Arithmetic/logic operations

Sum and Carry

- Decimal

	3 7 7
+	4 4 9
sum	6
carry	1

	3 7 7
+	4 4 9
sum	2 6
carry	?

	3 7 7
+	4 1 9
sum	8 2 6
carry	?

- Binary (1-bit) addition

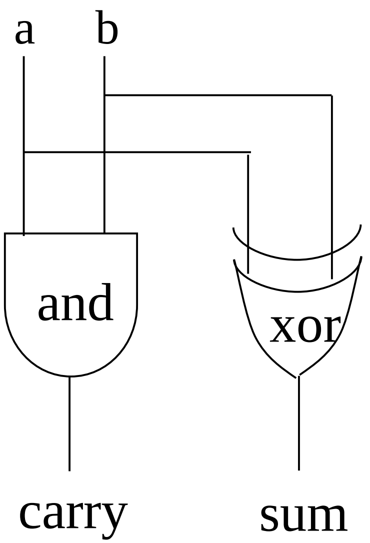
$$\begin{array}{r} a \\ +\ b \\ \hline \end{array} \qquad\qquad \begin{array}{r} c_{in} \\ a \\ +\ b \\ \hline \end{array}$$

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

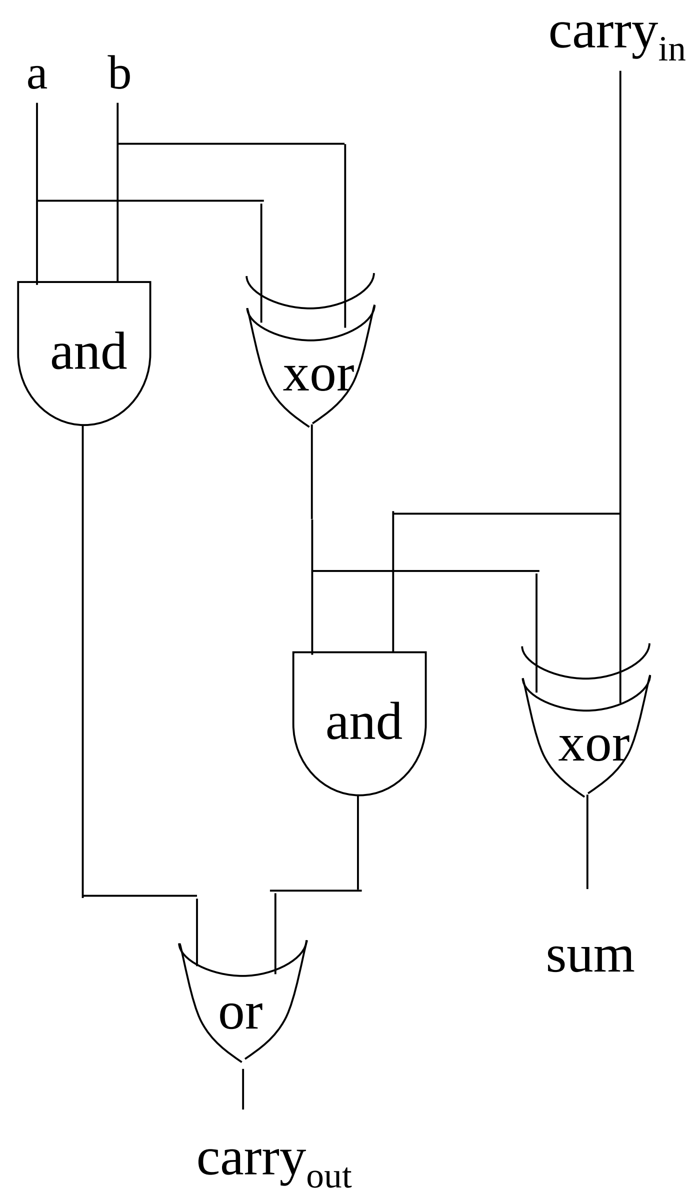
sum = a xor b
carry = a and b

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

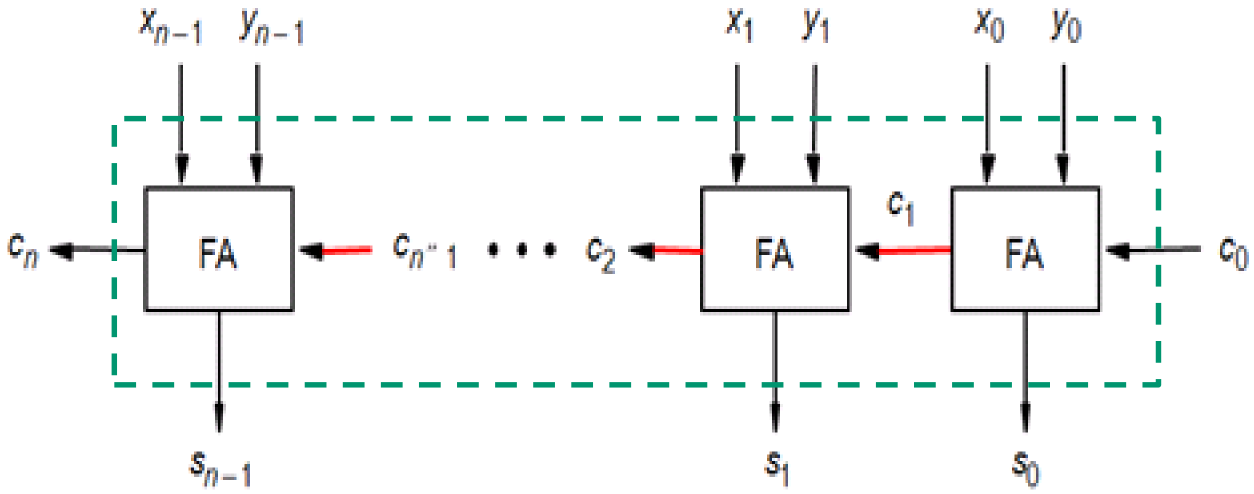
half adder



full adder



N-bit (binary) addition: Ripple carry adder



Addition: other method

	3 7 7
+	4 4 9
sum	7 1 6
carry	*0 1 1*
+	1 1
	8 2 6

	3 7 7
+	4 2 9
sum	7 9 6
carry	*0 0 1*
+	0 1
sum	7 0 6
carry	*0 1 0*
+	1
	8 0 6

Left-shift of carry

Binary addition using logic operations

- Addition using XOR and AND (C routine)

```
int add ( int a, int b ) {
    int s;        /* sum */
    int c;        /* carry */
    s = a ^ b;    /* xor operation */
    while (c = (a & b) << 1) {  /* shift left after carry-
        a = s;                      decision*/
        b = c;
        s = a ^ b;
    }
    return(s);
}
```

a **+** b

a, b: **n** bits

Binary addition using logic operations: Example

- 13_{10} + 11_{10}

001101 (a)

001011 (b)

sum 000110 (s) → (**a**)

carry 001001

carry<<1 01001 (c) → (**b**)

sum 010100 (**s**) → (**a**)

carry 000010

carry<<1 00010 (**c**) → (**b**)

sum 010000 (**s**) → (a)

carry 000100

carry<<1 00100 (**c**) → (b)

sum 011000

carry 000000

Subtraction for signed number

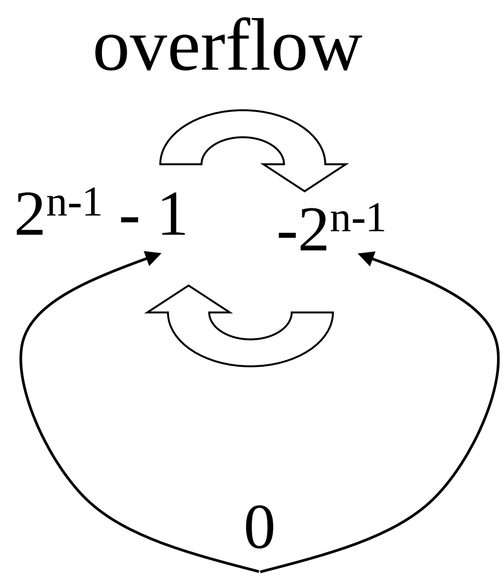
- Negative numbers in 2's complement representation
- Most Significant Bit (MSB) - 0: positive, 1: negative
- Subtraction is addition of 2's complement

✓ $x - y = x + (-y)$

$$= x + (2^n - 1 - y) + 1$$

n: number of bits

- Using n bits,
  - signed: $-2^{n-1} \sim 2^{n-1} - 1$
  - unsigned: $0 \sim 2^n - 1$



0000	0
0001	1
0010	2
...	
0111	7
1000	-8
1001	-7
....	
1111	-1

Example (when n = 8)

✓ Addition ignoring carry

6-3 = 6+(-3)

```
    +6: 00000110
+(-3): 11111101
--------------
        00000011
```

3-6=3+(-6)

```
    +3: 00000011
+(-6): 11111010
--------------
        11111101
```

Overflow example

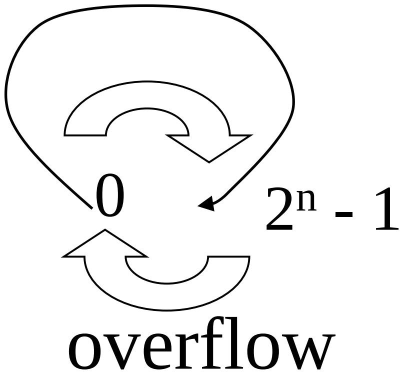
```
 127-(-1)=127+1          -128-1=-128+(-1)

 127: 01111111           -128:    10000000
+  1: 00000001         + (-1):    11111111
      10000000                    01111111
```

✓ Are calculation results reliable?

