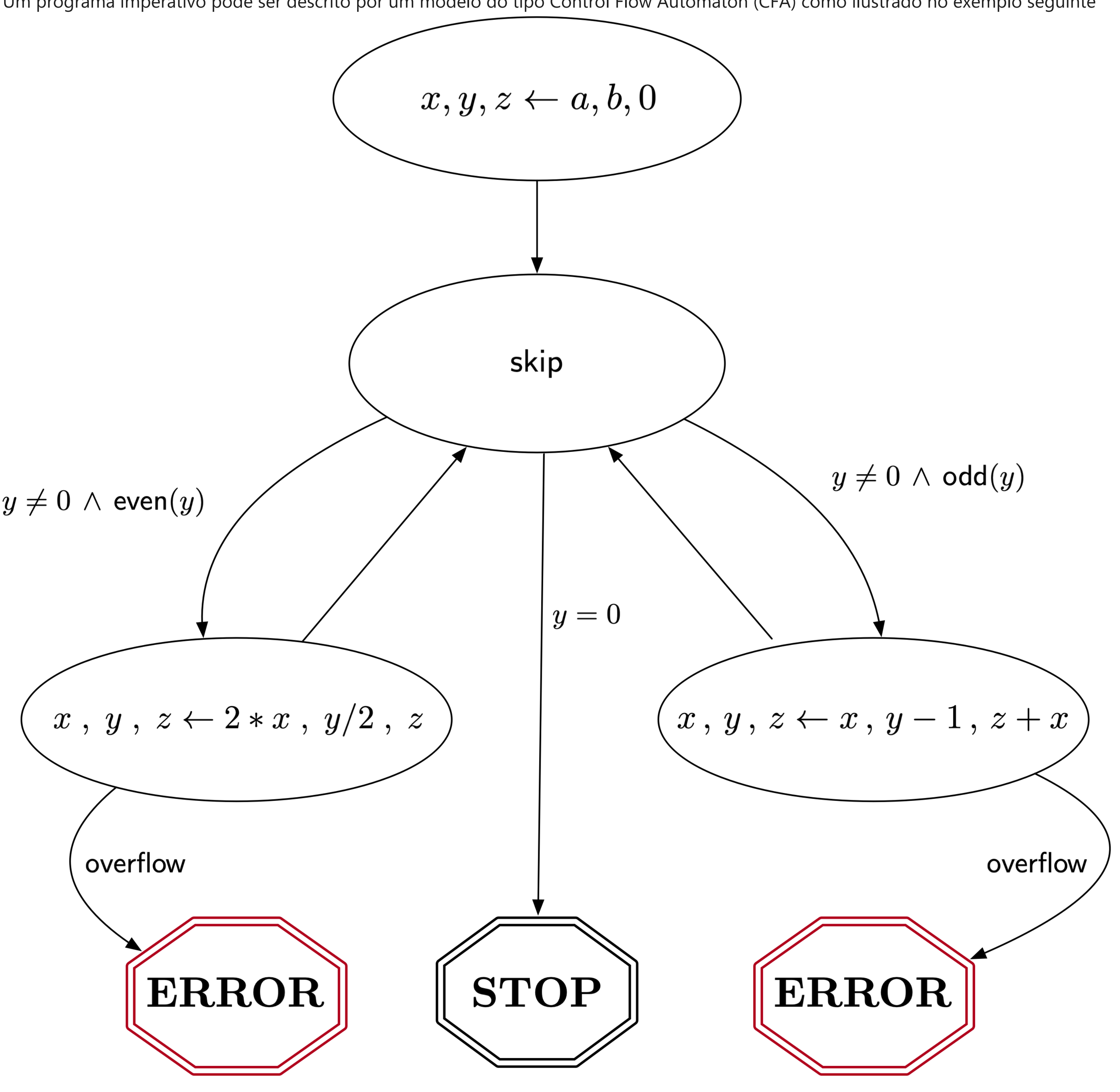
TP2 - Grupo 14

André Lucena Ribas Ferreira - A94956

Paulo André Alegre Pinto - A97391

Problema 1 - Control Flow Automaton

Um programa imperativo pode ser descrito por um modelo do tipo Control Flow Automaton (CFA) como ilustrado no exemplo seguinte



