Architecture des ordinateurs

Université de Tours

`Département informatique de Blois`

Examen 2018 - *Durée : 1h30 (une feuille A4 manuscrite autorisée)*

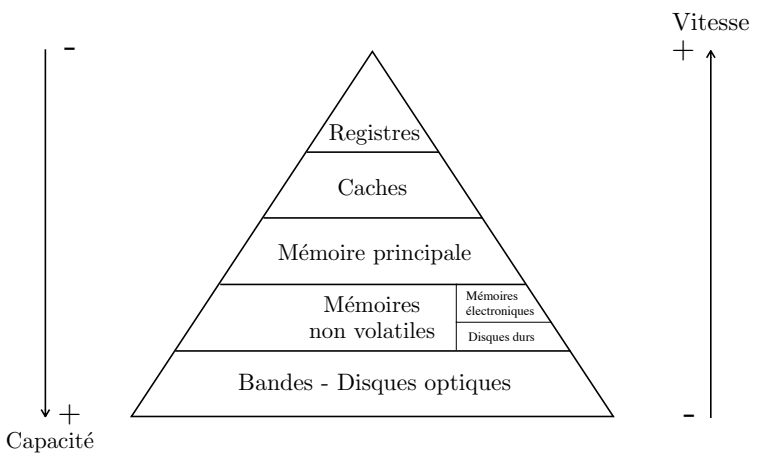
*
* *

Question de cours (3 pts : 1.5 + 1.5)

1. Compléter le schéma synthétique du fonctionnement d'un ordinateur donné en annexe.

‖ Voir annexe.

2. Rappeler la pyramide de hiérarchie mémoire. Dans quelle classe se trouve l'accumulateur de l'UAL ? Expliquer en quelques lignes son utilité.



L'*accumulateur* est un registre. Il sert à stocker les calculs intermédiaires de l'unité arithmétique et logique afin d'éviter de les verser en mémoire principale puis de les recharger ; on évite alors un ralentissement et un engorgement des bus.

Problème 1 (5 pts : 1.5 + 1.5 + 2)

On considère le programme suivant permettant le calcul d'une suite $(u_n)_{n\in\mathbb{N}}$:

```
1. float u_n(int n) {
2.     return n == 0 ? (float) 1.0 / 3 : 4 * u_n(n-1) - 1;
3. }
```

1. Modélisation mathématique :

   (a) Démontrer que : $\forall n \in \mathbb{N}, u_n = \frac{1}{3}$.

La suite modélisée par le programme est une suite récurrente $(u_n)_{n\in\mathbb{N}}$ de la forme :

$$\begin{cases} u_0 & = \frac{1}{3} \\ u_{n+1} & = 4u_n - 1 \end{cases}$$

On veut montrer que la propriété $P(n) : u_n = \frac{1}{3}$ est vraie pour tout $n \in \mathbb{N}$.

- Initialisation (pour $n = 0$)
  Vraie par définition de la suite. $P(0)$ est vraie.
- Hérédité
  On suppose que $\exists n \in \mathbb{N}$ telle que $P(n)$ est vraie. On veut montrer que $P(n) \Rightarrow P(n+1)$.
  $u_{n+1} = 4u_n - 1$
  $= 4 \times \frac{1}{3} - 1$
  $= \frac{4-3}{3} = \frac{1}{3}$
  $P(n+1)$ est vraie.
- Conclusion
  La propriété P est initialisée pour $n = 0$ et est héréditaire. Dès lors, $\forall n \in \mathbb{N}, u_n = \frac{1}{3}$.

(b) Démontrer que le nombre $\frac{1}{3}$ ne peut être représenté de manière exacte sur n bits.

On utilise l'algorithme de représentation des nombres flottants en binaire pour n indéterminé.
On précise que $b = 2, x = \frac{1}{3}$.

