2.3 Address Generation Unit

The AGU is one of the execution units in the SC140 core. The AGU performs effective address calculations using the integer arithmetic necessary to address data operands in memory. It also contains the registers used to generate the addresses. The AGU implements four types of arithmetic: linear, modulo, multiple wrap-around modulo, and reverse-carry. It operates in parallel with other chip resources to minimize address generation overhead. The AGU also generates change-of-flow program addresses as well as updates the stack pointer (SP), whenever needed.

2.3.1 AGU Architecture

The major components of the AGU are listed below:

- Eight low bank address registers (R0–R7)
- Eight high bank address registers (R8–R15), or alternatively, eight base address registers (B0–B7)
- Two stack pointers (NSP, ESP), only one of which is active at a time (SP)
- Four offset registers (N0–N3)
- Four modifier registers (M0–M3)
- A modifier control register (MCTL)
- Two address arithmetic units (AAU)
- One bit mask unit (BMU)

In this section, the registers are referred to as:

- Rn for any of the R0–R15 address registers
- Bn for any of the B0–B7 base address registers
- Ni for any of the N0–N3 offset registers
- Mj for any of the M0–M3 modifier registers

All the Rn, Bn, SP, Ni, and Mj registers are referred to as AGU registers. All of the AGU registers are 32-bits.

Figure 2-12 shows a block diagram of the AGU.

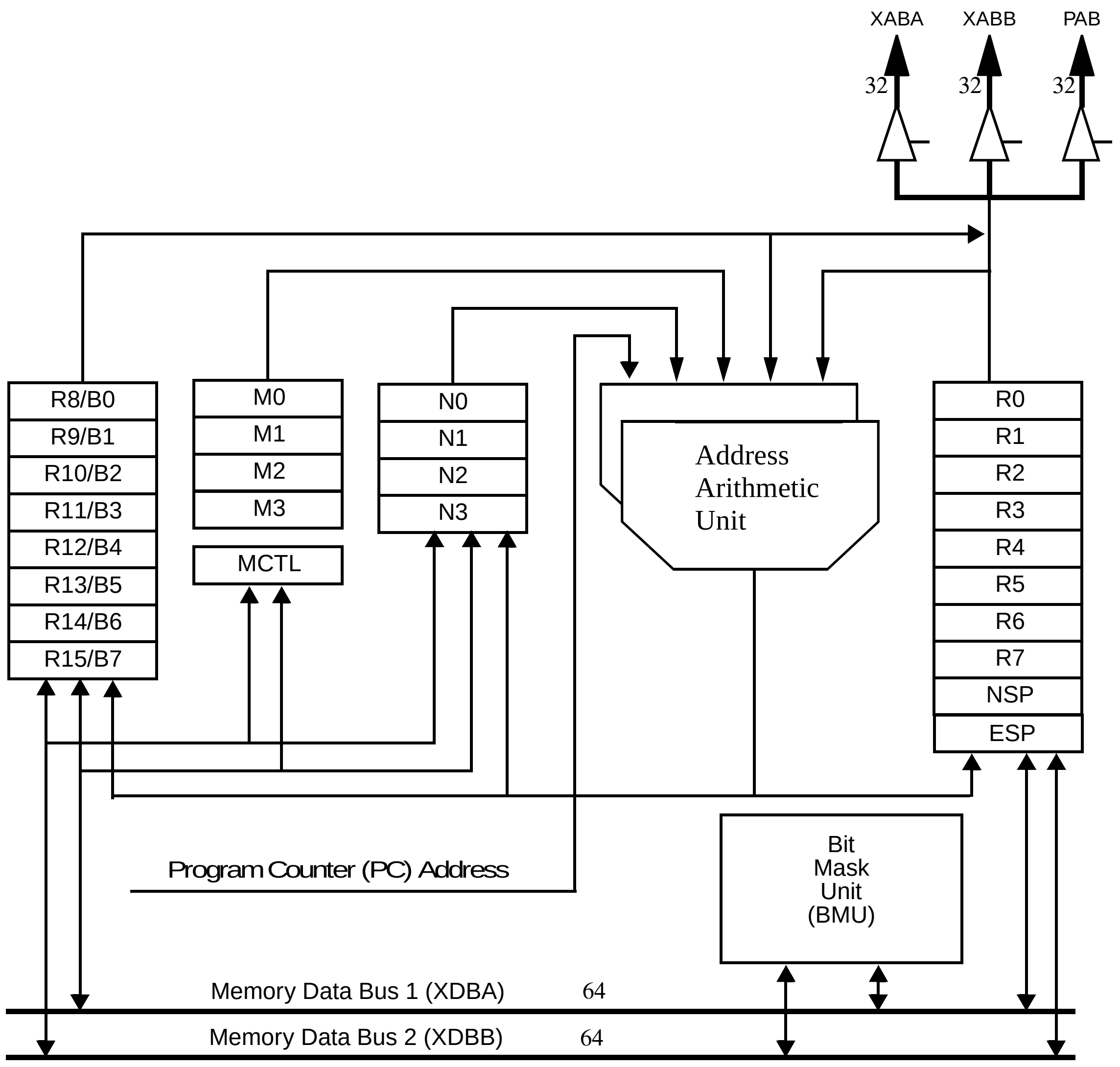


Figure 2-12. AGU Block Diagram

All sixteen address registers (R0–R15) as well as the NSP or ESP are used for generating addresses in the register indirect addressing modes. All four offset registers (N0–N3) can be used by all sixteen address registers. The four modifier registers (M0–M3) can only be used by the low bank of eight address registers (R0–R7).

The base address (Bn) registers are uniquely associated with the low bank of Rn registers such that B0 is used with R0, B1 with R1, and so on.

The BMU is used to perform bit mask operations such as setting, clearing, changing, or testing bits in a destination according to an immediate mask operand. Data is loaded into the BMU over the data memory buses XDBA or XDBB. The result is written back over XDBA or XDBB to the destinations in the next cycle. All bit mask instructions are typically executed in two cycles and work on 16-bit data. This data can be a memory location or a portion (high or low) of a register. For more information, see Section 2.3.6, “Bit Mask Instructions.”

During every instruction cycle, the two AAUs can generate one 32-bit program memory address on the PAB (in case of change of flow) or two 32-bit data memory addresses (one on each of the XABA and XABB). Each AAU can generate an address to access a byte, a 16-bit word, a 32-bit long word, or a 64-bit two-word long operand in memory to feed into the DALU in a single cycle.

Each AAU can update one address register during one instruction cycle. The modifier control register (MCTL) specifies the type of arithmetic to be used in the address register update calculation. The address arithmetic instructions provide arithmetic operations for address calculations or for general purpose calculations.

The two AAUs are identical. Each contains a 32-bit full adder, called an offset adder, which can perform the following:

- Add or subtract two AGU registers
- Add an immediate value
- Increment or decrement an AGU register
- Add the PC
- Add with reverse-carry

The offset adder can also perform compare or test operations as well as arithmetic and logical shifts. The offset values added in this adder can be pre-shifted left by 1, 2, or 3 bits according to the access width. In reverse-carry mode, the carry propagates in the opposite direction.

A second full adder, called a modulo adder, adds the summed result of the first full adder to a modulo value, M or minus M, where M is stored in the selected modifier register. In modulo mode, a modulo comparator tests whether the result is inside the buffer by comparing the results to the B register, choosing the correct result from the offset adder or the modulo adder.

For more information, see Section 2.3.5, "Arithmetic Instructions on Address Registers."

