More M68000 Instructions

- Recall category:

(1) data movement (move)

(2) Arithmetic (ADD, ADDA, ADDI, ADDQ
SUB, SUBA, SUBI, SUBQ)

(3) Logic

(4) Branching

(5) System / control. (Skip).

Go back and do add
sub

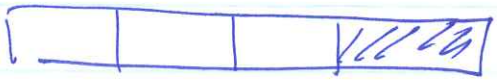
Addr. Rg dest:

- move a
- add a
- sub a

Data Conversion

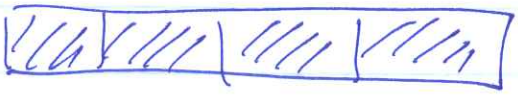
- Data registers can hold 3 types of operands:

byte



word.

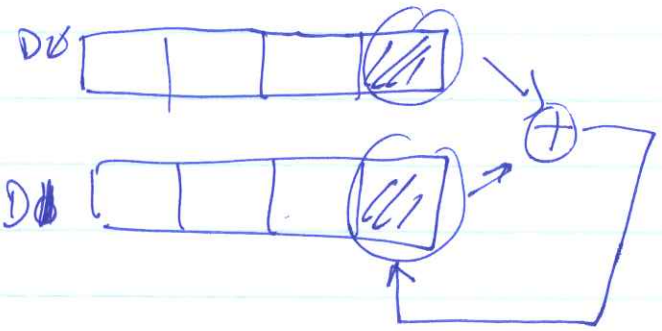
long Word:



- Operation will operate on operands of same type.

eg:

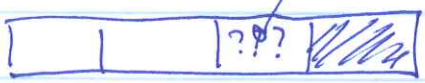
add.b d0, d1

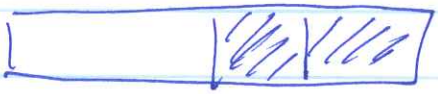


result put back into last 8 bits of D1.

- What if d0 and d1 contains operands of diff. sizes?

random bits.

Say: D0: 

D1: 

* Instruction EXT[.s] Dn sign extend register Dn.

s can be w or l

EXT.w Dn : extend byte representation into word repr.
EXT.l Dn : extend word repr. to long word repr.

* Repr: 3 = 00000011 byte repr.
= 00000000 00000011 word repr.
= 00000000 00000000 ... 00011 long word repr.

-3 = ~~11111101~~
= 11111101 byte repr
= 1111 1111 1111 1101 word repr.
= 11111111111111 ... 1101 long word repr.

Example:

```
char a;          a: ds.b 1
short b;         b: ds.w 1

b = a;           move.b  a, D0
                 ext.w   D0
                 move.w  D0, b.

a = b;           move.b  b, a   !!!
```

wrong:

right:
```
move.w  b, d0
move.b  d0, a
```

do:

```
char b;
short c

b = b + a;

a = b + a;
```

- Multiply

Proj 1: ~~access~~ Addressing modes.

- M68000 can handle numbers.

(signed = 2's comp

~~unsigned~~

- M68000 has 2 multiply instructions:

MULS	<ea>, Dn	multiply signed numbers
MULU	<ea>, Dn	multiply unsigned numbers.

- Effect:

<ea>.W * Dn.W → Dn.L

- <ea>.W: word operand at source (16 bits)
- Dn.W: word operand in Dn (16 bits)
- Dn.L: write result as long operand. (32 bits).

Example:

suppose: D0 = | | 1111.1111 | 1111.1110 |

D1 = | | 0000.0000 | 0000.0010 |

MULS D0, D1 results in:

D1 = | 1111.1111 | 1111.1111 | 1111.1111 | 1111.1100 |

Because: D0: | 1111.1111 | 1111.1110 | is −2 in 2's compl.

D1: | 0000.0000 | 0000.0010 | is +2 in 2's compl.

product is −4

and the above pattern represents the value −4 in 32 bit 2's compl. notation.