Data : $b > 1, x \in [0, 1[$
Result : $(x_1, ..., x_n)$
$i \leftarrow 1$;
do
  $x \leftarrow x \times b$;
  $x_i \leftarrow \lfloor x \rfloor$;
  $x \leftarrow x - x_i$;
  $i \leftarrow i + 1$;
while $x \neq 0$;

Rec

$1/3$		
$\frac{1}{3} \times 2$	$\frac{2}{3}$	0
$\frac{2}{3} \times 2$	$\frac{4}{3}$	1
$\frac{1}{3} \times 2$	$\frac{2}{3}$	0
$\vdots$		

On constate une boucle infinie au sein de l'algorithme. On voit que $\frac{1}{3} = \langle 0101\ 0101\ 0101\ ... \rangle_2$

Démonstration

$$\sum_{n=0}^{\infty} \frac{1}{2^{2+2n}} = \frac{1}{4} \sum_{n=0}^{\infty} \left(\frac{1}{4}\right)^n$$
$$= \frac{1}{4} \times \frac{1}{1-\frac{1}{4}}$$
$$= \frac{1}{4} \times \frac{4}{3} = \frac{1}{3}$$

2. Donner la représentation IEEE 754 de $\frac{1}{3}$ au sein du programme.

Au sein du programme, $\frac{1}{3}$ est codé à l'aide d'un `float`, on utilise 32 bits, dont 1 pour le signe s, 8 pour l'exposant e et 23 pour la mantisse f.

On sait que $\frac{1}{3} = (\langle 0 \rangle_2 , \langle 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ ... \rangle_2)$

- On décale la virgule 2 rangs vers la droite : $E = -2$. Dès lors $e = E + \varepsilon = -2 + 127 = 125 = \langle 0111\ 1101 \rangle_2$,
- La mantisse $f = \langle 0101\ 0101\ 0101\ 0101\ 0101\ 010 \rangle_2$,
- Le signe $s = 0$.

Dès lors :

0	0111 1101	0101 0101 0101 0101 0101 010

3. L'appel `u_n(42)` retourne la valeur 1.9215359×10^{17}. Expliquer la différence entre le résultat théorique et celui donné par le programme. Pouvez-vous estimer la croissance de cette erreur (linéaire, quadratique, exponentielle, autre) ?

   (a) Expliquer briévement la différence entre le résultat théorique et celui donné par le programme. Pouvez-vous estimer la croissance de cette erreur (logarithmique, linéaire, quadratique, exponentielle, autre) ?

   On a vu que la représentation de $\frac{1}{3}$ au sein du programme est incomplète. En particulier, il faudrait une infinité de bits pour modéliser ce nombre. Ainsi, au sein du programme, on commet une erreur $\delta > 0$ d'approximation de la valeur.

   Cette erreur est positive et comme le coefficient multiplicateur de l'erreur est supérieur à 1 (Ici 4), alors l'erreur ne sera pas bornée et va diverger.

   On constate aisément qu'elle est de nature exponentielle. (Voir quetion suivante).

   (b) Sur la base d'une erreur d'approximation $\delta > 0$, déterminer l'expression de l'erreur Δ_n commise pour un appel à la méthode `u_n()` au rang n.

   On a $\Delta_0 = \delta$,
   $\Delta_1 = 4\Delta_0 = 4\delta$,
   $\Delta_2 = 4\Delta_1 = 4^2\Delta_0 = 4^2\delta$
   ...
   Par récurrence, on obtient : $\Delta_n = 4^n\delta$.

Problème 2 (3.5 pts : 1.5 + 2)

L'opérateur *Nand* noté $\uparrow$ est un opérateur très utilisé en électronique et dans la réalisation des microprocesseurs car il forme un système complet de connecteurs à lui seul.

1. Montrer que $x \oplus y = [(x \uparrow y) \uparrow x] \uparrow [y \uparrow (x \uparrow y)]$. On rappelle que l'opérateur $\oplus$ désigne le *OU exclusif* (ou *Xor*).

   On rappelle que : $x \uparrow y = \neg(x \wedge y)$

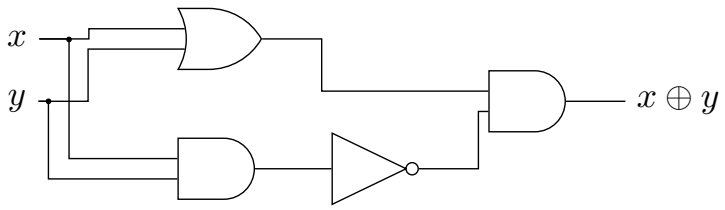
   On rappelle que : $x \oplus y = (x \vee y) \wedge (\neg x \vee \neg y)$
   $= (x \vee y) \wedge \neg(x \wedge y)$

$$
\begin{aligned}
&= (x \vee y) \wedge (x \uparrow y) \\
&= [x \wedge (x \uparrow y)] \vee [y \wedge (x \uparrow y)] \ \textit{Distributivité} \\
&= \neg\left(\neg [x \wedge (x \uparrow y)] \wedge \neg [y \wedge (x \uparrow y)]\right) \ \textit{Double négation} \text{ et } \textit{Loi de de Morgan} \\
&= \neg\left([x \uparrow (x \uparrow y)] \wedge [y \uparrow (x \uparrow y)]\right) \\
&= [x \uparrow (x \uparrow y)] \uparrow [y \uparrow (x \uparrow y)]
\end{aligned}
$$

