Manuel de référence du LAVAL COMPUTER

Introduction

Le LAVAL computer est le premier ordinateur à implémenter la révolutionaire microarchitecture LAVAL. Comme tout bon accronyme, il est défini récursivement: Laval Advanced Vectorized Architecture Laboratory.

C'est une architecture inédite, prête à révolutionner le monde de l'informatique par son organisation hautement parallèle.

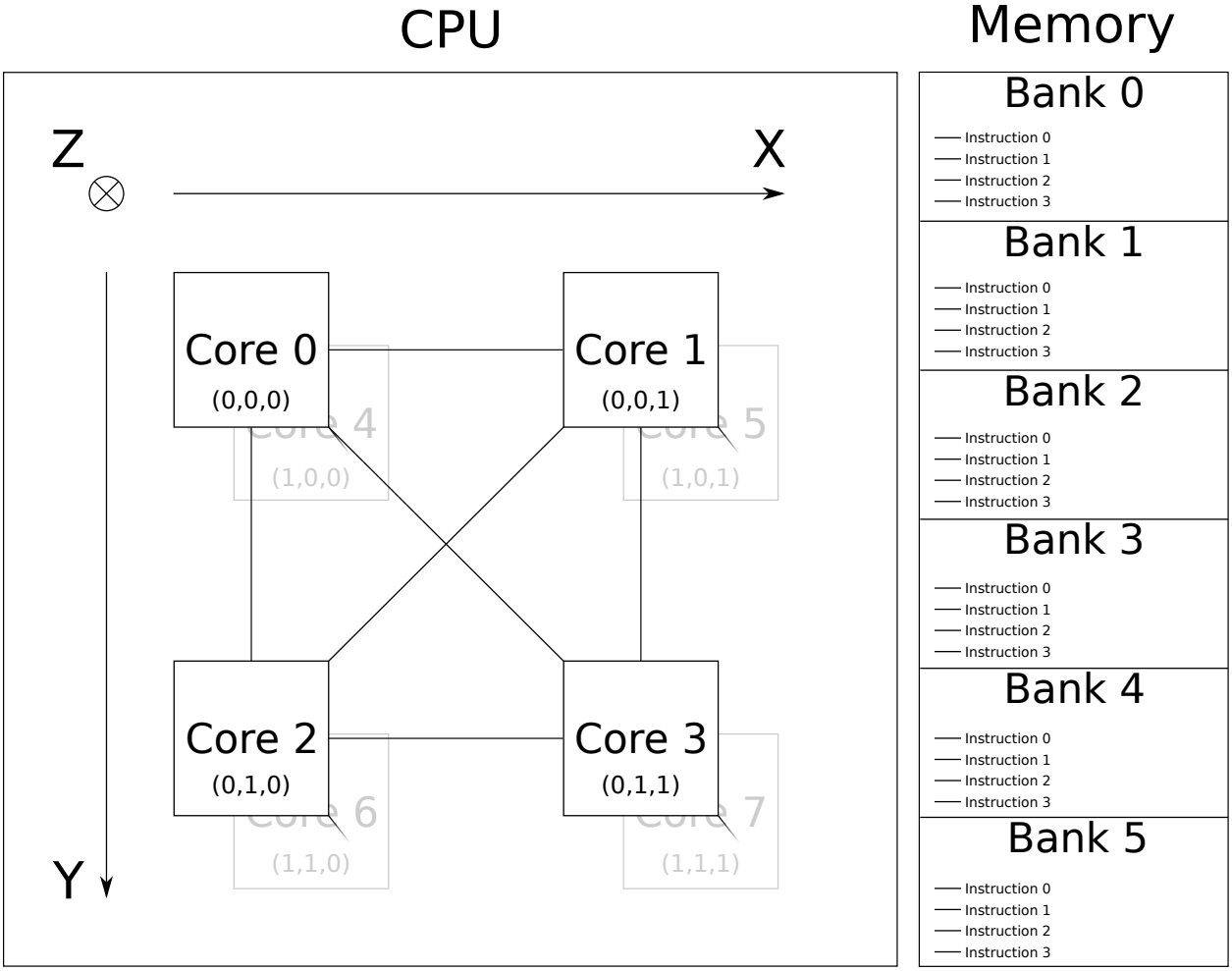
Architecture

L'architecture LAVAL se situe quelque part entre celle du CPU, GPU et du FPGA. Elle consiste en un grand nombre de coeurs simples localement connectés suivant un modèle de cube.

