Conservatoire National des Arts et Métiers

FIP-CPI 2013–2014

FPGA EPRM

Cours + TP

C.ALEXANDRE

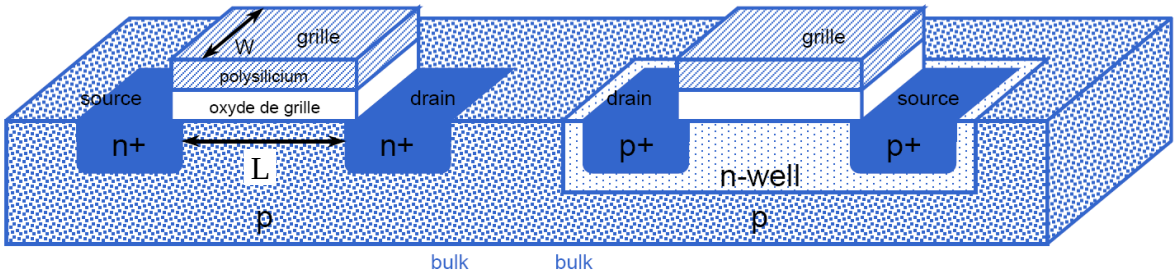
1 Introduction à la conception en VHDL

1.1 Familles de circuits intégrés numériques

Il existe une loi empirique, appelée **loi de Moore**, qui dit que la densité d'intégration dans les circuits intégrés numériques à base de silicium double tous les 18 à 24 mois, à prix du circuit équivalent. Les conséquences de la loi de Moore sont les suivantes :

- La longueur du canal L qui était égale à 10 µm dans les années 1970 a atteint 32 nm en 2010, ce qui fait un facteur de réduction en surface de $312^2 \cong 100000$ en 40 ans (car toutes les dimensions du transistor ont été réduites de la même manière).
- Le nombre de transistors par circuits a encore plus augmenté à cause de l'accroissement de la taille des puces qui atteint 600 mm^2 (avec quelques millions de transistors par mm^2). On pourrait atteindre facilement un milliard de transistors par circuit (toujours en 2010).
- La fréquence de fonctionnement a augmenté d'un facteur 10000 environ entre 1970 et 2010.

La figure suivante vous montre une paire de transistor MOS canal N et canal P qui forme la brique de base de la logique CMOS :



La loi de Moore s'est révélée remarquablement exacte jusqu'à ce jour et elle explique en grande partie l'évolution de l'électronique numérique de ses origines à nos jours. Durant les années 60, au début de l'ère des circuits intégrés numériques, les fonctions logiques telles que les portes, les registres, les compteurs et les ALU, étaient disponibles en circuit TTL. On parlait de composants SSI (Small Scale Integration) ou MSI (Medium Scale Integration) pour un tel niveau d'intégration.

Dans les années 70, le nombre de transistors intégrés sur une puce de silicium augmentait régulièrement. Les fabricants mettaient sur le marché des composants LSI (Large Scale Integration) de plus en plus spécialisés. Par exemple, le circuit 74LS275 contenait 3 multiplieurs de type Wallace. Ce genre de circuit n'était pas utilisable dans la majorité des applications. Cette spécialisation des boîtiers segmentait donc le marché des circuits intégrés et il devenait difficile de fabriquer des grandes séries. De plus, les coûts de fabrication et de conception augmentaient avec le nombre de transistors. Pour toutes ces raisons, les catalogues de composants logiques standards (série 74xx) se sont limités au niveau LSI. Pour tirer avantage des nouvelles structures VLSI (Very Large Scale Integration), les fabricants développèrent 4 nouvelles familles :

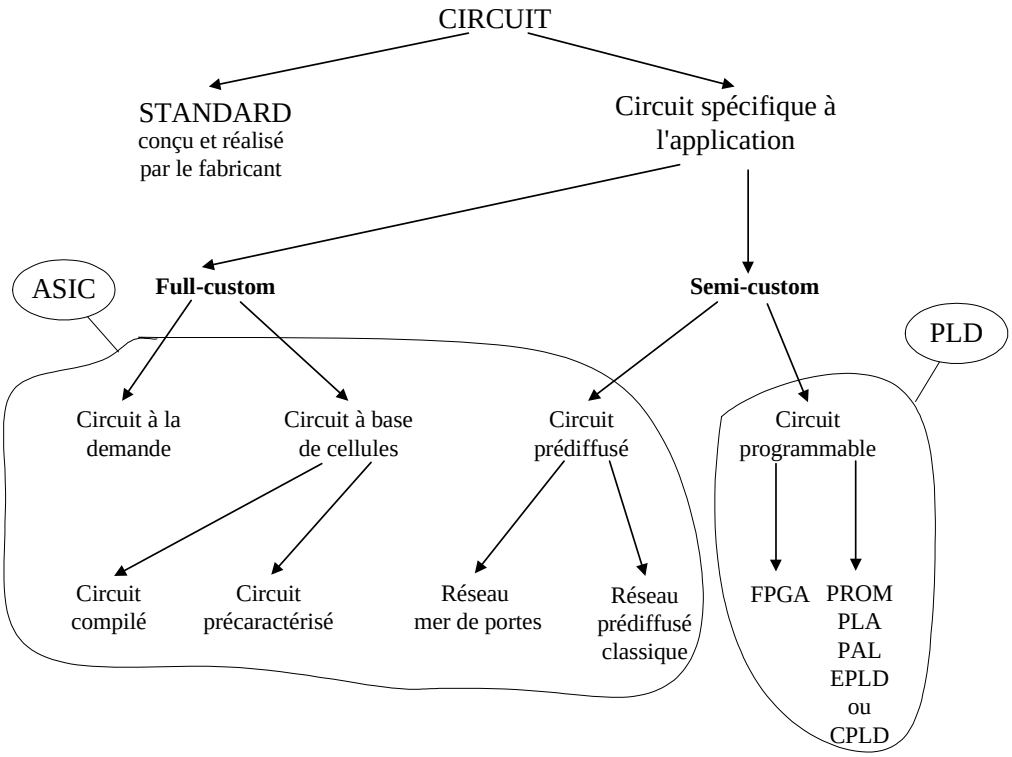
- Les microprocesseurs et les mémoires RAM et ROM : les microprocesseurs et les circuits mémoires sont attrayants pour les fabricants. Composants de base pour les systèmes informatiques, ils sont produits en très grandes séries.
- Les ASSP (Application Specific Standard Product) : ce sont des produits sur catalogue qui sont fabriqués en grande série. La fonction réalisée est figée par le constructeur, mais le domaine d'utilisation est spécifique à une application. Exemple : un contrôleur Ethernet, un encodeur MPEG-4, …
- Les circuits programmables sur site : n'importe quelle fonction logique, combinatoire ou séquentielle, avec un nombre fixe d'entrées et de sorties, peut être implantée dans ces circuits. A partir de cette simple idée, plusieurs variantes d'architecture ont été développées (PAL, EPLD, FPGA,…).
- Les ASIC (Application Specific Integrated Circuit) réalisés chez le fondeur : le circuit est conçu par l'utilisateur avec des outils de CAO, puis il est réalisé par le fondeur.

A l'heure actuelle, la plupart des circuits numériques est issue de ces 4 familles. Cependant, certains éléments simples du catalogue standard (famille 74) sont toujours utilisés.

Plus simplement, on peut distinguer deux catégories de circuits intégrés : les circuits standards et les circuits spécifiques à une application :

- Les circuits standards se justifient pour de grandes quantités : microprocesseurs, contrôleurs, mémoires, ASSP, …
- Les circuits spécifiques sont destinés à réaliser une fonction ou un ensemble de fonctions dans un domaine d'application particulier.

La figure suivante représente une classification des circuits intégrés numériques.



Dans la littérature, le terme ASIC est employé pour décrire l'ensemble des circuits spécifiques à une application. Or, dans le langage courant, le terme ASIC est presque toujours utilisé pour décrire les circuits réalisés chez un fondeur. On désigne, par le terme générique PLD (Programmable logic Device), l'ensemble des circuits programmables par l'utilisateur. Parmi les circuits numériques spécifiques à une application, il faut distinguer deux familles :

- les circuits conçus à partir d'une puce de silicium « vierge » (Full-custom),
- les circuits où des cellules standards sont déjà implantées sur la puce de silicium (Semi-custom).

Dans le premier groupe, les circuits appelés « Full custom », on trouve les circuits à la demande et ceux à base de cellules (CBIC : Cell Based Integrated Circuit). Le fondeur réalise l'ensemble des masques de fabrication. Dans le second groupe, les circuits appelés « Semi-custom », on trouve les circuits prédiffusés (GA : Gate Array) et les circuits programmables. Les cellules standards, déjà implantées sur la puce de silicium, doivent être interconnectées les unes avec les autres. Cette phase de routage est réalisée, soit par masquage chez le fondeur (prédiffusé), soit par programmation.

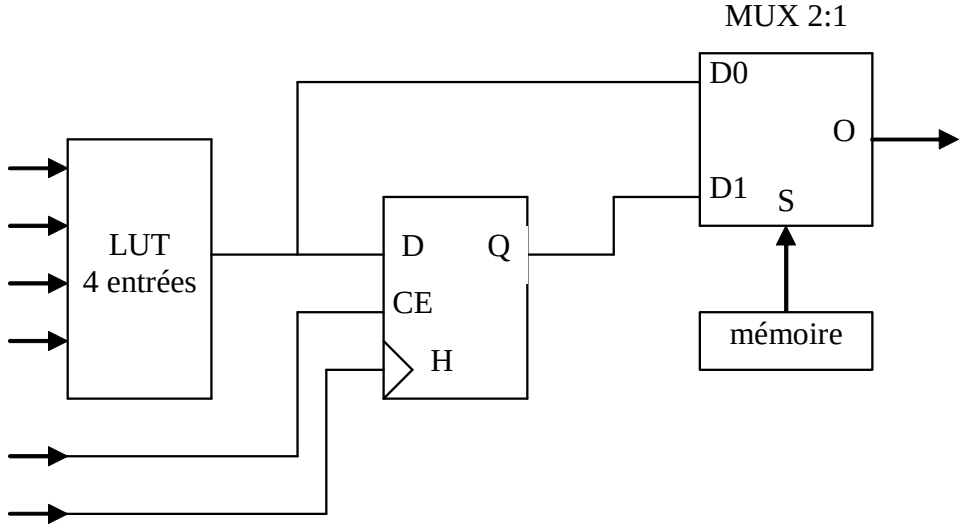
Tous les ASICs ont un point commun ; il est nécessaire de passer par un fondeur pour réaliser les circuits. Cela implique une quantité minimale de circuits à fabriquer (au moins quelques dizaines de milliers par an) ainsi qu'un délai de fabrication de quelques mois. De plus, le coût de fabrication initial (NRE) devient de plus en plus élevé (quelques millions de $ en 2010) et doit être amorti sur des quantités de circuits de plus en plus grandes. Ces inconvénients ont conduit les fabricants à proposer des circuits programmables par l'utilisateur (sans passage par le fondeur) qui sont devenus au fil des années, de plus en plus évolués. Rassemblés sous le terme générique PLD, les circuits programmables par l'utilisateur se décomposent en deux familles :

1. les PROM, les PLA, les PAL et les EPLD,
2. les FPGA.

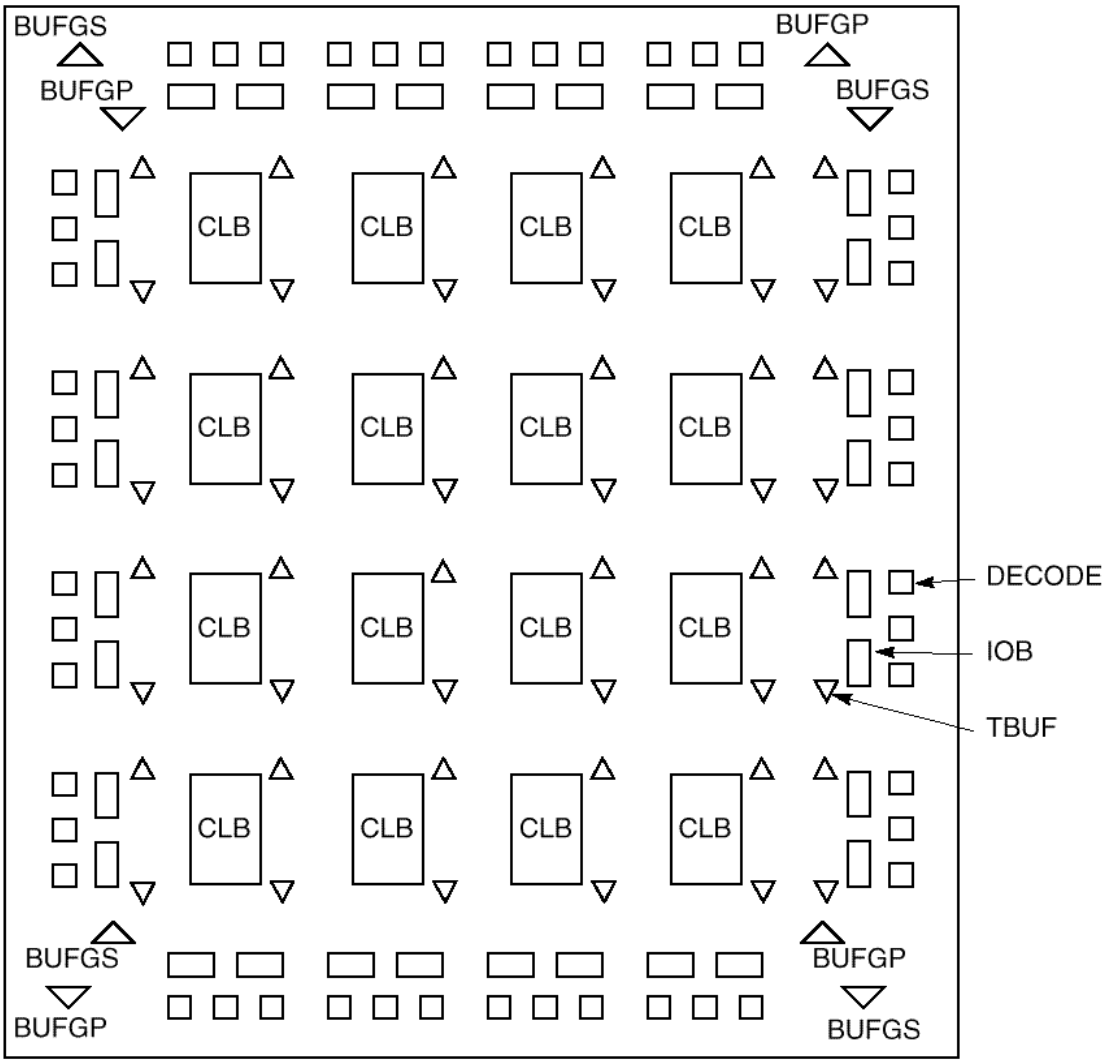
Dans ce cours, nous allons utiliser une maquette FPGA pour tester nos designs. Voyons plus en détail cette famille de circuits logiques programmables.

1.2 Les FPGA

Lancé sur le marché en 1984 par la firme XILINX, le FPGA (Field Programmable Gate Array) est un circuit prédiffusé programmable. Le concept du FPGA est basé sur l'utilisation d'une LUT (LookUp Table) comme élément combinatoire de la cellule de base. En première approximation, cette LUT peut être vue comme une mémoire (16 bits en général) qui permet de créer n'importe quelle fonction logique combinatoire de 4 variables d'entrées. Chez Xilinx, on appelle cela un générateur de fonction ou Function Generator. La figure suivante représente la cellule type de base d'un FPGA.

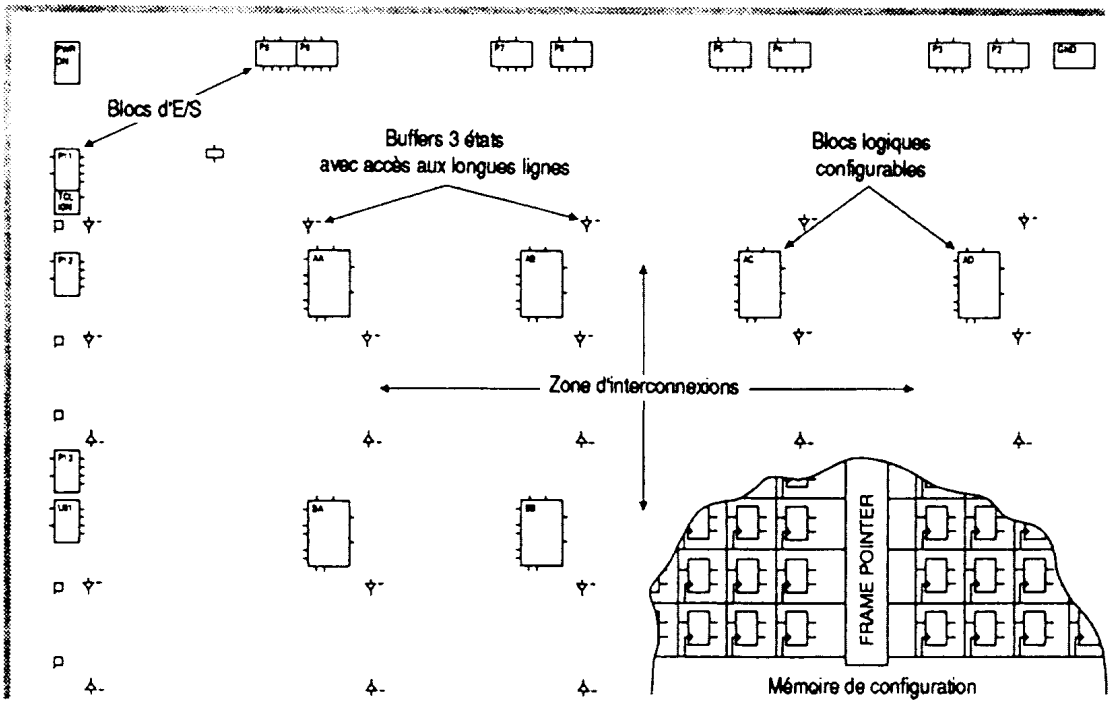


Elle comprend une LUT 4 entrées et une bascule D (D Flip-Flop). La bascule D permet la réalisation de fonctions logiques séquentielles. La configuration du multiplexeur 2 vers 1 de sortie autorise la sélection des deux types de fonction, combinatoire ou séquentielle. Les cellules de base d'un FPGA sont disposées en lignes et en colonnes. Des lignes d'interconnexions programmables traversent le circuit, horizontalement et verticalement, entre les cellules. Ces lignes d'interconnexions permettent de relier les cellules entre elles, et avec les plots d'entrées/sorties (IOB : Input Output Block).



Les connexions programmables sur ces lignes sont réalisées par des transistors MOS dont l'état est contrôlé par des cellules mémoires SRAM. Ainsi, toute la programmation d'un FPGA est contenue dans des cellules SRAM. La configuration est le processus qui charge le design dans la SRAM afin de programmer les fonctions des différents blocs et de réaliser leurs interconnexions. On voit (symboliquement), sur la figure ci-dessous, qu'il y a sous la logique

dédiée à l'application une mémoire de configuration qui contient toutes les informations concernant la programmation des CLB et des IOB ainsi que l'état des connexions. Cette configuration est réalisée, à la mise sous tension, par le chargement d'un fichier binaire contenu généralement dans une mémoire flash série qui se trouve à côté du FPGA sur la carte.



La fréquence maximale de fonctionnement d'une fonction logique est difficile à prévoir avant son implémentation. En effet, cela dépend fortement du résultat de l'étape de placement-routage. Cependant, une fréquence comprise entre 100 et 200 MHz est un bon ordre de grandeur. Tous les FPGA sont fabriqués en technologie CMOS, les plus gros d'entre eux intègrent jusqu'à 10000000 portes logiques utilisables. Leur prix est compris entre 10 et 1000 euros à l'unité.

Par rapport aux prédiffusés classiques, les interconnexions programmables introduisent des délais plus grands que la métallisation (environ 3 fois plus lents). Par contre, les cellules logiques fonctionnent à la même vitesse. Pour minimiser les délais de propagation dans un FPGA, il faut donc réduire le nombre de cellules logiques utilisées pour réaliser une fonction. Par conséquent, les cellules logiques d'un FPGA sont plus complexes que celles d'un prédiffusé.

1.3 Conception d'un design

1.3.1 Saisie du design

1.3.1.1 Saisie de schéma

Le rôle de la saisie de schéma est d'établir, à partir des composants utilisés et de leurs interconnexions, une liste de connectivité (netlist). Cette netlist peut ensuite être utilisée par le simulateur ou bien par les outils de placement–routage. L'utilisateur visualise son design à l'aide de schémas graphiques faciles à interpréter. Les composants sont représentés par des symboles graphiques. Les autres outils EDA (Electronic design automation) travaillent à partir de la netlist ASCII (ou binaire).

1.3.1.2 Langage de description de bas niveau

La saisie de schéma est un moyen puissant pour spécifier un design car le schéma graphique est une manière naturelle pour un électronicien de créer ou de lire un design. Elle a toutefois trois inconvénients principaux :

- L'incompatibilité entre les librairies qui rend le portage d'un design entre deux fabricants de FPGA quasiment impossible.
- La complexité d'un schéma devient très élevée quand le nombre de portes augmente. Au-delà de 10000 portes, il devient généralement ingérable.
- Une modification importante au milieu d'une page du schéma nécessite généralement la réécriture complète de la page.

Pour toutes ces raisons, on a essayé de développer des outils spécifiant le design avec une entrée de type texte plutôt qu'avec une entrée de type graphique. Les premiers langages de bas niveau ont été créés pour programmer les PAL puis les PLD. On trouvait principalement les deux langages suivants :

- **PALASM**. Il s'agissait d'un langage de programmation de PAL conçu par AMD/MMI. Aujourd'hui obsolète, les outils de développement avaient l'immense avantage d'être gratuits.
- **ABEL**. ABEL était un langage de programmation de PAL conçu par Data I/O. Il a été très utilisé aux USA mais les outils de développement étaient assez coûteux.

1.3.1.3 Langage de description matériel

Les deux langages précédents étaient loin d'être assez puissants pour pouvoir spécifier un ASIC ou un FPGA. Deux langages de description matériel sont apparus dans les années 80 : Verilog et VHDL (VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language). Les simulateurs et les synthétiseurs ne travaillent plus aujourd'hui qu'avec un de ces deux langages. Le schéma graphique, quand il est utilisé, est simplement traduit dans un de ces langages.

1.3.2 La chaîne complète de conception en VHDL

Nous allons maintenant voir la chaîne de développement complète utilisant l'outil ISE de la société Xilinx. Elle comprend les étapes suivantes.

1. Ecriture du modèle VHDL avec l'éditeur intégré du navigateur de projet. Nous allons voir au chapitre suivant une introduction à l'écriture de modèles VHDL synthétisables. Il y a deux parties à écrire :
   - ✓ L'entité (entrées/sorties),
   - ✓ L'architecture (la description du fonctionnement).

2. Ecriture du testbench (on utilisera l'outil HDL Bencher dans un premier temps). Pour pouvoir simuler le modèle VHDL de notre design, il faut écrire des vecteurs (stimuli) de test qui forment le testbench. Cette écriture se déroule en deux phases :
   - ✓ définition graphique des vecteurs de test,
   - ✓ sauvegarde de la saisie graphique et du fichier équivalent en langage VHDL.

3. Simulation fonctionnelle (simulateur ModelSim). A partir des fichiers design.vhd (modèle VHDL du design) et design_tb.vhw (vecteurs de test en VHDL), nous pouvons effectuer la simulation fonctionnelle du design. Il y a quatre phases :
   - ✓ compilation des fichiers,
   - ✓ lancement du simulateur,
   - ✓ exécution des vecteurs de test,
   - ✓ vérification du fonctionnement à l'aide des chronogrammes.

4. Synthèse logique avec XST. La synthèse est l'opération qui consiste, à partir du fichier texte, à produire la netlist contenant le schéma électrique destiné aux outils de placement-routage (fichier au format NGC). Cette étape est particulièrement délicate dans le cas des FPGA à cause de la complexité élevée (granularité) de la cellule de base du FPGA. La description VHDL n'utilise pas de bibliothèque propriétaire et elle est assez générale. La portabilité est donc bien meilleure qu'avec la saisie de schéma (les langages VHDL et Verilog sont normalisés), mais toute la complexité du processus de développement repose maintenant sur l'efficacité de la synthèse. Le rôle du synthétiseur est ici de comprendre et d'interpréter une description abstraite du design afin de générer un fichier NGC compréhensible par les outils d'implémentation, c'est-à-dire une netlist NGC composée de primitives simples.

La synthèse est l'opération qui permet de créer une netlist NGC à partir d'une description de haut niveau écrite en VHDL.

5. Implémentation. Le but de l'implémentation est de générer un fichier de configuration permettant de programmer le FPGA à partir du fichier NGC. Le tableau suivant indique les 4 étapes possibles ainsi que les rapports qui lui sont associés.

Etape	Signification
Translation	Création d'un fichier de design unique à plat (sans hiérarchie)
Mapping	Découpage du design en primitives existant dans le FPGA
Placement-routage	Placement et routage des primitives Assignation des broches du FPGA
configuration	Génération du fichier de configuration

6. Simulation temporelle (avec ModelSim). Après le placement-routage, on peut obtenir un modèle VHDL réel du design (modèle VITAL) ainsi que le fichier de timings qui lui est associé (fichier SDF). On utilise le fichier de stimuli généré par HDL Bencher pour effectuer la simulation de timing du design en quatre phases :
   - ✓ compilation des fichiers,
   - ✓ lancement du simulateur,
   - ✓ exécution des vecteurs de test,
   - ✓ vérification du fonctionnement à l'aide des chronogrammes.

7. Téléchargement. Une fois le design entièrement simulé, nous pouvons télécharger le fichier de configuration du FPGA dans la maquette grâce au logiciel Impact :
   - ✓ Initialisation de la chaîne JTAG,
   - ✓ Association du fichier .bit avec le FPGA,
   - ✓ Téléchargement dans le FPGA.

8. Vérification sur la maquette FPGA. Tous les designs réalisés dans ce cours seront testés sur la maquette FPGA. Un design simulé mais non testé matériellement, c'est comme un programme en C que l'on aurait compilé, mais pas exécuté.

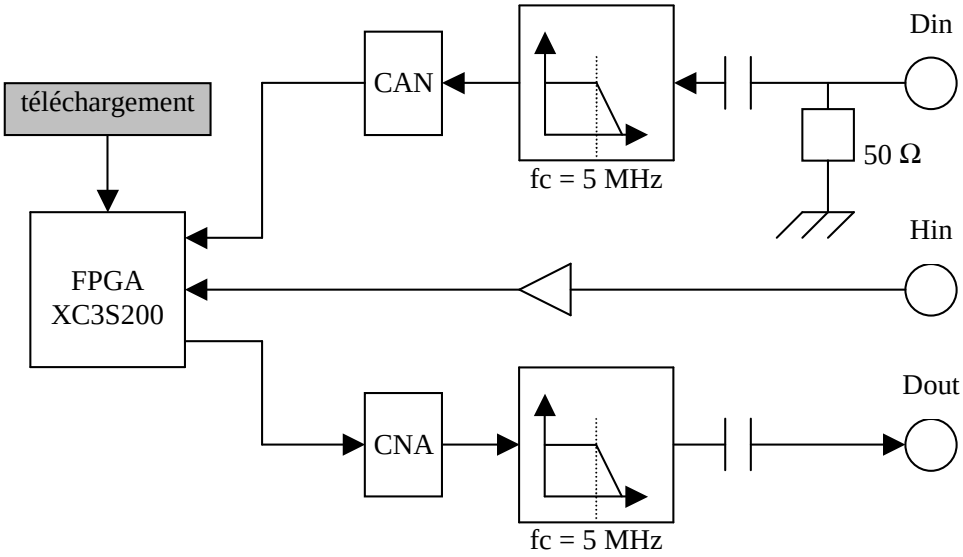
1.4 La maquette FPGA

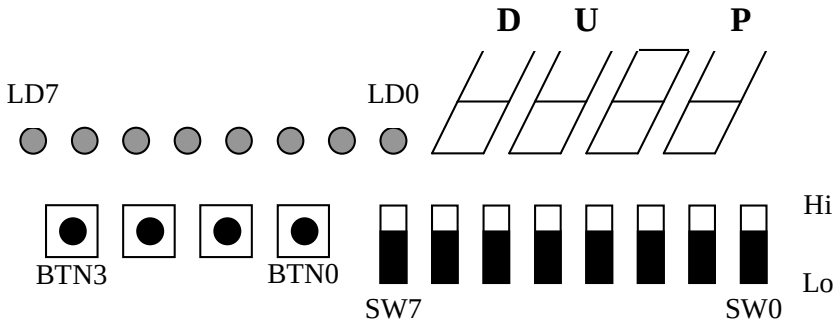
La maquette FPGA est constituée :

- D'un FPGA Xilinx XC3S200-4C en boîtier BGA 256 broches.

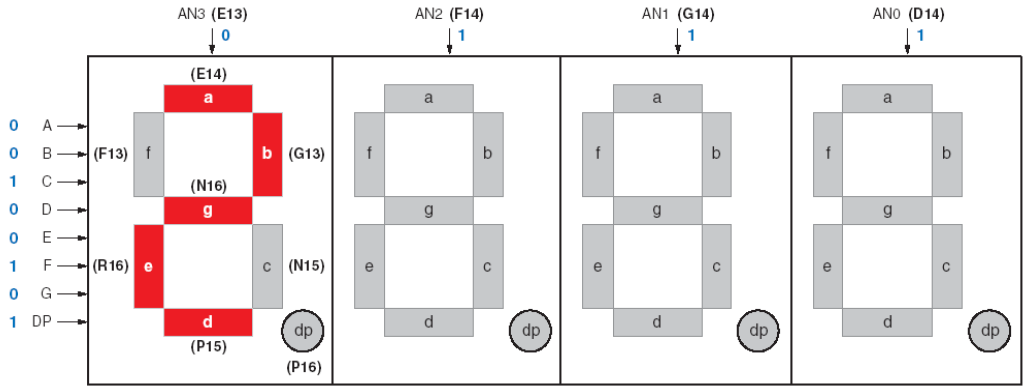
- D'un CAN 8 bits 32 MSPS (AD9280) précédé d'un filtre anti-repliement (Tchebyscheff du 3ème ordre, fc = 5 MHz). **Le niveau d'entrée sur la prise Din (adaptée 50 Ω) ne doit pas dépasser 1V crête à crête**. Le FPGA doit fournir un signal d'horloge (niveau CMOS 3.3V) à ce composant.

- D'un CNA 8 bits 125 MSPS (AD9708) suivi d'un filtre de lissage (Tchebyscheff du 3ème ordre, fc = 5 MHz). Le FPGA doit fournir un signal d'horloge (niveau CMOS 3.3V) à ce composant.





- De quatre afficheurs 7 segments multiplexés (dont trois nommés D (dizaine), U (unité) et P (puissance)). Un segment s'allume quand la sortie du FPGA qui lui est connecté est au niveau bas.



- De 8 leds nommées LD0 à LD7. Une led s'allume quand la sortie du FPGA qui lui est connectée est au niveau haut.

- De 8 interrupteurs (Lo/Hi) nommés SW0 à SW7. Quand un interrupteur est sur Hi, l'entrée du FPGA qui lui est connecté est au niveau haut.

- De quatre boutons poussoirs nommés BTN0 à BTN3. L'appui sur un de ces boutons déclenche une impulsion positive (met au niveau 1 l'entrée correspondante).

- D'une entrée d'horloge Hin (niveau CMOS 3.3V adaptée 50 Ω).

- D'un oscillateur 50 MHz (100 ppm, niveau CMOS 3.3V) connecté sur le FPGA.

- D'une prise de téléchargement JTAG pour programmer le FPGA. Le téléchargement s'effectue via une des prises parallèles du PC.

- D'un cordon d'alimentation. Il faut une alimentation 5 V, 1.6 A.

1.5 Introduction au langage VHDL

1.5.1 Définition

VHDL sont les initiales de VHSIC Hardware Description Langage, VHSIC étant celles de Very High Scale Integrated Circuit. Autrement dit, VHDL signifie : langage de description matériel s'appliquant aux circuits intégrés à très forte intégration.

1.5.2 Généralités

1.5.2.1 Nécessité

L'évolution des technologies induit une complexité croissante des circuits intégrés qui ressemblent de plus en plus aux systèmes complets d'hier. Aujourd'hui, on intègre dans une puce ce qui occupait une carte entière il y a quelques années. La simulation logique globale du système au niveau "porte" n'est plus envisageable en termes de temps de simulation. C'est donc tout naturellement que des simulateurs fonctionnels ont commencé à être utilisés en microélectronique. Dans les années 70, une grande variété de langages et de simulateurs était utilisée. Cette diversité avait pour conséquence une non portabilité des modèles et donc une impossibilité d'échange entre les sociétés. Un des rôles de VHDL est de permettre l'échange de descriptions entre concepteurs. Ainsi peuvent être mises en place des méthodologies de

modélisation et de description de bibliothèques en langage VHDL. L'effort de standardisation d'un langage tel que VHDL était nécessaire par le fait qu'il ne s'agissait pas de construire un seul simulateur VHDL (contrairement à la quasi-totalité des autres langages), mais de permettre l'apparition d'une multitude d'outils (de simulation, de vérification, de synthèse, ...) de constructeurs différents utilisant la même norme. Ceci garantit une bonne qualité (la concurrence) et l'indépendance de l'utilisateur vis à vis des constructeurs de ces outils.

1.5.2.2 Un peu d'histoire

Les langages de description matériel (HDL ou Hardware Description Language) ont été inventés à la fin des années 60. Ils s'appuyaient sur les langages de programmation et devaient permettre la description et la simulation de circuits. Entre les années 1968 et 1975, une grande diversité de langages et de simulateurs ont vu le jour. Cependant, leur syntaxe et leur sémantique étaient incompatibles et les niveaux de descriptions étaient variés. En 1973, le besoin d'un effort de standardisation s'est fait ressentir et c'est ainsi que le projet CONLAN (pour CONsensus LANguage) a été mis en place. Les principaux objectifs de ce projet étaient de définir un langage de description matériel permettant de décrire un système à plusieurs niveaux d'abstractions, ayant une syntaxe unique et une sémantique formelle et non ambiguë.

En mars 1980, le département de la défense des Etats Unis d'Amérique (DoD ou Department of Defense) lançait le programme VHSIC. En 1981, des demandes pour un nouveau langage de description de systèmes matériels, indépendant de toute technologie et permettant de couvrir tous les besoins de l'industrie microélectronique, ont été formulées. C'est en 1983, que d'importantes sociétés telles que IBM, Intermetrics ou encore Texas Instruments se sont investies dans ce projet et à la fin de l'année 1984, un premier manuel de référence du langage ainsi qu'un manuel utilisateur ont été rédigés. En 1985, des remises en cause et des évaluations ont donné naissance à la version 7.2 du langage VHDL, et en juillet 1986, Intermetrics a développé un premier compilateur et un premier simulateur.

C'est en mars 1986, qu'un groupe chargé de la standardisation du langage VHDL a été créé. Il s'agit du groupe américain VASG (VHDL Analysis and Standardization Group) qui est un sous-comité des DASS (Design Automation Standard Subcommittees), eux-mêmes émanant de l'IEEE (Institute of Electrical and Electronics Engineers). La norme VHDL IEEE 1076 a été approuvée le 10 décembre 1987. En tant que standard IEEE, le langage VHDL évolue tous

les cinq ans afin de le remettre à jour, d'améliorer certaines caractéristiques ou encore d'ajouter de nouveaux concepts. Ainsi en 1991 a débuté le processus de re-standardisation : regroupement et analyse des requêtes, définition des nouveaux objectifs du langage, spécifications des changements à apporter au langage. La nouvelle norme du langage VHDL a été votée en septembre 1993. La syntaxe et la sémantique de cette nouvelle norme ont donné lieu à un nouveau manuel de référence.

Il est à souligner que la nouvelle version du langage VHDL généralement notée VHDL'93 est, au moins théoriquement, la seule et unique version légale du langage. L'approbation de la nouvelle norme VHDL en 1993 rend le standard précédent (IEEE Std 1076 voté en 1987) désuet. Néanmoins, et heureusement, VHDL'93 reste compatible avec VHDL'87.

1.5.3 Les principales caractéristiques du langage VHDL

1.5.3.1 Un langage s'appliquant à plusieurs niveaux de descriptions

Le langage VHDL couvre tous les niveaux partant des portes logiques de base jusqu'aux systèmes complets (plusieurs cartes, chacune comprenant plusieurs circuits). Il s'applique tout aussi bien au niveau structurel qu'au niveau comportemental, en passant par le niveau transferts de registres ou RTL (Register Transfer Logic).

La description comportementale

Le circuit est décrit sans tenir compte de la réalisation concrète sur un composant donné (Elle ne comporte aucune horloge et les largeurs des bus ne sont pas spécifiées.). On utilise les fonctions de haut niveau d'abstraction de VHDL. Le code est donc portable, mais il dépend entièrement du synthétiseur pour le résultat (si un tel synthétiseur existait, ce qui n'est pas encore le cas). Ce style de description permet en particulier de spécifier le circuit sous forme d'un algorithme.

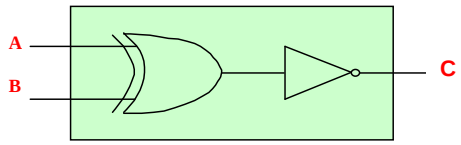
La description flot de données (RTL)

Le circuit est décrit grâce à plusieurs couche de registres (bascules D) reliées par de la logique combinatoire. On utilise généralement une liste d'instructions concurrentes d'affectations de signaux du type *"<signal> <= <expression> "* où <expression> peut représenter un simple signal, ou une expression utilisant des opérateurs logiques, arithmétiques ou relationnels, ou

une expression conditionnelle. C'est la manière traditionnelle d'écrire du VHDL et qui donne généralement les meilleurs résultats avec les synthétiseurs commercialisés. Ce niveau de description est souvent appelé "logiques à transfert de registres" ou RTL (Register Transfer Logic).

La description structurelle de bas niveau

Le circuit est décrit par sa structure, sous forme d'une liste de composants instanciés et des interconnexions les reliant. En fait, c'est l'équivalent littéral d'un schéma représentant l'interconnexion de portes élémentaires issues d'une bibliothèque.



La description structurelle de haut niveau

Le circuit est découpé en blocs fonctionnels de haut niveau qui sont reliés entre eux. Il s'agit toujours d'un schéma, mais il représente des composants qui peuvent être écrit en comportemental ou en RTL (mais aussi en structurel bas niveau). On retrouve en général ce type de description dans le design supérieur d'une description hiérarchique (top level design).

1.5.3.2 La portabilité

La portabilité constituait un des objectifs principaux pendant la phase de définition du langage VHDL. L'utilisation largement répandue de ce langage est essentiellement due au fait qu'il soit un standard. Un standard offre beaucoup plus de garanties (stabilité, fiabilité, etc.) vis-à-vis des utilisateurs que ne le permettrait un langage potentiellement précaire développé par une société privée. La portabilité signifie dans le cadre des FPGA, qu'il est possible à tout moment de changer de technologie cible (Altera, Xilinx, etc.) ou bien de famille au sein d'un même fabricant ; ou même d'avoir une stratégie de conception qui consiste à réaliser un prototype sur un circuit programmable (FPGA) avant de basculer sur des circuits spécifiques (ASIC).

1.5.3.3 La lisibilité

La représentation schématique n'est pas toujours d'une grande lisibilité. Il est difficile de saisir rapidement le fonctionnement d'une machine d'état parmi un enchevêtrement de registres et de portes logiques. Une documentation écrite est très souvent nécessaire à la compréhension d'un

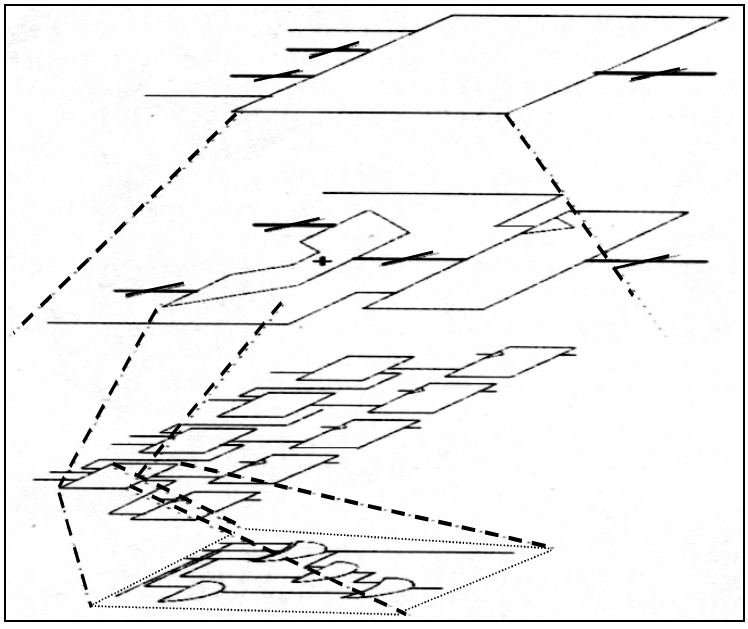
schéma. VHDL apporte par son principe de modularité et d'interconnexion de composants une grande lisibilité permettant la compréhension d'un système, sans pour autant supprimer l'utilité d'un dessin fonctionnel.

1.5.3.4 La modularité

La modularité est une des caractéristiques essentielles de VHDL. Une description partitionnée en plusieurs sous-ensembles est dite modulaire. VHDL supporte la conception modulaire et hiérarchique, qui offre de nombreux avantages :

- les descriptions sont plus simples,
- les durées de simulation, de synthèse et de mise au point sont raccourcies,
- la fiabilité est améliorée : chaque sous-ensemble peut être testé exhaustivement,
- les sous-ensembles sont réutilisables.

En VHDL, une description peut donc faire appel à des modules externes et les interconnecter de manière structurelle. Par exemple, la vision hiérarchique d'un additionneur consiste à traiter les blocs, l'additionneur et le registre, comme assemblage d'objets plus élémentaires comme le montre la figure suivante.



1.5.3.5 Le couple entité architecture

Un design quelconque (circuit intégré, carte électronique ou système complet) est complètement défini par des signaux d'entrées et de sorties et par la fonction réalisée en interne. L'élément essentiel de toute description en VHDL est formé par le couple entité/architecture qui décrit l'apparence externe d'un module et son fonctionnement interne.

Entité

L'entité décrit la vue externe du modèle : elle permet de définir les ports par où sont véhiculés les informations (signaux) et les paramètres génériques. Le code suivant donne un exemple d'entité définie pour un compteur N bits. Il est possible de définir des valeurs par défaut pour les paramètres ; dans cet exemple, le paramètre WDTH correspond au nombre de bits du compteur avec pour valeur par défaut 4.

```
entity compteur is
  generic (WDTH : integer :=4; STOP : integer :=10);
  port( CLK : in std_logic ;
         CE : in std_logic ;
      CLEAR : in std_logic;
        CEO : out std_logic;
       DOUT : out std_logic_vector(WDTH -1 downto 0));
end compteur;
```

Architecture

L'architecture définit la vue interne du modèle. Cette description peut être de type structurel, flot de données, comportemental ou une combinaison des trois. Tous les fonctionnements ne nécessitent pas le même degré de précision : tantôt une description globale permet d'obtenir un résultat satisfaisant, tantôt une description très précise s'avère nécessaire.

A chaque entité peut être associée une ou plusieurs architectures, mais, au moment de l'exécution (simulation ou synthèse), une seule architecture est utilisée. Ceci présente l'intérêt de comparer plusieurs architectures pour choisir la meilleure. L'architecture comprend aussi une partie déclarative où peuvent figurer un certain nombre de déclarations (de signaux, de composants, etc.) internes à l'architecture. A titre d'exemple, le code suivant donne une description possible d'architecture associée à l'entité décrite précédemment pour un compteur N bits.

```
architecture a1 of compteur is
  signal INT_DOUT : std_logic_vector(DOUT'range) ;
begin
  process(CLK, CLEAR) begin
    if (CLEAR='1') then
      INT_DOUT <= (others => '0');
    elsif (CLK'event and CLK='1') then
      if (CE='1') then
        if (INT_DOUT=STOP-1) then
          INT_DOUT <= (others => '0');
        else
          INT_DOUT <= (INT_DOUT + 1);
        end if;
      end if;
    end if;
  end process;
  CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and INT_DOUT(3) and CE;
  dout <= int_dout;
end;
```

1.5.3.6 Les principaux objets manipulés

Les constantes : Une constante peut être assimilée à un signal interne (au circuit) auquel est associée une valeur fixe et définitive. La constante peut être de tout type.

Les signaux : Les signaux sont spécifiques à la description matérielle. Ils servent à modéliser les informations qui passent sur les fils, les bus ou, d'une manière générale, qui transitent entre les différents composants. Les signaux assurent donc la communication.

Les variables : Une variable est capable de retenir une valeur pendant une durée limitée. Elle ne peut être employée qu'à l'intérieur d'un process. A l'opposé d'un signal, une variable n'est pas une liaison concrète et ne doit pas laisser de trace après synthèse.

1.5.3.7 Les types

Une des principales caractéristiques de VHDL est qu'il s'agit d'un langage fortement typé. Tous les objets définis en VHDL doivent appartenir à un type avant d'être utilisés. Deux objets sont compatibles s'ils ont la même définition de type. Un type définit l'ensemble des valeurs que peut prendre un objet ainsi que l'ensemble des opérations disponibles sur cet objet. Puisque VHDL s'applique au domaine logique, les valeurs '0' et '1' peuvent être considérées. L'ensemble de ces deux valeurs définit le type BIT. Il est possible de définir d'autres valeurs comme par exemple la valeur 'Z' désignant la valeur trois états. Un type plus complet, englobant neuf valeurs logiques différentes décrivant tous les états d'un signal électronique numérique, a été standardisé dans la librairie std_logic_1164 : le *std_logic*. Les valeurs que peut prendre un signal de ce type sont :

- ‘U’ : non initialisé,
- ‘X’ : niveau inconnu, forçage fort,
- ‘0’ : niveau 0, forçage fort,
- ‘1’ : niveau 1, forçage fort,
- ‘Z’ : haute impédance,
- ‘W’ : niveau inconnu, forçage faible,
- ‘L’ : niveau 0, forçage faible,
- ‘H’ : niveau 1, forçage faible,
- ‘-‘ : quelconque.

Les valeurs ‘0’ et ‘L’ sont équivalentes pour la synthèse, tout comme les valeurs ‘1’ et ‘H’. Les valeurs ‘U’, ‘X’ et ‘W’ ne sont utilisables que pour la simulation d’un design.

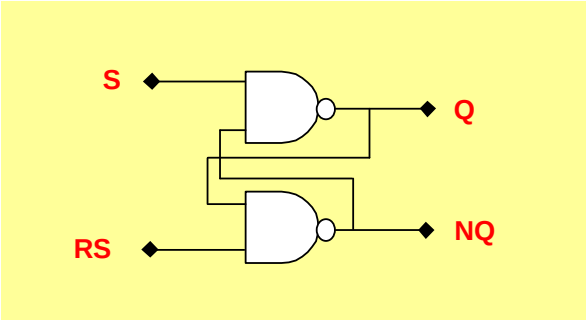
Il existe quatre familles de type en VHDL :

- les types scalaires, dont la valeur est composée d'un seul élément (*integer*, *bit*, *std_logic*, *boolean*, etc.),
- les types composés, dont la valeur comprend plusieurs éléments (*bit_vector*, *std_logic_vector*),
- les types accès, qui sont les pointeurs des langages de programmation,
- les types fichiers, qui ne sont utilisés que pour les objets fichiers.

Les deux dernières familles ne sont bien sûr pas utilisables pour le développement d’un circuit intégré.

1.5.3.8 Fonctionnement concurrent

Le comportement d'un circuit peut être décrit par un ensemble d'actions s'exécutant en parallèle. C'est pourquoi VHDL offre un jeu d'instructions dites concurrentes. Une instruction concurrente est une instruction dont l'exécution est indépendante de son ordre d'apparition dans le code VHDL. Par exemple, prenons le cas d'un simple latch, comme le montre la figure :



Les deux portes constituant ce latch fonctionnent en parallèle. Une description possible de ce circuit est donnée dans le code suivant (seule l'architecture est donnée).

```
architecture comportement of VERROU is
begin
        Q   <= S nand NQ;
        NQ <= R nand Q;
end comportement;
```

Ces deux instructions s'exécutent en même temps. Elles sont concurrentes ; leur ordre d'écriture n'est pas significatif ; quel que soit l'ordre de ces instructions, la description reste inchangée.

1.5.3.9 Fonctionnement séquentiel

Les instructions concurrentes qui viennent d'être présentées pourraient suffire à définir un langage de description matériel. Cependant, certains circuits sont plus faciles à décrire en utilisant des instructions séquentielles similaires à des instructions de langages classiques de programmation. En VHDL, les instructions séquentielles ne s'utilisent qu'à l'intérieur des processus. Un processus est un groupe délimité d'instructions, doté de trois caractéristiques essentielles :

- Le processus s'exécute à chaque changement d'état d'un des signaux auxquels il est déclaré sensible.
- Les instructions du processus s'exécutent séquentiellement.
- Les modifications apportées aux valeurs de signaux par les instructions prennent effet à la fin du processus.

L’exemple de la description suivante montre une architecture (seule) d’un latch D contenant un processus qui est exécuté lors du changement d’état de l’horloge CLK.

```
architecture  comportement  of basc_D  is
begin
   Process  (CLK)
   Begin
         If ( CLK= ‘1’)  then
            Q     <= D;
         End if ;
   End process  ;
end comportement;
```

1.5.4 VHDL par rapport aux autres langages

Bien que VHDL soit maintenant largement accepté et adopté, il n'est pas le seul langage de description matériel. Pendant ces trente dernières années, beaucoup d'autres langages ont été développés, ont évolué et sont encore utilisés aujourd'hui par les concepteurs de circuits intégrés. Créé pour être un standard, VHDL doit son succès à la fois à ses prédécesseurs et à la maturité de ses principes de base.

Verilog est un langage qui a été développé par une compagnie privée pour ses propres besoins (langage de spécification pour leurs outils de simulation). Verilog a tout d'abord été décrit par la société Gateway Design Automation qui a ensuite fusionné avec la compagnie Cadence Design Systems. Pour que Verilog puisse faire face à VHDL, Cadence a décidé de le rendre public en 1990. L'utilisation de Verilog est promue par le groupe Open Verilog International (OVI) qui a publié en Octobre 1991 la première version du manuel de référence du langage Verilog. En 1995, Verilog est devenu un standard sous la référence IEEE 1364.

Du point de vue syntaxique, VHDL s'est largement inspiré du langage ADA, alors que Verilog ressemble au langage C. VHDL est certainement le plus difficile à utiliser car il reste très général. De plus, il demande à l'utilisateur d'avoir des habitudes de programmeur (compilation séparée, langage fortement typé, notion de surcharge, etc…). Comparé à Verilog qui reste proche de la réalité physique, VHDL est sans aucun doute plus complexe. Mais il est aussi beaucoup moins ambigu dans ses descriptions du matériel.

1.5.5 Normes et extensions

Les normes suivantes définissent le langage et ses extensions pour la simulation, la synthèse et la rétro annotation.

Norme IEEE	sujet
1076.1	1987 et 1993, IEEE Standard VHDL Language Reference Manual.
1076.2	1996, IEEE Standard VHDL Mathematical Packages.
1076.3	1997, IEEE Standard VHDL Synthesis Packages.
1076.4	1995, IEEE Standard VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification.
1164	1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164).
P1497	Standard for (SDF) Standard Delay Format for the Electronic Design Process

Il existe plusieurs versions de VHDL : VHDL-87, 93, 2000, 2002, 2008 et VHDL-AMS (pour l'analogique). La version 93 est la plus utilisée (c'est celle que nous utiliserons). Il faut noter que :

- Le package IEEE 1076.3 concerne la synthèse. Il précise l'interprétation des nombres (comment interpréter un nombre entier par exemple) ainsi que les packages arithmétiques numeric_bit et numeric_std permettant de traiter les types signed et unsigned ainsi que de nombreuses fonctions de conversion. Hélas, la normalisation a été tardive (1996) et a permis la normalisation de fait des packages de Synopsys std_logic_unsigned, std_logic_signed, std_logic_arith qui sont souvent utilisés à la place (notamment par Synopsys).

- VITAL est une extension de la norme permettant la retro annotation en association avec un fichier SDF. VITAL permet l'écriture de modèle physique de composant vérifiant entre autres toutes ses caractéristiques de timing. Ces timings sont contenus dans le fichier SDF. On écrit rarement soi-même un modèle VITAL. C'est généralement l'outil de placement routage qui écrit le modèle réel du design en VITAL et génère le fichier SDF. Ces deux fichiers sont ensuite utilisés pour la simulation post-implémentation.

1.5.6 La synthèse

1.5.6.1 définition

La synthèse est définie comme une succession d'opérations permettant à partir d'une description de circuit dans un domaine fonctionnel (description comportementale) d'obtenir une description équivalente dans le domaine physique (description structurelle). Le processus de synthèse peut être défini comme une boîte noire ayant en entrée une description abstraite en termes de langage de description matériel, et comme sortie une description structurée en termes de dispositifs interconnectés (une netlist).

1.5.6.2 La synthèse automatique de circuits : dans quel but ?

Le premier intérêt de la synthèse est de permettre une description la plus abstraite possible d'un circuit physique. Le concepteur a de moins en moins de détails à donner. Par exemple, pour décrire un compteur, la description détaillée des signaux de contrôle explicitement utilisés n'est pas indispensable. Seule la fonctionnalité de comptage et les contraintes de synthèse (qui peuvent être des contraintes de temps, d'optimisation, de circuit cible, etc…) doivent être indiquées. Le but de l'abstraction est de réduire et de condenser les descriptions au départ et, par conséquent, de faciliter leur correction en cas d'erreurs. L'autre avantage de l'abstraction est la portabilité. Plus l'abstraction est élevée, plus la description est portable. En effet, une abstraction élevée ne fait pas référence à un composant cible car elle ne spécifie pas les détails.

Puisque les systèmes deviennent de plus en plus complexes, il ne sera bientôt plus possible d'envisager leur conception en saisie de schéma (sauf pour la saisie du top-level design qui est en général structurelle). Avec la synthèse, le nombre d'informations devant être fournies par le concepteur diminue. Ces informations consistent essentiellement en la description comportementale du circuit et des contraintes correspondantes. La synthèse amènera sans aucun doute dans les prochaines années, à des circuits plus sûrs, plus robustes et devrait, à l'avenir, être considérée comme une marque de qualité dans le cycle de conception. Puisque la synthèse permet de réduire la taille des descriptions, elle permet également de faciliter les remises à jour, de rendre les corrections plus rapides, et de pouvoir explorer un ensemble plus vaste de solutions architecturales (avec la synthèse comportementale). Dans ce contexte, le meilleur compromis entre coût et performance peut plus facilement être atteint par le

concepteur. Le grand nombre de descriptions déjà existantes couplé avec la possibilité de les paramétrer amènent à la création de ressources de bibliothèques réutilisables. Ceci permet d'améliorer la productivité des concepteurs de circuits intégrés.

1.5.7 Différences entre un langage de programmation et VHDL

VHDL n'est pas un langage de programmation comme le C, c'est un langage de description matériel. Il est important de bien comprendre la différence entre les deux :

- un **langage de programmation** est destiné à être traduit en **langage machine** puis à être exécuté par un **microprocesseur**.
- un **langage de description matériel** comme VHDL décrit une réalité matérielle, c'est-à-dire le fonctionnement d'un système numérique. Il va être traduit (synthétisé) en un ensemble de **circuits logiques combinatoires et séquentiels** qui vont être implémentés dans un **circuit intégré**. Il n'y a aucun microprocesseur pour exécuter la description VHDL dans un circuit intégré.

Des instructions telle que la boucle for peuvent se retrouver dans les deux langages, mais elles n'ont absolument pas le même rôle.

1.5.8 Bibliographie

Le langage VHDL est un langage complexe dont les possibilités sont très étendues. Si vous souhaitez approfondir vos connaissances sur ce sujet, les ouvrages suivants sont disponibles :

titre	auteur	éditeur	niveau	intérêt
VHDL : introduction à la synthèse logique	Philippe LARCHER	Eyrolles	débutant	*
Le langage VHDL	Jacques WEBER Maurice MEAUDRE	Dunod	Moyen à avancé	**
Fundamentals of digital logic with VHDL design	Stephen BROWN Zvonko VRANESIC	Mc Graw Hill	Débutant à avancé	***
HDL chip design	Douglas J.SMITH	Doone Pubns	Débutant à avancé	***
Digital systems design with VHDL and synthesis	K.C. CHANG	IEEE Computer Society	avancé	***

2 Logique combinatoire

2.1 Introduction, variables et fonctions logiques

Un circuit numérique est réalisé à partir d'un assemblage hiérarchique d'opérateurs logiques élémentaires réalisant des opérations simples sur des variables logiques. Ces variables logiques peuvent prendre les états : (**vrai, true**) ou bien (**faux, false,**). Vous connaissez déjà de nombreux systèmes physiques qui travaillent à partir de grandeurs ne pouvant prendre que deux états:

- interrupteur ouvert ou fermé,
- lampe allumée ou non, objet éclairé ou non,
- moteur en marche ou arrêté,
- affirmation vraie ou fausse,
- grandeur physique supérieure ou inférieure à un seuil fixé.

Par convention, on associe généralement à l'état vrai d'une variable logique la valeur binaire 1 et la valeur 0 à l'état faux. C'est la logique positive. On peut aussi faire le contraire (c'est la logique négative) en associant la valeur 0 à l'état vrai et la valeur 1 à l'état faux. Dans ce cours, on travaillera toujours en logique positive, sauf mention contraire. Electriquement, l'état vrai (ou la valeur 1) va être associé à un niveau de tension haut (la tension d'alimentation du montage V_{DD} en général) et l'état faux (valeur 0) va être associé à un niveau de tension bas (en général, la masse du montage GND). Les composants élémentaires du montage sont des paires de transistors MOS (logique CMOS) qui peuvent être vus comme des interrupteurs ouverts ou fermés.

En logique, tout raisonnement est décomposé en une suite de propositions élémentaires qui sont vraies ou fausses. Georges BOOLE, mathématicien britannique (1815-1864) a créé une algèbre qui codifie les règles (algèbre booléenne) à l'aide de variables logiques ne pouvant prendre que deux états et d'opérations élémentaires portant sur une ou 2 variables. L'algèbre de BOOLE ne traite que de la logique combinatoire, c'est à dire des circuits numériques dont la sortie ne dépend que de l'état présent des entrées (sans mémoire des états passés). A chaque opérateur logique booléen (NON, ET, OU, NON ET, NON OU, OU exclusif, NON OU exclusif), on va associer un circuit numérique combinatoire élémentaire. La logique séquentielle (avec mémoire) sera vue au chapitre suivant.

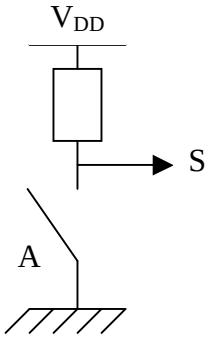
2.1.1 Les opérateurs fondamentaux

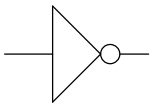
2.1.1.1 INV (NON)

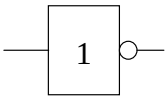
L'opérateur d'inversion ne porte que sur une seule variable d'entrée. Si A est la variable d'entrée, S la variable de sortie vaut : $S = \overline{A}$ (on prononce A barre). Le tableau suivant résume l'action de cet opérateur. Dans ce chapitre, l'interrupteur ouvert vaut 0 et l'interrupteur fermé vaut 1.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	$S = \overline{A}$
0	1
1	0





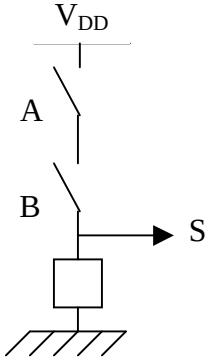


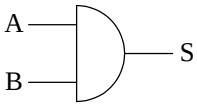
2.1.1.2 AND (ET)

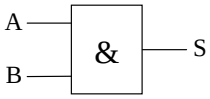
L'opérateur AND (ET) porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors S = A.B. S est vraie si A **ET** B sont vraies. L'opérateur AND est symbolisé par le point (**.**) comme la multiplication en mathématique (c'est d'ailleurs l'opération réalisée en binaire). On peut aussi voir cette fonction comme l'opérateur minimum (min) qui prend la plus petite des deux valeurs. Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	S = A.B
0	0	0
0	1	0
1	0	0
1	1	1





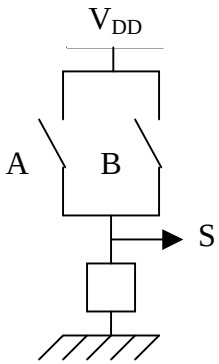


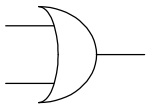
2.1.1.3 OR (OU)

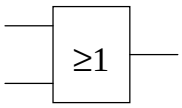
L'opérateur OR (OU) porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors S = A+B. S est vraie si A **OU** B sont vraies. L'opérateur OR est symbolisé par le plus (+) comme l'addition en mathématique. On peut voir cette fonction comme l'opérateur maximum (max) qui prend la plus grande des deux valeurs. Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	S = A+B
0	0	0
0	1	1
1	0	1
1	1	1





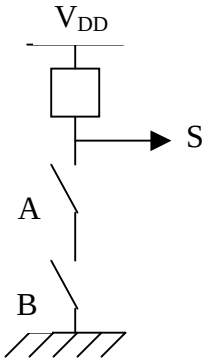


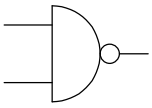
2.1.1.4 NAND (NON ET)

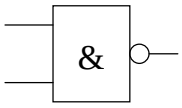
L'opérateur NAND (NON ET) porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors $S = \overline{A.B}$. S est **fausse** si A **ET** B sont vraies. L'opérateur NAND est l'inverse de l'opérateur AND. Son symbole est le symbole du ET suivi d'une bulle qui matérialise l'inversion. Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	$S = \overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0





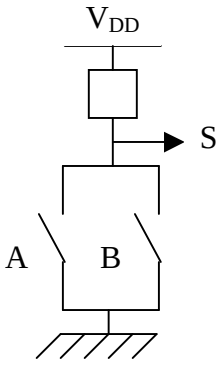


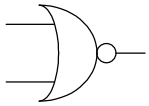
2.1.1.5 NOR (NON OU)

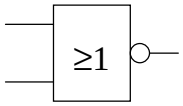
L'opérateur NOR (NON OU) porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors $S = \overline{A + B}$. S est **fausse** si A **OU** B sont vraies. L'opérateur NOR est l'inverse de l'opérateur OR. Son symbole est le symbole du OU suivi d'une bulle qui matérialise l'inversion. Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	$S = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0





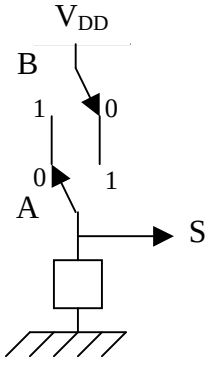


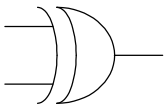
2.1.1.6 XOR (OU exclusif)

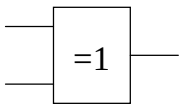
L'opérateur XOR (OU exclusif) n'est pas un opérateur de base car il peut être réalisé à l'aide des portes précédentes. Il porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors $S = A \oplus B = A.\overline{B} + \overline{A}.B$. S est vraie si A est **différent de** B. L'opérateur XOR est symbolisé par un + entouré d'un cercle ($\oplus$) car il réalise l'addition en binaire, mais modulo 2. Le OU normal (inclusif) est vrai quand A et B sont vrai (1+1=1). Le OU exclusif exclut ce cas (d'où son nom). Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	$S = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0





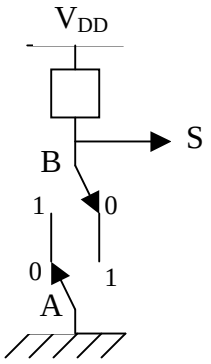


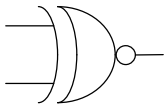
2.1.1.7 XNOR (NON OU exclusif)

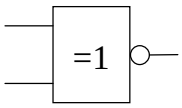
L'opérateur XNOR (NON OU exclusif) n'est pas non plus un opérateur de base. Il porte sur deux variables d'entrée. Si A et B sont les variables d'entrée, alors $S = \overline{A \oplus B} = \overline{A}.\overline{B} + A.B$. S est vraie si A **égale** B. L'opérateur XNOR est l'inverse de l'opérateur XOR. Son symbole est le symbole du XOR suivi d'une bulle qui matérialise l'inversion. Le tableau suivant résume l'action de cet opérateur.

Table de vérité	Montage	Symbole traditionnel	Symbole normalisé

A	B	$S = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

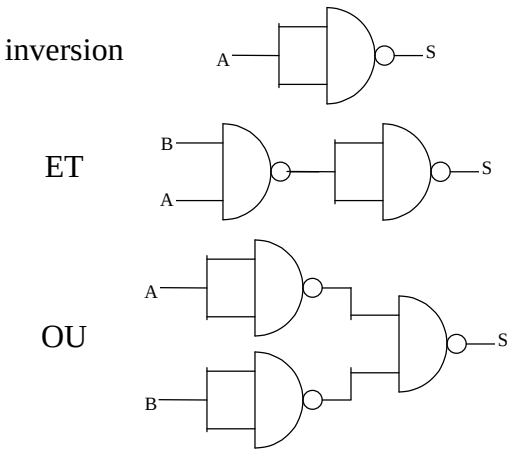






2.1.1.8 Portes universelles

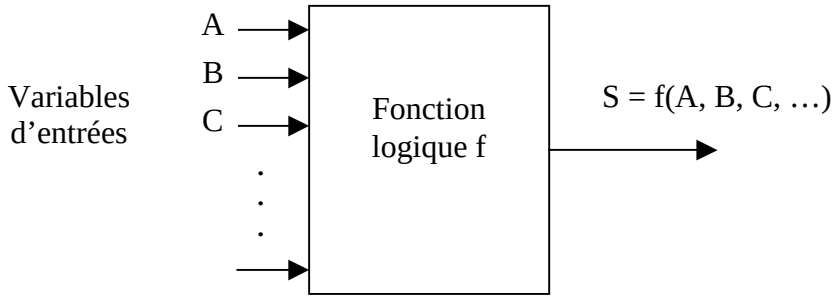
Les NAND et les NOR sont des portes universelles car elles permettent de réaliser toutes les opérations logiques élémentaires. Par exemple avec des NAND, on peut réaliser les opérations :



Le même genre de montage peut être réalisé avec des portes NOR. On verra plus tard que les portes NAND et NOR demandent le minimum de transistors pour être fabriquées (4 transistors pour une NAND/NOR à deux entrées) et sont les plus rapides.

2.1.2 Algèbre de BOOLE

L'algèbre de BOOLE porte sur des variables logiques (qui ne peuvent prendre que deux états, vrai ou faux). Elle possède trois opérateurs booléens : NOT (NON), AND (ET), OR (OU). L'algèbre de BOOLE permet de réaliser des fonctions à l'aide de variables booléennes et des trois opérateurs de base. Le résultat obtenu est booléen, c'est-à-dire vrai ou faux.



Les lois fondamentales de l'algèbre de BOOLE se vérifient en écrivant les tables de vérités et en testant tous les cas possibles. Ces lois sont les suivantes :

Commutativité	$A.B = B.A$ $A+B = B+A$
Associativité	$A.(B.C) = (A.B).C$ $A+(B+C) = (A+B)+C$
Distributivité	ET sur OU : $A.(B+C) = (A.B) + (A.C)$ OU sur ET : $A+(B.C) = (A+B) . (A+C)$ ET sur ET : $A.(B.C) = (A.B) . (A.C)$ OU sur OU : $A+(B+C) = (A+B) + (A+C)$
Idempotence	$A.A = A$ $A+A = A$
Complémentarité	$A.\overline{A} = 0$ $A + \overline{A} = 1$
Identités remarquables	$1.A = A$ $1+A=1$ $0.A = 0$ $0+A = A$

A partir de ces propriétés, on démontre les relations de base suivantes :

$A.B + A.\overline{B} = A$ $(A + B).(A + \overline{B}) = A$
$A + A.B = A$ $A.(A + B) = A$
$A + \overline{A}.B = A + B$ $A.(\overline{A} + B) = A.B$

Le OU exclusif a des propriétés particulières. Il possède les propriétés de commutativité et d'associativité, mais n'est pas distributif par rapport au ET ou au OU. On a vu que :

$$A \oplus B = \overline{A}.B + A.\overline{B}$$

On a de plus

$$\overline{A \oplus B} = \overline{A} \oplus B = A \oplus \overline{B}$$

et

$$1 \oplus A = \overline{A}$$
$$0 \oplus A = A$$

Le théorème de DE MORGAN complète les propriétés de l'algèbre de BOOLE. Il est indépendant du nombre de variables. Il s'énonce des deux manières suivantes :

1) La négation d'un produit de variables est égale à la somme des négations des variables. Par exemple :

$$\overline{A.B} = \overline{A} + \overline{B}$$

$$\overline{(\overline{A + B}).(A.\overline{B})} = \overline{(\overline{A + B})} + \overline{(A.\overline{B})}$$

2) La négation d'une somme de variables est égale au produit des négations des variables. Par exemple :

$$\overline{A + B} = \overline{A}.\overline{B}$$

$$\overline{(\overline{A + B}) + (A.\overline{B})} = \overline{(\overline{A + B})}.\overline{(A.\overline{B})}$$

D'une manière plus générale, on peut dire que $\overline{f(A, B, C, etc, +, .)} = f(\overline{A}, \overline{B}, \overline{C}, etc, ., +)$

2.1.3 Expression d'une fonction logique

Il existe deux méthodes pour exprimer une fonction logique : soit donner directement son équation logique (par exemple, $S = A + \overline{C} + \overline{A.B}.C$), soit utiliser une table de vérité. A un nombre fini N de variables d'entrée correspond 2^N combinaisons possibles. La table de vérité va donc indiquer la valeur de la fonction pour les 2^N valeurs possibles. Si par exemple S est une fonction de trois variables, il y aura 2^3 soit 8 combinaisons possibles. On va placer les trois variables dans un ordre arbitraire (à ne pas modifier ensuite !) A, B, C de gauche à droite par exemple et écrire les combinaisons dans l'ordre des entiers naturels (0, 1, 2, 3, …, 2^N-1). La table de vérité de S sera par exemple la suivante :

entrées					Sortie
Valeur entière	A	B	C	combinaison	S
0	0	0	0	$\overline{A}.\overline{B}.\overline{C}$	0
1	0	0	1	$\overline{A}.\overline{B}.C$	1
2	0	1	0	$\overline{A}.B.\overline{C}$	1
3	0	1	1	$\overline{A}.B.C$	1
4	1	0	0	$A.\overline{B}.\overline{C}$	0
5	1	0	1	$A.\overline{B}.C$	1
6	1	1	0	$A.B.\overline{C}$	0
7	1	1	1	A.B.C	0

Sous sa forme complète, l'équation logique de S se lit directement. C'est la somme du ET logique de chaque combinaison avec l'état de S correspondant. Ici, on obtient la forme canonique complète :

$$S = \overline{A}.\overline{B}.\overline{C}.0 + \overline{A}.\overline{B}.C.1 + \overline{A}.B.\overline{C}.1 + \overline{A}.B.C.1 + A.\overline{B}.\overline{C}.0 + A.\overline{B}.C.1 + A.B.\overline{C}.0 + A.B.C.0$$

On sait que 0.X = 0, donc on peut éliminer les termes qui valent 0. Cela nous donne :

$$S = \overline{A}.\overline{B}.C.1 + \overline{A}.B.\overline{C}.1 + \overline{A}.B.C.1 + A.\overline{B}.C.1$$

On sait aussi que 1.X = X, donc on peut écrire la forme canonique abrégée :

$$S = \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + \overline{A}.B.C + A.\overline{B}.C$$

C'est simplement la somme de combinaisons pour lesquelles S vaut 1. On la note aussi $\sum(1,2,3,5)$.

Nous avons maintenant un ensemble de règles algébriques et nous savons exprimer une fonction logique. La question importante est : comment être certain que l'équation logique de la fonction est simplifiée au maximum, c'est-à-dire qu'elle va utiliser un nombre de portes minimum ?

2.1.4 Simplification des fonctions logiques

L'expression d'une fonction logique sous sa forme la plus simple n'est pas quelque chose d'évident. Trois méthodes sont utilisables :

- Le raisonnement. On cherche, à partir du problème à résoudre, l'expression la plus simple possible. Evidemment, cette méthode ne garantit pas un résultat optimal.
- La table de vérité et les propriétés de l'algèbre de BOOLE. C'est plus efficace, mais il est facile de rater une simplification, notamment quand la fonction est compliquée.
- La méthode graphique des tableaux de Karnaugh. C'est la méthode la plus efficace car elle garantit le bon résultat. Cette méthode est utilisée sous forme informatique dans tous les outils de CAO.

2.1.4.1 Simplification algébrique

Prenons un exemple. On a deux voyants A et B. On veut déclencher une alarme quand au moins un des deux voyants est allumé, c'est-à-dire : soit A allumé avec B éteint, soit B allumé avec A éteint, soit A et B allumés. Par le raisonnement, le lecteur attentif aura trouvé que S = A + B. Ecrivons la table de vérité :

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

On obtient donc :

$$S = \overline{A}.B + A\overline{B} + AB$$

D'où on tire (d'après les propriétés de l'algèbre de BOOLE) :

$$S = A(\overline{B} + B) + B\overline{A} = A.1 + B\overline{A} = A + B\overline{A} = A + B$$

On voit bien que cette méthode de simplification devient peu évidente quand la fonction est plus complexe. Voyons maintenant une méthode plus efficace.

2.1.4.2 Simplification par les tableaux de Karnaugh

Cette méthode permet :

- d'avoir pratiquement l'expression logique la plus simple pour une fonction F.
- de voir si la fonction $\overline{F}$ n'est pas plus simple (cela arrive).
- de trouver des termes communs pour un système à plusieurs sorties, dans le but de limiter le nombre de portes.
- de tenir compte de combinaisons de variables d'entrées qui ne sont jamais utilisées. On peut alors mettre 0 ou 1 en sortie afin d'obtenir l'écriture la plus simple.
- de voir un OU exclusif caché.

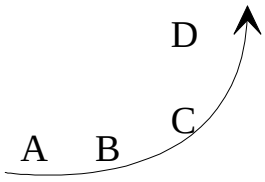
A la main, on traite 4 variables sans difficultés. Au maximum, on peut aller jusqu'à 5 voire 6 variables. En pratique, on utilise un programme informatique qui implémente généralement l'algorithme de QUINE-McCLUSKEY, qui reprend la méthode de Karnaugh, mais de manière systématique et non visuelle (et avec plus de 2 dimensions).

Définition : dans un code adjacent, seul un bit change d'une valeur à la valeur suivante. Exemple avec deux variables :

Code binaire naturel		Code GRAY	
A	B	A	B
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

Les tableaux de Karnaugh sont une variante des tables de vérité. Ils sont organisés de telle façon que les termes à 1 (ou à 0) adjacents soient systématiquement regroupés dans des cases voisines, donc faciles à identifier visuellement. En effet, deux termes adjacents (qui ne différent que par une variable) peuvent se simplifier facilement : $A.B.C + A.B.\overline{C}$ = $AB(C + \overline{C})$ = AB. On représente les valeurs de la fonction dans un tableau aussi carré que possible, dans lequel chaque ligne et chaque colonne correspondent à une combinaison des variables d'entrée exprimée avec un code adjacent (le code GRAY en général). Les traits gras correspondent à une zone où la variable vaut 1. Avec de l'habitude, il est inutile d'écrire les valeurs des variables, ces traits gras suffisent à la lecture.

Pour une écriture rapide à partir de la table de vérité, ou directement à partir du cahier des charges, ou à partir d'une équation déjà existante, on peut écrire les variables A, B, C, D dans l'ordre ci-contre. Cela permet une lecture systématique.

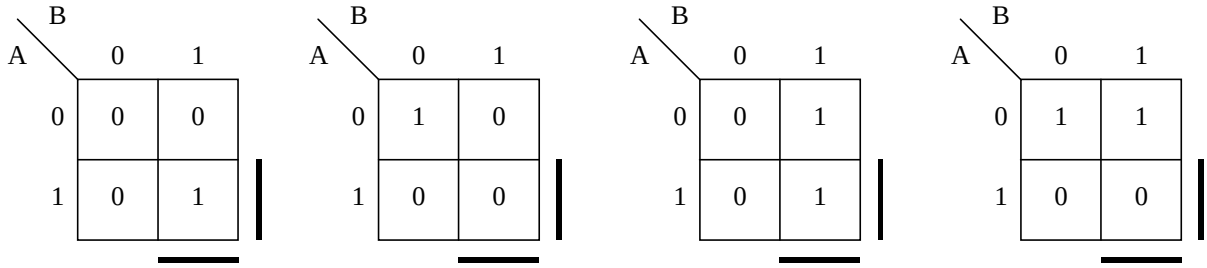


La disposition des 1 et 0 dépend de l'ordre des variables et du code adjacent choisi, mais quel que soit l'ordre, l'équation finale obtenue reste la même.

Pour la lecture et la simplification de l'expression, on cherche des paquets les plus gros possibles (de 1, 2, 4 ou 8 variables), en se rappelant que le code est aussi adjacent sur les bords (bord supérieur avec bord inférieur, bord gauche avec bord droit). On effectue une lecture par « intersection » en recherchant la ou les variables ne changeant pas pour le paquet. On ajoute alors les paquets. On obtient l'expression sous la forme d'une somme de produit. Une case peut être reprise dans plusieurs paquets.

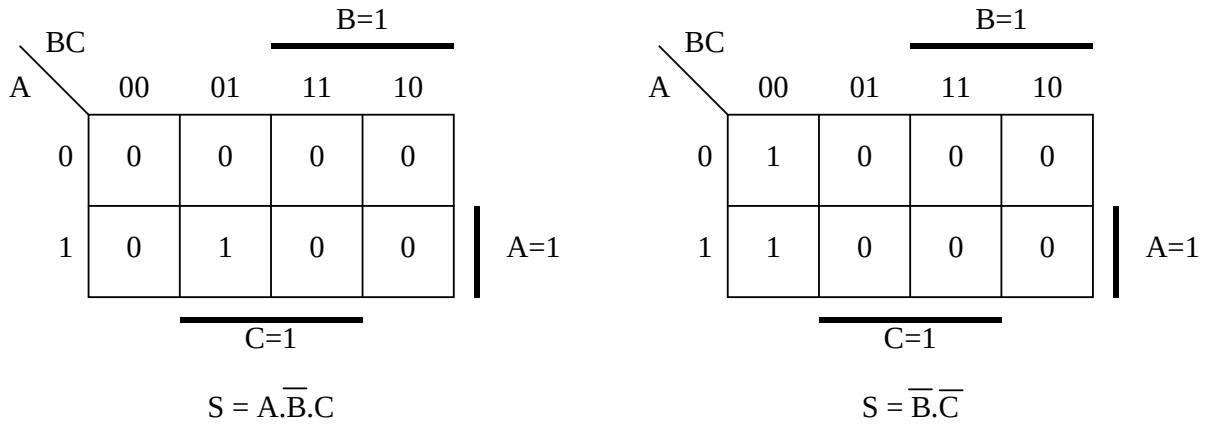
Exemple 1 : deux variables d'entrée A et B

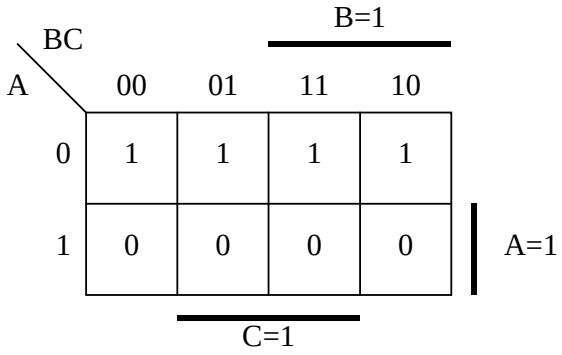
1 case = produit de 2 variables, 2 cases = 1 variable, 4 cases = 1 ou 0.



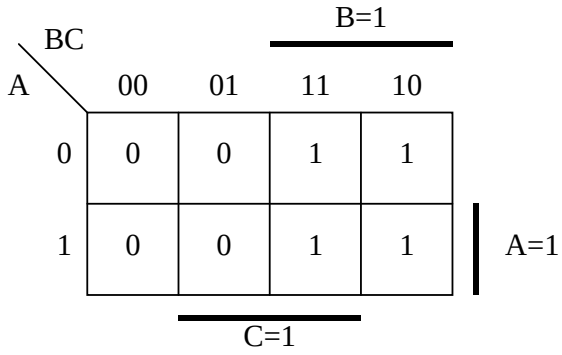
Exemple 2 : trois variables d'entrée A, B et C

1 case = produit de 3 variables, 2 cases = produit de 2 variables, 4 cases = 1 variable, 8 cases = 1 ou 0. Il y a parfois plusieurs solutions équivalentes.

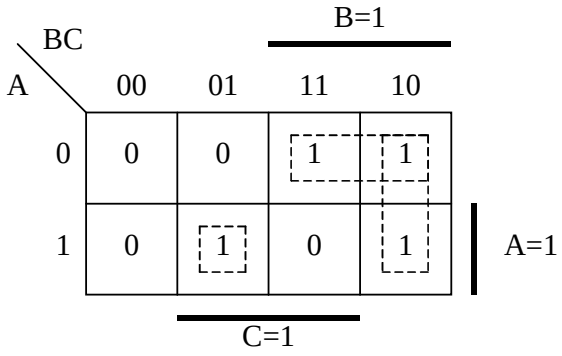




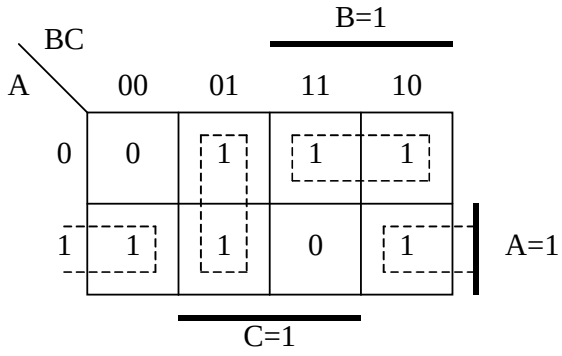
$S = \overline{A}$



$S = B$



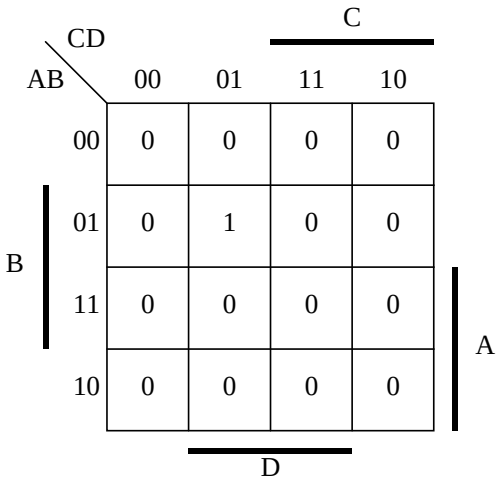
$S = \overline{A}.B+B.\overline{C}+A.\overline{B}.C$



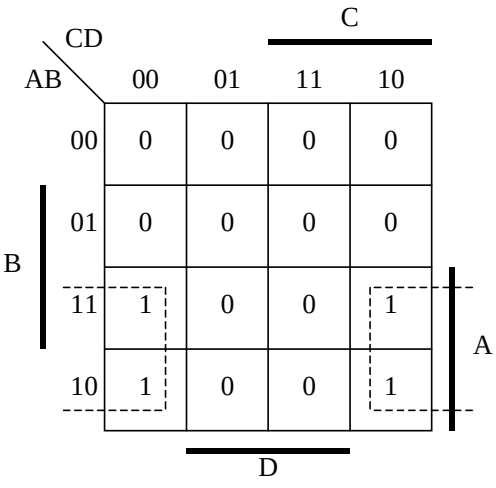
$S = A.\overline{C}+\overline{B}.C+\overline{A}.B$

<u>Exemple 3 : quatre variables d'entrée A, B, C et D</u>

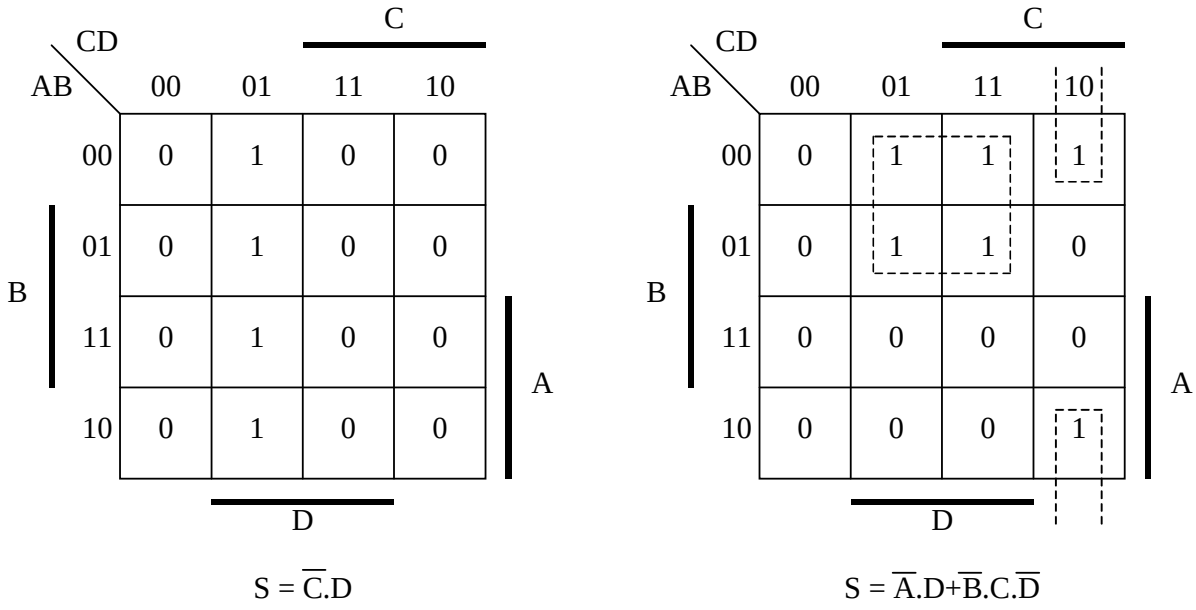
1 case = produit de 4 variables, 2 cases = produit de 3 variables, 4 cases = produit de 2 variables, 8 cases = 1 variable, 16 cases = 1 ou 0.



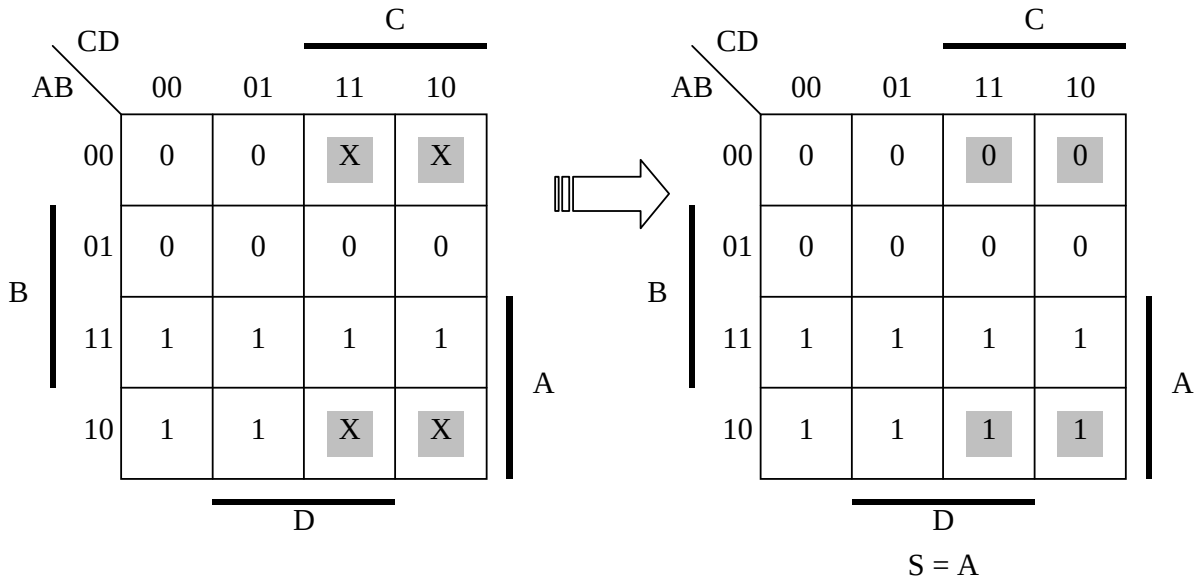
$S = \overline{A}.B.\overline{C}.D$



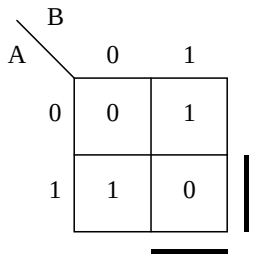
$S = A.\overline{D}$



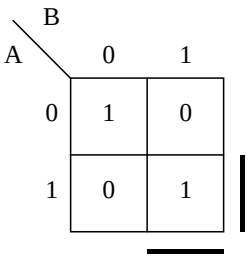
Dans une fonction logique, il peut exister des états non utilisés. Ces combinaisons n'ont pas d'importance pour le problème à résoudre. On les appelle en anglais des états « don't care » (ne pas tenir compte) et on les symbolise par un X. Ces états X peuvent prendre la valeur la plus pratique pour simplifier la fonction. Exemple :



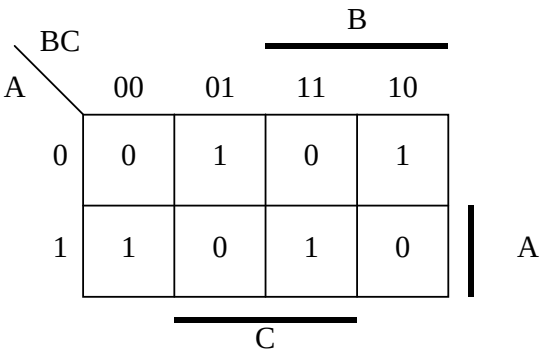
Il existe un cas particulier de fonction qui est parfois difficile à détecter, le OU exclusif (XOR) ou son complément. Il faut repérer des 1 disposés en quinconce. Pour voir s'il s'agit du XOR ou de son complément, il suffit alors de regarder la case « toutes entrées à 0 ». Si elle vaut 0, c'est un XOR, sinon c'est un XNOR.



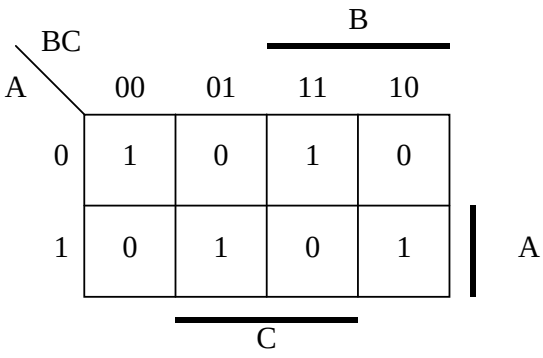
S = A⊕B



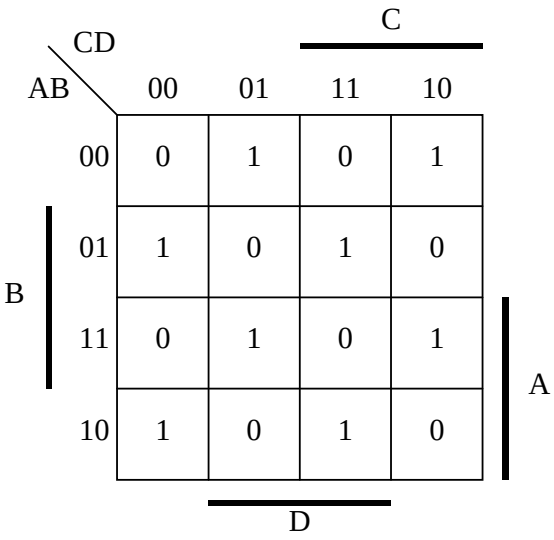
S = $\overline{A \oplus B}$



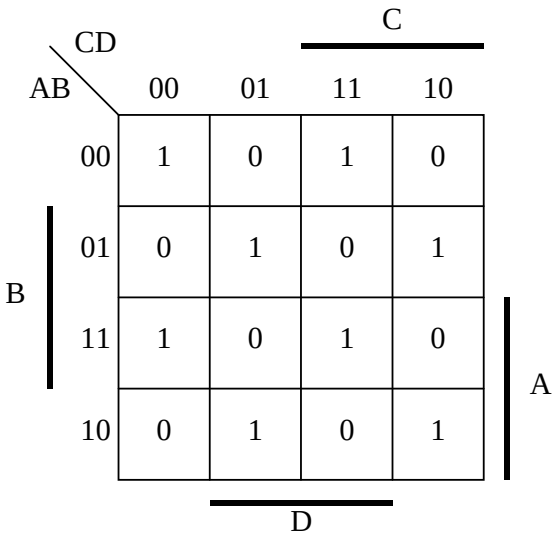
S = A⊕B⊕C



S = $\overline{A \oplus B \oplus C}$

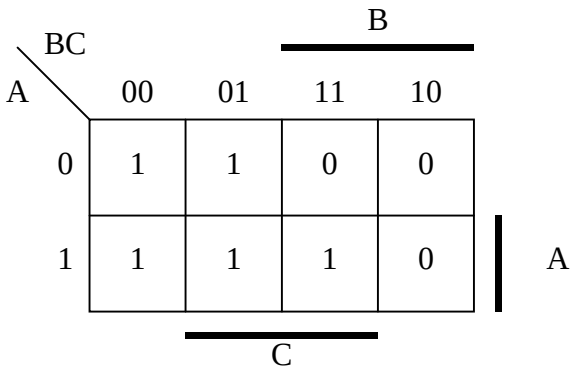


S = A⊕B⊕C⊕D



S = $\overline{A \oplus B \oplus C \oplus D}$

Il peut parfois être plus pratique de lire les 0 dans le tableau plutôt que les 1. On obtient alors $\overline{F}$ = somme de produits ce qui implique (par DE MORGAN) que F = produit de somme des variables inversées. Si par exemple on a :



En comptant les 1 : $S = \overline{B} + A.C$

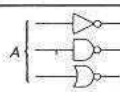
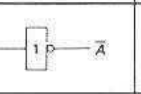
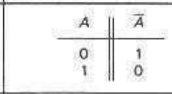
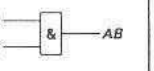
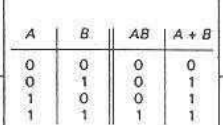
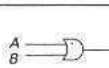
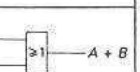
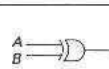
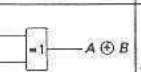
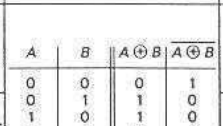
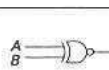
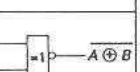
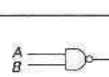
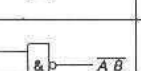
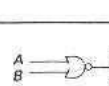
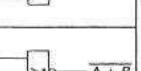
En comptant les 0 au lieu des 1, on obtient : $\overline{S} = B.\overline{C} + \overline{A}.B$. D'où on tire : $\overline{\overline{S}} = S = \overline{B.\overline{C} + \overline{A}.B}$ ce qui donne finalement : $S = (\overline{B} + C).(A + \overline{B})$. On peut utiliser la lecture directe sur les 0 quand il y a peu de 0 et beaucoup de 1.

2.2 Circuits logiques combinatoires

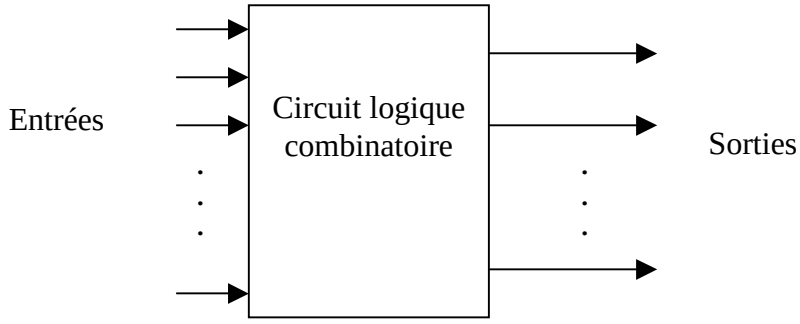
2.2.1 Circuits logiques fondamentaux

Il existe deux manières de dessiner ces opérateurs. La notation américaine est la plus utilisée (puisque les fabricants de composants sont souvent américains) et donc la plus connue mais la notation normalisée à l'avantage d'être plus facile à dessiner avec une table traçante. Dans ce cours, nous utiliserons uniquement la notation américaine. Le tableau suivant résume ce que nous avons déjà vu. Ces circuits logiques existent avec 2, 3 voire 4 entrées.

Tableau I. – Opérations logiques élémentaires.

Opération	Symbole usuel	Symbole normalisé	Table de vérité	Tableau de Karnaugh
NOT - INV				
AND - ET				
OR - OU				
XOR - OU Exclusif				
XNOR - NON OU Exclusif				
NAND - NON ET				
NOR - NON OU				

Les circuits logiques combinatoires sont des circuits constitués des portes ci-dessus fonctionnant simultanément et réalisant une ou plusieurs fonctions logiques.



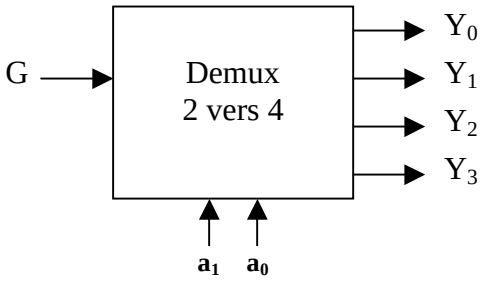
A une combinaison d'entrées (l'entrée) ne correspond qu'une seule combinaison de sorties (la sortie). La « sortie » apparaît après application de l' « entrée » avec un certain retard qui est le temps de propagation dans la logique interne. Ce temps est déterminé par la technologie utilisée, le nombre de portes traversées et la longueur des interconnections métalliques.

Les circuits combinatoires peuvent servir par exemple :

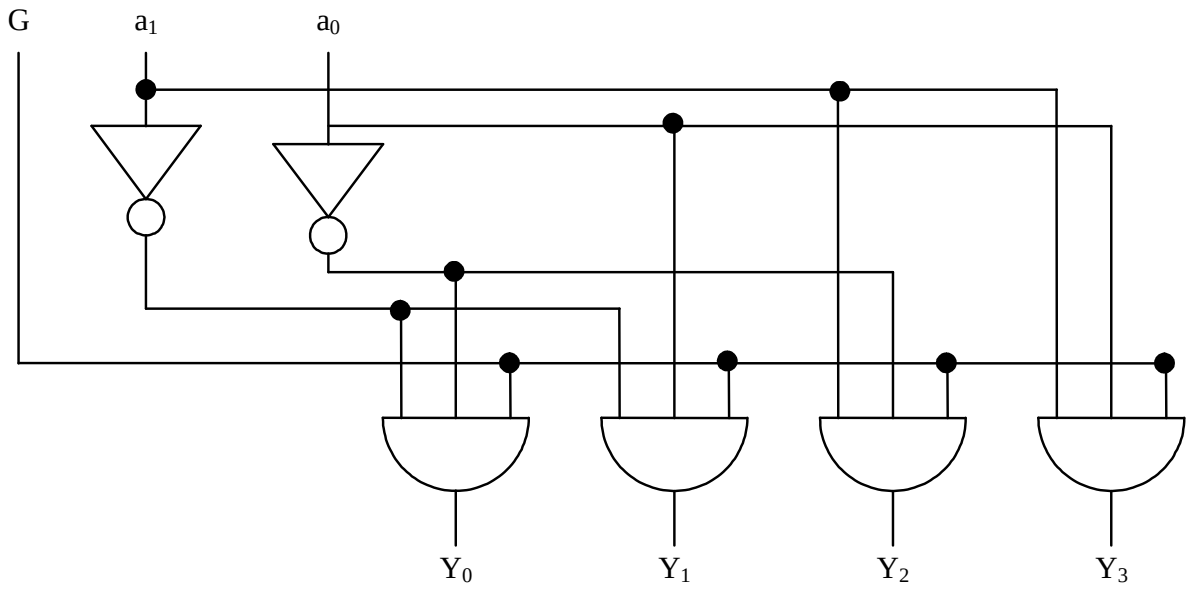
- à traduire des bits en chiffres représentant un nombre (ou des lettres ou un code particulier). On appelle ces circuits des codeurs (ou bien des décodeurs pour l'opération inverse). Par exemple, un codeur Gray ou bien BCD.
- à effectuer des opérations arithmétiques sur des nombres. Par exemple, un additionneur ou un multiplieur.
- à transmettre ou recevoir des informations sur une ligne unique de transmission (une ligne série), ce qui nécessite de transformer un nombre écrit sous forme parallèle en une suite de bits mis en série et vice-versa. C'est le rôle des circuits multiplexeur/démultiplexeur. Voici par exemple une transformation série/parallèle suivie d'une transformation parallèle/série :

2.2.2 Le démultiplexeur

C'est un circuit qui aiguille une entrée vers une sortie dont on donne l'adresse sous forme d'un nombre codé en binaire.



Le schéma fondamental d'un démultiplexeur est :



Sa table de vérité est égale à :

N	a_1	a_0	$Y_0 = \overline{a_0}.\overline{a_1}.G$	$Y_1 = a_0.\overline{a_1}.G$	$Y_2 = \overline{a_0}.a_1.G$	$Y_3 = a_0.a_1.G$
0	0	0	G	0	0	0
1	0	1	0	G	0	0
2	1	0	0	0	G	0
3	1	1	0	0	0	G

On obtient donc l'équation suivante : $Y_K(N) = G$ si $K = N$

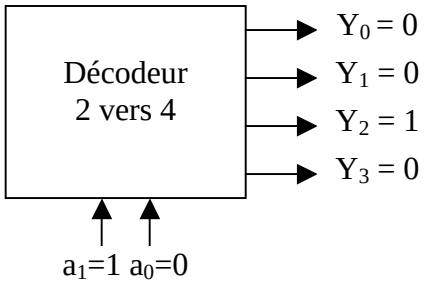
$Y_K(N) = 0$ si $K \neq N$

Si G est une donnée, le circuit est un démultiplexeur (un aiguillage divergent). Son utilisation première est la conversion série/parallèle des données. Il peut aussi servir pour réaliser des fonctions combinatoires.

La limitation de ce type de circuit est le nombre d'entrées/sorties qui croit fortement avec le nombre de variables d'adresse N. Il évolue en $N + 2^N + 1$.

2.2.3 Le décodeur

Le décodeur est de la même famille que le démultiplexeur, mais avec l'entrée G qui vaut 1 en permanence. On trouve alors en sortie un 1 parmi des 0.



On appelle aussi ce circuit un décodeur d'adresse utilisé pour adresser les différentes lignes d'une mémoire. Sa table de vérité est égale à :

N	a_1	a_0	$Y_0 = \overline{a_0}.\overline{a_1}$	$Y_1 = a_0.\overline{a_1}$	$Y_2 = \overline{a_0}.a_1$	$Y_3 = a_0.a_1$
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

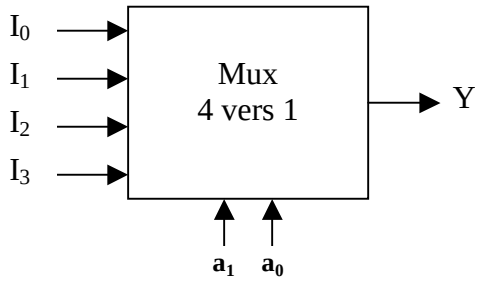
On obtient donc l'équation suivante : $Y_K(N) = 1$ si $K = N$

$Y_K(N) = 0$ si $K \neq N$

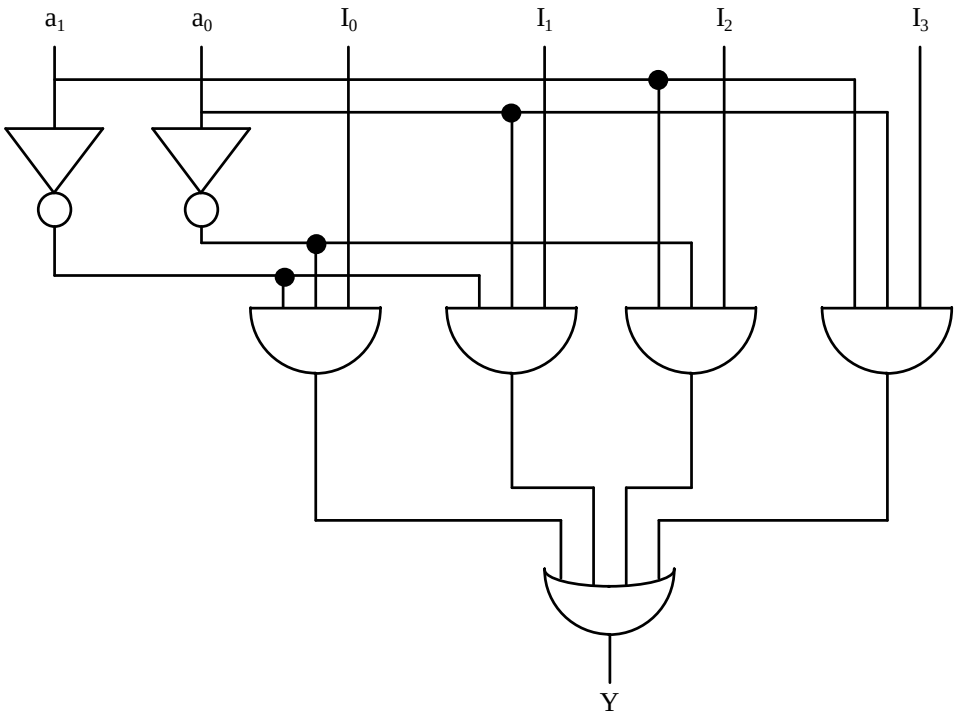
Comme pour le démultiplexeur, la limitation de ce type de circuit est le nombre d'entrées/sorties qui croit fortement avec le nombre de variables d'adresse N.

2.2.4 Le multiplexeur

Le multiplexeur est la fonction inverse du démultiplexeur. C'est un sélecteur de données ou aiguillage convergent. Il peut transformer une information apparaissant sous forme de n bits en parallèle en une information se présentant sous forme de n bits en série. La voie d'entrée, sélectionnée par son adresse, est reliée à la sortie.



Le schéma logique est le suivant :



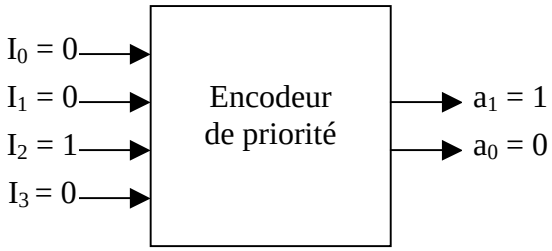
Son équation de sortie est égale à : $Y = I_0.\overline{a_1}.\overline{a_0} + I_1.\overline{a_1}.a_0 + I_2.a_1.\overline{a_0} + I_3.a_1.a_0$ avec la table de vérité :

	a_1	a_0	Y
0	0	0	I_0
1	0	1	I_1
2	1	0	I_2
3	1	1	I_3

Un multiplexeur peut servir à réaliser des fonctions logiques quelconques. La limitation de ce type de circuit est toujours le nombre d'entrées/sorties qui croit fortement avec le nombre de variables d'adresse N (de la même manière que le démultiplexeur).

2.2.5 L'encodeur de priorité

L'encodeur est la fonction inverse du décodeur. On met sur les entrées un 1 parmi des 0 et on obtient sur les sorties l'adresse de l'entrée à 1. La priorité ne sert que quand plusieurs entrées sont à 1 en même temps. Le circuit donne alors l'adresse de l'entrée dont le rang est le plus élevé. Il y a donc priorité aux entrées de rang le plus élevé.

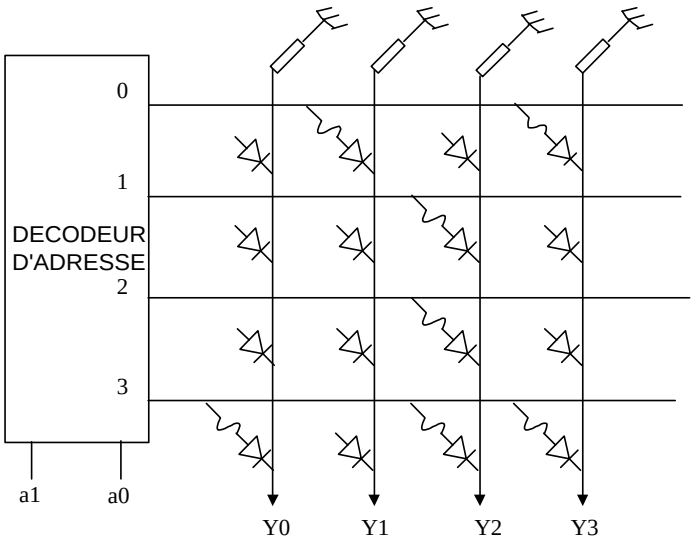


Sa table de vérité est égale à :

I_3	I_2	I_1	I_0	a_1	a_0
0	0	0	0	0	0
1	X	X	X	1	1
0	1	X	X	1	0
0	0	1	X	0	1
0	0	0	1	0	0

2.2.6 Les mémoires

Les PROM sont des mémoires mortes programmables. Le schéma ci-dessous montre un exemple de PROM bipolaire à fusible (procédé obsolète depuis la fin des années 1970).



L'adresse appliquée sur les entrées (a1 a0) provoque, via le décodeur d'adresses, la mise à 1 de la ligne d'adresse correspondante. Si le fusible existant entre cette ligne et la sortie Yn est intact, cette sortie vaut 1. Si le fusible a été claqué pendant la programmation, la sortie vaut 0 (à travers une résistance de pull-down). La programmation est réalisée avec un programmateur d'EPROM, le fichier contenant la configuration pouvant être au format JEDEC (Joint Electronics Design Engineering Council). Dans cet exemple, la PROM (4 mots de 4 bits) contient les données suivantes :

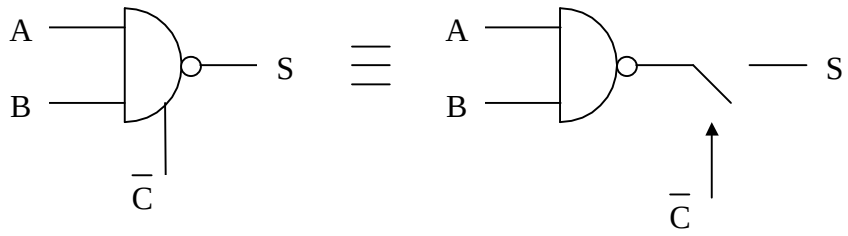
a1	a0	Y0	Y1	Y2	Y3
0	0	0	1	0	1
0	1	0	0	1	0
1	0	0	0	1	0
1	1	1	0	1	1

On n'utilise plus de PROM pour réaliser directement une fonction logique mais le principe, lui, demeure tout à fait d'actualité. Certains circuits, comme les circuits logiques programmables par exemple, utilisent en interne de mémoires pour réaliser des fonctions logiques.

2.2.7 Buffer bidirectionnel 3 états

2.2.7.1 La porte 3 états

Un signal de commande peut être ajouté à une porte logique afin de mettre sa sortie à l'état haute impédance (la sortie est en l'air). Par exemple, quand C = 0, les transistors de sortie sont tous ouverts et la sortie est flottante. Quand C = 1, la porte fonctionne normalement. Dans le cas d'une NAND, cela donne le schéma logique équivalent suivant :

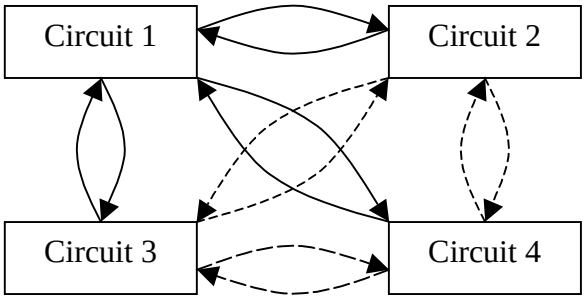


Quand la sortie S est à l'état haute impédance, on dit qu'elle est à l'état Z (ou HiZ).

2.2.7.2 Notion de bus

Une question fondamentale de l'électronique numérique est : comment faire dialoguer plusieurs circuits intégrés numériques entre eux de la manière la plus efficace possible ? Deux solutions sont possibles : la liaison point à point ou le bus.

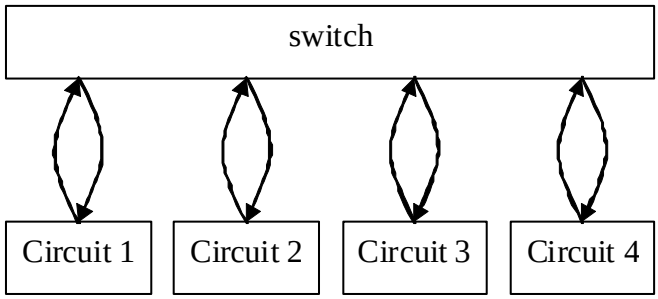
1. **La liaison point à point**. Pour chaque liaison, on tire une paire de fils, un fil qui envoie des données, l'autre fil qui en reçoit. On n'utilise pas l'état Z dans ce cas. Chaque circuit étant relié indépendamment avec ses voisins, le nombre de fils augmente très rapidement. Voici un exemple avec 4 circuits.



   On peut compter 1 + 2 + 3 = 6 paires de fils pour assurer la liaison point à point. Dans le cas général, avec N circuits, on a 1 + 2 + 3 + 4 + …+ N-2 + N-1 liaisons ce qui donne

$\frac{N^2 - N}{2}$ paires de fils pour N circuits. On voit que si on veut transmettre 32 bits en même temps, il faut 32 paires par liaison et le nombre de fils à tirer entre les circuits augmente comme 64 x (nombre de circuits)^2. Cette solution n'est pas valable sauf si on limite le nombre de circuits (5 ou 6 maximum) et le nombre de fils par paire (2 ou 4 au maximum). Mais le débit devient alors trop faible à moins d'augmenter massivement la fréquence de transfert.

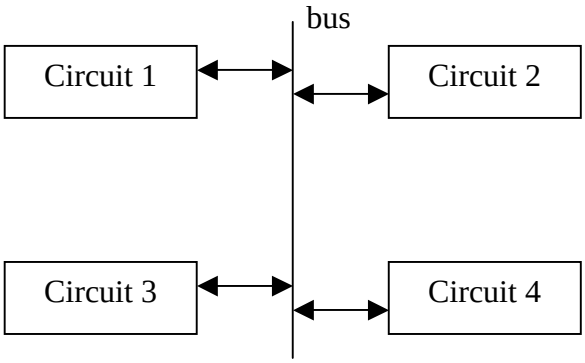
C'est pourtant la solution utilisée aujourd'hui en informatique. Dans les PC, on est passé en 10 ans d'interfaces parallèles (port parallèle, PATA, PCI,…) à des interfaces séries (USB, SATA, PCI Express, …). Pour cela, on a utilisé des technologies de type réseau Ethernet avec deux apports fondamentaux : le switch (concentrateur) et une montée massive en débit (2.5 Gbit/s en PCI-E 1.0). Le montage devient alors le suivant :



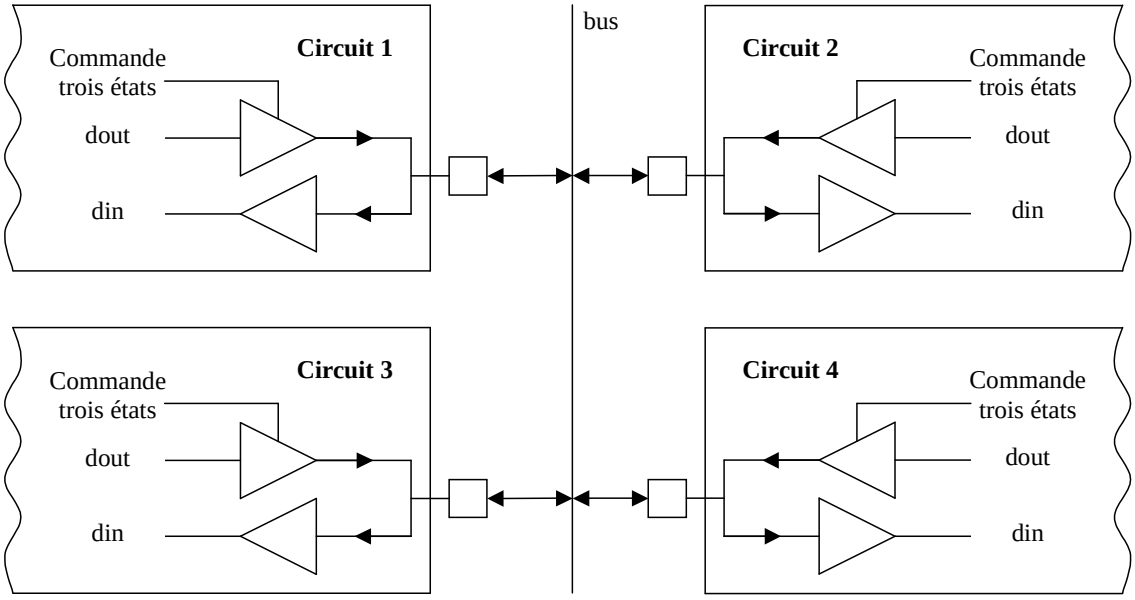
Le nombre de paires de fils augmente maintenant linéairement avec le nombre de circuits, car c'est le switch qui connecte (intelligemment) les circuits qui veulent échanger de l'information. Grâce au débit élevé (200 Mo/s net par fil en PCI-E 1.0) sur plusieurs paires (1x, 2x, 4x, 8x, 16x ou 32x), les performances sont excellentes.

Il y a bien évidemment un problème d'horloge pour lire les données. En effet, il est impossible que tous les circuits partagent la même horloge à 2.5 GHz parfaitement en phase pour lire les données. La deuxième innovation qui se cache derrière PCI-E est le fait que chaque liaison série (chaque fil) porte les informations d'horloge nécessaires au décodage des données grâce à un code 8B/10B. Chaque récepteur va pouvoir resynchroniser son horloge de décodage sur le flux de données reçu car chaque liaison à 2.5 Gbit/s porte ses propres informations d'horloge.

2. **Le bus**. Dans ce cas, tous les circuits sont branchés simultanément sur le même fil.



Tous les circuits ont la capacité de générer un état sur le bus ou bien de lire une valeur. Pour que cela soit possible, il faut que chaque broche soit bidirectionnelle, c'est-à-dire qu'elle fonctionne aussi bien en entrée qu'en sortie. Il faut donc connecter en parallèle un buffer d'entrée avec un buffer de sortie. Si on veut que l'entrée puisse lire une valeur envoyée par un autre circuit, il faut obligatoirement que le buffer de sortie cesse de générer un niveau sur le fil. La solution consiste à faire passer ce buffer à l'état haute impédance. L'ensemble forme un buffer bidirectionnel 3 états.



Quand un circuit veut envoyer un bit sur le bus, les autres circuits doivent mettre leur sortie à l'état Z et lire la valeur. Si deux circuits essayent de générer un niveau opposé en même temps, il y a conflit de bus et l'état devient indéterminé. Il y a donc forcément un maître de bus (par exemple, un microprocesseur) indiquant qui a le droit de générer sur le bus et qui

force les autres circuits à passer à l'état Z. En pratique, **un bus est composé de N fils de même nature**. Un bus d'adresses par exemple qui sera unidirectionnel ou encore un bus de données qui sera lui bidirectionnel. **Le bus de données est un bus parallèle, la bande passante est partagée entre les circuits.** Le bus PCI est un exemple typique de bus parallèle. Il a atteint la fréquence de 66 MHz avec 64 bits de données, soit un débit de 528 Mo/s partagé entre toutes les cartes. La montée en fréquence au-delà des 100 MHz devenant problématique, on a choisi PCI-Express pour le remplacer à partir de 2005.

2.3 Caractéristiques temporelles

Nous étudierons ici les différents temps que l'on trouve dans les circuits combinatoires.

Dans les notices du constructeur, les temps min ou max (selon le cas) sont donnés pour le plus mauvais des circuits, et dans le cas le plus défavorable (à température maximale et tension d'alimentation minimale). Une étude faite avec ces temps fournit un montage fonctionnant dans 100% des cas. Les temps typiques sont moins contraignants mais ils ne représentent pas le cas le plus défavorable. Ils peuvent être utilisés mais avec un plus grand risque.

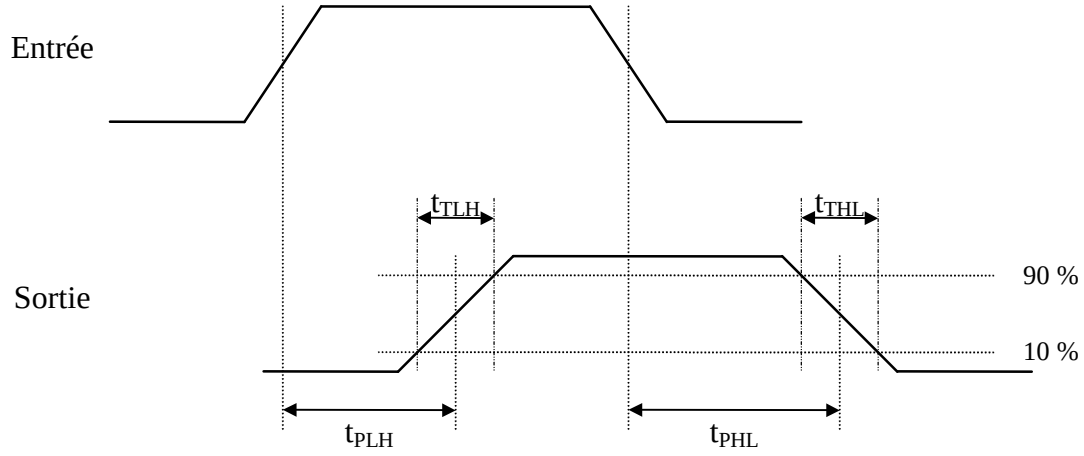
2.3.1 Caractéristiques temporelles

Considérons une porte logique. Les temps de transition et de propagation sont définis de la manière suivante :

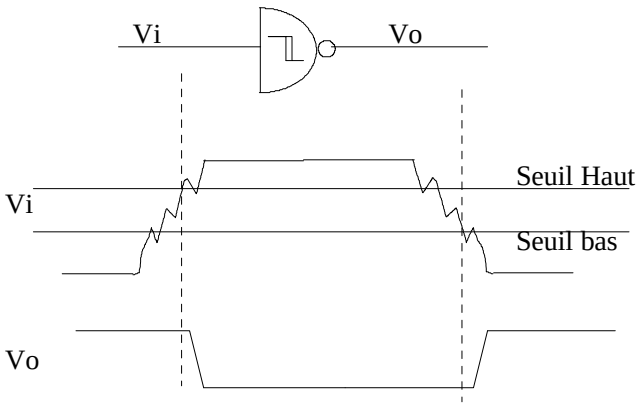
- le temps de propagation est le retard entre les signaux d'entrée et de sortie. Il est causé par le temps de traversée des transistors et des fils formant le circuit logique. Il en existe deux types : t_{PLH} (la sortie passe de 0 à 1 « low to high ») et t_{PHL} (la sortie passe de 1 à 0 « high to low »). En règle générale, ces deux temps sont aujourd'hui égaux (en CMOS).

- le temps de transition est le temps mis par une sortie pour changer d'état. Il est généralement pris entre 10 et 90 % du niveau maximum. Il en existe deux types : t_{TLH} (la sortie passe de 0 à 1) et t_{THL} (la sortie passe de 1 à 0). Ce temps est très dépendant de la charge (capacitive notamment) sur la sortie du circuit. En règle générale, ces deux temps sont aujourd'hui identiques (en CMOS).

Le dessin suivant en donne une illustration pour une porte non-inverseuse :



A l'entrée d'un circuit logique dépourvu d'un déclencheur à seuil (trigger de Schmitt), on doit respecter un temps de transition maximum (t_{mmax}), sous peine de transitions parasites en sortie. Dans le cas de signaux à temps de transitions trop longs ou présentant des parasites, une porte pourvue d'un trigger permet de mettre en forme ces signaux pour qu'ils puissent attaquer correctement la logique :

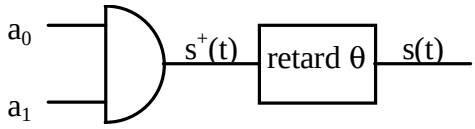


En sortie de circuit, les temps de transitions dépendent de la technologie employée mais pas du temps de montée du signal d'entrée, tant que celui-ci reste inférieur à 3 à 5 fois les temps de transitions usuels de cette technologie.

2.3.2 Etats transitoires

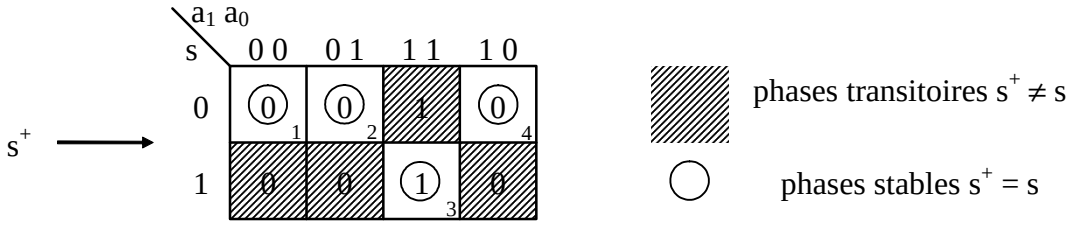
Le fonctionnement normal d'une carte logique combinatoire ne peut être déduit du fonctionnement des circuits qui la composent en supposant ces circuits idéaux (temps de transition nuls, temps de propagation nuls). Dans ce paragraphe, nous allons proposer une

méthode d'analyse des aléas de commutation en tenant compte des temps de propagation. Pour cela, nous allons définir le modèle asynchrone d'un circuit combinatoire :

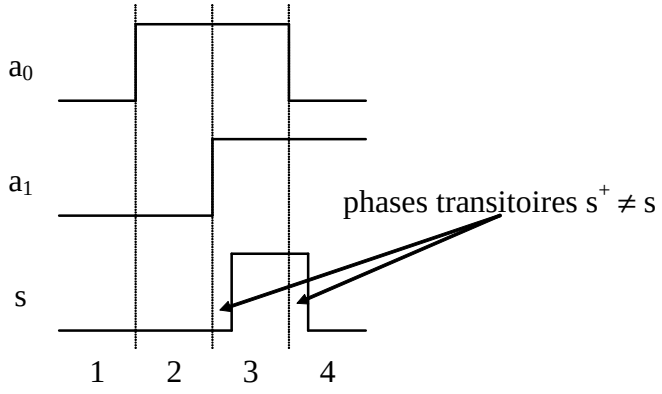


avec : $\theta = \sup(t_{PLH}, t_{PHL})$
$s(t) = s^+(t - \theta)$ ou
$s(t + \theta) = s^+(t)$

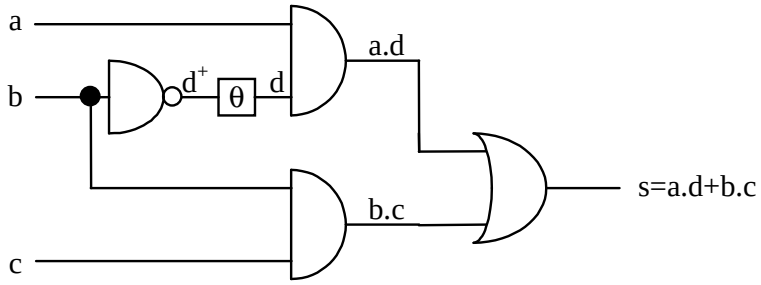
Il faut que la durée de l'impulsion active sur l'entrée d'un circuit combinatoire soit supérieure au retard θ pour que la sortie change d'état. Sinon, l'énergie de l'impulsion est insuffisante pour provoquer la commutation. Pour analyser le fonctionnement d'un circuit combinatoire à N entrées et une sortie, on détermine la table d'excitation du circuit. C'est un tableau de Karnaugh à N+1 variables (les entrées plus la sortie s) permettant de déterminer s^+. Quand s^+ est différent de s dans ce tableau (zone hachurée), nous sommes en présence d'un régime transitoire (s va changer après un temps θ). Quand s^+ est égal à s (valeur encerclée), nous sommes dans un état stable (s reste dans le même état).



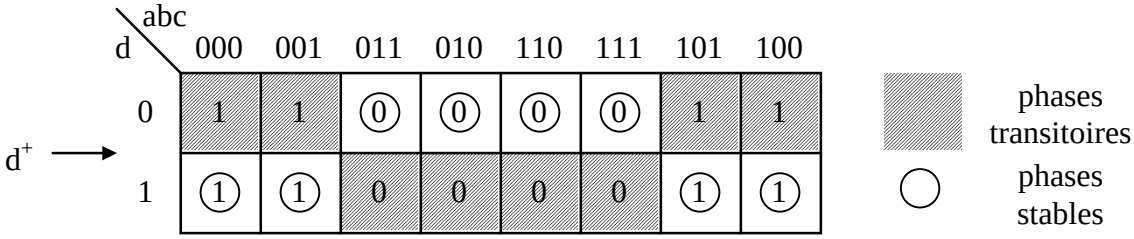
On se sert de cette table d'excitation pour établir le chronogramme du circuit à partir d'une séquence d'entrée quelconque (par exemple 0,0 ; 0,1 ; 1,1 ; 1,0) :



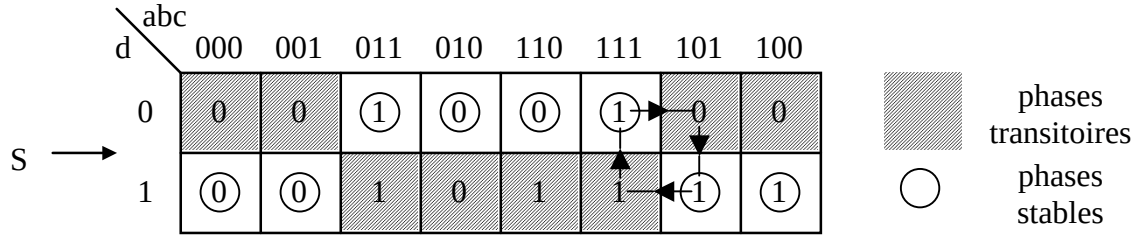
Voici un exemple de prévision des aléas de commutation. Soit le circuit suivant :



Pour la méthode d'analyse, toutes les portes sont à retard nul et un retard θ n'est ajouté que sur l'inverseur. La table d'excitation de d^+ en fonction de a, b, c, d est ($d^+ = \overline{b}$) :



On en déduit la table de vérité de la sortie du circuit (en conservant la grille des phases transitoires/stables :

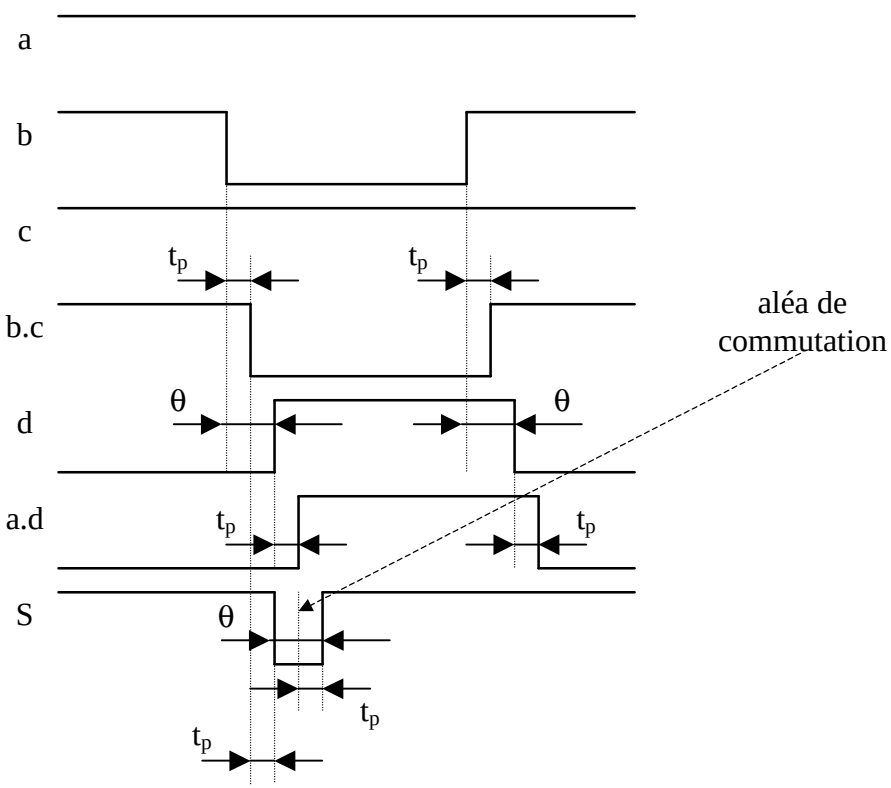


Etudions maintenant la transition b = 1, 0, 1 avec a et c qui restent en permanence à 1. On a alors l'évolution suivante en sortie :

abc = 111	S = 1	
abc = 101	Phase transitoire : S = 0 Etat final : S = 1.	Aléa de commutation
abc = 111	Phase transitoire : S = 1 Etat final : S = 1.	Pas d'aléa

Sans utiliser cette méthode d'analyse, on aurait trouvé S = 1. En effet, cet aléa de commutation n'apparaît pas sur la table de vérité purement combinatoire de S. On nomme cette impulsion dont la durée ne dépend que du temps de propagation θ de l'inverseur, un « glitch ».

