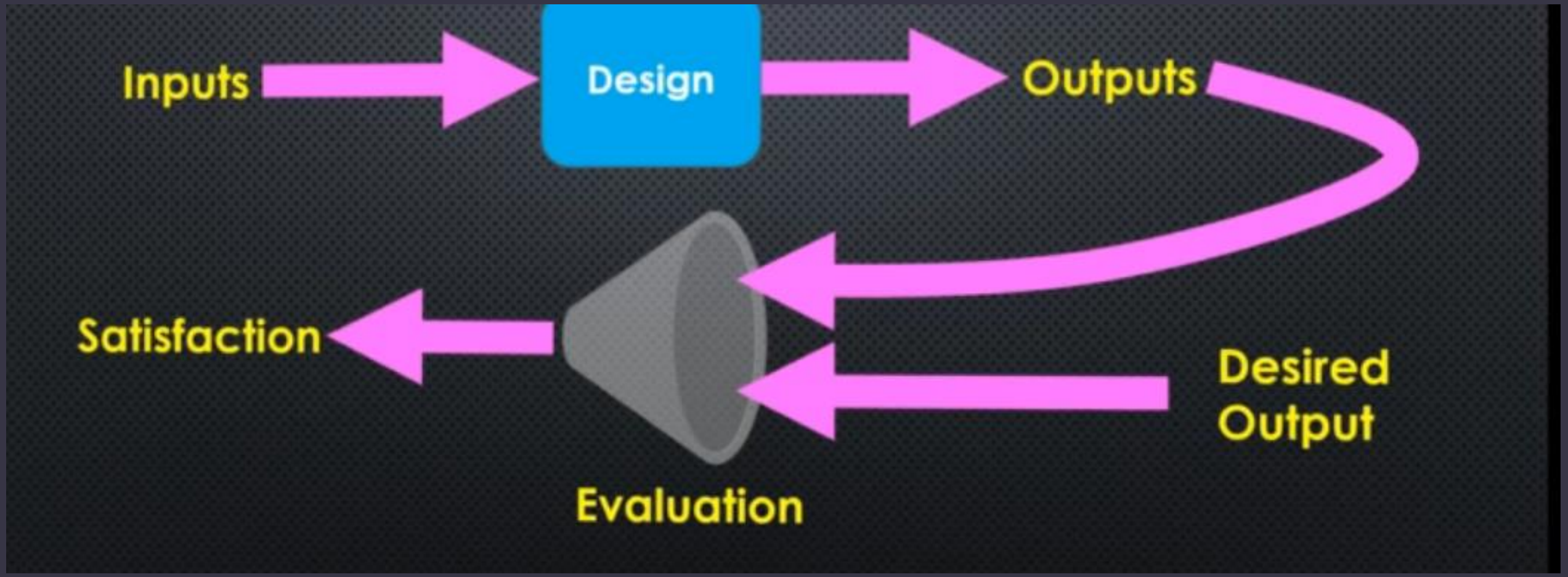
C/C++ TEST BENCH

Lecture - 4

CODE SIMULATION

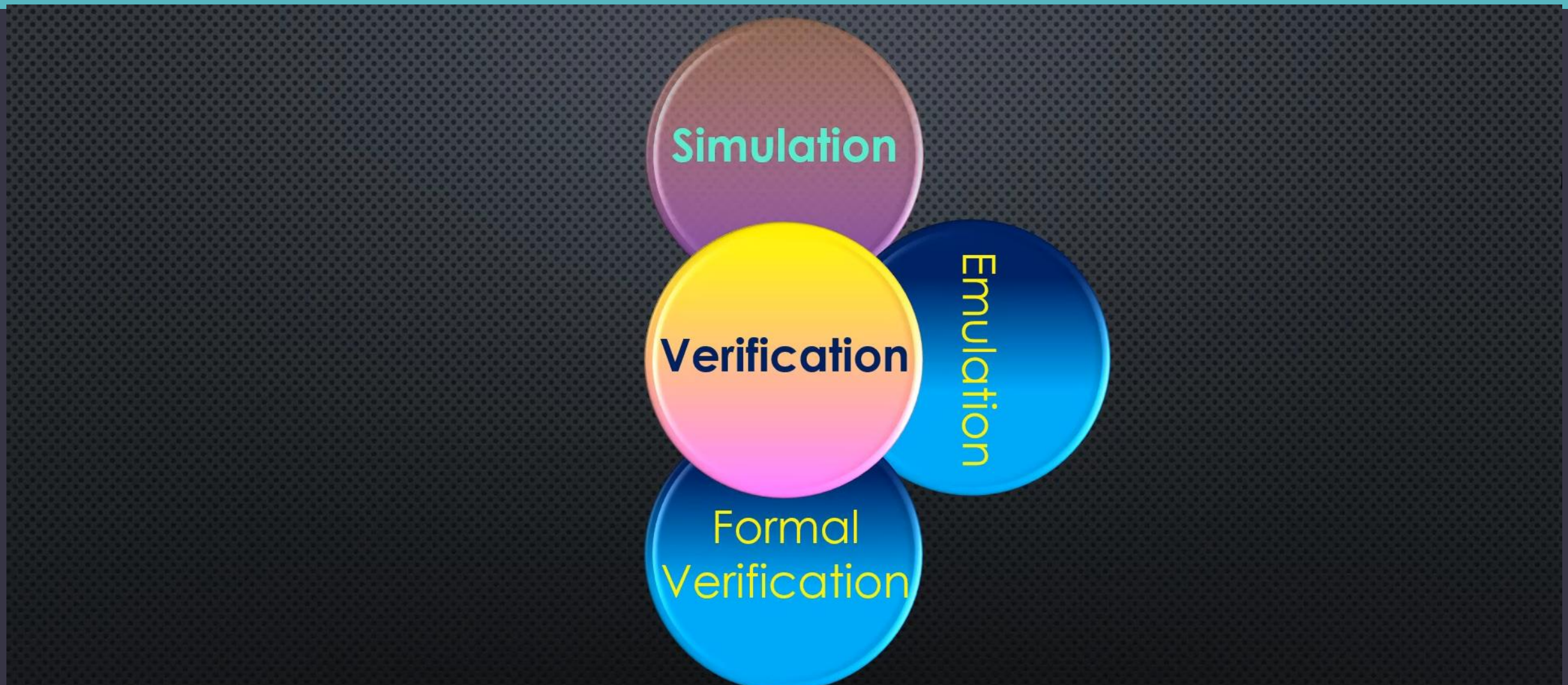


DESIGN VERIFICATION

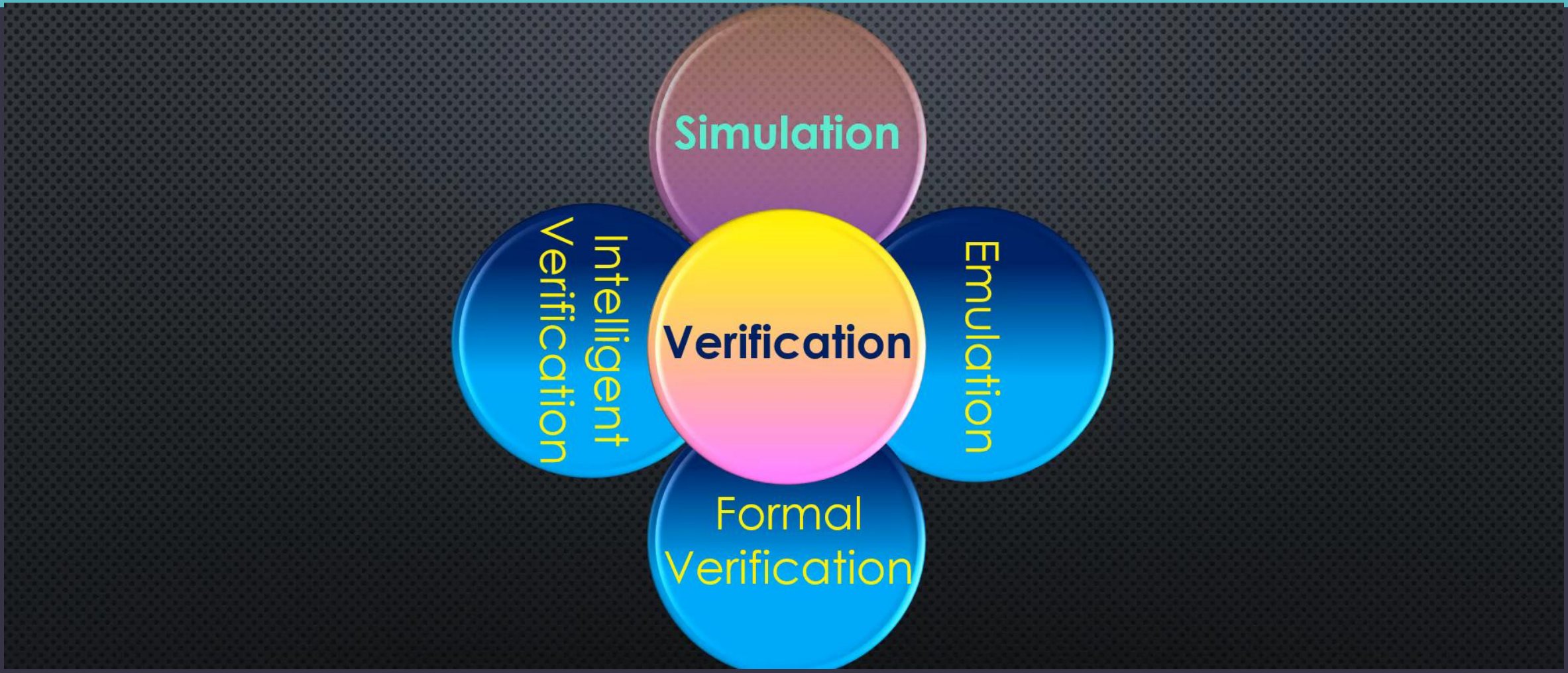
DESIGN VERIFICATION

DESIGN VERIFICATION

