Flot de conception et modèles formels pour la désynchronisation de circuits synchrones

François Bertrand

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Nano Électronique & Nano Technologies**

Arrêté ministériel : 25 mai 2016

Présentée par

François BERTRAND

Thèse dirigée par **Laurent FESQUET**
et co-encadrée par **Anthony MAURE**

Préparée au sein du **Laboratoire TIMA**
dans **l'École Doctorale Electronique, Electrotechnique, Automatique & Traitement du Signal (E.E.A.T.S)**

Flot de conception et modèles formels pour la désynchronisation de circuits synchrones

Thèse soutenue publiquement le **2 Juillet 2019**,
devant le jury composé de :

M. Philippe PANNIER
Professeur, Polytech Marseille, IM2NP, Président
M. Bruno ROUZEYRE
Professeur, Université Montpellier 2, LIRMM-CNRS, Rapporteur
M. Lilian BOSSUET
Professeur, Université Jean-Monnet de Saint-Etienne, LHC, Rapporteur
M. Laurent FESQUET
Maître de Conférences, Université Grenoble Alpes, TIMA, Directeur de thèse
M. Anthony MAURE
Ingénieur R&D, Idemia-StarChip, Co-Encadrant de thèse

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Le marché des *smartcards* évolue vers l'utilisation de dispositifs dits *duals*, *i.e.* qui peuvent fonctionner à la fois par contact avec la source d'alimentation, et de communication, mais aussi sans contact par le biais d'une alimentation, et d'une communication radio-fréquences. Dans ce dernier cas, le système complet à besoin d'une antenne permettant de récupérer l'énergie et les communications.

Dans l'industrie, un système intégré tel qu'un carte à puce est conçu pour être vendu à un grand nombre d'exemplaire. Par conséquent, le coût unitaire de chaque puce est un point critique pour les clients. Dans une optique de réduction de coûts, le projet LISA (FUI-AAP17), pour *ultra-**L**ow power **I**ntegrated circuit for **S**ecure RF **A**pplications*, vise à déplacer l'antenne qui est aujourd'hui déposée sur la carte en plastique vers le module qui intègre la puce et les plots de contact de carte. En effet, les coûts engendrés par la fabrication et l'intégration de l'antenne sur la carte en plastique seront supprimés. Il ne subsistera alors que le coût de l'intégration de la puce dans le module qui intégrera à la fois les plots de contact et l'antenne. Cette approche audacieuse va toutefois grandement réduire la taille de l'antenne et son efficacité.

Le dispositif de récupération d'énergie étant de taille plus petite, la quantité d'énergie récupérée va s'en trouver réduite. Il devient donc nécessaire de réduire en conséquence la puissance consommée par le circuit numérique de la *smartcard*. De plus, si une alimentation par contact fournie une alimentation relativement stable et d'une puissance suffisante, le budget puissance disponible en mode sans contact peut varier de manière significative. Cela demande donc d'ajouter des dispositifs utilisant le RSSI, pour *Received Signal Strength Indication*, afin d'adapter les performances du circuit aux conditions auxquelles il est soumis.

Cette thèse s'inscrit dans cette problématique en explorant une piste non conventionnelle utilisant des circuits asynchrones. En effet, ces derniers sont théoriquement de parfaits candidats pour les problématiques adressées dans ce projet. Les principaux atouts généralement listés pour les circuits asynchrones sont :

— Les performances : comparé à un circuit synchrone qui est contraint par son plus long chemin, un circuit asynchrone présente des performances qui correspondent à un temps

d'exécution moyens.

— La robustesse : les mécanismes de synchronisation étant locaux et soumis aux mêmes variations que le reste du circuit, les circuits asynchrones adaptent naturellement leurs comportements pour fonctionner dans une plage large de tension et de température.

— La faible consommation : l'activité dans un circuit asynchrone dépend uniquement des données en cours d'exécution. Les chemins de données non utilisés par les données ne sont pas activés, ce qui permet d'économiser de l'énergie.

— ...

Cependant, tous ces avantages sont contre-balancés par les deux principaux freins à l'adoption de circuits asynchrones par l'industrie :

— Le manque de formation des ingénieurs pour la conception de circuits asynchrones.

— Le manque d'outils prouvés et qualifiés industriellement

Étant donné qu'il parait difficilement envisageable, dans un premier temps en tout cas, de remettre en cause la formation de générations d'ingénieurs en conception numérique, l'approche envisagée durant les travaux de cette thèse a été de se placer au plus proche de la conception synchrone en concevant des circuits asynchrones dits *micropipelines*.

La proximité avec les circuits synchrones permet, d'une part, de conserver un flot proche de la conception synchrone tout en retirant certains avantages liés à la conception asynchrone. Enfin, l'usage des outils classiquement utilisés pour concevoir des circuits synchrones serait un plus. Ainsi, pour être techniquement proche de la conception des circuits synchrones, ces travaux ne proposerons pas à proprement parler la création d'un circuit asynchrone mais plutôt la transformation d'un circuit synchrone existant en un circuit asynchrone *micropipeline*. Nous appellerons cette procédure désynchronisation.

Dans le Chapitre 2, nous faisons un état des lieux des différentes techniques classiquement utilisées pour optimiser l'efficacité des circuits. Nous ferons aussi dans ce chapitre une présentation des différents types de circuits asynchrones existant et expliciterons également plus précisément les choix sur lesquels se basent nos travaux.

Ensuite, le Chapitre 3 présente les différents modèles utilisés dans la suite du manuscrit ainsi que les premières règles de conception à respecter. Il y sera également présenté une bibliothèque de composants pour la désynchronisation.

Le Chapitre 4 se concentrera sur la méthodologie de transformation des circuits synchrones en circuits désynchronisés. Nous nous attarderons aussi sur différentes optimisations pouvant être opérées pour améliorer la désynchronisation.

Dans le Chapitre 5, nous nous focalisons sur les mesures à prendre pour obtenir des circuits désynchronisés fonctionnels après fabrication.

Enfin, le Chapitre 6 présentera les résultats obtenus en simulation et sur silicium d'un circuit développé au sein de l'entreprise d'accueil. Nous compléterons cette étude avec des résultats obtenus uniquement par simulation.

Chapitre 2

État de l'art

L'évolution de la société et de la technique cadence le marché de la technologie. Cette évolution est directement ressentie par le monde du semi-conducteur qui voit la demande de fonctionnalités sans cesse croître.

Gordon Moore a évoqué cette tendance en 1965 [39] ou il projette un doublement du nombre de transistors par puce d'abord tous les ans puis tous les deux ans. Comme nous pouvons l'observer dans la Figure 2.1, la tendance exprimée par Moore est valable sur les 46 dernières années et il est pressenti qu'elle le sera pour encore quelques années [47] en particulier avec l'augmentation du nombre de cœurs dans les processeurs.

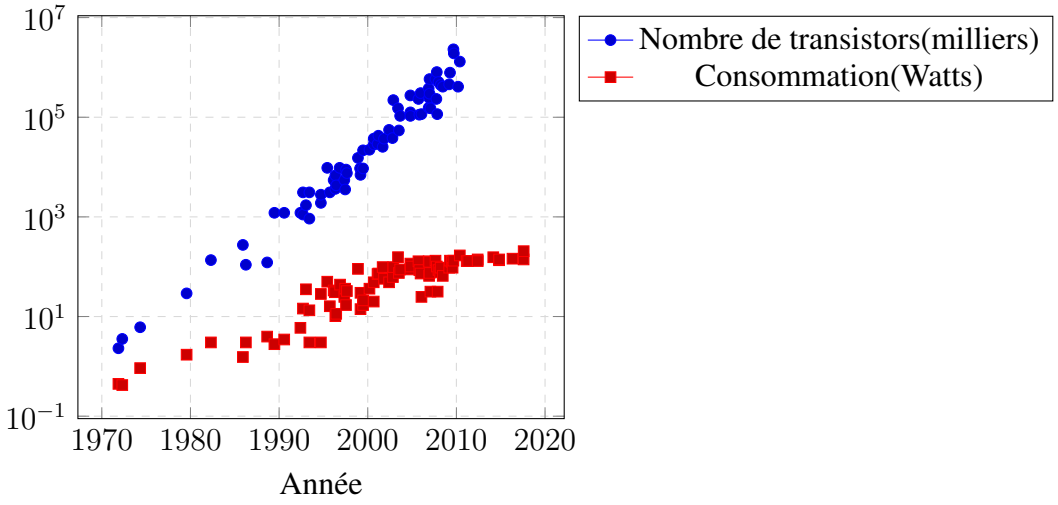


Figure 2.1: Évolution du nombre de transistors et de la consommation de processeur commerciaux de 1971 à 2017 [47]

Naïvement, nous pourrions nous attendre à ce que la consommation des systèmes considérés dans ce graphique suive la même tendance que l'augmentation du nombre de transistors. Cependant, et heureusement pour notre environnement, la puissance consommée par ces systèmes à tendance à stagner autour d'une centaine de watts, en particulier durant la dernière décennie.

Ce comportement n'est pas fortuit et est dû aux efforts fournis lors de la conception des

processeurs pour éviter de dissiper de l'énergie de manière excessive. L'industrie et la recherche dans le domaine des semi-conducteurs ont mis en place différentes techniques afin de limiter la consommation des circuits intégrés.

Ces solutions feront l'objet de la première section de ce chapitre, Section 2.1. Ensuite, nous nous focaliserons sur une autre façon de concevoir les circuits intégrés permettant de réduire leur consommation : les circuits asynchrones. Nous évoquerons tout d'abord les spécificités des circuits asynchrones, Section 2.2. Puis, nous nous intéresserons plus longuement aux types de circuits asynchrones que nous avons considérés dans cette thèse, Section 2.3.1 Enfin, les travaux précédemment menés sur la transformation de circuits synchrones en circuits asynchrones seront présentés, Section 2.4.

2.1 Techniques de Design *Low-Power* classiques

Une première étape pour réduire la consommation des circuits intégrés consiste d'abord en l'identification des sources de ladite consommation, Section 2.1.1. Nous pourrons ainsi dégager des axes pour réduire la consommation de ces circuits, Section 2.1.2

2.1.1 Sources de consommation dans les circuits intégrés

Dans les circuits intégrés, on peut catégoriser la puissance dissipée en deux grandes catégories :

— La consommation statique, aussi appelée *leakage*

— La consommation dynamique

2.1.1.1 Consommation statique

Tout d'abord, nous pouvons considérer la consommation statique d'un transistor, c'est à dire la consommation d'un transistor au repos. Comme nous pouvons le voir dans la Figure 2.2, la consommation statique des circuits intégrés qui était autrefois négligeable, devient aujourd'hui de plus en plus importante dans les nœuds avancés. Le courant de *leakage* provient de nombreux phénomènes physiques qui se produisent dans le transistor. La Figure 2.3 référence les différents courants à l'origine de la consommation statique. Ces courants proviennent de différents phénomènes et peuvent être brièvement décrits de la manière suivante [56] :

Sub-Threshold Current→ Ce courant existe dès lors que la tension de grille n'est pas suffisante pour créer un canal complet. Dans ce cas-là, la polarisation du drain et de la source entraîne une diffusion des porteurs minoritaires. C'est le principal contributeur des courants de *leakage*.

Drain Induced Barrier Lowering, Punch Through Current → La tension appliquée sur le drain cause une augmentation de taille de la zone de déplétion. Avec la réduction de

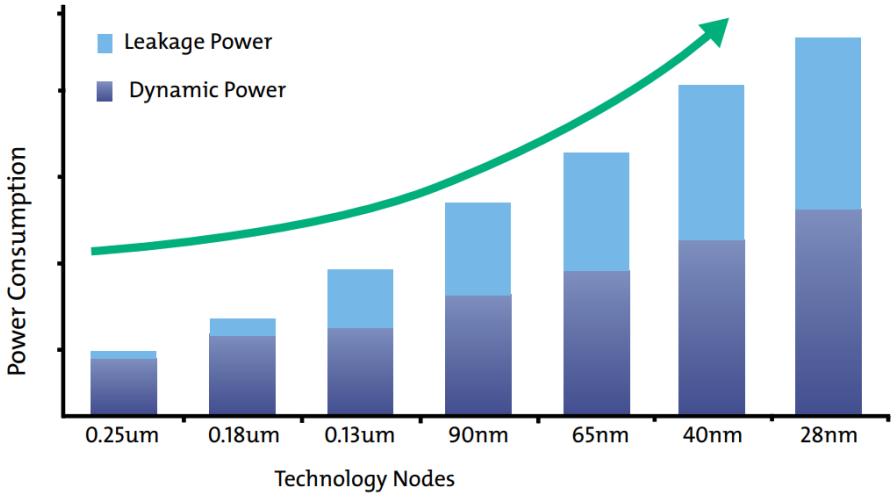


Figure 2.2: Tendance de l'évolution de la répartition consommation statique/dynamique dans un circuit fonction des nœuds technologiques [61]

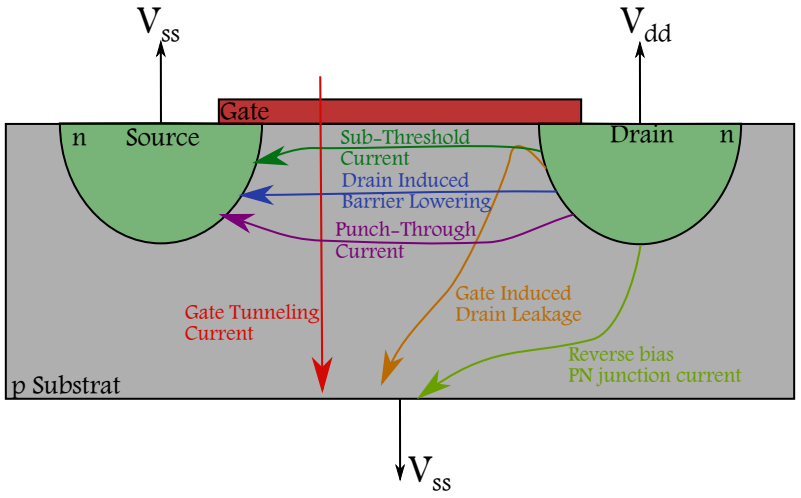


Figure 2.3: Courants de fuite dans un transistor NPN [56]

la taille de transistors, et donc de la distance drain-source, cette zone peut influer sur la source. Cela abaisse la barrière de potentiel entre la source et le drain ce qui a pour effet d'augmenter le nombre de porteurs pouvant traverser cette même barrière. Le *Punch Through Current* suit le même principe mais se produit pour des niveaux de tension plus élevés et peut par conséquent avoir des effets destructifs sur le transistor.

Gate Tunneling Current → Avec les avancées des nœuds technologiques, l'épaisseur de l'oxyde de grille se réduit de plus en plus. Lorsqu'une tension positive est appliquée sur la grille, de forts champs électriques se créent dans l'oxyde et des porteurs de charges peuvent traverser la grille par effet tunnel.

Gate Induced Drain Leakage → Si une forte différence de potentiel existe entre le drain et la grille, une zone de déplétion apparaît dans le drain et des effets de fort champ se produisent comme l'effet tunnel bande-à-bande. Les porteurs qui subissent cet effet

tunnel se retrouvent émis dans le drain et, comme le substrat est à un potentiel plus bas, les porteurs se trouvant proche de la zone de déplétion du drain sont évacués par le substrat.

Reverse Bias Reverse PN Junction Current → Entre le drain et le substrat, des jonctions PN parasites existent. Étant polarisé en inverse, un courant de diode inverse est créé.

Ces courants de fuite dépendent principalement des paramètres suivants :

— Le dimensionnement des transistors (W/L) : en augmentant la longueur du canal (L), on éloigne le drain de la source, l'échange, non voulu, de charges entre les deux zones est donc moins susceptible de se produire.

— La tension de seuil (V_t) : le *Sub-Threshold Current* augmente exponentiellement avec la chute de la tension de seuil ce qui en fait un contributeur important de ce courant.

— La tension d'alimentation (V_{dd}) : elle est liée au potentiel appliqué aux porteurs de charges. Par conséquent, plus elle est haute plus les courants de fuite sont entretenus.

— La température(T) : En augmentant celle-ci, on augmente l'énergie potentielle des porteurs de charge. Comme pour V_{dd}, cela a pour conséquence de faciliter la traversée des différentes barrières de potentiel, et donc les courants de fuite.

Si on appelle $I_{leakage}$ le courant de fuite total dans un circuit intégré, alors, la puissance de *leakage* dissipée suit l'Équation (2.1).

$$P_{leakage} \approx I_{leakage}.V_{dd} \tag{2.1}$$

2.1.1.2 Consommation dynamique

On distingue deux principales sources de consommation dynamique lorsqu'un transistor change de polarité :

— Le courant de court-circuit, appelé *Internal Current*

— Le courant de commutation, appelé *Switching Current*

Courant de court-circuit

C'est le courant qui traverse les réseaux de transistors P et N pendant la période de commutation d'une porte logique. Il ne contribue en rien à la commutation, c'est donc une perte pure d'énergie [1]. Pour mieux identifier comment se manifeste se courant, intéressons-nous à un inverseur CMOS tel que schématisé sur la Figure 2.4.

Prenons le cas d'une transition positive de l'entrée I, associée à la différence de potentiel V_I. Nous considérons également que pour les tensions de seuil $(V_t)_{NMOS} = (V_t)_{PMOS} = V_t$ pour simplifier les équations, mais aussi pour la tension d'alimentation $V_{dd} > 2.V_t$ ce qui est la plupart du temps vrai pour assurer un bon fonctionnement avec une vitesse correcte.

Nous pouvons alors distinguer 3 régions de V_I :

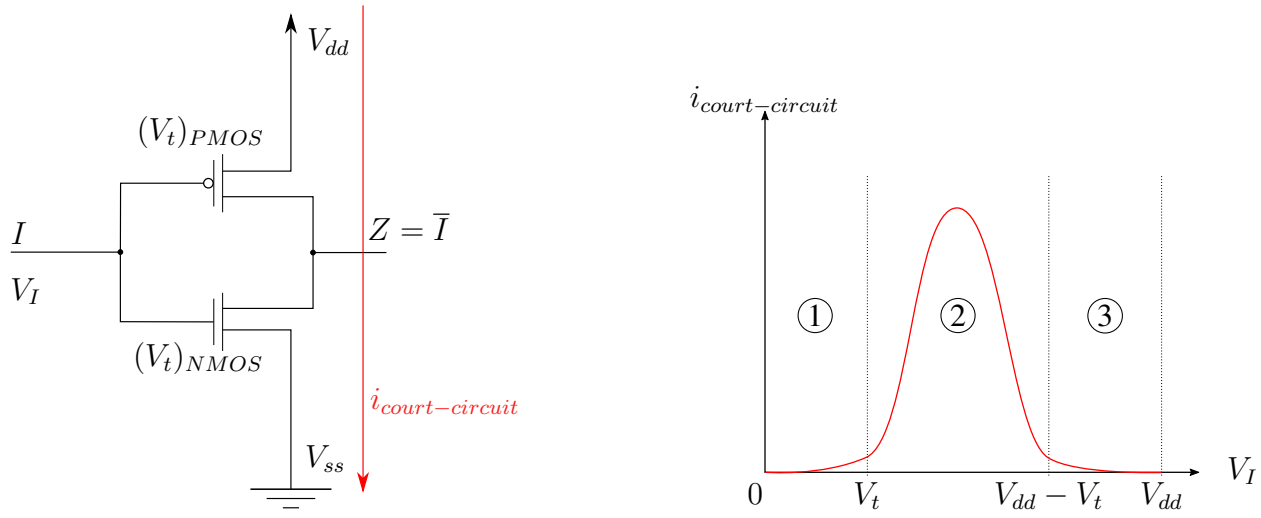


Figure 2.4: Inverseur CMOS, chemin et allure en fonction de V_I du courant de court-circuit

① $V_I < V_t$
Le NMOS ne conduit pas, le PMOS est passant $\rightarrow i_{court-circuit} \approx 0$

② $V_t < V_I < V_{dd} - V_t$
Le NMOS et le PMOS sont tous les deux passants $\rightarrow i_{court-circuit} > 0$

③ $V_{dd} - V_t < V_I$
Le NMOS est passant, le PMOS ne conduit pas $\rightarrow i_{court-circuit} \approx 0$

Pendant la phase (b), nous avons donc un chemin direct qui existe entre V_{dd} et V_{ss}(la masse). La quantité d'énergie perdue pendant cette phase a été mise en équation à de nombreuses reprises dans la littérature. Nous pouvons trouver dans [10] l'équation (Équation (2.2)).

$$P_{court-circuit} = K.\frac{W}{L}(V_{dd} - 2V_t)^3.\tau.f \tag{2.2}$$

Où K est une constante dépendant de la technologie, W/L est le rapport de forme du transistor(largeur sur longeur), τ correspond au temps de montée, ou de descente, du signal d'entré de 0 à V_{dd}, ou inversement, et f représente la fréquence du signal d'entrée I.

Dans cette équation, on remarque que la tension d'alimentation joue, comme pour la consommation de *leakage*, un rôle important puisqu'elle agit comme un facteur au cube. Nous pouvons aussi remarquer que la puissance perdue est aussi un linéaire avec le temps de montée, ou descente du signal. Or, ce τ est lui-même dépendant du temps de transition de la sortie des portes précédentes, ce qui comme nous le verrons dans le prochain paragraphe, est aussi lié à la tension d'alimentation.

Il existe également une dépendance de l'activité de l'inverseur puisque la puissance consommée est linéaire avec f la fréquence du signal d'entrée et un taux d'activité α.

Courant de commutation

Pour faire changer l'état de sortie d'une porte logique, il est nécessaire de charger la capacité de sortie de la porte. Cette capacité résulte :

— des capacités d'interconnexions

— des capacités d'entrée des différentes portes chargées auxquelles la sortie est connectée

Ces contributions sont en général représentées au niveau porte logique par une capacité C_{load} et au niveau circuit par une unique capacité équivalente C_{eq}.

Intéressons-nous au même inverseur que précédemment redessiné avec les chemins de courant dynamique dans la Figure 2.5. Considérons une charge de la capacité de sortie suivie d'une

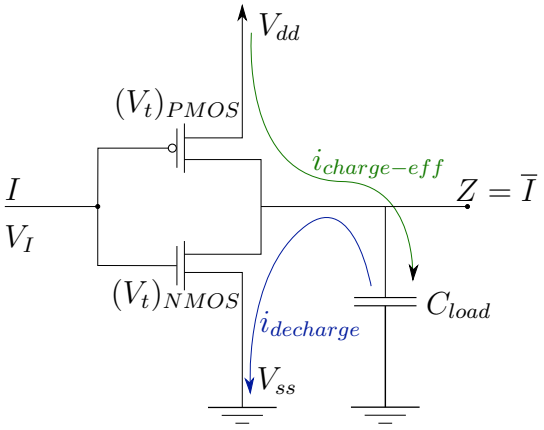


Figure 2.5: Inverseur CMOS, consommations dynamiques

décharge, soit la séquence $I : 1 \rightarrow 0 \rightarrow 1$.

Á l'état initial, le transistor NMOS est passant et le PMOS est en circuit ouvert. Lorsque I passe à 0, l'inverse se produit, le transistor PMOS devient passant alors que le transistor NMOS se bloque. On a donc une charge du condensateur, faite avec un courant i_{charge}. L'énergie consommée se répartit entre l'énergie stockée dans la capacité de charge C_{load}, égale à $\frac{1}{2}.C_{load}.V_{dd}^2$, et l'énergie dissipée thermiquement.

Ensuite, lorsque I repasse à 1, la capacité se décharge avec un courant $i_{decharge}$ et l'énergie contenue dans la capacité repasse à 0. La charge transférée lors de chaque transition est $Q = C_{load}.V_{dd}$. Ainsi, l'énergie d'un circuit s'exprime comme étant une grandeur dépendant du nombre total de commutations A, d'un facteur lié à la capacité équivalente du circuit, C_{eq}, et de la tension d'alimentation au carré, Équation (2.3)

$$P_{switching} = A.C_{eq}.V_{dd}^2 \tag{2.3}$$

Nous pouvons noter que la puissance instantanée peut s'exprimer avec l'Équation (2.4) où α est l'activité moyenne du circuit, *i.e.* le nombre moyen de bit changeant à chaque cycle d'horloge

et f est la fréquence du signal d'horloge

$$P_{switching} = \alpha.C_{eq}.V_{dd}^2.f \tag{2.4}$$

Nous pouvons remarquer une dépendance directe de la capacité équivalente du circuit intégré sur la puissance dynamique. Cependant, il est difficile de modifier la valeur de C_{eq} puisqu'elle vient principalement de paramètres technologiques. Pour ce qui est de l'activité, à première vue, il est difficile de l'améliorer puisque lorsqu'un calcul doit être effectué, le nombre de transistors à activer n'est pas forcément compressible. Nous verrons cependant dans la Section 2.1.2 ce qu'il est possible de faire pour améliorer ce facteur α.

La puissance moyenne est aussi linéaire avec la fréquence d'horloge du circuit. Cela nous permet donc en théorie de baisser la puissance moyenne autant que nécessaire du moment que les performances ne sont pas une contrainte.

Enfin, on remarque que la tension d'alimentation reste le principal facteur de l'équation avec une dépendance au carré. Réduire V_{dd} permet donc en théorie de réduire fortement la consommation dynamique du circuit. Cependant, la vitesse des transistors étant elle aussi dépendante de la tension d'alimentation, cela se fait au prix d'une perte de performances. Dans [22], on trouve une estimation de la valeur de la tension d'alimentation qui conjugue au mieux performance et puissance dynamique $V_{dd} = 3.V_t$ pour une technologie en $0.1\mu m$ avec $V_t = 0.7V$.

2.1.1.3 Récapitulatif des sources de consommation d'un circuit intégré

La puissance totale consommée par un circuit intégré peut se mettre sous la forme de (Équation (2.5)).

$$P_{total} = P_{leakage} + P_{court-circuit} + P_{switching} \tag{2.5}$$

Chacune de ces puissances ont des paramètres sur lesquels il est possible d'influer pour améliorer la consommation totale. La Tableau 2.1 répertorie les différents paramètres ainsi que l'importance qu'ils ont sur chacune des puissances composant P_{total}.

	V_{dd}	V_t	$f\|\alpha$	T	$\frac{W}{L}$
$P_{leakage}$	++	+		++	+
$P_{court-circuit}$	+++	++	+		
$P_{switching}$	++		+		

Tableau 2.1: Influence des différents paramètres sur la puissance totale dissipée dans un circuit intégré

Nous remarquons que V_{dd} est un facteur important pour toutes les puissances dissipées dans un circuit. Influer sur cette grandeur est donc essentiel pour améliorer la consommation globale.

La tension de seuil joue également un rôle important et plus particulièrement pour les consommations de court-circuit et de *leakage* qui restent néanmoins inférieures à la puissance

dynamique.

Deux facteurs importants à considérer sont donc la fréquence de fonctionnement et l'activité α qui influencent de manière sensible la puissance dynamique et de court-circuit. Il est à noter qu'influer sur l'activité permet de réduire l'énergie consommée par le circuit sans influencer les performances.

Enfin, le rapport de forme des transistors W/L, ayant à la fois un impact sur le courant de *leakage* et sur la tension de seuil des transistors à une influence qui est loin d'être négligeable sur la consommation. C'est aussi le seul facteur des transistors sur lequel le designer à un degré de liberté puisque les autres sont directement liés aux procédés de fabrication.

Pour la température, notions qu'il s'agit d'un paramètre environnemental, et qu'il est difficile d'avoir une influence dessus. Il est toutefois possible de limiter la puissance dissipée afin d'éviter une augmentation de la température du circuit et tout risque d'un emballement thermique [38].

2.1.2 Méthodes de design *low-power* classiques

Il existe plusieurs méthodes classiquement utilisées dans l'industrie pour réduire la puissance consommée par les circuits intégrés. Nous avons choisi de les catégoriser suivant les trois grands axes suivant :

— Axe technologique
— Axe système
— Axe microarchitectual

2.1.2.1 Axe technologique

Le levier technologique le plus utilisé par l'industrie pour réduire la consommation des systèmes intégrés est certainement de faire varier la tension de seuil des transistors. Nous avons vu dans Section 2.1.1 que le V_t influence fortement les courants *leakage* et de court-circuit. Dans le cas du *leakage*, il faut faire en sorte d'avoir une tension de seuil la plus haute possible. Pour ce qui est de la tension de court-circuit d'après l'Équation (2.2), il faut que V_t soit le plus proche possible de $\frac{V_{dd}}{2}$.