2. Sur la modélisation de $\oplus$:

   (a) Proposer un circuit bien modélisé de l'opérateur $\oplus$ à l'aide du système d'opérateurs $\{\vee, \wedge, \neg\}$.

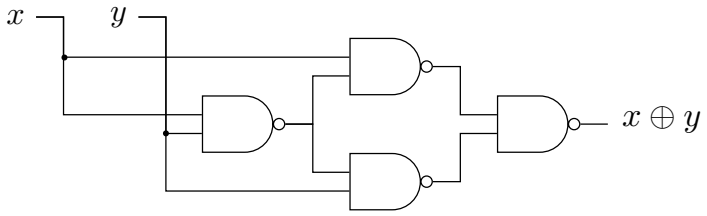
   On utilise le fait que $x \oplus y = (x \vee y) \wedge \neg(x \wedge y)$



   (b) Expliquer pourquoi la modélisation 2.(a) n'est pas satisfaisante. Proposer un circuit logique à l'aide de l'opérateur Nand. Pourquoi cette modélisation est meilleure ?

   Le circuit précédent n'est pas satisfaisant car il ne minimise pas le nombre de portes logiques différentes.

   À l'aide du connecteur Nand, on obtient le circuit suivant :



   Ce circuit est meilleur que le précédent car il contient le même nombre de portes logiques mais utilise uniquement le Nand qui est un système complet ; ceci permet des économies en terme de commande de composants ou en simplicité de gravure des wafers.

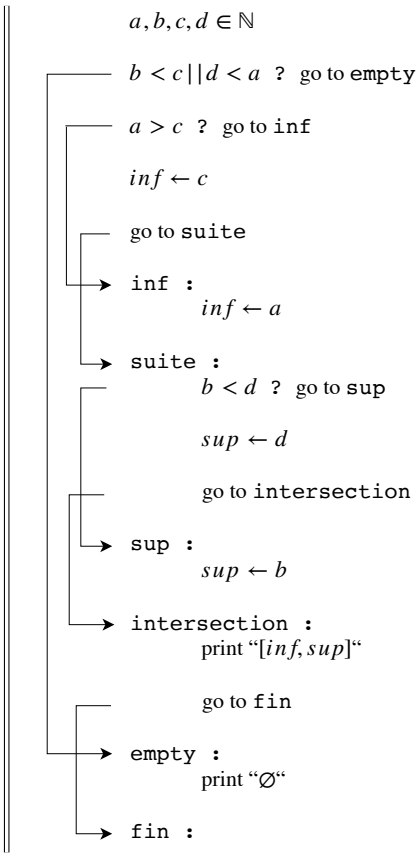
Problème 3 (3.5 pts)

Soient deux intervalles de nombres entiers $[\![a, b]\!]$ et $[\![c, d]\!]$. On précise que $a, b, c, d \in \mathbb{N}$ et on suppose que $a \leq b$ et que $c \leq d$.

1. Écrire la forme linéaire de algorithme de calcul de l'intersection $[\![a, b]\!] \cap [\![c, d]\!]$.

   On sait que si ($b < c$ ou $d < a$) alors $[\![a, b]\!] \cap [\![c, d]\!] = \emptyset$

   Sinon, $[\![a, b]\!] \cap [\![c, d]\!]$ s'obtient à l'aide du max et du min. En effet, l'intersection est l'opérateur qui contraint/restreint le plus l'ensemble, dès lors $[\![a, b]\!] \cap [\![c, d]\!] = [\![\max(a, c), \min(b, d)]\!]$.



2. Compléter l'annexe de programme en assembleur MIPS intersection.asm qui retourne "$\emptyset$" si et seulement si $[\![a, b]\!] \cap [\![c, d]\!] = \emptyset$ et qui retourne "$[x, y]$" avec $[\![x, y]\!] = [\![a, b]\!] \cap [\![c, d]\!]$ sinon.