2.3.2 AGU Programming Model

The programming model of the AGU is shown in Figure 2-13.

The address registers can be programmed for linear addressing, modulo addressing (regular or multiple wrap-around), and reverse-carry addressing. Automatic updating of address registers is available when using address register indirect addressing.

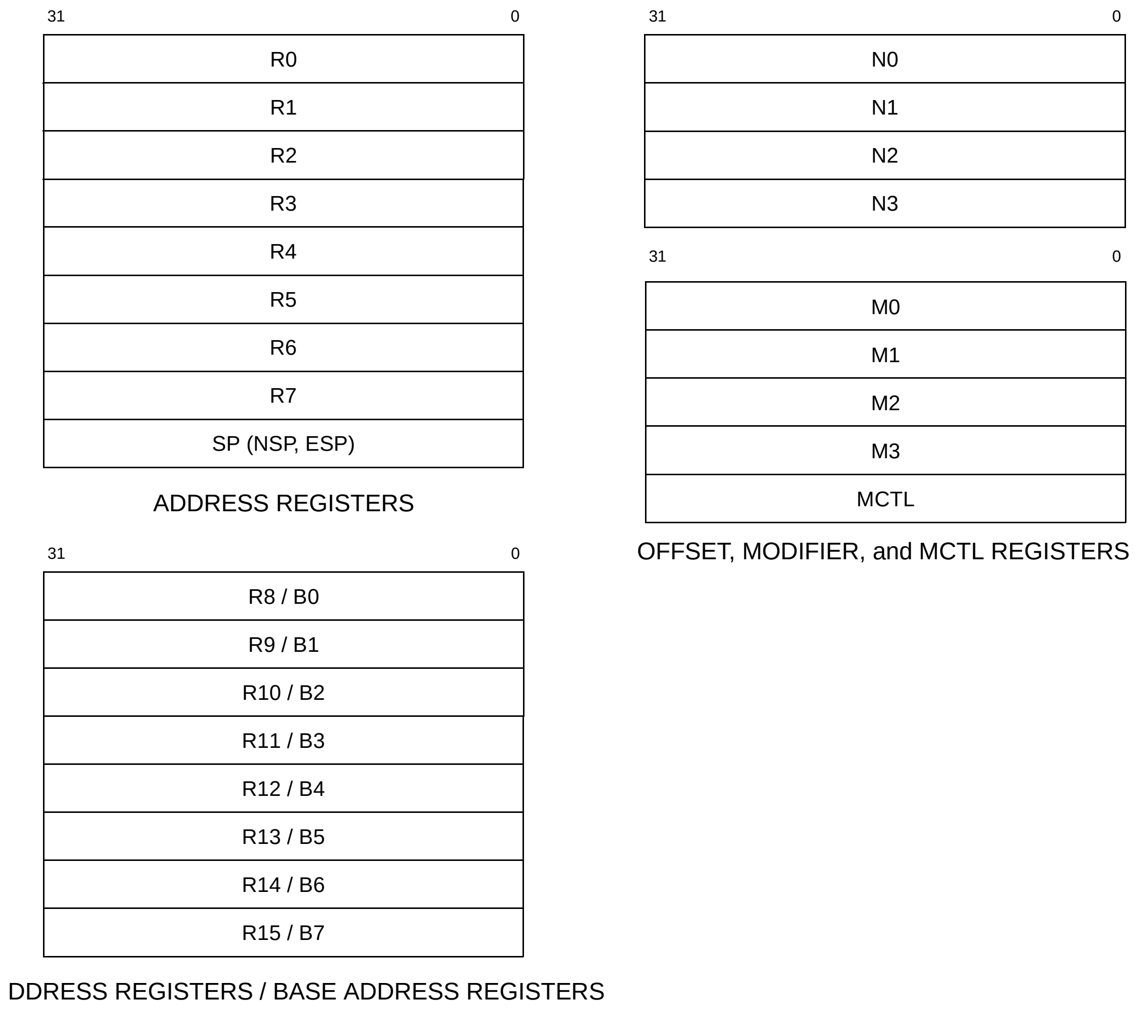


Figure 2-13. AGU Programming Model

2.3.2.1 Address Registers (R0–R15)

The sixteen 32-bit address registers R0–R15 can contain addresses or general-purpose data. These are 32-bit read/write registers. The 32-bit address in a selected address register is used in calculating the effective address of an operand. The contents of an address register can point directly to data, or can be used as an index.

The sixteen address registers R0–R15 are composed of two separate banks, a low bank (R0–R7) and a high bank (R8–R15). The high bank can be used alternatively as a base address register bank (B0–B7). Each address register Rn of the high bank can serve as an address register on condition that the corresponding B_{n-8} register is not used. Both Rn and B_{n-8} are mapped to the same physical register. For example, R8 is available only if R0 is not being used in modulo addressing since this requires the base address register B0.

For future compatibility, SC140 code must not interchange the use of B and R registers. Base modulo registers must use the B notation and pointers must use the R notation. For example, the following sequence is not allowed:

```
MOVE.L #ADDRESS, B0
...
MOVE.W (R8), D0
```

See Section 2.3.2.6, “Modifier Control Register (MCTL),” for further information. The high bank of registers can only be used as pointers in the linear mode of addressing since the other modes of addressing are only encoded for the low bank in the MCTL register.

In addition, an address register can be post-updated according to the addressing mode selected. If an address register is updated, one of the modifier registers (Mj) can be used to specify the type of update arithmetic. Offset registers (Ni) are used for post-incrementing and indexing by offset.

The address register modification can be performed by either of the two AAUs. Most addressing modes modify the selected address register in a read-modify-write fashion. The address register is read, its contents are modified by the associated modulo arithmetic unit, and the register is written with the appropriate output of the AAU. The form of address register modification performed by the address arithmetic unit is controlled by the contents of the offset and modifier registers described in the following sections.

2.3.2.2 Stack Pointer Registers (NSP, ESP)

The SC140 core has two stack pointer registers: the normal stack pointer (NSP) and the exception stack pointer (ESP). These 32-bit registers are used implicitly in all PUSH and POP instructions. Only one stack pointer is active at one time according to the mode:

- In Normal mode, the NSP is used.
- In Exception mode, the ESP is used.

The EXP bit in the status register (SR) determines the active mode. The active stack pointer (SP) is used explicitly for memory references when used with the address register indirect modes. The stack pointers point to the next unoccupied location in the stacks. They are post-incremented on all the implicit PUSH operations and pre-decremented on all the implicit POP operations.

Note: Both stack pointer registers must be initialized explicitly by the programmer after reset.

2.3.2.2.1 Shadow Stack Pointer Registers

Both stack pointers have shadow registers which contain a decremented value of the stack pointers. When the shadow register is not valid, the POP instruction is executed in two cycles. The first cycle is used to decrement the stack pointer. When the shadow register is valid, the POP instruction is executed in only one cycle.

When an SP is written by the AAU register transfer (TFRA), its shadow register automatically becomes invalid. When a PUSH/POP instruction is executed, the shadow register of the active SP becomes valid. As a result, during consecutive POPs, even in the worst case, only the first POP requires an additional cycle.

2.3.2.2.2 Initializing ESP

ESP should be initialized using the AAU register transfer (TFRA) instruction. This guarantees a valid ESP value even if execution of this instruction is interrupted by an exception. The TFRA instruction is considered an address arithmetic operation. The ESP is updated at the address generation pipeline stage, avoiding pipeline conflicts.

