# RAPPORT PROJET ASSEMBLEUR – KELBERT PAUL

## Table des matières

# Exemple 1

## But

Le but de cet exemple est de montrer le bon fonctionnement de notre compilateur, car il ne présente aucune variable ni fonction. Le résultat attendu est donc une table des symboles très petite, un arbre composé de deux nœuds et un code ASM ne renvoyant rien.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
```

## Arbre

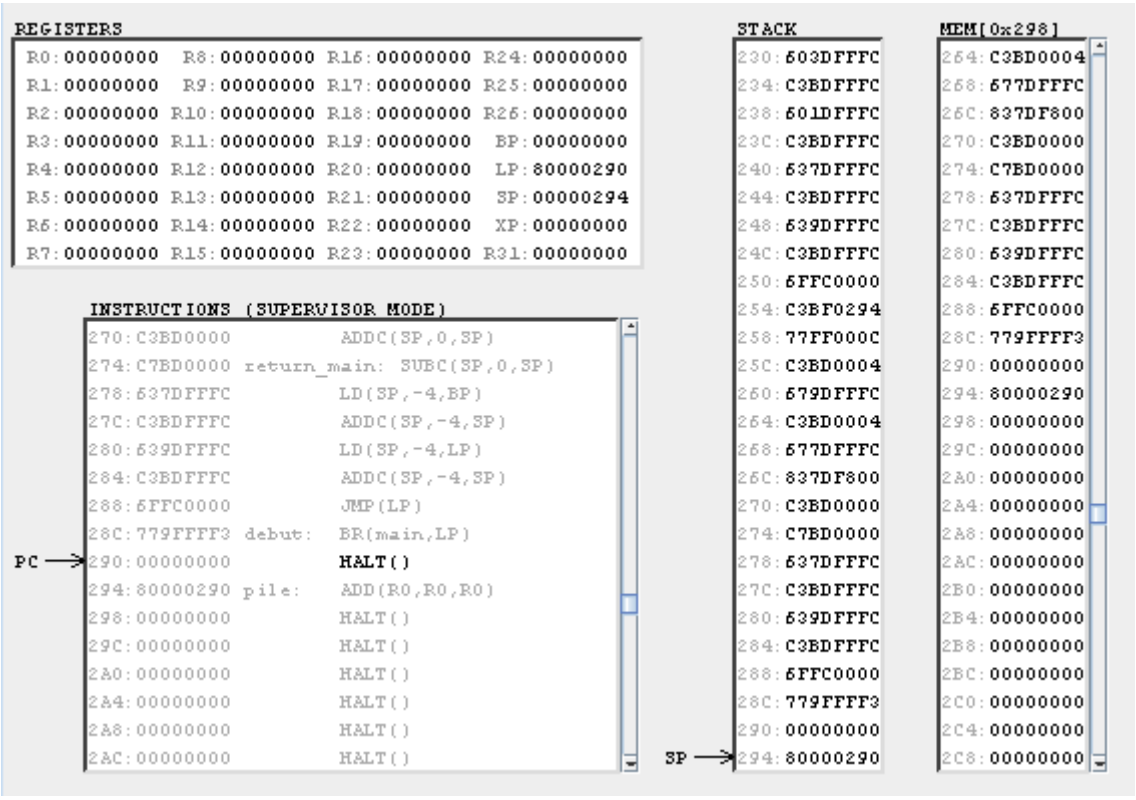
```
PROG
└─FONCTION/main
```

## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

# Analyse

## *BSim*




## *Fonctionnement*

Le retour de BSim ne présente aucuns problèmes, ce qui parait logique au vu du but de l'exemple.

# Exemple 2

## But

Le but de ce programme est le même que l'exemple précédent, mais avec quatre variables.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=i, type=int, cat=global, val=10}
{nom=j, type=int, cat=global, val=20}
{nom=k, type=int, cat=global}
{nom=l, type=int, cat=global}
```