On the other hand:

MULU D0,D1 will result:

D1 = | 0000.0000 | 0000.0001 | 1111.1111 | 1111.1100 |

Because D0 | 1111.1111 | 1111.1110 | unsigned =

$2^{15} + 2^{14} + + 2^{13} + \ldots + 2^{2} + 2^{1} =$

65534.

D1 | 0000.0000 | 0000.0010 | unsigned = 2.

product = 131068

This value (unsigned) is represented by pattern in D1 above.

- You don't have to worry about the representation just the effect:

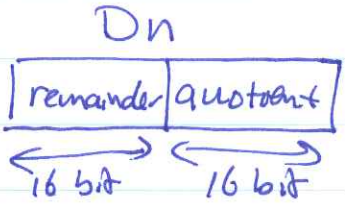
		signed	unsigned
Su:	D0 = 1111.1111 | 1111.1110	= "-2"	"65534"
	D1 = 0000.0000 | 0000.0010	= "+2"	"2"
D0 + D1		= "-4" in 32 bits 2's comp!	"FFFFFFFC" in 32 bits unsigned.

Division - Integer division.

- There are 2 divide instructions : signed & unsigned.
- Syntax:

DIVU	<sea>, Dn	unsigned division
DIVS	<sea>, Dn	signed division.

Effect : Dn.L ÷ <sea>.W →



Divident is 32 bits in Destmation Dn
Divisor is 16 bit in source

The quotient is stored in lower 16 bits in Dn

The remander is stored in upper 16 bits in Dn.

Example:

Suppose D$\emptyset$ = | 0000.0000 | 0000.0000 | 0000.0000 | 0000.1001 | "9"

Then: DIVS #2, D$\emptyset$ (Quotient = 4, remainder = 1).

results in:

D$\emptyset$ = | 0000.0000 | 0000.0001 | 0000.0000 | 0000.0100 |

"1" "4"

How do I get hold of the remainder?

How do I use the divide instruction?

Example:

```
A:  dc.l  9

Q:  ds.w  1
R:  ds.w  1
```

We want to assign:

Q = A / 2 (quotrant)
R = A % 2 (remainder).

Assembler code:

```
move.l  A, Dø
DIVS    #2, Dø
move.w  Dø, Q
SWAP    Dø
move.w  Dø, R
```

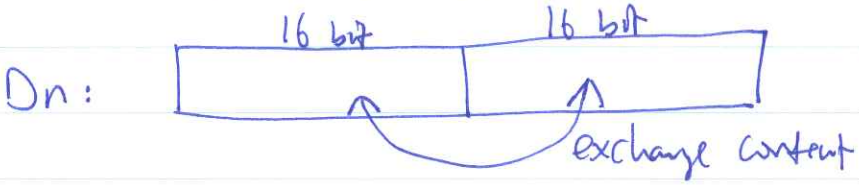
only if remainder is in lower 16 bits of Dø!

SWAP:

syntax: SWAP Dn

effect: swap the MSW & LSW of reg Dn.

most significant word.



Subtleties in Division:

suppose: word!

```
A :  ds.w  9

Q:   ds.w  1      ← Q = A/2
R:   ds.w  1.     ← R = A %2.
```

The assembler program:

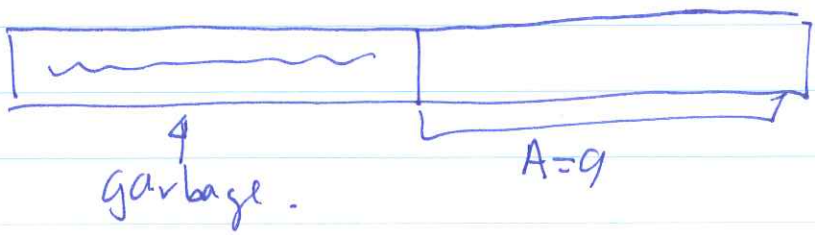
```
MOVE.W   A, D0

DIVS     #2, D0

move.w   D0, Q
SWAP     D0
move.w   D0, R
```

will not work.

