Université Paris-Nord — L1
Institut Galilée — Année 2009-2010
Licence 1 - 1er semestre

Travaux dirigés : programmation en mini-assembleur.

L'objectif de ce TD est de vous familiariser avec le cycle d'exécution d'un processeur et avec la notion de flux d'instructions. Pour cela, il vous est demandé d'écrire de petits programmes dans le langage assembleur présenté en cours et de simuler leur exécution par le processeur.

Correction.

Note aux chargés de TD.

- Dans le texte des corrections se trouvent également quelques explications de correction. Bien entendu, vous n'avez pas à en parler aux étudiants (boucle for, `gcc -S`, etc.).
- Amil est très rudimentaire dans ses possibilités d'écrire des commentaires. Dans cette feuille, il y a des commentaires ligne à ligne, mais aussi une tentative d'indiquer où sont les différents blocs de code dans un style balise xml/html : une ligne avec en commentaire `<bloc1>` signale la première ligne du bloc `bloc1`, une ligne avec en commentaire `</bloc1>` signale la dernière ligne du bloc, et une ligne avec en commentaire `<bloc1/>` signale l'unique ligne du bloc `bloc1`.
- les traces contiennent une colonne *instructions* : cette colonne n'a pas lieu d'être dans les corrections (elle aide juste à relire la trace, qui est générée automatiquement).

Cycles	CP	instruction	r0	10	11
INIT	1		?	5	?
1	2	lecture 10 r0	5		
2	3	ecriture r0 11			5
3	4	stop			

FIGURE 1 – Simulation de la copie de valeur

Jeu d'instructions (simplifié)

`stop`	Arrête l'exécution du programme.
`noop`	N'effectue aucune opération.
`saut i`	Met le compteur ordinal à la valeur i.
`sautpos ri j`	Si la valeur contenue dans le registre i est positive ou nulle, met le compteur de programme à la valeur j.
`valeur x ri`	Initialise le registre i avec la valeur x.
`lecture i rj`	Charge, dans le registre j, le contenu de la mémoire d'adresse i.
`ecriture ri j`	Écrit le contenu du registre i dans la mémoire d'adresse j.
`inverse ri`	Inverse le signe du contenu du registre i.
`add ri rj`	Ajoute la valeur du registre i à celle du registre j (la somme obtenue est placé dans le registre j).
`soustr ri rj`	Soustrait la valeur du registre i à celle du registre j (la différence obtenue est placé dans le registre j).
`mult ri rj`	Multiplie par la valeur du registre i celle du registre j (le produit obtenue est placé dans le registre j).
`div ri rj`	Divise par la valeur du registre i celle du registre j (le quotient obtenue, arrondi à la valeur entière inférieure, est placé dans le registre j).
Instructions plus avancées	
`et ri rj`	Effectue le et bit à bit de la valeur du registre i et de celle du registre j. Le résultat est placé dans le registre j.
`lecture *ri rj`	Charge, dans le registre j, le contenu de la mémoire dont l'adresse est la valeur du registre i
`ecriture ri *rj`	Écrit le contenu du registre i dans la mémoire dont l'adresse est la valeur du registre j.

1 Initialisation de la mémoire

Écrire les programmes répondant aux problèmes suivants :

1. Soit la valeur 8 contenue dans la case mémoire d'adresse 10. Recopier cette valeur à l'adresse 11.

Correction.

```
1       lecture 10 r0
2       ecriture r0 11
3       stop
4
```

2. Sans connaître la valeur contenue à l'adresse 10, écrire 7 à l'adresse 10.

Correction.

```
1         valeur 7 r0
2         ecriture r0 10
3         stop
4
```

3. Sans connaître la valeur contenue à l'adresse 10, incrémenter cette valeur de 1.

Correction.

```
1         lecture 10 r0
2         valeur 1 r1
3         add r1 r0
4         ecriture r0 10
5         stop
6
```

4. Soit une valeur quelconque, a, contenue à l'adresse 10. Initialiser la case d'adresse 11 à $a \times 2 + 1$.

Correction.

```
1         lecture 10 r0
2         valeur 2 r1
3         mult r1 r0
4         valeur 1 r1
5         add r1 r0
6         ecriture r0 11
7         stop
8
```

5. Soit une valeur quelconque, a, contenue à l'adresse 10. Initialiser la case d'adresse 11 à $a \times (2 + 1)$.

Correction.

```
1         lecture 10 r0
2         valeur 2 r1
3         valeur 1 r2
4         add r1 r2
5         mult r2 r0
6         ecriture r0 11
7         stop
8
```