Subtraction of unsigned number

- Using n bits, numbers between 0 and $2^n - 1$
- Same operation for unsigned/signed numbers

  → use same hardware, same instructions



✓ Example: addition ignoring carry

$6-3 = 6+(-3)$ $\qquad\qquad$ $3-6=3+(-6)$

```
    +6:  00000110          +3:  00000011
+(-3):   11111101      +(-6):   11111010
-----------------------------------------
       1 00000011             0 11111101
```

Addition/subtraction operations

name	operation
add	addition
addcc	addition with CC gen.
addx	addition including carry
addxcc	( )
sub	subtraction
subcc	subtraction with CC gen.
subx	subtraction including carry
subxcc	( )

- Example
  - ✓ add %l0, %l1, %l2
  - ✓ add %l0, 5, %l2
  - ✓ sub %l0, 7, %l3
  - ✓ subcc %l0, 7, %l3
  - ✓ addx %l1, %l2, %l3

✓ Format

op-code R, A, S

R: source register

A: s. register or immediate value(−4096 ~ 4095)

S: destination register

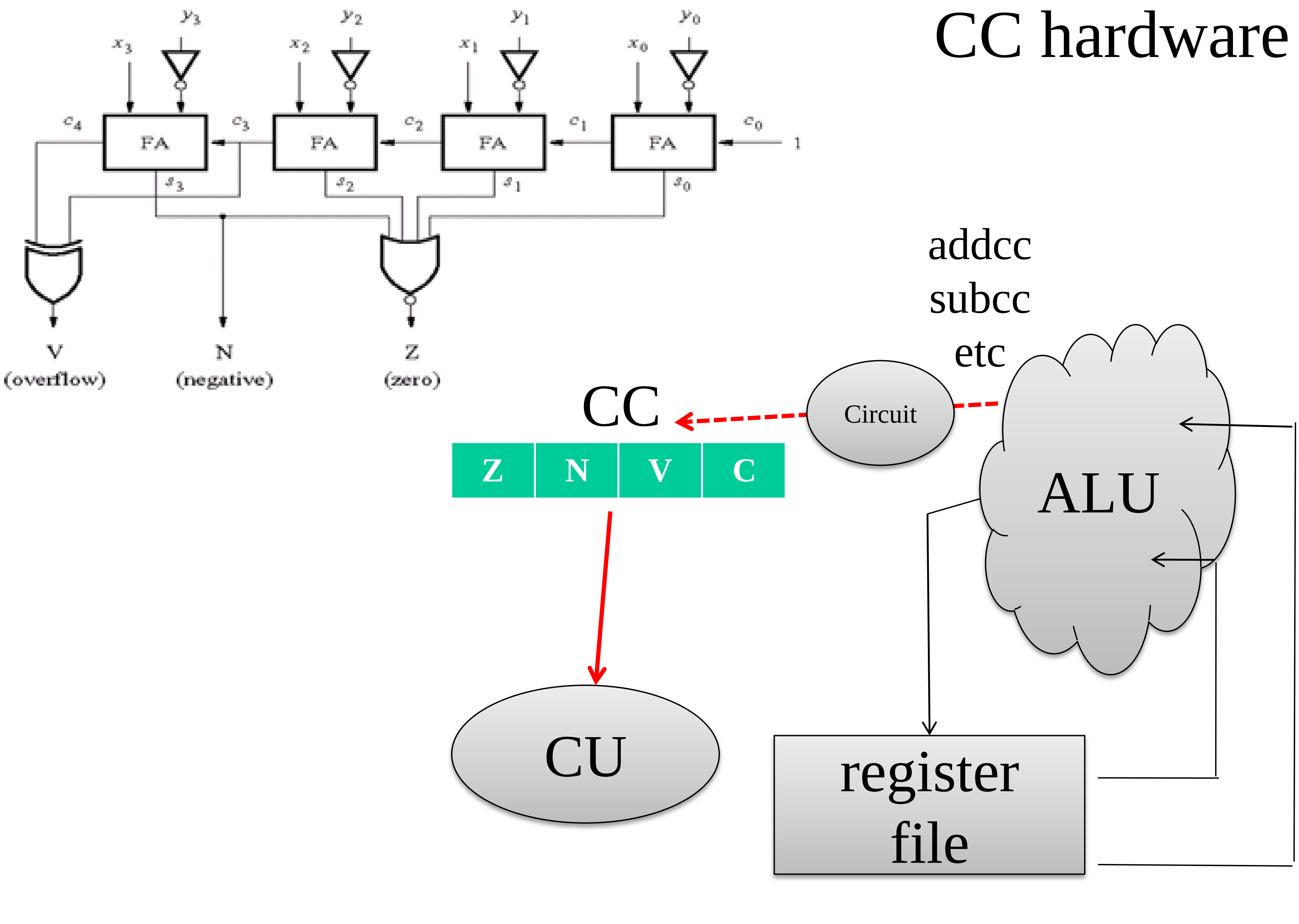
CC (condition code)

- CC changes with CC generation operations
  - add**cc**, sub**cc**, …

- Condition Codes

CC	meaning
Z	is op result zero?
N	is op result negative
V	overflow occurred?
C	carry

CC hardware



FA
c_4 c_3 c_2 c_1 c_0
x_3 y_3 x_2 y_2 x_1 y_1 x_0 y_0
s_3 s_2 s_1 s_0
1
V (overflow)
N (negative)
Z (zero)
addcc
subcc
etc
CC
Circuit
Z N V C
ALU
CU
register file

Arithmetic/logic machine instructions format

✓ op-code R, A, S

Bit no.	31 30	29 25	24 19	18 14	13	12 5	4 0
Field	OP (10)	S	OP-ext	R	**0**		A

Bit no.	31 30	29 25	24 19	18 14	13	12 0
field	OP (10)	S	OP-ext	R	**1**	immediate value

✓ OP-ext: Table 3.4

✓ Example

sub %l0, %l1, %o0

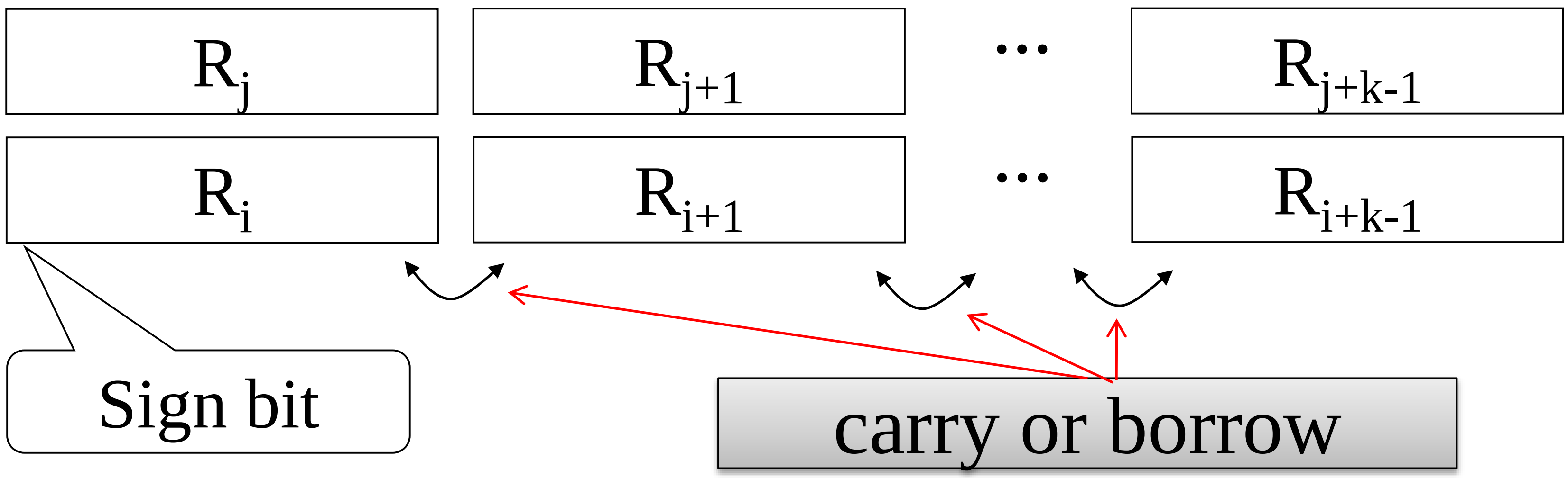
→ 10 01000 000100 10000 0 0000000010001

sub %l0, 5, %o0

→ 10 01000 000100 10000 1 0000000000101

Extended precision arithmetic

- Precision of integer numbers: 32 bits
- Extended precision on single precision hardware
  - ✓ use a number of registers
  - ✓ where is sign bit? MSB of register containing most significant word



