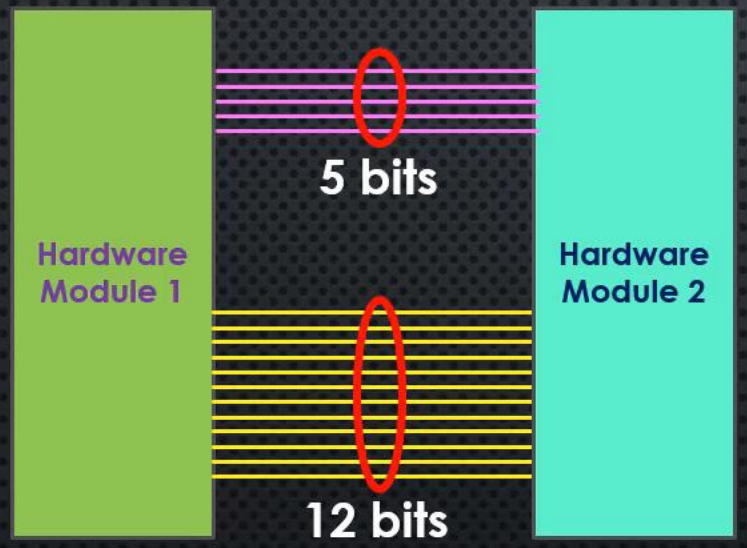
BIT PRECISION-DECLARATION

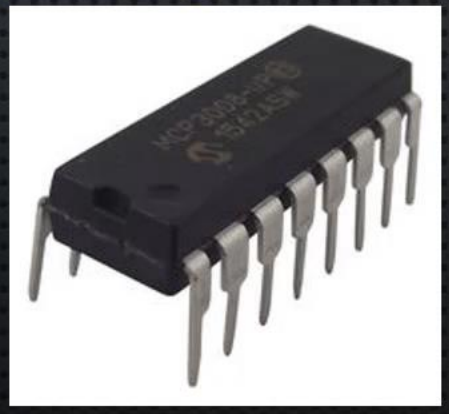
Lecture – 5-Data type-Part2

BIT PRECISION

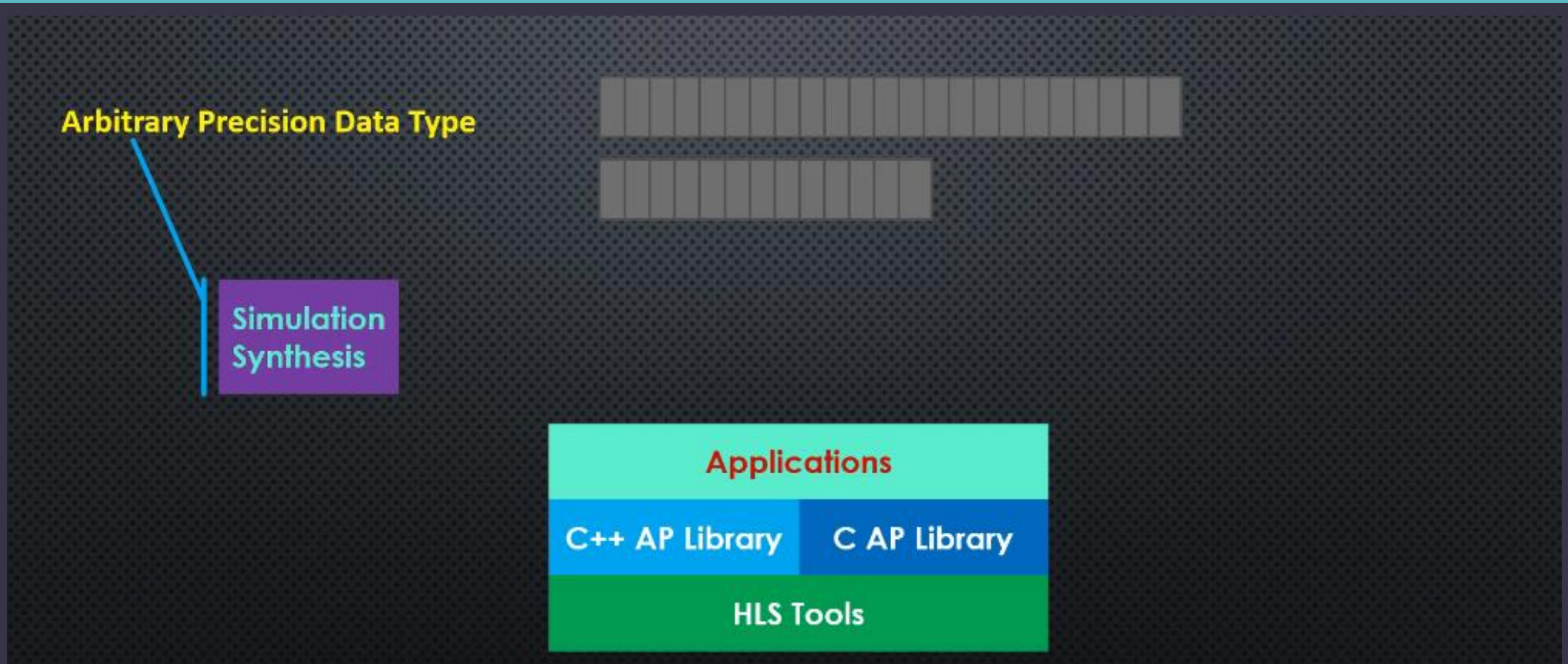


Instruction Coding

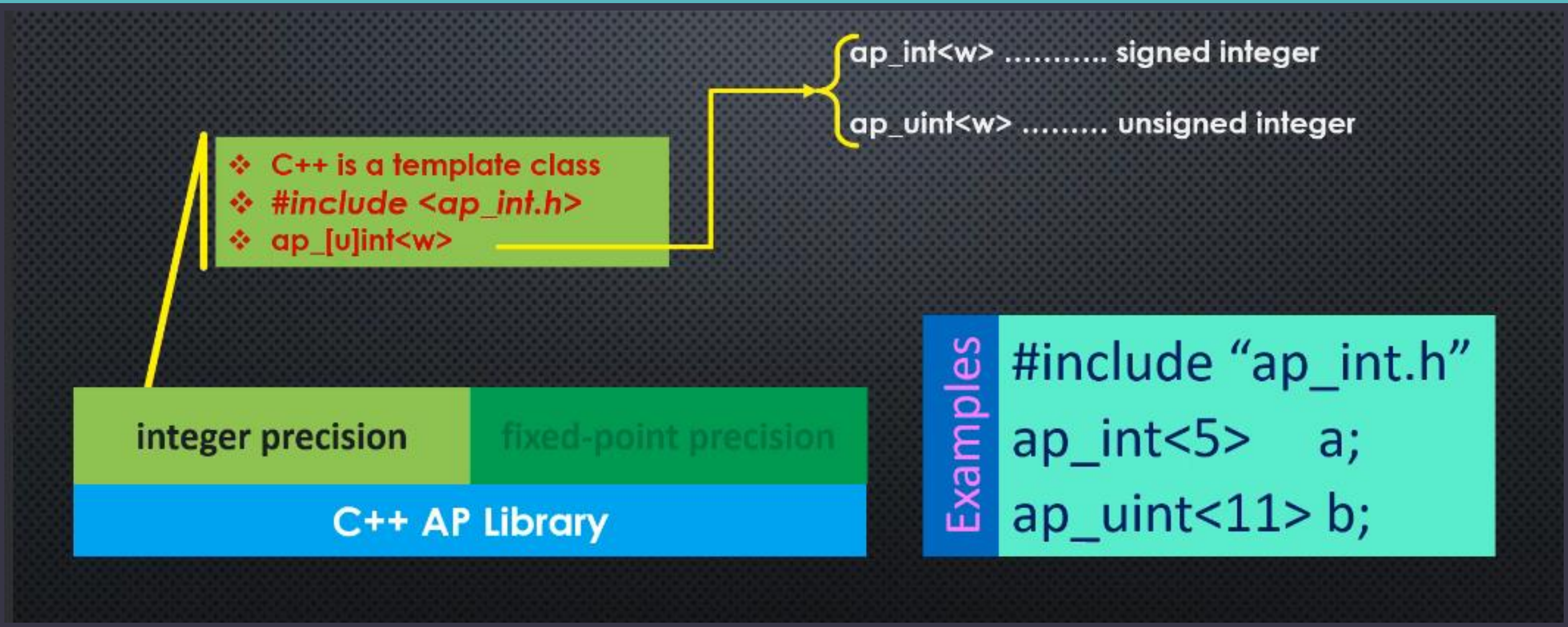
Analogue to Digital Converter, 10 bit,



BIT PRECISION IN HLS



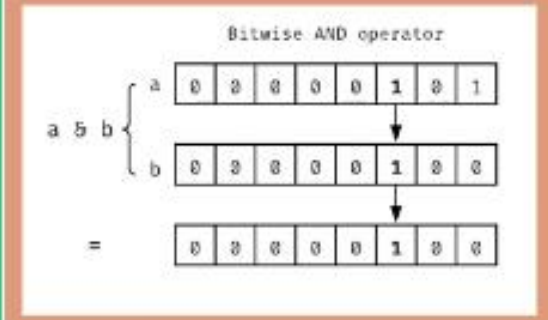
BIT PRECISION IN HLS LIBRARY



AP_[U]INT TEMPLATE CLASS

Arithmetic

Bitwise



Logical

Logical AND (&&):

A	B	A&&B
T	T	T
T	F	F
F	T	F
F	F	F

ARBITRARY BIT-WIDTHS

1 up to 1024 bits

AP_INT_MAX_W

a positive integer value less than or equal to 32768 before inclusion of the *ap_int.h* header file.

Example

```
#define AP_INT_MAX_W 8192

#include "ap_int.h"

ap_int<8192> wide_a;
```

ASSIGNMENT Q

What is the maximum default bit-width of an arbitrary-precision integer data type in Vivado-HLS?

BIT PRECISION-INITIALIZING

Lecture – 4-Data type

INITIALIZATION

```
ap_int<17> a = 0x17e;
```


```
ap_uint<57> b = 0xef42ed12cf72abULL;
```

INITIALIZATION (CONSTRUCTOR)

```
ap_[u]int<w> a = ap_[u]int<w>(string, radix);

ap_[u]int<w> a (string, radix);

ap_int<41> a1 = 04017;  //
ap_int<41> a1 = ap_int<41>("4017", 8);
ap_int<41> a1 = ap_int<41>("0o4017");
ap_int<41> a1 = ap_int<41>("0o4017", 8);

ap_int< 763 > al = ap_int< 763 >("40174017401740174017401740174017", 16);
ap_int< 763 > al = ap_int< 763 >("0x40174017401740174017401740174017");
ap_int< 763 > al = ap_int< 763 >("0x40174017401740174017401740174017", 16);
```

CONSOLE I/O(PRINTING)

```
ap_int<41> a = ap_int<41>("4017", 8);

std::cout << " a = " << a << std::endl;
```

a = 2063

```
ap_int< 763 > b = ap_int< 763 >("0x401740174017401740174017");

std::cout << " b = " << b << std::endl;
```

b = 19835148582765468633513017367

CONSOLE I/O(PRINTING)

```
ap_int< 763 > b = ap_int< 763 >("0x401740174017401740174017");



std::cout << " b = " << std::oct << b << std::endl;
std::cout << " b = " << std::dec << b << std::endl;
std::cout << " b = " << std::hex << b << std::endl;
```

```
b = 0o020013500056400272001350005640027
b = 19835148582765468633513017367
b = 0x401740174017401740174017
```

SUMMARY

- We can use the C/C++ initialisation to assign a value to a declared variable with length less than 64 bits.
- For arbitrary precision data types, more than 64 bits class constructors can be used for initialisation.

ASSIGNMENT Q

Declare a 362-bit width variable of unsigned integer type and initialise that with a value that set all bits to one.

BIT PRECISION-ASSIGNMENT

Lecture – 4-Data type

ASSIGNMENT

```
ap_int<17> a = 0x17e;
```

a

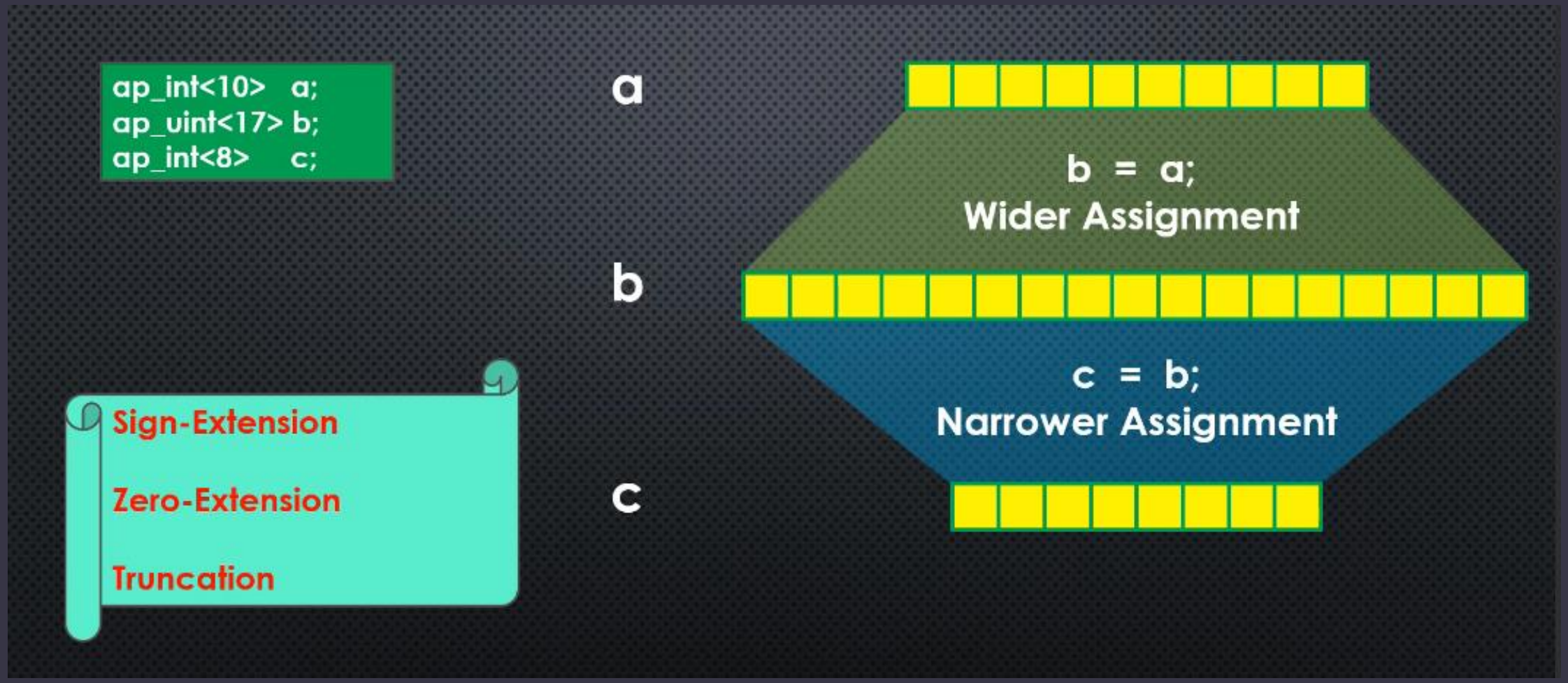
Assignment operators: =, +=, -=, *=, /=, %=, <<= , >>= , &= , ^= , |=

```
ap_int<17> c;

c = a;
```

