QUELQUES POINTS CONCERNANT LA GÉNÉRATION DE CODE

1. LANGAGE GÉNÉRÉ ET MACHINE CIBLE

Le code généré devra être en langage d'assemblage microPIUP/ASM[1] écrit dans un fichier texte au format Linux d'extension .src , en utilisant un sous-ensemble des instructions de la machine.

Le fichier généré devra être assemblé à l'aide de l'assembleur qui générera un fichier de code machine d'extension *.iup.

Ce dernier sera exécuté à l'aide du simulateur[2] du processeur APR[3] .

Ces deux outils (assembleur et simulateur) fonctionnent sur toute machine Windows ou Linux disposant d'un runtime java. Ils sont inclus dans le fichier (archive java) microPIUP.jar disponible sur le serveur neptune dans le dossier /home/depot/PFSI.

Ce fichier supporte les commandes suivantes, en supposant que le fichier microPIUP.jar soit dans le dossier courant.

Assembler un fichier toto.src dans le fichier de code machine toto.iup :

```
java -jar microPIUP.jar -ass toto.src
```

Exécuter en batch le fichier de code machine toto.iup :

```
java -jar microPIUP.jar -batch toto.iup
```

Lancer le simulateur sur interface graphique :

```
java -jar microPIUP.jar -sim
```

Suzanne Collin avait indiqué que vous pouvez utiliser MUL et DIV pour les expressions avec * et /

Sous-ensemble réduit des instructions et des modes d'adressage à utiliser :

INSTRUCTIONS	MODES	EXEMPLE	COMMENTAIRES
ADC, ADD, SUB, XOR, AND, OR, ***MUL, DIV***	registre	`SUB R1, R2, R3`	Opérations de traitement à 3 opérandes (2 sources et 1 destination) : R1 - R2 → R3
CMP	registre	`CMP R1, R2`	2 opérandes sources: CoMPare R1-R2 à 0
NEG, NOT, SWB, RRC, RLC, SHL, SRL, SRA	registre	`NEG R1, R2`	Opérations de traitement à 2 opérandes (1 source et 1 destination): -R1 → R2
ADQ, LDQ	rapide	`ADQ 4, R1` `LDQ -2, R3`	constante rapide ∈ [-128 ... 127] ADd Quick : 4 + R1 → R1 LoaD Quick: -2 → R1

LDW, LDB, STW, STB	registre, indirect, immédiat	`LDW R1, R2`
		`LDW R1, (R2)`
		`LDB R1, (R3)`
		`LDW R1, #0xD0D0`
		`LDW R1, #-52`
		`STW R1, (R5)`
		`STB R0, (R6)`
			LoaD et STore avec types: Word et Byte :
			R1 ← contenu de R2
			R1 ← mot mémoire d'adresse dans R2
			R1 ← byte mémoire d'adresse dans R3
			R1 ← hexadécimal D0D0
			R1 ← décimal -52
			mot dans R1 → mot mémoire d'adresse dans R5
			Byte droit de R0 → Byte mémoire d'adresse R6

MPC	registre	`MPC R4`	Move PC charge contenu de PC dans l'opérande: PC → R4
JEA	indirect	`JEA (R2)`	Jump to Effective Address opérande = instruction cible: saute à l'instruction d'adresse dans R2
BMP	rapide	`BMP loop-$-2` `BMP 4`	Branchement relatif *court* inconditionnel opérande = déplacement relatif au PC : branche à l'instruction d'adresse loop branche à l'instruction d'adresse $ + 2 + 4
Bcc avec cc = EQ, NE, GE, LE, GT, LW, AE, AB, BL, BE, VS, VC	rapide	`BEQ loop-$-2` `BLE -4`	Branchement relatif *court* conditionnel opérande = déplacement
TRP	registre	`TRP R1`	TRaP trappe logicielle; opérande=n°except.: exécute trappe de n° contenu dans R1
NOP	aucun	`NOP`	No Operation

[1] Le jeu d'instructions et son codage binaire ont été définis par Alexandre Parodi et la syntaxe du langage d'assemblage par Karol Proch.

[2] L'outil d'assemblage et de simulation **microPIUP** a été développé par Karol Proch.

[3] **APR** = Advanced Pedagogic RISC développé par Alexandre Parodi ; il comporte toutes les instructions et modes d'adressage pour permettre l'enseignement général de l'assembleur, mais un sous-ensemble forme une machine RISC facilitant l'implémentation matérielle sur une puce et (on l'espère) l'écriture d'un compilateur.

2. EXEMPLES DE MORCEAUX DE CODE

Cette partie a pour but de montrer comment effectuer les instructions classiques CISC avec les instructions disponibles. On suppose qu'on a écrit au début :

```
EXIT_EXC   EQU  64              // n° d'exception de EXIT
READ_EXC   EQU  65              // n° d'exception de READ  (lit 1 ligne)
WRITE_EXC  EQU  66              // n° d'exception de WRITE (affiche 1 ligne)
STACK_ADRS EQU  0x1000          // base de pile en 1000h   (par exemple)

// ces alias permettront de changer les réels registres utilisés

SP         EQU R15              // alias pour R15, pointeur de pile
WR         EQU R14              // Work Register (registre de travail)
BP         EQU R13              // frame Base Pointer (pointage environnement)
                                // R12, R11 réservés
                                // R0 pour résultat de fonction
                                // R1 ... R10 disponibles

LDW SP, #STACK_ADRS  // charge SP avec STACK_ADRS
```

Écrire une chaîne de caractères sur l'écran :

R0 contient l'adresse de la chaîne de caractères se terminant par NUL ;

```
LDQ WRITE_EXC, WR // charge n° de trappe WRITE dans registre WR
TRP WR            // lance la trappe WRITE
```

Lire une chaîne de caractères depuis le clavier après pression sur la touche "Entrée" :

R0 contient l'adresse de la zone mémoire où placer la chaîne de caractères qui sera terminée par NUL;

```
LDQ READ_EXC, WR // charge n° de trappe READ dans registre WR
TRP WR           // lance la trappe READ
```

Arrêter l'exécution :

```
LDQ EXIT_EXC, WR // charge n° de trappe EXIT dans registre WR
TRP WR           // lance la trappe EXIT
```

Empiler le contenu d'un registre R comme PSH R ou STW R, -(SP) :

```
ADQ -2, SP  // décrémente le pointeur de pile SP
STW R, (SP) // sauvegarde le contenu du registre R sur la pile
```

Dépiler le contenu d'un registre R comme POP R ou LDW R, (SP)+ :

```
LDW R,(SP)  // charge le registre R avec le sommet de pile
ADQ 2, SP   // incrémente le pointeur de pile SP
```

Appeler une fonction ou procédure d'adresse dans le registre R comme JSR (R) :

```
MPC WR        // charge le contenu du PC dans WR
ADQ 8, WR     // ajoute 8 à WR: WR contient l'adresse de retour
ADQ -2, SP    // décrémente le pointeur de pile SP
STW WR, (SP)  // sauvegarde l'adresse de retour sur le sommet de pile
JEA (R)       // saute à l'instruction d'adresse absolue dans R
```

Retourner d'une procédure ou fonction comme RTS :

```
LDW WR, (SP)  // charge WR avec l'adresse de retour
ADQ 2, SP     // incrémente le pointeur de pile SP
JEA (WR)      // saute à l'instruction d'adresse absolue dans WR
```

Effectuer un saut inconditionnel relatif long en mode immédiat comme JMP #toto-$-2 :

```
MPC R1               // charge PC dans R1
LDW WR, #toto-$-2+2 // charge WR avec toto-$
ADD R1, WR, WR       // WR = adresse cible
JEA (WR)             // saute à l'instruction d'adresse toto
```

BMP toto-$-2 ne marche pas si déplacement ∉ [-128 .. 127]

Effectuer LINK R :

```
ADQ -2, SP    // décrémente le pointeur de pile SP
STW BP, (SP)  // sauvegarde le contenu du registre BP sur la pile
LDW BP, SP    // charge contenu SP ds BP qui pointe sur sa sauvegarde
SUB SP, R, SP // réserve R octets sur la pile pour variables locales
```

Effectuer UNLINK :

```
LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
LDW BP, (SP)  // charge BP avec ancien BP
ADQ 2, SP     // ancien BP supprimé de la pile
```

