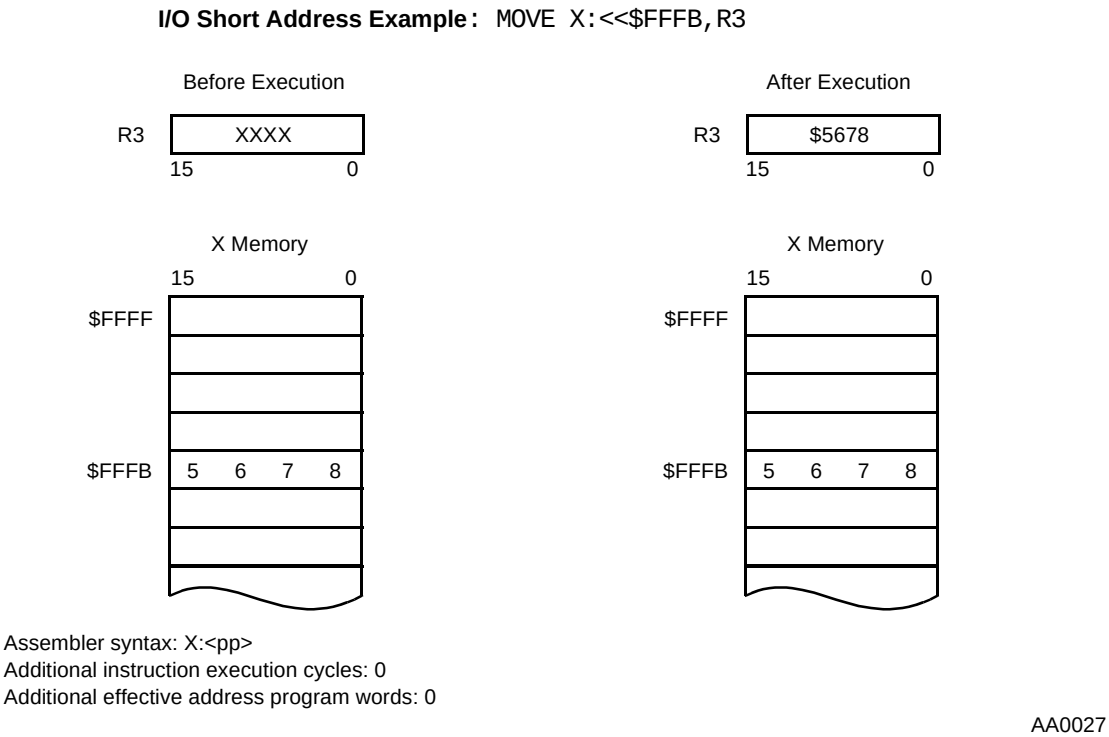


Figure 4-13. Special Addressing: Absolute Short Address

4.2.4.3 I/O Short Address (Direct Addressing): <pp>

For the I/O short addressing mode, the address of the operand occupies 6 bits in the instruction operation word and is one-extended. This allows direct access to the last 64 locations in X memory, which contain the on-chip peripheral registers. No registers are used to form the address of the operand. See Figure 4-14 for examples of using the I/O short direct addressing mode.



Assembler syntax: X:<pp>
Additional instruction execution cycles: 0
Additional effective address program words: 0

AA0027

Figure 4-14. Special Addressing: I/O Short Address

4.2.5 Implicit Reference

Some instructions make implicit reference to the program counter (PC), software stack, hardware stack (HWS), loop address register (LA), loop counter (LC), or status register (SR). The implied registers and their use are defined by the individual instruction descriptions. See Appendix A, “Instruction Set Details,“ for more information.

4.2.6 Addressing Modes Summary

Table 4-8 on page 4-24 contains a summary of the addressing modes discussed in the preceding subsections of Section 4.2.