L'Équation (2.6)) [22] nous indique également que la tension de seuil à un impact sur le temps de traversée, T, des portes logiques.

$$T \propto \frac{V_{dd}}{(V_{dd} - V_t)^2} \tag{2.6}$$

La variabilité de cette tension de seuil sera donc un compromis à faire entre consommation et performances.

Méthode statique, utilisation des librairies multi V_t

Il existe certaines librairies standards de cellules qui proposent deux ou trois implémentations avec des V_t différents. Ces librairies multi-V_t comprennent généralement les types de cellules suivants :

HVT pour *High Voltage Threshold*

RVT/SVT pour *Regular Voltage Threshold* ou *Standard Voltage Threshold*

LVT pour *Low Voltage Threshold*

Deux méthodes sont courantes pour utiliser ces multiples valeurs de V_t.

La première consiste à placer des transistors HVT, communément appelés *sleep transistors*, sur le chemin d'alimentation d'un bloc combinatoire comme présenté Figure 2.6. De cette manière, lorsque les transistors HVT ne conduisent pas, seule la puissance statique des transistors de contrôle est consommée.

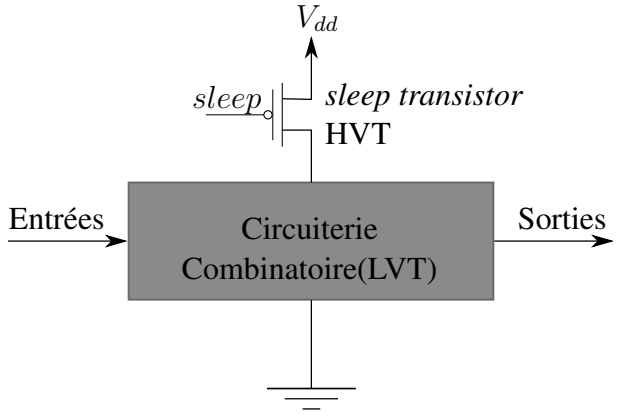


Figure 2.6: Exemple d'utilisation d'un *sleep transistor*

Cette technique présente cependant un désavantage notable [4]. Lorsque le *sleep transistor* est passant, il subsiste à ses bornes une différence de potentiel ce qui a pour effet de diminuer la différence de potentiel totale appliquée au réseau qui effectue le calcul. Ceci a pour effet de diminuer les performances de la partie combinatoire. Pour limiter cette perte de performance, il faut que le transistor choisi soit de grande taille, ce qui implique un surcoût en surface et en puissance consommée. Il est donc primordial de choisir avec parcimonie le nombre et la taille des *sleep transistors*. De plus, la génération du signal $sleep$ ne peut pas être mise en suspens, il faut donc qu'une partie du circuit dédiée à la gestion des signaux $sleep$ soit fonctionnelle à tout instant.

Une autre technique utilisant les différents V_t d'une librairie consiste à optimiser l'utilisation de chaque type de portes en fonction du chemin logique concerné. Dans un circuit synchrone, toutes les données sont capturées au même instant, généralement sur le front montant du signal d'horloge. Ce principe implique une contrainte, que nous appellerons contrainte de *capture*, entre le temps que met le signal à se propager d'un registre à un autre, le temps de *launch* (T_{launch}), et l'intervalle entre deux captures de données, la période du signal d'horloge(T_{clock}).

Nous reviendrons plus longuement sur cette contrainte dans le Chapitre 5 mais en prenant la Figure 2.7 comme exemple, l'Équation (2.7) peut être considérée comme une première approximation.

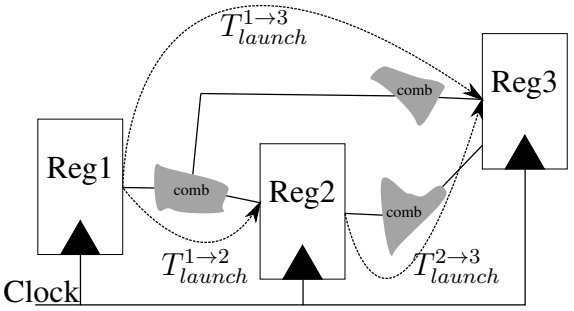


Figure 2.7: Contrainte de capture dans un circuit synchrone

$$T_{clock} > max(T_{launch}^{i\to j}) \tag{2.7}$$

Les performances d'un circuit synchrone sont donc directement liées au temps maximal qu'une donnée met à traverser un chemin de registres à registres. Placer des cellules rapides (LVT) permet alors d'améliorer la performance du circuit, au prix d'un courant de fuite généralement plus élevé dans ces cellules à bas V_t. Symétriquement, certains chemins peuvent être très rapides comparés au chemin critique. Remplacer les portes de ces chemins-là par des cellules HVT, plus lentes mais moins sujettes au courant de fuite permet alors de réduire les courants de fuite sans pour autant perdre en performances.

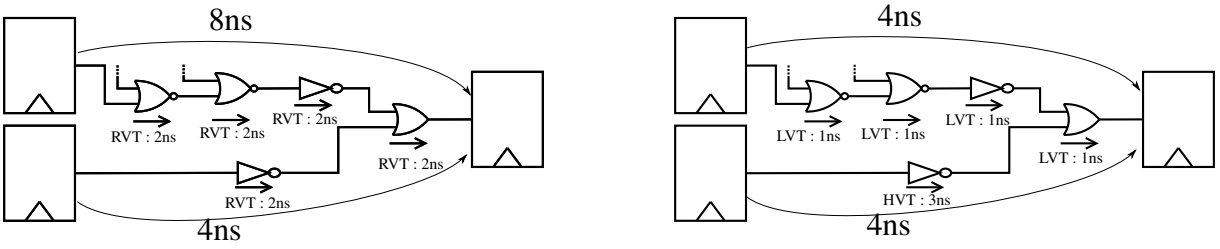


Figure 2.8: Exemple d'utilisation d'une librairie multi V_t

La Figure 2.8 présente une situation dans laquelle il est possible de faire passer le chemin critique de 8 à 4 nanosecondes simplement en remplaçant les types de cellules utilisées sur chaque chemin. Dans cet exemple, une cellule HVT est aussi utilisée dans le chemin rapide (la branche inférieure) sans pour autant détériorer les performances du système. Le principal avantage de cette technique est qu'elle ne demande pas d'effort supplémentaire au niveau design du circuit. Les outils de synthèse sont aujourd'hui conçus pour prendre en compte l'utilisation de ces cellules et proposer des compromis avec lesquels le designer devra faire son choix.

Méthode dynamique, utilisation des librairies avec un V_t variable

Nous n'allons pas ici entrer dans les détails des technologies VTCMOS, pour *Variable Thre-*

shold CMOS. Le principe de fonctionnement est de modifier la polarisation du substrat pour abaisser ou augmenter la valeur de la tension de seuil. Ainsi, pendant les phases de calcul intensif, le V_t est abaissé et donc la rapidité du système est améliorée. Dans les phases de pause, la tension de seuil sera au contraire maintenue haute ce qui permet de grandement limiter la dissipation statique d'énergie. Une implémentation notoire de ce principe a été faite par STMicroelectronics™ grâce à sa technologie FD-SOI [55].

2.1.2.2 Axe système

Il existe également des méthodes s'intéressant aux optimisations du système dans sa globalité, *System On Chip* ou *SoC*. Nous allons ici nous pencher sur quelques une des plus répandues.

Plusieurs domaines d'alimentation à V_{dd} unique, *Power-Gating*

Comme vu précédemment, V_{dd} est un facteur important de la puissance dissipée dans un circuit intégré. Lorsqu'une partie d'un *SoC* est inactive, la garder sous tension engendre alors une consommation statique non négligeable. La technique du *Power-Gating* consiste alors à couper totalement l'alimentation de certains blocs lorsqu'ils sont inutilisés. Ainsi, la puissance statique globale est réduite. Pour s'assurer du bon fonctionnement du système global, il est cependant nécessaire d'isoler chaque ilot pour éviter qu'une donnée, potentiellement corrompue, d'un ilot éteint ne vienne corrompre un module en cours de fonctionnement. Les cellules utilisées dans ce but sont appelées cellules d'isolation, ou *isolation cells*.

Plusieurs domaines de fréquence, statique et dynamique

Dans un *SoC*, tous les composants n'ont pas forcément besoin de fonctionner à la vitesse maximale imposée par l'horloge du système. Par exemple, les périphériques de communication sont généralement plus lents que le cœur opérationnel d'un processeur. Etant donné que la puissance dynamique varie linéairement avec la fréquence, il y a un avantage à réduire la vitesse de certains blocs. Cela revient à créer des domaines de fréquence, ou *frequency islands*, dans le système.

Cette méthode statique peut être étendue pour obtenir une gestion dynamique de chaque domaine de fréquence. Ainsi, chaque domaine fonctionnera à une fréquence déterminée par un gestionnaire de fréquence centralisé. Cette méthode est communément appelée *Dynamic Frequency Scaling* ou *DFS*.

La principale problématique dans ce genre de circuit est la communication entre chacun des domaines de fréquence. Plusieurs solutions existent, passant d'un simple système de requête/acquittement entre deux modules jusqu'à des *Network On Chip*, ou *NoC*, utilisant des protocoles de communication sophistiqués.

Plusieurs domaines d'alimentation avec plusieurs V_{dd}

De la même manière que pour la fréquence, chaque îlot peut être alimenté avec une tension d'alimentation différente correspondant au besoin de performance de chaque module. La réduction des puissances dynamiques et de *leakage* peut être significative étant donnée la forte dépendance de ces dernières avec la valeur de la tension d'alimentation. Cependant, faire communiquer différents niveaux de tension ne se fait pas directement et nécessite l'intervention de releveurs, ou abaisseurs, de tension communément appelés *level shifters*.

Plusieurs domaines de fréquence et d'alimentation dynamiques

La solution mixte d'ilots de fréquence et d'alimentation est potentiellement la solution la plus efficace. Etant donné que les performances d'un circuit intégré dépendent de la tension d'alimentation, il fait sens de réguler cette tension d'alimentation et la fréquence de concert. Cette technique communément appelée *Dynamic Voltage and Frequency Scaling*, ou DVFS, est largement employée dans bon nombre de processeurs commerciaux. Elle permet de tirer le maximum du système lors d'une opération et d'économiser une grande quantité d'énergie pendant les phases ou le calcul est moins intensif. Toutefois, l'utilisation de cette méthode demande l'implémentation d'un contrôleur de tension et de fréquence, des *level shifters* entre les différents domaines de tension et des synchroniseurs aux interfaces des différents domaines de fréquence.

2.1.2.3 Axe microarchitecture

Clock-gating

Le clock-gating est une méthode grandement utilisée dans la conception de circuits intégrés. Elle consiste à ne pas activer à chaque front d'horloge certains registres lorsqu'ils ne sont pas nécessaire au stockage de nouvelles données. Cela permet d'économiser l'activité générée dans la bascule par l'arrivée d'un front montant d'horloge ainsi que l'énergie dissipée dans l'arbre d'horloge. Pour ce faire, on utilise ce que l'on appelle une *clock-gating cell*, Figure 2.9. Le

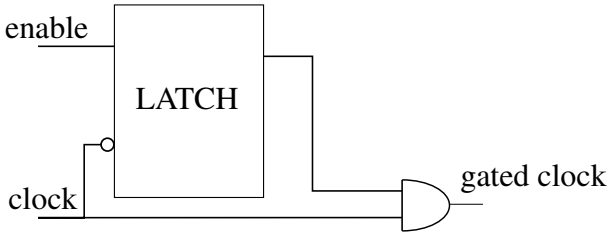


Figure 2.9: Implémentation d'une *clock gating cell*

signal *enable*, dépendant des données dans le circuit, conditionne la traversée ou non du signal d'horloge. Il est possible d'implémenter manuellement le clock-gating, cependant, la technique la plus usitée consiste à laisser les outils de synthèse procéder à l'insertion des cellules adéquates aux endroits stratégiques. Pour aider les outils de synthèse à faire les bons choix, il est nécessaire

de recourir aux «bonnes pratiques» de codage. Pour que les outils puissent inférer correctement un registre avec clock gating on utilise par exemple la syntaxe décrite ci-dessous.

```
module clock_gated_register (
  input wire enable, clk, data_in,
  output reg data_out
);
  always @(posedge clk)
    if (enable)
      data_out <= data_in;
endmodule
```

Evidemment, il n'est pas forcément judicieux d'effectuer du clock-gating sur un seul registre. En effet, l'ajout de la cellule de clock-gating à un coût énergétique. Le compromis n'est pas toujours aussi simple à faire puisque le gain va dépendre des données. Cela demande donc au designer de bien définir sur quels registres il faudra appliquer le clock gating.

Optimisations combinatoires

De nombreuses techniques d'optimisation au niveau combinatoire existent pour améliorer les performances énergétiques des circuits intégrés. Nous n'allons pas toutes les décrire en détails ici. En voici cependant une liste non-exhaustive [63] :

- Réduction de l'activité sur les bus : un bus est en général composé de longs fils, donc d'une capacité conséquente. Il est donc important de limiter les transitions sur ceux-ci au stricte nécessaire. C'est pourquoi il est préférable d'avoir en entrée du bus un registre associé à un signal d'*enable* afin de n'avoir que les transitions pertinentes traversant le bus.
- Partage de ressource : mettre en commun des ressources matérielles, comme un additionneur par exemple, ne permet pas d'économiser de l'activité mais peut néanmoins avoir un impact sur la surface, donc sur le *leakage*
- Eviter les transitions inutiles : certains signaux peuvent changer de niveau soit de manière intempestive, ce qu'on appelle des *glitches*, ou alors sans avoir aucun impact sur la fonctionnalité du système, par exemple une variation de l'entrée non sélectionnée d'un multiplexeur. Dans ce cas il est parfois préférable d'ajouter de la circuiterie pour limiter l'activité générée par ces transitions « inutiles ».
- Utiliser des encodages de données : les codages *One-hot* et de Gray sont deux des plus connus ; ils permettent de limiter le nombre de bits qui changent entre deux valeurs. Leur utilisation est en général recommandée pour les machines d'état où pour l'adressage des mémoires.
- ...

Les techniques permettant d'économiser de l'énergie jouent donc majoritairement sur deux points : la tension d'alimentation et l'activité. Comme nous l'avons vu dans Section 2.1.1, ces deux points sont en effet cruciaux dans l'équation de la consommation totale d'un circuit intégré. Toutes ces solutions de réduction de la consommation viennent cependant avec un certain coût, en général soit en performances, soit en surface mais aussi parfois en complexité d'utilisation. Leur utilisation dépend donc des compromis que le designer est prêt à faire pour améliorer l'efficacité énergétique du système.

Les méthodes présentées sont cependant toutes limitées par le paradigme dans lequel évolue l'environnement de conception : la synchronisation des données par un unique signal d'horloge.

2.2 Généralités sur les circuits asynchrones

Le concept de circuit asynchrone a été introduit par les travaux de David A. Huffman dans les années 50 [26]. Contrairement aux circuits synchrones qui se basent sur un signal de synchronisation global, le signal d'horloge, les circuits asynchrones utilisent eux des mécanismes de synchronisation locaux qui permettent aux différents éléments de communiquer aux moments opportuns [54]. Par rapport au synchrone ou la méthode générale est unique, les façons de concevoir les circuits asynchrones sont multiples. Les différents types de circuits asynchrones sont généralement classifiés de la manière suivante [54] :

— Circuit Insensible aux Délais (DI) : Cette classe de circuit ne suppose aucune hypothèse temporelle d'où la notion d'insensibilité aux délais.
— Circuit Quasi Insensible aux Délais (QDI) : Par rapport aux circuits DI, une hypothèse temporelle a été introduite. Elle concerne les fourches dites isochrones. Pour certaines des fourches présentes dans le circuit, le signal provenant d'une source unique est connectée à plusieurs points dont les instants de transition doivent avoir lieu au même moment. Cette hypothèse peut paraître théoriquement forte et contraignante mais dans la pratique elle est plutôt faible. De plus, il est possible de marquer les fourches isochrones pour une vérification post-layout.
— Circuit indépendant de la vitesse (SI) : Dans ce cas-là, il est supposé que les délais dans les fils sont nuls. Il n'y a pas de condition supplémentaire pour les portes logiques.
— Micropipeline : A la différence des types de circuits précédents où les données et les signaux de contrôle sont en général encodés dans un même jeu de signaux, on a dans les cas des circuits micropipelines des chemins de données et de contrôle différenciés. Le chemin de données est classique, semblable au chemin de donnée d'un circuit synchrone. Le chemin de contrôle est un circuit QDI auquel on ajoute un délai correspondant aux temps de traversée des chemins de données entre deux jeux registres. Le fonctionnement de ce type de circuits sera approfondi dans la Section 2.3.2
— Circuits de Huffman : Ces circuits utilisent les mêmes modèles de délais que les circuits synchrones. Ils supposent tous les délais bornés et connus dans le circuit ce qui permet de déterminer l'instant de capture des données. Ces circuits, en plus d'être difficiles à concevoir, sont aussi très sensibles aux variations de délais liées aux procédé de fabrication.

De manière graphique, on peut représenter ces différentes classes en fonction des hypothèses temporelles nécessaires pour assurer le fonctionnement d'un circuit. Un tel graphe est présenté Figure 2.10.

Cette classification peut aussi se faire en fonction de la complexité d'implémentation. Dans ce cas, le synchrone serait la méthode la plus simple jusqu'à l'implémentation DI qui est la plus complexe. En effet, ces derniers, les circuits DI et QDI, sont en général plus compliqués

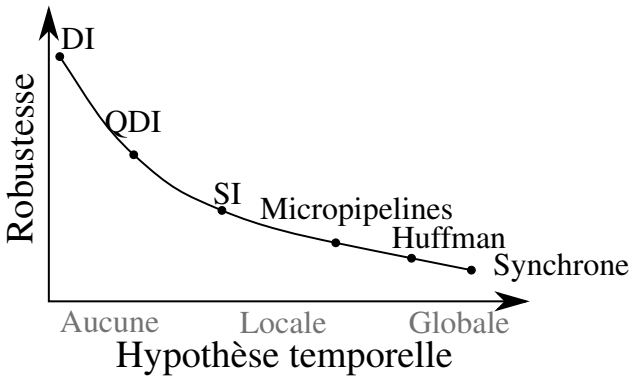


Figure 2.10: Robustesse des classes de circuit asynchrone en fonction de leur hypothèse temporelle [50]

à implémenter que du synchrone étant donné que les codages de donnée insensibles aux délais nécessitent une logique sans aléa produisant des circuits en moyenne deux fois plus gros que son homologue synchrone pour une fonctionnalité logique équivalente. En revanche, ils possèdent une remarquable tolérance aux variations des procédés de fabrication, de tension et de température. Cela est dû à l'absence, ou la quasi-absence, d'hypothèses temporelles contrairement aux circuits synchrones qui, eux, nécéssitent une hypothèse temporelle pire cas(le chemin critique du circuit entier) pour être fonctionnels.

Les travaux présentés, dans ma thèse considèrent des méthodes et des modèles de circuits le plus proche possible des méthodes synchrones existantes afin d'exploiter largement les outils standard utilisés dans l'industrie. Il est par ailleurs important d'améliorer la robustesse des circuits étudiés, notamment à basse tension, et de conserver une complexité d'implémentation raisonnable. L'approche choisie est celle des circuits micropipelines. Nous allons dans la section suivante préciser le cadre du travail ainsi que l'environnement que nous nous sommes fixés.

2.3 Motivation et cible : Micropipelines

2.3.1 Cible du travail, le projet LISA

Cette thèse s'inscrit dans le cadre du projet européen LISA, visant le marché des smartcards pour des applications aussi variées que : l'identité, le bancaire et le transport. La cible est un produit dit « dual », pouvant utiliser soit des plots, soit une technologie *Near Field Communication* (NFC), pour procéder à une opération de communication sécurisée. Les entreprises partenaires de ce projet sont les suivantes :

- Idemia - StarChip, entreprise leader du projet et hébergeant la thèse, spécialisée dans les cartes à puces
- TIMA, laboratoire de recherche implanté à Grenoble, hébergeant la thèse, apportant son expertise en design de circuits asynchrones
- SPS, société spécialisée dans la création et l'intégration d'antennes
- Dolphin Integration, société en charge de fournir des bibliothèques de cellules standards
- Idemia (anciennement Safran-Morpho/Oberthur), sociétée chargée de développer le système d'exploitation pour le produit
- IM2NP, laboratoire de recherche implanté à Marseille apportant son expertise en design analogique

Ce projet vise à réduire fortement la consommation d'un produit sécurisé afin de pouvoir l'intégrer, ainsi que l'antenne l'alimentant, dans une fraction de la surface aujourd'hui utilisée correspondant à la surface du module (où l'on trouve les plots pour les transactions par contact). Cette approche doit permettre une simplification de la procédure d'encartage et une réduction du coût associé. Comme la surface de l'antenne est réduite, pour compenser la perte d'énergie liée à ce choix, l'antenne a dû être optimisée avec un dispositif *booster*. Malgré ce dispositif, comme l'énergie collectée par l'antenne n'est pas suffisante pour une puce conventionnelle, il a été nécessaire d'en concevoir une nouvelle assurant la même fonctionnalité avec une qualité de service identique. Afin de retrouver les spécifications usuelles des cartes à puce, chaque acteur du projet a agi sur la consommation de la puce.

Pour ce qui concerne la partie numérique, la cible était de diviser la consommation par 5 par rapport à une implémentation équivalente dans une technologie précédemment utilisée par StarChip. Pour pouvoir atteindre la cible, plusieurs mesures ont été prises en lien avec les principales sources de consommation d'un circuit intégré vues dans 2.1.1 :

- Réduction de la technologie, le produit utilisera une technologie 55nm
- Utilisation d'une technologie *low-power*, conçue pour être utilisée à 0,9V
- Utilisation d'une librairie aux multiples tensions de seuil, trois niveaux, HVT, RVT et LVT
- Utilisation de technologies asynchrones

L'utilisation des technologies asynchrones est envisagée pour pouvoir réduire la consomma-

tion d'un système intégré. Cependant, l'idée n'était pas de recréer un circuit asynchrone en ne partant de rien.

La conception asynchrone, parmi ses atouts offre multitude de solutions, qui mal maîtrisée, peut constituer un frein et une difficulté de premier plan pour son adoption.

A contrario, ce champ des possibles permet pour chaque problématique de trouver la solution la plus adéquate. Cette non-standardisation empêche probablement les méthodologies asynchrones d'être majoritairement présentes dans le monde de l'industrie.

Il existe un certain nombre d'expériences et applications fructueuses plus ou moins récentes : ILLIAC [43], ATLAS [33], AMULET [20], MiniMIPS [37], LOIHI [17], ...Ces applications ont en outre nécessité des experts de la conception asynchrone pour voir le jour. Comparé au nombre de circuits synchrones commercialement disponibles, les succès industriels de circuits asynchrones restent minoritaires.

Cette thèse vise finalement à simplifier les premiers pas d'un concepteur de circuits intégrés synchrones dans le monde des circuits asynchrones. Les choix considérés dans la suite sont fait dans l'optique de faciliter la compréhension des problématiques asynchrones sans s'encombrer d'un haut niveau de complexité.

En utilisant l'expertise de l'entreprise en implémentation de systèmes sécurisés et l'expertise du laboratoire TIMA sur les circuits asynchrones, l'axe central de ce travail sera la conversion de systèmes synchrones déjà existants en systèmes asynchrones ayant la même fonctionnalité.

Pour une première approche, nous avons choisi les circuits Micropipeline car ils sont proches des circuits synchrones. Dans la suite, nous allons nous intéresser aux caractéristiques des circuits micropipelines qui les rendent compatibles avec notre méthode.

2.3.2 Micropipelines

2.3.2.1 Généralités

La notion de circuits micropipelines a été introduite par Ivan E. Sutherland [57]. Ses travaux mettent en avant les limitations de la méthode synchrone en particulier dans le cas de pipelines où de mémoire *First-In, First-Out*, aussi appelées FIFO. Dans le cas de cette dernière par exemple, une mémoire FIFO étant élastique par principe, la méthode synchrone impose l'intégration d'une circuiterie combinatoire pour indiquer quels éléments de la mémoire sont occupés ou non. De plus, Sutherland soulève aussi une autre difficulté : si l'entrée et la sortie de la FIFO sont contrôlées par deux horloges différentes alors la structure présente un risque de métastabilité. Tout au plus le designer pourra s'assurer statistiquement de rendre le temps moyen entre deux erreurs, ou MTBF (*Mean Time Between Failure*), suffisamment grand pour considérer ce risque comme très improbable.

La notion de micropipeline prend alors tout son sens puisque dans ce cas. Lorsqu'une donnée arrive en entrée de la FIFO, elle va alors se propager naturellement de place vide en place vide

jusqu'à être bloquée par une donnée déjà présente. À l'inverse, lorsqu'une donnée sort de la FIFO, elle libère alors une place en bout de FIFO et naturellement les autres données présentes dans le pipeline vont se décaler d'une place vers la sortie.

On distingue deux parties dans les circuits micropipelines :

— Le chemin de données : c'est un chemin de données classique tel que l'on peut le trouver entre deux étages de pipeline dans une implémentation synchrone. Il n'y a pas de condition particulière sur ce chemin-là.

— Le chemin de contrôle : un circuit synchrone utilise un signal de synchronisation global, le signal d'horloge. Dans le cas des circuits micropipelines, l'arbre d'horloge est remplacé par un contrôleur qui gère les communications entre les différents étages de registres. Ce contrôleur est généralement de la classe quasi-insensible aux délais(QDI) auquel on ajoute des délais permettant de de garantir les hypothèses temporelles locales faites dans les chemins de données.

La Figure 2.11 représente la différence entre un pipeline synchrone et un micropipeline. L'arbre d'horloge habituellement utilisé par les circuits synchrones est dans le cas asynchrone remplacé par des cellules de contrôle asynchrones qui sont reliées entre elles via des canaux de communication Section 2.3.2.2.

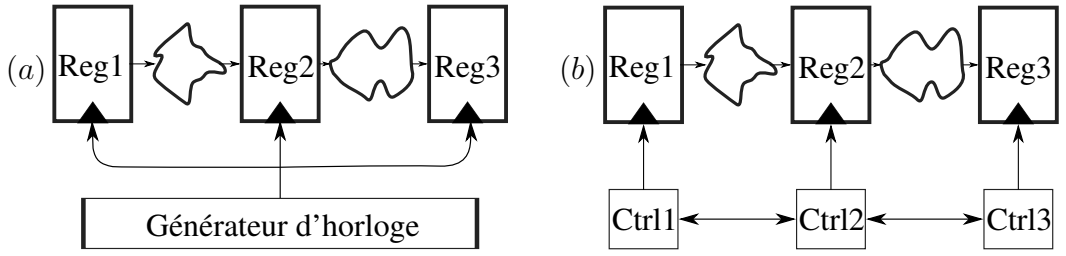


Figure 2.11: Représentation simplifié d'un pipeline synchrone(a) et d'un micropipeline(b)

Nous pouvons également remarquer sur ce schéma que le chemin de données, au travers des registres, n'a pas été modifié. C'est là le principal atout de ce type d'implémentation pour notre travail car nous ne modifions pas le chemin de données existant. Étant donné qu'un grand nombre de circuits existent déjà au sein de l'entreprise, il serait trop long et fastidieux, dans un premier temps en tout cas, de les reconcevoir de manière asynchrone.

Transformer les circuits existant en circuits micropipelines devrait nous apporter les avantages suivants :

— Réduction de l'activité, le micropipeline n'est actif que lorsqu'une donnée est fournie en entrée. Le reste du temps, le circuit est naturellement au repos et ne consomme quasiment rien, à la consommation de *leakage* près. Un circuit synchrone, sans *clock-gating*(2.1.2.2), aurait une consommation inutile avec l'activité de l'arbre d'horloge et des registres.

