Grupo 14

Trabalho realizado por:

- Beatriz Fernandes Oliveira A91640
- Catarina Martins Sá Quintas A91650

Problema

Consideremos o seguinte programa, que calcula o produto de dois inteiros de 16 bits.

```
        assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
    0: while y > 0:
    1:     if y & 1 == 1:
                   y , r  = y-1 , r+x
    2:     x , y = x<<1  ,  y>>1
    3: assert r == m * n
```

Ao longo deste trabalho, pretende-se provar a terminação deste programa e, ainda, verificar a correção total do mesmo, utilizando duas metodologias: a dos invariantes e a do "single assignment unfolding".

Prova da terminação do programa

Para tal, iremos utilizar um método semelhante ao utilizado no trabalho prático anterior. Assim sendo, necessitamos das variáveis do programa x, y, r, m, n e, ainda, da variável pc que nos permitirá saber qual a instrução que estamos a executar.

```
!pip install z3-solver
```

```
from z3 import *

def declare(i):
  s={}
  s['pc']= Int('pc'+str(i))
  s['x'] = BitVec('x'+str(i),16)
  s['y'] = BitVec('y'+str(i),16)
  s['m'] = BitVec('m'+str(i),16)
  s['n'] = BitVec('n'+str(i),16)
  s['r'] = BitVec('r'+str(i),16)
  return s
```

Observando a pré-condição, conseguimos perceber quais são as condições de inicialização. Assim sendo, o predicado init será da seguinte forma:

-

```
def init(s):
  return And(s['pc'] == 0, s['m'] >= 0, s['n'] >= 0, s['r']==0, s['x']==s['m'], s['
```

Resta-nos agora definir a função de transição. Ora, inicialmente, temos que o valor de pc é 0, a partir desta instrução, só existem dois casos possíveis:

- A condição de ciclo verifica-se, ou seja, entra no ciclo (pc passa para a ter valor de 1);

$$pc == 0 \; and \; y > 0 \; and \; pc' == 1 \; and \; y' == y \; and \; m' == m \; and \; n' == n \; and \; x' == x$$

- A condição do ciclo ser falsa, passando para a pós-condição e termina o programa;

$$pc == 0 \; and \; y <= 0 \; and \; pc' == 3 \; and \; y' == y \; and \; m' == m \; and \; n' == n \; and \; x' ==$$

A partir da primeiro caso, temos, novamente, dois casos possíveis:

- A condição do if verifica-se e, então, altera-se os valores das variáveis r e y;

$$pc == 1 \; and \; (y \wedge 1 == 1) \; and \; pc' == 2 \; and \; y' == y - 1 \; and \; m' == m \; and \; n' == n \; a$$

- A condição do if não se verifica-se, pelo que as variáveis r e y não são alteradas;

$$pc == 1 \; and \; Not(y \wedge 1 == 1) \; and \; pc' == 2 \; and \; y' == y \; and \; m' == m \; and \; n' == n \; a$$

É de realçar que, em ambos os casos anteriores, a variável pc passa para o valor de 2. Neste estado, altera-se o valor das variáveis x e y.

$$pc == 2 \; and \; pc' == 0 \; and \; and \; y' == y >> 1 \; and \; m' == m \; and \; n' == n \; and \; x' == x$$

Após o estado em que a variável pc tem o valor de 2, esta passa a ter variável 0, em que se verifica novamente a condição do ciclo. Falta, apenas, definir o lacete final, para quando a variável pc se encontra com valor de 3 e, como o programa termina, é necessário que se mantenha assim o valor.

$$pc == 3 \; and \; pc' == 3 \; and \; and \; y' == y \; and \; m' == m \; and \; n' == n \; and \; x' == x \; and$$

```
def trans(curr,prox):
  t01 = And(curr['pc'] == 0, curr['y'] > 0, prox['pc'] == 1, prox['y'] == curr['y']

  t03 = And(curr['pc'] == 0, curr['y'] <=0, prox['pc'] == 3, prox['x'] == curr['x']

  t12 = And(curr['pc'] == 1, curr['y'] & 1 == 1, prox['pc'] == 2, prox['x'] == curr

  t1_2 = And(curr['pc'] == 1, Not(curr['y'] & 1==1), prox['pc'] == 2,  prox['y']==c


  t20 = And(curr['pc'] == 2, prox['pc'] == 0, prox['x'] == curr['x'] << 1, prox['y'

  t33 = And(curr['pc'] == 3, prox['pc'] == 3, prox['y']== curr['y'], prox['m']==cur
```

```
  return Or(t01,t03,t12,t1_2,t20,t33)



def kinduction(declare, init, trans, inv, k):
  traco = [declare(i) for i in range(k)]

  # Testar inv para os estados inicias
  s = Solver()
  s.add(init(traco[0]))
  for i in range(k-1):
    s.add(trans(traco[i],traco[i+1]))

  s.add(Or([Not(inv(traco[i])) for i in range(k)]))

  if s.check() == sat:
    m = s.model()
    print("A propriedade falha em pelo menos um  dos",k,"primeiros estados")
    for i in range(i):
      print(i)
      for v in traco[i]:
        print(v,'=',m[traco[i][v]])
    return
  if s.check()==unknown:
    print("O resultado foi inconclusivo")
    return

  # Testar o invariante no passo indutivo
  s = Solver()
  for i in range(k-1):
    s.add(trans(traco[i],traco[i+1]))
    s.add(inv(traco[i]))

  s.add(Not(inv(traco[k-1]))) # No estado final, falha o invariante

  if s.check() == sat:
    m = s.model()
    print("A propriedade falha, no passo de indução, num dos estados")
    for i in range(i):
      print(i)
      for v in traco[0]:
        print(v,'=',m[traco[i][v]])
    return
  if s.check()==unknown:
    print("O resultado foi inconclusivo.")
    return

  print("A propriedade é válida!")
```