   On donne en annexe les mnémoniques communs utilisés en MIPS. De plus, le programme intersection.asm pré-défini certaines routines d'affichage, notamment la routine `print_intervalle` qui permet l'impression des registres `$a1` et `$a2` tels que "[`$a1`, `$a2`]". On suppose que les registres `$t0`, `$t1`, `$t2` et `$t3` contiennent respectivement les nombres a, b, c et d.

   Ex : Si $a = 1, b = 5, c = 3, d = 8$, on retourne "[3,5]". Si $a = 1, b = 3, c = 5, d = 8$, on retourne "$\emptyset$".

‖ Voir annexe.

Problème 4 (5 pts : 1.5 + 2.5 + 1)

On rappelle que les caractères ASCII sont contenus dans des octets mais ne sont codés que sur 7 bits, pour qu'on puisse leur adjoindre une parité, ceci afin de sécuriser leur transmission et leur stockage en mémoire.

Si l'on groupe plusieurs caractères ensemble, il est possible d'appliquer une méthode de codage φ par *parité longitudinale et transversale*. Pour un texte x de n caractères, on associe X la forme matricielle de x de taille $n \times 7$ où chaque ligne i correspond au codage binaire du caractère x_i.

Par exemple, le texte $x =$ "chat" possède la matrice associée $X = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$.

Ainsi, $\varphi(X)$ est la matrice de codage par parité longitudinale et transversale résultante de X. Sa taille est de $(n + 1) \times 8$.

1. Soit le texte $x =$ "texte x". Déterminer la matrice $\varphi(X)$ associée à x. On pourra utiliser la table des caractères ASCII donnée en annexe.

L'annexe nous donne le code hexadécimal ASCII des caractères.

Comme $\log_2(16) = 4$, on sait que un chiffre en base 16 correspond à 4 bits. On va pouvoir coder x plus rapidement.

t : 0x74 $= \langle 74 \rangle_{16} = \langle 0111\ 0100 \rangle_2$

e : 0x65 $= \langle 65 \rangle_{16} = \langle 0110\ 0101 \rangle_2$

x : 0x78 $= \langle 78 \rangle_{16} = \langle 0111\ 1000 \rangle_2$

(sp) : 0x20 $= \langle 20 \rangle_{16} = \langle 0010\ 0000 \rangle_2$

Le bit de tête des caractères est inutilisé (comme ils sont codés sur 7 bits). On le retire.

On met x sous forme matricielle puis on applique le codage par parité longitudinale et transversale.

$$\varphi(X) = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

2. Écrire un programme `static int[] encodage(String x)`, en Java ou en C qui encode une suite de caractères ASCII selon la méthode par parité longitudinale et transversale et redonne un tableau d'octets `tab` où `tab[i]` correspond au codage décimal de la ligne i de $\varphi(X)$.

   Ex : Pour $x =$ "chat" la méthode redonne $[198, 209, 195, 232, 60]$.

```java
public static int[] encode(String x) {

    int[] M = new int[x.length() + 1];
    int parite_l = 0;
    for (int i = 0; i < x.length(); i++) {
        int parite_t = 0;
        for (char c = x.charAt(i); c != 0; c >>= 1) {
            parite_t ^= (c & 1);
        }
        M[i] = (x.charAt(i) << 1) | parite_t;
        parite_l ^= M[i];
    }
    M[x.length()] = parite_l;
    return M;

}
```

3. Soit un texte y stocké en mémoire. On donne la matrice $\varphi(Y) = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$.
Que pouvez-vous déduire de cette matrice ? Décoder le texte y correspondant, éventuellement corrigé.

