L'IMPACT DES PHÉNOMÈNES PHYSIQUES DANS LA CYBERSÉCURITÉ

INTRODUCTION

Une pièce contient une ampoule éteinte, vous êtes à l'extérieur de cette pièce avec trois interrupteurs sur position « off ». Vous pouvez manipuler comme vous le souhaitez les interrupteurs mais vous ne pouvez entrer qu'une seule fois dans la pièce. Comment déterminer quel interrupteur allume l'ampoule ?

INTRODUCTION

a

b

a : https://pixabay.com/fr/vectors/chronographe-chronom%C3%A8tre-horloge-160753/
b : https://freesvg.org/1445945222

SOMMAIRE

- Les problèmes liés aux algorithmes :
    - Du code pin
    - De la mise en cache
- Une solution : le constant time
- Les limites du constant time
- Le code pin via de nouvelles approches

ALGORITHME CODE PIN

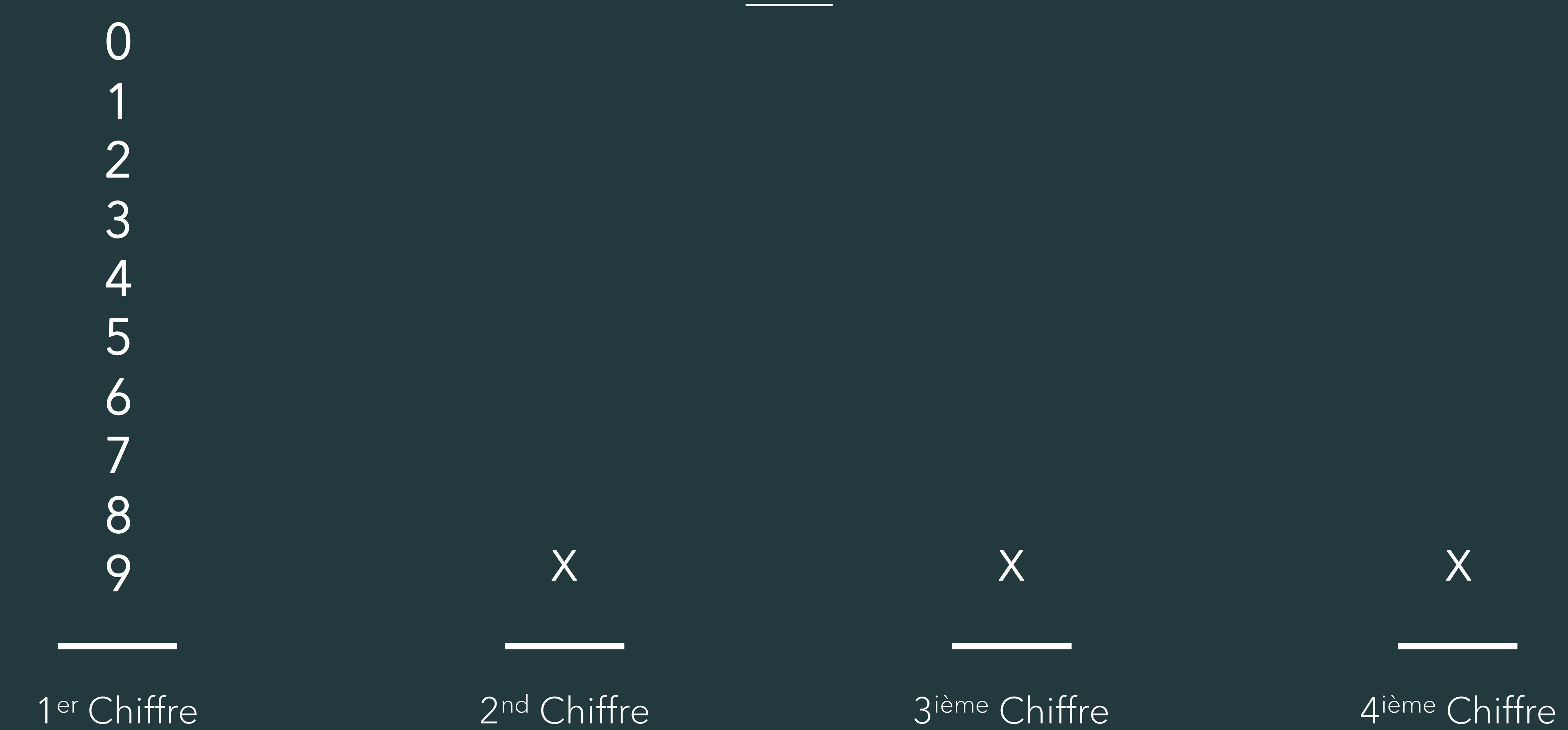
```
def code_bon(code, cle):
  for i in range(4):
    if code[i]!= cle[i]:
      return False
  return True
```

```
import time

def code_bon(code, cle):
  for i in range(4):
    if code[i]!= cle[i]:
      return False
  return True

tab = []
for i in range(1000):
  cpt=0
  for j in range(10000):
    start=time.process_time_ns()
    code_bon("1234","1234")
    end=time.process_time_ns()
    t1=(end-start)
    start=time.process_time_ns()
    code_bon("4234","1234")
    end=time.process_time_ns()
    t2=(end-start)
    if t1>t2:
      cpt+=1
  tab.append(cpt/100)

print(sum(tab)/len(tab))
```