2.3.2.3 Offset Registers (N0–N3)

The four 32-bit read/write offset registers N0–N3 can contain offset values used to increment or decrement address registers in address register update calculations. These registers can also be used for 32-bit general purpose storage. For example, the contents of an offset register can specify the offset into a table or the base of the table for indexed addressing, or can be used to step through a table at a specified rate (for example, five locations per step for waveform generation). Each address register can be used with each offset register. For example, R0 can be used with N0, N1, N2, or N3 for offset address calculations. The signed value in an offset register is pre-shifted to the left by 0, 1, 2, or 3 bits to align to the access width.

2.3.2.4 Base Address Registers (B0–B7)

The eight 32-bit read/write base address registers B0–B7 are used in modulo calculations. Each B register is associated with an R register (B0 with R0, and so on). When activating the modulo addressing mode, the B register contains the lower boundary value of the modulo buffer. The upper boundary of the modulo buffer is calculated by B+M-1, where M is the modifier register associated with the R register by MCTL.

When not used for modulo addressing, these registers can be used as high bank address registers (R8–R15). Both Rn and Bn_{-8} share the same physical register. For example, if R0 is not programmed for modulo addressing, the base address register B0 can serve as an additional address register R8.

2.3.2.5 Modifier Registers (M0–M3)

The four 32-bit read/write modifier registers M0–M3 can contain the value of the modulus modifier. These registers can also be used for general-purpose storage. When activating the modulo arithmetic, the contents of Mj specify the modulus. Each low address register can be used with each modifier register as programmed in the MCTL register.

2.3.2.6 Modifier Control Register (MCTL)

The MCTL register is a 32-bit read/write register. This control register is used to program the address mode (AM) for each of the eight low address registers (R0–R7). The addressing mode of the high address register file (R8–R15) cannot be programmed and functions in linear addressing mode only. The format of MCTL is shown in Figure 2-14.

Bit 31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	Bit 16
R7 AM[3:0]				R6 AM[3:0]				R5 AM[3:0]				R4 AM[3:0]			

Bit 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
R3 AM[3:0]				R2 AM[3:0]				R1 AM[3:0]				R0 AM[3:0]			

Figure 2-14. Modifier Control Register (MCTL) Format

The AM bits (AM3, AM2, AM1, AM0) associated with each address register (R0-R7) reflect the address modifier mode of this address register as shown in Table 2-17. Each of the Rn registers can use M0, M1, M2, or M3 as their associated modulo register either in modulo addressing mode, or in multiple wrap-around modulo addressing mode. When activating the modulo addressing mode, the corresponding B register is used to define the lower boundary value (B0 with R0, and so on). The linear or the reverse-carry addressing modes can also be used, freeing the B register to be used as an additional linear address register.

The high bank of the address register file (R8–R15) can only be used in linear addressing mode. Each Rn (n = 8:15) is available only if the corresponding B_{n-8} register is not used since both Rn and B_{n-8} are mapped to the same physical register.

Table 2-17. Address Modifier (AM) Bits

AM3	AM2	AM1	AM0	Address Modifier Modes
0	0	0	0	Linear addressing
0	0	0	1	Reverse-carry addressing
1	0	0	0	M0 used—Modulo addressing
1	0	0	1	M1 used—Modulo addressing
1	0	1	0	M2 used—Modulo addressing
1	0	1	1	M3 used—Modulo addressing
1	1	0	0	M0 used—Multiple wrap-around modulo addressing
1	1	0	1	M1 used—Multiple wrap-around modulo addressing
1	1	1	0	M2 used—Multiple wrap-around modulo addressing
1	1	1	1	M3 used—Multiple wrap-around modulo addressing

MCTL is initialized to zero at reset, setting a default linear mode for all Rn registers. All other AM field combinations are reserved and should not be used.

2.3.3 Addressing Modes

The SC140 core provides four types of addressing modes:

- Register direct
- Address register indirect
- PC relative
- Special

The addressing modes are related to where the operands are to be found and how the address calculations are to be made. These modes are described in the following sections:

2.3.3.1 Register Direct Modes

The register direct addressing modes specify that the operand is in one or more of the DALU registers, AGU registers, or control registers, and are classified as register references.

- **Data or Control Register Direct** — The operand is in one, two, or four DALU registers as specified in a portion of the data bus movement field in the instruction. An example is: `mac d4,d5,d6`, which uses data registers d4, d5, and d6 as sources for the multiply-accumulate operation. This addressing mode is also used to specify a control register operand for special instructions.
- **Address Register Direct** — The operand is in one of the twenty-seven AGU registers (R0–R7, R8–R15/B0–B7, N0–N3, M0–M3, MCTL, N/ESP) specified by a field in the instruction. An example is `addl1a r0,r1`, which performs a 1-bit arithmetic left shift on the data in R0, and adds the result to the data in R1.

2.3.3.2 Address Register Indirect Modes

The address register indirect modes specify that the address register is used to point to a memory location. The term indirect is used because the register contents are not the operand itself, but rather the operand address. These addressing modes specify that an operand is in a memory location and specify the effective address of that operand. These references are classified as memory references. The term “index” refers to an offset stored in a register. The term “displacement” refers to an offset from an immediate in the instruction.

- **No Update, (Rn)** — The operand address is in the address register. The contents of the address register are unchanged by executing the instruction. For R0-R7, the contents of the modifier control register (MCTL) are ignored. An example is: `bmclr.w #$004f,(r4)`. A word is read from memory location stored in r4, operated on, and written back to the same location. The address in r4 is unchanged.
- **Post-increment, (Rn)+** — The operand address is in the address register. After the operand address is used, it is incremented by the access width (1, 2, 4, or 8 bytes) and stored in the same address register. The access width is the number of bytes used by the active instruction on the memory data bus. Incrementing the operand address by the access width places the next available byte address in the register. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. An example is: `move.f (r3)+,d2`. The data in the location identified by the value in r3 is moved to data register d2. Then the value in r3 is incremented by two.

- **Post-decrement, (Rn)-** —The operand address is in the address register. After the operand address is used, it is decremented by the access width (1, 2, 4, or 8 bytes) and stored in the same address register. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. An example is: `move.l (r3)-,d2`. In this case, the value in r3 is decremented by four after the move has taken place.
- **Post-increment by Offset Ni, (Rn) + Ni** — The operand address is in the address register. After the operand address is used, it is incremented or decremented by an amount determined by the signed contents of the Ni register pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The result is stored in the same address register. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Ni register are unchanged. An example is: `move.w d3,(r2)+n3`. The access width is two, so the increment is twice the value in the n3 register.
- **Indexed By Offset N0, (Rn + N0)** — The operand address is the sum of the contents of the address register and the signed contents of the N0 register, pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Rn and N0 registers are unchanged. For example: `move.b  d6,(r3+n0)`. The access width is one, so the contents of the n0 register are used directly to modify the address before the move is done.

  Note that only the N0 offset register can be used in this addressing mode.
- **Indexed by Address Register Rm, (Rn + Rm)** — The operand address is the sum of the contents of the address register Rn and the contents of the address register Rm, pre-shifted to the left by 0, 1, 2, or 3 bits according to the access width. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. The contents of the Rn and Rm registers are unchanged. An example is: `move.l (r0+r2),d6`. Here, the access width is four, so the value in r2 is shifted left two bits before adding to the address in r0.