3. Exemple d'environnement de pile lors de l'exécution d'une fonction ou procédure

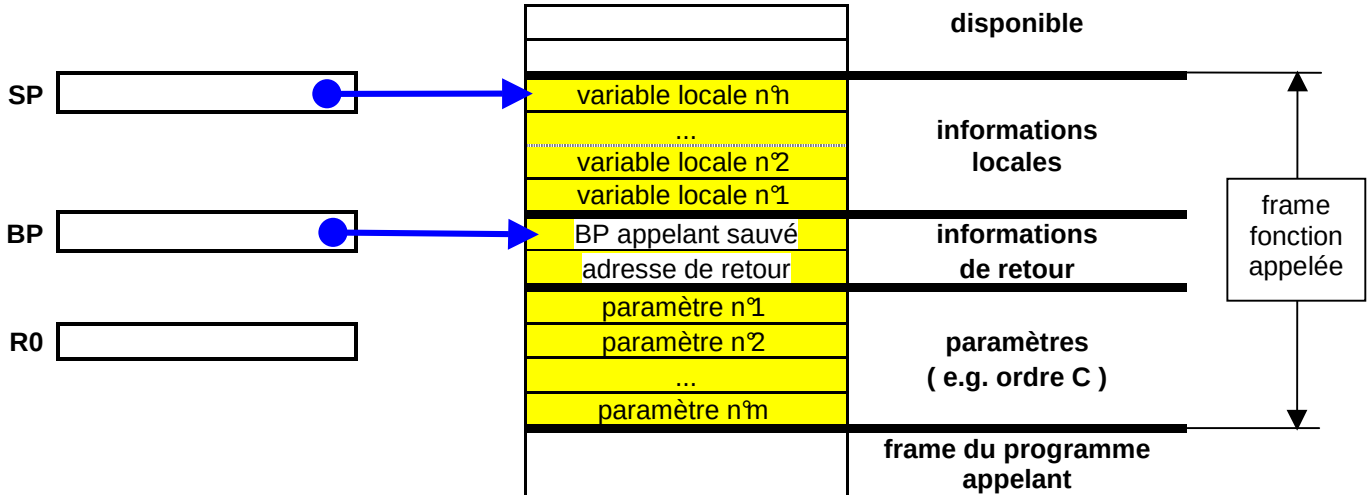
BP est le Base Pointer (a priori registre R13) facilitant l'accès aux paramètres et variables ;

SP est le "Stack Pointer" (nécessairement le registre R15) ;

R0 contient la valeur retournée par la fonction ;

Si aucun autre registre que PC, BP et SP ne contient d'information à conserver lors de l'appel à la fonction ou procédure, la zone pour sauver les registres peut être omise.

Ici on suppose ici qu'il n'y a pas un nombre de paramètres variable: "nb de paramètres" qui le facilite est omis.



En C, les paramètres sont empilés de droite à gauche; sinon pour les autres langages, c'est usuellement de gauche à droite.
Nous prenons la convention C dans notre exemple ci-après, bien qu'elle n'apporte pas d'avantage ici.
Il est par ailleurs plus facile de passer les paramètres dans l'ordre usuel, de gauche à droite, ce qui est possible aussi.

Dans les exemples, les symboles représentant des adresses de fonction sont appelés avec le nom en C concaténé avec le caractère souligné, mais ça n'est pas obligatoire.

4. EXEMPLE S DE CODAGE ET D'EXÉCUTION DE FONCTION

4.1. Exemple de code source C :

```
int substract(int x, int y)
{
int z;
z = x - y;
return z ;
}
```

définition de la fonction substract

```
void main(void)
{
int a;
int b;
a = 2;
b = 5;
a = substract(a, b);
}
```

définition du programme principal main.

La fonction doit normalement être déclarée avant son utilisation, afin que le compilateur puisse facilement vérifier la syntaxe de son appel.
En C, ceci est usuellement fait avec un prototype, par exemple ici :

```
int substract(int x, int y);
```

La définition (son comportement) peut ensuite être placé après son utilisation.

Toutefois, les prototypes ne semblent pas exister dans le sous-ensemble microC du projet de compilation.

En C si on n'utilise pas de prototype pour déclarer une fonction, la définition de la fonction doit être faite avant son utilisation.

C'est ce qui est fait dans l'exemple ci-dessus.

Dans les exemples de traduction en langage d'assemblage qui suivent, nous mettrons la traduction de la fonction et du programme principal dans le même ordre que dans le source par soucis de cohérence.

Toutefois, ça n'est pas une obligation: le fichier en langage d'assemblage fonctionne dans les deux cas.

4.2 Exemple du code généré avec environnement pile classique et jeu d'instructions CISC

La traduction avec l'ensemble des instructions CISC est plus compacte, ce qui permet d'améliorer la compréhension dans un premier temps. La traduction avec du code RISC strict sera présentée ensuite au § 4.2 .

```
EXIT_EXC    EQU   64              // n° d'exception de EXIT
READ_EXC    EQU   65              // n° d'exception de READ  (lit 1 ligne)
WRITE_EXC   EQU   66              // n° d'exception de WRITE (affiche texte)
STACK_ADRS  EQU   0x1000          // base de pile en 1000h   (par exemple)
LOAD_ADRS   EQU   0xF000          // adresse de chargement de l'exécutable
NIL         EQU   0               // fin de liste: contenu initial de BP

// ces alias permettront de changer les réels registres utilisés
SP          EQU   R15             // alias pour R15: Stack Pointer (pointeur de pile)
WR          EQU   R14             // alias pour R14: Work Register (reg. de travail)
BP          EQU   R13             // alias pour R13: frame Base Pointer (point. envir.)
                                  // R12, R11 réservés
                                  // R0 pour résultat de fonction
                                  // R1 ... R10 disponibles
```

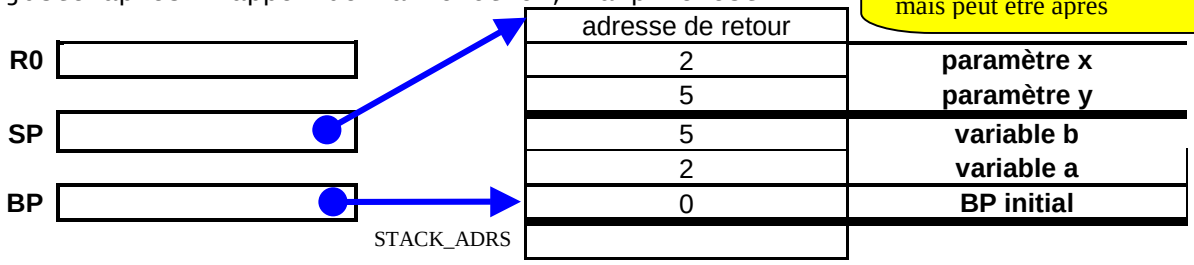
```
            ORG   LOAD_ADRS       // adresse de chargement
            START main_           // adresse de démarrage
```

à placer pour un programme ASM complet

```
// FONCTION APPELéE
```

ici avant le programme principal main comme dans le source, mais peut être après

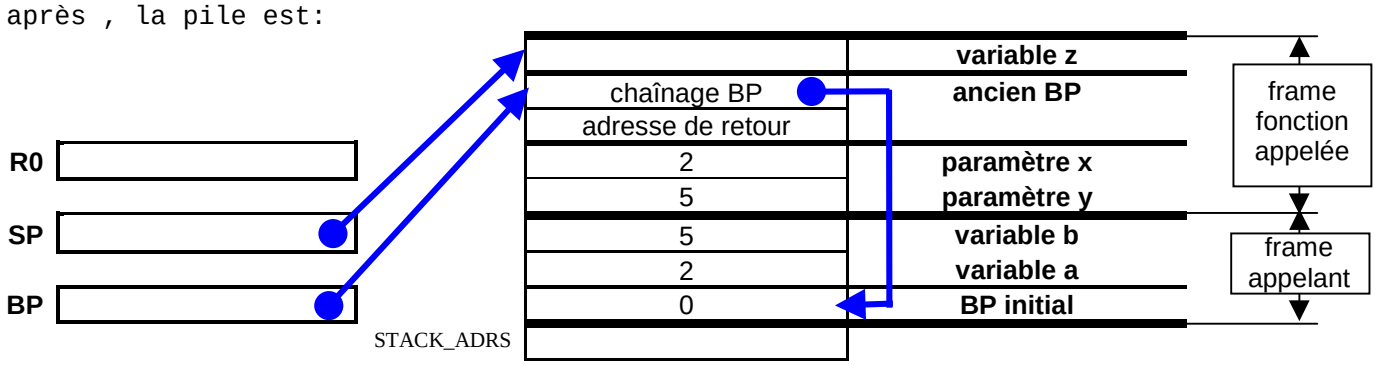
```
// juste après l'appel de la fonction, la pile est:
```




```
// int substract(int x, int y)
// {
// int z;

// prépare l'environnement de la fonction appelée (prologue) :

substract_ LDQ 2, R1      // R1 = taille données locales (ici z) de fonction appelée

    // LINK (R1)      // crée et lie l'environnement de fonction appelée
          ADQ -2, SP     // décrémente le pointeur de pile SP
          STW BP, (SP)   // sauvegarde le contenu du registre BP sur la pile
          LDW BP, SP     // charge contenu SP ds BP qui pointe sur sa sauvegarde
          SUB SP, R1, SP // réserve R1 octets sur la pile pour la variable locale z
```

n'existe pas sur APR

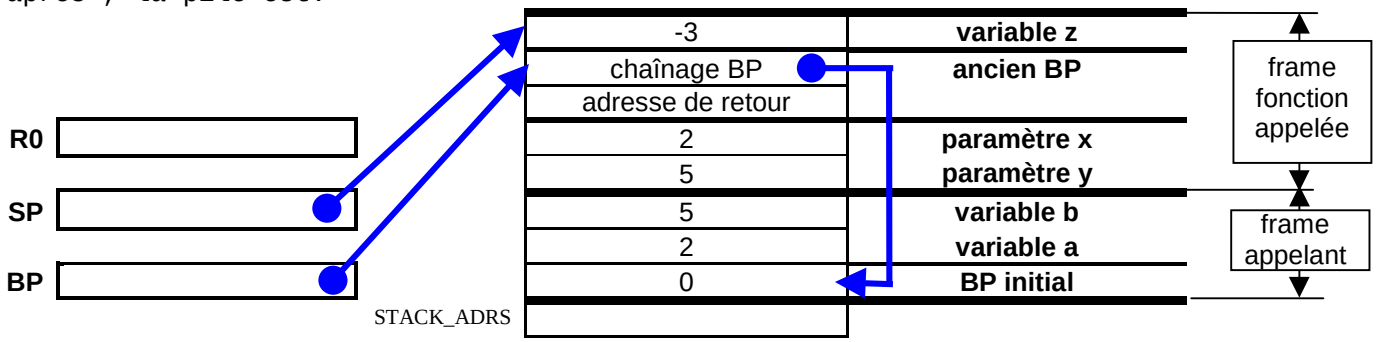
```
// après , la pile est:
```



```
// z = x - y;

        LDW R1, (BP)4  // charge le paramètre x de déplacement 4 dans R1

        LDW R2, (BP)6  // charge le paramètre y de déplacement 6 dans R2

    // calcule le résultat de la fonction

        SUB R1, R2, R1 // R1 = x - y = -3

        STW R1, (BP)-2 // affecte variable z de déplacement
```

on peut utiliser la pile pour calculer les expressions quelconques, mais il y assez de registres disponibles pour la plupart (R1 à R10)