- O variante nunca é negativo, ou seja, $G\,(V(s) \geq 0)$

- O variante descresce sempre (estritamente) ou atinge o valor 0, ou seja, $G\,(\forall s'.\, trans(s, s') \rightarrow (V(s') < V(s) \vee V(s') = 0))$
- Quando o variante é 0 verifica-se necessariamente ϕ, ou seja, $G\,(V(s) = 0 \rightarrow \phi(s))$

Ora, observando as diferentes transações e o código do programa, percebemos que o valor do y decresce ao longo da execução do mesmo. Pelo contrário, a variável pc aumenta até chegar ao valor de 2, passando a ter valor de 0 e, de seguida,aumenta, novamente, e assim sucessivamente, até ter o valor de 3. Pelo que, podemos concluir que ambas as variáveis irão fazer parte do variante.

Utilizando o lookhead, podemos, ainda, exigir, apenas, que o variante só decresça a cada l iterações. Ora, de acordo com o programa dado e com o intuito de provar a terminação do programa, vamos considerar um l com valor de 3.

Assim sendo, obtemos assim um variante com a seguinte forma: $V(s) = (v_y - v_{pc}) + 3$, sendo que v_y e v_{pc} , simbolizam os diferentes valores das variáveis y e pc, ao longo da execução.

```
def variante(s):
  return BV2Int(s['y'])-s['pc'] + 3


def positivo(s):
    return variante(s) >= 0


kinduction(declare,init,trans,positivo,3)

     A propriedade é válida!


def utilidade(s):
    return Implies(variante(s) == 0, s["pc"] == 3)


kinduction(declare,init,trans,utilidade,3)

     A propriedade é válida!


def decrescente(s):
    s1 = declare(-1)
    s2 = declare(-2)
    s3 = declare(-3)

    return ForAll(list(s1.values()) + list(s2.values()) + list(s3.values()),
                  Implies(And(trans(s, s1), trans(s1, s2), trans(s2, s3)),Or(varian



kinduction(declare,init,trans,decrescente,4)

     A propriedade é válida!
```

A linguagem dos programas anotados deste programa corresponde a:

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assert inv
havoc x; havoc y; havoc r
(assume y>0 and inv;
(assume (y & 1==1); y=y-1; r= r+x; || assume not (y & 1==1);skip;);
x=x<<1; y=y>>1; assert inv; assume False; || assume not(y>0) and inv;)
assert r == m * n
```

Como o invariante é $x * y + r == m * n \wedge y >= 0$, obtemos a seguinte linguagem dos programas anotados:

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
assert x* y + r == m * n  and y>=0
havoc x; havoc y; havoc r
(assume y>0 and inv;
(assume (y & 1==1); y=y-1; r= r+x; || assume not (y & 1==1);skip;);
x=x<<1; y=y>>1; assert inv; assume False; || assume not(y>0) and inv;)
assert r == m * n
```

Necessitamos agora de calcular as condições de verificação. Para isso, utilizaremos, inicialmente, as regras da técnica WPC.

