TP2 - Exercício 1

Grupo 1

Diogo Coelho da Silva A100092

Pedro Miguel Ramôa Oliveira A97686

Problema proposto: Considere o problema descrito no documento Lógica Computacional: Multiplicação de Inteiros. Neste documento usa-se um "Control Flow Automaton" como modelo do programa imperativo que calcula a multiplicação de inteiros positivos representados por vetores de bits.

Pretende-se:

1. Construir um SFOTS, usando BitVec's de tamanho n , que descreva o comportamento deste autómato; para isso identifique e codifique em `Z3` ou `pySMT`, as variáveis do modelo, o estado inicial , a relação de transição e o estado de erro.
2. Usando k-indução verifique nesse SFOTS se a propriedade $(x * y + z = a * b)$ é um invariante do seu comportamento.
3. Usando k-indução no FOTS acima e adicionando ao estado inicial a condição $(a < 2^{n/2}) \wedge (b < 2^{n/2})$, verifique a segurança do programa; nomeadamente prove que, com tal estado inicial, o estado de erro nunca é acessível.

Proposta de resolução:

A proposta de resolução apresentada neste código tem como objetivo modelar e verificar o comportamento de um algoritmo de multiplicação utilizando vetores de bits com 10 bits. Utilizando a biblioteca PySMT, o código declara variáveis de estado essenciais, como pc, x, y, a, b e z, e inicializa estas variáveis com valores fornecidos.

As condições de transição entre estados são definidas na função `trans(curr, prox)` , que simula as operações do algoritmo, garantindo que a lógica esteja correta. A função `invariant_check(state)` verifica a invariância da relação $x * y + z = a * b$, enquanto `overflow(state)` detecta condições de overflow, assegurando a integridade dos valores.

A função principal, `bmc_always(inv, K, x_val, y_val)` , utiliza um solver para avaliar invariantes e detectar overflows para os primeiros k estados do sistema, fornecendo feedback detalhado sobre o estado atual e relatando quaisquer falhas. Assim, a proposta não só simula o algoritmo de multiplicação, mas também verifica formalmente suas propriedades, demonstrando robustez através de testes com valores aleatórios.

1. Importar as bibliotecas importantes

- `pysmt` : Importa a biblioteca PySMT, uma ferramenta para a criação de expressões lógicas e verificação formal.
- `random` : Gera aleatoriedade para simular eventos, como a seleção aleatória de estados ou a determinação de valores em cenários de simulação, proporcionando variabilidade nos resultados e comportamento do programa.

In [20]:
```python
import random
from pysmt.shortcuts import *
from pysmt.typing import BVType
```

2. Definição da máquina de estados

Análise do Código

Este código define uma máquina de estados que simula um processo de multiplicação. A máquina possui variáveis de estado que são atualizadas em cada transição. Vamos detalhar as funções presentes no código:

1. Função `declare(i)`

A função `declare` é responsável por declarar as variáveis de estado para um índice de estado específico `i` . Ela cria um dicionário que contém as seguintes variáveis:

- `pc` : Contador de Programa (Program Counter) representando o estado atual.
- `x` , `y` , `a` , `b` , `z` : Variáveis de dados utilizadas no cálculo da multiplicação.

Cada variável é representada como um símbolo (usando `Symbol`) com um tamanho de bit especificado por `n` .

2. Função `init(state, x_val, y_val)`

A função init inicializa as variáveis de estado com valores específicos. Ela toma como entrada um estado e dois valores x_val e y_val e cria as seguintes condições:

- `PC` : O contador de programa inicia em 0.
- `X` : A variável x é inicializada com o valor de x_val.
- `Y` : A variável y é inicializada com o valor de y_val.
- `A` : A variável a é igual a x_val.
- `B` : A variável b é igual a y_val.
- `Z` : A variável z inicia em 0.

Adicionamos restrições vindas do enunciado para que a < 2^(n/2) and b < 2^(n/2)

- restricao_a = BVULT(state['a'], BV(half_max, n))
- restricao_b = BVULT(state['b'], BV(half_max, n))

As condições são representadas por expressões booleanas (usando Equals) que devem ser verdadeiras para o estado inicial ser válido.