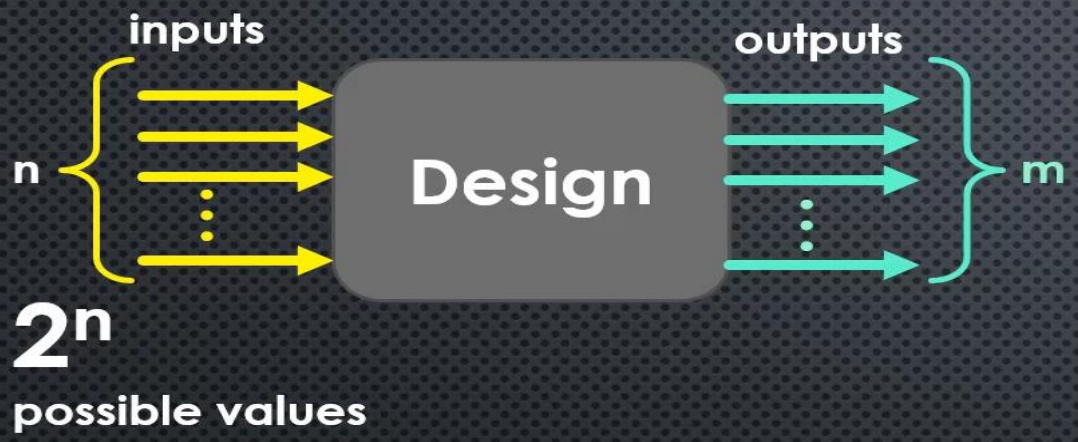
DESIGN VERIFICATION



DESIGN VERIFICATION



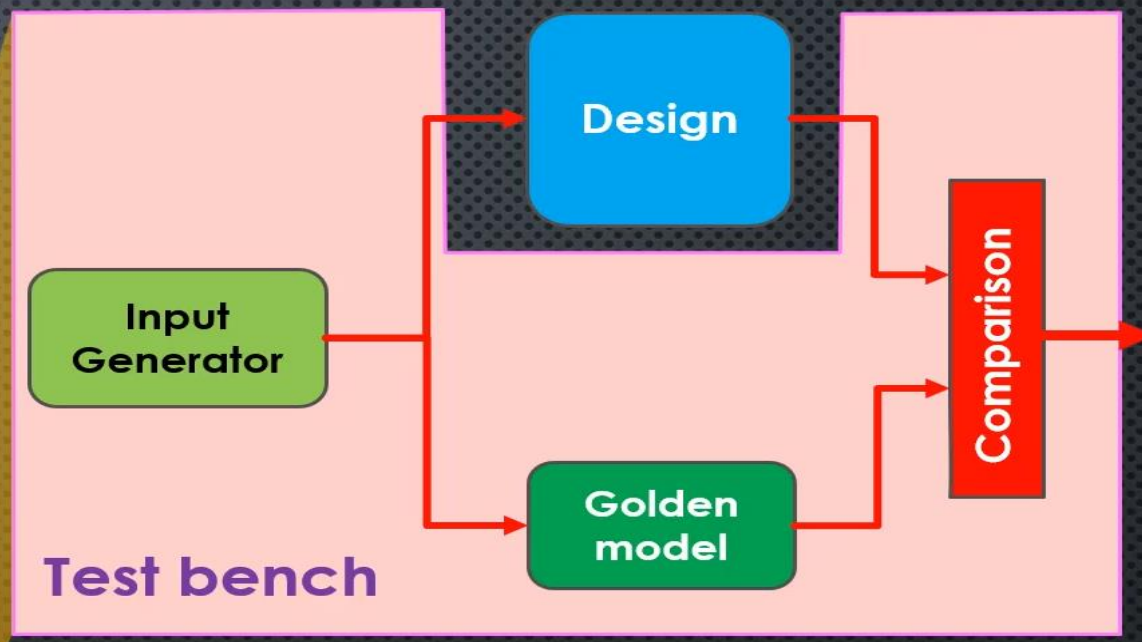
SIMULATION-TEST BENCH



2^n possible values

- Generate inputs
- Apply inputs to the design