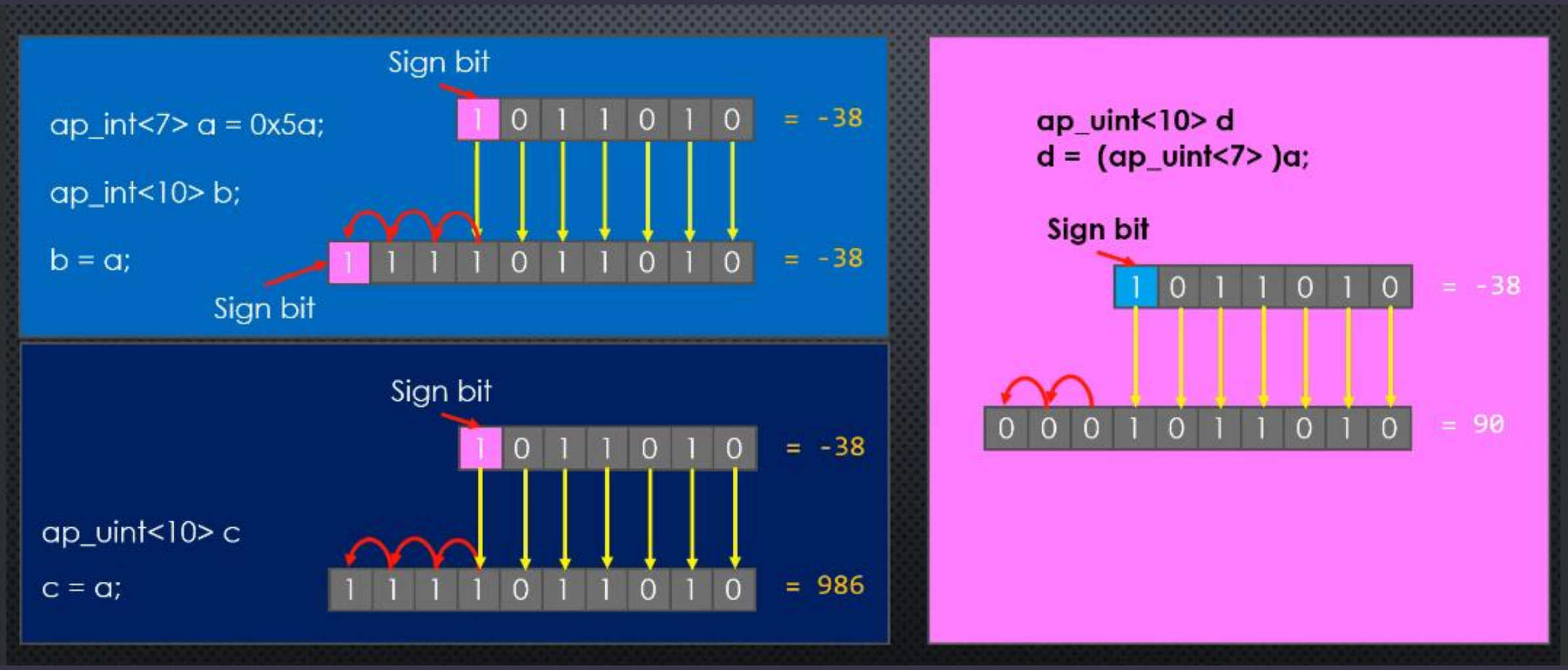
c


```
ap_int<17> c(a);
```

NARROWER OR WIDER ASSIGNMENT

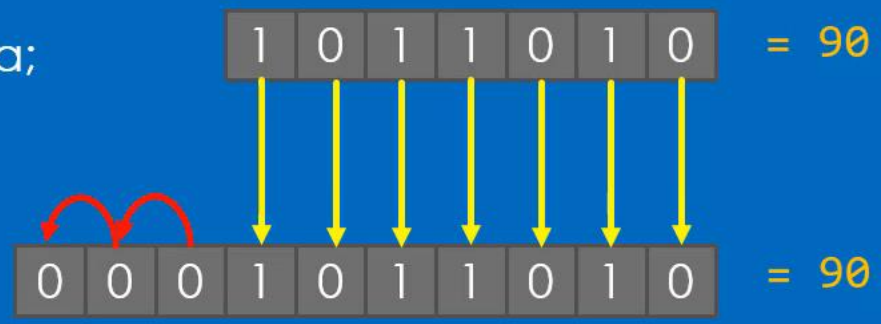


SIGN EXTENSION

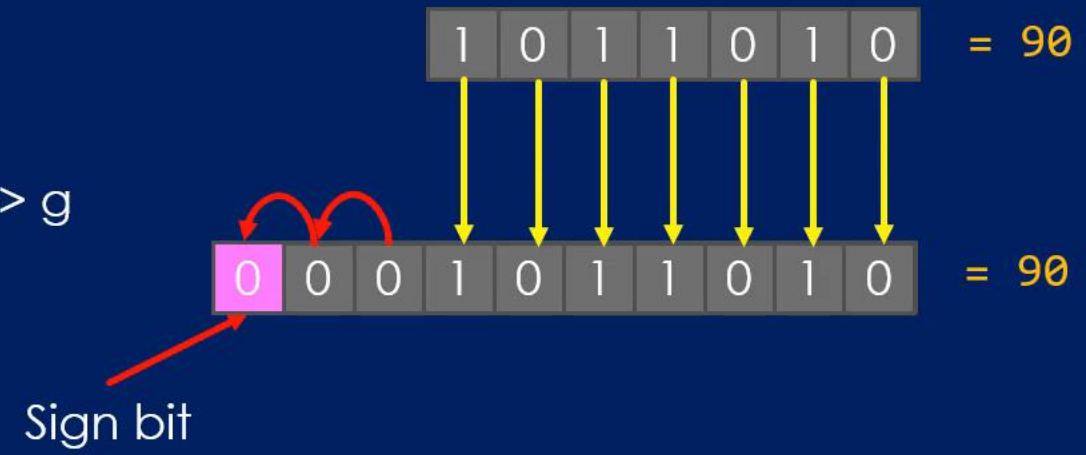


ZERO EXTENSION

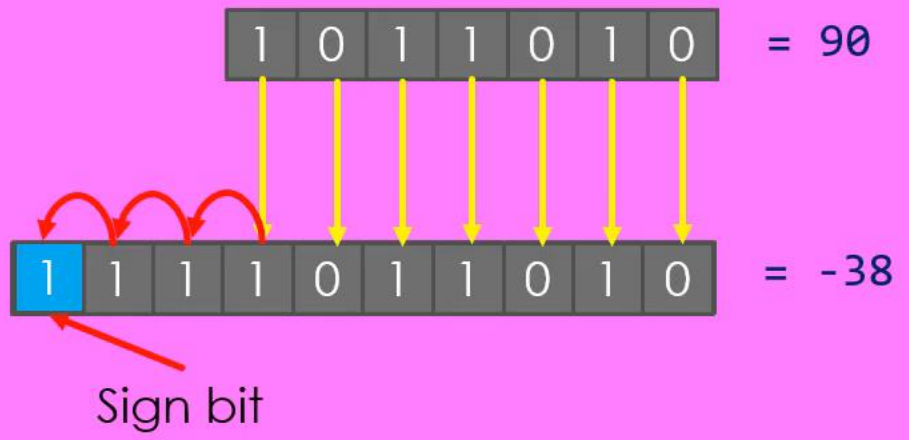
```
ap_uint<7> a = 0x5a;
ap_uint<10> f
f =a;
```




```
ap_int<10> g
g =a;
```

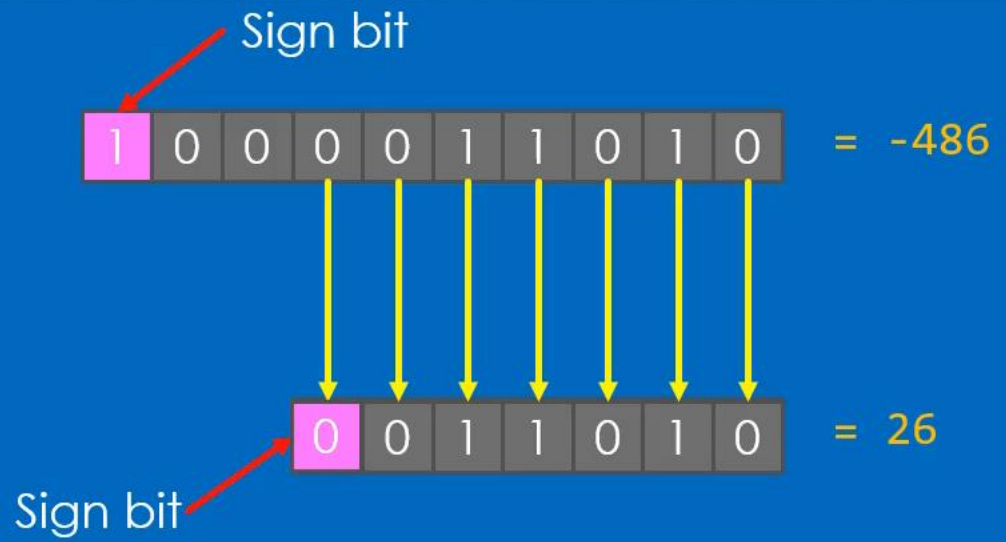



```
ap_int<10> h;
h = (ap_int<7> )a;
```

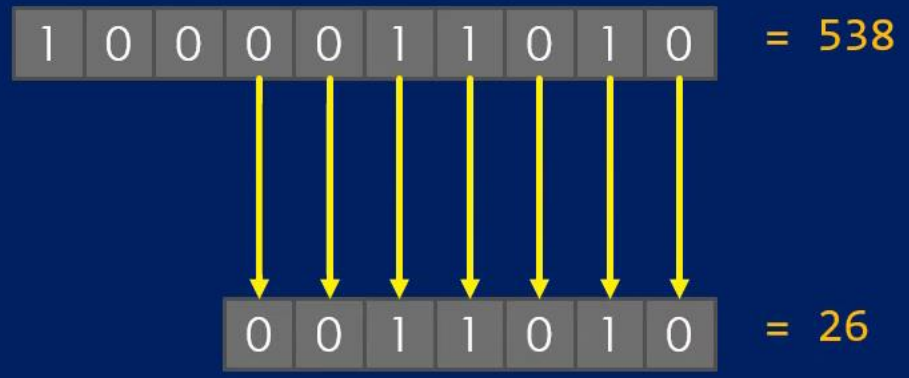


TRUNCATION

```
ap_int<10> I = 0x21a;

ap_int<7>   m ;

m = I;
```




```
ap_uint<10> n = 0x21a;

ap_uint<7>   k ;

k = n;
```



SUMMARY

- There are two different cases if we assign two variables that their bit-widths are not the same: wider assignment and narrower assignment.

- There are three techniques are introduced to handle these situations which are
  - Sign-extension
  - Zero-extension
  - And truncation.

ASSIGNMENT Q

What are the values of all variables in this code after execution?

BIT PRECISION-PRINT

Lecture – 5-Data type-Part2

CONSOLE I/O PRINTING

```
#include <iostream>

ap_uint<102> a("10fedcba9876543210«, 16);

std::cout << "a in decimal = " << a << std::endl;


std::cout << "a in octal = "       << std::oct   << a << std::endl;
std::cout << "a in decimal = "     << std::dec   << a << std::endl;
std::cout << "a in hexadecimal = " << std::hex   << a << std::endl;
```

```
a in decimal = 313512663723845890576
a in octal = 0o041773345651416625031020
a in decimal = 313512663723845890576
a in hexadecimal = 0x010FEDCBA9876543210
```

USING THE STANDARD C LIBRARY

- Convert the value to a C++ std::string using the to_string()
- Convert the result to a null-terminated C character string using the c_str() method

```
printf("a = %s\n", a.to_string().c_str());      //default radix is 2
printf("a = %s\n", a.to_string(2).c_str());
printf("a = %s\n", a.to_string(8).c_str());
printf("a = %s\n", a.to_string(10).c_str());
```

SUMMARY

- Vivado HLS supports printing values that require more than 64-bits to represent.
- The member function, called *to_string,* converts the value of an arbitrary precision integer data types into the corresponding C++ string.

ASSIGNMENT Q

What is the output of this code?

```
...
ap_uint<1024> a("10efffedcba9876543210", 16);
std::cout << "a in octal = "       << std::oct   << a << std::endl;
std::cout << "a in decimal = "     << std::dec   << a << std::endl;
std::cout << "a in hexadecimal = " << std::hex   << a << std::endl;


printf("a = %s\n", a.to_string().c_str());      //default radix is 2
printf("a = %s\n", a.to_string(2).c_str());
printf("a = %s\n", a.to_string(8).c_str());
printf("a = %s\n", a.to_string(10).c_str());
printf("a = %s\n", a.to_string(16).c_str());
...
```

BIT PRECISION-BIT LEVEL

Lecture – 5-Data type-Part2

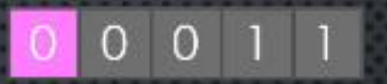
BIT-LEVEL LENGTH

```
length ()
ap_int<41> a = 0x23462af;
int len = a.length();
len = 41;
```

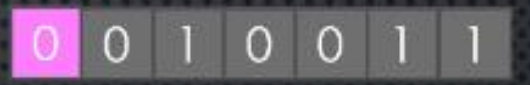
BIT-LEVEL OPERATIONS

Concatenation: the *concat* member function

```
ap_int<5> b = 0x3;
```




```
ap_int<7> c = 0x13;
```




```
ap_int<12> r1 = b.concat(c);
```


```
ap_int<12> r2 = c.concat(b);
```

BIT-LEVEL OPERATIONS

Concatenation: the overloaded , operator

```
ap_int<5> b = 0x3;
```


```
ap_int<7> c = 0x13;
```


```
ap_int<12> r1 = (b, c);
```


```
ap_int<12> r2 = (c, b);
```

BIT SELECTION

The overloaded [] operator

```
ap_uint<7> n = 0x2f;
```


```
int i = 4;
ap_uint<1> one_bit = n[i];
```

one_bit = 0

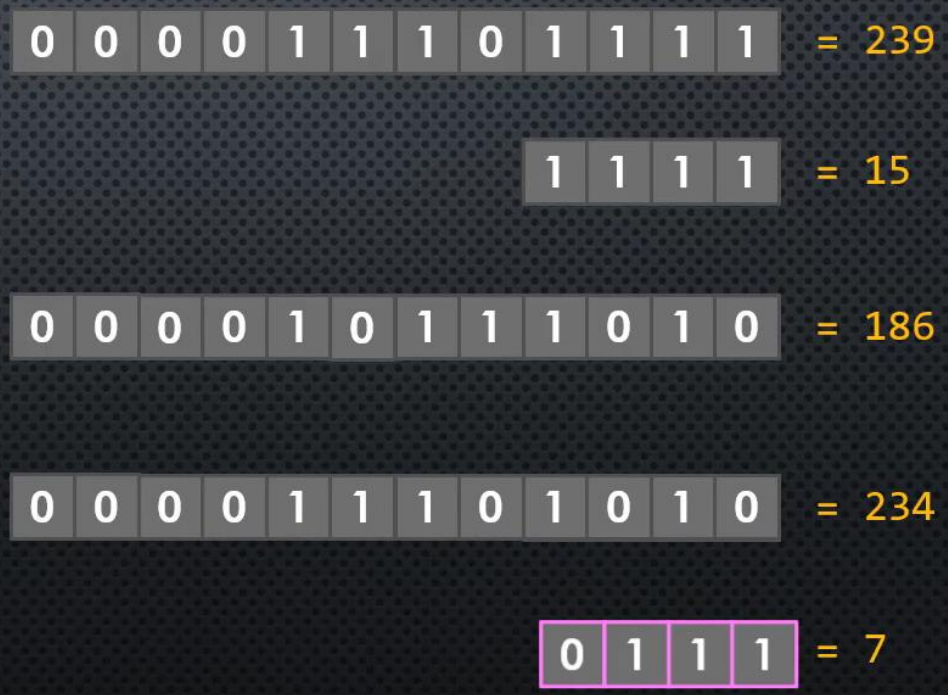
---

n[i] = 1;

RANGE SELECTION

Using the range member function.

```
ap_uint<4> r;
ap_uint<9> p = 0xef;

r = p.range(3, 0);

ap_uint<9> q = 0xba;

p(3,0) = q(3, 0);

r = p.range(4, 7);
```



ASSIGNMENT Q

What is your suggestion for the bit-width of the r variable and what would be its value after execution of this code snippet?

```
ap_uint<9> p = 0xef;
ap_int<5> b = 0x3;

ap_int<?> r = (p(3,0), b)
```

