Trabalho Prático 2

Grupo 04 - Renato Garcia (A101987) & Bernardo Moniz (A102497)

Problema 2

Enunciado

2. Considere o problema descrito no documento +Lógica Computacional: Multiplicação de Inteiros. Nesse 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

   a. 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.

   b. Usando k-indução, verifique nesse SFOTS se a propriedade $(x * y + z = a * b)$ é um invariante do seu comportamento.

   c. 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.

Resolução

- A função **declare** cria a declaração das variáveis do tipo BitVec, para **x**, **y** e **z**, com tamanho **n**, e do tipo Int para **pc**
- A função **init** desenvolve o estado inicial das variáveis **x**, **y**, **z** e **pc**
- A função **trans**, recebe **curr** que é o estado atual, **prox** que é o próximo estado e **n** o número de bits, desenvolve as transições para os estados possíveis, sendo:

  -> **pc = 0** o estado inicial, "***init***"

  -> **pc = 1** o estado de loop, "***skip***"

  -> **pc = 2** o estado em que **y** é par e diferente de 0, "***left***"

  -> **pc = 3** o estado em que **y** é ímpar e diferente de 0, "***right***"

  -> **pc = 4** o estado de overflow, "***error***"

  -> **pc = 5** o estado final, "***stop***"

- A função **error** desenvolve o estado de erro, que ocorre quando **pc = 4** e que não acontece **z == a * b**

In [1]:
```python
from z3 import *

def declare(i, n):
    state = {}

    state['pc'] = Int('pc'+str(i))
    state['x'] = BitVec('x'+str(i), n)
    state['y'] = BitVec('y'+str(i), n)
    state['z'] = BitVec('z'+str(i), n)

    return state

def init(state, a, b):
    return And(state['pc'] == 0,
               state['x'] == a,
               state['y'] == b,
               state['z'] == 0)

def trans(curr, prox, n):
    same_values = And(
        prox['x'] == curr['x'],
        prox['y'] == curr['y'],
        prox['z'] == curr['z']
    )

    # Estado inicial (pc = 0) para o loop (pc = 1)
    t01 = And(
        curr['pc'] == 0,
        prox['pc'] == 1,
        same_values
    )

    # y == 0
    t15 = And(
        curr['y'] == 0,
        curr['pc'] == 1,
        prox['pc'] == 5,
        same_values
    )

    # y != 0 ^ even(y)
    t12 = And(
        curr['y'] != 0,
        URem(curr['y'], 2) == 0,
        curr['pc'] == 1,
        prox['pc'] == 2,
        same_values
    )

    # Q2 -> Q1 ou Q2 -> Q4 (com verificação de overflow no shift)
    new_x = curr['x'] << BitVecVal(1, n)
    t21_24 = And(
        curr['pc'] == 2,
        prox['y'] == curr['y'] >> BitVecVal(1, n),
        prox['z'] == curr['z'],

        If(ULT(new_x, curr['x']),  # Detecta overflow no shift left

           # Se houver overflow, vai para Q4
           And(prox['pc'] == 4, prox['x'] == curr['x']),

           # Se não houver overflow, vai para Q1 com o novo valor de x
           And(prox['pc'] == 1, prox['x'] == new_x)
        )
    )

    # y != 0 ^ odd(y)
    t13 = And(
        curr['y'] != 0,
        URem(curr['y'], 2) == 1,
        curr['pc'] == 1,
        prox['pc'] == 3,
        same_values
    )

    # Q3 -> Q1 ou Q3 -> Q4 (com verificação de overflow na soma)
    new_z = curr['z'] + curr['x']
    t31_34 = And(
        curr['pc'] == 3,
        prox['x'] == curr['x'],
        prox['y'] == curr['y'] - BitVecVal(1, n),

        If(ULT(new_z, curr['z']),  # Detecta overflow na soma

           # Se houver overflow, vai para Q4
           And(prox['pc'] == 4, prox['z'] == curr['z']),

           # Se não houver overflow, vai para Q1 com o novo valor de z
           And(prox['pc'] == 1, prox['z'] == new_z)
        )
    )

    t_stop = And(
        prox['pc'] == curr['pc'],
        same_values,

        Or(
            And(curr['pc'] == 4, prox['pc'] == 4),
            And(curr['pc'] == 5, prox['pc'] == 5)
        )
    )

    return Or(t01, t15, t12, t21_24, t13, t31_34, t_stop)

def error(state):
    return Or(
        # Q4 - Estado de erro (overflow)
        state['pc'] == 4,

        # Q5 - Estado final com resultado incorreto
        And(state['pc'] == 5,
            state['y'] == 0,
            state['z'] != state['x'] * state['y']),

        # Outros estados inválidos
        state['pc'] < 0,
        state['pc'] > 5
    )
```

