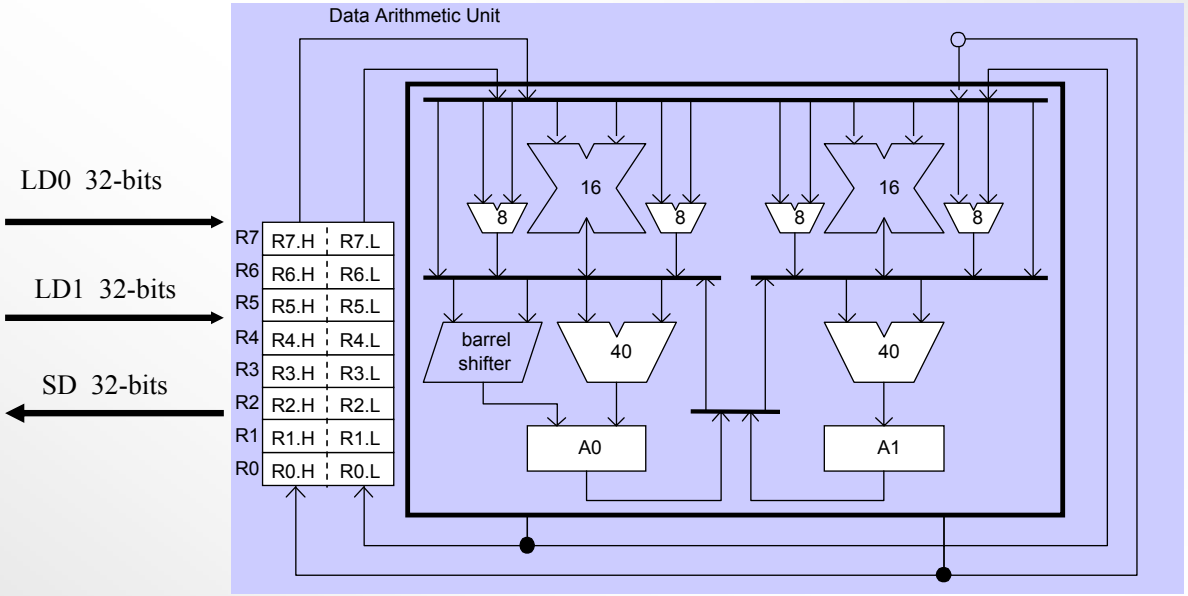
Section 4

Arithmetic Units

ALU

Arithmetic Logic Unit (ALU)



Arithmetic Logic Unit (ALU)

- Two 40-bit ALUs operating on 16-bit, 32-bit, and 40-bit input operands and output 16-bit, 32-bit, and 40-bit results.
- Functions
  - Fixed-point addition and subtraction
  - Addition and subtraction of immediate values
  - Accumulator and subtraction of multiplier results
  - Logical AND, OR, NOT, XOR, bitwise XOR (LFSR), Negate
  - Functions: ABS, MAX, MIN, Round, division primitives
  - Supports conditional instructions
- Four 8-bit video ALUs
  - Explained in more detail as part of Advanced Instructions section

40-bit ALU Operations

- 40-bit ALU operations support the following operations:
  - Single 16-Bit Operations
  - Dual 16-Bit Operations
  - Quad 16-Bit Operations
  - Single 32-Bit Operations
  - Dual 32-Bit Operations

ALU Operations
Single 16-Bit Operations

- **Single 16-bit Addition, Subtraction Operations**
  - Any two 16-bit register halves may be used as inputs.
  - One 16-bit result is deposited in designated 16-bit register half.
  - Must specify saturation option (s) or (ns)

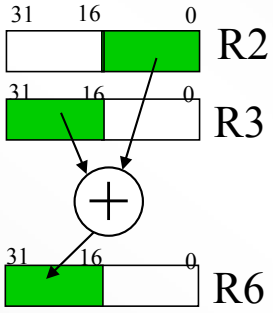
- **General Form:**

  Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi (sat_flag);

Example:

R6.H = R3.H + R2.L (s);

Single 16-bit addition



ALU Operations
Dual 16-Bit Operations

- Dual 16-bit Addition, Subtraction Operations
  - Any two 32-bit registers may be used as inputs.
  - Two 16-bit results are deposited in designated 32-bit register.
- General Form:

  Dreg = Dreg +|+ Dreg [(opt_mode_0)];

  Dreg = Dreg -|- Dreg [(opt_mode_0)];

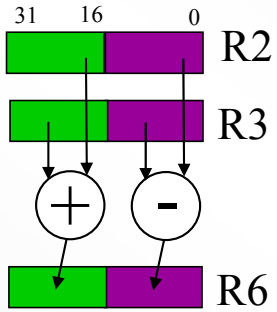
  Dreg = Dreg +|- Dreg [(opt_mode_0)];

  Dreg = Dreg -|+ Dreg [(opt_mode_0)];

Example:

R6 = R2 +|- R3;

Dual
16-bit addition



ALU Operations
Quad 16-Bit Operations

- **Quad 16-bit Addition, Subtraction Operations**
  - **Any two 32-bit registers may be used as inputs.**
  - **Four 16-bit results are deposited in two designated 32-bit registers.**
- **General Form:**

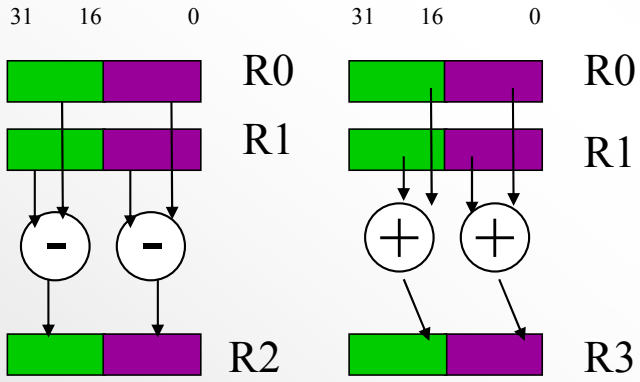
  Dreg = Dreg +|+ Dreg, Dreg = Dreg -|- Dreg [(opt_mode_0, opt_mode_2)];

  Dreg = Dreg +|- Dreg, Dreg = Dreg -|+ Dreg [(opt_mode_0, opt_mode_2)];

Example:

R3 = R0 + | + R1, R2 = R0 - | - R1;

Quad
16-bit addition



ALU Operations
Single 32-Bit Operations

- **Single 32-bit Addition, Subtraction Operations**
  - Any two 32-bit registers may be used as inputs.
  - One 32-bit result is deposited in designated 32-bit register.
  - Optional saturation flag
- **General Form:**

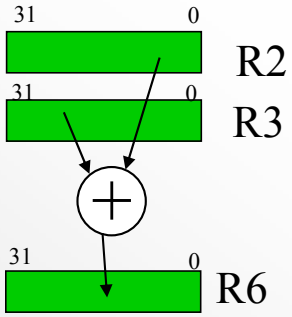
  Dreg = Dreg + Dreg [(sat_flag)];

  Dreg = Dreg – Dreg [(sat_flag)];

Example:

R6 = R2 + R3;

32-bit addition



ALU Operations
Dual 32-Bit Operations

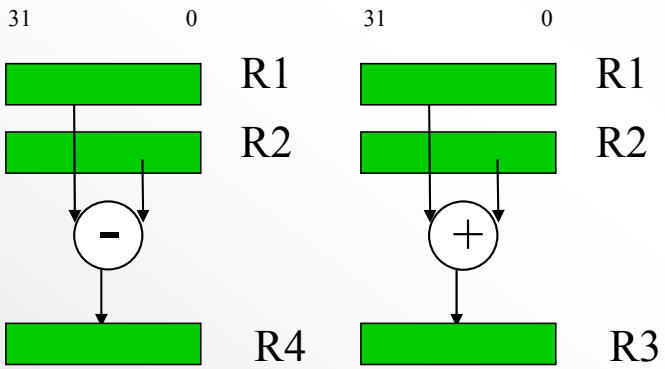
- **Dual 32-bit Addition, Subtraction Operations**
  - **Any two 32-bit registers may be used as inputs.**
  - **Two 32-bit result is deposited in designated 32-bit register.**
- **General Form:**

  Dreg = Dreg + Dreg, Dreg = Dreg – Dreg [(opt_mode_1)];

Example:

R3 = R1 + R2, R4 = R1 - R2;

Dual
32-bit operation



ALU Operation
Options and Examples

The Vector Add / Subtract instruction provides three option modes.