Table 4-8. Addressing Mode Summary

Addressing Mode	Uses M01[1]	Operand Reference							Assembler Syntax
		S[2]	C[3]	D[4]	A[5]	P[6]	X[7]	XX[8]	
Register Direct									
Data or control register	No		X	X					
Address register (Rn, SP)	No				X				Rn
Address modifier register (M01)	No				X				M01
Address offset register (N)	No				X				N
Hardware stack (HWS)	No	X							HWS
Software stack	No						X		
Address Register Indirect									
No update	No						X		(Rn)
Post-increment by 1	Yes					X	X	X	(Rn)+
Post-decrement by 1	Yes						X		(Rn)-
Post-update by offset N	Yes					X	X	X	(Rn)+N
Index by offset N	Yes						X		(Rn+N)
Index by short displacement	No						X		(R2+xx) or (SP-xx)
Index by long displacement	Yes						X		(Rn+xxxx) or (SP+xxxx)
Immediate, Absolute, and Implicit									
Immediate data	No					X			#xxxx
Immediate short data	No					X			#xx
Absolute address	No					X	X		xxxx
Absolute short address	No						X		<aa>
I/O short address	No						X		<pp>
Implicit	No	X	X			X	X		

1. The M01 modifier can only be used on the R0/N/M01 or R1/N/M01 register sets
2. Hardware stack reference
3. Program controller register reference
4. Data ALU register reference
5. Address Generation Unit register reference
6. Program memory reference
7. X memory reference
8. Dual X memory read

4.3 AGU Address Arithmetic

When an arithmetic operation is performed in the address generation unit, it can be performed using either linear or modulo arithmetic. Linear arithmetic is used for general-purpose address computation, as found in all microprocessors. Modulo arithmetic is used to create data structures in memory such as circular buffers, first-in-first-out queues (FIFOs), delay lines, and fixed-size stacks. Using these structures allows data to be manipulated simply by updating address register pointers, rather than by moving large blocks of data.

Linear versus modulo arithmetic is selected using the modifier register, MO1. Arithmetic on the R0 and R1 AGU registers may be performed using either linear or modulo arithmetic. The R2, R3, and SP registers can be modified using linear arithmetic only.

4.3.1 Linear Arithmetic

Linear arithmetic is “normal” address arithmetic, as found on general-purpose microprocessors. It is performed using 16-bit two’s-complement addition and subtraction. The 16-bit offset register N, or immediate data (+1, -1, or a displacement value), is used in the address calculations. Addresses are normally considered unsigned; offsets are considered signed.

Linear arithmetic is enabled for the R0 and R1 registers by setting the modifier register (M01) to $FFFF. The M01 register is set to $FFFF on reset.

NOTE:

To ensure compatibility with future generations of DSP56800-compatible DSP devices, care should be taken to avoid address arithmetic operations that can cause address register values to overflow. On DSP56800 Family chips, register values can be expected to “wrap” appropriately. Future generations may support address ranges > 64K, however, causing potential address-calculation errors.

4.3.2 Modulo Arithmetic

Many DSP and standard control algorithms require the use of specialized data structures, such as circular buffers, FIFOs, and stacks. The DSP56800 architecture provides support for these algorithms by implementing modulo arithmetic in the address generation unit.

4.3.2.1 Modulo Arithmetic Overview

To understand modulo address arithmetic, consider the example of a circular buffer. A circular buffer is a block of sequential memory locations with a special property: a pointer into the buffer is limited to the buffer’s address range. When a buffer pointer is incremented such that it would point past the end of the buffer, the pointer is “wrapped” back to the beginning of the buffer. Similarly, decrementing a pointer that is located at the beginning of the buffer will wrap the pointer to the end. This behavior is achieved by performing modulo arithmetic when incrementing or decrementing the buffer pointers. See Figure 4-15 on page 4-26.

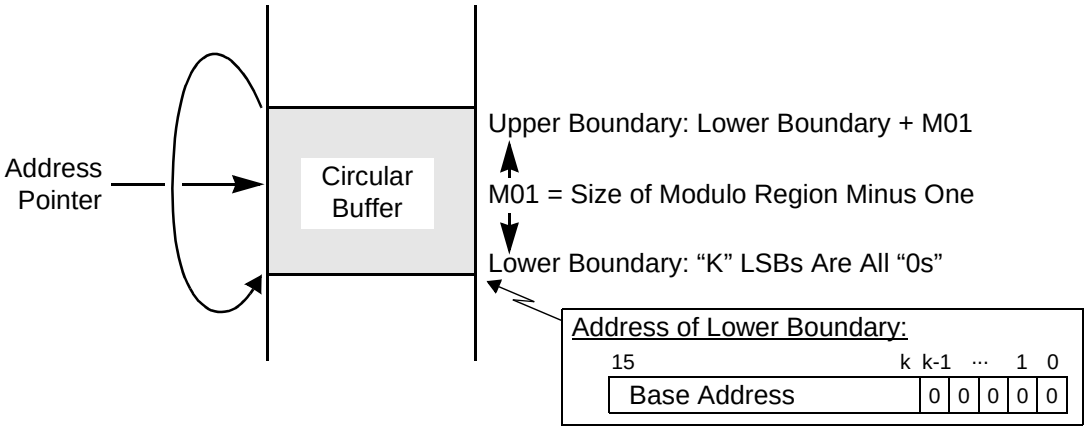


Figure 4-15. Circular Buffer

The modulo arithmetic unit in the AGU simplifies the use of a circular buffer by handling the address pointer wrapping for you. After establishing a buffer in memory, the R0 and R1 address pointers can be made to wrap in the buffer area by programming the M01 register.

