

# Esterel de A à Z

## 6. Clock gating, multi-horloges, implémentation et optimisation

Gérard Berry

Collège de France  
Chaire Algorithmes, machines et langages

Cours du 21 mars 2018

Suivi du séminaire de Timothy Bourke



COLLÈGE  
DE FRANCE  
— 1530 —

[gerard.berry@college-de-france.fr](mailto:gerard.berry@college-de-france.fr)  
<http://www-sop.inria.fr/members/Gerard.Berry>

# *Agenda*

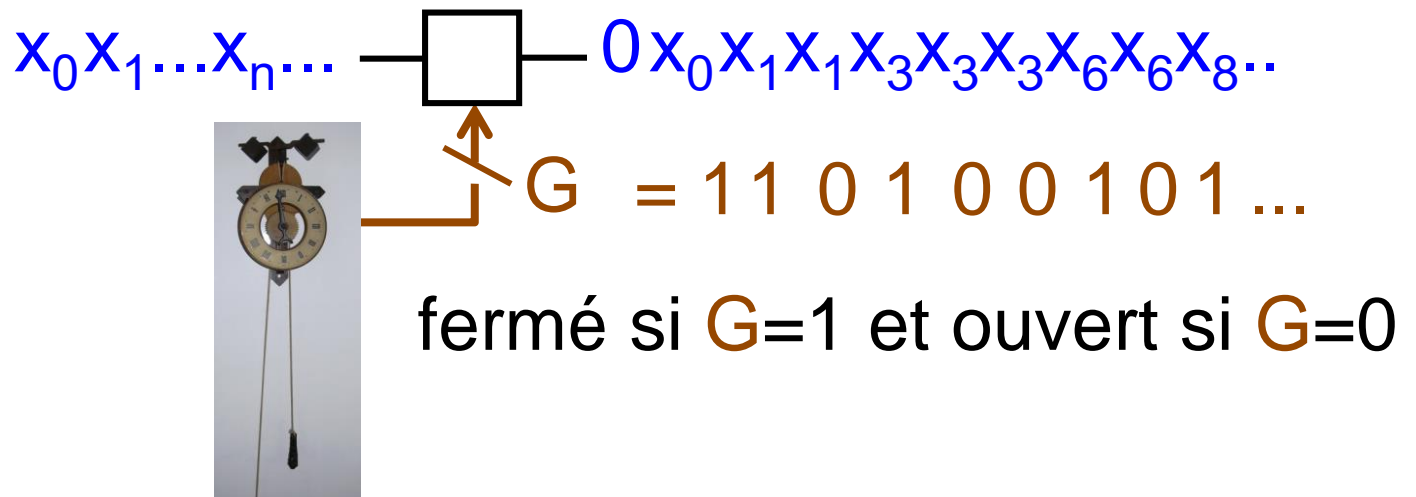
1. Weak suspend pour le clock gating

# Clock gating

Un moyen majeur de réduire la consommation électrique en réduisant les basculement inutiles de registres,

qui consomment beaucoup

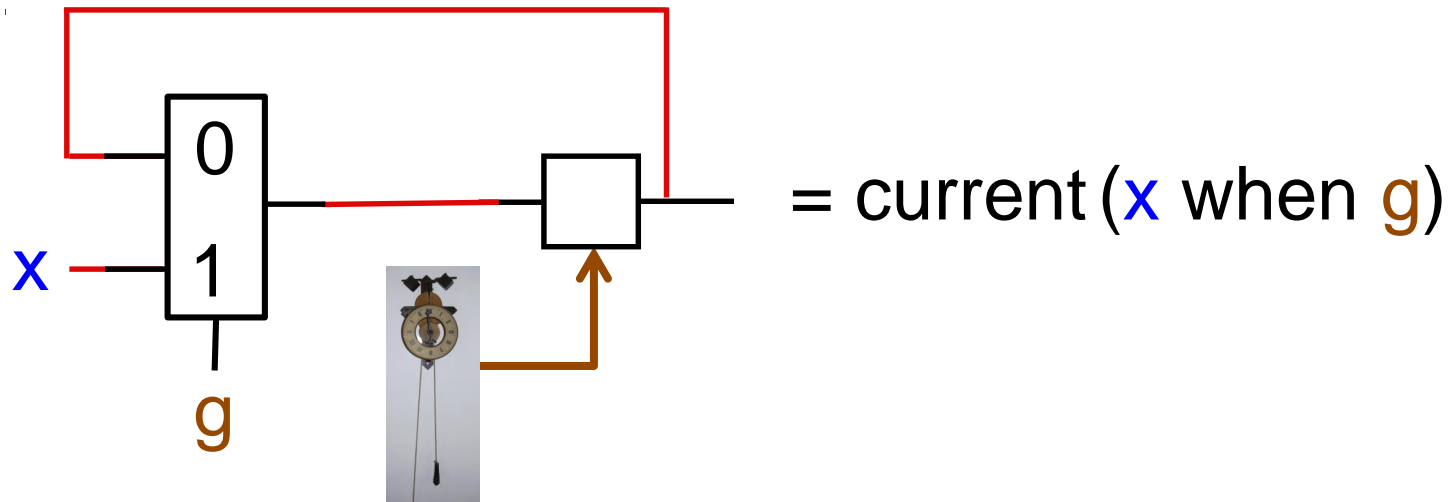
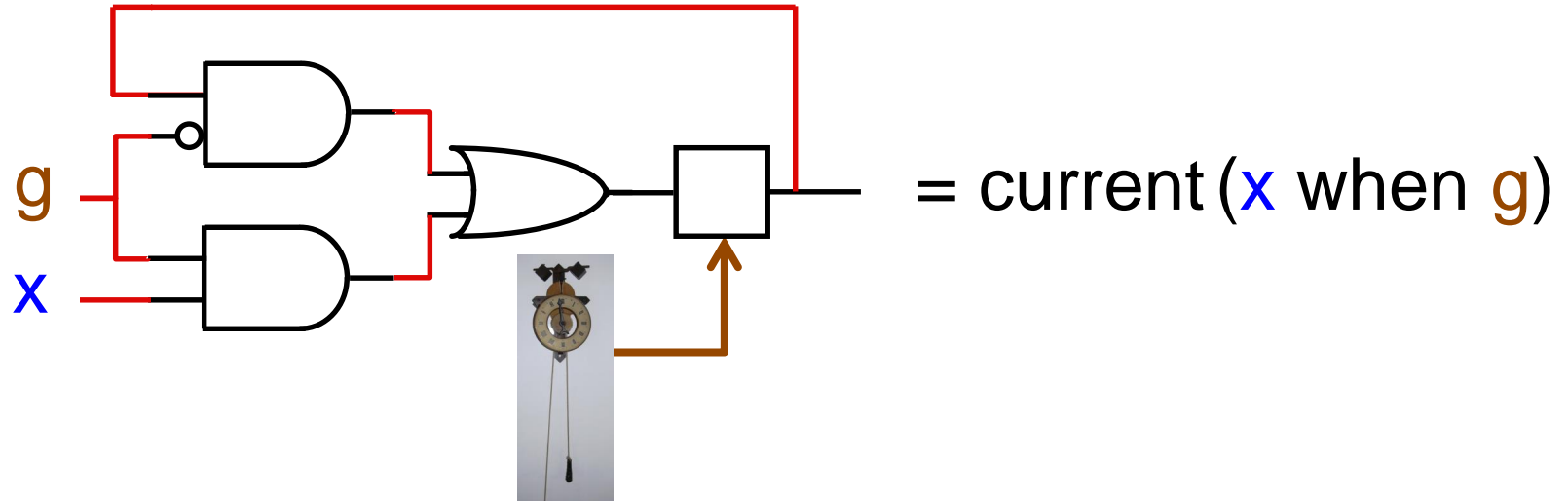
Aussi un moyen d'optimiser certains circuits (J. Vuillemin)



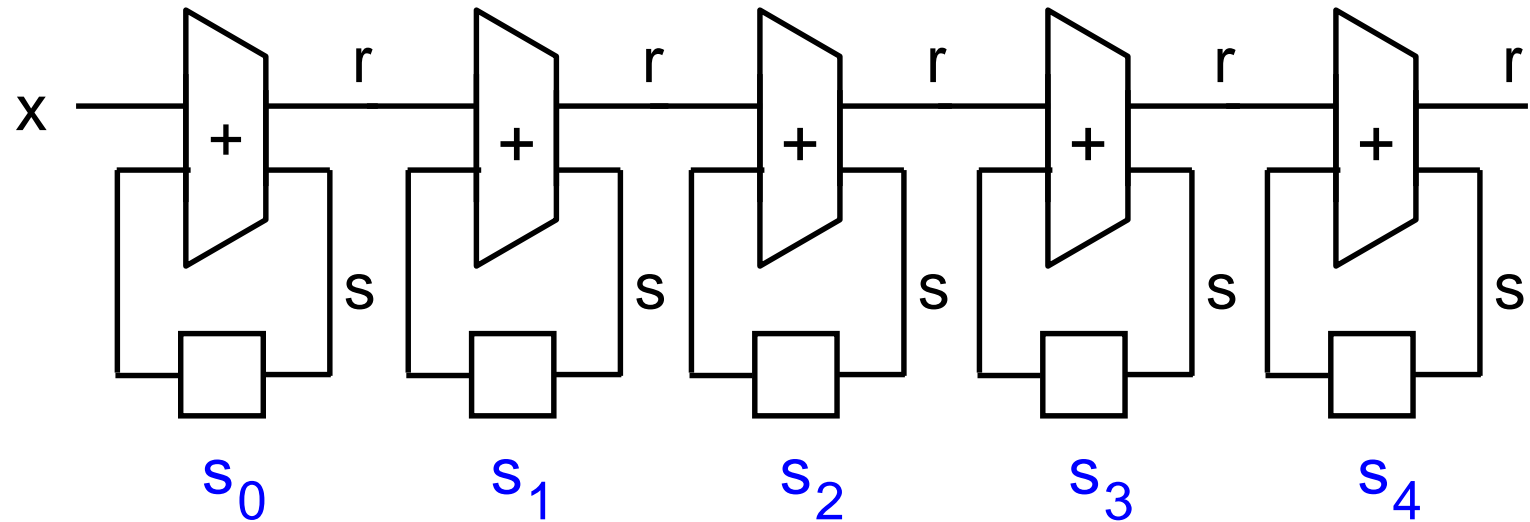
0 → current (x when G) en Lustre

N'agit que sur les registres, pas sur la logique !

# Faisable avec des portes (FPGA et simulation)



# Un compteur normal

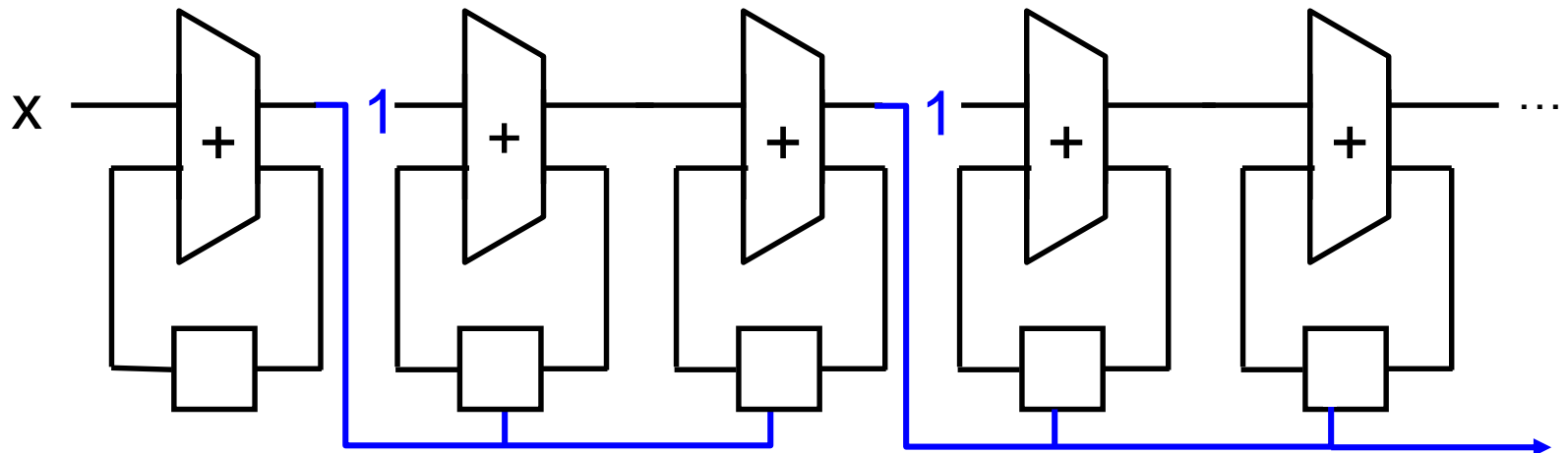
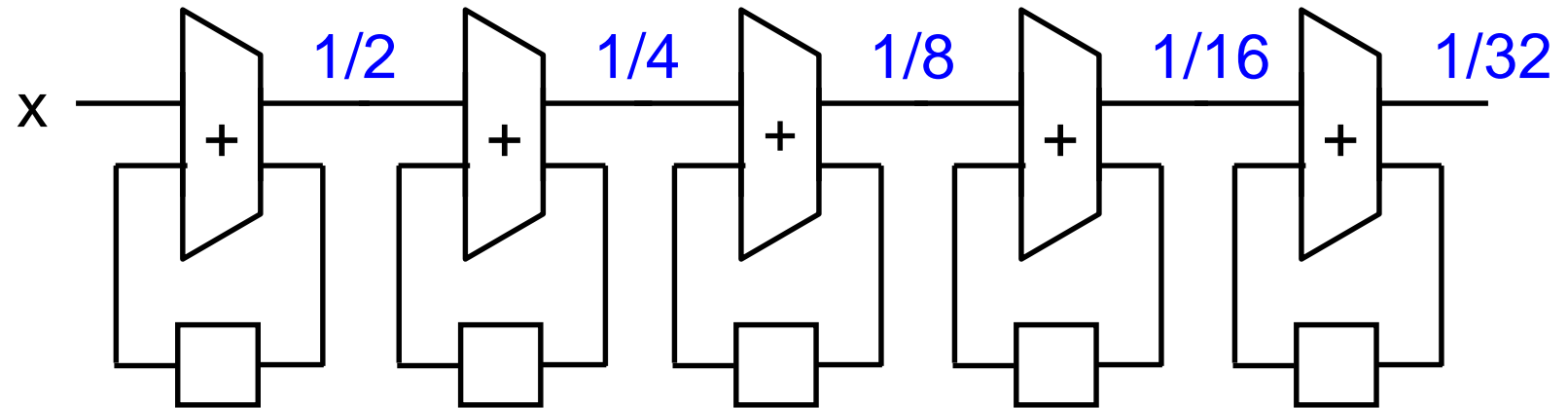


valeur binaire du compteur  
poids faibles d'abord

mais lent car la longueur du chemin critique  
dépend linéairement du nombre de bits

# *Le compteur ultra-rapide de Vuillemin*

La clef: regarder quand battent les retenues



8 registres!

# *Le compteur ultra-rapide de Vuillemin*

nombre de bits des tranches :

$$1 \rightarrow 2 = 2^1 \rightarrow 8 = 2^{1+2} \rightarrow 2048 = 2^{1+2+8} \rightarrow 2^{1+2+8+2048}$$

---

2059 bits, bien trop pour compter l'univers !

Vitesse quasi-constante en théorie

En pratique, attention au placement / routage  
la longueur des fils de clock-gating est limitante !  
 $\Rightarrow$  les remplacer par des arbres

# *L'instruction weak suspend (K. Schneider)*

- « suspend **p** when **S** » est naturel en logiciel ( $\hat{Z}$ ), car un code n'agit pas si on ne lui donne pas la main.
- Le circuit de suspend le fait en mettant **SUSP** à 1 et **RES** à 0
- Mais pourquoi ne pas laisser **p** agir, comme dans l'additionneur rapide ?
- En Esterel, il suffit de mettre **SUSP** et **RES** à 1 !

→ Instruction « weak suspend **p** when **S** »,  
et choix de l'utilisateur entre les deux suspensions,  
implémentables toutes deux en *clock gating*



# Sémantique par états

$$p \xrightarrow[I]{O, k} \bar{p}'$$

---

weak suspend  $p$  when  $S \xrightarrow[I]{O, k}$  weak suspend  $\bar{p}'$  when  $S$

(*go-weak-suspend*)

$$S \notin I \quad \hat{p} \xrightarrow[I]{O, k} \bar{p}'$$

---

weak suspend  $\hat{p}$  when  $S \xrightarrow[I]{O, k}$  weak suspend  $\bar{p}'$  when  $S$

(*res-weak-suspend-*)

$$S \in I \quad \hat{p} \xrightarrow[I]{\boxed{O}^k} \hat{p}' \quad k=1$$

---

weak suspend  $\hat{p}$  when  $S \xrightarrow[I]{\emptyset, 1}$  weak suspend  $\hat{p}$  when  $S$

(*res-weak-suspend+k=1*)

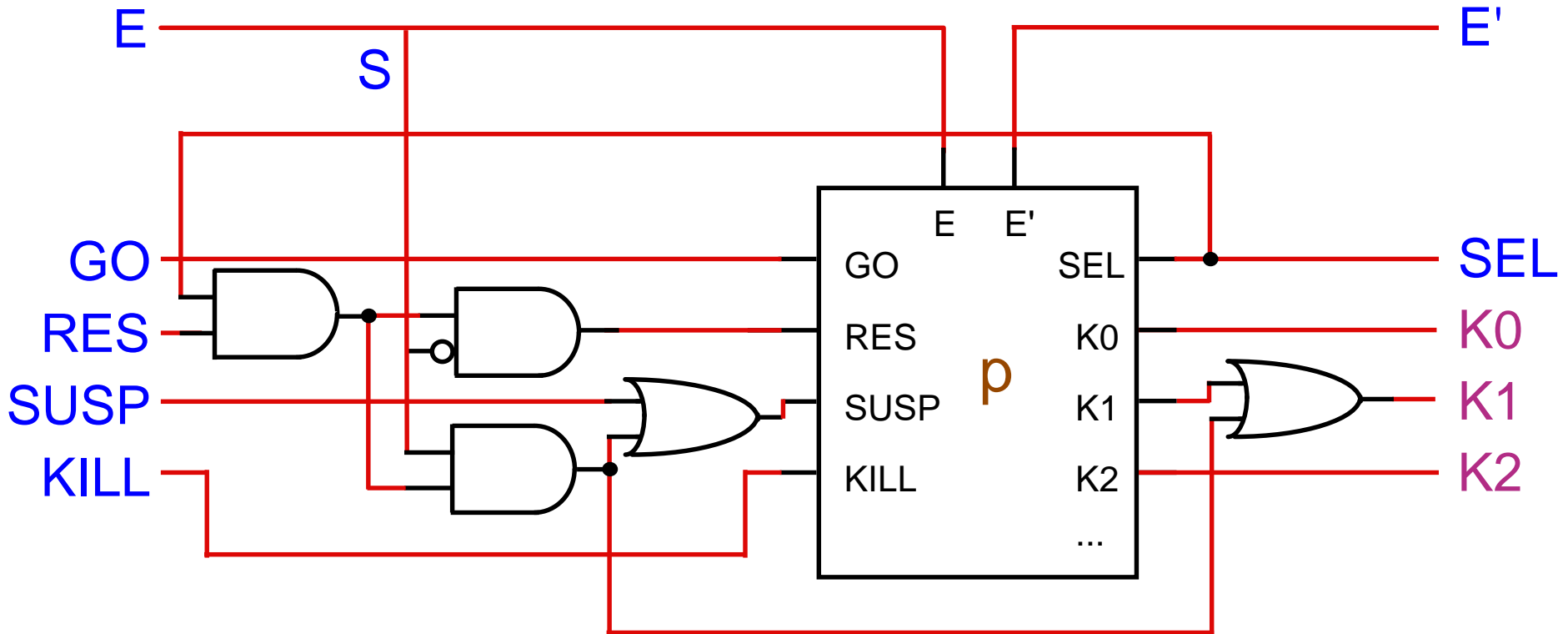
$$S \in I \quad \hat{p} \xrightarrow[I]{\boxed{O}^k} p \quad k \neq 1$$

---

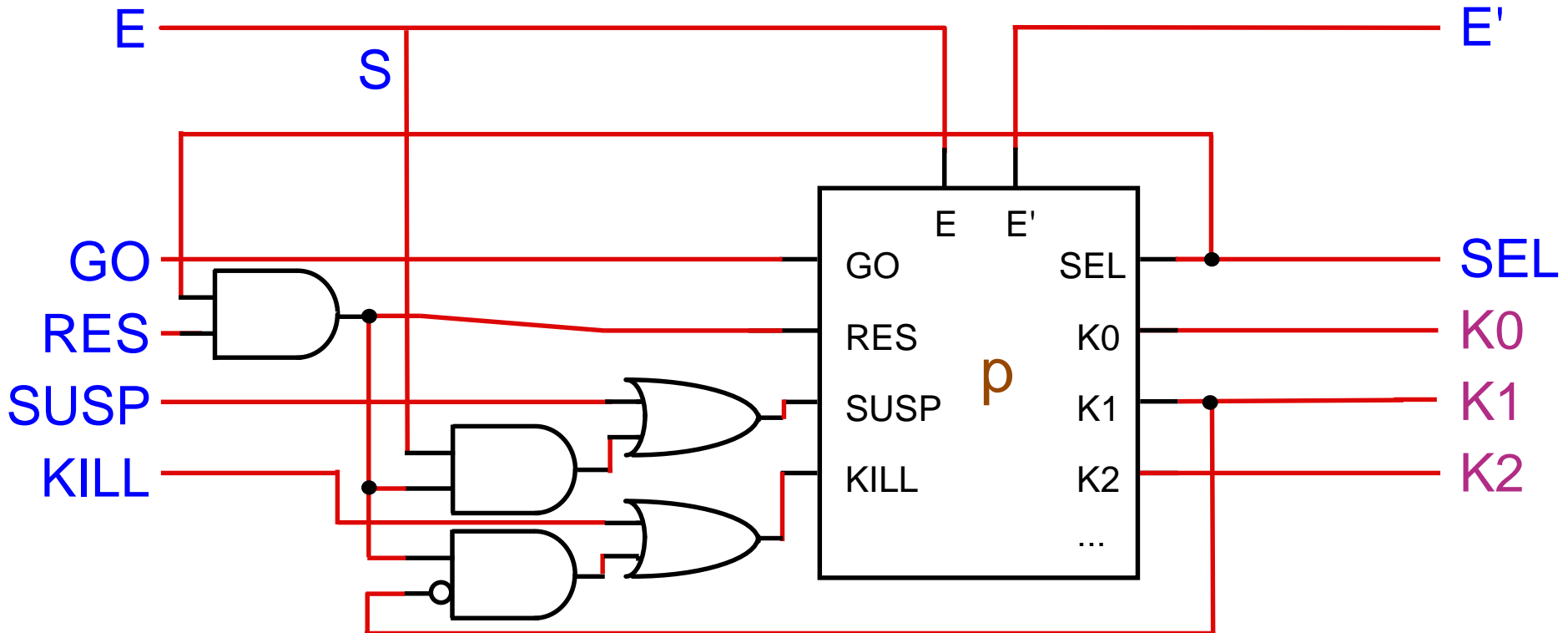
weak suspend  $\hat{p}$  when  $S \xrightarrow[I]{O, 1}$  weak suspend  $p$  when  $S$

(*res-weak-suspend+k≠1*)

# Circuit logique de suspend

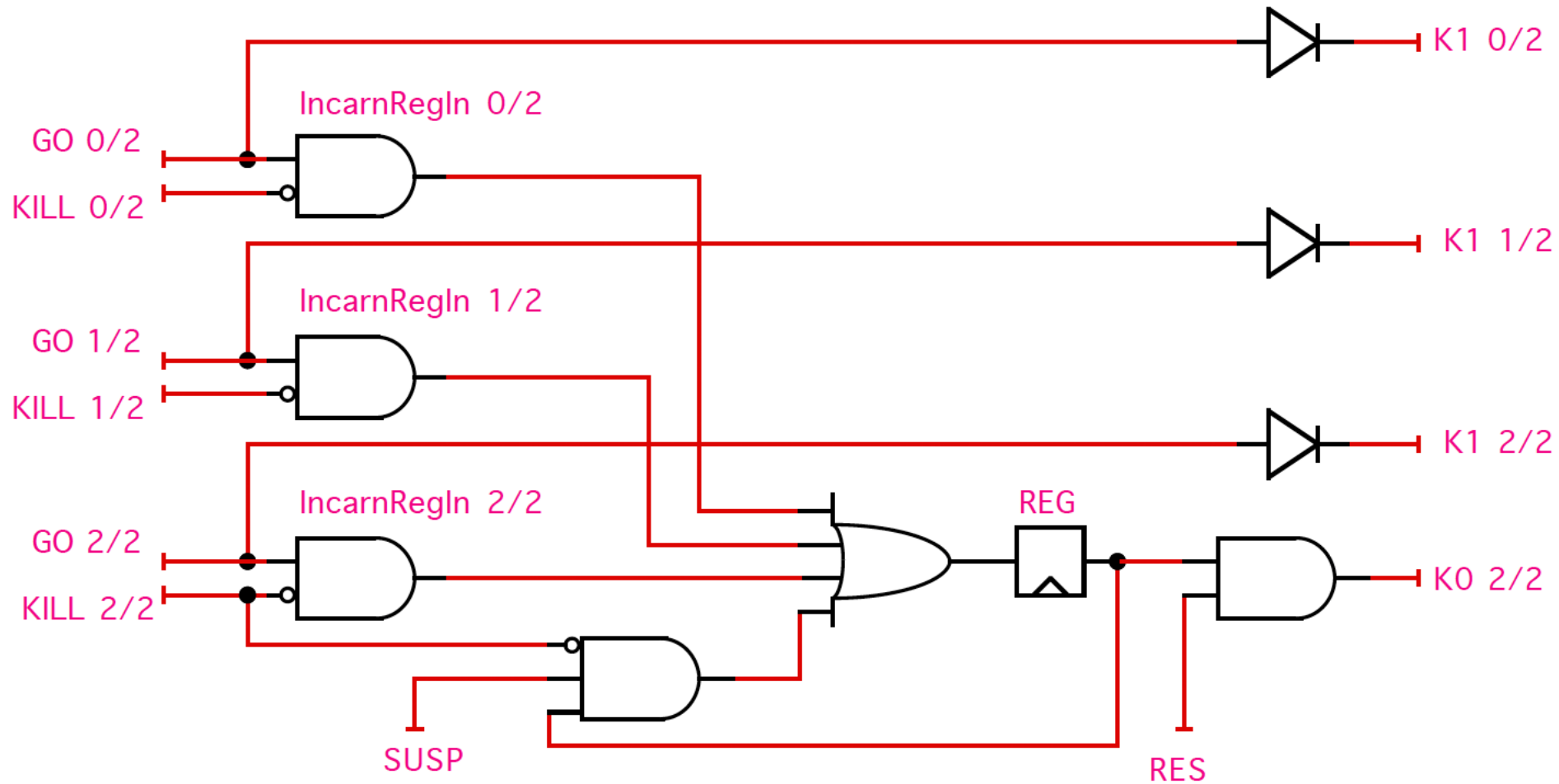


# Circuit logique de weak suspend



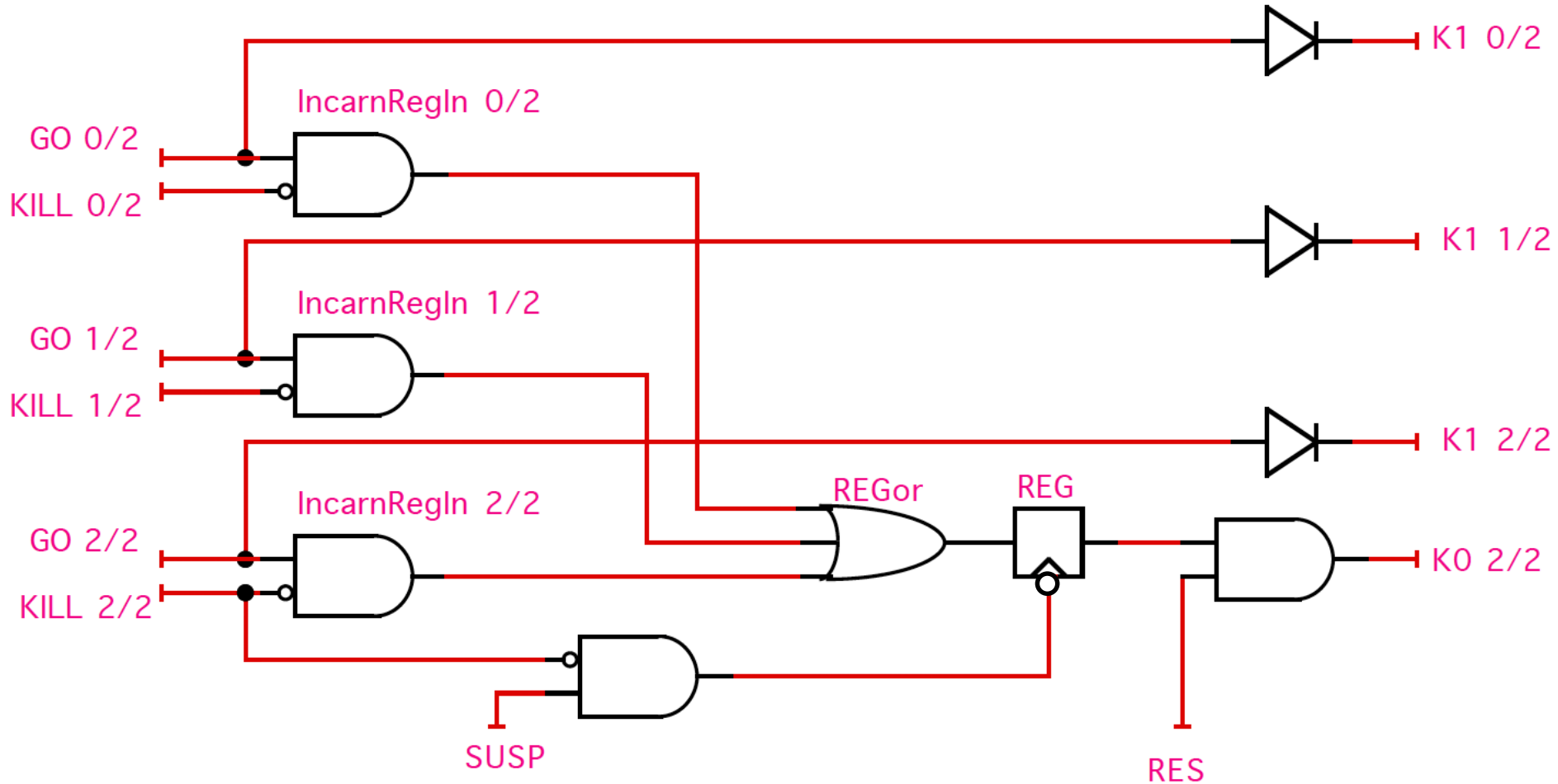
Si  $\hat{p}$  est suspendu, il est exécuté et donc met à 1 l'un des  $K_i$ . Si ce n'est pas  $K_1$ , il faut tuer  $\hat{p}$  par  $KILL$  (OK avec *clock gating*)

# Câblage de pause avec réincarnations



Marche pour suspend et weak suspend

# Le même avec clock gating



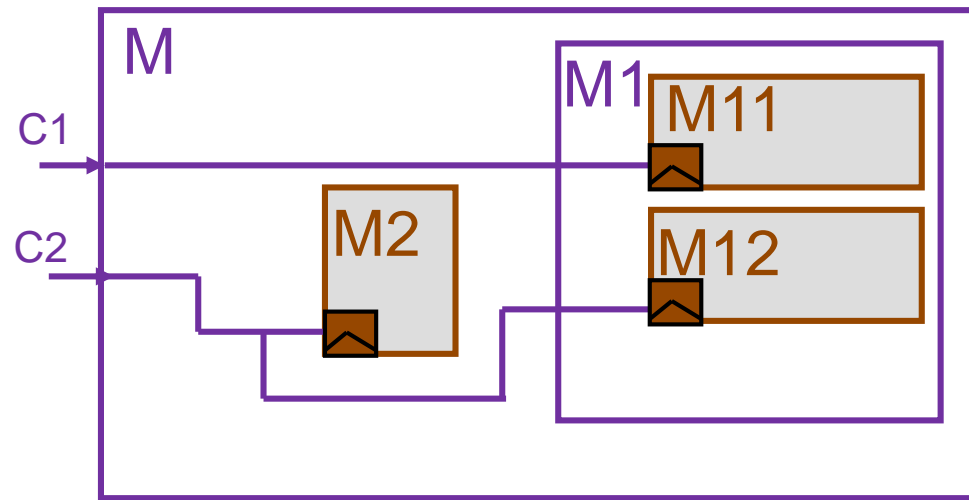
Marche pour suspend et weak suspend

# Note

- Les choix pour « weak suspend  $\hat{p}$  when  $S$  » on été longtemps hésitants pour  $k=0$  (terminaison) et  $k>1$  (exit)
  - Esterel v7\_60 : ignorer la terminaison de  $\hat{p}$  si  $k=0$ , et garder son état  $p$  ; mais autoriser la sortie par trappe
  - Draft IEEE 1668 (2008) : ignorer la terminaison et les sortie de trappes
- Sémantique proposée ici : accepte et réalise la terminaison et les sorties de trappes

Bien plus logique sémantiquement parlant selon moi, mais un *warning* est facile et ne fait pas de mal !

# Multi-horloges en Esterel v7



design hiérarchique  
multi-horloges

```
multiclock M:  
input C1,C2: clock;  
...  
run M1[C1/CC1,C2/CC2]  
||  
run M2[clock C2]  
end module
```

```
multiclock M1:  
input CC1,CC2:clock;  
...  
run M11[clock CC1]  
||  
run M12[clock CC2]  
end module
```

```
module M11:  
...  
end module
```

```
module M12:  
...  
end module
```

```
module M2:  
...  
end module
```

```

multiclock Multi :
input {C1, C2} : clock;
output C3 : clock;
input I, J;
output X, Y;
signal C4 : clock in
  sustain {
    C4 <= C1 if I,
    C3 <= mux(J, C2, C4)
  }
||
  run M1 [clock C1]
||
  run M2 [clock C4]
end multiclock

```



# Sémantique du multi-horloges

Traduction simple en Esterel standard :

1. Ajouter une horloge de base virtuelle
2. Transformer les horloges en signaux standards
3. Transformer run en weak suspend

```
run M [ clock C ]  
=>  
weak suspend  
  run M  
when immediate (not C)
```

# *weak suspend immédiat*

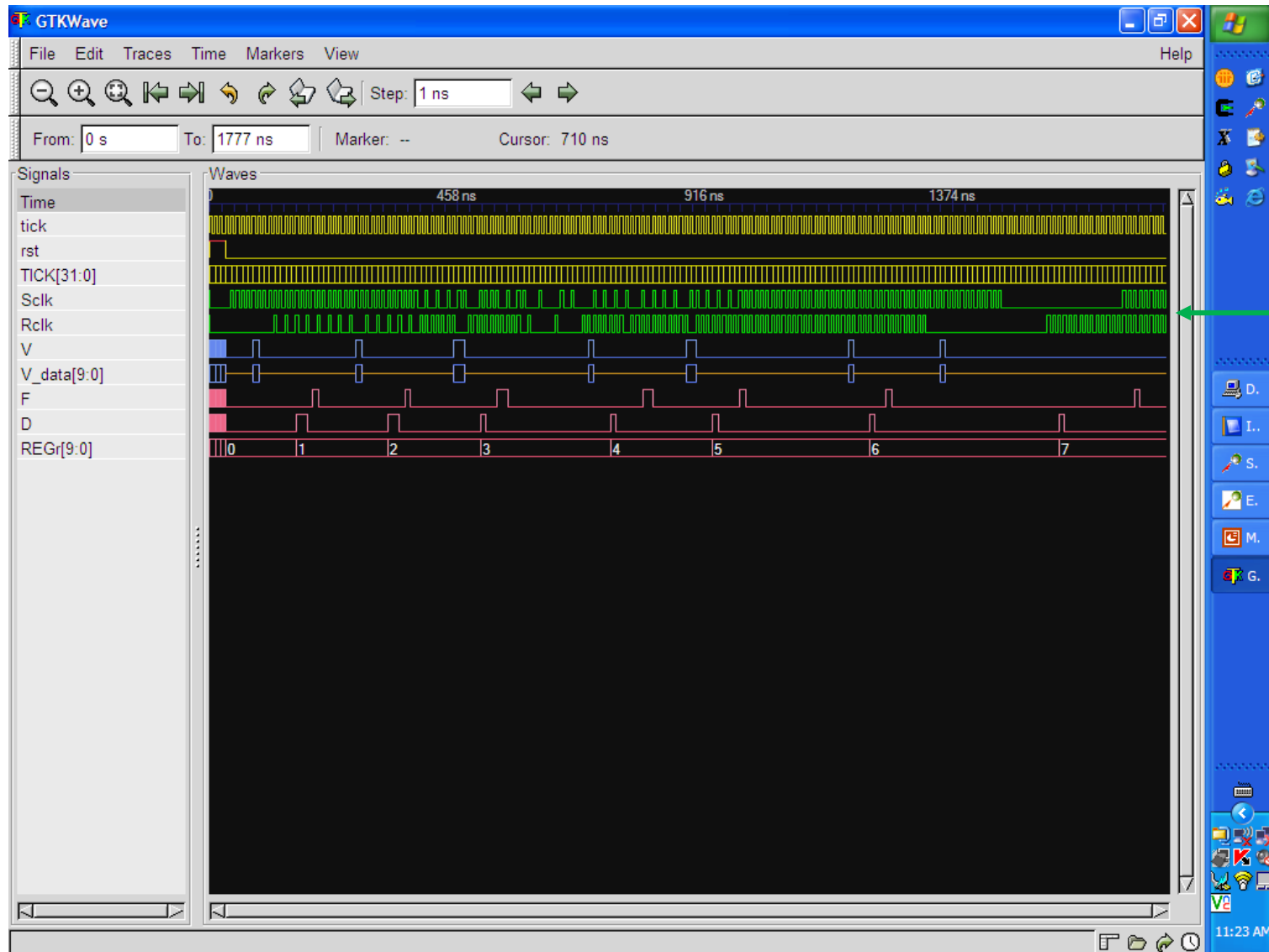
weak suspend  $p$  when immediate  $S$



Tordu, bien plus simple  
à axiomatiser directement !

```
→ trap Done in
  loop
    trap Immediate in
      {
        weak suspend
          p
        when S;
        ||
        if S then exit Immediate end
      };
    exit Done;
  end trap;
  pause
end loop
end trap
```

# Protocole de communication entre deux zones d'horloges

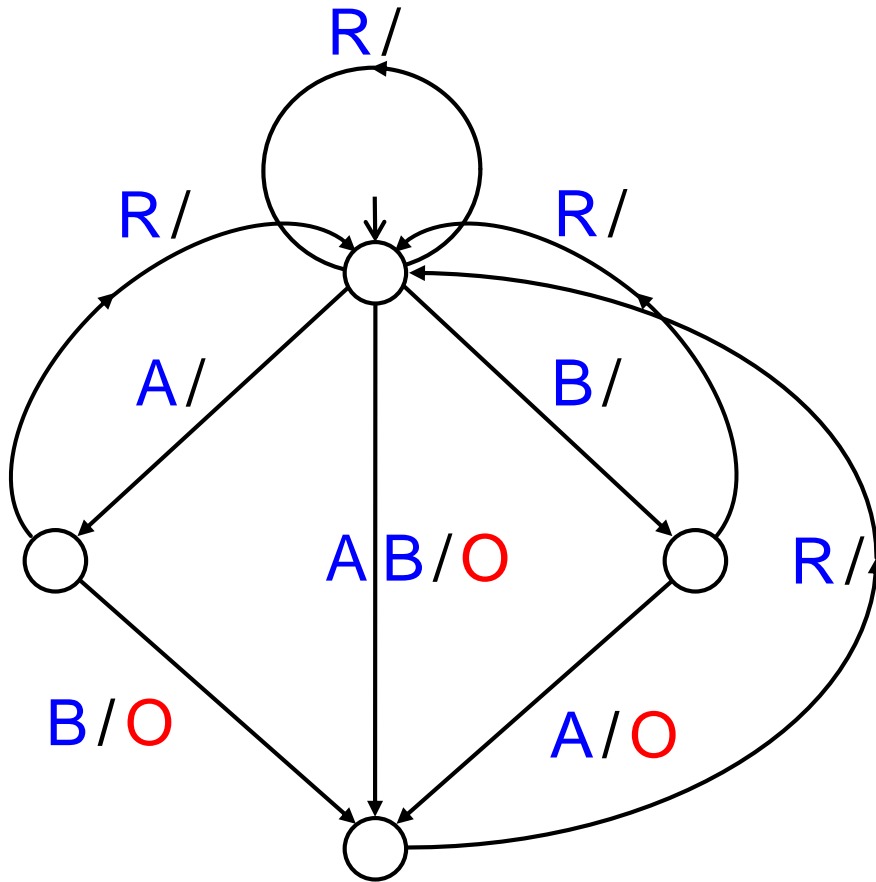


horloges

# *Agenda*

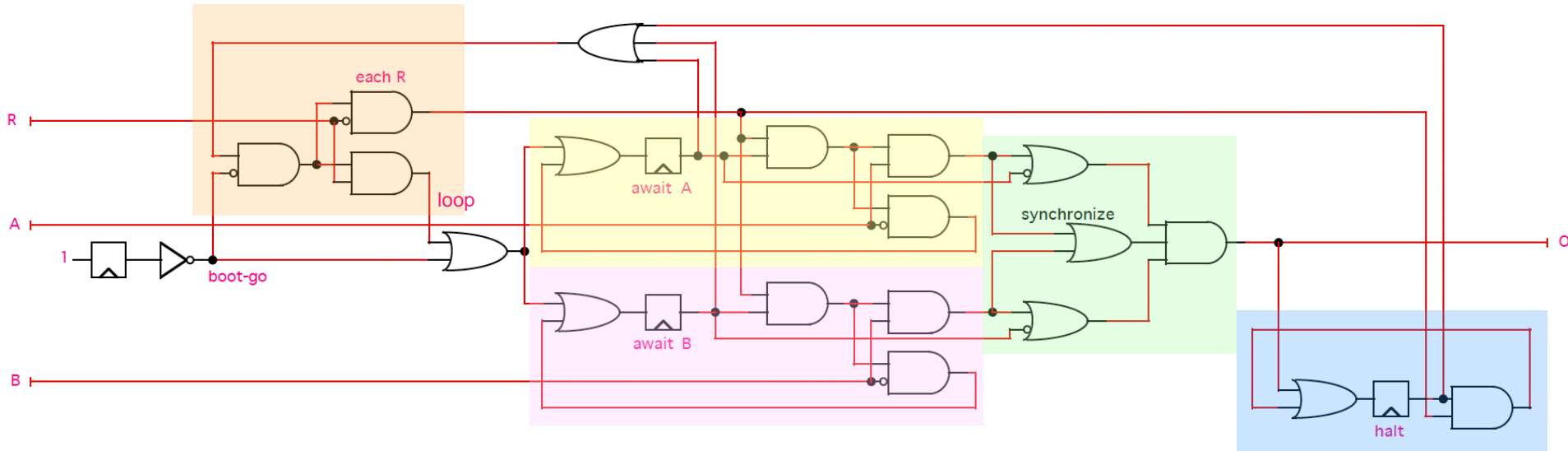
1. Weak suspend pour le clock gating
2. Réencodage et optimisation des circuits acycliques

# Rappel : ABRO



```
loop
  abort
  { await A || await B };
  emit O;
  halt
when R
end loop
```

# Le circuit compilé d'ABRO (simplifié)



```
loop
  abort
  { await A || await B };
  emit O;
  halt
when R
end loop
```

# Optimisation indispensable !

- Trop de portes combinatoires et de niveaux de logique !
  - mais tous les systèmes de CAO savent les réduire
- Trop de registres
  - les registres coûtent cher en place, en fils (horloge, *reset*), et en énergie; les CAO classiques ne savent pas les réduire

## Optimisation séquentielle :

réduire le nombre de registres, **mais pas trop**,  
en réduisant la logique.

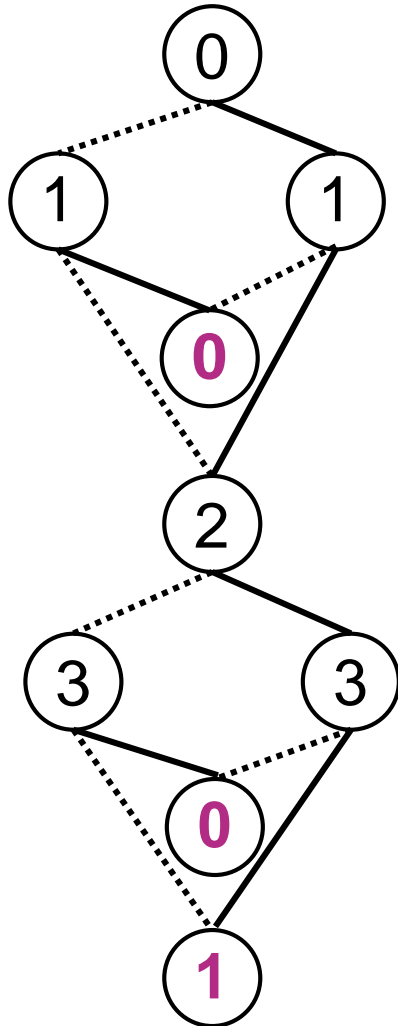
Ce qui compte est l'équilibre registres / logique  
 **$\log_2(n)$  registres pour  $n$  états est le plus souvent mauvais !**

Voir les cours des 21 mai 2013 et 9 mars 2016

# Les BDDs (Binary Decision Diagrams)

$$f(x_0, x_1, x_2, x_3) = (x_0 \Leftrightarrow x_1) \wedge (x_2 \Leftrightarrow x_3)$$

Ordre des variables fixé



Forme canonique  
pour un ordre de variables

Test d'égalité en temps 1  
(comparaison de pointeurs)

Satisfiabilité triviale  
comptage des solutions linéaire  
(les chemins vers 1)

Validité triviale  
formule réduite à 1 !

donc difficile à calculer dans le pire cas....



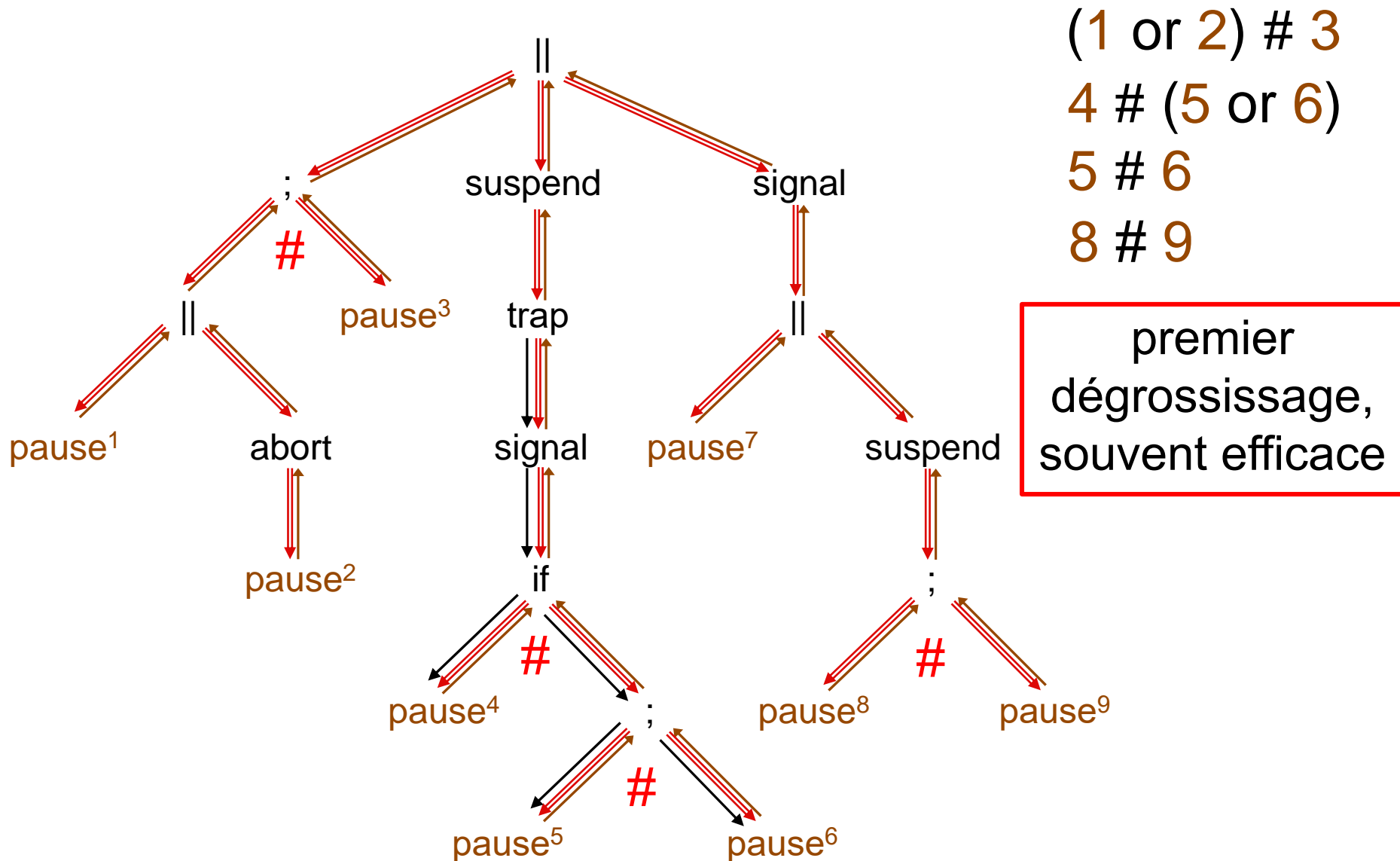
# Opération clef numéro 1

Si  $f : B^m \rightarrow B^n$  n'est appliquée qu'à un sous-ensemble  $D$  de  $B^m$ , calculer une autre fonction  $f_D : B^m \rightarrow B^n$  telle que  $\forall d \in D. f_D(d) = f(d)$  mais « plus simple » que  $f$ , en choisissant d'autres  $f_D(x)$  pour  $x \notin D$

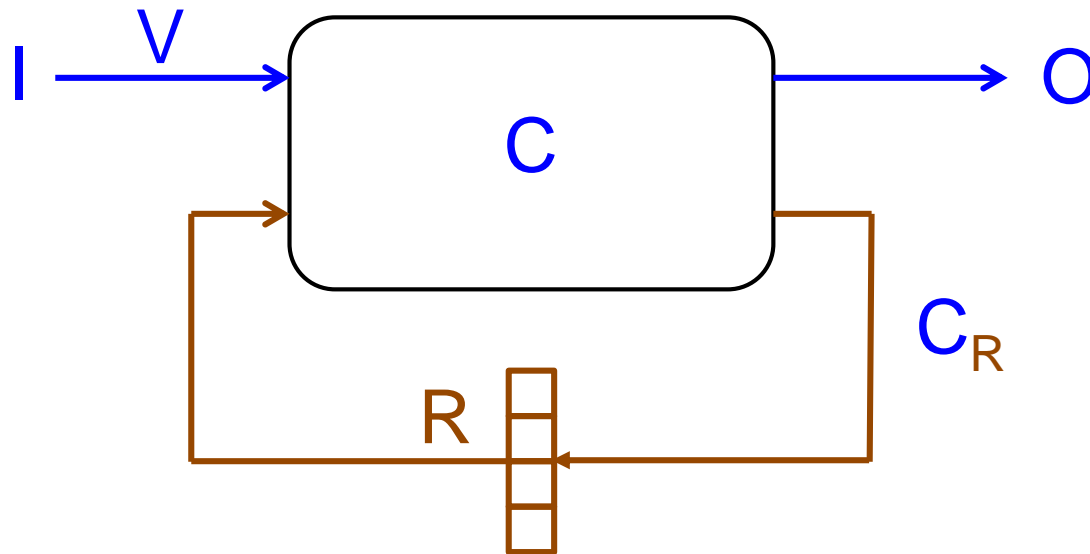
Exemple : pour  $\text{or}(x, y)$ , si on sait  $\text{not}(x \text{ and } y)$ , on peut choisir le ou exclusif  $\text{xor}(x, y)$ , de circuit plus simple

En pratique : calculs BDDs malins  
avec les opérateurs *restrict* et *constrain*  
(O. Coudert, J-C. Madre, voir cours du 9 mars 2016)

# 1. Invariant d'incompatibilité des pauses



## 2. Calcul des états accessibles (RSS) (voir cours du 21 mai 2013)



$$R_0 = 0$$

$$R_1 = R_0 U C_R(R_0, V)$$

...

$$R_{i+1} = R_i U C_R(R_i, V)$$

...

$$R_{n+1} = R_n U C_R(R_n, V) = R_n \Rightarrow \text{RSS} = R_n$$

A chaque étape,  
utiliser  $R_i$  et  $V$  comme  
simplifieurs pour  $C_R$



# Registres $\rightarrow$ logique (21 mai 2013)

Définition:  $r$  redondant s'il peut être remplacé par une fonction des autres registres, donc pas de la logique

$r_0$  redondant ssi  $\exists f(r_1, r_2, \dots, r_n)$  t.q.

$$\forall (r_0, r_1, r_2, \dots, r_n) \in \text{RSS}. r_0 = f(r_1, r_2, \dots, r_n)$$

$\Rightarrow r_0$  peut être remplacé par toute logique calculant  $f$

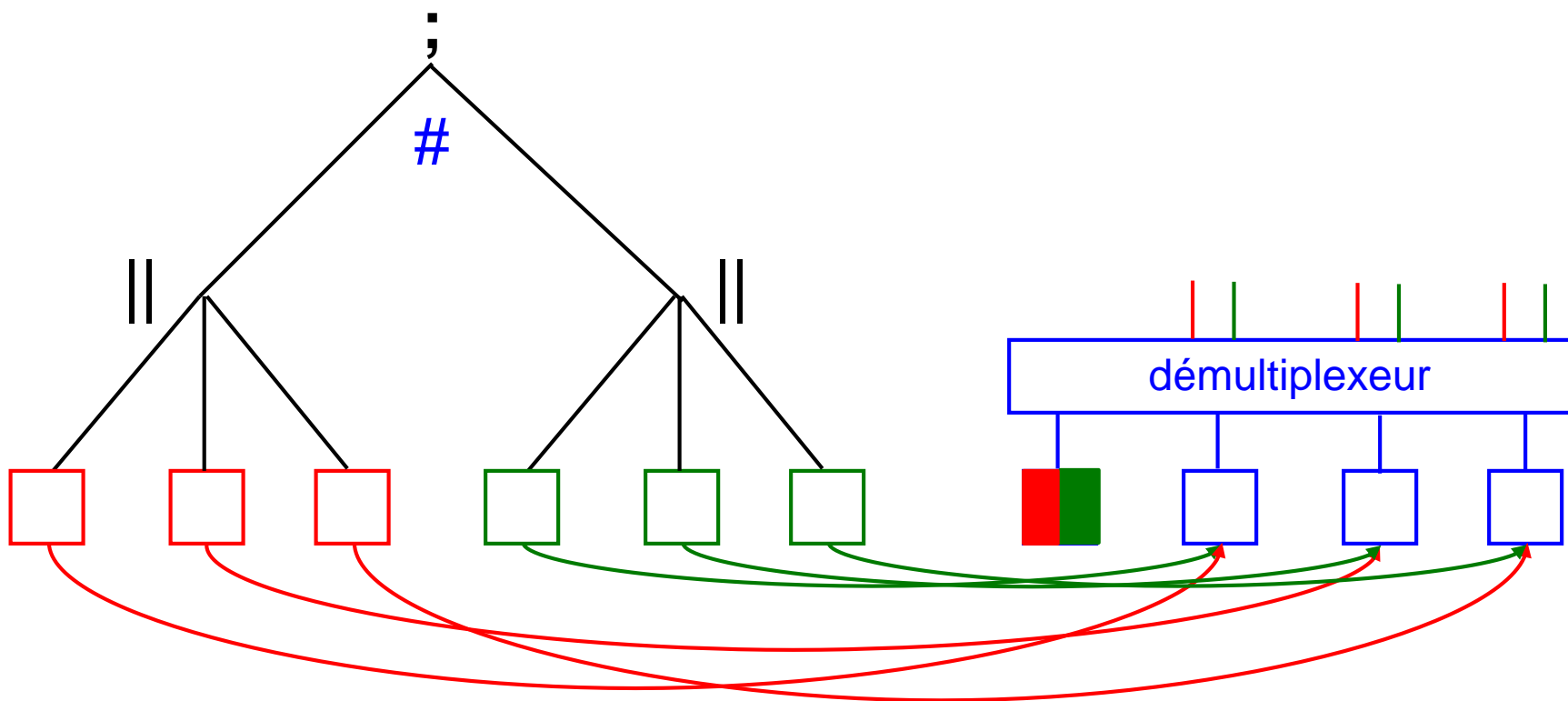
Théorème:  $r_0$  redondant ssi  $\text{RSS}[r_0 \leftarrow 1] \wedge \text{RSS}[r_0 \leftarrow 0] = 0$

Alors on peut remplacer  $r_0$  par  $f = \text{RSS}[r_0 \leftarrow 1]$

ou par  $f = \neg \text{RSS}[r_0 \leftarrow 0]$

Très utile si on peut calculer  $f$  avec un circuit simple !

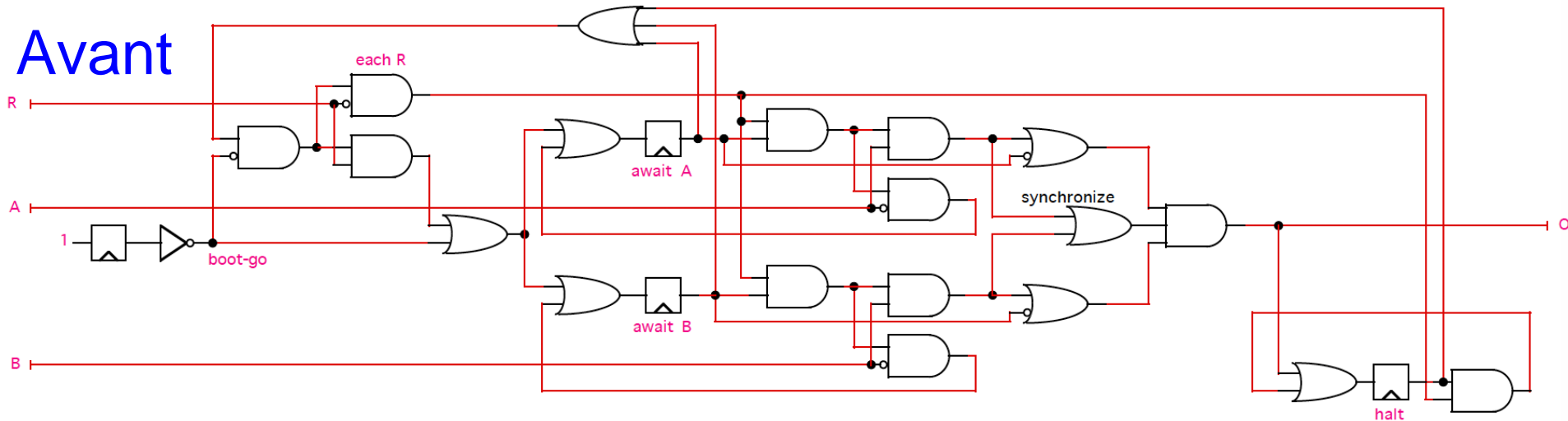
# Multiplexage de registres (21 mai 2013)



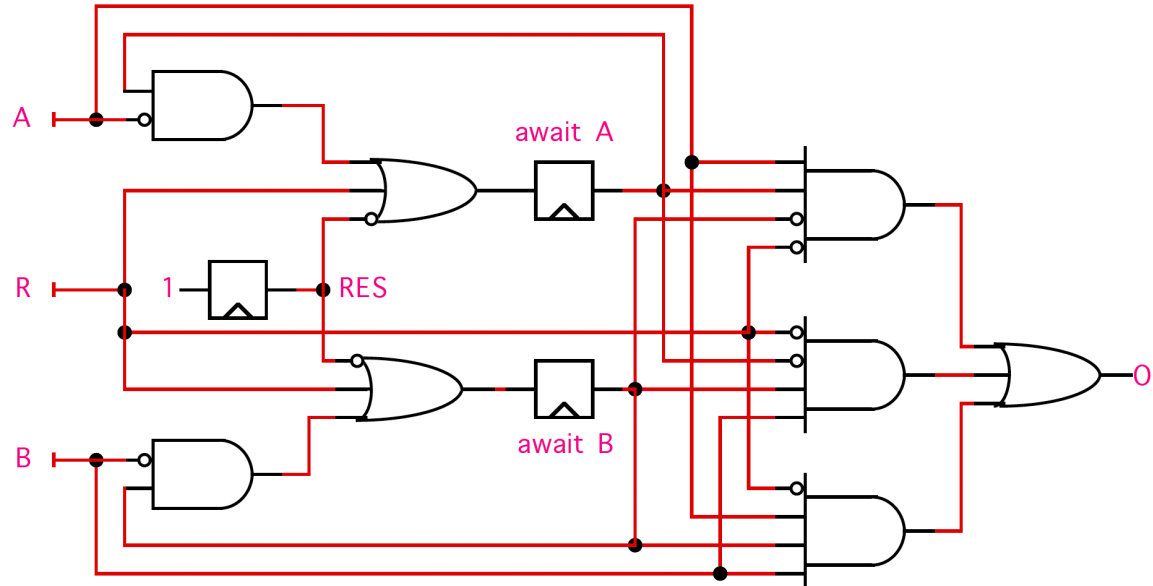
Pas toujours une bonne idée,  
à cause du coût en logique

# Application à ABRO

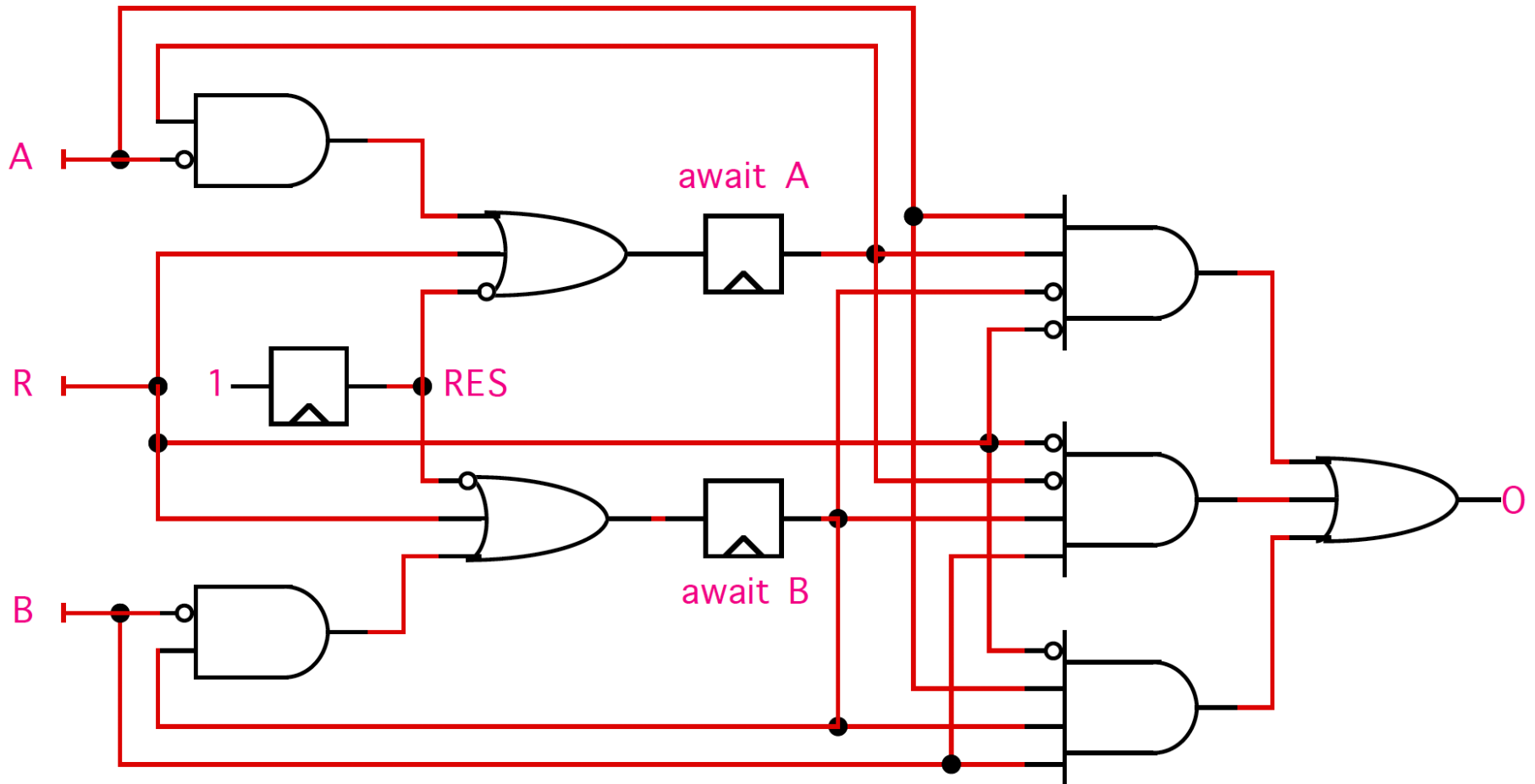
Avant



Après



# Circuit d'ABRO optimisé



Plus que 3 registres et 2 couches de logique !  
**halt = RES and not (awaitA or awaitB)**

# Application à la montre (21 mai 2013)

initial

WRISTWATCH nodes = 462 latches = 35 lits = 990 levels = 29

optimisation en vitesse (circuit matériel, FPGA)

WRISTWATCH nodes = 97 latches = 12 lits = 366 levels = 3

optimisation en surface (logiciel équationnel)

WRISTWATCH nodes = 98 latches = 11 lits = 195 levels = 15

BDDs → correction des optimisations (fsm-verify)

Grosses applications : optimisation module  
par module (optimisation compositionnelle)  
OK en Esterel v7, à faire en v5



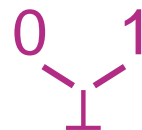
# *Agenda*

1. Weak suspend pour le clock gating
2. Réencodage et optimisation des circuits acycliques
3. Traitement des circuits cycliques

# Codage ternaire de la logique constructive

- Le circuit n'est qu'une machine électronique à calculer les preuves de la sémantique constructive, à la vitesse de la lumière (Curry-Howard électrique)
  - propagation des 1 (resp. 0) = calcul de *Must* (resp. *Cannot*)
- La logique constructive est équivalente à la logique Booléenne de Scott :  $\perp$  (inconnu), 1 (vrai), 0 (faux),  $\perp < 0$ ,  $\perp < 1$  et des fonctions croissantes

$$\frac{X = 0}{X \text{ and } Y = 0} \quad \frac{Y = 0}{X \text{ and } Y = 0} \quad \frac{X = 1 \quad Y = 1}{X \text{ and } Y = 1}$$


$$0 \text{ and } \perp = 0 \quad \perp \text{ and } 0 = 0 \quad 1 \text{ and } 1 = 1 \implies \perp \text{ and } \perp = \perp$$

Cf. cours du 09 décembre 2009

# Analyse de constructivité

- Pour chaque variable  $X$  du circuit, créer deux variables  $X^+$  et  $X^-$ 
  - $X^+ = 0, X^- = 0 \iff X$  est inconnu ( $\perp$ )
  - $X^+ = 1, X^- = 0 \iff X$  est constructivement démontré vrai
  - $X^+ = 0, X^- = 1 \iff X$  est constructivement démontré faux
  - $X^+ = 1, X^- = 1$  n'arrivera jamais
- Coder la logique constructive en logique booléenne par duplication des équations :
  - équation  $X = Y$  and  $Z \rightarrow X^+ = Y^+ \text{ and } Z^+$   
 $X^- = Y^- \text{ or } Z^-$
  - registres :  $X = \text{reg}(Y) \rightarrow X^+ = \text{reg}(Y^+), X^- = \text{reg}(Y^-)$

# *Analyse de constructivité*

- Calculer les BDDs des sorties et entrées de registres avec les nouvelles variables :
  - en utilisant les prédicats « not( $X^+$  and  $X^-$ ) » pour les entrées et les sorties de registres
  - en utilisant toutes les relations d'entrée
  - en calculant incrémentalement les états accessibles comme pour le cas acyclique, avec les résultats partiels comme prédicat d'entrée

# Analyse de constructivité

- Résultat

- Chaque variable  $X$  possède alors 2 BDDs sur les entrées  $I^+$ ,  $I^-$  et les sorties de registres  $RO^+$ ,  $RO^-$  :  
 $BDD^+(X)$  pour  $X^+$  et  $BDD^-(X)$  pour  $X^-$ , avec  $BDD^+(X)$  and  $BDD^-(X) = 0$
- Le circuit est **constructif**, i.e. **sémantiquement et électriquement correct** si et seulement si, pour chaque entrée valide + état accessible, chaque sortie combinatoire  $Y$  (sortie  $O$  ou entrée de registre  $RI$ ) est définie:  
 $BDD^+(Y)$  or  $BDD^-(Y) = 1$
- Mais leurs BDDs sont aussi des circuits, avec beaucoup de partage. On peut donc engendrer un **circuit acyclique équivalent**, puis **l'optimiser**



Calculs délicats, à conduire avec une grande finesse  
Le résultat acyclique peut aller de bon à explosif...

# Implémentation

- Optimisation des circuits acycliques implémentée dans le module `blifopt` du compilateur Esterel v5

```
$ esterel -O foo.str1
```

- Analyse de code cyclique et génération de code acyclique implémentées dans le module `sccausal` d'esterel v5 (T. Shiple, H. Touati A. Bouali)

```
$ esterel -causal [-O] foo.str1
```

Beaux exemples déjà discutés :

L'ILD (*Instruction Length Decoder*) du Pentium 26/03/2014  
Les IHM d'avions, cf. E. Ledinot (Dassault av.), 16/04/2013)

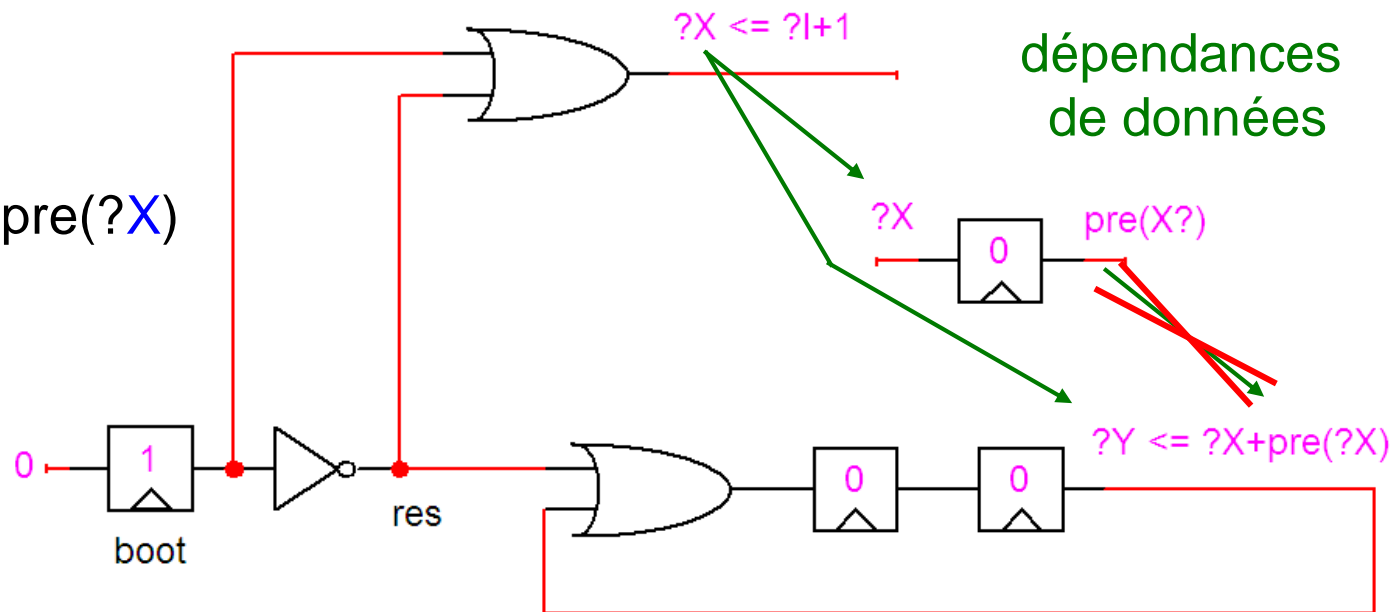
# *Agenda*

1. Weak suspend pour le clock gating
2. Réencodage et optimisation des circuits acycliques
3. Traitement des circuits cycliques
4. Ajout du traitement des données aux circuits

# Attacher les calculs de données aux fils

```
input I : integer;  
signal X, Y : integer in  
  sustain ?X <= ?I+1  
||  
loop  
  pause;  
  pause;  
  emit ?Y <= ?X+pre(?X)  
end loop  
end signal
```

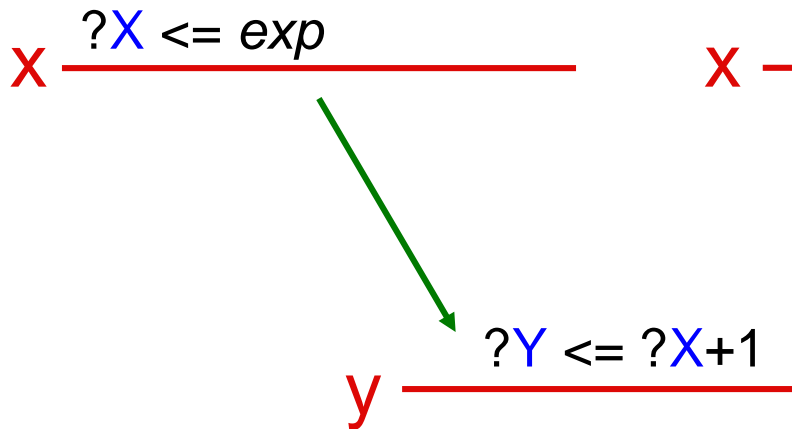
Effets de bords calculés si et quand leur fil passe à 1 (vision constructive)



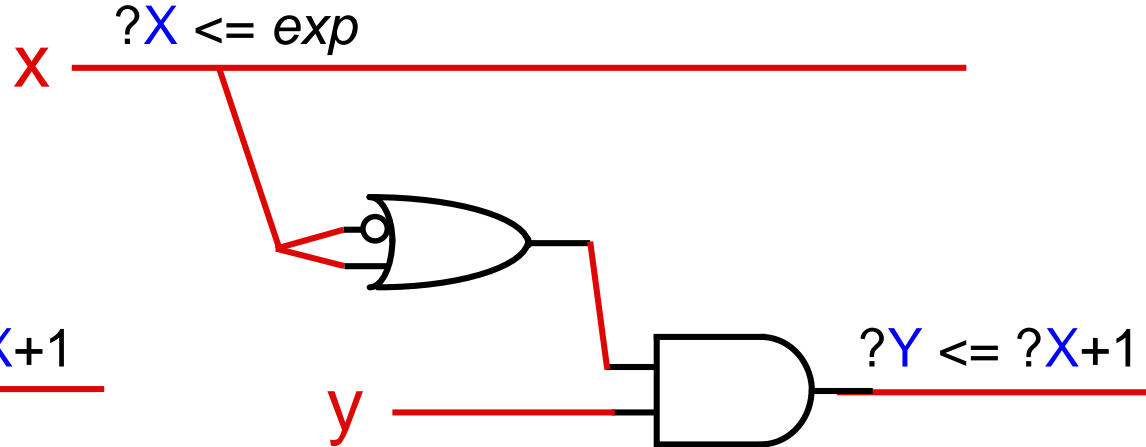
Permet de soumettre les cycles mixtes circuits + données à l'analyse de causalité par BDDs ([sccausal](#)) ...



# ... par codage constructif des dépendances



dépendance  
additionnelle



dépendance constructive  
→ esterele -causal

1.  $y = 0$  : le 0 traverse librement la porte « et »
  2.  $y = 1$ , il faut attendre que  $x$  vaille 0 ou 1,  
Attente **constructive** en logique, **physique** en électronique
- Attention** : détruit par toute optimisation classique !

# *Agenda*

1. Weak suspend pour le clock gating
2. Réencodage et optimisation des circuits acycliques
3. Traitement des circuits cycliques
4. Ajout du traitement des données aux circuits
5. Compilation ou interprétation logicielles

# *Interprétation constructive directe*

- Simuler la propagation constructive des valeurs dans le circuit acyclique ou cyclique en suivant les dépendances (facile, une table et une file d'attente)
- Calculer les effets de bord au passage pour les fils valant 1
- Si le calcul évalue tous les fils, **programme constructif** pour les entrées données
- Sinon, **programme non constructif**, expliquer pourquoi (recherche de cycles minimaux dans le graphe restant, pas simple...)

Compilateurs estereel v5/v7 : `estereel -I foo.str1`

Temps **linéaire** en la taille du programme

Implémente **exactement** la sémantique constructive

# Compilation linéaire dans le cas acyclique

- Si aucun cycle dans le graphe portes + dépendances, **trier topologiquement** les portes et exécuter les équations dans l'ordre avec leurs effets de bord
- Avant ou après optimisation booléenne du circuit
- Utilisable pour les programmes cycliques après l'analyse BDD de l'option `-causal`

Compilateurs estereel v5/v7 :  
option par défaut

```
estereel [-causal] [-O] foo.str1
```

# Traduction en C du circuit : ABRO non optimisé

```
int ABRO (void) {
static __SSC_BIT_TYPE E[8];
E[0] = __ABRO_R[1]||__ABRO_R[2];
E[1] = (__ABRO_R[3]||E[0])&&!(__ABRO_R[0]);
E[2] = E[1]&&!((
__ABRO_A3));
E[3] = __ABRO_R[1]&&E[2];
E[4] = E[3]&&(
__ABRO_A1);
E[5] = __ABRO_R[2]&&E[2];
E[6] = E[5]&&(
ABRO A2);
E[7] = (E[0]&&E[2]&&!(__ABRO_R[1]))||E[4];
E[0] = (E[0]&&E[2]&&!(__ABRO_R[2]))||E[6];
E[6] = (E[4]||E[6])&&E[7]&&E[0];
if (E[6]) {
__ABRO_A4;
}
```

K0

```
E[4] = !(_true);
E[1] = E[1]&&(
__ABRO_A3);
E[1] = E[1]||__ABRO_R[0];
E[3] = E[3]&&!((
__ABRO_A1));
__ABRO_R[1] = E[1]||(__ABRO_R[1]&&E[3]);
E[5] = E[5]&&!((
__ABRO_A2));
__ABRO_R[2] = E[1]||(__ABRO_R[2]&&E[5]);
__ABRO_R[3] = E[6]||(__ABRO_R[3]&&E[2]);
E[0] = ((__ABRO_R[1]||__ABRO_R[2])
&&(E[7]||__ABRO_R[1])
&&(E[0]||__ABRO_R[2]))||__ABRO_R[3];
__ABRO_R[0] = !(_true);
__ABRO__reset_input();
return E[0];
}
```

K1

synchroniseur du parallèle (non schizophrène)

Plus efficace, linéaire ou mieux après optimisation

# *Debug symbolique*

- Ajouter au code généré des annotations conservées pendant toute la compilation: pointeurs sur le source, etc.
- Lier avec la librairie de simulation d'esterel v5/v7

```
$ esterel [-I] -simul foo.str1  
$ cc -c foo foo.c $ESTEREL/lib/libcoresim.  
$ ./foo
```

Voir la démo de la montre en fin de cours

# Compilation en automate fini

- Puisque les états d'un programme se codent par les pauses allumés, son contrôle est **d'état fini**
- **Explorer explicitement ces états**, en engendrant seulement les tests et actions de calcul indispensables pour chaque état
- Introduit pour Esterel v2, bien amélioré ensuite
- Génération de byte-code très rapide, **disparition des signaux et communications** (devenues infiniment rapides !)
- Utile aussi pour les méthodes de preuve explicites, cf. cours du 06/04/2016

Mais **risque d'explosion exponentielle**  
(dépend fortement des applications)

# Exemples de compilation en automates

```
estere1 -A abro.str1
```

```
1:      2:      3:      4:      5:
<2>    5 (<2>) ()   5 (<2>) ()   5 (<2>) ()   5 (<2>) ()
      3 (1 (6 <3>) ()  <3>      1 (6 <3>) ()  3 (6 <3>) ()
        <4>) ()      %awaited: 2% <4>      %awaited: 0 2% <5>
      1 (<5>) ()      %awaited: 1 2%
      <2>
      %awaited: 0 1 2 %
```

0,1,2 : signaux A,B,R

1,2,3,4,5 : états

1,3,5 : tests de A,B,R

6 : émission de O

<2> : le prochain état sera 2

% awaited: 0 2% : au prochain état,  
seuls A et R provoqueront une transition



# La montre en automate

- Circuit acyclique : `estereel wristwatch.str1` → 42 états

```
cc -O3 wristwatch.c
```

TEXT	DATA	others	dec	(non optimisé par BDDs)
10158	483	448	11089	

- Automate : `estereel -A wristwatch.c`

```
cc -O3 wristwatch.c
```

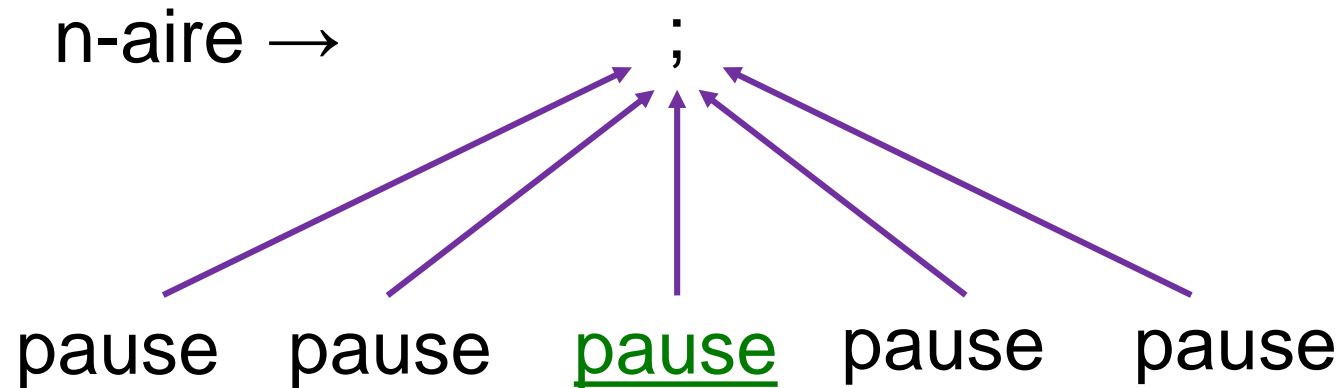
TEXT	DATA	others	dec
7993	4033	3040	15066

Pas d'explosion, 40% plus gros, mais **beaucoup plus rapide** (code des états strictement minimal)

# *Agenda*

1. Weak suspend pour le clock gating
2. Réencodage d'états et optimisation du circuit
3. Ajout du traitement des données au circuit
4. Compilation ou interprétation logicielles
5. Compilation logicielle rapide

# Remontée de la sélection dans les circuits

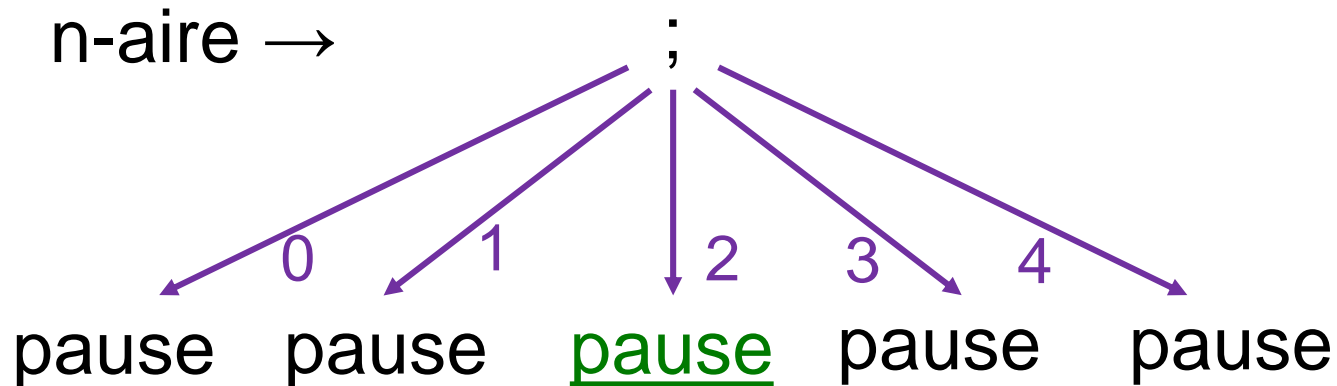


Circuit : électricité parallèle  $\rightarrow 1 \leq \text{temps} \leq \log_2(n)$

Logiciel : code séquentiel  $\rightarrow \text{temps } n$



# S. Edwards : numéroté les fils exclusifs



Profiter du fait que l'adressage en mémoire est constant  
⇒ le décodage de l'état devient **indépendant de la taille**

Pour le code combinatoire, utiliser au maximum if-then-else  
pour ne pas exécuter les parties de codes non activées  
(restreint aux programmes acycliques)

Technique reprise par [D. Poto](#) (Esterel v5) et [M. Perreaut](#) (v7)

Voir les détails dans le cours du 21 mai 2013, dans le livre [Compiling Esterel](#),  
et dans les transparents d'Esterel Technologies sur la page web de ce cours

# Conclusion

- weak suspend étend le champ expressif d'Esterel, en particulier pour les circuits mono- et multi-horloges
- Les algorithmes de compilation et d'optimisation sont variés, mais **tous conformes à la sémantique**. Certains traitent les programmes cycliques, d'autre pas, mais **les résultats sont toujours consistants**.
- Les **BDDs** cs@listes.societe-informatique-de-france.fr jouent un rôle essentiel pour l'optimisation et la vérification (cours du 28 mars), donc la pratique industrielle

Merci à J. Vuillemin, H. Touati, J-C. Madre, O. Coudert, A. Bouali, X. Fornari, E. Sentovich, H. Toma, S. Edwards, D. Potop-Butucaru, M. Perreaut, L. Arditi, et aux autres acteurs d'Esterel v5 et v7