$$
\begin{aligned}
&[\text{skip}] = True \\
&[\text{assume } \phi] = True \\
&[\text{assert } \phi] = \phi \\
&[x = e] = True \\
&[(C_1 || C_2)] = [C_1] \wedge [C_2] \\
&[\text{skip} \,;\, C] = [C] \\
&[\text{assume } \phi \,;\, C] = \phi \rightarrow [C] \\
&[\text{assert } \phi \,;\, C] = \phi \wedge [C] \\
&[x = e \,;\, C] = [C][e/x] \\
&[(C_1 || C_2) \,;\, C] = [(C_1; C) || (C_2; C)] \\
&[havoc\ x; C] = \forall x.\, [C] \\
&[\ \text{assume } False \ ; \text{assert } \psi \ ] = False \rightarrow \psi = True
\end{aligned}
$$

```
[assume m >= 0 and n >= 0 and r == 0 and x == m and y == n; assert inv;
havoc x; havoc y;havoc r;(assume y>0 and inv; (assume y & 1==1; y=y-1; r= r+x; ||
assume not (y & 1==1);skip;)x=x<<1; y=y>>1; assert inv; assume False; || assume not(y>0) a

= m >= 0 and n >= 0 and r == 0 and x == m and y == n => [assert inv;
havoc x; havoc y; havoc r;(assume y>0 and inv;(assume y & 1==1; y=y-1;
```

```
r= r+x; ||assume not (y & 1==1);) x=x<<1; y=y>>1; assert inv; assume False;
assert r == m * n; || assume not(y>0) and inv; assert r == m * n;)]

= m >= 0 and n >= 0 and r == 0 and x == m and y == n => (inv and
[havoc x; havoc y; havoc r; (assume y>0 and inv;
(assume y & 1==1; y=y-1; r= r+x;|| assume not (y & 1==1);) x=x<<1; y=y>>1; assert inv; ass
assume not(y>0) and inv;assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n => (inv and
forall x forall y forall r [(assume y>0 and inv; (assume y & 1==1; y=y-1; r= r+x; ||
assume not (y & 1==1);)x=x<<1; y=y>>1; assert inv; assume False;
assert r == m * n; || assume not(y>0) and inv;assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n => (inv and
forall x forall y forall r(y>0 and inv => [(assume y & 1==1; y=y-1; r= r+x; ||
assume not (y & 1==1);)x=x<<1; y=y>>1; assert inv;
assume False; assert r == m * n; || assume not(y>0) and inv; assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>(inv and forall x
forall y forall r (y>0 and inv => [(assume y & 1==1; y=y-1; r= r+x;
x=x<<1; y=y>>1; assert inv; assume False;assert r == m * n;|| assume not (y & 1==1); x=x<<
assume False; assert r == m * n;)|| assume not(y>0) and inv; assert r == m * n;)])

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>(inv and
forall x forall y forall y (y>0 & inv => [((assume y and 1==1; (assert inv; assume False;
[x<<1/x][r+x/r][y-1/y])|| (assume not (y & 1==1); (assert inv;assume False;
assert r == m * n;)[y>>1/y][x<<1/x])|| assume not(y>0) and inv; assert r == m * n;)])

= m>=0 and n >= 0 and r == 0 and x == m and y == n =>(inv and
forall x forall y forall r (y>0 and inv => ((y & 1==1 => (inv and (False => r == m * n;))
[y>>1/y][x<<1/x][r+x/r][y-1/y]) and [(assume not (y & 1==1); (assert inv;
assume False; assert r == m * n;)[y>>1/y] [x<<1/x]) and assume not(y>0) and inv; assert r

= m >= 0 and n >= 0 and r == 0 and x == m and y == n =>(inv and forall x \forall y forall
((y & 1==1 => (inv and (False => r == m * n))[y>>1/y][x<<1/x][r+x/r][y-1/y]) and
(not (y & 1==1) =>(inv and (False => r == m * n))[y>>1/y][x<<1/x]) and
[assume not(y>0) and inv; assert r == m * n;)])

= m>=0 and n>=0 and r==0 and x==m and y==n => inv and forall x forall y
forall r (y>0 and inv => (y & 1==1 => (inv and (False => r == m * n))
[y>>1/y][x<<1/x][r+x/r][y-1/y]) and (not (y & 1==1) =>(inv and
(False => r == m * n))[y>>1/y][x<<1/x]) and not(y>0) and inv => r == m * n)

=  m>=0 and n>=0 and r==0 and x==m and y==n => inv and forall x forall y
forall r (y>0 and inv => (y & 1==1 => (inv[y>>1/y][x<<1/x][r+x/r][y-1/y])
and (not (y & 1==1) =>(inv[y>>1/y][x<<1/x]) and not(y>0) and inv => r == m * n)
```

Temos que o invariante corresponde a $x * y + r == m * n \wedge y >= 0$. Logo, ficamos a seguinte fórmula:

```
m>=0 and n>=0 and r==0 and x==m and y==n => x* y + r == m * n and y>=0 and
forall x forall y forall r(y>0 and x* y + r == m * n and y>=0 =>
(y & 1==1 => (x* y + r == m * n and y>=0)[y>>1/y][x<<1/x][r+x/r][y-1/y])
and (not (y & 1==1) => (x* y + r == m * n and y>=0)[y>>1/y][x<<1/x]) and
not(y>0) and x* y + r == m * n and y>=0 => r == m * n)

= m>=0 and n>=0 and r==0 and x==m and y==n => x* y + r == m * n  and y>=0 and
forall x forall y forall r (y>0 and x* y + r == m * n  =>
(y & 1==1 => (x* y + r == m * n and y>=0)[y>>1/y][x<<1/x][r+x/r][y-1/y]) and
(not (y & 1==1) => (x* y + r == m * n and y>=0)[y>>1/y][x<<1/x]) and
not(y>0) and x* y + r == m * n  and y>=0 => r == m * n)
```

```
def prove(f):
  s = Solver()
  s.add(Not(f))
  if s.check() == unsat:
    print("Proved")
  else:
    print("Failed to prove")
    m = s.model()
    for v in m:
      print(v,'=', m[v])


x, y, r, m, n = BitVecs('x y r m n', 10)

inv = And(x*y + r == m*n, y>=0)
pre = And(m >= 0, n >= 0, r == 0, x == m, y == n)
pos = And(r == m * n)

z = y & 1==1
s = And(y>0, x*y+r == m*n)

p1=Implies(z,substitute(substitute(substitute(substitute(inv,(y,y>>1)),(x,x<<1)),(r
p2=Implies(Not(z),substitute(substitute(inv,(y,y>>1)),(x,x<<1)))
p3=Implies(And(Not(y>0),inv),pos)
Vc=Implies(And(y > 0, inv),And(p1,p2))

prog = Implies(pre, And(inv, ForAll([x, y,r], Vc),p3))

prove(prog)
```

```
    Proved
```