BIT PRECISION-BITWISE LOGICAL OPERATORS

Lecture – 5-Data type-Part2

BITWISE LOGICAL OPERATOR

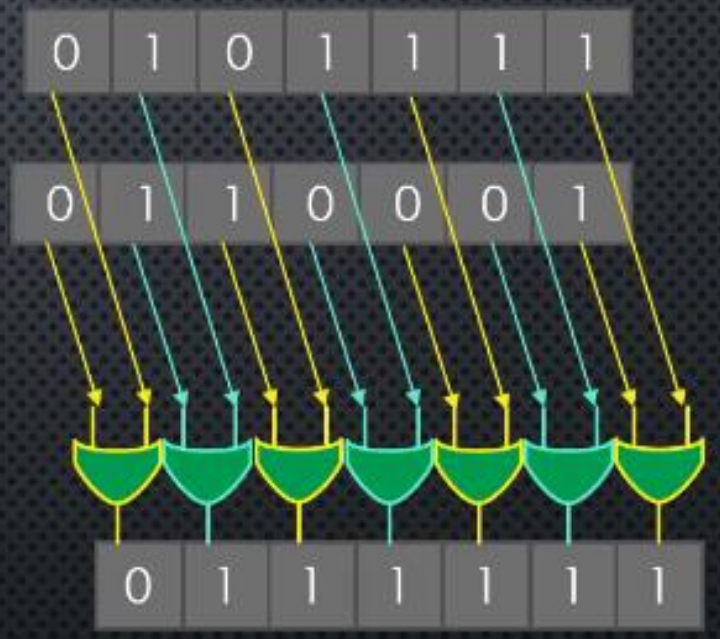
Bitwise OR ..... |

Bitwise AND ..... &

Bitwise XOR ..... ^

Bitwise Inverse ..... ~

```
ap_int<7> a = 0x2f;
ap_int<7> b = 0x31;
ap_int<10> c = a | b;
```



REDUCED LOGICAL FUNCTIONS

AND reduce .....and_reduce ()

OR reduce .....or_reduce ()

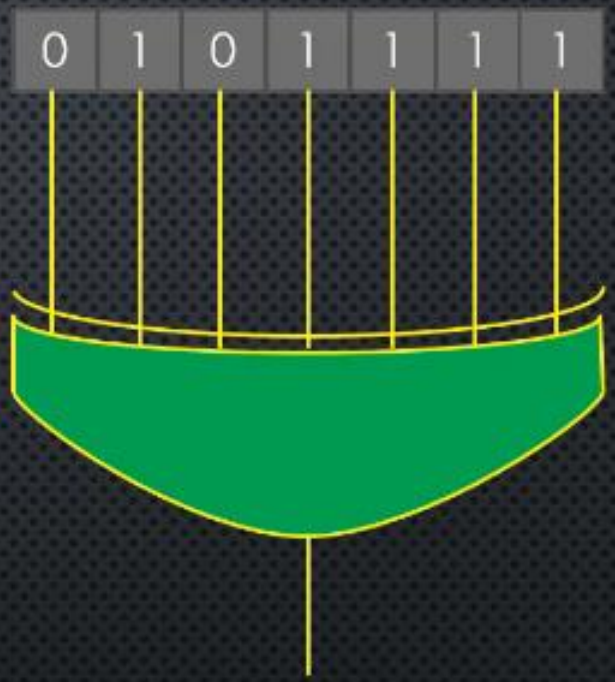
XOR reduce .....xor_reduce ()

NAND reduce .....nand_reduce ()

NOR reduce .....nor_reduce ()

XNOR reduce .....xnor_reduce ()

ap_int<7> a = 0x2f;



BIT REVERSE

Bit Reverse .....reverse()

```
ap_int<7> a = 0x2f;
```


```
a.reverse();
```

TEST/SET/CLEAR

```
ap_int<7> a = 0x2f;
```

Test Bit Value .....test (i) → bool t = a.test(5); returns True

Set Bit Value ..... set (i, v), set_bit(i, v) →
a.set(5, 0);
bool t = a.test(5); returns False

Set Bit (to 1) ..... set (i) →
a.set(5);
bool t = a.test(5); returns True

Clear Bit (to 0) ..... clear (i) →
a.clear(5);
bool t = a.test(5); returns False

SUMMARY

- The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands.
- Reduced functions apply a given operator to all bits of an arbitrary precision value and return a one-bit result.
- The bit-reverse member function reveres the bit position of its object value.

ASSIGNMENT Q

What is the output of this code snippet running by Vivado-HLS in C-Simulation mode?

```
ap_int<7> a = 0x15;
ap_int<7> b = 0x13;

ap_int<7> c = a | b;
ap_int<7> d = a;
d.reverse();
std::cout << " a = " << a.to_string() << std::endl;
std::cout << " b = " << b.to_string() << std::endl;
std::cout << " c = " << c.to_string() << std::endl;
std::cout << " a xor reduced = " << a.xor_reduce() << std::endl;
std::cout << " d = " << d.to_string() << std::endl;
std::cout << " a.test(5) " << a.test(5) << std::endl;
a.set(5, 0);
std::cout << " a(5) " << a.test(5) << std::endl;
a.set(5);
std::cout << " a(5) " << a.test(5) << std::endl;
a.clear(5);
std::cout << " a(5) " << a.test(5) << std::endl;
```

BIT PRECISION-SHIFT/ROTATE

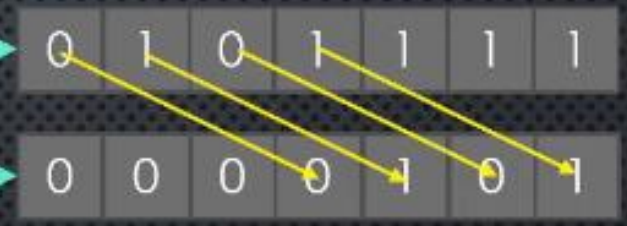
Lecture – 5-Data type-Part2

SHIFT OPERATOR

Overloaded shift operators available for HLS arbitrary precision data types

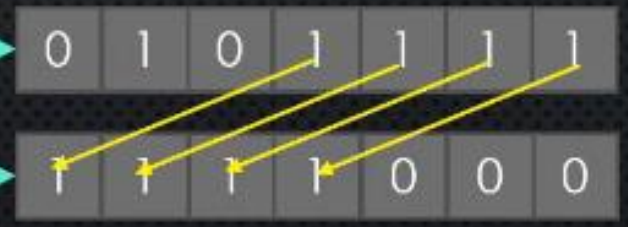
- Right shift (>>)

```
ap_int<7> a = 0x2f;
ap_int<7> b =  a >> 3;
```



- Left shift (<<)

```
ap_int<7> a = 0x2f;
ap_int<7> b =  a << 3;
```



SHIFT OPERATOR

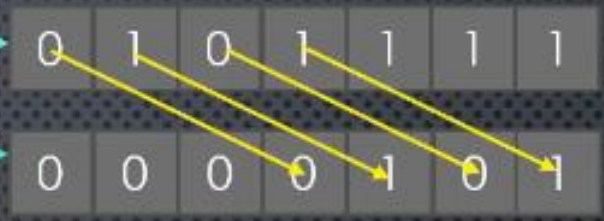
Signed ..... a << -3 ⟶ a >> 3

Unsigned..... a << 3

SIGNED/UNSIGNED SHIFT OPERATOR

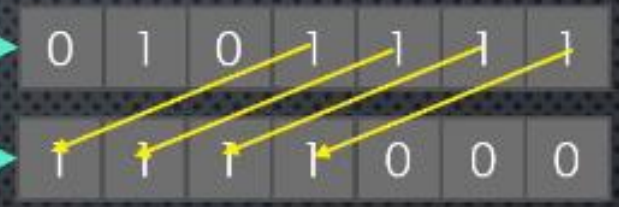
Positive Integer Shift Right

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a >> 3;
```



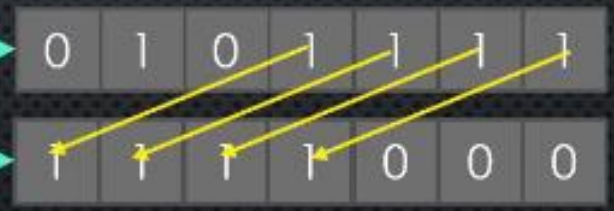
Negative Integer Shift Right

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a >> -3;
```



Positive Integer Shift Left

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a << 3;
```



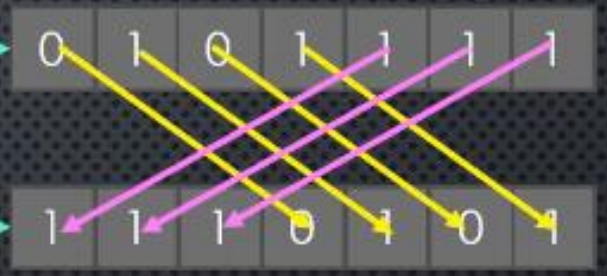
Negative Integer Shift Left

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a << -3;
```

ROTATE FUNCTIONS

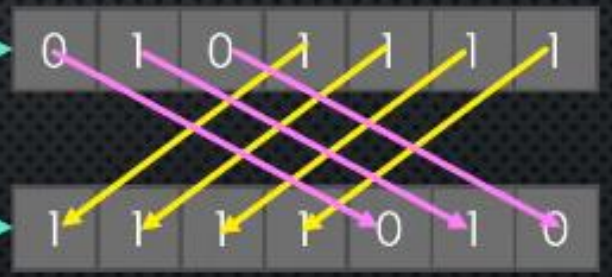
Rotate Right

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a;
b.rrotate(3);
```



Rotate Left

```
ap_int<7> a = 0x2f;
ap_int<7> b;
b = a;
b.lrotate(3);
```



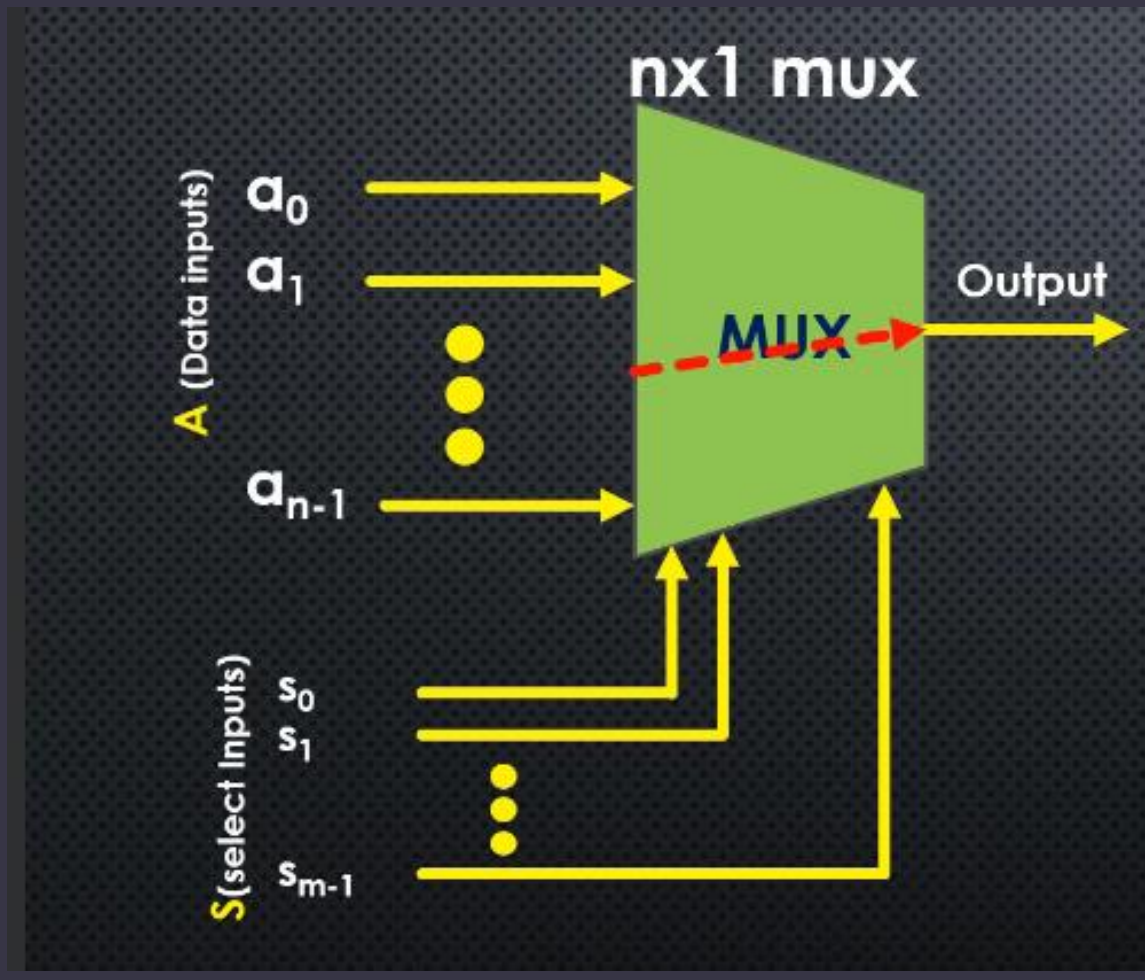
ASSIGNMENT Q

```
ap_uint<13> r1, r2, r3;
ap_uint<7> v1 = 0x41;

r1 = v1 << 6;
r2 = ap_uint<13>(v1) << 6;

ap_int<7> v2 = -63;
r3 = v2 >> 4;
```

CONDITIONAL STATEMENTS INTRODUCTION

Lecture – 5-Conditional Statements

MULTIPLEXER AND CONDITIONAL SELECTIONS



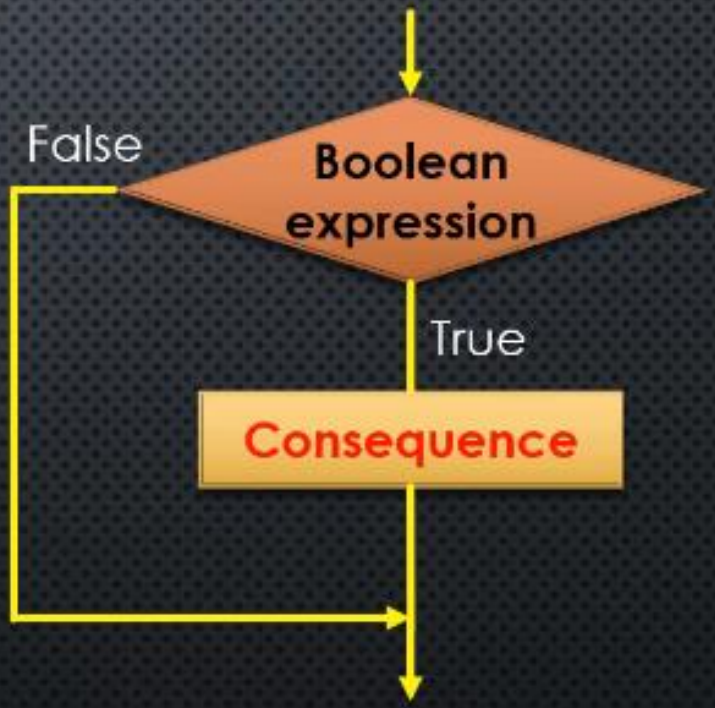

```
if (S==0) {
  output = a0;
} else if (S==1) {
  output = a1;
} else if (S==2) {
  output = a2;
}
...
```