## Arbre

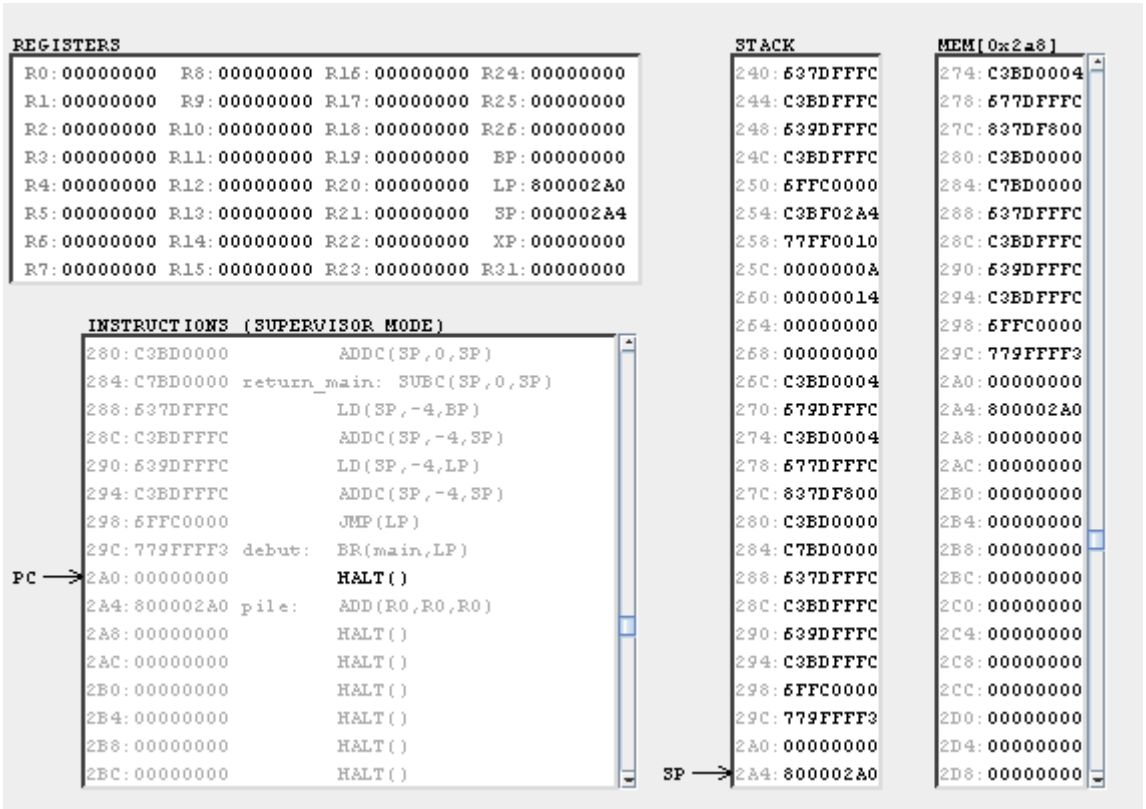
```
PROG
└FONCTION/main
```

## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
i:      LONG(10)
j:      LONG(20)
k:      LONG(0)
l:      LONG(0)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

## Analyse

### *BSim*



### *Fonctionnement*

Aucun problème de présent, le code généré est correct.

# Exemple 3

## But

Le but de cet exemple est un peu plus complexe que les deux précédents, car les variables sont à présent traitées.

k = 2 / i = 10 / j = 20

Nous voulons calculer l tel quel l = i + (3*j). Le résultat attendu est donc 70.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=i, type=int, cat=global, val=10}
{nom=j, type=int, cat=global, val=20}
{nom=k, type=int, cat=global}
{nom=l, type=int, cat=global}
```

## Arbre

```
PROG
└FONCTION/main
  └AFF
    └IDF/k
    └CONST/2
  └AFF
    └IDF/l
    └PLUS
      └IDF/i
      └MUL
        └CONST/3
        └IDF/j
```

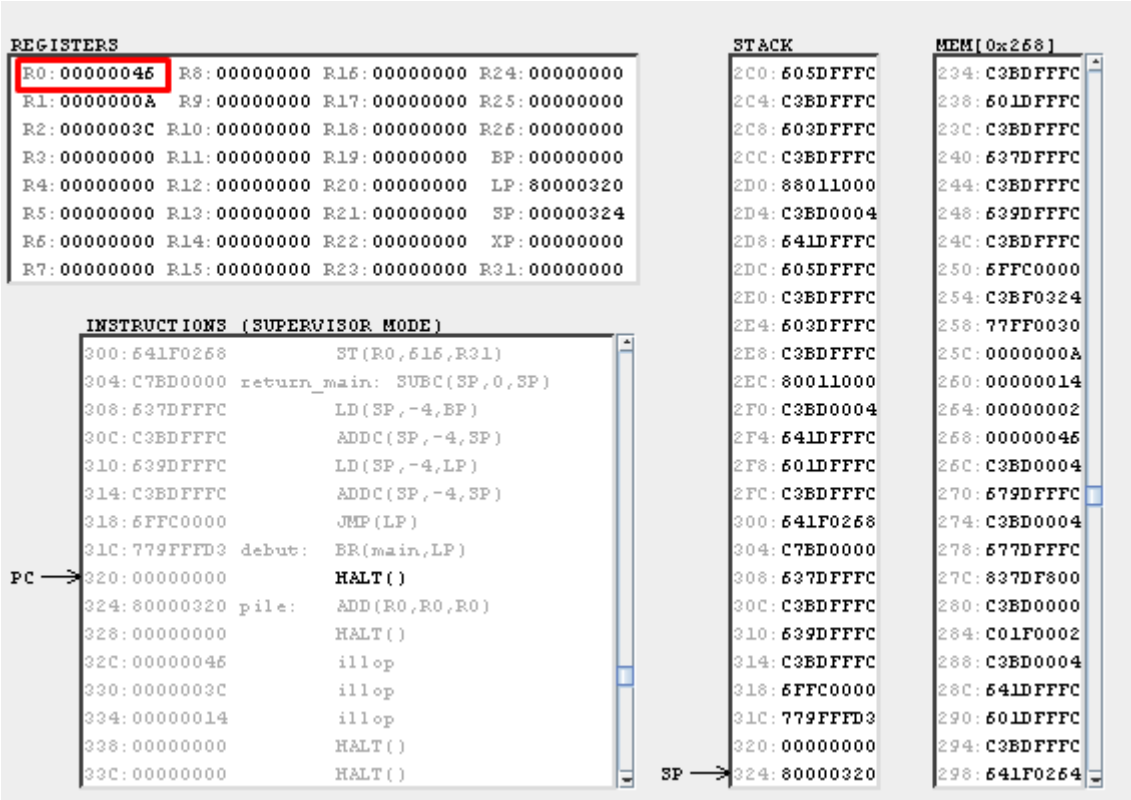
## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
i:      LONG(10)
j:      LONG(20)
k:      LONG(0)
l:      LONG(0)
```

```
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
        CMOVE(2,R0)
        PUSH(R0)
        POP(R0)
        ST(R0,k)
        LD(i,R0)
        PUSH(R0)
        CMOVE(3,R0)
        PUSH(R0)
        LD(j,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        MUL(R1,R2,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        ST(R0,l)
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

# Analyse

## *BSim*




## *Fonctionnement*

Comme le montre la valeur encadrée, le résultat est 46. Or, BSim renvoie les valeurs en hexadécimal, ce qui correspond à 70 en décimal. Donc le programme fonctionne.

# Exemple 4

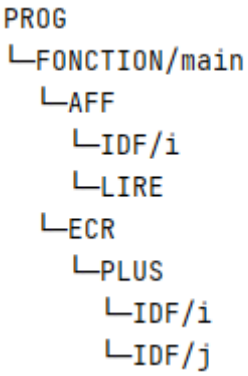
## But

Le but de cet exemple est de tester la fonction de lecture et d'écriture de notre générateur ASM. Ici, nous testerons avec i = 15 (en lecture) et j = 20. Le programme doit donc écrire 35.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=i, type=int, cat=global}
{nom=j, type=int, cat=global, val=20}
```