- Apply inputs to the golden model
- Compare the outputs of design and golden model

Test bench



ASSIGNMENT Q

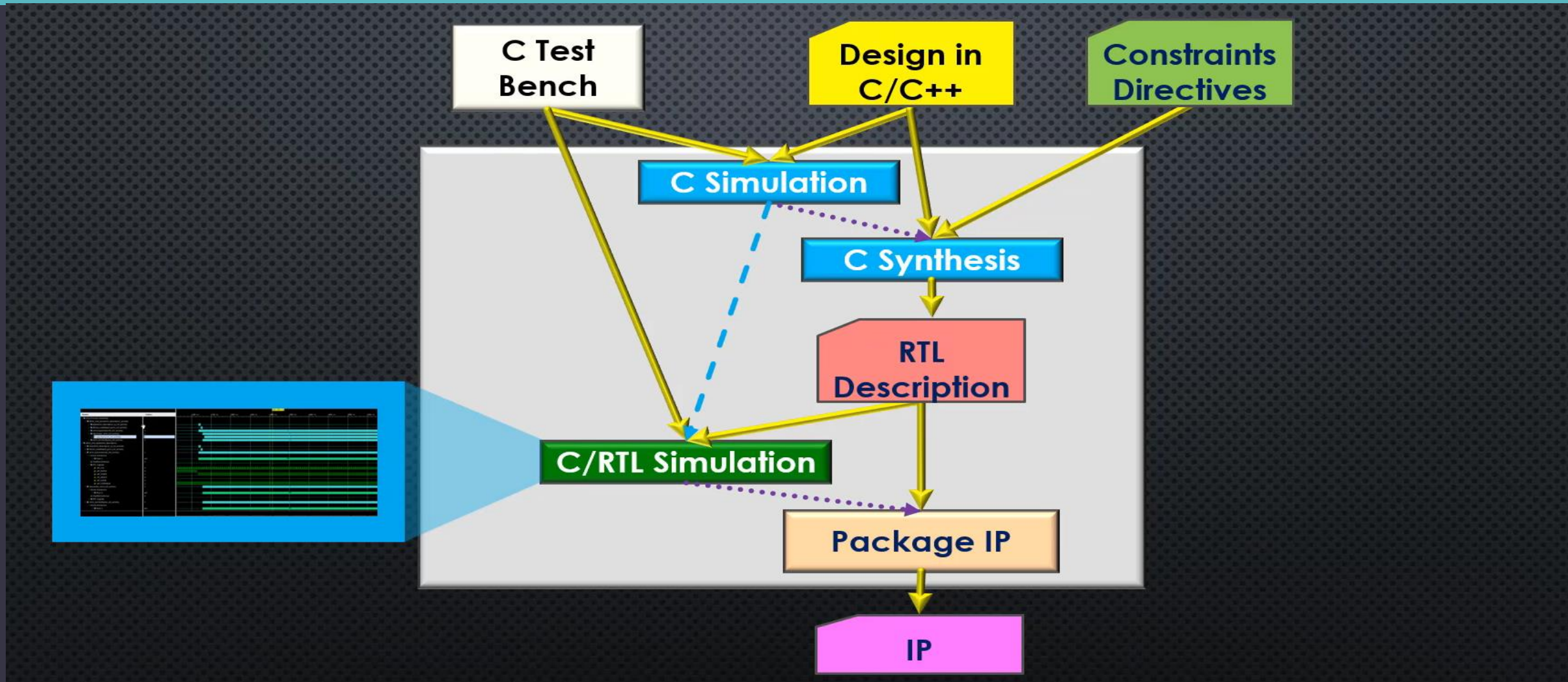
Assume a hardware design has five binary inputs and four binary outputs,