- ***opt_mode_0*** **supports the Dual and Quad 16-Bit Operations versions of this instruction.**
- ***opt_mode_1*** **supports the Dual 32-bit and 40-bit operations.**
- ***opt_mode_2*** **supports the Quad 16-Bit Operations versions of this instruction.**

Mode	Option	Description
opt_mode_0	S	Saturate the results at 16 bits.
	CO	Cross option. Swap the order of the results in the destination register.
	SCO	Saturate and cross option. Combination of (S) and (CO) options.
opt_mode_1	S	Saturate the results at 16 or 32 bits, depending on the operand size.
opt_mode_2	ASR	Arithmetic shift right. Halve the result (divide by 2) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.
	ASL	Arithmetic shift left. Double the result (multiply by 2, truncated) before storing in the destination register. If specified with the S (saturation) flag in Quad 16-Bit Operand versions of this instruction, the scaling is performed before saturation.

Examples:

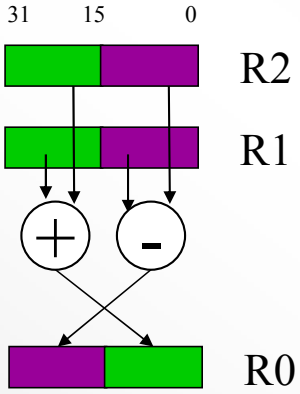
```
R6 = R0 -|+ R1 (s);
R7 = R3 -|- R6 (SCO);
```

ALU Operations
Dual 16-Bit Cross Options

- High result is placed in the low half of designated result register.
- Low result is placed in the high half of designated result register.

Example:

R0 = R2 +|- R1 (CO);



Rounding Instructions

The Round to Half-Word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction number by extracting and saturating bits 31–16, then discarding bits 15–0. The instruction supports only biased rounding, which adds a half LSB (in this case, bit 15) before
truncating bits 15–0. The ALU performs the rounding. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

Fractional data types such as the operands used in this instruction are always signed.

General Form
dest_reg = src_reg (RND)

Syntax
Dreg_lo_hi =*Dreg* (RND) ; /* round and saturate the source to 16 bits. (b) */

Example

```
/* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd) ; // . . . produces r1.l = 0xFFFD
// If r7 = 0x0001 8000, then rounding . . .
r1.h = r7 (rnd) ; // . . . produces r1.h = 0x0002
```

Other ALU Operations
Pointer Register Example Instructions

- **P5 = P3 + P0;** // add two 32-bit pointer registers
- **P5 += -4;** // add immediate value to P register

32-bit ALU Logical Operations

- **AND**
  General Form:
  Dreg = Dreg & Dreg;
  Example:
  R4 = R4 & R3;

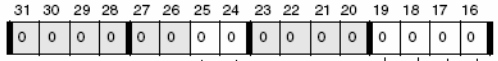
- **NOT**
  General Form:
  Dreg = ~Dreg;
  Example:
  R3 = ~ R4;

- **OR**
  General Form:
  Dreg = Dreg | Dreg;
  Example:
  R4 = R4 | R3;

- **XOR**
  General Form:
  Dreg = Dreg ^ Dreg;
  Example:
  R4 = R4 ^ R3;

ASTAT Register

Arithmetic Status Register (ASTAT)



Reset = 0x0000 0000

VS (Sticky Dreg Overflow)
Sticky version of V

V (Dreg Overflow)
0 - Last result written from ALU to Data Register File register has not overflowed
1 - Last result has overflowed

AV1S (Sticky A1 Overflow)
Sticky version of AV1

AV0 (A0 Overflow)
0 - Last result written to A0 has not overflowed
1 - Last result written to A0 has overflowed

AV0S (Sticky A0 Overflow)
Sticky version of AV0

AV1 (A1 Overflow)
0 - Last result written to A1 has not overflowed
1 - Last result written to A1 has overflowed

AC1 (ALU1 Carry)
0 - Operation in ALU1 does not generate a carry
1 - Operation generates a carry

AC0 (ALU0 Carry)
0 - Operation in ALU0 does not generate a carry
1 - Operation generates a carry

RND_MOD (Rounding Mode)
0 - Unbiased rounding
1 - Biased rounding

AQ (Quotient)
Quotient bit

AZ (Zero Result)
0 - Result from last ALU0, ALU1, or shifter operation is not zero
1 - Result is zero

AN (Negative Result)
0 - Result from last ALU0, ALU1, or shifter operation is not negative
1 - Result is negative

AC0_COPY
Identical to bit 12

V_COPY
Identical to bit 24

CC (Condition Code)
Multipurpose flag, used primarily to hold resolution of arithmetic comparisons. Also used by some shifter instructions to hold rotating bits.

ALU Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Preg = Preg + Preg ;	–	–	–	–	–	–	–
Preg += Preg ;	–	–	–	–	–	–	–
Preg -= Preg ;	–	–	–	–	–	–	–
Dreg = Dreg + Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg – Dreg (S) ;	*	*	*	–	–	*	–
Dreg = Dreg + Dreg, Dreg = Dreg - Dreg ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	–	–	*	–
Dreg_lo_hi = Dreg_lo_hi - Dreg_lo_hi (S) ;	*	*	*	–	–	*	–
Dreg = Dreg +|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|+ Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg –|– Dreg ;	*	*	*	–	–	*	–
Dreg = Dreg +|+Dreg, Dreg = Dreg –|– Dreg ;	*	*	–	–	–	*	–
Dreg = Dreg +|– Dreg, Dreg = Dreg –|+ Dreg ;	*	*	–	–	–	*	–
Dreg = An + An, Dreg = An – An ;	*	*	*	–	–	*	–
Dreg += imm7 ;	*	*	*	–	–	*	–
Preg += imm7 ;	–	–	–	–	–	–	–
Dreg= (A0 += A1) ;	*	*	*	*	–	*	–
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	–	*	–
A0 += A1 ;	*	*	*	*	–	–	–
A0 –= A1 ;	*	*	*	*	–	–	–
DIVS (Dreg, Dreg) ;	*	*	*	*	–	–	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	–	–	d
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–

ALU Instruction Summary

Instruction	ASTAT Status Flags						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	–	–	–	**/–	–
Dreg = ABS Dreg (V) ;	*	**	–	–	–	*	–
An = ABS An ;	*	**	–	*	*	*	–
An = ABS An, An = ABS An ;	*	**	–	*	*	*	–
An = –An ;	*	*	*	*	*	*	–
An = –An, An =– An ;	*	*	*	*	*	*	–
An = An (S) ;	*	*	–	*	*	–	–
An = An (S), An = An (S) ;	*	*	–	*	*	–	–
Dreg_lo_hi = Dreg (RND) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg - Dreg (RND12) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo_hi = Dreg - Dreg (RND20) ;	*	*	–	–	–	*	–
Dreg_lo = SIGNBITS Dreg ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS Dreg_lo_hi ;	–	–	–	–	–	–	–
Dreg_lo = SIGNBITS An ;	–	–	–	–	–	–	–
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V) ;	–	–	–	–	–	–	–
Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);	–	–	–	–	–	–	–
Dreg = Dreg & Dreg ;	*	*	**	–	–	**/–	–
Dreg = ~ Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg | Dreg ;	*	*	**	–	–	**/–	–
Dreg = Dreg ^ Dreg ;	*	*	**	–	–	**/–	–
Dreg =– Dreg ;	*	*	*	–	–	*	–

Conditional Code (CC) Bit in ASTAT

- CC bit is used in several instructions
  - Action taken in the instruction depends on the value of CC
  - If cc jump here; //if cc = 1, jump to label “here”
  - If cc r3 = r0; // perform move if cc=1

- CC bit value is based on a comparison of two registers, pointers or accumulators
- CC bit can be moved to and from a data register or ASTAT bit
- CC bit can be negated

CC Bit Instructions

General Syntax for Data/Pointer Register Compare Operations

CC = operand_1 == operand_2
CC = operand_1 < operand_2
CC = operand_1 <= operand_2
CC = operand_1 < operand_2 (IU)
CC = operand_1 <= operand_2 (IU)

Examples

CC = *Dreg* == *Dreg* ; /* equal, register, signed (a) */
CC = *Dreg* == *imm3* ; /* equal, immediate, signed (a) */

CC = *Preg* == *Preg* ; /* equal, register, signed (a) */
CC = *Preg* == *imm3* ; /* equal, immediate, signed (a) */