Este programa implementa a multiplicação de dois inteiros a, b , fornecidos como "input", e com precisão limitada a n bits (fornecido como parâmetro do programa). Note-se que

- Existe a possibilidade de alguma das operações do programa produzir um erro de "overflow".
- Os nós do grafo representam ações que actuam sobre os "inputs" do nó e produzem um "output" com as operações indicadas
- Os ramos do grafo representam ligações que transferem o "output" de um nodo para o "input" do nodo seguinte. Esta transferência é condicionada pela satisfação da condição associada ao ramo

Como 'inputs', o programa recebe:

1. O número de bits da precisão n, ao qual todas as variáveis do problema são limitados;
2. Os dois inteiros a se multiplicar, a e b.

Análise

Pretende-se modelar o autómato em cima descrito a partir de um FOTS usando BitVectors de tamanho n. Para tal, é necessário definir as variáveis do modelo, o predicado que define o estado inicial e a relação de transição.

O modelo terá 3 variáveis do tipo BitVector, x, y e z, este último que terminará com o resultado da execução. Para além destes, também terá uma variável inteira p que representa o estado de execução. Definiu-se um inteiro para cada um dos estados, nomeadamente:

- -1 para Estado de erro.
- 0 para Estado central (loop). Nele testam-se os casos.
- 1 para Estado de execução se y for par (não zero).
- 2 para Estado de execução se y for ímpar (não zero).
- 3 para Estado Final.

Decidiu-se manter-se o número de estados descritos no autómato de modo a evitar fazer comparações de erro quando estas não são necessárias.

Considerando o estado inicial, uma optimização possível à execução do autómato será manter considerar o valor b o menor dos dois. Isto diminui o número de operações totais necessárias, no pior caso. O estado inicial será então defindo pelo predicado seguinte.

$$p = 0 \land x = a \land y = b \land z = 0$$

Uma consideração a ter em conta para a transição é a de colocar condições opostas para a transição que efetua o cálculo e a que testa a condição de erro, já que partem ambas do mesmo estado. Outra será a de criar uma transição ora do estado -1 ora do 3 para si próprio, montando assim um *loop* já que esses estados são finais. A relação de transição entre dois estados s e s' será definido pelo seguinte predicado:

$$\begin{gathered}
(p = 0 \land y \equiv 0 \pmod 2 \land y \neq 0 \land p' = 1 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = 0 \land y \equiv 1 \pmod 2 \land y \neq 0 \land p' = 2 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = 1 \land 2x < 2^n \land p' = 0 \land x' = 2x \land y' = y/2 \land z' = z) \\
\lor \\
(p = 1 \land 2x \leq 2^n \land p' = -1 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = 2 \land 2^n - 1 - z \geq x \land p' = 0 \land x' = x \land y' = y - 1 \land z' = z + x) \\
\lor \\
(p = 2 \land 2^n - 1 - z < x \land p' = -1 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = 0 \land y = 0 \land p' = 3 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = -1 \land p' = -1 \land x' = x \land y' = y \land z' = z) \\
\lor \\
(p = 3 \land p' = 3 \land x' = x \land y' = y \land z' = z)
\end{gathered}$$

Um traço de execução é uma sequência de estados, onde dois estados consequentes validam um predicado de transição. Como o número de estados é finito, já que os valores que a e b podem tomar estão limitados pela precisão n, e porque as operações executadas tendem para um dos estados de *loop*, qualquer traço de execução deste problema é limitado. Dessa forma, pode-se sempre calcular o traço até ao momento em que um estado transiciona para outro que já ocorreu no traço, descrevendo assim um loop.

Por fim, pode-se verificar, para qualquer estado de um traço de execução, se o invariante $x * y + z = a * b$ é válido.

