

Architecture des ordinateurs

Elements de correction
sur le TD3

Rappels boucles

- En assembleur pour obtenir une boucle "répéter n fois" il est plus simple d'avoir un compteur qui décompte.
Exemple : répéter 10 fois

```
LDR  r0,=10      @ initialiser compteur à 10

maBoucle:      @ étiquette de début
-----
-----      @ code à répéter 10 fois
-----

SUBS  r0,r0,#1  @ décrémenter compteur...
BNE   maBoucle @ boucler si compteur ≠ 0
-----
-----      @ code après la boucle
```

Rappels boucles

- La boucle précédente se comporte comme un *do while* en C :

```
r0=10;    // initialiser compteur à 10

do        // début de boucle
{
  -----
  ----- // code à répéter 10 fois
  -----
  r0=r0-1; // décrémenter compteur...
} while (r0!=0); // boucler si compteur ≠ 0
-----
----- // code après la boucle
```

Rappels boucles

- Le choix du registre utilisé comme compteur est libre (r0...r12)
- Les valeurs successives du compteur **dans** la boucle seront 10 9 8 7 6 5 4 3 2 1 (dernier passage)
- à la **fin** du dernier passage le compteur passe à 0 et le branchement en début de boucle n'est pas pris (on sort de la boucle)
- C'est parce qu'on décrémente r0 avec SUBS juste avant le BNE que la boucle "sait" que r0 est le compteur de boucle
- SUBS met à jour les codes conditions (NZCV) :
 - si le résultat de la décrémentation est non nul alors
 - Z mis à 0 donc BNE est exécuté (on boucle)
 - si le résultat de la décrémentation est nul alors
 - Z mis à 1 donc BNE n'est pas exécuté (ne pas boucler)

Rappels boucles

- Pour compter en sens direct (0 1 2 ... 9) le plus simple est d'utiliser un **autre** registre qui compte en même temps

```
LDR  r1,=0      @ initialiser cpt direct à 0
LDR  r0,=10     @ initialiser cpt boucle à 10
```

```
maBoucle:      @ étiquette de début
-----
-----      @ code à répéter 10 fois
-----
ADD    r1,r1,#1 @ incrémenter cpt direct
SUBS   r0,r0,#1 @ décrémenter cpt boucle
BNE    maBoucle @ boucler si compteur ≠ 0
-----
-----      @ code après la boucle
```

asmMemset

- Exemple de boucle : sous-programme asmMemset qui remplit une zone mémoire avec des valeurs constantes.

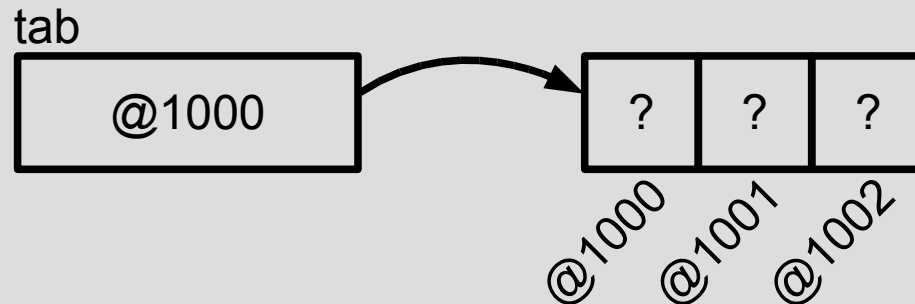
```
// Prototype C
// s adresse du début de zone mémoire
// c valeur de remplissage
// n nombre d'octets consécutifs à remplir
void asmMemset(char *s, char c, int n);

int main(){
    char tab[3];

    // Exemple : init des 3 cases avec val. 65
    asmMemset(tab,65,3);
    ...
}
```

asmMemset

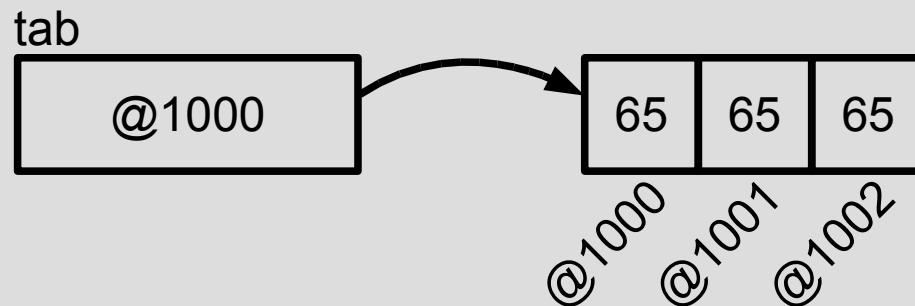
- Avant l'appel le tableau n'est pas initialisé :



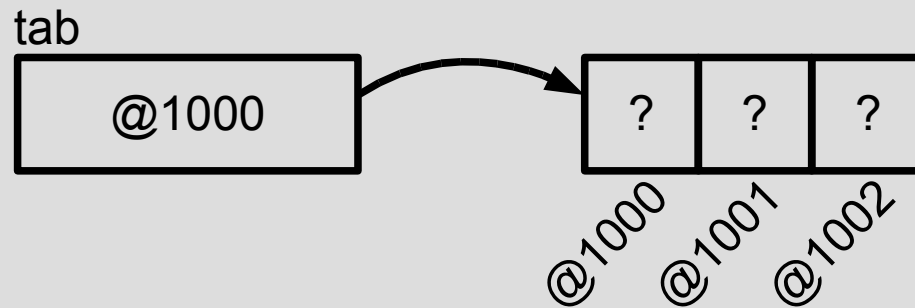
nous supposons
que le tableau est
à l'adresse 1000...

```
asmMemset (tab , 65 , 3) ;
```

- Après l'appel le tableau est initialisé :



asmMemset



```
// Appel au niveau du C  
asmMemset (tab, 65, 3) ;
```

- Au niveau du sous programme assembleur, les valeurs des paramètres seront reçues dans les registres :

r0 vaut 1000

r1 vaut 65

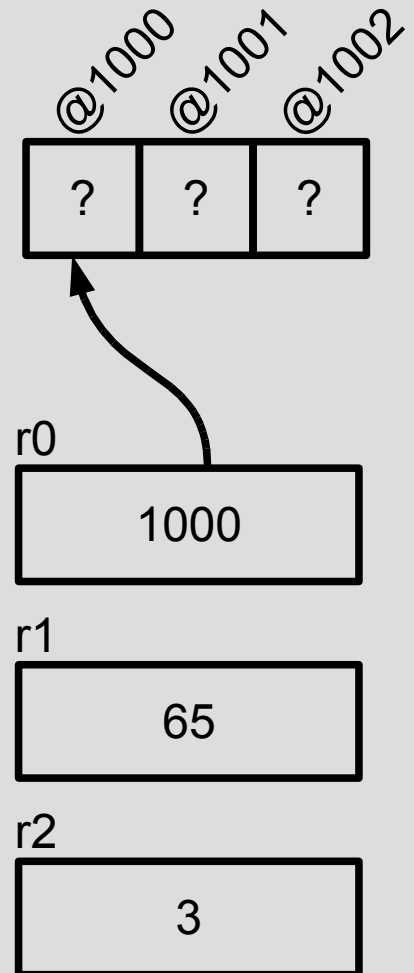
r2 vaut 3

asmMemset

- L'algorithme sera :