— Performances en temps moyen. Dans un circuit synchrone, nous avons vu avec les Figure 2.7 et Équation (2.7) que les performances sont limitées par le chemin combinatoire

le plus long du circuit entre deux registres que l'on appelle chemin critique. Dans le cas d'un pipeline élastique, cette notion de chemin critique existe toujours mais seulement localement entre les registres constituant un étage du pipeline. Étant donné que chaque étage communique seulement avec ses voisins directs, seuls ces chemins sont déterminants vis à vis de la contrainte temporelle. En prenant l'exemple de la Figure 2.11 et en considérant les temps de traversée des données suivant :

— $T_{Reg1 \rightarrow Reg2} = 10ns$

— $T_{Reg2 \rightarrow Reg3} = 5ns$

Alors une donnée aurait besoin de $2.max(T_{Reg1 \rightarrow Reg2}, T_{Reg2 \rightarrow Reg3}) = 20ns$ pour traverser le pipeline synchrone. Pour le micropipeline par contre, vu que chaque étage est contraint indépendamment, le temps pour traverser le pipeline serait de $T_{Reg1 \rightarrow Reg2} + T_{Reg2 \rightarrow Reg3} = 15ns$. Avec une contrainte locale plutôt que globale on a des performances dépendant du temps moyen entre deux registres plutôt que du chemin critique global.

Néanmoins, la performance n'étant pas la cible de notre travail, en réduisant la tension d'alimentation on peut réduire la vitesse du circuit asynchrone pour ramener ses performances à un équivalent synchrone. Or nous avons vu dans la 2.1.1.3 que la tension d'alimentation impacte très fortement la consommation totale d'un circuit intégré. Un gain modeste en performances pourrait donc se traduire par un gain de puissance significatif.

2.3.2.2 Canal et protocole de communication

Des canaux de communication sont utilisés pour synchroniser les éléments asynchrones afin qu'ils puissent échanger des données.

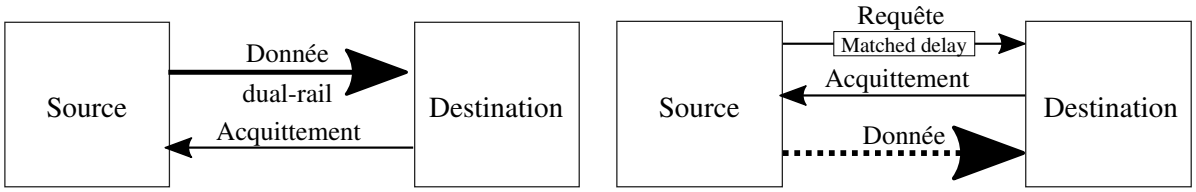
De chaque côté du canal, on trouve d'une part un module source et d'autre part un module destination. Dans notre cas, la source est celle qui initiera la communication avec une requête, et la destination le composant qui signalera que la communication est effectuée avec un acquittement. Il est à noter qu'il est possible d'échanger les rôles de la source et de la destination. Dans ce cas, cette dernière sera à même d'initier la communication.

On retrouve classiquement deux grandes familles de canaux représentés graphiquement dans la Figure 2.12 :

— Les canaux multi-rails, ou *1-of-N encoding* : Ces canaux utilisent plusieurs fils pour transporter les données. Une application classique est l'usage des canaux dual-rail, *i.e.* qui utilisent deux fils par bit de donnée. L'avantage de cet encodage est que les fils de données et de requête sont les mêmes. La validité de la donnée est donc directement encodée dans le canal ce qui en fait un protocole souvent utilisé dans les circuits QDI [62].

— Les canaux à données groupées, ou *bundled-data channel* : Ils sont composés de deux

signaux de contrôle, requête et acquittement, et de n-bits de données *single-rail*. La principale contrainte de ce type de canal est que le signal de requête doit être retardé d'un délai, appelé *matched-delay*. Ce délai compense le temps de calcul de la donnée afin de s'assurer que la donnée soit disponible en entrée du module destination avant que la requête n'arrive. En effet, la requête sert de signal d'échantillonnage des données dans les mémoires (*flip-flops* ou *latches*). Ce type de canal n'ayant pas besoin de codage particulier sur les données, permet de réutiliser les chemins de données existant dans les circuits synchrones [27].



(a) Canal dual-rail *(b)* Canal Bundled-Data

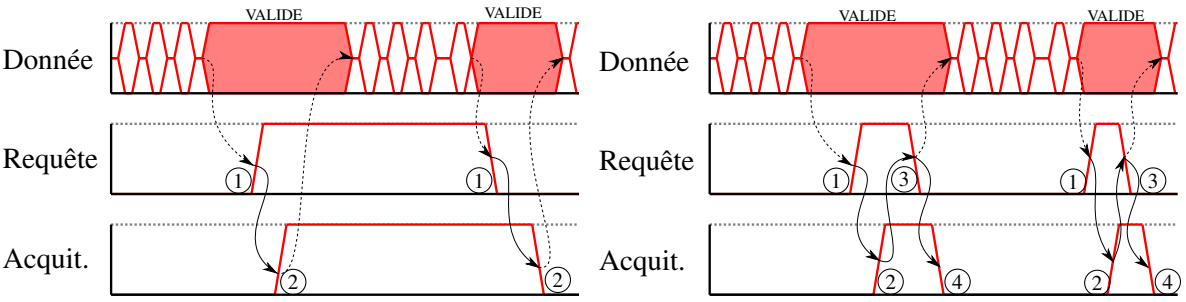
Figure 2.12: Types de canaux de communication asynchrone

L'utilisation d'un canal dual-rail n'est pas envisageable dans un premier temps puisque, comme cela a été expliqué dans la Section 2.3.1, le but est de réutiliser des circuits existant en modifiant le moins possible le chemin de données. Notre choix se portera donc sur les canaux bundled-data.

Les canaux de communication servent de support à la communication, afin de transmettre des données entre une source et une destination, pour lesquels il faut implémenter un protocole de communication.

Les protocoles que nous pouvons considérer avec les canaux à données groupées sont représentés dans la Figure 2.13 et décrits ci-dessous :

— Protocole 2 phases, ou *Non-Return to Zero* (NRZ)
  Avec ce protocole un front représente un évènement qu'il soit montant ou descendant. Dans l'ordre, nous avons donc :
  — La donnée est valide et un front se produit sur le signal de requête ①
  — Un front se produit sur le signal d'acquittement ②, la donnée est consommée et une nouvelle transaction peut démarrer.
— Protocole 4 phases, ou *Return to Zero* (RZ)
  Dans le cas de ce protocole, le signalement est effectué sur niveau haut. Le déroulement d'une communication est donc le suivant :
  — Avec une donnée valide, le signal de requête passe au niveau haut ①
  — Le signal d'acquittement passe au niveau haut ②
  — Le signal de requête retourne à un niveau bas ③
  — Le signal d'acquittement retourne à un niveau bas ④. Une nouvelle transaction peu démarrer.



(a)Protocole 2-phases (b)Protocole 4-phases

Figure 2.13: Chronogramme des protocoles de communication 2-phases(a) et 4-phases(b)

Le protocole 2 phases possède des avantages qui pourraient nous aider à réduire l'activité. En effet, étant donné que chaque transition représente un évènement, l'activité dans le circuit de contrôle devrait être divisée par 2 comparée à une approche 4-phases. Cependant, utiliser une approche 2-phases demanderait de changer le paradigme de conception du chemin de données, où *datapath*, pour passer à un circuit utilisant des *latches* plutôt que des *flip-flops*. Pour conserver l'approche de simplification de la démarche de désynchronisation, nous utiliserons dans la suite uniquement le protocole 4-phases.

2.3.2.3 Réseaux de Petri et graphes de transition de signaux

La conception asynchrone s'appuie fortement sur l'utilisation de représentations graphiques. Les deux principales sont certainement les réseaux de Petri, ou *Petri net*, et les graphes de transition de signaux, ou *Signal Transition Graphs*(STG), qui sont des dérivés des premiers.

Ce travail n'a pas pour but de complètement décrire ces deux outils mais nous allons néanmoins nous intéresser à leur définition et leur utilisation.

Intéressons nous tout d'abord, aux réseaux de Petri. Ils sont définis à partir de 6 éléments : (P, T, F, M_0, W, K) [9] :

- P est l'ensemble des places du réseau
- T est l'ensemble des transitions
- F représente les arcs transition→place, ou place→transition. Un arc ne peux pas connecter une transition vers une autre transition, ou une place vers une autre place.
- M est le marquage réseau, il est en général représenté par une matrice qui référence le nombre de jetons dans chaque place.
- W est l'ensemble des poids des différents arcs $f \in F$. Ce poids est un entier $w \in \mathbb{N}$ représentant le nombre de *token* consommés pour un arc place→transition, le nombre de *tokens* produits pour un arc transition→place
- K correspond à la limite de capacité de chacune des places. Chaque place $p \in P$ à une limite de capacité $k \in \mathbb{N}$

Un réseau de Petri est en général représenté par un graphe bipartie, place/transistion, comme

dans la Figure 2.14. Nous considérerons pour l'exemple que les arcs non annotés seront de poids 1 et que toutes les places ont une capacité infinie.

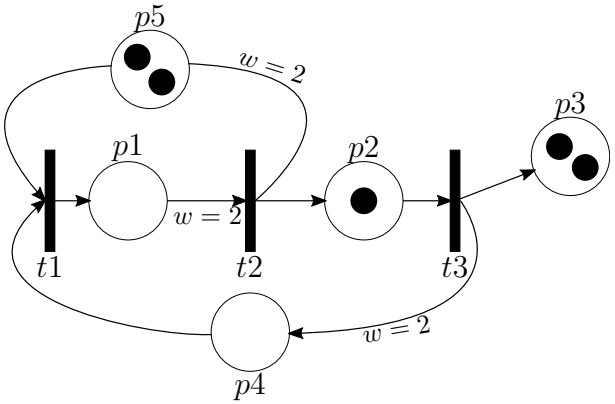


Figure 2.14: Exemple de réseau de Petri

Les *tokens*, ou jetons, dans le réseau de Petri peuvent se déplacer de place en place en franchissant, ou tirant, les transitions. Pour qu'une transition puisse être franchie il faut que chaque place d'entrée de la transition contienne autant de jetons que les poids respectifs de chacun des arcs place→transition. Lorsqu'une transition est activée, un ou des *tokens* sont produits dans chaque place en fonction du poids de l'arc transition→place.

Une restriction supplémentaire est ajoutée lorsqu'une place p à une capacité finie k : les transitions précédant p ne peuvent être activées uniquement si la capacité restante $k(p) - M(p)$ est inférieure ou égale au poids de l'arc arrivant. Cette règle présentée dans [42] est nommée règle de transition stricte.

Intéressons-nous à la propagation de *tokens* dans le réseau de Petri :

— Dans l'état initial présenté dans la Figure 2.14, une seule transition peut être activée : $t3$. Cette activation résulte en un *token* supplémentaire dans $p3$ et deux *tokens* dans $p4$. La Figure 2.15 présente le résultat de ce processus.

— Maintenant que des *tokens* sont présents dans $p4$, la transition $t1$ peut être activée. Un *token* de $p5$ et un *token* de $p4$ sont consommés, produisant un *token* dans $p1$. Le résultat

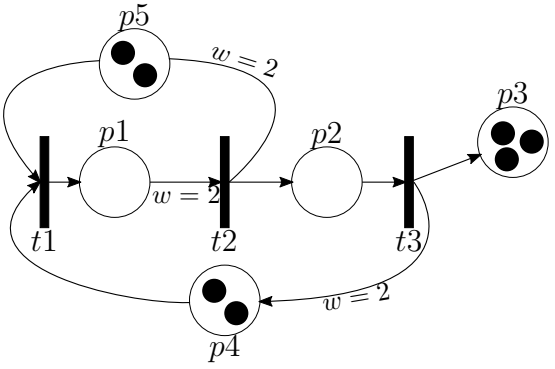


Figure 2.15: Exemple de réseau de Petri après activation de $t3$

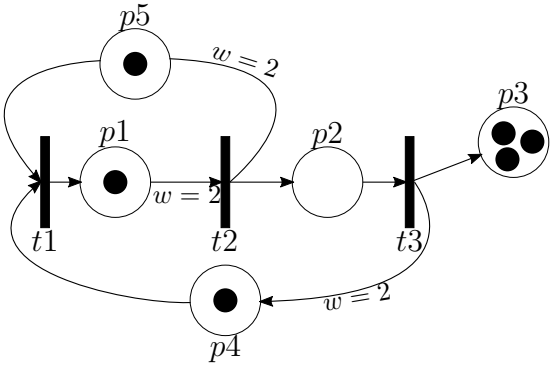


Figure 2.16: Exemple de réseau de Petri, activation de $t1$

de cette transition est le réseau de la Figure 2.16.

— Après ces phases, la transition $t1$ reste la seule à pouvoir être activée puisque l'arc $p1 \rightarrow t2$ a un poids de 2 et nécessite donc 2 *tokens* dans $p1$ pour être activée.

— Une fois deux *tokens* présents dans $p1$, $t2$ peut être activée. On se retrouvera alors dans la situation initiale à la différence près d'avoir un *token* de plus dans $p3$

Les réseaux de Petri peuvent être utilisés pour décrire des séquences d'événements physiques.

En utilisant cette modélisation,nous pouvons faire correspondre les transitions aux événements qui ont lieu dans le circuit. Les places, et leur marquage, représentent l'état global du circuit. Elles traduisent donc les conditions d'activation pour les différentes transitions, ou événements, dans le circuit.

Ainsi, si toutes les places en amont d'une transition sont pleines, toutes les conditions sont remplies pour que l'événement associé à cette transition fasse changer l'état du système.

Pour que les réseaux de Petri soient utilisables comme modèles de nos systèmes asynchrones, il faut généralement considérer les propriétés suivantes :

— *Safeness*, un réseau de Petri est dit *safe*, ou sûr, lorsqu'à partir d'un marquage initial M_0, tout marquage atteignable ne contient qu'un seul *token* par place.

— *Liveness*, un réseau de Petri est dit vivace, si pour un marquage initial M_0 il ne peut pas apparaître de situation de bloquage où *deadlock*.

— *Free-Choice*, chaque place ne peut avoir qu'un seul arc entrant et un seul arc sortant.

Assurer ces éléments permet alors d'utiliser une représentation plus adéquate des circuits : les graphes de transition de signaux, ou Signal Transition Graph(STG). Ces graphes sont des *Petri nets* interprétés où les places ne sont pas représentées et seulement les transitions, montantes$(+)$ ou descendantes$(-)$ des différents signaux sont indiqués.

Cette représentation à l'avantage de décrire plus intuitivement les séquences d'événements dans un circuit. De plus, des travaux ont été menés pour synthétiser des circuits indépendants de la vitesse, généralement considérés comme QDI, respectant la spécification de ces STG. La plus notable est certainement Petrify introduit par Cortadella *et al.* dans [13]. La Figure 2.17, est un STG décrivant un système à deux entrées,a et b, et une sortie c [12]. Obtenir une implé-

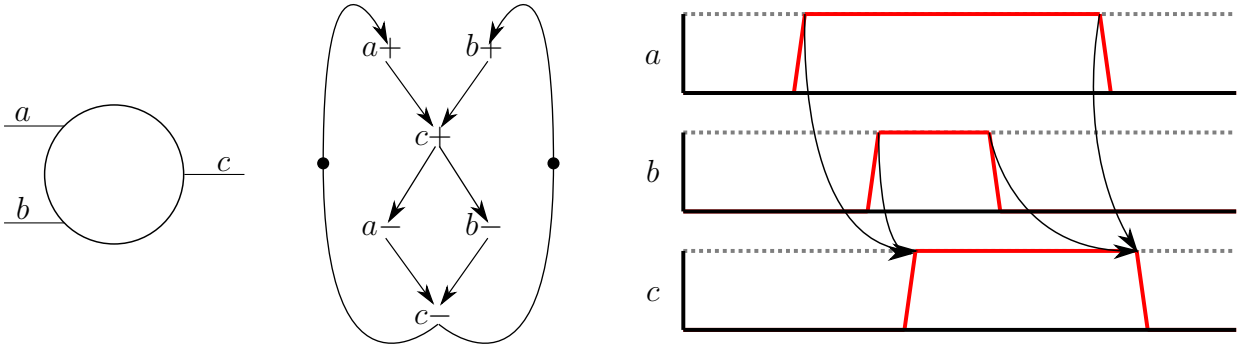


Figure 2.17: Représentation schématique, Signal Transition Graph, et chronogramme associé d'un C-element, ou cellule de Muller

mentation de cette cellule est finalement directement possible avec Petrify. L'équation logique à implémenter donnée par ce STG est celle dans l'Équation (2.8).

$$c = c.(a + b) + a.b \tag{2.8}$$

Ce circuit n'a pas été pris au hasard, il s'agit en effet d'un C-element, ou porte de Muller, introduite par Muller dans [41]. Cette cellule est à la base d'un grand nombre de constructions asynchrones. Nous nous pencherons plus longtemps sur ce circuit dans la Section 3.3.

2.3.2.4 Spécification et architecture des contrôleurs

En définissant un STG pour un protocole particulier, il sera donc possible de générer un circuit implémentant de manière sûre celui-ci.

En adoptant cette démarche, les travaux de Furber et Day dans [19] présentent différents types de contrôleurs 4-phases pour des pipelines utilisant des *latches*. Nous allons adapter la méthode utilisée dans leurs travaux à notre cas, c'est-à-dire à l'utilisation de bascules D. La première étape dans la création d'un contrôleur est de spécifier les comportements qui doivent être observés.

La Figure 2.18 présente la boite noire utilisée dans un premier temps pour modéliser un contrôleur [19].

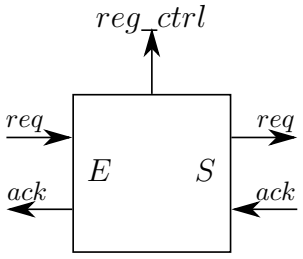


Figure 2.18: Contrôleur de registre, vue abstraite

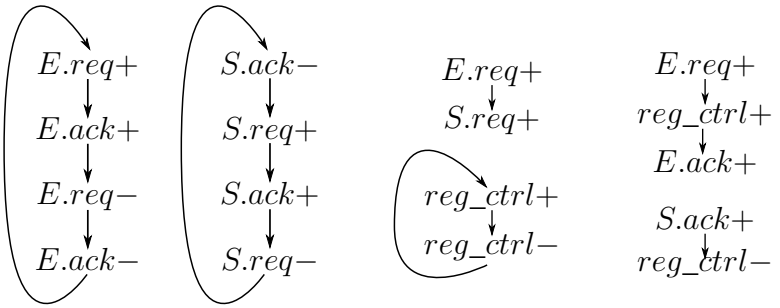


Figure 2.19: Fragments de STG à respecter pour construire le contrôleur 4-phases

La cible de notre démarche est de contrôler une bascule D avec ce contrôleur, il faut donc un signal qui devra servir d'horloge locale sortant de ce contrôleur, nous l'appellerons reg_ctrl. Ensuite, il faudra interfacer ce module avec d'autres afin de créer un contrôleur « global ». On a donc besoin de définir un un canal d'entrée E et d'un canal de sortie S. Bien que les canaux représentés ne fassent pas apparaître les données, ce sont bien des canaux *bundled-data*.

Plusieurs séquences d'événements sont à décrire. Elles sont spécifiées dans les fragments de STG présentés dans la Figure 2.19. De gauche à droite et de haut en bas, on retrouve dans ces fragments :

— Une montée de la requête en entrée déclenche la montée de l'acquittement sur le même

canal. De la même manière, un passage au niveau bas sur la requête déclenche une descente du niveau de l'acquittement d'entrée.

— La transition basse de l'acquittement de sortie permet la propagation d'une nouvelle requête en sortie. Cette dernière provoque une montée de l'acquittement de sortie qui fait à son tour tomber la requête de sortie.

— Une requête en entrée déclenche une requête en sortie.

— On veut un échelon sur le signal de contrôle, donc on a une relation cyclique entre les fronts montants et descendants.

— Une requête en entrée fait monter le signal de contrôle avant de faire monter l'acquittement d'entrée.

— Le retour d'acquittement en sortie fait retomber le signal de contrôle du registre.

Pour construire le STG global du contrôleur, il suffit alors d'assembler les différents fragments en s'assurant que toutes les séquences décrites soient respectées. Naturellement, il existe plusieurs solutions pour le protocole considéré, ces différentes variantes permettent plus ou moins de performances en fonction de l'application mais souvent au prix d'une complexité accrue.

Dans [19], Furber et Day présentent un certain nombre de ces variantes de protocole. Par la suite de nombreux travaux ont porté sur ces protocoles et les avantages que chacun d'eux peuvent apporter. Nous pouvons par exemple trouver des contrôleurs implémentant des mécanismes de précharge [36], d'autres utilisant différents niveaux de concurrence dans les travaux de Furber et Day, ou encore des contrôleurs permettant d'économiser de la surface, et de gagner en performances, en mettant en commun des délais [51].

Dans cette thèse, nous conserverons l'approche la plus simple possible présentée par Furber et Day dans leurs travaux. Le STG complet obtenu en assemblant les différents fragments décrits précédemment est dessiné sur la Figure 2.20. L'implémentation insensible aux délais est celle décrite dans Figure 2.21.

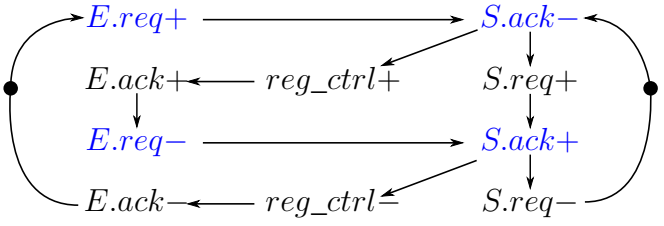


Figure 2.20: STG complet d'un contrôleur 4-phase simple [19]

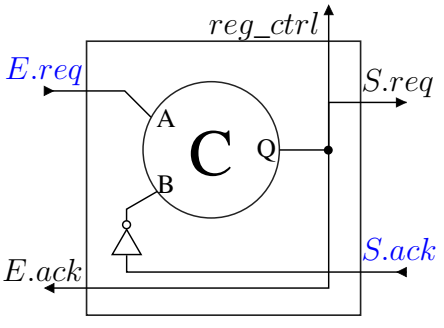


Figure 2.21: Implémentation du contrôleur simple

Nous pouvons remarquer que cette implémentation utilise un C-element dont le STG et l'équation logique ont été présentés précédemment. La fonction de rendez-vous implémentée

par cette dernière permet en effet de synchroniser dans ce cas une requête en entrée, $E.req$, tout en s'assurant que la sortie soit libre, *i.e.* $S.ack$ soit bas. Lorsque c'est le cas, alors la donnée est enregistrée, reg_ctrl et la requête est transférée à la suite, $S.req$, et le canal d'entrée est notifié du traitement de la donnée, $E.ack$.

2.4 Précédents travaux sur la désynchronisation

Obtenir un circuit asynchrone depuis une spécification synchrone a été envisagé par la communauté scientifique depuis de nombreuses années. Citons quelques uns de ces travaux :

- Kessels *et al.* dans [29] : Ce travail porte sur la synchronisation de multiples domaines d'horloge dans les circuits connus sous le nom de GALS, Globally Asynchronous Locally Synchronous. Les auteurs partent du fait que la synchronisation de différents domaines d'horloge peut s'avérer compliquée en raison des risques de métastabilité que cela peut engendrer (nous nous intéresserons à la métastabilité plus longuement dans la Section 3.3.6.1). Au lieu d'utiliser des synchroniseurs classiques, l'idée développée est plutôt de faire communiquer ensemble les différents domaines d'horloge via un protocole par poignée de main. L'utilisation d'un langage non standard dans l'industrie, Tangram, rend la méthode difficilement appréhendable par les designers synchrones.
- Linder *et al.* dans [35] : L'idée de ce travail est de partir d'une spécification synchrone et de remplacer toutes les portes logiques par un élément de synchronisation utilisant un encodage insensible au délai. Cette technique demande cependant de modifier fortement le chemin de donnée et produit en général un impact conséquent en termes de surface.
- Theseus Logic dans [34] : Dans ce document, les auteurs proposent une méthode permettant de synthétiser directement une spécification synchrone et ensuite d'opérer une transformation directe en circuit asynchrone. Pour cela, dans la *netlist* générique chaque cellule est remplacée par un autre d'un bibliothèque spécifique qui comportent des portes à seuil. L'intérêt de cette méthode est qu'elle permet d'obtenir un circuit asynchrone QDI mais l'implémentation des acquittements ne se fait pas complètement automatiquement.
- Blunno *et al.*([8]), proposent une solution automatique qui synthétise une spécification Verilog en micropipeline. L'outil développé, que les auteurs ont nommé *pipefitter*, à besoin que la spécification Verilog soit compatible avec lui et demande donc une adaptation des designs existants et surtout un apprentissage d'un sous ensemble de language habituellement non synthétisable.
- Cortadella *et al.*([7,14]), présentent une méthode de désynchronisation utilisant uniquement des outils de conception classique. Ils s'appuient sur des graphes de type réseau de Petri pour prouver que la désynchronisation conserve une équivalence de trace, *i.e.* avec des séquences de données d'entrées équivalentes, le système synchrone et le systèmes désynchronisé fournissent la même séquence de sortie. Cette solution est très proche

de ce que l'on veut obtenir, mais nous pouvons distinguer deux points qui nous posent encore problème pour notre application :

— L'utilisation de *latches* maître-esclave au lieu de bascules standards : Cela présente en effet un certain avantage puisqu'il est connu ([11]) qu'en arrangeant ces deux *latches* dans le chemin de donnée, il est possible de gagner en performances. C'est ce qu'on appelle plus communément du retiming. Cependant, le *latch-based* design n'est généralement pas géré nativement par les outils et est par exemple difficilement utilisable sur FPGA où les registres sont des bascules indivisibles.

— L'utilisation d'un contrôleur complexe : Le contrôleur préconisé dans les travaux de Cortadella et collab. à été optimisé pour la désynchronisation. Il est cependant relativement compliqué et difficile à appréhender pour un néophyte du design asynchrone. Pour simplifier la démarche, nous nous contenterons dans ce travail d'utiliser le contrôleur le plus simple possible même si avec son utilisation vont venir s'ajouter des désavantages que nous détaillerons plus loin dans le Chapitre 4.

Pour conclure ce chapitre, cette thèse va s'articuler autour de deux articles fondateurs : [14,57] à partir desquels nous allons extraire une méthode de modélisation puis de désynchronisation des systèmes synchrones. Nous terminerons avec l'application de plusieurs variantes de la méthode de désynchronisation proposée, à plusieurs designs existants, afin de comparer les avantages et inconvénients rencontrés en fonction des spécificité architecturales des circuits de base.

Chapitre 3

Modélisation pour la désynchronisation

Dans le cadre de cette thèse, nous cherchons à transformer des circuits synchrones existants en circuits asynchrones micropipelines. Le principe de cette « désynchronisation » repose sur le remplacement de l'arbre d'horloge du circuit synchrone par des contrôleurs qui activeront indépendamment chaque registre. Afin de faire communiquer les différents contrôleurs entre eux, il est nécessaire de connaitre les dépendances de données entre chaque point de mémorisation.

Nous avons déjà vu dans la Figure 2.21 comment nous allons remplacer la connexion au signal d'horloge des bascules D (*flip-flops*). Afin de créer le contrôleur global du circuit, et de s'assurer de l'équivalence entre les circuits synchrones initiaux et leurs implémentations désynchronisées, nous allons utiliser une représentation sous forme de réseaux de Petri.

Un troisième niveau de représentation, proche de l'implémentation en portes logiques, sera utilisé pour décider de la meilleure stratégie de placement des délais afin d'optimiser les performances et l'efficacité énergétique.

Les trois modélisations décrites seront :

— Réseau de registre ou Register Network(RN), la représentation des liens entre les différents registres du système.

— Réseau asynchrone ou Asynchronous Controller(AN), un réseau de Petri modélisant le contrôleur asynchrone.

— Réseau du contrôleur ou Controller Network(CN), une représentation plus proche du niveau portes du contrôleur asynchrone.