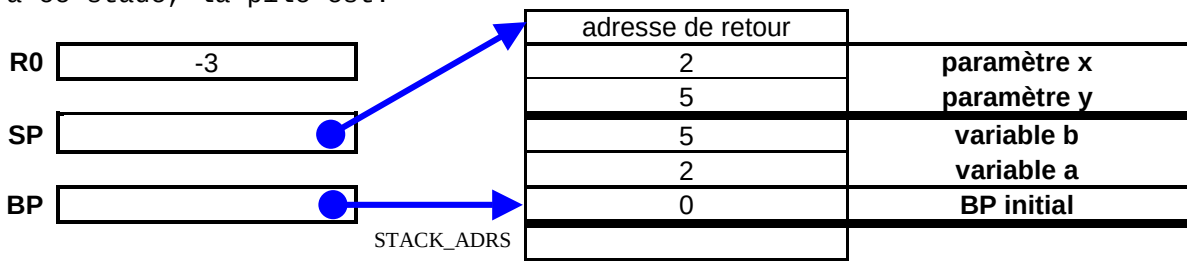
```
// après , la pile est:
```



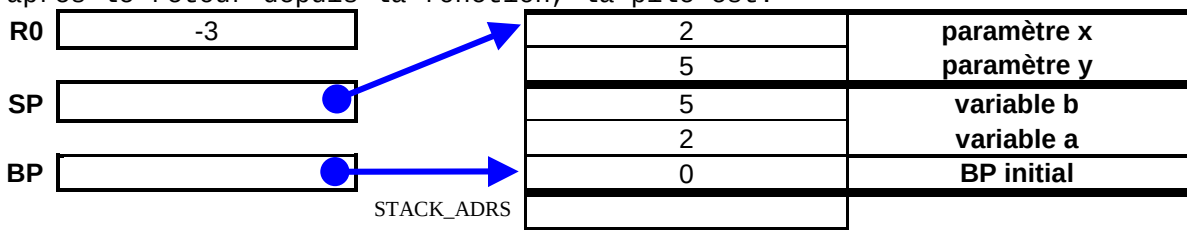

```
// return z ;

        LDW R0, (BP)-2   // charge variable z de déplacement -2 dans R0 :

// }  accolade fermante de la définition de la fonction

// fin de la fonction (épilogue) :

    // UNLINK
        LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
        LDW BP, (SP)  // charge BP avec ancien BP
        ADQ 2, SP     // ancien BP supprimé de la pile
```

n'existe pas sur APR

```
// à ce stade, la pile est:
```




```
        RTS  // retour au programme appelant:

// après le retour depuis la fonction, la pile est:
```



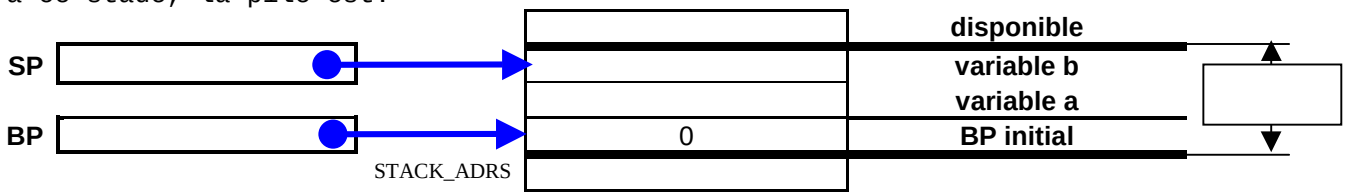
```
// PROGRAMME PRINCIPAL

// void main(void)
// {
// int a;
// int b;

main_       LDW SP, #STACK_ADRS              // charge SP avec STACK_ADRS
            LDQ NIL, BP                      // charge BP avec NIL=0
```

initialisation de SP et BP ici pour simplifier; ne font usuellement pas partie de main mais de l'OS avant.

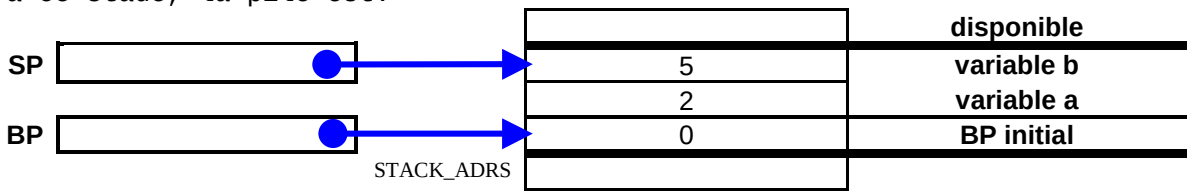

```
// prépare l'environnement (frame) du programme principal qui a 2 variables int a, b;

            LDQ 2*2, R1     // R1 = taille données locales prog. principal:
                            // 2 variables * 2 octets / variable ici

   // LINK (R1)        // crée et lie l'environnement du prog. principal
            ADQ -2, SP      // décrémente le pointeur de pile SP
            STW BP, (SP)    // sauvegarde le contenu du registre BP sur la pile
            LDW BP, SP      // charge contenu SP ds BP qui pointe sur sa sauvegarde
            SUB SP, R1, SP // réserve R1 octets sur la pile pour variables locales
```

n'existe pas sur APR

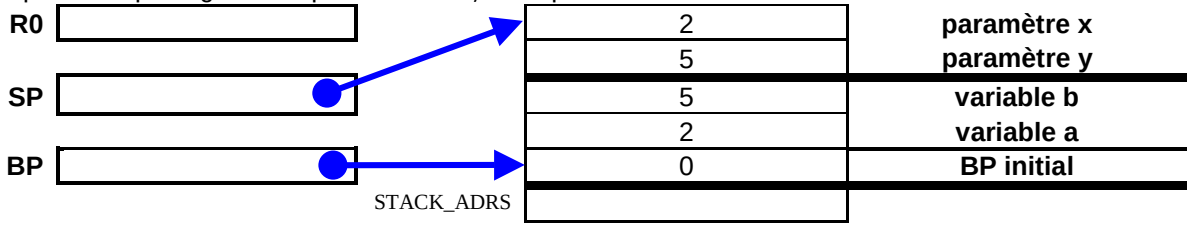
```
// à ce stade, la pile est:
```



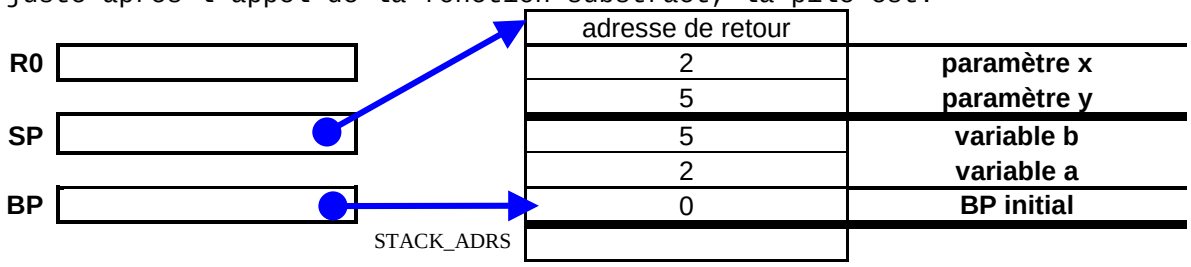

```
// a = 2
            LDW R1, #2      // R1 = 2
            STW R1, (BP)-2 // affecte variable a de déplacement -2 avec contenu de R1

// b = 5
            LDW R1, #5      // R1 = 5
            STW R1, (BP)-4 // affecte variable b de déplacement -4 avec contenu de R1

// à ce stade, la pile est:
```