Vamos prosseguir com o cálculo da condições do programa, utilizando a técnica das SPC.

$$
\begin{aligned}
&[\text{skip}] = True \\
&[\text{assume } \phi] = \phi \\
&[\text{assert } \phi] = \phi \\
&[x = e] = (x = e) \\
&[(C_1 || C_2)] = [C_1] \vee [C_2] \\
&[C \text{ ; skip }] = [C] \\
&[C \text{ ; assume } \phi] = [C] \wedge \phi \\
&[C \text{ ; assert } \phi] = [C] \rightarrow \phi \\
&[C \text{ ; } x = e] = [C] \wedge (x = e) \\
&[C \text{ ; } (C_1 || C_2)] = [(C; C_1) || (C; C_2)]
\end{aligned}
$$

- Comecemos por substituir o ciclo por um if e, a partir, disso iremos alongar o mesmo o número de vezes necessárias. Ora, como estamos a utilizar números com 16 Bits e uma das operações que utilizamos são os shifts, não podemos realizar mais do 16 shifts. Ou seja, no máximo, só se podem iterar o ciclo 16 vezes.

```
assume m >= 0 and n >= 0 and r == 0 and x == m and y == n
    if(y > 0):
      if y & 1 == 1:
        y , r  = y-1 , r+x
      x , y = x<<1  ,  y>>1
      if(y > 0):
        if y & 1 == 1:
          y , r  = y-1 , r+x
        x , y = x<<1  ,  y>>1
        if(y > 0):
          if y & 1 == 1:
            y , r  = y-1 , r+x
          x , y = x<<1  ,  y>>1
          if(y > 0):
            if y & 1 == 1:
              y , r  = y-1 , r+x
            x , y = x<<1  ,  y>>1
           if(y > 0):
             if y & 1 == 1:
               y , r  = y-1 , r+x
             x , y = x<<1  ,  y>>1
             if(y > 0):
               if y & 1 == 1:
                 y , r  = y-1 , r+x
               x , y = x<<1  ,  y>>1
               if(y > 0):
                 if y & 1 == 1:
                   y , r  = y-1 , r+x
```

```
                    x , y = x<<1  ,  y>>1
                    if(y > 0):
                      if y & 1 == 1:
                        y , r  = y-1 , r+x
                      x , y = x<<1  ,  y>>1
                      if(y > 0):
                        if y & 1 == 1:
                          y , r  = y-1 , r+x
                        x , y = x<<1  ,  y>>1
                        if(y > 0):
                          if y & 1 == 1:
                            y , r  = y-1 , r+x
                          x , y = x<<1  ,  y>>1
                          if(y > 0):
                            if y & 1 == 1:
                              y , r  = y-1 , r+x
                            x , y = x<<1  ,  y>>1
                            if(y > 0):
                              if y & 1 == 1:
                                y , r  = y-1 , r+x
                              x , y = x<<1  ,  y>>1
                              if(y > 0):
                                if y & 1 == 1:
                                  y , r  = y-1 , r+x
                                x , y = x<<1  ,  y>>1
                                if(y > 0):
                                  if y & 1 == 1:
                                    y , r  = y-1 , r+x
                                  x, y = x<<1  ,  y>>1
                                  if(y > 0):
                                    if y & 1 == 1:
                                      y , r  = y-1 , r+x
                                    x , y = x<<1  ,  y>>1
assert r == m * n
```

- De seguida, vamos escrever a versão SA do programa

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
    if(y0 > 0):
      if y0 & 1 == 1:
        y1 , r1  = y0-1 , r0+x0
      else:
```

```
  y1=y0
  r1=r0
x1 , y2 = x0<<1  ,  y1>>1

if(y2 > 0):
  if y2 & 1 == 1:
    y3 , r2  = y2-1 , r1+x1
  else:
    y3=y2
    r2=r1
  x2 , y4 = x1<<1  ,  y3>>1

  if(y4 > 0):
    if y4 & 1 == 1:
      y5 , r3  = y4-1 , r2+x2
    else:
      y5=y4
      r3=r2
    x3 , y6 = x2<<1  ,  y5>>1

    if(y6 > 0):
      if y6 & 1 == 1:
        y7 , r4  = y6-1 , r3+x3
      else:
        y7=y6
        r4=r3
      x4 , y8 = x3<<1  ,  y7>>1

      if(y8 > 0):
        if y8 & 1 == 1:
          y9 , r5  = y8-1 , r4+x4
        else:
          y9=y8
          r5=r6
        x5 , y10 = x4<<1  ,  y9>>1

        if(y10 > 0):
          if y10 & 1 == 1:
            y11 , r6  = y10-1 , r5+x5
          else:
            y11=y10
            r6=r5
          x6 , y12 = x5<<1  ,  y11>>1

          if(y12 > 0):
            if y12 & 1 == 1:
```

```
  y13 , r7  = y12-1 , r6+x6
else:
  y13=y12
  r7=r6
x7 , y14 = x6<<1  ,  y13>>1

