Corewar

Cheat Sheet

Par son altesse royal
Ewen "Princess Luna" le Gouguec

(tcha tcha tcha tcham, tapis rouge, trompettes, champagne, toossa toossa ...)

Avant toutes choses

Voici la playlist tres aléatoire écoutée par l'auteur de ce document durant sa rédaction. L'ecouter, de preference dans le desordre pour eviter toute forme de cohérence, tout en lisant ce texte, vous aidera a vous mettre dans l'etat d'esprit dérangé de son createur et ainsi faciliter sa comprehension.

ABBA - Dancing Queen	Imagine Dragons - It's Time
ABBA - Fernando	Justice - Canon
ABBA - S.O.S.	Justice - Helix
ABBA - Waterloo	Marvin Berry and the Starlighters - Earth Angel
Ace of base - The Sign	Mary Wells - My Guy
Ace of base - Tokyo Girl	New Order - Blue Monday
Alestorm - 1741	Nik kershaw - The Riddle
Alestorm - Hangover	OMFG - Hello
AC/DC - Back in black	Richard Sanderson - Dreams are my reality
AC/DC - Thunderstruck	Roomie - Shmoyoho's Dayum Cover
Badfinger - Baby Blue	Shakira - Waka Waka
Eddie Carlson - E.Johnson's Cliffs of Dover Cover	Stevie Wonder - Superstition
Elton John - Don't Go Breaking My Heart	The Archies - Sugar Sugar
Imagine Dragons - Radioactive	The Chordettes - Mr. Sandman (1954)
Imagine Dragons - Every Nights	Totem - Bullshit
Imagine Dragons - On the Top of the World	Wham - Wake me up before you go go
Imagine Dragons - Deamon	

Types de parametres

REGISTRE : Codé sur 1 octet Identifiant d'un registre

Source : Charge le contenu du registre
Destination : Stock la valeur dans le registre

Important : Si une instruction est appellée avec un registre inexistant, l'instruction est invalide et le processus appelant crash.

INDEX : Codé sur 2 octets Addresse d'un entier en RAM

Source : Charge le contenu des 4 octets suivant l'index
Destination : Stock la valeur dans les 4 octets suivant l'index

DIRECT : Codé sur 4 octets Nombre entier
Codé sur 2 octets Addresse en RAM

Source : La valeur tel quel

Octet de codage des parametres

L'octet de codage des parametres, ou OCP, permet a la VM de savoir comment charger les parametres d'une instruction. Il est divisé en 4 paires de bits, trois determinant le type d'un parametre, et une quatrieme inutilisée. Elles sont reparties comme suit :

128	64	32	16	8	4	2	1
Parametre #3		Parametre #2		Parametre #1		Non utilisé	

Pour chaque parametre, le type est codé sur le modèle :

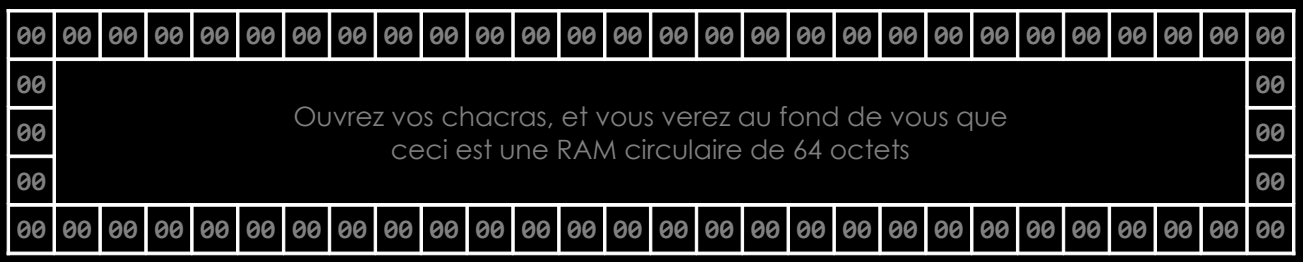
Bit superieur	Bit inferieur	Type
0	0	(Abscent)
0	1	Registre
1	0	Direct
1	1	Index

RAM et Adressage

La RAM de la VM est circulaire, et n'as aucun point zero ou autre repere. Des lors, l'adressage absolut est impossible. L'adressage est relatif a l'instruction courante, dont la position est elle meme relative au point de depart du programme...

