ARCSYS : ce qu'il faut retenir

C. Ferry

December 17, 2015

Contents

1 Logique booléenne **1**

2 Logique CMOS **2**

3 Circuits combinatoires **2**

4 Circuits synchrones **3**
4.1 Du point de vue des composants 3
4.2 Du point de vue de la représentation 4

5 Calculateur à logique câblée : Décomposition traitement/commande **4**

6 Composants mémoire **4**
6.1 SRAM asynchrone 5
6.2 SRAM synchrone 5
6.3 DRAM 5

7 Processeur à jeu d'instructions **5**

8 Le cas du NIOS **6**
8.1 Instructions 6
8.2 Directives d'assembleur 7
8.3 Stack 7

9 Entrées et sorties **8**
9.1 Interruptions matérielles 8

10 Cache **9**

11 Pipeline **10**

12 Annexe : complément à deux et extension de signe **10**

13 Annexe : représentation des objets en mémoire **11**

1 Logique booléenne

- Monôme sous forme canonique : mettez votre formule sous forme

$$\bigwedge_{i=1}^{n} x_i$$

avec les x_i éventuellement complémentés.

- Formule sous (1ère) forme canonique : forme normale disjonctive (OU de monômes). Par exemple, dressez la table de vérités et faites la somme (avec l'opération OU) des lignes où la formule est satisfaite.

- Simplification de formules : méthode de Karnaugh, non détaillée en cours.
  Pour réaliser cette méthode :

$\downarrow c, d \mid a, b \mid \rightarrow$	00	01	11	10	ne changer qu'un bit par colonne
00	1	0	0	0	
01	0	0	0	1	
11	1	1	1	0	
10	0	1	1	1	
ne changer qu'un bit par ligne					

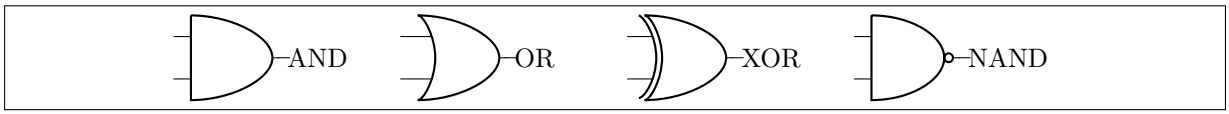
  Former dans ce tableau les plus grands rectangles possibles : la construction des lignes et des colonnes (code de Gray[1]) donne une formule facile à faire, ici pour le carré du bas : $b \wedge c$ (car a, d parcourent $\{0, 1\}$ et la formule reste vraie si $b = c = 1$).

2 Logique CMOS

- Transistor MOS : "interrupteur" contrôlé électroniquement. Il a trois pattes (pour votre culture : *émetteur-base-collecteur*).

- Temps de commutation $T_{com} > 0$! (ne pas considérer que la commutation est instantanée). De manière plus générale, il y a un temps de commutation dans le circuit.

- Avec des transistors, on fabrique des portes logiques... Retenir leurs symboles !



- Avec des NAND, vous pouvez toutes les fabriquer...

- Porte tristate : forcée à 0, forcée à 1, laissée libre. Typiquement, c'est ça qu'on trouve dans les bus partagés.

3 Circuits combinatoires

- Circuits faisant des calculs booléens, fabriqués avec des composants de logique combinatoire (les portes ci-dessus). Pas d'horloge, la sortie dépend uniquement de l'entrée (pas de l'état du système). **PAS DE BOUCLES !!**

- Le temps de propagation dans un tel circuit : estimation pessimiste, trouver le plus long chemin.

- Composants combinatoires usuels : multiplexeur (2^p entrées, sélection d'entrée sur p bits), décodeur (entrée sur p bits, 2^p sorties, la sortie sélectionnée est $\sum_{i=0}^{p-1} x_i 2^i$ + switch direct/inverse), additionneur 1 bit, additionneur n bits pour tout n.

- Méthodes de réalisation :

  - méthode booléenne (utiliser les formes canoniques vues ci-dessus, mettre des AND éventuellement précédés de NOT - modélisé par $\circ$ - puis un gros OR). **Peu efficace** mais simple à mettre en œuvre.
  - exploitation de la composition de fonctions : un bloc pour chaque fonction, dont la sortie va à l'entrée d'un autre. Sélectionner un résultat avec des multiplexeurs. **Plus efficace.**

[1] 70100

  - récurrence, dichotomie : résoudre les sous-problèmes, en faire l'entrée d'un problème plus général. Attention, ces fonctions récursives sont bornées (on ne met pas une infinité de composants sur une carte).

