

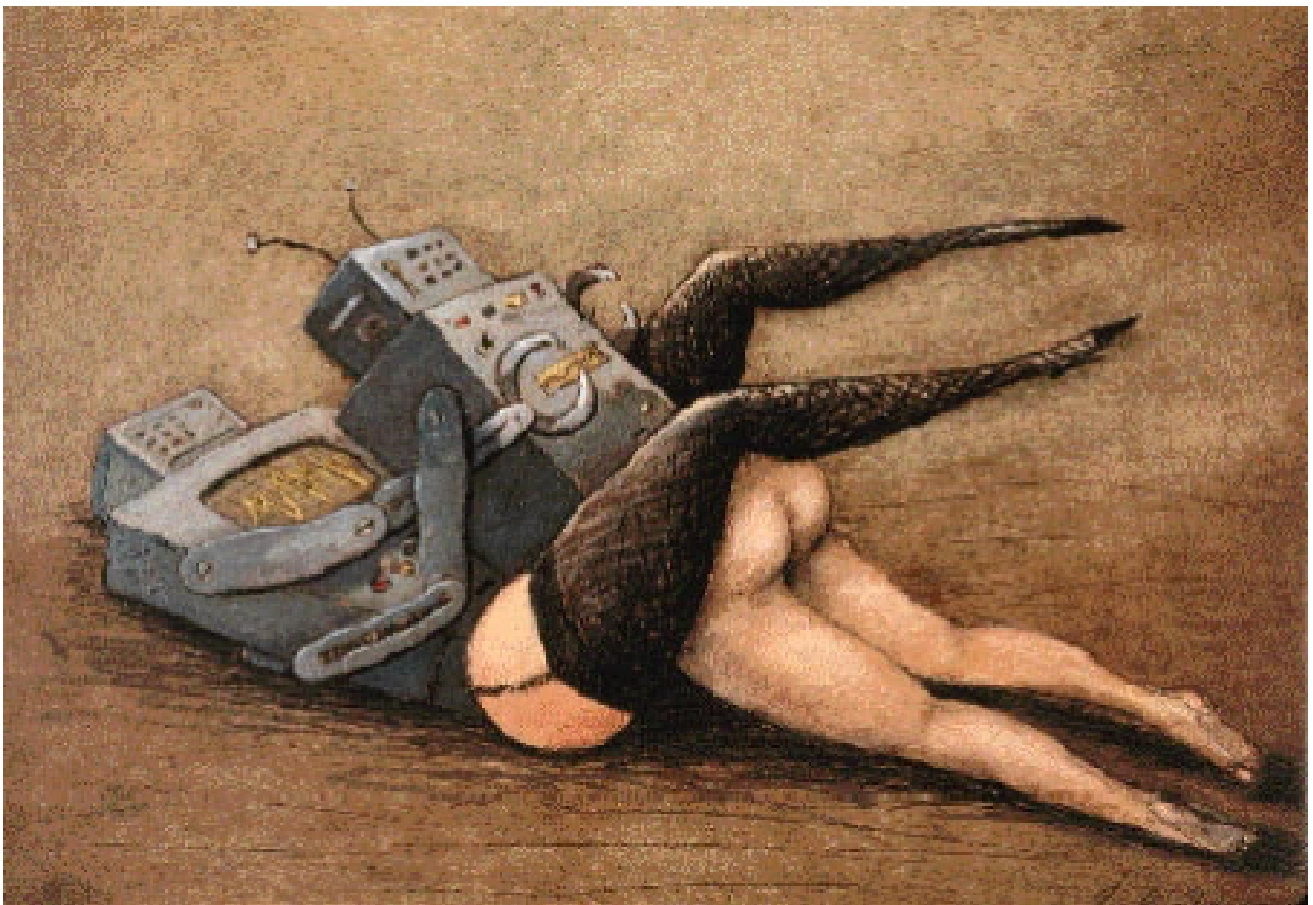
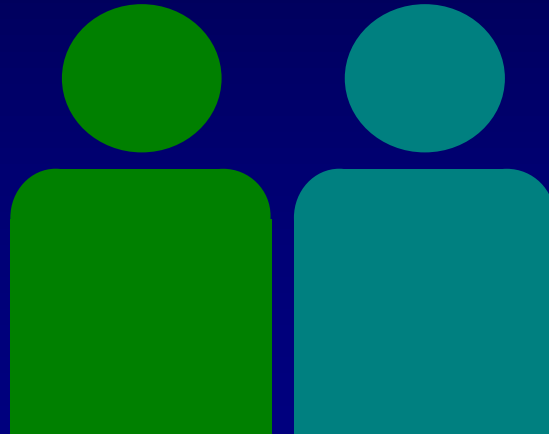
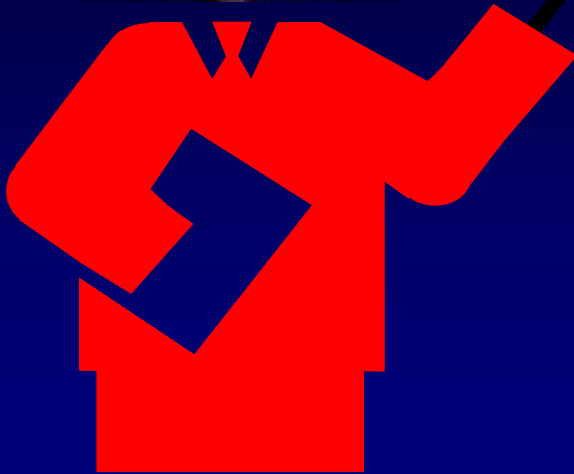


Vida Artificial

Eduardo Sanchez



Ecole Polytechnique Fédérale de Lausanne



Vida artificial

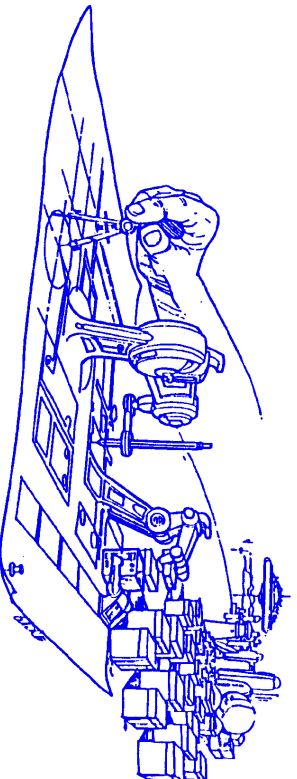
- Estudio de los principios fundamentales que gobiernan los fenómenos biológicos para aplicarlos a otros medios físicos
- Software: vida virtual
Wetware: vida alternativa
Hardware: vida sintética
- Biología: enfoque esencialmente analítico
Vida artificial: enfoque esencialmente sintético
- Comenzó en los años 50 con el proyecto de von Neumann de crear una máquina autoreproductora

Introduction

- Dans les années 50 John von Neumann voulait construire une machine capable de s'auto-reproduire
- A la même période Stanislaw Ulam travaillait sur la réalisation par ordinateur des motifs (**patterns**) recursifs: des objets géométriques définis récursivement
- Ulam suggéra à von Neumann de construire un monde abstrait, régi par des règles bien déterminées, pour analyser les principes logiques de l'auto-reproduction: c'est les **automates cellulaires**

Autoreproducción según von Neumann

- La pregunta que quiso responder von Neumann es: Una máquina puede reproducirse?
- El interés era comprender la lógica necesaria para la reproducción
- Creación, con Stanislaw Ulam, de un modelo: los **autómatas celulares**



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



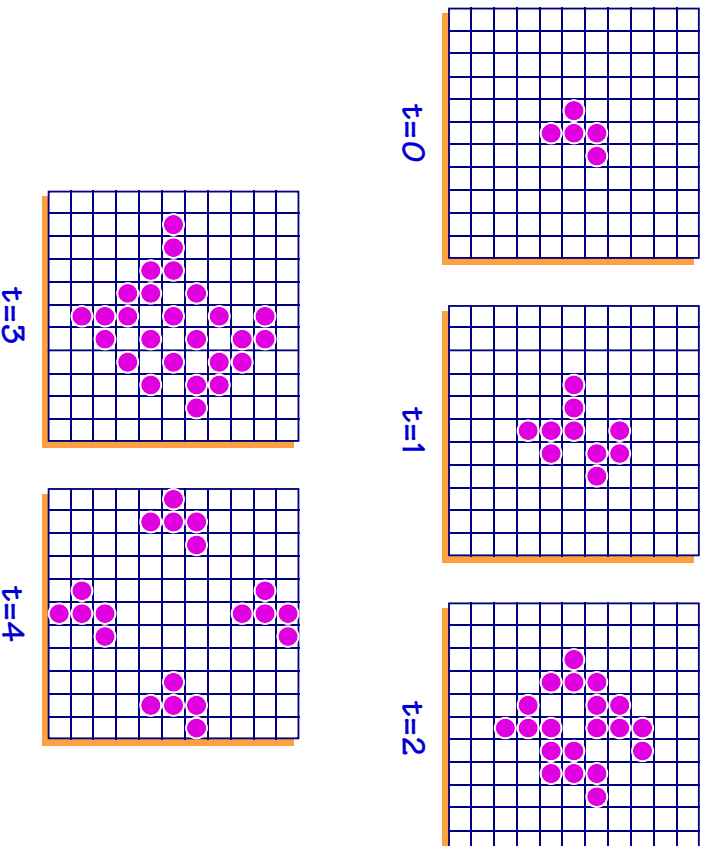
Définition

- Une matrice de cellules identiques
- Chaque cellule est une machine d'états
- L'état suivant d'une cellule dépend seulement de son état présent et de ceux de ses voisines
- Le changement d'état des cellules se fait de façon synchrone
- Problème direct: étant donné la table d'états d'une cellule, déduire le comportement de l'automate
- Problème inverse: étant donné la description de certaines propriétés, trouver la table d'états de la cellule d'un automate possédant ces propriétés (von Neumann devait trouver un automate avec un motif auto-reproducteur)

Auto-reproduction triviale

- Edward Friedkin créa en 1960 un automate cellulaire capable d'auto-reproduire n'importe quel motif de départ
- Les caractéristiques de cet automate sont:
 - ◆ deux états (mort et vivant)
 - ◆ voisinage de von Neumann (4 voisines)
 - ◆ toute cellule avec un nombre pair de voisines sera morte à l'instant suivant
 - ◆ toute cellule avec un nombre impair de voisines sera vivante à l'instant suivant
 - ◆ après 2ⁿ instants d'horloge (où n est une fonction du motif), tout motif initial sera reproduit 4 fois (au nord, au sud, à l'est et à l'ouest). Les 4 copies seront placées à une distance de 2ⁿ cellules du motif initial, disparu

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Le jeu de la vie

- Inventé par John M. Conway (Université de Cambridge)
- Popularisé par Martin Gardner (Scientific American, octobre 1970, février 1971)
- Deux états par cellule: morte et vivante
- Huit voisines

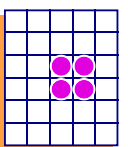
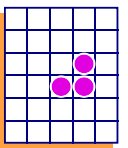
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Juego de la vida

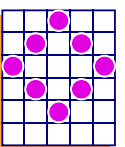
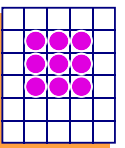
- **Autómata celular inventado en 1970 por John Conway y popularizado por Martin Gardner**

- Nacimiento de una célula



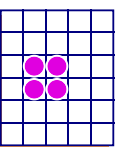
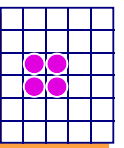
- Tres vecinas

- Muerte de una célula



- Más de tres vecinas
- Menos de dos vecinas

- Supervivencia de una célula

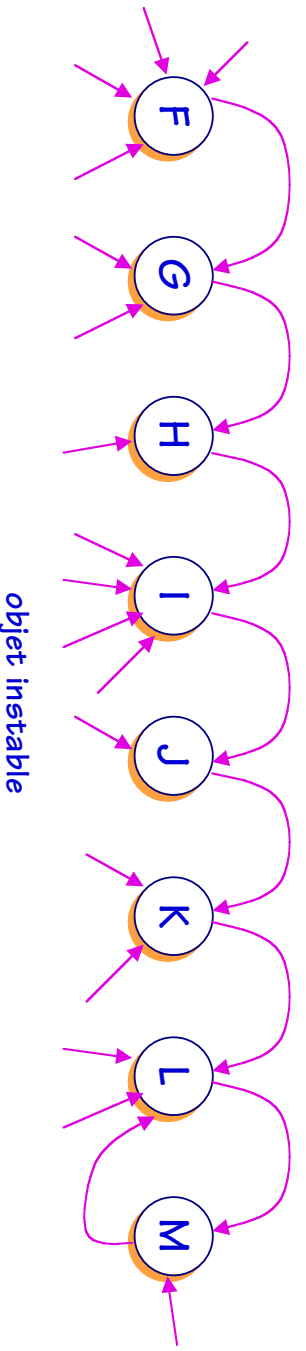
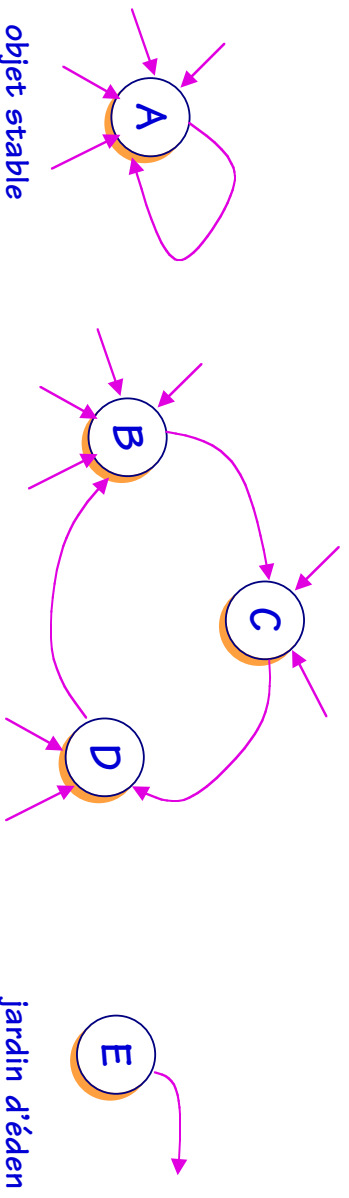


- Dos o tres vecinas

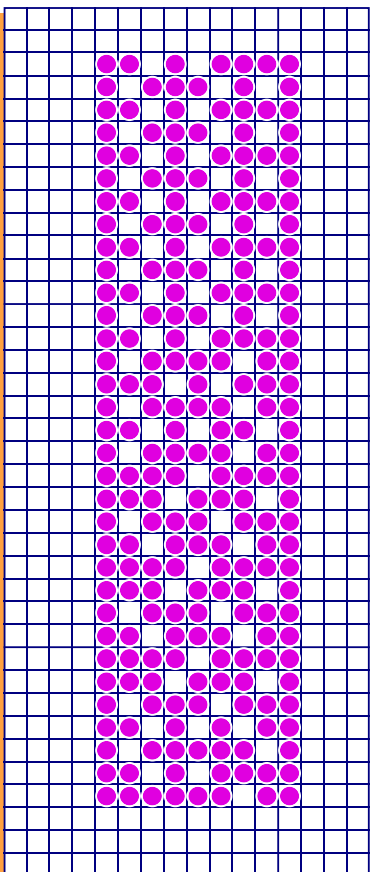
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- Le jeu de la vie est **déterministe en avant**: un motif de départ donné évolue toujours vers un même et unique motif d'arrivée
- Le jeu de la vie n'est pas déterministe en arrière: un motif possède en général plusieurs motifs qui pourraient l'avoir précédé (pour un même motif d'arriver il peut y en avoir plusieurs de départ)
- Le motif d'arrivé d'un certain motif de départ est complètement imprédictible: il n'y a pas de moyen plus simple de le calculer que d'exécuter les transitions pas à pas



- Un **jardin d'éden** est un motif sans passé: un motif instable sans prédécesseurs

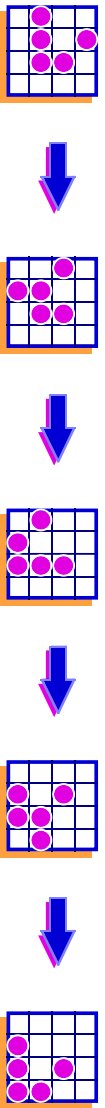


Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Le *glider*

- Déplacement d'un *glider*:

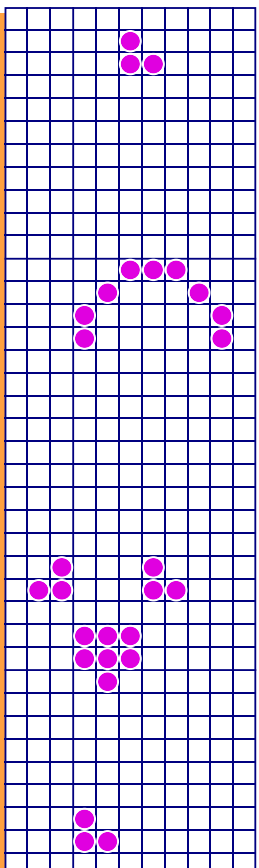


- Les gliders sont des moyens de transmission de l'information dans le jeu de la vie
- La vitesse maximale de croissance d'un motif est d'une cellule par génération: c'est la **vitesse de la lumière** du jeu de la vie. Les gliders avancent à un quart de la vitesse de la lumière: une cellule en diagonale chaque 4 générations. Aucun motif ne peut se déplacer à une vitesse supérieure à la moitié de la vitesse de la lumière

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

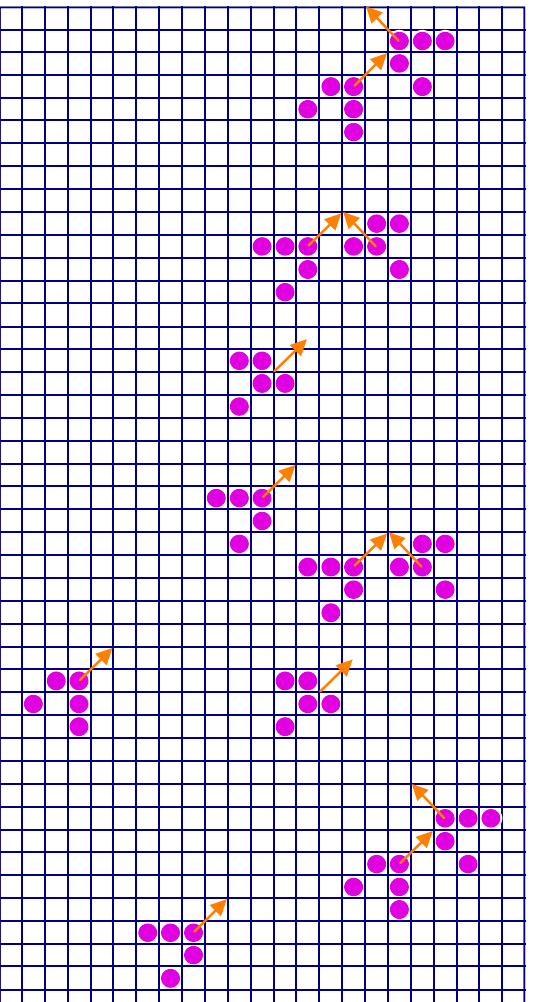


- Pistolet à glider:



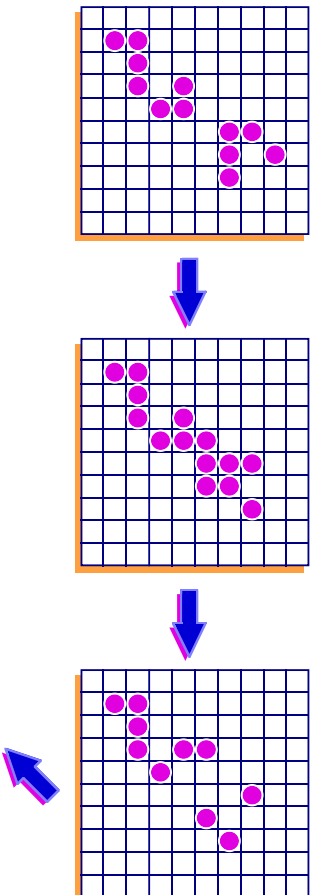
Ce motif est une machine à mouvement perpétuel:
un nouveau glider est produit chaque 30
générations

- Un certain motif, composé de 13 gliders, donne lieu à un pistolet à gliders:



Le pistolet est assemblé 67 instants plus tard et
le premier glider part au temps 92

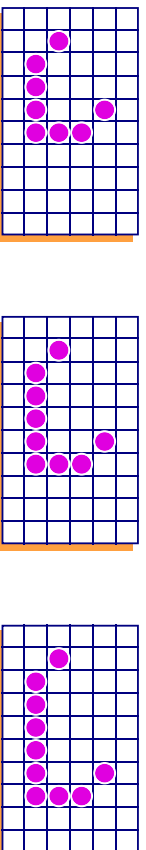
- Un motif, appelé *mangeur de gliders*, présente des caractéristiques d'auto-réparation: il se reconstruit après avoir heurté un *glider*



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- D'autres motifs se déplacent horizontalement: les *vaisseaux*

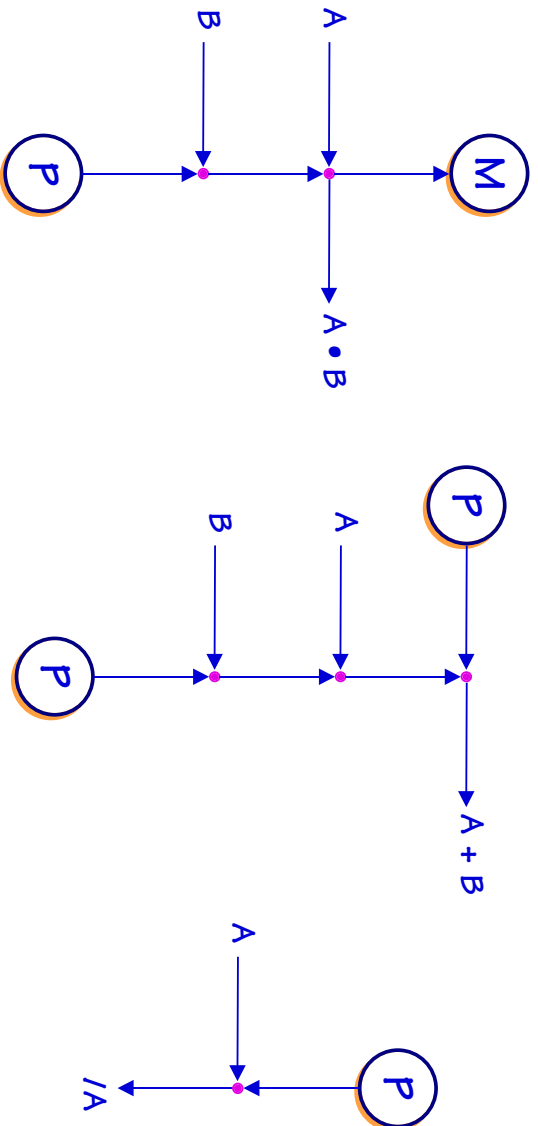


Les déchets générés pendant le mouvement détruisent des vaisseaux plus longs. Toutefois, ils peuvent survivre entourés d'autres vaisseaux, chargés de détruire les déchets: c'est les *flottes*

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Implémentation des fonctions logiques



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



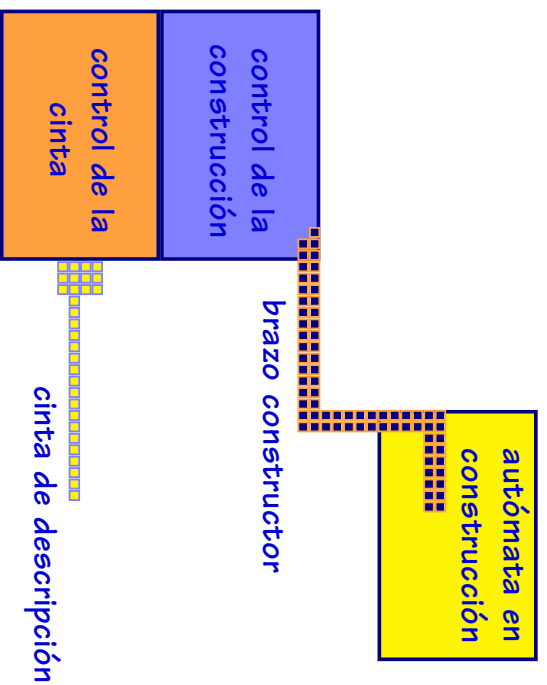
Autómata autoreproductor de von Neumann

- Dada la descripción de una máquina cualquiera, un **constructor universal** es capaz de construirla a partir de las partes disponibles
- Dada su propia descripción, un constructor universal puede construirse a sí mismo (la copia no incluye la descripción)
- Para realizar una auténtica autoreproducción es necesario que la copia contenga su propia descripción
- La descripción es interpretada y copiada: el mismo principio del ADN, descubierto años después