2. Função `trans(curr, prox)`

A função trans define as condições de transição entre estados, levando em conta o estado atual (curr) e o próximo estado (prox). Existem quatro transições definidas:

Transições:

- `t01` : Transição do estado 0 para o estado 1.
  - O `pc` passa de 0 para 1.
  - As variáveis x, y, a, b permanecem inalteradas.
  - z inicia em 0.
- `t12` : Transição do estado 1, onde y não é zero.
  - O `pc` continua em 1.
  - y é decrementado em 1.
  - z é atualizado somando curr['z'] com curr['x'].
- `t23` : Transição do estado 1 para o estado 2, onde y é zero.
  - O `pc` muda para 2.
  - As variáveis x, y, a, b permanecem inalteradas.
  - z permanece igual ao seu valor anterior.
- `t33` : Transição no estado 2, onde não há mudanças.
  - O `pc` permanece em 2.
  - Todas as variáveis continuam com seus valores inalterados.

In [21]:
```python
# Função para declarar variáveis de estado para cada índice i
# Cria um dicionário com símbolos para: contador de programa (pc),
# e variáveis x, y, a, b e z
n = 10

# Este código implementa uma máquina de estados para simular multiplicação usando operações básicas

# Variáveis e estruturas principais:
# - n: tamanho em bits para os números (definido como 10)
# - state: dicionário que mantém o estado actual da máquina, incluindo:
#   * pc: contador de programa
#   * x, y: operandos da multiplicação
#   * a, b: cópias dos operandos
#   * z: resultado da operação

# A função declare(i):
# Cria um novo estado com símbolos únicos para cada variável usando um índice i
# Utiliza tipos de bitvector (BVType) para representar os números

def declare(i):
    state = {}
    state['pc'] = Symbol('pc' + str(i), BVType(n)) # Contador de programa
    state['x'] = Symbol('x' + str(i), BVType(n)) # Primeiro operando
    state['y'] = Symbol('y' + str(i), BVType(n)) # Segundo operando
    state['a'] = Symbol('a' + str(i), BVType(n)) # Cópia de x
    state['b'] = Symbol('b' + str(i), BVType(n)) # Cópia de y
    state['z'] = Symbol('z' + str(i), BVType(n)) # Resultado
    return state

# A função init(state, x_val, y_val):
# Inicializa o estado com valores específicos para x e y
# Define restrições importantes:
# - Contador de programa começa em 0
# - Variáveis a e b devem ser menores que 2^(n/2)
# - z começa em 0
def init(state, x_val, y_val):
    half_max = 2 ** (n // 2)
    PC = Equals(state['pc'], BV(0, n))
    X = Equals(state['x'], BV(x_val, n))
    Y = Equals(state['y'], BV(y_val, n))
    A = Equals(state['a'], BV(x_val, n))  # Mantem a igual a x_val
    B = Equals(state['b'], BV(y_val, n))  # Mantem b igual a y_val
    Z = Equals(state['z'], BV(0, n))  # z inicia a 0

# Adiciona restrições para garantir que os valores não ultrapassem os limites:
    restricao_a = BVULT(state['a'], BV(half_max, n))
    restricao_b = BVULT(state['b'], BV(half_max, n))

# Combina todas as condições num único retorno:
    return And(PC, X, Y, A, B, Z, restricao_a, restricao_b)  # Retorna a conjunção lógica de todas as condições iniciais e restrições


# A função trans(curr, prox):
# Define as transições possíveis entre estados:
# t01: Estado inicial -> Estado de multiplicação
# t12: Estado de multiplicação em progresso (y > 0)
#      Adiciona x ao resultado z e decrementa y
# t23: Estado de multiplicação -> Estado final (quando y = 0)
# t33: Estado final (mantém os valores)
def trans(curr, prox):
    t01 = And(
        Equals(curr['pc'], BV(0, n)),
        Equals(prox['pc'], BV(1, n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['z'], BV(0, n))
    )

    t12 = And(
        Equals(curr['pc'], BV(1, n)),
        NotEquals(curr['y'], BV(0, n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], BVSub(curr['y'], BV(1, n))),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['z'], BVAdd(curr['z'], curr['x'])),
        Equals(prox['pc'], BV(1, n))
    )

    t23 = And(
        Equals(curr['pc'], BV(1, n)),
        Equals(curr['y'], BV(0, n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['z'], curr['z']),
        Equals(prox['pc'], BV(2, n))
    )

    t33 = And(
        Equals(curr['pc'], BV(2, n)),
        Equals(prox['pc'], BV(2, n)),
        Equals(prox['x'], curr['x']),
        Equals(prox['y'], curr['y']),
        Equals(prox['a'], curr['a']),
        Equals(prox['b'], curr['b']),
        Equals(prox['z'], curr['z'])
    )

    return Or(t01, t12, t23, t33)
```