Ces niveaux de représentation se déduisent dans l'ordre présenté et ont chacun leur importance dans les étapes du flot que nous décrirons dans le Chapitre 4.

3.1 Le réseau de registre : Register Network (RN)

Le réseau de registre peut être représenté de deux manières différentes :

— Sous forme de graphe

— Sous forme matricielle

Cette représentation a pour but de tracer les différentes dépendances de données. Sous la forme de graphe, chaque registre est représenté par un carré aux bords arrondis et chaque lien entre registre est représenté par un arc dirigé ; un exemple est donné sur la Figure 3.1.

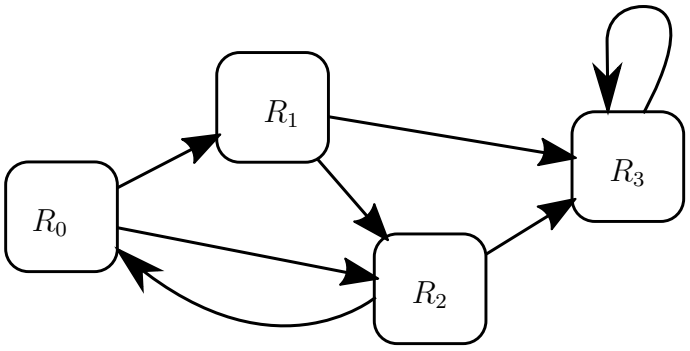


Figure 3.1: Exemple de réseau de registre, ou RN

Cette représentation graphique permet de facilement appréhender l'architecture du circuit mais aussi de pouvoir visuellement identifier la présence de boucles qui, comme nous le verrons par la suite, impliquent la prise de quelques précautions.

La seconde forme de RN est une forme matricielle. Elle sera utilisée dans le Chapitre 4 pour effectuer des opérations sur le réseau de registres et est définie en utilisant les notations suivantes :

— $\mathbb{R}egs$ l'ensemble, borné de cardinal n, des registres R_i d'un circuit avec $i \in (\mathbb{N} \cap [0, n[)$

— Si un registre R_j est dans le cône combinatoire (*fanout*) de R_i, la dépendance de R_j vis à vis de R_i est notée $R_i \longmapsto R_j$

— La matrice du RN, M_{RN} est une matrice carrée $n \times n$ et la valeur de chacun de ses éléments est défini de la manière suivante :

$$\forall (i,j) \in (\mathbb{N} \cap [0,n[)^2, (M_{RN})_{i,j} = \bullet \Leftrightarrow R_i \longmapsto R_j$$

Avec cette définition, on obtient pour la Figure 3.1 la représentation matricielle dans la Figure 3.2.

$$\begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \\ R_3 \end{array} \begin{array}{c} \begin{array}{cccc} R_0 & R_1 & R_2 & R_3 \end{array} \\ \begin{bmatrix} & & \bullet & \\ \bullet & & & \\ \bullet & \bullet & & \\ & \bullet & \bullet & \bullet \end{bmatrix} \end{array}$$

Figure 3.2: Représentation matricielle du RN présenté dans la Figure 3.1

Une manière de lire cette matrice est de considérer que chaque colonne correspond au *fanout*

du registre en indice et que chaque ligne correspond au *fanin* du registre indice.

Cette représentation matricielle est importante puisqu'elle permet d'opérer de manière simple des opérations algorithmiques sur le RN. Elle servira également de base pour les étapes suivantes du flot puisque nous pouvons en déduire une modélisation du contrôleur asynchrone, le réseau asynchrone.

En considérant le RN comme un graphe au sens formel, considérons les définitions générales associées :

— Un chemin, ou *path*, est une succession de $R_k \in \mathbb{R}egs$ consécutifs liés par des arcs dans le RN. Par exemple dans la Figure 3.1 on peut trouver le *path* : $R_0 \rightarrow R_1 \rightarrow R_3$. On apelle $\mathbb{P}aths$ l'ensemble des chemins d'un RN et profondeur le nombre de nœuds uniques dans le chemin, on défini $depht : \mathbb{P}aths \rightarrow \mathbb{N}$ l'application qui associe sa profondeur à un chemin.

— Un circuit est un chemin qui commence et se termine par le même registre. Par exemple dans le RN précédemment présenté, nous pouvons trouver le circuit : $R_1 \rightarrow R_2 \rightarrow R_1$. On appelle l'ensemble des circuits d'un RN $\mathbb{C}ircuits$.

— Une boucle est un circuit de profondeur 1, c'est à dire dans le cas d'un RN, un registre connecté à lui même.

3.2 Le réseau asynchrone : Asynchronous Network(AN)

Le réseau asynchrone(AN) est un réseau de Petri ordinaire, plus précisément un graphe marqué ou *marked graph* (MG), que nous allons dériver du RN. Il correspond à une représentation «channel accurate» du contrôleur asynchrone associé au RN. L'utilisation du formalisme des *Petri nets* permet d'effectuer des vérifications mathématiques qui assurent d'une part que le réseau ne présente pas de situation de *deadlock*, qu'il est vivace (propriété de *liveness*), et d'autre part qu'il existe une équivalence de flot de données entre le circuit synchrone et le circuit désynchronisé. La définition de cette représentation ainsi que ses propriétés ont été présentées par Simatic *et. al* dans [53], et sont retranscrites dans ce manuscrit.

3.2.1 Obtention du réseau asynchrone depuis le réseau de registres

Dans les circuits que nous étudions, nous utilisons un canal *bundled-data* implémentant un protocole 4 phases. La propagation de la requête dans les canaux suit la propagation de la donnée associée dans le *datapath*. Les requêtes seront donc représentées de manière exhaustive dans les AN. Dans ces derniers, les places représentent les canaux de communication et les transitions représentent les points de synchronisations entre les requêtes *i.e.* :

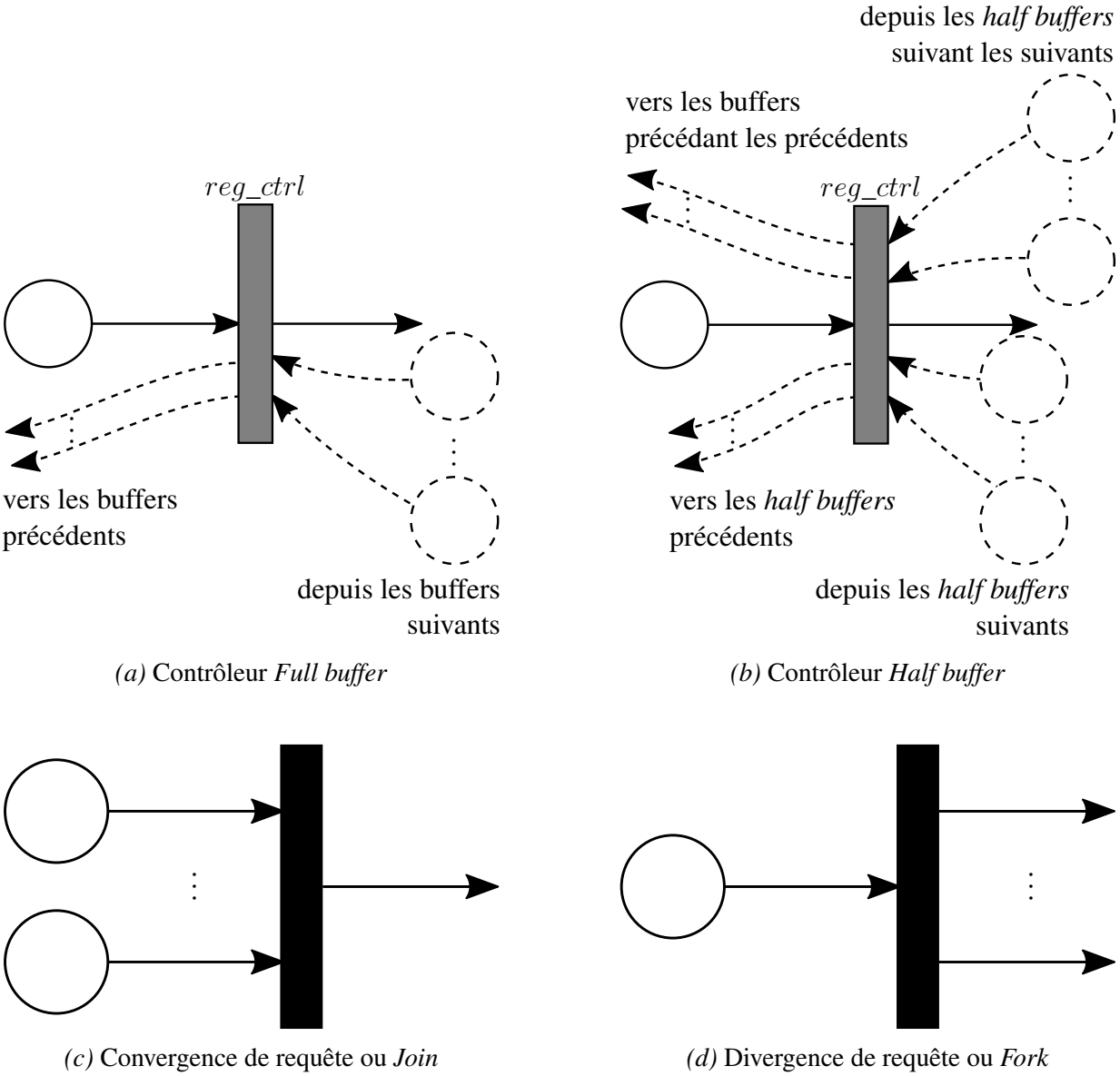
— les contrôleurs de registre

— les divergences de requêtes : sélectives (*split*) ou non sélectives (*fork*)

— les convergences de requêtes : sélectives (*merge*) ou non sélectives (*join*)

Afin d'assurer une représentation correcte, il faut également représenter les acquittements dans le réseau. Un registre ne peut en effet contenir qu'une seule donnée à la fois. Cela suppose que la donnée doit avoir « libéré » un contrôleur de registre avant qu'il puisse en accepter une autre. Il est à noter que cette limitation est liée au choix précédement effectué du type de contrôleur. Par conséquent, l'utilisation d'autres contrôleurs, par exemple des contrôleurs possédant plus de concurence, changerait la représentation des arcs d'acquittement.

Dans la Figure 3.3, on retrouve différents fragments de registre asynchrone.



(a) Contrôleur *Full buffer*

(b) Contrôleur *Half buffer*

(c) Convergence de requête ou *Join*

(d) Divergence de requête ou *Fork*

Figure 3.3: Fragments de réseau asynchrone

Les arcs pleins représentent les chemins de requête. Ceux en pointillés représentent quant à eux les chemins d'acquittement. Afin de simplifier la représentation, les arcs d'acquittement sont représentés de manière minimale. De cette manière, chaque contrôleur de registre possède

un arc d'acquittement qui va le lier à chacune des transitions du contrôleur précédent, ou celles précédant les précédents dans le cas des *half buffer*.

Comme l'indique leur nom, les contrôleurs complets ou *full buffer* peuvent contenir une donnée chacun, on dit alors qu'ils ont une capacité, ou un *slack* de 1. En ce qui concerne les demi-contrôleurs, ou *half buffer*, leur *slack* est de $1/2$. Pour cette raison, les arcs d'acquittement dans le AN sont reportés avec une profondeur supplémentaire par rapport aux *full buffer*. Nous reviendrons plus précisément sur la construction de ces différents contrôleurs et leur comportement dans la Section 3.3.

La création du réseau asynchrone depuis le registre est assez simple. Elle consiste à simplement remplacer chaque registre du RN par un contrôleur complet et à placer les transitions de manière à respecter les *fanin* et *fanout* de chaque registre.

On distingue deux types de transitions : les transitions observables et les transitions internes. Les première sont l'équivalent de registres dans le RN, et sont représentées par des transitions de contrôleur dans l'AN. Les transitions internes sont elles des transitions de convergence, ou divergence.

Dans le cas du RN présenté précedemment, le AN associé serait celui présenté dans la Figure 3.4.

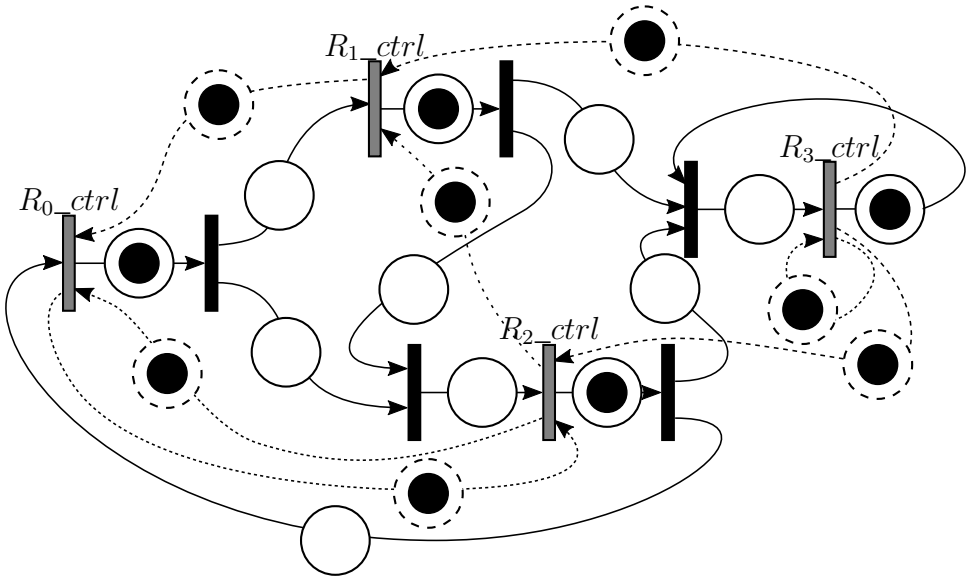


Figure 3.4: Transformation directe du réseau de registre de la Figure 3.1 en réseau asynchrone (non *live*)

Comme nous le verrons bientôt dans ce chapitre, chaque transition observable est associée à un token en situation initiale.

En l'état, le réseau ne remplit pas la propriété de *liveness*. En effet, deux fragments sont problématiques, les boucles $R_0 \rightarrow R_2 \rightarrow R_0$ et $R_3 \rightarrow R_3$. En respectant la règle de tirage stricte, *c.f. Section 2.3.2.3*, alors l'arc d'acquittement empêche tout tirage. Pour régler ces problèmes d'interbloquage, il faut ajouter des transitions de contrôleur, considérées observables, ce

qui permet de déporter les arcs d'acquittement sur ces dernières.

Il est à noter qu'ajouter une transition intern n'aurait pas d'intérêt puisque ces transitions n'ont pas de capacité (*slack*).

Ainsi, on ajoute des places vides dans le réseau qui laissent la possibilité au jeu de jetons, ou *token game*, de se dérouler en respectant la règle de tirage stricte.

Soit $\mathbb{C}ircuits$ l'ensemble des boucles dans un RN, et par conséquent l'ensemble des boucles du AN puisque ce dernier est tiré du réseau de registres.

De manière formelle, si on considère $nb_{token}(c)$ le nombre de tokens dans un circuit c du AN, alors il faut assurer que le *slack*, $S_{circuit}$ de la boucle, *i.e.* la somme des slacks des transitions de contrôleur, soit strictement plus grande que le nombre de tokens. La Règle 1 sera à respecter afin d'obtenir un contrôleur vivace.

Règle 1 (Capacité minimale des boucles, ou Slack Constraint)**.**

$$\forall c \in \mathbb{C}ircuits, \left(\sum^{reg \in c} slack(c) \right) = S(c) > nb_{token}(c)$$

En applicant cette règle aux circuits présents dans le AN, on peut alors déterminer les endroits où ajouter des transitions de contrôleur afin d'ajouter de la capacité dans chaque boucle. Dans notre exemple, une solution est celle représentée dans la Figure 3.5.

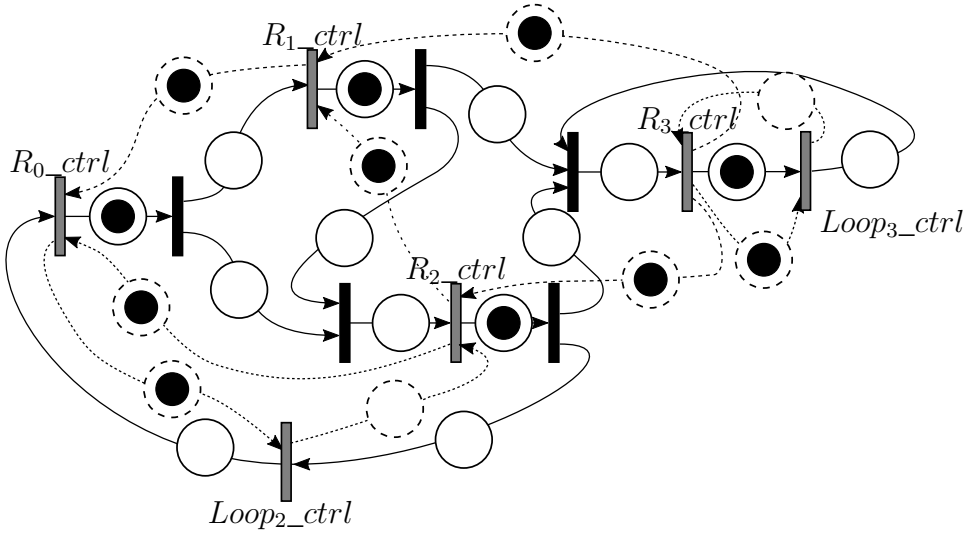
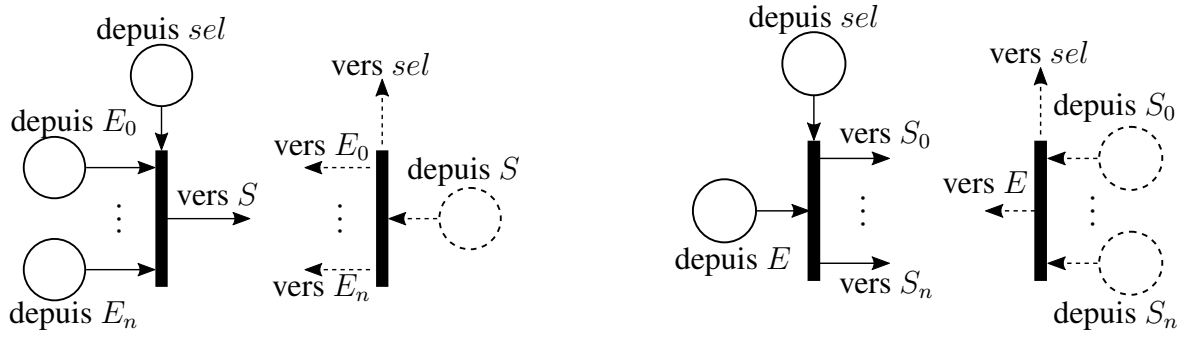


Figure 3.5: Insertion de transitions observables dans les boucles pour rendre le AN de la Figure 3.4 vivace

La détermination des endroits les mieux adaptés où insérer ces contrôleurs sera un des points du Chapitre 4. Il est à noter que l'ajout de transitions observables n'est pas sans conséquences sur le chemin de donnée. Nous verrons cependant dans la Section 3.4.2 comment rendre cette modification transparente d'un point de vue global.

Dans les réseaux de Petri, il est possible de représenter des structures de choix (multiplexeur ou démultiplexeur de *token* par exemple). Elle sont modélisées par des pondérations variables sur les arcs d'entée ou de sortie de la transition. Bien qu'elles ne soient pas utilisées dans cet exemple, elles pourront être utilisées par la suite. Les représentations sous forme de réseau asynchrone de ces transitions sélectives sont présentées dans la Figure 3.6.



(a) Multiplexeur de requête ou *Merge* *(b)* Démultiplexeur de requête ou *Split*

Figure 3.6: Fragment de réseau asynchrone pour les structures de choix

L'utilisation de ces éléments de choix de manière automatique peut cependant s'avérer compliquer à mettre en place dans une démarche de désynchronisation. Une utilisation notable de ces élements peut cependant être trouvée dans [52].

Nous pouvons remarquer que la représentation directe s'avère relativement lourde. De nombreux travaux ont néanmoins porté sur la simplification des réseaux de Petri. Après quelques définitions, nous nous intéresserons à ces simplifications qui permettent de conserver les propriétés intéressantes de notre modélisation.

3.2.2 Définitions

***N.B. :** Toutes les définitions et règles données par la suite à propos des AN ont été présentées, et prouvées, dans [53].*

Comme définition du réseau asynchrone en tant que réseau de Petri, nous allons adapter celle précédemment décrite, dans la Section 2.3.2.3, $AN = (P, T, F, w)$ à notre utilisation :

— P est l'ensemble des places.
— T est l'ensemble des transitions.
— $F \subset (P \times T) \cap (T \times P)$ est l'ensemble des arcs place $\rightarrow$ transition et transition $\rightarrow$ place.
— $w : F \rightarrow \{0, 1\}$ est la fonction de poids de chacun des arcs : $w(x, y)$ est le poids de l'arc (x, y).

En considérant le *Petri net* défini, AN, nous allons aussi utiliser les notations suivantes :

— $M : P \rightarrow \{0, 1\}$ est le nombre de tokens dans chaque place. On note généralement M_0 le marquage initial.

— On appelle *preset* de $y \in T \cup P$, ${}^\bullet y = \{x \in T \cup P / (x, y) \in F\}$ l'ensemble des éléments antécédents de y au travers les arcs de F.
— On appelle *postset* de $x \in T \cup P$, $x^\bullet = \{y \in T \cup P / (x, y) \in F\}$ l'ensemble des éléments suivants de x au travers les arcs de F.

Le modèle d'éxécution de AN prend en compte la règle d'activation stricte telle que décrite dans [42]. Dans le cas qui nous concerne, les graphes marqués (MG), cela correspond au respect des deux règles suivantes :

$$\forall p \in {}^\bullet t, M(p) \geq w(p, t)$$
$$\forall q \in t^\bullet, M(q) \leq 1 - w(t, q).$$

Si ces règles de transition sont respectées, on obtient le marquage suivant, M', tel que :

$$\forall p \in {}^\bullet t, M'(p) = M(p) - w(p, t)$$
$$\forall q \in t^\bullet, M'(q) = M(q) + w(t, q)$$
$$\forall o \in P / ({}^\bullet t \cup t^\bullet), M'(o) = M(o)$$

De plus, nous définissons les notations suivantes :

— $M \xrightarrow{t} M'$ signifie que la transition t peut être tirée et que M' est le marquage résultant de l'activation de t à partir de M
— une séquence d'activation $\sigma = t_1.t_2.\ldots.t_n$ représente les transitions successives tels qu'il existe des marquages M_1, M_2, …, M_{n+1} vérifiant $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_{n+1}$

Dans nos travaux, le réseau de Petri AN représente le circuit asynchrone qui va contrôler les différentes *flip-flops* du chemin de données. D'un point de vue « extérieur », seules les données contenues dans ces *flip-flops* sont observables. C'est pourquoi les transitions observables sont celles associées à un contrôleur de registre. Les autres transitions, définies comme non observables (internes), sont seulement nécessaires pour la synchronisation entre les requêtes, ou données, qu'elles représentent.

Nous pouvons donc assurer l'équivalence de trace par l'équivalence de trace des transitions observables. Afin de simplifier les notations pour la suite, nous noterons toutes les transitions internes par 1. Par conséquent, les transitions observables feront donc partie de l'ensemble $T \backslash \{1\}$. Nous définissons également l'opérateur $\widetilde{\ }$ qui associe à une séquence de transitions quelconque, σ, la séquence de transitions observable, $\widetilde{\sigma}$. Cet opérateur est défini de manière récursive et présenté dans l'Équation (3.1).

$$\widetilde{\sigma.t} = \begin{cases} \widetilde{\sigma} & \text{if } t = 1 \\ \widetilde{\sigma}.t & \text{if } t \in T \setminus \{1\} \end{cases} \tag{3.1}$$

L'*Asynchronous Network* est le modèle d'un circuit, nous définissons l'exactitude de ce modèle comme l'équivalence de traces [6] (Définition 1).

Définition 1 (Exactitude du modèle)**.** *Le modèle d'un circuit est correct si celui-ci et le circuit qu'il modélise ont le même ensemble de séquences observables.*

On suppose que le réseau asynchrone obtenu en connectant les fragments, correspondant au chemin de données d'un RN, est correct.

3.2.3 Initialisation du AN

Dans un réseau de Petri, tel que l'AN, des *tokens* représentent des données se propageant dans le réseau. Partant d'une spécification synchrone, chaque registre est considéré comme contenant une donnée. C'est pourquoi dans le passage d'un réseau de registre à un réseau asynchrone, chaque transition observable sera associée à une donnée, donc un *token*, à l'état initial.

Ce postulat pour l'initialisation des systèmes désynchronisés est aussi celle utilisée dans [15]. Il a aussi l'avantage d'affranchir tout questionnement du concepteur vis à vis de l'initialisation ou non d'une place, en particulier dans le cas de boucles.

Il reste cependant à la charge du concepteur de simuler le réseau de Petri obtenu afin de vérifier que la consommation, ou la génération, de *tokens* n'engendre pas de situation de *deadlock*. Une attention particulière est à porter dans le cas des transitions sélectives étant donné que le flot de requêtes est lié au données.

3.2.4 Transformations du AN

De nombreux travaux ont été menés sur la simplification des réseaux de Petri. Dans le cas de notre application, celles conservant les propriétés de *safeness*, *liveness* ([42]) et de *reachability* ([23]) sont les plus pertinentes. Elles ont été adaptées à notre cas d'utilisation et prouvées dans [52].

3.2.4.1 Simplifications de transitions

Deux transformations sont possibles afin de simplifer certaines transitions non-observables. Prenons deux transitions $(t_n, t_p) \in T^2$ et $p \in P$ telles que :

— $t_n \in T \cap \{1\}$, ou $t_p \in T \cap \{1\}$, *i.e.* soit t_p soit t_n est une transition interne

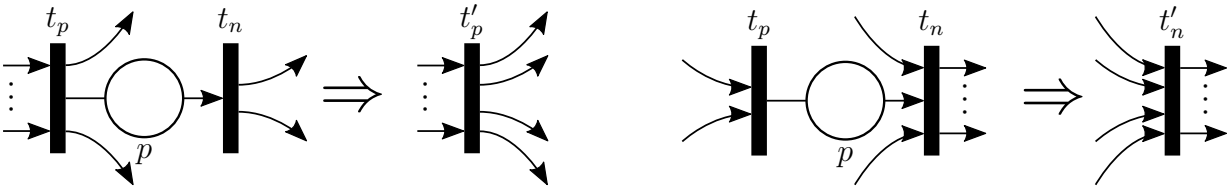
— $p \in {}^\bullet t_p$ et $p \in t_n^\bullet$, *i.e.* p est entre les deux transitions

— $M_0(p) = 0$, *i.e.* il n'y a pas de token en situation initiale dans p

Si ${}^\bullet t_p = \{p\}$, et t_n est une transition interne, on peut remplacer $t_p \rightarrow p \rightarrow t_n$ par t'_p tel que $t'^\bullet_p = t_p^\bullet \cup t_n^\bullet$ et ${}^\bullet t'_n = {}^\bullet t_n$ *i.e.* les prédécésseurs de t_n sont la combinaison de ceux de t_n et de t_p, Figure 3.7a.

De manière symmétrique, si $t_p^\bullet = \{p\}$, et t_p est une transition interne, on peut remplacer $t_p \rightarrow p \rightarrow t_n$ par t'_n tel que ${}^\bullet t'_n = {}^\bullet t_p \cup {}^\bullet t_n$ et $t'^\bullet_n = t_n^\bullet$ *i.e.* les prédécesseurs de t'_n sont la combinaison de ceux de t_n et de t_p, Figure 3.7b.

Démonstration. Mettre en commun deux transitions t_n et t_p suppose qu'un observateur extérieur ne voit pas de différence suivant l'ordre d'activation de t_n et t_p. Si une transition est obser-



(a) Simplification de *fork* *(b)* Simplification de *join*

Figure 3.7: Simplifications de transitions

vable, alors son activation impacte directement la trace. De fait, on ne peut mettre en commun deux transitions observables puisque l'ordre ne serait pas conservé.