2 Trace de programme assembleur

Nous allons désormais produire une représentation de l'exécution pas à pas de nos programmes. Une *trace* d'un programme assembleur sera un tableau dont chaque ligne correspond à l'exécution d'une instruction par le processeur. Une colonne contiendra le cycle d'horloge du processeur et une autre colonne le compteur de programme. Il y aura également une colonne par registre utilisé dans le programme et par case mémoire où des données sont lues ou écrites par les instructions du programme. Une première ligne, d'initialisation, montrera l'état de ces registres et cases mémoires avant le début du programme. Ensuite, chaque exécution d'instruction sera représentée par une nouvelle ligne du tableau, jusqu'à exécution de l'instruction `stop`. Dans cette ligne nous représenterons le cycle d'horloge du processeur (en commençant à 0 pour la ligne d'initialisation), la valeur du compteur de programme après exécution de l'instruction, et, s'il y a lieu, la valeur du registre ou de la case mémoire modifiée par le programme.

2.1 Exemple de trace

Soit le programme ci-dessous :

```
1   lecture 10 r0
2   valeur 5 r2
3   soustr r2 r0
4   sautpos r0 8
5   valeur 0 r0
6   ecriture r0 11
7   saut 9
8   ecriture r0 11
9   stop
10  14
11  ?
```

On représentera alors sa trace comme ceci :

Instructions	Cycles	CP	r0	r2	10	11
Initialisation	0	1	?	?	14	?
`lecture 10 r0`	1	2	14			
`valeur 5 r2`	2	3		5		
`soustr r2 r0`	3	4	9			
`sautpos r0 8`	4	**8**				
`ecriture r0 11`	5	9				9
`stop`	6	10				

La colonne *Instructions* n'est pas nécessaire, elle sert juste ici à la compréhension, sans cette colonne le mot-clé *initialisation* pourra être placé dans la colonne *Cycles*. Remarquez par exemple qu'il n'y a pas de colonne pour le registre 1, puisque celui-ci n'est pas utilisé.

2.2 Première trace

Faire la trace du même programme où la valeur 14 est remplacée par la valeur 2, à l'adresse mémoire 10.

Quelles sont les valeurs contenues dans les registres 0, 1 et 2 après arrêt sur l'instruction `stop` ?

Correction.

Instructions	Cycles	CP	r0	r2	10	11
Initialisation	0	1	?	?	14	?
`lecture 10 r0`	1	2	14			
`valeur 5 r2`	2	3		5		
`soustr r2 r0`	3	4	9			
`sautpos r0 8`	4	**8**				
`ecriture r0 11`	5	9				9
`stop`	6	10				

La valeur contenue dans le registre 0 est 0, celle contenue dans le registre 1 est indéterminée, celle contenue dans le registre 2 est 5.

3 Exécution conditionnelle d'instructions

À l'aide de l'instruction `sautpos`, écrire les programmes correspondant aux algorithmes[1] suivants et les exécuter sur un exemple (en faisant une trace) afin de tester leur correction :

1. Soient la valeur a à l'adresse 15, b à l'adresse 16. Si $a >= b$ alors écrire a à l'adresse 17 sinon écrire b à l'adresse 17.

   Correction.

```
1   lecture 15 r0 # a
2   lecture 16 r1 # b
3   soustr r1 r0  # r0 vaut a - b
4   sautpos r0 8  # si a - b >= 0 (cad a >= b) on saute sur le alors
5   lecture 16 r0 #  sinon ecrire b a l'adresse 17
6   ecriture r0 17
7   saut 10
8   lecture 15 r0 # alors ecrire a a l'adresse 17
9   ecriture r0 17
10  stop
11  ?
12  ?
13  ?
14  ?
15  23 # a
16  45 # b
17  ?  # <-- valeur de sortie
```

2. Soit l'âge d'une personne à l'adresse 15. Si cette personne est majeure alors écrire 1 à l'adresse 16 sinon écrire 0 à l'adresse 16.

Correction.

```
1   lecture 15 r0 # si age >= 18
2   valeur -18 r1
3   add r0 r1  # r1 vaut age - 18
4   sautpos r1 8
5   valeur 0 r0 # sinon ecrire 0 a l'adresse 16
6   ecriture r0 16
7   saut 10
8   valeur 1 r0 # alors ecrire 1 a l'adresse 16
```

1. Wikipédia : «Un algorithme est un processus systématique de résolution, par le calcul, d'un problème permettant de présenter les étapes vers le résultat à une autre personne physique (un autre humain) ou virtuelle (un calculateur). [...] Un algorithme énonce une résolution sous la forme d'une série d'opérations à effectuer. La mise en œuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique.»