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- 29 estados por célula
- Más de 200'000 células

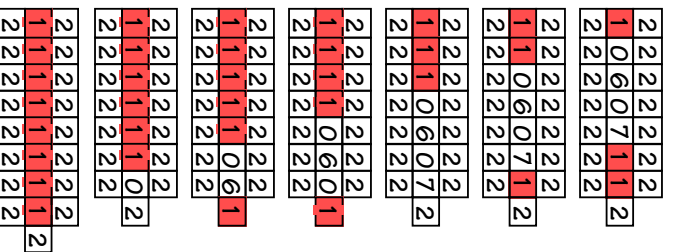


Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

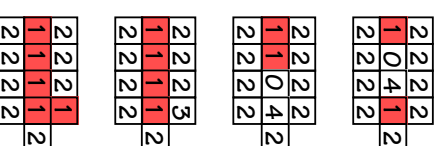


Automate auto-reproducteur de Codd (1968)

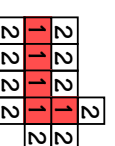
Extension d'un chemin



Tourner à gauche



envoi de 05



envoi de 06

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



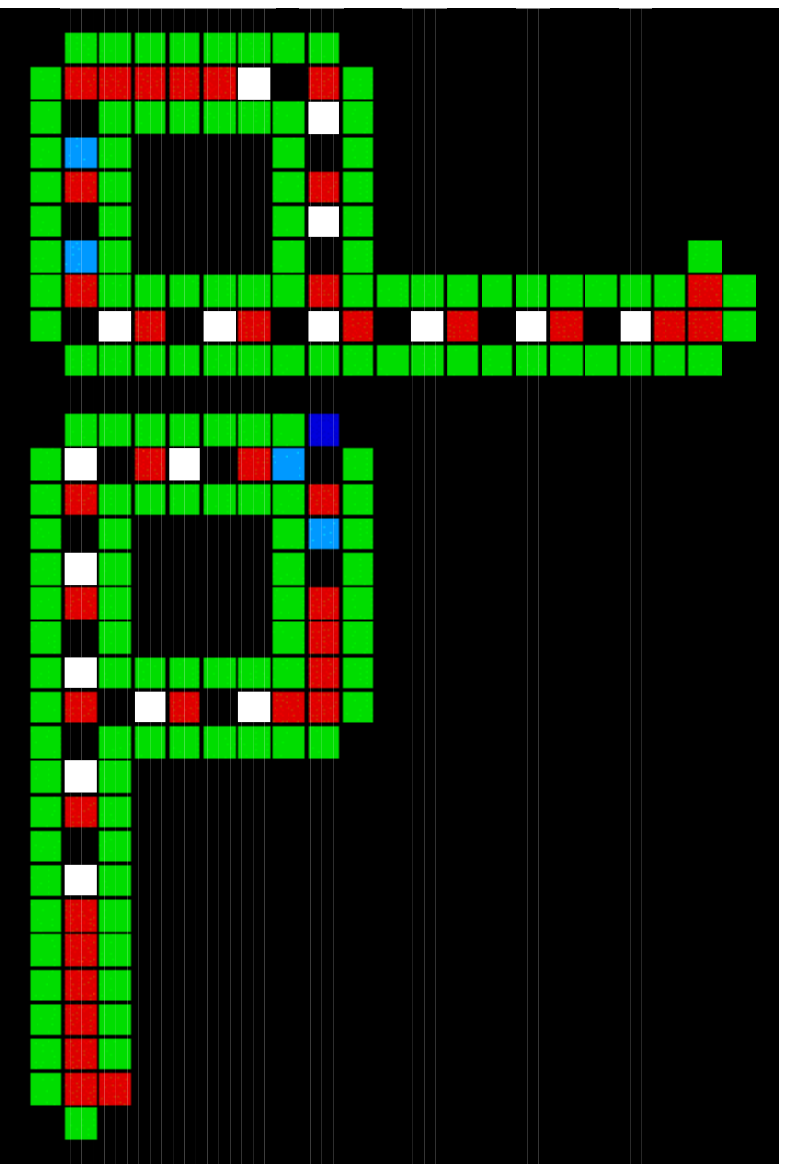
Autómata autoreproductor de Langton (1985)

- Ocho estados por célula
- Cuatro vecinos

■ Tabla de descripción con 219 transiciones

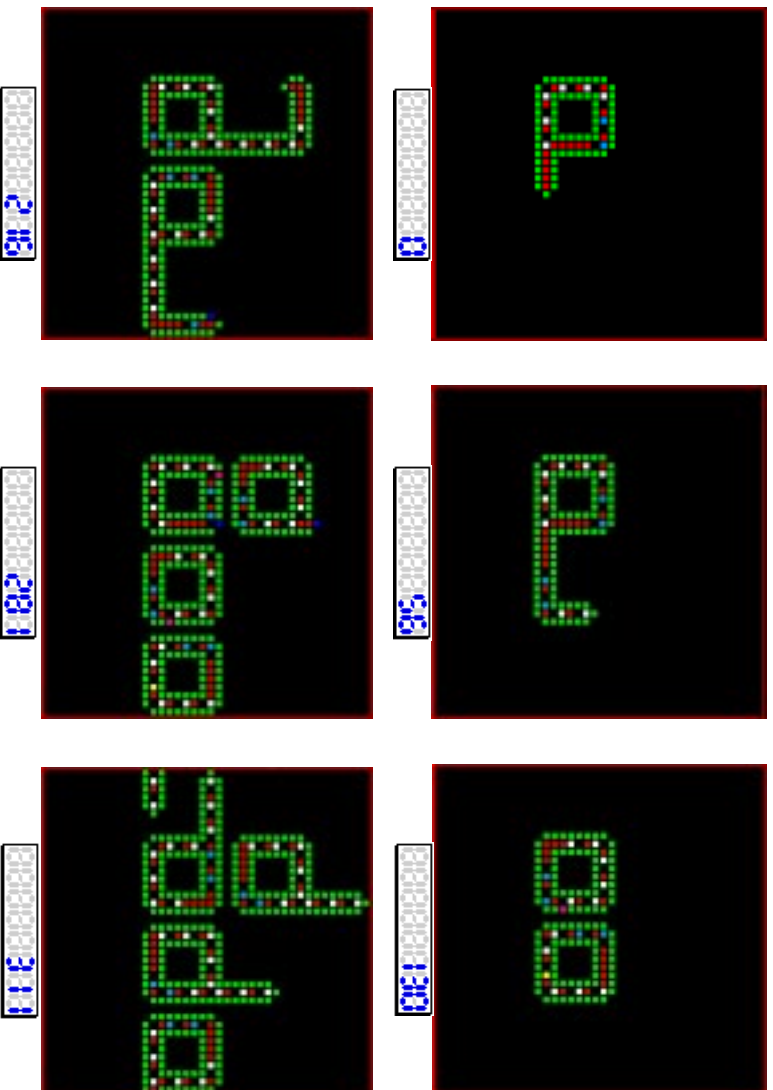
	2	2	2	2	2	2	2											
2	1	7	0	1	4	0	1	4	2									
2	0	2	2	2	2	2	2	0	2									
2	7	2					2	1	2									
2	1	2					2	1	2									
2	0	2					2	1	2									
2	7	2					2	1	2									
2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2	2	2	
2	0	7	1	0	7	1	0	7	1	1	1	1	1	1	1	1	1	2
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



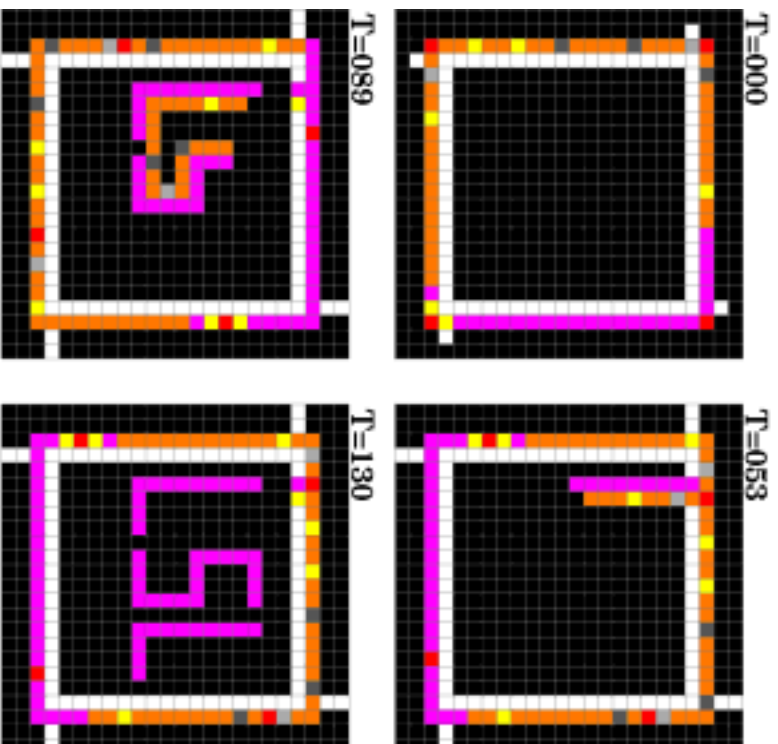


Automate auto-reproducteur de Byl (1989)

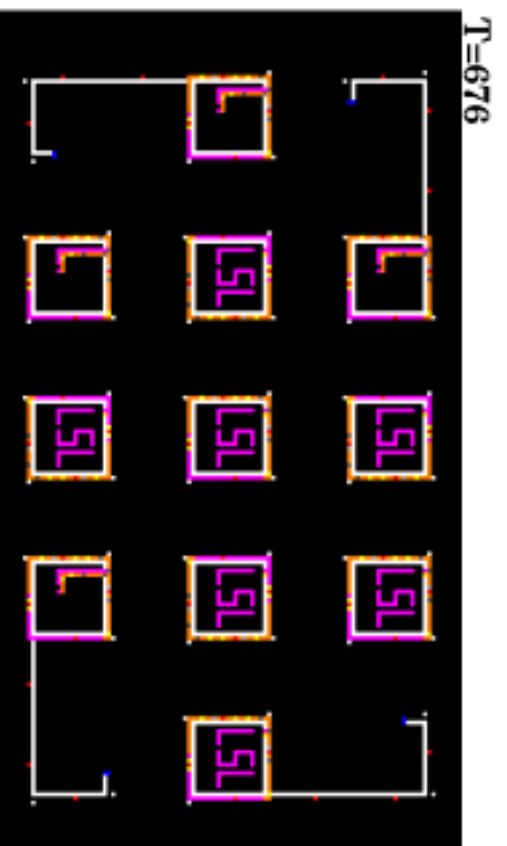
- Six états par cellule
- Motif initial à 11 cellules
- Tableau à 56 transitions

	2	2		
2	3	1		
2	3	4	2	
	2	5		

Automate auto-reproducteur de Tempesti



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Les virus informatiques

- **Virus**: segment de code exécutable, auto-répliquable, placé à l'intérieur d'un programme hôte.
Le virus modifie le programme hôte de façon à prendre le contrôle lorsque l'ordinateur démarre l'exécution du programme hôte. Pendant son exécution, le virus cherche un nouveau hôte dans lequel se répliquer.
- **Worm**: programme indépendant qui, pendant son exécution, essaie d'infecter d'autres ordinateurs interconnectés à son hôte. Similaire aux virus: le programme hôte est l'OS et le code infecté est un processus. Développé à l'origine, chez Xerox, comme une méthode de balance des charges dans un système distribué.
- **Cheval de Troie**: programme contenant du code pour réaliser des fonctions non voulues par l'utilisateur ou non spécifiées dans le mode d'emploi du programme.

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Types de virus

- **Bénin**: tous les programmes infectés sont non pathogènes et non contagieux
- **Cheval de Troie**: aucun programme infecté n'est contagieux mais au moins un est pathogène
- **Propagateur**: aucun programme infecté n'est pathogène mais au moins un est contagieux
- **Méchant**: au moins un programme infecté est pathogène et au moins un est contagieux

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- Premiers programmes auto-réplicateurs: “rabbits” dans les années 60
- 1972: *When Harlie was one* par David Gerrold décrit un type de worm, implémenté ensuite par des chercheurs de Bolt Beranek and Newman pour montrer les capacités d'un nouveau OS
- 1975: le mot *worm* est employé pour la première fois dans le livre *The shockwave rider* par John Brunner
- Mi-70: Shoch et Hupp réalisent des worms chez Xerox pour balancer la distribution des charges dans leur réseau local
- 1980: les premiers virus sont faits sur des Apple II

```

ELK cloner :
the program with a personality
It will get on all your disks
It will infiltrate your chips
Yes it's cloner!
It will stick to you like glue
It will modify RAM too
Send in the cloner!

```

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- 1983: Ken Thompson révèle, lors de son discours du prix Turing, l'existence d'un cheval de Troie dans le compilateur C de AT&T
- 1984: *Neuromancer* par William Gibson
- 1986: premier virus pour IBM PC: *Brain*, écrit par les frères pakistanaïes Basit et Amjad Farooq Alvi. En 1990 il représentait le 7% des cas d'infection
- 1986: première réunion du Chaos Computer Club de Hambourg
- 1987: *WIR* pour Macintosh
- 1987: virus pour l'Amiga, écrit par la Swiss Cracker's Association (SCA)

```

Something wonderful has happened. Your AMIGA is alive!!!
and, even better...
Some of your disks are infected by a VIRUS!!!
Another masterpiece of the Mega-Mighty SCA!!

```

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- 1987: saturation de BITNET par une lettre de Noël
- 1988: Peter Norton: “les virus sont un exemple de mythe urbain”
- 1988: un virus dans la version officielle de FreeHand
- 1988: un “Atari virus construction set” pouvait être acheté à la CeBIT (interdit au moins de 18 ans...)
- 1988: worm sur Internet. Dégâts estimés à \$96 M
- 1989: numéro de Communications of the ACM consacré au worm d'Internet

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Warning: There's a new virus on the loose that's worse than anything I've seen before! It gets in through the power line, riding on the powerline 60 Hz subcarrier. It works by changing the serial port pinouts, and by reversing the direction one's disks spin. Over 300,000 systems have been hit by it here in Murphy, West Dakota alone! And that's just in the last twelve minutes. It attacks DOS, Unix, TOPS-20, Apple II, VMS, MVS, Multics, Mac, RSX-11, ITS, TRS-80, and VHS systems.

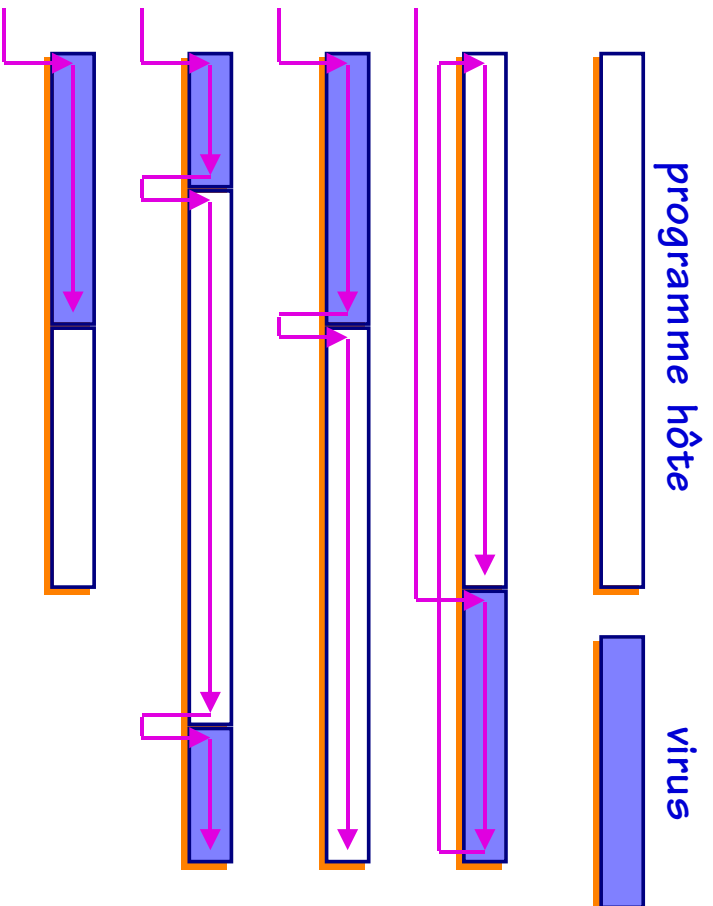
To prevent the spread of this dastardly worm:

- 1) Don't use the powerline.
- 2) Don't use batteries either, since there are rumors that this virus has invaded most major battery plants and is infecting the positive poles of the batteries. (You might try hooking up just the negative pole.)
- 3) Don't upload or download files.
- 4) Don't store files on floppy disks or hard disks.
- 5) Don't read messages. Not even this one!
- 6) Don't use serial ports, modems, or phone lines.
- 7) Don't use keyboards, screens, or printers.
- 8) Don't use switches, CPUs, memories, microprocessors, or mainframes.
- 9) Don't use electric lights, electric or gas heat or airconditioning, running water, writing, fire, clothing, or the wheel.

I'm sure if we are all careful to follow these 9 easy steps, this virus can be eradicated, and the precious electronic fluids of our computers can be kept pure.

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



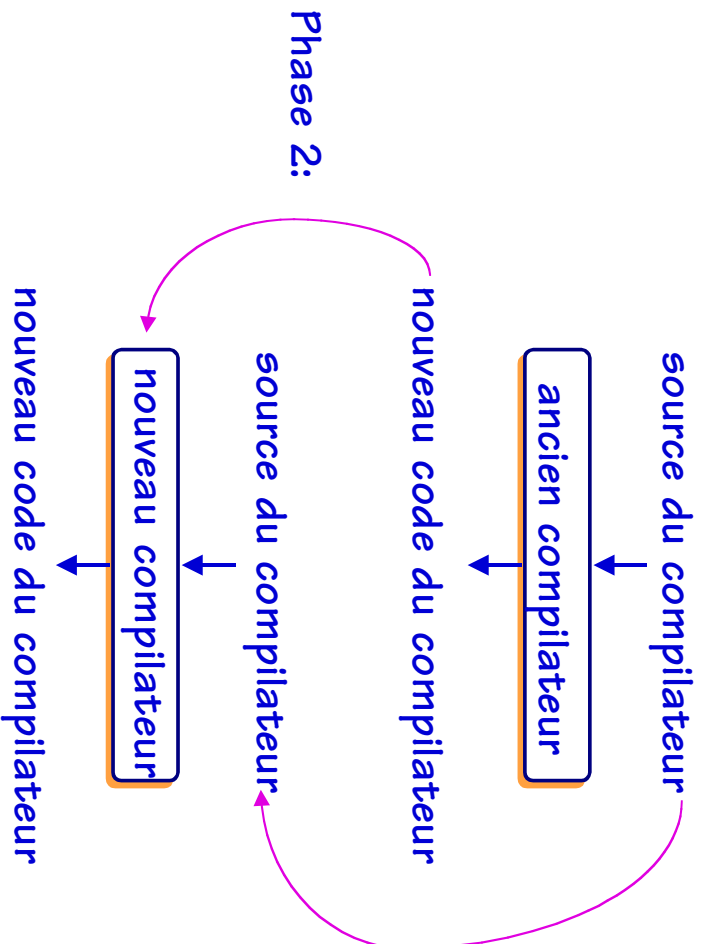


```

program virus
signature 1234567

subroutine infect_executable
begin
    loop: get random file
        if first line of file = 1234567
            then goto loop
        prepend virus to file
    end
end
subroutine do_damage
begin
    i!i$#%^$%@i
end
subroutine trigger_pulled
begin
    check for a particular system state
end
end
infect_executable
if trigger_pulled then do_damage
end
host program starts here
  
```

Phase 1:



```
compile (program)  
begin  
  if match(program, "login pattern") then  
    compile("login trojan")  
  return  
  else  
    ...  
  end  
end
```

```
compile (program)  
begin  
  if match(program, "compiler pattern") then  
    compile("compiler trojan")  
  return  
  else if match(program "login pattern") then  
    compile("login trojan")  
  return  
  else  
    ...  
end
```

Caractéristiques du vivant

- Auto-reproduction
- Comportement émergent
- Métabolisme
- Adaptabilité à l'environnement
- Evolution

Comportement émergent

- Le comportement d'un organisme vivant n'est pas dirigé par un programme centralisé. Le comportement émerge d'une interaction complexe entre les éléments individuels de l'organisme.
- Le comportement global d'un organisme manifeste des caractéristiques qui ne peuvent pas être déduites du comportement individuel de chacun de ses éléments.
- Un système manifeste un comportement fortement émergent si un ou plusieurs aspects de son comportement son théoriquement incalculables.