Considérons maintenant que p est initialement vide ($M_0(p) = 0$) et que soit t_n soit t_p est interne. En supposant que $\{p\} = {}^\bullet t_n$ et $t_n = 1$, comme dans la Figure 3.7a, alors si t_p est activée alors t_n peut directement être activée.

Formellement, en pour n'importe quelle séquence d'activation $\sigma = \sigma_0.t_p.\sigma_1.t_n.\sigma_2$ avec $\sigma_0, \sigma_1 \in (T \backslash \{t_n\})^*$, $\sigma' = \sigma_0.t_p.t_n.\sigma_1.\sigma_2$ est aussi une séquence correcte, et $\widetilde{\sigma} = \widetilde{\sigma}' = \widetilde{\sigma_0}.t_p.\widetilde{\sigma_1}.\widetilde{\sigma_2}$. De manière récursive, en appliquant ce raisonnement à la séquence d'activaton σ_2, on prouve que $\{p\} = {}^\bullet t_n$ est suffisant pour pouvoir simplifier le cas où t_n et t_p sont internes et le cas où seulement t_n l'est.

Supposons maintenant que $t_p^\bullet = \{p\}$ et $t_p = 1$(Figure 3.7b). Cette fois, pour toute séquence d'activation $\sigma = \sigma_0.t_p.\sigma_1.t_n.\sigma_2$ avec $\sigma_0, \sigma_1 \in (T \backslash \{t_n\})^*$, alors $\sigma' = \sigma_0.\sigma_1.t_p.t_n.\sigma_2$ est aussi une séquence correcte, et $\widetilde{\sigma} = \widetilde{\sigma}' = \widetilde{\sigma_0}.\widetilde{\sigma_1}.t_n.\widetilde{\sigma_2}$. De la même manière que dans le cas précédent, cela prouve que $t_p^\bullet = \{p\}$ est suffisant pour pouvoir effectuer la simplification dans le cas où t_n et t_p sont internes et dans le cas où seulement t_p l'est. □

Extension aux structures de choix

Les transitions *merge* et *split* peuvent être modélisées de la même manière que les transitions *join* et *fork* avec des arcs ayant des poids $w_i \in \{0, 1\}, i \in [0, n[$ avec n le nombre de canaux d'entrée, respectivement de sortie, de l'élement. La règle de simplification ne change pas, les poids des arcs regroupés sont simplement reportés dans les poids des nouveaux arcs. La Figure 3.8 illustre ces types de transformations.

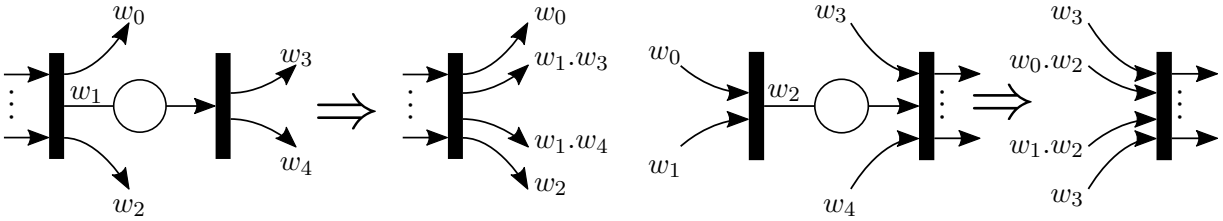


Figure 3.8: Simplifications de transitions sélectives

3.2.4.2 Simplification de places parallèles

Il peut se produire la situation où deux places partagent les mêmes transitions antécédantes et suivantes. Dans ce cas, il est en général possible de simplifier l'une des deux places si elles ont le même marquage initial.

Considérons $(p, p') \in P^2$ avec $p \neq p'$ et $(t_n, t_p) \in T^2$. Si :

— $t_n^\bullet \supset p \subset {}^\bullet t_p$

— $t_n^\bullet \supset p' \subset {}^\bullet t_p$

— $M_0(p) = M_0(p')$

Alors on peut alors enlever p' du réseau et le marquage initial est conservé, Figure 3.9.

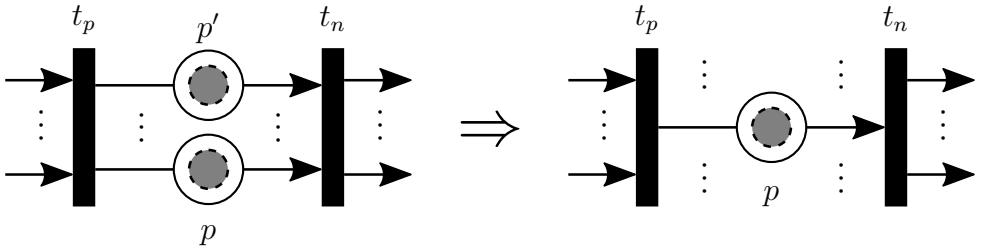


Figure 3.9: Simplification places parallèles

Démonstration. Prenons G un graphe marqué et $\overline{G}$ le graphe marqué obtenu en retirant p' de G. En raisonnant par récurence sur la longueur des séquences d'activation dans G et $\overline{G}$, pour tout marquage M atteignable dans G, si la valeur de $M(p) \oplus M(p')$ est préservée, alors le marquage $\overline{M}$, de $\overline{G}$, obtenu en enlevant p à M reste également atteignable et t_n, ou t_p, est activable dans $\overline{G}$ si et seulement si la transition est activable dans G. □

Il est à noter que dans le cas d'utilisation de structures de choix il faut que les arcs soit de poids égaux afin de procéder à cette simplification.

3.2.4.3 Modification du marquage initial

Considérons les marquages M_0, M_0' et une transition t. Si il existe $t \in \{1\}$ tel que $M_0 \xrightarrow{t} M_0'$ ou $M_0' \xrightarrow{t} M_0$, alors M_0' peut être considéré comme un marquage initial valide, Figure 3.10.

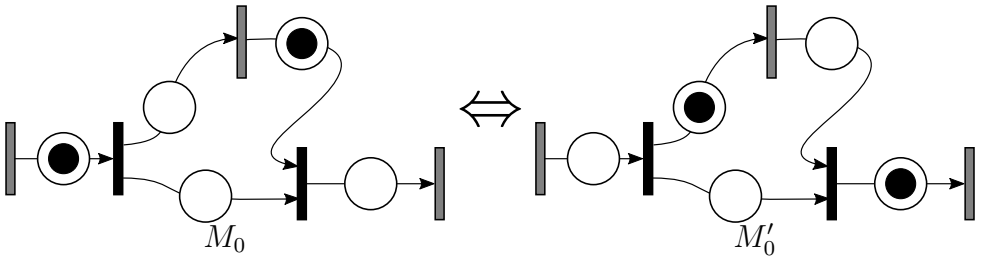


Figure 3.10: Modification du marquage initial d'un AN

Démonstration. Les deux cas étant symmétriques, considérons $M_0 \xrightarrow{t} \overline{M_0}$. Les graphes marqués sont dit persistants, c'est-à-dire qu'une transition activable sera soit activée soit restera indéfiniment activable.

Par conséquent, pour une séquence d'activation quelconque $\sigma_0.t.\sigma_1$ de M_0 avec $\sigma_0 \in (T \backslash \{t\})^*$, $t.\sigma_0.\sigma_1$ est un séquence d'activation correcte pour $\overline{M_0}$.

De la même manière, pour n'importe quelle séquence d'activation de $\overline{M_0}$, $t.\sigma$ est une séquence d'activation correcte de M_0. Étant donné que t est interne, les séquences d'activation observables sont identiques. □

Cela signifie que les tokens du marquage initial du AN peuvent se propager dans le réseau tant qu'ils ne traversent pas de transition observable. Cette propriété permet d'arranger les tokens dans le réseau afin de procéder aux autres transformations de simplification.

3.2.4.4 Simplification d'acquittements

Certains signaux d'acquittement peuvent êtres simplifiés. Cela permet d'alléger la représentation du AN dans le cas de boucle requête/acquittement.

Considérons p et p' des places entre deux transitions observables t_n et t_p. Avec les conditions suivantes :

— $t_n^\bullet \supset p \subset {}^\bullet t_p$, un chemin de requête
— $t_p^\bullet \supset p' \subset {}^\bullet t_n$, un chemin d'acquittement
— $M_0(p) = |1 - M_0(p')|$

Alors, on peut supprimer l'arc d'acquittement et p' en conservant le marquage de p. La Figure 3.11 illustre les deux simplifications possibles suivant cette règle.

Figure 3.11: Simplification des chemins d'acquittement entre deux contrôleurs consécutifs

Cette transformation demandant de respecter la règle d'activation stricte, elle doit être effectuée après toutes les autres transformations. Elle peut être prouvée de la même manière que dans la Section 3.2.4.2.

3.2.4.5 Application des simplifications au AN exemple

Pour l'exercice, appliquons les différentes règles vues au réseau asynchrone de la Figure 3.5. La première simplification que nous pouvons effectuer concerne toutes les transitions internes. Il est en effet toujours possible de déplacer un jeton, seulement au travers de transitions internes, de manière à tomber dans la situation permettant la simplification de ces transitions.

Nous n'avons pas de situation de places parallèles dans cet exemple.

Concernant les simplifications d'acquittement, elles sont possibles dans ce cas autour des transitions des contrôleurs buffers ajoutés pour rendre le réseau *live*.

Le réseau simplifié résultant est présenté dans la Figure 3.12. La procédure de simplification à en effet réduit le nombre de chemins et permet maintenant une plus simple simulation de ce réseau afin de trouver des situations qui pourraient s'avérer problématiques.

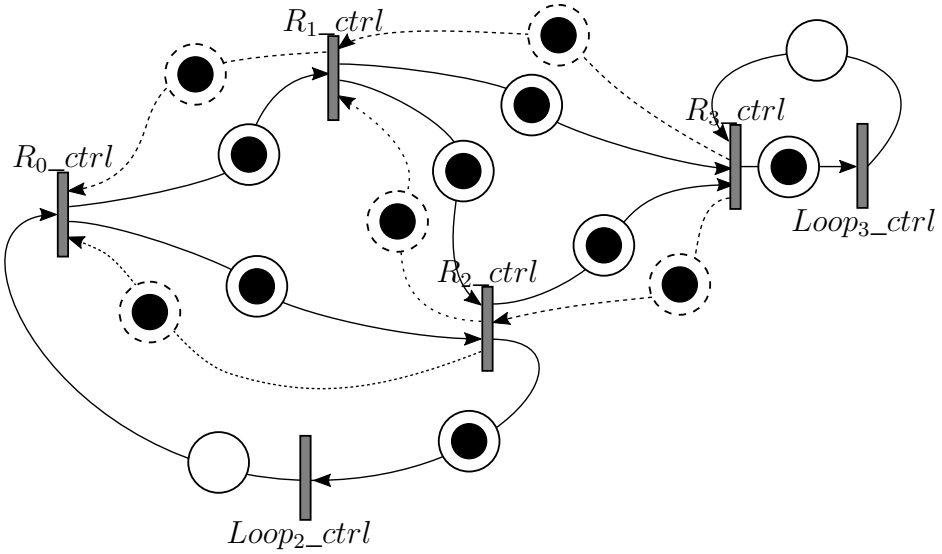


Figure 3.12: Réseau asynchrone du RN de la Figure 3.1 après application des règles de simplification

3.3 Bibliothèques de composants asynchrone

Afin de transcrire un AN en réseau de contrôleur, ou *controller network*(CN), nous avons besoin de définir de manière précise les structures de portes logiques à utiliser. Une fois cette bibliothèque définie, nous pourrons alors déterminer quelles vérifications supplémentaires sont nécessaires à l'obtention d'un contrôleur asynchrone fonctionnel et correct.

> ***N.B. :*** *Les différents éléments décrits dans cette section sont directement lié au choix du couple canal/protocole effectué dans la Section 2.3.2.2. L'utilisation d'un autre canal et/ou protocole demanderait de ré-implémenter les différents éléments décrits dans cette section.*

La principale précaution à prendre dans le cas de la conception de circuits asynchrone est de s'assurer que le circuit ne comporte pas de source d'aléas, statiques ou dynamiques, aussi appelés *glitches*. Dans des circuits synchrones, des *glitches* sont généralement problématiques en termes d'activité des circuits, et donc de consommation d'énergie, mais sont de manière plus rare la source de problèmes fonctionnels vu leur nature transitoire.

En ce qui concerne les circuits asynchrones cependant, tous les évènements prenant place dans le circuit ont une signification. Si l'on reprend la représentation du AN précedemment

vue, un *glitch* sur une requête représente une « fausse » donnée, et correspond à l'insertion d'un *token* dans le réseau asynchrone. Outre le fait que cela peut entrainer des situations de *deadlock*, cela signifie aussi qu'un défaut transitoire tel qu'un *glitch* se transforme finalement en défaut permanent et modifie le flot de donnée dans les circuits asynchrones.

La seule façon de se débarasser de ce *token* parasite serait donc de réinitialiser totalement le circuit avec une procédure de *reset* par exemple.

De manière similaire, un glitch sur un signal d'acquittement considérerait une requête comme acquittée avant que la requête ne soit effectivement traitée. Dans le réseau asynchrone, cela correspondrait à la disparition d'un *token* ce qui peut avoir les mêmes conséquences qu'un ajout.

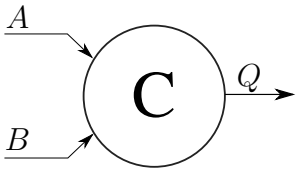
Dans notre cas, cette contrainte se limite à la partie contrôleur qui viendra remplacer l'arbre d'horloge. En effet, le chemin de donnée est totalement dé-correlé de la partie de contrôle donc il n'y a pas de modification à opérer du côté du datatpath.

Pour limiter le risque d'aléa, ou *hazards*, les cellules de Muller introduites dans la Section 2.3.2.3 sont communément utilisées en conception asynchrone. Leur fonction mémorisante permet en effet d'éviter de propager un aléa en entrée sur la sortie. Dans nos circuits de contrôle, nous limiterons au maximum l'utilisation de cellules logiques classique et surtout, nous ferons en sorte que chaque requête et acquittement passent par un *C-element* afin de filtrer les aléas.

3.3.1 La cellule de Muller, ou C-element

Cette cellule a été pour la première fois décrite par David E. Muller dans [41]. Elle est comprend 2 entrées, nommée dans la suite A et B avec une sortie appelée Q.

Dans la Figure 3.13 nous pouvons trouver sa représentation symbolique graphique ainsi que la table de vérité associée, avec Q^{-1} signifiant que la porte conserve l'état qu'elle avait précédemment.



(a)

A	B	Q
0	0	0
1	0	Q^{-1}
0	1	Q^{-1}
1	1	1

(b)

```
1 module c_element (
2  input wire A,B,
3  output reg Q  );
4  always @*
5   if      (  A &&  B)
6    Q <= 1'b1;
7   else if ( ~A && ~B)
8    Q <= 1'b0;
9 endmodule
```

(c)

Figure 3.13: (a) Représentation schématique, (b) table de vérité et (c) description RTL d'une cellule de Muller, où C-element

L'opération implémentée par cette cellule est le « rendez-vous » : Q ne change pour prendre la valeur de ses entrées que lorsque A et B sont identiques. Cette fonctionnalité peut être implémentée simplement en utilisant des portes logiques classiques au niveau transistor [18]. Deux de ces implémentations naïves sont présentées Figure 3.14.

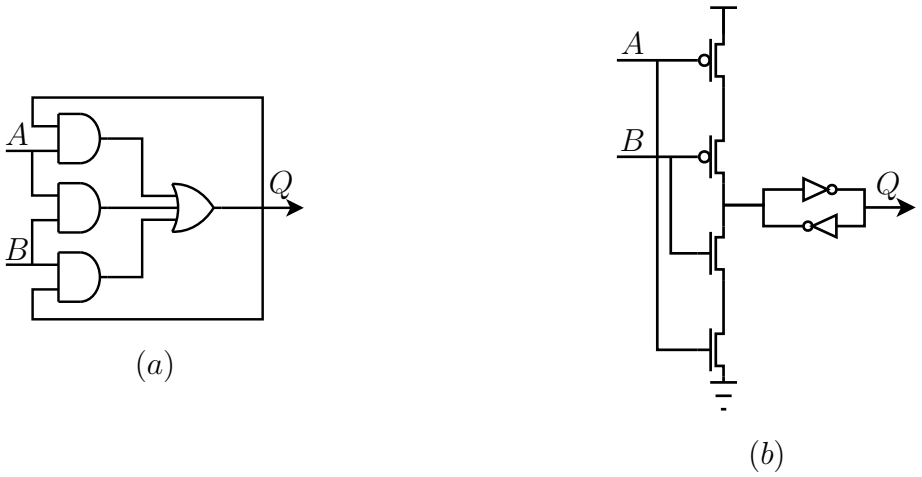


Figure 3.14: Implémentation niveau porte logique (a) et niveau transistor (b) d'un C-element

Il existe de nombreuses implémentations de cette porte de Muller suivant les cas d'application : [3, 48, 49]. Dans un premier temps, cette implémentation est généralement suffisante d'autant plus qu'elle à l'avantage de pouvoir être implémentée sur une carte de prototypage de type *Field-Programable Gate Array* où FPGA [45]. Notre partenaire dans le projet, Dolphin Integration, nous a fourni des cellules de Muller au format de cellules standards.

Le *C-element* est la brique de base qui compose la majeure partie des blocs asynchrones. Pour pouvoir effectuer des comparaisons de surface par la suite, il est intéressant d'avoir une mesure de la taille de la cellule de Muller et de la mettre en rapport avec les différentes cellules standards. Pour avoir un point de comparaison le plus juste possible, nous avons relevé les différentes surfaces de cellules standards dans la librairie utilisée. Les différentes valeurs sont référencées dans le Tableau 3.1 en $u.a.$, ou unité d'aire. La référence étant la surface d'un l'inverseur qui représente 3 $u.a.$.

Cellule combinatoire	Inverseur	Buffer	NAND2	AND2	NOR	OR2	XOR2	NXOR2
Surface($u.a.$)	3	4	4	5	4	6	12	14

Cellule séquentielle	D latch	D Flip Flop	C-element
Surface($u.a.$)	16	31	14

Tableau 3.1: Surface en unité de surface arbitraire $u.a.$, de différentes cellules standard

On remarque que la cellule de Muller est de taille conséquente comparativement aux cellules standards. Un *latch* reste cependant de taille similaire ce à quoi nous pouvons nous attendre étant donné que la fonctionnalité est similaire.

Nous utiliserons ces valeurs par la suite pour évaluer la taille des différents composants asynchrones. Nous ne prendrons pas en compte dans nos approximations la surface occupée par les interconnexions entre ces cellules.

Nous allons maintenant voir comment utiliser les cellules de Muller avec d'autres cellules standards pour créer une bibliothèque de composants utiles aux circuits micropipelines.

3.3.2 Canal de communication asynchrone, *Channel*

La conception de circuit asynchrone possède un atout important vis-à-vis de sa modularité. Les canaux de communication peuvent en effet être considéré comme une seule entité implémentant les signaux nécessaires à la communication.

Pour obtenir cette modularité, il est important de correctement spécifier le canal en question. Avec le langage *SystemVerilog*, il est possible d'utiliser la construction ***interface*** qui s'avère pertinent pour ce genre d'application.

Dans la Figure 3.15 est décrit le canal de communication utilisé dans la suite de ce travail.

```
1 interface Channel ();
2   wire req, ack;
3   modport in  (input req, output ack);
4   modport out (input ack, output req);
5 endinterface
```

Figure 3.15: Implémentation RTL d'un canal de communication 4-phases à données groupées

L'avantage de cette construction est que l'on peut définir plusieurs sens de connexion pour un même jeu de fils. Cela nous permettra donc d'utiliser cette seule construction pour connecter tous nos éléments asynchrones.

Les interfaces sont d'autant plus intéressantes qu'elles permettent l'utilisation d'assertions formelles pour vérifier pendant la simulation que le protocole implémenté est respecté.

Il est à noter que dans les circuits considérés, nous utilisons un canal *bundled-data*, *i.e.* contenant les signaux de contrôle et les données. Cependant, vis à vis de la contrainte visant à ne pas modifier le chemin de donnée, nous ne pouvons pas inclure les données dans la spécifications Verilog des canaux. Dans la suite, les données seront tout de même considérées comme associées au canal contenant les signaux de contrôles correspondants.

N.B. : *Pour plus de clarté dans les schémas de ce document, une légende globale sera utilisée par la suite. En l'absence de mention contraire dans un schéma particulier, les codes utilisés seront ceux spécifiés dans la Figure 3.16.*

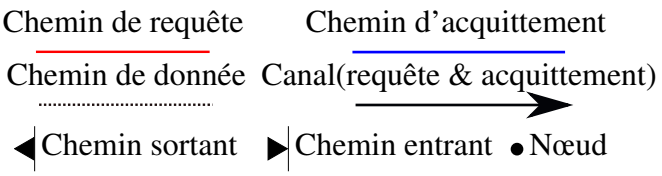
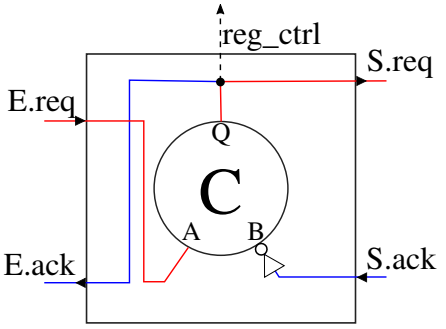


Figure 3.16: Légende générales des signaux de requête, acquittement, donnée, canal de communication et de direction port

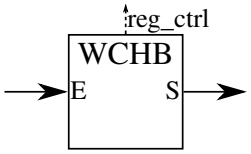
3.3.3 Contrôleur de registre, le *Weak-Conditionned Half-Buffer*

Nous nous sommes déjà penché sur la construction du contrôleur de registre que nous allons utiliser dans nos travaux dans la Section 2.3.2.4.

La Figure 3.17 représente un contrôleur au niveau portes logiques, identique à celle décrite dans la Figure 2.21, sa représentation que nous utiliserons pour le CN et son interface en *SystemVerilog* telle que nous l'avons implémenté.



(a)



(b)

```
1 module WCHB (
2     Channel.in  E,
3     Channel.out S);
4 /*
5 ...Body...
6 */
7 endmodule
```

(c)

Figure 3.17: Implémentation niveau porte logique (a), représentation niveau CN(b) et interface RTL(c) d'un contrôleur WCHB

Ce contrôleur est très compact puisqu'il ne contient qu'une seule cellule de Muller et un inverseur, sa surface équivalente est donc de 17 $u.a.$. Cette compacité permet de limiter la taille du contrôleur global à un niveau raisonnable. En utilisant cette architecture, la complexité du contrôleur global reste également abordable pour un designer non habitué à ce genre d'architectures. À la cellule présentée peut aussi être adjoint un mécanisme d'initialisation (*set* où *reset*), pour pouvoir initialiser les contrôleurs contenant des données, un *token* dans le AN, où vide.

Ce type de contrôleur spécifique est appellé dans la littérature *Weak-Conditionned Half-Buffer*. En effet, comme l'explique Andrew Lines dans [36], les canaux d'entrée et sortie E et S ne peuvent pas contenir deux données distinctes, deux WCHB sont donc nécessaires pour contenir totalement une donnée.

Prenons maintenant l'exemple de deux WCHB accolés comme présenté dans la Figure 3.18 et etudions le comportement des requêtes et acquittements dans ces contrôleurs.

Dans cette figure, à l'étape ① arrive une requête à travers le canal In à l'entrée de WCHB1.

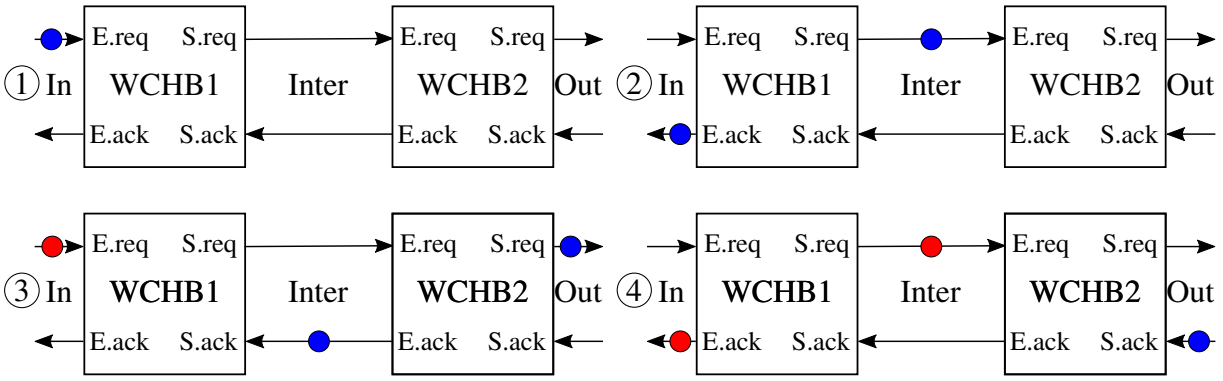


Figure 3.18: Étapes de la propagation d'une donnée au travers de deux WCHB. Un point représente une requête (un niveau haut du signal), la couleur différencie les données.

Elle est transmise pendant l'étape ② jusqu'à l'entrée de WCHB2. Étant donné que le canal de sortie est libre, la donnée bleue peut être transmise par WCHB2 jusqu'au canal Out pendant l'étape ③. Dans le même temps, un autre token, rouge cette fois ci, est arrivé dans le canal d'entrée. Celle-ci ne peut cependant pas traverser WCHB1 puisqu'un token est toujours présente sur le canal Inter dans le chemin d'acquittement. Il faut donc que la donnée bleu soit transmise totalement au travers de WCHB2, étape ④, pour que la donnée rouge puisse traverser WCHB1 et occuper le canal Inter.

Cela illustre bien qu'un contrôleur WCHB ne peut avoir deux données différentes en entrée et en sortie. En revanche, avec deux contrôleurs WCHB, cette situation peut se produire uniquement durant l'étape③. Deux half-buffers sont donc nécessaires et suffisants pour contenir totalement une donnée.

Ce point à un impact sur les règles à respecter lors de la construction du contrôleur asynchrone que nous verrons dans la Section 3.4.2.

Dans les travaux de Sutherland, le contrôleur du micropipeline est constitué d'une succession de *half-buffers* chacun contrôlant un *latch*, chaque *latch* mémorisant une « demi-donnée ». Dans notre cas cependant, étant donné que nous voulons être le plus proche possible du design synchrone, nous utilisons des bascules D standard et donc ce sont finalement deux half-buffers qui sont associés avec un registre.

De plus, dans le cas présenté par Sutherland, on s'intéresse particulièrement aux pipelines linéaires, une succession d'étages de pipeline. Les pipelines synchrones peuvent prensenter des divergences et convergences dans le flot de données ce qui implique l'utilisation de mécanisme de synchronisation entre les canaux.

Plus largement, entre chaque contrôleur de registre, il existe des communications d'un contrôleur de registre vers plusieurs, qualifiées de divergentes, et des communications dites convergentes, quand plusieurs contrôleurs de registres communiquent en direction d'un seul contrôleur. Dans le réseau de Petri associé modélisant notre réseau de contrôleurs (AN), ces opérations de communications linéaires, convergentes et divergentes sont modélisés par des

transitions.

3.3.4 Transitions

De la même manière que pour les transitions des réseaux asynchrones, les transitions dans le réseau du contrôleur permettent de refléter les opérations effectuées dans le chemin de donnée associé à un ou plusieurs canaux. On peut distinguer plusieurs types de transitions :

— Les transitions non-sélectives : divergences et convergences.
— Les transitions sélectives : elles agissent comme des multiplexeurs ou des démultiplexeurs de requêtes.
— Les transition d'extrémité : ce sont des producteurs ou consommateurs de requêtes.

Les transitions décrites dans la suite sont tirées de plusieurs travaux sur les pipelines élastiques utilisant le protocole bundled-data 4-phases : [18, 46, 53]. Les transitions utilisées ne sont cependant pas celles utilisées classiquement avec le protocole WCHB. Les transition implémentant le protocole WCHB servent en effet à la fois de contrôleur et de transition, c'est à dire qu'elles peuvent contenir des données.