```
switch(S) {
case 0:
  output = a0;
  break;
case 1:
  output = a1;
  break;
...
}
```

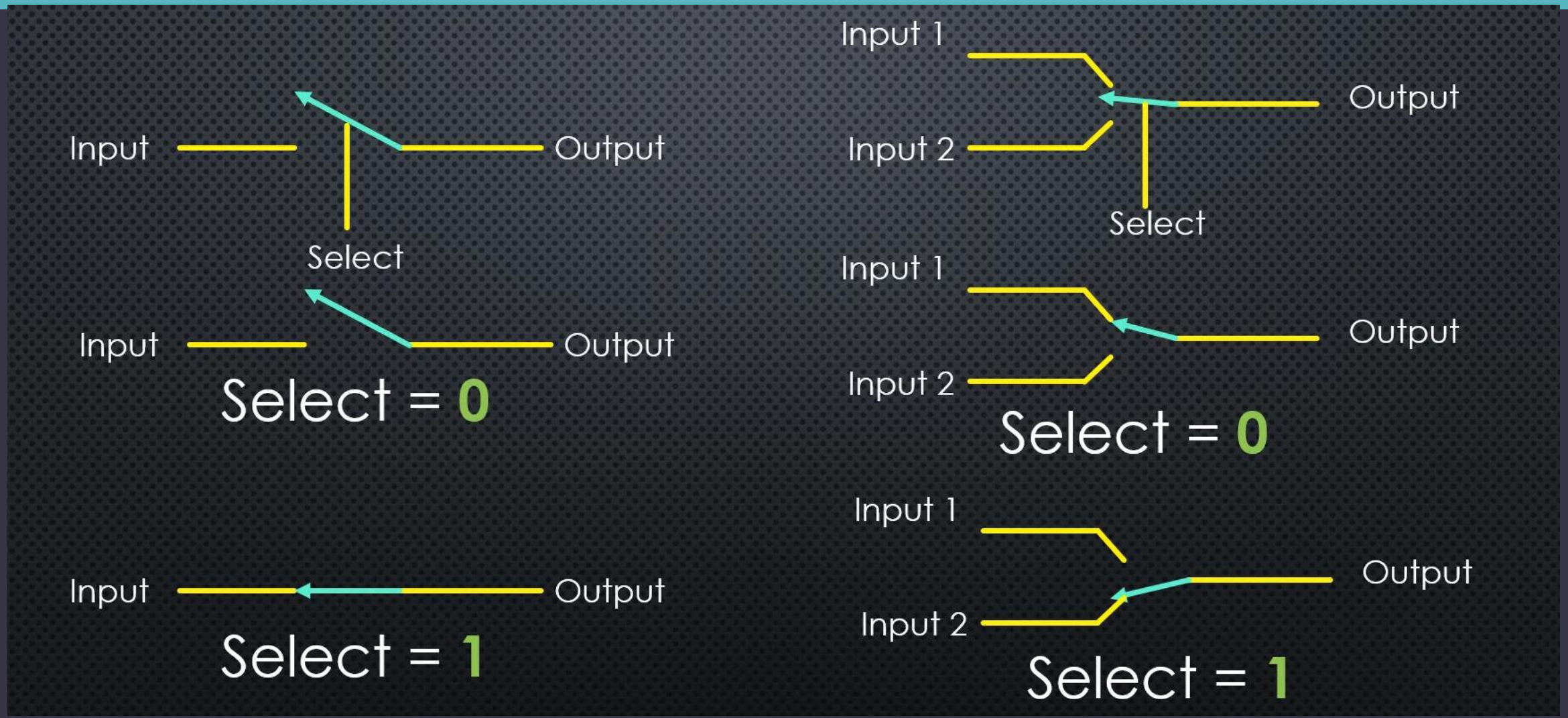
CONDITIONAL STATEMENT IN SOFTWARE

Hypothesis → Conclusions