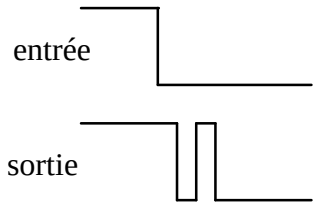
Dans l'analyse précédente, on a pris un temps de propagation nul pour les portes autres que l'inverseur afin de simplifier le calcul. Introduire un temps de propagation t_p sur ces portes ne change rien au résultat final comme nous le montre le chronogramme suivant. L'aléa est simplement décalé de $2.t_p$, mais sa durée reste égale à θ.



Quand un circuit logique combinatoire a la possibilité de produire cet état transitoire, on dit qu'il existe un « hazard » (que le glitch se produise ou non). Ces notions de glitch et de hazard ne s'appliquent que pour des circuits logiques combinatoires purs, c'est-à-dire sans rebouclage des sorties sur les entrées. Deux types de hazard existent :

- **Hazard statique.** Un hazard statique à 1 est la possibilité pour un circuit de produire un glitch à 0 alors que la sortie aurait normalement dû rester à 1. C'est le cas dans l'exemple précédent. Un hazard statique à 0 est la possibilité pour un circuit de produire un glitch à 1 alors que la sortie aurait normalement dû rester à 0.

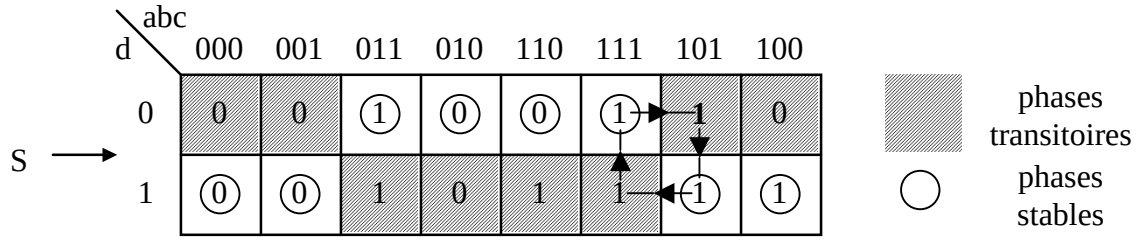
- **Hazard dynamique.** Un hazard dynamique est la possibilité que la sortie d'un circuit change plus d'une fois pour un seul changement logique sur une entrée.



  De multiples transitions sur une sortie peuvent se produire s'il y a plusieurs chemins logiques avec des retards différents entre l'entrée et la sortie qui changent.

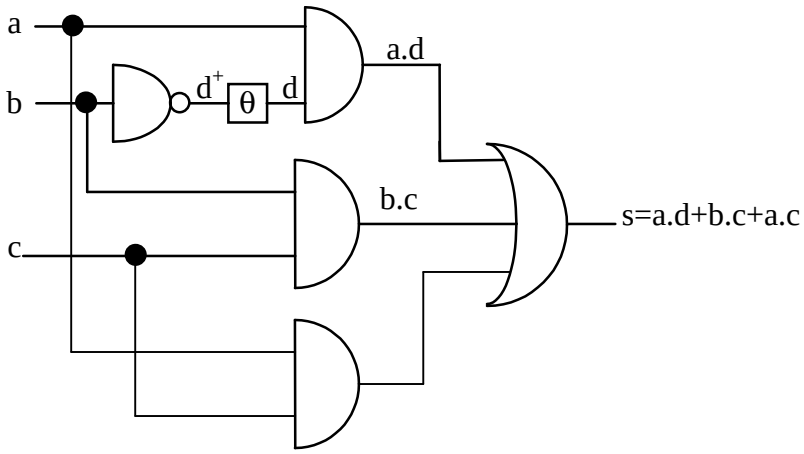
Il est important de noter que le hazard statique ou dynamique est borné dans le temps par rapport au dernier changement sur les entrées. Une fois que le chemin combinatoire qui traverse le plus de portes logiques (**le chemin critique**) a été traversé, le signal est forcément stable en sortie. Le temps de traversée du chemin critique s'appelle **le temps critique**. ***Le signal en sortie du circuit combinatoire est stable « temps critique » après le dernier changement sur les entrées.*** Il ne peut se produire aucun changement passé ce délai.

Pour la conception de circuits logiques séquentiels (avec rebouclage des sorties sur les entrées), il faut utiliser des circuits logiques combinatoires sans hazards (« hazard-free circuits »). Il est possible d'éliminer les hazards d'un circuit en utilisant la méthode d'analyse vue précédemment et en modifiant le tableau de karnaugh afin d'en éliminer les états transitoires. En effet, les phases transitoires du tableau de S peuvent être modifiées sans que cela influe sur l'état stable de S. Cela ne change S que pendant les transitions. Dans notre exemple, il suffit de changer l'état abcd = 1010 en passant S de 0 à 1 pour supprimer le glitch.



Cela revient à modifier l'équation de S qui est maintenant S = a.d + b.c + a.c. La transition b = 1, 0, 1 avec a et c qui restent en permanence à 1 ne peut plus provoquer d'aléa de

commutation puisque S vaut 1 dans tous les cas. Le schéma du circuit « hazard-free » est donc le suivant. La méthode marche, mais elle sera difficilement utilisable en pratique.



Remarque : il est important de noter que pour une porte logique élémentaire, il ne peut y avoir de glitch en sortie si une seule de ses entrées change à la fois. C'est la même chose pour une LUT de FPGA

2.4 Description en VHDL

2.4.1 Introduction

Il existe deux langages de description matériel (HDL : Hardware Description Language), VHDL et Verilog HDL. Le langage VHDL est assez aride à apprendre, verbeux et fortement typé (pas de mélange de types de donnée sans fonction de conversion), c'est un dérivé du langage ADA. Verilog HDL dérive quant à lui du langage C avec tous ses avantages (comme la concision) mais aussi tous ses inconvénients : il est notamment plus ambigu que VHDL, c'est-à-dire que certaines descriptions peuvent être interprétées de plusieurs manières. Verilog est le langage de conception d'ASIC le plus utilisé dans le monde sauf en Europe ou VHDL est aussi très utilisé. Pour les FPGA, c'est VHDL qui est le plus utilisé sauf par les ingénieurs de conception ASIC quand ils se mettent aux FPGA.

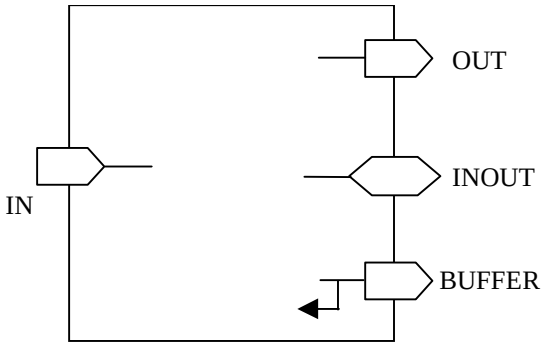
Pour apprendre VHDL, nous allons utiliser la méthode des exemples commentés. Nous allons reprendre les circuits logiques combinatoires vus précédemment et voir leur description en VHDL. Nous introduirons les caractéristiques du langage au fur et à mesure de la description des circuits. Nous reverrons de plus toutes ces notions en travaux pratiques. Le langage VHDL n'est pas sensible aux différences majuscules/minuscules (case insensitive).

2.4.2 Portes combinatoires

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity GATE is
4.   port(D1, D2, D3 : in  std_logic;
5.        Y1, Y2, Y3, Y4, Y5 : out std_logic);
6. end GATE;

7. architecture RTL of GATE is
8.   signal tmp : std_logic;
9. begin
10.  Y1 <= D1 nor D2;
11.  Y2 <= not (D1 or D2 or D3);
12.  Y3 <= D1 and D2 and not D3;
13.  Y4 <= D1 xor (D2 xor D3);
14.  tmp <= D1 xor D2;
15.  Y5 <= tmp nand D3;
16.end RTL;
```

- La ligne 1 définit l'appel à la librairie IEEE grâce à l'instruction « `library nom_de_la_librairie` ». La ligne 2 indique l'utilisation de tous les éléments (fonctions, composants ou type comme std_logic) du package std_logic_1164 grâce à l'instruction « `use nom_de_la_librairie.nom_du_package.all` ».

- Lignes 3 à 6 : définition des entrées-sorties du design GATE dans l'entité. Le mot clé port annonce la liste des signaux d'interface. 4 modes sont possibles pour ces entrées-sorties: IN, OUT, INOUT, BUFFER.



En VHDL, un signal déclaré en sortie (OUT) ne peut pas être lu dans l'architecture, même pour un test dans une condition. Il faut le déclarer en BUFFER, ce qui correspond à une sortie dont on peut lire la valeur dans l'architecture. On peut se passer du mode

BUFFER en utilisant un signal temporaire interne, puis en le copiant sur la sortie OUT. En pratique, le mode BUFFER n'est jamais utilisé.

Dans l'entité, les types des signaux peuvent être les suivants :

Types prédéfinis : integer, natural (entier >= 0), positive (entier > 0), bit, bit_vector, boolean, real, time, **std_logic, std_logic_vector.** Exemples :

```
NUM : in integer range -128 to 127;        (–128 < NUM < +127)
NUM1 : in integer;                         (–2^31 < NUM1 < +2^31 - 1)
DATA1 : in bit_vector(15 downto 0);        (DATA1 : bus 16 bits)
DATA2 : out std_logic_vector(7 downto 0);  (DATA2 : bus 8 bits)
```

Types définis par l'utilisateur :

Les types énumérés. Exemples :

```
type ETAT is (UN, DEUX, TROIS);
TOTO : out ETAT;
```

Les tableaux. Exemples :

```
Type TAB8x8 is array (0 to 7) of std_logic_vector(7 downto 0);
tableau : out TAB8x8;
```

Les sous-types. Exemples :

```
subtype OCTET is bit_vector(7 downto 0);
BUS : inout OCTET;
```

- Lignes 7 à 16 : la partie déclarative de l'architecture (entre les mots clés architecture et begin) est destinée à la déclaration des objets internes (les E/S sont déclarées dans l'entité) utilisés dans cette architecture. Ces objets sont généralement des signaux, constantes, variables ou alias.

✓ Les signaux. Ils représentent les fils d'interconnexion sur la carte. Exemple :

```
signal BUS : std_logic_vector(15 downto 0);
```

- ✓ Les constantes. Une constante peut être assimilée à un signal interne ayant une valeur fixe. Exemples :

```
constant ZERO : bit_vector(7 downto 0);
constant HIZ : std_logic_vector(15 downto 0);
```

- ✓ Les variables. Une variable est un objet capable de retenir une valeur pendant une durée limitée. Ce n'est pas une liaison physique, mais un objet abstrait qui doit être interprété par le synthétiseur. Elle ne doit être utilisée qu'à l'intérieur d'un process. Exemple :

```
variable TEMP : integer;
```

- ✓ Les alias. Ils permettent de nommer un objet de différentes manières. Exemples :

```
signal DBUS : bit_vector(15 downto 0);
alias OCTET0 : bit_vector(7 downto 0) is DBUS(7 downto 0);
```

La ligne 8 définit donc un signal temporaire `tmp` de type `std_logic`.

Le corps de l'architecture (lignes 9 à 16) contient la description du design. VHDL reconnaît les opérateurs logiques suivant : and, nand, or, nor, xor, xnor et not. Leur signification est évidente. Il faut juste faire attention à leur associativité. Il est possible d'écrire :

`X <= A and B and C;` puisque le and est associatif (A . B . C = A . (B . C))

L'opérateur d'affectation (ou d'assignation) '<=' permet d'affecter une valeur à un signal. Par contre, on n'a pas le droit d'écrire :

`X <= A nor B nor C;` puisque le nor n'est pas associatif ($\overline{A+B+C} \neq \overline{A+\overline{(B+C)}}$)

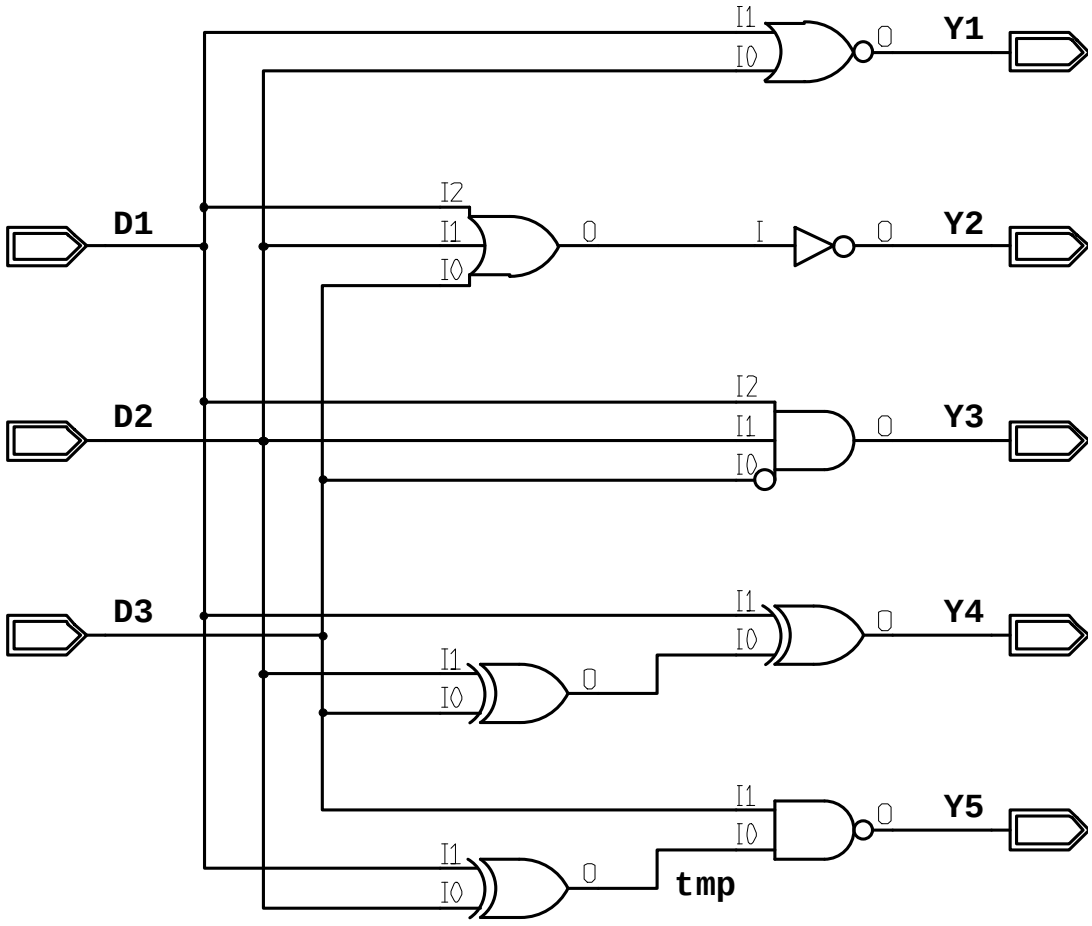
Il faut obligatoirement mettre des parenthèses pour indiquer ce que l'on souhaite réaliser.

```
X <= A nor (B nor C);
```

Pour faire un vrai nor 3 entrées, il suffit d'écrire (car le or est associatif) :

```
X <= not (A or B or C);
```

Compte tenu de ce qui précède, on voit que le circuit GATE réalise les fonctions logiques suivantes :



2.4.3 Multiplexeurs (mémorisation implicite)

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity MUX is
4.   port(Sel : in std_logic_vector(1 downto 0) ;
5.        A, B, C, D : in  std_logic;
6.        Y1, Y2, Y3, Y4 : out std_logic);
7. end MUX;

8. architecture RTL of MUX is
9.   signal tmp : std_logic ;
10.begin

11.  Y1 <= A when Sel(0) = '0' else B;
```

```
12.  p0 : process (A, B, Sel)
13.  begin

14.    if (Sel(0) = '0') then
15.      Y2 <= A;
16.    else
17.      Y2 <= B;
18.    end if;

19.    if (Sel(1) = '1') then
20.      Y3 <= A;
21.    end if;

22.  end process;

23.  p1 : process (A, B, C, D, Sel)
24.  begin
25.    case Sel is
26.      when "00" => Y4 <= A;
27.      when "01" => Y4 <= B;
28.      when "10" => Y4 <= C;
29.      when "11" => Y4 <= D;
30.      when others => Y4 <= A;
31.    end case;
32.  end process;
33.end RTL;
```

Les constantes littérales désignent des valeurs numériques, des caractères ou des valeurs binaires. Les constantes numériques sont de type entier ou flottant et peuvent exprimer une valeur dans une base comprise entre 2 et 16. La lettre E (ou e), pour exposant, est acceptée pour les entiers et pour les flottants. La base par défaut est 10. Exemples de constantes littérales:

14	0	1E4	123_456	*Entiers en base 10*
14.0	0.0	1.0e4	123.456	*Flottants en base 10*
2#1010#	16#A#	8#12#	10	*Le nombre 10*
'A'	'B'	'5'	';'	*Des caractères*
"coucou"	"01234"	"un essai"	"- +"	*Des chaines de caractères*
'1'	'0'			*Des bits seuls*
"1010"	B"1010"			*Une chaine binaire en base 2*
X"ff"	O"56"			*Une chaine binaire en base16 et 8*

- Lignes 8 à 33. Les opérateurs relationnels suivants sont présents dans les instructions conditionnelles de VHDL telles que « `when…else` », « `if…then…else` » ou « `case` » : `=`, `/=`, `<`, `<=`, `>`, `>=`. Ils retournent un résultat booléen (vrai/faux). Exemple :

```
if (a > b) then …
```

Faites attention à ne pas confondre l'affectation `X <= Y;` avec l'opérateur relationnel inférieur ou égal. Les opérateurs relationnels peuvent être combinés aux opérateurs logiques de la manière suivante :

```
if (a <= b and b=c) then …
```

L'opérateur de concaténation & permet de juxtaposer deux objets de type std_logic ou std_logic_vector. Exemples :

```
signal octet1, octet2 : std_logic_vector(7 downto 0) ;
signal mot1, mot2 : std_logic_vector(15 downto 0) ;

mot1 <= octet1&octet2 ;
mot2 <= "100"&octet1(3 downto 0)&octet2&'1' ;
```

Il permet de simplifier l'écriture de :

```
if (A = '1' and B = '0') then …
```

en écrivant :

```
if (A&B = "10") then …
```

- La ligne 11 décrit un multiplexeur 2 entrées avec l'assignation conditionnelle `when`.

```
Y1 <= A when Sel(0) = '0' else B;  -- Sel(0) est le bit 0 de Sel
```

En effet, la phrase « Y1 prend la valeur de A si Sel(0) est égal à 0 sinon (Y1 prend la valeur de) B » décrit bien un multiplexeur deux entrées A et B vers une sortie Y1 avec une entrée de sélection reliée au bit de poids faible de Sel. La condition testée doit être booléenne, ce qui est bien le cas puisque les opérateurs relationnels fournissent un résultat booléen.

- Lignes 12 à 32 : fonctionnement concurrent et séquentiel. En électronique, les composants fonctionnent simultanément (fonctionnement parallèle ou concurrent) alors qu'en programmation traditionnelle, les instructions s'exécutent les unes à la suite des autres de façon séquentielle. Les deux modes de fonctionnement coexistent dans VHDL suivant que le code se trouve hors d'un processus (fonctionnement concurrent) ou dans un processus (fonctionnement séquentiel**, avec des restrictions**). Prenons par exemple :

```
A <= B;
B <= C;
```

En programmation traditionnelle, la séquence signifierait « A prend la valeur de B, puis B prend la valeur de C ». A la fin du programme, A et B ont des valeurs différentes. Si on change l'ordre des instructions, la signification change. C'est le fonctionnement séquentiel. En VHDL, il faut comprendre : à tout moment, A prend la valeur de B et à tout moment, B prend la valeur de C. En clair A, B et C ont tout le temps la même valeur quel que soit l'ordre des instructions. C'est le fonctionnement concurrent.

Voyons maintenant ce qu'est un processus. C'est un groupe délimité d'instructions doté de trois caractéristiques :

1. Le processus s'exécute à chaque changement d'état d'un des signaux auxquels il est déclaré sensible.
2. Les instructions dans le processus s'exécutent séquentiellement.
3. Les modifications apportées aux valeurs des signaux par les instructions d'affectation prennent effet **simultanément** à la fin du processus. C'est une restriction très importante et il faut voir le §2.4.4 pour saisir pleinement ce que signifie séquentiel en VHDL.

La structure d'un processus est la suivante :

```
Nom_de_processus : process(liste_de_sensibilité)
  -- déclaration des variables locales du processus
begin
  -- corps du processus
end process Nom_de_processus ;
```

Ligne 12 : le processus p0 se déclenche sur chaque changement d'état des signaux A, B et Sel.

Ligne 13 à 18 : on décrit à nouveau un multiplexeur 2 entrées avec l'assignation conditionnelle « `if` … `then` … `else` … ». En effet, la phrase « si Sel(0) est égal à 0, Y2 prend la valeur de A sinon Y2 prend la valeur de B » décrit bien un multiplexeur deux entrées A et B vers une sortie Y2 avec une entrée de sélection reliée au bit de poids faible de Sel.

Ligne 23 : le processus p1 se déclenche sur chaque changement d'état des signaux A, B, C, D et Sel. Lignes 25 à 31 : on décrit un multiplexeur 4 entrées avec l'assignation sélective `case`.

Ligne 25 : le sélecteur est le bus Sel, de largeur 2 bits.
Ligne 26 : Quand Sel vaut 00, Y4 prend la valeur de A,
Ligne 27 : Quand Sel vaut 01, Y4 prend la valeur de B,
Ligne 28 : Quand Sel vaut 10, Y4 prend la valeur de C,
Ligne 29 : Quand Sel vaut 11, Y4 prend la valeur de D,
Ligne 30 : pour toutes les autres valeurs de Sel (n'oubliez pas que Sel est un std_logic_vector dont chaque bit peut prendre 9 états), Y4 prend la valeur de A. C'est la valeur par défaut qui est **obligatoire**. On verra un peu plus loin que cela permet d'éviter d'inférer (de générer) un latch.

On a bien décrit un multiplexeur 4 entrées A, B, C, D, une sortie Y4 et deux bits de sélection Sel.

Nous allons maintenant lister les instructions séquentielles qui doivent se trouver obligatoirement à l'intérieur d'un processus et les instructions concurrentes qui doivent obligatoirement se trouver à l'extérieur d'un processus.

- Instructions en mode concurrent (**hors process**).

Assignation inconditionnelle. Forme générale : `signal <= expression;`.

```
A <= B and C; -- A prend la valeur du résultat de l'opération (B and C).
X <= '0';     -- X prend la valeur 0.
```

Assignation conditionnelle. Forme générale : `signal <= expression when condition else expression;`.

```
Y1 <= A when Sel(0) = '0' else B; -- Y1 prend la valeur de A si Sel(0) égal
                                  0 sinon (Y1 prend la valeur de) B.
```

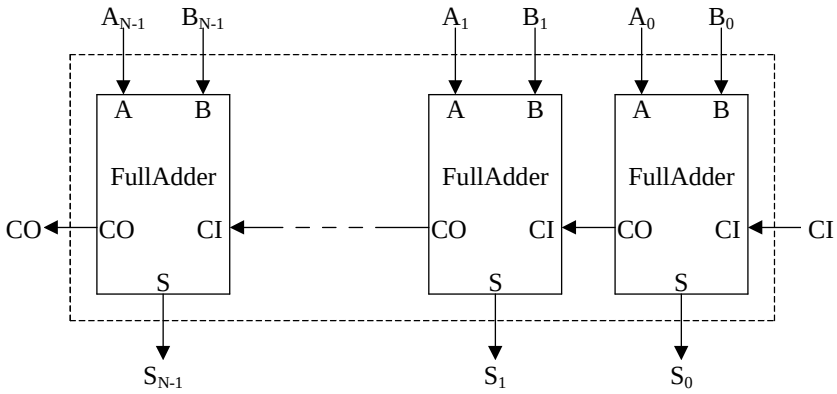
Assignation sélective. Forme générale : `with selecteur select signal <= {expression when valeur_selecteur,};`.

```
With ETAT select
X <= A when "00", -- X prend la valeur de A si le signal ETAT, utilisé comme
     B when "01", -- sélecteur, vaut 00, B si ETAT vaut 01, etc.
     C when "10",
     D when others; -- tous les choix sont couverts
```

Instanciation de composant. Forme générale : `nom_instance : nom_composant port map (liste_ entrée_sortie_composant);`.

```
U0 : XOR4 port map (A, B, C, D, S); -- on insère le composant XOR4 dans le design (à la manière d'un symbole dans un schéma). U0 est le nom d'instance. A, B, C, D sont les entrées de U0, S est sa sortie.
```

Instruction generate. Cette instruction permet de générer plusieurs répliques d'un composant ou d'une équation à l'aide d'un for. Exemple, un additionneur 8 bits construit à partir de 8 additionneurs 1 bit (composant FullAdder).




```
gen : for j in 7 downto 0 generate -- pour j allant de 7 à 0
  genlsb : if j = 0 generate -- génération de l'additionneur du bit de poids faible
    fa0 : FullAdder port map (A => A(0), B => B(0), CI =>
    CI, S => S(0), COUT =>  C(1));
  end generate;
  genmid : if (j>0) and (j<7) generate -- génération des autres additionneurs
    fax : FullAdder port map (A => A(j), B => B(j), CI =>
    C(j), S => S(j), COUT => C(j+1));
  end generate;
  genmsb : if j = 7 generate -- génération de l'additionneur du bit de poids fort
    fa7 : FullAdder port map (A => A(j), B => B(j), CI =>
    C(j), S => S(j), COUT => COUT);
  end generate;
end generate;
```

- Instructions en mode séquentiel (**dans un process**).

Assignation inconditionnelle de signal. Forme générale : `signal <= expression;`.

```
A <= B and C; -- A va prendre la valeur du résultat de l'opération (B and C) à
              la fin du processus.
X <= '0'; -- X prend la valeur 0 à la fin du processus.
```

Assignation inconditionnelle de variable. Forme générale : `var := expression;`.

```
V := '0'; -- X prend la valeur 0 immédiatement.
```

Assignation conditionnelle de signal ou de variable.

```
if (Sel(0) = '0') then  -- si Sel(0) est égal à 0,  Y2 prend la valeur de A
      Y2 <= A;
else  -- sinon Y2 prend la valeur de B
      Y2 <= B;
end if; -- branches incomplètes possibles
```

Assignation sélective.

```
case Sel is
 when "00" => Y4 <= A; -- si Sel vaut 00, Y4 prend la valeur de A
 when "01" => Y4 <= B; -- si Sel vaut 01, Y4 prend la valeur de B
 when "10" => Y4 <= C; -- si Sel vaut 10, Y4 prend la valeur de C
 when "11" => Y4 <= D; -- si Sel vaut 11, Y4 prend la valeur de D
 when others => Y4 <= A; -- pour toute autre valeur de Sel, Y4 égal A
end case; -- branches incomplètes possibles
```

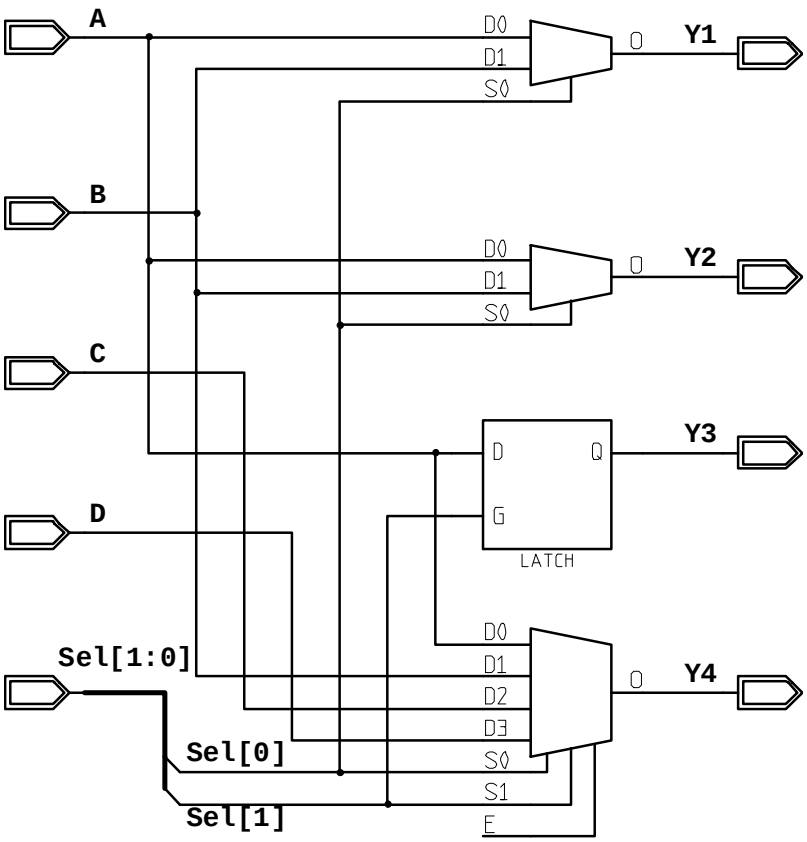
Boucles.

```
for i in 0 to 7 loop -- pour i allant de 0 à 7
 datari(i) <= "000"&datar(i); -- datari(i) = "000" concaténé avec datar(i)
end loop;
```

- Lignes 19 à 21 : en VHDL, les signaux ont une valeur courante et une valeur prochaine déterminée par l'opérateur d'assignation. Si, lors d'une instruction conditionnelle (concurrente ou séquentielle), un signal reçoit une assignation dans une branche alors il doit recevoir une assignation dans toutes les autres branches. Si tel n'est pas le cas, chaque absence d'assignation signifie que la prochaine valeur est identique à la valeur courante et le synthétiseur doit générer une logique de mémorisation (bascule D ou latch). Dans notre exemple, si Sel(1) vaut 1, Y3 prend la valeur de A et sinon, Y3 garde sa valeur courante. Le synthétiseur infère donc un latch au lieu d'un multiplexeur. C'est la **mémorisation implicite**.

Il faut noter ici un point important : la mémorisation implicite n'est possible qu'à l'intérieur d'un process. En effet, les instructions conditionnelles hors process (*when*, *with* … *select*) ne peuvent être incomplètes d'un point de vue syntaxique. Seuls le *if* et le *case* autorisent les branches incomplètes, donc obligatoirement dans un process.

Le circuit MUX réalise donc les fonctions logiques suivantes :



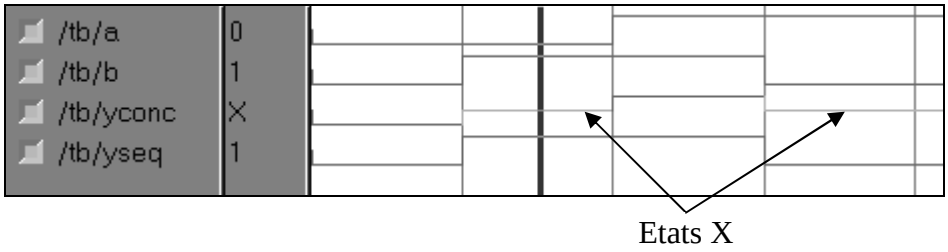
2.4.4 Assignation inconditionnelle de signal : séquentiel contre concurrent

Le design suivant tente d'éclaircir un point parfois un peu obscur, c'est-à-dire la différence de comportement des assignations inconditionnelles de signal entre le mode séquentiel (dans le process) et le mode concurrent (hors process).

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity conc_seq is
4. port( A, B  : in  std_logic;
5.       Yconc : out std_logic;
6.       Yseq  : out std_logic);
7. end conc_seq;
```

```
8. architecture a1 of conc_seq is
9. begin
10.  Yconc <= A;
11.  Yconc <= B;

12.  process(A, B) begin
13.    Yseq <= A;
14.    Yseq <= B;
15.  end process;
16.end a1 ;
```

En mode concurrent, les deux fils A et B sont reliés ensemble sur Yconc. Quand A et B sont dans un état différent (01 ou 10), Yconc passe à l'état indéterminé X puisqu'il y a conflit entre A et B. C'est le fonctionnement électronique traditionnel.



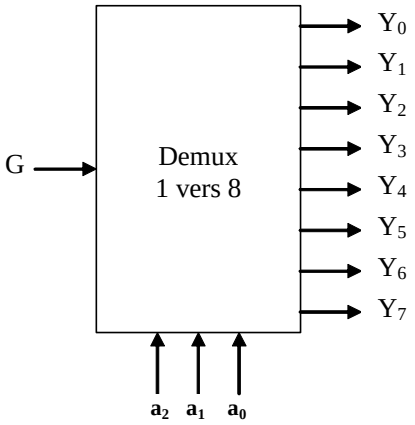
En mode séquentiel, A et B sont copiés dans Yseq. Mais il n'y a pas de conflit, car comme nous sommes en séquentiel, c'est la dernière assignation qui est prise en compte (Yseq <= B). Il faut comprendre qu'avec des signaux (ce n'est pas vrai pour des variables), toutes les assignations sont préparées dans le process, puis exécutées en même temps en sortant du process. S'il y a plusieurs assignations sur un même signal (Yseq dans notre cas), elles sont toutes exécutées, mais c'est la dernière qui est prise en compte. C'est en ce sens que l'exécution est séquentielle.

2.4.5 Démultiplexeur - décodeur

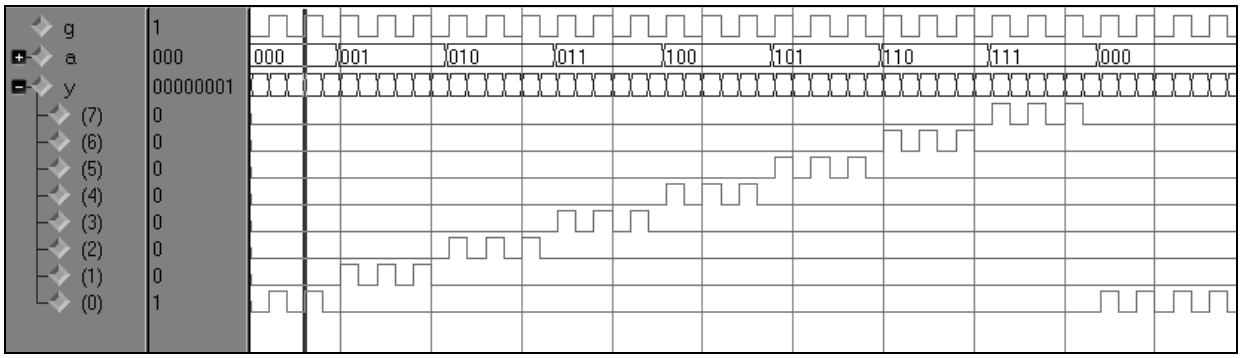
```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;

3.  entity Demux_1vers8 is
4.    port (G : in  std_logic;
5.          A : in  std_logic_vector(2 downto 0);
6.          Y : out std_logic_vector(7 downto 0) );
7.  end Demux_1vers8;

8.  architecture a1 of Demux_1vers8 is
9.  begin
10.   PROCESS (G, A) BEGIN
11.     if (G = '0') then
12.       Y <= (others => '0');
13.     else
```

```
14.       case A is
15.         when "000" => Y <= "00000001";
16.         when "001" => Y <= "00000010";
17.         when "010" => Y <= "00000100";
18.         when "011" => Y <= "00001000";
19.         when "100" => Y <= "00010000";
20.         when "101" => Y <= "00100000";
21.         when "110" => Y <= "01000000";
22.         when "111" => Y <= "10000000";
23.         when others => Y <= "00000001";
24.       end case;
25.     end if;
26.   END PROCESS;
27. end;
```

Le fonctionnement est le suivant : si G = 0, Y est mis à 0. L'expression `(others => '0')` est un agrégat. La même valeur 0 est affectée à tous les bits de Y. Si G = 1, Y(A) = 1. On obtient bien le démultiplexeur du §2.2.2 dans sa version 1 vers 8 (3 bits d'adresses). Il connecte G sur la sortie Y(A), A pouvant aller de 0 à 7.



Pour réaliser un décodeur, il suffit de mettre G à 1 en permanence au moment de l'utilisation du composant. Le chronogramme suivant montre le fonctionnement du circuit :



Le problème avec l'instruction `case`, c'est qu'il faut autant de lignes que de sorties pour décrire le fonctionnement du circuit. On peut réécrire le démultiplexeur d'une manière plus générale, c'est-à-dire indépendamment de sa taille, de la manière suivante :

```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;
3.  use IEEE.std_logic_unsigned.all;

4.  entity Demux_1vers8 is
5.    port (G : in  std_logic;
6.          A : in  std_logic_vector(2 downto 0);
7.          Y : out std_logic_vector(7 downto 0) );
8.  end Demux_1vers8;
9.  architecture a1 of Demux_1vers8 is
10. begin
11.   PROCESS (G, A) BEGIN
12.     Y <= (others => '0'); -- forcément en premier
13.     if (G = '1') then
14.       Y(CONV_INTEGER(A)) <= '1';
15.     end if;
16.   END PROCESS;
17. end;
```

Y est mis à 0 en premier. Si G = 0, Y reste à 0. Si G = 1, on copie 1 sur le bit A de Y. Comme nous sommes dans un process, il n'y a pas de conflit sur le bit A mis à 1 car c'est la dernière instruction qui est prise en compte (mode séquentiel). Le problème avec cette description est que l'indice du bit dans Y doit être de type entier alors que A est de type `std_logic_vector`. Il faut donc utiliser une fonction de conversion définie dans le package std_logic_unsigned, `CONV_INTEGER`. La conversion du `std_logic_vector` en entier est obligatoire. Le fonctionnement du nouveau démultiplexeur est strictement identique à celui utilisant l'instruction `case` (ainsi d'ailleurs que le nombre de portes utilisées). Il faut aussi noter que les instructions :

```
if (G = '1') then
  Y(CONV_INTEGER(A)) <= '1';
end if;
```

peuvent être remplacées par :

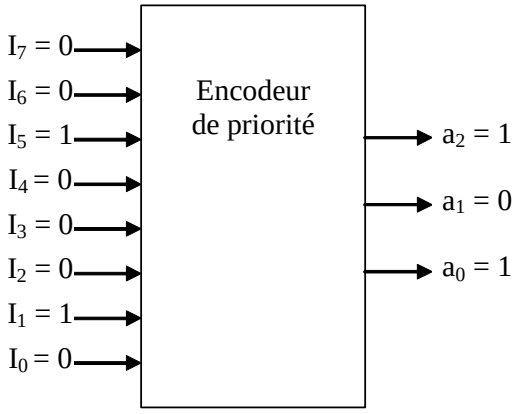
```
Y(CONV_INTEGER(A)) <= G;
```

2.4.6 Encodeur de priorité

```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;
3.  use IEEE.std_logic_arith.all;

4.  entity Priority_encoder_8vers3 is
5.    port (DataIn : in  std_logic_vector(7 downto 0);
6.          DataOut : out std_logic_vector(2 downto 0) );
7.  end Priority_encoder_8vers3;
```

```
8.  architecture a1 of Priority_encoder_8vers3 is
9.  begin
10.   PROCESS (DataIn) BEGIN
11.     DataOut <= (others => '0');
12.     Search : for I in DataIn'range loop
13.       if DataIn(I) = '1' then
14.         DataOut <= CONV_STD_LOGIC_VECTOR(I,3);
15.         exit Search;
16.       end if;
17.     end loop Search;
18.   END PROCESS;
19. end;
```

La description de cet encodeur de priorité est indépendante de sa taille, grâce à l'utilisation d'une boucle `for`. L'encodeur réalisé est un 3 vers 8 tel que nous l'avons décrit au §2.2.5. Il donne en sortie l'adresse de l'entrée de rang le plus élevé valant 1. Dans l'exemple suivant, I_5 est l'entrée à 1 de rang le plus élevé. On trouvera donc 5 sur les sorties a_i.



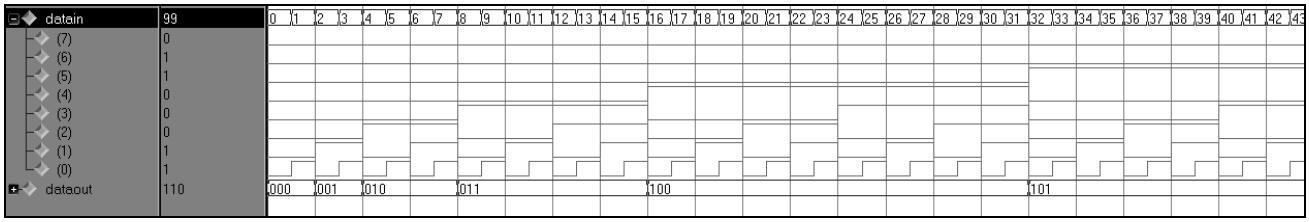
`DataOut` est mis à 0 en premier. Comme dans le démultiplexeur, on s'appuie sur le fonctionnement séquentiel du process pour réaliser une description compacte. Etudions la boucle `for`. Il existe en VHDL des informations complémentaires attachées à chaque objet déclaré : ce sont les attributs. Dans ce langage, on dit que les attributs décorent les objets auxquels ils sont rattachés. La référence à un attribut se fait par le nom de l'objet en préfixe suivi du nom de l'attribut en suffixe : `objet'attribut`. Il ne faut jamais oublier que VHDL est un langage orienté objet comme ADA dont il est dérivé. Il existe des attributs prédéfinis et des attributs définis par l'utilisateur. Nous les verrons au fur et à mesure de l'avancement du cours. L'attribut `range` définit la plage de variation de l'indice de l'objet. Dans notre exemple, `DataIn'range` est égal à `7 downto 0` à cause de la déclaration à la ligne 6 :

```
  port (DataIn : in  std_logic_vector(7 downto 0);
```

Donc la boucle `for` (de la ligne 12 à la ligne 17) va faire 8 tours (pour I = 7, 6, 5, 4, 3, 2, 1, 0). `Search` est l'étiquette de la boucle. Dans cette boucle, on teste la valeur du bit I de `DataIn`. S'il vaut 1, alors on copie la valeur de I dans `DataOut` puis on sort immédiatement de la boucle for (instruction `exit Search;`).

```
12.     Search : for I in DataIn'range loop
13.       if DataIn(I) = '1' then
14.         DataOut <= CONV_STD_LOGIC_VECTOR(I,3);
15.         exit Search;
16.       end if;
17.     end loop Search;
```

Le problème avec cette description est que I est de type entier alors que `DataOut` est de type `std_logic_vector`. Il faut donc utiliser une fonction de conversion définie dans le package std_logic_arith, `CONV_STD_LOGIC_VECTOR`. La conversion entier vers `std_logic_vector` est obligatoire. Le chronogramme suivant montre le fonctionnement du circuit :

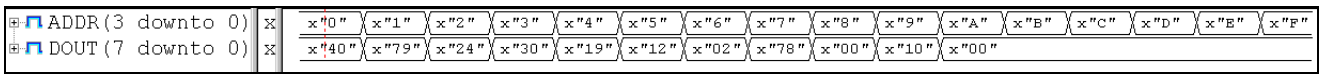


2.4.7 Mémoire ROM

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.std_logic_unsigned.all;

4. entity mem is
5.   port (addr : in std_logic_vector(3 downto 0);
6.         dout : out std_logic_vector(7 downto 0));
7. end mem;

8. architecture a1 of mem is
9.   TYPE mem_data IS ARRAY (0 TO 15) OF std_logic_vector(7 DOWNTO 0);
10.  constant data : mem_data := (
11.    ("01000000"),
12.    ("01111001"),
13.    ("00100100"),
14.    ("00110000"),
15.    ("00011001"),
16.    ("00010010"),
17.    ("00000010"),
18.    ("01111000"),
19.    ("00000000"),
20.    ("00010000"),
21.    ("00000000"),
22.    ("00000000"),
```

```
23.     ("00000000"),
24.     ("00000000"),
25.     ("00000000"),
26.     ("00000000"));
27.begin
28.  dout <= data(CONV_INTEGER(addr));
29.end;
```

Le type mem_data définit un tableau (ARRAY) de 16 cases de largeur 8 bits. La constante `data` qui représente le contenu de la mémoire est initialisée aux lignes 10 à 26. La ligne 29 définit le fonctionnement de la mémoire. Elle utilise une fonction de conversion définie dans le package std_logic_unsigned, `CONV_INTEGER`. En effet, l'index du tableau data doit être un nombre entier alors que le signal data est de type std_logic_vector. La conversion `std_logic_vector` vers entier est obligatoire. Le chronogramme suivant montre le fonctionnement de la mémoire :



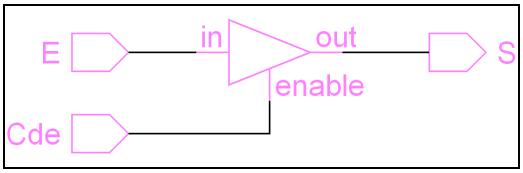
2.4.8 buffer bidirectionnel trois états

```
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;

3.  entity trois_etats is
4.  port(E : in std_logic;
5.       S : out std_logic;
6.     Cde : in std_logic);
7.  end trois_etats;

8.  architecture comporte of trois_etats is
9.  begin
10.   S <= E when Cde='1' else 'Z';
11. end comporte ;
```

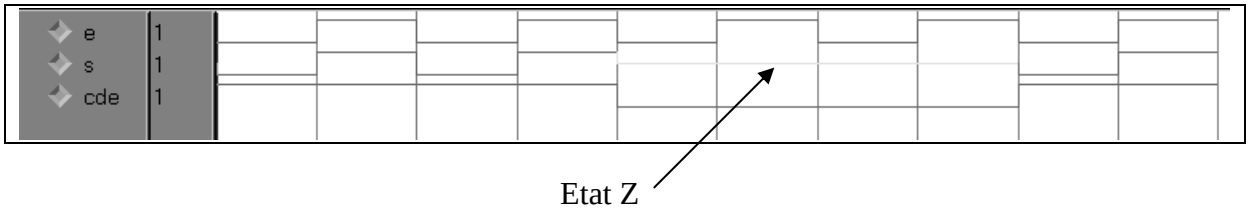
Un buffer trois états ressemble fortement à un multiplexeur dans sa description :

```
S <= E when Cde='1' else 'Z';
```

Il faut traduire par : S prend la valeur de E si Cde est égal à 1 sinon S prend la valeur Z, ce qui est bien la description d'un buffer trois état.



Le chronogramme suivant montre l'évolution des signaux :

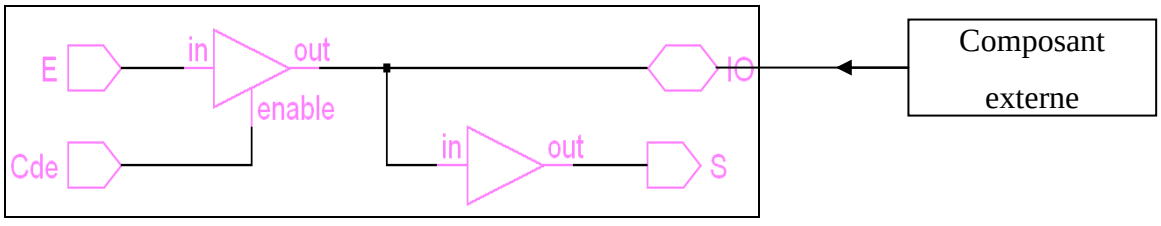


Un buffer bidirectionnel est un buffer trois états dont la sortie est réinjectée dans le design :

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity bidir is
4. port(E : in std_logic;
5.      S : out std_logic;
6.     IO : inout std_logic;
7.    Cde : in std_logic);
8. end bidir;

9. architecture comporte of bidir is
10.  begin
11.    IO <= E when Cde='1' else 'Z';
12.    S <= IO;
13.  end comporte ;
```

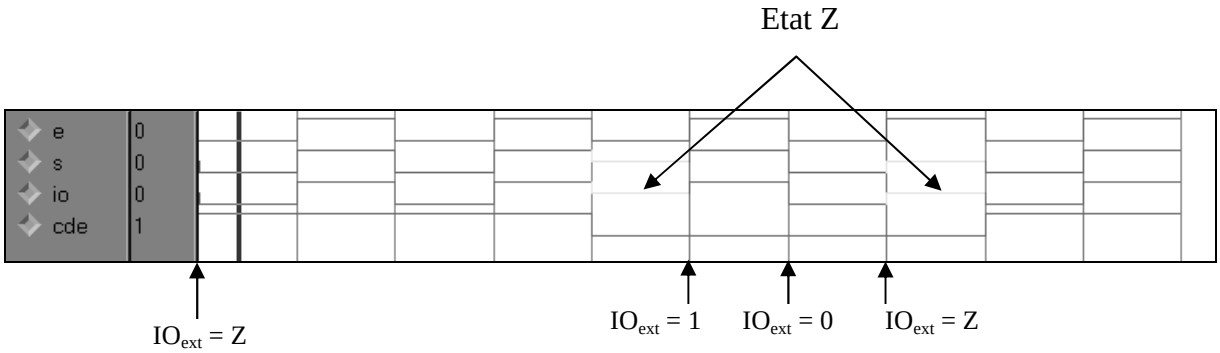
IO est maintenant bidirectionnelle (à la fois entrée et sortie). Si Cde = 1, IO est une sortie, si Cde = 0 alors le buffer 3 états passe à l'état haute impédance et on peut utiliser IO comme une entrée. Pour pouvoir écrire une valeur depuis l'extérieur sur IO, il faut que Cde soit égale à 0. Une broche bidirectionnelle est utilisée par exemple pour lire et écrire des données dans une mémoire de type RAM.

Pour simuler correctement un buffer bidirectionnel, il faut créer un composant externe afin de générer un état sur IO quand ce signal est une entrée.



bidir

Le chronogramme suivant montre l'évolution des signaux en signalant la valeur imposée sur IO depuis l'extérieur :



Lorsque le buffer trois états n'est pas à l'état Z (Cde = 1), il faut que le composant connecté sur IO depuis l'extérieur impose un état haute impédance sur cette broche. Sinon, il y aura un conflit de bus.

3 Représentation des nombres

3.1 Les codes

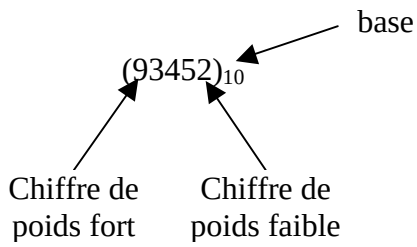
On regroupe souvent les bits par paquets afin de coder des nombres entiers, réels ou bien des caractères. Pour cela, un groupe de bits doit former un code. Il existe de très nombreux codes et nous allons voir maintenant les principaux.

3.1.1 Entiers naturels

3.1.1.1 Base d'un système de numération

Une base est constituée par l'ensemble des chiffres (ou caractères) différents qu'utilise le système de numération. Tout nombre peut se décomposer en fonction des puissances entières, positives ou négatives, de la base de numération. Il peut donc se mettre sous forme polynomiale. Prenons l'exemple de la base 10.

Dans la base décimale, un chiffre peut prendre 10 valeurs de 0 à 9. Un nombre est un polynôme en puissance de 10, 10^n avec n positif, nul ou négatif. Par exemple :



La forme polynomiale est :

$(93452)_{10} =$	$9.10^4 +$	$3.10^3 +$	$4.10^2 +$	$5.10^1 +$	2.10^0
rang	4	3	2	1	0

Un autre exemple avec un nombre réel :

$(23,64)_{10} =$	$2.10^1 +$	$3.10^0 +$	$6.10^{-1} +$	4.10^{-2}
rang	1	0	-1	-2

Le rang du chiffre d'un nombre représenté dans une base est l'exposant de la base associé à ce chiffre, dans la représentation polynomiale considérée.

Soit d'une manière générale, une base b contenant b caractères a_i tels que $a_i \in (0, 1, 2, \ldots, (b-1))$. La représentation polynomiale d'un nombre entier N est :

$$N = \sum_{i=0}^{n} a_i.b^i$$, n étant le rang du chiffre de poids fort.

Parmi toutes les bases possibles, 2 sont particulièrement intéressantes, les bases 2 et 16. La base 2 permet de traduire sous forme numérique l'état des variables booléennes (vrai = 1 / faux = 0 en logique positive). Les a_i peuvent donc valoir (0, 1) et la représentation polynomiale est :

$(10110)_2 =$	$1.2^4 +$	$0.2^3 +$	$1.2^2 +$	$1.2^1 +$	0.2^0
rang	4	3	2	1	0

Le bit le plus à gauche est le bit de poids fort (**MSB** : Most Significant Bit), le bit le plus à droite est le bit de poids faible (**LSB** : Least Significant Bit). Un chiffre en base 16 peut prendre 16 valeurs allant de 0 à 15. Il peut être codé avec 4 bits. Comme les chiffres s'arrêtent à 9 en décimal, on utilise les lettres a, b, c, d ,e et f pour représenter les derniers états.

décimal	binaire	hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

La représentation polynomiale en base hexadécimale (base 16) est :

$(a0c25)_{16}$ =	$10.16^4 +$	$0.16^3 +$	$12.16^2 +$	$2.16^1 +$	5.16^0
rang	4	3	2	1	0

3.1.1.2 Changement de base

Une question importante est le passage d'une base à l'autre : le changement de base. Soit $(N)_{10}$, combien vaut $(N)_2$ ou $(N)_{16}$.

La conversion vers la base 10 est la plus simple. Il suffit de calculer la valeur du polynôme. En reprenant les exemples précédents, il est évident que (au moins avec une calculatrice) :

$$(10110)_2 = 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0 = (22)_{10}$$

$$(a0c25)_{16} = 10.16^4 + 0.16^3 + 12.16^2 + 2.16^1 + 5.16^0 = (658469)_{10}$$

Remarque : il existe en binaire un regroupement très utile, l'**octet** (ou byte) qui est un groupe de 8 bits. Par exemple l'octet :

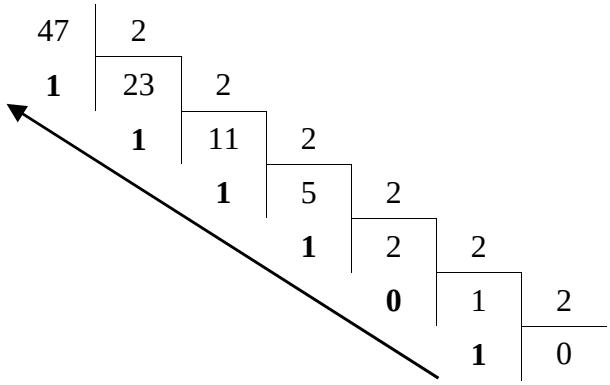
$$(10000001)_2 = 1*2^7 + 1*2^0 = (129)_{10}$$

La valeur d'un octet est comprise entre 0 et $(255)_{10}$. Pour raccourcir l'écriture, on utilise souvent la notation hexadécimale (base 16) en regroupant les bits par paquets de 4. Par exemple l'octet :

$$(10000001)_2 = (81)_{16}$$

Le changement de base décimal $\rightarrow$ binaire est moins évident. Il faut diviser le nombre à convertir par 2. On conserve le reste (qui vaut 0 ou 1) et on répète la division sur le quotient,

jusqu'à ce que le quotient vaille 0. On écrit alors tous les restes à la suite, le premier reste obtenu étant le poids faible. Exemple :



On obtient :

$$(47)_{10} = (101111)_2$$

Le passage d'hexadécimal vers binaire et de binaire vers hexadécimal est plus simple. Dans le premier cas, il suffit de remplacer chaque chiffre hexadécimal par le groupe de 4 bits correspondant (voir tableau de conversion). Exemple :

$$(ab85)_{16} = (1010\ \ 1011\ \ 1000\ \ 0101)_2$$

Dans le deuxième cas, il suffit de regrouper les bits par paquet de 4 en partant du poids faible. Il faudra éventuellement rajouter des bits à 0 à gauche pour terminer le groupe de 4 bits de poids fort. Exemple :

$$(101001000010101111)_2 = (\mathbf{00}10\ \ 1001\ \ 0000\ \ 1010\ \ 1111)_2 = (290af)_{16}$$

Pour passer d'hexadécimal en décimal (et vice versa), vous pouvez passer par l'intermédiaire du binaire ou faire le calcul directement par la méthode des divisions successives.

3.1.2 Entiers signés

La méthode naturelle pour coder un nombre avec signe est d'utiliser la méthode traditionnelle de la base 10 : on indique le signe suivi de la valeur absolue du nombre. Ce type de code est connu sous le nom de « signe-valeur absolue ». En binaire, cela signifie que pour un nombre codé sur N bits, le bit de poids fort sert à coder le signe (0 = +, 1 = -) et les N-1 bits restant codent la valeur absolue du nombre. Par exemple :

$$(-16)_{10} = (1\ 10000)_2$$

Dans un tel code, la plage des valeurs que l'on peut coder (la dynamique) est :

$2^{N-1}-1$	0 11…11
…	…
2	0 00…10
1	0 00…01
0	**0 00…00**
-0	**1 00…00**
-1	1 00…01
-2	1 00…10
…	…
$-(2^{N-1}-1)$	1 11…11

vous noterez qu'il y a deux valeurs 0

Cette méthode simple pour l'être humain est plutôt compliquée à utiliser dans un ordinateur (ou un circuit numérique) car pour additionner ou soustraire deux nombres, il faut commencer par déterminer quel sera le signe du résultat ce qui oblige à commencer tout calcul par une comparaison qui fait intervenir les signes et les valeurs absolues des deux nombres. C'est la raison pour laquelle on utilisera plutôt le code complément à 2.

Le code complément à 2 est le code utilisé pour représenter les nombres entiers dans un ordinateur. Il n'est pas très parlant pour un être humain, mais il présente l'intérêt majeur de permettre une arithmétique simple (notamment, il n'y a plus de soustraction). La construction du code sur n bits découle directement de la définition modulo 2^n des nombres binaires. Etant donné un nombre A :

- Si $A \geq 0$, le code de A est l'écriture en binaire naturel de A, éventuellement complété à gauche par des 0. Par exemple, $A = (23)_{10}$, codé sur 8 bits s'écrit : $(00010111)_{ca2}$.
- Si $A < 0$, le code de A est l'écriture en binaire naturel de $2^n - |A|$ ou ce qui revient au même $2^n + A$. Par exemple, $A = (-23)_{10}$, codé sur 8 bits s'écrit : $(11101001)_{ca2}$ qui est la représentation en binaire naturel de $(256)_{10} - (23)_{10} = (233)_{10}$.

On remarquera que le bit de poids fort indique le signe (0 = +, 1 = -). Le calcul de l'opposé d'un nombre codé sur n bits est toujours : $-A = 2^n - A$ modulo 2^n. Par exemple $-(-23) = 256 + 23$ modulo $256 = 23$. Il existe une astuce de calcul pour obtenir l'expression binaire de l'inverse d'un nombre dont on connaît le code :

- On remarque que $2^n - A = 2^n - 1 - A + 1$.
- On sait que $2^n - 1$ est le nombre dont tous les chiffres valent 1.
- On en déduit que $2^n - 1 - A$ est le nombre que l'on obtient en remplaçant dans le code de A les 1 par des 0 et réciproquement. Ce nouveau nombre s'appelle le complément à 1 de A et se note A_{ca1} ou $\overline{A}$. On en déduit finalement que :

$$-A = A_{ca1} + 1$$

Par exemple :

$$(23)_{10} = (00010111)_2 = (11101000)_{ca1}$$
$$-23 = (11101000)_{ca1} + 1 = (11101001)_{ca2}$$

La lecture en décimal d'un nombre binaire codé en CA2 est simple si on garde en mémoire le poids de chaque rang (avec le rang de poids fort négatif) :

$$(10110)_{CA2} = 1.\mathbf{-2^4} + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0 = (-10)_{10}$$

Il faut absolument connaitre à l'avance le nombre de bits utilisés pour coder la valeur. Dans le cas précédent, le nombre codé sur 5 bits est négatif. Codé sur 8 bits (avec les trois 0 implicites à gauche du nombre que l'on ne place généralement jamais en binaire naturel), il est positif.

$$(00010110)_{CA2} = 0.2^7 + 0.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0 = (22)_{10}$$

Dans ce code, la dynamique est :

$2^{N-1}-1$	0 11…11
…	…
2	0 00…10
1	0 00…01
0	0 00…00
-1	1 11…11
-2	1 11…10
…	…
$-(2^{N-1})$	1 00…00

Un des avantages de ce code est que la soustraction n'existe plus. Au lieu de soustraire un nombre, il n'y a qu'à additionner l'inverse de ce nombre. Par exemple, au lieu de calculer 50 – 23, on va pouvoir calculer 50 + (-23). Cela paraît bête, mais cela élimine la nécessité d'utiliser un soustracteur binaire. On n'utilise en conception logique que des additionneurs.

Un autre avantage du code complément à 2 est que si on effectue des additions successives, tant que le résultat final est dans la dynamique autorisée (pas de débordement du résultat final), on peut ignorer les débordements éventuels des résultats intermédiaires de calcul. Par exemple, on code avec 3 bits (dynamique de -4 à +3) et on additionne 2 + 2 - 3 ce qui doit donner 1 en base 10. Lors du premier calcul, 2 + 2 = 4, il y a un débordement mais le résultat final est quand même bon. En effet, $(010)_{CA2} + (010)_{CA2} = (100)_{CA2}$ qui vaut -4 (donc débordement) mais $(100)_{CA2} + (101)_{CA2} = (001)_{CA2}$ ce qui donne bien 1. Cela paraît magique, mais en fait, c'est une conséquence directe de l'arithmétique modulo 2^n.

Il y a malgré tout des inconvénients au code complément à 2 en ce qui concerne les comparaisons, les multiplications et les divisions, mais ses avantages l'emportent largement sur ses inconvénients.

3.1.3 Nombres réels

3.1.3.1 Virgule flottante

Les nombres flottants (floating point numbers) permettent de représenter, de **manière approchée**, une partie des nombres réels. La valeur d'un réel ne peut être ni trop grande, ni trop précise. Cela dépend du nombre de bits utilisés (en général 32 ou 64 bits). C'est un

domaine très complexe et il existe une norme, IEEE-754, pour que tout le monde obtienne les mêmes résultats (faux bien sûr). Le codage en binaire est de la forme :

s : signe	**e** : exposant (signé)	**zzzz** : partie fractionnaire de la mantisse (non signé)

C'est-à-dire que si X est un nombre flottant :

$$X = (-1)^{\mathbf{s}} \times 2^{\mathbf{e}} \times 1.\mathbf{zzzz}\ldots$$

où s est le signe de X, e est l'exposant signé et zzzz… est la partie fractionnaire de la mantisse. C'est, en binaire, la notation scientifique traditionnelle. Voici quelques exemples de formats usuels :

nombre de bits	format binaire (s + e + zzzz)	valeur max	précision max
32	1 + 8 + 23	$2^{128} \approx 10^{+38}$	$2^{-23} \approx 10^{-7}$
64	1 + 11 + 52	$2^{1024} \approx 10^{+308}$	$2^{-52} \approx 10^{-15}$
80	1 + 15 + 64	$2^{16384} \approx 10^{+4932}$	$2^{-64} \approx 10^{-19}$

Le calcul en virgule flottante est très coûteux en termes de portes et de consommation (mais pas en fréquence de fonctionnement). Il est donc rarement utilisé en électronique numérique (comme en DSP d'ailleurs). On utilise plutôt le calcul en virgule fixe.

3.1.3.2 virgule fixe

Le calcul avec des nombres en virgule fixe (fixed point numbers) revient à faire tous les calculs avec des entiers en recadrant les résultats à l'aide d'une virgule fictive. Transposer un algorithme utilisant des flottants et le passer en virgule fixe est une tâche longue et délicate dans le cas général.

3.1.4 Des codes particuliers

3.1.4.1 Le code BCD

Le code BCD (Binary Coded Decimal) permet de coder un nombre décimal en binaire. A chaque chiffre décimal, on fait correspondre un groupe de 4 bits comme pour la base hexadécimale. Mais ici, il n'y a pas de valeur supérieure à 9. Le code est le suivant :

décimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

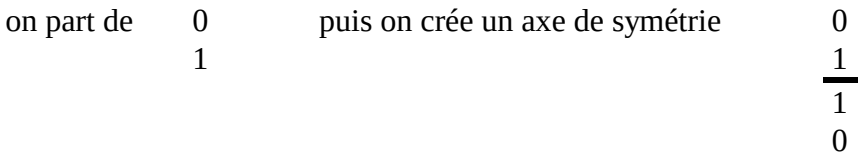
On voit que c'est un code incomplet car il n'utilise que 10 combinaisons sur les 16 possibles. Ce code est utile quand on veut traiter des nombres décimaux en électronique numérique (par exemple pour dialoguer avec un être humain ou pour compter directement en décimal).

3.1.4.2 Le code Gray

Le code Gray est un code adjacent (où un seul bit change quand on passe d'une valeur à la valeur suivante). On l'appelle aussi le code binaire réfléchi. On l'utilise dans les tableaux de Karnaugh mais aussi en conception numérique. Voici un exemple de code Gray sur 3 bits :

gray		
g_2	g_1	g_0
0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

La construction peut se faire graphiquement par itération ou bien à partir du code en binaire naturel. Graphiquement :



On ajoute le bit de poids fort en binaire naturel.

0 0
0 1
1 1
1 0

Pour prolonger le code (passer sur 3 bits), on récrée un nouvel axe de symétrie sur les deux bits faibles, puis on ajoute un bit supplémentaire en binaire naturel :

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

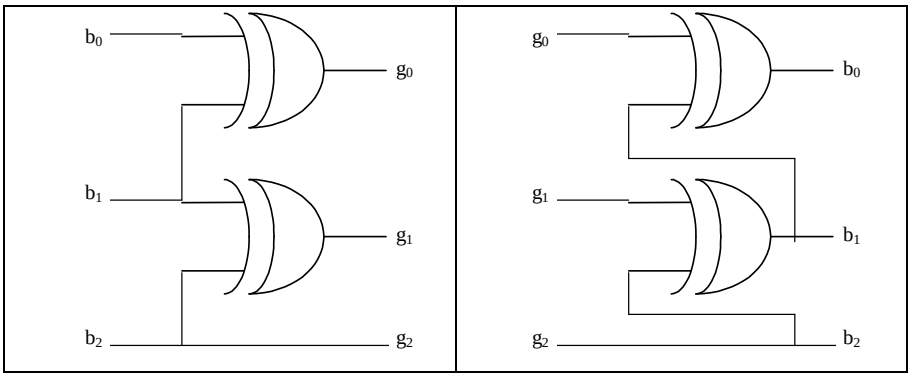
Et on recommence si on veut ajouter un bit supplémentaire. En comparant le code binaire naturel et le code Gray sur 3 bits :

binaire		
b_2	b_1	b_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Gray		
g_2	g_1	g_0
0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

On en déduit les équations et les schémas suivants :

De binaire naturel vers code Gray	De code Gray vers binaire naturel
$g_0 = b_0 \oplus b_1$	$b_0 = g_0 \oplus b_1$
$g_1 = b_1 \oplus b_2$	$b_1 = g_1 \oplus b_2$
$g_2 = b_2$	$b_2 = g_2$



Sur un nombre quelconque de bits, on obtient les équations :

De binaire naturel vers code Gray	De code Gray vers binaire naturel
$g_i = b_i$ XOR b_{i+1} pour $0 < i < N-2$ $g_{N-1} = b_{N-1}$	$b_i = g_i$ XOR b_{i+1} pour $0 < i < N-2$ $g_{N-1} = b_{N-1}$

Le transcodage binaire-Gray est utilisé par exemple pour réaliser un compteur. On compte en binaire, puis on transcode en Gray. Le transcodage Gray-binaire est rarement utilisé à cause du cumul des temps de propagation dans le montage qui le rend beaucoup plus lent que le transcodage binaire–Gray.

3.1.4.3 Le code Johnson

Le code Johnson est un autre exemple de code adjacent que l'on peut utiliser à la place du code Gray. En voici un exemple sur 3 bits :

Johnson		
b_2	b_1	b_0
0	0	0
0	0	1
0	1	1
1	1	1
1	1	0
1	0	0

On voit que c'est un code incomplet car il n'utilise que 6 combinaisons sur les 8 possibles.

3.1.4.4 Le code 1 parmi N

Le code 1 parmi N est très utilisé en conception numérique, notamment pour encoder les machines à états finis. Les Américains l'appellent le codage « one hot ». En voici un exemple sur 3 bits :

1 parmi 3		
b_2	b_1	b_0
0	0	1
0	1	0
1	0	0

On voit que c'est un code incomplet car il n'utilise que 3 combinaisons sur les 8 possibles.

3.1.4.5 Le code ASCII

Il n'y a pas que les nombres qui doivent être codé en binaire, mais aussi les caractères comme les lettres de l'alphabet, les signes de ponctuation, … Le code le plus connu et le plus utilisé est le code ASCII (American Standard Code for Information Interchange). Le code ASCII est un jeu normalisé de 128 caractères codés sur 7 bits, devenu un standard quasi universel. Il comporte tous les caractères alphanumériques non accentués et est lisible par pratiquement n'importe quelle machine. Ce sont les 8 premières lignes du tableau suivant.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
1	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del
8	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	np	cr	so	si
9	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
a		¡	ć	£	ď	ě	ę	ğ	Í	©	ł			ŋ	ő	
b	°	±	²	³	ť	μ	¶	ů	ÿ	¹	ž	ż	ij	¡	¿	¿
c	À	Á	Â	Ã	Ä	Å	Æ	Ç	É	È	Ê	Ë	Ì	Í	Î	Ï
d	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Œ	Ø	Ù	Ú	Û	Ü	Ý	Þ	β
e	à	á	â	ã	ä	ȧ	æ	ç	è	é	ê	ë	ì	í	î	ï
f	ǒ	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Les 32 premiers codes sont utilisés comme caractères de contrôle pour représenter, par exemple, une fin de ligne ou une tabulation. Le code ASCII ne contient pas de caractères

accentués et il a été complété par le code ISO-8859-1 (ou Latin 1). Les 128 premiers caractères correspondent au code ASCII, les 128 suivants aux caractères accentués et caractères spéciaux (voir les 8 dernières lignes du tableau).

Unicode est un jeu de caractères codé sur 16 bits (contre 7 ou 8 bits pour les standards actuels) qui permet le codage des caractères utilisés par toutes les langues du monde au sein d'une table unique. 16 bits permettent de coder 65 536 (2^{16}) caractères différents ce qui couvre largement les besoins en la matière. Les 256 premiers caractères d'Unicode correspondent au jeu ISO Latin 1.

3.1.4.6 Les codes détecteurs et/ou correcteurs d'erreurs

Si on craint que des erreurs se produisent dans un mot binaire (au cours d'une transmission par exemple), on rajoute au code des bits supplémentaires pour que le récepteur puisse détecter les erreurs et éventuellement les corriger. Par exemple, il y a un code détecteur et correcteur d'erreurs dans le train binaire enregistré sur un compact disque audio (code Reed-Solomon). Ces codes redondants sont un domaine d'étude complet de la théorie de l'information.

3.2 Les circuits arithmétiques

Les circuits arithmétiques effectuent des opérations binaires (signées ou non signées) telles que l'addition, la multiplication, la comparaison ou combinent ces opérateurs de base au sein d'une UAL (Unité Arithmétique et Logique).

3.2.1 L'additionneur/soustracteur

Prenons l'exemple d'une addition en binaire naturel sur 4 bits :

A	$(5)_{10}$	$(0101)_2$
B	$(3)_{10}$	$(0011)_2$
S	$(8)_{10}$	$(1000)_2$

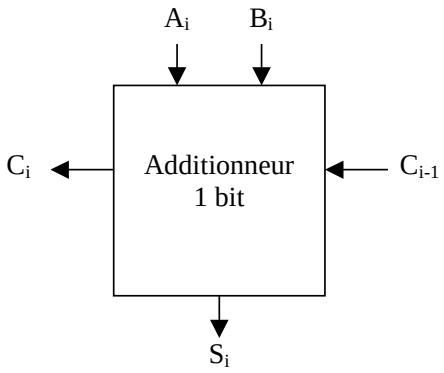
1ère colonne : 1 + 1 = 0 avec une retenue sortante à 1 ($(2)_{10} = (10)_2$).

2ème colonne : la retenue rentrante vaut 1. 1 + 0 + 1 = 0 avec une retenue sortante à 1.

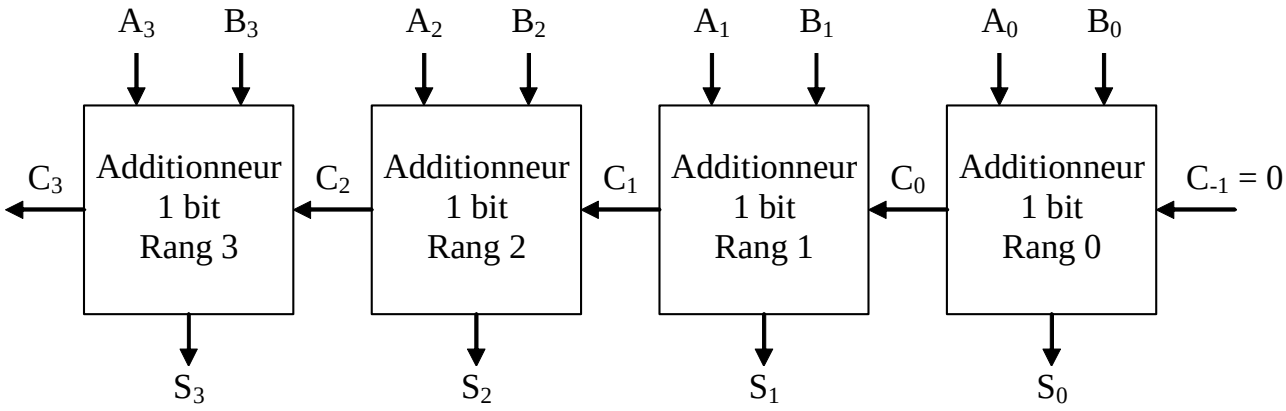
3ème colonne : la retenue rentrante vaut 1. 1 + 1 + 0 = 0 avec une retenue sortante à 1.

4ème colonne : la retenue rentrante vaut 1. 1 + 0 + 0 = 1.

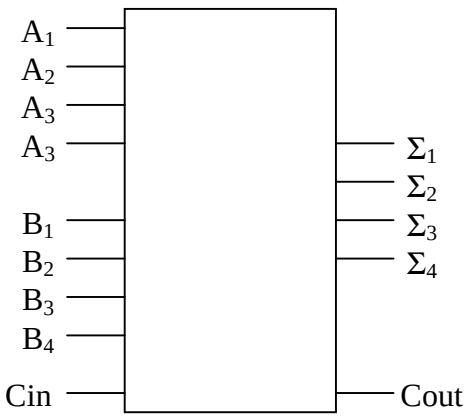
Sur les deux colonnes du milieu, il y a une retenue entrante et une retenue sortante. On peut donc définir les entrées-sorties d'un additionneur traitant le ième bit du calcul :



L'entrée C_{i-1} est la retenue entrante (Carry en anglais) et la sortie C_i est la retenue sortante. L'additionneur complet est construit de la manière suivante :



Vu de l'extérieur, un additionneur (ou un soustracteur, en CA2 c'est la même chose) se présente donc sous la forme suivante.



Ils effectuent l'addition (ou la soustraction) de deux nombres codés sur N bits. Le résultat est codé sur N bits avec une sortie supplémentaire pour la retenue. Il y a aussi une entrée de retenue pour pouvoir mettre en cascade les circuits. Sur 4 bits, L'opération réalisée sous forme décimale est (le + est une addition, le . est une multiplication) :

$$Cin + (A_1+B_1)+2.(A_2+B_2)+ 4.(A_3+B_3)+8.(A_4+B_4) = \Sigma_1+2.\Sigma_2+4.\Sigma_3+8.\Sigma_4+16.Cout$$

3.2.2 Débordement de calcul

Le résultat d'une addition ou d'une soustraction est supposé tenir dans la plage de la dynamique choisie pour représenter les nombres. Si N bits ont été choisis en CA2, alors le résultat doit tenir dans la plage -2^{N-1} : $2^{N-1}-1$. Si le résultat de l'opération ne tient pas dans cette plage, alors on dit qu'il y a débordement de calcul (« arithmetic overflow »). Pour s'assurer du bon fonctionnement d'un circuit arithmétique, il faut être capable de détecter ce débordement. Prenons l'exemple des 4 opérations possibles avec les valeurs 7 et 2 en CA2 codé sur 4 bits. Les retenues C_2 et C_3 sont celles qui ont été définies au paragraphe précédent.

```
  (+7)    0111          (-7)    1001
+ (+2)  + 0010        + (+2)  + 0010
  +9      1001          -5      1011
          C2=1, C3=0            C2=0, C3=0

  (+7)    0111          (-7)    1001
+ (-2)  + 1110        + (-2)  + 1110
  +5     10101          -9     10011
          C2=1, C3=1            C2=0, C3=1
```

La plage utilisable en CA2 4 bits est comprise entre -8 et +7. Les résultats +9 et -9 débordent donc (les deux opérandes ont même signe), mais pas les résultats +5 et -5 (les deux opérandes ont des signes contraires). On voit que le résultat est correct quand les deux retenues C_2 et C_3 sont identiques et qu'il y a débordement quand elles sont différentes. Dans notre exemple, la détection du débordement est donc : overflow = $C_2 \oplus C_3$ et d'une manière générale sur N bits :

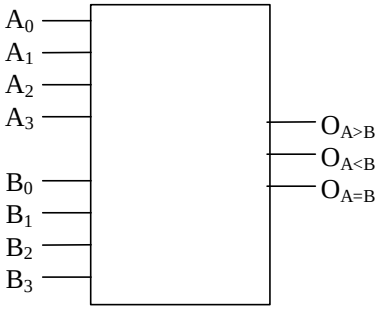
$$overflow = C_{N-2} \oplus C_{N-1}$$

Il est donc facile de rajouter ce ou exclusif sur un additionneur pour vérifier les débordements. Pour s'assurer de l'absence de débordement, on peut aussi remarquer que l'addition ou la soustraction de deux nombres codés sur N bits donne un résultat codé sur N+1 bits. Si on prévoit au départ le nombre de bits nécessaire pour contenir le résultat, le débordement est impossible. Retenez-donc que :

N bits ± N bits = N+1 bits

3.2.3 Le comparateur

Les comparateurs sont une autre grande famille de circuit arithmétique. Ils effectuent sur N bits, les opérations d'égalité, de supériorité ou d'infériorité. Le circuit suivant est un comparateur sur 4 bits :



La table de vérité d'un comparateur non signé sur 2 bits est la suivante :

A_1	A_0	B_1	B_0	A=B	A>B	A<B
0	0	0	0	1	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	0	1	0

1	0	1	0	1	0	0
1	0	1	1	0	0	1
1	1	0	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	0	1	0
1	1	1	1	1	0	0

3.2.4 Le multiplieur

La multiplication est une opération compliquée à réaliser et consommatrice de ressources en portes logiques. Avant toute étude, il faut s'assurer que cette opération est indispensable dans le design. Il faut remarquer tout d'abord qu'un décalage à gauche d'un mot binaire correspond à une multiplication par 2 et qu'un décalage à droite correspond à une division par 2. Cela implique que :

Décalage de k bits à gauche (<< k) = Multiplication par 2^k
Décalage de k bits à droite (>> k) = Division par 2^k

Ceci est parfaitement exact pour un nombre non signé, mais demande à être regardé de plus près pour un nombre en CA2. Prenons un exemple de nombre positif (31) codé en CA2 8 bits et décalons-le de 3 rangs vers la gauche.

```
00011111 << 3  = 11111000 = -8
```

31x8 n'est pas égal à -8, mais le bon résultat (248) ne tient pas sur 8 bits. En réalisant le décalage, on a débordé par valeur positive et le résultat est devenu négatif. Pour réaliser l'opération, il faut vérifier au préalable qu'il y a suffisamment de bits disponibles. Prenons un autre exemple avec cette fois un nombre négatif (-8), et décalons-le de 1 rang vers la droite.

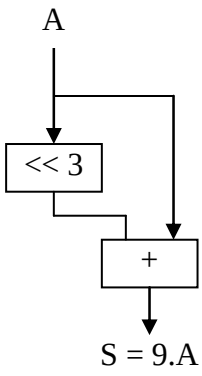
```
11111000 >> 1  = 01111100 = 124
```

-8/2 n'est pas égal à 124. Le problème ici vient du fait que l'on a inséré un 0 à gauche du nombre. Or en CA2, il faut effectuer dans ce cas ce que l'on appelle **l'extension de signe**. En cas de décalage à droite, si le bit de poids fort (MSB) est égal à 0 (nombre positif), alors on

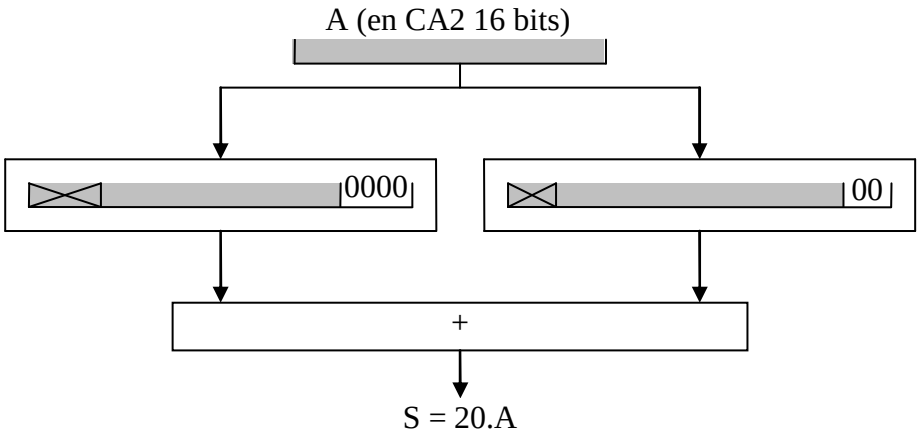
insère un 0 à gauche et si le MSB est égal à 1 (nombre négatif) alors on insère un 1 à gauche. Les résultats sont alors corrects (en prenant la partie entière du résultat bien sûr) :

31/8 = 00011111 >> 3 = **000**00011 = 3
-8/4 = 11111000 >> 2 = **11**111110 = -2

Est-il possible de faire mieux avec un simple décalage et une addition ? Supposons que l'on souhaite effectuer l'opération : S = A x 9. On remarque que A x 9 = A x 8 + A. Donc, on peut réaliser le montage suivant :



Cette méthode permet de réaliser un grand nombre de multiplication avec un coût assez faible. En effet, un décalage constant de N bits à gauche ne consomme en général aucune porte logique. Voyez sur l'exemple suivant une multiplication par 20 avec des nombres CA2 codés sur 16 bits :



Le décalage constant à gauche consiste simplement à insérer k bits à 0 à droite de A et à ignorer les k bits à gauche de A.

La vraie multiplication (quelconque et non constante) est une opération beaucoup plus compliquée. Prenons un exemple **non signé** sur 4 bits (11x14 = 154) et effectuons le calcul à la main :

```
   (14) multiplicande        1110
x  (11) multiplieur        x 1011
                          ------
                            1110
                           1110
                          0000
                         1110
                        --------
  (154) produit         10011010
```

Il faut tout de suite remarquer que dans notre exemple, 4 bits x 4 bits = 8 bits. D'une manière générale :

N1 bits x N2 bits = N1+ N2 bits

La multiplication **non signée** consiste en une série d'addition de la version décalée du multiplicande là où le bit correspondant du multiplieur vaut 1 (algorithme « Shift and Add »). Cette méthode ne marche absolument pas en CA2. Il suffit pour s'en persuader de regarder l'exemple suivant :

```
   (-1) multiplicande        1111
x  ( 1) multiplieur        x 0001
                          ------
                            1111
                           0000
                          0000
                         0000
                        --------
  (15) produit          0001111
```

Pour que cet algorithme marche en CA2, il faut convertir les deux opérandes en valeurs positives et faire la multiplication en non signé ce qui donne un résultat sur 6 bits. Puis on regarde les signes du multiplieur et du multiplicande. S'ils sont identiques, le résultat est positif et il faut ajouter 00 à gauche du produit. S'ils sont différents, le résultat est négatif et il faut convertir le produit en CA2 puis rajouter 11 sur sa gauche. Vous noterez que le résultat comporte deux bits de signe identiques. Cet algorithme est beaucoup trop lourd pour qu'on l'utilise réellement. De nombreuses études ont eu lieu pour déterminer des algorithmes plus efficaces tels que l'algorithme de Booth. Leur étude sort du cadre de ce cours. Il existe

aujourd'hui dans les FPGA des multiplieurs câblés prêt à l'emploi (par exemple : multiplieur signé CA2 18 bits x 18 bits = 36 bits).

3.2.5 Le diviseur

Plus encore que la multiplication, la division est le pire ennemi du designer. Elle est toujours plus lente qu'une multiplication (dans un microprocesseur style PowerPC, la multiplication se calcule en une période d'horloge alors que la division se calcule en 4 périodes d'horloge). Dans la mesure du possible, on évite toujours d'utiliser cette opération en conception de circuit intégré (ainsi d'ailleurs qu'en développement logiciel). Si on ne peut pas l'éviter, la méthode des décalages peut toujours être utile pour une division par une puissance de 2. Dans le cas général, on pose Numérateur = Dénominateur x Quotient + Reste, ce qui donne :

$$\frac{N}{D} = Q, \quad |R| < D, \text{ par exemple : } \frac{10}{3} = 3, \quad R = 1$$

Le problème du nombre de bits pour le codage n'a rien de simple. Supposons que N et D soient des entiers codés sur 16 bits, alors Q a de bonnes chances de rester à 0 la plupart du temps, en tout cas dès que D sera plus grand que N ($\frac{99}{100} = 0$ avec des entiers). Si N n'est pas beaucoup plus grand que D, le résultat de la division risque d'être très peu précis. Si D = 0, la division est impossible.

Pour s'en sortir, il faut raisonner en virgule fixe. Un Q15/16 est un nombre codé sur 16 bits dont 15 à droite de la virgule : -,----------. Sa dynamique va de -1 à +(1 - 2^{-15}). Le bit à gauche de la virgule est le bit de signe. Pour que la division ait une dynamique maximale en sortie, il faut effectuer $\frac{2^{16}.N}{D} = Q$. D'un point de vue du codage, 2^{16} x Q15/16 = Q31/32. On décale la virgule de 16 rangs vers la gauche en multipliant par 2^{16}, ce qui donne -,----------0000000000000000, donc un Q31/32 (dynamique : -1 à +(1 - 2^{-15})). Regardons maintenant le codage en virgule fixe du résultat de la division :

$$\frac{Q31/32}{Q15/16} = Q16/32$$

Comment obtient-on Q16/32 en sortie ? Il faut raisonner en termes de dynamique. Prenons des valeurs positives :

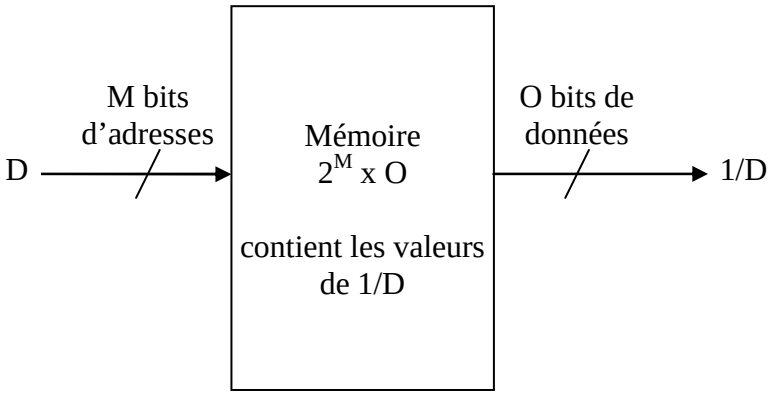
N max ÷ D min	N min ÷ D max
<u>en entier :</u> sur 16 bits : Nmax = $2^{15} - 1$ et Dmin = 1 Q = $2^{16}.(2^{15} - 1)/1 = 2^{31}-2^{16}$ Q = 01111111111111110000000000000000	<u>en entier :</u> sur 16 bits : Nmin = 1 et Dmax = $2^{15} - 1$ Q = $2^{16}.1/(2^{15} - 1) = 2{,}00006\ldots = 2$ Q = 00000000000000000000000000000010
<u>en virgule fixe :</u> Nmax (en Q31/32) = $(1 - 2^{-15})$ Dmin (en Q15/16) = 2^{-15} Q = $(1 - 2^{-15})/ 2^{-15} = 2^{15} - 1$ Q = 0111111111111111,0000000000000000 la virgule est au milieu du mot, c'est bien un Q16/32	<u>en virgule fixe :</u> Nmin (en Q31/32) = 2^{-15} Dmax (en Q15/16) = $1 - 2^{-15}$ Q = $2^{-15} / (1 - 2^{-15}) = 2^{-15}$ Q = 0000000000000000,0000000000000010 la virgule est au milieu du mot, c'est bien un Q16/32

Facile non ? Vous noterez qu'il faut rester sur 32 bits pour conserver la dynamique complète de la division. Si vous désirez revenir sur 16 bits, ce qui est en général le cas, il va falloir choisir les 16 bits à extraire en fonction de la dynamique souhaitée. La division ne marchera donc pas dans tous les cas.

Il existe des algorithmes de calcul, qui consomment dans notre exemple entre 20 et 40 cycles d'horloge suivant le contenu de N et D. Le même problème existe aussi avec un DSP. L'étude des algorithmes de division sort du cadre de ce cours. Il faut noter qu'il existe une méthode efficace quoique par forcément précise qui consiste à transformer la division en multiplication par l'inverse du dénominateur.

$$Q = \frac{N}{D} = N.\frac{1}{D}$$

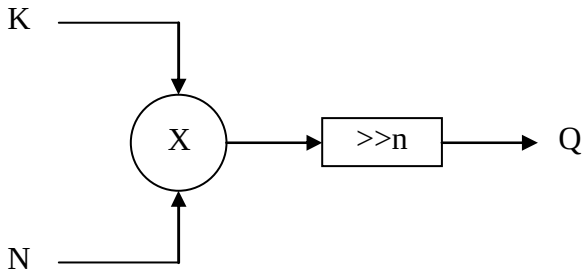
Il suffit maintenant de coder la valeur de 1/D dans une mémoire, de mettre D sur les adresses puis de lire 1/D sur les données :



Le choix de M et de O dépend des ressources disponibles, de la précision désirée sur la valeur de 1/D et de l'écart entre 2 valeurs de 1/D adjacentes. Avec cette méthode, la division N/D prend en général 2 périodes d'horloges, une pour l'accès mémoire (qui est généralement synchrone) et une pour la multiplication.

Une autre astuce est aussi possible avec la formule $Q = \frac{N}{D} = N.\frac{1}{D}$ mais en posant $\frac{1}{D} \approx \frac{K}{2^n}$.

Cela conduit au montage suivant :



Prenons l'exemple d'une division par 7 : K = 73, n = 9 d'où $\frac{1}{D} = \frac{73}{512}$ ce qui donne :

$$Q = N.\frac{73}{512} = \frac{N}{7.013}$$

En jouant sur le nombre de bits de codage de K et n, on peut améliorer la précision du calcul. Cette méthode est efficace (un cycle d'horloge) pour une division par une constante, mais la gestion de la précision n'est pas évidente si D a une dynamique très élevée.

3.3 Description en VHDL

3.3.1 Représentation des nombres en VHDL

Il existe deux types de données scalaires permettant de représenter des nombres en VHDL :

- Les entiers (qui sont synthétisables) avec par exemple :

  `signal I : integer range -128 to 127;` I sera codé sur 8 bits en CA2

  `signal J : integer range 0 to 15;` J sera codé sur 4 bits non signés

  `signal K : integer range -32768 to 32767;` K sera codé sur 16 bits en CA2

  `signal L : integer;` L sera codé sur 32 bits en CA2 (par défaut)

- Les réels (qui ne sont pas synthétisables mais que l'on peut utiliser dans un testbench) avec par exemple :

  `signal X : real;` X peut être compris entre -10^{38} et 10^{38} (float 32 bits par défaut)

On peut aussi utiliser des tableaux de variables de type `bit` comme `bit_vector` ou de type `std_logic` comme `std_logic_vector`, ce dernier étant le plus utilisé. Par défaut, ils sont non typés et ne permettent donc pas les opérations arithmétiques telles que l'addition ou la multiplication. Pour pouvoir faire de l'arithmétique notamment avec le type `std_logic_vector`, on peut utiliser deux familles de packages équivalentes mais non identiques : les packages propriétaires synopsys (std_logic_arith, std_logic_unsigned et std_logic_signed abusivement placés dans la librairie IEEE) et les packages normalisés IEEE 1076.3 (numeric_bit et numeric_std).

Les packages IEEE 1076.3 sont hélas arrivés trop tardivement (en 1997) alors que la première version du langage VHDL (IEEE 1076) date de 1987 et que la société Synopsys a développé le premier synthétiseur en 1990. Comme il n'y avait alors aucun package permettant l'interprétation binaire pour l'arithmétique, Synopsys a développé les siens et ils ont été repris par tous les autres fabricants de synthétiseurs. Les packages Synopsys sont devenus la norme de fait. Le temps que les packages IEEE soient développés puis reconnus par les vendeurs d'outils de synthèse, tous les concepteurs avaient pris l'habitude d'utiliser ceux de Synopsys d'autant que les deux familles de packages offrent les mêmes fonctionnalités. Aujourd'hui, les packages Synopsys et IEEE sont reconnus par tous les synthétiseurs mais les packages Synopsys restent les plus utilisés. Voyons maintenant le contenu de ces derniers.

3.3.2 Le package std_logic_arith

3.3.2.1 Les types signed et unsigned

Les types `signed` et `unsigned` ont la même définition que le type `std_logic_vector` (`type` **`std_logic_vector`** `is` **`array`** (**`NATURAL`** `range <>) of std_logic;`). Ce sont des types proches (closely related). Les opérateurs arithmétiques, de comparaisons et les fonctions de conversion définis dans std_logic_arith traiteront de la même manière les types `signed` et `unsigned` sauf que `signed` sera interprété comme un nombre binaire signé codé en CA2 et `unsigned` sera interprété comme un nombre binaire non signé.

3.3.2.2 Les fonctions de conversion

3.3.2.2.1 conv_integer

Cette fonction convertit un argument (`arg`) de type `signed` ou `unsigned` (mais pas un `std_logic_vector`, voir §3.3.3) en `integer` :

```
function conv_integer(arg: unsigned) return integer;
function conv_integer(arg: signed) return integer;
```

Exemples d'utilisation :

```
signal b : std_logic_vector(3 downto 0);
signal u1 : unsigned(3 downto 0);
signal s1 : signed(3 downto 0);
signal i1, i2, i3 : integer;

u1 <= "1001";
s1 <= "1001";
b <= "0001";
i1 <= conv_integer(u1);  -- 9
i2 <= conv_integer(s1);  -- -7
i3 <= conv_integer(b);   -- erreur en simulation et synthèse
```

3.3.2.2.2 conv_unsigned

Cette fonction convertit un argument de type `signed` ou `integer` (mais pas un `std_logic_vector`, voir §3.3.3) en `unsigned` sur `size` bits.

```
function conv_unsigned(arg: integer, size: integer) return unsigned;
function conv_unsigned(arg: signed, size: integer) return unsigned;
```

Exemples d'utilisation :

```
signal u1, u2 : unsigned(3 downto 0);
signal s1 : signed(3 downto 0);
signal i1, i2 : integer;

s1 <= "0101";
i1 <= 13;
```

```
u1 <= conv_unsigned(s1, 4);   -- = "0101" = 5
u2 <= conv_unsigned(i1, 4);   -- = "1101" = 13
```

3.3.2.2.3 conv_signed

Cette fonction convertit un argument de type `unsigned` ou `integer` (mais pas un `std_logic_vector`, voir §3.3.3) en `signed` sur `size` bits.

```
function conv_signed(arg: integer, size: integer) return signed;
function conv_signed(arg: unsigned, size: integer) return signed;
```

Exemples d'utilisation :

```
signal u1 : unsigned(3 downto 0);
signal s1, s2, s3 : signed(3 downto 0);
signal i1, i2 : integer;

u1 <= "0101";
i1 <= 6;
i2 <= -2;

s1 <= conv_signed(u1, 4);   -- = "0101" = +5
s2 <= conv_signed(i1, 4);   -- = "0110" = +6
s3 <= conv_signed(i2, 4);   -- = "1110" = -2
```

3.3.2.2.4 conv_std_logic_vector

Cette fonction convertit un argument de type `signed`, `unsigned` ou `integer` en std_logic_vector sur `size` bits.

```
function conv_std_logic_vector(arg: integer, size: integer) return std_logic_vector;
function conv_std_logic_vector(arg: unsigned, size: integer) return std_logic_vector;
function conv_std_logic_vector(arg: signed, size: integer) return std_logic_vector;
```

Exemples d'utilisation :

```
signal u1 : unsigned(3 downto 0);
signal s1 : signed(3 downto 0);
signal v1, v2, v3, v4 : std_logic_vector(3 downto 0);
signal i1, i2 : integer;

u1 <= "1101";
s1 <= "1101";
i1 <= 13;
i2 <= -2;

v1 <= conv_std_logic_vector(u1, 4);   -- = "1101"
v2 <= conv_std_logic_vector(s1, 4);   -- = "1101"
v3 <= conv_std_logic_vector(i1, 4);   -- = "1101"
v4 <= conv_std_logic_vector(i2, 4);   -- = "1110"
```

3.3.2.2.5 Conversion de types proches (closely related)

Si les types à convertir sont suffisamment proches, le type lui-même peut servir de fonction de conversion. Par exemple on peut écrire `signed(i)` ou `unsigned(j)` ou encore `std_logic_vector(k)`. Le mélange des types étant interdit en VHDL (le langage est

fortement typé), il faut faire la conversion même si les types ont la même définition. Bien sûr, les valeurs ne changent pas. Voici des exemples d'utilisation :

```
signal u1, u2, u3 : unsigned(3 downto 0);
signal s1, s2, s3 : signed(3 downto 0);
signal v1, v2, v3 : std_logic_vector(3 downto 0);
signal i1, i2 : integer;

u3 <= "1101";
s3 <= "0001";
v3 <= "1001";

v1 <= std_logic_vector(u3);   -- = "1101"
v2 <= std_logic_vector(s3);   -- = "0001"
s1 <= signed(u3);             -- = "1101"
s2 <= signed(v3);             -- = "1001"
u1 <= unsigned (v3);          -- = "1001"
u2 <= unsigned (s3);          -- = "0001"
i1 <= conv_integer(unsigned(v3)); -- conv_integer n'accepte pas les std_logic_vector
```

3.3.2.3 Opérateur arithmétiques : +, -, *

Chaque opérateur peut porter sur un type `signed`, `unsigned` ou `integer` mais en aucun cas sur un `std_logic_vector` (voir §3.3.3). Voici des exemples d'utilisation :

```
signal u1, u2 : unsigned (3 downto 0);
signal s1, s3 : signed (3 downto 0);
signal s2 : signed (4 downto 0);
signal v1 : std_logic_vector (3 downto 0);
signal v2 : std_logic_vector (4 downto 0);
signal i1, i2 : integer;

u1 <= "1001";    -- = 9
s1 <= "1001";    -- = -7
i1 <= 16;

s2 <= u1 + s1;   -- = 2
v2 <= u1 + s1;   -- = "00010"
u2 <= i1 - u1;   -- = 7 : erreur à la synthèse car i1=32 bits et u1=4 bits
s3 <= -s1;       -- = 7
```

3.3.2.4 Opérateurs de comparaison : <, <=, >, >=, =, /=

Chaque opérateur peut porter sur un type `signed`, `unsigned` ou `integer` mais en aucun cas sur un `std_logic_vector` (voir §3.3.3). Il retourne un booléen (vrai/faux). On reconnait les comparaisons suivantes :

<	inférieur	>	supérieur	=	Egal à
<=	inférieur ou égal	>=	supérieur ou égal	/=	Différent de

Exemples d'utilisation :

```
signal u1, u2 : unsigned (3 downto 0);
signal s1 : signed (3 downto 0);
signal o1,o2 : std_logic;
```

```
u1 <= "1001";    -- = 9
u2 <= "0111";    -- = 7
s1 <= "1001";    -- = -7
…
if u1 > u2 then
  o1 <= '1';       -- o1 set to '1'
else
  o1 <= '0';
end if;
…
if s1 > u2 then
  o2 <= '1';
else
  o2 <= '0';       -- o2 set to '0'
end if;
```

3.3.2.5 Fonctions de décalage : SHL, SHR

La fonction de décalage à gauche SHL (SHift Left) et à droite SHR (SHift Right) portent sur un type `signed` ou `unsigned`. Lorsque SHR porte sur un type `signed`, il réalise l'extension de signe.

```
function shl(arg: unsigned; count: unsigned) return unsigned;
function shl(arg: signed; count: unsigned) return signed;
function shr(arg: unsigned; count: unsigned) return unsigned;
function shr(arg: signed; count: unsigned) return signed;
```

Il faut indiquer à la fonction le nombre de décalage à prendre en compte avec une variable de type `unsigned`. Voici des exemples d'utilisation :

```
architecture comporte of dec is
  signal u1, u2, u3 : unsigned(3 downto 0);
  signal s1, s2, s3 : signed(3 downto 0);
begin
  u1 <= "0010";
  u2 <= shl(u1, "10");  -- "1000"
  u3 <= shr(u2, "10");  -- "0010" : u2 non signé, pas d'extension de signe

  s1 <= "0010";
  s2 <= shl(s1, "10");  -- "1000"
  s3 <= shr(s2, "10");  -- "1110" : u2 signé, extension de signe
end comporte ;
```

3.3.3 Les packages std_logic_unsigned et std_logic_signed

La fonction de conversion `conv_integer()` vue dans la std_logic_arith ne peut pas convertir un `std_logic_vector` en `integer` car elle ne peut pas déterminer s'il s'agit d'une valeur signée ou non signée. Les packages std_logic_unsigned et std_logic_signed permettent de résoudre ce problème. Ils contiennent les mêmes opérateurs arithmétiques (+, -, *), de comparaisons (<, <=, >, >=, =, /=) et fonctions de décalage (SHR, SHL) que la std_logic_arith plus la fonction `conv_integer()` qui accepte un `std_logic_vector` et qui retourne un `integer`.

Si vous utilisez le package std_logic_unsigned avant la déclaration de l'entité en VHDL, tous les `std_logic_vector` seront traités comme des entiers non signés (en fait, comme un type `unsigned`) y compris avec la fonction `conv_integer()`.

Si vous utilisez le package std_logic_signed avant la déclaration de l'entité en VHDL, tous les `std_logic_vector` seront traités comme des entiers signés (en fait, comme un type `signed`) y compris avec la fonction `conv_integer()`.

Attention, tous les `std_logic_vector` de l'entité et de l'architecture seront soit signés, soit non signés. Le mélange des deux est impossible.

3.3.4 Exemples

3.3.4.1 Additionneur simple non signé ou signé

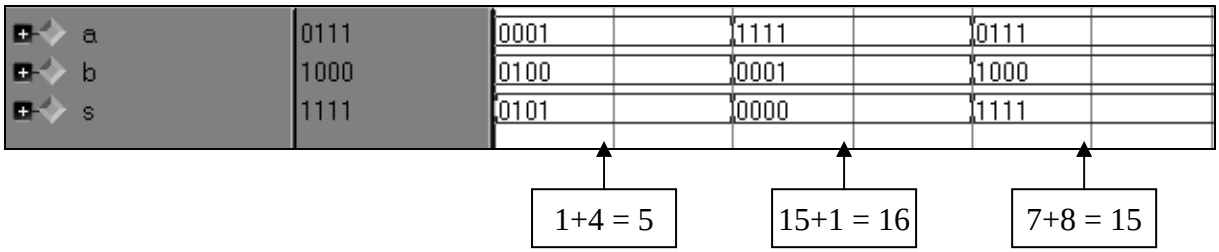
La réalisation d'un additionneur 4 bits non signé est assez intuitive comme le montre l'exemple suivant :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity add is
  port(A : in std_logic_vector(3 downto 0);
       B : in std_logic_vector(3 downto 0);
       S : out std_logic_vector(3 downto 0));
end add;

architecture comporte of add is
begin
     S <= A + B;
end comporte ;
```

Il faut juste savoir que l'opérateur d'addition **+** ne gère pas les retenues entrantes et sortantes. En clair, les deux entrées et la sortie doivent être de même taille :

N bits + N bits = N bits

Dans cet exemple, l'utilisation du package std_logic_unsigned implique que tous les `std_logic_vector` sont considérés comme des `unsigned`. Le résultat de la simulation est le suivant :



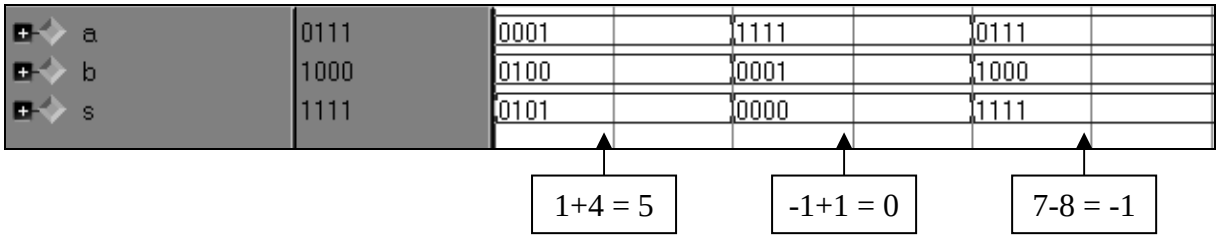
Pour réaliser un additionneur signé, il suffit de remplacer la ligne :

```
use IEEE.std_logic_unsigned.all;
```

par la ligne :

```
use IEEE.std_logic_signed.all;
```

Le résultat de la simulation est alors :



En interprétant les signaux A, B et S comme des nombres signés, on obtient bien les résultats attendus. Mais en binaire, les résultats sont strictement identiques à l'exemple non signé. Ceci est parfaitement normal puisque l'opérateur d'addition agit de la même manière sur les nombres signés et non signés.

Dans le cas d'une addition, travailler en signé ou en non signé est une vue de l'esprit. En binaire, les résultats sont identiques dans les deux cas. Mais ce n'est pas vrai pour les calculs de débordement (overflow).

3.3.4.2 Additionneur avec retenue entrante et sortante

Pour pouvoir gérer les retenues entrantes et sortantes sans trop de difficulté, il suffit de réaliser l'addition avec 1 bit supplémentaire sur le poids fort. Dans l'exemple suivant, on réalise une addition signée sur 4 bits en utilisant le package std_logic_arith (pour varier un peu car on aurait parfaitement pu utiliser la std_logic_signed, les deux solutions étant équivalentes).

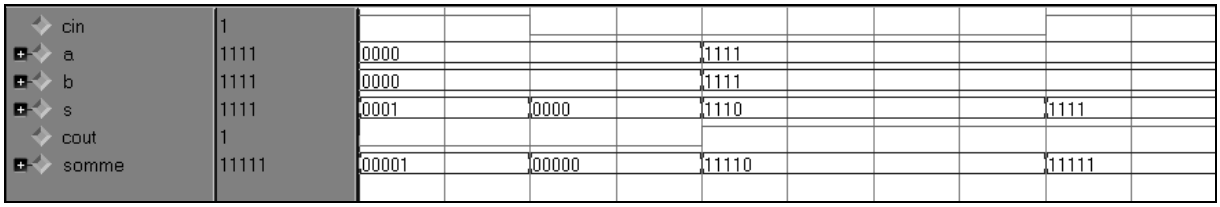
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
entity add is
  port(Cin : in std_logic;
         A : in signed(3 downto 0);
         B : in signed(3 downto 0);
         S : out signed(3 downto 0);
      Cout : out std_logic);
end add;

architecture comporte of add is
     signal Somme : signed(4 downto 0);
begin
     Somme <= (A(3)&A) + (B(3)&B) + Cin;
     S <= Somme(3 downto 0);
     Cout <= Somme(4);
end comporte ;
```

A la ligne :

```
     Somme <= (A(3)&A) + (B(3)&B) + Cin;
```

A et B sont étendus sur 5 bits par concaténation (avec extension de signe car nous sommes en signé) puis additionnés avec la retenue entrante Cin. Les parenthèses sont obligatoires autour de la concaténation car l'addition + est plus prioritaire que la concaténation &. Somme est sur 5 bits. La sortie S correspond aux 4 bits de poids faible de Somme et la retenue Cout correspond à son bit de poids fort. On peut constater sur la simulation suivante que la gestion des retenues est correcte.



3.3.4.3 Additions multiples sans gestion de retenue

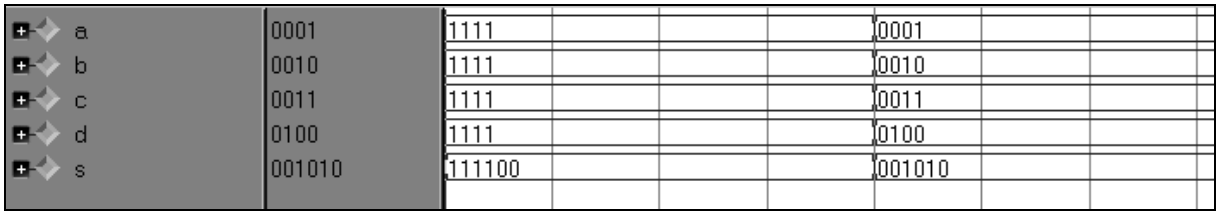
Quand on veut additionner plusieurs valeurs, la gestion des retenues peut devenir assez rapidement pénible. Il existe une méthode simple pour s'en passer : il suffit de prévoir la taille du résultat, de convertir les entrées à cette taille puis d'additionner sans s'occuper des retenues. Dans l'exemple suivant, on additionne 4 variables 4 bits A, B, C et D ce qui doit donner un résultat sur 6 bits. On va convertir les entrées sur 6 bits en concaténant 2 bits à gauche de chaque variable (00 dans cet exemple puisque nous travaillons en non signé, avec extension de signe pour un exemple signé), puis effectuer l'addition :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity add is
  port(A : in std_logic_vector(3 downto 0);
       B : in std_logic_vector(3 downto 0);
       C : in std_logic_vector(3 downto 0);
       D : in std_logic_vector(3 downto 0);
       S : out std_logic_vector(5 downto 0));
end add;

architecture comporte of add is
     signal S1 : std_logic_vector(5 downto 0);
     signal S2 : std_logic_vector(5 downto 0);
begin
     S1 <= ("00"&A) + ("00"&B);
     S2 <= ("00"&C) + ("00"&D);
     S <= S1 + S2;
end comporte ;
```

Les parenthèses sont obligatoires autour de la concaténation. On aurait pu réaliser l'addition sur une seule ligne de la manière suivante :

```
     S <= ("00"&A) + ("00"&B) + ("00"&C) + ("00"&D);
```

On peut constater sur la simulation suivante que le résultat S obtenu est correct.



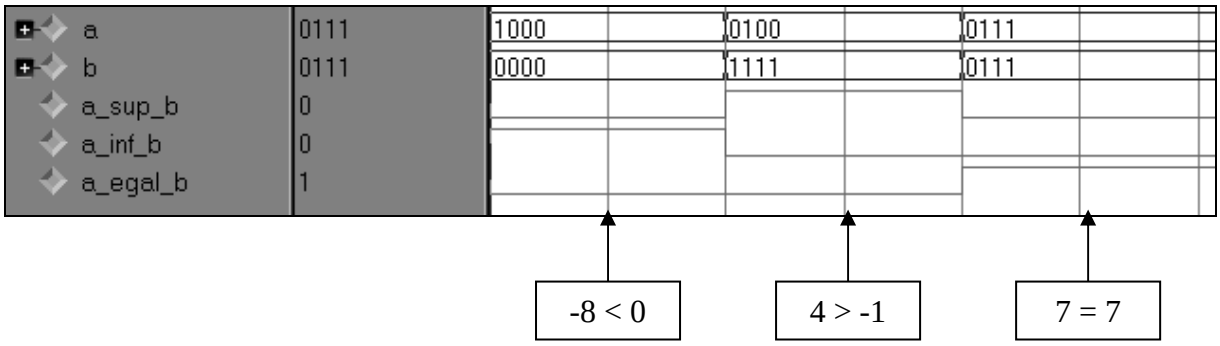
3.3.4.4 Comparateur

La réalisation d'un comparateur est assez évidente. Dans l'exemple suivant, tous les signaux sont signés :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
```

```
entity comp is
  port(  A : in std_logic_vector(3 downto 0);
         B : in std_logic_vector(3 downto 0);
   A_sup_B : out std_logic;
   A_inf_B : out std_logic;
  A_egal_B : out std_logic);
end comp;

architecture comporte of comp is
begin
     process(A, B) begin
          if (A > B) then
               A_sup_B <= '1';
               A_inf_B <= '0';
               A_egal_B <= '0';
          elsif (A < B) then
               A_sup_B <= '0';
               A_inf_B <= '1';
               A_egal_B <= '0';
          else
               A_sup_B <= '0';
               A_inf_B <= '0';
               A_egal_B <= '1';
          end if;
     end process;
end comporte ;
```

Le résultat de la simulation est le suivant :



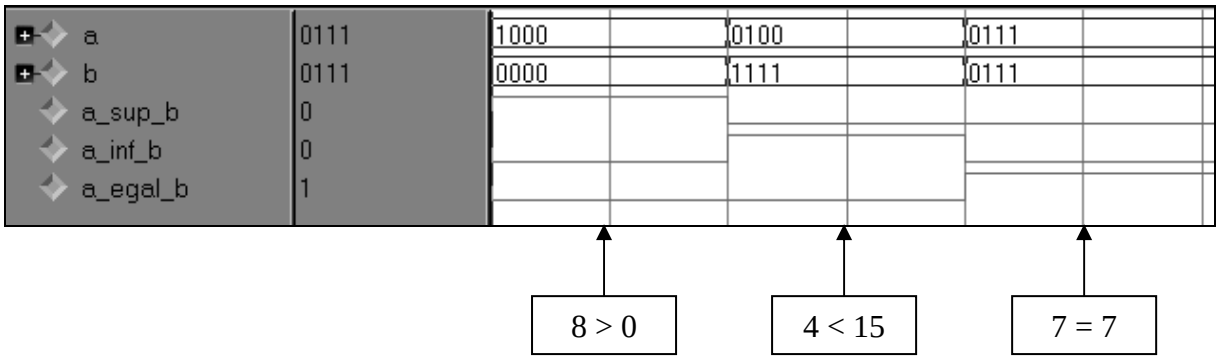
Pour réaliser un comparateur non signé, il suffit de remplacer la ligne :

```
use IEEE.std_logic_signed.all;
```

par la ligne :

```
use IEEE.std_logic_unsigned.all;
```

Les résultats de la simulation sont alors différents du cas précédent :



Dans le cas d'une comparaison d'inégalité, travailler en signé ou en non signé peut donner des résultats différents.

Un piège traditionnel des comparaisons signées est illustré par l'exemple suivant :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity comp is
  port(A : in std_logic_vector(3 downto 0);
  flag_A : out std_logic);
end comp;

architecture comporte of comp is
begin
     process(A) begin
          if (A > 7) then
               flag_A <= '1';
          else
               flag_A <= '0';
          end if;
     end process;
end comporte;
```

Ce comparateur n'est jamais activé en simulation et le synthétiseur ne génère aucune porte logique pour ce design. En effet, A est codé en signé sur 4 bits et ne peut donc pas dépasser la valeur 7. La condition :

```
if (A > 7) then
```

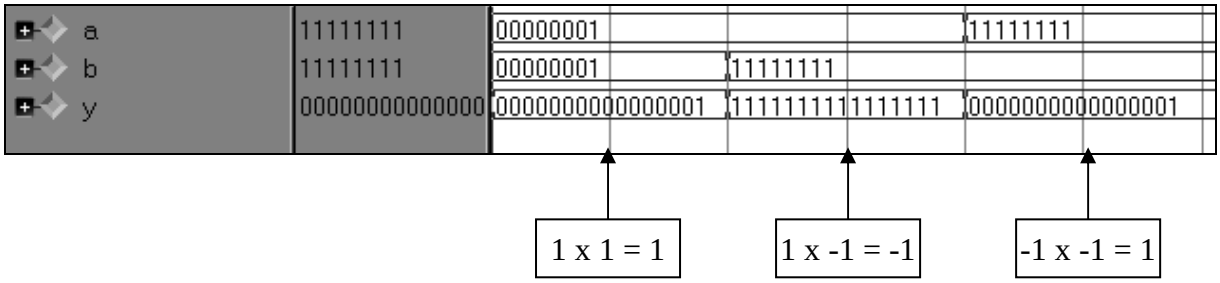
n’est jamais vraie et le signal flag_A est toujours égal à 0. Mais si vous utilisez la std_logic_unsigned à la place de la std_logic_signed, alors A peut aller de 0 jusqu’à 15 et la comparaison peut être activée. Le synthétiseur génère alors un comparateur.

