COMPUTER SYSTEMS AND ORGANIZATION

Bitwise Operations

Daniel Graham

University of Virginia | ENGINEERING

REVIEW

PARITY

Suppose you want to want to calculate the even parity of x.

If the number of one's bit is number is odd the parity value is 1, otherwise it is zero

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```

PARITY

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```

0010
⊕ 0 — Partity bit starting value

0 — Result of xor

0010
⊕ 0 — Result shifted one

1

0010
⊕ 1

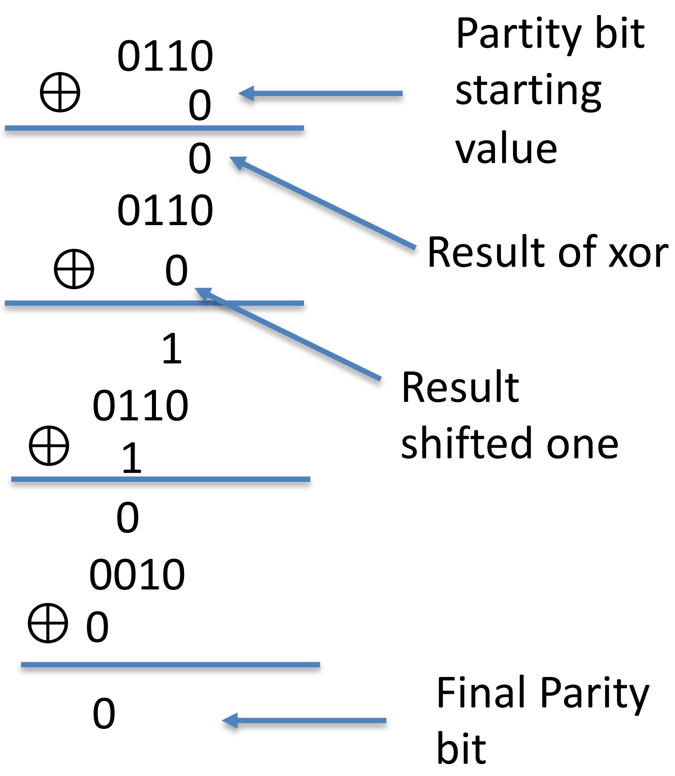
1

0010
⊕ 1

1 — Final Parity bit

UNIVERSITY of VIRGINIA | ENGINEERING

PARITY



0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
    x >>= 1
```

PARALLEL EVALUATION

Observe that xor is both transitive and associative; thus we can re-write

$x0 \oplus x1 \oplus x2 \oplus x3 \oplus x4 \oplus x5 \oplus x6 \oplus x7$

using transitivity as

$x0 \oplus x4 \oplus x1 \oplus x5 \oplus x2 \oplus x6 \oplus x3 \oplus x7$

and using associativity as

$(x0 \oplus x4) \oplus (x1 \oplus x5) \oplus (x2 \oplus x6) \oplus (x3 \oplus x7)$

and then compute the contents of all the parentheses at once via

`x ^ (x>>4)`.

PARALLEL EVALUATION

$x0 \oplus x1 \oplus x2 \oplus x3 \oplus x4 \oplus x5 \oplus x6 \oplus x7$
using transitivity as

$x0 \oplus x4 \oplus x1 \oplus x5 \oplus x2 \oplus x6 \oplus x3 \oplus x7$

and using associativity as
$(x0 \oplus x4) \oplus (x1 \oplus x5) \oplus (x2 \oplus x6) \oplus (x3 \oplus x7)$

and then compute all at once via
x ^ (x>>4).