## Arbre

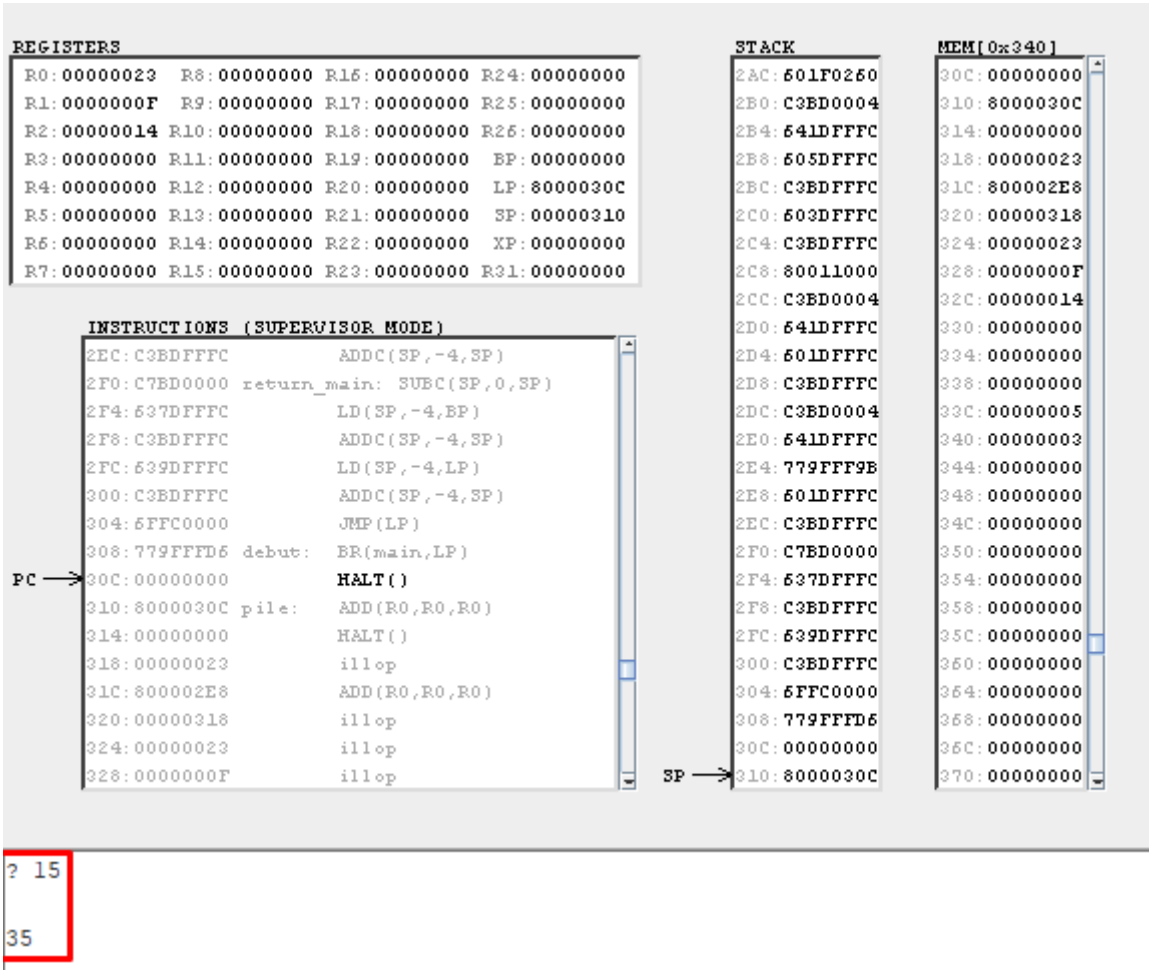



## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
i:      LONG(0)
j:      LONG(20)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
        RDINT()
        PUSH(R0)
        POP(R0)
        ST(R0,i)
        LD(i,R0)
        PUSH(R0)
        LD(j,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        WRINT()
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

# Analyse

## *BSim*




```
? 15

35
```

## *Fonctionnement*

Nous pouvons voir que la valeur 35 est bien affichée. Donc cet exemple est correct !

# Exemple 5

## But

Ici nous voulons tester l'écriture de conditions. C'est pour cela que l'utilisateur va devoir entrer une valeur, et que le programme va écrire 1 si cette valeur est supérieure à 10, ou alors 2 si elle est inférieure.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=i, type=int, cat=global}
```

## Arbre