3.3.4.5 Multiplieur

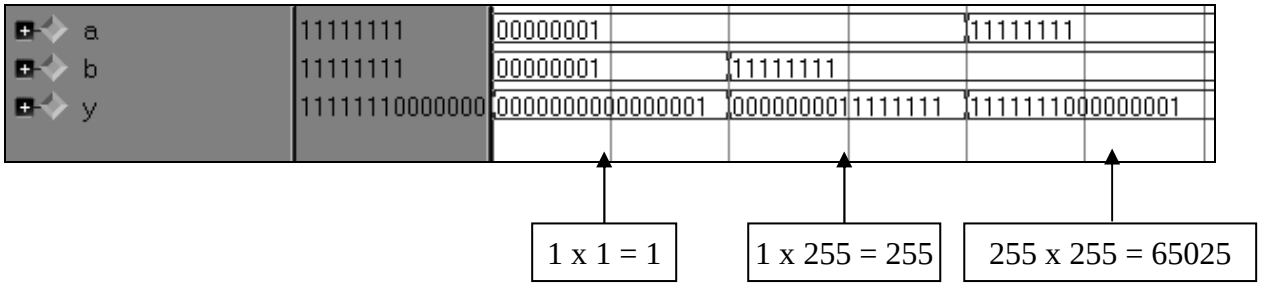
La multiplication est un autre exemple de différence de comportement entre opération signée et non signée. L’exemple suivant est une multiplication signée 8 bits x 8 bits = 16 bits :

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity mult is
  port(A : in std_logic_vector(7 downto 0);
       B : in std_logic_vector(7 downto 0);
       Y : out std_logic_vector(15 downto 0));
end mult;

architecture comporte of mult is
begin
     Y <= A * B;
end comporte ;
```

Le résultat de la simulation est le suivant :



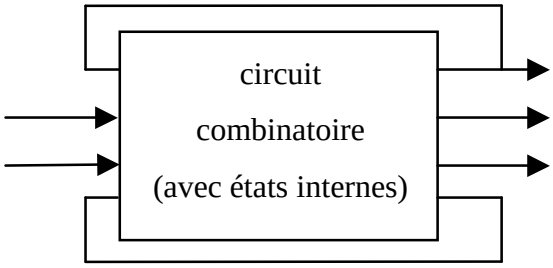
En utilisant la std_logic_unsigned à la place de la std_logic_signed, la multiplication devient non signée et la simulation donne les résultats suivants :



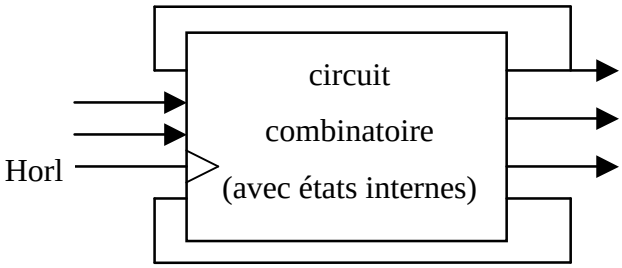
4 Logique séquentielle

Dans un circuit combinatoire, une sortie est uniquement fonction des entrées. Par contre, dans un circuit séquentiel, une sortie est une fonction des entrées mais aussi des sorties du circuit. Il y a rebouclage (rétroaction) des sorties sur les entrées. Cela signifie qu'un circuit séquentiel garde la mémoire des états passés. Il existe deux grandes catégories de circuit séquentiel :

- Le circuit séquentiel asynchrone. Les sorties du montage peuvent changer à tout moment dès qu'une ou plusieurs entrées changent après un temps de propagation qui peut être différent pour chaque sortie.

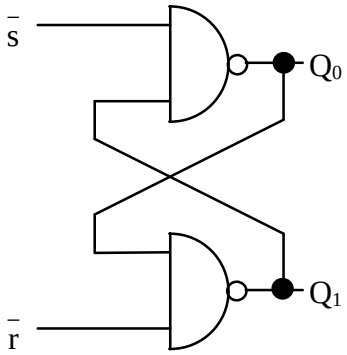


- Le circuit séquentiel synchrone. Le changement sur les sorties se produit après le changement d'état (front montant ou descendant) d'un signal maître, l'horloge. Les entrées servent à préparer le changement d'état, mais ne provoquent pas de changement des sorties. Tout changement d'état interne du montage est synchronisé sur le front actif de l'horloge.



4.1 Circuits séquentiels asynchrones

La forme la plus élémentaire de circuit séquentiel, que l'on appelle un latch (verrou), est la suivante :



$$Q_0 = \overline{\bar{s}.\overline{Q_1}} = s + \overline{Q_1}\,,\ Q_1 = \overline{\bar{r}.\overline{Q_0}} = r + \overline{Q_0}$$

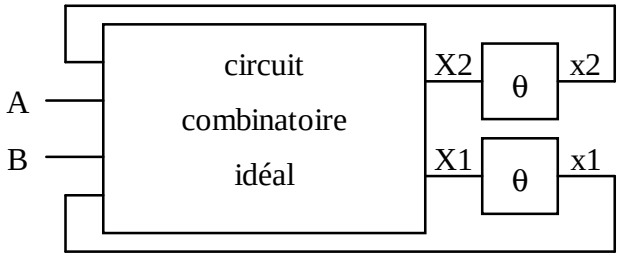
$$\Rightarrow Q_0 = s + \bar{r}.Q_0\,,\ Q_1 = r + \bar{s}.Q_1$$

Sa table de vérité est :

$\bar{s}$	$\bar{r}$	Q_0	Q_1
0	0	interdit	interdit
0	1	1	0
1	0	0	1
1	1	Q_0	Q_1

L'état 0,0 est interdit car il conduit à un état instable comme nous le verrons par la suite. Nous allons appliquer la méthode des tables d'excitation pour analyser dans le détail le fonctionnement de ce circuit.

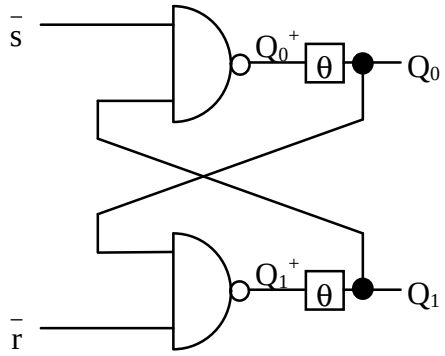
Tout circuit logique asynchrone comportant des boucles de réaction possède un fonctionnement séquentiel. On peut modéliser un circuit séquentiel en ajoutant des retards dans les boucles de réaction. Bien que les retards physiques, fonctions des temps de propagation à travers les portes élémentaires, ne soient pas égaux, on convient de symboliser la fonction « retard » par une même valeur θ pour l'étude des circuits asynchrones.



On définit :

- Les variables primaires A et B. Ce sont les entrées réelles du circuit.
- Les variables secondaires (ou variable interne) x1 et x2. Ce sont les sorties réinjectées sur l'entrée du circuit combinatoire idéal.
- Les variables d'excitation X1 et X2. Ce sont les sorties du circuit combinatoire idéal sur lesquelles a été placé un retard.

Dans le cas du latch, on obtient le schéma asynchrone équivalent :



$\bar{s}$ $\bar{r}$ sont les variables primaires, Q_0 Q_1 sont les variables secondaires et Q_0^+ Q_1^+ sont les variables d'excitation. On établit la table d'excitation donnant $Q_0^+,Q_1^+=F(\bar{s},\bar{r},Q_0,Q_1)$ avec les équations combinatoires $Q_0^+ = \overline{\bar{s}.Q_1}$ et $Q_1^+ = \overline{\bar{r}.Q_0}$.

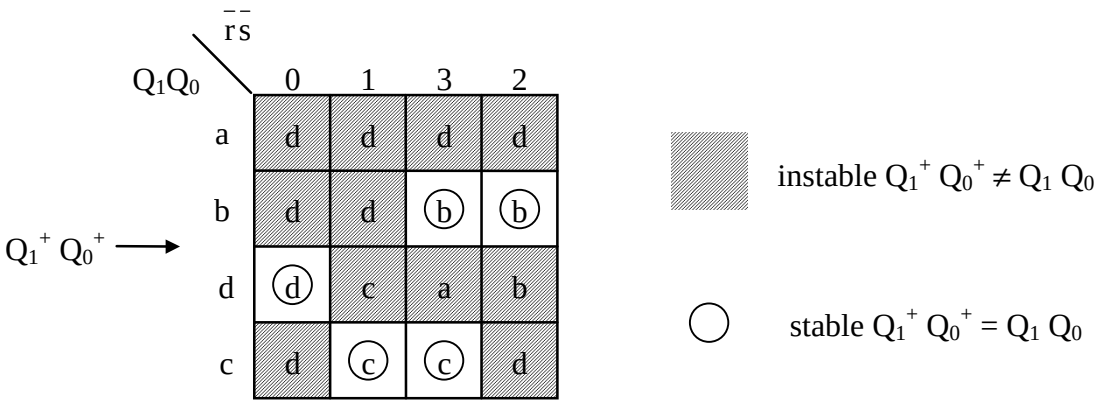
$Q_1^+ Q_0^+$ ($Q_1 Q_0$ \ $\bar{r}\bar{s}$)	0 0	0 1	1 1	1 0
00	11	11	11	11
01	11	11	01	01
11	11	10	00	01
10	11	10	10	11

Puis on change de notation pour obtenir une table d'états internes :

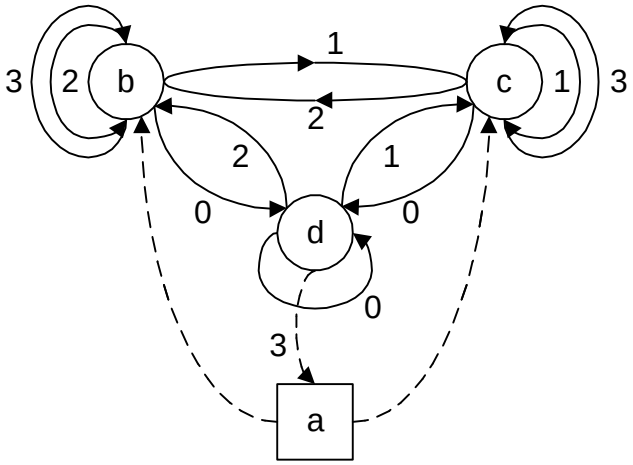
$\bar{s}$	$\bar{r}$	notation
0	0	0
0	1	1
1	0	2
1	1	3

Q_0	Q_1	notation
0	0	a
0	1	b
1	0	c
1	1	d

La table d'états internes obtenue est :



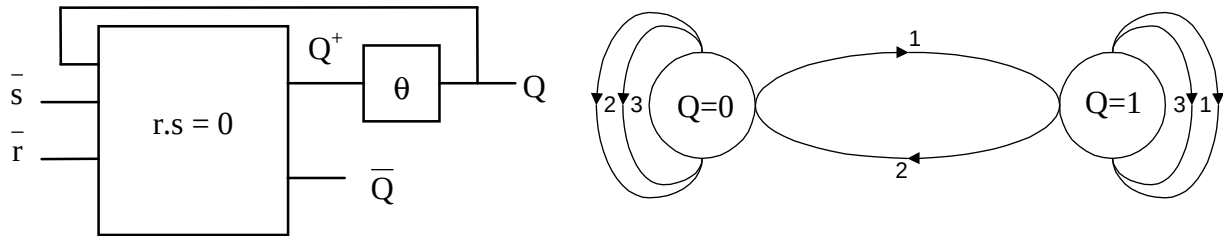
On voit que l'état interne (a) est toujours instable. A partir de l'état (b), on cherche sur la table des états internes les effets de toutes les commandes, puis on recommence pour les états (c) et (d). On obtient ainsi le graphe d'évolution du montage.



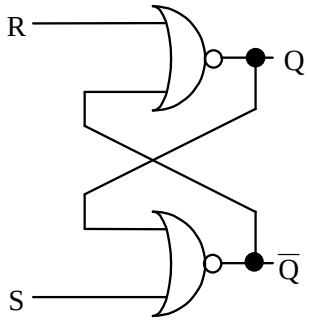
En (d), si $\bar{s}\,\bar{r}=3$, on va en (a), état instable. On a deux choix possibles : vers l'état b ou l'état c. L'état choisi dépendra de la vitesse des portes (des valeurs de θ), la porte la plus rapide passant à 0 en premier. Dans un circuit séquentiel, quand plusieurs changements d'états internes se produisent pour un seul changement sur les commandes, alors on dit qu'il se produit une « race condition ». Quand l'état final ne dépend pas de l'ordre des changements d'états internes, alors il s'agit d'une « noncritical race ». Quand l'état final du circuit dépend de l'ordre des changements d'états internes, alors il s'agit d'une « critical race ». C'est le cas pour le latch $\bar{s}\,\bar{r}$. Il faut toujours s'assurer qu'aucune « critical race » ne peut se produire dans ce type de circuit séquentiel avec rétroaction combinatoire car son comportement devient alors totalement imprédictible. L'interprétation du graphe est donc la suivante :

- L'état (d) est forcé par 0 et gardé en mémoire par 0.
- L'état (b) est forcé par 2 et gardé en mémoire par 3 ou 2.
- L'état (c) est forcé par 1 et gardé en mémoire par 3 ou 1.
- Les commandes sont équivalentes à :
  0. Commande interdite. On évite ainsi le problème dû à la commande 3 sur (d). On impose r.s = 0 à l'entrée de la bascule, ce qui supprime la « critical race ».
  1. Remise à un ⇒ état (c).
  2. Remise à zéro (RAZ) ⇒ état (b).
  3. Mémoire.

Le modèle final du latch valable si r.s = 0 et son graphe d'évolution sont :



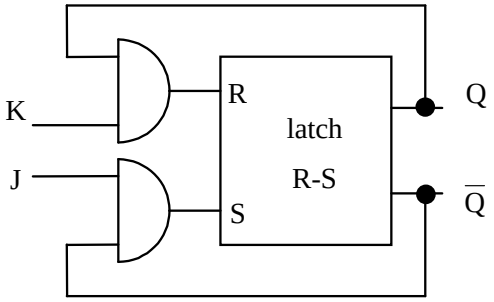
avec l'équation d'excitation : $Q^+ = s + \bar{r}.Q$. On peut réaliser une variante de ce latch avec des portes NOR, le latch RS :



L'équation ne change pas : $Q^+ = S + \overline{R}.Q$ avec R.S = 0. La table de vérité est :

R	S	Q^+	fonction
0	0	Q	mémoire
0	1	1	mise à 1
1	0	0	mise à 0
1	1	X	interdit

Afin de s'affranchir de l'état instable, on définit le latch JK de la manière suivante :



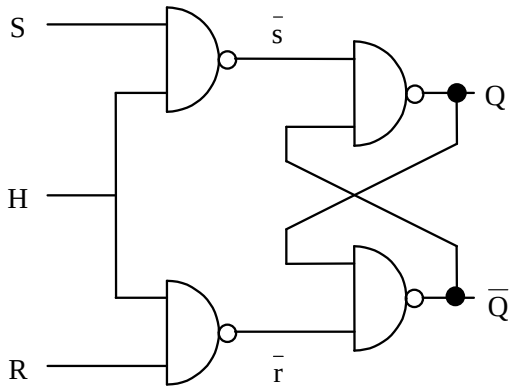
On garantit de cette manière que l'état R = 1, S = 1 ne se produira jamais en posant R = K.Q et $S = J.\overline{Q}$ ce qui nous donne l'équation de sortie suivante : $Q^+ = J.\overline{Q} + \overline{K}.Q$. La table de vérité vaut alors :

J	K	Q^+	fonction
0	0	Q	mémoire
0	1	0	mise à 0
1	0	1	mise à 1
1	1	$\overline{Q}$	inversion

L'état instable du latch RS s'est transformé en un état inversion.

4.2 Bistables synchronisés sur un niveau

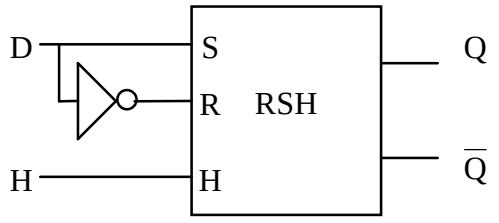
Nous allons maintenant essayer de synchroniser le fonctionnement du latch avec un signal d'horloge à l'aide du schéma :



En respectant r.s = 0, on a $Q^+ = s + \bar{r}.Q$, d'où on tire $Q^+ = S.H + \overline{R.H}.Q$ avec (R.H).(S.H) = R.S.H = 0. On obtient la table de vérité suivante :

R	S	H	Q_{n+1}	fonction
X	X	0	Q_n	mémoire
0	0	1	Q_n	mémoire
0	1	1	1	mise à 1
1	0	1	0	mise à 0
1	1	1	X	interdit

Quand H vaut 1, le bistable fonctionne normalement. Quand H vaut 0, il garde en mémoire l'état antérieur. On appelle ce circuit un bistable RSH. Il est synchronisé sur le niveau de l'horloge H. Il en existe une variante importante : le bistable DH (D pour delay, retard).

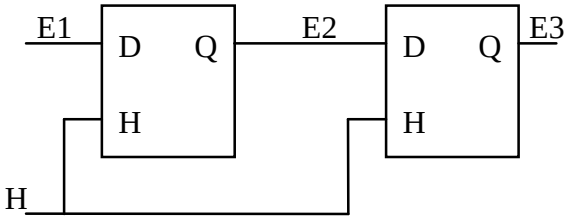


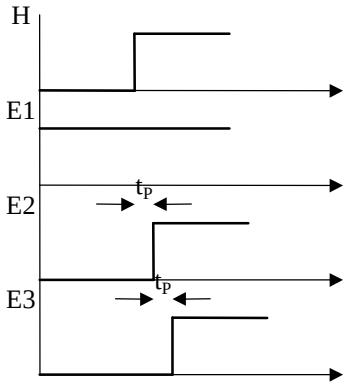
On pose S = D et R = $\overline{D}$ ce qui implique : D = 0 $\Rightarrow$ S = 0, R = 1 donc mise à 0 et D = 1 $\Rightarrow$ S = 1, R = 0 donc mise à 1. Ce bistable ne présente plus de combinaison interdite d'entrée car on a toujours R.S = 0. Il est appelé **latch transparent** et réalise la fonction $Q_{n+1} = \overline{H}.Q_n + H.D$:

- l'état copie, Q_{n+1}= D pour H = 1.
- l'état verrou, Q_{n+1}= Q_n pour H= 0.

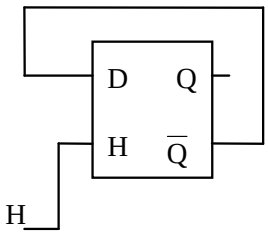
On n'exploite Q que pendant la phase verrou. C'est là, la principale limitation des bistables. En effet, les montages suivants sont interdits :

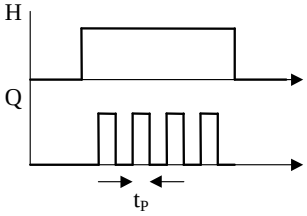
- Association synchrone en cascade. L'état à l'entrée de la chaîne se propage presque instantanément sur la sortie (après un temps de propagation t_p).





- Rétroaction. Le montage oscille.

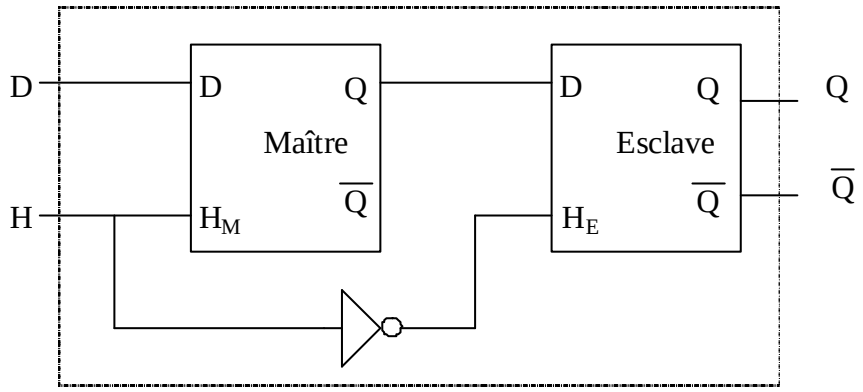




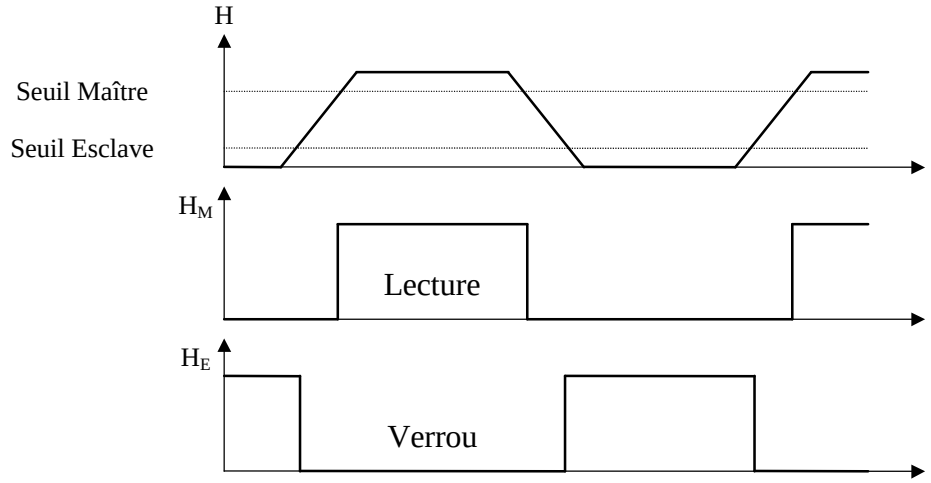
Les latches transparents sont toujours utilisés dans les ASIC et dans les FPGA, mais il est nécessaire de définir un nouveau circuit pour travailler en logique synchrone : c'est la bascule.

4.3 Bascules D maître-esclave (master-slave D flip-flop)

Une bascule est un bistable qui ne change d'état qu'une seule fois par période d'horloge. Le premier montage de ce type a été la bascule maître-esclave. Sa structure est la suivante :



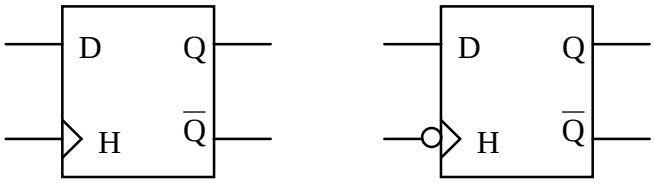
En réglant habilement les seuils de basculement d'horloge du bistable maître et du bistable esclave, on obtient le chronogramme suivant :



La commande D est lue sur le niveau haut de H et elle est exécutée sur son front descendant. C'est donc une sorte de bascule synchronisée sur front descendant. Ce type de bascule règle les deux problèmes vus au paragraphe précédent, l'association en cascade et la rétroaction. Toutefois, elle reste sensible aux parasites car la bascule maître accepte la commande pendant la totalité du niveau haut de l'horloge. On va donc définir un type de bascule qui n'accepte la commande que pendant le front actif de l'horloge. Cependant, le montage maitre-esclave est encore utilisé pour des raisons de processus de fabrication des circuits intégrés CMOS.

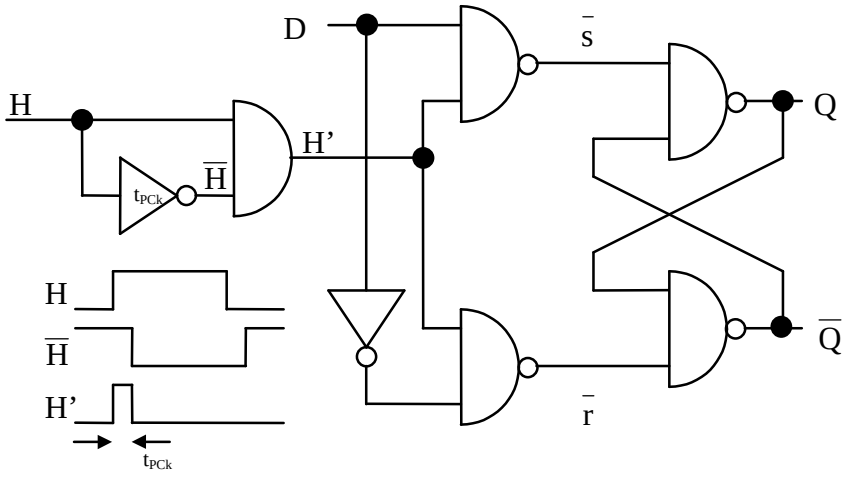
4.4 Bascules D synchronisées sur un front (edge-triggered D flip-flop)

Ces bascules font l'acquisition de la donnée et réalisent la commande sur un front d'horloge. Ce sont les bascules actuellement utilisées pour la conception en logique synchrone (avec les maître-esclave). Elles peuvent être actives sur le front descendant ou sur le front montant de l'horloge.

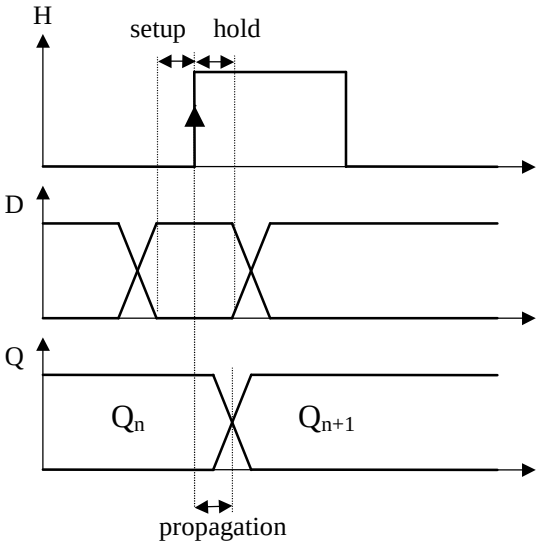


Front montant Front descendant

D'un point de vue purement théorique (il ne s'agit pas du montage utilisé réellement), on peut voir une bascule D commandée par un front montant de la manière suivante :



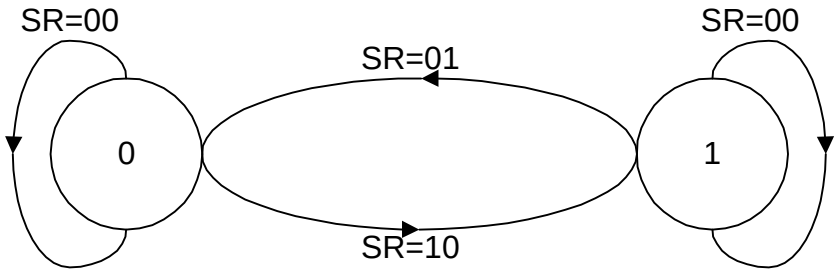
En dosant de manière adéquate le temps de propagation t_{PCK} de l'inverseur, on obtient en H' une impulsion juste assez large pour permettre le transfert de l'information D vers le latch RS. On voit bien avec ce montage que pour un fonctionnement correct de la bascule, la donnée doit être présente un certain temps (setup) avant le début de l'impulsion (le front actif) et rester stable un certain temps après (hold). Si la fenêtre de stabilité (setup+hold) n'est pas respectée, un phénomène de métastabilité apparaît (§4.6.4). La donnée n'apparaît sur la sortie Q qu'après un temps de propagation.



4.5 Bascules usuelles

Il existe quatre types de bascules usuelles :

- La bascule RS. Son équation de sortie est $Q_{n+1} = S + \overline{R}.Q_n$ avec S.R = 0. Son graphe d'évolution est :



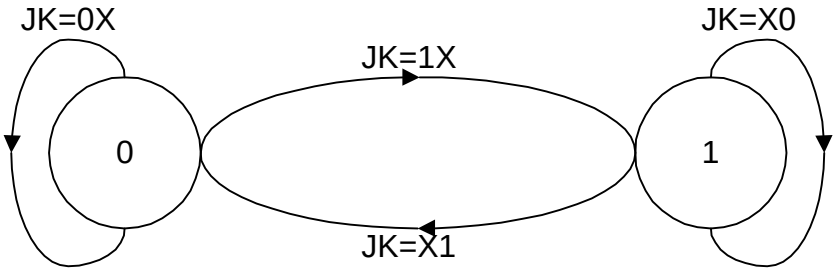
La bascule RS synchrone n'existe pas sous la forme de composant discret et n'est jamais utilisée directement dans un montage. Toutefois, sa structure se retrouve dans toutes les autres bascules. Sous la forme du latch RS, elle sert à réaliser des interrupteurs anti-rebonds (voir exercice 4.1). On obtient la table de vérité :

R	S	H	Q_{n+1}	fonction
0	0	↑	Q_n	mémoire
0	1	↑	1	mise à 1
1	0	↑	0	mise à 0
1	1	↑	X	interdit

- La bascule JK. Son équation de sortie vaut $Q_{n+1} = J.\overline{Q_n} + \overline{K}.Q_n$ ce qui donne la table de vérité :

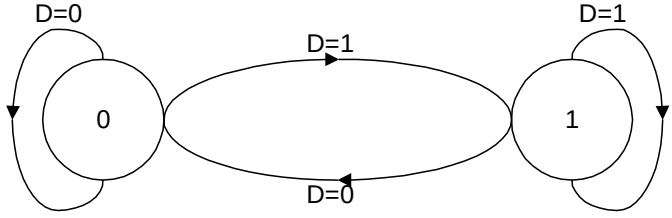
J	K	H	Q_{n+1}	fonction
0	0	↑	Q_n	mémoire
0	1	↑	0	mise à 0
1	0	↑	1	mise à 1
1	1	↑	$\overline{Q_n}$	$\overline{\text{mémoire}}$

Son graphe d'évolution est :



X est un état indifférent (don't care). Cette bascule permet la réalisation de montages ayant un minimum de portes combinatoires. Elle nécessite toutefois un câblage plus complexe qu'une bascule D car elle a deux entrées alors que la bascule D n'en a qu'une.

- La bascule D (**D flip-flop**). Elle est obtenue en posant D = S = $\overline{R}$ avec une bascule RS ou D = J = $\overline{K}$ avec une bascule JK, ce qui donne l'équation Q_{n+1} = D et le graphe d'évolution :



Sa table de vérité est :

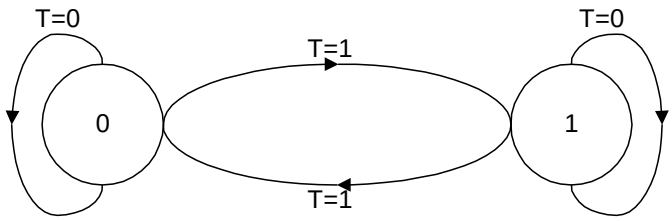
D	H	Q_{n+1}	fonction
0	↑	0	mise à 0
1	↑	1	mise à 1

Cette bascule existe sous forme de composant discret et permet la réalisation de montages nécessitant un minimum de câblage (car elle n'a qu'une entrée) mais plus de portes combinatoires qu'avec des bascules JK. Toutefois, comme le problème du câblage est très important dans les circuits intégrés récents (alors que les portes logiques sont quasiment gratuites), **la bascule D est la seule utilisée (avec le latch transparent)** dans les circuits programmables et dans les ASIC.

- On peut noter une variante de la bascule JK, la bascule T (**T flip-flop**) pour laquelle on pose J = K = T (T pour toggle). Son équation est $Q_{n+1} = T \oplus Q_n$ avec la table de vérité :

T	H	Q_{n+1}	fonction
0	↑	Q_n	mémoire
1	↑	$\overline{Q_n}$	$\overline{\text{mémoire}}$

Son graphe d'évolution est :



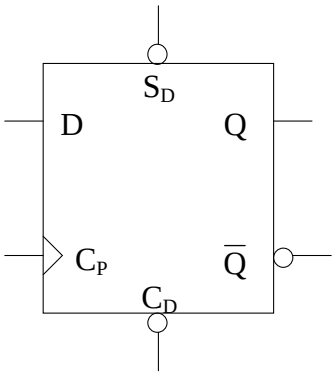
La bascule T n'existe pas sous forme de composant discret car elle est très facile à réaliser à partir d'une bascule JK ou d'une bascule D. Elle est particulièrement efficace dans la réalisation de compteurs binaires.

4.6 Caractéristiques temporelles des circuits séquentiels synchrones

Nous allons définir dans ce chapitre les intervalles de temps importants utilisés pour caractériser les circuits séquentiels synchrones.

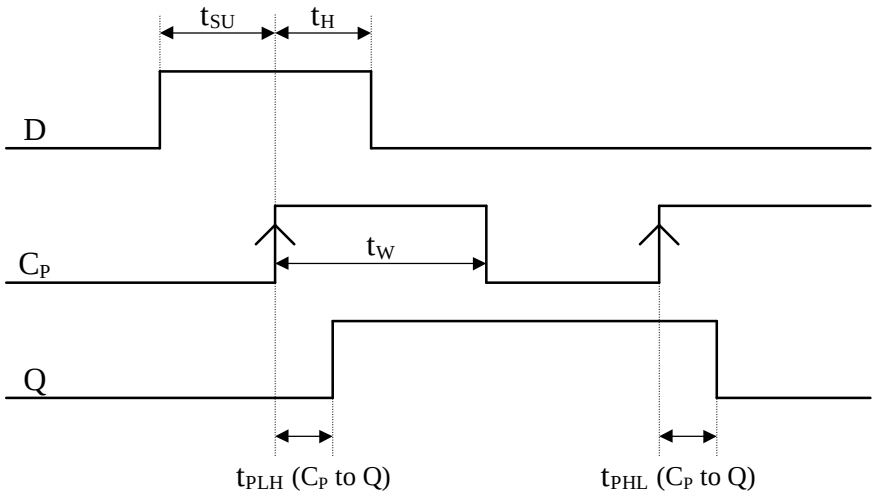
4.6.1 Définitions

Nous allons prendre l'exemple d'une bascule D de type SN74LS74 :

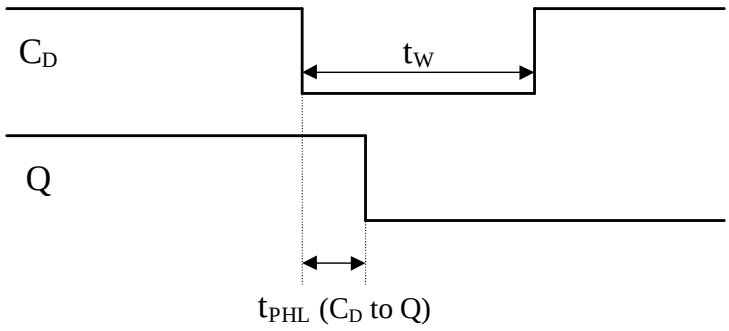


Les temps étudiés pour cette bascule se retrouveront (à quelques variantes près) dans pratiquement tous les autres circuits séquentiels. Les signaux à l'entrée d'un circuit séquentiel peuvent être classés en deux catégories :

- Les signaux à action synchrone. L'entrée D de la bascule est recopiée sur les sorties Q et $\overline{Q}$ après un temps de propagation t_{PLH} ou t_{PHL} au moment du front actif (ici le front montant) de l'horloge (notée C_K, C_P ou H). La donnée doit être présente sur l'entrée D un temps t_{SU} (setup time) avant le front actif et être maintenue un temps t_H (hold time) après ce front. L'impulsion active de l'horloge (ici l'impulsion positive) doit avoir une durée minimale t_W (width time) pour être prise en compte par la bascule.



- les signaux à action asynchrone. Les signaux de mise à 0 C_D (reset ou clear) et de mise à 1 S_D (set) ont une action immédiate et prioritaire sur toutes les autres commandes. Ils ne sont pas synchronisés sur le front actif de l'horloge. Ces signaux sont actifs sur un niveau (ici le niveau 0). Tant que le niveau actif est maintenu, l'effet de la commande persiste. Le niveau actif de l'impulsion doit avoir une durée minimale t_W. Le dessin suivant donne un exemple de chronogramme pour le signal clear. Les temps de transitions ne sont pas représentés.



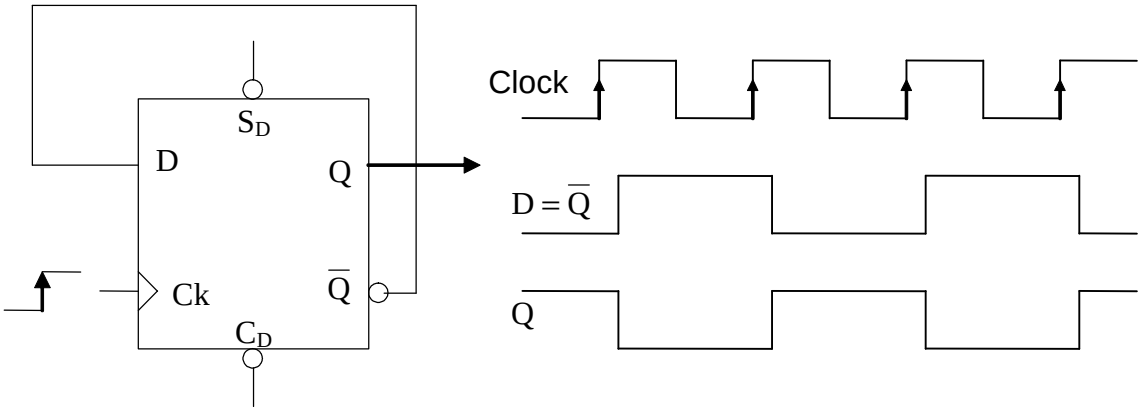
Afin de voir immédiatement comment agit un signal, il est pratique de respecter la notation suivante :

- le signal ayant un niveau actif à 0 est nommé $\overline{\text{SIGNAL}}$.
- le signal ayant un niveau actif à 1 est nommé SIGNAL.
- le signal réalisant deux fonctions (une sur chaque niveau) est nommé $\text{SIGNAL1}/\overline{\text{SIGNAL2}}$ (comme par exemple le signal de lecture/écriture $R/\overline{W}$ sur une mémoire).

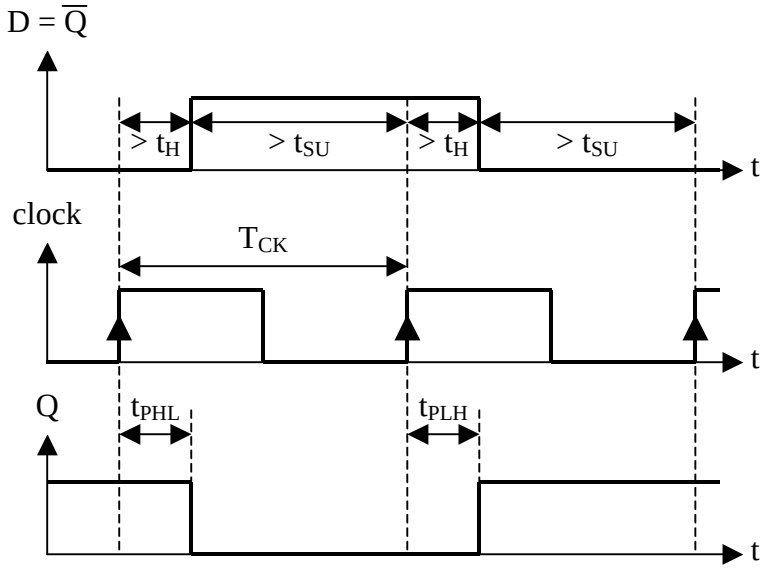
Remarque : comme nous l'avons vu pour la logique combinatoire, les temps t_{PLH} et t_{PHL} sont généralement égaux en technologie CMOS.

4.6.2 Calcul de la fréquence maximale d'horloge d'une bascule D

Il existe un autre paramètre utilisé pour caractériser un circuit séquentiel, la fréquence maximale de fonctionnement f_{MAX}. Ce paramètre est mesuré, dans le cas d'une bascule D, grâce au montage suivant :



Ce type de montage est appelé montage toggle car la sortie change d'état à chaque front actif d'horloge. La sortie Q est donc égale au signal d'horloge mais avec une fréquence divisée par 2. Agrandissons l'échelle des temps pour faire apparaître les différents timings de la bascule :



Pour que le montage fonctionne correctement, les paramètres du circuit doivent vérifier :

- $\mathbf{T_H < min(t_{PHL}, t_{PLH})}$. Cette relation ne dépend pas de la fréquence de l'horloge et elle est toujours vérifiée car on s'arrange pour que le temps de maintien de la bascule soit nul en fabrication. Cette relation n'intervient pas dans le calcul de la fréquence maximale du circuit.
- $\mathbf{T_{SU} < T_{CK} - max(t_{PHL}, t_{PLH})}$. Cette relation dépend de la fréquence d'horloge et elle permet de calculer la fréquence maximale du circuit. En effet, on remarque que pour un fonctionnement normal, la donnée D doit être présente t_{SU} avant le front montant de

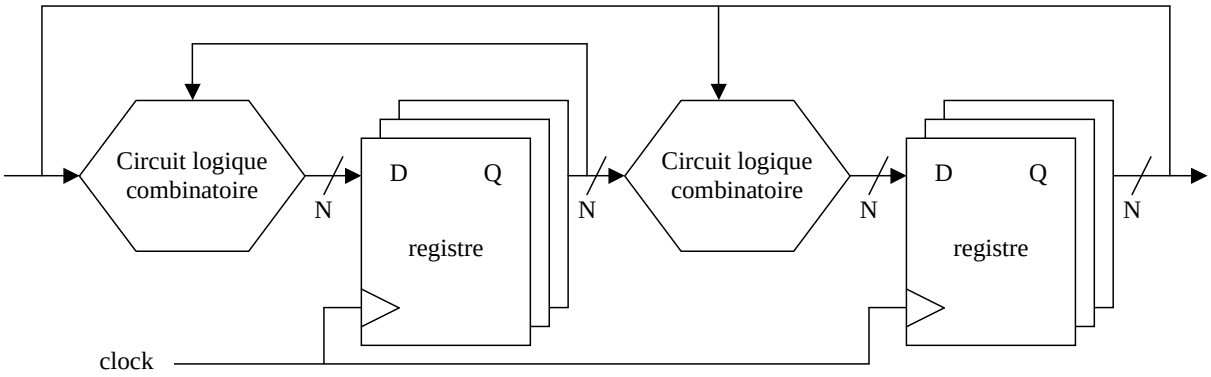
l'horloge. Or la donnée D est égale à la sortie $\overline{Q}$ qui apparaît un temps t_{PHL} ou t_{PLH} (C_P to $\overline{Q}$) après le front montant de l'horloge. La période de l'horloge T_{ck} ne peut donc être inférieure à $t_{SU} + t_P$ ou t_p est le plus grand des temps de propagation clock to Q. Ce qui donne la relation : $T_{CK} > T_{SU} + \max(t_{PHL}, t_{PLH})$. La fréquence maximale de fonctionnement de la bascule (toggle rate) est donc définie par :

$$f_{max} < \frac{1}{t_{SU} + \max(t_{PHL}, t_{PLH})}$$

Hélas, la valeur de f_{max} n'est pas toujours prise dans ces conditions dans les feuilles de caractéristiques (**data sheet**) et la plus grande prudence s'impose dans l'exploitation de cette donnée. Une lecture attentive des caractéristiques constructeurs est nécessaire mais pas toujours suffisante pour déterminer la fréquence maximale de fonctionnement d'un circuit séquentiel synchrone. Il faut aussi noter qu'en technologie CMOS, le temps de propagation de la bascule est le même à l'état haut et à l'état bas : $t_{PHL} = t_{PLH} = t_{p\ clock\ to\ Q}$.

4.6.3 Calcul de la fréquence maximale d'horloge dans le cas général

Dans le cas général, un circuit logique séquentiel synchrone peut toujours être mis sous la forme de registres (N bascules D fonctionnant en parallèle) reliées par de la logique combinatoire :



Certaines sorties de registres sont rebouclées sur des entrées. Il y a des entrées pures et des sorties pures. Dans un montage réel, le nombre de bascules peut s'élever à plusieurs dizaines ou centaines de milliers. Comment calcule-t-on la fréquence maximale de fonctionnement d'un tel « monstre ». En fait, c'est très simple. Il suffit de reprendre la formule vue

précédemment et de lui ajouter le temps de propagation dans la logique combinatoire (t_{prop}). On obtient donc :

$$f_{max} < \frac{1}{t_{SU} + t_{prop} + \max(t_{PHL}, t_{PLH})}$$

ce qui traduit le fait que la donnée doit être stable t_{su} avant le front actif suivant de l'horloge. Toute la question est : quel temps de propagation combinatoire doit-on choisir ? On ne peut pas le calculer à la main dans le cas général. Un outil de CAO va calculer tous les temps de propagation de toutes les sorties de bascules D vers toutes les entrées de bascules D. Le chemin le plus long, **le chemin critique**, va donner le temps le plus long, **le temps critique**. C'est ce temps qui va déterminer f_{max}.

$$f_{max} < \frac{1}{t_{SU} + t_{critique} + \max(t_{PHL}, t_{PLH})}$$

La fréquence la plus élevée d'un montage de ce type est égale à la fréquence maximale de fonctionnement d'une bascule D (qui ne dépend que de la technologie de fabrication utilisée pour construire la bascule). C'est le cas où le temps critique est nul, ce qui implique que le montage ne réalise pas grand-chose étant donné qu'il ne comporte pas de logique combinatoire.

La fréquence maximale d'un montage réel est donc déterminée par le temps critique qui est fonction de la complexité du montage réalisé et du talent de l'ingénieur qui conçoit le circuit. Il est important de noter que ce raisonnement ne vaut que pour un circuit logique séquentiel synchrone (il n'y a qu'une seule horloge connectée à toutes les bascules). C'est la première raison pour laquelle on n'utilise en pratique que ce type de montage dans les circuits logiques.

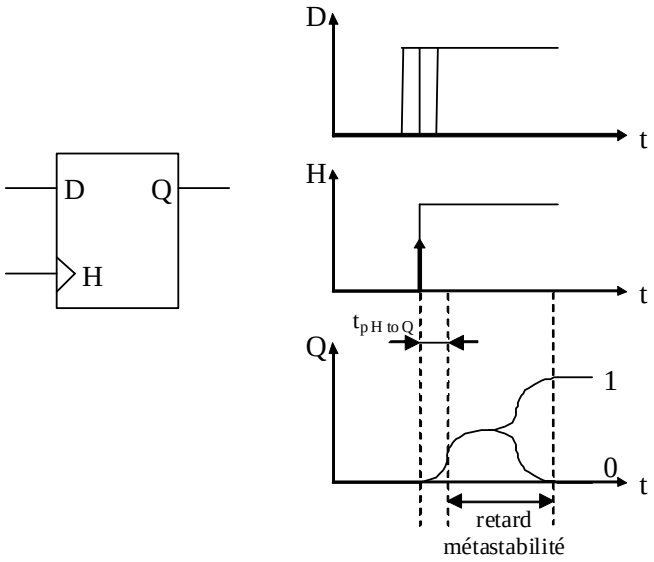
4.6.4 Métastabilité

Il reste un problème que nous avons soigneusement évité jusqu'à maintenant. Que se passe-t-il si le montage ne respecte pas le temps de setup, c'est-à-dire si la donnée n'est pas stable t_{su} avant le front actif de l'horloge ?

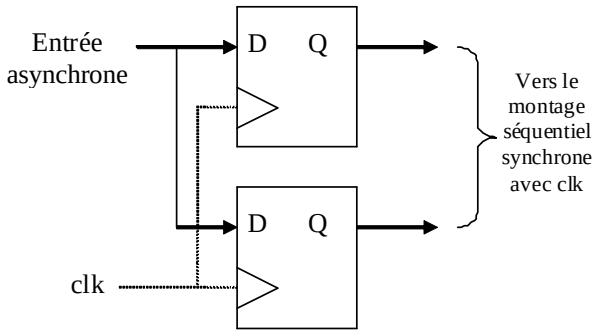
En fait, c'est un problème très courant que l'on peut rencontrer dans deux cas :

1. Dépassement de la fréquence maximale du montage. Il s'agit là d'une erreur d'utilisation du composant. La solution est évidente : il faut respecter la f_{max} du montage.
2. Entrées asynchrones. A partir du moment où le montage lit une donnée extérieure avec une bascule D, il y aura forcément une violation du temps de setup à un moment ou à un autre. Par exemple, dans le cas d'un bouton poussoir actionné par un opérateur humain ou bien d'un capteur qui indique un dépassement de trop plein ou encore d'une liaison série (RS-232) venant d'un ordinateur. On ne voit pas très bien comment un être humain pourrait être synchronisé avec l'horloge du montage.

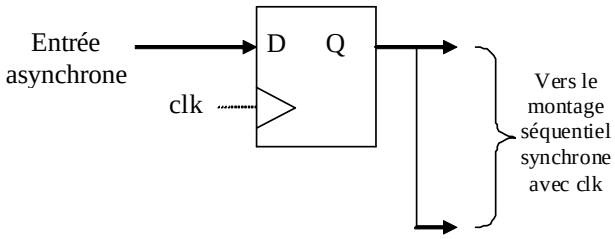
On peut faire beaucoup d'études savantes concernant le phénomène de métastabilité. Ce phénomène intervient quand la bascule hésite à changer d'état parce que la donnée n'est pas stable t_{su} avant le front actif de l'horloge. Elle se place alors dans un état intermédiaire entre le 0 et le 1, l'état « métastable ». En pratique, cela se traduit par un allongement du temps de propagation $t_{p\ clock\ to\ Q}$ de la bascule ; plus la violation du temps de setup est importante et plus le temps de propagation augmente. Bien sûr, si la violation est trop importante, le changement en sortie n'a pas lieu.



Dans le cas des entrées asynchrones, il est difficile de lutter contre la métastabilité car il s'agit d'un phénomène physique naturel inévitable. Il y a une erreur de base à ne jamais commettre, c'est la lecture multiple d'une entrée asynchrone :

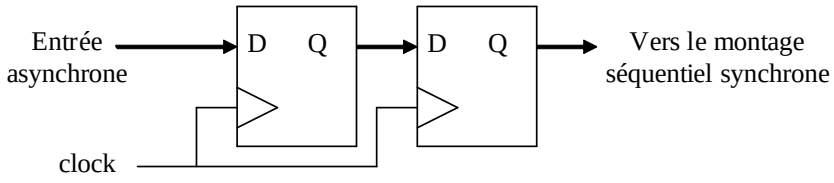


MAUVAIS : *une bascule peut lire correctement l'entrée et pas l'autre. On peut avoir deux versions différentes de l'entrée dans le design.*



BON : *une seule bascule lit l'entrée. On aura deux versions identiques de l'entrée dans le design.*

Une méthode simple et efficace pour lire une entrée asynchrone consiste à effectuer une double (voir triple) re-synchronisation. Cela permet de minimiser l'importance du phénomène.



Elle est efficace à condition que :

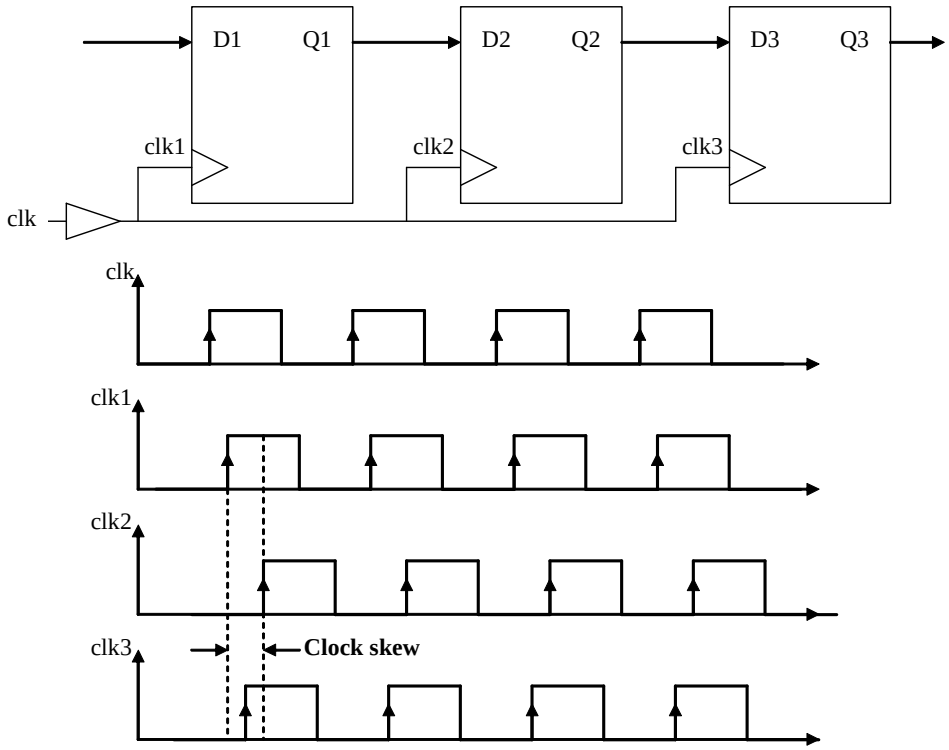
1. La fréquence de l'horloge f_{clock} ne soit pas trop élevée (< 100 MHz).
2. La fréquence de commutation de l'entrée $f_{entrée}$ soit très inférieure (10 fois) à f_{clock}.

Si ces conditions ne sont pas respectées, les solutions suivantes sont possibles pour lire des données asynchrones :

1. $f_{entrée}$ et f_{clock} sont du même ordre de grandeur, mais l'échange de données est assez lent. On utilise un protocole d'échange : le handshake.
2. Pour tous les autres cas, on utilise soit une mémoire double port, soit une FIFO.

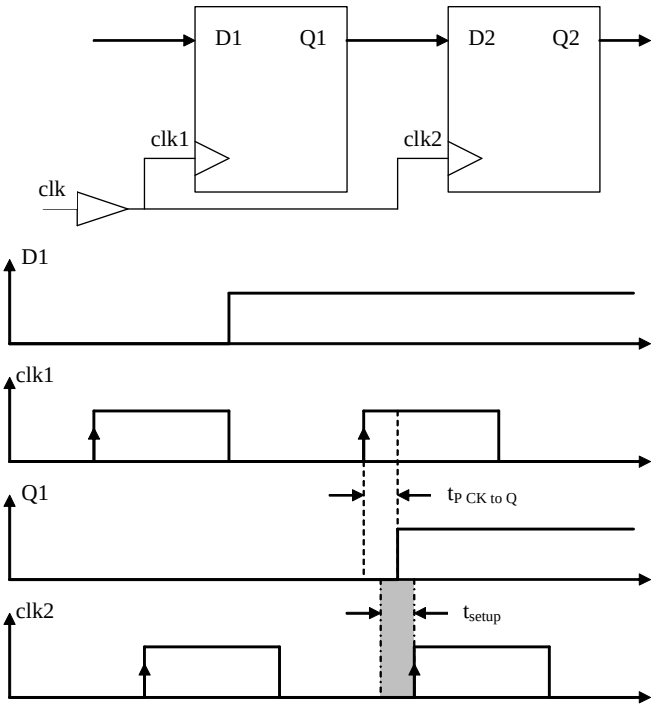
4.6.5 Distribution des horloges : « clock skew »

Le bon fonctionnement d'un montage séquentiel synchrone repose sur une hypothèse très importante : le front actif de l'horloge arrive au même moment sur toutes les bascules. En réalité, c'est impossible à réaliser (à cause de la longueur des interconnexions par exemple) et il existe une incertitude sur l'arrivée de ce front : c'est le « clock skew ».

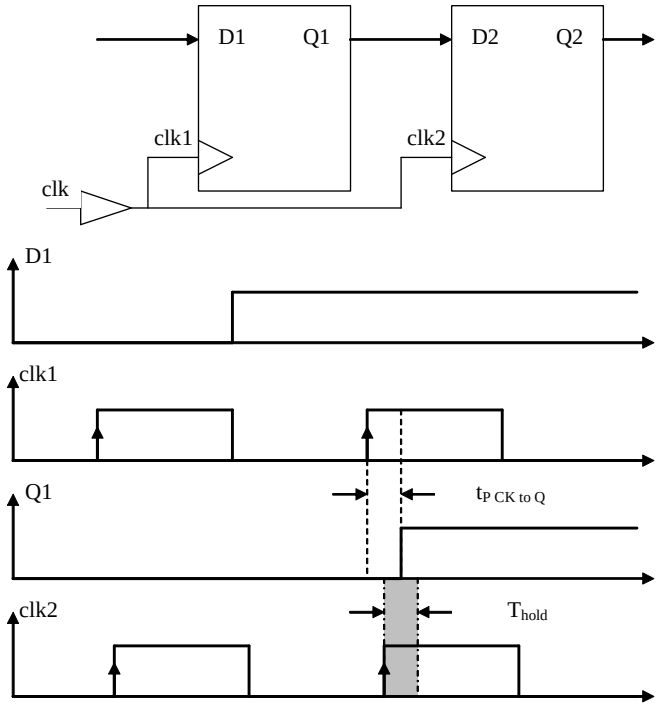


Heureusement, le clock skew ne concerne que les bascules adjacentes. Si les bascules affectées par ce phénomène sont séparées par au moins une autre bascule, il ne peut y avoir de problème. Quelles peuvent être les conséquences du clock skew sur deux bascules adjacentes ? Deux cas sont possibles suivant le sens du décalage :

- Si clk2 arrive juste après Q1, il y a violation du temps de setup de la deuxième bascule.



- Si clk2 arrive juste avant Q1, il y a violation du temps de hold de la deuxième bascule.



Cette violation du temps de hold ou du temps de setup est très gênante car elle ne peut être éliminée en diminuant la fréquence de fonctionnement du montage. Pour éliminer ce problème, il faut que le clock skew soit faible devant le temps de propagation de la bascule. Au sens strict, il faut que :

$$\text{retard de clk2 sur clk1} < t_{p\ \text{clock to Q}} - t_{\text{Hold}}$$

Si clk2 est un peu en avance sur clk1, il n'y pas de conséquence. En général, les différences d'heures d'arrivées entre les horloges sont dues aux délais de routage du circuit intégré (différence des longueurs des interconnexions). Les horloges doivent subir un traitement spécial pour compenser les retards à l'intérieur du circuit. Dans un ASIC, on crée pour cela un **circuit de distribution d'horloge** (ou arbre de distribution d'horloge).

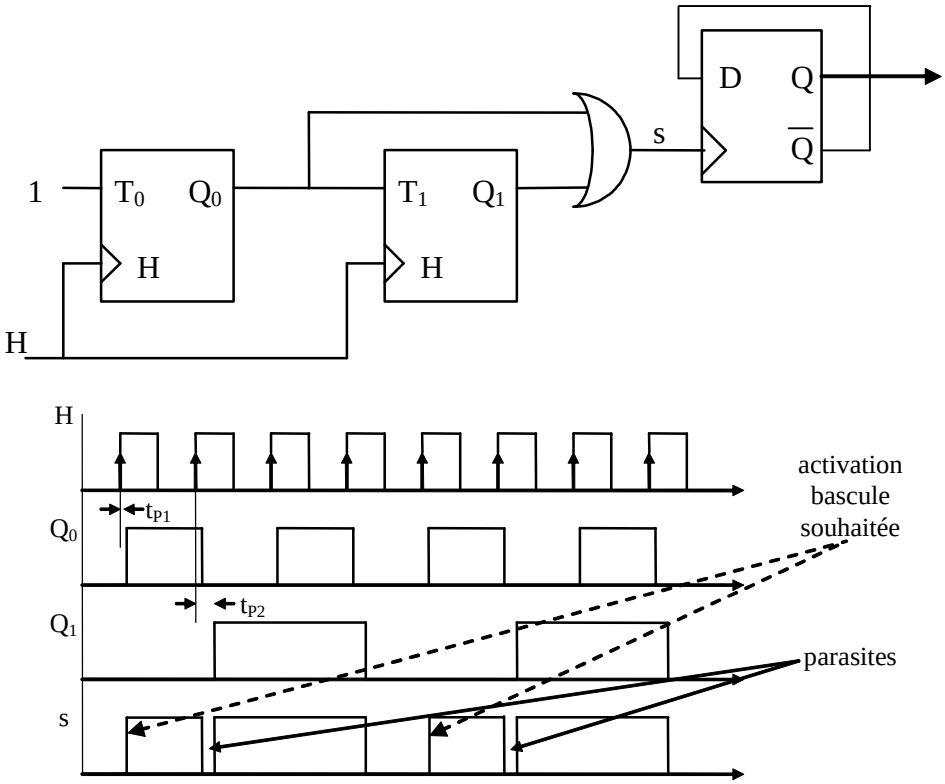
Dans un circuit logique programmable, l'arbre de distribution des horloges est déjà créé par le fabricant et il suffit de l'utiliser en connectant l'horloge à un buffer spécial (le buffer d'horloge, généralement BUFG chez Xilinx). L'utilisation de ce buffer vous garantit le bon fonctionnement de l'horloge, c'est-à-dire un « clock skew » qui ne gêne pas le bon

fonctionnement du montage. Un synthétiseur efficace reconnait les horloges d'un design et les connecte automatiquement au circuit de distribution d'horloge. C'est un point qu'il est préférable de vérifier soigneusement car si le synthétiseur n'utilise pas le buffer spécialisé pour l'horloge, celle-ci sera routée comme une donnée et le clock skew ainsi créé aura de bonnes chances de perturber le fonctionnement de votre montage d'une manière assez aléatoire (à cause des violations de temps de setup et de hold qui ont tendance à introduire des problèmes de métastabilité).

4.7 Règles de conception

4.7.1 Influence des aléas de commutation

Nous avons vu dans le chapitre sur la logique combinatoire que dans le cas général, il peut se produire un aléa de commutation (un glitch) en sortie de tout montage combinatoire sauf si celui-ci a fait l'objet d'une conception « hazard-free » extrêmement contraignante. Quelle va être l'influence de ce glitch dans un montage séquentiel ? Prenons un exemple simple :



Sur cet exemple, on voit bien l'impulsion parasite (le glitch) apparaître à la sortie de la porte OR. Il est dû à la différence de temps de propagation (clock to Q) entre les deux bascules T.

Sa durée est indépendante de la fréquence de fonctionnement et totalement dépendante du processus de fabrication des bascules. Certains couples de bascules ne provoqueront pas de parasites, d'autres le feront. Si le signal S est utilisé pour attaquer l'entrée d'horloge d'une bascule, le glitch risque d'être vu comme un front supplémentaire et d'actionner la bascule. La seule solution efficace pour éviter ce genre de problème est d'utiliser la logique séquentielle synchrone.

4.7.2 Règles de conception synchrone

Les circuits séquentiels synchrones sont constitués de bascules synchronisées sur les mêmes fronts d'**une seule horloge** séparées par des couches de logiques combinatoires. Toutes les commandes sont prises en compte sur le front actif de l'horloge et les « hazards », s'il y en a, sont forcément terminés sur le front actif suivant de l'horloge (si bien entendu le montage respecte la fréquence maximale de fonctionnement). D'autre part, il ne peut y avoir de « race condition » puisqu'il n'y a pas de rétroactions combinatoires directes mais seulement des rebouclages sur les entrées des bascules. Le fonctionnement d'un circuit en logique synchrone est donc extrêmement fiable et portable d'une famille technologique à l'autre. De plus, sa vitesse augmente linéairement avec l'accroissement de la vitesse des éléments qui le composent. L'amélioration de la densité d'intégration conduit donc automatiquement à une augmentation des performances.

Par construction, en logique séquentielle synchrone, les aléas de commutation peuvent être ignorés en toute sécurité car ils sont forcément terminés avant le front actif suivant de l'horloge. C'est une autre raison (avec le calcul de f_{max}) pour laquelle on n'utilise en pratique que ce type de montage dans les circuits logiques.

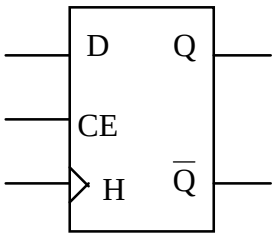
Toutefois, il est nécessaire pour éviter tout problème de respecter certaines règles :

1. Il ne faut pas attaquer une entrée d'horloge avec une combinaison de sorties de bascules. D'une manière plus générale, il faut traiter séparément le chemin des données et les lignes d'horloge.
2. Il ne faut pas utiliser les entrées asynchrones d'une bascule (entrée Clear et Preset par exemple) pour réaliser une réaction. Il est d'ailleurs conseillé de n'avoir que des entrées synchrones sur les bascules.

4.7.3 Le rôle du CE

Il reste toutefois un problème dont nous n'avons pas parlé. Si toutes les bascules sont activées par la même horloge, comment peut-on inhiber une bascule ou si vous préférez comment empêcher la copie de D sur Q sur le front actif de l'horloge ?

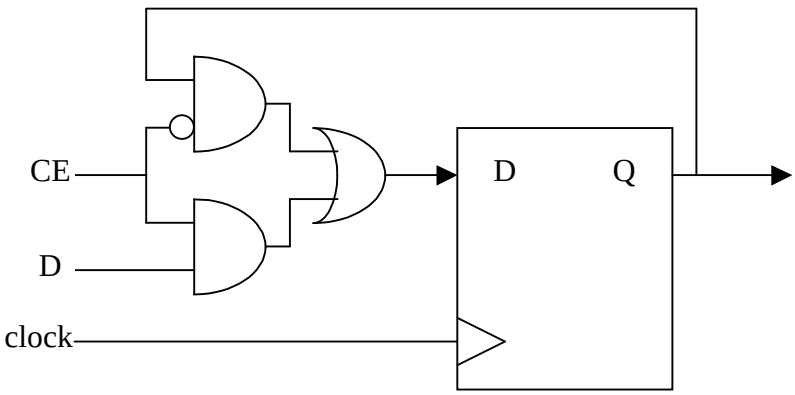
Il manque pour cela une entrée de validation sur nos bascules. C'est l'entrée CE ou « Chip Enable ». Sans cette broche supplémentaire, il est impossible de réaliser un montage séquentiel synchrone.



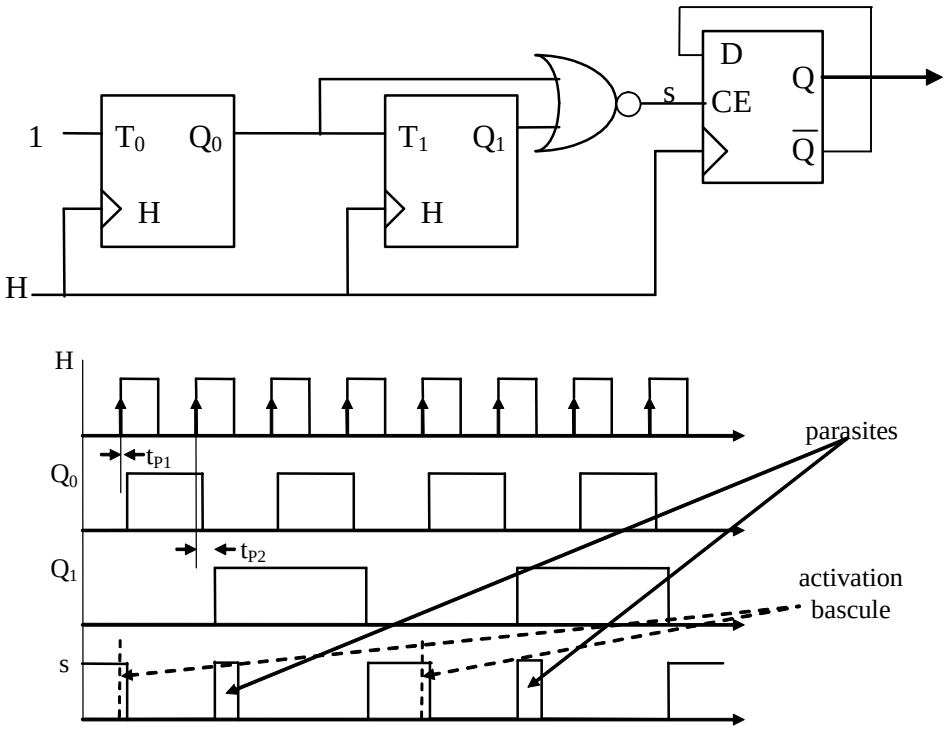
Son fonctionnement est très simple :

1. CE = 1 $\Rightarrow$ Q_{n+1} = D sur le front actif de l'horloge.
2. CE = 0 $\Rightarrow$ Q_{n+1} = Q_n sur le front actif de l'horloge.

En ce qui concerne la réalisation interne, il suffit d'ajouter un multiplexeur 2 vers 1 commandé par CE sur l'entrée D d'une bascule pour obtenir :



Nous pouvons maintenant reprendre et corriger le montage précédent :



L'équation logique de s a changé, mais la sortie du montage reste la même (vous pouvez vérifier que la bascule est activée au même moment). Les hazards sont toujours là, mais il ne nous gêne plus.

4.7.4 Asynchrone contre synchrone

En résumé, la logique séquentielle synchrone a les avantages suivants :

1. **Fiabilité** car les aléas de commutation n'interviennent pas dans le fonctionnement du montage.
2. **Calcul simple de la fréquence maximale de fonctionnement** du montage grâce à la formule suivante. Les outils de CAO savent calculer le temps critique du montage.

$$f_{max} < \frac{1}{t_{SU} + t_{critique} + \max(t_{PHL}, t_{PLH})}$$

3. **portabilité** d'une famille technologique à l'autre.
4. **La fréquence augmente linéairement** avec l'accroissement de la vitesse des éléments qui composent le montage. L'amélioration de la densité d'intégration conduit donc automatiquement à une augmentation des performances.

Comment définir la logique séquentielle asynchrone ? On la définit plutôt par opposition à la logique séquentielle synchrone. Tout montage qui ne respecte pas strictement les règles de conception synchrone est asynchrone. Pour le débutant, c'est hélas la solution qui vient en premier. Elle est généralement caractérisée par l'absence d'horloge maîtresse et/ou par la présence de rebouclage séquentiel asynchrone. Lorsqu'elle est utilisée dans les règles, c'est-à-dire lorsqu'elle est conçue correctement et de manière fiable (*c'est-à-dire quasiment jamais sauf par des ingénieurs très expérimentés : il faut savoir que les outils de CAO marchent très mal en asynchrone, ce qui ne facilite pas la conception*), elle a les avantages suivants :

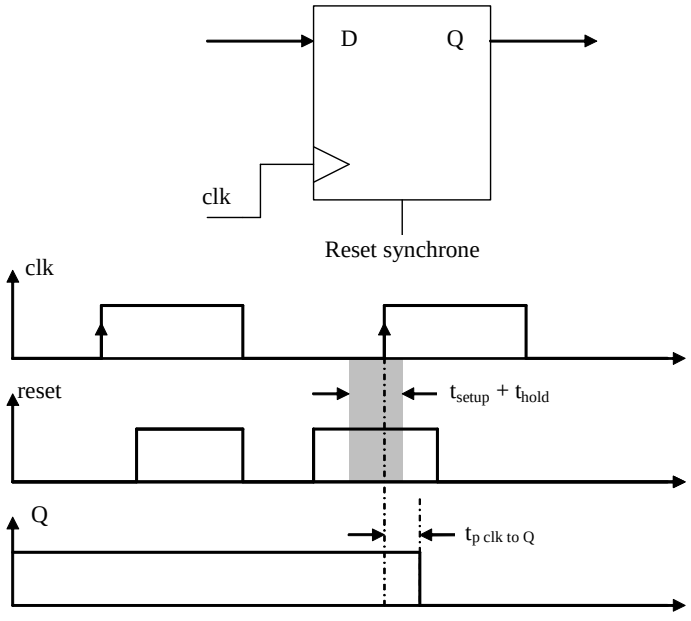
1. Fréquence de fonctionnement plus élevée qu'en logique synchrone ;
2. Consommation plus faible qu'en logique synchrone à fréquence de fonctionnement identique.

Pour toutes ces raisons, depuis le début des années 80, tous les montages correctement conçus utilisent la logique séquentielle synchrone.

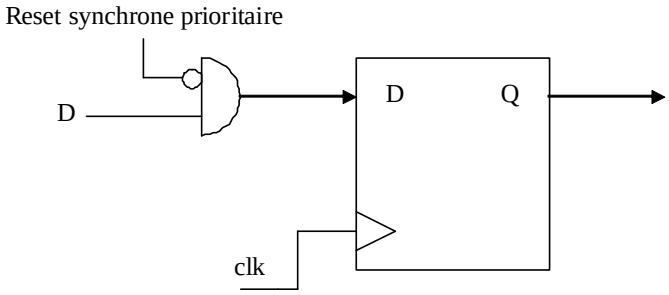
4.7.5 Le reset

Dans le cas général, il est important qu'un montage séquentiel synchrone démarre dans un état connu. On utilise généralement pour cela un signal de mise à 0 ou « reset » qui consiste à mettre à 0 toutes les sorties Q des bascules D du montage. En réalité, on devrait parler d'un signal d'initialisation qui peut charger indifféremment un 1 ou un 0 dans une bascule. Le signal reset existe sous deux formes :

- Synchrone (prioritaire sur D).

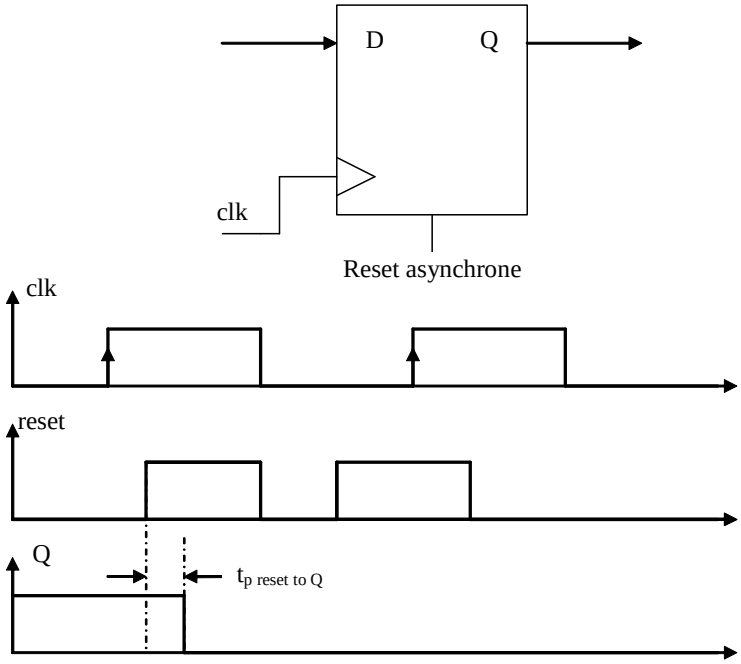


La réalisation d'une bascule D avec un reset synchrone ne pose pas de problème particulier. Il suffit d'ajouter un AND sur l'entrée D comme sur le schéma suivant.



N'importe quelle bascule peut être équipée d'un reset synchrone qui est vu comme une donnée et sera routé comme tel. Le chemin critique sera bien sûr allongé et la fréquence maximale de fonctionnement de la bascule baissera.

- Asynchrone (prioritaire sur toutes les autres entrées).



Le reset asynchrone doit faire partie du design de base de la bascule D, au moment de sa conception. Il est impossible de le rajouter après coup si la bascule n'a pas été conçue avec. Pour savoir s'il existe, notamment dans un circuit logique programmable de type EPLD ou FPGA, il faut étudier attentivement la documentation sur le fonctionnement des bascules.

Quel reset utiliser ? Au vu du paragraphe 4.7 sur les règles de conception synchrone, la question du choix du reset semble évidente. Il faut utiliser un reset synchrone qui sera considéré par l'outil de synthèse comme une donnée et qui sera analysé par l'analyseur de timing de la même manière. La seule conséquence prévisible, c'est plus de portes logiques combinatoires sur les entrées des bascules D du montage et donc plus de longueur de routage, c'est-à-dire une fréquence de fonctionnement plus faible. Cette réponse évidente est vraie pour les ASICs, mais moins évidente pour les FPGA/EPLD.

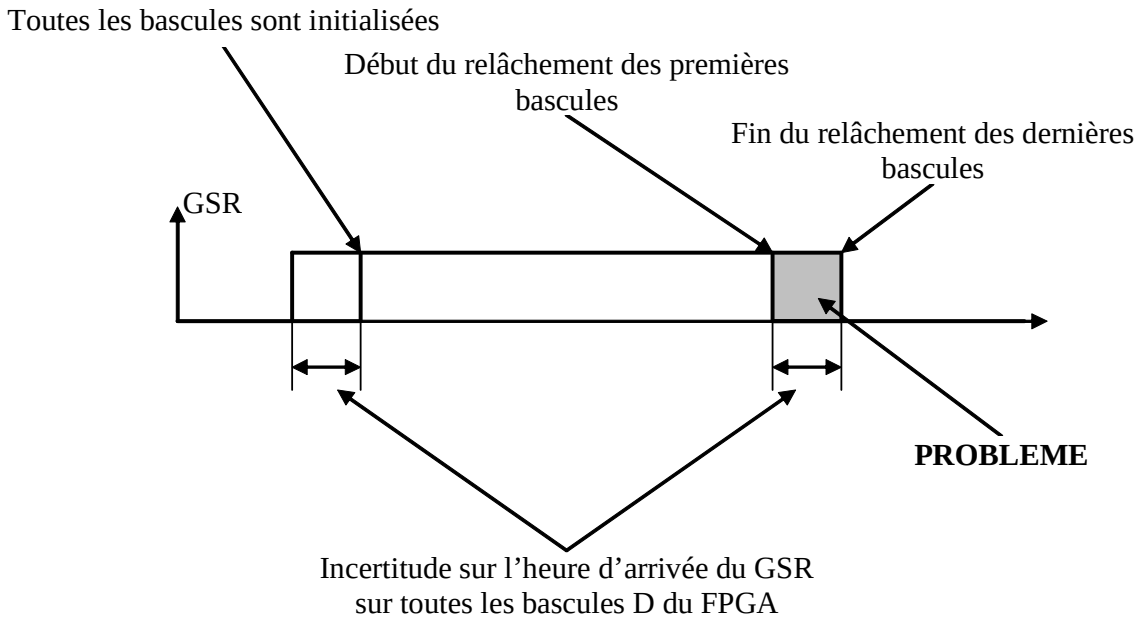
En effet, ces circuits subissent à la mise sous tension une configuration binaire qui est suivie généralement d'un reset asynchrone automatique. Il existe dans un FPGA (par exemple chez Xilinx) des ressources de routage dédiées qui sont reliées à tous les éléments de mémorisation du montage (y compris donc les bascules D). C'est le Global Set Reset (GSR) chez Xilinx. C'est l'activation du GSR à la mise sous tension qui initialise les bascules D et les blocs mémoires du design. Si vous ne spécifiez pas de valeurs dans le code VHDL, la valeur 0 est copiée par défaut dans tous ces éléments.

Vous pouvez utiliser le GSR dans votre design à condition d'utiliser un reset asynchrone. Le synthétiseur détecte tous les reset asynchrone du montage, les relie ensemble puis les connecte au GSR (à l'aide d'un composant Startup comme nous le verrons en TP). Cette ressource GSR qui existe dans le FPGA est gratuite et ne consomme ni porte logique ni ressources de routages utilisateur. Elle n'a donc pas d'influence sur la fréquence de fonctionnement du design. Evidemment, il s'agit d'un set/reset asynchrone qui ne peut pas être fonctionnel dans le design. Peut-on l'utiliser quand même ?

Il y a plusieurs aspects à considérer :

1. La simulation fonctionnelle/post-layout. Un signal de type std_logic non initialisé dans un design démarre à l'état U (Unknown = inconnu), ce qui reflète l'idée que l'état de sortie d'une bascule est indéterminé à la mise sous tension. Pour donner un état connu aux bascules D d'un montage, un reset asynchrone est largement suffisant. On active le reset au temps 0 pendant quelques dizaines de nanosecondes, on désactive puis on démarre les autres signaux tels que les horloges. Cela reflète l'activation du GSR à la mise sous tension du FPGA.

2. <u>Dans le FPGA.</u> Le problème vient du fait que le temps de propagation du GSR dans le circuit n'est nullement garanti. Il peut y avoir une grande incertitude (plusieurs ns) dans l'heure d'arrivée du signal sur les éléments de mémorisation du circuit. A l'activation du GSR, après un temps de propagation inconnu, tous les éléments sont initialisés. Jusque-là, tout va bien. Mais au moment où on désactive le GSR (donc à la fin du reset), toutes les bascules ne seront pas relâchées en même temps. Certaines parties du design vont démarrer avant les autres puisque généralement les horloges sont activées en permanence.



Cela peut poser un problème fonctionnel pour certains montages tels que les filtres récursifs ou bien les contrôleurs dans les machines d'états, en fait dans tous les montages où il y a rebouclage synchrone des sorties sur les entrées. Par contre, les montages sans rebouclage tel que les filtres non récursifs ne sont pas affectés par le problème car les fausses valeurs du démarrage sont éliminées rapidement par de nouveaux échantillons à traiter.

<u>Remarque</u> : il faut bien comprendre que le GSR sera activé dans tous les cas à la mise sous tension. Certaines personnes croient, à tort, que s'ils n'utilisent que des reset synchrones, ils n'ont pas à se préoccuper du GSR. Grossière erreur : si votre montage ne supporte pas un reset asynchrone, il ne démarrera pas correctement.

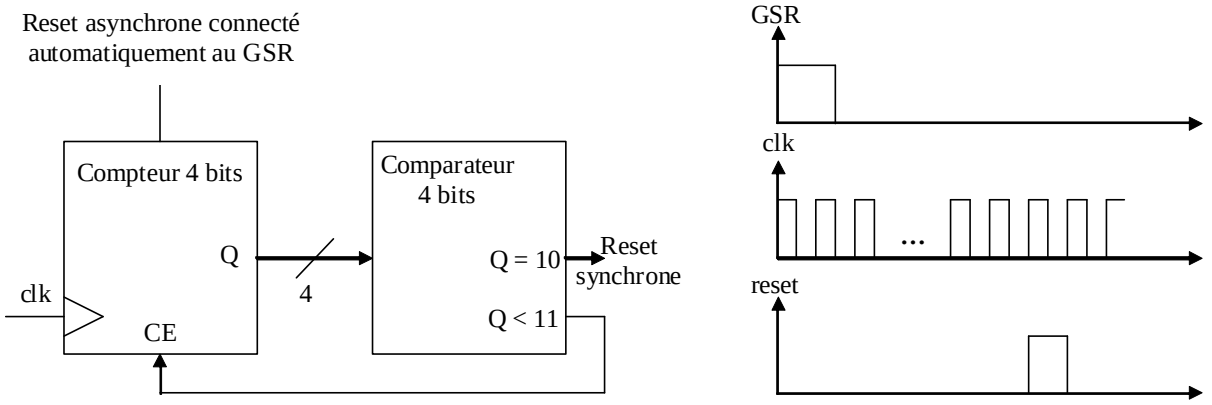
La règle finale est assez simple à énoncer, mais pas toujours évidente à respecter :

On peut utiliser un reset asynchrone dans un design à condition qu'il ne soit pas fonctionnel, c'est-à-dire que le montage marche quelques soient les valeurs des bascules au départ. Si nécessaire, on crée un reset synchrone pour les parties sensibles du design.

Il faut aussi savoir qu'il existe dans les FPGA de plus en plus de parties spécifiques qui ne fonctionnent qu'avec un reset synchrone. Si vous essayez d'utiliser ces composants associés à un reset asynchrone, le synthétiseur refuse de les inférer. Quelques exemples chez Xilinx :

- Registre 36 bits associé au multiplieur dans la Spartan-3,
- SRL16 dans les familles Spartan et Virtex,
- Bloc DSP dans la Virtex.

Comment créer ce reset synchrone fonctionnel ? Par exemple en utilisant un compteur qui est initialisé automatiquement par le GSR au démarrage et qui va compter jusqu'à N. Quand il atteint cette valeur, il délivre une impulsion pendant une période d'horloge (c'est le signal reset synchrone) puis il s'arrête de compter. Exemple avec N = 10 :

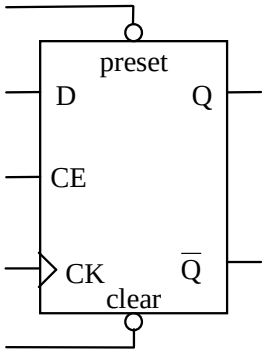
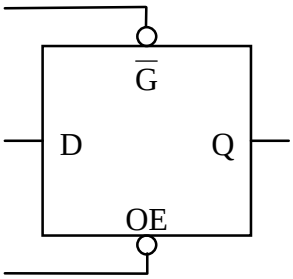


Nous n'aurons pas besoin d'utiliser ce montage dans ce cours car nos montages ne sont pas assez complexes. Nous utiliserons un reset asynchrone, ce qui permettra de simuler le fonctionnement correctement. Ce reset sera relié à un bouton poussoir sur la maquette. Le cas échéant, nous créerons un reset synchrone en fonction de nos besoins.

4.8 Circuits élémentaires

4.8.1 Les bascules élémentaires

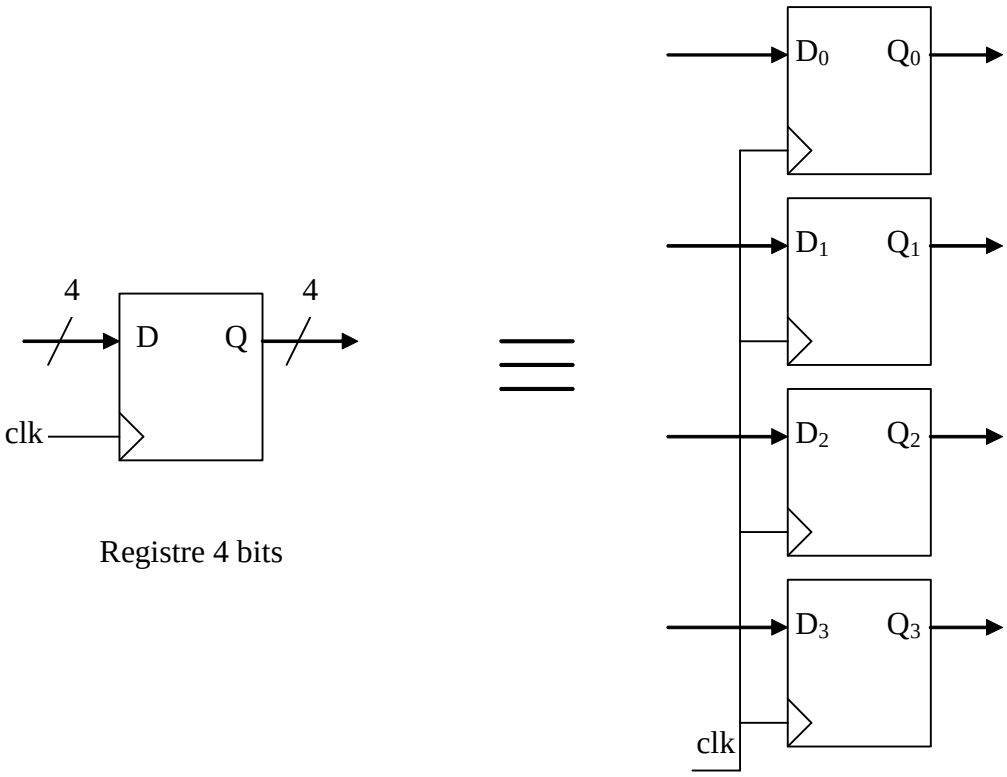
Parmi les bascules que nous avons étudiées précédemment, seuls les deux modèles suivants sont encore utilisés dans les ASICs et dans les circuits logiques programmables :

appellation	notation
D flip-flop synchrone sur front d'horloge	
Latch transparent	

On peut encore trouver la fonction 7474 en technologie CMOS dans un boitier 14 broches (Small Outline), mais elle est en voie de disparition rapide. La fonction latch n'existe plus sous forme discrète.

4.8.2 Le registre

Un registre est l'association en parallèle de plusieurs bascules D identiques. On parle par exemple d'un registre 16 bits (composé de 16 bascules D identiques reliées en parallèle). Un registre N bits peut être vu comme une bascule D qui traiterait des mots binaires de N bits. Il permet de stocker un mot de N bits. Voici un exemple de registre 4 bits :



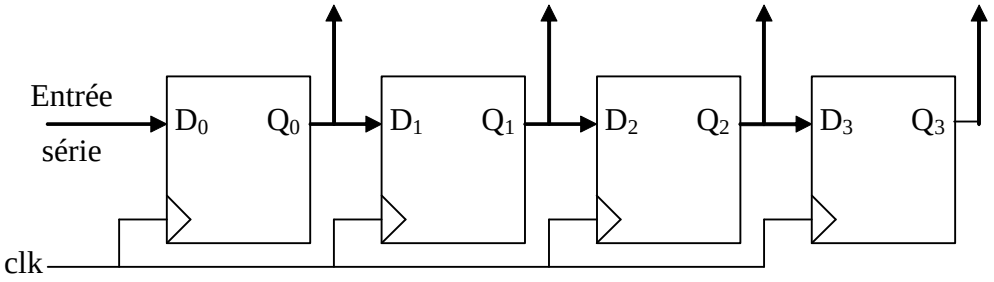
Registre 4 bits

Les caractéristiques temporelles du registre sont celles de la bascule D qui le compose. On trouve encore des registres en technologie CMOS sous forme discrète. Par exemple la fonction 74374, registre 8 bits, est toujours commercialisée en technologie CMOS 2,5 V et on en trouve sur toutes les cartes mères de PC.

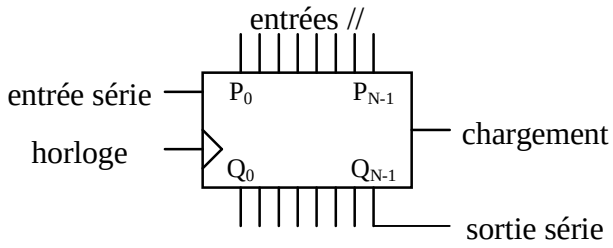
4.8.3 Le registre à décalage

4.8.3.1 Définition

Un registre à décalage est une association en cascade de bascules D permettant de décaler à gauche ou à droite une série de bits de données. L'exemple suivant montre un registre à décalage élémentaire sur 4 bits avec entrée série et sorties parallèles :



Dans un modèle plus élaboré, on peut trouver une entrée de données en série, N entrées de chargement en parallèle et N sorties. La Nième sortie peut servir de sortie série.



Il y a trois principaux modes d'utilisation :

1. L'entrée série avec sortie parallèle. Prenons un exemple avec un registre 4 bits et voyons l'évolution des données en sortie. A chaque coup d'horloge, la donnée sur l'entrée série est copiée sur l'étage 0 et la donnée se trouvant à l'étage i-1 est transférée à l'étage i. Ainsi, les données rentrées en série sont mises en parallèle sur les sorties.

entrée série	Ck	Q0	Q1	Q2	Q3
b0	↑	b0	0	0	0
b1	↑	b1	b0	0	0
b2	↑	b2	b1	b0	0
b3	↑	b3	b2	b1	b0
b4	↑	b4	b3	b2	b1

2. L'entrée parallèle avec sortie série. Prenons un exemple avec un registre 4 bits et voyons l'évolution des données en sortie. Sur le front actif de l'horloge avec l'entrée de chargement à 1, les données parallèles sont copiées sur les sorties. Ensuite, à chaque coup d'horloge, la donnée se trouvant à l'étage i-1 est transférée à l'étage i et un 0 est copié sur l'étage 0. On voit apparaître sur la sortie série Q3 les données parallèles mises en série.

chargement	Ck	Q0	Q1	Q2	Q3
1	↑	P0	P1	P2	P3
0	↑	0	P0	P1	P2
0	↑	0	0	P0	P1
0	↑	0	0	0	P0

3. L’entrée série avec sortie série. On reprend le tableau du mode 1, mais on considère les données sortant sur Q3. Ce sont les données de l’entrée série apparaissant après un retard de 4 coups d’horloge.

4.8.3.2 Applications

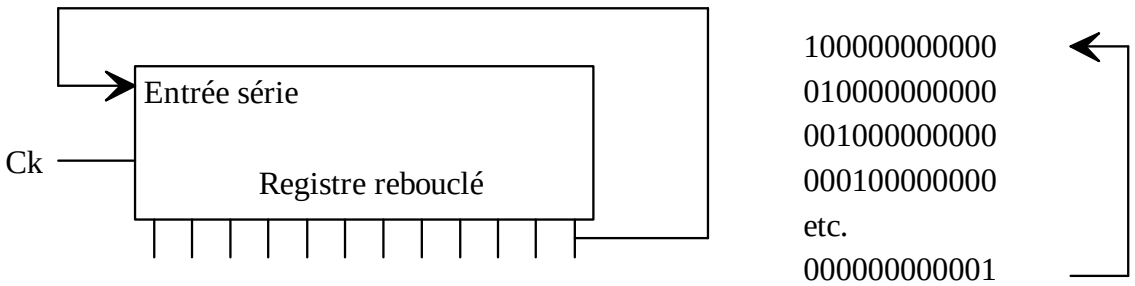
Les registres à décalage ont de nombreuses applications. Parmi celles-ci, on trouve :

- La conversion de données série-parallèle utilisée pour la transmission de données. On prend des bits en série et on les convertit (généralement sur un octet) en parallèle.

- La conversion de données parallèle-série utilisée pour la transmission de données. On prend des données binaires (généralement un octet) en parallèle et on les convertit en un train binaire série.

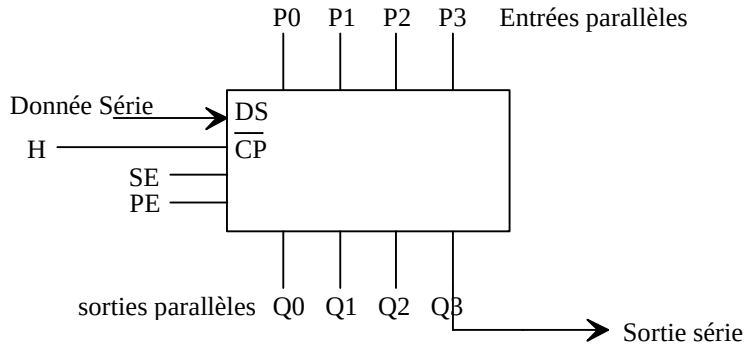
- Les opérations arithmétiques pour réaliser des multiplications et divisions (opérations utilisant le décalage). On entre et on sort alors en parallèle.

- Les compteurs en anneau (entrée série, sortie série). Il faut effectuer un chargement initial, puis à chaque coup d'horloge le contenu se décale en suivant une permutation circulaire. On peut réaliser des générateurs d'horloges décalées (code Johnson), des chenillards, ...



4.8.3.3 Le SN74LS178, registre à décalage polyvalent de 4 bits

Finissons ce paragraphe avec un exemple de registre à décalage du TTL data book, le SN74LS178. Il n’y a plus de registre à décalage de ce type commercialisé, mais la fonction réalisée est intéressante à étudier. C'est un registre à décalage à entrées parallèles, sorties parallèles, entrée et sortie série.



Le plus souvent, dans les registres à décalage, les bits à gauche ont les poids les plus faibles. Cette notation peut être gênante, mais c'est celle des circuits que l'on trouvait dans le commerce. Les signaux SE et PE sélectionnent 3 types de fonctionnement possibles :

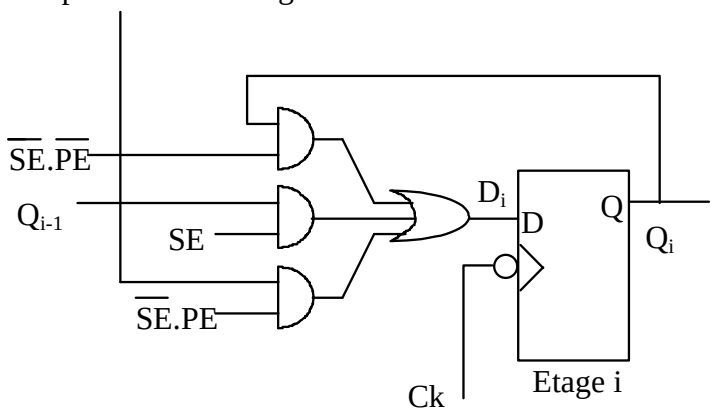
SE	PE	$\overline{CP}$	Action
H	X	↓	décalage : DS → Q0 → Q1 → Q2 → Q3
L	H	↓	chargement : Pi → Qi
L	L	X	pas d'action

Le registre est formé de 4 bascules D. D'après le tableau ci-dessus, on peut écrire immédiatement l'équation au niveau de l'ième étage :

$$D_i = SE.Q_{i-1} + \overline{SE}.PE.P_i + Q_i.\overline{SE}.\overline{PE}$$

D'où le schéma suivant pour l'étage i :

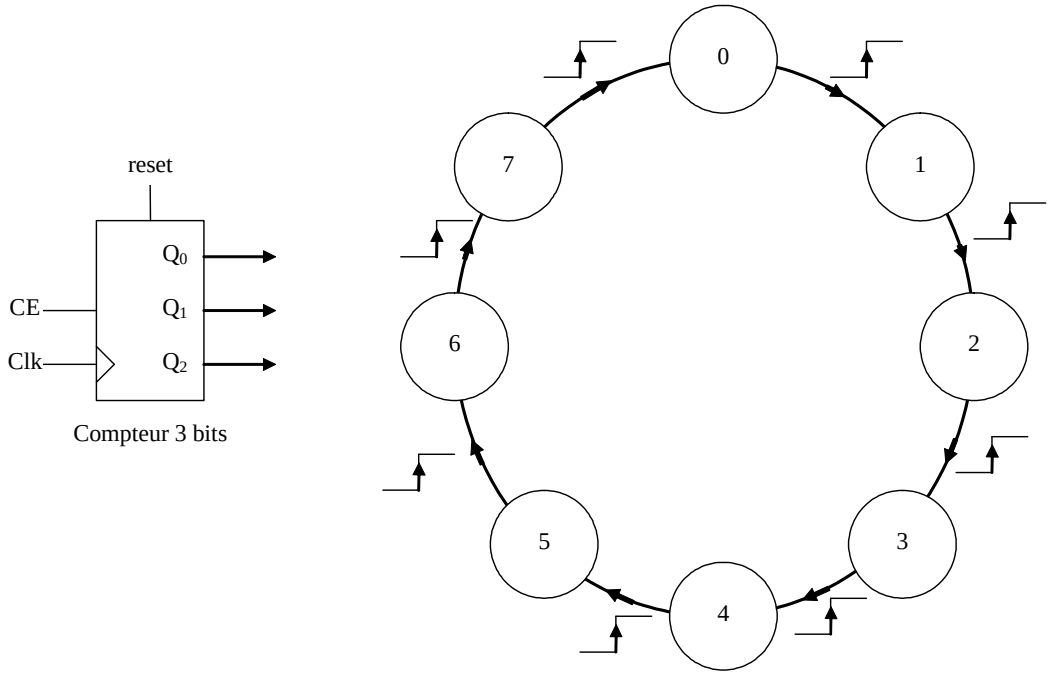
Pi : Entrée parallèle de l' étage



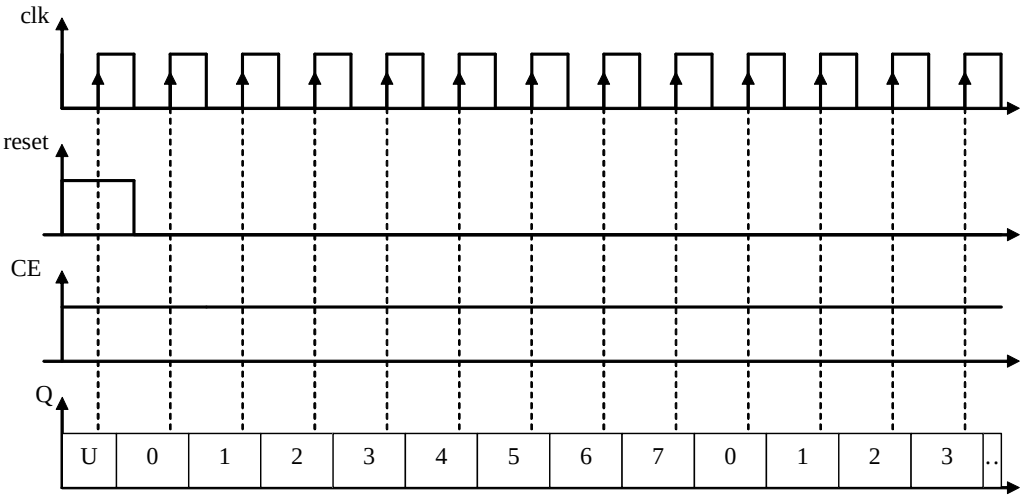
4.8.4 Les compteurs

4.8.4.1 Introduction

La fonction comptage est l'une des plus importantes de l'électronique numérique. Rares sont les designs qui n'en comportent pas un grand nombre. Le principe est simple : à chaque coup d'horloge, la sortie d'un compteur est incrémentée. La séquence de sortie pour un compteur N bits va donc évoluer de 0 à 2^{N-1}, puis repasse à 0 et reprend le cycle. L'exemple suivant montre le graphe d'évolution d'un compteur 3 bits :



Le chronogramme réalisé est donc le suivant :



Il existe deux familles de compteurs :

1. les compteurs synchrones : toutes les sorties changent d'état un temps $t_{P\ clock\ to\ Q}$ après le front actif de l'horloge.
2. les compteurs asynchrones : la sortie N change d'état un temps $t_{P\ clock\ to\ Q}$ après la sortie N-1, la sortie 0 change d'état un temps $t_{P\ clock\ to\ Q}$ après le front actif de l'horloge (aucune sortie ne change en même temps que les autres).

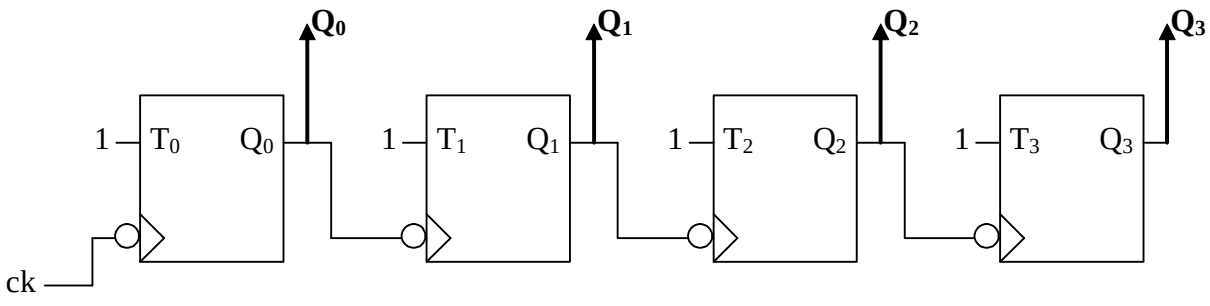
On utilise principalement des compteurs binaires N bits (modulo 2^N) ou décimaux (par décades). On peut aussi réaliser des compteurs de type Gray (à partir d'un compteur binaire, voir §3.1.4.2), one hot ou Johnson (avec un registre à décalage). Ces compteurs peuvent posséder des fonctions supplémentaires telles que :

- La remise à zéro de l'état du compteur.
- Le chargement parallèle. Cette entrée permet de charger en parallèle une valeur dans le compteur, cette valeur devant être présente sur les entrées de chargement parallèle.
- Le décomptage. Au lieu d'incrémenter la sortie du compteur à chaque front d'horloge, on la décrémente. La séquence devient donc (pour un compteur binaire) : 2^{N-1}, 2^{N-2}, …, 0, 2^{N-1}, 2^{N-2}, …
- la retenue (aussi appelée CEO pour Chip Enable Output ou encore RCO pour Ripple Carry Output) qui indique le dépassement de capacité et sert à mettre les compteurs en cascade.
- L'entrée de validation CE pour autoriser le comptage ou mémoriser la valeur précédente.

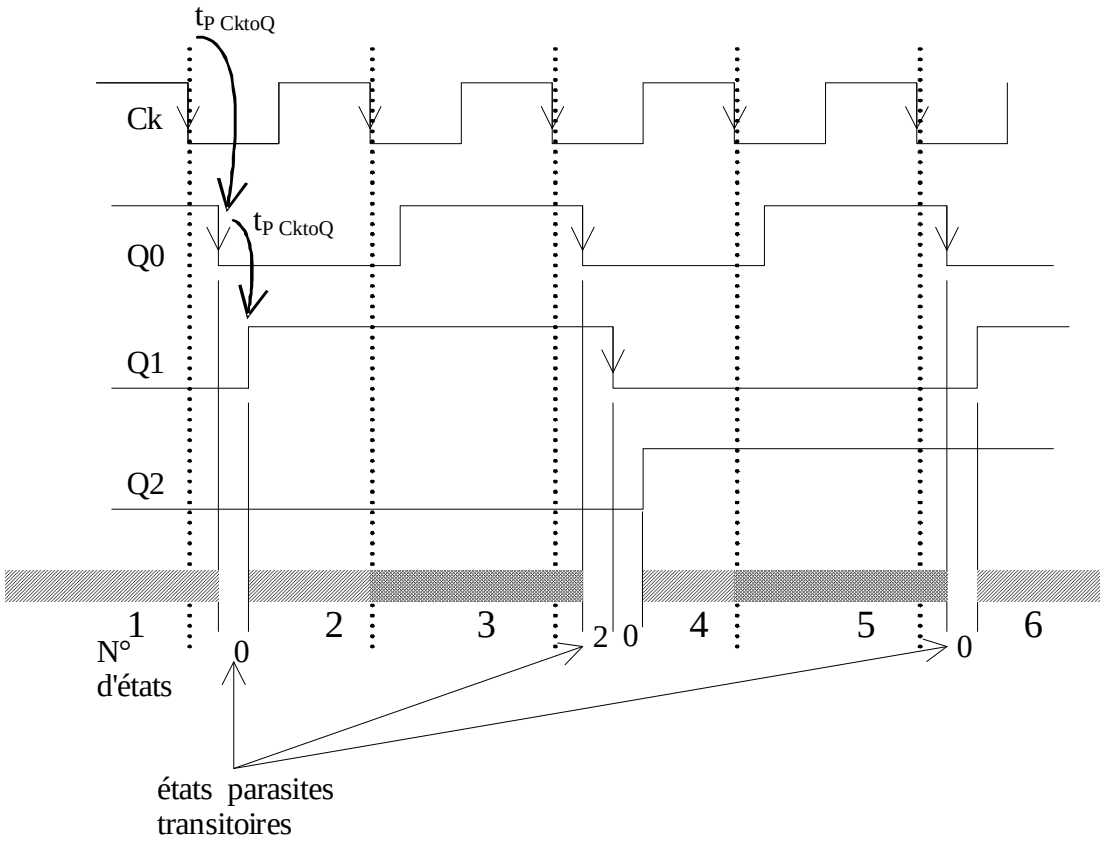
Les entrées peuvent être actives à 0 ou à 1 et être à action synchrone ou asynchrone.

4.8.4.2 Compteurs binaires asynchrones et synchrones

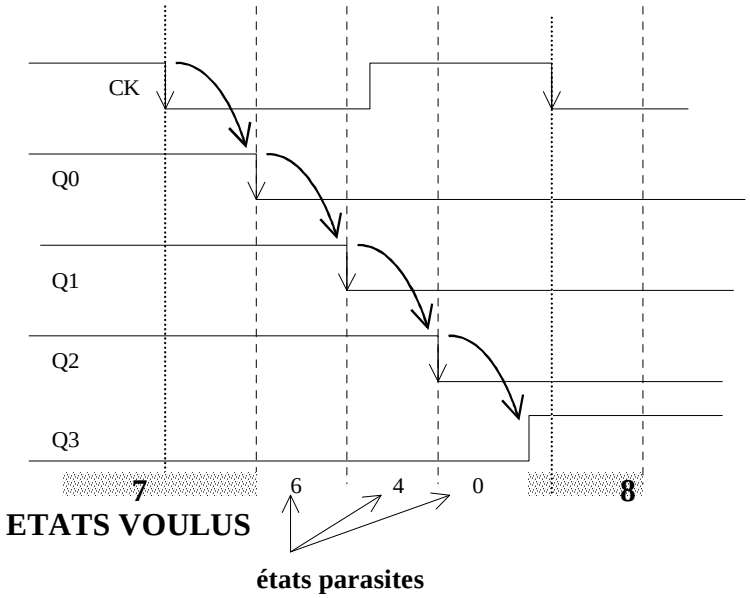
Les compteurs asynchrones, qui ne sont plus commercialisés, peuvent toujours être utilisés comme fonction dans un circuit logique. Leur principe de fonctionnement est le suivant :



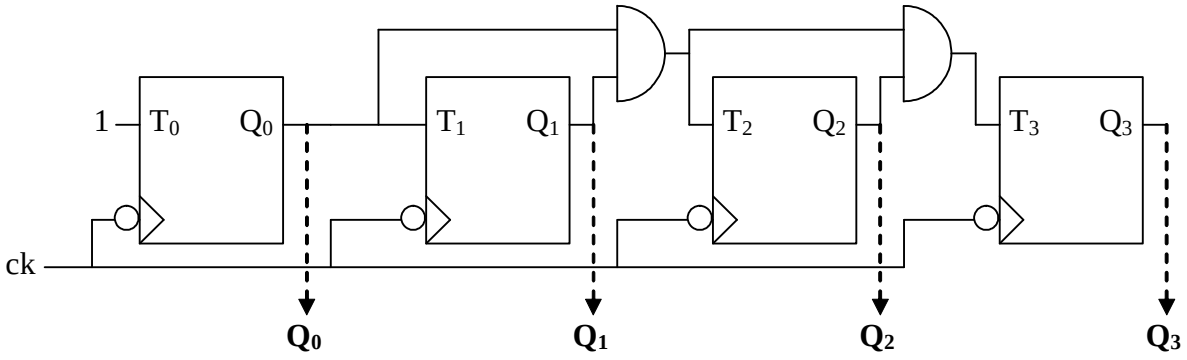
Cet exemple réalise un comptage de 0 à 15 puis retour à 0 puisque, pour chaque bascule T, $Q_{n+1} = \overline{Q_n}$. Les temps de propagation $t_{P\ Ck\ to\ Q}$ s'ajoutant, un dessin en haute fréquence montre des états transitoires de commutation :



Si la fréquence augmente, les états transitoires peuvent durer plus longtemps que les états réels du compteur. Toutefois, si on ne réalise qu'une division de fréquence, ils ne sont pas du tout gênants, et on obtient alors la structure de diviseur de fréquence la plus rapide car la fréquence maximale du compteur est égale à la fréquence maximale de la bascule de poids faible. De plus, ce montage consomme moins que son équivalent synchrone car chaque bascule fonctionne à la fréquence minimale possible pour réaliser la fonction. Par contre, on ne peut exploiter les états de sortie du compteur que quand ils sont stabilisés, ce qui en limite fortement l'usage. Il faudra compter sur un front et échantillonner les sorties par une bascule D active sur l'autre front (dans le cas où la somme des $t_{P\ ck\ to\ Q}$ ne dépasse pas la demi-période de l'horloge). Malgré cela, on atteint très rapidement les limites utilisables. Par exemple, si on augmente trop la fréquence d'un compteur 4 bits, on peut obtenir le chronogramme suivant :



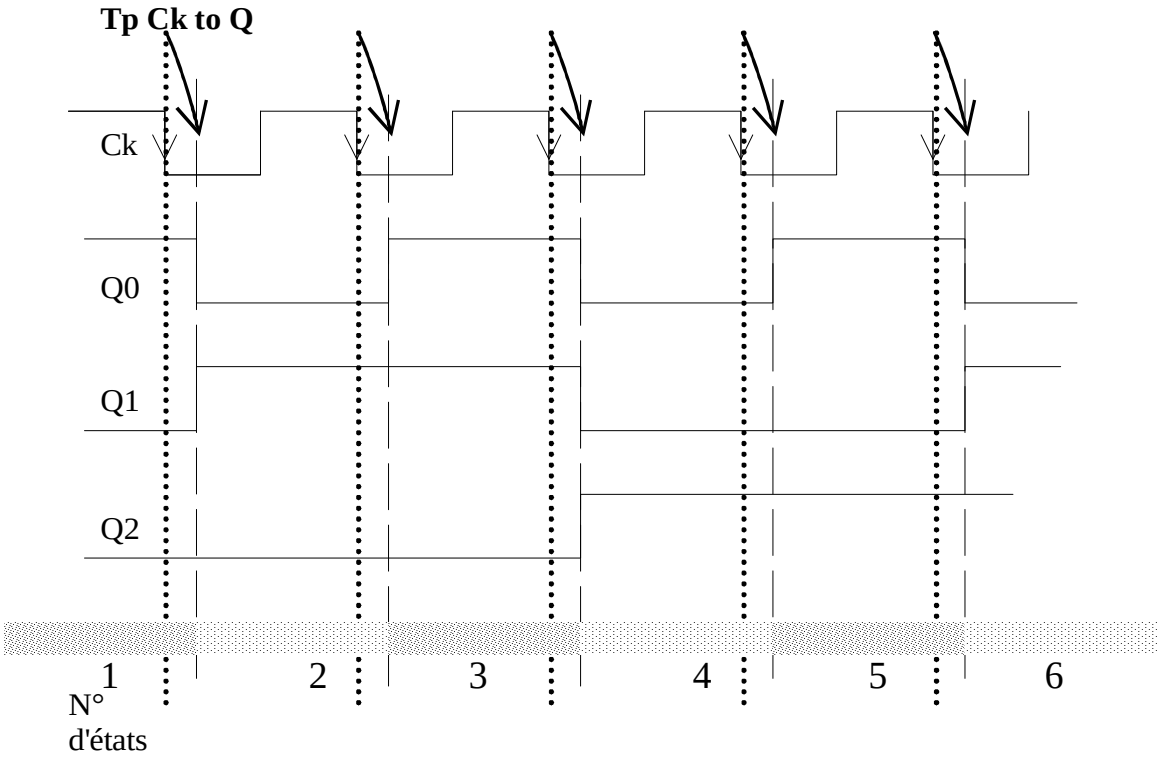
Comme il est impossible d'exploiter les états de sortie de ce compteur à cette fréquence, il faut passer en logique synchrone avec un schéma du type :



Toutes les sorties du compteur changent un temps $t_{P\ ck\ to\ Q}$ après le front actif de l'horloge. Il n'y a plus d'états parasites transitoires. Par contre, pour une simple division de fréquence, ils sont moins performants que les compteurs asynchrones car la fréquence maximale d'utilisation est plus faible :

$$f_{max} = \frac{1}{t_{P\,Ck\,to\,Q} + t_{setup} + t_{P\,logique\ combinatoire}}$$

Le chronogramme de sortie est donc :

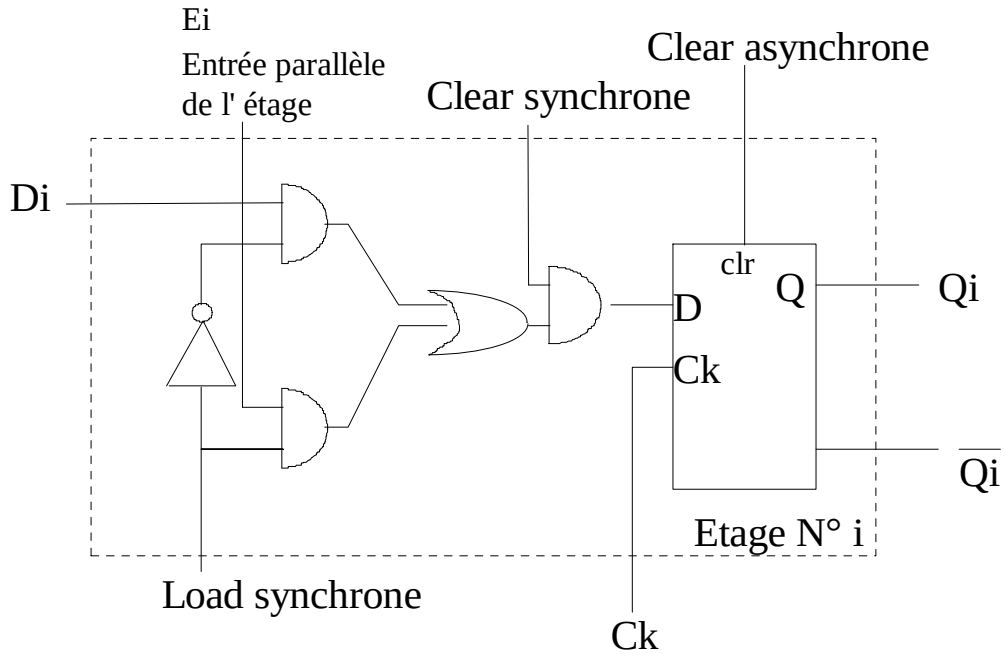


Ces deux montages montrent bien les différences entre logique synchrone et asynchrone. Le tableau suivant résume leurs caractéristiques :

type	fréquence maximale	consommation avec N étages	exploitation des sorties	utilisation
asynchrone	f_{max} bascule T	1 + ½ + ¼ + … = 2	très difficile en général	diviseur d'horloge
synchrone	$\frac{1}{t_{P\ Ck\ to\ Q} + t_{setup} + t_{P\ AND}}$	1 + 1 + 1 + … = N	après le temps $t_{p\ ck\ to\ Q}$	universelle

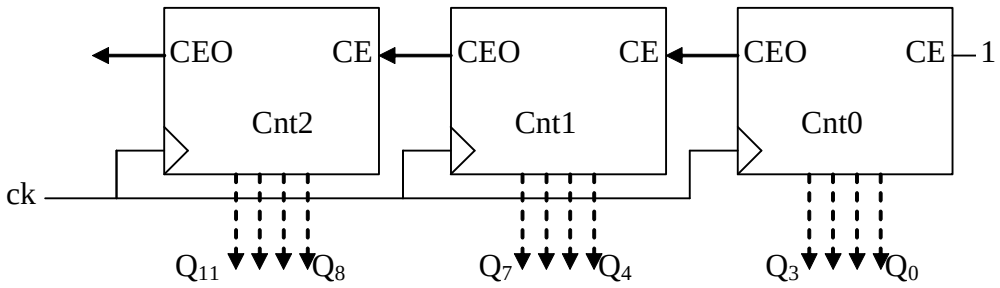
4.8.4.3 Réalisation des fonctions supplémentaires

Elles sont réalisées à partir de bascules D, avec un rebouclage adéquat de Q sur D. Les chargements synchrones, remises à zéros asynchrones ou synchrones sont effectués par le circuit suivant (étage i) :



4.8.4.4 Mise en cascade de compteurs

Nous n'aborderons ici que la mise en cascade des compteurs synchrones. La question qui se pose est : comment former un compteur plus grand en associant plusieurs compteurs identiques plus petits ? La question se pose par exemple lorsque que l'on veut compter en décimal. Exemple : on va utiliser plusieurs compteurs BCD (qui sont codés sur 4 bits) et les associer pour former un compteur qui compte de 0 à 999 :



Nous avons besoin de trois compteurs BCD. La sortie CEO doit valider le compteur de poids plus élevé au bon moment pour le coup d'horloge suivant. Quel est le bon moment ? Il faut pour répondre à cette question reprendre la séquence de comptage :

CEO_2	**Cnt2**	CEO_1	**Cnt1**	CEO_0	**Cnt0**
0	**0**	0	**0**	0	**0**
0	**0**	0	**0**	0	**1**
…	**…**	…	**…**	…	**…**
0	**0**	0	**0**	0	**8**
0	**0**	0	**0**	1	**9**
0	**0**	0	**1**	0	**0**
0	**0**	0	**1**	0	**1**
…	**…**	…	**…**	…	**…**
0	**0**	0	**8**	0	**8**
0	**0**	0	**8**	1	**9**
0	**0**	0	**9**	0	**0**
0	**0**	0	**9**	0	**1**
…	**…**	…	**…**	…	**…**
0	**0**	0	**9**	0	**8**
0	**0**	1	**9**	1	**9**
0	**1**	0	**0**	0	**0**
0	**1**	0	**0**	0	**1**
…	**…**	…	**…**	…	**…**
0	**1**	0	**0**	0	**8**
0	**1**	0	**0**	1	**9**
0	**1**	0	**1**	0	**0**

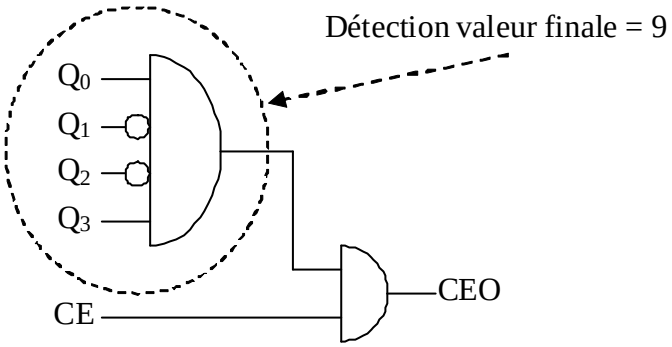
On constate sur la séquence que :

- Cnt0 est activé en permanence donc CE premier étage = 1.
- Cnt1 est activé quand Cnt0 = 9 via CEO_0.
- Cnt2 est activé quand Cnt1 = 9 et quand Cnt0 = 9 via CEO_1.

Du fonctionnement de cnt1, on pourrait déduire hâtivement que CEO vaut 1 quand le compteur atteint la valeur 9. C'est vrai pour cnt1 mais faux pour cnt2. En effet, on voit que CEO_1 = 1 quand cnt1 = 9 et quand son CE = 1, c'est-à-dire quand cnt0 = 9.

On incrémente la décade supérieure quand toutes les décades inférieures ont atteint 9, donc CEO = détection de la valeur finale de comptage AND CE

Le CEO du compteur BCD est donc réalisé de la manière suivante :

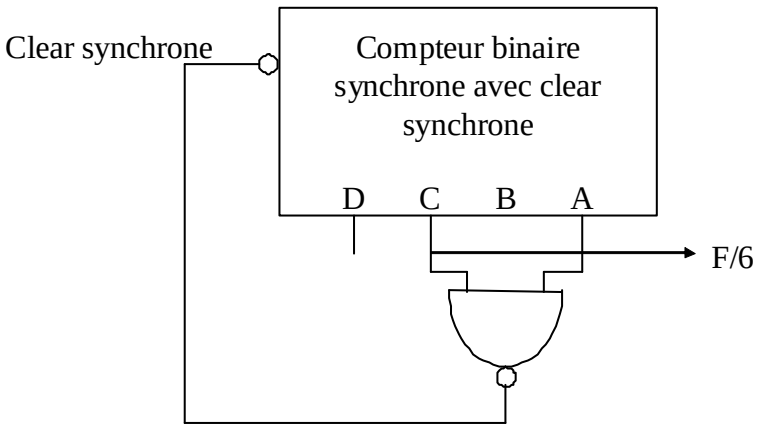


Ce résultat peut être étendu aux autres familles de compteurs. Seule la valeur finale détectée change.

4.8.4.5 Réalisation de compteurs modulo quelconque.

4.8.4.5.1 Action sur l'entrée Clear synchrone

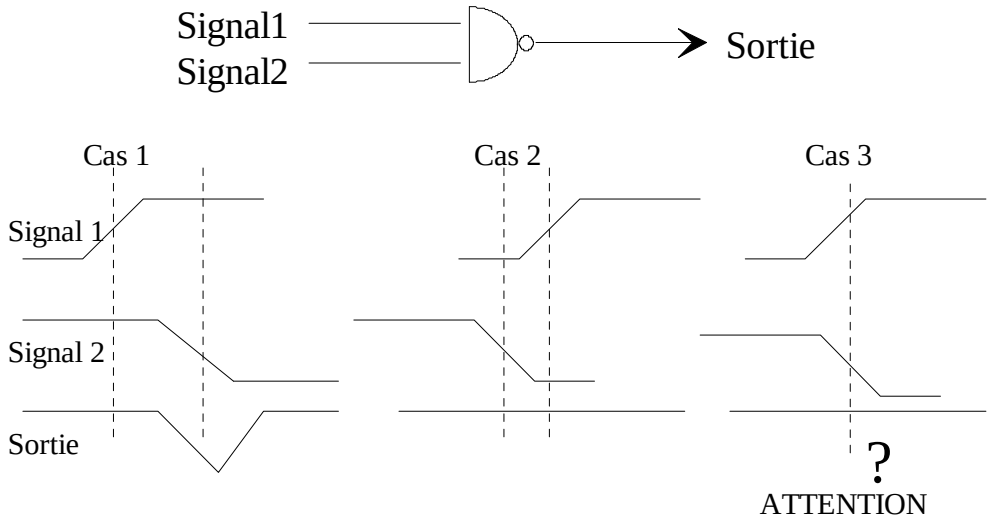
Lorsque l'état final du compteur est détecté, le front actif suivant de l'horloge remet à zéro le compteur. Une détection incomplète est possible et même souhaitable pour limiter les aléas, elle consiste à ne détecter que les bits à 1. Dans l'exemple suivant, on réalise un diviseur par 6 en détectant l'état 5, puis en réinitialisant le compteur au coup d'horloge suivant. Les états en sortie seront donc 0, 1, 2, 3, 4, 5, 0, 1, ...



Bien que le compteur soit synchrone, de petites dispersions dans les $t_{P\ ck\text{-}to\text{-}Q}$ peuvent fournir un état parasite transitoire. Ces aléas ne sont pas forcément gênants, nous allons les étudier ci-dessous.

4.8.4.5.2 Rappel des cas possibles d'aléas

Un aléa peut survenir lorsque deux entrées d'une porte élémentaire changent d'état à peu près en même temps. Si les temps de transitions sont voisins du temps de traversée de la porte, on a trois possibilités :



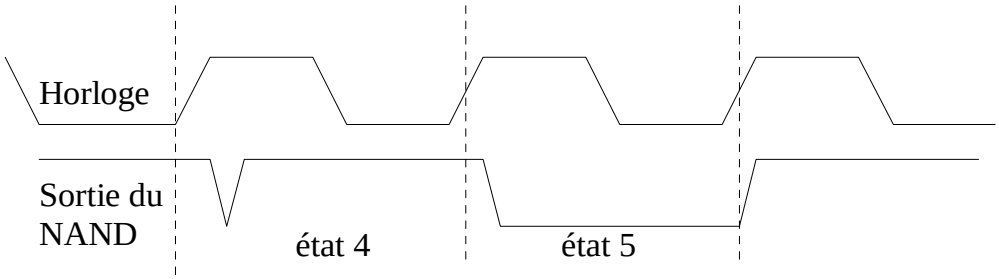
Dans le cas particulier de cette NAND, le cas n°1 provoquera systématiquement un parasite à 0 car la transition 0-1 sur le signal 1 fait changer la sortie (avec signal 2 à 1). Par contre, le cas n°2 ne provoquera jamais d'aléa puisque la transition 1-0 sur le signal 2 ne fait pas changer la sortie (avec signal 1 à 0). Quant au cas n°3, tout dépendra des caractéristiques réelles de chaque porte. Certaines portes produiront un aléa, d'autres non.

4.8.4.5.3 Influence de l'aléa

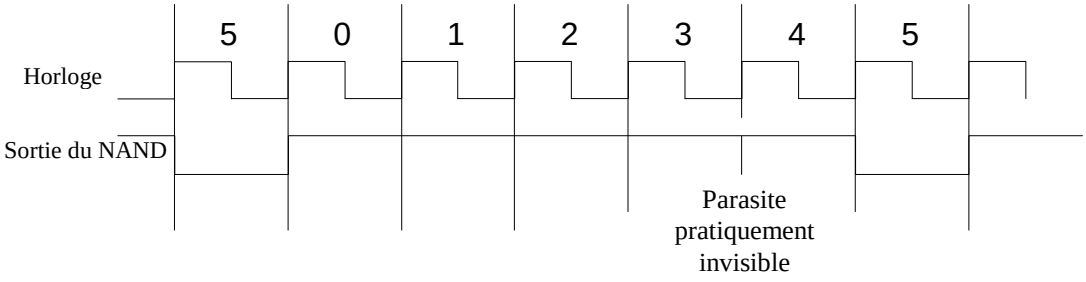
Reprenons l'exemple du diviseur par 6. L'aléa peut exister en sortie de la porte NAND détectant l'état 5, au passage de l'état 3 à l'état 4. Sur une entrée synchrone de remise à zéro, cet aléa n'a pas d'action car il ne se produit pas sur le front actif de l'horloge. Avec un clear synchrone, les aléas de commutation n'ont aucun effet. L'étude de ce type de montage s'en trouve grandement facilitée. Par contre, si on utilise un clear asynchrone, alors l'aléa au passage de l'état 3 à l'état 4 peut remettre le compteur à 0. Cela dépendra de sa durée et de la rapidité du compteur. Il est donc interdit d'utiliser un clear asynchrone pour réaliser un diviseur par N.

Comme nous l'avons déjà vu, il ne faut jamais utiliser comme horloge d'un compteur ou d'une bascule une combinaison logique de sorties d'un compteur. Des impulsions parasites (des glitches) se trouveront sur ce signal et seront prises pour des fronts d'horloge supplémentaires.

Par contre, en utilisant une sortie de bascule (comme dans notre exemple), vous avez la garantie d'un signal sans aléa de commutation. La largeur de ces glitches dépendra uniquement des temps de propagation du compteur (clock to Q) et de la rapidité de la porte mais pas de la fréquence de fonctionnement. L'observation du glitch est très délicate et il est difficile de l'observer sans savoir où il se trouve. Si par exemple, on a le chronogramme suivant à la fréquence de 100 MHz :



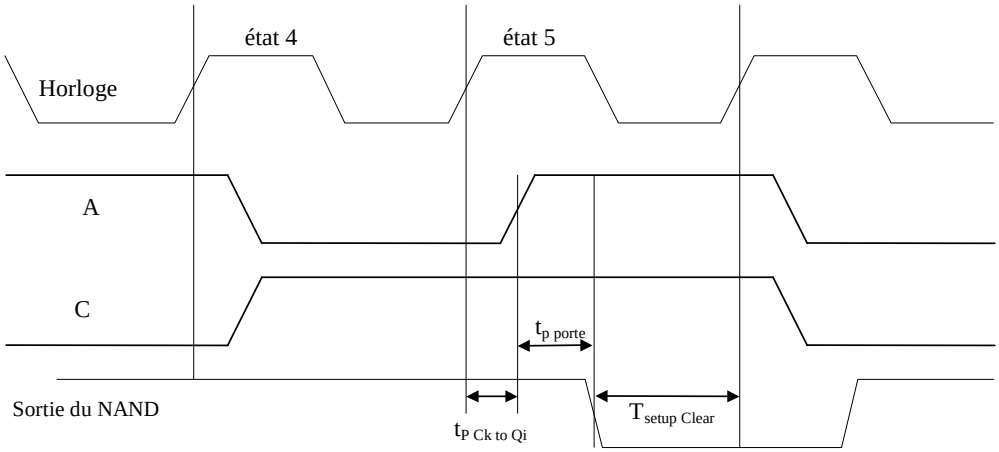
A 100 kHz, on aura un parasite de même largeur mais il ne sera pratiquement plus visible quoique toujours actif.



Si vous ne le cherchez pas, vous aurez du mal à le trouver. De plus, il faut un oscilloscope performant pour pouvoir l'observer. Une formule est à connaître : $t_{montée} = \frac{0.35}{bande\ passante}$.

Elle signifie qu'il faut un oscilloscope à bande passante élevée pour pouvoir observer un aléa court, par exemple 3.5 GHz pour un aléa d'une durée de 100 ps. Même si vous savez où il se trouve, l'observation d'un glitche rapide nécessite du matériel performant et donc coûteux.

La fréquence maximale de fonctionnement (pour l'exemple du diviseur par 6) est déterminée par le temps séparant le changement d'état en sortie de la NAND et le front actif suivant de l'horloge.

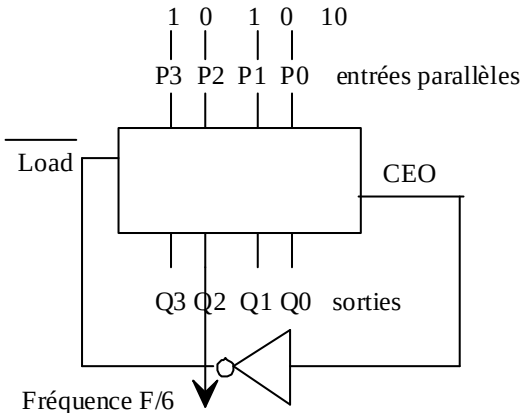


On déduit de ce chronogramme la fréquence maximale du compteur :

$$F_{max} = \frac{1}{t_{ck\ to\ Qi\ max} + t_{p\ porte\ max} + t_{setup\ Clear\ min}}$$

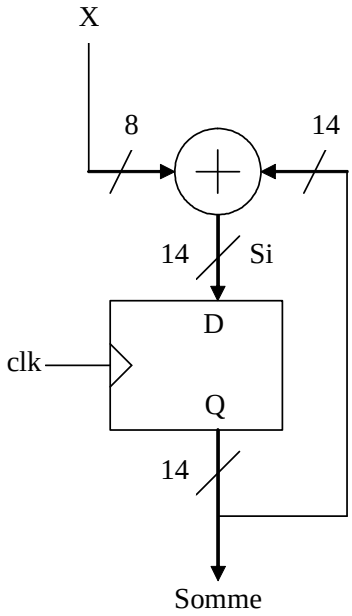
4.8.4.5.4 Action sur l'entrée LOAD synchrone

En mode comptage, au moment du passage de l'état 2^n-1 à l'état zéro, le compteur fournit un signal de retenue « CEO ». On l'utilise pour charger une valeur qui servira de valeur initiale dans la séquence. L'exemple suivant reprend le diviseur par 6 mais avec en sortie les états 10, 11 ,12 ,13, 14, 15, 10, 11, …



4.8.5 L'accumulateur de somme

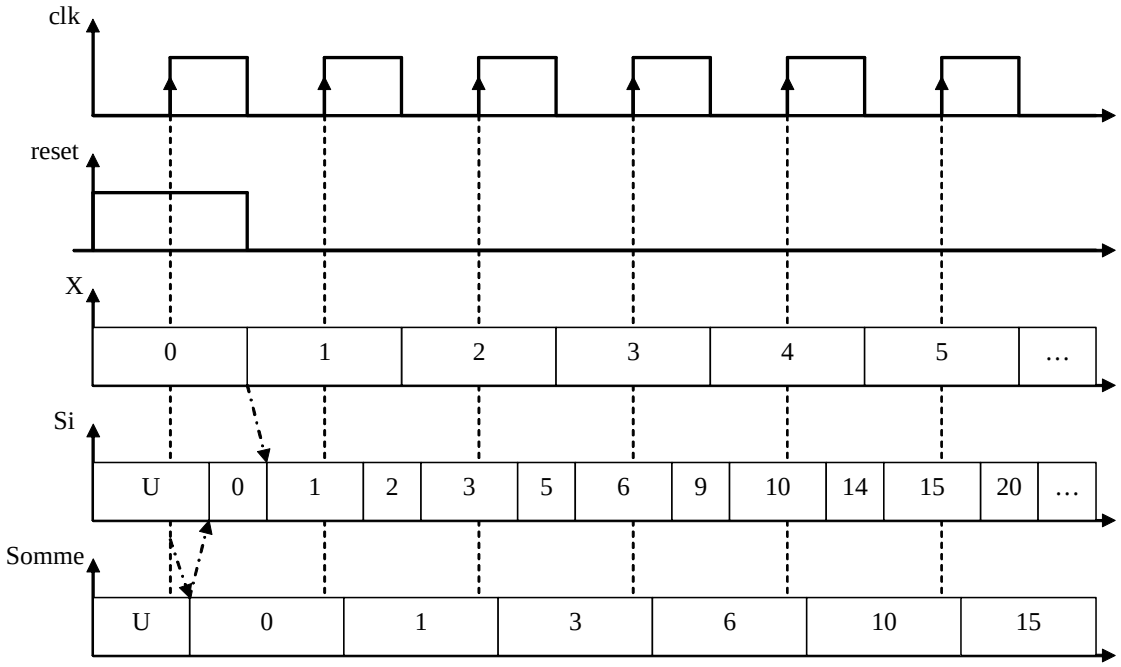
L'accumulateur de somme permet de réaliser une somme multiple à raison d'un coup d'horloge par terme à sommer.



Equation réalisée : $Somme = \sum_{i=1}^{L} X_i$

Exemple d'utilisation : calcul d'une moyenne mobile

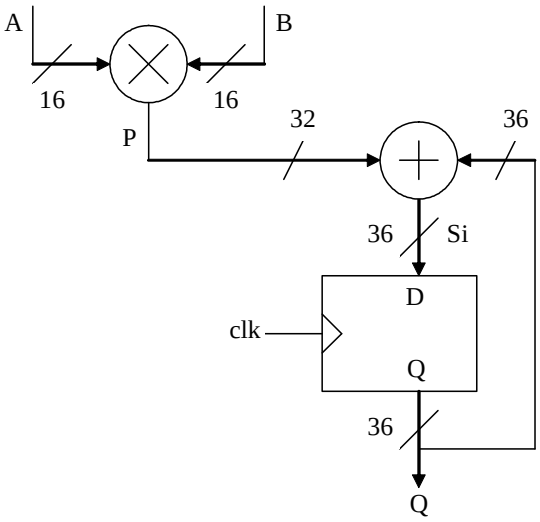
Il est composé d'un additionneur N bits suivi d'un registre N+k bits. Le paramètre k dépend du nombre de valeur N bits à sommer sans qu'il y ait débordement. Prenons un exemple avec N = 8 bits et 64 valeurs à additionner, c'est-à-dire que k = 6 bits (l'addition de 64 valeurs 8 bits tient sur 14 bits, donc k = 14 – 8 = 6 bits). Le chronogramme de fonctionnement est le suivant (on présente X sur le front descendant de l'horloge, registre mis à 0 au démarrage) :



Le registre doit être initialisé à 0 au début du calcul de la somme. Il faut utiliser un reset synchrone car ce reset est fonctionnel.

4.8.6 L'accumulateur de produit : MAC

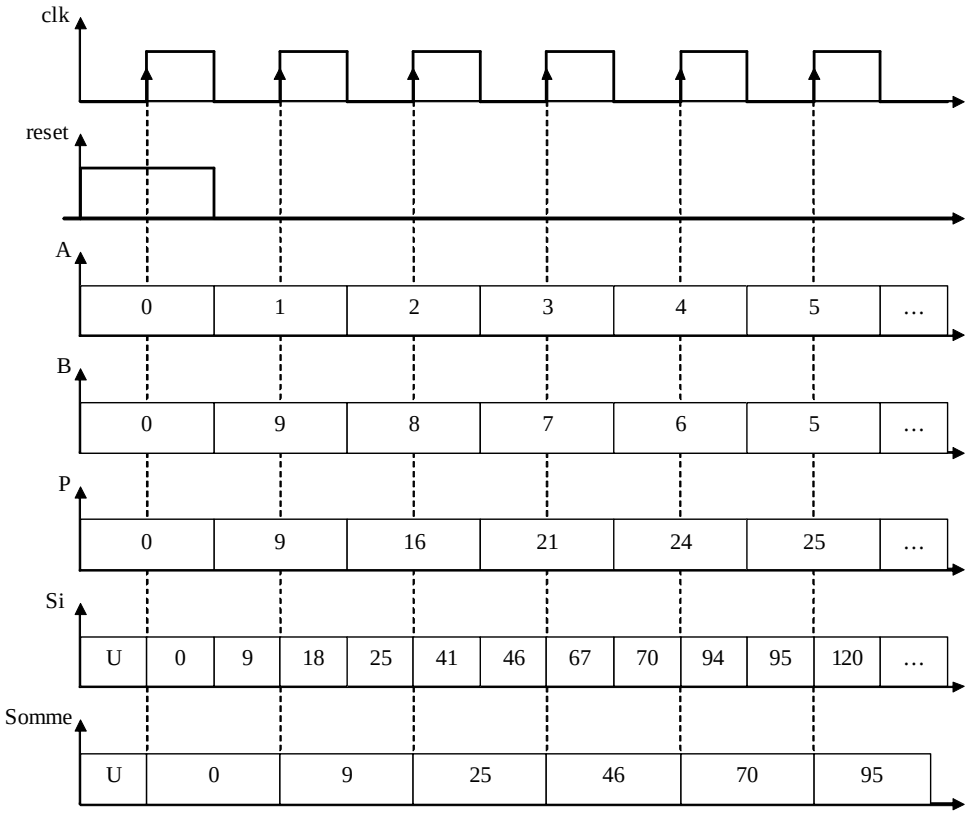
L'accumulateur de produit (MAC : Multiplier ACcumulator) est un accumulateur de somme précédé d'un multiplieur sur l'entrée X.



Equation réalisée : $Q = \sum_{i=1}^{L} A_i.B_i$

Exemple d'utilisation : la multiplication-accumulation permet de calculer un produit de convolution. C'est l'opération de base du filtrage numérique.

Le chronogramme de fonctionnement est le suivant (on présente A et B sur le front descendant de l'horloge, registre mis à 0 au démarrage) :



Le registre doit être initialisé à 0 au début du calcul de la somme des produits. Il faut utiliser un reset synchrone car ce reset est fonctionnel.

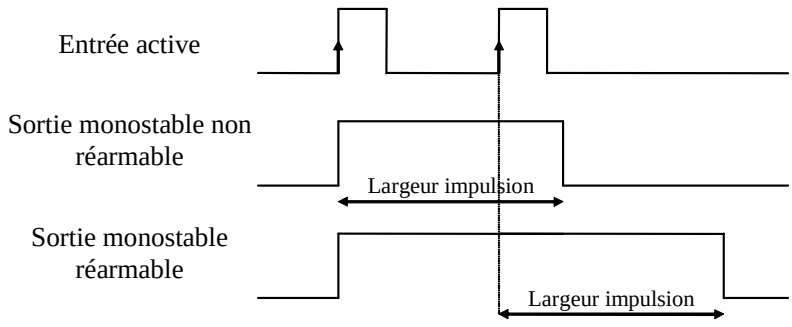
4.8.7 Les monostables

Un monostable est un circuit logique séquentiel qui délivre, sur le front actif d'un signal de commande, une impulsion de durée variable ajustée par un réseau RC. La durée de l'impulsion est approximativement égale à $0,7.R_{ext}.C_{ext}$. Son ordre de grandeur est compris entre une dizaine de nanosecondes et plusieurs secondes. Nous allons examiner deux types courants de monostables :

- SN74LS221 : double monostable non réarmable avec entrée trigger de Schmitt. Si un front actif se produit avant que l'impulsion de sortie ne revienne au repos, il n'est pas pris en compte. Sa table de vérité est la suivante :

entrées			sorties	
clear	A	B	Q	$\overline{Q}$
0	X	X	0	1
X	1	X	0	1
X	X	0	0	1
1	0	↑	⊓	⊓
1	↓	1	⊓	⊓
↑	0	1	⊓	⊓

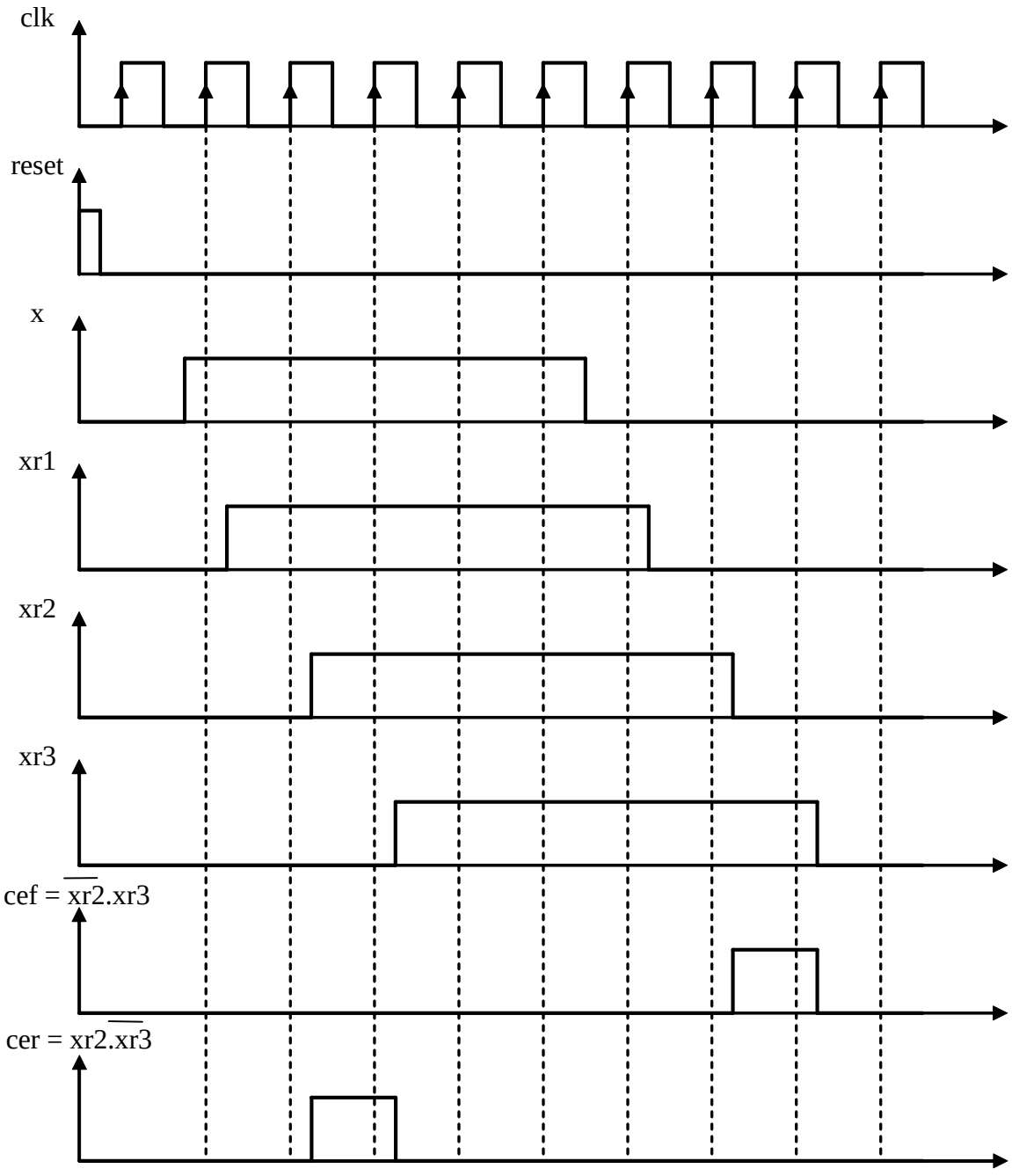
- SN74LS123 : double monostable réarmable. Si un front actif se produit avant que l'impulsion de sortie ne revienne au repos, sa durée est étendue de la valeur initiale. Sa table de vérité est identique à la précédente.



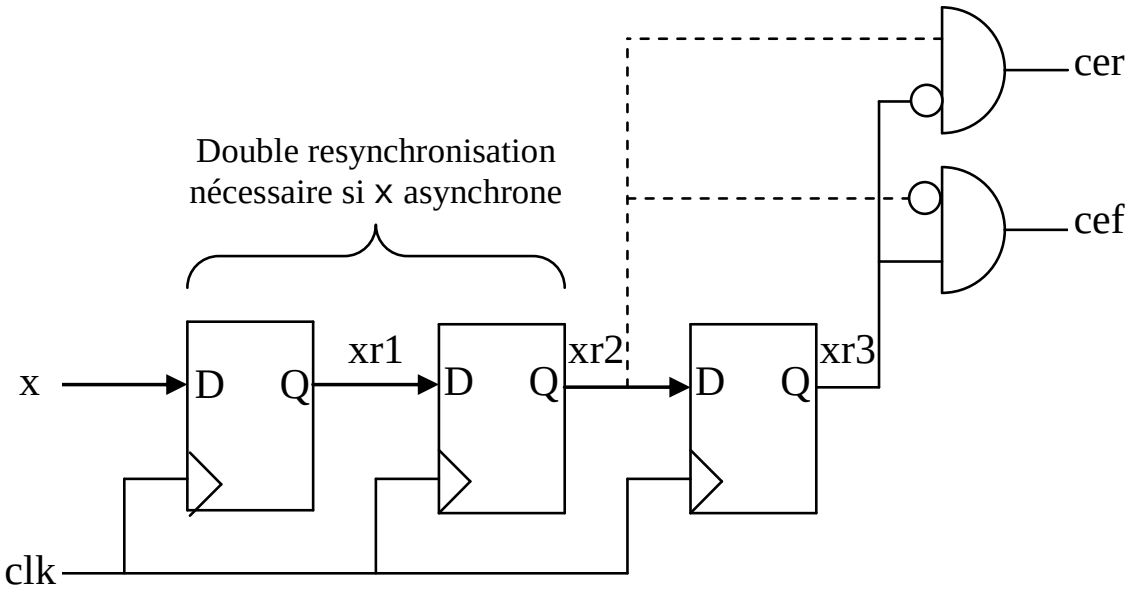
Les monostables ne sont plus utilisés aujourd'hui en conception des circuits logiques. Mais le principe est toujours intéressant, notamment sous sa forme séquentielle synchrone (sans réseau RC bien sûr).

4.8.8 Circuit de détection de front

Il arrive très souvent dans un design qu'il soit nécessaire de détecter si un signal a changé d'état : c'est la détection de front, montant ou descendant. En logique synchrone, nous allons utiliser pour cela une horloge qui doit être plus rapide que le rythme du changement d'état sur le signal à détecter (par exemple, au moins 10 fois plus rapide). Prenons l'exemple d'un signal x qui est une entrée asynchrone par rapport à l'horloge du design. Les chronogrammes suivants montrent le principe de la détection du front :



Le montage correspondant est le suivant :



`cer` est un signal de validation qui vaut 1 pendant une période de `clk` s'il y a eu un front montant sur `x`. `cef` vaut 1 sur un front descendant de `x`. Si `x` est synchrone, on peut supprimer les deux premières bascules du montage qui servent seulement à réduire la métastabilité si `x` est asynchrone (méthode de la double resynchronisation).

4.9 Description en VHDL

4.9.1 Latch transparent et bascule D

Nous avons vu à la fin de l'exemple du multiplexeur (§2.4.3) le phénomène de la mémorisation implicite qui générait un latch quand toutes les branches d'une assignation conditionnelle n'étaient pas définies. Cela pouvait sembler être un inconvénient majeur. En fait, cette mémorisation implicite permet la simplification de la description des circuits séquentiels. L'exemple suivant vous montre une bascule D la plus simple possible :

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. entity basc is
4.   port(D : in std_logic;
5.      clk : in std_logic;
6.        Q : out std_logic);
7. end basc;
```

```
8. architecture comporte of basc is
9. begin

10. process(clk) begin
11.   if (clk'event and clk ='1') then
12.     Q <= D;
13.   end if;
14. end process;

15. end comporte ;
```

Le process est sensible au seul signal actif du design, l'horloge clk. A la ligne 11, dans le if, on trouve l'attribut `event`. Accolé à un signal, il retourne un booléen qui sera vrai si un événement (c'est à dire un changement d'état) s'est produit sur ce signal. Par exemple, la forme :

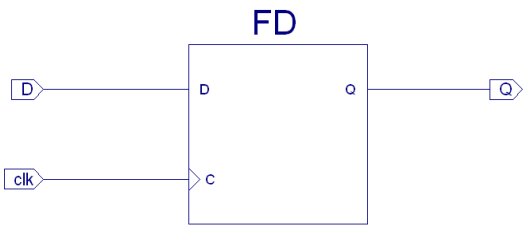
```
if (CLK'event and CLK='1') then
```

sera utilisée pour détecter un front montant d'horloge (s'il y a eu changement d'état sur CLK et si CLK vaut 1). La forme :

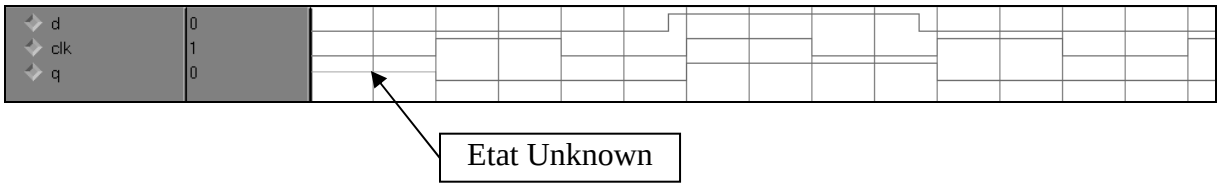
```
if (CLK'event and CLK='0') then
```

sera utilisée pour détecter un front descendant d'horloge (s'il y a eu changement d'état sur CLK et si CLK vaut 0). Le package std_logic_1164 définit deux fonctions équivalentes **`rising_edge(CLK)`** et **`falling_edge(CLK)`** qui pendant longtemps n'ont pas été reconnus par les synthétiseurs (notamment celui de Synopsys). C'est pourquoi beaucoup de designers utilisent encore la forme (`CLK'event and CLK='1'`). **N'oubliez surtout pas le `CLK'event`** même si dans cet exemple on pourrait croire qu'il ne sert à rien. C'est vrai en simulation, mais le synthétiseur en a besoin pour inférer correctement une bascule D.