if(y14 > 0):
  if y14 & 1 == 1:
    y15 , r8  = y14-1 , r7+x7
  else:
    y15=y14
    r8=r7
  x8 , y16 = x7<<1  ,  y15>>1

  if(y16 > 0):
    if y16 & 1 == 1:
      y17 , r9  = y16-1 , r8+x8
    else:
      y17=y16
      r9=r8
    x9 , y18 = x8<<1  ,  y17>>1

    if(y18 > 0):
      if y18 & 1 == 1:
        y19 , r10  = y18-1 , r9+x9
      else:
        y19=y18
        r10=r9
      x10 , y20 = x9<<1  ,  y19>>1

      if(y20 > 0):
        if y20 & 1 == 1:
          y21 , r11  = y20-1 , r10+x10
        else:
          y21=y20
          r11=r10
        x11 , y22 = x10<<1  ,  y21>>1

        if(y22 > 0):
          if y22 & 1 == 1:
            y23 , r12  = y22-1 , r11+x11
          else:
            y23=y22
            r12=r11
          x12 , y24 = x11<<1  ,  y23>>1
```

```
  if(y24 > 0):
    if y24 & 1 == 1:
      y25 , r13  = y24-1 , r12+x12
    else:
      y25=y24
      r13=r12
    x13 , y26 = x12<<1  ,  y25>>1

    if(y26 > 0):
      if y26 & 1 == 1:
        y27 , r14  = y26-1 , r13+x13
      else:
        y27=y26
        r14=r13
      x14 , y28 = x13<<1  ,  y27>>1

      if(y28 > 0):
        if y28 & 1 == 1:
          y29 , r15  = y28-1 , r14+x14
        else:
          y29=y28
          r15=r14
        x15, y30 = x14<<1  ,  y29>>1

        if(y30 > 0):
          if y30 & 1 == 1:
            y31 , r16  = y30-1 , r15+x15
          else:
            y31=y30
            r16=r15
          x16 , y32 = x15<<1  ,  y31>>1
          assert not(y32>0)

        else:
          r16=r15

      else:
        r16=r14

    else:
      r16=r13

  else:
    r16=r12

else:
```

```
                                r16=r11

                            else:
                              r16=r10

                          else:
                            r16=r9

                        else:
                          r16=r8

                      else:
                        r16=r7

                    else:
                      r16=r6

                  else:
                    r16=r5

                else:
                  r16=r4

              else:
                r16=r3

            else:
              r16=r2

          else:
            r16=r1

        else:
          r16=r0

    assert r16 == m0 * n0
```

- Procederemos à tradução do programa sem ciclos para a linguagem de fluxos.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
        (assume (y0 > 0)
          (assume  y0 & 1 == 1
            y1 , r1  = y0-1 , r0+x0
          || assume (not (y0 & 1 == 1))
            y1=y0
            r1=r0)
```

```
x1 , y2 = x0<<1  ,  y1>>1

(assume (y2 > 0)
  (assume  y2 & 1 == 1
    y3 , r2  = y2-1 , r1+x1
  || assume (not (y2 & 1 == 1))
    y3=y2
    r2=r1)
  x2 , y4 = x1<<1  ,  y3>>1

  (assume (y4 > 0)
    (assume  y4 & 1 == 1
      y5 , r3  = y4-1 , r2+x2
    || assume (not (y4 & 1 == 1))
      y5=y4
      r3=r2)
    x3 , y6 = x2<<1  ,  y5>>1

    (assume (y6 > 0)
      (assume  y6 & 1 == 1
        y7 , r4  = y6-1 , r3+x3
      || assume (not (y6 & 1==1))
        y7=y6
        r4=r3)
      x4 , y8 = x3<<1  ,  y7>>1

      (assume (y8 > 0)
        (assume  y8 & 1 == 1
          y9 , r5  = y8-1 , r4+x4
        || assume (not (y8 & 1==1))
          y9=y8
          r5=r6)
        x5 , y10 = x4<<1  ,  y9>>1

        (assume (y10 > 0)
          (assume  y10 & 1 == 1
            y11 , r6  = y10-1 , r5+x5
          || assume (not (y10 & 1==1))
            y11=y10
            r6=r5)
          x6 , y12 = x5<<1  ,  y11>>1

          (assume (y12 > 0)
            (assume  y12 & 1 == 1
              y13 , r7  = y12-1 , r6+x6
            || assume (not (y12 & 1==1))
```

```
  y13=y12
  r7=r6)
x7 , y14 = x6<<1  ,  y13>>1

(assume (y14 > 0)
  (assume  y14 & 1 == 1
    y15 , r8  = y14-1 , r7+x7
  || assume (not (y14 & 1==1))
    y15=y14
    r8=r7)
  x8 , y16 = x7<<1  ,  y15>>1

  (assume (y16 > 0)
    (assume  y16 & 1 == 1
      y17 , r9  = y16-1 , r8+x8
    || assume (not (y16 & 1==1))
      y17=y16
      r9=r8)
    x9 , y18 = x8<<1  ,  y17>>1

    (assume (y18 > 0)
      (assume  y18 & 1 == 1
        y19 , r10  = y18-1 , r9+x9
      || assume (not (y18 & 1==1))
        y19=y18
        r10=r9)
      x10 , y20 = x9<<1  ,  y19>>1

      (assume (y20 > 0)
        (assume  y20 & 1 == 1
          y21 , r11  = y20-1 , r10+x10
        || assume (not (y20 & 1==1))
          y21=y20
          r11=r10)
        x11 , y22 = x10<<1  ,  y21>>1

        (assume (y22 > 0)
          (assume  y22 & 1 == 1
            y23 , r12  = y22-1 , r11+x11
          || assume (not (y22 & 1==1))
            y23=y22
            r12=r11)
          x12 , y24 = x11<<1  ,  y23>>1

          (assume (y24 > 0)
            (assume  y24 & 1 == 1
```

```
      y25 , r13  = y24-1 , r12+x12
    || assume (not (y24 & 1==1))
      y25=y24
      r13=r12)
    x13 , y26 = x12<<1  ,  y25>>1

    (assume (y26 > 0)
      (assume  y26 & 1 == 1
        y27 , r14  = y26-1 , r13+x13
      ||assume (not (y26 & 1==1))
        y27=y26
        r14=r13)
      x14 , y28 = x13<<1  ,  y27>>1

      (assume (y28 > 0)
        (assume  y28 & 1 == 1
          y29 , r15  = y28-1 , r14+x14
        || assume (not (y28 & 1==1))
          y29=y28
          r15=r14)
        x15, y30 = x14<<1  ,  y29>>1

        (assume (y30 > 0)
          (assume  y30 & 1 == 1
            y31 , r16  = y30-1 , r15+x15
           || assume (not (y30 & 1==1))
            y31=y30
            r16=r15)
          x16 , y32 = x15<<1  ,  y31>>1
          assert not(y32>0)
         ||
        assume (not (y30 > 0))
          r16=r15)
       ||
      assume (not (y28 > 0))
        r16=r14)
     ||
    assume (not (y26 > 0))
      r16=r13)
   ||
  assume (not (y24 > 0))
    r16=r12)
 ||
assume (not (y22 > 0))
  r16=r11)
||
```

```
                        assume (not (y20 > 0))
                          r16=r10)
                        ||
                       assume (not (y18 > 0))
                         r16=r9)
                      ||
                     assume (not (y16 > 0))
                       r16=r8)
                    ||
                   assume (not (y14 > 0))
                     r16=r7)
                 r16=r5)
               ||
              assume (not (y8 > 0))
                r16=r4)
             ||
            assume (not (y6 > 0))
              r16=r3)
           ||
          assume (not (y4 > 0))
            r16=r2)
         ||
        assume (not (y2 > 0))
          r16=r1)
       ||
      assume (not (y0 > 0))
        r16=r0)

  assert r17 == m0 * n0
```