- A função **gera_traco** imprime o valor das variáveis à medida que vão percorrendo os estados, através das variáveis do estado, em que recebe **declare** que cria as variáveis, **init** que define o estado inicial, **trans** que define a relação de transição, **error** que define o estado de erro, **k** o tamanho do traço, **n** o número de bits, **a** e **b** os valores a multiplicar

In [2]:
```python
def gera_traco(declare, init, trans, error, k, n, a, b):
    s = Solver()

    trace = [declare(i, n) for i in range(k)]

    s.add(init(trace[0], a, b))

    for i in range(k - 1):
        s.add(trans(trace[i], trace[i+1], n))

    # Adicionar condição de erro em algum ponto do traço
    error_condition = Or([error(state) for state in trace])
    s.add(error_condition)

    if s.check() == sat:
        m = s.model()

        for i in range(k):
            print("-" * 20)
            print(f"Passo {i}\n")
            for v in trace[i]:
                val = m[trace[i][v]]
                if val is not None:
                    print(f"{v} = {val}")
```

Exemplo em que chega ao estado final

In [3]:
```python
gera_traco(declare, init, trans, error, 10, 8, 50, 3)
```

```
--------------------
Passo 0

pc = 0
x = 50
y = 3
z = 0
--------------------
Passo 1

pc = 1
x = 50
y = 3
z = 0
--------------------
Passo 2

pc = 3
x = 50
y = 3
z = 0
--------------------
Passo 3

pc = 1
x = 50
y = 2
z = 50
--------------------
Passo 4

pc = 2
x = 50
y = 2
z = 50
--------------------
Passo 5

pc = 1
x = 100
y = 1
z = 50
--------------------
Passo 6

pc = 3
x = 100
y = 1
z = 50
--------------------
Passo 7

pc = 1
x = 100
y = 0
z = 150
--------------------
Passo 8

pc = 5
x = 100
y = 0
z = 150
--------------------
Passo 9

pc = 5
x = 100
y = 0
z = 150
```

Exemplo em que chega ao estado de erro

In [4]:
```python
gera_traco(declare, init, trans, error, 8, 8, 100, 4)
```

```
--------------------
Passo 0

pc = 0
x = 100
y = 4
z = 0
--------------------
Passo 1

pc = 1
x = 100
y = 4
z = 0
--------------------
Passo 2

pc = 2
x = 100
y = 4
z = 0
--------------------
Passo 3

pc = 1
x = 200
y = 2
z = 0
--------------------
Passo 4

pc = 2
x = 200
y = 2
z = 0
--------------------
Passo 5

pc = 4
x = 200
y = 1
z = 0
--------------------
Passo 6

pc = 4
x = 200
y = 1
z = 0
--------------------
Passo 7

pc = 4
x = 200
y = 1
z = 0
```

- A função **check_inv** $(x * y + z = a * b)$ é um invariante, recebe os argumentos **state** onde estão as variáveis, **a** e **b** os valores a multiplicar e **n** o número de bits

In [5]:
```python
def check_inv(state, a, b, n):
    return (state['x'] * state['y'] + state['z'] == BitVecVal(a, n) * BitVecVal(b, n))
```

- A função **k_induction** verifica se a propriedade $(x * y + z = a * b)$ é um invariante do comportamento do SFOTS, em que recebe **declare** que cria as variáveis, **init** que define o estado inicial, **trans** que define a relação de transição, **error** que define o estado de erro, **inv** é o invariante, **k** o tamanho do traço, **n** o número de bits, **a** e **b** os valores a multiplicar