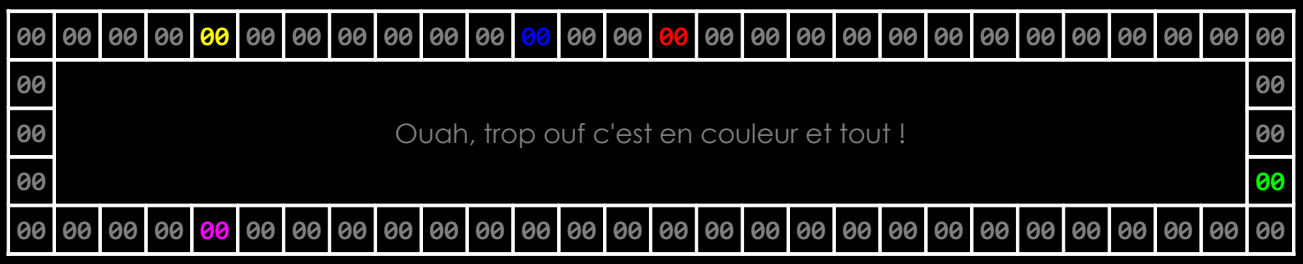
Bon, tout ca, c'est pas tres clair ..
Essayons plutot avec un petit exemple :

Admettons une RAM circulaire de 64 octets, initialement vide



Joli non ? Bon je sais, c'est un rectangle, mais bon, on fait avec ce qu'on a hein ...

Donc, comme il n'y a pas d'addresse absolue, on ne peut definir la position d'une case que par rapport a une autre case. Demonstration :



Choisissons en une. Tiens, celle en bleu, là.
Cette case devient alors l'adresse 0. Si on lit la RAM dans le sens horaire, la case rouge est la case 3, la verte est la case 19, la jaune est la -7 et la violette, la -19. Simple non ? Maintenant, un peu de mind fuck. La RAM etant circulaire, la case jaune, c'est aussi la case 57.
Et la bleu, bah c'est la 64, la 128, la 192, la 448, la -4608, la 17344, en plus bien sur d'etre la 0 (si si). On peut ainsi tourner a l'infini, dans le sens qu'on veut, tout ca n'a rien de bien compliqué.

Si maintenant on part de la rouge, on a donc : bleu = -3, jaune = -10, violette = -22 et verte = 16, such magic, so shybe.

Bon, maintenant que vous avez tout compris, voyons comment la VM implemante ce principe.

Le point de depart d'un programme est le point de reference d'adressage effectif.

Chaque processus a un process counter (pc), codé sur 2 octets, qui compte le nombre de case entre l'instruction courante, et le point de depart du programme. Il designe donc la postion de l'instruction courante par rapport au point d'adressage effectif.

Chaque adresse est relative a la case contenant l'opcode de l'instruction courante.

La valeure effective d'une adresse est egale a sa somme avec le process counter.

Consequence : le pc etant strictement positif ou nul, et codé sur 2 octets, le rayon d'action du processus en RAM est limité entre sont point d'origine, et sont point d'origine + 0xFFFF (ce qui permet d'acceder a toute la RAM des lors qu'elle ne depasse pas les 65 535 octets).

Lecteur perplexe > "Attend, si l'addressage est strictement positif ou nul, comment t'accede a une case precedant l'instruction courante ?"

Et bien grace a la magie de l'overflow ! (et ouais, qu'est c'qu'il y'a !?)

Addmetons une valeure numerique binaire x = 1111 1111 (255)
si j'incremente x, x = 1 0000 0000 (256)

Maintenant, si x est codé sur 8 bits (1 octet)
si x = 1111 1111 (255), et que j'incremente x
alors x = 0000 0000 (0), le neuvieme bit disparaissant.
si je dit maintenant x = x + 1 0000 0010 (258), alors x = 0000 0010 (2)

Amusant, n'est il pas ?