3. Restrição para o invariante

A função `invariant_check` verifica uma propriedade específica que deve ser verdadeira em todos os estados de um sistema de computação. Neste caso, a propriedade a ser verificada é:

$$x \cdot y + z = a \cdot b$$

Componentes da Função

1. **Entrada `state`**:
   - A função recebe como argumento um dicionário `state` que contém as variáveis do estado atual da máquina. Essas variáveis incluem `x` , `y` , `z` , `a` , e `b` .
2. **Verificação do Invariante**:
   - A expressão `BVAdd(BVMul(state['x'], state['y']), state['z'])` calcula $x * y + z$.
   - A expressão `BVMul(state['a'], state['b'])` calcula $a * b$.
   - A função `Equals` verifica se estas duas expressões são iguais.

Portanto, a função retorna uma condição booleana que é verdadeira se e somente se a relação $x \cdot y + z = a \cdot b$ se mantiver no estado atual da máquina.

Verificação da Propriedade como Invariante usando k-Indução

A utilização da k-indução para verificar a propriedade $x \cdot y + z = a \cdot b$ como um invariante envolve duas etapas principais:

1. **Base de Indução**:
   - Inicialmente, você deve verificar se a propriedade é verdadeira no estado inicial da máquina (ou seja, o estado quando o programa começa a executar). Isso geralmente é feito usando a função de inicialização (como `init`), onde você define as variáveis de estado. No estado inicial, você assegura que x, y, z, a, e b estão configurados de tal forma que a relação é válida.
2. **Passo de Indução**:
   - A seguir, deve-se assumir que a propriedade é verdadeira para um estado qualquer k (hipótese de indução) e, em seguida, demonstrar que ela continua a ser verdadeira para o próximo estado $k + 1$. Isso envolve analisar as transições definidas na função `trans` e garantir que, se a propriedade é verdadeira no estado k, ela também se mantém válida no estado $k + 1$. Ou seja, deve-se garantir que as operações realizadas nas variáveis de estado durante a transição não violam a igualdade.

A função `invariant_check` é essencial para assegurar que a propriedade $x \cdot y + z = a \cdot b$ se mantém ao longo do tempo, independentemente das operações que a máquina realiza. Usar k-indução para verificar este invariante proporciona um método robusto para confirmar que o comportamento do sistema é consistente e correto em relação a esta propriedade ao longo de todas as suas execuções.

In [22]:
```python
#Verificação do invariante
def invariant_check(state):
    return Equals(
        BVAdd(BVMul(state['x'], state['y']), state['z']),
        BVMul(state['a'], state['b'])
    )
```

4. Função para verificar se existe overflow

A função `overflow` verifica se ocorre um overflow em qualquer uma das variáveis de estado relevantes para a operação. O overflow é um problema que pode acontecer quando um valor excede a capacidade máxima que pode ser representada com um número binário de tamanho fixo.

Componentes da Função

1. **Valor Máximo**:
   - A variável `max_val` é definida como o maior valor que pode ser representado com `n` bits. Isso é calculado como $2^n - 1$.
2. **Verificação de Overflow**:
   - A função retorna `True` se qualquer uma das variáveis `x` , `y` , `a` , `b` , ou `z` for maior que `max_val` . Isso é feito usando a função `BVUGT` , que compara as variáveis de estado com `max_val` .