General Syntax for Accumulator Compare Operations

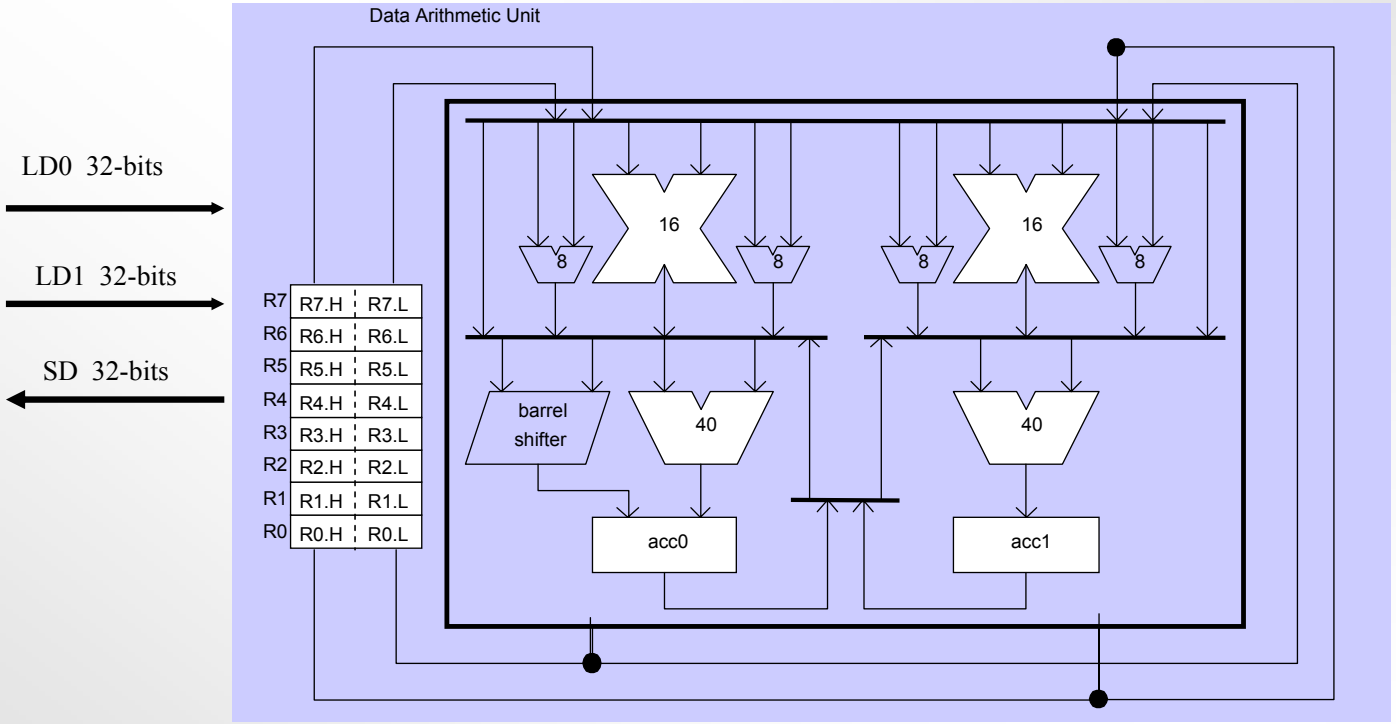
CC = A0 == A1
CC = A0 < A1
CC = A0 <= A1

ALU Exercise

LAB 3

Multiply-Accumulators (MAC)

Multiply-Accumulators (MAC)

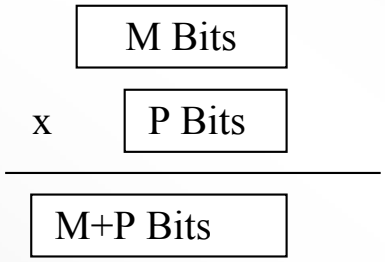


Multiply-Accumulators (MAC)

- Two identical MACs
  - Each can perform fixed point multiplication and multiply-and-accumulate operations on 16-bit fixed point input data and outputs 32-bit or 40-bit results depending the destination.
- Functions
  - Multiplication
  - Multiply-and-accumulate with addition (optional rounding)
  - Multiply-and-accumulate with subtraction (optional rounding)
  - Dual versions of the above
- Features
  - Saturation of accumulator results
  - Optional rounding of multiplier results

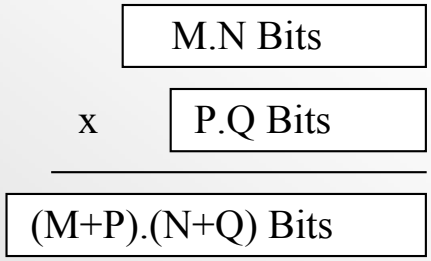
Placement of Binary Point in Multiplication

- Binary Integer Multiplication



Example: 16.0 x 16.0 => 32.0

- Mixed/Fractional Multiplication

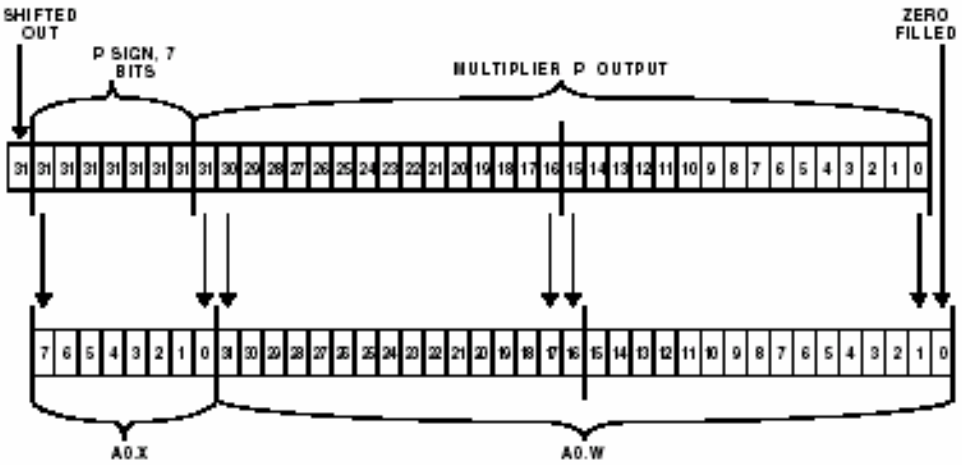


Example: 1.15 x 1.15 => 2.30**
4.12 x 1.15 => 5.27

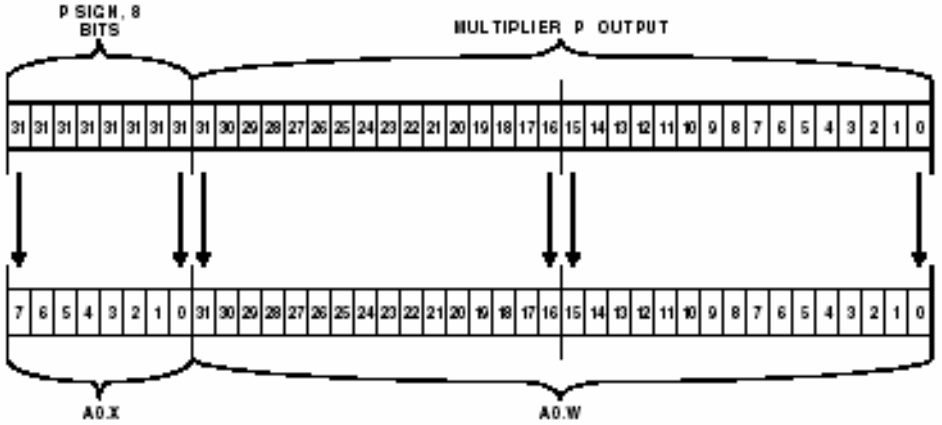
** In fractional mode the result of a multiplication will be automatically left shifted by 1 bit resulting in a 1.31 format