Introduction

- Dans la nature: la structure naît du *fitness*, via la sélection naturelle et les effets créatifs du *croisement génétique* et de la *mutation*
- Evolution naturelle:
 - ◆ incertaine
 - ◆ non déterministe
 - ◆ asynchrone
 - ◆ non coordonnée
 - ◆ locale
 - ◆ indépendante, sans contrôle centrale
- Un programme est une structure complexe: pourrait-elle naître également du *fitness*? En d'autres mots: un ordinateur pourrait-il résoudre un problème sans avoir été explicitement programmé pour cela?
- Le résultat de la programmation génétique est rarement la structure minimale capable de résoudre le problème posé. Par contre, il a des structures non utilisées ou inefficaces, reflet de l'histoire de la structure, plus que de sa fonctionnalité

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



L'algorithme génétique

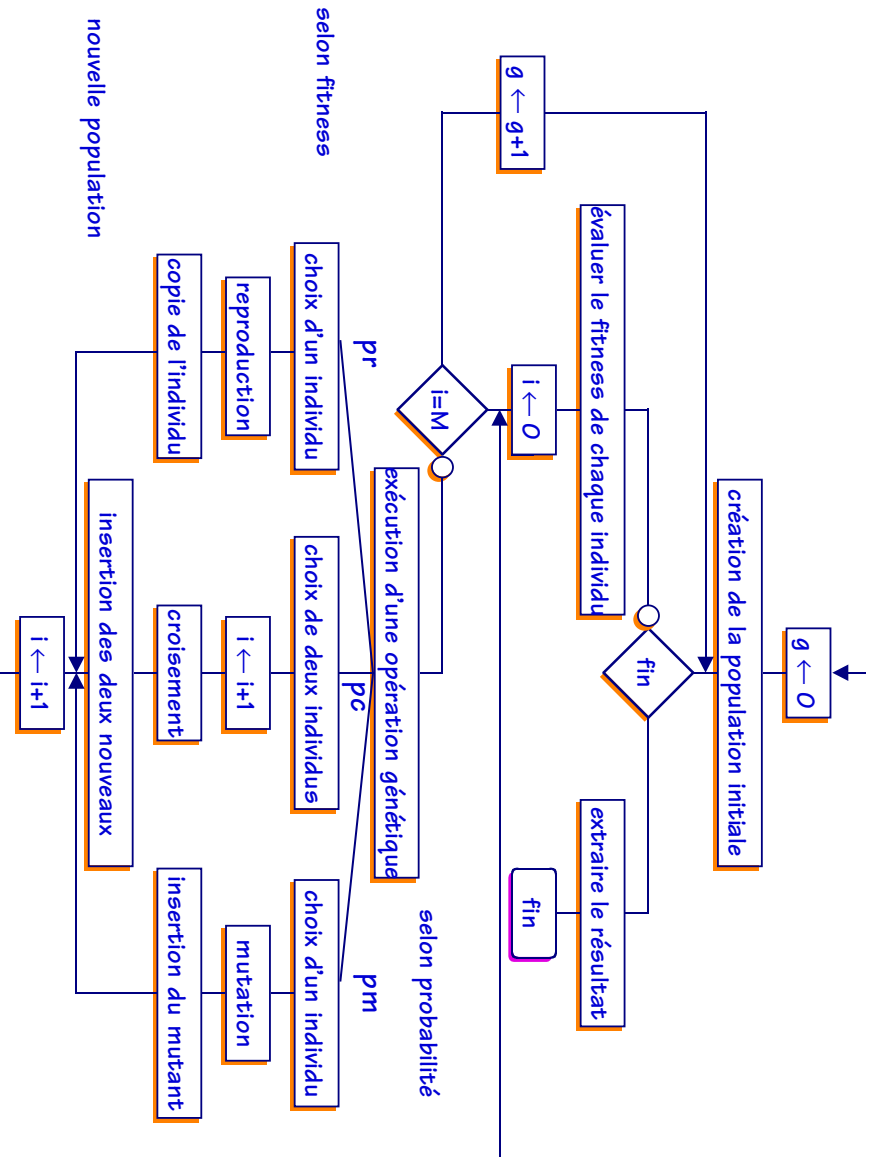
- Algorithme mathématique hautement parallèle qui transforme une population d'objets mathématiques individuels, chacun avec un *fitness* associé, dans une nouvelle population (génération suivante), en utilisant les principes darwiniens de reproduction et de survie des plus aptes, après réalisation de quelques opérations génétiques (notamment le *croisement sexuel*)

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- Pour utiliser un algorithme génétique sur un problème donné il faut réaliser quatre pas préalables:
 - ◆ déterminer un schéma de représentation: chaque point possible dans l'espace de solutions est représenté par une chaîne de caractères de longueur L, sur un alphabet de taille K
 - ◆ déterminer la mesure du fitness
 - ◆ déterminer les paramètres et variables de contrôle de l'algorithme (taille de la population, nombre de générations à produire, fréquence des opérations génétiques, etc)
 - ◆ déterminer la façon de choisir le résultat et le critère de terminaison de l'algorithme
- Pour une chaîne de longueur L et un alphabet de taille K, il y a K^L points dans l'espace de recherche de la solution du problème.
Exemple: pour une chaîne binaire de longueur 90, le nombre possible de solutions est:

$$2^{90} \approx 10^{27}$$
 si l'on suppose qu'on teste 109 points par seconde, pour tester tous les points il aurait fallu commencer au début de l'univers! (15×10^9 ans)



Représentation génétique d'une machine séquentielle

- Un gène par état total
- Le contenu du gène est l'état futur plus la sortie
- Un gène particulier contient l'état initial



Exemple:

Table d'états:

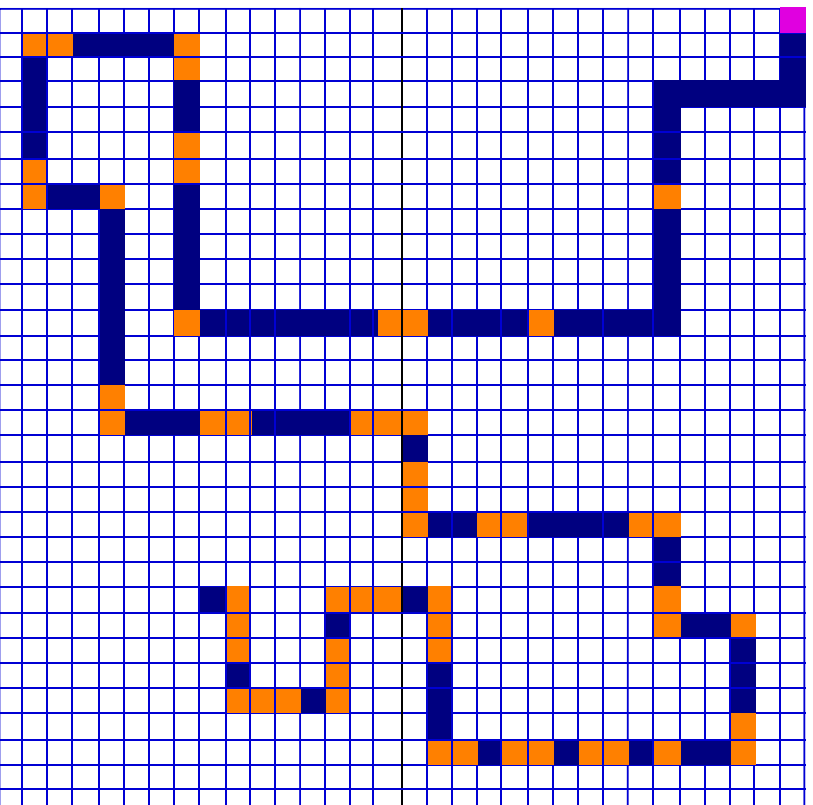
Y+,Z	X	
	0	1
00	01,1000,11	
01	10,0100,11	
10	11,0100,11	
11	00,1000,11	

Génome:

00	0110	0011	1001	0011	1101	0011	0010	0011
----	------	------	------	------	------	------	------	------



Exemple: la piste de Santa Fe



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- On essaie avec une machine séquentielle à 32 états, une seule variable d'entrée (détecteur de nourriture en face) et trois actions de sortie, codées avec 2 bits (avancer, tourner à droite, tourner à gauche)
- Le génome aura 453 bits: 64 gènes de 7 bits chacun (5 bits pour l'état et 2 pour la sortie) plus le gène de l'état initial (5 bits)
- Le *fitness* est tout simplement le nombre de pièces de nourriture trouvées dans un certain laps de temps (0 à 89 en 200 pas)
- La taille de la population initiale est de 65536
- Le nombre maximal de générations est de 200
- Dans une Connection Machine, une solution fut trouvée après 200 générations (les 89 pièces de nourriture étaient trouvées en 200 pas!)

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



La programmation génétique

- Trouver dans l'espace des programmes possibles celui qui est le mieux adapté au problème posé. L'espace de recherche est formé par tous les programmes nés de la combinaison des **fonctions** et des **terminaux** adéquats au domaine du problème
- On commence avec une population initiale faisant partie de l'espace de recherche. Chaque membre de la population possède un *fitness*. Ensuite on produit de nouvelles générations, en appliquant des opérations génétiques (notamment la reproduction et le croisement sexuel)

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- L'ensemble de terminaux et l'ensemble de fonctions pour un problème donné doivent satisfaire aux contraintes de:

- ◆ **closure**: toute fonction doit être définie pour toute combinaison d'arguments qu'elle puisse rencontrer
- ◆ **sufficiency**: une solution existe pour ces ensembles

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Introduction à LISP

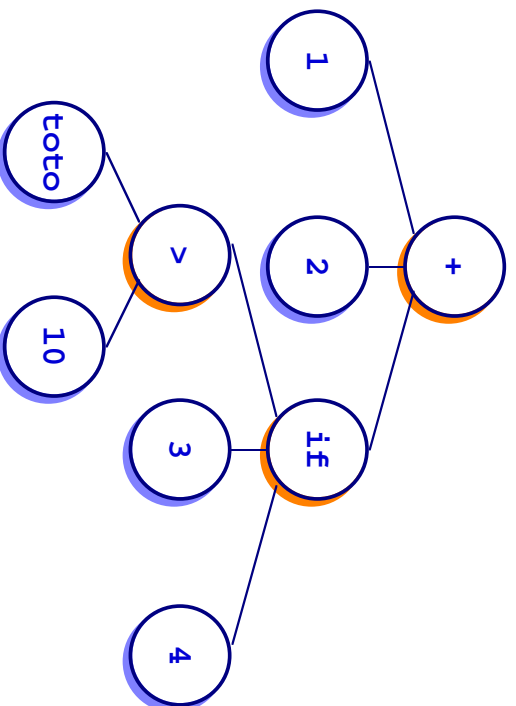
- Seulement deux entités:
 - ◆ les atomes et
 - ◆ les listes
- Un programme est une **S-expression**, qu'on peut représenter par un arbre ordonné, l'arbre d'analyse grammaticale (**parse**) du programme

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



■ Exemple:

(+ 1 2 (if (> toto 10) 3 4))



La racine et les nodes internes de l'arbre sont des fonctions; les feuilles sont des atomes (ou terminaux)

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

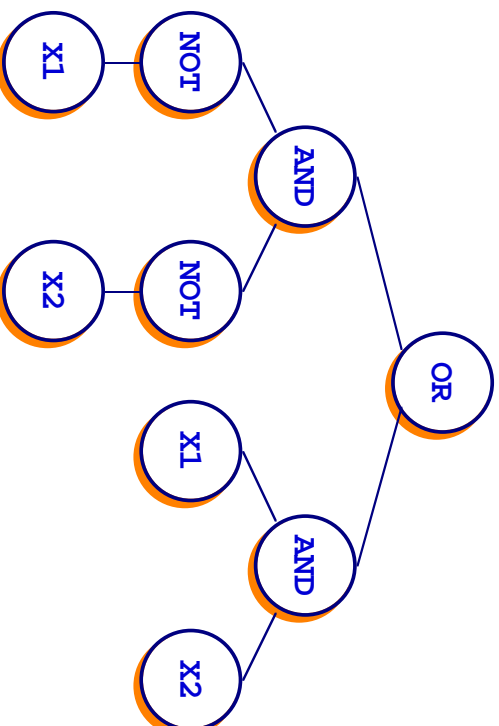


- Exemple:
pour obtenir un programme réalisant la fonction booléenne
XNOR on peut commencer avec:

$F = \{\text{AND, OR, NOT}\}$

$T = \{X1, X2\}$

$(\text{OR } (\text{AND } (\text{NOT } X1)) (\text{NOT } X2)) (\text{AND } X1 X2))$



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- Les pas préalables pour appliquer la programmation génétique à un problème donné sont:

- ◆ déterminer l'ensemble de terminaux
- ◆ déterminer l'ensemble de fonctions
- ◆ déterminer la mesure de fitness
- ◆ déterminer les paramètres et variables de contrôle
- ◆ déterminer la méthode de choix du résultat et le critère de terminaison

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- Pour la création de la population initiale on peut utiliser plusieurs méthodes, selon la contrainte imposée sur la longueur des branches:
 - ◆ *full*: toutes les branches ont la même longueur, égale à une valeur maximale spécifiée
 - ◆ *grow*: les branches ont des longueurs différentes, mais qui ne dépassent pas une valeur maximale spécifiée
 - ◆ *ramped half-and-half*: il y a un nombre égal d'arbres pour chaque longueur entre 2 et une valeur maximale spécifiée. Et pour chaque longueur il y a une moitié générée avec la méthode *full* et une autre avec la méthode *grow*

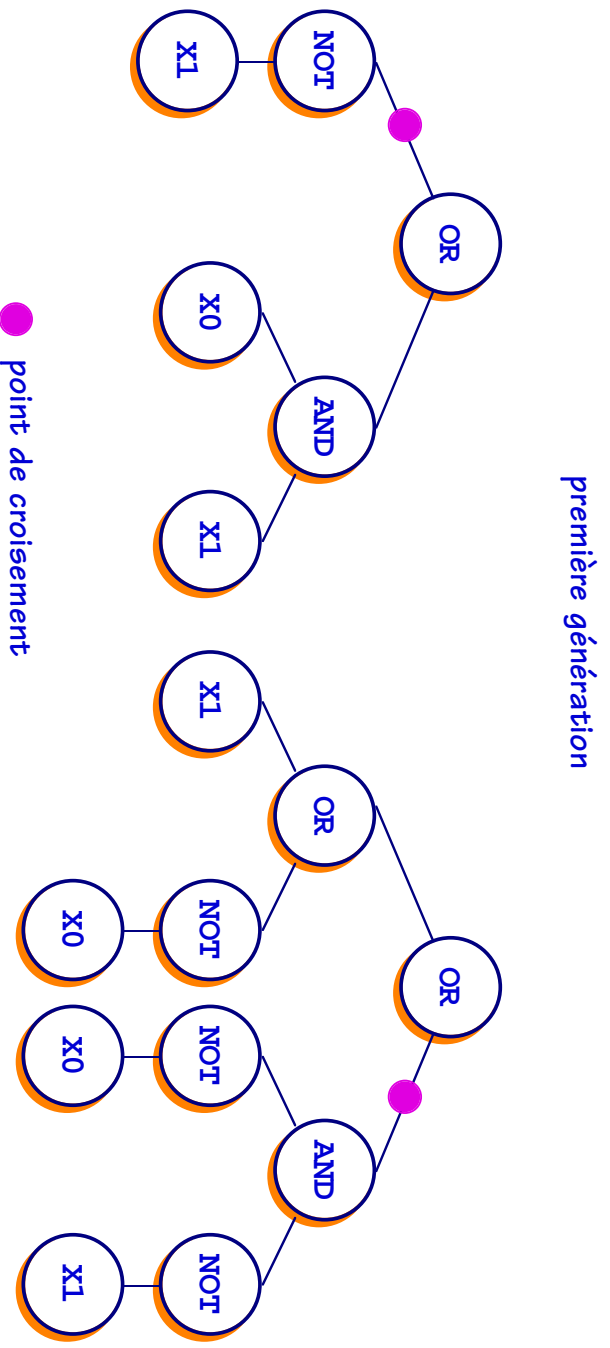
Opérations génétiques

- Opérations génétiques principales:
 - ◆ *reproduction*
 - ◆ *croisement*
- Opérations génétiques secondaires (optionnelles):
 - ◆ *mutation*: un point de mutation est choisi au hasard et son sous-arbre est remplacé par un autre généré au hasard
 - ◆ *permutation*: une fonction est choisie au hasard et ses arguments sont permutés
 - ◆ *édition*: une fonction est remplacée par son résultat
 - ◆ *encapsulation*: un sous-arbre, choisi au hasard, est remplacé par une fonction équivalente, sans arguments. Cette fonction est ajoutée à F
 - ◆ *décimation*: un certain pourcentage de la population est effacé à un certain moment

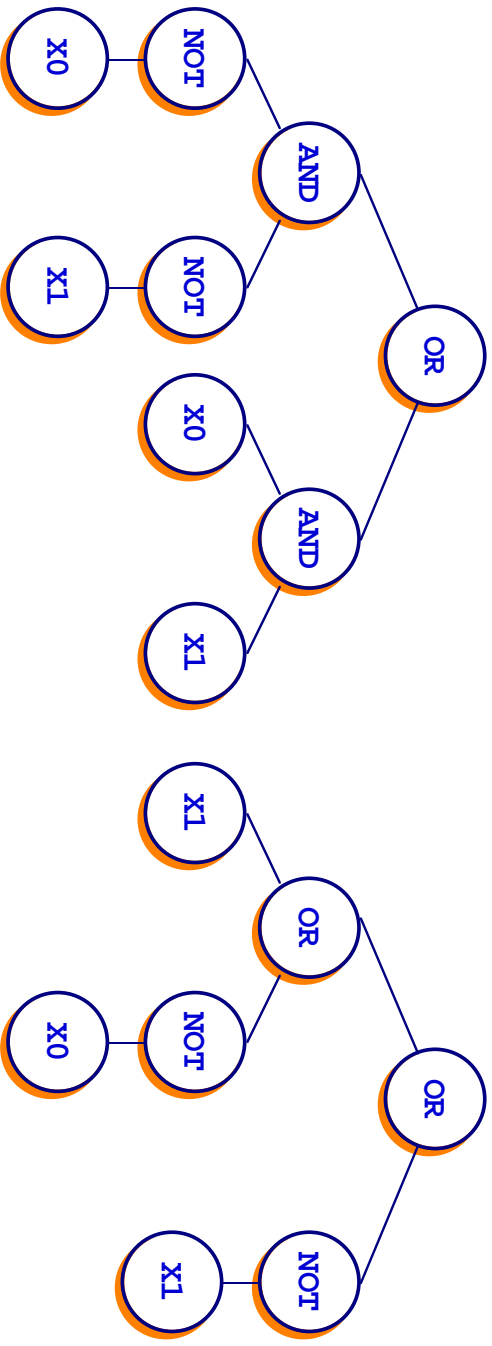
Le croisement

- Deux S-expressions sont utilisées comme parents pour en générer deux nouvelles: c'est sexuel
- Les deux parents sont choisis par la même méthode utilisée pour la reproduction. Ensuite, un point de croisement est choisi aléatoirement dans chaque parent. Les deux sous-arbres respectifs sont échangés
- Les deux nouveaux programmes ainsi générés sont toujours des S-expressions correctes syntaxiquement (contrainte de *closure*). Et, contrairement à l'algorithme génétique, les deux enfants de deux parents identiques sont différents (sauf si les deux points de croisement sont les mêmes): ce qui amène une plus grande diversité et réduit l'importance de la mutation
- Pour éviter de longs calculs, il est interdit aux enfants de dépasser une certaine longueur (Koza utilise 17 pour cette limite, ce qui est énorme: pour des fonctions dyadiques, le programme le plus grand aurait $2^{17} = 131072$ nodes, c'est-à-dire 33000 lignes de LISP, si l'on suppose 4 fonctions ou terminaux par ligne)

Exemple:



deuxième génération



Paramètres de contrôle

- taille de la population ($M = 500$)
- nombre maximal de générations ($g = 51$)
- probabilité de croisement ($p_c = 0.9$)
- probabilité de reproduction ($p_r = 0.1$)
- pourcentage de points de croisement choisis parmi les fonctions (90%)
- longueur maximale après une opération ($D_{created} = 17$)
- longueur maximale initiale ($D_{init} = 6$)
- probabilité de mutation ($p_m = 0$)
- probabilité de permutation ($p_p = 0$)
- fréquence d'édition ($f_{ed} = 0$)
- probabilité d'encapsulation ($p_{en} = 0$)
- condition pour appeler la décimation (nil)
- pourcentage de décimation ($p_d = 0$)
- méthode de génération de la population initiale (*ramped half-and-half*)
- méthode de sélection pour la reproduction et pour le premier parent du croisement (proportionnelle au *fitness*)
- méthode de sélection pour le deuxième parent du croisement (proportionnelle au *fitness*)
- mesure du *fitness*
- utilisation de la sur-sélection (seulement pour des populations ≥ 1000)
- stratégie élitiste (non employée)

Exemple: la piste de Santa Fe

- $F = \{\text{if-food-ahead, progn2, progn3}\}$
 $T = \{\text{move, right, left}\}$
- Mesure du *fitness*: quantité de nourriture trouvée dans un certain laps de temps (0 à 89 dans 400 pas (seulement les terminaux prennent un pas))
- Un seul cas est traité: la position initiale est (0,0), la fourmi se dirige à l'est et une seule trace doit être suivie
- Condition de terminaison: atteindre un *fitness* de 89 ou la génération 51
- Génération initiale: *fitness* moyen = 3.5
meilleur *fitness* = 32
pire *fitness* = 0

- A la 21ème génération un programme a été trouvé avec un *fitness* optimal: une *S-expression* à 18 points

```
(if-food-ahead (move)
  (progn3 (left)
    (progn2 (if-food-ahead (move)
      (right))
      (progn2 (right)
        (progn2 (left)
          (right))))))
  (progn2 (if-food-ahead (move)
    (left))
    (move))))
```

- Si la fonction *left* est effacée, on trouve une solution avec 20 points à la 19ème génération

Exemple: un multiplexeur à 3 variables de contrôle

- $F = \{\text{and, or, not, if}\}$
- $T = \{a_0, a_1, a_2, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
- Cas intéressant parce que l'espace de recherche est calculable:
 $2^{2k+2k} = 2^{23+23} = 2^{2048} \approx 10616$
- On prend comme *fitness* le nombre de cas corrects parmi les 2048 combinaisons possibles des 11 variables d'entrée
- Taille de la population: 4000
- Critère de terminaison: atteindre le *fitness* optimal ou la génération 51
- La génération initiale avait des *fitness* entre 768 et 1280

- Une solution fut trouvée à la 8ème génération:

```
(if a0 (if a2 (if a1 d7 (if a0 d5 d0))
      (if a0 (if a1 (if a2 d7 d3) d1) d0))
 (if a2 (if a1 d6 d4)
      (if a2 d4 (if a1 d2 (if a2 d7 d0))))))
```


Questions

- Est-ce valable pour des problèmes complexes, conduisant à des très grands programmes?
- Est-ce valable pour des ensembles complexes de terminaux et de fonctions?
- Quel est le comportement pour des cas non pris en compte lors du calcul du fitness?
- Est-ce possible d'optimiser en même temps la justesse, la taille et l'efficacité du programme?

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



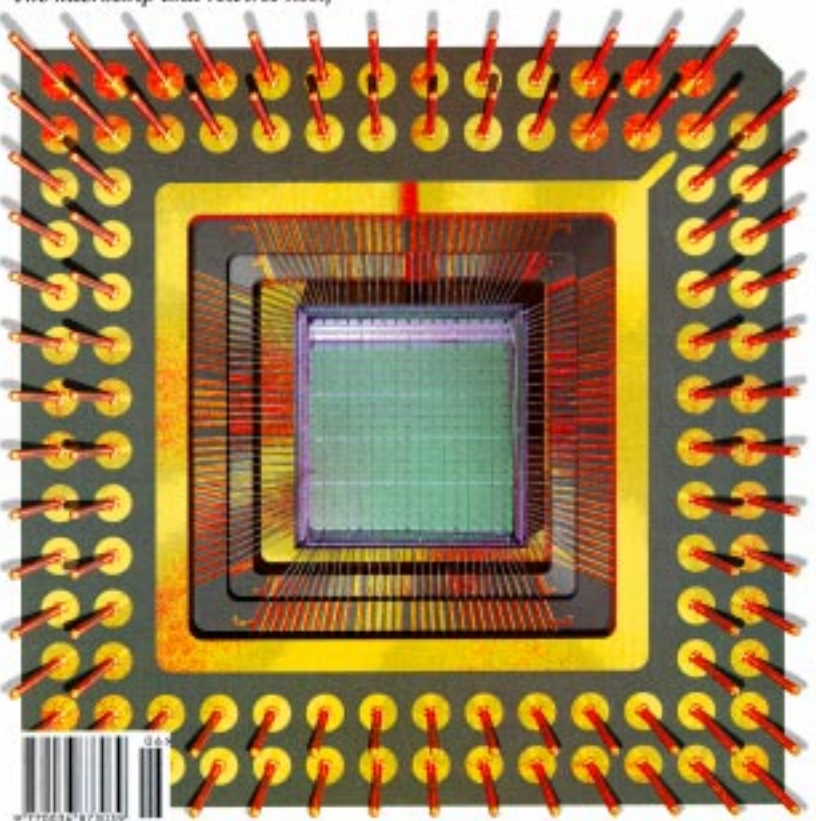
IRAN'S NUCLEAR MENACE • MAPPING THE SEAFLOOR • FROM CATS TO QUANTUM PHYSICS

SCIENTIFIC AMERICAN

JUNE 1997 \$4.95 U.K. £3.00

SPECIAL REPORT
GENE THERAPY:
HOW IT WILL WORK
AGAINST CANCER, AIDS,
ALZHEIMER'S AND MORE

The microchip that rewrites itself



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Programmable vs configurable

- The programmable paradigm involves a (general-purpose) processor, able to execute a limited set of operations, known as the **instruction set**
- The configurable-computing paradigm can also be regarded as one involving a processor that is able to execute but a given set of operations - however, these are at a much lower level. One controls the actual type of the logic devices, the input signals, and the output signals
- In both cases the algorithm is ultimately expressed as a string of bits that is stored in memory, with the difference being the manner in which these bits are interpreted

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



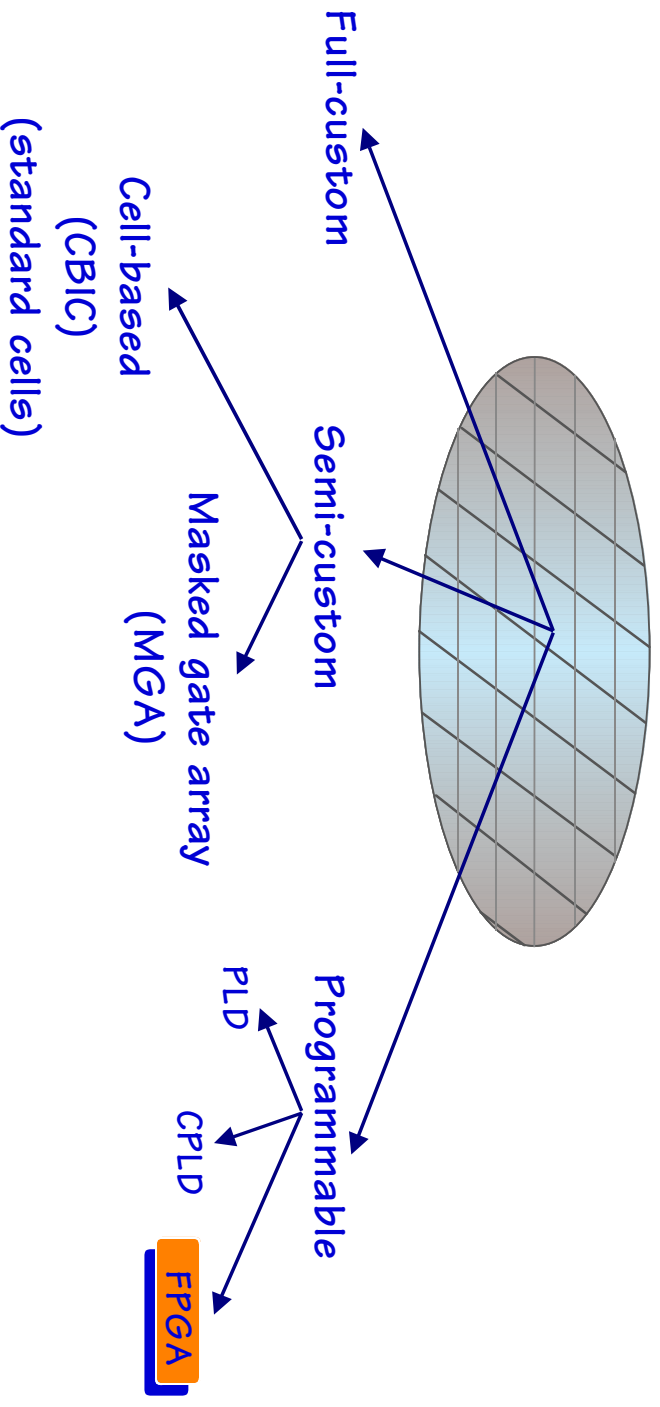
Static vs dynamic

- One can distinguish two types of configuration strings: static and dynamic
- A static configuration string aims to configure the processor in order to perform a given function. It is loaded (once) at the outset, then it does not change during the execution of the given task.
- A static configuration has two main objectives:
 - ◆ improving performance (i.e., execution speed) for a given function: an extension of the coprocessor concept
 - ◆ optimizing the utilization of resources (gates and power consumption) so as to use as much of the chip surface as possible, at each clock cycle
- Dynamic configuration involves a configuration string that can change during the execution of the task at hand, with the two main objectives being:
 - ◆ to adapt to changing (dynamic) specifications as well as to be able to handle incomplete specifications
 - ◆ to eliminate human design altogether

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Application-Specific Integrated Circuits



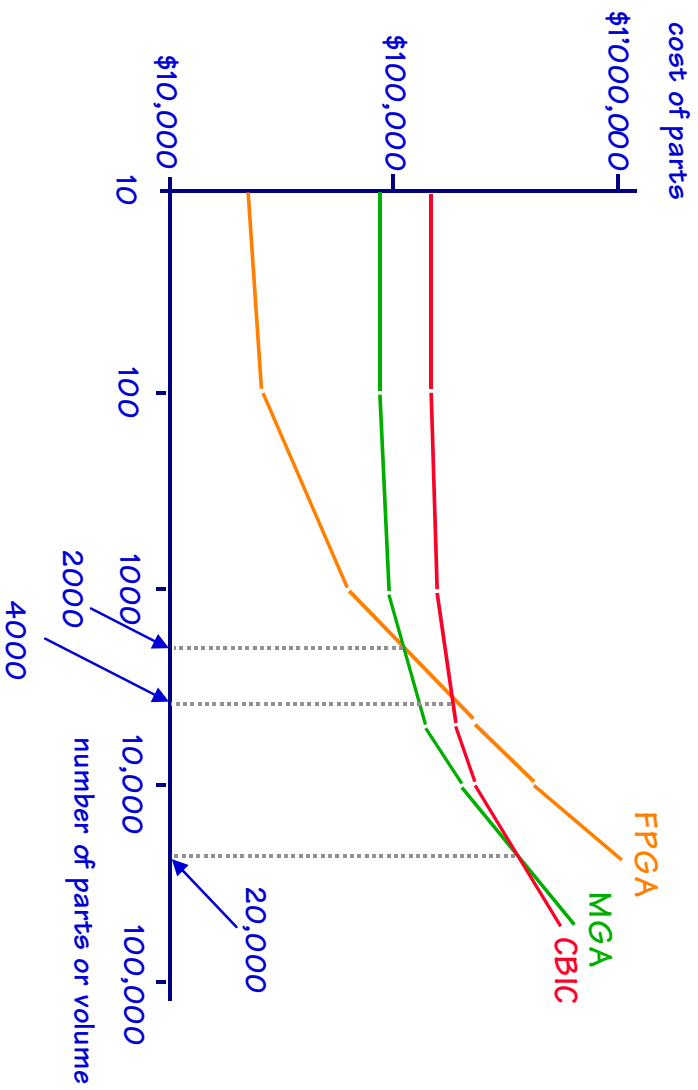
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



	 FPGA 	 MGA 	 CBIC 	
wafer size	6	6	6	inches
wafer cost	1.4	1.3	1.5	K\$
design density	10	10	10	Kgates
utilization	10	20	25	Kgates/cm2
die size	60	85	100	%
die/wafer	1.67	0.59	0.4	cm2
defect density	88	248	365	
yield	1.1	0.9	1	defects/cm2
die cost	65	72	80	%
price/gate	25	7	5	\$
part cost	0.39	0.1	0.08	cents
	39	10	8	\$

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne





Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

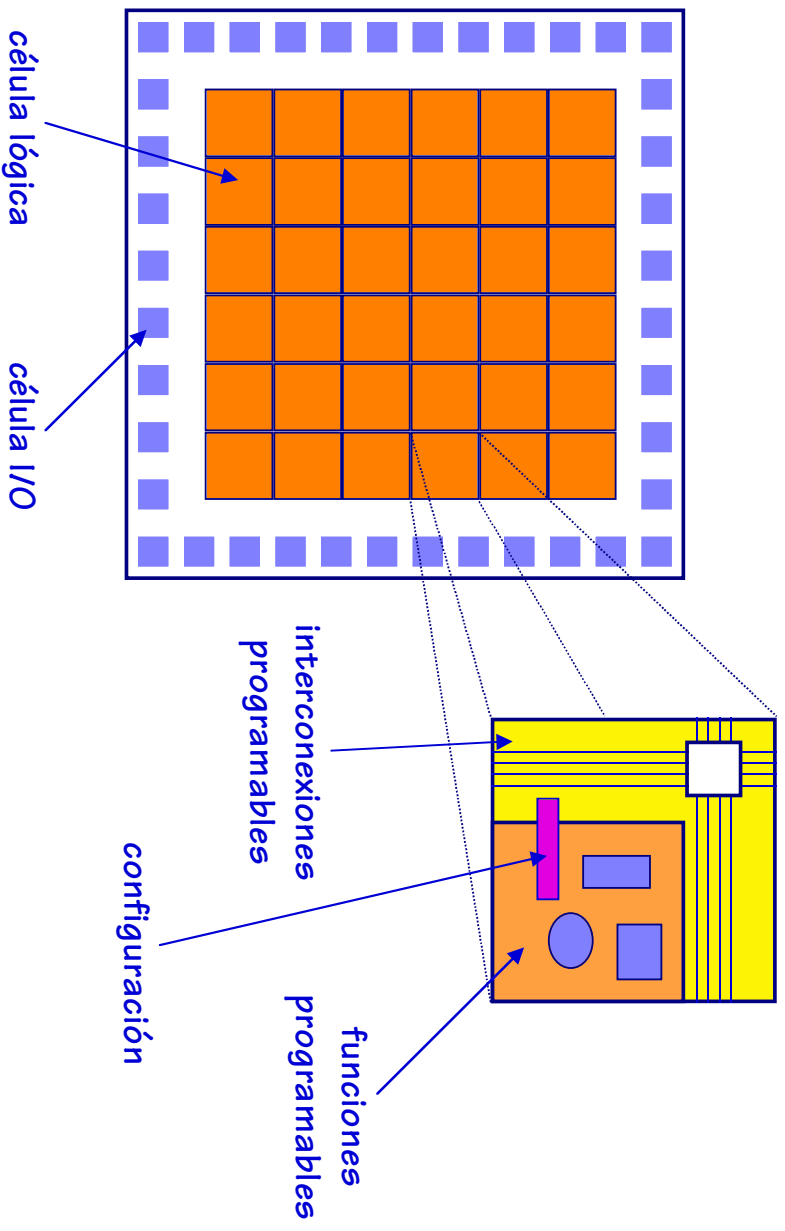


Field-Programmable Gate Arrays

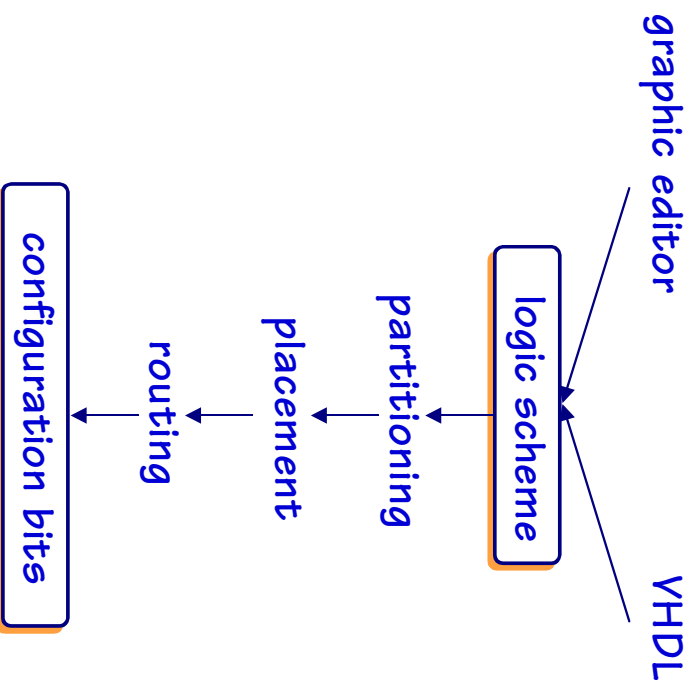
- Matriz de células lógicas
- Cada célula es capaz de realizar unas función, entre varias posibles: la selección se hace por programación
- Las interconexiones entre las células son igualmente programables
- Dos tipos según la complejidad de la célula:
 - ◆ granularidad fina
 - ◆ granularidad gruesa
- Dos tipos según el modo de programación:
 - ◆ RAM
 - ◆ anti-fusibles

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne





Synthesis methodology



Xilinx circuits

- Each logic cell is called Configurable Logic Block (CLB), it is programmed by means of a *look-up table* (LUT)
- The loading of the configuration can take several milliseconds. During loading, the circuit cannot be used.
- Several families are available. They can be classified into two groups:
 - ◆ coarse grain families (XC2000, XC3000, XC4000, XC5200): it is not possible to partially configure it.
 - ◆ the fine grain XC6200 family: it is possible to directly access every single cell.

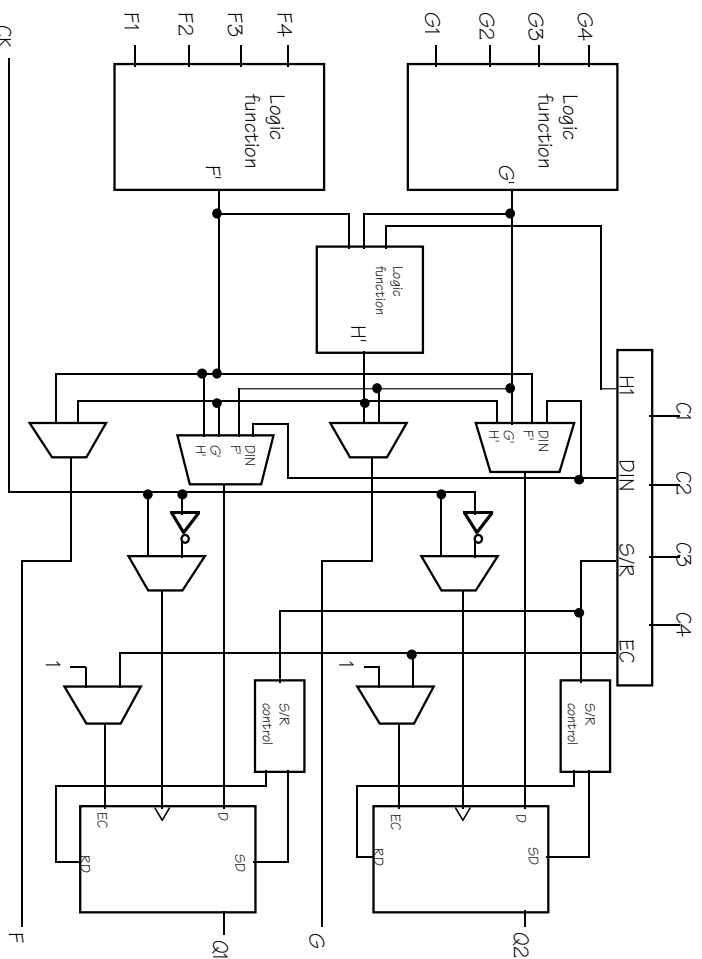


■ Comparison of the different families

	2000	3000	4000	5200
Functions/CLB	2	2	3	4
Inputs/CLB	4	5	9	20
Outputs/CLB	2	2	4	12
RAM bits	no	no	yes	no
Decoders	no	no	yes	no
Drivers 3-state	no	yes	yes	yes
Clock signals	2	8	4	
CLBs (max)	100	320	4624	1936
IOBs (max)	74	144	544	244
Flip flops (max)	174	928	10336	1936
Logic gates(max)		125K	15K	

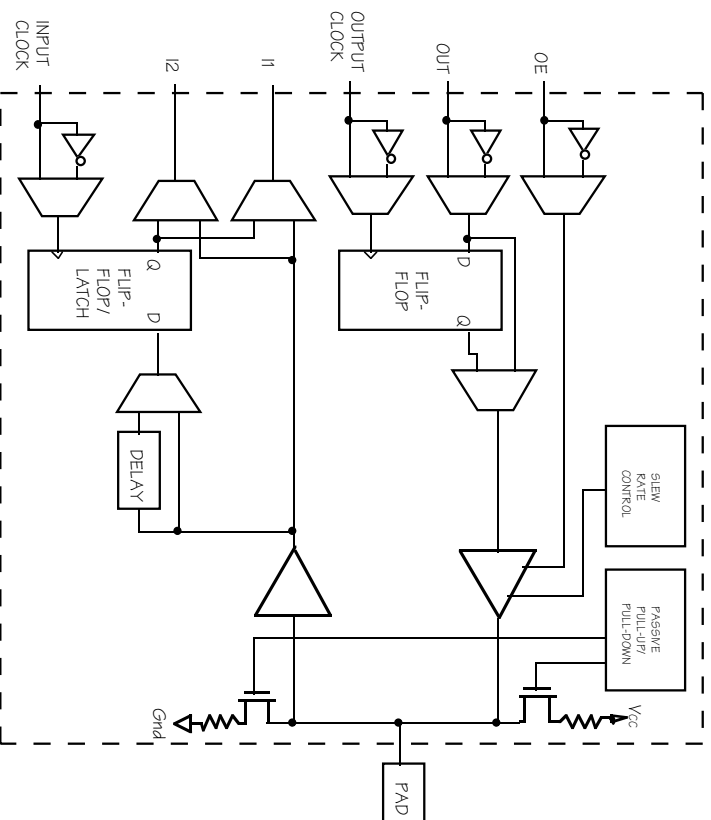


- X4000 family CLB:



- One can generate two combinational or sequential outputs per CLB
- It is possible to generate two 4-variable functions, one 5-variable function or certain 9-variable functions.
- The look-up tables can also be used as RAM. In such case, there are several possible configurations: one RAM 16x2, one RAM 32x1, two RAM 16x1 or one RAM 16x1 and one 4-variable combinatorial function.
- The flip-flops maintain set/reset programmable signals and can be used independently of the logic functions of the cell.

■ X4000 family I/O cell (IOB) :



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



■ Programming modes:

- ◆ **master:**
 - ❖ serial
 - ❖ parallel with increasing addresses
 - ❖ parallel with decreasing addresses
- ◆ **peripheral:**
 - ❖ parallel asynchronous
 - ❖ parallel synchronous
- ◆ **slave:** serial

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



- **XC6200 Family**
This is the latest Xilinx FPGA family. It presents unique characteristics which makes it very well adapted to evolvable hardware applications:
 - ◆ the configuration bits are internally organized as a very fast SRAM memory.
 - ◆ the configuration is not global: it is possible to configure a certain number of cells without stopping the circuit.
 - ◆ it is possible to read the state of each logic cell
 - ◆ every configuration bit string is valid.
- The routing mechanisms are hierarchic, with interconnection lines between the different levels
- The logic cell is fine grain: it contains a single flip-flop and a two-input logic gate

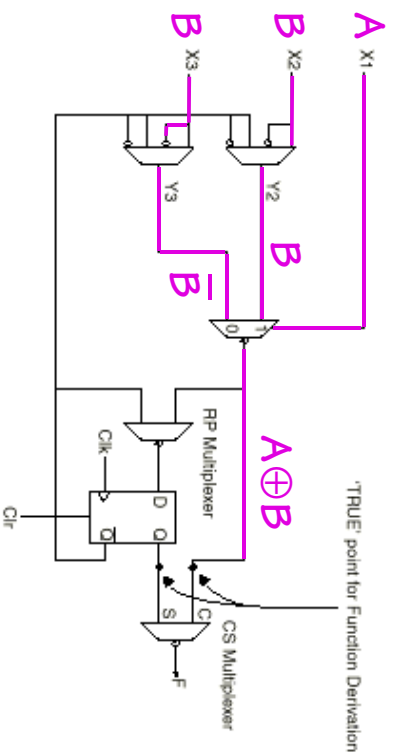


Main features

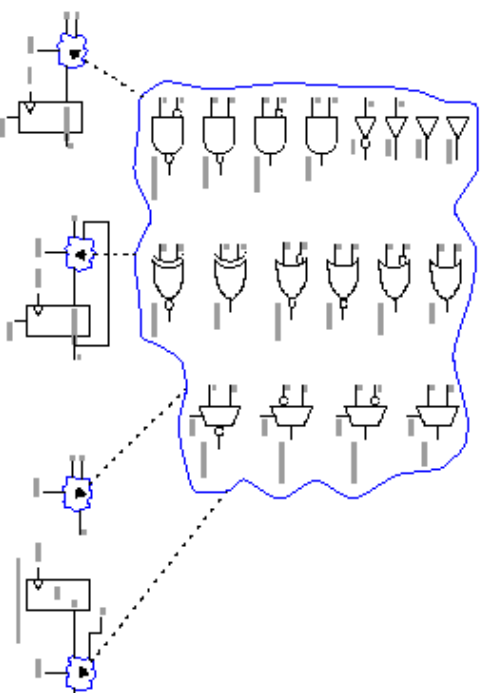
	XC6216	XC6264
Number of gates	16000-24000	64000-100000
Number of cells	4096	16384
Number of flip-flops	4096	16384
Number of IOBs	256	512
Organization	64x64	128x128



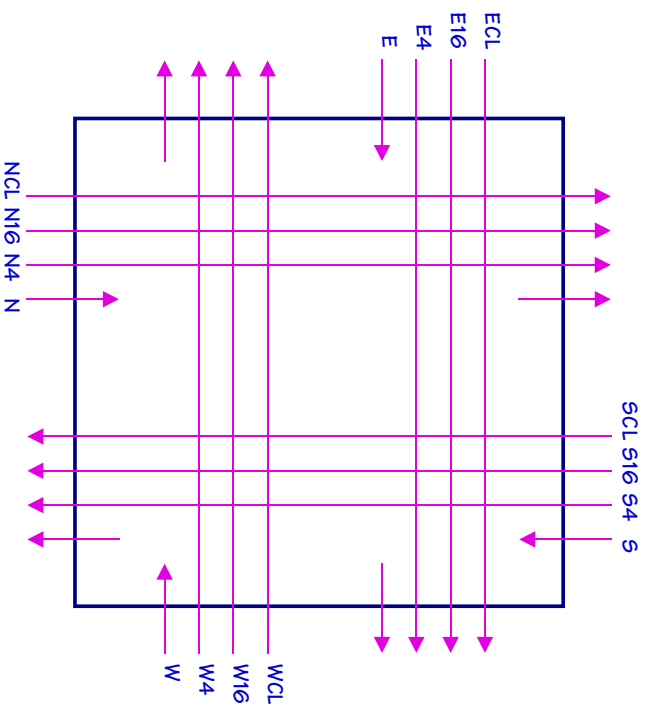
Example of a function



Possible functions of a cell



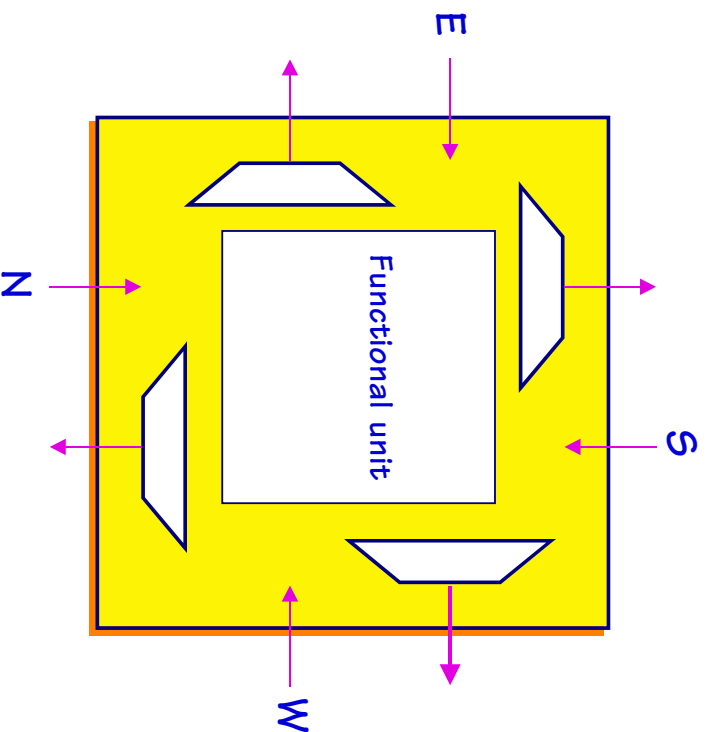
Interconnection lines of a cell



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



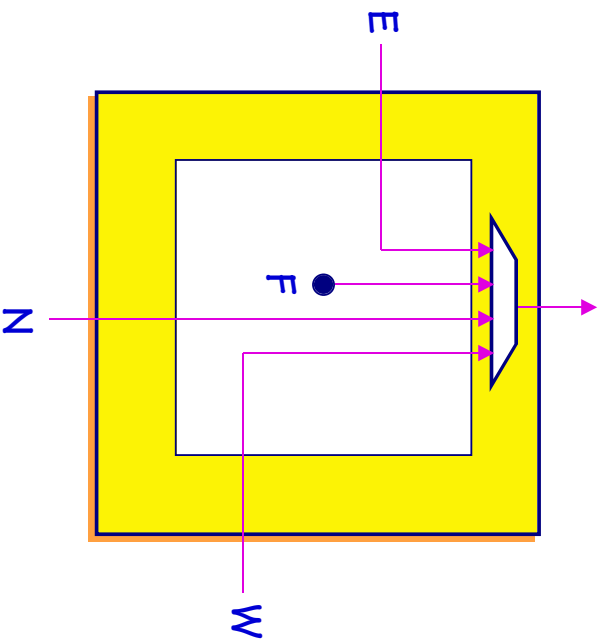
Interconnection lines 1x1



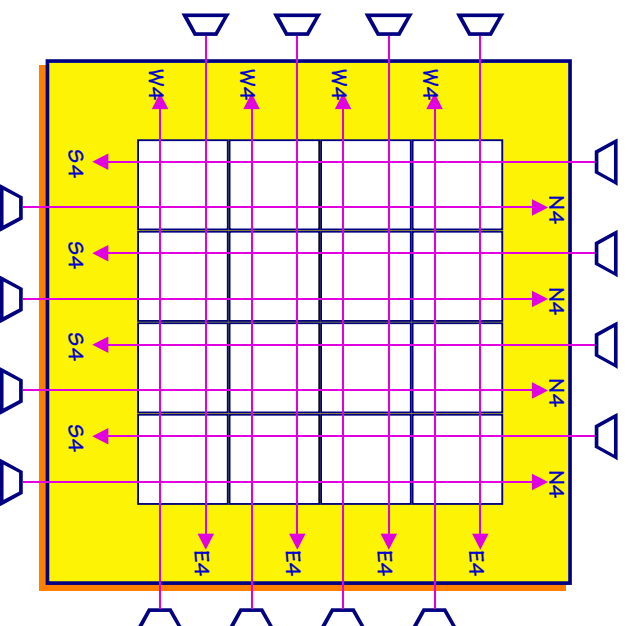
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



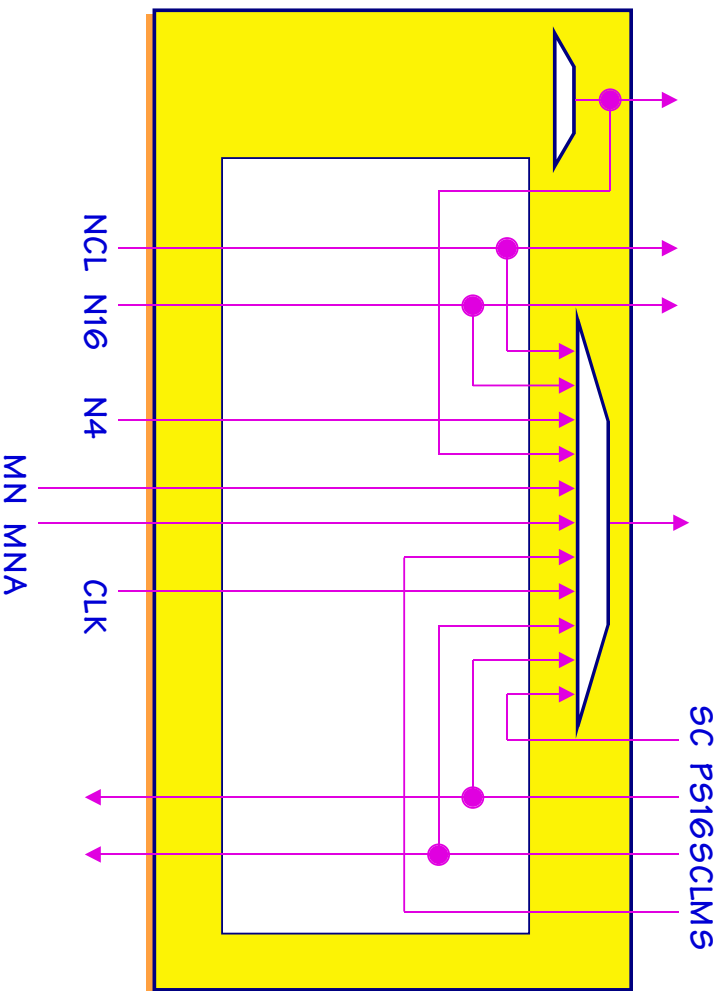
Example: north 1x1



Interconnection lines 4x4



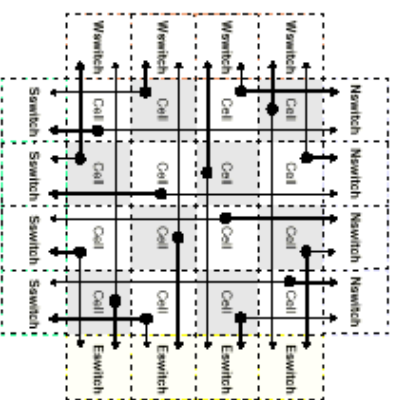
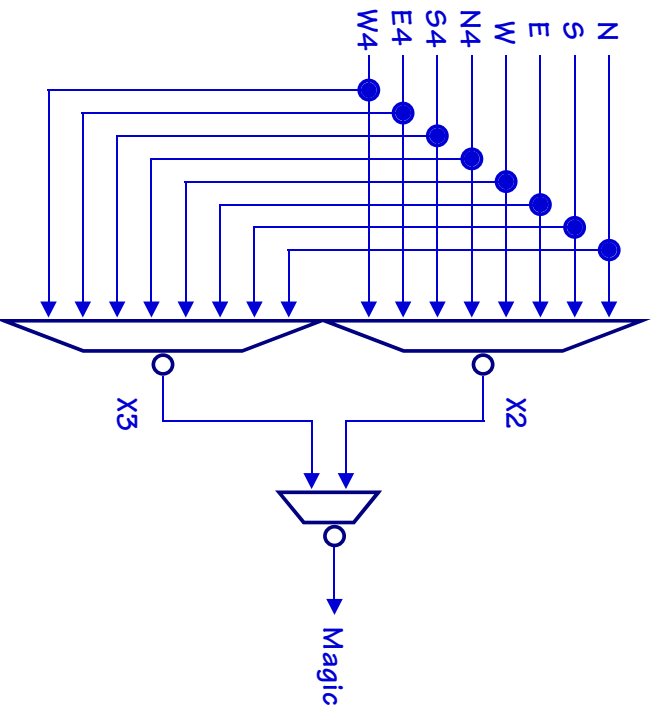
Example: north 4x4



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



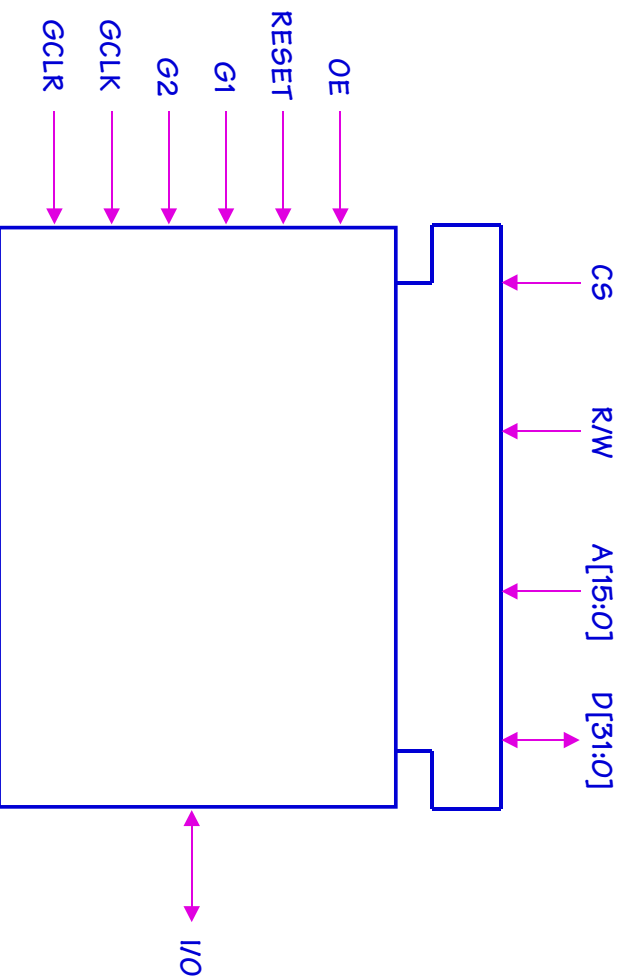
“Magic” interconnection lines



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



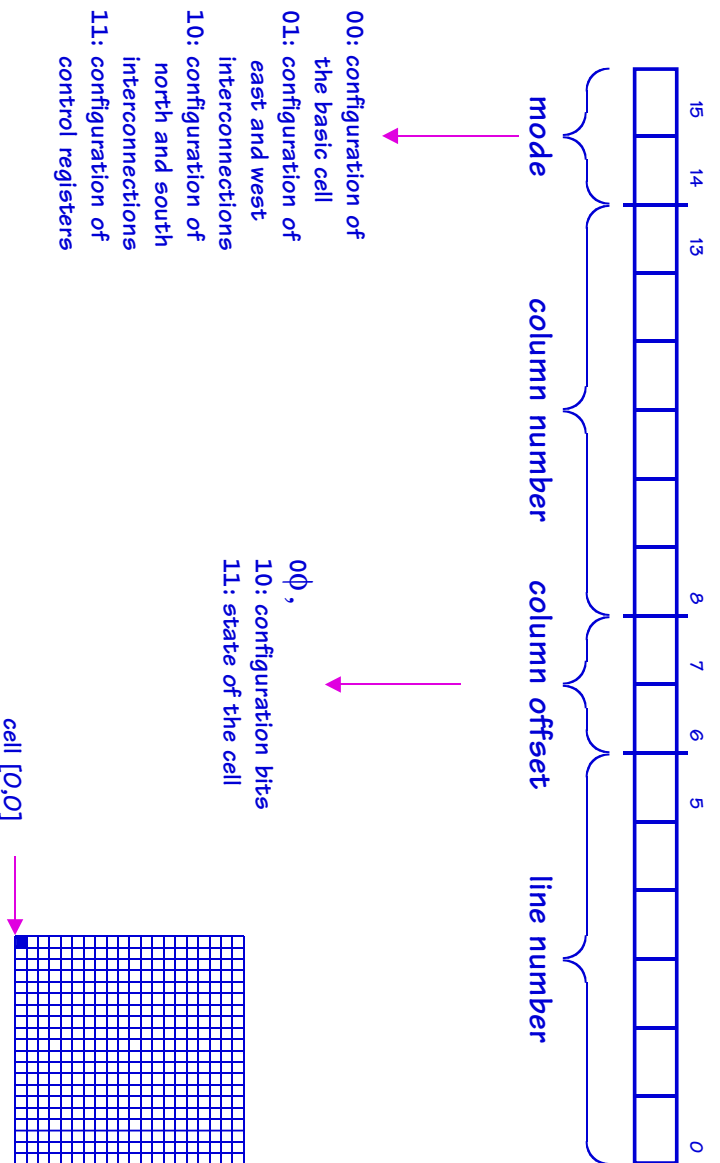
XC6216 configuration memory organization



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Configuration memory addressing :



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Altera circuits

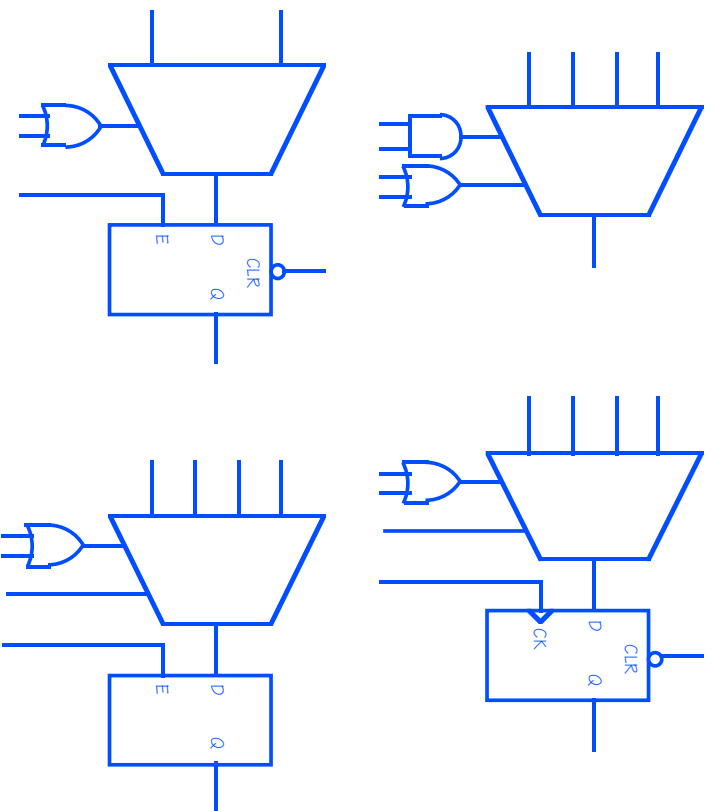
- Reprogrammables circuits with logic cells with two levels of organization
- Main FLEX 8000 circuits:

Gates	8282	8452	8636	8820	81188	81500
Flip-Flops	5000	8000	12000	16000	24000	32000
Logic elements	282	452	636	820	1188	1500
Inputs/outputs	208	336	504	672	1008	1296
	78	120	136	152	184	208

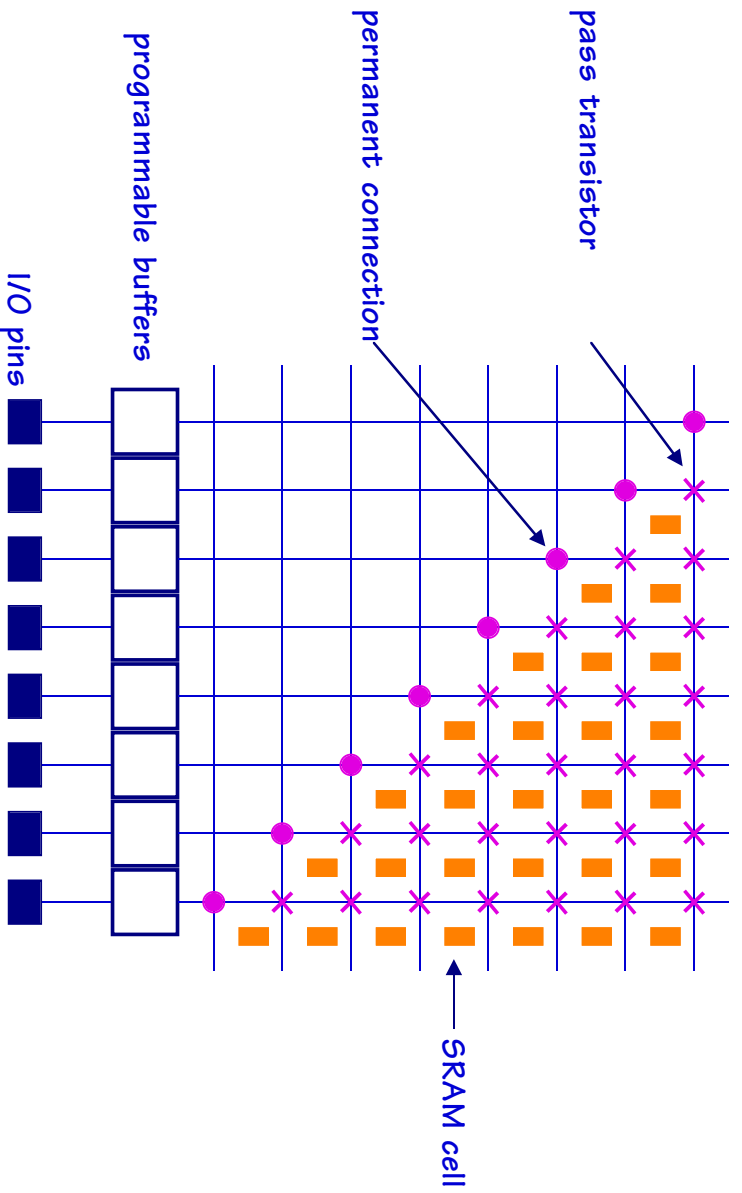
Actel Circuits

- The Actel circuits are not reprogrammable: the configuration is carried out by means of anti-fuses.
- The logic cells are fine grain, and are multiplexor-based, because of its universal character
- Three families are currently available: ACT1, ACT 2 et ACT3

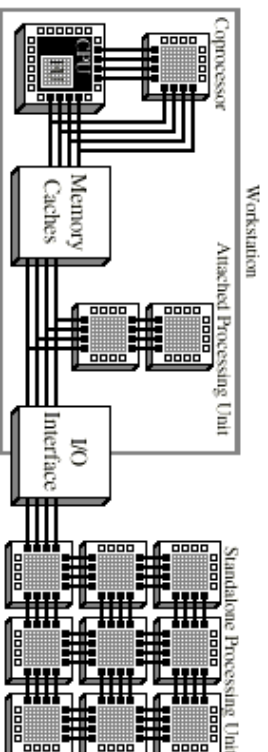
■ Logic cell structures:



Field Programmable Interconnection Circuits



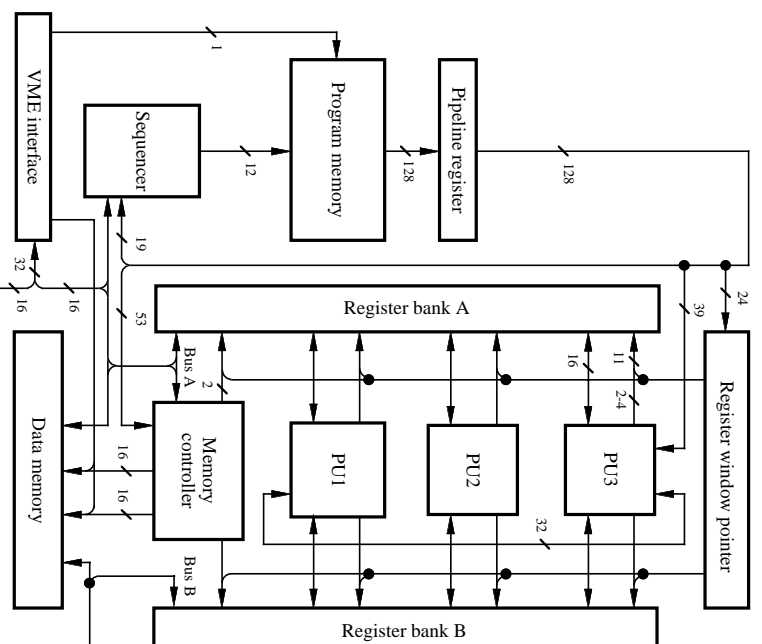
Static systems



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



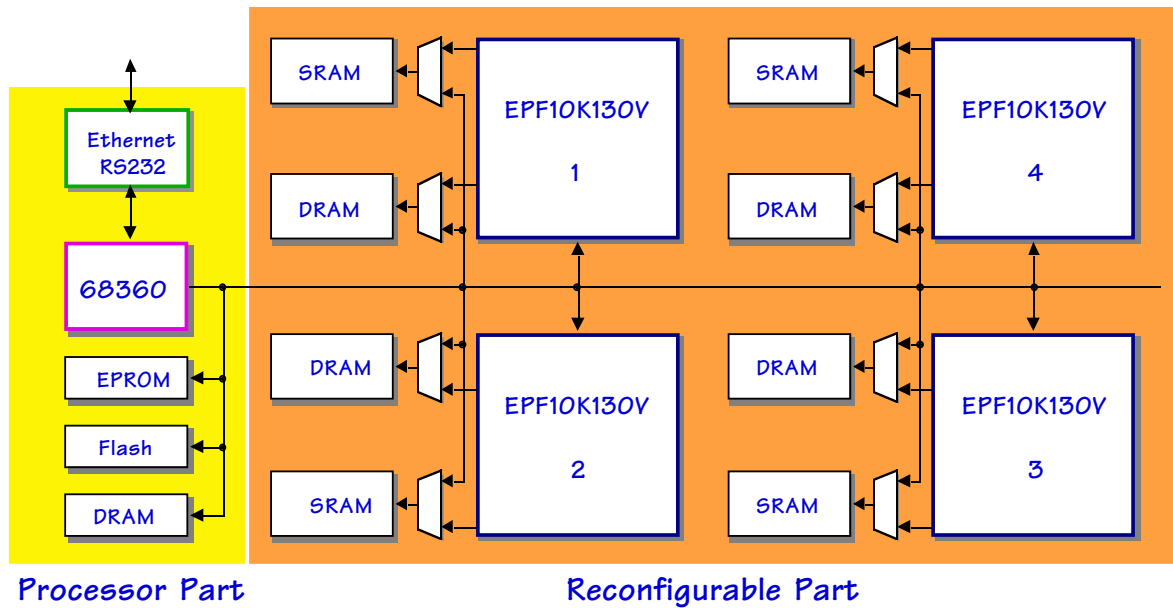
Spyder architecture



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



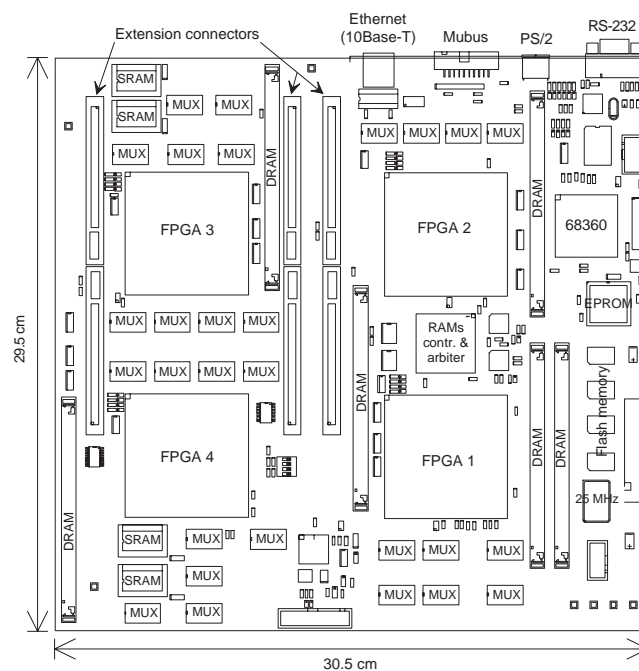
RENCO structure



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



RENCO printed circuit board



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Features

■ Processor part:

- ◆ MC68EN360
- ◆ boot EPROM (512KB, 16 bits)
- ◆ Flash memory (2MB, 32 bits)
- ◆ DRAM (two SIMM sockets, up to 64M, 32 bits)
- ◆ full access to FPGA DRAMs
- ◆ access to FPGAs in peripheral mode
- ◆ 25MHz

■ Communication interfaces:

- ◆ Ethernet (10BaseT)
- ◆ RS232
- ◆ Mubus
- ◆ PS/2 keyboard

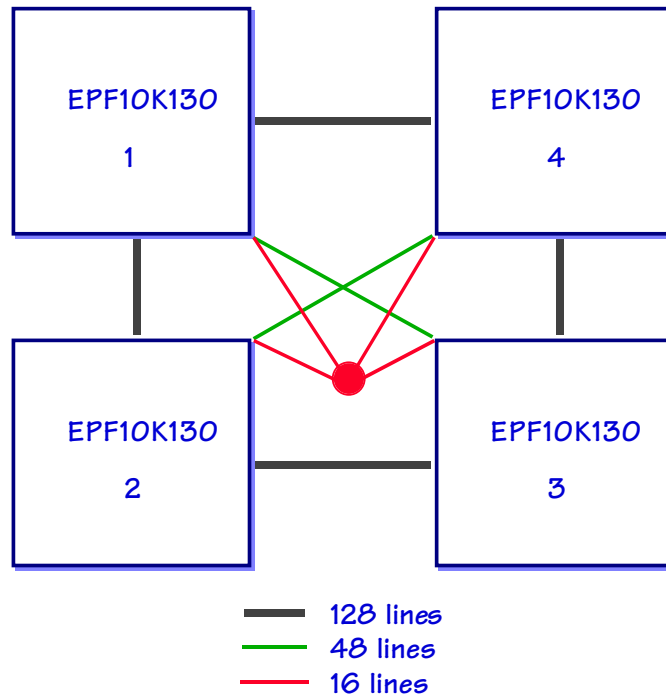


■ Reconfigurable part:

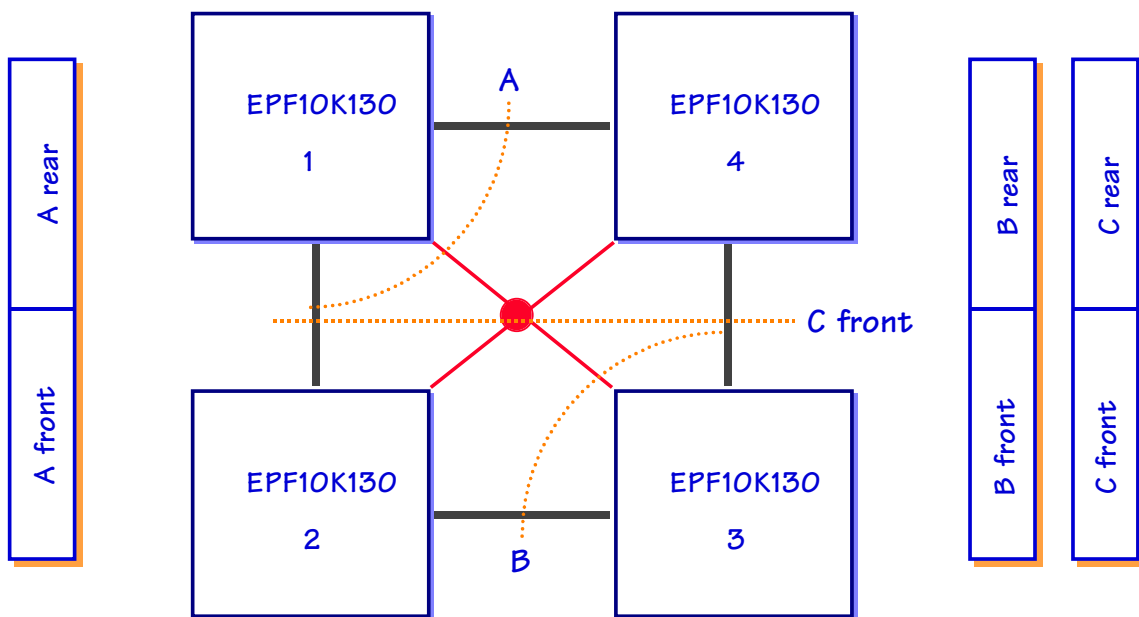
- ◆ four EPF10K130 (520'000 gates)
- ◆ future upgrade to EPF10K250 (1'000'000 gates)
- ◆ highly interconnected
- ◆ SRAM (512KB, 8 bits)
- ◆ DRAM (one SIMM socket, up to 32MB, 32 bits)
- ◆ wide read (128 bits)
- ◆ programmable clock frequency (320KHz to 100MHz)
- ◆ extension slots



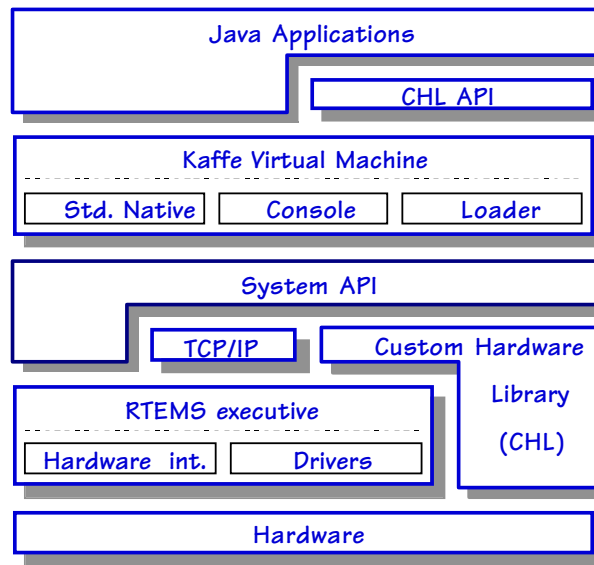
FPGA Interconnection



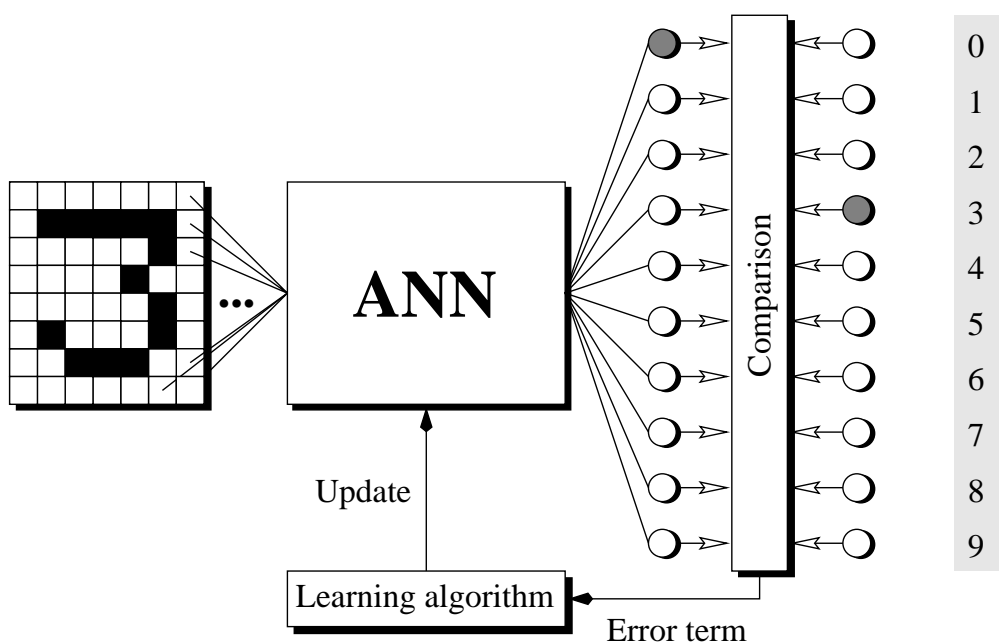
Extension slots



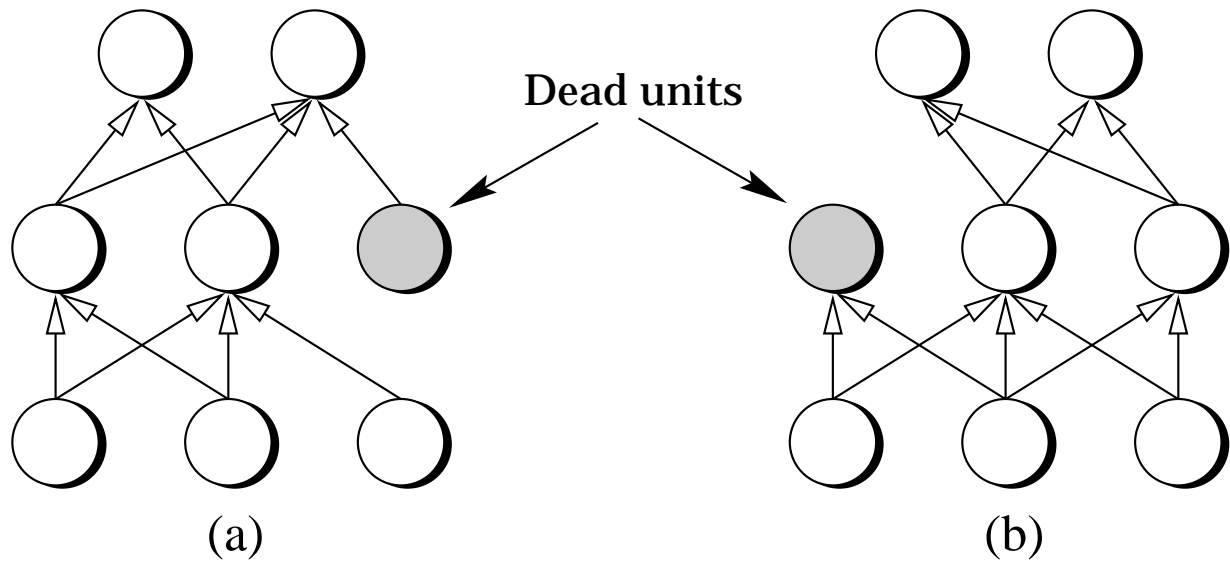
RENCO software layers



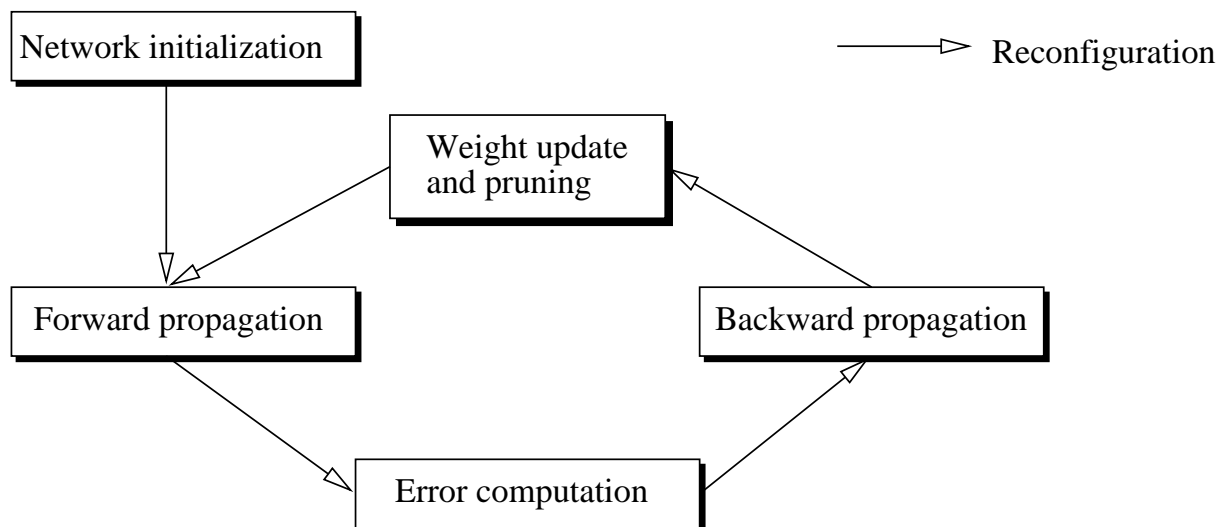
Supervised learning principles



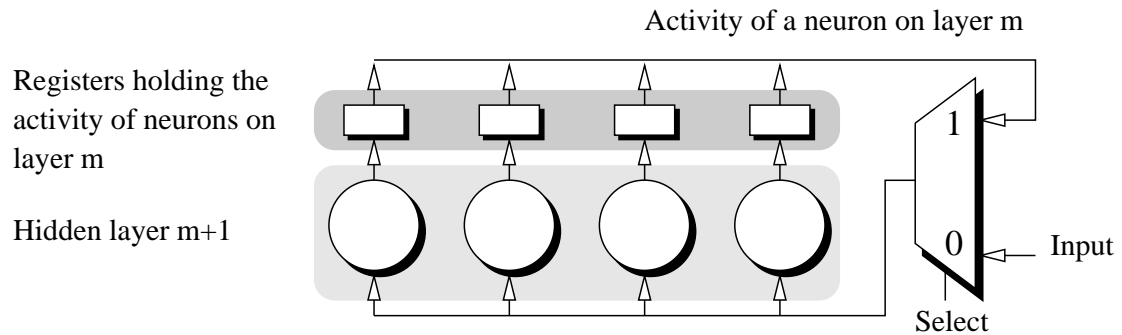
Networks with dead units



A decomposition of the backprop algorithm



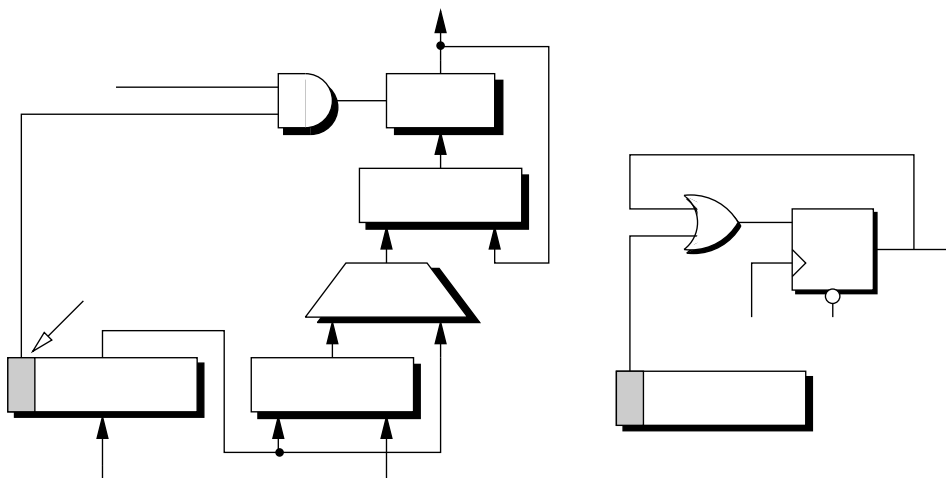
The time multiplexed interconnection scheme



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Architecture of a neuron and dead unit detection mechanism



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Dynamic systems

- The main objective of evolvable hardware is the design and implementation of hardware systems with the possibility of adaptation to changing tasks in a complex and dynamic environment.
- To accomplish this objective, the phylogenetic axis of bio-inspired systems uses two techniques: the genetic algorithms and the complex programmable devices (more specifically the FPGAs or *Field Programmable Gate Arrays*)



Evolvable hardware

- Up to date, the work that has been done under the label evolvable hardware is most closely related to Evolutionary Circuit Design.
- The forerunner in this domain is J. Koza, the father of genetic programming (GP). In his early works he used GP to synthesize a multiplexor and an adder.
- Currently, Koza uses the same techniques to synthesize analogic circuits.
- One can classify present works in evolvable hardware in accordance with the genome encoding and the fitness calculation method.



Example: a multiplexor with 3 control variables

- $F = \{\text{and, or, not, if}\}$
 $T = \{a_0, a_1, a_2, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$
- interesting case because the search space is computable:
 $2^{2^k+2^k} = 2^{2^3+2^3} = 2^{2048} \approx 10616$
- The *fitness* is the number of correct outputs (out of the 2048 possible combinations of the 11 input variables)
- Size of the population: 4000
- Stop criteria: achieve an optimal *fitness* or get to generation 51
- The initial generation achieves *fitness* between 768 and 1280



- One solution after 9 generations:

```
(if a0 (if a2 (if a1 d7 (if a0 d5 d0))
      (if a0 (if a1 (if a2 d7 d3) d1) d0))
 (if a2 (if a1 d6 d4)
      (if a2 d4 (if a1 d2 (if a2 d7 d0))))))
```



Genome encoding

- It should be possible to obtain a real circuit from the genome of the. If we use programmable circuits, it means to obtain the device configuration bit string from the genome.
- The easiest solution is to use as genome the device configuration bit string. The disadvantage of this approach is the size of the bit string, which can be in the order of thousands of bits. Furthermore, most of such bit strings may correspond to unusable circuits.
- The XC6200 FPGA family gets ride off the low-level encoding disadvantages. This FPGA is partially configurable, so the genome size can be considerably reduced, and any configuration bit string corresponds to a valid circuit.



- Another approach is to use as genome a certain description of the circuit.
- This description can be as simple as a list of interconnection of basic elements or a high-level language like VHDL.
- In this case, the disadvantage is the necessary transformation of the description into the bit string configuration



Fitness calculation

- The fitness can be calculated *offline* or *online* , if one uses the real circuit or a simulated one.
- When one uses a high-level representation to code the genome, the fitness is calculated by simulation. The genome of the final solution is then transformed into the real circuit configuration bit string. This is what is known as offline evolvable hardware.
- When one uses directly the configuration bit string as genome, one can use the real circuit during the evolution. This is what we call online evolvable hardware.



Common features of current evolvable hardware implementations

- The evolution is done with a predefined objective: the design of a circuit with a precise specification.
- A real population does not exist. In some applications, a single circuit sequentially implements every possible configuration (the population) in order to compute the corresponding fitness.
- The lack of a real population gets very difficult the interaction between individuals, and, sometimes it does not even exist. The result is a completely local fitness calculation.



- When dealing with a digital synthesis problem, it is not possible to allow approximations.

The evolutionary selection of possible solutions is achieved by comparing the current results with a table representing the desired result. Since this table is itself a description of the solution being searched, the evolutionary synthesis of digital systems is questionable.

- The basic operations of the evolutionary process (genetic operations and fitness calculation) are realized offline.
- The different steps of the evolution process are sequentially realized under the control of a centralized program.



Classes of evolvable hardware

- It is possible to divide the phylogenetic axis in 4 different classes:
- The first class corresponds to the evolutionary design of circuits: every single operation is carried out in software. The final solution is then implemented as a real system.
- In the second class a real circuit is used during all the evolutionary process. In particular, the fitness is computed in a real environment. However, most of the genetic operations are carried out by software.



- A third class refers to evolutionary processes where the genetic operations and the fitness calculation are realized online using the corresponding hardware. Only two characteristics of natural evolution are not present: an open-ended evolution (evolution without a predefined objective) and a dynamic environment. The firefly machine, developed at the Logic Systems Laboratory is an example of this class.
- Finally, we find a population of hardware organisms evolving in an open-ended manner in a dynamic environment. Only the hardware systems belonging to this class can be truly called evolvable hardware. Unfortunately, to our knowledge, there is not exist one of such systems yet.



The Firefly machine

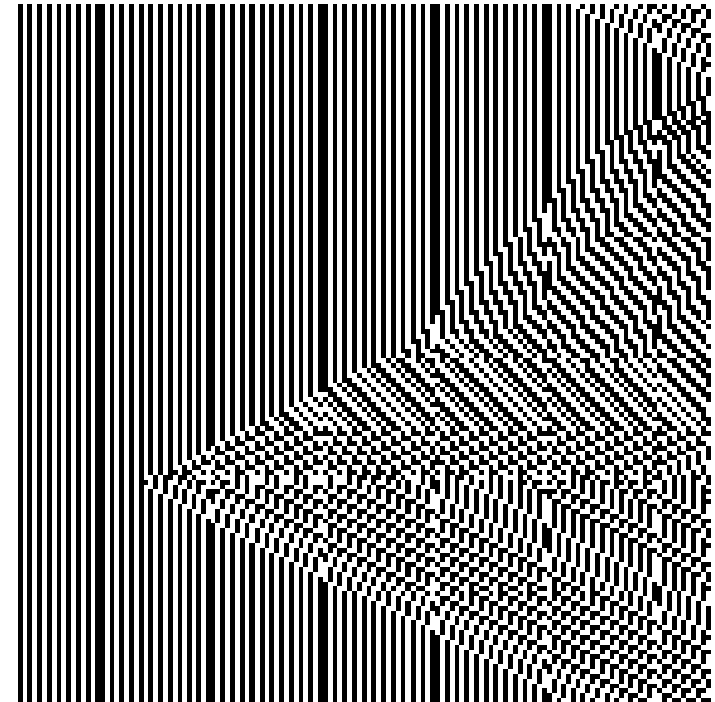
- The Firefly machine implements an online evolvable cellular automata by means of X4000 FPGAs
- The cellular automata is heterogeneous and one-dimensional
- The cellular automata starts with a random configuration and realizes a synchronization task: after a finite number of steps, every cell oscillates between 0 and 1 synchronously.
- The future state of each cell depends on its preceding state and the state of its two neighbors: its state machine is therefore described by an 8-bit table, which is the genome of the cell.
- All the evolutionary operations are realized in hardware without any contact with any other external machine.



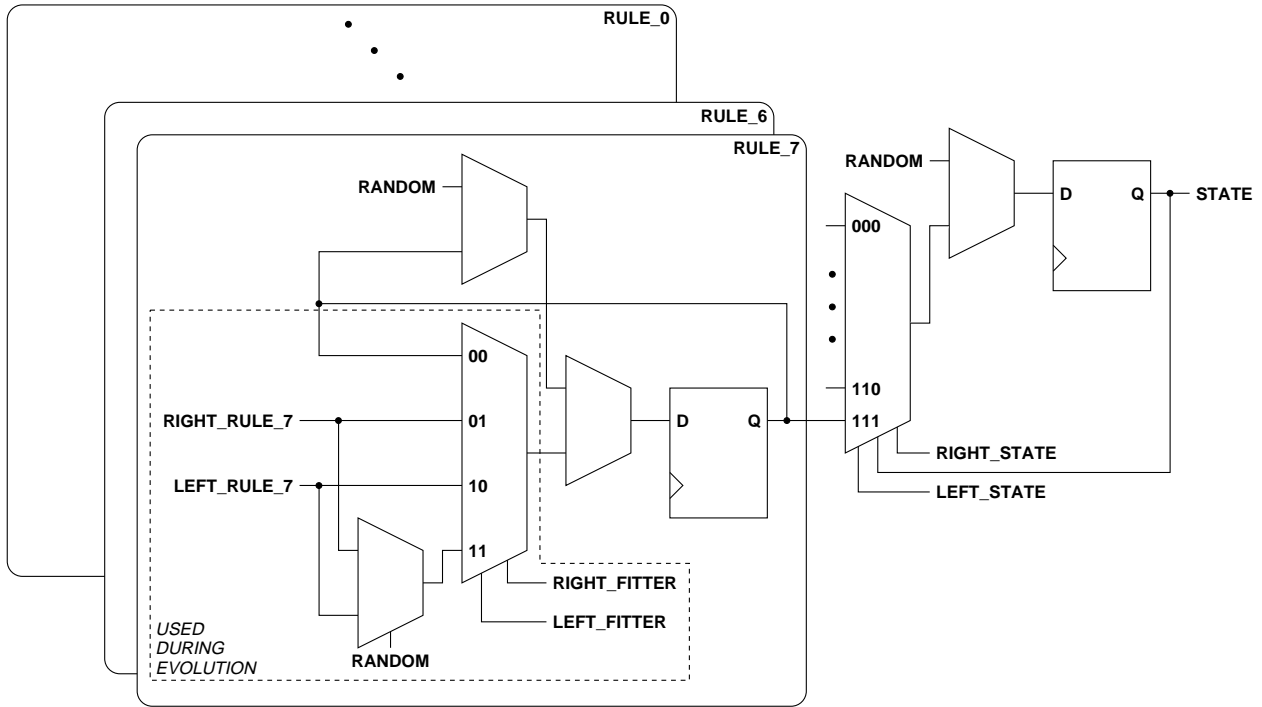
```

for each cell  $i$  in CA do in parallel
  initialize rule table of cell  $i$ 
   $f_i = 0$  {fitness value}
end parallel for
 $c = 0$  {initial configurations counter}
while not done do
  generate a random initial configuration
  run CA on initial configuration for  $M$  time steps
  for each cell  $i$  do in parallel
    if cell  $i$  is in the correct final state then
       $f_i = f_i + 1$ 
    end if
  end parallel for
   $c = c + 1$ 
  if  $c \bmod C = 0$  then {evolve every  $C$  configurations}
    for each cell  $i$  do in parallel
      compute  $nf_i(c)$  {number of fitter neighbors}
      if  $nf_i(c) = 0$  then rule  $i$  is left unchanged
      else if  $nf_i(c) = 1$  then replace rule  $i$  with the
        fitter neighboring rule
      else if  $nf_i(c) = 2$  then replace rule  $i$  with the
        uniform crossover of the two fitter
        neighboring rules
      end if
       $f_i = 0$ 
    end parallel for
  end if
end while

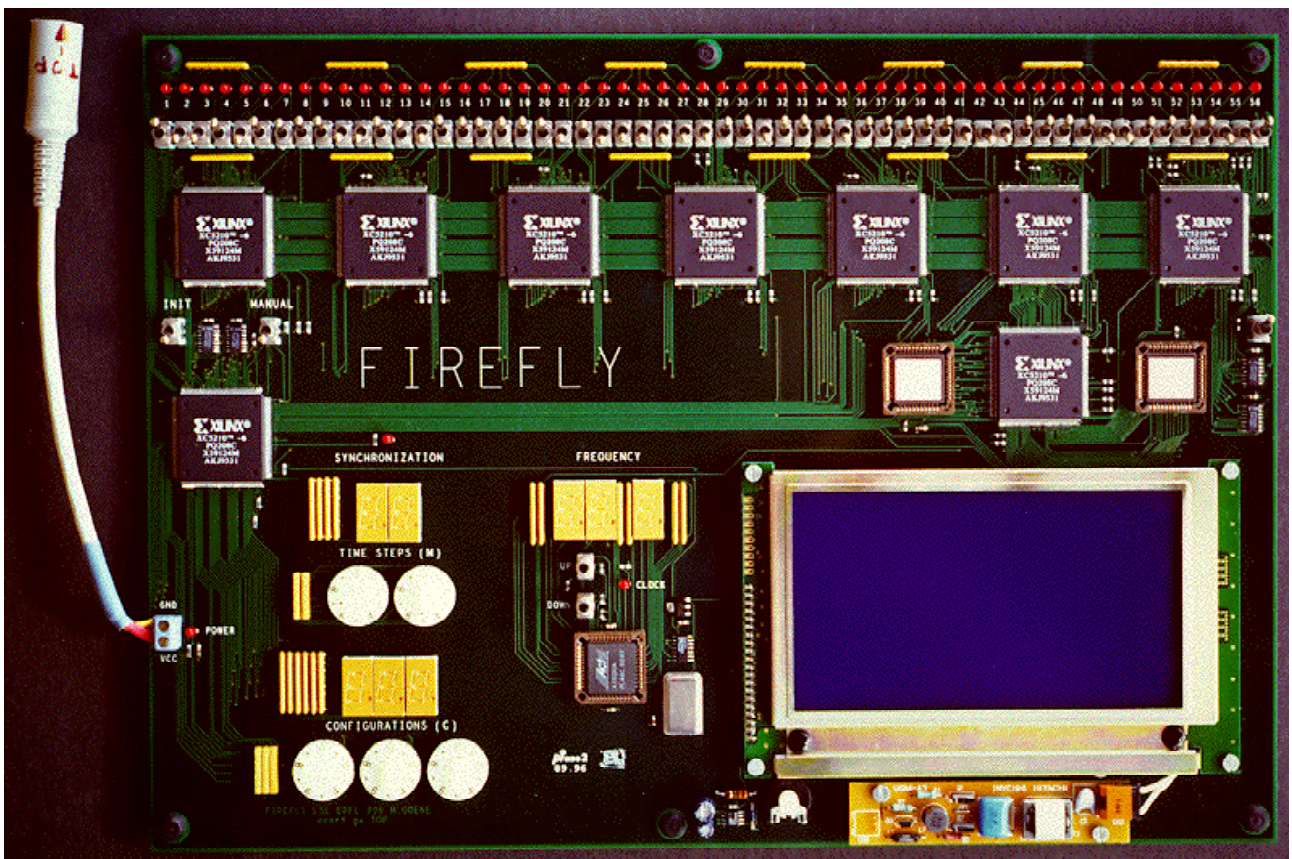
```