Maintenant, avec notre process counter. Par exemple :

si pc = 0x0000, pc + 0xFFFF = 0xFFFF.
si pc = 0x0001, pc + 0xFFFF = 0x0000, equivaut a - 0x0001
si pc = 0x0004, pc + 0xFFFE = 0x0002, equivaut a - 0x0002
si pc = 0xFFFF, pc + 0x0001 = 0x0000, equivaut a - 0xFFFF

On constate qu'enfait, l'addressage est cyclique.

Pour rendre cela plus clair, prenons ce petit shema, representant un segment de RAM, dont l'addressage serait relatif a un hypothetique pc codé sur 4 bits.

La case jaune represente le point de reference d'addressage courant
Chaque nombre represente l'addresse d'une case
Les cases en gris sont hors de portée de l'addressage

PC = 0

x	x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	x	x

PC = 1

x	x	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	x	x

PC = 2

x	x	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	x	x

PC = 7

x	x	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	x	x

PC = B

x	x	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	x	x

Fascinant, non ?

Lecteur perspicace > "Euh .. oui bon, tout ca c'est tres sexy, mais, a quoi ca sert tout ce merdier ? Pourquoi ne pas utiliser directement des addresses negatives ? C'est completement con !"

LE interet de ce systeme, est d'empecher un player de parcourir la RAM vers l'arriere, tout en ayant la possibilité de revenir sur ces pas, permetant par exemple de faire des boucles

(et puis ca aurait été trop facile sinon, hein ? faut pas déconner ...)

Pour finir, mettons tout ca en pratique, avec un petit programme.

Prennons le programme d'exemple du sujet, j'ai nommé : Zork

zork en assembleur	zork en hexadecimal
l2: sti r1, %:live, %1 and r1, %0, r1 live: live %1 zjmp %:live	0b 68 01 00 0f 00 01 06 64 01 00 00 00 00 01 01 00 00 00 01 09 ff fb

Chargons maintenant zork dans notre RAM :

00	00	0b	68	01	00	0f	00	01	06	64	01	00	00	00	00	01	01	00	00	00	01	09	ff	fb	00	00	00
00																											00
00																											00
00																											00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Bon, on l'a mis la, mais on aurait pu le foutre ailleur hein,
ca change rien, on s'en fout

La VM genere un processus ayant pour point de depart la case memoire contenant l'opcode de la premiere instruction. Des lors, cette case devient le point de reference d'addressage effective du processus. Le registre 1 du processus contient l'ID du player, qui sera dans notre exemple, 0.

00	00	0b	68	01	00	0f	00	01	06	64	01	00	00	00	00	01	01	00	00	00	01	09	ff	fb	00	00	00
00																											00
00																											00
00																											00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

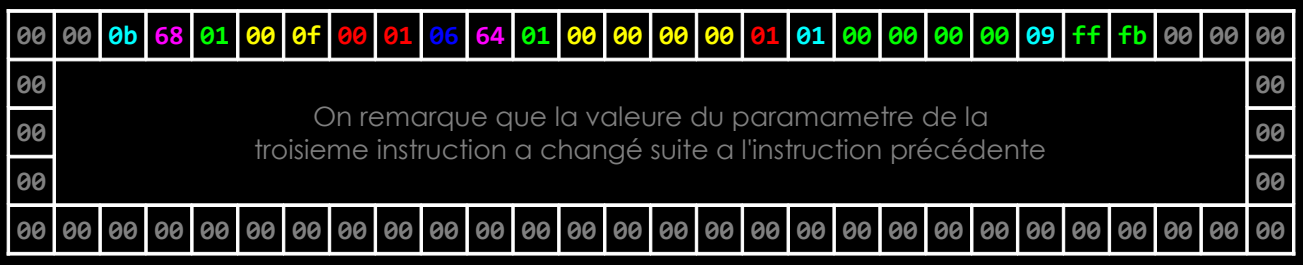
Legende : opcode d'instruction courante, opcode d'autres instructions
OCP, parametre #1, parametre #2, parametre #3

Nous avons donc au depart : PC = 0 Carry = 0

la VM decode la premiere instruction.
opcode 0x0b : Store indirect
OCP > p1 : REGISTRE, p2 : DIRECT, p3 : DIRECT

La VM additionne les 2 derniers paramametres, ce qui donne 0x0010 et stock a cette addresse, la valeure contenu dans le premier registre.