Multiplier Results

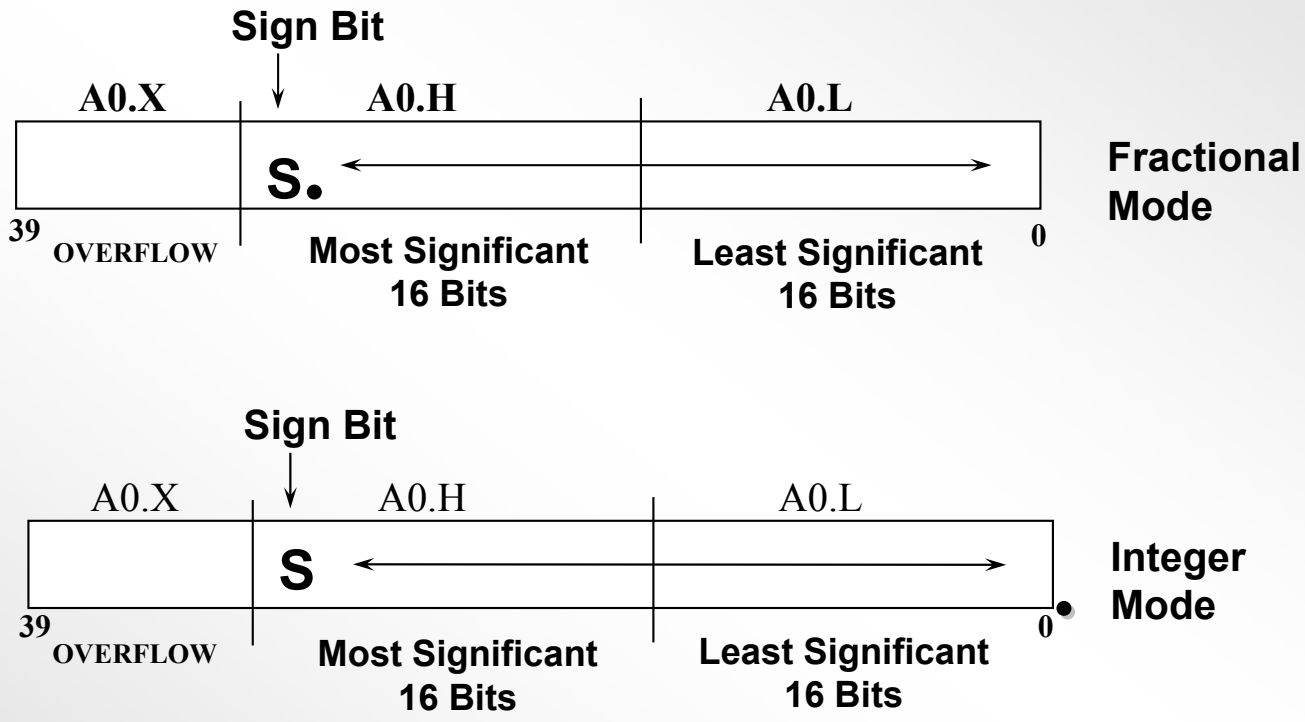
Fractional mode



Integer mode



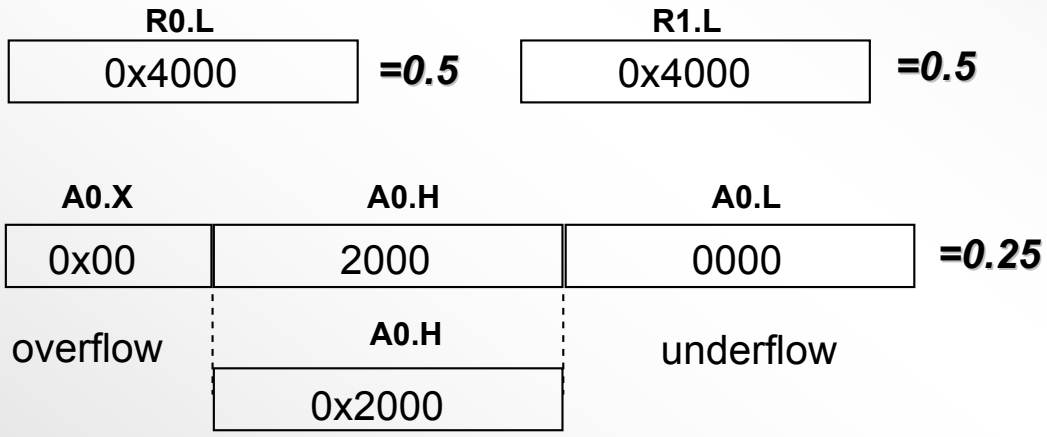
Placement of Binary Point in A0



Multiplication Modes -- Fractional Mode

Mode 1: fractional mode

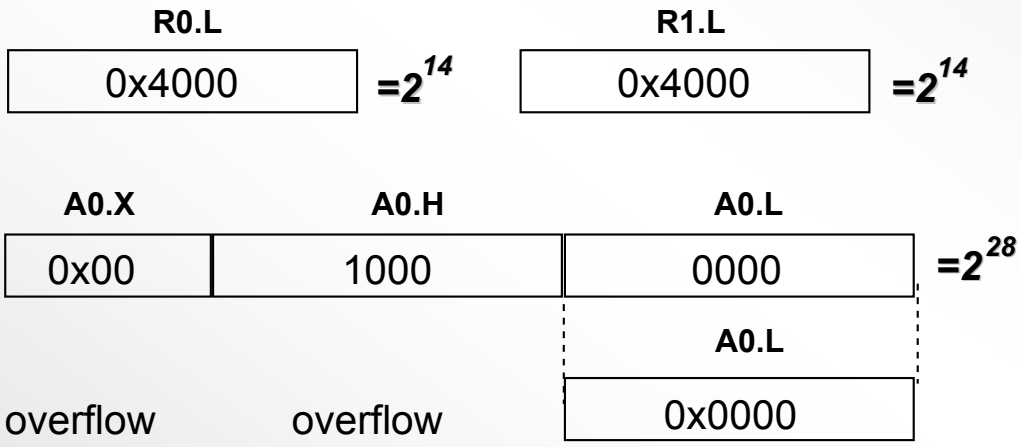
- Multiplier assumes all numbers in a 1.15 format
- Multiplier automatically shifts product 1-bit left before accumulation (Result forced to 1.31 format)
- Example: A0 = R0.L * R1.L;



Multiplication Modes -- Integer Mode

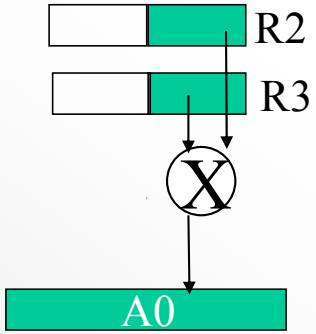
Mode 2: integer mode

- Multiplier assumes all numbers in a 16.0 format
- No automatic left-shift necessary
- Example: A0 = R0.L * R1.L (IS);

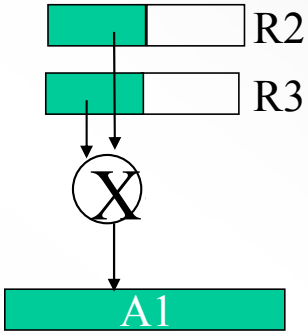


Multiply Operations
Example Instructions

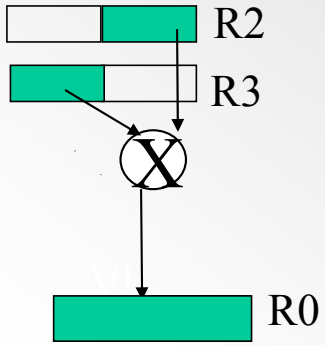
A0 = R2.L * R3.L;



A1 = R2.H * R3.H;

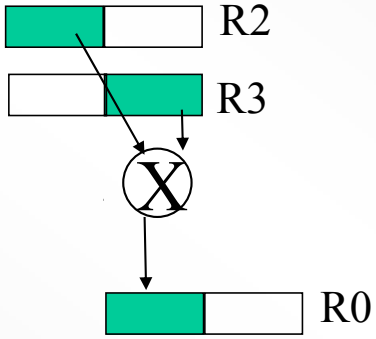


R0 = R2.L * R3.H;



- **Example input operand combinations**
- **Accumulator or data register or half-register can be the destination**

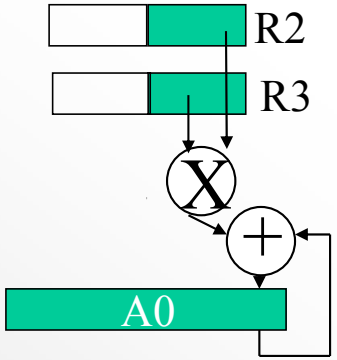
R0.H = R2.H * R3.L;



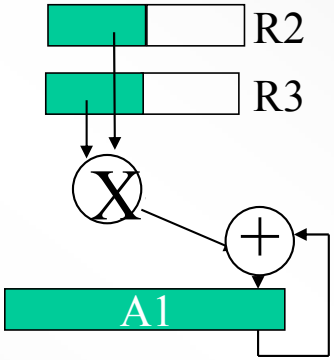
Multiply and Accumulate Operations
Example Instructions