how many input vectors should a test bench generate to check the functionality of the design.

SIMULATION FLOW

Lecture - 4

HLS DESIGN FLOW WITH SIMULATION



HLS DESIGN/SIMULATION FLOW

ASSIGNMENT Q

Which sentence is correct

1. Cycle accurate debugging is possible in C-simulation.
2. Cycle accurate debugging is possible in C/RTL-simulation.

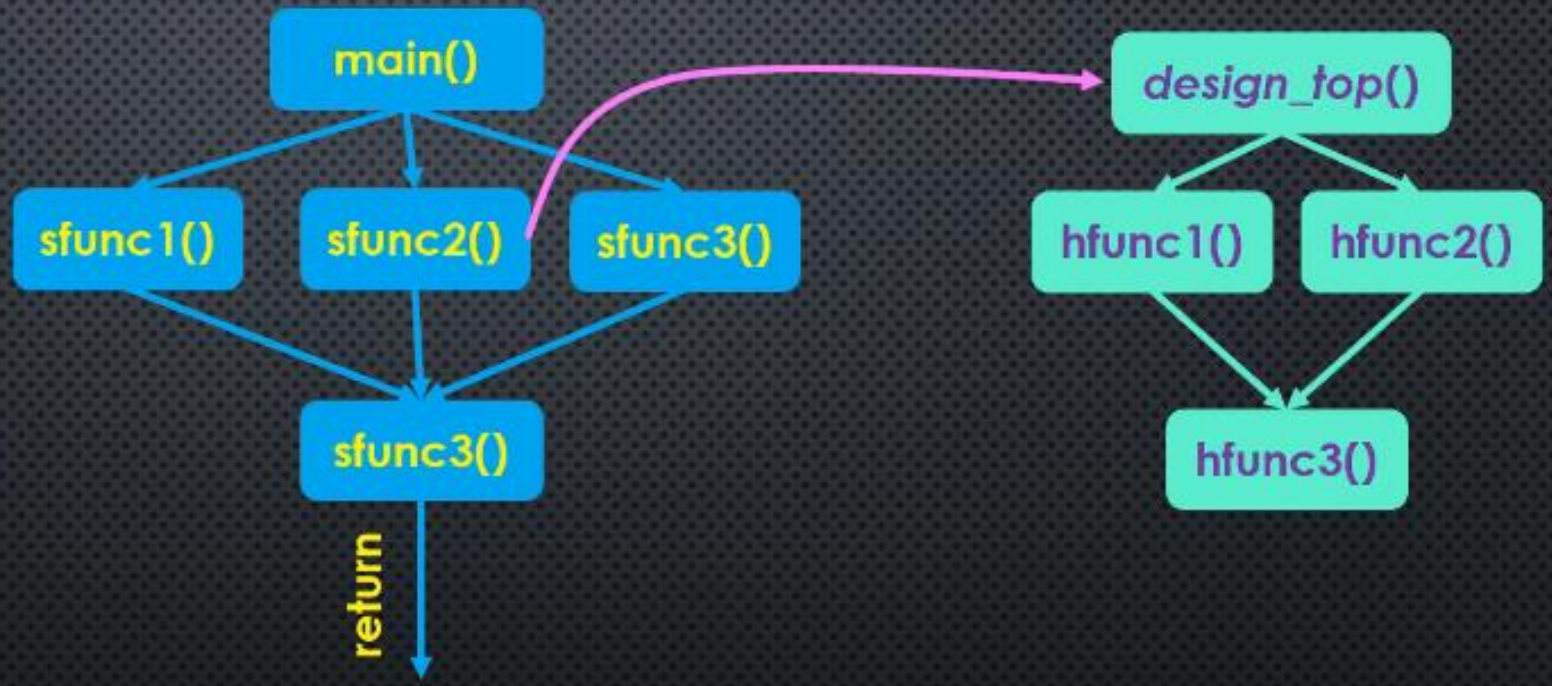
HLS CODING

Lecture - 4

TEST BENCH – TOP FUNCTION

Test bench top-function

```
int main() {


  return ...;
}
```



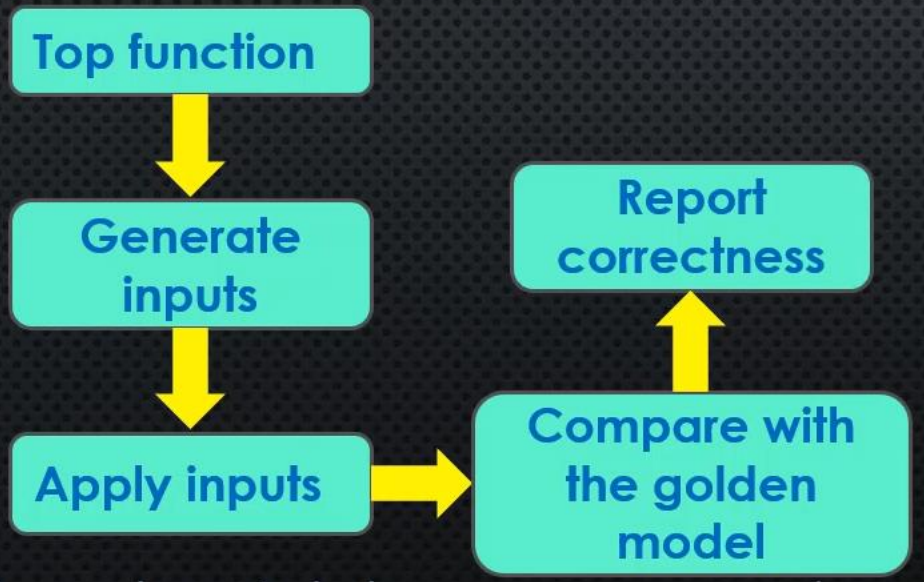
SELF-CHECKING CONCEPT

```
int main() {
  int status = 0;

   …
   status = …
   …

  return status;
}
```

Test bench top-function

TEST BENCH EXAMPLE

```
void basic_input_output (
  char     input,
  char   *output )
{
   *output = input;
}
```




```
int main() {
  int status = 0;
  unsigned char a, b;
  for (int i = 0; i < 256; i++) {
    a = (unsigned char)i;
    basic_input_output(a, &b);
    if (b != a) {
      status = -1;
      break;
    }
  }
  if (status == 0) {
    std::cout << "test passed" << std::endl;
  } else {
    std::cout << "test failed" << std::endl;
  }

  return status;
}
```

TEST BENCH CONSTRAINT

- You are responsible for ensuring that the test bench checks the results.
- The test bench must *not* require the execution of interactive user inputs.

SUMMARY

- A test bench top-function of a design consists of four main parts:
  - Generating the design inputs