Instructions	Cycles	CP	r0	r1	15	16	17
Initialisation	0	1	?	?	23	45	?
`lecture 15 r0`	1	2	23				
`lecture 16 r1`	2	3		45			
`soustr r1 r0`	3	4	-22				
`sautpos r0 8`	4	5					
`lecture 16 r0`	5	6	45				
`ecriture r0 17`	6	7					45
`saut 10`	7	**10**					
`stop`	8	11					

FIGURE 2 – Simulation du calcul du max de a et b (1)

Instructions	Cycles	CP	r0	r1	15	16	17
Initialisation	0	1	?	?	45	23	?
`lecture 15 r0`	1	2	45				
`lecture 16 r1`	2	3		23			
`soustr r1 r0`	3	4	22				
`sautpos r0 8`	4	**8**					
`lecture 15 r0`	5	9	45				
`ecriture r0 17`	6	10					45
`stop`	7	11					

FIGURE 3 – Simulation du calcul du max de a et b (2)

```
 9  ecriture r0 16
10  stop
11  ?
12  ?
13  ?
14  ?
15  17 # a
16  ?  # <-- valeur de sortie
```

Soit la donnée x d'adresse 15. On veut écrire la valeur absolue de x à l'adresse 16.

Cycles	CP	instruction	r0	r1	15	16
INIT	1		?	?	17	?
1	2	lecture 15 r0	17			
2	3	valeur -18 r1		-18		
3	4	add r0 r1		-1		
4	5	sautpos r1 8				
5	6	valeur 0 r0	0			
6	7	ecriture r0 16				0
7	**10**	saut 10				
8	11	stop				

FIGURE 4 – Simulation du test de la majorité (1)

Cycles	CP	instruction	r0	r1	15	16
INIT	1		?	?	20	?
1	2	lecture 15 r0	20			
2	3	valeur -18 r1		-18		
3	4	add r0 r1		2		
4	**8**	sautpos r1 8				
5	9	valeur 1 r0	1			
6	10	ecriture r0 16				1
7	11	stop				

FIGURE 5 – Simulation du test de la majorité (2)

1. Décrire un algorithme réalisant cette tâche.

   Correction.
   – Si $x < 0$
   – Alors écrire $-x$ à l'adresse 16
   – Sinon écrire x à l'adresse 16

2. Écrire un programme réalisant cette tâche.

   Correction.
   – Si $x < 0$

```
1   lecture 15 r0
2   sautpos r0 6    <-- saute sur le sinon
```

   – Alors écrire $-x$ à l'adresse 16

```
3   inverse r0
4   ecriture r0 16
5   saut 7          <-- saute sur le stop
```

   – Sinon écrire x à l'adresse 16

```
6   ecriture r0 16
```

   – suite du programme

```
7   stop
8   ?
⋮
14  ?
15  5
16  ?
```

3. Construire une trace de votre programme lorsque x vaut 5, puis lorsque x vaut -5.

Correction. Voir les figures 6 et 7.

Instructions	Cycles	CP	r0	15	16
Initialisation	0	1	?	5	?
`lecture 15 r0`	1	2	5		
`sautpos r0 6`	2	**6**			
`ecriture r0 16`	3	7			5
`stop`	4	8			

FIGURE 6 – Trace du calcul de la valeur absolue de 5

Instructions	Cycles	CP	r0	15	16
Initialisation	0	1	?	-5	?
`lecture 15 r0`	1	2	-5		
`sautpos r0 6`	2	3			
`inverse r0`	3	4	5		
`ecriture r0 16`	4	5			5
`saut 7`	5	**7**			
`stop`	6	8			

FIGURE 7 – Trace du calcul de la valeur absolue de -5

4 Boucles d'instructions

4.1 Boucles infinies

1. Avec l'instruction `saut`, écrire un programme qui ne termine jamais.

   Correction.

   ```
   1   saut 1
   2   stop # <- jamais atteint
   ```

2. Avec l'instruction `sautpos`, écrire un programme qui ne termine jamais.

   Correction.

   ```
   1   valeur 0 r0
   2   sautpos r0 1
   3   stop # <- jamais atteint
   ```

4.2 Boucles de calculs itératifs

Soit un entier n d'adresse 15.