Voyons le fonctionnement de la bascule D, lignes 10 à 14. Le process est activé (ligne 10) par l'horloge clk. Si le front montant de l'horloge arrive (ligne 11) alors Q prend la valeur de D (ligne 12). Sinon (le « `if… then… else…` » est incomplet), Q garde son état précédent. Il s'agit là de la description d'une bascule D. On voit ici tout l'intérêt de la mémorisation implicite. Il suffit donc de mettre une assignation entre deux signaux **dans la branche sensible à l'horloge** « `if (clock 'event and clock ='1') then` » pour générer (inférer) une bascule D. La description est équivalente au schéma :



La bascule D ainsi générée est dépourvue de reset. L'état de Q est donc inconnu au départ comme le montre la simulation suivante :



La description d'un latch transparent est aussi simple. Il suffit de retirer le `clk'event` dans le if.

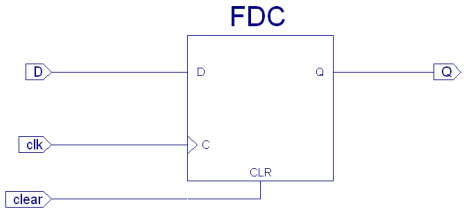
```
10.  process(clk, D) begin     -- D active le process car quand
11.    if (clk ='1') then      -- clk = 1, D est copiée dans Q
12.      Q <= D;               -- en permanence
13.    end if;
14.  end process;
```

Quand clk vaut 1, on copie D sur Q en permanence, et quand clk vaut 0, on garde le dernier état stocké sur Q. La description est équivalente au schéma :

Revenons à notre bascule D. On peut lui ajouter un reset asynchrone de la manière suivante :

```
process(clk, clear) begin -- clk et clear active le process, clear prioritaire
  if (clear ='1') then -- si clear vaut 1
    Q <= '0'; -- alors Q prend la valeur 0
  elsif (clk'event and clk ='1') then -- sinon, si front montant sur clk
    Q <= D; -- alors Q prend la valeur de D
  end if; -- sinon Q garde sa valeur précédente
end process;
```

Le clear est bien asynchrone puisque le « `if (clear ='1')` » est en dehors de la branche sensible à l'horloge « `elsif (clock 'event and clock ='1')` ». `clear` est dans la liste de sensibilité du process puisque c'est un signal actif. **Le `elsif` est ici absolument obligatoire**. On ne doit tester le front d'horloge que si `clear` est à 0 car le clear est prioritaire sur l'horloge. Cette description est équivalente au schéma :



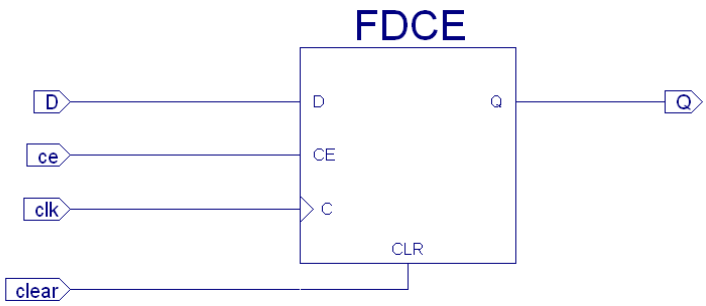
Si le « `if (clear ='1')` » était dans la branche sensible à l'horloge « `elsif (clock 'event and clock ='1')` » alors le clear serait synchrone comme dans l'exemple suivant. Clear a disparu de la liste de sensibilité du process car ce n'est plus un signal actif. Seule l'horloge clk est active.

```
process(clk) begin -- clear a disparu car il est synchrone.
  if (clk'event and clk ='1') then --si front montant sur clk
    if (clear ='1') then -- si clear vaut 1
      Q <= '0'; -- alors Q prend la valeur 0
    else
      Q <= D; -- sinon Q prend la valeur de D
    end if;
  end if; -- sinon Q garde sa valeur précédente
end process;
```

On peut de plus ajouter un signal de validation "`ce`" de la manière suivante :

```
process(clk, clear) begin -- clk et clear active le process
  if (clear ='1') then -- si clear vaut 1
    Q <= '0'; -- alors Q prend la valeur 0
  elsif (clk'event and clk ='1') then -- sinon, si front montant sur clk
    if (ce ='1') then -- si ce vaut 1
      Q <= D; -- alors Q prend la valeur de D
    end if; -- sinon Q garde sa valeur précédente
  end if; -- sinon Q garde sa valeur précédente
end process;
```

On pourrait en toute logique ajouter la condition sur le `ce` directement dans la condition en écrivant `elsif (clock'event and clock='1' and ce='1')`. Ce type de description a une nette tendance à perturber le synthétiseur. Il vaut mieux décomposer les conditions. Cette description est équivalente au schéma :

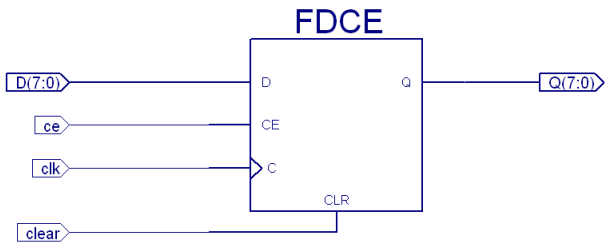


4.9.2 Registre

La description d'un registre N bits en VHDL diffère fort peu de la description d'une bascule D. Il suffit de déclarer D et Q comme `std_logic_vector(N-1 downto 0)`. L'exemple suivant décrit un registre 8 bits :

```
library IEEE;
use IEEE.std_logic_1164.all;

entity reg is
  port(D : in std_logic_vector(7 downto 0);
   clear : in std_logic;
      ce : in std_logic;
     clk : in std_logic;
       Q : out std_logic_vector(7 downto 0));
end reg;

architecture comporte of reg is
begin

  process(clk, clear) begin
    if (clear ='1') then
      Q <= (others => '0'); -- tous les bits de Q à 0
    elsif (clk 'event and clk ='1') then
      if (ce ='1') then
        Q <= D;
      end if;
    end if;
  end process;

end comporte ;
```

On voit encore une fois l'intérêt de la mémorisation implicite. Il suffit de mettre une assignation entre deux signaux dans la branche « `if (clock 'event and clock ='1') then` » pour générer (inférer) un registre. C'est la taille des signaux qui détermine la taille du registre. Vous voyez ici tout l'intérêt d'utiliser un langage comme VHDL plutôt qu'une saisie de schéma. La description est équivalente au schéma :



4.9.3 Registre à décalage

Le registre à décalage est à peine plus compliqué à décrire que le registre simple. Il nécessite l'utilisation d'une boucle `for` :

```
1.   library IEEE;
2.   use IEEE.std_logic_1164.all;

3.   entity shift is
4.     port( CLK : in std_logic ;
5.         CLEAR : in std_logic;
6.            CE : in std_logic;
7.           Din : in std_logic;
8.             Q : out std_logic_vector(7 downto 0));
9.   end shift;

10.  architecture RTL of shift is
11.    signal Qint : std_logic_vector(Q'range);
12.  begin
13.    process(CLK, CLEAR) begin
14.      if (CLEAR='1') then
15.        Qint <= (others => '0');
16.      elsif (CLK'event and CLK='1') then
17.        if (ce = '1') then
18.          for i in 1 to 7 loop
19.            Qint(8-i) <= Qint(7-i);
20.          end loop;
21.          Qint (0) <= Din;
22.        end if;
23.      end if;
24.    end process;
25.    Q <= Qint;
26.  end;
```

Les points suivants sont importants :

- Ligne 11 : l'attribut range. Il fait partie des nombreux attributs de VHDL. Si le signal A est défini par un `std_logic_vector(X downto Y)`, alors `A'range` est équivalent à `X downto Y` et peut être utilisé à la place. Dans notre exemple :

```
signal Qint : std_logic_vector(Q'range);
```

Qint a la même largeur que Q (de 7 à 0). L'attribut range simplifie l'écriture des modèles génériques.

- Ligne 25. Il est impossible en VHDL d'utiliser la valeur d'un signal dans l'architecture d'un composant s'il a été déclaré en sortie pure dans l'entité. L'assignation suivante ne peut pas être effectuée sur le signal Q.

```
Qint(8-i) <= Qint(7-i);
```

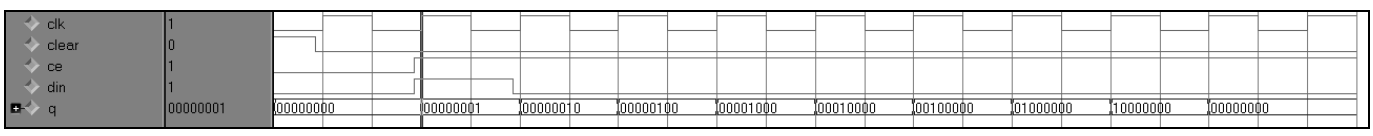
On effectue donc l'assignation sur un signal déclaré localement Qint, puis on assigne ce signal temporaire à la sortie Q en dehors du process (pour ne pas générer une bascule de trop).

```
Q <= Qint;
```

On aurait pu contourner ce problème en déclarant Q en mode buffer, ce qui aurait autorisé son utilisation dans l'architecture. On n'utilise jamais ce mode, car il y a trop de restrictions à son utilisation. Le mode buffer n'est plus utilisé en VHDL, la déclaration d'un signal temporaire comme Qint donne une représentation strictement équivalente.

- Le décalage du registre est réalisé aux lignes 18 à 21. Si CE vaut 1, alors on copie Qint(6) dans Qint(7), puis Qint(5) dans Qint(6), puis Qint(4) dans Qint(5), …, puis Qint(0) dans Qint(1). Il suffit ensuite de copier din dans Qint(0) pour terminer le décalage. Vous noterez l'utilisation de la boucle for qui est spatiale en VHDL alors qu'elle est temporelle dans un langage de programmation comme le C.

Le chronogramme suivant montre l'évolution des signaux :



4.9.4 Compteur

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.std_logic_unsigned.all;

4. entity compteur is
5.   port( CLK : in std_logic ;
6.          CE : in std_logic ;
7.       CLEAR : in std_logic;
8.         CEO : out std_logic;
9.        DOUT : out std_logic_vector(3 downto 0));
10.end compteur;

11.architecture a1 of compteur is
12.  signal INT_DOUT : std_logic_vector(DOUT'range) ;
13.begin
14.  process(CLK, CLEAR) begin
15.    if (CLEAR='1') then
16.      INT_DOUT <= (others => '0');
17.    elsif (CLK'event and CLK='1') then
18.      if (CE='1') then
19.          INT_DOUT <= (INT_DOUT + 1);
20.      end if;
21.    end if;
22.  end process;
23.  CEO <= INT_DOUT(0) and INT_DOUT(1) and INT_DOUT(2) and INT_DOUT(3) and CE;
24.  dout <= int_dout;
25.end;
```

Les points suivants doivent être notés :

- Ligne 3. Le package std_logic_1164 définit le type `std_logic`, mais pas les opérateurs arithmétiques qui vont avec. Le package propriétaire Synopsys `std_logic_unsigned` définit ces opérateurs en considérant que le type std_logic est non signé. Cette déclaration est obligatoire par exemple pour effectuer l'opération suivante avec INT_DOUT déclaré comme un signal std_logic_vector :

  ```
  INT_DOUT <= (INT_DOUT + 1);
  ```

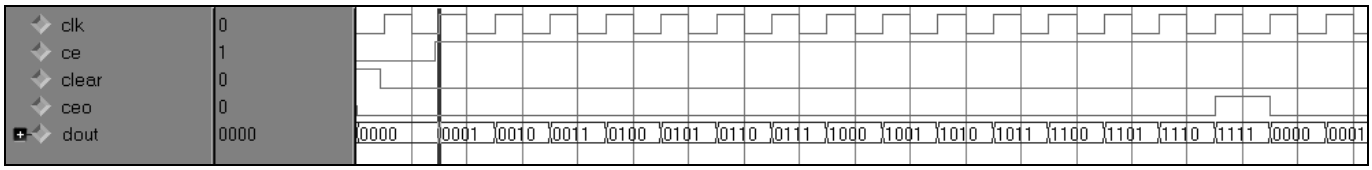
- Ligne 23. Le signal CEO passe à 1 quand l'état final de la séquence de comptage est atteint (ici, la valeur 15) et quand le signal de validation d'entrée CE vaut 1. Ceci est traduit par l'assignation :

  ```
  CEO <= INT_DOUT(0) and INT_DOUT(1) and INT_DOUT(2) and INT_DOUT(3) and CE;
  ```

  Pourquoi cette assignation n'est-elle pas dans le processus synchrone ? parce que CEO passerait à 1 sur le front d'horloge qui suit le déclenchement de la condition (c'est à dire quand la sortie du compteur est égale à 0. N'oubliez pas la bascule D qui est

automatiquement générée ! Il faudrait donc détecter la valeur (état final – 1) pour que CEO passe à 1 sur l'état final, ce qui ne serait pas très clair pour la compréhension du design. La solution consiste donc à réaliser une détection purement combinatoire de l'état final, donc en dehors du process synchrone.

Le chronogramme suivant montre le fonctionnement du compteur. On voit bien que CEO passe à 1 pendant l'état 15 :



Pour réaliser un compteur BCD (qui va de 0 à 9), il suffit de remplacer la ligne 19 :

```
INT_DOUT <= (INT_DOUT + 1);
```

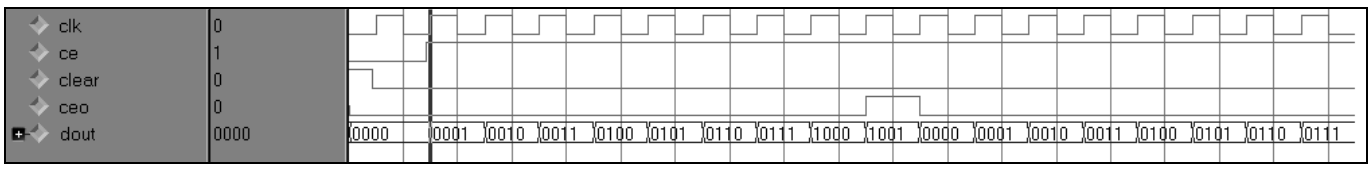
Par les lignes suivantes :

```
if (INT_DOUT=9) then
  INT_DOUT <= (others => '0');
else
  INT_DOUT <= (INT_DOUT + 1);
end if;
```

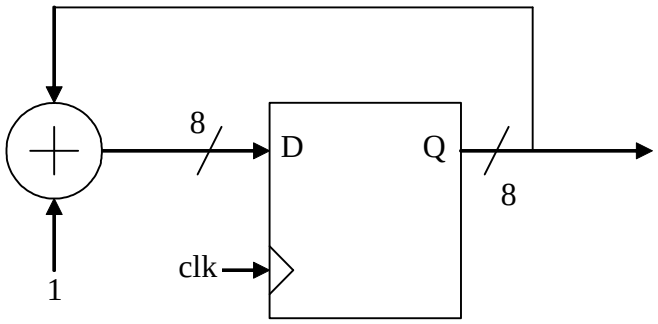
Le CEO doit être modifié de la manière suivante :

```
CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and
INT_DOUT(3) and CE;
```

Le chronogramme suivant montre le nouveau fonctionnement du compteur :



La logique utilisée pour réaliser un compteur est généralement basée sur un additionneur associé à un registre, notamment dans les FPGA. C'est la méthode la plus simple pour réaliser le comptage. Elle utilise un minimum de logique et de ressource de routage et donne la fréquence maximum la plus élevée. Voici l'exemple d'un compteur binaire 8 bits.



Equation réalisée : $Q_{n+1} = Q_n + 1$

Pour réaliser un décompteur, il suffit de remplacer le 1 par un -1 dans le schéma précédent et de remplacer la ligne 19 du compteur par :

```
INT_DOUT <= (INT_DOUT - 1);
```

4.9.5 Accumulateur

Voici un exemple d'accumulateur de somme non signé :

```
1.   library IEEE;
2.   use IEEE.std_logic_1164.all;
3.   use IEEE.std_logic_unsigned.all;

4.   entity accu_som is
5.     port( CLK : in std_logic ;
6.           CE : in std_logic ;
7.        CLEAR : in std_logic;
8.            X : in std_logic_vector(7 downto 0);
9.        Somme : out std_logic_vector(13 downto 0));
10.  end accu_som;

11.  architecture a1 of accu_som is
12.    signal Si : std_logic_vector(Somme'range) ;
13.    signal Somme_int : std_logic_vector(Somme'range) ;
14.  begin

15.    Si <= "000000"&X + Somme_int;
16.    Somme <= Somme_int;
17.    process(CLK) begin
18.      if (CLK'event and CLK='1') then
19.        if (CLEAR='1') then
20.          Somme_int <= (others => '0');
21.        elsif (CE='1') then
22.          Somme_int <= Si;
23.        end if;
24.      end if;
25.    end process;

26.  end;
```

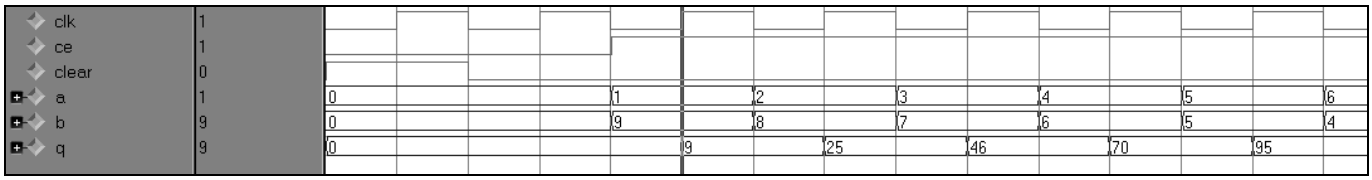
La description du design ne pose pas de problèmes particuliers. A la ligne 15, on concatène 6 zéros à gauche de X pour convertir le signal 8 bits non signé en un signal non signé de 14 bits que l'on additionne avec Somme. Et à la ligne 22, on copie le résultat Si dans le registre 14 bits. Comme pour le compteur, on travaille avec un signal temporaire Somme_int que l'on copie dans Somme à la ligne 16. Dans cet exemple, le Clear est fonctionnel et doit être **obligatoirement synchrone**. Le fonctionnement est conforme à la théorie vue au §4.8.5.

4.9.6 MAC

Il suffit de modifier légèrement l'accumulateur de somme précédent pour réaliser un MAC (signé cette fois) :

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.std_logic_signed.all;

4. entity mac is
5.   port( CLK : in std_logic;
6.         CE : in std_logic;
7.       CLEAR : in std_logic;
8.           A : in std_logic_vector(15 downto 0);
9.           B : in std_logic_vector(15 downto 0);
10.          Q : out std_logic_vector(35 downto 0));
11.  end mac;

12.  architecture a1 of mac is
13.    signal P : std_logic_vector(31 downto 0);
14.    signal Si, Q_int : std_logic_vector(Q'range);
15.  begin
16.    P <= A*B;
17.    Si <= P(31)&P(31)&P(31)&P(31)&P + Q_int;
18.    Q <= Q_int;

19.    process(CLK) begin
20.      if (CLK'event and CLK='1') then
21.        if (CLEAR='1') then
22.          Q_int <= (others => '0');
23.        elsif (CE='1') then
24.          Q_int <= Si;
25.        end if;
26.      end if;
27.    end process;
28.  end;
```

La description du design est assez directe. A la ligne 16, on réalise le produit signé P = A*B. A la ligne 17, on convertit P de 32 bits signés en 36 bits signés par extension de signe puis on l'additionne avec Q. Et à la ligne 24, on copie le résultat `Si` dans le registre 36 bits. Dans cet exemple, le Clear est fonctionnel et doit être **obligatoirement synchrone**. Le fonctionnement est conforme à la théorie vue au §4.8.6.

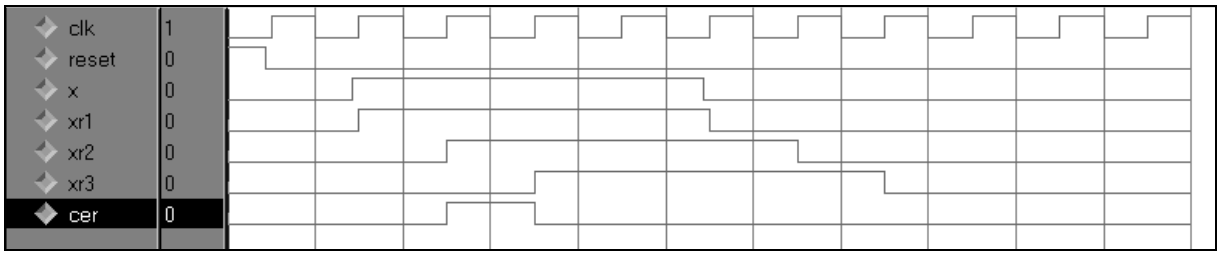


4.9.7 Circuit de détection de front

La description en VHDL est évidente et n'appelle pas de commentaires particuliers. Voici un exemple de circuit de détection de front montant :

```
library IEEE;
use IEEE.std_logic_1164.all;

entity front is
    port(clk : in std_logic ;
       reset : in std_logic;
           x : in std_logic;
         cer : out std_logic);
end front;


architecture a1 of front is
     signal xr1 : std_logic;
     signal xr2 : std_logic;
     signal xr3 : std_logic;
begin
     cer <= xr2 and not xr3;
     process(clk, reset) begin
          if (reset = '1') then
               xr1 <= '0';
               xr2 <= '0';
               xr3 <= '0';
          elsif (clk'event and clk='1') then
               xr1 <= x;
               xr2 <= xr1;
               xr3 <= xr2;
          end if;
     end process;
end;
```

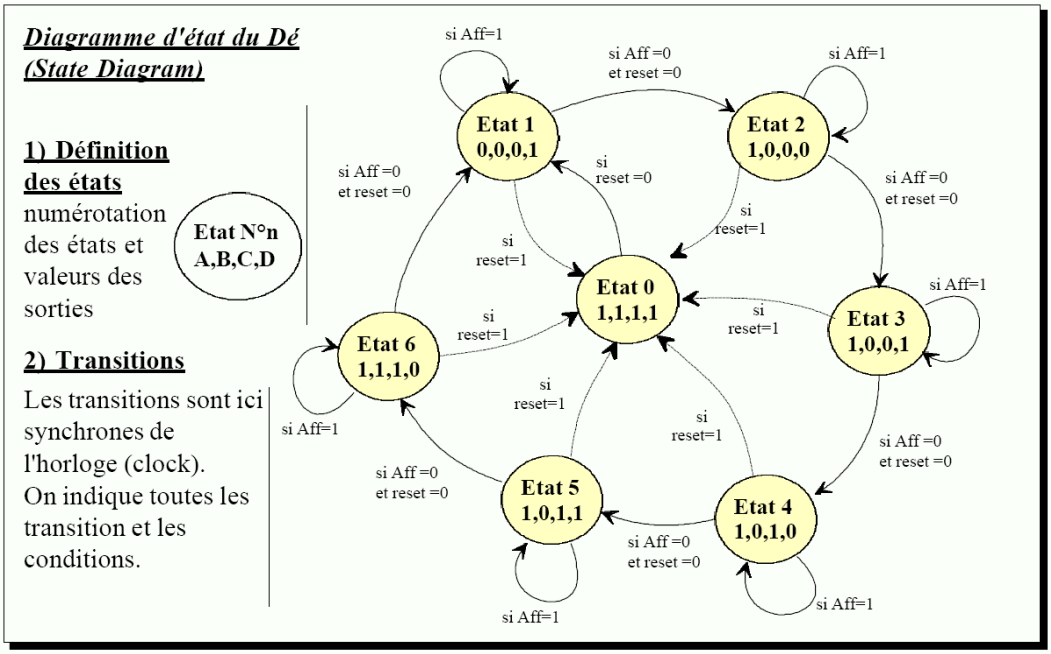
Le chronogramme suivant montre son fonctionnement :


clk
1
reset
0
x
0
xr1
0
xr2
0
xr3
0
cer
0

5 Montages en logique séquentielle

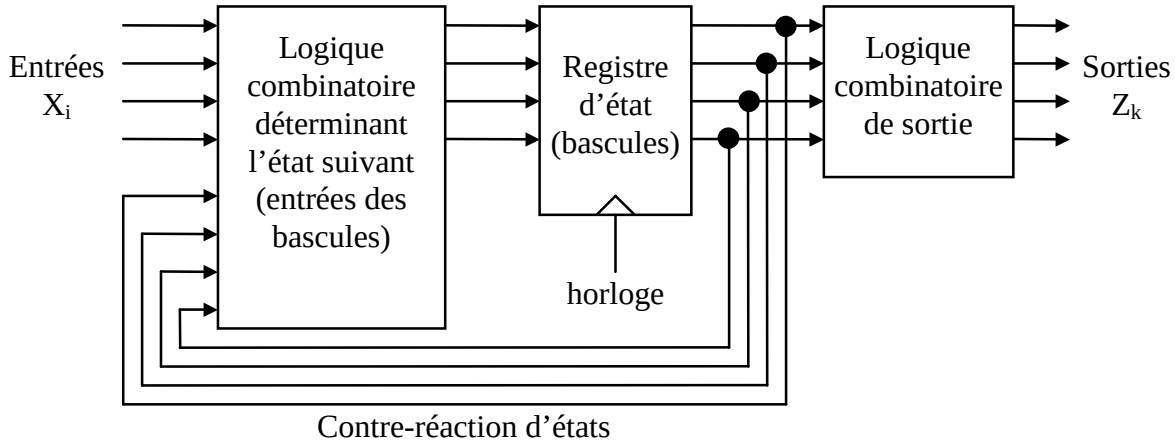
5.1 Machines d'états

Dans ce paragraphe, nous allons examiner une des familles les plus importantes de circuit séquentiel : la machine d'état ou machine à états finis (FSM : Finite State Machine). Une machine d'état est appelée ainsi car la logique séquentielle qui l'implémente ne peut prendre qu'un nombre fini d'états possibles. Il s'agit d'une manière de formaliser les automates qui est très utilisée en automatisme. En électronique, on peut s'en servir pour décrire le fonctionnement d'un petit contrôleur. On utilise pour cela une table de transition d'état ou diagramme d'état. L'exemple suivant montre un diagramme d'état pour un montage électronique qui réalise un dé à jouer : Les cercles représentent les 7 états et les flèches représentent les transitions entre les états.

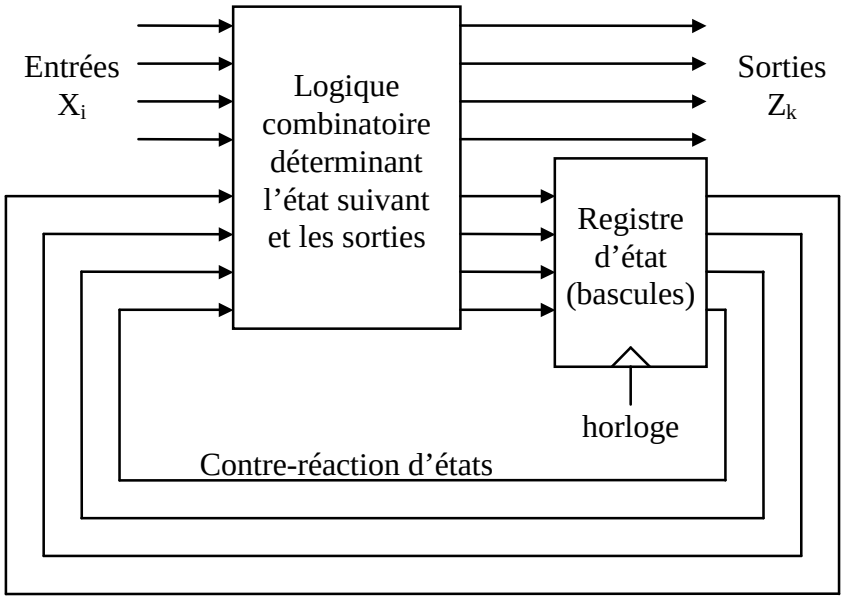


Il existe deux familles de machines d'état :

- la machine de Moore. Les sorties dépendent seulement de l'état présent n. Un circuit combinatoire d'entrée détermine, à partir des entrées et de l'état présent n, les entrées des bascules D du registre d'état permettant de réaliser l'état futur n+1. Les sorties sont une combinaison logique de la sortie du registre d'état et changent de manière synchrone avec le front actif de l'horloge. L'inconvénient de cette machine, c'est son temps de réaction à un changement sur ses entrées qui est égale à une période de l'horloge dans le pire des cas.



- la machine de Mealy. Les sorties dépendent de l'état présent n mais aussi de la valeur des entrées. La sortie peut donc changer de manière asynchrone en fonction de la valeur des entrées. Son temps de réaction à un changement sur X_i est bien meilleur (c'est un temps de propagation combinatoire). Il existe une variante synchrone de la machine de Mealy avec un registre placé sur les sorties et activé par l'horloge. Toutefois, la machine de Moore est plus adaptée pour réaliser une machine d'état totalement synchrone.



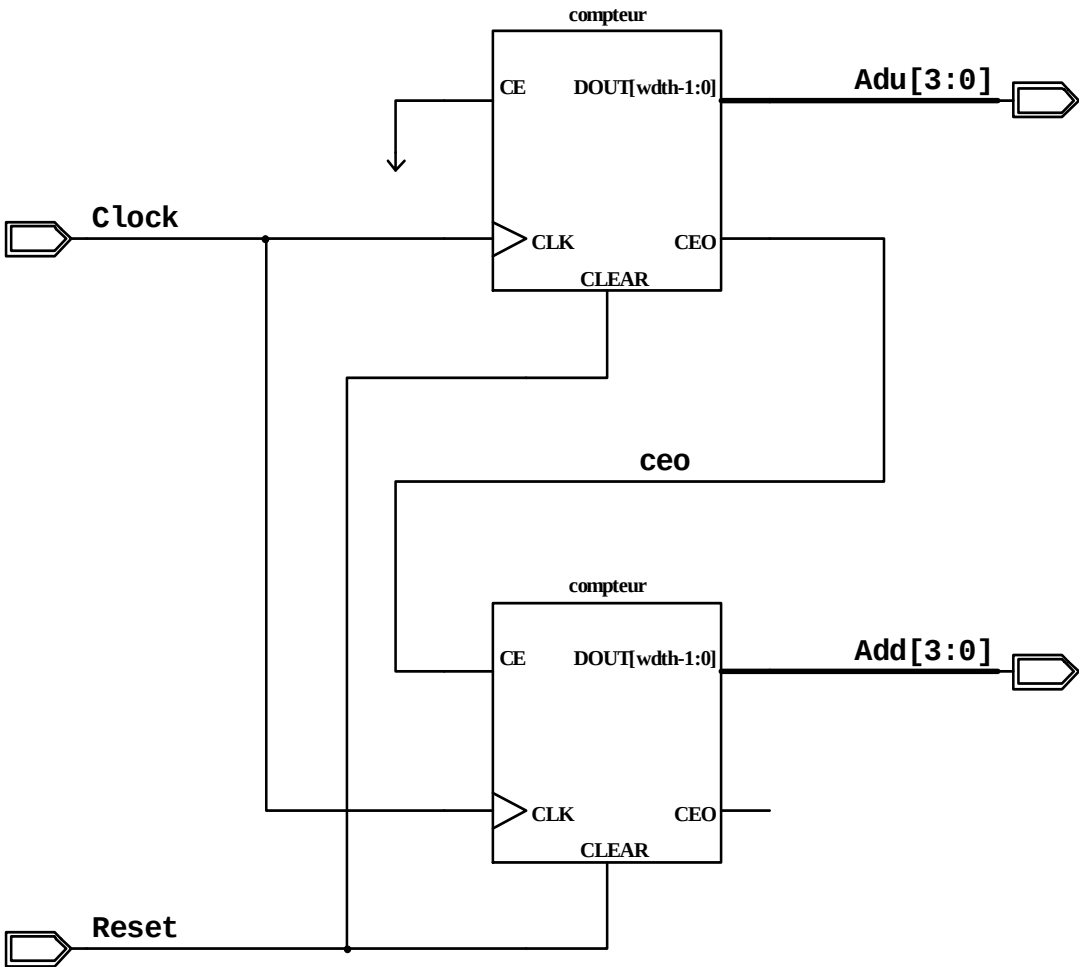
Les machines d'états servent à réaliser des automatismes complexes (contrôleur de feux tricolores par exemple) et ne nous intéressent pas directement. N'oublions pas que l'aboutissement ultime de la machine d'état en tant que système de contrôle, c'est bien sûr le microprocesseur.

5.2 Description en VHDL

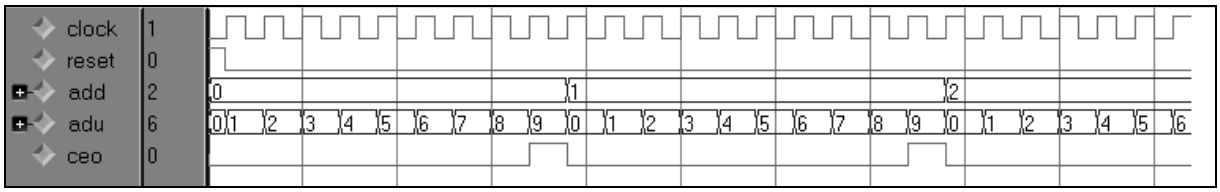
5.2.1 Description modulaire et paramètres génériques

Lorsque la complexité du design augmente, on le divise en blocs plus petits pour maintenir dans chaque bloc un niveau de complexité acceptable. On utilise pour cela une description modulaire hiérarchique, c'est-à-dire une arborescence hiérarchique de composants (un composant correspond un bloc). Le design principal à la racine de l'arborescence s'appelle généralement le top level design (parfois le root design). L'utilisation d'un composant dans le design s'appelle **l'instanciation d'un composant**. Instancier un composant en VHDL est équivalent en saisie de schéma à poser le symbole graphique du composant sur le schéma.

La description modulaire conduit assez naturellement à la notion de composant générique, c'est-à-dire un composant dont la taille est déterminée au moment où on l'instancie dans le design. Il existe deux méthodes en VHDL pour faire de la description modulaire. Nous allons maintenant voir la première, la description sans package. Nous souhaitons réaliser la fonction logique suivante :



Il s'agit de deux compteurs BCD montés en cascade pour compter de 0 à 99. Le chronogramme suivant montre le fonctionnement du montage :



La description modulaire sans package du design est la suivante. Dans cet exemple, on définit d'abord un composant `compteur` (lignes 1 à 31), puis le top level design `gen_cnt` (lignes 32 à 53). Le composant et le top level peuvent être mis dans des fichiers séparés. Dans l'architecture de `gen_cnt` (lignes 40 à 53), il faut définir les entrées-sorties du composant via l'instruction component (lignes 41 à 48) qui a la même forme que la déclaration de l'entité correspondante. Il ne reste plus ensuite qu'à instancier les 2 composants `compteur` (lignes 51 et 52).

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.std_logic_arith.all;
4. use IEEE.STD_LOGIC_UNSIGNED.all;

5. entity compteur is
6.   generic (WDTH : integer :=4; STOP : integer :=10);
7.   port( CLK : in std_logic ;
8.          CE : in std_logic ;
9.       CLEAR : in std_logic;
10.     CEO : out std_logic;
11.    DOUT : out std_logic_vector(WDTH -1 downto 0));
12.end compteur;

13.architecture a1 of compteur is
14.  signal INT_DOUT : std_logic_vector(DOUT'range) ;
15.begin
16.  process(CLK, CLEAR) begin
17.    if (CLEAR='1') then
18.      INT_DOUT <= (others => '0');
19.    elsif (CLK'event and CLK='1') then
20.      if (CE='1') then
21.        if (INT_DOUT=STOP-1) then
22.          INT_DOUT <= (others => '0');
23.        else
24.          INT_DOUT <= (INT_DOUT + 1);
25.        end if;
26.      end if;
27.    end if;
28.  end process;
29.  CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and
   INT_DOUT(3) and CE;
30.  dout <= int_dout;
31.end;
```

```
32.library IEEE;
33.use IEEE.std_logic_1164.all;

34.entity gen_cnt is
35.  port(Clock : in std_logic;
36.       Reset : in std_logic;
37.         add : out std_logic_vector(3 downto 0);
38.         adu : out std_logic_vector(3 downto 0));
39.end gen_cnt;

40.architecture comporte of gen_cnt is

41.  COMPONENT compteur
42.    generic (WDTH : integer :=4; STOP : integer :=10);
43.    port( CLK : in std_logic ;
44.           CE : in std_logic ;
45.        CLEAR : in std_logic;
46.          CEO : out std_logic;
47.         DOUT : out std_logic_vector(WDTH -1 downto 0));
48.  END COMPONENT;

49.  signal ceo : std_logic;

50.begin
51.  cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset, open, add);
52.  cd4u : compteur generic map(4,10) port map(Clock, '1', Reset, ceo, adu);
53.end comporte ;
```

L'instanciation (l'appel) d'un composant (ligne 51 et 52) se déroule de la manière suivante :

Nom_instance : **nom_composant generic map(**paramètres génériques**) port map(**signaux interconnections**)**

a. Il faut d'abord écrire sur la ligne d'appel un **nom d'instance** (dans notre exemple, cd4d à la ligne 51 et cd4u à la ligne 52). Ce nom d'instance sert à différencier plusieurs instanciations d'un même composant. Ce nom est optionnel, mais le synthétiseur en créera un automatiquement si vous l'omettez.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

b. On indique ensuite le **nom du composant** à instancier. Il s'agit du nom donné à la ligne 41 qui doit être le même que le nom de l'entité ligne 5.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

c. On écrit ensuite les **paramètres génériques** s'il y en a. Grâce à ces paramètres, on peut écrire un modèle générique d'un composant (ici, un compteur) et définir ses caractéristiques au moment de son appel dans le design. Voyons notre exemple. L'entité `compteur` déclare deux paramètres, la largeur du compteur en bits WDTH et le nombre d'état du compteur STOP. On peut définir des valeurs par défaut pour ces deux paramètres (ici, 4 pour WDTH et 10 pour STOP) utilisés si on ne passe pas de valeurs au moment de son appel.

```
ligne 6. generic (WDTH : integer :=4; STOP : integer :=10);
```

Au moment de l'instanciation, on passe les deux valeurs dans l'ordre de la définition (ligne 6) après le mot clé « `generic map` ». Si on omet le generic map, les paramètres par défaut seront utilisés.

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);
```

Ces paramètres peuvent être utilisés soit dans la déclaration des entrées sorties (ici, on définit la largeur du bus de sortie) :

```
ligne 11. DOUT : out std_logic_vector(WDTH-1 downto 0));
```

soit dans l'architecture du composant (ici, la valeur d'arrêt du compteur) :

```
ligne 21. if (INT_DOUT=STOP-1) then
```

Les paramètres génériques apportent une grande flexibilité aux descriptions de composants et permettent l'écriture de bibliothèques standardisées. Vous noterez l'utilisation de l'attribut range.

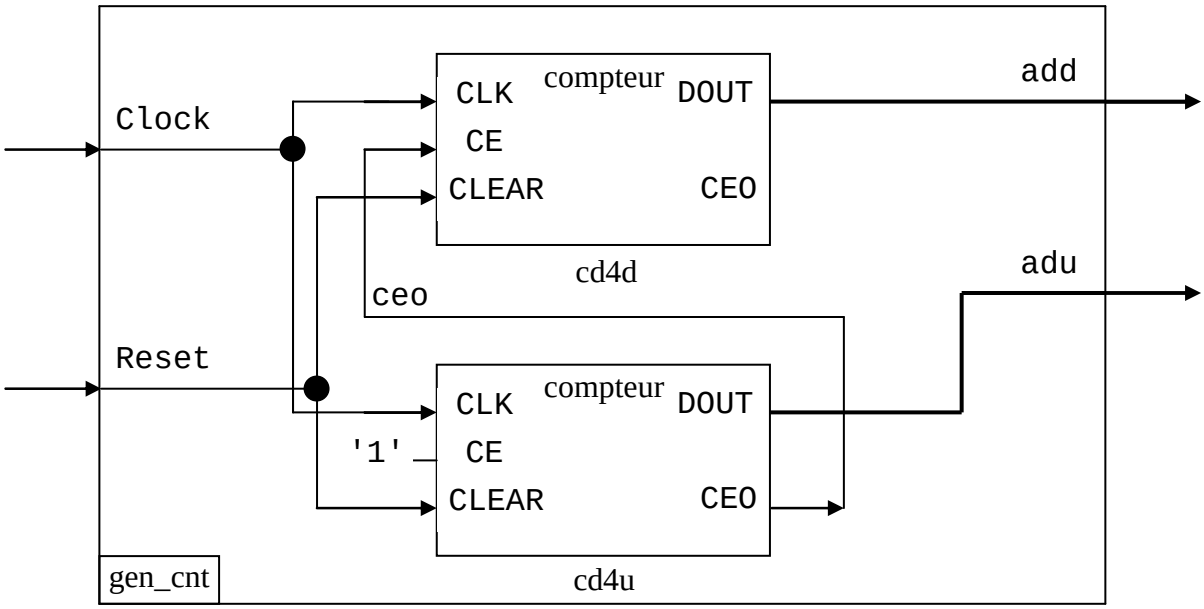
```
ligne 14. signal INT_DOUT : std_logic_vector(DOUT'range);
```

INT_DOUT a la même largeur que DOUT (de WDTH-1 à 0). L'attribut range simplifie l'écriture des modèles génériques.

d. Finalement, il faut relier les composants avec les autres éléments du design en utilisant des signaux d'interconnexions. Au moment de l'instanciation, on passe les noms des signaux après le mot clé « `port map` ».

```
ligne 51. cd4d : compteur generic map(4,10) port map(Clock,
ceo, Reset, open, add);

ligne 52. cd4u : compteur generic map(4,10) port map(Clock,
'1', Reset, ceo, adu);
```

Voici les liaisons que nous souhaitons réaliser :



Prenons l'exemple du signal d'horloge `Clock`. On appelle paramètre formel le nom de la broche interne CLK du composant `compteur` et paramètre réel le nom du signal `Clock` dans `gen_cnt`. Il va falloir relier le signal `Clock` de `gen_cnt` avec le signal interne `CLK` de `compteur`. Deux méthodes sont possibles : la méthode implicite et la méthode explicite. Dans la méthode implicite, il suffit de mettre dans la parenthèse qui suit le `port map` les signaux de `gen_cnt` en suivant l'ordre de l'entité de `compteur` :

```
port(CLK : in std_logic;
       CE : in std_logic;
    CLEAR : in std_logic;
      CEO : out std_logic;
     DOUT : out std_logic_vector(WDTH -1 downto 0));
```

A la ligne 51, les liaisons se font automatiquement dans l'ordre de l'entité :

```
cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset,
open, add);
```

Clock est reliée à CLK, ceo est relié à CE, … Quand la sortie d'un composant n'est reliée à rien, on utilise le mot-clef `open`.

Dans la méthode explicite, on indique pour chaque signal : `paramètre_formel => paramètre_réel`. On peut aussi utiliser la méthode explicite pour les paramètres génériques. Nous avons utilisé uniquement la méthode implicite jusqu'à maintenant. Voici les deux instanciations avec la méthode explicite :

```
cd4d : compteur
generic map(WDTH => 4,STOP => 10)
port map(CLK => Clock,
          CE => ceo,
       CLEAR => Reset,
         CEO => open,
        DOUT => add);
```

```
cd4u : compteur
generic map(WDTH => 4,STOP => 10)
port map(CLK => Clock,
          CE => '1',
       CLEAR => Reset,
         CEO => ceo,
        DOUT => adu);
```

La méthode explicite est plus longue à écrire, mais les signaux peuvent être mis dans le désordre. Cette solution peut être plus pratique quand le nombre de signaux à connecter devient élevé.

La description modulaire sans package devient relativement fastidieuse quand la taille du design augmente. D'où l'idée de regrouper les composants dans un paquetage (ou package) et dans un fichier séparé. Cela évite de redéclarer les composants dans le top level design et cela peut accélérer l'analyse du design au moment de la synthèse. En effet, avec plusieurs fichiers, le synthétiseur n'analyse que les fichiers qui ont été modifiés.

Dans la description modulaire avec package, la déclaration du composant `compteur` est effectuée dans un fichier séparé `cnt_pkg.vhd`. On trouve au début de ce fichier la déclaration des entrées sorties de tous les composants du package `gen_cnt_pkg` (lignes 3 à 12) puis la description complète de ces composants (entité et architecture, lignes 13 à 43).

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. package gen_cnt_pkg is

4.   COMPONENT compteur
5.     generic (WDTH : integer :=4; STOP : integer :=10);
6.     port( CLK : in std_logic;
7.            CE : in std_logic;
8.         CLEAR : in std_logic;
9.           CEO : out std_logic;
10.        DOUT : out std_logic_vector(WDTH -1 downto 0));
11.     END COMPONENT;

12.end gen_cnt_pkg;

13.library IEEE;
14.use IEEE.std_logic_1164.all;
15.use IEEE.std_logic_arith.all;
16.use IEEE.STD_LOGIC_UNSIGNED.all;

17.entity compteur is
18.  generic (WDTH : integer :=4; STOP : integer :=10);
19.  port( CLK : in std_logic;
20.         CE : in std_logic;
21.      CLEAR : in std_logic;
22.        CEO : out std_logic;
23.       DOUT : out std_logic_vector(WDTH -1 downto 0));
24.end compteur;

25.architecture a1 of compteur is
26.  signal INT_DOUT : std_logic_vector(DOUT'range) ;
27.begin
28.  process(CLK, CLEAR) begin
29.    if (CLEAR='1') then
30.      INT_DOUT <= (others => '0');
31.    elsif (CLK'event and CLK='1') then
32.      if (CE='1') then
33.        if (INT_DOUT=STOP-1) then
34.          INT_DOUT <= (others => '0');
35.        else
36.          INT_DOUT <= (INT_DOUT + 1);
37.        end if;
38.      end if;
39.    end if;
40.  end process;
41.  CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and
  INT_DOUT(3) and CE;
42.  dout <= int_dout;
43.end;
```

placé avant chaque composant du package

Le top level design se trouve dans le fichier `cnt.vhd` (les noms des fichiers utilisés sont sans importance). Le simulateur ou le synthétiseur va compiler le package dans la librairie par

défaut `work`. Pour pouvoir l'utiliser dans `gen_cnt`, il suffit de déclarer dans la liste des packages utilisés :

```
use work.gen_cnt_pkg.all;
```

Les composants peuvent ensuite être instanciés comme précédemment.

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use work.gen_cnt_pkg.all;

4. entity gen_cnt is
5.   port(
6.     Clock : in std_logic;
7.     Reset : in std_logic;
8.       add : out std_logic_vector(3 downto 0);
9.       adu : out std_logic_vector(3 downto 0));
10.end gen_cnt;

11.architecture comporte of gen_cnt is
12.  signal ceo : std_logic;
13.begin
14.  cd4d : compteur generic map(4,10) port map(Clock, ceo, Reset, open, add);
15.  cd4u : compteur generic map(4,10) port map(Clock, '1', Reset, ceo, adu);
16.end comporte ;
```

Il faut noter que :

a) Un package peut contenir un nombre quelconque d'éléments.
b) Un package peut contenir autre chose que des composants : par exemple, des définitions de types ou des fonctions.
c) On met en général dans un package des éléments utilisables dans des applications similaires.
d) Une description peut faire appel à plusieurs packages. Il suffit d'inclure autant de clause `use` qu'il y a de packages à utiliser.
e) Il est tout à fait possible de compiler un package dans une libraire autre que la librairie `work` et de le rendre accessible à d'autres développeurs. C'est nécessaire quand le projet nécessite plusieurs designers. C'est le cas notamment pour la librairie IEEE et les packages `std_logic_1164` ou `numeric_bit` par exemple.

La description modulaire avec package est la méthode normale d'organisation du travail quand on développe un projet en VHDL.

Une question se pose encore : quel type de signal doit-on utiliser dans l'entité d'un composant ?

En général, les paramètres génériques sont de type entiers ou bien `std_logic`. Pour les entrées-sorties, on utilise normalement des `std_logic` et `std_logic_vector`, mais pas de type `signed` ou `unsigned`. Cela simplifie l'instanciation des composants. Si nécessaire, on réalisera des conversions dans le composant pour travailler en signé ou en non-signé.

Il reste un dernier point à régler : le compteur n'est pas générique à cause du calcul du CEO. En effet, dans notre exemple, CEO vaut 1 si CE vaut 1 et si INT_DOUT = 9.

```
41. CEO <= INT_DOUT(0) and not INT_DOUT(1) and not INT_DOUT(2) and INT_DOUT(3) and CE;
```

Le paramètre générique STOP n'intervient pas dans le calcul de CEO. Deux modifications sont possibles. La plus évidente est le remplacement de la ligne 41 par :

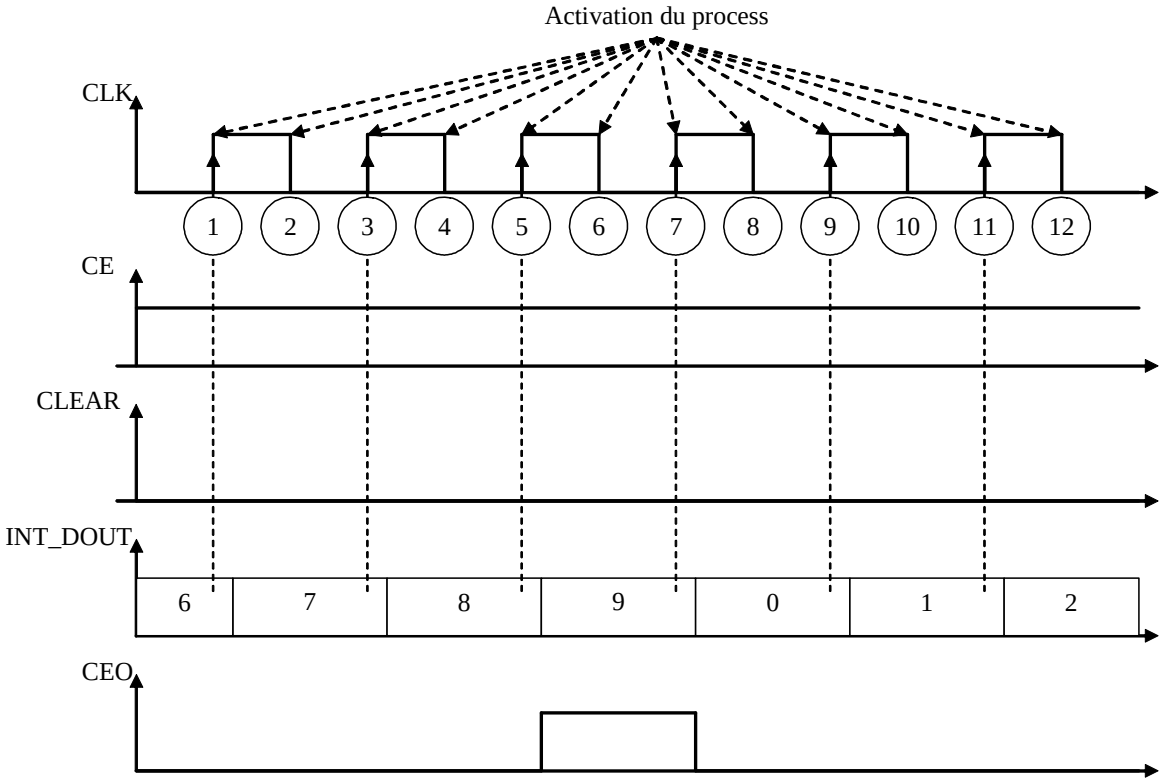
```
41. CEO <= CE when (INT_DOUT=STOP-1) else '0';
```

On obtient bien CEO = 1 si CE = 1 et INT_DOUT = STOP-1. La ligne étant hors process, la solution est purement combinatoire. On peut aussi essayer une solution séquentielle en modifiant le process de la manière suivante :

```
25.architecture a1 of compteur is
26.  signal INT_DOUT : std_logic_vector(DOUT'range);
27.begin
28.  process(CLK, CLEAR) begin
29.    if (CLEAR='1') then
30.      INT_DOUT <= (others => '0');
31.      CEO <= 0;
32.    elsif (CLK'event and CLK='1') then
33.      if (CE='1') then
34.        if (INT_DOUT=STOP-1) then
35.          INT_DOUT <= (others => '0');
36.        else
37.          INT_DOUT <= (INT_DOUT + 1);
38.        end if;
39.        if (INT_DOUT=STOP-2) then
40.          CEO <= 1;
41.        else
42.          CEO <= 0;
43.        end if;
44.      end if;
45.    end if;
46.  end process;
47.  dout <= int_dout;
48.end;
```

La compréhension est simple, mais les conditions sont un peu plus difficiles à comprendre. Pourquoi `if (INT_DOUT=STOP-2)` et pas `STOP-1`. Pour comprendre avec précision le problème, il suffit de tracer un chronogramme et de se rappeler que :

1. Le processus s'exécute à chaque changement d'état d'un des signaux auxquels il est déclaré sensible.
2. Les modifications apportées aux valeurs de signaux par les instructions prennent effet à la fin du processus.



Aux instants 2, 4, 6, 8, 10 et 12, le process est activé : comme CLEAR vaut 0, la première condition est fausse (ligne 29) et on passe au `elsif` (ligne 32). Il y a eu un événement sur CLK mais CLK vaut 0 donc la deuxième condition est fausse. On sort du process.

A l'instant 1, le process est activé : comme CLEAR vaut 0, on passe à la deuxième condition. Il y a eu un événement sur CLK et CLK vaut 1 donc la condition du `elsif` est vraie et on rentre dans la branche. Comme CE vaut 1, on va tester la valeur de INT_DOUT : mais attention, **c'est l'ancienne valeur de INT_DOUT que l'on va tester** (celle qui précède le front actif de l'horloge), c'est-à-dire 6. On suppose que STOP = 10. Le premier test, INT_DOUT = 9 est faux, donc on incrémente INT_DOUT mais **sa valeur ne change pas**

immédiatement : INT_DOUT vaut toujours 6. Le deuxième test, INT_DOUT = 8 est faux, donc CEO reste à 0. On sort du process et INT_DOUT passe à 7.

A l'instant 3, le process est activé. INT_DOUT vaut 7 et est incrémenté, CEO reste à 0. A la sortie du process, INT_DOUT passe à 8.

A l'instant 5, le process est activé. INT_DOUT vaut 8. Le premier test, INT_DOUT = 9 est faux, donc on incrémente INT_DOUT mais **sa valeur ne change pas** immédiatement : INT_DOUT reste à 8. Le deuxième test, INT_DOUT = 8 est vrai, donc CEO passera à 1 en sortant du process. On sort du process, INT_DOUT passe à 9 et CEO passe à 1.

A l'instant 7, le process est activé. INT_DOUT vaut 9. Le premier test, INT_DOUT = 9 est vrai, donc INT_DOUT passera à 0 en sortant du process : INT_DOUT reste à 9. Le deuxième test, INT_DOUT = 8 est faux, donc CEO passera à 0 en sortant du process. On sort du process, INT_DOUT passe à 0 et CEO passe à 0.

Aux instants 9 et 11, le process est activé. INT_DOUT sera incrémenté à la sortie du process et CEO reste à 0.

C'est ce principe qu'il faut comprendre pour savoir comment exprimer correctement les conditions dans un processus séquentiel.

5.2.2 Registre à décalage générique

Reprenons le registre à décalage vu au chapitre précédent et essayons de le rendre générique en réalisant un registre à N étages travaillant sur K bits.

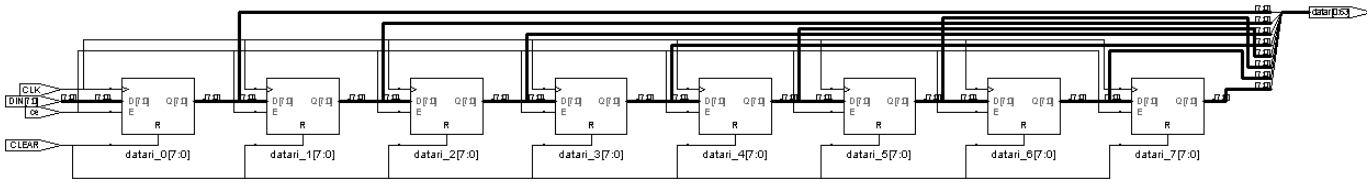
```
1. library IEEE;
2. use IEEE.std_logic_1164.all;

3. package tp4_pkg is
4.   TYPE data8x8 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);
5. end tp4_pkg;

6. library IEEE;
7. use IEEE.std_logic_1164.all;
8. use IEEE.std_logic_arith.all;
9. use IEEE.STD_LOGIC_UNSIGNED.all;
10.use work.tp4_pkg.all;
```

```
11.entity regMxN is
12.  generic (NbReg : integer :=8; NbBit : integer :=8);
13.  port( CLK : in std_logic ;
14.       CLEAR : in std_logic;
15.          CE : in std_logic;
16.         DIN : in std_logic_vector(NbBit -1 downto 0);
17.       datar : out data8x8);
18.end regMxN;

19.architecture RTL of regMxN is
20.  signal datari : data8x8;
21.begin
22.  process(CLK, CLEAR) begin
23.    if (CLEAR='1') then
24.      for i in 0 to 7 loop
25.        datari(i) <= (others => '0');
26.      end loop;
27.    elsif (CLK'event and CLK='1') then
28.      if (ce = '1') then
29.        for i in 1 to 7 loop
30.          datari(8-i) <= datari(7-i);
31.        end loop;
32.        datari(0) <= DIN;
33.      end if;
34.    end if;
35.  end process;
36.  datar <= datari;
37.end;
```

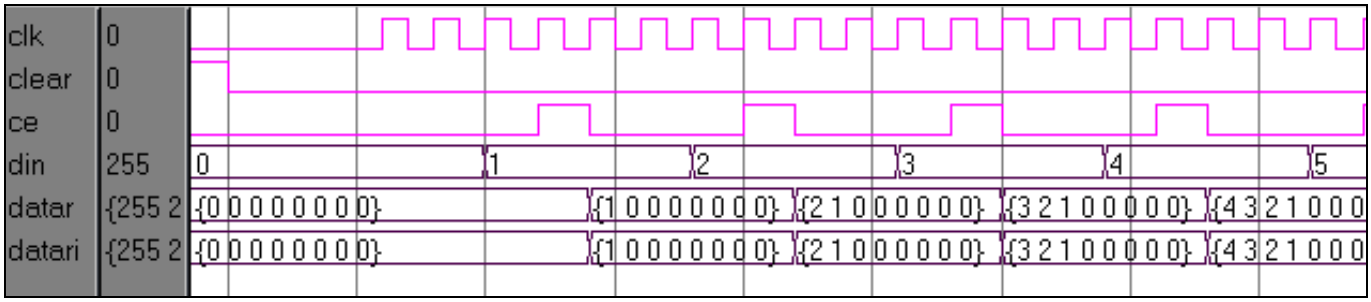
Le problème concernant la réalisation d'un registre N étages sur K bits, c'est qu'il va comporter N bus sur k bits en sortie. Il faut donc utiliser un tableau de N signaux de type `std_logic_vector(K-1 downto 0)`. Il est très difficile de rendre un tel tableau générique, d'autant que le type doit être connu avant de pouvoir l'utiliser dans l'entité. La solution la plus simple consiste à créer un nouveau type dans un package :

```
TYPE data8x8 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);
```

Puis à l'utiliser dans l'entité du registre. Vous noterez à la ligne 10 que l'on peut utiliser un élément du package à un autre endroit dans le même package. Le type data8x8 définit un tableau de 8 signaux de largeur 8 bits. L'initialisation de ce signal se fait à l'aide d'une boucle for (lignes 24 à 26). Le décalage du registre est réalisé aux lignes 28 à 33. Si CE vaut 1, alors on copie datari(6) dans datari(7), puis datari(5) dans datari(6), puis datari(4) dans datari(5), …, puis datari(0) dans datari(1). Il suffit ensuite de copier din dans datari(0) pour terminer le décalage. Le design regMxN réalise donc la fonction logique suivante :



Le chronogramme suivant montre l'évolution des signaux avec les valeurs par défaut (8,8) :



Bien sûr, ce composant n'est pas complètement générique puisque le tableau ne s'ajuste pas aux paramètres `NbReg` et `NbBit`. Il faut créer au démarrage un type dataNxK suffisamment grand pour contenir les valeurs. Les paramètres génériques ne peuvent résoudre tous les problèmes.

5.2.3 Conception avec plusieurs horloges ou avec plusieurs CE

5.2.3.1 Introduction

Plaçons-nous maintenant dans le cas d'un design qui comporte plusieurs parties fonctionnant à des rythmes différents. Il existe deux manières de traiter ce problème : la méthode localement synchrone (ou synchrone par bloc) avec autant d'horloges que de blocs travaillant à des fréquences différentes et la méthode totalement synchrone avec une seule horloge et plusieurs signaux de validation CE. Etudions plus en détail ces deux solutions :

1. Avec un circuit diviseur d'horloge, on peut créer à partir de l'horloge principale (généralement un oscillateur à quartz sur la carte) une nouvelle horloge qui sera utilisée dans le design. S'il y a dans le design plusieurs blocs qui travaillent à des fréquences différentes (on parle de domaines d'horloge), on créera autant d'horloges que de blocs, chaque bloc étant synchrone avec son horloge mais pas avec les horloges des autres blocs. Cette méthode a pour avantage de minimiser la consommation du circuit, car chaque bloc travaille à sa fréquence minimale. Son principal inconvénient est que le design n'est plus synchrone avec une seule horloge, mais synchrone par bloc. Tout le problème (très délicat dans le cas général) va consister à échanger de manière fiable des données entre les différents blocs, chaque bloc travaillant avec sa propre horloge (voir §4.6.4).
2. Il y a une autre manière de traiter le problème, la méthode totalement synchrone (fully synchronous). Tout le design travaille avec l'horloge du bloc le plus rapide, et on crée un signal de validation (CE_x) pour chacun des blocs avec un circuit générateur de CE. Ce signal de validation qui vaut 1 toutes les N périodes de l'horloge sera connecté aux

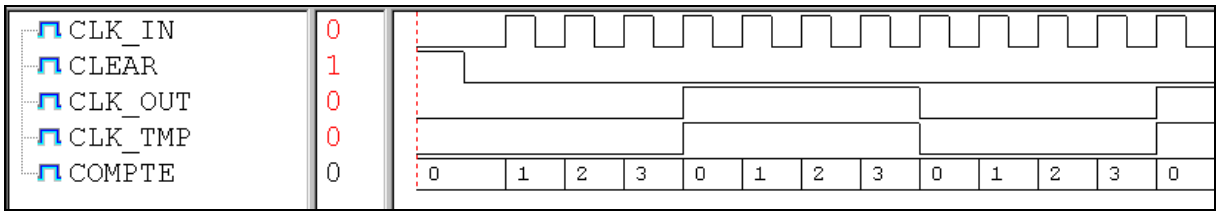
différentes bascules D des blocs. Les bascules seront donc activées à une cadence plus faible que l'horloge. Cette méthode a pour avantage de faciliter l'échange des données entre les blocs. En effet, tous les échanges sont synchrones puisqu'il n'y a plus qu'une seule horloge. Le premier inconvénient de cette méthode, c'est une consommation élevée puisque toutes les bascules du montage travaillent à la fréquence maximale. Le deuxième inconvénient, c'est que la contrainte de fréquence maximale va s'appliquer à tout le design, ce qui rend plus compliqué le placement-routage du circuit.

Voyons maintenant les composants utilisés par ces deux méthodes.

5.2.3.2 Circuit diviseur d'horloge

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.STD_LOGIC_UNSIGNED.all;

4. entity clk_div_N is
5.   generic (div : integer := 8); -- div doit être pair
6.   port (clk_in : in std_logic;
7.          clear : in std_logic;
8.        clk_out : out std_logic);
9. end clk_div_N;

10.architecture RTL of clk_div_N is
11.  signal clk_tmp : std_logic;
12.  signal compte : integer range 0 to (div/2)-1;
13.begin
14.  PROCESS (clk_in, clear) BEGIN
15.    if (clear = '1') then
16.      clk_tmp <= '0';
17.      compte <= 0;
18.    elsif (clk_in'event and clk_in='1') then
19.      if compte = ((div/2)-1) then
20.        compte <= 0;
21.        clk_tmp <= not clk_tmp;
22.      else
23.        compte <= compte + 1;
24.      end if;
25.    end if;
26.  END PROCESS;
27.  clk_out <= clk_tmp;
28.end;
```

A ce stade du cours, le fonctionnement de ce circuit diviseur d'horloge ne doit pas vous poser de problème de compréhension. Il ne fonctionne que pour des valeurs paires de `div`. Le signal `compte` évolue entre les valeurs 0 et div/2 – 1 (voyez sa déclaration à la ligne 12). Vous noterez qu'il s'agit d'un entier et pas d'un type std_logic. Cela permet une réalisation plus simple du diviseur. Quand `compte` atteint la valeur finale `div/2 – 1`, le signal `clk_tmp` est inversé. Le chronogramme suivant explique le fonctionnement du circuit avec une division par 8 (valeur par défaut) :

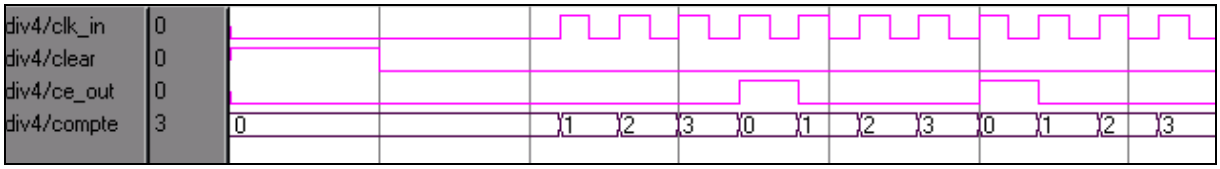


Si vous utilisez le type `std_logic`, les bascules sont toutes à l'état U (unknown) au démarrage. Vous devez obligatoirement prévoir une mise à zéro ou bien une mise à un pour pouvoir utiliser votre design. C'est un avantage de ce type vis à vis du type bit car il est proche du fonctionnement réel d'un circuit intégré. En effet, à la mise sous tension, les éléments de mémorisation (dont les bascules) sont dans un état indéterminé. **Vous ne devez jamais supposer que l'état de départ de vos bascules est connu quand vous concevez un design.** L'emploi du type `std_logic` vous y oblige, d'où son intérêt.

5.2.3.3 Circuit générateur de CE

```
1. library IEEE;
2. use IEEE.std_logic_1164.all;
3. use IEEE.STD_LOGIC_UNSIGNED.all;

4. entity gen_ce_div_N is
5.   generic (div : integer := 4);
6.   port (clk_in : in std_logic;
7.         clear : in std_logic;
8.        ce_out : out std_logic);
9. end gen_ce_div_N;