  - Calling the design top-function using the generated inputs
  - Comparing the design outputs with the golden model outputs
  - Returning the success of the design verification.

ASSIGNMENT Q

Write a testbench function for the following design.

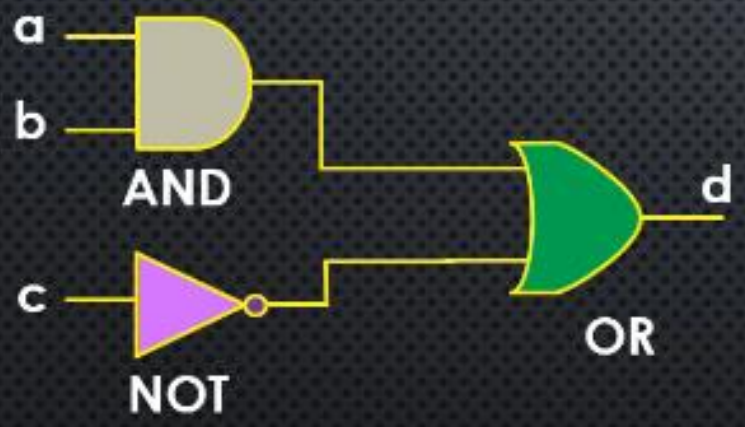
```
void design_file(
  short unsigned int     input,
  unsigned int   *output )
{
   *output = 2*input;
}
```

HLS LAB

Lecture - 4

SIMPLE COMBINATIONAL CIRCUIT

```
*d = (a && b) || !c;
```



a	b	c	d
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

DATA TYPE

Lecture - 4

DATA TYPE IN SOFTWARE

DATA TYPE IN HARDWARE

Bit-accurate data types

```
#include “ap_int.h”
ap_int<5>     a;
ap_uint<11> b;
```

23-bit data

NATIVE C++ DATA TYPES

Data Type	Example
bool	0, 1 or false, true
char	45, -11
int	45, -14753

- signed
- unsigned
- short
- long

Type specifier	Bit width	Range
bool	1 bit	0 or 1
char	8 bits	-128 to 127
unsigned char	8 bits	0 to 255
signed char	8 bits	-128 to 127
int	32 bits	-2147483648 to 2147483647
unsigned int	32 bits	0 to 4294967295
signed int	32 bits	-2147483648 to 2147483647
short int	16 bits	-32768 to 32767
unsigned short int	16 bits	0 to 65,535
signed short int	16 bits	-32768 to 32767
long int	32 bits	-2147483648 to 2147483647
signed long int	32 bits	-2147483648 to 2147483647
unsigned long int	32 bits	0 to 4294967295
long long int	64 bits	$-(2^{63})$ to $(2^{63})-1$
unsigned long long int	64 bits	0 to $(2^{64})-1$