1. Écrire un programme qui lorsque $n \geq 0$, écrit la valeur n à l'adresse 16, puis, toujours à l'adresse 16, écrit la valeur $n-1$, puis $n-2$, et ainsi de suite jusqu'à écrire 0, après quoi le programme termine. Si n est négatif, le programme ne fait rien.

   Correction.
   – x vaut n
   – Tant que $x \geq 0$: écrire x puis enlever 1 à x
   Pour la traduction en assembleur de l'algorithme, on imite le schéma de traduction standard du `while (...) { ... }` . Dans cette traduction le test décidant de (la répétition de) l'exécution de la boucle, vient juste après le corps de la boucle plutôt qu'avant. Et il y a juste avant le corps de boucle un saut vers ce test. Ainsi la condition de saut traduit la condition d'exécution (non sa négation) en un minimum de lignes.

   ```
   1   lecture 15 r0
   2   saut 6          <-- saut sur le test d'execution de boucle
   3   ecriture r0 16  <corps de boucle>
   4   valeur -1 r1
   5   add r1 r0       </corps de boucle>
   6   sautpos r0 3    <condition de boucle/>, saut sur le corps de boucle
   7   stop
   8   ?
   9   ?
   10  ?
   11  ?
   12  ?
   13  ?
   14  ?
   ```

Instructions	Cycles	CP	r0	r1	15	16
Initialisation	0	1	?	?	3	?
`lecture 15 r0`	1	2	3			
`saut 6`	2	**6**				
`sautpos r0 3`	3	**3**				
`ecriture r0 16`	4	4				3
`valeur -1 r1`	5	5		-1		
`add r1 r0`	6	6	2			
`sautpos r0 3`	7	**3**				
`ecriture r0 16`	8	4				2
`valeur -1 r1`	9	5		-1		
`add r1 r0`	10	6	1			
`sautpos r0 3`	11	**3**				
`ecriture r0 16`	12	4				1
`valeur -1 r1`	13	5		-1		
`add r1 r0`	14	6	0			
`sautpos r0 3`	15	**3**				
`ecriture r0 16`	16	4				0
`valeur -1 r1`	17	5		-1		
`add r1 r0`	18	6	-1			
`sautpos r0 3`	19	7				
`stop`	20	8				

FIGURE 8 – Décrément pour $n = 3$

```
15   3
16   ?
```

2. Décrire un algorithme calculant la somme des entiers 0, 1, ..., n. Par convention cette somme, $\sum_{i=0}^{n} i = 0+1+\ldots+n$ est nulle lorsque $n < 0$. Écrire le programme correspondant. Le résultat de la sommation sera écrit à l'adresse 16.

Correction. Attention plusieurs réponses correctes sont possibles, mais il faudra donner la correction suivante.

La correction suivante reprend la traduction en assembleur (`gcc -S`) d'un `while (...) ...`, ou d'une boucle `for( i = 0 ; i <= n ; i = i + 1)` en C. L'algorithme consiste en copier la formule de la somme :
– la somme vaut 0, i vaut 0.
– Tant que $i \leq n$, ajouter x_i à la somme puis ajouter 1 à i.
Le programme est alors :

```
1   valeur 0 r1      :: initialisation a zero de l'accumulateur pour la somme
2   valeur 0 r2      :: initialisation a zero de l'indice de boucle, i
3   saut 7           :: saut sur la condition d'execution de boucle
4   add r2 r1        <corps_de_boucle> :: ajoute i a l'accumulateur
5   valeur 1 r3
6   add r3 r2          :: increment de 1 l'indice de boucle </corps_de_boucle>
7   lecture 15 r0    <condition_de_boucle>
```

```
8   soustr r2 r0      :: r0 vaut n - i
9   sautpos r0 4      :: si n >= i saut sur corps_de_boucle </condition_de_boucle>
10  ecriture r1 16  :: ecriture du resultat
11  stop
12  ?
13  ?
14  ?
15  3               <-- n
16  ?               <-- valeur de retour
```

La boucle incrémente i. Dans amil, à cause de la difficulté à écrire des tests tels que $i \leq n$, il serait plus rapide sur cet exercice de décrémenter n jusqu'à 0 comme à la question précédente :

– x vaut n, la somme vaut 0
– Tant que $x \geq 0$, ajouter x à la somme et décrementer x.