Modulo arithmetic is enabled by programming the M01 register with a value that is one less than the size of the circular buffer. See Section 4.3.2.2, “Configuring Modulo Arithmetic,” for exact details on programming the M01 register. Once enabled, updates to the R0 or R1 registers using one of the post-increment or post-decrement addressing modes are performed with modulo arithmetic, and will wrap correctly in the circular buffer.

The address range within which the address pointers will wrap is determined by the value placed in the M01 register and the address contained within one of the pointer registers. Due to the design of the modulo arithmetic unit, the address range is not arbitrary, but limited based on the value placed in M01. The lower bound of the range is calculated by taking the size of the buffer, rounding it up to the next highest power of two, and then rounding the address contained in the R0 or R1 pointers down to the nearest multiple of that value.

For example: for a buffer size of M, a value 2^k is calculated such that $2^k \geq M$. This is the buffer size rounded up to the next highest power of two. For a value M of 37, 2^k would be 64. The lower boundary of the range in which the pointer registers will wrap is the value in the R0 or R1 register with the low-order k bits all set to zero, effectively rounding the value down to the nearest multiple of 2^k (64 in this case). This is shown in Figure 4-16 on page 4-27.

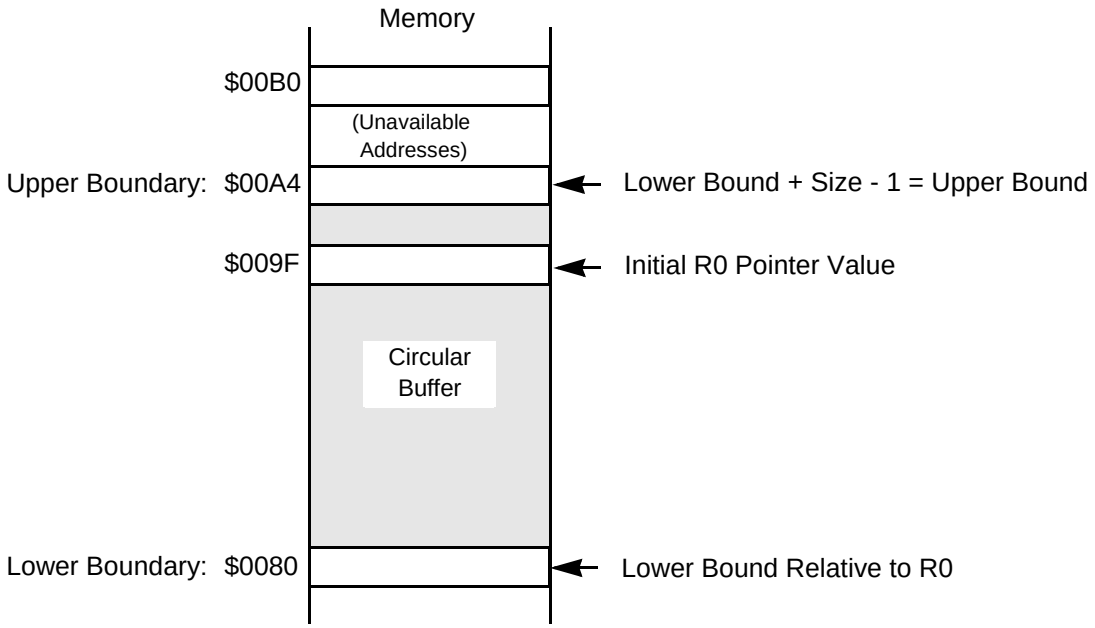


Figure 4-16. Circular Buffer with Size M=37

When modulo arithmetic is performed on the buffer pointer register, only the low-order k bits are modified; the upper 16 - k bits are held constant, fixing the address range of the buffer. The algorithm used to update the pointer register (R0 in this case) is as follows:

R0[15:k] = R0[15:k]
R0[k-1:0] = (R0[k-1:0] + offset) MOD (M01 + 1)

Note that this algorithm can result in some memory addresses being unavailable. If the size of the buffer is not an even power of two, there will be a range of addresses between M and 2^k-1 (37 and 63 in our example) that are not addressable. Section 4.3.2.7.3, “Memory Locations Not Available for Modulo Buffers,” addresses this issue in greater detail.