- **Example input operand combinations**

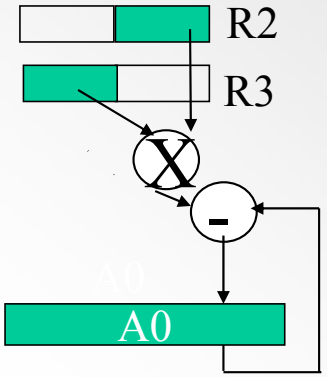
A0 += R2.L * R3.L;



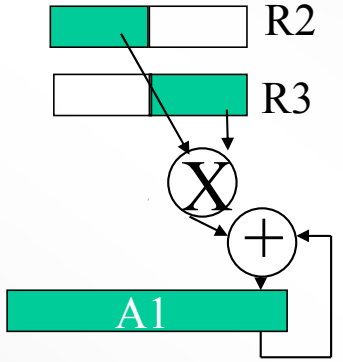
A1 += R2.H * R3.H;



A0 -= R2.L * R3.H;

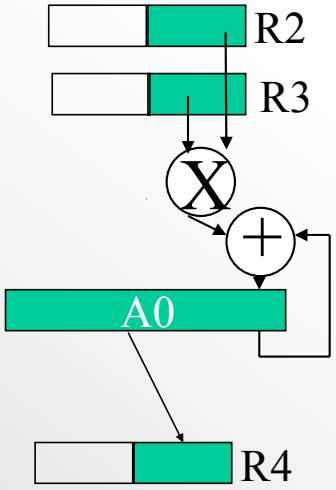


A1 += R2.H * R3.L;

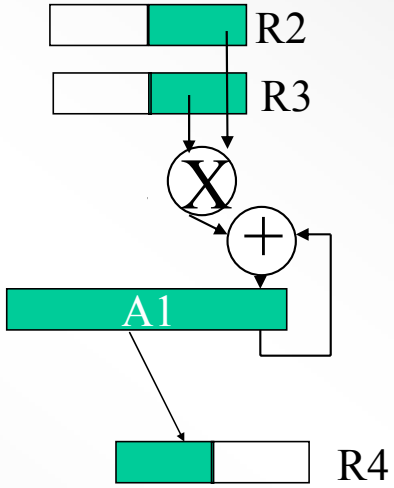


Multiply and MAC Operations
When Result is Transferred From the Accumulator to a 16-bit Data Register

R4.L = (A0 += R2.L * R3.L);

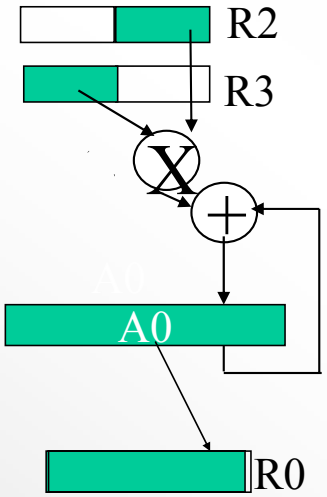


R4.H = (A1 += R2.L * R3.L);



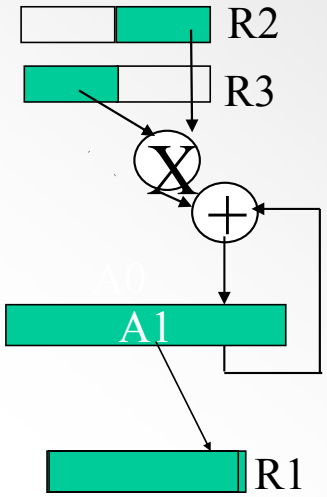
Multiply and MAC Operations
When Result is Transferred From the Accumulator to a 32-bit Data Register

R0 = (A0 += R2.L * R3.H);



When A0 is used, the destination must be to an even Data Register, e.g. R0, R2, R4, R6

R1 = (A1 += R2.L * R3.H);



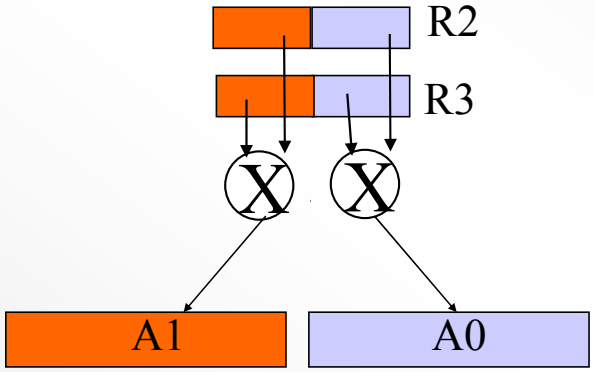
When A1 is used, the destination must be to an odd Data Register, e.g. R1, R3, R5, R7

In both cases, the accumulate can be removed or replaced by a subtraction

Dual Multiply Operations
Example Instruction

- **Both Multipliers can be used in the same operation to double the throughput. The same two 32-bit input registers must be used.**

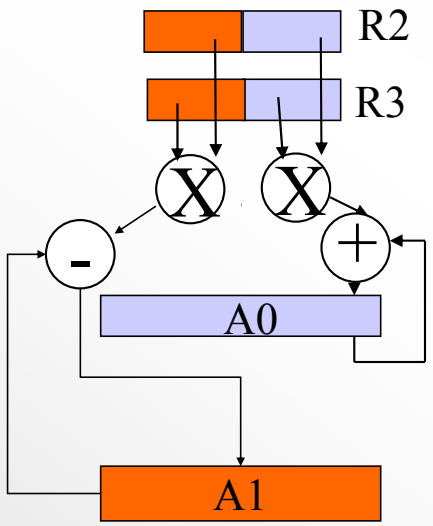
A1 = R2.H * R3.H, A0 = R2.L * R3.L;



Dual MAC Operations
Example Instruction

- **Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers must be used (R2 and R3 in this example).**

A1 -= R2.H * R3.H, A0 += R2.L * R3.L;

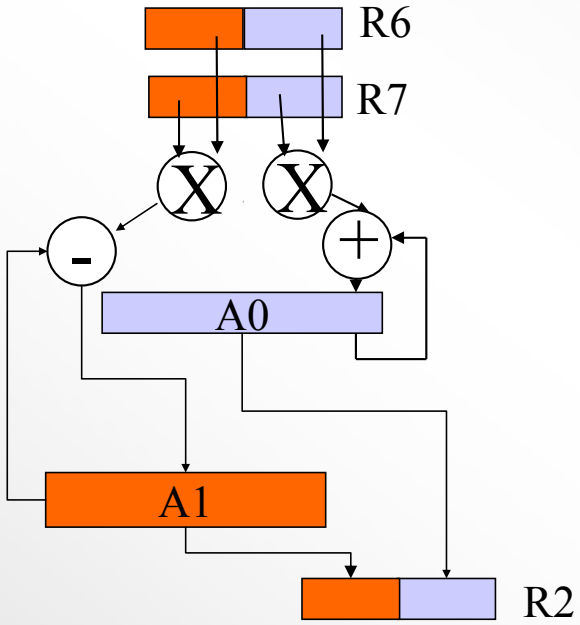


In both cases, the accumulate and subtraction are interchangeable

Dual MAC Operations
With Destination of Two 16-bit Data Registers

- **Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers must be used (R6 and R7 in this example).**

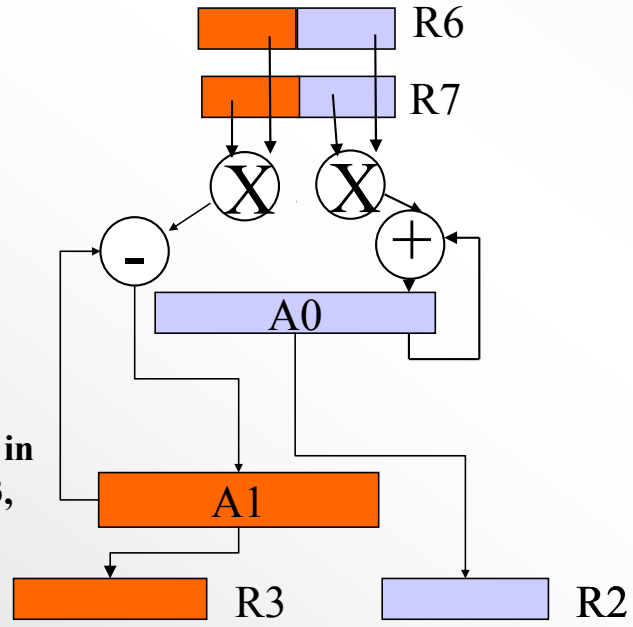
R2.H = (A1 += R7.H * R6.H), R2.L = (A0 += R7.L * R6.L);