- Addition

  addx R, A, S

  addxcc R, A, S

  ✓ $S = R + A + C$

  c : carry

- Subtraction

  subx R, A, S

  subxcc R, A, S

  ✓ $S = R - A - C$

  c : borrow

- Addition and subtraction of 96-bit integers

1st operand : %l0 %l1 %l2

2nd operand : %l3 %l4 %l5

Results : %o0 %o1 %o2

✓ Addition

addcc %l2, %l5, %o2

addxcc %l1, %l4, %o1

addx %l0, %l3, %o0

✓ Subtraction

subcc %l2, %l5, %o2

subxcc %l1, %l4, %o1

subx %l0, %l3, %o0

Subtraction using add inst.

- (%lo %l1 %l2) - (%l3 %l4 %l5)
  = (%lo %l1 %l2) + (%l3 %l4 %l5)’ +1

```
not      %l5, %l5           ! 1's complement
not      %l4, %l4
not      %l3, %l3
inccc    %l5                ! 2's complement
addxcc   %l4, %g0, %l4      ! carry 전파
addx     %l3, %g0, %l3

addcc    %l2, %l5, %o2      ! add bits 0 - 31
addxcc   %l1, %l4, %o1      ! add bits 32 - 63 + C
addx     %l0, %l3, %o0      ! add bits 64 - 95 + C
```

addcc %l5, 1, %l5

Multiplication

- Decimal multiplication

```
        1 2 3
      * 3 2 1
      -------
        1 2 3
      2 4 6
+   3 6 9
-------------
    3 9 4 8 3
```

- Problem

1) Storage requirements? Can be reduced?

```
    1 2 3
  * 3 2 1
 -----------
    1 2 3 (partial product)
+ 2 4 6
-----------
  2 5 8 3 (partial product)
+ 3 6 9
-----------
3 9 4 8 3 results
```

```
      1 2 3
    * 3 2 1
  -------------
0 1 2 3 3 2 1
+ 2 4 6
  -------------
2 5 8 3 3 2
+ 3 6 9
-------------
3 9 4 8 3
```

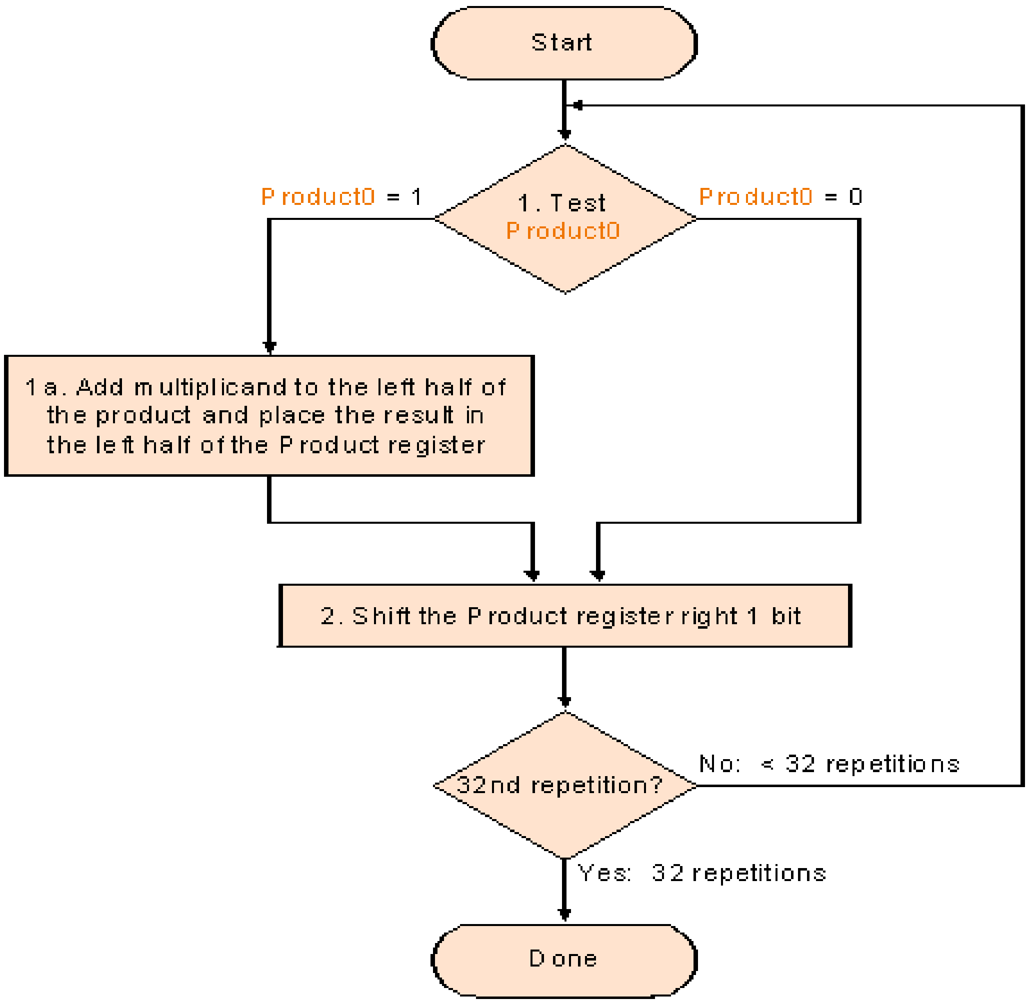
Multiplication operation (1)

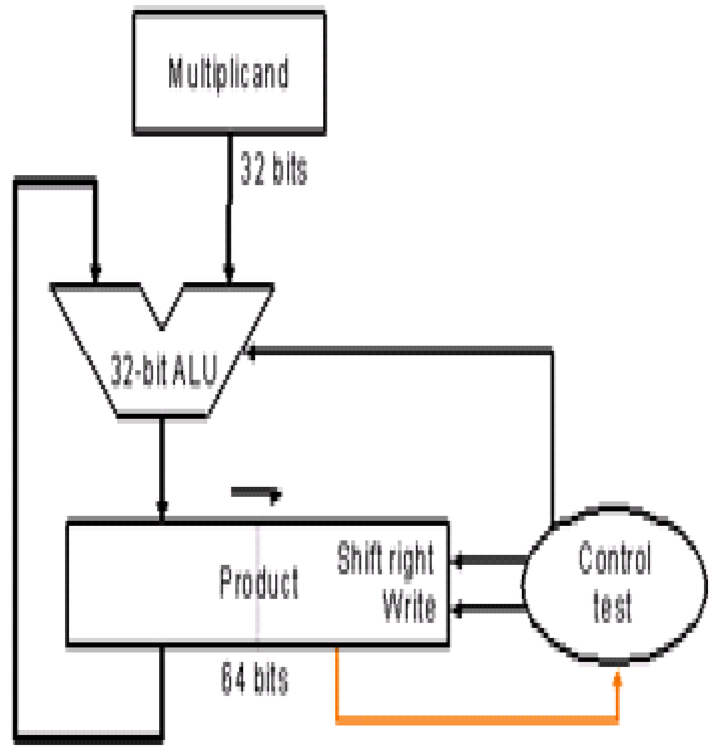
- Number of digits of product = Sum of numbers of digits of multiplier & multiplicand
  - One register for multiplicand
  - Two registers for product and multiplier
    - ✓ (partial) product register: initialized with zero

- Repeat below *N* times where N is the number of bits of multiplier (: binary case.)