4.3.2.2 Configuring Modulo Arithmetic

As noted in Section 4.3.2.1, “Modulo Arithmetic Overview,” modulo arithmetic is enabled by programming the address modifier register, M01. This single register enables modulo arithmetic for both the R0 and R1 registers, although in order for modulo arithmetic to be enabled for the R1 register it must be enabled for the R0 register as well. When both pointers use modulo arithmetic, the sizes of both buffers are the same. They can refer to the same or different buffers as desired.

The possible configurations of the M01 register are given in Table 4-9.

Table 4-9. Programming M01 for Modulo Arithmetic

16-Bit M01 Register Contents	Address Arithmetic Performed	Pointer Registers Affected
$0000	(Reserved)	—
$0001	Modulo 2	R0 pointer only
$0002	Modulo 3	R0 pointer only

Table 4-9. Programming M01 for Modulo Arithmetic (Continued)

16-Bit M01 Register Contents	Address Arithmetic Performed	Pointer Registers Affected
...	...	...
$3FFE	Modulo 16383	R0 pointer only
$3FFF	Modulo 16384	R0 pointer only
$4000	(Reserved)	—
...	...	...
$7FFF	(Reserved)	—
$8000	(Reserved)	—
$8001	Modulo 2	R0 and R1 pointers
$8002	Modulo 3	R0 and R1 pointers
...	...	...
$BFFE	Modulo 16383	R0 and R1 pointers
$BFFF	Modulo 16384	R0 and R1 pointers
$C000	(Reserved)	—
...	...	...
$FFFE	(Reserved)	—
$FFFF	Linear Arithmetic	R0 and R1 pointers both set up for linear arithmetic

The high-order two bits of the M01 register determine the arithmetic mode for R0 and R1. A value of 00 for M01[15:14] selects modulo arithmetic for R0. A value of 10 for M01[15:14] selects modulo arithmetic for both R0 and R1. A value of 11 disables modulo arithmetic. The remaining 14 bits of M01 hold the size of the buffer minus one.

NOTE:

The reserved values ($0000, $4000-$8000, and $C000-$FFFE) should not be used. The behavior of the modulo arithmetic unit is undefined for these values, and may result in erratic program execution.

4.3.2.3 Supported Memory Access Instructions

The address generation unit supports modulo arithmetic for the following address-register-indirect modes:

(Rn)	(Rn)+
(Rn)-	(Rn)+N
(Rn+N)	(Rn+xxxx)

As noted in the preceding discussion, modulo arithmetic is only supported for the R0 and R1 address registers.

4.3.2.4 Simple Circular Buffer Example

Suppose a five-location circular buffer is needed for an application. The application locates this buffer at X:$800 in memory. (This location is arbitrary—any location in data memory would suffice.) In order to configure the AGU correctly to manage this circular buffer, the following two pieces of information are needed:

The size of the buffer: five words
The location of the buffer: X:$0800 – X:$0804

Modulo addressing is enabled for the R0 pointer by writing the size minus one ($0004) to M01[13:0], and 00 to M01[15:14]. See Figure 4-17.

