**1)** decimal to binary

decimal to binary

```
n=float(input("Enter the input decimal number = "))
i,f=str(n).split('.')
i,f=int(i),float('0.'+f)
r1=''
while i>0:
   r1=str(i%2)+r1
   i//=2
r2=''
frac=f
for _ in range(5):
   frac*=2; d=int(frac); r2+=str(d); frac-=d
print(f"Decimal {int(n)} converted into base-2 = {r1}")
print(f"Fractional decimal {str(f)[2:]} converted into base-2 = {r2}")
print(f"Hence base-2 equivalent of input decimal number = {r1}.{r2}")
```

**2) decimal to hexadecimal**

```
n=float(input("Enter the input decimal number = "))
i,f=str(n).split('.')
i,f=int(i),float('0.'+f)
digits='0123456789ABCDEF'
r1=''
while i>0:
   r1=digits[i%16]+r1
   i//=16
r2=''
frac=f
for _ in range(5):
   if frac==0: break
   frac*=16
   d=int(frac)
   r2+=digits[d]
   frac-=d
print(f"Decimal {int(n)} converted into base-16 = {r1}")
print(f"Fractional decimal {str(f)[2:]} converted into base-16 = {r2}")
print(f"Hence base-16 equivalent of input decimal number = {r1}.{r2}")
```

**3) Booths multiplication**

```
def to_binary(n, bits):
    """Convert to signed binary string with given bits."""
    if n < 0:
        n = (1 << bits) + n
    return format(n, f'0{bits}b')

M = int(input("Enter the Multiplier (M) = "))
Q = int(input("Enter the Multiplicand (Q) = "))


bits = 5
A = M
B = Q
acc = 0
q = B
q_1 = 0

for _ in range(bits):
    if (q & 1) == 1 and q_1 == 0:
        acc -= A
    elif (q & 1) == 0 and q_1 == 1:
        acc += A


    q_1 = q & 1
    combined = (acc << (bits + 1)) | (q << 1) | q_1
    acc = combined >> (bits + 1)
    q = (combined >> 1) & ((1 << bits) - 1)


product = M * Q
binary_product = to_binary(product, bits * 2)


print("\nOutput:\n")
print(f"Enter the Multiplier (M) = {M}")
print(f"Enter the Multiplicand (Q) = {Q}")
print(f"Binary representation of Multiplicand (Q) = {to_binary(Q, bits)}")
print(f"Binary representation of Multiplier (M) = {to_binary(M, bits)}")
print(f"Result of multiplication in binary = {binary_product}")
```

**4) Restoring division**

```
M=int(input("Enter the Divisor (M)="))
Q=int(input("Enter the Dividend (Q)="))
bin_Q=format(Q,'04b')
bin_M=format(M,'04b')
n=len(bin_Q)
A=0
Q_bin=Q
M_bin=M
for i in range(n):
  A=(A<<1)|((Q_bin>>(n-1))&1)
  Q_bin=((Q_bin<<1)&((1<<n)-1))
  A=A-M_bin
  if A<0:
    A=A+M_bin
    Q_bin=Q_bin&~1
  else:
    Q_bin=Q_bin|1
print(f"Binary representation of Dividend (Q)={bin_Q}")
print(f"Binary representation of Divisor (M)={bin_M}")
print(f"Quotient in Binary={format(Q_bin,'04b')}")
print(f"Remainder in Binary={format(A,'04b')}")
```

**5) Non restoring division**

```
M=int(input("Enter the Divisor(M)="))
Q=int(input("Enter the Dividend(Q)="))
bin_Q=format(Q,'04b')
bin_M=format(M,'04b')
n=len(bin_Q)
A=0
Q_bin=Q
for i in range(n):
  A=(A<<1)|((Q_bin>>(n-1-i))&1)
  if A>=M:
    A=A-M
    Q_bin=(Q_bin<<1)|1
  else:
    Q_bin=(Q_bin<<1)
if A<0:
  A=A+M
print(f"Binary representation of Dividend(Q)={bin_Q}")
print(f"Binary representation of Divisor(M)={bin_M}")
print(f"Quotient in binary={format(Q//M,'04b')}")
print(f"Remainder in binary={format(Q%M,'04b')}")
```

**6) IEEE 754**

```
import struct

def float_to_ieee754(num):
    print(f"Enter the Decimal Number = {num}")
    int_part = int(abs(num))
    frac_part = abs(num) - int_part
    int_bin = bin(int_part).replace("0b", "")
    frac_bin = ""
    while frac_part and len(frac_bin) < 10:
        frac_part *= 2
        if frac_part >= 1:
            frac_bin += "1"
            frac_part -= 1
        else:
            frac_bin += "0"
    print(f"Given number in Binary = {int_bin}.{frac_bin}")
    shift = len(int_bin) - 1
    mantissa = int_bin[1:] + frac_bin
    print(f"Given number in Scientific Notation = 1.{mantissa} * 2^{shift}")
    print(f"Real Exponent = {shift}")
    biased_exp = shift + 127
    exp_bin = format(biased_exp, "08b")
    print("Select the destination floating point format = 32 bit")
    print(f"Biased Exponent = {shift} + 127 = {biased_exp} = {exp_bin}")
    mantissa_23 = (mantissa + "0" * 23)[:23]
    print(f"Actual fractional part = {mantissa}")
    print(f"Mantissa of 23 bits = {mantissa_23}")
    sign_bit = "0" if num >= 0 else "1"
    print(f"Sign bit = {sign_bit}")
    ieee_32bit = sign_bit + exp_bin + mantissa_23
    print(f"32 bit representation of the given number = {ieee_32bit}")
    packed = struct.pack('!f', num)
    hex_str = hex(int.from_bytes(packed, 'big')).upper().replace("0X", "")
    print(f"Hex representation = {hex_str}")

num = float(input("Enter a decimal number: "))
float_to_ieee754(num)
```

**7)** Multilevel Hierarchy (2)

```
def multilevel_memory_hierarchy():
    print("Cost per bit (INR)")
    c1=float(input("C1="))
    c2=float(input("C2="))
    print("\nSize (bits)")
    s1=int(input("S1="))
    s2=int(input("S2="))
    print("\nHit rate/ratio")
    h1=float(input("H1="))
    h2=float(input("H2="))
    print("\nAccess time (microseconds)")
    t1=float(input("T1="))
    t2=float(input("T2="))
    total_cost=(c1*s1)+(c2*s2)
    total_size=s1+s2
    avg_cost_per_bit=total_cost/total_size
    avg_access_time=(h1*t1)+(1-h1)*(h2*t2)
    print("\n-----Results-----")
    print(f"Average Cost per Bit:{avg_cost_per_bit:.6f}INR")
    print(f"Average Access Time:{avg_access_time:.6f}microseconds")
multilevel_memory_hierarchy()
```

**8) Multilevel memory (3)**

```
def multilevel_memory_analysis():
    print("Cost per bit (INR)")
    c1=float(input("C1="))
    c2=float(input("C2="))
    c3=float(input("C3="))
    print("\nSize (bits)")
    s1=int(input("S1="))
    s2=int(input("S2="))
    s3=int(input("S3="))
    print("\nHit rate/ratio")
    h1=float(input("H1="))
    h2=float(input("H2="))
    h3=float(input("H3="))
    print("\nAccess time (microseconds)")
    t1=float(input("T1="))
    t2=float(input("T2="))
    t3=float(input("T3="))
    total_cost=(c1*s1)+(c2*s2)+(c3*s3)
    total_size=s1+s2+s3
    avg_cost_per_bit=total_cost/total_size
    avg_access_time=(h1*t1)+(1-h1)*(h2*t2+(1-h2)*h3*t3)
    print("\n-----Results-----")
```

```
    print(f"Average Cost per Bit:{avg_cost_per_bit:.6f}INR")
    print(f"Average Access Time:{avg_access_time:.6f}microseconds")
multilevel_memory_analysis()
```

**9) Direct mapping**

```
import math
def direct_mapping():
    cache_size_kb = int(input("Enter size of Cache memory (in KB): "))
    main_size_mb = int(input("Enter size of Main memory (in MB): "))
    line_size = int(input("Enter size of each cache line (in Bytes): "))

    cache_size = cache_size_kb * 1024
    main_size = main_size_mb * 1024 * 1024
    address_bits = int(math.log2(main_size))

    cache_banks = 1
    cache_bank_size = cache_size // cache_banks
    cache_lines = cache_bank_size // line_size
    main_blocks = main_size // line_size
    byte_bits = int(math.log2(line_size))

    line_bits = int(math.log2(cache_lines))
    tag_bits = address_bits - (byte_bits + line_bits)
    print(f"\nSize of Cache memory = {cache_size_kb} KB")
    print(f"Size of Main memory = {main_size_mb} MB")
    print(f"Main memory address = {address_bits} bits")
    print(f"Size of each cache line = {line_size} Bytes")
    print("Select cache mapping policy: Direct mapping")
    print(f"Number of cache banks = {cache_banks}")
    print(f"Hence, size of cache bank = {cache_bank_size // 1024} KB")
    print(f"Cache lines per cache bank = {cache_size_kb} KB/ {line_size} Bytes =
{cache_lines // 1024} K = {cache_lines} (Line No-0 to Line No-{cache_lines - 1})")
    print(f"Main memory address of {address_bits} bits is interpreted in 3 fields as
calculated below:")
    print(f"LSB {byte_bits} bits for Byte selection")
    print(f"Number of main memory blocks = {main_size_mb} MB/{line_size} Bytes=
{main_blocks / 1000000:.1f} M = {main_blocks} (Block -0 to Block No
{main_blocks - 1})")
    print(f"Middle {line_bits} bits for Cache line selection")
    print(f"MSB {tag_bits} bits (remaining) for the Tags")
    block_num = int(input("\nInput any Main memory block number for cache
mapping = "))
    cache_line = block_num % cache_lines
    print(f"Block {block_num} is mapped into cache line number = {cache_line}")
if _name_ == "_main_":
    direct_mapping()
```

**10) Two way**

```
import math

def two_way_mapping():
    cache_size_kb = int(input("Enter size of Cache memory (in KB): "))
    main_size_mb = int(input("Enter size of Main memory (in MB): "))
    line_size = int(input("Enter size of each cache line (in Bytes): "))

    cache_size = cache_size_kb * 1024
    main_size = main_size_mb * 1024 * 1024
    address_bits = int(math.log2(main_size))

    cache_banks = 1
    cache_bank_size = cache_size // cache_banks
    cache_lines = cache_bank_size // line_size
    main_blocks = main_size // line_size
    byte_bits = int(math.log2(line_size))

    sets = cache_lines // 2
    set_bits = int(math.log2(sets))
    tag_bits = address_bits - (byte_bits + set_bits)
    print("\nCache mapping policy: Two-Way Set Associative Mapping")
    print(f"Number of sets = {sets} (Set No 0 to {sets - 1})")
    print(f"Lines per set = 2")
    print(f"Number of main memory blocks = {main_blocks} (Block No 0 to
{main_blocks - 1})")
    block_num = int(input("\nEnter any Main memory block number for cache
mapping: "))
    set_number = block_num % sets
    print(f"\nBlock {block_num} is mapped into set number = {set_number}")
    print("Block can be placed in either of the two lines in this set.")
    print("\nMain memory address of", address_bits, "bits is interpreted in 3 fields:")
    print(f"LSB {byte_bits} bits for Byte selection")
    print(f"Middle {set_bits} bits for Set selection")
    print(f"MSB {tag_bits} bits for Tags")

if _name_ == "_main_":
    two_way_mapping()
```

**11) Fully**

```
import math

def fully_associative_mapping():
    cache_size_kb = int(input("Enter size of Cache memory (in KB): "))
    main_size_mb = int(input("Enter size of Main memory (in MB): "))
    line_size = int(input("Enter size of each cache line (in Bytes): "))

    cache_size = cache_size_kb * 1024
    main_size = main_size_mb * 1024 * 1024
    address_bits = int(math.log2(main_size))

    cache_banks = 1
    cache_bank_size = cache_size // cache_banks
    cache_lines = cache_bank_size // line_size
    main_blocks = main_size // line_size
    byte_bits = int(math.log2(line_size))

    tag_bits = address_bits - byte_bits
    print("\nCache mapping policy: Fully Associative Mapping")
    print(f"Number of cache banks = {cache_banks}")
    print(f"Hence, size of cache bank = {cache_bank_size // 1024} KB")
    print(f"Cache lines per cache bank = {cache_lines} (Line No 0 to {cache_lines - 1})")
    print(f"Number of main memory blocks = {main_blocks} (Block No 0 to {main_blocks - 1})")
    block_num = int(input("\nEnter any Main memory block number for cache mapping: "))
    cache_line = int(input(f"Enter cache line number (0 to {cache_lines - 1}): "))
    print(f"\nBlock {block_num} is mapped into cache line number = {cache_line}")
    print("\nMain memory address of", address_bits, "bits is interpreted in 3 fields:")
    print(f"LSB {byte_bits} bits for Byte selection")
    print(f"Middle 0 bits for Cache line selection (fully associative)")
    print(f"MSB {tag_bits} bits for Tags")

if _name_ == "_main_":
    fully_associative_mapping()
```