Afin de conserver une séparation entre les deux grands types d'éléments asynchrones, transitions et contrôleurs, et simplifier les modélisation et méthodes de désynchronisation (Chapitre 4), nous utilisons les transitions dites séquentielles qui implémentent également un protocole 4-phases, et donc compatibles avec les contrôleurs choisis.

Il est à noter que l'approche est équivalente puisque les transitions dites WCHB sont au final les transitions séquentielles associées aux contrôleurs WCHB sur les canaux d'entrée ou de sortie suivant la transition considéré.

3.3.4.1 Transition convergente, le *join*

Plusieurs données peuvent être nécessaires pour effectuer une opération. Si nous prenons le cas de l'addition de deux valeurs par exemple, nous pouvons considérer que chacune des données sont contenues dans des registres différents, donc contrôlés par des WCHB différents. Dans la Figure 3.19 est représenté une telle situation.

Pour stocker le résultat de l'opération résultant de ces deux données, il est nécessaire de faire converger les canaux de ces deux données en un seul canal. Ainsi, une donnée ne sera capturée par le Reg3, par le biais de Ctrl3, que lorsque des données seront présentes à la fois dans Reg1 et dans Reg2.

Cette fonction de convergence, aussi appelée *join*, peut être implémenté grâce aux portes de Muller. En effet, dans le cas de deux canaux convergents, il suffit alors d'utiliser la fonction de rendez-vous du *C-element* pour faire se rencontrer les deux requêtes.

L'implémentation de cette fonction pour la convergence de deux canaux et le chronogramme associé sont décrits Figure 3.20.

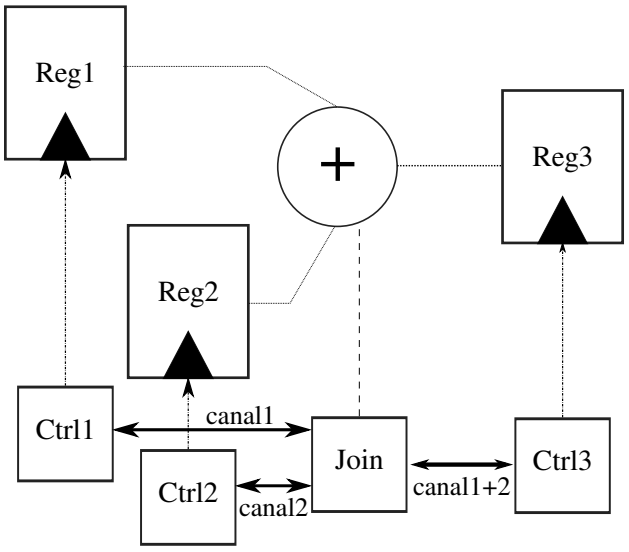


Figure 3.19: Exemple d'une situation de convergence de deux données

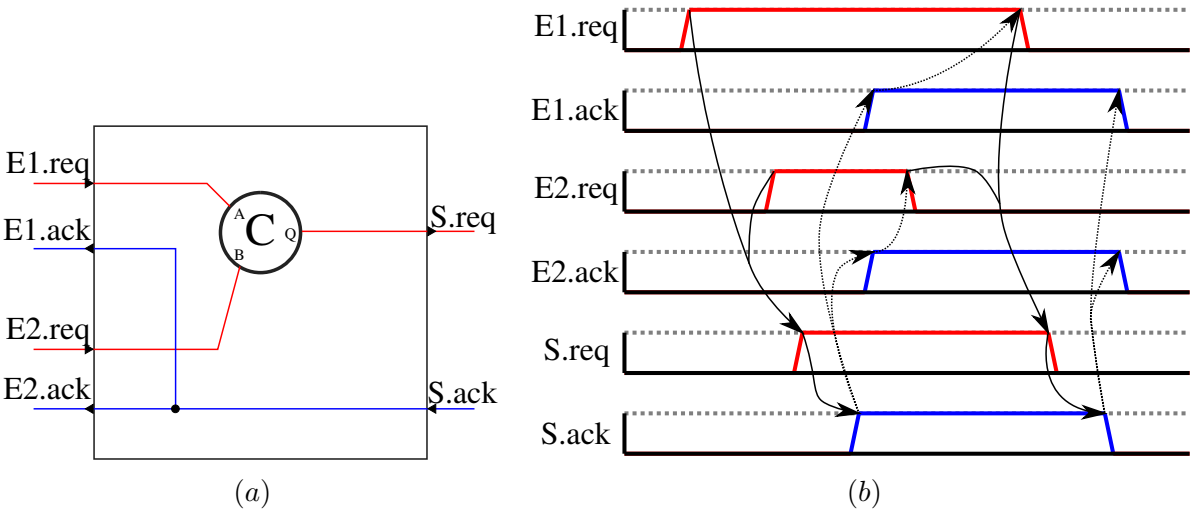


Figure 3.20: Implémentation au niveau portes logiques(a) et chronogramme(b) d'une transition *join* 2 vers 1

Ainsi, lorsqu'une requête arrive au travers du canal E1, avant de transmettre la requête à la sortie, il faut d'abord que le canal E2 reçoive une requête. Une fois les deux requêtes reçues, elle est transmise au travers du canal S.

Ensuite, lorsqu'un acquittement est reçu depuis le canal de sortie, il est directement transmis aux deux canaux d'entrée. Enfin, symétriquement au mécanisme d'arrivée de la requête, pour que la requête de sortie repasse au niveau bas, il faut que les deux requêtes d'entrée repassent à un niveau bas.

La transition *join* est assez compacte en termes de surface puisqu'elle ne contient qu'un seul *C-element* pour 2 entrées, sa surface équivalente est donc de $14\ u.a.$.

3.3.4.2 Transition divergente, le *fork*

De la même manière que pour le join, une donnée peu partant d'un même registre se retrouver dans deux branche distinctes d'un pipeline. On appelle cette divergence une fourche, où *fork*.

La construction de cette transition est très similaire à celle de la transition join à la différence près que cette fois ce sont les acquittements des sorties qui ont besoin de se synchroniser pour être transmis à la source.

La Figure 3.21 présente la façon d'implémenter cette transition ainsi que le chronogramme associé.

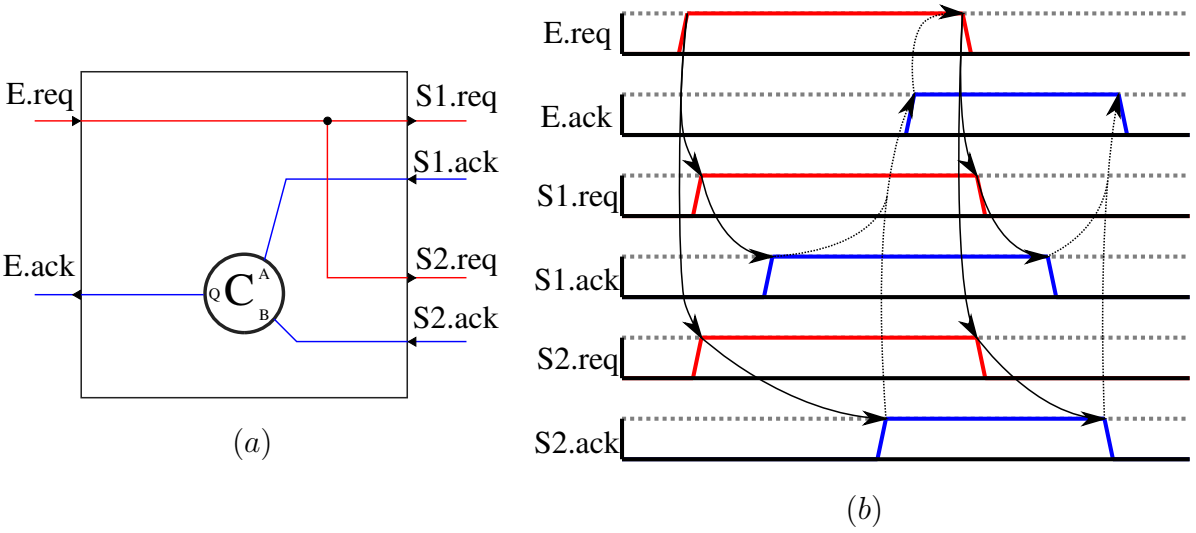


Figure 3.21: Implémentation au niveau portes logiques(a) et chronogramme(b) d'une transition *fork* 1 vers 2

Avec cette implémentation, la requête est directement transmise de l'entrée E vers les deux sorties S1 et S2.

Chacune de ces sorties peut recevoir un acquittement à un instant différent, cependant l'acquittement vers l'entrée ne sera transmis que lorsque les deux sorties auront effectivement été acquittées.

Pareillement au *join*, la transition *fork* est compacte et n'a besoin que d'un seul *C-element* pour 2 sorties. On a donc une surface équivalente de $14\ u.a.$.

Les transitions *fork* et *join* sont des transitions sans sélection. Or, un des avantages des circuits asynchrones repose sur le fait de ne générer de l'activité que lorsque cela est nécessaire. Pour obtenir ce genre de fonctionnalités, nous avons donc besoin de transitions qui permettent de diriger une donnée dans la partie du *datapath* utile au calcul. Des transitions avec sélection sont donc nécessaires.

3.3.4.3 Transition convergente sélective, le *merge*

La transition *merge* agit comme un multiplexeur de requête. Elle comprend deux types de canaux, les canaux d'entrée à sélectionner et un canal de sélection qui joue le rôle d'arbitre dans le choix de quelle donnée va franchir la transition.

La Figure 3.22 présente l'implémentation au niveau portes logiques et la Figure 3.23 le chronogramme associé au passage d'une donnée via le canal E1.

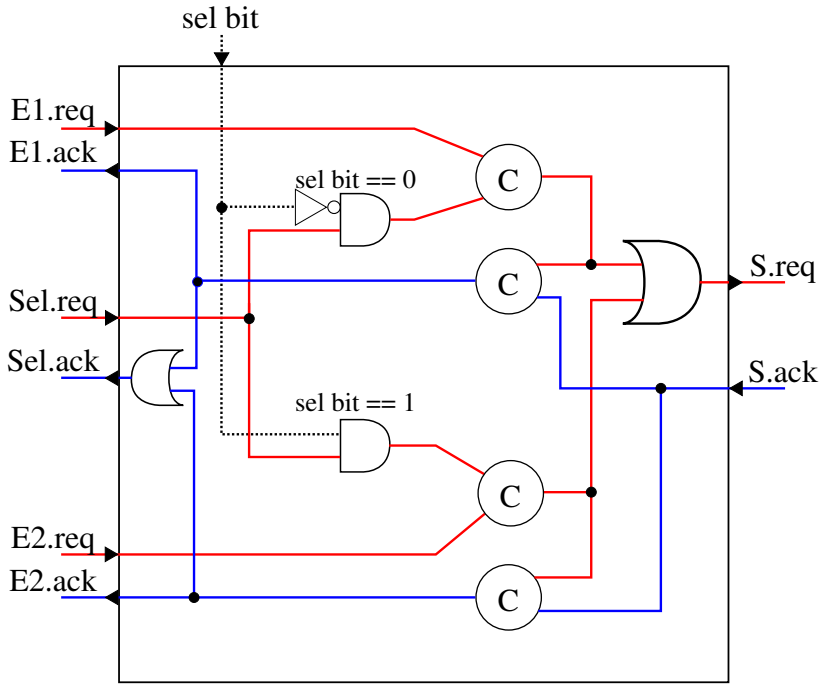


Figure 3.22: Implémentation au niveau portes logiques d'une transition *merge* 2 vers 1

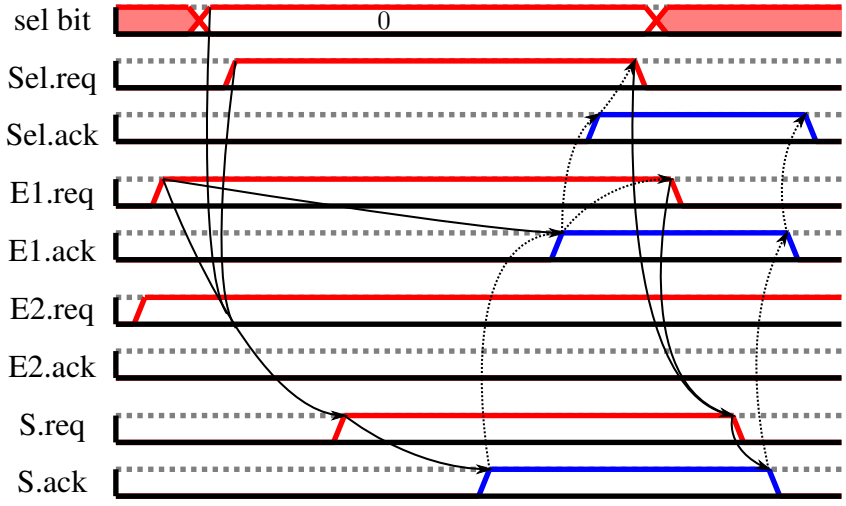


Figure 3.23: Chronogramme d'une requête au travers d'une transition *merge* 2 vers 1

En suivant le déroulement avec le chronogramme, on a donc :

— 2 requêtes arrivent sur les canaux E1 et E2
— la valeur de sélection (sel bit) s'établit et la requête associée à la donnée arrive sur le canal Sel.
— La requête présente sur E1 est transmise, car sel bit est à 0, sur le canal de sortie S.
— L'acquittement de la sortie passe au niveau haut.
— E1 est acquittée car la requête sélectionnée est celle d'E1 et l'acquittement de Sel suit.
— Les requêtes présentes sur E1 et Sel étant acquittées elles reviennent à zéro, et, une fois basses font que la requête de S repasse à zéro.
— L'acquittement de S est maintenant libre de retomber. Lorsqu'au niveau bas, il déclenchera la retombée des acquittements de E1 et Sel.

Durant tout ce processus, une requête est présente sur le canal E2. Cependant, le canal n'étant pas sélectionné, la requête n'a pas été transmise. Nous pouvons aussi remarquer que de la même manière que pour un *join*, 2 requêtes, et donc 2 données, sont nécessaires pour produire une donnée en sortie : une donnée en entrée et une donnée en sélection.

Un elément *merge* 2 vers 1 est composé de :

— 4 *C-elements*
— 2 AND
— 2 OR
— 1 Inverseur

Un module *merge* 2 vers 1 comptabilise finalement une surface approximative de $81\ u.a.$ avec les *C-element* représentant environ 70% du total. Cet élement de sélection est donc de taille conséquente par rapport aux composants vus auparavant et cela sera à prendre en compte dans le compromis surface/performances énergétiques.

3.3.4.4 Transition divergente sélective, le *split*

L'élément complémentaire du *merge* est le démultiplexeur de requêtes, ou *split*. L'utilisation de ce type de composant peut être moins intuitive pour un designer synchrone. En conception synchrone en effet, une donnée est généralement envoyée dans tout les chemins accessibles dans le cône combinatoire. La donnée pertinente est ensuite sélectionnée par un multiplexeur proche de l'entrée du registre suivant.

En ce qui concerne le design asynchrone cependant, contrôler finement dans quelle branche va transiter un *token* présente un intérêt certain puisque cela permet de ne générer de l'activité dans le contrôleur, et dans le chemin de données, uniquement lorsque cela est pertinent. Par cette opération, il est possible d'améliorer l'efficacité énergétique du système global.

Une implémentation au niveau portes logiques du composant *split* est présentée dans la Figure 3.24 et le chronogramme associé à une transaction avec cette structure est présenté sur la Figure 3.25.

Suivant le chronogramme associé à une structure de *split*, la transmission d'une requête vers

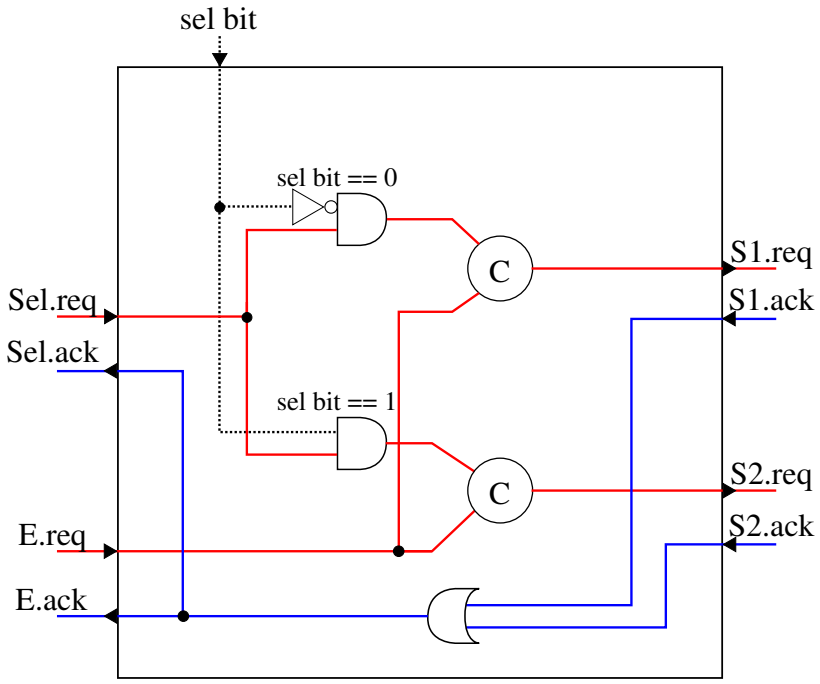


Figure 3.24: Implémentation au niveau portes logiques d'une transition *split* 1 vers 2

le canal S2 va se dérouler de la manière suivante :

- La donnée de sélection s'établit, sel bit passe à 1, et la requête associée est levée dans le canal Sel.
- Une donnée arrive en entrée par le canal E.
- Étant donné que le bit sélection est à 1, la requête est dirigée vers le canal S2.
- Lorsque la requête de sortie est acquittée, l'acquittement est propagé vers les canaux E et Sel.
- Une fois que les requêtes des deux canaux d'entrée sont remises à zéro, alors la requête sur S passe aussi au niveau bas.
- L'acquittement de la sortie retombe et la transaction se termine avec le transfert de donnée et l'acquittement des canaux d'entrées E et Sel.

Pour le *split*, la sortie non sélectionnée, le canal S1 dans l'exemple, ne présente aucune activité. La circuiterie en aval ne sera donc pas sollicitée.

Comme dans le cas du *merge*, 2 données sont nécessaires pour transmettre une donnée en sortie. Ce point sera à prendre en compte dans les vérifications à effectuer pour s'assurer du bon fonctionnement du système que nous analyserons dans la suite de ce chapitre.

Concernant l'estimation de la surface d'une transition *split*, on a donc un module composé de :

- 2 *C-elements*
- 2 AND2
- 1 OR2

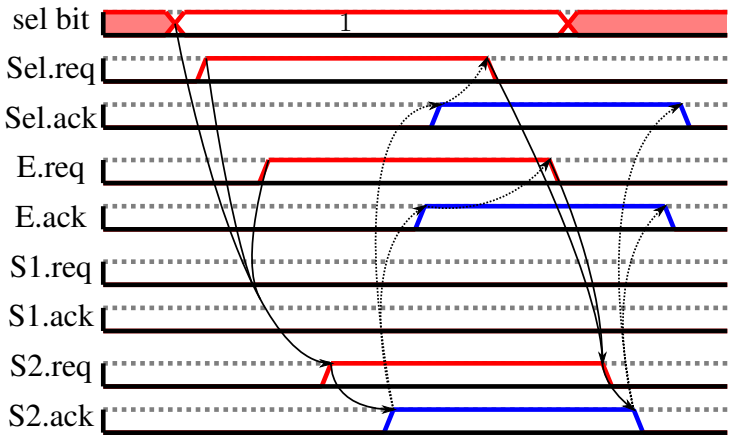


Figure 3.25: Chronogramme d'une requête au travers d'une transition *split* 1 vers 2

— 1 Inverseur

Cette composition nous amène à une surface équivalente de $47\ u.a.$ ce qui représente environ 60% de la surface équivalente d'un *merge* 2 vers 1. Bien que de taille plus modérée, le *split* 2 vers 1 reste non-négligeable en termes de surface. Son utilisation, comme pour le *merge*, devra donc être à mettre en regard avec le gain apporté.

3.3.4.5 Généralisation à plus de deux canaux d'entrée ou de sortie

Un autre avantages des modules asynchrones est qu'avec un interface de communication commune, le canal défini précédement, ils peuvent être facilement associés pour en créer de nouveaux. Que ce soit pour les transitions non-sélectives ou sélectives il est possible de généraliser leur construction à un nombre arbitraire d'entrées ou de sorties en utilisant les structures décrites dans les paragraphes précédents.

Intéressons-nous d'abord aux structures *fork* et *join*. Une manière triviale d'étendre leur fanout, respectivement fanin, est d'assembler les composants 1 vers 2, ou 2 vers 1, que l'on connait déjà. Ainsi, nous obtiendrons alors un arbre de *forks*, respectivement *joins*, comme cela est présenté sur la Figure 3.26.

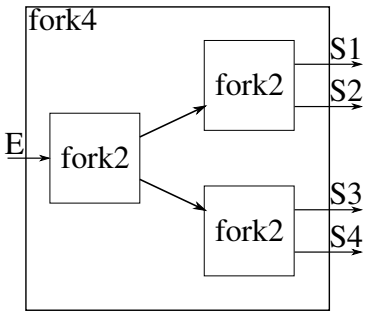
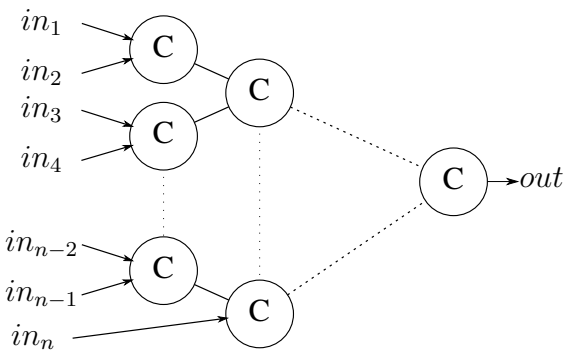


Figure 3.26: Assemblage de modules *fork* 1 vers 2 pour obtenir un *fork* 1 vers 4

Cependant, dans un souci de lisibilité et de maintenabilité du code RTL, il est préférable

d'avoir une implémentation généralisée à toutes les configurations de canaux d'entrée et de sortie possibles.

Pour parvenir à une implémentation généralisée des *fork* et *join* il suffit d'utiliser en lieu et place de l'unique *C-element* des modules à 2 entrée ou à 2 sorties, un arbre de cellules de Muller comme cela est présenté sur la Figure 3.27a. Avec le formalisme des interfaces précédemment présenté, les interfaces des *fork-join* peuvent alors se présenter de manière généralisées tel que dans la Figure 3.27b.



(a) Arbre de *C-element* pour les joins et forks à N entrées/sorties

```
1  module forkn ( E, S);
2    parameter N = 2;
3    Channel.in  E;
4    Channel.out S[NB];
5  endmodule
6
7  module joinn ( E, S);
8    parameter N = 2;
9    Channel.in  E[NB];
10   Channel.out S;
11 endmodule
```

(b) Interface à n canaux d'entrée/sortie pour les transitions fork/join

Afin de pouvoir continuer les comparaisons en termes de surface, il faut que nous soyons capables de dénombrer le nombre de cellules dans un arbre.

Premièrement, on peut remarquer que chaque cellule possède deux entrées pour une sortie. On considère un arbre avec k entrées. Si on ajoute une cellule à cet arbre, un C-element, alors une entrée existante est « consommée », pour brancher la sortie de la nouvelle cellule, et deux nouvelles entrées sont « créées ». Le nombre d'entrées du nouvel arbre est maintenant de $(k - 1) + 2 = k + 1$.

Si l'on prend comme situation initiale un arbre à une entrée et une sortie, autrement dit un fil, on a le nombre d'entrées de l'arbre qui est 1. Avec un raisonnement par récurrence, $n - 1$ cellules seront nécessaires pour obtenir un arbre à n entrées.

N.B. : *Ce raisonnement est valable pour toutes cellules à 2 entrées et 1 sortie. Le résultat sera donc utilisé par la suite pour estimer simplement la taille des arbres de portes logiques classiques (AND, OR,...)*

Etant donné que pour les transitions non-sélectives, seul l'arbre de *C-element* contient des cellules standards, alors la totalité de la surface équivalente telle qu'exprimée jusqu'ici est donc comprise dans cet arbre. Pour un nombre n d'entrées de l'arbre, on aura alors une surface équivalente de $14(n - 1)\ u.a.$

Étant donné que, mis à part l'arbre de *C-elements*, les transitions non-sélectives n'utilisent que des fils, la surface équivalente des composants *fork* et *join* généralisés peut être approchée en utilisant la même formule.

Étendons maintenant l'étude aux transitions sélectives, *merge* et *split*.

Comme vu précédemment, une première manière d'obtenir des structures généralisées est d'utiliser les différents composants 2 vers 1, ou 1 vers 2.

Prenons tout d'abord le cas d'une transition *merge* à 4 entrées. En assemblant directement des modules existants, on obtient l'architecture présentée dans la Figure 3.28.

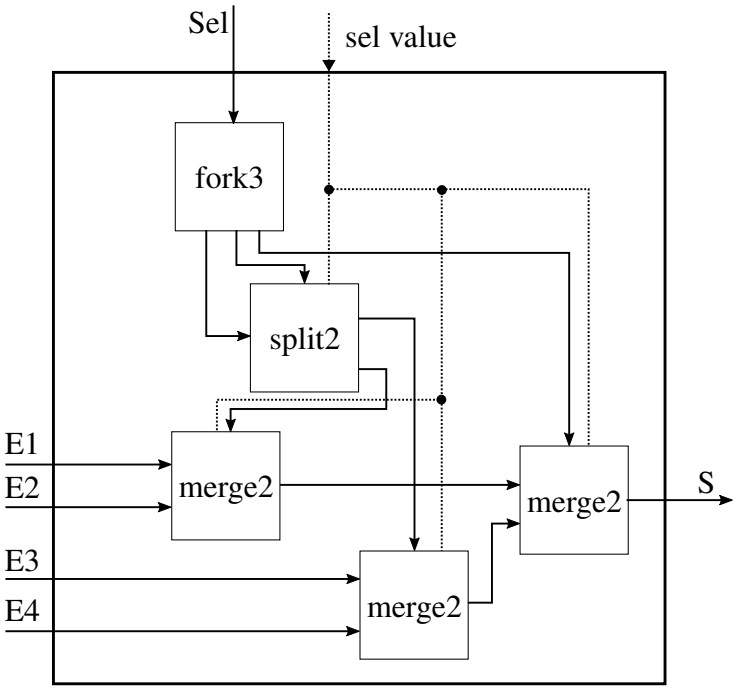


Figure 3.28: Transition *merge* à 4 entrée vers 1 sortie implémentée avec des éléments simples

Comme dans le cas des transitions non-sélectives, nous pouvons remarquer la présence d'un arbre de composants *merge* 2 vers 1. Nous avons en en effet besoin de deux étages de décision avec un premier qui va sélectionner soit les requêtes venant des canaux E1/E2 soit celles venant des canaux E3/E4. La sélection suivante se fait alors entre les deux canaux résultants. Le canal de sélection nécessite cependant un traitement particulier.

Comme pour un *merge* 2 vers 1, on veut consommer une donnée de sélection et une donnée d'entrée pour générer une donnée en sortie. Afin de ne pas avoir de requête de sélection sur les deux *merge* d'entrée, il faut tout d'abord rediriger celle-ci avec un *split* sur l'élément d'entrée concerné.

Si cette redirection de la sélection n'était pas faite, alors une seule requête de sélection pourrait potentiellement traiter deux données en entrée, une pour chaque *merge* d'entrée. La seconde requête traitée ne pourrait cependant pas être transmise à travers le *merge* de sortie car la première donnée le traversant l'aura consommée auparavant.

Au final, pour un composant *merge* 4 vers 1 on a besoin des composants suivants :

— 1 *fork* à 3 sorties pour dupliquer le canal *select*

— 1 *split* à 2 sorties pour diriger le canal *select* sur le bon *merge* d'entrée

— 3 *merges* à 2 entrées qui composent l'arbre de sélection

Un composant *merge* 4 vers 1 construit de cette manière totalise une surface équivalente de $318\ u.a.$ ce qui représente environ 10 bascules D avec la technologie utilisée, mais surtout un facteur de presque 4 par rapport à un *merge* 2 vers 1.