Dual MAC Operations
With Destination of Two 32-bit Data Registers

- **Both MACs can be used in the same operation to double the MAC throughput. The same two 32-bit input registers must be used.**

R3 = (A1 += R7.H * R6.H), R2 = (A0 += R7.L * R6.L);

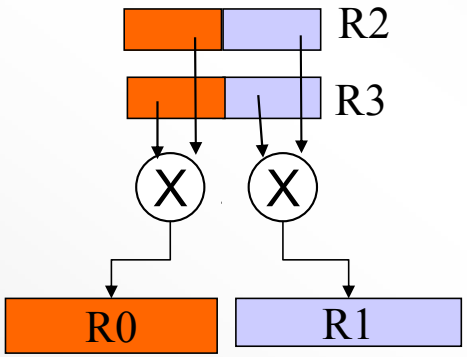


32-bit Data Register Destinations must be used in pairs, e.g. R0:R1 or R2:R3, R4:R5 or R6:R7

Dual Multiply Operations With Destination of Two 32-bit Data Registers

- **Both Multipliers can be used in the same operation to double the throughput. The same two 32-bit input registers must be used.**

R0 = R2.H * R3.H, R1 = R2.L * R3.L;



32-bit Data Register Destinations must be used in pairs, e.g. R0:R1 or R2:R3 or R4:R5 or R6:R7

16-bit Multiplier Options

Option	Description
default	Both operands of both MACs are treated as signed fractions with left-shift correction to normalize the fraction.
(FU)	Unsigned fraction operands. No shift correction.
(IS)	Signed integer operands. No shift correction.
(IU)	Unsigned integer operands. No shift correction. Available only for the 16-bit destination versions of this instruction.
(T)	Signed fraction operands. Truncate the result to 16 bits when copying to the destination half register. Available only for the 16-bit destination versions of this instruction.
(TFU)	Unsigned fraction operands. Truncate the result to 16 bits when copying to the destination half register. Available only for the 16-bit destination versions of this instruction.

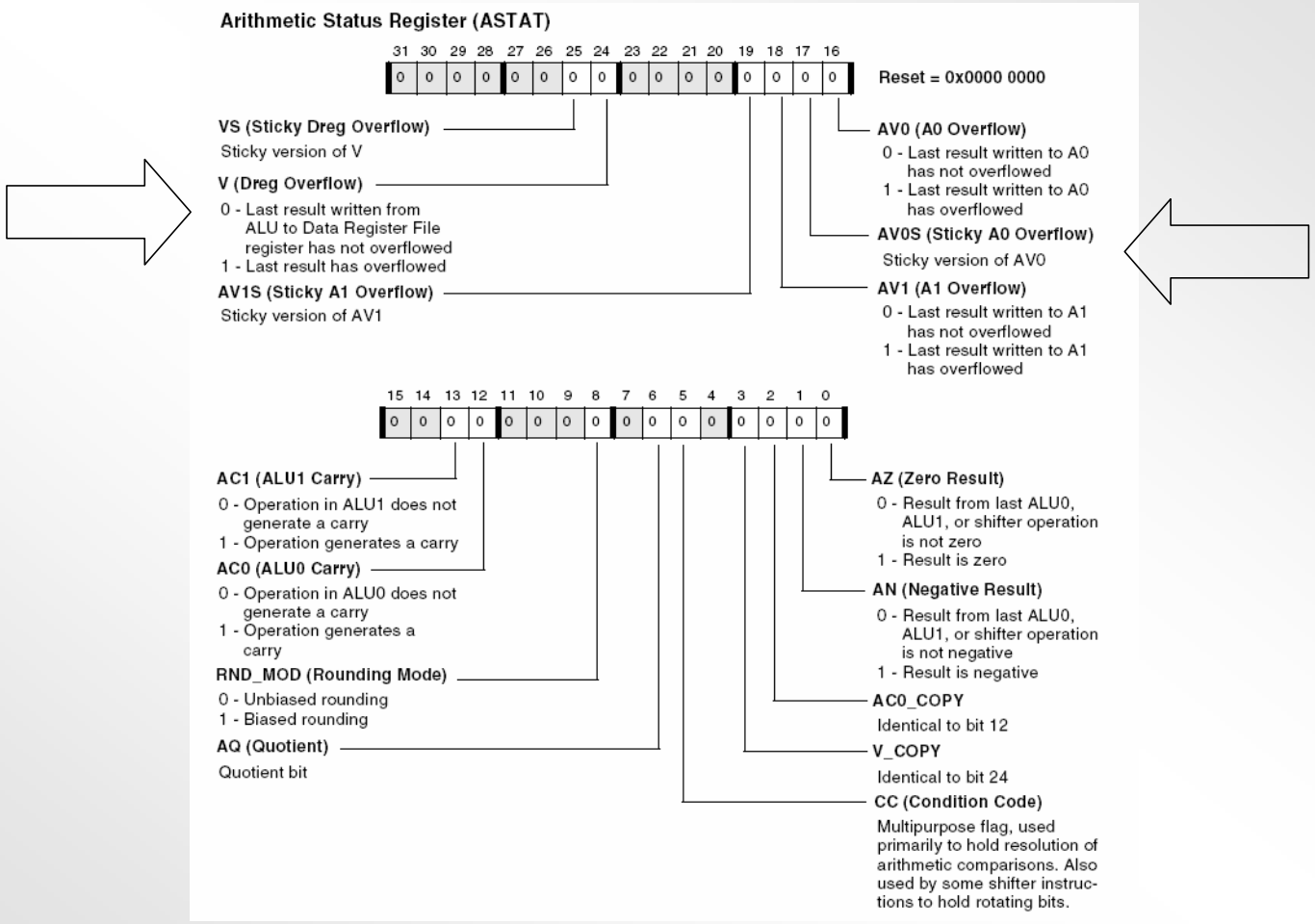
Option	Description
(S2RND)	Signed fraction operands with left-shift correction to normalize the fraction. Scale the result (multiply x2 by performing a one-place shift left) when copying to the destination half register. If scaling produces a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value.
(ISS2)	Signed integer operands. Scale the result (multiply x2 by performing a one-place shift left) when copying to the destination half register. If scaling produces a signed value larger than 16 bits, the number is saturated to its maximum positive or negative value.
(IH)	Integer multiplication with high half-word extraction. The result is saturated at 32 bits and bits 31–16 of that value are copied into the destination half register. Available only for the 16-bit destination versions of this instruction.
(M)	Mixed multiply mode. MAC1 multiplies a signed fraction by an unsigned fraction operand with no left-shift correction. Src_reg_0 is signed and src_reg_1 is unsigned. MAC0 performs an unmixed multiply on signed fractions by default or another format as specified. The (M) option can be used alone or in conjunction with one other format option, but only with MAC1 versions of this instruction. When used together, the option flags must be enclosed in one set of parenthesis and separated by a comma. Example: (M, IS).