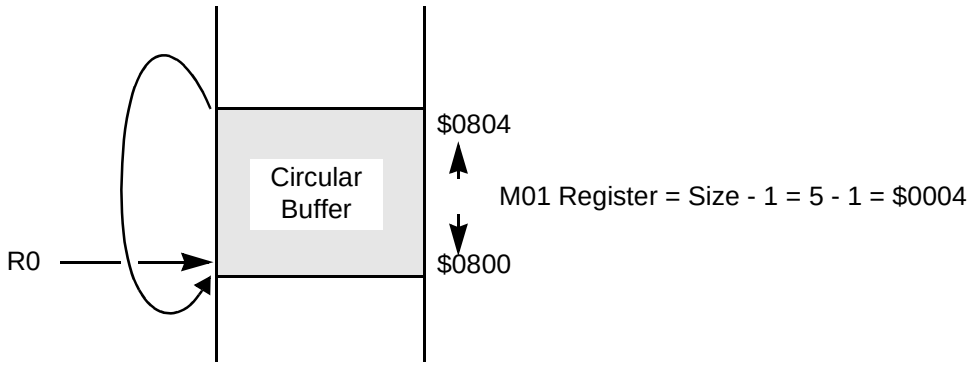


Figure 4-17. Simple Five-Location Circular Buffer

The location of the buffer in memory is determined by the value of the R0 pointer when it is used to access memory. The size of the memory buffer (five in this case) is rounded *up* to the nearest power of two (eight in this case). The value in R0 is then rounded *down* to the nearest multiple of eight. For the base address to be X:$0800, the initial value of R0 must be in the range X:$0800 – X:$0804. Note that the initial value of R0 does not have to be X:$0800 to establish this address as the lower bound of the buffer. However, it is often convenient to set R0 to the beginning of the buffer. The source code in Example 4-1 shows the initialization of the example buffer.

Example 4-1. Initializing the Circular Buffer

```
MOVE   #(5-1),M01   ; Initialize the buffer for five locations
MOVE   #$0800,R0    ; R0 can be initialized to any location
                    ; within the buffer. For simplicity, R0
                    ; is initialized to the value of the lower
                    ; boundary
```

The buffer is used simply by accessing it with MOVE instructions. The effect of modulo address arithmetic becomes apparent when the buffer is accessed multiple times, as in Example 4-2 on page 4-30.

Example 4-2. Accessing the Circular Buffer

```
MOVE   X:(R0)+,X0   ; First time accesses location $0800
                    ; and bumps the pointer to location $0801
MOVE   X:(R0)+,X0   ; Second accesses at location $0801
MOVE   X:(R0)+,X0   ; Third accesses at location $0802
MOVE   X:(R0)+,X0   ; Fourth accesses at location $0803
MOVE   X:(R0)+,X0   ; Fifth accesses at location $0804
                    ; and bumps the pointer to location $0800

MOVE   X:(R0)+,X0   ; Sixth accesses at location $0800 <=== NOTE
MOVE   X:(R0)+,X0   ; Seventh accesses at location $0801
MOVE   X:(R0)+,X0   ; and so forth...
```

For the first several memory accesses, the buffer pointer is incremented as expected, from $0800 to $0801, $0802, and so forth. When the pointer reaches the top of the buffer, rather than incrementing from $0804 to $0805, the pointer value “wraps” back to $0800.

The behavior is similar when the buffer pointer register is incremented by a value greater than one. Consider the source code in Example 4-3, where R0 is post-incremented by three rather than one. The pointer register correctly “wraps” from $0803 to $0801—the pointer does not have to land exactly on the upper and lower bound of the buffer for the modulo arithmetic to wrap the value properly.

Example 4-3. Accessing the Circular Buffer with Post-Update by Three

```
MOVE   #(5-1),M01   ; Initialize the buffer for five locations
MOVE   #$0800,R0    ; Initialize the pointer to $0800
MOVE   #3,N         ; Initialize “bump value” to 3
NOP
NOP
MOVE   X:(R0)+N,X0  ; First time accesses location $0800
                    ; and bumps the pointer to location $0803
MOVE   X:(R0)+N,X0  ; Second accesses at location $0803
                    ; and wraps the pointer around to $0801

MOVE   X:(R0)+N,X0  ; Third accesses at location $0801
                    ; and bumps the pointer to location $0804
MOVE   X:(R0)+N,X0  ; Fourth accesses at ...
```

In addition, the pointer register does not need to be incremented; it could be decremented instead. Instructions that post-decrement the buffer pointer also work correctly. Executing the instruction `MOVE X:(R0)-,X0` when the value of R0 is $0800 will correctly set R0 to $0804.

4.3.2.5 Setting Up a Modulo Buffer

The following steps detail the process of setting up and using the 37-location circular buffer shown in Figure 4-16 on page 4-27.