Implementação

Para a resolução do problema em questão, decidiu-se usar o módulo `pysmt.shortcuts`, com as funcionalidades possíveis para a utilização de um SMT Solver. Importam-se também os tipos deste Solver, a partir do módulo `pysmt.typing`. Para modelar este problema, irá usar-se BitVectors, então escolheu-se o `z3` como Solver.

Para além disso, é necessário importar o módulo `numpy` para eventuais necessidades de aleatoriedade.

In [1]:
```
from pysmt.shortcuts import *
import pysmt.typing as types
import numpy as np
```

Como 'input', define-se apenas n, a precisão. Os valores a e b serão tratados futuramente.

In [2]:
```
n = 64      # precisão
```

Como função auxiliar de BitVectors, definiu-se `bv_sel(z,i)`, que seleciona o i-ésimo bit do vetor z.

In [3]:
```
def bv_sel(z,i):                        # seleciona o bit i do BitVec "z"
    return BVExtract(z,start=i,end=i)
```

Como funções necessárias para a modelação, definem-se `declare(i)`, que cria a i-ésima cópia do estado; `init(state)`, que devolve um predicado que testa o estado inicial; `init_ab(state, a, b)`, que recebe a e b separadamente do tratamento do Solver; e `trans(curr, prox)`, que gera um predicado a partir de dois estados que define as condições de transição entre eles.

Para ser possivel fazer o cálculo de $a * b$ para se verificar o invariante, decidiu-se usar BitVectors de tamanho $2n$, já que:

$$log_2(a) \leq n \land log_2(b) \leq n \implies log_2(a * b) \leq 2n$$

Para `declare(i)`, definem-se as variáveis já descritas na Análise.

In [4]:
```
def declare(i): #declara um bitvector de tamanho n
    state = {}
    #declaram-se os BV com margem de manobra para operações (invariante)
    state['x'] = Symbol('x'+str(i),types.BVType(2*n))
    state['y'] = Symbol('y'+str(i),types.BVType(2*n))
    state['z'] = Symbol('z'+str(i),types.BVType(2*n))
    state['p'] = Symbol('p'+str(i),INT) #-1 - erro; 0 - loop; 1 - par; 2 - ímpar; 3 - final
    return state
```

Para `init(state)`, define-se um predicado que gera valores apenas na gama $\{0..2^n - 1\}$ e que também gera valores de y menores que de x. Os valores considerados aqui podem ser considerados também os gerados para a e b.

In [5]:
```
#por o y sempre o mais pequeno dos dois (cuidado com cenas mais à frente)
def init(state):
    return And(Equals(state['z'],BVZero(2*n)), Equals(state['p'], Int(0)), BVULT(state['x'], BV(2**n, 2*n)),
               BVULT(state['y'], BV(2**n, 2*n)), BVULT(state['y'], state['x']))
```

Semelhantemente, `init_ab(state, a, b)` escolherá para y o menor dos dois valores.

In [6]:
```
#a será sempre o valor maior
def init_ab(state,a,b):
    if a < b:
        a,b = b,a
    a = BV(a,n)
    b = BV(b,n)
    return And(Equals(state['z'],BVZero(2*n)), Equals(state['p'], Int(0)), Equals(state['x'], BVZExt(a,n)),
               Equals(state['y'], BVZExt(b,n)))
```

De seguida, definiu-se a função `trans(curr, prox)`, de acordo com a especificação na Análise. Notavelmente, observa-se o seguinte:

1. $2x \geq 2^n \equiv x_{n-1} = 1$
2. $y \geq 0 \pmod 2 \equiv y_0 = 0$
3. $y \geq 1 \pmod 2 \equiv y_0 = 1$
4. $2x = x << 1$
5. $2y = y >> 1$