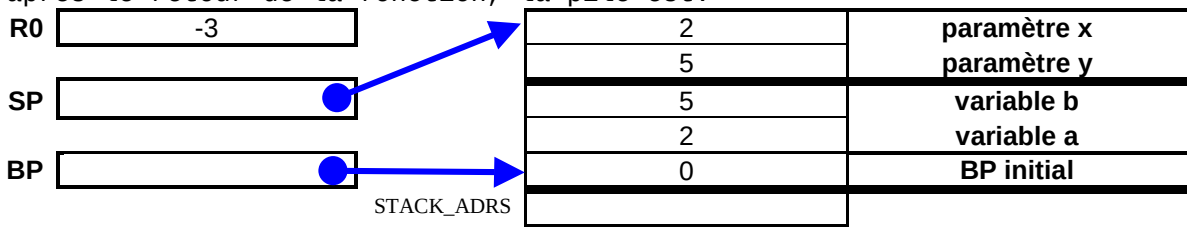

```
// a = substract(a, b);

// empile les paramètres de la fonction
            LDW R1, (BP)-4 // charge variable b de déplacement -4 dans R1 :

            STW R1, -(SP)  // empile paramètre y = b contenu dans R1 :

            LDW R1, (BP)-2 // charge variable a de déplacement -2 dans R1 :

            STW R1, -(SP)  // empile paramètre x = a contenu dans R1 :

// après empilage des paramètres, la pile est:
```



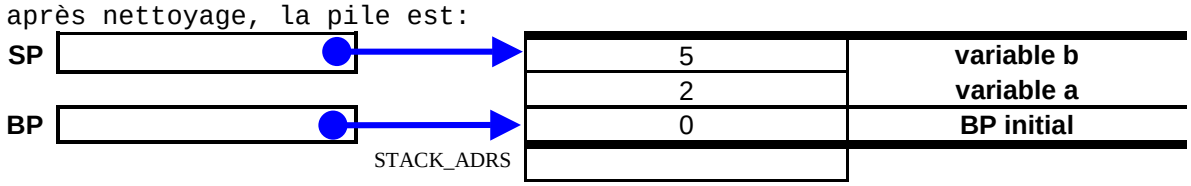
```
// appelle la fonction substract d'adresse substract_ :
            JSR @substract_ // appelle la fonction d'adresse substract_:

// juste après l'appel de la fonction substract, la pile est:
```



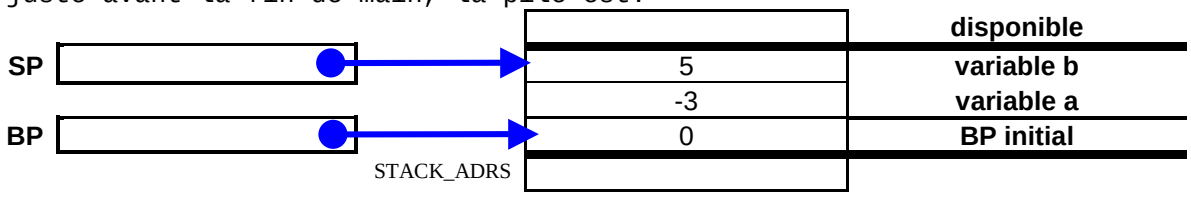

```
// après le retour de la fonction, la pile est:
```




```
// nettoyage de la pile par le programme appelant
          ADQ 2*2, SP   // SP + 2 * 2 -> SP

// après nettoyage, la pile est:
```



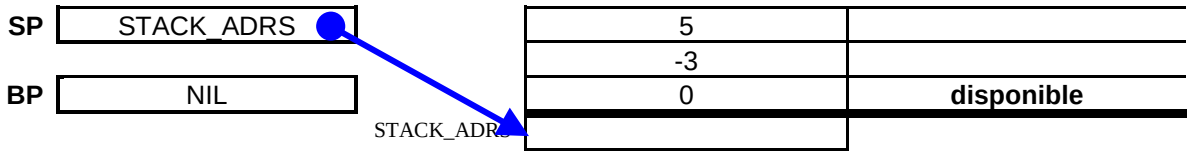

```
            STW R0, (BP)-2 // affecte variable a de déplacement -2 avec contenu de R0

// juste avant la fin de main, la pile est:
```




```
// }   accolade fermante de la définition du programme principal main
      // UNLINK
            LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
            LDW BP, (SP)  // charge BP avec ancien BP
            ADQ 2, SP     // ancien BP supprimé de la pile
```

n'existe pas sur APR

```
// après la fin du programme principal main, la pile est :
```




```
// arrête le programme
            TRP #EXIT_EXC       // EXIT: arrête le programme

// gère le redémarrage du programme
            JEA @main_          // si on redemande l'exécution, saute à main_
```

nécessaire sur le simulateur, hors mode "pas à pas" sinon le programme ne s'arrête pas !

4.3 Exemple de code généré avec environnement pile classique et jeu d'instructions RISC strict

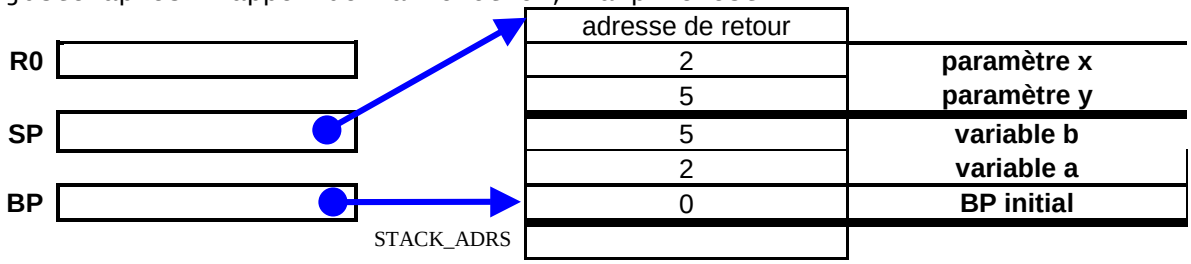
```
EXIT_EXC    EQU   64              // n° d'exception de EXIT
READ_EXC    EQU   65              // n° d'exception de READ  (lit 1 ligne)
WRITE_EXC   EQU   66              // n° d'exception de WRITE (affiche texte)
STACK_ADRS  EQU   0x1000          // base de pile en 1000h   (par exemple)
LOAD_ADRS   EQU   0xF000          // adresse de chargement de l'exécutable
NIL         EQU   0               // fin de liste: contenu initial de BP

// ces alias permettront de changer les réels registres utilisés
SP          EQU   R15             // alias pour R15: Stack Pointer (pointeur de pile)
WR          EQU   R14             // alias pour R14: Work Register (reg. de travail)
BP          EQU   R13             // alias pour R13: frame Base Pointer (point. envir.)
                                  // R12, R11 réservés
                                  // R0 pour résultat de fonction
                                  // R1 ... R10 disponibles
```

à placer pour un programme ASM complet

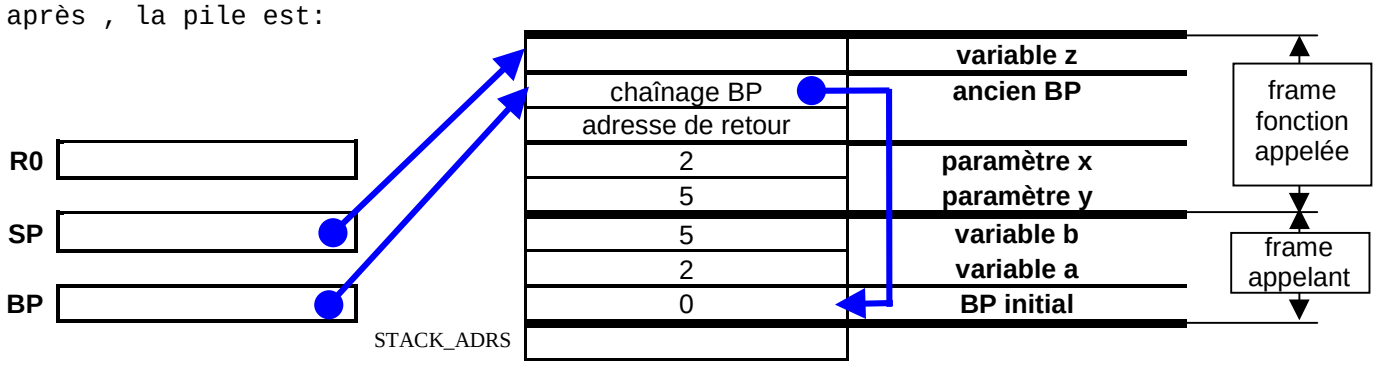
```
            ORG   LOAD_ADRS       // adresse de chargement
            START main_           // adresse de démarrage

//=============================================================
// FONCTION APPELéE
```

ici avant le programme principal comme dans le source, mais peut être après.

```
// juste après l'appel de la fonction, la pile est:
```




```
// int substract(int x, int y)
// {
// int z;

// prépare l'environnement de la fonction appelée (prologue) :

substract_ LDQ 2, R1      // R1 = taille données locales (ici z) de fonction appelée

    // LINK (R1)      // crée et lie l'environnement de fonction appelée
           ADQ -2, SP     // décrémente le pointeur de pile SP
           STW BP, (SP)   // sauvegarde le contenu du registre BP sur la pile
           LDW BP, SP     // charge contenu SP ds BP qui pointe sur sa sauvegarde
           SUB SP, R1, SP // réserve R1 octets sur la pile pour la variable locale z

// après , la pile est:
```