Dans le cas d'une transition *split* 1 vers 4 maintenant, nous pouvons appliquer le même raisonnement. Nous obtenons alors la structure décrite sur la Figure 3.29.

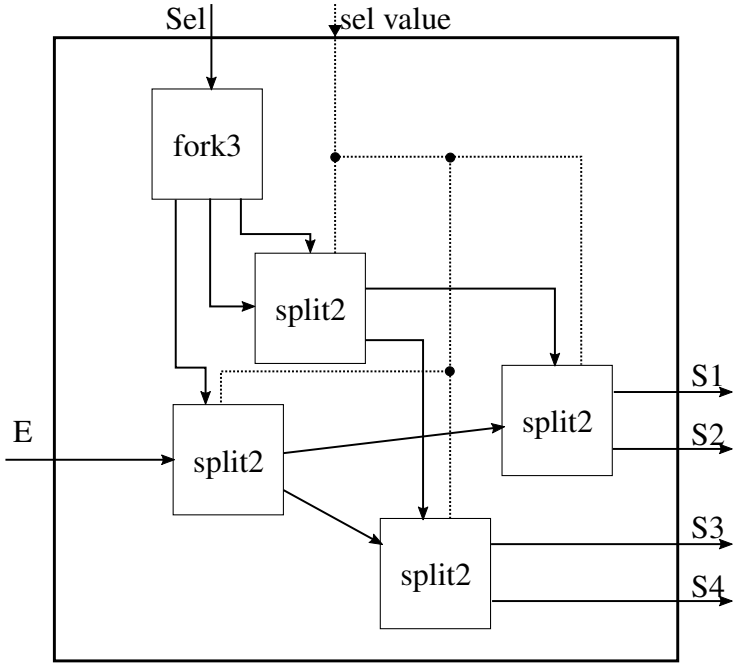


Figure 3.29: Structure *split* 1 vers 4 utilisant un assemblage de composants 1 vers 2

Comme pour le *merge*, nous pouvons remarquer la présence d'un arbre de composants *split* 1 vers 2 auquel s'ajoute la gestion de la requête de sélection. Cette implémentation représente en termes de surface équivalente $230\ u.a.$ soit un peu plus d'un facteur 4 d'augmentation de surface comparé au *split* à 2 sorties.

Afin de limiter cette augmentation de surface avec le nombre d'entrées, une autre méthode pour augmenter le nombre d'entrées d'un composant *merge* peut être envisagée. Nous pouvons en effet remarquer que dans les structures des *merge* à 2 entrées et des *split* à 2 sorties, les chemins de sélection de canaux sont très similaires, seule la valeur du bit de sélection change d'un chemin à l'autre.

Les Figures 3.30 et 3.31 présentent les deux fragments de circuits en question.

Le fragment $InChannelSelect[i]$ permet d'effectuer la fonction de rendez-vous entre une requête de sélection Sel, filtrée par le bit i du signal de donnée sel, et une requête sur le canal d'entrée Ei indexé par i. Il prend aussi en compte le retour de l'acquittement avec le rendez-

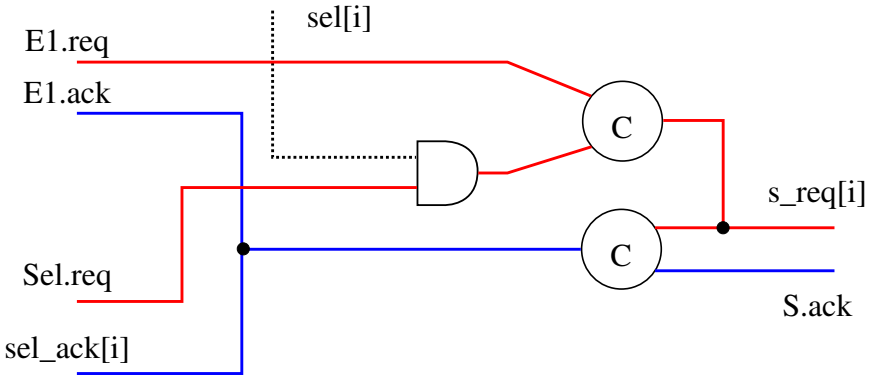


Figure 3.30: Fragment de sélection d'un canal d'entrée indexé par i, $InChannelSelect[i]$

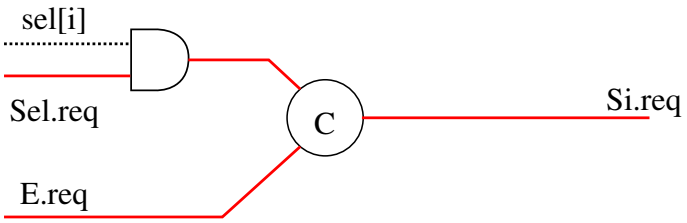


Figure 3.31: Fragment de sélection d'un canal de sortie indexé par i, $OutChannelSelect[i]$

vous entre l'acquittement de sortie et la requête envoyée. Cela donne le signal d'acquittement du canal d'entrée ainsi qu'une partie du signal d'acquittement du canal de sélection. Cet élément de sélection de canal est relativement compact puisque sa surface équivalente est de $33\ u.a.$

Le fragment $OutChannelSelect[i]$ est lui plus simple. La sélection de canal de sortie n'a en effet pas d'aspect de retour d'acquittement. Une seule cellule de Muller est donc nécessaire pour faire se rencontrer la requête d'entrée avec la requête de sélection filtrée par la valeur du bit i du signal sel. Cet élément occupe pour sa part une surface équivalente de $19\ u.a.$.

Utiliser ces fragments de sélection demande cependant d'honorer une contrainte : le signal de sélection doit comprendre autant de bits que de canaux à sélectionner et suivre un codage 1 parmi n, aussi appelé *onehot*. Cette contrainte n'est en général pas excessive puisqu'il suffit d'ajouter la circuiterie combinatoire pour encoder la donnée en amont du contrôleur.

En utilisant ces fragments de circuit, on obtient alors les structures de *merge* et *split* généralisées comme présentées sur les Figures 3.32 et 3.33.

En considérant que les portes ORn sont des arbres de portes OR2, les surfaces équivalentes de ces structures en fonction du nombre n d'entrées ou de sorties peuvent être exprimées de la manière suivante :

— *Merge* n vers $1 : 2(n-1) * 6 + n * 33 = 45n - 12$

— *Split* 1 vers $n : (n-1) * 6 + n * 19 = 25n - 6$

Si l'on compare ces surfaces de composants généralisés avec ceux obtenus auparavant, on obtient $168\ u.a.$ et $94\ u.a.$ au lieu de $318\ u.a.$ pour les *merge4* et $230\ u.a.$ pour les *split4*. Le gain en surface en utilisant ces implémentations généralisées est d'environ un facteur 2 et ce seulement

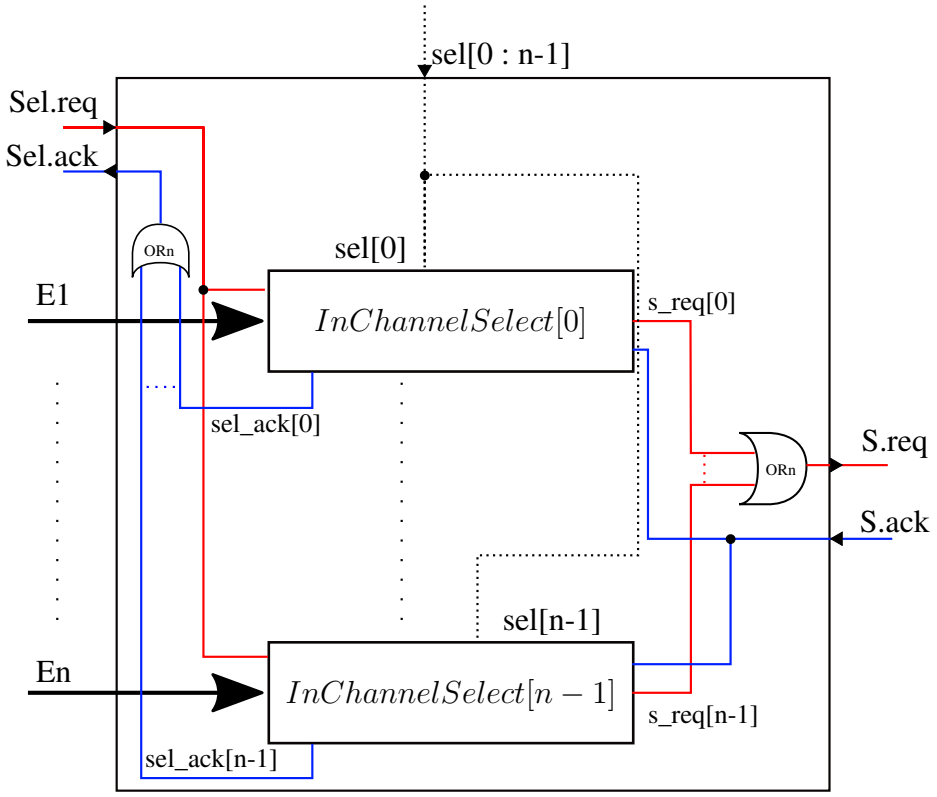


Figure 3.32: Merge généralisé n vers 1 utilisant une donnée de sélection *onehot*, sel

pour un passage de 2 à 4 entrées ou sorties.

En outre, de telles constructions permettent également d'obtenir des transitions sélectives pour n'importe quel nombre d'entrées ou de sorties, ce qui n'est pas forcément aussi aisé en utilisant un assemblage de transitions 2 vers 1, ou 1 vers 2. Avec les implémentations générales présentées, nous avons donc des solutions simples, et paramétrées, à implémenter pour tous les cas de figure que nous pourrions rencontrer par la suite.

3.3.5 Transitions d'extrémité

Un circuit asynchrone ne fonctionne pas si aucune requêtes ne transite en son sein. Symétriquement, un circuit asynchrone « saturé » de requêtes peut se trouver dans une situation menant au blocage. Il n'est plus alors à même de traiter de nouvelles requêtes tant que certaines n'auront pas été consommées en sortie.

3.3.5.1 Transition génératrice de requête, la requête permanente

La transition requête permanente est un élement qui respecte le protocole 4 phases sans avoir la capacité de contenir une donnée tout en fournissant un token dès qu'il en a la possibilité. La construction et le fonctionnement de cette transition sont présentés dans les Figures 3.34 et 3.35.

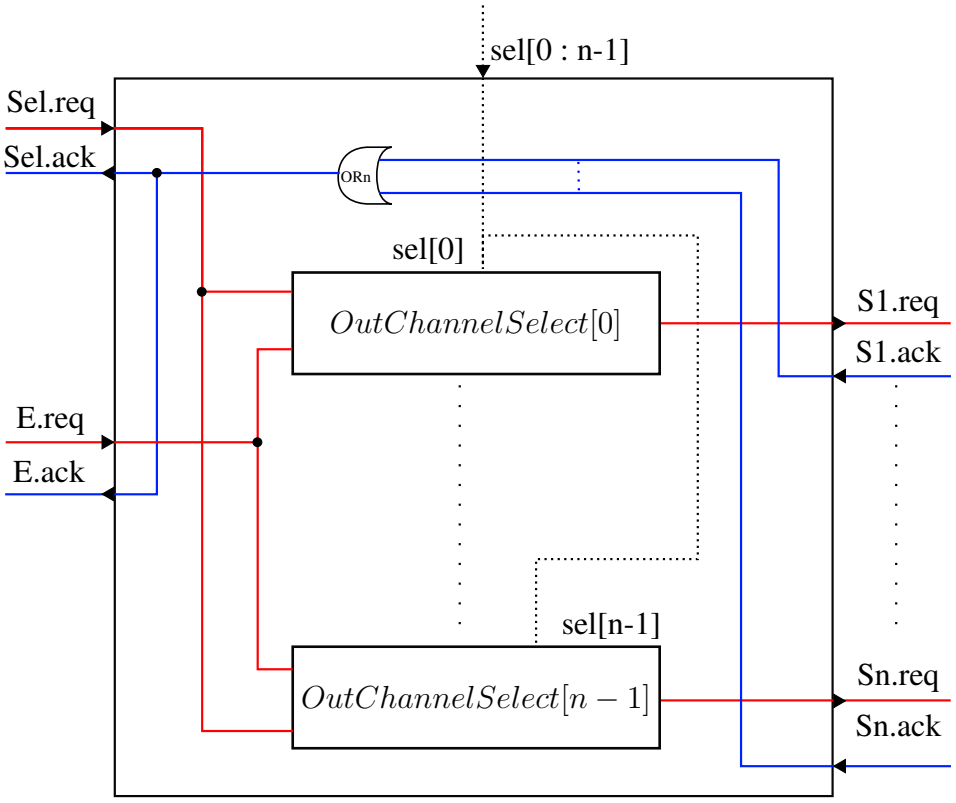


Figure 3.33: Split généralisée 1 vers n utilisant une donnée de sélection *onehot*, sel

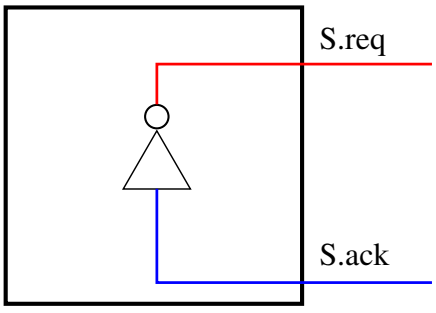


Figure 3.34: Transition requête constante

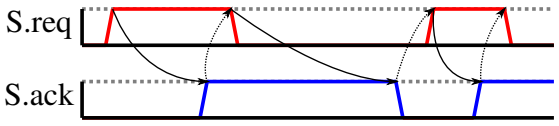


Figure 3.35: Chronogramme de la génération de requêtes avec une transition requête constante

Le principe de fonctionnement de cette requête constante est finalement de générer une requête dès que l'acquittement est à un niveau bas. Symmétriquement, lorsque l'acquittement est à 1, la requête doit repasser à un niveau bas. La surface équivalente utilisée par cette transition est de $3\ u.a.$.

3.3.5.2 Transition consommatrice de requête, le *token trap*

La transition *token trap* est une transition qui sert à consommer une requête en l'acquittant immédiatement après sa réception. La construction et le fonctionnement de cette transition sont présentés dans les Figures 3.36 et 3.37.

Le token trap consiste en un *buffer* qui renvoie un acquittement dès qu'une requête est reçue.

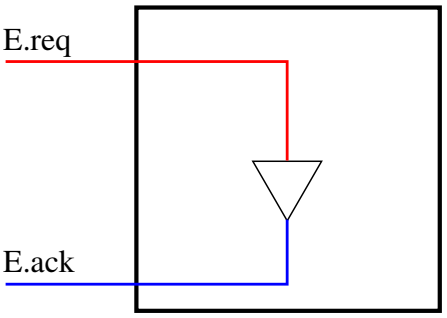


Figure 3.36: Transition token trap

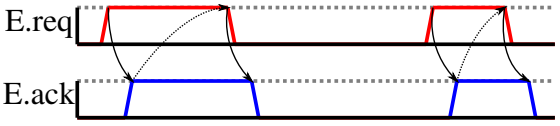


Figure 3.37: Chronogramme de la consommation de requêtes avec une transition *token trap*

Par conséquent, son coût en surface est faible, $4\ u.a.$ voire $0\ u.a.$ si l'on décide décide que le *buffer* utilisé est inutile.

3.3.6 Interfaces avec le monde synchrone

Dans le cadre de ce travail, il n'est pas prévu d'obtenir un circuit totalement asynchrone d'un point de vue global. La principale raison est la nécessité d'utiliser des mécanismes de communication classique comme les interfaces, *Serial Peripherial Interface*(SPI) ou *Inter-Integrated Circuit*(I^2C).

Les circuits asynchrones seront donc des îlots dans un circuit qui sera vu de l'extérieur comme synchrone. Il est cependant nécessaire de permettre au monde synchrone et au monde asynchrone d'interagir entre eux.

3.3.6.1 Interfaces asynchrone vers synchrone

Comme nous l'avons vu dans la Section 2.1.2.1, la contrainte de *setup* doit être respectée, *i.e.* à chaque front d'horloge il faut assurer que la donnée ait eu le temps de s'établir pour assurer qu'elle soit correctement échantillonnée. La problématique est cependant plus complexe que cela puisqu'il existe en réalité une fenêtre encadrant le moment de capture de la donnée, le front montant sur la pin d'horloge dans notre cas.

N.B.*La taille de cette fenêtre, appelée setup-hold window, dépend des paramètres technologiques. Nous aborderons ces deux contraintes avec plus de précisions dans le Chapitre 5.*

Les requêtes dans le circuit asynchrone peuvent arriver à n'importe quel instant en regard des signaux d'horloge. Si le signal d'entrée, par exemple une donnée originaire du monde asynchrone, venait à être capturée dans cette fenêtre, la bascule deviendrait alors métastable et le temps d'établissement de sa sortie ainsi que sa valeur seront imprédictibles. Pendant ce temps de métastabilité, le signal concerné peut également varier de manière aléatoire et donc générer de l'activité inutile dans de la circuiterie combinatoire en aval de la bascule métastable.

Comme démontré dans [31], empêcher totalement la métastabilité n'est pas formellement possible en cherchant à synchroniser des signaux indépendants. Il est cependant possible de

faire en sorte que son impact soit négligeable dans des cas d'utilisation normaux. Pour arriver à cela, le principe est de limiter la probabilité d'avoir une erreur due à la métastabilité, autrement dit, de rendre le temps moyen entre deux erreurs (MTBF, pour *Mean Time Between Failure*) le plus grand possible. Il existe des implémentations sophistiquées comme par exemple dans [30], où est décrite une architecture de synchroniseur utilisant un vote de majorité entre plusieurs bascules potentiellement métastables pour décider quelle valeur propager. Pour notre travail cependant, nous nous contenterons de la structure la plus simple de synchroniseur présentée dans la Figure 3.38, le chronogramme associé est présenté dans la Figure 3.39.

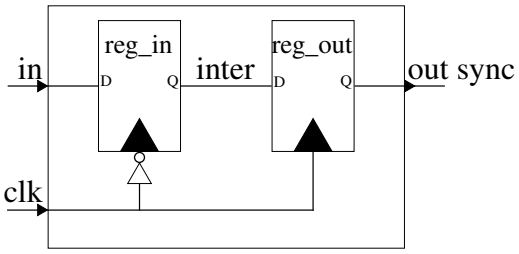


Figure 3.38: Implémentation au niveau portes logiques d'un synchroniseur simple

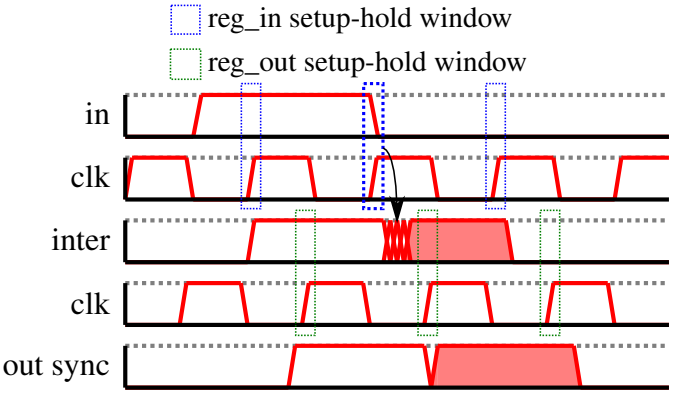


Figure 3.39: Chronogramme d'un transfert de données à travers un synchroniseur

Le synchroniseur utilisé est composé de deux bascules chacune cadencée sur les fronts descendants du signal d'horloge pour le premier registre (reg_in), et sur les fronts montant pour le second(reg_out). Dans ce cas, si une situation de métastabilité se présente, alors le signal $inter$ peut se retrouver instable pendant un certain temps.

Il est à noter que dans nos travaux, nous utilisons des synchroniseurs avec des registres en opposition de phase afin de gagner un demi cycle d'horloge lors d'un transfert du monde asynchrone vers le monde synchrone. Les paramètres de la technologie utilisée nous le permettent mais il est possible que ce ne soit le cas pour une technologie différente.

Il est important de remarquer cependant que la donnée en sortie après une métastabilité peut être erronée bien que stable. Cela implique de prendre des précautions pour éviter d'utiliser des données erronées.

D'autre part, nous pouvons pressentir qu'un synchroniseur est coûteux en surface. En utilisant les mêmes mesures de comparaison qu'auparavant, synchroniser un bit demande 65 $u.a.$. Pour éviter d'avoir un impact rédhibitoire sur la surface, il est préférable d'utiliser ces synchroniseurs pour les signaux de contrôle, qui concernent la validité des données, plutôt que pour les données elles-mêmes.

En utilisant ce composant de synchronisation comme base, nous pouvons finalement créer une variété de modules permettant d'interfacer le monde asynchrone avec le monde synchrone.

3.3.6.2 Interface synchrone vers asynchrone

Dans notre cas, les modules asynchrones étant interfacés avec l'extérieur via le monde synchrone, il est nécessaire de pouvoir écrire des données depuis le monde synchrone vers le monde asynchrone. Nous pouvons distinguer deux types de registres :

— Les registres points d'entrée de pipeline : une écriture du monde synchrone déclenche l'arrivée d'une requête

— Les registres mixtes : ces registres peuvent être écrits de manière indépendante soit par le monde synchrone soit par le monde asynchrone.

Registres points d'entrée de pipeline

Dans ce cas, une commande d'écriture du monde synchrone déclenche l'écriture d'un registre et la propagation d'une requête dans la suite du pipeline.

Une implémentation triviale de cette fonctionnalité est simplement d'utiliser un contrôleur de registre WCHB classique comme vu auparavant tout en l'adjoignant à un synchroniseur pour le retour du signal d'acquittement. Les Figures 3.40 et 3.41 représentent un tel module et son fonctionnement.

Pour pouvoir l'utiliser, il est nécessaire que la circuiterie synchrone en amont de ce module implémente un protocole 4-phases entre les signaux wr et wr_done, qui sont assimilables à un canal tel qu'utilisé dans le reste du contrôleur.

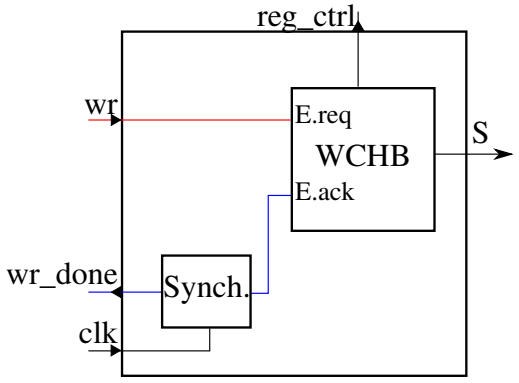


Figure 3.40: Implémentation au niveau portes logiques d'un point d'entrée de pipe asynchrone

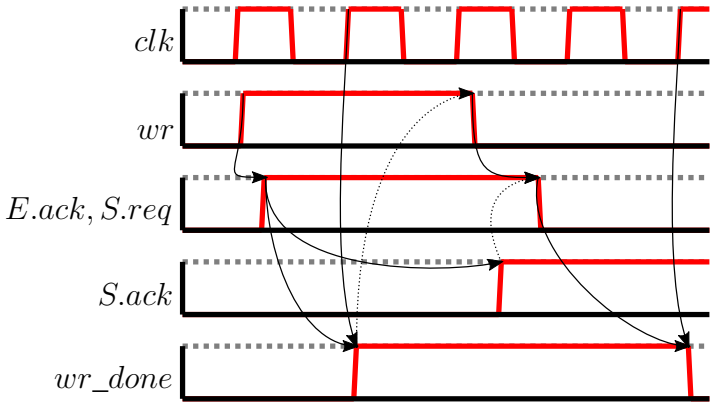


Figure 3.41: Chronogramme d'un transfert de données à travers un point d'entrée de pipe asynchrone

Ce contrôleur a pour surface équivalente 82 $u.a.$.

Suivant les besoins de l'application, une structure de type *First-In First-Out* avec une entrée synchrone et une sortie asynchrone peut être envisagée pour obtenir plus de flexibilité et de performances.

Registres mixtes

Des registres mixtes sont des registres qui peuvent à la fois être modifiés par le monde synchrone

et le monde asynchrone. Les contrôleurs associés à de tels registres n'ajoutent pas de *token* dans le réseau de contrôle et sont le plus souvent associés à des registres présents dans des boucles du contrôleur asynchrone.

L'implémentation de ces contrôleurs est très dépendante de l'interface synchrone habituellement utilisée. Pour cette raison, leur architecture précise n'est pas pertinente et ne sera pas présentée dans ces travaux.

3.3.6.3 Interface asynchrone vers synchrone

Transférer une donnée du monde asynchrone vers le monde synchrone peut se faire de manière assez simple en « synchronisant » le canal *bundled-data* 4 phases. En effet, il suffit pour chaque regitre nécessitant de transférer des données vers l'extérieur de dupliquer le canal de sortie et de convertir le protocole 4-phases asynchrone en un protocole 4-phases synchrone d'une manière proche de celle utilisée pour les registres points d'entrée mais en synchronisant cette fois la requête plutôt que l'acquittement.

Dans les Figures 3.42 et 3.43 sont présentés l'implémentation niveau portes logique et le chronogramme associé.

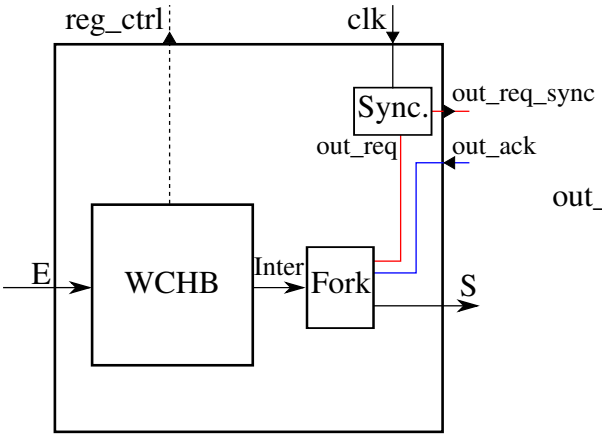


Figure 3.42: Implémentation au niveau portes logiques d'un point de sortie de pipe asynchrone

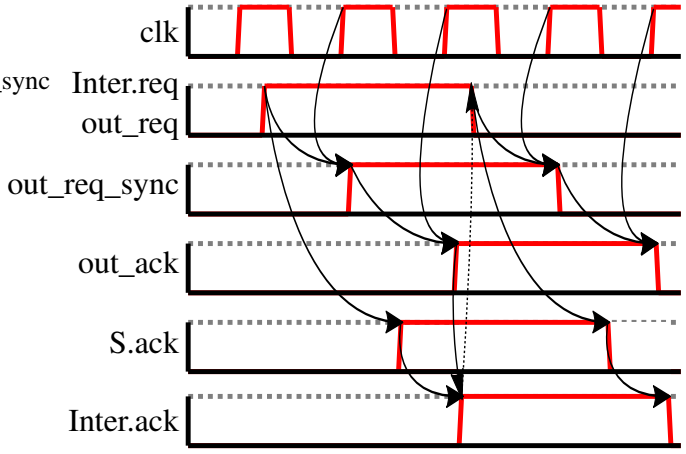


Figure 3.43: Chronogramme d'un transfert de données à travers un point de sortie de pipe asynchrone

En implémentant un protocole 4-phases synchrones en aval, il est alors possible de facilement procéder à des lectures. Il est cependant à noter que ce type d'interface est suceptible de ralentir fortement le systême puisqu'un protocole 4-phases synchrones peu prendre jusqu'à 4 cycles d'horloges. De la même manière que pour les entrées de *pipe*, en lieu et place de cette interface, il est envisageable d'utiliser des FIFO asynchrone→synchrone pour limiter la perte de performances.

Le module présenté a pour surface équivalente $84\ u.a.$ et peut être réduite à $72\ u.a.$ dans le cas où il n'y à pas besoin du canal S, c'est à dire si le registre contrôlé est seulement un point de sortie du circuit sans interaction avec des registres en aval.