- Ne pas hésiter à simplifier un peu, c'est plus joli. Par exemple, supprimer les OU dont une entrée est à V_{cc}, supprimer ce qui suit un ET dont une entrée est à GND.

4 Circuits synchrones

4.1 Du point de vue des composants

- Le circuit a un état interne, on lui donne une horloge. Il met à jour son état interne à chaque cycle d'horloge en fonction d'une entrée et de son état. Plus formellement:

$$\begin{aligned} e &:= f_t(e, v) \\ s &= f_s(e, v) \end{aligned}$$

  où e est l'état de la machine, v la donnée d'entrée. f_t, f_s sont combinatoires, l'horloge est sur la mémoire qui contient e.

- f_t = fonction de transition, f_s = fonction de sortie.

- Machines:

  - De Mealy : sortie fonction de l'état et de l'entrée.
  - De Moore[2] : sorties uniquement fonction de l'état.

- On doit respecter le temps de propagation de l'information dans le circuit :

  - pour le changement d'état, T_{cy} le temps de cycle minimal comprenant T_{su} (set-up time, temps de calcul de l'état futur à partir de l'entrée), T_h le temps de maintien.
  - pour les sorties : T_{vs} temps de calcul des sorties à partir des entrées, T_{es} temps de calcul des sorties à partir de l'état.
  - Ces notations temporelles bizarres ne sont certainement pas à savoir par cœur, mais il faut savoir qu'il y a un temps minimal de calcul, incompressible, au-delà duquel l'horloge ne peut aller.

- Des circuits séquentiels :

  - les bascules
    * D : mémoire 1 bit, $Q := D$
    * T : inverseuse d'état, $Q := (\neg T \wedge Q) \vee (T \wedge \neg Q)$
    * JK : suivant J, K, garder Q, inverser Q, forcer 0, forcer 1, $Q := (J \wedge \neg Q) \vee (\neg K \wedge Q)$
    * RS : suivant R, S, garder Q, forcer 0, forcer 1), $Q := S \vee (\neg R \wedge Q)$
  - les compteurs : permanent, compteur/décompteur (avec une entrée INC, une entrée DEC, si aucune n'est mise, le compteur ne bouge pas), compteur avec chargement (et entrée CH).
  - les registres : à chargement systématique (n bascules D en parallèle), à chargement commandé, à décalage systématique, à décalage droite-gauche...

- Circuit synchrone : interconnexion de circuits synchrones ou combinatoires, mais **UNE UNIQUE HORLOGE !!!** On peut avoir des boucles sur les composants séquentiels.

- **L'HORLOGE NE DOIT JAMAIS ÊTRE PRISE COMME ENTRÉE DANS UN CIRCUIT !!!!!**

[2] pas de vanne !

4.2 Du point de vue de la représentation

- Les états de nos machines sont finis (entrées finies, mémoires finies à états finis) : on peut faire des automates finis pour les représenter.
- On peut fusionner les états équivalents pour simplifier l'automate.

5 Calculateur à logique câblée : Décomposition traitement/commande

- On décompose le circuit en une unité de traitement (qui prend l'entrée extérieure, fait le calcul proprement dit) et une unité de commande (qui modélise l'automate du chapitre précédent et génère les commandes pour lancer les calculs).
- L'UT et l'UC communiquent par connexions simples (une seule source, plusieurs destinataires) ou par bus (plusieurs sources, plusieurs destinataires). Les bus sont fabriqués avec des portes tristate, **qui permettent de laisser libre le bus pour que d'autres composants l'utilisent**.
- UC est en général un système synchrone standard, dotée d'un compteur, d'un module de génération des commandes (sorties de l'UC) et d'un module de séquencement (qui prend des entrées extérieures, pouvant aussi venir de l'UT au titre de feedback -par exemple le signe d'une addition dans un processeur NIOS).
- UT est en général composée de mémoires, de registres, d'opérateurs logiques interconnectés. Elle prend en entrée des commandes (venant de l'UC) et éventuellement des valeurs venant de l'extérieur (qu'il est bon de faire transiter par l'UC, par exemple les valeurs immédiates des instructions NIOS qui sont décodées par l'UC).
- Pour la communication UT/UC, on a une demande de l'UC suivie d'une réponse de l'UT (ces demandes/réponses étant entrelacées).
  Avant de commencer son calcul, l'UT attend qu'une valeur lui soit envoyée (DEM=1 par exemple); une fois la valeur reçue, elle signale REP=1 et attend DEM=0 pour se rendre compte que l'utilisateur a bien vu que la machine avait pris sa valeur.
  Ce protocole est un **handshake**.
- Pour créer l'UT/UC, il faut écrire un algo de son fonctionnement, suffisamment détaillé (les entrées/sorties et leurs attentes doivent être complètes). Ainsi, on fait apparaître les opérateurs qui nous serviront.
- Paralléliser l'algorithme, i.e. mettre en évidence les calculs qui peuvent être faits en même temps. On sépare les instructions exécutées séquentiellement par ";", les parallèles par ",".
- Si les opérateurs ne sont pas utilisés simultanément, en insérer le moins d'exemplaires possibles et utiliser des bus partagés.
- On doit pouvoir obtenir ainsi un circuit pour l'UT.
- L'UC se déduit de l'algo : il y a autant d'états dans l'automate de l'UC que de paquets d'instructions réalisables simultanément.
- L'UC se chargera d'envoyer les bonnes commandes aux composants séquentiels de l'UT pour qu'ils exécutent l'algo comme il faut.

6 Composants mémoire

- Dans les systèmes que nous conçevons (durée de vie des données courte) : SRAM, DRAM se tirent la bourre.
  - Le prix ! SRAM bien plus chère que DRAM.
  - Mais le temps d'accès... 1 ns pour SRAM, 10/20 ns pour DRAM.

6.1 SRAM asynchrone

- Dispose d'entrées WE (Write Enable), CS (Chip Select), OE (Output Enable), A (Address) et d'un bus DQ (data) qui sert en entrée et en sortie.
- L'accès à cette RAM se fait comme un circuit combinatoire : envoyez votre commande, modulo le délai de propagation vous avez votre réponse.

6.2 SRAM synchrone

- On rajoute aux entrées de la SRAM synchrone, une horloge CLK.
- Lorsqu'une donnée est demandée au cycle n, elle est visible sur DQ au cycle $n + 1$.
- Pour l'écriture : dès que $WE = 1$, au front montant, l'écriture est activée et prend un cycle. L'adresse sur A doit être présente en même temps que les données sur DQ.

6.3 DRAM

- Une DRAM est une matrice de cellules mémoire : k lignes, m colonnes.
- Pour accéder à une donnée particulière, la DRAM charge la ligne dans laquelle cette donnée se situe dans un buffer, puis renvoie la colonne correspondante issue du buffer. Il y a donc un délai de mise en tampon puis un délai de lecture dans le tampon.
  Conclusion : pour bien utiliser la DRAM, **localisez vos accès** ! (rapprochez les valeurs utilisées dans des intervalles de temps faibles).
- La DRAM doit être maintenue "vivante" (durée de vie des données très faible) : il y a réécriture dans la matrice même après chaque lecture. En effet, la DRAM est une sorte de circuit RC, qui se décharge donc avec le temps (fuite de courant même à circuit ouvert).
- La SDRAM (DRAM synchrone) fonctionne en comptant des cycles : on fait des lectures en rafale (burst) pour des adresses contigues, une lecture par cycle, pour limiter le nombre de bufferings.
- On sait aussi (culturel) faire des RAM à double front d'horloge : entrée des adresses et commandes sur front montant, retour des données sur front descendant.

7 Processeur à jeu d'instructions

- On sait construire un automate et une UT tels que le système résultat fait, par exemple, la recherche du minimum dans un tableau.
- On se dote pour cela d'une UT "générique" : une unité arithmétique/logique (UAL, ALU pour les puristes), une file de registres (taille limitée), une mémoire de données. L'UC est adaptée à notre besoin : l'automate fait exactement la recherche du minimum.
- On peut avoir un processeur plus générique : on rajoute une unité de contrôle par dessus notre UC. Cette super-UC va passer des commandes à l'UC de base, qu'elle comprendra. C'était l'objet du TP "construction d'un processeur" où l'on a dessiné le gros automate.
- La super-UC va avoir pour fonction de chercher une instruction dans une mémoire d'instruction, et de la passer à l'UC interprète, qui décode l'instruction et la fait exécuter.
- Pour que cette super-UC sache que lire : il faut un compteur d'instructions : le **Program Counter** ! Et un registre (**instruction register**) pour stocker l'instruction en cours d'exécution et en permettre le décodage.
- Les instructions machine sont standardisées : on définit un codage, par exemple sur le NIOS, on a trois types d'instructions (J, R, I). On y reviendra à la section NIOS.

- Machines CISC/RISC : les machines CISC[3] étaient là historiquement pour simplifier le travail des programmeurs, qui n'avaient pas de langage de haut niveau (ex : comparaison de chaînes de caractères en une instruction...). Les RISC[4] sont basées sur le fait que 90% des instructions sont utilisées 10% du temps (et donc 10% des instructions, 90% du temps). Du coup, moins d'instructions ⇒ CPU moins complexe.
- Les processeurs modernes (Core 2, ...) ont un cœur RISC et un système de traduction de x86 vers leur jeu RISC.

8 Le cas du NIOS

8.1 Instructions

- La plupart des NIOS ont un accès mémoire aligné au mot (4 octets) (ATTENTION ! Votre programme plante si vous demandez l'accès à une adresse non alignée).
- Endianness : le NIOS est est **little endian** (| poids faible | .. | .. | poids fort |). Exemple, 0xB16B00B5 se représente | 0xB5 | 0x00 | 0x6B | 0x16 |.
- Le format I, J R des instructions sur 32 bits:
  - I : | rA (5 bits) | rB (5 bits) | IMM16 (16 bits) | OP (6 bits) |
  - J : | rA (5 bits) | rB (5 bits) | rC (5 bits) | OPx (11 bits) | OP (6 bits) |
  - R : | IMM26 (26 bits) | OP (6 bits) |
- Quelques instructions à retenir... (PC + 4 : incrément PC post-instruction, PI : Pseudo-Instruction, traduite par l'assembleur en multiples instructions)

Instr	T	Syntaxe	Effet	PC+4	PI
Instructions arithmétiques					
add	J	`add rC, rA, rB`	$rC := rA + rB$	X	
addi	I	`addi rB, rA, IMM16`	$rB := rA + \sigma(\text{IMM16})$	X	
sub	J	`sub rC, rA, rB`	$rC := rA - rB$	X	
subi	I	`subi rB, rA, IMM16`	$rB := rA - \sigma(\text{IMM16})$	X	X
Idem pour and, andi, or, ori, xor, xori.					
Instructions de branchement, appels...					
call	R	`call label`	$r31 := PC + 4$; $PC := PC_{31..28} : (\text{IMM26} \ll 2)$		
callr	I	`callr rA`	$r31 := PC + 4; PC := rA$		
ret	R	`ret`	$PC := r31$		
jmp	R	`jmp rA`	$PC := rA$		
jmpi	I	`jmpi label`	$PC := PC_{31..28} : (\text{IMM26} \ll 2)$		
beq	I	`beq rA, rB, label`	$PC := PC + 4$ $+ (\texttt{if}\ rA = rB\ \texttt{then}\ \sigma(\text{IMM16})\ \texttt{else}\ 0)$		
bne	I	`bne rA, rB, label`	$PC := PC + 4$ $+ (\texttt{if}\ rA! = rB\ \texttt{then}\ \sigma(\text{IMM16})\ \texttt{else}\ 0)$		
bge	I	`bge rA, rB, label`	$PC := PC + 4$ $+ (\texttt{if}\ rA \geqslant rB\ \texttt{then}\ \sigma(\text{IMM16})\ \texttt{else}\ 0)$		
Idem pour ble, blt, bgt (less or equal, less than, greater than)					
Instructions de mouvement, (dé)chargement mémoire					
movia	I	`movia rB, IMM16`	$rB := \text{IMM16}$	X	X
movi	I	`movi rB, IMM16`	$rB := \sigma(\text{IMM16})$	X	X

[3] Complex Instruction Set Computer
[4] Reduced Instruction Set Computer

mov	J	`mov rC, rA`	$rC := rA$	X	X
ldw	I	`ldw rB, offset14(rA)`	$rA := \mathbf{MEM}(rA + \text{offset14})$	X	
stw	I	`stw rB, offset14(rA)`	$\mathbf{MEM}(rA + \text{offset14}) := rB$	X	

- Note culturelle : sur les x86, on a l'instruction lea (Load Effective Address) qui permet de calculer l'adresse effective (offset + valeur du registre) sans faire appel à la mémoire (on fait un mov après). Elle est intéressante car contrairement à add, elle ne modifie pas les flags du processeur (retenue, overflow, signe) et permet de faire de l'arithmétique moins limitée que add[5].
- Adressage direct/indirect/basé :
  - L'adressage direct, le plus simple, consiste à passer directement l'adresse cible en mémoire dans l'instruction (`ldw rB, [addr](r0)`)
  - L'adressage indirect consiste à utiliser un registre relais... par exemple pour utiliser une adresse sur 32 bits (sachant que les instructions font 32 bits). Typiquement, `ldw rB, 0(rA).`
  - L'adressage basé consiste à utiliser un registre relais et une valeur immédiate... par exemple pour lire une valeur dans une matrice dont on connaît l'adresse de départ. Typiquement, `ldw rB, offset(rA).`

8.2 Directives d'assembleur

- Dans l'assembleur : les directives `.text` et `.data` permettent de séparer le code (`.text`) des données statiques (`.data`).
- Réservation de données statiques: les labels sont facultatifs
  - Chaînes : [label] .string "Text"
  - Octets : [label] .byte value [:length]
  - Mots : [label] .word 0x12345678
  - Demi-mots : [label] .half 0x1234
  - Alignement : .align 16 (saute à la prochaine adresse alignée sur 16 bits). Sur le NIOS, .align 32 aligne à 4 octets ce qui correspond au requirement d'alignement dans les adresses.
  - Exportation : .globl label (rend label visible hors de votre programme, pour qu'on puisse y faire appel depuis l'extérieur - pensez aux librairies partagées).

8.3 Stack

- Pile : contient les arguments des fonctions appelées, les adresses de retour... Schéma de la mémoire d'un programme.

Program	
Program Static Data	
Heap (goes ↓)	
Stack top (goes ↑)	← Stack pointer points here
Stack	
(Stack bottom)	

- Le NIOS a un registre contenant le sommet de la pile (stack pointer, r27). Lorsque vous empilez des données, décrémentez-le **avant** empilement ! Lorsque vous en dépilez, incrémentez-le après dépilement.

[5]What does that give us? Two things that ADD doesn't provide: the ability to perform addition with either two or three operands, and the ability to store the result in any register; not just one of the source operands. *(Abrash, Zen of Assembly)*

- **ATTENTION ! Le Stack Pointer doit TOUJOURS être aligné sur un mot !** Ne tentez surtout pas de travailler dessus octet par octet (prévoyez le cas où votre programme est interrompu et où il faut en sortir... que devient la pile ?)
- Lorsque vous faites appel à une fonction, vous empilez la valeur de r31 (ra, return address) pour pouvoir retrouver cette adresse après coup en la dépilant et retourner là où votre programme l'avait prévu. Cela permet les appels imbriqués.
- Le Frame pointer (r28) est un registre mis à jour avant chaque appel de fonction, contenant la valeur du stack pointer à ce moment : si vous laissez des données dans la pile en retournant, le stack pointer est remis à sa valeur d'avant l'appel... ce qui ne vous dispense pas de bien gérer la pile !
- Attention aux conventions de sauvegarde en pile des registres. (Celle de gcc impose que l'appelant sauvegarde les registres 8 à 15, et que l'appelé sauvegarde les registres 16 à 23).
- Par convention, les registres 2 et 3 sont utilisés pour le retour des résultats de votre programme, les registres 4 à 7 servent pour ses arguments.

9 Entrées et sorties

- Des coupleurs matériels (contrôleurs) reliés au CPU, auquel on rajoute:
  - un bus d'adresse I/O
  - un bus de données I/O
  - un bus de commande (choix du coupleur)
- Chaque coupleur a:
  - un bus d'adresse en entrée
  - un bus servant aux données (entrée + sortie dotée d'un tristate, sur le même bus)
  - une entrée CS (chip select) : le coupleur lui-même a une adresse et son chip select n'est activé que si sur le bus d'adresse on trouve son adresse.
  - une sortie Waitreq (acquittement de réception des commandes, et de bonne exécution de celles-ci)
- On peut disposer de ports de lecture/écriture multiples sur un même coupleur en réservant des bits sur le bus d'adresse pour une sous-adresse :

adresse chip (comparée avec un ET, résultat en chipselect) (eg. 6 bits)	adresse port (eg. 2 bits)

- Le coupleur a typiquement quelques registres : pour les lire, le processeur effectue une transaction (demande, attente d'acquittement, lecture); pour écrire, idem (demande avec envoi de données, attente d'acquittement d'écriture, libération des bus).
- Point technique : le processeur a un registre d'état, référençant entre autres le mode superutilisateur (droit de parler au matériel), l'autorisation des interruptions (voir paragraphe suivant).
- Instructions non dites dans le tableau : ldwio, stwio, parlent au matériel (**seulement en mode superutilisateur**, donc seul le noyau de l'OS a le droit de les utiliser).

9.1 Interruptions matérielles

- Les coupleurs matériels ont un drapeau (flag) indiquant s'ils ont des données prêtes à être traitées par le processeur. Le processeur interrompt le programme en cours, sauvegarde ses registres d'état et PC, exécute une routine de traitement d'interruption (à une adresse fixée, EXCEPTION_HANDLER).

- Pendant l'exécution de la routine d'interruption, le CPU ne peut être interrompu à nouveau : les flags d'interruption sont soumis à un AND avec un flag du processeur, le Processor Interrupt Enable (**PIE**). Si ce flag est à 0, le processeur n'est pas interrompu.
- Sections critiques : ce sont des moments où votre programme ne doit pas être interrompu... On y reviendra en système, mais on peut masquer soi-même les interruptions (dans le cas où la routine de traitement met à jour une variable partagée).

10 Cache

- Le cache a pour vocation de réduire le temps d'accès aux données souvent utilisées, ou locales relativement à celles sur lesquelles on travaille.
- Si vous cuisinez, vous ne remballez pas la farine au placard entre chaque cuillère, vous la gardez sur la table !
- Cache : mémoire rapide (SRAM, typiquement) localisée proche du processeur (voire, maintenant, dans celui-ci). Faible capacité.
- On trouve des méta-données et les données cherchées dans un cache : par exemple

Valid	Tag	Data
1 bit	20 bits	32 bits

- Un cache est donc "plus large" qu'une simple case mémoire.
- Une info que l'on ne voit pas sur le schéma : l'adresse dans le cache ! Celle-ci est une info utilisée, elle n'est pas choisie au hasard, c'est le "set". Tag et set se déduisent de l'adresse :

Tag (20 bits)	Set (10 bits)	2 bits non utilisés (00, alignement au mot)
Poids forts		Poids faibles

- La taille du champ set dépend de la taille du cache... du coup, la taille du tag aussi ! Ne pas toujours considérer 20 bits.
- Lecture dans le cache :
  - Demande CPU : chargement adresse `[addr]`
  - Extraction `set`, `tag`
  - Comparaison : `cache[set].tag == tag` ET `Valid == 1`
  - Si vrai, alors **hit** (on a les données en cache, on les renvoie)
  - Sinon, **miss** (il faut aller chercher les données en mémoire). Avant d'aller les chercher, si le cache est de type write-back et `Valid == 1`, on réécrit les anciennes données en cache qui sont plus à jour que la mémoire, à l'ancienne adresse.
- Cache à n lignes seulement organisé de cette manière : peu performant (parcours de longs tableaux, données dont les `set` se marchent dessus...).
- Mieux : le cache associatif. On concatène deux entrées de cache ordinaire, pour le même set...

set k :	Valid	Tag	Data	Valid	Tag	Data	...	Valid	Tag	Data	(m voies)

- Alors pour l'accès au cache, on fait un ET sur chaque tag par rapport au tag de l'adresse cible, puis un OU nous indique si le cache a les données ou pas. Un multiplexeur choisit la bonne voie[6].
- Cache associatif "parfait" : une seule entrée, N voies... Oui, mais trop de comparateurs, logique trop compliquée.