In [6]:
```python
def k_induction(declare, init, trans, error, inv, k, n, a, b):
    with Solver() as solver:
        s = [declare(i, n) for i in range(k)]
        solver.add(init(s[0], a, b))
        for i in range(k-1):
            solver.add(trans(s[i], s[i+1], n))

        for i in range(k):
            solver.push()
            solver.add(Not(inv(s[i], a, b, n)))
            solver.add(error(s[i]))
            if solver.check() == sat:
                print(f"> Contradição! O invariante não se verifica nos k estados iniciais.")
                m = solver.model()
                for j, state in enumerate(s[:i+1]):
                    print("-" * 20)
                    print(f"Estado {j}\n")
                    print(f"pc = {m[state['pc']]}")
                    print(f"x = {m[state['x']]}")
                    print(f"y = {m[state['y']]}")
                    print(f"z = {m[state['z']]}")
                return
            solver.pop()

        s2 = [declare(i+k,n) for i in range(k+1)]

        for i in range(k):
            solver.add(inv(s2[i], a, b, n))
            solver.add(trans(s2[i], s2[i+1],n))
            solver.add(error(s2[i]))

        solver.add(Not(inv(s2[-1], a, b, n)))

        if solver.check() == sat:
            print(f"> Contradição! O passo indutivo não se verifica.")
            m = solver.model()
            for i, state in enumerate(s2):
                print("-" * 20)
                print(f"Estado {i}\n")
                print(f"pc = {m[state['pc']]}")
                print(f"x = {m[state['x']]}")
                print(f"y = {m[state['y']]}")
                print(f"z = {m[state['z']]}")
            return

        print(f"> A propriedade verifica-se por k-indução (k={k}).")

k_induction(declare, init, trans, error, check_inv, 20, 10, 150, 2)
```

```
> A propriedade verifica-se por k-indução (k=20).
```

- A função **k_induction_safety** implementa a verificação de segurança por k-indução com estado inicial restrito, $(a < 2^{n/2})(b < 2^{n/2})$, para que o estado de erro não seja acessível, em que recebe **declare** que cria as variáveis, **init** que define o estado inicial, **trans** que define a relação de transição, **error** que define o estado de erro, **inv** é o invariante, **k** o tamanho do traço, **n** o número de bits, **a** e **b** os valores a multiplicar

In [7]:
```python
def k_induction_safety(declare, init, trans, error, inv, k, n, a, b):
    """
    Implementa verificação de segurança por k-indução com estado inicial restrito
    """
    print(f"Verificação de segurança com K = {k}")

    def init_restricted(state, init, a, b):
        a_bv = BitVecVal(a, n)
        b_bv = BitVecVal(b, n)

        limit = BitVecVal(2 ** (n//2), n)

        return And(
            init(state, a, b),
            ULT(a_bv, limit),  # a < 2^(n/2)
            ULT(b_bv, limit)   # b < 2^(n/2)
        )

    with Solver() as solver:
        s = [declare(i, n) for i in range(k)]
        solver.add(init_restricted(s[0], init, a, b))
        for i in range(k-1):
            solver.add(trans(s[i], s[i+1], n))

            for i in range(k):
                solver.push()
                solver.add(Not(inv(s[i], a, b, n)))
                solver.add(error(s[i]))
                if solver.check() == sat:
                    print(f"> Contradição! O invariante não se verifica nos k estados iniciais.")
                    m = solver.model()
                    for j, state in enumerate(s[:i+1]):
                            print("-" * 20)
                            print(f"Estado {j}\n")
                            print(f"pc = {m[state['pc']]}")
                            print(f"x = {m[state['x']]}")
                            print(f"y = {m[state['y']]}")
                            print(f"z = {m[state['z']]}")
                    return
                solver.pop()

    s2 = [declare(i+k,n) for i in range(k+1)]

    for i in range(k):
        solver.add(inv(s2[i], a, b, n))
        solver.add(trans(s2[i], s2[i+1], n))
        solver.add((error(s2[i])))

    solver.add(Not(inv(s2[-1], a, b, n)))

    if solver.check() == sat:
        print(f"> Contradição! O passo indutivo não se verifica.")
        m = solver.model()
        for i, state in enumerate(s2):
            print("-" * 20)
            print(f"Estado {i}\n")
            print(f"pc = {m[state['pc']]}")
            print(f"x = {m[state['x']]}")
            print(f"y = {m[state['y']]}")
            print(f"z = {m[state['z']]}")
        return

    print(f"> A propriedade verifica-se por k-indução (k={k}).")

k_induction_safety(declare, init, trans, error, check_inv, 20, 10, 150, 2)
```

```
Verificação de segurança com K = 20
> A propriedade verifica-se por k-indução (k=20).
```