Unbiased and Biased Rounding

- **Unbiased Rounding: Returns number closest to the original number**
  - When it lies exactly halfway between 2 numbers …
    - The nearest even number is returned
- **Biased Rounding: Returns number closest to the original number**
  - When it lies exactly halfway between 2 numbers …
    - The larger of the numbers is returned

- RND_MOD (bit 8 of ASTAT register) set to “1” enables biased rounding

ASTAT Register



Multiplier Instruction Summary

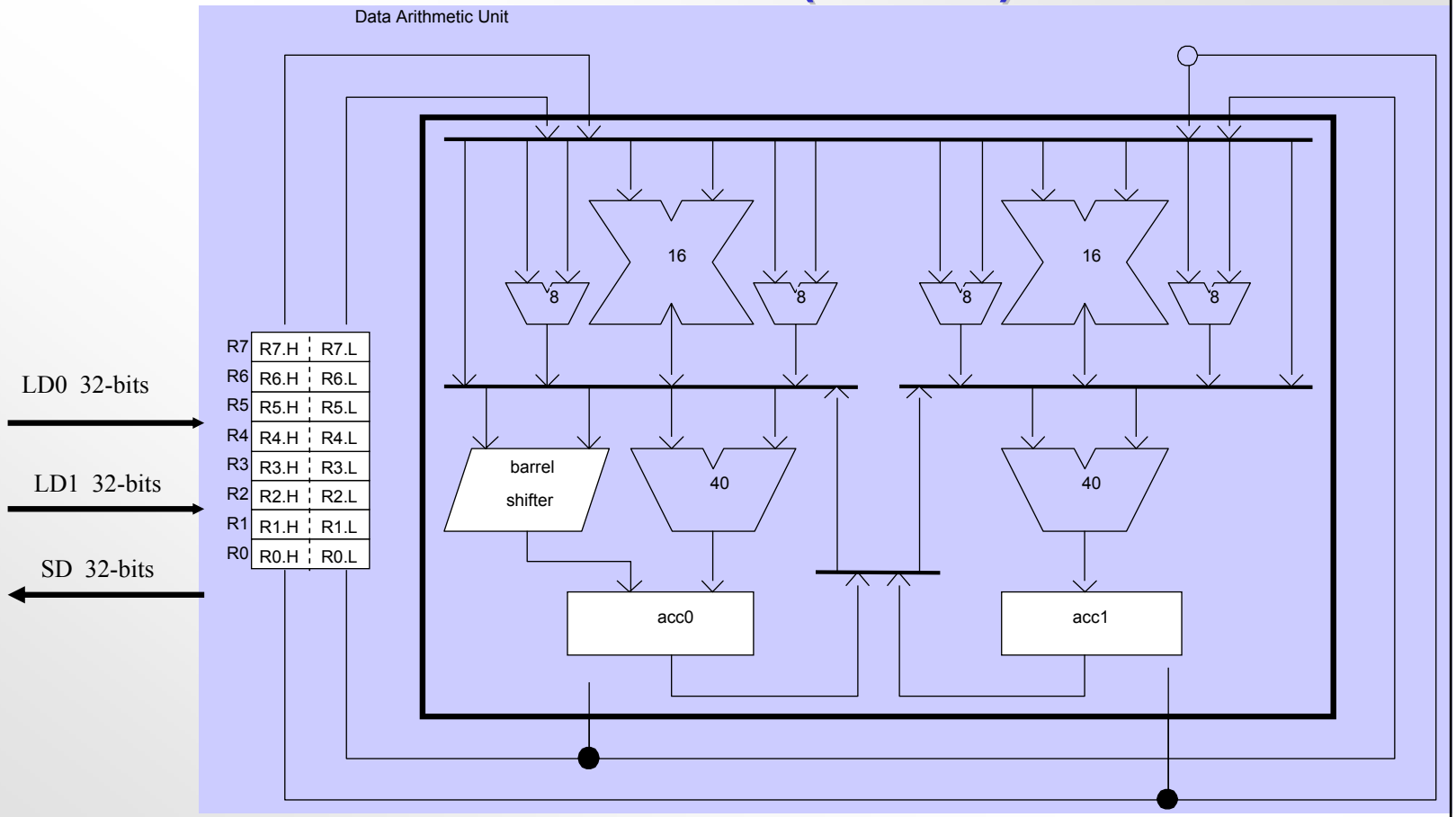
Instruction	ASTAT Status Flags		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	–	–	*
An = Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
An += Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
An –= Dreg_lo_hi * Dreg_lo_hi ;	*	*	–
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (An = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (An += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (An –= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	–	–	–

Multiplier Exercise

LAB 4

Barrel-Shifter (Shifter)

Barrel-Shifter (Shifter)



Barrel-Shifter (Shifter)

- The shifter performs bitwise shifting for 16-bit, 32-bit or 40-bit inputs and yields 16-bit, 32-bit, or 40-bit outputs.
- Functions
  - Arithmetic Shift: The Arithmetic Shift instruction shifts a registered number a specified distance and direction while preserving the sign of the original number. The sign bit value back-fills the left-most bit positions vacated by the arithmetic right shift.
  - Logical Shift: The Logical Shift instruction logically shifts a registered number a specified distance and direction. Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

Barrel-Shifter (Shifter)

- **Functions**
  - **Rotate: The Rotate instruction rotates a registered number through the CC bit a specified distance and direction.**
  - **Bit Operations**
  - **Field Extract and Deposit**

Shifter Instructions

Arithmetic Shift

The ">>>=" and ">>>" versions of this instruction supports only arithmetic right shifts. If left shifts are desired, the programmer must explicitly use arithmetic "<<" (saturating) or logical "<<" (non-saturating) instructions.

The Arithmetic Shift instruction supports 16-bit and 32-bit instruction length.

- The ">>>=" syntax instruction is 16 bits in length, allowing for smaller code at the expense of flexibility.
- The ">>>", "<<", and "ASHIFT" syntax instructions are 32 bits in length, providing a separate source and destination register, alternative data sizes, and parallel issue with Load/Store instructions.

Logical Shift

Syntax	Description
">>=" and "<<="	The value in dest_reg is shifted by the number of places specified by shift_magnitude. The data size is always 32 bits long. The entire 32 bits of the shift_magnitude determine the shift value. Shift magnitudes larger than 0x1F produce a 0x00000000 result.
">>", "<<", and "LSHIFT"	The value in src_reg is shifted by the number of places specified in shift_magnitude, and the result is stored into dest_reg. The LSHIFT versions can shift 32-bit Dreg and 40-bit Accumulator registers by up to –32 through +31 places.

Arithmetic Shift
Example Instructions

- **Immediate Shift Magnitude**

```
R3.L = R0.H >>> 7;          /* arithmetic right shift, half word */
R5 = R2 << 24 (S);          /* arithmetic left shift */
```

- **Registered Shift Magnitude**

```
R3.L = ashift R0.H by R7.L;     /* arithmetic shift, half-word */
A0 = ashift A0 by R7.L;         /* arithmetic shift, accumulator */
```

Logical Shift
Example Instruction

- Pointer shift, fixed magnitude

```
P3 = P2 >> 1;          /* pointer right shift by 1 */
P0 = P1 << 2;          /* pointer left shift by 2 */
```

- Data shift, immediate shift magnitude

```
R3.L = R0.L >> 4;      /* data right shift, half word register */
R3 = R0 << 12;         /* data left shift, 32-bit word */
A0 = A0 << 7;          /* accumulator left shift */
```

- Data shift, registered shift magnitude

```
R3.H = lshift R0.L by R2.L;    /* logical shift, half word register */
A1 = lshift A1 by R7.L;        /* logical shift, accumulator */
```

Rotate
Example Instruction

- **Immediate Rotate Magnitude**

  R4 = rot R1 by 8; **/* rotate left by 8 */**

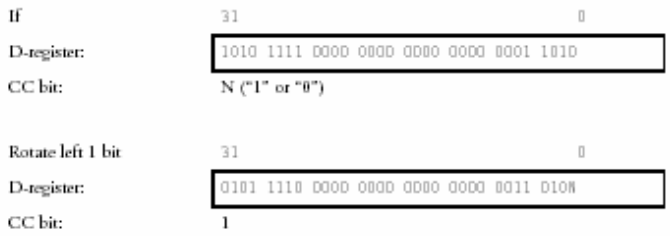
  A0 = rot A0 by –5; **/* rotate right by 5 */**

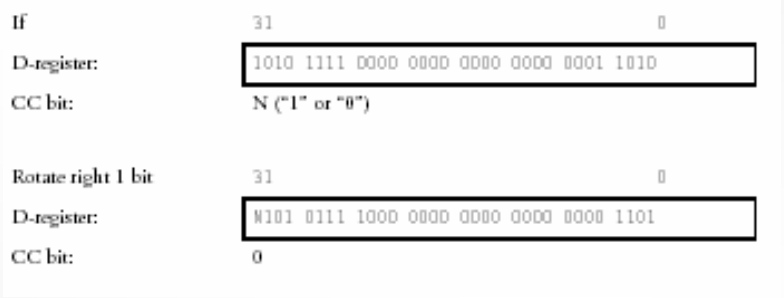
- **Registered Rotate Magnitude**

  R4 = rot R1 by R2.L **/* rotate by value in R2.L */**

  A1 = rot A1 by R7.L **/* rotate by value in R7.L */**

Rotation shifts all the bits either right or left. Each bit that rotates out of the register (the LSB for rotate right or the MSB for rotate left) is stored in the CC bit, and the CC bit is stored into the bit vacated by the rotate on the opposite end of the register.