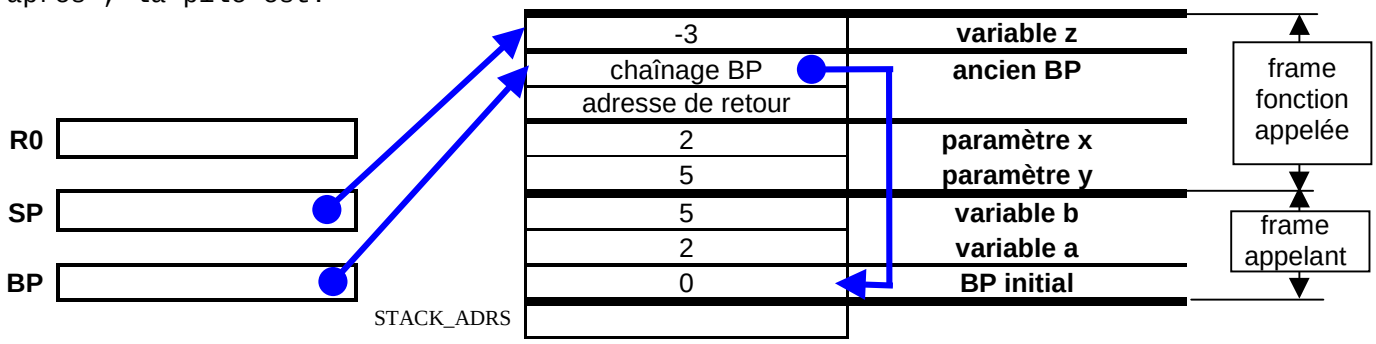



```
// z = x - y;

    // LDW R1, (BP)4  // charge le paramètre x de déplacement 4 dans R1 :
           LDW WR, BP     // WR = BP
           ADQ 4, WR      // WR pointe sur paramètre x
           LDW R1, (WR)   // R1 = x
```

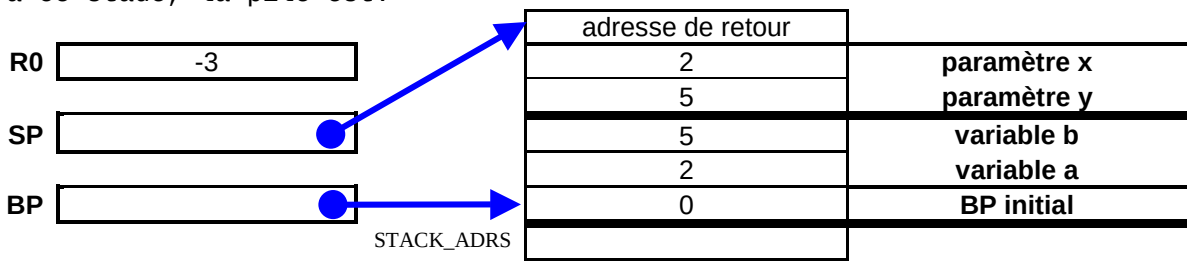
```
    // LDW R2, (BP)6  // charge le paramètre y de déplacement 6 dans R2 :
           LDW WR, BP      // WR = BP
           ADQ 6, WR       // WR pointe sur paramètre y
           LDW R2, (WR)    // R2 = y

    // calcule le résultat de la fonction
           SUB R1, R2, R1 // R1 = x - y = -3

    // STW R1, (BP)-2 // affecte variable z de déplacement -2 avec contenu de R1 :
           LDW WR, BP      // WR = BP
           ADQ -2, WR      // WR pointe sur variable a
           STW R1, (WR)    // z = -3
```

on peut utiliser la pile pour calculer les expressions quelconques, mais il y assez de registres disponibles pour la plupart (R1 à R10)

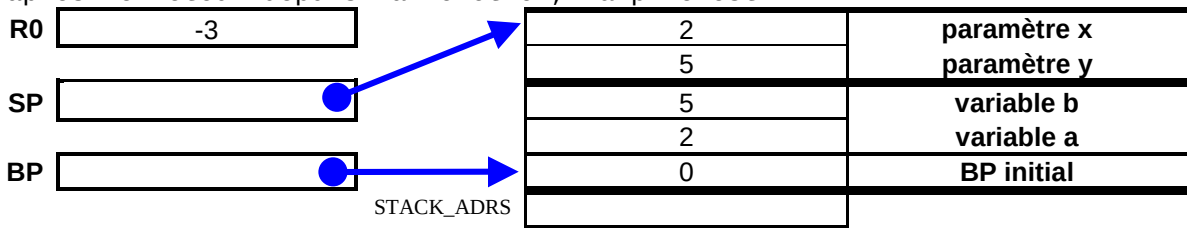
// après , la pile est:




```
// return z ;

// LDW R0, (BP)-2   // charge variable z de déplacement -2 dans R0 :
           LDW WR, BP      // WR = BP
           ADQ -2, WR      // WR pointe sur variable a
           LDW R0, (WR)    // R0 = z

// }  accolade fermante de la définition de la fonction

// fin de la fonction (épilogue) :

    // UNLINK
           LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
           LDW BP, (SP)  // charge BP avec ancien BP
           ADQ 2, SP     // ancien BP supprimé de la pile

// à ce stade, la pile est:
```



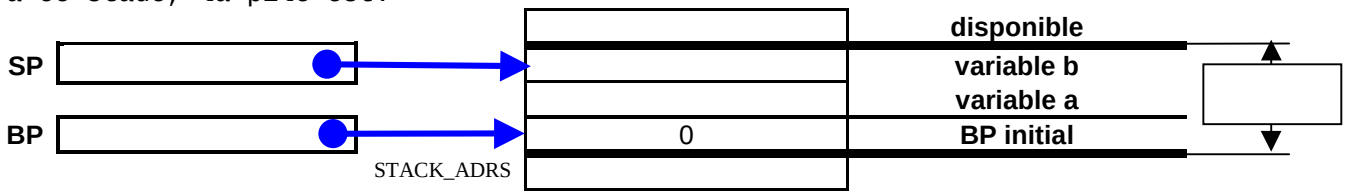

```
      // RTS  // retour au programme appelant:
           LDW WR, (SP)  // charge WR avec l'adresse de retour
           ADQ 2, SP     // incrémente le pointeur de pile SP
           JEA (WR)      // saute à l'instruction d'adresse absolue dans WR

// après le retour depuis la fonction, la pile est:
```



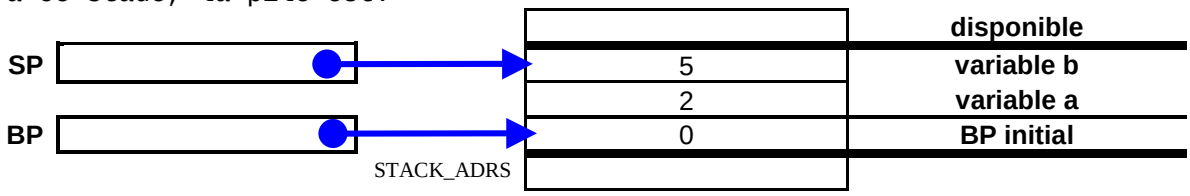
```
// PROGRAMME PRINCIPAL

// void main(void)
// {int a;
// int b;

main_       LDW SP, #STACK_ADRS          // charge SP avec STACK_ADRS
            LDQ NIL, BP                  // charge BP avec NIL
```

initialisation de SP et BP ici pour simplifier; ne font usuellement pas partie de main mais de l'OS avant.


```
// prépare l'environnement (frame) du programme principal qui a 2 variables int a, b;

            LDQ 2*2, R1     // R1 = taille données locales prog. principal:
                            // 2 variables * 2 octets / variable ici

   // LINK (R1)       // crée et lie l'environnement du prog. principal
            ADQ -2, SP      // décrémente le pointeur de pile SP
            STW BP, (SP)    // sauvegarde le contenu du registre BP sur la pile
            LDW BP, SP      // charge contenu SP ds BP qui pointe sur sa sauvegarde
            SUB SP, R1, SP // réserve R1 octets sur la pile pour variables locales

// à ce stade, la pile est:
```



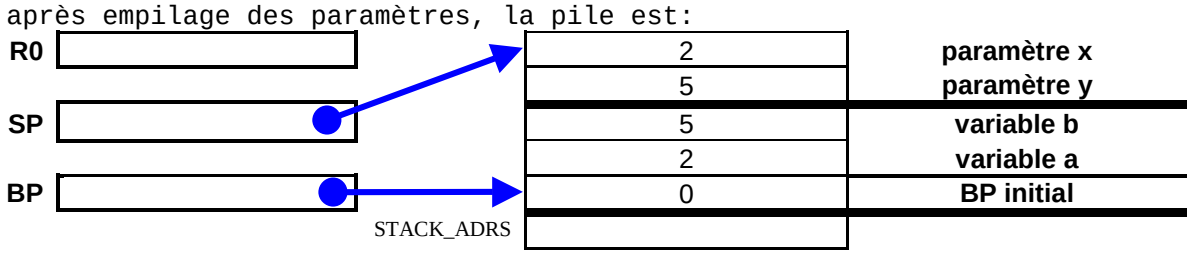

```
// a = 2
            LDW R1, #2      // R1 = 2
      // STW R1, (BP)-2 // affecte variable a de déplacement -2 avec contenu de R1 :
            LDW WR, BP      // WR = BP
            ADQ -2, WR      // WR pointe sur variable a
            STW R1, (WR)    // a = 2

// b = 5
            LDW R1, #5      // R1 = 5
      // STW R1, (BP)-4 // affecte variable b de déplacement -4 avec contenu de R1 :
            LDW WR, BP      // WR = BP
            ADQ -4, WR      // WR pointe sur variable b
            STW R1, (WR)    // b = 5

// à ce stade, la pile est:
```




```
// a = substract(a, b);

// empile les paramètres de la fonction
      // LDW R1, (BP)-4 // charge variable b de déplacement -4 dans R1 :
            LDW WR, BP      // WR = BP
            ADQ -4, WR      // WR pointe sur variable a
            LDW R1, (WR)    // R1 = b

      // STW R1, -(SP) // empile paramètre y = b contenu dans R1 :
            ADQ -2, SP      // décrémente le pointeur de pile SP
            STW R1, (SP)    // sauvegarde le contenu du registre R1 sur la pile
```

```
      // LDW R1, (BP)-2   // charge variable a de déplacement -2 dans R1 :
           LDW WR, BP     // WR = BP
           ADQ -2, WR     // WR pointe sur variable a
           LDW R1, (WR)   // R1 = a

      // STW R1, -(SP)    // empile paramètre x = a contenu dans R1 :
           ADQ -2, SP     // décrémente le pointeur de pile SP
           STW R1, (SP)   // sauvegarde le contenu du registre R1 sur la pile
```