Reason: D0 :



Division takes 32 bits in D0 & divides it.
So before you divide, you must have 32 "good" bits.

```
move.w   A,D0
```

will put the value 9 in 16 bit representation

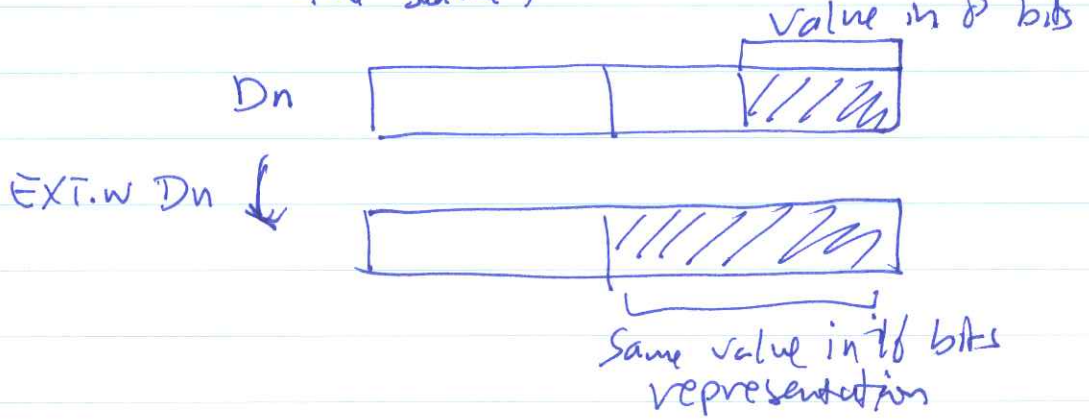
You must make the representation into 32 bits.
before you can divide.:

For this purpose, 68000 has the sign extend operation:

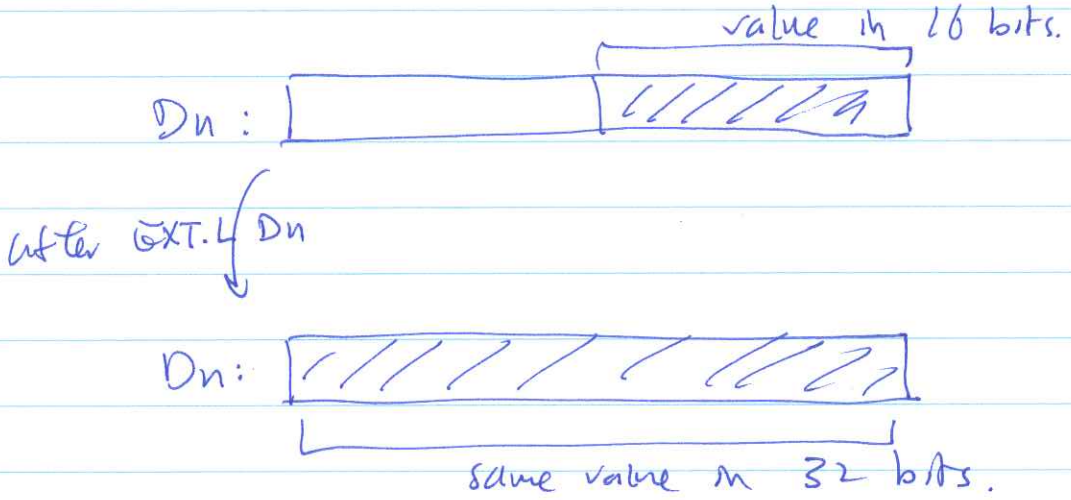
Syntax:

```
EXT.s   Dn          s = W or L.
```

EXT.w Dn extends a byte operand in Dn to a word operand (value remands the same)