```
PROG
└FONCTION/main
  └AFF
    └IDF/i
    └LIRE
  └SI/1
    └SUP
      └IDF/i
      └CONST/10
    └BLOC
      └ECR
        └CONST/1
    └BLOC
      └ECR
        └CONST/2
```

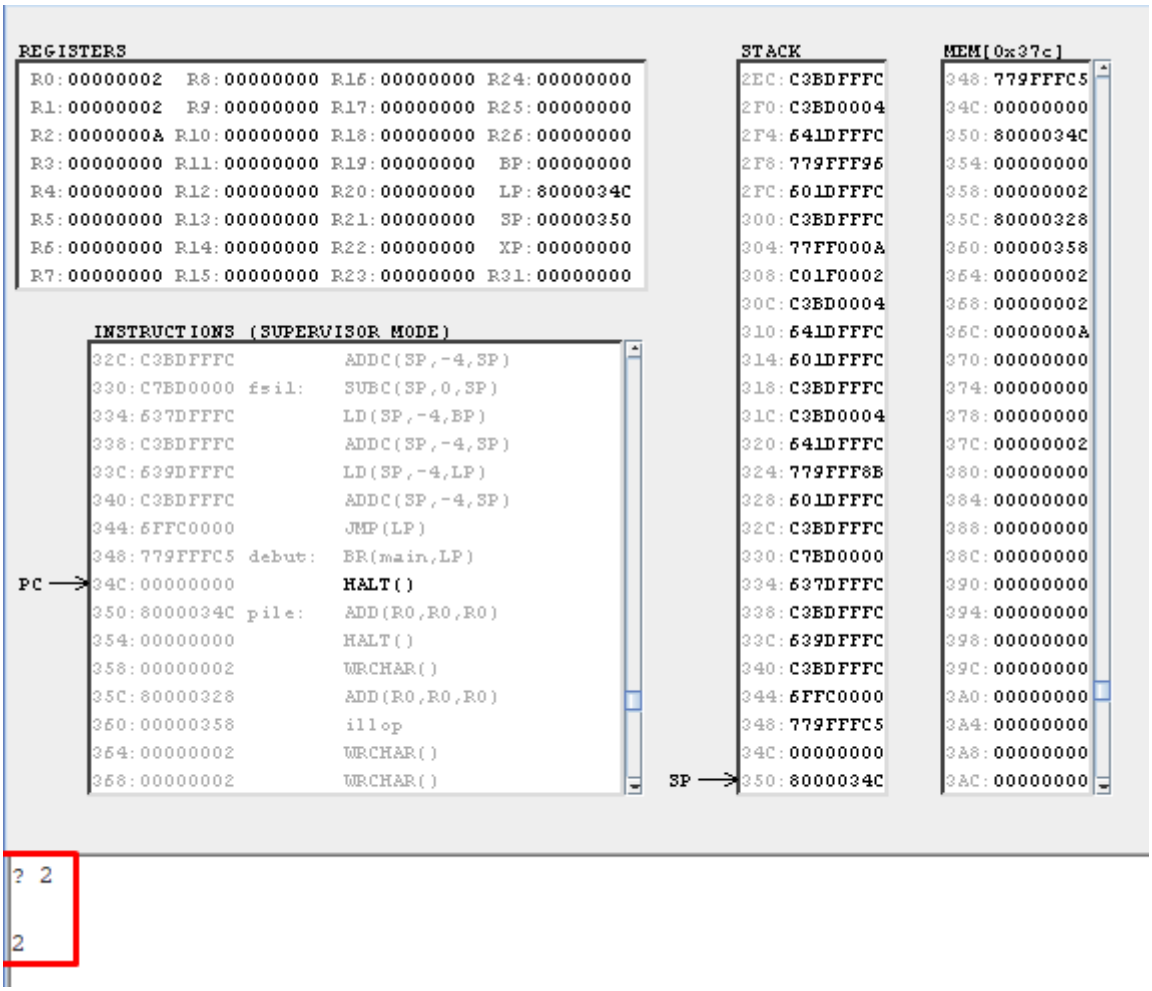
## Code assembleur

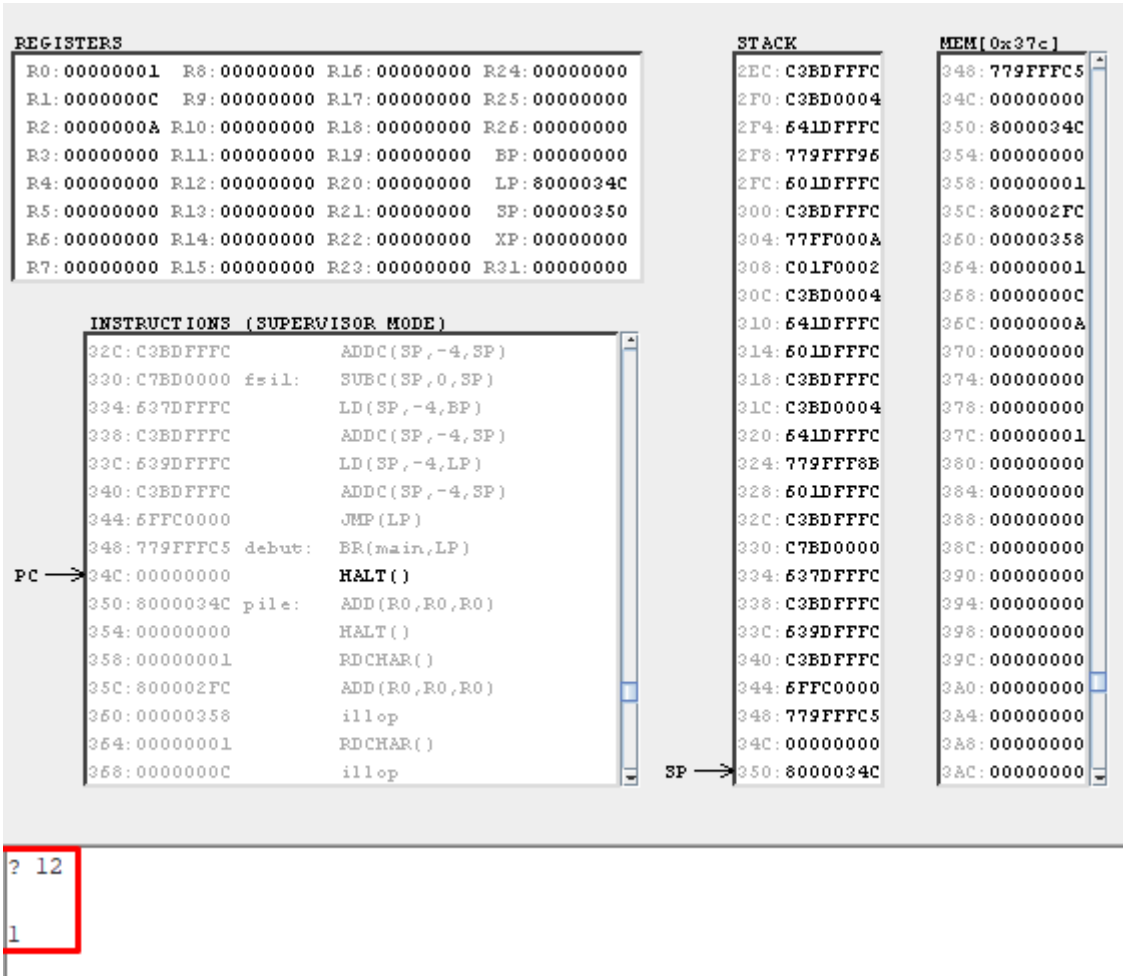
```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
i:      LONG(0)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
        RDINT()
        PUSH(R0)
        POP(R0)
        ST(R0,i)
        LD(i,R0)
        PUSH(R0)
        CMOVE(10,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        CMPLT(R2,R1,R0)
        PUSH(R0)
        POP(R0)
        BF(R0, sinon1)
        CMOVE(1,R0)
        PUSH(R0)
        POP(R0)
        WRINT()
        BR(fsi1)
sinon1:
```

```
        CMOVE(2,R0)
        PUSH(R0)
        POP(R0)
        WRINT()
fsi1:
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

## Analyse

### *BSim*






### *Fonctionnement*

Comme nous pouvons le voir sur les deux captures précédentes, le programme fonctionne.