```
x ^= (x>>16)
x ^= (x>>8)
x ^= (x>>4)
x ^= (x>>2)
x ^= (x>>1)
parity = (x & 1)
```

TODAYS LECTURE

1. Endianness
2. Representing Floating Point Numbers

ENDIANNESS

00000000	42	4D	BA	9B	00	00	00	00	00	00	7A	00	00	00	6C	00
00000010	00	00	5C	00	00	00	90	00	00	00	01	00	18	00	00	00
00000020	00	00	40	9B	00	00	23	2E	00	00	23	2E	00	00	00	00
00000030	00	00	00	00	00	00	42	47	52	73	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	02	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	E2	D5	C9	E4	D8	CE
00000080	EB	DF	D5	E2	D6	CC	DE	D2	C8	E7	DB	D1	E3	D7	CD	E7

What 32 –bit number is stored at location 0x12

0x5C000000 (1543503872_{10}) or 0x0000005C (92_{10})

Answer: it depends

ENDIANNESS

00000010	00	00	5C	00	00	00	90	00
00000020	00	00	40	9B	00	00	23	2E
00000030	00	00	00	00	00	00	42	47
00000040	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00
00000080	EB	DF	D5	E2	D6	CC	DE	D2

Little ENDIAN
0x0000005C (92_{10})

Less significant at Lowest address

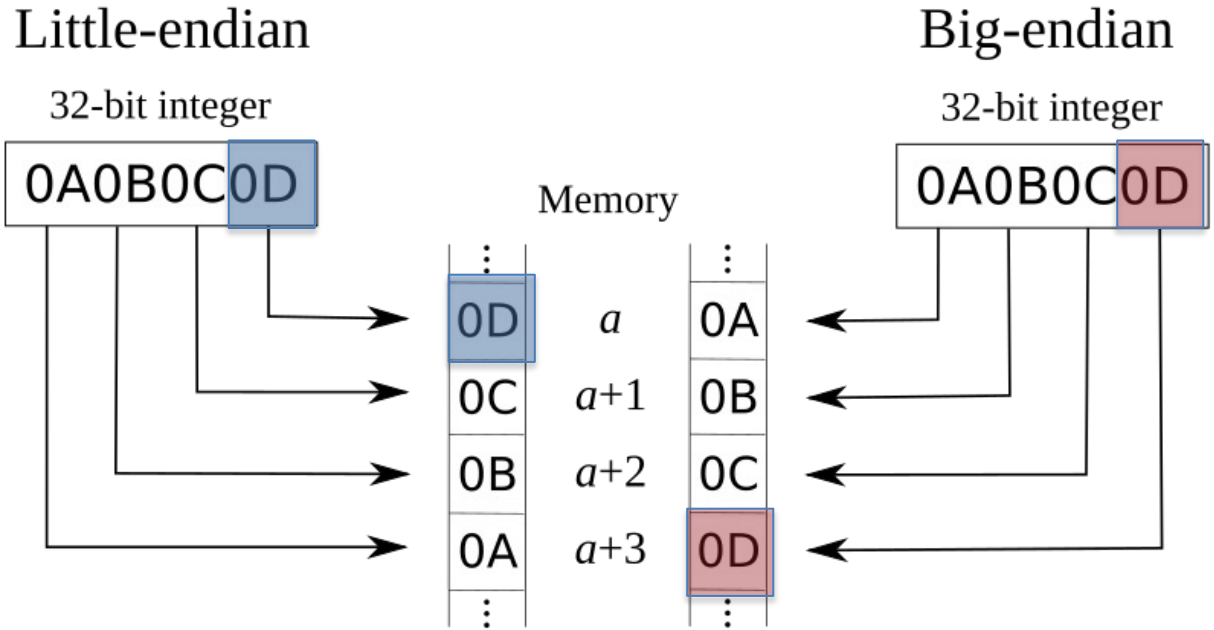
ENDIANNESS

00000010	00	00	5C	00	00	00	90	00
00000020	00	00	40	9B	00	00	23	2E
00000030	00	00	00	00	00	00	42	47
00000040	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00
00000080	EB	DF	D5	E2	D6	CC	DE	D2

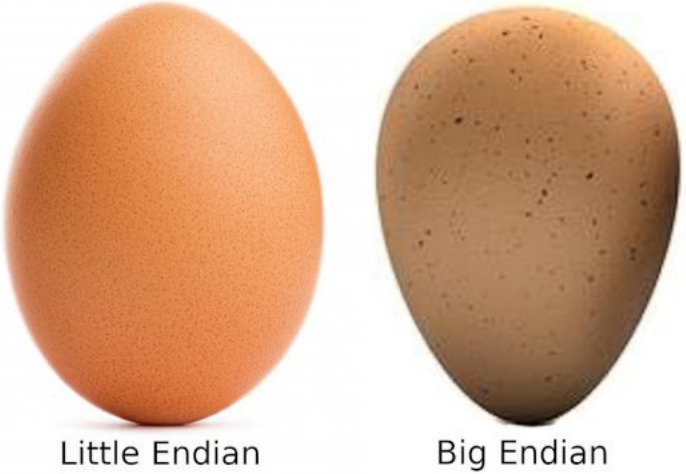
Big ENDIAN
0x5C000000 (1543503872_{10})

Most significant Byte at lowest address

ENDIANNESS


Little-endian
32-bit integer
0A0B0C0D
Memory
0D
0C
0B
0A
a
a+1
a+2
a+3
0A
0B
0C
0D
Big-endian
32-bit integer
0A0B0C0D

WHICH END SHOULD YOU CRACK YOUR EGGS AT



A term borrowed from Gulliver's travels: The Big-Endians, who broke their boiled eggs at the big end, rebelled against the king, who demanded that his subjects break their eggs at the little end.

FLOATING POINT