Le PC passe a 7, debut de l'instruction suivante

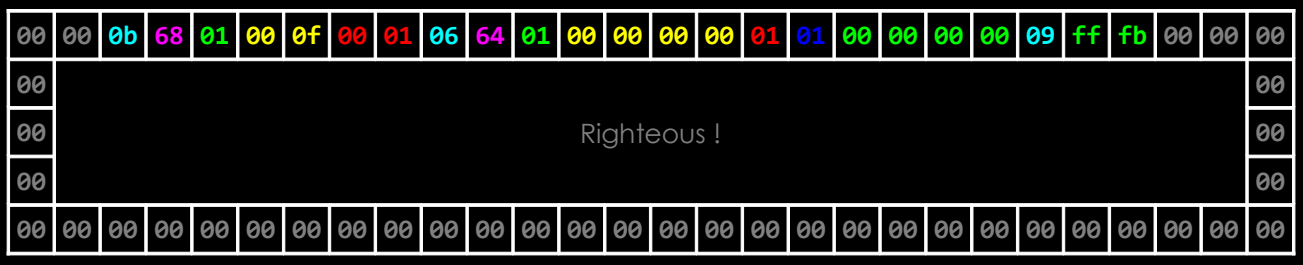


PC = 7 Carry = 0

opcode 0x06 : Logical AND
OCP > p1 : REGISTRE, p2 : DIRECT, p3 : REGISTRE

La VM effectue un AND logique entre les 2 premiers paramametres. Le second paramametre etant egale a zero, le resultat est toujours egale a zero, donc le carry passe a l'etat 1. Le resultat est stocké dans le premier registre

Le PC passe a 14, debut de la troisieme instruction

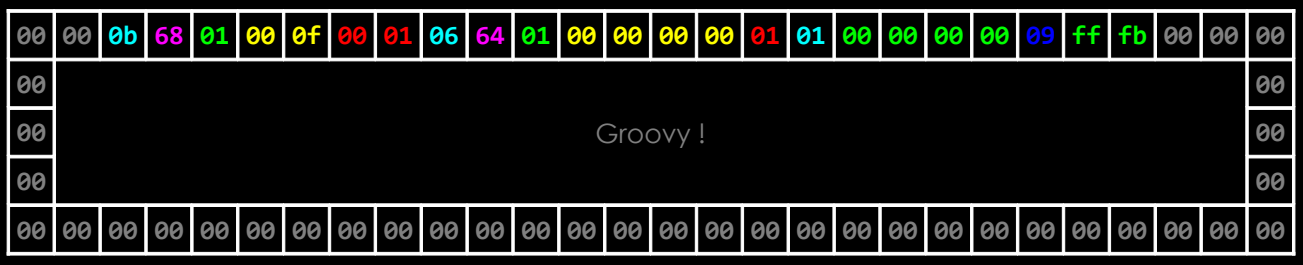


PC = 14 Carry = 1

ocpcode 0x01 : Live
Pas d'OCP, le seul parametre est toujours DIRECT (4 octets)

La VM reporte le player 0 comme etant en vie, et affiche un message en consequence dans le terminal.

Le PC passe a 19, debut de la quatrieme instruction



PC = 19 Carry = 1

opcode : 0x09
pas d'OCP, le seul paramametre est toujours DIRECT (2 octets)

La VM ajoute la valeure passée en parametre au PC.

Ici le parametre vaut 0xFFFB, et le PC vaut 19, soit 0x0013.
0x0013 + 0xFFFB = 0x1000E.

Or, comme le PC est codé sur 2 octets, sa valeure maximale est 0xFFFF.
Donc seul les 2 octets inferieurs du resultat subsisterons, soit 0x000E.

Le PC vaut donc 14, retour a la troisieme instruction.

A partir de la, le programme est dans une boucle infinie, se contentant de repéter les instruction 3 et 4.