NATIVE C/C++ DATA TYPES

Type specifier	Bit width
float	32 bits
double	64 bits

- **float**: single-precision 32 bit
  - 24-bit fraction
  - 8-bit exponent
- **double**: double-precision 64 bit
  - 53-bit fraction
  - 11-bit exponent

Vivado HLS supports float and double types for synthesis. Both data types are synthesized with IEEE-754 standard compliance.
https://en.wikipedia.org/wiki/IEEE_754

ASSIGNMENT Q

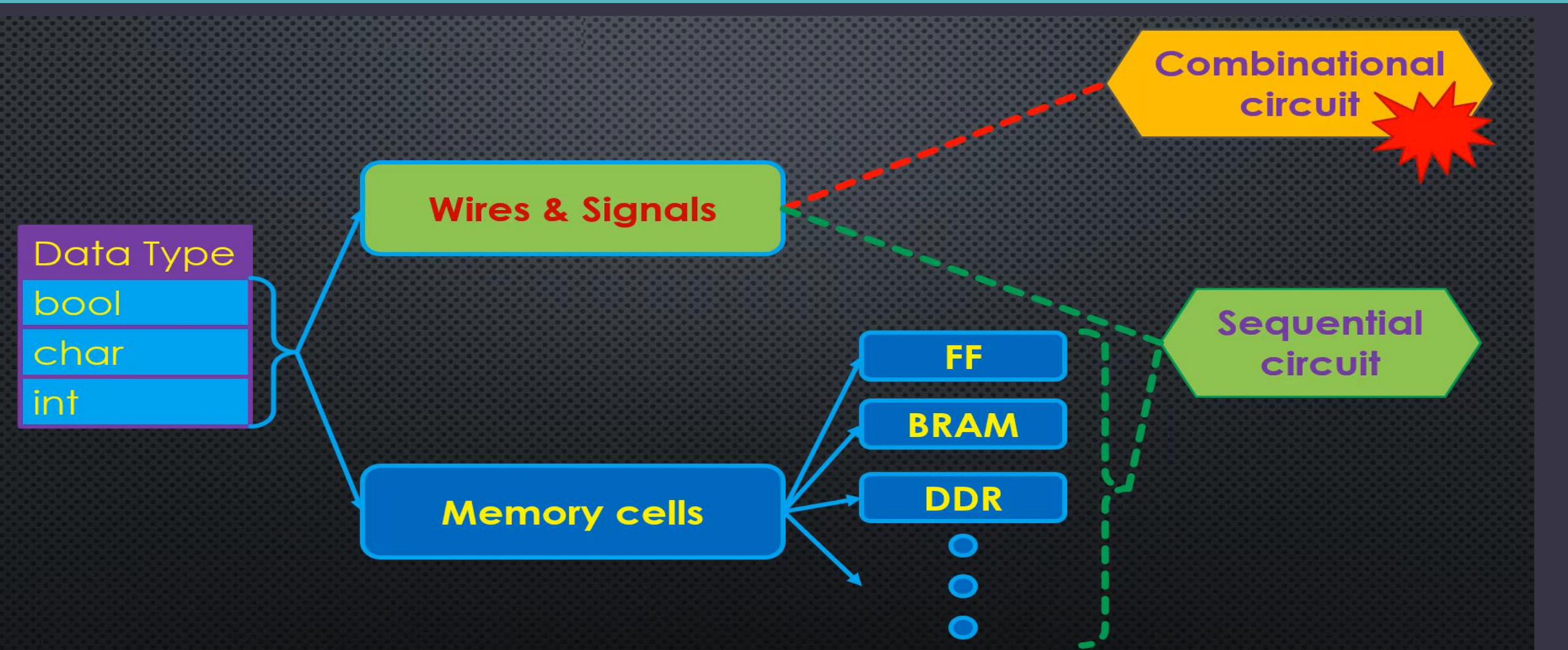
Write the corresponding bit-width in this table