Bit Operations
Example Instructions

- **Bit Clear: BITCLR(Dreg, uimm5);**

  bitclr(R2, 3);
- **Bit Set: BITSET(Dreg, uimm5);**

  bitset(R2, 7);
- **Bit Toggle: BITTGL(Dreg, uimm5);**

  bittgl(R2, 24);
- **Bit Test: CC = BITTST (Dreg, uimm5);**

  cc = bittst(r7, 15);
- **Bit Test: CC = !BITTST (Dreg, uimm5);**

  cc = !bittst(r3, 0);

Field Extract and Deposit Example Instructions

- **Bit Field Extraction**

  R7 = extract (R4, R3.L) (z); **//zero-extended**

  R7 = extract (R4, R3.L) (x); **//sign-extended**

- **Bit Field Deposit**

  R7 = deposit (R4, R3);

  R7 = deposit (R4, R3) (x); **//sign-extended**

```
r7 = deposit (r4, r3) ;
```

- If
  - R4=0b1111 1111 1111 1111 1111 1111 1111 1111
    where this is the background bit field
  - R3=0b0000 0000 0000 0000 0000 0111 0000 0011
    where bits 31–16 are the foreground bit field, bits 15–8 are the position, and bits 7–0 are the length

then the Bit Field Deposit (unsigned) instruction produces:

- R7=0b1111 1111 1111 1111 1111 1100 0111 1111

```
r7 = extract (r4, r3.l) (z) ;   /* zero-extended*/
```

- If
  - R4=0b1010 0101 1010 0101 1100 0011 1010 1010
    where this is the scene bit field
  - R3=0bxxxx xxxx xxxx xxxx 0000 0111 0000 0100
    where bits 15–8 are the position, and bits 7–0 are the length

then the Bit Field Extraction (unsigned) instruction produces:

- R7=0b0000 0000 0000 0000 0000 0000 0000 0111

Shifter Instruction Summary

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
BITSET (Dreg, uimm5) ;	**	*	**	–	–	–	**/–
BITTGL (Dreg, uimm5) ;	*	*	**	–	–	–	**/–
CC= BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
CC= !BITTST (Dreg, uimm5) ;	–	–	–	–	–	*	–
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	–	–	–	**/–
BITMUX (Dreg, Dreg, A0) ;	–	–	–	–	–	–	–
Dreg_lo = ONES Dreg ;	–	–	–	–	–	–	–
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	–	–	–	–	–	–	–
Dreg >>>=uimm5 ;	*	*	–	–	–	–	**/–
Dreg >>=uimm5 ;	*	*	–	–	–	–	**/–
Dreg <<=uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >>> uimm5 ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm5 ;	*	*	–	–	–	–	**/–

Shifter Instruction Summary

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg = Dreg << uimm5 ;	*	*	–	–	–	–	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg >> uimm4 (V) ;	*	*	–	–	–	–	**/–
Dreg = Dreg << uimm4 (V) ;	*	*	–	–	–	–	*
An = An >>>uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
An = An >>uimm5 ;	*	*	–	** 0/–	** 1/–	–	–
An = An <<uimm5 ;	*	*	–	* 0	* 1	–	–
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	–	–	–	–	*
Dreg >>>= Dreg ;	*	*	–	–	–	–	**/–
Dreg >>= Dreg ;	*	*	–	–	–	–	**/–
Dreg <<= Dreg ;	*	*	–	–	–	–	**/–
Dreg = ASHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	*
Dreg = LSHIFT Dreg BY Dreg_lo ;	*	*	–	–	–	–	**/–
Dreg = ROT Dreg BY imm6 ;	–	–	–	–	–	***	–

Shifter Instruction Summary

Instruction	ASTAT Status Flag						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	*
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;	*	*	–	–	–	–	**/–
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	*
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	–	–	–	–	**/–
An = An ASHIFT BY Dreg _lo ;	*	*	–	* 0	* 1	–	–
An = An ROT BY imm6 ;	–	–	–	–	–	***	–
Preg = Preg >> 1 ;	–	–	–	–	–	–	–
Preg = Preg >> 2 ;	–	–	–	–	–	–	–
Preg = Preg << 1 ;	–	–	–	–	–	–	–
Preg = Preg << 2 ;	–	–	–	–	–	–	–
Dreg = (Dreg + Dreg) << 1 ;	*	*	*	–	–	–	*
Dreg = (Dreg +Dreg) << 2 ;	*	*	*	–	–	–	*
Preg = (Preg +Preg) << 1 ;	–	–	–	–	–	–	–
Preg = (Preg +Preg) << 2 ;	–	–	–	–	–	–	–
Preg = Preg + (Preg << 1) ;	–	–	–	–	–	–	–
Preg = Preg + (Preg << 2) ;	–	–	–	–	–	–	–

Barrel Shifter Exercise

LAB 5

Reference Material

Arithmetic Units

Rounding Instructions

The Add/Subtract – Prescale Up instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale up both input operand values by shifting them four places to the left
- Add or subtract the operands, depending on the instruction version used
- Round and saturate the upper 16 bits of the result
- Extract the upper 16 bits to the *dest_reg*

```
Dreg_lo_hi = Dreg + Dreg (RND12) ;
Dreg_lo_hi = Dreg - Dreg (RND12) ;
```

The Add/Subtract -- Prescale Down instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale down both input operand values by arithmetically shifting them four places to the right
- Add or subtract the operands, depending on the instruction version used
- Round the upper 16 bits of the result
- Extract the upper 16 bits to the *dest_reg*

```
Dreg_lo_hi = Dreg + Dreg (RND20) ;
Dreg_lo_hi = Dreg - Dreg (RND20) ;
```

Bitwise XOR Instructions

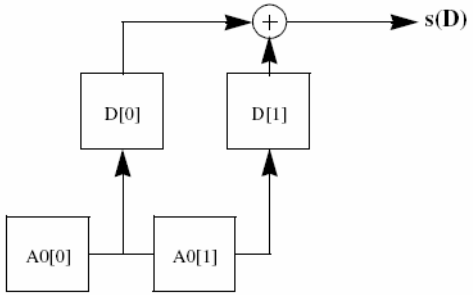
- **These instructions are used to implement Linear Feedback Shift Registers (LFSR's)**
- **Applications include CRC (Cyclic Redundancy Check) calculations and PRN (Pseudo Random Number) generators**

LFSR Type I (Without Feedback)

```
Dreg_lo = CC = BXORSHIFT ( A0, Dreg ) ;     /* (b) */
Dreg_lo = CC = BXOR ( A0, Dreg ) ;     /* (b) */
```

LFSR Type I (With Feedback)

```
Dreg_lo = CC = BXOR ( A0, A1, CC ) ;     /* (b) */
A0 = BXORSHIFT ( A0, A1, CC ) ;     /* (b) */
```



In the following circuits describing the BXOR instruction group, a bit-wise XOR reduction is defined as:

$$Out = (((((B_0 \oplus B_1) \oplus B_2) \oplus B_3) \oplus \ldots) \oplus B_{n-1})$$

In the figure above, the bits A0 bit 0 and A0 bit 1 are logically AND'ed with bits D[0] and D[1]. The result from this operation is XOR reduced according to the following formula.

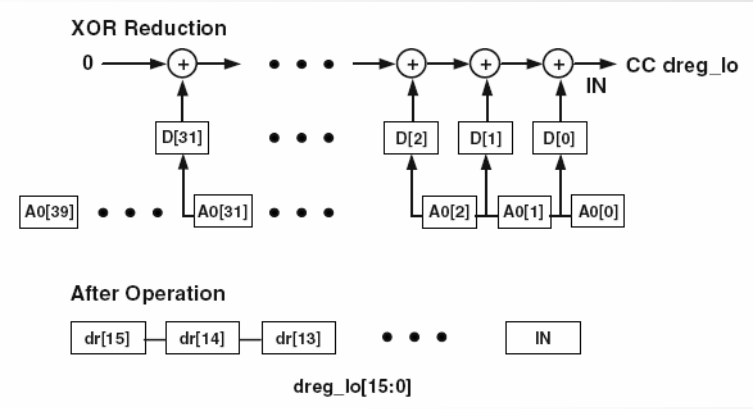
$$s(D) = (A0[0] \& D[0]) \oplus (A0[1] \& D[0])$$

Bitwise XOR Instructions

Example (no feedback)

```
Dreg_lo = CC = BXOR(A0, dreg)

r0.l = cc = bxor (a0, r1) ;
```



Example (feedback)

```
A0 = BXORSHIFT(A0, A1, CC)

a0 = bxorshift (a0, a1, cc) ;
```