Restriction de l'adressage

Certaines instructions (voir jeu d'instruction) limitent la porté de l'adressage autour du point d'adressage relatif en appliquant un modulo sur la valeure du decalge entre l'adresse de reference et l'adresse visé. Le valeure du modulo appliqué est definie par la constante IDX_MOD.

Dans ce cas, la VM procede comme suis :

- Calcul de l'adresse effective visée
  Adresse effective = PC + Adresse visée

- Calcul de la difference entre l'adresse de reference et l'adresse visée
  Adresse effective = Adresse effective - PC

- Calcul du modulo IDX_MOD de la difference
  Adresse effective = Adresse effective % IDX_MOD

- Calcul de l'adresse effective finale
  Adresse effective = PC + Adresse effective

Si on reprend notre hypothetique RAM sur 4 bits du chapitre precedent,

PC = 7, IDX_MOD = 4, Sans restriction de l'adressage

x	x	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	x	x

PC = 7, IDX_MOD = 4, Avec restriction de l'adressage

x	x	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	x	x

PC = 7, IDX_MOD = 4, Avec restriction de l'adressage

x	x	x	x	x	x	D 9	E A	F B	0 4 8 C	1 5	2 6	3 7	x	x	x	x	x	x	x

Jeu d'instruction

live	Live	0x01
Usage : `live S(D4)`		Durée : 10
OCP : Non	Adressage Restreint : Non	Modifie le carry : Non
Rapporte le joueur designé par le premier parametre comme etant en vie. L'instruction ecrit sur la sortie standard un message du type "Le joueur $player_name ($player_id), a été raporter comme étant en vie". Libre a vous de 'pimper' le message comme bon vous semble, du moment que l'idée passe et qu'il contienne les variables sus nommée. Un joueur ne vie que tant qu'au moins un processus effectue un live avec sont id, et ce au minimun une fois tout les CYCLE_TO_DIE. Si le parametre passé ne correspond a l'id d'aucun joueurs, le comportement est indefinit. A vous de decider si c'est une erreur et que le processus crash, ou si osef, l'instruction ne fait rien et on passe a la suite, avec eventuelement en supplément un petit message sur la sortie standard, message incohérent ou message d'avertissement, votre seul limite est celle de votre creativité.		

ld	Direct Load	0x02
Usage : `ld S(ID/D4), D(RG)`		Durée : 5
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Transfert direct RAM > Registre. Charge le premier parametre dans le registre passé en second parametre. Si la valeur du premier parametre est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

st	Direct Store	0x03
Usage : st S(RG), D(RG/ID)		Durée : 5
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Transfert direct Registre > RAM / Registre. Charge le contenu du registre passé en premier parametre dans le second parametre. Si la valeur du premier parametre est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

add	Aritmetical Addition	0x04
Usage : add S(RG), S(RG), D(RG)		Durée : 10
OCP : Oui	Adressage Restreint : Non	Modifie le carry : Oui
Ajoute le second parametre au premier parametre, et stock le resultat dans le troisieme parametre. Si la valeur resultante est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

sub	Aritmetical Substraction	0x05
Usage : sub S(RG), S(RG), D(RG)		Durée : 10
OCP : Oui	Adressage Restreint : Non	Modifie le carry : Oui
Soustrait le second parametre au premier parametre, et stock le resultat dans le troisieme parametre. Si la valeur resultante est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

and	Logical AND	0x06
Usage : `and S(RG/ID/D4), S(RG/ID/D4), D(RG)`		Durée : 6
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Effectue un AND logique entre les deux premiers paramametres et stock le resultat dans le troisieme paramametre. Si la valeur resultante est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

or	Logical OR	0x07
Usage : `or S(RG/ID/D4), S(RG/ID/D4), D(RG)`		Durée : 6
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Effectue un OR logique entre les deux premiers paramametres et stock le resultat dans le troisieme paramametre. Si la valeur resultante est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

xor	Logical XOR	0x08
Usage : `xor S(RG/ID/D4), S(RG/ID/D4), D(RG)`		Durée : 6
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Effectue un XOR logique entre les deux premiers paramametres et stock le resultat dans le troisieme paramametre. Si la valeur resultante est egale a zero, alors le carry passe a l'etat un, sinon a l'etat zero.		

