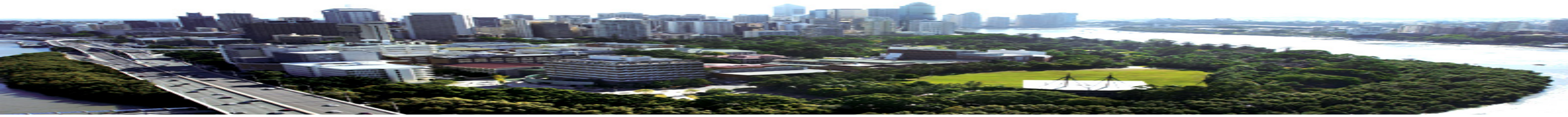
ENB 350 Real-time Computer based systems

Lecture 3 – EXTENDED PRECISION ARITHMETIC

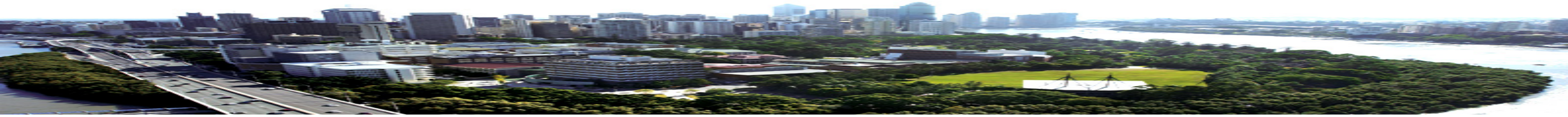
V. Chandran



CRICOS No. 00213J

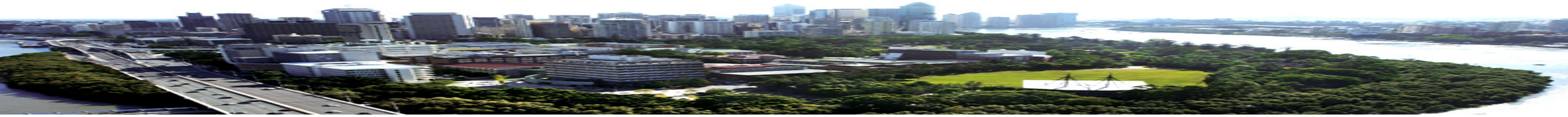
Contents

- Review of R4000 instructions
- The stack – push and pop
- Floating point and Fixed point
- Fixed point 8.8 multiplication
- Extended precision 16.16 multiplication



Example

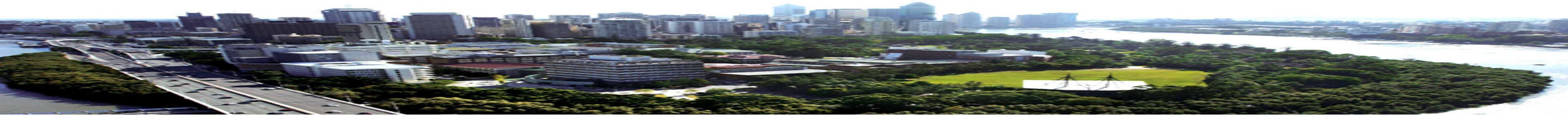
If CF = 1 A = 00110110 and contents of the memory address given by HL are 00110111 what is the result in A?



Example

If CF = 1 A = 00110110 and contents of the memory address given by HL are 00110111 what is the result in A?

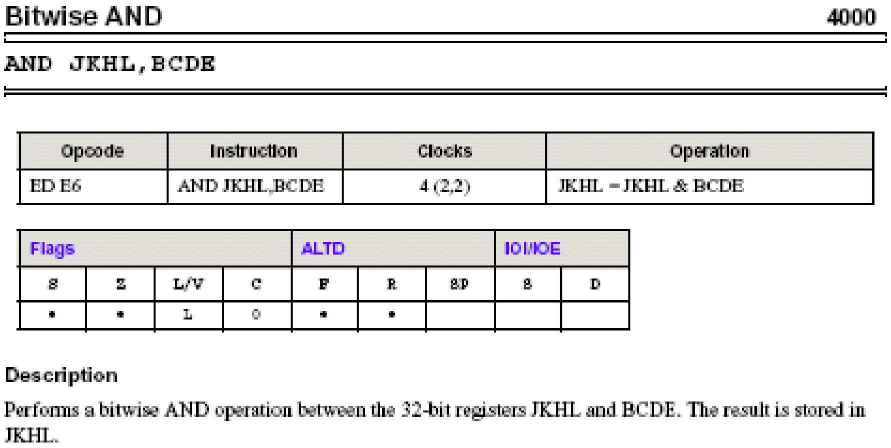
$$\begin{array}{r} 1 \\ 00110110 \\ \underline{00110111} \\ 01101110 \end{array}$$



R4000 instruction (example 2)

32 bit operation

Bitwise AND



Bitwise AND 4000

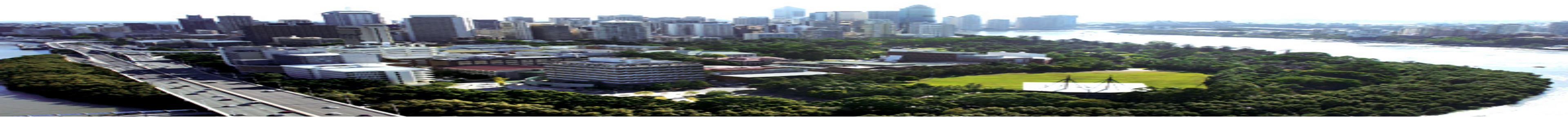
```
AND JKHL,BCDE
```

Opcode	Instruction	Clocks	Operation
ED E6	AND JKHL,BCDE	4 (2,2)	JKHL = JKHL & BCDE

Flags				ALTD			IOI/IOE	
S	Z	L/V	C	F	R	SP	S	D
•	•	L	0	•	•			

Description

Performs a bitwise AND operation between the 32-bit registers JKHL and BCDE. The result is stored in JKHL.



Example

In HEX

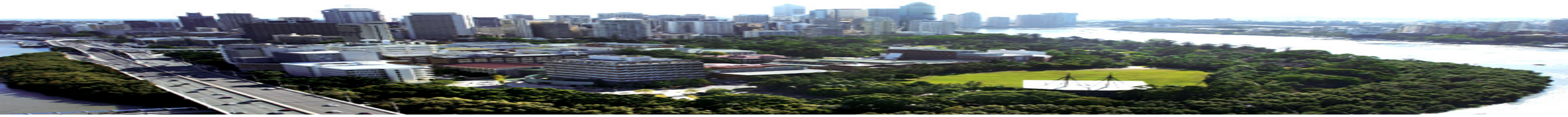
J=00, K=10, H=21,L=30,B=01,C=11,D=20,E=13 what is the result?

JKHL = 00000000000100000010000100110000

BCDE= 00000001000100010010000000010011

Result=

This is 0x ? and will be stored in ?.



Example

In HEX

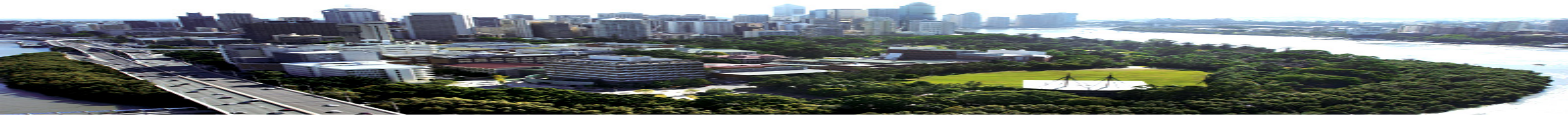
J=00, K=10, H=21,L=30,B=01,C=11,D=20,E=13 what is the result?

JKHL = 00000000000100000010000100110000

BCDE= 00000001000100010010000000010011

Result= 00000000000100000010000000010000

This is 0x00102010 and will be stored in JKHL.

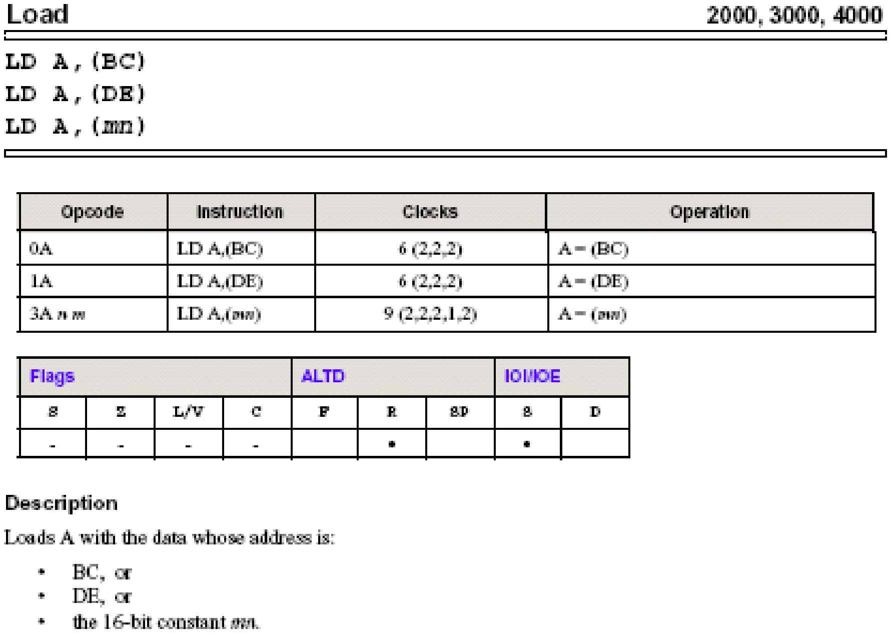


R4000 instruction (example 3)

8 bit operation, loads A

With contents of

Memory Address (16 bit) in register pair BC or DE or immediate constant



Load 2000, 3000, 4000

```
LD A,(BC)
LD A,(DE)
LD A,(mn)
```

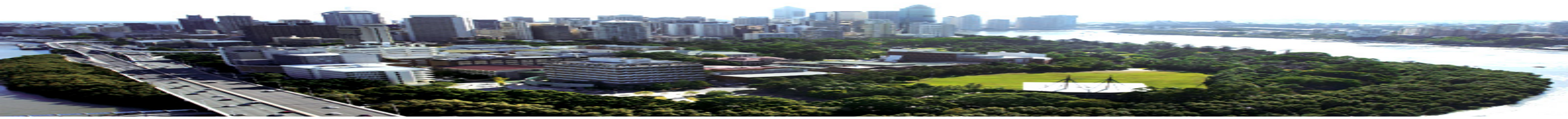
Opcode	Instruction	Clocks	Operation
0A	LD A,(BC)	6 (2,2,2)	A = (BC)
1A	LD A,(DE)	6 (2,2,2)	A = (DE)
3A *n m*	LD A,(*mn*)	9 (2,2,2,1,2)	A = (*mn*)

Flags				ALTD			IOI/IOE	
S	Z	L/V	C	F	R	SP	S	D
-	-	-	-		•		•	

Description

Loads A with the data whose address is:

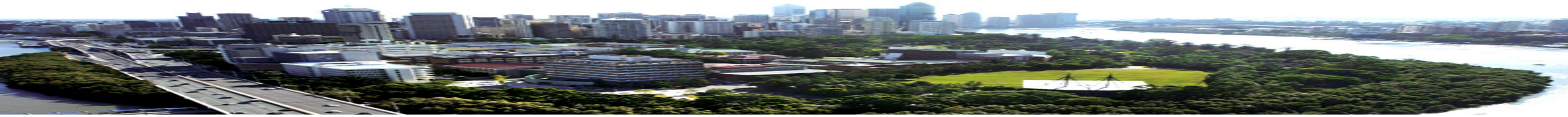
- BC, or
- DE, or
- the 16-bit constant *mn*.