In [23]:
```python
#condição para overflow
def overflow(state):
    max_val = BV((2**n) - 1, n)
    return Or(
        BVUGT(state['x'], max_val),
        BVUGT(state['y'], max_val),
        BVUGT(state['a'], max_val),
        BVUGT(state['b'], max_val),
        BVUGT(state['z'], max_val)
    )
```

5. Execução do código

In [24]:
```python
def print_state(k, state, solver):
    # Utiliza f-strings para formatar a saída
    # solver.get_value() obtém o valor atual de cada variável
    # bv_unsigned_value() converte o bitvector para um inteiro sem sinal
    print(f"Estado {k}: pc = {solver.get_value(state['pc']).bv_unsigned_value()}, "
          f"x = {solver.get_value(state['x']).bv_unsigned_value()}, "
          f"y = {solver.get_value(state['y']).bv_unsigned_value()}, "
          f"a = {solver.get_value(state['a']).bv_unsigned_value()}, "
          f"b = {solver.get_value(state['b']).bv_unsigned_value()}, "
          f"z = {solver.get_value(state['z']).bv_unsigned_value()}")


"""
    Função de verificação por bounded model checking (BMC)

    Args Entrada:
    - inv: função que define o invariante a verificar
    - K: número de estados a verificar
    - x_val, y_val: valores iniciais para multiplicação

    Funcionamento passo a passo:
    1. Cria um novo solver
    2. Declara K+1 estados da máquina
    3. Inicializa o estado 0 com os valores fornecidos
    4. Para cada estado k até K:
        - Verifica a transição do estado anterior
        - Imprime o estado atual
        - Verifica se o invariante se mantém
        - Verifica se existe overflow
        - Interrompe se encontrar violação
    5. No final:
        - Confirma que o invariante se verifica
        - Calcula e verifica o resultado final
    """
def bmc_always(inv, K, x_val, y_val):
    with Solver() as solver:
        states = [declare(i) for i in range(K + 1)]
        solver.add_assertion(init(states[0], x_val, y_val))

        for k in range(K):
            if k > 0:
                solver.add_assertion(trans(states[k - 1], states[k]))
            solver.push()
            if solver.solve():
                print_state(k, states[k], solver)
            solver.pop()

            solver.push()
            if solver.solve([Not(inv(states[k]))]):
                print(f"> Invariante falha no estado {k}")
                print_state(k, states[k], solver)
                solver.pop()
                return

            if solver.solve([overflow(states[k])]):
                print(f"> Overflow detetado no estado {k}")
                print_state(k, states[k], solver)
                solver.pop()
                return

            solver.pop()

        print(f"> Invariante verifica-se para os primeiros {K} estados.")

        final_x = solver.get_value(states[K-1]['x']).bv_unsigned_value()
        final_y = solver.get_value(states[K-1]['y']).bv_unsigned_value()
        final_z = solver.get_value(states[K-1]['z']).bv_unsigned_value()
        final_a = solver.get_value(states[K-1]['a']).bv_unsigned_value()
        final_b = solver.get_value(states[K-1]['b']).bv_unsigned_value()

        expected_sum = final_x * final_y + final_z
        expected_product = final_a * final_b

        print(f"Resultado final: x * y + z = {expected_sum}, a * b = {expected_product}. Verificação: {'Correto' if expected_sum == expected_product else 'Incorreto'}."


# Exemplo de teste com valores aleatórios para a e b
for _ in range(3):
    a_val = random.randint(1, 2**(n // 2) - 1)  # Gerar valor aleatório para a
    b_val = random.randint(1, 2**(n // 2) - 1)  # Gerar valor aleatório para b
    print(f"Testando com a = {a_val}, b = {b_val}")
    bmc_always(invariant_check, 20, a_val, b_val)
    bmc_always(invariant_check, 20, a_val, b_val)