1. Determine the value for the M01 register.
   - — Select the size of the desired buffer; it can be no larger than 16,384 locations. If modulo arithmetic is to be enabled only for the R0 address register, this gives the following:
     M01 = # locations - 1 = 37 - 1 = 36 = $0024
   - — If modulo arithmetic is to be enabled for both the R0 and R1 address registers, be sure to set the high-order bit of M01:
     M01 = # locations - 1 + $8000 = 37 - 1 + 32768 = 32804 = $8024

2. Find the nearest power of two greater than or equal to the circular buffer size. In this example, the value would be $2^k \geq 37$, which gives us a value of k = 6.
3. From k, derive the characteristics of the lower boundary of the circular buffer. Since the "k" least-significant bits of the address of the lower boundary must all be 0s, then the buffer base address must be some multiple of 2^k. In this case, k = 6, so the base address is some multiple of $2^6 = 64$.
4. Locate the circular buffer in memory.
   — The location of the circular buffer in memory is determined by the upper 16 - k bits of the address pointer register used in a modulo arithmetic operation. If there is an open area of memory from locations 111 to 189 ($006F to $00BD), for example, then the addresses of the lower and upper boundaries of the circular buffer will fit in this open area for J = 2:
   Lower boundary = (J x 64) = (2 x 64) = 128 = $0080
   Upper boundary = (J x 64) + 36 = (2 x 64) + 36 = 164 = $00A4
   — The exact area of memory in which a circular buffer is prepared is specified by picking a value for the address pointer register, R0 or R1, whose value is inclusively between the desired lower and upper boundaries of the circular buffer. Thus, selecting a value of 139 ($008B) for R0 would locate the circular buffer between locations 128 and 164 ($0080 to $00A4) in memory since the upper 10 (16 - k) bits of the address indicate that the lower boundary is 128 ($0080).
   — In summary, the size and exact location of the circular buffer is defined once a value is assigned to the M01 register and to the address pointer register (R0 or R1) that will be used in a modulo arithmetic calculation.
5. Determine the upper boundary of the circular buffer, which is the lower boundary + # locations - 1.
6. Select a value for the offset register if it is used in modulo operations.
   — If the offset register is used in a modulo arithmetic calculation, it must be selected as follows: $|N| \leq M01 + 1$ [where |N| refers to the absolute value of the contents of the offset register]
   — The special case where N is a multiple of the block size, 2^k, is discussed in Section 4.3.2.6, "Wrapping to a Different Bank."
7. Perform the modulo arithmetic calculation.
   — Once the appropriate registers are set up, the modulo arithmetic operation occurs when an instruction with any of the following addressing modes using the R0 (or R1, if enabled) register is executed:
   (Rn)
   (Rn)+
   (Rn)-
   (Rn)+N
   (Rn+N)
   (Rn+xxxx)
   — If the result of the arithmetic calculation would exceed the upper or lower bound, then wrapping around is correctly performed.

4.3.2.6 Wrapping to a Different Bank

For the normal case where |N| is less than or equal to M01, the primary address arithmetic unit will automatically wrap the address pointer around by the required amount. This type of address modification is useful in creating circular buffers for FIFOs, delay lines, and sample buffers up to 16,384 words long. It is also used for decimation, interpolation, and waveform generation.

If |N| is greater than M01, the result is data dependent and unpredictable except for the special case where $N = L*(2^k)$, a multiple of the block size, 2^k, where L is a positive integer. For this special case when using the (Rn)+N addressing mode, the pointer Rn will be updated using linear arithmetic to the same relative address that is L blocks forward in memory (see Figure 4-18). Note that this case requires that the offset N must be a positive two's-complement integer.