LD A,(0x0347)

A = ?

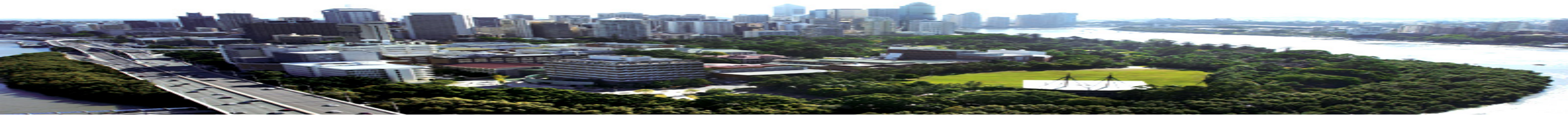
	Address (in Hex)	Data (in binary)	
	---	---	
	034C	10011100	
	034B	11001110	
	034A	10011101	
	0349	10001000	
	0348	01101010	
	0347	01000111	
	0346	01000110	
	0345	00010001	
	---	---	



LD A,(0x0347)

A = 01000111

	Address (in Hex)	Data (in binary)	
	---	---	
	034C	10011100	
	034B	11001110	
	034A	10011101	
	0349	10001000	
	0348	01101010	
→	0347	01000111	
	0346	01000110	
	0345	00010001	
	---	---	

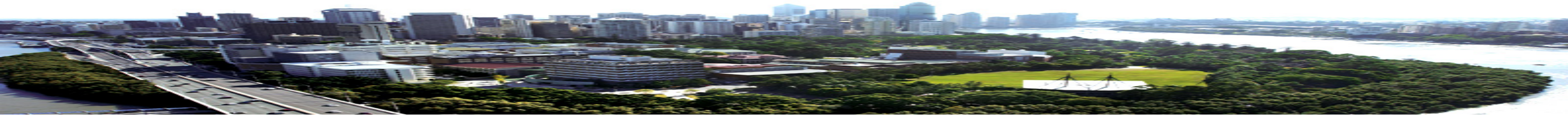


LD (HL),BCDE

HL = 0x0348
B=00000001
C=00000010
D=00000011
E=00000100

What are the new values in the table?

	Address (in Hex)	Data (in binary)	
	---	---	
	034C	10011100	
	034B	11001110	
	034A	10011101	
	0349	10001000	
	0348	01101010	
	0347	01000111	
	0346	01000110	
	0345	00010001	
	---	---	



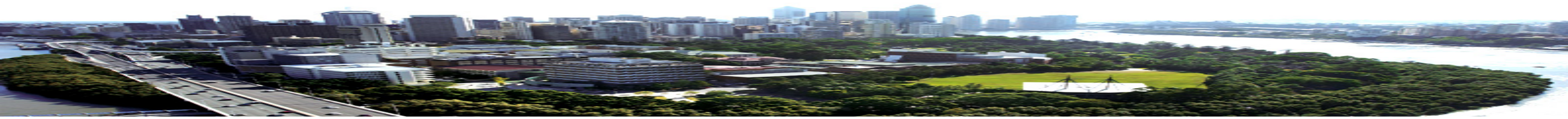
LD (HL),BCDE

HL = 0x0348
B=00000001
C=00000010
D=00000011
E=00000100

What are the new values in the table?

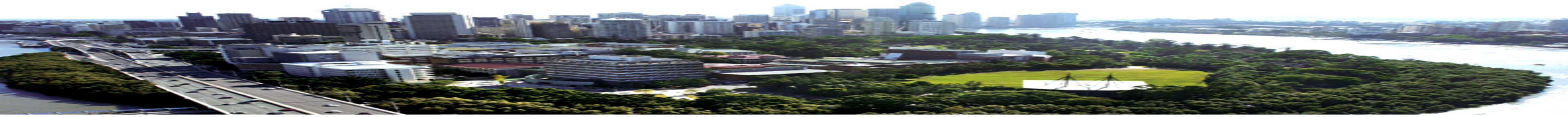
Note: “little endian”
Low order byte first

	Address (in Hex)	Data (in binary)	
	---	---	
	034C	10011100	
	034B	00000001	B
	034A	00000010	C
	0349	00000011	D
→	0348	00000100	E
	0347	01000111	
	0346	01000110	
	0345	00010001	
	---	---	



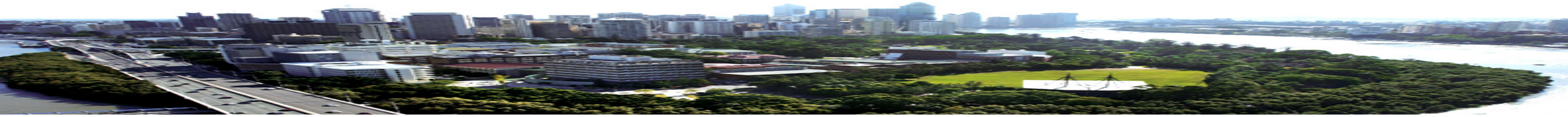
R4000 arithmetic instructions

- 8 bit, 16 bit and 32 bit operations
- Add, Subtract
- 16 bit Multiply unsigned (MULU) and signed (MUL) with result in 32 bit register
- No divide, No floating point co processor
- Dynamic C uses library functions for real arithmetic



Stack

- Special region of memory in which data is ordered
- Data stored in same order as written; retrieved in reverse order
- LIFO (Last In First Out)
- Stack Pointer – holds address of stack top
- PUSH and POP instructions
- Stack used extensively by procedure calls to store return address, parameters and local variables
- In assembly code sp refers to the stack pointer and (sp) the contents of the stack at sp.

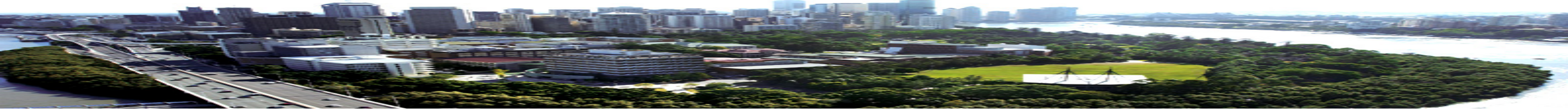


POP BCDE

SP = 0xD949

B = ?
C = ?
D = ?
E = ?
SP = ?

	Address (in Hex)	Data (in binary)	
	---	---	
	D945	10011100	
	D946	11001110	
	D947	10011101	
	D948	10001000	
→	D949	01101010	
	D94A	01000111	
	D94B	01000110	
	D94C	00010001	
	D94D	00100100	



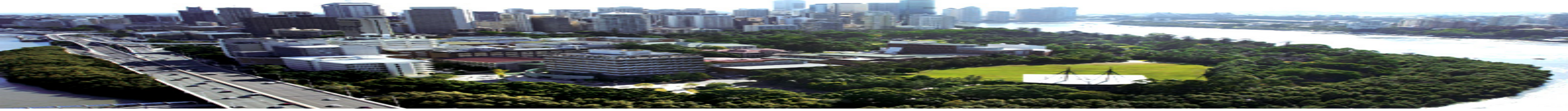
POP BCDE

SP = 0xD949

B = 00010001
C = 01000110
D = 01000111
E = 01101010
SP = 0xD94D

Note: Stack grows towards lower addresses. Data is stored little endian in memory.

	Address (in Hex)	Data (in binary)	
	---	---	
	D945	10011100	
	D946	11001110	
	D947	10011101	
	D948	10001000	
	D949	01101010	E
	D94A	01000111	D
	D94B	01000110	C
	D94C	00010001	B
→	D94D	00100100	



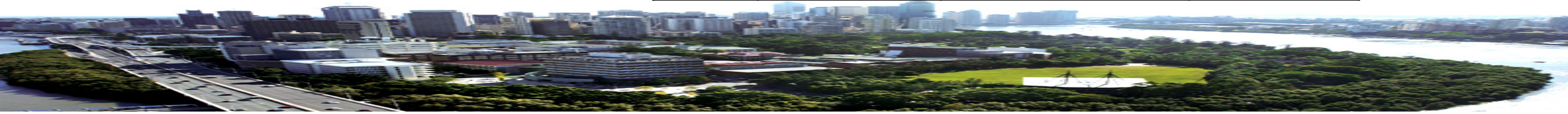
PUSH BCDE

SP = 0xD949

B = 00000001
C = 00000010
D = 00000011
E = 00000100

New stack entries = ?
SP = ?

	Address (in Hex)	Data (in binary)	
	---	---	
	D945	10011100	
	D946	11001110	
	D947	10011101	
	D948	10001000	
→	D949	01101010	
	D94A	01000111	
	D94B	01000110	
	D94C	00010001	
	D94D	00100100	



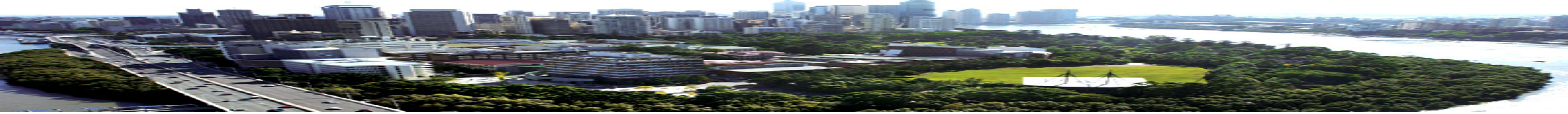
PUSH BCDE

SP = 0xD949

B = 00000001
C = 00000010
D = 00000011
E = 00000100

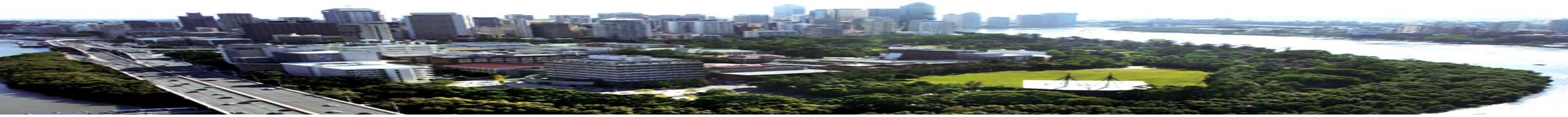
New stack entries = ?
SP = 0xD945?