10.  architecture RTL of gen_ce_div_N is
11.    signal compte : integer range 0 to div-1;
12.  begin
13.    PROCESS (clk_in, clear) BEGIN
14.      if (clear = '1') then
15.        compte <= 0;
16.        ce_out <= '0';
17.      elsif (clk_in'event and clk_in = '1') then
18.        if (compte = div-1) then
19.          ce_out <= '1';
20.          compte <= 0;
21.        else
22.          ce_out <= '0';
23.          compte <= compte + 1;
24.        end if;
25.      end if;
26.    END PROCESS;
27.  end;
```

Le fonctionnement du montage est évident. ce_out vaut 0 sauf quand compte est égal à la valeur `div-1` (div = 4 dans le chronogramme suivant). Il passe alors à 1.



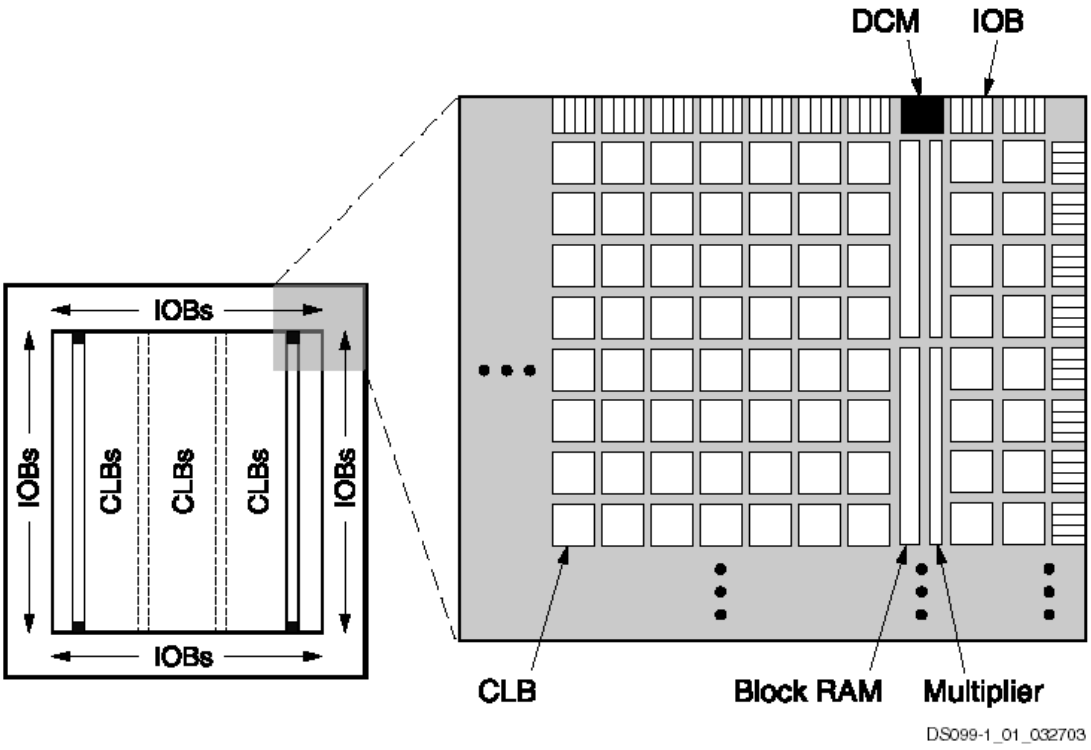
Une bascule D reliée à l'horloge clk_in dont le chip enable (CE) serait connecté à ce_out serait donc activée un front d'horloge sur 4.

6 Un exemple de FPGA : la famille Spartan-3

Ce paragraphe détaille la structure interne des FPGA de la famille Spartan-3 fabriqués par la société XILINX.

6.1 Caractéristiques générales

La figure suivante représente la structure simplifiée de la famille Spartan-3. On reconnaît là, une structure de type prédiffusé.



La famille Spartan-3 est une famille de FPGA CMOS SRAM faible coût basée sur la famille Virtex-II (FPGA complexité élevée). La liste suivante résume ses caractéristiques :

- matrice de blocs logiques programmables ou CLB (Configurable Logic Block),
- blocs d'entrée/sortie programmables ou IOB (Input Ouput Block) dont le nombre varie suivant le type de boîtier (QFP ou BGA). Ils supportent 23 standards d’E-S plus le contrôle des impédances d’entrée et de sortie via DCI,
- réseau de distribution d'horloge avec une faible dispersion via les DCM,
- des blocs RAM 18 kbits,
- des multiplieurs 18 bits x 18 bits,
- de nombreuses ressources de routage.

Cette famille comprend 8 membres allant d'une capacité de 1728 à 74880 cellules logiques (une LUT associée à une bascule D) :

	XC3S50	XC3S200	XC3S400	XC3S1000	XC3S1500	XC3S2000	XC3S4000	XC3S5000
System Gates	50K	200K	400K	1000K	1500K	2000K	4000K	5000K
Logic Cells	1,728	4,320	8,064	17,280	29,952	46,080	62,208	74,880
Block RAM Blocks	4	12	16	24	32	40	96	104
Block RAM Bits	72K	216K	288K	432K	576K	720K	1,728K	1,872K
DCMs	2	4	4	4	4	4	4	4
IO Standards	23	23	23	23	23	23	23	23
Max Single Ended I/O	124	173	264	391	487	565	712	784
Max Differential I/O	56	76	116	175	221	270	312	344

Les boîtiers suivants sont disponibles :

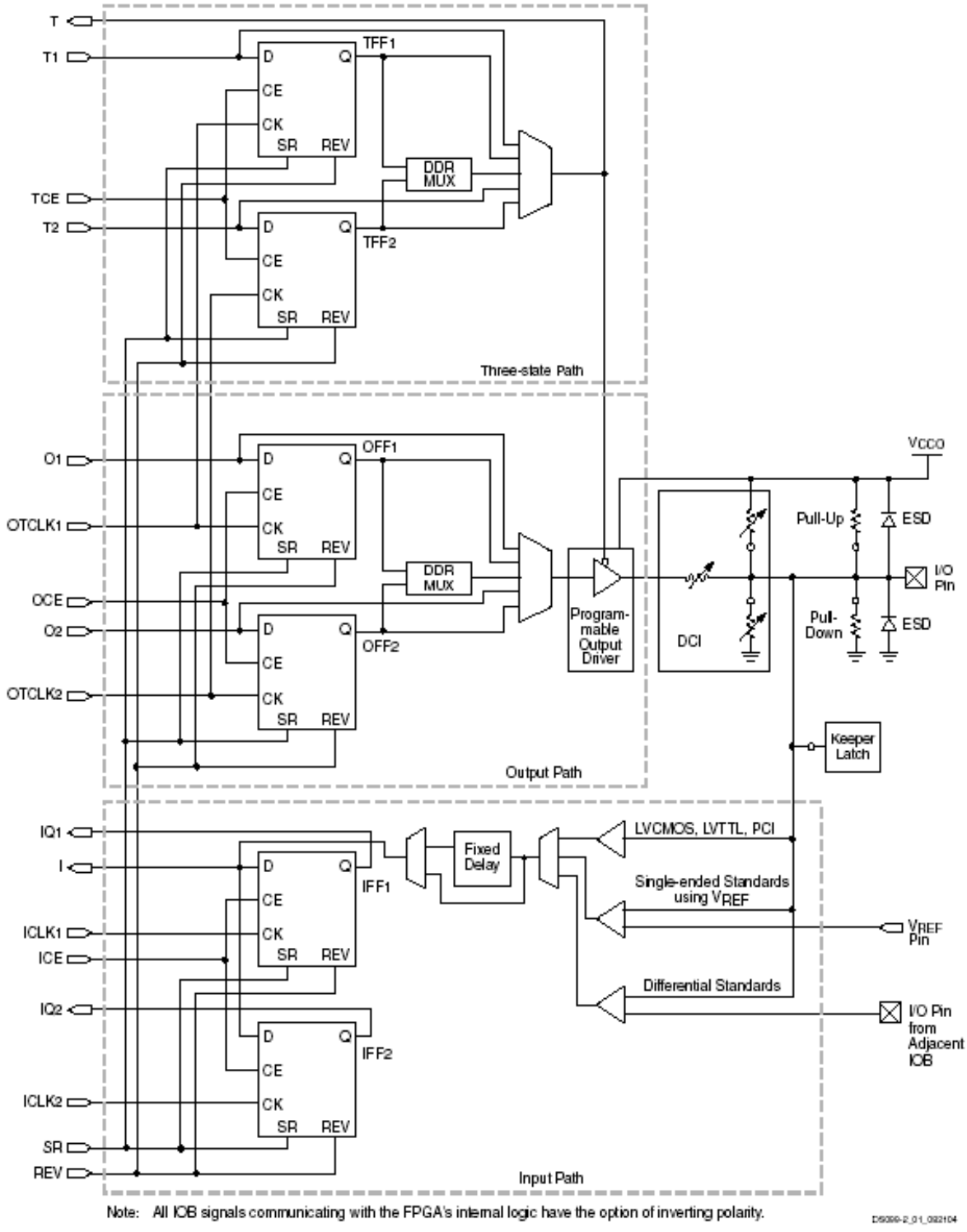
Device	XC3S50	XC3S200	XC3S400	XC3S1000	XC3S1500	XC3S2000	XC3S4000	XC3S5000
VQ100	63	63						
TQ144	97	97	97					
PQ208	124	141	141					
FT256		173	173	173				
FG456			264	333	333			
FG676				391	487	489		
FG900						565	633	633
FG1156							712	784

Le circuit utilisé pour les TP est le XC3S200-FT256-4C. Nous allons maintenant reprendre chaque point clé de ce FPGA plus en détail.

6.2 Blocs d'entrée-sortie (IOB)

6.2.1 Généralités

Des blocs d'entrée-sortie (IOB) configurables sont répartis sur toute la périphérie du boîtier. Chaque IOB assure l'interface entre une broche d'entrée/sortie du boîtier et la logique interne. La figure suivante représente le schéma bloc simplifié d'un IOB.



Standards d'entrée/sortie supportés (en différentiel, il faut utiliser une paire d'IOB) :

Single Eended I/O	Differential I/O	Impedance Matching I/O
LVTTL, LVCMOS, SSTL2 I&II, SSTL18 I, HSTL18 I,II&III, GTL, GTL+, PCI33	LVDS, LVDS EXT, LDT, ULVDS, BLVDS, RSDS	DCI for LVCMOS, DCI for LVDS & LVDS-EXT, DCI for HSTL, DCI for SSTL, DCI for GTL & GTL+

6.2.2 Caractéristiques d’entrée

Le signal sur la broche d'entrée (I/O pin) est amené vers les CLB soit directement via le signal I, soit à travers une paire de bascules D (ou de latch) via IQ1 et IQ2. Les caractéristiques de l'entrée de l’IOB sont les suivantes :

- diodes de protection ESD,
- résistance de "pull-up" ou "pull-down",
- Contrôle de l’impédance d’entrée (DCI),
- 23 standards d’entrée (différentiels ou non),
- horloge indépendante de la sortie,
- Un délai de quelques ns peut-être inséré dans le chemin de la donnée d'entrée pour compenser le retard de l’horloge,
- Support du Double Data Rate pour écrire dans les SDRAM DDR.

6.2.3 Caractéristiques de sortie

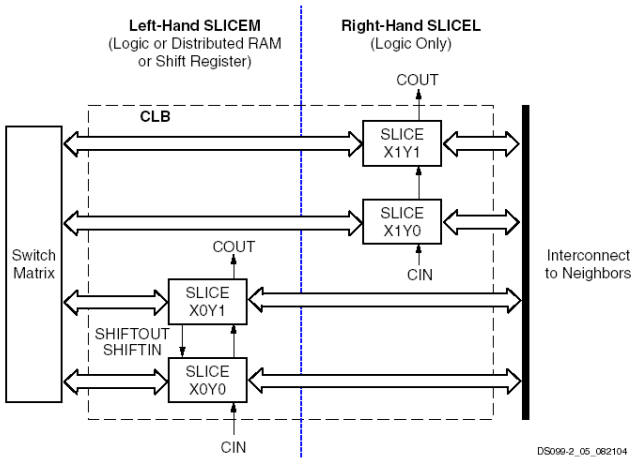
Le signal de sortie peut être optionnellement inversé à l’intérieur de l’IOB et sortir directement sur la broche ou bien être mis en mémoire par une paire de bascules D actives sur un front. Les caractéristiques de la sortie d'un IOB sont les suivantes :

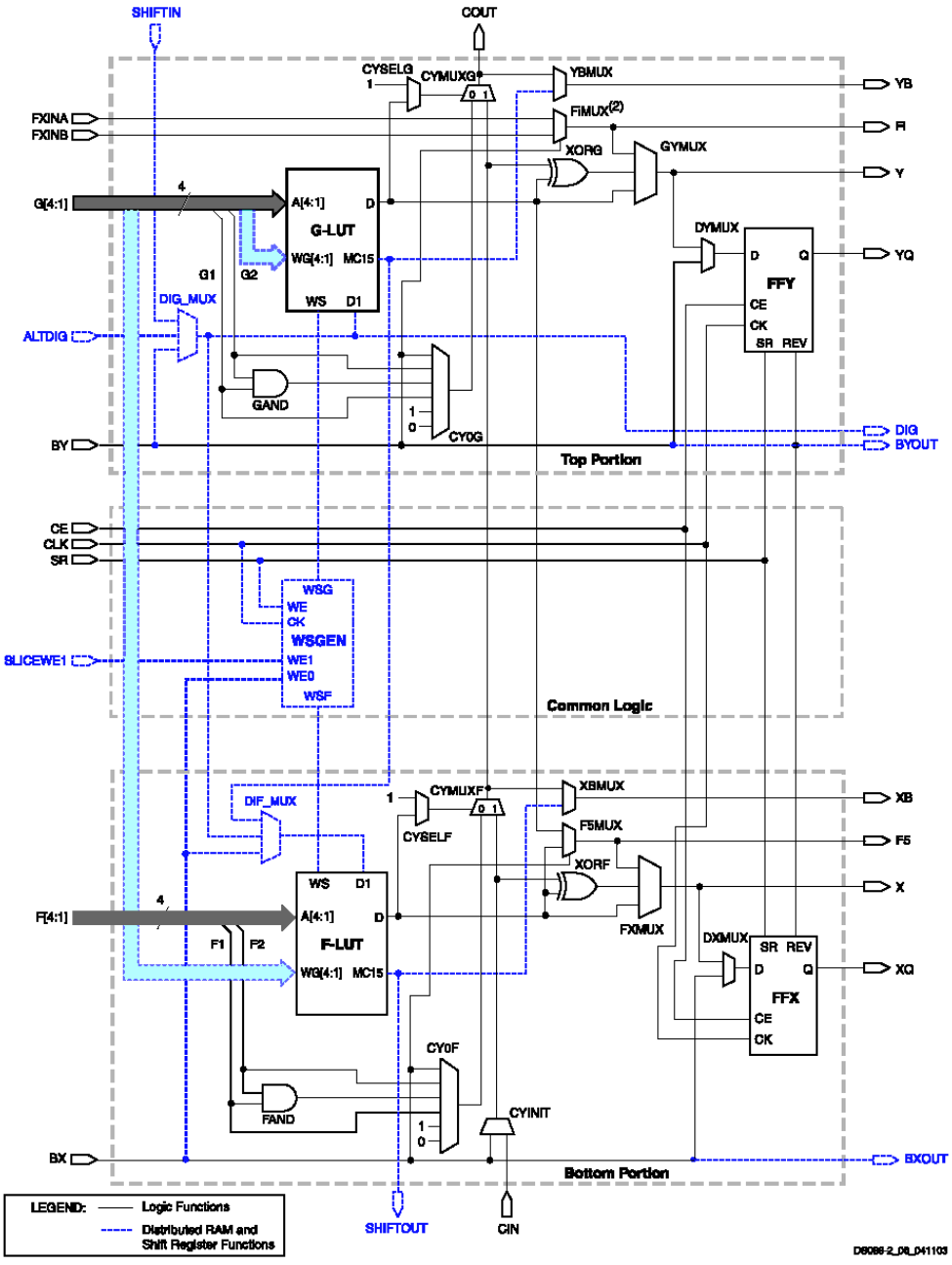
- buffer 3 états piloté par une paire de bascules D,
- sortie collecteur ouvert,
- 23 standards de sortie (différentiels ou non),
- contrôle de "slew-rate" (rapide ou lent),
- Contrôle de l’impédance de sortie (DCI),
- Support du Double Data Rate pour lire dans les SDRAM DDR.
- sortance de 24 mA max.

6.3 Bloc logique configurable (CLB)

6.3.1 Généralités

Le CLB est l'élément fonctionnel de base de ce FPGA. Sa programmation permet à l'utilisateur de réaliser des fonctions logiques combinatoires ou séquentielles. Le schéma bloc simplifié d'un CLB est représenté à la page suivante. Il est constitué de 4 SLICES, qui sont eux-mêmes formés de deux cellules logiques.


Left-Hand SLICEM
(Logic or Distributed RAM
or Shift Register)
Right-Hand SLICEL
(Logic Only)
COUT
CLB
SLICE
X1Y1
SLICE
X1Y0
Switch
Matrix
COUT
CIN
Interconnect
to Neighbors
SLICE
X0Y1
SHIFTOUT
SHIFTIN
SLICE
X0Y0
CIN
DS099-2_05_082104


SHIFTIN
COUT
CYSELG
CYMUXG
YBMUX
YB
FXINA
FXINB
FiMUX(2)
Fi
XORG
GYMUX
Y
G[4:1]
A[4:1]
D
G-LUT
WG[4:1]
MC15
WS
D1
G1
G2
DYMUX
YQ
FFY
CE
CK
SR
REV
DIG_MUX
ALTDIG
GAND
CY0G
DIG
BYOUT
BY
Top Portion
CE
CLK
SR
WSG
WE
CK
WSGEN
WE1
WE0
WSF
SLICEWE1
Common Logic
DIF_MUX
CYMUXF
XBMUX
XB
CYSELF
F5MUX
F5
XORF
X
F[4:1]
F-LUT
F1
F2
FXMUX
DXMUX
XQ
FFX
CY0F
FAND
CYINIT
BX
BXOUT
Bottom Portion
LEGEND:
Logic Functions
Distributed RAM and
Shift Register Functions
SHIFTOUT
CIN

Un slice (figure précédente) est constitué essentiellement de 2 générateurs de fonctions (LUT) F et G et de 2 bascules D, FFX et FFY.

6.3.2 Générateurs de fonctions

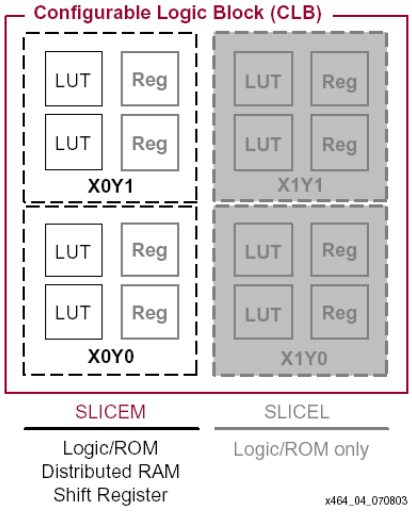
Les deux générateurs de fonctions F et G peuvent réaliser chacun n'importe quelle fonction combinatoire de 4 variables. En combinant les LUT des différents slices du CLB, il est aussi possible de réaliser des multiplexeurs à grand nombre d'entrées : par exemple, un mux 4:1 dans un slice, un mux 16:1 dans un CLB ou encore un mux 32:1 dans deux CLB. La polyvalence du CLB est la meilleure manière d'améliorer la vitesse de fonctionnement du système à réaliser.

6.3.3 Bascules

Ces générateurs de fonctions peuvent être connectés directement vers les sorties du slice (sorties X et Y), ou bien être mis en mémoire par deux bascules D (sorties XQ et YQ). Ces deux bascules ont la même horloge (CLK), le même signal de validation (CE) et la même logique de mise à 0 ou de mise à 1 asynchrone (SR). L'état de sortie de la bascule à la mise sous tension est programmable. Les deux bascules peuvent être utilisées indépendamment (entrée sur BX et BY) ou à la suite des générateurs de fonctions.

6.3.4 Configuration en ROM, RAM et registre à décalage

Tous les générateurs de fonctions F et G du CLB peuvent être utilisés comme des ROM. En effet, chaque LUT est une ROM 16 bits et on peut donc former au maximum une ROM 128x1 dans un CLB. Chaque LUT des 2 Slices de gauche du CLB peut aussi être programmée en RAM (simple et double port) ainsi qu'en registre à décalage.



Ainsi, on peut trouver au maximum dans un CLB :

- 4 mémoires 16x1 bits synchrones simple port,
- 2 mémoires 32x1 bits synchrones simple port,
- 1 mémoire 64x1 bits synchrone simple port,
- 2 mémoires 16x1 bits synchrones double port,
- 1 registre à décalage 64 bits.

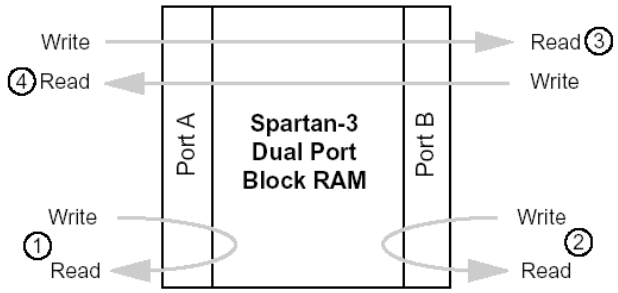
Ces mémoires sont très rapides et elles ont l'avantage d'être situées au cœur de la fonction à réaliser. Il n'y a donc pas de délais de routage. La mémoire synchrone est avantageuse car elle est plus rapide et plus facilement exploitable que la mémoire asynchrone. La mémoire double port possède deux ports (adresse, donnée, contrôle) indépendants. Elle peut être utilisée pour réaliser des FIFO. Le contenu de ces RAM ou de ces ROM peut être initialisé à la mise sous tension.

6.3.5 Logique de retenue rapide

Chaque slice contient une logique arithmétique dédiée pour générer rapidement une retenue (carry). Cette logique dédiée accélère grandement toutes les opérations arithmétiques telles que l'addition, la soustraction, l'accumulation, la comparaison... Elle accélère aussi la vitesse de fonctionnement des compteurs. Chaque slice peut être configuré comme un additionneur 2 bits avec retenue qui peut être étendu à n'importe quelle taille avec d'autres CLB. La sortie retenue (COUT) est passée au CLB se trouvant au-dessus. La retenue se propage en utilisant une interconnexion directe.

6.4 Block RAM (SelectRAM)

Tous les FPGA de la famille Spartan-3 incorporent des blocs de mémoire RAM 18kbits synchrones simple ou double ports (de 4 à 104 suivant la taille du circuit). Les 4 modes suivants sont possibles :



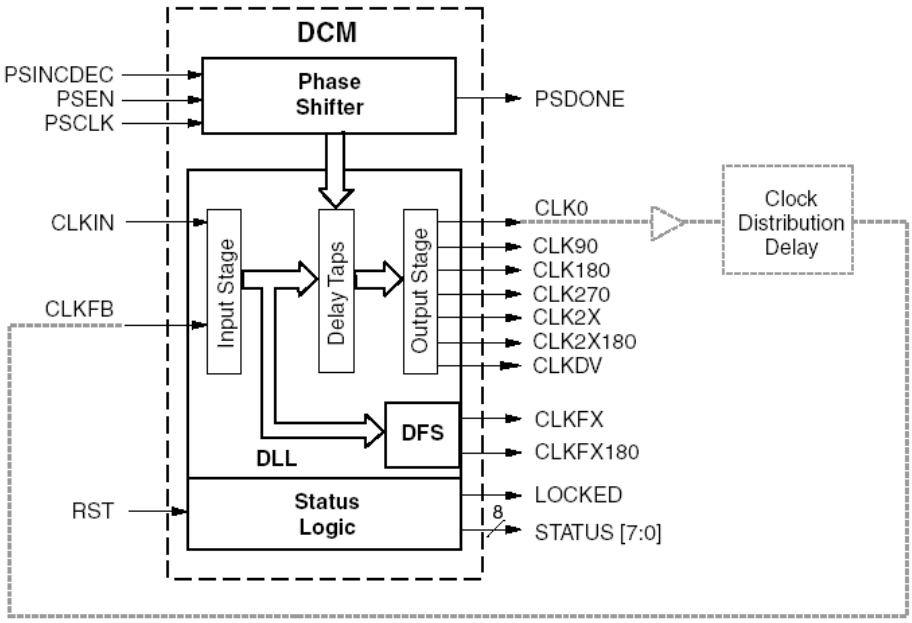
Chaque BlockRAM peut être configuré dans les modes : 16kx1, 8kx2, 4kx4, 2kx8, 1kx16 et 512x32 en simple port ou double ports.

6.5 Multiplieurs dédiés

Associé à chaque BlockRAM, on trouve un multiplieur 18x18 = 36 bits signé en complément à 2 (ou 17x17 = 34 non signé). On les utilisent surtout pour effectuer des opérations de traitement du signal, mais ils peuvent aussi être utilisés avec profit pour réaliser des opérations logiques (comme un décalage en un coup d'horloge par exemple).

6.6 Gestionnaire d'horloges

Le FPGA possède un gestionnaire d'horloge (Digital Clock Manager : DCM) particulièrement élaboré. Il permet par exemple de créer des horloges décalées en phase (pour piloter des DDR SDRAM par exemple), ou encore il élimine le skew (décalage des arrivées d'horloge sur les bascules D) des horloges dans le FPGA ou bien encore il permet de synthétiser des horloges avec des rapports (M x Fin) / D (avec M entier compris entre 2 et 32 et D entier compris entre 1 et 32). Vous pouvez par exemple créer une horloge sur CLKFX dont la fréquence est égale à 11/7 de CLKIN.

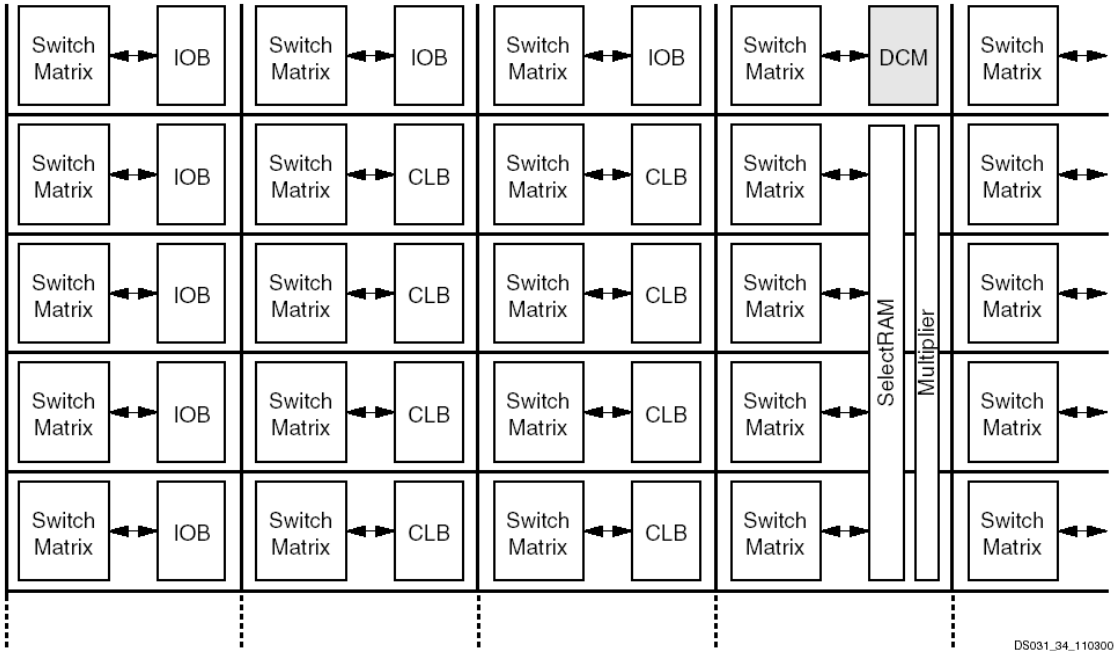


D'autre part, le FPGA possède 8 buffers spéciaux (BUFG) pour distribuer les horloges dans le circuit.

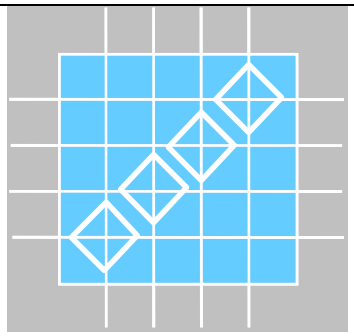
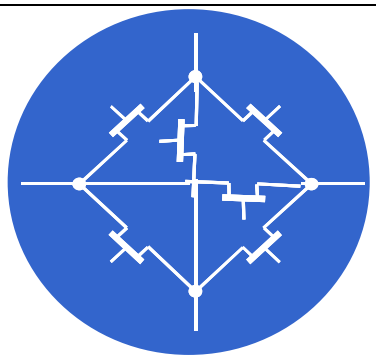
6.7 Ressources de routage et connectivité

6.7.1 Généralités

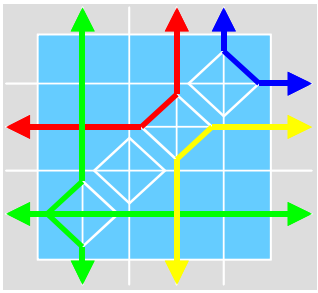
Les FPGA de la série Spartan-3 disposent d'un nombre important de ressources de routage et de connectivité, ce qui leur confère une très grande souplesse d'utilisation. Toutes les connexions sont constituées de segments métalliques reliés par des matrices de contacts programmables (Programmable Switch Matrix ou PSM). Chaque élément vu précédemment est relié à une ou plusieurs PSM.



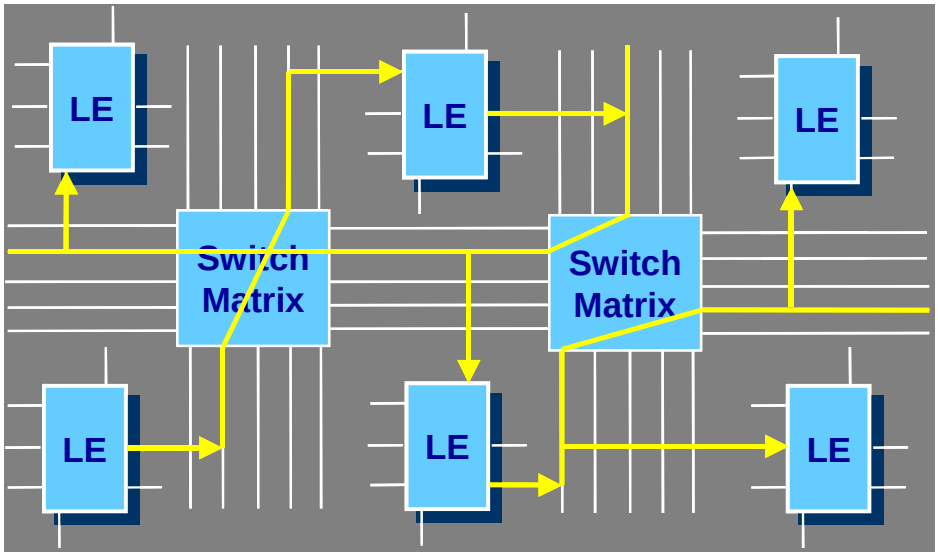
Chaque PSM est constituée de transistors permettant d'établir une connexion entre lignes horizontales et verticales comme le montre le schéma suivant.

	
PSM	6 transistors par points de connexion

Après programmation, la PSM permet par exemple la configuration suivante :



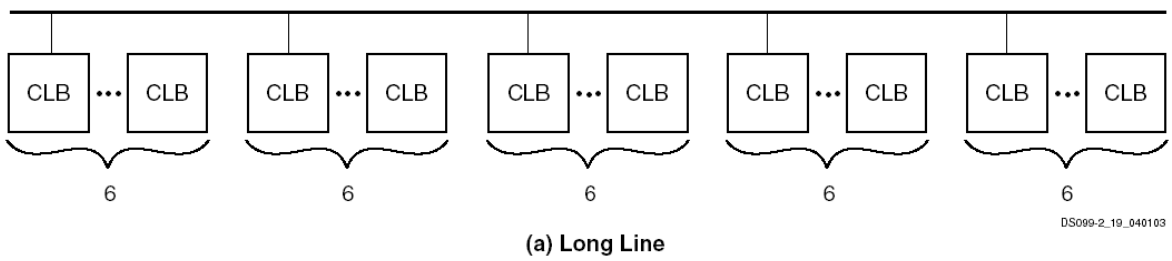
L'objectif final est de relier les différents éléments logiques (LE) entre eux comme sur la figure suivante :



6.7.2 Routage hiérarchique

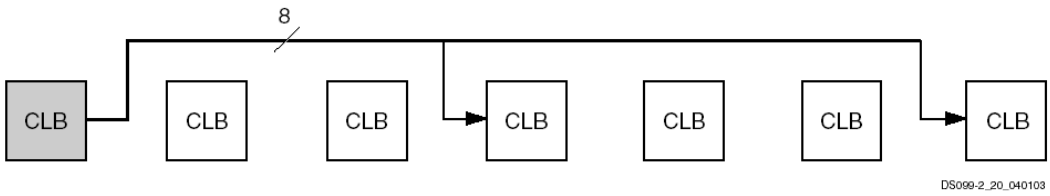
Hélas, chaque passage par une PSM introduit un délai de propagation qui est très pénalisant pour les liaisons à longue distance. Aussi les interconnexions ont été spécialisées en fonction de la distance à parcourir. 4 types d'interconnexion sont disponibles :

- Les longues lignes qui connectent un CLB sur 6 et donc sautent 5 PSM sur 6.



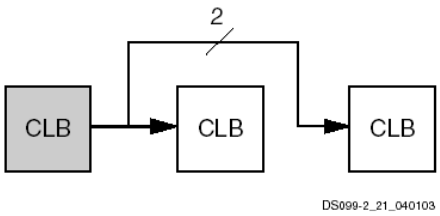
(a) Long Line

- Les lignes triples qui connectent un CLB sur 3 et donc sautent 2 PSM sur 3.



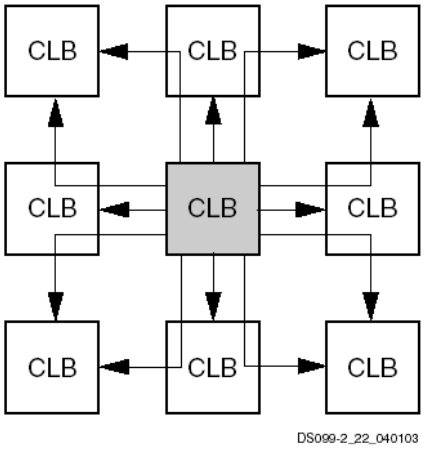
(b) Hex Line

- Les lignes doubles qui connectent un CLB sur 2 et donc sautent 1 PSM sur 1.



(c) Double Line

- Les lignes directes qui connectent un CLB avec ses voisins immédiats.

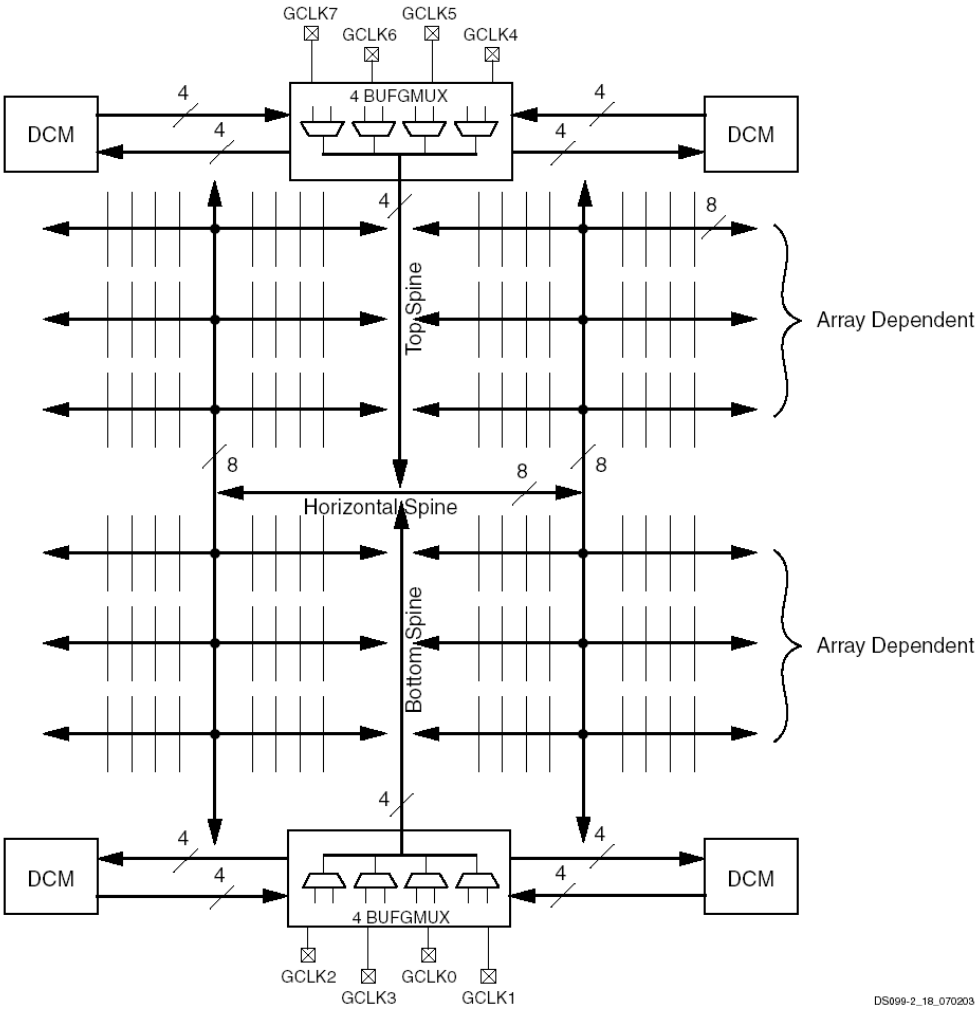


(d) Direct Lines

- Des lignes dédiées. Ce sont des connexions fixes utilisées pour propager les retenues, cascader les registres à décalage formés dans les slices à gauche du CLB,... Il existe aussi des lignes dédiées pour propager les horloges.

6.7.3 Lignes dédiées d'horloge

Il existe 8 entrées externes pour les horloges globales nommées GCLK0 à GCLK7. Il y a aussi 8 multiplexeurs (2:1) d'horloge globale BUFGMUX. Ces BUFGMUX acceptent en entrée soit une horloge globale, soit une sortie de DCM, soit un signal créé à l'intérieur du FPGA. La sortie des BUFGMUX attaque le réseau de distribution d'horloge. Une horloge globale dans un design peut passer soit par un BUFGMUX, soit par un BUFG (Global Clock Buffer).



Pour minimiser la puissance dynamique dissipée, les lignes d'horloge non utilisées sont désactivées.

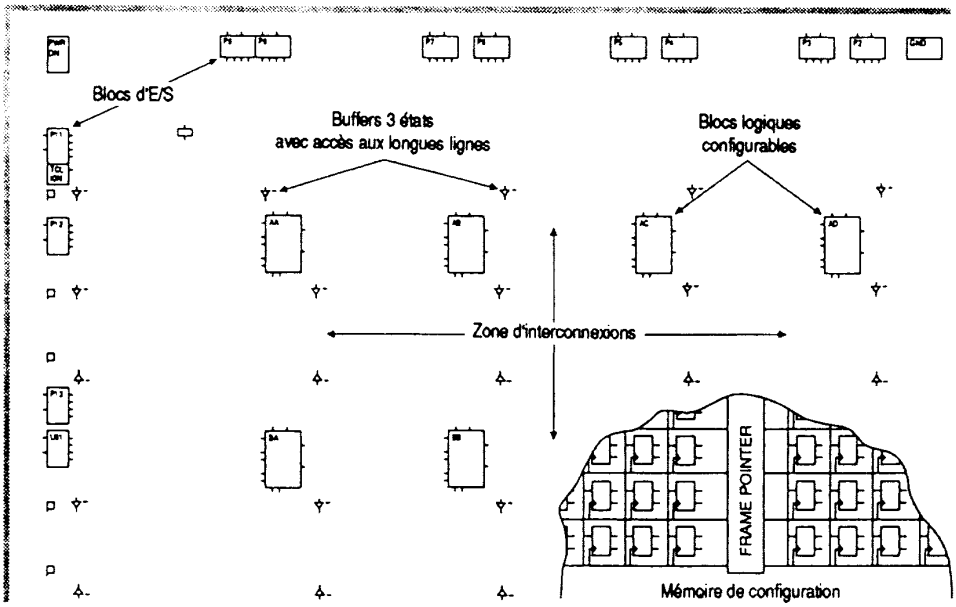
6.8 Configuration

Il y a trois types de broches sur un FPGA :

- Les broches dédiées en permanence. Ce sont les broches d'alimentation et de masse (un couple (VCC, GND) par coté), les buffers d'horloge...
- Les broches utilisables exclusivement par l'utilisateur pour réaliser son système.

- Les broches utilisables par l'utilisateur mais qui ont un rôle durant la configuration du circuit.

La configuration est le processus qui charge le design dans la SRAM afin de programmer les fonctions des différents blocs et de réaliser leurs interconnexions.

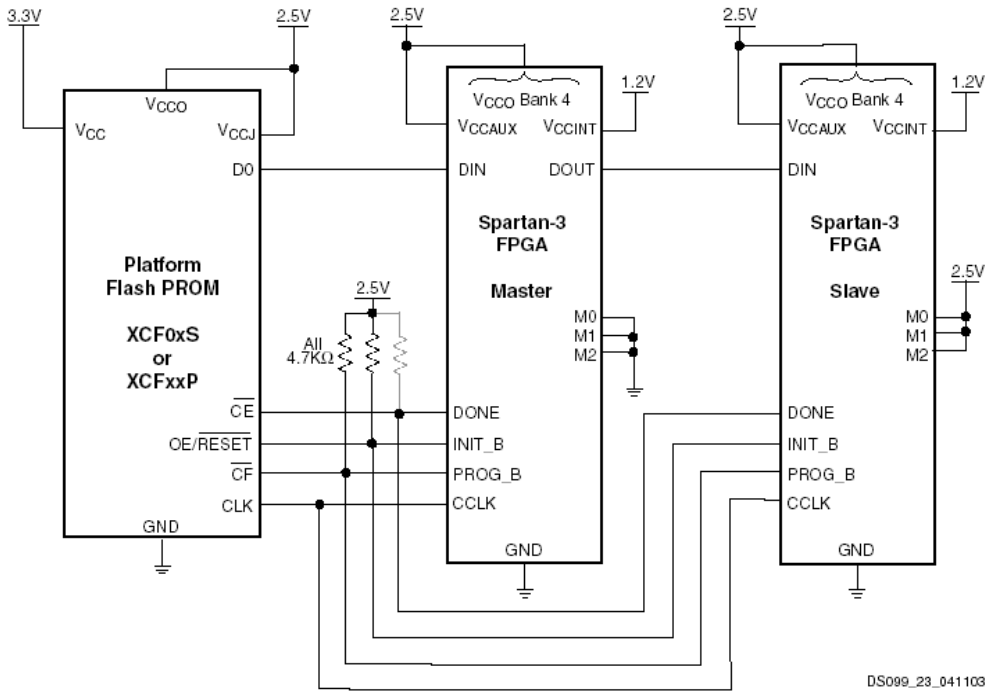


On voit, sur la figure ci-dessus, qu'il y a sous la logique dédiée à l'application une mémoire de configuration qui contient toutes les informations concernant la programmation des CLB et des IOB ainsi que l'état des connexions. Cette configuration est réalisée, à la mise sous tension, par le chargement d'un fichier binaire dont la taille varie en fonction du nombre de portes du circuit :

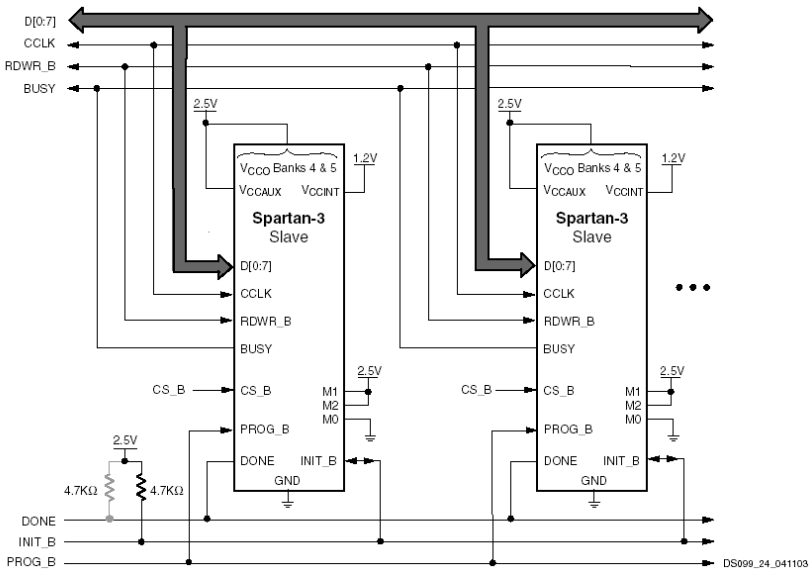
Device	File Sizes	Xilinx Platform Flash PROM	
		Serial Configuration	Parallel Configuration
XC3S50	439,264	XCF01S	XCF08P
XC3S200	1,047,616	XCF01S	XCF08P
XC3S400	1,699,136	XCF02S	XCF08P
XC3S1000	3,223,488	XCF04S	XCF08P
XC3S1500	5,214,784	XCF08P	XCF08P
XC3S2000	7,673,024	XCF08P	XCF08P
XC3S4000	11,316,864	XCF16P	XCF16P
XC3S5000	13,271,936	XCF16P	XCF16P

La configuration se fait en 4 étapes. A la mise sous tension, la mémoire de configuration est effacée puis le circuit est initialisé. Ensuite a lieu la configuration qui est suivie du démarrage du circuit programmé. Un CRC (« Cyclic Redundancy Check ») contenu dans le fichier de configuration permet au circuit de vérifier l'intégrité des données au moment du chargement. Le circuit peut se configurer en série (bit par bit) ou en parallèle (octet par octet). Il peut être maître ou bien esclave en cas de configuration de plusieurs boîtiers. En mode maître, c'est lui qui fournit les signaux nécessaires à sa propre configuration mais il peut aussi configurer un ou plusieurs circuits esclaves montés en cascade (daisy chain). Le fichier de configuration contient successivement tous les fichiers de configuration des circuits à programmer. Le mode parallèle esclave correspond au mode périphérique microprocesseur. Les principaux modes sont les suivants :

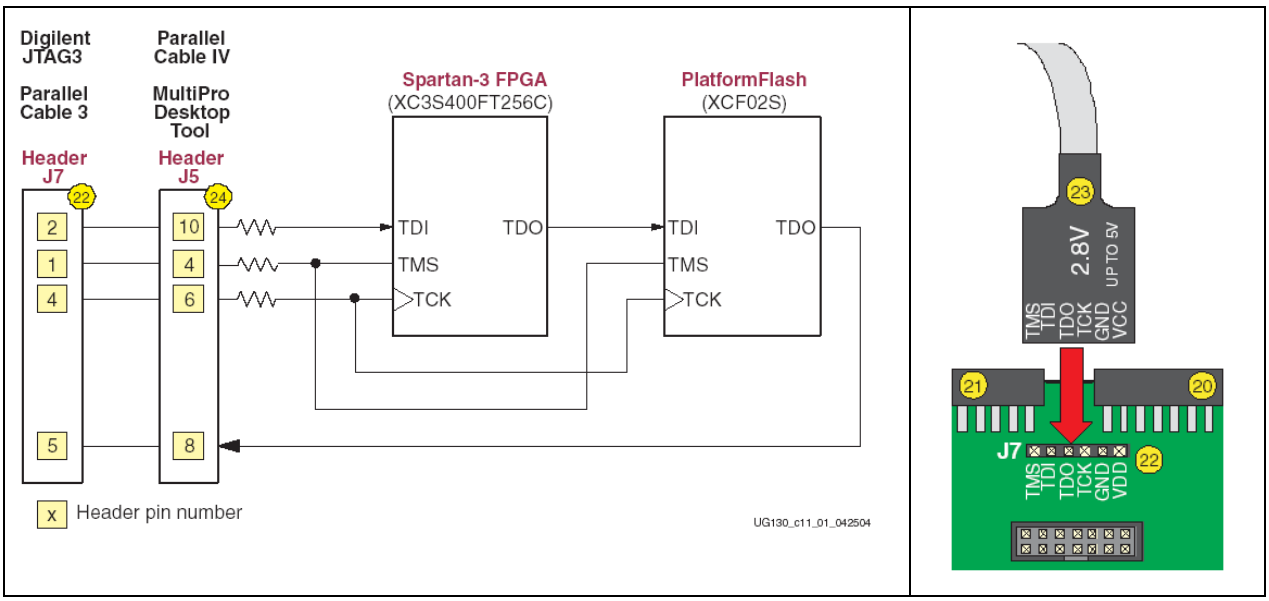
- En mode série maître, il faut utiliser une mémoire Flash série (XCF02S dans notre maquette). On s'en sert en phase de production car c'est celui qui nécessite le moins de câblage. Le FPGA génère une horloge CCLK pour lire les bits en série.



- En mode parallèle esclave, c'est un microcontrôleur externe qui charge la configuration octet par octet. L'horloge est fournie par le microcontrôleur.

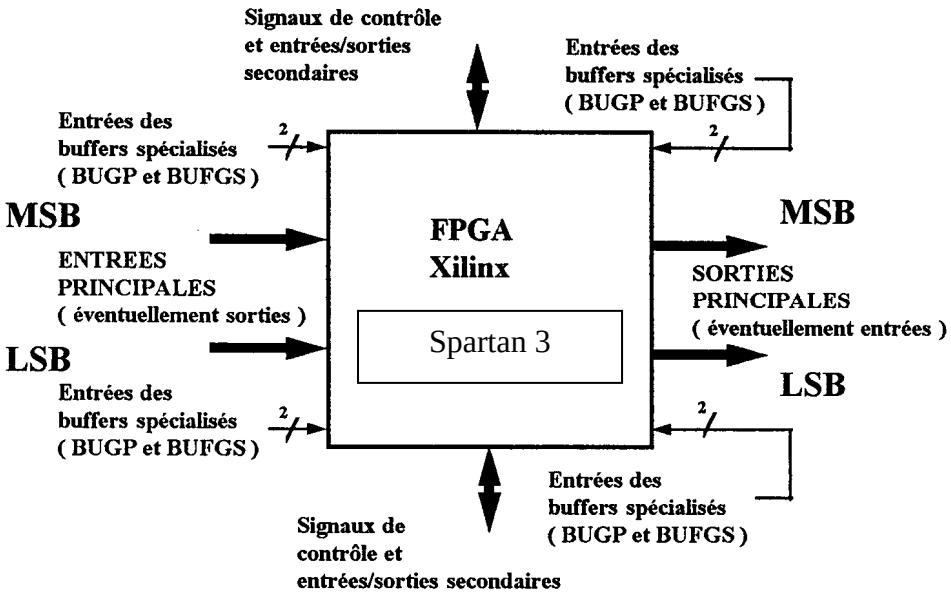


- En mode JTAG. La prise JTAG est constituée de 6 fils : VCC, GND, TDI, TDO, TMS et TCK. Conçu à l'origine pour le test de carte équipée, le JTAG est utilisé chez tous les fabricants de FPGA et d'EPLD pour la programmation In Situ (ISP) de leurs circuits y compris les PROM Flash série. Les circuits sont montés en Daisy Chain et on fabrique à faible coût un cordon qui se branche sur le port parallèle du PC. A l'aide d'Impact (chez Xilinx), on identifie les circuits de la chaine JTAG, puis on leur associe le fichier de configuration correspondant. Le téléchargement se fait alors en un clic de souris. Il est possible de vérifier la programmation du circuit en utilisant le ReadBack. L'horloge JTAG (TCK) est fournie par le cordon JTAG, donc par le PC.



6.9 Méthodologie de placement

Compte tenu de la structure du composant, il est préférable de placer les broches d'entrées/sorties de la manière suivante :



Le point critique, ce sont les retenues qui montent dans le circuit, ce qui implique que les additionneurs ont forcément le poids faible en bas et les bits dans l'ordre croissant en montant. Les bus à l'entrée du FPGA doivent donc être mis dans l'ordre, LSB en bas et MSB en haut. Ce point est toutefois bien moins critique que par le passé depuis que l'on peut effectuer des changements dans l'affectation des broches du composant avec une faible pénalité temporelle. Ce qui est vital quand on fabrique une carte avec des FPGA, c'est de placer :

1. Les alimentations,
2. les masses,
3. Les entrées d'horloges,
4. Les broches de configuration,
5. Les bus (dans le bon ordre de préférence s'il y a des problèmes de performance).

Le reste des fils peut être placé pour faciliter le routage de la carte. C'est le cas aussi pour les bus s'ils ne sont pas très rapides.

7 Travaux pratiques

7.1 Travail pratique N°1

Ce premier TP a pour but de permettre la prise en main des outils de CAO Xilinx sur un exemple simple : un compteur 8 bits avec reset et sortie sur les leds de la maquette FPGA. Nous allons passer en revue toutes les phases du développement d'un design en mode saisie de schéma puis en VHDL. Ce texte a pour but de vous indiquer l'enchaînement des tâches nécessaires, les explications détaillées concernant la finalité des commandes vous seront données oralement au fur et à mesure du déroulement des travaux pratiques.

7.1.1 Ouverture de session

Le nom de l'utilisateur est fpga, le mot de passe est fpga.

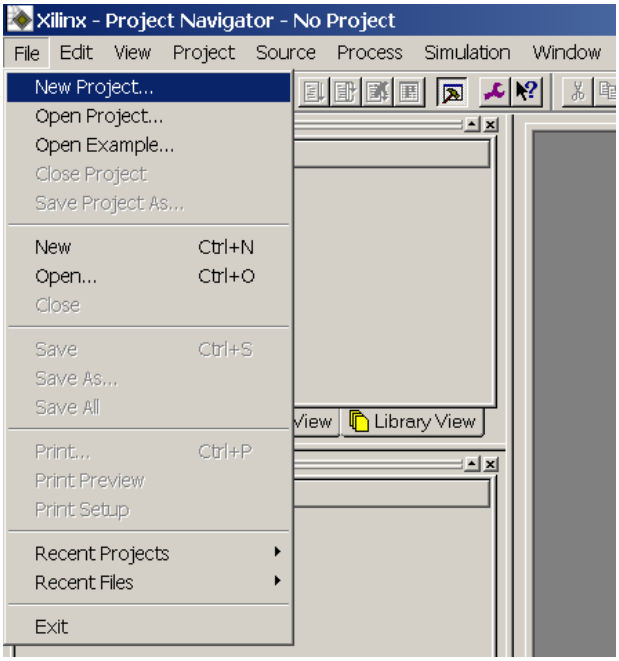
7.1.2 Lancement de « Project Navigator »

Le lancement de l'application principale « Project Navigator » s'effectue en cliquant deux fois sur l'icône 

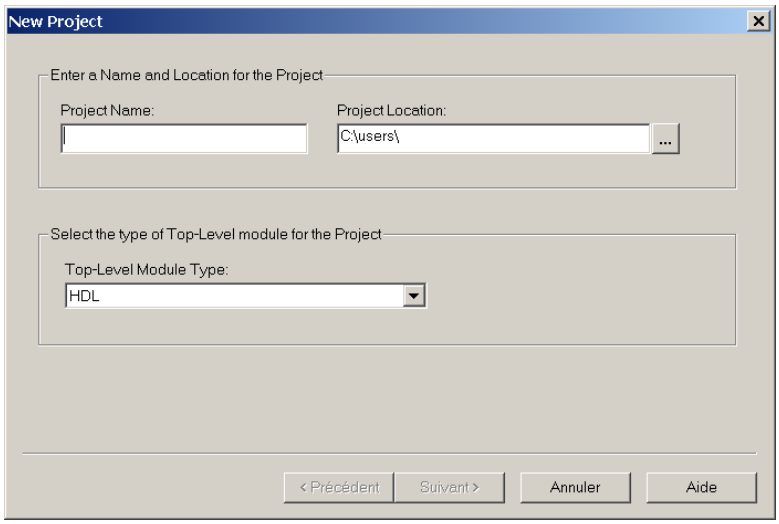
se trouvant sur le bureau.

7.1.3 Création du projet

Après un long moment (30 secondes la première fois), la fenêtre de « Project Navigator » s'ouvre. Cliquez sur le menu « File » et le sous-menu « New Project…».



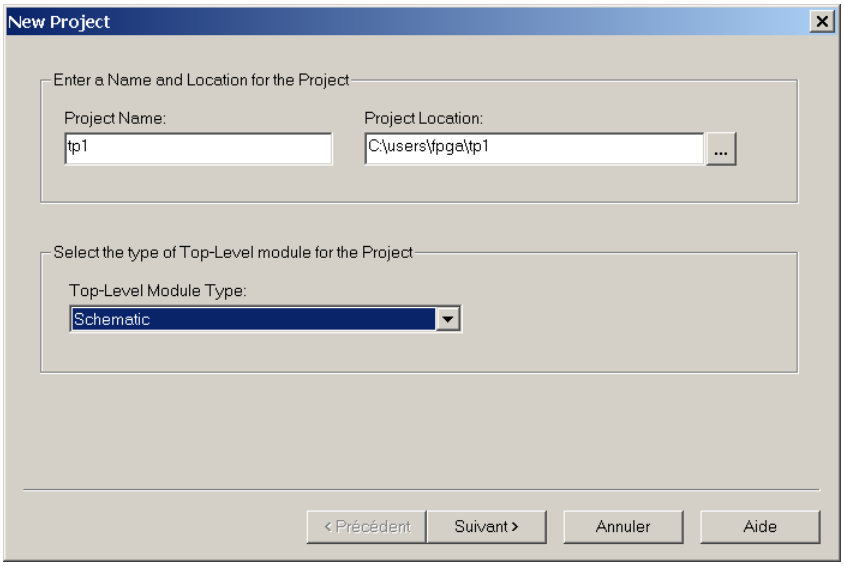
La fenêtre de création du projet apparaît.



Dans l'ordre suivant :

1. Tapez le répertoire du projet c:\users\fpga dans le champ « Project Location »,
2. Tapez le nom du projet tp1 dans le champ « Project Name »,
3. Sélectionnez le type Schematic pour le design principal.

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

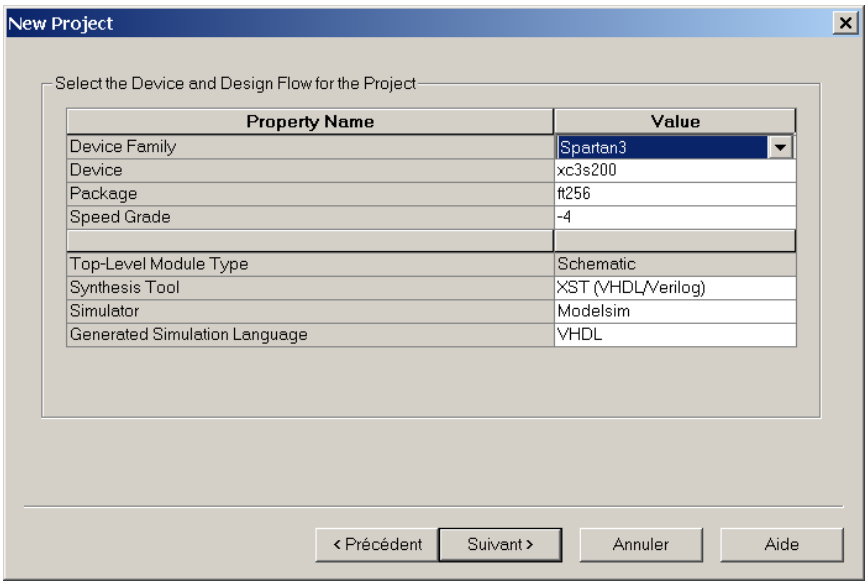


Dans la fenêtre qui s'ouvre, sélectionnez :

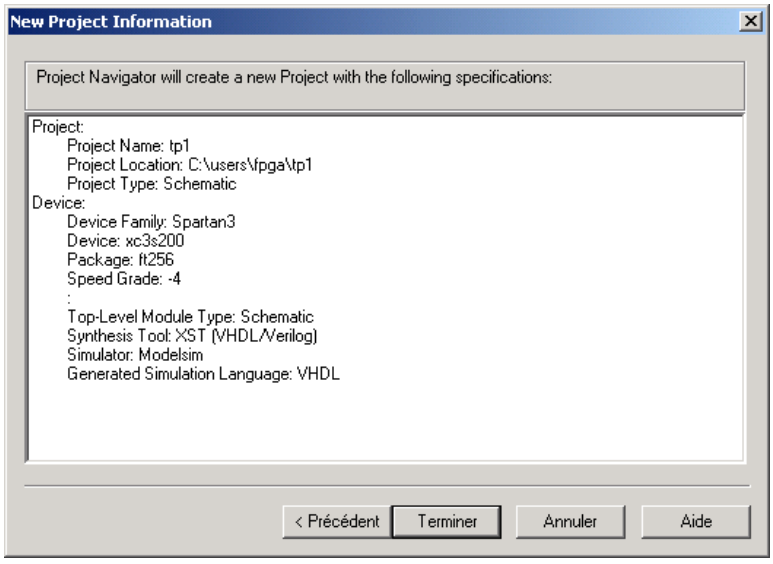
- la famille de FPGA utilisée (Spartan3 dans le champ « Device Family »),
- le circuit utilisé (xc3s200 dans le champ « Device »),

- le boîtier (ft256 dans le champ « Package »),
- la vitesse (-4 dans le champ « Speed Grade »),
- les outils du flot de développement (synthétiseur : XST, simulateur : modelsim, langage VHDL).

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

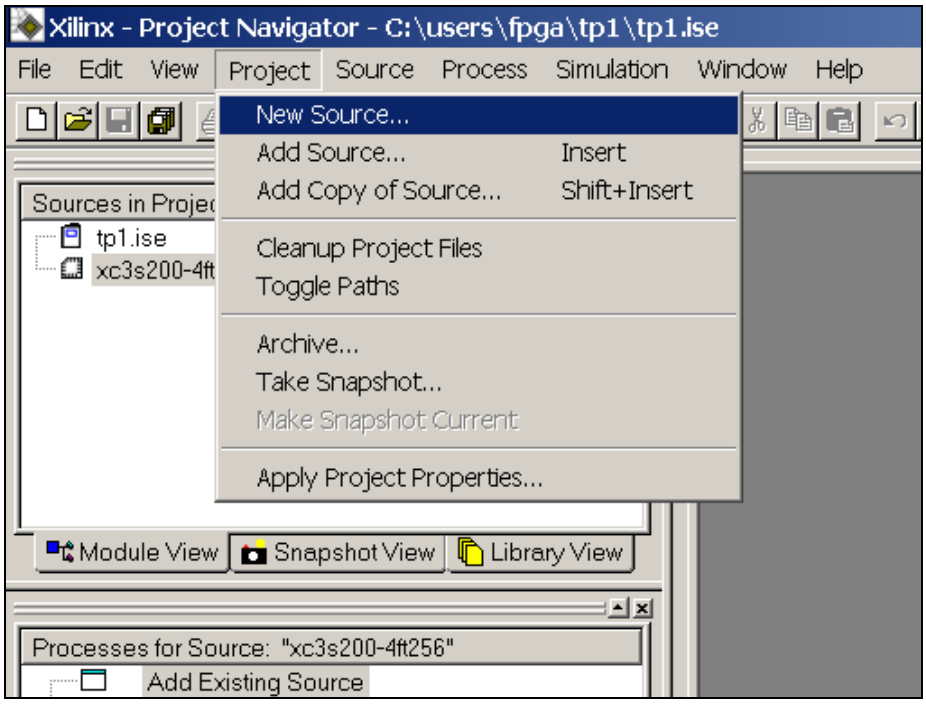


Cliquez sur « Suivant » dans les deux fenêtres suivantes, puis sur « Terminer » dans la fenêtre finale :

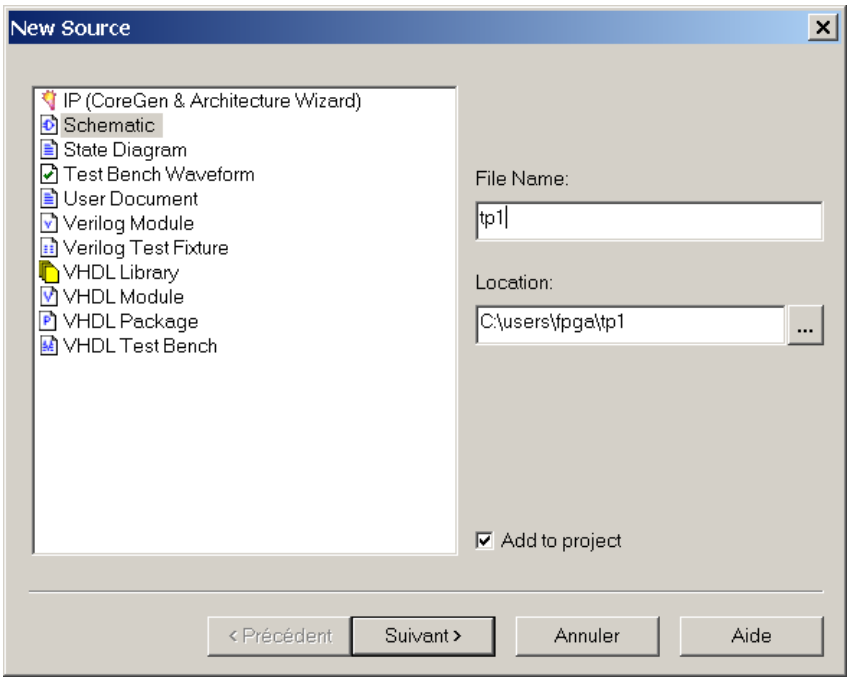


7.1.4 Création du schéma

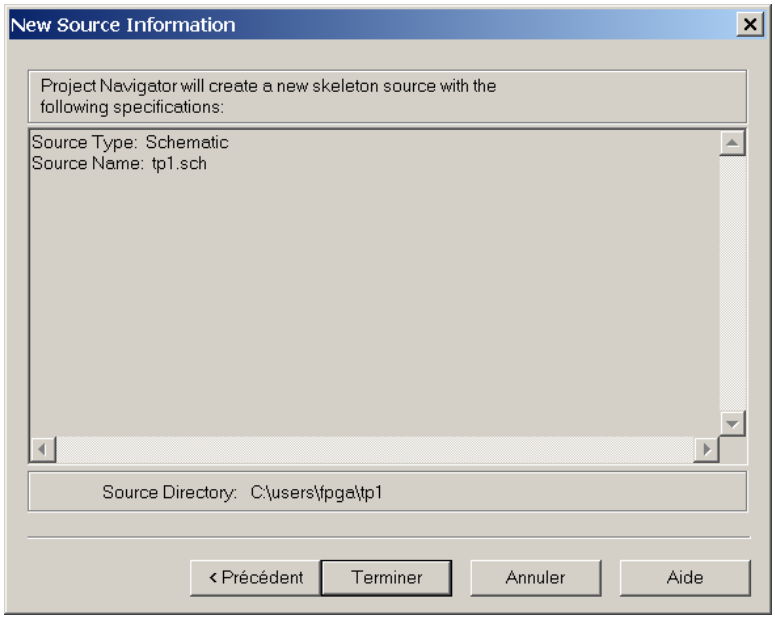
La saisie du schéma s'effectue à l'aide de l'application ECS (Engineering Schematic Capture). Pour la lancer, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



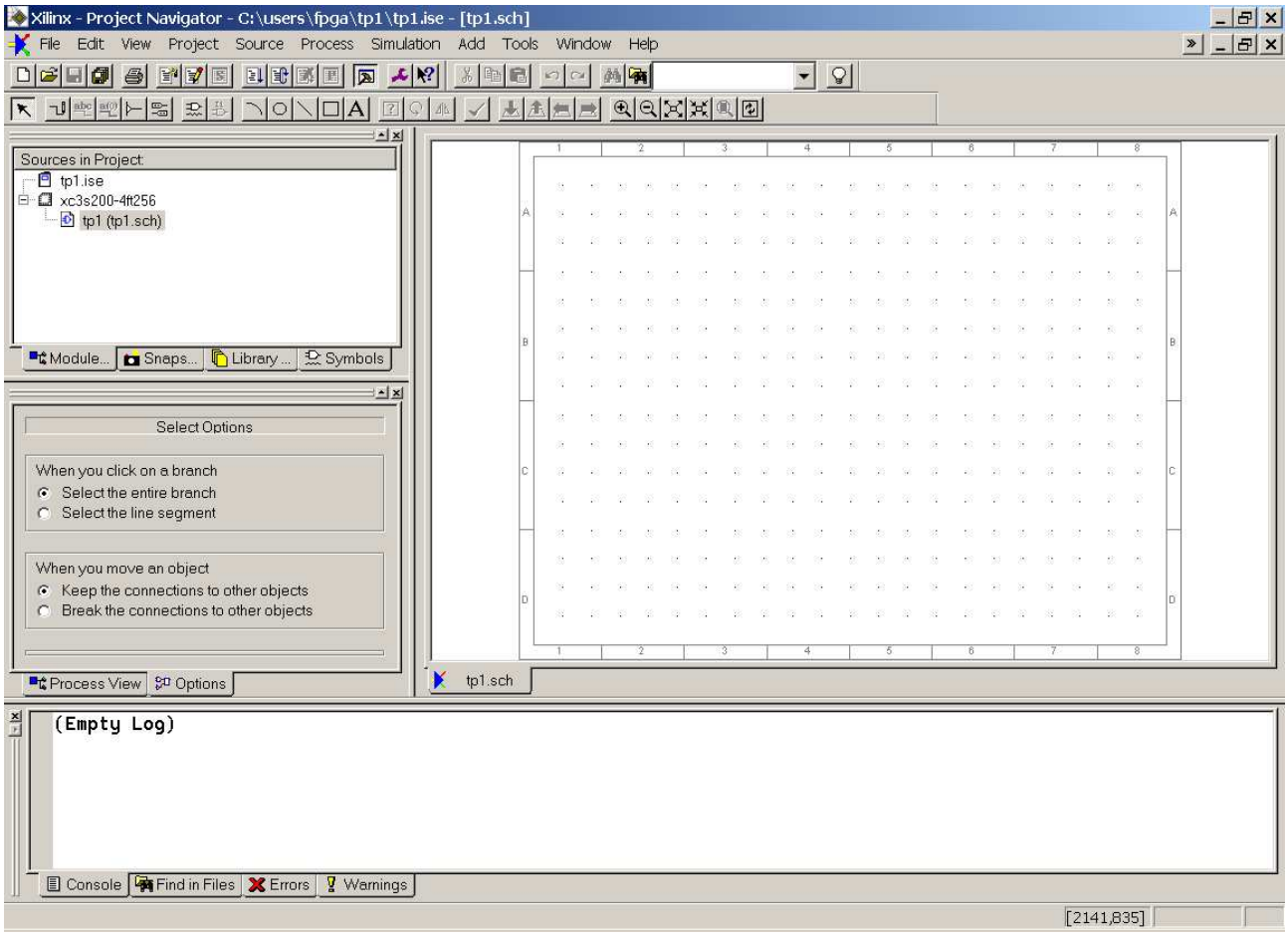
La fenêtre suivante s'ouvre. Sélectionnez « Schematic », tapez le nom du schéma tp1 dans le champ « File Name » puis cliquez sur « Suivant ».



Dans la fenêtre d'information qui s'ouvre alors, cliquez sur « Terminer ».

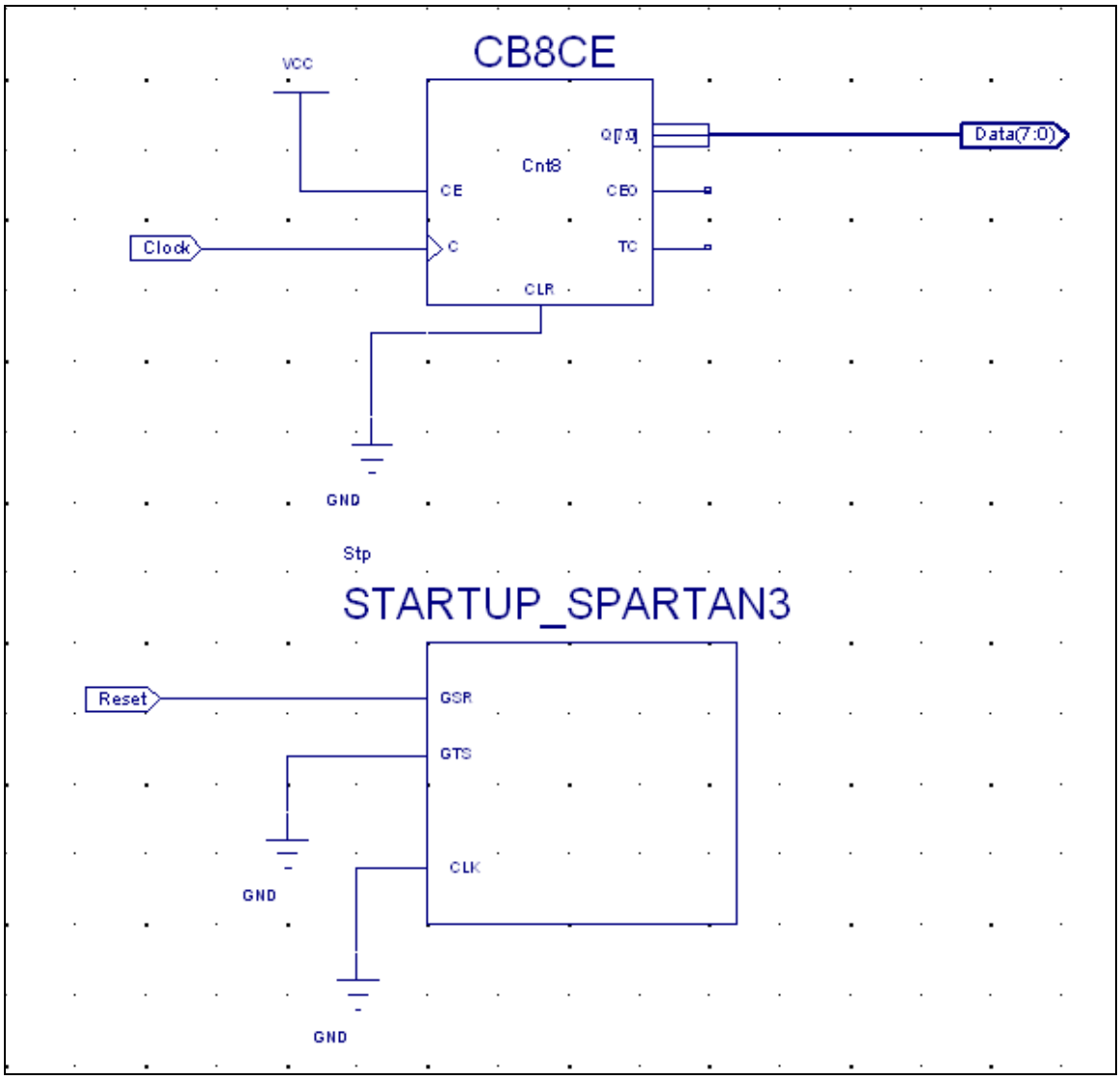


La fenêtre de saisie de schéma apparaît dans Project Navigator.



Pour travailler plus à votre aise, vous allez détacher la fenêtre d'ECS du navigateur de projet en cliquant sur le bouton >> en haut à droite de la fenêtre :

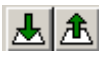
ECS est maintenant séparé du navigateur. Si vous souhaitez le rattacher à nouveau, cliquez sur  . La fenêtre est séparée en 4 zones : les menus déroulants, les 2 niveaux de barre d'outils, la zone de saisie de schéma et la fenêtre options/symbols. Dans ce premier TP, nous allons réaliser une fonction simple pour apprendre à maîtriser les différents outils logiciels ; un compteur 8 bits. Le schéma suivant doit être obtenu avant de passer à la simulation.



Il comprend les symboles :

Categories	Symbol	Label
general	STARTUP_SPARTAN3 (interface utilisateur vers les ressources globales GSR, GTS et horloge de configuration) VCC (force au niveau 1) GND (force au niveau 0)	Stp
counter	CB8CE (compteur binaire 8 bits avec chip enable et clear actifs à 1)	Cnt8

Nous allons maintenant apprendre à nous servir de la saisie de schéma. Passons en revue les commandes de la barre d'outils qui vont nous être utiles dans ce TP :

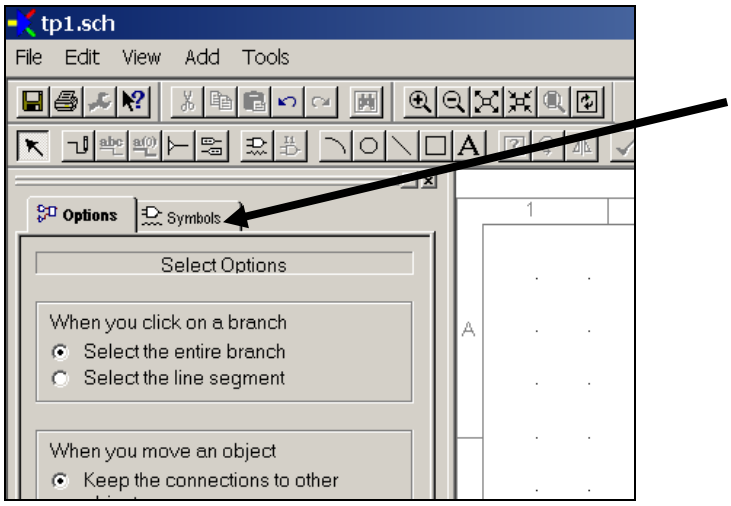
Icône	Nom	Fonction
	Save	Sauve le schéma en cours
	Check Schematic	Vérifie la conformité du schéma
	Zoom Full View (F6)	Affiche plein cadre le schéma actif
	Zoom In (F8)	Augmente l'agrandissement du schéma
	Zoom Out (F7)	Diminue l'agrandissement du schéma
	Zoom To Box	Agrandissement avec sélection de zone
	Zoom to Selected	Agrandissement de la partie sélectionnée
	Push Into Symbol or Return to Calling Schematic	Visualise l'intérieur d'un symbole (descend d'un niveau hiérarchique) ou remonte d'un niveau hiérarchique
	Add Wire	Place un fil entre deux symboles
	Undo	Annule la dernière action
	Redo	Exécute la dernière action annulée
	Add I/O Marker	Ajoute un connecteur d'entrée-sortie

Il existe trois possibilités de zoom.

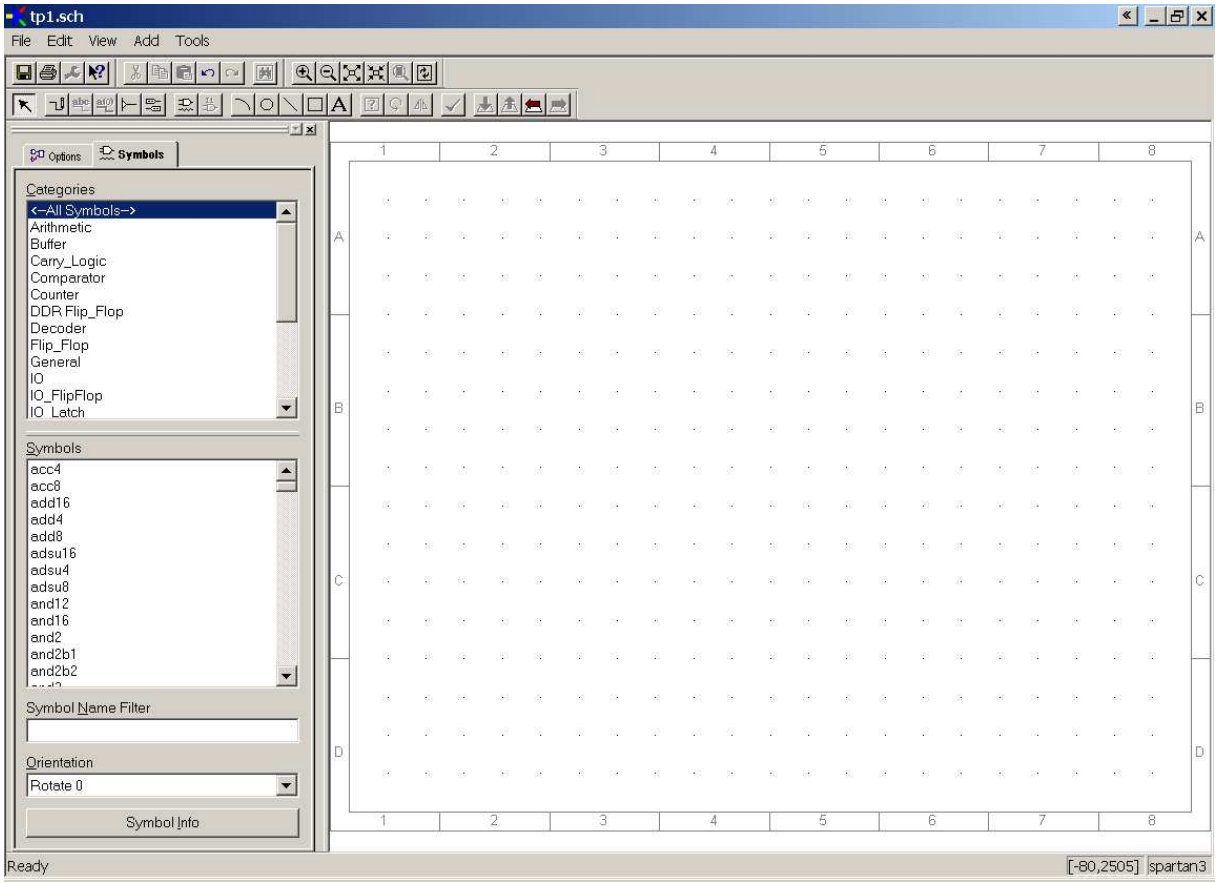
- Avec F7 et F8, vous agrandissez ou réduisez le schéma autour du pointeur de la souris. Avec F6, vous visualisez immédiatement l'ensemble du schéma sur l'écran.
- Quand l'icône « Zoom To Box » est activée, sélectionnez la zone (de gauche à droite) que vous voulez agrandir (placer le curseur de la souris en haut à gauche de la zone à agrandir, cliquer sur le bouton gauche et déplacer le curseur vers le bas à droite tout en maintenant le bouton gauche appuyé, relâcher le bouton de la souris), le zoom s'effectue automatiquement sur la zone sélectionnée. Si vous refaites cette opération du bas à droite vers le haut à gauche (donc en sens inverse), vous annulez le zoom précédent.
- Sélectionnez la zone du schéma que vous voulez agrandir puis cliquez sur le bouton « Zoom to Selected » . L'agrandissement est alors automatique.

Pour déplacer le schéma dans la fenêtre de saisie, utilisez les barres de défilement horizontale et verticale. Le schéma se déplace alors en sens inverse du mouvement de la souris. Pour désactiver (si nécessaire) un mode de placement (symbole ou wire) ou de zoom, cliquez une fois avec le bouton de droite de la souris puis une fois avec le bouton de gauche ou bien appuyez sur la touche « Echap » du clavier.

Les composants à utiliser pour réaliser la fonction souhaitée se trouvent dans des librairies fournies par Xilinx. Pour faire apparaître la liste des symboles disponibles, cliquez sur l'onglet « symbols » dans la fenêtre de gauche d'ECS :



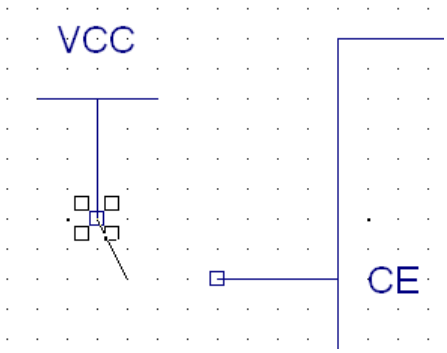
La liste des symboles disponibles apparaît :



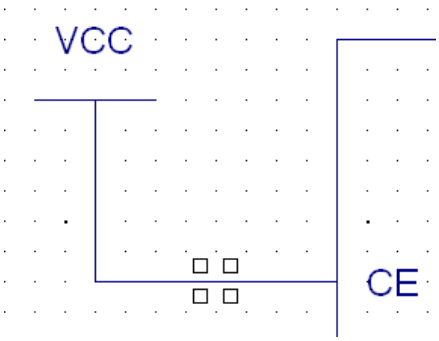
Pour placer un symbole sur le schéma, sélectionnez une catégorie, puis un symbole dans cette catégorie. Lorsque vous déplacez le pointeur de la souris dans la zone de saisie de schéma, le symbole graphique apparaît et bouge avec le mouvement de la souris. Cliquez autant de fois que vous le souhaitez pour placer le symbole aux endroits désirés sur le schéma. Pour arrêter le placement, cliquez une fois avec le bouton de droite de la souris puis une fois avec le bouton de gauche ou bien appuyez sur la touche « Echap » du clavier.

Une fois tous les symboles du schéma placés, relions les entre eux par le biais d'un fil (« wire » en anglais). **Agrandissez suffisamment sur le schéma pour bien voir ce que vous faites.**

Pour relier deux symboles avec un fil, cliquez sur le bouton  dans la barre d'outils. Amenez le pointeur de la souris sur l'extrémité du symbole de départ (4 carrés apparaissent) puis cliquez une fois :

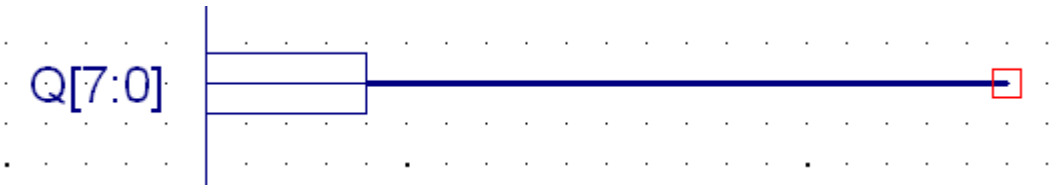


Un fil apparaît, relié à cette extrémité. Amenez l'autre extrémité du fil sur le symbole d'arrivée (4 carrés doivent alors apparaître) puis cliquez à nouveau. La liaison est réalisée.



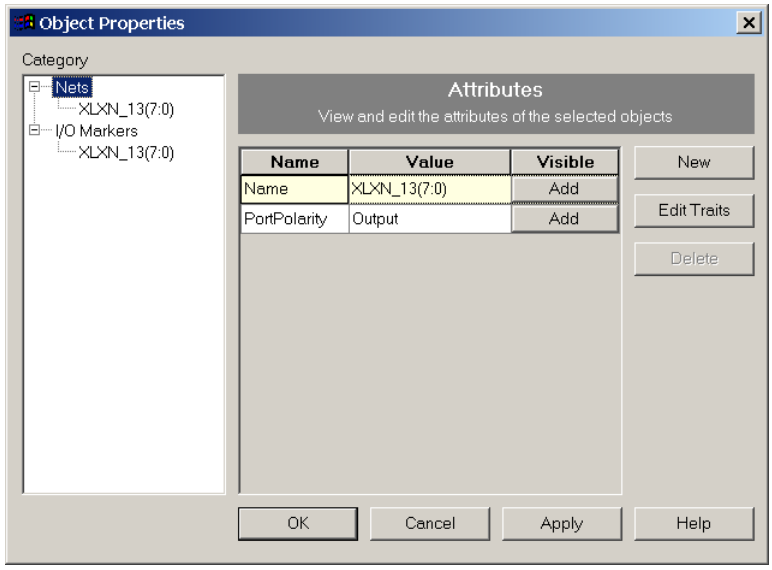
Si le fil doit effectuer un angle droit, cliquez simplement sur l'origine et sur la destination ; ECS se charge d'effectuer le tournant automatiquement. Placez les fils nécessaires sur le schéma.

Occupons-nous maintenant des connecteurs d'entrée-sortie. Vous ne pouvez placer un connecteur sur un fil que si celui-ci existe déjà. Prenons l'exemple de la sortie Data(7:0). Il faut placer le fil à la sortie du compteur et laisser l'autre extrémité non connectée. Pour cela, cliquez une fois sur l'extrémité de l'inverseur, puis double cliquez pour arrêter le placement du fil.

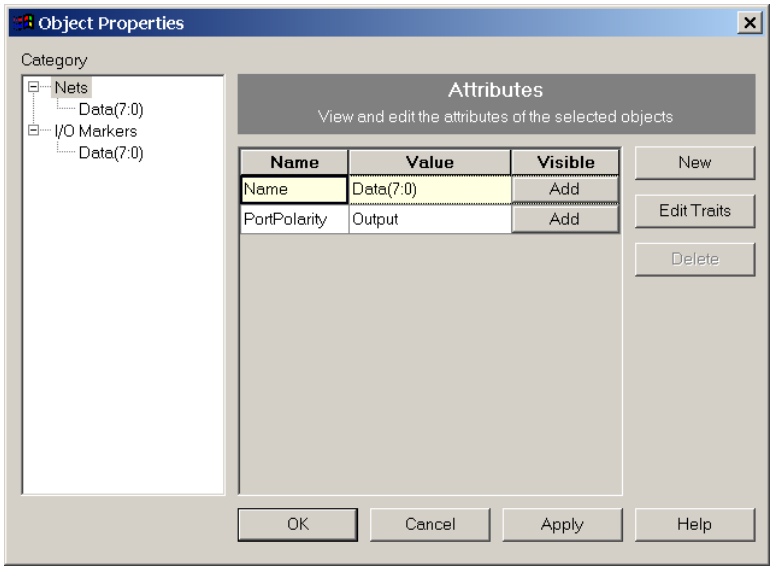


Pour placer le connecteur, cliquez sur l'icône  puis amenez le pointeur sur le carré rouge à l'extrémité du fil. Lorsque les 4 carrés apparaissent, cliquez une fois. Le connecteur est placé.

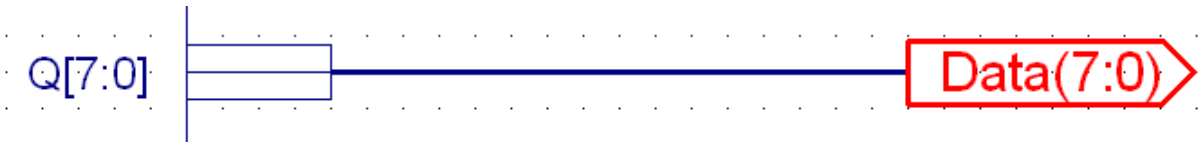
Pour changer le nom des connecteurs, double-cliquez sur le symbole pour faire apparaître la fenêtre des propriétés de l'objet :



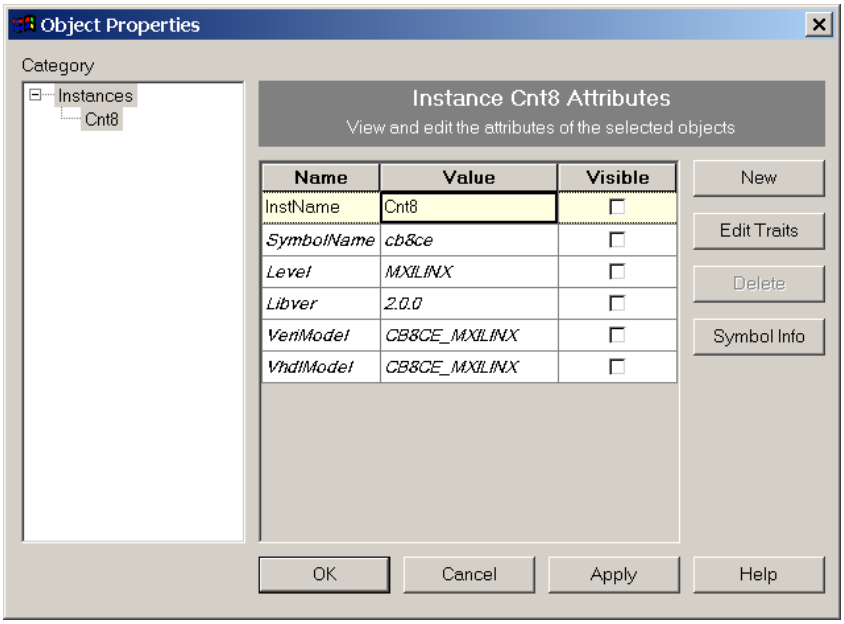
Cliquez sur le champ Name et tapez Data(7:0). Vous devez obtenir la fenêtre suivante avant de cliquer sur OK.



Le schéma est maintenant modifié.

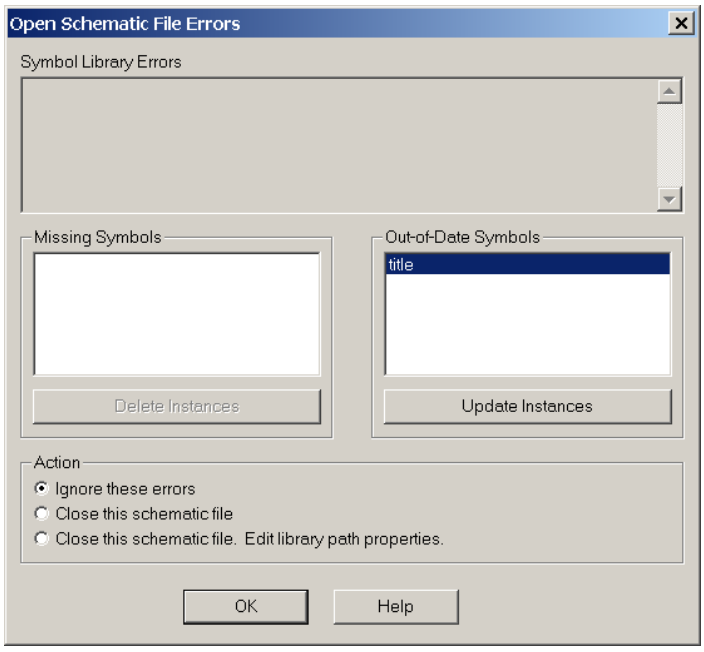


Vous pouvez maintenant placer les deux connecteurs Clock et Reset. Vous devez aussi placer des labels (des noms d'instance) sur les symboles CB8CE et STARTUP_SPARTAN3 (voir tableau des symboles). Pour cela, double cliquez sur le symbole (ou sur le fil) et modifiez son nom. Par exemple, double cliquez sur le compteur et tapez Cnt8 dans le champ InstName :

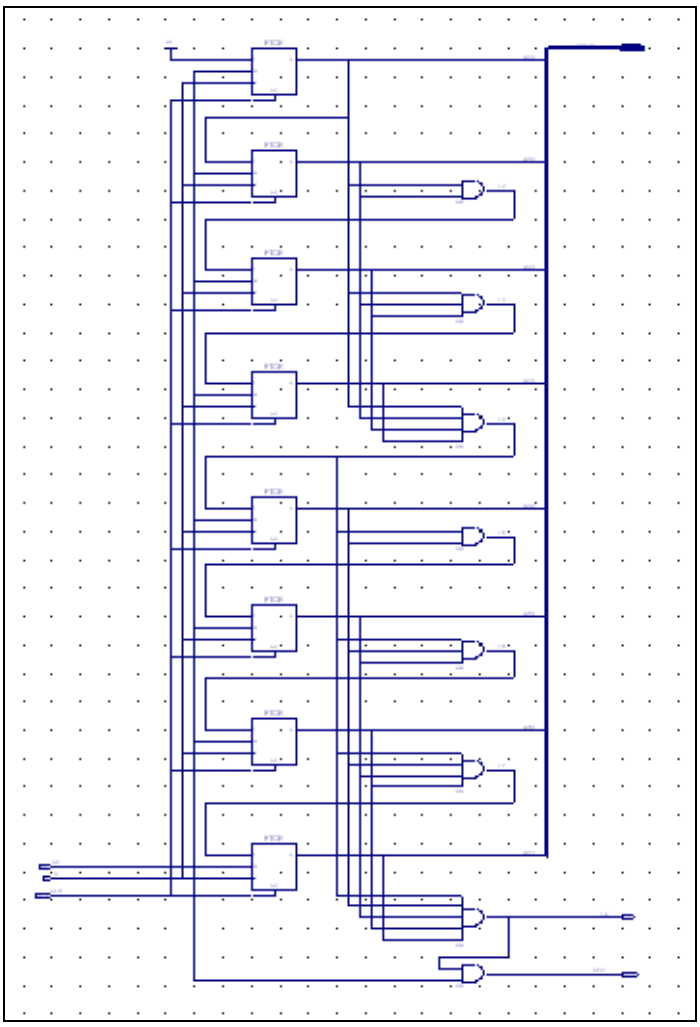


Attention, ne changez pas le nom des fils connectés aux symboles VCC et GND. Laissez la valeur par défaut.

Vous pouvez visualiser l'intérieur d'un symbole en le sélectionnant puis en cliquant sur le bouton  dans la barre d'outils. Si la fenêtre suivante s'ouvre, cliquez sur « OK » pour ignorer l'erreur. C'est un problème de date de fichier.



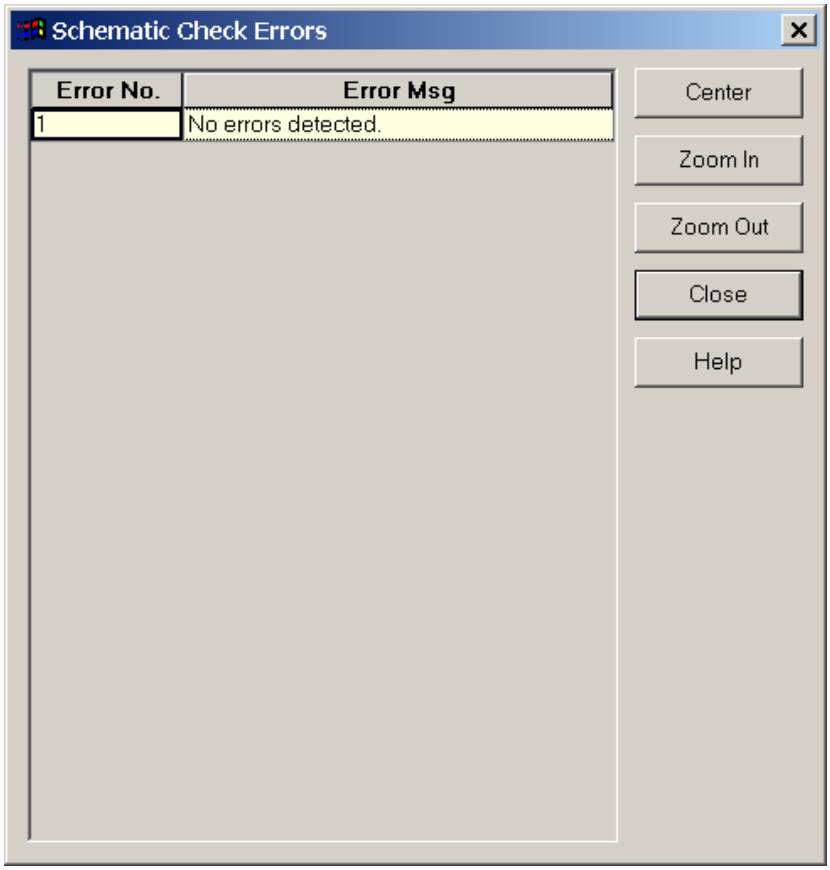
Le schéma correspondant au symbole (Cnt8 dans cet exemple) apparaît alors :



Vous pouvez remonter au schéma racine en cliquant sur le bouton dans la barre d'outils. A ce point de la manipulation, vous avez normalement obtenu le schéma désiré grâce aux trois phases de la saisie de schéma :

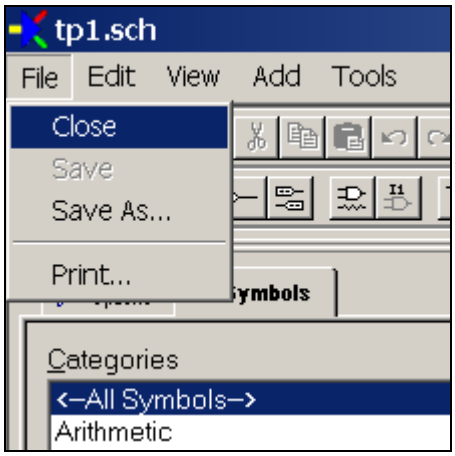
1. Placement des symboles,
2. Liaisons entre les symboles par le biais de fils,
3. Affectations des labels.

Vous devez maintenant vérifier la cohérence de votre schéma grâce au bouton de la barre d'outils. Le menu suivant apparaît si vous n'avez pas commis d'erreurs :

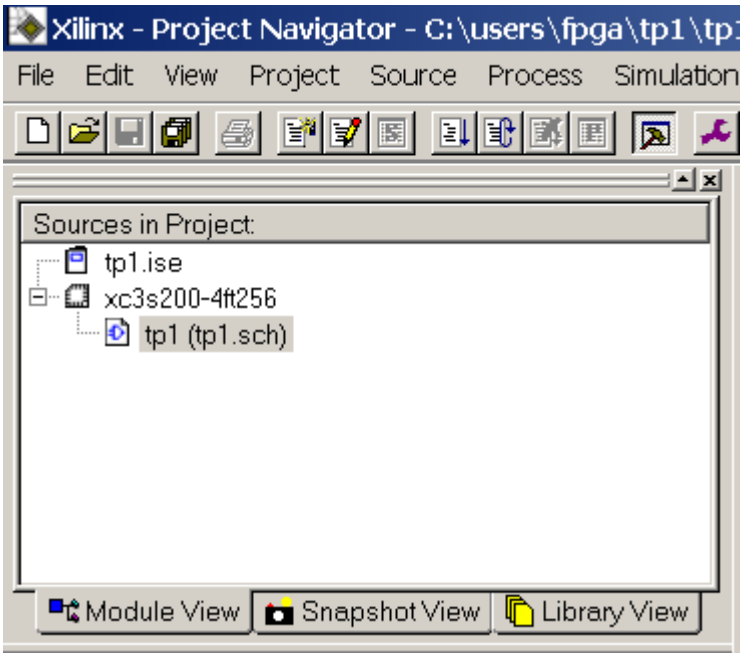


En cas d'erreurs, sélectionnez le message. La zone qui pose problème apparaît sur le schéma surlignée en jaune. Corrigez l'erreur, puis refaites une vérification.

Le schéma est maintenant vérifié et doit être sauvé. Cliquez sur le bouton de la barre d'outils puis fermez la saisie de schéma :

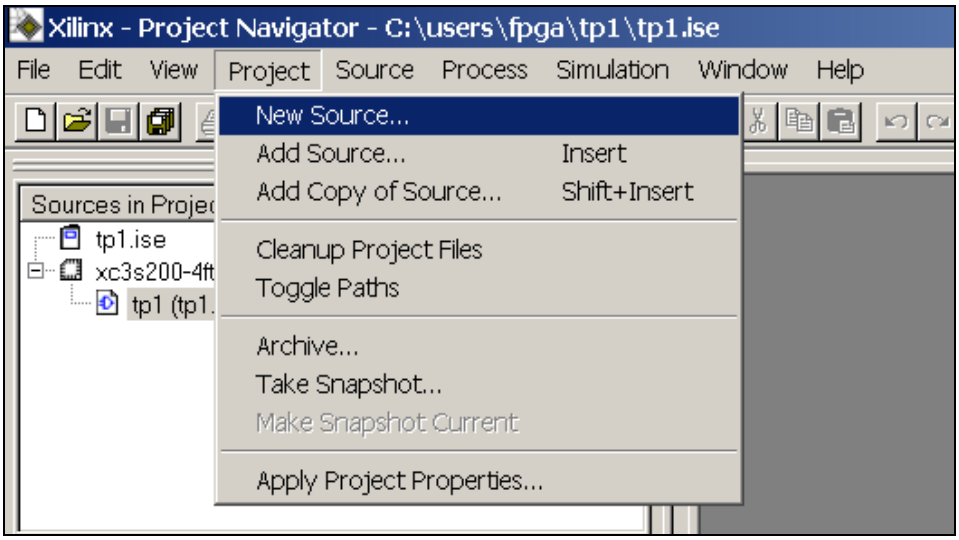


Le schéma apparaît maintenant dans les sources du navigateur de projet :

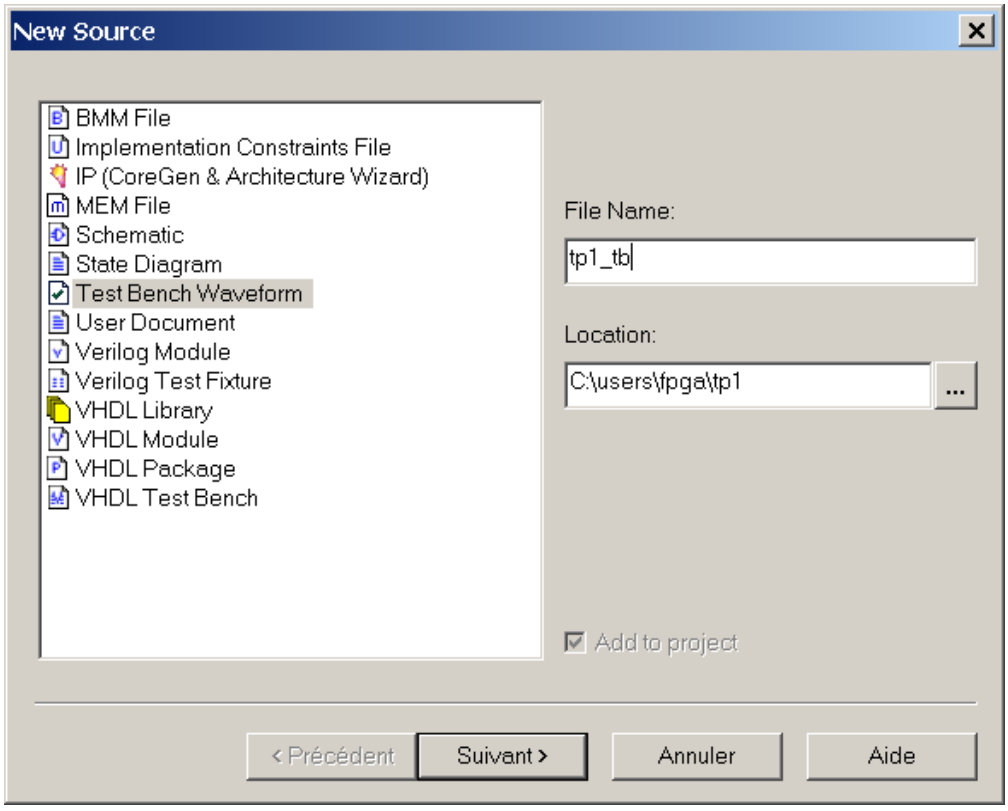


7.1.5 Génération du fichier de stimuli VHDL

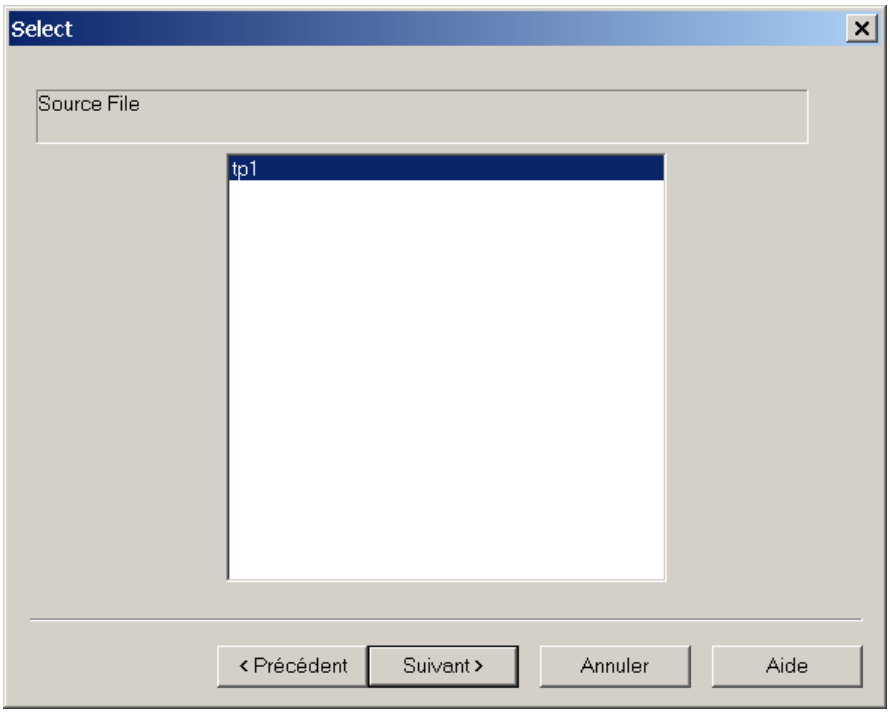
Nous avons maintenant un design prêt à être simuler et nous devons écrire un fichier en langage VHDL décrivant les signaux d'entrées (dans ce TP, il s'agit seulement d'un signal d'horloge et du Reset). Ce fichier s'appelle un fichier de stimuli (un testbench en VHDL) qui contient des vecteurs de test. Il y a parmi les outils Xilinx un outil graphique pour définir les stimuli : Waveform Editor. Pour lancer cette application, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



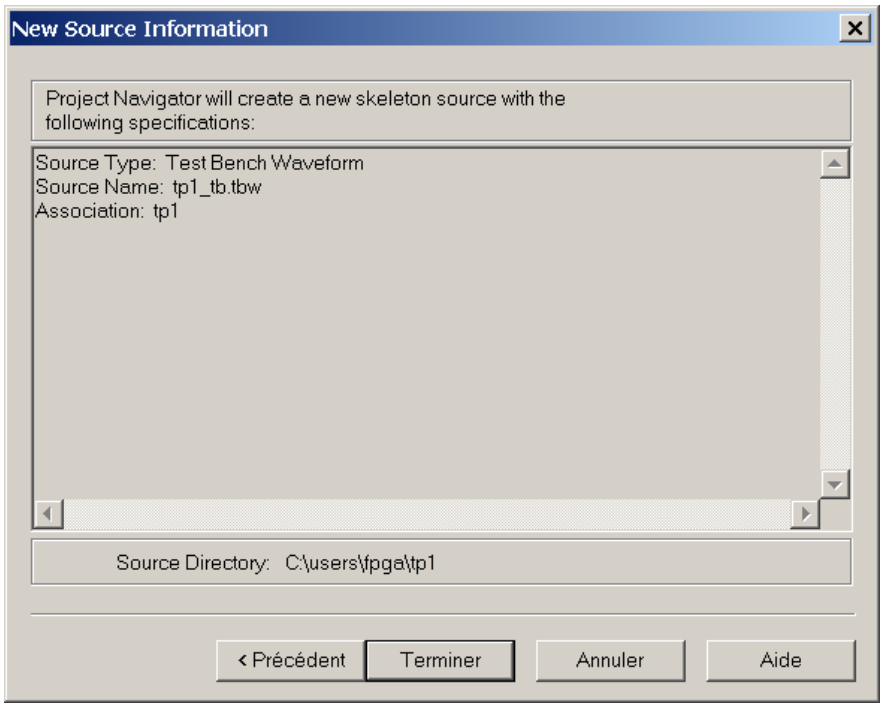
La fenêtre suivante apparaît alors à l'écran. Sélectionner le type « Testbench waveform » et tapez tp1_tb comme nom de fichier pour notre testbench :



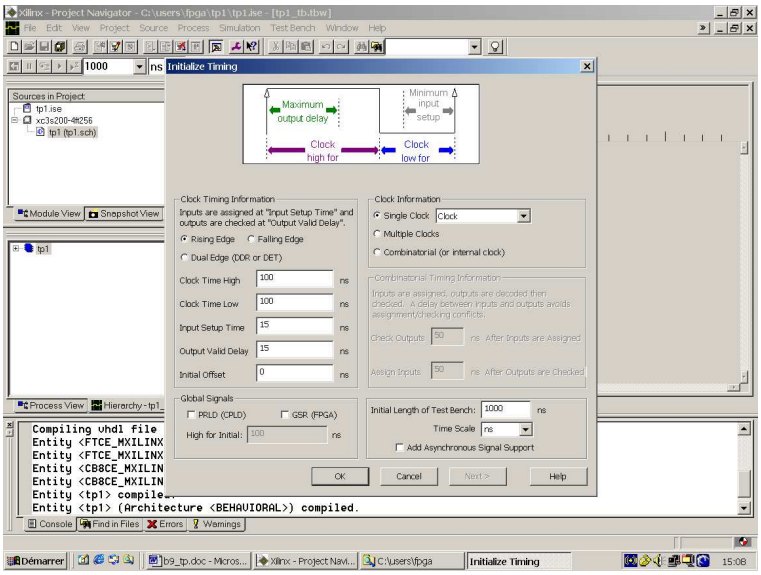
Cliquez sur suivant pour associer le testbench avec le design tp1 :



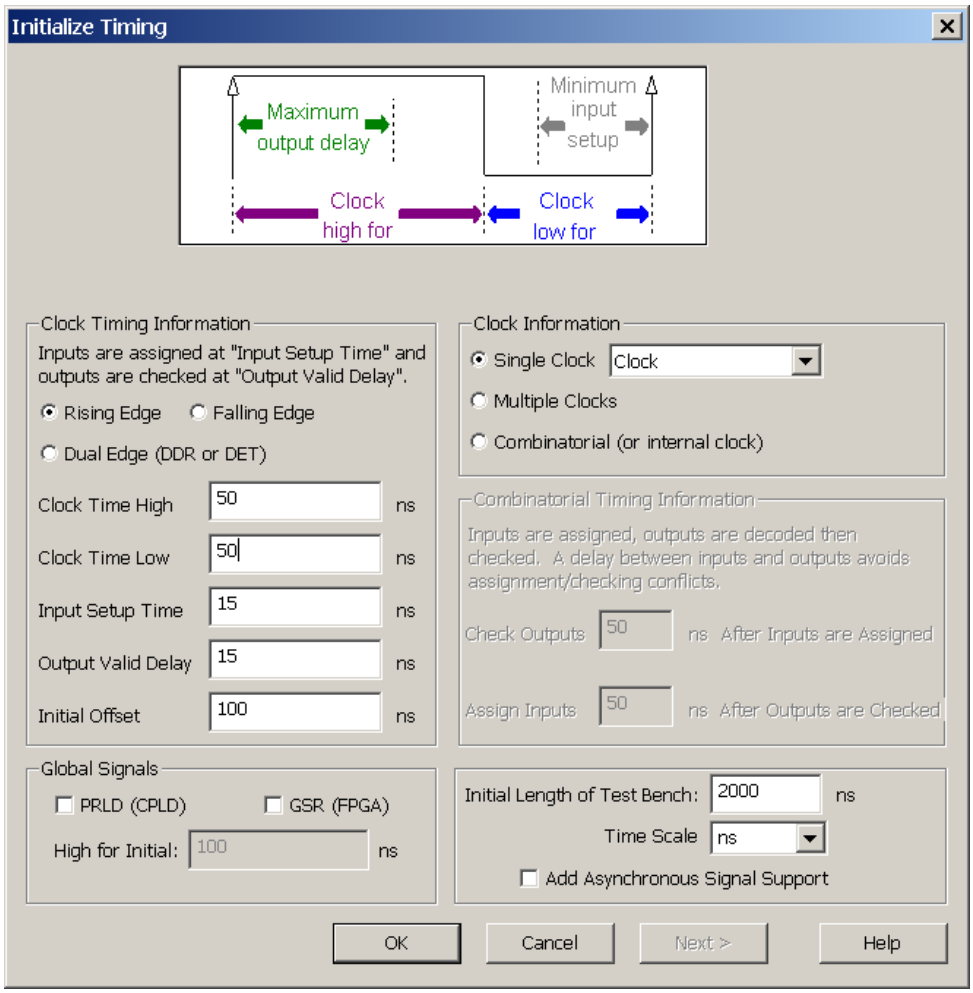
puis sur terminer pour lancer l'éditeur de stimuli :



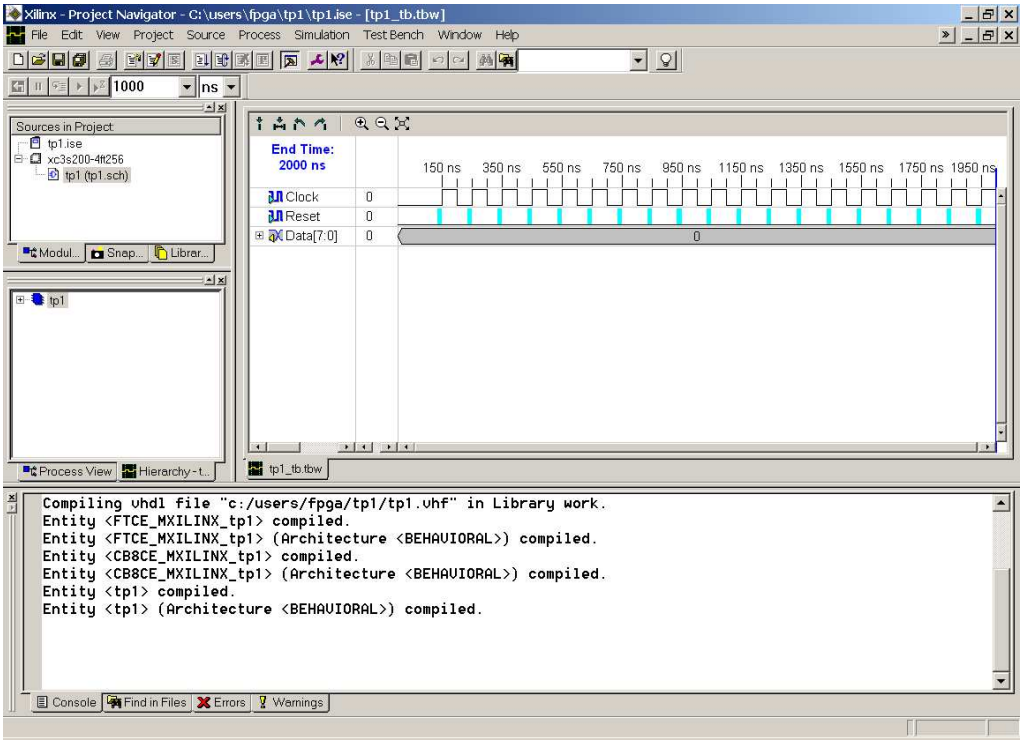
L'application démarre.



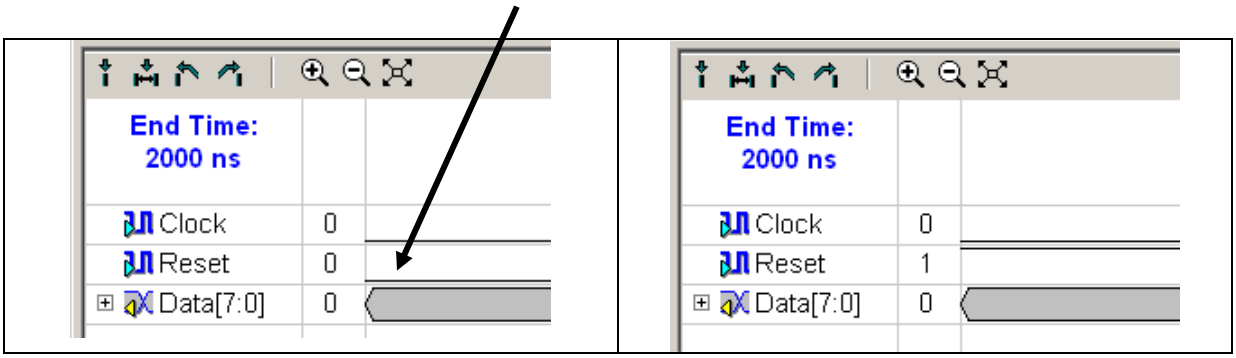
L'horloge du design tp1 est bien Clock. Nous souhaitons une horloge à 10 MHz, avec un temps mort de 100 ns au démarrage (pour faire le reset) et une durée de simulation égale à 2000 ns. Remplissez les différents champs comme sur la fenêtre suivante avant de cliquer sur « OK » :



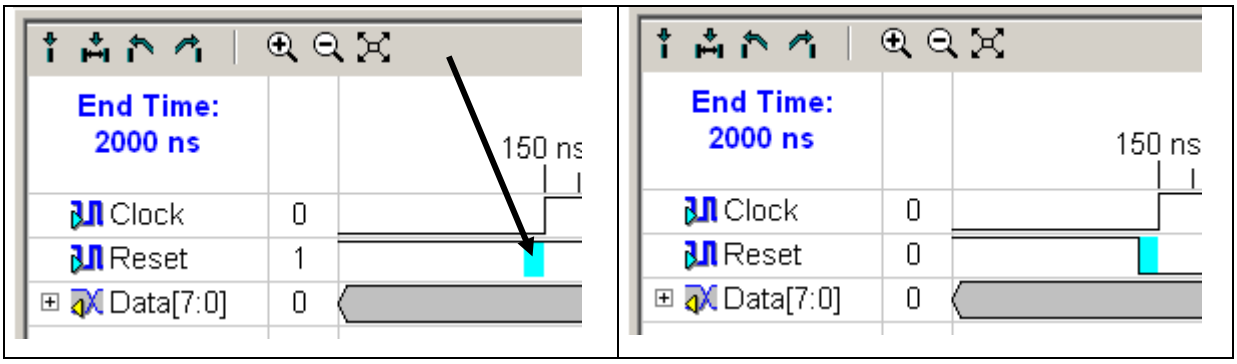
Un chronogramme apparaît dans la fenêtre de droite du navigateur de projet :



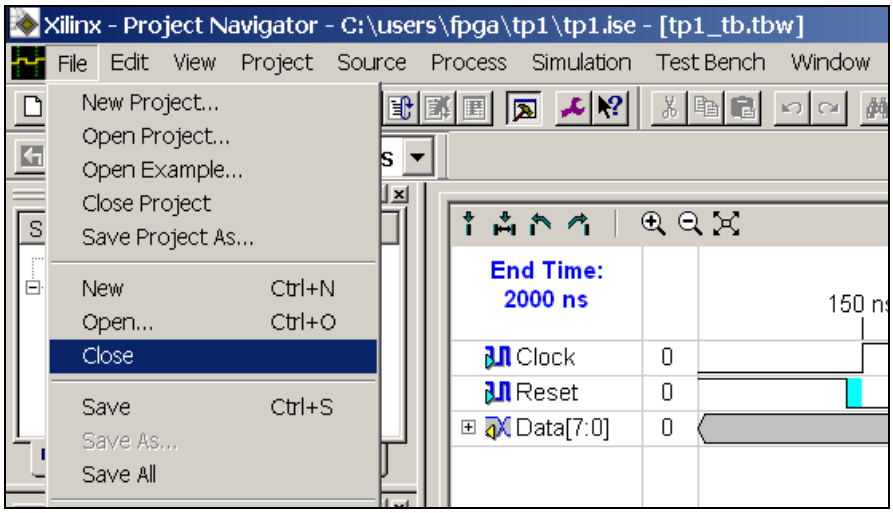
Nous pouvons maintenant spécifier les stimuli sous forme graphique. Vous devez voir deux entrées (Reset et Clock) et une sortie (Data[7:0]). Si tel n'est pas le cas, vous avez sûrement oublié de mettre un connecteur sur le schéma. Fermez la fenêtre sans la sauvegarder puis, dans le navigateur, sélectionnez tp1, cliquez avec le bouton droit de la souris puis sur « Open ». La fenêtre de saisie de schéma s'ouvre à nouveau. Faites les modifications nécessaires, sauvez votre schéma puis relancez Waveform Editor. Renouvelez ces opérations jusqu'à ce que vous ayez les bonnes entrées-sorties. Une fois la bonne fenêtre obtenue, cliquez sur le chronogramme du Reset pour le faire passer à 1 :



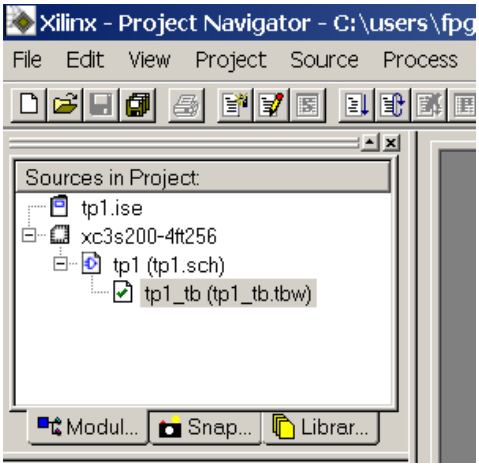
Cliquez ensuite sur la première zone bleue pour faire passer le Reset à 0 :



Cliquez sur l'icône  pour sauver le testbench, puis sur le menu « File », « Close » pour quitter l'application :



Le testbench apparaît maintenant dans les sources du navigateur de projet. Vous pouvez à tout moment relancer Waveform Editor en double cliquant sur tp1_tb.tbw.

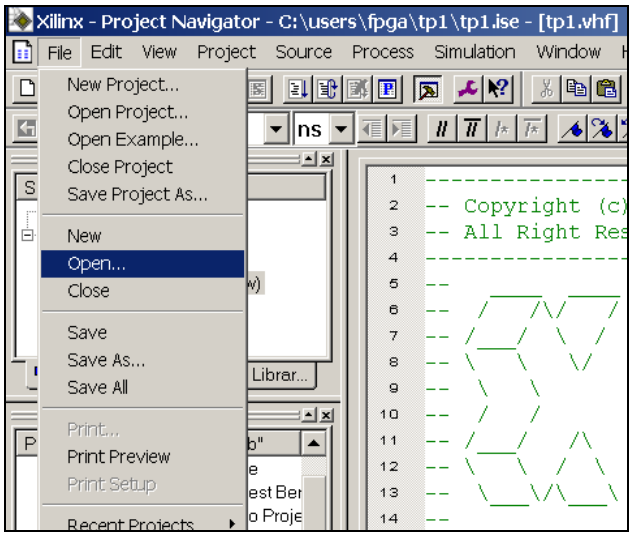


7.1.6 Simulation fonctionnelle

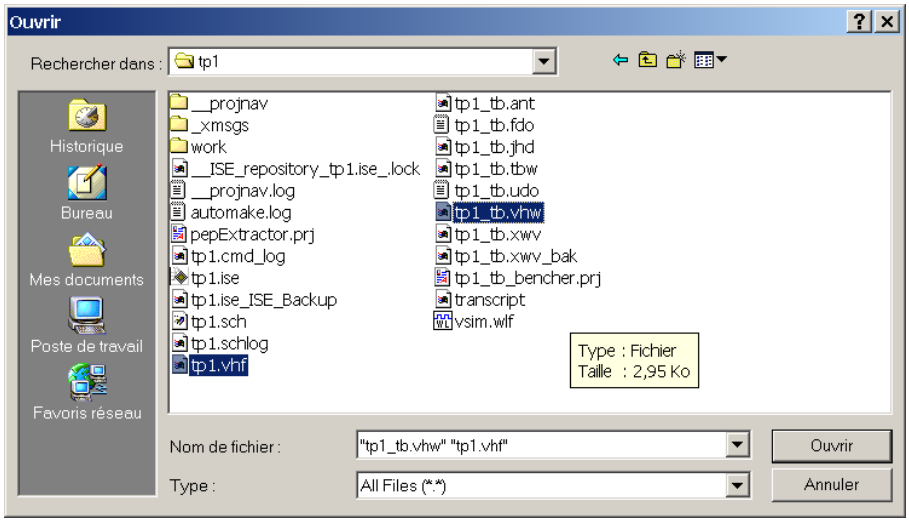
Nous sommes maintenant en possession de deux fichiers :

1. un fichier VHDL représentant le design à réaliser (tp1.vhf). Ce fichier est la traduction du schéma sous forme d'une netlist VHDL.
2. un fichier VHDL représentant les signaux que l'on souhaite appliquer sur ses entrées (tp1_tb.vhw). Ce fichier, généré par Waveform Editor, est la représentation en VHDL des stimuli créés précédemment sous forme graphique.

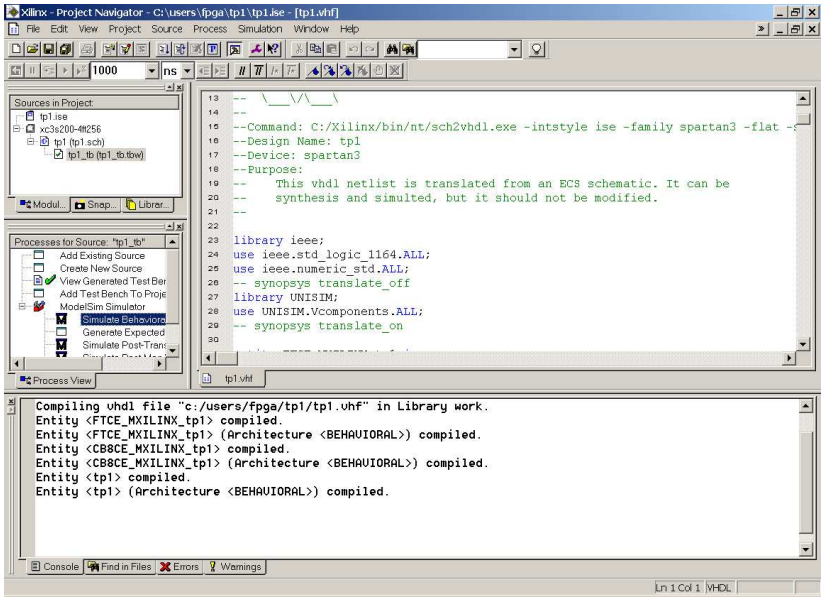
Vous pouvez visualiser ces deux fichiers à l'aide du navigateur de projet en cliquant sur le menu File puis sur le sous-menu Open :



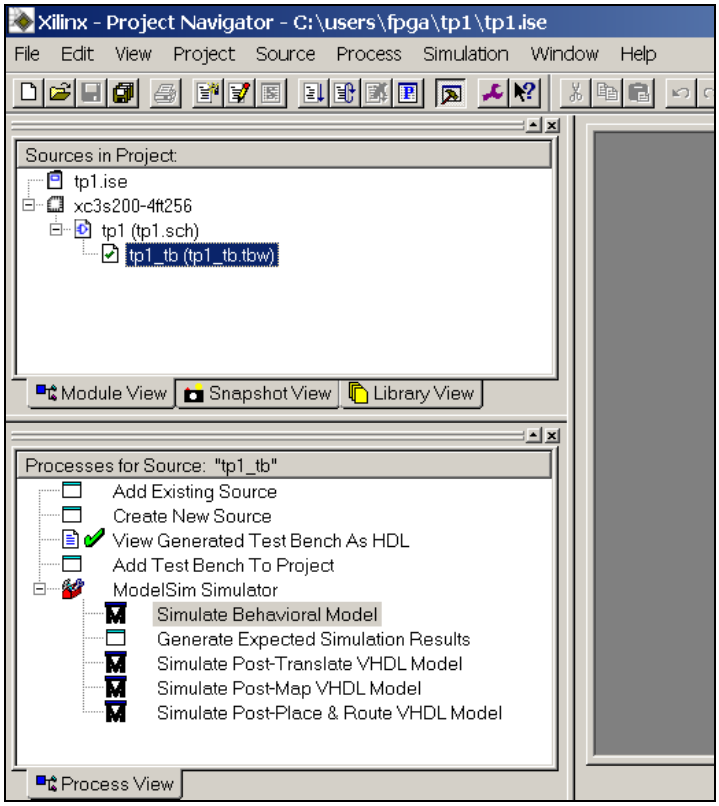
Sélectionnez le type de fichier « All Files » puis le fichier. Cliquez alors sur Ouvrir.



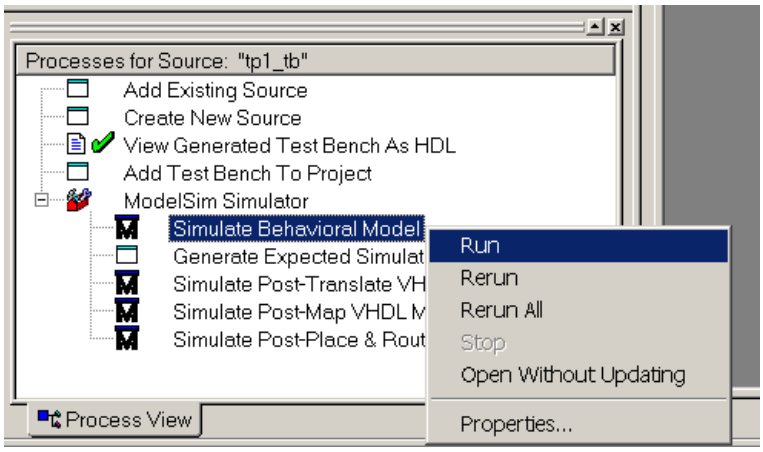
Le contenu du fichier apparaît dans la fenêtre d'édition du navigateur :



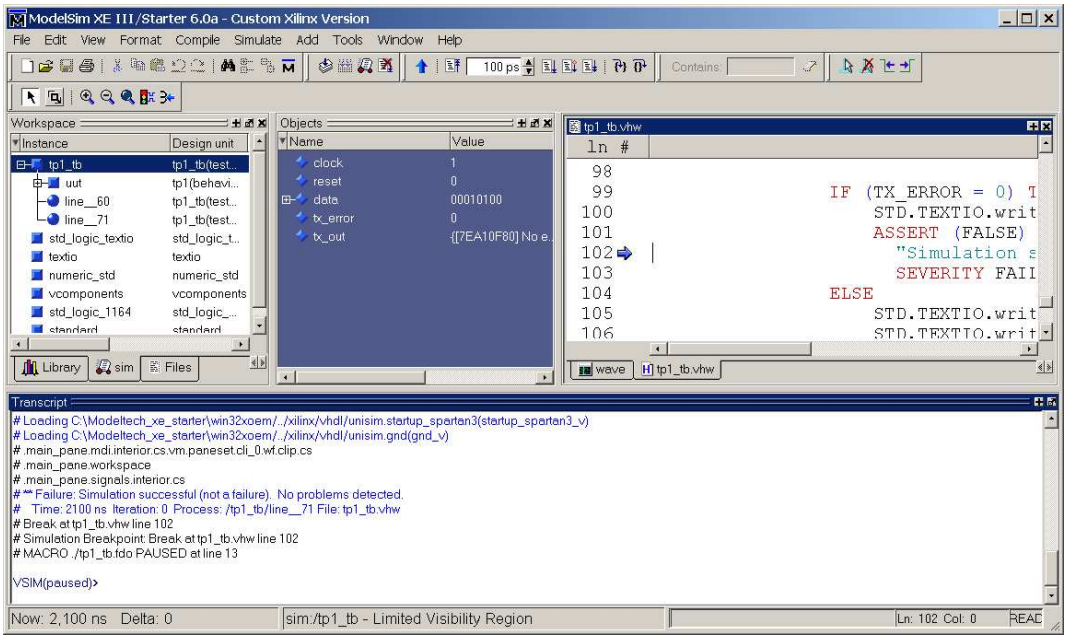
Fermez la fenêtre d'édition. Nous allons maintenant simuler le design à l'aide du simulateur VHDL ModelSim. Pour cela, sélectionnez le fichier tp1_tb.tbw dans la fenêtre « Sources In Project ». Les actions qui peuvent être effectuées avec ce fichier apparaissent dans la fenêtre inférieure « Processes for Source ».



Sélectionnez le process « Simulate Behavorial Model » puis cliquez avec le bouton droit de la souris sur le menu Run :



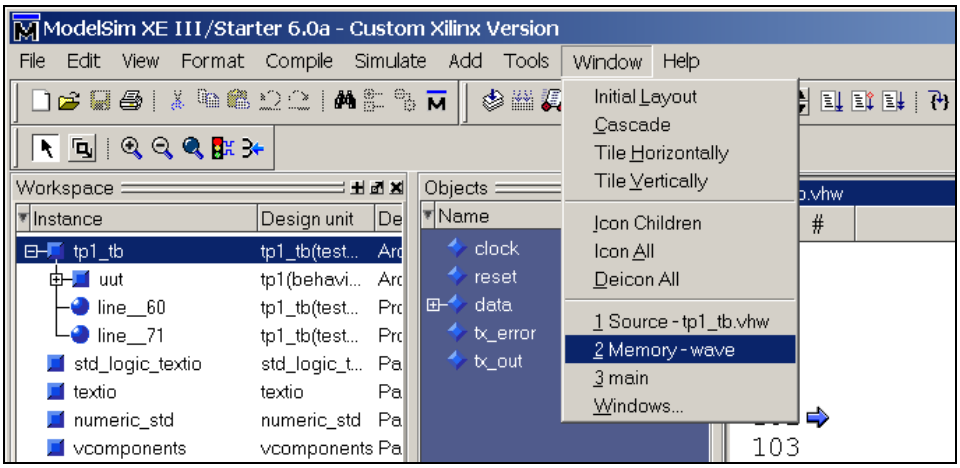
Le simulateur VHDL ModelSim démarre et la fenêtre suivante s'ouvre à l'écran. Si tel n'est pas le cas, c'est parce que ModelSim s'exécute sous le navigateur de projet. Il faut alors cliquer sur son icône dans la barre de tache de Windows pour le mettre au premier plan.



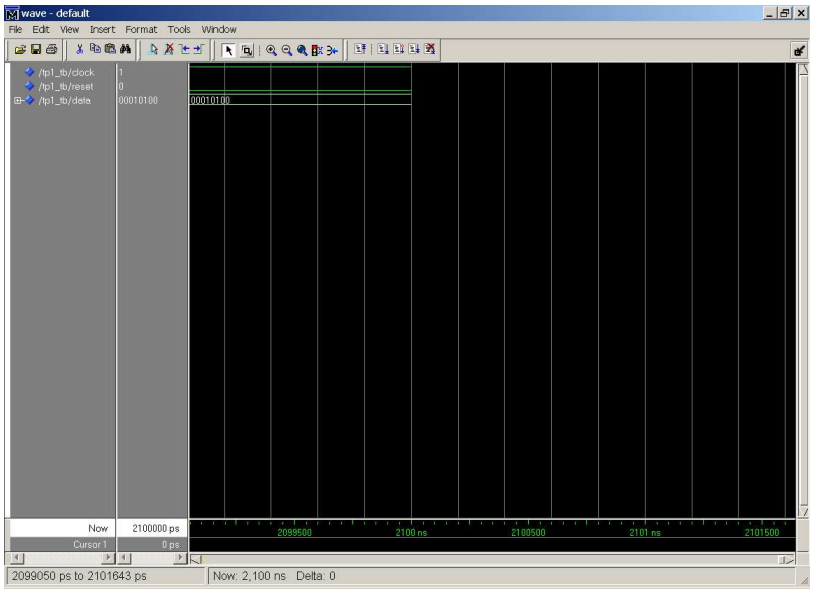
ModelSim va enchaîner automatiquement :

- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) en cliquant sur le menu Windows, Memory-wave :

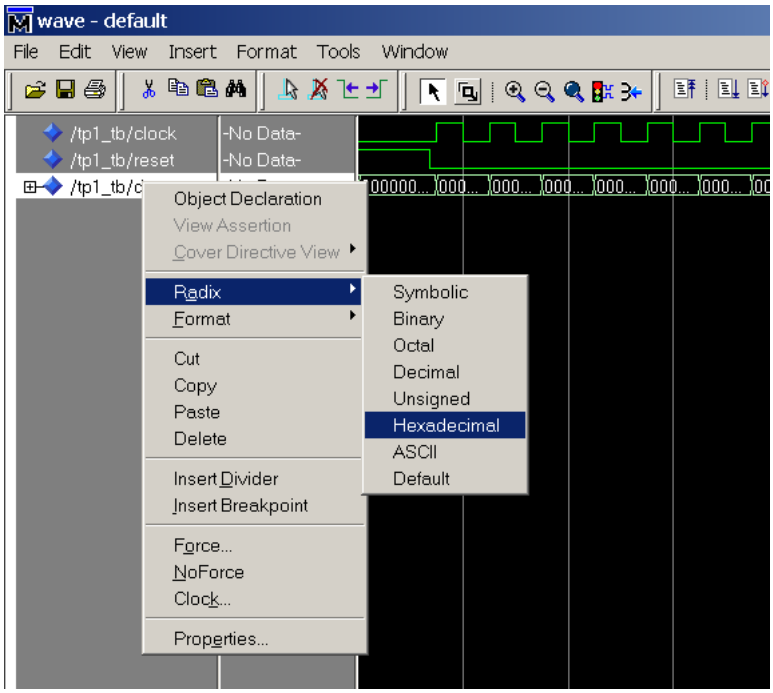


Nous allons détacher la fenêtre Wave de la fenêtre principale de Modelsim afin de mieux voir les chronogrammes. Pour cela, cliquez sur le bouton du milieu  (en haut à droite de la fenêtre wave) puis mettez la fenêtre wave plein écran.

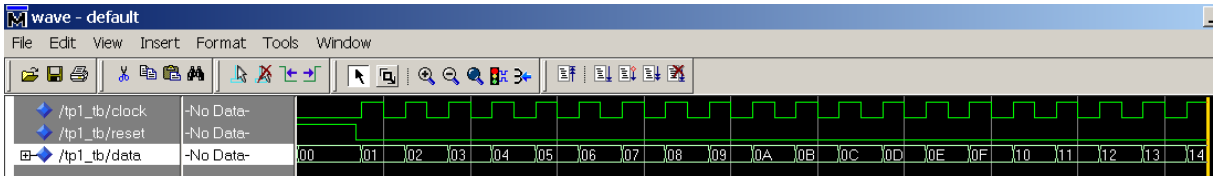


Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation (de 0 à 2100 ns). Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes.

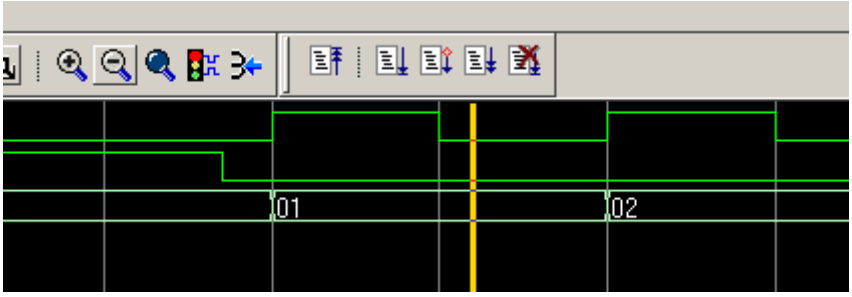
Sélectionnez le bus data puis cliquez avec le bouton droit de la souris sur le menu Radix, puis sur Hexadecimal.



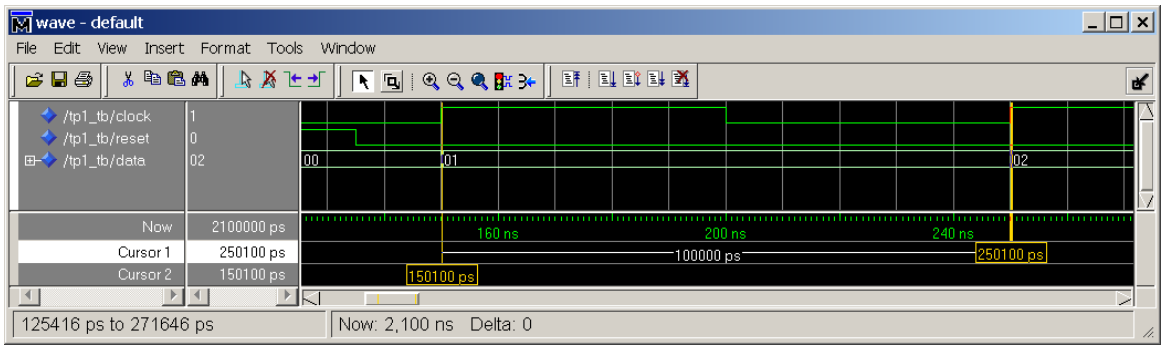
Le bus contient maintenant des valeurs hexadécimales au lieu de valeurs binaires.



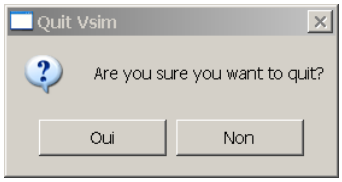
Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Cliquez n'importe où sur le chronogramme. Un curseur jaune apparaît :



Cliquez sur le bouton « Insert Cursor »  de la barre d'outils. Un deuxième curseur apparaît. Vous voyez en bas de la fenêtre « Wave » le temps correspondant à la position de chaque curseur ainsi que l'écart qui les sépare. Sur l'exemple suivant, on mesure la période de l'horloge.

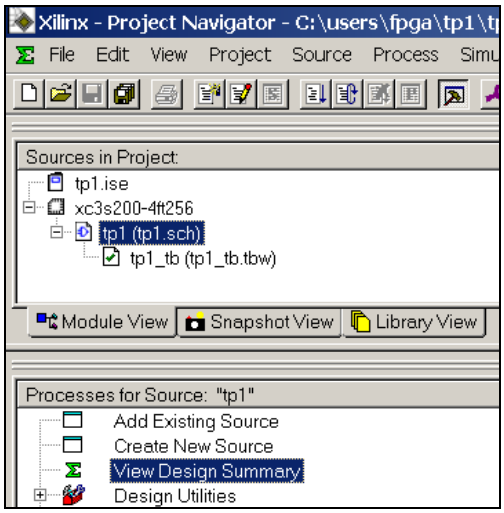


Quand la vérification est terminée, faites apparaître la fenêtre de ModelSim, puis cliquez sur le menu « File » dans la fenêtre principale, puis sur « Quit ». Quand le message suivant apparaît à l'écran, cliquez sur Oui.

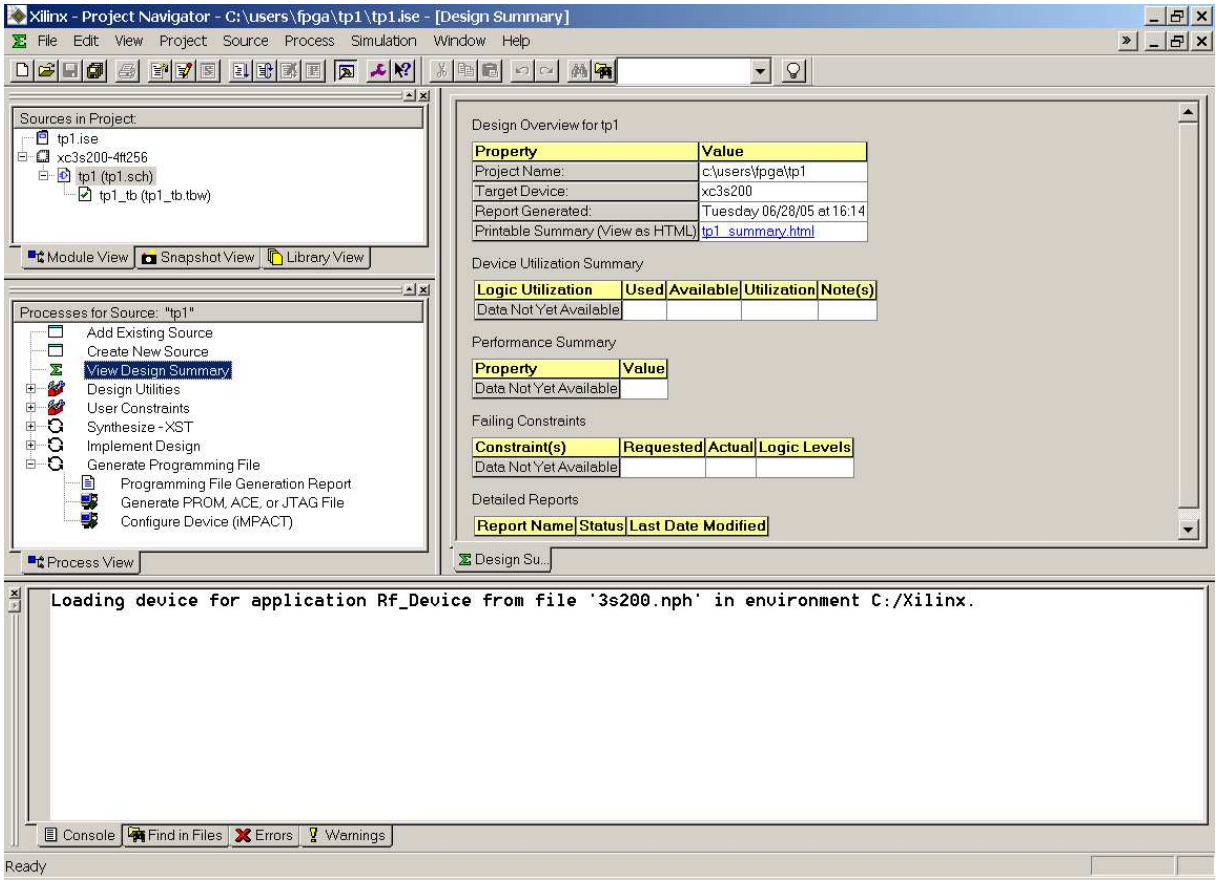


7.1.7 Synthèse

Nous avons fini la première phase de création et de vérification du design. Vous pouvez voir à tout moment le résumé des différentes étapes du design en sélectionnant tp1, puis en double-cliquant sur « View Design Summary » :

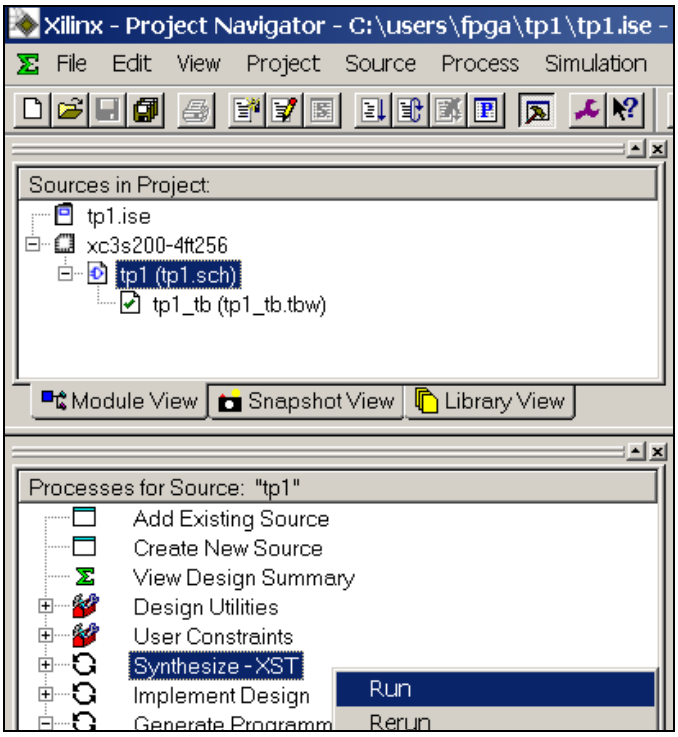


La page de résumé s'affiche à droite du navigateur de projet :



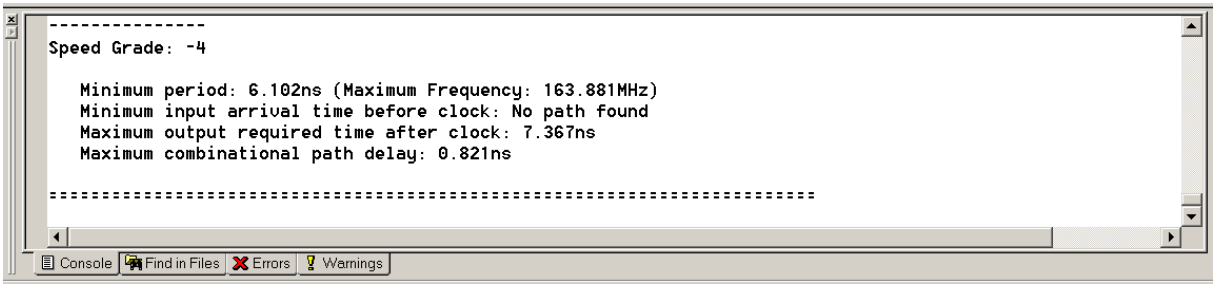
Elle contient pour l'instant peu d'informations, mais elle va s'enrichir au fur et à mesure de l'avancement du projet. Il faut maintenant traduire la netlist VHDL en une netlist NGC. C'est le rôle du synthétiseur XST. En réalité, la synthèse est l'opération qui permet de créer une netlist NGC à partir d'une description de haut niveau écrite en VHDL (ou en Verilog). Comme nous partons ici d'un schéma (donc d'une netlist), la synthèse est beaucoup plus facile puisque le fichier VHDL contient déjà des composants qui peuvent être compris par les outils de placement-routage.

Sélectionnez le design tp1.sch dans la fenêtre « Sources » puis « Synthesize » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :

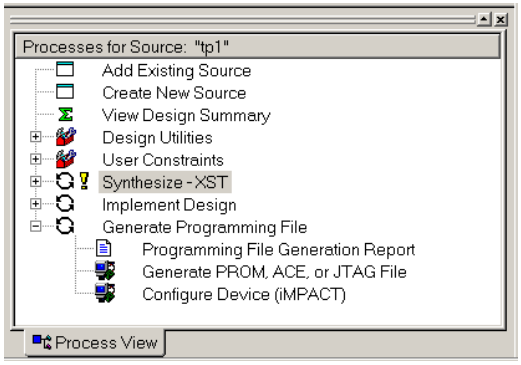


La synthèse démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant son déroulement apparaît :

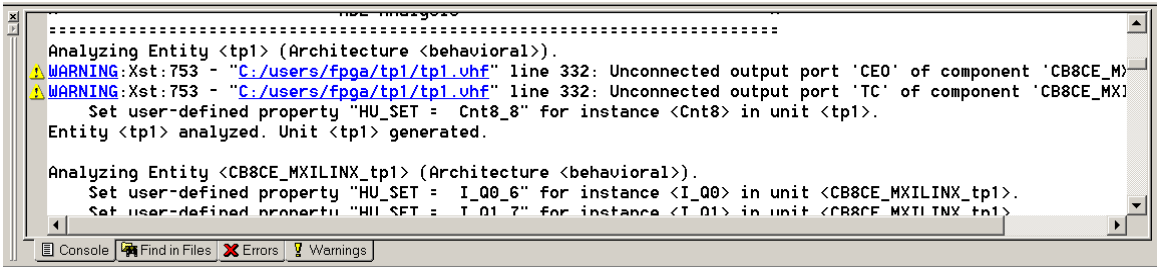
Lorsque la synthèse est finie, vous devez voir une estimation de la fréquence maximale de fonctionnement, ce qui indique que la synthèse s'est bien terminée.



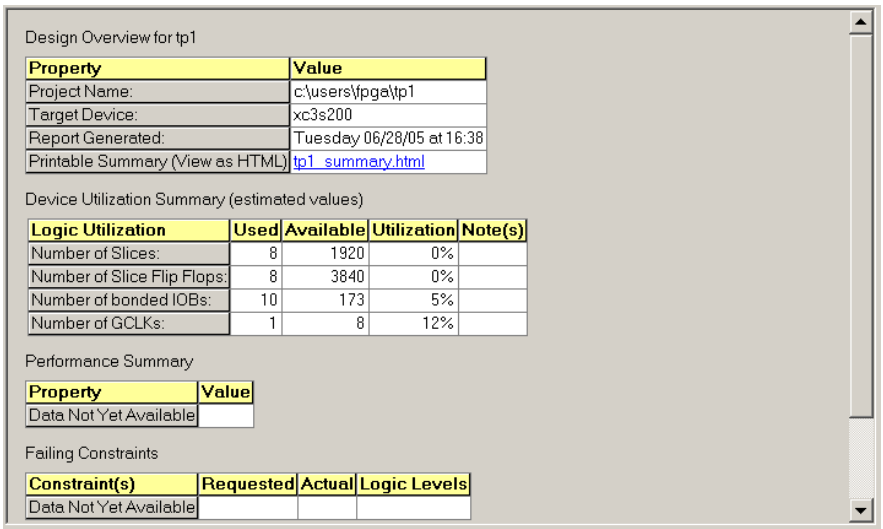
Un point d'exclamation jaune apparaît dans la fenêtre Processes :



Il indique que des messages d'avertissement (Warnings) ont été émis pendant la synthèse. Vous pouvez voir ces Warnings en faisant défiler le rapport dans la Console à l'aide de la barre de défilement de droite. Ils concernent des sorties du compteur qui sont non-connectées. C'est normal, nous pouvons donc les ignorer.



Vous ne pouvez pas visualiser le fichier NGC issu de la synthèse car il s'agit d'un format de netlist binaire qui ne peut donc être visualisé avec un éditeur ASCII. La fenêtre de résumé a évolué et reflète maintenant les informations de synthèse :



Design Overview for tp1

Property	Value
Project Name:	c:\users\fpga\tp1
Target Device:	xc3s200
Report Generated:	Tuesday 06/28/05 at 16:38
Printable Summary (View as HTML)	tp1_summary.html

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	8	1920	0%	
Number of Slice Flip Flops:	8	3840	0%	
Number of bonded IOBs:	10	173	5%	
Number of GCLKs:	1	8	12%	

Performance Summary

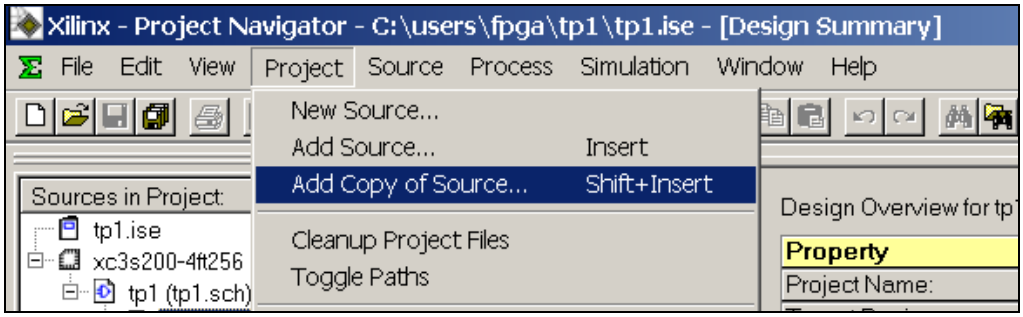
Property	Value
Data Not Yet Available	

Failing Constraints

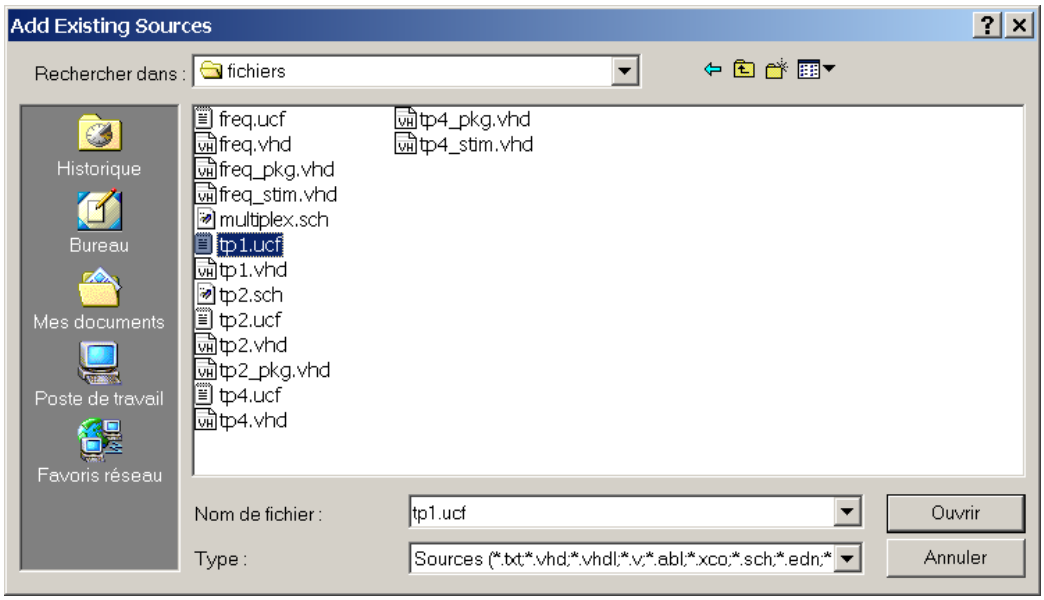
Constraint(s)	Requested	Actual	Logic Levels
Data Not Yet Available			

7.1.8 Implémentation

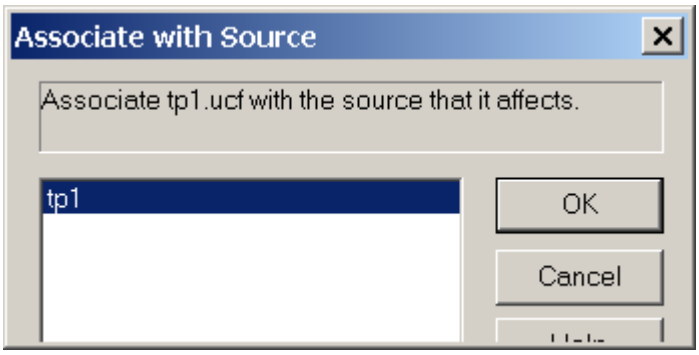
Nous pouvons maintenant travailler sur le circuit FPGA lui-même. C'est le rôle des outils d'implémentation. Nous allons commencer par affecter les broches Clock, Reset et Data aux broches du FPGA selon le câblage de la maquette (voir §1.4). Vous utiliserez pour cela un fichier de contraintes utilisateur déjà écrit qui se nomme tp1.ucf (User Constraint File). Pour l'ajouter au projet tp1, sélectionnez le menu Project, Add Copy of Source :



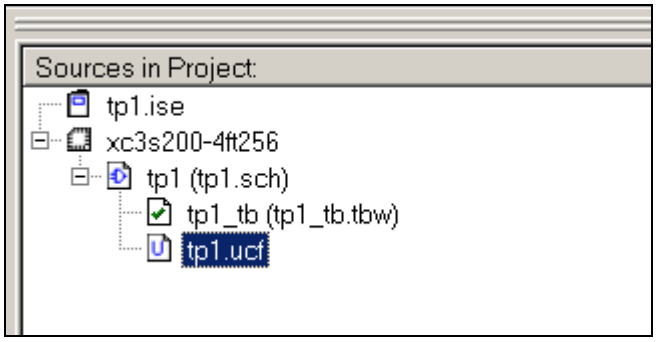
Dans la fenêtre qui s'ouvre, sélectionnez le répertoire c:\users\fpga\fichiers, cliquez sur le fichier tp1.ucf puis sur « Ouvrir » :



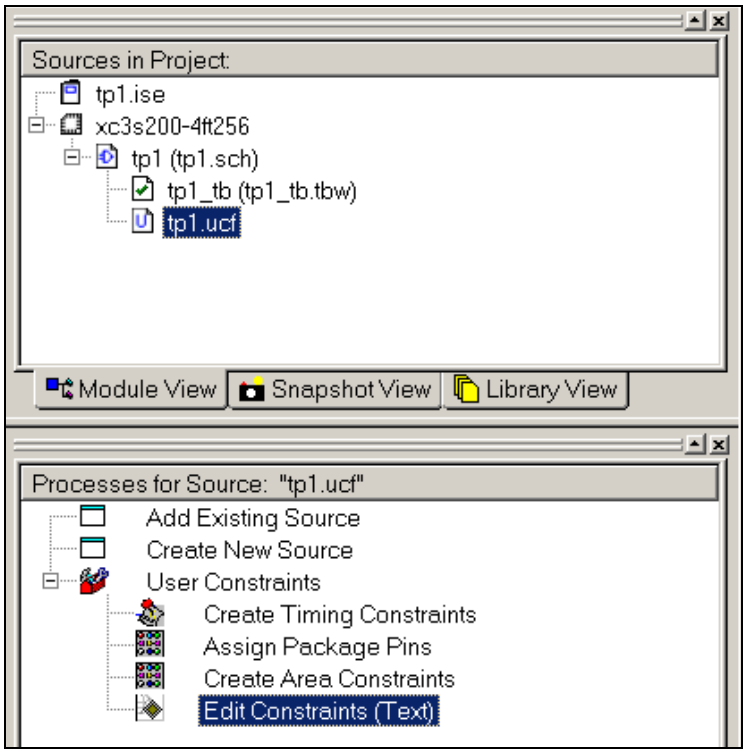
Cliquez sur le bouton « OK » dans la petite fenêtre qui s'ouvre alors :



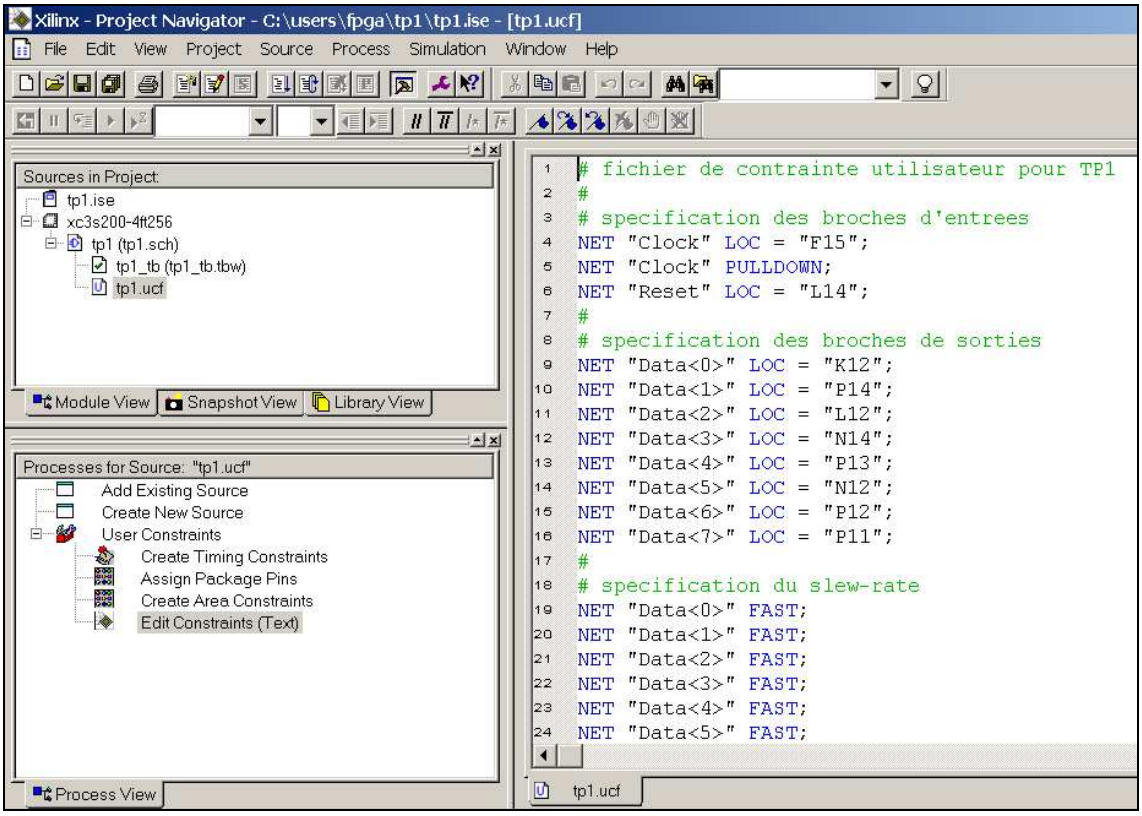
Le fichier est maintenant inclus dans le projet :



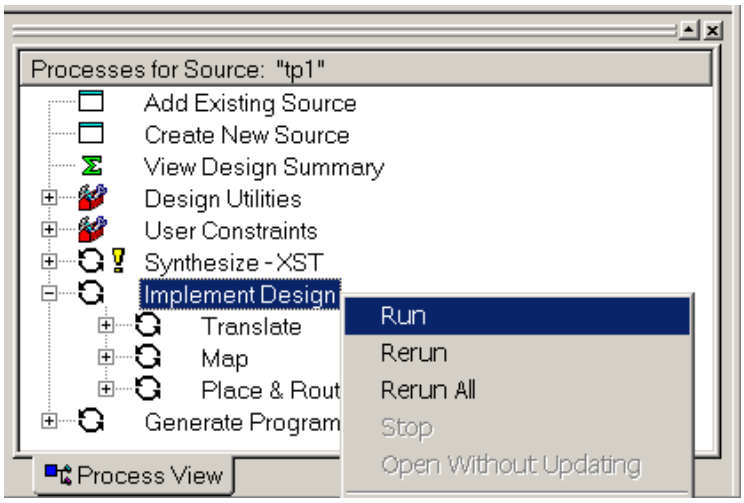
Pour voir le contenu de ce fichier, il suffit de le sélectionner dans la fenêtre « Sources », puis de double-cliquer dans la fenêtre Processes sur « Edit Constraints (Text) » :



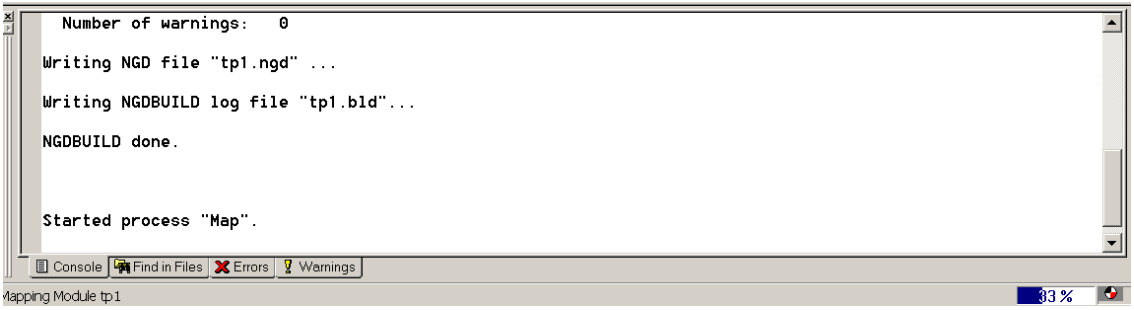
Le contenu du fichier apparaît alors dans l'éditeur :



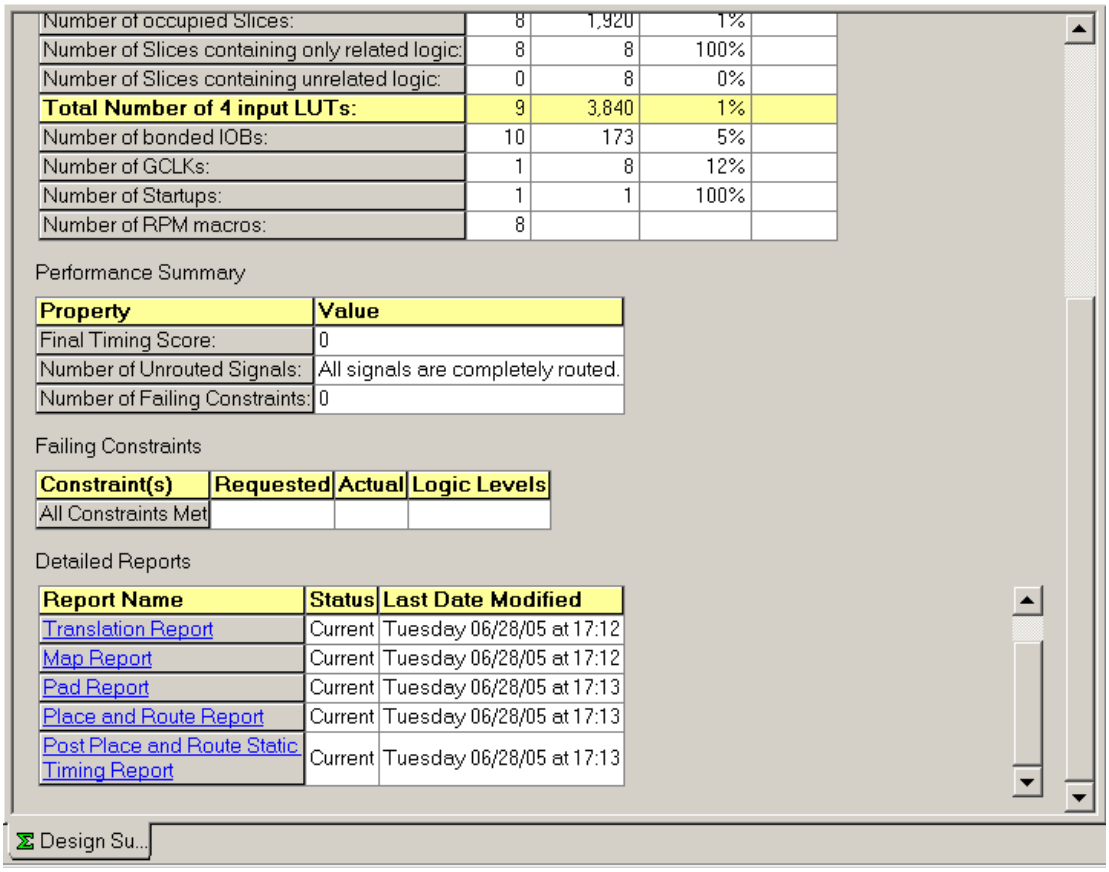
Ne modifiez surtout pas ce fichier et fermez l'éditeur. Nous pouvons lancer l'implantation du FPGA (implementation en anglais). Pour cela, sélectionnez le design tp1.sch dans la fenêtre « Sources » puis « Implement Design » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



L'implémentation démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



Lorsque l'implémentation est finie, vous devez voir le message : PAR done ! qui indique que l'implémentation s'est bien terminée. Des points d'exclamation jaunes peuvent apparaître dans la fenêtre Processes. Ils indiquent que des messages d'avertissement (Warnings) ont été émis pendant l'implémentation. Vous pouvez voir ces Warnings en faisant défiler le rapport dans la Console à l'aide de la barre de défilement de droite. Il ne doit pas y en avoir dans notre exemple. Un certain nombre de rapports concernant les différentes étapes de l'implémentation sont maintenant disponibles dans la fenêtre Design Summary :



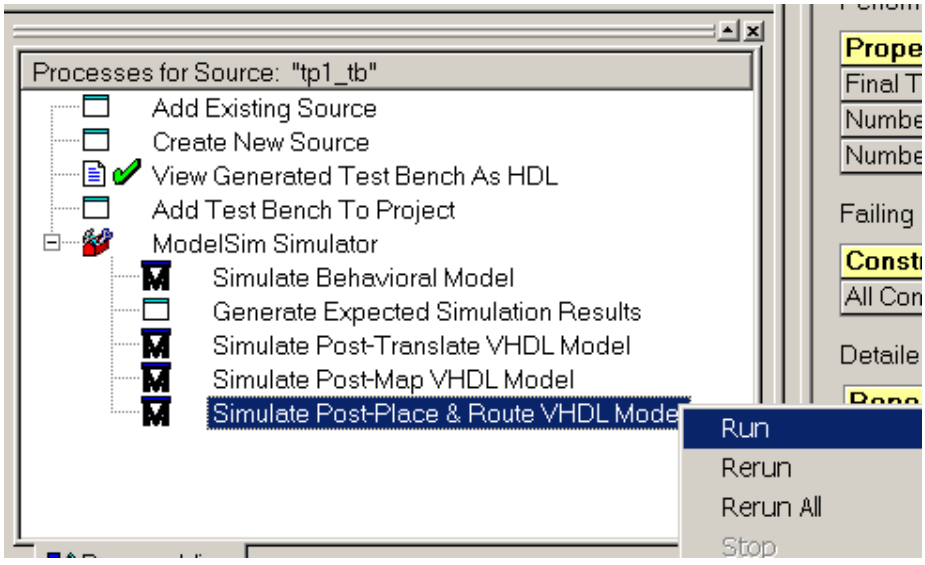
Le tableau suivant vous indique le rôle des différentes étapes de l'implémentation ainsi que les rapports associés :

Etape	Rapport	Signification
Translation	Translation report	Création d'un fichier de design unique
Mapping	Map report	Découpage du design en primitives
Placement-routage	Place & Route report	Placement et routage des primitives
	Post Place & Route static timing report	Respect des contraintes temporelles et fréquence max de fonctionnement
	Pad report	Assignation des broches du FPGA

7.1.9 Simulation de timing

Les outils de placement-routage peuvent fournir un nouveau modèle VHDL (tp1_timesim.vhd) qui correspond au modèle de simulation réel du circuit ainsi qu'un fichier contenant tous les timings du FPGA (tp1_timesim.sdf). On l'appelle le **modèle VITAL**. A l'aide de ces deux fichiers et du fichier de stimuli tp1_tb.timesim_vhw (généré précédemment par Waveform Editor), nous allons pouvoir vérifier le fonctionnement réel de notre design. C'est la simulation de timing (ou simulation Post-Place&Route ou encore simulation Post-layout).

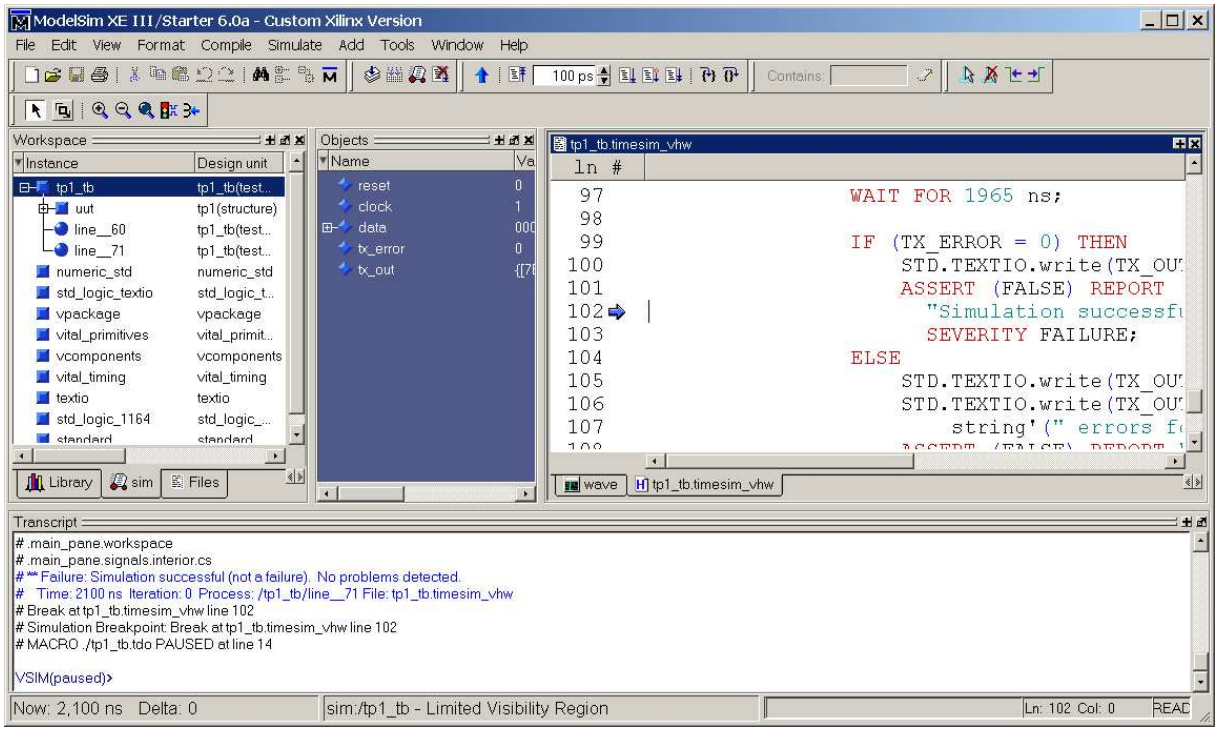
Pour lancer cette simulation, sélectionnez le fichier tp1_tb.tbw dans la fenêtre « Sources In Project ». Sélectionnez le processus « Simulate Post-Place&Route VHDL Model » puis cliquez avec le bouton droit de la souris sur le menu Run :



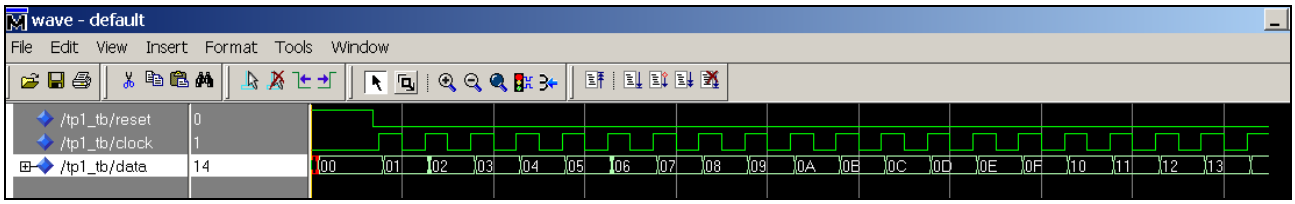
Le simulateur VHDL ModelSim démarre. Il va enchaîner automatiquement :

- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

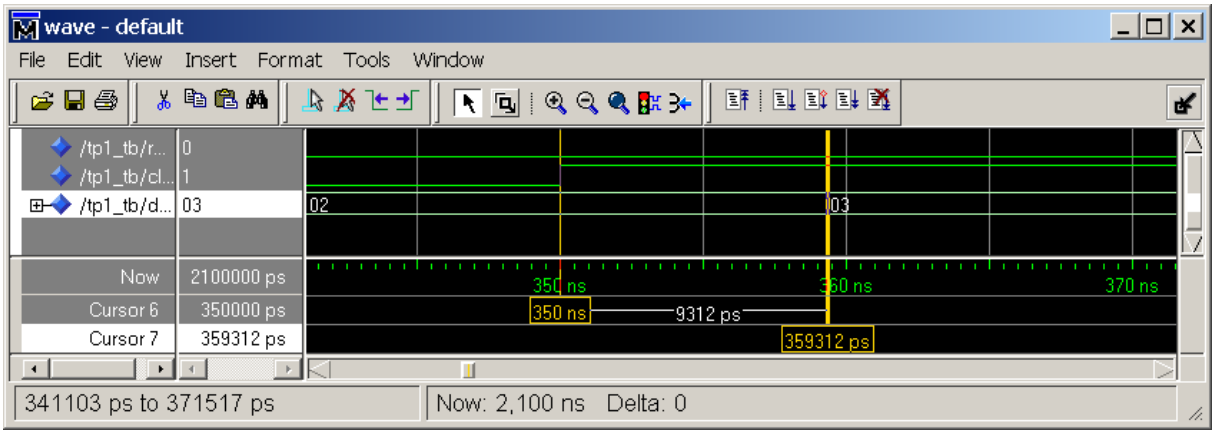
A la fin, la fenêtre Modelsim est la suivante :



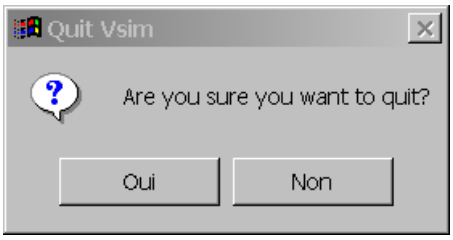
Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) puis détachez-la de la fenêtre ModelSim. Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation (de 0 à 2100 ns). Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes. Sélectionnez le bus data puis cliquez avec le bouton droit de la souris sur le menu Radix, puis sur Hexadécimal.



Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Sur l'exemple de chronogramme suivant, on voit clairement apparaître le décalage entre l'horloge et les sorties. Utilisez les curseurs temporels pour mesurer l'écart entre le front d'horloge actif et le changement en sortie.

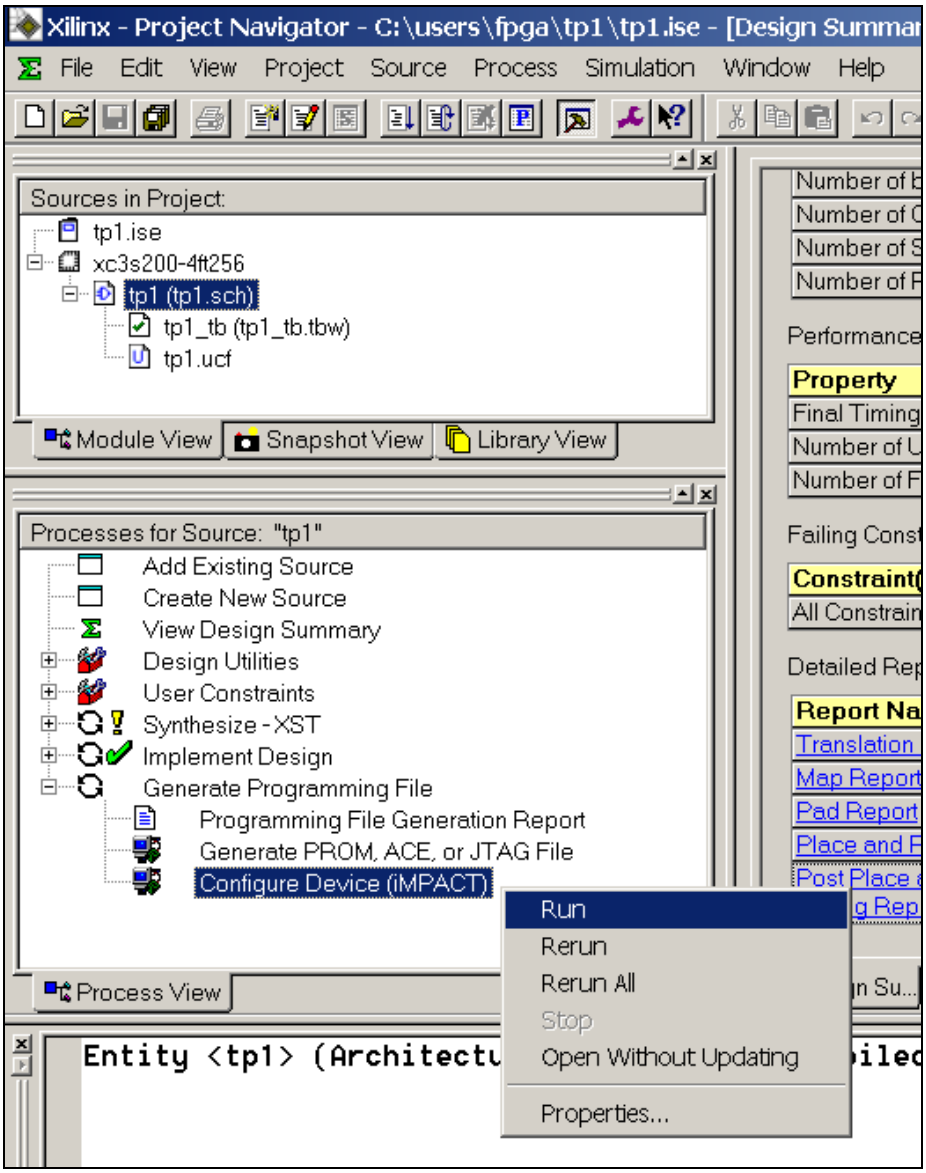


Quand la vérification est terminée, cliquez sur le menu « File » dans la fenêtre ModelSim, puis sur « Quit». Quand le message ci-dessous apparaît à l'écran, cliquez sur Oui.

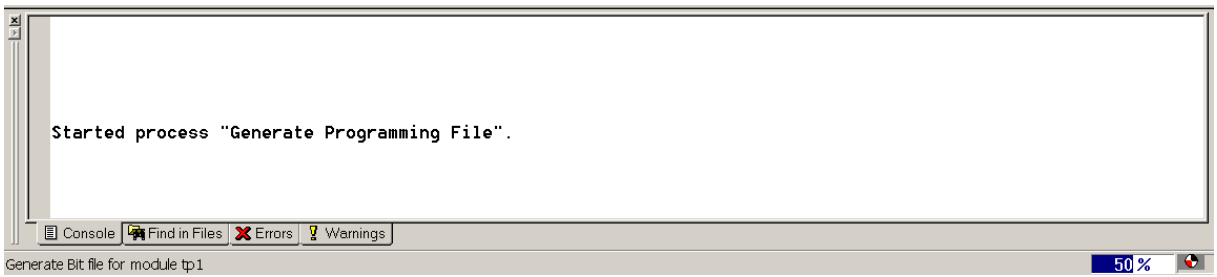


7.1.10 Configuration de la maquette

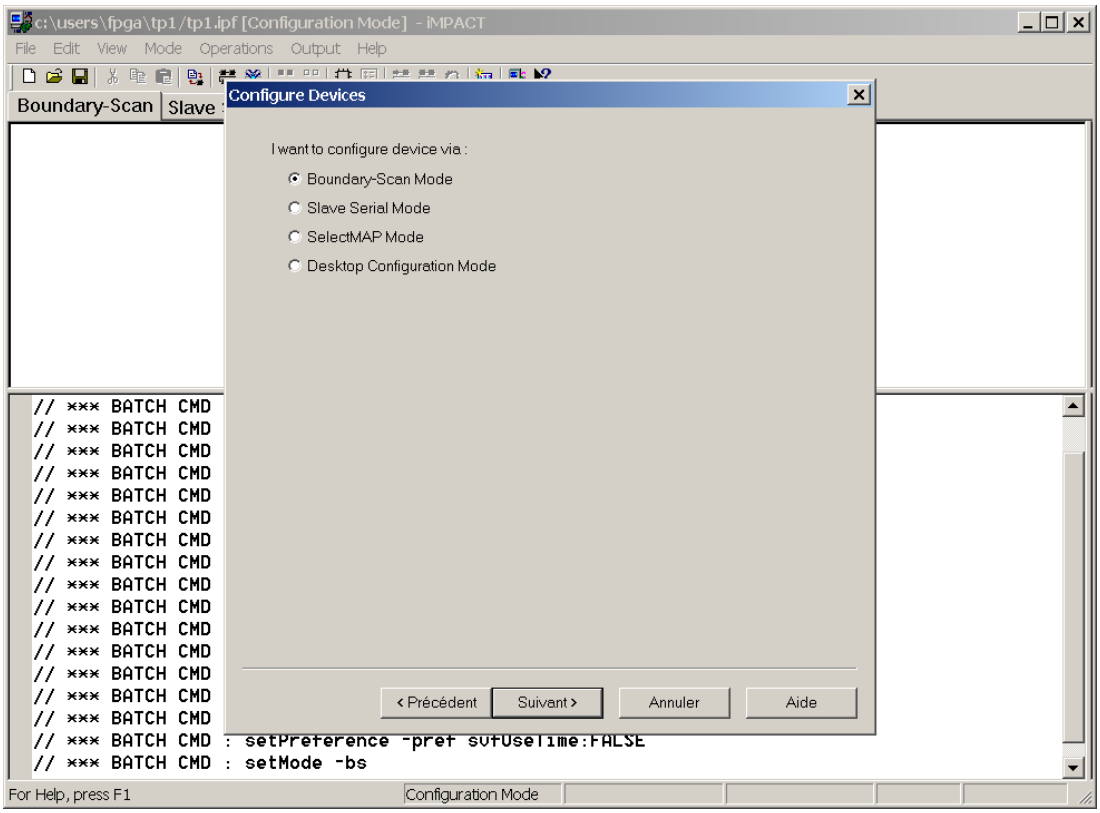
A ce point du TP, le design est entièrement vérifié. Nous pouvons maintenant le télécharger dans le FPGA. Sélectionnez le design tp1.sch dans la fenêtre « Sources » puis « Configure Device » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



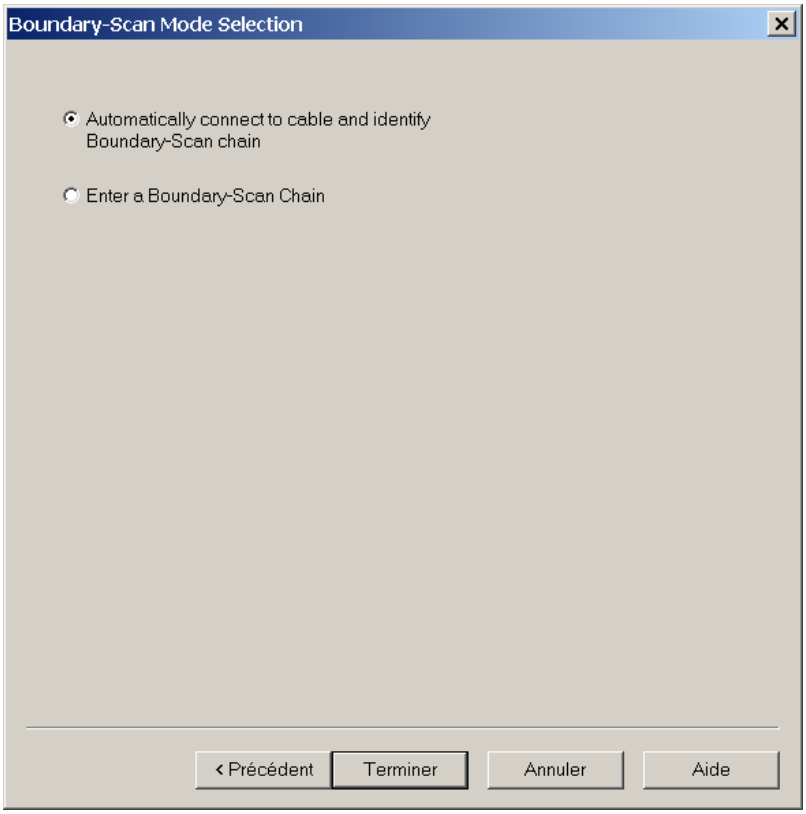
Le processus démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



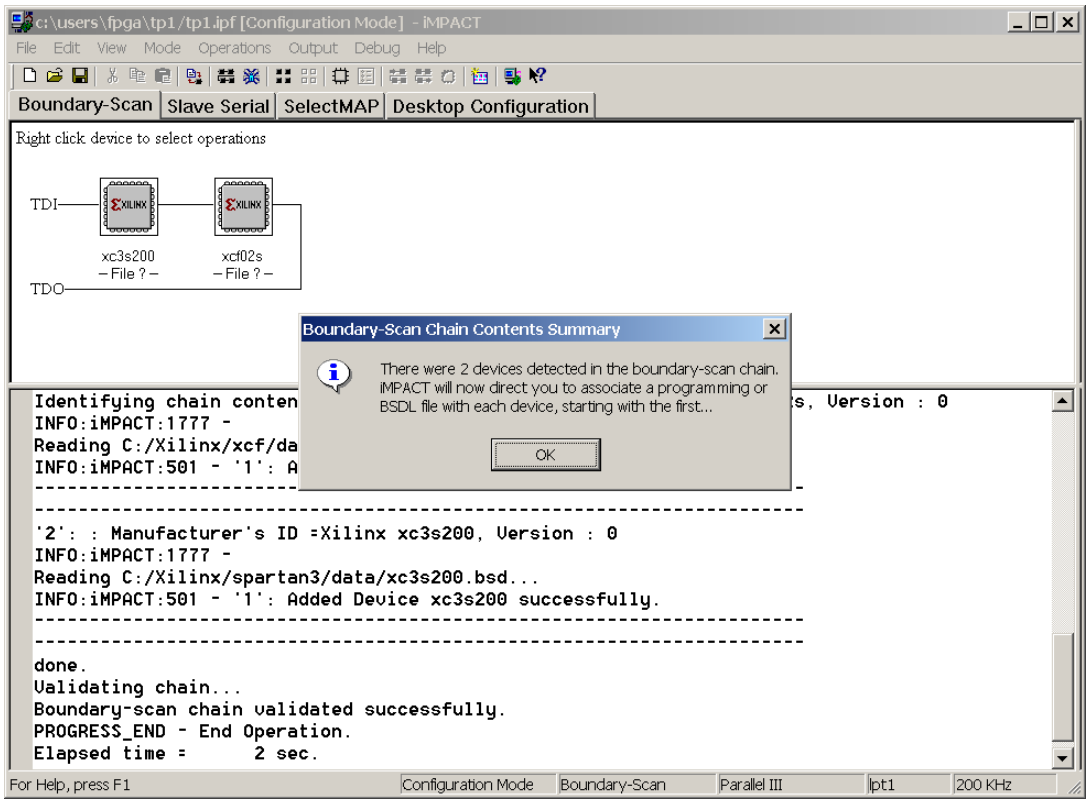
Puis la fenêtre de l'application Impact apparaît à l'écran. Nous allons configurer le FPGA en utilisant le mode « Boundary-scan » (JTAG). Cliquez sur le bouton « Suivant » :



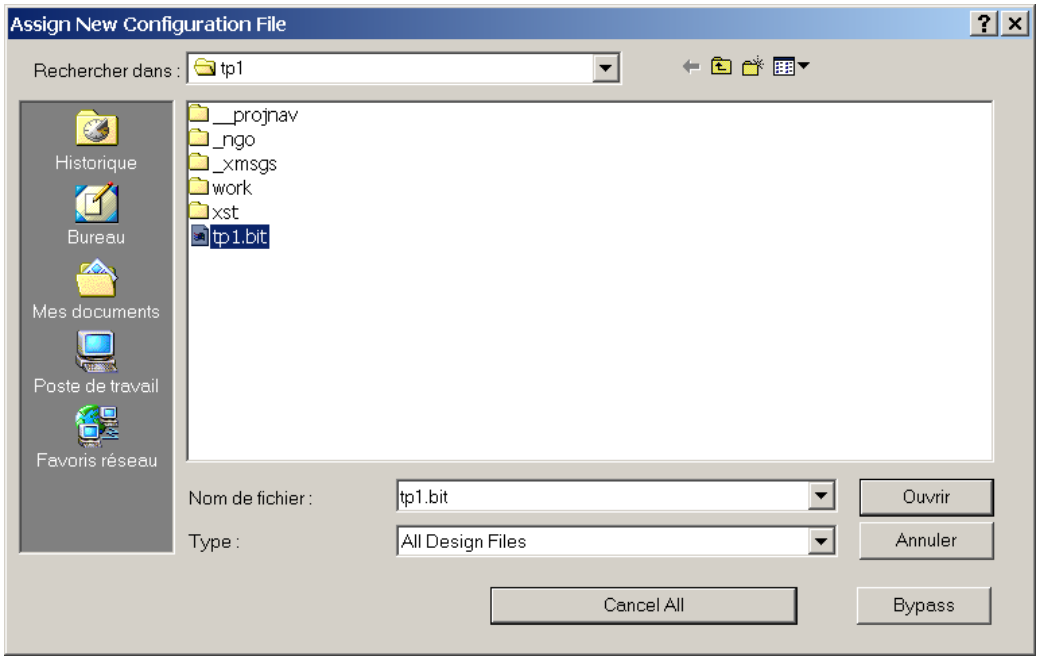
Laissons Impact découvrir automatiquement les circuits connectés sur la chaîne JTAG. Cliquez sur le bouton « Terminer » :



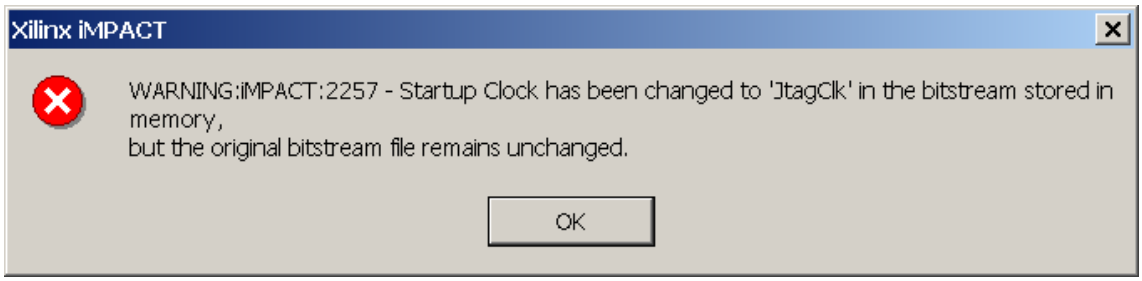
Impact a trouvé 2 circuits dans la chaîne JTAG, le FPGA et une mémoire Flash série que nous n'allons pas utiliser maintenant. Cliquez sur le bouton « OK » :



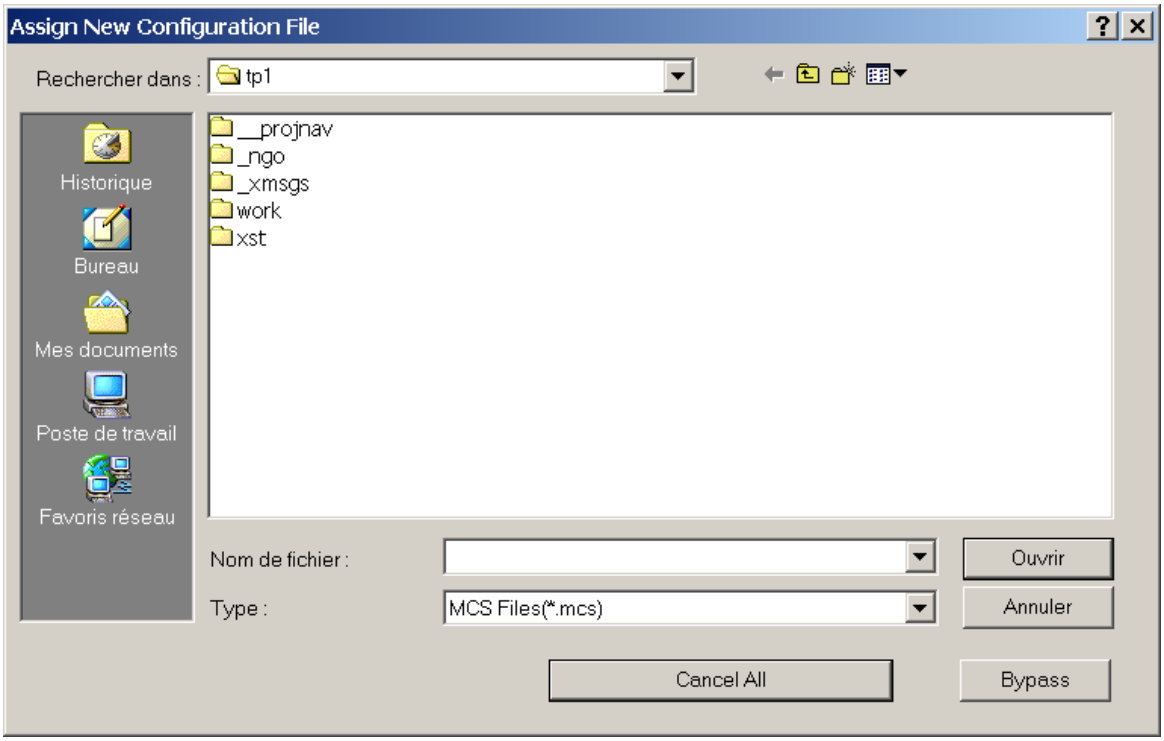
La question suivante concerne le nom du fichier de configuration à associer au FPGA. Dans notre exemple, il s'agit de tp1.bit. Sélectionnez-le, puis cliquez sur « Ouvrir » :



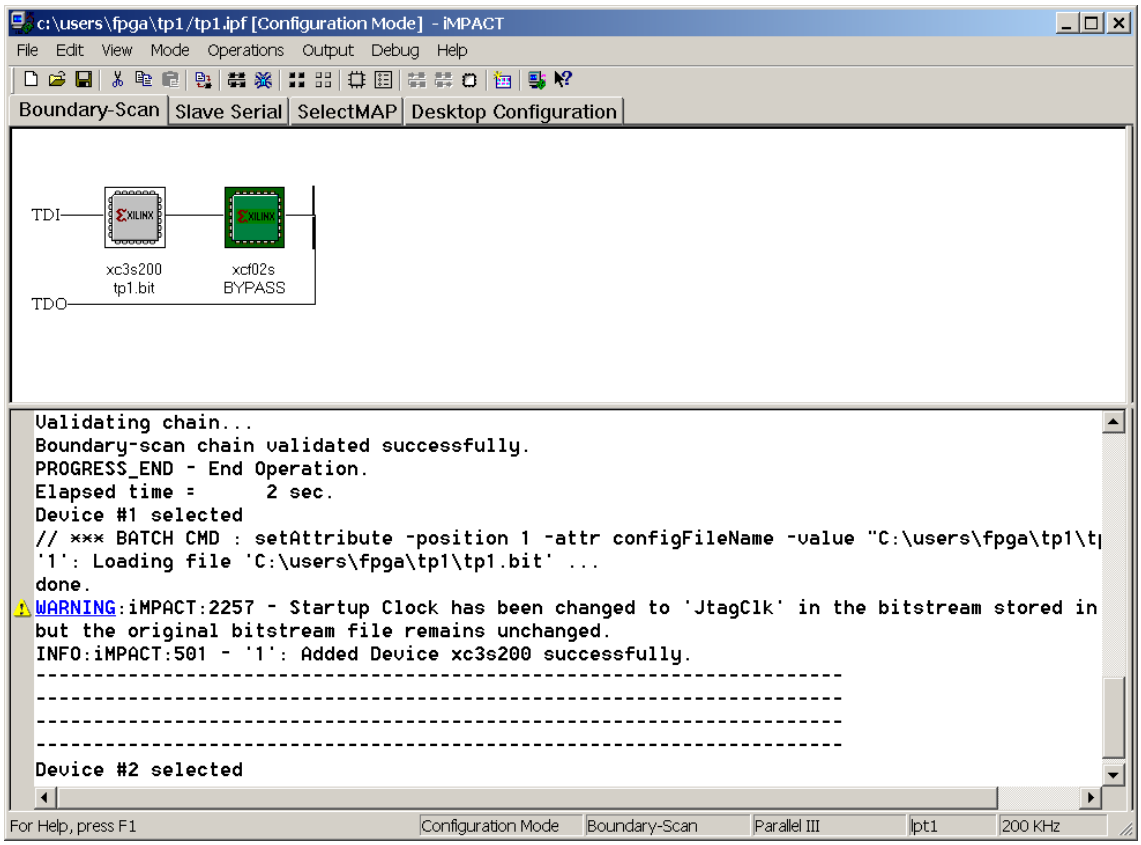
Par défaut, le FPGA est en mode maître, c'est à dire qu'il génère l'horloge CCLK de lecture de la configuration. Comme en JTAG, le FPGA est esclave, la fenêtre suivante vous avertit que l'horloge dans le fichier de configuration tp1.bit a été passée en mode JTAG (sur la copie en mémoire seulement). Cliquez sur « OK » pour poursuivre :



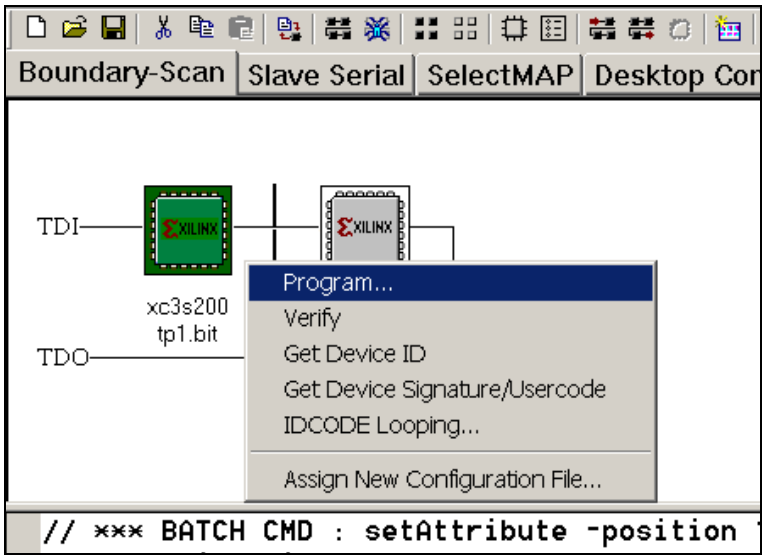
Impact demande ensuite quel fichier va être associé avec la mémoire Flash série. Comme nous n'avons pas créé le fichier pour la PROM (format MCS-86), cliquez sur « Bypass » pour sauter cette étape :



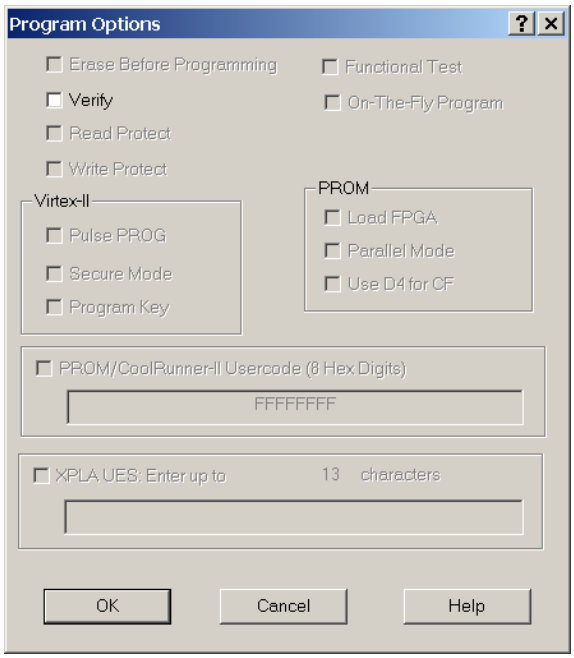
Finalement, on obtient la fenêtre suivante :



Pour configurer le FPGA, il suffit de le sélectionner en cliquant dessus, puis de cliquer avec le bouton droit de la souris puis sur « Program… » :



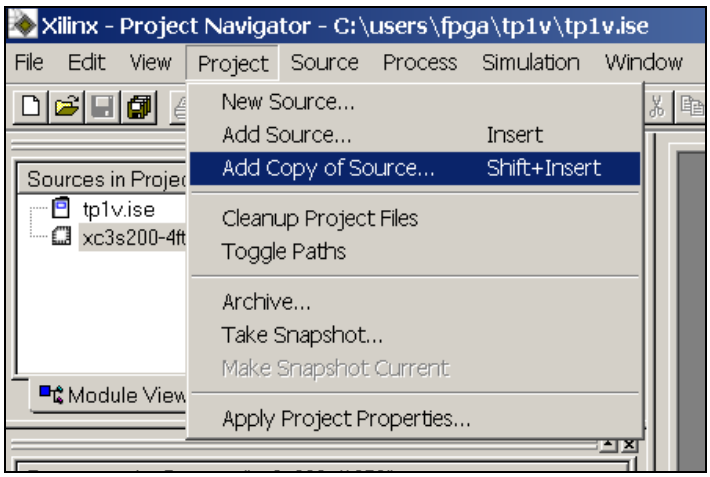
Dans la fenêtre qui s'ouvre, cliquez sur « OK » :



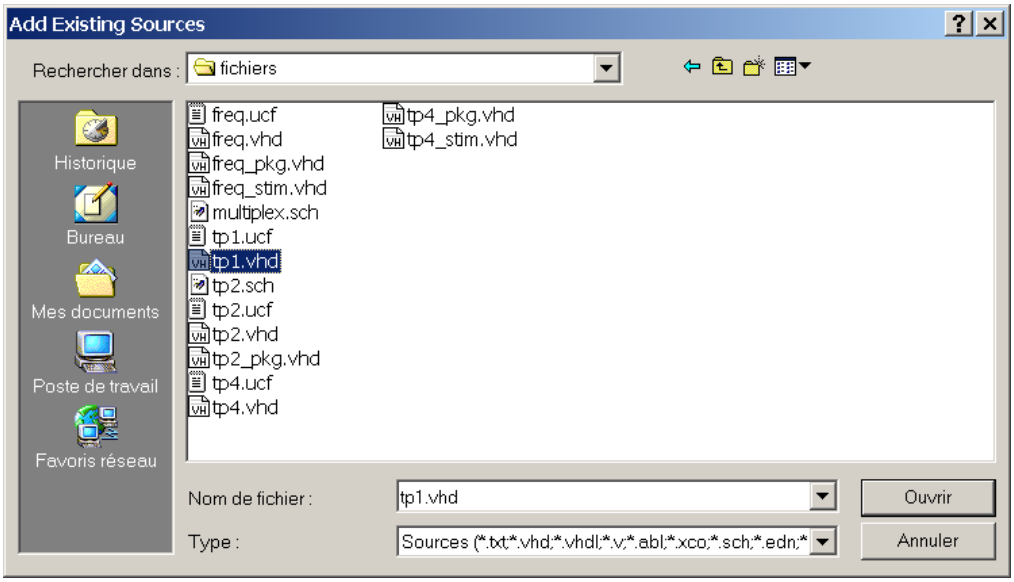
En moins de 10 secondes, le téléchargement est terminé. Fermez la fenêtre Impact sans sauver le projet. Connectez un générateur d'horloge compatible CMOS 3.3V (entre 0 et 3 Volt, fréquence ≈ 10 Hz) sur l'entrée Hin (qui est chargée sur 50 Ω). Vérifiez le comptage sur les LED. Le bouton poussoir BTN3 (User Reset) réinitialise le montage.

7.1.11 Le flot VHDL

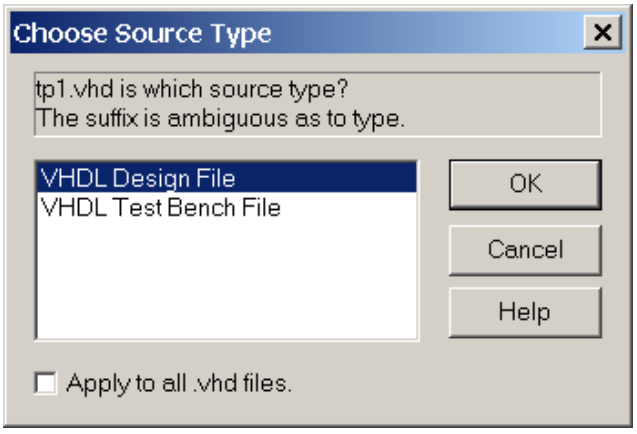
Nous allons maintenant refaire le même TP avec un design écrit directement en langage VHDL. Pour cela, fermez le projet TP1 (menu « File », « Close Project ») puis créez en un nouveau appelé TP1V (**Top-Level Module Type : HDL**). Le compteur 8 bits est déjà écrit en VHDL (tp1.vhd) et se trouve dans le répertoire c:\users\fpga\fichiers. Pour l'insérer dans le projet TP1V, cliquez sur « Project », « Add Copy of Source ».



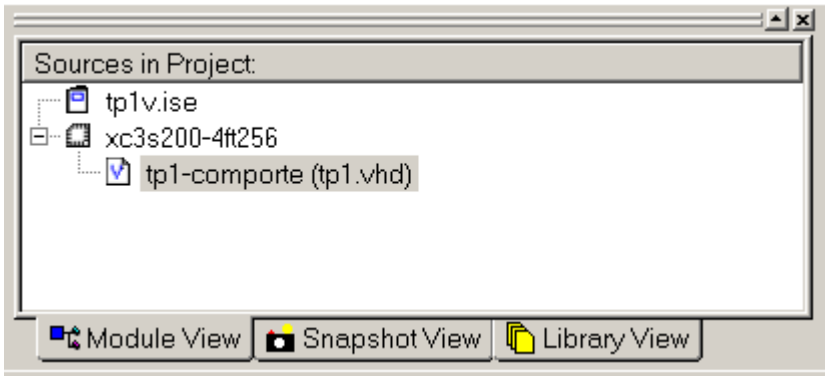
Sélectionnez le fichier tp1.vhd puis cliquez sur ouvrir :



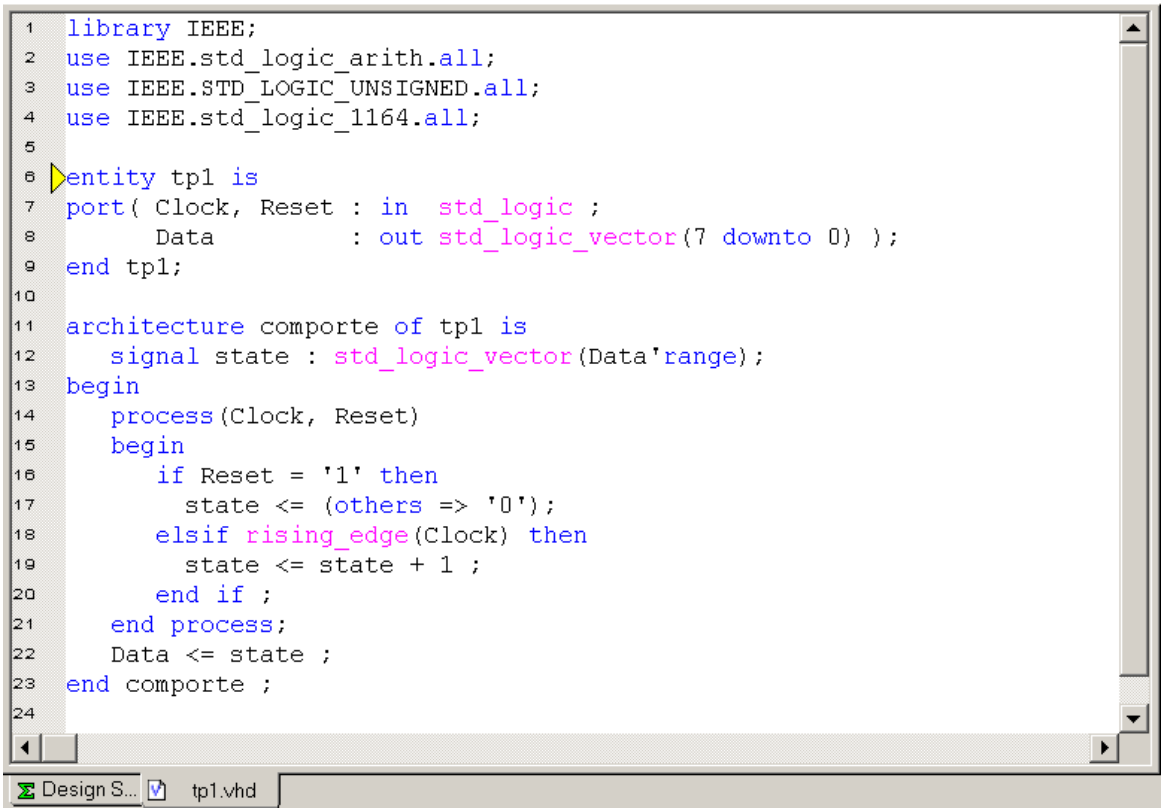
Et enfin sélectionnez « VHDL Design File » puis cliquez OK.



Le design apparaît maintenant dans la fenêtre Sources.



Pour l'éditer, sélectionnez-le, cliquez avec le bouton droit de la souris puis sur « Open ». Le contenu du fichier apparaît dans la fenêtre d'édition.

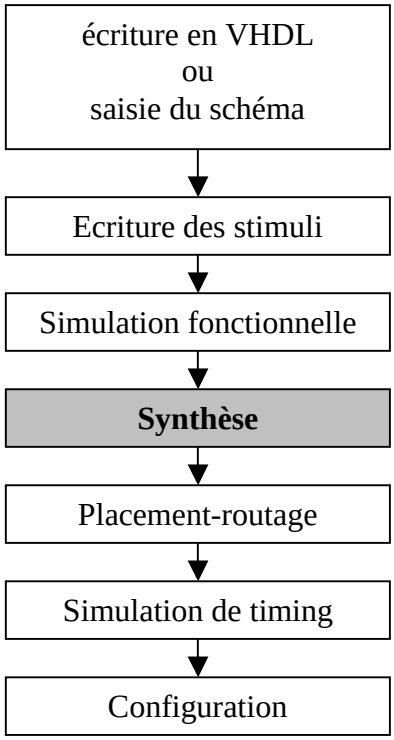


Il faut bien comprendre que le schéma dans le TP précédent était sauvegardé sous la forme d'une netlist écrite en VHDL alors que dans ce TP, le code VHDL est d'un niveau d'abstraction bien plus élevé. Dans les deux cas, la synthèse est obligatoire, mais pas pour la même raison.

En saisie de schéma, le design est composé de primitives simples contenues dans une bibliothèque fournie par Xilinx. La netlist VHDL correspondante doit donc simplement être traduite en une netlist NGC pour être compréhensible par les outils d'implémentation. Le synthétiseur XST est utilisé comme un simple traducteur VHDL-NGC.

La description VHDL tp1.vhd n'utilise pas de bibliothèque propriétaire et elle est beaucoup plus générale. Le rôle du synthétiseur est ici de comprendre et d'interpréter cette description plus abstraite du compteur afin de générer un fichier NGC compréhensible par les outils d'implémentation, c'est-à-dire une netlist NGC composée de primitives simples.

A par cette différence importante sur le rôle du synthétiseur, le flot de conception est identique :



Le déroulement de ce TP est identique au précédent. Reportez-vous au texte de tp1 pour poursuivre tp1v.

7.1.12 Fichier de contraintes