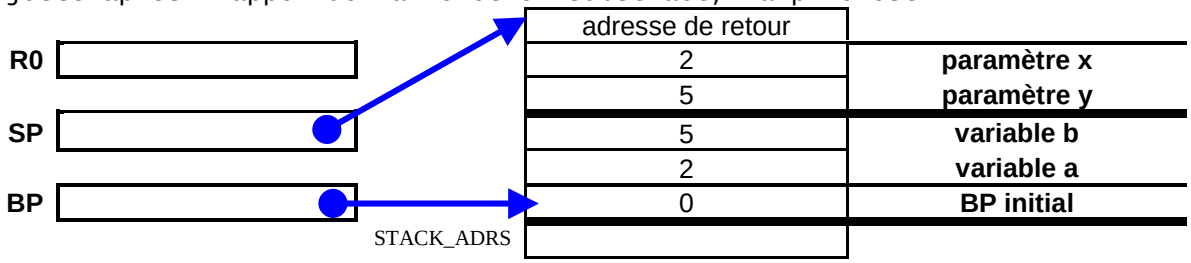
// après empilage des paramètres, la pile est:



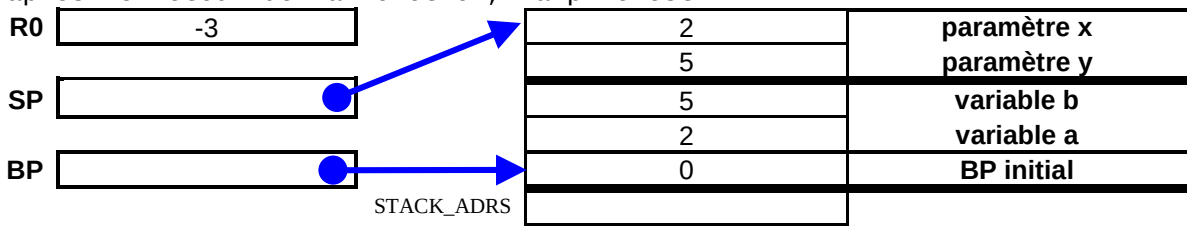
// appelle la fonction substract d'adresse substract_ :

```
    // JSR @substract_ // appelle la fonction d'adresse substract_:
           LDW R1, #substract_ // charge l'adresse substract_ de la fonction dans R1
           MPC WR         // charge le contenu du PC dans WR
           ADQ 8, WR      // ajoute 8 à WR: WR contient l'adresse de retour
           ADQ -2, SP     // décrémente le pointeur de pile SP
           STW WR, (SP)   // sauvegarde l'adresse de retour sur le sommet de pile
           JEA (R1)       // saute à l'instruction d'adresse absolue dans R1
```

// juste après l'appel de la fonction substract, la pile est:

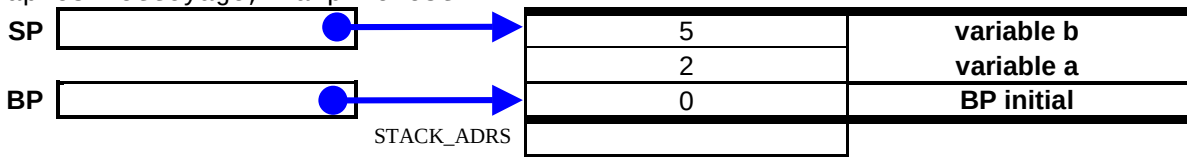


// après le retour de la fonction, la pile est:

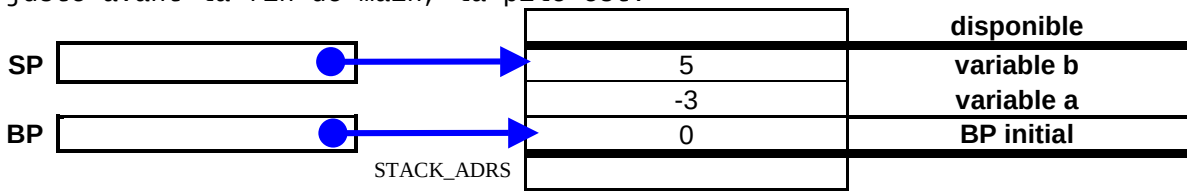



```
// nettoyage de la pile par le programme appelant
         ADQ 2*2, SP   // SP + 2 * 2 -> SP
```

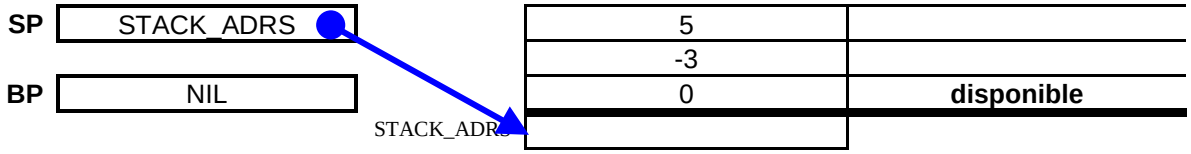
// après nettoyage, la pile est:



```
// STW R0, (BP)-2 // affecte variable a de déplacement -2 avec contenu de R0 :
          LDW WR, BP     // WR = BP
          ADQ -2, WR     // WR pointe sur variable a <-- bug corrigé: dep =-2 pas -4
          STW R0, (WR)   // a = -3

// juste avant la fin de main, la pile est:
```




```
// }   accolade fermante de la définition du programme principal main
      // UNLINK
           LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
           LDW BP, (SP)  // charge BP avec ancien BP
           ADQ 2, SP     // ancien BP supprimé de la pile

// après la fin du programme principal main, la pile est :
```

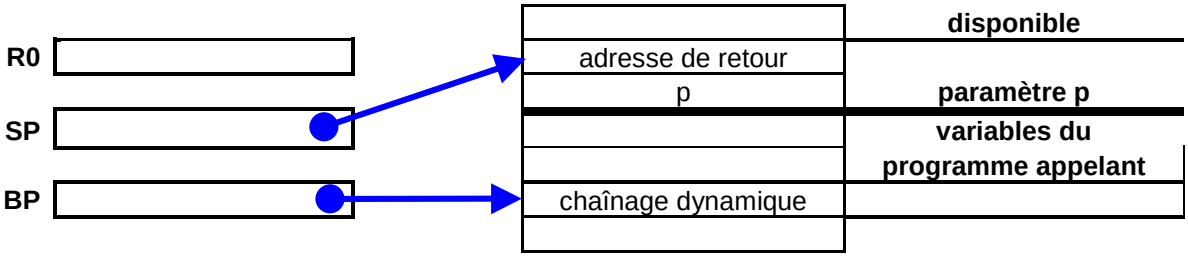



```
// arrête le programme
      // TRP #EXIT_EXC       // EXIT: arrête le programme
           LDW WR, #EXIT_EXC // WR = EXIT_EXC = n° exception de EXIT
           TRP WR            // exécute la trappe logicielle "EXIT"

// gère le redémarrage du programme
      // JEA @main_          // si on redemande l'exécution, saute à main_
           LDW WR, #main_
           JEA (WR)
```

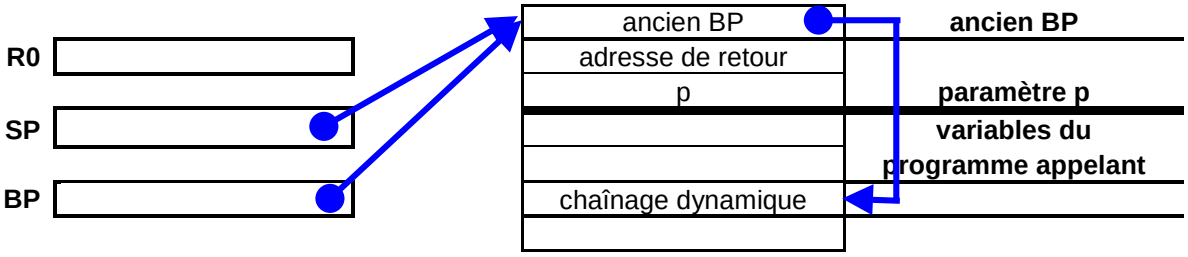
nécessaire sur le simulateur, hors mode "pas à pas" sinon le programme ne s'arrête pas !