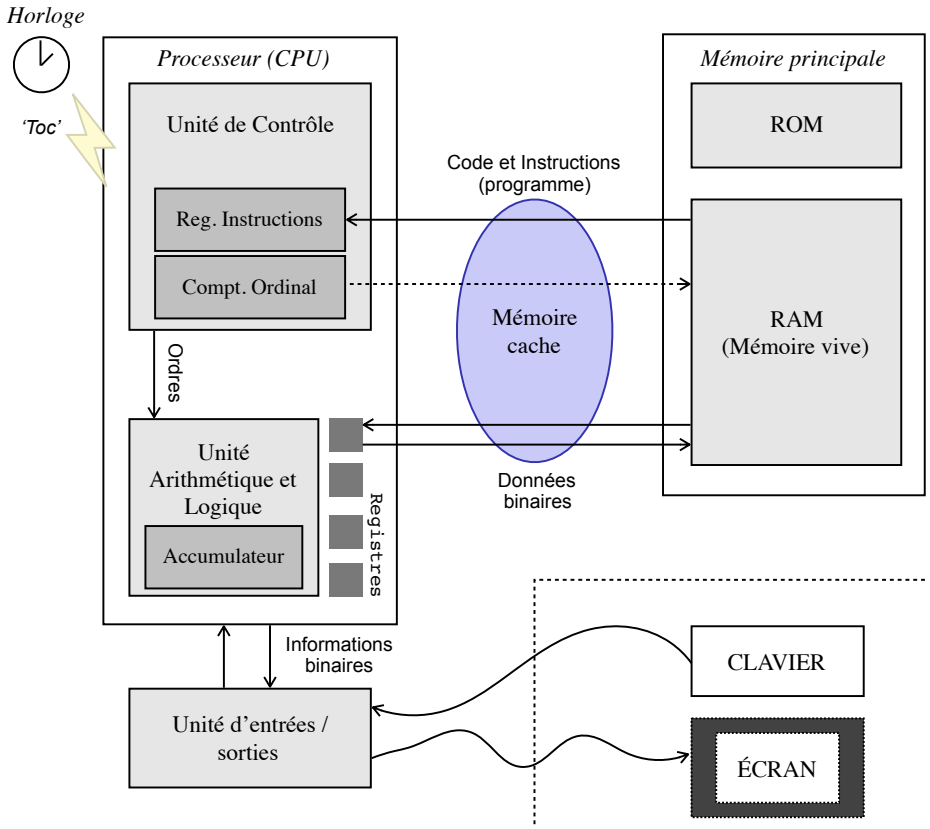
En regardant la parité des lignes et des colonnes 1 et 2, on constate deux anomalies dans la matrice $\varphi(Y)$, notée en rouge.

$$\varphi(X) = \begin{pmatrix} \color{red}{0} & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & \color{red}{0} & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Dès lors, on a :
$$\begin{aligned} y &= \langle 1010100\ 1101111\ 110001\ 1110011\ 1110100 \rangle_2 \\ &= \langle (0101)(0100)\ (0110)(1111)\ (0110)(0001)\ (0111)(0011)\ (0111)(0100) \rangle_2 \\ &= \langle 54\ 6F\ 61\ 73\ 74 \rangle_{16} \\ &= \text{"Toast"} \end{aligned}$$

Annexes

Schéma à compléter : Question de cours



Programme assembleur MIPS : Problème 3

```
##
# @author Clément Moreau
# File_name : intersection.asm
# Description : Calcul de [[a, b]] ∩ [[c, d]].
##
### Data section (Vous pouvez déclarer ici d'autres données et constantes) ###
.data
VIRGULE   : .asciiz ","
EMPTY_SET : .asciiz "Ensemble vide"
CROCHET_G : .asciiz "["
CROCHET_D : .asciiz "]"

### Text section ###
.text
.globl _main_
### Début du programme (à compléter) ###
_main_ :

    li $t0, a
    li $t1, b
    li $t2, c
    li $t3, d
```

```
     # Si (b < c ou d < a) alors [[a,b]] ∩ [[c,d]] = ∅
     # Sinon [[a,b]] ∩ [[c,d]] = [[max(a,c), min(b,d)]]

     blt $t1, $t2, empty                          move  $a2, $t3
     blt $t3, $t0, empty                          j termine
     ## calcul de max(a,c)                    min :
     bgt $t0, $t2, max                            move $a2, $t1
     move  $a1, $t2                           termine :
     j suite                                      jal print_intervalle
     max :                                        j fin
        move $a1, $t0                         empty  :
     suite :                                      la $a0, EMPTY_SET
     ## calcul de min(b,d)                        jal print_string
        blt $t1, $t3, min
```

```
### Sorti du programme ###
fin :

     li $v0, 10
     syscall

## Routines d'impression ##
# print_int et print_string impriment le contenu du registre $a0
# print_intervalle imprime ``[$a1, $a2]''
##
print_int :

     li $v0, 1
     syscall
     jr $ra

print_string :

     li $v0, 4
     syscall
     jr $ra

print_intervalle :

     la $a0, CROCHET_G
     jal print_string
     move $a0, $a1
     jal print_int
     la $a0, VIRGULE
     jal print_string
     move $a0, $a2
     jal print_int
     la $a0, CROCHET_D
     jal print_string
     jr $ra
```

Mnémoniques communs MIPS : Problème 3

Soient les registres $\$\mathtt{r}_{i\in\{0,1,2\}}$, le registre $\mathtt{\$CO}$ correspondant au compteur ordinal.