Data Type	Bit-Width
short int	
int	
long int	
long long int	

SYNTHESIS

Lecture – 4-Data type

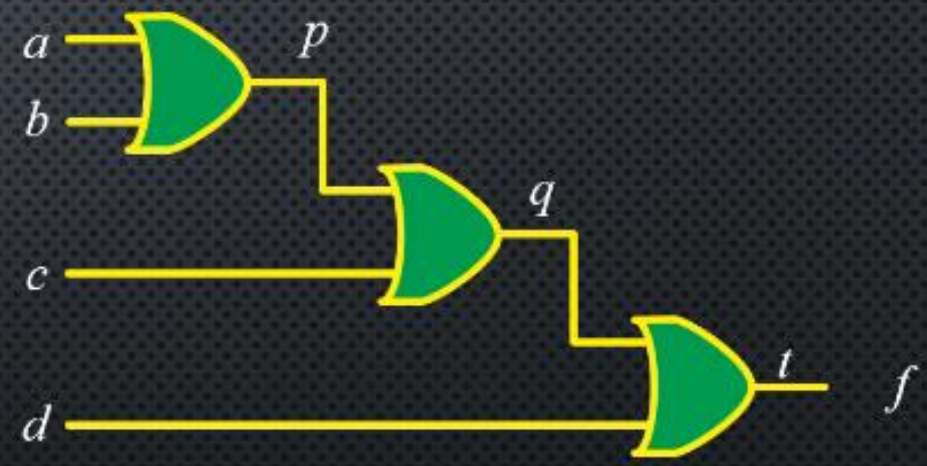
DATA TYPE SYNTHESIS



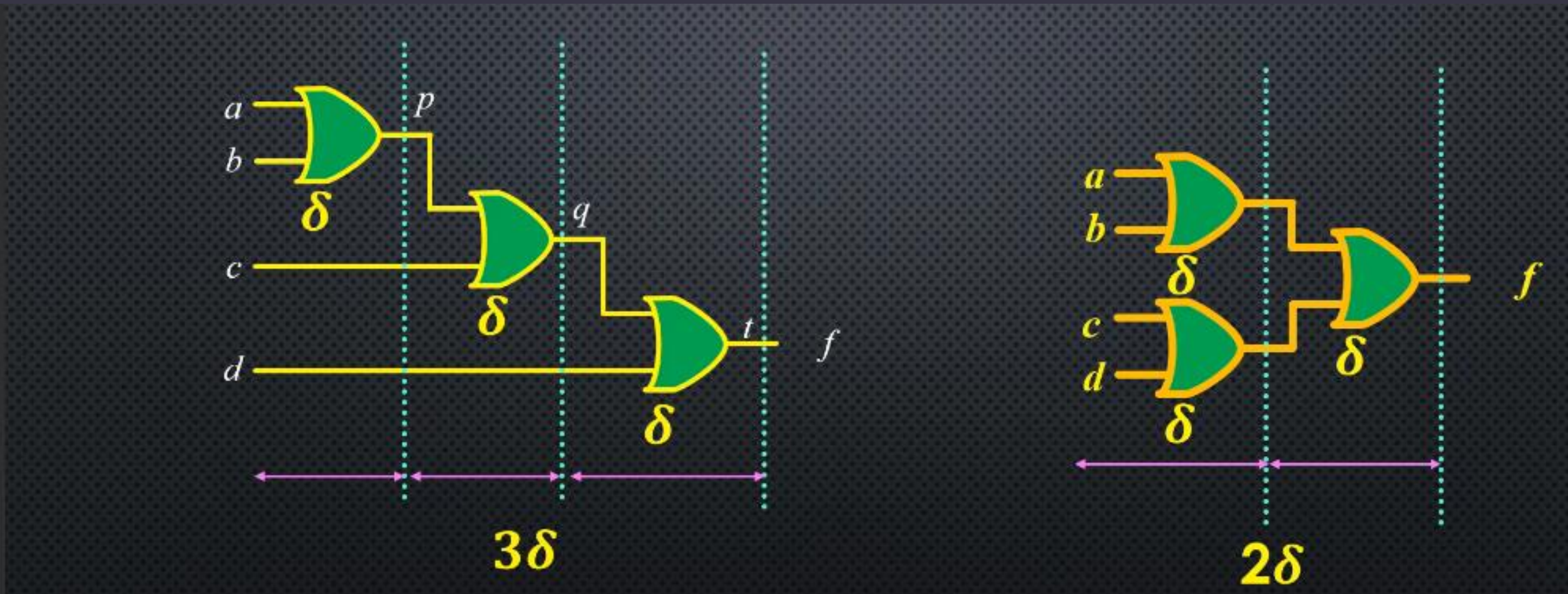
DATA TYPE SYNTHESIS EXAMPLE

$$f = a \text{ OR } b \text{ OR } c \text{ OR } d;$$

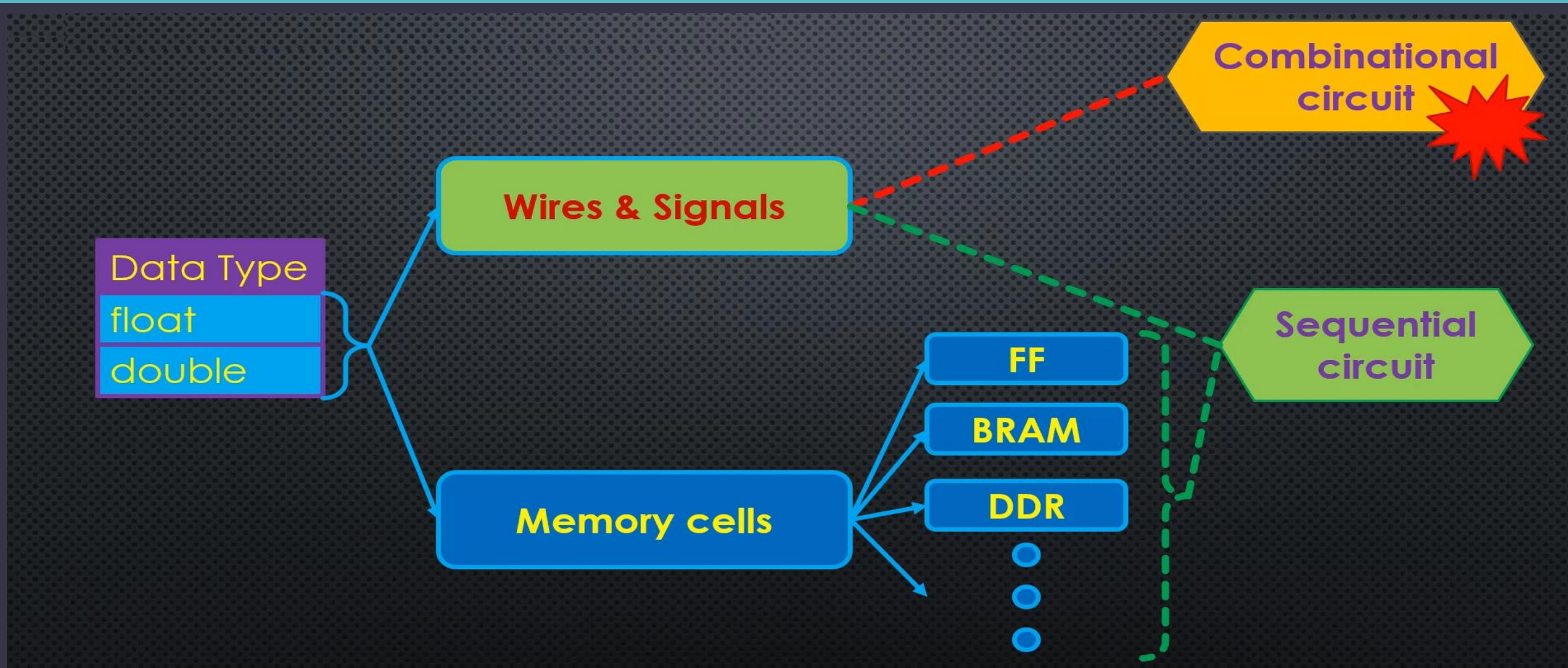
```
void or4(bool a, bool b,bool c, bool d, bool *f ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=a
#pragma HLS INTERFACE ap_none port=b
#pragma HLS INTERFACE ap_none port=c
#pragma HLS INTERFACE ap_none port=d
#pragma HLS INTERFACE ap_none port=f

  bool p = a | b;
  bool q = p | c;
  bool t = q | d;

  *f = t;
}
```