zjmp	Jump if zero	0x09
Usage : `zjmp S(D2)`		Durée : 20
OCP : Non	Adressage Restreint : Non	Modifie le carry : Non
Saute a l'adresse passé en parametre si le carry est a l'etat un. L'adresse devient alors celle de la prochaine instruction. Si le carry est a l'etat zero, rien ne se passe et le flot continue normalement jusqu'a l'instruction suivante. Rien ne precise si l'instruction consomme la totalité de ces cycles dans ce cas, a vous d'en decider.		

ldi	Indirect Load	0x0A
Usage : `ldi S(RG/ID/D2), S(ID/D2), D(RG)`		Durée : 25
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Transfert indirect RAM > Registre. Charge la valeur a l'adresse resultante de l'addition des deux premiers paramametres, dans le registre passé en troisieme parametre. Si cette valeur est nulle, alors le carry passe a l'etat un, sinon a l'ettat zero.		

sti	Indirect Store	0x0B
Usage : `sti S(RG), S(RG/ID/D2), S(ID/D2)`		Durée : 25
OCP : Oui	Adressage Restreint : Oui	Modifie le carry : Oui
Transfert indirect Registre > RAM. Charge la valeur contenu dans le registre passé en premier parametre a l'adresse resultante de l'addition des deux derniers paramametres. Si cette valeur est nulle, alors le carry passe a l'etat un, sinon a l'ettat zero.		

fork	Fork	0x0C
Usage : fork S(D2)		Durée : 800
OCP : Non	Adressage Restreint : Oui	Modifie le carry : Non
Genere un nouveau processus a l'adresse passée en parametre par copie du processus appelant. Le nouveau processus garde donc l'etat de tout les registres et du carry, seul le PC differe (sauf dans le cas d'un fork %0).		

lld	Long Direct Load	0x0D
Usage : lld S(ID/D4), D(RG)		Durée : 10
OCP : Oui	Adressage Restreint : Non	Modifie le carry : Oui
Identique a Direct Load mais sans restriction de l'adressage.		

lldi	Long Indirect Load	0x0E
Usage : lldi S(RG/ID/D2), S(ID/D2), D(RG)		Durée : 50
OCP : Oui	Adressage Restreint : Non	Modifie le carry : Oui
Identique a Indirect Load mais sans restriction de l'adressage.		

lfork	Long Fork	0x0F
Usage : lfork S(D2)		Durée : 1000
OCP : Non	Adressage Restreint : Non	Modifie le carry : Non
Identique a Fork mais sans restriction de l'adressage.		

aff	Aff	0x10
Usage : aff S(RG)		Durée : 2
OCP : Oui	Adressage Restreint : Non	Modifie le carry : Oui

Affiche a l'ecran le char correspondant a la valeure du registre passé en parametre, modulo 256. Si ce char est NUL, alors le carry passe a l'etat 1, sinon a l'état 0. A vous de choisir le formattage de la sortie ecran. Vous pouvez par exemple preciser a chaque aff l'id de sont processus d'origine, ou bien attribuer une couleur a chaque processus, ou encore attribuer une ligne de aff par processus, ou n'importe quoi d'autre avec les processus, l'important reste que votre sortie de aff soit le plus swag possible.

Idée de bonus : L'instruction aff a pour seul but de taunter son adversaire (ce qui est une part non negligeable du jeu). Mais, en envoyant les chars au compte goute, il est tres difficile de sortir une phrase propre a l'ecran, sans etre interompu par un live ou un autre aff. Pour palier a ca, vous pouvez bufferiser le aff. Dans ce cas chaque processus possede son buffer. A chaque appel de aff, l'instruction rajoute le char dans le buffer du processus. Lors d'un appel de aff avec le char NUL, l'instruction vide le buffer a l'ecran. Un char NUL definissant egalement le carry a 1, cette construction permet de facilement faire des boucle pour display des strings, si tant est qu'elles se terminent par un NUL char.