# Lógica Computacional TP4

**Realizado por: Miguel Gonçalves a90416 João Nogueira a87973**

Universidade do Minho
Escola de Ciências

# Trabalho 4

**Considere o seguinte programa, em Python anotado, para multiplicação de dois inteiros de precisão limitada a 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
```

1. **Prove por indução a terminação deste programa**
2. **Pretende-se verificar a correção total deste programa usando a metodologia dos invariantes e a metodologia do “single assignment unfolding”. Para isso:**

> > **a) Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa.**
> >
> > **b) Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.**
> >
> > **c) Construa a definição iterativa do “single assignment unfolding” usando um parâmetro limite $N$ e aumentando a pré-condição com a condição: $(n < N) \land (m < N) \backslash$ O número de iterações vai ser controlado por este parâmetro $N$**

# Resolução:

**1. Prove por indução a terminação deste programa**

**Para verificar a terminação deste programa vamos usar a técnica do model checking, começamos então por definir o estado inicial:**

> $$m >= 0 \land n >= 0 \land r == 0 \land x == m \land y == n \land \text{pc}$$

**Passamos agora, a mostrar as transições possíveis no FOTS, estas são caracterizadas pelo seguinte predicado:**

> $$(\text{pc} = 0 \land \text{pc}' = 1 \land y <= 0 \land m' = m \land n' = n \land r' = r \land x' = x \land y' = y)$$
> $$\lor$$

$$(\text{pc} = 0 \wedge \text{pc}' = 0 \wedge y > 0 \wedge m' = m \wedge n' = n \wedge r' = r + x \wedge x' = x << 1 \wedge y' = (y-1) >> 1 \wedge \neg(y = 0))$$
$$\vee$$
$$(\text{pc} = 0 \wedge \text{pc}' = 0 \wedge y > 0 \wedge m' = m \wedge n' = n \wedge r' = r \wedge x' = x << 1 \wedge y' = y >> 1 \wedge (y = 0))$$
$$\vee$$
$$(\text{pc} = 1 \wedge \text{pc}' = 1 \wedge m' = m \wedge n' = n \wedge r' = r \wedge x' = x \wedge y' = y)$$

In [4]:

```
from z3 import *
from random import randint
def declare(i):
    state = {}
    state['pc'] = Int('pc'+str(i))
    state['m'] = BitVec('m'+str(i),16)
    state['n'] = BitVec('n'+str(i),16)
    state['r'] = BitVec('r'+str(i),16)
    state['x'] = BitVec('x'+str(i),16)
    state['y'] = BitVec('y'+str(i),16)
    return state

def init(state):
    return And(state['m']==randint(0,20),state['n']==randint(0,20),
               state['r']==0,state['x']==state['m'],
               state['y']==state['n'],state['pc']==0)

def trans(curr,prox):
    # define igualdade dos valores do estado anterior para o seguinte #
    ti=And(prox['m']==curr['m'],prox['n']==curr['n'],prox['r']==curr['r'],prox['x']==curr
['x'],prox['y']==curr['y'])

    # as possiveis tranciçoes #
    t1=And(curr['pc']==0,prox['pc']==1,curr['y']<=0,ti)
    t2=And(curr['pc']==0,prox['pc']==0,curr['y']>0,prox['m']==curr['m'],prox['n']==curr[
'n'],prox['r']==curr['r']+curr['x'],
          prox['x']==(curr['x']<<1),prox['y']==((curr['y']-1)>>1),Not(curr['y']==0))
    t3=And(curr['pc']==0,prox['pc']==0,curr['y']>0,prox['m']==curr['m'],prox['n']==curr[
'n'],prox['r']==curr['r'],
          prox['x']==(curr['x']<<1),prox['y']==(curr['y']>>1),curr['y']==0)
    t4=And(curr['pc']==1,prox['pc']==1,ti)
    return Or(t1,t2,t3,t4)
```

In [5]:

```
def gera_traco(declare,init,trans,k):
    s = Solver()
    state =[declare(i) for i in range(k)]
    s.add(init(state[0]))
    for i in range(k-1):
        s.add(trans(state[i],state[i+1]))
    if s.check()==sat:
        m=s.model()
        for i in range(k):
            print(i)
            for x in state[i]:
                print(x,"=",m[state[i][x]])
gera_traco(declare,init,trans,7)
```

```
0
pc = 0
m = 12
n = 5
r = 0
x = 12
y = 5
1
pc = 0
m = 12
```

```
n = 5
r = 12
x = 24
y = 2
2
pc = 0
m = 12
n = 5
r = 36
x = 48
y = 0
3
pc = 1
m = 12
n = 5
r = 36
x = 48
y = 0
4
pc = 1
m = 12
n = 5
r = 36
x = 48
y = 0
5
pc = 1
m = 12
n = 5
r = 36
x = 48
y = 0
6
pc = 1
m = 12
n = 5
r = 36
x = 48
y = 0
```

**Para provar a terminação deste programa, modelamos em lógica temporal linear LT a seguinte propriedade:**

> $F(pc = 1)$

In [6]:

```
def termina(state):
    return (state['pc'] == 1)
```

In [7]:

```
def bmc_eventually(declare,init,trans,prop,K):
    for k in range(1,K+1):
        s = Solver()
        traco = {}
        for i in range(k):
          traco[i] = declare(i)
        s.add(init(traco[0]))
        for i in range(k-1):
          s.add(trans(traco[i],traco[i+1]))

        for i in range(k):
          s.add(Not(prop(traco[i])))
        s.add(Or([trans(traco[k-1], traco[i]) for i in range(k)]))

        status = s.check()
        if status == sat:
          m = s.model()
          for i in range(k):
```

```
            print(i)
            for v in traco[i]:
              print(v,"=",m[traco[i][v]])
          return
    print("A propriedade pode ser verdade")

bmc_eventually(declare,init,trans,termina,16)
```

```
A propriedade pode ser verdade
```

**2a) Codifique usando a LPA (linguagem de programas anotadas) a forma recursiva deste programa**

**Seguindo a notação da linguagem de programas anotadas, chegamos à seguinte forma:**

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

**2b) Proponha o invariante mais fraco que assegure a correção, codifique-o em SMT e prove a correção.**

**Analisando o programa, chegamos ao seguinte invariante:**

```
Invariante: y >= 0 and y <= n and x == m + r
```

**Como no enunciado do problema pede o invariante mais fraco, então temos que:**

```
Invariante: y >= 0
```

**Para provar a correção deste programa vamos usar o método *havoc*. Começamos então por proceder à sua tradução para a linguagem de fluxos com havocs.**

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

```
assume pre;
assert inv;
havoc r,havoc x,havoc y;
((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||assume not(y and 1==1);)
  ;x==x<<1;y==y>>1;assert inv; assume False;assert pos;)||
 (assume not(y>0) and inv;assert pos;))

#==
pre->(inv and (havoc r,havoc x,havoc y;
                ((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||assume no
t(y and 1==1);)
                 ;x==x<<1;y==y>>1;assert inv; assume False;assert pos;)
                ||assume not(y>0) and inv;assert pos;)))

#== havoc

pre->(inv and ForAll([r,x,y],
                ((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||assume no
t(y and 1==1););
                 x==x<<1;y==y>>1;assert inv; assume False;assert pos;)
```

```
                 ||assume not(y>0) and inv;assert pos;)))

#== false->..=TRUE

pre->(inv and ForAll([r,x,y],
                ((assume y>0 and inv;((assume y and 1==1;y==y-1;r==r+x)||assume no
t(y and 1==1);)
                 x==x<<1;y==y>>1;assert inv;)))
                 and assume not(y>0) and inv;assert pos;)

#== transformaçao

pre->(inv and ForAll([r,x,y],
                (y>0 and inv->(((y and 1==1)-> inv;[y>>1/y][x<<1/x][r+x/r][y-1/y])
                                and (not(y and 1==1)->inv;[y>>1/y][x<<i/x]))
                ))
                 and (not(y>0) and inv) -> pos;)
```

In [11]:

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

In [15]:

```
m= BitVec('m',16)
n= BitVec('n',16)
r= BitVec('r',16)
x= BitVec('x',16)
y= BitVec('y',16)

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

d1=Implies(And(Not(y==0),1==1),substitute(substitute(substitute(substitute(inv,(y,y>>1))
,(x,x<<1)),(r,r+x)),(y,y-1)))
d2=Implies(Not(And(Not(y==0),1==1)),substitute(substitute(inv,(y,y>>1)),(x,x<<1)))
f1=inv
f2=ForAll([r,x,y],Implies(And(y>0,inv),And(d1,d2)))
f3=Implies(And(Not(y>0),inv),pos)
prove(Implies(pre,And(f1,f2,f3)))
```

```
Proved
```

**2c) Construa a definição iterativa do “single assignment unfolding” usando um parâmetro limite $N$ e aumentando a pré-condição com a condição: $(n < N) \wedge (m < N)$**

**Nesta alínea pede novamente para provar a correção do programa, mas agora usando o método de Unfold de ciclos, que consiste em desenrolar os ciclos um certo número de vezes.**

**Pegando então no nosso programa**

```
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
```

**Vamos desenrolar o ciclo em if's (desenrolamos no máximo 16 vezes pois é o tamanho máximo do BitVec):**

```
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
                        assert not (y > 0)
assert r == m * n
```

**Como é pedido no enunciado para ser em "single assignment unfolding":**

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
if (y0 > 0):
    if y0 & 1 == 1:
        ya1 , r1  = y0-1 , r0+x0
    else:
        r16 = r0
    x1 , y1 = x0<<1  ,  ya1>>1
    if (y2 > 0):
        if y1 & 1 == 1:
            ya2 , r2  = y1-1 , r1+x1
        else:
            r16 = r1
        x2 , y2 = x1<<1  ,  ya2>>1
        if (y2 > 0):
            if y2 & 1 == 1:
                ya3 , r3  = y2-1 , r2+x2
            else:
                r16 = r2
            x3 , y3 = x2<<1  ,  ya3>>1

                    (...)

                    if (y15 > 0):
                        if y15 & 1 == 1:
                            ya16 , r16  = y16-1 , r15+x15
```

```
                            ya16 , r16  = y15-1 , r15+x15
                        else:
                            r16 = r15
                        x16 , y16 = x15<<1 , ya16>>1
                        assert not (y16 > 0)

                    else:
                        r16 = r15
        else:
            r16 = r2
    else:
        r16 = r1
else:
    r16 = r0

assert r16 == m * n
```

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
    (assume y0 & 1 == 1;
        ya1 , r1  = y0-1 , r0+x0
    ||
    assume not y0 & 1 == 1;
        ya1=y0)
    x1 , y1 = x0<<1  ,  ya1>>1
    assume (y1 > 0);
        (assume y1 & 1 == 1;
            ya2 , r2  = y1-1 , r1+x1
        ||
        assume not (y1 & 1 == 1);
            ya2=y1)
        x2 , y2 = x1<<1  ,  ya2>>1
        assume (y2 > 0);
            (assume y2 & 1 == 1;
                ya3 , r3  = y2-1 , r2+x2
            ||
            assume not (y2 & 1 == 1);
                ya3=y2)
            x3 , y3 = x2<<1  ,  ya3>>1

                    (...)

                    assume (y15 > 0);
                        (assume y15 & 1 == 1;
                            ya16 , r16  = y15-1 , r15+x15
                        ||
                        assume not (y15 & 1 == 1);
                            ya16=y15)
                        x16 , y16 = x15<<1 , ya16>>1
                        assert not (y16 > 0);
                        ||
                        assume not (y16 > 0);
                            r16 = r15
        ||
        assume not (y2 > 0);
            r16 = r2
    ||
    assume not (y1 > 0);
        r16 = r1
||
assume not (y0 > 0);
```