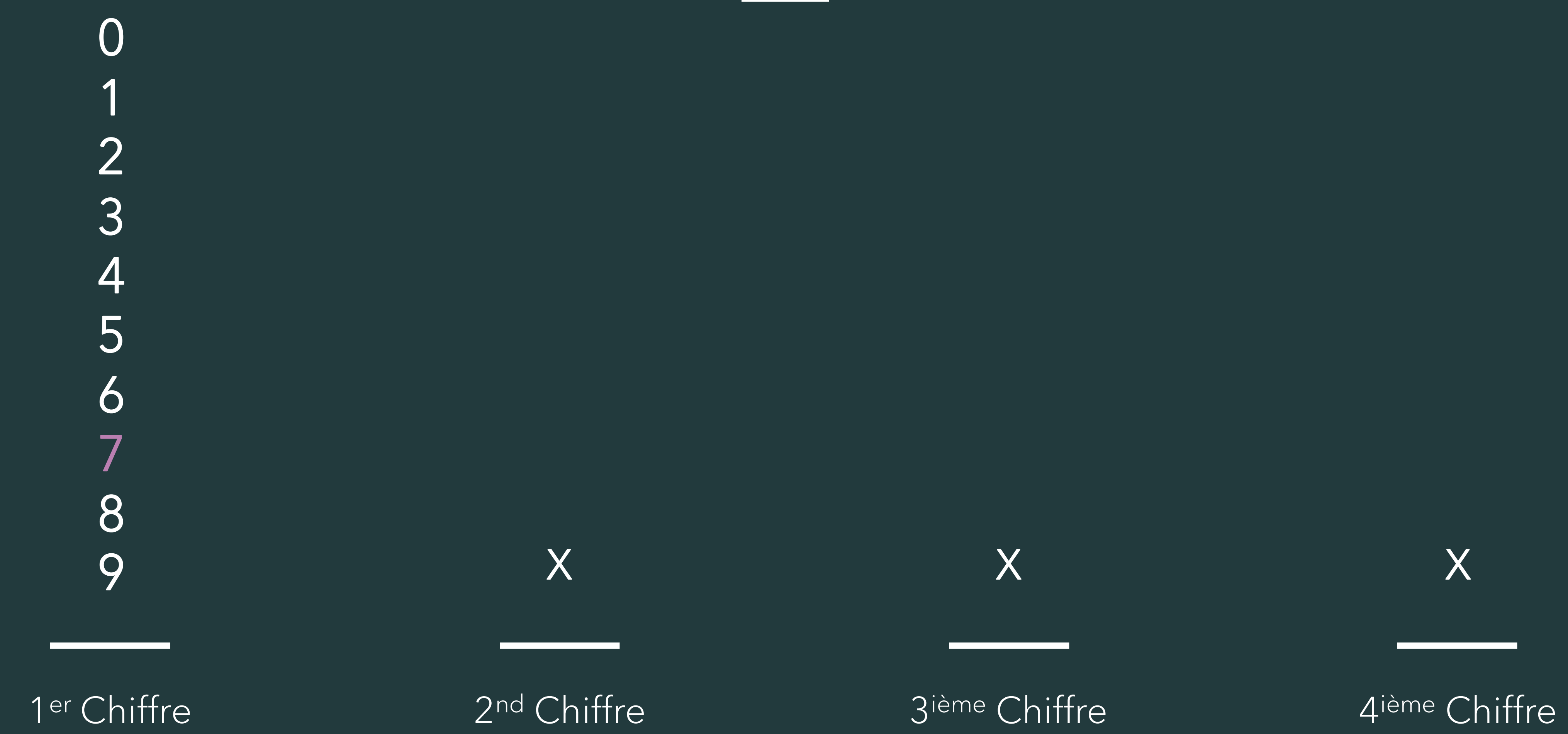
ALGORITHME CODE PIN

Place du chiffre faux	1ère	2ième	3ième
Pourcentage d'arrêt plus rapide	99,4%	99,2%	98,2%
Écart-type de la série	0,28	0,44	1,50

ALGORITHME CODE PIN



ALGORITHME CODE PIN

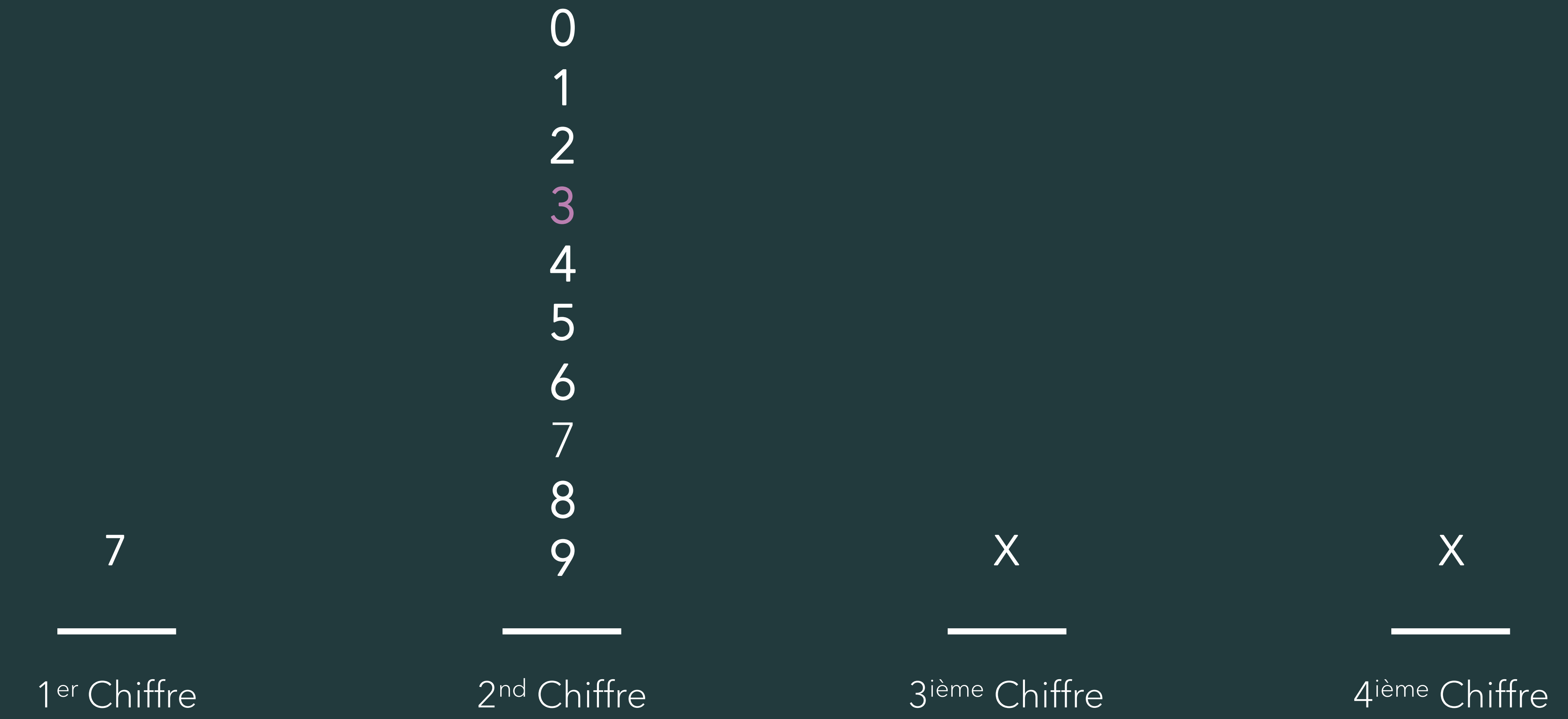


ALGORITHME CODE PIN

0
1
2
3
4
5
6
7
8
9

7	0–9	X	X
1er Chiffre	2nd Chiffre	$3^{ième}$ Chiffre	$4^{ième}$ Chiffre

ALGORITHME CODE PIN



ACCÈS AU CACHE

```
def verification_clef(tableau, clef, secret):
  a = tableau[secret]
  b = tableau[clef]
  if a == b:
    return True
  elif a!= b:
    return False
```

```
import time
import random

def verification_clef(tableau, clef, secret):
  a = tableau[secret]
  b = tableau[clef]
  if a == b:
    return True
  elif a!= b:
    return False

t = []

for i in range(10000):
  tab = [random.random() for j in range (1000)]
  tab_temps = []
  valeur_secrete = 614
  chiffrement_secret = tab[valeur_secrete]

  for j in range(1000):
    start = time.process_time_ns()
    verification_clef(tab,j,valeur_secrete)
    end = time.process_time_ns()
    tab_temps.append(end-start)

  cpt = 0
  for j in range (len(tab_temps)):
    if tab_temps[valeur_secrete] < tab_temps[j]:
      cpt+=1
  t.append(cpt)

for i in range (len(t)):
  t[i] = t[i]/1000*100
moy = sum(t)/len(t)
```

ACCÈS AU CACHE

Pourcentage d'arrêt plus rapide	Écart-type de la série
66,2%	28,1

La mise en cache étant une action qui a très peu d'influence sur ce type de petit programme, la différence est minime et difficile à déterminer sans un matériel plus précis ce qui explique l'écart-type important.

LES ALGORITHMES CONSTANT TIME

Définition : En cryptographie, un programme est dit constant time si et seulement si son temps d'exécution ne varie pas en fonction de la valeur secrète de celui-ci. Il y a donc une indépendance entre le secret et le temps.