  Note that only address registers (R0–R7) can be used as Rm.
- **Short Displacement, (Rn + x)** — The operand address is the sum of the contents of the address register Rn and a short displacement x that occupies three bits in the instruction word. The displacement (unsigned) is first shifted to the left by 0, 1, 2, or 3 bits according to the access width. It is then zero-extended to 32 bits and added to Rn to obtain the operand address. Thus, the displacement can range from [0] to [+7] bytes, words, long words, or two long words according to the access width. The contents of the Rn register are unchanged. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register. An example is: `move.l d4,(r3+$1c)`. The access width is four, and the displacement encoded in the instruction is seven (4 x 7 = 28 = $1c).
- **Word Displacement, (Rn + xxxx)** — The operand address is the sum of the contents of the address register Rn and an immediate displacement. The displacement is a signed 15-bit word that requires a second instruction word. It is sign-extended to 32 bits and then added to Rn to obtain the operand address. Thus, the displacement can range from [-16,384] to [+16,383] bytes, [-8192] to [+8191] words, [-4096] to [+4095] long words, or [-2048] to [+2047] two long words according to the access width. The contents of the Rn register are unchanged. The type of arithmetic used for updating R0-R7 is determined by programming the MCTL register.
- **SP Short Displacement, (SP – xx)** — The instruction word contains a 5-bit or 6-bit short unsigned immediate index field. This field is first shifted to the left by 1 or 2 bits according to the access width, then zero-extended to form a 32-bit offset and subtracted from the active stack pointer (NSP in Normal mode, ESP in Exception mode) to obtain the operand address. Thus, the displacement can range from [0] to [31/63] words or long words according to the access width. The contents of the

active SP register are unchanged. The type of arithmetic used is always linear. An example is: `move.w #$ffff,(sp–$3e)`. The encoded displacement is 31,the maximum value of five bits, and the actual displacement is 62 ($3e), since the access width is two.

- **SP Word Displacement, (SP + xxxx)**—The operand address is the sum of the contents of the active stack pointer (SP) and an immediate displacement. The displacement is a signed 15-bit word that requires a second instruction word. It is sign-extended to 32 bits and added to the active stack pointer (NSP in Normal mode, ESP in Exception mode) to obtain the operand address. Thus, the displacement can range from [-16,384] to [+16,383] bytes, [-8192] to [+8191] words, [-4096] to [+4095] long words, or [-2048] to [+2047] two long words according to the access width. The contents of the active SP register are unchanged. The type of arithmetic used is always linear. An example is: `move.l  (sp+$2000),d2.e`. Here, the positive value $2000 is added to the active stack pointer before the memory access.

2.3.3.3 PC Relative Mode

The PC relative address mode is used to calculate the program destination of change-of-flow instructions such as branches (BRA). In the PC relative addressing mode, the instruction encoding contains a signed displacement operand. The operand address is obtained by left-shifting (multiplying by two) the displacement and adding the result to the value of the program counter (PC). The operand is left-shifted because the addresses of the program instructions are word-aligned, and memory addressing is in units of bytes. The arithmetic used is always linear. For example, `bra _label2`. Assume that PC=$0010 and that _label2 is at location $0020. The encoded displacement will be ($0020 – $0010)/2 = $0008.

The number of bits occupied by the displacement in the instruction differs with the different kinds of PC relative instructions. In all cases, the displacement is first sign-extended to 32 bits, then multiplied by two, and added to the PC to obtain the operand address.

In the one-word conditional branch instructions, the displacement occupies 8 bits of the instruction word and can range from [-256] to [254] words. In the one-word unconditional branch instructions, the displacement occupies 10 bits of the instruction word and can range from [-1024] to [1022] words. In the two-word branch instructions, the displacement occupies 20 bits and can range from [-1,048,576] to [1,048,574] words. In the DOSETUP instruction, the displacement occupies 16 bits of the instruction. The displacement for the start address (SA) can range from [-65,536] to [65,534] words.

2.3.3.4 Special Addressing Modes

The special addressing modes do not use an address register when specifying an effective address. They either use an immediate value that is included in the instruction for the data value, such as the data value address, or they use a register that is implicitly referenced by the instruction for the data value.

- **Immediate Short Data** — A 5-bit, 6-bit, or 7-bit operand is part of the instruction operation word. The 5-bit zero-extended operand is used for DALU and AGU arithmetic instructions. The 6-bit zero-extended operand is used for DALU instructions to move short immediate data to an LCn register. The 7-bit sign-extended operand is used for immediate moves to a register. This reference is classified as a program reference. An example is: `doen2 #$3f`. The value $3f, 63, is loaded to loop counter 2.
- **Immediate Word Data** — This addressing mode requires a one-word instruction extension. The immediate data is a 16-bit operand. This reference is classified as a program reference. An example is: `doen2 #$40`. The value 64 is loaded to loop counter 2. The value exceeds the 6-bit limit for immediate short data, so an extra word is needed for the encoding.
- **Immediate Long Data** — This addressing mode requires a two-word instruction extension. The immediate data is a 32-bit operand. This reference is classified as a program reference. An example is: move.l `#$f00d0d01,n0`. The 32-bit unsigned value is moved to the general register n0.
- **Absolute Word Address** — This addressing mode requires a one-word instruction extension. The operand address occupies 16 bits in the instruction operation words, and is zero-extended to form a 32-bit address. This reference is classified as a memory reference. An example is: `move.w ($8a20),d0`.
- **Absolute Long Address** — This addressing mode requires a two-word instruction extension. A 32-bit address is contained in the instruction words. This reference is classified as a memory reference. An example is: `move.w ($34008a20),d0`.
- **Absolute Jump Address** — The operand occupies 32 bits in the instruction operation words. It requires a two-word instruction extension. This reference is classified as a program reference. An example is: `jmp lbl4`, where the instruction is encoded with the program memory address of lbl4.
- **Implicit Reference** — Some instructions make implicit reference to the PC, normal or exception stack, loop registers (SA0, SA1, SA2, SA3, LC0, LC1, LC2, LC3), or status register (SR). These registers are implied by the instruction, and their use is defined by the individual instruction descriptions. An example is: `tfra osp,r2`, which transfers the 32-bit word stored at the other (non-active) stack pointer to address register R2.

2.3.3.5 Memory Access Width

The SC140 core supports variable width access to data memory. With every memory access, the core sends one of four signals to the memory interface to designate whether the access width is 8 bits, 16 bits, 32 bits, or 64 bits wide. The access width is determined by the type of MOVE instruction being used. For example, MOVE.B is used for byte access. MOVE.W is used for word access. For long-word access, MOVE.L, MOVE.2F, and MOVE.2W are used. And, for two long-word access, MOVE.2L, MOVE.4F, and MOVE.4W are used.

The memory addresses are always in units of bytes. For example, addresses for two-word MOVE operations to/from memory are available in multiples of four in order to best align the data with the byte addressing.

Address calculations and register update calculations are performed according to the memory access width as shown in Table 2-18.