```
# fichier de contrainte utilisateur pour TP1
#
# specification des broches d'entrees
NET "Clock" LOC = "F15";
NET "Clock" PULLDOWN;
NET "Reset" LOC = "L14";
#
# specification des broches de sorties
NET "Data<0>" LOC = "K12";
NET "Data<1>" LOC = "P14";
NET "Data<2>" LOC = "L12";
NET "Data<3>" LOC = "N14";
NET "Data<4>" LOC = "P13";
NET "Data<5>" LOC = "N12";
```

```
NET "Data<6>" LOC = "P12";
NET "Data<7>" LOC = "P11";
#
# specification du slew-rate
NET "Data<0>" FAST;
NET "Data<1>" FAST;
NET "Data<2>" FAST;
NET "Data<3>" FAST;
NET "Data<4>" FAST;
NET "Data<5>" FAST;
NET "Data<6>" FAST;
NET "Data<7>" FAST;
```

7.2 Travail pratique N°2

Ce deuxième TP a pour but de permettre la prise en main des outils de CAO Xilinx sur un exemple en VHDL simple : le design « portes combinatoires » du §2.4.2 du cours. Les entrées seront reliées sur les interrupteurs de la maquette FPGA et les sorties sur les leds. Nous allons passer en revue toutes les phases du développement d'un design en VHDL avec création d'un fichier UCF.

7.2.1 Ouverture de session

Le nom de l'utilisateur est fpga, le mot de passe est fpga.

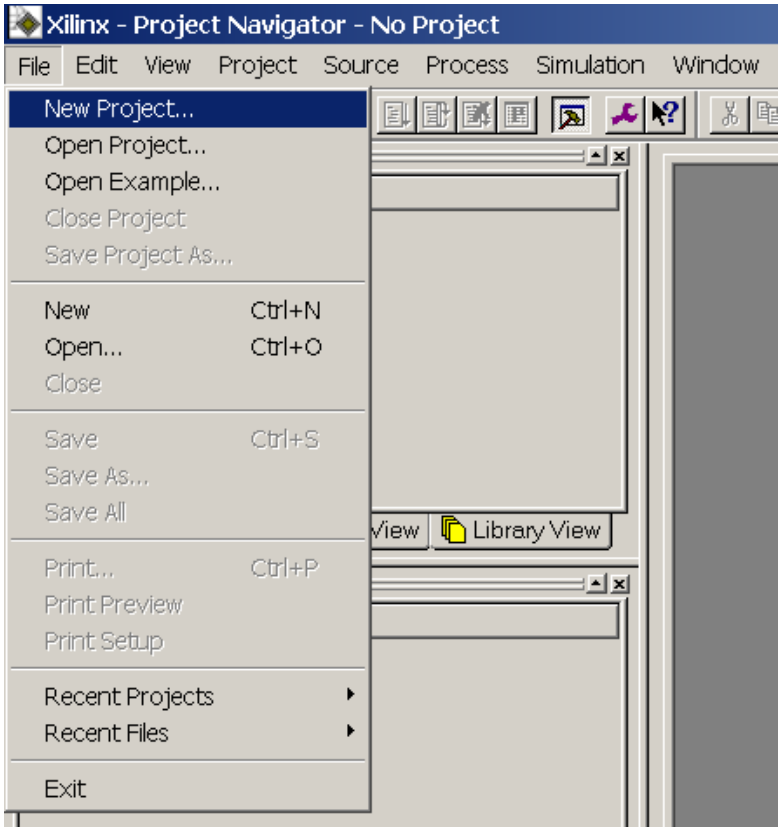
7.2.2 Lancement de « Project Navigator »

Le lancement de l'application principale « Project Navigator » s'effectue en cliquant deux fois sur l'icône 

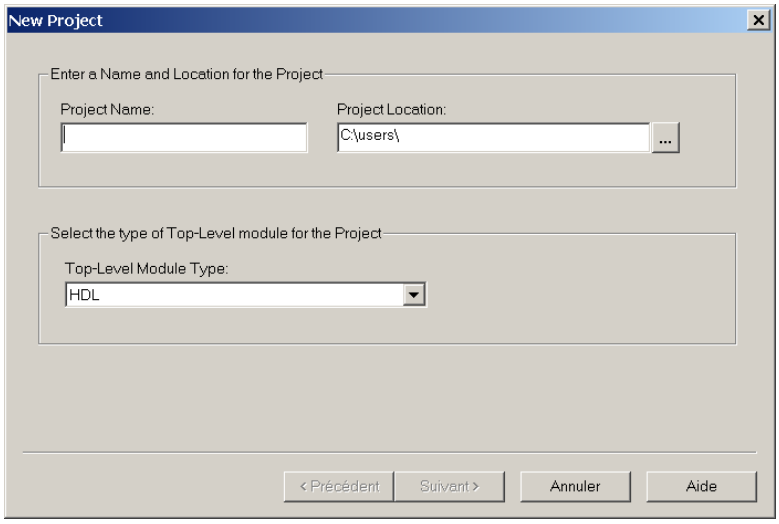
se trouvant sur le bureau.

7.2.3 Création du projet

Après un long moment (30 secondes la première fois), la fenêtre de « Project Navigator » s'ouvre. Cliquez sur le menu « File » et le sous-menu « New Project…».



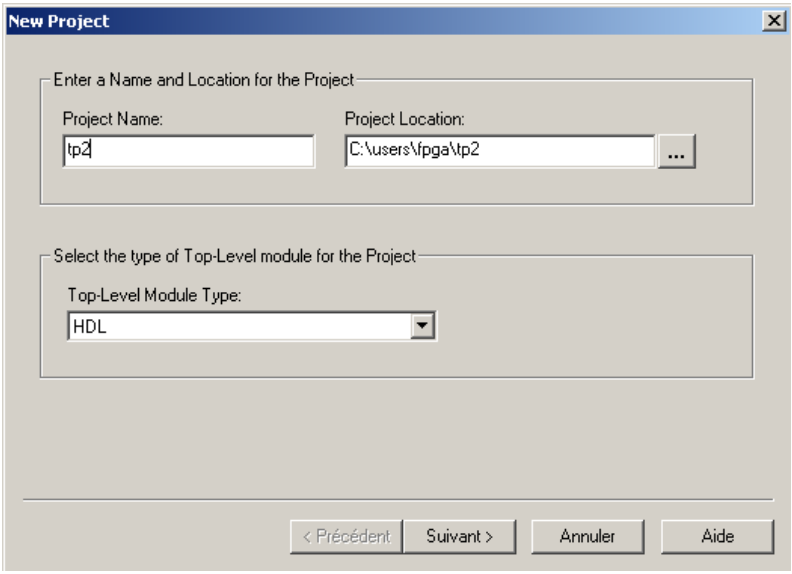
La fenêtre de création du projet apparaît.



Dans l'ordre suivant :

4. Tapez le répertoire du projet c:\users\fpga dans le champ « Project Location »,
5. Tapez le nom du projet tp2 dans le champ « Project Name »,
6. Sélectionnez le type HDL pour le design principal.

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

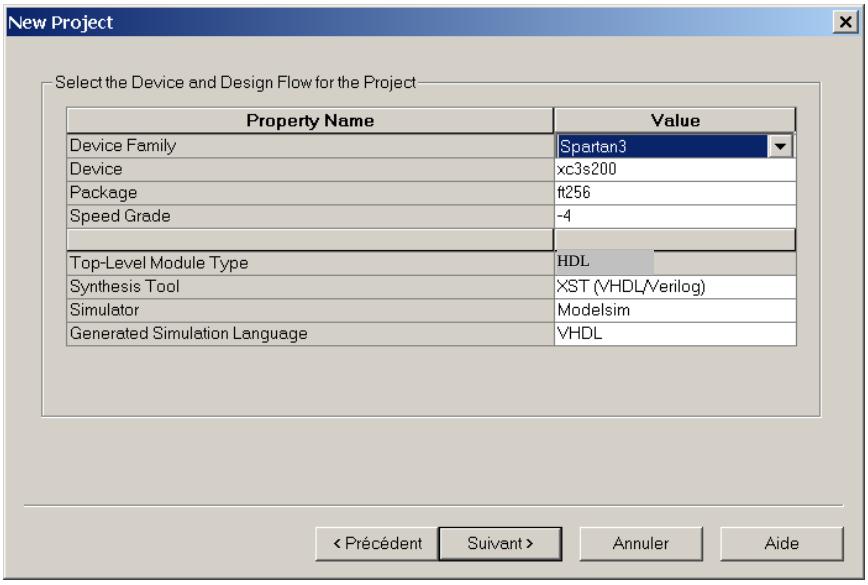


Dans la fenêtre qui s'ouvre, sélectionnez :

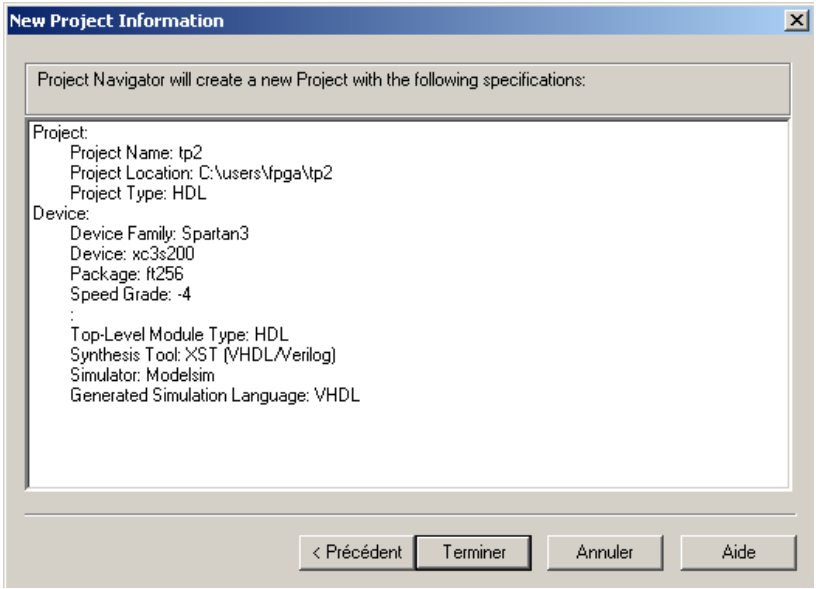
- la famille de FPGA utilisée (Spartan3 dans le champ « Device Family »),

- le circuit utilisé (xc3s200 dans le champ « Device »),
- le boîtier (ft256 dans le champ « Package »),
- la vitesse (-4 dans le champ « Speed Grade »),
- les outils du flot de développement (synthétiseur : XST, simulateur : modelsim, langage VHDL).

Vous devez finalement obtenir la fenêtre suivante avant de cliquer sur « Suivant » :

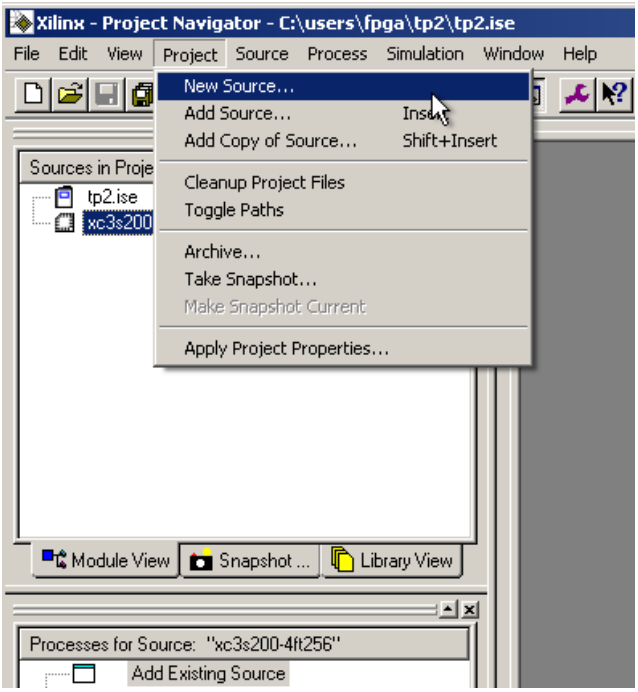


Cliquez sur « Suivant » dans les deux fenêtres suivantes, puis sur « Terminer » dans la fenêtre finale :

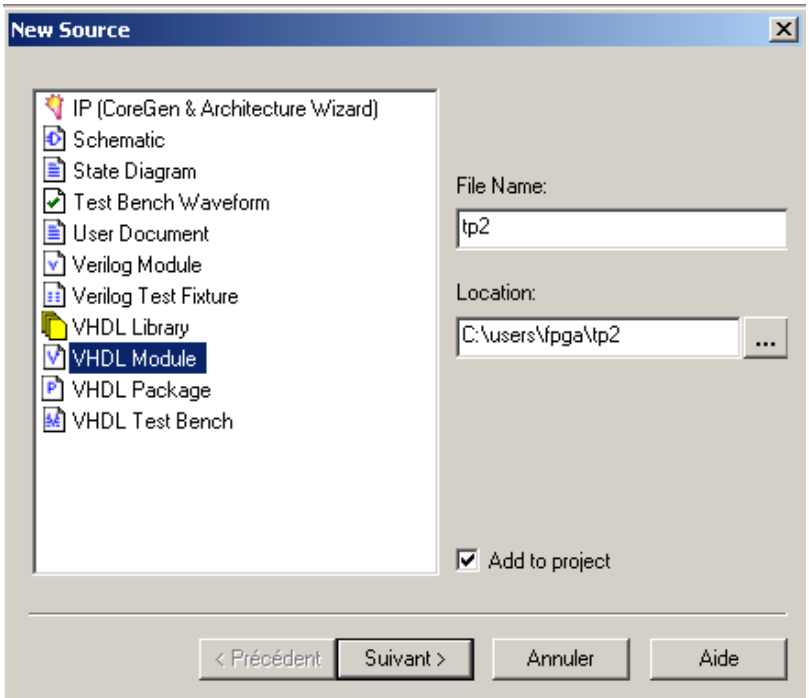


7.2.4 Création du design VHDL

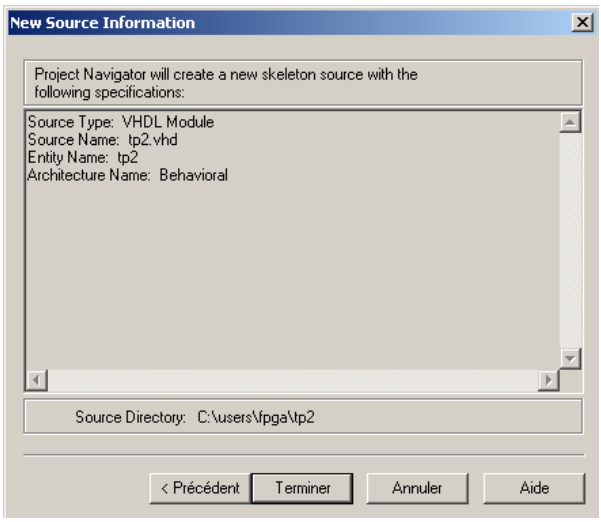
La saisie du design s'effectue à l'aide de l'éditeur intégré. Pour le lancer, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



La fenêtre suivante s'ouvre. Sélectionnez « VHDL Module », tapez le nom du design tp2 dans le champ « File Name » puis cliquez sur « Suivant ».



Cliquez sur « Suivant » dans la fenêtre suivante, puis sur « Terminer » dans la fenêtre d'information :



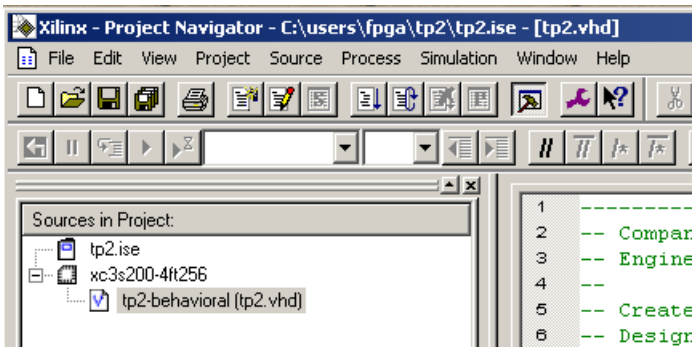
ISE crée automatiquement un fichier VHDL qui contient l'appel de certaines librairies et la déclaration de l'entité et de l'architecture. Ce fichier est ouvert automatiquement dès sa création :

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    10:45:25 11/07/13
6  -- Design Name:
7  -- Module Name:    tp2 - Behavioral
8  -- Project Name:
9  -- Target Device:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22 use IEEE.STD_LOGIC_ARITH.ALL;
23 use IEEE.STD_LOGIC_UNSIGNED.ALL;
24
25 ---- Uncomment the following library declaration if instantiating
26 ---- any Xilinx primitives in this code.
27 --library UNISIM;
28 --use UNISIM.VComponents.all;
29
30 entity tp2 is
31 end tp2;
32
33 architecture Behavioral of tp2 is
34
35 begin
36
37
38 end Behavioral;
```

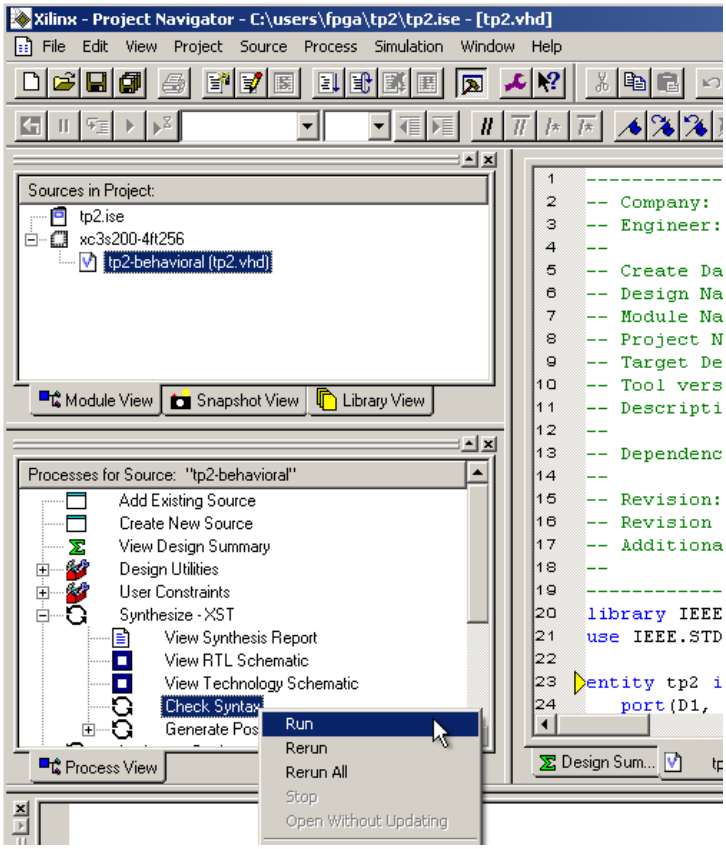
Il reste à le modifier pour obtenir :

```
1  ----------------------------------------------------------------------------------
2  -- Company:
3  -- Engineer:
4  --
5  -- Create Date:    10:45:25 11/07/13
6  -- Design Name:
7  -- Module Name:    tp2 - Behavioral
8  -- Project Name:
9  -- Target Device:
10 -- Tool versions:
11 -- Description:
12 --
13 -- Dependencies:
14 --
15 -- Revision:
16 -- Revision 0.01 - File Created
17 -- Additional Comments:
18 --
19 ----------------------------------------------------------------------------------
20 library IEEE;
21 use IEEE.STD_LOGIC_1164.ALL;
22
23 entity tp2 is
24    port(D1, D2, D3 : in  std_logic;
25         Y1, Y2, Y3, Y4, Y5 : out std_logic);
26 end tp2;
27
28 architecture Behavioral of tp2 is
29    signal tmp : std_logic;
30 begin
31    Y1 <= D1 nor D2;
32    Y2 <= not (D1 or D2 or D3);
33    Y3 <= D1 and D2 and not D3;
34    Y4 <= D1 xor (D2 xor D3);
35    tmp <= D1 xor D2;
36    Y5 <= tmp nand D3;
37 end Behavioral;
38
```

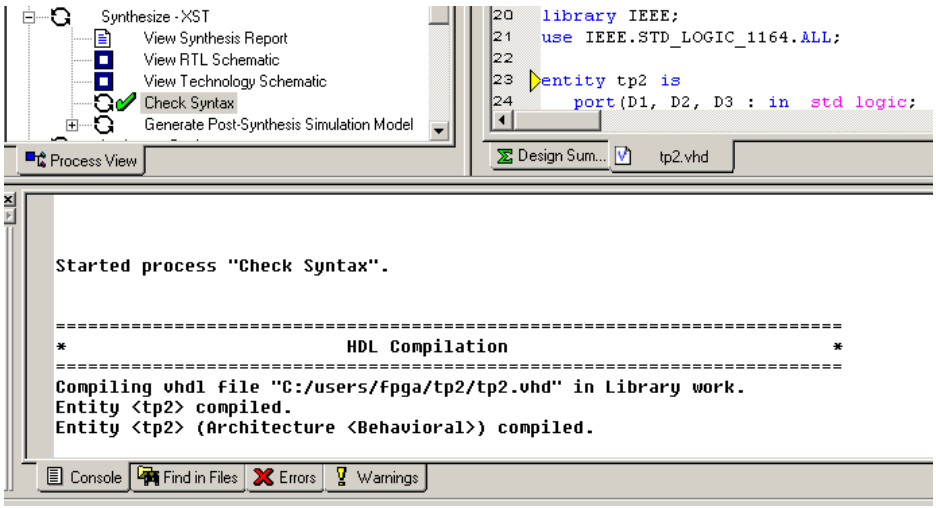
Sauvez votre design en cliquant sur  dans la barre d'outil. Vous devez normalement obtenir la fenêtre suivante dans les sources du navigateur de projet.



Avant toute chose, nous allons vérifier la syntaxe du code VHDL. Pour cela, sélectionnez le fichier tp2.vhd dans la fenêtre « Sources In Project ». Les actions qui peuvent être effectuées avec ce fichier apparaissent dans la fenêtre inférieure « Processes for Source ». Sélectionnez « Synthesize – XST », puis double-cliquez sur « Check Syntax » :



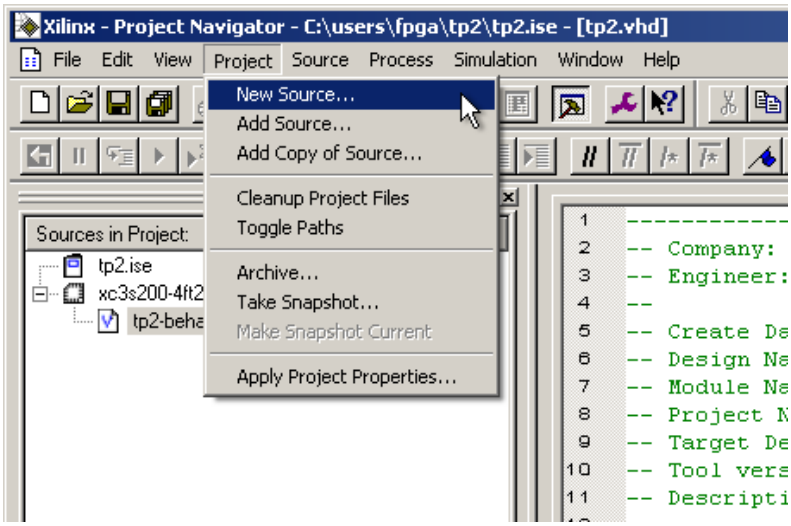
Le rapport de compilation apparaît dans la fenêtre Console du navigateur de projet :



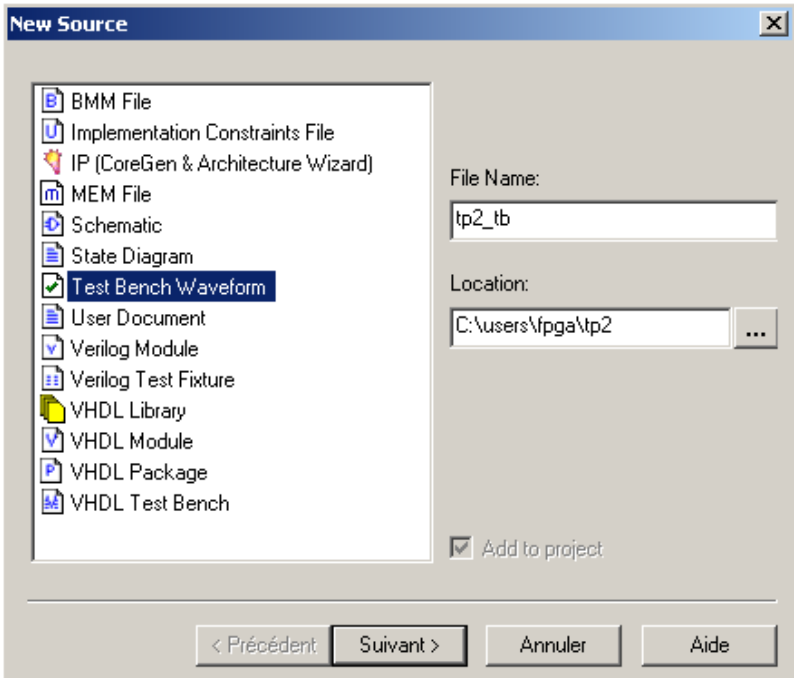
Corrigez toutes vos erreurs avant de passer à la suite.

7.2.5 Génération du fichier de stimuli VHDL

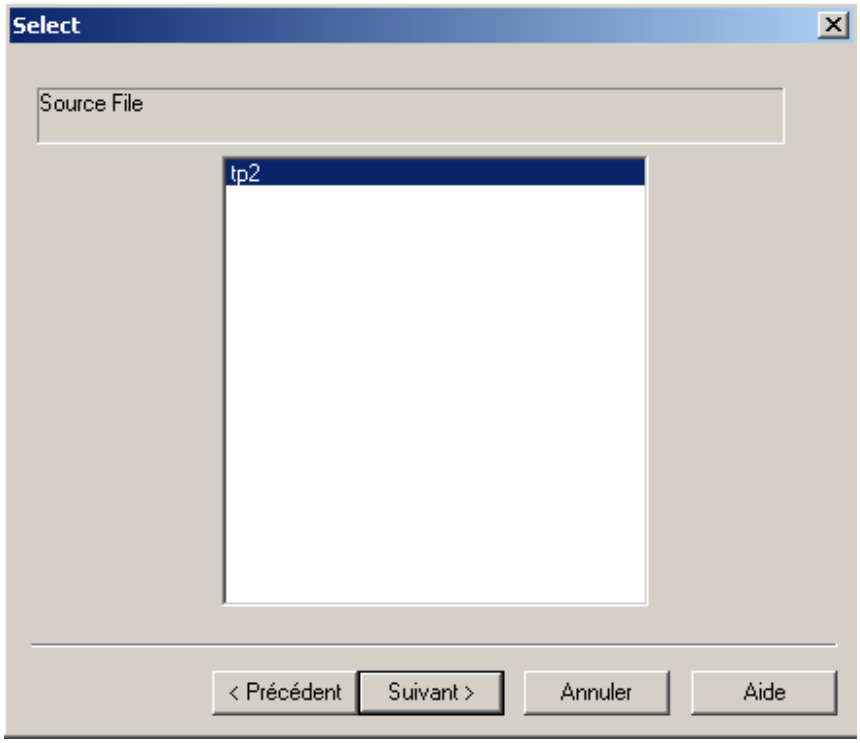
Nous avons maintenant un design prêt à être simulé et nous devons écrire un fichier en langage VHDL décrivant les signaux d'entrées. Ce fichier s'appelle un fichier de stimuli (un testbench en VHDL) qui contient des vecteurs de test. Il y a, parmi les outils Xilinx, un outil graphique pour définir les stimuli : Waveform Editor. Pour lancer cette application, cliquez sur le menu « Project » puis sur le sous-menu « New Source… ».



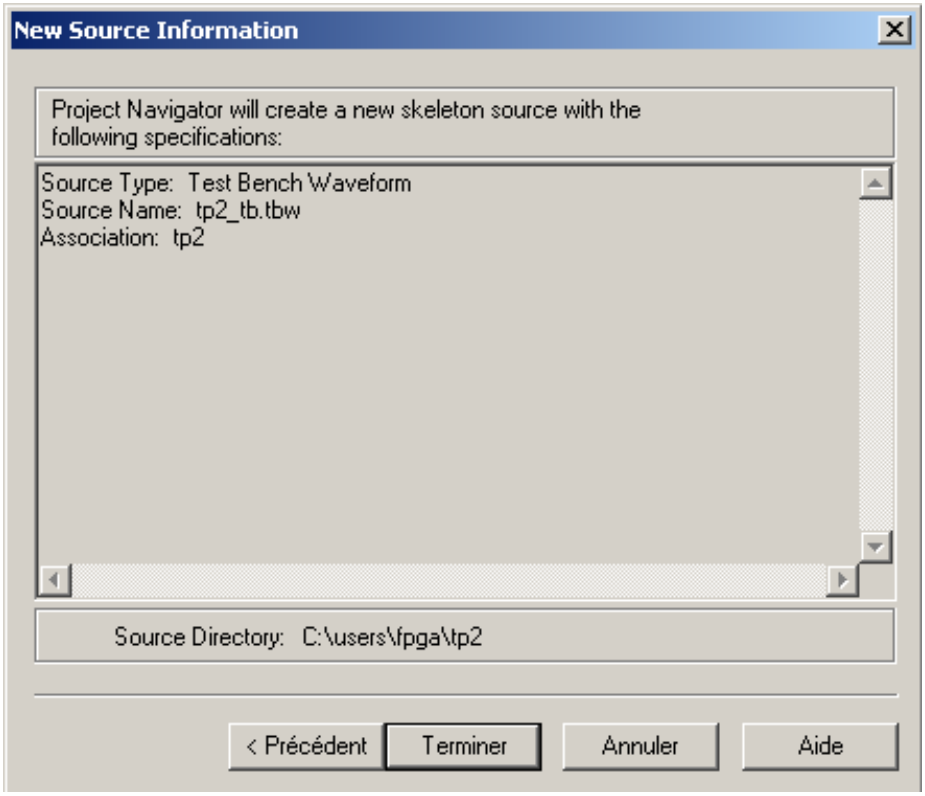
La fenêtre suivante apparaît alors à l'écran. Sélectionner le type « Testbench waveform », tapez tp2_tb comme nom de fichier pour notre testbench puis cliquez sur « Suivant » :



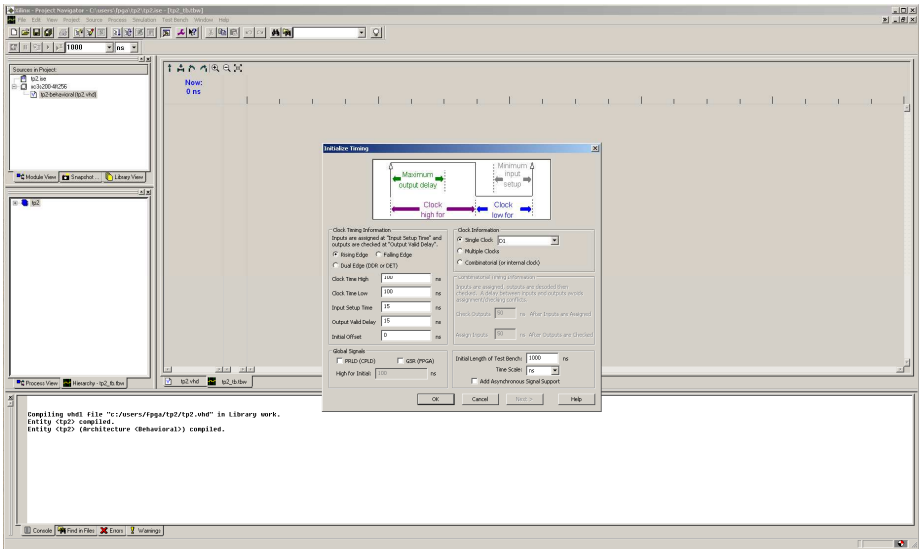
Cliquez sur suivant pour associer le testbench avec le design tp2 :



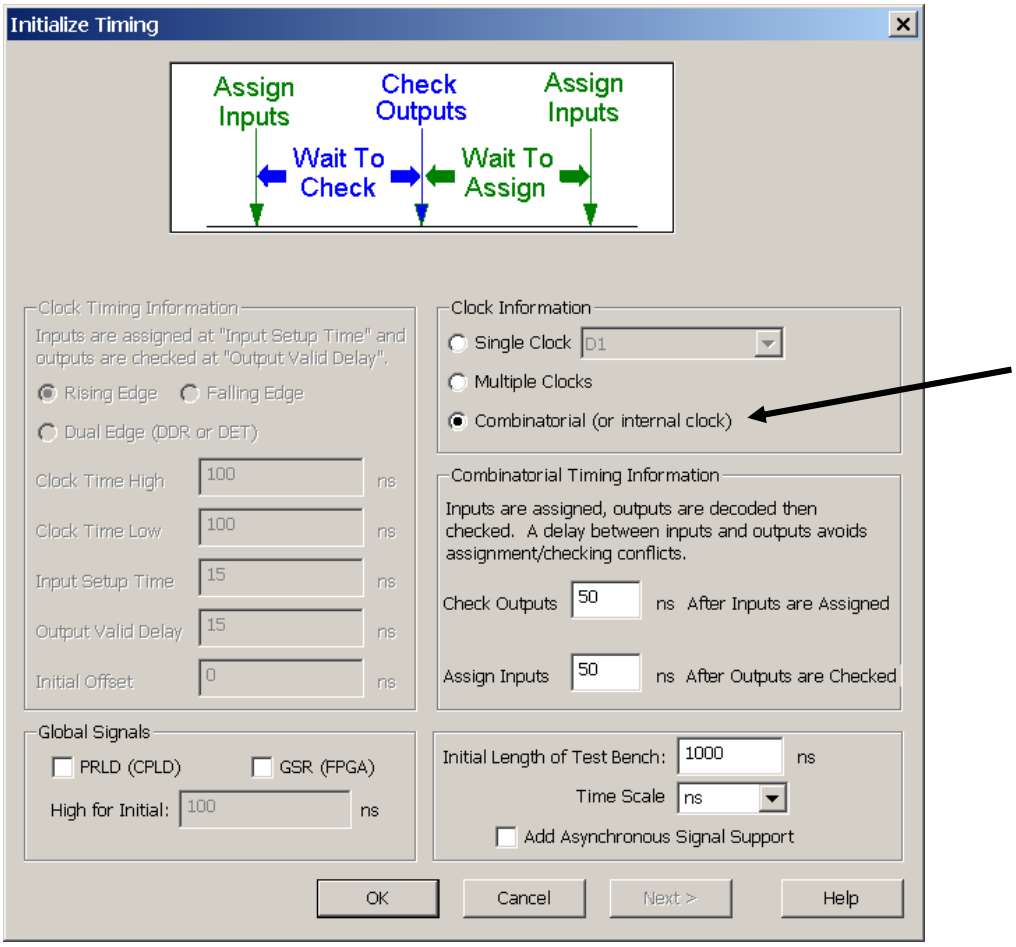
puis sur terminer pour lancer l'éditeur de stimuli :



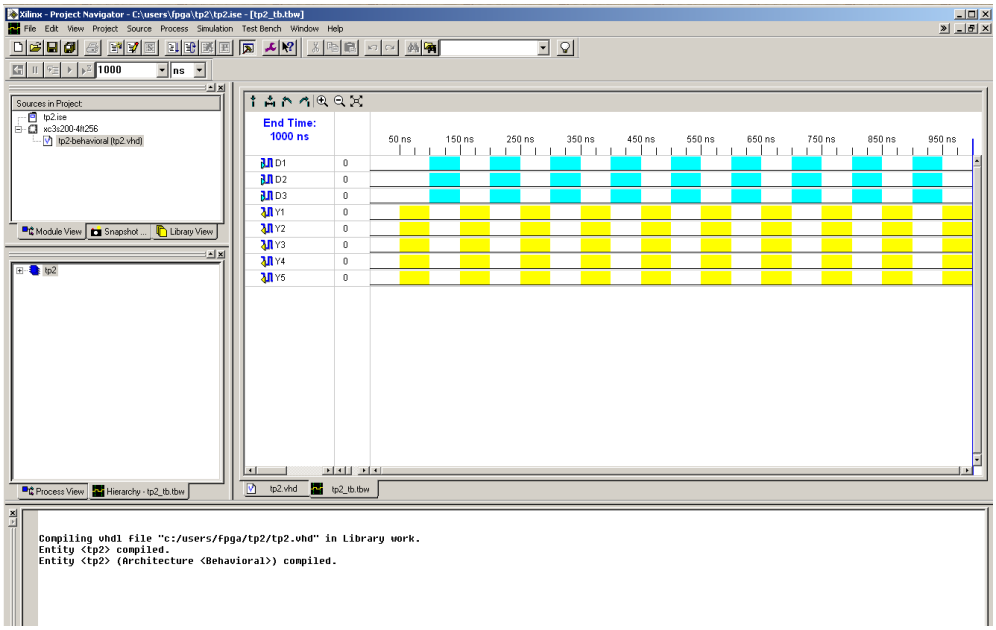
L’application démarre.



Notre design est purement combinatoire et ne possède donc pas d’horloge. Remplissez les différents champs comme sur la fenêtre suivante avant de cliquer sur « OK » :

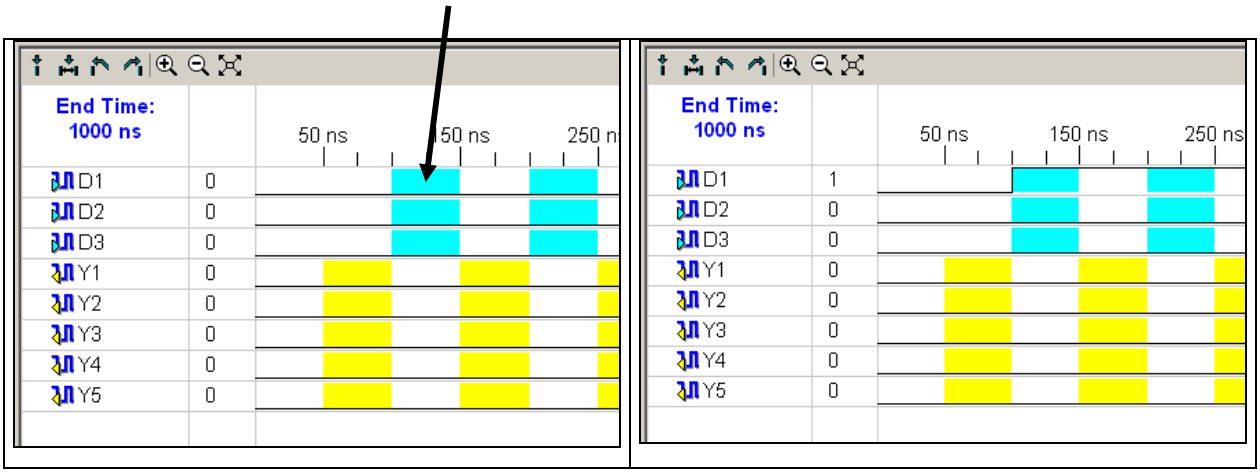


Un chronogramme apparaît dans la fenêtre de droite du navigateur de projet :

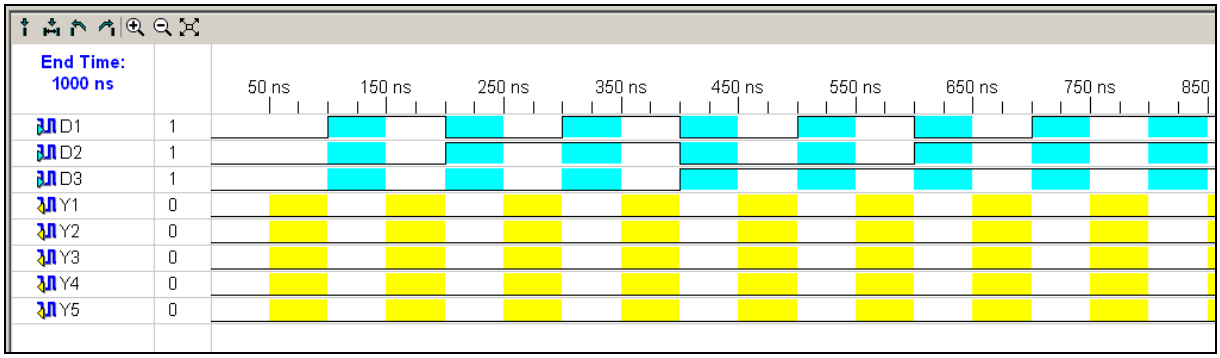


Nous pouvons maintenant spécifier les stimuli sous forme graphique.

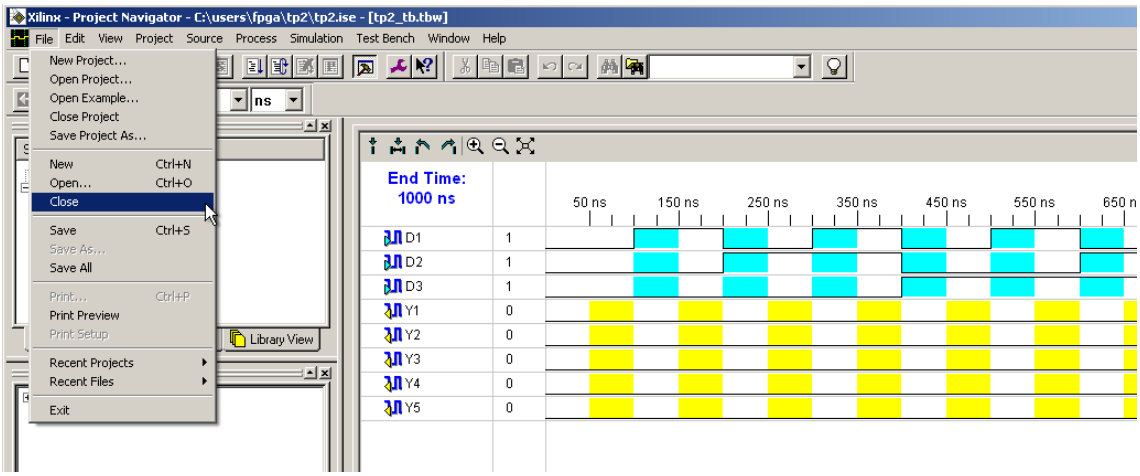
Vous devez voir vos trois entrées (D1, D2 et D3) et 5 sorties (Y1 à Y5). Si tel n'est pas le cas, vous avez sûrement oublié un signal dans l'entité. Fermez la fenêtre sans la sauvegarder puis, dans le navigateur, sélectionnez tp2, cliquez avec le bouton droit de la souris puis sur « Open ». La fenêtre de l'éditeur s'ouvre à nouveau. Faites les modifications nécessaires, sauvez votre fichier puis relancez Waveform Editor. Renouvelez ces opérations jusqu'à ce que vous ayez les bonnes entrées-sorties. Une fois la bonne fenêtre obtenue, cliquez sur la première zone bleue de D1 pour le faire passer à 1 :



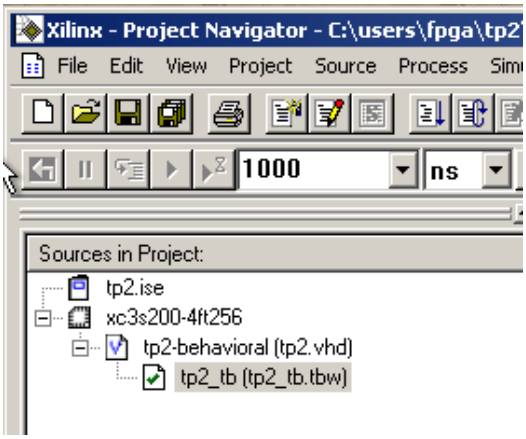
Cliquez sur les autres zones bleues jusqu'à obtenir les chronogrammes suivants :



Cliquez sur l'icône  pour sauver le testbench, puis sur le menu « File », « Close » pour quitter l'application :



Le testbench apparaît maintenant dans les sources du navigateur de projet. Vous pouvez à tout moment relancer Waveform Editor en double cliquant sur tp2_tb.tbw.

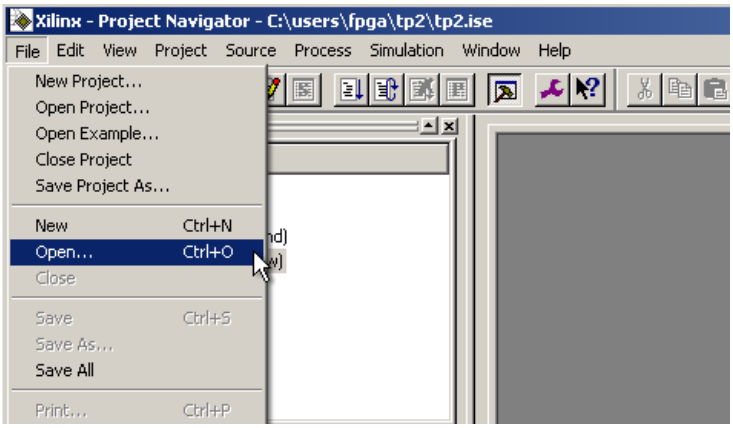


7.2.6 Simulation fonctionnelle

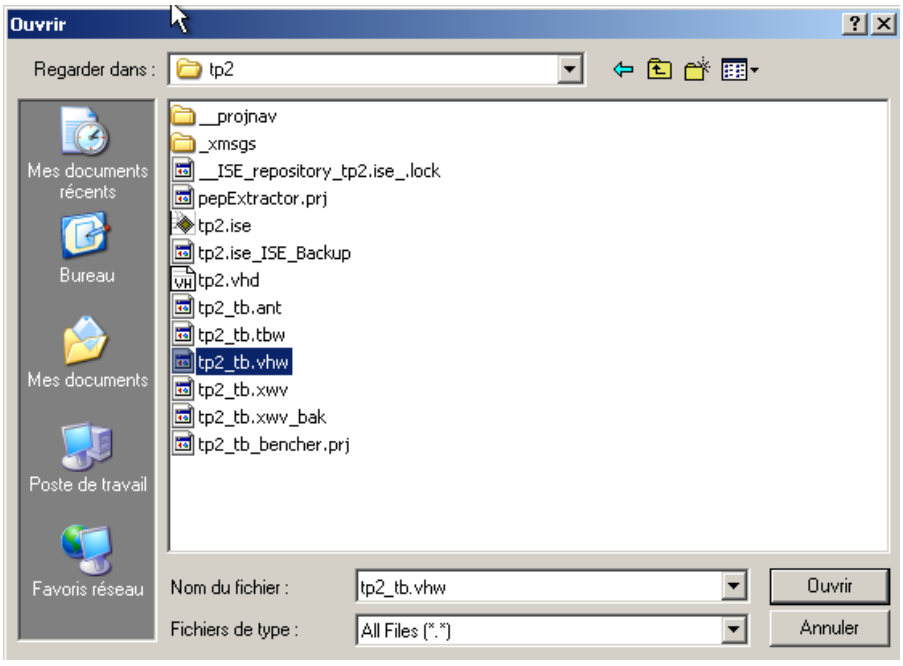
Nous sommes maintenant en possession de deux fichiers :

3. un fichier VHDL représentant le design à réaliser (tp2.vhd).
4. un fichier VHDL représentant les signaux que l'on souhaite appliquer sur ses entrées (tp2_tb.vhw). Ce fichier, généré par Waveform Editor, est la représentation en VHDL des stimuli créés précédemment sous forme graphique.

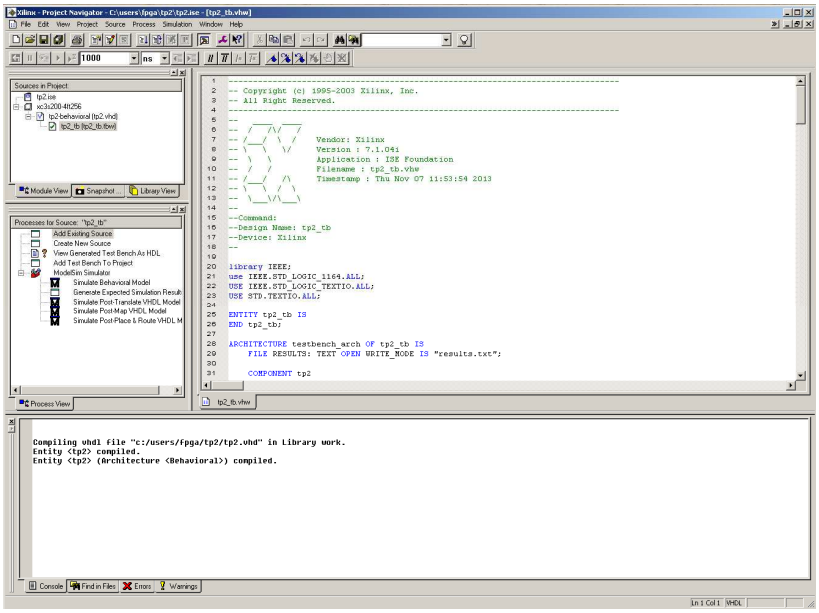
Vous pouvez visualiser ce fichier (**à faire seulement la première fois pour voir, sinon c'est inutile**) à l'aide du navigateur de projet en cliquant sur le menu File puis sur le sous-menu Open :



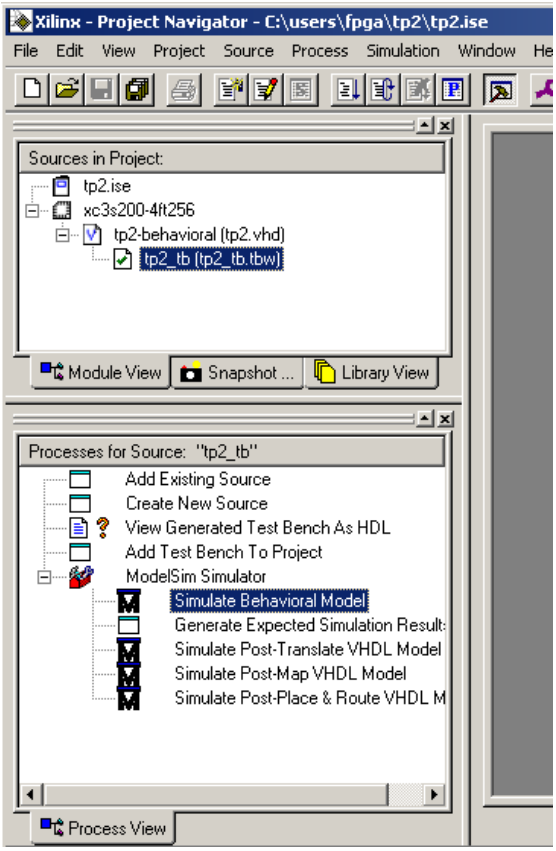
Sélectionnez le type de fichier « All Files » puis le fichier tp2_tb.vhw. Cliquez alors sur Ouvrir.



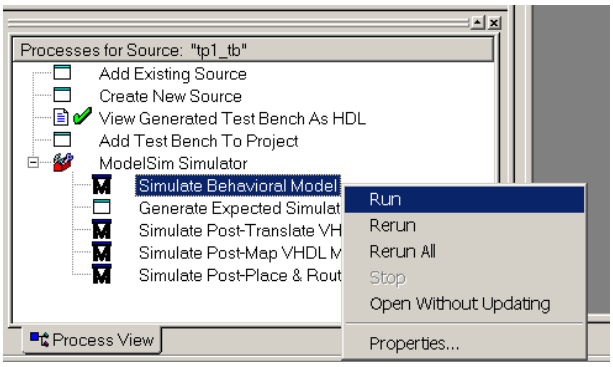
Le contenu du fichier apparaît dans la fenêtre d'édition du navigateur :



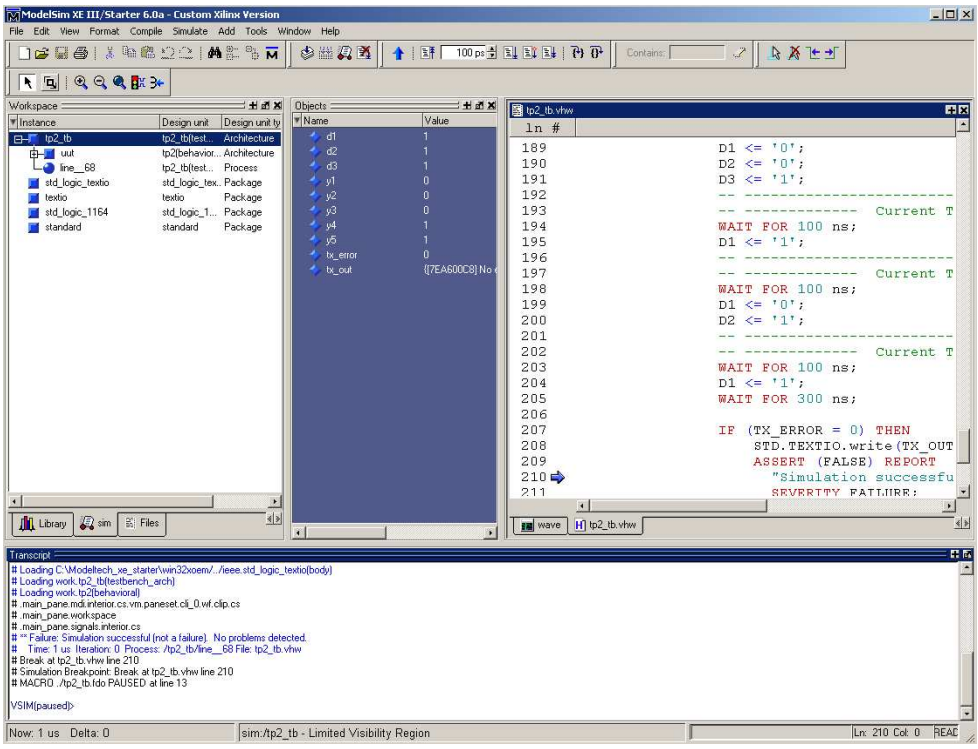
Fermez la fenêtre d'édition. Nous allons maintenant simuler le design à l'aide du simulateur VHDL ModelSim. Pour cela, sélectionnez le fichier tp2_tb.tbw dans la fenêtre « Sources In Project ». Les actions qui peuvent être effectuées avec ce fichier apparaissent dans la fenêtre inférieure « Processes for Source ».



Sélectionnez le process « Simulate Behavorial Model » puis cliquez avec le bouton droit de la souris sur le menu Run :



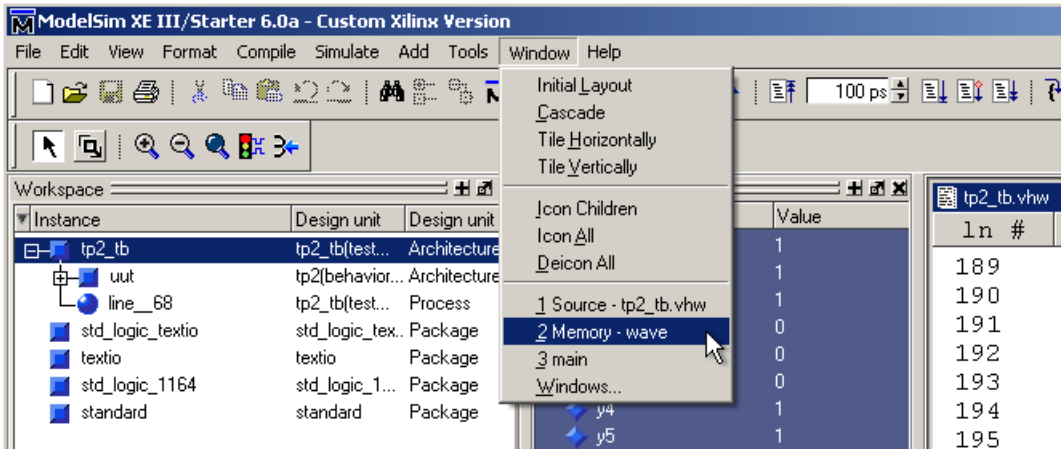
Le simulateur VHDL ModelSim démarre et la fenêtre suivante s'ouvre à l'écran. Si tel n'est pas le cas, c'est parce que ModelSim s'exécute sous le navigateur de projet. Il faut alors cliquer sur son icône dans la barre de tache de Windows pour le mettre au premier plan.



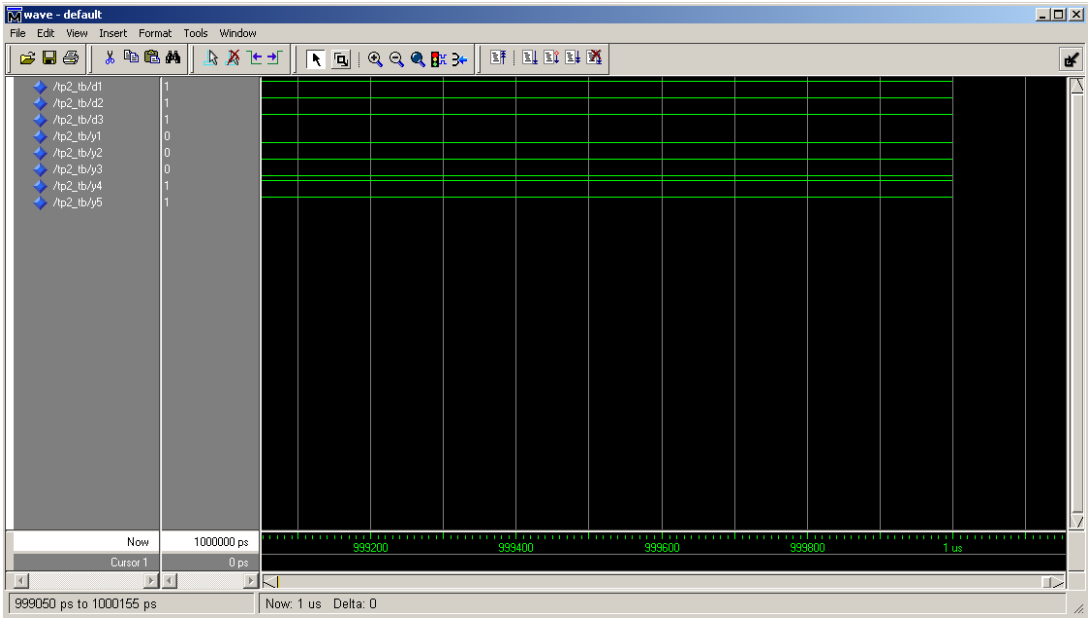
ModelSim va enchaîner automatiquement :

- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

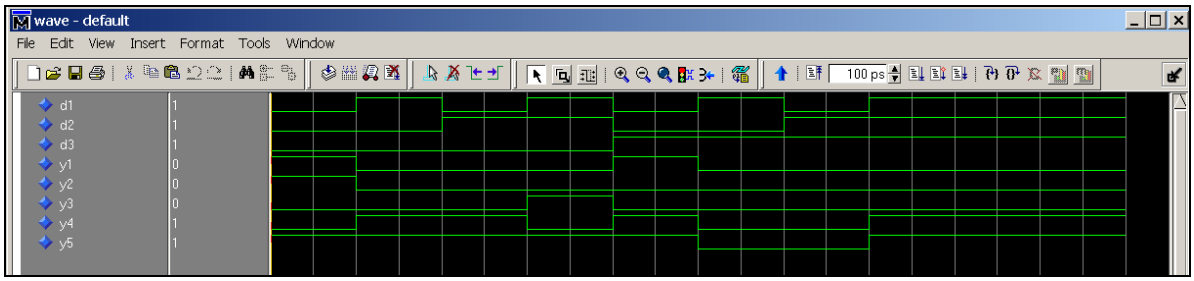
Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) en cliquant sur le menu Windows, Memory-wave :



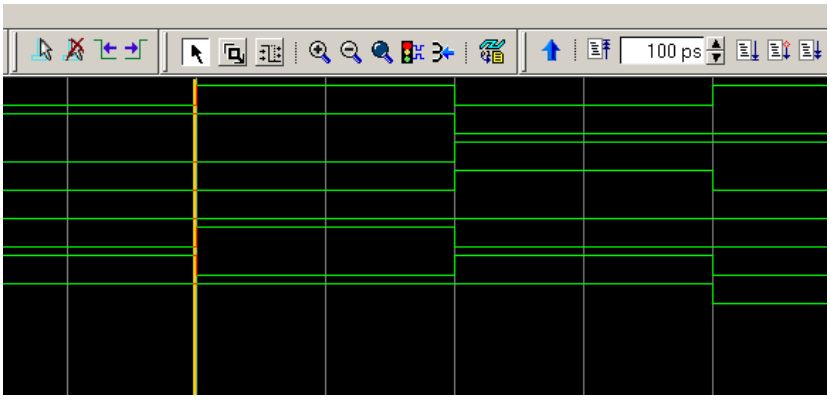
Nous allons détacher la fenêtre Wave de la fenêtre principale de Modelsim afin de mieux voir les chronogrammes. Pour cela, cliquez sur le bouton du milieu  (en haut à droite de la fenêtre wave) puis mettez la fenêtre wave plein écran.



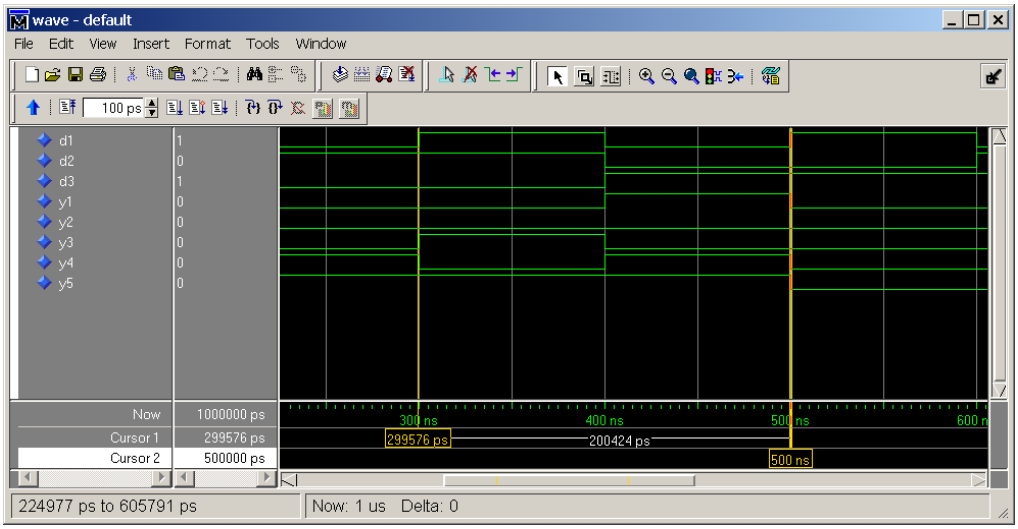
Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation (de 0 à 1000 ns). Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes.



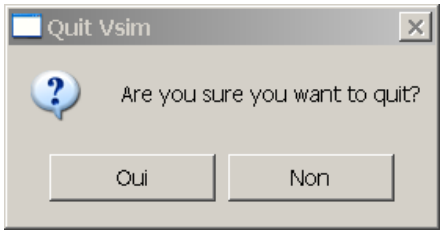
Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Cliquez n'importe où sur le chronogramme. Un curseur jaune apparaît :



Cliquez sur le bouton « Insert Cursor »  de la barre d'outils. Un deuxième curseur apparaît. Vous voyez en bas de la fenêtre « Wave » le temps correspondant à la position de chaque curseur ainsi que l'écart qui les sépare.

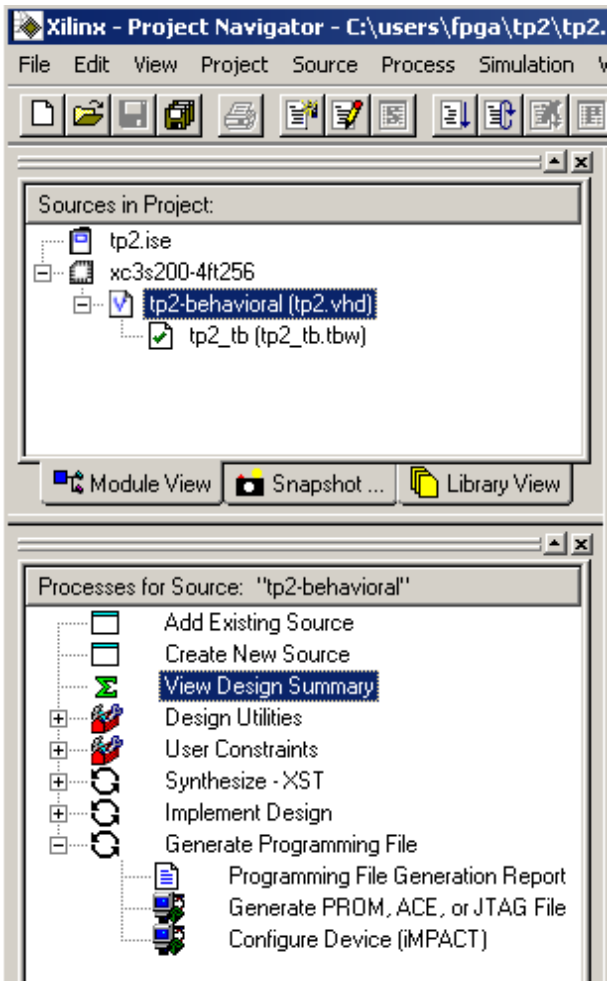


Quand la vérification de votre design est terminée, faites apparaître la fenêtre de ModelSim, puis cliquez sur le menu « File » dans la fenêtre principale, puis sur « Quit ». Quand le message suivant apparaît à l'écran, cliquez sur Oui.

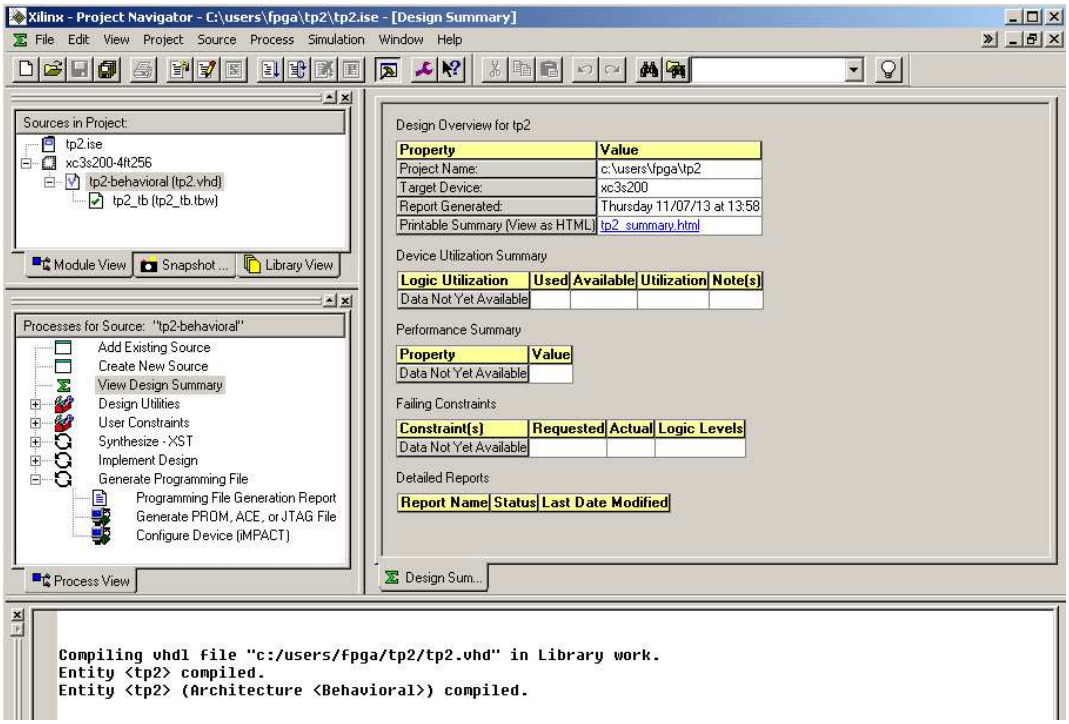


7.2.7 Synthèse

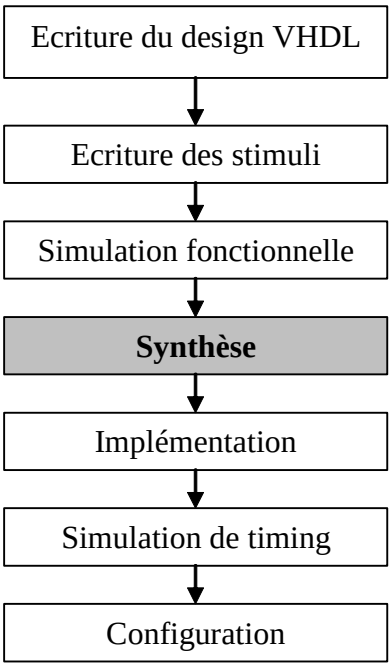
Nous avons fini la première phase de création et de vérification du design. Vous pouvez voir à tout moment le résumé des différentes étapes du design en sélectionnant tp2, puis en double-cliquant sur « View Design Summary » :



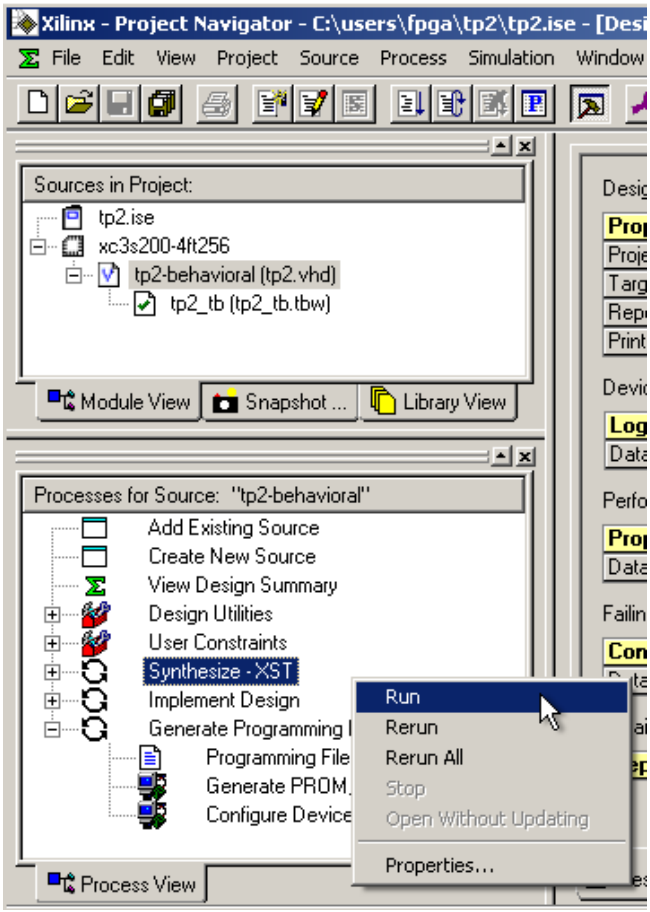
La page de résumé s'affiche à droite du navigateur de projet :



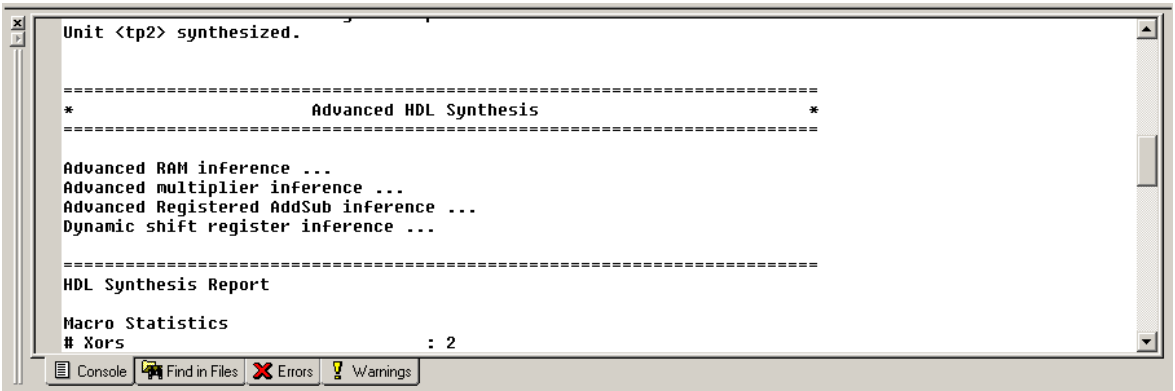
Elle contient pour l'instant peu d'informations, mais elle va s'enrichir au fur et à mesure de l'avancement du projet. La description VHDL tp2.vhd est assez générale et ne prend en compte aucun circuit cible en particulier. Le rôle du synthétiseur XST est de comprendre et d'interpréter cette description générale afin de générer un fichier NGC compréhensible par les outils d'implémentation, c'est-à-dire une netlist NGC composée de primitives simples. Cette netlist est spécifique à notre FPGA.



Sélectionnez le design tp2.vhd dans la fenêtre « Sources » puis « Synthesize » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



La synthèse démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant son déroulement apparaît :

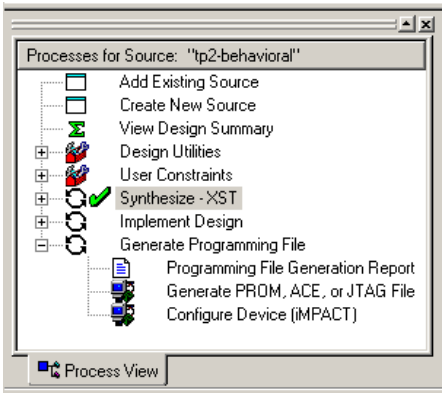


Lorsque la synthèse est finie, vous devez voir une estimation des timings du design, ce qui indique que la synthèse s'est bien terminée.

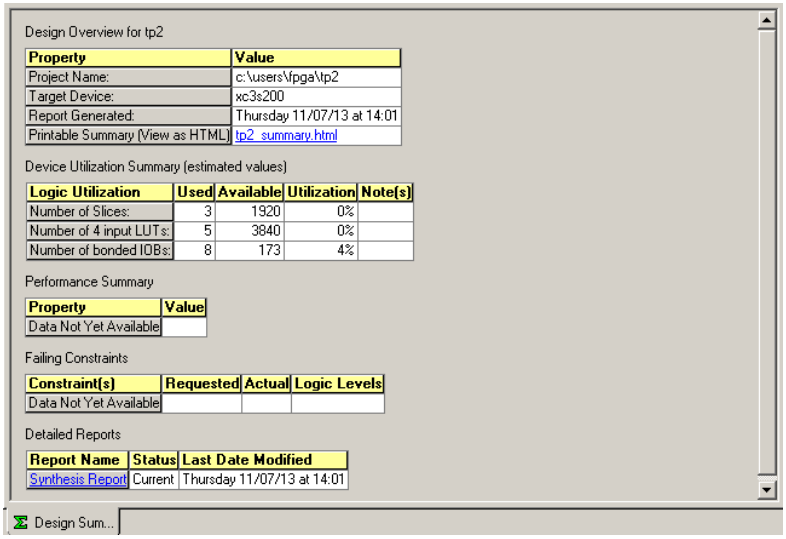
```
-------------------
No clock signals found in this design

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: No path found
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: No path found
   Maximum combinational path delay: 9.077ns

=========================================================================
```

Console | Find in Files | Errors | Warnings

Une marque verte apparaît dans la fenêtre Processes. Elle indique que la synthèse s'est bien déroulée. Un point d'exclamation jaune signalerait un message d'avertissement (un Warning) et une croix rouge indiquerait une erreur. Tous ces messages apparaissent dans le rapport de synthèse.



Vous ne pouvez pas visualiser le fichier NGC issu de la synthèse car il s'agit d'un format de netlist binaire qui ne peut donc être visualisé avec un éditeur ASCII. La fenêtre de résumé a évoluée et reflète maintenant les informations de synthèse :



Design Overview for tp2

Property	Value
Project Name:	c:\users\fpga\tp2
Target Device:	xc3s200
Report Generated:	Thursday 11/07/13 at 14:01
Printable Summary (View as HTML)	tp2_summary.html

Device Utilization Summary (estimated values)

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slices:	3	1920	0%	
Number of 4 input LUTs:	5	3840	0%	
Number of bonded IOBs:	8	173	4%	

Performance Summary

Property	Value
Data Not Yet Available	

Failing Constraints

Constraint(s)	Requested	Actual	Logic Levels
Data Not Yet Available			

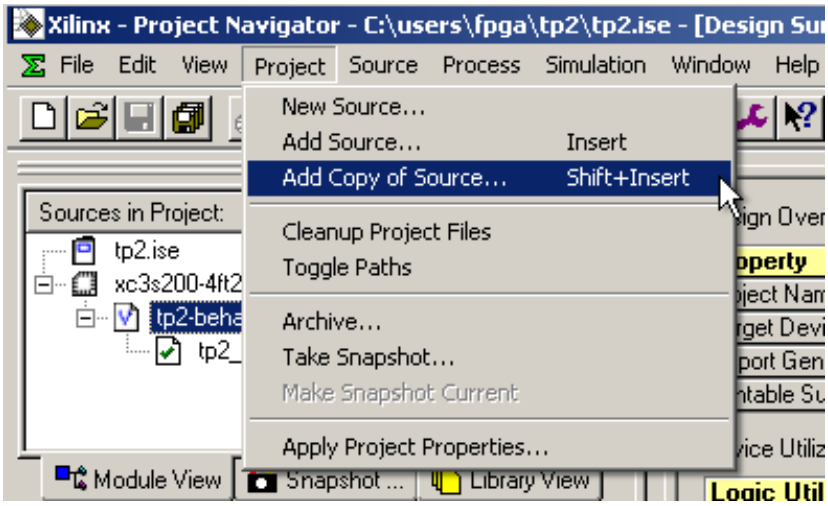
Detailed Reports

Report Name	Status	Last Date Modified
Synthesis Report	Current	Thursday 11/07/13 at 14:01

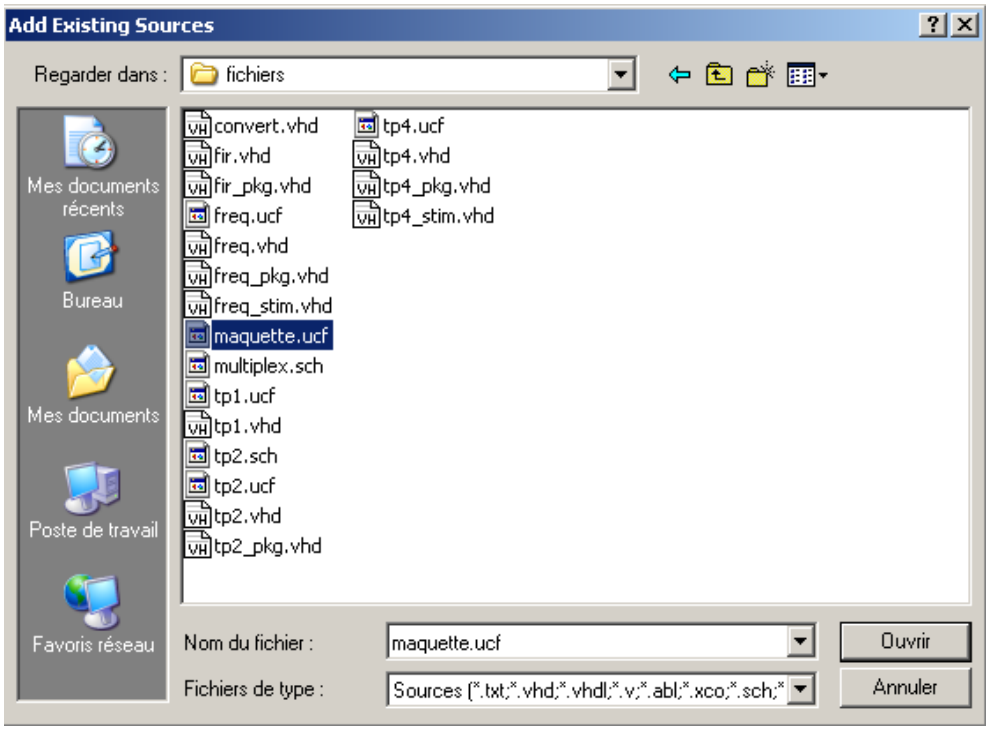
Design Sum...

7.2.8 Implémentation

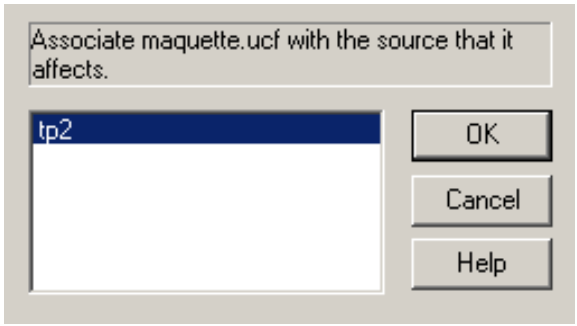
Nous pouvons maintenant travailler sur le circuit FPGA lui-même. C'est le rôle des outils d'implémentation. Nous allons commencer par affecter les broches D1 à D3 et Y1 à Y5 aux broches du FPGA selon le câblage de la maquette. Vous utiliserez pour cela un fichier de contraintes utilisateur déjà écrit qui se nomme maquette.ucf (User Constraint File). Pour l'ajouter au projet tp2, sélectionnez le menu Project, Add Copy of Source :



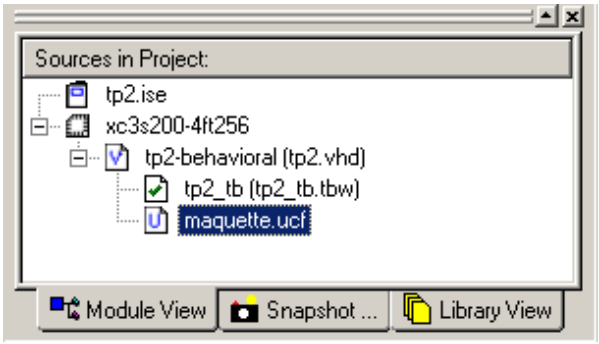
Dans la fenêtre qui s'ouvre, sélectionnez le répertoire c:\users\fpga\fichiers, cliquez sur le fichier maquette.ucf puis sur « Ouvrir » :



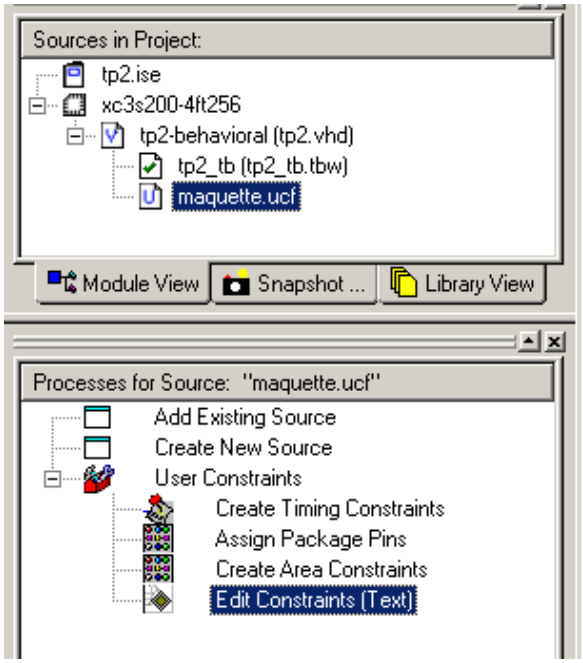
Cliquez sur le bouton « OK » dans la petite fenêtre qui s'ouvre alors :



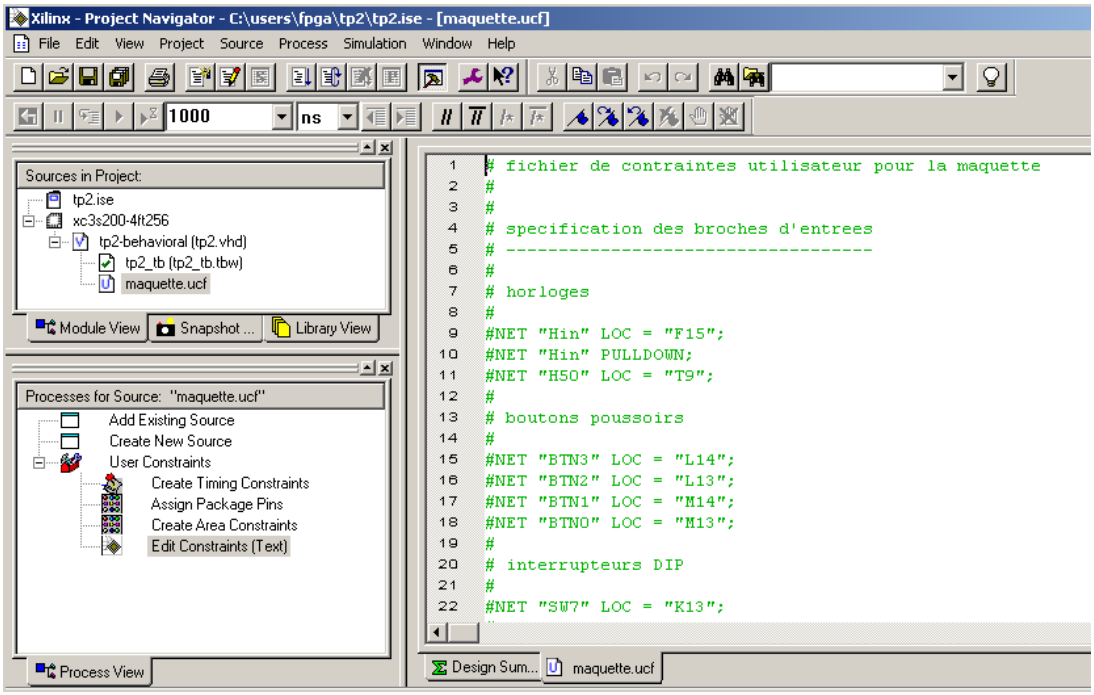
Le fichier est maintenant inclus dans le projet :



Pour voir le contenu de ce fichier, il suffit de le sélectionner dans la fenêtre « Sources », puis de double-cliquer dans la fenêtre Processes sur « Edit Constraints (Text) » :



Le contenu du fichier apparaît alors dans l'éditeur :



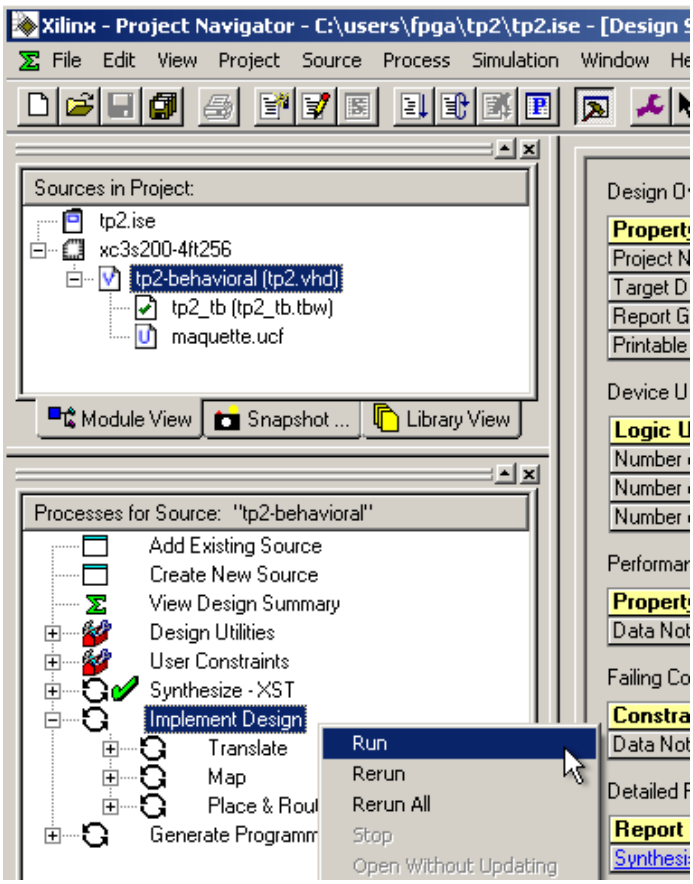
Toutes les lignes de ce fichier sont commentées avec un #. Nous allons affecter D1, D2 et D3 sur les dip switch SW0, SW1 et SW2. Pour cela, il suffit de décommenter les 3 lignes et de changer les noms des signaux :

```
#NET "BTN1" LOC = "M14";
#NET "BTN0" LOC = "M13";
#
# interrupteurs DIP
#
#NET "SW7" LOC = "K13";
#NET "SW6" LOC = "K14";
#NET "SW5" LOC = "J13";
#NET "SW4" LOC = "J14";
#NET "SW3" LOC = "H13";
NET "D3" LOC = "H14";
NET "D2" LOC = "G12";
NET "D1" LOC = "F12";
#
# specification des broches de sorties
# ------------------------------------
#
```

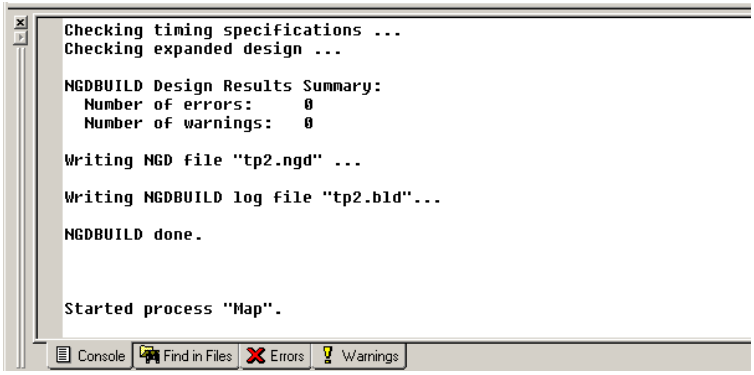
Nous allons ensuite affecter Y1 à Y5 sur les leds led0 à led4. Pour cela, il suffit de décommenter les 5 lignes et de changer les noms des signaux :

```
27  NET "D3" LOC = "H14";
28  NET "D2" LOC = "G12";
29  NET "D1" LOC = "F12";
30  #
31  # specification des broches de sorties
32  # -----------------------------------
33  #
34  # leds
35  #
36  #NET "Led7" LOC = "P11";
37  #NET "Led6" LOC = "P12";
38  #NET "Led5" LOC = "N12";
39  NET "Y5" LOC = "P13";
40  NET "Y4" LOC = "N14";
41  NET "Y3" LOC = "L12";
42  NET "Y2" LOC = "P14";
43  NET "Y1" LOC = "K12";
44  #
45  # afficheur 7 segments
```

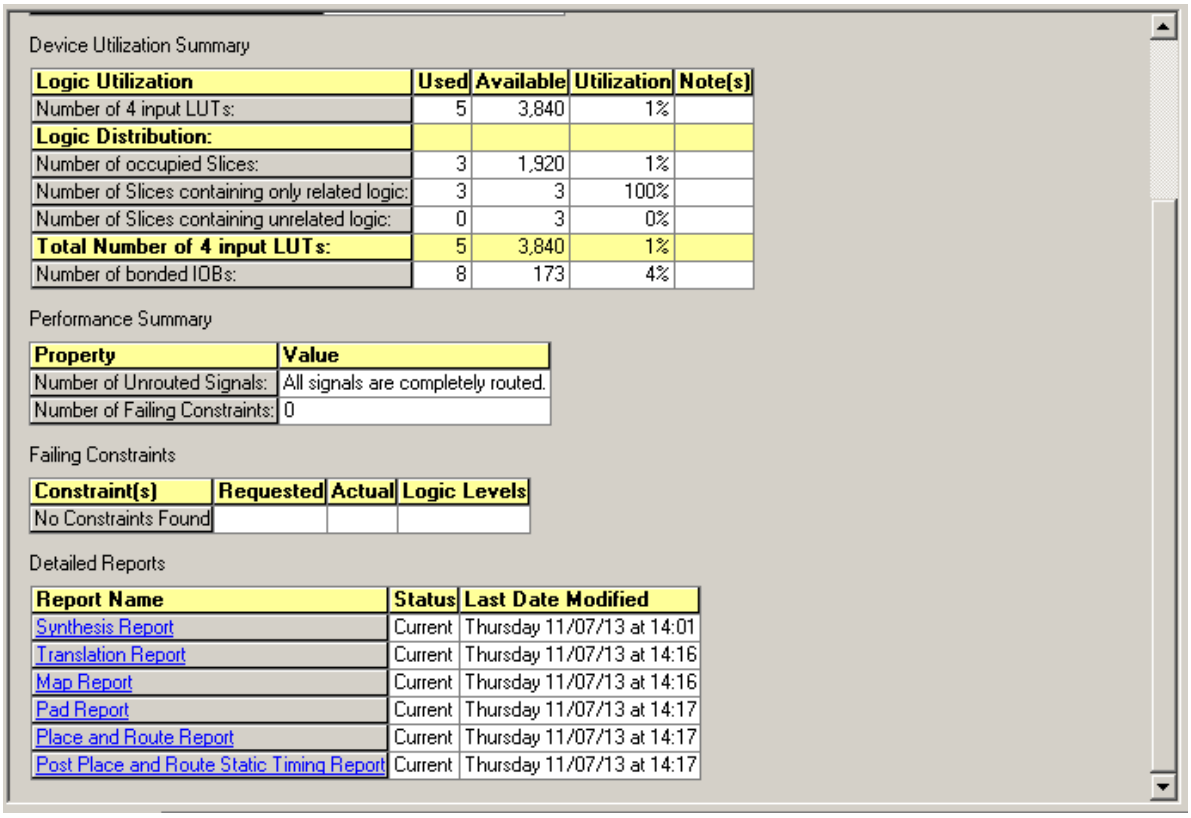
Vous pouvez maintenant sauver le fichier puis fermer l'éditeur. Nous pouvons lancer l'implantation du FPGA (implementation en anglais). Pour cela, sélectionnez le design tp2.vhd dans la fenêtre « Sources » puis « Implement Design » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



L'implémentation démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



Lorsque l'implémentation est finie, vous devez voir le message : PAR done ! qui indique que l'implémentation s'est bien terminée. Des points d'exclamation jaunes peuvent apparaître dans la fenêtre Processes. Ils indiquent que des messages d'avertissement (Warnings) ont été émis pendant l'implémentation. Vous pouvez voir ces Warnings en faisant défiler le rapport dans la Console à l'aide de la barre de défilement de droite. Il ne doit pas y en avoir dans notre exemple. Un certain nombre de rapports concernant les différentes étapes de l'implémentation sont maintenant disponibles dans la fenêtre Design Summary :



Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of 4 input LUTs:	5	3,840	1%	
Logic Distribution:				
Number of occupied Slices:	3	1,920	1%	
Number of Slices containing only related logic:	3	3	100%	
Number of Slices containing unrelated logic:	0	3	0%	
Total Number of 4 input LUTs:	5	3,840	1%	
Number of bonded IOBs:	8	173	4%	

Performance Summary

Property	Value
Number of Unrouted Signals:	All signals are completely routed.
Number of Failing Constraints:	0

Failing Constraints

Constraint(s)	Requested	Actual	Logic Levels
No Constraints Found			

Detailed Reports

Report Name	Status	Last Date Modified
Synthesis Report	Current	Thursday 11/07/13 at 14:01
Translation Report	Current	Thursday 11/07/13 at 14:16
Map Report	Current	Thursday 11/07/13 at 14:16
Pad Report	Current	Thursday 11/07/13 at 14:17
Place and Route Report	Current	Thursday 11/07/13 at 14:17
Post Place and Route Static Timing Report	Current	Thursday 11/07/13 at 14:17

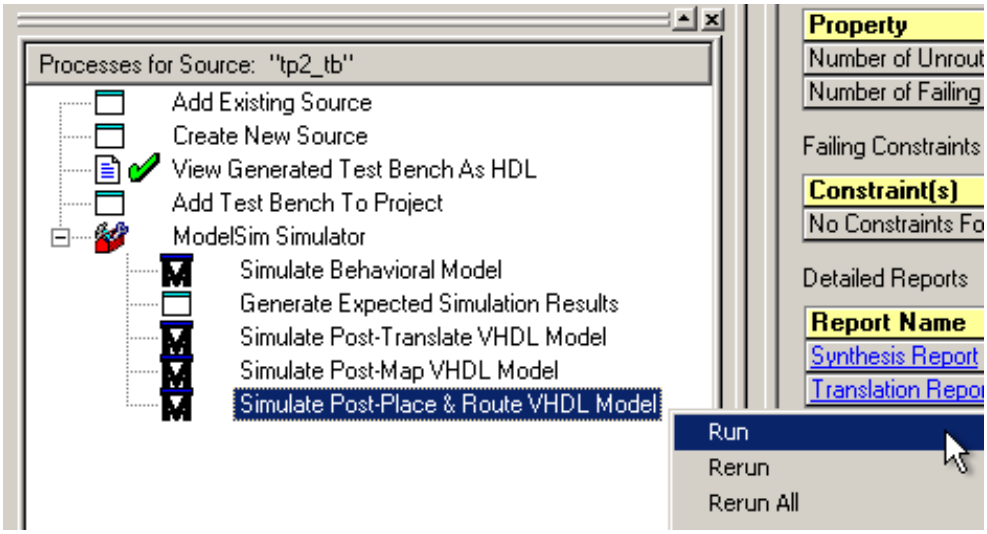
Le tableau suivant vous indique le rôle des différentes étapes de l'implémentation ainsi que les rapports associés :

Etape	Rapport	Signification
Translation	Translation report	Création d'un fichier de design unique
Mapping	Map report	Découpage du design en primitives
Placement-routage	Place & Route report	Placement et routage des primitives
	Post Place & Route static timing report	Respect des contraintes temporelles et fréquence max de fonctionnement
	Pad report	Assignation des broches du FPGA

7.2.9 Simulation de timing

Les outils de placement-routage peuvent fournir un nouveau modèle VHDL (tp2_timesim.vhd) appelé **modèle VITAL** qui correspond au modèle de simulation réel du circuit ainsi qu'un fichier contenant tous les timings du FPGA (tp2_timesim.sdf) appelé fichier **SDF**. A l'aide de ces deux fichiers et du fichier de stimuli tp2_tb.timesim_vhw généré en même temps, nous allons pouvoir vérifier le fonctionnement réel de notre design. C'est la simulation de timing (ou simulation Post-Place&Route ou encore simulation Post-layout ou simulation Post-Implementation).

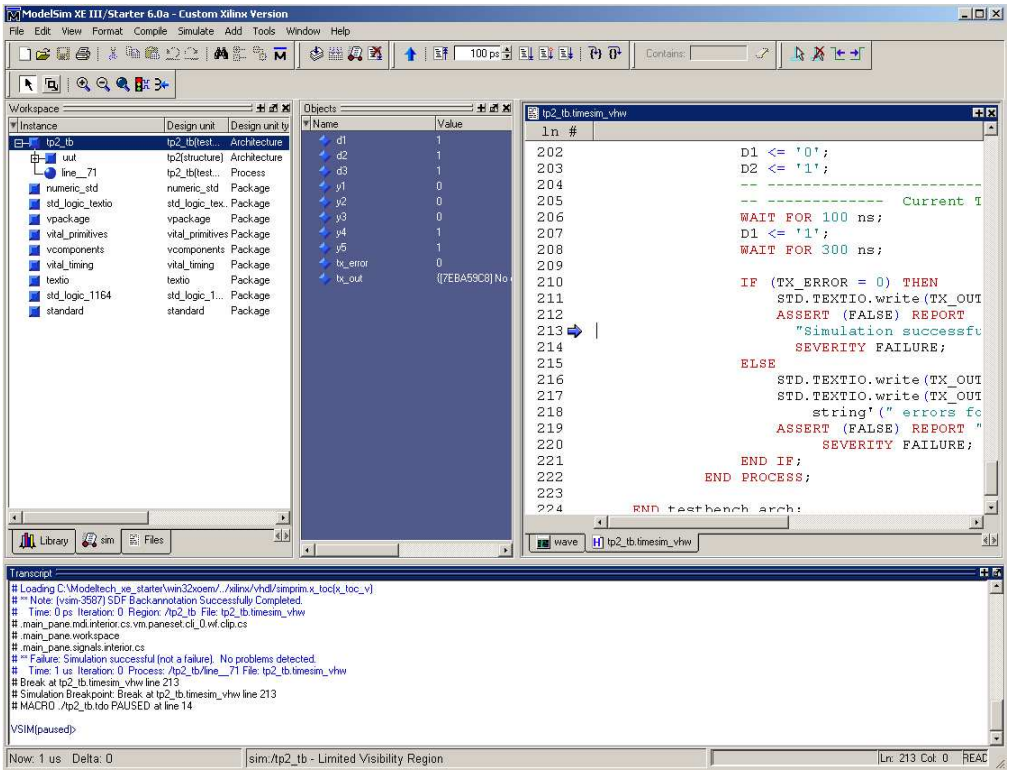
Pour lancer cette simulation, sélectionnez le fichier tp2_tb.tbw dans la fenêtre « Sources In Project ». Sélectionnez le processus « Simulate Post-Place&Route VHDL Model » puis cliquez avec le bouton droit de la souris sur le menu Run :



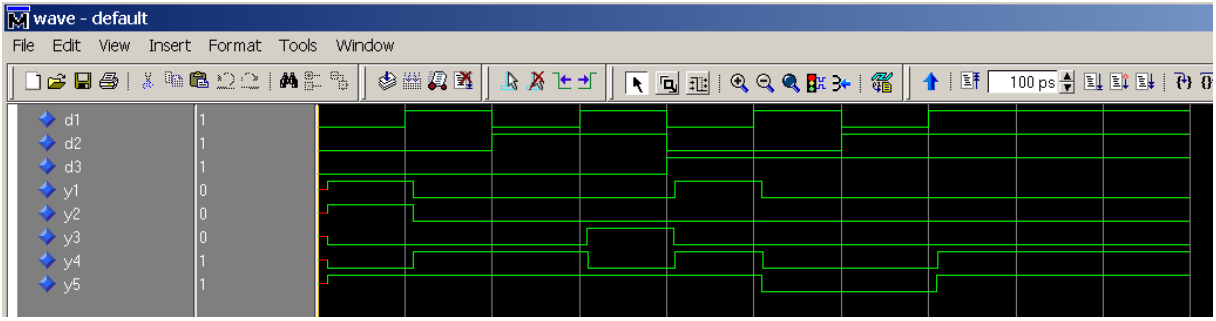
Le simulateur VHDL ModelSim démarre. Il va enchaîner automatiquement :

- la compilation du design et des stimuli,
- le lancement du simulateur,
- la simulation jusqu'à l'arrêt automatique du testbench.

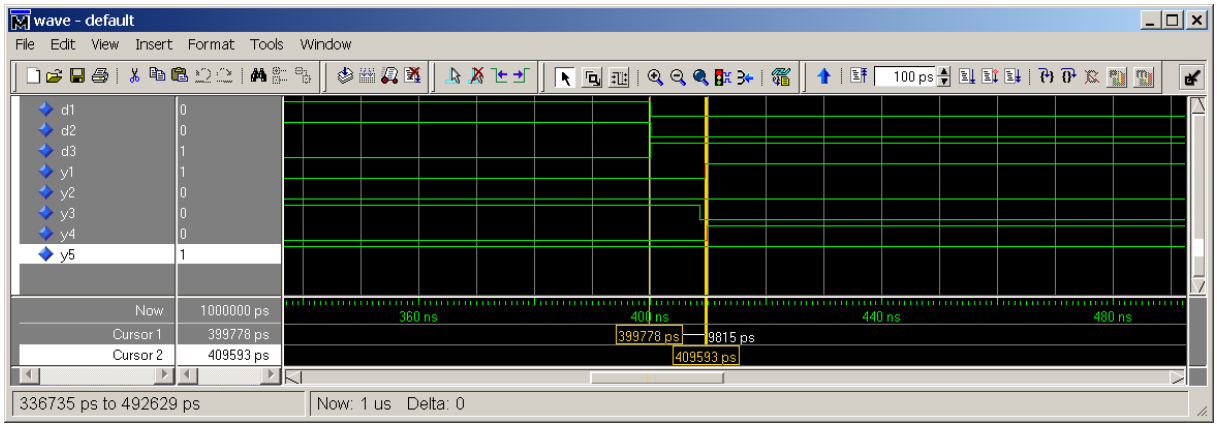
A la fin, la fenêtre Modelsim est la suivante :



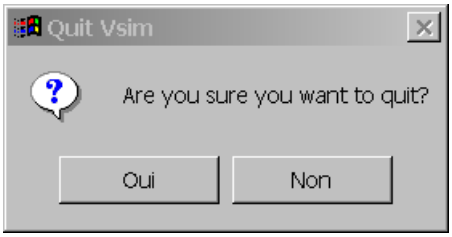
Sélectionnez la fenêtre Wave (là où se trouvent les chronogrammes) puis détachez-la de la fenêtre ModelSim. Cliquez sur le bouton « Zoom Full »  de la barre d'outils de cette fenêtre pour visualiser du début à la fin de la simulation. Vous pouvez maintenant vérifier le fonctionnement de votre montage à l'aide des chronogrammes.



Faîtes un zoom sur le chronogramme en cliquant en haut et à gauche de la zone à agrandir avec le bouton du milieu de la souris. Maintenez cliqué et déplacez la souris en bas et à droite de la zone (un rectangle bleu apparaît). Relâchez le bouton du milieu. Le zoom s'exécute. Sur l'exemple de chronogramme suivant, on voit clairement apparaître le décalage entre les entrées et les sorties. Utilisez les curseurs temporels pour mesurer ce délai.

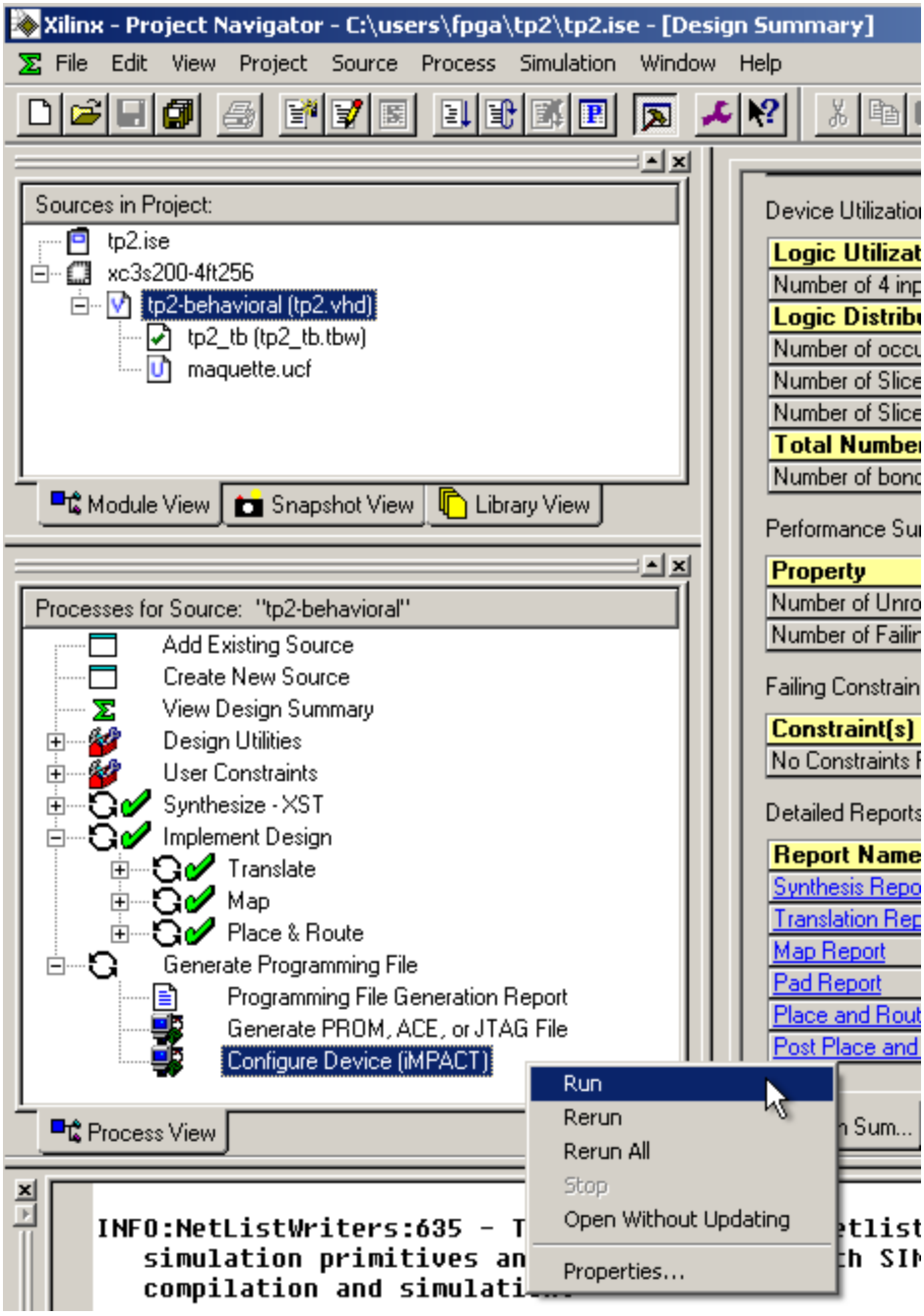


Quand la vérification est terminée, cliquez sur le menu « File » dans la fenêtre ModelSim, puis sur « Quit». Quand le message ci-dessous apparaît à l'écran, cliquez sur Oui.