Pas de valeurs secrètes :

- Dans les boucles for et while
- Dans les instructions if, then et else
- Pour accéder à une case d'un tableau

LE CODE PIN EN CONSTANT TIME

```c
#include <stdbool.h>

bool code_pin(int *input, int *pin)
{
    for (int i = 0; i < 4; i++)
    {
        if (pin[i] != input[i])
        {
            return false;
        }
    }
    return true;
}
```

```c
#include <stdbool.h>

bool code_pin(int *input, int *pin)
{
    bool error = false;
    for (int i = 0; i < 4; i++)
    {
        error = (pin[i] != input[i]) ? true : error;
    }
    return error;
}
```

L'ACCÈS AU CACHE EN CONSTANT TIME

```c
#include <stdbool.h>

bool verify_fey(int *tab, int key, int input)
{
    int v1 = tab[key];
    int v2 = tab[input];
    if (v1 == v2)
    {
        return true;
    }
    else if (v1 != v2)
    {
        return false;
    }
}
```

```c
#include <stdbool.h>

bool verify_key(int *tab, int key, int input)
{
    bool same = (tab[key] == tab[input]) ? true : false;
    return same;
}
```

LES LIMITES DU CONSTANT TIME

```c
int main()
{
    int cpt = 0;
    for (int i = 0; i < 10; i++)
    {
        cpt += i + 1;
    }
    return cpt;
}
```

LES LIMITES DU CONSTANT TIME

```
	.file	"somme.c"
	.text
	.def	__main;	.scl	2;	.type	32;	.endef
	.section	.text.startup,"x"
	.p2align 4
	.globl	main
	.def	main;	.scl	2;	.type	32;	.endef
	.seh_proc	main


main:
	subq	$40, %rsp
	.seh_stackalloc	40
	.seh_endprologue
	call	__main
	movl	$55, %eax
	addq	$40, %rsp
	ret
	.seh_endproc
	.ident	"GCC: (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders) 11.2.0"
```

LES LIMITES DU CONSTANT TIME

```c
int main()
{
    int cpt = 0;
    for (int i = 0; i < 10; i++)
    {
        cpt += i + 1;
    }
    return cpt;
}
```

```asm
main:
    subq    $40, %rsp
    .seh_stackalloc 40
    .seh_endprologue
    call    __main
    movl    $55, %eax
    addq    $40, %rsp
    ret
```

Compilé avec -O3

LES LIMITES DU CONSTANT TIME

Le compilateur peut ne pas conserver du constant time par exemple :

```
int mask = -b;
x = (y & mask) | (x & ~mask);
```

Est constant time en temps normal mais si l'on compile avec clang version 5.0 et les drapeaux –O2 –m32 –march=i686, ce code est équivalent à :

```
if (b) x=y
```

Qui n'est plus constant time.

https://hal.archives-ouvertes.fr/hal-01959560

LE CODE PIN VIA DE NOUVELLES APPROCHES

Le code pin avec ordre de vérification aléatoire

Des implémentations physique possibles

ALGORITHME CODE PIN AVEC VÉRIFICATION ALÉATOIRE

```
def code_pin_random(key, input):
    order_to_verify=[0,1,2,3]
    random.shuffle(order_to_verify)
    for i in range(4):
        if (key[order_to_verify[i]]!=input[order_to_verify[i]]):
            return False
    return True
```

```
import random
import time


def code_pin_random(key, input):
    order_to_verify=[0,1,2,3]
    random.shuffle(order_to_verify)
    for i in range(4):
        if (key[order_to_verify[i]]!=input[order_to_verify[i]]):
            return False
    return True

tab = []
for i in range(1000):
  cpt=0
  for j in range(10000):
    t1,t2=0,0
    for k in range(21):
      start=time.process_time_ns()
      code_pin_random("1234","1111")
      end=time.process_time_ns()
      t1+=(end-start)
      start=time.process_time_ns()
      code_pin_random("1234","9999")
      end=time.process_time_ns()
      t2+=(end-start)
    if t1>t2:
      cpt+=1
  tab.append(cpt/100)

moy = sum(tab)/len(tab)
print(moy)
```

ALGORITHME CODE PIN AVEC VÉRIFICATION ALÉATOIRE

Pourcentage d’arrêt plus rapide pour des chiffres non présents	65,05%
Écart-type de la série	1,05

EN PRATIQUE

Pour n essais, la probabilité de tirer au moins une fois chaque chiffre est : $1 - Pn$ où Pn est la probabilité de ne pas tirer tous les chiffres au rang n.

EN PRATIQUE

Pour n essais, la probabilité de tirer au moins une fois chaque chiffre est : $1 - Pn$ où Pn est la probabilité de ne pas tirer tous les chiffres au rang n.

La probabilité de tirer un chiffre est : $\frac{1}{4}$ ainsi la probabilité de ne pas le tirer est : $\frac{3}{4}$

EN PRATIQUE

Pour n essais, la probabilité de tirer au moins une fois chaque chiffre est : $1 - Pn$ où Pn est la probabilité de ne pas tirer tous les chiffres au rang n.

La probabilité de tirer un chiffre est : $\frac{1}{4}$ ainsi la probabilité de ne pas le tirer est : $\frac{3}{4}$

La probabilité de ne pas le tirer jusqu'au rang n est alors : $(\frac{3}{4})^n$

Ainsi $Pn = 4 \times (\frac{3}{4})^n$ avec $n > 4$

EN PRATIQUE

Pour n essais, la probabilité de tirer au moins une fois chaque chiffre est : $1 - Pn$ où Pn est la probabilité de ne pas tirer tous les chiffres au rang n.

La probabilité de tirer un chiffre est : $\frac{1}{4}$ ainsi la probabilité de ne pas le tirer est : $\frac{3}{4}$

La probabilité de ne pas le tirer jusqu'au rang n est alors : $(\frac{3}{4})^n$

Ainsi $Pn = 4 \times (\frac{3}{4})^n$ avec $n > 4$

Si l'on souhaite $1 - Pn > 0{,}99$, il faut $Pn < 0{,}01$ soit $n > 20$ ($1 - P20 = 0{,}0127$ et $1 - P21 = 0{,}0095$)

On prend alors $n = 21$

EN PRATIQUE

```python
import random
import time

code=str(random.randint(1000,9999))

def code_pin_random(key, input):
    order_to_verify=[0,1,2,3]
    random.shuffle(order_to_verify)
    for i in range(4):
        if (key[order_to_verify[i]]!=input[order_to_verify[i]]):
            return False
    return True

def indice_max(l, cpt=0):
  if cpt == 4 :
    return[]
  l2 = l.copy()
  m=l[0]
  indice = 0
  for i in range (len(l)):
    if l[i]>m:
      m=l[i]
      indice = i
  l2[indice]=0
  return [indice]+indice_max(l2,cpt+1)
```

```python
t = [0 for i in range(10)]
for k in range (700):
  tab = []
  for i in range(10):
      l=[]
      input=str(i)+str(i)+str(i)+str(i)
      for j in range(21):
          start=time.process_time_ns()
          code_pin_random(code,input)
          end=time.process_time_ns()
          l.append(end-start)
      tab.append(sum(l))
  indice = indice_max(tab)
  for i in indice:
    t[i]+=1

resultat = []
while len(resultat)<4:
  i = indice_max(t)
  if t[i[0]]>670:
    resultat+=[i[0],i[0],i[0],i[0]]
  elif t[i[0]]>620:
    resultat+=[i[0],i[0],i[0]]
  elif t[i[0]]>500:
    resultat+=[i[0],i[0]]
  else:
    resultat.append(i[0])
  t[i[0]]=0
print(resultat)
print(code)
```