	Address (in Hex)	Data (in binary)	
	---	---	
→	D945	00000100	E
	D946	00000011	D
	D947	00000010	C
	D948	00000001	B
	D949	01101010	
	D94A	01000111	
	D94B	01000110	
	D94C	00010001	
	D94D	00100100	



Why?

- Computers use binary arithmetic
- Conversion to decimal required for human readable displays
- Numbers with fractional parts (real numbers) require special representation and arithmetic is performed with algorithms implemented as a library that may be speeded up with a co-processor.



Conversion: binary to decimal

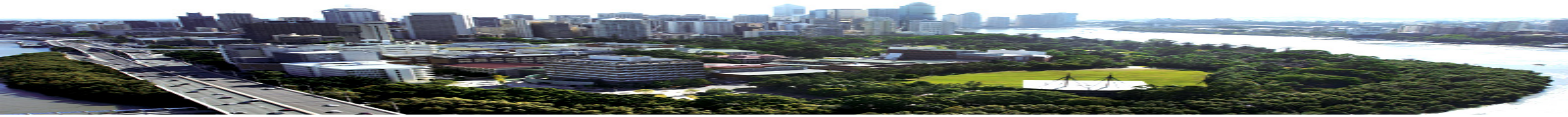
We can use polynomial evaluation: (note: this is ‘human’ conversion. Computer representation of decimal may be in binary 2s complement or binary coded decimal and each decimal digit will need to be separated and displayed)

10110101_2

$= \quad 1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 +$
$0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

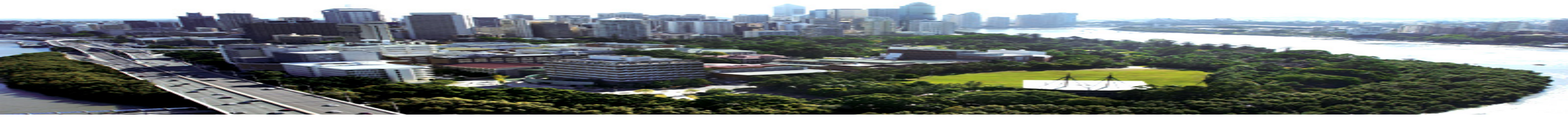
$= \quad 128 + 32 + 16 + 4 + 1$

$= \quad 181_{10}$



decimal to binary

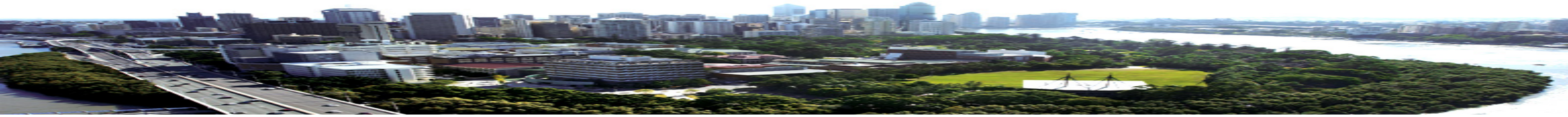
- Whole part and fractional parts must be handled separately!
  - – Whole part: Use *repeated division*.
  - – Fractional part: Use *repeated multiplication*.
  - – Combine results when finished.
- Example: 97.1 = ?



Integer part – repeated division

97 ÷ 2 ➔	quotient = 48,	remainder = 1 (LSB)
48 ÷ 2 ➔	quotient = 24,	remainder = 0.
24 ÷ 2 ➔	quotient = 12,	remainder = 0.
12 ÷ 2 ➔	quotient = 6,	remainder = 0.
6 ÷ 2 ➔	quotient = 3,	remainder = 0.
3 ÷ 2 ➔	quotient = 1,	remainder = 1.
1 ÷ 2 ➔	quotient = 0 (Stop)	remainder = 1 (MSB)

Result = 1 1 0 0 0 0 1



fractional part – repeated multiplication

.1 × 2 ➔ 0.2 (fractional part = .2, whole part = 0)

.2 × 2 ➔ 0.4 (fractional part = .4, whole part = 0)

.4 × 2 ➔ 0.8 (fractional part = .8, whole part = 0)

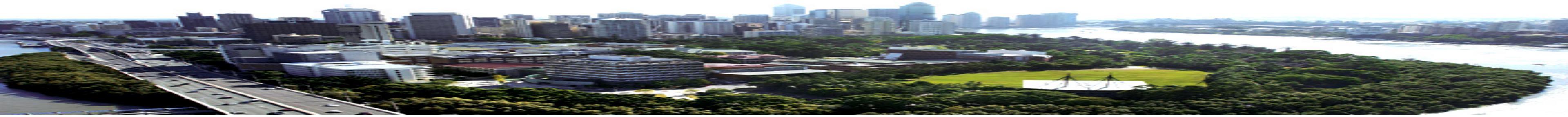
.8 × 2 ➔ 1.6 (fractional part = .6, whole part = 1)

.6 × 2 ➔ 1.2 (fractional part = .2, whole part = 1)

Result = .00011…..

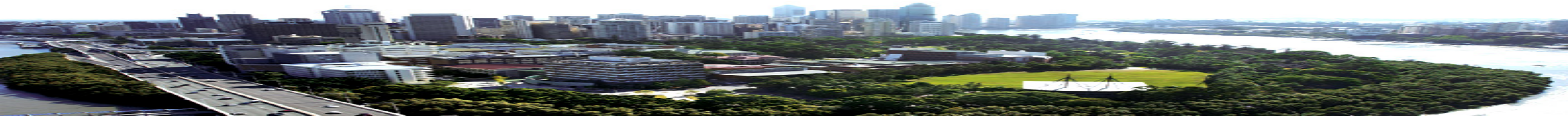
(could continue on)

97.1 = 100001.00011…



Finite word length

- Computers represent data with a finite number of bits.
- Real numbers (such as 97.1 in the previous example) cannot be EXACTLY represented in binary with finite number of bits
- Testing for equality on real numbers is not good practice
- Integers can also overflow and rollover.

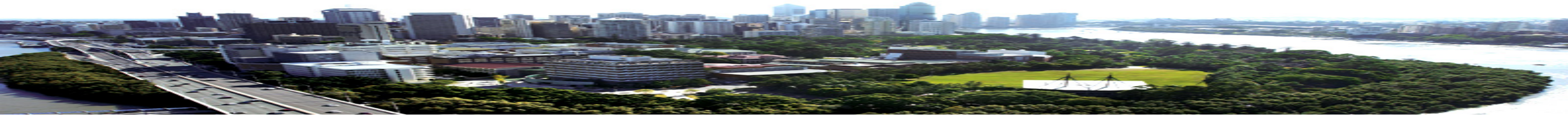


Signed 2s complement

Sign bit : 0 = positive 1 = negative

Number	Sign magnitude	1s complement	2s complement
1	0 0000001		00000001
2	0 0000010		00000010
57	0 0111001		00111001
Negative numbers			
-1	1 0000001	1 1111110	11111111
-2	1 0000010	1 1111101	11111110
-57	1 0111001	1 1000110	11000111

Complement all bits except sign



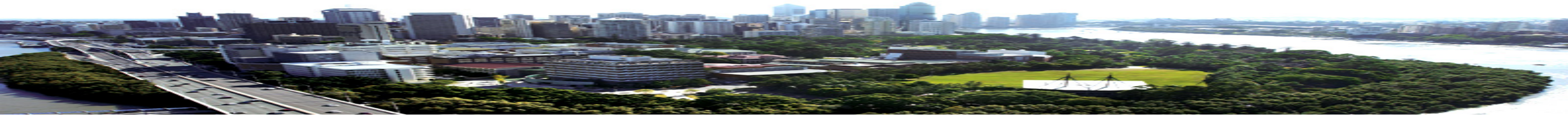
As polynomial expansions

$57 = 00111001 = 0\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$

$-57 = 11000111 = -1\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$

For the 2s complement representation

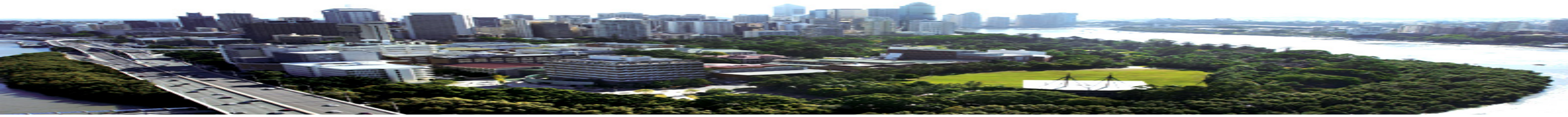
(Verify this)



Signed or Unsigned

$$167_{10} \xleftarrow{\text{unsigned}} 10100111_2 \xrightarrow{\text{Signed (2s Comp)}} -89_{10}$$

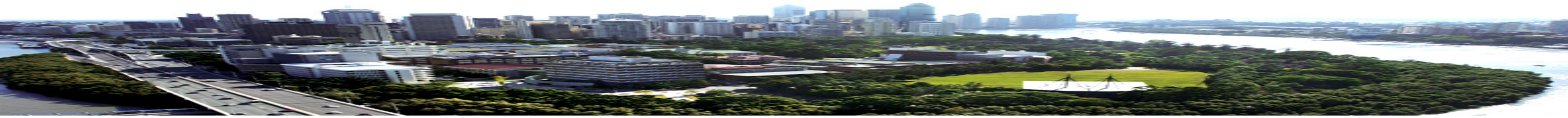
- Signed or Unsigned is a matter of interpretation
- <u>a single bit pattern can represent two different values.</u>