Table 2-18. Access Width Support for Address and Register Update Calculations

Addressing Mode	Calculation	Memory Access Width			
		Byte	**Word**	**Long**	**Two Long**
Post-increment (Rn) + Post-decrement (Rn) -	Rn register post-increment or post-decrement by —>	1	2	4	8
Post-increment by Offset (Rn)+Ni	Rn register post-increment by ->	Ni*1	Ni*2	Ni*4	Ni*8
Indexed by Offset N0 (Rn + N0)	Actual address offset	N0	2*N0	4*N0	8*N0
Indexed by Address Register Rm (Rn + Rm)	Actual address offset	Rm	2*Rm	4*Rm	8*Rm
Short Displacement (Rn + x)	Actual address displacement	x	2*x	4*x	8*x
Word Displacement (Rn + xxxx)	Actual address displacement	xxxx	xxxx	xxxx	xxxx
SP update in Push/Pop	SP post-increment or pre-decrement by —>	8	8	8	8
SP Short Displacement (SP - xx)	Actual address displacement	NA	-2*xx	-4*xx	NA
SP Word Displacement	Actual address displacement	xxxx	xxxx	xxxx	xxxx

2.3.3.6 Memory Access Misalignment

Each access to the memory generated by the core should be aligned according to the access type. If the alignment rule is violated, erroneous data may be fetched from the memory. In addition, an exception may be generated to identify that an unaligned access occurred. For more information, see Section 5.7, "Exception Processing," on page 5-41.

Table 2-19 summarizes the memory address alignment rule for each type of memory access.

Table 2-19. Memory Address Alignment

Access Type	Aligned Address
Byte access	Any address
Word access	Multiple of 2
Long-word access	Multiple of 4
Two long-word access	Multiple of 8

2.3.3.7 Addressing Modes Summary

Table 2-20 provides a summary of the addressing modes described in the previous sections. The Operand Reference columns are abbreviated as follows:

- S = Software Stack Reference in data memory (uses NSP or ESP according to mode)
- C = Program Control Unit Register Reference
- D = DALU Register Reference
- A = AGU Register Reference
- P = Program Memory Reference
- X = Data Memory Reference

Table 2-20. Addressing Modes Summary

Addressing Modes	R0-R7 Uses MCTL	Operand Reference						Assembler Syntax
		S	C	D	A	P	X	
Register Direct								
Data or Control Register	—		√	√				Dn Dn Dm Dn Dm Di Dj MCTL PCTL0, PCTL1 SR, EMR, VBA LC0, LC1 LC2, LC3 SA0, SA1 SA2, SA3
Address Register (Rn)	—				√			Rn
Address Modifier Register (Mj)	—				√			Mj
Base Address Register (Bn)	—				√			Bn
Address Offset Register (Ni)	—				√			Ni
Stack Pointer	—				√			SP

Table 2-20. Addressing Modes Summary (Continued)

Addressing Modes	R0-R7 Uses MCTL	Operand Reference						Assembler Syntax
		S	C	D	A	P	X	
Address Register Indirect								
No Update, (Rn)	No						√	(Rn)
Post-increment, (Rn)+	Yes						√	(Rn)+
Post-decrement, (Rn)–	Yes						√	(Rn)–
Post-increment by Offset Ni, (Rn)+Ni	Yes						√	(Rn) + Ni
Indexed by offset N0, (Rn+N0)	Yes						√	(Rn + N0)
Indexed by Address Register Rm, (Rn+Rm)	Yes						√	(Rn + Rm)
Short Displacement, (Rn+x) Word Displacement, (Rn+xxxx)	Yes						√	(Rn + x) (Rn + xxxx)
SP Short Displacement, (SP-xx)	—	√					√	(SP - xx)
SP Word Displacement, (SP+xxxx)	—	√					√	(SP + xxxx)
PC Relative								
PC Relative with Displacement	—					√		#xx (8 bits) #xxx (10 bits) #xxxx (16 bits) #xxxxx (20 bits)
Special								
Immediate Short Data Immediate Word Data Immediate Long Data	—					√		#xx (5, 6, or 7bits) #xxxx (16 bits) #xxxxxxxx(32 bits)
Absolute Word Address Absolute Long Address	—						√	xxxx (16 bits) xxxxxxxx (32 bits)
Absolute Jump Address	—					√		xxxxxxxx (32 bits)
Implicit Reference	—	√	√			√		

Note: The "—" that appears in the "R0-R7 Uses MCTL" heading means that it is not applicable for that addressing mode.

2.3.4 Address Modifier Modes

The AAU supports linear, reverse-carry, modulo, and multiple wrap-around modulo arithmetic types for address register indirect modes operating on R0-R7. These arithmetic types allow the easy creation of data structures in memory for First-In/First-Out (FIFO) queues, delay lines, circular buffers, stacks, and reverse-carry Fast Fourier Transform (FFT) buffers.

Data is manipulated by updating address registers (Rn) used as pointers rather than moving large blocks of data. The contents of the modifier control register MCTL define the type of arithmetic to be performed for address calculations. For modulo arithmetic, the address modifier register Mj specifies the modulus. Each of the address register lower banks (R0–R7) can be used with any of the modifier registers (M0–M3) as programmed in the MCTL register.

2.3.4.1 Linear Addressing Mode

Linear addressing is useful for general-purpose addressing such as stacks. In linear addressing mode, the address is calculated using standard binary arithmetic. The entire memory space is addressable. Linear addressing mode is selected by setting the AM3–0 bits to 0000 in the MCTL register. This is the default state.

2.3.4.2 Reverse-carry Addressing Mode

Reverse-carry addressing is useful for 2^k point FFT addressing. This mode is selected for R0-R7 by setting the AM3-0 bits to 0001 in the MCTL register. Address modification is performed in the hardware by propagating the carry from each pair of added bits in the reverse direction (from the MSB end toward the LSB end). For the +Ni addressing mode, reverse-carry is equivalent to:

- Bit-reversing the contents of Rn (redefining the MSB as the LSB, the next MSB as bit 1, and so on)
- Shifting the offset value in Ni left by 0, 1, 2, or 3 according to the access width
- Bit-reversing the shifted Ni
- Adding normally
- Bit-reversing the result

This address modification is useful for addressing the twiddle factors in 2^k point FFT addressing as well as to unscramble 2^k point FFT data. The range of values for Ni is 0 to 2^{32}-1, which allows reverse-carry addressing for FFTs up to 4,294,967,296 points.

Note: To achieve correct reverse-carry accessing for access widths of 2, 4, or 8, the last 1, 2, or 3 least significant bits (respectively) of the address calculation result are forced to zero.

2.3.4.3 Modulo Addressing Mode

Modulo address modification is useful for creating circular buffers for FIFO queues, delay lines, and sample buffers up to 2^{31} words long.

Modulo addressing is selected by writing the MCTL AM3-0 bits of the MCTL register (as shown in Table 2-10) as well as writing the desired modulus to the corresponding Mj register. Address modification is performed in modulo M, where M ranges from 1 to $+2^{32}$-1. Modulo M arithmetic causes the address register values to remain within an address range of size M, thus defining a buffer with a lower and an upper address boundary.

Each base address register (Bn register) is associated with an Rn register (B0 with R0, and so on). Each register Rn has one Mj register assigned to it by encoding in the MCTL. The lower boundary value of the buffer resides in the Bn register, and the upper boundary is calculated as Bn+Mj-1.

The modulo addressing definition, using a base register (Bn) and a modulo register (Mj), enables the programmer to locate the modulo buffer at any address. The buffer start address is only required to be aligned to the access width.

The address pointer Rn is not required to start at the lower address boundary, nor to end on the upper address boundary. Rn can initially point anywhere (aligned to its access width) within the defined modulo address range, $Bn \le Rn < B+Mj$. Assuming the (Rn)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer (base address + Mj-1), it wraps around through the base address (lower boundary). Alternatively, assuming the (Rn)- indirect addressing mode, if the address decrements past the lower boundary (base address), it wraps around through the base address + Mj-1 (upper boundary).