```
dgg6b@portal07:~$ 
```

FLOATING POINT

- How can we represent decimal values in binary?
- Why do errors like these occur?

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> (0.1 + 0.1 + 0.1) == 0.3
False
>>> x = 0.1 + 0.1 + 0.1
>>> x
0.30000000000000004
```

Floating point rounding error

```
>>> 0.3 + 0.3 + 0.3
>>> 0.8999999999999999
```

FLOATING POINT

Base Ten fraction representation

0.125 has value 1/10 + 2/100 + 5/1000

Base 2 fractions representation

0.001 has value 0/2 + 0/4 + 1/8.

FLOATING POINT

Some fractions can only be approximated when written in a base. For example, 1/3 can only be approximated with written in base 10

0.3 == 1.0/3.0.   (False in python)

0.33333 == 1.0/3.0   (False in python)

0.3333333 == 1.0/3.0 (False in python)

0.3333333333333333 == 1.0/3.0 (True in Python) But not really, because no matter how many digits we write don't it will not be equal to 1/3 (Because this is a repeating fraction, like PI)

FLOATING POINT

Similarly, 0.1 is an infinitely repeating fraction in base 2.

0.0001100110011001100110011001100110011001100110011001100...

So, we have a precision problem. How can we represent floating point numbers?

FLOATING POINT

$7.4 * 10^3$

Floating point is scientific notation in base 2

Notice that all the values are the same the point just floats

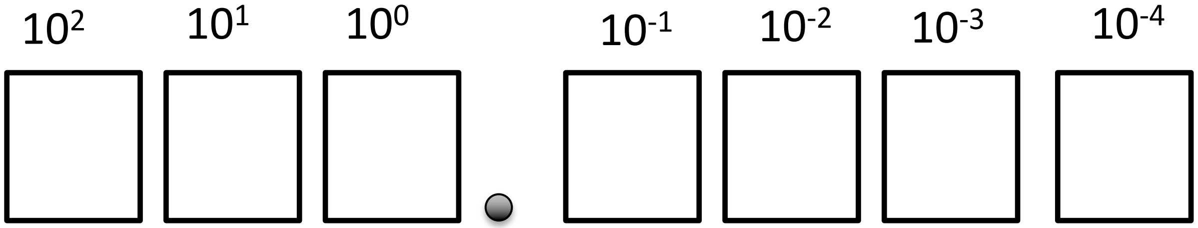
$74.0 * 10^2$

$740.0 * 10^1$

$7400.0 * 10^1$

```
>>> x = 1.7E308
>>> x
1.7e+308
>>> z = x + 0.1e308
>>> z
inf
>>>
```