# Exemple 6

## But

Le but de cet exemple est de tester notre générateur de Tant Que. Le programme doit donc afficher 0 1 2 3 4 5.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=i, type=int, cat=global}
{nom=n, type=int, cat=global, val=5}
```

## Arbre

```
PROG
└FONCTION/main
  └AFF
    └IDF/i
    └CONST/0
  └TQ/1
    └SUP
      └IDF/i
      └IDF/i
    └BLOC
      └ECR
        └IDF/i
      └AFF
        └IDF/i
        └PLUS
          └IDF/i
          └CONST/1
```

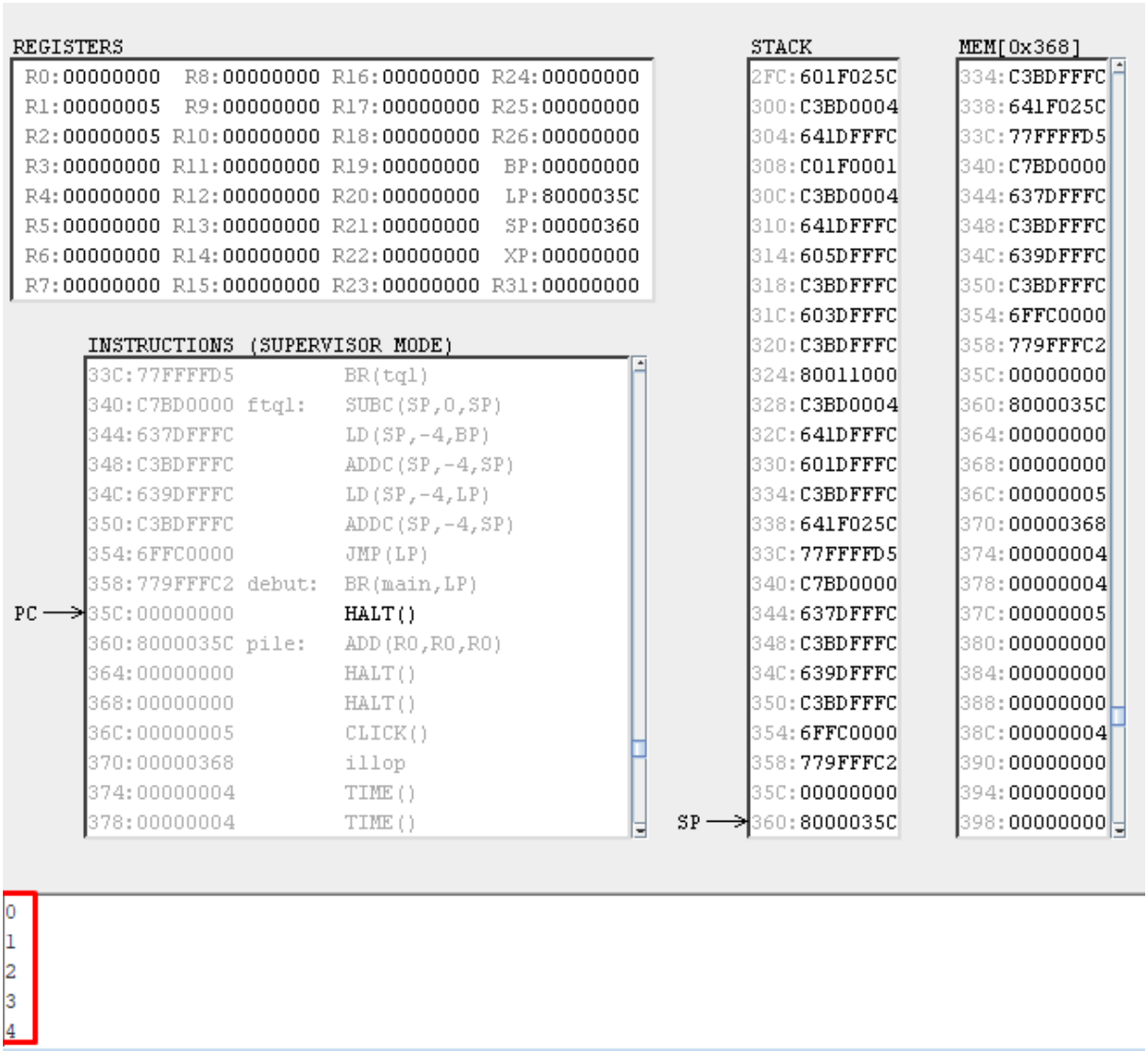
## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
i:      LONG(0)
n:      LONG(5)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
        CMOVE(0,R0)
        PUSH(R0)
        POP(R0)
        ST(R0,i)
tq1:
        LD(i,R0)
        PUSH(R0)
        LD(n,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        CMPLT(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        BF(R0,ftq1)
        LD(i,R0)
```

```
        PUSH(R0)
        POP(R0)
        WRINT()
        LD(i,R0)
        PUSH(R0)
        CMOVE(1,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        ST(R0,i)
        BR(tq1)
ftq1:

return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

# Analyse

## *BSim*




## *Fonctionnement*

Tout est bien affiché, cela fonctionne.

# Exemple 7

## But

Le résultat souhaité est 5.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=a, type=int, cat=global, val=10}
{nom=f, type=void, cat=fonction, nbParam=1, nbBloc=2}
{nom=i, type=int, cat=param, rang=0, scope=f}
{nom=x, type=int, cat=local, rang=0, scope=f}
{nom=y, type=int, cat=local, rang=1, scope=f}
```