The following constraints apply:

1. For proper modulo addressing, if an offset Ni is used in the address calculation, the 32-bit absolute effective value |Ni| must be less than or equal to Mj, where “effective” means the programmed Ni is multiplied by the access width. For example, move.w (r0)+n0,d0 translates to the restriction $2*n0 \le Mj$, and move.l (r0)+,d0 translates to $4 \le Mj$. If effective Ni > Mj, the result of the address calculation is undefined. Multiple wrap-around modulo addressing supports the situation of effective Ni greater than Mj.

2. Mj must be aligned to the access width used. For example, if the buffer is used with a MOVE.2L instruction, Mj must be aligned to 8 (be a multiple of 8). If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.

3. When Bn is used as a base address register, the use of R_{n+8} as a pointer is illegal since this is the same physical register.

Modulo addressing is illustrated in Figure 2-15. Addresses will be kept within the eleven addresses shown. For the instruction, `move.w (r0+$000e),d0`, the access will be made from $26 (38), if the base address is $20, the modulus is $c, and r0 is $24. The operation is 36+14=50=38 in modulus 12, base address 32 (50–44 + 32 = 38).

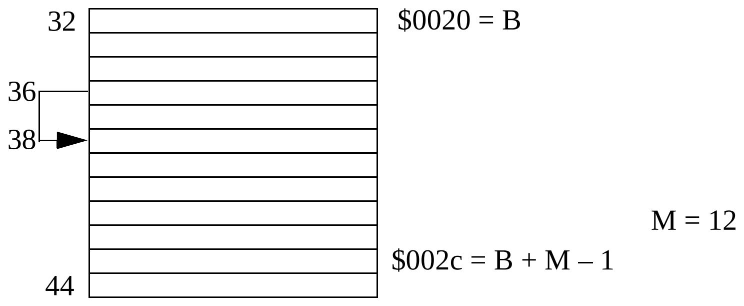


Figure 2-15. Modulo Addressing Example

Table 2-21 describes the modulo register values and the corresponding address calculation.

Table 2-21. Modulo Register Values for Modulo Addressing Mode

Modifier Mj	Address Calculation Arithmetic
\$0000 0000	Unused
\$0000 0001	Modulo 1
\$0000 0002	Modulo 2
\$FFFF FFFE	Modulo 2^{32}-2
\$FFFF FFFF	Modulo 2^{32}-1

2.3.4.4 Multiple Wrap-Around Modulo Addressing Mode

Multiple wrap-around addressing is useful for decimation, interpolation, and waveform generation. The multiple wrap-around capability can be used for argument reduction. In multiple wrap-around modulo addressing mode, the modulus M is a power of 2 in the range of 2^1 to 2^{31}. The value M-1 is stored in the modifier register (Mj). The B registers B0 to B7 are not used for multiple wrap-around modulo addressing; therefore, their corresponding R8–R15 registers can be used for linear addressing.

The lower and upper boundaries are derived from the contents of Mj. The lower boundary (base address) value has zeros in the k LSBs where M = 2^k and therefore must be a multiple of M. The Rn register involved in the memory access is used to set the MSBs of the base address. The base address is set so that the initial value in the Rn register is within the lower and upper boundaries. The upper boundary is the lower boundary plus the modulo size minus one (base address + M–1).

The size of the modulo buffer must be aligned to (be a multiple of) the access width. If the modulus is less than the access width, the data accessed as well as the address calculations are undefined.

If an offset Ni is used in the address calculations, it is not required to be less than or equal to M for proper modulo addressing. The multiple wrap-around modulo addressing mode supports unlimited boundary wraps.

When using the (Rn)+ and (Rn)- addressing modes with a modulus $2^k \geq 8$, there is no functional difference between the multiple wrap-around and normal modulo modes since the address can only be wrapped around once.

As an example, consider the instruction `move.w (r0 + $0042),d0`. If the mctl is set to \$000c, and m0 is set to \$000f, then M0 = 16. If r0 is initially \$24 (36), the lower boundary is \$20 (32) and the upper boundary is \$2f (47). The memory access is done from address \$26 (38), calculated by 36 + 66 = 102, 102–48=54, 54–3x16=6, 6+32=38.

Table 2-22 describes the modulo register Mj values and the corresponding multiple wrap-around address calculation.

Table 2-22. Modulo Register Values for Wrap-Around Modulo Addressing Mode

Modifier Mj	Address Calculation Arithmetic
\$0000 0001	Multiple Wrap-around Modulo 2
\$0000 0003	Multiple Wrap-around Modulo 4
\$0000 0007	Multiple Wrap-around Modulo 8
\$7FFF FFFF	Multiple Wrap-around Modulo 2^{31}
\$FFFF FFFF	Linear

2.3.5 Arithmetic Instructions on Address Registers

The SC140 core provides arithmetic instructions on the address registers (R0–R15), offset registers (N0–N3), the stack pointer (SP), and the program counter (PC).

Address modification modes can affect the arithmetic results stored in R0-R7 using instructions ADDA, SUBA, ADDL1A, or ADDL2A. In addition, an address calculation that increments or decrements address register R0-R7 is affected by the modifier mode. When updating R0-R7 in modulo addressing mode, the modulo registers hold the modulus.

Table 2-23 lists the arithmetic instructions that are executed in the AGU unit. A more detailed description of the operations is provided in Appendix A, "SC140 DSP Core Instruction Set."

Table 2-23. AGU Arithmetic Instructions

Instruction	Description
ADDA	AGU Add (affected by the modifier mode)
ADDL2A	AGU Add with 2-bit left shift of source operand (affected by the modifier mode)
ADDL1A	AGU Add with 1-bit left shift of source operand (affected by the modifier mode)
ASL2A	AGU Arithmetic shift left by 2 bits (32-bit)
ASLA	AGU Arithmetic shift left (32-bit)
ASRA	AGU Arithmetic shift right (32-bit)
CMPEQA	AGU Compare for equal
CMPGTA	AGU Compare for greater than
CMPHIA	AGU Compare for higher (unsigned)
DECA	AGU Decrement register
DECEQA	AGU Decrement and set T if result is zero

Table 2-23. AGU Arithmetic Instructions (Continued)

Instruction	Description
DECGEA	AGU Decrement and set T if result is equal to or greater than zero
INCA	AGU Increment register
LSRA	AGU Logical shift right (32-bit)
SUBA	AGU Subtract (affected by the modifier mode)
SXTA.B	AGU Sign-extend byte
SXTA.W	AGU Sign-extend word
TFRA	AGU Register transfer
TSTEQA	AGU Test for equal to zero
TSTEQA.W	AGU Test for equal to zero on lower 16 bits
TSTGEA	AGU Test for greater than or equal to zero
TSTGTA	AGU Test for greater than zero
ZXTA.B	AGU Zero-extend byte
ZXTA.W	AGU Zero-extend word

2.3.6 Bit Mask Instructions

The SC140 core provides bit mask instructions on all address registers (R0–R15), all DALU registers (D0–D15), all control registers (EMR, VBA, PCTL0, PCTL1, SR, MCTL), and all memory locations.

Bit mask instructions provide an easy way of setting, clearing, inverting, or testing a selected but not necessarily adjacent group of bits in a register or memory location.

All bit mask instructions work on 16-bit data. This data can be the contents of a memory location or a portion (high or low) of a register.

Only a single bit mask instruction is allowed in one execution set since only one execution unit exists for these instructions. A subgroup of the bit mask instructions (BMTSET) supports hardware semaphores. For more information, see Section 2.3.6.1, “Bit Mask Test and Set (Semaphore Support) Instruction.”