- `li $r0, n` effectue $\$\mathtt{r}_0 \leftarrow \mathtt{n}$.

- `lb $r0, n($r1)` effectue $\$r_0 \leftarrow RAM[\$r_1 + n]$.
- `sb $r0, n($r1)` effectue $RAM[\$r_1 + n] \leftarrow \$r_0$.
- `move $r0, $r1` effectue $\$r_0 \leftarrow \$r_1$.
- Opérations logiques et arithmétiques. Avec `Op` $\in \{$`and`, `or`, `xor`, `sll`, `srl`, `add`, `sub`, `mul`, `div`$\}$.

  `Op $r0, $r1, $r2` effectue $\$r_0 \leftarrow Op(\$r_1, \$r_2)$, et

  `Op $r0, $r1, n` effectue $\$r_0 \leftarrow Op(\$r_1, n)$.
- `j label` effectue \$CO $\leftarrow RAM[$`label`$]$
- Branchement conditionnel. Avec `Op` $\in \{$`beq`, `bne`, `blt`, `ble`, `bgt`, `bge`$\}$.
  - `beq $r0, $r1, label` effectue \$CO $\leftarrow RAM[$`label`$]$ si et seulement si $\$r_0 = \$r_1$, sinon, le programme se poursuit séquentiellement. (`bne` pour $\$r_0 \neq \$r_1$).
  - `blt $r0, $r1 label` effectue \$CO $\leftarrow RAM[$`label`$]$ si et seulement si $\$r_0 < \$r_1$, sinon, le programme se poursuit séquentiellement. (`ble` est utilisée pour la relation $\leq$).
  - `bgt $r0, $r1 label` effectue \$CO $\leftarrow RAM[$`label`$]$ si et seulement si $\$r_0 > \$r_1$, sinon, le programme se poursuit séquentiellement. (`bge` est utilisée pour la relation $\geq$).

  Ces opérations sont aussi valables pour la signature `Op $r0, n, label`. Dans ce cas, $\$r_1$ est substitué par `n` $\in \mathbb{Z}$. La sémantique de l'opération de branchement est conservée.

Table ASCII : Problème 4

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(nul)	0	0000	0x00	(sp)	32	0040	0x20	@	64	0100	0x40	‘	96	0140	0x60
(soh)	1	0001	0x01	!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
(stx)	2	0002	0x02	"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
(etx)	3	0003	0x03	#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
(eot)	4	0004	0x04	$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
(enq)	5	0005	0x05	%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
(ack)	6	0006	0x06	&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
(bel)	7	0007	0x07	’	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(bs)	8	0010	0x08	(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
(ht)	9	0011	0x09	)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
(nl)	10	0012	0x0a	*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
(vt)	11	0013	0x0b	+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
(np)	12	0014	0x0c	,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
(cr)	13	0015	0x0d	-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
(so)	14	0016	0x0e	.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
(si)	15	0017	0x0f	/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
(dle)	16	0020	0x10	0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
(dc1)	17	0021	0x11	1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
(dc2)	18	0022	0x12	2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
(dc3)	19	0023	0x13	3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
(dc4)	20	0024	0x14	4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
(nak)	21	0025	0x15	5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
(syn)	22	0026	0x16	6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
(etb)	23	0027	0x17	7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
(can)	24	0030	0x18	8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
(em)	25	0031	0x19	9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
(sub)	26	0032	0x1a	:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
(esc)	27	0033	0x1b	;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
(fs)	28	0034	0x1c	<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
(gs)	29	0035	0x1d	=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
(rs)	30	0036	0x1e	>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
(us)	31	0037	0x1f	?	63	0077	0x3f	_	95	0137	0x5f	(del)	127	0177	0x7f