Répéter r2 fois

- mettre la valeur de r1
à l'adresse pointée par r0
- incrémenter l'adresse dans r0
(pour pointer sur la case suivante)



asmMemset

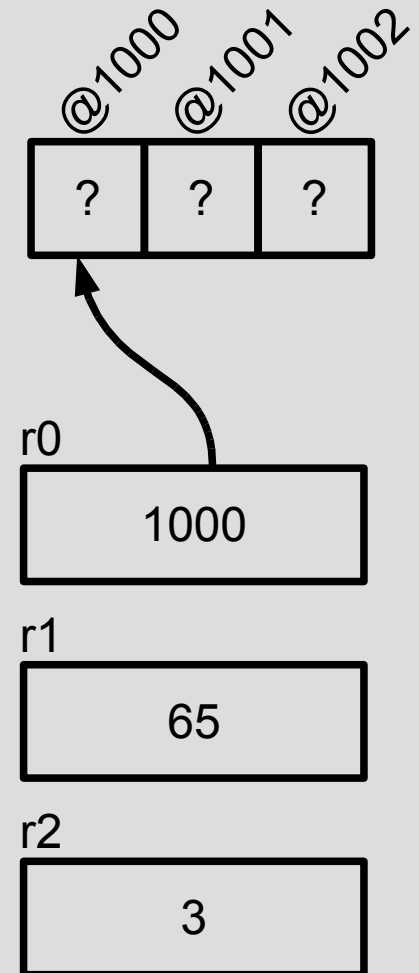
- Code assembleur :

```
@ r2 arrive déjà initialisé,  
@ on attaque directement  
@ la boucle répéter r2 fois  
msBoucle:
```

```
@ mettre r1 à adresse  
@ pointée par r0  
STRB    r1, [r0]
```

```
@ pointer case suivante  
ADD     r0, r0, #1
```

```
@ gestion boucle  
SUBS    r2, r2, #1  
BNE     msBoucle
```



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

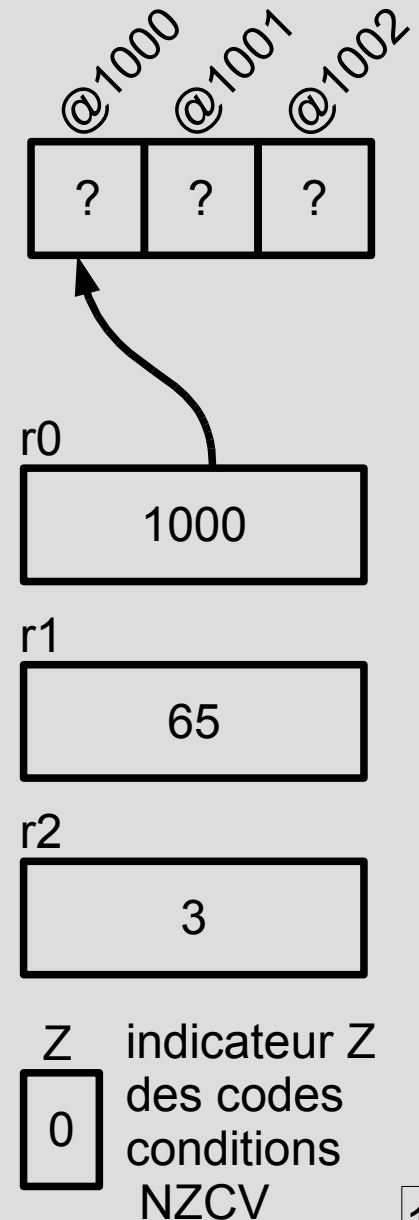
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

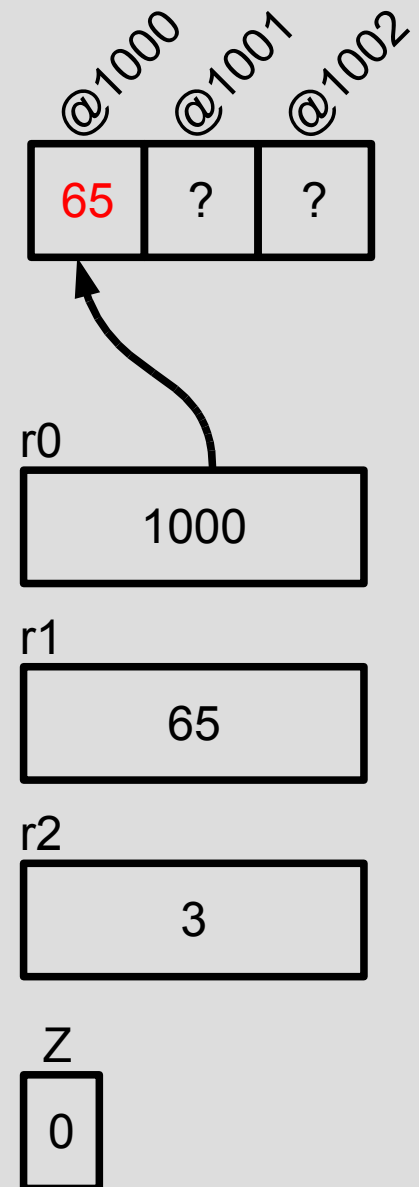
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

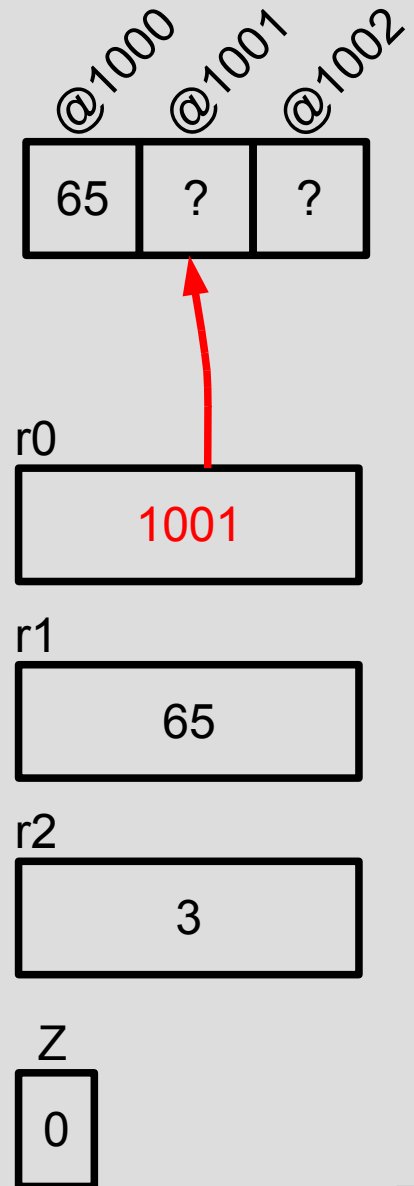
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS    r2, r2, #1
```