```
if (Boolean expression) {
  Consequence;
}
```



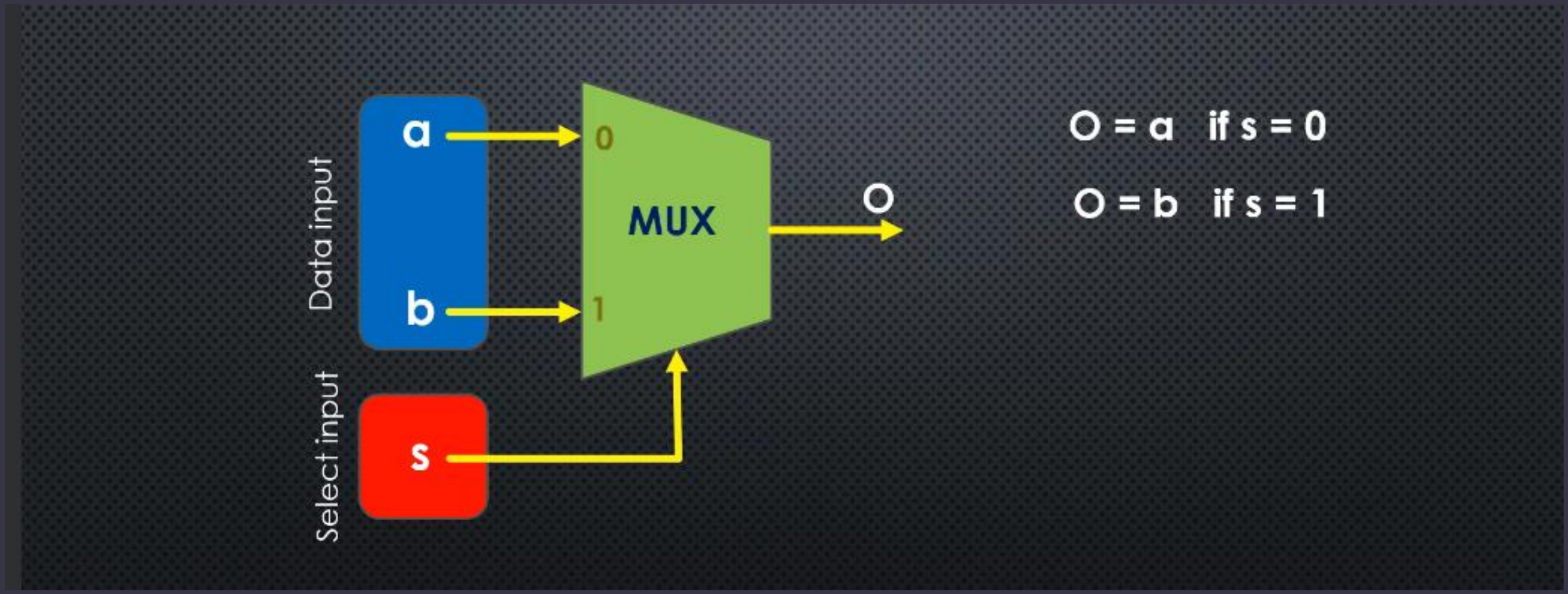
CONDITIONAL STATEMENT IN HARDWARE



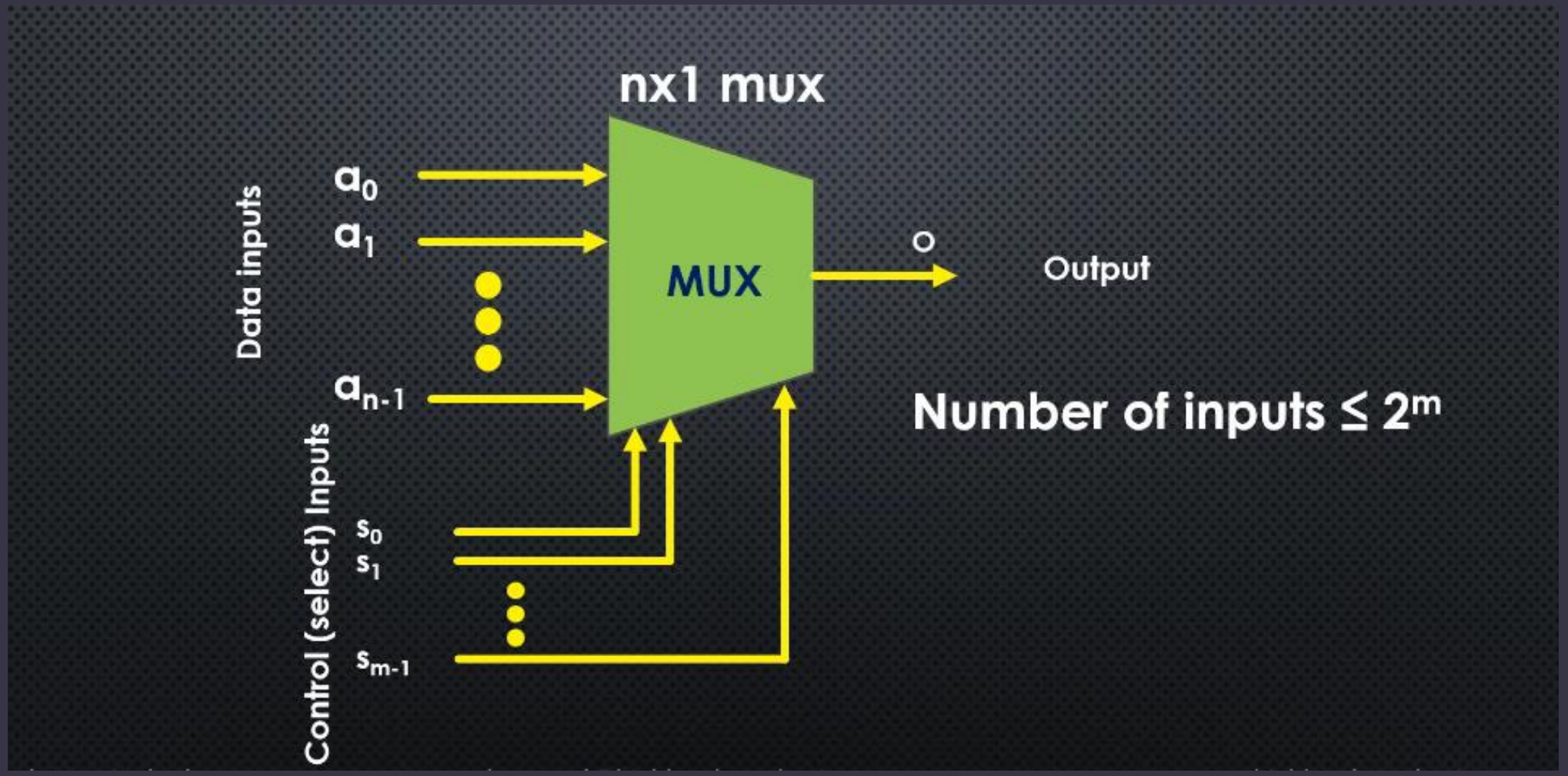
MULTIPLEXER

Lecture – 5-Conditional Statements

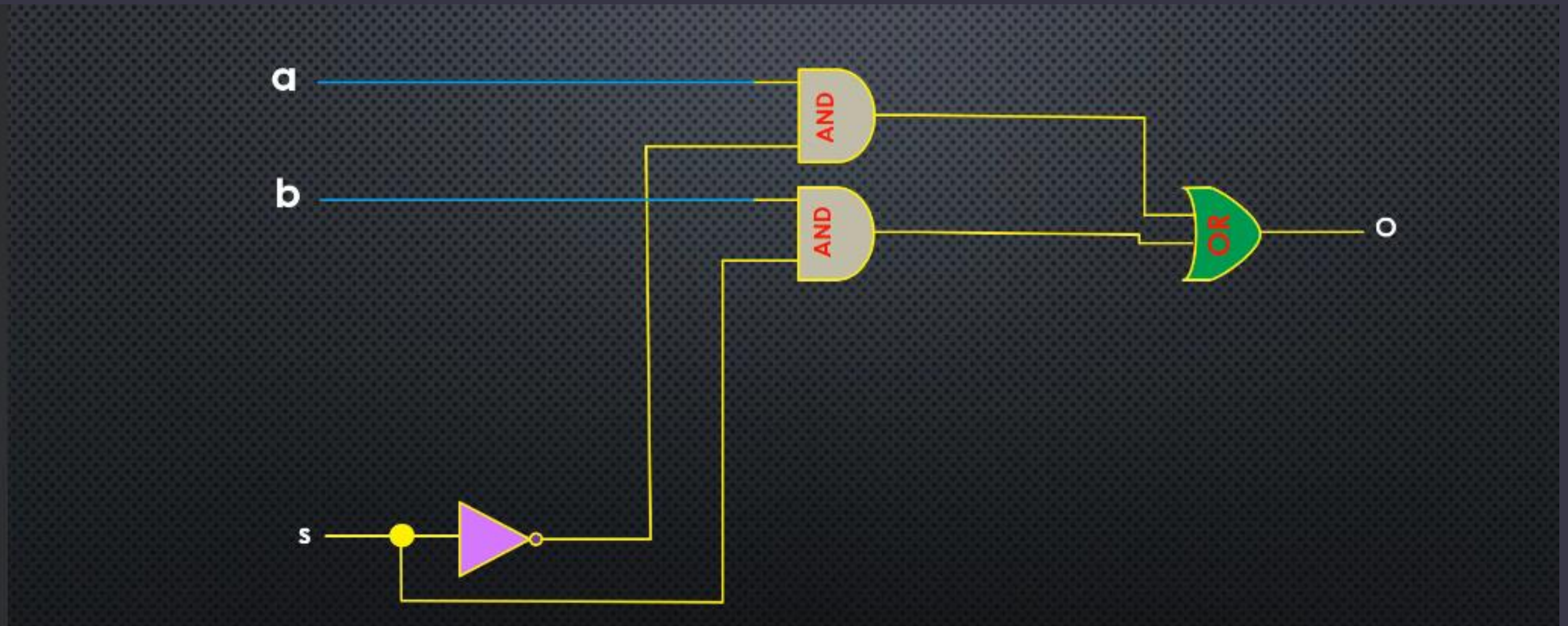
MULTIPLEXER



N×1 MULTIPLEXER

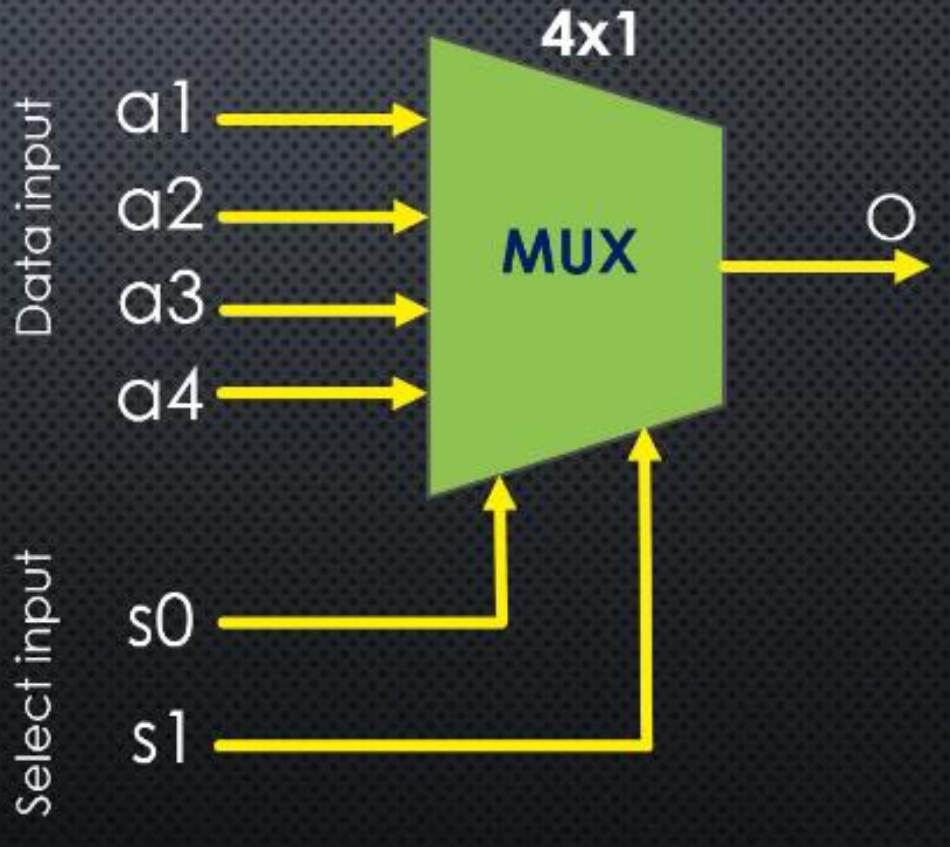


HOW TO BUILD A MULTIPLEXER WITH LOGIC GATES



ASSIGNMENT Q

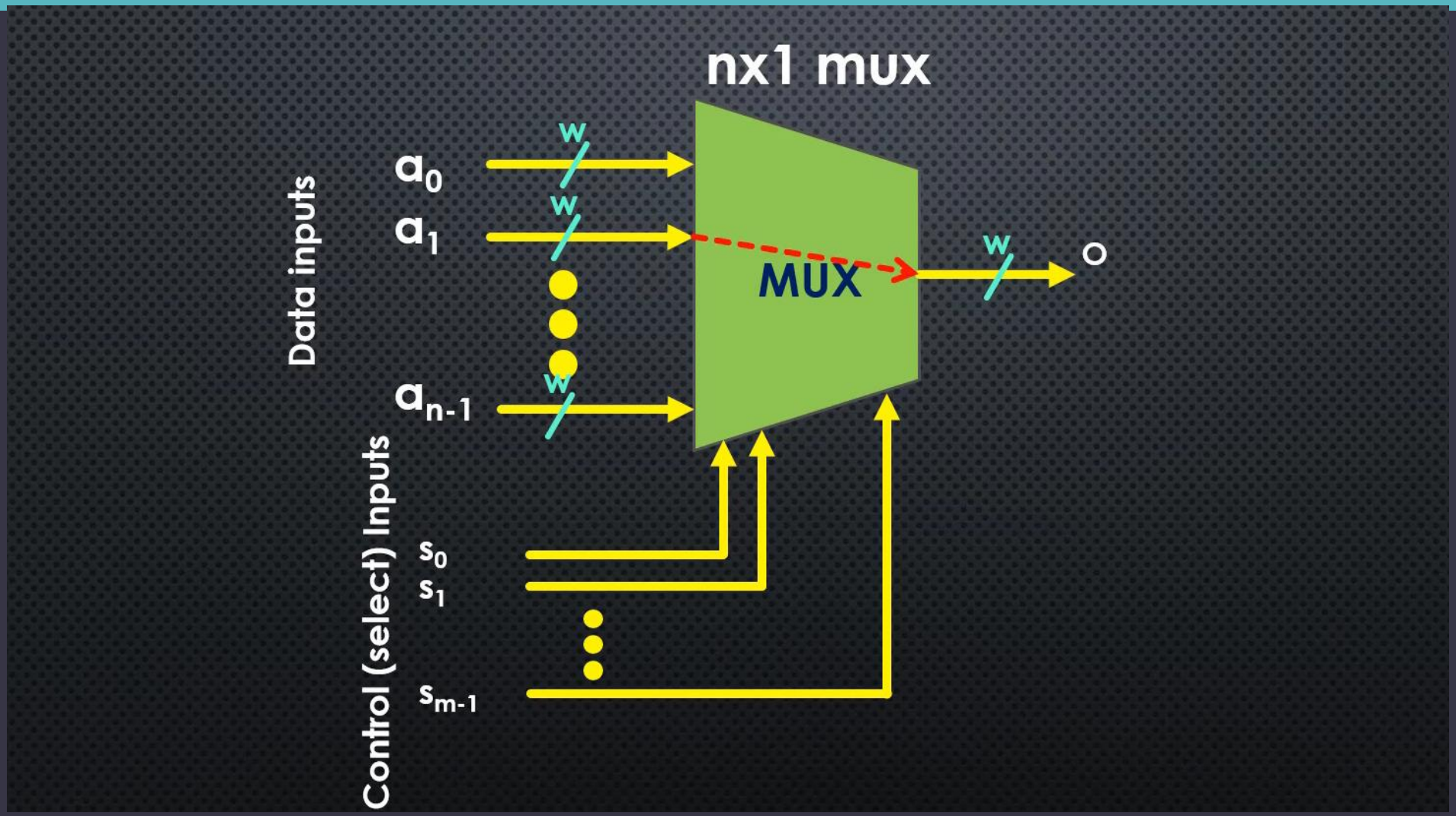
How can a group of basic logic gates implement this 4x1 multiplexer?



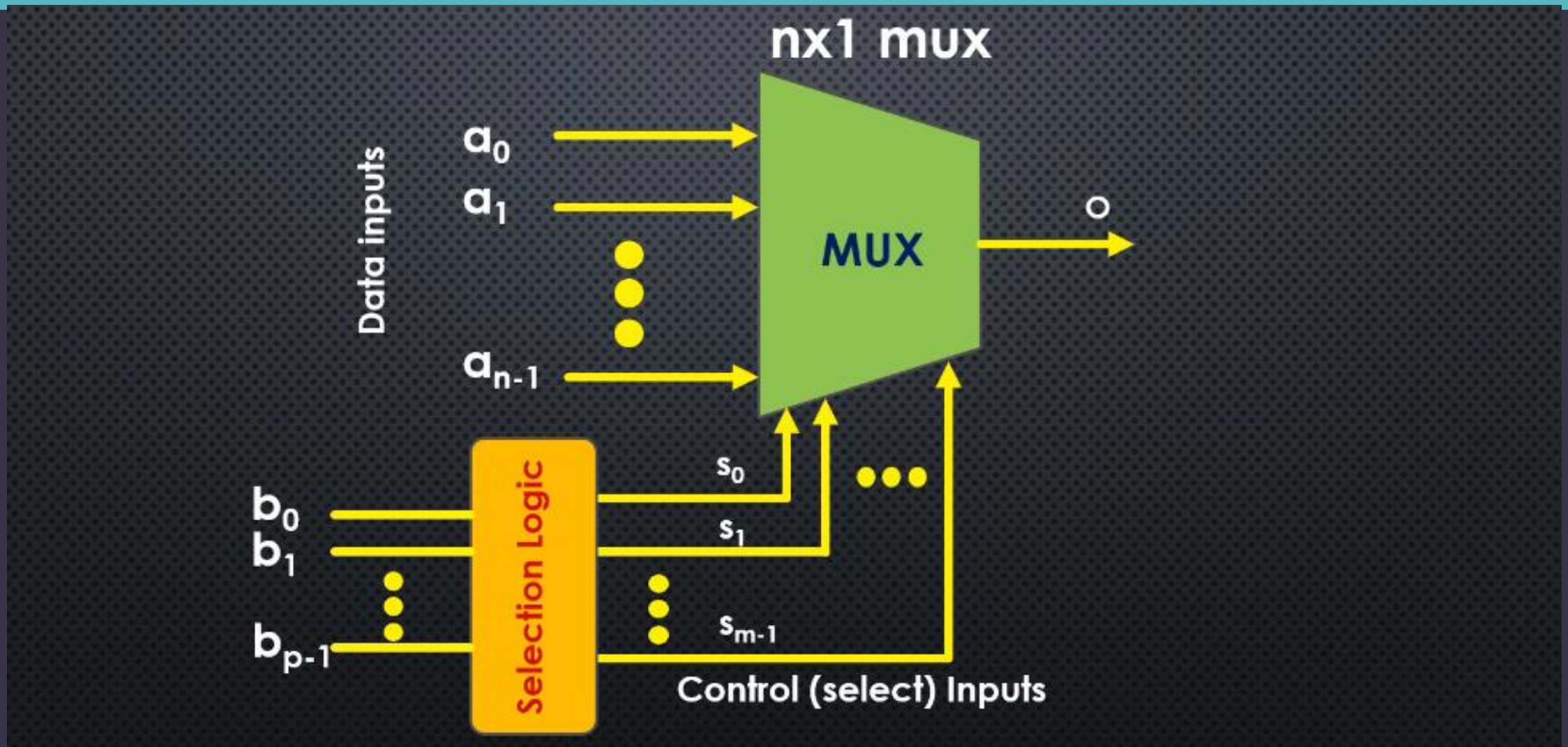
MULTIPLEXER-GENERAL CASE

Lecture – 4-Data type

WIDE BIT-WIDTH INPUTS (GENERAL CASE 1)



COMPLEX DECISION MAKING

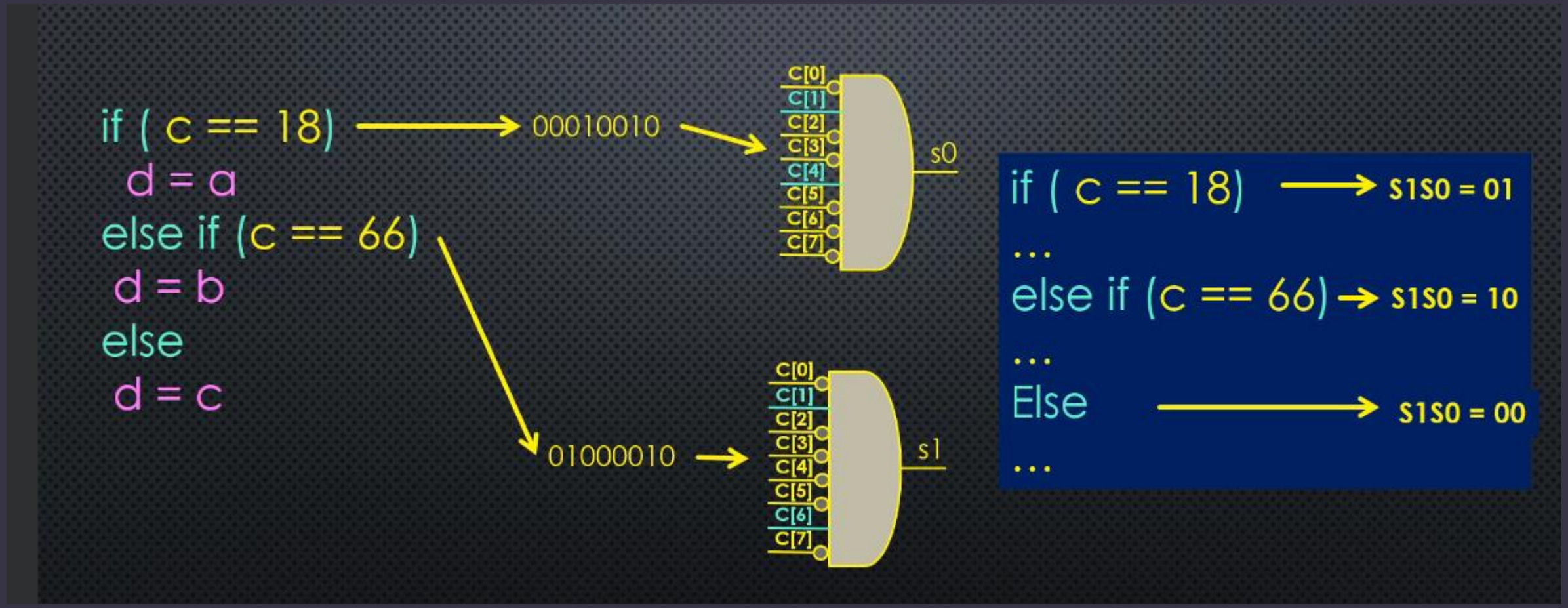


COMPLEX DECISION MAKING

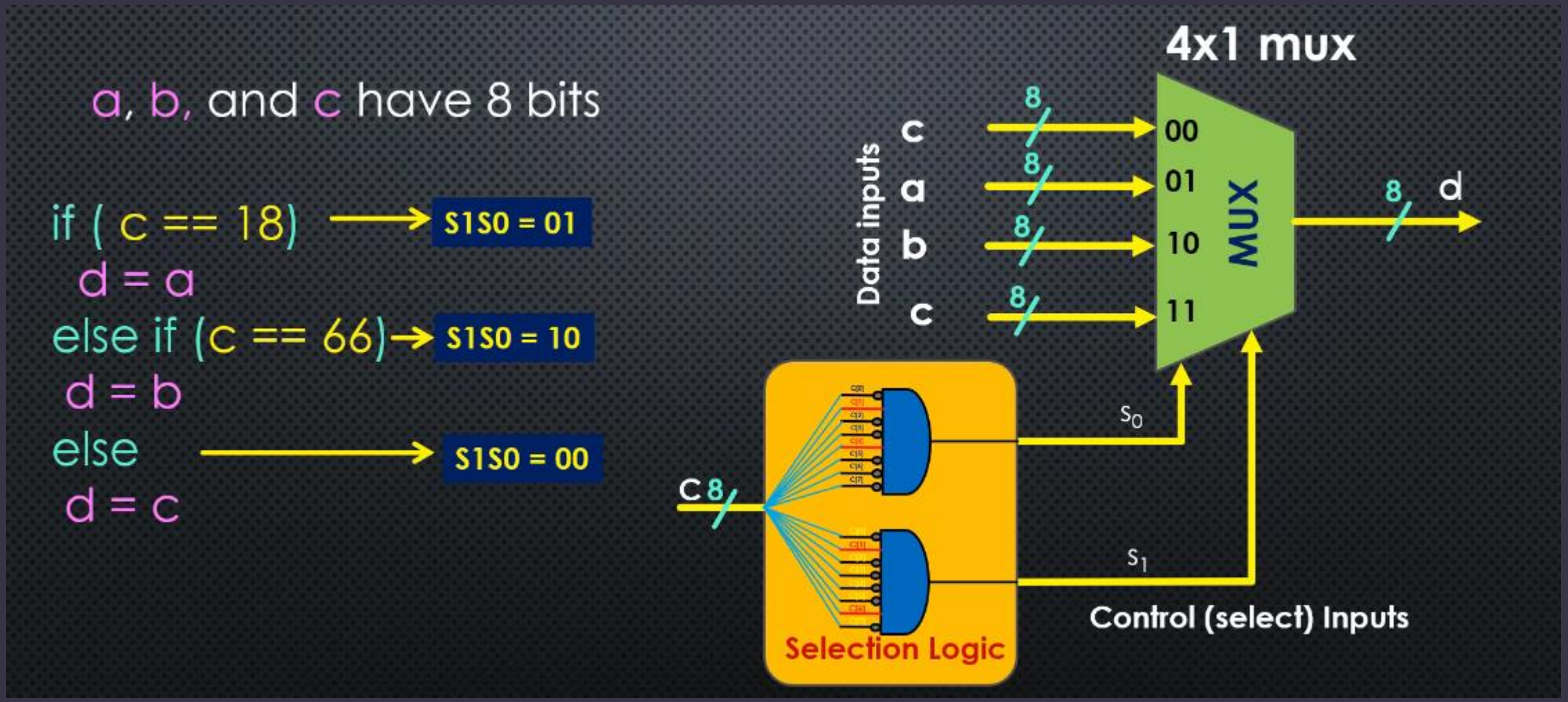
a, b, c and d have 8 bits