- Consideremos, apenas, os fluxos não-deterministas deste programa, com o intuito de simplificar a verificação.

1. Fluxo que corresponde a verificar todas as condições do if.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
        (assume (y0 > 0)
          assume  y0 & 1 == 1:
            y1 , r1  = y0-1 , r0+x0
          x1 , y2 = x0<<1  ,  y1>>1

          (assume (y2 > 0):
            assume  y2 & 1 == 1:
              y3 , r2  = y2-1 , r1+x1
            x2 , y4 = x1<<1  ,  y3>>1
```

```
(assume (y4 > 0):
  assume  y4 & 1 == 1:
    y5 , r3  = y4-1 , r2+x2
  x3 , y6 = x2<<1  ,  y5>>1

  (assume (y6 > 0):
    assume  y6 & 1 == 1:
      y7 , r4  = y6-1 , r3+x3
    x4 , y8 = x3<<1  ,  y7>>1

    (assume (y8 > 0):
      assume  y8 & 1 == 1:
        y9 , r5  = y8-1 , r4+x4
      x5 , y10 = x4<<1  ,  y9>>1

      (assume (y10 > 0):
        assume  y10 & 1 == 1:
          y11 , r6  = y10-1 , r5+x5
        x6 , y12 = x5<<1  ,  y11>>1

        (assume (y12 > 0):
          assume  y12 & 1 == 1:
            y13 , r7  = y12-1 , r6+x6
          x7 , y14 = x6<<1  ,  y13>>1

          (assume (y14 > 0):
            assume  y14 & 1 == 1:
              y15 , r8  = y14-1 , r7+x7
            x8 , y16 = x7<<1  ,  y15>>1

            (assume (y16 > 0):
              assume  y16 & 1 == 1:
                y17 , r9  = y16-1 , r8+x8
              x9 , y18 = x8<<1  ,  y17>>1

              (assume (y18 > 0):
                assume  y18 & 1 == 1:
                  y19 , r10  = y18-1 , r9+x9
                x10 , y20 = x9<<1  ,  y19>>1

                (assume (y20 > 0):
                  assume  y20 & 1 == 1:
                    y21 , r11  = y20-1 , r10+x10
                  x11 , y22 = x10<<1  ,  y21>>1

                  (assume (y22 > 0):
```

```
                                assume  y22 & 1 == 1:
                                  y23 , r12  = y22-1 , r11+x11
                                x12 , y24 = x11<<1  ,  y23>>1

                                (assume (y24 > 0):
                                  assume  y24 & 1 == 1:
                                    y25 , r13  = y24-1 , r12+x12
                                  x13 , y26 = x12<<1  ,  y25>>1