```
BNE     msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

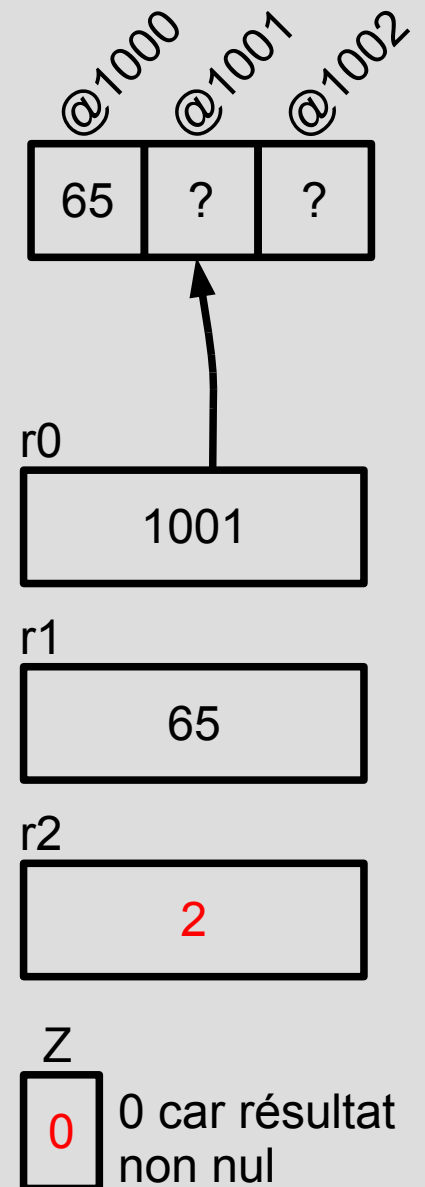
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE     msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

STRB r1, [r0]

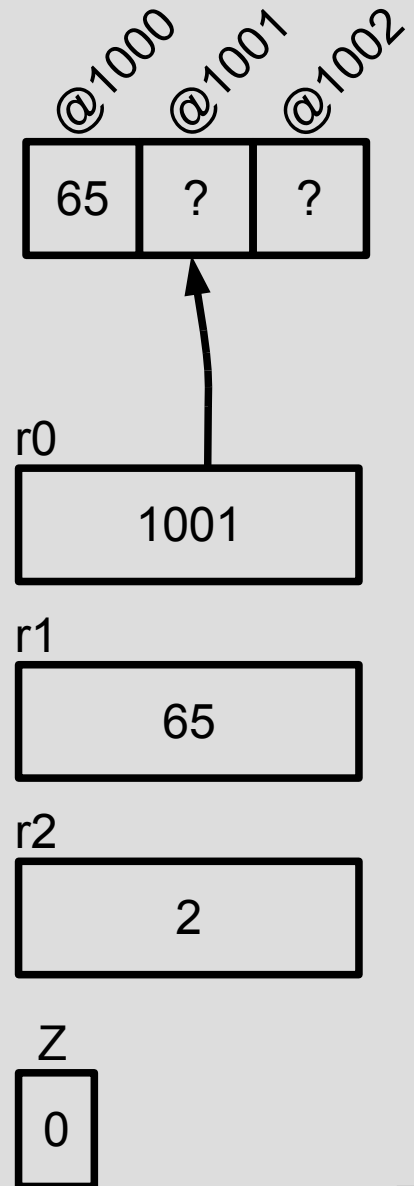
ADD r0, r0, #1

SUBS r2, r2, #1

BNE msBoucle

Branchement effectif
car Z = 0 (postfixe NE)

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

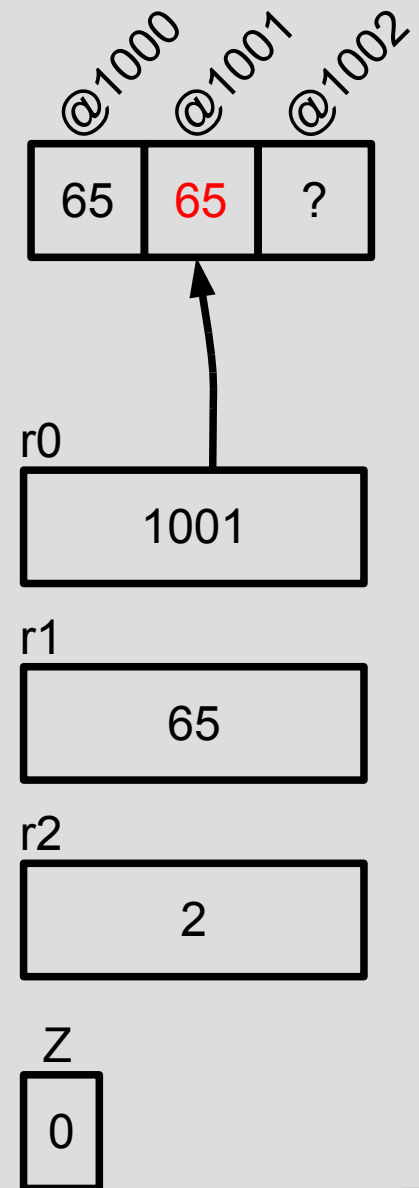
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

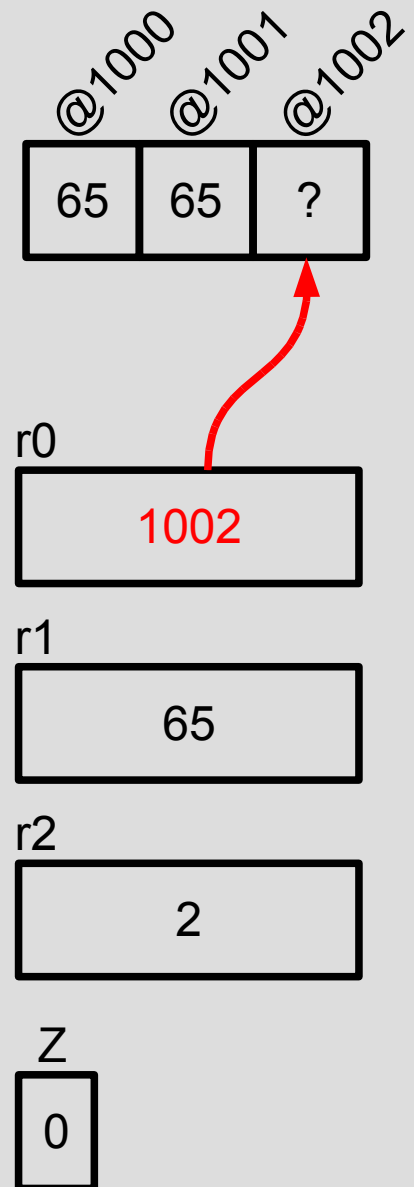
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS    r2, r2, #1
```

```
BNE     msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

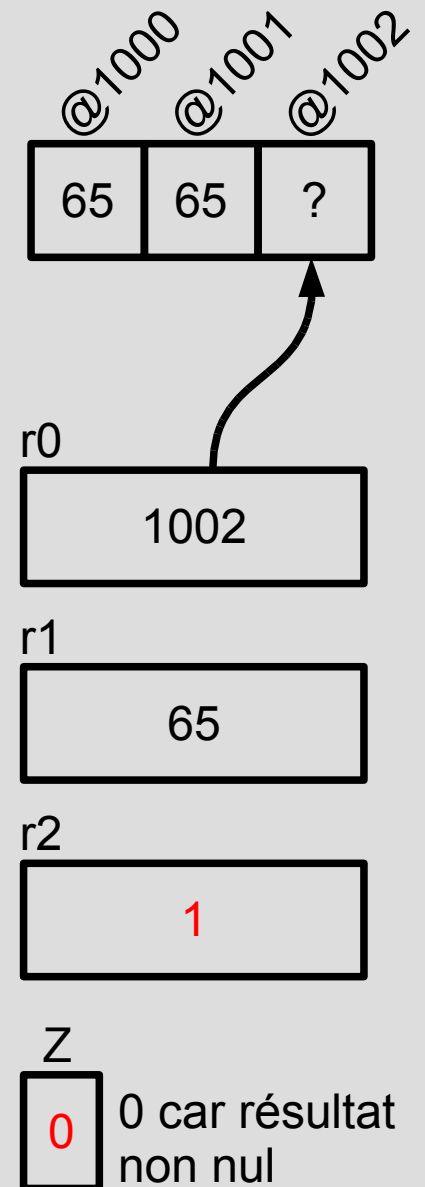
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE     msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