Circuit design of a Firefly cell



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

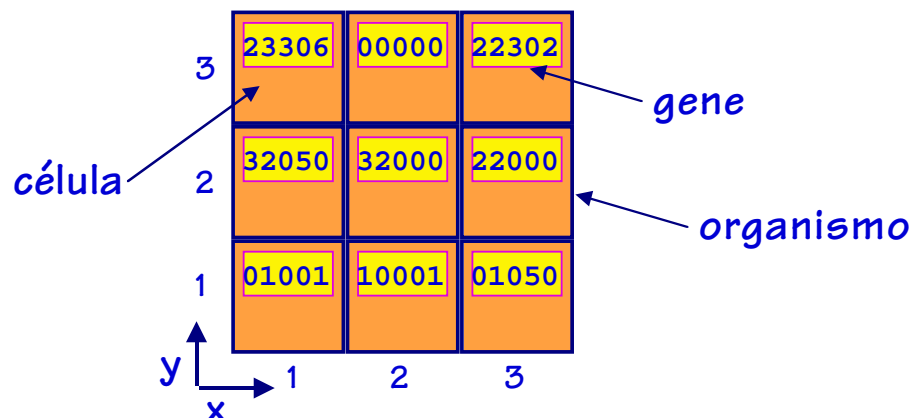


Objetivo

- Realización de circuitos integrados con ciertas propiedades de los seres vivos (**autoreproducción** y **autoreparación**, por ejemplo)
- Principios de base:
 - ◆ Organización multicelular
 - ◆ Diferenciación celular
 - ◆ División celular

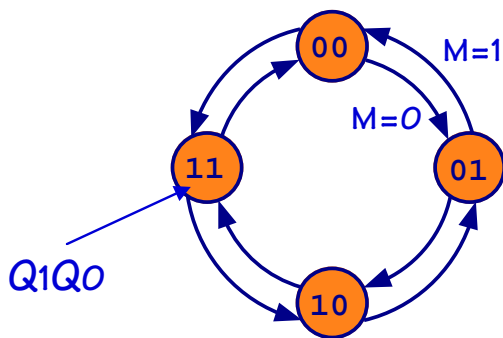
Organización multicelular

- El organismo (un sistema lógico) está dividido en un número finito de células idénticas. Cada célula realiza una función particular, determinada por el gene de la célula



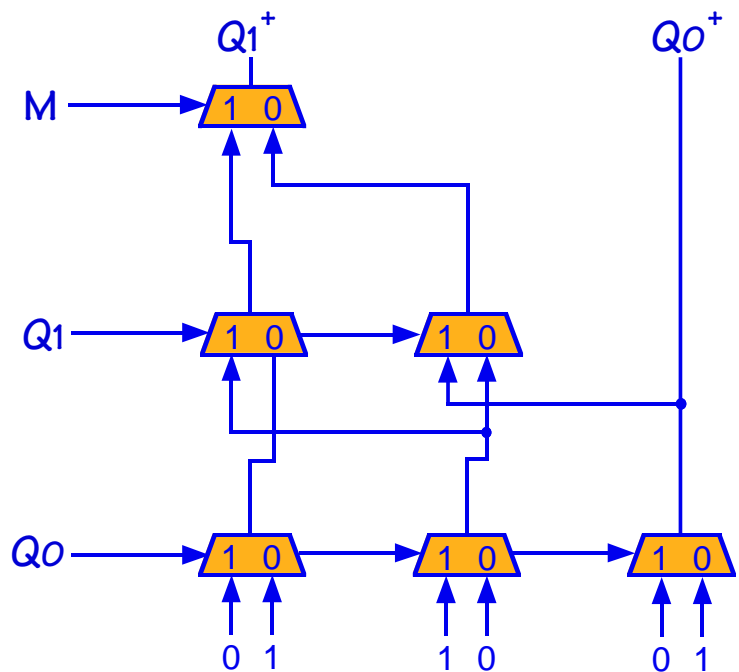
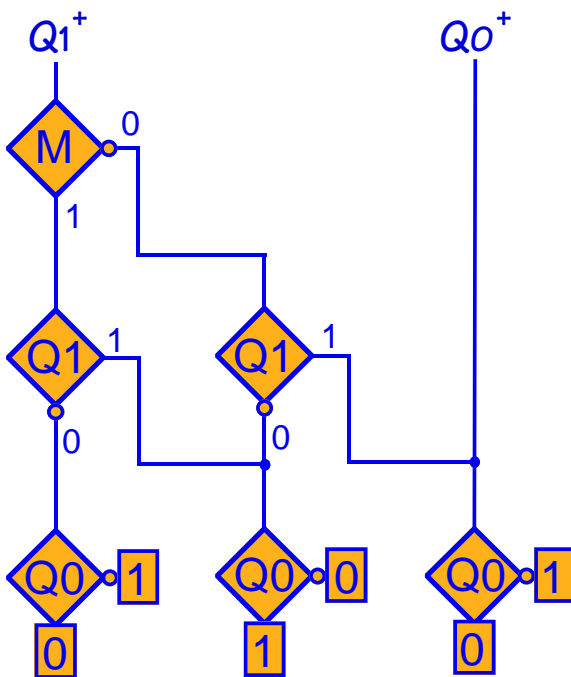
Algoritmos de decisión binaria

- Cualquier función lógica puede ser realizada por un algoritmo de decisión binaria, interconexión de tests de variables binarias
- Un multiplexor es la realización material de un test binario
- Ejemplo: contador reversible, dos bits



$Q_1^+ Q_0^+$	M	
	0	1
00	01	11
01	10	00
10	11	01
11	00	10
Q_1Q_0		

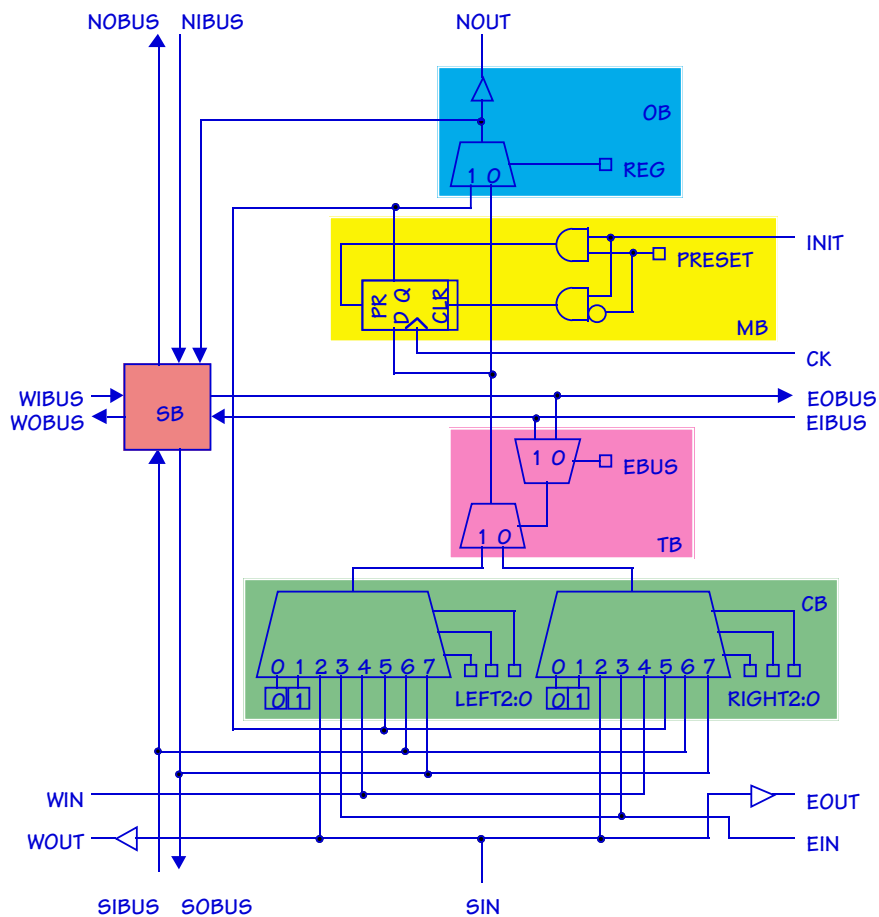
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

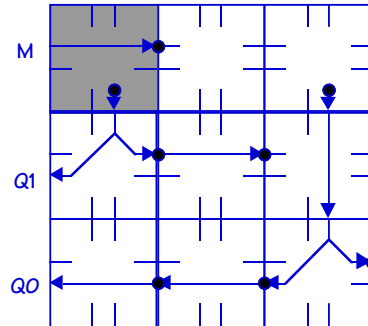
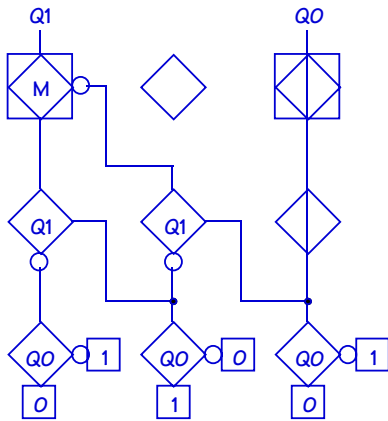
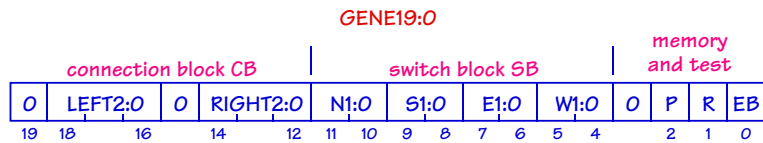


Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



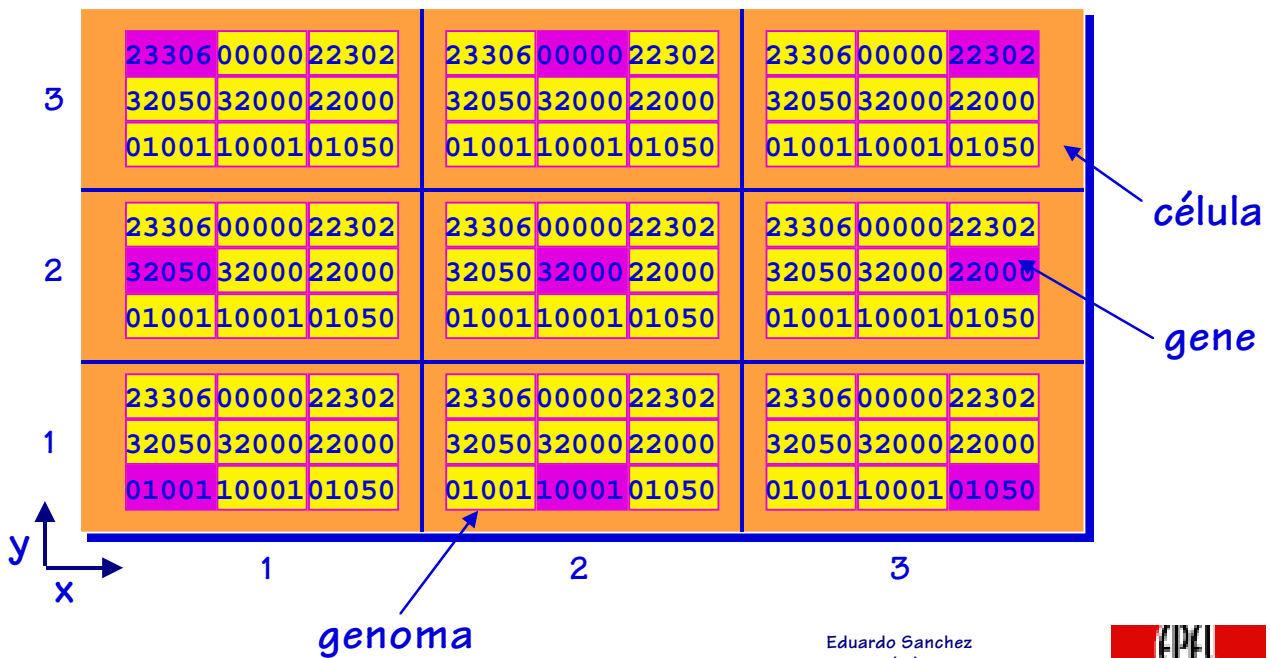
- Cada célula de nuestro sistema debe ser capaz de realizar un test binario
- La parte funcional de nuestra célula estará compuesta de un multiplexor más las líneas de interconexión con las células vecinas





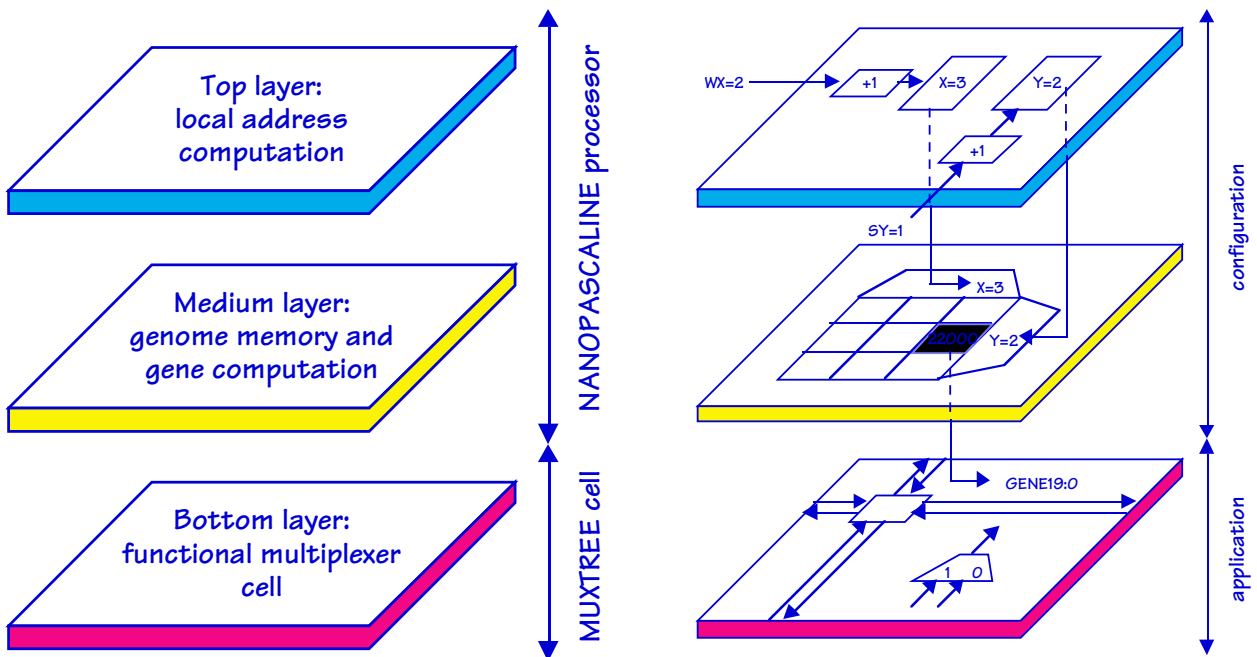
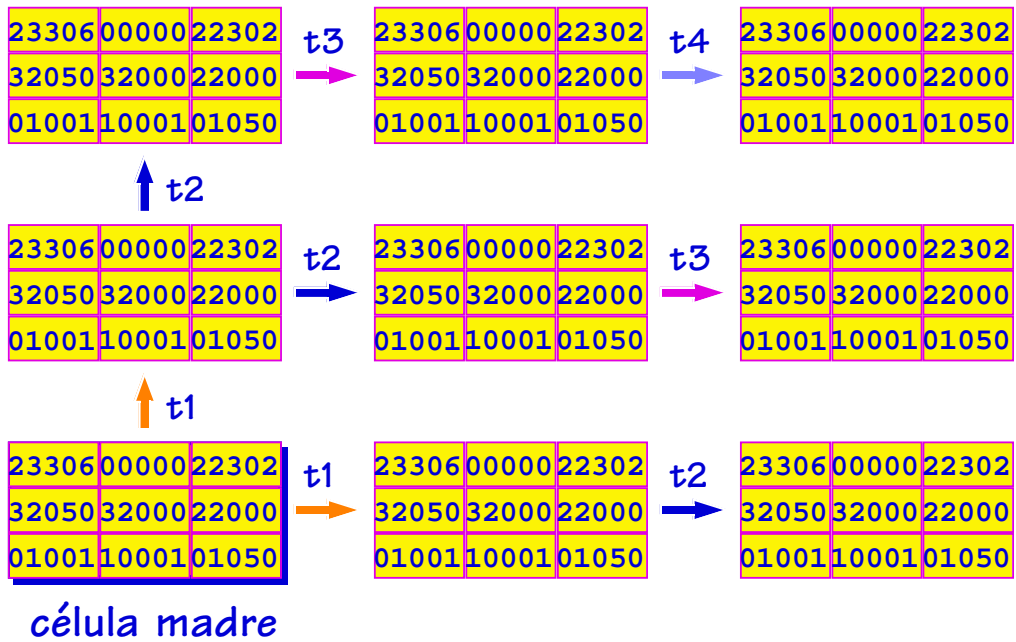
Diferenciación celular

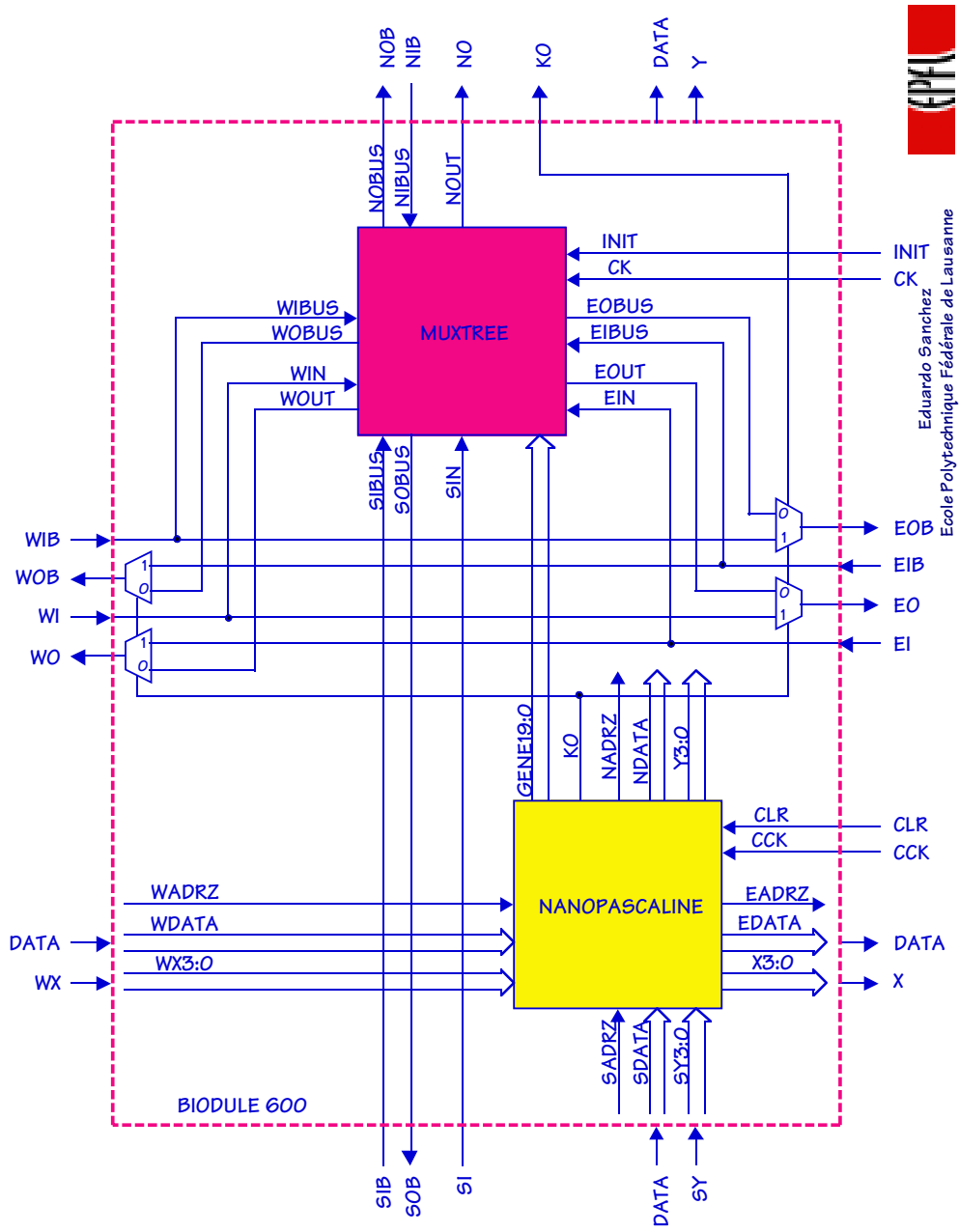
- Cada célula contiene el genoma completo, lo cual la hace universal. Cada célula puede ejecutar cualquier gene, en función de su coordenada



División celular

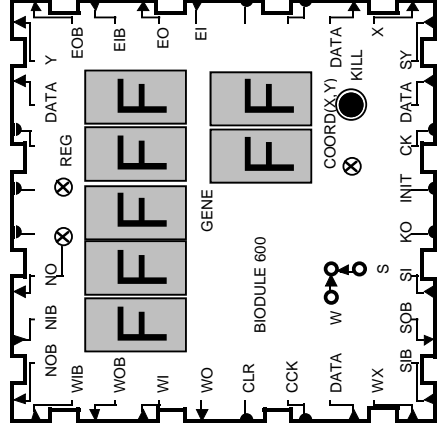
- El organismo se construye a partir de una célula madre (zigoto), situada en la coordenada (1,1)





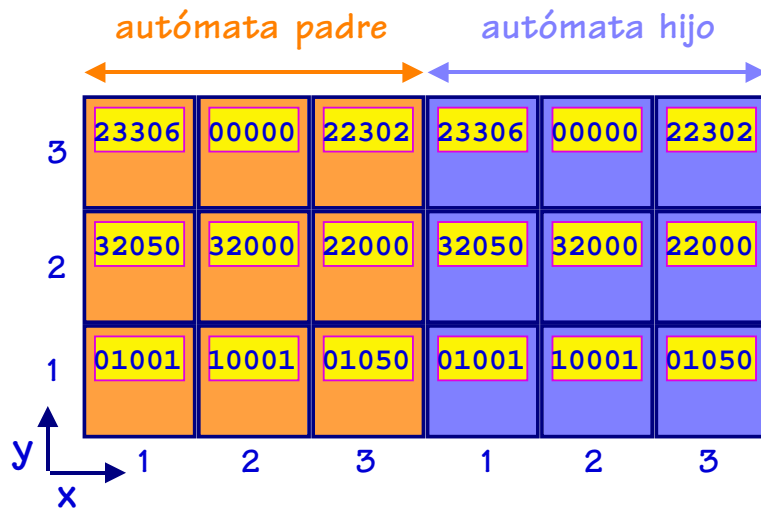
Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