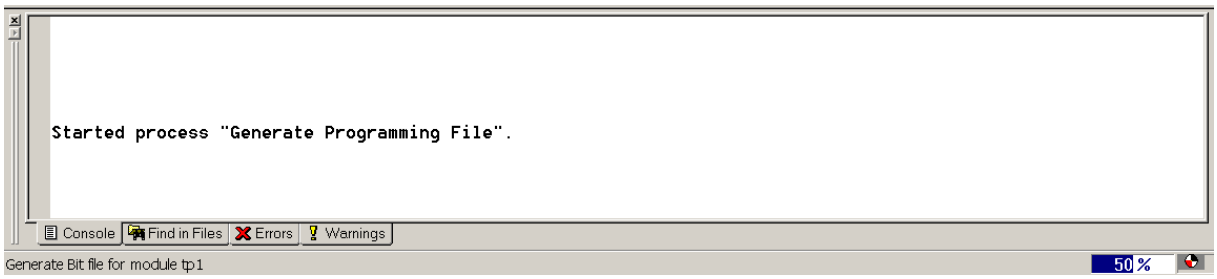


7.2.10 Configuration de la maquette

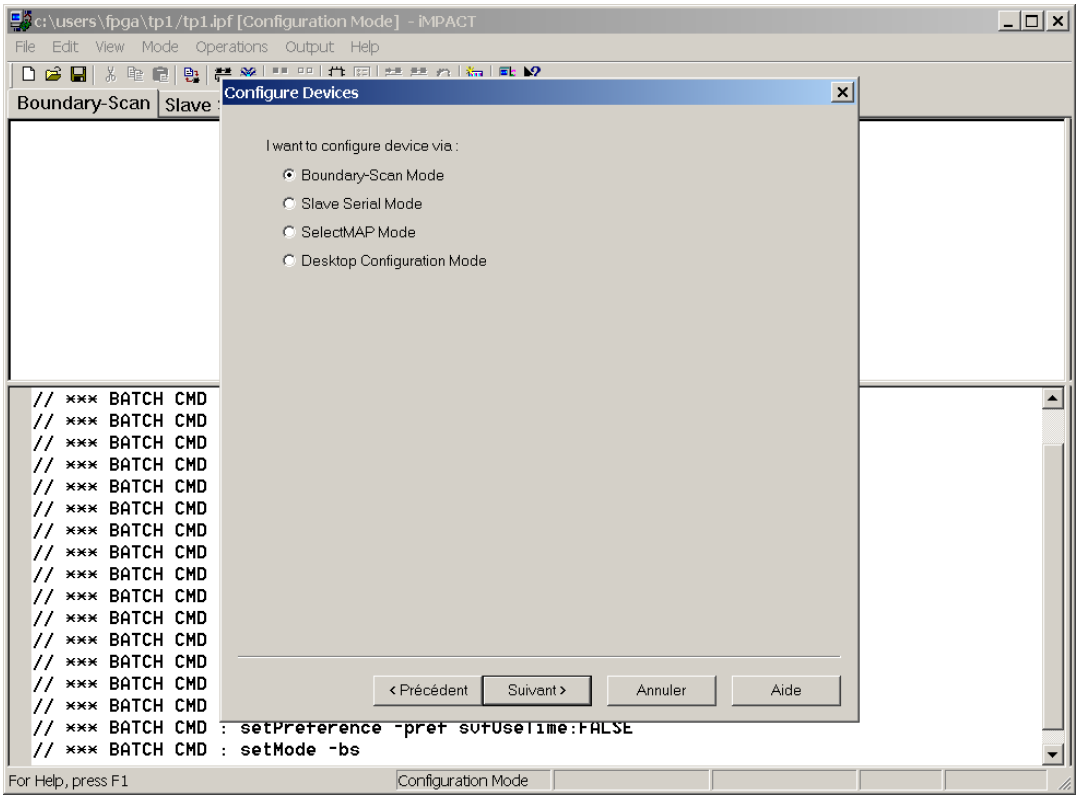
A ce point du TP, le design est entièrement vérifié. Nous pouvons maintenant le télécharger dans le FPGA. Sélectionnez le design tp2.vhd dans la fenêtre « Sources » puis « Configure Device » dans la fenêtre « Processes ». Cliquez avec le bouton droit de la souris puis cliquez sur « Run » :



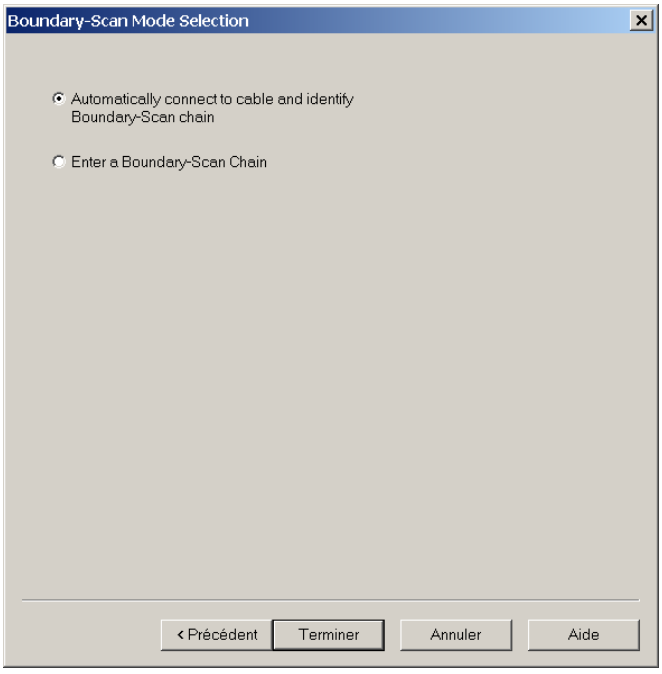
Le processus démarre. Dans la fenêtre Console du navigateur de projet, le rapport concernant les différentes opérations apparaît :



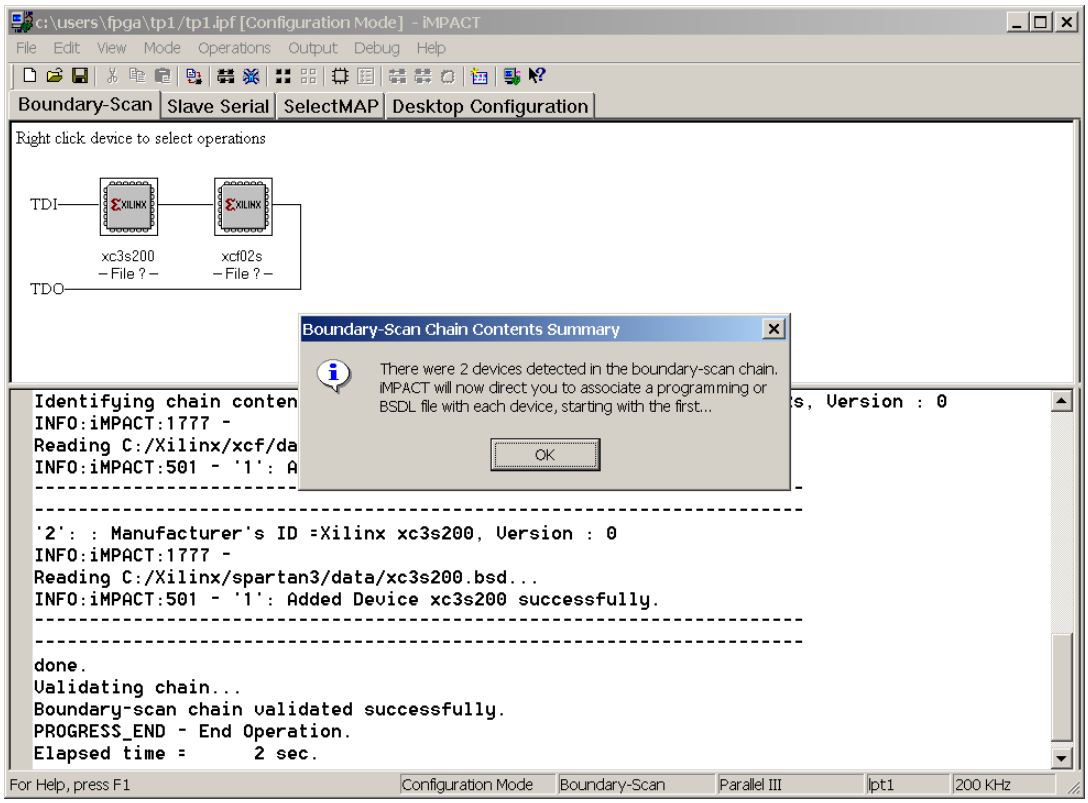
Puis la fenêtre de l'application Impact apparaît à l'écran. Nous allons configurer le FPGA en utilisant le mode « Boundary-scan » (JTAG). Cliquez sur le bouton « Suivant » :



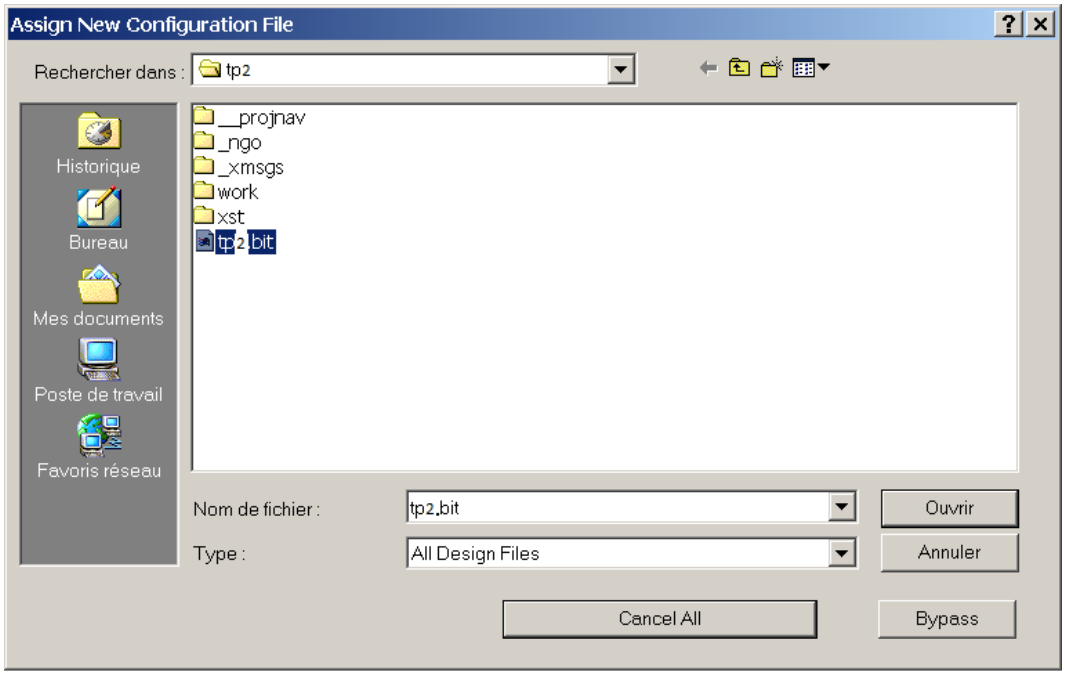
Laissons Impact découvrir automatiquement les circuits connectés sur la chaîne JTAG. Cliquez sur le bouton « Terminer » :



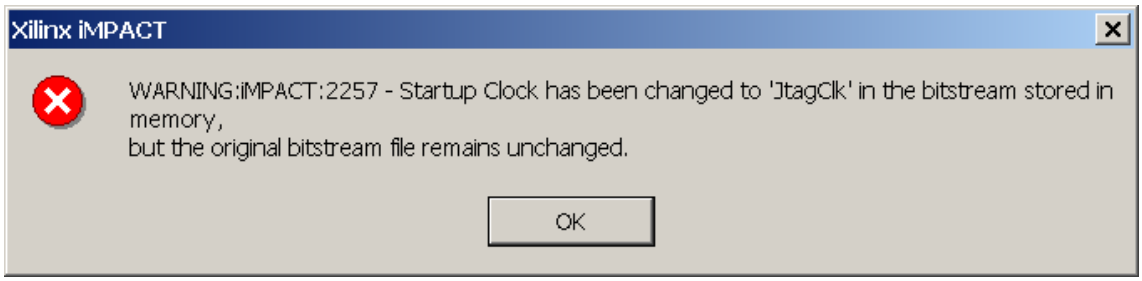
Impact a trouvé 2 circuits dans la chaîne JTAG, le FPGA et une mémoire Flash série que nous n'allons pas utiliser maintenant. Cliquez sur le bouton « OK » :



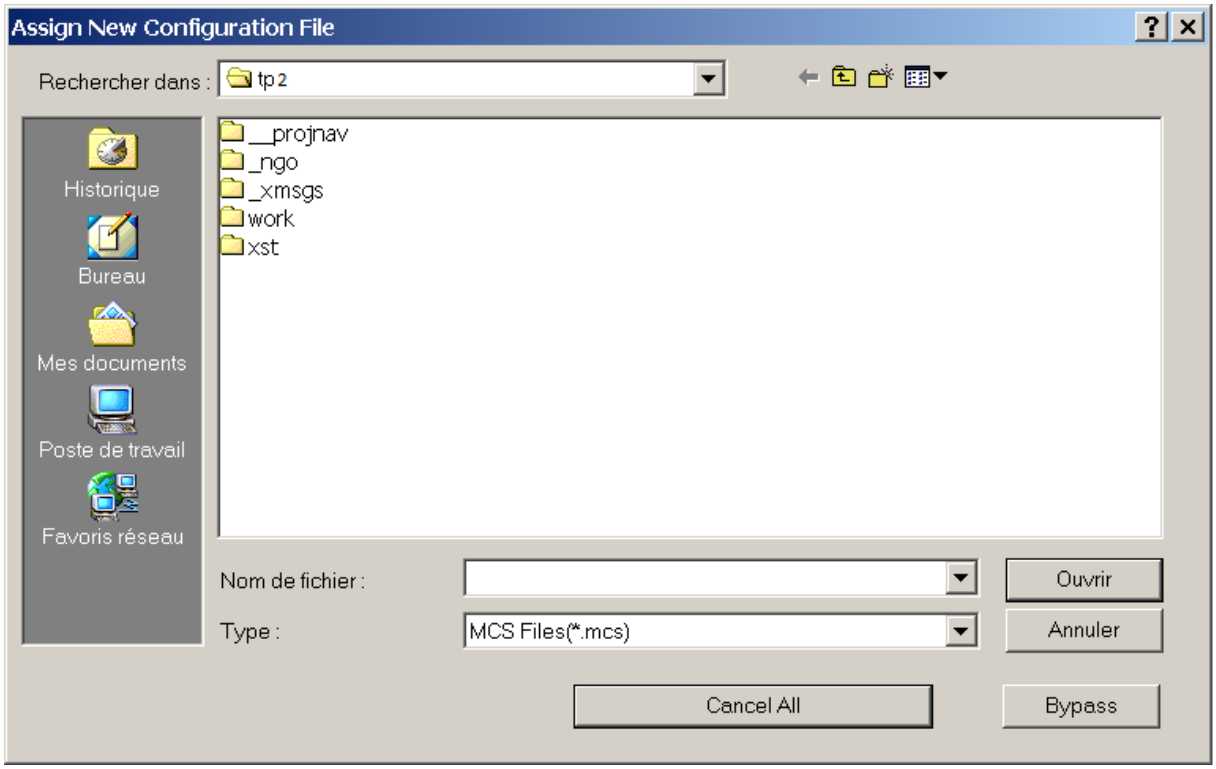
La question suivante concerne le nom du fichier de configuration à associer au FPGA. Dans notre exemple, il s'agit de tp2.bit. Sélectionnez-le, puis cliquez sur « Ouvrir » :



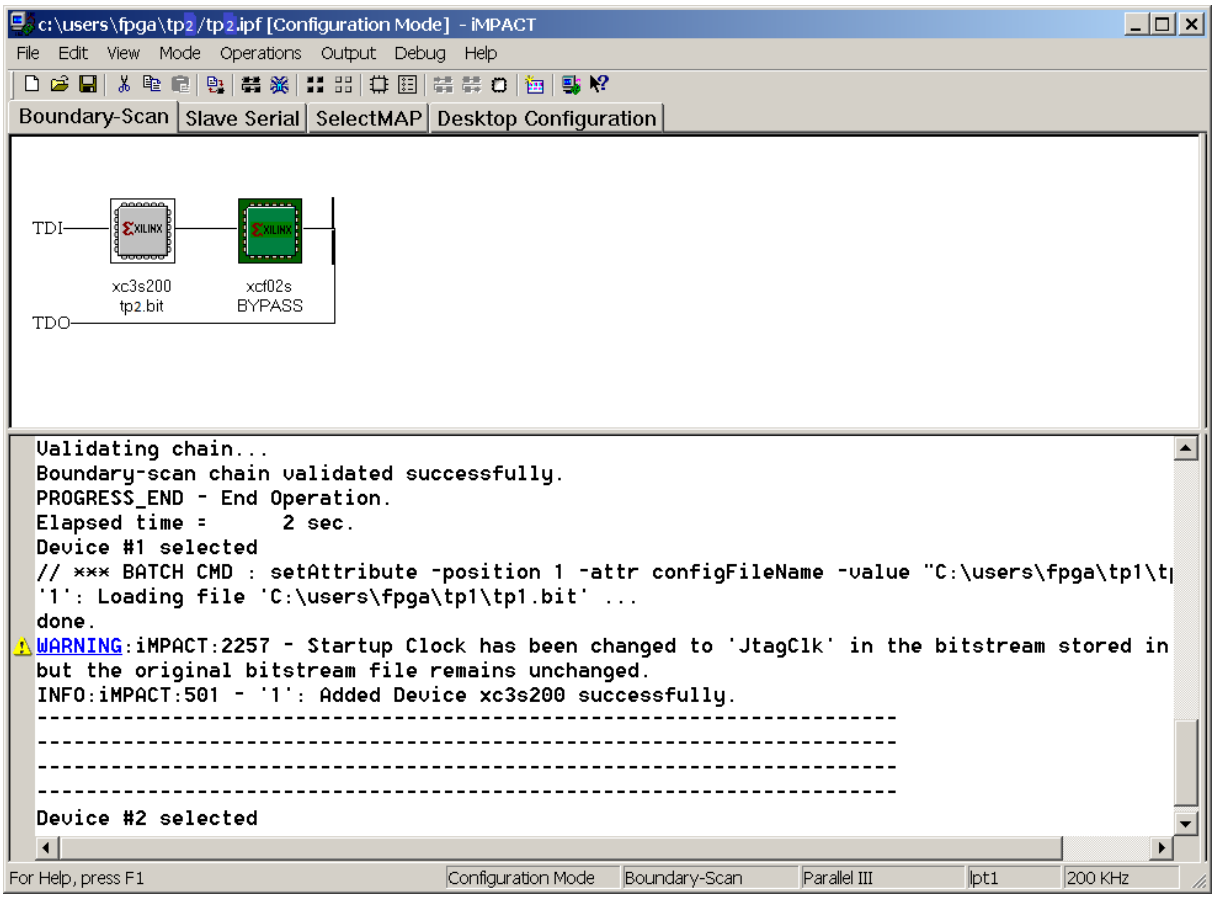
Par défaut, le FPGA est en mode maître, c'est à dire qu'il génère l'horloge CCLK de lecture de la configuration. Comme en JTAG, le FPGA est esclave, la fenêtre suivante vous avertit que l'horloge dans le fichier de configuration tp2.bit a été passée en mode JTAG (sur la copie en mémoire seulement). Cliquez sur « OK » pour poursuivre :



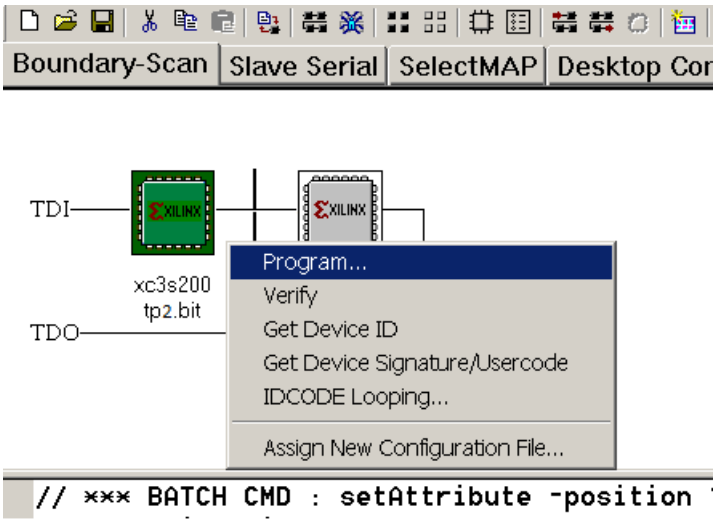
Impact demande ensuite quel fichier va être associé avec la mémoire Flash série. Comme nous n'avons pas créé le fichier pour la PROM (format MCS-86), cliquez sur « Bypass » pour sauter cette étape :



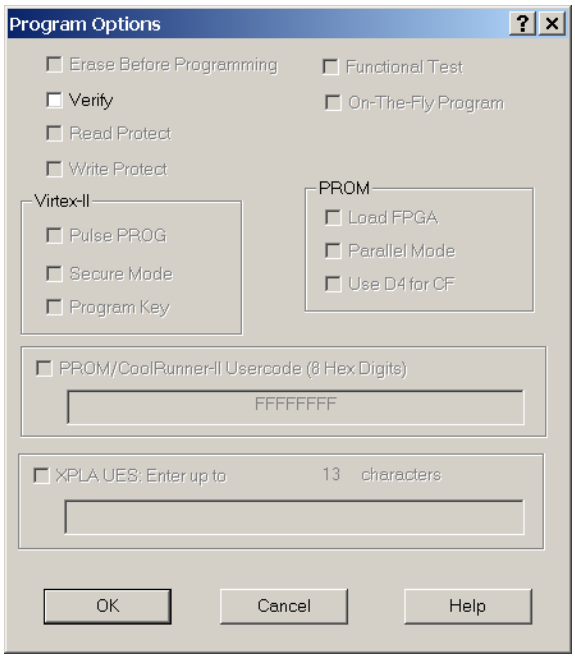
Finalement, on obtient la fenêtre suivante :



Pour configurer le FPGA, il suffit de le sélectionner en cliquant dessus, puis de cliquer avec le bouton droit de la souris puis sur « Program… » :



Dans la fenêtre qui s’ouvre, cliquez sur « OK » :



En moins de 10 secondes, le téléchargement est terminé. Fermez la fenêtre Impact sans sauver le projet. Essayez les différentes combinaisons de SW0, SW1 et SW2 et vérifiez à l'aide des leds LD0 à LD4 le bon fonctionnement du montage.

7.3 Travail pratique N°3

En utilisant le même flot de développement, nous allons pouvoir tester certains designs vus en cours, à savoir :

- Multiplexeurs (voir §2.4.3),
- Mémoire ROM (voir §2.4.7).

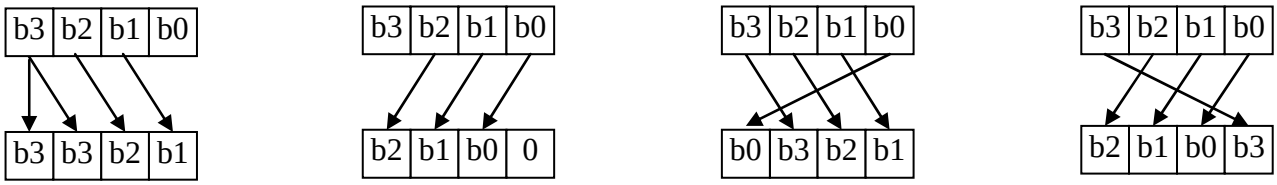
Vous allez créer successivement 2 nouveaux projets TP3_1 et TP3_2 et saisir vos designs. Vous écrirez un testbench pour faire la simulation fonctionnelle de chacun des designs, puis vous les implémenterez dans le FPGA. Vous appellerez un enseignant pour valider chaque design. Vous connecterez les entrées sur les dip switches SW0 à SW7 et les sorties sur les leds LD0 à LD7 en partant des poids faibles. N'oubliez pas de modifier le fichier UCF pour chaque design.

7.4 Travail pratique N°4

7.4.1 Cahier des charges

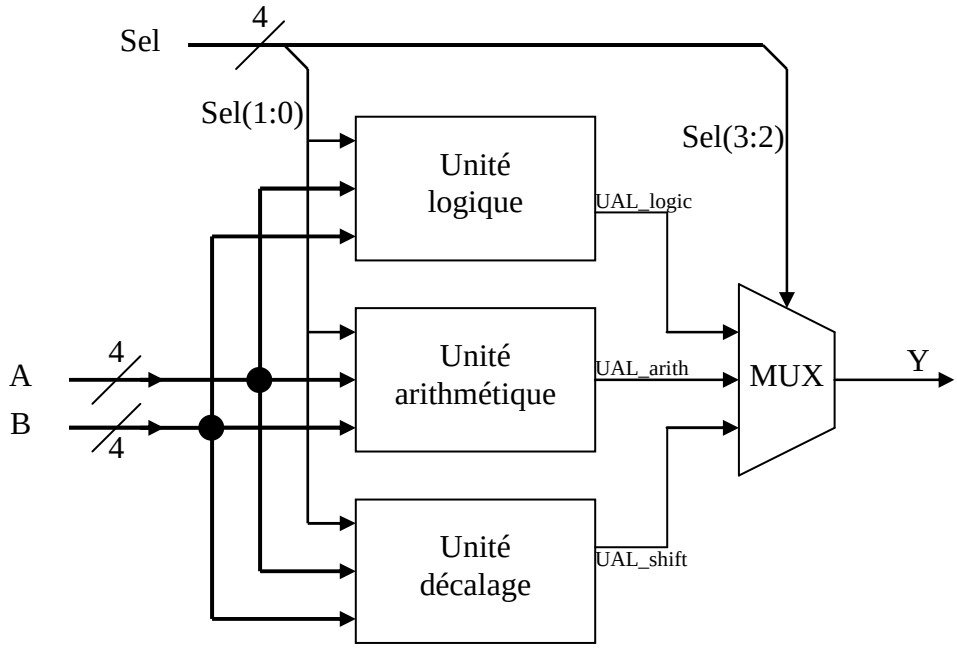
Le but de ce TP est de développer une Unité Arithmétique et Logique (UAL ou ALU en anglais) qui va réaliser :

- Des opérations arithmétiques (addition, soustraction, incrémentation, décrémentation),
- Des opérations logiques (and, or, xor et not),
- Des décalages ou des rotations (à gauche et à droite) symbolisés par la figure suivante. Vous noterez que le décalage à droite est réalisé avec extension de signe.



Décalage à droite (Shift right) | Décalage à gauche (Shift left) | rotation droite (rotate right) | rotation gauche (rotate left)

Le synoptique général est le suivant :



On a deux opérandes A et B sur 4 bits en entrée et une sortie sur 4 bits, Y (tous non signés). Le signal Sel, sur 4 bits, sert à sélectionner l'opération à réaliser, selon le tableau suivant :

Sel(3)	Sel(2)	Sel(1)	Sel(0)	opération	
0	0	0	0	Y = A + B	arithmétique
0	0	0	1	Y = A - B	
0	0	1	0	Y = A + 1	
0	0	1	1	Y = A - 1	
0	1	0	0	Y = A and B	logique
0	1	0	1	Y = A or B	
0	1	1	0	Y = A xor B	
0	1	1	1	Y = not B	
1	0	0	0	Y = shift left A	décalage, rotation
1	0	0	1	Y = shift right A	
1	0	1	0	Y = rotate left A	
1	0	1	1	Y = rotate right A	

Les 2 bits de poids fort de Sel servent à sélectionner l'unité de traitement. Les 2 bits de poids faible de Sel servent à sélectionner une opération dans l'unité en cours d'utilisation.

7.4.2 Réalisation pratique

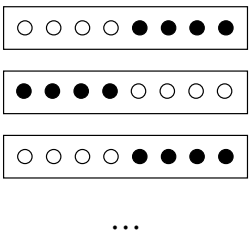
Vous allez créer un nouveau projet TP4 et saisir votre design. Vous écrirez un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez Sel sur les boutons poussoirs de la carte (BTN0 à BTN 3), A et B sur les dip switches SW0 à SW7 et Y sur les leds LD0 à LD3. Vous vous aiderez de la simulation fonctionnelle pour valider l'UAL sur la maquette.

7.5 Travail pratique N°5

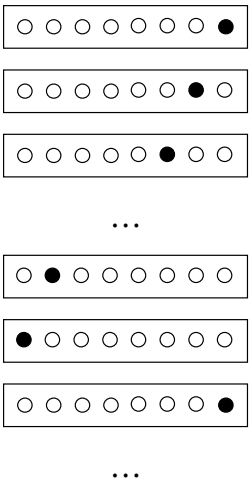
7.5.1 Cahier des charges

Le but de ce TP est de développer un chenillard qui va réaliser trois séquences de clignotement différentes sur les 8 leds avec une horloge externe basse fréquence (entrée Hin sur la maquette, fréquence 2 Hz) :

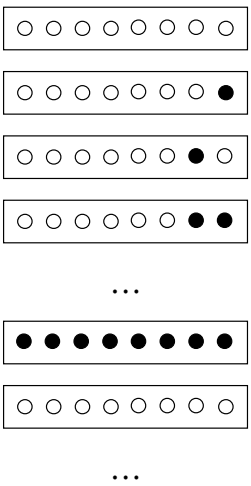
- Séquence 1 (registre),



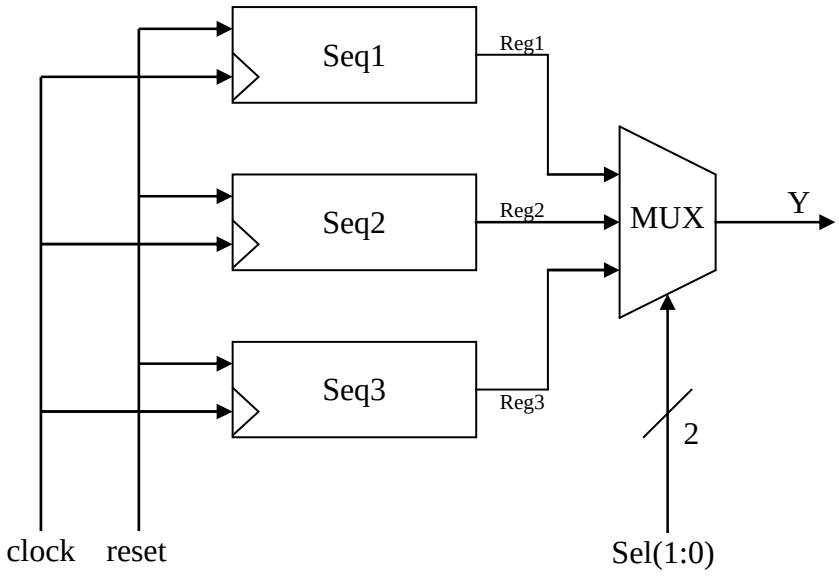
- Séquence 2 (décalage),



- Séquence 3 (comptage),



Le synoptique général sera le suivant :



Le signal Sel, sur 2 bits, sert à sélectionner la séquence, selon le tableau suivant :

Sel(1)	Sel(0)	
0	0	Séquence 1
0	1	Séquence 2
1	0	Séquence 3
1	1	Toutes les leds éteintes

La logique fonctionnera sur front montant avec un reset asynchrone.

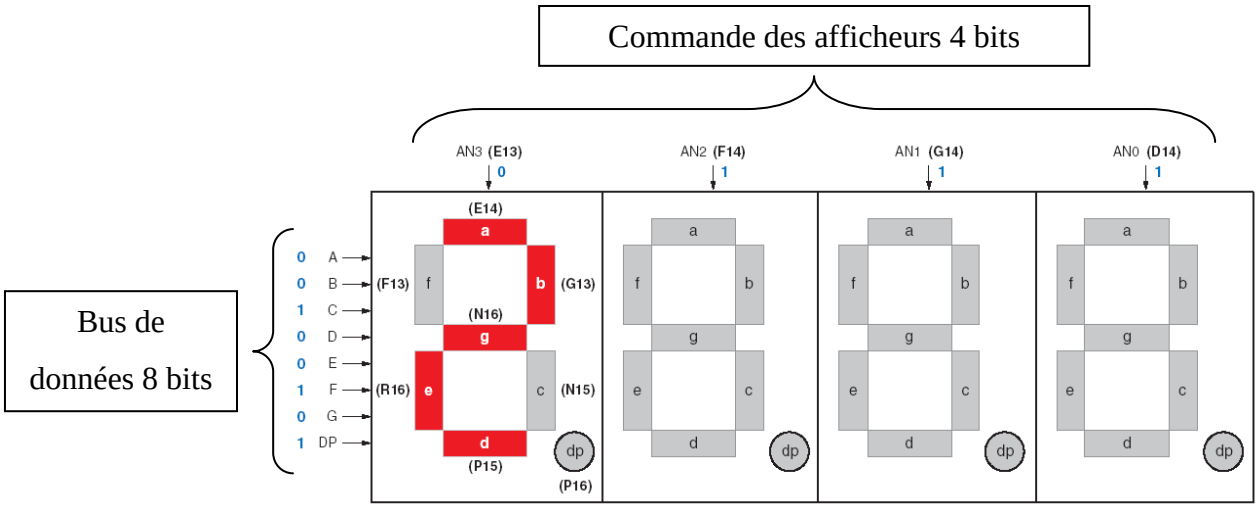
7.5.2 Réalisation pratique

Vous allez créer un nouveau projet TP5 et saisir votre design. Vous écrirez un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez Sel sur les dip switches SW0 et SW1, l'horloge sur Hin, le Reset sur le bouton poussoir BTN3 et Y sur les leds LD0 à LD7.

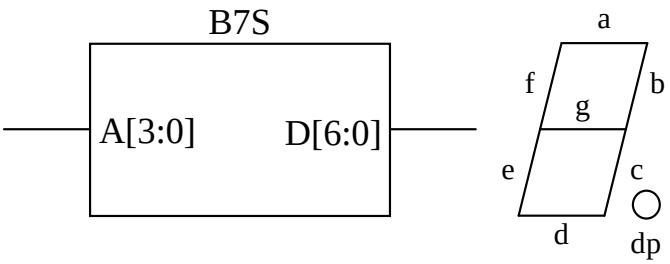
7.6 Travail pratique N°6

7.6.1 Cahier des charges

Le but de ce TP est de réaliser un compteur décimal allant de 0 à 9999 qui utilise les 4 afficheurs 7 segments de la maquette. Ces quatre afficheurs 7 segments sont multiplexés. Un segment s'allume quand l'entrée correspondante (A, B, …, G, DP) est au niveau bas et quand la commande AN (pour anode) de l'afficheur est à 0.

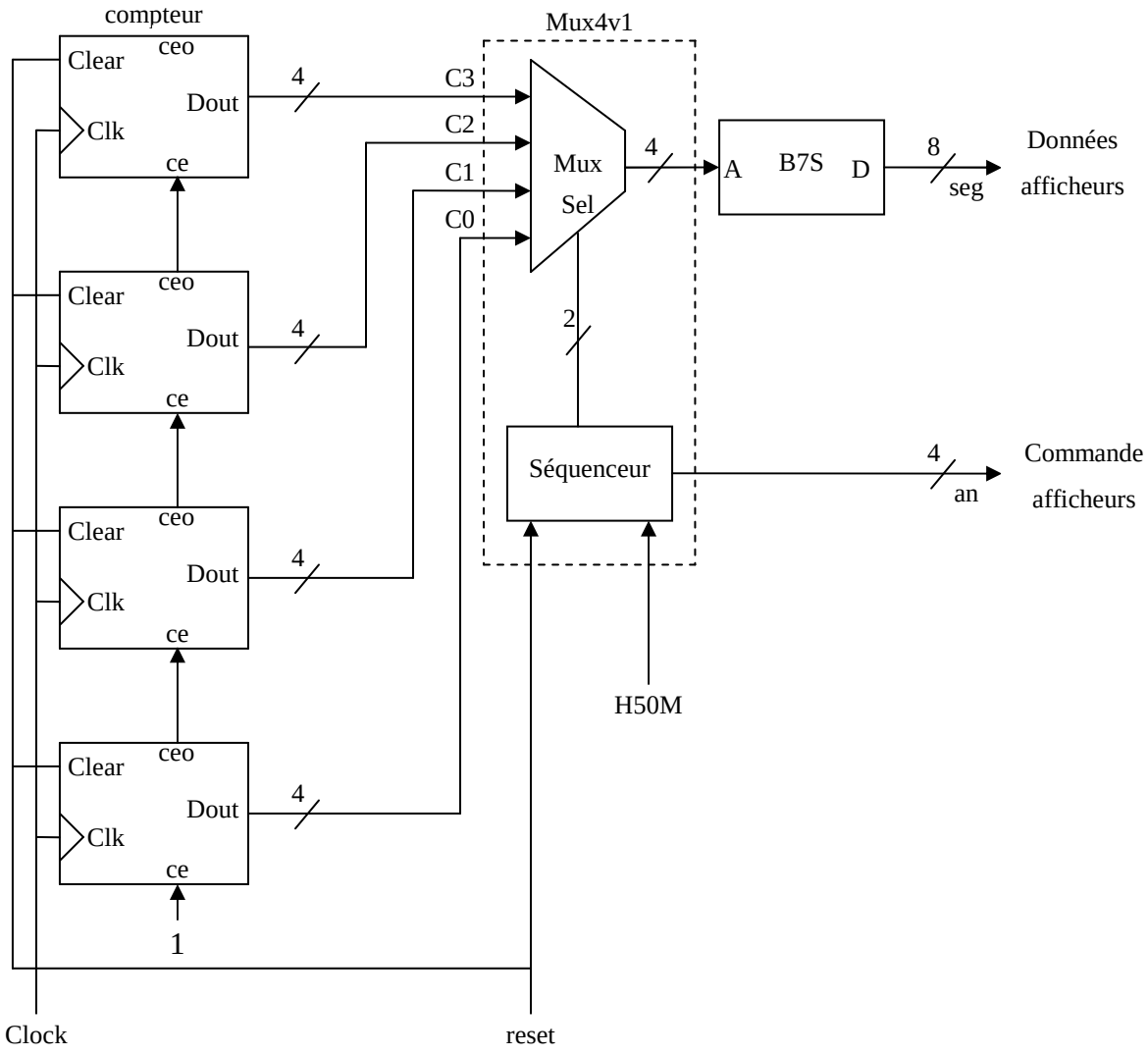


Une mémoire morte (16 x 8 bits) assurera la conversion BCD-7 segments afin de pouvoir commander un afficheur 7 segments à partir d'un compteur BCD. Le tableau suivant nous donne le contenu de la mémoire :

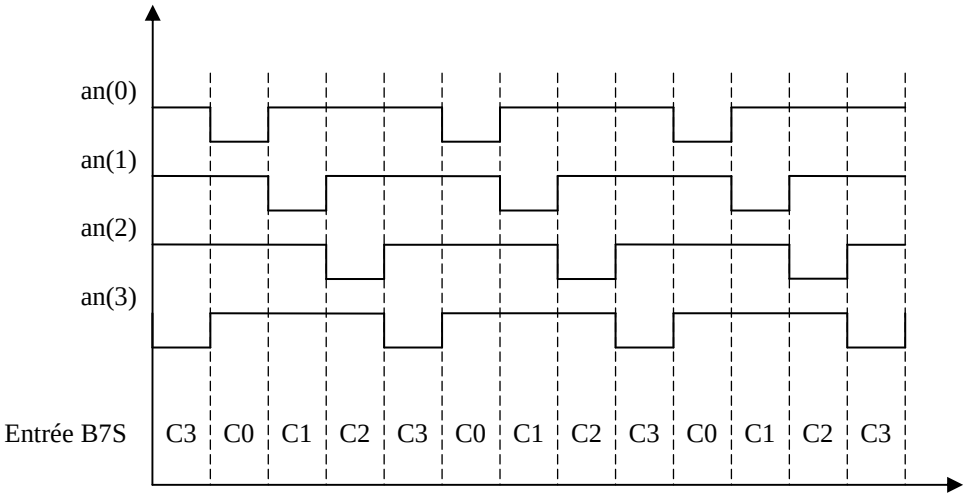


	D[7:0]							
A[3:0]	dp	g	f	e	d	c	b	a
0	0	1	0	0	0	0	0	0
1	0	1	1	1	1	0	0	1
2	0	0	1	0	0	1	0	0
3	0	0	1	1	0	0	0	0
4	0	0	0	1	1	0	0	1
5	0	0	0	1	0	0	1	0
6	0	0	0	0	0	0	1	0
7	0	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	1	0	0	0	0

Pour utiliser les 4 afficheurs en même temps, il faut les allumer successivement et présenter la donnée correspondant au chiffre allumé sur le bus de données. L'allumage de chaque afficheur doit durer environ une milliseconde pour qu'il n'y ait pas de fluctuation lumineuse pendant l'affichage (grâce à la persistance rétinienne, l'œil humain est incapable de voir défiler les afficheurs). L'horloge interne 50 MHz de la maquette servira à créer le signal de commande des afficheurs. Le synoptique général du design sera le suivant :

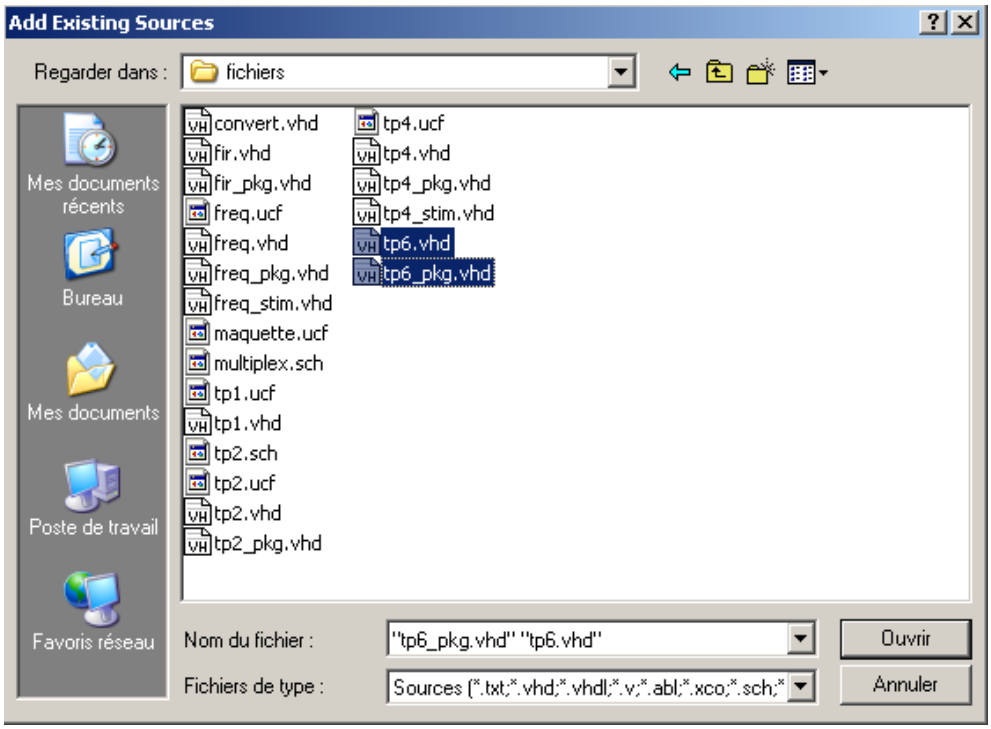


La logique fonctionnera sur front montant avec un reset asynchrone. Le chronogramme suivant montre la relation entre la commande des afficheurs et le chiffre placé sur les adresses du bloc B7S :

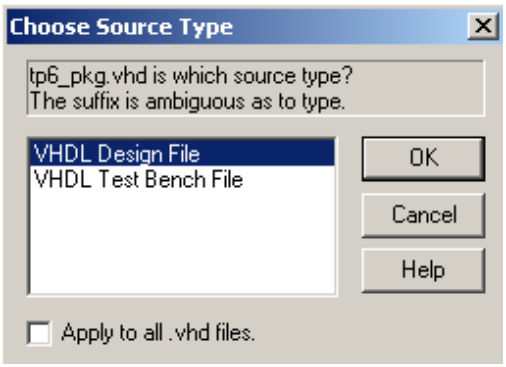


7.6.2 Ecriture du modèle en VHDL

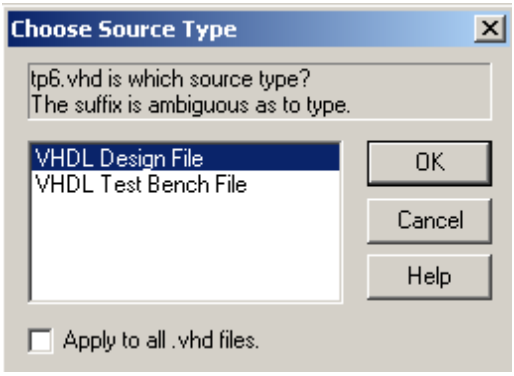
Le projet TP6 va comporter 2 fichiers : le top level design (tp6.vhd) et un package contenant les composants nécessaires à son fonctionnement (tp6_pkg.vhd). Ces deux fichiers sont déjà partiellement écris et se trouvent dans c:\users\fpga\fichiers. Pour les insérer dans le projet, cliquez sur « Project », « Add Copy of Source ». Sélectionnez les 2 fichiers (cliquez sur le nom du premier, appuyez sur la touche Ctrl du clavier et cliquez sur le nom du deuxième) puis cliquez sur ouvrir :



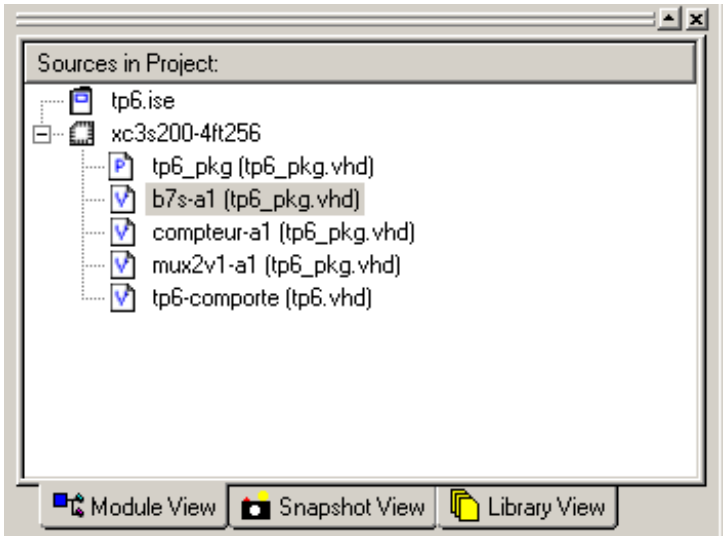
La première fenêtre qui s'ouvre demande quel est le type du fichier tp6_pkg.vhd. Comme il s'agit d'un fichier de design, sélectionnez « VHDL Design File », puis cliquez sur OK :



Le deuxième fichier tp6.vhd contient le design principal tp6. Sélectionnez toujours « VHDL Design File » dans la deuxième fenêtre, puis cliquez sur OK :



Le design apparaît maintenant dans le navigateur de projet :



Le package tp6_pkg.vhd est complet et ne doit pas être modifié. Il contient les composants nécessaires à la réalisation du projet. Son contenu est le suivant :

```
library IEEE;
use IEEE.std_logic_1164.all;

package tp6_pkg is

     COMPONENT compteur
          generic (WDTH : integer :=4; STOP : integer :=10);
          port( CLK : in std_logic ;
                 CE : in std_logic ;
             CLEAR : in std_logic;
               CEO : out std_logic;
              DOUT : out std_logic_vector(WDTH -1 downto 0));
     END COMPONENT;

     COMPONENT b7s
        port (addr : in std_logic_vector(3 downto 0);
              dout : out std_logic_vector(7 downto 0));
     END COMPONENT;

     COMPONENT mux4v1
         port( H50M : in std_logic;
              CLEAR : in std_logic;
                 C0 : in std_logic_vector(3 downto 0);
                 C1 : in std_logic_vector(3 downto 0);
                 C2 : in std_logic_vector(3 downto 0);
                 C3 : in std_logic_vector(3 downto 0);
            chiffre : out std_logic_vector(3 downto 0);
                 an : out std_logic_vector(3 downto 0));
     END COMPONENT;

end tp6_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity compteur is
          generic (WDTH : integer :=4; STOP : integer :=10);
          port( CLK : in std_logic ;
                 CE : in std_logic ;
             CLEAR : in std_logic;
               CEO : out std_logic;
              DOUT : out std_logic_vector(WDTH -1 downto 0));
end compteur;

architecture a1 of compteur is
     signal INT_DOUT : std_logic_vector(DOUT'range) ;
begin

process(CLK, CLEAR) begin
      if (CLEAR='1') then
         INT_DOUT <= (others => '0');
     elsif (CLK'event and CLK='1') then
         if (CE='1') then
            if (INT_DOUT=STOP-1) then
               INT_DOUT <= (others => '0');
            else
               INT_DOUT <= (INT_DOUT + 1);
            end if;
         end if;
     end if;
end process;

   CEO  <=  INT_DOUT(0)  and  not  INT_DOUT(1)  and  not  INT_DOUT(2)  and
INT_DOUT(3) and CE;
```

```
   dout <= int_dout;
end;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity b7s is
     port (addr : in std_logic_vector(3 downto 0);
           dout : out std_logic_vector(7 downto 0));
end b7s;

architecture a1 of b7s is
     TYPE mem_data IS ARRAY (0 TO 15) OF std_logic_vector(7 DOWNTO 0);
     constant data : mem_data := (
     ("01000000"),
     ("01111001"),
     ("00100100"),
     ("00110000"),
     ("00011001"),
     ("00010010"),
     ("00000010"),
     ("01111000"),
     ("00000000"),
     ("00010000"),
     ("00000000"),
     ("00000000"),
     ("00000000"),
     ("00000000"),
     ("00000000"),
     ("00000000"));
begin
     dout <= data(CONV_INTEGER(addr));
end;


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux4v1 is
     port( H50M : in std_logic ;
          CLEAR : in std_logic;
             C0 : in std_logic_vector(3 downto 0);
             C1 : in std_logic_vector(3 downto 0);
             C2 : in std_logic_vector(3 downto 0);
             C3 : in std_logic_vector(3 downto 0);
        chiffre : out std_logic_vector(3 downto 0);
             an : out std_logic_vector(3 downto 0));
end mux4v1;

architecture a1 of mux4v1 is
     signal compte : std_logic_vector(15 downto 0) ;
     signal an_int : std_logic_vector(3 downto 0) ;
begin

     process(H50M, CLEAR) begin
        if (CLEAR='1') then
           compte <= (others => '0');
           an_int <= "1110";
        elsif (H50M'event and H50M='1') then
           if (compte = 49999) then
              compte <= (others => '0');
```

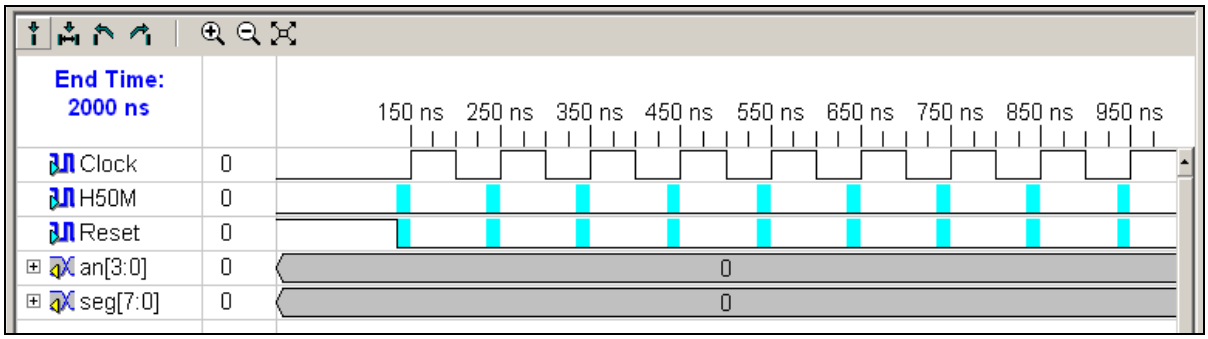
```
            an_int(3) <= an_int(2);
            an_int(2) <= an_int(1);
            an_int(1) <= an_int(0);
            an_int(0) <= an_int(3);
         else
            compte <= compte + 1;
         end if;
      end if;
   end process;

   process(an_int, C3, C2, C1, C0) begin
        if (an_int(3) = '0') then -- chiffres de gauche à droite
           chiffre <= C3;
        elsif (an_int(2) = '0')   then
           chiffre <= C2;
        elsif (an_int(1) = '0')   then
           chiffre <= C1;
        else
           chiffre <= C0;
        end if;
   end process;

   an <= an_int;
end;
```

Le top-level design tp6.vhd doit être complété avec l'instanciation des différents composants du package :

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.tp6_pkg.all;

entity tp6 is
    port(
        Clock : in std_logic;
        H50M : in std_logic;
        Reset : in std_logic;
        an : out std_logic_vector(3 downto 0);
        seg : out std_logic_vector(7 downto 0));
end tp6;

architecture comporte of tp6 is
    signal C0, C1, C2, C3 : std_logic_vector(3 downto 0);
    signal chiffre : std_logic_vector(3 downto 0);
    signal ceo0, ceo1, ceo2 : std_logic;
begin

--complétez l'architecture

end comporte ;
```

7.6.3 Testbench

Clock est l'horloge principale de notre design. On va lui affecter une fréquence égale à 10 MHz, avec un temps mort au démarrage de 100 ns. La durée de simulation est égale à 2000 ns. Vous devez obtenir le chronogramme suivant avant de passer à la simulation. N'oubliez pas le reset :



L'horloge à 50 MHz va rester à 0 pendant la simulation. Cette horloge sert à multiplexer les afficheurs 7 segments. En effet, pour des raisons de consommation, on n'allume pas les 4 afficheurs en même temps, mais les uns à la suite des autres. Dans le bloc Mux4v1, on divise l'horloge 50 MHz par 50000, puis on crée sur les 4 bits de `an` la séquence 1110, 1101, 1011,… On envoie alternativement la valeur des compteurs vers l'afficheur sélectionné. Pour garder une durée de simulation acceptable, nous ne simulerons pas cette partie car elle demanderait au moins 250000 périodes d'horloge pour afficher quelque chose d'exploitable.

7.7 Travail pratique N°7

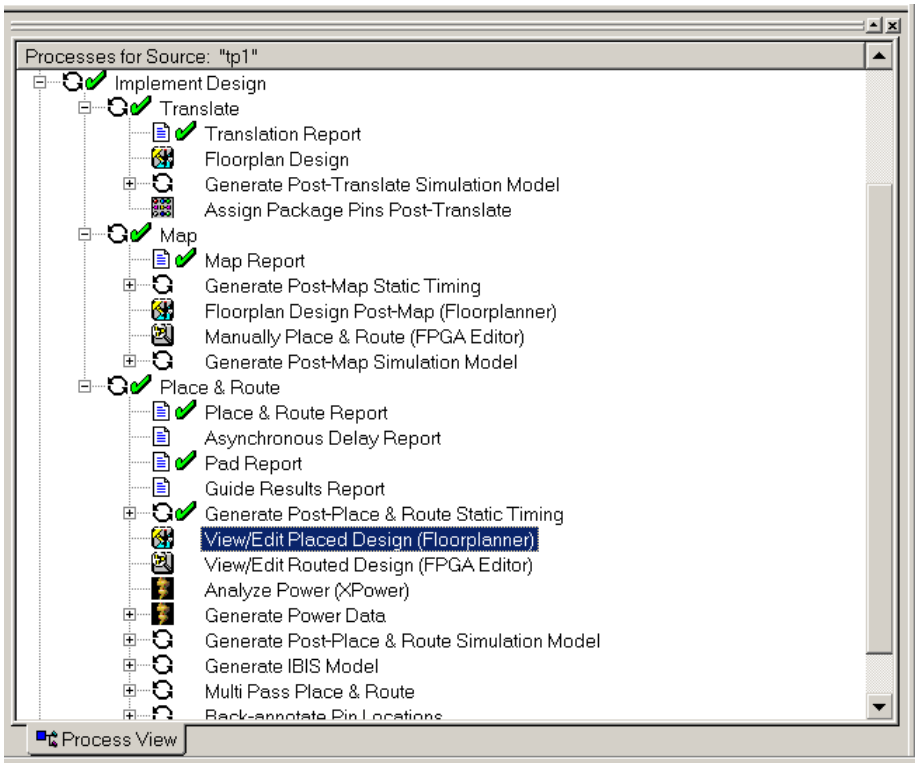
Ce troisième TP a pour but de vous exposer des aspects importants de l'implémentation d'un FPGA :

1. Les rapports après implémentation,
2. L'analyse de timing,
3. Les contraintes,
4. La vérification de la programmation du FPGA,
5. La programmation de la mémoire Flash série,
6. L'accès à l'information.

Nous allons pour cela utiliser les fichiers générés dans TP1.

7.7.1 Les rapports d'implémentation

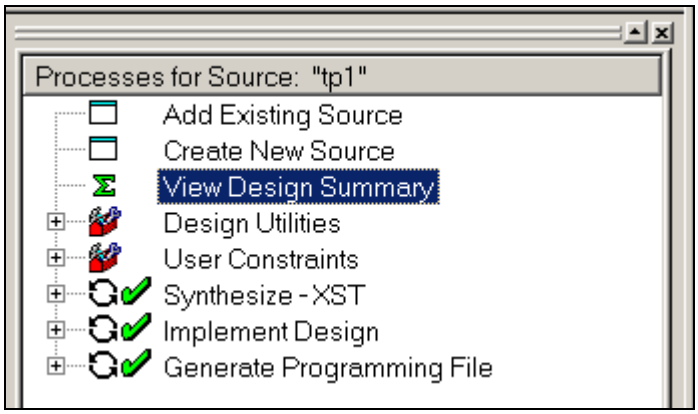
Vous pouvez consulter dans le navigateur de projet de nombreux rapports concernant les différentes phases de l'implémentation. Il faut pour cela regarder toutes les icônes de type texte () dont le nom se termine par Report dans la fenêtre « Processes for Current Sources : ». Une marque verte ou encore un **!** jaune signale que le rapport concernant le processus est disponible. Si le rapport n'existe pas, c'est parce que le processus correspondant n'a pas été exécuté. Pour lire un rapport, double-cliquez sur son nom.



Le tableau suivant vous indique la signification de ces rapports :

Etape	Rapport	Signification
Translate	Translation report	Création d'un fichier de design unique
Map	Map report	Découpage du design en primitives
	Post Map Static timing report	Timing sur primitives uniquement (pas de délais de routage)
Place&Route	Place & Route report	Placement et routage des primitives
	Pad report	Assignation des broches du FPGA
	Asynchronous delay report	Timing sur certains nets
	Post Place&Route static timing report	Fréquence maximale de fonctionnement, respect des contraintes
Generate Programming File	Programming File Generation report	Génération du fichier de configuration

La fenêtre de résumé, accessible en double-cliquant sur :



vous permet d'accéder plus rapidement aux rapports les plus importants. Dans quelles conditions les utilise-t-on ? Il y a plusieurs cas possibles :

1. Le flot d'implémentation s'arrête en cours d'exécution sur une erreur. Le dernier rapport généré contient le message d'erreur (le processus correspondant est signalé avec une marque rouge). Il doit être consulté pour remédier au problème.

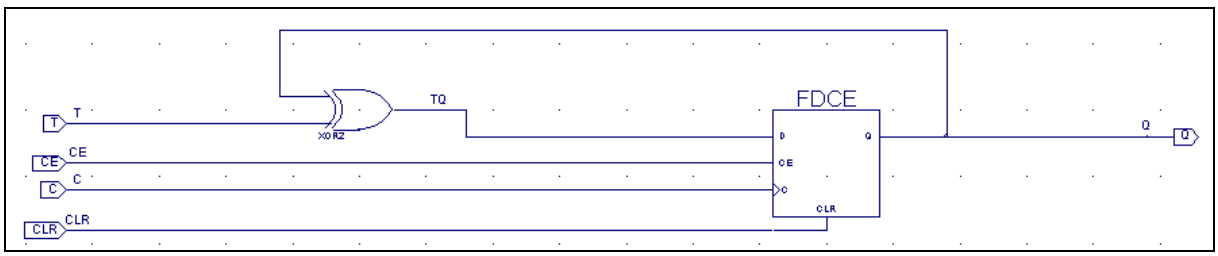
2. Le flot d'implémentation s'exécute correctement, mais le design ne fonctionne pas une fois téléchargé dans la maquette. Les rapports suivants doivent être consultés pour trouver le problème :
   - ✓ Pad report. Vérifier l'affectation des broches.
   - ✓ Post Place&Route static timing report. Vérifier le respect des contraintes temporelles.
   - ✓ Map report. Ce rapport est très important car il explique la manière dont le mapper a optimisé le design. Cette phase d'optimisation modifie le design puisqu'elle consiste en la suppression de la logique jugée inutile pour la réalisation du design. Par exemple, les sorties CEO et TC du compteur CB8CE dans TP1 ne sont pas utilisées, donc la logique correspondante peut être supprimée lors de l'implémentation. Le rapport est séparé en sections. Parmi celles-ci, la section « Removed Logic » peut s'avérer particulièrement intéressante.

3. Le design fonctionne une fois téléchargé dans la maquette. Seul le «Post Place&Route static timing report » doit être lu pour vérifier le respect des contraintes temporelles. Il faut généralement le compléter par une analyse de timing pour être certain du bon fonctionnement du design dans le pire des cas.

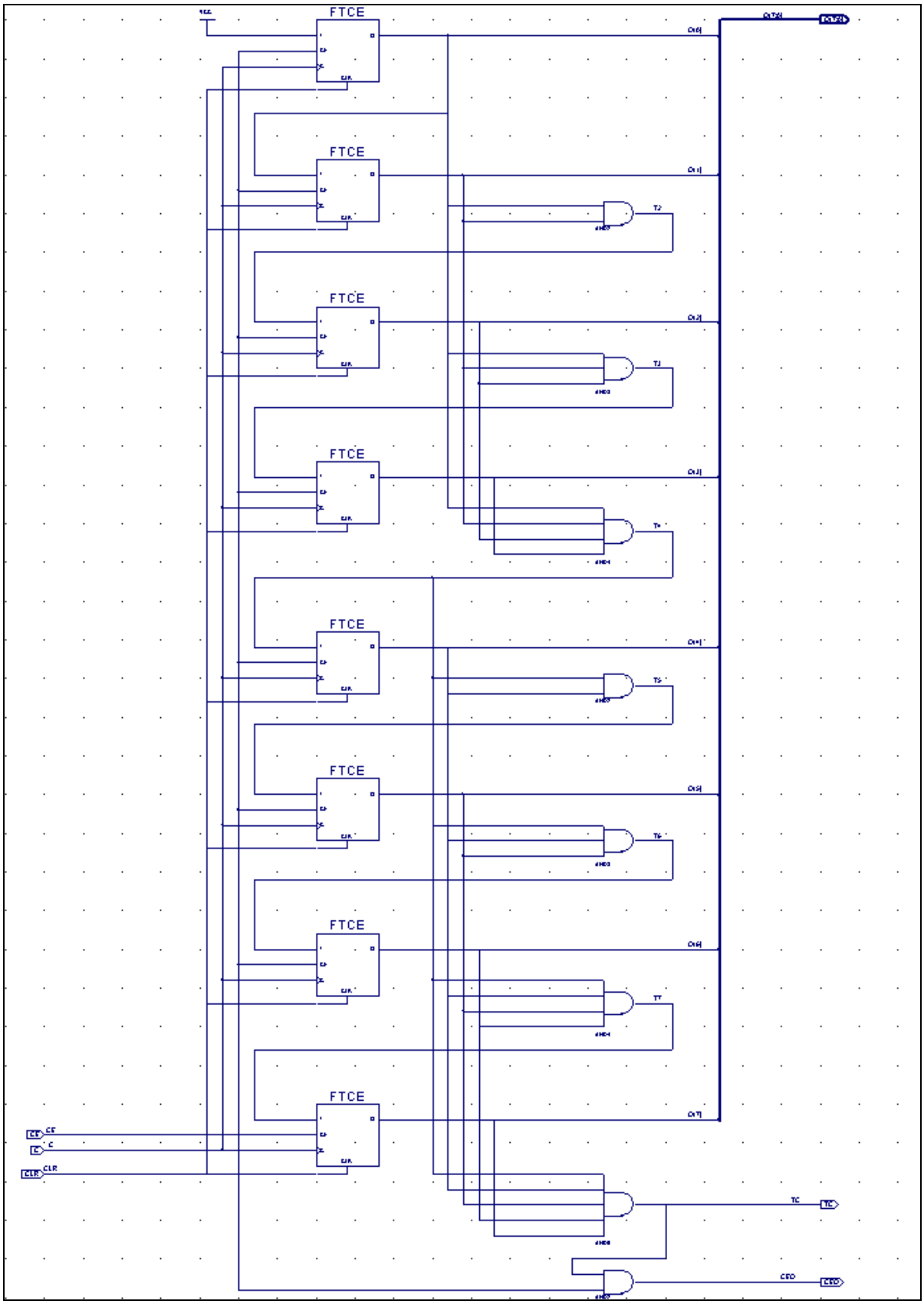
Editez les différents rapports pour le projet TP1.

7.7.2 L'analyse temporelle

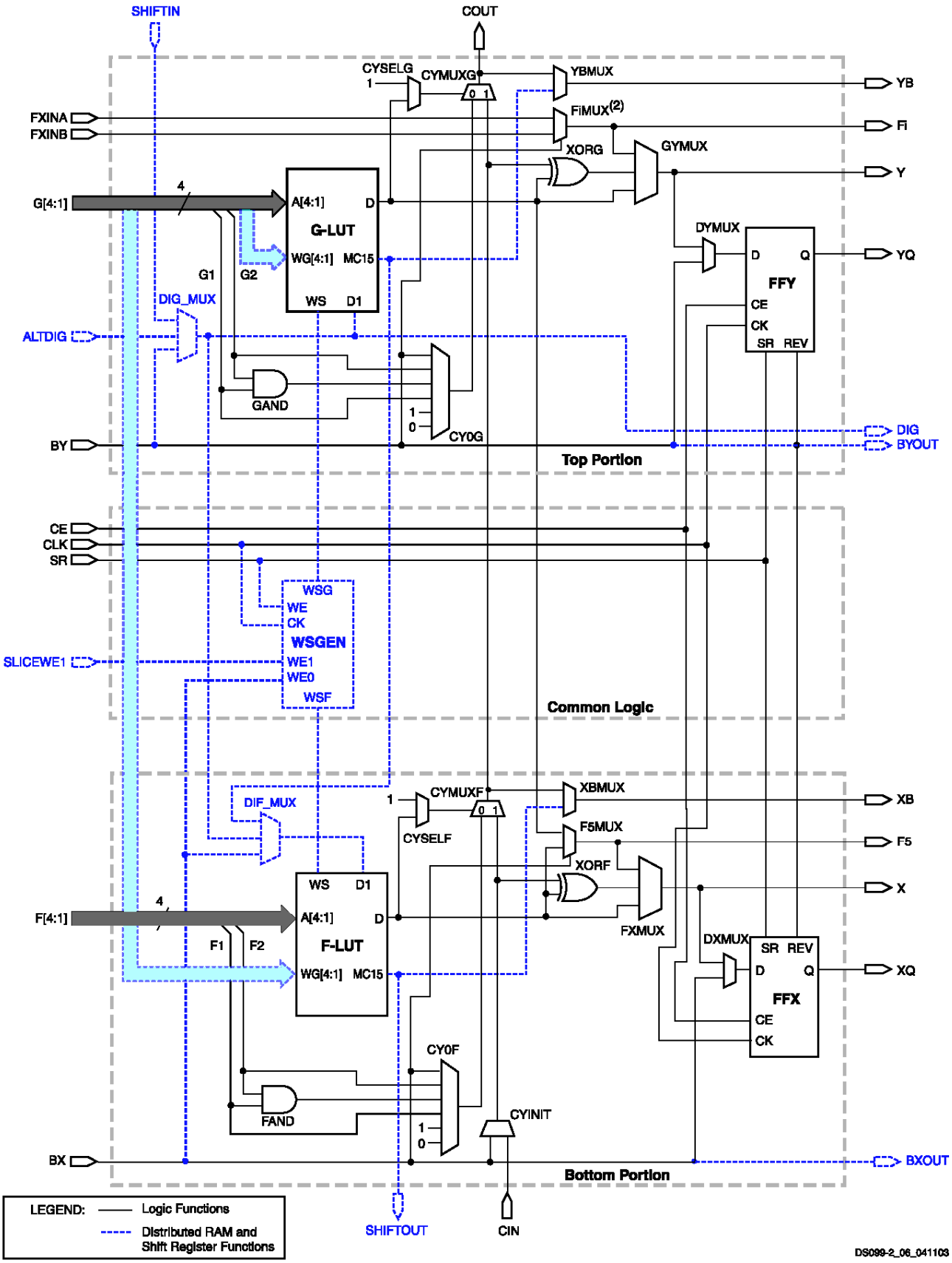
L'analyse temporelle permet de calculer les différents temps de propagation à l'intérieur du FPGA en vue de calculer la fréquence de fonctionnement du design. Pour comprendre le fonctionnement de l'application « Timing Analyzer », nous allons travailler sur le projet TP1. Le compteur CB8CE est formé à partir de bascules T (FTCE) constituées d'une bascule D (FDCE) et d'un ou exclusif :



Quand T vaut 0, la sortie de la bascule garde le même état alors que quand T vaut 1, la bascule fonctionne en diviseur de fréquence par 2. Le schéma complet du compteur est le suivant :



Pour comprendre une analyse de timing, il faut avoir en mémoire le schéma d'un CLB :

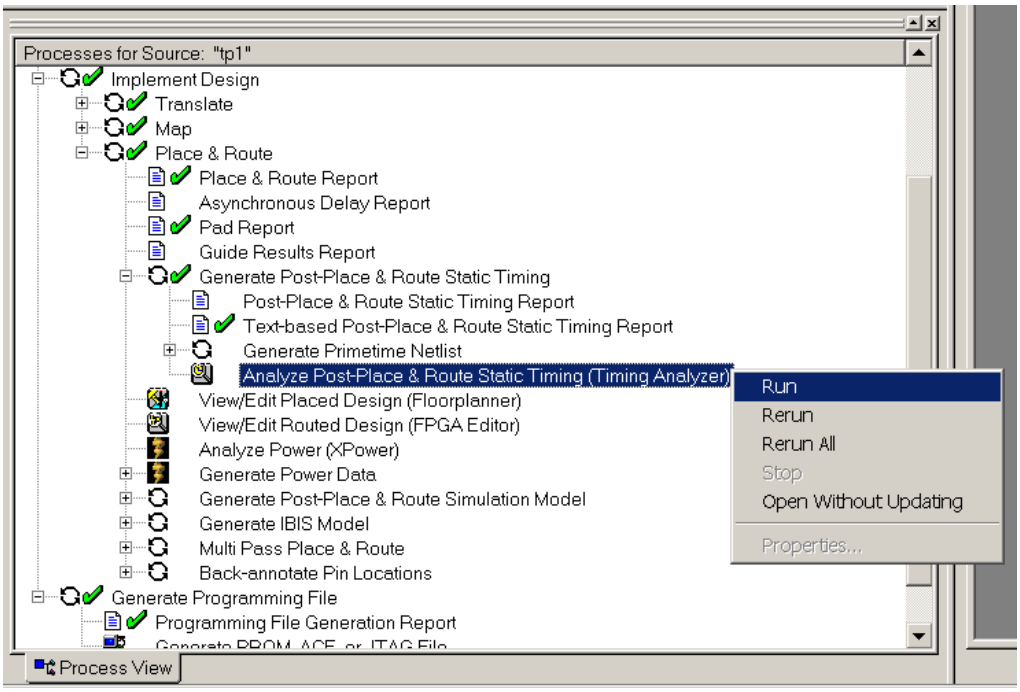


ainsi que les timings qui lui sont associés :

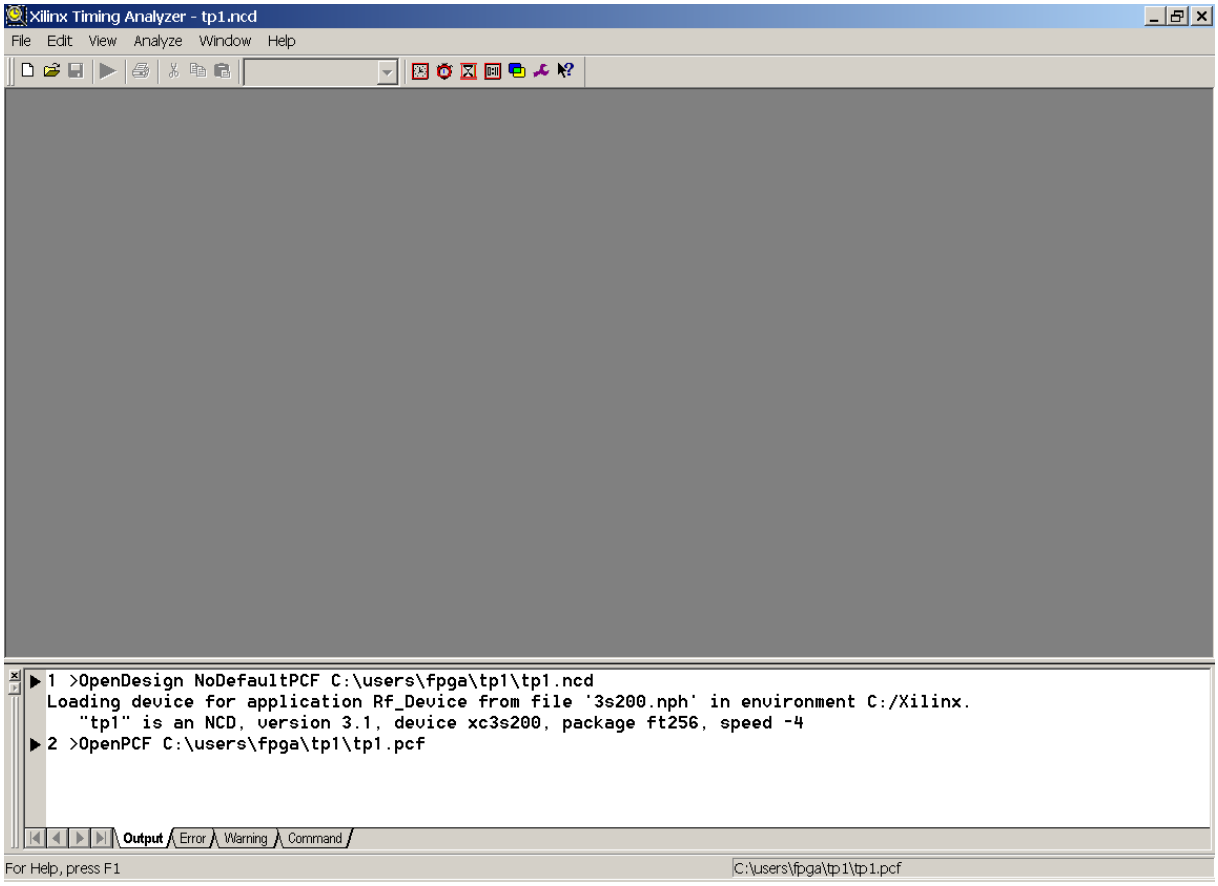
Symbol	Description	Speed Grade				Units
		-5		-4		
		Min	Max	Min	Max	
Clock-to-Output Times						
T_{CKO}	When reading from the FFX (FFY) Flip-Flop, the time from the active transition at the CLK input to data appearing at the XQ (YQ) output	-	0.63	-	0.72	ns
Setup Times						
T_{AS}	Time from the setup of data at the F or G input to the active transition at the CLK input of the CLB	0.46	-	0.53	-	ns
T_{DICK}	Time from the setup of data at the BX or BY input to the active transition at the CLK input of the CLB	0.18	-	0.21	-	ns
Hold Times						
T_{AH}	Time from the active transition at the CLK input to the point where data is last held at the F or G input	0	-	0	-	ns
T_{CKDI}	Time from the active transition at the CLK input to the point where data is last held at the BX or BY input	0.25	-	0.29	-	ns
Clock Timing						
T_{CH}	The High pulse width of the CLB's CLK signal	0.69	-	0.79	-	ns
T_{CL}	The Low pulse width of the CLK signal	0.69	-	0.79	-	ns
F_{TOG}	Maximum toggle frequency (for export control)	-	724	-	632	MHz
Propagation Times						
T_{ILO}	The time it takes for data to travel from the CLB's F (G) input to the X (Y) output	-	0.53	-	0.61	ns
Set/Reset Times						
T_{RPW}	The pulse width, High or Low, of the CLB's SR signal	0.66	-	0.76	-	ns

L'entrée de la huitième bascule T du compteur est un ET des sorties des 7 bascules de poids faibles. Ces 7 sorties suivies du ET (constitué de deux AND4 en cascade) puis du fil reliant la sortie du ET à l'entrée de la bascule constituent le **chemin critique**, c'est-à-dire le chemin combinatoire le plus long à l'intérieur du compteur. C'est le temps de propagation sur ce chemin critique qui détermine la fréquence maximale de fonctionnement du compteur.

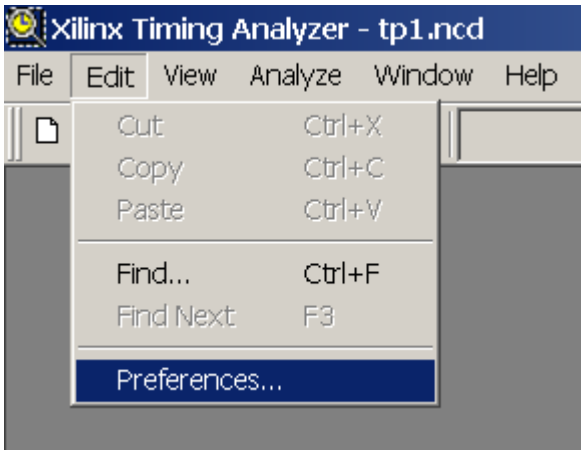
Sélectionnez le design tp1.sch dans le navigateur de projet, puis lancez l'application « Timing Analyzer » :



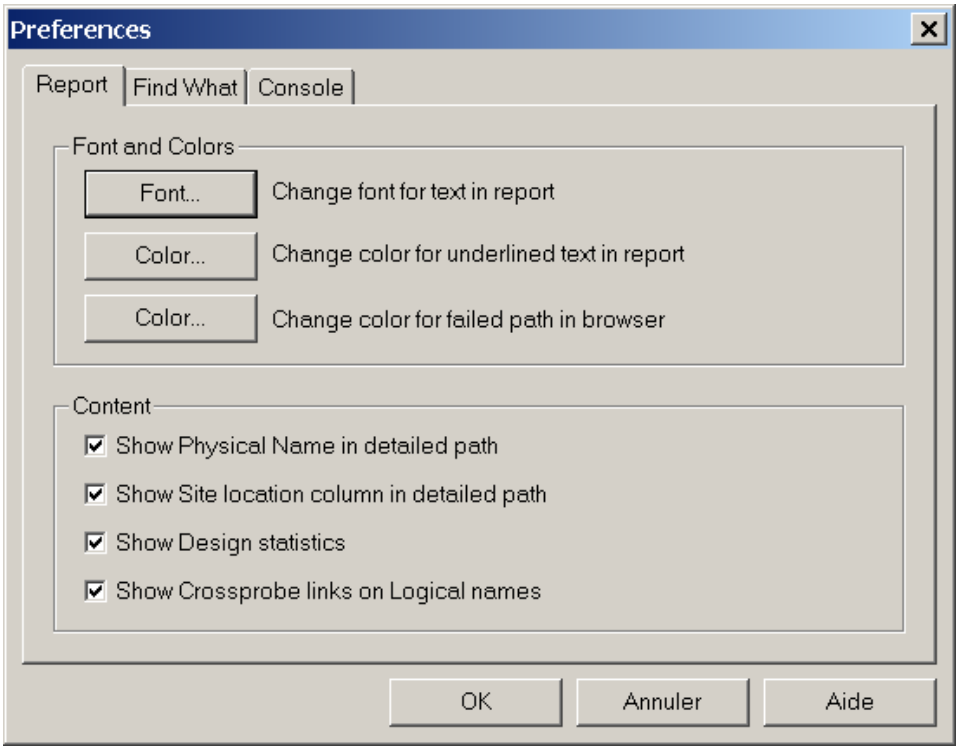
La fenêtre de l'analyseur temporel s'ouvre à l'écran :



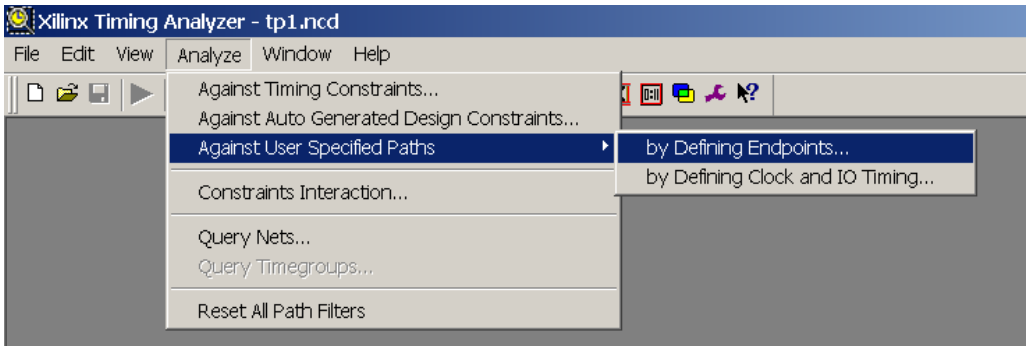
Commençons d'abord par modifier certains réglages par défaut. Cliquez sur le menu « Edit » puis sur « Preferences… » :



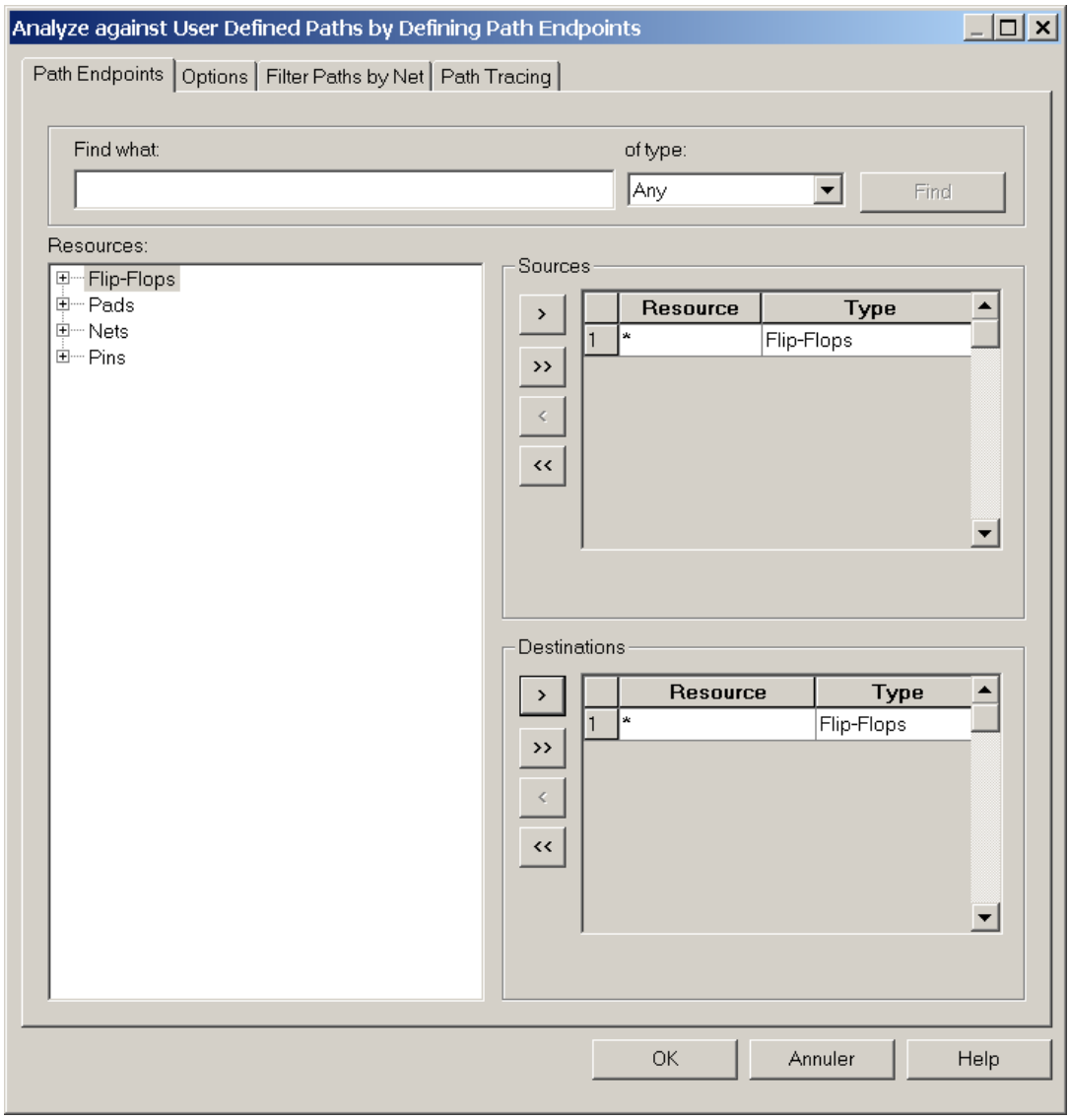
Dans la fenêtre qui apparaît, cochez toutes les cases avant de cliquer sur le bouton OK.



Analysons maintenant la fréquence maximale de fonctionnement du design. Pour cela, cliquez sur le menu « Analyze », sur le sous-menu « Against User Specified Paths » puis sur « by Defining EndPoints… » :



Nous allons sélectionner les bascules D (flip-flop) comme source et destination de l'analyse temporelle à effectuer. Pour cela, sélectionnez Flip-Flops dans « Resources » puis cliquez sur la flèche  dans Sources et Destinations. Vous devez obtenir la fenêtre suivante avant de cliquer sur OK :



Le rapport d'analyse apparaît. En sélectionnant les flip-flops comme source et destination de l'analyse, nous avons en fait sélectionné l'horloge Clock comme source et comme destination de l'analyse temporelle puisque dans une bascule D, c'est l'horloge qui sert de référence unique pour toutes les actions. Nous allons donc mesurer un temps Clock to Setup qui va nous donner la fréquence maximale de fonctionnement du design. Timing Analyzer a analysé tous les chemins (il y en a 36) correspondant à la sélection (source, destination) et indique les timings des 3 chemins les plus critiques classés par ordre décroissant (vous pouvez suivre ce chemin à l'aide de FPGA Editor).

```
Timing constraint: PATH PATHFILTERS = FROM TIMEGRP "SOURCES" TO TIMEGRP "DESTINATIONS"

 36 items analyzed, 0 timing errors detected. (0 setup errors, 0 hold errors)
 Maximum delay is   4.453ns.
--------------------------------------------------------------------------------
Delay:                  4.453ns (data path - clock path skew + uncertainty)
  Source:               Cnt8/I_Q0/I_36_35 (FF)
  Destination:          Cnt8/I_Q7/I_36_35 (FF)
  Data Path Delay:      4.453ns (Levels of Logic = 3)
  Clock Path Skew:      0.000ns
  Source Clock:         Clock_BUFGP rising
  Destination Clock:    Clock_BUFGP rising
  Clock Uncertainty:    0.000ns
  Constraint Improvement Wizard
  Data Path: Cnt8/I_Q0/I_36_35 to Cnt8/I_Q7/I_36_35
    Location             Delay type         Delay(ns)  Physical Resource
                                                       Logical Resource(s)
    -------------------------------------------------  -------------------
    SLICE_X37Y9.YQ       Tcko                  0.720   Data_0_OBUF
                                                       Cnt8/I_Q0/I_36_35
    SLICE_X36Y7.G1       net (fanout=6)        0.787   Data_0_OBUF
    SLICE_X36Y7.Y        Tilo                  0.608   Data_4_OBUF
                                                       Cnt8/I_36_15
    SLICE_X34Y4.G1       net (fanout=4)        1.025   Cnt8/T4
    SLICE_X34Y4.Y        Tilo                  0.608   Data_7_OBUF
                                                       Cnt8/I_36_28
    SLICE_X34Y4.F4       net (fanout=1)        0.015   Cnt8/T7
    SLICE_X34Y4.CLK      Tfck                  0.690   Data_7_OBUF
                                                       Cnt8/I_Q7/I_36_32
                                                       Cnt8/I_Q7/I_36_35
    -------------------------------------------------  ---------------------------
    Total                                      4.453ns (2.626ns logic, 1.827ns route)
                                                       (59.0% logic, 41.0% route)
```

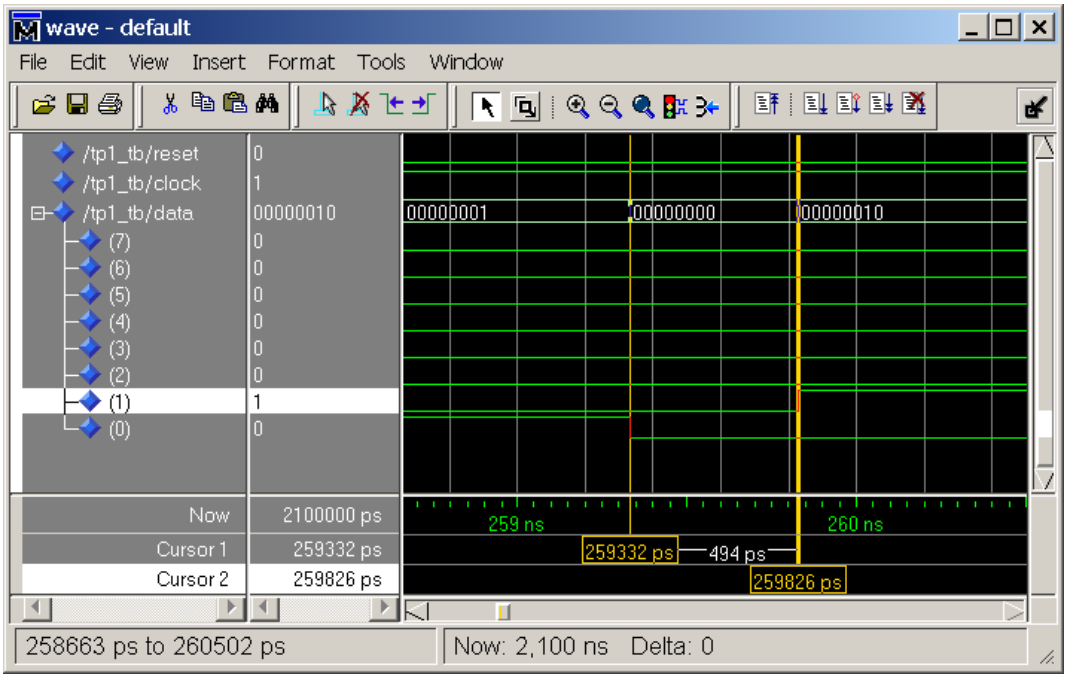
Nous pouvons vérifier sur le schéma du compteur cette analyse temporelle. La durée minimale entre deux coups d'horloge de Clock est égale à :

Description	Symbole	Valeur [ns]
Clock CLK to outputs YQ (de Clock vers SLICE_X37Y9.YQ)	T_{CKO}	0,720
Temps de propagation sur un fil (de SLICE_X37Y9.YQ vers SLICE_X36Y7.G1)	net	0,787

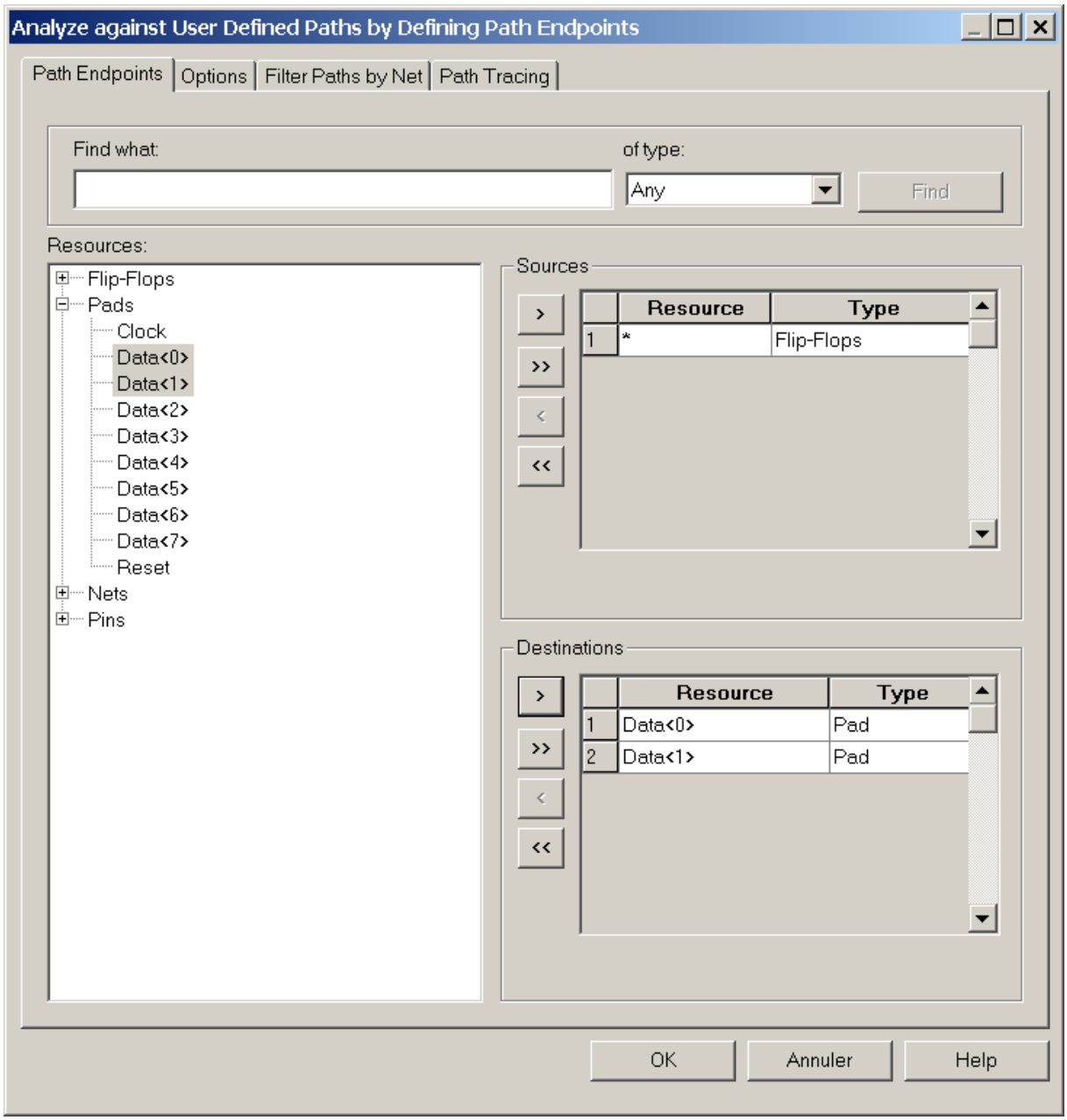
Propagation time : F/G inputs to X/Y outputs (de SLICE_X36Y7.G1 vers SLICE_X36Y7.Y)	T_{ILO}	0.608
Temps de propagation sur un fil (de SLICE_X36Y7.Y vers SLICE_X34Y4.G1)	net	1,025
Propagation time : F/G inputs to X/Y outputs (de SLICE_X34Y4.G1 vers SLICE_X34Y4.Y)	T_{ILO}	0.608
Temps de propagation sur un fil (de SLICE_X34Y4.Y vers SLICE_X34Y4.F4)	net	0,015
Temps de setup : F/G inputs to CLK (de SLICE_X34Y4.F4 vers SLICE_X34Y4.CLK)	T_{fck}	0,69
total		4,453 ns

Tous les temps sont maximums, ce qui implique une fréquence de fonctionnement d'environ 224 MHz dans le pire des cas.

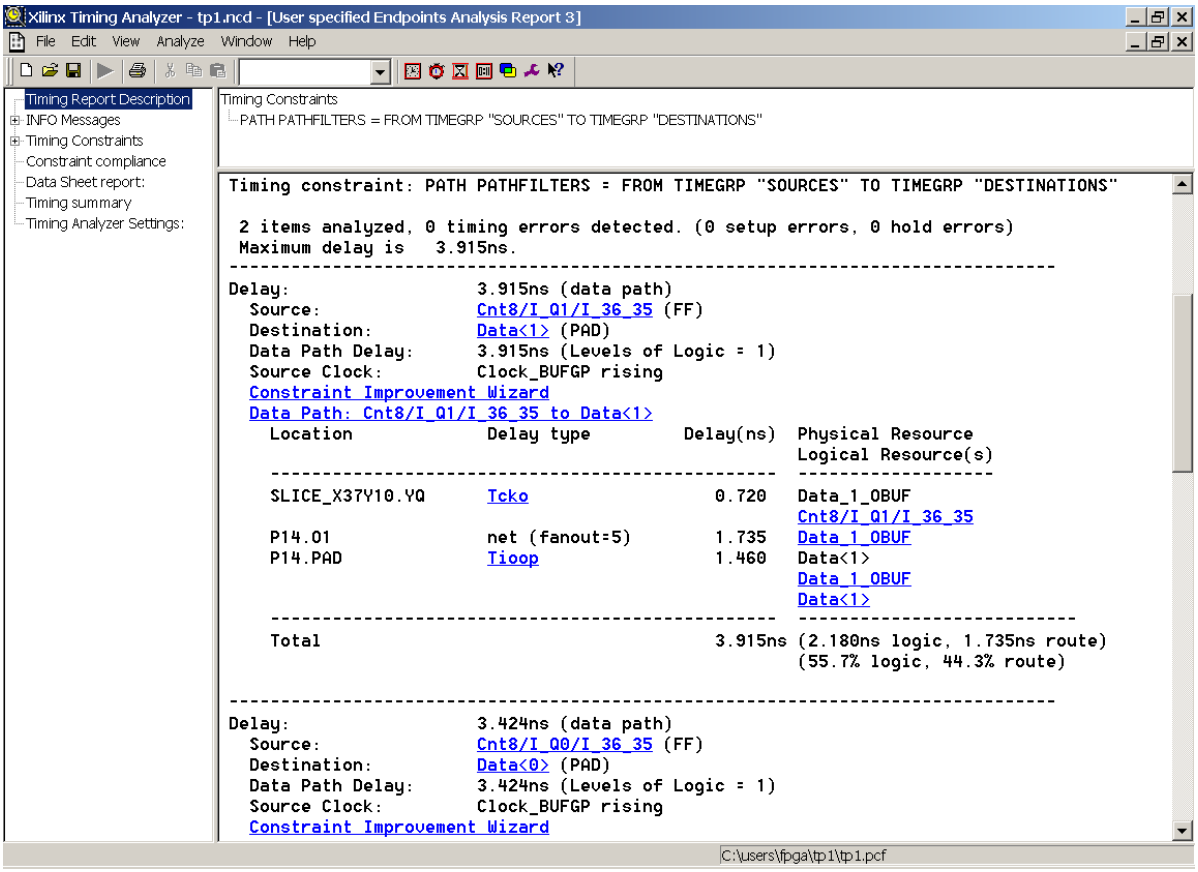
Nous allons maintenant travailler sur un phénomène qui a été vu lors de la simulation de timing sur TP1. Nous avions vu un écart temporel entre l'arrivée de la donnée sur Data<0> et l'arrivée de la donnée sur Data<1>. La fenêtre des chronogrammes était la suivante :



Nous pouvons avoir confirmation de ce décalage avec l'analyseur temporel. Pour cela, cliquez sur le menu « Analyze », sur le sous-menu « Against User Specified Path » puis sur « by Defining EndPoints… ». Sélectionnez Flip-Flops comme source et Data<0>, Data<1> comme Pads de destination puis cliquez sur OK.



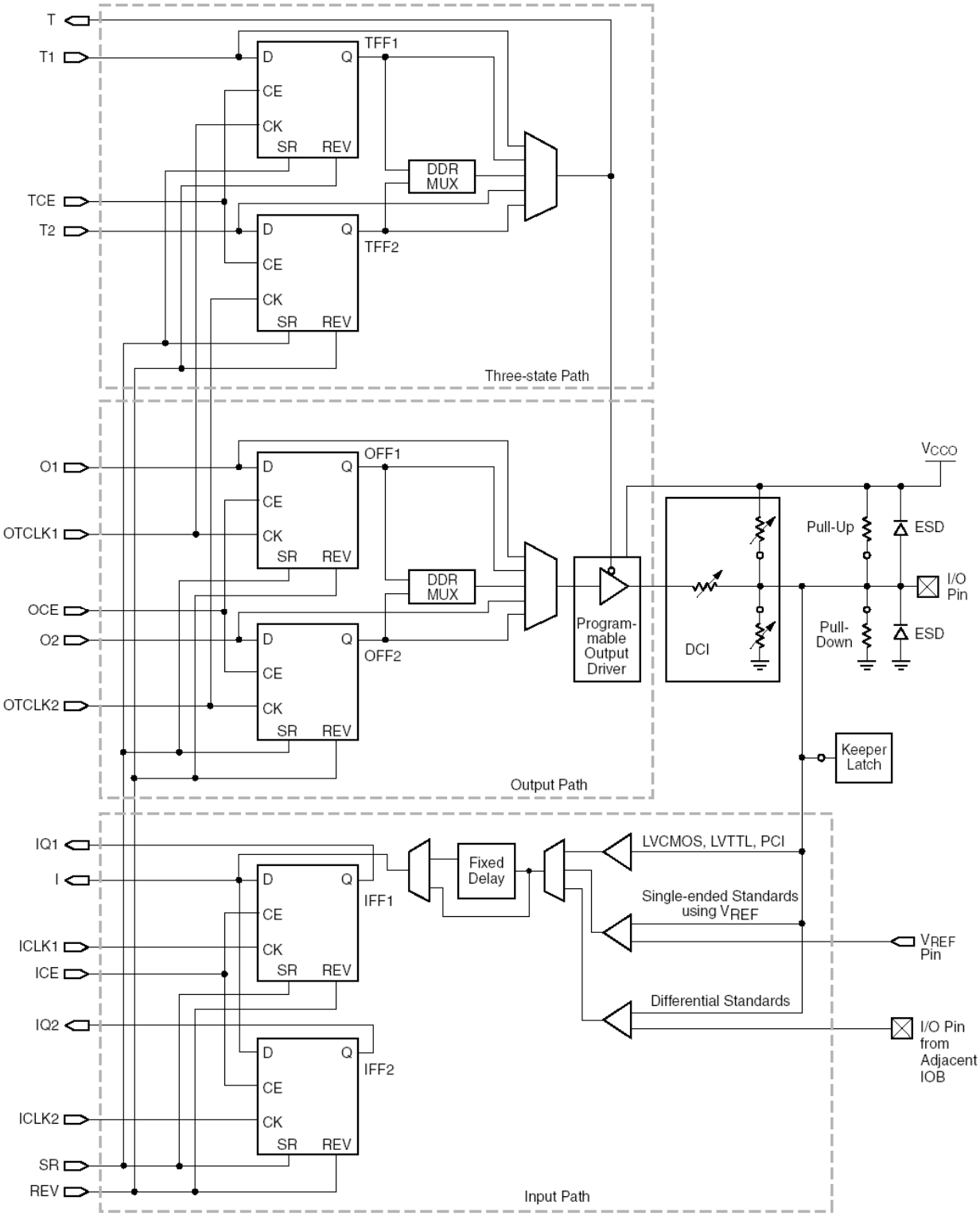
Le détail du chemin critique de Clock vers Data<0> et vers Data<1> (analyse clock to pad) apparaît dans le rapport :



Pour comprendre cet exemple, nous devons avoir en mémoire les timings d'un IOB :

Symbol	Description	Conditions	Speed Grade -5 Max	Speed Grade -4 Max	Units
Clock-to-Output Times					
T_{IOCKP}	When reading from the Output Flip-Flop (OFF), the time from the active transition at the OTCLK input to data appearing at the Output pin	LVCMOS25(2), 12mA output drive, Fast slew rate	1.50	1.72	ns
Propagation Times					
T_{IOOP}	The time it takes for data to travel from the IOB's O input to the Output pin	LVCMOS25(2), 12mA output drive, Fast slew rate	1.28	1.46	ns
T_{IOOLP}	The time it takes for data to travel from the O input through the OFF latch to the Output pin		1.50	1.72	ns
Set/Reset Times					
T_{IOSRP}	Time from asserting the OFF's SR input to setting/resetting data at the Output pin	LVCMOS25(2), 12mA output drive, Fast slew rate	2.18	2.51	ns
T_{IOGSRQ}	Time from asserting the Global Set Reset (GSR) net to setting/resetting data at the Output pin		8.07	9.28	ns

ainsi que son schéma interne :



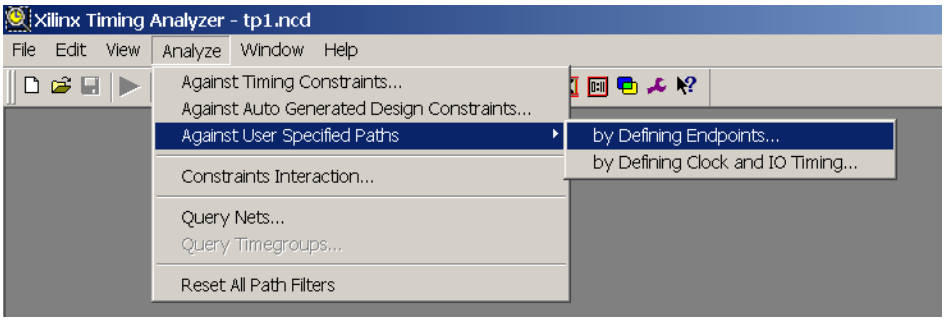
Nous pouvons vérifier cette analyse temporelle. La durée entre le front d'horloge Clock et l'arrivée de la donnée sur Data<1> est :

Description	Symbole	Valeur [ns]
Clock CLK to outputs YQ (de Clock vers SLICE_X37Y10.YQ)	T_{CKO}	0,720
Temps de propagation sur un fil (de SLICE_X37Y10.YQ vers P14.O1)	net	1,735
Propagation time : output (O) to pad (de P14.O1 vers P14.PAD)	T_{IOOP}	1,460
total		3,915

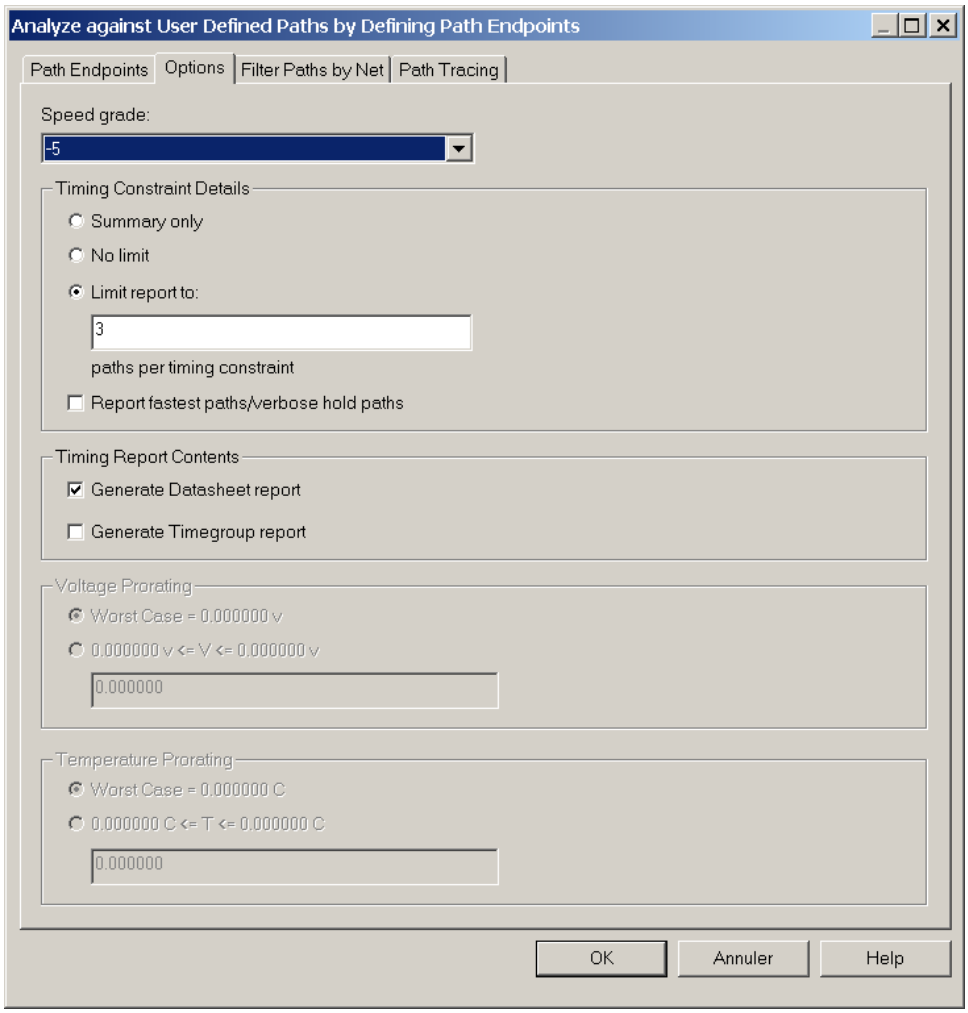
En réalisant la même analyse sur Data<0>, on obtient un temps total égal à 3,424 ns. Vous pouvez vérifier la différence (491 ps) entre les arrivées des deux signaux à l'aide de la simulation de timing. On peut aussi expliquer l'intervalle de 9,337 ns entre le front montant de Clock et l'arrivée de la donnée sur Data<0>. Il faut rajouter aux 3,424 ns précédentes le temps de propagation entre le pad F15 et le net Clock. A l'aide de l'analyseur (source : Pads (clock), destination : Flip-Flops), on trouve un temps de propagation égal à 5,913 ns :

```
================================================================================
Timing constraint: PATH PATHFILTERS = FROM TIMEGRP "SOURCES" TO TIMEGRP "DESTINATIONS"