5. EXEMPLE DE FONCTION D'AFFICHAGE

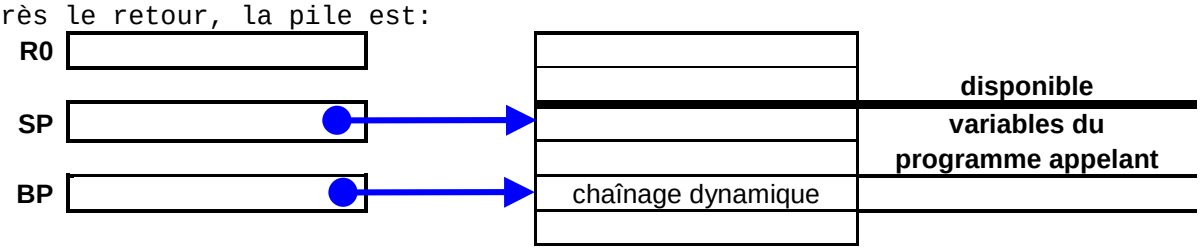
```
// void print(char* p)  // imprime le texte pointé par paramètre p

// juste après l'appel de la fonction print, la pile est:
```




```
// prépare l'environnement de la fonction appelée (prologue) :

print_     LDQ 0, R1      // R1 = taille données locales (ici 0) de fonction appelée

    // LINK (R1)      // crée et lie l'environnement de fonction appelée
           ADQ -2, SP     // décrémente le pointeur de pile SP
           STW BP, (SP)   // sauvegarde le contenu du registre BP sur la pile
           LDW BP, SP     // charge contenu SP ds BP qui pointe sur sa sauvegarde
           SUB SP, R1, SP // réserve R1 octets sur la pile pour la variable locale z
```




```
// charge R0 avec le paramètre p de déplacement 4
     // LDW R0, (BP)4 // R0 = M[BP + 4]
           LDW R0, BP          // R0 = BP
           ADQ 4, R0           // R0 pointe sur p
           LDW R0, (R0)        // R0 = p = adresse du début du texte à afficher

// affiche texte pointé par R0
     // TRP WR, #WRITE_EXC // lance trappe n° WRITE_EXC: affiche texte d'adresse R0
           LDW WR, #WRITE_EXC // on suppose que symbole WRITE_EXC déjà défini
           TRP WR             // lance trappe dont n° dans WR

// fin de la fonction (épilogue) :

    // UNLINK
           LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
           LDW BP, (SP)  // charge BP avec ancien BP
           ADQ 2, SP     // ancien BP supprimé de la pile

    // RTS  // retour au programme appelant:
           LDW WR, (SP)  // charge WR avec l'adresse de retour
           ADQ 2, SP     // incrémente le pointeur de pile SP
           JEA (WR)      // saute à l'instruction d'adresse absolue dans WR

après le retour, la pile est:
```



6. EXEMPLE D'AFFICHAGE

6.1. Exemple de code source C :

```
void main(void)
{
print("Hello world !  ");
print("Il fait beau ...");
}
```

définition du programme principal main.

Affiche "Hello world ! Il fait beau ... " avec la fonction ***prédéfinie*** void print(char* p) précédente.

La fonction doit normalement être déclarée avant son utilisation, afin que le compilateur puisse facilement vérifier la syntaxe de son appel. En C, ceci est usuellement fait avec un prototype, par exemple ici :

```
void print(char* p);
```

Toutefois, ici on suppose que cette fonction print (cf. §5) est prédéfinie au début ou à la fin du fichier en langage d'assemblage produit.

On peut de même définir en langage d'assemblage[4] :

```
void print_int_d(int i); /* imprime un entier i en décimal */
```

Dans la fonction main, il y a des chaînes de caractères constantes "Hello world !" et `"Il fait beau ..."`

Où faut-il les placer ?

o dans la zone d'informations locale de la pile ?
o au dessus de la pile ?$
o ailleurs en mémoire ?

Une solution simple pour placer une chaîne de caractères ***constante*** comme "Hello world !" serait de la mentionner dans le programme en langage d'assemblage, avant le point d'entrée en utilisant la **directive d'assemblage string**.

string réserve une zone de taille paire et y place la chaîne de caractère indiquée, terminée par le caractère NUL=0,
puis éventuellement par un caractère supplémentaire pour que la zone réservée ait un nombre pair de caractères;
ce dernier point est ***à vérifier, sinon il faut faire suivre de :***

```
$ = (($+1)/2)*2) // ce qui suit sera placé à une adresse paire
```

Normalement, pour un assembleur complet, cette chaîne devrait être définie dans une section de données,
afin qu'elle soit ensuite chargée dans un segment (ou des pages) de données au moment du chargement du programme dans la mémoire par le système opérateur.

Mais, par soucis de simplification, il n'y a pas de telle disposition dans microPIUP/ASM.

La directive d'assemblage :

TOTO string "Hello !"

place en mémoire (juste après ce qui précède) :

TOTO	H	e
	l	l
	o	SP
	!	NUL

Le symbole TOTO représente donc ensuite l'adresse du début de la chaîne de caractères.

---

[4] Toutefois, si microC gère les pointeurs, il est alors facile de définir cette fonction en C avec la fonction itoa qui peut peut être définie en C aussi. :

```
char* itoa(int i , char* string, int radix);
/* convertit un entier en une chaîne de caractères placée à partir de l'adresse string */
```

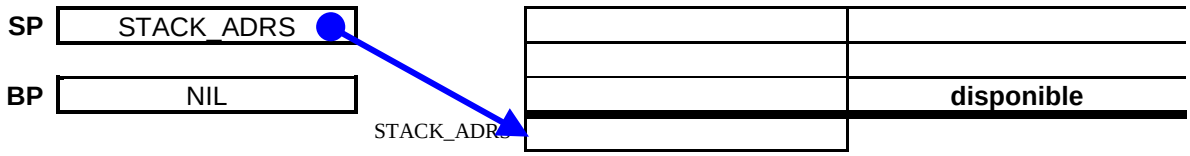
6.2. Programme d'affichage résultant en langage d'assemblage CISC

```
EXIT_EXC    EQU   64              // n° d'exception de EXIT
READ_EXC    EQU   65              // n° d'exception de READ  (lit 1 ligne)
WRITE_EXC   EQU   66              // n° d'exception de WRITE (affiche texte)
STACK_ADRS  EQU   0x1000          // base de pile en 1000h   (par exemple)
LOAD_ADRS   EQU   0xF000          // adresse de chargement de l'exécutable
NIL         EQU   0               // fin de liste: contenu initial de BP

// ces alias permettront de changer les réels registres utilisés
SP          EQU   R15             // alias pour R15: Stack Pointer (pointeur de pile)
WR          EQU   R14             // alias pour R14: Work Register (reg. de travail)
BP          EQU   R13             // alias pour R13: frame Base Pointer (point. envir.)
                                  // R12, R11 réservés
                                  // R0 pour résultat de fonction
                                  // R1 ... R10 disponibles

            ORG   LOAD_ADRS       // adresse de chargement
            START main_           // adresse de démarrage
```

à placer pour un programme ASM complet

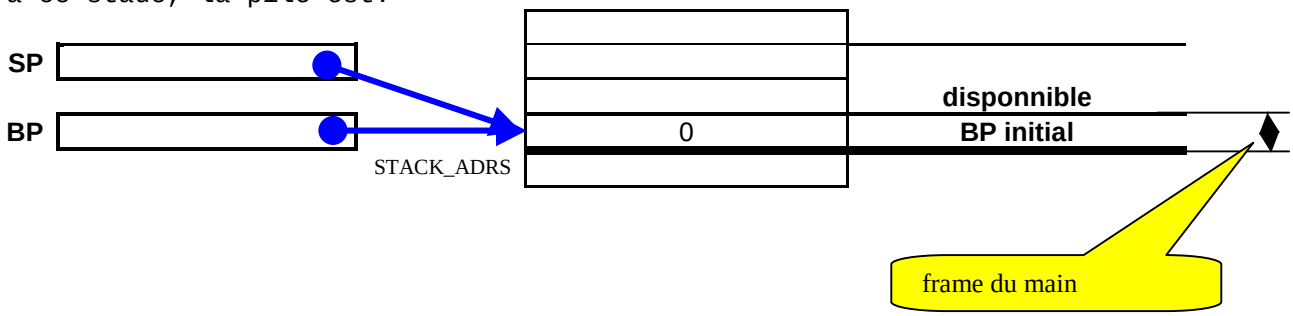
```
// la fonction print peut être prédéfinie ici ou à la fin du fichier


// PROGRAMME PRINCIPAL

// void main(void)
// {

// CHAINES DE CARACTERES CONSTANTES DE MAIN

// définies juste avant le point d'entrée main_
STRING0     string "Hello world !"     // place la chaîne terminée par NUL
STRING1     string "Il fait beau ..."  // place la chaîne terminée par NUL

main_       LDW SP, #STACK_ADRS            // charge SP avec STACK_ADRS
            LDQ NIL, BP                    // charge BP avec NIL=0
```