[6] dans le cas d'un cache à 2 voies, on met la sortie du `Valid == 1` ET `cache[set][0].tag == tag` sur le sélecteur du muxer. Si la voie 0 a l'info, c'est correct (le muxer sort la voie 0); si la voie 0 n'a pas l'info et la voie 1 l'a, c'est correct; sinon, les données en sortie du muxer ne sont pas lues (miss)

- Qui évincer lorsque l'on a un miss ? Politique : LRU (Least Recently Used) : on évince le moins récemment utilisé.
- Un cache encore meilleur pour la localité spatiale : le cache en lignes, pour charger les données adjacentes à celles sur lesquelles on travaille (il y a des chances qu'en lisant une valeur, on lise celles qui la suivent, eg. si on lit un tableau).

  set k :

Valid	Tag	Data[0]	Data[1]	Data[2]	Data[3]

  (nombre de données: puissance de 2!)
- Dans ce cas, le tag n'est pas le même :

Tag	Set	Offset	2 bits non utilisés
18 bits	10 bits	2 bits (cas présent)	

- L'`offset` est passé à un muxer pour qu'il choisisse la bonne voie de données (ce qui nécessite une largeur des données en puissance de 2). Ce cache fonctionne comme le premier (comparaison de `tag` à `set` donné).

11 Pipeline

Chapitre récent. Je n'ai pas encore eu le temps de l'écrire. Si je trouve le temps... Parler des étages (étages de base : Fetch, Decode, EXecute, MEMory, WriteBack), des aléas structurels, de données (attente RAW); parler des pipelines multiples (sur certaines microarchis) et des WAW qui se présentent).

12 Annexe : complément à deux et extension de signe

Le complément à deux d'un nombre se calcule en inversant tous ses bits et en ajoutant 1. Cela permet d'en obtenir l'opposé.

Exemple sur 8 bits :

$$\begin{aligned} 36 &= 00100100 \\ -36 = \overline{36} &= \overline{00100100} \\ &= 11011011 + 1 \\ &= 11011100 \end{aligned}$$

Pourquoi le complément à deux ? Pour que 0 ait une unique représentation; sinon, si on se contentait d'ajouter le bit de signe, on aurait $0 = 10000000 = 00000000$ (sur 8 bits).

L'extension de signe consiste à copier le bit de signe d'un nombre sur moins de bit que son conteneur de destination.

Par exemple, pour passer de 16 à 32 bits :

$$\begin{aligned} 42_{16} &= 0000000000101010 \\ -42_{16} &= 1111111111010101 + 1 \\ &= 1111111111010110 \end{aligned}$$

Si on se contente de copier en ajoutant des zéros comme padding, pour des nombres positifs, tout va bien, mais là... :

$$00000000000000001111111111010110 = 65494$$

Si on étend le bit de signe :

$$11111111111111111111111111010110 = -42_{32}$$

D'où la nécessité de l'extension de signe notée σ.

13 Annexe : représentation des objets en mémoire

- D'abord un peu de terminologie. Communément, elle est admise, mais on peut vous sortir une architecture où ces termes diffèrent : prudence donc.
  - Mot : 32 bits (word, en C : `int`)
  - Demi-mot : 16 bits (half-word, en C : `short`)
  - Octet : 8 bits (byte, en C : `char`[7])
  - Mot long : 64 bits (long word, en C : `long int`)
- Les structures de données en C permettent de mettre à la suite en mémoire un certain nombre de données : l'ordre de stockage en mémoire dépend de l'ordre dans lequel vous avez déclaré les variabes !
- Exemple:

```
struct packet {
        int date;
        char* origin;
        char packetStatus;
}
struct packet monPaquet; // création de monPaquet statiquement -> dans la stack
```

- Cette structure est de taille 4 + 4 + 1 = 9 octets... mais vous allez utiliser 12 octets pour la fabriquer (vous en perdez 3) : vous avez les champs `date` et `origin` qui font chacun 32 bits (4 octets), donc chacun doit être aligné au mot (4 octets). Le `char packetStatus` qui reste fait cavalier seul sur un autre mot de la stack.
- Les contraintes d'alignement peuvent donc induire un gaspillage de mémoire ! Mais il n'y a pas de réorganisation de l'ordre dans lequel vous pouvez trouver les variables. Préférez `monPaquet.origin` à `monPaquet+4` dans le cas où vous feriez un code qui tourne sur plusieurs architectures.
- **À retenir :** les variables de votre programme (locales ou pas) sont alignées au mot sur le NIOS II, et au sein des structures, les contraintes d'alignement sont respectées -*en particulier, en aucun cas on ne trouvera un paquet de données de moins d'un mot à cheval sur deux mots.*

[7] à ne pas prononcer "kar" mais bien "shar" !!!!