EN PRATIQUE

	Brute force	Analyse du temps d'exécution
Nombre d'essais	10 000	147 024
Pourcentage de code trouvé	100%	92,2%

$$700 \times 10 \times 21 + 4! = 147\,024$$

EN PRATIQUE

```python
import random
import time

code=str(random.randint(100000,999999))

def code_pin_random(key, input):
    order_to_verify=[0,1,2,3,4,5]
    random.shuffle(order_to_verify)
    for i in range(6):
        if (key[order_to_verify[i]]!=input[order_to_verify[i]]):
            return False
    return True

def indice_max(l, cpt=0):
  if cpt == 6 :
    return[]
  l2 = l.copy()
  m=l[0]
  indice = 0
  for i in range (len(l)):
    if l[i]>m:
      m=l[i]
      indice = i
  l2[indice]=0
  return [indice]+indice_max(l2,cpt+1)
```

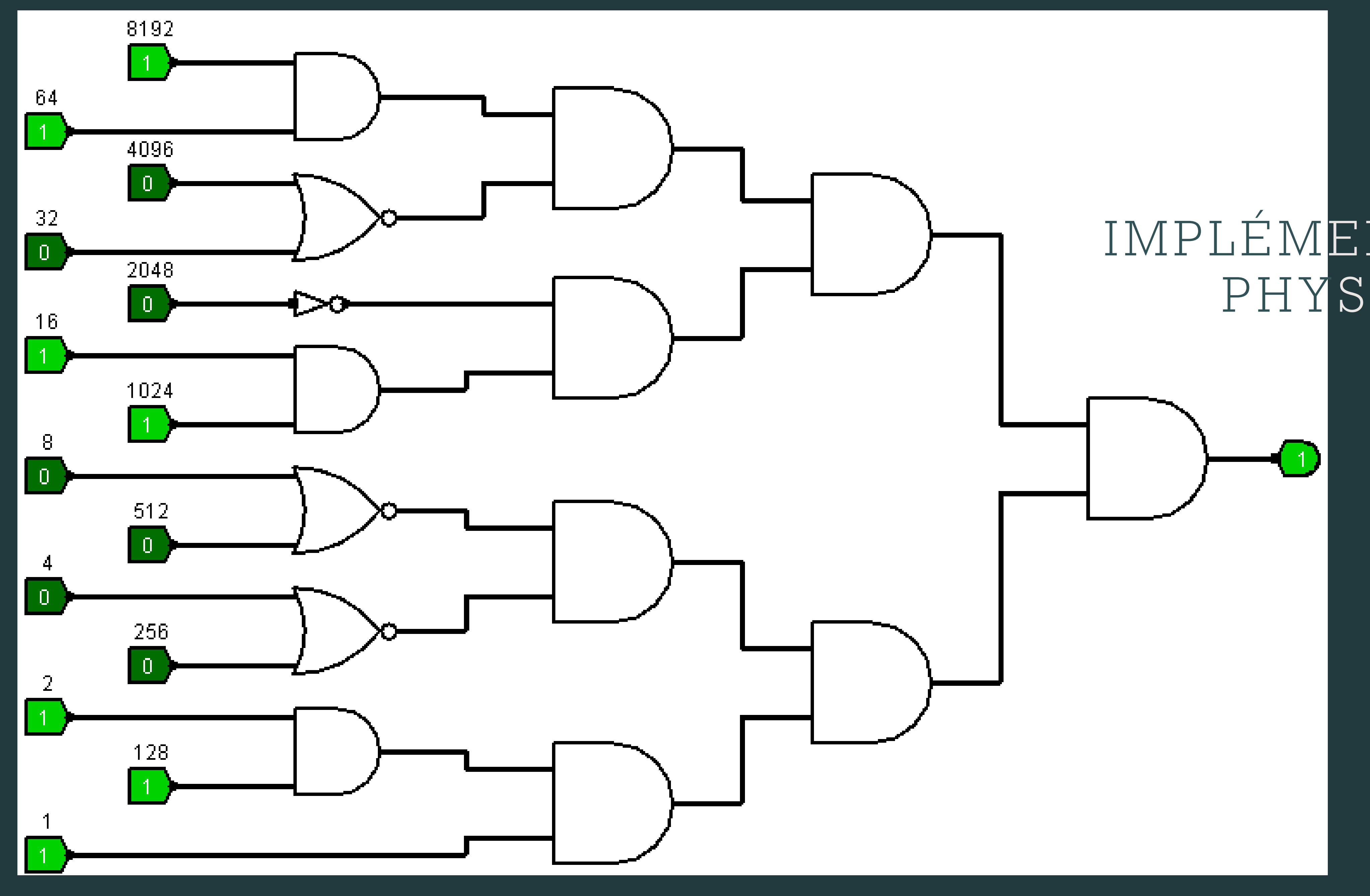
```python
t = [0 for i in range(10)]
for k in range (1500):
  tab = []
  for i in range(10):
      l=[]
      input=str(i)+str(i)+str(i)+str(i)+str(i)+str(i)
      for j in range(21):
          start=time.process_time_ns()
          code_pin_random(code,input)
          end=time.process_time_ns()
          l.append(end-start)
      tab.append(sum(l))
  indice = indice_max(tab)
  for i in indice:
    t[i]+=1
resultat = []
while len(resultat)<6:
  i = indice_max(t)
  if t[i[0]]>1490:
    resultat+=[i[0],i[0],i[0],i[0],i[0],i[0]]
  elif t[i[0]]>1475:
    resultat+=[i[0],i[0],i[0],i[0],i[0]]
  elif t[i[0]]>1400:
    resultat+=[i[0],i[0],i[0],i[0]]
  elif t[i[0]]>1350:
    resultat+=[i[0],i[0],i[0]]
  elif t[i[0]]>1130:
    resultat+=[i[0],i[0]]
  else:
    resultat.append(i[0])
  t[i[0]]=0
print(resultat)
```