EI BIODULE

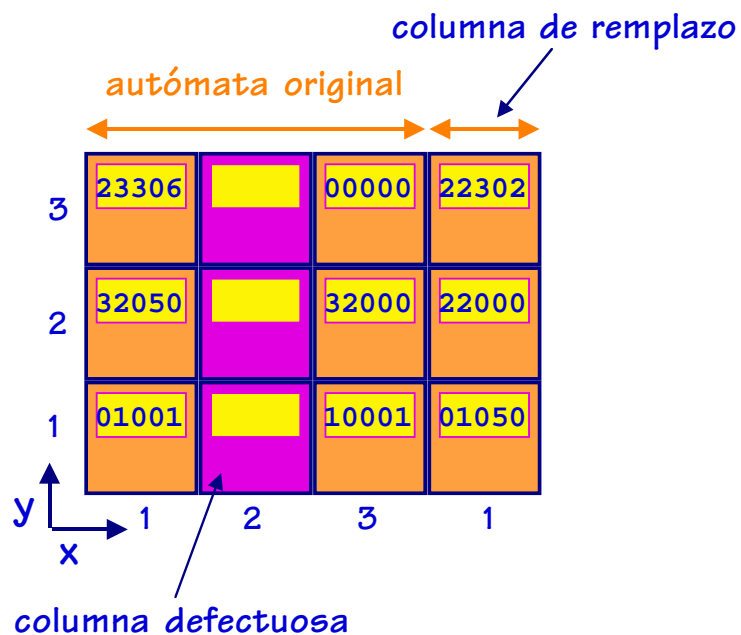


Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne

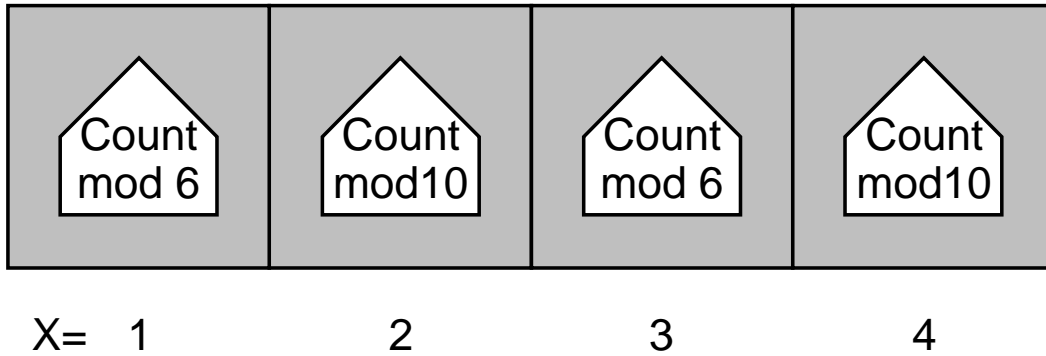
Autoreproducción



Autoreparación



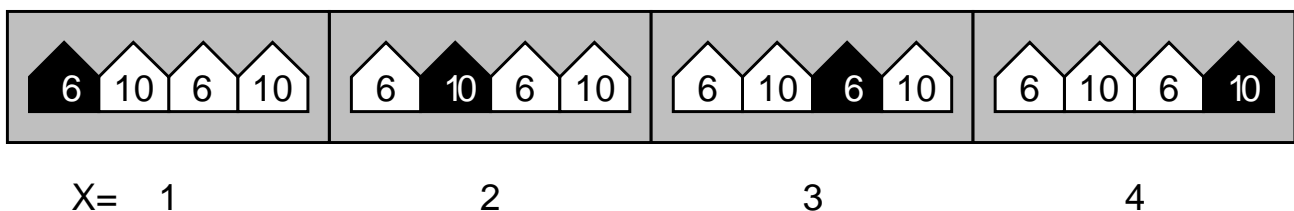
Multicellular organization of the BioWatch



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



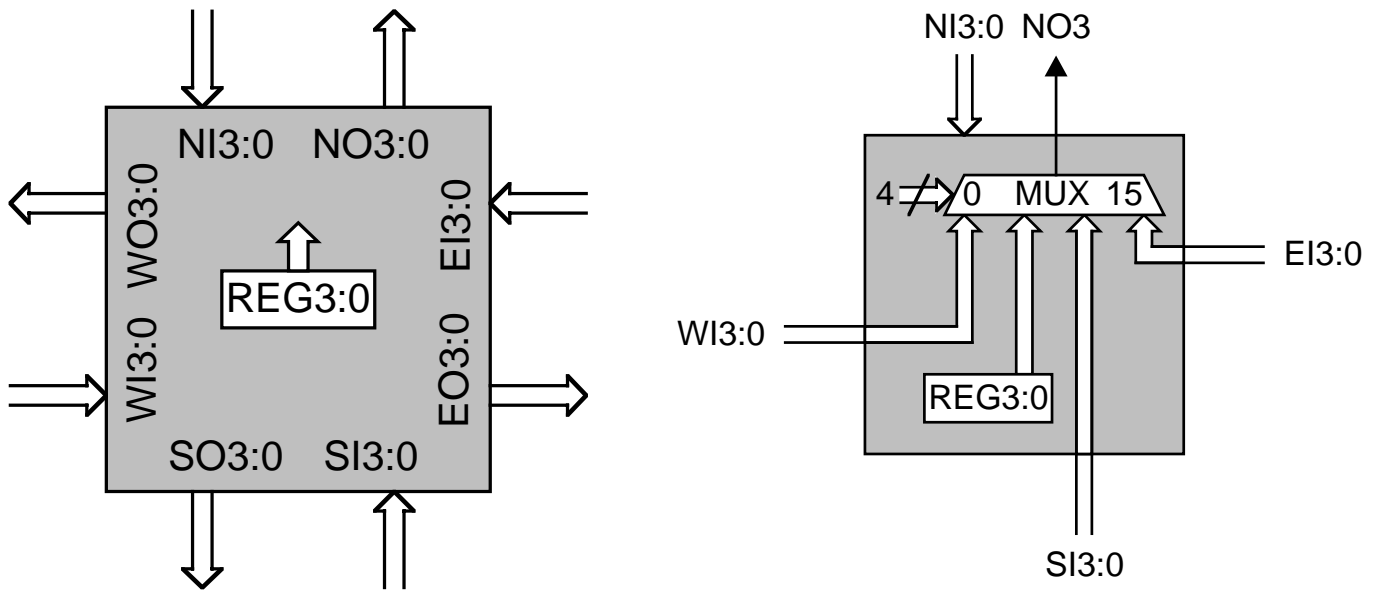
Cellular differentiation of the BioWatch



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



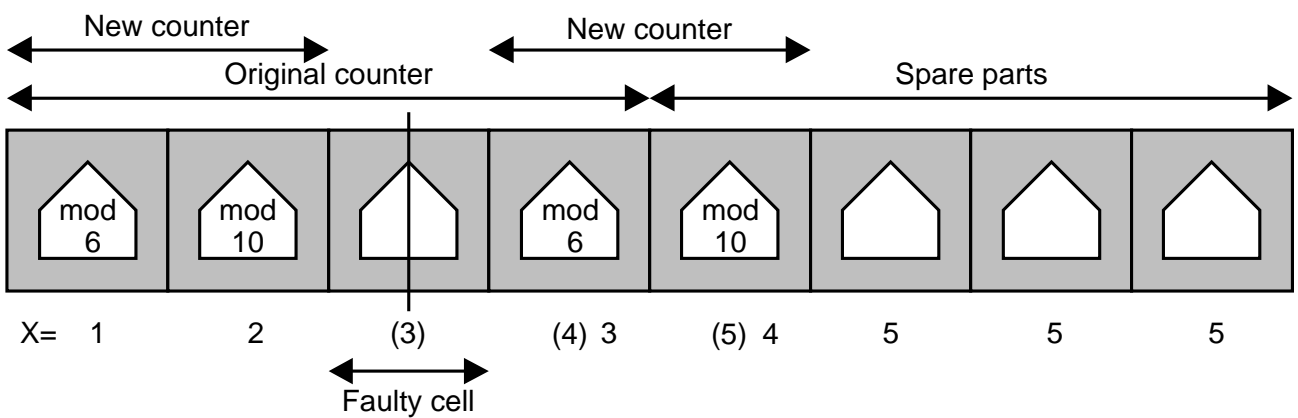
MICTREE cell



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



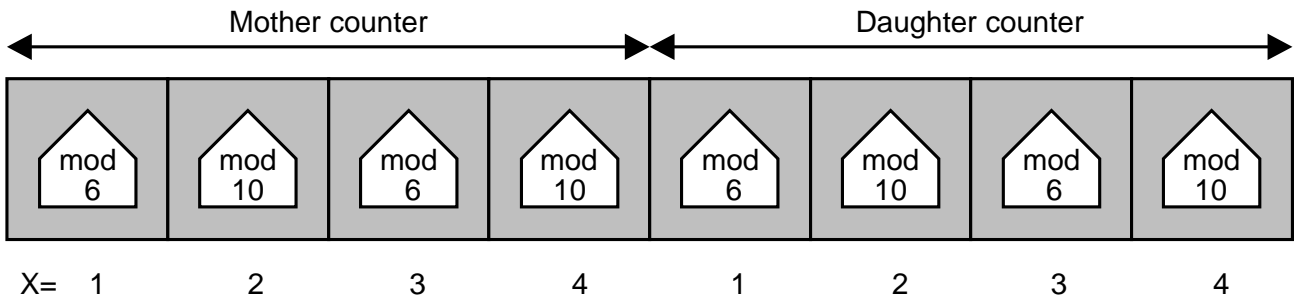
Self-repair of the BioWatch



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Self-replication of the BioWatch



Architecture of the FAST neuron

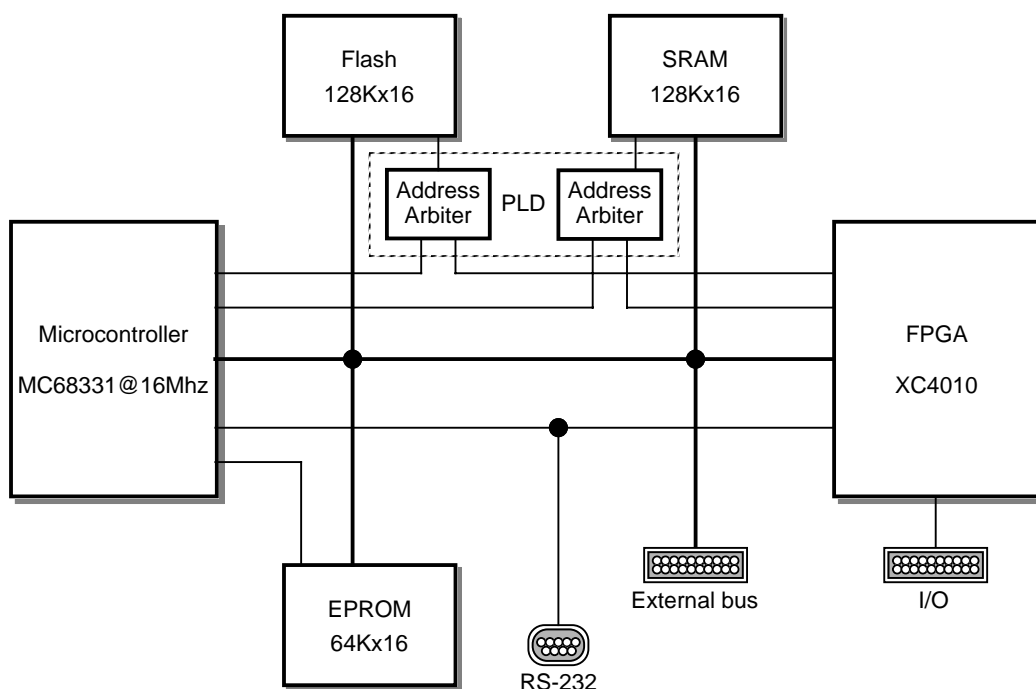
Codesign: the need for a change in the engineering curriculum

- In the classical university curriculum there is a hard distinction between programmable computing and configurable computing:
 - ◆ the programmable computing, considered as a software practice, is under the responsibility of the computer science department
 - ◆ the configurable computing is regarded as a hardware practice, thereby to be taught at the electrical engineering department
- We believe that this clear-cut frontier is dissolving, with these dichotomous domains slowly merging into a continuum. One prominent aspect of this fusion is the increasing importance attached to the **codesign** issue: the decision of which parts of the application are to be designed as software and which shall be designed directly as hardware

Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



LABOMAT architecture



Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne



Un système est vivant à deux conditions:

- il est capable d'auto-reproduction
- il est capable d'évoluer sans but prédéfini

Sans auto-reproduction il n'y a pas d'évolution sans but prédéfini: la sélection est effectuée par un contrôleur central et les individus ne peuvent pas inventer leurs propres fonctions de fitness

Core War

- Une mémoire circulaire à 8000 mots
- Un répertoire d'instructions (Redcode)
- Un programme de contrôle, MARS (Memory Array Recode Simulator)
- Deux programmes qui s'affrontent
- Les deux programmes sont chargés à des positions aléatoires, mais pas plus proches de 1000 positions
- MARS exécute alternativement une instruction de chaque programme, jusqu'à ce qu'il trouve un code non exécutable. Le programme avec le code incorrect perd la partie.

Redcode (répertoire d'instructions pour Core War)

MOV	A, B	M[PC+A] → M[PC+B]
MOV	@A, B	M[M[PC+A]+PC+A] → M[PC+B]
MOV	#A, B	A → M[PC+B]
ADD	A, B	M[PC+A] + M[PC+B] → M[PC+B]
SUB	A, B	M[PC+B] - M[PC+A] → M[PC+B]
JMP	A	PC + A → PC
JMZ	A, B	if M[PC+B]=0 then PC + A → PC
JMG	A, B	if M[PC+B]>0 then PC + A → PC
DJZ	A, B	M[PC+B] - 1 → M[PC+B],
CMP	A, B	if M[PC+B]=0 then PC + A → PC
DAT	B	if M[PC+A]<> M[PC+B] then skip next
SPL	A	B is a data value

split execution into next instruction
and the instruction at M[PC+A]

EPFL - LSL

Eduardo Sanchez

Tierra

3

Exemple

DAT	-1
ADD	#5, -1
MOV	#0, @-2
JMP	-2

Chaque cinquième adresse la constante 0 est stockée: c'est le code d'une instruction non exécutable. Pour éviter le suicide, il faut placer le programme à l'adresse 1.

EPFL - LSL

Eduardo Sanchez

Tierra

4

Exemple

DAT	0	pointeur à la source
DAT	99	pointeur à la destination
MOV	@-2,@-1	copie la source dans la destination
CMP	-3,#9	si toutes les 10 lignes ont été copiées...
JMP	4	... on quitte la boucle;
ADD	#1,-5	si non la source est incrementedée...
ADD	#1,-5	... ainsi que la destination...
JMP	-5	... et on recommence la boucle
MOV	#99,93	la destination est restaurée
JMP	93	et on passe à la nouvelle copie

Le programme se copie lui-même 100 positions plus loin et le contrôle est passé à la nouvelle copie.

Exemple

```
SPL 2
JMP -1
MOV 0,1
```

C'est une boucle de génération du programme auto-reproducteur le plus simple possible

Dans le cas d'un programme à plusieurs branches, une seule instruction d'une branche est exécutée dans le tour du programme: plus on a des branches, plus lentement elles sont exécutées. Mais un programme est déclaré perdant seulement si toutes ses branches sont mortes.

Vie organique: utilisation de l'énergie pour organiser la matière

*Vie digitale: utilisation du temps CPU pour organiser la mémoire
Des algorithmes auto-reproducteurs entrent en
compétition pour le temps CPU et l'espace mémoire.
Des stratégies sont développées pour exploiter
l'environnement (mémoire, CPU, OS)
Un individu est un programme auto-reproducteur*

EPFL - LSL

Eduardo Sanchez

Tierra

7

*Langage organique: alphabet de 4 nucléotides
 des groupes de 3 nucléotides forment des mots (codons)
 des 64 codons possibles, 20 sont traduits en amino-acides*

*Langage digital: seulement 32 instructions, codées en 5 bits
 les opérandes sont éliminés
 des patterns sont utilisés pour trouver l'adresse de saut*

EPFL - LSL

Eduardo Sanchez

Tierra

8

Vie organique: la membrane de la cellule définit ses limites et préserve son intégrité chimique

Tierra: chaque individu est protégé par une membrane semi-perméable d'allocation de mémoire

Chaque individu possède des privilèges exclusifs d'écriture dans son bloc de mémoire. Par contre, les privilèges de lecture et d'exécution ne sont pas protégés. Le privilège d'écriture s'étend aussi sur le bloc fils, au moment de l'instruction MAL (memory allocation). Mais ce privilège est perdu au moment de la division.

EPFL - LSL

Eduardo Sanchez

Tierra

9

Mortalité

Des individus commencent à mourir à partir d'un 80% de remplissage de la mémoire.

Il existe une queue avec tous les individus classés par nombre d'erreurs d'exécution de leur programme.

EPFL - LSL

Eduardo Sanchez

Tierra

10

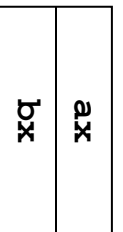
Mécanismes d'évolution

Deux types de mutation:

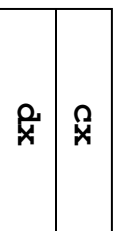
- modification aléatoire de la mémoire
- erreur aléatoire de copie lors de la reproduction

Exécution non déterministe de certaines instructions

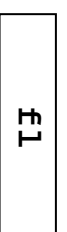
Architecture du processeur



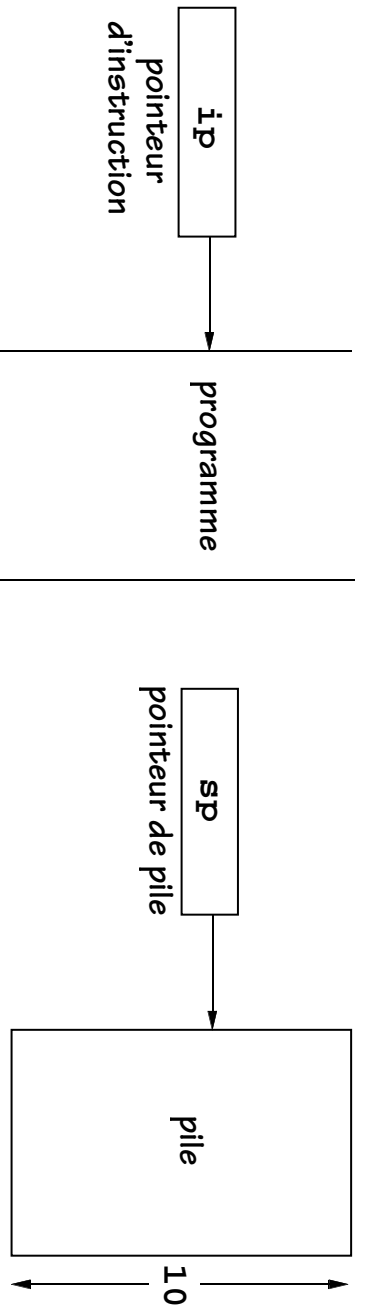
registres d'adresse



registres de données



flags



Répertoire d'instructions

No operations:

nop0
nop1

Déplacement de données:

pushax	ax → stack	subab	ax - bx → cx
pushbx	bx → stack	subac	ax - cx → ax
pushcx	cx → stack	inca	ax + 1 → ax
pushdx	dx → stack	incb	bx + 1 → bx
popax	stack → ax	incc	cx + 1 → cx
popbx	stack → bx	decc	cx - 1 → cx
popcx	stack → cx	zero	0 → cx
popdx	stack → dx	not0	not cx[0] → cx[0]
movcd	cx → dx	shl	2*cx → cx
movab	ax → bx	Saut:	
movil	M[bx] → M[ax]	ifz	if cx = 0 execute next else skip
		jmp	jump to template
		jmpb	jump backwards to template
		call	ip → stack, jump to template
		ret	stack → ip

EPFL - LSL

Eduardo Sanchez

Tierra

13

Biologiques:

adr search outward for template, address → ax, size → cx
adrb search backward for template, address → ax, size → cx
adrf search forward for template, address → ax, size → cx
mal allocate amount of space specified in cx
divide cell division

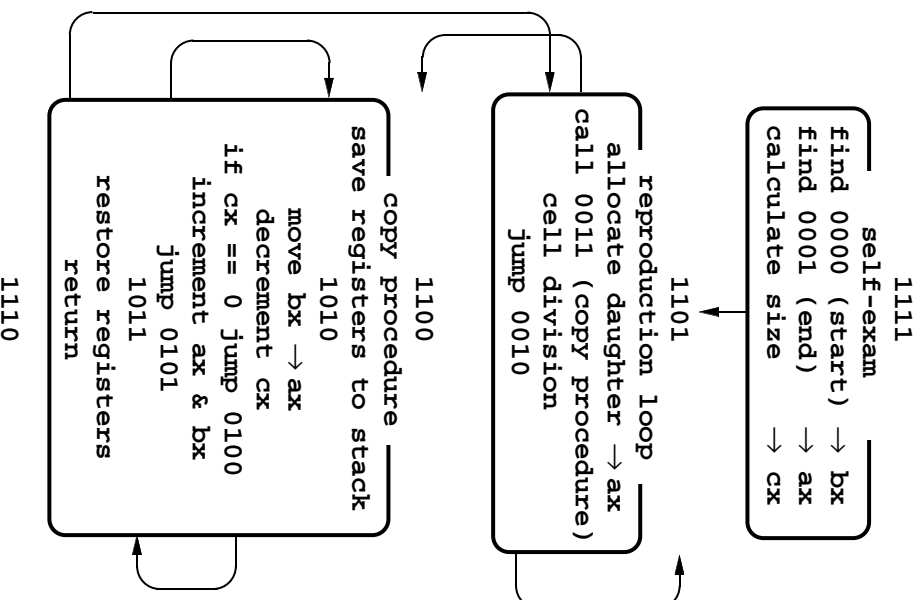
EPFL - LSL

Eduardo Sanchez

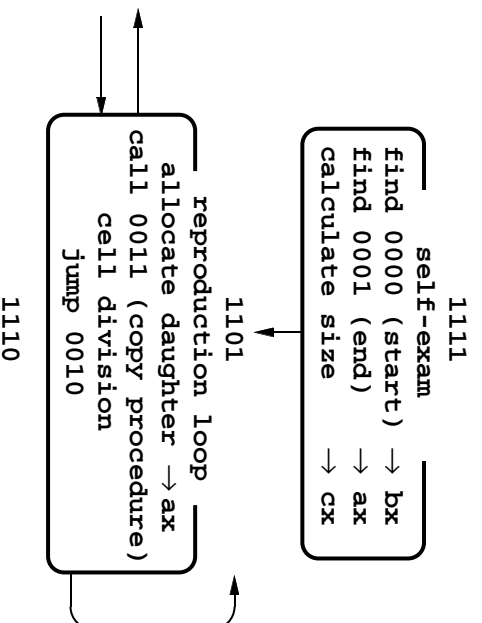
Tierra

14

L'ancêtre



Un parasite



Répertoire 4

No operations:
nop0
nop1

Déplacement des données:

movdi	bx → M[ax+cx]	inc	cx + 1 → cx
movid	M[bx+cx] → ax	dec	cx - 1 → cx
movli	M[bx+cx] → M[ax+cx]	add	cx + dx → cx
pushax	ax → stack	sub	cx - dx → cx
pushbx	bx → stack	zero	0 → cx
pushcx	cx → stack	not0	not cx[0] → cx[0]
pushdx	dx → stack	shl	2*cx → cx
popax	stack → ax	Saut:	
popbx	stack → bx	ifz	if cx = 0 execute next else skip
popcx	stack → cx	iff1	if flag = 1 execute next else skip
popdx	stack → dx	jmp	jump to template
put	dx → output buffer	jmpb	if no template jump to ax jump back to template
get	input port → dx	call	if no template jump back to ax ip + 1 → stack, jump

EPFL - LSL

Eduardo Sanchez

Tierra

17

Biologique:

adr	search outward for template address → ax size in dx, offset in cx start search at toffset
adrb	search backward for template address → ax size in dx, offset in cx start search at -offset
adrf	search forward for template address → ax size in dx, offset in cx start search at +offset
mal	allocate memory amount in cx address → ax
divide	cell division

EPFL - LSL

Eduardo Sanchez

Tierra

18

Répertoire	taille de l'ancêtre	taille de la créature minimale
11	73	22
12	94	54
13	93	34
14	82	23

efficacité = (nombre de cycles pour la reproduction)/(taille de la créature)

Pour 14:

taille	cycles	efficacité
82	688	8.4
24	95	3.9