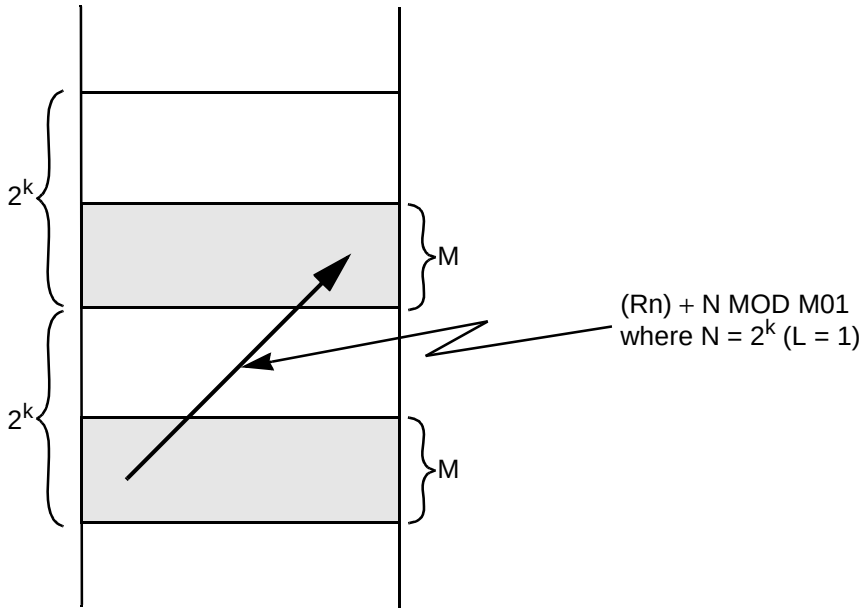


Figure 4-18. Linear Addressing with a Modulo Modifier

This technique is useful in sequentially processing multiple tables or N-dimensional arrays. The special modulo case of (Rn)+N with $N = L*(2^k)$ is useful for performing the same algorithm on multiple blocks of data in memory (e.g., implementing a bank of parallel IIR filters).

4.3.2.7 Side Effects of Modulo Arithmetic

Due to the way modulo arithmetic is implemented by the DSP56800 Family, there are some side effects of using modulo arithmetic that must be kept in mind. Specifically, since the base address of a buffer must be a power of two, and since the modulo arithmetic unit can only detect a single wraparound, there are some restrictions and limitations that must be considered.

4.3.2.7.1 When a Pointer Lies Outside a Modulo Buffer

If a pointer is outside the valid modulo buffer range and an operation occurs that causes R0 or R1 to be updated, the contents of the pointer will be updated according to modulo arithmetic rules. For example, a `MOVE B,X:(R0)+N` instruction, where R0 = 6, M01 = 5, and N = 0, would apparently leave R0 unchanged since N = 0. However, since R0 is above the upper boundary, the AGU calculates R0 + N - (M01 + 1) for the new contents of R0 and sets R0 = 0.

4.3.2.7.2 Restrictions on the Offset Register

The modulo arithmetic unit in the AGU is only capable of detecting a single wraparound of an address pointer. As a result, if the post-update addressing mode, (Rn)+N, is used, care must be taken in selecting the value of N. The 16-bit absolute value |N| must be less than or equal to M01 + 1 for proper modulo addressing. Values of |N| larger than the size of the buffer may result in the Rn address value wrapping twice, which the AGU cannot detect.

4.3.2.7.3 Memory Locations Not Available for Modulo Buffers

For cases where the size of a buffer is not a power of two, there will be a range of memory locations immediately after the buffer that are not accessible with modulo addressing. Lower boundaries for modulo buffers always begin on an address where the lowest k bits are zeros—that is, a power of two. This means that for buffers that are not an exact power of two, there are locations above the upper boundary that are not accessible through modulo addressing.

In Figure 4-16 on page 4-27, for example, the buffer size is 37, which is not a power of two. The smallest power of two greater than 37 is 64. Thus, there are 64 - 37 = 27 memory locations which are not accessible with modulo addressing. These 27 locations are between the upper boundary + 1 = $00A5 and the next power of two boundary address - 1 = $00C0 - 1 = $00BF.

These locations are still accessible when no modulo arithmetic is performed. Using linear addressing (with the R2 or R3 pointers), absolute addresses, or the no-update addressing mode makes these locations available.

4.4 Pipeline Dependencies

There are some cases within the address generation unit where the pipelined nature of the DSP core can affect the execution of a sequence of instructions. The pipeline dependencies are caused by a write to an AGU register immediately followed by an instruction that uses that same register in an address arithmetic calculation. When there is a dependency caused by a write to the N register, the DSP automatically stalls the pipeline one cycle. If a dependency is caused by a write to the R0-R3, SP, or M01 registers, however, there is no pipeline stall. This is also true if a bit-field operation is performed on the N register. Instead, the user must take care to avoid this case by rearranging the instructions or by inserting a NOP instruction to break the instruction sequence.

Several instruction sequences are presented in the following examples to examine cases where their pipeline dependency occurs, how this affects the machine, and how to correctly program to avoid these dependencies.