Les coeurs exécute des programmes stoqués dans la banque de mémoire auquel ils sont connectés. Cette mémoire ne peut être modifiée, il s'agit donc d'une architecture Harvard. Les branchements sont faits en changeant l'exécution d'une banque de mémoire à une autre.

Plusieurs paramètres du CPU, comme le nombre de coeurs, sont définis par l'implémentation. Référez-vous au manuel spécifique de votre modèle de CPU pour plus d'informations.

Par exemple, cette figure présente une petite implémentation 2x2x2 avec six banques de mémoire de quatre octets chaque.



Veuillez noter que seul certains liens coeur-à-coeur sont dessiné par soucis de clareté.

Survol des coeurs

La composante principale de l'architecture LAVAL est l'unité CORE. C'est un coeur de faible complexité capable d'exécuter des instructions de 8 bits, *arguments inclus*. Toutes les instructions s'exécutes en un cycle.

Les coeurs possèdent un seul registre d'usage générale et ne peuvent adresser de la RAM, plutôt, la mémoire collective de tous les coeurs représente l'état du programme. En conséquent, dès qu'un algorithme nécessite plus d'une variable de 8 bits, plusieurs coeurs sont nécessaires.

Une conséquence de ces caractéristiqus est qu'aucune instruction ne peut travailler avec des constantes plus grandes que 4 bits (0xf). Pour palier à cet inconvénient, il est souvent possible d'arriver au résultat souhaité en utilisant deux instructions ou en programmant un coeur pour produire la constante désirée.

Référez-vous à la section sur le jeu d'instruction pour plus d'information.

Chaque coeur possède un multiplexeur lui permettant de cibler l'un de ses 26 voisins. La communication inter-coeurs est largement optimisé pour les patrons d'accès constant, puisqu'une instruction (et donc un cycle) est nécessaire pour aiguiller le multiplexeur. Les algorithmes bâtis pour que les coeurs nécessitent qu'un petit nombre d'entrées sont ainsi largement plus efficace.

Registres

Cette section décrit les registres présents sur chaque coeur. La plupart des opérations sur ces registres sont abstraites par le jeu d'instructions, mais comprendre leur fonction peut aider à saisir le fonctionnement interne du CPU.

Tous les registres font 8 bits.

VAL

VAL est le seul registre d'usage général présent sur une unité CORE. La plupars des instructions lisent ou modifient VAL.

Les nombres négatifs sont encodés avec la notation complément 2.

MUX

MUX est le registre du multiplexeur de coeur.

Sa valeur détermine quel coeur un coeur peut accéder dans un cycle.

Son contenu est encodé comme un nombre en base 3 dont le trit le plus significatif adresse la dimension Z et le moins significatif, la dimension X.

Un trit encode un position relative au coeur courant de cette façon:

- 0: Avant (BEFORE)
- 1: Courant (CURRENT)
- 2: Après (AFTER)

Par exemple, la direction `BEFORE, CURRENT, AFTER` est encodée comme `(0 * 3^2) + (1 * 3^1) + (2 * 3^0)`.

Référez-vous à la section sur les Entrées/Sorties pour plus d'information au sujet de l'utilisation du multiplexeur.

La valeur initiale de ce registre est `CURRENT, CURRENT, CURRENT`, qui n'est pas une direction légale d'acquisition.

PC

PC est le «program counter» (compteur ordinal).

Sa valeur détermine quelle instruction de la banque de mémoire courante va être exécuté en prochain. PC est incrémenté après chaque instruction n'ayant pas bloquée la chaîne de traitement (pipeline) du coeur. Le pipeline est bloqué quand une instruction tentant d'atteindre une valeur par le multiplexeur échoue à ce faire. Cela se produit quand le coeur pointé n'a pas exécuté une instruction de synchronisation (SYN).

Comme tout registre incrémentant, sa valeur peut déborder. Cela signifie que sa valeur est remise à zéro plutôt que d'atteindre 0x100 (256).

Ce registre est initialisée à zéro au démarrage.

MEMBANK

MEMBANK est le registre du pointeur de mémoire.