Table 2-24 lists the arithmetic instructions that are executed in the BMU.

Table 2-24. AGU Bit Mask Instructions (BMU)

Instruction	Description
AND.W	Logical AND on a 16-bit operand
BMCHG	Bit mask change Inverts every bit in the destination (register or memory) that has the value 1 in the mask.
BMCLR	Bit mask clear Clears every bit in the destination (register or memory) that has the value 1 in the mask.
BMSET	Bit mask set Sets every bit position in the destination (register or memory) that has the value 1 in the mask.
BMTSET	Bit mask test (if set) and set Sets the T bit if every bit that has the value 1 in the mask is 1 in the destination (register or memory). Sets (writes) every bit in the destination (register or memory) that has the value 1 in the mask, and sets the T-bit if the set (write) failed. See Section 2.3.6.1, “Bit Mask Test and Set (Semaphore Support) Instruction.”
BMTSTC	Bit mask test if clear Sets the T-bit, if every bit position that has the value 1 in the mask is 0 in an operand.
BMTSTS	Bit mask test if set Sets the T bit if every bit position that has the value 1 in the mask is 1 in an operand.
EOR.W	Logical exclusive OR on a 16-bit operand
NOT.W	Binary inversion of a 16-bit operand
OR.W	Logical OR on a 16-bit operand

2.3.6.1 Bit Mask Test and Set (Semaphore Support) Instruction

The bit mask test and set instruction (BMTSET) provides support for hardware semaphores. A semaphore is a signal which can be set to indicate whether a program resource can be accessed or not. The destination of this instruction can be a register or a memory location in either internal or external memory. If the semaphore indicates that the resource is available, the T bit has the value 0. If the semaphore indicates that the resource is not available (T = 1), a jump can be made to skip the resource code.

This instruction performs the following tasks:

1. Reads the destination register, tests the data, and sets the T bit, if every bit that has the value 1 in the mask is 1 in the destination.
2. Writes back to the destination a word with ones for the masked bits, and the original destination bits for the unmasked bits.
3. Sets the T bit if the set (write) failed.
   Normally, the BMTSET consists of three indivisible operations: read, update the T bit, and write. A set (write) failed condition occurs if the destination failed to be written indivisibly from the previous read operation of that BMTSET instruction. The memory subsystem signals the core of a write failure if a memory access that is initiated by another master source intervenes between the read and the write accesses of the BMTSET operation. As a result of the non-exclusive write indication, the T bit is set, signalling that the resource may not be available, thereby avoiding a hazard condition.

2.3.6.1.1 Example of Normal Usage of the Semaphoring Mechanism

The following sequence accesses a resource controlled by a semaphore.

```
label : BMTSET.W #mask,(R0)
        JT label
```

Normally, the mask enables only one bit. In this case, the memory destination pointed to by (R0) is read, and the enabled bit is tested. The enabled bit is then set, and the memory destination is written back.

The T bit is set if the enabled bit was originally 1 (meaning that it was semaphore-occupied), or that the write-back failed. A T bit value of TRUE indicates to the conditional jump that the attempt to obtain the resource has failed, and that the jump should be taken. The T bit is cleared if the enabled bit was originally zero. This means that the semaphore was not allocated. Therefore, the resource was available, and the instruction was successful in setting the semaphore exclusively. A successful allocation writeback results.

When the destination is a register, the write is always successful.

2.3.6.2 Semaphore Hardware Implementation

During the address phase of the read and write accesses associated with the BMTSET instruction, an output of the core is asserted. This assertion indicates that the read and the following write are an uninterruptible sequence.

During the data phase of the write access, a core input provides the core with the result of the access (de-asserted = write failed).

2.3.7 Move Instructions

The SC140 instruction set supports various types of move instructions which differ in the following properties:

- Access width — Byte (8 bits), word (16 bits), long-word (32 bits), and two long words (64 bits)
- Data type — Signed integer, unsigned integer, fractional (with or without limiting)
- Multi-register moves — Some move operations split data between two or four registers
- Addressing mode — For example, absolute, relative to an address pointer (with various offset and post-update options), and relative to the stack pointer

The move instructions perform data movement over the XDBA and XDBB buses (for data moves). Move instructions do not affect the status register with the exception of the sticky scaling bit in reading a DALU register.

Table 2-25 lists the move instructions. The suffix just before the period in the MOVE nomenclature indicates the following:

- None = Signed
- U = Unsigned
- S = Scaling and limiting (saturation) enabled

The suffix just after the period in the MOVE nomenclature indicates the following:

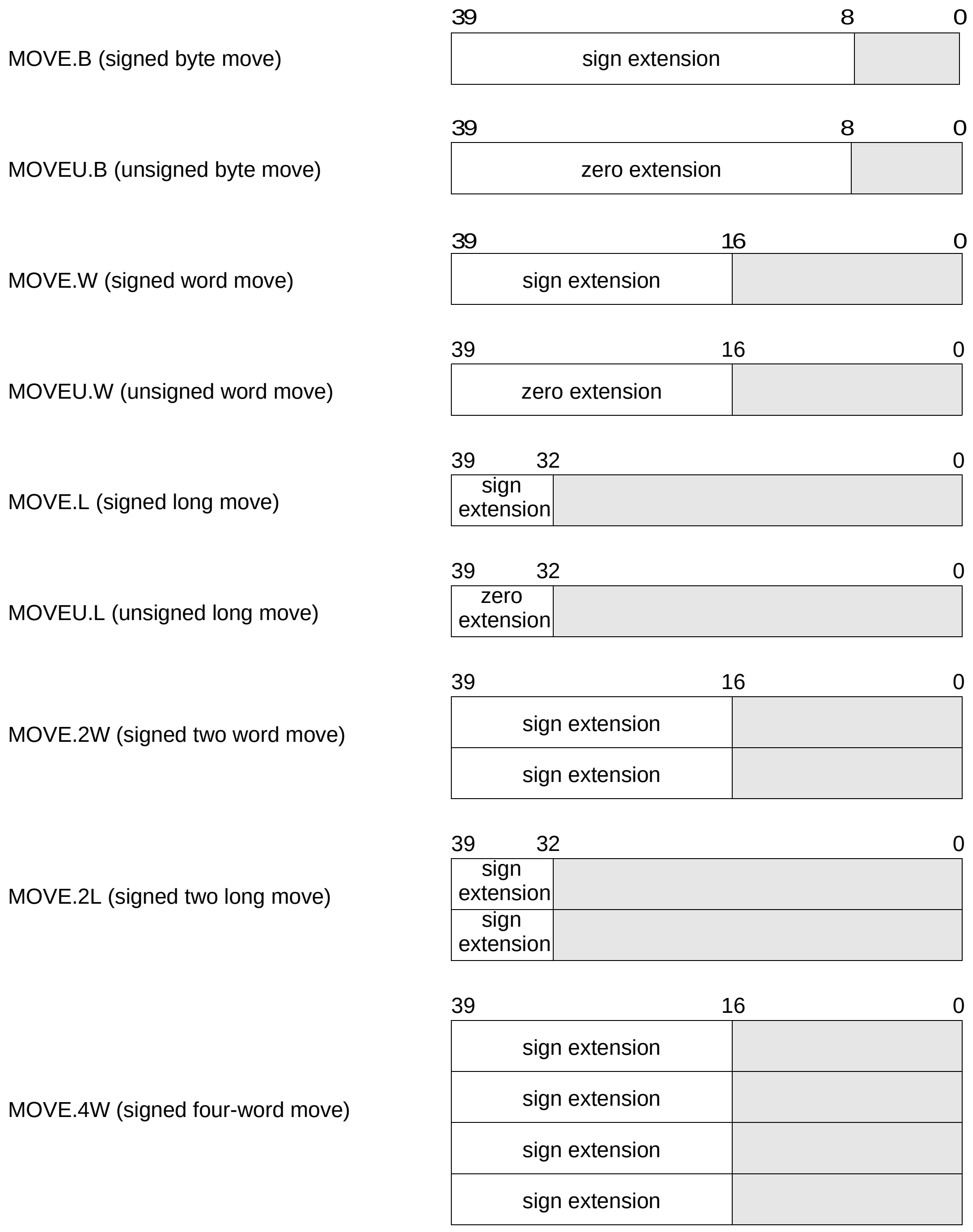
- B = Byte
- W = Integer word (16 bits)
- L = Long word (32 bits)
- F = Fractional word (16 bits)

Either a two or four may modify the last suffix.