In Example 4-4 there is no pipeline dependency since the N register is not used in the second instruction. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-4. No Dependency with the Offset Register

```
MOVE   #$7,N              ; Write to the N register
MOVE   X:(R2)+,X0         ; N not used in this instruction
```

In Example 4-5 there is no pipeline dependency since the R2 and N registers, used in the address calculation, are not written in the previous instruction. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-5. No Dependency with an Address Pointer Register

```
MOVE   #$7,R1             ; Write to R1 register
MOVE   X:(R2)+N,X0        ; R1 not used in this instruction
```

In Example 4-6 there is no pipeline dependency since there is no address calculation performed in the second instruction. Instead, the R1 register is used as the source operand in a MOVE instruction, for which there is no pipeline dependency. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-6. No Dependency with No Address Arithmetic Calculation

```
MOVE   #$7,R1              ; Write to R1 register
MOVE   R1,X:$0004          ; No address arithmetic calculation
                           ; performed
```

Example 4-7 represents a special case. For the X:(Rn+xxxx) addressing mode, there is no pipeline dependency even if the same Rn register is written on the previous cycle. This is true for R0-R3 as well as the SP register. Since there is no dependency, no extra instruction cycles are inserted.

Example 4-7. No Dependency with (Rn+xxxx)

```
MOVE   #$7,R1              ; Write to R1 register
MOVE   X:(R1+$3456),X0     ; X:(Rn+xxxx) addressing mode
                           ; using R1
```

In Example 4-8 there is a pipeline dependency since the N register is used in the second instruction. This is true for using N to update R0-R3 as well as the SP register. For the case where a dependency is caused by a write to the N register, the DSP core automatically stalls the pipeline by inserting one extra instruction cycle. Thus, this sequence is allowed. This dependency also exists for the (Rn+N) addressing mode.

Example 4-8. Dependency with a Write to the Offset Register

```
MOVE   #$7,N               ; Write to the N register
MOVE   X:(R2)+N,X0         ; N register used in address
                           ; arithmetic calculation
```

In Example 4-9 there is a pipeline dependency since the N register is used in the second instruction. This is true for using N to update R0-R3 as well as the SP register. For the case where a dependency is caused by a bit-field operation on the N register, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions. This dependency only applies to the BFSET, BFCLR, or BFCHG instructions. There is no dependency for the BFTSTH, BFTSTL, BRCLR, or BRSET instructions. This dependency also exists for the (Rn+N) addressing mode.

Example 4-9. Dependency with a Bit-Field Operation on the Offset Register

```
BFSET  #$7,N               ; Bit-field operation on the N
                           ; register
MOVE   X:(R2)+N,X0         ; N register used in address
                           ; arithmetic calculation
```

In Example 4-10 there is a pipeline dependency since the address pointer register written in the first instruction is used in an address calculation in the second instruction. For the case where a dependency is caused by a write to one of these registers, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions.

Example 4-10. Dependency with a Write to an Address Pointer Register

```
MOVE   #$7,R2              ; Write to the R2 register
MOVE   X:(R2)+,X0          ; R2 register used in address
                           ; arithmetic calculation
```

In Example 4-11 there is a pipeline dependency since the M01 register written in the first instruction is used in an address calculation in the second instruction. For the case where a dependency is caused by a write to the M01 register, this sequence is not allowed and is flagged by the assembler. This sequence may be fixed by rearranging the instructions or inserting a NOP between the two instructions.

Example 4-11. Dependency with a Write to the Modifier Register

```
        MOVE   #$7,M01              ; Write to the M01 register
        MOVE   X:(R0)+,X0           ; M01 register used in address
                                    ; arithmetic calculation
```

In Example 4-12 there is a pipeline dependency since the SP register written in the first instruction is used by the immediately following JSR instruction to store the subroutine return address. The stack pointer will not be updated with the immediate value in this case. This sequence may be fixed by inserting a NOP between the two instructions.

Example 4-12. Dependency with a Write to the Stack Pointer Register

```
        MOVE   #$3800,SP            ; Write to the SP register
        JSR    LABEL                ; SP implicitly used to save the return address
                                    ; of the subroutine call
```

In Example 4-13 there is a pipeline dependency due to contention in the LF bit of the SR register. During the first execution cycle of the BFSET instruction, the SR, whose LF bit is zero, is read. At the same time, the first operand of the DO instruction is fetched. During the second execution cycle of the BFSET instruction, the SR's content is modified and written back to the SR. This is also the DO instruction decode cycle, when the LF bit is set. In this case, the LF bit is first set by the DO decode, then cleared by the BFSET SR modification. A cleared LF bit signals the end of a DO loop, so the DO loop is executed only once. This sequence can be fixed by inserting a NOP instruction between these two instructions.

Example 4-13. Dependency with a Bit-Field Operation and DO Loop

```
        BFSET  #$0200,SR            ; Write to the SR register
        DO     #8,ENDLOOP           ; Repeat 8 times body of loop
.....
ENDLOOP:
```