EXT.L Dn extends a word operand in Dn to a long word operand



So, ~~then~~ the correct program is:

```
move.w   A,D0
EXT.L    D0

DIVS     #2, D0
move.w   D0, Q
SWAP     D0
move.w   D0, R
```

What if:

```
A:  dc.b   9

Q:  ds.w   1
R:  ds.w   1.
```

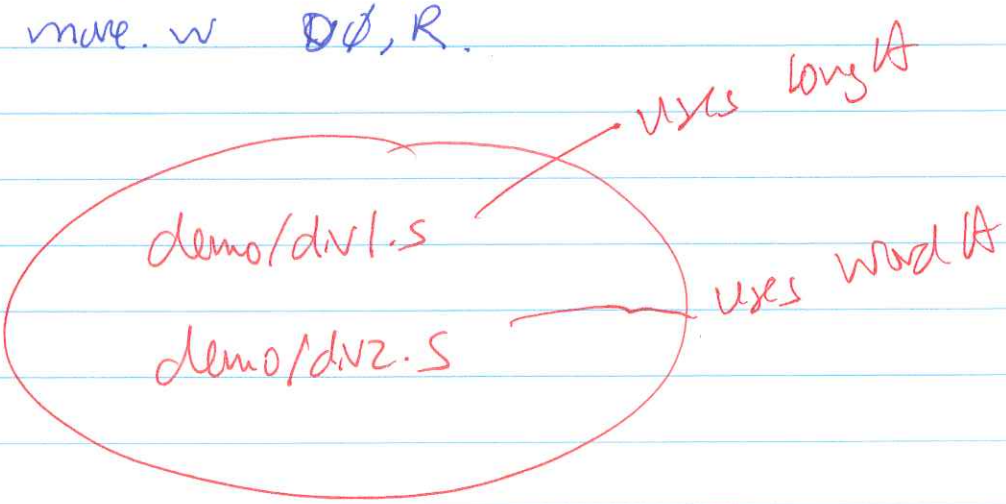
The division now goes:

```
move.b   A, D0

ext.w    D0
ext.L    D0

DIVS     #2, D0

move.w   D0, Q
swap     D0
move.w   D0, R.
```



Negate:

Syntax: NEG [.S] <ea>

Used only on signed numbers: negates the 2's compl. number at destination <ea>.

Eg: NEG.L D0

negates the 32 bit 2's compl. number in D0.

Logic operations

(1) AND: computes bitwise and operation.

Syntax: AND.s <source>, Dn
AND.s Dn, <dest>

Effect: <dest> := <source> AND <dest>

eg: AND.L D0, D1.

(2) OR: computes bitwise OR operation.

syntax: OR.s <source>, Dn.
OR.s Dn, <dest>.

effect: <dest> := <source> OR <dest>

eg: OR.B D0, D1

(3) EOR: computes bitwise exclusive OR.

syntax: EOR.s <source>, Dn
EOR.s Dn, <dest>

effect: <dest> := <source> ⊕ <dest>

eg: EOR.L D0, D1

→ Before we do branch & other instr., take a look at addr. modes !!!

(4) NOT : Computes bitwise complement (0→1, 1→0)

Syntax : NOT.s <dest>

effect : <dest> := NOT <dest>

eg : NOT.B D0.

SHIFT & ROTATE

Skip shift & rotate !
Not likely to be used anyway!