```

```
Testando com a = 30, b = 15
Estado 0: pc = 0, x = 30, y = 15, a = 30, b = 15, z = 0
Estado 1: pc = 1, x = 30, y = 15, a = 30, b = 15, z = 0
Estado 2: pc = 1, x = 30, y = 14, a = 30, b = 15, z = 30
Estado 3: pc = 1, x = 30, y = 13, a = 30, b = 15, z = 60
Estado 4: pc = 1, x = 30, y = 12, a = 30, b = 15, z = 90
Estado 5: pc = 1, x = 30, y = 11, a = 30, b = 15, z = 120
Estado 6: pc = 1, x = 30, y = 10, a = 30, b = 15, z = 150
Estado 7: pc = 1, x = 30, y = 9, a = 30, b = 15, z = 180
Estado 8: pc = 1, x = 30, y = 8, a = 30, b = 15, z = 210
Estado 9: pc = 1, x = 30, y = 7, a = 30, b = 15, z = 240
Estado 10: pc = 1, x = 30, y = 6, a = 30, b = 15, z = 270
Estado 11: pc = 1, x = 30, y = 5, a = 30, b = 15, z = 300
Estado 12: pc = 1, x = 30, y = 4, a = 30, b = 15, z = 330
Estado 13: pc = 1, x = 30, y = 3, a = 30, b = 15, z = 360
Estado 14: pc = 1, x = 30, y = 2, a = 30, b = 15, z = 390
Estado 15: pc = 1, x = 30, y = 1, a = 30, b = 15, z = 420
Estado 16: pc = 1, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 17: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 18: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 19: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 450, a * b = 450. Verificação: Correto.
Estado 0: pc = 0, x = 30, y = 15, a = 30, b = 15, z = 0
Estado 1: pc = 1, x = 30, y = 15, a = 30, b = 15, z = 0
Estado 2: pc = 1, x = 30, y = 14, a = 30, b = 15, z = 30
Estado 3: pc = 1, x = 30, y = 13, a = 30, b = 15, z = 60
Estado 4: pc = 1, x = 30, y = 12, a = 30, b = 15, z = 90
Estado 5: pc = 1, x = 30, y = 11, a = 30, b = 15, z = 120
Estado 6: pc = 1, x = 30, y = 10, a = 30, b = 15, z = 150
Estado 7: pc = 1, x = 30, y = 9, a = 30, b = 15, z = 180
Estado 8: pc = 1, x = 30, y = 8, a = 30, b = 15, z = 210
Estado 9: pc = 1, x = 30, y = 7, a = 30, b = 15, z = 240
Estado 10: pc = 1, x = 30, y = 6, a = 30, b = 15, z = 270
Estado 11: pc = 1, x = 30, y = 5, a = 30, b = 15, z = 300
Estado 12: pc = 1, x = 30, y = 4, a = 30, b = 15, z = 330
Estado 13: pc = 1, x = 30, y = 3, a = 30, b = 15, z = 360
Estado 14: pc = 1, x = 30, y = 2, a = 30, b = 15, z = 390
Estado 15: pc = 1, x = 30, y = 1, a = 30, b = 15, z = 420
Estado 16: pc = 1, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 17: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 18: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
Estado 19: pc = 2, x = 30, y = 0, a = 30, b = 15, z = 450
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 450, a * b = 450. Verificação: Correto.
Testando com a = 5, b = 10
Estado 0: pc = 0, x = 5, y = 10, a = 5, b = 10, z = 0
Estado 1: pc = 1, x = 5, y = 10, a = 5, b = 10, z = 0
Estado 2: pc = 1, x = 5, y = 9, a = 5, b = 10, z = 5
Estado 3: pc = 1, x = 5, y = 8, a = 5, b = 10, z = 10
Estado 4: pc = 1, x = 5, y = 7, a = 5, b = 10, z = 15
Estado 5: pc = 1, x = 5, y = 6, a = 5, b = 10, z = 20
Estado 6: pc = 1, x = 5, y = 5, a = 5, b = 10, z = 25
Estado 7: pc = 1, x = 5, y = 4, a = 5, b = 10, z = 30
Estado 8: pc = 1, x = 5, y = 3, a = 5, b = 10, z = 35
Estado 9: pc = 1, x = 5, y = 2, a = 5, b = 10, z = 40
Estado 10: pc = 1, x = 5, y = 1, a = 5, b = 10, z = 45
Estado 11: pc = 1, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 12: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 13: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 14: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 15: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 16: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 17: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 18: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 19: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 50, a * b = 50. Verificação: Correto.
Estado 0: pc = 0, x = 5, y = 10, a = 5, b = 10, z = 0