Son contenu défini de quelle banque de mémoire la prochaine instruction sera récupérée. Son contenu est intialement configuré en fonction du paramètre `core_to_mem`. Sa valeur peut aussi être changé en cours d'exécution en utilisant l'une des instruction de branchement.

Status

Chaque coeur inclus aussi un ou plusieurs registre de status À ce jour, ils ne sont pas exposés à l'utilisateur et leur configuration exacte est définie par l'implémentation. Une future version du jeu d'instructions pourrait standardiser leur configuration.

Communication inter-coeurs

Chaque coeur possède un multiplexeur pouvant être pointé sur l'un de ses 26 voisins. Un coeur peut par la suite utiliser son multiplixeur pour **charger** une value du coeur cible.

Attention: Vous ne pouvez pas connecter un coeur à lui même.

Ces transfers sont synchronisés, l'instruction de *chargement* ne n'aboutira pas tant et aucun registre ne sera modifié tant que le coeur cible n'émetera pas une instruction de synchronization. La même chose est vrai pour les instructions de synchronisation; un coeur va bloquer sur une telle instruction tant qu'un moins un coeur n'a pas récupéré une valeur de lui.

Un coeur exécutant une instruction de synchronisation rend ses registres disponibles à tous les coeurs connectés. Cela signifie que plusieurs coeurs peuvent charger une valeur d'un même coeur et une seule instruction de synchronisation est alors requise.

Une fois un qu'une donnée a été acquise d'un coeur, la drapeau de synchronisation de ce dernier est remis à 0 et une autre instruction de synchronisation sera requise pour la prochaine acquisition.

Finalement, un coeur ne peut acquérir d'un coeur iexistant, sauf si ce premier est configuré comme entrée du CPU (plus de détails plus tard).

Exemple en pseudo-code:

```
Coeur0:
    VAL <- 5
    Synchroniser  ; Procède immédiatement puisque qu'au moins un coeur fait une acquisition.

Coeur1:
    Connecter au coeur 0
    Charger du coeur connecté
    ; Charge immédiatement puisque le coeur cible exécute une instruction de synchronisation
    ; VAL est maintenant 5

Coeur2:
    Connecter au coeur 0
    Charger du coeur connecté
    ; Charge immédiatement puisque le coeur cible exécute une instruction de synchronisation
    ; VAL est maintenant 5
```

```
Core0:
    VAL <- 5
    Rien faire
    Synchroniser  ; Procède immédiatement puisque qu'au moins un coeur fait une acquisition.

Core1:
    Connecter au coeur 0
    Charger du coeur connecté
    ; Attend pour un cycle puisque le coeur cible n'a pas encore synchronisé, puis charge.
    ; VAL est maintenant 5

Core2:
```

```
    Connecter au coeur 0
    Charger du coeur connecté
    ; Attend pour un cycle puisque le coeur cible n'a pas encore synchronisé, puis charge.
    ; VAL est maintenant 5

Core0:
    VAL <- 5
    Synchroniser  ; Procède immédiatement puisque qu'au moins un coeur fait une acquisition.

Core1:
    Connecter au coeur 0
    Charger du coeur connecté
    ; Charge immédiatement puisque le coeur cible exécute une instruction de synchronisation
    ; VAL est maintenant 5

Core2:
    Connecter au coeur 0
    Rien faire
    Charger du coeur connecté
    ; Le coeur cible à remis à zéro son drapeau de synchronisation. Attend indéfiniement.

Core0:
    VAL <- 5
    Synchroniser  ; Wait for one cycle since no one is fetching from this core in this cycle.

Core1:
    Connecter au coeur 0
    Rien faire
    Charger du coeur connecté
    ; Charge immédiatement puisque le coeur cible exécute une instruction de synchronisation
    ; VAL est maintenant 5

Core2:
    Connecter au coeur 0
    Rien faire
    Charger du coeur connecté
    ; Charge immédiatement puisque le coeur cible exécute une instruction de synchronisation
    ; VAL est maintenant 5
```

Entrées/Sorties

Le CPU, tel que décrit jusqu'à date, ne peut travailler qu'avec des constantes prédéfinies; ce qui est plutôt inutile en soit. Pour être utile, un CPU doit interagir avec le monde réel.

Entrées

Un coeur peut être connecté à une entrées du CPU. Cela permet à ce coeur de recevoir des données de l'extérieur.

Référez-vous à la section sur l'assembleur pour plus d'information sur comment configurer les entrées.