EN PRATIQUE

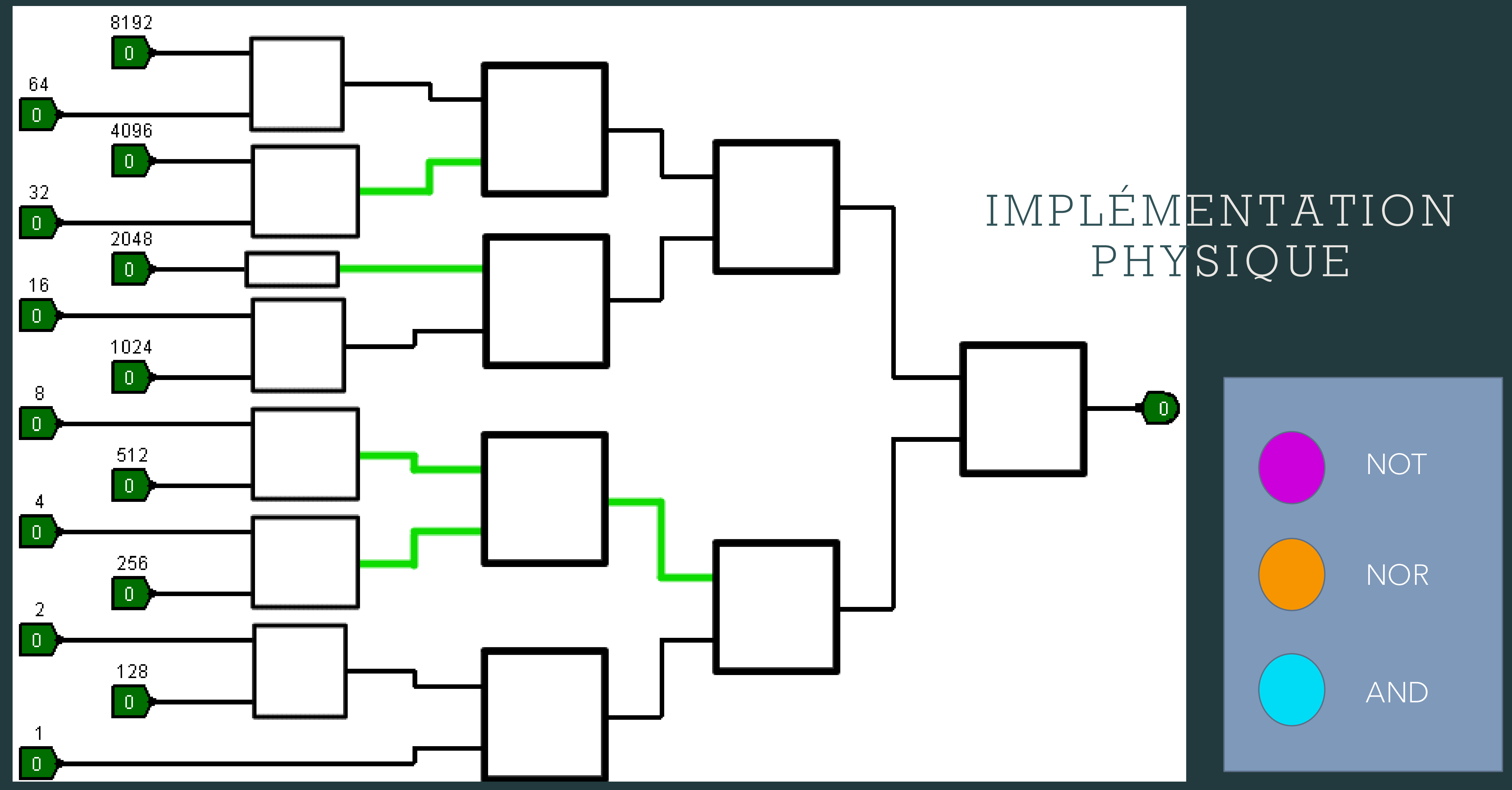
	Brute force	Analyse du temps d'exécution
Nombre d'essais	1 000 000	315 720
Pourcentage de code trouvé	100%	85,3%

$$1500 \times 10 \times 21 + 6! = 315\,720$$

IMPLÉMENTATION PHYSIQUE



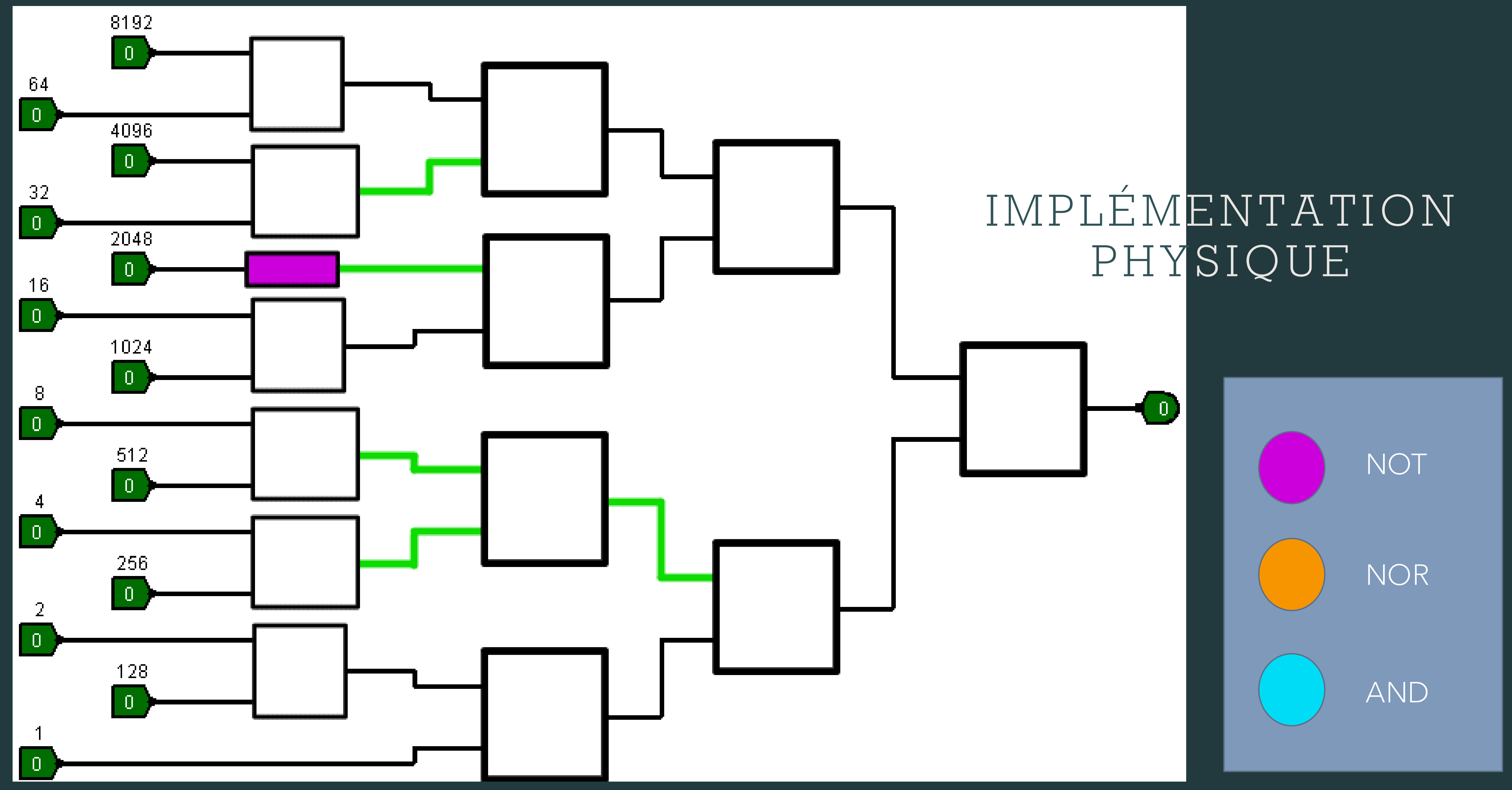
IMPLÉMENTATION PHYSIQUE

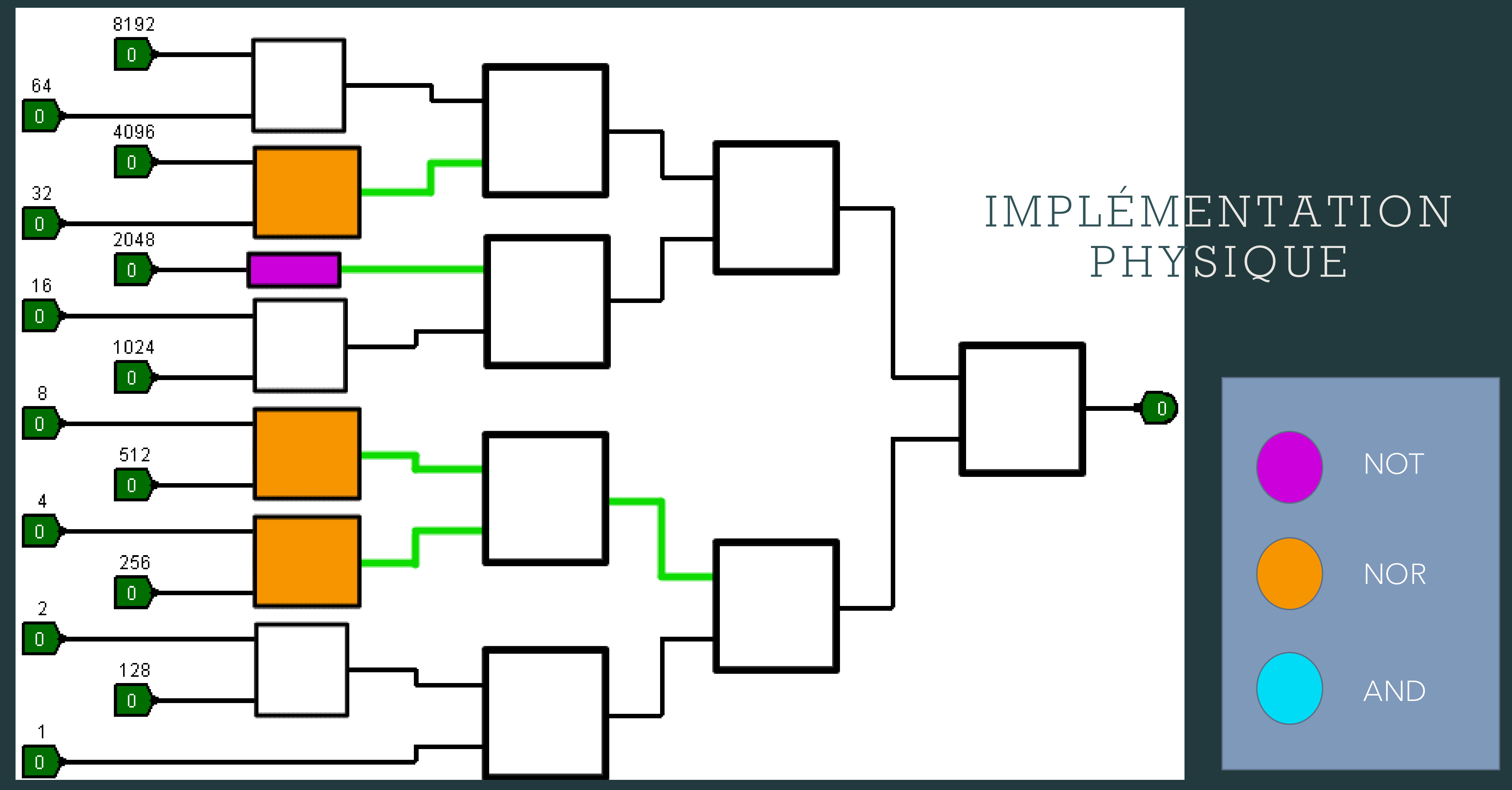


NOT

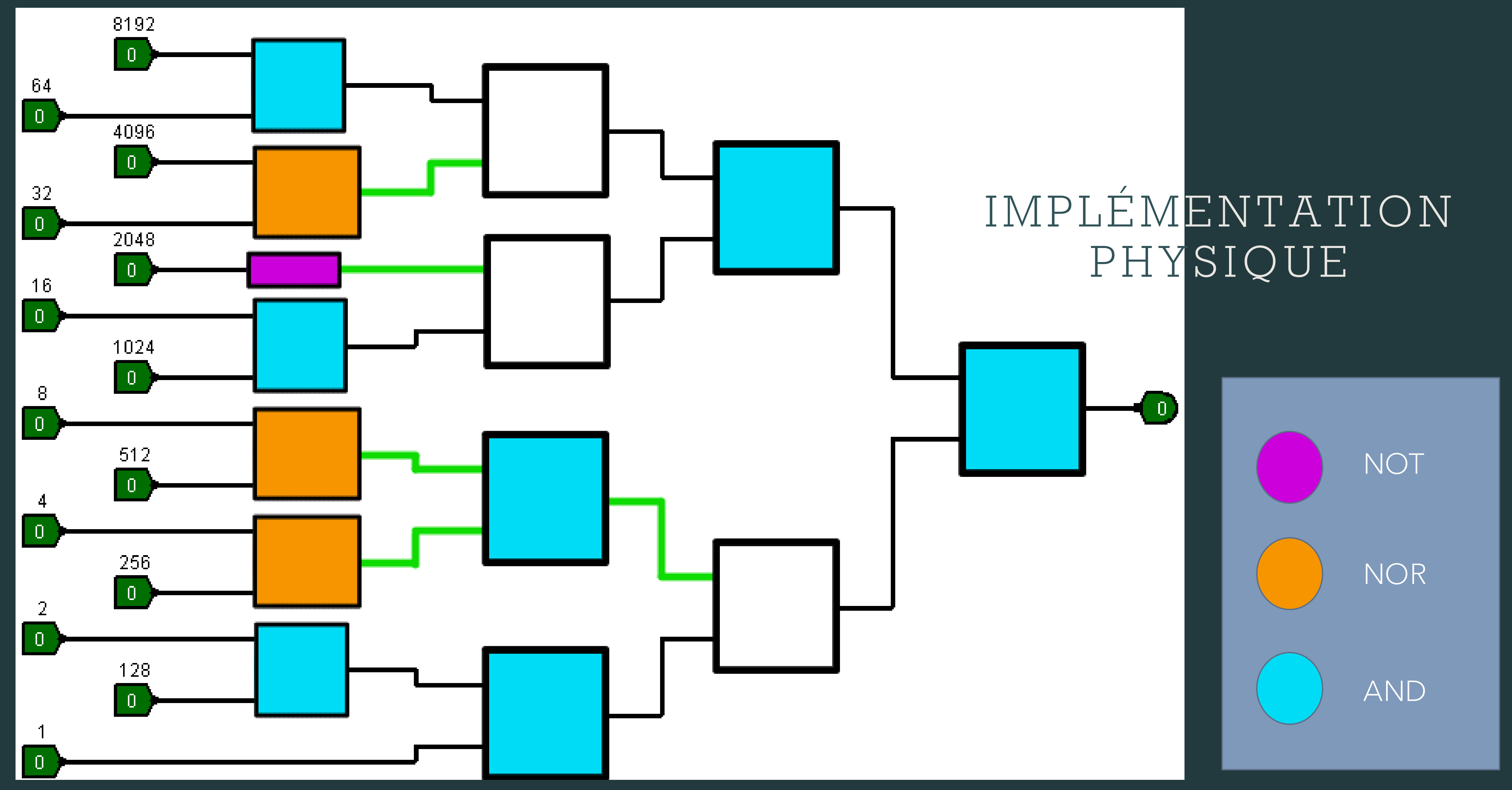
NOR

AND

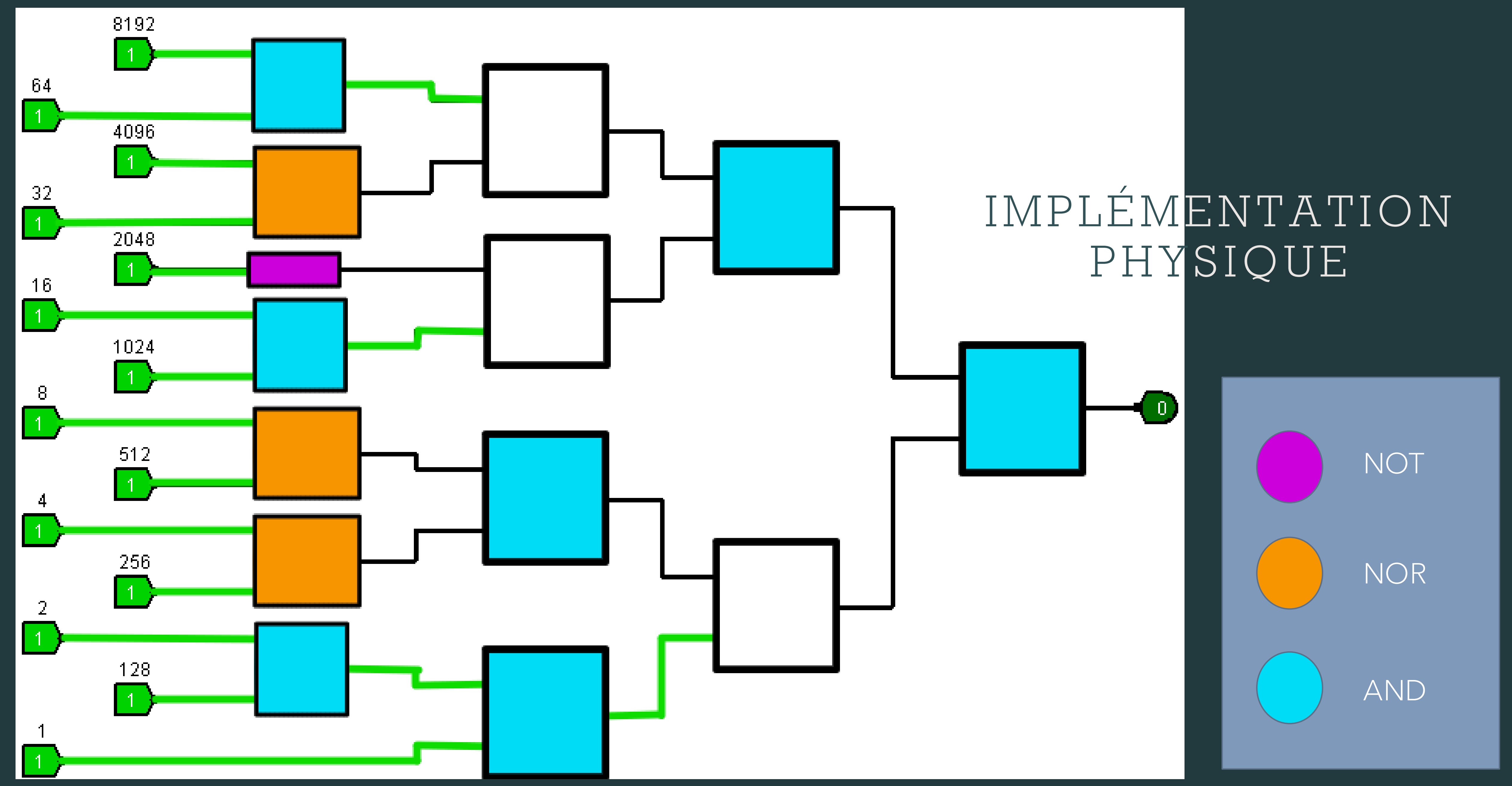

IMPLÉMENTATION PHYSIQUE
8192
64
4096
32
2048
16
1024
8
512
4
256
2
128
1
0
NOT
NOR
AND

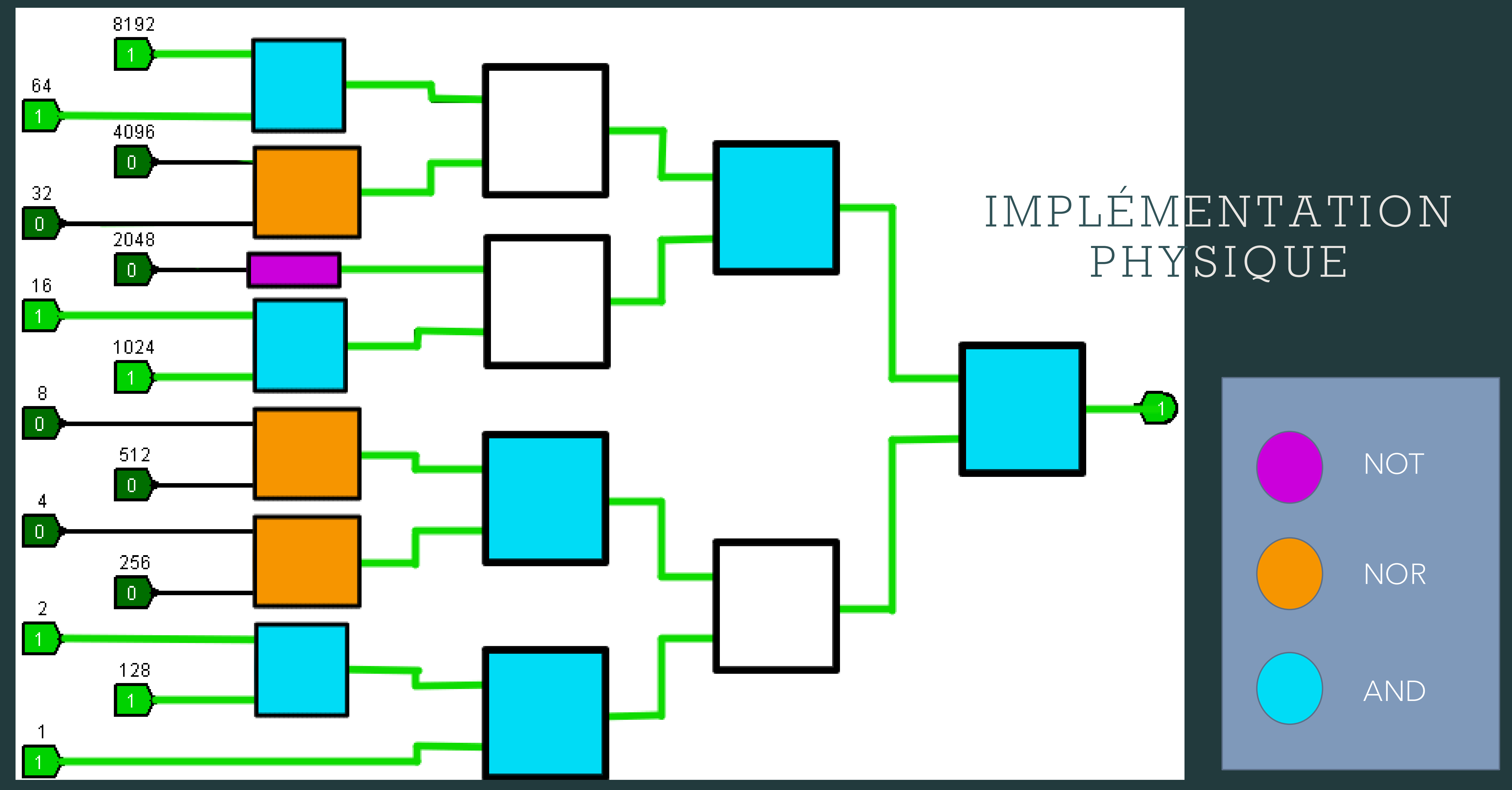


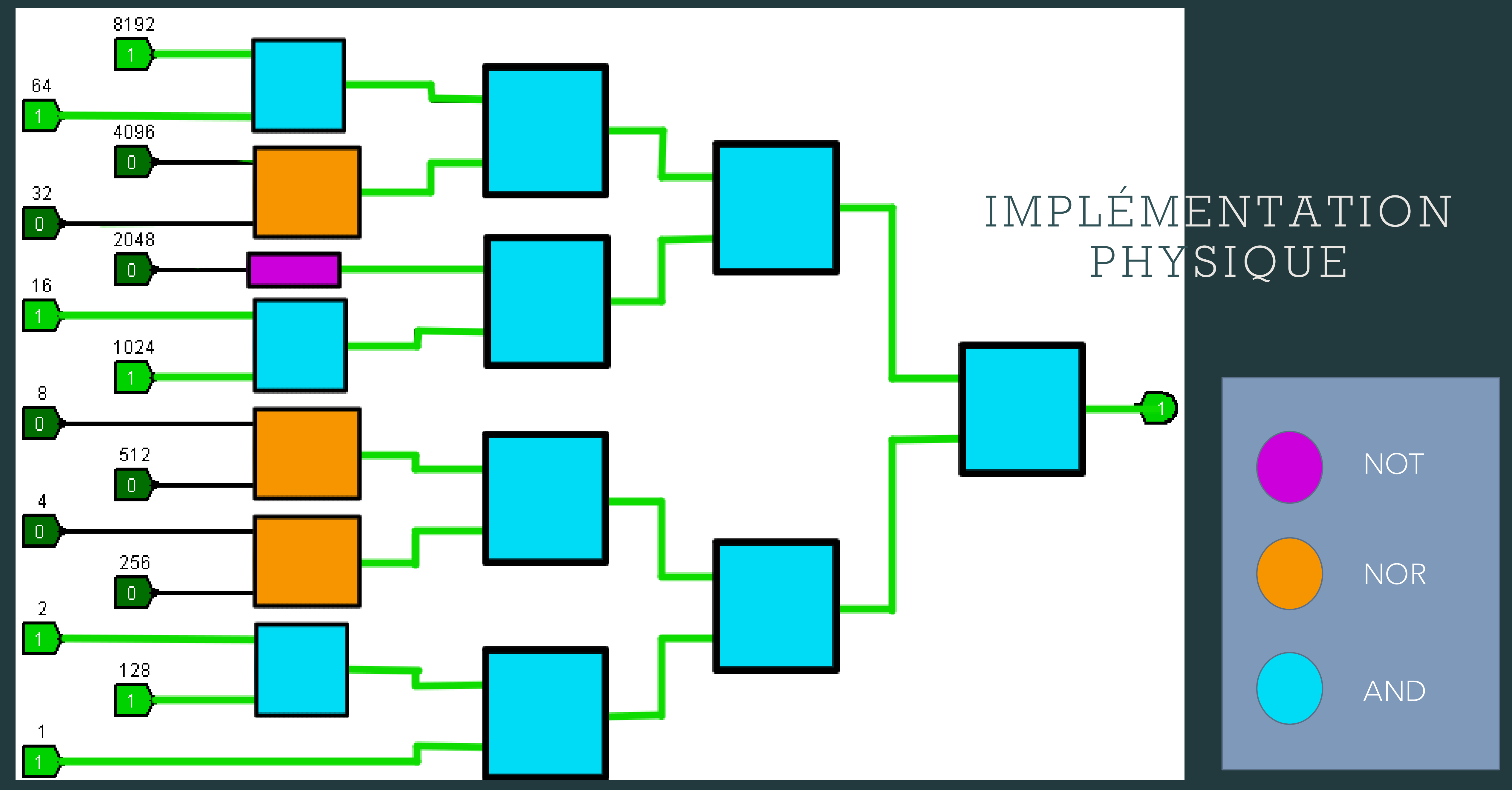
IMPLÉMENTATION PHYSIQUE


IMPLÉMENTATION PHYSIQUE
8192
64
4096
32
2048
16
1024
8
512
4
256
2
128
1
0
NOT
NOR
AND

IMPLÉMENTATION PHYSIQUE



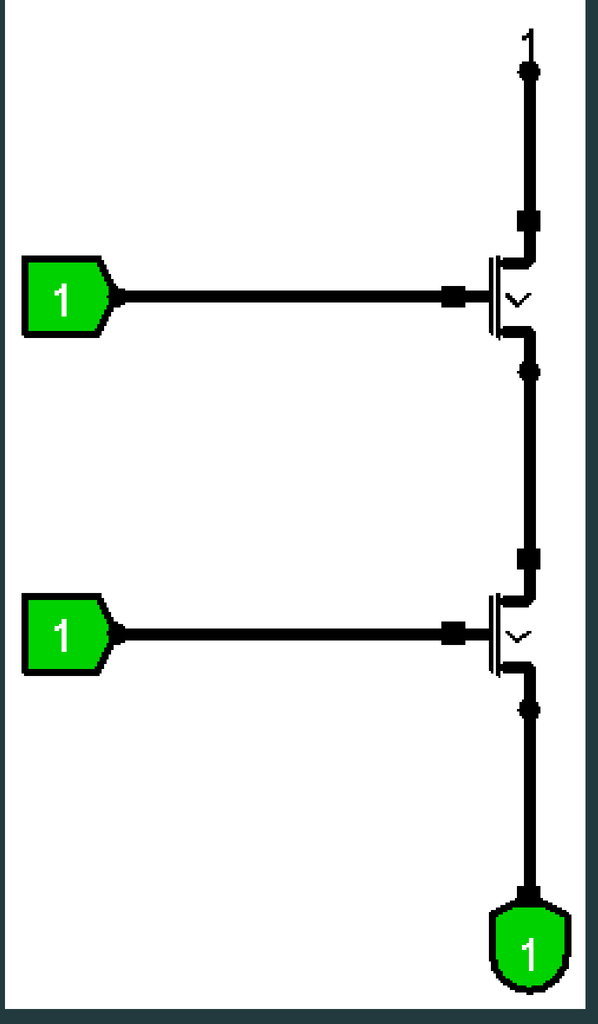

IMPLÉMENTATION
PHYSIQUE
8192
1
64
1
4096
0
32
0
2048
0
16
1
1024
1
8
0
512
0
4
0
256
0
2
1
128
1
1
1
1
NOT
NOR
AND



IMPLÉMENTATION PHYSIQUE
8192
1
64
1
4096
0
32
0
2048
0
16
1
1024