                                  (assume (y26 > 0):
                                    assume  y26 & 1 == 1:
                                      y27 , r14  = y26-1 , r13+x13
                                    x14 , y28 = x13<<1  ,  y27>>1

                                    (assume (y28 > 0):
                                      assume  y28 & 1 == 1:
                                        y29 , r15  = y28-1 , r14+x14
                                      x15, y30 = x14<<1  ,  y29>>1

                                      (assume (y30 > 0):
                                        assume  y30 & 1 == 1:
                                          y31 , r16  = y30-1 , r15+x15
                                        x16 , y32 = x15<<1  ,  y31>>1
                                        assert not(y32>0)))))))))))))))))

 assert r17 == m0 * n0
```

2. Fluxo que corresponde a não verificar o primeira if.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
assume (not (y0 > 0)); r16=r0;
assert not(y32>0);
assert r17 == m0 * n0
```

3. Fluxo que corresponde a verificar o primeiro if e a não verificar a condição dos if seguintes.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
(assume (y0 > 0);
assume (not (y0 & 1 == 1)); y1=y0; r1=r0; x1=x0<<1; y2 = y1>>1;
assume (not (y30 > 0)); r16=r15;
assume (not (y28 > 0)); r16=r14;
assume (not (y26 > 0)); r16=r13;
assume (not (y24 > 0)); r16=r12;
assume (not (y22 > 0)); r16=r11;
assume (not (y20 > 0)); r16=r10;
```

```
assume (not (y18 > 0)); r16=r9;
assume (not (y16 > 0)); r16=r8;
assume (not (y14 > 0)); r16=r7;
assume (not (y12 > 0)); r16=r6;
assume (not (y10 > 0)); r16=r5;
assume (not (y8 > 0)); r16=r4;
assume (not (y6 > 0)); r16=r3;
assume (not (y4 > 0)); r16=r2;
assume (not (y2 > 0)); r16=r1)
assert r17 == m0 * n0
```

- Temos agora que, para cada um destes fluxos, a sua respetiva denotação lógica(VC).

1. Fluxo que corresponde a verificar todas as condições do if.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
        (assume (y0 > 0)
          assume  y0 & 1 == 1:
            y1 , r1  = y0-1 , r0+x0
          x1 , y2 = x0<<1  ,  y1>>1

          (assume (y2 > 0):
            assume  y2 & 1 == 1:
              y3 , r2  = y2-1 , r1+x1
            x2 , y4 = x1<<1  ,  y3>>1

            (assume (y4 > 0):
              assume  y4 & 1 == 1:
                y5 , r3  = y4-1 , r2+x2
              x3 , y6 = x2<<1  ,  y5>>1

              (assume (y6 > 0):
                assume  y6 & 1 == 1:
                  y7 , r4  = y6-1 , r3+x3
                x4 , y8 = x3<<1  ,  y7>>1

                (assume (y8 > 0):
                  assume  y8 & 1 == 1:
                    y9 , r5  = y8-1 , r4+x4
                  x5 , y10 = x4<<1  ,  y9>>1

                  (assume (y10 > 0):
                    assume  y10 & 1 == 1:
                      y11 , r6  = y10-1 , r5+x5
                    x6 , y12 = x5<<1  ,  y11>>1
```

```
(assume (y12 > 0):
  assume  y12 & 1 == 1:
    y13 , r7  = y12-1 , r6+x6
  x7 , y14 = x6<<1  ,  y13>>1

  (assume (y14 > 0):
    assume  y14 & 1 == 1:
      y15 , r8  = y14-1 , r7+x7
    x8 , y16 = x7<<1  ,  y15>>1

    (assume (y16 > 0):
      assume  y16 & 1 == 1:
        y17 , r9  = y16-1 , r8+x8
      x9 , y18 = x8<<1  ,  y17>>1

      (assume (y18 > 0):
        assume  y18 & 1 == 1:
          y19 , r10  = y18-1 , r9+x9
        x10 , y20 = x9<<1  ,  y19>>1

        (assume (y20 > 0):
          assume  y20 & 1 == 1:
            y21 , r11  = y20-1 , r10+x10
          x11 , y22 = x10<<1  ,  y21>>1

          (assume (y22 > 0):
            assume  y22 & 1 == 1:
              y23 , r12  = y22-1 , r11+x11
            x12 , y24 = x11<<1  ,  y23>>1

            (assume (y24 > 0):
              assume  y24 & 1 == 1:
                y25 , r13  = y24-1 , r12+x12
              x13 , y26 = x12<<1  ,  y25>>1

              (assume (y26 > 0):
                assume  y26 & 1 == 1:
                  y27 , r14  = y26-1 , r13+x13
                x14 , y28 = x13<<1  ,  y27>>1

                (assume (y28 > 0):
                  assume  y28 & 1 == 1:
                    y29 , r15  = y28-1 , r14+x14
                  x15, y30 = x14<<1  ,  y29>>1

                  (assume (y30 > 0):
```

```
                                        assume  y30 & 1 == 1:
                                          y31 , r16  = y30-1 , r15+x15
                                        x16 , y32 = x15<<1  ,  y31>>1
                                        assert not(y32>0)))))))))))))))))

 assert r17 == m0 * n0
```