In [7]:
```
def trans(curr, prox):
    tend = And(Equals(curr['p'], Int(0)), Equals(prox['p'], Int(3)), Equals(curr['y'], BVZero(2*n)),
               Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    tendl = And(Equals(curr['p'], Int(3)), Equals(prox['p'], Int(3)),
                  Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    todd = And(Equals(curr['p'], Int(0)), Equals(prox['p'], Int(2)), Equals(bv_sel(curr['y'],0), BVOne(1)),
               Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    toddt = And(Equals(curr['p'], Int(2)), Equals(prox['p'], Int(0)), Equals(prox['y'], curr['y'] - BVZExt(BVOn
                Equals(prox['z'], curr['z'] + curr['x']), Equals(prox['x'], curr['x']), Not(curr['x'] > BVSub(B

    toddof = And(Equals(curr['p'], Int(2)), Equals(prox['p'], Int(-1)), curr['x'] > BVSub(BV(2**n-1,2*n), curr[
                 Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    teven = And(Equals(curr['p'], Int(0)), Equals(prox['p'], Int(1)), Not(Equals(curr['y'], BVZero(2*n))), Equa
                Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    tevent = And(Equals(curr['p'], Int(1)), Equals(prox['p'], Int(0)), Equals(prox['x'], BVLShl(curr['x'], BVZE
                 Equals(prox['y'], BVLShr(curr['y'], BVZExt(BVOne(1), 2*n-1))), Equals(curr['z'], prox['z']), N

    tevenof = And(Equals(curr['p'], Int(1)), Equals(prox['p'], Int(-1)), Equals(bv_sel(curr['x'], n-1), BVOne(1
                  Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    tofl = And(Equals(curr['p'], Int(-1)), Equals(prox['p'], Int(-1)),
               Equals(curr['x'], prox['x']), Equals(curr['y'], prox['y']), Equals(curr['z'], prox['z']))

    return Or(tend, tendl, todd, toddof, toddt, teven, tevenof, tevent, tofl)
```

Após definir as funções `declare` e `trans` definimos a função de ordem superior `gera_tracok` que cria um traço aleatório que começa em `'init'` (estado inicial), é formado por `k` cópias do estado, usando as funções declare para declarar `k` estados do traço e `trans` para, como descrito no autómato, transicionar os estados.

Com esta função, pode-se simular a execução do problema.

In [17]:
```
def gera_tracok(declare,init,trans,k):
    with Solver(name="z3") as s:
        trace = [declare(i) for i in range(k)]
        s.add_assertion(init(trace[0]))
        for i in range(k-1):
            s.add_assertion(trans(trace[i],trace[i+1]))
        if s.solve():
            for i in range(k-1):
                print(f'(p: %s, x: %s, y: %s, z: %s)' % (s.get_value(trace[i]['p']), s.get_value(trace[i]['x'])
                                                         s.get_value(trace[i]['y']),  s.get_value(trace[i]['z'])))
            print(f'(p: %s, x: %s, y: %s, z: %s)' % (s.get_value(trace[i]['p']), s.get_value(trace[i]['x']),
                                                     s.get_value(trace[i]['y']),  s.get_value(trace[i]['z'])))

gera_tracok(declare,init,trans,10)
```

```
(p: 0, x: 72057594037927936_128, y: 2_128, z: 0_128)
(p: 1, x: 72057594037927936_128, y: 2_128, z: 0_128)
(p: 0, x: 144115188075855872_128, y: 1_128, z: 0_128)
(p: 2, x: 144115188075855872_128, y: 1_128, z: 0_128)
(p: 0, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
(p: 3, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
(p: 3, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
(p: 3, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
(p: 3, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
(p: 3, x: 144115188075855872_128, y: 0_128, z: 144115188075855872_128)
```

A função de ordem superior `bmc_always` para além de gerar um traço também verifica o input e condição `inv` em cada estado e retorna os tamanhos do traço em que essa propriedade é válida.