1
8
0
512
0
4
0
256
0
2
1
128
1
1
1
1
NOT
NOR
AND

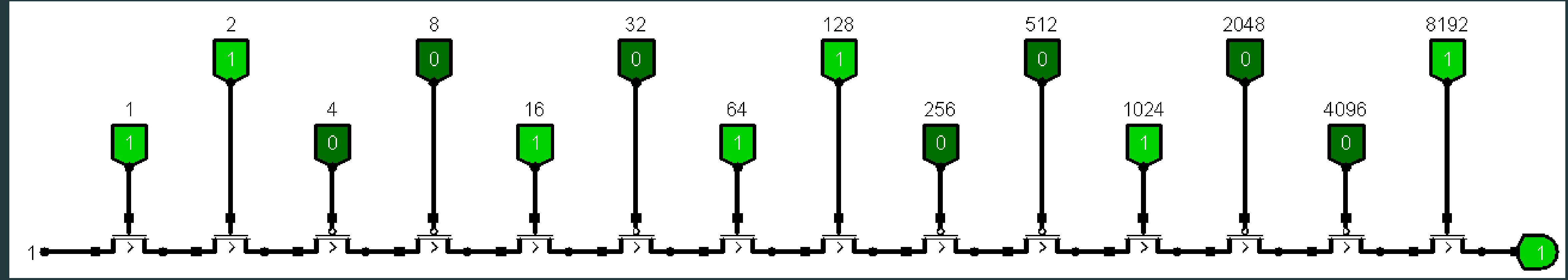
IMPLÉMENTATION PHYSIQUE

Porte NOR



Porte AND

IMPLÉMENTATION PHYSIQUE



CONCLUSION

Ces deux implémentations ne sont pas efficaces pour sécuriser ce type de programme. La solution est alors les algorithmes constant time puis vérifier que le code assembleur généré n'a pas perdu cette propriété à la compilation.