Estado 1: pc = 1, x = 5, y = 10, a = 5, b = 10, z = 0
Estado 2: pc = 1, x = 5, y = 9, a = 5, b = 10, z = 5
Estado 3: pc = 1, x = 5, y = 8, a = 5, b = 10, z = 10
Estado 4: pc = 1, x = 5, y = 7, a = 5, b = 10, z = 15
Estado 5: pc = 1, x = 5, y = 6, a = 5, b = 10, z = 20
Estado 6: pc = 1, x = 5, y = 5, a = 5, b = 10, z = 25
Estado 7: pc = 1, x = 5, y = 4, a = 5, b = 10, z = 30
Estado 8: pc = 1, x = 5, y = 3, a = 5, b = 10, z = 35
Estado 9: pc = 1, x = 5, y = 2, a = 5, b = 10, z = 40
Estado 10: pc = 1, x = 5, y = 1, a = 5, b = 10, z = 45
Estado 11: pc = 1, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 12: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 13: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 14: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 15: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 16: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 17: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 18: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
Estado 19: pc = 2, x = 5, y = 0, a = 5, b = 10, z = 50
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 50, a * b = 50. Verificação: Correto.
Testando com a = 29, b = 26
Estado 0: pc = 0, x = 29, y = 26, a = 29, b = 26, z = 0
Estado 1: pc = 1, x = 29, y = 26, a = 29, b = 26, z = 0
Estado 2: pc = 1, x = 29, y = 25, a = 29, b = 26, z = 29
Estado 3: pc = 1, x = 29, y = 24, a = 29, b = 26, z = 58
Estado 4: pc = 1, x = 29, y = 23, a = 29, b = 26, z = 87
Estado 5: pc = 1, x = 29, y = 22, a = 29, b = 26, z = 116
Estado 6: pc = 1, x = 29, y = 21, a = 29, b = 26, z = 145
Estado 7: pc = 1, x = 29, y = 20, a = 29, b = 26, z = 174
Estado 8: pc = 1, x = 29, y = 19, a = 29, b = 26, z = 203
Estado 9: pc = 1, x = 29, y = 18, a = 29, b = 26, z = 232
Estado 10: pc = 1, x = 29, y = 17, a = 29, b = 26, z = 261
Estado 11: pc = 1, x = 29, y = 16, a = 29, b = 26, z = 290
Estado 12: pc = 1, x = 29, y = 15, a = 29, b = 26, z = 319
Estado 13: pc = 1, x = 29, y = 14, a = 29, b = 26, z = 348
Estado 14: pc = 1, x = 29, y = 13, a = 29, b = 26, z = 377
Estado 15: pc = 1, x = 29, y = 12, a = 29, b = 26, z = 406
Estado 16: pc = 1, x = 29, y = 11, a = 29, b = 26, z = 435
Estado 17: pc = 1, x = 29, y = 10, a = 29, b = 26, z = 464
Estado 18: pc = 1, x = 29, y = 9, a = 29, b = 26, z = 493
Estado 19: pc = 1, x = 29, y = 8, a = 29, b = 26, z = 522
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 754, a * b = 754. Verificação: Correto.
Estado 0: pc = 0, x = 29, y = 26, a = 29, b = 26, z = 0
Estado 1: pc = 1, x = 29, y = 26, a = 29, b = 26, z = 0
Estado 2: pc = 1, x = 29, y = 25, a = 29, b = 26, z = 29
Estado 3: pc = 1, x = 29, y = 24, a = 29, b = 26, z = 58
Estado 4: pc = 1, x = 29, y = 23, a = 29, b = 26, z = 87
Estado 5: pc = 1, x = 29, y = 22, a = 29, b = 26, z = 116
Estado 6: pc = 1, x = 29, y = 21, a = 29, b = 26, z = 145
Estado 7: pc = 1, x = 29, y = 20, a = 29, b = 26, z = 174
Estado 8: pc = 1, x = 29, y = 19, a = 29, b = 26, z = 203
Estado 9: pc = 1, x = 29, y = 18, a = 29, b = 26, z = 232
Estado 10: pc = 1, x = 29, y = 17, a = 29, b = 26, z = 261
Estado 11: pc = 1, x = 29, y = 16, a = 29, b = 26, z = 290
Estado 12: pc = 1, x = 29, y = 15, a = 29, b = 26, z = 319
Estado 13: pc = 1, x = 29, y = 14, a = 29, b = 26, z = 348
Estado 14: pc = 1, x = 29, y = 13, a = 29, b = 26, z = 377
Estado 15: pc = 1, x = 29, y = 12, a = 29, b = 26, z = 406
Estado 16: pc = 1, x = 29, y = 11, a = 29, b = 26, z = 435
Estado 17: pc = 1, x = 29, y = 10, a = 29, b = 26, z = 464
Estado 18: pc = 1, x = 29, y = 9, a = 29, b = 26, z = 493
Estado 19: pc = 1, x = 29, y = 8, a = 29, b = 26, z = 522
> Invariante verifica-se para os primeiros 20 estados.
Resultado final: x * y + z = 754, a * b = 754. Verificação: Correto.
```