```
if ( c == 18)
  d = a
else if (c == 66)
 d = b
else
 d = c
```

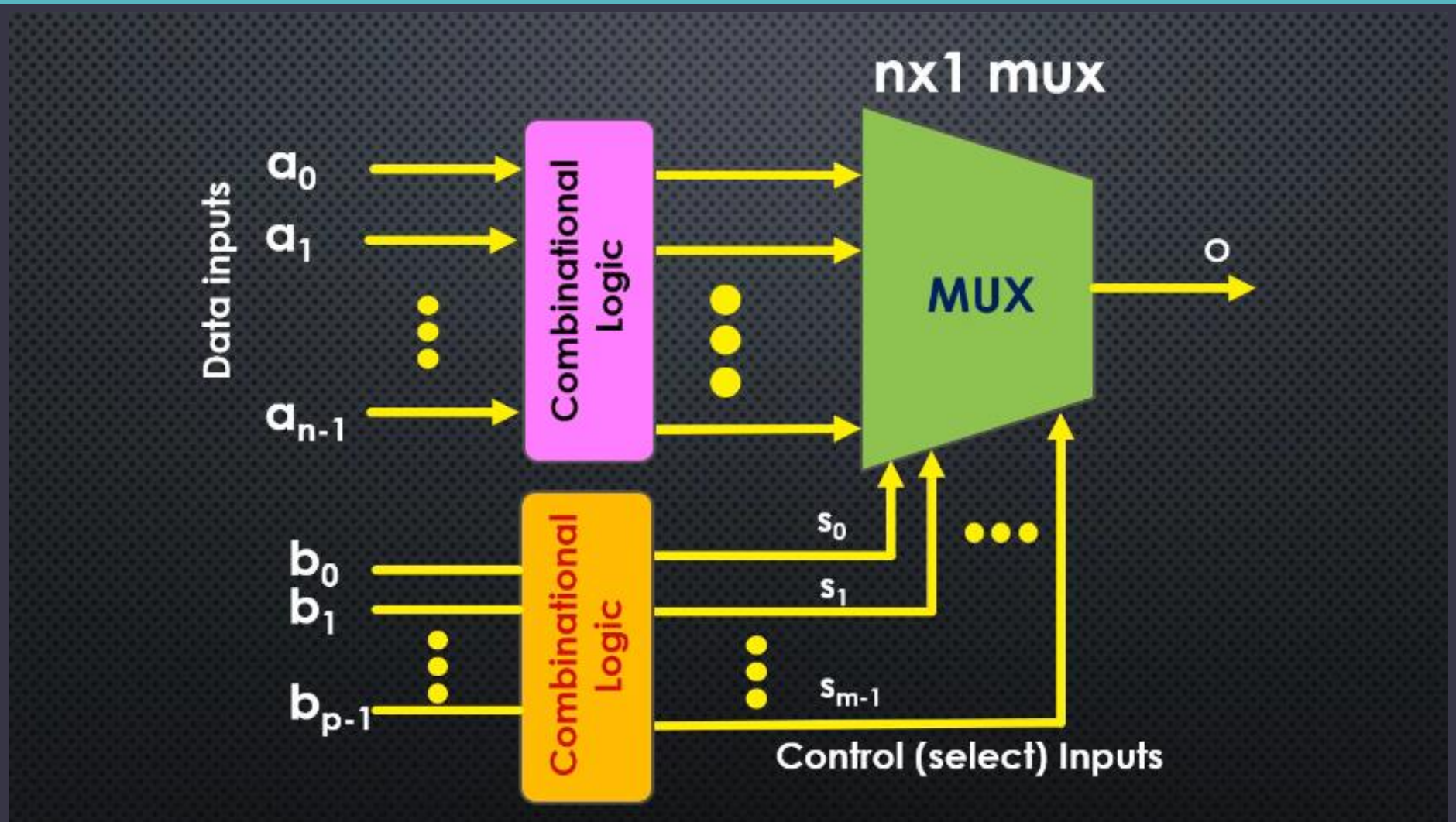
COMPLEX DECISION MAKING



COMPLEX DECISION MAKING



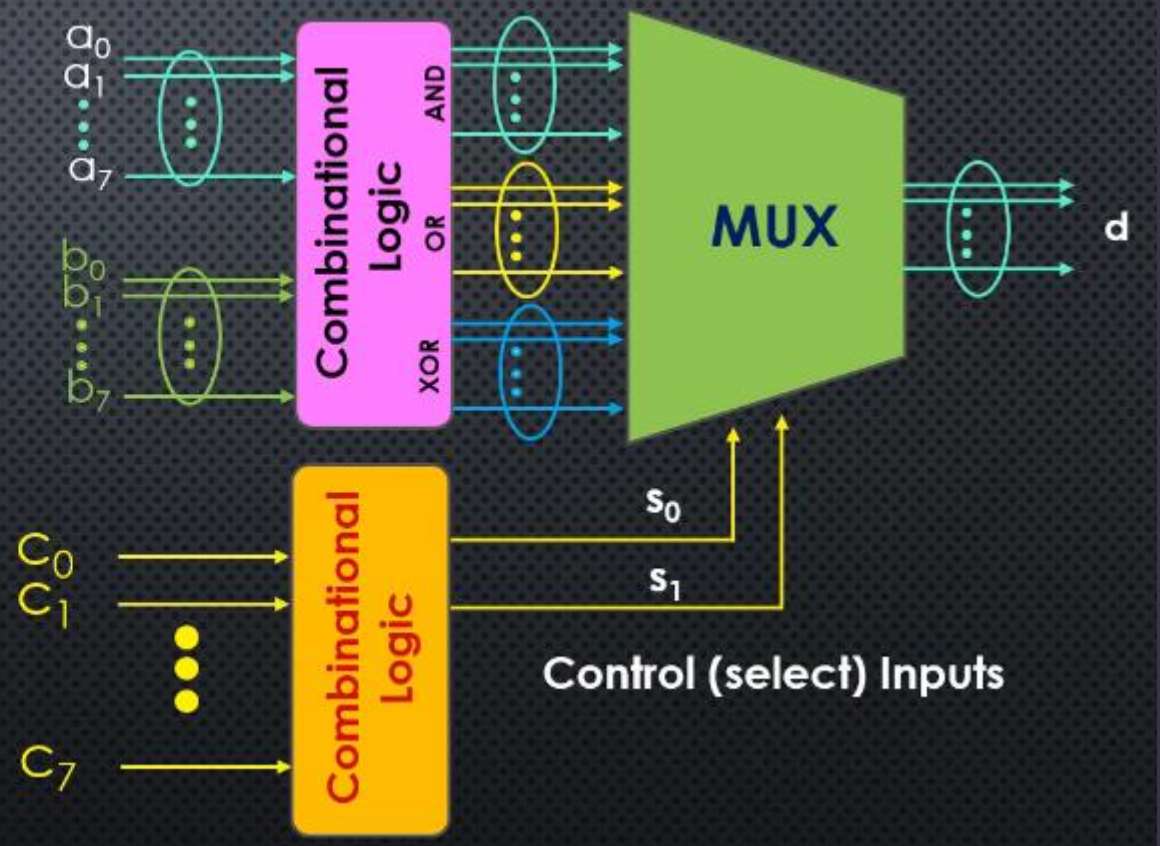
COMPLEX DATA PREPARATION (GENERAL CASE 3)



COMPLEX DATA PREPARATION EXAMPLE

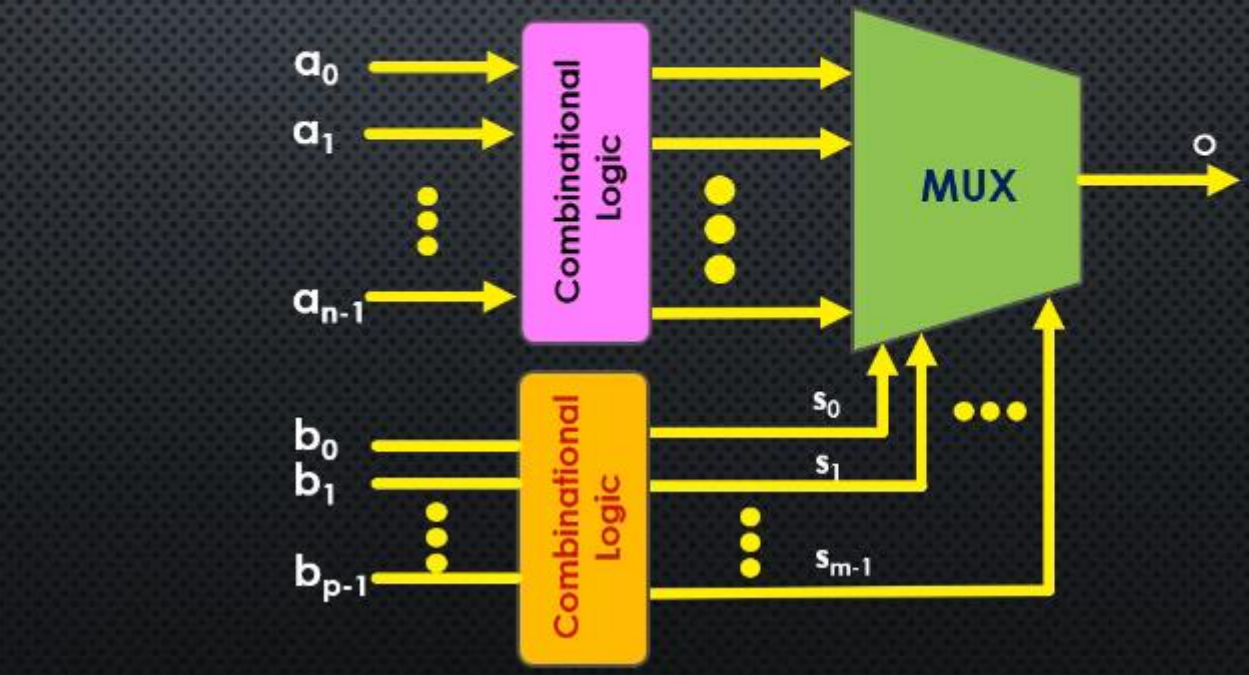
a, b, c and d have 8 bits

```
if ( c == 18)
  d = a & b
else if (c == 66)
  d = a | b
else
  d = a ^ b
```



SUMMARY

A decision-making process can be implemented by a mux and two combinational circuits feeding the mux data and control inputs.



ASSIGNMENT Q

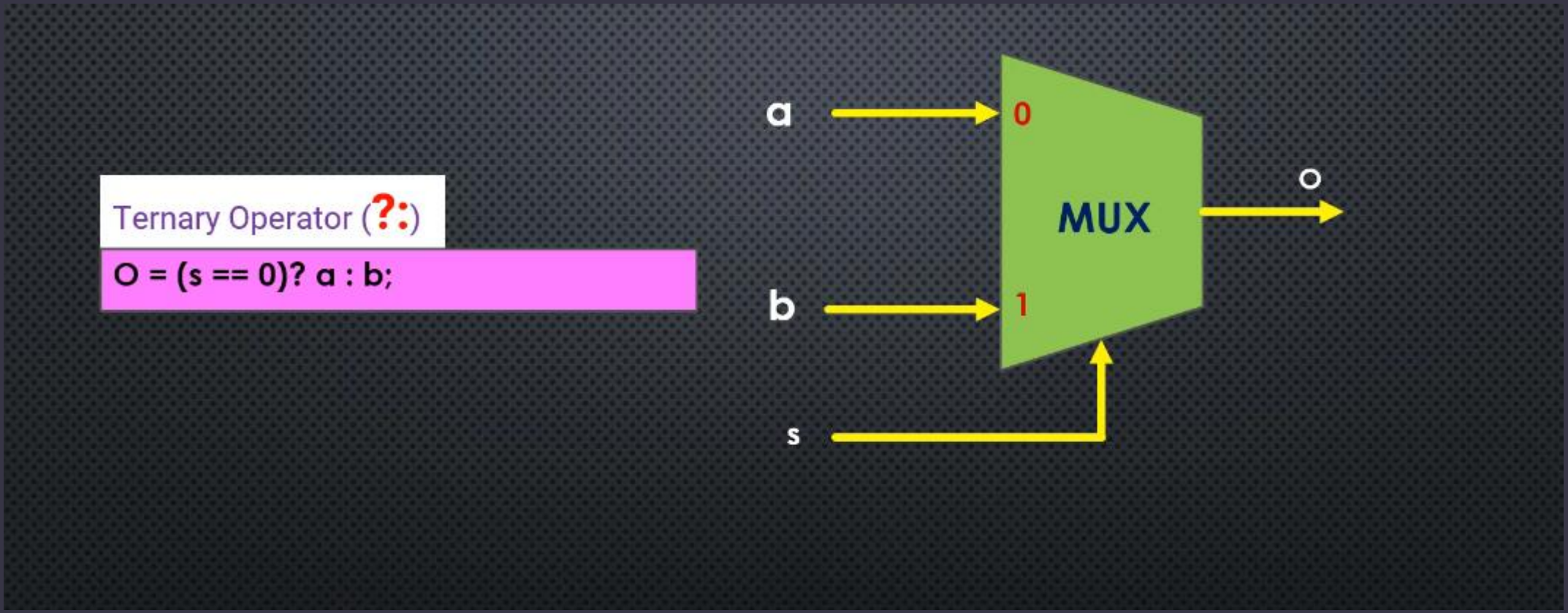
What is the hardware structure that implements this decision-making process?

```
if ( c < 18)
  d = a & b
else if (c > 20)
  d = a | b
else
  d = a ^ b
```

MULTIPLEXER-HLS

Lecture – 4-Data type

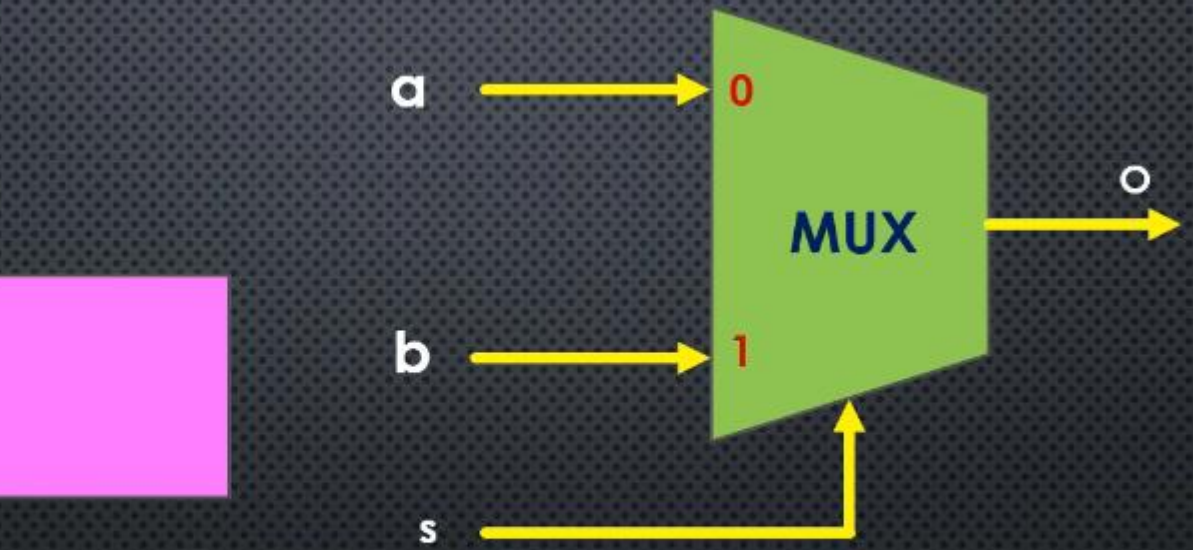
MULTIPLEXER IN HLS



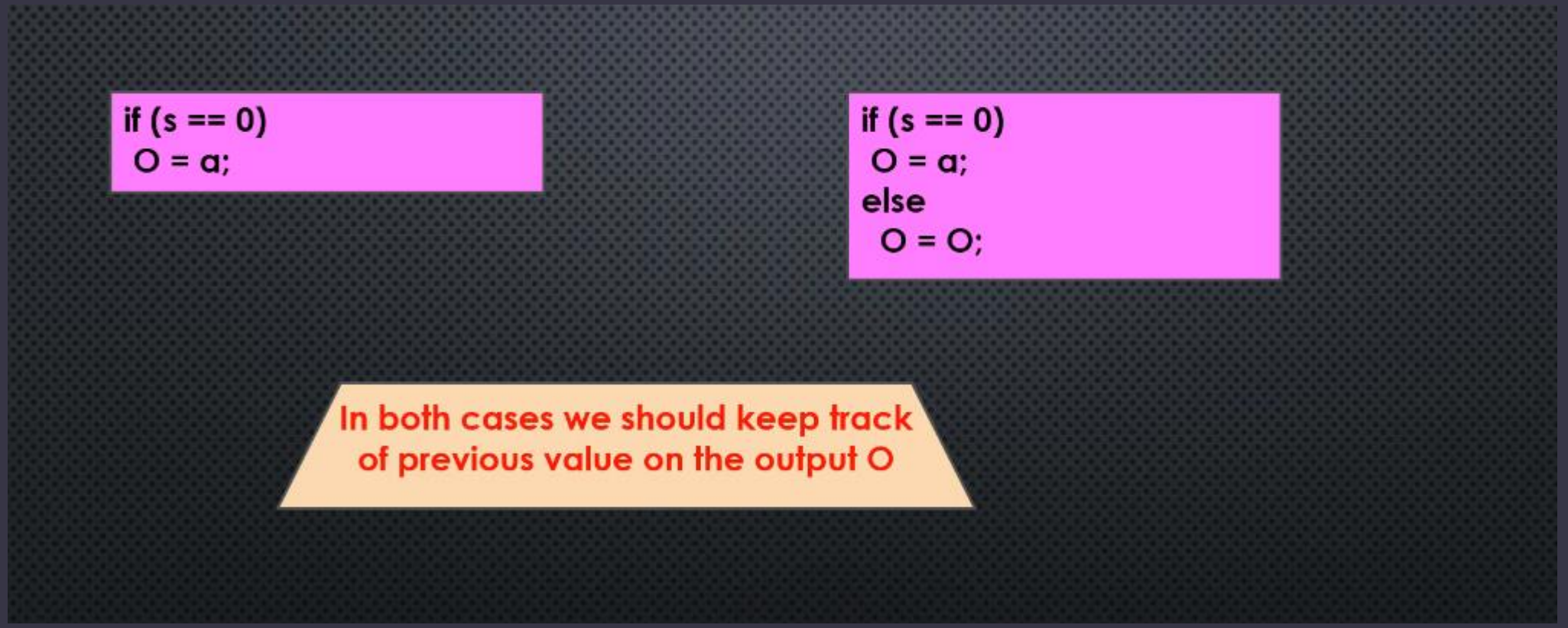
MULTIPLEXER IN HLS

If-else comparison

```
if (s == 0)
 O = a;
else
  O = b;
```



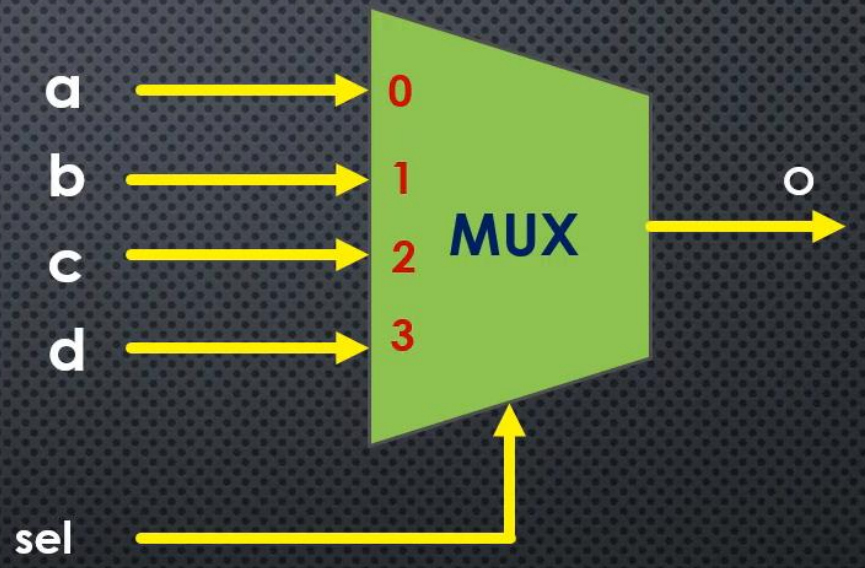
NOTE


if (s == 0)
O = a;
if (s == 0)
O = a;
else
O = O;
In both cases we should keep track of previous value on the output O

MULTIPLEXER IN HLS

Switch-Case

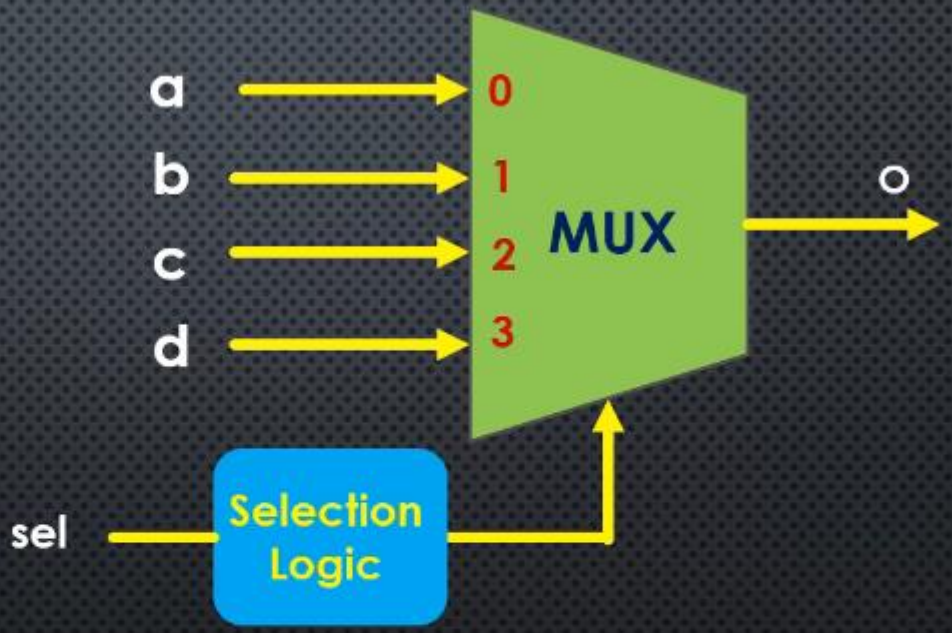
```
ap_int<2> sel;
...
switch (sel) {
case 0:
  O = a;
  break;
case 1:
  O = b;
  break;
case 2:
  O = c;
  break;
case 3:
  O = d;
  break;
}
```



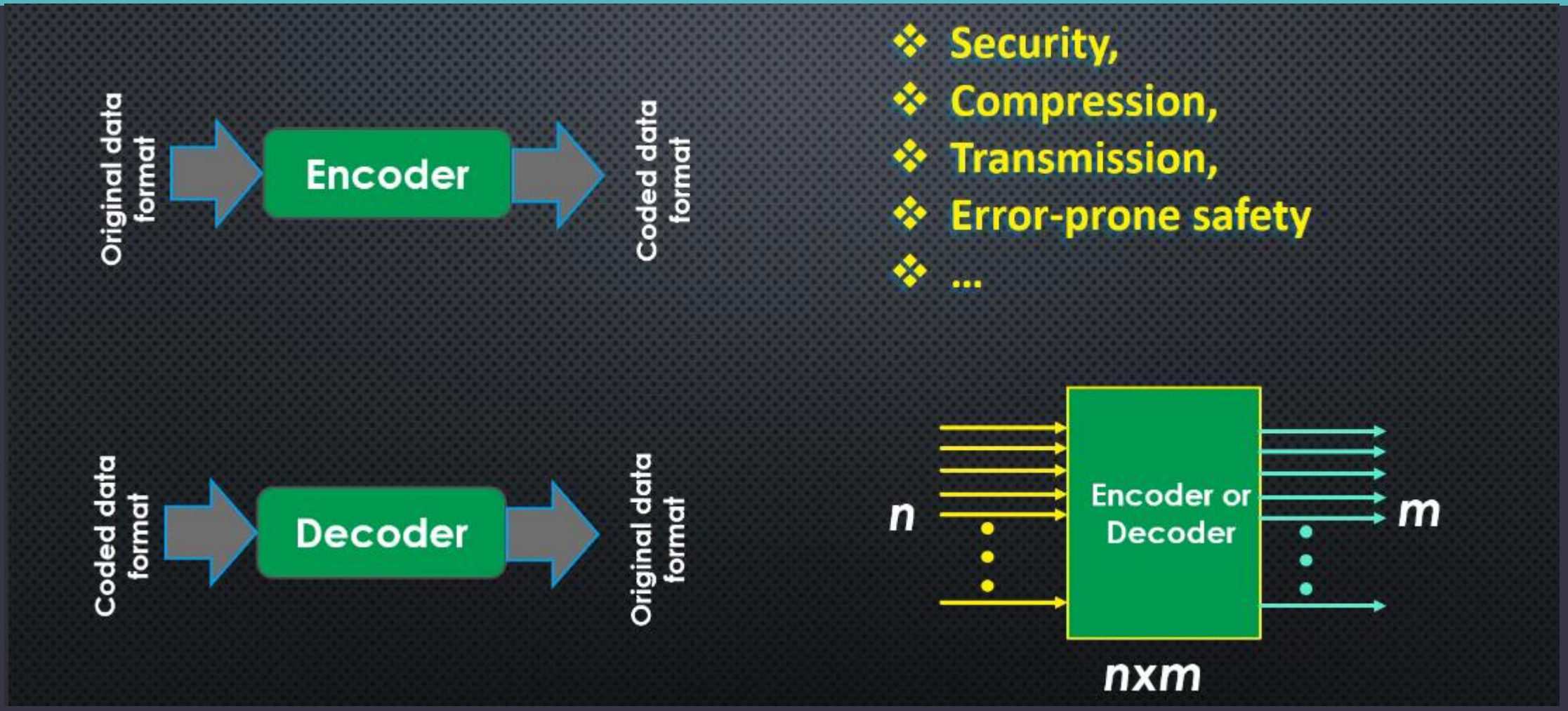
MULTIPLEXER IN HLS (SWITCH-CASE)

Switch-Case

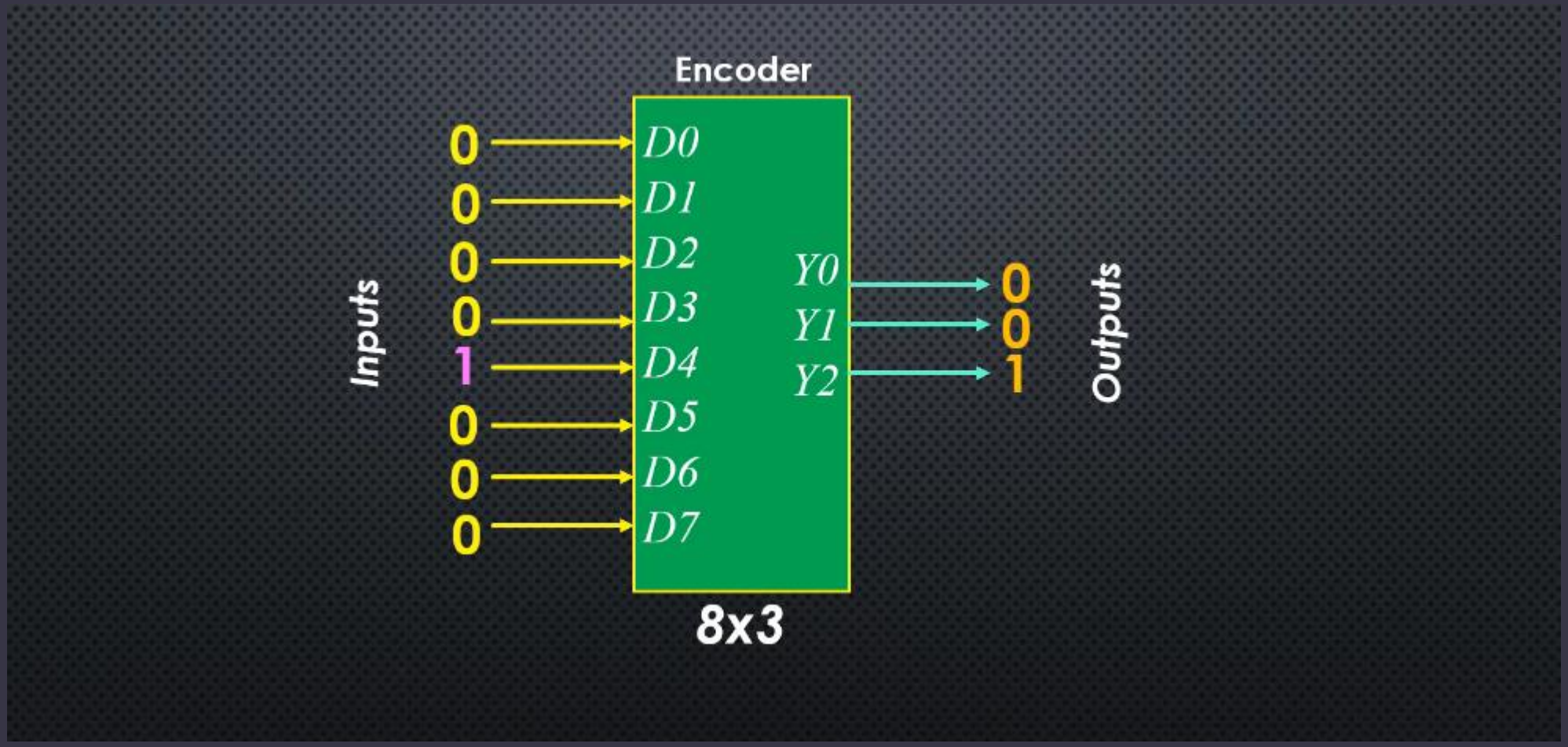
```
ap_int<8> sel;
...
switch (sel) {
case 0:
  O = a;
  break;
case 1:
case 2:
  O = b;
  break;
case 3:
case 4:
  O = c;
  break;
default:
  O = d;
  break;
}
```



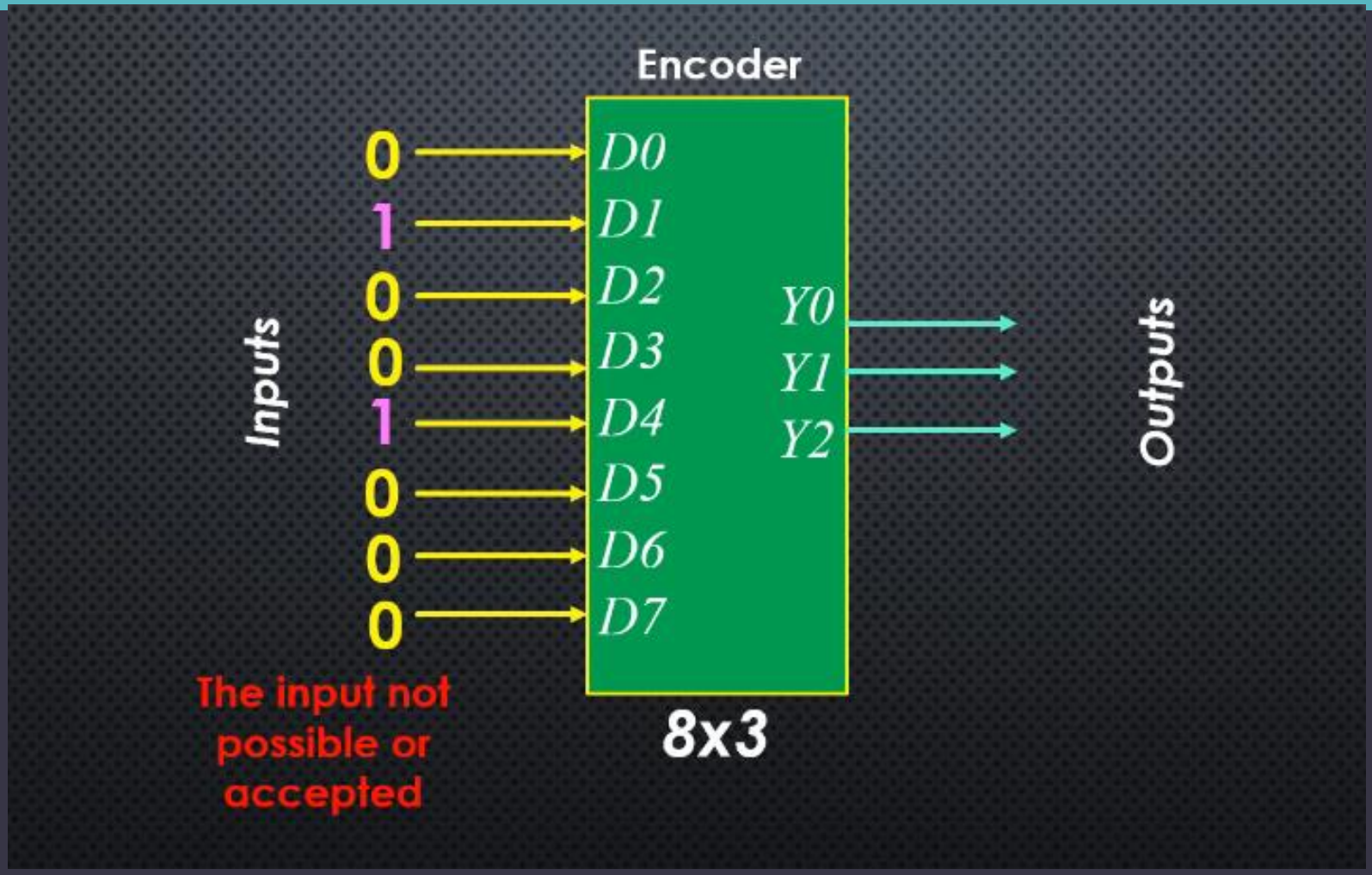
ENCODER AND DECODER



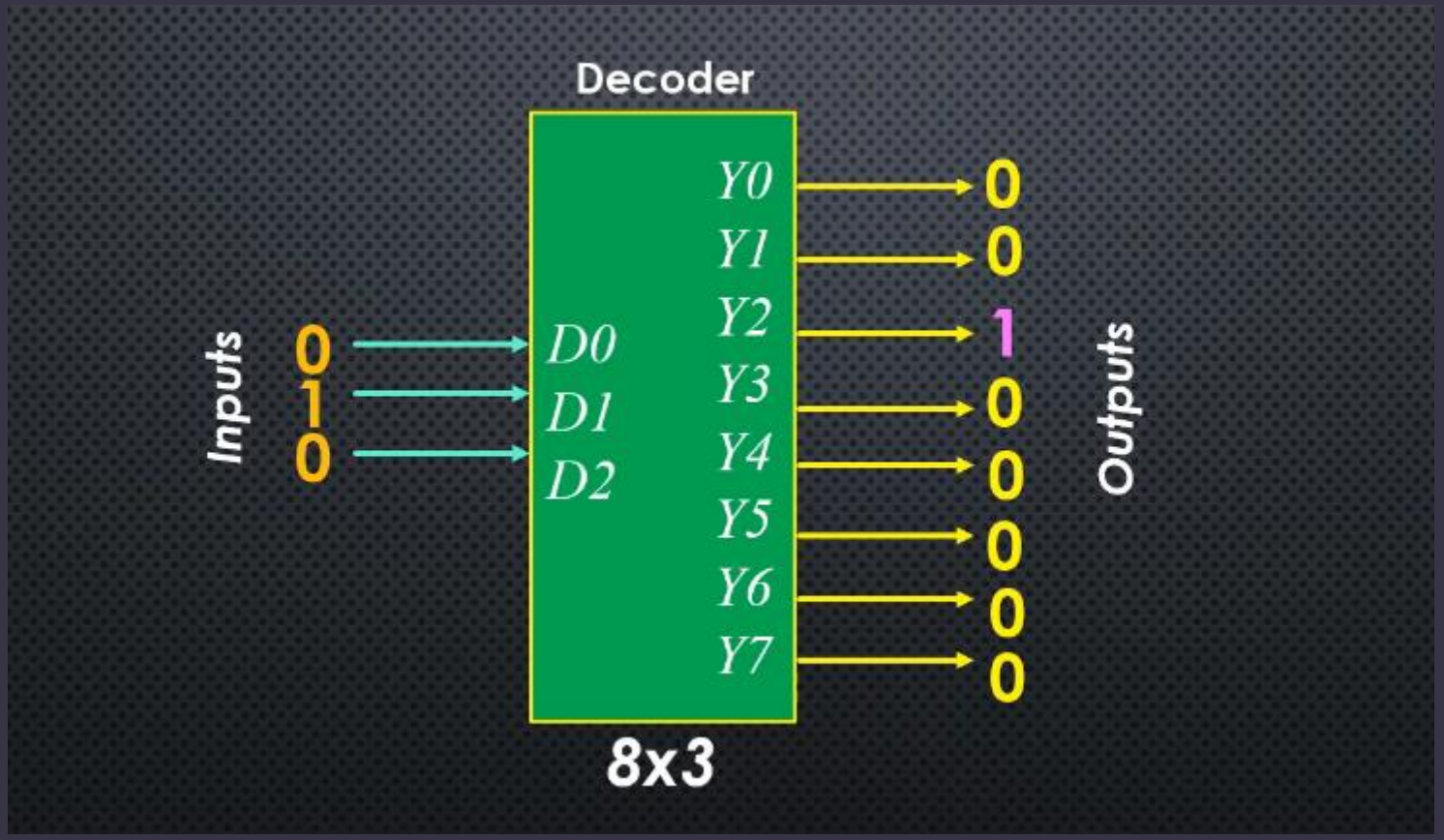
EXAMPLE: OCTAL TO BINARY ENCODER



EXAMPLE: OCTAL TO BINARY ENCODER



EXAMPLE: OCTAL TO BINARY DECODER



ENCODER: DEFINITION

a3	a2	a1	a0	b1	B0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

DECODER: DEFINITION

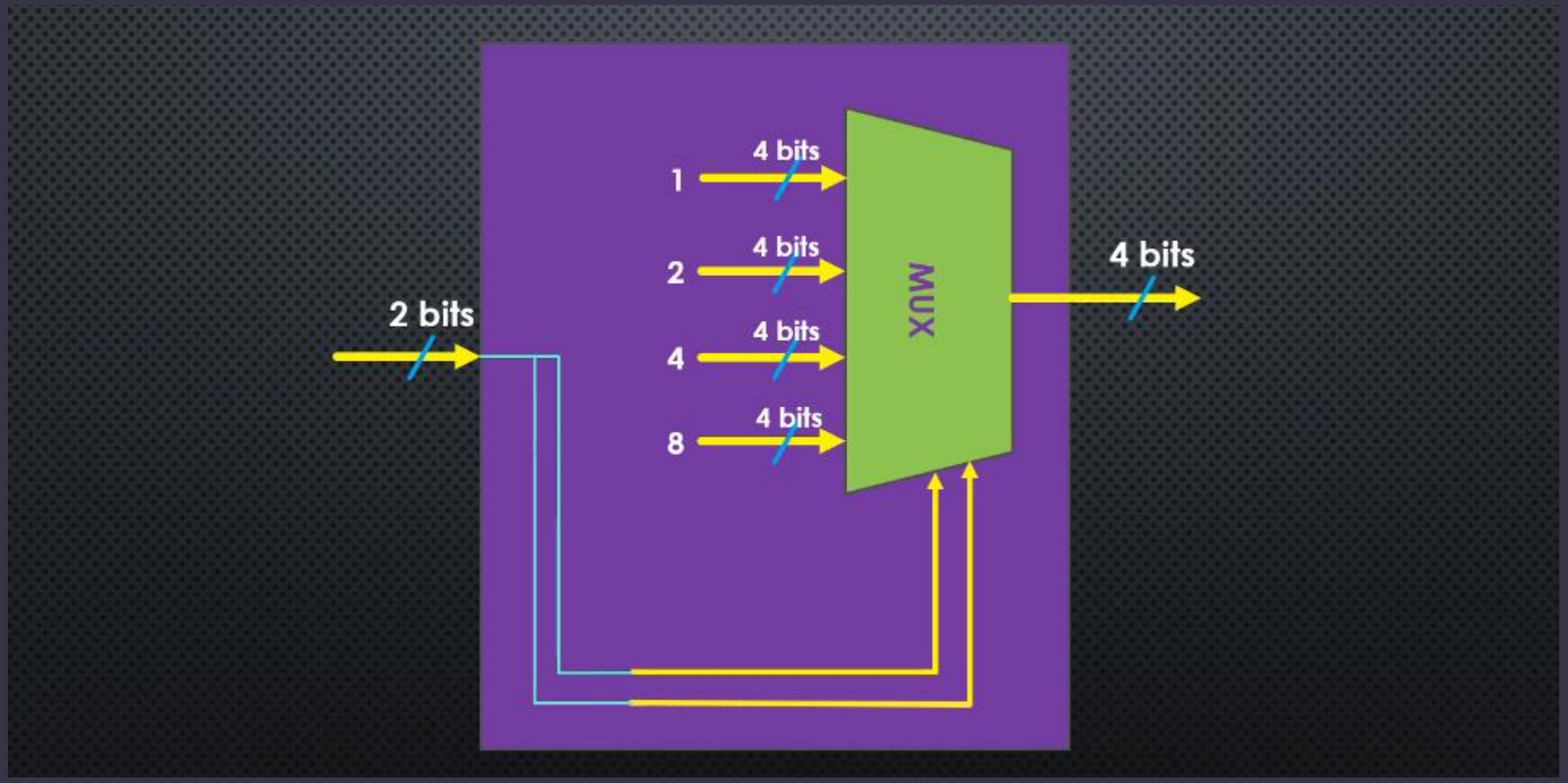
a1	a0	b3	b2	b1	B0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

DECODER IN HLS

a1	a0	b3	b2	b1	B0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