2. Fluxo que corresponde a não verificar o primeira if.

```
[assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;
assume (not (y0 > 0)); r16=r0;
assert not(y32>0);
assert r17 == m0 * n0; ]
=
[assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;
assume (not (y0 > 0)); r16=r0;] => not(y32>0) => r17 == m0 * n0;
=
[assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;
assume (not (y0 > 0))] and r16=r0 => not(y32>0) => r17 == m0 * n0;
=
[assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0;]
and not (y0 > 0) and r16=r0 => not(y32>0) => r17 == m0 * n0;
=
m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
and not (y0 > 0) and r16=r0 => not(y32>0) => r17 == m0 * n0;
```

3. Fluxo que corresponde a verificar o primeiro if e a não verificar a condição dos if seguintes.

```
assume m0 >= 0 and n0 >= 0 and r0 == 0 and x0 == m0 and y0 == n0
(assume (y0 > 0);
assume (not (y0 & 1 == 1)); y1=y0; r1=r0; x1=x0<<1; y2 = y1>>1;
assume (not (y30 > 0)); r16=r15;
assume (not (y28 > 0)); r16=r14;
assume (not (y26 > 0)); r16=r13;
assume (not (y24 > 0)); r16=r12;
assume (not (y22 > 0)); r16=r11;
assume (not (y20 > 0)); r16=r10;
assume (not (y18 > 0)); r16=r9;
assume (not (y16 > 0)); r16=r8;
assume (not (y14 > 0)); r16=r7;
assume (not (y12 > 0)); r16=r6;
assume (not (y10 > 0)); r16=r5;
assume (not (y8 > 0)); r16=r4;
assume (not (y6 > 0)); r16=r3;
assume (not (y4 > 0)); r16=r2;
```

```
assume (not (y2 > 0)); r16=r1)
assert r17 == m0 * n0
```

Temos, agora, que provar a validade destes fluxos. Utilizando o “single assignment unfolding” e considerando um parâmetro limite N que irá controlar o número de iterações. Vamos, ainda, considerar a nova pré-condição:

$$assume\ m >= 0\ and\ n >= 0\ and\ r == 0\ and\ x == m\ and\ y == n\ and\ n < N\ and\ m <$$

```
!pip install pysmt


import pysmt.shortcuts as ps
import pysmt.typing as pt

def prime(v):
  return ps.Symbol("next(%s)" % v.symbol_name(), v.symbol_type())

def fresh(v):
  return ps.FreshSymbol(typename=v.symbol_type(),template=v.symbol_name()+"_%d")

class EPU(object):
    """deteção de erro"""

    def __init__(self, variables, init , trans, error, sname="z3"):

        self.variables = variables
        self.init  = init
        self.error = error
        self.trans = trans


        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [self.error]       # inializa com uma só frame: a situação de

        self.solver = ps.Solver(name=sname)
        self.solver.add_assertion(self.init)     # adiciona o estado inicial como u

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        T = self.trans.substitute(dict(zip(self.prime_variables,freshs)))
        F = self.frames[-1].substitute(dict(zip(self.variables,freshs)))
        self.frames.append(ps.Exists(freshs, ps.And(T, F)))

    def unroll(self,bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: tentativas ultrapassam o limite %d "%bound)
                break
            elif self.solver.solve(self.frames):
                self.new_frame()
                n += 1
```

```
            else:
                print("sucesso: tentativa %d "%n)
                break

class Cycle(EPU):
    def __init__(self,variables,pre,pos,control,body,sname="z3"):
        init  = pre
        trans = ps.And(control,body)
        error = ps.Or(control,ps.Not(pos))
        super().__init__(variables, init, trans, error, sname)




N = ps.BV(((2**bits)-1), 16)

# O ciclo
m = ps.Symbol("m", pt.BV16)
n = ps.Symbol("n", pt.BV16)
x = ps.Symbol("x", pt.BV16)
y = ps.Symbol("y", pt.BV16)
r = ps.Symbol("r", pt.BV16)

variables = [m, n, x, y, r]

pre = ps.And(n < N, m < N, m >= ps.BVZero(16), n >= ps.BVZero(16), r.Equals(ps.BVZe
pos = r.Equals(m * n)
cond = y > ps.BVZero(16)

ifBody = ps.And(
    ps.Implies(ps.Equals(ps.BVAnd(y, ps.BVOne(16)), ps.BVOne(16)), ps.And(
        ps.Equals(prime(y), ps.BVSub(y, ps.BVOne(16))),
        ps.Equals(prime(x), ps.BVAdd(r, x))
    )),
    ps.Implies(ps.Not(ps.Equals(ps.BVAnd(y, ps.BVOne(16)), ps.BVOne(16))), ps.And(
            ps.Equals(prime(y), y),
            ps.Equals(prime(x), x)
    ))
)

trans = ps.And(
    ifBody,
    ps.Equals(prime(x), ps.BVLShl(x, ps.BVOne(16))),
    ps.Equals(prime(y), ps.BVLShr(y, ps.BVOne(16)))
)


W = Cycle(variables,pre,pos,cond,trans)
W.unroll(16)
```

```
 sucesso: tentativa 2
```