STRB r1, [r0]

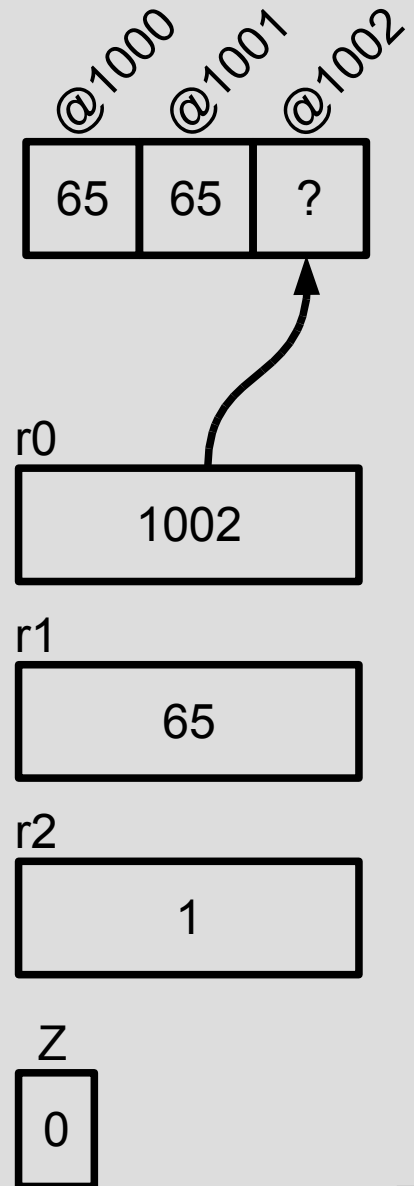
ADD r0, r0, #1

SUBS r2, r2, #1

BNE msBoucle

Branchement effectif
car Z = 0 (postfixe NE)

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

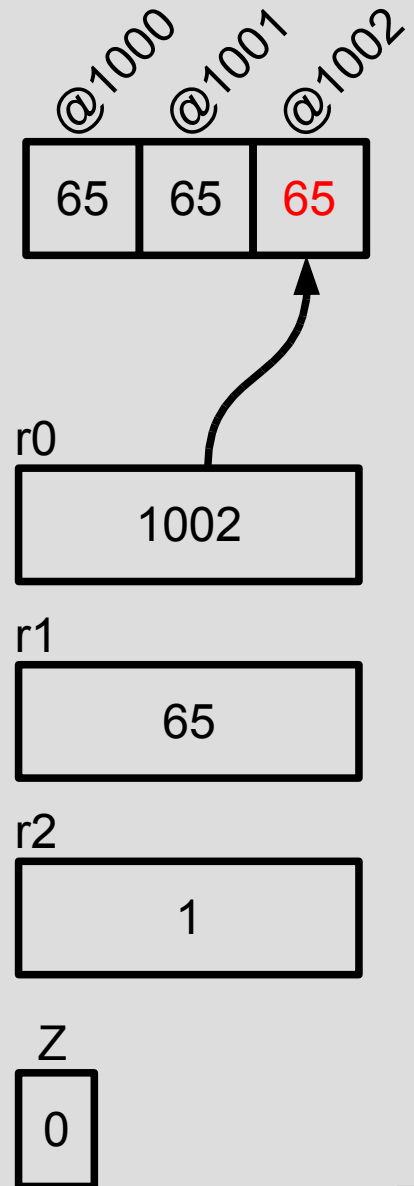
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

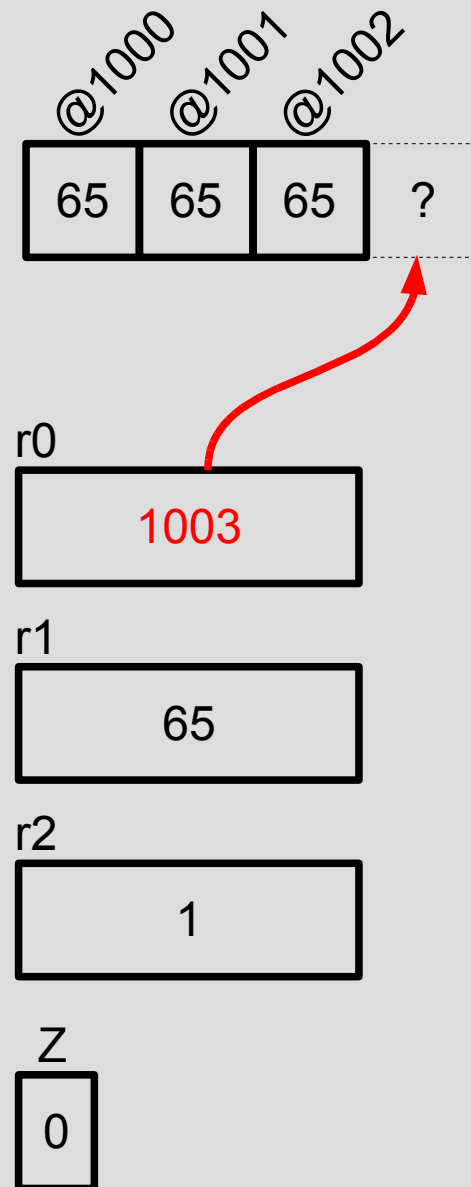
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

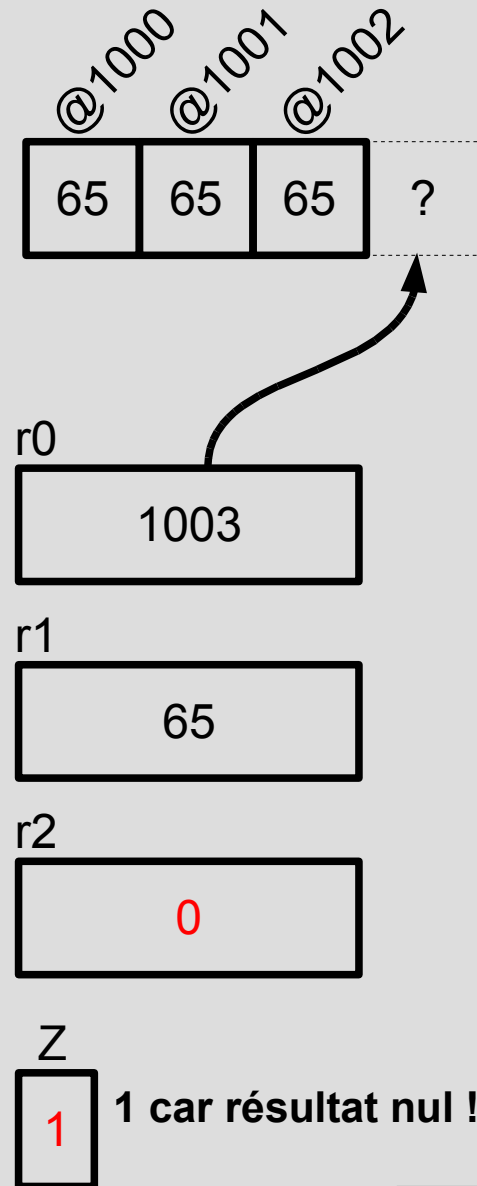
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE     msBoucle
```

@ Fin du sous prog



asmMemset

- Exécution (en rouge étape courante) :

@ Arrivée dans le sous prog

msBoucle:

```
STRB    r1, [r0]
```

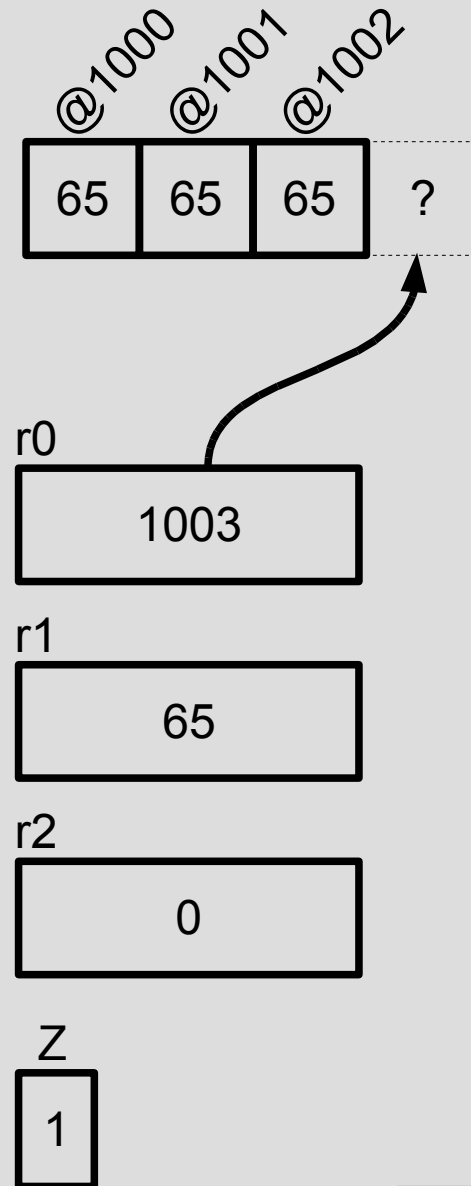
```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE   msBoucle
```

Branchement **non** effectif
car Z = 1 (postfixe NE)
l'exécution continue...

@ Fin du sous prog



asmMemset

- Exécution terminée (retour au main) :

@ Arrivée dans le sous prog

msBoucle:

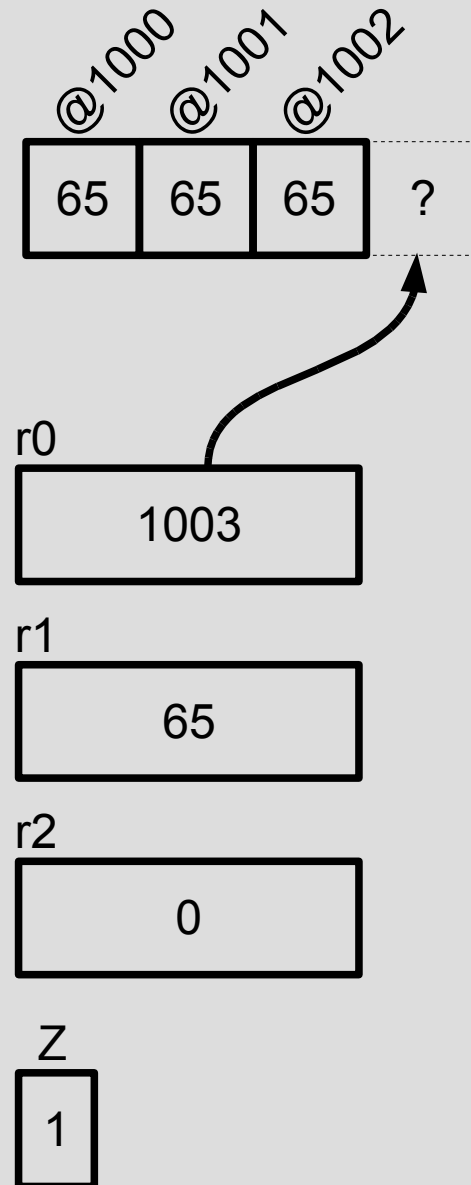
```
STRB    r1, [r0]
```

```
ADD     r0, r0, #1
```

```
SUBS   r2, r2, #1
```

```
BNE    msBoucle
```

@ Fin du sous prog



asmMemcpy

- Autre exemple : sous-programme **asmMemcpy** qui copie une zone mémoire vers une autre

```
// Prototype C
// dest : adresse début de zone destination
// src  : adresse début de zone source
// n nombre d'octets consécutifs à remplir
void asmMemcpy(char *dest, char *src, int n);
```

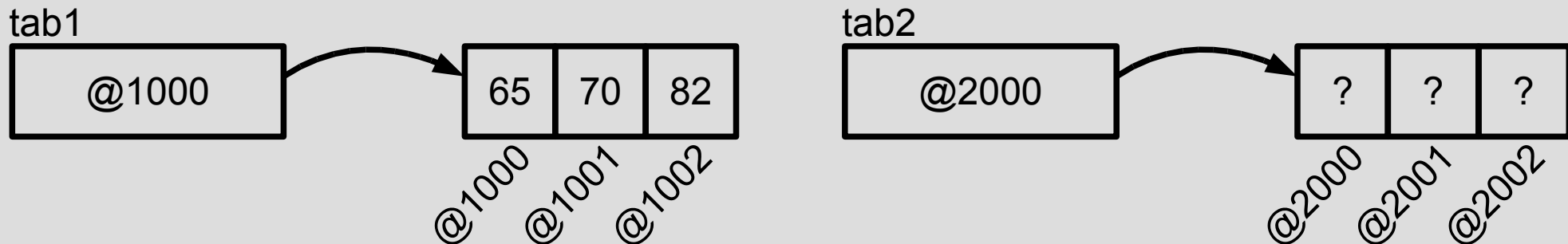
```
int main() {
    char tab1[3]={65,70,82};
    char tab2[3];

    // Exemple : copie des 3 cases
    //           de tab1 vers tab2
    asmMemcpy (tab2, tab1, 3);
    ....
}
```

asmMemcpy

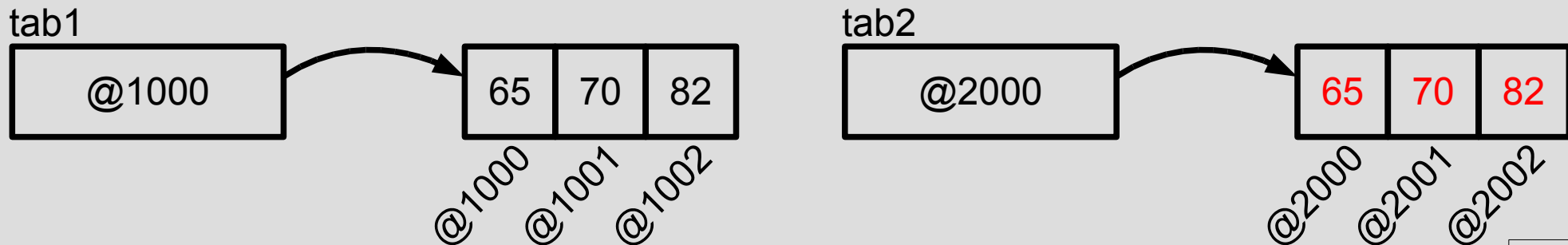
- Avant l'appel

nous supposons que le tab1 est à l'adresse 1000 et tab2 à l'adresse 2000

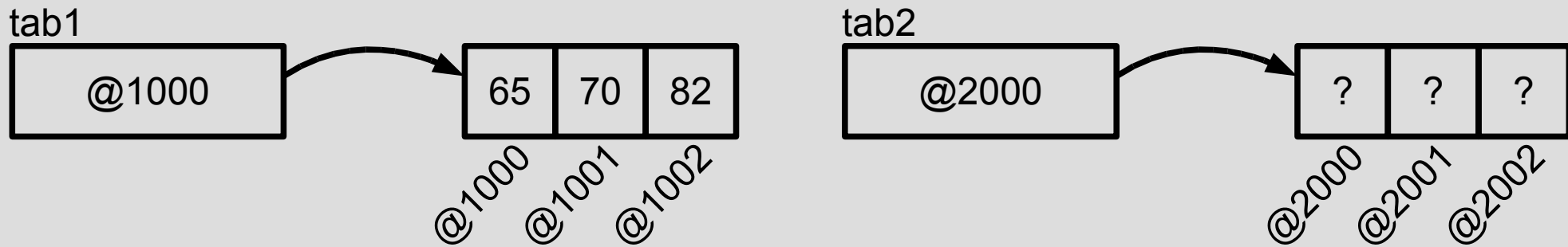


`asmMemcpy (tab2 , tab1 , 3) ;`

- Après l'appel le tableau tab2 a le contenu de tab1 :



asmMemcpy