À l'exception de la configuration, recevoir des données de l'extérieur ou d'un autre coeur fonctionne de la même façon.

Pour utiliser son entrée, un coeur doit pointer son multiplexeur vers une direction normalement invalide, mais autre que lui-même. Cela signifie que, par design, seulement un coeur situé en bordure du «cube» du

CPU peut avoir une entrée.

Il est interdit de brancher deux entrées ou à la fois une entrées et une sortie sur un coeur.

Outputs

Un coeur peut être connecté à une sortie du CPU. Cela permet à ce coeur de transmettre des données à l'extérieur.

Référez-vous à la section sur l'assembleur pour plus d'information sur comment configurer les sorties.

À l'exception de la configuration, envoyer des données à l'extérieur ou à un autre coeur fonctionne de la même façon.

Pour utiliser sa sortie, un coeur doit simplement synchroniser (SYN) son actuel VAL. La sortie du CPU va alors automatiquement acquérir la valeur et débloquer le coeur. Le résultat est le même que si un autre coeur aurait procédé à un chargement dans le même cycle que la synchronisation. The result is the same as if another core loaded from it in the same clock cycle.

Par design, seulement un coeur situé en bordure du «cube» du CPU peut avoir une sortie.

Il est interdit de brancher deux entrées ou à la fois une entrées et une sortie sur un coeur.

Assembleur

Cette section décrit comment utiliser l'assembleur pour écrire un programme ciblant le jeu d'instrction LAVAL.

L'assembleur ne prend pas compte des espaces tant que les espaces requis sont présents (par exemple, entre une instruction et ses arguments) et que les jetons sont entiers (aucun espace n'est permi au millieu d'une mnémonique d'instruction).

Configuration

Les programmes en assembleur commencent par une section de configuration. La configuration indique les requis d'un CPU exécutant ce programme. Les paramètres sont écris, une par ligne, suivant le format suivant.

`\.(\w+) ([\d, ]*)`

- Le premier groupe de capture est le nom du paramètre
- Le second groupe de capture contient les arguments du paramètre.

Aucun paramètre ne doit être présent après la première déclaration de banque de mémoire.

.cores

Dimensions du CPU

Argument	Longueur	Description
0	0..65535	Nombre de coeur sur la dimension Z
1	0..65535	Nombre de coeur sur la dimension Y
2	0..65535	Nombre de coeur sur la dimension X

.mem_number

Nombre de banque de mémoire

Argument	Longueur	Description
0	0..255	Nombre de banque de mémoire

.mem_size

Taille des banques de mémoire

Argument	Longueur	Description
0	0..255	Taille en octet de chaque banque de mémoire

.core_to_mem

Assigne les coeurs à leur banque de mémoire initiale

Argument	Longueur	Description
0	0..255	Banque de mémoire associé au coeur 0 au démarrage
1	0..255	Banque de mémoire associé au coeur 1 au démarrage
n	0..255	Banque de mémoire associé au coeur n au démarrage

`n` doit correspondre au nombre de coeurs.

.in

Assignation des entrées

Argument	Longueur	Description
0	0..255	Coeur sur lequel est connecté l'entrée #0
1	0..255	Coeur sur lequel est connecté l'entrée #1
n	0..255	Coeur sur lequel est connecté l'entrée #n

`n` est le nombre d'entrées du programme et est limité à 65535.

.out

Assignation des sorties

Argument	Longueur	Description
0	0..255	Coeur sur lequel est connecté l'entrée #0
1	0..255	Coeur sur lequel est connecté l'entrée #1
n	0..255	Coeur sur lequel est connecté l'entrée #n

`n` est le nombre de sorties du programme et est limité à 65535.

Memory bank

Les banques de mémoire sont programmées en déclarant un bloc d'instructions démarrant par le jeton suivant:

`(\d+):`

- Le groupe de capture indique quelle banque de mémoire va contenir les instructions qui suivent.

Prenez note que les identifiants de banque de mémoire sont indexés 0.

Instructions

Les instructions suivent le format suivant:

`(\w{3})( -?\d+(?:, ?\d+)*)?`

- Le premier groupe de capture est la mnémonique
- Le second groupe de capture (optionnel) contient les arguments de l'instructions.