FLOATING POINT BASE 10

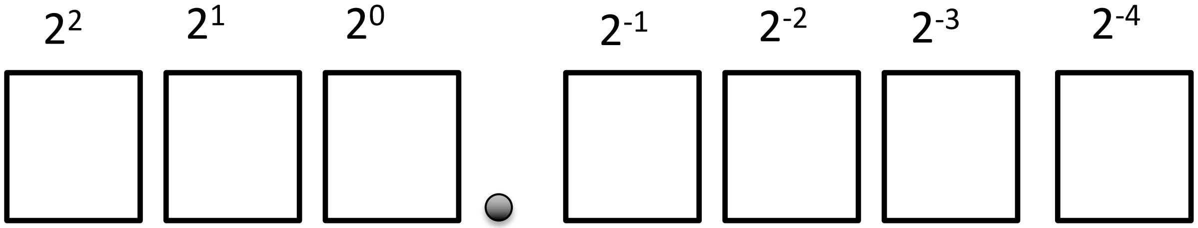


FLOATING POINT BASE 10

10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}	10^{-4}
1	3	2	.	1	7	4	3

$$1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0 + 1 \times 1/10 + 7 \times 1/100 + 4 \times 1/1000 + 3/10000 = 32.1743$$

FLOATING POINT BASE 2



FLOATING POINT BASE 2

2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	.	1	1	0	1

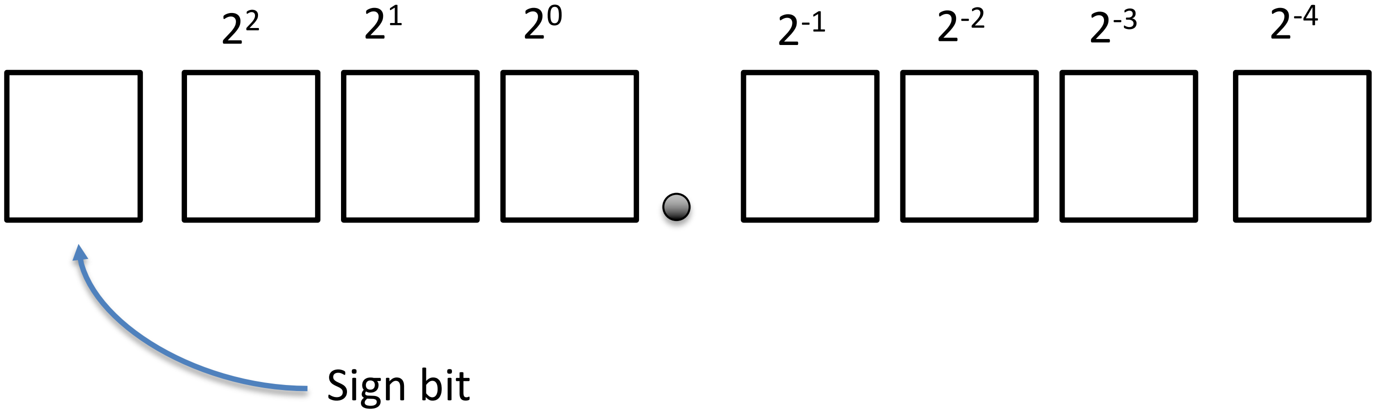
$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

$1 \times 1/2 + 1 \times 1/4 + 0 \times 1/8 + 1 \times 1/16 = 13/16$

$5\ 13/16 = 5.\ 0.8125$

UNIVERSITY of VIRGINIA | ENGINEERING

FLOATING POINT BASE 2 (NEGATIVE NUMBERS)



NOW WE JUST NEED THE EXPONENT

sign	whole	.	Decimal	Exponent

Now we the exponent so we can float the point.

$74.0 * 10^2$
$740.0 * 10^1$
$7400.0 * 10^1$

$-74.0 * 10^2$
$- 740.0 * 10^1$
$- 7400.0 * 10^1$

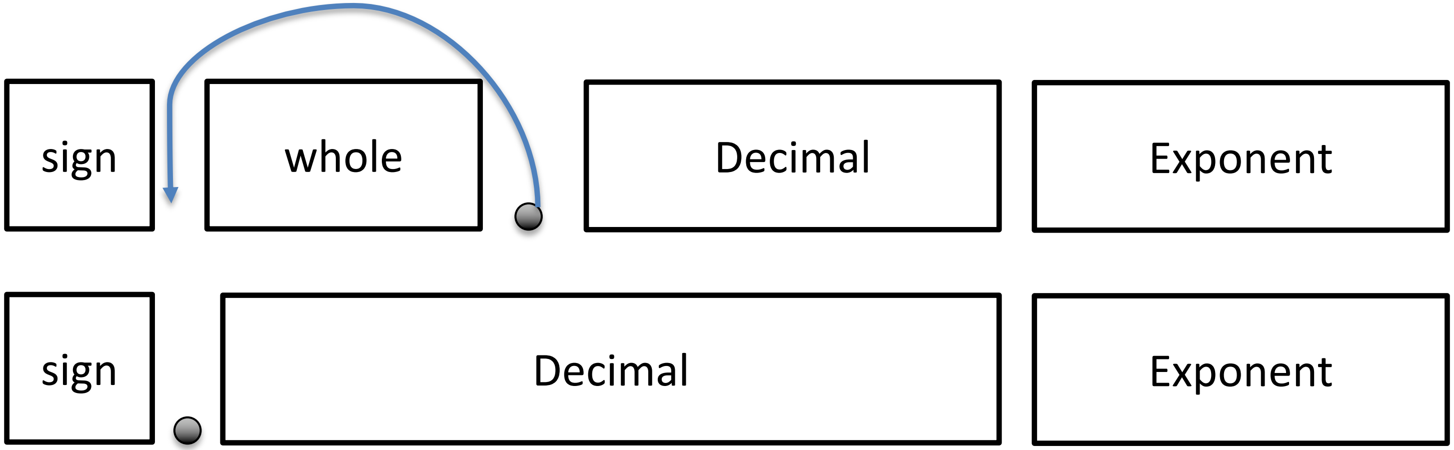
NOW WE JUST NEED THE EXPONENT

sign	whole	.	Decimal	Exponent

$$\text{number} = \text{sign}(\text{whole} + \text{decimal}) \times 2^{\text{exponent}}$$