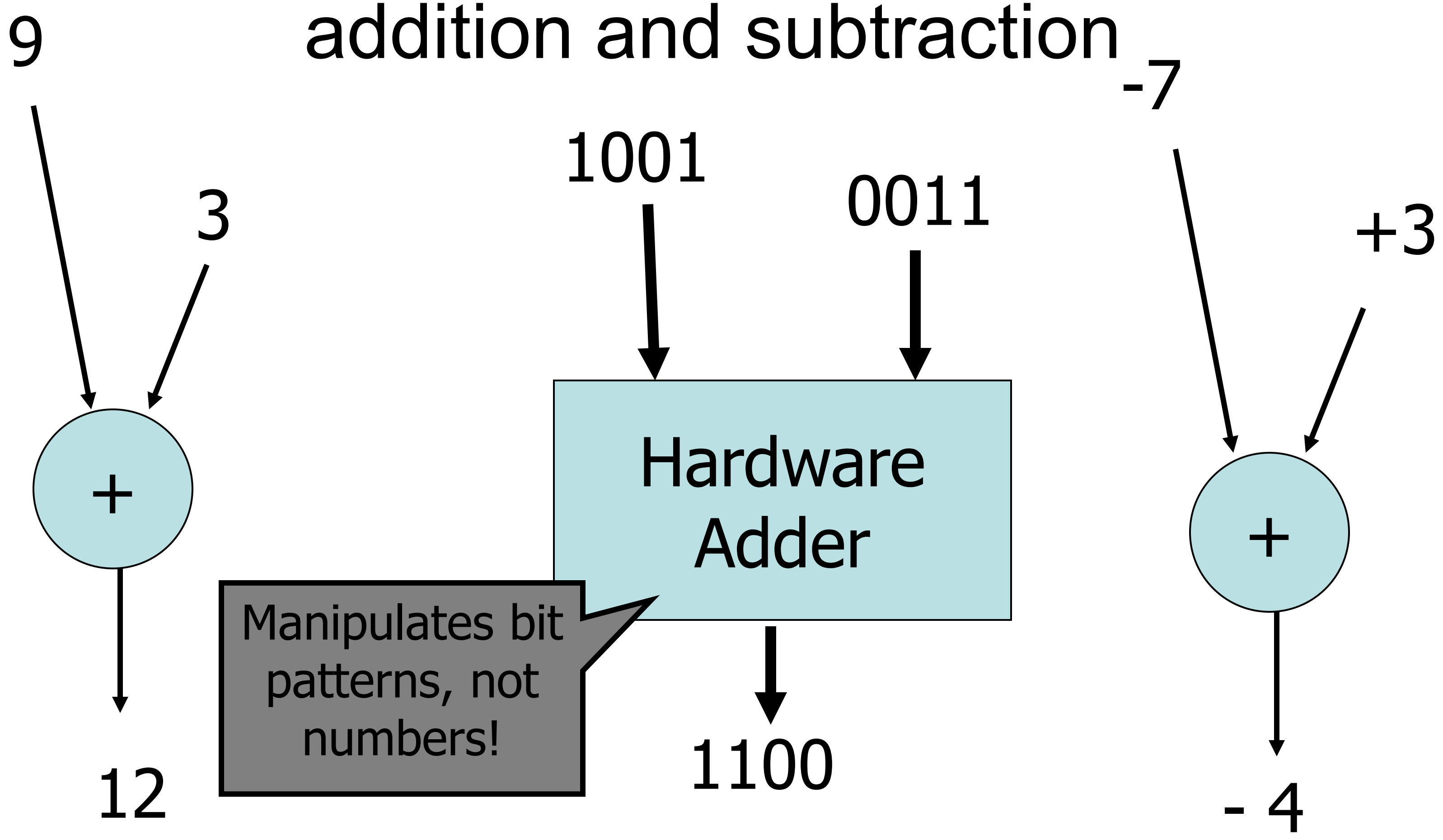
Why 2s complement?

+3	011
+2	010
+1	001
0	000
-1	111
-2	110
-3	101
-4	100

1. Just as easy to determine sign as in sign-magnitude form.
2. Almost as easy to change the sign of a number. 1s complement and add 1.
3. Addition can proceed without worrying about which operand is larger.
4. A single zero!
5. One hardware adder works for both signed and unsigned operands.



One hardware Adder handles both addition and subtraction



9

3

+

12

1001

0011

Hardware Adder

1100

Manipulates bit patterns, not numbers!

-7

+3

+

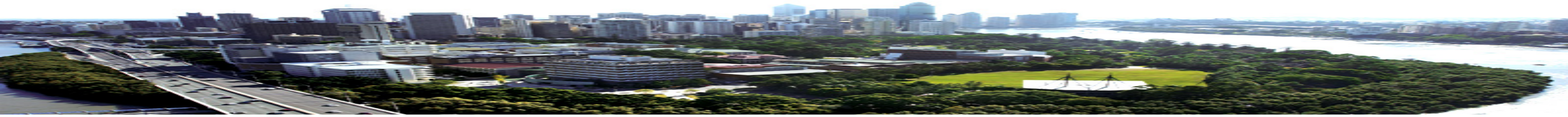
- 4

Which is greater? 1001 or 0011

- Answer: It depends!
  - (unsigned: 9 > 3, signed: -7 < 3)

- So how does the computer decide:
- “if (x > y)..”   /* Is this true or false? */

- It’s a matter of interpretation, and depends on how x and y were declared: signed? Or unsigned?



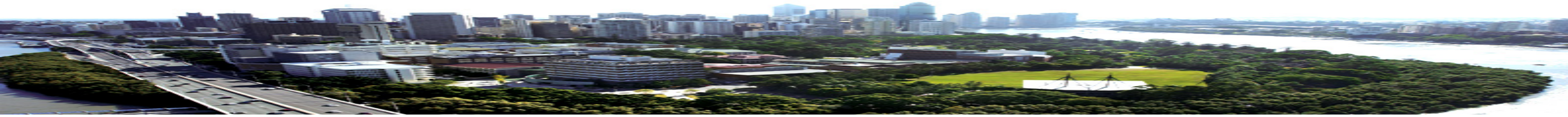
Unsigned Overflow

```
  1100      (12)
 +0111      ( 7)
 -----
 10011
Lost ↑

   ↓ (Result limited by word size)

  0011      ( 3)     wrong
```

Value of lost bit is 2^n (16).

16 + 3 = 19

(The right answer!)



Signed Overflow

-120_{10} ➔ 10001000_2

$\underline{-17}_{10}$ $+\underline{11101111}_2$

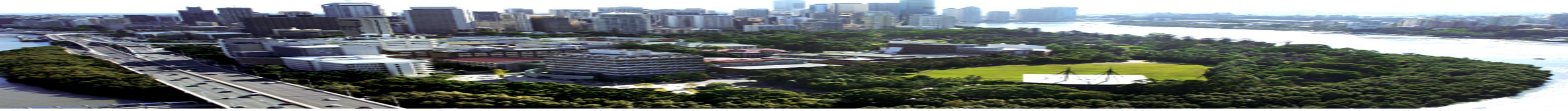
sum: -137_{10} 101110111_2

01110111_2 (keep 8 bits)

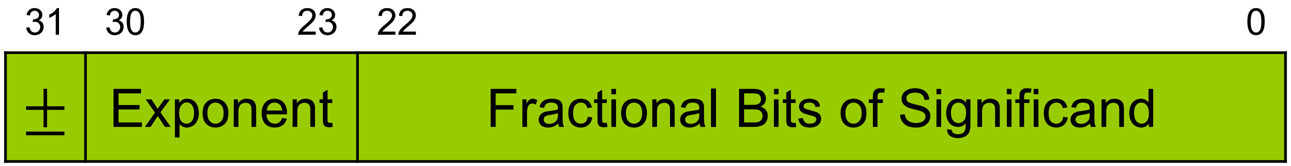
$(+119_{10})$ **wrong**

Note: $119 - 2^8 = 119 - 256 = -137$

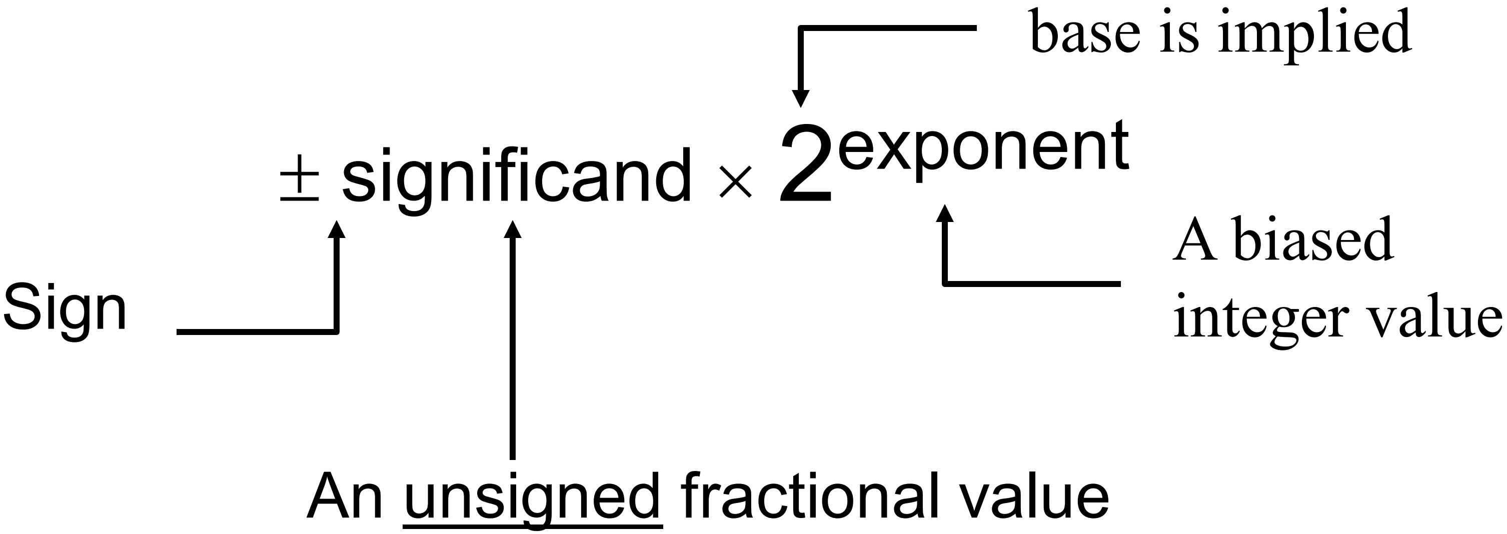
This means that algorithms can be devised to use the overflow bit and extend arithmetic to longer word lengths

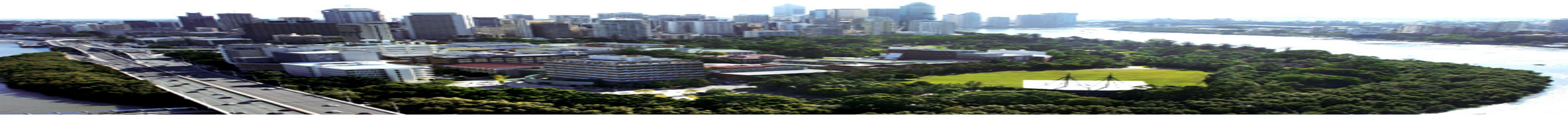


Floating-Point Real



Three components:





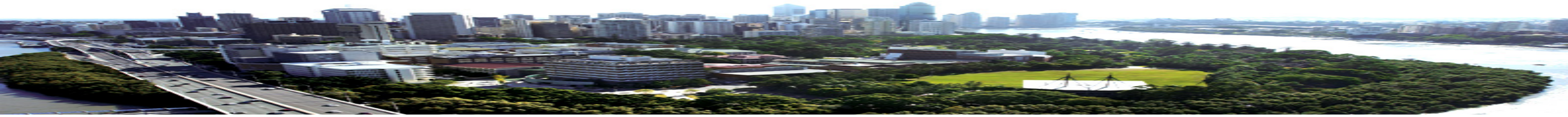
Floating pt representation of -2

-2

Sign bit = 1 being negative

$2 = 1.0 \times 2^1$

Leading 1 is ignored in the significand because it is always 1
Exp bias is +127
Thus S =1
Significand = .00000000000000000000000
Exponent = 1 + 127 = 128 = 10000000