Prenez note que le nombre et la longueur spécifique des arguments dépendent de l'instruction. Référez-vous à leur documentation pour plus d'information.

Les instructions doivent obligatoirement faire parties d'une banque de mémoire.

Commentaires

Un commentaire peut être déclaré en utilisant le caractère `;`. Le reste de la ligne sera alors ignoré par l'assembleur.

Préprocesseur

L'assembleur mandate l'utilisation d'un preprocesseur. Celui-ci est utilisé pour fournir quelques constantes au programmeur:

Constante	Valeur
BEFORE	0
CURRENT	1
AFTER	2

Les constantes sont simplement textuellement remplacées avant que le fichier soit passé à l'assembleur.

Exemples

```
; Déplace à répétition une valeur d'une unique entrée vers une unique sortie. Optimisé pour la vitesse.

.cores 1, 1, 1
.mem_number 2
.mem_size 2
.core_to_mem 0
.in 0
.out 0

0:
```

```
    MUX CURRENT, BEFORE, CURRENT
    JMP 1

1:
    SYN
    JMP 1
```

Ce programme est optimisé pour la vitesse. Il prend exactement deux cycles pour déplacer une valeur de l'entrée vers la sortie. Dans un système plutôt contraint en ressources, il pourrait être approprié de sacrifier un peu de vitesse pour réduire le coût. Le programme démontré plus haut peut être modifié pour atteindre cet objectif de cette façon:

```
; Déplace à répétition une valeur d'une unique entrée vers une unique sortie. Optimisé pour le coût.

.cores 1, 1, 1
.mem_number 1
.mem_size 3
.core_to_mem 0
.in 0
.out 0

0:
    MUX CURRENT, BEFORE, CURRENT
    SYN
    JMP 0
```

Dans cette version, le programme est plus petit (1 banque x 3 octets vs 2 banques x 2 octets) mais prend 3 cycles d'horloge pour bouger une valeur. Les programmes peuvent souvent être optmisé dans un sens ou dans l'autre en utilisant des techniques semblables.

Jeu d'instruction

Instructions de base

NOP

No operation

Argument: Aucun

Description:

Ne fait rien pour un cycle

SYN

Sync

Argument: Aucun

Description:

Synchronise VAL avec le(s) multiplexeur(s) connecté(s). Plusieurs coeurs peuvent alors recevoir la même valeur tant qu'ils en font la demande dans le même cycle. Cette instruction va bloquer tant qu'au moins un coeur n'a pas récupéré la valeur.

DBG

Output to debugger

Argument: Aucun

Description:

Envoie le status du coeur (la valuer de tous les registres) au débuggeur connecté.

HLT

Halt

Argument: Aucun

Description:

Arrête l'exécution du CPU et retourne VAL. Deux coeurs retournant une valeur simultanément produit une valeur indéfinie.

HCF

Argument: Aucun

Description:

???

Controlling mux

MUX

Set multiplexer to another core

Arguments:

Taille	Description
0..2	Offset on dimension 0
0..2	Offset on dimension 1
0..2	Offset on dimension 2

Description:

Pointe le multiplexeur sur un autre coeur en assignant au registre MUX tel qu'indiqué par les arguments de l'instruction.

Notes:

Un coeur ne peut être connecté à lui-même.

Example:

```
MUX CURRENT, BEFORE, AFTER
```

CTC

Connect to carry

Argument: Aucun

Description:

Connecte le multiplexeur du coeur au bit de retenue de sa cible.

Notes:

Le multiplexeur peut aussi être connecté au registre VAL de sa cible en utilisant l'instruction CTV.

CTV

Connect to VAL

Argument: Aucun

Description:

Connecte le multiplexeur du coeur au registre VAL de sa cible. Il s'agit de l'état par défaut.

Notes:

Le multiplexeur peut aussi être connecté au bit de retenue de sa cible en utilisant l'instruction CTC.

Lire du multiplexeur

MXD

Multiplexer discard

Argument: Aucun

Description:

Récupère et ignore une valeur du multiplexeur. Utilisez cette instruction à des fins de synchronisation pour débloquer un coeur en attente sur une instruction SYN.

Notes:

Cette instruction n'impacte pas le registre VAL.