1) Check LSB of multiplier
   - ✓ If 1 then add multiplicand to product
2) Shift-right (sra) product-multiplier register pair

Multiplication algorithm/hardware





Multiplication operation (2)

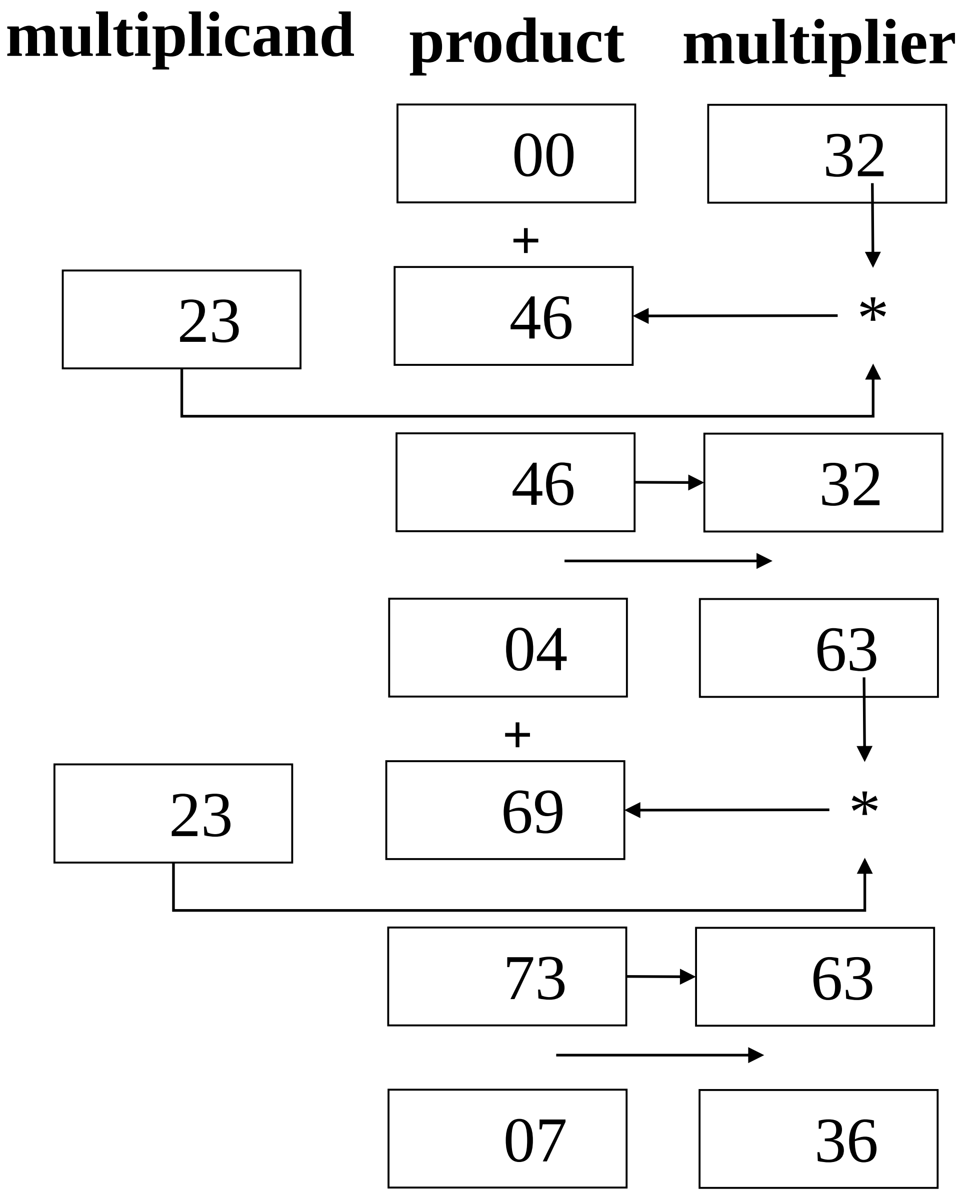
- Negative multiplicand positive multiplier case
  - ✓ Ex: (-3) * 5

- Negative multiplier case
  - ✓ Ex: 3 * (-5)

→

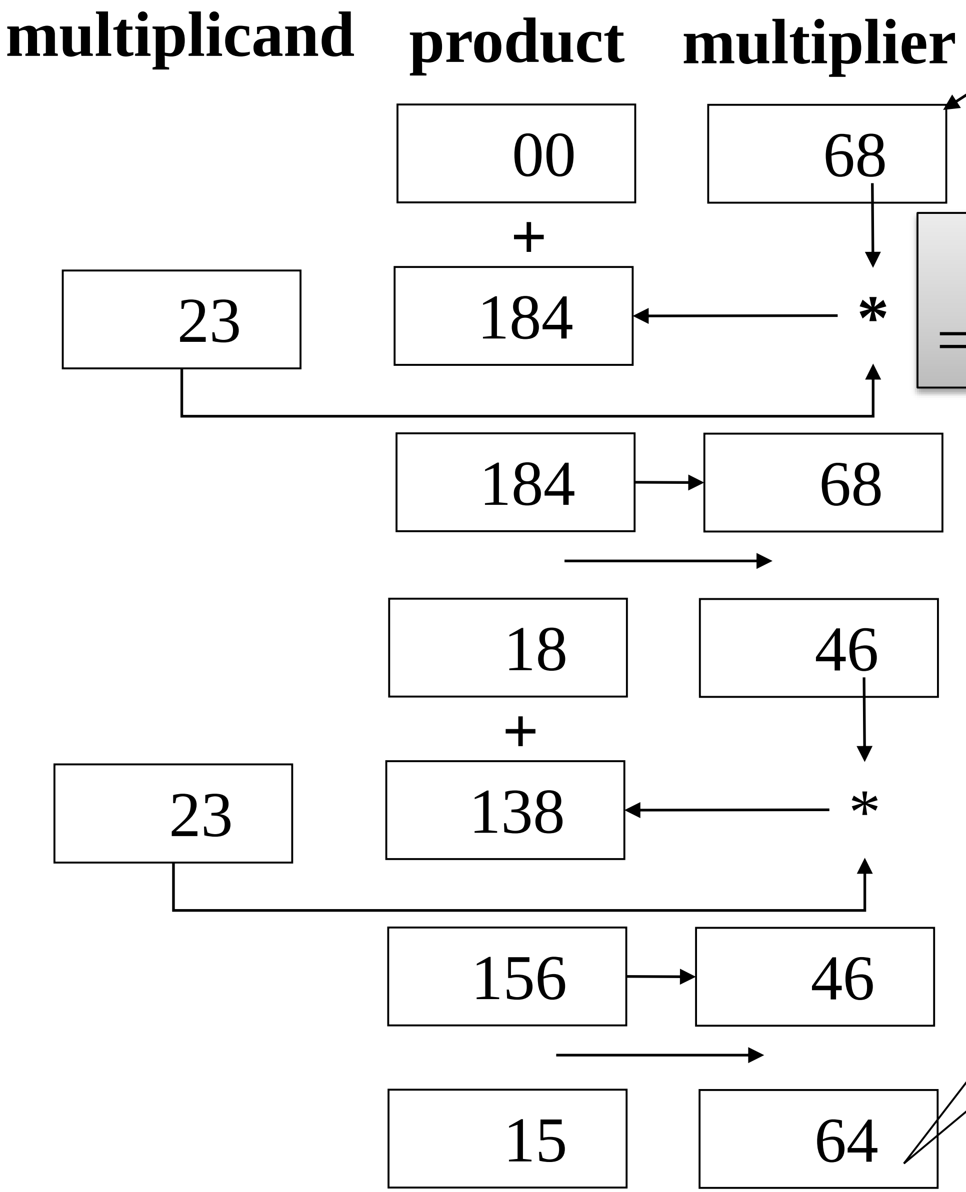
  - ✓ Make multiplier positive: (-3) * 5
  - ✓ Do calculation as it is

Multiplication (decimal number case)




```
     23
*    32
-------
     46
+   69
-------
    736
```

Multiplication (cont.)