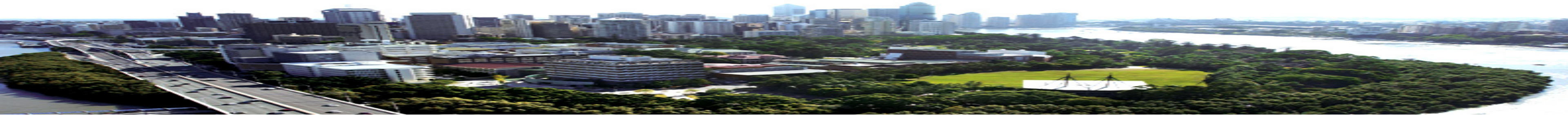
Floating pt representation of -0.75

-0.75

Sign bit = 1 being negative

$0.75 = 1.5 \times 2^{-1}$

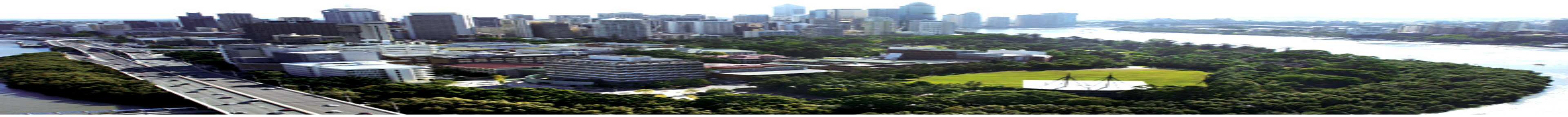
Leading 1 is ignored in the significand because it is always 1
Exp bias is +127
Thus S =1
Significand = .10000000000000000000000   Note this is the fractional part
Exponent = -1 + 127 = 126 = 01111110



CRICOS No. 00213J

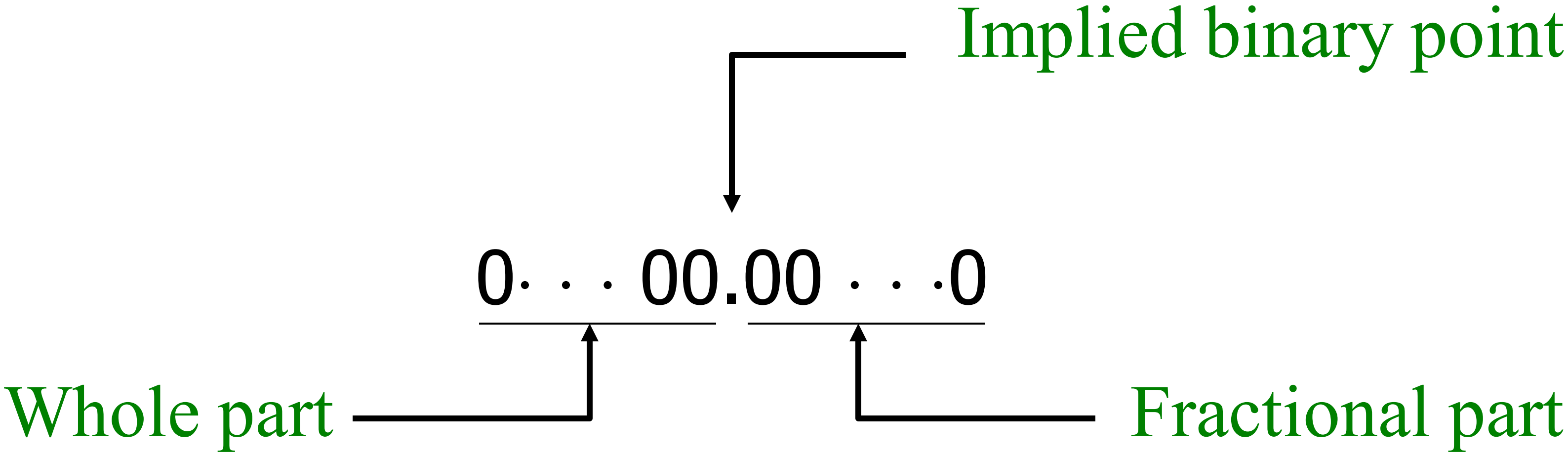
Single-precision Floating-point Representation

```
        S  Exp+127           Significand

 2.000 0 10000000 (1).00000000000000000000000
 1.000 0 01111111 (1).00000000000000000000000
 0.750 0 01111110 (1).10000000000000000000000
 0.500 0 01111110 (1).00000000000000000000000
 0.000 0 00000000 (0).00000000000000000000000
-0.500 1 01111110 (1).00000000000000000000000
-0.750 1 01111110 (1).10000000000000000000000
-1.000 1 01111111 (1).00000000000000000000000
-2.000 1 10000000 (1).00000000000000000000000
```



Fixed-Point Real

Three components:

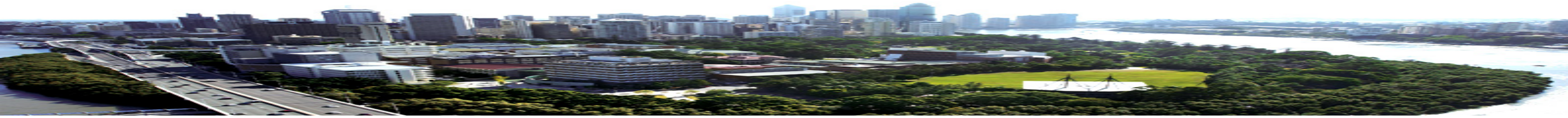


Implied binary point

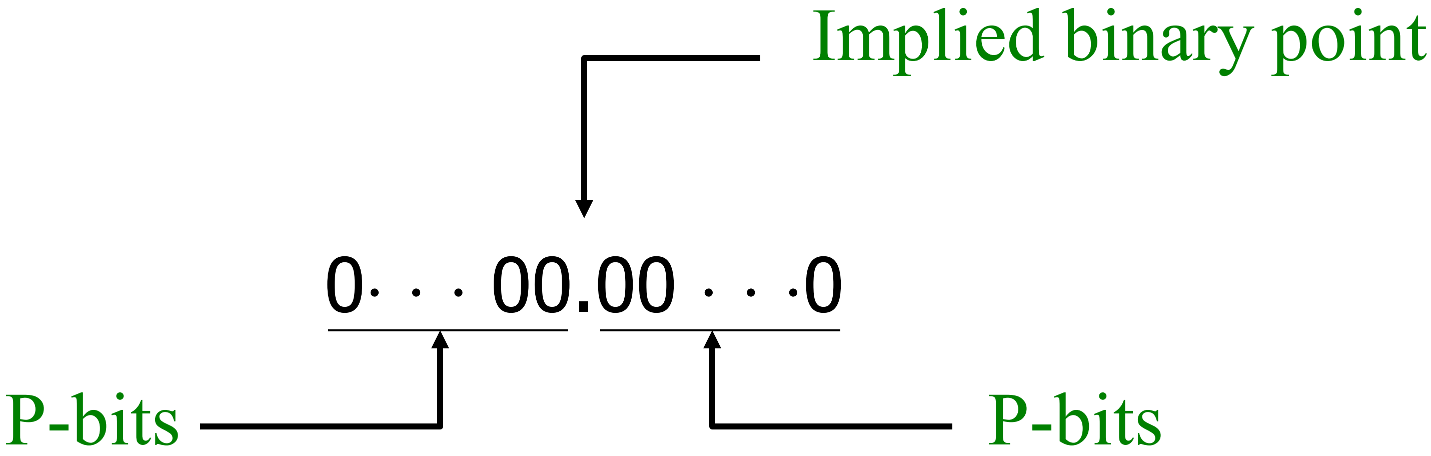
0· · · 00.00 · · ·0

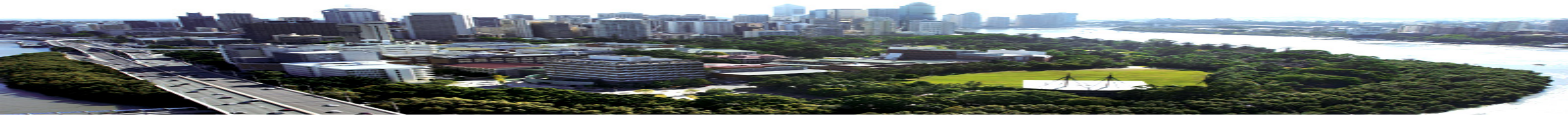
Whole part

Fractional part



P.P Fixed-Point Format

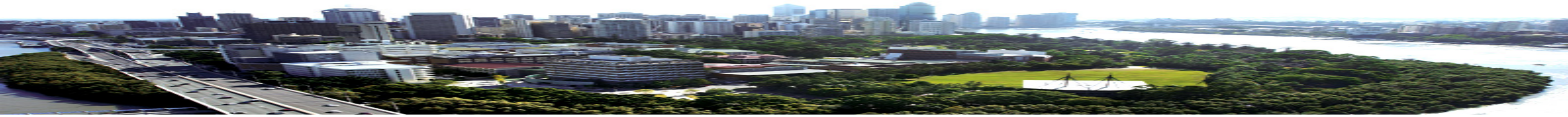




Scale factor

There is an assumed scale factor (which converts the integer to the fixed point and moves the binary point left by P bits)

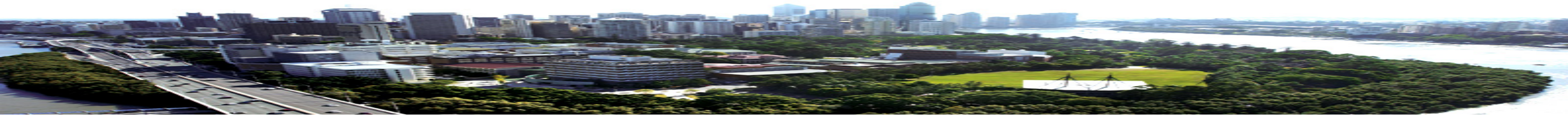
0001.0000 = 1 (16×2^{-4})



Addition

Align binary points. Same as integer addition. Correct if the result is within the range of representation.

$$\begin{array}{ll}
0001.0000 = 1 & (16 \;\; \text{x}\; 2^{-4}\;) \\
0001.1000 = 1.5 & (24 \;\; \text{x}\; 2^{-4}\;) \\
\hline
0010.1000 = 2.5 & (40 \;\; \text{x}\; 2^{-4}\;)
\end{array}$$

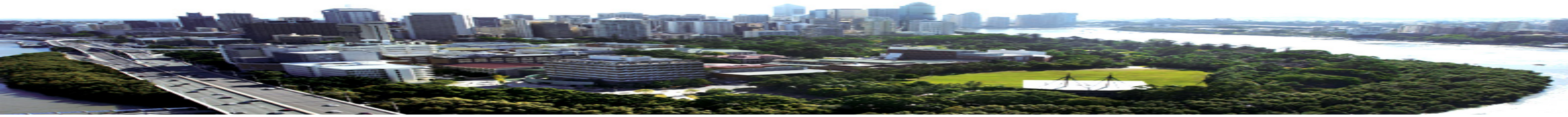


Negative number / subtraction

Align binary points. Same as integer operation. Correct if the result is within the range of representation.

$$\begin{array}{ll} 0001.0000 = 1 & (16 \ \mathrm{x} \ 2^{-4}) \\ \underline{1110.1000 = -1.5} & \underline{(-24 \ \mathrm{x} \ 2^{-4})} \\ 1111.1000 = -0.5 & (-8 \ \mathrm{x} \ 2^{-4}) \end{array}$$

Check and convince yourself that the above 2s complement representations of negative numbers are correct.