```
// Appel au niveau du C  
asmMemcpy (tab2 , tab1 , 3) ;
```

- Au niveau du sous programme assembleur, les valeurs des paramètres seront reçues dans les registres :

r0 vaut 2000

r1 vaut 1000

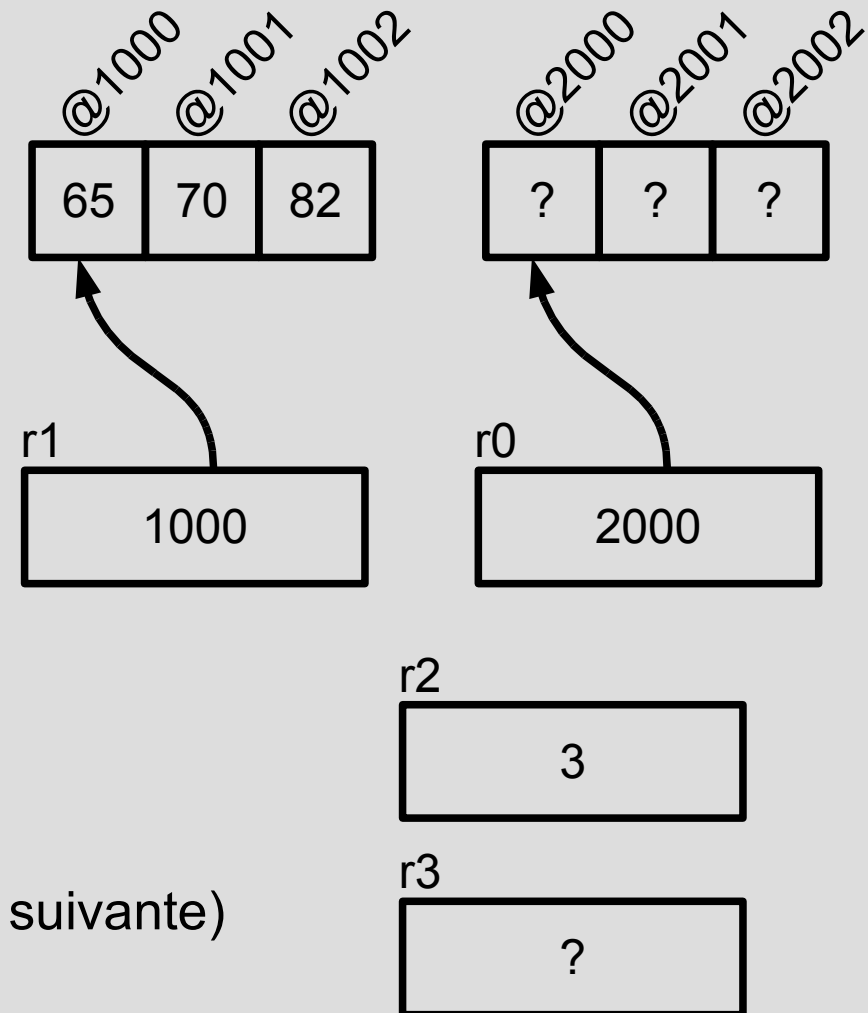
r2 vaut 3

asmMemcpy

L'algorithme sera :

Répéter r2 fois

- mettre valeur adresse pointée par r1 dans r3
- mettre valeur de r3 à l'adresse pointée par r0
- incrémenter l'adresse dans r1 (src suivante)
- incrémenter l'adresse dans r0 (dest suivante)



asmMemcpy

- Code assembleur :

```
@ r2 arrive déjà initialisé (par l'appel)
```

```
mcBoucle:
```

```
    @ copier la case pointée par r1
```

```
    @ vers la case pointée par r0
```

```
LDRB    r3, [r1]
```

```
STRB    r3, [r0]
```

```
    @ cases suivantes
```

```
ADD     r1, r1, #1
```

```
ADD     r0, r0, #1
```

```
    @ gestion boucle
```

```
SUBS    r2, r2, #1
```

```
BNE     mcBoucle
```

asmMemcpy

- Exécution accélérée :

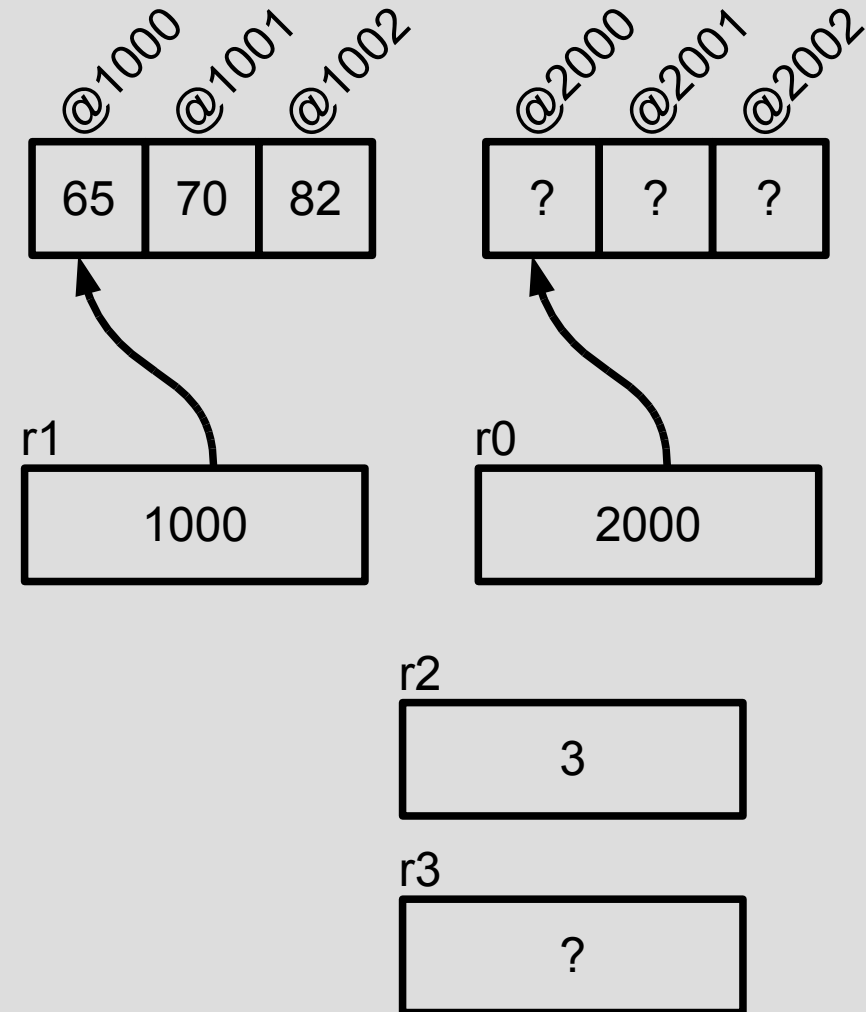
@ Arrivée sous-prog

mcBoucle:

```
LDRB    r3, [r1]
STRB    r3, [r0]

ADD     r1, r1, #1
ADD     r0, r0, #1

SUBS    r2, r2, #1
BNE     mcBoucle
```



@ Fin sous-prog

asmMemcpy

- FIN 1er passage de boucle

@ Arrivée sous-prog

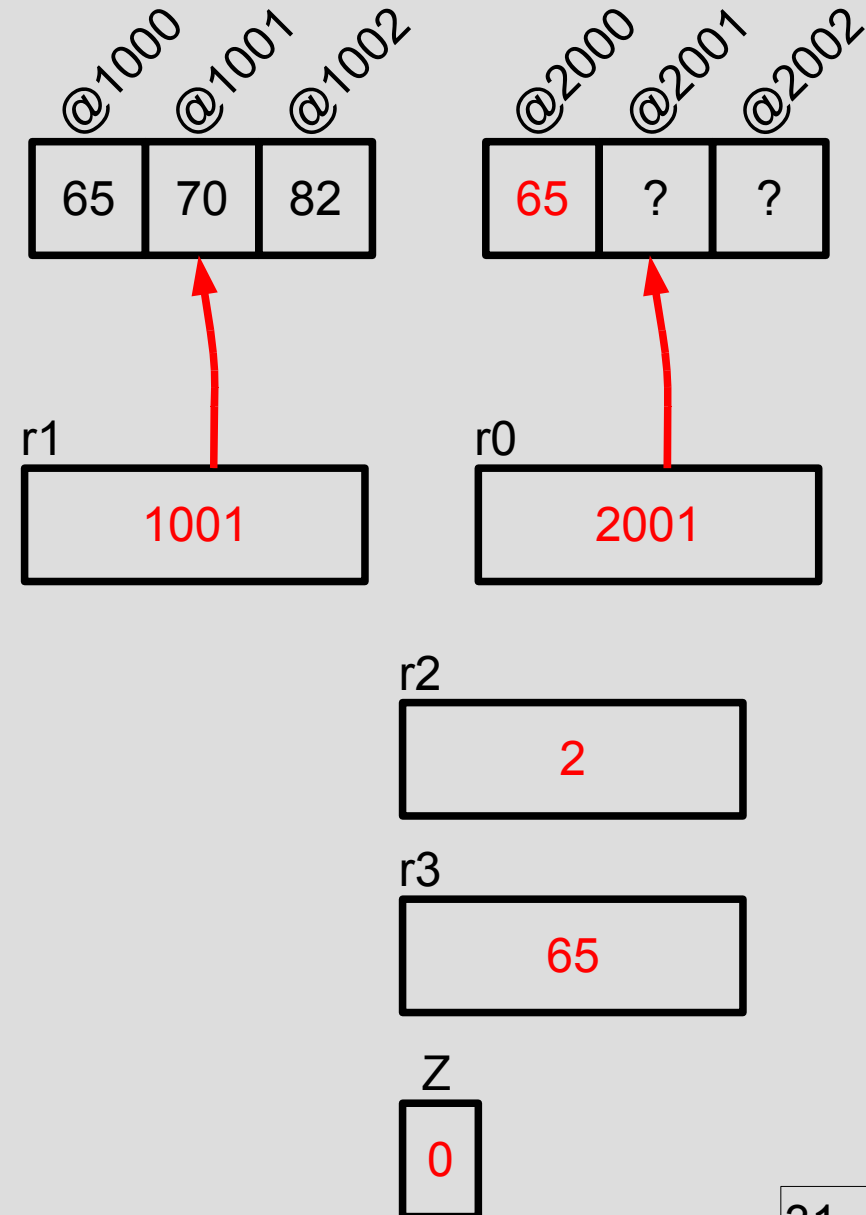
mcBoucle:

```
LDRB    r3, [r1]
STRB    r3, [r0]

ADD     r1, r1, #1
ADD     r0, r0, #1

SUBS    r2, r2, #1
BNE    mcBoucle
```

@ Fin sous-prog



asmMemcpy

- FIN 2eme passage de boucle

@ Arrivée sous-prog

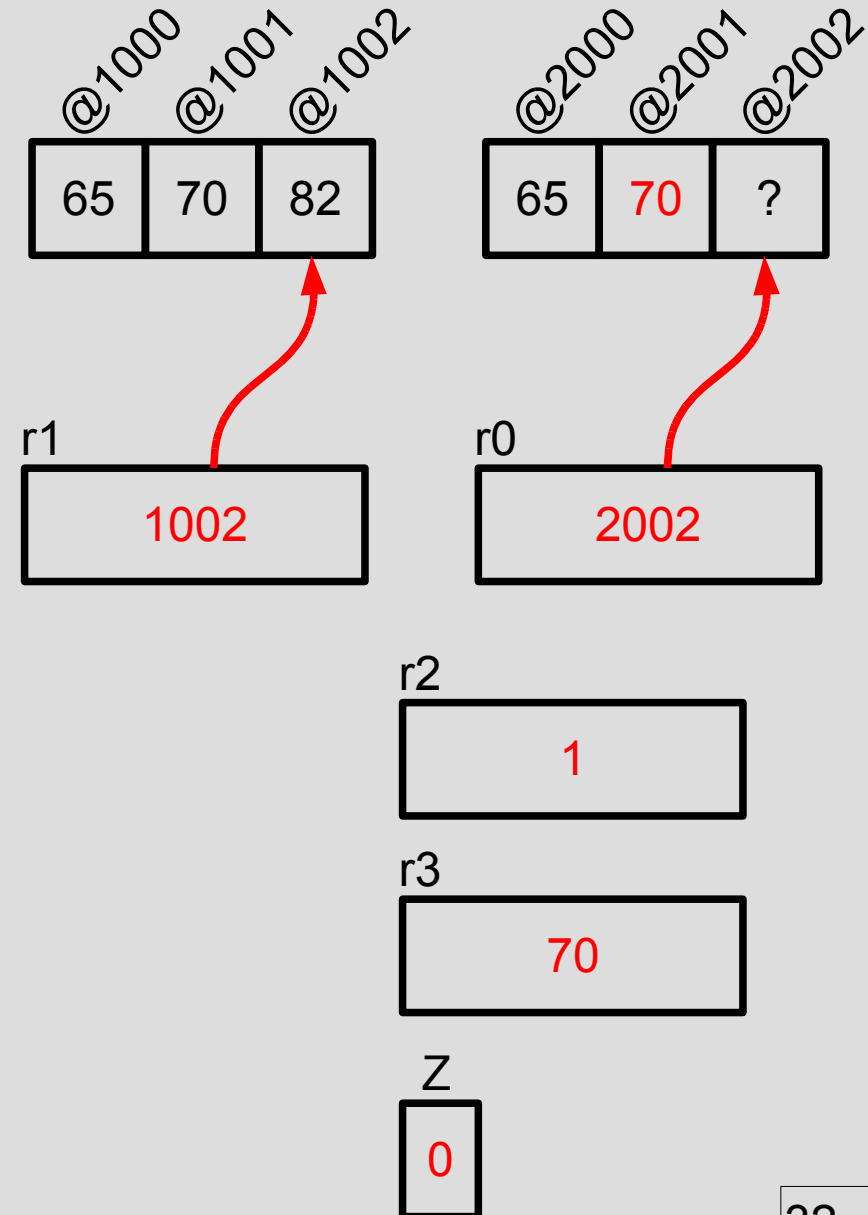
mcBoucle:

```
LDRB    r3, [r1]
STRB    r3, [r0]

ADD     r1, r1, #1
ADD     r0, r0, #1

SUBS    r2, r2, #1
BNE    mcBoucle
```

@ Fin sous-prog



asmMemcpy

- FIN 3eme passage de boucle

@ Arrivée sous-prog

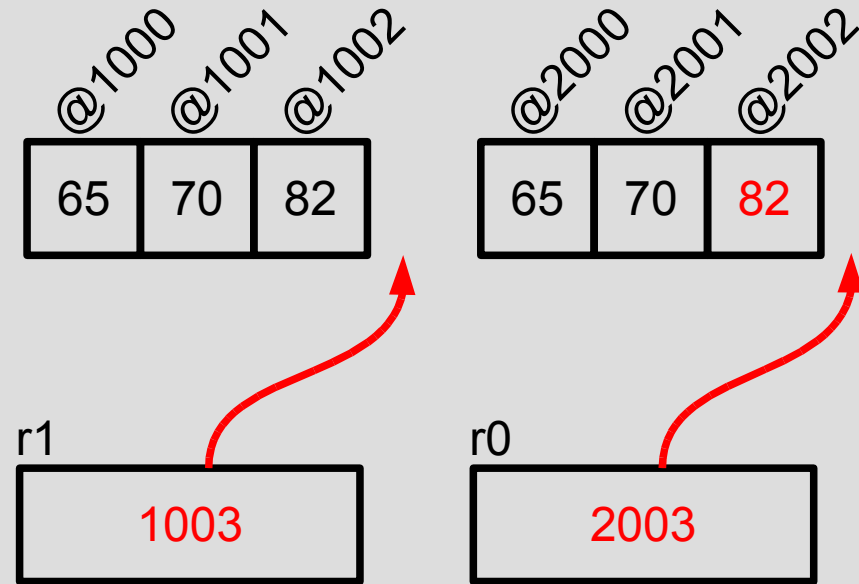
mcBoucle:

```
LDRB    r3, [r1]
STRB    r3, [r0]
```

```
ADD     r1, r1, #1
ADD     r0, r0, #1
```

```
SUBS    r2, r2, #1
BNE    mcBoucle
```

@ Fin sous-prog



Branchement **non** effectif
car Z = 1 (postfixe NE)
l'exécution continue...

r2
0

r3
82

Z
1 1 car résultat
SUBS nul !

asmMemcpy

- Exécution terminée (retour au main)

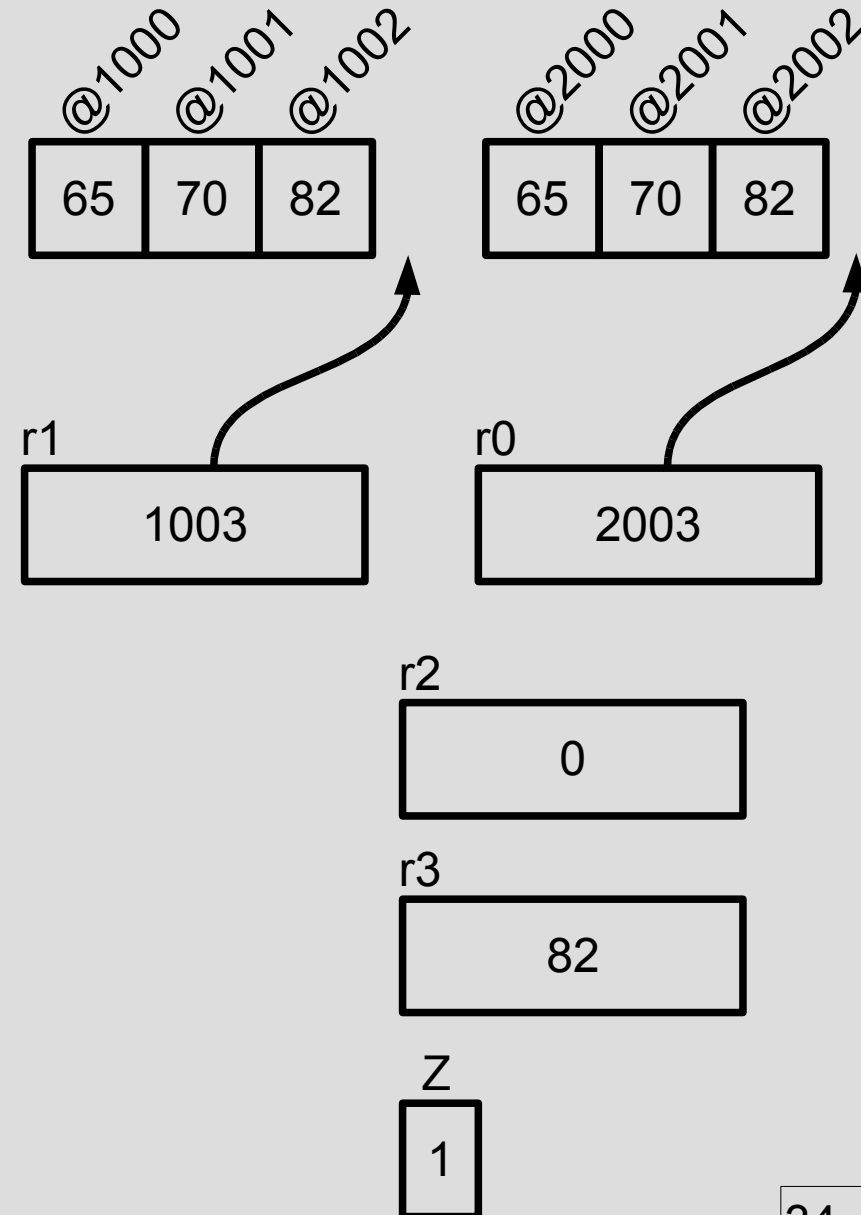
@ Arrivée sous-prog

mcBoucle:

```
LDRB    r3, [r1]
STRB    r3, [r0]

ADD     r1, r1, #1
ADD     r0, r0, #1

SUBS    r2, r2, #1
BNE     mcBoucle
```



@ Fin sous-prog

asmMemcpy

- Pour copier d'un emplacement mémoire vers un autre emplacement mémoire on est obligé de passer par un registre (ici on a choisi r3 qui était disponible)

```
LDRB  r3, [r1]  @ load registre depuis mém.  
STRB  r3, [r0]  @ store registre vers mém.
```

- Ici on accède à un seul octet à la fois, donc on précise B en postfixe pour signaler Byte (octet)
- Pour copier des blocs de 2 octets (short int) il faudrait utiliser **LDRH** et **STRH** (H pour Half Word, demi-mot de 32 bits)
- Pour copier des blocs de 2 octets il faudrait aussi incrémenter les adresses de 2 pour les suivants

```
ADD    r1, r1, #2  
ADD    r0, r0, #2
```