DATA TYPE SYNTHESIS EXAMPLES OPTIMIZATION



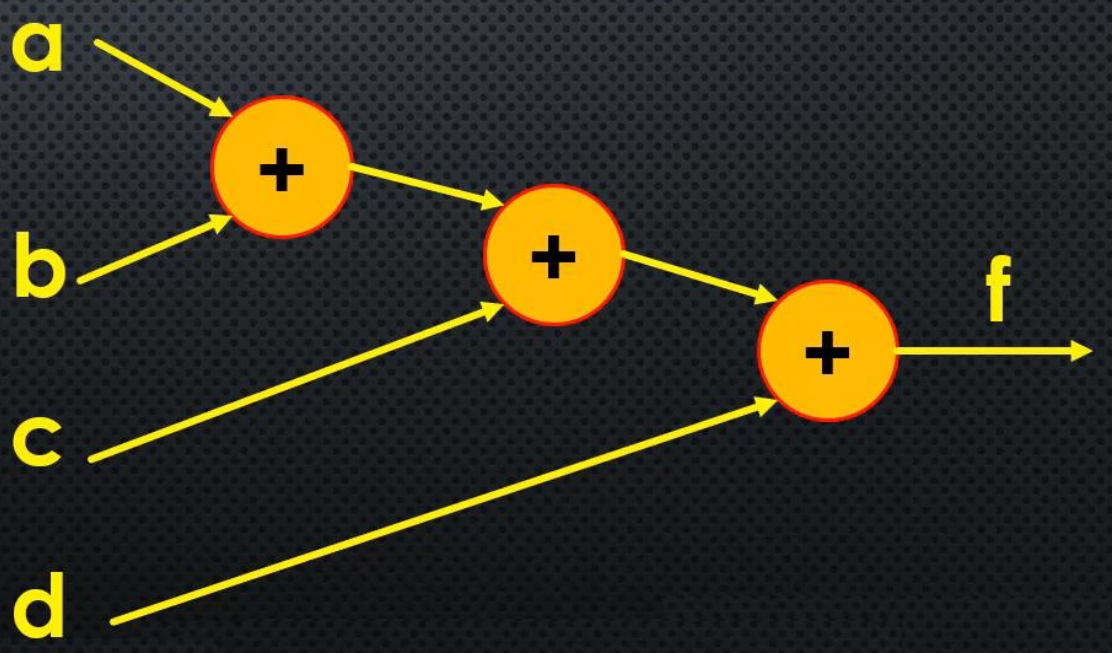
FLOAT DATA TYPES



FLOAT TYPE SYNTHESIS EXAMPLE

$$f = a + b + c + d;$$

```
void addition4(float a, float b, float c, float d, float *f ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=a
#pragma HLS INTERFACE ap_none port=b
#pragma HLS INTERFACE ap_none port=c
#pragma HLS INTERFACE ap_none port=d
#pragma HLS INTERFACE ap_none port=f

  float p = a + b;
  float q = p + c;
  float t = q + d;

  *f = t;
}
```



SUMMARY

- The HLS optimisation process may eliminate variables in a code or create new intermediate variables.
- The remaining variables after optimisation should be mapped to real hardware elements. A synthesis tool usually implements a variable by a bunch of wires or memories.

ASSIGNMENT Q

Find the dataflow graph of the following function after synthesis.

```
void or5(bool a, bool b,bool c, bool d, bool e, bool &f ) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=a
#pragma HLS INTERFACE ap_none port=b
#pragma HLS INTERFACE ap_none port=c
#pragma HLS INTERFACE ap_none port=d
#pragma HLS INTERFACE ap_none port=f

  f = a | b | c | d | e;
}
```

Any Question...

Thank you