Multiplication

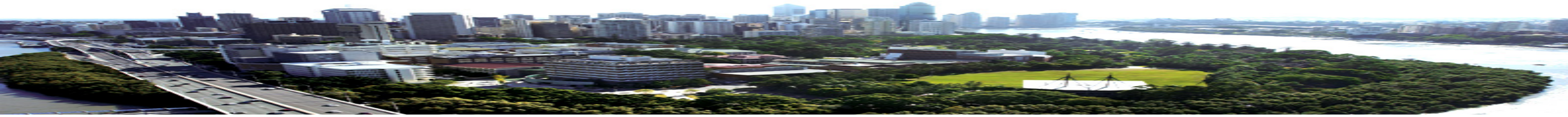
Align binary points. Binary point shifts and loss of bits can occur at MSB and LSB.

0001.0000 = 1 (16 x 2^{-4})

1110.1000 = -1.5 (-24 x 2^{-4})

11111110.10000000 = -1.5 (-384 x 2^{-8})

Check and convince yourself that the above 2s complement representation is actually -384. Note: sign extensions are made when negative numbers in 2s complement are extended in length



Fixed vs. Floating

- Floating-Point:

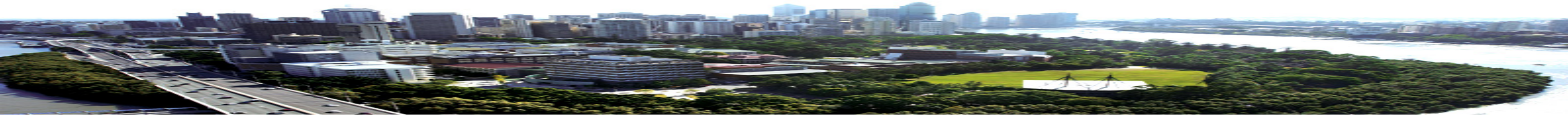
  Pro: Large dynamic range determined by exponent; resolution determined by significand.

  Con: Implementation of arithmetic in hardware is complex (slow).

- Fixed-Point:

  Pro: Arithmetic is implemented using regular integer operations of processor (fast).

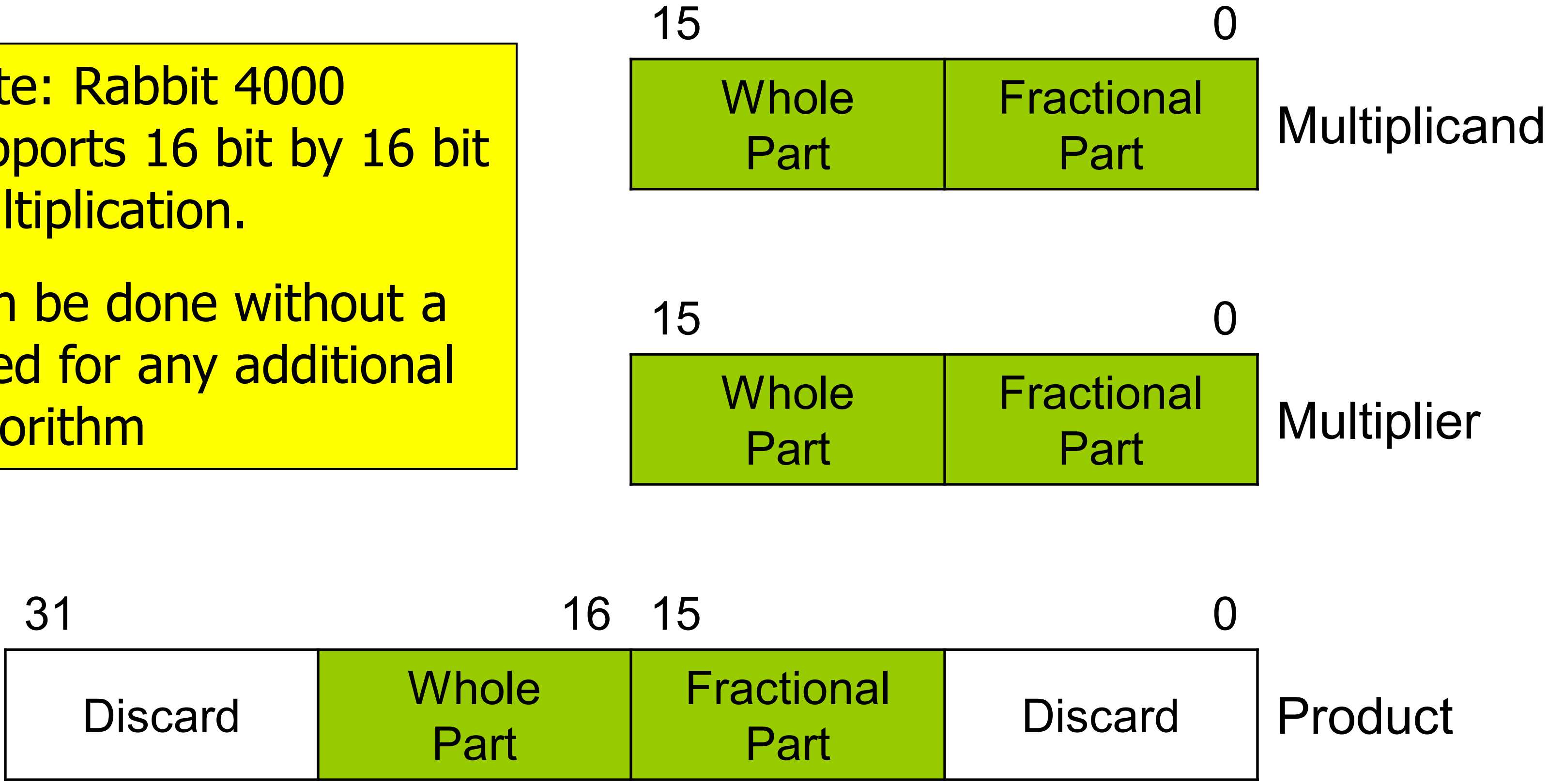
  Con: Limited range and resolution.



8.8 Fixed-Point Multiplication

Note: Rabbit 4000 supports 16 bit by 16 bit multiplication.

Can be done without a need for any additional algorithm



15 0

Whole Part | Fractional Part

Multiplicand

15 0

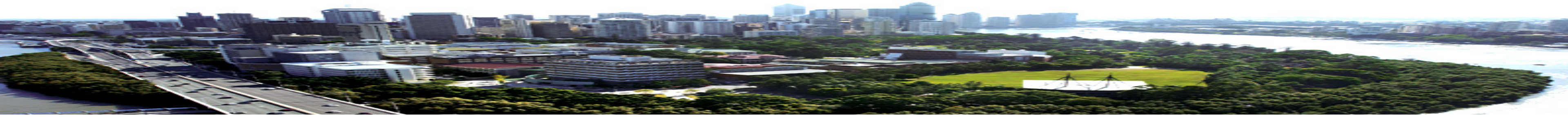
Whole Part | Fractional Part

Multiplier

31 16 15 0

Discard | Whole Part | Fractional Part | Discard

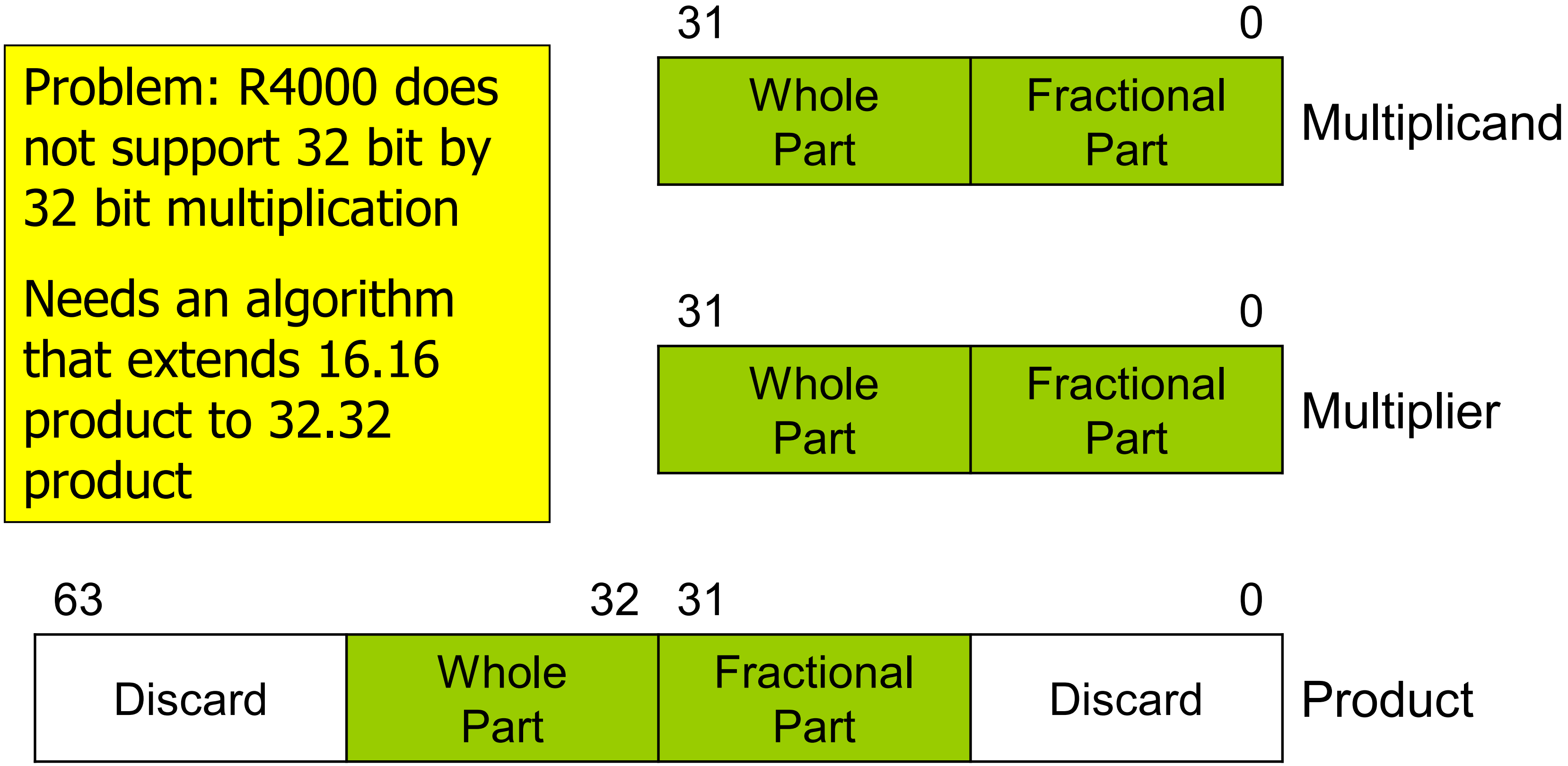
Product



16.16 Fixed-Point Multiplication

Problem: R4000 does not support 32 bit by 32 bit multiplication

Needs an algorithm that extends 16.16 product to 32.32 product



31 0

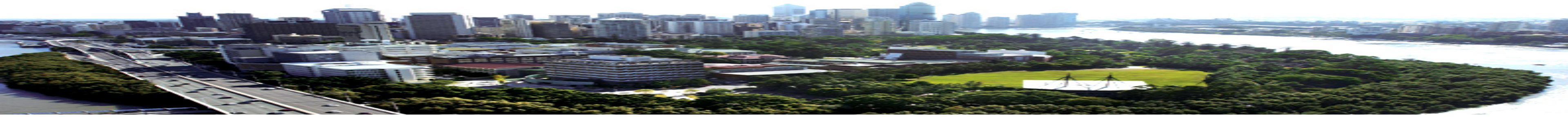
Whole Part | Fractional Part — Multiplicand