Table 2-25. AGU Move Instructions

Instruction	Description
MOVE.2F	Move two fractional words from memory to a register pair
MOVE.2L	Move two longs to/from a register pair
MOVE.2W	Move two integer words to/from memory and a register pair
MOVE.4F	Move four fractional words from memory to a register quad
MOVE.4W	Move four integer words to/from memory and a register quad
MOVE.B	Move byte to/from memory
MOVE.F	Move fractional word to/from memory
MOVE.L	Move long
MOVE.W	Move integer word to/from memory, or immediate to register or memory
MOVEc	Conditional move between address registers
MOVES.2F	Move two fractional words to memory with scaling and limiting enabled
MOVES.4F	Move four fractional words to memory with scaling and limiting enabled
MOVES.F	Move fractional word to memory with scaling and limiting enabled
MOVES.L	Move long to memory with scaling and limiting enabled
MOVEU.B	Move unsigned byte from memory
MOVEU.L	Move unsigned long from immediate
MOVEU.W	Move unsigned integer word from memory or from immediate
VSL.2F	Viterbi shift left—specialized move to support Viterbi kernel
VSL.2W	Viterbi shift left—specialized move to support Viterbi kernel
VSL.4F	Viterbi shift left—specialized move to support Viterbi kernel
VSL.4W	Viterbi shift left—specialized move to support Viterbi kernel

Integer moves from memory (byte, word, long, two long) are right-aligned in the destination register, and by default are sign-extended to the left. Unsigned moves are marked with “U” (for example, MOVEU.B), and zero extended in the destination register. A schematic representation of integer moves from memory into a 40-bit register is shown in Figure 2-16. Moves from registers to memory use the appropriate portion from the source register. Moves to registers of less than 40 bits behave the same as in Figure 2-16 up to their bit length.



MOVE.B (signed byte move)
39
8
0
sign extension
MOVEU.B (unsigned byte move)
39
8
0
zero extension
MOVE.W (signed word move)
39
16
0
sign extension
MOVEU.W (unsigned word move)
39
16
0
zero extension
MOVE.L (signed long move)
39
32
0
sign extension
MOVEU.L (unsigned long move)
39
32
0
zero extension
MOVE.2W (signed two word move)
39
16
0
sign extension
sign extension
MOVE.2L (signed two long move)
39
32
0
sign extension
sign extension
MOVE.4W (signed four-word move)
39
16
0
sign extension
sign extension
sign extension
sign extension

Figure 2-16. Integer Move Instructions

Fractional moves are supported only to DALU registers. Moves from memory are put in the high portion of the data register, sign-extended to the extension, and zero-filled in the low portion. MOVE.L and MOVE.2L may also be considered fractional moves since alignment in the destination register is the same for integer long moves and fractional long moves. A schematic representation of fractional moves from memory to 40-bit data registers is shown in Figure 2-17.

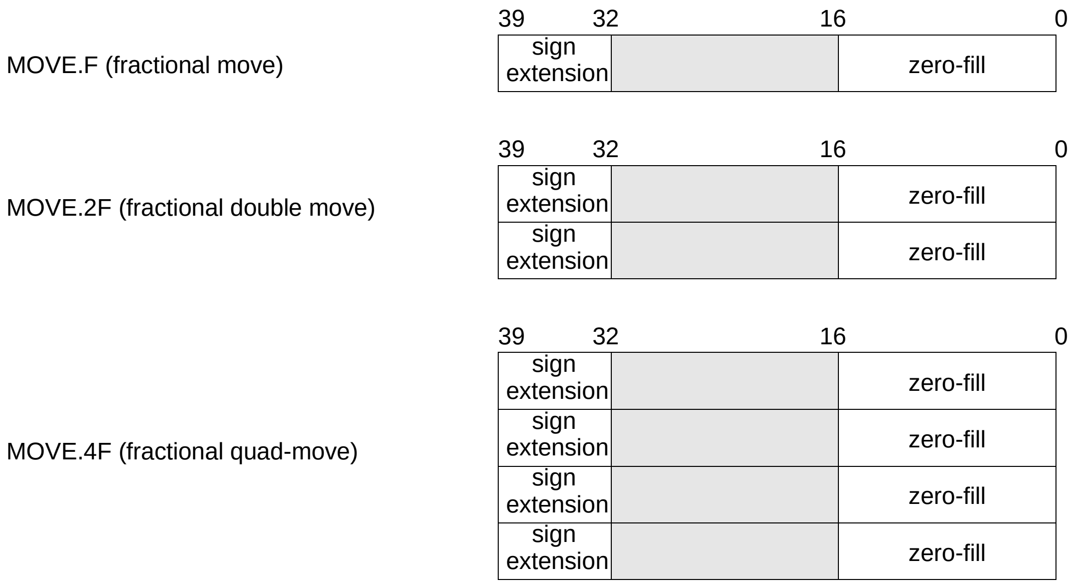

39
32
16
0
MOVE.F (fractional move)
sign extension
zero-fill
MOVE.2F (fractional double move)
MOVE.4F (fractional quad-move)

Figure 2-17. Fractional Move Instructions

The four instructions MOVES.F, MOVES.2F, MOVES.4F, and MOVES.L move data from data registers to the memory with scaling and limiting. The first three operate on 16-bit data. The MOVES.L instruction performs 32-bit scaling and limiting before the move.

For all moves on the SC140, the syntax requires that the source of the data be specified first followed by the destination (SRC, DST). The source and destination are separated by a comma with no spaces either before or after the comma.

Multi-register move instructions originate or update several registers. Registers that are accessed as part of the same move instruction are specified with a colon separator. For example, a MOVE.4F from a memory location pointed by R0 to the registers D0, D1, D2, and D3 is written as:

```
MOVE.4F (R0),D0:D1:D2:D3
```

In this case, let the address in R0 be noted as A0. The fractional word in location A0 then goes to D0, the word in A0 + 2 goes to D1, the word in A0 + 4 goes to D2, and the word in A0 + 6 goes to D3. The addresses increment by two since the addressing unit is always a byte. Moves to or from more than one register are treated according to the same principle.

A special MOVE.L instruction supports moving data to and from data register extensions (Dn.e). In order to support full saving and restoring of the machine state, extension moves also include the limit bit Ln of the register, and are therefore nine bits wide. In one case of the MOVE.L instruction, two extensions belonging to two consecutive data registers are moved concurrently from the registers to the memory as part of a 32-bit access.