```
ap_int<4> decoder4x2(ap_uint<2> a) {
  ap_int<4> d;

  switch(a) {
  case 0:
    d = 0b0001;
    break;
  case 1:
    d = 0b0010;
    break;
  case 2:
    d = 0b0100;
    break;
  default:
    d = 0b1000;
  }
  return d;
}
```

DECODER HARDWARE AFTER HLS SYNTHESIS



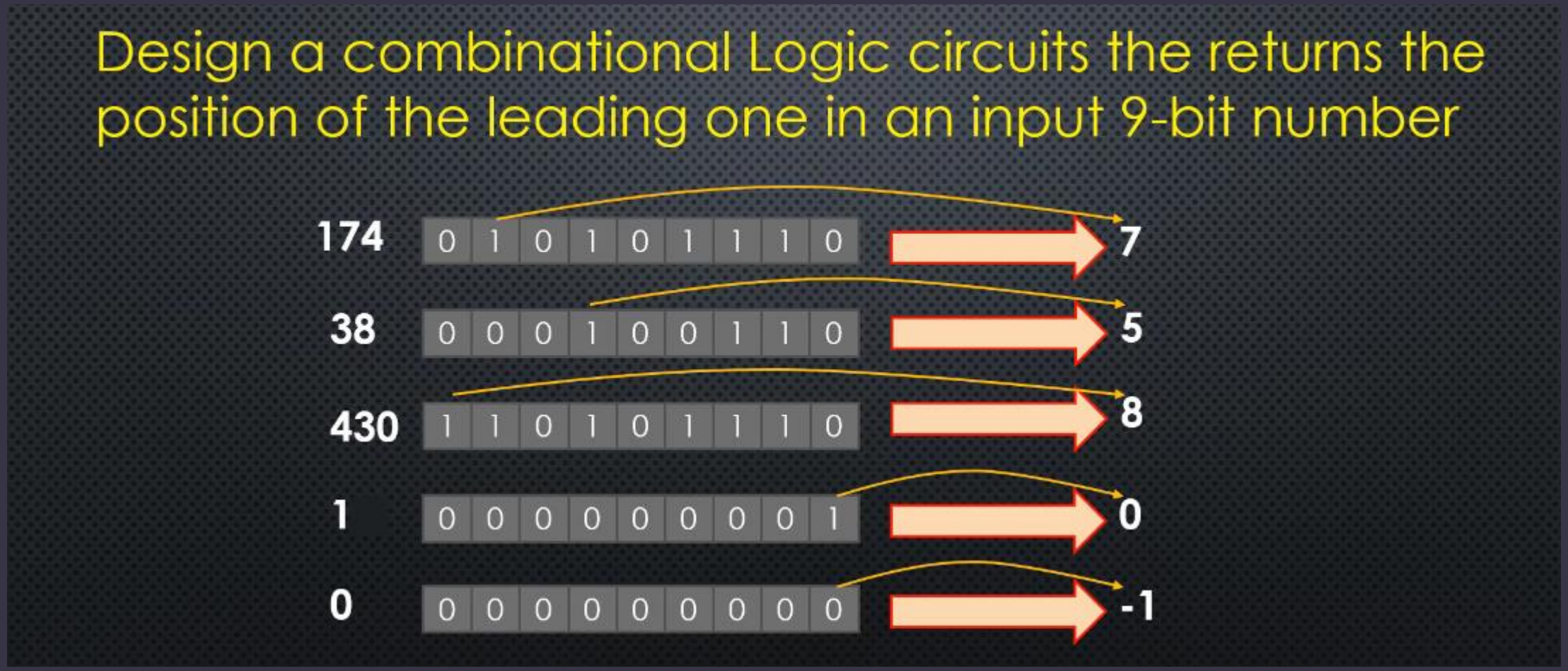
ASSIGNMENT Q

Use a switch-case statement in HLS to describe an 8 by 3 encoder circuit.

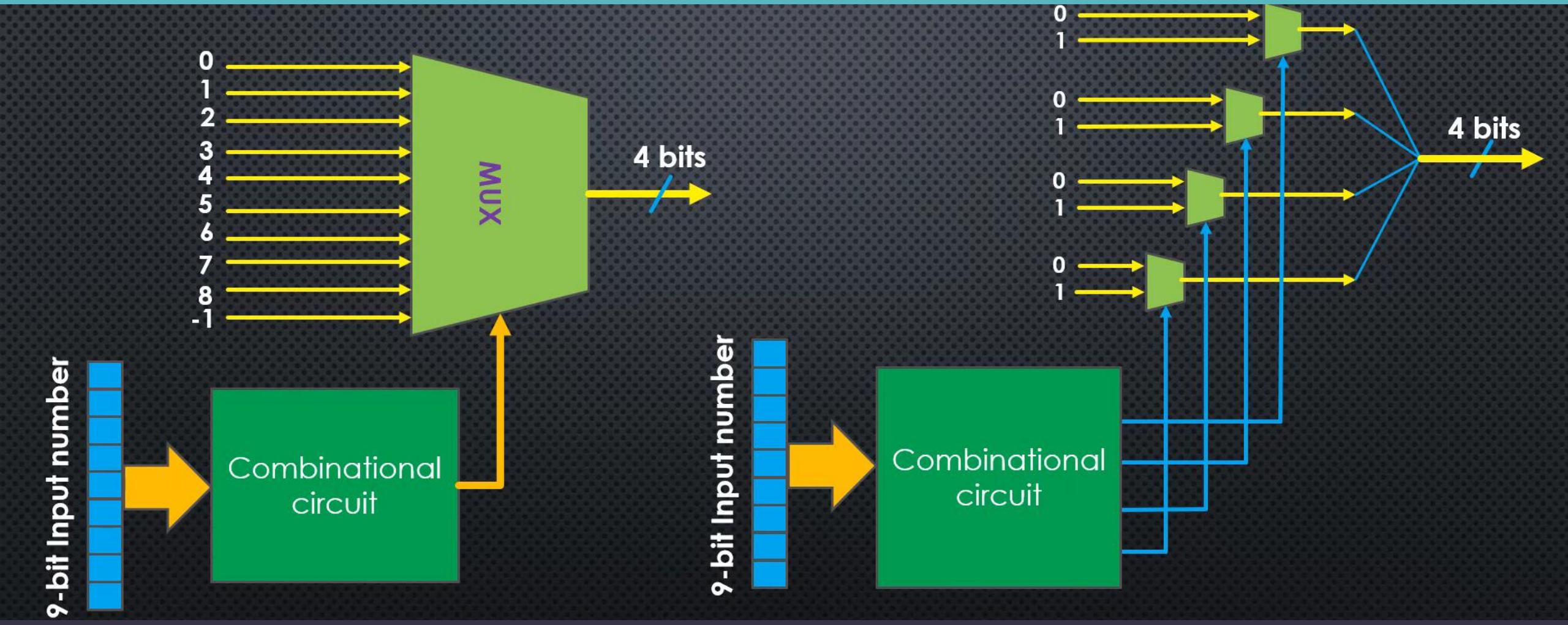
MULTIPLEXER-EXAMPLE

Lecture – 5

LEADING ONE: DEFINITION



LEADING ONE: LOGIC



LEADING ONE: HLS

```
ap_int<5> find_leading_one(ap_uint<9> a) {
  ap_int<5> index;

 if (a[8] == 1) {
    index = 8;
  } elseif (a[7] == 1) {
    index = 7;
  } elseif (a[6] == 1) {
    index = 6;
  } elseif (a[5] == 1) {
    index = 5;
  } elseif (a[4] == 1) {
    index = 4;
  } elseif (a[3] == 1) {
    index = 3;
  } elseif (a[2] == 1) {
    index = 2;
  } elseif (a[1] == 1) {
    index = 1;
  } elseif (a[0] == 1) {
    index = 0;
  } else {
    index = -1;
  }
  return index;
}
```

ASSIGNMENT Q

This code implements a 4x2 encoder; its structure is very similar to the leading-one example. Find the missing numbers that determined by red question marks.

```
ap_int<?> encoder4x2(ap_uint<?> a) {
  ap_int<?> code;

  if (a[?] == 1) {
    code = 3;
  } elseif (a[?] == 1) {
    code = 2;
  } elseif (a[?] == 1) {
    code = 1;
  } else {
    code = 0;
  }

  return code;
}
```

Any Question...

Thank you