 8 items analyzed, 0 timing errors detected. (0 setup errors, 0 hold errors)
 Maximum delay is   5.913ns.
--------------------------------------------------------------------------------
Delay:                  5.913ns (data path)
  Source:               Clock (PAD)
  Destination:          Cnt8/I_Q2/I_36_35 (FF)
  Data Path Delay:      5.913ns (Levels of Logic = 2)
  Constraint Improvement Wizard
  Data Path: Clock to Cnt8/I_Q2/I_36_35
    Location            Delay type         Delay(ns)  Physical Resource
                                                      Logical Resource(s)
    ----------------------------------------------  -------------------
    F15.I               Tiopi                 1.938   Clock
                                                      Clock
                                                      Clock_BUFGP/IBUFG
    BUFGMUX5.I0         net (fanout=1)        2.963   Clock_BUFGP/IBUFG
    BUFGMUX5.O          Tgi0o                 0.001   Clock_BUFGP/BUFG
                                                      Clock_BUFGP/BUFG
    SLICE_X37Y11.CLK    net (fanout=8)        1.011   Clock_BUFGP
    ----------------------------------------------  ---------------------------
    Total                                     5.913ns (1.939ns logic, 3.974ns route)
                                                      (32.8% logic, 67.2% route)
```

Ce qui nous donne bien un total de 9,337 ns. L'analyseur temporel possède une fonctionnalité intéressante pour sélectionner la vitesse du circuit nécessaire à la réalisation du design. Nous allons analyser à nouveau la fréquence de fonctionnement maximale du montage :



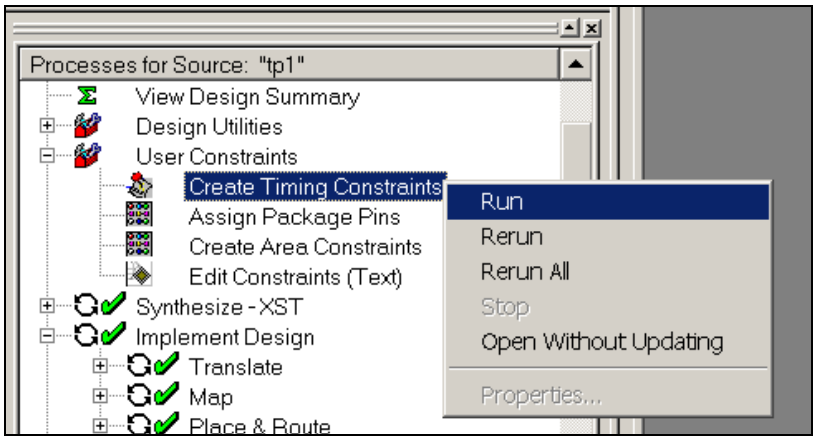
en sélectionnant les flip-flop comme source et destination de l'analyse temporelle à effectuer. Nous obtenons une période égale à 4.453 ns. Si vous voulez savoir à quelle vitesse tournerait le design dans un XC3S200-5, fermez ce rapport, et relancez l'analyse. Dans la fenêtre suivante, cliquez sur l'onglet « Options », sélectionnez une vitesse -5 (Speed Grade) puis cliquez sur OK.



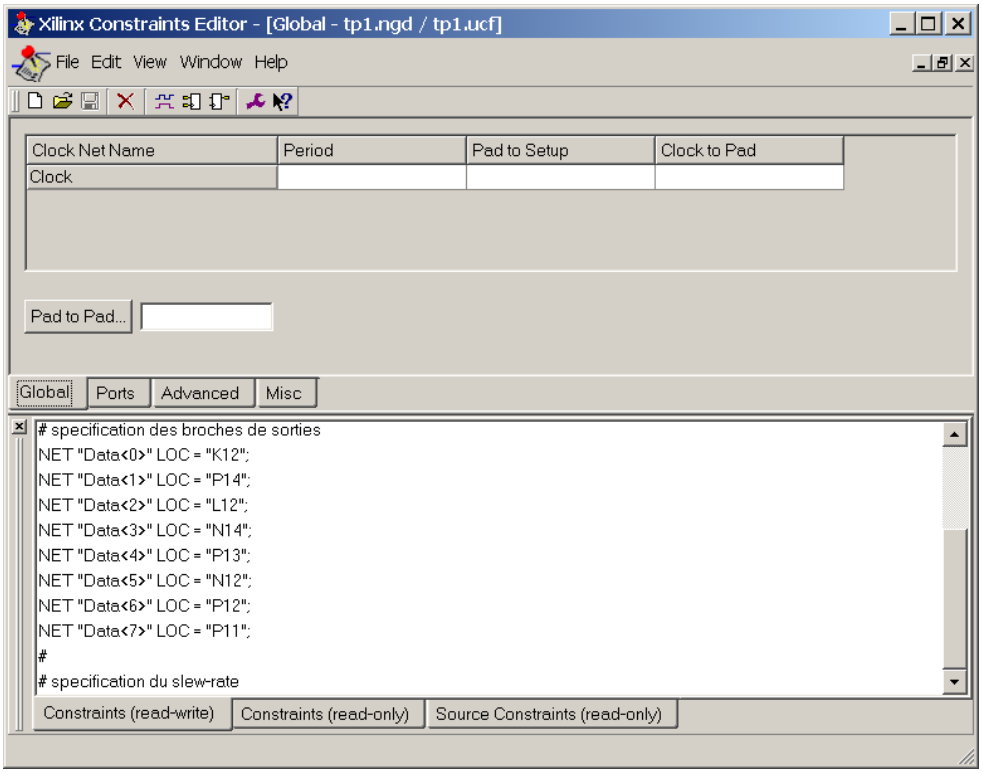
La période minimale vaut maintenant 3,887 ns. Fermez l'analyseur temporel.

7.7.3 Les contraintes

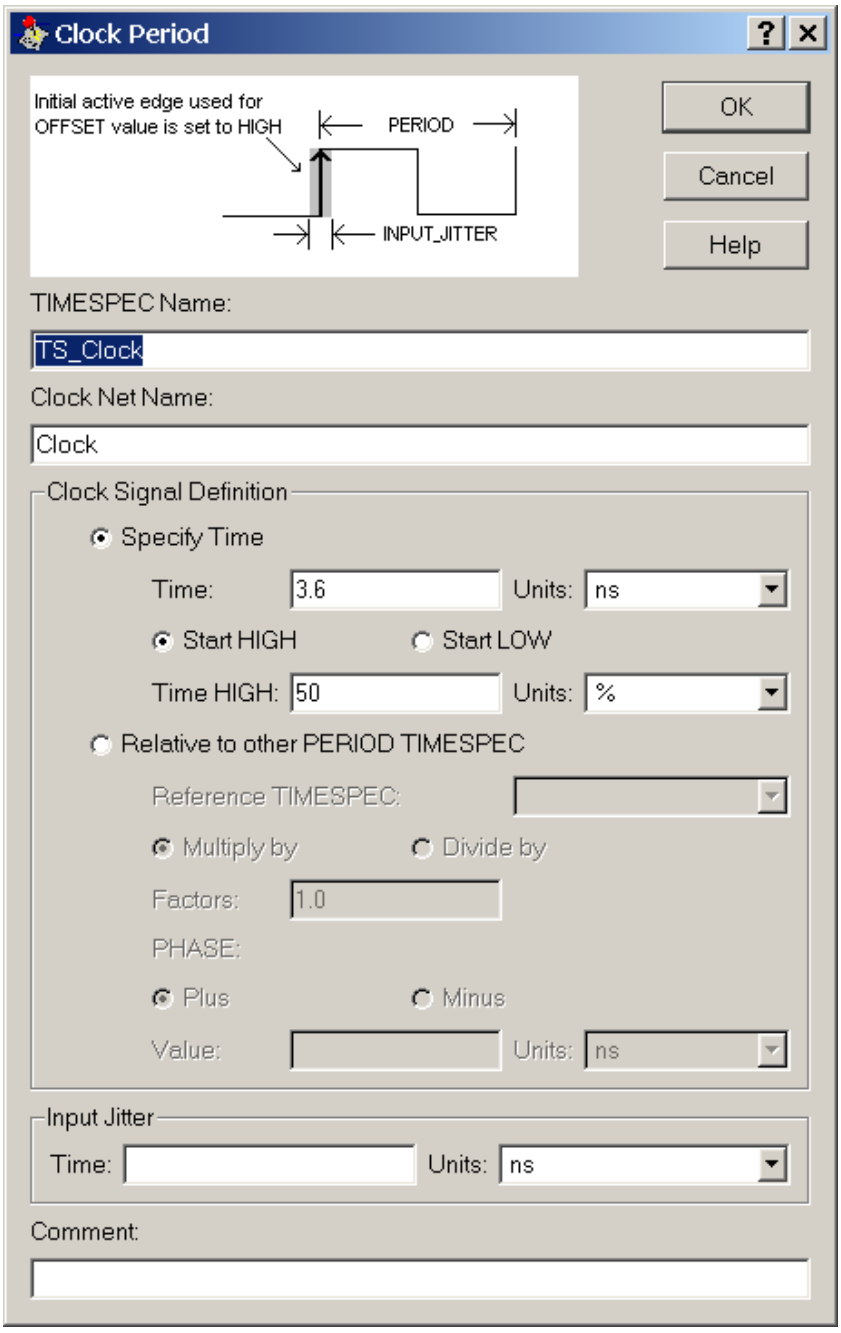
Nous n'avons pour l'instant donné aucune indication à l'outil de placement-routage concernant les performances du design. La manière la plus élémentaire (et a priori une des plus efficaces) de spécifier une fréquence de fonctionnement est la contrainte PERIOD placée dans le fichier de contraintes. Continuons de travailler avec TP1. L'éditeur de contraintes permet de modifier le fichier tp1.ucf. On le lance en sélectionnant tp1.sch dans la fenêtre « Sources » puis en lançant « Create Timing Constraints » dans la fenêtre « Processes » :



La fenêtre suivante apparaît :



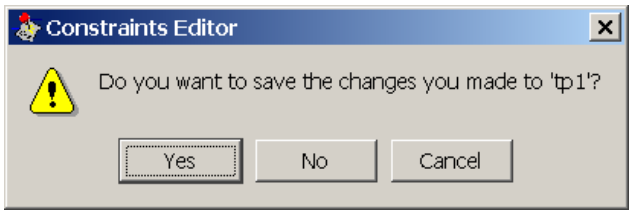
Sur la ligne Clock (onglet « Global » de la fenêtre supérieure), double-cliquez sur la case « Period ». Dans la fenêtre qui s'ouvre, tapez 3.6 dans la case « Time » puis cliquez sur OK.



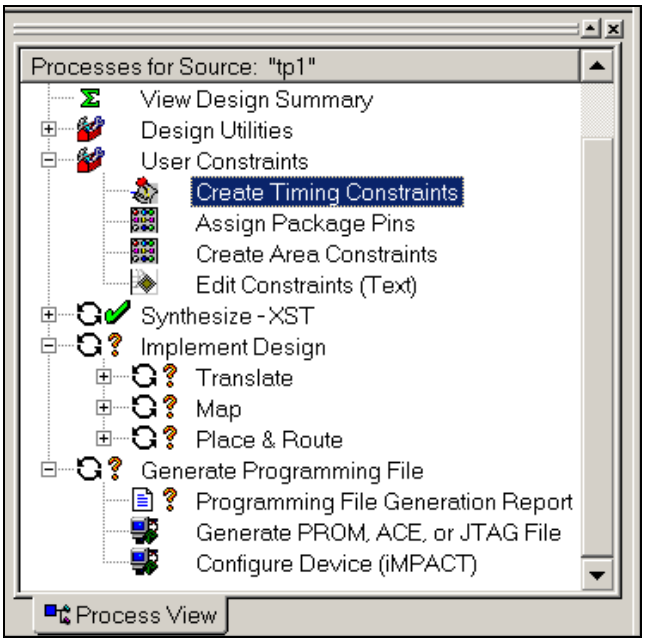
La nouvelle case « Period » doit maintenant ressembler à :

Clock Net Name	**Period**	Pad to Setup	Clock to Pad
Clock	3.6 ns HIGH 50 %		

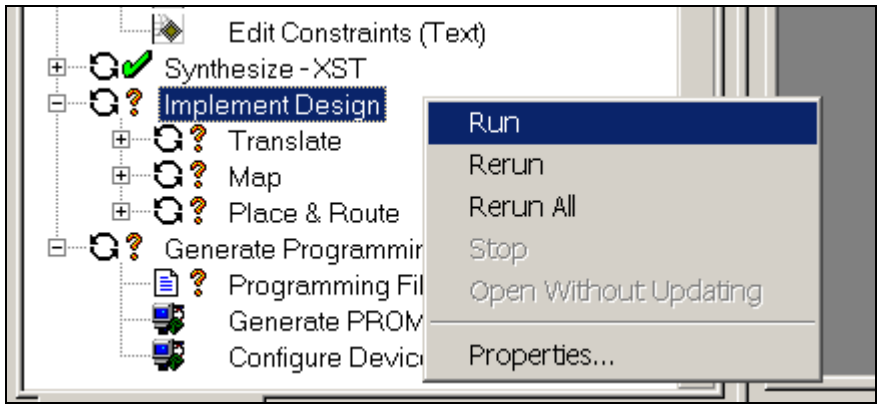
Cliquez sur « File », « Exit » pour sortir de l'éditeur de contraintes. Cliquez sur Yes quand la fenêtre suivante apparaît :



Les marques vertes ont été remplacées par des points d'interrogation dans les différentes phases de l'implémentation.



Il faut la relancer pour prendre en compte la contrainte temporelle sur l'horloge Clock.



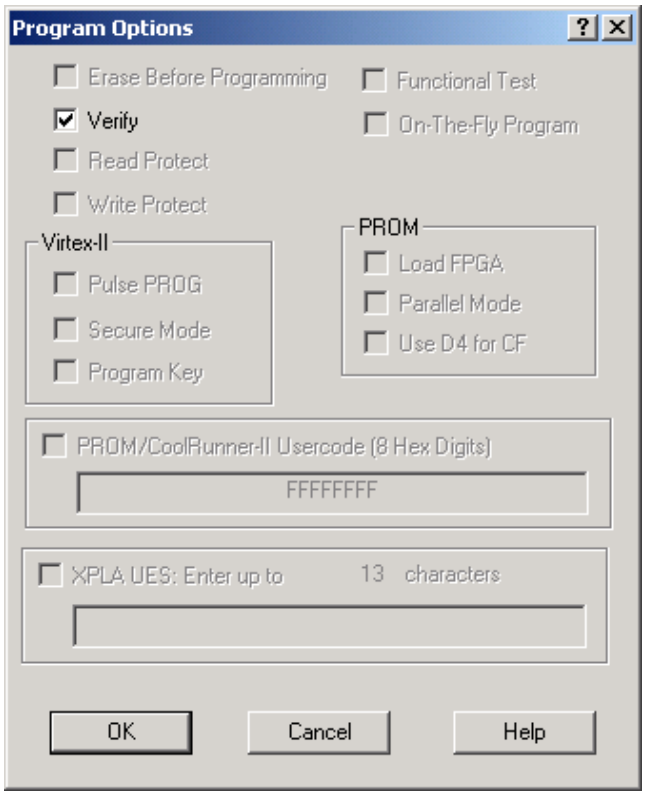
Comparons en détail les timings du chemin critique. Lancez l'analyseur temporel et cliquez sur le bouton  pour réaliser l'analyse de la contrainte sur l'horloge.

	Pas de contrainte	Période = 3.6 ns
T_{CKO}	0,720 ns	0,720 ns
net	0,787 ns	0,725 ns
T_{ILO}	0,608 ns	0,551 ns
net	1,025 ns	0,291 ns
T_{ILO}	0,608 ns	0,608 ns
net	0,015 ns	0,015 ns
T_{FCK}	0,69 ns	0,69 ns
total	4,453 ns	3,600 ns

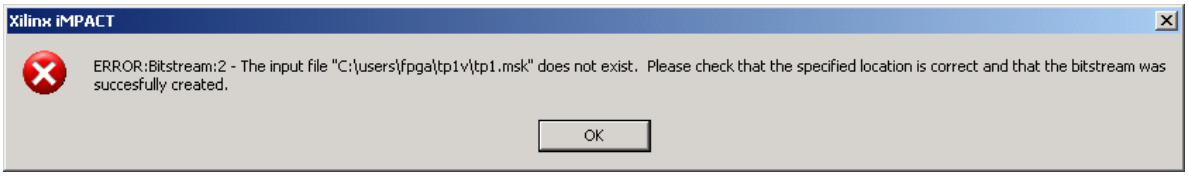
Le tableau précédent vous donne les valeurs des timings avec et sans contraintes. On voit bien que l'outil de placement-routage optimise les délais des interconnexions pour satisfaire la contrainte. Bien entendu, les délais logiques ne sont pas modifiés (sauf le 1^{er} T_{ILO}, mais le fanout du net qui l'attaque est passé de 6 à 3). Il faut noter qu'il y a des différences entre deux essais successifs d'implémentation car le processus de placement-routage est initialisé par une valeur pseudo-aléatoire qui change à chaque essai. Une période égale à 3,6 ns semble être le minimum possible.

7.7.4 Vérification de la programmation du FPGA

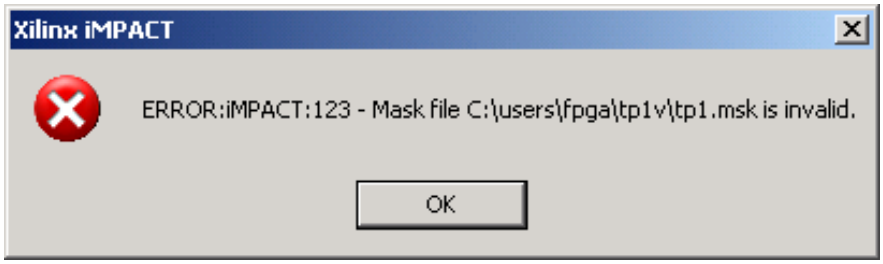
Reprenons le design **TP1V**. Lors de la programmation du FPGA, nous n'avions pas coché la case « Verify ». Si vous essayez maintenant :



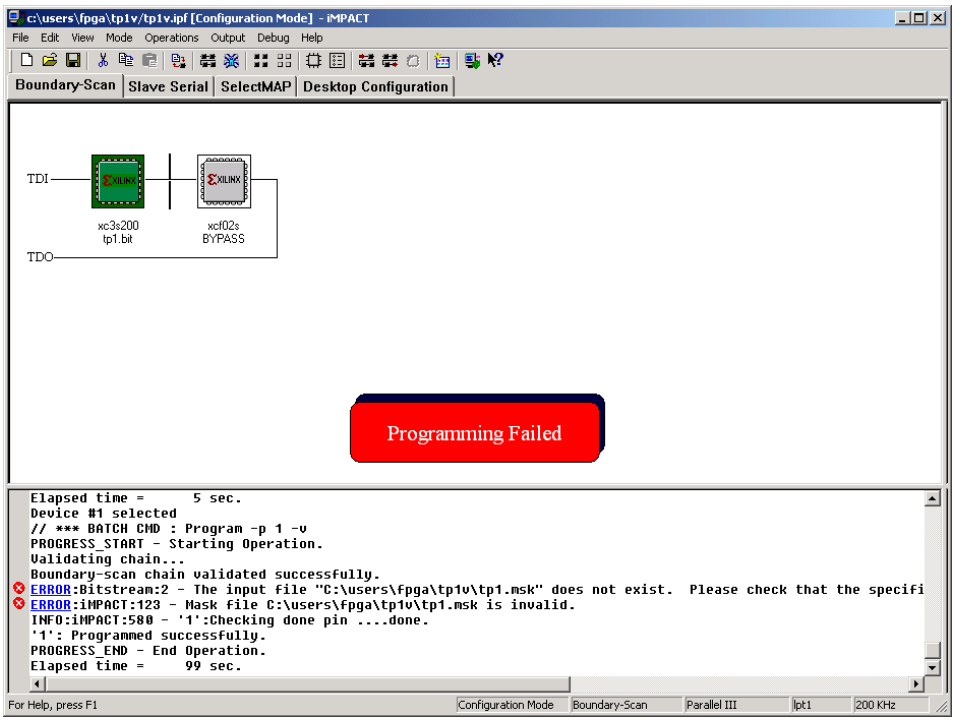
Vous obtenez le message d'erreur suivant :



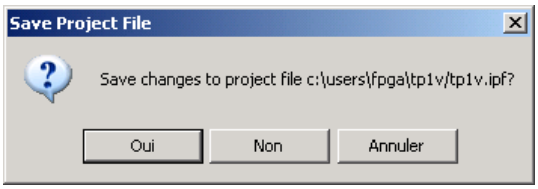
Puis :



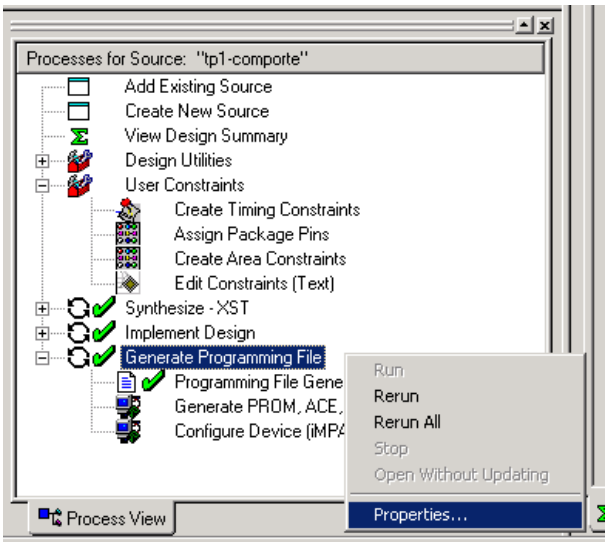
Finalement, la programmation échoue :



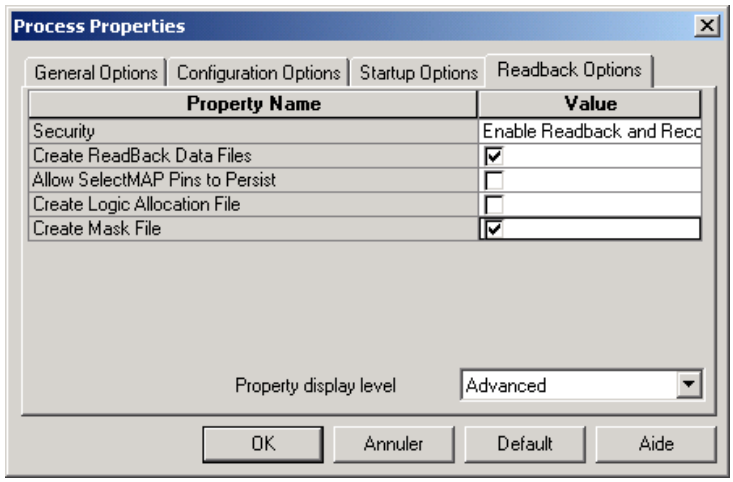
Fermez Impact et cliquez sur Non dans la fenêtre suivante :



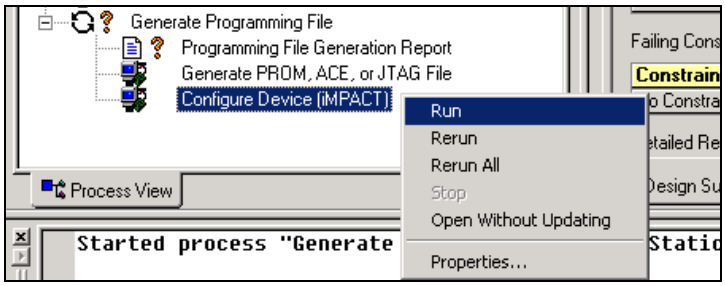
Pour que la vérification soit possible, il faut modifier les propriétés de la génération du fichier de configuration :



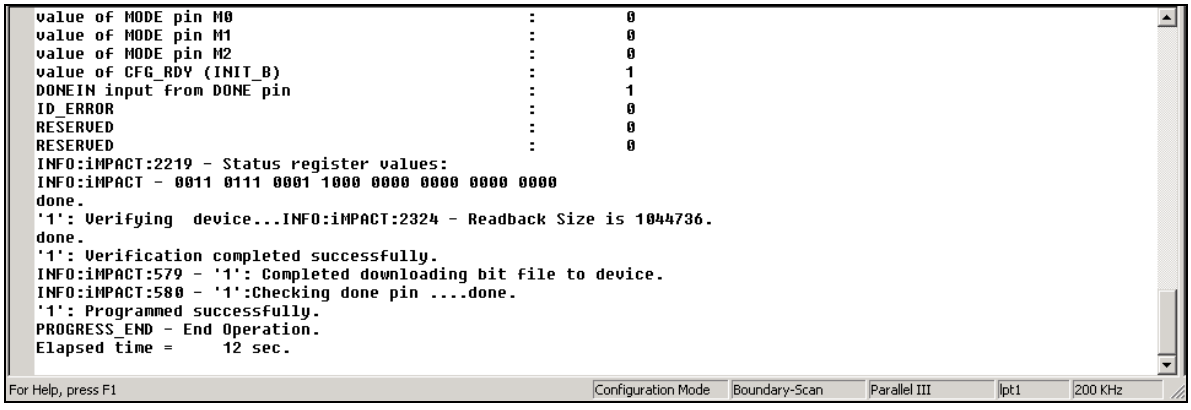
Dans l'onglet « Readback Options », sélectionnez « Create ReadBack Data Files » puis « Create Mask File » et cliquez sur OK :



Relancez Impact :



Vous pouvez maintenant lancer la vérification. Impact configure le FPGA avec le fichier .bit en passant par le JTAG, puis il vérifie la programmation (toujours par le JTAG) en relisant le contenu de la SRAM du FPGA et en comparant ce contenu avec celui du fichier .bit (C'est là qu'il utilise le fichier .msk).



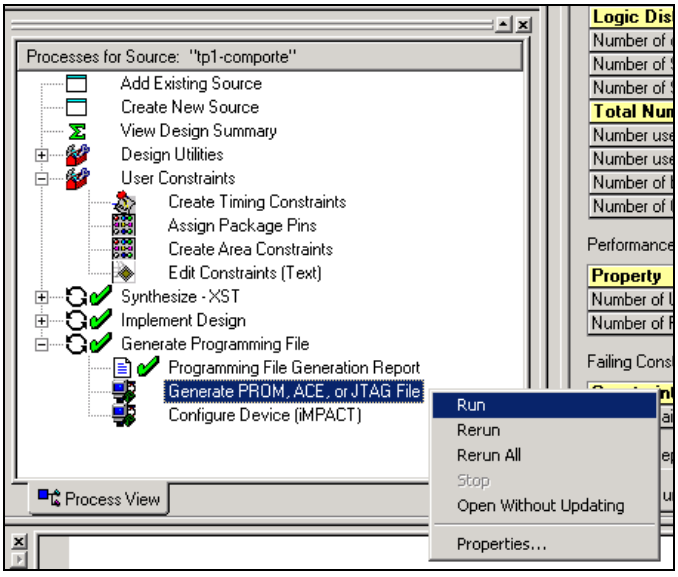
Il arrive parfois que la vérification échoue sans pour cela que la programmation soit forcément mauvaise. La fiabilité du téléchargement via le cordon JTAG n'est pas toujours excellente. Elle dépend des paramètres suivants :

- la qualité du cordon JTAG,
- la vitesse de la programmation,
- l'alimentation du cordon JTAG,
- la qualité du port parallèle du PC,
- la qualité du circuit imprimé de la carte FPGA.

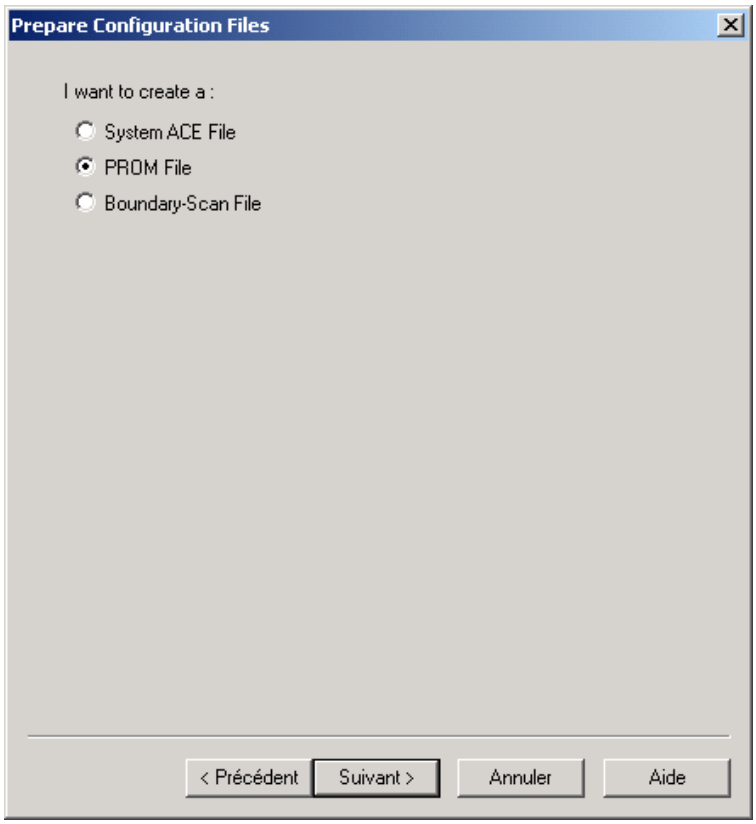
Il y a parfois un aspect un peu « magique » derrière tout cela, certains PC s'obstinant à refuser tout téléchargement fiable à travers le port parallèle (quand il existe encore, notamment sur les portables). Pour assurer une programmation de bonne qualité, il est souvent nécessaire de programmer la mémoire Flash série sur la carte plutôt que le FPGA directement. La fiabilité est en général bien meilleure avec la Flash qu'avec le FPGA.

7.7.5 Programmation de la mémoire Flash série

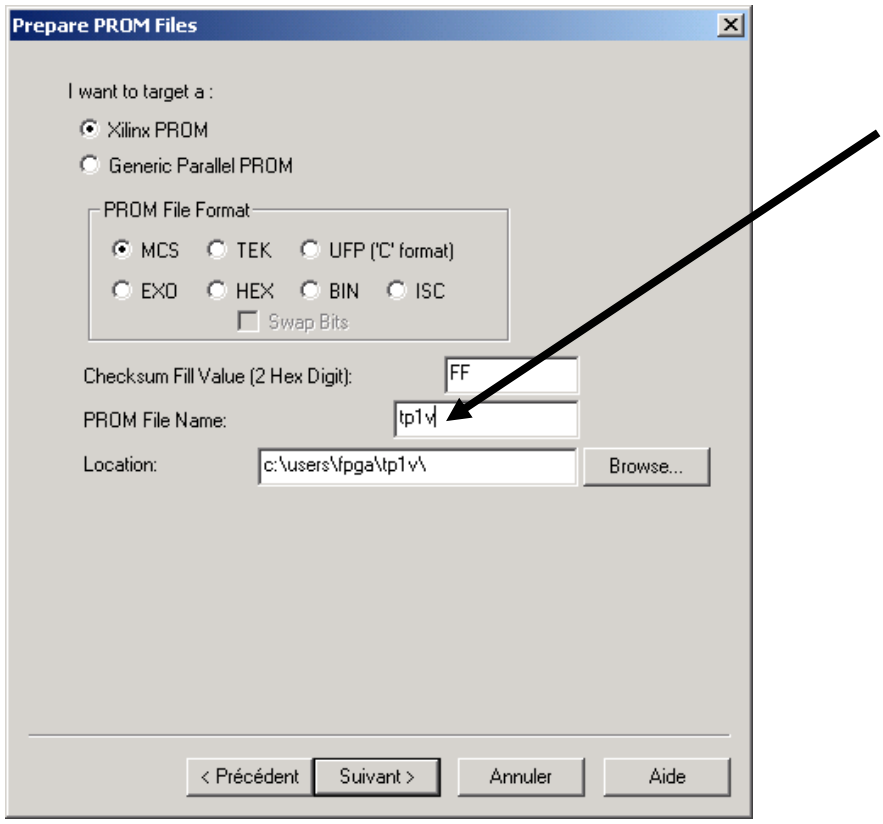
Continuons avec le design TP1V. La programmation de la Flash s'effectue avec un fichier au format MCS-86. Pour le créer, il faut lancer :



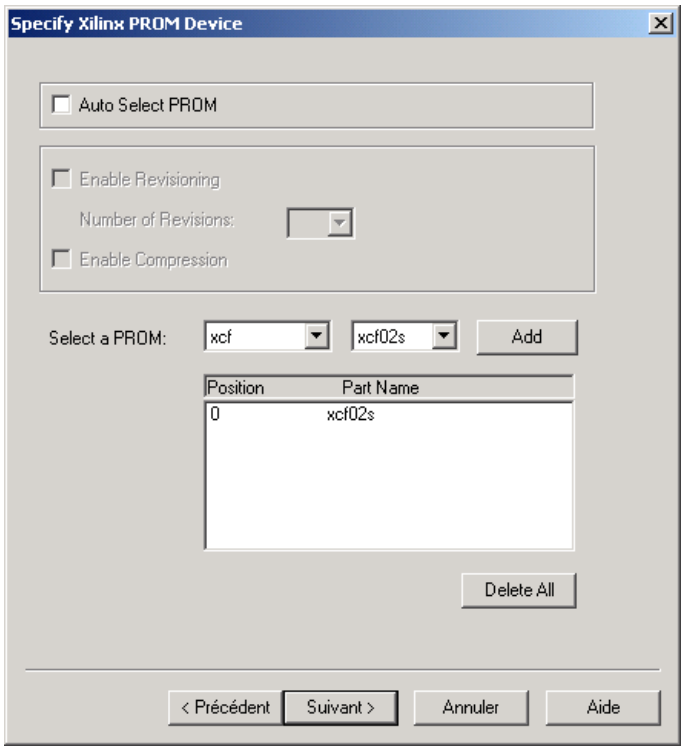
Dans la fenêtre qui s'ouvre (il s'agit toujours d'Impact), sélectionnez « PROM File », puis cliquez sur « Suivant » :



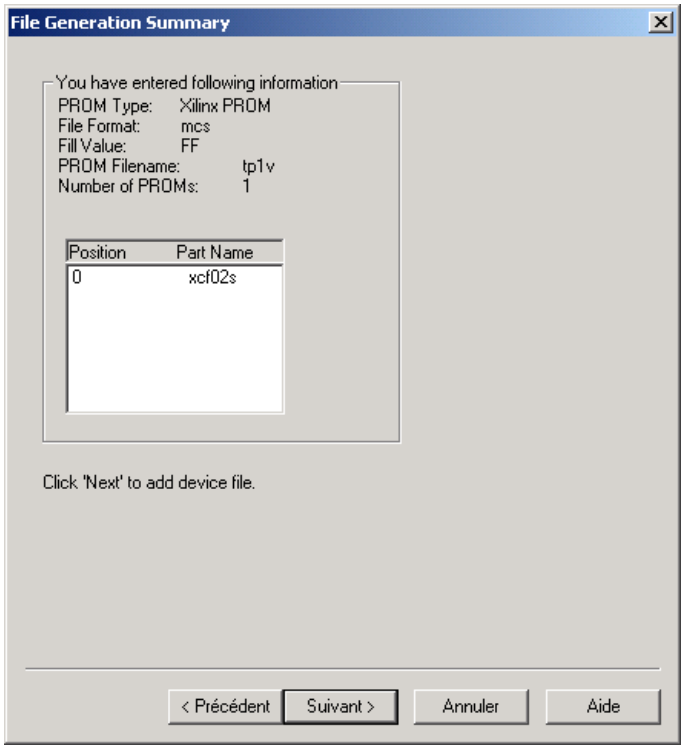
Tapez `tp1v` dans le champ « PROM File Name », puis cliquez sur « Suivant » :



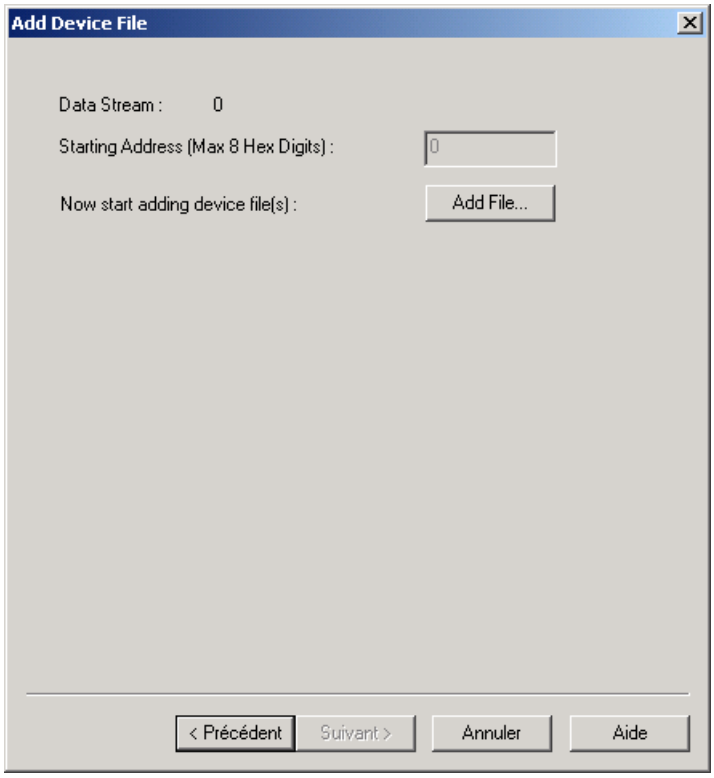
Sélectionnez xcf02s puis cliquez sur le bouton « Add ». Vous devez obtenir cette fenêtre avant de cliquer sur « Suivant » :



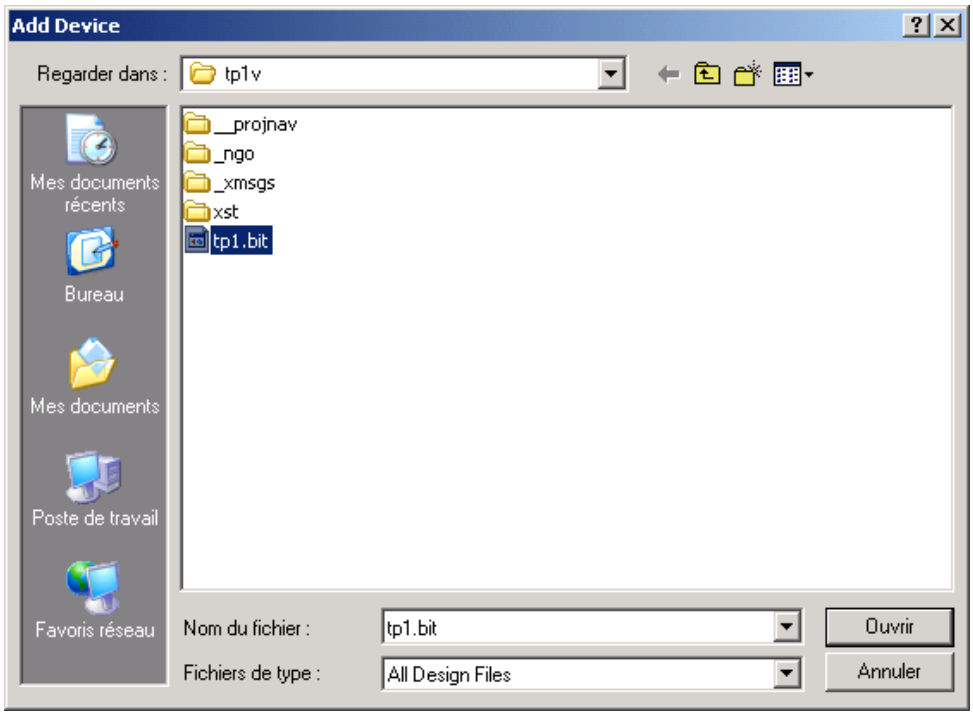
Vérifiez sur la page de résumé que tout est correct avant de cliquer sur « Suivant » :



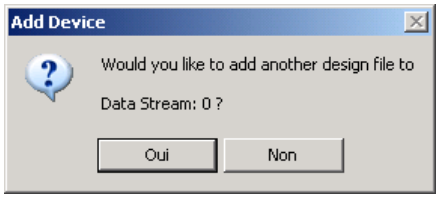
Dans la fenêtre qui s'ouvre alors, cliquez sur « Add File… » :



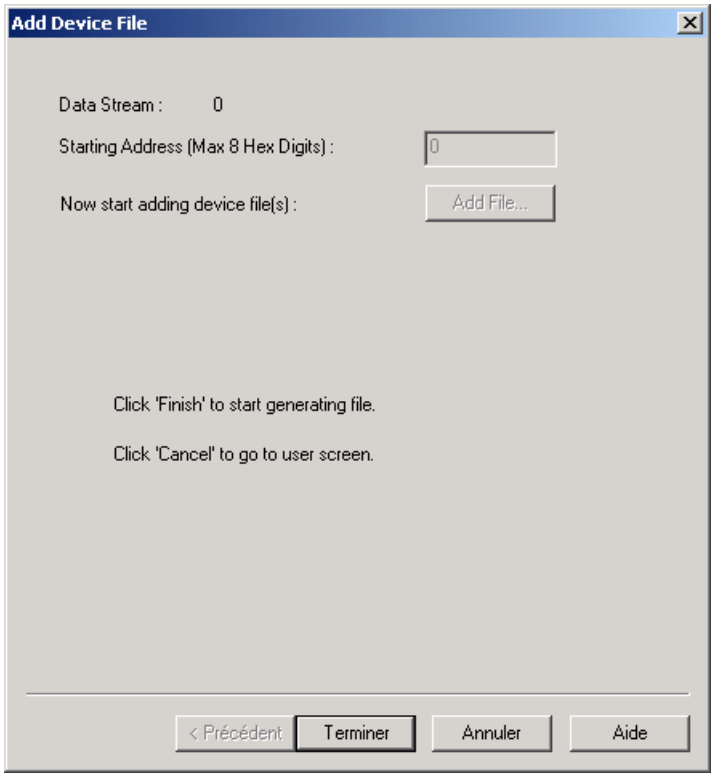
Puis sélectionnez le fichier tp1.bit dans le répertoire c:\users\fpga\tp1v et cliquez sur « Ouvrir » :



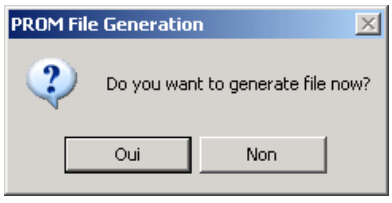
Dans la fenêtre qui s'ouvre, cliquez sur « Non » :



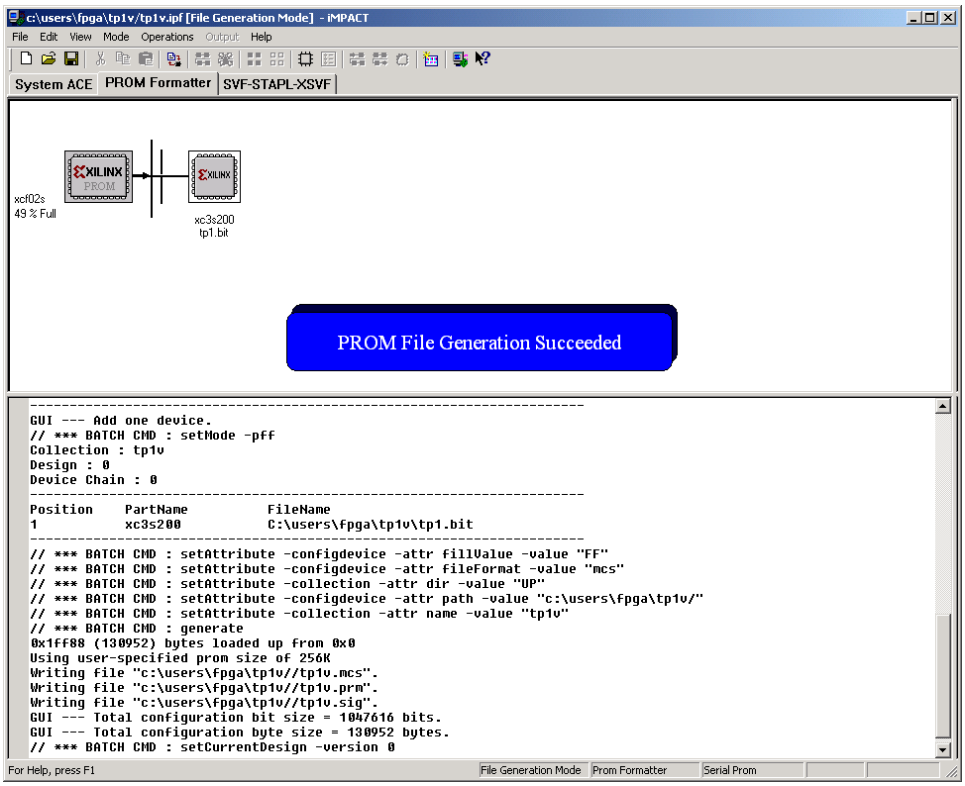
En cliquant sur « Terminer » :



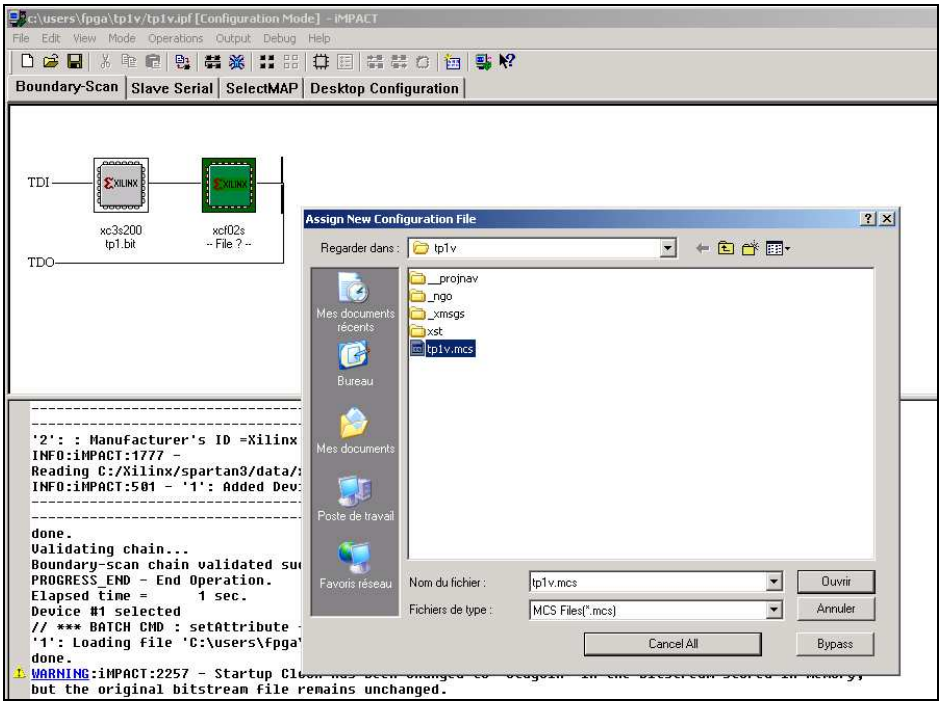
Puis sur « Oui » :



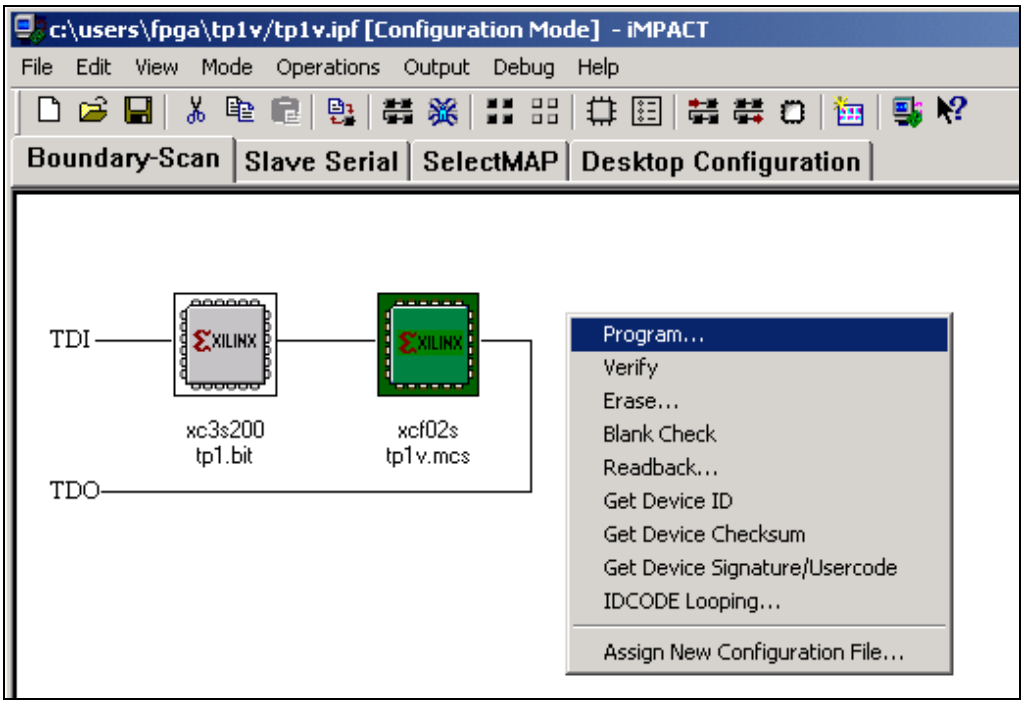
Le fichier MCS est généré. Attention, la fenêtre d'Impact a tendance à passer sous celle d'ISE. Si cela se produit, remettez-la au premier plan.



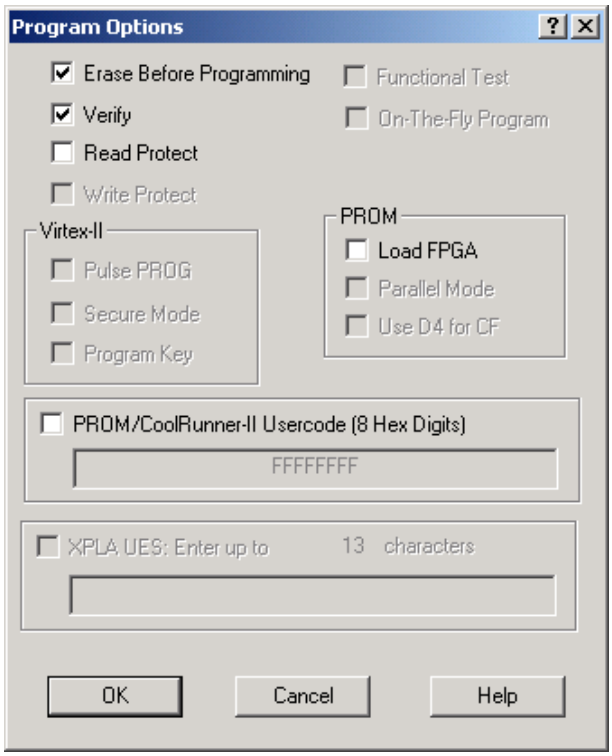
Vous noterez la facilité et le coté intuitif de la procédure. Bravo Xilinx, c'est du grand art (ou pourquoi faire simple quand on peut faire compliqué). Le fichier MCS étant créé, nous pouvons relancer la configuration. Nous avons maintenant un fichier tp1v.mcs à associer à la PROM :



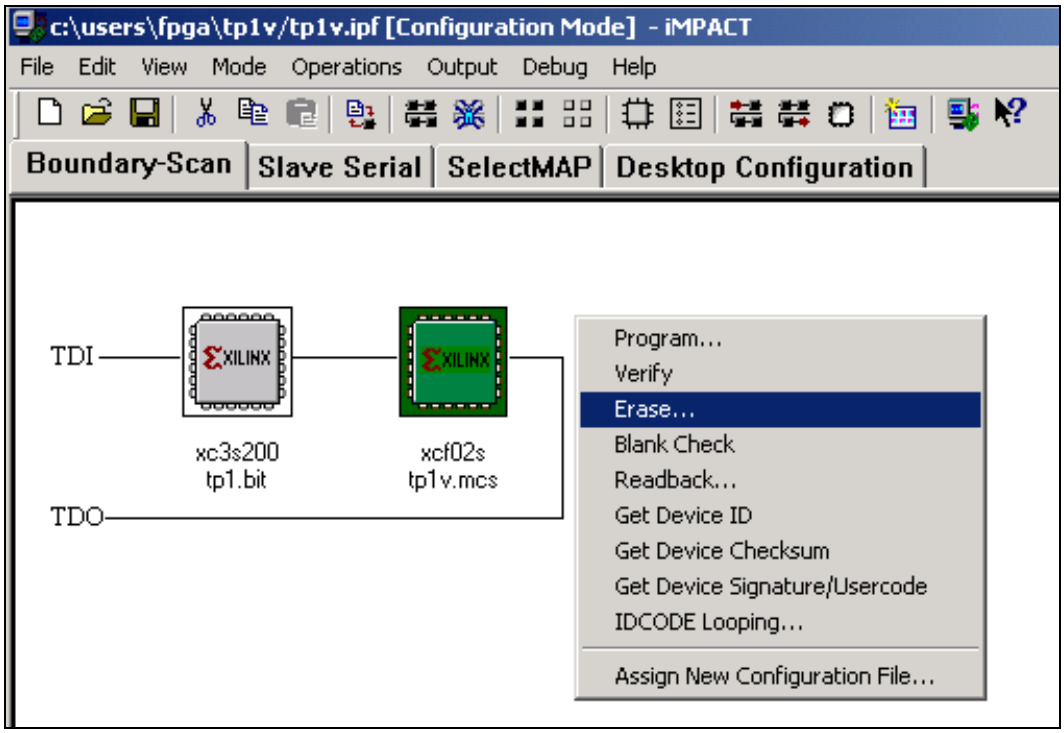
Nous pouvons maintenant programmer la mémoire Flash en la sélectionnant et en cliquant sur le bouton droit de la souris :



Nous allons effacer la PROM, la programmer puis vérifier sa programmation :



L'opération dure environ 40 secondes. Si vous éteignez puis rallumez la carte, vous voyez que le design est bien rechargé dans le FPGA. Si maintenant vous effacez la PROM :

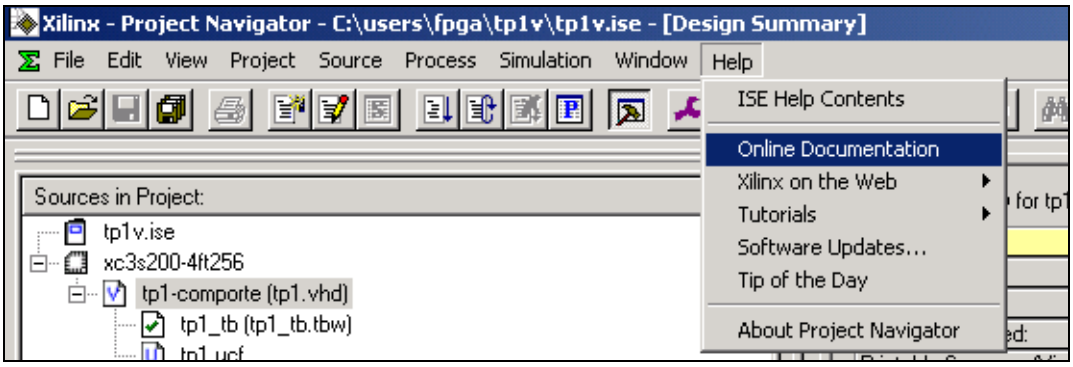


Vous constaterez en éteignant puis en rallumant la carte qu'aucun design n'est plus chargé dans le FPGA. **Effacez la PROM avant de passer à la suite du TP.**

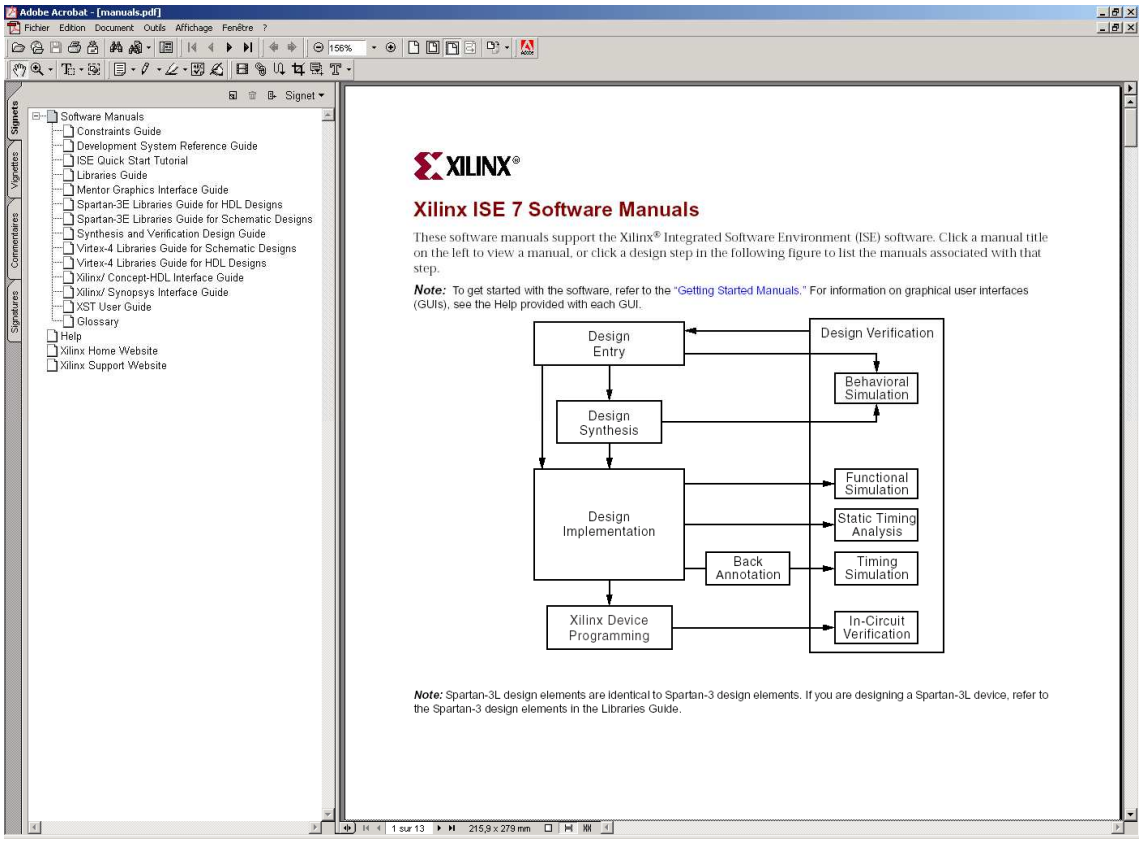
7.7.6 Accès à l'information

Il est très important de bien connaître les différents emplacements où se trouve l'information. L'accès à l'information est vital pour le concepteur de circuit intégré. Il y a deux grandes sources d'informations :

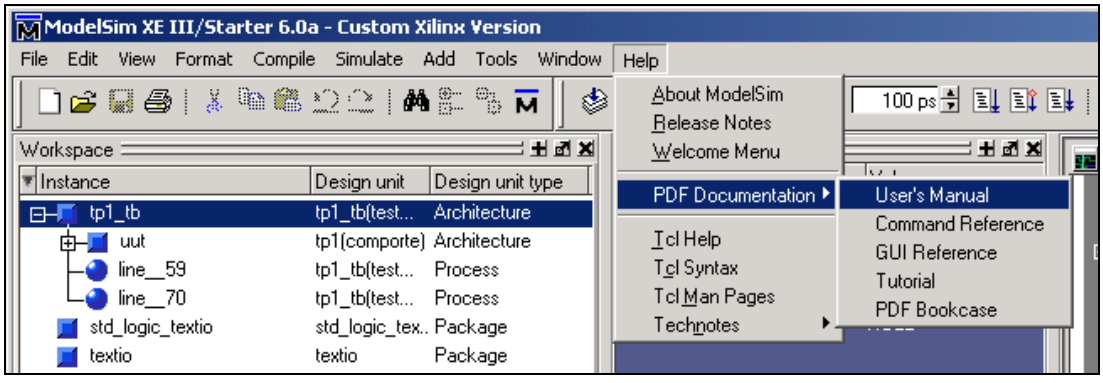
- Première source : la documentation installée avec ISE qui doit être consultée en premier.



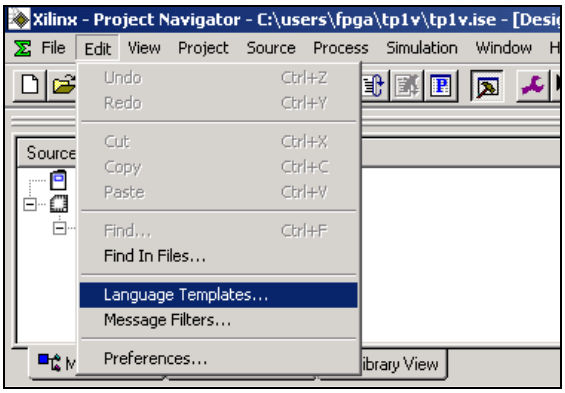
Elle est au format PDF et décrit de manière complète le fonctionnement des outils ainsi que les différentes étapes du développement.



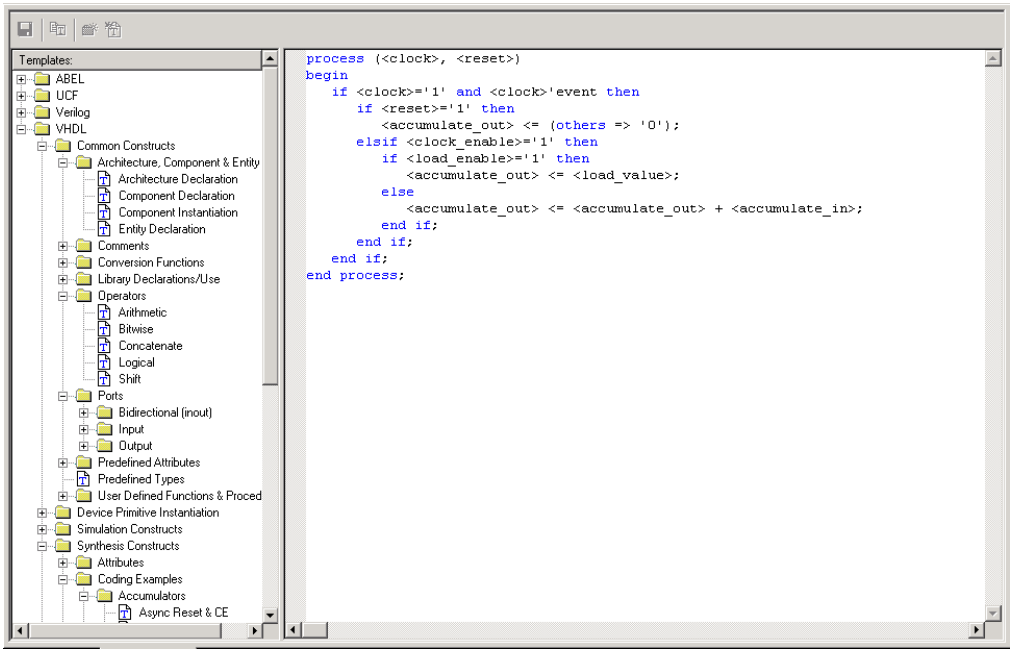
Concernant VHDL, il y a aussi la documentation interne de ModelSim qui est très complète :



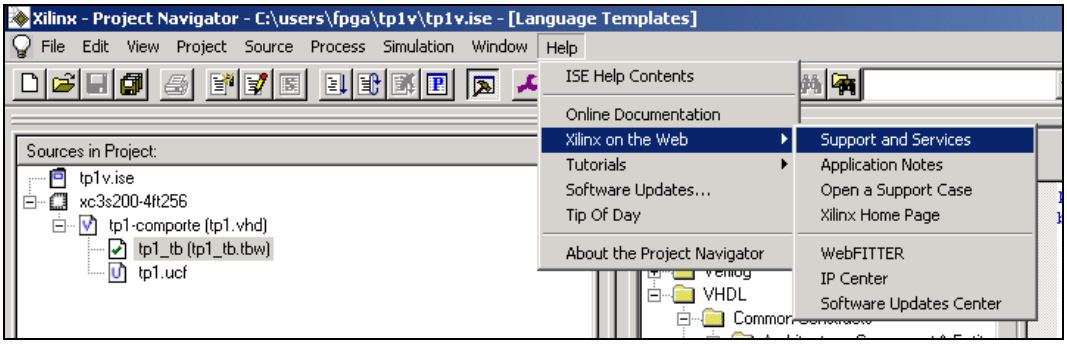
Concernant la syntaxe et les possibilités des différents langages utilisés, vous avez accès à un assistant dans ISE :



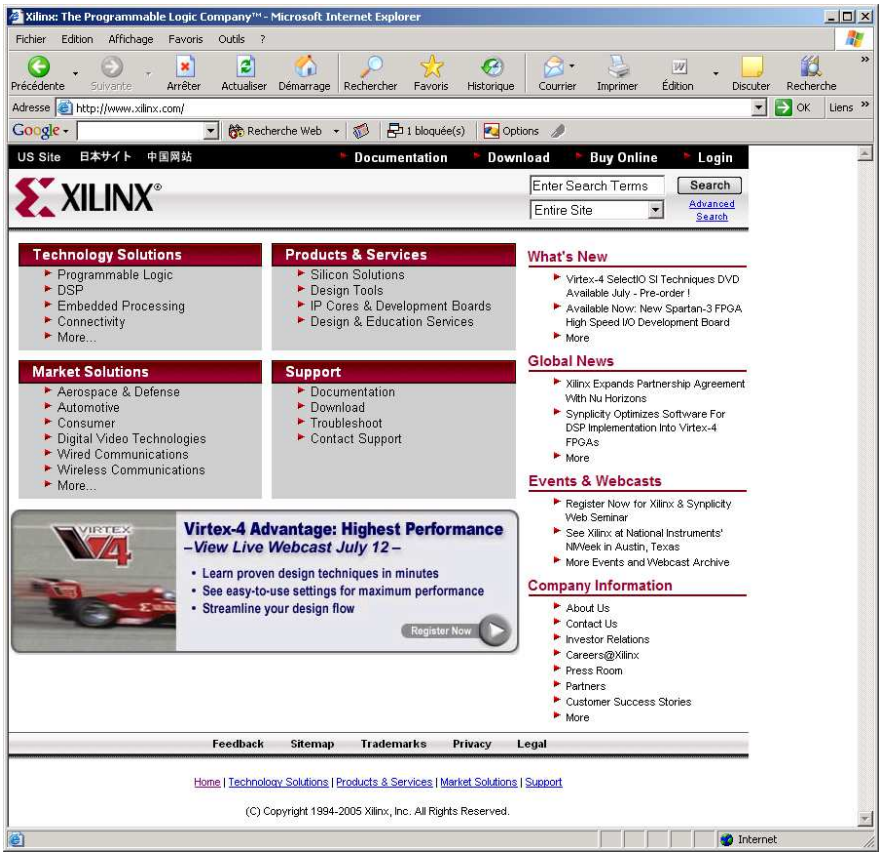
Cet assistant est un aide-mémoire très puissant concernant l'écriture des fichiers UCF, VHDL ou Verilog.



- Deuxième source de documentation : Internet. Le site de Xilinx est accessible à travers ISE.



Ou bien de préférence directement sur le Web (www.xilinx.com).



On trouve sur ce site :

- Les datasheets des composants,
- Des notes d'application,
- La documentation des outils,
- Les mises à jour des logiciels,
- Une base de connaissance pour faire des recherches en cas de problème,
- Un service de support en ligne (webcase),
- De forums de discussion internes à Xilinx,
- Et bien d'autres choses encore.

Il existe aussi des forums de discussion publics spécialisés dans les FPGA (comp.arch.fpga) et dans VHDL (comp.lang.vhdl) qui sont de véritables mines d'informations.

7.8 Projet

7.8.1 Cahier des charges

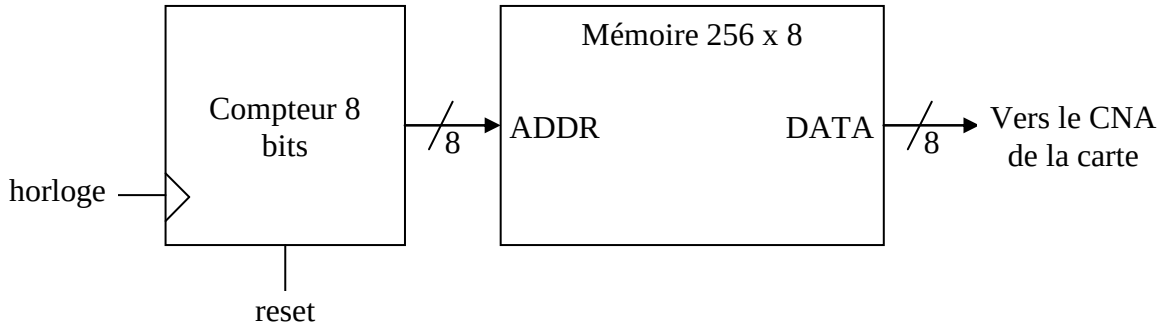
Le but de ce projet est de générer un signal sinusoïdal analogique en essayant différentes méthodes. Les 5 premiers montages utiliseront le convertisseur numérique analogique (CNA) de la maquette avec une fréquence d'échantillonnage égale à 50 MHz. Le dernier montage utilisera un CNA interne au FPGA. Les montages sont les suivants :

1. Un compteur associé à une mémoire contenant une période de signal sinusoïdal. La fréquence est fixe.
2. Même montage avec une entrée sur 8 bits pour générer une fréquence variable.
3. Synthèse directe de fréquence en VHDL.
4. Synthèse directe de fréquence avec un IP Coregen.
5. Filtre à réponse impulsionnelle infinie ayant un 0 au dénominateur de sa fonction de transfert.
6. On reprendra le montage 2, mais la sortie se fera avec un CNA sigma delta et un filtre RC externe. On n'utilisera pas le CNA de la maquette.

La logique fonctionnera sur front montant avec un reset asynchrone. Vous allez créer au fur et à mesure 6 nouveaux projets, SIN_1, SIN_2, SIN_3, SIN_4, SIN_5 et SIN_6 et saisir votre design. Vous écrirez un testbench pour faire la simulation fonctionnelle (**pas de synthèse sans simulation**), puis vous implémenterez votre design dans le FPGA. Vous connecterez Deltax (si nécessaire) sur les dip switches SW0 à SW7, l'horloge sur H50, et le Reset sur le bouton poussoir BTN3.

7.8.2 Montage n°1

La solution la plus simple pour générer un signal sinusoïdal consiste à utiliser le principe suivant :



La mémoire contient une période de la fonction `sinus`(addr*2*π/256) avec addr, bus d'adresse de la mémoire, variant de 0 à 255. Voici un exemple de programme en C permettant de générer les valeurs de la mémoire :

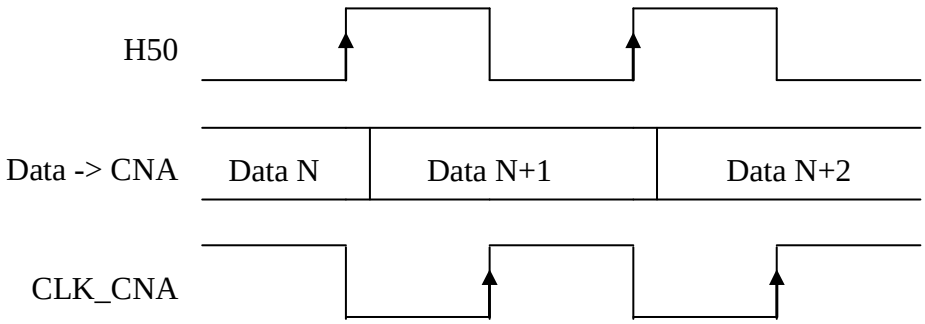
```
#include<stdio.h>
#include <math.h>

int main()
{
      double x,y;
      int i, tmp;
      FILE *filetab;
      char *tabfilename=".\\mem_sin.txt";

      filetab=fopen(tabfilename,"w");


      for (i = 0; i < 256; i++) {
            x = 2.0*3.1415927*i / 256.0;
            y = sin(x);
            tmp = (int)(y*128.0);

            fprintf(filetab, "(%4d), ", tmp);
            if ((i+1)%16 == 0)
                  fprintf(filetab, "\n");
      }
}
```

Vous devrez compléter le code VHDL suivant pour implémenter ce premier design. Le fichier sin_1_vide.vhd se trouve dans c:\users\fpga\fichiers.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity sin_1 is
  port( H50_I : in std_logic;
        Reset : in std_logic;
        CLK_CNA : out std_logic;
        data_cna : out std_logic_vector(7 downto 0));
end sin_1;

architecture a1 of sin_1 is
      TYPE mem_data IS ARRAY (0 TO 255) OF integer range -128 to 127;
      constant sin : mem_data := (
      (  0),(  3),(  6),(  9),( 12),( 15),( 18),( 21),( 24),( 28),( 31),( 34),( 37),( 40),( 43),( 46),
      ( 48),( 51),( 54),( 57),( 60),( 63),( 65),( 68),( 71),( 73),( 76),( 78),( 81),( 83),( 85),( 88),
      ( 90),( 92),( 94),( 96),( 98),( 100),( 102),( 104),( 106),( 108),( 109),( 111),( 112),( 114),( 115),( 117),
      ( 118),( 119),( 120),( 121),( 122),( 123),( 124),( 124),( 125),( 126),( 126),( 127),( 127),( 127),( 127),( 127),
      ( 127),( 127),( 127),( 127),( 127),( 127),( 126),( 126),( 125),( 124),( 124),( 123),( 122),( 121),( 120),( 119),
      ( 118),( 117),( 115),( 114),( 112),( 111),( 109),( 108),( 106),( 104),( 102),( 100),(  98),(  96),(  94),(  92),
      (  90),(  88),(  85),(  83),(  81),(  78),(  76),(  73),(  71),(  68),(  65),(  63),(  60),(  57),(  54),(  51),
      (  48),(  46),(  43),(  40),(  37),(  34),(  31),(  28),(  24),(  21),(  18),(  15),(  12),(   9),(   6),(   3),
      (   0),(  -3),(  -6),(  -9),( -12),( -15),( -18),( -21),( -24),( -28),( -31),( -34),( -37),( -40),( -43),( -46),
```

```
        ( -48),( -51),( -54),( -57),( -60),( -63),( -65),( -68),( -71),( -73),( -76),( -78),( -81),( -83),( -85),( -88),
        ( -90),( -92),( -94),( -96),( -98),(-100),(-102),(-104),(-106),(-108),(-109),(-111),(-112),(-114),(-115),(-117),
        (-118),(-119),(-120),(-121),(-122),(-123),(-124),(-124),(-125),(-126),(-126),(-127),(-127),(-127),(-127),(-127),
        (-127),(-127),(-127),(-127),(-127),(-127),(-126),(-126),(-125),(-124),(-124),(-123),(-122),(-121),(-120),(-119),
        (-118),(-117),(-115),(-114),(-112),(-111),(-109),(-108),(-106),(-104),(-102),(-100),( -98),( -96),( -94),( -92),
        ( -90),( -88),( -85),( -83),( -81),( -78),( -76),( -73),( -71),( -68),( -65),( -63),( -60),( -57),( -54),( -51),
        ( -48),( -46),( -43),( -40),( -37),( -34),( -31),( -28),( -24),( -21),( -18),( -15),( -12),(  -9),(  -6),(  -3));

        …
begin

        CLK_CNA <= not H50_I;
        data_cna <= not data(7) & data(6 downto 0);

        process(H50_I, Reset) begin
                if (Reset='1') then
                …
                elsif (H50_I'event and H50_I='1') then
                …
                end if;
        end process;
end;
```

CLK_CNA est le signal qui va servir d'horloge pour le CNA. Comme le CNA lit les données sur le front montant de l'horloge et que le FPGA travaille aussi sur front montant, on inverse H50 avant de la copier sur CLK_CNA afin que le CNA lise les données issues du FPGA à un moment où elles sont stables.



Comme la mémoire contient des valeurs signées codées en complément à 2 (CA2) et que le CNA travaille sur des données non-signées (codage en binaire naturel), il faut inverser le bit de poids fort du bus de données de la mémoire avant de l'envoyer vers le CNA.

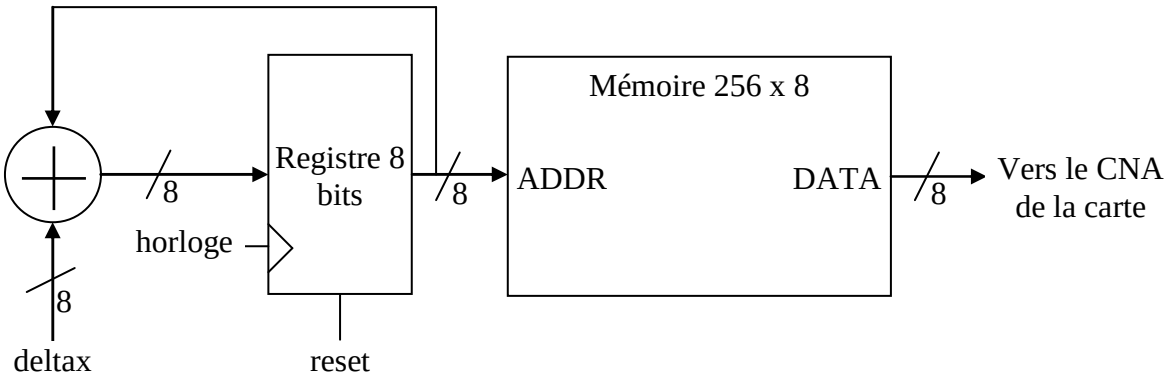
Le compteur étant incrémenté de 1 à chaque front d'horloge, il faut 256 périodes de H50 pour lire une période complète. Donc, la fréquence du signal sinusoïdal est égale à :

$$Fsin = H50 / 256 = 195312.5 \text{ Hz}$$

7.8.3 Montage n°2

L'inconvénient du montage précédent est que la fréquence du signal sinusoïdal est fixe. Une amélioration simple consiste à incrémenter le compteur par pas égal à Δx au lieu d'utiliser un pas de 1. On obtient alors le montage suivant dont la fréquence de sortie est égale à :

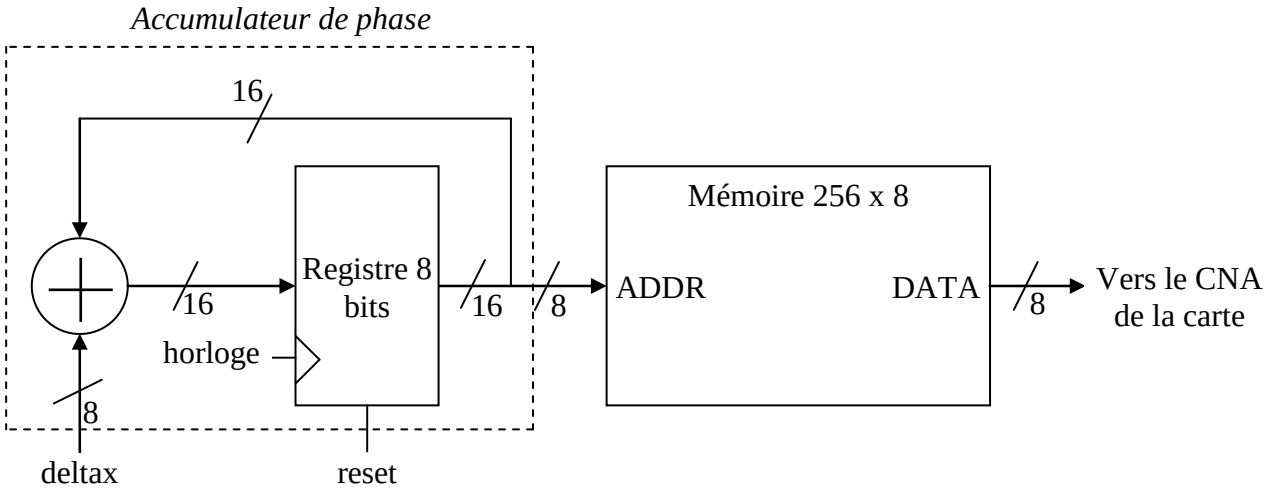
$$Fsin = H50 . \Delta x / 256$$



Modifiez le design sin_1 en rajoutant une entrée `deltax` sur 8 bits (nouveau design sin_2). Réglez les dip switches SW0 à SW7 pour obtenir `deltax` = 16 et vérifiez la fréquence obtenue. Que se passe-t-il quand `deltax` = 128 ? C'est impossible à observer à cause du filtre passe bas en sortie de la maquette, par contre on peut le simuler.

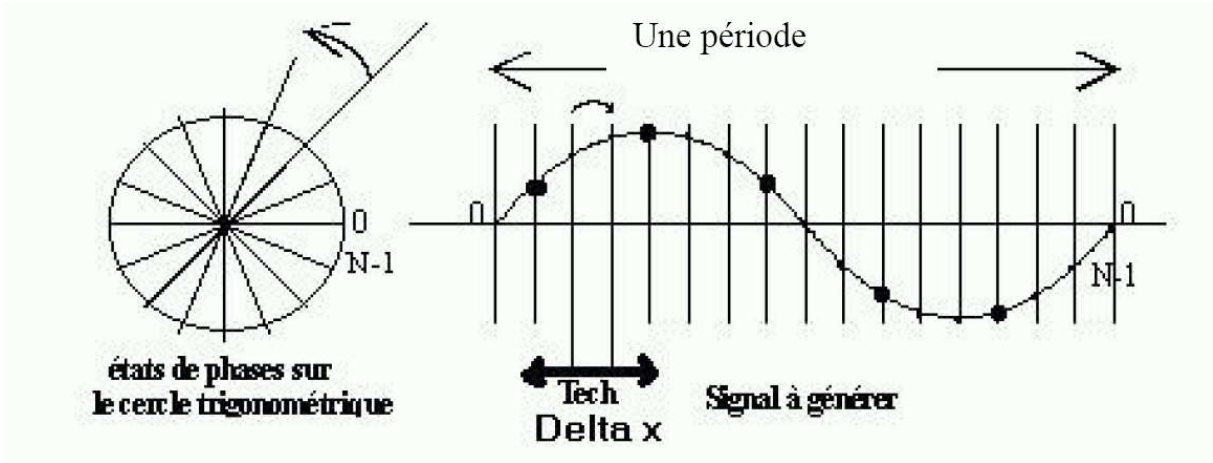
7.8.4 Montage n°3

Une simple modification du design précédent permet de changer profondément le principe de fonctionnement du design. Le compteur passe sur 16 bits et le bus d'adresse de la mémoire est branché sur les 8 bits de poids fort. C'est la **synthèse directe de fréquence**.



Principe de fonctionnement :

Soit une période d'un signal quelconque (ici sinusoïdal) que l'on peut décrire par une succession d'états de phases. On peut exprimer ces phases par un nombre entier x, variant de 0 à N-1, N étant le nombre maximum d'états possibles et pouvant être à priori très grand.



A partir d'une phase quelconque, (entre 0 et N-1) on passe à la suivante par $x = x + \Delta x$ (modulo N). On obtient donc $\frac{N}{\Delta x}$ états par période (sur la figure : 16/3 = 5,33). Si les échantillons sont fournis à la cadence Tech = 1 / Fech, la période du signal obtenu est :

$$T = N \,.\, Tech / \Delta x$$

Le théorème de Shannon doit être respecté. On génère ainsi toute une série de fréquences F, comprises entre 0 et presque Fech/2.

La fréquence du signal de sortie est égale à F = Δx . Fech / N avec une résolution de Fech / N. Plus N est grand, meilleure est la résolution en fréquence (Il n'y a théoriquement pas de limite).

Dans notre design, N = 2^{16} et Fech = 50 MHz. La résolution sera donc égale à 762,94 Hz. Comme deltax n'est codé que sur 8 bits, on pourra obtenir les fréquences suivantes :

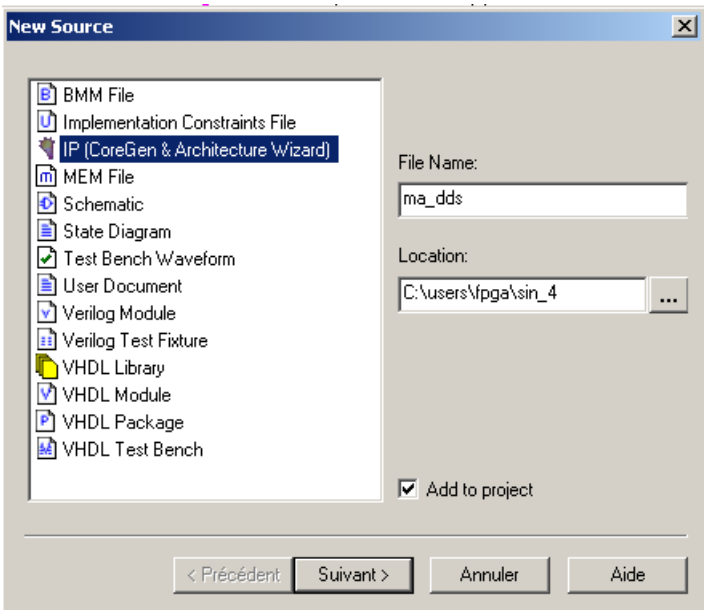
Deltax	Fsin [Hz]
1	762,94
2	1525,88
3	2288,82
…	…
255	194549,56

Modifiez le design sin_2 pour réaliser la synthèse directe de fréquence (nouveau design sin_3). Réglez les dip switches SW0 à SW7 pour faire varier `deltax` et vérifiez les fréquences obtenues.

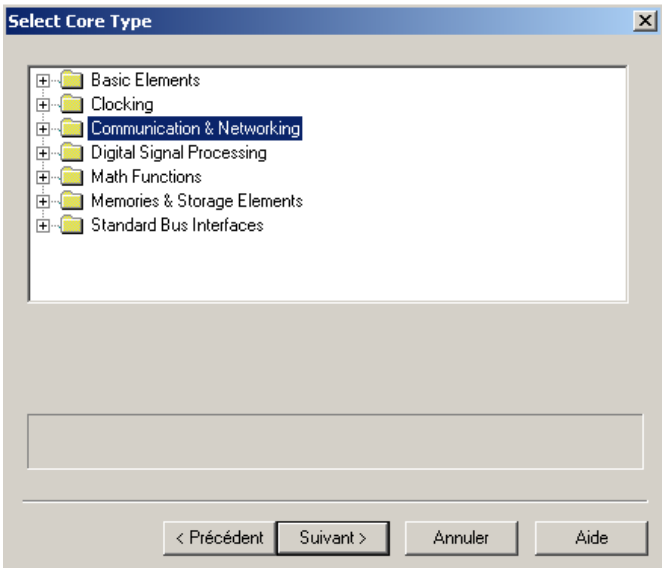
7.8.5 Montage n°4

Pour améliorer la productivité des concepteurs, les fabricants de FPGA (et d'ASICs) fournissent des composants paramétrables prêts à l'emploi. On les appelle des IP pour Intellectual Property core (en français, des cœurs de propriété intellectuelle). De très nombreux paramétrages sont disponibles pour les configurer. Certains IP sont payants, mais la plupart sont fournis gratuitement (comme la DDS).

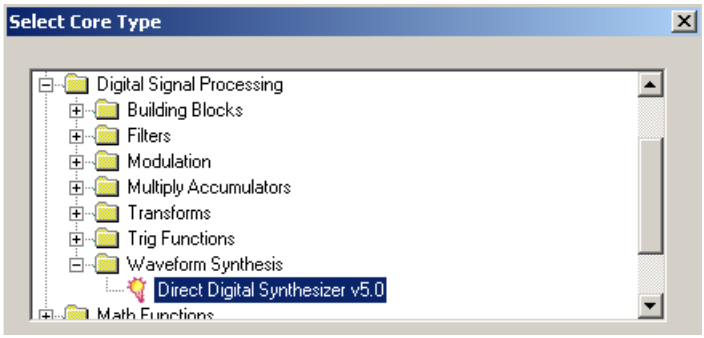
Modifiez le design sin_3 pour réaliser la synthèse directe de fréquence avec un IP (nouveau design sin_4). Pour insérer un IP DDS dans votre nouveau design, ajoutez une nouvelle source (menu project, new source). Dans la fenêtre qui s'ouvre, sélectionnez IP et tapez le nom ma_dds :



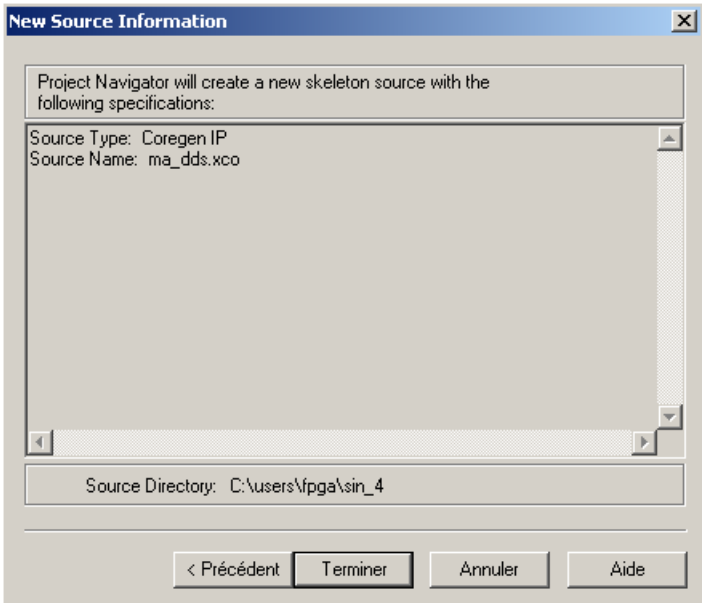
Cliquez sur suivant. La fenêtre de sélection du type d'IP s'ouvre :



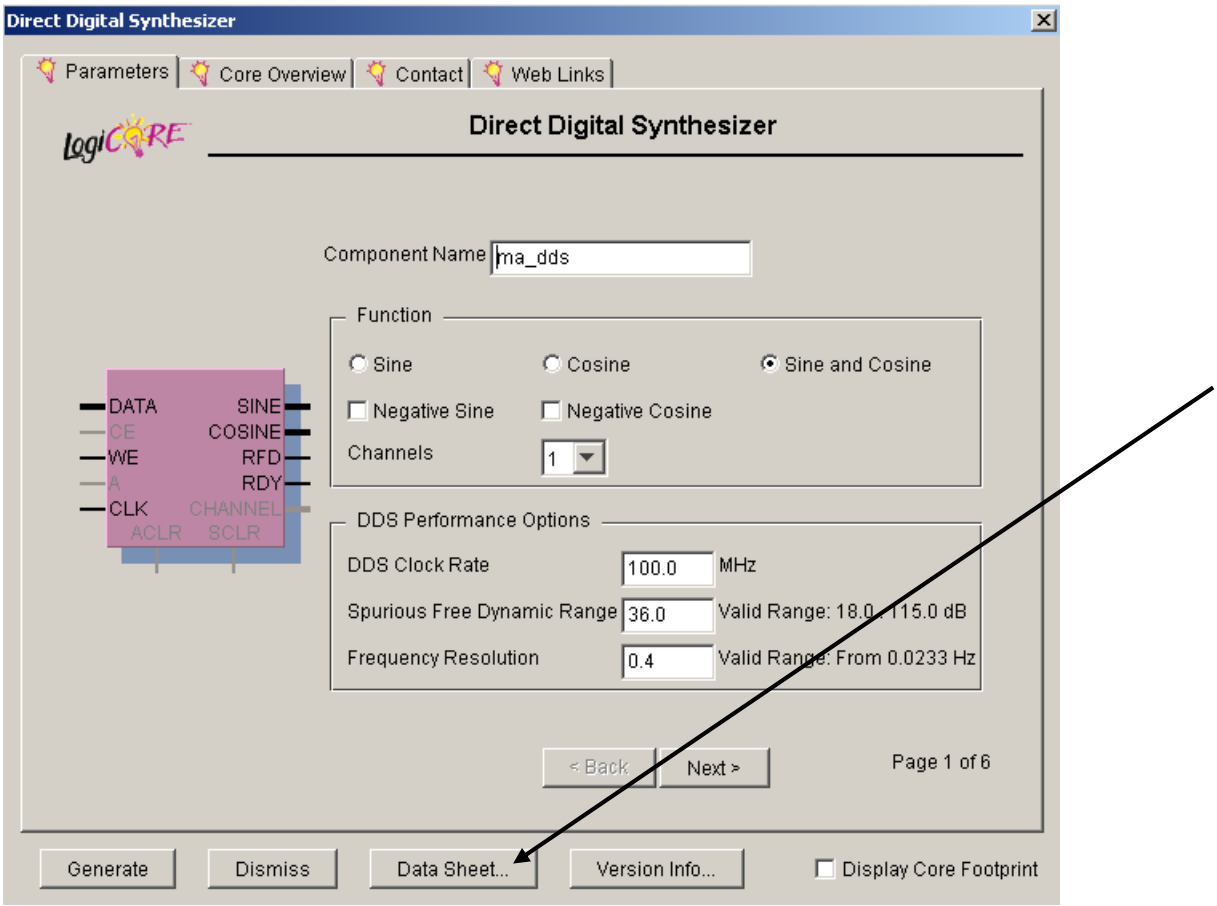
Dans Digital Signal Processing, Waveform Synthesis, sélectionnez DDS puis cliquez sur Suivant :



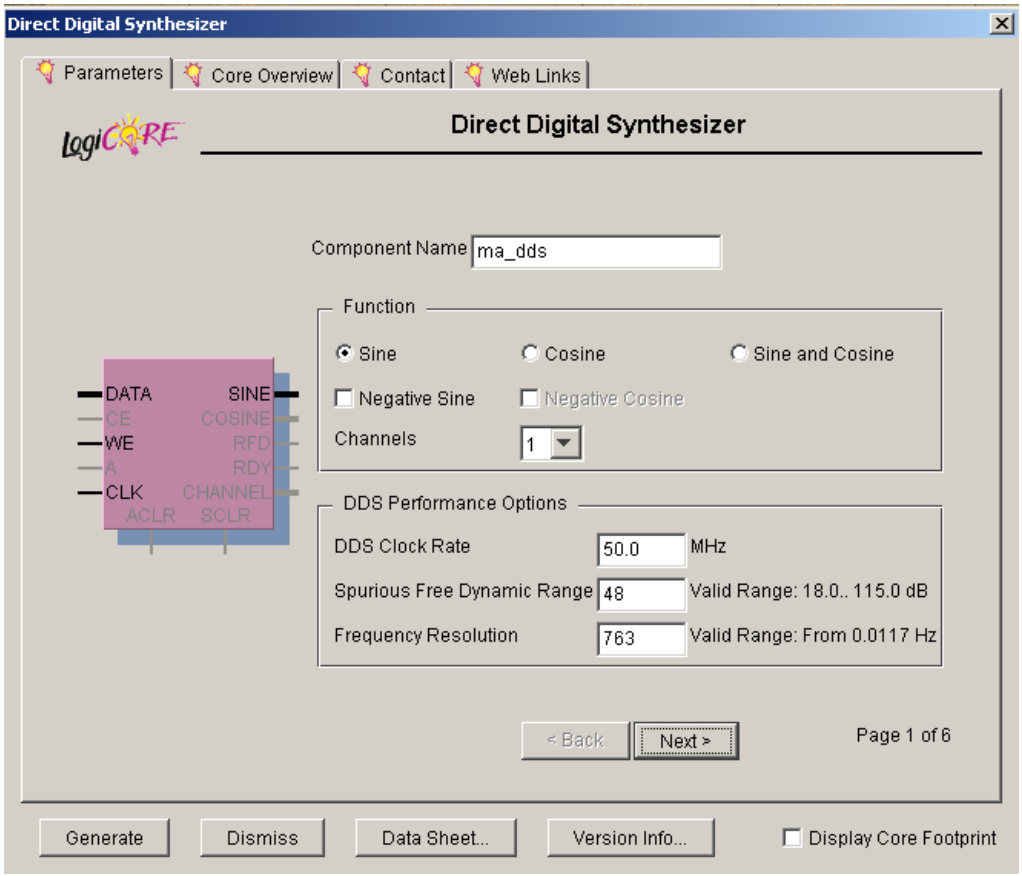
Cliquez sur Terminer dans la fenêtre suivante :

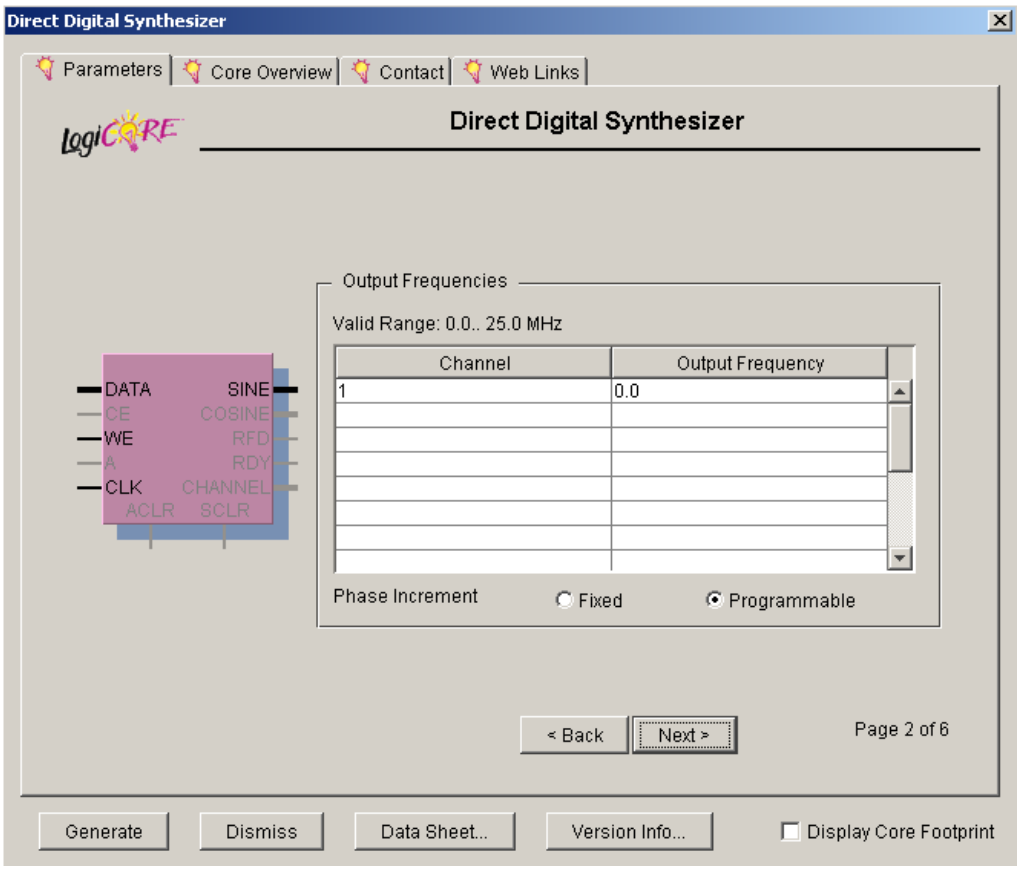


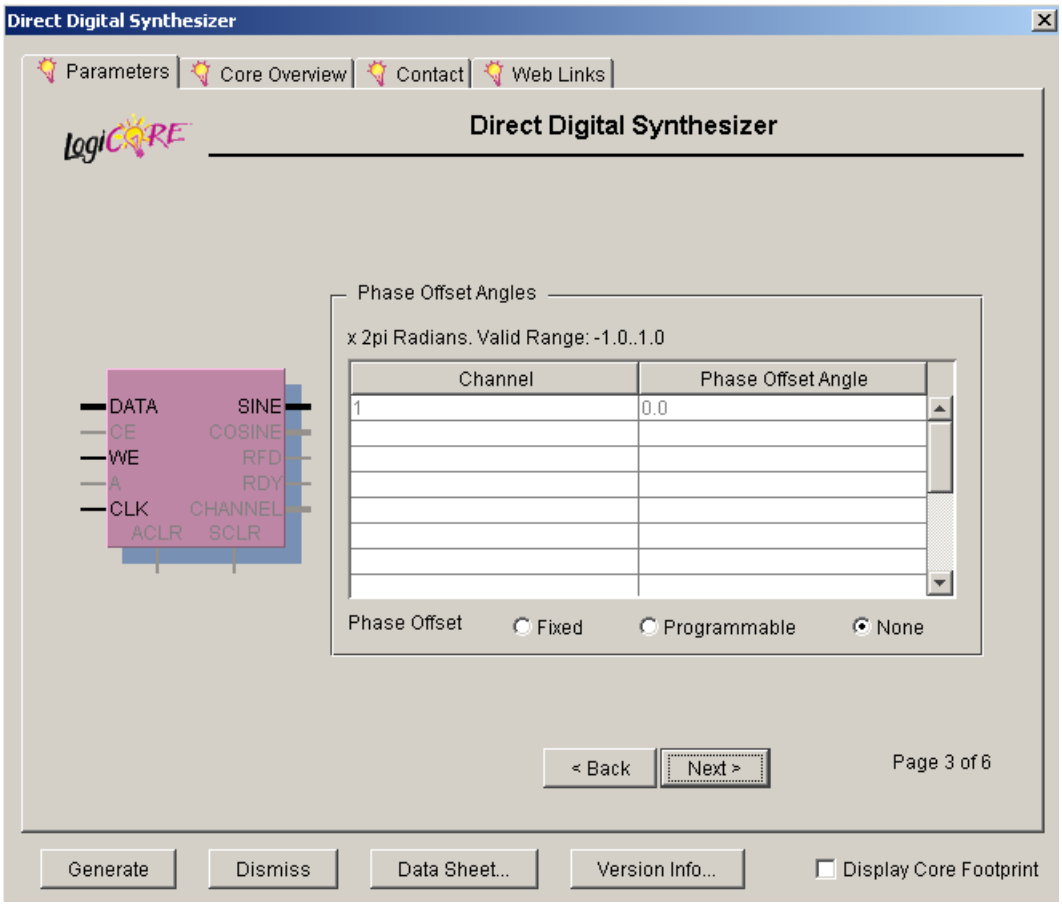
La fenêtre de configuration de l'IP s'ouvre. En cliquant sur le bouton « Data Sheet », vous pouvez accéder à la documentation du composant :

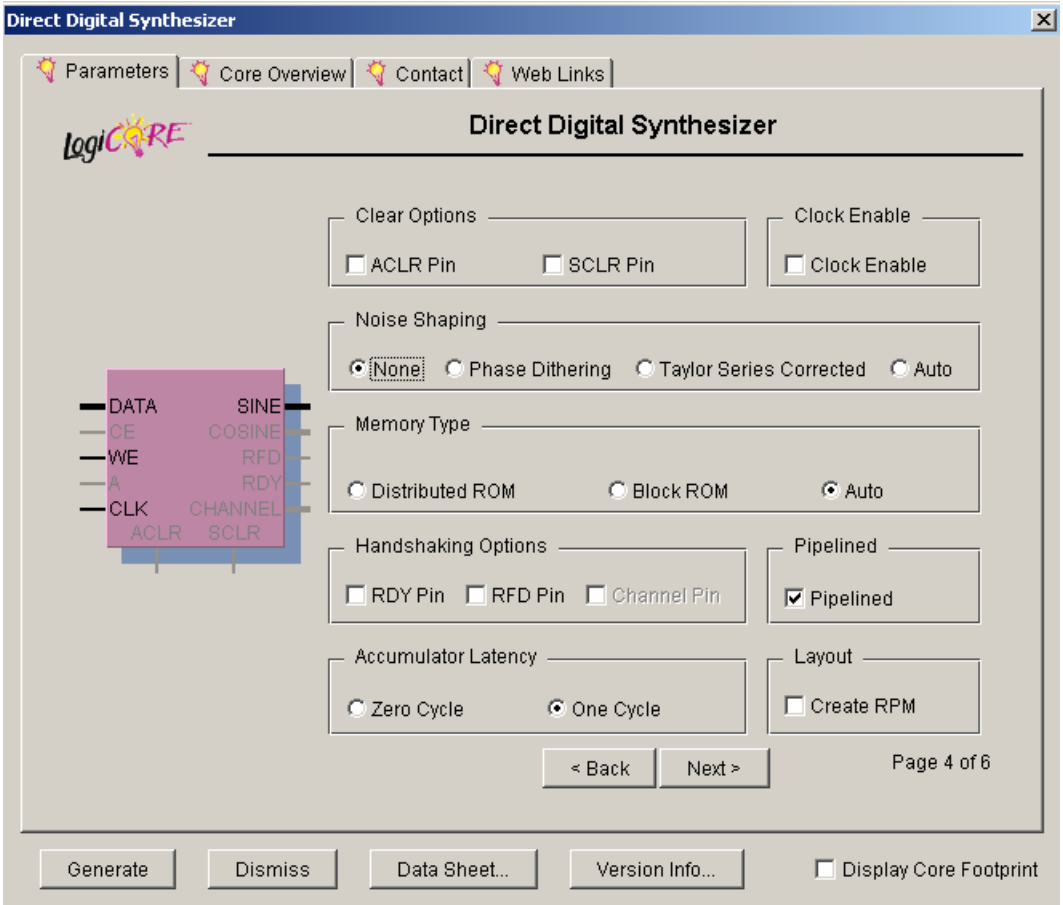


Nous allons maintenant renseigner les 4 pages de configuration de Coregen :

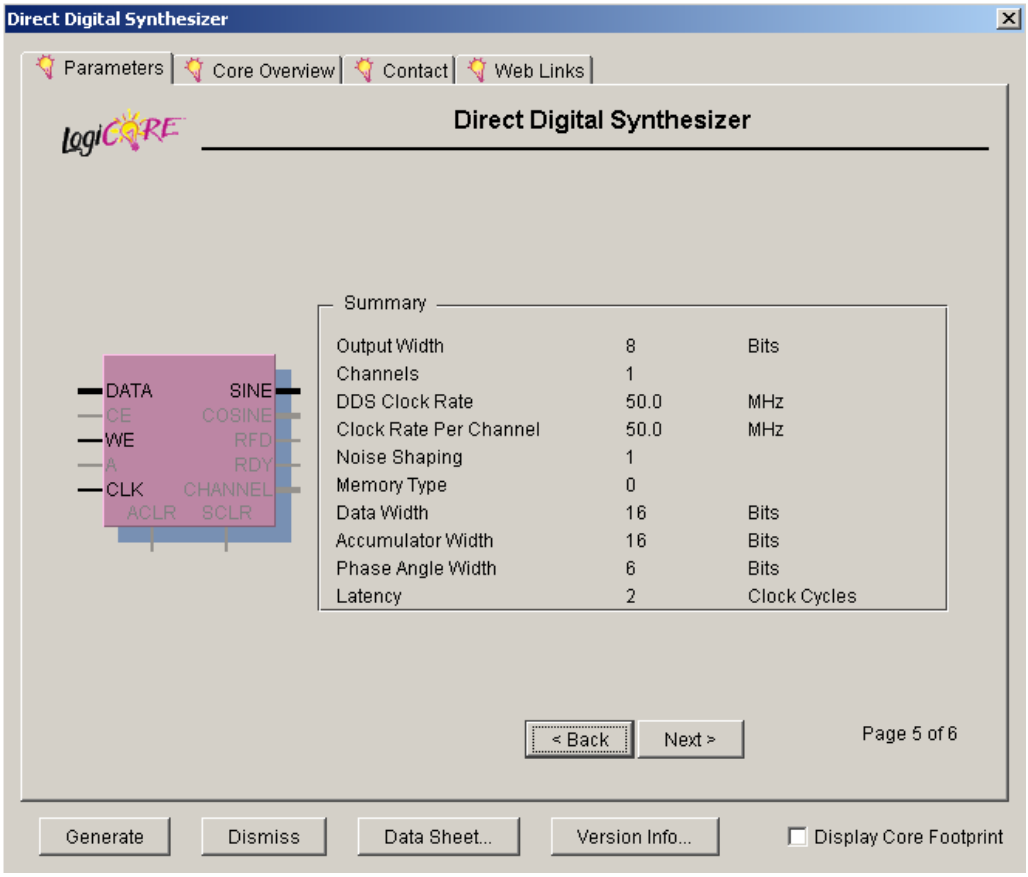


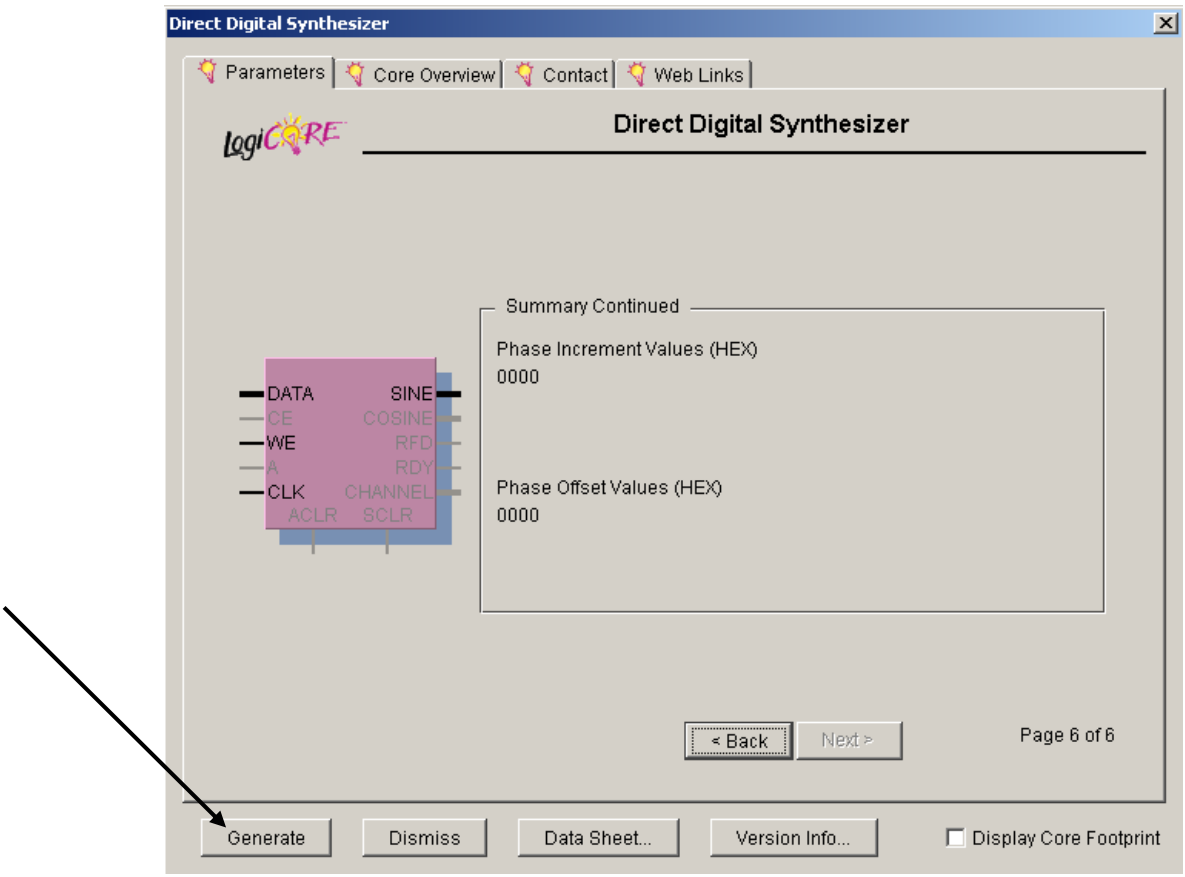



Direct Digital Synthesizer
Parameters
Core Overview
Contact
Web Links
Direct Digital Synthesizer
DATA
CE
WE
A
CLK
ACLR
SINE
COSINE
RFD
RDY
CHANNEL
SCLR
Phase Offset Angles
x 2pi Radians. Valid Range: -1.0..1.0
Channel
Phase Offset Angle
1
0.0
Phase Offset
Fixed
Programmable
None
< Back
Next >
Page 3 of 6
Generate
Dismiss
Data Sheet...
Version Info...
Display Core Footprint

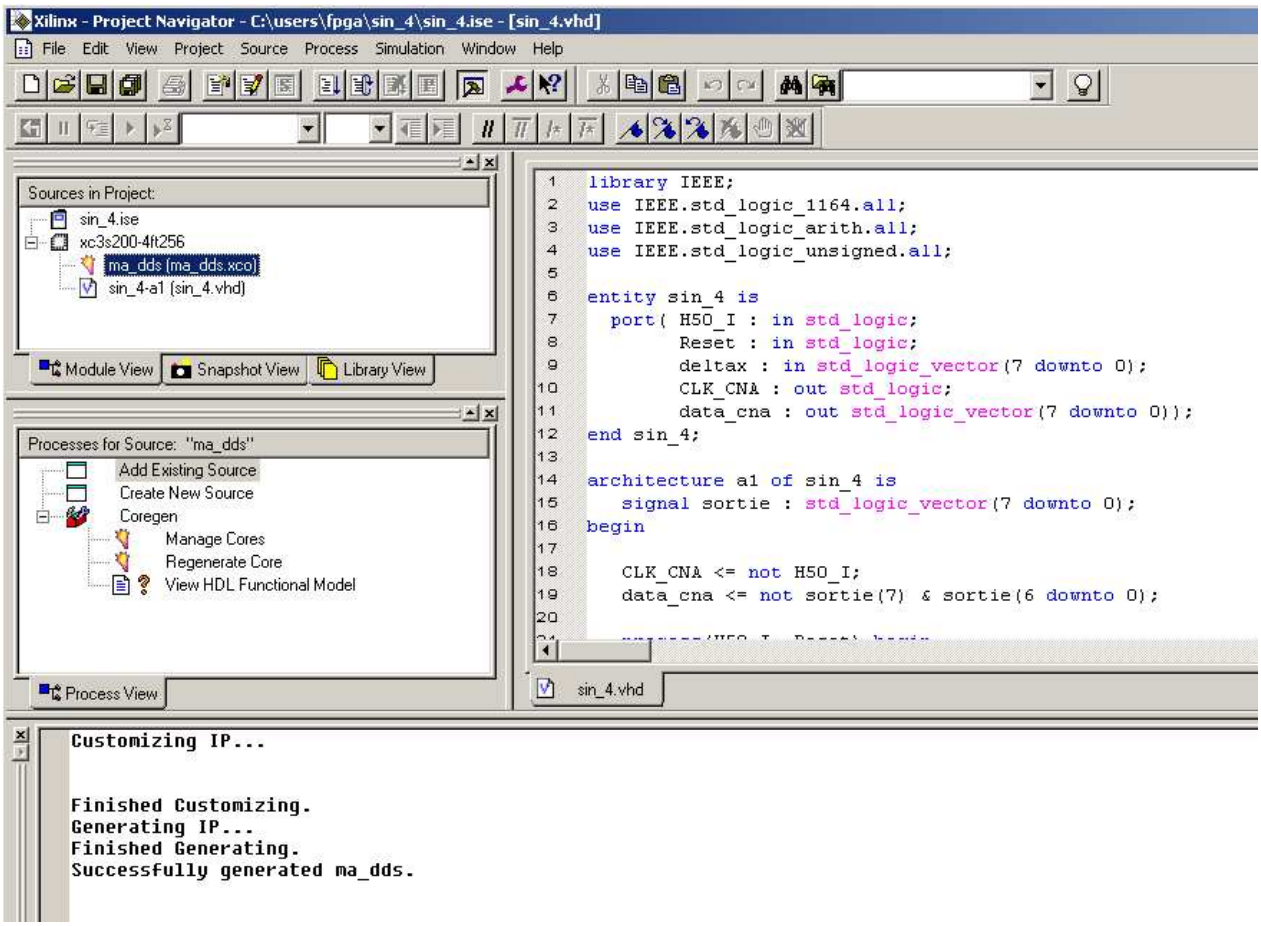

Direct Digital Synthesizer
Parameters
Core Overview
Contact
Web Links
Direct Digital Synthesizer
DATA
CE
WE
A
CLK
ACLR
SINE
COSINE
RFD
RDY
CHANNEL
SCLR
Clear Options
ACLR Pin
SCLR Pin
Clock Enable
Clock Enable
Noise Shaping
None
Phase Dithering
Taylor Series Corrected
Auto
Memory Type
Distributed ROM
Block ROM
Auto
Handshaking Options
RDY Pin
RFD Pin
Channel Pin
Pipelined
Pipelined
Accumulator Latency
Zero Cycle
One Cycle
Layout
Create RPM
< Back
Next >
Page 4 of 6
Generate
Dismiss
Data Sheet...
Version Info...
Display Core Footprint

Sur les deux dernières pages de vérification, vous devez obtenir :

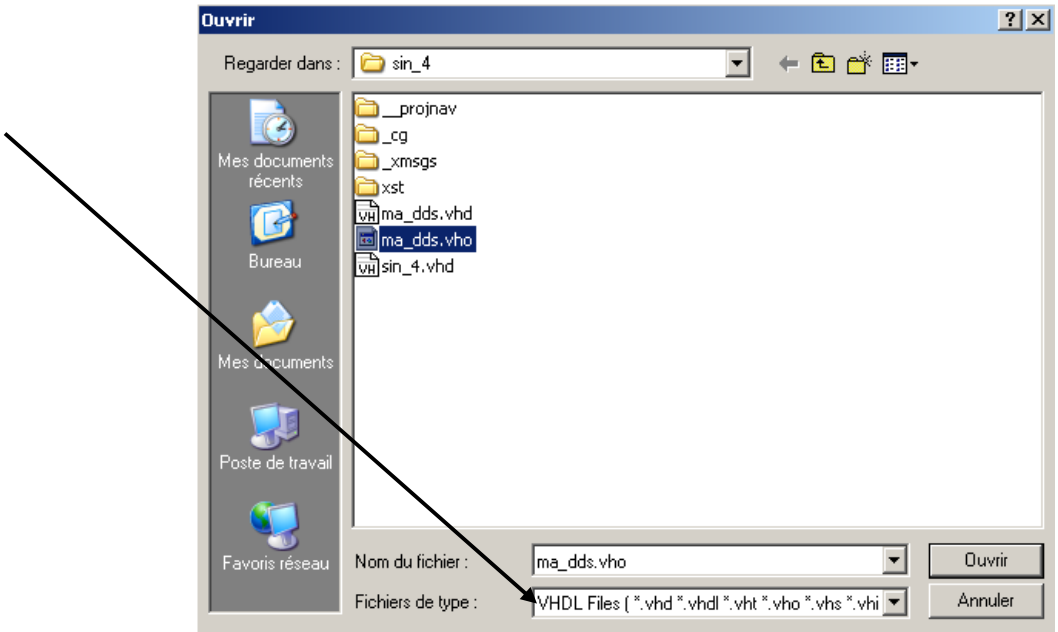




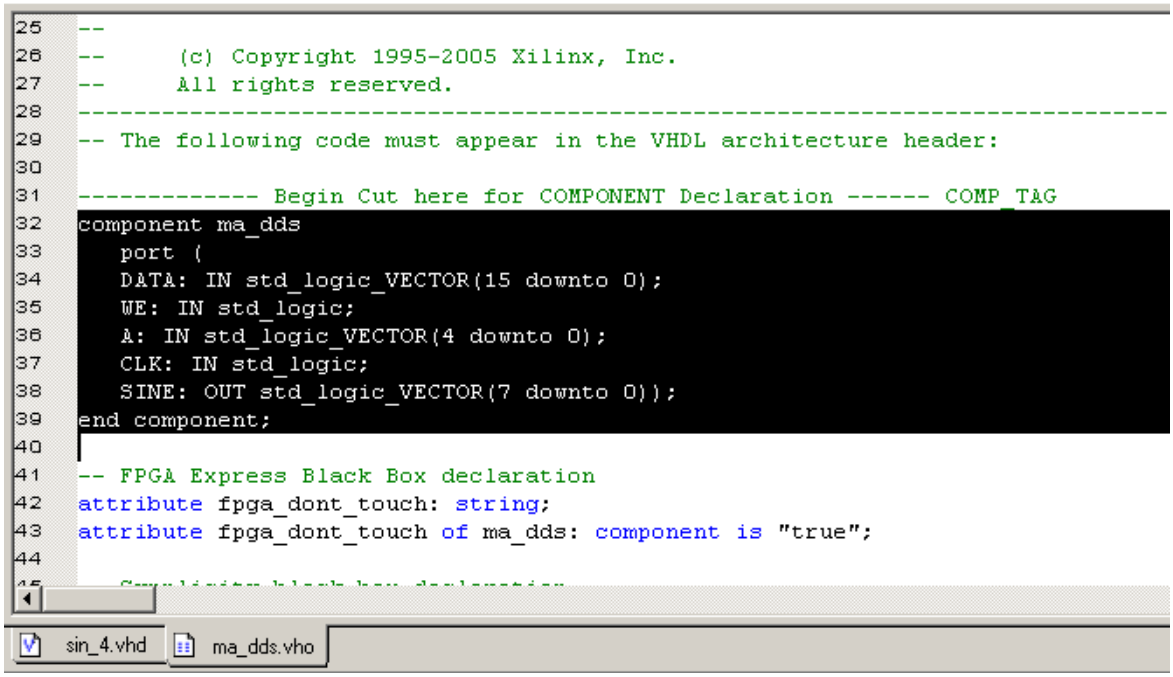
Cliquez sur Generate pour créer le composant. La fenêtre de Coregen se ferme et vous obtenez :



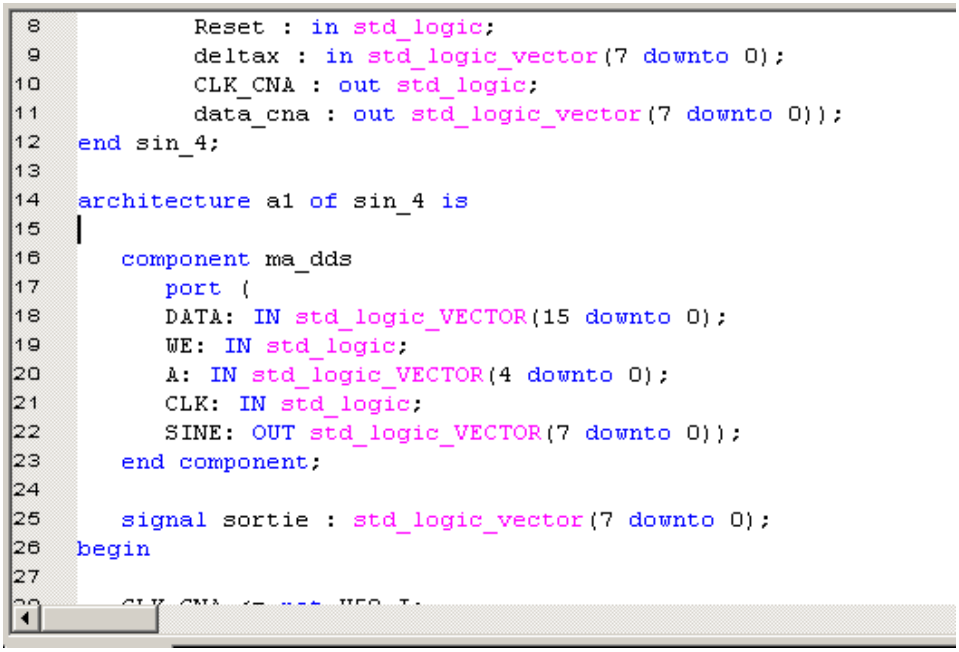
Pour instancier le composant ma_dds, allez dans le menu File, Open et ouvrez le fichier ma_dds.vho :



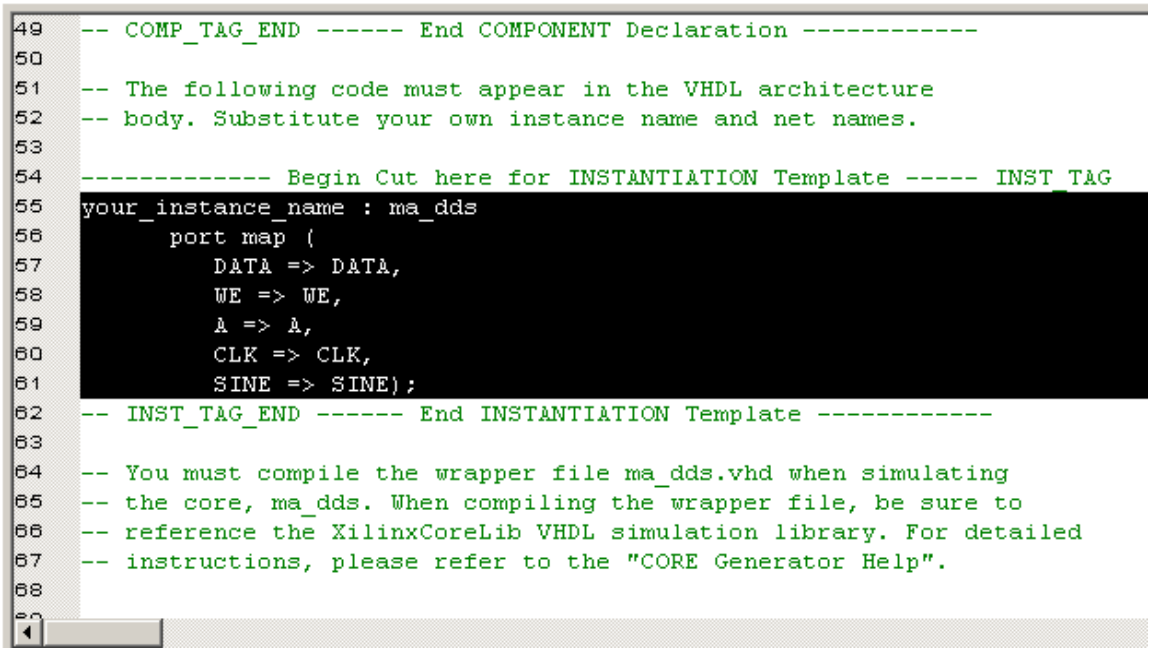
Sélectionnez la déclaration du composant ma_dds dans ma_dds.vho :



Et copiez-le dans sin_4.vhd :

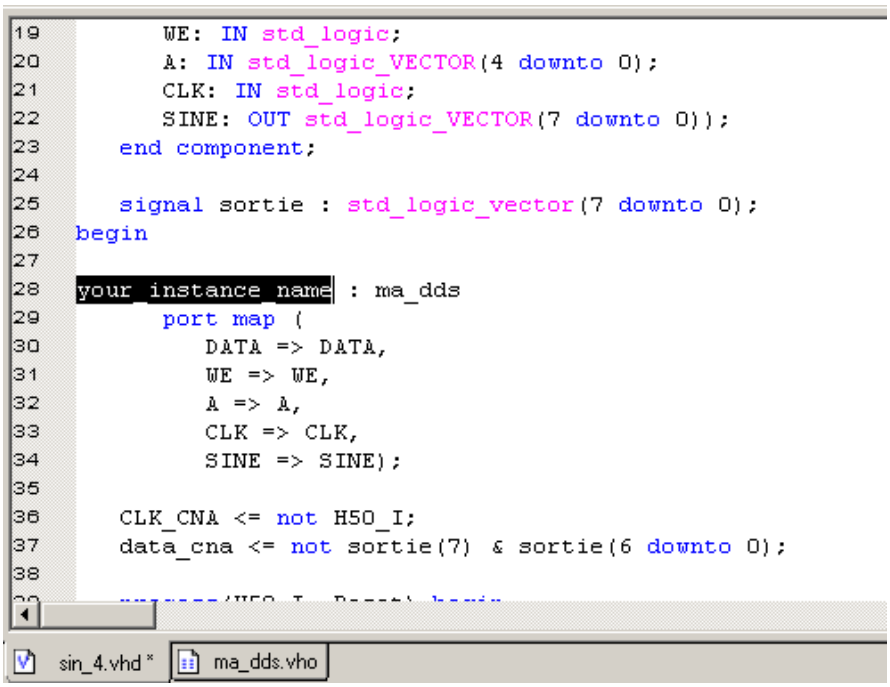


Puis sélectionnez l'exemple d'instanciation du composant ma_dds dans ma_dds.vho :



```
49  -- COMP_TAG_END ------ End COMPONENT Declaration ------------
50
51  -- The following code must appear in the VHDL architecture
52  -- body. Substitute your own instance name and net names.
53
54  ------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG
55  your_instance_name : ma_dds
56        port map (
57           DATA => DATA,
58           WE => WE,
59           A => A,
60           CLK => CLK,
61           SINE => SINE);
62  -- INST_TAG_END ------ End INSTANTIATION Template ------------
63
64  -- You must compile the wrapper file ma_dds.vhd when simulating
65  -- the core, ma_dds. When compiling the wrapper file, be sure to
66  -- reference the XilinxCoreLib VHDL simulation library. For detailed
67  -- instructions, please refer to the "CORE Generator Help".
68
```

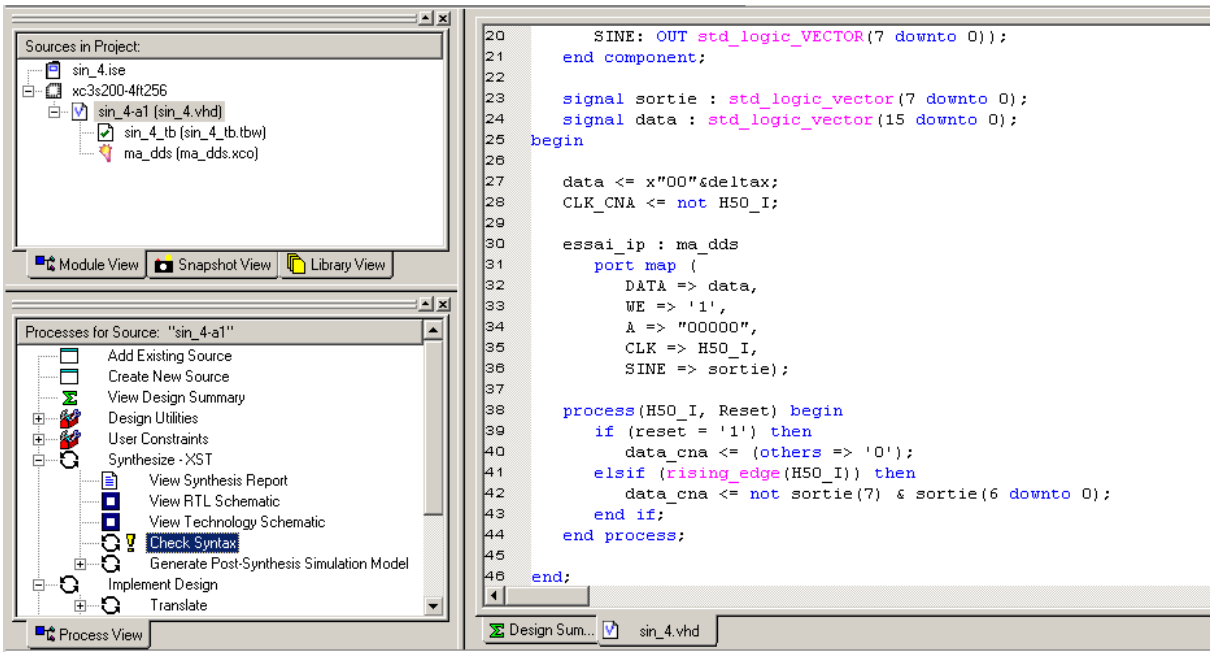
Et copiez-le dans sin_4.vhd :



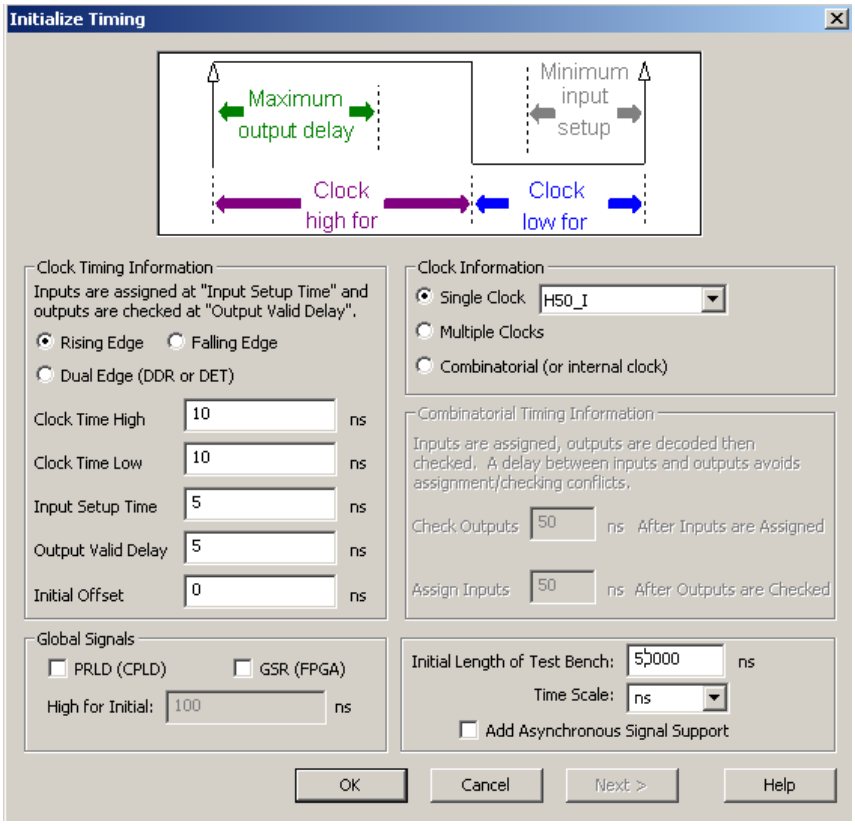
```
19          WE: IN std_logic;
20          A: IN std_logic_VECTOR(4 downto 0);
21          CLK: IN std_logic;
22          SINE: OUT std_logic_VECTOR(7 downto 0));
23       end component;
24
25       signal sortie : std_logic_vector(7 downto 0);
26    begin
27
28    your_instance_name : ma_dds
29          port map (
30             DATA => DATA,
31             WE => WE,
32             A => A,
33             CLK => CLK,
34             SINE => SINE);
35
36       CLK_CNA <= not H50_I;
37       data_cna <= not sortie(7) & sortie(6 downto 0);
38
```

sin_4.vhd * | ma_dds.vho

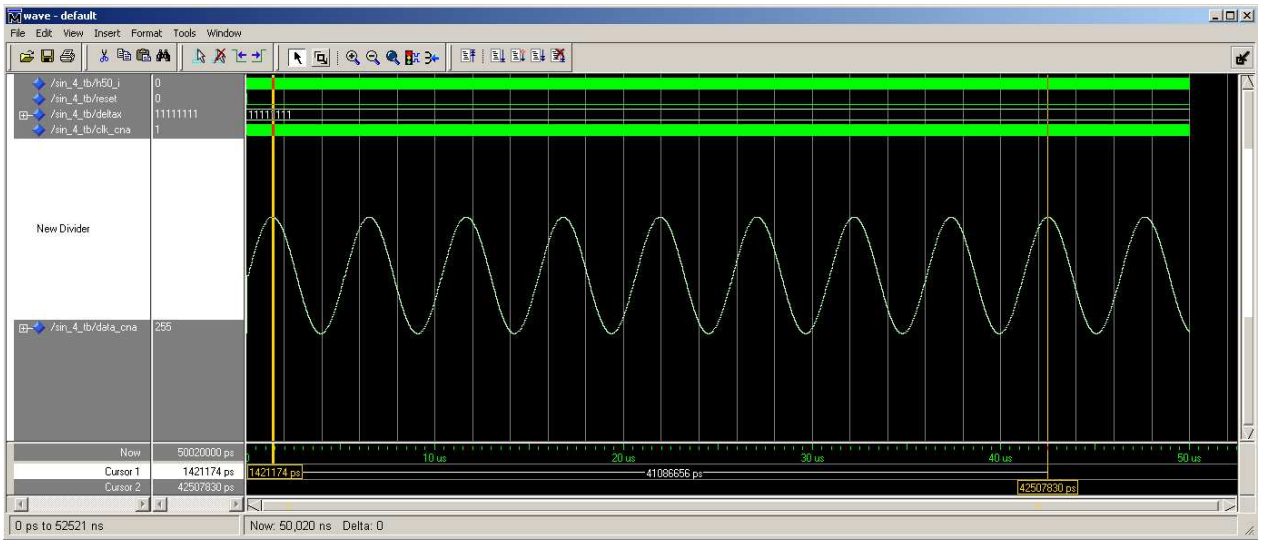
Il faut maintenant brancher la dds sur les signaux existants dans sin_4. Modifiez sin_4.vhd pour obtenir :



Pour la simulation, vous utiliserez les réglages suivants puis vous assignerez deltax = 255 :



Avec deltax = 255, vous vérifierez que la fréquence de sortie est bien égale à 194550 Hz.



Sur la maquette, réglez les dip switches SW0 à SW7 pour faire varier `deltax` et vérifiez les fréquences obtenues.

7.8.6 Montage n°5

Nous allons maintenant étudier une bizarrerie : un filtre récursif du second ordre réglé pour osciller. Vous vous reporterez à votre cours de traitement du signal pour plus de détails théoriques. On rappelle que pour un filtre IIR du second ordre, on a l'équation de récurrence :

$$y_n = (1 + a_1 + a_2)x_n - a_1 y_{n-1} - a_2 y_{n-2}$$

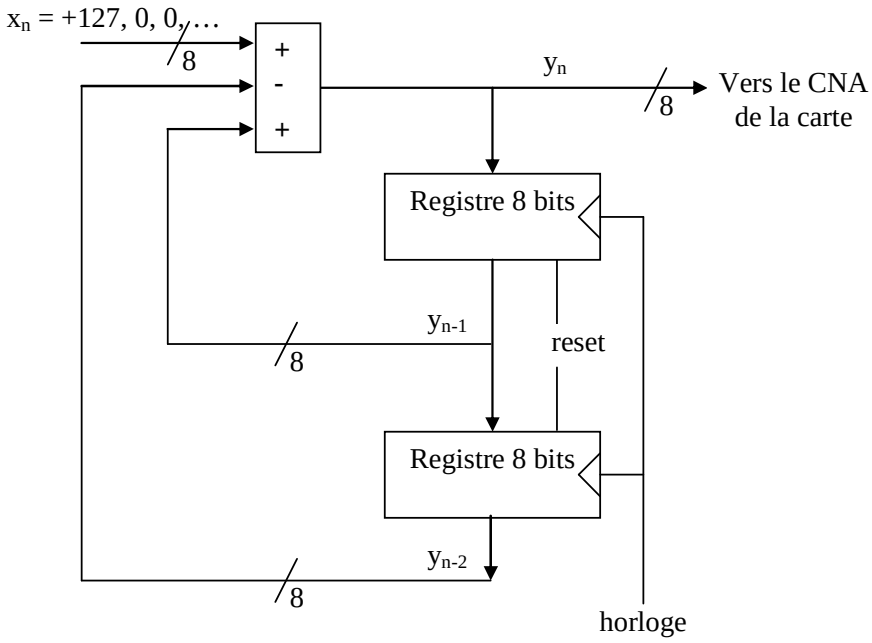
Et que le montage oscille pour $a_2 = 1$. La fréquence d'oscillation f_0 est égale à :

$$\frac{f_0}{fech} = \frac{1}{2\pi} . Arc\cos\left|\frac{-a_1(1+a_2)}{4a_2}\right|$$

On prendra $a_1 = -1$ pour simplifier le montage. L'équation de récurrence devient alors :

$$y_n = x_n + y_{n-1} - y_{n-2}$$

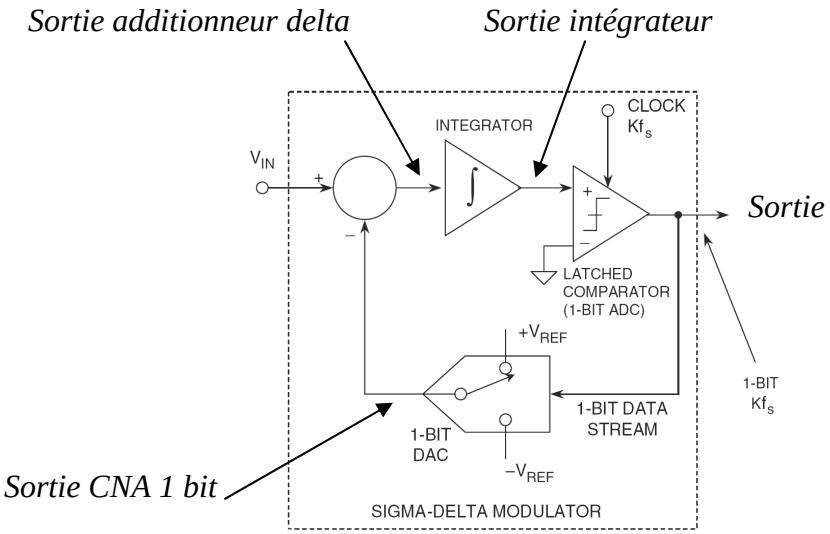
La fréquence d'oscillation est égale à 50/6 MHz. On travaillera en CA2 8bits pour tous les signaux. Pour que le montage démarre correctement, on appliquera un dirac positif (+127) sur x_n à l'initialisation, puis x_n reviendra à 0. Le montage à réaliser est donc le suivant :



Modifiez le design sin_1 (nouveau design sin_5). Vérifiez la fréquence obtenue (l'amplitude est un peu faible en sortie du CNA à cause du filtre passe-bas).

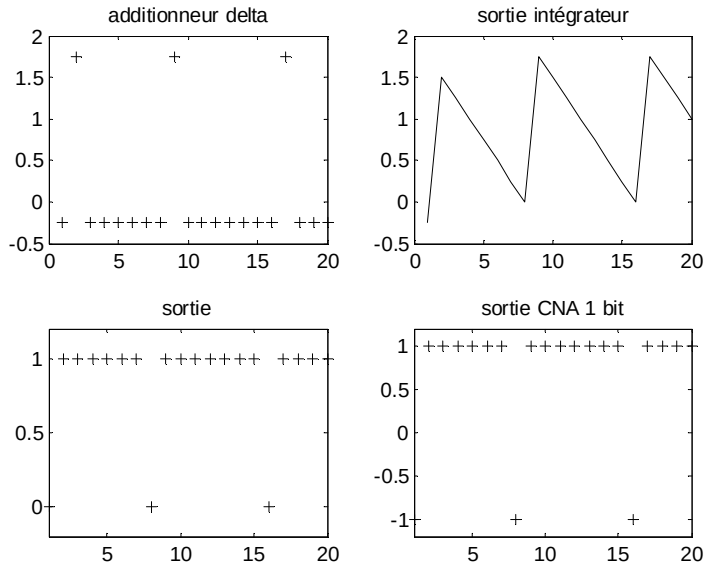
7.8.7 Montage n°6

Dans ce dernier design, nous allons reprendre sin_2 et utiliser un CNA sigma-delta interne au FPGA à la place du CNA se trouvant sur la carte. Le convertisseur sigma-delta (Σ-Δ) utilise le principe de la modulation sigma-delta :



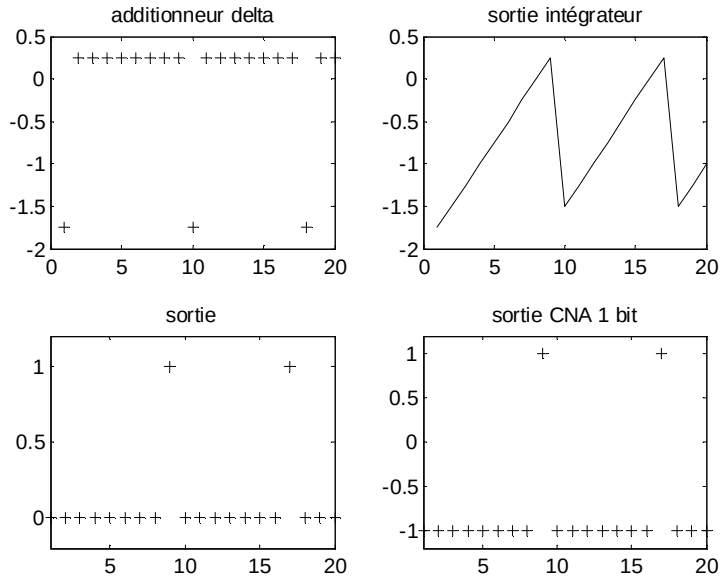
La simulation Matlab suivante vous donne l'évolution des différents signaux avec Vref = 1 et Vin = 0,75. Avec une constante sur son entrée, un intégrateur génère une simple rampe dont la pente est proportionnelle à l'entrée. Donc l'intégrateur Σ génère une rampe proportionnelle à la différence (Δ) entre l'entrée Vin et la sortie du CNA 1 bit. La sortie du montage (après un comparateur à 0) est codée sur 1 bit. Si on place un filtre RC sur la sortie, on obtient une valeur moyenne = 7/8 (7 bits à 1, 1 bit à 0).

Vin = +0.75



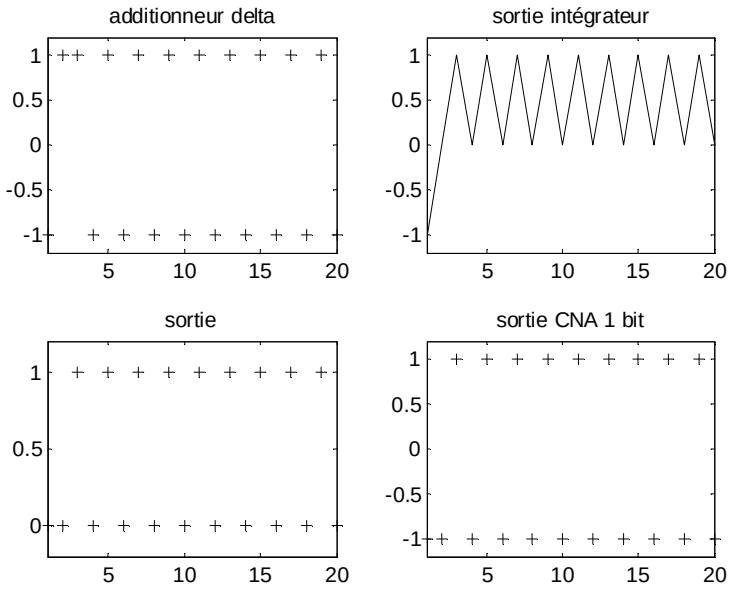
Avec Vin = -0.75, la valeur moyenne sur la sortie tombe à 1/8 (7 bits à 0, 1 bit à 1).

Vin = -0.75

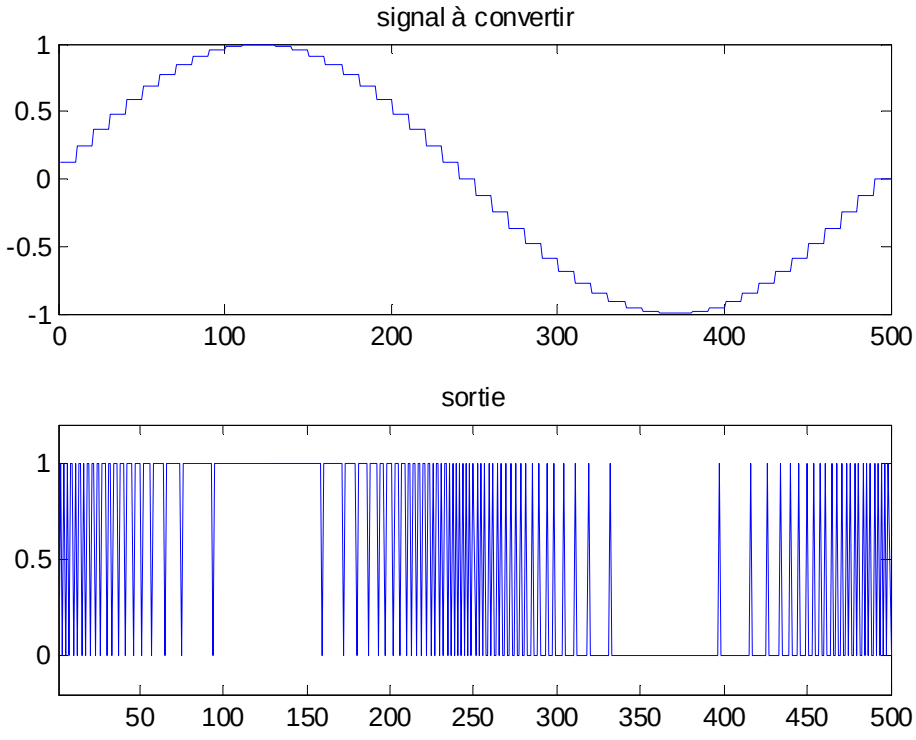


Avec Vin = 0, la valeur moyenne sur la sortie est égale à 4/8 (1 bit à 0, 1 bit à 1).

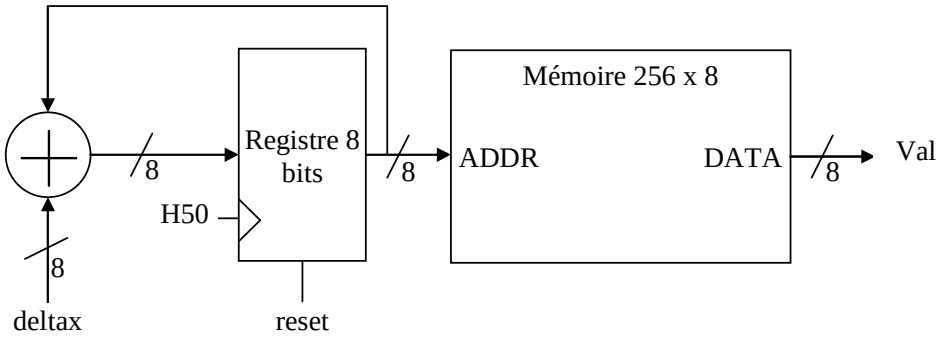
Vin = 0



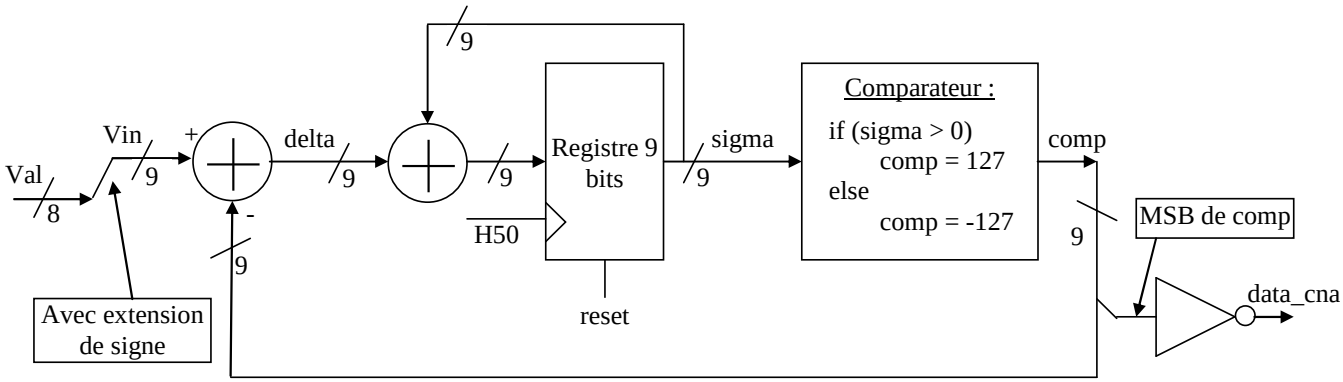
La sortie après filtrage RC est donc proportionnelle à Vin, à condition que Vin change lentement pour laisser le temps à la sortie de s'adapter. Si Vin suit une valeur sinusoïdale en restant suffisamment longtemps sur chaque palier, alors la valeur moyenne de la sortie suit la même variation :



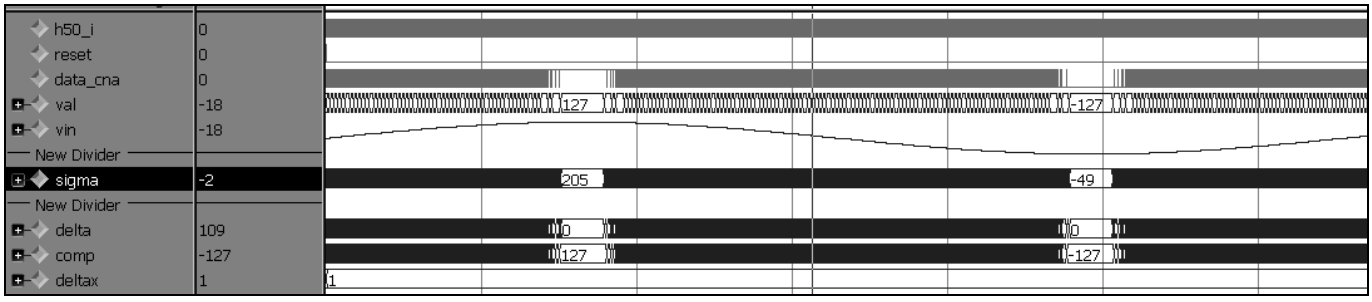
Nous allons reprendre le design sin_2 et appeler sa sortie val :



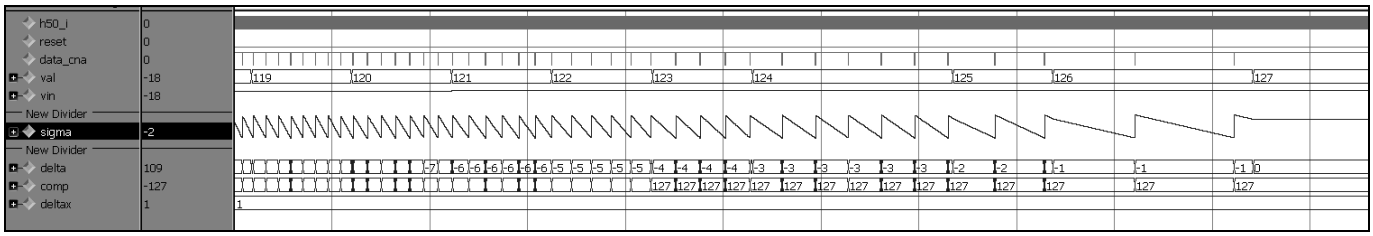
Puis nous allons rentrer val sur Vin (passage de 8 à 9 bits avec extension de signe). Le modulateur Σ-Δ tout numérique est construit de la façon suivante (tous les signaux en CA2) :



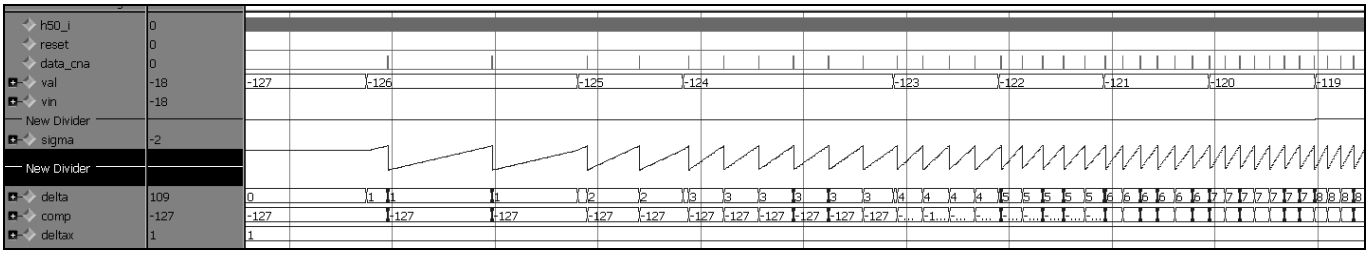
On retrouve les différentes parties du modulateur analogique : l'additionneur Δ, l'accumulateur sigma (l'intégrateur Σ) et le comparateur qui est ici fusionné avec le CNA 1 bit. Modifiez le design sin_2 (nouveau design sin_6). Attention, data_cna est maintenant codé sur un seul bit (relié à la broche N7 du FPGA). Vous devez obtenir une simulation équivalente aux chronogrammes suivants :



Simulation générale sur 2 ms



Zoom sur la valeur max positive : val = 127



Zoom sur la valeur max négative : val = -127

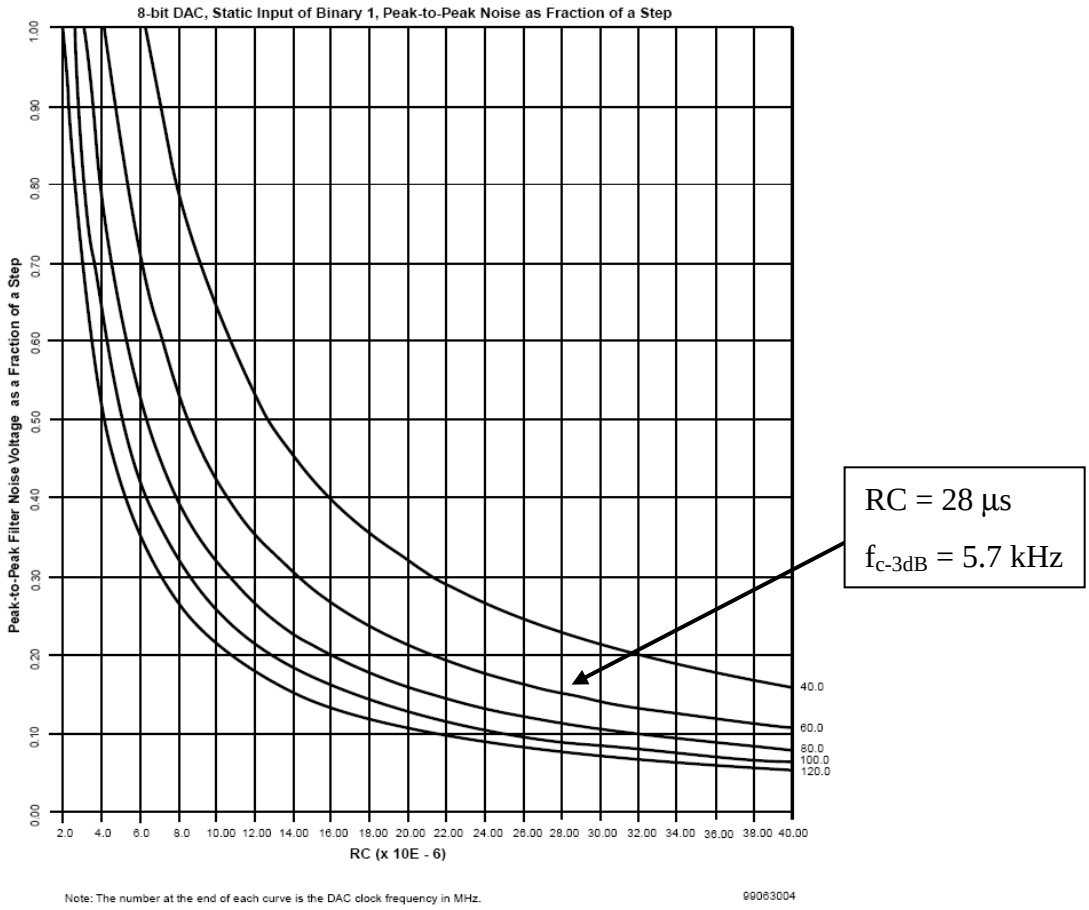
Avec ce montage réel, nous pouvons calculer la fréquence d'échantillonnage Fech du montage. A la sortie de l'intégrateur :

- La rampe maximale positive se produit quand val = 126. Elle démarre à 255 et décroit par pas de 1 jusqu'à 0. Cela nous donne sur data_cna : 255 bits à 1 suivi d'un bit à 0. Pour val = 127, la rampe est une constante et la sortie vaut 1 en permanence.
- La rampe maximale négative se produit quand val = -126. Elle démarre à -254 et croit par pas de 1 jusqu'à +1. Cela nous donne sur data_cna : 255 bits à 0 suivi d'un bit à 1. Pour val = -127, la rampe est une constante et la sortie vaut 0 en permanence.

Sigma doit donc être codé sur 9 bits en CA2, ce qui implique que tous les signaux du modulateur doivent être en 9 bits CA2 (sauf l'entrée val qui est en 8 bits CA2). On voit que chaque échantillon en sortie du CNA est codé avec 256 bits successifs et donc que l'on a :

Fech = 50 MHz / 256 = 195312,5 Hz

La fréquence du signal sinusoïdal généré par le design devra bien sûr être inférieure à Fech / 2 (Shannon doit toujours être respecté). Dernier point : quelles valeurs de RC faut-il utiliser pour filtrer data_cna ? Sans entrer dans les détails de calcul, on peut utiliser l'abaque suivant fourni par Xilinx. Elle indique le niveau de bruit maximum en sortie du filtre en fonction d'un pas de sortie (LSB). Dans le cas d'un CNA 8 bits, un niveau de 1 correspond à 1/256 du niveau maximum du signal de sortie. Le choix de RC est assez approximatif.



Branchez le filtre RC (1,3 kΩ, 22 nF) comme indiqué sur la photo suivante. Vous connecterez la sonde de l'oscilloscope entre R et C pour observer le signal sinusoïdal.