```
    r16 = r0

assert r16 == m * n
```

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
(assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
(assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
||
assume not (y1 & 1 == 1);
ya2=y1)
x2 , y2 = x1<<1  ,  ya2>>1
assume (y2 > 0);
(assume y2 & 1 == 1;
ya3 , r3  = y2-1 , r2+x2
||
assume not (y2 & 1 == 1);
ya3=y3)
x3 , y3 = x2<<1  ,  ya3>>1

(...)

assume (y15 > 0);
(assume y15 & 1 == 1;
ya16 , r16  = y15-1 , r15+x15
||
assume not (y15 & 1 == 1);
ya16=y15)
x16 , y16 = x15<<1 , ya16>>1
assert not (y16 > 0) and r16 == m * n


||


(...)


||


assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
(assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
(assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
||
```

```
assume not (y1 & 1 == 1);
ya2=y1)
x2 , y2 = x1<<1  ,  ya2>>1
assume not (y2 > 0);
r16 = r2
assert r16 == m * n


||


assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
(assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
||
assume not y0 & 1 == 1;
ya1=y0)
x1 , y1 = x0<<1  ,  ya1>>1
assume not (y1 > 0);
r16 = r2
assert r16 == m * n


||


assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume not (y0 > 0);
r16 = r0
assert r16 == m * n
```

**como** $y > 0 \implies y \neq 0$
**e** $1 = 1 \implies True$
**temos que** $(y > 0 \implies y \neq 0) = True$
**por causa de antes vir uma condição que verifica se** $y > 0$
**e caso não seja essa parte do codigo não é executada.**

```
assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
x2 , y2 = x1<<1  ,  ya2>>1
assume (y2 > 0);
assume y2 & 1 == 1;
ya3 , r3  = y2-1 , r2+x2

x3 , y3 = x2<<1  ,  ya3>>1

(...)

assume (y15 > 0);
assume y15 & 1 == 1;
ya16 , r16  = y15-1 , r15+x15
x16 , y16 = x15<<1 , ya16>>1
assert not (y16 > 0) and r16 == m * n
```

```
||

(...)

||

assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume (y1 > 0);
assume y1 & 1 == 1;
ya2 , r2  = y1-1 , r1+x1
x2 , y2 = x1<<1  ,  ya2>>1
assume not (y2 > 0);
r16 = r2
assert r16 == m * n

||

assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume (y0 > 0);
assume y0 & 1 == 1;
ya1 , r1  = y0-1 , r0+x0
x1 , y1 = x0<<1  ,  ya1>>1
assume not (y1 > 0);
r16 = r2
assert r16 == m * n

||

assume m >= 0 and n >= 0 and r0 == 0 and x0 == m and y0 == n
assume not (y0 > 0);
r16 = r0
assert r16 == m * n
```

In [16]:

```
r=[BitVec('r'+str(i),16) for i in range(17)]
x=[BitVec('x'+str(i),16) for i in range(17)]
ya=[BitVec('ya'+str(i),16) for i in range(17)]
y=[BitVec('y'+str(i),16) for i in range(17)]
m= BitVec('m',16)
n= BitVec('n',16)

pre=And( m >=0,n >=0,r[0] == 0,x[0] == m,y[0] == n)
pos= r[16] == m*n

def cond(u):
    if u==0:
        return Implies(And(pre,Not(y[u]>0),r[16]==r[u]),pos)

    a=And([And(r[i+1]==r[i]+x[i],ya[i+1]==y[i]-1,x[i+1]==x[i]<<1,
```

```
            y[i+1]==ya[i+1]>>1) for i in range(u)])

    if u==16:
        con=Implies(And(pre,a),And(Not(y[u]>0),pos))
    else:
        con=Implies(And(pre,a,Not(y[u]>0),r[16]==r[u]),pos)
    return Or(con,cond(u-1))

prove(cond(16))
```

```
Proved
```