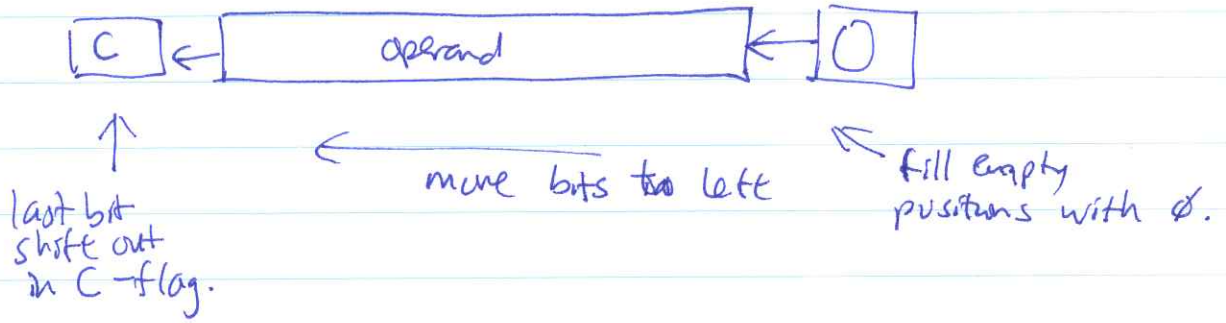
Shift & Rotate

- Shift & rotate instructions move the bit pattern in a register left or right.

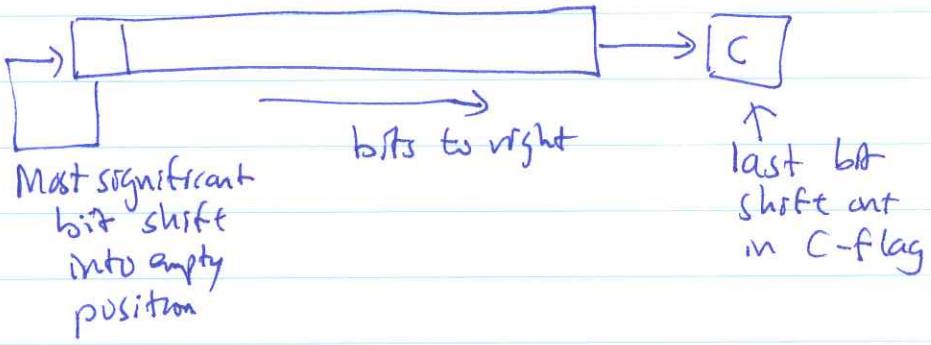
Arithmetic shift:

ASL = arithmetic shift to left (size = B, W, L)
ASR = arithmetic shift to right. (size = B, W, L).

ASL:



ASR:



Syntax:

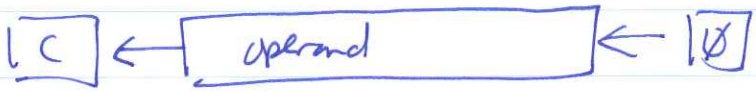
```
ASL  Dx, Dy        Or:   ASL  #n, Dy
ASR  Dx, Dy              ASR  #n, Dy
```

Dx: count shift. Dy: operand. #n: count. Dy: operand.

Logical shift :

LSL = logical shift to left
LSR = logical shift to right.

LSL:



same as ASL.

LSR:

Note: C has shift operators:

<< = logical shift to left
>> = logical shift to right.

eg: 2 << 1 = 4 !!!

2 = 0000010
shift result: 0000100 = 4!

Rotate :

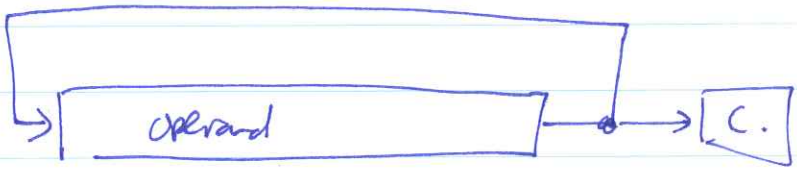
ROL = rotate left

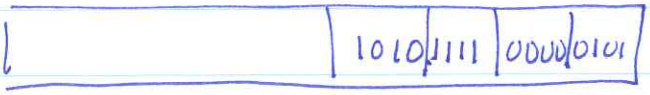
ROR = Rotate right.

ROL :

last bit rotated out is in carry.

ROR :



eg: D0 = 

ROL #2, D0

D0 =