31 0

Whole Part | Fractional Part — Multiplier

63 32 31 0

Discard | Whole Part | Fractional Part | Discard — Product



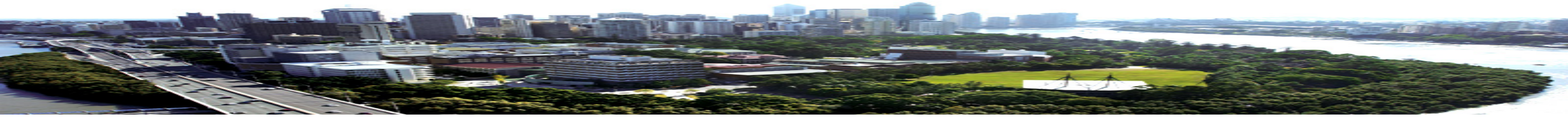
16.16 Fixed-Point Multiplication

First consider a 32-bit unsigned number:

$$\begin{aligned} A_u &= 2^{31}A_{31} + 2^{30}A_{30} + \ldots + 2^{0}A_{0} \\ &= 2^{31}A_{31} + (2^{30}A_{30} + \ldots + 2^{0}A_{0}) \\ &= 2^{31}A_{31} + A_{30..0} \end{aligned}$$

where $A_{30..0} = 2^{30}A_{30} + \ldots + 2^{0}A_{0}$

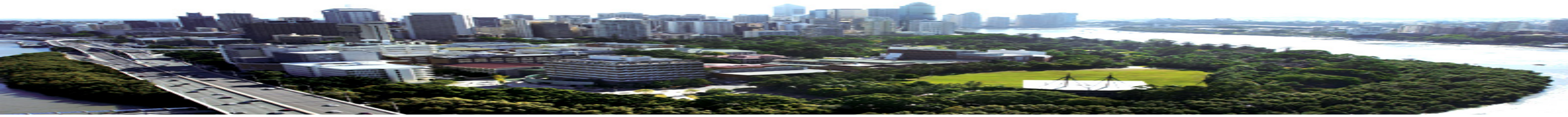
First a formula for signed product is developed in terms of unsigned product



16.16 Fixed-Point Multiplication

Thus the 64-bit product of two 32-bit unsigned operands would be:

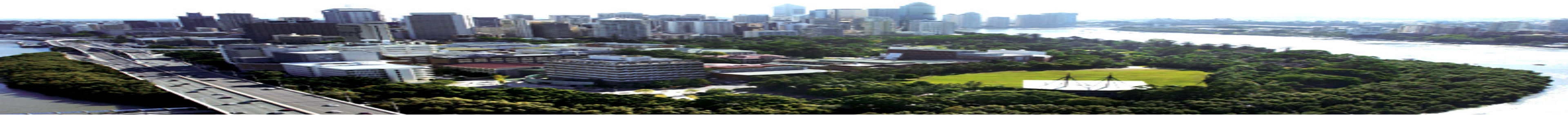
$$
\begin{aligned}
A_u B_u &= (2^{31}A_{31} + A_{30..0})(2^{31}B_{31} + B_{30..0}) \\
&= 2^{62}A_{31}B_{31} + 2^{31(}A_{31}\,B_{30..0} + B_{31}\,A_{30..0}) \\
&\qquad + A_{30..0}\,B_{30..0}
\end{aligned}
$$



16.16 Fixed-Point Multiplication

Now consider a 32-bit signed number in 2s complement form:

$$\begin{aligned} A_s &= -2^{31}A_{31} + 2^{30}A_{30} + \dots + 2^{0}A_{0} \\ &= -2^{31}A_{31} + (2^{30}A_{30} + \dots + 2^{0}A_{0}) \\ &= -2^{31}A_{31} + A_{30..0} \end{aligned}$$



16.16 Fixed-Point Multiplication

Thus the 64-bit product of two 32-bit signed operands would be:

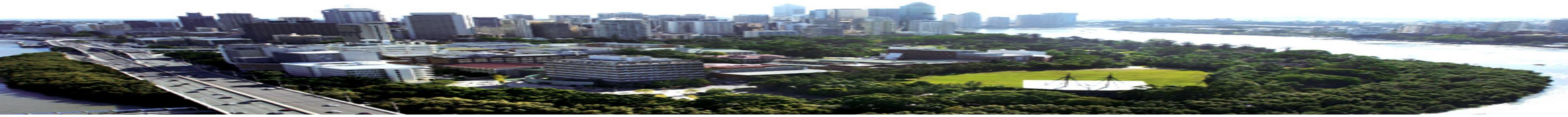
$$A_sB_s = (-2^{31}A_{31} + A_{30..0})(-2^{31}B_{31} + B_{30..0})$$

$$= 2^{62}A_{31}B_{31} - 2^{31(}A_{31}\,B_{30..0} + B_{31}\,A_{30..0})$$

$$+ A_{30..0}\,B_{30..0}$$

$$= A_uB_u - 2\,(2^{31}A_{31}\,B_{30..0} + 2^{31}B_{31}\,A_{30..0})$$

$$= A_uB_u - 2^{32}A_{31}\,B_{30..0} - 2^{32}B_{31}\,A_{30..0}$$



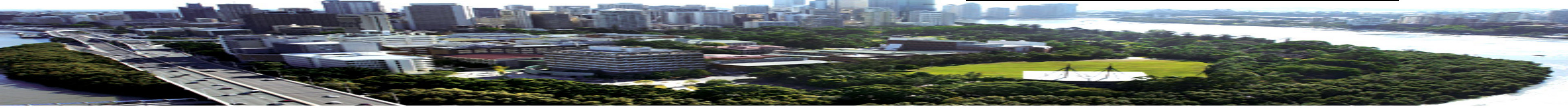
16.16 Fixed-Point Multiplication

What does this result mean?

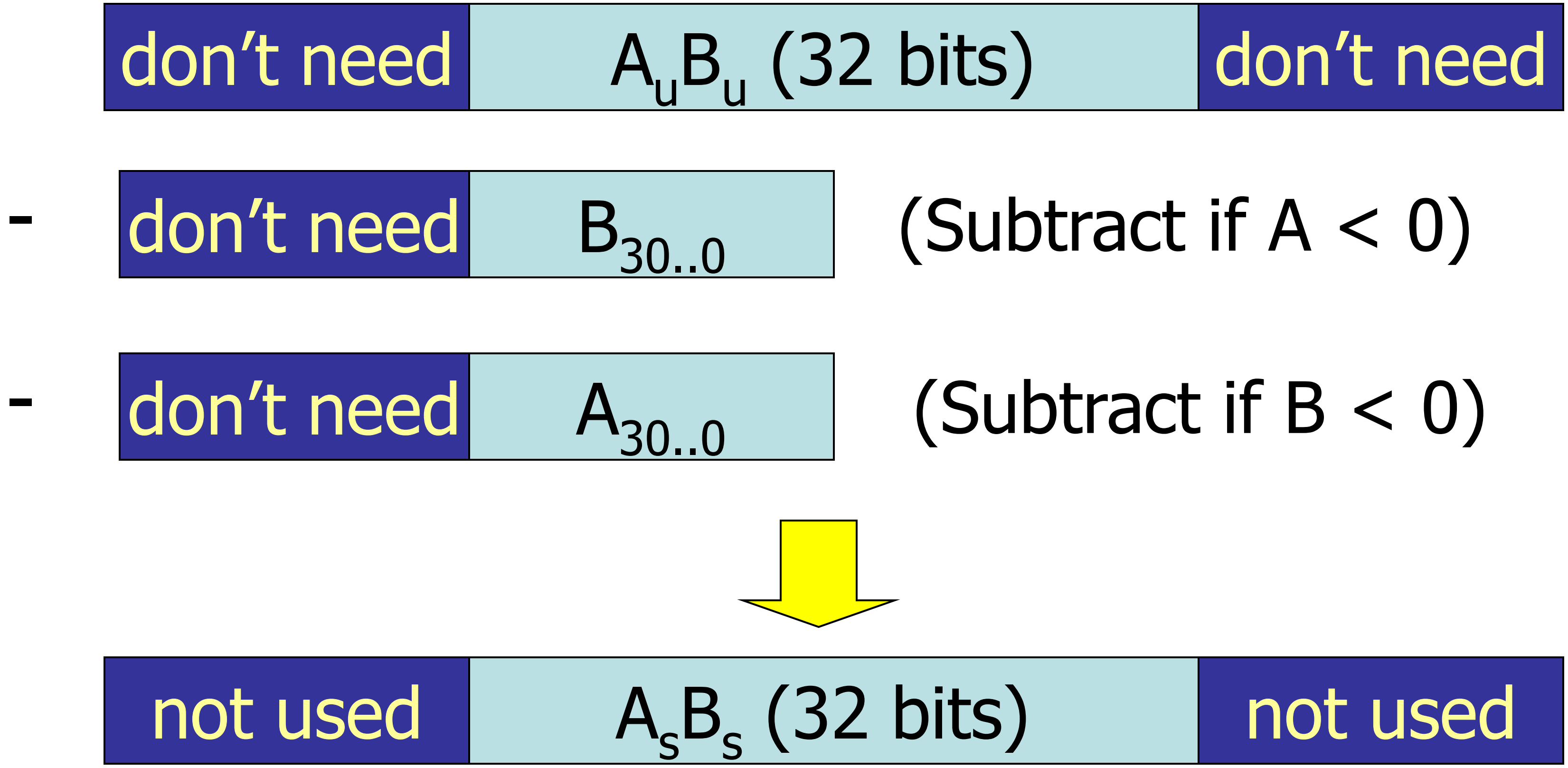
$$A_sB_s = A_uB_u - 2^{32}A_{31}\,B_{30..0} - 2^{32}B_{31}\,A_{30..0}$$