$$1\ 11.111 \times 2^{E}$$

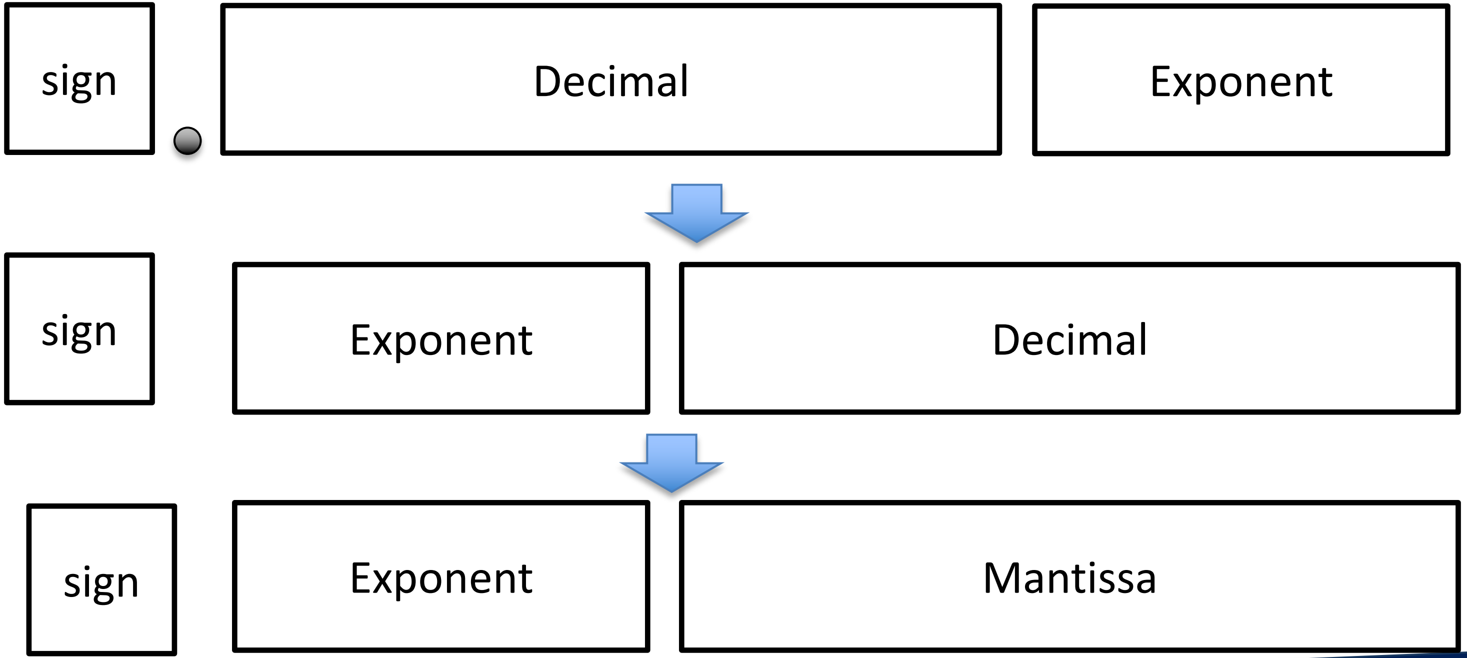
ADDING THE EXPONENT DELETING THE WHOLE NUMBER SECTION



$-74.01 * 10^2$

$-0.7401 * 10^4$

IEEE 754 FLOATING POINT STANDARD



IEEE 754

sign	Exponent	Mantissa

$$\text{number} = \text{sign}(\,1 + \text{Mantissa}) \times 2^{\text{exponent} - \text{bias}}$$

On 32 bit machines bias in normal 127 (Yes this is bias representation we talked about earlier)

IEEE 754

sign	Exponent	Mantissa

$$\text{number} = \text{sign}(1 + \text{Mantissa}) \times 2^{\text{exponent} - \text{bias}}$$

Remember this is a base 2 binary string

$$1.ffff \times 2^{\text{exponent} - \text{bias}}$$

BINARY STRING

$$0.1101 = 1.101 \times 2^{-1}$$

Keep going until you get to your first 1.

$$0.01101 = 1.101 \times 2^{-2}$$

$$0.001101 = 1.101 \times 2^{-3}$$

IEEE 754 FOUR CASES

		Exponent	Mantissa
$\pm\infty$	sign	11…111	000..0000
Not a Number, or NaN	sign	11…111	
$0.ffff \times 2^{1\ -bias}$	sign	00..000	
$1.ffff \times 2^{exponent\ -bias}$	sign		

CONVERSION EXAMPLE

Let's convert 0.8125 to floating-point representation

0.8125 x 2=1.6250 **1**
0.6250 x 2=1.2500 **1**
0.2500 x 2=0.5000 **0**
0.5000 x 2=1.0000 **1**

2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	0	1

1 x 1/2 + 1x 1/4 + 0 x 1/8 + 1 x 1/16 = 13/16

0.1101
= 1.101 x 2-1

CONVERSION EXAMPLE PART 2

$0.1101 = 1.\boxed{101} \times \boxed{2^{-1}}$

Sign: 0

Mantissa: 101

Exponent: -1 + 127 = 126(d) = 1111110(b)

0 01111110 1010000 00000000 00000000

REMEMBER TO ALWAYS BE CAREFUL WITH FLOATING POINT

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Numbers are not always what they seem.

Some guidance

1. Don't test for equality with floating point numbers
2. Worry more about addition and subtraction than Multiplication and Division
3. Numeric Operations don't always return numbers

Ref https://www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming

ENGINEERING