Example:

```
.cores 1, 1, 2
.mem_number 2
.mem_size 3
.core_to_mem 0, 1

0:
    LCL 1
    SYN     ; Va bloquer sur cette instruction pour un cycle puis s'exécuter sur le suivant
    LCL 2   ; Cette instruction va donc s'exécuter sur le quatrième cycle

1:
    NOP
    NOP
    MXD     ; VAL est toujours de zéro
```

MXL

Multiplexer load

Argument: Aucun

Description:

Récupère et assigne à VAL une valeur du multiplexeur.

MXA

Multiplexer addition

Argument: Aucun

Description:

Récupère et additionne à VAL une valeur du multiplexeur.

MXS

Multiplexer subtraction

Argument: Aucun

Description:

Récupère et soustrait à VAL une valeur du multiplexeur.

Jumping

JMP

Jump unconditionally

Argument:

Taille	Description
0..15	Membank id

Description:

Pointe le coeur courant sur une autre banque de mémoire tel qu'indiqué par l'argument. PC est réinitialisé à 0.

Example:

```
.cores 1, 1, 1
.mem_number 2
.mem_size 2
.core_to_mem 0, 1

0:
    NOP     ; Premier cycle.
    JMP 1   ; Second cycle.

1:
    NOP     ; Troisième cycle.
```

JLZ

Jump if less than zero

Argument:

Taille	Description
0..15	Membank id

Description:

Pointe le coeur courant sur une autre banque de mémoire tel qu'indiqué par l'argument si VAL est plus petit que 0. PC est réinitialisé à 0.

JEZ

Jump if equal to zero

Argument:

Taille	Description
0..15	Membank id

Description:

Pointe le coeur courant sur une autre banque de mémoire tel qu'indiqué par l'argument si VAL est égal à 0. PC est réinitialisé à 0.

JGZ

Jump if greater than zero

Argument:

Taille	Description
0..15	Membank id

Description:

Pointe le coeur courant sur une autre banque de mémoire tel qu'indiqué par l'argument si VAL est plus grand que 0. PC est réinitialisé à 0.

Utilisation de constantes

LCL

Load constant into low part

Argument: Aucun

Taille	Description
0..15	Valeur à charger

Description:

Charge une constante de 4 bits dans les 4 bits les moins significatifs de VAL. Les 4 autres bits de VAL ne sont pas affectés.

Notes:

Pour charger dans les autres bits de VAL, utilisez LCH.

LCH

Load constant into high part

Argument: Aucun

Taille	Description
0..15	Valeur à charger

Description:

Charge une constante de 4 bits dans les 4 bits les plus significatifs de VAL. Les 4 autres bits de VAL ne sont pas affectés.

Notes:

Pour charger dans les autres bits de VAL, utilisez LCL.

LSL

Logical shift, left

Argument: Aucun

Taille	Description
0..15	Nombre de position

Description:

Effectue un décalage logique vers la gauche de VAL d'un nombre de position donné par l'argument. Les bits se retrouvant vacants sont remplacés par des zéros. Ne préserve pas le bit de signe, si applicable.

LSR

Logical shift, right

Argument: Aucun

Taille	Description
0..15	Nombre de position

Description:

Effectue un décalage logique vers la droite de VAL d'un nombre de position donné par l'argument. Les bits se retrouvant vacants sont remplacés par des zéros. Ne préserve pas le bit de signe, si applicable.

CAD

Constant addition

Argument: Aucun

Taille	Description
0..15	Valeur à additionner

Description:

Additionne une constante à VAL.

CSU

Constant subtraction

Argument: Aucun

Taille	Description
0..15	Valeur à soustraire

Description:

Soustrait une constante à VAL.

CAN

Constant AND

Argument: Aucun

Taille	Description
0..15	Masque

Description:

Applique un masque logique ET aux quatres bits les moins significatifs de VAL. Les quatre bits les plus significatifs sont remis à zéros.

COR

Constant OR

Argument: Aucun

Taille	Description
0..15	Masque

Description:

Applique un masque logique OU aux quatres bits les moins significatifs de VAL. Les quatre bits les plus

significatifs ne sont pas affectés.

L'instruction COR ne peut à elle seul affecter tous les bits de VAL. It pourrait alors être utile de la combiner avec l'instruction CAN pour restreindre VAL aux valeurs affectées.

Simulateur

Pour des détails sur l'utilisation du simulateur, référez-vous à son aide incluse: `simulator --help`. For simulator usage, refer to its included help: `simulator --help`.