In [9]:
```
def bmc_always(declare,init,trans,inv,K):
    for k in range(1,K+1):
        with Solver(name="z3") as s:
            trace = [declare(i) for i in range(k)]
            s.add_assertion(init(trace[0]))
            for i in range(k-1):
                s.add_assertion(trans(trace[i],trace[i+1]))
                s.add_assertion(Not(inv(trace[i], trace[0]['x'], trace[0]['y'])))
            s.add_assertion(Not(inv(trace[k-1], trace[0]['x'], trace[0]['y'])))
            if s.solve():
                print(s.get_model())
                for i in range(k-1):
                    print(f'(p: %s, x: %s, y: %s, z: %s)' % (s.get_value(trace[i]['p']), s.get_value(trace[i]['
                                                         s.get_value(trace[i]['y']),  s.get_value(trace[i]['z'])))
                print(f'(p: %s, x: %s, y: %s, z: %s)' % (s.get_value(trace[i]['p']), s.get_value(trace[i]['x'])
                                                         s.get_value(trace[i]['y']),  s.get_value(trace[i]['z'])))
                print('Deu solve!')
                return
    print(f"Propriedade válida para traços de tamanho <= %d." % k)

def inv(state,a,b):
    return Equals(BVAdd(BVMul(state['x'], state['y']), state['z']), BVMul(a,b))

bmc_always(declare,init,trans,inv,10)
```

```
Propriedade válida para traços de tamanho <= 10.
```

Uma consideração importante é o limite de tamanho do traço. Na verdade, segundo a nossa definição de estados, supondo $b = 2^n - 1$, b é constituído apenas por 1's. Nesse caso, a alternância entre os estados 0, 1 e 2 implica 4 estados até chegarmos a um valor de $y = 2^{n-1} - 1$, nomeadamente $0 -> 2 -> 0 -> 1$ ou seja, cujo bit mais significativo deixou de ser 1. Tal repete-se até ao valor de $y = 1$, onde o caso será $0 \rightarrow 2 \rightarrow 0 \rightarrow 3$. Nesse sentido, o número de passos máximos será $4 * log_2 b$. Para além disso, a optimização de escolher o valor de b o menor dos dois, implica que o valor máximo para b que não gera *overflow* é de $2^{n/2} - 1$, sendo esse também o valor para b que maximiza o número de passos. Então, o número de passos máximo é igual a $2n$, o que nos proporciona uma forma de provar o invariante para todos os traços possíveis.

In [10]:
```
bmc_always(declare,init,trans,inv,2*n)
```

```
Propriedade válida para traços de tamanho <= 128.
```

Exemplos

In [18]:
```
gera_tracok(declare,init,trans,5)
```

```
(p: 0, x: 18443023335223725285_128, y: 309506097156_128, z: 0_128)
(p: 1, x: 18443023335223725285_128, y: 309506097156_128, z: 0_128)
(p: -1, x: 18443023335223725285_128, y: 309506097156_128, z: 0_128)
(p: -1, x: 18443023335223725285_128, y: 309506097156_128, z: 0_128)
(p: -1, x: 18443023335223725285_128, y: 309506097156_128, z: 0_128)
```

In [19]:
```
gera_tracok(declare,init,trans,10)
```

```
(p: 0, x: 64_128, y: 13_128, z: 0_128)
(p: 2, x: 64_128, y: 13_128, z: 0_128)
(p: 0, x: 64_128, y: 12_128, z: 64_128)
(p: 1, x: 64_128, y: 12_128, z: 64_128)
(p: 0, x: 128_128, y: 6_128, z: 64_128)
(p: 1, x: 128_128, y: 6_128, z: 64_128)
(p: 0, x: 256_128, y: 3_128, z: 64_128)
(p: 2, x: 256_128, y: 3_128, z: 64_128)
(p: 0, x: 256_128, y: 2_128, z: 320_128)
(p: 0, x: 256_128, y: 2_128, z: 320_128)
```

In [20]:
```
gera_tracok(declare,init,trans,15)
```

```
(p: 0, x: 732279039066048_128, y: 4_128, z: 0_128)
(p: 1, x: 732279039066048_128, y: 4_128, z: 0_128)
(p: 0, x: 1464558078132096_128, y: 2_128, z: 0_128)
(p: 1, x: 1464558078132096_128, y: 2_128, z: 0_128)
(p: 0, x: 2929116156264192_128, y: 1_128, z: 0_128)
(p: 2, x: 2929116156264192_128, y: 1_128, z: 0_128)
(p: 0, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
(p: 3, x: 2929116156264192_128, y: 0_128, z: 2929116156264192_128)
```

In [ ]: