
# DESSIN
# Développement de SystèmeS Intégrés Numériques

http://www.comelec.enst.fr/dessin/

---

Deuxième partie

Lirida Naviner
Jean-Luc Danger
Yves Mathieu
Jean Provost
Sylvain Guilley
Alexis Polti

**Avril 2004**

Département Communications et Electronique
Ecole Nationale Supérieure des Télécommunications

# Le langage VHDL, de la spécification au modèle

## Very high speed integrated circuits Hardware Description Language

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- Syntaxe
  - *VHDL séquentiel*
  - *VHDL concurrent*
- Les paquetages normalisés
- La synthèse
- Conseils

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- Syntaxe
  - *VHDL séquentiel*
  - *VHDL concurrent*
- Les paquetages normalisés
- La synthèse
- Conseils

# Origines du langage VHDL

- **VHDL (IEEE 1076-1987) est né en 1987 des efforts conjoints de :**
  - ***IEEE ->***
    ***Computer Society ->***
    ***Design Automation Technical Committee ->***
    ***Design Automation Standards Subcommittee ->***
    ***VHDL Analysis and Standardization Group***
  - ***CAD Language Systems Inc.***
- **VHDL (IEEE 1076-2002) est la révision actuelle**
- **D'autres standards « périphériques »**
  - ***VITAL***
  - ***Synthèse***
  - ***etc.***



Le langage standard IEEE VHDL (VHSIC Hardware Description Langage) a été développé par le Groupe d'Analyse et de Standardisation VHDL (VASG, pour "VHDL Analysis and Standardization Group"). Larry Saunders est le coordinateur de VASG. La société CLSI (CAD Langage Systems Inc.), représentée par le Docteur Moe Shahdad et M. Erich Marschner a préparé une série d'analyses et de recommandations dont a été tirée en Février 1986 la version 7.2 de VHDL, point de départ du futur standard. La collaboration de CLSI au projet était financée par un contrat passé avec l'Air Force Wright Aeronautical Laboratories, représentée par le Docteur John Hines. Le standard définitif a été adopté vers le milieu de l'année 1987.

Pour en savoir plus : http://vhdl.org

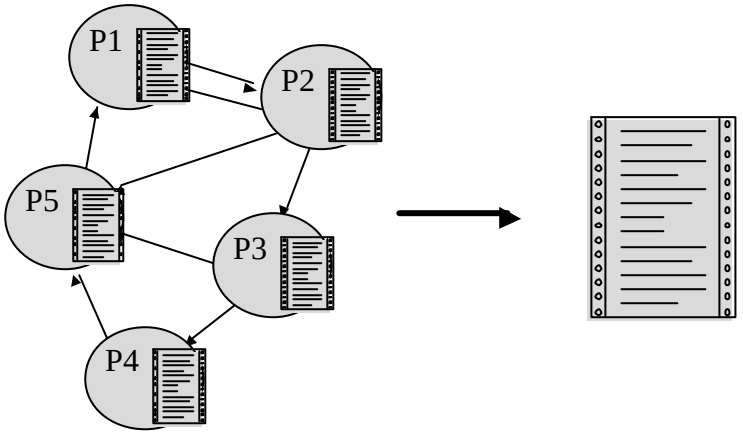
# Avertissement

- **Ce cours présente le langage VHDL dans sa version 1993.**
  - ***Des modifications de la norme, survenues dans les versions ultérieures, peuvent entrer en conflit avec certains points abordés ici.***
  - ***Pour plus de précisions se reporter aux différents manuels de référence du langage (éditions IEEE).***
- **Ce choix a été fait pour plusieurs raisons dont la plus importante est que certains outils n'implémentent pas l'intégralité des versions plus récentes.**

# Plan

- Introduction
- **Principes de la simulation évènementielle**
- **Organisation informatique des projets**
- **Les unités de compilation**
- **Syntaxe**
  - ***VHDL séquentiel***
  - ***VHDL concurrent***
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# Systèmes parallèles sur machines séquentielles





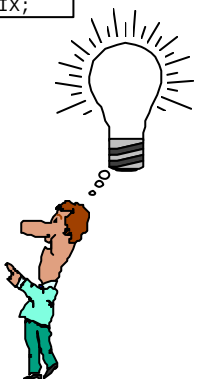

L'esprit humain est familier de la programmation séquentielle. Il nous est en effet naturel de décrire un traitement sous la forme d'un enchaînement d'opérations à effectuer dans un ordre pré-établi. Le matériel est, par essence, parallèle. Dans un circuit intégré, tous les composants sont actifs simultanément. De ces deux considérations on déduit que la forme la plus probable des langages fonctionnels de description de matériel comportera des aspects séquentiels et des aspects parallèles. En d'autres termes, un modèle de composant électronique sera composé de l'assemblage de plusieurs programmes séquentiels, s'exécutant en parallèle. Le concepteur partitionnera son problème en modules assez simples pour être décrits sous la forme de programmes séquentiels classiques, puis il assemblera ces modules que le simulateur exécutera en parallèle.

Le problème, car problème il y a, nait de ce que le simulateur est, en général, installé sur une machine séquentielle, capable d'exécuter une instruction à la fois, donc un programme à la fois. Il ne pourra donc pas parallèliser réellement les différents programmes. Il faut donc trouver un moyen d'émuler le parallélisme sur une machine séquentielle. Comment faire ?

# Systèmes parallèles sur machines séquentielles

- **On veut introduire le parallélisme**
- **On invente un nouveau genre de variables dédié à la communication entre les programmes séquentiels (processus) : le signal.**
- **L'exécution d'une instruction ne modifie pas instantanément la valeur d'un signal.**
- **Les signaux sont modifiés lorsque tous les processus ont été exécutés (après 1 Δ).**

```
SIG1 <= 1;
SIG2 <= ROUGE;
SIG3 <= DIX;
```





L'émulation du parallélisme repose sur une idée très simple et très efficace. Pour que chaque programme séquentiel, qu'on appellera désormais processus, s'exécute "comme si" le parallélisme était réel, il faut que son environnement (ses variables d'entrée) ne change pas pendant l'exécution des autres processus. Ainsi, l'ordre d'exécution des processus n'a plus d'importance et tout se passe comme s'ils s'exécutaient en parallèle. Pour parvenir à ce résultat, il faut que les variables partagées entre processus conservent leur valeur jusqu'à ce que tous les processus aient fini leur exécution. En quelque sorte, on simule pas à pas et, à chaque pas :

- le simulateur exécute tous les processus dans un ordre quelconque

- toute instruction modifiant la valeur d'une variable partagée verra son exécution différée ; on se contentera d'enregistrer la nouvelle valeur dans une zone temporaire, la variable partagée conservant son ancienne valeur

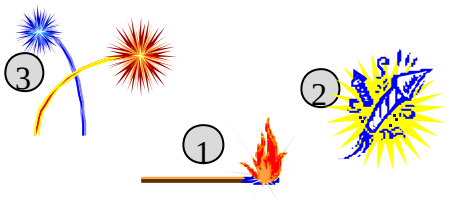
- lorsque tous les processus ont été exécutés, on modifie toutes les variables partagées en allant chercher leur nouvelle valeur dans la zone temporaire

- et on recommence un nouveau pas ...

Les variables partagées ou globales sont donc traitées d'une façon particulière. On appelle ces variables des signaux, pour les distinguer des variables classiques.

# Le temps symbolique

- **Permet d'exprimer des relations de dépendance entre événements, de les ordonner**
- **Permet de distinguer la cause de l'effet**
- **Est le seul que les synthétiseurs reconnaissent**




```
wait until A = 25;
B <= 3;  -- (after Δ)
```

```
wait until B = 3;
C <= 12; -- (after Δ)
```



On appelle delta (Δ) la durée séparant l'exécution d'une instruction d'affectation sans délai sur un signal de la modification réelle de la valeur de ce signal. Un delta ne possède pas de durée physique. Le delta est l'unité de base du temps symbolique, comme la seconde est une unité du temps physique.

Le temps symbolique permet d'ordonner les événements apparemment "simultanés", c'est à dire qu'il donne la possibilité de déterminer quel événement a provoqué l'autre dans le cas de deux événements ayant la même date physique.

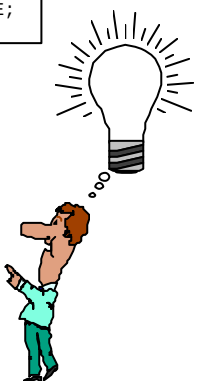
Un événement possède donc une date complète, composée d'une date physique (1 heure, 17 minutes, 43 secondes ...) et une date symbolique (... 4Δ). Lorsqu'une affectation de signal ne comporte pas de précision sur la date physique à laquelle l'événement doit avoir lieu, il est sous-entendu que la date sera la date à laquelle l'affectation a été exécutée augmentée d'un Δ. Lorsque la date physique est précisée, la date de l'événement est exactement cette date physique.

Notons que les synthétiseurs logiques n'utilisent pas le temps physique. Seul l'ordre des opérations que subissent les données sont pertinentes pour la synthèse. Le temps symbolique est donc le seul que les synthétiseurs comprennent.

# Systèmes parallèles sur machines séquentielles

- **On veut conserver l'aspect séquentiel :**
- **Les variables existent toujours, à l'intérieur des processus.**
- **L'affectation des variables est instantanée.**
- **Les processus s'exécutent de façon classique.**

```
VAR1 := 1;
VAR2 := ROUGE;
VAR3 := DIX;
```





L'aspect séquentiel est conservé à l'intérieur des processus. Ils obéissent aux règles très classiques de la programmation séquentielle. On trouve les même instructions et les même structures de contrôle qu'en langage C, en Pascal ou en Ada. Les variables locales aux processus sont traitées comme dans tout langage de programmation classique. A la différence des signaux, elles sont modifiées dès l'exécution d'une instruction d'affectation qui les concerne. Ces variables locales ne sont, bien sûr, pas visibles de l'extérieur du processus où elles sont déclarées. Un autre processus ne peut ni les lire, ni les modifier. Elles ne sont pas réinitialisées d'une exécution à l'autre du processus ; elles conservent leur valeur, comme si le processus était en fait une boucle infinie dans un langage de programmation classique.

La seule exception à l'exécution séquentielle des processus concerne l'affectation des signaux. Une instruction d'affectation spéciale est créée. Lorsqu'une telle instruction est exécutée, elle ne prend pas effet immédiatement. Elle est "enregistrée" par le simulateur, qui la traitera effectivement lorsque tous les processus auront été exécutés. C'est un piège classique des langages fonctionnels de description de matériel que de croire l'affectation de signal instantanée. Le paradoxe apparent provient du fait qu'après une telle affectation le signal n'a pas changé de valeur ...

# Le temps physique

- **Les phénomènes physiques ont une date physique**
- **On veut modéliser des phénomènes physiques**
- **Il faut modéliser le temps physique**

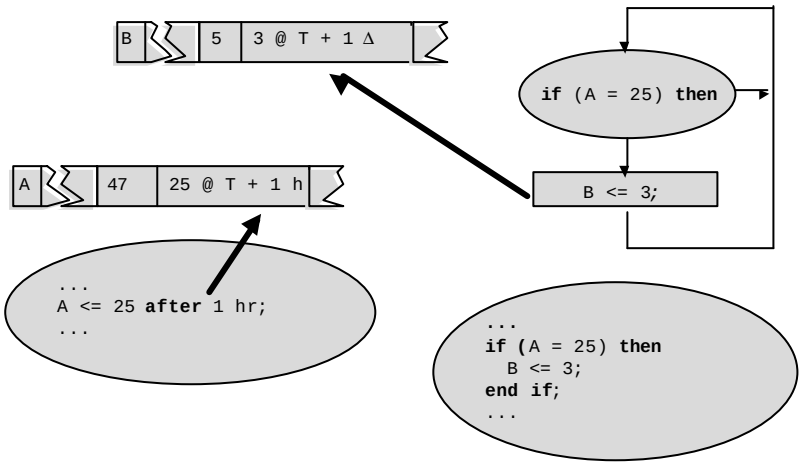
```
A <= 25 after 1 hr;
```



Les phénomènes physiques ont une date physique. Ils se produisent à un instant donné que l'on peut, en général, déterminer avec précision. Il en va ainsi en électronique. Les modifications d'état des composants se produisent à une date physique précise. Un langage de modélisation ne peut pas ignorer un tel phénomène. La simulation doit associer à chaque événement une date physique. Comment intégrer cette nouvelle contrainte dans le langage ? Quelle stratégie peut utiliser le simulateur pour prendre en compte le temps physique?

Le langage permet, lors de l'affectation de signaux, de préciser le délai physique après lequel la modification de valeur doit effectivement avoir lieu. Le simulateur ne modifiera la valeur du signal concerné que lorsque ce délai sera écoulé et non pas à la fin du pas de simulation en cours.

# Le signal et son échéancier






A chaque assignation de signal correspond une transaction avec la zone mémoire temporaire du simulateur où sont mémorisées les requêtes concernant les signaux. Chaque signal possède donc dans cette zone un échéancier destiné à recevoir ces requêtes. Une requête est composée de la valeur que doit prendre le signal et de la date complète (date physique plus date symbolique) du futur événement. Lorsque tous les processus ont été exécutés et que toutes les requêtes ont été enregistrées, le simulateur peut ainsi déterminer quel signal doit changer de valeur lors du prochain pas de simulation. Il modifie donc les signaux en question et avance le temps d'un pas (1 $\Delta$), puis il recommence l'exécution des processus.

# La synchronisation des processus

- **Les deltas n'ont pas de durée physique.**
  - ***Le temps physique ne peut donc pas évoluer en cours de simulation.***
- **Pour la plupart des processus, l'exécution pas à pas est très inefficace**
  - ***Comment exécuter les processus lorsque c'est nécessaire et seulement lorsque c'est nécessaire ?***
- **Le simulateur, à chaque pas de simulation ne "réveille" que les processus dont les entrées ont changé :**
  - ***Il doit donc être en mesure de déterminer quels signaux sont des entrées du processus***
  - ***Et s'ils ont changé de valeur depuis la dernière exécution du processus (ou depuis le dernier pas de simulation)***



Le mécanisme que nous venons de présenter ne fonctionne pas car les deltas n'ont pas de durée physique. Il s'en suit qu'une exécution par pas de un delta ne peut faire progresser le temps physique. La solution à ce problème est simple et élégante. Elle possède en outre la bonne propriété d'accélérer considérablement les simulations :

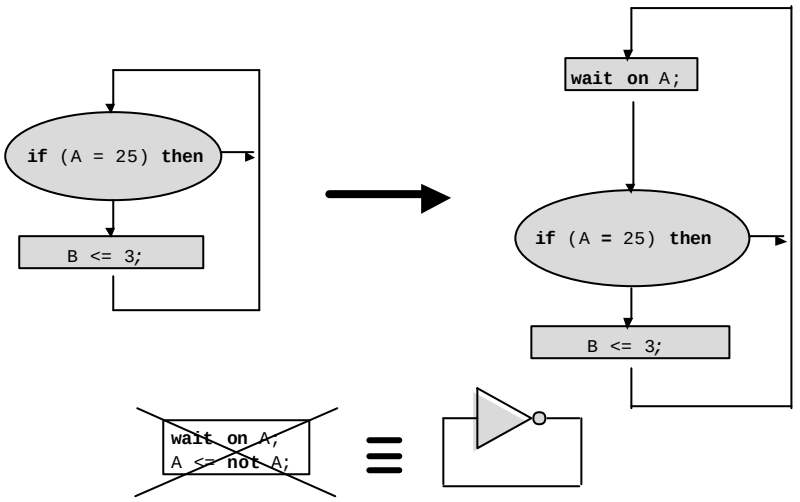
Afin de réduire les temps de simulation on optimise la stratégie du simulateur. Les exécutions inutiles sont supprimées. Le simulateur n'utilise donc pas la technique du pas à pas. Lorsque le simulateur doit relancer l'exécution des processus, il consulte tout d'abord les échéanciers de tous les signaux pour déterminer la date du prochain événement ainsi que la liste des signaux qui doivent changer de valeur à cette date. Cette analyse permet :

- de relancer l'exécution aux seuls instants « intéressants » ;

- de ne réveiller que les processus concernés.

Ainsi, si les deux entrées d'un processus décrivant une porte ET n'ont pas changé, il est inutile de relancer son exécution puisque la sortie conserverait la même valeur.

# La synchronisation des processus





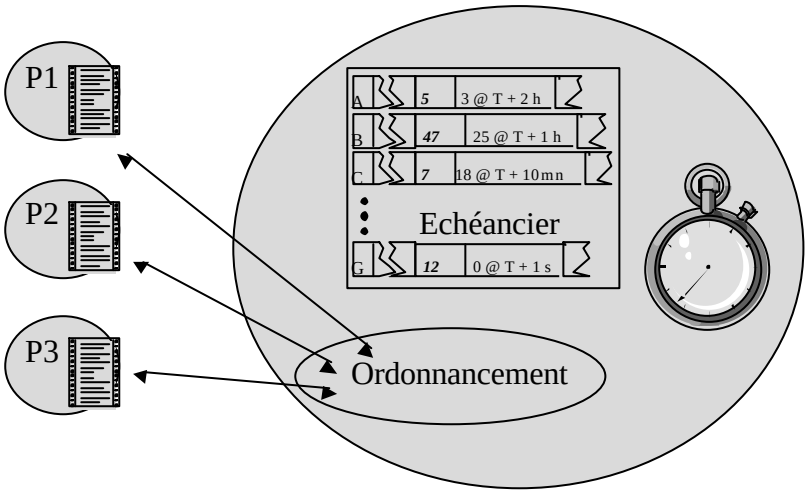

Le simulateur peut désormais faire avancer le temps d'une valeur quelconque. Parmi tous les événements à venir, il détermine le ou les plus proches, les réalise effectivement et positionne la date courante en conséquence. Il fait ainsi l'économie de pas de simulation inutiles.

En fait, cette optimisation est même indispensable puisque la durée physique d'un $\Delta$ est nulle. Les durées symboliques ont pour seule fonction d'ordonner les événements les uns par rapport aux autres. Si le simulateur ne faisait avancer le temps que d'un $\Delta$ à la fois, le temps physique n'avancerait pas.

Chaque processus est une boucle infinie. Il doit donc comporter au moins un point d'arrêt. Faute de quoi, le simulateur est dans l'obligation de l'exécuter indéfiniment (temps physique et temps symbolique n'évoluent pas, il s'agit d'un processus cyclique sans point d'arrêt). Ce point d'arrêt est muni d'une condition de réveil qui est la liste des signaux auxquels le processus est "sensible". Le simulateur ne réveille le processus que lorsque l'un des signaux de cette liste a changé de valeur.

Attention, la condition de réveil associée au point d'arrêt ne doit pas être toujours vraie, sinon seul le temps symbolique évolue. Les processus cycliques avec point d'arrêt sont un autre piège des langages fonctionnels de description de matériel.

# Le moteur de simulation






Le moteur de simulation est le chef d'orchestre. Il maîtrise le temps. C'est lui qui décide de faire avancer le temps d'une quantité qu'il détermine après examen des échéanciers des signaux et des points d'arrêt des processus.

Voici la description d'une étape de simulation :

1) La date courante est choisie

2) Les signaux sont positionnés à leur nouvelle valeur

3) Pour chaque processus :

- Le moteur de simulation teste la condition associée au point d'arrêt

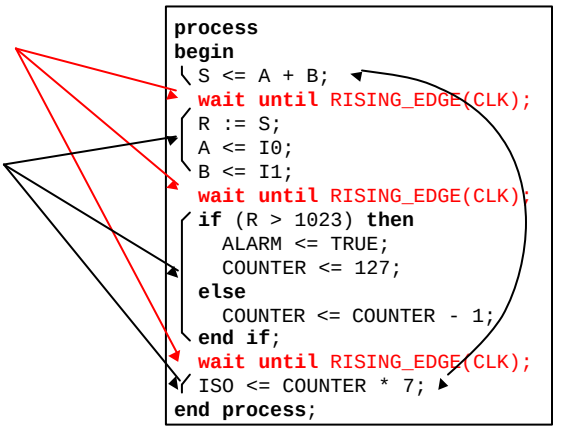
- Si la condition associée au point d'arrêt n'est pas vérifiée, le processus n'est pas réveillé et on passe au processus suivant

- Sinon, pendant l'exécution du processus le moteur de simulation enregistre les requêtes émises dans les échéanciers des signaux concernés. Lorsque l'exécution rencontre un point d'arrêt, le moteur de simulation enregistre la condition associée et passe au processus suivant

Les processus sont traités dans un ordre quelconque ; c'est pourquoi les variables restent locales aux processus : en cours d'exécution on ne peut pas savoir si elles ont déjà été affectées ou non.

# Le processus et le temps

- **Le temps avance lorsque le processus est arrêté sur un point de synchronisation.**
- **Entre deux points de synchronisation le temps n'avance pas.**
- **On parle d'exécution à temps nul.**
- **L 'affectation des signaux est différée.**
- **Le processus est une boucle infinie.**






Un processus est un programme séquentiel infini car il est rebouclé sur lui-même. Il doit contenir un ou plusieurs points de synchronisation (on parle aussi de points d'arrêt). Lorsque l'exécution arrive à un point de synchronisation elle stoppe et le simulateur passe la main à un autre processus. Pendant l'exécution le temps est comme suspendu, il n'avance pas. C'est ce qui permet l'émulation du parallélisme. Par contre, lorsque le processus est stoppé sur un point d'arrêt, le simulateur fait progresser le temps (après que chaque processus soit également stoppé).

Ainsi, les points d 'arrêt sont le lieu de l'écoulement du temps alors que les segments séquentiels qui les séparent sont sans durée.

L'affectation des signaux est reportée à la fin de la phase d'exécution de tous les processus. L'affectation des variables est instantanée. Dans l'exemple ci-dessus après la ligne :

```
R := S;
```

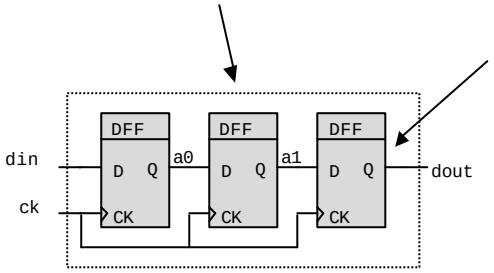
la variable `R` a déjà changé de valeur alors qu'après la ligne :

```
A <= I0;
```

le signal `A` possède toujours la même valeur.

# Exemple

```
entity REGS is

  port (CK, DIN : in BIT;
        DOUT : out BIT);

end entity REGS;
```




```
architecture ARC of REGS is

  signal A0, A1 : BIT;

begin

  REGS_PR : process(CK)

  begin

    if (CK = '1') then
      A0   <= DIN;
      A1   <= A0;
      DOUT <= A1;
    end if;

  end process REGS_PR;

end architecture ARC;
```



Le principe de l'affectation retardée des signaux permet de modéliser le registre à décalage sans risque de transparence abusive.

L'entité est la vision externe (la boîte et ses entrées/sorties). L'architecture décrit l'intérieur de la boîte.

Cette architecture est constituée d'un unique processus, REGS_PR. Elle possède deux signaux internes, A0 et A1. Les entrées CK et DIN peuvent être utilisées en lecture dans l'architecture comme s'il s'agissait de signaux normaux. La sortie DOUT peut être utilisée en écriture dans l'architecture comme s'il s'agissait d'un signal normal.

La liste de sensibilité du processus REGS_PR (CK) exprime la condition de réveil. La structure de contrôle "if then" permet d'enrichir la condition de réveil du processus REGS_PR. Cette condition s'exprime ainsi :

- "il y a un événement sur le signal CK (changement de valeur)"

- "et la nouvelle valeur de CK est '1'"

c'est à dire :

"il y a un front montant de CK"

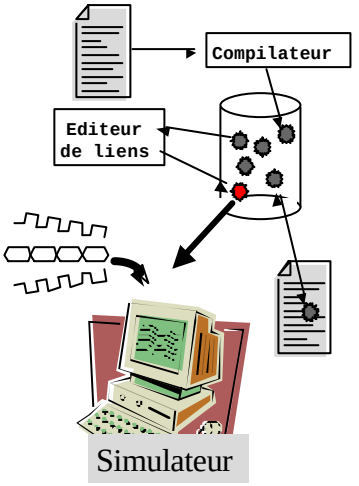
L'ordre dans lequel sont écrites les trois affectations de signaux est indifférent. Pourquoi ?

# Plan

- Introduction
- Principes de la simulation évènementielle
- **Organisation informatique des projets**
- **Les unités de compilation**
- **Syntaxe**
  - ***VHDL séquentiel***
  - ***VHDL concurrent***
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# Structure du langage

- **Pour simuler il faut :**
  - ***Analyser (compilation) les fichiers source.***
  - ***Elaborer (édition de lien) les résultats de compilation.***
  - ***On simule toujours le résultat d'une élaboration.***
- **Le résultat d'une analyse ou d'une élaboration est rangé dans une bibliothèque.**
- **Le contenu des bibliothèques est réutilisable dans un programme, à condition de l'avoir déclaré.**






Les différents programmes dont l'assemblage produit un modèle sont écrits en VHDL et conservés dans différents fichiers. Pour simuler le modèle, il faut d'abord compiler ces fichiers ; en VHDL, on appelle cette compilation l'analyse. Le compilateur range le résultat des analyses dans une bibliothèque ("library" en anglais). L'utilisateur peut utiliser autant de bibliothèques qu'il le désire pour ranger ses résultats d'analyse. Il faut donc préciser au compilateur quelle est la bibliothèque dans laquelle il doit ranger les programmes analysés.

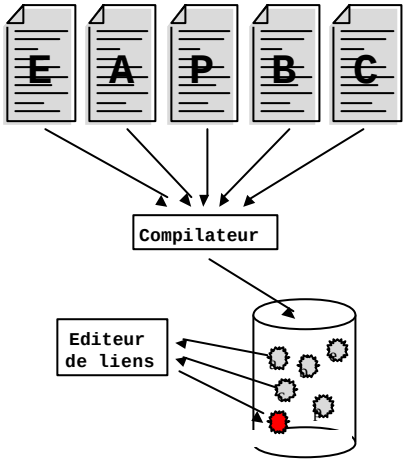
A l'intérieur d'un programme VHDL, on peut faire référence à un objet préalablement analysé que l'on désire utiliser. Pour ce faire, il faut, bien sûr, préciser dans quelle bibliothèque l'objet analysé a été rangé.

Lorsque tous les programmes ont été convenablement analysés (sans erreurs), il faut les fusionner pour produire un unique fichier exécutable. C'est ce qu'on appelle l'édition de liens dans la plupart des langages de programmation classiques et l'élaboration en VHDL. On élabore toujours le résultat d'une analyse. Parmi tous les résultats d'analyse on élabore toujours celui de plus haut niveau et celui de plus haut niveau seulement (celui qui regroupe tous les autres dans une description structurelle hiérarchique, l'équivalent du schéma de plus haut niveau). Là encore, l'éditeur de liens a besoin de connaître le nom de la bibliothèque dans laquelle il doit ranger le résultat de l'élaboration.

C'est le résultat de l'élaboration que l'on simule.

# Structure du langage

- **Il existe 5 unités de compilation :**
  - ***L'entité***
  - ***L'architecture***
  - ***La déclaration de paquetage***
  - ***Le corps de paquetage***
  - ***La configuration***
- **Seules les unités de compilation peuvent être analysées (compilées)**
- **Seuls les résultats de compilation d'architecture ou de configuration peuvent être élaborés.**






Il existe 5 types d'objets analysables. En VHDL on appelle un objet analysable "unité de compilation". Parmi ces 5 types d'unités de compilation, 2 seulement peuvent, après analyse, être élaborés. Une unité de compilation doit être écrite dans un seul fichier. Il ne peut pas y avoir d'unité de compilation répartie sur plusieurs fichiers. Il peut, par contre, y avoir plusieurs unités de compilation dans le même fichier.

# Les bibliothèques

- **Le nom symbolique `WORK` désigne la bibliothèque cible de la compilation.**
- **Pour accéder à une bibliothèque, il est nécessaire de la déclarer :**

```
library BIB;
use BIB.PAQ.OBJ;
```

- **La création, gestion des bibliothèques ne font pas partie de la définition du langage VHDL. Chaque environnement constructeur est donc différent à cet égard.**
- **Les bibliothèques peuvent être partagées entre utilisateurs.**



Les bibliothèques sont la clef d'une bonne organisation de projet. Elles permettent à plusieurs concepteurs de travailler ensemble sur le même projet. Elles rendent le langage indépendant du système d'exploitation de la machine hôte. Chaque bibliothèque a un nom, par lequel on peut y faire référence à l'intérieur d'une unité de compilation. Il existe des bibliothèques accessibles en lecture seulement (on peut utiliser leur contenu, mais ni le modifier, ni y ranger de nouvelles unités de compilation). Parmi celles-ci, on peut citer la bibliothèque STD, qui contient deux paquetages définis par la norme VHDL. Il existe également des bibliothèques dites "utilisateur", ouvertes en lecture et en écriture. Parmi ces dernière, il en est une qui possède deux noms : la bibliothèque dans laquelle le compilateur et l'éditeur de liens rangent le résultat de leur action. Cette bibliothèque possède un nom normal, comme toute bibliothèque "utilisateur" et un nom générique qui est : WORK. Ainsi, dans un programme VHDL, on peut faire référence à WORK lorsque l'on ne veut pas utiliser le nom normal de la bibliothèque.

Par défaut, les bibliothèques STD et WORK n'ont pas besoin d'être déclarées pour être utilisables. Tout se passe comme si un programme VHDL commençait toujours par :

```
library STD;
library WORK;
```

Pour utiliser le contenu d'un paquetage, il faut déclarer la bibliothèque dans laquelle il se trouve (sauf, éventuellement, si c'est `WORK`) et le paquetage :

```
use BIBLIOTHEQUE.PAQUETAGE.all;
```

ou, si l'on ne veut pas utiliser tout le paquetage :

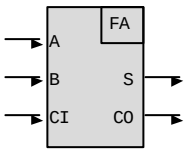
```
use BIBLIOTHEQUE.PAQUETAGE.OBJET;
```

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- **Les unités de compilation**
- **Syntaxe**
  - ***VHDL séquentiel***
  - ***VHDL concurrent***
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# L'entité

- **C'est la description d'interface. Elle précise :**
  - ***Le nom du module (FA)***
  - ***Ses ports d'entrée - sortie***
    - **Nom**
    - **Direction (in, out, inout, ...)**
    - **Type (BIT, BIT_VECTOR, BOOLEAN, INTEGER, ...)**
- **Les ports sont des signaux utilisables dans l'architecture associée, sans qu'il soit nécessaire de les y redéclarer.**

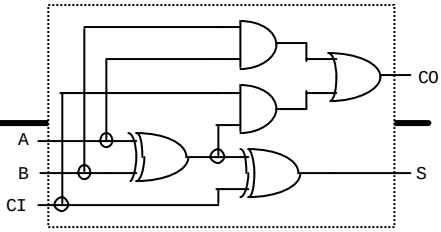



```
entity FA is

  port(A, B, CI: in  BIT;
       S, CO:    out BIT);

end entity FA;
```



L'entité est l'équivalent du symbole dans les représentations schématiques. Elle précise le nom de l'objet et ses entrées et sorties. Les entrées et sorties s'appellent des "ports" en VHDL. Pour chacun d'eux, on donne le nom, la direction et le type. Chaque port sera visible de l'intérieur de la boîte (architecture), comme s'il s'agissait d'un signal.

# L'architecture




- **C'est la description interne. Elle est toujours associée à une entité.**
- **Il peut y avoir plusieurs architectures pour la même entité.**

```
architecture BEV of FA is
begin
  PR: process(A, B, CI)
  begin
    S    <= A xor B xor CI;
    COUT <= (A and B) or (A and CI) or
            (B and CI);
  end process PR;
end architecture BEV;
```

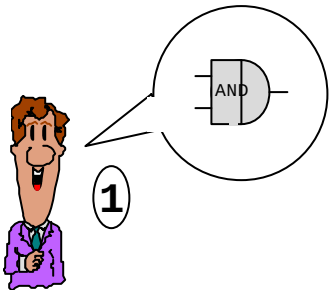
```
architecture DF of FA is
  signal I0, I1: BIT;
begin
  P1: process(A, B)
  begin
    I0 <= A xor B;
    I1 <= A and B;
  end process P1;
  P2: process(I0, I1, CI)
  begin
    S <= I0 xor CI;
    CO <= (I0 and CI) or I1;
  end process P2;
end architecture DF;
```

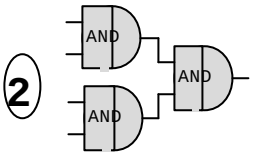


L'architecture décrit l'intérieur de la boîte. C'est là que l'on va expliciter la fonction de l'objet grâce à des processus concurrents. Une architecture est toujours associée à une entité. Il n'est pas possible d'analyser une architecture si l'entité associée n'a pas déjà été analysée.

# Structure du langage

- **VHDL est un langage déclaratif :**
  - ***On déclare un objet avant de l'utiliser.***
    - **Variable**
    - **Signal**
    - **Constante**
    - **Fonction**
    - **Procédure**
    - **Composant**
    - **...**
  - ***Il existe des zones de déclaration***
  - ***On ne peut pas déclarer n'importe quoi n'importe où.***






Le besoin de fiabilité du langage (la volonté de réduire tant que faire se peut le nombre d'erreurs dues au concepteur) se traduit par des contraintes fortes. Entre autres, on peut citer l'aspect fortement déclaratif et l'aspect fortement typé de VHDL. Il faut déclarer systématiquement et de façon très complète tous les objets que l'on veut utiliser ; cela permet au compilateur de procéder à des vérifications assez poussées. La typologie forte des objets est également un moyen puissant de vérification à la compilation - les conversions implicites de type sont interdites - mais aussi à la simulation - les débordements de tableau ou de type contraint provoquent un arrêt de la simulation, assorti d'un message très explicite. C'est une différence notable avec des langages plus classiques, comme le C, où les conversions implicites sont autorisées (on peut transformer un entier en réel et vice versa sans précaution particulière) et où les débordements de tableau produisent des erreurs d'exécution à retardement accompagnées de messages moins explicites ("segmentation fault - core dumped").

# L'architecture

- **Zone de déclarations**
- **Corps (instructions concurrentes)**
  - ***Processus***
  - ***Affectation concurrente de signal***
  - ***Instanciation de composant ...***
- **Exécution parallèle. Ordre d'écriture indifférent.**

```
architecture DF of FA is

  signal SI: BIT;
  component MAJ
    port(X, Y, Z: in BIT; M: out BIT);
  end component;

begin

  PP: process(SI, CI)
  begin
    S  <= SI xor CI;
  end process PP;

  SI <= A  xor B;

  RET: MAJ port map(X => A, Y => B,
                    Z => CI, M => CO);

end architecture DF;
```



L'architecture possède sa zone de déclaration. Celle-ci est située entre la ligne "architecture ARCH of ENT is" et le mot clef "begin". C'est dans cette zone que l'on déclare les signaux internes, les fonctions et procédures, les composants, etc. que l'on souhaite utiliser pour décrire la fonctionnalité de l'objet.

Attention :
- il est inutile - et donc interdit - de déclarer les ports d'entrée et de sortie comme signaux. Ils ont déjà été déclarés dans l'entité.
- il est impossible - et donc interdit - de déclarer des variables dans la zone de déclarations d'une architecture. Cela voudrait en effet dire que ces variables sont globales à tous les processus de l'architecture. Or les variables sont locales aux processus. Ce sont les signaux, et les signaux seulement, qui sont responsables de la communication entre les processus.

Le corps de l'architecture est la partie située entre le mot clef "begin" et le mot clef "is". Il contient des instructions concurrentes (destinées à s'exécuter en parallèle). L'ordre dans lequel sont écrites ces instructions n'a pas d'importance. Un exemple d'une telle instruction est le processus. Il existe d'autres instructions concurrentes comme l'instanciation de composant (qui sert à la description structurelle) et l'assignation concurrente de signal qui est un raccourci d'écriture pour des processus très simples.

# L'architecture

- **Les ports de l'entité associée sont considérés comme des signaux accessibles dans l'architecture, sans qu'il soit nécessaire de les y redéclarer.**
- **Les ports d'entrée (`in`) sont accessibles en lecture seulement.**
- **Les ports de sortie (`out`) sont accessibles en écriture seulement.**

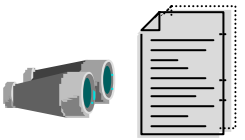
```
entity FA is

  port(A, B, CI: in  BIT;
       S, CO:    out BIT);

end entity FA;

architecture BUG of FA is

  signal A, B, CI, S, CO: BIT;

begin

  A  <= B or CI;

  CO <= (A and B) or (A and CI) or
        (B and CI);

  S <= (A or B or CI) and not CO or
       A and B and CI;

end architecture BUG;
```



Parmi les sources d'erreur fréquentes on peut citer la redéclaration des ports d'entrée et de sortie, la tentative d'affectation d'un port d'entrée ou la tentative de lecture d'un port de sortie.

# Le paquetage

- **C'est une collection d'objets réutilisables**
- **Il se décompose en deux unités de compilation :**
  - ***Déclaration de paquetage***
  - ***Corps de paquetage***
- **Le contenu de la déclaration de paquetage est "visible" depuis une autre unité de compilation si elle en a déclaré l'utilisation**
- **Le contenu du corps de paquetage est "invisible" des autres unités de compilation**



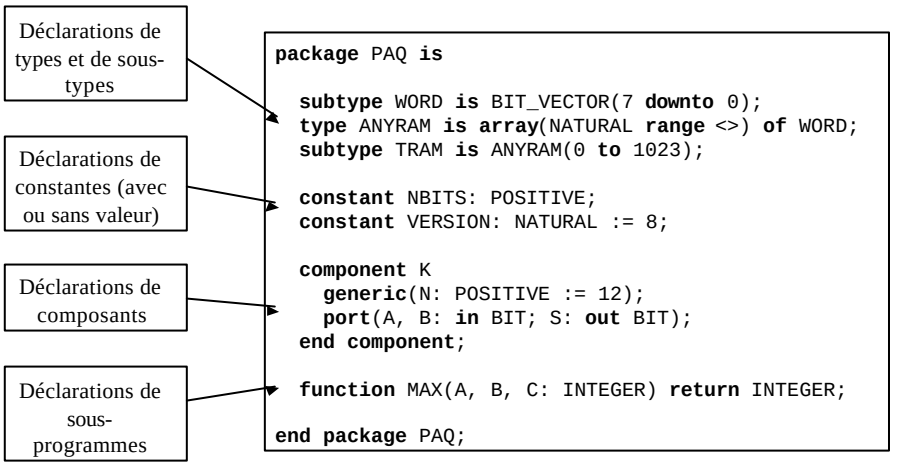


Le paquetage est un moyen pratique de ranger des objets lorsque l'on pense les réutiliser dans plusieurs unités de compilation. Il se décompose en deux unités de compilation :
- la déclaration de paquetage qui contient les objets "visibles" de l'extérieur - c'est la partie émergée de l'iceberg. Par exemple, si une constante est rangée dans la déclaration de paquetage, elle est accessible (utilisable) dans toute unité de compilation qui déclare l'utilisation du paquetage.
- le corps de paquetage qui contient les objets "invisibles" de l'extérieur mais nécessaires à l'utilisation du paquetage. Si une constante y est rangée, elle ne pourra pas être utilisée ailleurs que dans le corps de paquetage.

On peut regrouper dans le paquetage des fonctions ou des procédures, des constantes, des composants, des déclarations de type, etc. Attention : une unité de compilation ne peut pas contenir d'autres unités de compilation. On ne peut donc pas ranger des entités, des architectures ou des paquetages dans un paquetage.

Lorsqu'une unité de compilation utilise un paquetage, la déclaration de celui-ci doit être analysée en premier. Le corps du paquetage ne doit être analysé qu'avant l'élaboration d'un module où il est utilisé. Ainsi, on peut modifier à loisir un corps de paquetage ; il y a peu à faire pour remettre l'ensemble du projet à jour. Par contre, une modification de la déclaration entraîne forcément une nouvelle analyse de toutes les unités qui l'utilisent, et donc de celles qui en dépendent, ...

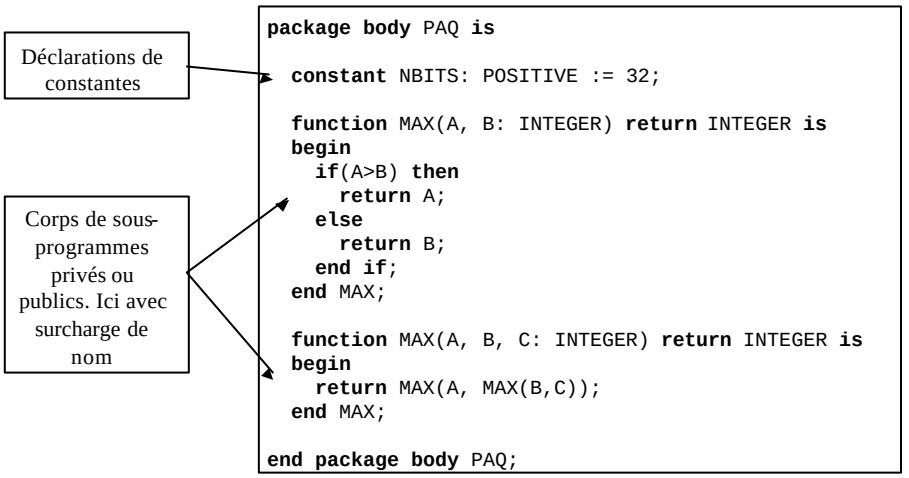
# Le paquetage : déclaration






La déclaration de paquetage peut contenir des déclarations de types et de sous-types, des déclarations de constante (avec ou sans sans valeur), des déclarations de composants, des déclarations de sous-programmes (fonctions et procédures), ...

# Le paquetage : corps






Le corps de paquetage précise les valeurs des constantes et contient les corps de sous-programmes.

# Les composants

- **Description structurelle = assemblage de dispositifs plus simples.**
- **Basée sur la déclaration et l'instanciation d'un composant.**
- **Le composant n'est pas une unité de compilation. C'est le prototype d'une entité.**
- **L'utilisation des composants permet une conception descendante.**

```
entity AND3 is
  port(A0, A1, A2: in BIT;
       Z: out BIT);
end entity AND3;
architecture STR of AND3 is
  signal TMP: BIT;
  component AND2
    port(A, B: in BIT;
         C:    out BIT);
  end component;
begin
  I0: AND2 port map(A => A0,
                    B => A1,
                    C => TMP);
  I1: AND2 port map(A => A2,
                    B => TMP,
                    C => Z);
end architecture STR;
```



Si le composant n'existait pas, il faudrait l'inventer. En effet, une conception descendante implique que l'on puisse analyser une architecture dans laquelle sont instanciés des sous-objets alors que ces objets n'existent pas encore et ne sont donc *a fortiori* pas encore analysés. Grâce au composant on retarde le moment où il faudra effectivement décrire ces sous-objets. L'architecture d'un objet hiérarchique de niveau élevé est autonome. Elle contient ses propres déclarations de composants. Le compilateur peut donc vérifier la cohérence du code VHDL.

Attention : le composant n'est pas une unité de compilation. C'est un prototype, une déclaration d'intention. Il signifie aux yeux du compilateur qu'il existera plus tard un couple entité / architecture dont l'interface sera compatible avec celle qui est décrite.

On peut déclarer les composants dans un paquetage, afin de les utiliser dans plusieurs architectures sans avoir à les redéclarer à chaque fois.

# La configuration

- **Elle décrit les correspondances entre instances de composants et couples entité / architecture.**
- **C'est elle qu'on élabore, lorsqu'elle existe, pour pouvoir simuler.**
- **Le simulateur en a toujours besoin.**
- **Les synthétiseurs ne la comprennent pas toujours.**

```
configuration CF1 of AND2 is
  for CMP
  end for;
end configuration CF1;

library BIB;

configuration CF2 of AND3 is
  for STR
    for I0: AND2
      use configuration BIB.CF1;
    end for;
    for I1: AND2
      use entity BIB.AND2(CMP);
    end for;
  end for;
end configuration CF2;
```



Avant de pouvoir élaborer, il faudra préciser les associations entre composants et couples entité / architecture existants. C'est le rôle de la configuration. Cette cinquième et dernière unité de compilation sert à préciser par quoi l'éditeur de lien doit remplacer les instances de composants pour que l'ensemble du projet soit effectivement simulable. C'est donc la configuration qui est l'unité de compilation de plus haut niveau d'un projet hiérarchique. C'est elle dont on élabore le résultat d'analyse pour pouvoir simuler.

Les synthétiseurs ne sont pas toujours capables de l'interpréter. Ils disposent en effet parfois d'une méthode de configuration par défaut. Cette méthode exige en général :
- que le nom du composant soit le même que celui de l'entité à laquelle il doit être associé (ce qui interdit d'associer différentes instances du même composant à plusieurs entités différentes)
- que l'entité associée n'ait qu'une seule architecture

Il est parfois également nécessaire de séparer physiquement (dans des fichiers différents) le couple entité / architecture et sa configuration pour éviter les messages d'erreur du synthétiseur à la lecture de la configuration.

# La configuration

- **Elle peut être “à plat” ou hiérarchique (on élabore alors celle de plus haut niveau).**
- **La clause `for all` la simplifie beaucoup.**
- **Même vide elle contient des informations.**

```
configuration C3 of E3 is
  for A3
    for I2: K2
      use entity BIB.E2(A2);
      for A2
        for all: K1
          use entity BIB.E1(A1);
        end for;
      end for;
    end for;
  end for;
end configuration C3;
```



Deux méthodes permettent de décrire la configuration d’un projet hiérarchique complexe :
- la configuration hiérarchique. De loin la meilleure parce que la plus simple et la plus facile à maintenir. Pour chaque couple entité / architecture, on crée une configuration (même vide, s’il n’y a pas de composants instanciés). Dans chaque configuration on utilise la clause **use configuration** et jamais la clause **use entity**. Ainsi, au lieu de déclarer que l’on va remplacer les instances du composant K1 par le couple entité / architecture E1(A1) de la bibliothèque BIB, on déclare que l’on va remplacer les instances du composant K1 par le couple entité / architecture défini dans la configuration C1 de la bibliothèque BIB. Cette configuration, même vide, contient les informations entité / architecture.
- la configuration “à plat” dont on voit un exemple simple ci-dessus. Beaucoup plus lourde et difficile à maintenir, elle doit être réservée aux cas très simples.

# Configuration immédiate

- **On peut associer directement composants instanciés et couples entité / architecture au moyen d'une clause de configuration immédiate. Une clause de configuration immédiate a la syntaxe :**

```
for all: NOM_DE_COMPOSANT use entity ENTITE(ARCHITECTURE);
```

- **ou bien :**

```
for ETIQ1, ETIQ2: NOM_DE_COMPOSANT use entity ENTITE(ARCHITECTURE);
```

- **Les clauses de configuration immédiate doivent apparaître après les déclarations locales et avant le corps de l'architecture.**



La configuration immédiate, bien que moins lourde en apparence, doit être réservée aux cas où le concepteur veut pouvoir simuler très vite une hypothèse et ne s'intéresse pas à la réutilisabilité ou à la maintenabilité de son travail.

# Instanciations d'entité ou de configurations

- **On peut renoncer aux composants et instancier directement :**
  - ***Un couple entité – architecture***
  - ***Une configuration***

```
entity AND3 is
  port(A0, A1, A2: in BIT;
       Z: out BIT);
end entity AND3;
architecture STR of AND3 is
  signal TMP: BIT;
begin
  I0: entity WORK.AND2(CMP)
      port map(A => A0, B => A1, C => TMP);
  I1: configuration WORK.CF2
      port map(A => A2, B => TMP, C => Z);
end architecture STR;
```



Cette approche de la description structurelle ne permet pas la conception descendante car, pour compiler l'architecture AND3(STR) if faut avoir déjà compilé, donc conçu, l'architecture AND2(CMP) et la configuration CF2.

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- **Syntaxe**
  - ***VHDL séquentiel***
  - ***VHDL concurrent***
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# Les processus

- **Un processus est un programme séquentiel.**
- **Les objets qu'il manipule ont un type.**
- **Il manipule les objets à l'aide d'opérateurs.**
- **Son déroulement est décrit à l 'aide de structures de contrôle.**

```
int max, i, sum, moy;
int tab[10];
...
max=0;
for(i=0; i<10; i++) {
  if(tab[i]>max)
    max=tab[i];
  sum+=tab[i];
  }
moy=sum/10;
```

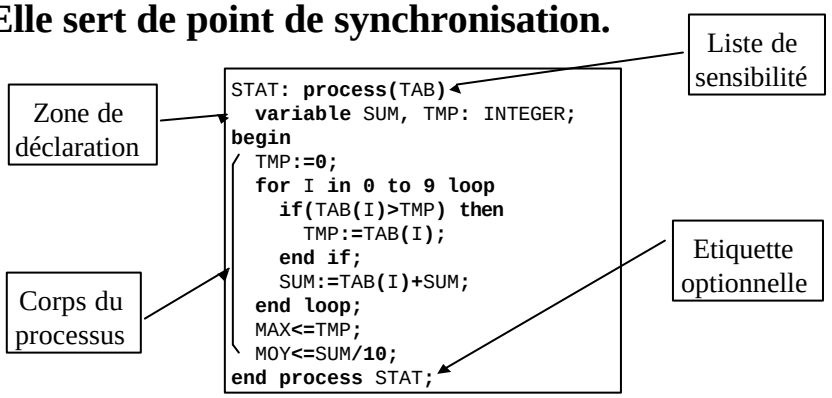
```
variable MAX, SUM, MOY: INTEGER;
type T is array(0 to 9) of INTEGER;
variable TAB: T;
...
MAX:=0;
for I in 0 to 9 loop
  if(TAB(I)>MAX) then
    MAX:=TAB(I);
  end if;
  SUM:=TAB(I)+SUM;
end loop;
MOY:=SUM/10;
```



Nous allons tout d 'abord passer en revue la syntaxe dédiée aux aspects séquentiels du langage. C'est cette partie du langage qui va nous permettre d'écrire des processus. A quelques exceptions près la syntaxe est très proche de celle d'autres langages de programmation, comme le langage C.

# Les processus avec liste de sensibilité

- **Un processus peut posséder une liste de sensibilité.**
- **La liste de sensibilité contient des signaux.**
- **Elle sert de point de synchronisation.**






Chaque fois que l'un des signaux de la liste de sensibilité change de valeur le processus est à nouveau exécuté.

# Exemples de processus

- **Ce processus décrit une fonction combinatoire des signaux X et Y. Laquelle ? Dès que X ou Y est modifié, la fonction est recalculée. Ecrivez un processus combinatoire décrivant la fonction majorité de trois signaux. Idem avec l'additionneur.**
- **Ce processus est synchrone. Il décrit le fonctionnement d'une bascule D. Quelles en sont l'horloge, l'entrée et la sortie ? Expliquez le fonctionnement de ce processus. Ecrivez un processus décrivant une bascule synchrone sur front montant d'une horloge avec reset asynchrone.**

```
signal X, Y, Z: BIT;
P1: process(X, Y)
begin
  if X = '1' then
    Z <= '1';
  elsif Y = '1' then
    Z <= '1';
  else
    Z <= '0';
  end if;
end process P1;
```

```
signal X, Y, Z: BIT;
P2: process(Z)
begin
  if Z = '1' then
    Y <= X;
  end if;
end process P2;
```



On peut classer les processus en trois catégories :

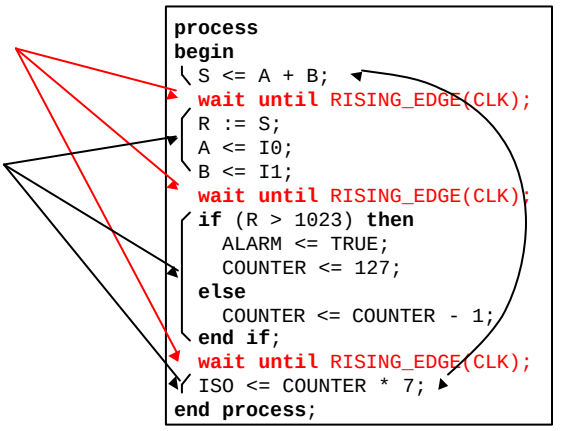
1) les processus correspondant à une partie combinatoire de circuit
- leur liste de sensibilité, s'ils en ont une, contient toutes les entrées de la partie combinatoire
- ils sont donc évalués dès que l'une au moins des entrées change de valeur

2) les processus dits « synchrones » correspondant à une structure de mémorisation de circuit (registres, bascules D, latches, RAM, etc.)
- leur liste de sensibilité , s'ils en ont une, contient les horloges
- l'écriture est plus délicate il faut bien réfléchir aux conditions dans lesquelles le processus doit être évalué afin d'en déduire la liste de sensibilité

3) les autres processus qui mélangent combinatoire et synchrone.

# Le processus et le temps

- **Le temps avance lorsque le processus est arrêté sur un point de synchronisation.**
- **Entre deux points de synchronisation le temps n'avance pas.**
- **On parle d'exécution à temps nul.**
- **L 'affectation des signaux est différée.**
- **Le processus est une boucle infinie.**






Un processus est un programme séquentiel infini car il est rebouclé sur lui-même. Il doit contenir un ou plusieurs points de synchronisation (on parle aussi de points d'arrêt). Lorsque l'exécution arrive à un point de synchronisation elle stoppe et le simulateur passe la main à un autre processus. Pendant l'exécution le temps est comme suspendu, il n'avance pas. C'est ce qui permet l'émulation du parallélisme. Par contre, lorsque le processus est stoppé sur un point d'arrêt, le simulateur fait progresser le temps (après que chaque processus soit également stoppé).

Ainsi, les points d 'arrêt sont le lieu de l'écoulement du temps alors que les segments séquentiels qui les séparent sont sans durée.

L'affectation des signaux est reportée à la fin de la phase d'exécution de tous les processus. L'affectation des variables est instantanée. Dans l'exemple ci-dessus après la ligne :

```
R := S;
```

la variable `R` a déjà changé de valeur alors qu'après la ligne :

```
A <= I0;
```

le signal `A` possède toujours la même valeur.

# Commentaires, identificateurs, littéraux, ...

- **Les commentaires commencent par deux tirets (`--`) et s'étendent jusqu'à la fin de la ligne.**
- **Les identificateurs sont des suites de lettres, chiffres et souligné (_). Ils commencent nécessairement par une lettre. VHDL ne tient pas compte de la casse.**
- **Les littéraux sont des valeurs explicites :**
  - `45` ***et*** `7.89` ***sont des littéraux numériques***
  - `"chaine de caractères"`
  - `'C'` ***est un littéral caractère***
  - `"000111010110"`**,** `B"000111010110"`**,** `O"726"` ***et*** `X"1E6"` ***sont des littéraux chaîne de bits***
  - `null` ***est un littéral pointeur***
- **Les expressions sont terminées par un point virgule (`;`).**



Il n'existe pas, comme en langage C, de commentaire sur plusieurs lignes. Du fait de la grande complexité du langage VHDL le commentaire est très utile. Il faut l'employer sans modération.

Il n'est parfois pas possible de déterminer le type d'un littéral au vu de sa seule expression. Par exemple, `"000111010110"` peut aussi bien représenter une chaîne de caractères qu'une chaîne de bits. Le contexte ou la qualification permettent de lever les ambiguïtés.

# Classes d ’objets

- **Les conteneurs d’information appartiennent à l’une des trois classes que définit le langage VHDL :**
  - ***Variables, semblables aux variables de tout autre langage de programmation, elles sont dédiées à la programmation séquentielle classique.***
  - ***Constantes, là encore semblables à ce que l’on rencontre dans d’autres langages.***
  - ***Signaux, spécificité de VHDL, ils sont dédiés à la programmation parallèle et, plus précisément, à l’échange d’informations entre plusieurs programmes parallèles.***
- **Pour éviter les confusions les affectations de valeur sont notées différemment selon la classe :**
  - `A := 178` ***pour les variables et les constantes***
  - `S <= 178` ***pour les signaux***



Le concept de classe des objets est très important. Il définit ce qu’il est possible de faire d’un objet. Un objet ne peut changer de classe mais sa valeur peut être utilisée pour modifier la valeur d’un objet d’une autre classe. Ainsi : `S <= A` est valide même si `S` est un signal et `A` une variable.

# Initialisation des variables et signaux

- **Par défaut une variable ou un signal est initialisé en début de simulation à la valeur de gauche de son type.**
- **On peut initialiser variables et signaux à une autre valeur lors de leur déclaration :**

```
signal S: INTEGER := 0;
...
variable V: BOOLEAN := TRUE;
```

- **C'est très souvent une mauvaise idée car on se prive ainsi d'un excellent mécanisme de détection des défauts de « reset ».**

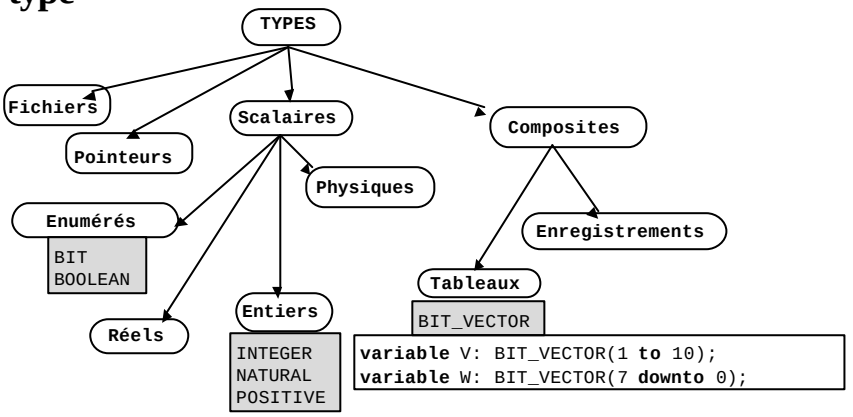


Le mécanisme d'initialisation par défaut des variables et signaux est souvent très utile pour détecter des erreurs de conception. Ainsi, le type STD_ULOGIC a pour valeur gauche (qui est donc sa valeur d'initialisation) le littéral 'U', pour UNINITIALIZED. Initialiser un signal ou une variable de ce type lors de sa déclaration c'est renoncer à détecter un défaut de conception dans la phase d'initialisation du circuit.

Les synthétiseurs n'utilisent pas les valeurs d'initialisation.

# Typologie

- **Signaux, variables et constantes ont toujours un type**






VHDL, comme ADA ou Pascal, est un langage fortement typé. Tout objet appartient à un type clairement défini. Le langage fournit des types de base et des constructeurs à partir desquels l'utilisateur peut définir sa propre typologie.

VHDL servant à décrire du matériel, un certain effort a été fait pour la représentation de types de données adaptés.

Les types les plus courants (parce que déclarés dans le paquetage STANDARD de la bibliothèque STD) sont :

- INTEGER, type des nombres entiers
- NATURAL, type des nombres entiers positifs ou nuls
- POSITIVE, type des nombres entiers positifs
- BOOLEAN, type des booléens (valeur FALSE ou TRUE)
- BIT, type à deux valeurs ‘0’ et ‘1’, à ne pas confondre avec 0 et 1
- BIT_VECTOR, type des tableaux à une dimension de BIT (les bus)

Attention : le type BIT_VECTOR est non contraint. C’est à dire qu’il n’a pas de taille *a priori*. Lorsque l’on déclare un objet de ce type il ne peut pas avoir une taille infinie. Il faut donc déclarer sa taille. En fait, on déclare l’intervalle de définition de son indice.

# Les types entiers

- **Le type entier s'appuie sur le type entier de la machine hôte. Il est requis que ce soit un type entier au moins sur 32 bits. On peut définir des sous-types :**

```
type INTEGER is range DEPENDANT_DE_LA_MACHINE;
```

- **Les types `NATURAL`, et `POSITIVE`, sont des sous types "intervalle", (`range`) du même type entier de base.**

```
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
```

- **Un sous type hérite des propriétés de son type père. Des erreurs de compatibilité peuvent survenir lors de l'affectation.**
- **La notation `INTEGER'HIGH` représente le plus grand élément du type `INTEGER`. Il est dépendant de la machine. La notation `'HIGH` est un attribut de type, comme en ADA.**



Toutes les expressions numériques entières sont du type prédéfini “entier_universel”.

Pour chaque type entier il existe une fonction de conversion implicite avec “entier_universel”.

Un environnement VHDL doit permettre la déclaration d’un type entier contenant l’intervalle [-2147483647..+2147483647].

Les opérateurs arithmétiques sont prédéfinis pour tous les types entiers.

Le seul type entier prédéfini est `INTEGER`. Son intervalle de définition dépend de la machine mais il contient l’intervalle [-2147483647..+2147483647].

Les sous-types intervalle des types entiers (`NATURAL`, `POSITIVE` et ceux définis par le programmeur) permettent la vérification statique et dynamique de contraintes. Il faut les utiliser aussi souvent que possible.

# Les types entiers

- **Les types entiers servent à définir des index de tableaux, des indices de boucle, des données, ...**
- **Ils sont définis à partir d'un type de base, en donnant les bornes et le sens (croissant ou décroissant) :**

```
subtype UN_A_DIX is NATURAL range (1 to 10);
subtype DIX_A_UN is NATURAL range (10 downto 1);
```

- **Attention aux différences entre types et sous-types :**

```
type UN_A_DIX is range 1 to 10;
type DIX_A_UN is range 10 downto 1;
```

- **Il doit naturellement y avoir compatibilité entre le type de base et les bornes.**
- **Attention : un type dans lequel l'ordre des bornes et l'orientation sont différents est un type VIDE**



Les types entiers restreints sont très pratiques pour contrôler les débordements en cours d'exécution. Il sont également utiles aux synthétiseurs qui peuvent traduire la dynamique déclarée en nombre de bits.

Attention : il ne faut pas confondre un type et un sous-type. Un sous-type hérite des propriétés de son type père avec lequel il reste compatible. Par exemple, on ne peut pas additionner deux objets des types UN_A_DIX et DIX_A_UN, alors qu'il est possible de le faire avec deux objets des sous-types UN_A_DIX et DIX_A_UN ci-dessus définis.

# Les type réels

- **Le type REAL s'appuie sur l'architecture physique de la machine hôte du système VHDL. Le LRM dit qu'il émule le comportement mathématique des nombres réels, et il est requis que sa dynamique permette la représentation des nombres de `-1.0E+38` à `1.0E+38`**
- **En VHDL, un nombre réel s'écrit :**

```
+/-nombre.nombre{E+/-nombre}
```

- **Exemples :**

```
A := 1.0;
A := 1.0E10;
A := 1.5E-20;
```



Toutes les expressions numériques réelles sont du type prédéfini “réel_universel”.

Pour chaque type réel il existe une fonction de conversion implicite avec “réel_universel”.

Un environnement VHDL doit permettre la déclaration d’un type réel contenant l’intervalle [-1E38..+1E38] avec une précision de six chiffres, au moins.

Les opérateurs arithmétiques sont prédéfinis pour tous les types réels.

Le seul type réel prédéfini est REAL. Son intervalle de définition dépend de la machine mais il contient l’intervalle [-1E38..+1E38].

# Les types physiques

- **VHDL permet de définir des types physiques, qui sont destinées à représenter une grandeur physique, comme le temps, la tension, etc. Un type physique est la combinaison d'un type entier et d'un système d'unité...**
- **Le type `TIME`, est le seul type physique prédéfini :**

```
type TIME is range DEPENDANT_DE_LA_MACHINE
Units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  Ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
end units;
```



Paradoxalement, la plupart des synthétiseurs n'acceptent pas les types physiques. L'intérêt que ces types présentent pour la modélisation est assez évident. Cela ne doit pas faire oublier que VHDL n'est pas un langage adapté à la description des phénomènes électriques.

# Les types énumérés

- **Un type énuméré est un type défini par énumération exhaustive :**

```
type COULEURS is (ROUGE, JAUNE, BLEU, VERT, ORANGE);
type QUATRE_ETATS is ('X', '0', '1', 'Z');
type STD_ULOGIC is (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ’W’, ’L’, ’H’, ’-’)
```

- **L’ordre dans lequel un type énuméré est déclaré est important. Par exemple, un signal ou une variable du type énuméré** `T` **prend la valeur** `T’LEFT` **à l’initialisation.**



Les types énumérés sont très pratiques pour déclarer les différents états d’une machine à états sans pour autant définir leur codage. On peut ainsi laisser le soin au synthétiseur de choisir le codage optimal.

# Les types énumérés prédéfinis

```
type BOOLEAN is (FALSE, TRUE);
type BIT is ('0', '1');
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
type CHARACTER is
     (   NUL,     SOH,     STX,     ETX,     EOT,     ENQ,     ACK,     BEL,
         BS,      HT,      LF,      VT,      FF,      CR,      SO,      SI,
         DLE,     DC1,     DC2,     DC3,     DC4,     NAK,     SYN,     ETB,
         CAN,     EM,      SUB,     ESC,     FSP,     GSP,     RSP,     USP,
         ' ',     '!',     '"',     '#',     '$',     '%',     '&',     ''',
         '(',     ')',     '*',     '+',     ',',     '-',     '.',     '/',
         '0',     '1',     '2',     '3',     '4',     '5',     '6',     '7',
         '8',     '9',     ':',     ';',     '<',     '=',     '>',     '?',
         '@',     'A',     'B',     'C',     'D',     'E',     'F',     'G',
         'H',     'I',     'J',     'K',     'L',     'M',     'N',     'O',
         'P',     'Q',     'R',     'S',     'T',     'U',     'V',     'W',
         'X',     'Y',     'Z',     '[',     '\',     ']',     '^',     '_',
         '`',     'a',     'b',     'c',     'd',     'e',     'f',     'g',
         'h',     'i',     'j',     'k',     'l',     'm',     'n',     'o',
         'p',     'q',     'r',     's',     't',     'u',     'v',     'w',
         'x',     'y',     'z',     '{',     '|',     '}',     '~',
         'DEL', etc.);
```



Un certain nombre de types énumérés sont définis dans le paquetage STANDARD de la bibliothèque STD.

# Les types tableau

- **Les types tableau sont des collections d'objets identiques indexées par des intervalles sur un type entier ou énuméré.**
- **Exemple :**

```
type BUS is array (0 to 31) of BIT;
type RAM is array (0 to 1024, 0 to 31) of BIT;
type PRIX is range 0 to INTEGER'HIGH
Units
  balles;
  francs = 100 balles;
  thunes = 5 francs;
  plaques = 10000 francs;
  patates = 100 plaques;
end units;
type COULEURS is (BLANC, BLEU, VERT, ROUGE, JAUNE, NOIR, ARC_EN_CIEL);
type PRIX_PEINTURES is array (COULEUR range BLANC to NOIR) of PRIX;
```



Les tableaux à plus d'une dimension ne sont pas toujours acceptés par les synthétiseurs.

Il ne faut pas confondre les tableaux à plusieurs dimensions et les tableaux de tableaux.

# Les tableaux non contraints

- **Il est possible de déclarer un type tableau dont la taille n'est pas connue ; ainsi le type** `BIT_VECTOR` **:**

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

- **Les tableaux non contraints servent, par exemple, à décrire des paramètres de procédure ou de fonction. Il ne peut pas exister d'objets de taille variable. Pour créer un objet de type** `BIT_VECTOR`**, il faut préciser sa dimension effective :**

```
subtype TYPE_BUS is BIT_VECTOR(0 to 31);
variable VARIABLE_BUS1: TYPE_BUS;
variable VARIABLE_BUS2: BIT_VECTOR(0 to 31);
```



Les sous-types peuvent permettre de définir un sous-type contraint à partir d'un type non contraint. Ainsi, la déclaration d'objets de ce sous-type ne nécessitera pas de déclaration de contrainte.

# Le type STRING

## VHDL définit les chaînes de caractères ainsi :

```
type STRING is array (POSITIVE range <>) of CHARACTER;
"Ceci est une chaine"     -- STRING
```

## Certaines notations sont ambiguës et peuvent ne pas être résolues par le contexte d'évaluation.

```
'1'              -- BIT ou CHARACTER ?
B"01010101"      -- BIT_VECTOR en binaire
O"0120768"       -- BIT_VECTOR en octal
X"0134DF54"      -- BIT_VECTOR en hexadécimal
"01010101"       -- BIT_VECTOR ou STRING ?
```

## On est alors amené à utiliser une notation qualifiée :

```
BIT_VECTOR'("01010101")
```



Le problème de qualification se pose surtout avec le type `STRING`, lors des entrées - sorties sur fichiers.

# Les enregistrements

- **Un enregistrement est un objet, dont les composantes sont hétérogènes.**
- **Exemple :**

```
type OPTYPE is (ADD, SUB, MPY, DIV, JMP);
type INSTRUCTION is
record
  OPCODE: OPTYPE;
  SRC: INTEGER;
  DST: INTEGER;
end record;
```

- **Contrairement à la plupart des langages de programmation, il n'y a pas d'enregistrements avec variantes en VHDL.**



Les enregistrements sont identiques à ceux de la plupart des langages de programmation classiques.

# Les pointeurs

- **Bien que ce concept soit très éloigné du matériel, on peut créer en VHDL des objets de type pointeur pour les structures de données dynamiques.**
- **Les objets dynamiques sont créés au moyen de l'ordre** `new`
- **La destruction se fait au moyen de la procédure** `deallocate` **qui est automatiquement déclarée en même temps que le type** `access`
- **Exemple :**

```
type FIFO_ELEMENT is array(0 to 3) of STD_LOGIC;
type FIFO_ACCESS is access FIFO_ELEMENT;
variable FIFO_PTR: FIFO_ACCESS;
FIFO_PTR := new FIFO_ELEMENT;
dealocate(FIFO_PTR);
```



Est-il utile de rappeler que les synthétiseurs ne les acceptent pas ?

“ Le langage VHDL est utilisé principalement pour la description de matériel électronique numérique ” (1)

“ Le matériel électronique numérique ignore la notion de pointeur “ (2)

“ Les pointeurs sont une source préoccupante d’erreurs en programmation de machines informatiques “ (3)

(1) et (2) => “ les pointeurs ne sont pas très utiles en VHDL “ (4)

(3) et (4) => “ N’utilisez pas les pointeurs (sauf cas de force majeure) “

# Les pointeurs

- **Un objet dynamique est créé et élaboré suivant les règles générales des objets VHDL. Dans le cas où l'on créé des listes chaînées, les objets se référençant eux-même , il est possible de faire ce qu'on appelle une déclaration incomplète :**

```
type T;
type T_PTR is access T;
type T is
record
  VALEUR: INTEGER;
  SUIVANT: T_PTR;
end record;
type PI is access INTEGER;
```

```
variable V, W: T_PTR;
variable VI: PI;
...
V := new T'(1, null);
V.SUIVANT := new T'(2, new T'(3, null));
V.SUIVANT.SUIVANT.SUIVANT := new T;
W := V.SUIVANT.SUIVANT.SUIVANT;
W.all := (4, null);
VI.all := W.VALEUR = 4;
```



Les déclarations incomplètes offrent la possibilité de créer des listes chaînées. Les variables sont les seuls objets qui peuvent être de type `access`.

# Les types fichiers

- **Un fichier est un objet externe au système VHDL permettant d'échanger des données avec l'extérieur.**
- **Un fichier est une suite d'enregistrements séquentiels du même type de base (scalaire, enregistrement ou tableau contraint), qu'on peut lire ou écrire.**
- **Un type fichier se déclare par :**

```
type FT is file of TM;
```

- **Dès qu'un type fichier est déclaré, VHDL crée des sous-programmes associés :**
  - ***Procédure d'ouverture, fermeture, fonction de test de fin de fichier :***

```
procedure FILE_OPEN (file F: FT; External_Name: in STRING;
   Open_Kind: in FILE_OPEN_KIND := READ_MODE);
procedure FILE_OPEN (Status: out FILE_OPEN_STATUS; file F: FT;
   External_Name: in STRING; Open_Kind: in FILE_OPEN_KIND := READ_MODE);
procedure FILE_CLOSE (file F: FT);
function ENDFILE (file F: FT) return BOOLEAN;
```



Bien que potentiellement visibles simultanément par plusieurs processus (il suffit pour cela de déclarer des fichiers pointant sur le même fichier physique dans plusieurs processus), les fichiers ne doivent pas être utilisés pour contourner la règle d'or de la communication entre processus : "le seul vecteur d'échange d'informations entre processus concurrents est le signal".

L'organisation des fichiers au sens du système d'exploitation n'est pas précisée dans la norme.

# Les types fichiers

- **Dès qu'un type fichier est déclaré, VHDL crée des sous-programmes associés :**
  - ***Procédure de lecture et d'écriture :***

```
procedure READ (file F: FT; VALUE: out TM);
procedure WRITE (file F: FT; VALUE: in TM);
```

- **On peut déclarer un objet `F1` de type `FT` en par :**

```
file F1: FT; FILE_OPEN(F1, "toto.txt"); -- lecture
file F1: FT is "toto.txt"; -- lecture
file F1: FT open WRITE_MODE is "toto.txt"; -- ecriture
```

- **Un fichier ne peut être ouvert qu'en lecture (`READ_MODE`) ou en écriture (`WRITE_MODE`).**
- **Remarque : lorsque `TM` est un type tableau non contraint la procédure `READ` est déclarée :**

```
procedure READ (file F: FT; VALUE: out TM; LENGTH: out NATURAL);
```

# Les fichiers de texte

- Le paquetage TEXTIO de la bibliothèque STD regroupe des outils d'entrée - sortie sur fichiers de texte.
- Il existe un type fichier TEXT
- Il existe deux fichiers TEXT prédéfinis : INPUT et OUTPUT

```
use STD.TEXTIO.all;
```

```
type TEXT is file of STRING;
```

```
file INPUT: TEXT open READ_MODE
  is "STD_INPUT";
file OUTPUT: TEXT open WRITE_MODE
  is "STD_OUTPUT";
```

```
file TOTO: TEXT is "toto.txt";
```

```
file TITI: TEXT open WRITE_MODE
  is "titi.txt";
```



Le paquetage TEXTIO de la bibliothèque STD contient les définitions nécessaires aux manipulations de fichier de texte. A la différence du paquetage STANDARD de cette même bibliothèque (qui contient les définitions des types les plus utilisés) son utilisation n'est pas déclarée par défaut. Il est donc nécessaire de préciser que l'on désire y faire référence avec la clause **use** appropriée.

Le type fichier TEXT permet la lecture et l'écriture de fichiers ASCII. L'entrée standard et la sortie standard sont des fichiers de type TEXT.

Le mode d'utilisation d'un fichier doit être précisé lors de la déclaration du fichier. Ce mode ne peut être que **in** ou **out** (pas de mode **inout**).

# Les fichiers de texte

- **En plus des fonctions et procédures implicites on peut accéder aux fichiers** `TEXT` **par ligne en utilisant des objets de type** `LINE`
- **Les procédures** `READLINE` **et** `WRITELINE` **permettent la lecture et l'écriture d'une ligne de fichier de texte.**
- **Les procédures** `READ` **et** `WRITE` **permettent la lecture et l'écriture dans la ligne ; elles sont définies pour les types** `BIT`, `BIT_VECTOR`, `BOOLEAN`, `CHARACTER`, `INTEGER`, `REAL`, `STRING` **et** `TIME`

```
type LINE is access STRING;
```

```
procedure READLINE(F: in TEXT;
  L: out LINE);
procedure WRITELINE(F: out TEXT;
  L: out LINE);
```

```
procedure READ(L: inout LINE;
  VALUE: out BIT;
  GOOD: out BOOLEAN);
procedure READ(L: inout LINE;
  VALUE: out BIT);
procedure WRITE(L: inout LINE;
  VALUE: in BIT;
  JUSTIFIED: in SIDE := RIGHT;
  FIELD: in WIDTH := 0);
```



Les accès aux fichiers de type `TEXT` peuvent se faire par ligne. Le type `LINE` est un pointeur sur une chaîne de caractères (`STRING`) qui est automatiquement allouée et desallouée par les procédures `READLINE` et `WRITELINE`.

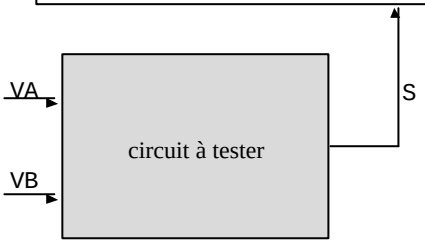
Les paramètres optionnels de la procédure `WRITE` sont :
- `JUSTIFIED`, de type `SIDE`, qui peut prendre deux valeurs : `RIGHT` et `LEFT`
- `FIELD`, de type `WIDTH` (`NATURAL`), qui représente la taille totale en nombre de caractères
- `DIGITS`, de type `NATURAL`, pour le type `REAL` seulement qui représente le nombre de chiffres après la virgule
- `UNIT`, de type `TIME`, pour le type `TIME` seulement qui représente l'unité et peut prendre une valeur parmi `fs`, `ps`, `ns`, `us`, `ms`, `sec`, `min`, `hr`

# Les fichiers

- Exemples d'utilisation : les environnements de simulation (lecture des entrées du circuit à tester dans un fichier, écriture des sorties dans un second fichier).

```
LECTURE: process
  variable L: LINE;
  file ENTREES: TEXT is "ent.dat";
  variable A: BIT_VECTOR(7 downto 0);
  variable B: NATURAL range 0 to 11;
begin
  READLINE(ENTREES, L);
  READ(L, A);
  VA <= A;
  READ(L, B);
  VB <= B;
  wait for 20 ns;
end process LECTURE;
```

```
ECRITURE: process(S)
  variable L: LINE;
  file SORTIES: TEXT open WRITE_MODE
    is "res.dat";
begin
  WRITE(L, S);
  WRITE(L, STRING'(" a la date "));
  WRITE(L, NOW);
  WRITELINE(SORTIES, L);
end process ECRITURE;
```






Les fichiers sont souvent utilisés dans les environnements de simulation.

# Les attributs de type

- **Les attributs permettent la consultation des types déclarés.**

```
type COULEURS is (ROUGE, JAUNE, BLEU, VERT, ORANGE);
type QUATRE_ETATS is ('X', '0', '1', 'Z');
```

- **Certains attributs sont déclarés automatiquement lors de la création du type. Exemples :**

```
T'BASE -- renvoie le type de base du type T
COULEURS'LEFT = ROUGE
COULEURS'RIGHT = ORANGE
QUATRE_ETATS'HIGH = 'Z'
QUATRE_ETATS'LOW = 'X'
```

- **Exercice : que valent ces cinq attributs pour ce type ?**

```
subtype A_L_ENVERS is COULEURS range ORANGE downto ROUGE;
```



Les attributs LEFT, RIGHT, HIGH et LOW ne sont définis que pour les types scalaires.

# Les attributs de type

- **Si une variable A est du type discret T**

```
T'POS(A) -- renvoie la position de A dans le type
T'VAL(N) -- renvoie la N-ième valeur dans le type
T'SUCC(A) -- renvoie le successeur de A
          -- T'SUCC(A) = T'VAL(T'POS(A) + 1)
T'PRED(A) -- renvoie le prédécesseur de A
          -- T'PRED(A) = T'VAL(T'POS(A) - 1)
T'LEFTOF(A) -- renvoie l'élément à gauche de A
            -- dans la déclaration du type
T'RIGHTOF(A) -- renvoie l'élément à droite de A
             -- dans la déclaration du type
```

- **VHDL étant un langage fortement typé, un appel de ces attributs pour lequel le résultat est hors du type provoque une erreur à la compilation ou, plus probablement, à l'exécution, suivant ce qui peut être détecté.**



Les types discrets sont les types entiers, physiques et énumérés.

# Les attributs de type tableau

```
type T is array (0 to 3, 7 downto 0) of BIT;
variable TAB: T;
TAB'LEFT(1)               -- renvoie 0
TAB'LEFT(2)               -- renvoie 7
TAB'RIGHT(1)              -- renvoie 3
TAB'RIGHT(2)              -- renvoie 0
TAB'HIGH(1)               -- renvoie 3
TAB'HIGH(2)               -- renvoie 7
TAB'LOW(1)                -- renvoie 0
TAB'LOW(2)                -- renvoie 0
TAB'RANGE(1)              -- renvoie 0 to 3
TAB'RANGE(2)              -- renvoie 7 downto 0
TAB'REVERSE_RANGE(2)      -- renvoie 0 to 7
TAB'REVERSE_RANGE(1)      -- renvoie 3 downto 0
TAB'LENGTH(1)             -- renvoie 4
TAB'LENGTH(2)             -- renvoie 8
```



Ces attributs n’ont de sens que sur les types tableau contraints. Ils sont très utiles lors de la déclaration de variables destinées à servir d’indice ou lors de l’écriture de sous-programmes capables de manipuler des paramètres de type tableau et de taille variable.

# La notation d'agrégat

```
type OPTYPE is (ADD, SUB, MPY, DIV, JMP)
type T is array (1 to 5) of OPTYPE;
type U is
  record
    R1, R2, R3: INTEGER range 0 to 31;
    OP: OPTYPE;
  end record;
variable A: T; variable B: U;
...
A := (ADD, SUB, MPY, DIV, JMP);
A := (ADD, SUB, MPY, 5 => JMP, 4 => DIV);
A := (3 => ADD, SUB, MPY, JMP, DIV);
A := (ADD, 2 | 4 => MPY, others => DIV);
A := (SUB, 2 to 4 => DIV, 5 => JMP);
B := (0, 1, 2, ADD);
B := (OP => JMP, others => 0);
```



Très pratique pour initialiser les tableaux, la notation d'agrégat est aussi très riche.

Il n'est pas possible d'utiliser la notation positionnelle après avoir eu recours à la notation par dénomination au sein d'un agrégat.

Un agrégat a une seule valeur doit utiliser la notation par dénomination pour éviter toute confusion avec une expression parenthèsée.

# Les opérateurs

- **Logiques :** `and, or, nand, nor, xor, not`
- **Relationnels :** `=, /=, <, <=, >, >=`
- **Additifs :** `+, -, & (concaténation)`
- **De signe :** `+, -`
- **Multiplicatifs :** `*, /, mod, rem`

```
A = (A / B) * B + (A rem B)
signe(A rem B) = signe(A)
abs(A rem B) < abs(B)
(-A) / B = -(A / B) = A / (-B)
∃ N, A = B * N + (A mod B)
signe(A mod B) = signe(B)
abs(A mod B) < abs(B)
```

- **Divers :** `** (exponentiation), abs (valeur absolue)`



Les opérateurs sont très classiques. Attention toutefois à la définition précise des opérateurs multiplicatifs. Attention également à l'absence du « xnor » qui n'est pas défini.

# Les structures de contrôle

```
if C1 = -65 then
  A := 10;
  B := '0';
elsif C1 = -64 then
  A := 20;
  B := '1';
elsif C1 >= -63 and C1 <= -60 then
  A := 20;
  B := '0';
elsif C1 = -59 or C1 = 187 then
  A := 30;
  B := '0';
else
  A := 30;
  B := '1';
end if;
```

```
case C1 is
  when -65 => A := 10;
              B := '0';
  when -64 => A := 20;
              B := '1';
  when -63 to -60 => A := 20;
                     B := '0';
  when -61 | 187 => A := 30;
                    B := '0';
  when others => A := 30;
                 B := '1';
end case;
```



Les structures de contrôle appartiennent au domaine séquentiel. On ne peut donc les rencontrer qu'à l'intérieur d'un processus.

Le **if** est rigoureusement semblable à ce que l'on rencontre dans les langages de programmation séquentielle. Attention, toutefois, l'exemple montré ci-dessus est un seul et unique **if** ; il n'y a donc qu'un seul **end if**.

Le **case** doit contenir la liste de tous les choix possibles une fois et une seule fois chacune (d'où l'intérêt de la clause **others** ).

# Les structures de contrôle

- Les indices de boucles ne doivent pas être déclarés
- Les indices de boucle sont considérés comme des constantes à l'intérieur du corps de la boucle
- Les labels de boucle sont optionnels
- On peut altérer l'exécution par les clauses `next` et `exit`

```
L1: for I in 0 to 13 loop
  ...
  L2: loop
    ...
    L3: while NON_STOP_L3 loop
      ...
      exit L2 when STOP_L2;
      next L3 when SUITE_L3;
      if STOP_L1 then
        exit L1;
      end if;
    end loop L3;
  end loop L2;
end loop L1;
```



Les trois schémas itératifs proposés par les boucles et les altérations de boucle permettent toutes les combinaisons habituellement rencontrées en programmation.

# L'instruction `wait`

- **Un processus doit nécessairement posséder un ou plusieurs points d'arrêt.**
  - ***Soit implicite : une liste de signaux, dite liste de sensibilité, est déclarée dans l'en tête du processus. L'instruction `wait` équivalente (`wait on liste`) est placée à la fin du corps du processus. Il n'est pas possible de faire apparaître d'autres `wait`.***
  - ***Soit explicites : Il n'y a pas de liste de signaux dans l'entête. On peut alors faire apparaître plusieurs instructions `wait` dans le corps du processus.***
- **Forme complète de l'instruction `wait` :**

```
wait [on S1, S2, ...] [until CONDITION] [for DUREE];
```

`S1`, `S2` **doivent être des signaux**

`CONDITION` **est une expression booléenne**

`DUREE` **est le temps maximal d'attente**



Elle partage avec l'affectation de signal le titre d'instruction la plus complexe du langage. Elle est abondamment utilisée en modélisation de haut niveau, plus rarement lors de l'écriture de code en vue de la synthèse où on lui préférera la liste de sensibilité.

## `wait` : exemples

- **Eternel :**

```
wait;
```

- **Pas de condition, pas de durée :**

```
wait on S1, S2;
```

- **Pas de liste, pas de durée :**

```
wait until (S1 = '0') and (S2 > ORANGE);
```

- **Boucle équivalente :**

```
loop
  wait on S1, S2;
  exit when (S1 = '0') and (S2 > ORANGE);
end loop;
```

- **Attention si la condition ne contient aucun signal, on ne sort jamais du `wait`. Exemple classique :**

```
wait until NOW > 10 s;
```



Il est essentiel de retenir le mécanisme de l'instruction `wait` : en l'absence de limite de durée, seuls des événements sur des signaux peuvent provoquer la sortie de l'instruction et donc le réveil du processus.

## wait : exemples

### Ces deux processus sont équivalents

```
signal X, Y, Z: BIT;

PA: process(X, Y)

begin

  if X = '1' then
    Z <= '1';
  elsif Y = '1' then
    Z <= '1';
  else
    Z <= '0';
  end if;

end process PA;
```

```
signal X, Y, Z: BIT;

PA: process

begin

  if X = '1' then
    Z <= '1';
  elsif Y = '1' then
    Z <= '1';
  else
    Z <= '0';
  end if;
  wait on X, Y;

end process PA;
```



En VHDL, toute activité de calcul, assignation de valeur se fait à l'intérieur d'un processus.

Un processus est une suite d'instructions séquentielles. Un processus se synchronise sur des signaux en entrée (la liste de sensibilité), en attendant un changement d'état. Il effectue des calculs en un temps de simulation réputé nul et affecte (éventuellement) des signaux en sortie.

Un processus est une boucle infinie. Lorsqu'il arrive à la fin de son code le processus continue son exécution en reprenant au début. Les variables locales du processus ne sont pas réinitialisées entre deux pas de simulation.

Un processus qui n'a pas de signal en sortie est un processus « passif ». Il sert à effectuer des contrôles sur les signaux en entrée.

Le parallélisme de VHDL est un parallélisme entre processus.

# assert

- **Vérifie une propriété et génère un message d'erreur (voire stoppe la simulation) lorsqu'elle n'est pas vérifiée. Permet de contrôler l'utilisation conforme d'un modèle. VHDL procure l'instruction d'assertion pour réaliser cela :**

```
assert CONDITION
  [report MESSAGE]
  [severity NIVEAU];
```

`CONDITION` **: condition booléenne supposée vraie**

`MESSAGE` **: chaîne de caractères à imprimer**

`NIVEAU` **: niveau d'erreur pré-défini :** `NOTE`, `WARNING`, `ERROR`, `FAILURE`

- **Une assertion de niveau supérieur à** `ERROR`**, entraîne généralement la fin de la simulation.** `ERROR` **est le niveau par défaut.**



L'assertion, bien qu'alourdissant le code et bien que coûteuse en temps de calcul est un très bon moyen d'économiser sur le temps de mise au point d'un programme complexe.

# Affectations de signaux

- **Les affectations de signaux sont de deux types :**
  - ***Inertielle***

```
SIG1 <= reject 2 ns inertial 3 after 2 ns, 5 after 6 ns, 0 after 10 ns;
SIG1 <= inertial 3 after 2 ns, 5 after 6 ns, 0 after 10 ns;
SIG1 <= 3 after 2 ns, 5 after 6 ns, 0 after 10 ns;
```

  - ***Transport***

```
SIG1 <= transport 3 after 2 ns, 5 after 6 ns, 0 after 10 ns;
```

- **Le délai de réjection (période de coupure) :**
  - ***Inférieur au délai de la première transaction***
  - ***Par défaut est égal au délai de la première transaction***
  - ***Nul pour l'affectation de type transport***
- **Mise à jour de l'échéancier (affectation de type transport) :**
  - ***1) Destruction des anciennes transactions dont la date est égale ou postérieure à la première transaction de la nouvelle forme d'onde.***
  - ***2) Ajout des nouvelles transactions à la fin de l'échéancier.***



Un signal reçoit une forme d'onde, c'est à dire une liste de valeurs associées à des instants relatifs ordonnés.

Dans la réalité, une équipotentielle peut être plus complexe que cela : on peut lui associer de la mémorisation propre, elle peut interconnecter plusieurs émetteurs en écriture (logique à conflit), ou bien être totalement déconnectée (haute impédance). On peut lui associer des délais inertiels ou filtrants (capacités des fils).

VHDL tente de modéliser tous ces phénomènes au moyen de diverses technique.

# Affectations de signaux

- **Mise à jour de l'échéancier (affectation de type inertiel) :**
  - ***1) Destruction des anciennes transactions dont la date est égale ou postérieure à la première transaction de la nouvelle forme d'onde.***
  - ***2) Ajout des nouvelles transactions à la fin de l'échéancier.***
  - ***3) Les nouvelles transactions sont marquées.***
  - ***4) Les anciennes transactions dont la date est antérieure à la date de la première nouvelle transaction moins la limite de réjection sont marquées.***
  - ***5) Toute transaction précédant une transaction marquée de même valeur est marquée.***
  - ***6) La transaction qui pilote actuellement le signal est marquée.***
  - ***7) Toutes les transactions non marquées sont détruites.***



L'affectation de type inertiel est l'affectation par défaut. C'est aussi la plus complexe et la moins « naturelle ». Dans la pratique elle est généralement utilisée avec des formes d'onde à une seule transaction sur des signaux dont l'échéancier ne contient qu'une seule transaction : la transaction active. Elle est alors beaucoup plus simple à comprendre …

# Affectation de signaux

- **Soit un signal `A` dont l'échéancier à la date `1 ns` est :**

```
[0@0ns][5@3ns][1@5ns][3@6ns][8@12ns]
```

- **On exécute l'affectation :**

```
A <= transport 1 after 5 ns, 2 after 10 ns, 3 after 15 ns;
```

- **L'échéancier devient :**

```
R1: [0@0ns][5@3ns][1@5ns]
R2: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
```

- **Si on exécute l'affectation :**

```
A <= 1 after 5 ns, 2 after 10 ns, 3 after 15 ns; -- rejection = 5 ns
```

- **L'échéancier devient :**

```
R1: [0@0ns][5@3ns][1@5ns]
R2: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
R3: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
R4: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
R5: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
R6: [0@0ns][5@3ns][1@5ns][1@6ns][2@11ns][3@16ns]
R7: [0@0ns][1@5ns][1@6ns][2@11ns][3@16ns]
```

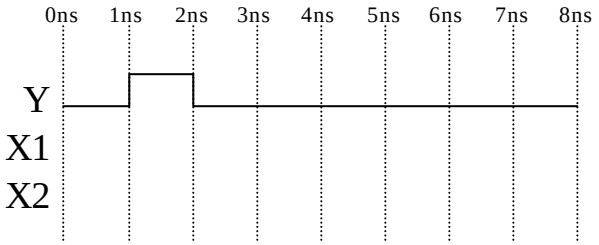


L'exemple ci-dessus montre l'évolution d'un échéancier dans le cas d'une affectation de type transport et dans le cas d'une affectation de type inertiel.

# Exercice

- **Dessinez les chronogrammes des signaux X1 et X2**

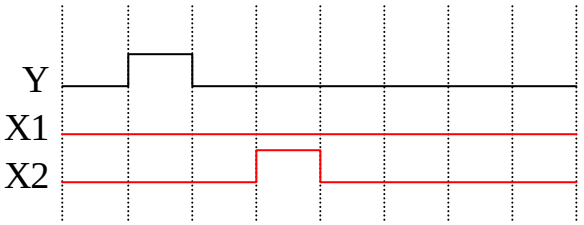
```
process(Y)
begin
  X1 <= Y after 2 ns;
  X2 <= transport Y after 2 ns;
end process;
```



# Exercice (corrigé)

- **Dessinez les chronogrammes des signaux X1 et X2**

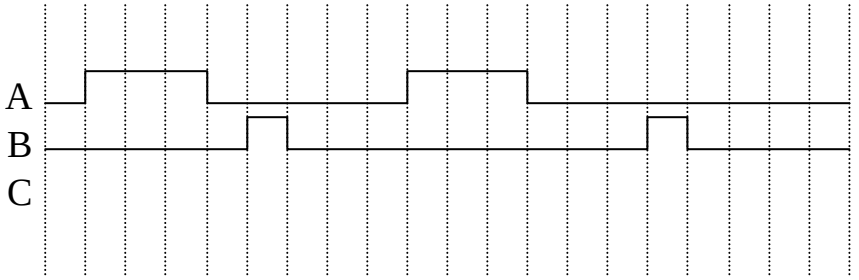
```
process(Y)
begin
  X1 <= Y after 2 ns;
  X2 <= transport Y after 2 ns;
end process;
```



# Exercice

- **Dessinez le chronogramme du signal** `C`

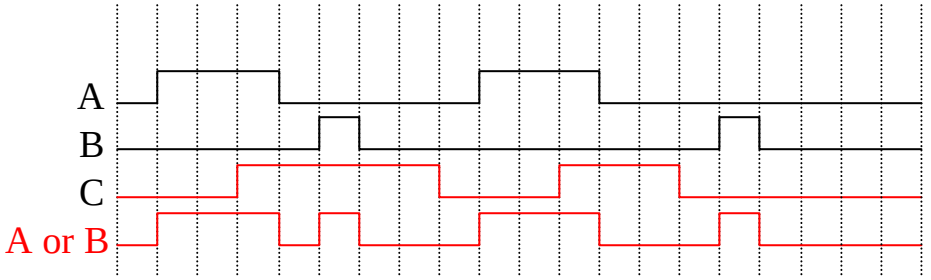
```
process(A, B)
begin
  C <= A or B after 2 ns;
end process;
```



# Exercice

- **Dessinez le chronogramme du signal** c

```
process(A, B)
begin
  C <= A or B after 2 ns;
end process;
```



# Attributs sur signaux

- **Si `S` est un signal, VHDL définit des attributs qui permettent d'interroger l'état du signal :**
  - ***L'attribut*** `S'EVENT` ***est une fonction de résultat*** `BOOLEAN`***. Elle renvoie*** `TRUE` ***si le signal a changé d'état au cours du cycle de simulation. Exemple : pour détecter le front montant d'une horloge :***

```
if (CLK = '1') and CLK'EVENT then
  Q <= D;
end if;
```

  - ***L'attribut*** `S'LAST_EVENT` ***est une fonction de résultat*** `TIME`***. Elle renvoie le temps écoulé depuis le dernier événement sur*** `S`***.***
  - ***L'attribut*** `S'LAST_VALUE` ***est une fonction dont le résultat est du type de*** `S`***. Elle renvoie la valeur du signal*** `S` ***avant le dernier événement sur*** `S`***.***
  - ***L'attribut*** `S'STABLE(T)` ***est un signal de type*** `BOOLEAN`***. Il vaut*** `TRUE` ***si aucun événement n'a eu lieu sur le signal*** `S` ***pendant*** `T`***.***



Les attributs sur signaux concernent soit les transactions soit les événements (un événement est une transaction qui provoque un changement de valeur du signal).

# Attributs sur signaux

- **Les attributs suivants portent sur les transactions et non plus sur les événements :**
  - ***L'attribut*** `S'ACTIVE` ***est une fonction de résultat*** `BOOLEAN`***. Elle renvoie*** `TRUE` ***si le signal a fait l'objet d'une exécution de transaction au cours du cycle de simulation.***
  - ***L'attribut*** `S'LAST_ACTIVE` ***est une fonction de résultat*** `TIME`***. Elle renvoie le temps écoulé depuis la dernière exécution de transaction sur*** `S`**.**
  - ***L'attribut*** `S'QUIET(T)` ***est un signal de type*** `BOOLEAN`***. Il vaut*** `TRUE` ***si aucune exécution de transaction n'a eu lieu sur le signal*** `S` ***pendant*** `T`**.**
- **L'attribut** `S'TRANSACTION` **est un signal de type** `BIT` **qui change d'état à chaque exécution de transaction sur** `S`**.**
- **L'attribut** `S'DELAYED(T)` **est un signal du type de** `S`**. Il se comporte comme le signal** `R` **dans :**

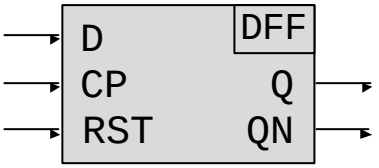
```
R <= transport S after T;
```



Les attributs sur transaction sont d'une utilité très relative.

# Exercice

- **Concevoir le (ou les) processus réalisant la fonctionnalité d'une bascule D dont les ports sont :**
  - `RST` ***est un « reset » asynchrone actif à*** `'0'`
  - `CP` ***est l'horloge ; la bascule est synchrone sur front montant de*** `CP`
  - `D` ***est l 'entrée de la bascule***
  - `Q` ***est la sortie,*** `QN` ***la sortie inversée***

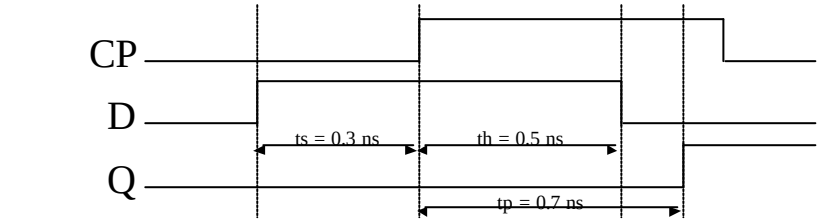


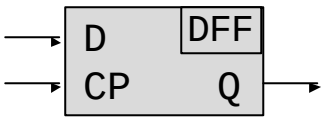
# Exercice (corrigé)

```
P1: process(CP, RST)
begin
  if (RST = '0') then
    Q <= '0';
  elsif (CP = '1' and CP 'event) then
    Q <= D;
  end if;
end process P1;

QN <= not Q;
```

# Exercice

- **Concevoir le (ou les) processus réalisant la fonctionnalité d'une bascule D dont les temps de *setup*, de *hold* et de propagation sont représentés par le chronogramme suivant :**



# Exercice (corrigé)

```
P1: process(CP)
begin
  if (CP = '1') then
    Q <= D after 0.7 ns;
    assert (D'LAST_EVENT > 0.3 ns)
      report "Setup violation"
      severity WARNING;
  end if;
end process P1;

P2: process(D)
begin
  assert ((CP'LAST_EVENT > 0.5 ns) or
          (CP'LAST_VALUE /= '0'))
      report "Hold violation"
      severity WARNING;
end process P2;
```

# Les sous-programmes

- **Il existe deux types de sous-programmes :**
  - ***Les fonctions***
    - **Paramètres d'appels non modifiables**
    - **Valeur retournée**
  - ***Les procédures***
    - **Paramètres d'appels modifiables**
    - **Pas de valeur retournée**
- **Ils appartiennent au domaine séquentiel**
- **Ils ont le même intérêt que dans tous les langages de programmation.**

```
function F(A: BIT_VECTOR)
  return BIT is
  ...
begin
  ...
end function F;
```

```
procedure P(A: in BIT_VECTOR;
            S: out BIT) is
  ...
begin
  ...
end procedure P;
```



Les sous-programmes servent à regrouper des instructions séquentielles que l'on désire utiliser souvent.

Les paramètres des procédures sont déclarés sous une forme qui ressemble à celle des ports d'entrée-sortie des entités ou des composants. En plus de leur nom on précise leur mode et leur type. Les paramètres de mode **`in`** peuvent être utilisés dans le corps de la procédure mais non modifiés. Les paramètres de mode **`out`** peuvent être modifiés dans le corps de la procédure mais non utilisés. Les paramètres de mode **`inout`** peuvent être utilisés et modifiés.

Les fonctions ne peuvent pas modifier leurs paramètres d'appel (ils sont déclarés en mode **`in`** seulement) mais retournent une valeur. C'est une erreur de terminer l'exécution d'une fonction autrement que par la clause **`return`**.

On peut, et c'est même conseillé, les regrouper dans les paquetages. La déclaration de paquetage est le lieu des déclarations de sous-programmes (les prototypes). Le corps de paquetage est le lieu des corps de sous-programme. A défaut, on peut les placer dans la partie déclarative de l'architecture ou du processus utilisateur. Le corps de sous-programme suffit alors.

# Surcharge de nom

- **En VHDL, comme en ADA, plusieurs sous-programmes peuvent porter le même nom.**
- **Pour trouver le bon sous-programme le compilateur utilise :**
  - ***Le nom spécifié lors de l'appel.***
  - ***Le profil des paramètres d'appel :***
    - **Le nombre et le type des paramètres d'appel.**
    - **Le type du résultat (pour les fonctions).**
- **Une erreur de compilation se déclenche si après passage de ces filtres il ne reste pas un et un seul candidat. Le compilateur indique généralement tous les sous-programmes candidats qu'il a considérés.**
- **On peut également surcharger les opérateurs.**



Plusieurs fonctions (ou procédures) peuvent porter le même nom, à condition d'être différenciées par le type ou le mode de leurs paramètres ou de leur valeur de retour. C'est la surcharge.

On peut surcharger les opérateurs, qui sont des fonctions (`"+"`, `"-"`, `"and"`, etc.)

# Exemple de sous-programme

```
function NAT2VEC(VAL: NATURAL; SIZE: POSITIVE) return BIT_VECTOR;
```

```
function NAT2VEC(VAL: NATURAL; SIZE: POSITIVE) return BIT_VECTOR is
  variable RES: BIT_VECTOR(SIZE - 1 downto 0) := (others => '0');
  variable TMP: NATURAL := VAL;
begin
  for I in 0 to SIZE - 1 loop
    exit when TMP = 0;
    if (TMP mod 2 = 1) then RES(I) := '1'; end if;
    TMP := TMP / 2;
  end loop;
  assert (TMP = 0) report "NAT2VEC: debordement de capacite"
    severity WARNING;
  return RES;
end function NAT2VEC;
```

# Exercices

## Imaginez la fonction VEC2NAT

```
VEC2NAT(“001011100”) = 92
VEC2NAT(“1001001”)   = 73
```

## Ecrivez les fonctions PP et PG

```
PP(12, 18)        = 12
PP(“001”, “110”) = “001”
PG(7, 0)          = 7
PG(“001”, “110”) = “110”
```

# Corrigé

```
function VEC2NAT(VAL: BIT_VECTOR) return NATURAL;
```

```
function VEC2NAT(VAL: BIT_VECTOR) return NATURAL is
  variable TMP: BIT_VECTOR(VAL'LENGTH - 1 downto 0) := VAL;
  variable RES: NATURAL := 0;
begin
  assert (VAL'LENGTH < 32)
    report "VEC2NAT: debordement de capacite"
    severity WARNING;
  for I in VAL'LENGTH - 1 downto 0 loop
    RES := RES * 2;
    if (TMP(I) = '1') then RES := RES + 1; end if;
  end loop;
  return RES;
end function VEC2NAT;
```

# Corrigé

```
function PP(VAL1, VAL2: NATURAL) return NATURAL;
function PP(VAL1, VAL2: BIT_VECTOR) return BIT_VECTOR;
```

```
function PP(VAL1, VAL2: NATURAL) return NATURAL is
begin
  if (VAL1 < VAL2) then return VAL1 else return VAL2;
end PP;

function PP(VAL1, VAL2: BIT_VECTOR) return BIT_VECTOR is
begin
  assert ((VAL1'LENGTH = VAL2'LENGTH) and (VAL1'LENGTH < 32))
    report "PP: debordement de capacite ou utilisation illegale"
    severity WARNING;
  return NAT2VEC(PP(VEC2NAT(VAL1), VEC2NAT(VAL2)), VAL1'LENGTH);
end function PP;
```

# Corrigé

```
function PG(VAL1, VAL2: NATURAL) return NATURAL;
function PG(VAL1, VAL2: BIT_VECTOR) return BIT_VECTOR;
```

```
function PG(VAL1, VAL2: NATURAL) return NATURAL is
begin
  if (VAL1 > VAL2) then return VAL1 else return VAL2;
end PG;

function PG(VAL1, VAL2: BIT_VECTOR) return BIT_VECTOR is
begin
  assert ((VAL1'LENGTH = VAL2'LENGTH) and (VAL1'LENGTH < 32))
    report "PG: debordement de capacite ou utilisation illegale"
    severity WARNING;
  return NAT2VEC(PG(VEC2NAT(VAL1), VEC2NAT(VAL2)), VAL1'LENGTH);
end function PG;
```

# Processus combinatoires : les pièges

- **La liste de sensibilité doit être complète.**
- **Les sorties doivent recevoir une valeur dans tous les cas.**
- **Les bons synthétiseurs signalent ces défauts.**
- **Le non respect de ces règles provoquera une différence de comportement entre la simulation et le circuit synthétisé.**
- **L'apparition après synthèse d'unités de mémorisation au sein d'une partie purement combinatoire est le signe que ces règles n'ont pas été respectées.**

NON !

```
process(A, B)
begin
  if(A='1') then
    S<='1';
  elsif(B='1') then
    S<='1';
  end if;
end process;
```

OUI

```
process(A, B)
begin
  if(A='1') then
    S<='1';
  elsif(B='1') then
    S<='1';
  else
    S<='0';
  end if;
end process;
```



Il faut penser matériel avant tout, c'est le seul moyen de ne pas rencontrer de différences d 'interprétation entre concepteur, simulateur et synthétiseur.

# Pièges : le processus sans point d'arrêt

- **C'est l'erreur la plus fréquente. Un processus ne possède ni liste de sensibilité, ni instructions `wait`. Attention, il se peut que des instructions `wait` soient présentes mais rendues inopérantes par des structures de contrôle (`if`, `case`, `loop`, . . .).**
- **Effet : la simulation n'avance pas, rien ne se produit les temps symboliques et physiques restent constants. Le simulateur exécute en effet le même processus indéfiniment sans jamais donner la main à un autre.**
- **Remède : stopper la simulation, déterminer quel processus le simulateur exécutait au moment de l'arrêt et corriger le processus fautif.**



Exemples de processus sans points d'arrêt :

```
process
begin
  A <= not A after 20 ns;
end process;

process
begin
  if (A = '0') then
    wait until A = '1';
  else
    A <= '1';
  end if;
end process;
```

# Pièges : le processus cycle combinatoire

- C’est un processus équivalent à un cycle combinatoire à temps de propagation nul.
- Effet : seul le temps symbolique avance, le temps physique reste constant. Rien ne se passe.
- Remède : stopper la simulation, déterminer quel processus le simulateur exécutait au moment de l’arrêt et corriger le processus fautif.

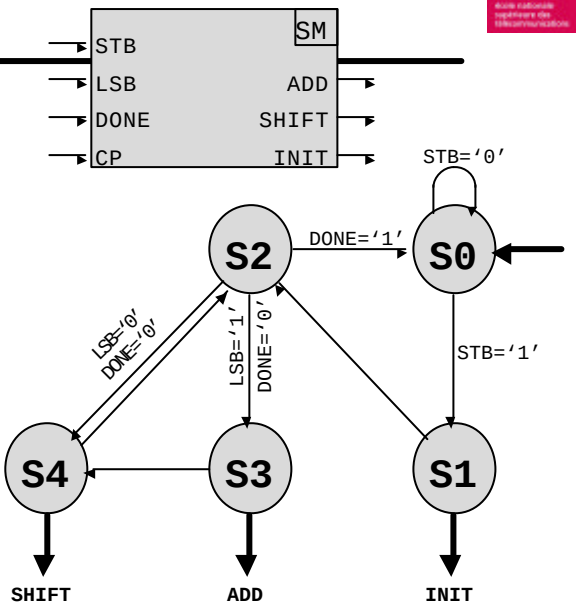


Exemple :

```
process
begin
  wait on A;
  A <= not A;
end process;
```

# Exercice

- **Voici l'interface d'une machine à états de Moore et son diagramme. La machine est synchrone sur front montant de l'horloge CP. Les entrées-sorties sont actives à '1'. Concevoir le ou les processus nécessaires à modéliser cette machine.**



# Exercice (corrigé)

```
type ST is (S0, S1, S2, S3, S4);

signal STATE, NEXT_STATE: ST;
```

```
P1: process(CP)

begin

  if (CP = '1') then
    STATE <= NEXT_STATE;
  end if;

end process P1;
```

```
P2: process(STATE)

begin

  if (STATE = S1) then
    INIT <= '1';
  else
    INIT <= '0';
  end if;
  if (STATE = S3) then
    ADD <= '1';
  else
    ADD <= '0';
  end if;
  if (STATE = S4) then
    SHIFT <= '1';
  else
    SHIFT <= '0';
  end if;

end process P2;
```

# Exercice (corrigé)

```
P3: process(STATE, STB, DONE, LSB)
begin
  case STATE is
    when S0 => if (STB = '0') then
                 NEXT_STATE <= STATE;
               else
                 NEXT_STATE <= S1;
               end if;
    when S1 => NEXT_STATE <= S2;
    when S2 => if (DONE = '1') then
                 NEXT_STATE <= S0;
               elsif (LSB = '0') then
                 NEXT_STATE <= S4;
               else
                 NEXT_STATE <= S3;
               end if;
    when S3 => NEXT_STATE <= S4;
    when S4 => NEXT_STATE <= S2;
  end case;
end process P3;
```

```
entity SM is
  port(LSB, STB, DONE,
       CP: in BIT;
       INIT, SHIFT,
       ADD: out BIT);
end entity SM;
architecture ARC of SM is
  type ST is (S0, S1, S2,
              S3, S4);
  signal STATE,
         NEXT_STATE: ST;
begin
  P1: process(CP)
  begin
  …
  end process P1;
  P2: process(STATE)
  …
  P3: …
end architecture ARC;
```

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- **Syntaxe**
  - *VHDL séquentiel*
  - ***VHDL concurrent***
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# VHDL concurrent

- **VHDL décrit des structures par assemblage d ’instructions concurrentes.**
- **Les 5 instructions concurrentes, sont :**
  - ***Les processus***
  - ***Les instanciations de composants***
  - ***Les appels de procédures concurrentes***
  - ***Les assertions concurrentes***
  - ***Les affectations concurrentes de signaux***
- **En mode concurrent, les instructions de VHDL s'exécutent en pseudo parallélisme, l'ordre dans lequel elles sont écrites n'a aucune importance sur le calcul du résultat.**

# VHDL concurrent

- **Un assemblage d 'instructions concurrentes constitue une architecture.**
- **Un processus est UNE instruction concurrente.**

```
architecture ARC of FOO is
  signal I0, I1: INTEGER;
begin

  P1: process(A, B)
  begin
    I0 <= A+B;
    I1 <= A*B;
  end process P1;

  P2: process(I0, I1)
  begin
    if(I0 /= 0) then
      S <= I1/I0;
      ALARM <= FALSE;
    else
      S <= 0;
      ALARM <= TRUE;
    end if;
  end process P2;

end architecture ARC;
```



L'assemblage des instructions concurrentes permet d'écrire une architecture (comme l'assemblage d'instructions séquentielles permet d'écrire un processus). L'architecture est le lieu de la description concurrente.

# VHDL concurrent

- **Dans un soucis de souplesse et de simplicité d'utilisation, un certain nombre de formes alternatives du processus ont été définies, ce sont**
  - ***les affectations concurrentes de signaux***
  - ***les assertions concurrentes***
  - ***les appels concurrents de procédures***
- **Ces instructions concurrentes ressemblent à des instructions séquentielles mais n'en sont pas.**
- **Il est important de se souvenir qu'elles sont en réalité des processus déguisés.**



Certaines instructions du domaine séquentiel ont un «équivalent » concurrent. Il ne faut cependant pas les confondre : les fonctionnements induits sont différents.

# Raccourcis d'écriture

```
architecture AR of SOMME is
begin
  PR: process(A, B, CI)
  begin
    S <= A xor B xor CI;
  end process PR;
end architecture AR;
```

```
architecture AR of SOMME is
begin
  S <= A xor B xor CI;
end architecture AR;
```



Les affectations concurrentes de signal, dont on voit ici la forme la plus simple, ne sont que des raccourcis d'écriture. Il existe un processus équivalent dont elles sont une forme simplifiée. La liste de sensibilité de ce processus contient l'ensemble des signaux apparaissant dans l'expression à droite du symbole <=.

# Raccourcis d'écriture

```
architecture AR of SOMME is

begin

  PR: process(A, B, CI)

  begin

    if (A = '0') then
      S <= B xor CI;
    elsif (B = '0') then
      S <= not CI;
    else
      S <= CI;
    end if;

  end process PR;

end architecture AR;
```

```
architecture AR of SOMME is

begin

  S <= B xor CI when (A = '0') else
       not CI when (B = '0') else
       CI;

end architecture AR;
```



Cette affectation concurrente de signal, plus élaborée que la précédente, est également la version simplifiée d'un processus équivalent utilisant une clause **if**.

# Raccourcis d'écriture

```
architecture AR of SOMME is

begin

  PR: process(ETAT)

  begin

    case ETAT is
      when INIT =>
        ETAT_SUIVANT <= RUN;
      when RUN =>
        ETAT_SUIVANT <= WAIT;
      when WAIT =>
        ETAT_SUIVANT <= INIT;
      when others =>
        ETAT_SUIVANT <= INIT;
    end case;

  end process PR;

end architecture AR;
```

```
architecture AR of SOMME is

begin

  with ETAT select
    ETAT_SUIVANT <= RUN when INIT,
                    WAIT when RUN,
                    INIT when WAIT,
                    INIT when others;

end architecture AR;
```



Cette affectation concurrente de signal est encore la version simplifiée d'un processus équivalent utilisant cette fois une clause **case**.

# Assertion concurrente

- **L'assertion concurrente est équivalente à un processus contenant une assertion séquentielle où** `LISTE_DE_SIGNAUX` **serait la liste des signaux apparaissant dans** `CONDITION`**.**

```
ETIQUETTE: assert CONDITION
  report "Message"
  severity NIVEAU;
```

```
ETIQUETTE: processus
begin
  assert CONDITION
    report "Message"
    severity NIVEAU;
  wait on LISTE_DE_SIGNAUX;
end process ETIQUETTE;
```



Si aucun signal n'apparaît dans la condition, l'**assert** ne sera vérifié que lors de l'élaboration :

```
assert NOW > 10 sec
  report "Fin de simulation"
  severity FAILURE;
```

# Procédures concurrentes

- **Il est possible d'utiliser un appel de procédure concurrent. Cette construction peut, par exemple, servir à surveiller des signaux. Elle est équivalente à un processus où `LISTE_DE_SIGNAUX` serait la liste des signaux de `LISTE_DE_PARAMETRES` qui ont pour mode `in` ou `inout`.**

```
ETIQUETTE: NOM_DE_PROCEDURE(LISTE_DE_PARAMETRES);
```

```
ETIQUETTE: processus
begin
  NOM_DE_PROCEDURE(LISTE_DE_PARAMETRES);
  wait on LISTE_DE_SIGNAUX;
end processus ETIQUETTE;
```



Attention : les variables locales de la procédure sont élaborées à chaque nouvelle activation du processus.

Attention : si la procédure n'a pas de signaux en mode **in** ou **inout** dans sa liste de paramètres, le processus ne sera exécuté qu'une seule fois (lors de la phase d'élaboration).

# VHDL structurel

- **VHDL permet de décrire des assemblages de composants, en respectant le principe de décomposition hiérarchique. Pour pouvoir séparer la structure du graphe de décomposition des composants réellement utilisés, deux mécanismes sont utilisés :**
  - ***Lors de l'assemblage, le compilateur vérifie la compatibilité de la déclaration de l'enveloppe du composant (directive** component**) avec les signaux qui lui sont effectivement connectés.***
  - ***Dans la configuration du circuit, on peut préciser explicitement les versions de circuit à utiliser.***
- **Tout ceci peut conduire à des descriptions un peu lourdes textuellement, mais très faciles à générer et à exploiter automatiquement.**



La description structurelle passe nécessairement par l'intermédiaire du composant configuré. C'est incontournable !! La redondance qui en découle n'est bien souvent qu'apparente ; elle est de plus largement compensée par les avantages que procure ce mécanisme.

# La déclaration de composant

- **Le mot clef** component **sert à déclarer le prototype d'interconnexion d'une entité externe dans une architecture. La syntaxe est la même que la syntaxe de l'entité :**

```
component NOM_DE_COMPOSANT
  port(DECLARATION_DE_PORTS);
end component;
```

- **Une déclaration de composant peut apparaître**
  - ***Dans la partie déclarations d'une architecture***
  - ***Dans un paquetage (réutilisable)***
- **Exemple :**

```
component AND2
  port(A, B: in STD_LOGIC; C: out STD_LOGIC);
end component;
```



La déclaration de composant n ’est pas une unité de compilation. Son principal intérêt est de permettre la compilation alors que les couples entité / architecture correspondants ne sont pas encore décrits.

# L’instanciation de composant

- **L'instanciation d'un composant se fait dans le corps d'une architecture au moyen de l'instruction suivante :**

```
ETIQUETTE: NOM_DE_COMPOSANT
  port map(LISTE_DE_PORTS);
```

- **Où `ETIQUETTE` est le label de l'instance.**
- **La clause `port map` décrit l’association des ports d'entrée sortie formels et actuels.**
- **La syntaxe des associations peut être :**
  - ***Soit positionnelle (une simple liste d'identificateurs qui seront associés aux identificateurs formels de même rang).***
  - ***Soit par nom ; chaque association est alors de la forme.***

```
IDENTIFICATEUR_FORMEL => IDENTIFICATEUR_ACTUEL
```

  - ***Comme pour les agrégats, on peut mélanger les deux syntaxes.***



L ’instanciation de composant est une instruction concurrente au même titre que le processus.

# L'instanciation de composant : exemple

```
entity AND2 is
  port(A, B: in BIT;
       C: out BIT);
end entity AND2;

architecture ARC of AND2 is
begin
  C <= A and B;
end architecture ARC;

configuration AND2_CFG of AND2 is
  for ARC
  end for;
end configuration AND2_CFG;
```

```
entity AND3 is
  port(I1, I2, I3: in BIT;
       S: out BIT);
end entity AND3;
```

```
architecture STR of AND3 is
  component AND2
    port(A, B: in BIT; C: out BIT);
  end component;
  signal TMP: BIT;
begin
  A0: AND2 port map(A => I1, B => I2,
                    C => TMP);
  A1: AND2 port map(A => I3, B => TMP,
                    C => S);
end architecture STR;

configuration AND3_CFG of AND3 is
  for STR
    for all: AND2
      use configuration WORK.AND2_CFG;
    end for;
  end for;
end configuration AND3_CFG;
```

# Règles de correspondance

- **Quand on configure un composant VHDL exige de connaître la correspondance exacte entres les ports formels du composant et ceux de l'entité équivalente.**
- **Si l'entité associée n'a pas exactement la même interface déclarée (nom de ports), la clause de configuration doit comporter une clause `port map`**

```
for all: NAND2 use entity WORK.AND2(BEHAVE)
  port map(I1 => A, I2 => B, S => C);
```



Le mécanisme de configuration peut être très simple lorsque la déclaration de composant est en tous points semblable à la déclaration de l'entité associée. Elle peut aussi être beaucoup plus complexe (parce que plus riche et plus puissante) lorsque ce n'est pas le cas.

# Généricité

- **VHDL propose la notion de généricité**
- **Un paramètre générique est considéré comme une constante dans l'architecture.**

```
entity ADD is
  generic(N: POSITIVE range 1 to 32 := 8);
  port(A, B: in  BIT_VECTOR(N - 1 downto 0);
       CI:   in  BIT;
       S:    out BIT_VECTOR(N - 1 downto 0);
       CO:   out BIT);
end entity ADD;
```



La généricité est le mécanisme qui permet de créer des composants avec variante. C'est le B. A. BA de la réutilisabilité.

# Généricité

- **Les paramètres génériques peuvent posséder une valeur par défaut.**
- **La valeur des paramètres génériques peut être fixée :**
  - ***Par la clause d'instanciation du composant (`generic map`)***
  - ***Par les valeurs par défaut de la déclaration de composant***
  - ***Par les valeurs par défaut de l'entité associée***



La valeur des paramètres génériques peut être fixée de diverses façons :

1) si l'entité possède un paramètre générique mais pas la déclaration de composant associée :
- si la configuration du composant comporte une clause `generic map` c'est cette clause qui fixe la valeur ;
- sinon c'est la valeur par défaut définie par l'entité ;
- si l'entité ne définit pas de valeur par défaut une erreur d'élaboration se produira ;

2) si le composant définit également un paramètre générique ;
- la configuration du composant doit préciser l'association entre paramètres génériques de l'entité et du composant (sauf si l'association est évidente) ;
- la valeur est fixée par la clause `generic map` de l'instanciation ;
- si l'instanciation ne comporte pas de clause `generic map` la valeur est fixée par la valeur par défaut de la déclaration du composant ;
- si la déclaration du composant ne définit pas de valeur par défaut une erreur de compilation se produira.

# Généricité, exemple d'utilisation

```
architecture ARC of MUL is
  component ADD
    generic(N: POSITIVE range 1 to 32 := 8);
    port(A, B: in  BIT_VECTOR(N - 1 downto 0);
         S:    out BIT_VECTOR(N - 1 downto 0));
  end component;
  signal X1, X2, S: BIT_VECTOR(16 downto 0);
. . .
begin
. . .
  I_ADD: ADD generic map(N => 17);
             port map(A => X0, B => X1, S => Z);
. . .
end architecture ARC;
```

# Generate

- **La clause generate augmente la portée de la généricité en proposant des structures de contrôle concurrentes.**

```
architecture RTL of ADD is
  component ADD1
    port(A, B, CI: in  BIT;
         S, CO:    out BIT);
  end component;
  signal C: BIT_VECTOR(N downto 0);
begin
  G: for I in 0 to N - 1 generate
    IA: ADD1 port map(A(I), B(I), C(I), S(I), C(I + 1));
  end generate G;
  C(0) <= CI;
  CO <= C(N);
end architecture RTL;
```



Il existe aussi un **`if ... generate`**.

La configuration doit faire apparaître le bloc **`generate`** :

```
configuration ADD_RTL_CFG of ADD is
  for RTL
    for G
      for all: ADD1
        use configuration BIB.ADD1_RTL_CFG;
      end for;
    end for;
  end for;
end configuration ADD_RTL_CFG;
```

et non pas :

```
configuration ADD_RTL_CFG of ADD is
  for RTL
    for all: ADD1
      use configuration BIB.ADD1_RTL_CFG;
    end for;
  end for;
end configuration ADD_RTL_CFG;
```

# Generate

```
entity FA
  port(X, Y, Z: in BIT;
       I, J:    out BIT);
end entity FA;

architecture BEV of FA is
begin
  I <= X xor Y xor Z;
  J <= (X and (Y or Z)) or
       (Y and Z);
end architecture BEV;
```

```
configuration CFG of ADD is
  for RTL
    for G
      for IA: ADD1
        use entity BIB.FA(BEV);
        port map(X => A, Y => B,
                 Z => CI, I => S,
                 J => CO);
      end for;
    end for;
  end for;
end configuration CFG;
```



Attention : les bornes de boucle d'un `for generate` ainsi que les conditions des `if generate` doivent être des expressions statiques calculables à l'élaboration. Pendant l'exécution elles sont constantes. Toute autre situation n'aurait pas grand sens du point de vue matériel puisque cela consisterait à manipuler du matériel dynamique, c'est à dire des transistors apparaissant et disparaissant au fur et à mesure des besoins.

# Les fonctions de résolution

- **Lorsqu'un signal est piloté par plusieurs processus, il possède autant d'échéanciers que de processus pilotes. Pour déterminer quelle est sa valeur il est nécessaire de lui associer une fonction dite de "résolution". Cette fonction est associée au type du signal qui est dit "résolu". Exemple :**

```
function OU_CABLE(VAL: BIT_VECTOR)
  return BIT is
begin
  if (VAL'LENGTH = 0) then
    return '0';
  end if;
  for I in VAL'RANGE loop
    if (VAL(I) = '1') then
      return '1';
    end if;
  end loop;
  return '0';
end function OU_CABLE;
```

```
subtype RESOLVED_BIT is
  OU_CABLE BIT;

signal S1: RESOLVED_BIT;
signal S2: OU_CABLE BIT;
```

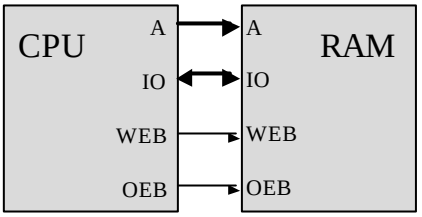


Il faut limiter l'usage des types résolus aux seuls cas de pilotes multiples. En effet, non seulement les fonctions de résolution sont coûteuses en temps de calcul, mais en outre une utilisation abusive empêche le compilateur de détecter certaines erreurs de programmation.

La bibliothèque `IEEE` contient un paquetage `STD_LOGIC_1164` qui définit une logique multivaluée à 9 valeurs :

```
type STD_ULOGIC is ('U',  -- Uninitialized
                    'X',  -- Forcing  Unknown
                    '0',  -- Forcing  0
                    '1',  -- Forcing  1
                    'Z',  -- High Impedance
                    'W',  -- Weak     Unknown
                    'L',  -- Weak     0
                    'H',  -- Weak     1
                    '-'   -- Don't care);
type STD_ULOGIC_VECTOR is (NATURAL range <>) of
  STD_ULOGIC;
function RESOLVED (S: STD_ULOGIC_VECTOR) return
  STD_ULOGIC;
subtype STD_LOGIC is RESOLVED STD_ULOGIC;
...
```

# Exemple d'utilisation des fonctions de résolution




```
entity CPU is
  port(A: out STD_ULOGIC_VECTOR(7 downto 0);
       IO: inout STD_LOGIC_VECTOR(15 downto 0);
       WEB, OEB: out STD_ULOGIC);
end entity CPU;
architecture ARC of CPU is
. . .
  IO <= "010011000110111"; -- ecriture
. . .
  IO <= "ZZZZZZZZZZZZZZZZ"; -- lecture
. . .
end architecture ARC;
```



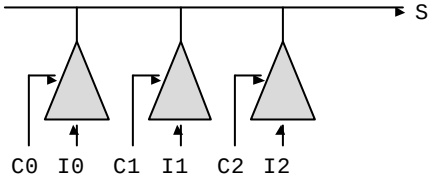
La mémoire simple port est un exemple classique d'utilisation des types résolus. En effet, dans la situation décrite ci-dessus, lorsque le CPU écrit dans la RAM c'est lui qui impose une valeur au bus de données IO. Lorsqu'il lit dans la RAM c'est elle qui impose sa valeur à ce même bus. Si CPU et RAM sont décrits dans un seul et même processus on utilise pour le signal IO un type non résolu car il est piloté par un seul processus. Il possède alors un seul échéancier, géré de façon classique.

Si, comme c'est le plus souvent le cas, le CPU et la RAM sont décrits par des processus différents le signal IO possède plusieurs émetteurs. Il est alors indispensable d'utiliser un type résolu car seule la fonction de résolution associée sera capable de déterminer la valeur résultant du conflit entre les deux échéanciers.

Dans notre exemple nous avons utilisé le type STD_LOGIC_VECTOR déclaré dans le paquetage STD_LOGIC_1164 de la bibliothèque IEEE. Pour écrire le CPU dépose la valeur à écrire sur le bus tandis que la RAM met le bus en haute impédance en utilisant la valeur 'Z'. Réciproquement, pour lire le CPU met le bus en haute impédance et la RAM dépose la valeur lue sur le bus.

La fonction de résolution associée au type STD_LOGIC donne la priorité aux valeurs « fortes » sur les valeurs « faibles ». Un conflit entre 'Z' d'une part et '0' ou '1' d'autre part sera résolu en '0' ou '1'.

# Exemple d'utilisation des fonctions de résolution




```
signal S: STD_LOGIC;
. . .
S <= I0 when C0 = '1' else
     'Z' when others;
. . .
S <= I1 when C1 = '1' else
     'Z' when others;
. . .
S <= I2 when C2 = '1' else
     'Z' when others;
. . .
```



La logique multiplexée à buffers trois états est un autre exemple classique d'utilisation des types résolus.

# Types résolus : les pièges

- **Il ne faut jamais utiliser un signal de type résolu inutilement.**
- **Le compilateur et l'élaborateur ne peuvent détecter les conflits involontaires que sur les types non résolus.**
- **Utiliser systématiquement les types résolus (`STD_LOGIC` au lieu de `STD_ULOGIC`) c'est se priver d'une sécurité bien utile.**
- **Utiliser systématiquement les types résolus (`STD_LOGIC` au lieu de `STD_ULOGIC`) c'est ralentir inutilement les simulations.**

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- Syntaxe
  - *VHDL séquentiel*
  - *VHDL concurrent*
- **Les paquetages normalisés**
- **La synthèse**
- **Conseils**

# La bibliothèque STD, le paquetage STANDARD

- **La bibliothèque STD est fournie en standard avec tout environnement de développement VHDL.**
- **Il n 'est pas nécessaire de déclarer la bibliothèque STD dans un programme VHDL car elle est l'est déjà, de façon implicite.**
- **Le paquetage STANDARD définit les types de base :**
  - ***Enumérés :*** `BOOLEAN, BIT, CHARACTER, SEVERITY_LEVEL`
  - ***Numériques :*** `INTEGER, NATURAL, POSITIVE, REAL, TIME`
  - ***Composites :*** `STRING, BIT_VECTOR`
- **ainsi que la fonction NOW.**
- **Il n'est pas nécessaire de déclarer le paquetage STANDARD dans un programme VHDL car il l'est déjà implicitement.**



Le paquetage STANDARD est indispensable et c'est pourquoi il est implicitement déclaré dans tout programme VHDL. Les deux clauses :

```
library STD;
use STD.STANDARD.all;
```

sont donc optionnelles.

Le paquetage STANDARD n'est, en général, pas compilable.

# La bibliothèque STD, le paquetage TEXTIO

- **C'est le paquetage dédié aux entrées-sorties sur fichier de texte.**
- **Il est malheureusement très imparfait**
- **Le paquetage TEXTIO définit :**
  - ***Les types*** LINE, TEXT, SIDE ***et*** WIDTH
  - ***Les fichiers*** INPUT ***et*** OUTPUT
  - ***Les procédures*** READLINE, READ, WRITELINE ***et*** WRITE
  - ***La fonction*** ENDLINE
- **Il est nécessaire de déclarer le paquetage dans un programme VHDL car il ne l'est pas implicitement :**

```
use STD.TEXTIO.all;
```



Le type LINE est un type pointeur sur chaînes de caractères. Le type TEXT est un type fichier de chaînes de caractères. Les types SIDE et WIDTH servent à formater les données.

Les différentes procédures READ et WRITE sont définies pour les types du paquetage STANDARD. Les procédures READ possèdent deux versions : une version simple et une version avec un paramètre de contrôle booléen permettant de vérifier si la lecture s'est bien passée.

Le fichier INPUT correspond à l'entrée standard (clavier) et le fichier OUTPUT à la sortie standard (écran).

Lors de l'utilisation de la procédure WRITE avec un paramètre littéral de type STRING ou BIT_VECTOR il est nécessaire de qualifier le paramètre afin d'aider le compilateur à lever l'ambiguïté :

`WRITE(LIGNE, "01001 ");` ne fonctionnera pas. Il faut écrire :

```
WRITE(LIGNE, BIT_VECTOR' ("01001"));
```

Remarque : il en va de même lorsque l'ambiguïté semble pourtant évidente à lever :

```
WRITE(LIGNE, STRING' ("JKHDL"));
```

# La bibliothèque IEEE, le paquetage

STD_LOGIC_1164

- **Ce paquetage propose une logique multivaluée sous la forme d'un type énuméré**

```
type STD_ULOGIC is ('U',  -- Uninitialized
                    'X',  -- Forcing Unknown
                    '0',  -- Forcing  0
                    '1',  -- Forcing  1
                    'Z',  -- High Impedance
                    'W',  -- Weak Unknown
                    'L',  -- Weak 0
                    'H',  -- Weak 1
                    '-'   -- Don't care);
```

- **Il fournit également une fonction de résolution pour** STD_ULOGIC **et un type résolu associé :** STD_LOGIC
- **Il fournit également les types vecteur** STD_ULOGIC_VECTOR **et** STD_LOGIC_VECTOR**.**



Attention à l'utilisation des types STD_ULOGIC et STD_LOGIC. Ils ne sont pas interchangeables. L'un est résolu, l'autre non (d'ailleurs le U de STD_ULOGIC signifie Unresolved ; et non pas Unsigned comme on le dit parfois). En conséquence un signal piloté par plusieurs processus ne pourra pas être de type STD_ULOGIC, une erreur d'élaboration se produirait. Utiliser systématiquement le type STD_LOGIC pour éviter ces erreurs est une très mauvaise idée, malheureusement répandue. Ce faisant on se prive en effet d'une vérification automatique bien utile. Si le concepteur a, par erreur, connecté entre elles les sorties de plusieurs processus et s'il a pris la précaution de réserver le type STD_LOGIC aux seules situations de logique à conflit, un message de l'élaborateur le préviendra du problème. Sinon … il perdra du temps sur les outils de déboggage.

Une autre mauvaise idée répandue consiste à déclarer systématiquement signaux et variables de ces types avec une valeur d'initialisation. Ce faisant on évite souvent quelques messages d'alerte en début de simulation alors que les données ne sont pas encore initialisées. Mais on se prive alors de la détection des défauts d'initialisation. En effet, en l'absence de valeur d'initialisation une variable (ou un signal) prend au début de la simulation la valeur de gauche dans la déclaration de son type. Ici 'U' pour « Uninitialized ». Si le concepteur a mal conçu son mécanisme de « reset » il constatera rapidement la propagation de valeurs indéterminées … sauf s'il a défini des valeurs d'initialisation comme '0' ou '1'.

# La bibliothèque IEEE, le paquetage
STD_LOGIC_1164

## Le paquetage définit également :

- *un certain nombre de sous-types résolus de* STD_ULOGIC *(*X01*,* X01Z*, etc.)*
- *l'ensemble des surcharges des opérateurs logiques pour* STD_ULOGIC, STD_LOGIC, STD_ULOGIC_VECTOR *et* STD_LOGIC_VECTOR
- *les fonctions de conversion entre tous ces types et sous-types et avec les types* BIT *et* BIT_VECTOR
- *les fonctions* RISING_EDGE *et* FALLING_EDGE *de détection de fronts sur un signal de type* STD_ULOGIC
- *les fonctions* IS_X *de détection de valeurs inconnues (*'U', 'X' , 'Z' , 'W' , '-'*).*

# La bibliothèque IEEE, autres paquetages

- IEEE contient d'autres paquetages.
- Mentionnons en particulier ceux dédiés à l'arithmétique sur les vecteurs :
  - NUMERIC_BIT :
    - définit les types SIGNED et UNSIGNED (tableaux de BIT)
    - surcharge les opérateurs arithmétiques, logiques et relationnels pour ces types
    - définit des fonctions de conversion avec les types entiers
    - propose des fonctions diverses (rotations, décalages, etc.)
  - NUMERIC_STD
    - définit les types SIGNED et UNSIGNED (tableaux de STD_LOGIC)
    - surcharge les opérateurs arithmétiques , logiques et relationnels pour ces types
    - définit des fonctions de conversion avec les types entiers
    - propose des fonctions diverses (rotations, décalages, etc.)



Les paquetages NUMERIC_BIT et NUMERIC_STD permettent de décrire des traitements arithmétiques directement sur des types vecteur. Ils sont donc très utiles.

Attention, avant la normalisation de ces paquetages les vendeurs de CAO (Synopsys ®, Cadence ®, Compas ®, Mentor Graphics ®, etc.) ont proposé des paquetages alternatifs destinés au même usage. Ils ont souvent eu le tort de les placer dans la bibliothèque IEEE sans que cette noble institution n'ait donné son aval. On trouve ainsi fréquemment des paquetages nommés STD_LOGIC_ARITH, STD_LOGIC_SIGNED, STD_LOGIC_UNSIGNED, etc. dans les environnements fournis par ces vendeurs. Lorsque cela est possible il est préférable d'utiliser les paquetages normalisés. Ils sont plutôt mieux conçus et surtout, ils rendent le code portable, ce qui n'est pas le cas des autres dont il existe de multiples versions, plus ou moins cohérentes entre elles.

On peut remarquer que les types SIGNED et UNSIGNED du paquetage NUMERIC_STD sont résolus. C'est regrettable pour les mêmes raisons que celles évoquées plus haut. Il convient donc de redoubler de prudence lorsque l'on utilise ces types. Les conflits accidentels ne sont pas dénoncés par l'élaborateur.

# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- Syntaxe
  - *VHDL séquentiel*
  - *VHDL concurrent*
- Les paquetages normalisés
- **La synthèse**
- **Conseils**

# La synthèse

- **Inférence de logique combinatoire pour les variables ou signaux**
  - ***affectés sans condition***
  - ***avant d'être utilisés***
  - ***à chaque modification des signaux dont dépend l'expression***
- **Ou encore**
  - ***conditionnellement affectés***
  - ***pour toute condition possible***
  - ***à chaque modification des signaux dont dépend l'expression***

```
process (a, b, c)
begin
  z <= a+b+c;
end process;
```

```
process (a, b, c)
begin
  if b = '1' then
    z <= a;
  else
    z <= c;
  end if;
end process;
```

# La synthèse

## Inférence de *latches* pour les variables ou signaux

- ***incomplètement affectés***
- ***sous contrôle d'un signal sur niveau***
- ***avec ou sans initialisation***
  - **synchrone**
  - **asynchrone**

```
process(data_in, enable)
begin
  if enable = '1' then
    data_out <= data_in;
  end if;
end process;
```

```
process(data_in, enable,
        set_sig, reset_sig)
begin
  if set_sig = '1' then
    data_out <= '1';
  elsif reset_sig = '1' then
    data_out <= '0';
  elsif enable = '1' then
    data_out <= data_in;
  end if;
end process;
```

# La synthèse

## Inférence de bascules D pour les variables ou signaux

- *incomplètement affectés*
- *sous contrôle d'un signal sur front*
- *avec ou sans initialisation*
  - **synchrone**
  - **asynchrone**

```
process(clk)
begin
  if clk = '1' and
     clk'event then
    data_out <= data_in;
  end if;
end process;
```

```
process(clk)
begin
  if clk = '1' and
     clk'event then
    if set_sig = '1' then
      data_out <= '1';
    elsif reset_sig = '1' then
      data_out <= '0';
    else
      data_out <= data_in;
    end if;
  end if;
end process;
```

# La synthèse

- **Inférence de bascules D pour les variables ou signaux**
  - ***incomplètement affectés***
  - ***sous contrôle d'un signal sur front***
  - ***avec ou sans initialisation***
    - **synchrone**
    - **asynchrone**

```
process(clk, set_sig, reset_sig)
begin
  if set_sig = '1' then
    data_out <= '1';
  elsif reset_sig = '1' then
    data_out <= '0';
  elsif clk = '1' and clk'event then
    data_out <= data_in;
  end if;
end process;
```

# La synthèse

## Spécifications de fronts d'horloge :

```
if (clk'event and clk = '1') then
wait until (clk'event and clk = '1');
if (rising_edge(clk)) then
wait until rising_edge(clk);
if (clk'event and clk =  '0') then
wait until (clk'event and clk =  '0');
if (falling_edge(clk)) then
wait until falling_edge(clk);
...
```

# La synthèse

## L ’inférence accidentelle de latches

```
signal curr_state, next_state, modifier:
    std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
  case curr_state is
    when "000"  => next_state <= "100" or  modifier;
    when "001"  => next_state <= "110" or  modifier;
    when "010"  => next_state <= "001" and modifier;
    when "100"  => next_state <= "101" and modifier;
    when "101"  => next_state <= "010" or  modifier;
    when "110"  => next_state <= "000" and modifier;
    when others => null;
  end case;
end process;
```

# La synthèse

## Eviter l'inférence accidentelle de latches

```
signal curr_state, next_state, modifier:
    std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
  next_state <= "100";
  case curr_state is
    when "000"  => next_state <= "100" or  modifier;
    when "001"  => next_state <= "110" or  modifier;
    when "010"  => next_state <= "001" and modifier;
    when "100"  => next_state <= "101" and modifier;
    when "101"  => next_state <= "010" or  modifier;
    when "110"  => next_state <= "000" and modifier;
    when others => null;
  end case;
end process;
```

# La synthèse

## Eviter l'inférence accidentelle de latches

```
signal curr_state, next_state, modifier:
    std_logic_vector(2 downto 0);
process(curr_state, modifier)
begin
  case curr_state is
    when "000"  => next_state <= "100" or  modifier;
    when "001"  => next_state <= "110" or  modifier;
    when "010"  => next_state <= "001" and modifier;
    when "100"  => next_state <= "101" and modifier;
    when "101"  => next_state <= "010" or  modifier;
    when "110"  => next_state <= "000" and modifier;
    when others => next_state <= "100";
  end case;
end process;
```

# La synthèse

## Les boucles

- *on peut les utiliser*
- *mais elles sont déroulées*
- *les bornes des boucles* `for` *doivent être statiques :*
  - **on peut écrire :** `for I in 0 to 7 loop`
  - **mais pas :** `for I in F(X) to G(Y) loop` **(sauf si** `X` **et** `Y` **sont des constantes)**
- *les conditions des boucles* `while` *doivent être statiques*
  - **on peut écrire :** `while FALSE loop`
  - **mais pas :** `while C(X, Y) loop` **(sauf si** `X` **et** `Y` **sont des constantes)**
  - **les boucles** `while` **et inconditionnelles sont rarement synthétisables**

# La synthèse

- **Les clauses `wait` sont parfois acceptées mais avec de fortes restrictions ; exemple :**
  - ***une seule clause `wait` par processus***
  - ***forcément en première instruction***
  - ***uniquement pour la spécification de fronts d'horloge***
  - ***avec l'horloge comme seul signal de la clause de sensibilité***
  - **...**

# La synthèse

## Les directives de synthèse

- *soit commentaires interprétés*
- *soit attributs VHDL*
- *usages multiples*
  - `synthesis on/off`
  - `translate on/off`
  - **set et reset**
  - **architectures arithmétiques**
  - **codage de types énumérés**
  - **logique multiplexée ou câblée (pour les `case`)**
  - **codage et optimisations de machines à états**
  - **interprétation de fonctions de résolution**
  - **...**

# La synthèse

## Il existe des paquetages dédiés à la synthèse

- ***Normalisés :***
  - `IEEE.STD_LOGIC_1164`
  - `IEEE.NUMERIC_BIT`
  - `IEEE.NUMERIC_STD`
- ***Propriétaires :***
  - **déclarations des attributs - directives de synthèse**
  - **arithmétiques propriétaires**
  - **modèles VHDL de macro-focntions**
  - **modèles VHDL de cellules de bibliothèques**
  - **...**

# La synthèse

- **Arithmétique `BIT` (ou `STD_ULOGIC`) normalisée**
  - ***on utilise les types `SIGNED` et `UNSIGNED`***
  - ***les opérateurs arithmétiques classiques sont surchargés pour les types `SIGNED` et `UNSIGNED`***
  - ***on peut mélanger `SIGNED` et entiers ou `UNSIGNED` et entiers dans les expressions arithmétiques***
  - ***il existe des fonctions de conversion entiers <-> vecteurs***
    - `TO_INTEGER`
    - `TO_SIGNED, TO_UNSIGNED`
  - ***les types vecteurs sont compatibles entre eux ; on les convertit avec les fonctions de conversion auto-déclarées :***
    - `SIGNED`
    - `UNSIGNED`
    - `BIT_VECTOR`

# La synthèse

## Attention :

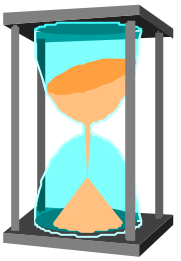
- *aux registres involontaires*
  - *flip-flops*
  - *latches*
- *aux listes de sensibilité incomplètes*
- *aux boucles*
- *aux cycles combinatoires*
- *au signe lors des opérations arithmétiques*
- *au partitionnement*
- *à la portabilité*
  - **l'interprétation au sens de la synthèse n'est pas normalisée**
  - **les paquetages propriétaires sont ... propriétaires**

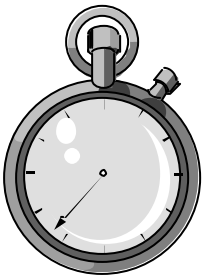
# Plan

- Introduction
- Principes de la simulation évènementielle
- Organisation informatique des projets
- Les unités de compilation
- Syntaxe
  - *VHDL séquentiel*
  - *VHDL concurrent*
- Les paquetages normalisés
- La synthèse
- **Conseils**

# Conseils : synchroniser

```
...
wait for 10 * PERIODE;
...
```



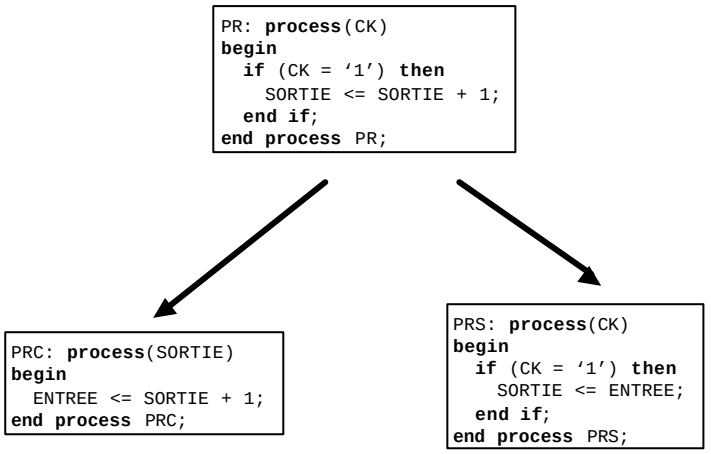


```
...
for I in 0 to 9 loop
  wait until (CK = '1');
end loop;
...
```



Les deux styles d'écriture proposés ne sont pas du tout équivalents. C'est une conséquence des principes sous-jacents de la simulation événementielle. Bien qu'elle soit probablement plus lente en simulation on préférera toujours la synchronisation sur signaux à la synchronisation sur durée. Cette politique évite généralement bien des déboires.

# Conseils : pour la synthèse séparer synchrone et combinatoire






Une telle stratégie augmentera en général la lisibilité du code et donc sa maintenabilité. La recherche des éventuels "bugs" en sera également simplifiée.

# Conseils : pour la simulation économiser les signaux

```
architecture ARC of REGS is
  signal A0, A1: BIT;
begin
  REGS_PR: process(CK)
  begin
    if (CK = '1') then
      A0 <= DIN;
      A1 <= A0;
      DOUT <= A1;
    end if;
  end process REGS_PR;
end ARC;
```

```
architecture ARC of REGS is
begin
  REGS_PR: process(CK)
    variable A0, A1: BIT;
  begin
    if (CK = '1') then
      DOUT <= A1;
      A1 := A0;
      A0 := DIN;
    end if;
  end process REGS_PR;
end ARC;
```



Cette recommandation ne vaut que pour la simulation. En effet, pour le simulateur, les signaux sont plus complexes à gérer que les variables. Ils nécessitent l'allocation d'une plus grande quantité de mémoire. Quand on a le choix entre l'utilisation de signaux ou de variables le choix des variables se traduira donc par de meilleures performances en simulation.

Par contre, l'utilisation des variables est parfois plus risquée (voir l'exemple ci-dessus). De plus, les performances en temps CPU du synthétiseur ne sont pas sensibles au choix variable-signal.

# Conseils : pour la synthèse compter les registres

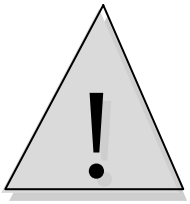
```
architecture ARC of REGS is

begin

  REGS_PR: process(CK) -- 3 registres
    variable A0, A1: BIT;
  begin
    if (CK = '1') then
      DOUT <= A1;
      A1 := A0;
      A0 := DIN;
    end if;
  end process REGS_PR;

end ARC;
```



Chaque fois que vous écrivez un processus destiné à être traduit sous forme de bascules D, ajoutez un commentaire précisant le nombre de bascules attendues et vérifiez lors de la synthèse que le résultat est conforme à vos espérances.

# Conseils : commenter et rester méfiant

- **Une ligne de code pour 10 lignes de commentaires**
- **HDL /= matériel**





Sans commentaire … et souvenez vous toujours que le modèle n'est pas l'objet modélisé.

# Conseils : la maîtrise de la complexité

- Il est fréquent qu'un problème simple en apparence soit dans les faits bien plus complexe que prévu. Dans une telle situation le concepteur ajoute progressivement des traitements particuliers à son code pour prendre en compte les difficultés nouvelles.
- Effet : si les difficultés sont grandes le code devient rapidement très volumineux, inutilisable, impossible à maintenir.
- Remède : repenser la question en tenant compte des problèmes découverts. Repenser le partitionnement et les structures de données. Reprendre entièrement l'écriture du code.

# Conseils : l'informatique et le matériel

- Le langage VHDL est un langage de programmation. Mais sa destination première est la description de matériel. Un concepteur qui n'a pas une idée claire de la forme matérielle qu'il décrit ne produira rien de bon.
- Effets : impossibilité de raffiner le code en VHDL synthétisable, différences de comportement constatées entre la simulation avant synthèse et la simulation après synthèse.
- Remède : penser le matériel d'abord. Pour décrire une architecture matérielle il faut avant tout en avoir une idée précise et claire.

# Validation



# Plan

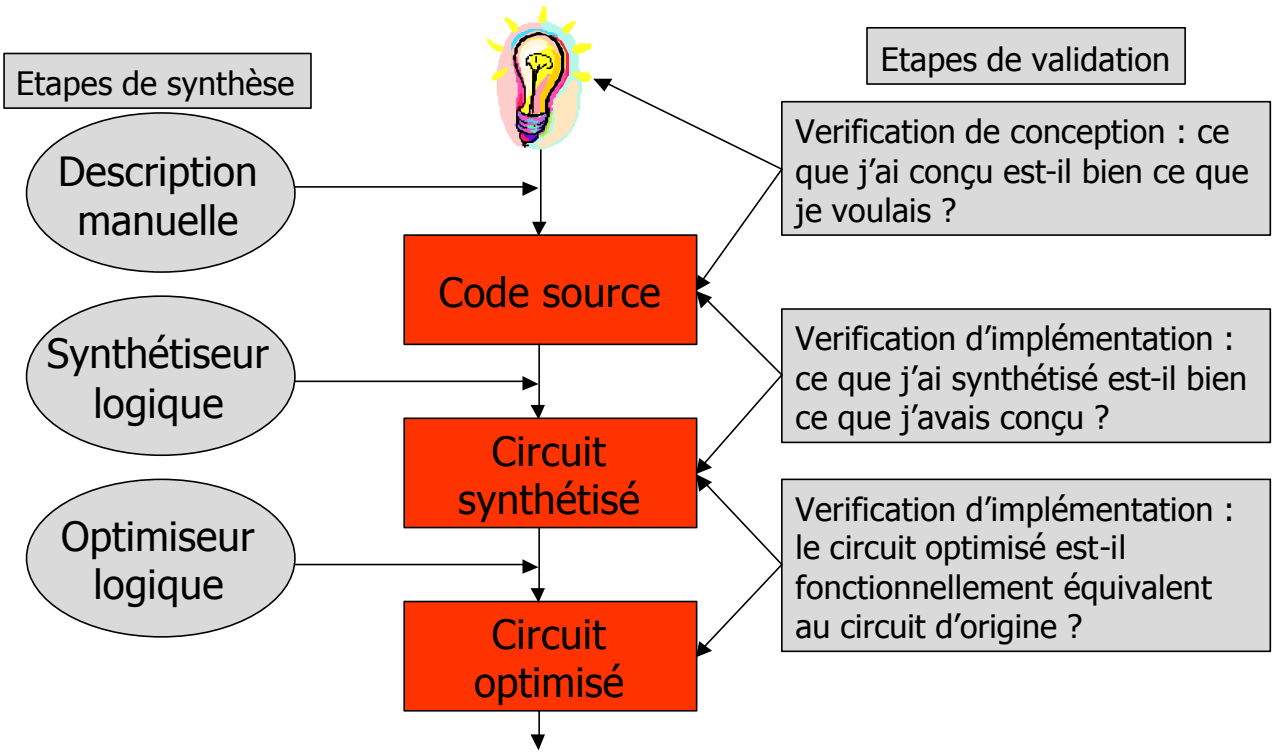
- Introduction
- La simulation
- La vérification formelle
- L'accélération matérielle

# Plan

- Introduction
- La simulation
- La vérification formelle
- L'accélération matérielle



# Valider



# Les bugs du Pentium 4

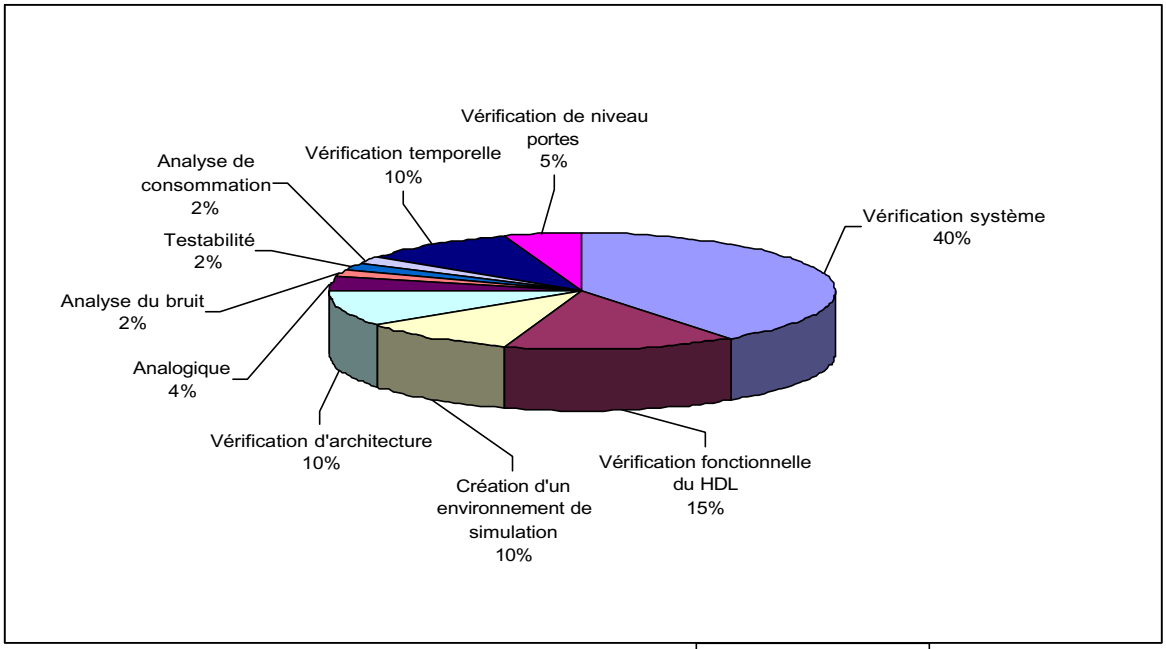
- Source: EE Times, July 4, 2001
- 42 millions de transistors
- Plus d'un million de lignes de code RTL
- 100 bugs logiques de haut niveau trouvés par vérification formelle

| Types de bugs | % |
|---|---|
| **Stupide** | **12,7** |
| **Problème de communication** | **11,4** |
| **Micro-architecture** | **9,3** |
| **Modifications de logique ou de micro-code** | **9,3** |
| **Cas aux limites** | **8** |
| **Powerdown (clock gating)** | **5,7** |
| **Documentation** | **4,4** |
| **Complexité** | **3,9** |
| **Initialisation** | **3,4** |
| **Définitions tardives** | **2,8** |
| **Assertions RTL fausses** | **2,8** |
| **Erreurs de conception** | **2,6** |



# L'effort de vérification

- Plus de 50% de l'effort total




Source: 1999 ITRS

# L'effort de vérification

- **Chez Motorola quand décide-t-on de partir en fonderie ?**
  - ***On a simulé 40 milliards de cycles sans trouver de bug***
  - ***On a terminé les tests directs du plan de validation***
  - ***Les objectifs de couverture du code et de couverture fonctionnelle sont atteints***
  - ***Le taux de découverte de bugs a significativement baissé***
  - ***La date prévue est atteinte***

**(Source: EE Times, July 4, 2001)**



# Plan

- **Introduction**
- **La simulation**
- **La vérification formelle**
- **L'accélération matérielle**

# La simulation fonctionnelle

- **Permet de valider la fonction**
  - ***De la totalité du modèle (exhaustif)***
  - ***Sur une séquence particulière (non exhaustif)***
- **A besoin d'un environnement**
  - ***Stimuli d'entrée***
  - ***Résultats attendus (modèle de référence)***



# La simulation fonctionnelle

- **Permet de trouver vite de nombreuses erreurs « simples »**
- **N'est pas adaptée à la recherche des erreurs « complexes » (cas aux limite)**
- **Peut être précise au cycle près et au bit près (*Cycle Accurate*, *Bit Accurate* ou *CABA*)**
  - ***Seules les caractéristiques physiques sont ignorées***
- **Ou non**
  - ***Modèle algorithmique***
- **C'est la seule qui autorise des simulations intensives du circuit complet. Les modèles doivent être optimisés en vitesse de simulation**

# La simulation fonctionnelle

- **Pour être rapide un modèle doit :**
  - ***Utiliser le bon langage***
  - ***Economiser les évènements, donc les signaux***
  - ***Economiser la mémoire, donc les signaux***
  - ***Souvent s'éloigner de l'architecture, même pour un modèle* CABA**
  - ***On mesure la performance en cycles simulés par seconde de CPU. Exemple IDCT 8x8* CABA *sur Ultra-5 :***

| Langage | Cycles / s |
|---|---|
| **SystemC** | **141000** |
| **Verilog** | **113000** |
| **VHDL** | **33500** |



# La simulation fonctionnelle

- **Attention, vitesse et puissance d'analyse sont ennemis**
  - ***Vérifications dynamiques (VHDL)***
  - ***Débordements de capacité***
- **Attention, vitesse et puissance d'expression sont ennemis**
  - ***Parallélisme***
  - ***Ordonnancement séquentiel***
  - ***Variables versus signaux***

# La génération de test, la qualimètrie

- Certains outils proposent la génération automatique d'environnements de simulation
  - *Verisity (Specman Elite)*
  - *Synopsys (VERA)*
- Ils permettent (théoriquement) d'atteindre des taux de couverture plus élevés (code + fonctionnel) avec des efforts moindres
- Ils exploitent une description d'environnement (langage dédié)
- Ils sont basés sur des solveurs de contrainte



# Types de modèles fonctionnels *CABA*

- L'émulateur
  - *Une coquille vide qui utilise des fichiers de données*
  - *Vitesse = 32x*
- Le modèle non synthétisable optimisé en langage
  - *SystemC ou C/C++ encapsulé par les API VHDL ou Verilog*
  - *Vitesse = 16x*
- Le modèle non synthétisable VHDL ou Verilog
  - *Vitesse = 8x*
- Le modèle synthétisable
  - *Vitesse = 4x*
- Le modèle synthétisé
  - *Vitesse = 1x*

# La simulation cycle

- Sur un modèle à un seul signal : l'horloge
- Demande un effort particulier de conception (remplacer tous les signaux par des variables : ordonnancer)
- Peut s'automatiser (s'apparente alors à la synthèse)
- Permet de gagner en vitesse de simulation (facteur 10 à 100)

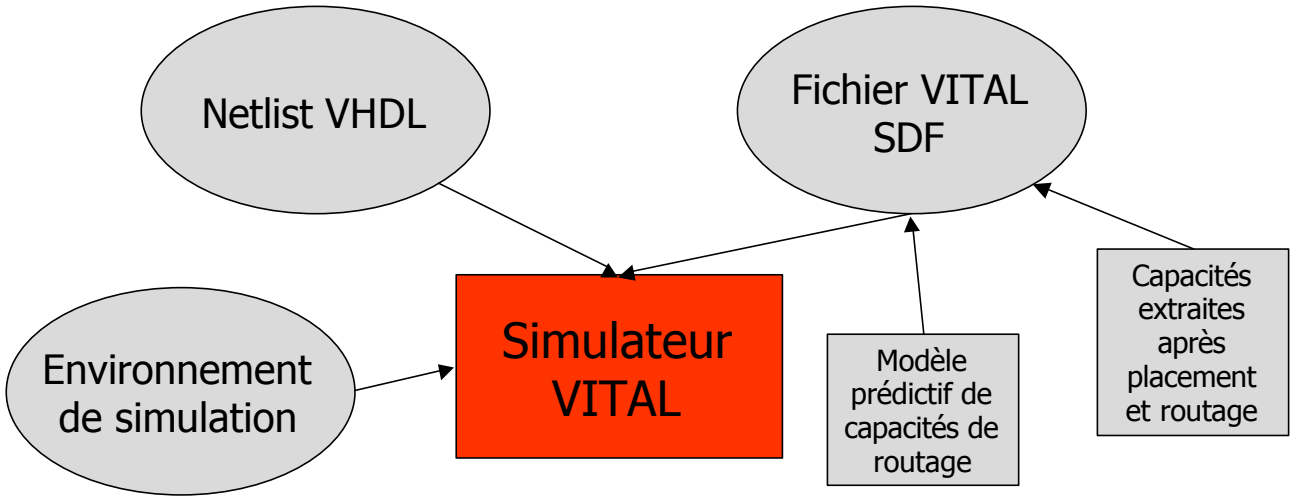


# La simulation en portes

- On peut simuler une description sous forme de réseau de portes logiques
- La simulation en portes intègre ou non un modèle temporel
- Il existe de nombreux modèles temporels
  - *« Prop Ramp Delay »*
  - *« Input Slope Model »*
  - *« Wave tabular »*

# La simulation en portes

- La norme VITAL (IEEE 1076.4-1995) permet d'annoter un réseau de portes :






# Le modèle « Prop Ramp Delay »

- Utilisé pour des estimations grossières
- Ou pour des technologies anciennes (à peu près abandonné vers 0,5 µ)

$$tpA\text{->}Z_{lh} = tpA\text{->}Z_{lh0} + \Delta tpA\text{->}Z_{lh0} + \Delta tpA\text{->}Z_{lh} * C_{ld}$$

- Temps de propagation pour une transition montante de la sortie Z due à une transition sur l'entrée A
- Temps de propagation à vide (intrinsèque, sans capacité de sortie)
- Temps de propagation dû à la capacité de sortie (= $\Delta tpA\text{->}Z_{lh} * C_s$)
- Temps de propagation dû à la capacité de charge $C_{ld}$

# Les autres modèles de délai

- « Input Slope Model »
  - *Tient compte de la pente du signal d'entrée*
  - *Donne des résultats proches du « Prop Ramp Delay » pour les fortes pentes*
  - *Plus complexe, donc plus lent*
- « Wave tabular »
  - *Tient compte de la forme d'onde des entrées*
  - *Approximation linéaire par morceaux*
  - *Encore plus complexe*



# La simulation électrique

- Pour avoir des résultats encore plus précis il faut simuler au niveau transistor (SPICE, ELDO)
- Utilisée pour la caractérisation des cellules de bibliothèque ou pour des parties « full custom »
- Limitée à quelques centaines de transistors

# Plan

- Introduction
- La simulation
- La vérification formelle
- L'accélération matérielle



# La vérification formelle

- Simulation versus vérification formelle
- Preuve d'équivalence combinatoire
- Preuve d'équivalence séquentielle
- Preuve de modèle

# La simulation

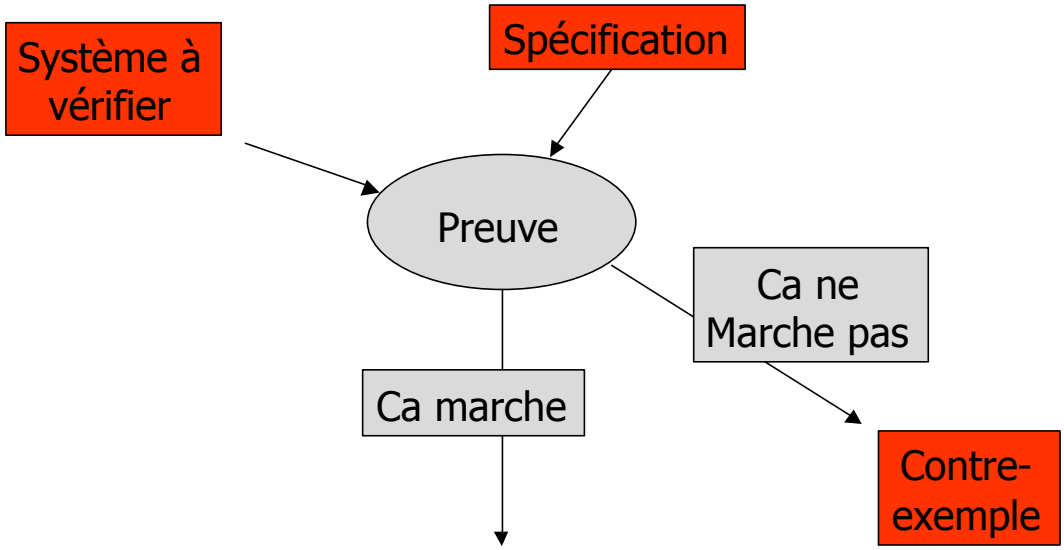
- La vérification par simulation
  - *On produit des stimuli d'entrée*
    - Aléatoires
    - Spécifiques à une fonctionnalité particulière
  - *On produit les résultats attendus*
  - *On simule avec les stimuli d'entrée*
  - *On compare les sorties avec les résultats attendus*



# Les problèmes de la simulation

- Il faut simuler des milliards de cycles
- La simulation n'est pas exhaustive
  - *Pas de garantie pour les séquences non simulées*
- Pour générer les résultats attendus il faut un autre modèle
- Les stimuli d'entrée efficaces sont difficiles à obtenir
  - *Ils doivent avoir un taux de couverture élevé*
  - *Cas aux limites*
  - *Les erreurs sont souvent là où les concepteurs ont été le moins attentif*

# Qu'est-ce que la vérification formelle ?






# La vérification formelle

- **Exhaustive (pour une certaine propriété)**
  - ***Le résultat est garanti mathématiquement***
- **Pas besoin de produire les résultats attendus**
- **Peut produire un contre exemple en cas d'échec**
- **Une puissante tapette à bugs**

# La vérification formelle, histoire

- La théorie est déjà ancienne
  - *En 1980 on pouvait vérifier des systèmes à $10^6$ états*
- En 1990 Clarke et al. (CMU) font des progrès considérables avec SMV
  - *On peut vérifier des systèmes à $10^{20}$ états*
- Des bugs « durs » ont été découverts dans des systèmes réels
  - *Protocole de cohérence de cache, …*
- Aujourd'hui
  - *Il existe des outils commerciaux*
  - *Les industriels s'y intéressent*
  - *On peut vérifier des systèmes à $10^{100}$ états*



# Simulation versus vérification formelle

- Simulation
  - *Non exhaustive*
  - *Génération des résultats attendus*
  - *Difficulté à couvrir les cas aux limites*
- Vérification formelle
  - *Exhaustive (pour la spécification)*
  - *Génération des résultats attendus inutile*
  - *Cas aux limites couverts automatiquement*

# Simulation versus vérification formelle

- Simulation
  - *Temps CPU élevé (milliards de cycles)*
  - *Peut s'appliquer à des gros systèmes*
- Vérification formelle
  - *Très gourmande en mémoir*
  - *Structures de données internes (BDD)*
  - *Taille mémoire liée à taille des systèmes à vérifier*



# Simulation versus vérification formelle

- La simulation est toujours un moyen efficace et rapide de vérifier tôt
- La vérification formelle ajoute de la confiance
- Les deux techniques sont complémentaires

# Preuve d'équivalence combinatoire

- On compare mathématiquement deux circuits
- Ne s'applique qu'à des vues
  - *Synthétisables ou synthétisées*
  - *Dont les éléments de mémorisation sont identiques (presque)*
- Rapide (plusieurs générations par jour pour des millions de portes)
- Exhaustif
- Peu cher (outils et prise en main)
- Suppose l'existence d'un modèle de référence



# Preuve d'équivalence combinatoire

- Uniquement fonctionnel (caractéristiques physiques ignorées)
- Deux circuits sont identiques s'ils ont les mêmes éléments de mémorisation et des fonctions combinatoires équivalentes
- Permet
  - *De valider rapidement des modifications mineures de code synthétisable*
  - *De valider des modifications de « netlist » (synthèse, optimisation, test, amplification d'horloge, placement – routage)*

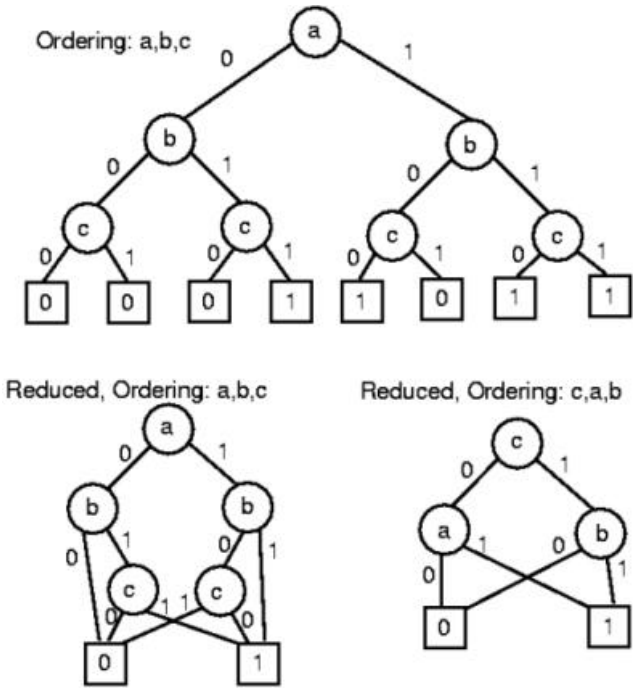
# Preuve d'équivalence combinatoire

- **Le problème de la comparaison booléenne est NP-complet**
  - ***Equivalent à un parcours d'arbre de profondeur polynomiale, nombre de branches exponentiel***
  - ***La recherche de solution est exponentielle (pour l'instant, cf conjecture P = NP)***
  - ***La vérification de solution est polynomiale***
- **La représentation des fonctions est critique en mémoire et en temps**



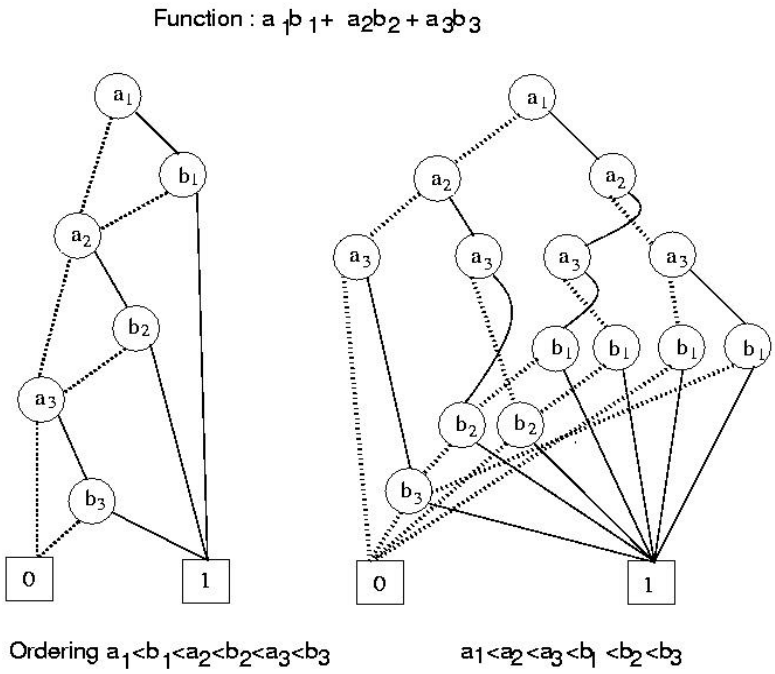
# Preuve d'équivalence combinatoire

- **Les BDD (Binary Decision Diagrams) représentent des fonctions booléennes (ensembles d'états, transitions d'états, etc.)**



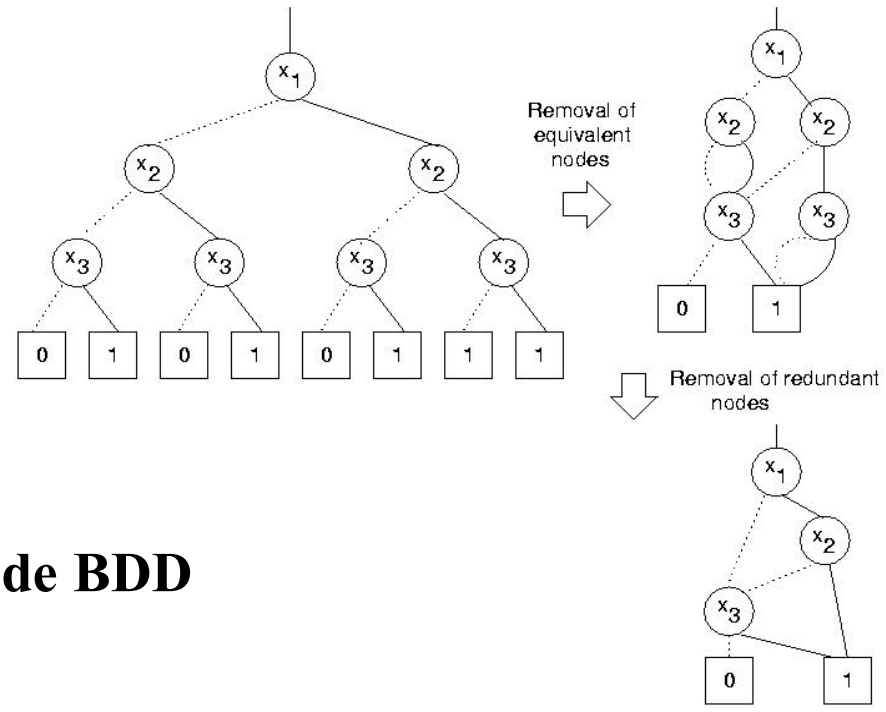
# Preuve d'équivalence combinatoire

- **L'ordre des variables influe sur la taille des BDD**





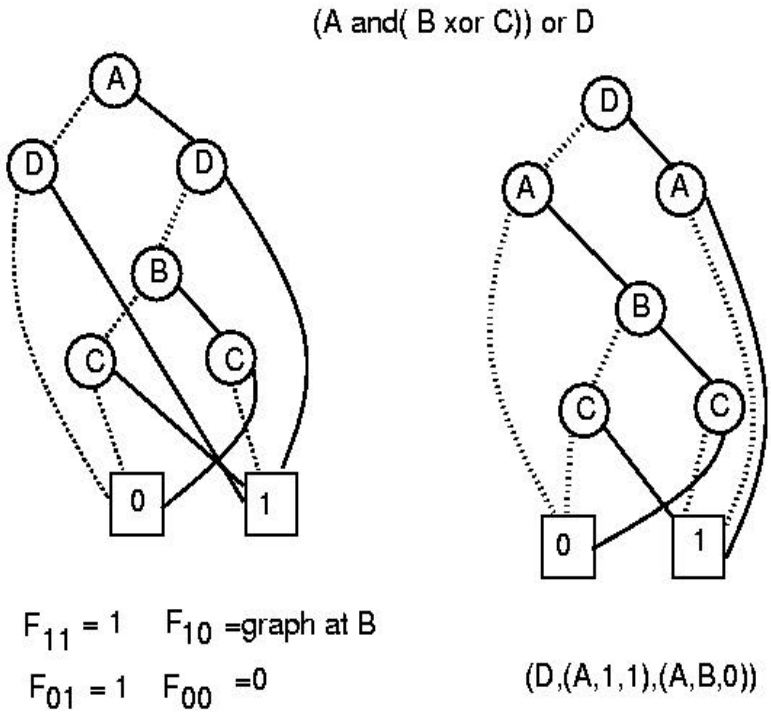

# Preuve d'équivalence combinatoire




- **Réduction de BDD**

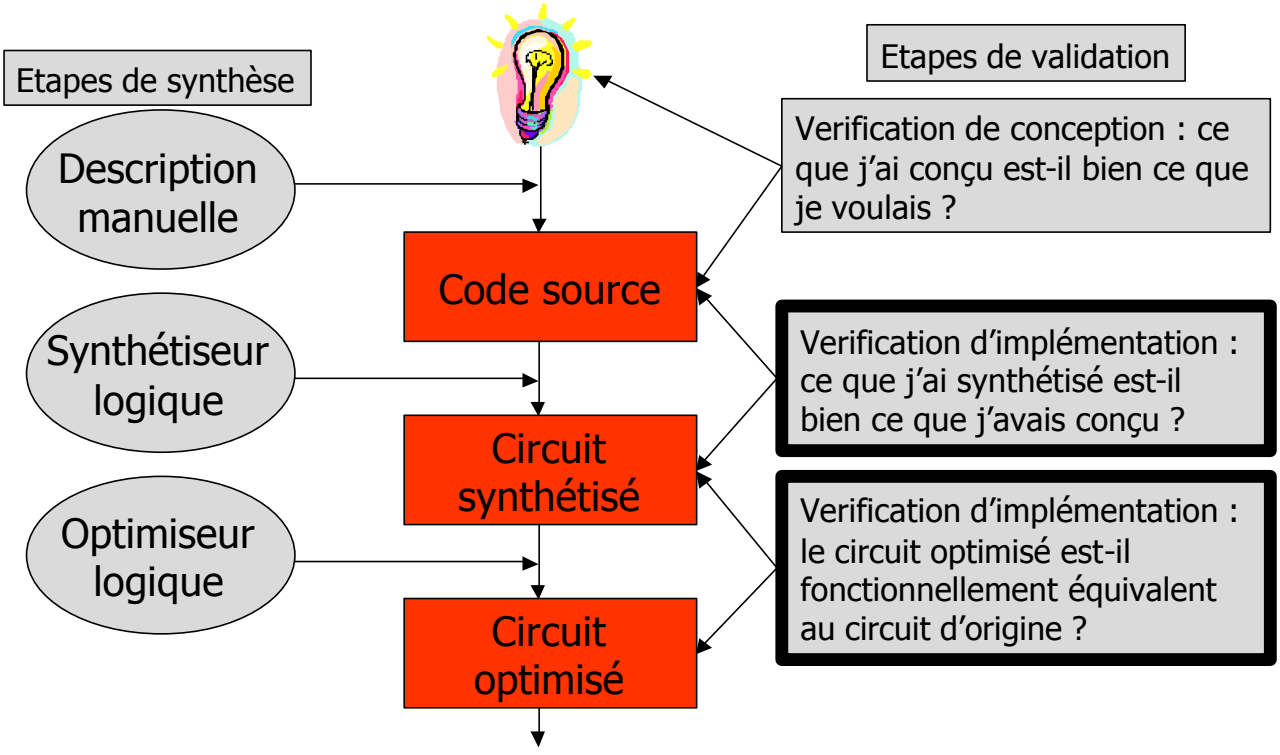
# Preuve d'équivalence combinatoire

- **Réordonner les variables**






# Preuve d'équivalence combinatoire



# Preuve d'équivalence combinatoire

- Les outils commerciaux :
  - *Verplex (Tuxedo)*
  - *Mentor Graphics (FormalPro)*
  - *Synopsys (Formality)*
  - *Chrysalis -> Avant! -> Synopsys (DesignVerifyer)*



# Preuve d'équivalence séquentielle

- On compare mathématiquement deux circuits
- S'applique à des vues
  - *Synthétisables ou synthétisées*
  - *Sans contrainte sur les éléments de mémorisation*
- Lent et gourmand en mémoire (quelques centaines de bascules)
- Exhaustif
- Peu cher (outils et prise en main)
- Suppose l'existence d'un modèle de référence

# Preuve d'équivalence séquentielle

- Uniquement fonctionnel (caractéristiques physiques ignorées)
- Deux circuits sont identiques s'ils ont le même comportement aux interfaces (au cycle près et au bit près)
- Permet
  - *De valider des modifications majeures de code synthétisable*
- Pas (peu ?) d'outils commerciaux



# Preuve de modèle (« model checking »)

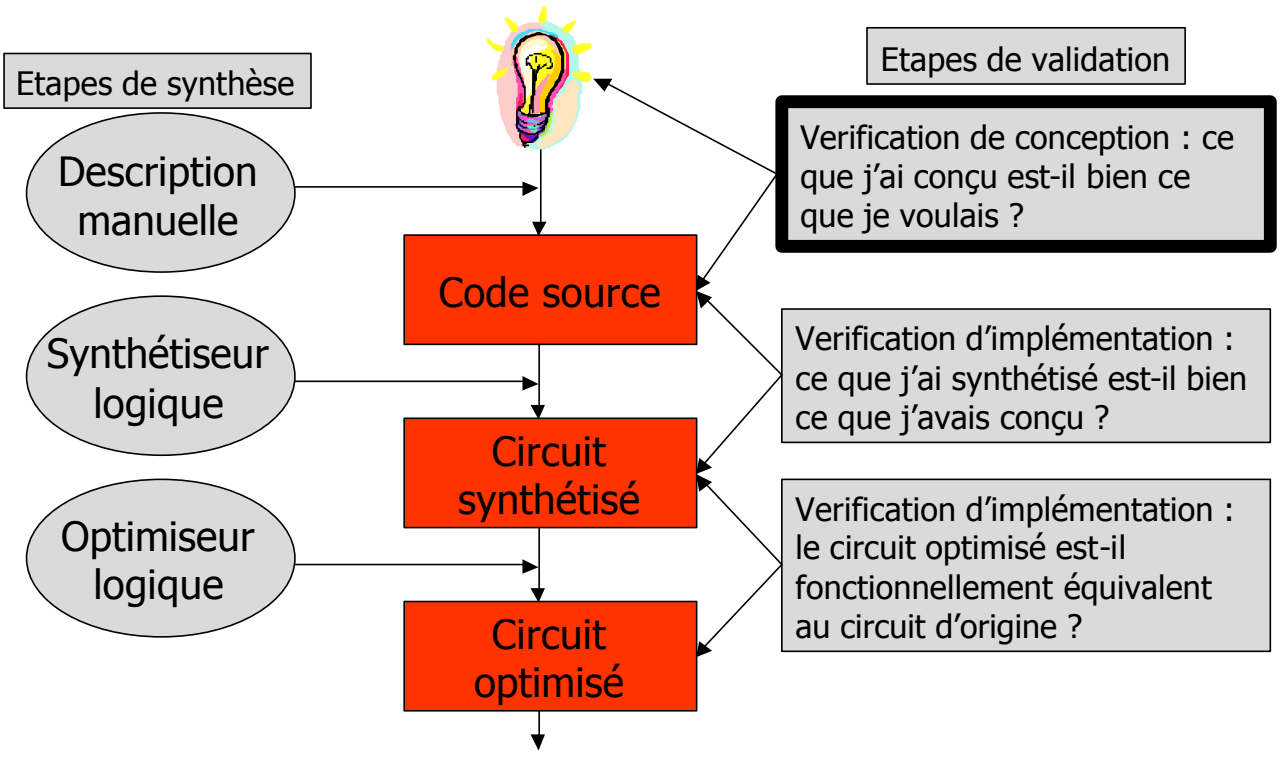
- On compare mathématiquement un circuit avec des propriétés logico-temporelles
- Ne s'applique qu'à des vues synthétisables ou synthétisées
- Exhaustif
- Pas donné (surtout prise en main)
- Adapté aux parties de contrôle (automates) mais pas aux parties de calcul (explosion combinatoire)

# Preuve de modèle (« model checking »)

- Uniquement fonctionnel (caractéristiques physiques ignorées)
- Un circuit vérifie une propriété s'il n'existe aucun contre-exemple
- Permet
  - *De valider le modèle synthétisable*
  - *De valider des modifications*



# Preuve de modèle (« model checking »)



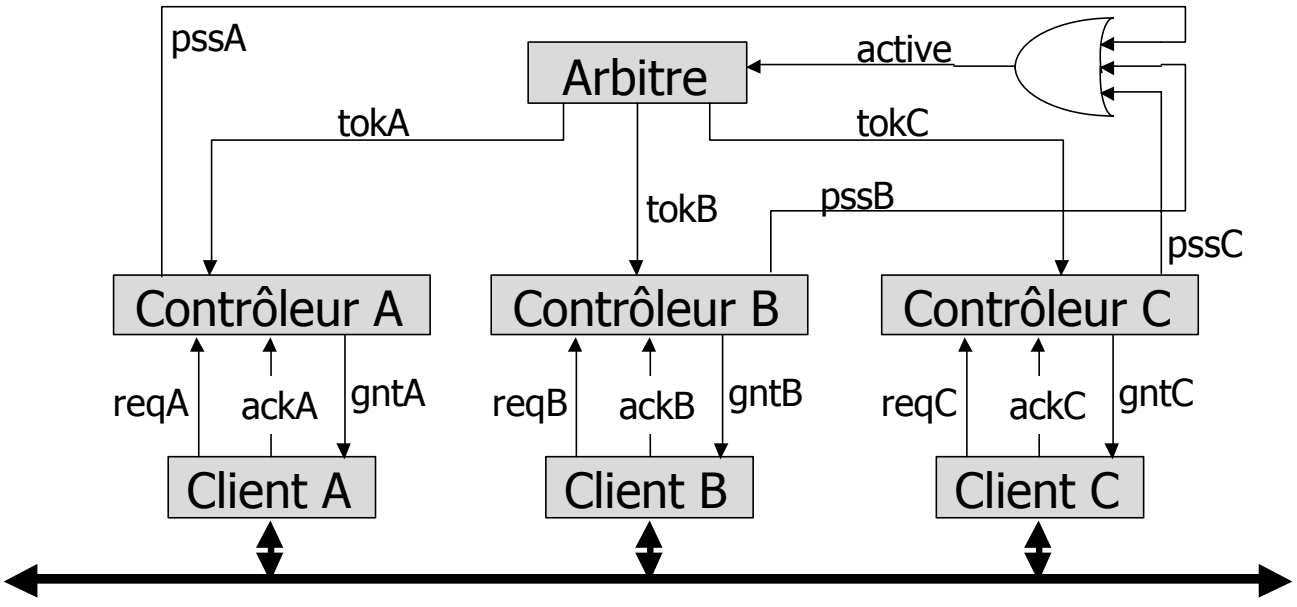
# Preuve de modèle (« model checking »)

- Les outils commerciaux :
  - *IBM (RuleBase)*
  - *Cadence (FormalCheck)*
  - *TINA-Valyosys (ImproveHDL)*

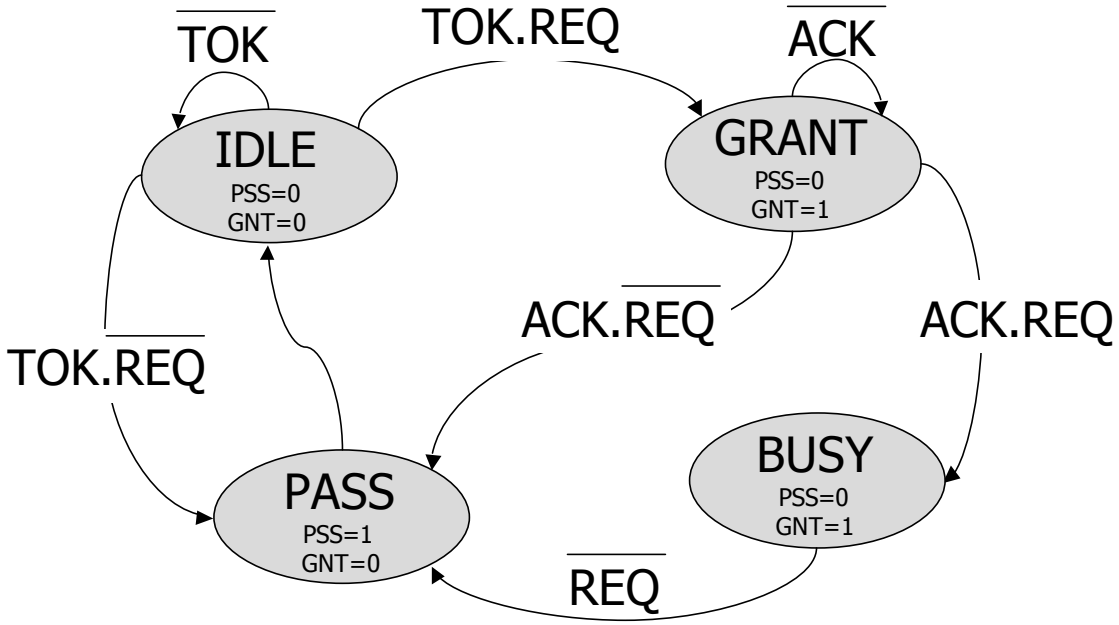


# Model checking

- Notre exemple : contrôle de bus



# Le contrôleur






# « Model checking » du contrôleur

- **Pour vérifier la fonctionnalité du modèle du contrôleur par « model checking » if faut :**
  - ***Le modèle HDL du contrôleur***
  - ***La spécification du contrôleur sous forme de propriétés logico-temporelles***
  - ***Une description de l'environnement du contrôleur (arbitre et clients)***
  - ***Un « model checker »***
  - ***De la mémoire, du temps, du café***

# Le modèle HDL du contrôleur

```
entity CTRL is
  port(CLK, RSTN, REQ,
       ACK, TOK: in BOOLEAN;
       PSS, GNT: out BOOLEAN);
end CTRL;

architecture BEH of CTRL is

  type STATE_TYPE is (IDLE, GRANT, BUSY, PASS);
  signal STATE: STATE_TYPE;

begin

  process(CLK, RSTN)
  begin
    if (not RSTN) then
      STATE <= IDLE;
    elsif RISING_EDGE(CLK) then
      case STATE is
        when IDLE =>
          if (TOK and REQ) then
            STATE <= GRANT;
          elsif (TOK and not REQ) then
            STATE <= PASS;
          end if;
…
```

```
…
        when GRANT =>
          if (ACK and REQ) then
            STATE <= BUSY;
          elsif (ACK and not REQ) then
            STATE <= PASS;
          end if;
        when BUSY =>
          if (not REQ) then
            STATE <= PASS;
          end if;
        when PASS => STATE <= IDLE;
      end case;
    end if;
  end process;

  GNT <= STATE = GRANT or STATE = BUSY;
  PSS <= STATE = PASS;

end BEH;
```



# La spécification du contrôleur

- On spécifie le contrôleur en utilisant une logique temporelle
  - *CTL (Computation Tree Logic)*
  - *PLTL (Propositional Linear Temporal Logic)*
  - *CTL* (CTL + PLTL)*
- Le choix d'une logique temporelle fixe la puissance d'expression
  - *Les logiques temporelles ne sont pas équivalentes*

# Les logiques temporelles

- Utilisent les combinateurs booléens classiques
  - *true, false,* $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- Utilisent des propositions atomiques
  - P = $ACK \wedge \neg REQ$
- Utilisent des combinateurs temporels qui permettent de caractériser une exécution (une branche de l'arbre des états)
  - *X (neXt) : X* P ≡ *l'état suivant vérifie* P
    - STATE = PASS ⇒ X STATE = IDLE
  - *F (Futur) : F* P ≡ *un état futur vérifie* P
    - REQ ⇒ F ACK



# Les logiques temporelles

- Combinateurs temporels
  - *G (Global) : G* P ≡ *tous les états futurs vérifient* P
    - G ¬ (gntA ∧ gntB)
  - *G est le dual de F :* $G\,\Phi \equiv \neg F \neg \Phi$ *(*$\Phi$ *est une formule)*
    - ¬ F (gntA ∧ gntB)
  - *U (Until) :* P *U* Q ≡
    - *Q* sera vérifiée un jour
    - En attendant, *P* reste vraie
    - REQ ⇒ X (REQ U GNT)
  - *W (Weak until) :* $\Phi_1\ W\ \Phi_2 \equiv (\Phi_1\ U\ \Phi_2) \vee G\ \Phi_1 \equiv$
    - *P* reste vraie jusqu'à ce que *Q* soit vraie
    - Ou, si *Q* n'est jamais vraie, *P* reste vraie toujours
    - REQ ⇒ X (REQ W GNT)

# Les logiques temporelles

- **On peut combiner les combinateurs**
  - *$GF \equiv F^{\infty}$ : $GF\ \Phi \equiv$ le long de l'exécution $\Phi$ sera vraie une infinité de fois*
  - *$FG \equiv G^{\infty}$ : $FG\ \Phi \equiv$ le long de l'exécution $\Phi$ sera vraie tout le temps à partir d'un certain moment (ou $\Phi$ sera fausse un nombre fini de fois)*
- **Les quantificateurs de chemins**
  - *Expriment le comportement arborescent (plusieurs futurs possibles à partir d'une situation donnée)*
  - *$A\ \Phi$ : toutes les exécutions partant de l'état courant vérifient la formule $\Phi$*



# Les logiques temporelles

- **Quantificateurs de chemins**
  - *$E\ \Phi$ : à partir de l'état courant il existe une exécution qui vérifie la formule $\Phi$*
  - *Attention, $A \neq G$.*
    - A signifie « toutes les exécutions »
    - G signifie « tous les états » le long de l'exécution considérée
    - A et E quantifient sur les chemins
    - G et F quantifient sur les positions le long d'un chemin donné
- **Quantificateurs de chemins et combinateurs temporels vont souvent par paires :**
  - *EF* **P** *: il existe une exécution telle que* **P** *(un jour)*
  - *AF* **P** *: quelle que soit l'exécution,* **P** *(un jour)*

# Les logiques temporelles

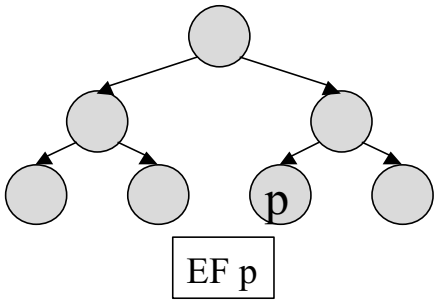
## Paires quantificateurs - combinateurs

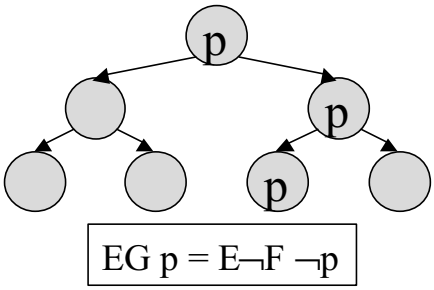
- ***AG* P : P *est toujours vraie, quelle que soit l'exécution, quel que soit l'état (sécurité)***
  - AG ¬ (gntA ∧ gntB)
- ***EG* P : *il existe une exécution le long de laquelle* P *est toujours vraie***
  - EG ¬ reqA
- ***AX* P : *à partir de l'état courant tous les états suivants possibles vérifient* P**
- ***EX* P : *à partir de l'état courant il existe un état suivant possible qui vérifie* P**

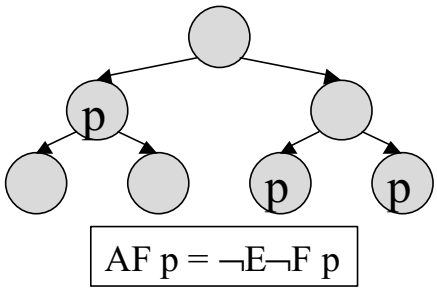


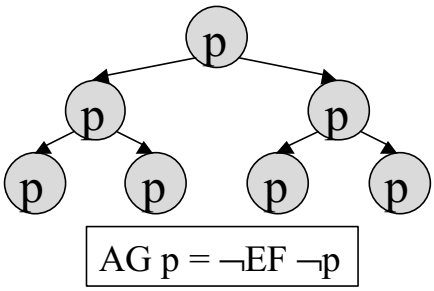
# Les logiques temporelles

## A, E, F et G



# Les logiques temporelles

- **Le rôle des quantificateurs de chemins est complexe :**
  - ***A GF* P *: quelle que soit l'exécution (A), à tout moment (G), on rencontrera inévitablement plus tard (F) un état vérifiant* P. *Autrement dit P sera inévitablement vérifiée une infinité de fois (A GF* P ≡ *A F*$^{\infty}$ P*)***
  - ***AG EF* P *: quelle que soit l'exécution (A), à tout moment (G), il serait possible (E) d'atteindre (F)* P. *Autrement dit* P *est toujours potentiellement atteignable. AG EF* P *peut être vraie même si, dans une exécution donnée,* P *n'est jamais vraie.***



# Les logiques temporelles

- **CTL* : aucune contrainte d'utilisation des combinateurs temporels et des quantificateurs de chemins**
- **PLTL : c'est CTL* sans les quantificateurs de chemins**
  - ***PLTL s'intéresse à l'ensemble des exécutions, pas à leur organisation en arbre***
  - ***Les formules PLTL sont des formules de chemin (logique du temps linéaire)***
  - ***PLTL ne sait pas exprimer les potentialités : AG EF* P *(« *P *est toujours potentiellement atteignable ») ne s'exprime pas en PLTL***

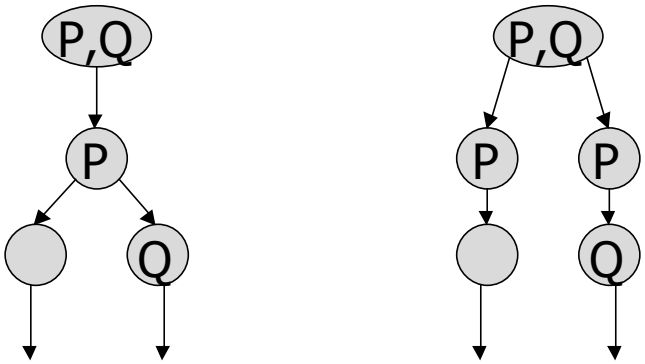
# Les logiques temporelles

- **CTL : c'est CTL* avec la contrainte que chaque combinateur temporel (X, F, G, U) est sous la portée d'un quantificateur**
  - ***Les formules CTL sont des formules d'état***
  - ***Les formules CTL ne dépendent que de l'état courant, pas de l'exécution courante***
  - ***CTL ne sait pas exprimer $F^{\infty}$***
- **FCTL : une extension de CTL (F pour Fair) qui permet d'exprimer $F^{\infty}$**
- **Le « model checking » de CTL et FCTL est beaucoup plus efficace que celui de PLTL**



# Les logiques temporelles

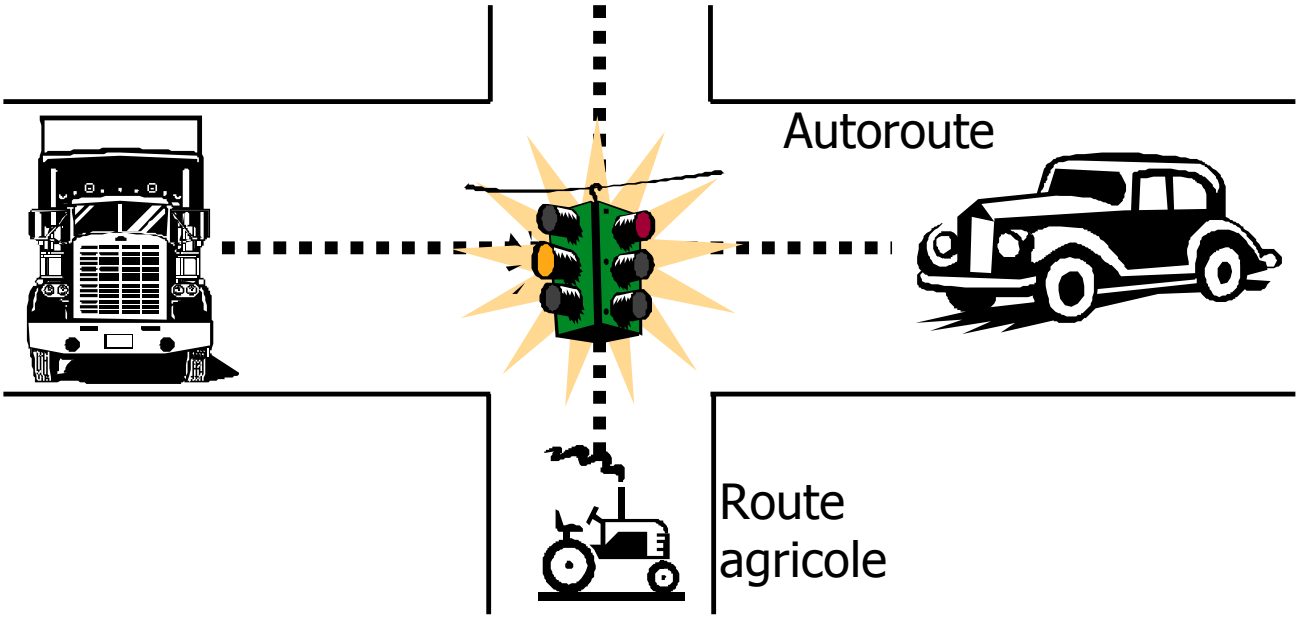
- **PLTL ne sait pas distinguer :**




- **CTL sait les distinguer. Comment exprimer en CTL que le choix entre $\neg Q \wedge \neg P$ et $Q \wedge \neg P$ reste ouvert plus longtemps dans un cas que dans l'autre ?**
  - ***AX (EX ¬ Q)***

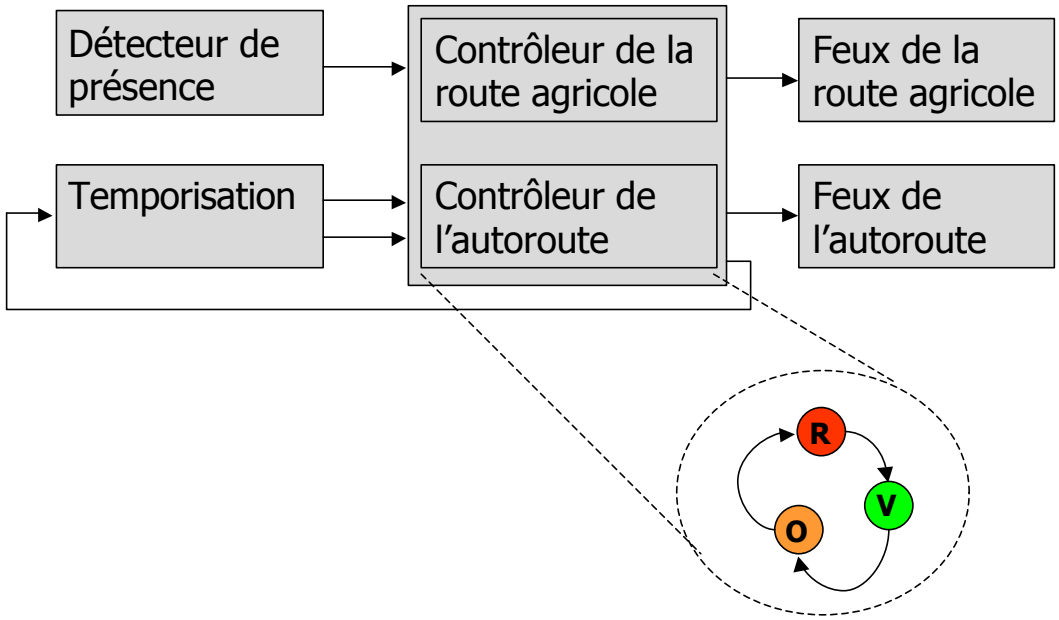
# La logique CTL

## Exemple super bateau : le feu tricolore






# La logique CTL



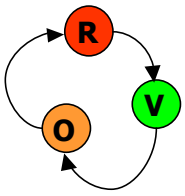
# La logique CTL

## Exemples de propriétés

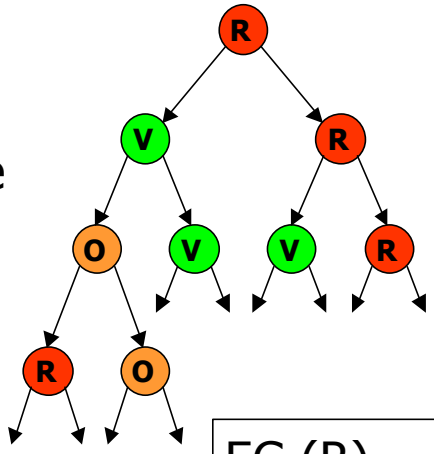
- *Les feux de l'autoroute et ceux de la route agricole ne sont jamais simultanément verts (sécurité)*
- *Si une voiture attend sur la route agricole elle finira par avoir le feu vert (vivacité)*



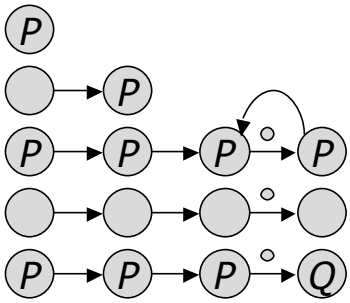
# Le temps arborescent




On déroule



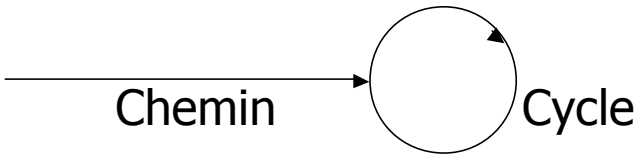

- $P$ :
- EX $P$ :
- EG $P$ :
- EF $P$ :
- E ($P$ U $Q$) :




| | |
|---|---|
| EG (R) | True |
| E (R U G) | True |
| AF (V) | False |
| AG EF(V) | True |
| A (R U V) | False |

# Les contre-exemples

- **Un contre-exemple peut-être fini**
  - ***Sécurité***
  - ***AG (R) : R, V***
- **Un contre-exemple peut-être infini**
  - ***Vivacité***
  - ***AF (V) : R, R, R, R, …***
  - ***Il est constitué d'un chemin et d'un cycle tous deux finis***






# Spécifications CTL du contrôleur

- **Si le client rend le bus, le contrôleur rend le jeton au cycle suivant :**
  - $AG\ (REQ \Rightarrow AX\ (REQ \vee (\neg\ REQ \wedge AX\ PSS)))$
- **Si le contrôleur reçoit un jeton alors que son client ne soumet pas de requête, il le rend au cycle suivant :**
  - $AG\ (TOQ \wedge \neg\ REQ \Rightarrow AX\ PSS)$
- **Si le client soumet une requête, lorsque le jeton arrive le contrôleur active GNT au cycle suivant :**
  - $AG\ (REQ \wedge TOQ \Rightarrow AX\ GNT)$
- **Le contrôleur ne rend pas le jeton tant que le client n'a pas relâché sa requête :**
  - $\neg\ EF\ (REQ \wedge EX\ (REQ \wedge PSS))$

# La description de l'environnement du contrôleur

- Pour réduire la complexité du problème en limitant les comportements aux seuls réalistes
- Pour ne pas obtenir de faux contre-exemples
- Peut utiliser une logique temporelle :
  - $REQ \Rightarrow AX\,(REQ \vee ACK)$
- Peut utiliser un langage spécifique (EDL)
- Peut utiliser un HDL étendu au non-déterminisme

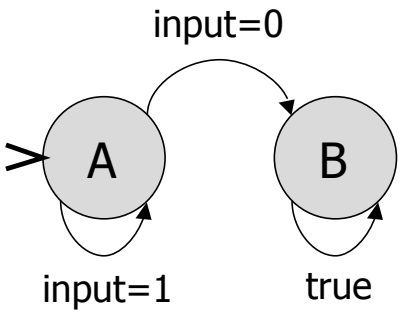


# Le non déterminisme

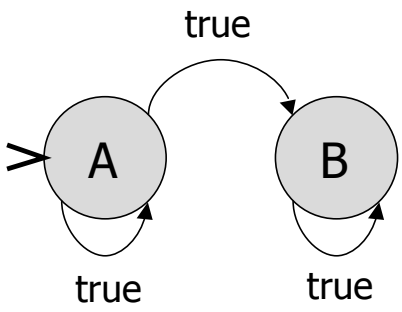
- Les systèmes déterministes
  - *Pour chaque paire {entrées, état} il y une seule paire {état suivant, sorties}*
  - *Les systèmes numériques implémentés sont toujours déterministes*
- Les systèmes non-déterministes
  - *Il existe une paire {entrées, état} pour laquelle la paire {état suivant, sorties} n'est pas unique*
  - *Décrit un ensemble de comportements*

# Le non déterminisme

Déterministe




Non-déterministe






# Pourquoi le non-déterminisme ?

- **Pour modéliser des environnements**
- **Exemple :**
  - ***Système à vérifier = arbitre + contrôleurs***
  - ***Environnement du système = clients***
    - **Une requête est soumise de façon non-déterministe**
    - **Le non-déterminisme permet de décrire la totalité de comportements possibles des clients**
- **Dans la suite $ND sera la construction utilisée pour modéliser le non-déterminisme :**

```
wire rand;
assign rand = $ND;
if (rand) …
```

```
signal RAND: BOOLEAN;
RAND <= $ND;
if (RAND) then
```

# Le non déterminisme

```
entity CLIENT is
  port(CLK, RSTN, GNT: in BOOLEAN;
       ACK, REQ: out BOOLEAN);
end CTRL;

architecture BEH of CLIENT is

  type STATE_TYPE is (IDLE, REQUEST, BUSY);
  signal STATE: STATE_TYPE;

begin

  process(CLK, RSTN)
  begin
    if (not RSTN) then
      ACK <= FALSE;
      STATE <= IDLE;
    elsif RISING_EDGE(CLK) then
      ACK <= FALSE;
      case STATE is
        when IDLE => if ($ND) then
                       STATE <= REQUEST;
                     end if;
…
```

```
…
        when REQUEST => if (GNT and $ND) then
                          ACK <= TRUE;
                          if ($ND) then
                            STATE <= BUSY;
                          else
                            STATE <= IDLE;
                          end if;
                        end if;
        when BUSY => if ($ND) then
                       STATE <= IDLE;
                     end if;
      end case;
    end if;
  end process;

  REQ <= STATE = REQUEST or STATE = BUSY;

end BEH;
```



# Les « fairness contraints »

- **Elles sont l'extension de CTL vers FCTL**
- **Elles expriment qu'un état ou un ensemble d'états doit être atteint un nombre infini de fois**
  - ***Exemple pour le client : $F^{\infty}$ IDLE (l'état IDLE doit être atteint un nombre infini de fois ; le client ne peut pas conserver le bus indéfiniment)***
- **Elles permettent de limiter les comportements non-déterministes**

# Les algorithmes du « model checking » de CTL

- Dus à Queille, Sifakis, Clarke, Emerson et Sistla (1982, 1986)
- Améliorés ensuite à de nombreuses reprises
- Linéaires en taille de l'automate et de la formule
- Basés sur le marquage des états
  - *Soit un automate* A *et une formule CTL* Φ*, pour chaque sous-formule* Ψ *de* Φ *et pour chaque état* q *de* A*, on marque* q *si* Ψ *est vraie dans l'état* q.
  - *A la fin, pour chaque état et chaque sous-formule* q.psi *vaut* `true` *si* q *vérifie* Ψ*,* `false` *sinon.*



# Les algorithmes du « model checking » de CTL

- La mémorisation est importante car le marquage de *q.phi* utilise les marquages des *q'.psi* pour des sous-formules *psi* de *phi* et des états *q'* atteignables à partir de *q*
- Quand le marquage de *phi* est terminé on regarde $q_0$*.phi* (où $q_0$ est l'état initial de *A*) et on sait alors si *A* vérifie *Φ*

# Les algorithmes du « model checking » de CTL

- **Notations :**
  - **Q** *est l'ensemble des états de* **A**
  - **l(q)** *est l'ensemble des propriétés atomiques* **P** *vérifiées par l'état* **q**
  - **T** *est l'ensemble des transitions* **(q,q')** *d'un état* **q** *de* **A** *à un état* **q'** *de* **A**
  - **degre(q)** *est le nombre de successeurs de* **q** *dans le graphe de* **A**
- **Cas n° 1 :** ***phi = P***

```
procedure marquage(phi)
    pour tout q dans Q, si P dans l(q) alors
                        faire q.phi = true,
                        sinon faire q.phi = false.
```

O(|Q|)



# Les algorithmes du « model checking » de CTL

- **Cas n° 2 :** ***phi = ¬ psi***

```
procedure marquage(phi)
    faire marquage(psi);
    pour tout q dans Q, faire q.phi := non (q.psi).
```

O(|Q|)

- **Cas n° 3 :** ***phi = psi1 ∧ psi2***

```
procedure marquage(phi)
    faire marquage(psi1); marquage(psi2);
    pour tout q dans Q, faire q.phi := et (q.psi1, q.psi2).
```

O(|Q|)

- **Cas n° 4 :** ***phi = EX psi***

```
procedure marquage(phi)
    faire marquage(psi);
    pour tout q dans Q, faire q.phi := false; // initialisation
    pour tout (q,q') dans T, si q'.psi := true alors
                             faire q.phi := true.
```

O(|T|)

# Les algorithmes du « model checking » de CTL

## Cas n° 5 : *phi* = E *psi1* U *psi2*

```
procedure marquage(phi)
    faire marquage(psi1); marquage(psi2);
    pour tout q dans Q,
      q.phi := false; q.dejavu := false; // initialisation
    L := {}; // L : ensemble des etats a traiter
    pour tout q dans Q,
      si q.psi2 := true alors faire L := L + {q};
    tant que L non vide {
      prendre un q dans L; // il faut marquer q
      L := L - {q};
      q.phi := true;
      pour tout (q',q) dans T { // q' est un predecesseur de q
        si q'.dejavu := false alors faire {
          q'.dejavu := true;
          si q'.psi1 := true alors faire L := L + {q'};
        }
      }
    }
```

O(|Q| + |T|)



# Les algorithmes du « model checking » de CTL

## Cas n° 6 : *phi* = A *psi1* U *psi2*

```
procedure marquage(phi)
    faire marquage(psi1); marquage(psi2);
    L := {}; // L : ensemble des etats a traiter
    pour tout q dans Q,
      q.nb := degre(q); q.phi := false; // initialisation
    pour tout q dans Q, si q.psi2 = true alors faire L := L + {q};
    tant que L non vide {
      prendre un q dans L; // il faut marquer q
      L := L - {q};
      q.phi := true;
      pour tout (q',q) dans T { // q' est un predecesseur de q
        q'.nb := q'.nb - 1;
        si (q'.nb = 0) et (q'.psi1 = true) et (q'.phi = false)
          alors faire L := L + {q'};
      }
    }
```

O(|Q| + |T|)

## Le « model checking » CTL d'une formule $\Phi$ est en $O(|A| \times |\Phi|)$

# La complexité du « model checking »

- L'explosion combinatoire en nombre d'états est très rapide
- De nombreux travaux de recherche tentent de repousser les limites (abstraction)
- La réduction préalable du problème est essentielle. On réduit :
  - *En exploitant les contraintes de l'environnement*
  - *En éliminant tout ce qui est sans rapport avec la propriété*
  - *Le « model checking » n'est donc pas exhaustif :*
    - Il travaille sur tous les cas mais pas sur la totalité du circuit
    - La simulation travaille sur la totalité du circuit mais pas sur tous les cas
    - La simulation permet de trouver des erreurs que l'on ne cherchait pas



# Exercices

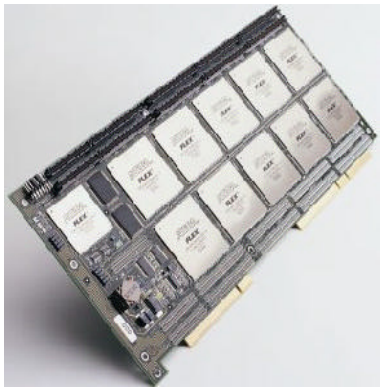
- « Si un client demande le bus il ne relâche pas sa requête avant d'avoir été servi ».
  - $AG\ (REQ \wedge \neg\ GNT \Rightarrow AX\ (REQ))$
- « Si le client A possède le bus et que le client B le demande il n'est pas possible que le client A relâche le bus puis le redemande et l'obtienne à nouveau avant le client B ».
  - $AG\ (gntA \wedge reqB \Rightarrow AX\ (gntA \vee A\ (\neg\ gntA\ U\ gntB) \vee AG\ (\neg\ gntA \wedge \neg\ gntB)))$

# Plan

- Introduction
- La simulation
- La vérification formelle
- L'accélération matérielle



# A quoi ça ressemble ?

# Les produits commerciaux

- Les énormes machines
  - *Meta Systems (Mentor Graphics)*
  - *Quickturn (Cadence)*
- Les machines moyenne gamme
  - *Aptix*
  - *Axis*
  - *Ikos*
- Les cartes génériques
  - *Nallatech*
  - *PLD Applications*

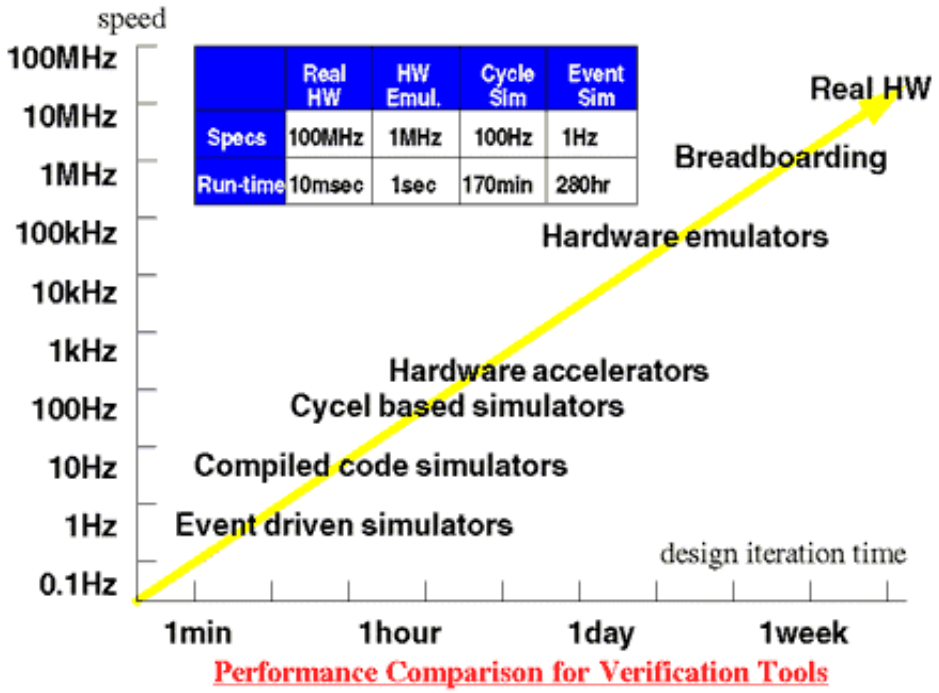


# Simulation ou prototype

- Un prototype peut permettre de valider des systèmes
  - *Simples*
  - *Utilisant des composants sur étagère*
- Pour des SOC complexes :
  - *La simulation remplace le prototype*
  - *Le prototype générique accélère la simulation*

# De la simulation à l'émulation

Source : Mentor Graphics Meta Systems Solutions (fabricant), 2000

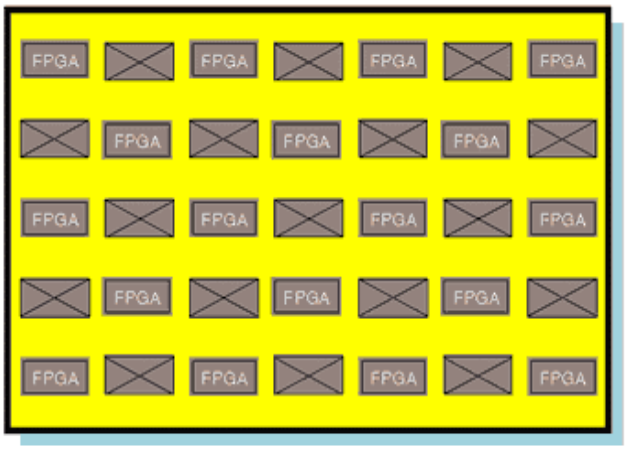





# Définitions

- **Accélérateurs matériels : machines spécialisées dans l'accélération de simulations cycle**
- **Emulateurs : réseaux de processeurs ou de FPGA. C'est une technologie cible, au même titre qu'une bibliothèque ASIC**
- **Breadboarding : carte de prototypage**

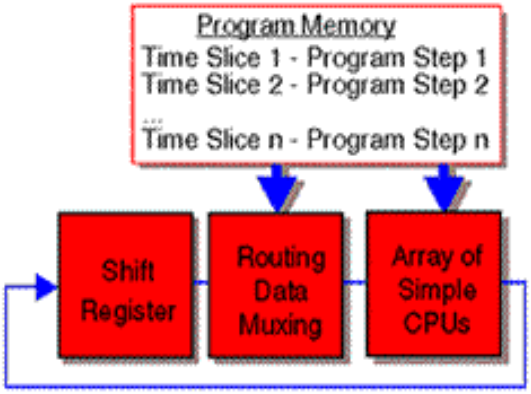
# Les types d'émulateurs




FPGA Based Emulator

FPGA commerciaux

FPGA propriétaires




Processor Based Emulator



# Les caractéristiques importantes

- Fréquence de fonctionnement
- Temps de compilation / configuration
- Limitations imposées sur le style de conception
  - *Synchrone*
  - *Domaines d'horloge multiples*
  - *« Gated clocks »*
- Taille des systèmes émulables
- Emulation dans le système d'accueil
- Debug interactif
- Co-simulation

# A quoi ça sert ?

- Accélérer la simulation du code synthétisable
  - *Attention, ça ne se justifie pas toujours*
  - *Attention, on ne simule pas ce qu'on croit*
- Accélérer la simulation de faute
- Emuler le circuit dans son environnement système
- Faciliter le développement conjoint des parties matérielles et logicielles
- Rassurer les concepteurs
- Chauffer les locaux



# A quoi ça ne sert pas ?

- Valider des modèles non synthétisables
- Simuler le circuit « tel qu'il sera en définitive »
  - *Pour être implantées en machine les « netlists » sont transformées*
    - Logique à 3 états -> multiplexage
    - Mémoires -> mémoires disponibles
    - …
- Vérifier des caractéristiques physiques
- Valider des transformations de « netlist »

# Faut-il émuler le code synthétisable ?

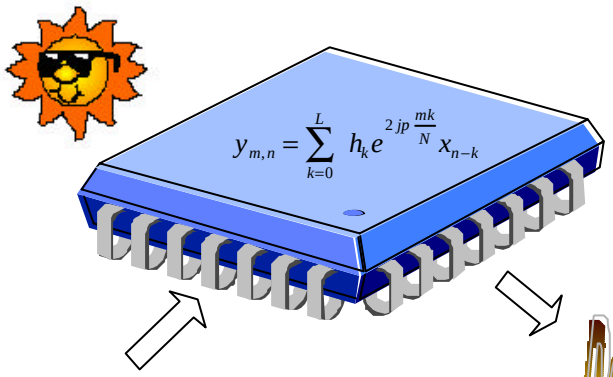
- Validation système ?
  - *Il est préférable de disposer d'un modèle plus haut niveau*
  - *Comparer les performances*
- Validation du raffinement vers la synthèse ?
  - *A comparer avec les solutions concurrentes :*
    - « model checking »
    - Preuve d'équivalence
    - Simulation multi-niveau



# Conclusion

- C'est compliqué
- C'est cher
- C'est pas toujours très bien payé
- C'est pas toujours très bien considéré
- Mais ça peut être vraiment intéressant
- Un domaine plein de potentialités

# La consommation




**DESSIN**




# PLAN

- Justifications
- Modélisation de la consommation
- Optimisation technologique
- Evaluation de la consommation
- Conception en vue de la basse consommation

# PLAN

- Justifications
- Modélisation de la consommation
- Optimisation technologique
- Evaluation de la consommation
- Conception en vue de la basse consommation



# Justifications

- Grande puissance dissipée
  - *Surchauffe des circuits*
  - *Performances dégradées (Freq = f(temp))*
  - *Durée de vie limitée*
  - *Boîtiers spécialisés coûteux*
  - *Systèmes de ventilation coûteux*
- Exemple : « fermes de PC » pour centres Internet
- USA : 10% Energie consommée / informatique…

# Justifications

- **Electronique nomade**
  - ***Durée de vie des batteries***
  - ***Faible croissance des performances des batteries***
  - ***Forte croissance des puissances de calcul***
  - ***Energie dans une batterie format AA : max 2wh***
    - **Soit 2W pendant une heure, 1W pendant 2 heures…**



# PLAN

- **Justifications**
- **Modélisation de la consommation**
- **Optimisation technologique**
- **Evaluation de la consommation**
- **Conception en vue de la basse consommation**

# Quelques définitions

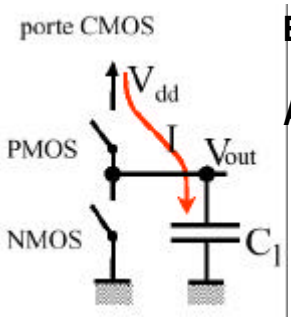
- **Puissance dissipée instantanée maximale**
  - ***Des pics de courants peuvent se produire dans les circuits CMOS (front d'horloge)***
  - ***Conséquences :***
    - **Baisses de tension locales**
    - **Performances réelles # performances estimées**
    - **Durée de vie réduite**
  - ***Règles de dessin à respecter***
    - **Largeur des lignes d'alimentation**
    - **Largeur des lignes d'horloges…**
- **Puissance moyenne dissipée**
  - ***Tension d'alimentation * Courant moyen***



# Courants dans une porte CMOS

- **Courants dans un inverseur**
  - ***Courant de charge des équipotentielles***
    - **Charges des drains des transistors NMOS et PMOS**
    - **Charges des grilles des transistors NMOS et PMOS**
    - **Charges des capacités des connections**
  - ***Courants de court-circuits***
    - **Pendant les transitions NMOS et PMOS conduisent simultanément**
  - ***Courants de fuite***
    - **Un transistor NMOS conduit même si Vgs < Vt**
    - **Les drains et sources forment des diodes avec le substrat**

# Energie de charge et décharge




**Effet joule :**

$1/2CV_{dd}^2$ à chaque transition

**Alimentation :**

$CV_{dd}^2$ à chaque transition montante

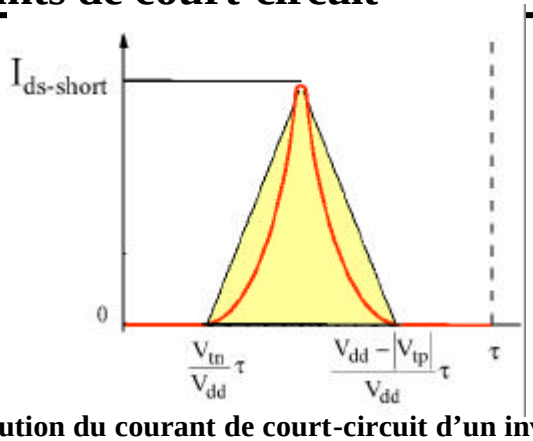
Diminuer C
Diminuer Vdd

$$E_T = \int_0^{tf} V_{dd} I_C dt = \int_0^{tf} V_{dd} C \frac{dv}{dt} dt = \int_0^{V_{dd}} V_{dd} C dv = C V_{dd}^2$$

$$E_S = \int_0^{tf} U_C I_C dt = \int_0^{tf} U_C C \frac{dv}{dt} dt = \int_0^{V_{dd}} U_C C dv = \frac{1}{2} C V_{dd}^2$$



# Courants de court-circuit




- **Evolution du courant de court-circuit d'un inverseur pendant une transition**
- t **est le temps de transition en entrée**

# Courants de court-circuit

- **Si**
  - ***les transistors NMOS et PMOS sont de caractéristiques électriques identiques***
  - ***Et le nœud de sortie n'a aucune capacité parasite (..)***
- **Alors le courant maximal est**
  - $Isc\_Max = K(W/L)(Vdd/2 - Vt)^2$
- **Energie (modèle en triangle):**

$$E_{sc} = \int_{\tau\frac{Vt}{Vdd}}^{\tau\frac{(Vdd-Vt)}{Vdd}} V_{dd} I_{sc} dt = \tau I_{sc\max} \left(\frac{V_{dd}}{2}\right)\left(\frac{V_{dd}-2V_t}{V_{dd}}\right) =$$

$$E_{sc} = \tau \frac{K}{2}\frac{W}{L}\left(V_{dd}-2V_t\right)^3$$



# Courants de court-circuit

- **Dans la réalité**
  - ***Le nœud de sortie a une capacité parasite (..)***
  - ***Dans le cas d'une charge (entrée descendante)***
    - **Isc-réel = Ipmos – Icapa**
  - ***Dans le cas d'une décharge (entrée montante)***
    - **Isc-réel = Inmos - Icapa**
- **Diminuer Vdd**
- **Diminuer les temps de transition en entrée**
- **Augmenter les capas en sortie (Icapa)**
- **Potentiellement nul si Vdd < 2 Vt**

# Courants de fuite

- **Courant des transistors sous le seuil**
  - ***Isub = K1*exp(-Vt)***
  - ***Toute diminution de 0.1 V de la tension de seuil des transistors multiplie par 10 le courant de fuite***
  - ***Ne pas diminuer Vt***
- **Courant des diodes**
  - ***Id = Is*(exp(Vd/Vr)-1)***
  - ***Il suffit de bien polariser en inverse, Id ne dépasse pas Is***

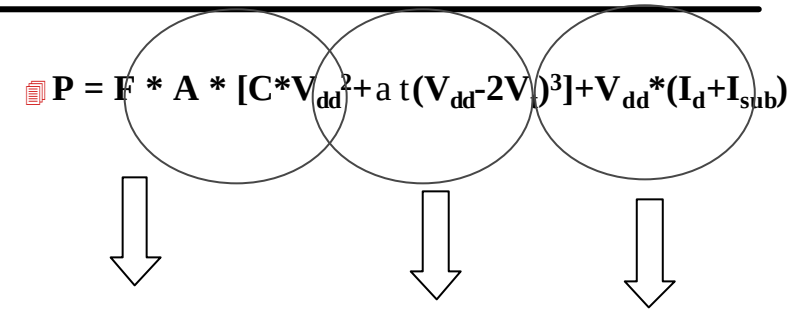


# Puissance dynamique

- **Puissance**
  - ***Energie / Temps***
- **Energie**
  - ***(Energie/Transition)****Nombre de transitions***
- **Nombre de transitions**
  - ***(Nombre de transitions/Sec) *Temps***
- **Nombre de transitions/Sec**
  - ***Fréquence d'horloge*Activité***

# Puissance totale

- $P = F * A * [C*V_{dd}^2 + a\,t(V_{dd}-2V_t)^3] + V_{dd}*(I_d+I_{sub})$



Terme principal

Limité en contraignant les temps de transitions (biblilothèques ASIC)

Contrôle du Vt Silicium sur Isolant



# Valeurs typiques (vieille techno…)

- **Technologie CMOS 0,7 mm , 5v**
- **Charge typique 100 fF**
- **Ec : 1 picoJoule**
- **Esc : 0,1 picoJoule**
- **Ifuite = 0,2 nA /porte**
- **Circuit de 100k portes à 50 Mhz**
  - ***Pcmax = 5W PscMax = 0,5W Pfuite=0,0001W***

# Conséquences…

- **logique CMOS classique**
  - ***Négliger la consommation due aux fuites***
  - ***Agglomérer consommation de court-circuit et consommation statique***
- **Logiques exotiques, RAM, parties analogiques**
  - ***La consommation de court circuit contient une partie constante indépendante de la fréquence et peu devenir très importante***



# PLAN

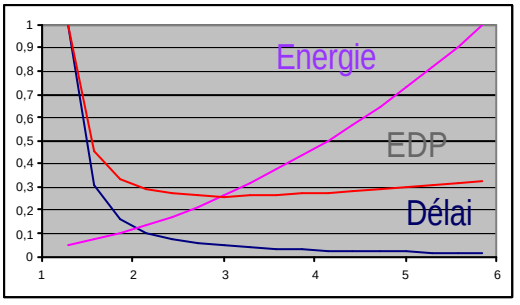
- **Justifications**
- **Modélisation de la consommation**
- **Optimisation technologique**
- **Evaluation de la consommation**
- **Conception en vue de la basse consommation**

# Exemple de recherche de compromis

- **Temps de charge d'une capacité:**
  - ***Charge totale : Q = CVdd***
  - ***Courant de charge : Ids= K(Vdd-Vt)²***
  - ***Temps de charge : Q/Ids = KCVdd/(Vdd-Vt) ²***
- **Fréquence maximale de fonctionnement**
  - ***F = 1/T***
  - ***T ∝ Vdd/(Vdd-Vt) ²***
- **Energie par transition**
  - ***E = CVdd²***



# Choix Tension de Seuil / Alimentation




- **Critère d'optimisation : Produit Energie x Delai**
- **Optimum : Vdd = 3 Vt**
- **Attention : logique à multiplexage (perte d'un Vt..)**

## Très basse consommation

- On accepte de perdre en performances
- Vdd = 2*Vt
- Fréquence de fonctionnement divisée par 2
- Energie par transition divisée par deux
- A puissance de calcul égale (parallélisme x 2)
- Puissance moyenne au moins divisée par deux…
- Plus de logique à multiplexage …



## PLAN

- Justifications
- Modélisation de la consommation
- Optimisation technologique
- Evaluation de la consommation
- Conception en vue de la basse consommation

# Estimation par simulation électrique

- Hypothèse : un bloc de logique combinatoire
- Outil : simulateur Electrique type Spice
- Générer toutes les transitions d ’entrées possibles ($2^{2n}$ pour n entrées)
- Simuler, mesurer l’énergie consommée
- Pondérer , pour chaque transition par sa probabilité d’occurrence
- Lourd, quasi insurmontable



# Estimation par simulation électrique

- Travailler de manière statistique:
- Estimer la puissance moyenne par tranches de p vecteurs pris au hasard (p,2p,3p,4p…)
- Lorsque d’une tentative à l’autre l’évaluation de la puissance n’évolue plus, stopper le processus…
- Mais on ne peut plus pondérer…

# Pour accélérer la simulation

- **Un simulateur au niveau interrupteur :**
  - ***Transistor = Resistance + Capa + Interrupteur***
- **Simulation purement temporelle**
- **Permet(tait) de simuler des circuits entiers**
- **Mais la complexité des circuits progresse plus vite que la vitesse des simulateurs…**



# Traitement au niveau « porte »

- **Caractériser une porte en consommation**
  - ***Simulation électrique***
  - ***Exemple : une porte nand = 25pW/Mhz***
- **Déterminer les capacités de charge en sortie de chaque porte (déjà fait car nécessaire pour l'estimation de la vitesse)**
- **Déterminer l'activité de chacune des sorties des portes**
- **Calculer la consommation par simple sommation :**

$$P = F \times \sum Acti \times (Egatei + CiVdd2)$$

# Déterminer l'activité

- **Solution 1 : Simulation logico-temporelle**
- **Utilisation de simulateurs logiques rapides**
- **Résultats relativements bons.**
- **Attention : les transitions courtes ou incomplètes sont ignorées : sous estimation de la consommation**



# Déterminer l'activité

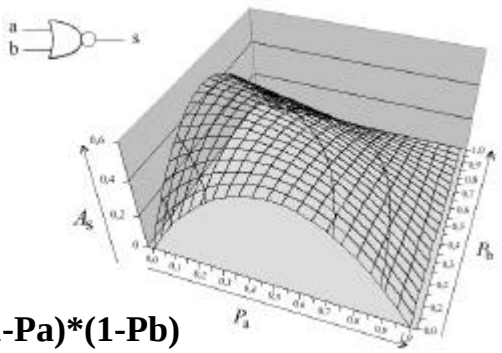
- **Solution 2 : Calcul analytique**
  - ***Propagation des probabilités dans la logique***
  - ***Calcul direct pour une fonction complexe***
- **Définitions :**
  - ***Pa = probabilité que le nœud « a » soit à un***
  - ***Pt= probabilité de transition du nœud a***

# Probabilité de transition

- **Probabilité de transition 0->1**
  - ***P01 = (1-Pa)*Pa***
- **Probabilité de transition 1->0**
  - ***P10 = Pa*(1-Pa)***
- **Probabilité de transition ou activité**
  - ***Pt = 2*Pa*(1-Pa)***
- **Attention : la formule n'est valable que si les valeurs successives de a sont décorrélées.**
- **Cas particulier : Pa=1/2 Pt=1/2**

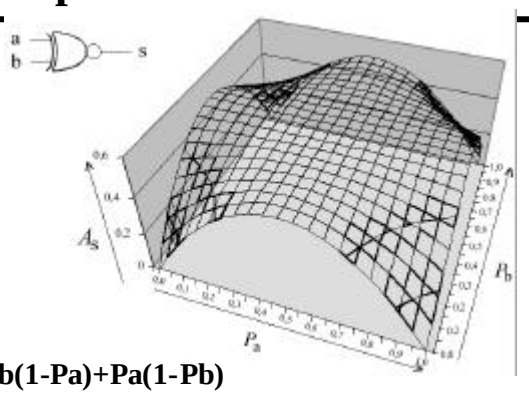


# Exemple : la porte Nor




- **Ps = (1-Pa)*(1-Pb)**
- **As = 2(1-(1-Pa)*(1-Pb))(1-Pa)*(1-Pb)**
- **(Pa=1/2 Pb=1/2) Ps=1/4 As=3/8**
- **Attention : valable si a et b sont décorrelées**

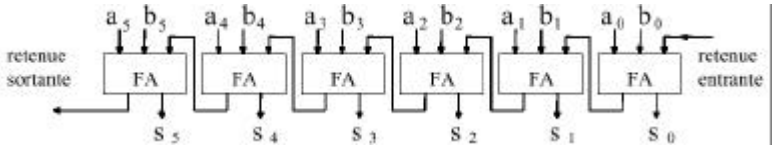
# Exemple : la porte Xor




- **Ps = Pb(1-Pa)+Pa(1-Pb)**
- **As = 2(1-(Pb(1-Pa)+Pa(1-Pb)))(Pb(1-Pa)+Pa(1-Pb))**
- **(Pa=1/2 Pb=1/2) Ps=1/2 As=1/2**
- **Attention :Pas de diminution de l'activité..**



# L'aditionneur à propagation de retenue




- **Si Pai=1/2 Pbi=1/2 pas de corrélation**
  - ***Pri=1/2, Psi=1/2***
- **Si on ne tient pas compte des temps de propagation dans les cellules : A(n) = n/2**
- **Mais : transitions multiples…**
- **Modèle à délai = 1**

# Pire cas de transitions

| A/B | 1/1 | 0/0 | 1/1 | 0/0 | 1/1 | 0/0 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |

- Valeur des retenues pour un premier couple A/B



# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |

- Nouveau couple A/B

# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |

- Changement des retenues



# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |

- Propagation des retenues

# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |
| R(3) | 0 | 1 | 0 | 1 | 1 | 1 |

- Propagation des retenues



# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |
| R(3) | 0 | 1 | 0 | 1 | 1 | 1 |
| R(4) | 1 | 0 | 1 | 1 | 1 | 1 |

- Propagation des retenues

# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |
| R(3) | 0 | 1 | 0 | 1 | 1 | 1 |
| R(4) | 1 | 0 | 1 | 1 | 1 | 1 |
| R(5) | 0 | 1 | 1 | 1 | 1 | 1 |

- Propagation des retenues



# Pire cas de transitions

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |
| R(3) | 0 | 1 | 0 | 1 | 1 | 1 |
| R(4) | 1 | 0 | 1 | 1 | 1 | 1 |
| R(5) | 0 | 1 | 1 | 1 | 1 | 1 |
| R(6) | 1 | 1 | 1 | 1 | 1 | 1 |

- Propagation des retenues

# Pire cas de transitions

- Activité des retenues $n^2/2$

| A/B | 1/0 | 1/0 | 1/0 | 1/0 | 1/0 | 1/1 |
|---|---|---|---|---|---|---|
| R(0) | 1 | 0 | 1 | 0 | 1 | 0 |
| R(1) | 0 | 1 | 0 | 1 | 0 | 1 |
| R(2) | 1 | 0 | 1 | 0 | 1 | 1 |
| R(3) | 0 | 1 | 0 | 1 | 1 | 1 |
| R(4) | 1 | 0 | 1 | 1 | 1 | 1 |
| R(5) | 0 | 1 | 1 | 1 | 1 | 1 |
| R(6) | 1 | 1 | 1 | 1 | 1 | 1 |
| Transitions | 6 | 5 | 4 | 3 | 2 | 1 |



# Bilan d'activité de l'additionneur

- Les transitions sur la somme sont déterminées par celles de la retenue
- Activité pire cas d'un additionneur n bits : $n^2/2$
- Activité moyenne asymptotiquement : 3n/4
- Activité « Utile » : n/2
- Activité « parasite » : n/4

# Evaluation au niveau architecture

- L'évaluation de la consommation par simulation ne peut qu'intervenir tardivement dans la conception
- Des modèles « globaux » peuvent permettre de faire une estimation rapide et même guider des choix d'architecture
- Utiliser des tables de bilan d'énergie par opérateur
- Déterminer pour l'algorithme à implanter le nombre de ces opérations



# Bilan d'énergie

| Module/Opération | Energie (pJ) | |
|---|---|---|
| Addition en carry select (16 bits) | 18 | 1 |
| Multiplieur (16 bits) | 64 | 3,6 |
| Bascule D (16 bits) | 4 | 0,22 |
| Lecture en RAM (8x128x16) | 80 | 4,4 |
| Ecriture en RAM(8x128x16) | 160 | 9 |
| Accès externe (16 bits) | 180 | 10 |

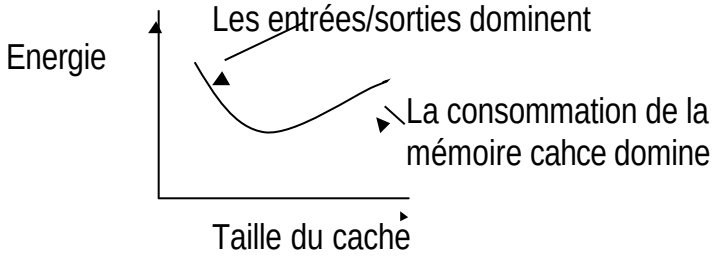
# Bilan d'énergie

- **Très facile à utiliser pour des architectures « flot de données »**
- **Permet d'évaluer le coût des accès externes**
- **Pas utilisable pour le contrôle**
- **Intégrable dans des outils « haut niveau »**



# Exemple

- **Consommation d'un système de cache**
  - ***Consommation de la mémoire cache***
  - ***+ Consommation des accès externes***

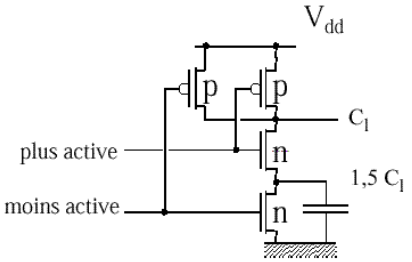


# PLAN

- Justifications
- Modélisation de la consommation
- Optimisation technologique
- Evaluation de la consommation
- Conception en vue de la basse consommation

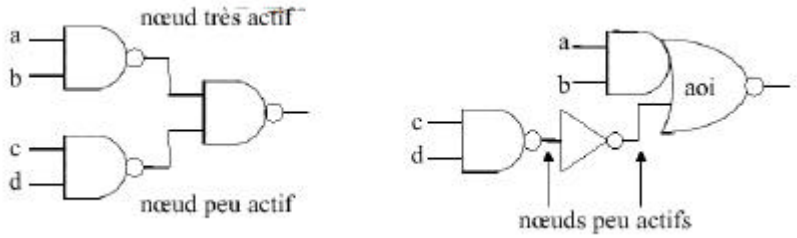


# Activité




- Réordonnancement des données
- Automatisable
- Minimiser Somme(Ai*C1i) sur l'ensemble des entrées

# Activité




- Dissymétrie des chemins pour diminuer la capacité sur le chemin le plus actif
- Power Compiler (Synopys)



# Réduction de l'activité redondante

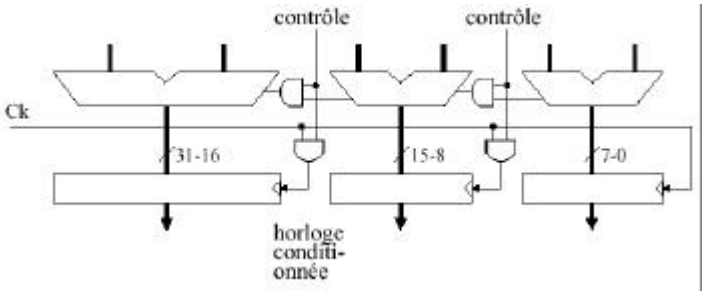
- De 10% à 40% de l'activité totale
- Equilibrer les délais des chemins convergent vers une même porte
- Pas d 'activité redondante en sortie de registre: insérer du pipeline…
- Exemple : accumulation en arbre plus intéressante que l'accumulation linéaire
- Exemple : carry-save plus intéressant que propagation

# Horloge conditionnée

- Désactiver un bloc du circuit lorsqu'il n'est pas utilisé
- Solution : conditionner l'horloge par de la logique
- Problème : testabilité, fiabilité de l'évaluation de performance (qualité des outils)
- Calculer la condition de désactivation

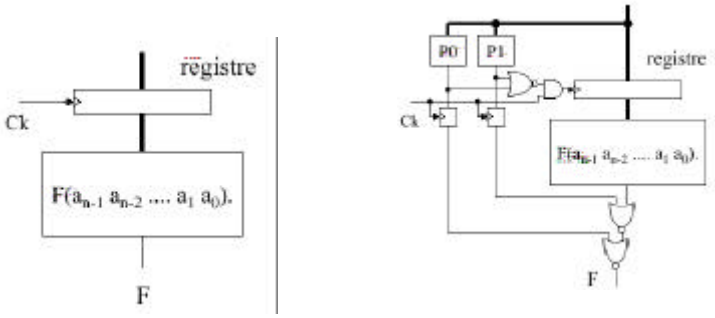


# Ajustement de la dynamique des ALU




- Dans un mp : le décodage de l'instruction permet de déterminer la largeur des données

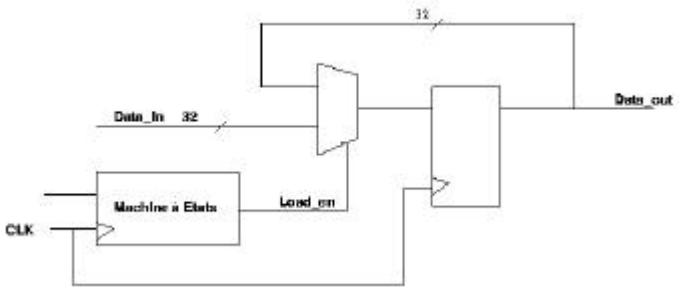
# Précalcul




- **Idée : trouver des fonctions P0 et P1 (plus simples que F) telles que (P0=1 -> F=0) et (P1=1 -> F=1)**

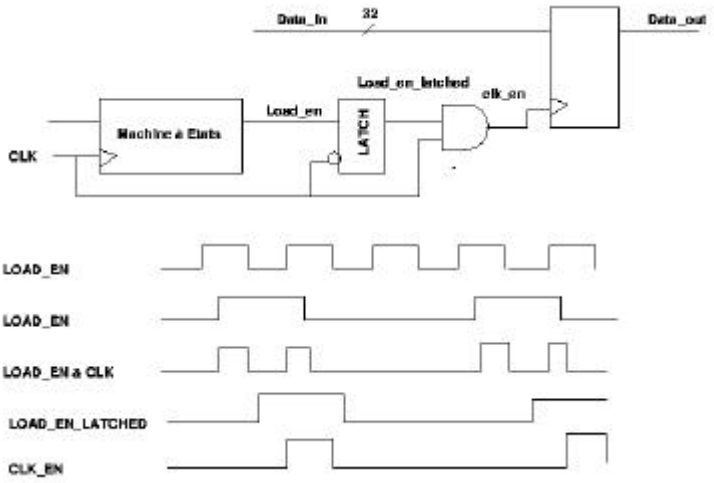


# Horloge conditionnée




- **Dans cet exemple le surcoût du conditionnement est mineur devant le nombre de registres concernés.**
- **Attention à la réalisation du conditionnement**

# Horloge conditionnée






# Représentation des données

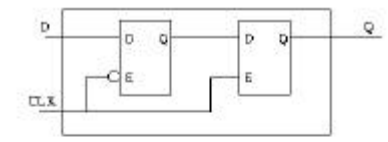
- On peut diminuer le taux d'activité en utilisant une représentation des données adaptée
- En complément à 2 des données centrées sur zero génèrent une forte activité sur les poids forts : utiliser du « binaire signé »
- De manière plus générale : exploiter la connaissance du signal traité (image, voix…)

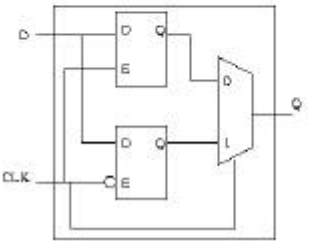
# Signal d'horloge

- Signal fortement chargé, jusqu'à 30% de la consommation du circuit.
- Faire un arbre d'horloge optimisé en consommation (si la vitesse le permet)
- Utiliser des bascules sensibles sur les deux fronts (permet de diviser par deux la fréquence d'horloge)



# Bascule active sur les deux fronts




- Bascule D classique
- Bascule D active sur 2 fronts

# Dans l'avenir

- **De nombreuses méthodes sont proposées:**
  - ***Logique à très faible excursion***
  - ***Logique adiabatique***
  - ***Logique autoséquencée***
- **Toutes nécessitent une remise en cause des techniques de conception automatisées**
- **Ne peuvent émerger qu'associées à des outils**

# Vitesse



# Plan

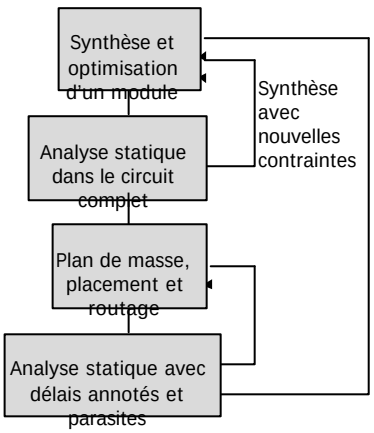
- Introduction
- Les modèles temporels
- La définition d'environnement
- L'analyse
- La diaphonie
- Conclusion

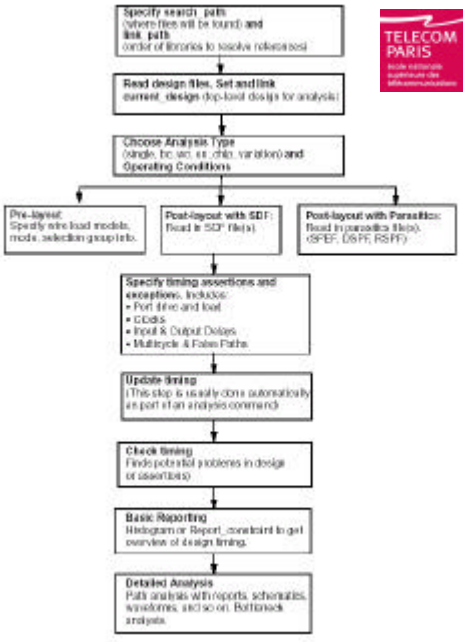
# Plan

- **Introduction**
- **Les modèles temporels**
- **La définition d'environnement**
- **L'analyse**
- **La diaphonie**
- **Conclusion**



# Introduction



# Utilisations de l'analyse temporelle statique

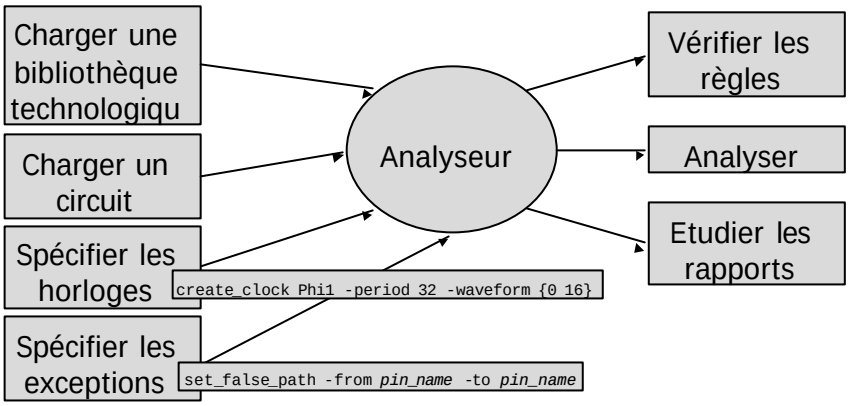
- **Temps de pré-positionnement (*setup*) et de maintient (*hold*) des éléments séquentiels**
- **Temps de pré-positionnement (*setup*) et de maintient (*hold*) des horloges commandées (*gated clocks*)**
- **Périodes et phases minimales des horloges**
- **Temps de propagation minimums et maximums**
- **Contraintes d'arrivées aux sorties**
- **Contraintes de déphasages maximums d'horloges (*skew*)**
- **Exploration PVT (*Process, Voltage, Temperature*)**
- **Analyse de diaphonie (*crosstalk*)**



# Analyse statique / simulation

- **Analyse statique**
  - ***Méthode exhaustive***
  - ***On analyse un circuit, on caractérise tous les chemins possibles et on les compare avec les contraintes***
  - ***Inconvénients :***
    - **Souvent limitée aux parties synchrones**
    - **Identifie fréquemment des faux chemins**
    - **Souvent pessimiste**
- **Simulation**
  - ***On simule une* netlist*annotée en temps***
  - ***Avec des vecteurs d'entrée***
  - ***Pas de faux chemins***
  - ***Utilisable sur des circuits synchrone ou non***
  - ***Inconvénients :***
    - **Lent**
    - **Le chemin critique n'est pas forcément exercé**

# L'environnement de l'analyseur statique






# Exemple de rapport d'analyse

```
****************************************
Report : timing
-path full
-delay max
-max_paths 1
Design : TOP
****************************************
Startpoint: INBUS_0
(input port clocked by Phi1)
Endpoint: XOUT_2
(output port clocked by Phi1)
Path Group: Phi1
Path Type: max
Point                                          Incr Path
-----------------------------------------------------------
clock Phi1 (rise edge)                         0.00  0.00
clock network delay (ideal)                    0.00  0.00
input external delay                           24.00 24.00
^
INBUS_0 (in)                                   0.00  24.00
^
ALU_0/U678/YN (nor3_b)                         2.24  26.24
^
ALU_0/U677/YN (nand3_c)                        1.12  27.36
^
ALU_0/U683/YN (nand2_c)                        0.83  28.19
^
ALU_0/U691/YN (inv_d)                          0.47  28.66
^
ALU_0/U689/YN (o2ai_b)                         0.93  29.59
^
ALU_0/U692/YN (inv_c)                          0.71  30.30
^
XOUT_2 (out)                                   0.00  30.30
^
data arrival time                                    30.30

clock Phi1 (rise edge)                         32.00 32.00
clock network delay (ideal)                    0.00  32.00
clock uncertainty                              -1.60 30.40
output external delay                          -4.00 26.40
data required time                             26.40
-----------------------------------------------------------
data required time                             26.40
data arrival time                              -30.30
-----------------------------------------------------------
slack (VIOLATED)                               -3.90
```

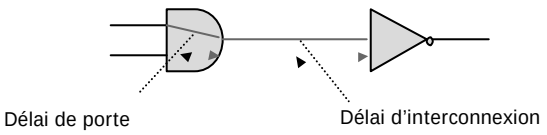
# Plan

- Introduction
- Les modèles temporels
- La définition d'environnement
- L'analyse
- La diaphonie
- Conclusion



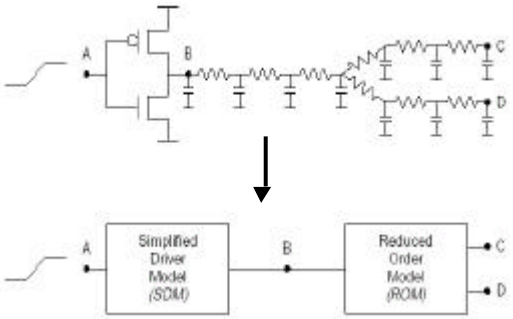
# Les deux sources de délais

- Les délais des portes logiques (temps de propagation d'une entrée quelconque de la porte vers une sortie quelconque de la porte)
- Les délais d'interconnexions (temps de propagation d'une sortie quelconque de porte vers une entrée quelconque d'une autre porte)



# Les modèles simplifiés

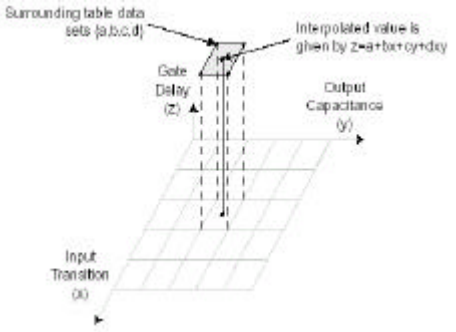
- **Comment passer d'un modèle « à la SPICE » à un modèle simplifié ?**



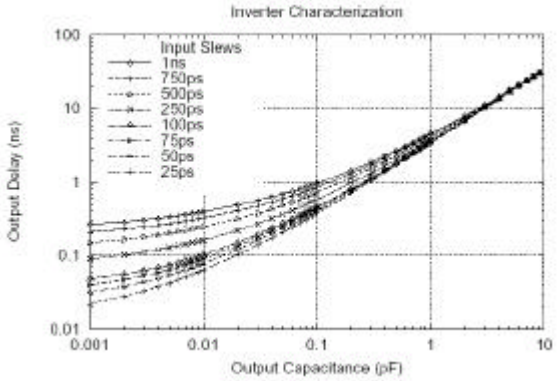



# Tables et interpolation

- **On utilise une bibliothèque de données pré-caractérisées :**
  - $D_{out} = F(C_{out}, S_{in})$
  - $S_{out} = G(C_{out}, S_{in})$
- **On interpole**

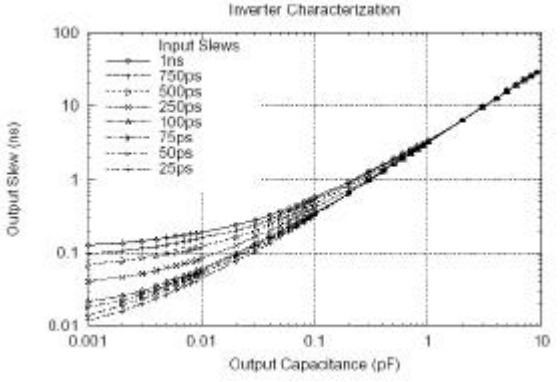


# Exemple de caractérisation en délai






# Exemple de caractérisation en pente



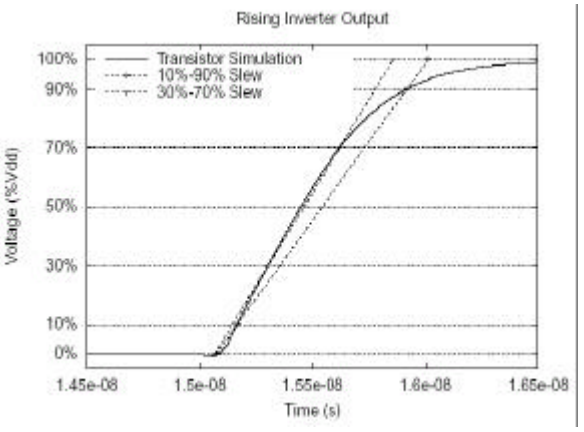
# Conditions de validité du modèle

- Le pas de quantification des tables doit être suffisamment fin
- Il peut être variable
- L'extrapolation (calculs sur des points de fonctionnement hors de la table) peut introduire de très importantes erreurs
- Il faut respecter les limites imposées par le fondeur pour que les calculs restent valables



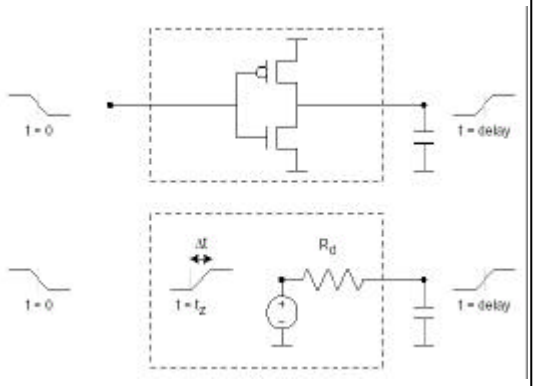
# Instants d'observation

- Ce choix des instants d'observation est déterminant
- D'une définition à l'autre les écarts sont importants



# Paramètres des modèles

- **Trois paramètres**
  - ***Resistance du pilote***
  - ***Date de début de rampe***
  - ***Durée de rampe***
- **Un modèle pour chaque couple entrée / sortie et pour chaque sens de transition en sortie**






# Simplifications

- **L'analyseur construit un modèle simplifié de porte à partir de la bibliothèque :**
  - ***$t_z$ = délai de la bibiothèque***
  - ***$D_t$ = pente de la bibliothèque***
  - ***$R_d$ = sensibilité de la bibliothèque à la capacité de charge***
- **Puis il simplifie le réseau RC en une unique capacité de charge $C_{eff}$, telle que le délai introduit par le modèle simplifié connecté au réseau RC réel soit égal au délai de la bibliothèque pour $C_{eff}$**

# Plan

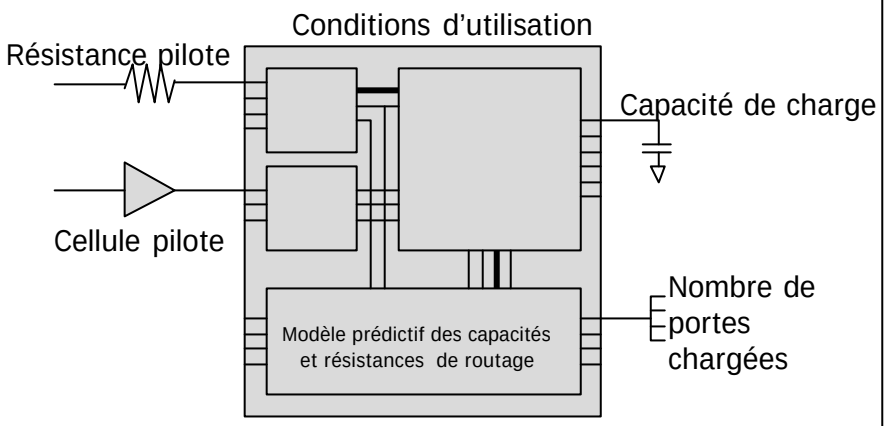
- Introduction
- Les modèles temporels
- La définition d'environnement
- L'analyse
- La diaphonie
- Conclusion



# Les définitions nécessaires

- Formes d'onde des horloges et caractéristiques
- Délais d'entrée / sortie
  - *Date d'arrivée de chaque entrée (% horloge)*
  - *Date attendue de positionnement des sorties*
- Pilotes des entrées
  - *Les types de cellules qui pilotent les entrées*
  - *Ou les résistances équivalentes*
  - *Ou encore les durées de transition*
- Les capacités externes aux entrées / sorties
- Les conditions d'utilisation
  - *Température*
  - *Tension d'alimentation*
  - *Qualité du procédé de fabrication*
- Modèle prédictif de capacité et de résistance de routage

# Les définitions nécessaires



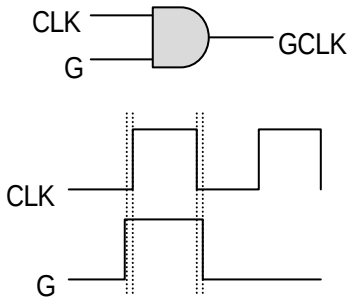



# Les horloges

- **Elles peuvent être multiples, de fréquences et de formes d'onde différentes**
  - ***Réelles et ont pour source l'une des entrées du circuit***
  - ***Virtuelles et servent alors de référence pour d'autres définitions***
- **Elles peuvent être générées à partir d'une autre horloge (diviseur de fréquence, par exemple)**
- **Latences, déphasages et temps de transition peuvent servir de spécification à la synthèse de l'arbre ou être extraits d'un arbre déjà réalisé**

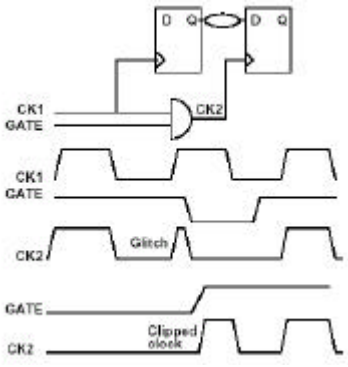
# Les horloges

- **Les horloges peuvent être contrôlées par de la logique (*gated clocks*). Il faut alors vérifier les temps de pré-positionnement (*setup*) et de maintient (*hold*) sur le signal de commande**

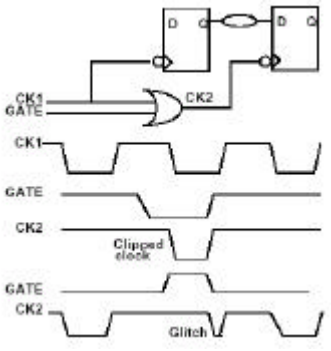





# *Setup* et *hold*



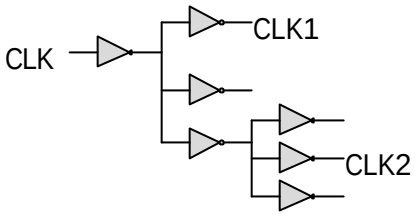

Violations de *setup*

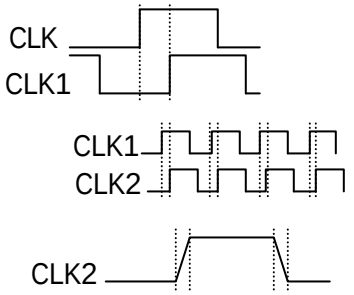



Violations de *hold*

# Arbres d'amplification d'horloge

- **Latence par rapport à la source**
- **Déphasages (*skew*) aux feuilles**
- **Temps de transition aux feuilles**






# Les conditions d'utilisation

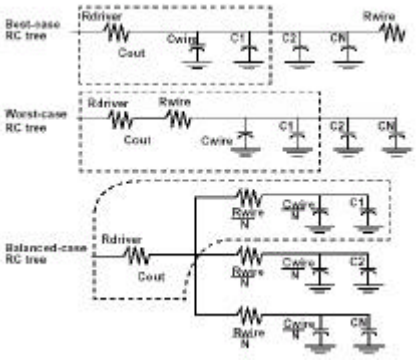
- **La qualité du procédé de fabrication**
  - ***Caractérise les variations***
  - ***Modélisée par un coefficient multiplicatif***
  - ***0,9 Þ mieux que l'hypothèse utilisée lors de la caractérisation de la bibliothèque ; 1,1 Þ moins bien***
- **La température**
  - ***Dépend de***
    - **La température de l'air ambiant**
    - **La consommation**
    - **Le type de boîtier (plastique, céramique)**
    - **Les méthodes de refroidissement (radiateurs, ventilateur, etc.)**
  - ***Modélisée par un coefficient multiplicatif***

# Les conditions d'utilisation

- **La tension d'utilisation**
  - ***Plus elle est élevée et plus les performances en vitesse augmentent …***
  - ***… et plus on consomme***
  - ***Modélisée par un coefficient multiplicatif***
- **Le type de modèle d'interconnexions :**
  - ***Deux nœuds avec les mêmes résistances et capacités mais des topologies RC différentes peuvent avoir des délais différents***



# Le modèle d'interconnexions



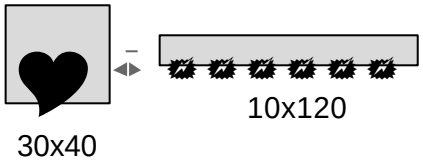
# Les conditions d'utilisation

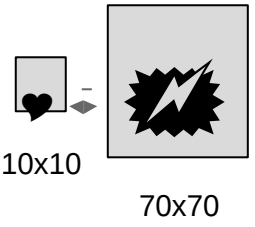
- **On définit plusieurs conditions d'utilisation**
  - ***Pire cas* (worst case)**
  - ***Meilleur cas* (best case)**
  - ***Typique* (typical)**
- **On peut demander à l'analyseur :**
  - ***De travailler en pire cas, meilleur cas ou typique***
  - ***De travailler en pire cas – meilleur cas***
    - **Selon la caractéristique mesurée il applique l'un ou l'autre**
    - **Exemple : pire cas pour les temps de *setup* et meilleur cas pour les temps de *hold***



# Le modèle prédictif de capacité et de résistance de routage

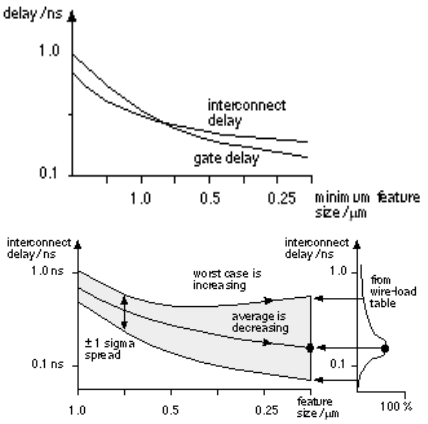
- **Sert à estimer l'impact du routage sur les performances**
- **Est fourni par le fondeur**
- **Dépend d'un facteur de complexité**
  - ***Taille du bloc***
  - ***Facteur de forme***



# Les délais d'interconnexion

- Les délais dus aux interconnexions sont prépondérants. Leur estimation est donc critique.
- Le pire cas augmente car la taille des circuits ne diminue pas.






# Sans placement routage

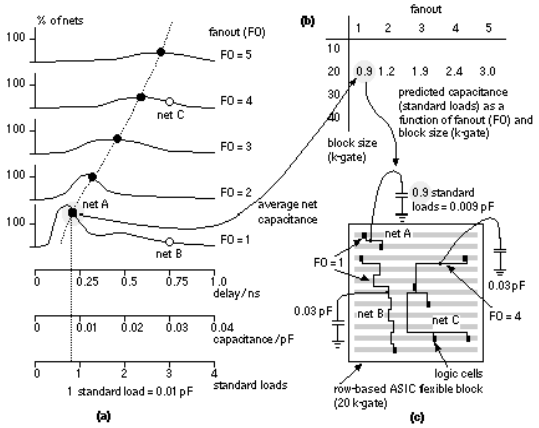
- On connaît pour chaque nœud
  - *Le nombre de cellules qu'il attaque (*fanout*)*
  - *Eventuellement la taille du bloc dans lequel il sera placé-routé*
- On ne peut pas estimer la topologie du nœud (donc on ne peut pas estimer le réseau RC) mais on peut estimer sa longueur, donc sa capacité.
- Les estimations de capacité sont issues de statistiques sur des circuits existants.
- A partir des statistique on crée des tables de capacités prédites en fonction du *fanout* et de la taille du bloc.

# On sait par expérience

- **Que 60 à 70 % des nœuds ont un *fanout* de 1**
  - ***Qui ont des capacités très variables***
  - ***La distribution de leur capacité présente deux pics (local, global)***
- **Les distributions des autres nœuds sont plus symétriques et plus « plates »**
- **Les statistiques changent beaucoup en fonction de la taille des blocs. On a besoin de beaucoup d'entrées dans la table.**



# Statistiques

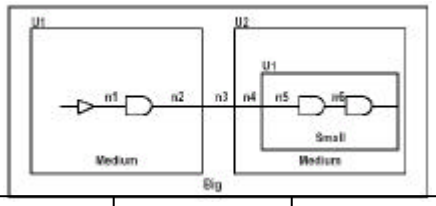


# Le modèle prédictif de capacité et de résistance de routage

- **Trois modes de calcul**
  - ***Plus haut niveau : on applique à tous les nœuds le modèle associé au plus haut niveau de la hiérarchie***
  - ***Contenu : on applique à un nœuds le modèle associé au niveau de hiérarchie qui le contient entièrement***
  - ***Segmenté : les nœuds sont segmentés et on leur applique la somme des modèles des niveaux hiérarchiques traversés***
- **Le mode « contenu » est en général le plus précis**
- **Les fondeurs préconisent en général un mode**



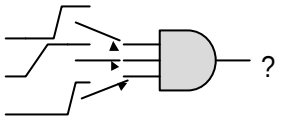
# Exemple




| Mode | Modèle | S'applique à |
|---|---|---|
| Plus haut niveau (*Top*) | Big | Tous les nœuds |
| Contenu (*Enclosed*) | Big | U1/n2, n3, U2/n4, U2/U1/n5 |
| | Medium | U1/n1 |
| | Small | U2/U1/n6 |
| Segmenté (*Segmented*) | Big | n3 |
| | Medium | U1/n1, U1/n2, U2/n4 |
| | Small | U2/U1/n5, U2/U1/n6 |

# La propagation des pentes

- Il existe plusieurs façons de propager les pentes :
  - *Sélection de la pente du signal le plus lent*
  - *Sélection de la plus longue pente*
- La première méthode est plus précise mais parfois optimiste
- La deuxième méthode est toujours pessimiste






# Plan

- Introduction
- Les modèles temporels
- La définition d'environnement
- L'analyse
- La diaphonie
- Conclusion

# Vérifications de règles de conception

- **Chemins non contraints**
- **Entrées horloge d'éléments de mémorisation non connectées à une horloge**
- **Cycles combinatoires**
- **Non recouvrement des horloges sur *latches* maître – esclave**
- ***Latches* sur niveaux qui attaquent des *latches* sur même niveau**



# Vérifications de règles de conception

- **Réduction de consommation et précision des modèles temporels :**
  - ***Limites maximales et minimales de capacités***
  - ***Limites maximales et minimales de temps de transition***
  - ***Nombres maximaux et minimaux de portes chargées* (fanout)**
- **Règles fournies par le fondeur pour sa bibliothèque de cellules**
- **Vérification des contraintes de l'utilisateur**

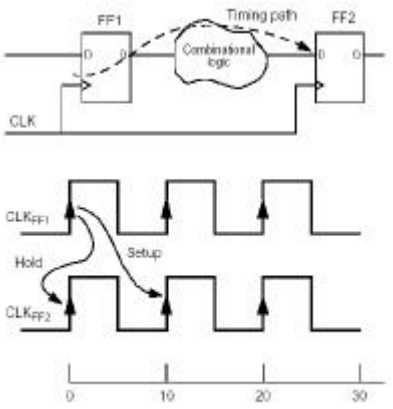
# La définition des exceptions

- **Concerne :**
  - ***Les faux chemins***
  - ***Les délais min et max de chemins***
  - ***Les chemins multi cycles***
- **Pour les spécifier il faut comprendre les algorithmes utilisés par l'analyseur**
  - ***Dans les configurations à une horloge***
  - ***Dans les configurations à plusieurs horloges***



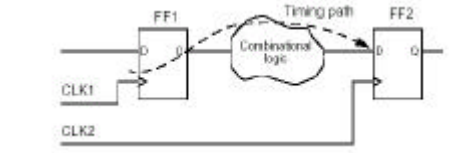
# Une seule horloge

- **L'analyseur vérifie**
  - ***Que les données arrivent à temps pour le deuxième front (*setup*)***
  - ***Que les données n'arrivent pas trop vite pour le premier front (*hold*)***

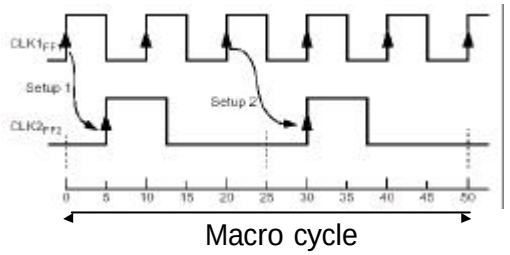


# *Setup* à deux horloges

- **Pour chaque front de l'horloge destination pendant un macro cycle l'analyseur teste le temps de *setup* par rapport au front précédent de l'horloge source**



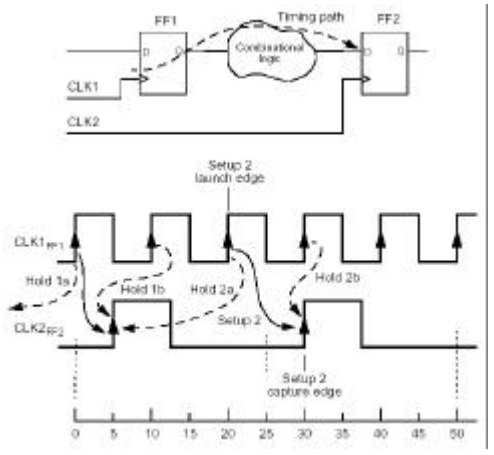

```
create_clock -period 10 -waveform {0 5} CLK1
create_clock -period 25 -waveform {5 12.5} CLK2
```






# *Hold* à deux horloges

- **Pour chaque front de l'horloge destination pendant un macro cycle l'analyseur test le temps de *hold* par rapport au front précédent et au front suivant de l'horloge source**

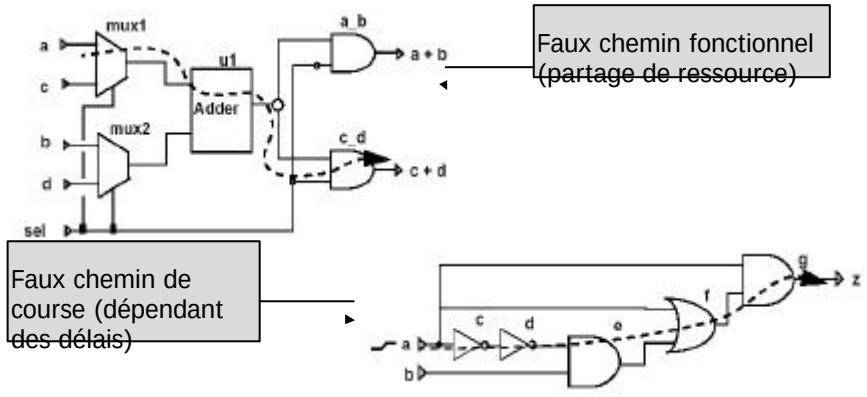


# Les faux chemins

- **Ce sont des chemins que l'analyseur considère mais dont le concepteur sait qu'ils ne sont pas fonctionnels**
  - ***Modes de fonctionnement exclusifs (test)***
  - ***Domaines d'horloges non synchrones***
- **Il faut les déclarer à l'analyseur**
  - ***Soit en spécifiant un point de départ A et un point d'arrivée B. Tous les chemins de A à B sont alors ignorés.***
  - ***Soit en spécifiant une horloge source S et une horloge destination D. Tous les chemins de S à l'entrée d'un registre commandé par D sont alors ignorés.***



# Les faux chemins

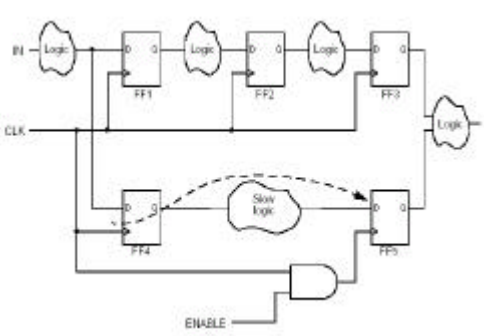


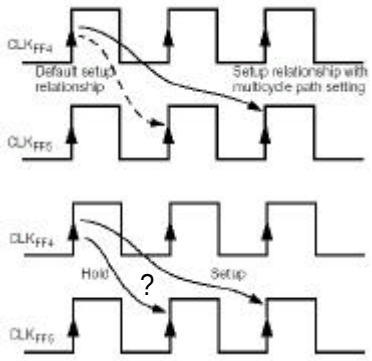
# Les délais min et max de chemins

- **On peut forcer les délais le long d'un chemin entre deux registres A et B**
  - ***L'analyseur ignore les horloges de A et B***
  - ***Il dénonce comme violations temporelles les délais qui ne respectent pas ces spécifications***
- **On peut particulariser ces délais pour une transition montante ou descendante au point d'arrivée**



# Les chemins multi cycles




```
set_multicycle_path -setup 2 -from FF4 -to FF5
```

# Analyse hiérarchique

- **Modélisation par la logique d'interface**
  - ***On extrait d'un module son interface. C'est elle qui constitue le modèle temporel abstrait.***
- **Langage de modélisation**
  - ***On écrit un modèle temporel du module dans un langage dédié***
    - **Approximatif (phase d'exploration d'architecture)**
    - **Précis (phase de conception fine)**
- **Extraction de modèle**
  - ***On utilise un outil d'extraction qui construit un modèle temporel à partir de la* netlist**



# Plan

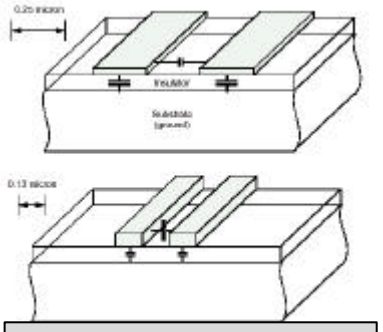
- **Introduction**
- **Les modèles temporels**
- **La définition d'environnement**
- **L'analyse**
- **La diaphonie**
- **Conclusion**

# Origine et effets

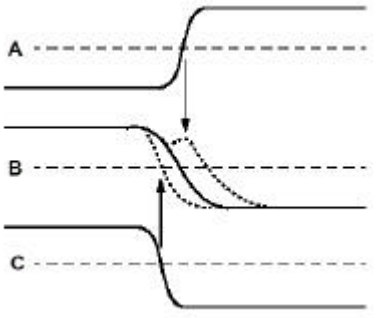
- **Les capacités de couplage entre équipotentielles sont de plus en plus importantes**
  - ***Les capacités diminuent Þ les capacités de couplage augmentent en proportion***
  - ***Les couches de métal sont plus nombreuses Þ on est plus loin du substrat Þ les capacités de couplage à la masse diminuent***
- **La diaphonie influe sur la vitesse de transition, voire sur la fonctionnalité**



# Origine et effets



La diaphonie est de plus en plus importante



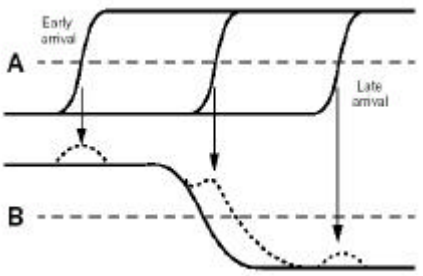

Les effets de la diaphonie

# Victimes et agresseurs

- On parle de nœud *victime* et de nœud *agresseur*
- Les effets dépendent
  - *De la valeur de la capacité de couplage*
  - *Des pentes et dates de transition*
  - *Du sens des transitions*
  - *Du nombre de victimes et agresseurs impliqués*

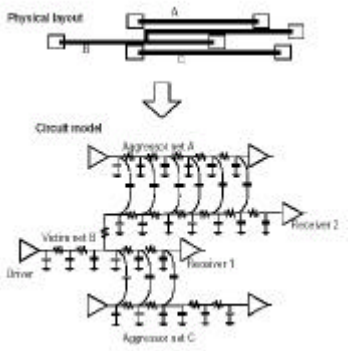


# Selon les situations on peut ignorer ou pas le phénomène



# Modélisation

- Les modèles physiques sont complexes
  - *On les simplifie*
- La présence de diaphonie « gênante » est déterminée par un calcul de fenêtres temporelles glissantes
- La prise en compte de la diaphonie va devenir incontournable






# Plan

- Introduction
- Les modèles temporels
- La définition d'environnement
- L'analyse
- La diaphonie
- Conclusion

## Comment choisir un analyseur ?

- Même moteur d'analyse et modèles que le synthétiseur logique
- Gestion des conversions « résultats d'analyse » Û « environnement d'analyse »
- Approche hiérarchique
  - *Langage de modélisation*
- Langage de commande ouvert (tcl)
- Capacité en nombre de portes
- Quantité de mémoire utilisée
- Présentation et exploration des résultats



## Comment choisir un analyseur ?

- Caractéristiques fonctionnelles
  - *Horloges multiples*
  - *Chemins multi-cycles*
  - Latches *transparents*
  - *Diaphonie*
  - *Logique sur horloges, etc.*
- Formats d'entrée
  - *VHDL, EDIF, Verilog*
  - *SDF, RSPF, DSPF, SPEF, …*
- Interfaçage avec les outils de conception physique

# Synthèse



## Synthèse de matériel

Frédéric Pétrot
Département ASIM du LIP6
Université Pierre et Marie Curie

Frederic.Petrot@lip6.fr

# Plan du cours

- Introduction
- Langages de description de matériel
- Niveaux de synthèse
- Synthèse au niveau transfert de registre
- Synthèse séquentielle
- Synthèse de haut niveau





# Introduction

Les différents niveaux de conception

**Circuit** :
population de transistors

**Structurel** :
ressources de base (portes, bascules)

**Micro-architectural** :
ressources évoluées (registres, banc de registres, opérateurs, boîtes à opérations, machines à états, ...)

**Système** :
processeur, mémoires, contrôleurs, ...

## Vues d'un circuit

Niveaux d'abstraction ou encore *Vues* d'un circuit

**Physique (*layout*)** :
information géométrique *comment est-il réalisé ?*

**Structurelle (*netlist*)** :
information schématique *comment ses constituants sont-ils liés ?*

**Comportementale (*behavior*)** :
information fonctionnelle *que fait-il ?*

| | |
|---|---|
| Nombre d'éléments: | Physique $>$ Structurelle $>$ Comportementale |
| Sémantique | Physique $<$ Structurelle $<$ Comportementale |

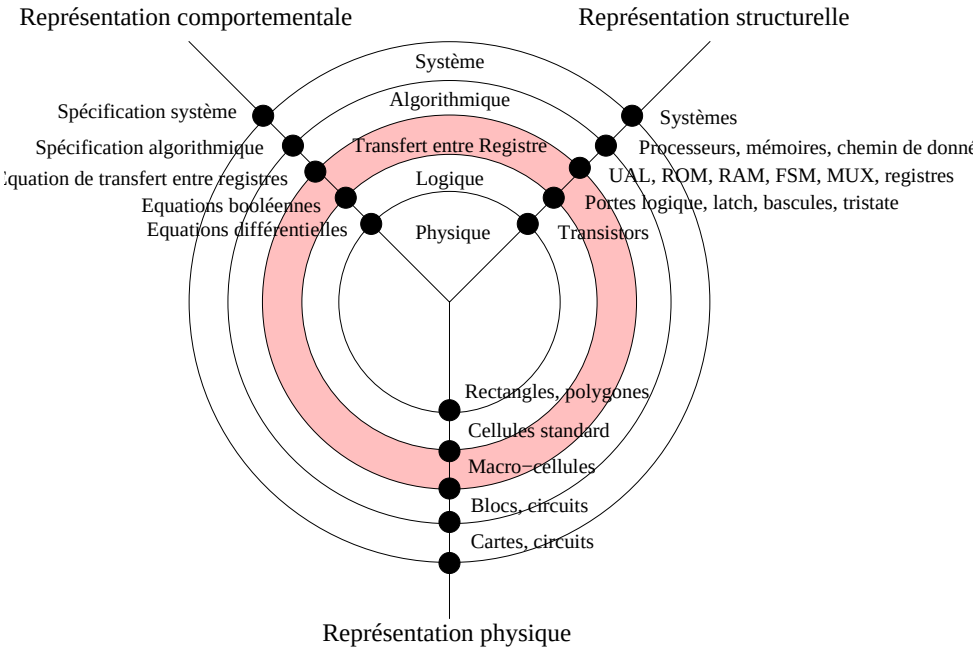




## Passage d'une vue à une autre

- On sait faire, en une étape :
  1. Structurelle $\Rightarrow$ Physique : floorplanning, placement, routage, générateur de blocs
  2. Physique $\Rightarrow$ Structurelle : extraction
  3. Comportementale $\Rightarrow$ Structurelle : **synthèse**
  4. Structurelle $\Rightarrow$ Comportementale : abstraction fonctionnelle
- Le passage n'est pas possible à tous les niveaux de conception
- Les niveaux de conception n'ont pas des axes gradués de façon identique

# Diagramme en Y de Gajski




**Synthèse :** *transition du domaine comportemental vers le domaine structurel, ou passage d'un niveau d'abstraction à un autre sur l'axe comportemental*





# Plan du cours

- Introduction
- Langages de description de matériel
- Niveaux de synthèse
- Synthèse au niveau transfert de registre
- Synthèse séquentielle
- Synthèse de haut niveau

## Langages de description de matériel

But

- spécification *exécutable* d'un circuit
  pour simuler
  pour synthétiser
  pour vérifier formellement
- un seul fichier de la spécification à la réalisation physique

Contraintes

- standards : développement de matériel coûteux/long
  $\Rightarrow$ on cherche à pourvoir partager/acheter des choses existantes
- exécutable rapidement : la définition et mise au point des spécifications est l'étape la plus longue





## Pourquoi des langages spécifiques ?

les langages de programmation (C, C++) :

- ont des type complexes, mais abstraits :
  - le nombre de bit est implicite, et varie selon les architectures
  - la longueur est un multiple du *byte*
  - on ne sait pas exprimer un morceau de vecteur « naturellement »

## Pourquoi des langages spécifiques ?

Les langages de programmation (C, C++) :

- sont intrinsèquement séquentiels :
  - on ne sait pas exprimer aisément le parallélisme du matériel, ni la synchronisation entre processus
  - ils sont utilisés en forte intéraction avec un système (Unix, ...)
  - difficile de spécifier clairement des mécanismes de gestion d'exception (parallélisme implicite)
  - notable exception: ADA
- ne permettent pas de décrire des schémas d'interconnexions

$\Rightarrow$ Langages non adaptés à la représentation du matériel
$\Rightarrow$ Langages adaptés aux spécifications fonctionnelles





## Exemple de VHDL haut niveau

```
package mypack is
   subtype bit8 is integer range 0 to 255
end mypack

use work.mypack.all;

entity diffeq is
port(dx_p, a_p, x_p, u_p : in bit8;
     y_p                 : inout bit8;
     ck, go              : in bit);
end diffeq;
```

```
architecture behavior of diffeq is
begin
   process
      variable x, y, a, u, dx, xl, yl, ul : bit8;
      begin
         wait until go = '1' and go'event;
         x := x_p; y := y_p; a := a_p; u := u_p; dx := dx_p;
         while (x < a) loop
            wait until ck = '1' and ck'event;
            xl := x + dx;
            ul := u - (3 * x * u * dx) - (3 * y * dx);
            yl := y + (u * dx);
            x := xl; u := ul; y := yl;
         end loop;
         y_p <= y;
   end process;
end behavior;
```

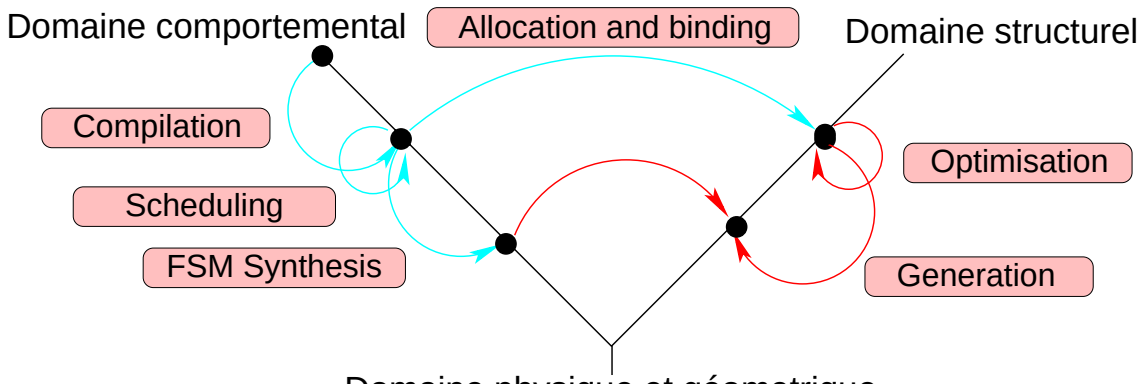




# Plan du cours

- Introduction
- Langages de description de matériel
- Niveaux de synthèse
- Synthèse au niveau transfert de registre
- Synthèse séquentielle
- Synthèse de haut niveau

# Synthèse de haut niveau





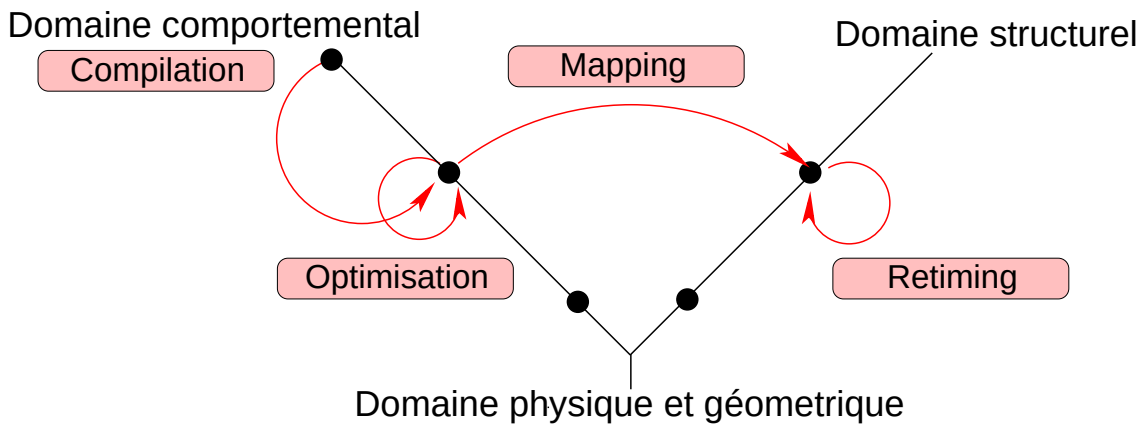



# Synthèse de haut niveau

Synthèse d'architecture, synthèse comportementale, synthèse de haut niveau

- entrée :
  description algorithmique, sans information architecturale ni temporelle
- résultat :
  interconnexion de mémoires, chemins de données, automates de contrôle, au niveau RTL
- méthode :
  - définition des ressources matérielles nécessaires à l'implantation de l'algorithme
  - choix d'affectation d'une opération sur une ressource
  - construction d'un séquenceur capable de piloter ces ressources

# Synthèse RTL








# Synthèse RTL

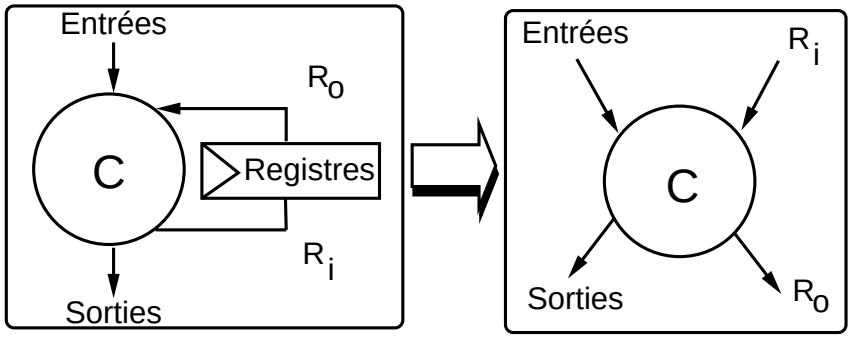
Synthèse niveau transfert entre registres, synthèse RTL

- entrée :
  description d'un comportement cycle par cycle, ensemble d'assignations concurentes et processus séquentiels communiquants
- résultat :
  ensemble d'équations booléennes, assignées de façon concurente, séparées par des registres clairement identifiés
- méthode :
  - identification des registres
  - partage d'opérateurs
  - déplacement de registres pour réduire des chemins temporels

  sans modification du comportement cycle par cycle

# Synthèse logique

Synthèse logique, optimisation combinatoire




- seules les parties combinatoires sont concernées, mais c'est une étape très importante de la réalisation
- étape ayant amené l'acceptation de l'automatisation





# Synthèse logique

Synthèse logique, optimisation combinatoire

- entrée :
  description d'un comportement par des équations booléennes concurentes, ou les registres ont été identifiés
- résultat :
  ensemble d'équations booléennes concurrentes optimisées
- méthode :
  - ▶ simplification des expressions booléennes
  - ▶ factorisation de parties d'équations

## Projection structurelle

Projection structurelle

- entrée :
  ensemble d'équations concurrentes optimisées.
  - Projection structurelle sur une technologie cible (deux niveaux ou multi-niveaux)
  - Utilisation d'une bibliothèque de composants élémentaires
- Il n'y a que les parties combinatoires qui sont concernées, mais c'est une étape très importante de la réalisation
- résultat :
  réseau de portes logiques existantes physiquement interconnectées



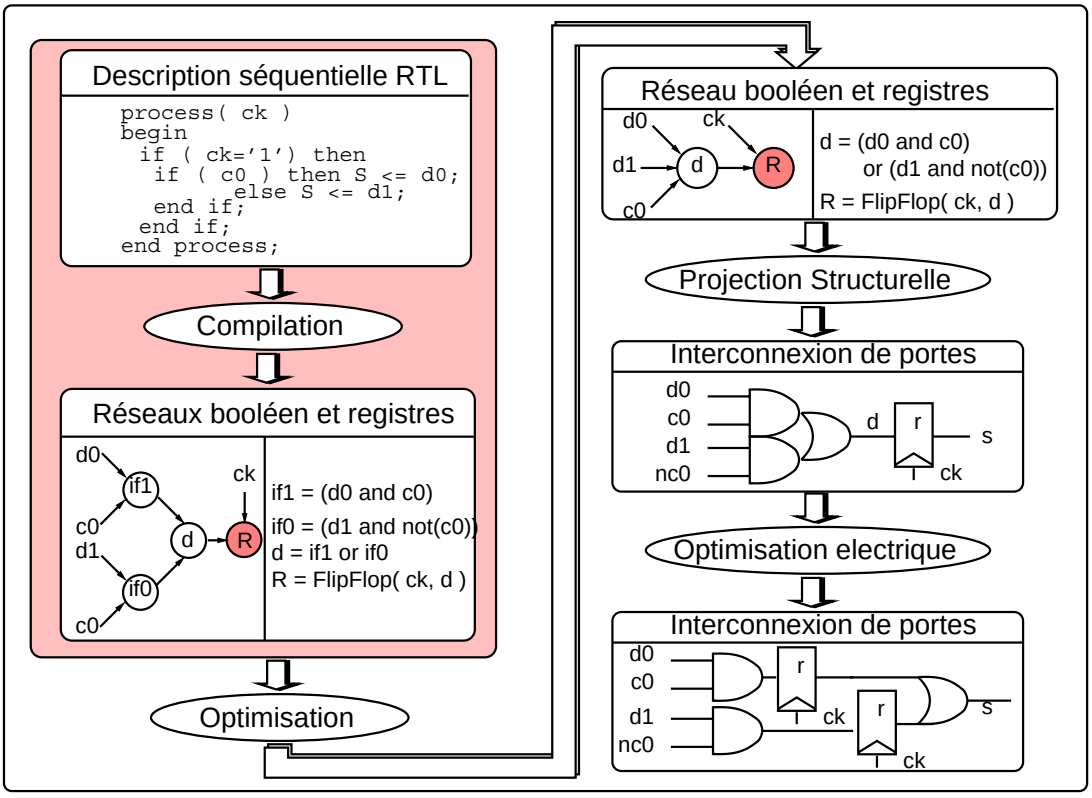


## Optimisation électrique

Optimisation temporelle/électrique des chemins, retiming

- entrée :
  réseau de portes logiques existantes interconnectées
- résultat :
  réseau de portes logiques interconnectées
- méthode :
  - Adaptation des sortances
  - Équilibrage des temps de propagation des chemins
    - Par ajout de buffers
    - Par déplacement des registres
  - Utilisation des caractéristiques temporelles fines des cellules

# Résumé sur la synthèse RTL








# Plan du cours

- Introduction
- Langages de description de matériel
- Niveaux de synthèse
- Synthèse au niveau transfert de registres
- Synthèse séquentielle
- Synthèse de haut niveau

## Reconnaissance des éléments mémorisants

- Par utilisation de gabarits prédéfinis dans les outils industriels:

```
process (ck, reset)
begin
   if reset then
      q <= 0;
   elsif ck and ck'event then
      if we = '1' then
         q <= d;
      end if;
   end if;
end process;
```

- Par identification formelle dans des expérimentations universitaires (visant une « vraie » portabilité du VHDL): prise en compte de la liste de sensibilité du processus, analyse des chemins d'assignation des variables, ...





## Reconnaissance des éléments mémorisants

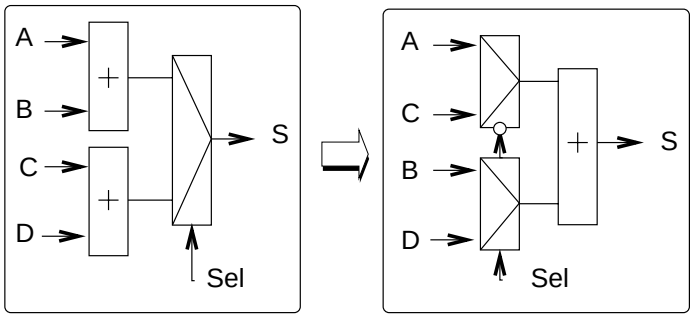
```
process (ck)
begin
  if ck = '0' then
    q <= d;
  end if;
end process;
```

```
process (ck)
variable v : std_logic;
begin
  if ck = '0' then
    v := d;
  else
    v := q;
  end if;
  q <= v;
end process;
```

```
process (ck)
variable b : boolean;
begin
  b := ck='0' and ck'event
  if b then
    q <= d;
  end if;
end process;
```

# Partage des opérateurs complexes

```
process(sel, a , b, c, d)
begin
  if sel = '0' then
    s <= a + b;
  else
    s <= c + d;
  end if;
end process;
```




- simple car le comportement est donné cycle par cycle





# Équations booléennes

Les fonctions booléennes sont représentées par des **équations**, définies grâce à :

- des parenthèses
- des litéraux : $x, y, z, \overline{x}, \overline{y}, \overline{z}$ où $(x, y, z, ...) \in \mathcal{B}^k$ avec $\mathcal{B} = \{0, 1\}$
- des opérateurs booléens : ou $+$, et $\times$, ou-exclusif, ...
- la complémentation : $\overline{x} + y$, $\overline{x + y}$

Exemples:

$$f = x_1 \times \overline{x_2} + \overline{x_1} \times x_2 = (x_1 + x_2) \times (\overline{x_1} + \overline{x_2})$$

On remplace usuellement le $\times$ par la concaténation : $a \times b \rightarrow ab$

## Fonctions logiques

- il y a $2^n$ points (nœuds) dans l'espace d'entrée $\mathcal{B}^n$
  Ces points sont nommés *minterms*, et chacun représente une conjonction, parmis les $2^n$, de littéraux
- il y a $2^{2^n}$ fonctions logiques distinctes
  Chaque sous ensemble possible de $\mathcal{B}^n$ représente une fonction distincte
- il y a un nombre infini d'équations booléennes pour une fonction logique donnée :

  $$f = x + y = x\overline{y} + xy + \overline{x}\,y = x\overline{x} + x\overline{y} + y = (x+y)(x+\overline{y}\,) + \overline{x}\,y = \ldots$$

- Optimisation booléenne : trouver la formule qui a le moins de littéraux





## Minimisation « 2-niveaux »

- Somme de produits (de cubes) :

  $$f = \overline{a}\,\overline{b}\,\overline{c}\,d + \overline{a}\,\overline{b}\,cd + \overline{a}\,b\overline{c}\,d + a\overline{b}\,\overline{c}\,d + a\overline{b}\,cd + ab\overline{c}\,d + abc\overline{d} + abcd$$

- minimisation « 2-niveaux » : minimiser le nombre de cubes
  - important historiquement : PLA
  - important théoriquement : solution optimale trouvable
- méthode de Quine/Mac Cluskey (1956): sorte de table de Karnaugh généralisées
  - méthode énumérative
  - façilement implémentable informatiquement
  - permet de trouver la solution optimale

## Minimisation « 2-niveaux »

- méthode de Quine/Mac Cluskey vue d'avion
  1. on calcul les *primes* de $\{x|f(x) = 1 \text{ ou } *\}$, (c.-à-d. les groupes maximaux de la table de Karnaugh)
  2. on recouvre les *minterms* de la fonction avec un sous ensemble des *primes*
  3. complexité de la méthode
     - jusqu'à $2^n$ *minterms* dans une fonction à $n$ entrées
     - jusqu'à $3^n/n$ cubes premiers dans une fonction à $n$ entrées
     - et le problème de couverture minimal est NP complet (pas d'algorithme polynomial pour le résoudre exactement)
- améliorations : la meilleur méthode connue a été découverte par un docteur de l'ENST !





## Synthèse multi-niveaux

Expression multi-niveaux

- pas de contrainte sur l'imbrication des opérateurs

Représentation par un réseau booléen :

- graphe dirigé acyclique
- nœud : expression booléenne
- arc : dépendance de donnée

Cibles

- bibliothèques de cellules précaractérisées
- FPGA
- bref, ce qui se fait aujourd'hui !

## Approche heuristique

Application de *transformations* logiques qui permettent de modifier le réseau booléen à comportement constant :

- transformations locales : la structure du graphe reste identique, on ne change que l'« intérieur » des nœuds
- transformations globales : elles agissent sur la structure du graphe, en ajoutant ou otant des nœuds, des arcs, etc.

Illustration sur l'exemple suivant :

$$
\begin{array}{lcl@{\qquad}lcl}
p & = & ce + de & v & = & \overline{a}\,d + bd + \overline{c}\,d + a\overline{e} \\
q & = & a + b & w & = & v \\
r & = & p + \overline{a} & x & = & s \\
s & = & r + \overline{b} & y & = & y \\
t & = & ac + ad + bc + bd + e & z & = & u \\
u & = & \overline{q}\,c + q\overline{c} + qc & & &
\end{array}
$$





## Élimination

- l'élimination d'un nœud interne consiste à le faire disparaitre du réseau
- toute occurence de la variable est remplacée par l'expression correspondante

Exemple: élimination de $r$ implique $s = p + \overline{a} + \overline{b}$

## Décomposition

- la décomposition d'un nœud interne consiste en son remplacement par 1 ou plusieurs nœuds formant un sous-réseau équivalent

Exemple: reécriture de $v = \overline{a}\,d + bd + \overline{c}\,d + a\overline{e}$ en $v = (\overline{a} + b + \overline{c}\,)d + a\overline{e}$ puis décomposition en

$$\begin{aligned} j &= \overline{a} + b + \overline{c} \\ v &= jd + a\overline{e} \end{aligned}$$





## Extraction

- l'extraction d'une sous-expression commune à deux fonctions consiste à créer un nouveau nœud associé à cette sous-expression.

Exemple: $p = ce + de$ et $t = ac + ad + bc + bd + e$ peuvent être reécrit comme :

$$\begin{aligned} p &= (c + d)e \\ t &= (c + d)(a + b) + e \end{aligned}$$

En notant $k = c + d$, on a $p = ke$ et $t = ka + kb + e$

On sent bien que cette « mise en facteur commun » peut être très intéressante du point de vue de la minimisation des fonctions

## Simplification

- il s'agit ici d'optimiser localement la fonction, en utilisant des techniques similaires à celles de la logique 2 niveaux
  Si le nombre de variables est petit, il existe des approches qui sont très bonnes
  Cette transformation est locale

Exemple: $u = \overline{q}\,c + q\overline{c} + qc$ donne $u = q + c$





## Substitution

- on tente de réduire la complexité d'une fonction en utilisant une entrée qui n'apparaissait pas dans son support
  On ajoute une dépendance, mais dans l'espoir d'en éliminer d'autres

Exemple: $t = ka + kb + e$ peut être simplifiée si on voit que $q = a + b$
Alors, $t = qk + e$

## Comment réaliser ces transformations ?

Le modèle algébrique (Brayton)

- l'idée est de réduire l'espace de recherche dans les expressions en considérant un modèle simplifié
  - une expression booléenne est considérée comme un polynome multilinéaire avec des variables de coefficient 1
  - les règles de simplification n'existent plus
  - la complémentation n'existe plus : une variable et son complément son deux variables différentes
  - la notion de *don't care* n'existe plus
  - on considère localement des sommes de produits
- Définitions : $f_{diviseur}$ est un diviseur algébrique de $f_{dividende}$ lorsque $f_{dividende} = f_{diviseur} \times f_{quotient} + f_{reste}$, avec $f_{diviseur} \times f_{quotient} \neq 0$, et le support de ces 2 fonctions est disjoint





## Division algébrique

On considère les expressions $A$ et $B$ comme des ensembles d'éléments (de cubes) :
Soit $A = \{c_j^a, j = 1, 2, \ldots, l\}$, le dividende
soit $B = \{c_i^b, i = 1, 2, \ldots, n\}$ le diviseur

- le quotien $Q$ et le reste $R$ sont la somme des cubes des ensembles résultats de l'opération
- en triant les cubes, l'agorithme de division peut avoir un temps en $O(n)$ ou $O(n \log n)$

# Division algébrique

```
division(A, B) {
   Q = ∅
   for (i = 1 to n)
      D = {c_j^a  tel que  c_j^a ⊇ c_i^b}
      if (D = 0)
         return (∅, A)
      D_i = D où les variables du support de c_i^b sont effacées
      Q = Q ∩ D_i
   }
   R = A − Q × B
   return (Q, R)
```





# Exemples

- Exemple 1
  - soit $f_{dividende} = ac + ad + bc + bd + e$ et $f_{diviseur} = a + b$
  - $X = \{ac, ad, bc, bd, e\}$ et $Y = \{a, b\}$
    - $i = 1$, d'ou $c_1^y = a$, $D = \{ac, ad\}$
    - $D_1 = \{c, d\}$ et $Q = \{c, d\}$
    - $i = 2$, d'ou $c_2^y = b$, $D = \{bc, bd\}$
    - $D_2 = \{c, d\}$ et $Q = \{c, d\} \cap \{c, d\} = \{c, d\}$
    - et finalement $Q = \{c, d\}$ et $R = e$, et on conclu $f_{quotient} = c + d$ et $f_{reste} = e$

# Exemples

- Exemple 2
  - soit $f_{dividende} = axc + axd + bc + bxd + e$ et $f_{diviseur} = ax + b$
  - $X = \{axc, axd, bc, bxd, e\}$ et $Y = \{ax, b\}$
    - $i = 1$, d'ou $c_1^y = ax$, $D = \{axc, axd\}$
    - $D_1 = \{c, d\}$ et $Q = \{c, d\}$
    - $i = 2$, d'ou $c_2^y = b$, $D = \{bc, bxd\}$
    - $D_2 = \{c, xd\}$ et $Q = \{c, d\} \cap \{c, xd\} = \{c\}$
    - et finalement $Q = \{c\}$ et $R = \{axd, bxd, e\}$, et on conclu $f_{quotient} = c$ et $f_{reste} = axd + bxd + e$





# Projection structurelle

# Optimisation électrique

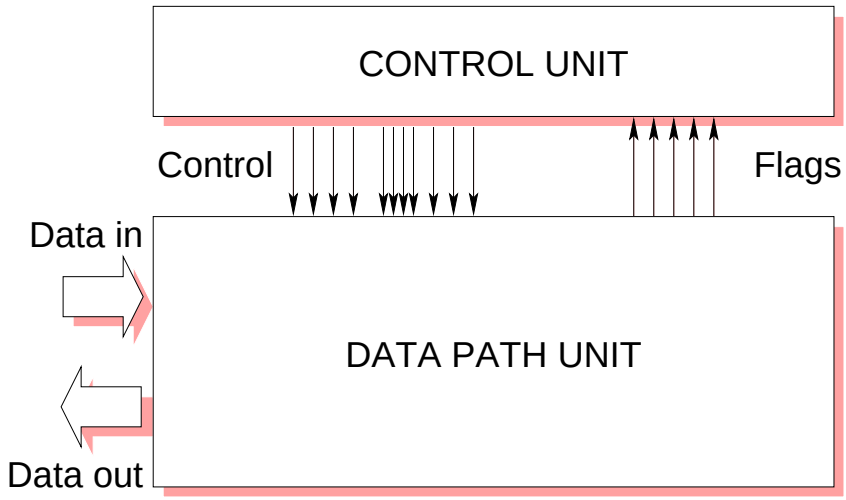




# Synthèse d'architecture

Objectif : Créer du matériel à partir d'une description sequentielle
Pourquoi :

- impact fondamental des décisions architecturales
  $\Rightarrow$ il faut pouvoir les valider, en grand nombre, rapidement



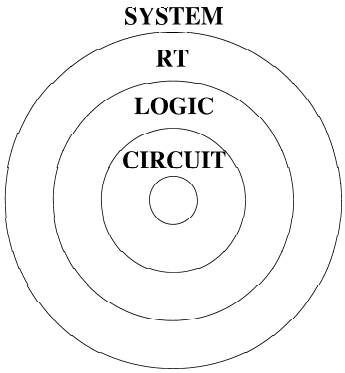
# Partie I

# INTRODUCTION





## NIVEAUX DE CONCEPTION
## NIVEAU D'ABSTRACTION




**CIRCUIT** : transistors

**LOGIC** : ressources de base
(portes, bascules)

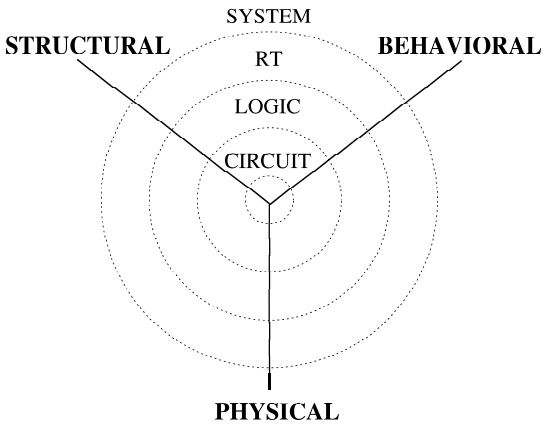
**RT** micro-arch. : ressources évoluées
(registres, ALUs, FSMs)

**SYSTEM** : controleurs, processeurs, RAM





## DEGRE D'ABSTRACTION




**PHYSIQUE** : topologie
(où)

**STRUCTUREL** : contenu, schématique
(comment il est fait)

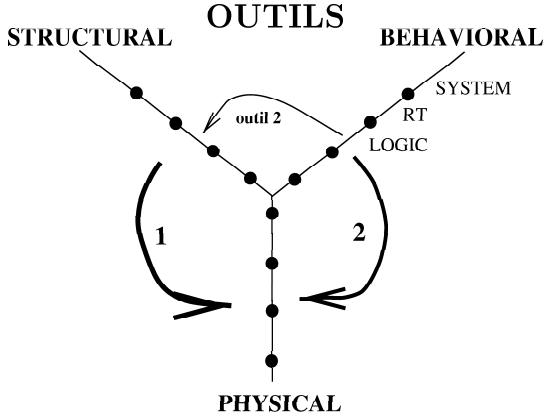
***COMPORTEMENTAL*** : fonction
(qu'est ce qu'il fait)

physique ⇒ structurel ⇒ comportemental





## OUTILS




**1** floor-planning, placement,
routage, générateurs de blocs

**2** synthèse logique, d'automate

conception de circuit est faite au niveau structurel pour le système et RT, au niveau comportemental pour le logique.

# BEMOL

**structurel ⇒ physique**

**floor-planning**
(manuel)

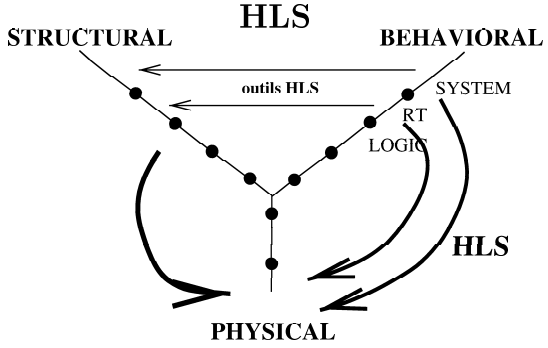
**placement,routage**
(paramètres, fin parfois manuelle)

**changement de bibliothèques**
(en particulier si course de signaux)





# HLS




# CRITERES D'UTILISABILITE

**pont réel vers le physique**
souvent: un bloc diagramme + FSM grossiers
contraintes connectiques/électriques/temporelles

**résultats pas trop éloignés d'une conception classique**

**exploration de l'espace de conception**
comportemental ⇒ structure : nb Possibilités
critères utilisateurs : petit, rapide, delays





# HLS AVANTAGES

Raccourcir le cycle de conception

Diminuer le nombre d'erreurs

Meilleure documentation

Exploration rapide de l'espace de conception

Reprise des CIs

Portabilité des conceptions

Démocratisation de la conception

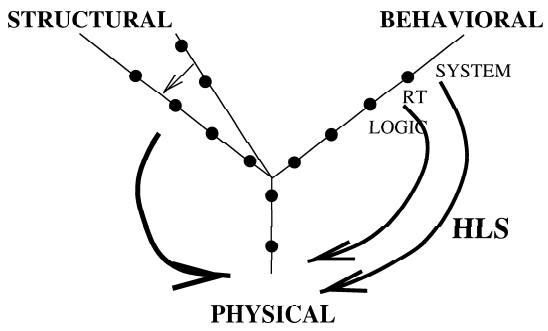




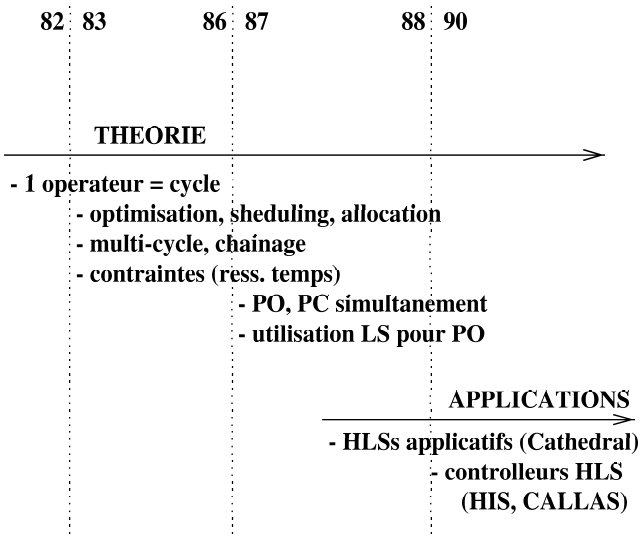
# Synthèse commerciale

ASIC SYNTHESIZER de COMPASS

L'OUTIL DE SYNOPSYS



## HISTORIQUE






# Partie II

# PRESENTATION



# Chapitre 1

# EXEMPLE



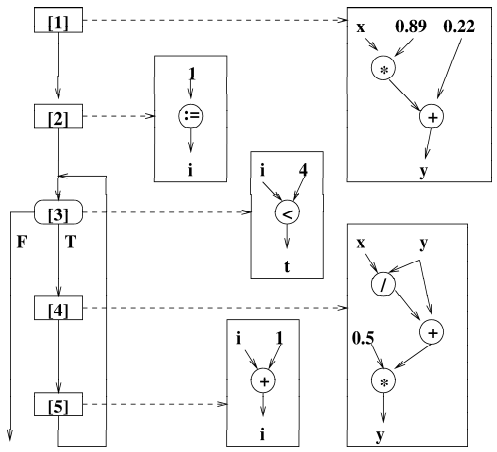


## COMPILATION

[1] y := 0.22 + 0.89 * x; [2] i :=0; [3] while ( i ¡ 4 ) do [4] y := 0.5 * (y + x/y); [5] i := i + 1; done

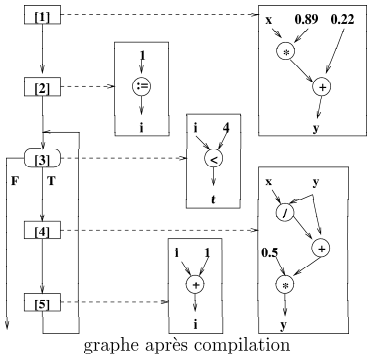
```
[1] y := 0.22 + 0.89 * x;
[2] i :=0;
[3] while ( i < 4 ) do
[4]       y := 0.5 * (y + x/y);
[5]       i := i + 1;
      done
```

**Description Comportementale**

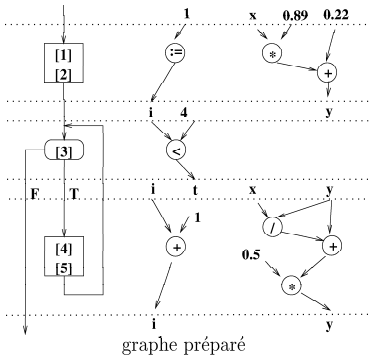



**Graphe après compilation**

# PREPARATION




graphe après compilation




**CFG** graphe préparé **DFG**





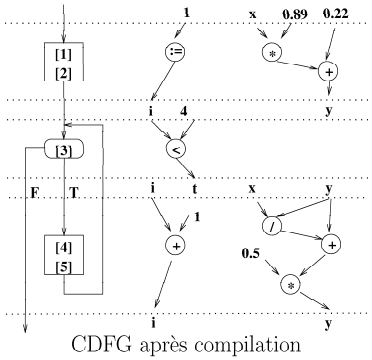
# OPTIMISATION

**BUT:** se rapprocher du physique

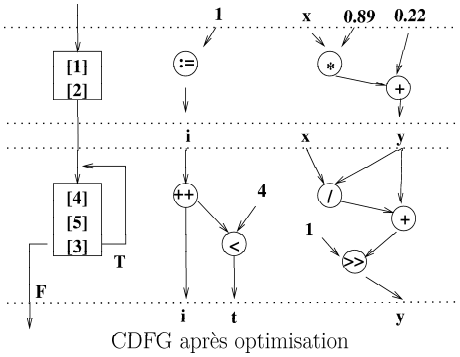
- **Opérateurs matériels**
  - opération ⇒ opérateur physique
    ($+1 \rightarrow ++$, $*0.5 \rightarrow >>$)
- **Simplification des DFGs**
  - propagation des constantes
  - simplification d'expressions
- **Modification du CFG**
  - élagage
  - déplacement de noeuds








CDFG après compilation




CDFG après optimisation





# HLSM

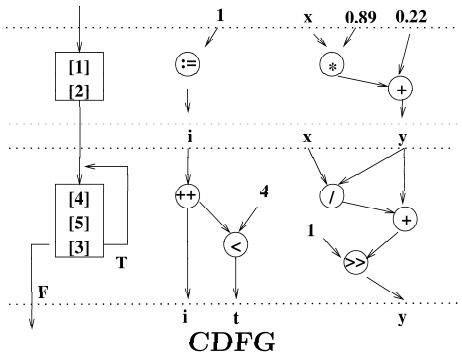
**H**igh **L**evel **S**tate **M**achine

| current state | cond | next state | fonction |
|---|---|---|---|
| BB0 | ... | BB1 | ... |
| BB1 | $x == 0$ | $BB_i$ | ... |
| | $x! = 0$ | $BB_j$ | ... |
| ... | ... | ... | ... |

**Pont entre comportemental et RT**

- FSM où l'état est un Basic Bloc
- Basic Bloc: une fonction décrite en data-flow
- Basic Bloc: durée de 1 à N cycles
- squelette de l'automate final

**HLSM : Forme cannonique de la synthèse**




***CDFG***

| current state | cond | next state | fonction |
|---|---|---|---|
| BB0 | ... | BB1 | ... |
| BB1 | | BB2 | $y <= 0.22 + 0.89 * x$<br>$i <= 1$ |
| BB2 | t=true | BB2 | $i <= incr(i)$<br>$t <= (incr(i) < 4)$<br>$y <= shr(y + x/y, 1)$ |
| | t=false | BB3 | |
| BB3 | ... | ... | ... |

**HLSM**





# SCHEDULING

**BUT:** Ordonnancer au niveau du cycle les BBs du HLSM

Allocation temporelle des opérateurs

**1 BB du HLSM** +

| op | time | area |
|---|---|---|
| ++ | 30 | 1 |
| + | 40 | 2 |
| < | 40 | 2 |
| * | 120 | 8 |
| / | 120 | 8 |
| >> | 0 | 0 |

↓

l'automate de contrôle du BB

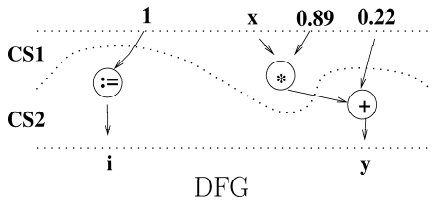
| cs | opérateurs |
|---|---|
| 1 | * * + ++ |
| 2 | / − |
| 3 | + |

- le plus rapide possible (↕ minimiser le nb de CS)
- le plus petit possible (↔ minimiser le nb d'opérateur)





## Scheduling du Basic Bloc BB1




DFG

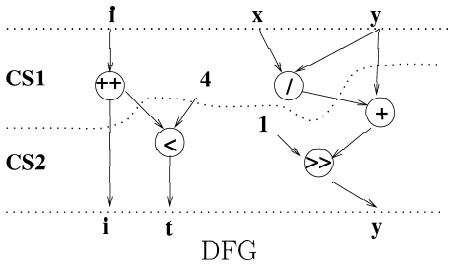
| CS | opérateurs |
|---|---|
| CS1 | * |
| CS2 | := + |

Table d'allocation temporelle des opérateurs





## Scheduling du Basic Bloc BB2




DFG

| CS | opérateurs |
|---|---|
| CS1 | ++ / |
| CS2 | < + >> |

Table d'allocation temporelle des opérateurs

## les BBs schedulés ⇒ FSM

BB1

| CS | opérateurs |
|---|---|
| CS1 | $*$ |
| CS2 | $:=$ $+$ |

| CS | opérateurs |
|---|---|
| CS1 | $++$ $/$ |
| CS2 | $<$ $+$ $>>$ |

BB2

| current | cond | next | fonction |
|---|---|---|---|
| BB1 | | BB2 | $y <= 0.22 + 0.89 * x$<br>$i <= 1$ |
| BB2 | t=true | BB2 | $i <= incr(i)$<br>$t <= (incr(i) < 4)$<br>$y <= shr(y + x/y, 1)$ |
| | t=false | BB3 | |

HLSM

| current | cond | next | fonction | op |
|---|---|---|---|---|
| BB1 CS1 | | BB1 CS2 | $tmp0 <= 0.89 * x$ | $*$ |
| BB1 CS2 | | BB2 CS1 | $y <= 0.22 + tmp0$<br>$i <= 1$ | $:=, +$ |
| BB2 CS1 | | BB2 CS2 | $i <= incr(i)$<br>$tmp1 <= x/y$ | $++, /$ |
| BB2 CS2 | t=true | BB2 CS2 | $i <= incr(i)$<br>$t <= (i < 4)$<br>$y <= shr(y + tmp1, 1)$ | $<, +, >>$ |
| | t=false | BB3 CS1 | | |

FSMD





# ALLOCATION

**BUT:** Générer la partie opérative

### Opérations:

- déterminer le nombre de registres
- allouer les variables sur les registres
- déterminer le nombre d'unités fonctionnelles
- allouer les opérateurs sur les unités fonctionnelles
- créer les connexions entre les registres et les unités

### Contraintes:

- minimiser le nombre de registres
- minimiser le nombre d'unités fonctionnelles
- minimiser la connectique
- minimiser la surface totale

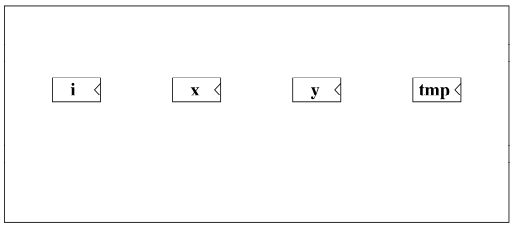




## 1) Allocation des registres

**Règle:** Toute variable écrite à un CS et lue dans un autre

| current | cond | next | fonction | op |
|---|---|---|---|---|
| BB1 CS1 | | BB1 CS2 | $tmp0 <= 0.89 * x$ | $*$ |
| BB1 CS2 | | BB2 CS1 | $y <= 0.22 + tmp0$<br>$i <= 1$ | $:=, +$ |
| BB2 CS1 | | BB2 CS2 | $i <= incr(i)$<br>$tmp1 <= x/y$ | $++, /$ |
| BB2 CS2 | t=true | BB2 CS2 | $i <= incr(i)$<br>$t <= (i < 4)$<br>$y <= shr(y + tmp1, 1)$ | $<, +, >>$ |
| | t=false | BB3 CS1 | | |




**candidates:** x, y, i, tmp0, tmp1
tmp0 et tmp1 ont des durées de vie disjointes

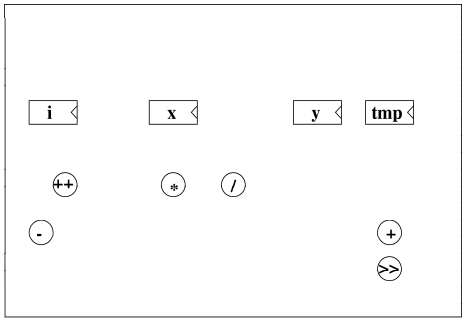




## 2) Allocation des unités fonctionnelles

**Règle:** Dans un CS, une unité fonctionelle pour chaque opération

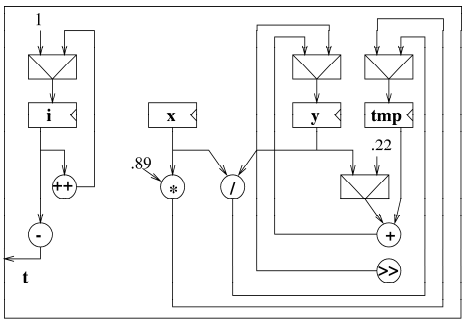
| current | cond | next | fonction | op |
|---|---|---|---|---|
| BB1 CS1 | | BB1 CS2 | $tmp0 <= 0.89 * x$ | $*$ |
| BB1 CS2 | | BB2 CS1 | $y <= 0.22 + tmp0$<br>$i <= 1$ | $:=, +$ |
| BB2 CS1 | | BB2 CS2 | $i <= incr(i)$<br>$tmp1 <= x/y$ | $++, /$ |
| BB2 CS2 | t=true | BB2 CS2 | $i <= incr(i)$<br>$t <= (i < 4)$<br>$y <= shr(y + tmp1, 1)$ | $<, +, >>$ |
| | t=false | BB3 CS1 | | |



### 3) Création de la connectique

Architecture à mux

| current | cond | next | fonction | op |
|---|---|---|---|---|
| BB1 CS1 | | BB1 CS2 | $tmp0 <= 0.89 * x$ | $*$ |
| BB1 CS2 | | BB2 CS1 | $y <= 0.22 + tmp0$<br>$i <= 1$ | $:=, +$ |
| BB2 CS1 | | BB2 CS2 | $i <= incr(i)$<br>$tmp1 <= x/y$ | $++, /$ |
| BB2 CS2 | t=true | BB2 CS2 | $i <= incr(i)$<br>$t <= (i < 4)$<br>$y <= shr(y + tmp1, 1)$ | $<, +, >>$ |
| | t=false | BB3 CS1 | | |








## FSM

### BUT: Générer la Partie Contrôle

**A ce stade on a tout:**

- la PO donne le mot d'instruction
  - les enables des registres
  - les commandes des muxs
- la pseudo-FSM donne le mot pour chaque CS
- Déterminer la période d'horloge

| current | cond | next | fonction | i | x | y | tmp | mi |
|---|---|---|---|---|---|---|---|---|
| CS11 | | CS12 | $tmp0 <= 0.89 * x$ | 0 | 0 | 0 | 1 | - |
| CS12 | | CS21 | $y <= 0.22 + tmp0$<br>$i <= 1$ | 1 | 0 | 1 | 0 | 0 |
| CS21 | | CS22 | $i <= incr(i)$<br>$tmp1 <= x/y$ | 1 | 0 | 0 | 1 | 1 |
| CS22 | t | CS22 | $i <= incr(i)$<br>$t <= (i < 4)$<br>$y <= shr(y + tmp1, 1)$ | 1 | 0 | 1 | 0 | 1 |
| | !t | ... | | | | | | |





## MASQUES

descriptions de la PC
$+$
description de la PO
$\Downarrow$
CAD classique
$\Downarrow$
masques

Après cette génération, on a:

- la surface exacte
- la période d'horloge minimale

elles sont obtenues par la CAD

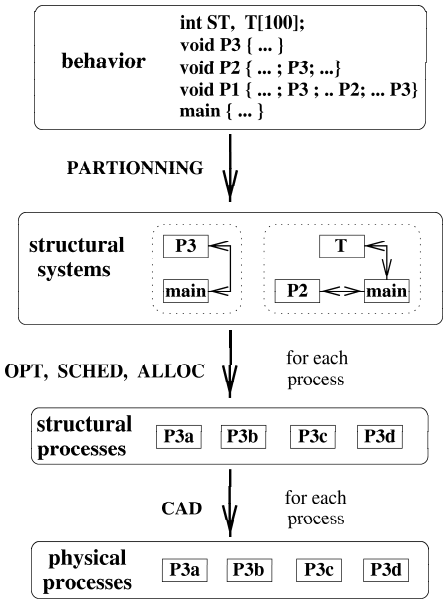


# Chapitre 2

# PRINCIPE

# SCHEMA

## Etapes et espaces de conception








# ESTIMATEURS

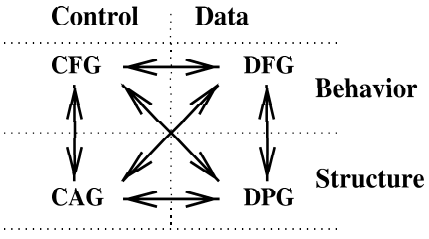
exploration de l'espace de conception
⇓
la connaissance des caractéristiques
des solutions

les caractéristiques ne sont
précisément connues
qu'après la génération
⇓
**HLS grand consommateur d'estimateurs**





# GESTION




- 4 vues orthogonales
- 4 vues extrèmement liées



# Partie III

# PHASES

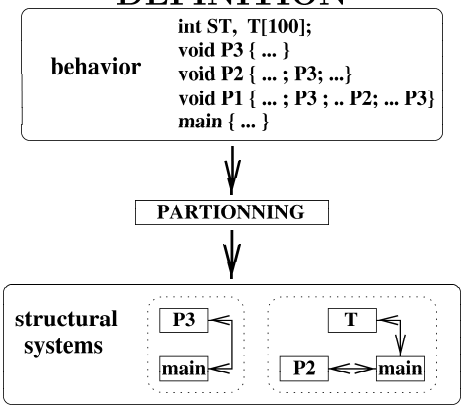
# Chapitre 1

# PARTITIONNEMENT





## DEFINITION




### Procédural ⇒ Processus concurrents

- découper le problème
  - pour les outils qui suivent
  - pour des raisons matérielles
- recherche de performance
  - parallélisme
  - pipeline





## CONTRAINTES

- surface
  - découper en fonctions/processus
  - diminuer la connectique (sérialiser)
  - si plusieurs chips: le nombre de pins et le floor-plannig
- performance
  - parallélisme statique (inline,duplication)
  - parallélisme dynamique (duplication)
  - pipeline
  - minimiser les échanges (paralléliser)
  - choix des protocoles





## PROBLEMES

### Problèmes complexes

- échanges: protocoles/connectique
  - évaluer la quantité (boucles)
  - évaluer le flux (périodique,épisodique, par lot)
  - choisir inline ou fonction ou processus
  - si processus choisir regrouper/sérialiser et le protocole
- surface
  - estimer la taille d'un processus
  - si floor-planning (la forme)

## Exemple

VHDL comportemental de la parité

function Even(bv : bit_vector) return bit is variable S: int = 0; begin for i in bv'range loop if bv[i]='1' then S := S + 1; end if; end loop; return if (S mod 2)=0 then '0'; else '1'; end if; end Even

```
function Even(bv : bit\_vector) return bit is
variable S: int = 0;
begin
for i in bv'range loop
    if bv[i]='1' then S := S + 1; end if;
end loop;
return if (S mod 2)=0 then '0';
                      else '1';
       end if;
end Even
```

**suivant la qualité de *HLS* et le temps passé**

un xor
un arbre de petits adders
une boucle avec un petit adder
une boucle avec un grand adder





## Exemple

Algorithme utilisant Even

process p1 process p2 process p3 ... ... ... b=even(bv); if (...) then for i in bv'range loop ... b:=even(bv); send(bv[i]) ... end if; end loop c=even(bv); ... send(even(bv); ... ... ...

```
process p1          process p2          process p3
...                 ...                ...
b=even(bv);         if (...) then      for i in bv'range
...                   b:=even(bv);       send(bv[i])
...                 end if;            end loop
c=even(bv);         ...                send(even(bv);
...                 ...                ...
```

choix dépend du contexte

comment choisir inline/fonction/processus sans la générer dans p1, p2, p3 ?

comment générer p1, p2, p3 sans avoir déjà even ?





# PRATIQUE

**Dans la pratique, le partionnemment se limite**

- utilisateur donne les processus
- les processus appellent des fonctions
- fonctions prises 1 à 1
  (contexte se limitant aux nombre d'appels)
  - inline ou processus
  - si processus protocole est "hand shake"



# Chapitre 2

# OPTIMISATION

## TRANSFORMATION DES CFGs et DFGs

**Propagation des constantes**

**Précisions des variables**

**Simplification d'expressions**

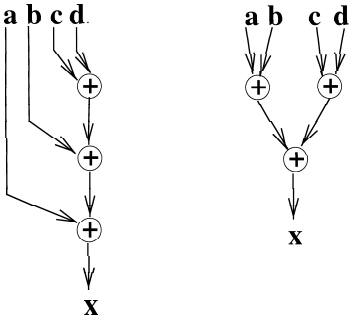
**Choix des opérateurs**

**Traitement des boucles**

**Câblage des si**

# EXPRESSIONS

$x = a + b + c + d$




$x = a + a + b \Rightarrow x = (a << 2) + b$





# OPERATEURS

$i = i + 1 \Rightarrow$ **incrémenteur**

$s = a + b + 1 \Rightarrow$ **additionneur (cin à 1)**

$s = a/8 \Rightarrow$ **décalage de 3 bits**

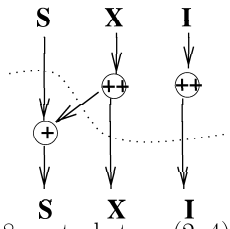




# BOUCLES

$$S = \sum_{k=X+1}^{X+4} k$$

```
S=0; X=?
for ( i=0 ; i<4 ; i++) {
  X= X+1;
  S= S+X
}
```

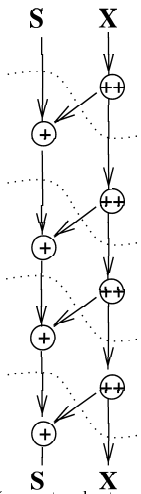



8 control-steps (2x4)



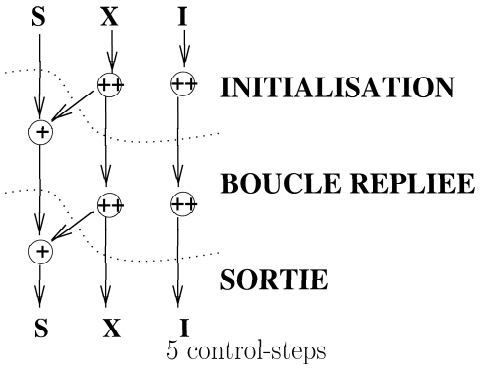


DEROULEMENT




5 control-steps

REPLIEMENT



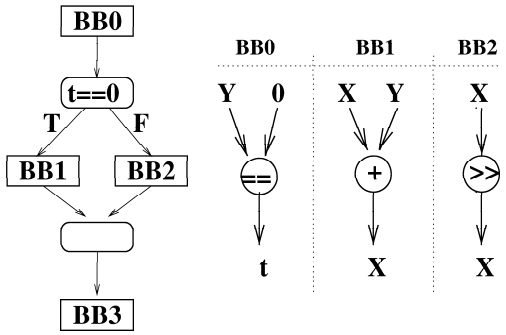

5 control-steps





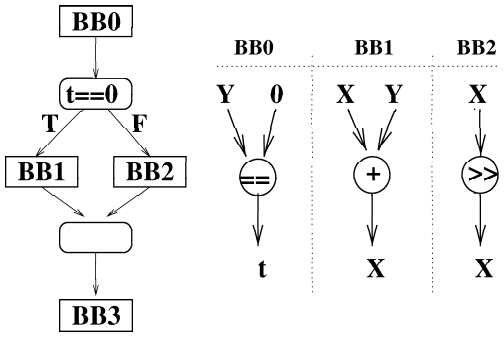
## SI CABLE

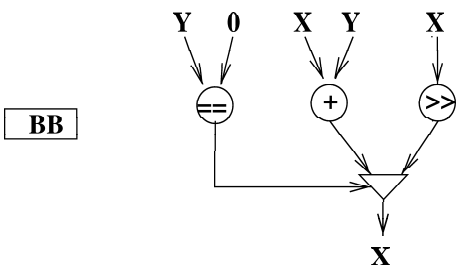
```
if (y−−0)
    x = x + y;
else
    x = y >> 2;
```










# Chapitre 3

# SCHEDULING

# DEFINITION

**HLSM +** **caractéristiques temporelles & physiques des opérateurs**

$\Downarrow$

**FSMD**
**(Description au niveau RT)**

**1 ALLOCATION TEMPORELLE DES OPERATIONS**

**2 DETERMINATION DU NOMBRE MINIMUM D'OPERATEURS PHYSIQUES**





# ALGORITHME

## Régles

1 opérateur ne peut être utilisé qu'une fois dans 1 CS
toute valeur qui traverse un CS est mémorisée

**contraintes utilisateurs**

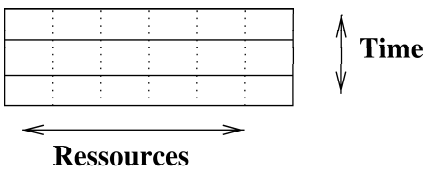
le plus vite possible pour le moins cher
le plus vite possible pour un prix donné





## Algorithmes de scheduling

- élémentaires
  - ASAP : As Soon As Possible
  - ALAP : As Late As Possible
- time constrained
  - minimiser le nombre d'opérateurs
- ressources constrained
  - minimiser le temps d'exécution

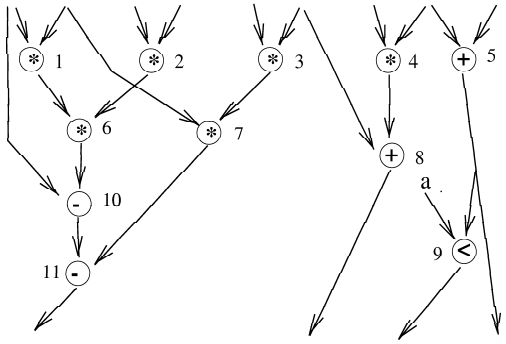







# ALGO DE BASE
## ASAP

**Principe:** placer au plus tôt les noeuds sans prédecesseur.

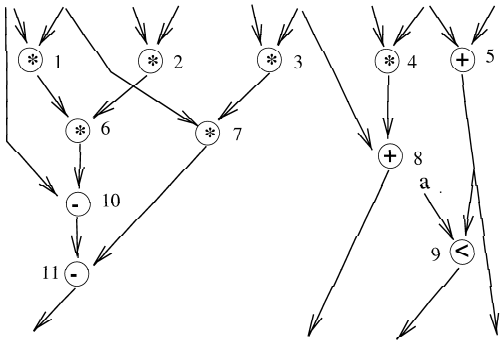



| | |
|---|---|
| **CS0** | $*(1)$ $*(2)$ $*(3)$ $*(4)$ $+(5)$ |
| **CS1** | $*(6)$ $*(7)$ $+(8)$ $<(9)$ |
| **CS2** | $-(10)$ |
| **CS3** | $-(11)$ |

Utilise: 4 $*$ et 2 $+/-$

## ALAP

**Principe:** placer au plus tard les noeuds sans succcsscur.




| | | | | |
|---|---|---|---|---|
| **CS0** | $*(1)$ | $*(2)$ | | |
| **CS1** | $*(6)$ | $*(3)$ | | |
| **CS2** | $-(10)$ | $*(7)$ | $*(4)$ | $+(5)$ |
| **CS3** | $-(11)$ | | $+(8)$ | $<(9)$ |

Utilise: $2*$ et $3+/-$





## RESS. CONSTRAINED
### DEFINITION

Faire le scheduling optimum pour un coût matériel donné avec

- scheduling optimum
  - minimiser le nombre de control-steps
- coût matériel donné
  - surface maximale
  - consommation maximale

### PRATIQUEMENT

- coût matériel donné
  - la surface globale des opérateurs
  - le nombre d'opérateurs d'un type donné
  - le nombre de registres
  - le nombre de bus

### ALGORITHMES

List Based Scheduling
([HAL 89], [CALLAS 93])

Integer Linear Schedulig





## First Come First Served

### Principe

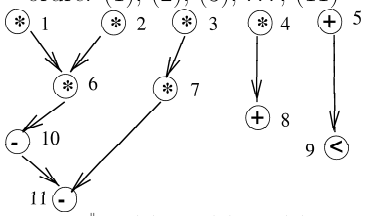
- ordonner les noeuds
- se placer sur le premier CS
- placer au plus tôt les noeuds sans prédecesseur sur les UF disponiples (si conflit, choisir le premier dans l'ordre)
- passer au CS suivant et recommencer





## First Come First Served

**Exemple:**
contraintes: $2*$, $2+/-$
ordre: (1), (2), (3), ..., (11)




| | | | | |
|---|---|---|---|---|
| CS0 | $*(1)$ | $*(2)$ | $+(5)$ | |
| CS1 | $*(3)$ | $*(4)$ | $<(9)$ | |
| CS2 | $*(6)$ | $*(7)$ | $+(8)$ | |
| CS3 | $-(10)$ | | | |
| CS4 | $-(11)$ | | | |

**REMARQUE:**
La qualité du scheduling dépend de l'ordre.

## LIST BASED SHEDULING

### Principe

- identique à l'ASAP resource-constrained
- on joue sur l'ordre des noeuds
  (liste prioritaire de noeuds)

### Variantes

- On recalcule cette liste à chaque CS

### priorités usuelles

- la mobilité (ALAP-ASAP)
- longueur du chemin jusqu'à la fin
- le nombre de noeuds dépendants

### Qualité

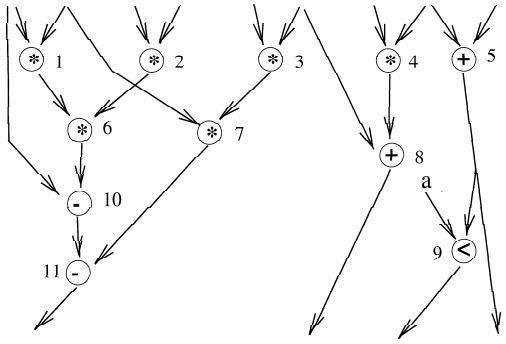
- répond bien au problème
- heuristique → pas de garantie d'optimum
- fonctionne d'autant mieux que les BBs sont courts





## LIST BASED SHEDULING (ex)

contraintes: 2 $*$, 2 $+/-$




| | ASAP | ALAP |
|---|---|---|
| CS0 | $*$ (1) $*$ (2) $*$ (3) $*$ (4) $+$(5) | $*$ (1) $*$ (2) |
| CS1 | $*$ (6) $*$ (7) $+$(8) $<$ (9) | $*$ (6) $*$ (3) |
| CS2 | $-$(10) | $-$(10) $*$ (7) $*$ (4) $+$(5) |
| CS3 | $-$(11) | $-$(11) $+$(8) $<$ (9) |

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ALAP | 1 | 1 | 2 | 3 | 3 | 2 | 3 | 4 | 4 | 3 | 4 |
| ASAP | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 4 |
| *mob.* | 0 | 0 | 1 | 2 | 2 | 0 | 1 | 2 | 2 | 0 | 0 |

liste prioritaire: 1 2 6 10 11 3 7 4 5 8 9



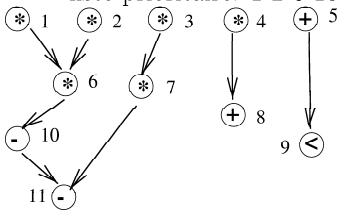


## LIST BASED SHEDULING (ex)

contraintes: 2 $*$, 2 $+/-$

### PAS 1

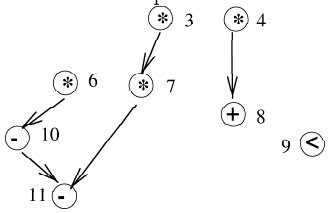
liste prioritaire: 1 2 6 10 11 3 7 4 5 8 9




**CS0 *(1) *(2) +(5)**

### PAS 2

liste prioritaire: 6 10 11 3 7 4 8 9




**CS0 *(1) *(2) +(5)**

**CS1 *(3) *(6) <(9)**



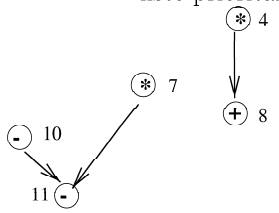


## LIST BASED SHEDULING (ex)

contraintes: 2 $*$, 2 $+/-$

### PAS 3

liste prioritaire: 10 11 7 4 8



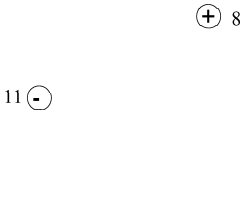

**CS0 *(1) *(2) +(5)**

**CS1 *(3) *(6) <(9)**

**CS2 *(7) *(4) -(10)**

### PAS 4

liste prioritaire: 10 8




**CS0 *(1) *(2) +(5)**

**CS1 *(3) *(6) <(9)**

**CS2 *(7) *(4) -(10)**

**CS4 -(11) +(8)**

### Scheduling en 4 CS

# TIME CONSTRAINED
## DEFINITION

Faire le scheduling optimum pour un coût minimal avec

- scheduling optimum
  - minimiser le nombre de control-steps
- coût minmal
  - minimiser la surface

### PRATIQUEMENT

- minimiser la surface
  - la surface globale des opérateurs
  - le nombre de registres
  - la connectique

### PLUSIEURS ALGORITHMES

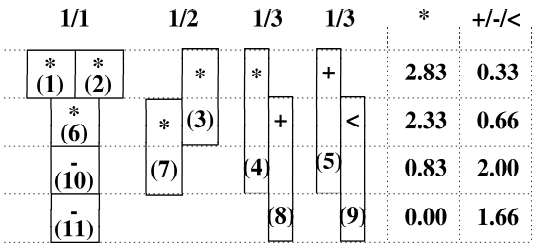
- ILS (Interger Linear Scheduling) [Lee 89]
- IRS (Iterative Refinement Scheduling) [Park 91]
- FDS (Force Directed Scheduling) ([HAL 89])
- ...





## Force-Directed Scheduling

| | ASAP | | | | | ALAP | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| CS0 | $*(1)$ | $*(2)$ | $*(3)$ | $*(4)$ | $+(5)$ | $*(1)$ | $*(2)$ | | | |
| CS1 | $*(6)$ | $*(7)$ | $+(8)$ | $<(9)$ | | $*(6)$ | $*(3)$ | | | |
| CS2 | $-(10)$ | | | | | $-(10)$ | $*(7)$ | $*(4)$ | $+(5)$ | |
| CS3 | $-(11)$ | | | | | $-(11)$ | | $+(8)$ | $<(9)$ | |




**distribution de la concurence**

$DG_{op}(i) > DG_{op}(j)$

$\Downarrow$

dans CSi, la concurence de op est potentiellement plus grande que dans CSj





## Force-Directed Scheduling

**Force de rappel d'une allocation** $A_{c,o}$

$$F_{A_{c,o}} = \sum_{cs,op} DG_{op}(cs) * dp_{cs,op}$$

**avcc**

$DG_{op}(cs)$ distribution de l'operation op au CS cs
$dp_{cs,op}$ différence de probabilité de l'opération op au CS cs après l'allocation $A_{c,o}$

$$F_{A_{c1,o1}} > F_{A_{c2,o2}}$$

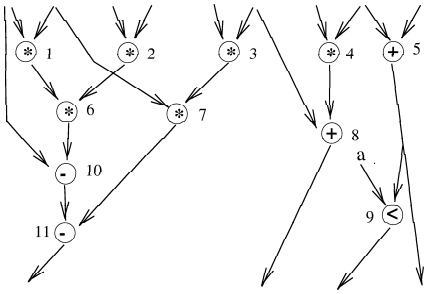
$\Downarrow$

$A_{c1,o1}$ met plus de parallèlisme que $A_{c2,o2}$

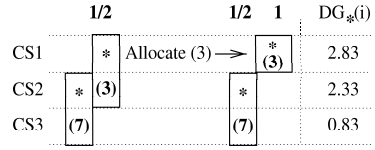




## Force-Directed Scheduling (ex)

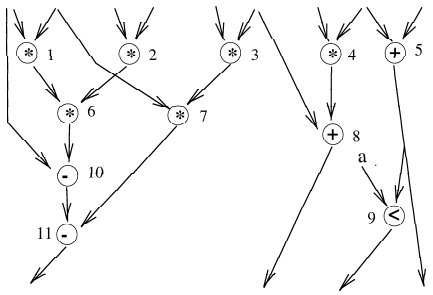



$$F_{A_{c,o}} = \sum_{cs,op} DG_{op}(cs) * dp_{cs,op}$$

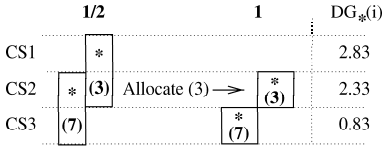



$$F_{A_{1,3}} = DG_3(1) * dp_{1,3} + DG_3(2) * dp_{2,3}$$
$$- 2.83 * (1 - 1/2) + 2.33 * (0 - 1/2) - 0.25$$

## Force-Directed Scheduling (ex)




$$F_{A_{c,o}} = \sum_{cs,op} DG_{op}(cs) * dp_{cs,op}$$




$$\begin{aligned} F_{A_{2,3}} &= DG_3(1) * dp_{1,3} + DG_3(2) * dp_{2,3} + \\ & DG_7(2) * dp_{2,7} + DG_7(3) * dp_{3,7} \\ &= 2.83 * (0 - 1/2) + 2.33 * (1 - 1/2) \\ & 2.33 * (0 - 1/2) + 0.83 * (1 - 1/2) = -1 \end{aligned}$$



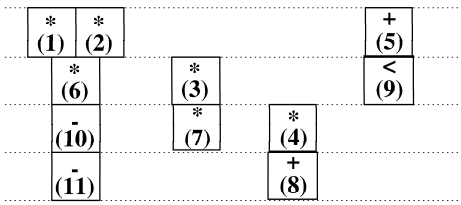


## Force-Directed Scheduling

### ALGORITHME

- calculer les $DG_o(c)$
- parmi toutes les allocations $A_{c,o}$ possibles choisir celle dont $F_{A_{c,o}}$ est minimale
- la scheduler
- recommencer tant qu'il reste des operations non schedulées

| operation | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CS 1 | P | P | 0.25 | 0.83 | -0.66 | | | | | | |
| CS 2 | | | -1 | 0.50 | 0.05 | P | 1 | 0.05 | -1.45 | | |
| CS 3 | | | | -0.95 | 1.22 | | -0.75 | 1,14 | 0.28 | P | |
| CS 4 | | | | | | | | 0.22 | 0.22 | | P |




**2 +/-, 2 ***





## Force-Directed Scheduling

### HEURISITIQUE

ne garantit pas l'optimum

..., the following improved force calculation was obtained after intensive experimentation:

$$F_{A_{c,o}} = \sum_{cs,op} DG_{op}(cs) * dp_{cs,op} + dp_{cs,op}{}^2$$





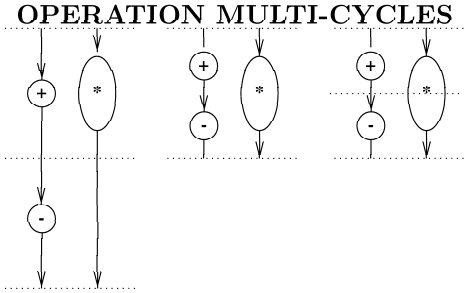
# REALISTE
## APPROCHE DE LA REALITE

**cycle ≠ opérateur le plus lent**
- le chaînage d'opérateurs
- les opérateurs multi-cycles

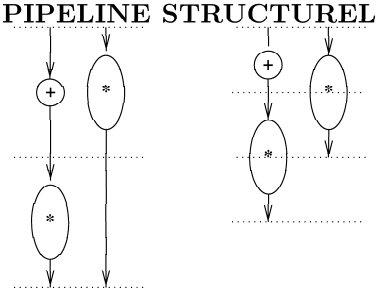
**choix des opérateurs**
- pipeline structurel
- opérateur multi-fonction: opérations ≠ partagent le même opérateur (+ 10 bits, + 5 bits), (+, - : ASB)
- des opérateurs peuvent avoir plusieurs sorties
- opérateurs ≠ pour la même fonction (mult, pmult 2 étages, pmult 4 étages)

**optimisation globale**
- pipeline fonctionnel
- optimisation globale de la FSM. (prise de décision: si, cas, boucles)
- optimisation globale de la surface.

## CHAINAGE D'OPERATIONS
## OPERATION MULTI-CYCLES



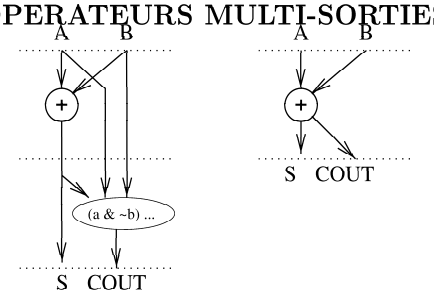
## PIPELINE STRUCTUREL



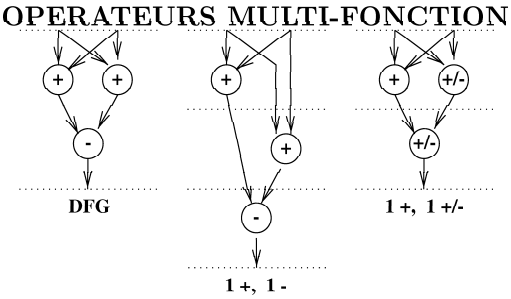




## OPERATEURS MULTI-SORTIES



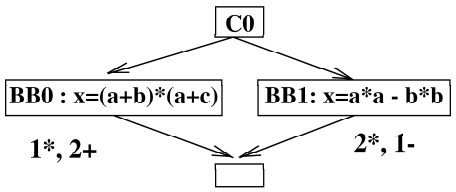

## OPERATEURS MULTI-FONCTION



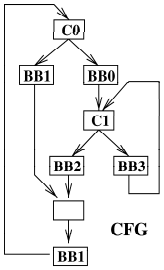





## OPTIMISATION GLOBALE DE SURFACE




**total: 2*, 2+, 1-**




BB1: 1 ∗1 stage BB2: 1 ∗2 stages BB3: 1 ∗4 stages

total: 3 ⋆

## C'EST UNE NECESSITE

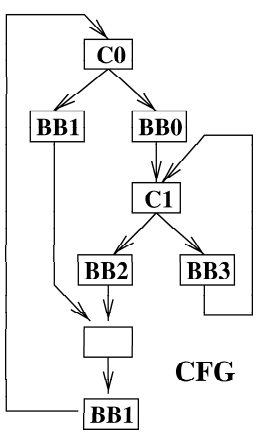




## OPTIMISATION GLOBALE DE FSM

PRINCIPE
Minimiser le nombre de BBs
Agrandir les BBs

MOYEN
Dépacer les instructions entre les BBs
câbler les SIs
Dérouler les boucles
Replier les boucles



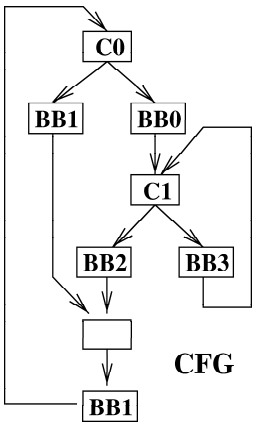
## ALGORITHMES

User constrained (nombre UFs, registres)

Force Directed Scheduling [HAL 89]

Path Scheduling [HIS 91]

Trace Scheduling [Fisher 81]




CFG



# Chapitre 4

# ALLOCATION





## DEFINITION

**FSMD** (description au niveau RT) + **caracteristiques physiques des opérateurs**

⇓

**Description structurelle de la PO**
**Description de la FSM de la PC**



## FONCTIONS A REALISER

**1 Allocation**
- des unités de mémorisations
- des unités fonctionelles
- des bus et des multiplexeurs de connection

**2 Générer la FSM**
- les enables des registres
- les selecteurs des mux et/ou tristates

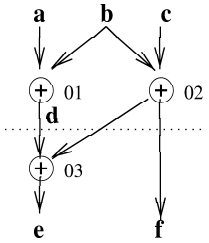
## CONTRAINTES

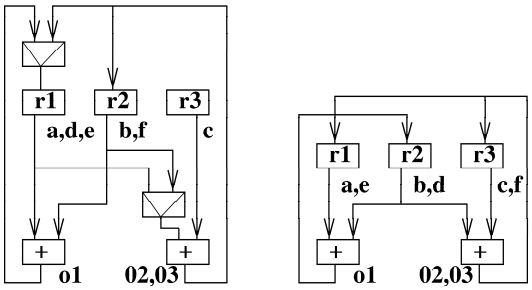
**1 Architecture cible**
- à bus
- à multiplexeurs

**2 Surface**

**3 Contraintes électriques**

# INTERDEPENDANCE




**Dépendance entre** allocation des registres et des opérateurs

Dans la pratique une puis l'autre



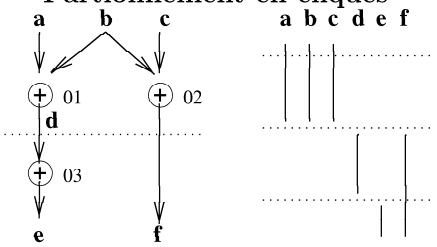


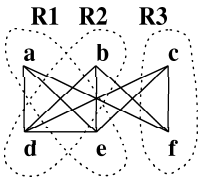
# REGISTRES

**But**

Minimiser le nombre de registres
Allouer les variables sur les registres

**Partionnement en cliques**







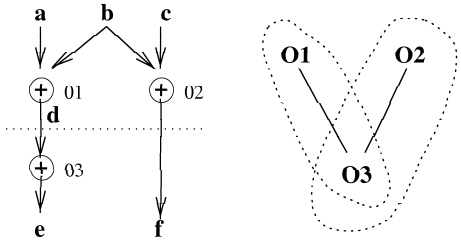

# OPERATEURS

**But**

Minimiser le nombre d'opérateurs
Allouer les opérations sur les opérateurs

**Partionnement en cliques**

noeud: opération
arc: si l'opération peut être faite par le même opérateur








# RISQUE

création d'une connectique importante

# AMELIORATION

Poids sur les arcs

heuristique: $P = N_{in} + N_{out}$
**coût** $= \Sigma P_c$

avec
$N_{in}$: nombre d'entrées communes
$N_{out}$: nombre de sorties communes

Algorithme

Partionner en cliques de coût maximal

# CONNECTIONS
## But

Minimiser le nombre de bus et de multiplexeurs

Générer la description structurelle de la PO

1 seul degré de liberté pour les operateurs commutatifs





# SYNTHESE LOGIQUE
## PRINCIPE

traduire le FSMD en équations booléennes
donner le tout à la synthèse logique.

### AVANTAGES

simplicité
très bonnes optimisation globale

portabilité

### INCONVENIENTS

Limite de la synthèse logique

Scheduling plus approximatif
delays estimés car opérateurs inconnus

impossibilite d'utiliser des compilateurs de macro-cells

### DOMAINE D'UTILISATION
réservé à de petits controlleurs



# Partie IV

# CONCLUSION





# PROBLEMES OUVERTS

### INTERNES AUX PHASES

### INTERDEPENDANCES DES PHASES

### CONTRAINTES UTILISATEUR

### LANGAGES D'ENTREE

# INTERNE AUX PHASES
# PROBLEMES SPECIFIQUES

## OPTIMISATION

- gestion des opérateurs multi-fonctions
- gestion des opérateurs à plusieurs sorties
- mettre en place des architectures pipelinées
- sélection des opérateurs

## SCHEDULING

- optimisation globales
- gestion efficace du multi-cycle et du chaînage
- abandon de la règle mémorisation des variables à chaque CS
- scheduler sur des modèles temporels précis

## ALLOCATION

on sait

- minimiser le nombre de registres
- minimiser le nombre de ressources

on sait moins bien

- minimiser la surface totale
- générer des POs correctes (arbre de clocks, de buffers, ...)





# PROBLEMES COMMUNS

**Représentation des UFs pour la portabilité**
(fonction, caractérisation, structure)

**Utilisation des data-paths de différents types**

**Interface avec les outils classiques de CAD**

**Prise en compte des capacités de routage**



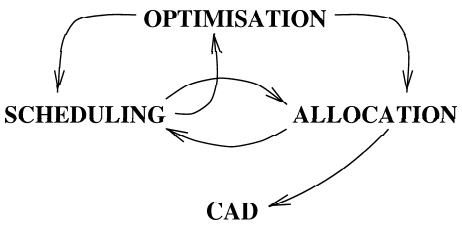


# INTERDEPENDANCES

OPTIMISATION $\Rightarrow$ SCHEDULING et ALLOCATION
- câblage des si
- traitement des boucles

SCHEDULING $\Rightarrow$ OPTIMISATION et ALLOCATION
- optimisation pour scheduler
- allocation pour avoir les modèles temporels
- allocation pour positionner une opération dans un CS plus que dans un autre

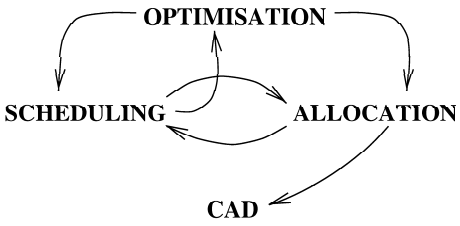
ALLOCATION $\Rightarrow$ SCHEDULING








# IMPASSE




# ESSAIS SUCCESSIFS

réalisation du floor-planning

# RESOLUTION

estimateurs

# CONTRAINTES

## IDEAL

petit & économique

rapide

bon compromis

synchrone

## EN PRATIQUE

limitation de la surface des opérateurs

limitation du nombre de certains opérateurs
( 1 ALU, 2 adder, 1 multiplieur )

limitation du nombre de registres

limitation du nombre de bus

séquence d'instructions à faire en
un temps ou nombre de cycles précis





# LANGAGE D'ENTREE

## COMPORTEMENTAL PUR

(pascal, C, ...)

### Description générale de la parité

```
int parite(int x)
{
int ret=0;
while (x!=0) {
 ret += x%2;
 x >>= 1;
 }
return( ret%2 );
}
```

### Problèmes

tailles de x, ret, parite

HLS génère une boucle





# COMPORTEMENTAL ORIENTE STRUCTUREL

(manipulation de bits et de vecteurs)

### Descriptions de parité

```
#define N 10

bit parite(bit x[N])
{
int i;
int ret=0;
for (i=0 ; i<N ; i++) {
 ret += x[i];
 }
return( ret%2 );
}
```

```
#define N 10

bit parite(bit x[N])
{
int i;
bit ret=0;
for (i=0 ; i<N ; i++) {
 ret ^= x[i];
 }
return( ret );
}
```

### Problèmes

taille de ret

Pas de description générale
2 algorithmes équivalents ⇒ 2 structures différentes

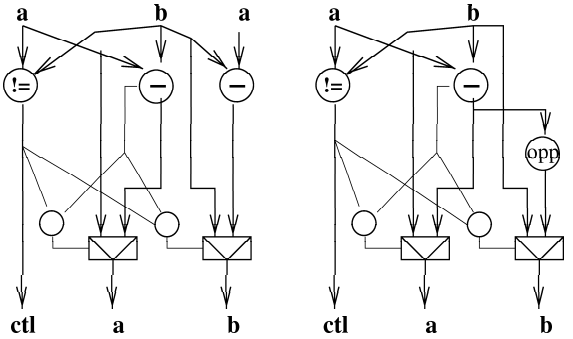




# INFLUENCE DE L'ALGORITHME

```
pgcd(a,b)
{
while (a!=b)
 if (a<b)  b= b-a;
 else      a= a-b;
return(a);
}
```

```
pgcd(a,b)
{
while (a!=b) {
 x= a-b;
 if (x<0)  b= -x;
 else      a= x;
}
return(a);
}
```




### Problèmes

2 algorithmes équivalents ⇒ 2 structures différentes
Utilisateur ⇒ structurel préalablement
Utilisateur → connaissance de l'outil

# INFLUENCE DE L'ALGORITHME

pipeline

### Descriptions de la racine carrée

```
void rac(float x,float *ret)
{
float y;
int i;
y= 0.22 + 0.89 * x;
for (i=0 ; i<5 ; i++)
  y = (y + x/y)/2;
*ret = y;
}
```

```
void racp(float x0,x1,*ret0,*ret1)
{ float y0,y1a,y1b;
  int i;
/* init of pipe */
  y0= 0.22 + 0.89 * x0;
  y1a,y1b= y0,x0/y0;
  y0= 0.22 + 0.89 * x1;
  perm(&x0,&x1);
/* loop */
  for (i=0 ; i<9 ; i++) {
    y0,y1a,y1b= (y1a+y1b)/2,y0,x0/y0;
    perm(&x0,&x1);
  }
/* get results */
  *ret0= y0;
  y0= (y1a+y1b)/2;
  *ret1= y0;
}
```

### Problèmes

algorithme pipeliné ⇒ structurel préalablement

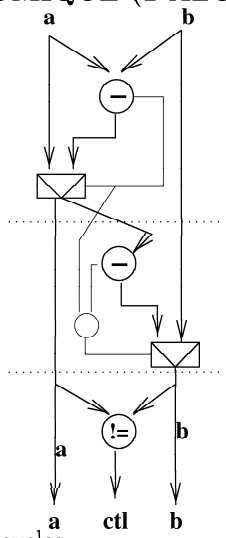




# CONTRAINTES

### PGCD ECONOMIQUE (1 ALU)

```
pgcd(a,b)
{
while (a!=b)
 if (a<b)
   b = b-a;
 else
   a = a-b;
return(a);
}
```




3 cycles



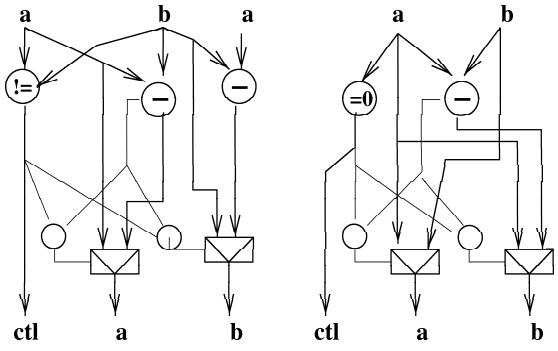


# CONTRAINTES

### PGCD ECONOMIQUE ET EFFICACE SANS CONTRAINTE

```
pgcd(a,b)
{
while (a!=b)
 if (a<b)
   b = b-a;
 else
   a = a-b;
return(a);
}
```

```
pgcd(a,b)
{
while (a!=0)
 if (a>b)
   a,b = b,a;
 else
   b = b-a;
return(b);
}
```








# CONTRAINTES

### Problèmes

containtes physiques ⇒ modification de l'algorithme

Utilisateur ⇒ structurel préalablement

Utilisateur ⇒ connaissance de l'outil

### Autre exemple:

RISC

Posibilités hardware ⇒ le jeu d'instruction

## PERSPECTIVES

## COMPORTEMENTAL PUR

peu probable même à long terme

## COMPORTEMENTAL ORIENTE STRUCTUREL

des implémentations existent

résultats probants pour des petits controleurs

niveaux supérieurs: conçoit, crypte, HLS décryptera

## LANGAGE APPLICATIF

comme le traitement du signal
résultats les plus probants

Large





*Fast and Near Optimal Scheduling in Automatic Data Path Synthesis"*,
Proceedings of the 28th Design Automation Conference, pp 680-685, 1991.

[Lee 89] Lee j., Hsu Y., Lin Y.
*"A New Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis"*,
Proceedings of the International Conference on Computer-Aided Design, pp 20-23, 1889.

[Fisher 81] Fisher J.A
*"Trace Scheduling: A Technique for Global Microcode Compaction."*,
IEEE Transactions on Computer, Vol. C-30, NO. 7, Jul. 1981.



## Bibliography

[HLS] Camposano R.
*"From Behavior To Structure: High-Level SYNTHESIS"*
IEEE DESIGN & TEST OF COMPUTERS, Oct 1990.

[Gajski 92] Gajsky D., Nikil D. and all
*High-Level Synthesis*
Kluwer Academic Publisher, 1992

[HAL 89] Paulin P.G, Knight J.P.
*"Force-Directed Scheduling for the Behavioral Synthesis of ASIC's"*,
IEEE Transactions on Computer-Aided Design (CAD),Vol.8, NO. 6, Jan. 1989.

[HIS 91] Camposano R.
*"Path-Based Scheduling for Synthesis."*
IEEE Transactions on Computer-Aided Design (CAD),Vol.10, NO. 1, Jan. 1991.

[CALLAS 93] Biesenack J., Koster M. and all
*"The Simens High Level Synthesis System CALLAS"* IEEE Transactions on very large scale integration systems, Vol 1, NO 3, Sep 1993.

[Park 91] Park I-C, Kyung C-M.

# Placement et routage

**Placement et routage, rétro-annotation. Les algorithmes de placement - routage, la synthèse d'arbres d'amplification d'horloge, le dimensionnement des alimentations, la connexion avec le boîtier, l'extraction des capacités d'interconnexion, le retour dans le flot des informations extraites.**



# Plan

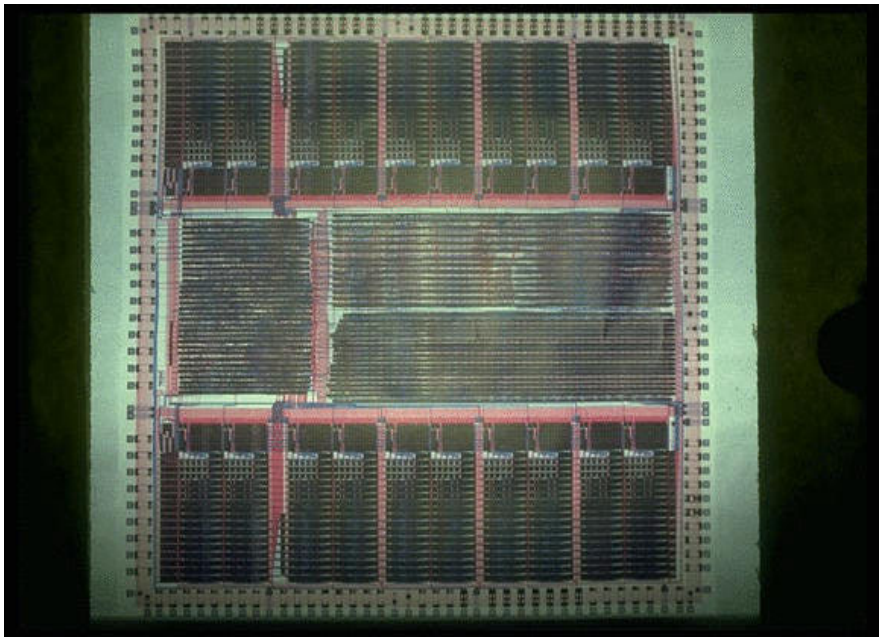
- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*

# Plan

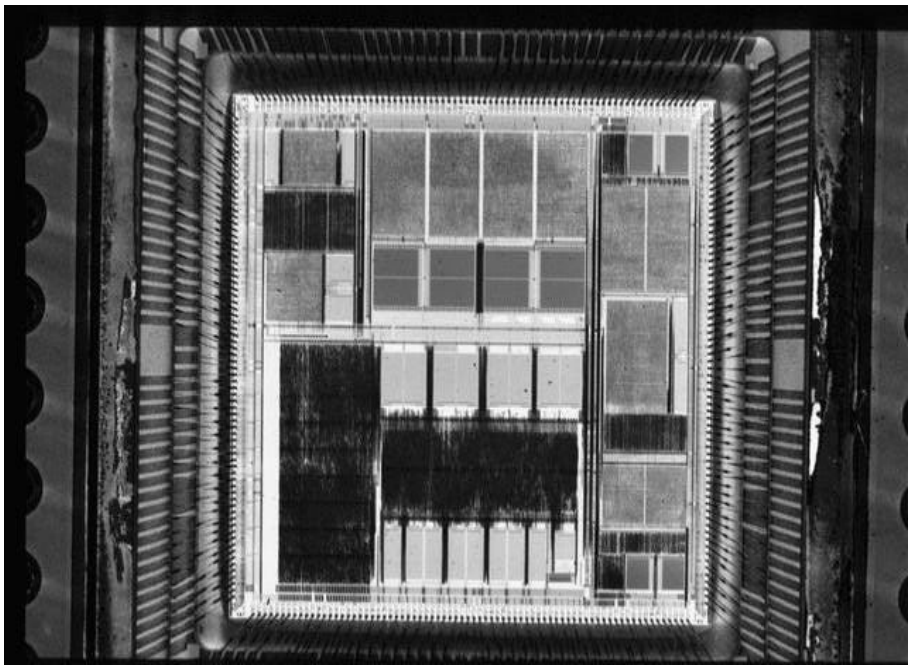
- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*



# Avant la fonderie



# Après la fonderie





# L'évolution des technologies

- La synthèse et le placement sont aujourd'hui indissociables (importance des délais d'interconnexion)
- La complexité et le coût des outils conduit à l'externalisation de la partie physique (« *back end* »)
- Très peu d'acteurs sur ce segment de marché
- Ingénieur « *back end* » : un métier très spécialisé en perpétuelle évolution

# Plan

- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*



# Le plan de masse

- C'est la première étape de la partie physique
- Dans les flots de conception sub-microniques cette étape apparaît très tôt
- On peut commencer un plan de masse dès la fin de l'étude d'architecture
- Les *netlists* peuvent être virtuelles, seule leur taille approximative est nécessaire
- Il faut avoir une estimation des densités de communication entre blocs
- Il faut disposer des blocs « durs » (RAM, ROM, chemins de données, CPU, etc.)

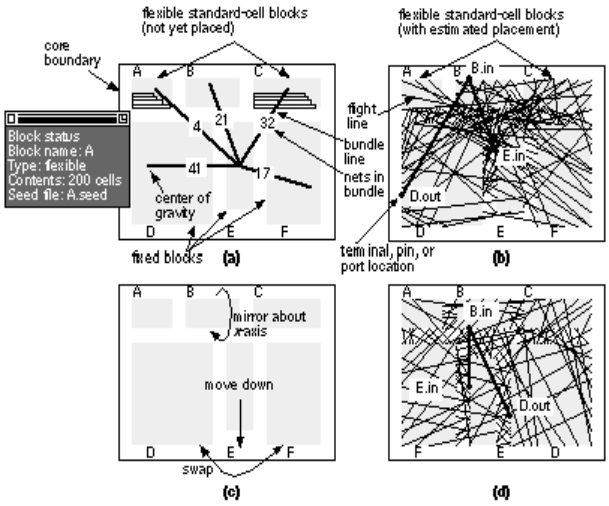
# Les différents objectifs

- Positionnement des blocs sur le circuit
- Emplacement des entrées / sorties
- Emplacement et nombre de plots d'alimentation
  - *Couronne*
  - *Cœur*
- Type et topologie de distribution de puissance
- Type et topologie de distribution d'horloge
- On veut minimiser la surface (facile à mesurer)
- On veut minimiser les délais (plus dur à mesurer)



# Une organisation en blocs

- Un plan de masse contient des blocs « mous » (zones de cellules standard non encore placées) et des blocs « durs »
- L'outil de plan de masse produit une carte des congestions
  - *Soit de centre à centre*
  - *Soit de connecteurs à connecteurs*
- En déplaçant les blocs on réduit les problèmes de congestion



# Blocs durs et blocs mous

- Les blocs « durs » sont figés. Le concepteur peut uniquement les déplacer et opérer des symétries.
- Les zones de cellules standard sont flexibles :
  - *Le concepteur répartit les cellules entre les zones*
  - *Il peut définir le facteur de forme*
  - *Il peut définir l'emplacement des connecteurs d'entrée et de sortie*
  - *Les contraintes du concepteur peuvent être*
    - Absolues, à respecter absolument
    - Indicatives, ce sont des suggestions que les outils utilisent pour construire une solution initiale

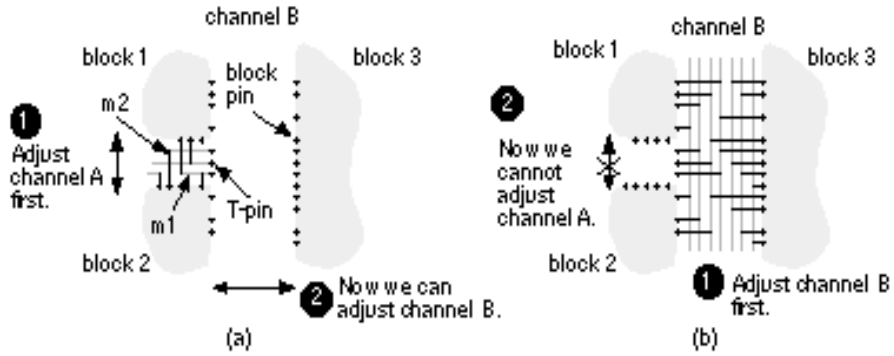


# Plan

- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*

# Les canaux de routage

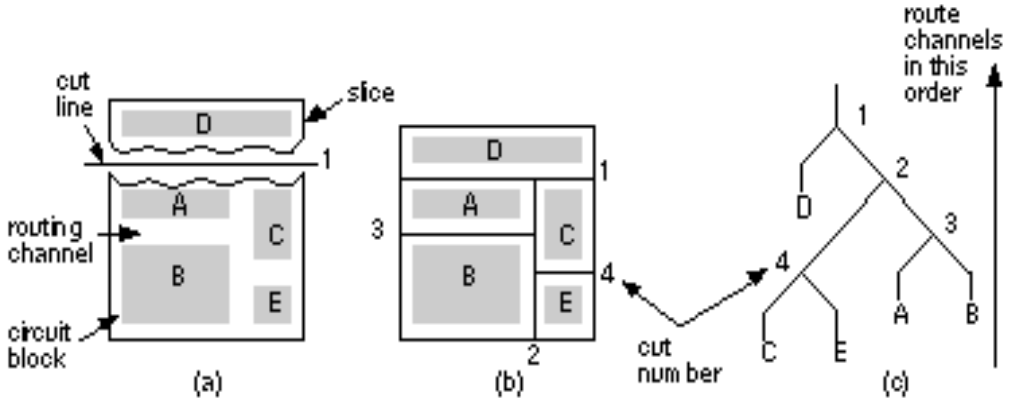
- La plupart des routeurs bloc ne savent router que des canaux rectangulaires
- Dans une configuration en T l'ordre de routage est important





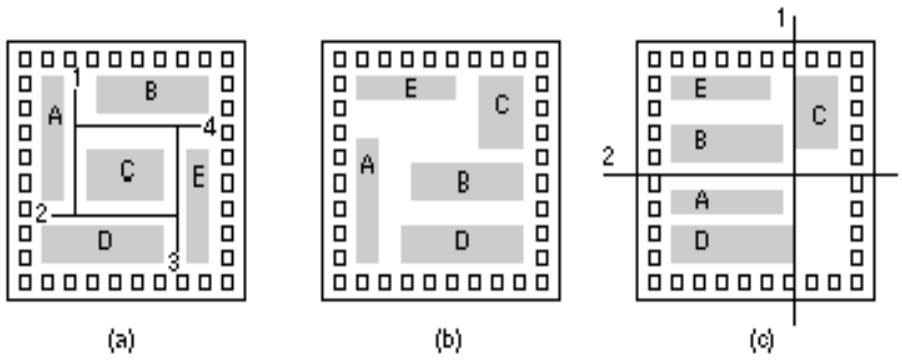

# Ordre de routage

- L'algorithme utilisé pour ordonner les canaux est basé sur un découpage itératif du plan de masse



# Les cycles

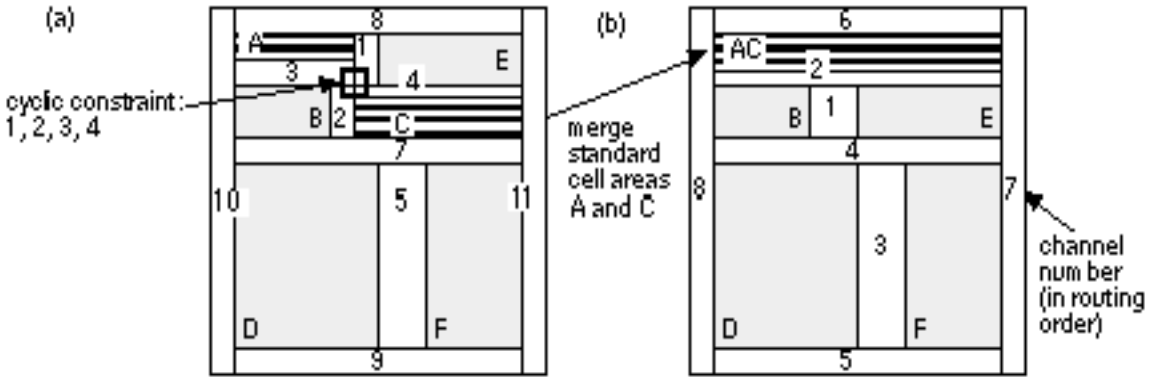
- Pour que l'algorithme d'ordonnancement des canaux soit applicable il faut éviter les cycles






# Résolution des cycles

- On résout parfois les cycles en fusionnant (ou en séparant) des zones de cellules standard



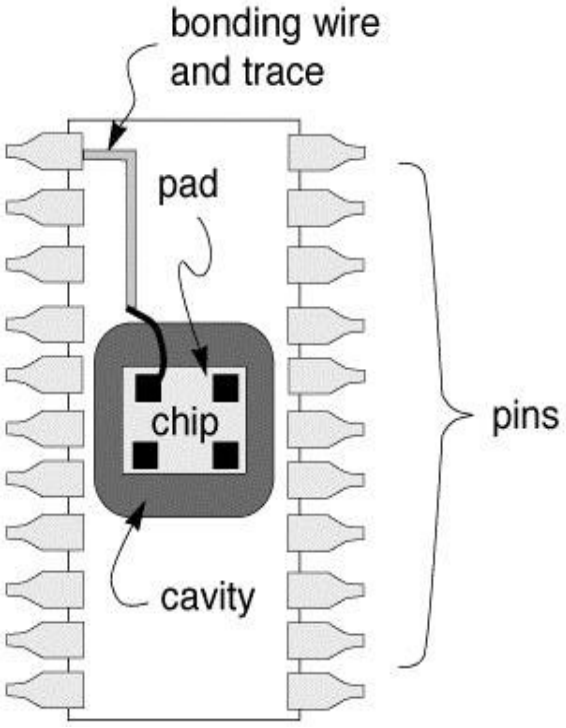
# Plan

- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*



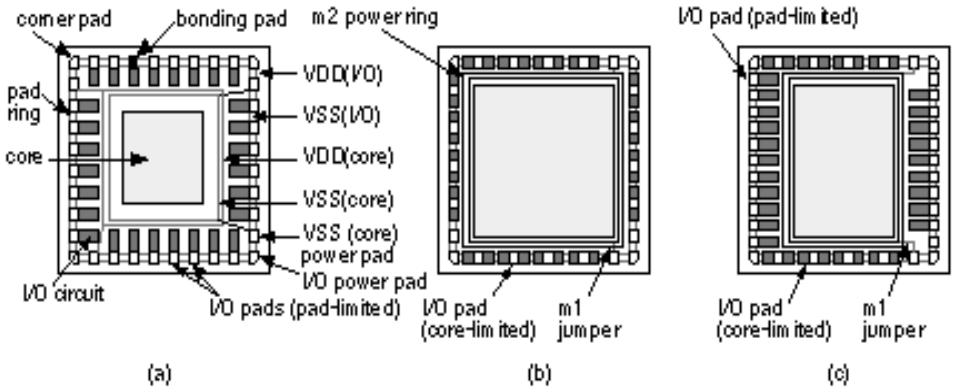
# Du cœur du circuit à la carte

- Les équipotentielles du circuit sont reliées aux plots
- Les plots sont reliés aux pistes du boîtier (« *bonding* »)
- Les pistes sont reliées aux bornes du boîtier



# Les alimentations

- On sépare alimentations du cœur et alimentations des entrées – sorties
- Un circuit peut être « pad limited » ou « core limited »






# La consommation des sorties

- Les amplificateurs de sortie attaquent des capacités de l'ordre de la dizaine de pF (contre 0,01 pF en interne)
- De très forts appels de courant sont produits par les sorties qui commutent simultanément (SSO)
- Il faut des alimentations dédiée pour ne pas perturber le cœur
- On encadre N plots de sortie par un couple de plots VDDO / VSSO (N est préconisé par le fondeur)

# Les plots

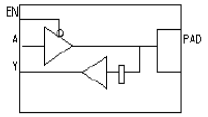
- **Les plots sont des objets complexes**
  - *Amplificateurs*
  - *Protections contre les décharges électrostatiques*
  - **« boundary scan »**
  - *Surfaces de soudage*
  - *Conversions de tensions*

**AMS** 0.35 µm CMOS — BBC1P

BBC1P is a pad-limited CMOS Bidirectional Buffer with 1 mA drive strength.

**Truth Table**

| A | EN | PAD | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| X | 1 | 0 | 0 |
| X | 1 | Z | X |
| X | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |




**Capacitance**

| Pin | Cap [pF] |
|---|---|
| A | 0.034 |
| EN | 0.033 |
| PAD | 4.519 |

**Area**

52.762 mils$^2$
34040 µm$^2$

**Power**

115.6 µW/MHz

Delay [ns] = tpd.. = f(SL, L) with SL = Input Slope [ns] ; L = Output Load [pF]
Output Slope [ns] = op_sl.. = f(SL, L) with L = Output Load [pF]

AC Characteristics: Tj = 25°C VDD = 3.3V Typical Process

**AC Characteristics**

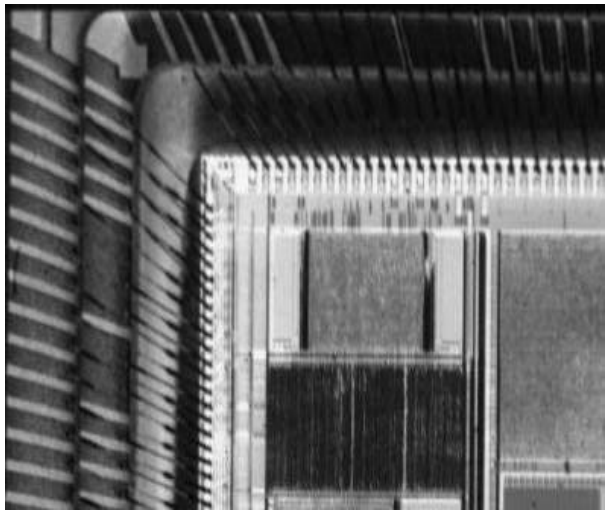
| | Rise | | | | Fall | | | |
|---|---|---|---|---|---|---|---|---|
| **Slope [ns]** | **0.1** | | **2** | | **0.1** | | **2** | |
| **Load [pF]** | **5** | **50** | **5** | **50** | **5** | **50** | **5** | **50** |
| Delay A => PAD | 2.06 | 9.86 | 2.17 | 9.82 | 2.7 | 10.9 | 3.04 | 11.22 |
| Delay EN => PAD | 2.21 | 10.8 | 3 | 10.75 | 3 | 12 | 3 | 12 |
| Slew A => PAD | 4.51 | 26.14 | 4.58 | 26.1 | 3.55 | 20.43 | 3.51 | 20.53 |
| Slew EN => PAD | 7.22 | 39.99 | 6.67 | 38.88 | 3.91 | 27.23 | 4.18 | 26.58 |

| | Rise | | | | Fall | | | |
|---|---|---|---|---|---|---|---|---|
| **Slope [ns]** | **0.1** | | **2** | | **0.1** | | **2** | |
| **Load [pF]** | **0.5** | **5** | **0.5** | **5** | **0.5** | **5** | **0.5** | **5** | |
| Delay PAD => Y | 0.46 | 1.65 | 0.82 | 2.02 | 0.54 | 2.46 | 0.68 | 2.59 |
| Slew PAD => Y | 0.5 | 3.87 | 0.5 | 3.95 | 0.51 | 4.45 | 0.55 | 4.43 |



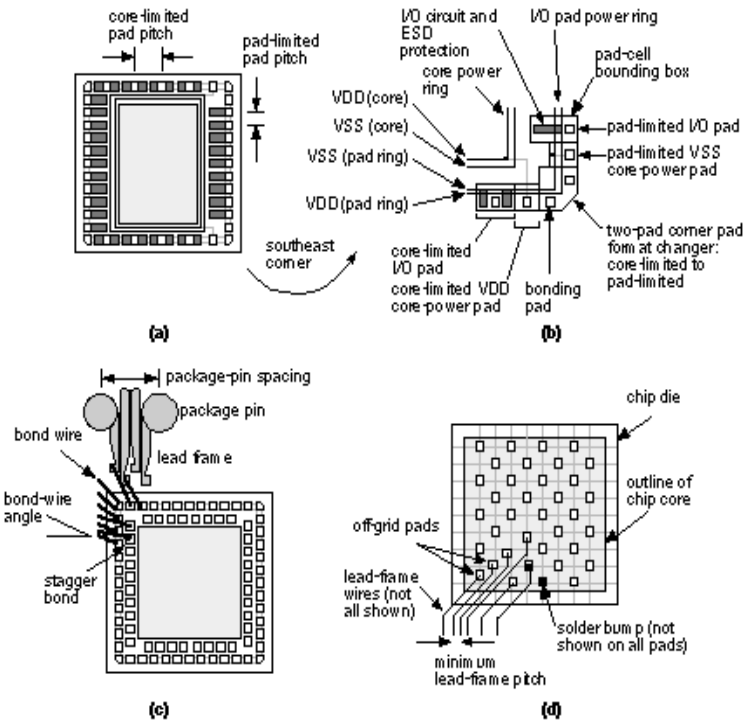
# Le « bonding »

- **La connexion aux bornes du boîtier (« *bonding* ») est soumise à des contraintes fortes**
  - *Longueurs minimum et maximum des fils*
  - *Angles autorisés*
  - *Espacements entre fils*
  - *Inductances*



# Les boîtiers

- **Le type de boîtier influe sur le placement-routage**
  - ***Taille et facteur de forme de la cavité***
  - ***Connexion des plots aux bornes***
  - ***Etc.***



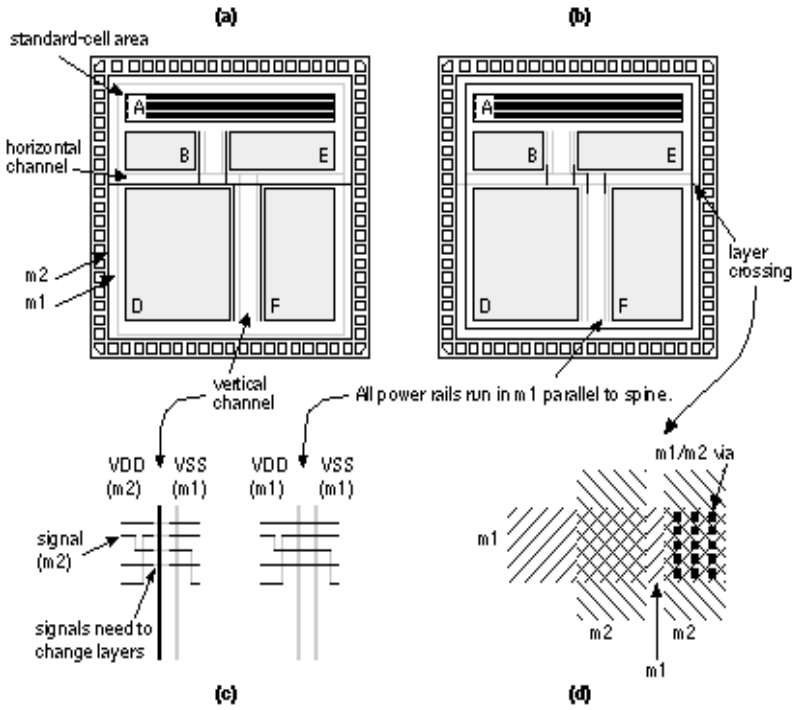



# Plan

- **Introduction**
- **Le plan de masse**
  - ***Le routage bloc***
  - ***Les entrées – sorties***
  - ***Les alimentations***
  - ***La distribution d'horloge***
- **Le placement**
- **Le routage**
  - ***Routage global***
  - ***Routage détaillé***

# Distribuer l'alimentation

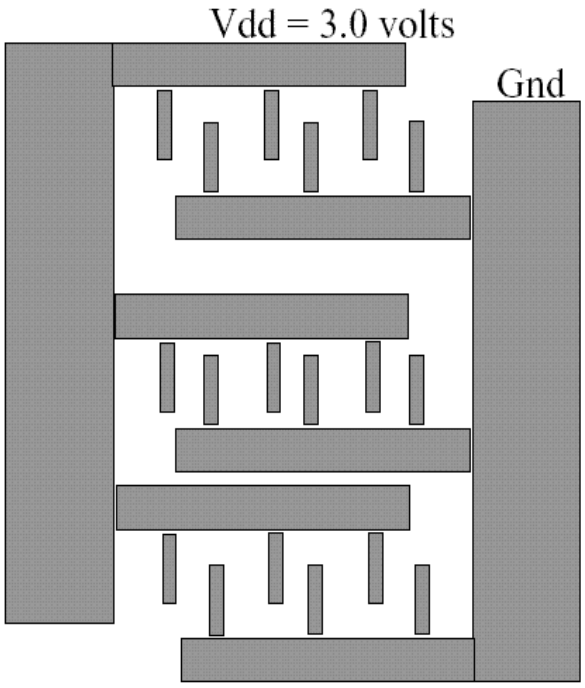
- La distribution d'alimentation est critique
  - *Taille des rails*
  - *Stratégie de circulation des rails*






# Exemple de distribution en peigne

- Des troncs de 5 mm de long
- 50 branches de 200 µm de long par tronc
- 20 cellules par branche, 1 mA par cellule
- Résistance d'une ligne de 1 µm de large : 30 mΩ / µm
- Intensité max : 1 mA / µm
- Chute de tension max : 0,1 V



# Exemple de distribution en peigne (suite)

- Largeur de chaque branche : (20 * 1 mA) / 1 mA $\mu m^{-1}$ = 20 µm
- Largeur des troncs : (50 * 20 mA) / 1 mA $\mu m^{-1}$ = 1000 µm
- Résistance d'un tronc : 30 mΩ * 5000 µm / 1000 µm = 150 mΩ
- Chute de tension due au tronc : $\delta V = RI$ = 150 mΩ * 50 * 20 mA = 150 mV
- Résistance d'une branche : 30 mΩ * 200 µm / 20 µm = 300 mΩ
- Résistance d'un segment de branche : 300 mΩ / 20 = 15 mΩ
- Chute de tension due à un segment : $\delta V = RI$ = 15 mΩ * 1 mA = 15 µV
- Chute de tension due à une branche : 15 µV + 30 µV + 45 µV + … (20 fois) = 15 * (1 + 2 + … + 20) µV = 15 * 20 * 21 / 2 µV = 3150 µV
- Chute de tension au bout d'une branche : 153,15 mV > 0,1 V



# Plan

# Les objectifs, deux stratégies

- **Difficultés :**
  - ***Limiter la latence***
  - ***Réduire le déphasage (« skew »)***
  - ***Supporter les appels de courant***
  - ***Limiter la consommation***
- **Deux stratégies**
  - ***L'arrête de poisson***
    - **Principalement utilisée dans les mers de portes et les FPGA**
    - **Utilise plusieurs plots**
    - **Consomme beaucoup**
    - **Gourmande en surface de métal**
    - **Assez facile à concevoir**
  - ***L'arbre d'amplification***
    - **Difficile à équilibrer**
    - **Un seul plot**
    - **Consomme beaucoup … mais moins que l'arrête de poisson**
    - **Surface réduite**



# Exemple d'arrête de poisson

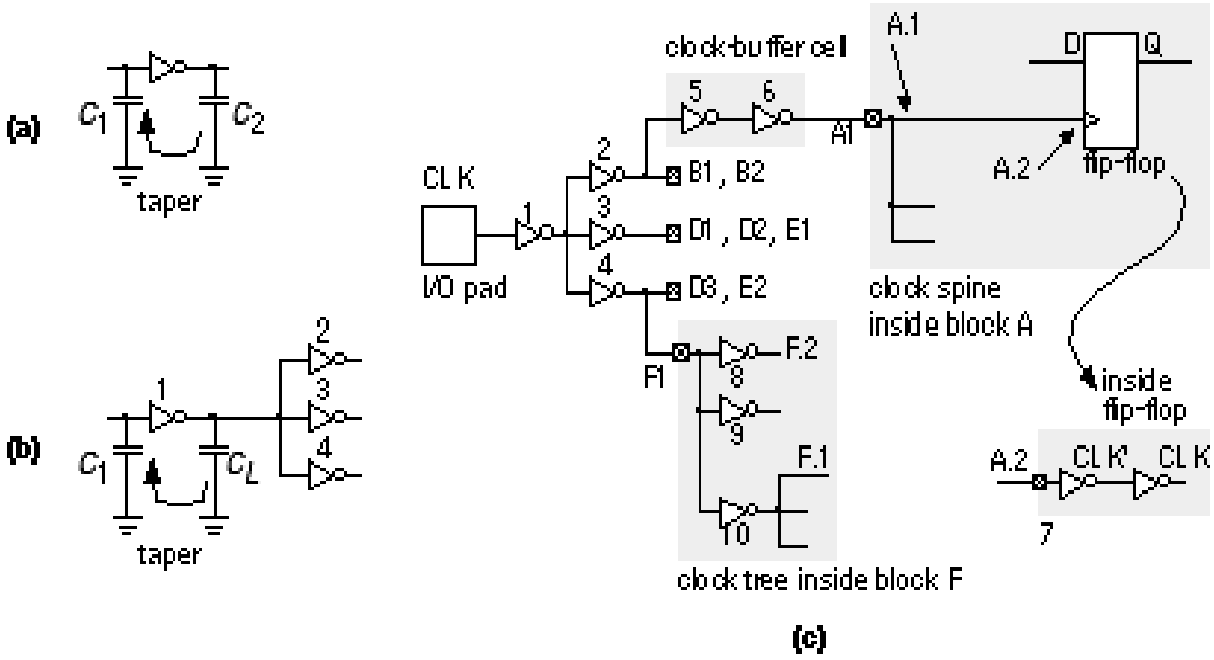
- **Le délai à travers une chaîne d'amplificateurs est minimal lorsque les rapports des capacités d'entrée et de sortie est de 2,7.**
- **Le circuit contient 40000 bascules D**
- **La capacité d'entrée d'horloge des bascules D est de 0,025 pF**
- **La fréquence de fonctionnement est de 200 MHz**
- **VDD = 3,3 V**
- **Taille du circuit 20 mm x 20 mm**
- **L'arrête est constituée de 200 lignes de 20 mm**
- **La capacité linéaire est de 2 pF cm$^{-1}$**

# Exemple d'arrête de poisson (suite)

- La capacité totale de l'arrête est de 200 * 2 cm * 2 pF $cm^{-1}$ = 800 pF
- La capacité des bascules est de 40000 * 0,025 pF = 1000 pF
- Si la capacité d'entrée du premier étage de la chaîne d'amplificateurs est de 0,025 pF alors il faut ln(1800 / 0,025) = 11.18 = 12 étages
- La consommation de l'arbre est : P = f * C * $VDD^2$ = 200 MHz * 1800 pF * $(3,3\ V)^2$ = 3,92 W
- Si le temps de transition en sortie de la chaîne est de 0,1 ns, le pic de courant est : I = C * dV / dt = 1800 pF * 3,3 V / 0,1 ns = 59,4 A !!!



# La distribution d'horloge en arbre



# Plan

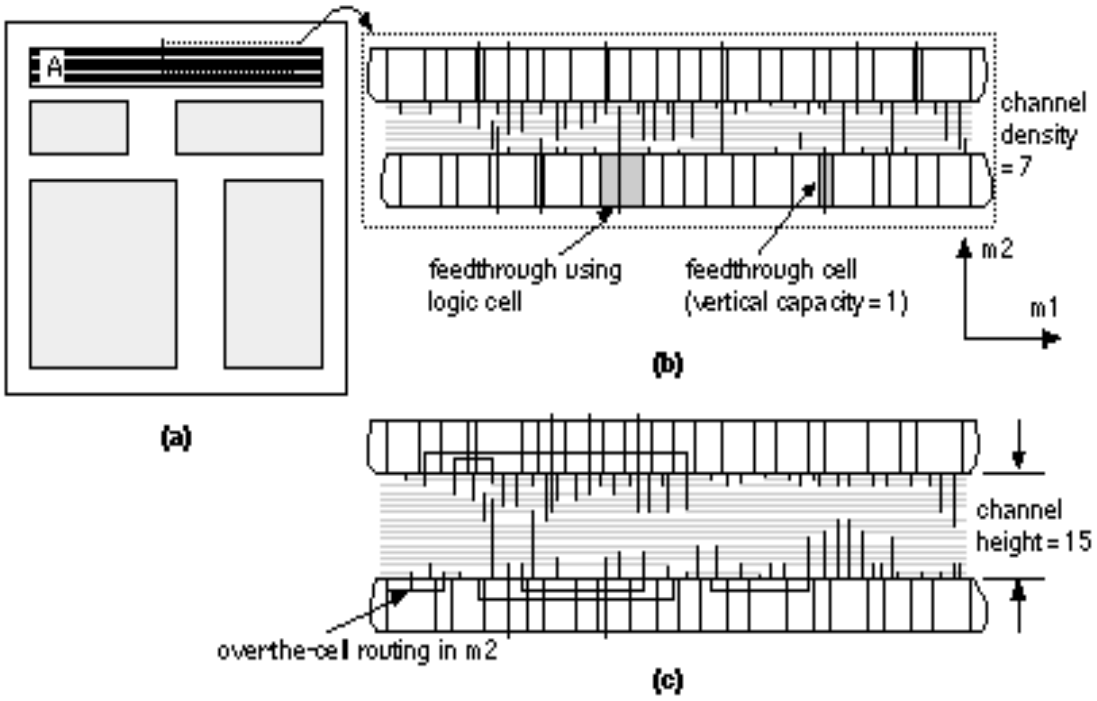
- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*



# Le placement

- Intervient après le plan de masse
- Concerne uniquement les zones de cellules standard
- Plus facile à automatiser que le plan de masse
  - *Les cellules sont toutes de la même hauteur*
  - *Canaux rectangulaires et parallèles entre eux*
  - *Zones rectangulaires*
- Après le placement on peut prédire les capacités et résistances parasites et retourner aux étapes de synthèse

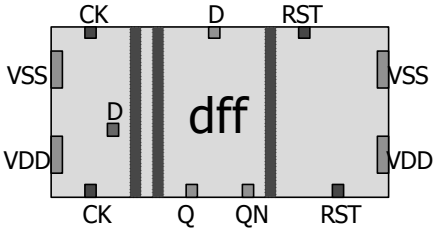
# Cellules standard et canaux






# Les cellules standard

- Les cellules ont en général leurs entrées – sorties sur la périphérie
- Un même port logique peut avoir plusieurs ports physiques associés
- Les alimentations sont traversantes et horizontales (constitution d'un rail par aboutement)
- La longueur des cellules dépend de leur complexité
- Les cellules utilisent en général un ou deux niveaux de métal



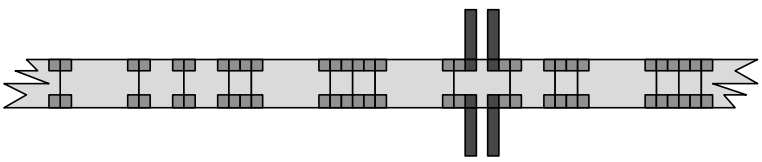
# On place pour router

- Le routage des cellules utilise deux ou trois niveaux de métal
- On peut router à travers les rangées de cellules à condition
  - *D'emprunter les traces prévues dans les cellules (« feedthrough »)*
  - *D'ajouter des cellules de transparence (« feedthrough cells »)*
  - *D'utiliser des niveaux de métal différents de ceux utilisés dans les cellules*
- On route en général avec un métal les lignes horizontales et avec un deuxième les lignes verticales. Si on a un troisième métal il est également utilisé en horizontal
- On appelle capacité du canal le nombre de lignes parallèles aux rangées de cellules qu'il peut accueillir



# On ajoute des cellules

- Des « *feedthrough cells* »
- Des cellules d'écartement qui permettent d'avoir des rangées de même longueur
- Des cellules d'extrémité qui servent à la connexion aux alimentations
- Des cellules d'alimentation lorsque les rangées sont trop longues afin d'éviter les chutes de tension



# Objectifs

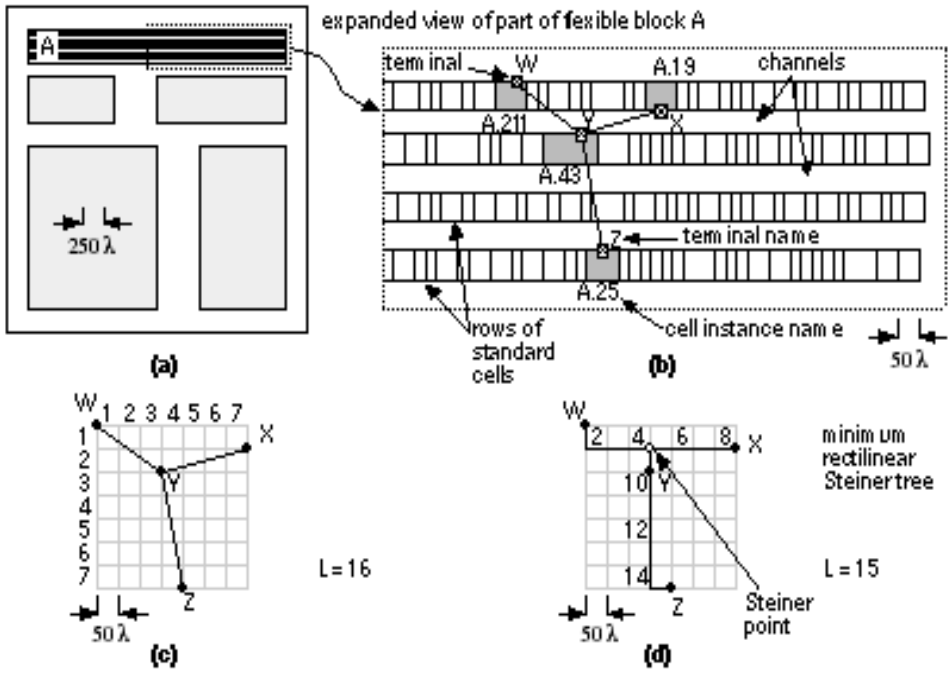
- **Objectifs abstraits :**
  - ***Garantir la « routabilité »***
  - ***Minimiser les délais des nœuds critiques***
  - ***Atteindre la plus grande densité possible***
  - ***Minimiser la consommation***
  - ***Minimiser la diaphonie***
- **Objectifs concrets :**
  - ***Minimiser la longueur total des interconnexions***
  - ***Atteindre les contraintes temporelles sur les nœuds critiques***
  - ***Minimiser les congestions***



# Mesure de qualité d'un placement

- **On veut mesurer la qualité d'un placement**
- **Chaque nœud peut être représenté sous forme d'un arbre**
- **Les arbres de Steiner minimisent la longueur totale des interconnexions**
- **Trouver l'arbre rectilinéaire minimal de Steiner est un problème NP-complet coûteux en temps de calcul**
- **On utilise des approximations rapides de la longueur des arbres de Steiner**

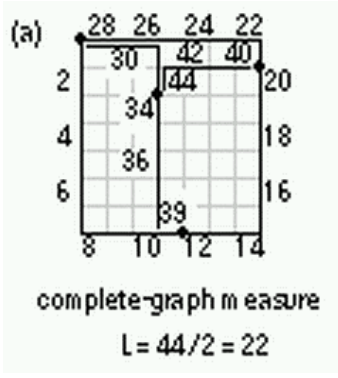
# Les arbres de Steiner





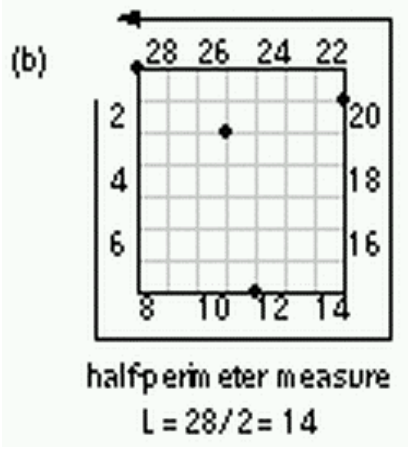

# L'approximation du graphe complet

- Dans un graphe complet il existe une connexions entre deux broches quelconques
- On peut réaliser un graphe complet avec n * (n-1) connexions (où n est le nombre de broches à connecter). Chaque connexion entre deux broches quelconques est alors doublée.
- Le plus petit graphe complet contient donc n * (n – 1) / 2 connexions
- Il suffit de (n – 1) connexions pour relier toutes les broches (théorème des piquets et des intervalles), soit n / 2 fois moins
- On peut donc approcher la longueur de Steiner par la longueur du graphe complet divisée par n / 2



# L'approximation du demi-périmètre

- Elle approche la mesure de Steiner par la moitié du périmètre de l'enveloppe rectangulaire
- Pour des nœuds à 2 ou 3 broches (50 % d'un circuit, en moyenne) c'est la mesure de Steiner
- Pour des nœuds plus complexes elle est un peu optimiste



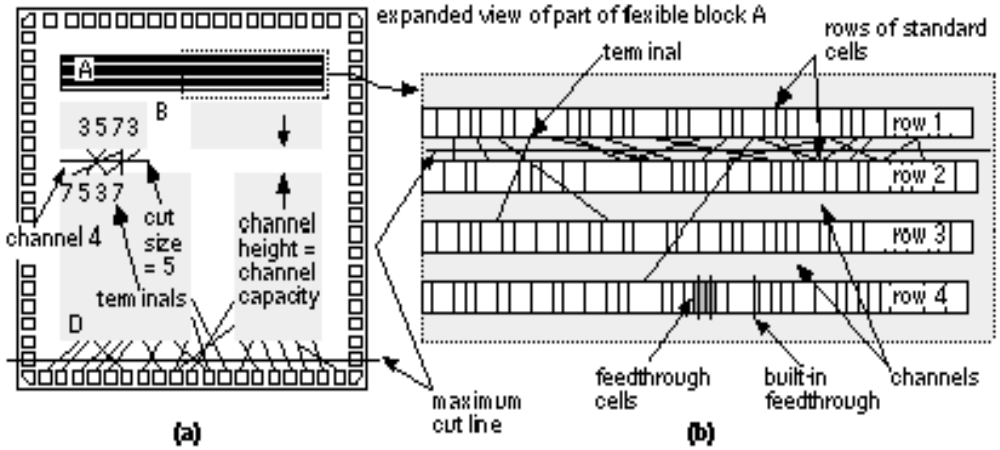



# Le rôle des approximations

- Utiliser l'une des deux approximations pour le placement d'un circuit c'est chercher à minimiser la somme des approximations
- Peut importe que les approximations soient mauvaises si elles sont bien corrélées aux longueurs mesurées après placement et routage
- On peut améliorer les approximations en utilisant les résultats expérimentaux
- La mesure de Steiner pose plusieurs problèmes
  - *Elle n'est pas forcément atteinte par le routeur*
  - *Elle ne correspond pas forcément aux délais minimaux*
  - *Elle ne garantit pas la routabilité car elle n'optimise pas la congestion*

# Mesure de congestion

- On mesure la congestion par les mesures de longueur des coupes






# Critères d'optimisation

- Un placement optimal au sens de la mesure de Steiner et de la congestion risque de pénaliser des nœuds point à point
- Un nœud point à point peut être critique
- Les placements optimaux en délais d'interconnexion ne peuvent utiliser qu'une approximation très simple
- Un routage optimal en délai n'est pas forcément optimal au sens de la mesure de Steiner
- Le placement optimal en délai est un problème très difficile

# Deux passes

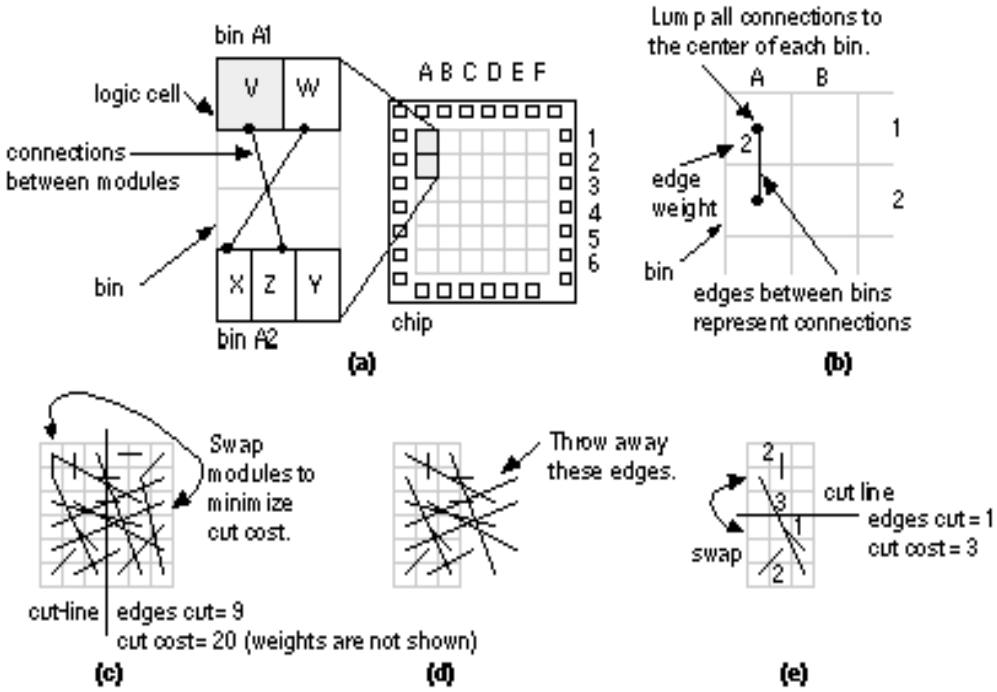
- On construit un placement
  - *En tenant éventuellement compte des germes fournis par le concepteur*
  - *Par une méthode de minimisation de coupe*
  - *Ou par la méthode des valeurs propres*
- On l'optimise par une méthode itérative
  - *Recuit simulé*
  - *Ou autre méthode d'optimisation*



# La méthode de minimisation de coupe

- On divise la zone en boîtes
- On découpe en deux parties
- On minimise la fonction de coût des connexions coupées en échangeant des boîtes entre les deux parties
- On recommence avec chacune des moitiés …
- Jusqu'à ce que chaque cellule soit placée
- On peut commencer avec des grosses boîtes et finir avec des boîtes unitaires

# La méthode de minimisation de coupe (suite)






# L'algorithme des valeurs propres

- On cherche à minimiser la fonction de coût

$$f = \frac{1}{2}\sum_{i,j=1}^{n} c_{ij} d_{ij}^{2}$$

où C = $[c_{ij}]$ est la matrice de connexion (éventuellement pondérée) et $d_{ij}$ la distance euclidienne entre les centres des cellules *i* et *j*

- C'est un problème qu'on peut résoudre mathématiquement à condition de le simplifier un peu
- On obtient une solution approchée qui peut être améliorée par une méthode itérative

# Les méthodes d'optimisation itératives

- **Il faut :**
  - ***Un critère de sélection des cellules que l'on va tenter de déplacer***
  - ***Une mesure de qualité pour accepter ou refuser la modification***
- **Elles s'apparentent à l'auto-répartition des rugbymen à élastiques**
- **Elles permettent de gagner 10 % à 20 % sur la longueur totale des connexions**

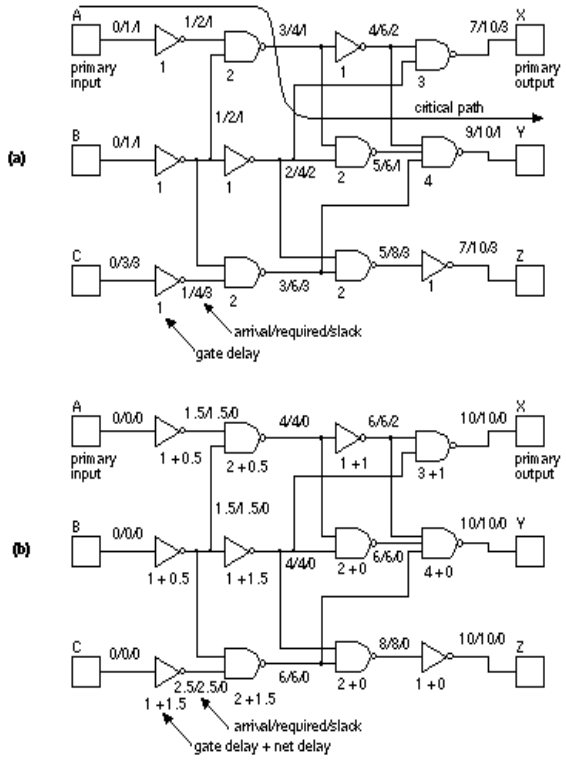


# Le placement contraint

- **Pour contraindre le placement en temps n'importe quelle méthode convient à condition d'être capable de pondérer les connexions**
- **Appliquer les bons poids aux bonnes connexions est un problème très compliqué**
- **Les concepteurs préfèrent spécifier des chemins critiques que des poids d'équipotentielles**
- **L'algorithme de l'annulation de marge (« *zero-slack* ») est souvent utilisé.**

# L'algorithme d'annulation de marge

- On part d'une situation à délais d'interconnexions nuls
- On calcule pour chaque nœud la date d'arrivée, la date d'arrivée requise et la marge
- On distribue des délais d'interconnexion sur les nœuds pour annuler les marges



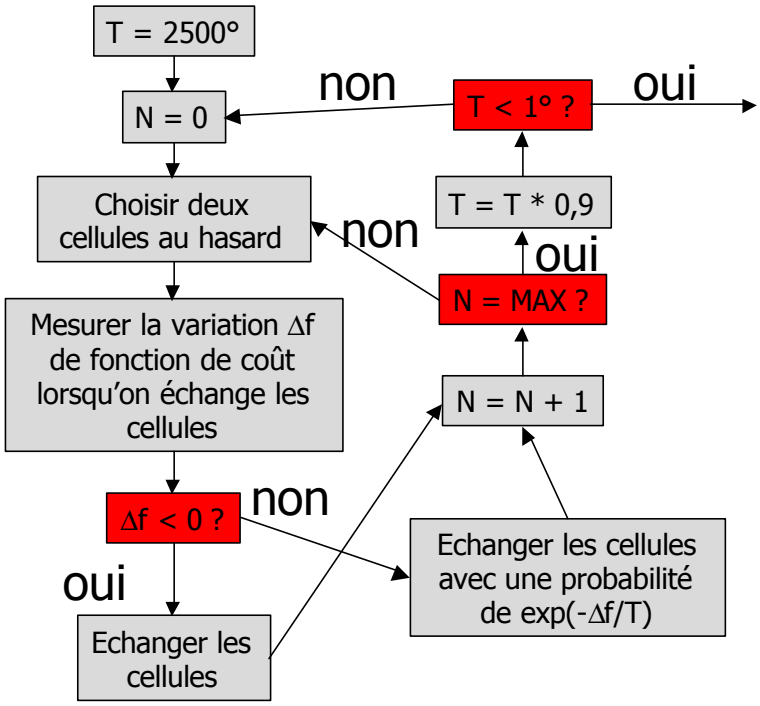



# L'algorithme d'annulation de marge (suite)

- Ajuster les poids des équipotentielles ne complique pas l'algorithme de placement
- L'algorithme d'annulation des marges simplifie le problème du placement optimisé en délai mais il le sur-contraint
- Il existe des algorithmes basé sur les délais de chemins mais ils sont trop complexes (pour l'instant)

# Exemple de recuit simulé

- 1) avec des cellules de rangées différentes
- 2) avec des cellules de la même rangée
- 3) avec des cellules voisines
- 4) symétries sur une seule cellule






# Le test « *scan path* »

- Dans une stratégie de test « *scan path* » on ajoute un multiplexeur devant chaque bascule
- L'ordre dans lequel on connecte les chaînes de « *scan* » est indifférent au sens du test
- Pour éviter que le test ne modifie le placement
  - *On insère le matériel de test sans connecter les chaînes*
  - *On effectue un placement dont on extrait les positions des bascules*
  - *On connecte les chaînes à partir des positions des bascules*
  - *On refait un placement*

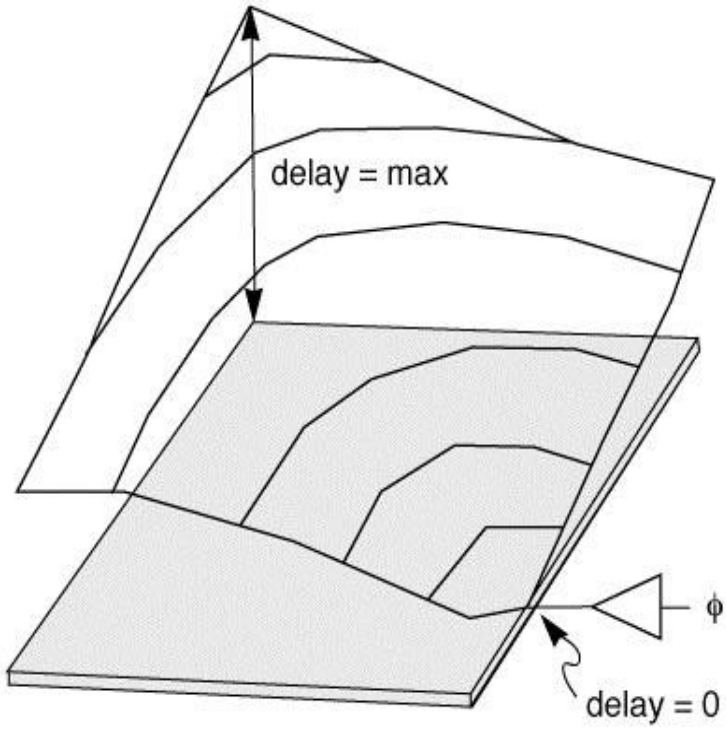
# L'arbre d'horloge

- La synthèse d'un arbre d'horloge ne peut plus se faire avant le placement
- On synthétise et on place le circuit en faisant l'hypothèse d'une horloge parfaite
- On insère des amplificateurs d'horloge dans le placement constitué en tenant compte des parasites
- On route



# Le « skew », ennemi ou allié

- On peut tirer parti du « skew » introduit par l'arbre d'amplification d'horloge
- Attention, c'est tentant mais c'est
  - *Très dangereux*
  - *Difficilement modifiable*



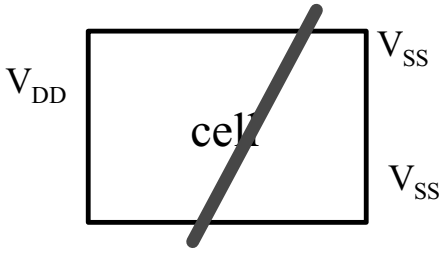
# Quelques règles de placement

- Faire un plan de câblage (quels niveaux, par où on passe) pour les nœuds critiques Fusionner les petits blocs : une porte NAND toute seule dans un coin peut rendre le travail beaucoup plus compliqué
- Concevoir un câblage simple ; s'il a l'air compliqué, il est compliqué
- Faire très tôt un plan de câblage séparé pour les alimentations et les horloges ; ce sont des nœuds critiques
- Essayer de faire un plan de câblage planaire quand c'est possible et si ça ne coûte pas trop cher

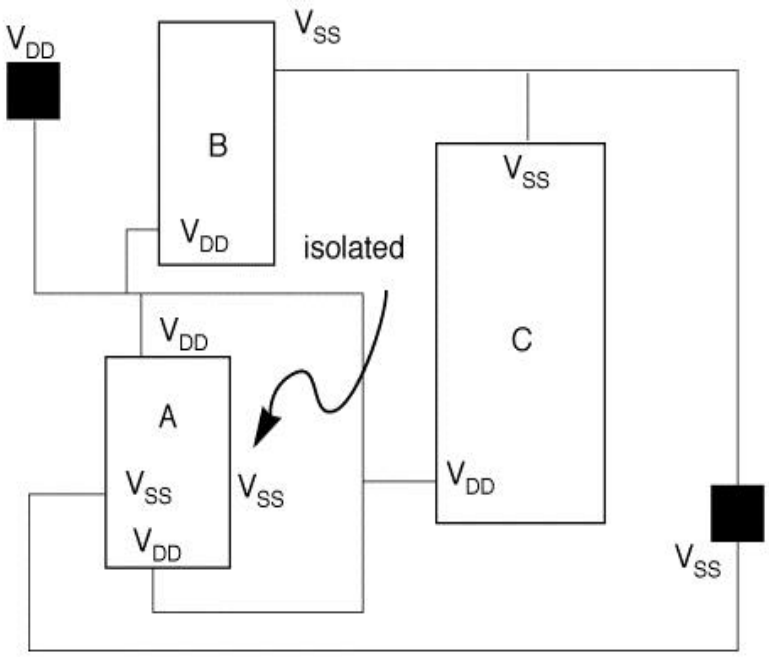


# Alimentation planaire

- Si la distribution d'alimentation est planaire elle est simplifiée
- Pour avoir une organisation planaire chaque bloc doit pouvoir être coupé en deux par une ligne droite qui sépare alimentations et masses
- Si le plan de masse a placé toutes les alimentations du même côté alors il existe un routage bloc qui évite alors de croiser alimentations et masses



# Alimentation planaire (suite)






# Plan

- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*

# Deux étapes de deux phases

- **Les opérations de routage sont en général séparées en deux étapes :**
  - ***Le routage global***
  - ***Le routage final (ou détaillé)***
- **Chacune de ces deux opérations est également la succession de deux phases :**
  - ***Le routage cellules***
  - ***Le routage bloc***

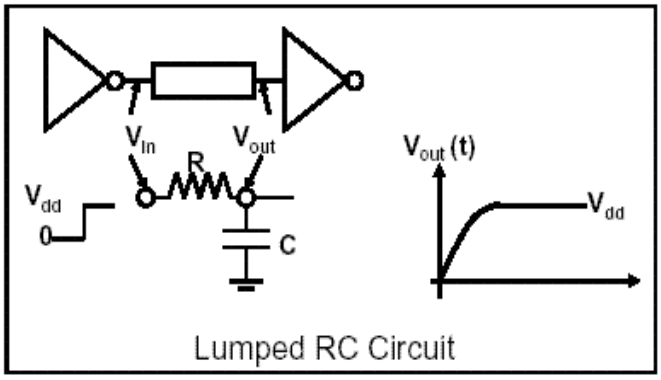


# Le routage global

- **A pour point de départ un placement fini**
- **Consiste à sélectionner les canaux à emprunter pour construire chaque nœud du circuit**
- **A pour objectifs**
  - ***De minimiser la longueur totale d'interconnexions***
  - ***D'assurer que le routage final pourra se faire***
  - ***De minimiser les délais des chemins critiques***
- **Le placement est parfaitement connu**
  - ***Le routeur global peut utiliser des estimations des délais d'interconnexion plus précises que celles utilisées lors du plan de masse ou du placement***
  - ***Le modèle le plus fréquemment utilisé est celui de la constante d'Elmore***

# Les délais RC, modèle agrégé

- $\tau = RC$ est la constante de temps
- Le délai est en général de la forme $T_p = \alpha\tau$

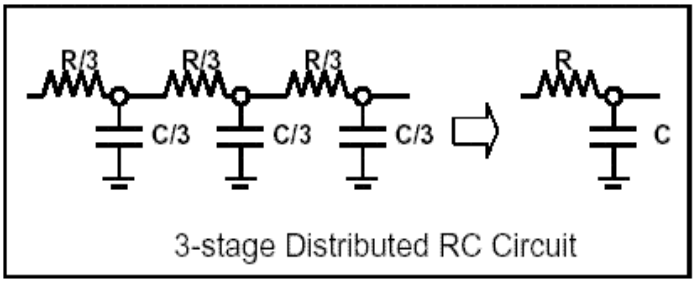


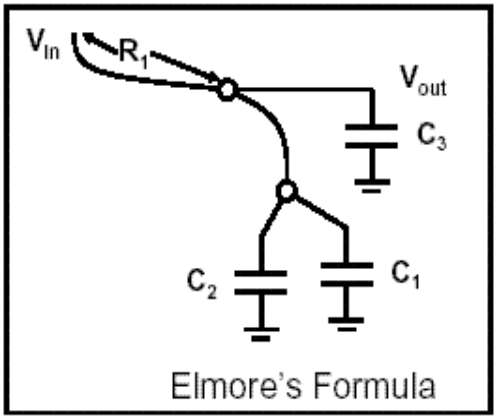

$$V(t) = V_{DD} \exp(- t / RC)$$
$$T_p = t_{50} = ln(1 / 2)\, RC = 0{,}69\, RC$$



# Les délais RC, modèle distribué

- $\tau = 2/3\ RC = 0{,}67\ RC < RC$
- Le modèle agrégé est un pire cas
- La constante de Elmore
  - $C_i$ *est la* i*ème capacité*
  - $R_i$ *est la résistance commune au chemin de l'entrée vers la capacité* i *et au chemin de l'entrée vers la sortie*

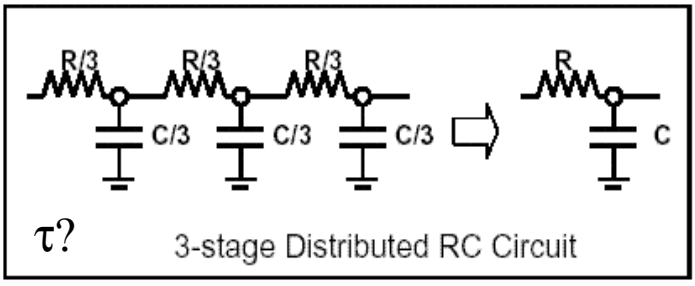



$$\tau = \Sigma_i\, C_i\, R_i$$

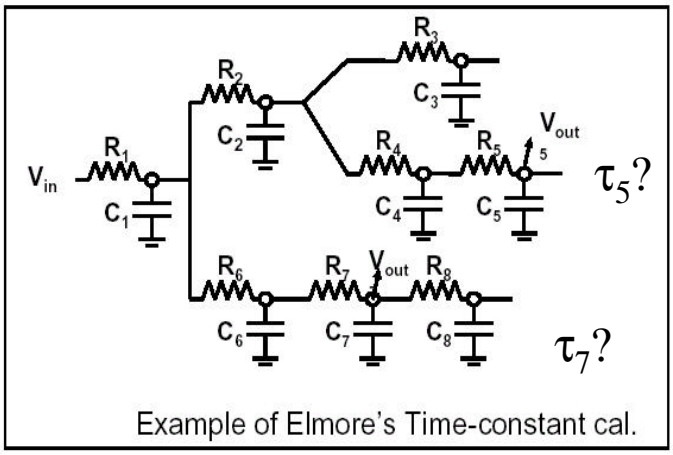
# La constante de Elmore

$$\tau = \frac{R}{3}\frac{C}{3} + 2\frac{R}{3}\frac{C}{3} + 3\frac{R}{3}\frac{C}{3} = 2\frac{RC}{3}$$




$$\tau_5 = R_1C_1 + (R_1+R_2)(C_2+C_3) + (R_1+R_2+R_4)C_4 + (R_1+R_2+R_4+R_5)C_5 + R_1(C_6+C_7+C_8)$$

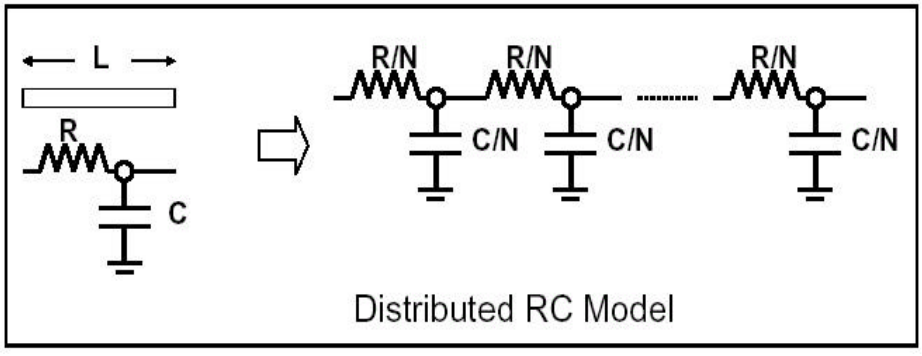
$$\tau_7 = R_1(C_1+C_2+C_3+C_4+C_5) + (R_1+R_6)C_6 + (R_1+R_6+R_7)(C_7+C_8)$$