If A is negative, subtract $B_{30..0}$ from the most-significant half of A_uB_u

If B is negative, subtract $A_{30..0}$ from the most-significant half of A_uB_u



16.16 Fixed-Point Multiplication

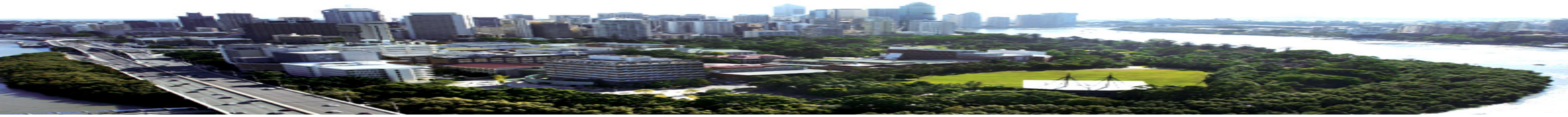


don't need | A_uB_u (32 bits) | don't need

- don't need | $B_{30..0}$ (Subtract if A < 0)

- don't need | $A_{30..0}$ (Subtract if B < 0)

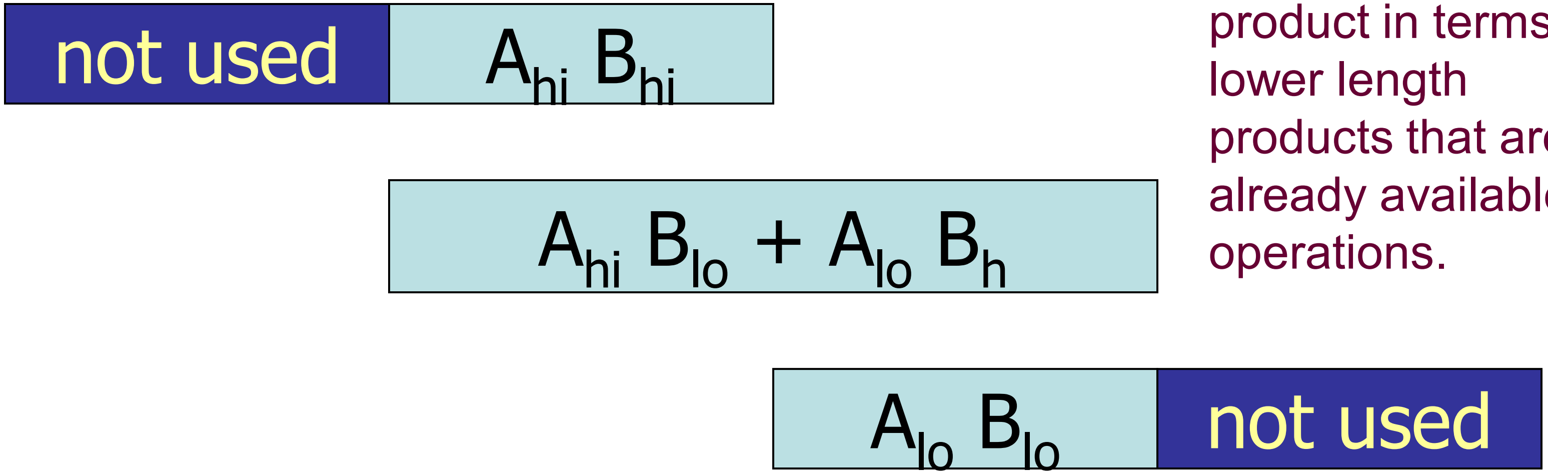
not used | A_sB_s (32 bits) | not used



16.16 Fixed-Point Multiplication

$$A_uB_u = (2^{16}A_{hi} + A_{lo})(\ 2^{16}B_{hi} + B_{lo})$$

$$= 2^{32}A_{hi}\ B_{hi} + 2^{16}(A_{hi}\ B_{lo} + A_{lo}\ B_h) + A_{lo}\ B_{lo}$$

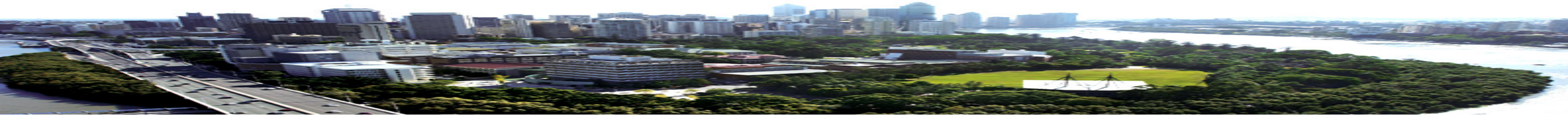


not used | $A_{hi}\ B_{hi}$

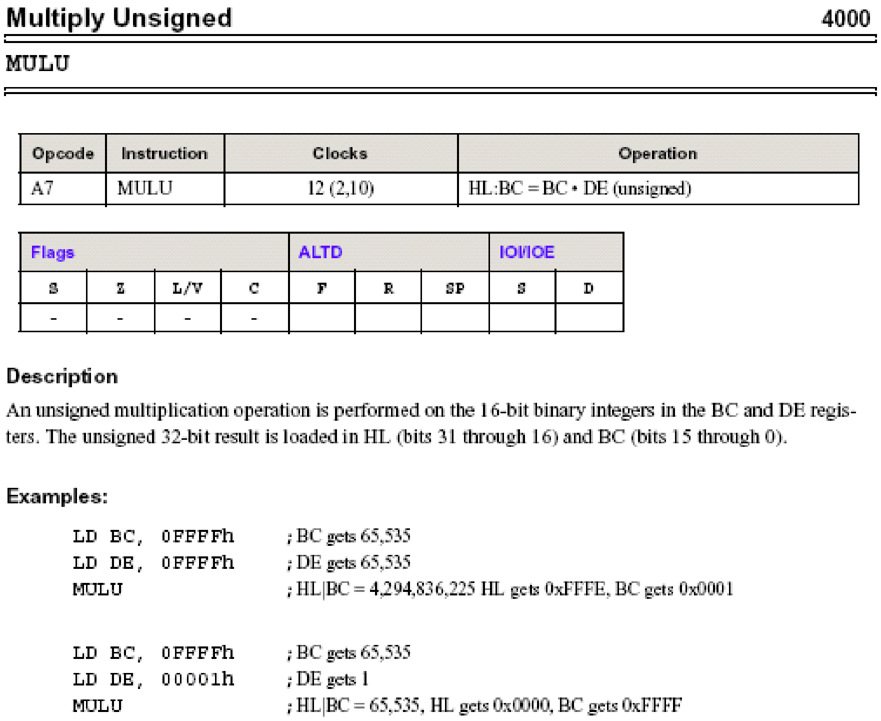
$A_{hi}\ B_{lo} + A_{lo}\ B_h$

$A_{lo}\ B_{lo}$ | not used

Then get unsigned product in terms of lower length products that are already available as operations.



MULU



Multiply Unsigned 4000

MULU

Opcode	Instruction	Clocks	Operation
A7	MULU	12 (2,10)	HL:BC = BC • DE (unsigned)

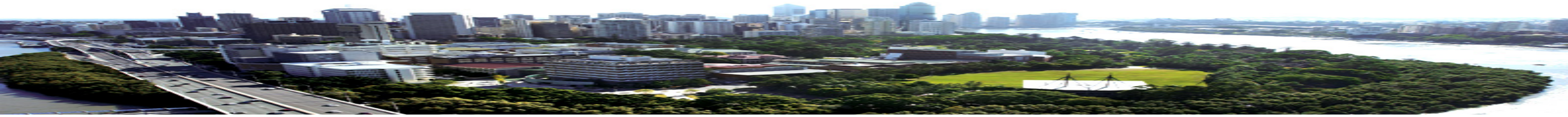
Flags				ALTD			IOI/IOE	
S	Z	L/V	C	F	R	SP	S	D
-	-	-	-					

Description

An unsigned multiplication operation is performed on the 16-bit binary integers in the BC and DE registers. The unsigned 32-bit result is loaded in HL (bits 31 through 16) and BC (bits 15 through 0).

Examples:

```
LD BC, 0FFFFh       ; BC gets 65,535
LD DE, 0FFFFh       ; DE gets 65,535
MULU                ; HL|BC = 4,294,836,225 HL gets 0xFFFE, BC gets 0x0001

LD BC, 0FFFFh       ; BC gets 65,535
LD DE, 00001h       ; DE gets 1
MULU                ; HL|BC = 65,535, HL gets 0x0000, BC gets 0xFFFF
```

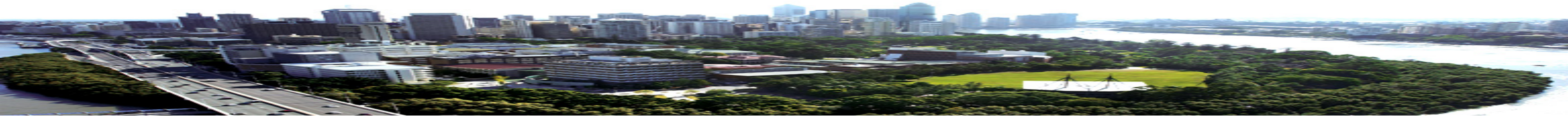


Multiplication of unsigned 16 bit

```
unsigned long prodUnsigned(unsigned int x, unsigned int y);
```

```
#asm
prodUnsigned::
    push hl
    ld hl, (sp + 0x04)
    ex de,hl
    ld hl, (sp + 0x06)
    ex de,hl
    ld bc,hl
    MULU
    ex bc,hl
    ld de,hl
    pop hl
    ret
#endasm
```

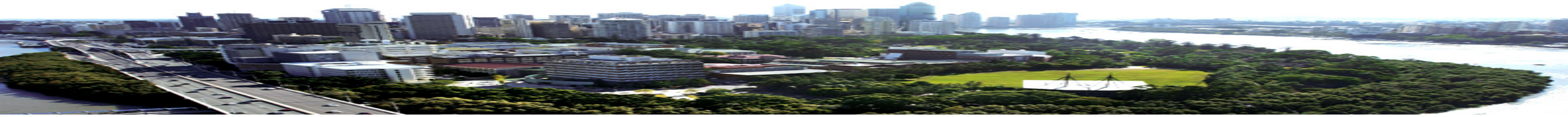
Save value of hl register in the stack
Load parameter x from the stack into hl
Move it to de by exchanging de and hl
Load parameter y from the stack into hl
Again exchange de and hl
Now the two operands are in de and hl
Parameter y is in de and x is in hl
Move x to bc. Operands are in bc and de.
MULU multiplies them
Result is put into HL and BC
Return value is expected in BC and DE
So exchange HL and BC and then
Move HL to DE. Then return.



CRICOS No. 00213J

Driver to test it

```
main()
{

unsigned int a, b;
unsigned long prodab;
a = (unsigned int) 0x0008;
b = (unsigned int) 0x8000;

prodab = (unsigned long) prodUnsigned(a,b);

printf("a = %4x\n",a);
printf("b = %4x\n",b);
printf("product = %8lx\n",prodab);

exit(0);
}
```



Conclusion

- Data in binary form may need to be manipulated by an embedded systems programmer
- Not all embedded systems have a floating point coprocessor
- Floating point operations using library functions can be slower
- Comparisons of real numbers are problematic because adjustment of precision can result in loss of information
- Fixed point arithmetic can have advantages such as deterministic and fixed (faster) computation times.