## Arbre

```
{nom=main, type=void, cat=fonction}
{nom=a, type=int, cat=global, val=10}
{nom=f, type=void, cat=fonction, nbParam=1, nbBloc=2}
{nom=i, type=int, cat=param, rang=0, scope=f}
{nom=x, type=int, cat=local, rang=0, scope=f}
{nom=y, type=int, cat=local, rang=1, scope=f}
```

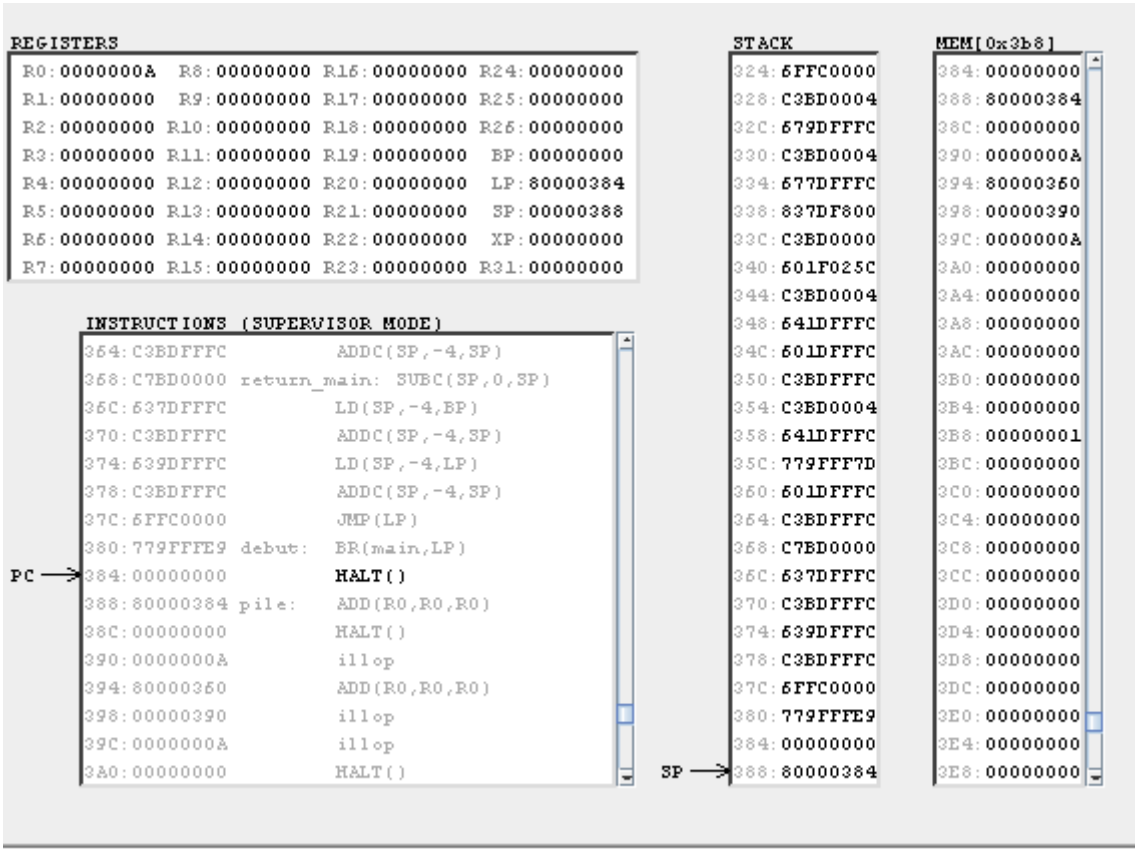
## Code assembleur

```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
a:      LONG(10)
f:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(2)
        CMOVE(0,R0)
        PUSH(R0)
        POP(R0)
        PUTFRAME(R0,-12)
        CMOVE(1,R0)
        PUSH(R0)
        POP(R0)
        PUTFRAME(R0,4)
        GETFRAME(-12,R0)
        PUSH(R0)
        GETFRAME(0,R0)
        PUSH(R0)
        GETFRAME(4,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        ST(R0,a)
return_f:
        DEALLOCATE(2)
        POP(BP)
        POP(LP)
        RTN()
main:
```

```
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)

        LD(a,R0)
        PUSH(R0)
        POP(R0)
        WRINT()
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
pile:
```

# Analyse

## *BSim*




```
10
```

## *Fonctionnement*

Comme nous pouvons le constater, le programme ne renvoie pas la bonne valeur. Cela est peut être dû au fait que la fonction créée ne renvoie rien (void), donc la valeur affichée reste la même (a = 10).

# Exemple 8

## But

Le résultat attendu de la part de cet exemple est 13.

