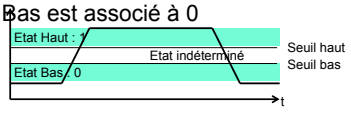
3. Algèbre de Boole et circuits logiques

Plan

- Algèbre de Boole et fonctions booléennes
  - Algèbre de Boole
  - Représentations des fonctions booléennes
    - Formes normales
    - Minimisation
- Circuits combinatoires
- Circuits séquentiels

Motivation

- Information Etat Haut - Etat Bas
- Abstraction : v
  - Etat Haut est associé à 1
  - Etat Bas est associé à 0



On étudie les propriétés formelles de la représentation abstraite uniquement en vue d'une *réalisation matérielle.*
->Vision très partielle

Algèbre de Boole

Soit *E* un ensemble non vide, muni de deux opérations binaires + et ., d'une opération unaire notée $^-$, et deux éléments particuliers de E notés 0 et 1. (*E*, 0, 1, +, ., $^-$) est une algèbre de Boole si

(i) les opérations + et . sont commutatives et associatives

(ii) 0 est neutre pour + et 1 pour .

(iii) + et . sont distributives l'une sur l'autre

(iv) $\forall a \in E, a.\bar{a} = 0$ et $a + \bar{a} = 1$

L'algèbre de Boole des circuits logiques

• E = {0, 1} et +, . , $^-$ sont définies formellement par :

+	0	1
0	0	1
1	1	1

.	0	1
0	0	0
1	0	1

$^-$	
0	1
1	0

• Interprétation logique, au sens philosophique,

. = ET, + = OU, $^-$ = NEGATION (NOT)

• Notation : priorité de . sur +

Propriétés

Si (E,0,1,+,.,-) est une algèbre de Boole, (E,1,0,.,+,-) est aussi une algèbre de Boole.

!!a = a

a + a = a et a.a = a

a + 1 = 1 et a.0 = 0

a + 0 = a et a.1 = a

Règles de De Morgan

$\forall a,b \in E, \overline{a+b} = \bar{a}.\bar{b}$

$\forall a,b \in E, \overline{a.b} = \bar{a} + \bar{b}$

Fonctions booléennes

- Fonctions de $\{0,1\}^n$ vers $\{0,1\}$
- Représentations
  - En extension : tables de vérité
  - Algébrique : f(a,b,c) = !b + a.c
  - La représentation algébrique n'est pas unique : a + a = a

a	b	c	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Minterms

Un terme produit (minterm) est un produit de toutes les variables d'entrées, complémentées ou non.

- Pour n variables d'entrée, il y a 2^n minterms
- Chaque minterm correspond à une ligne de la table de vérité. Chaque variable est complémentée si dans cette ligne sa valeur est 0, non complémentée sinon

$m_0 = \overline{x}_1.\overline{x}_0$
$m_1 = \overline{x}_1.x_0$
$m_2 = x_1.\overline{x}_0$
$m_3 = x_1.x_0$

x0	x1	m3	m2	m1	m0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

« Théorème » de Shannon

$f(x_1, \ldots x_n) = \overline{x_1}.\ f(0, x_2, \ldots x_n) + x_1.\ f(1, x_2, \ldots x_n)$

Trivial: Les deux membres sont identiques pour $x_1 = 0$ et $x_1 = 1$

Applications

- Ecrire la fonction comme un polynôme aux x_i et x_i complémentés (FDN, FCN)
- Exprimer l'expression à partir de fonctions multiplexeurs

Forme disjonctive normale

La forme disjonctive normale d'une fonction est le OU des minterms pour lesquels la fonction vaut 1

a	b	c	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$$f = \bar{a}.\bar{b}.\bar{c} + \bar{a}.\bar{b}.c + a.\bar{b}.\bar{c} + a.\bar{b}.c + a.b.c$$

Notation : Σm_i
Exemple
$f = m_0 + m_1 + m_4 + m_5 + m_7$

Forme conjonctive normale

- Un terme somme (maxterm) est une somme de toutes les variables d'entrées, complémentées ou non.
- Chaque maxterm correspond à une ligne de la table de vérité. Chaque variable est complémentée si dans cette ligne sa valeur est 1, non complémentée sinon.

La forme conjonctive normale d'une fonction est le ET des maxterms pour lesquels la fonction vaut 0.

a	b	c	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

$$f = (a + \bar{b} + c).(a + \bar{b} + \bar{c}).(\bar{a} + \bar{b} + c)$$

Notation : ΠM_i
Exemple
$f = M_2 . M_3 . M_6$

Contexte : expressions booléennes

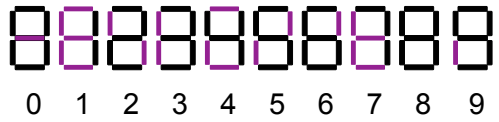
- Toute expression de n variables booléennes
- Grâce aux lois de De Morgan, on peut considérer que les négations ne concernent que les variables individuelles
- Forme disjonctive normale (FDN)
  OU de ET
  a.!b + a.c.d
- Forme conjonctive normale (FCN)
  ET de OU
  (a + !b) . (a+c+d)

Contexte : expressions booléennes

- FCN : chaque clause exprime une contrainte
- Question fondamentale : satisfiabilité
- Satisfiabilité d'une expression en FDN :
  au moins un ET ne contient pas une variable et son complément
- Le problème général de la satisfiabilité est NP complet

Fonctions incomplètement spécifiées

La valeur de la fonction n'est pas significative pour certaines valeurs des variables d'entrée.



0 1 2 3 4 5 6 7 8 9

Le segment a est allumé pour 0, 2, 3, 5, 6, 7 ,8 ,9

$$a = e_1 + e_3 + \overline{e_2}.\overline{e_0} + e_2.e_0$$

Minimisation

- Expression minimisée d'une fonction logique : « plus simple»
  - Moins de minterms : plus petit OU
  - Moins de variables dans chaque minterm : plus petits ET
- Contrainte : logique à 2 niveaux – FD ou FN

a	b	f
0	0	1
0	1	0
1	0	1
1	1	1

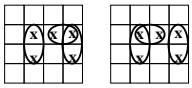
$$\overline{a}\overline{b} + a\overline{b} + ab = a + \overline{b}$$

Minimisation

- Monôme *premier* : qui n'est contenu dans aucun autre
- *Base irredondante* : ensemble de monômes premiers
  - (a) Dont la somme est égale à la fonction
  - (b) Telle que si on enlève un monôme, (a) n'est plus vraie
- *Base irredondante minimale* : en nombre de monômes, puis en taille de monômes
  - Pas d'unicité
  - Le problème est NP-complet. Des algorithmes heuristiques professionnels existent.
  - Tables de Karnaugh: méthode jouet

Bases irredondantes minimales

Pas d'unicité



Equivalentes

Plus gros monôme
Non optimale

Principe des tables de Karnaugh

Exploiter la propriété

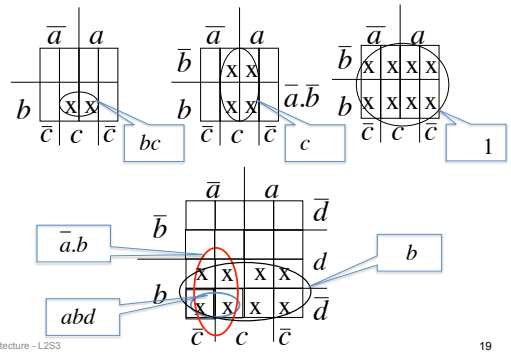
a + !a = 1

par une représentation graphique

$$\overline{a}\overline{b} + a\overline{b} + ab = (\overline{a}\overline{b} + a\overline{b}) + (a\overline{b} + ab)$$
$$= \overline{b} + a$$

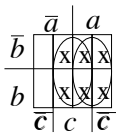
	$\overline{a}$	a
$\overline{b}$	X	X
b		X

Tables de Karnaugh : exemples



Tables de Karnaugh : exemples

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



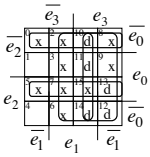
$$f = a + c$$

La fonction peut s'écrire comme le OU des regroupements par puissances de 2

Fonctions incomplètement spécifiées

La valeur de la fonction n'est pas significative pour certaines valeurs des variables d'entrée.

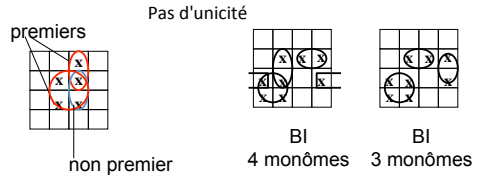
Le segment a est allumé pour 0, 2, 3, 5, 6, 7 ,8 ,9



$$a = e_1 + e_3 + \overline{e_2}.\overline{e_0} + e_2.e_0$$

Retour sur définitions

- Monôme premier : qui n'est contenu dans aucun autre
- *Base* irredondante : ensemble de monômes premiers
  (a) Dont la somme est égale à la fonction
  (b) Telle que si on enlève un monôme, (a) n'est plus vraie



Plan

- Algèbre de Boole et fonctions booléennes
- Circuits combinatoires
  - Portes logiques
  - Opérateurs élémentaires : multiplexeurs, décodeurs, encodeurs
  - Additionneur
- Circuits séquentiels

Circuits combinatoires

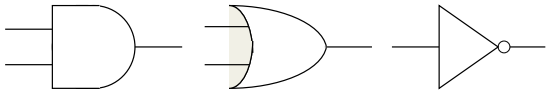
- Dispositif matériel qui calcule = réalise une ou des fonctions booléennes
  $f_0(e_0, e_1, \ldots e_{n-1}), \ldots, f_{p-1}(e_0, e_1, \ldots e_{n-1})$
- Synthèse logique : méthodes de conception des circuits combinatoires
- Compromis temps de propagation / encombrement / facilité de conception

Les opérateurs élémentaires

- Les opérateurs logiques : par les portes ET, OU, NAND, NOR, XOR…
- Portes complexes, dépendant de la technologie
- La sélection : par le multiplexeur ou le décodeur

Portes logiques

Porte logique : circuit qui réalise une fonction élémentaire



ET
AND
.

OU
OR
+

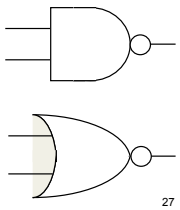
Inverseur
NOT
–

Portes NAND et NOR

- Les portes ET et OU ne sont pas les plus simples à réaliser en technologie CMOS
- Les vraies portes élémentaires sont les portes NAND, NOR et NOT

$$NAND(a,b) = \overline{a.b}$$

$$NOR(a,b) = \overline{a + b}$$



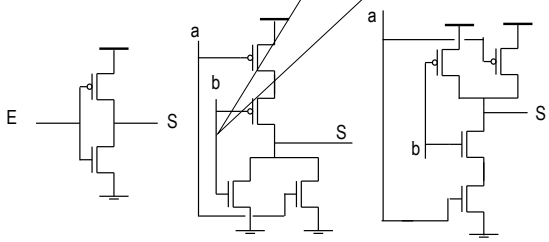
Technologie CMOS

- CMOS = Complementary Metal-oxide semiconductor
- Ici, transistor = interrupteur
- Le transistor P est ouvert si G=haut, fermé si G = bas
- Le transistor N est ouvert si G= bas, fermé si G = haut

Transistor N Transistor P

Portes

Dès que a ou b vaut 1, un des deux t. en série est ouvert et un des 2 t. en // est passant



Inverseur Porte NOR Porte NAND

Portes NAND et NOR

- NAND et NOR sont fonctionnellement complets : on peut réaliser ET, OU et NOT a partir de chacun d'entre eux

$\bar{a} = \overline{a.1} = \text{NAND}(a, 1)$

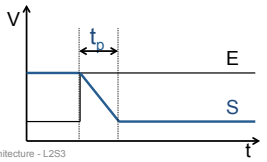
$\bar{a} = \overline{a.a} = \text{NAND}(a, a)$

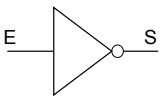
$a.b = \overline{\overline{a.b}} = \text{NOT}(\text{NAND}(a,b))$

$a + b = \overline{\bar{a}.\bar{b}} = \text{NAND}(\text{NOT}(a), \text{NOT}(b))$

Portes logiques et algèbre de Boole

- Les portes logiques ne réalisent les fonctions logiques qu'à certains points dans le temps
- Le *temps de propagation* d'un circuit est le temps qui sépare le positionnement des entrées de celui de la sortie
- Ordres de grandeur
  - Lumière dans le vide : 4ns/m – nano seconde – 10^{-9}
  - Temps de commutation: pico seconde – ps – 10^{-12}

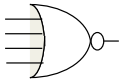




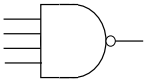
$NAND_k$ et NOR_k

$NAND_k(x_1, \ldots, x_k) = \overline{x_1.x_2. \ldots .x_k}$

$NOR_k(x_1, \ldots, x_k) = \overline{x_1 + x_2 + \ldots + x_k}$



NOR4



NAND4

Mais les contraintes technologiques limitent à un petit nombre d'entrées.

FDN et portes logiques

La forme disjonctive normale peut aussi s'exprimer par un NAND de NAND, en utilisant éventuellement des portes $NAND_k$

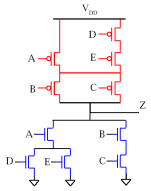
$$f = \bar{a}.\bar{b}.\bar{c} + \bar{a}.\bar{b}.c + a.\bar{b}.\bar{c} + a.\bar{b}.c + a.b.c$$
$$= \overline{\bar{a}.\bar{b}.\bar{c} + \bar{a}.\bar{b}.c + a.\bar{b}.\bar{c} + a.\bar{b}.c + a.b.c}$$
$$= NAND_4(NAND_3(\bar{a}.\bar{b}.\bar{c}), NAND_3(\bar{a}.\bar{b}.c), NAND_3(a.\bar{b}.\bar{c}), NAND_3(a.\bar{b}.c), NAND_3(a.b.c))$$

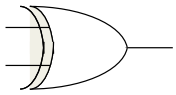
Le OU exclusif - XOR

a	b	a ⊕ b
0	0	0
0	1	1
1	0	1
1	1	0

$$a \oplus b = \bar{a}.b + a.\bar{b}$$

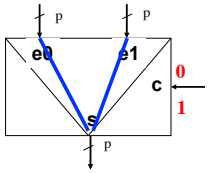
Porte XOR





Les Multiplexeurs

- Fonction : sélection
- Le + simple : 1 parmi 2
  - si **c** == 0
  - alors
    - s = e0
  - sinon
    - s = e1



Les Multiplexeurs 2^n:1

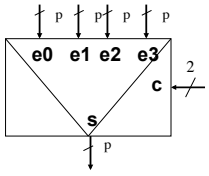
1 parmi 2^n

2^nxp entrées de données,

n entrées de contrôle,

1xp sortie

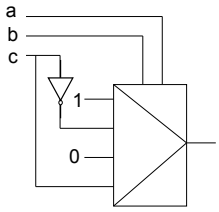
$S = e_i$ si $c_{n-1}...c_0 = i$



Multiplexeurs

Toute fonction Booléenne de n variables peut être implementée avec un multiplexeur 1 parmi 2^{n-1}.

a	b	c	f	
0	0	0	1	S = 1
0	0	1	1	
0	1	0	1	S = $\bar{c}$
0	1	1	0	
1	0	0	0	S = 0
1	0	1	0	
1	1	0	0	S = c
1	1	1	1	



Multiplexeurs

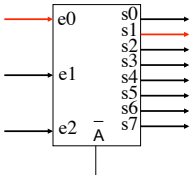
Pour une fonction de *n* variables, par exemple f(A,B,C,D)

- Avec un MUX 1 parmi 2^{n-1} donc *n*-1 lignes de sélection
- Définir un ordre des variables, par exemple e3 = A, e2 = B, e1 = C, e0 = D
- Les *n*-1 bits de poids fort vont vers les *n*-1 lignes de sélection, par exemple A,B,C
- Pour deux lignes consécutives, les valeurs possibles de la sortie sont soit 0, soit 1, soit e0, soit !e0.
- Connecter 0, e0, !e0, ou 1 à chaque entrée d'après le résultat de l 'étape précédente.

Les décodeurs

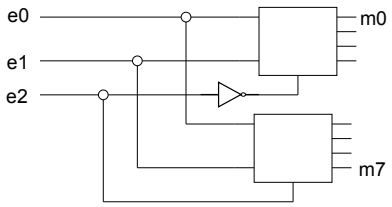
- *n* entrées de données, 2^n sorties = les minterms
  Donc, 1 seule sortie active un instant donné

  $S_i = 1$ si $e_{n-1} \ldots e_0 = i$
- Toute fonction booléenne peut être générée avec un décodeur et un OR : FDN



Réalisation structurée

Un décodeur 3:8 à partir de décodeurs 2:4

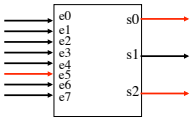


Les encodeurs

Fonction réciproque du décodeur :

Une seule entrée à 1, écriture binaire du numéro de l'entrée

e3	e2	e1	e0	s1	s0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

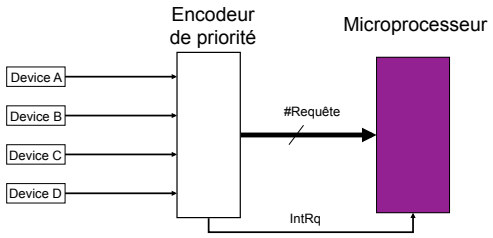


Encodeur de priorité

- Priorité au 1 de plus fort poids
- Bit de validité
- Encodeur de priorité 4:2

e3	e2	e1	e0	s1	s0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	d	0	1	1
0	1	d	d	1	0	1
1	d	d	d	1	1	1

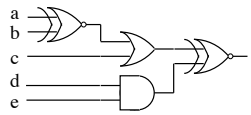
Gestion d'interruption



Réalisation de fonctions booléennes

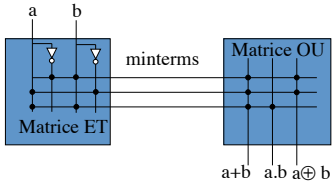
a. Une fonction de quelques variables, pas de régularité : logique "anarchique"

(a Å b +c) Å (d.e)



Réalisation de fonctions booléennes

b. Plusieurs fonctions de plusieurs variables, pas de régularité : Programmable Logic Array (PLA)
   - Optimise la surface et la faisabilité : q minterms, q à minimiser
   - Collectivement sur un ensemble de fonctions

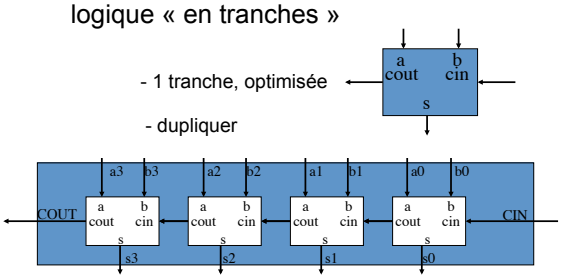


Réalisation de fonctions booléennes

c. nx1 fonction de plusieurs variables : logique « en tranches »

- 1 tranche, optimisée

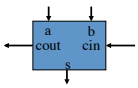
- dupliquer



L'additionneur

L'additionneur 1 bit

a	b	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

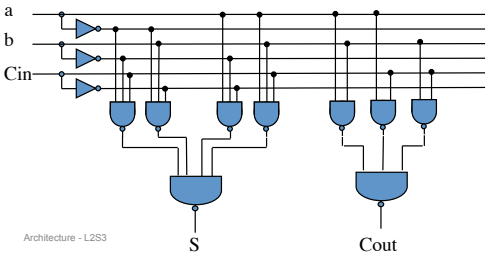


$S = \bar{a}.\bar{b}.Cin + \bar{a}.b.\overline{Cin} + a.\bar{b}.\overline{Cin} + a.b.Cin$

$Cout = a.b + a.Cin + b.Cin$

L'additionneur 1 bit en portes $NAND_k$

$S = \bar{a}.\bar{b}.Cin + \bar{a}.b.\overline{Cin} + a.\bar{b}.\overline{Cin} + a.b.Cin$ $\quad Cout = a.b + a.Cin + b.Cin$



L'additionneur 1 bit

Hypothèse : 1*t* par porte $NAND_k$ ou NOR_k

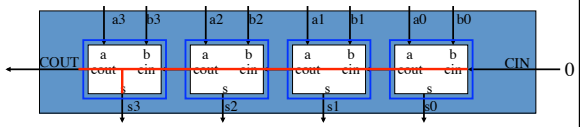
$$S = \bar{a}.\bar{b}.Cin + \bar{a}.b.\overline{Cin} + a.\bar{b}.\overline{Cin} + a.b.Cin$$

Temps de propagation : 3*t*

$$Cout = a.b + a.Cin + b.Cin$$

Temps de propagation : 2*t*

L'additionneur *n* bits à propagation de retenue



Chemin critique : plus long chemin (en temps) des entrées vers une sortie

Additionneur :

- 2*nt* pour COUT
- (2*n* + 1)*t* pour Sn, chemin critique
- Le problème provient du temps de propagation de la retenue

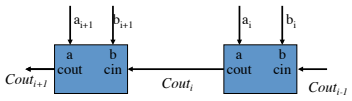
Retenue anticipée

- Accélérer le calcul de la retenue de sortie

Soient $P = a + b$ *propager une retenue à 1*

$G = a.b$ *générer une retenue*

Alors $Cout = G + P.\ Cin$



Retenue anticipée

Si $P = a + b$ et $G = a.b$
alors $Cout = G + P \,.\, Cin$

En effet

$Cout = a.b + a.Cin + b.Cin$
$= G + P \,.\, Cin$

Remarque: vrai aussi pour $P = a$ XOR b

Circuit anticipateur de retenue

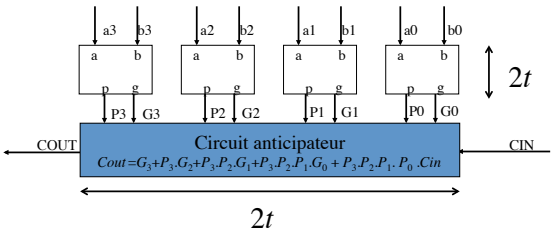
- Calcul en 2 couches logiques de la retenue sortante.

Exemple sur 4 bits

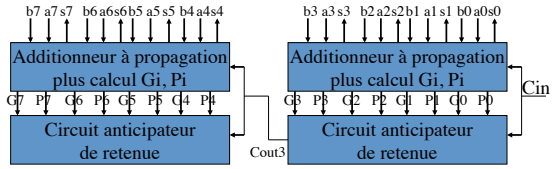
$Cout_3 = G_3 + P_3 \,.\, Cout_2$
$= G_3 + P_3 \,.(G_2 + P_2 \,.\, Cout_1)$
$= G_3 + P_3 \,.(G_2 + P_2 \,.(G_1 + P_1 \,.\, Cout_0)$
$= G_3 + P_3 \,.(G_2 + P_2 \,.(G_1 + P_1 \,.(G_0 + P_0 \,.\, Cout_{-1})$

$Cout_3 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0 \,.\, Cout_{-1}$

Circuit anticipateur de retenue



Additionneur à retenue anticipée – Version 1



*S*3 9*t*, le calcul anticipé ne change rien
*Cout*3 2*t* pour *Gi*, *Pi*
plus 2*t* pour le circuit anticipateur : 4*t*
*S*7 temps de calcul de *Cout*3 plus additionneur à propagation : 13*t*
Au lieu de 17*t* dans l'additionneur à propagation simple

Additionneur à retenue anticipée – Version 2

- Calcul anticipé de toutes les retenues

$$Cout_0 = G_0 + P_0 .Cout_{-1}$$
$$Cout_1 = G_1 + P_1.G_0 + P_1. P_0 .Cout_{-1}$$
$$Cout_2 = G_2 + P_2.G_1 + P_2. P_1 G_0 + P_2. P_1 P_0 .Cout_{-1}$$

etc.

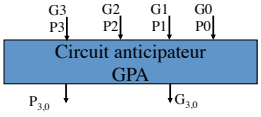
- Toutes les retenues se calculent en 4t
  - 2t pour les Gi, Pi
  - 2t pour chaque retenue indépendamment
- Tous les Si en 7t : 4 pour les retenues et 3 pour le calcul de Si

Additionneur à retenue anticipée

- Problème : l'hypothèse 2 couches logiques = 2*t* n'est valide que pour un petit nombre d'entrées, par exemple 4.
- Le même principe peut s'appliquer récursivement

$$Cout_3 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.Cout_{-1}$$

$G_{3,0}$ $P_{3,0}$

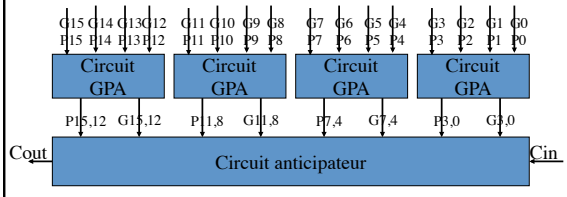


Retenue anticipée par blocs

$Cout_3 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0 .Cout_{-1}$
$Cout_3 = G_{3,0} + P_{3,0}. Cout_{-1}$
$Cout_7 = G_7 + P_7.G_6 + P_7.P_6.G_5 + P_7.P_6.P_5.G_4 + P_7.P_6.P_5.P_4 .Cout_3$
$Cout_7 = G_{7,4} + P_{7,4}. Cout_3$
$= G_{7,4} + P_{7,4}.(G_{3,0} + P_{3,0}. Cout_{-1})$
…
$Cout_{15} = G_{15,12} + P_{15,12}.G_{11,8} + P_{15,12}.P_{11,8} G_{7,4} +$
$P_{15,12}.P_{11,8} P_{7,4} G_{3,0} + P_{15,12}.P_{11,8} P_{7,4} P_{3,0} Cout_{-1}$

Retenue anticipée par blocs

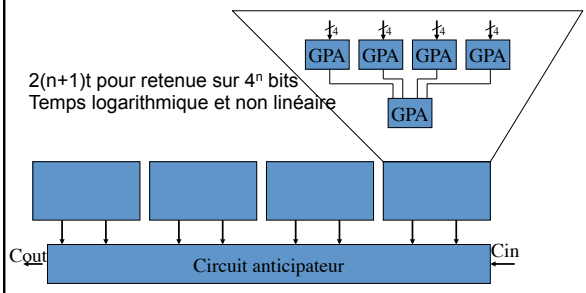
$Cout_{15} = G_{15,12} + P_{15,12}.G_{11,8} + P_{15,12}.P_{11,8} G_{7,4} +$
$P_{15,12}.P_{11,8} P_{7,4} G_{3,0} + P_{15,12}.P_{11,8} P_{7,4} P_{3,0} Cout_{-1}$



6*t*

Retenue anticipée par blocs

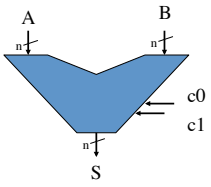
2(n+1)t pour retenue sur 4^n bits
Temps logarithmique et non linéaire



L'UAL

- Addition, soustraction, opérations logiques bit à bit
- Implique entrée de sélection : commande

c1	c0	S
0	0	A AND B
0	1	A OR B
1	0	A + B
1	1	A - B



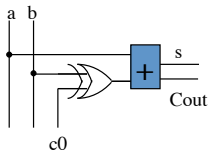
Une tranche 1 bit de l'UAL

- Soustraction = ajouter l'opposé = complément bit à bit + 1
- Choix : sur c0
  - Si c0 = 0, ajouter b
  - Si c0 = 1, ajouter Not(b)

  Mais

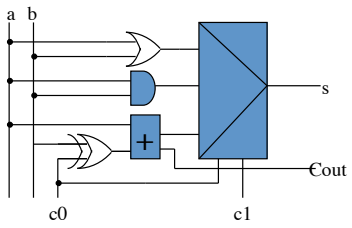
  a XOR 0 = a

  A XOR 1 = !a



Une tranche 1 bit de l'UAL

Version naïve, multiplexeurs et propagation de retenue

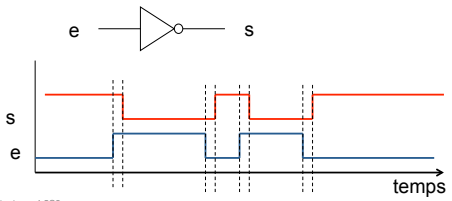


Plan

- Algèbre de Boole et fonctions booléennes
- Circuits combinatoires
- Circuits séquentiels
  - Bascules
  - Compteurs
  - Automates

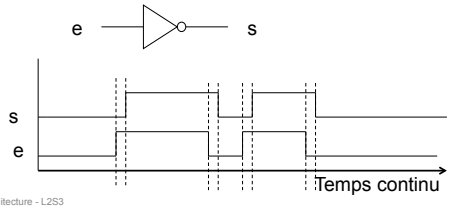
Comportement temporel des circuits combinatoires

Les entrées suivent les sorties - avec un temps de retard



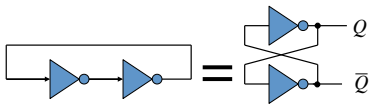
Objectif des circuits séquentiels

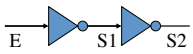
- Mémorisation en circuiterie logique
- Définir les valeurs observables dans les circuits en discrétisant le temps

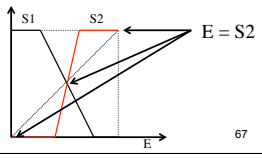


Le Bistable

2 points fixes stables, 1 point instable. Pas d'entrées, irréversible.

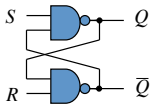


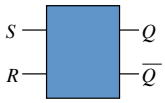




La bascule RS

Positionnement ou mémorisation



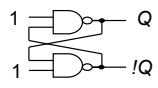


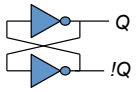
Ecriture d'un 1 : $S = 0$, $R = 1$
Ecriture d'un 0 : $S = 1$, $R = 0$
$R = S = 1$: Mémorisation de la valeur courante

La bascule RS

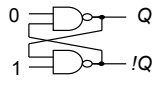
Nand (x, 1) = !x Nand (x, 0) = 1

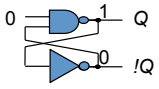
• $R = S = 1$: Mémorisation de la valeur courante





• $R = 1$ $S = 0$: $Q = 1$ et $!Q = 0$ Écriture d'un 1

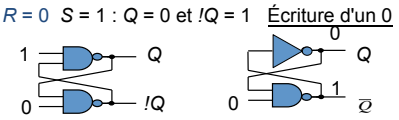




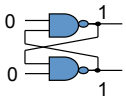
La bascule RS

Nand (x, 1) = !x Nand (x, 0) = 1

• *R* = 0 *S* = 1 : *Q* = 0 et *!Q* = 1 Écriture d'un 0

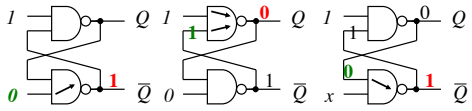


• R = 0 S = 0 : configuration interdite, les deux sorties ne sont plus complémentaires 1, 1



La bascule RS

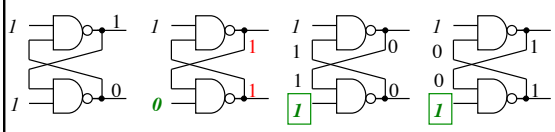
• Ecriture d'un 0 : détail du fonctionnement



A la fin de la deuxième étape, *Q* à 0 verrouille le 1 sur !Q
R peut changer ensuite : ***R*** doit être maintenu à 0 au moins $2t_p$
De même, pour l'écriture d'un 1, ***S*** doit être maintenu à 0 au moins $2t_p$

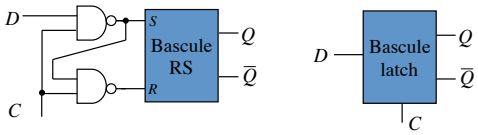
La bascule RS

• Ecriture d'un 0 : fonctionnement pathologique si non -respect des tp



Etat initial S à 0 Après 1 tp S repasse à 1 prématurément Etat final incohérent

La bascule latch

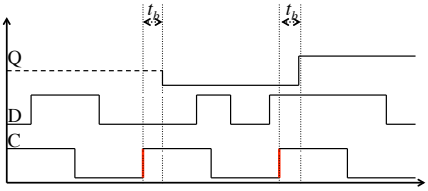


- *C* = 0 : *S* = *R* = 1, mémorisation ; état opaque
- *C* = 1 : *S* = !D , *R* = *D*, donc *Q* = *D* ; état transparent

La bascule D

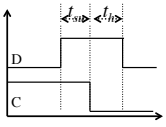
Bascule sur front

- Enregistre l'état de D sur la transition montante
- (0->1) de C. L'état est recopié sur Q après t_b.



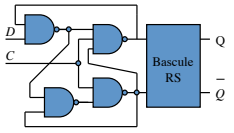
La bascule D

Contraintes : temps d'établissement et de maintien

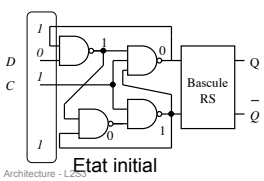


Réalisation de la bascule D

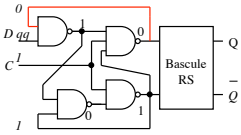
- Pour *C* = 0, la bascule RS est en mémorisation



- Ecriture d'un 0



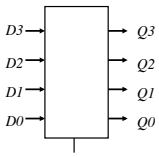
Etat initial

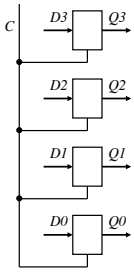


Verrouillage

Registre

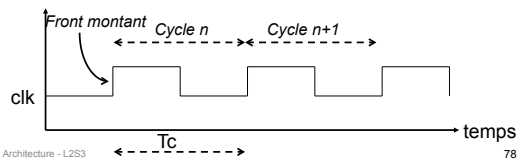
Un ensemble de bascules commandées par le même signal : mémorisent simultanément *n* bits au lieu de 1.





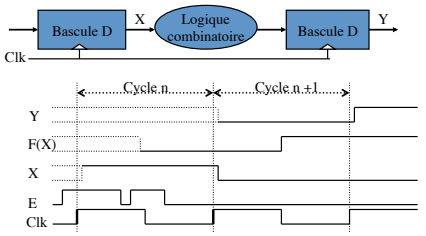
Horloge

- Signal périodique. Période = Temps de cycle T_c
  Fréquence $F = 1/T_c$
- Fournit une référence de temps discret commune à un ensemble de circuits



Opérateurs synchrones

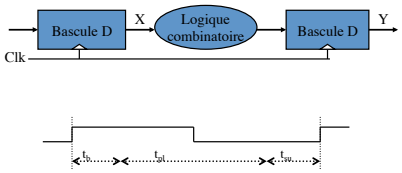
$$Y_{cycle\ n+1} = G(X_{cycle\ n})$$



Fréquence de fonctionnement

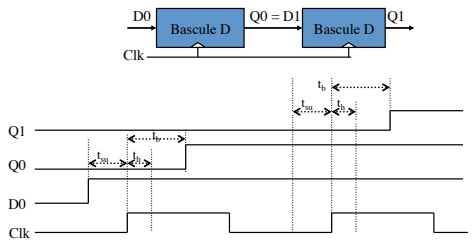
$$T_c \geq t_b + t_{pl} + t_{su} = T_{c\min}$$

$$F_{\max} = \frac{1}{T_{c\min}}$$



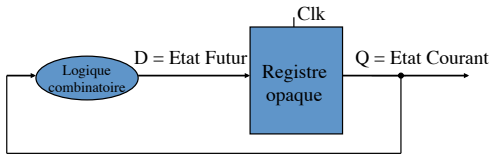
Décaleur

$$Q1_{cycle\ n+1} = Q0_{cycle\ n}$$



Compteurs synchrones

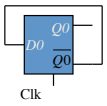
$X_{i+1} = (1+X_i) \bmod N$



Réalisation = définir les fonctions de la logique combinatoire

Compteurs

- Par 2

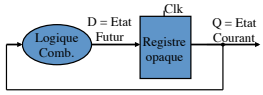


$D0 = \overline{Q0}$

- Par 4

En code entier naturel

Etat Courant			Etat Futur		
Val	Q1	Q0	D1	D0	Val
0	0	0	0	1	1
1	0	1	1	0	2
2	1	0	1	1	3
3	1	1	0	0	0



$D0 = \overline{Q0}$

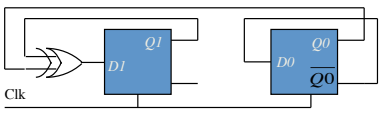
$D1 = Q1.\overline{Q0} + \overline{Q1}.Q0 = Q1 \oplus Q0$

Compteur par 4 synchrone

$D0 = \overline{Q0}$

$D1 = Q1.\overline{Q0} + \overline{Q1}.Q0 = Q1 \oplus Q0$

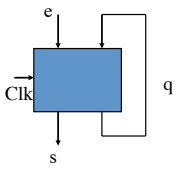
En utilisant les deux sorties des bascules D, on récupère directement les sorties complémentées.



Automates synchrones

$(q_{n+1}, s) = f(q_n, e)$

- q : état de l'automate
- e : entrées
- s : sorties - commandes

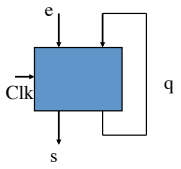


Un compteur est un automate avec e absent, s = q

Application des automates synchrones

$(q_{n+1}, s) = f(q_n, e)$

- Circuits : de l'automate de machine à laver au contrôle de microprocesseurs
- Théorie et pratique des langages

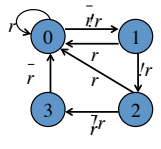


Automate de Moore

Un automate de Moore est un sextuplet
{Q, q0, T, E, S, f}

Q ensemble des états
q0 état initial
E ensemble des entrées
S ensemble des sorties
T transitions, inclus dans QxExQ
f :Q -> S = calcul des sorties

Exemple : compteur par 4 avec RAZ

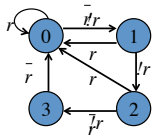


f (q) = q

Automate de Moore

Les sorties sont déterminées par l'état

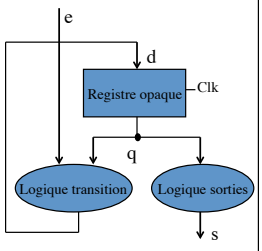
Exemple : compteur par 4 avec RAZ



f (q) = q

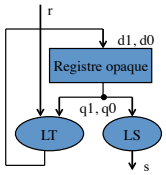
Réalisation d'un automate de Moore

- Etat : registre opaque
- Transitions et sortie : logique combinatoire
- Réaliser un automate :
  - Taille du registre d'états
  - Synthèse des fonctions booléennes transition et sortie : PLA



Compteur avec RAZ

Etat courant				Etat futur		
N	q1	q0	r	d1	d0	N
0	0	0	0	0	1	1
0	0	0	1	0	0	0
1	0	1	0	1	0	2
1	0	1	1	0	0	0
2	1	0	0	1	1	3
2	1	0	1	0	0	0
3	1	1	0	0	0	0
3	1	1	1	0	0	0



$$d_0 = \overline{q_0}.\bar{r}$$

$$d_1 = \overline{q_1}.q_0.\bar{r} + q_1.\overline{q_0}.\bar{r}$$

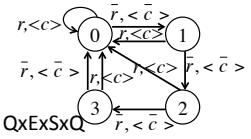
$$s_1 = q_1, \quad s_0 = q_0$$

Automate de Mealy

Un automate de Mealy est un quintuplet {Q, q0, T, E, S}

- Q ensemble des états
- q0 état initial
- E ensemble des entrées
- S ensemble des sorties
- T transitions, inclus dans QxExSxQ

Exemple : compteur par 4 avec RAZ et signal

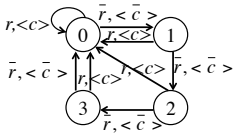


Automate de Mealy

T transitions, inclus dans QxExSxQ

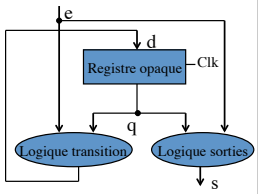
Les sorties sont déterminées par la transition

Exemple : compteur par 4 avec RAZ et signal

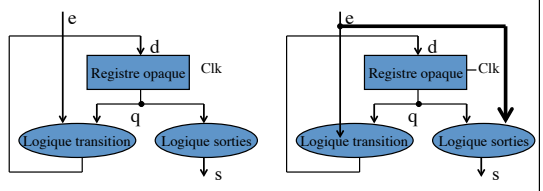


Réalisation d'un automate de Mealy

- Etat : registre opaque
- Transitions et sortie : logique combinatoire
- Réaliser un automate :
  - Taille du registre d'états
  - Synthèse des fonctions booléennes transition et sortie : PLA



Comparaison Moore-Mealy



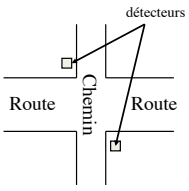
Comparaison Moore-Mealy

- Les deux modèles sont équivalents : étant donné un automate de Mealy (resp Moore), on peut toujours construire un automate de Moore (resp Mealy) qui a la même séquence de sorties pour une séquence d'entrées données.
- Mais l'automate de Moore a toujours au moins autant (et souvent beaucoup plus) d'états que l'automate de Mealy équivalent.

Exemple : contrôle de croisement

Contraintes :
- Sûr et non bloquant chemin
- Rouge Route < Temps Long
- Au moins Temps Long entre détection et Route Rouge
- Orange pendant Temps Court (exact)

Signaux :
- a = 1 : détection (donné)
- tl/tc = 1 : Temps Long/Court écoulé
- Contrôle des feux



Exemple : contrôle de croisement

Sans temporisation

RV : route vert, chemin rouge

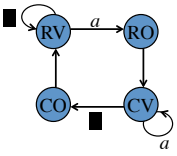
RO : route orange, chemin rouge

CV : route rouge, chemin vert

CO : route rouge, chemin orange

Seuls états possible pour la sécurité

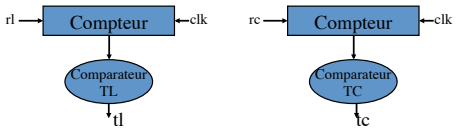
Moore : la valeur des feux est associée à l'état



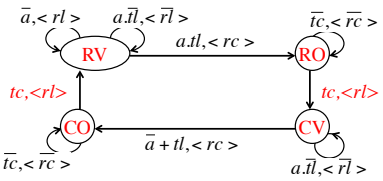
Exemple : contrôle de croisement

Avec temporisation :

Aux instants adéquats, RAZ d'un compteur



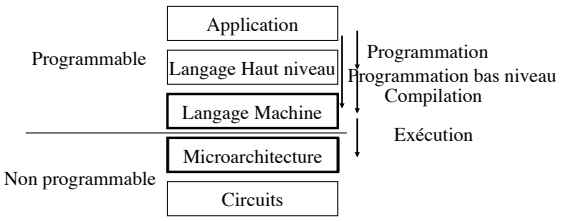
Exemple : contrôle de croisement



4. Micro-architecture

Objectif : Comment réaliser un sous-ensemble représentatif de l'architecture logicielle MIPS32

Les niveaux d'un système informatique



Organisation générale

RI <- Mem[PC] et PC <- PC + 4

Exécution de l'instruction

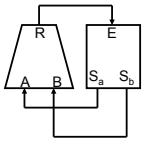
- Partie opérative : transferts registre-registre, registre-mémoire, calculs.
- Partie contrôle : séquencement des actions élémentaires

Instructions arithmétiques et logiques

Format R-R

Spécification : Rd <- Ra OP Rb

Réalisation: *Rd <- Ra OP Rb*

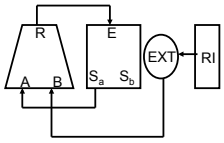


$T_c \geq T_{reg} + T_{UAL}$

Format R-I

Spécification : Rd <- Ra OP ES(Imm_{16})
ou Rd <- Ra OP EZ(Imm_{16})

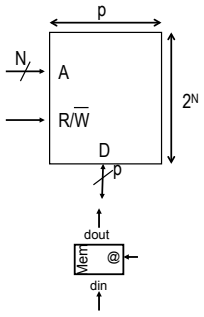
Réalisation: *Rd <- Ra OP EXT($RI_{15:0}$)*



$T_c \geq T_{reg} + T_{UAL}$

Mémoire

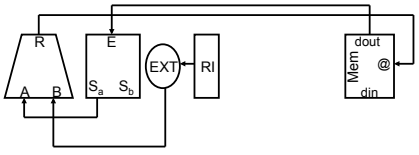
- Un tableau linéaire (1 dimension)
  - N bits d'adresse
  - p bits de données
  - 2^N entités de P bits sont adressables
- une entité = un mot mémoire
- Adressable = lue/écrite individuellement
- Lecture et Ecriture sont mutullement exclusives
- Pour ce chapitre, $T_{mem} = T_{UAL}$ mais décrit en fait un cache



Instruction de chargement - LOAD

Spécification : Rd <- Mem [Ra +ES(Imm_{16})]

Réalisation 1 : Rd <- Mem [Ra +EXT(RI)]



$T_c \geq T_{reg} + T_{UAL} + T_{Mem}$

Temps de cycle et temps d'exécution

- T_{ex} = temps d'exécution d'un programme comportant NI instructions
  - dépend de l'architecture logicielle
- CPI= nombre de cycles par instruction
  - dépend de l'architecture logicielle et de la micro-architecture
- T_c = temps de cycle
  - dépend du matériel et de l'implémentation

$$T_{ex} = NI \times CPI \times T_c$$

Instruction de chargement - LW

Spécification : Rd <- Mem [Ra +ES(Imm_{16})]

Réalisation 1 : Rd <- Mem [Ra +EXT(RI)]

$$T_c \geq T_{reg} + T_{UAL} + T_{Mem}$$

Le temps de cycle est doublé pour **TOUTES** les instructions.

✓ Il faut chercher une autre réalisation.

Instruction de chargement - LW

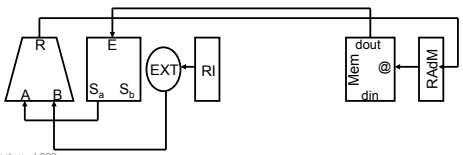
Spécification : Rd <- Mem [Ra +ES(Imm_{16})]

Réalisation 2 :

- *Cycle 1 : RAdM <- Ra +EXT(RI)*
- *Cycle 2 : Rd <- Mem[RAdM]*

$T_c \geq T_{reg} + T_{UAL}$

✓ *Les instructions de chargement prennent 2 cycles, et les instructions arithmétiques 1 cycle*



Instruction de rangement - SW

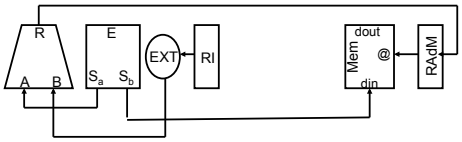
Spécification : Rd <- Mem [Ra +ES(Imm_{16})]

Réalisation 2 :

- *Cycle 1 : RAdM <- Ra +EXT(RI)*
- *Cycle 2 : Mem[RAdM] <-Rd*

$T_c \geq T_{reg} + T_{UAL}$

✓ *Les instructions de rangementprennent 2 cycles, et les instructions arithmétiques 1 cycle*



Instructions de branchement

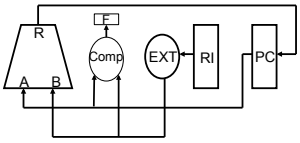
Spécification : PC <- PC + ES(dep_{16}||0x00) si Rs cond Rd

Réalisation :

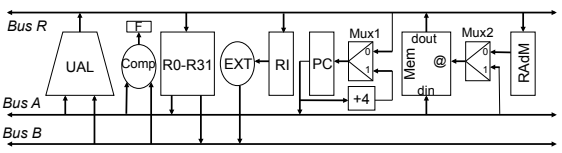
- Cycle 1 : F <- Rs cond Rd
- Cycle 2 : Si F = 1 alors PC <- PC OP EXT($RI_{15:0}$)

$T_c \geq T_{reg} + T_{UAL}$

Condition : voir partie contrôle



Architecture 3 bus

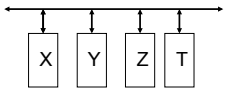


Les bus (1)

- Bus = ensemble de 32 « fils » interne au micro-processeur
- Pas de protocole : très différent d'un bus de carte (SCSI, USB,…)
- Pas de mémorisation
- **Contrainte : UNE SEULE information à la fois sur un bus**

Les bus (2)

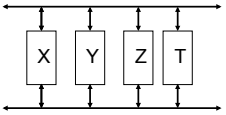
Contrainte : UNE SEULE information à la fois sur un bus



- Copier X dans Y et Z : 1 cycle
  Z <- X; Z <- Y
- Copier X dans Y et Z dans T : 2 cycles
  Cycle 1 : Y <- X
  Cycle 2 : T <- Z

Les bus (3)

Contrainte : UNE SEULE information à la fois sur un bus



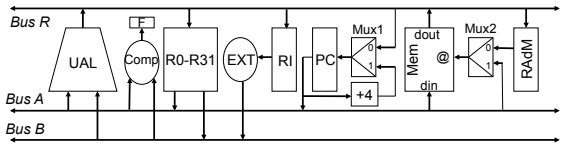
- Copier X dans Y et Z dans T : 1 cycle
  Y <- X ; T <- Z

Langage transfert

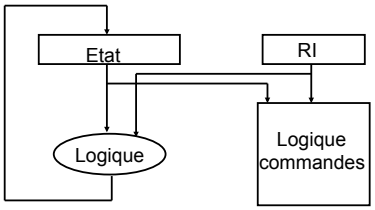
- Description de la partie opérative
- **Contraintes :**
  - **UNE SEULE information à la fois sur un bus**
  - **Par des connexions existantes**
  - **Entre éléments de mémorisation : Registre-registre ou registre mémoire**
- Syntaxe : une ligne par cycle – transferts simultanés séparés par « ; »
- On peut vérifier que la carte machine 3 bus permet la description en langage transfert correspondant aux textes « réalisation »

La lecture de l'instruction

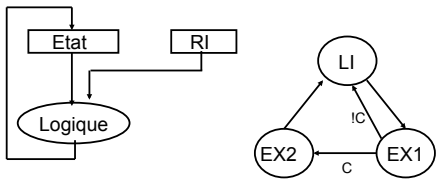
Spécification : RI <- Mem[PC] ; PC <- PC +4



Partie Contrôle



Automate d'états

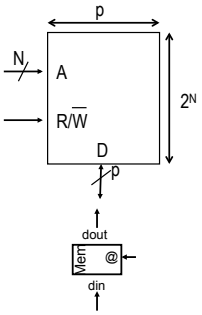


C = Instruction arithmétique et logique
ou branchement

5. Mémoire

Mémoire

- Un tableau linéaire (1 dimension)
  - N bits d'adresse
  - p bits de données
  - 2^N entités de P bits sont adressables
- une entité = un mot mémoire
- Adressable = lue/écrite individuellement
- Lecture et Ecriture sont mutullement exclusives
- Pour ce chapitre, $T_{mem} = T_{UAL}$ mais décrit en fait un cache



Mémoire statique - SRAM

- Technologie semiconducteurs logique
- 6 T par point mémoire
  Un bistable plus deux interrupteurs
- Evoluent en fréquence comme les processeurs

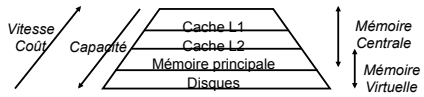
Mémoire dynamique DRAM

- Technologie non CMOS
- Equivalent 1,5 transistors
- Résistance + capacité
- Demande un rafraîchissement périodique (réécriture)
- La lecture est destructrice -> Temps de cycle
- Le temps d'accès évolue moins vite que celui des processeurs

Faits technologiques (1)

- Gap processeur mémoire
  1 calcul = 3 opérandes
- Loi de Moore: densité d'intégration X 2 / 18 mois
  - Joue à plein pour les SRAM
  - Effet plus complexe sur les processeurs
- Performance des processeurs x 1,5 / an
- Débit DRAM x 1,2 / an
- Débit disques x 1,6 / an
- Les meilleures SRAM
  temps d'accès = Tc processeur
- 1 ordre de grandeur SRAM -> DRAM
- 3 ordres de grandeur DRAM -> disques

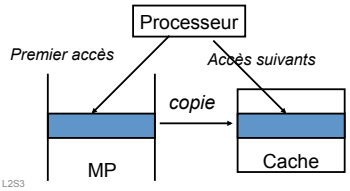
Hiérarchie mémoire



- A un instant donné, chaque niveau de la hiérarchie contient un extrait du niveau inférieur
- L'accès étant plus rapide, la copie offre l'illusion d'une mémoire + rapide à faible coût
- La gestion des copies et problèmes reliés doit être transparente au niveau de l'exécution des instructions
  - Cache : entièrement en matériel
  - Mémoire virtuelle : matériel et logiciel système

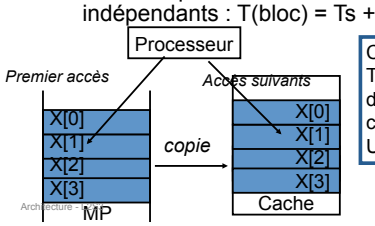
Le principe de localité

- Localité temporelle = Accès répétés à une même adresse
  - Boucles dans le code
  - Eventuellement en données



Le principe de localité

- Localité spatiale
  - Accès à des adresses consécutives
  - Le coût d'un accès bloc est beaucoup moindre que le coût de n accès indépendants : T(bloc) = Ts + n Ti



Conséquence
Transferts blocs d'adresses consécutives
Unité Cache : *Ligne*

Le produit Matrice-Vecteur

Conséquences pour le programmeur

- Attention aux conventions des compilateurs
- Les accès par indirection peuvent coûter TRES cher

```
pour i= 1,N
   ... A[i] ...
fin pour
```
Bon

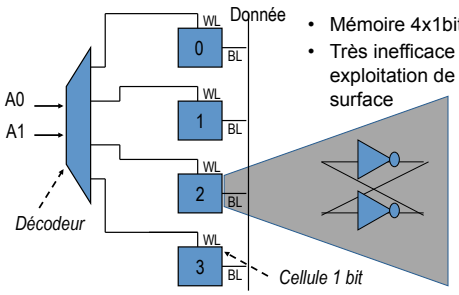
```
pour i= 1,N
   ...
    A[L[i]] ...
fin pour
```
Mauvais

Conséquences pour le programmeur

Des gains de performances importants peuvent être obtenus en localisant les accès

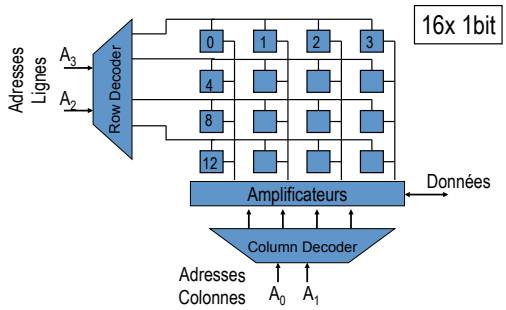
- Il existe des outils d'analyse des performances processeur et mémoire
  - VTune pour Windows
  - Associés à des compilateurs commerciaux sous Linux/Unixes
  - Quelques logiciels libres produits de recherche
- L'optimisation automatique est un sujet de recherche

Organisation interne : 1D



- Mémoire 4x1bit
- Très inefficace en exploitation de la surface

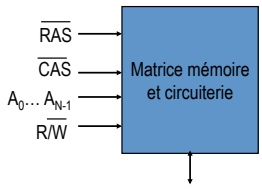
Organisation interne matricielle



Organisation matricielle

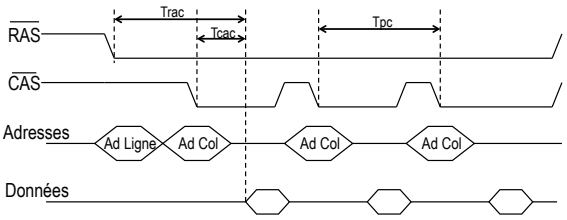
- L'organisation matricielle concerne une mémoire 2^{2m}x1 bit
- Pour une mémoire 2^{2m}xp bits, la structure est répliquée

Organisation des DRAM



- !RAS et !CAS valident respectivement les parties hautes et basses des adresses

Chronogramme en lecture



Organisation matricielle + validation

- Les données d'une même ligne sont présentées simultanément, puis sélectionnées
- Possibilité d'accès consécutifs aux données d'une *même* ligne *plus rapides* qu'à ceux de lignes distinctes
  - Localité spatiale

```
pour i= 1,N
   ... A[i] ...
fin pour
```
Bon

```
pour i= 1,N
   ...
    A[L[i]] ...
fin pour
```
Mauvais

6. Introduction aux architectures avancées : pipeline

Améliorer les performances

Micro-architecture élémentaire

- 2 cycles pour les instructions arithmétiques
- 3 cycles pour les instructions mémoire

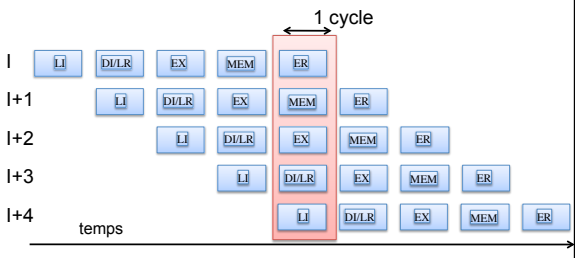
Question : en supposant 20% d'instructions mémoire, quel est le nombre moyen d'instructions par cycle ?

L'exécution d'une instruction

- Les étapes fondamentales

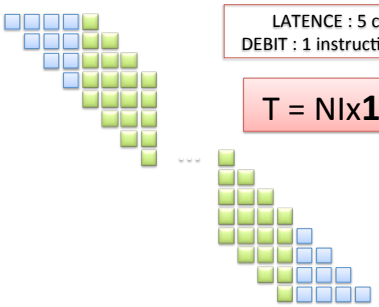
Instructions UAL	*Instructions Mémoire*	*Instructions Branchement*
Lecture instruction	Lecture instruction	Lecture instruction
Incrémentation CP	Incrémentation CP	Incrémentation CP
Décodage de l'instruction	Décodage de l'instruction	Décodage de l'instruction
Lecture des opérandes	Calcul de l'adresse mémoire	Calcul de l'adresse de branchement
Exécution	Accès mémoire	Exécution
Ecriture du résultat	Rangement du résultat	

Principe du pipeline



- Réaliser simultanément les étapes de plusieurs instructions

Performance du pipeline simple

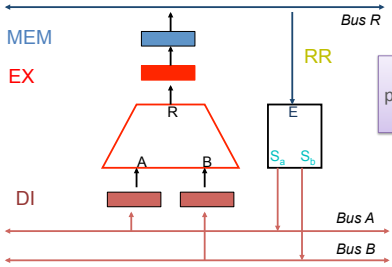


LATENCE : 5 cycles
DEBIT : 1 instruction/cycle

$T = NIx\mathbf{1}xTc$

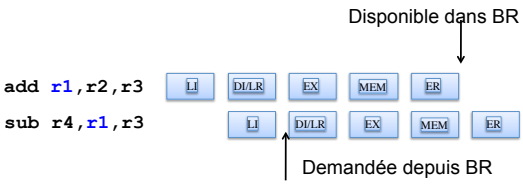
Micro-architecture

Exemple pour les instructions arithmétiques et logiques



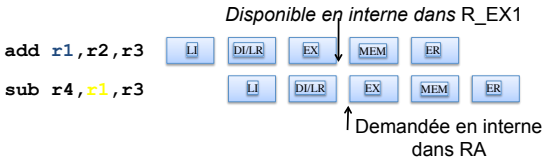
Aussi : un registre par étape pour CP et RI

Aléas de données

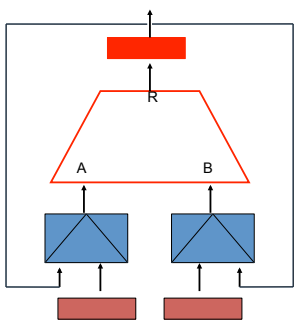


- Exemple de *Dépendance de Données*
  Flot, Read after Write, Producteur-consommateur
- Autres domaines
  Compilation, Exécution parallèle

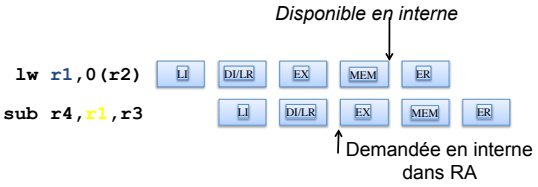
Traitement : *anticipation*



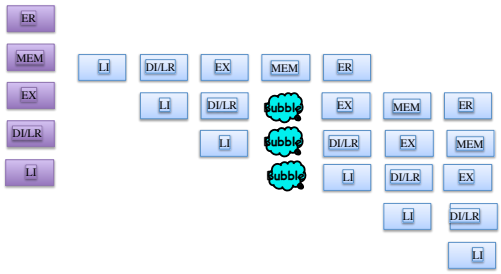
Réalisation de l'anticipation



Aléas de données incontournables



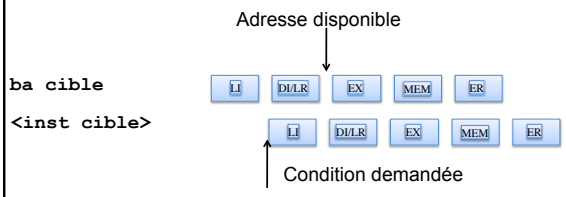
Supension (ou bulle)



Supension

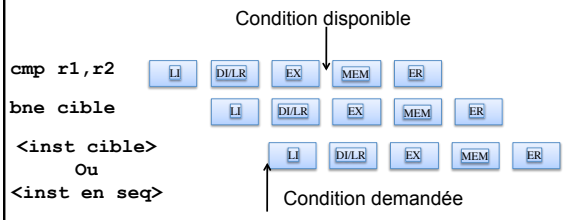
- Question : si 20% des instructions amènent une suspension dans le pipeline, sans confusion des suspensions, quel est le nombre moyen d'instructions par cycle ?

Aléa de branchement



- *Dépendance de contrôle*
- Demande un additionneur supplémenatire

Branchement conditionnel



- *Dépendance de contrôle*

Traitement des aléas de branchement

- Annulation par matériel de l'instruction qui suit.
  - Toute instruction de contrôle prend 2 cycles.
- Insérer une instruction NOP
  - Toute instruction de contrôle prend 2 cycles
- Saut/branchement retardé d'un cycle
  - L'instruction après le branchement est exécutée avant que le branchement ou le saut soit effectué
  - Si le compilateur peut réordonnancer les instructions, saut et branchement en 1 cycle

Exemple de réordonnancement pour branchement retardé

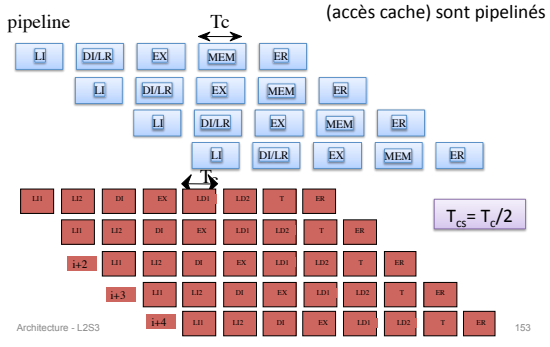
```
        r10 <- @X                    r10 <- @X
        r11 <-@Y                     r11 <-@Y
        r12 <-@Z                     r12 <-@Z
Deb:    Load r1, 0(r10)      Deb:    Load r1, 0(r10)
        Load r2, 0(r11)              Load r2, 0(r11)
        Add r3, r1, r2               Add r3, r1, r2
        Store r3, 0(r12)             Store r3, 0(r12)
        Add r10, r10, 4              Add r10, r10, 4
        Add r11, r11, 4              Cmp r10, r4
        Cmp r10, r4                  Ble Deb
        Ble Deb                      Add r11, r11, 4
```


Aléas de branchement

- Les délais de branchement ne peuvent plus être traités par optimisation logicielles
- Support matériel
  - Prédiction de branchement
  - Cache des adresses des cibles de branchement (BTB)
- Suppression de la plupart des branchements
  - Instructions prédiquées, ex. ARM

Approche superpipeline

Les étages critiques (accès cache) sont pipelinés



Limites des superpipelines

- Fréquence d'horloge plus élevée : dissipation
- Circuits d'anticipation plus complexes
- Délais de branchement et chargement plus élevés

Pour aller plus loin

- Pipeline pour instructions flottantes
  - Exécution dynamique "dans le désordre"
  - Optimisations logicielles : déroulage de boucle et pipeline logiciel
- Intégration pipeline et caches
- Superscalaires
- GPU