Cette réponse est correcte et on peut encourager dans un premier temps les étudiants qui cherchent dans cette direction, mais, pour les préparer à la suite du cours, il faut leur donner la correction qui incrémente l'indice.

Parmi les autres programmes corrects possibles, des étudiants peuvent même penser à la formule de Gauss $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

3. (Optionnel) Tester votre programme en construisant sa trace pour $n = 3$.

Correction. Cette question n'est à traitée qu'au cas où les étudiants auraient eu de grosses difficultés avec la précédente trace. Dans un groupe ou les traces passent mal, la similitude avec l'exercice précédent peut aider à leur faire refaire une trace (dans ce cas on donne la correction). La trace est donnée figure 9.

4.3 Boucles de parcours (exercice facultatif)

Soient n entiers $x_1, \ldots, x_n$ rangées aux adresses $30+1, \ldots, 30+n$. La valeur $n > 0$ est elle même rangée à l'adresse 30.

1. Décrire un algorithme qui effectue la somme $\sum_{i=1}^{n} x_i = x_1 + \ldots + x_n$.

Correction.
– La somme vaut 0
– Pour i allant de 1 à n, ajouter x_i à la somme.

2. Écrire un programme réalisant cet algorithme. (Vous aurez besoin des instructions `lecture *ri rj` et `ecriture ri *rj`). Le résultat sera écrit à l'adresse $30 + n + 1$.

Correction.

```
1   valeur 0 r0       :: initialisation de l'accumulateur
2   valeur 1 r2       :: initialisation de l'indice de boucle
3   saut 9            :: saut sur la condition de boucle
4   valeur 30 r4      <corps_de_boucle>
5   add r2 r4
6   lecture *r4 r5
```

Instructions	Cycles	CP	r0	r1	r2	r3	15	16
Initialisation	0	1	?	?	?	?	3	?
`valeur 0 r1`	1	2		0				
`valeur 0 r2`	2	3			0			
`saut 7`	3	**7**						
`lecture 15 r0`	4	8	3					
`soustr r2 r0`	5	9	3					
`sautpos r0 4`	6	**4**						
`add r2 r1`	7	5		0				
`valeur 1 r3`	8	6				1		
`add r3 r2`	9	7			1			
`lecture 15 r0`	10	8	3					
`soustr r2 r0`	11	9	2					
`sautpos r0 4`	12	**4**						
`add r2 r1`	13	5		1				
`valeur 1 r3`	14	6				1		
`add r3 r2`	15	7			2			
`lecture 15 r0`	16	8	3					
`soustr r2 r0`	17	9	1					
`sautpos r0 4`	18	**4**						
`add r2 r1`	19	5		3				
`valeur 1 r3`	20	6				1		
`add r3 r2`	21	7			3			
`lecture 15 r0`	22	8	3					
`soustr r2 r0`	23	9	0					
`sautpos r0 4`	24	**4**						
`add r2 r1`	25	5		6				
`valeur 1 r3`	26	6				1		
`add r3 r2`	27	7			4			
`lecture 15 r0`	28	8	3					
`soustr r2 r0`	29	9	-1					
`sautpos r0 4`	30	10						
`ecriture r1 16`	31	11						6
`stop`	32	12						

FIGURE 9 – Somme des entiers de 0 à 3

```
7   add r5 r0          :: r0 contient la somme des r2 premiers elements
8   add 1 r2         </corps_de_boucle>
9   lecture 30 r1    <condition_de_boucle>
10  soustr r2 r1       :: r1 vaut n - r2
11  sautpos r3 4     </condition_de_boucle>, saut sur corps_de_boucle
12  valeur 31 r4
13  add r1 r4
14  ecriture r0 *r4
15  stop
16  ?
17  ?
18  ?
19  ?
20  ?
21  ?
22  ?
23  ?
24  ?
25  ?
26  ?
27  ?
28  ?
29  ?
30  3
31  100
32  1
33  10
34  ? <-- resultat
```

3. Écrire un algorithme qui trouve le plus petit des entiers parmi $x_1, \ldots, x_n$. En déduire un programme réalisant cette recherche du minimum. Le résultat sera écrit à l'adresse $30 + n + 1$.