10’s compliment

$$\begin{array}{rr} & 23 \\ * & -32 \\ \hline & -46 \\ + & -69 \\ \hline & -736 \end{array}$$

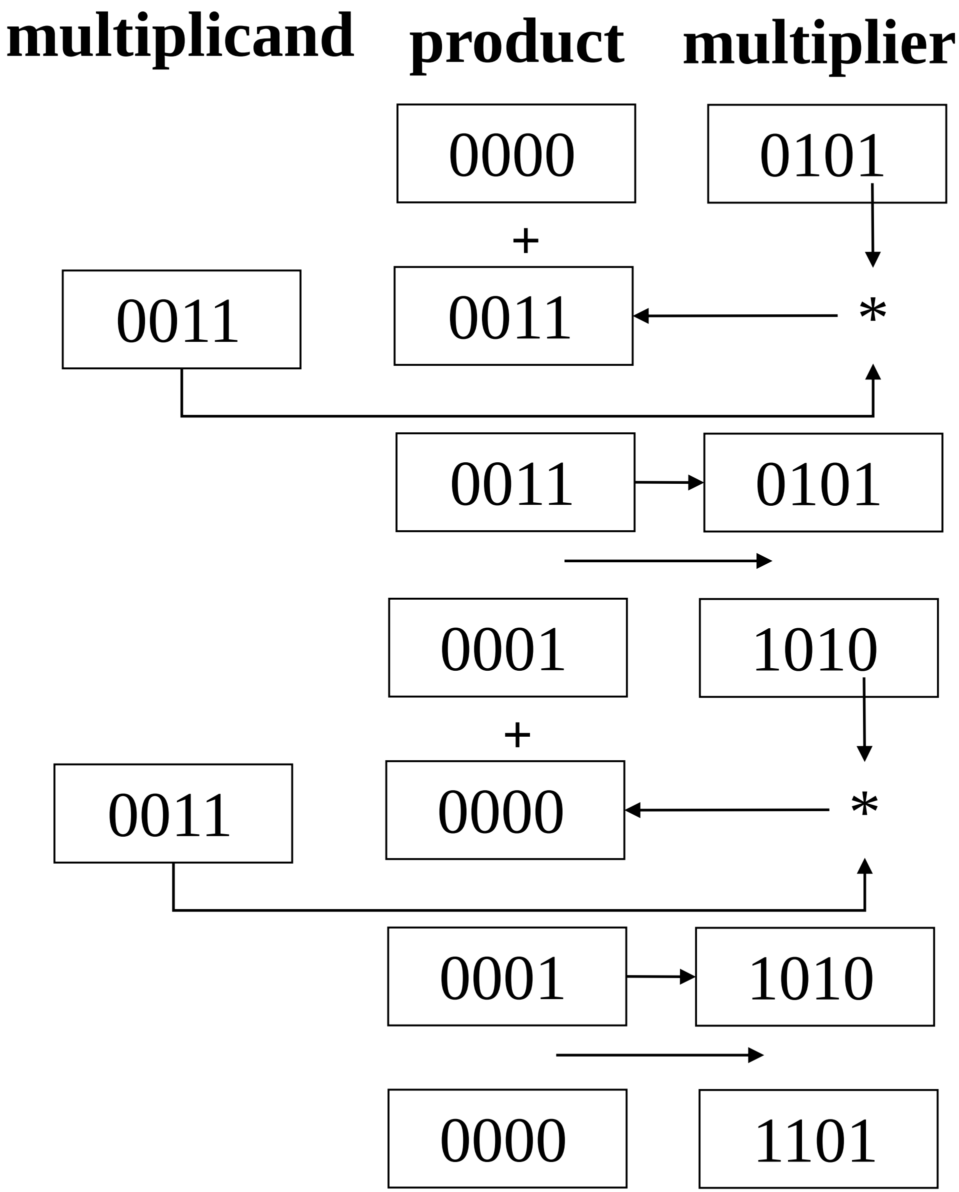
$23*(10^2 - 32)$
$= 23*10^2 - 23*32$

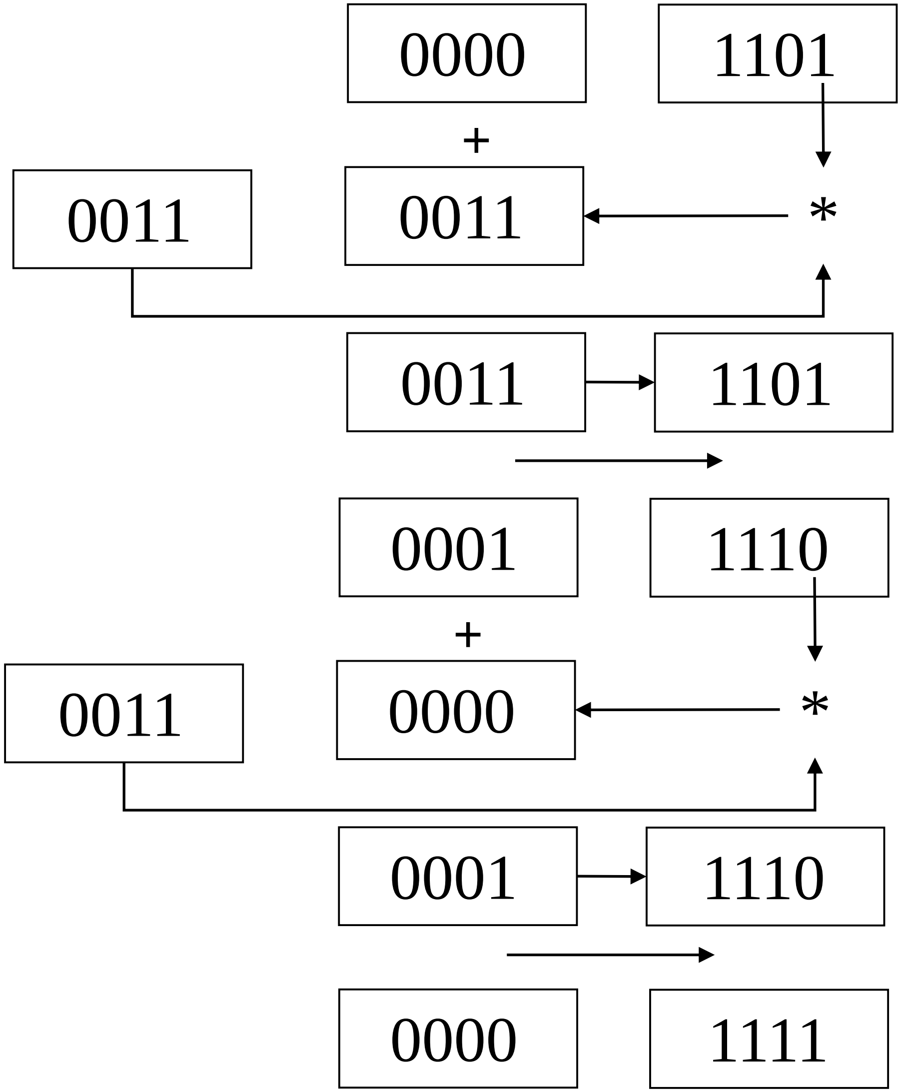
9264 is 10’s compliment of -736

- adjustment:

$$\begin{array}{rr} 15 & 64 \\ +77 & 00 \\ \hline 92 & 64 \end{array}$$

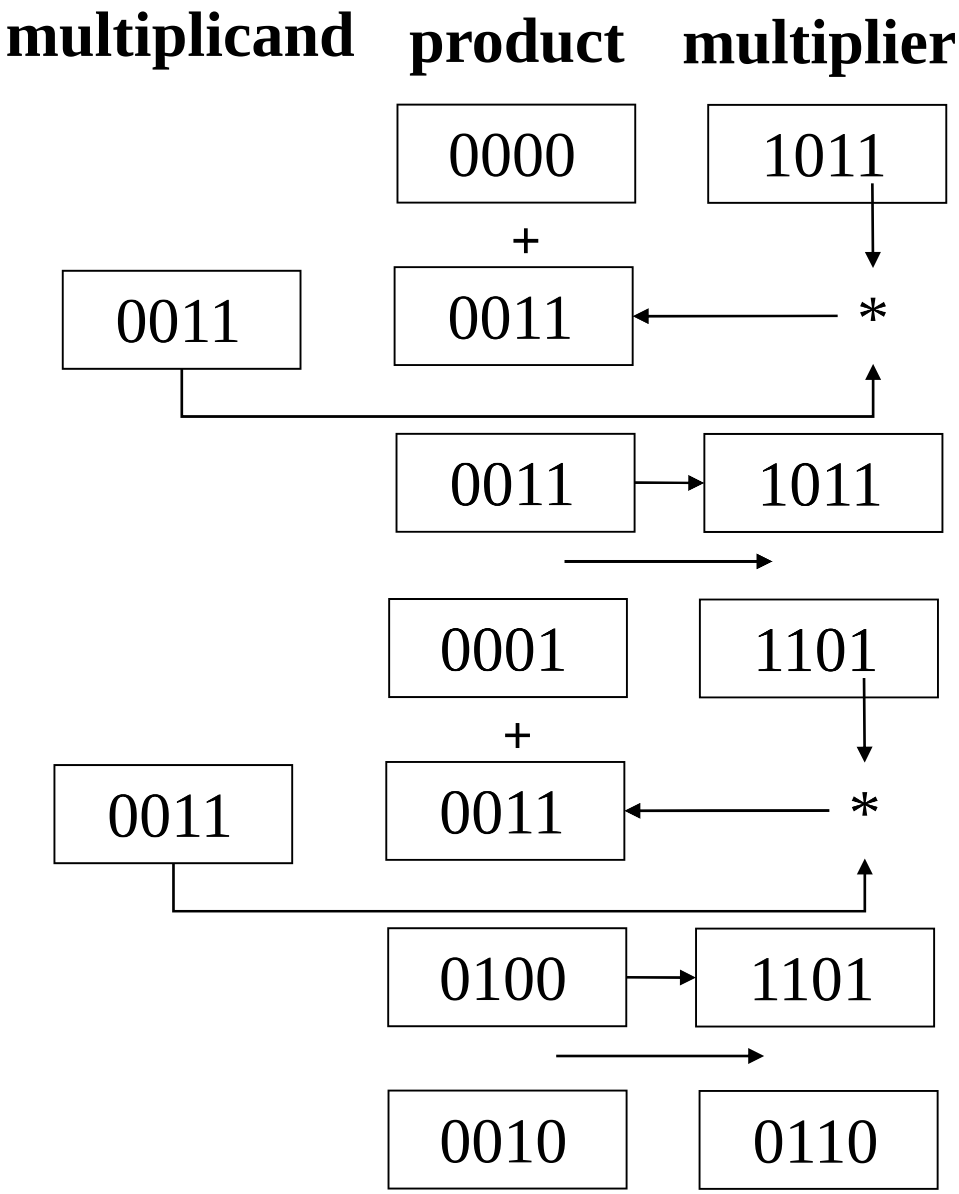
Multiplication: 3 x 5



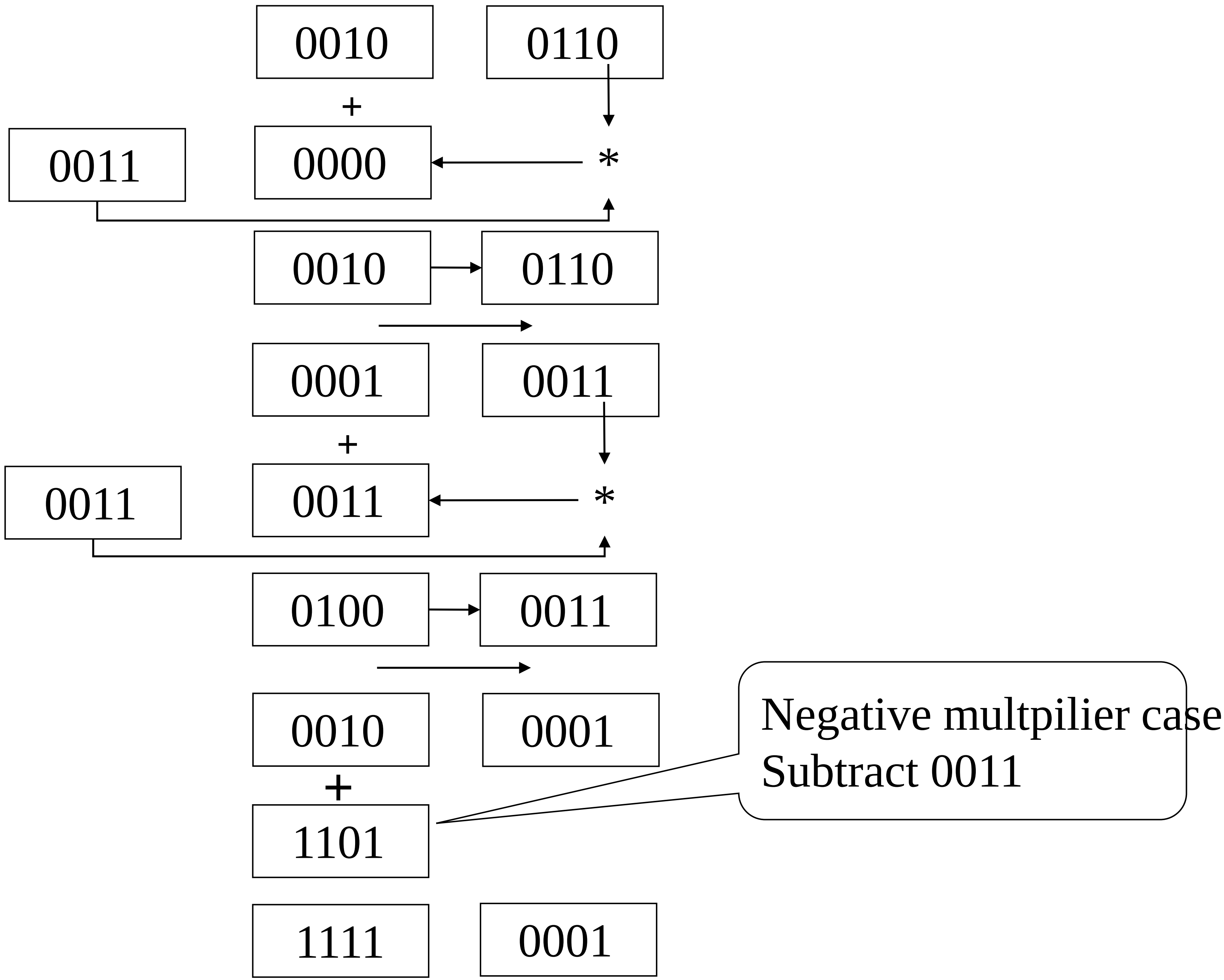


0000
1101
+
0011
0011
*
0011
1101
0001
1110
+
0011
0000
*
0001
1110
0000
1111

Multiplication: 3 x -5



$$
\begin{aligned}
& -5 \\
= \; & -0101_2 \\
= \; & 1010_2 + 1 \\
= \; & 1011_2
\end{aligned}
$$


0010
0110
+
0011
0000
*
0010
0110
0001
0011
+
0011
0011
*
0100
0011
0010
0001
+
1101
Negative multpilier case
Subtract 0011
1111
0001