## Table des symboles

```
{nom=main, type=void, cat=fonction}
{nom=a, type=int, cat=global}
{nom=f, type=int, cat=fonction, nbParam=2, nbBloc=1}
{nom=x, type=int, cat=local, rang=0, scope=f}
{nom=i, type=int, cat=param, rang=0, scope=f}
{nom=j, type=int, cat=param, rang=1, scope=f}
```

## Arbre

```
PROG
└FONCTION/f
  └AFF
    └IDF/x
    └PLUS
      └IDF/i
      └IDF/j
  └RET/f
    └PLUS
      └IDF/x
      └CONST/10
└FONCTION/main
  └AFF
    └IDF/a
    └APPEL/f
      └CONST/1
      └CONST/2
  └ECR
    └IDF/a
```

## Code assembleur

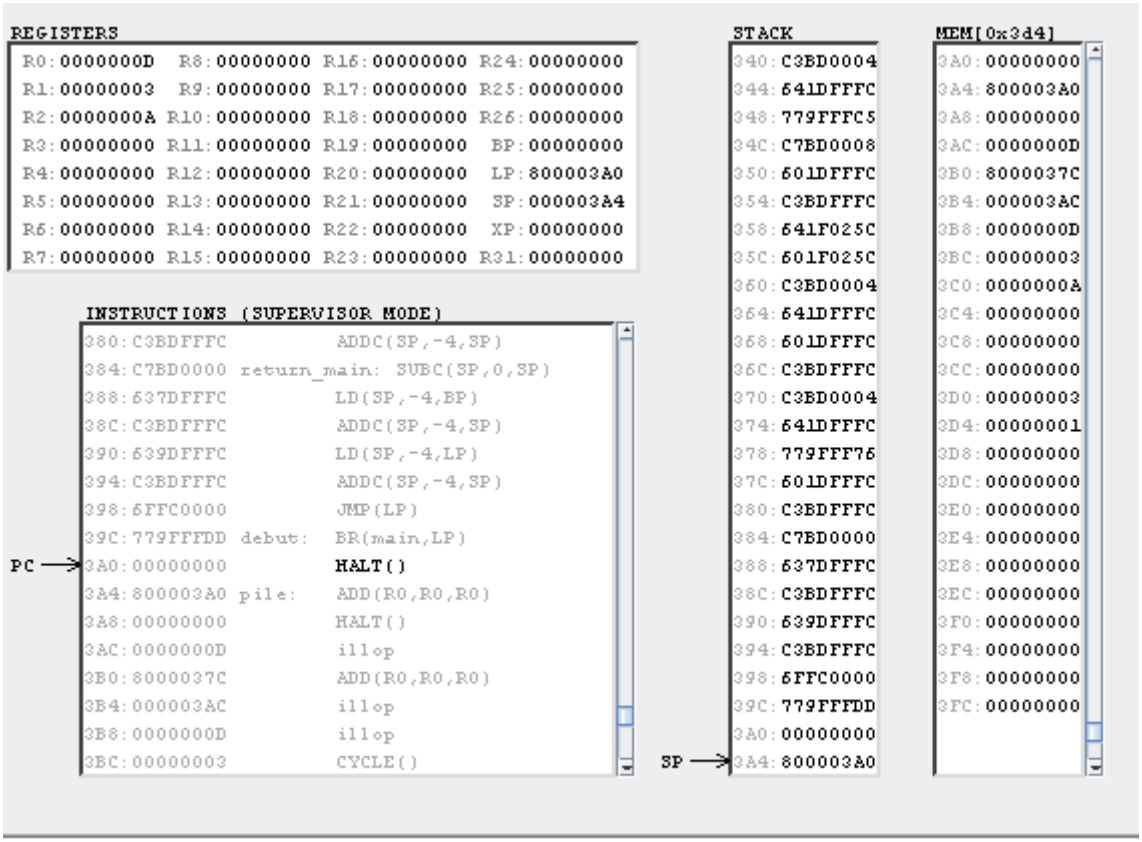
```
.include beta.uasm
.include intio.uasm
.options tty

        CMOVE(pile,SP)
        BR(debut)
a:      LONG(0)
f:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
```

```
        ALLOCATE(1)
        GETFRAME(-16,R0)
        PUSH(R0)
        GETFRAME(-12,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        PUTFRAME(R0,0)
        GETFRAME(0,R0)
        PUSH(R0)
        CMOVE(10,R0)
        PUSH(R0)
        POP(R2)
        POP(R1)
        ADD(R1,R2,R0)
        PUSH(R0)
        POP(R0)
        PUTFRAME(R0,-20)
        BR(return_f)

return_f:
        DEALLOCATE(1)
        POP(BP)
        POP(LP)
        RTN()
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
        ALLOCATE(0)
        ALLOCATE(1)
        CMOVE(1,R0)
        PUSH(R0)
        CMOVE(2,R0)
        PUSH(R0)
        CALL(f)
        DEALLOCATE(2)
        POP(R0)
        ST(R0,a)
        LD(a,R0)
        PUSH(R0)
        POP(R0)
        WRINT()
return_main:
        DEALLOCATE(0)
        POP(BP)
        POP(LP)
        RTN()
debut:
        CALL(main)
        HALT()
```

pile:

# Analyse

## *BSim*




13

## *Fonctionnement*

Le programme fonctionne parfaitement.