Correction.
- Prendre x_1 comme étant le minimum.
- Pour i allant de 2 à n :
  - Si x_i est plus petit que le minimun alors prendre x_i comme étant le minimum.

```
1   lecture 30 r1
2   lecture 31 r0    :: initialisation du minimum a x1
3   valeur 2 r2      :: initialisation de l'indice de boucle
4   saut 12          :: saut sur la condition de boucle
5   valeur 30 r4     <corps_de_boucle>
6   add r2 r4
7   lecture *r4 r5     :: r5 contient l'element suivant, xi
8   soustr r0 r5       <condition_du_si>  r5 vaut xi - minimum (xi < minimum ?)
9   sautpos r5 11      </condition_du_si>  saute apres le bloc alors
10  lecture *r4 r0     <alors/>
```

Instructions	Cycles	CP	r0	r1	r2	r4	r5	30	31	32	33	34
Initialisation	0	1	?	?	?	?	?	3	100	1	10	?
`valeur 0 r0`	1	2	0									
`valeur 1 r2`	2	3			1							
`saut 9`	3	**9**										
`lecture 30 r1`	4	10		3								
`soustr r2 r1`	5	11		2								
`sautpos r3 4`	6	**4**										
`valeur 30 r4`	7	5				30						
`add r2 r4`	8	6				31						
`lecture *r4 r5`	9	7					100					
`add r5 r0`	10	8	100									
`add 1 r2`	11	9			2							
`lecture 30 r1`	12	10		3								
`soustr r2 r1`	13	11		1								
`sautpos r3 4`	14	**4**										
`valeur 30 r4`	15	5				30						
`add r2 r4`	16	6				32						
`lecture *r4 r5`	17	7					1					
`add r5 r0`	18	8	101									
`add 1 r2`	19	9			3							
`lecture 30 r1`	20	10		3								
`soustr r2 r1`	21	11		0								
`sautpos r3 4`	22	**4**										
`valeur 30 r4`	23	5				30						
`add r2 r4`	24	6				33						
`lecture *r4 r5`	25	7					10					
`add r5 r0`	26	8	111									
`add 1 r2`	27	9			4							
`lecture 30 r1`	28	10		3								
`soustr r2 r1`	29	11		-1								
`sautpos r3 4`	30	**4**										
`valeur 30 r4`	31	5				30						
`add r2 r4`	32	6				34						
`lecture *r4 r5`	33	7				34						

FIGURE 10 – Somme des entiers $x_1, \ldots, x_n$

```
11  add 1 r2          </corps_de_boucle>
12  valeur 0 r3       <condition_de_boucle>
13  add r1 r3
14  soustr r2 r3        :: r3 vaut n - r2
15  sautpos r3 5      </condition_de_boucle> saut sur corps_de_boucle
16  valeur 31 r4
17  add r1 r4         :: r4 vaut 30 + n + 1
18  ecriture r0 *r4
19  stop
20  ?
21  ?
22  ?
23  ?
24  ?
25  ?
26  ?
27  ?
28  ?
29  ?
30  3                 :: n
31  10                :: x1
32  100               :: ...
33  1                 :: xn
34  ? <-- resultat
```

Instructions	Cycles	CP	r0	r1	r2	r3	r4	r5	30	31	32	33	34
Initialisation	0	1	?	?	?	?	?	?	3	10	100	1	?
`lecture 30 r1`	1	2		3									
`lecture 31 r0`	2	3	10										
`valeur 2 r2`	3	4			2								
`saut 12`	4	**12**											
`valeur 0 r3`	5	13				0							
`add r1 r3`	6	14				3							
`soustr r2 r3`	7	15				1							
`sautpos r3 5`	8	**5**											
`valeur 30 r4`	9	6					30						
`add r2 r4`	10	7					32						
`lecture *r4 r5`	11	8						100					
`soustr r0 r5`	12	9						90					
`sautpos r5 11`	13	**11**											
`add 1 r2`	14	12			3								
`valeur 0 r3`	15	13				0							
`add r1 r3`	16	14				3							
`soustr r2 r3`	17	15				0							
`sautpos r3 5`	18	**5**											
`valeur 30 r4`	19	6					30						
`add r2 r4`	20	7					33						
`lecture *r4 r5`	21	8						1					
`soustr r0 r5`	22	9						-9					
`sautpos r5 11`	23	10											
`lecture *r4 r0`	24	11	1										
`add 1 r2`	25	12			4								
`valeur 0 r3`	26	13				0							
`add r1 r3`	27	14				3							
`soustr r2 r3`	28	15				-1							
`sautpos r3 5`	29	16											
`valeur 31 r4`	30	17					31						
`add r1 r4`	31	18					34						
`ecriture r0 *r4`	32	19											1
`stop`	33	20											

FIGURE 11 – Minimum parmi $x_1, \ldots, x_n$