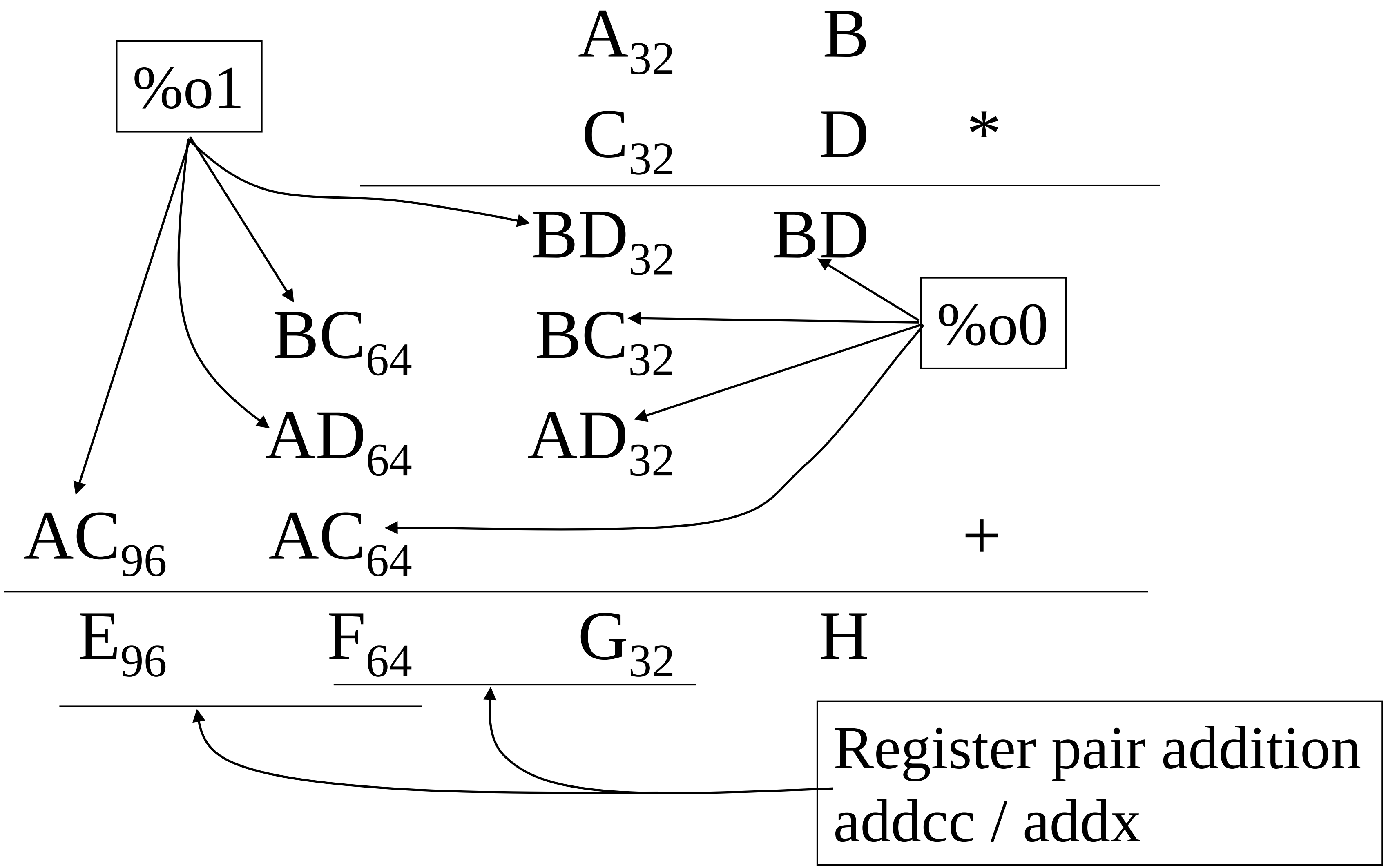
Multiplication subroutines .mul & .umul

- Multiplication subroutines
  - ✓ **.mul** : signed number
  - ✓ **.umul** : unsigned number
- Subroutine invocation: call
- Arguments
  - ✓ multiplicand: %o0
  - ✓ multiplier: %o1
- Results
  - ✓ %o0 : low order bits
  - ✓ %o1 : high order bits

✓ Ex: calc 5 * 7

```
mov  5, %o0
mov  7, %o1
call  .mul
nop
```

Multiplication in extended precision (64 bit)



Multiplication of 64-bit unsigned number

```
mov     %B,     %o0             ! B * D
call    .umul                   !  Product    %o1:  high
mov     %D,     %o1             !             %o0:  low

mov     %o0,    %H              ! H = BD
mov     %o1,    %G              ! G = BD32

mov     %B,     %o0             ! B * C
call    .umul
mov     %C,     %o1

addcc   %o0,    %G,     %G      ! G = G + BC32
addx    %o1,    %g0,    %F      ! F = BC64 ,  no carry

mov     %D,     %o0             ! A * D
call    .umul
mov     %A,     %o1
```

```
addcc    %o0,    %G,    %G     ! G = G + AD32
addxcc   %o1,    %F,    %F     ! F = AD64
addx     %g0,    %g0,   %E     !  Carry 처리

mov      %C,     %o0           ! A * C
call     .umul
mov      %A,     %o1

addcc    %o0,    %F,    %F     ! F = F + AC64
addx     %o1,    %E,    %E     ! E = AC96

mov      1, %g1
ta       0
```

- Biggest number from B * C

B:	1 1 1 1 ………… 1 1	=	$2^{32} - 1$
C:	1 1 1 1 ………… 1 1	=	$2^{32} - 1$
B*C:	1 1 1 1……. 10 0 0 ………0 1	=	$2^{64} - 2^{33} + 1$

MS 32-bit   LS 32-bit

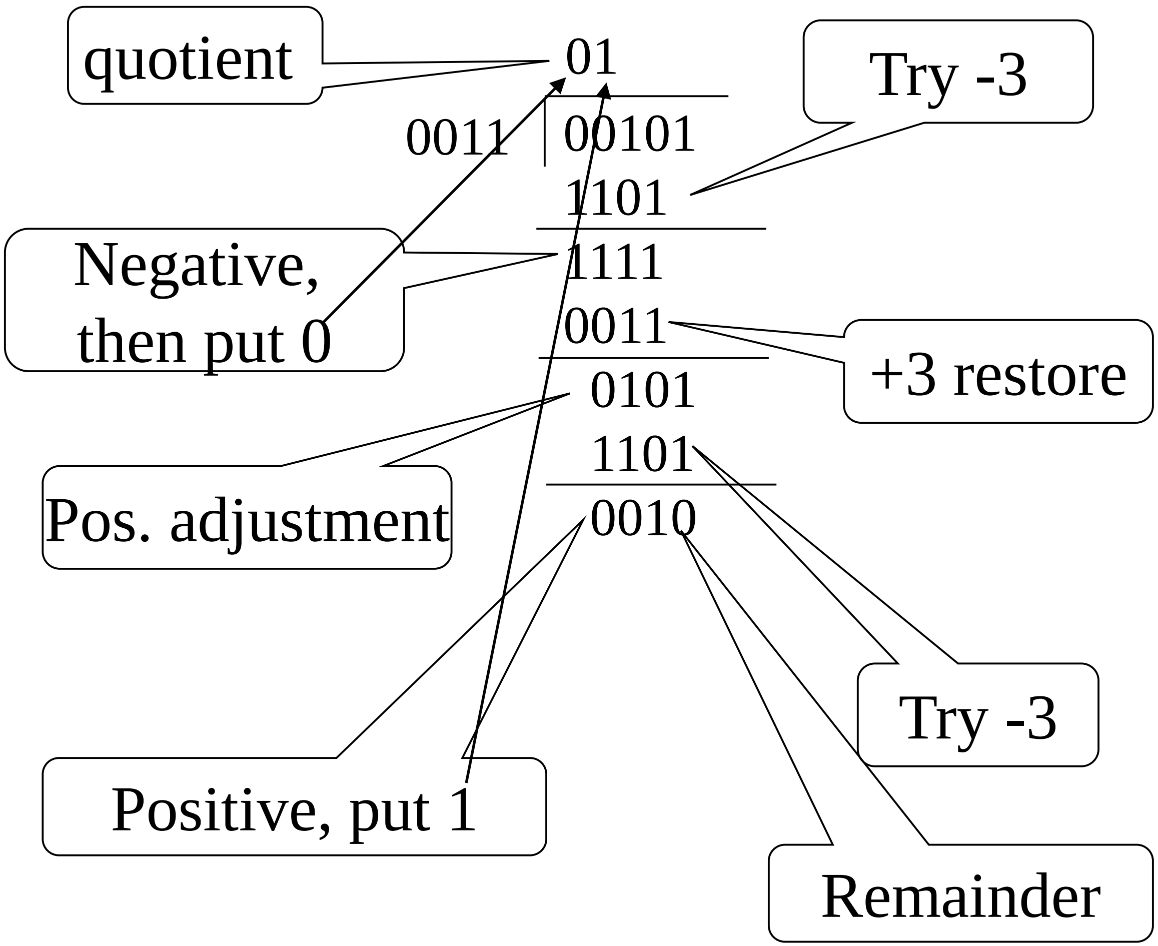
Division (1)

- Pencil and paper algorithm: $1001010_{ten} \div 1000_{ten}$

```
                            1 0 0 1 ten      (quotient)
(Divisor) 1 0 0 0 ten  | 1 0 0 1 0 1 0 ten    (dividend)
                      - 1 0 0 0
                        -------
                        0 0 0 1
                          0 0 1 0
                            0 1 0 1
                              1 0 1 0
                            - 1 0 0 0
                              -------
                              0 0 1 0 ten    (remainder)
```

Division (2)