# Modèle distribué

- **Constante de temps τ du modèle distribué d'une ligne de longueur *L*, de résistance *R* et de capacité *C* ?**
- **Limite lorsque *N* tend vers l'infini (on pose *r* et *c* résistances et capacités par unité de longueur)**




$$\tau = \frac{(1+2+3+\ldots+N)RC}{N^2} = \frac{N(N-1)RC}{2N^2} = \frac{(N-1)RC}{2N}$$

$$\tau = \frac{RC}{2} = rc\frac{L^2}{2}$$

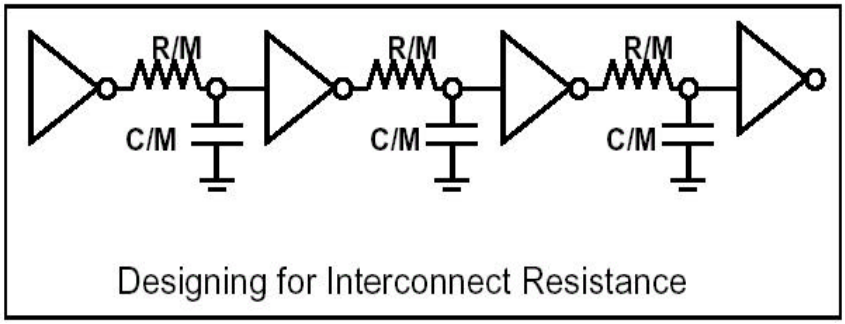
# Validité de la méthode de Elmore

- Le modèle distribué a une constante de temps de 0,5 *RC* d'après la méthode de Elmore
  - *C'est la moitié du modèle agrégé*
  - *La constante de temps augmente comme le carré de la longueur*
- Des simulations SPICE montrent $T_p = 0{,}38\ RC$ contre $T_p = \alpha\tau = (0{,}69)\ (0{,}5\ RC) = 0{,}35\ RC$ avec la méthode de Elmore



# Optimisation

- Comment optimiser le temps de propagation le long d'une ligne RC ?
  - $T_p = 0{,}38\ RC / M + M\ T_{pbuf}$
  - $\delta\ Tp / \delta\ M = -\ 0{,}38\ RC / M^2 + T_{pbuf}$
  - $M_{opt} = (0{,}38\ RC / T_{pbuf})^{1/2} = L\ (0{,}38\ rc / T_{pbuf})^{1/2}$



Designing for Interconnect Resistance

# Optimisation (suite)

- **Le résultat précédent est faux**
  - ***Il suppose que le temps de propagation des « buffers » est indépendant de M***
  - ***Il ignore les capacités d'entrée des « buffers »***
- **Un synthétiseur d'arbre d'amplification d'horloge utilise un calcul semblable mais plus précis**

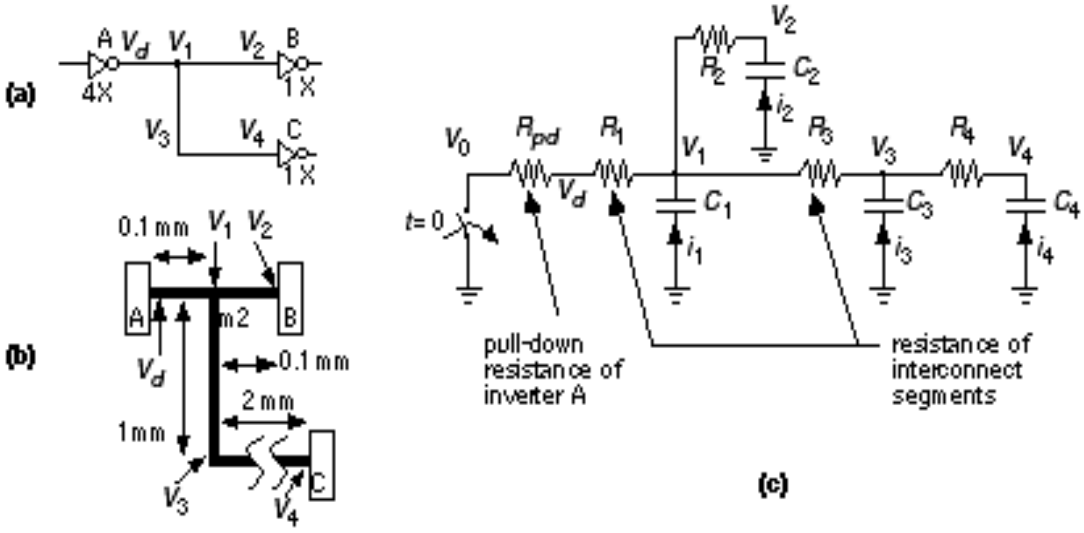


# Compromis précision - complexité

- **Elmore fait des hypothèses simplificatrices :**
  - ***La pente d'entrée est infinie***
  - ***Délai mesuré à partir du changement de valeur à l'entrée des portes***
  - ***Délai proportionnel à la constante de temps d'une courbe exponentielle qui approche la courbe réelle***
  - ***Les interconnexions sont modélisées par des résistances et des capacités discrètes***
- **Ces hypothèses introduisent des erreurs**
- **Le routeur pourrait utiliser des estimations plus précises mais**
  - ***Le temps de calcul risquerait de devenir génant***
  - ***Le routeur ne peut de toute façon pas atteindre une précision absolue car il ignore***
    - **Quels métaux seront utilisés lors du routage final**
    - **Combien de changements de niveaux (vias) seront nécessaires**

# Calcul de « *skew* »

- Exemple de calcul de « *skew* » par la méthode de Elmore






# Calcul de « *skew* » : données

- Résistance de *m*2 = 50 mΩ / carré
- Capacité de *m*2 (largeur min) = 0,2 pFmm$^{-1}$
- Délai de l'inverseur 4x = 0,02 ns + 0,5 $C_L$ ns ($C_L$ en picofarads)
- Délais mesurés à 50 % de l'excursion
- Largeur minimum de *m*2 = 3 λ = 0,9 µm
- Capacité d'entrée des inverseurs = 0,02 pF

# Calcul de « *skew* » : résultats

- $t_2 = (R_{pd} + R_1)(C_1 + C_3 + C_4) + (R_{pd} + R_1 + R_2)(C_2 + C_i)$
- $t_4 = (R_{pd} + R_1)(C_1 + C_2) + (R_{pd} + R_1 + R_3)\, C_3 + (R_{pd} + R_1 + R_3 + R_4)(C_4 + C_i)$
- $\Delta t = 0{,}69\, t_4 - 0{,}69\, t_2 = 0{,}69\, (t_4 - t_2) = 0{,}69\, (R_3\, C_3 + (R_3 + R_4)(C_4 + C_i) - R_2\, (C_2 + C_i)) = 0{,}69\, \rho\, \gamma\, (L_3^2 + (L_3 + L_4)\, L_4 - L_2^2 + (L_3 + L_4 - L_2)\, L_i)$
- $\rho = 50.10^{-3} / 0{,}9\ \Omega\mu^{-1}$ ; $\gamma = 0{,}2.10^{-15}\ F\mu^{-1}$ ; $L_i = 100\ \mu$
- $\Delta t = 0{,}69.10^{-17} / 0{,}9\ (10^6 + 6.10^6 - 10^4 + 2{,}9.10^5) \cong 56$ ps
- **Que trouverait-on avec la méthode agrégée ?**



# Optimiser des chemins

- **Optimiser en délai un chemin et un nœud sont deux choses différentes**
  - ***Le chemin qui minimise le délai entre deux points d'une même équipotentielle n'est pas forcément le chemin qui minimise la longueur totale de l'équipotentielle***
- **On ne peut plus utiliser d'approximations grossières (demi-périmètre)**
- **De nombreuses méthodes de routage global sont tout de même basées sur les problèmes d'arbres dans les graphes (voir arbres de Steiner)**

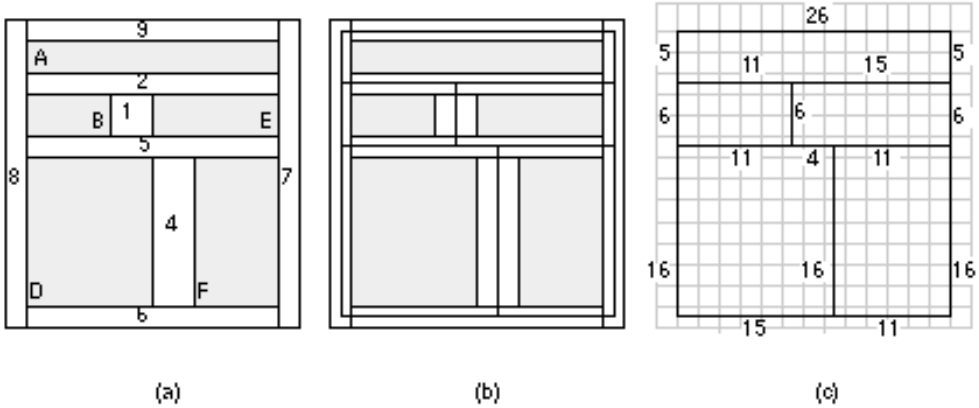
# Routage séquentiel et hiérarchique

- **Routage séquentiel**
  - *Pour chaque nœud on calcule le plus court chemin en empruntant uniquement les canaux de routage utilisables*
  - *Le routage indépendant de l'ordre*
    - Ne tient pas compte de la congestion (comme si les canaux de routage étaient de capacité infinie)
    - Une seconde passe déplace certaines connexions d'un canal congestionné vers un autre moins chargé
  - *Le routage dépendant de l'ordre*
    - Considère la congestion en cours de construction
  - *Une optimisation itérative améliore le résultat*
- **Routage hiérarchique**
  - *Partitionne le circuit en niveaux de hiérarchie*
  - *Route les niveaux hiérarchiques les uns après les autres*
  - *On peut router*
    - Du plus haut niveau vers le plus bas (« *top-down* »)
    - Du plus bas niveau vers le plus haut (« *bottom-up* »)



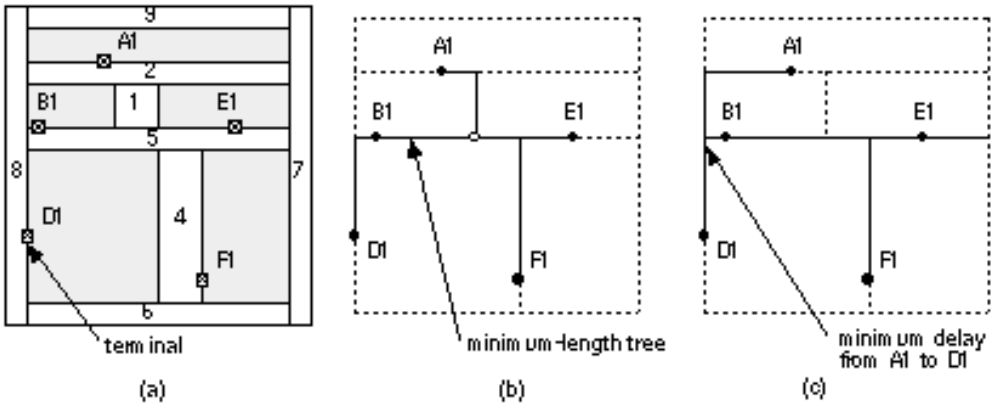
# Le routage bloc

- **Le routeur ne peut emprunter que les canaux**
- **On peut attribuer un poids aux canaux en fonction de leur longueur**



# Longueur vs délai

- Le délai minimum entre deux points n'est pas forcément atteint pour la longueur totale minimale

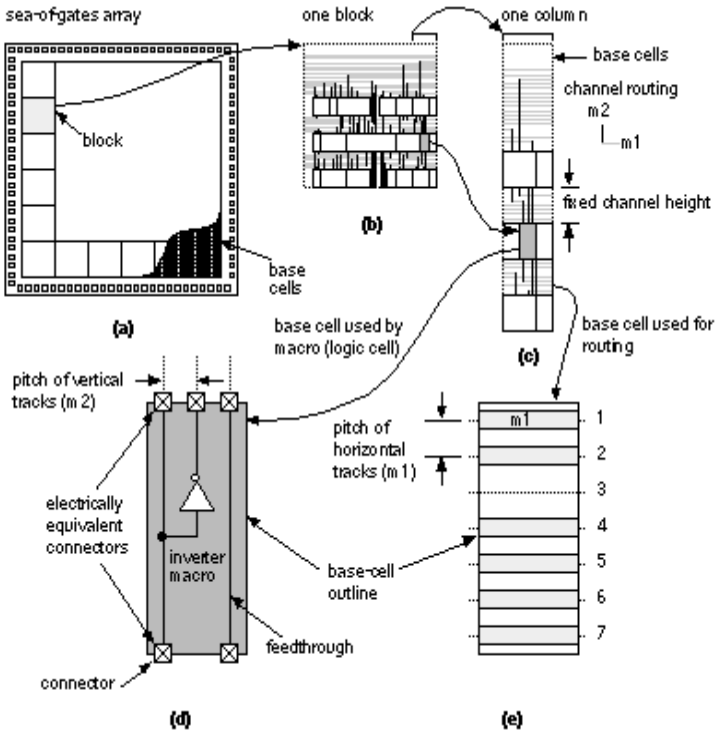





# FPGA et ASIC

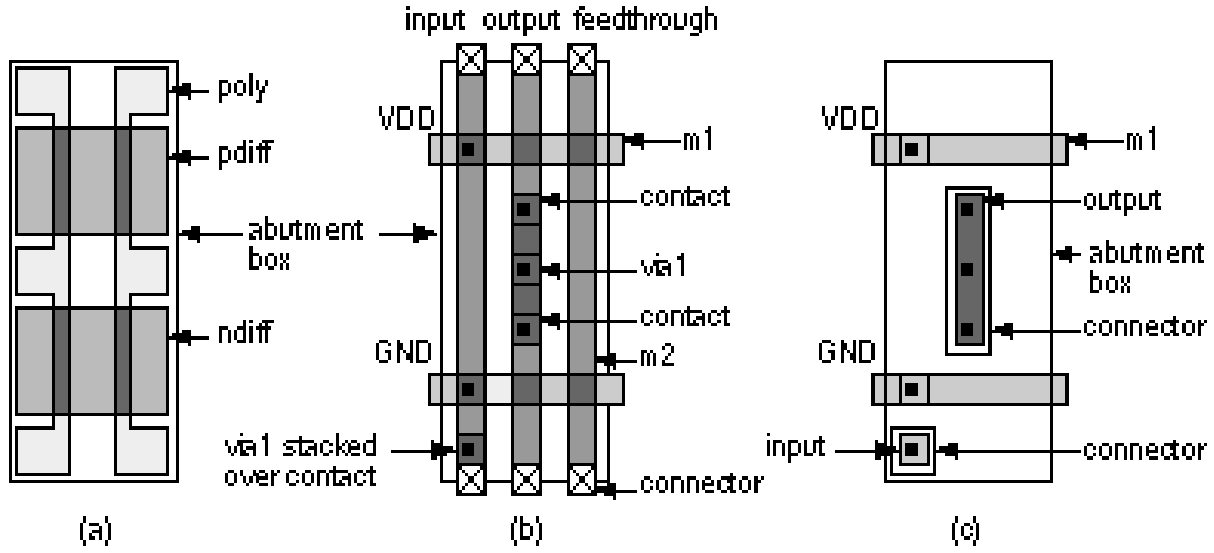
- Dans un FPGA ou un « *gate array* » à canaux la capacité des canaux de routage est fixe
  - *Le routeur peut utiliser librement les pistes puisqu'elles sont là mais …*
  - *… il ne peut pas en ajouter*
- Dans un ASIC en cellules standard ou dans un « *gate array* » sans canaux le routeur peut allouer de nouvelles pistes si nécessaire

# Le routage global, exemple des « *gate array* »






# Les cellules des « *gate array* »

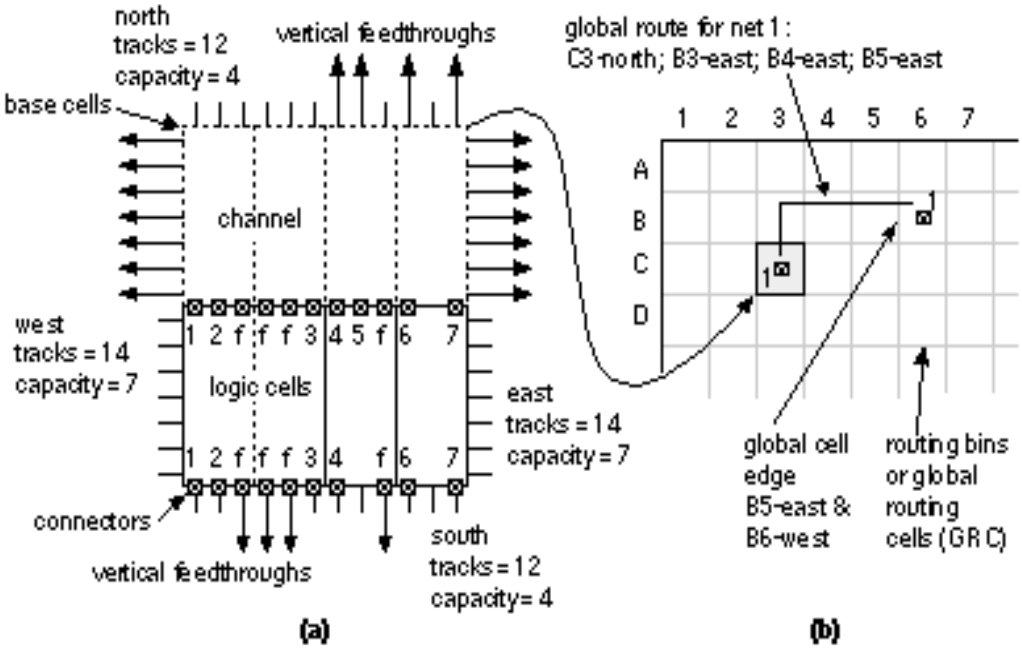


# La mise en boîtes

- **On divise le plan en boîtes non recouvrantes**
  - ***Les grandes boîtes réduisent la complexité du problème de routage***
  - ***Les petites boîtes améliorent la précision des estimations de capacités et résistances de routage***
  - ***Certains routeurs adaptent la taille des boîtes aux difficultés rencontrées***
- **Le routeur maintient les capacités de bord des boîtes et les utilise pour trouver le chemin le plus court entre deux points**
- **Le chemin trouvé est passé au routeur final**



# La mise en boîtes (suite)



# Méthodes contraintes en délai

- Mêmes problèmes que pour le placement
- Arbre minimal de Steiner pas forcément optimal
- Fonction de coût = méthode d'Elmore
- Deux catégories
  - *Basées sur les nœuds*
    - Plus simples
    - Sur-contraintes)
  - *Basées sur les chemins*
    - Plus compliquées
    - Capables d'optimiser A → C en dégradant A → B pour améliorer B → C)



# Le routage global, conclusion

- Les estimations du placeur et celles du routeur global doivent être compatibles
- Il faut s'assurer que son routeur global donne de bons résultats à partir de son placeur
- La compatibilité des outils de CAO ne va pas de soi. C'est un « *vrai* » problème.
- Lorsque le routage global est terminé on peut rétro-annoter vers la synthèse et l'optimisation
- Les RC complètent les estimations de longueur totale que le placeur avait déjà produites
- On s'assure qu'il n'y a pas de mauvaises surprises
- On pratique des optimisations en place
  - *Modification des puissances de sortie des portes*

# Plan

- Introduction
- Le plan de masse
  - *Le routage bloc*
  - *Les entrées – sorties*
  - *Les alimentations*
  - *La distribution d'horloge*
- Le placement
- Le routage
  - *Routage global*
  - *Routage détaillé*

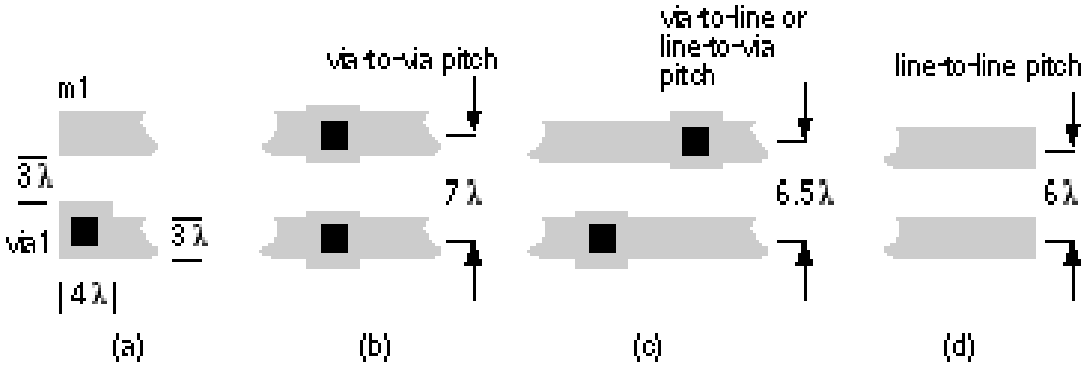


# Le routage détaillé (ou final)

- Alloue des pistes aux nœuds
- Décide des niveaux de métal à utiliser
- Respecte les règles de dessin du fondeur
  - *Distances via – via*
  - *Distances via – métal*
  - *Distances métal – métal*
  - *Empilage de vias et de contacts*
  - *Etc.*

# Choix d'un pas de routage

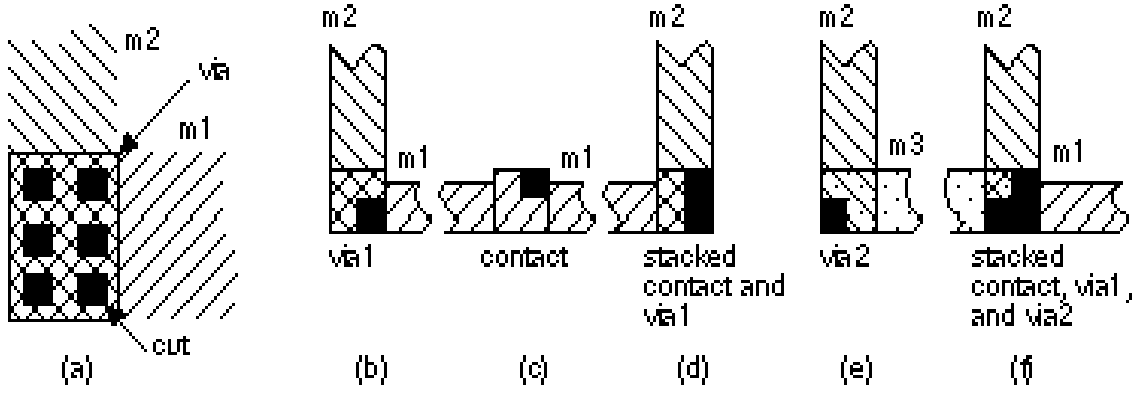
- La distance métal – métal est la plus contraignante
- La distance métal – via limite les possibilités du routeur
- La distance via – via est la moins contraignante et la plus utilisée





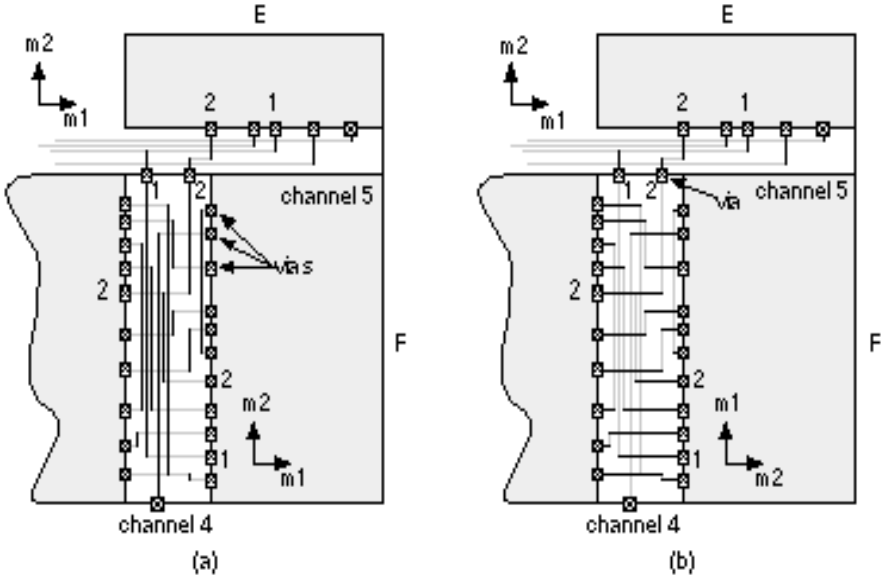

# Vias

- Les gros vias sont parfois plusieurs petits
- La résistance peut être meilleure
- Le procédé de fabrication peut être plus simple



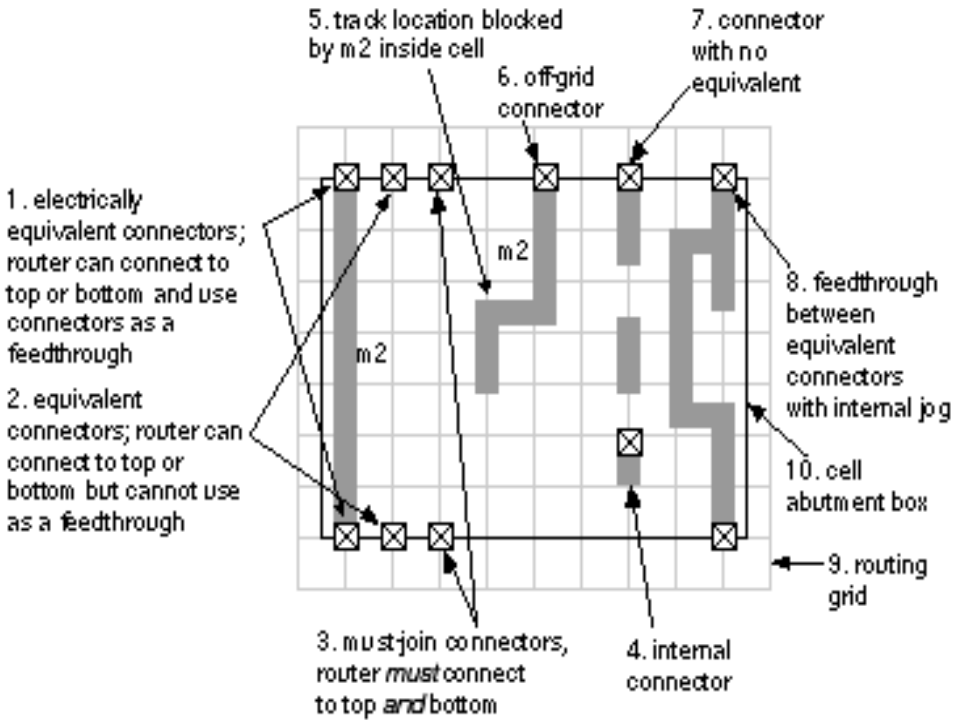
# On peut choisir un métal par direction

- Cela peut poser des problèmes à l'intersection de deux canaux
- Ou à la connexion avec les blocs (métal des connecteurs)

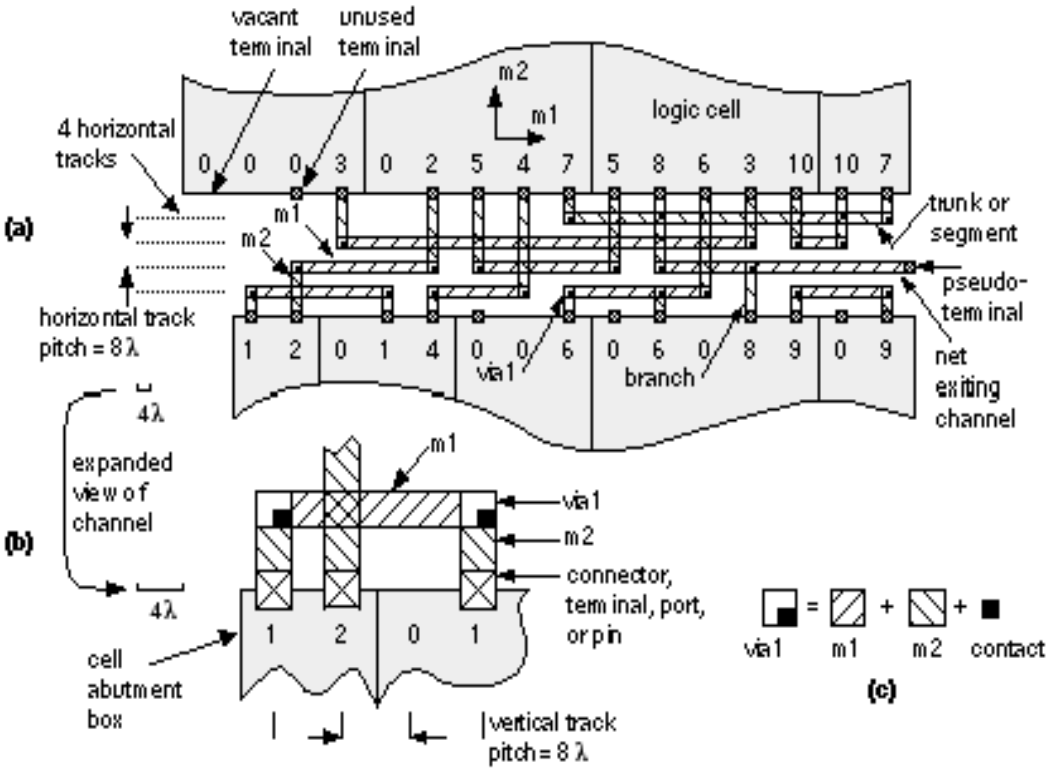





# Fantômes de cellules



# Un nœud peut utiliser plusieurs troncs



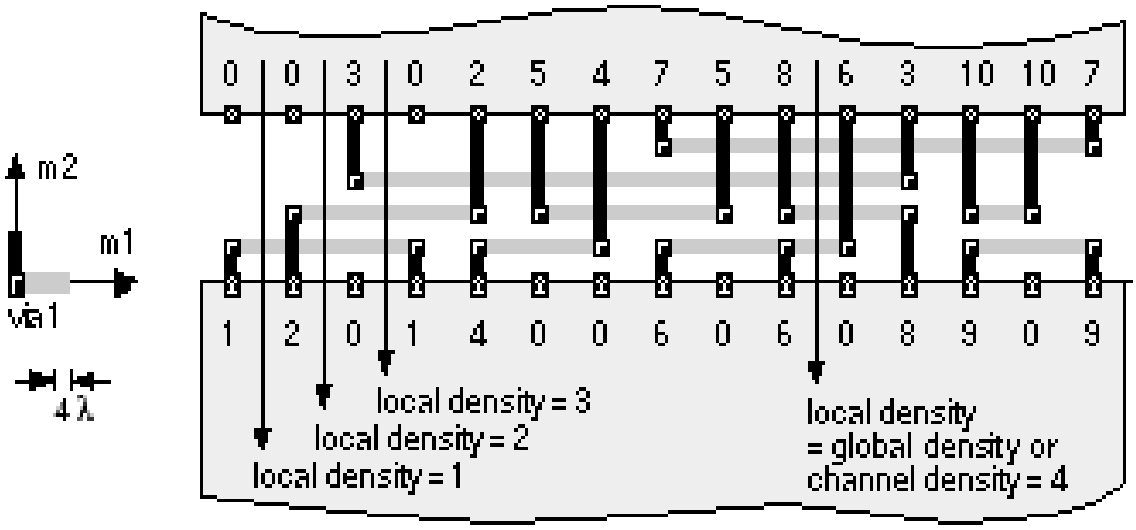



# Les objectifs du routage final

- **Optimiser la longueur totale d'interconnexions et la surface**
- **Optimiser le nombre de changements de niveau de métal par connexion**
  - ***Pour réduire résistances et capacités***
- **Optimiser les délais le long des chemins critiques**
- **Le routeur final ne parvient parfois pas à router**
  - ***Dans les ASIC en cellules standard on peut faire de la place***
  - ***Dans les FPGA et les « gate array » à canaux fixes***
    - **Il faut tout reprendre (plan de masse, placement, routage) …**
    - **… ou choisir un plus gros circuit**

# Mesure de densité de canal

- Deux listes de nœuds, les nœuds 0 sont inutilisés

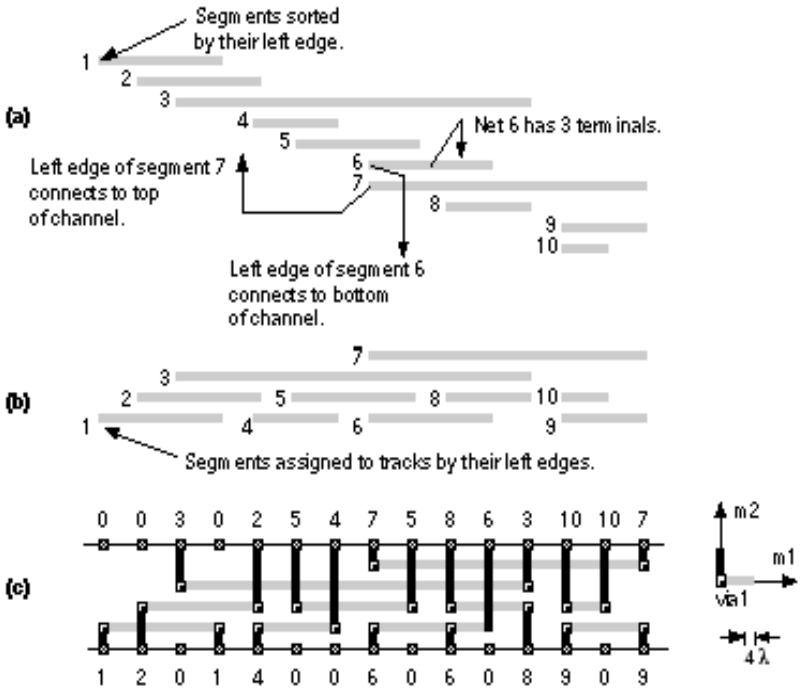





# Routage à deux métaux

- Le problème du routage de canal simplifié et à deux métaux
  - *Un seul segment horizontal par nœud*
  - *L'algorithme du bord gauche (Hashimoto et Stevens 1971)*
    - (1) Trier les nœuds en fonction du bord gauche de leur segment horizontal
    - (2) Allouer la première piste au premier de la liste
    - (3) Chercher le prochain de la liste auquel on puisse allouer la même piste
    - (4) Recommencer 3 jusqu'au blocage
    - (5) Recommencer 2 - 4 jusqu'à la fin
    - (6) Connecter les verticales

# L'algorithme du bord gauche

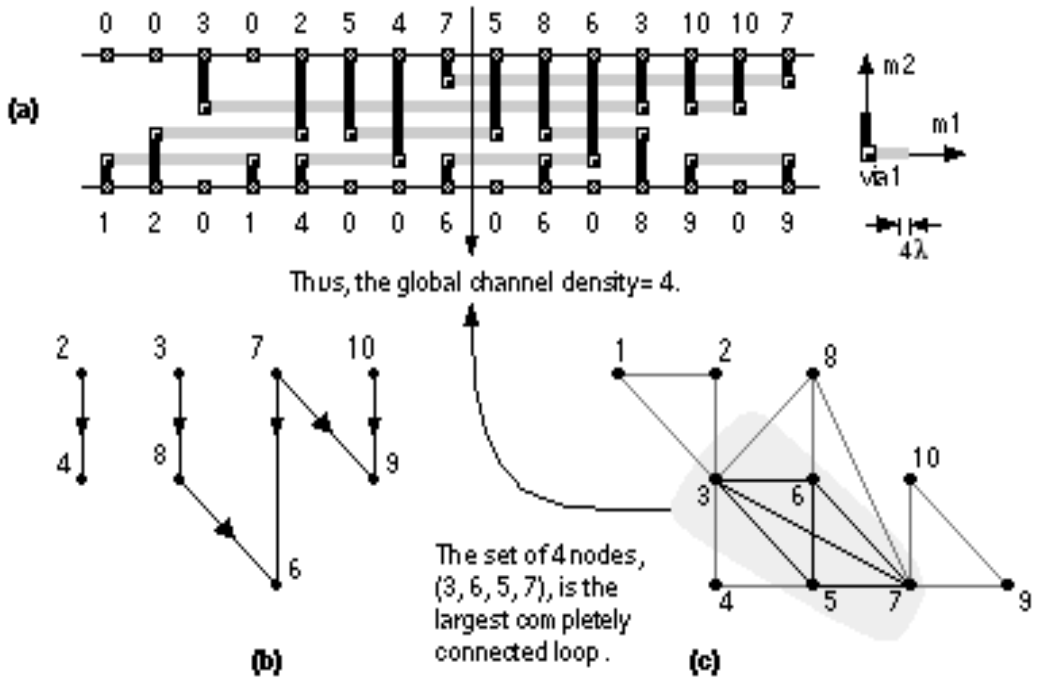





# Contraintes verticales

- **L'algorithme du bord gauche peut provoquer des court-circuits lorsque deux connecteurs sont sur la même colonne mais appartiennent à deux nœuds différents**
- **Dans ce cas le connecteur du haut impose une contrainte verticale sur le connecteur du bas**
- **On peut représenter les contraintes verticales sous forme d'un graphe orienté**
  - ***Le sens des arcs indique le sens de la contrainte***
  - ***Le segment horizontal du nœud qui impose la contrainte doit être au-dessus du segment horizontal du nœud qui la subit***

# Graphe des contraintes verticales



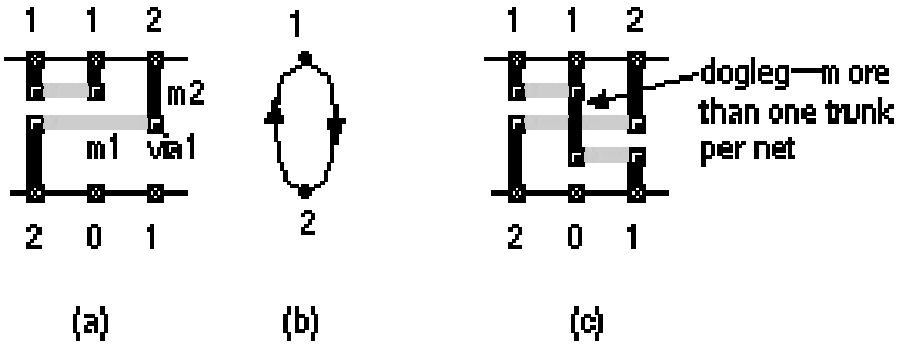



# Densité globale de canal

- **On peut également construire un graphe des contraintes horizontales**
  - ***Deux nœuds sont reliés si leurs segments horizontaux sont recouvrants***
  - ***Ce graphe de contraintes horizontal est non orienté***
  - ***En extrayant du graphe le sous groupe totalement interconnecté maximal on obtient la mesure de la densité globale du canal***
- **En l'absence de contraintes verticales le routeur sait atteindre la densité globale par l'algorithme du bord gauche**

# Un problème NP-complet

- Avec des contraintes verticales le problème devient … NP-complet
- Il existe des situations où le graphe de contraintes verticales est cyclique
  - *Algorithmes du bord gauche inopérants*
  - *Impossible de router avec un seul segment horizontal*






# Diaphonie

- Pour réduire les capacités de couplage on interdit la superposition des différents niveaux de métal
- Mais dans les technologies récentes les capacités de couplage entre lignes de même métal sont du même ordre que les capacités de routage entre lignes superposées de métaux différents
- En levant l'interdiction on peut descendre sous la densité globale de canal (avec deux métaux : moitié de la densité globale de canal)
- Certains routeurs acceptent des contraintes de couplage
  - *Longueur maximale pour des connexions parallèles*
  - *Longueur maximale pour des connexions superposées*
- Il y a encore beaucoup de travail à faire pour prendre en compte la diaphonie

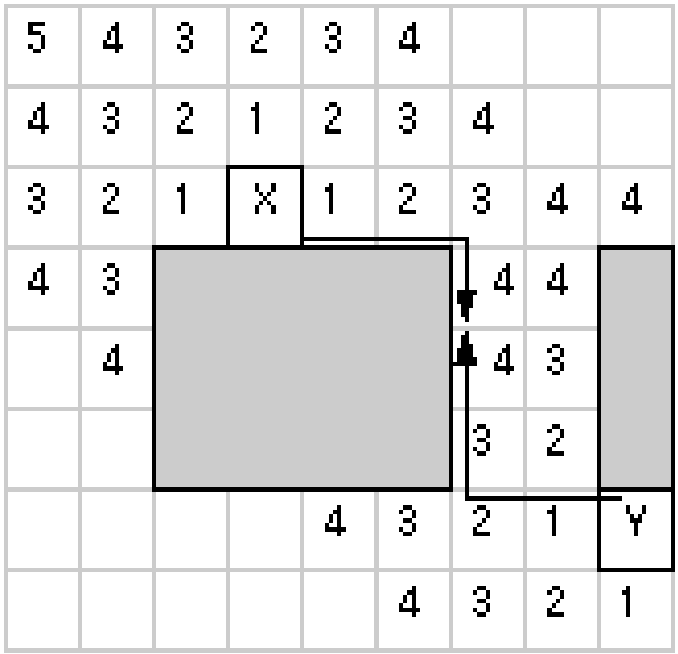
# Le routage à deux métaux, conclusion

- Une compaction après routage final permet de gagner 15 à 20 % (Cheng et al 1992)
- Les routeurs récents atteignent pratiquement le maximum théorique en deux métaux
  - *Ce problème peut être considéré comme résolu*
  - *Les difficultés reviennent*
    - Lorsque le nombre de métaux est supérieur ou égal à 3
    - Lorsque les canaux ne sont plus rectangulaires
    - Les routeur de surfaces sont plus adaptés à ces classes de problèmes



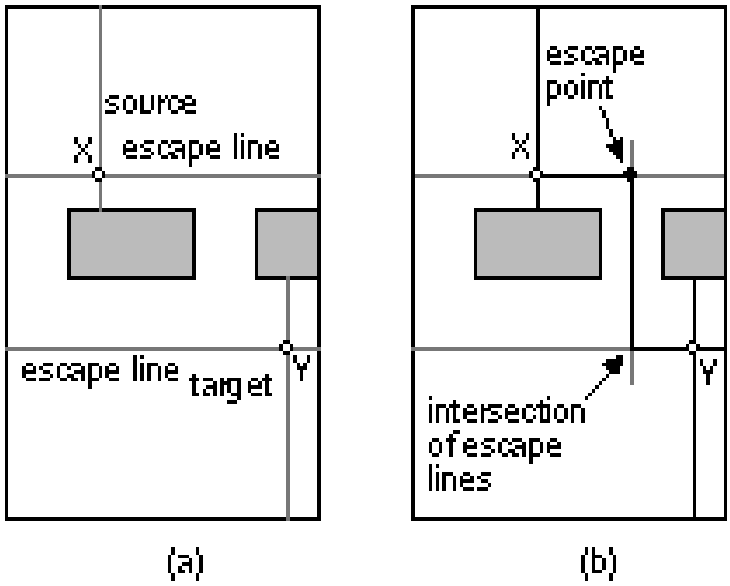
# Les algorithme de routage de surfaces : labyrinthe

- Les plus anciens
- Propagation d'ondes
- Réduction du nombre de changements de directions



# Les algorithme de routage de surfaces : lignes

- Les plus récents
- Plus efficaces






# Routage à plus de 2 métaux

- Plusieurs approches
  - *Les niveaux de métal sont réservés à une direction. C'est le cas le plus fréquent*
    - HVH : m1 et m3 en horizontal, m2 en vertical
    - VHV : m1 et m3 en vertical, m2 en horizontal
    - On choisit en fonction du niveau des ports d'entrée– sortie des cellules et des niveaux interdits sur les cellules
  - *Les niveaux ne sont pas réservés*
- Avec plus de trois niveaux on réserve souvent les niveaux supplémentaires aux alimentations ou aux horloges

# Routage à plus de 2 métaux (suite)

- **Le routage est simplifié lorsque les pas des métaux sont multiples les uns des autres**
  - ***Les technologies submicroniques profondes tentent d'offrir le même pas pour tous les niveaux***
- **Les canaux de routage disparaissent si le nombre de métaux est suffisant et si les ressources de routage par dessus les cellules sont suffisantes**
  - ***On crée parfois des cellules plus grosses pour faire des circuits plus petits***



# Optimisations

- **Après le routage global il reste peu à faire**
  - ***Eviter de trop nombreux vias***
  - ***Ajuster la largeur des pistes***
  - ***Réduire les superpositions***
  - ***Gains importants sur des longues distances***
- **Pour les nœuds à haute fréquence (horloge) on peut :**
  - ***Adapter les formes***
  - ***Chanfreiner les angles (adaptation d'impédance, réflexions)***
- **Si le routeur ne parvient pas à router certains nœuds**
  - ***On peut être obligé de retourner jusqu'au plan de masse***
  - ***On peut reprendre à la main avec l'aide d'un routeur local***
- **Lorsque le routage est achevé**
  - ***On retire les vias inutiles***
  - ***On compacte les canaux***

# Et après …

- **Vérification des règles de dessin (*DRC*)**
- **Extraction des capacités et résistances parasites**
  - ***Tenant compte des différences de géométries masques / silicium***
  - ***Format SPF, RSPF ou DSPF***
- **Vérification de cohérence masques / « *netlist* » (*LVS*)**
- **On écrit son nom quelque part (attention au DRC)**
- **Plan de câblage au boîtier (« *bonding diagram* »)**
- **Marques d'alignement**
- **Marge de découpe**
- **Fonderie, angoisse, champagne ou …**
- **… correction les bugs**

# Introduction

**ETRANGE, présentation du cahier des charges, fonctionnalité, environnement matériel et logiciel du produit**



# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé

# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé



# Les objectifs de CANEX

- Vous transmettre un savoir opérationnel
  - *Mise en œuvre des outils industriels*
    - Simulation
    - Synthèse
    - Preuve ?
    - Placement et routage
  - *Sur un cas réaliste mais pas trop*
    - Partitionnement logiciel – matériel
    - Assez simple pour finir

# Les objectifs de CANEX

- **Attention :**
  - ***Les objectifs sont pédagogiques***
    - **Le but n'est pas de finir un SOC et de le vendre**
    - **Il faut apprendre**
  - ***C'est une première (comme en vrai)***
    - **Nous n'avons pas fait le SOC avant vous (comme en vrai)**
    - **Les outils de CAO que nous utiliserons**
      - ***N'ont jamais été mis en œuvre à l'ENST sur un projet de cette taille (comme …)***
      - ***Sont bogués (comme …)***
      - ***Ne sont pas toujours bien documentés (comme …)***
    - **Vous pouvez compter sur nous mais n'oubliez pas que nous comptons sur vous : on fait partie de l'équipe**



# L'évaluation

- **Vous tiendrez à jour un cahier de TP**
  - ***Qu'est-ce que je fais***
  - ***Comment je le fais***
  - ***Pourquoi je le fais***
  - ***Quels sont les résultats obtenus***
  - ***Etc.***
- **C'est sur la base de ce cahier de TP que vous serez notés**
- **Il doit constituer la preuve que vous avez compris dans les grandes lignes comment on conçoit un SOC**
- **Vous serez notés par binôme**
  - ***Une note pour deux***
  - ***Sauf cas particulier (syndrome de la chaise d'à côté)***

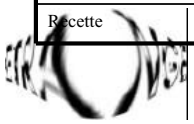
# L'organisation

- Leçons et TD (7 TH) en F900 (sauf recette en B206)
- TP (19 TH) en F301-2-3 (sauf peut-être un ou deux)
- Encadrants
  - *Jean-Luc Danger*
  - *Yves Mathieu*
  - *Lirida Naviner*
  - *Renaud Pacalet*
  - *Alexis Polti*
- 4 équipes de 3 binômes
  - *Un module matériel différent par binôme*
  - *Mêmes modules logiciels pour les 3 binômes*



# L'organisation

| | TH | Description |
|---|---|---|
| Introduction | **1** | **Présentation du cahier des charges, fonctionnalité, environnement matériel et logiciel du produit.** |
| Architecture | **5** | **Etude d'architecture, partitionnement logiciel / matériel. Choix des IP, spécification du circuit.** |
| VHDL haut niveau | **4** | **Prise en main des outils, conception d'un modèle de référence de haut niveau. Validation par simulation et comparaison avec un modèle algorithmique de référence.** |
| Logiciel | **4** | **Conception des modules logiciels. Co-validation logiciel / matériel par simulation, optimisation des modules logiciels.** |
| VHDL RTL | **5** | **Raffinement du modèle VHDL de haut niveau en modèle de niveau transfert de registres. Validations fonctionnelles par simulation et *model checking*, essais de synthèse.** |
| Synthèse | **3** | **Synthèse, optimisation, analyse temporelle, optimisation du modèle synthétisable. Preuve d'équivalence, simulation logico-temporelle.** |
| Test | **1** | **Insertion du matériel de test, génération de vecteurs de test, mesure de taux de couverture, analyse temporelle, extraction de paramètres, preuve d'équivalence.** |
| Placement et routage | **2** | **Analyse de consommation, dimensionnement des alimentations, arbre d'amplification d'horloge, placement et routage, connexion au boîtier, rétro-annotation.** |
| Recette | **1** | **Recette finale, interface avec le fondeur.** |



# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé

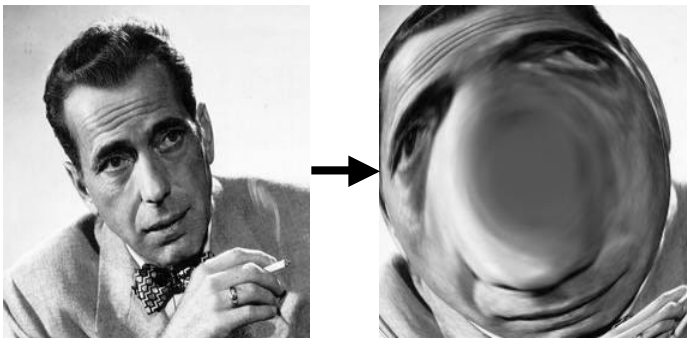


# Spécifications

- ETRANGE pour …
  - *Extracteur*
  - *de TRANsformations*
  - *GEométriques*

# Les spécifications

- **La transformation géométrique d'images (anamorphose, *morphing*)**
  - ***Truquage vidéo***
  - ***Correction des distorsions introduites par certaines optiques ou afficheurs (TVHD)***
  - ***Imagerie médicale (redressement des images d'échographie)***
  - ***Brique DESSIN***
  - ***etc.***



Humphrey avant

Humphrey après

*The Big Sleep* (1946) de Howard Hawks. D'après Raymond Chandler. Avec Lauren Bacall et Humphrey Bogart



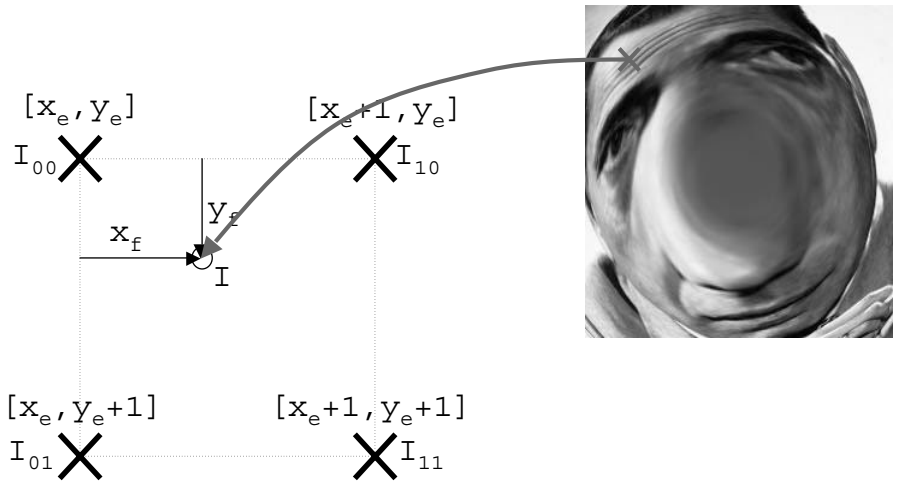
# Les spécifications

- **La technique du *backward warping***
  - ***On calcule la transformée inverse. Pour chaque pixel [X,Y] de l'image transformée on calcule [x, y], les coordonnées de son antécédent dans l'image d'origine***
  - ***Les coordonnées [x, y] du pixel antécédent sont fractionnaires [$x_e$+$x_f$ , $y_e$+$y_f$]***
    - **On utilise les parties entières [$x_e$ , $y_e$ ] pour sélectionner les pixels voisins**
    - **On utilise les parties fractionnaires [$x_f$ , $y_f$] pour pondérer l'interpolation bilinéaire**
  - ***Exemple avec des transformations polynomiales de degré 3 :***

$$x = a_{30}X^3+a_{21}X^2Y+a_{12}XY^2+a_{03}Y^3+a_{20}X^2+a_{11}XY+a_{02}Y^2+a_{10}X+a_{01}Y+a_{00}$$
$$y = b_{30}X^3+b_{21}X^2Y+b_{12}XY^2+b_{03}Y^3+b_{20}X^2+b_{11}XY+b_{02}Y^2+b_{10}X+b_{01}Y+b_{00}$$

# Les spécifications

- **L'interpolation bilinéaire**




$$I = (1-x_f)(1-y_f)I_{00}+(1-x_f)y_fI_{01}+x_f(1-y_f)I_{10}+x_fy_fI_{11}$$



# Les spécifications

- **On veut traiter des images numériques**
  - *En 256 niveaux de gris (8 bits par pixel)*
  - *Au format télévision numérique (720 x 576)*
  - *Progressif européen (25 images par secondes)*
- **On peut définir le format de l'entrée et de la sortie vidéo**
- **Les transformations sont polynomiales de degré 3**
- **On définit une transformation (20 coefficients) pour chaque bloc 16x16 de l'image transformée**
  - *Cela permet d'atteindre des transformations plus élaborées avec une complexité « raisonnable »*
- **On aimerait bénéficier de la souplesse du logiciel pour faire évoluer le produit**

# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé



# La cible technologique

- Fondeur : AMS (http://www.austriamicrosystems.com/)
- Technologie CMOS 0,35 µm
  - *Trois niveaux de métal*
  - *3,3 Volts*
  - *Délai typique*
    - NAND2 chargée par 1 mm de métal : 100 ps
    - DFF : 800 ps
  - *Consommation typique d'une NAND2 : 2 µW/MHz*
  - *Surface d'une NAND2 : 54,6 µm²*
  - *Bibliothèque* standard cells *et* IO

# La cible technologique

- **SRAM simple port**
  - *Jusqu'à 262144 bits*
  - *Mots de 8 à 64 bits*
  - *128 à 32768 mots par RAM*
  - *Bus de données séparés ou communs*
  - *Bus de données sortantes à trois états*



# La cible technologique

- **0,35 µm ; 3,3 V ; 25 °C ; 1 pF**

| SRAM simple port, surface ($mm^2$) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **0,18** | **0,26** | **0,44** | **0,70** | **1,24** | **2,18** | **3,93** |

| SRAM simple port, temps d'accès (ns) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **2,63** | **2,73** | **2,83** | **2,96** | **3,05** | **3,86** | **4,68** |

| SRAM simple port, consommation (mA/MHz) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **0,127** | **0,148** | **0,150** | **0,175** | **0,180** | **0,226** | **0,285** |

# La cible technologique

- **SRAM double port**
  - *Lecture et écriture sur chaque port*
  - *Jusqu'à 65536 bits*
  - *Mots de 8 à 64 bits*
  - *128 à 8192 mots par RAM*
  - *Bus de données séparés ou communs par port*
  - *Bus de données sortantes à trois états*



# La cible technologique

- **0,35 µm ; 3,3 V ; 27 °C ; 1 pF**

| SRAM double port, surface ($mm^2$) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **0,34** | **0,48** | **0,75** | **1,26** | **2,23** | **3,78** | **7,15** |

| SRAM double port, temps d'accès (ns) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **2,93** | **3,07** | **3,33** | **3,42** | **3,54** | **4,41** | **4,69** |

| SRAM double port, consommation (mA/MHz) | | | | | | |
|---|---|---|---|---|---|---|
| **1 kbits** | **2 kbits** | **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** |
| **0,175** | **0,196** | **0,239** | **0,255** | **0,279** | **0,362** | **0,402** |

# La cible technologique

- **ROM**
  - *Jusqu'à 524288 bits*
  - *Mots de 4 à 128 bits*
  - *512 à 65536 mots par ROM*
  - *Bus de données sortantes à trois états*



# La cible technologique

- **0,35 µm ; 3,3 V ; 27 °C ; 1 pF**

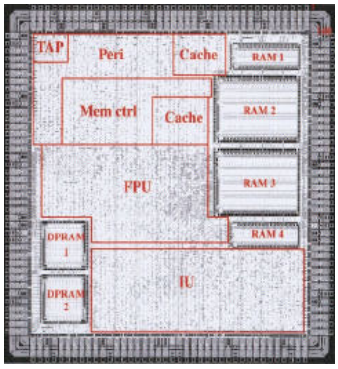
| ROM, surface ($mm^2$) | | | | | |
|---|---|---|---|---|---|
| **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** | **128 kbits** |
| **0,12** | **0,16** | **0,22** | **0,33** | **0,53** | **0,93** |

| ROM, temps d'accès (ns) | | | | | |
|---|---|---|---|---|---|
| **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** | **128 kbits** |
| **2,87** | **3,14** | **3,27** | **3,67** | **3,81** | **4,34** |

| ROM, consommation (mA/MHz) | | | | | |
|---|---|---|---|---|---|
| **4 kbits** | **8 kbits** | **16 kbits** | **32 kbits** | **64 kbits** | **128 kbits** |
| **0,089** | **0,120** | **0,138** | **0,172** | **0,285** | **0,448** |

# La cible technologique

- On peut embarquer de la DRAM
  - *C'est pas vrai mais ça nous arrange*
- On peut utiliser LEON (http://www.gaisler.com/)






# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé

# LEON

- **Sparc V8 sous licence GNU LGPL développé par l'ESA (http://www.esa.int/), maintenu par Gaisler Research**

- SPARC V8 compatible integer unit with 5-stage pipeline
- Hardware multiply, divide and MAC units
- Separate, direct-mapped instruction and data caches
- Full implementation of AMBA-2.0 AHB and APB on-chip buses
- Data cache snooping on AHB bus
- Programmable 8/16/32-bits memory controller for external prom and static ram
- On-chip debug support unit with trace buffer
- On-chip peripherals such as uarts, timers, interrupt controller and 16-bit I/O port
- Interfaces for Meiko floating-point unit and user-defined co-processor
- Open-source IEEE-754 floating-point unit (add, sub & compare only)
- Power-down mode

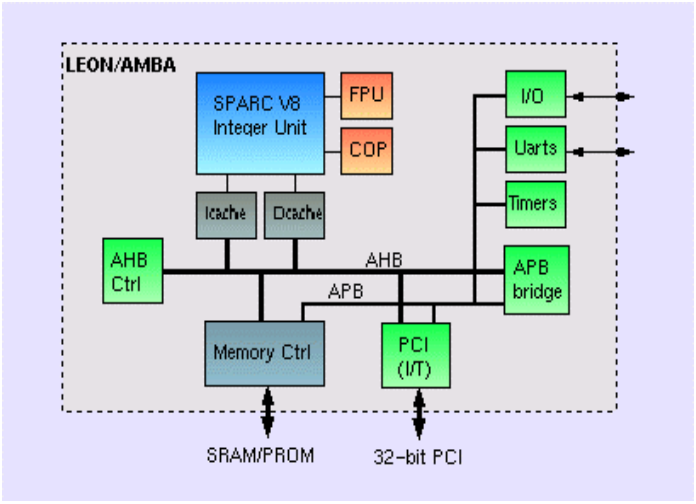


# LEON

- **Modèles VHDL synthétisables**
- **Environnement de développement LECCS**
  - ***GNU C/C++ compiler (gcc-2.95.2)***
  - ***Linker, assembler, archiver etc. (binutils-2.11)***
  - ***Standalone C-library (newlib-1.8.2 from Cygnus)***
  - ***RTEMS real-time kernel (rtems-4.5.0+)***
  - ***Boot-prom utility (mkprom-1.3.3)***
  - ***GNU debugger with Tk front-end (gdb-5.0 + insight-5.0)***
  - ***DDD graphical user interface for gdb (ddd-3.1.3)***
  - ***Remote target monitor (rdbmon-1.3)***

# LEON

- **LEON est largement configurable**
  - ***On utilise ce dont on a besoin***
  - ***On peut dimensionner les ressources***
    - **Taille des caches**
    - **Tailles des mémoires**
    - **Performances et tailles des multiplieurs**
    - **Etc.**






# LEON

- **L'interfaçage**
  - ***Interruptions***
    - **15 interruptions dont 4 sur PIO**
  - ***Parallel IO***
    - **PIO[15:0]**
  - ***2 UART***
  - ***Memory mapped IO***

# Plan

- Introduction
- Spécifications
- Technologie
- LEON
- Planning détaillé



# Planning détaillé

- Mercredi 27 février (3 heures)
  - *Une proposition d'architecture globale*
  - *Les binômes et les équipes sont constitués*
  - *Dans chaque équipe l'attribution des modules matériels est faite*
- Vendredi 1 mars (3 heures)
  - *Les entrées – sorties de chaque bloc sont définies*
  - *Les spécifications de chaque bloc sont arrêtées*
- Lundi 4 mars (1 heure ½)
  - *Les spécifications des modules logiciels sont arrêtées*

# Planning détaillé

- **Lundi 11 mars (6 heures)**
  - *Les modèles VHDL haut niveau des modules matériels sont validés*
  - *L'assemblage du processeur complet est terminé*
  - *L'environnement de simulation de ETRANGE est terminé, se compile, s'élabore*
  - *Il ne manque que les modules logiciels pour simuler l'ensemble*
- **Vendredi 15 mars, 15h00 (6 heures)**
  - *Les modules logiciels sont terminés, validés, optimisés*
  - *La simulation de ETRANGE tourne sans problème*



# Planning détaillé

- **Vendredi 20 mars (7 heure ½)**
  - *Les modèles VHDL RTL sont terminés et validés*
  - *Les simulations globales tournent avec les modèles VHDL RTL*
  - *Les parties contrôles ont été validées par* **model checking**
  - *Les modèles RTL se synthétisent sans problème*
- **Lundi 25 mars (4 heures ½)**
  - *Les synthèses et optimisations sont terminées et validées*

# Planning détaillé

- **Mercredi 27 mars, 10h00 (1 heure ½)**
  - ***L'insertion du matériel de test est faite***
  - ***Les vecteurs de test sont générés et le taux de couverture est correct***
- **Mercredi 27 mars (3 heures)**
  - ***Le placement routage est terminé***
  - ***Les* timing *sont bons***
  - ***Le* bonding diagram *est fait***