initialisation de SP et BP ici pour simplifier; ne font usuellement pas partie de main mais de l'OS avant.




```
// prépare l'environnement (frame) du programme principal qui n'a aucune variable;

            LDQ 0, R1       // R1 = taille données locales prog. principal:
                            // rien ici

   // LINK (R1)         // crée et lie l'environnement du prog. principal
            ADQ -2, SP      // décrémente le pointeur de pile SP
            STW BP, (SP)    // sauvegarde le contenu du registre BP sur la pile
            LDW BP, SP      // charge contenu SP ds BP qui pointe sur sa sauvegarde
            SUB SP, R1, SP  // réserve R1 octets sur la pile pour variables locales

// à ce stade, la pile est:
```

n'existe pas sur APR



frame du main

```
// print("Hello world !");

// empile les paramètres de la fonction

            LDW R1, #STRING0   // charge adresse de la chaîne n°0 dans R1

            STW R1, -(SP) // empile paramètre p = STRING0 contenu dans R1 :


// appelle la fonction print d'adresse print_ :

            JSR @print_ // appelle la fonction d'adresse print_:


// print("Il fait beau ...");

// empile les paramètres de la fonction

            LDW R1, #STRING1   // charge adresse de la chaîne n°0 dans R1

            STW R1, -(SP) // empile paramètre p = STRING0 contenu dans R1 :


// appelle la fonction print d'adresse print_ :

            JSR @print_ // appelle la fonction d'adresse print_:


// }   accolade fermante de la définition du programme principal main
      // UNLINK
            LDW SP, BP    // charge SP avec contenu de BP: abandon infos locales
            LDW BP, (SP)  // charge BP avec ancien BP
            ADQ 2, SP     // ancien BP supprimé de la pile

// après la fin du programme principal main, la pile est :
```


```
// arrête le programme
            TRP #EXIT_EXC       // EXIT: arrête le programme

// gère le redémarrage du programme
            JEA @main_          // si on redemande l'exécution, saute à main_
```

nécessaire sur le simulateur, hors mode "pas à pas" sinon le programme ne s'arrête pas !

6.3. Programme d'affichage résultant en langage d'assemblage RISC strict

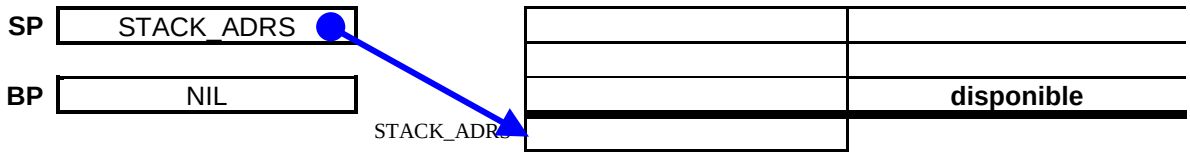
```
EXIT_EXC    EQU   64              // n° d'exception de EXIT
READ_EXC    EQU   65              // n° d'exception de READ  (lit 1 ligne)
WRITE_EXC   EQU   66              // n° d'exception de WRITE (affiche texte)
STACK_ADRS  EQU   0x1000          // base de pile en 1000h   (par exemple)
LOAD_ADRS   EQU   0xF000          // adresse de chargement de l'exécutable
NIL         EQU   0               // fin de liste: contenu initial de BP

// ces alias permettront de changer les réels registres utilisés
SP          EQU   R15             // alias pour R15: Stack Pointer (pointeur de pile)
WR          EQU   R14             // alias pour R14: Work Register (reg. de travail)
BP          EQU   R13             // alias pour R13: frame Base Pointer (point. envir.)
                                  // R12, R11 réservés
                                  // R0 pour résultat de fonction
                                  // R1 ... R10 disponibles

            ORG   LOAD_ADRS       // adresse de chargement
            START main_           // adresse de démarrage
```

à placer pour un programme ASM complet

// la fonction prédéfinie print peut être prédéfinie ici ou à la fin du fichier

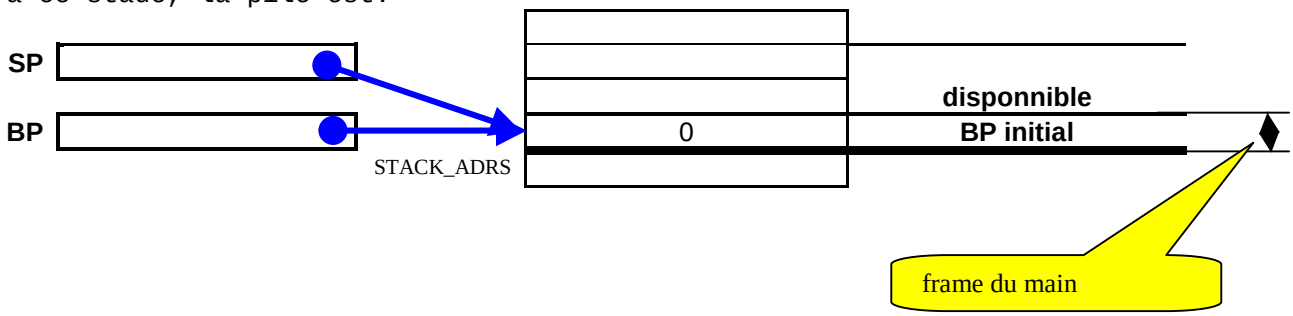
```
// PROGRAMME PRINCIPAL

// void main(void)
// {

// CHAINES DE CARACTERES CONSTANTES DE MAIN

// définies juste avant le point d'entrée main_
STRING0     string "Hello world !"     // place la chaîne terminée par NUL
STRING1     string "Il fait beau ..."  // place la chaîne terminée par NUL

main_       LDW SP, #STACK_ADRS            // charge SP avec STACK_ADRS
            LDQ NIL, BP                    // charge BP avec NIL=0
```

initialisation de SP et BP ici pour simplifier; ne font usuellement pas partie de main mais de l'OS avant.




```
// prépare l'environnement (frame) du programme principal qui n'a aucune variable;

            LDQ 0, R1       // R1 = taille données locales prog. principal:
                            // rien ici

   // LINK (R1)         // crée et lie l'environnement du prog. principal
            ADQ -2, SP      // décrémente le pointeur de pile SP
            STW BP, (SP)    // sauvegarde le contenu du registre BP sur la pile
            LDW BP, SP      // charge contenu SP ds BP qui pointe sur sa sauvegarde
            SUB SP, R1, SP  // réserve R1 octets sur la pile pour variables locales

// à ce stade, la pile est:
```

n'existe pas sur APR



frame du main

```
// print("Hello world !");

// empile les paramètres de la fonction

            LDW R1, #STRING0   // charge adresse de la chaîne n°0 dans R1

      // STW R1, -(SP) // empile paramètre p = STRING0 contenu dans R1 :
            ADQ -2, SP     // décrémente le pointeur de pile SP
            STW R1, (SP)   // sauvegarde le contenu du registre R1 sur la pile


// appelle la fonction print d'adresse print_ :

    // JSR @print_ // appelle la fonction d'adresse print_:
            LDW R1, #print_ // charge l'adresse print_ de la fonction dans R1
            MPC WR         // charge le contenu du PC dans WR
            ADQ 8, WR      // ajoute 8 à WR: WR contient l'adresse de retour
            ADQ -2, SP     // décrémente le pointeur de pile SP
            STW WR, (SP)   // sauvegarde l'adresse de retour sur le sommet de pile
            JEA (R1)       // saute à l'instruction d'adresse absolue dans R1


// print("Il fait beau ...");

// empile les paramètres de la fonction

            LDW R1, #STRING1   // charge adresse de la chaîne n°0 dans R1

      // STW R1, -(SP) // empile paramètre p = STRING0 contenu dans R1 :
            ADQ -2, SP     // décrémente le pointeur de pile SP
            STW R1, (SP)   // sauvegarde le contenu du registre R1 sur la pile


// appelle la fonction print d'adresse print_ :

    // JSR @print_ // appelle la fonction d'adresse print_:
            LDW R1, #print_ // charge l'adresse print_ de la fonction dans R1
            MPC WR         // charge le contenu du PC dans WR
            ADQ 8, WR      // ajoute 8 à WR: WR contient l'adresse de retour
            ADQ -2, SP     // décrémente le pointeur de pile SP
            STW WR, (SP)   // sauvegarde l'adresse de retour sur le sommet de pile
            JEA (R1)       // saute à l'instruction d'adresse absolue dans R1

// }   accolade fermante de la définition du programme principal main
      // UNLINK
            LDW SP, BP     // charge SP avec contenu de BP: abandon infos locales
            LDW BP, (SP)   // charge BP avec ancien BP
            ADQ 2, SP      // ancien BP supprimé de la pile

// après la fin du programme principal main, la pile est :
```


```
// arrête le programme
      // TRP #EXIT_EXC       // EXIT: arrête le programme
            LDW WR, #EXIT_EXC // WR = EXIT_EXC = n° exception de EXIT
            TRP WR            // exécute la trappe logicielle "EXIT"

// gère le redémarrage du programme
      // JEA @main_          // si on redemande l'exécution, saute à main_
            LDW WR, #main_
            JEA (WR)
```

nécessaire sur le simulateur, hors mode "pas à pas" sinon le programme ne s'arrête pas !