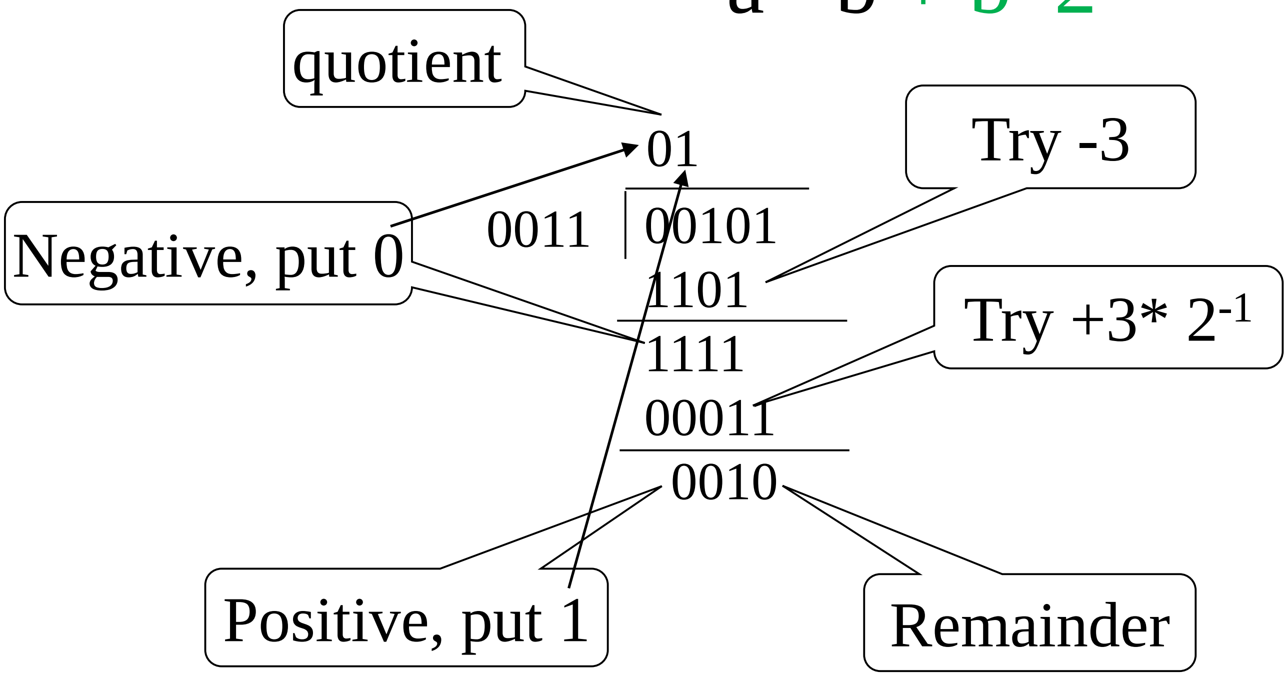
- Repetition of subtraction (ex: 5/3)



Division: non-restoring method

- $a - b + b - b*2^{-1} = a - b + (2b - b)*2^{-1}$

$$= a - b + b*2^{-1}$$



Division algorithm

```
for i = 1 to num_of_bits do
    (HR LR) << 1
    if (HR >= 0) then
        HR = HR - DIVISOR
    else
        HR = HR + DIVISOR
    endif
    if (HR > 0) then LR(lsb) = 1 endif
endfor
```

DIVIDEND

Result
- HR: Remainder
- LR: Quotient

Example

HR	LR	operation/behavior
0000	0101	HR:LR << 1
0000	1010	[HR]-3
1101		
1101	1010	LR(lsb) = 0, HR:LR << 1
1011	0100	[HR] + 3
0011		
1110	0100	LR(lsb) = 0, HR:LR << 1
1100	1000	[HR] + 3
0011		
1111	1000	LR(lsb) = 0, HR:LR << 1
1111	0000	[HR] + 3
0011		
0010	0001	LR(lsb) = 1, termination

Division subroutine

- **.div** : integer division, return quotient
- **.rem** : integer division, return remainder
- **.udiv** : pos integer div, return quotient
- **.urem** : pos integer div, return remainder
- Arguments
  - ✓ %o0: dividend
  - ✓ %o1: divisor
  - ✓ %o0: result

✓ ex: calc. 722/3

```
mov   722,  %o0
mov    3,  %o1
call   .div
nop
```

Logic and bit operations

- Logic operations
  - ✓ operation is applied to each bit independently
  - ✓ a op b ← a, b is 1-bit no.

a	0	0	1	1	logic operation	SPARC	if %g0
b	0	1	0	1	(op)	instruction	is used
	0	0	0	1	a and b	and, andcc	
	0	0	1	0	a and (not b)	andn, andncc	
	0	1	1	0	a xor b	xor, xorcc	
	0	1	1	1	a or b	or, orcc	mov, clr
	1	0	0	1	a xor (not b)	xnor, xnorcc	not
	1	0	1	1	a or (not b)	orn, orncc	

Synthetic instruction

- Logic instruction format

  op R, A, D ! R op A → D

  - ✓ R: source register **1**
  - ✓ A: source register 2 or immediate value
  - ✓ D: destination register
  - ✓ op: and, or, xor

Assembly instruction	machine insturction
not A	xnor A, *%g0*, A
not A, B	xnor A, *%g0*, B
mov A, B	or *%g0*, A, B
clr A	or *%g0*, *%g0*, A
tst A	orcc A, *%g0*, *%g0*

Example (1)

- not %o0, %l1 = xnor %o0, %g0, %l1
- mov %o0, %l1 = or %g0, %o0, %l1

%o0	00111010
%g0	00000000
%l1	11000101

xnor

%g0	00000000
%o0	00111010
	00111010

or

Example (2)

- if (a > 0)   b++;

1)

```
    cmp  %a_r,  0
    ble   next
    nop
    add  %b_r,  1,  %b_r
next:
```

2)

```
    subcc  %a_r,  %g0,  %g0
    ble    next
    nop
    add   %b_r,  1,  %b_r
next:
```

3)

```
    tst  %a_r
    ble  next
    nop
    add  %b_r,  1,  %b_r
next:
```

4)

```
    orcc  %a_r,  %g0,  %g0
    ble  next
    nop
    add  %b_r,  1,  %b_r
next:
```

1-bit (flag) manipulation synthetic instruction

assembly instruction	machine instruction	
bset A, R	or R, A, R	set bit to 1
bclr A, R	andn R, A, R	set bit to 0
btog A, R	xor R, A, R	reverse bit
btst A, R	andcc R, A, %g0	bit test

- ✓ A: mask use to specify positions of bits to manipulate
- ✓ Ex: A: 0011 R: 0101

bset A, R	R: 0111		btog A, R	R: 0110
bclr A, R	R: 0100		btst A, R	0001

Example

- Check if there exists either 0x10 or 0x8 bit in register %a_r

```
        btst   0x18, %a_r
        be     clear
        nop
set:
        :
clear:
        :
```

%a_r	………………..? ? …
0x18	0000 … 0000 0001 1000
andcc	
result	0000 … 0000 000? ?000

??	??	result
00	00	0
01	01	not 0
10	10	not 0
11	11	not 0

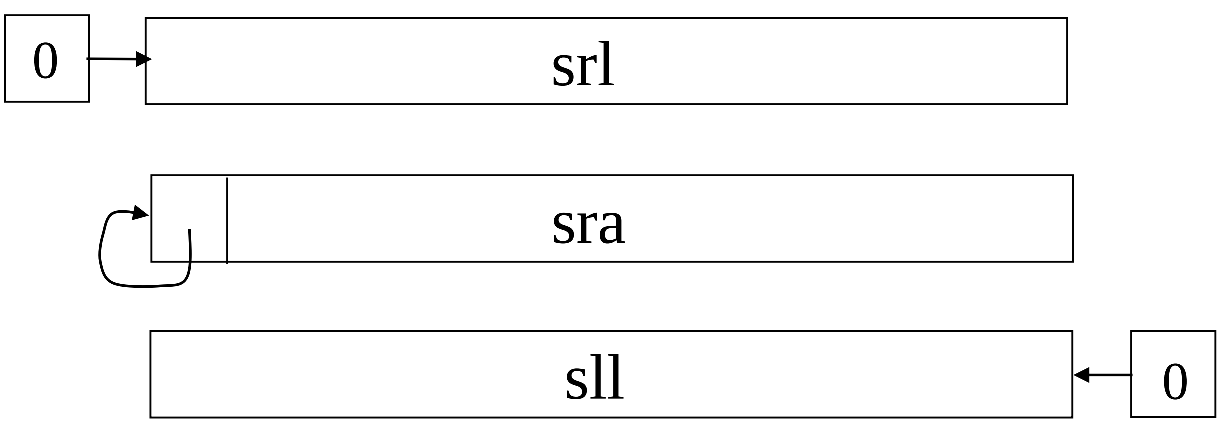
Shift operations

- **sll** (shift left logical)
- **sra** (shift right arithmetic)
- **srl** (shift right logical)
- Format
  - ✓ Op R, A, D
    - ✓ **R, D: registers**
    - ✓ **A: register or immediate value**

      How many bits to be shifted? specified in least significant 5 bits of A or r[A]

arithmetic: numeric data
logical: non-numeric data

Shift n-bit left = multiply by 2^n



- Example
  - ✓ sll %l0, 2, %o0

```
=> mov  %l0, %o0
   mov  4, %o1
   call .mul
   nop
```

  - ✓ %l0 * 5 101_2

```
mov  %l0, %o0
sll  %l0, 2, %l0
add  %l0, %o0, %o0
```

Characters

- 7-biy ASCII(American Standard Code for Information Interchange)

- Using ASCII in assembly program

  - ✓ mov 0141, %l0
  - ✓ mov 0x61, %l0
  - ✓ mov 'a', %l0
  - ✓ mov "a", %l0

Adder/substractor