3.4 Le réseau du contrôleur : Controller Network(CN)

3.4.1 Construction du *Controller Network*

Nous avons maintenant à notre disposition tous les éléments nécessaires à la création d'un contrôleur asynchrone au niveau global. En partant de l'*Asynchronous Network*, la transformation en *Controller Network* consiste en un remplacement par les éléments de la bibliothèque décrite des différentes transitions. Nous ne considérons que les connexions de chemins de requête (arcs plein dans l'AN).

Tout d'abord intéressons-nous à la correspondance AN/RN avec les différentes combinaisons de nombres d'entrées et de sorties des transitions. En considérant en premier lieu le nombre d'arcs d'entrée (*fanin*) :

— Une transition avec un canal d'entrée et un canal de sortie est une transition « transparente », *i.e.* un simple canal, et ne demande donc aucun matériel particulier.
— Une transition avec un *fanin* de $n > 1$ arcs de requêtes correspond à un module *join* à n canaux d'entrées.
— Une transition sélective de n arcs de requêtes en entrée et un canal de sélection correspond à un module *merge* à n canaux d'entrée, associé à un canal de sélection.
— Une transition sans canaux d'entrée est une requête permanente.

Pareillement, pour le nombre d'arcs de sortie (*fanout*) :

— Une transition avec un *fanout* de $n > 1$ arcs de requêtes correspond à un module *fork* à n canaux de sorties.
— Une transition sélective de n arcs de requêtes en sortie et un canal de sélection correspond à un module *split* à n, canaux d'entrée associé à un canal de sélection.
— Une transition sans canaux de sortie est un *token trap*.

Entre les modules d'entrée et de sortie, traduisant les *fanin* et *fanout*, il faut ajouter un, ou plusieurs, contrôleur(s). Une différenciation est faite en fonction de l'initialisation de la transition. Elle sera considérée initialisée si un *token* est présent sur son, ou ses, arc(s) de sortie.

— Une donnée nécessite deux contrôleurs WCHB pour être totalement contenue comme nous l'avons vu dans Section 3.3. Par conséquent, une transition initialisée se traduira dans le *Controller Network* par deux contrôleurs WCHB accolés, l'un contrôlant un registre, appelé contrôleur de *capture*, et l'autre n'en contrôlant pas, appelé contrôleur *bubble* ou *dummy*. Pour représenter la présence d'une donnée dans ces contrôleurs, celui de *capture* est initialisé avec les sorties S.req, E.ack et reg_ctrl à 1, c'est à dire la sortie du *C-element* à 1.
— D'un autre côté, une transition observable initialisée vide ne nécessite qu'un seul contrôleur WCHB, un contrôleur de *capture*.

En considérant l'AN précédemment étudié et simplifié (Figure 3.12), nous pouvons en déduire le CN associé présenté sur la Figure 3.44.

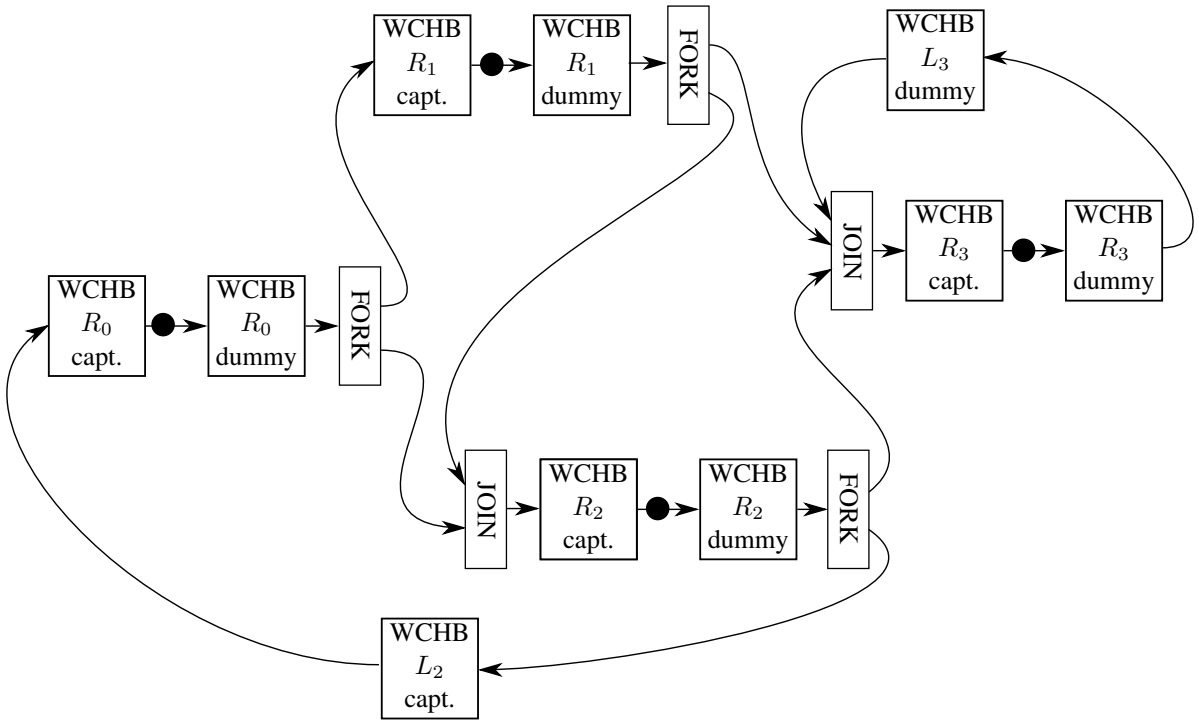


Figure 3.44: Controller Network obtenu à partir du AN simplifié présenté dans la Figure 3.12

Dans la représentation du CN, des *tokens* sont placés sur les arcs représentant les places initialisées pleines dans l'AN. Cela signifie que le contrôleur WCHB précédent doit être initialisé plein. Comme vu dans la Figure 3.18, une requête occupe le canal de part et d'autre d'un contrôleur WCHB, ce qui représente le fait qu'un *token* occupe deux contrôleurs WCHB. À l'initialisation cependant, seulement un contrôleur WCHB a besoin d'être initialisé, c'est pourquoi il n'y a pas de *token* sur les arcs de sorties des contrôleurs WCHB *dummy*.

3.4.2 Contraintes vis à vis de l'arrangement des contrôleurs dans le *Controller Network*

Les contrôleurs WCHB du CN dans la Figure 3.44 ont été placés afin d'obtenir un circuit correct. Il faut cependant être vigilant au placement de ces contrôleurs pour s'assurer de ne pas subir de « perte » de données. En effet, nous avons vu précédemment, dans la Section 3.3, que 2 contrôleurs WCHB sont nécessaires **et** suffisants pour contenir une donnée. Par conséquent, si deux contrôleurs WCHB *dummy* sont accolés, ils peuvent contenir un jeton de donnée sans pour autant être associés à un registre dans le chemin de données.

La Figure 3.45 présente un arrangement de contrôleurs WCHB où une perte de donnée peut survenir.

Intéressons-nous à la propagation de données dans ce fragment de CN :

① En situation initiale, le fragment de contrôleur est vide de toutes données (requête). Les deux registres R_a et R_b contiennent d'anciennes données inconnues. Un *token*, rouge

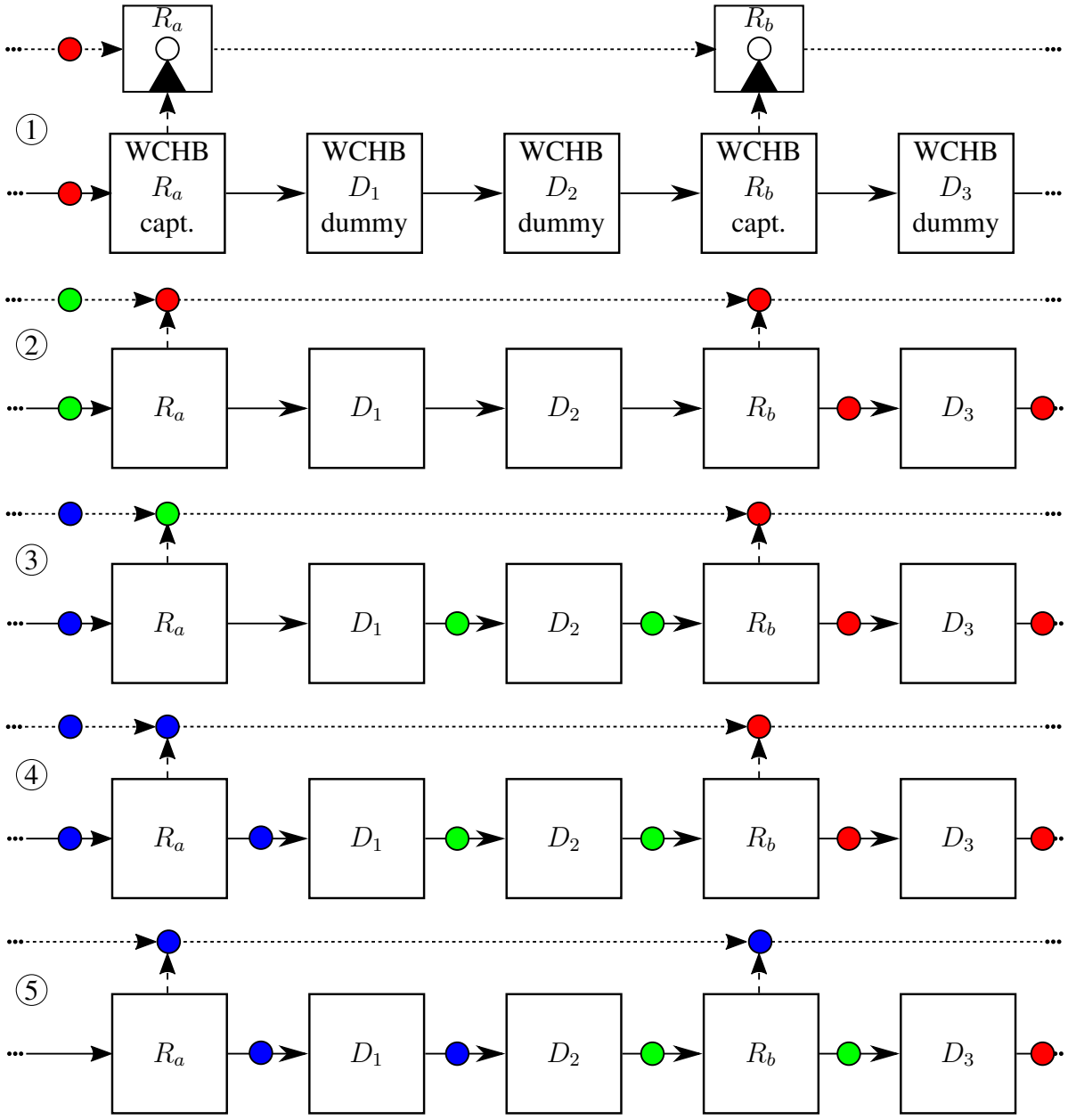


Figure 3.45: Arrangement de contrôleurs WCHB pouvant présenter une situation de perte de donnée

arrive en entrée du fragment de CN.

② Le *token* se propage jusqu'en sortie de D_3 où il se trouve bloqué en attendant qu'une place se libère en aval. Sur son chemin, les deux registres ont été actualisés avec une donnée associée au jeton rouge. Un nouveau *token*, vert, arrive en entrée de R_a.

③ Le jeton rouge bloquant la traversée du contrôleur associé à R_b, le *token* vert se propage jusqu'à l'entrée de ce dernier. Sur son chemin, il actualise la valeur de R_a avec la valeur associée à sa requête. Le jeton vert est contenu sur les deux contrôleurs WCHB dummy D_1 et D_2. Une nouvelle requête, bleue cette fois ci, se présente à l'entrée du fragment considéré.

④ Le jeton vert étant totalement contenu dans D_1 et D_2, il ne bloque pas la traversée du contrôleur associé à R_a. De fait, le *token* bleu traverse ce dernier et actualise la valeur stockée avec celle qui lui est associée. **La donnée associée au *token* vert à été écrasée.**

⑤ Le flot de jetons en aval est débloqué, le *token* vert peut alors traverser R_b. Cependant, la valeur stockée dans R_a étant maintenant celle associée au jeton bleu, R_b prends la valeur associée au jeton bleu.

Ce phénomène a une implication importante dans le cas des boucles pouvant être présentes dans le contrôleur. En effet, dans la Section 3.2, nous avons pu voir que dans ces boucles il est nécessaire d'ajouter des transitions afin d'assurer la propriété de *liveness* du réseau. Cependant, ces transitions se traduisent par un contrôleur WCHB dans le CN et ne sont associées à aucun banc de registres. Il sont implémentés avec des contrôleurs WCHB *bubble*.

Nous pouvons différencier deux types de boucles :

— La boucle unitaire, ou *self-loop*, correspond à un registre rebouclé sur lui-même. D'une part, un seul registre dans la boucle signifie qu'une seule donnée ou *token* peut se trouver dans la boucle. D'autre part, pour assurer la propriété de *liveness*, au moins un *token* doit initialement se trouver dans la boucle. Afin de respecter la Règle 1, il est nécessaire d'ajouter au minimum une transition obeservable dans l'AN, et donc un contrôleur WCHB (de *slack* $1/2$) dans le CN. Si l'on utilise un contrôleur sans groupe de registres associé, la situation présentée dans Figure 3.45 se présentera. Cependant, l'« écrasement » de la donnée n'étant possible que par la donnée elle-même, il n'y aura pas réellement de perte de donnée. Cela n'est cependant valable que si le second contrôleur WCHB *bubble* est présent sur la boucle de retour et non sur un chemin connecté à d'autres registres que lui-même. L'équivalence de traces est conservée dans ce cas puisqu'il n'y a pas de modification du chemin de données.

— La boucle à multiple registres. De la même manière que pour les *self-loops*, pour obtenir un contrôleur vivace (Règle 1) il est nécessaire d'ajouter au minimum une transition observable dans l'*Asynchronous Network*, et donc un contrôleur WCHB dans la boucle associée du *Controller Network*. Nous verrons également plus tard qu'il est nécessaire d'avoir autant de *tokens* initialisés dans la boucle que de registres présents. Par conséquent, l'ajout d'une transition observable dans l'AN et du contrôleur associé dans le RN est tout aussi nécessaire. Si un contrôleur WCHB est inséré en tant que contrôleur *dummy*, il ne sera associé à aucun groupe de registres. Dans ce cas, le lecteur peut s'appuyer sur le fragment $R_0 \rightarrow R_2 \rightarrow L_2$ de la Figure 3.44 pour se convaincre qu'il n'y a pas d'arrangement possible pour éviter une situation de perte de donnée, *i.e.* deux contrôleurs *dummy* adjacents.

  Afin de s'affranchir de cette problématique, le contrôleur WCHB doit être de type *capture* et, par conséquent, associé à un groupe de registres tampons, ou *buffers*, qui sera

le miroir du groupe de registres précédent (connecté directement). Cette opération ne garanti pas la conservation de l'équivalence de trace. Cependant en plaçant une barrière de registres buffers, la transition ajoutée n'impacte en rien le flot de données « réel » observé. Il est à noter que l'ajout de ces registres est à limiter au maximum puisqu'il peuvent rapidement devenir coûteux en surface étant donné qu'un bit de registre représente une surface équivalente de $31\ u.a.$. La problématique de placement de ces registres *buffers* sera l'un des sujets du Chapitre 4.

Une règle à respecter afin de ne pas se heurter au problème de perte de donnée a été définie :

Règle 2 (*Intégrité des données*)**.** *Deux contrôleurs WCHB de type capture ne doivent pas être séparés par plus d'un contrôleur WCHB de type bubble. Les boucles de retour d'une self-loop sont exemptes de cette contrainte.*

En respectant cette dernière règle, on obtient avec le CN une représentation d'un contrôleur implémentable en utilisant une bibliothèque de cellules standards.

Chapitre 4

Méthode de désynchronisation

Avec les différentes modélisations présentées dans le Chapitre 3, nous pouvons maintenant nous atteler au cœur du problème : la désynchronisation.

Nous allons commencer par présenter une approche directe que nous appellerons approche standard puis nous nous intéresserons à l'optimisation des contrôleurs obtenus afin de rendre la désynchronisation plus simple mais aussi plus efficace.

4.1 Approche standard de la désynchronisation

Cette approche est proche de celle présentée par Cortadella *et al.* dans [15] mais au lieu d'utiliser des *latches master* et *slave*, nous utilisons des bascules D plus classiquement utilisées dans l'industrie.

Environnement de développement

La méthode de désynchronisation peut être appliquée de manière manuelle en ayant une bonne connaissance du *circuit*. Cependant, la taille et la complexité des circuits peuvent rapidement devenir importantes et difficiles à gérer manuellement. C'est pourquoi dans la suite de ce manuscrit nous considérerons une approche qui utilise des listes de commandes, ou scripts, qui offrent une adaptabilité et un automatisme bienvenus.

Tout au long de cette thèse, nous avons utilisé des outils standards utilisés dans l'industrie du semi-conducteur. Le langage d'automatisation des opérations, ou langage de script, utilisé par ces outils est le *Tcl*. Ce dernier, ainsi que les outils de la suite *Synopsys*™ étant en constante évolution, certains scripts développés au moment de la thèse ne fonctionneront peut être pas avec d'autres versions d'outils. Nous donnons à titre informatif les différentes versions des langages et outils utilisés :

Tcl 8.6 (intégré dans les outils *Synopsys*™)

Tcllib 1.19 [59]

Synopsys Design-Compiler™ L-2016.03-SP4

Synopsys Primetime™ L-2016.06-SP2

Synopsys IC Compiler™ L-2016.03-SP4

La *Tcllib* est nécessaire pout notre travail puisqu'elle implémente les représentations sous forme de graphes et de matrices dont nous aurons besoin dans la suite. Cependant, les outils industriels n'autorisent pas la modification du cœur *Tcl* installé dans leur outils. Afin d'utiliser ces fonctionnalités avancées, il faut donc manuellement ajouter la librairie au chemin reconnu par l'interpréteur.

Á titre d'information, la méthode utilisée pendant cette thèse pour pouvoir utiliser la *Tcllib* dans les outils *Synopsys*™ a été la suivante :

— Télécharger de la bibliothèque *Tcllib*.

— Executer le fichier *'installer.tcl'*, choisir le dossier d'installation de *Packages* dans la fenêtre interactive.

— Dans le répertoire choisi pour les *Packages*, ajouter le script *'load_tcllib.tcl'* contenant :

```
if {![info exist _dir]} {
    set _dir [file dirname [file normalize [info script]]]
}
source [file join $_dir pkgIndex.tcl]
```

— Dans l'outil, éxecuter le fichier *'load_tcllib.tcl'*.

Au terme de ce chapitre, nous aurons vu la méthode nécessaire pour obtenir une *netlist* qui représente le circuit sous forme de portes logiques et d'interconnexions. Pour rendre le système physiquement implémentable et fonctionnel, il faudra par la suite considérer les délais réels de propagation au sein du circuit. Nous nous intéresserons à cette partie dans le Chapitre 5 et nous verrons aussi les mesures prises dans le cadre de nos travaux afin de d'obtenir des circuits désynchronisés fonctionnels une fois implémentés sur silicium.

Les différentes étapes de la synthèse dans le cadre d'une désynchronisation sont présentées dans le graphe de flux de la Figure 4.1.

Dans la suite de ce chapitre, nous allons nous intéresser aux détails de chaque étape du flot de synthèse de la désynchronisation.

N.B. : *Les transitions sélectives ne sont pas utilisées automatiquement dans la procédure de désynchronisation proposée dans la suite. Des essais ont néanmoins été entamés afin de les exploiter automatiquement. Il s'avère cependant que les circuits synchrones n'étant généralement pas conçus pour tirer parti de la « directivité » des données, nous n'avons pas abouti à une solution assez générale pour la présenter dans ce manuscrit.*

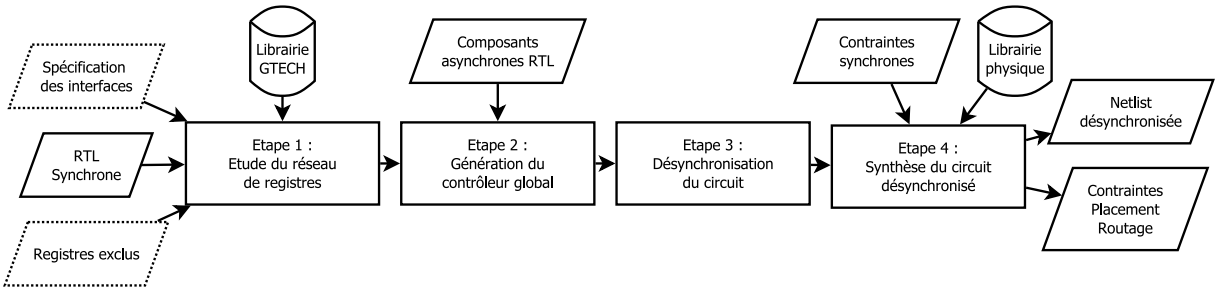


Figure 4.1: Flot de synthèse de la désynchronisation, vue globale

4.1.1 Étude du réseau de registres

La base de notre méthode de désynchronisation repose sur le *Register Network* qui représente les interconnexions entre les registres. Nous pouvons en effet à partir de cette représentation déduire l'architecture du contrôleur global à implémenter et, grâce à l'effort de paramétrage des composants asynchrones fournis dans la Section 3.3, le générer.

La première étape de génératio et d'analyse du RN peut se décomposer comme dessiné dans la Figure 4.2.

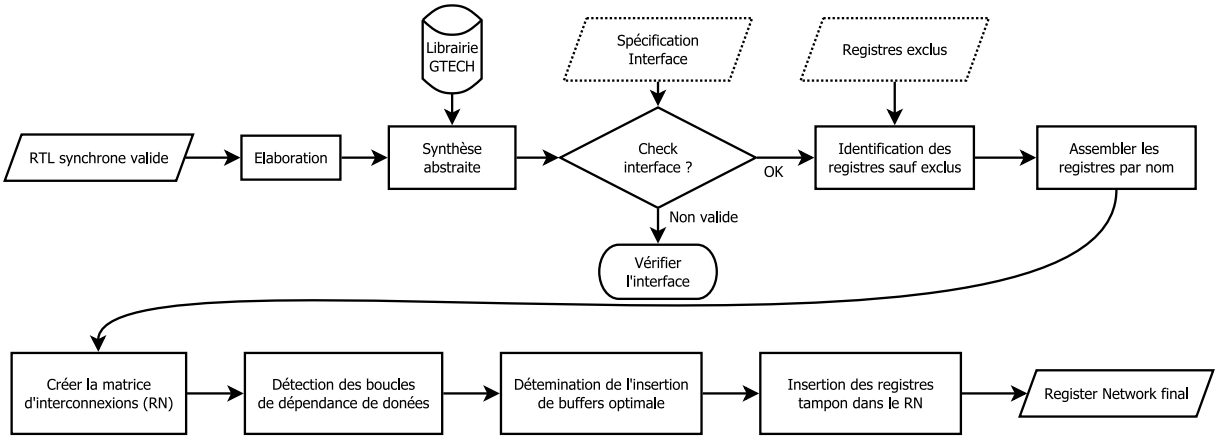


Figure 4.2: Flot de désynchronisation, détail de l'étape 1 : Étude du réseau de registre

Tout d'abord, nous pouvons distinguer 4 éléments d'entrée :

— Le RTL synchrone du circuit à désynchroniser. Il est supposé valide en fonctionnement synchrone.

— Une librairie de cellules (GTECH est fournie par les outils *Synopsys*™) ou toute autre librairie virtuelle. L'utilisation d'un libraire virtuelle permet de ne pas dépendre de la technologie utilisée pour les premières étapes du flot.

— La spécification des interfaces (optionnelle) est nécessaires dans le cas de registres mixtes (registres pouvant à la fois être écrits par le monde synchrone et par le monde asynchrone) et dépend de l'environnement dans lequel sera intégré le circuit désynchronisé.

— La liste des registres exclus (optionnelle) est la liste des registres à ignorer lors de la désynchronisation (s'ils existent).

4.1.1.1 Initialisation des données nécessaires à la désynchronisation

Dans un premier temps, intéressons-nous aux étapes allant de l'élaboration jusqu'à l'assemblage des registres par nom.

Élaboration

Lors de l'élaboration, le code RTL est vérifié être synthétisable et est ensuite converti sous la forme de structures de données représentant les blocs matériels « génériques ». La commande d'élaboration est récursive. Autrement dit, elle s'effectuera automatiquement sur les niveaux de hiérarchie inférieurs dès lors que la spécification RTL de ces derniers est disponible dans les fichiers d'entrée. Á cette étape, chaque module est vu comme une « boite » et les différents éléments de la hiérarchie sont connectés entre eux avec des fils, ou *wires*. Si un module n'est pas défini, il sera alors considéré comme un module inconnu sans fonctionnalité appelé *black-box*. La structure obtenue ne permet cependant pas de tracer les chemins de données du circuit correspondant à une technologie (ce qui nous sera nécessaire pour la suite). En effet, à ce stade, les opérations ont été réalisés indépendamment de la technologie.

Check interface

Dans le cas de l'utilisation d'une interface avec l'extérieur , des signaux seront utilisées pour gérer les transferts. Cependant, le moteur de synthèse peut, dans un souci d'optimisation, faire disparaitre, ou simplement renommer, un signal crucial pour cette interface. Afin d'éviter ce genre de désagrément, il est important de vérifier que ces signaux existent et soient protégés d'une procédure de d'optimisation en utilisant la commande *'set_dont_touch'* sur les fils pertinents. Dans le cas contraire, la procédure sera arrêtée et le concepteur devra vérifier la spécification de l'interface.

Synthèse abstraite

Pendant l'étape de synthèse abstraite, l'outil utilise la librairie générique GTECH et synthétise le circuit en utilisant ces cellules standards génériques. Bien qu'elle ne soit pas implémentable au niveau physique, la bibliothèque GTECH permet de refléter la fonctionnalité logique de chaque cellule et donc d'identifier les chemins combinatoires dans le circuit. Le moteur de synthèse effectue également pendant cette phase un certain nombre d'optimisations logiques. Il est à noter que les cellules n'ayant pas d'empreinte physique en termes de surface, de consommation et de performances, les optimisations qu'il peut apporter se cantonnent à des optimisations logiques telles la simplification de signaux constants ou le réarrangement de la logique combinatoire.

Identification des regitres et assemblage

Design-Compiler offre une commande permettant de récupérer tous les registres du circuit courant. Cette commande, *'all_registers'*, renvoie cependant chaque bit de chaque registre comme une entité à part entière.

De cette collection, en suivant le jargon des outils, le concepteur peut choisir d'exclure les registres qu'il ne veut pas voir apparaitre dans le circuit désynchronisé. L'exemple type de registre qu'il ne serait pas souhaitable de désynchroniser sont les registres utilisés pour effectuer une mesure de temps, qui sans le temps d'échantillonnage synchrone, n'auraient pas de sens.

Chaque bascule est pour l'instant considérée comme étant un registre en lui-même. Bien que cette approche ait du sens d'un point de vue asynchrone, la complexité résultante de la représentation sous forme de RN, et par la suite sous la forme d'un contrôleur global, deviendrait très vite prohibitive en temps de calcul et en surface. Pour limiter cette complexité, il est naturel de regrouper les registres d'un même vecteur ensemble.

Au terme de ces opérations, nous avons à disposition un dictionnaire qui indexe par nom de registre la liste des bascules associées et qui nous permettra de retrouver un certain nombre d'informations.

4.1.1.2 Création et étude du *Register Network*

Création de la matrice d'interconnexion et du graphe associé

Le RN sous forme matricielle est une matrice carrée avec autant de lignes et de colonnes que de registres dans le circuit. Cette matrice est d'abord créée de la bonne dimension, remplie de 0, chaque élément étant associé à la connexion d'un registre vers un autre comme indiqué dans Section 3.1, à la différence près qu'on utilise les valeurs 1 pour la présence d'une connexion et 0 dans le cas contraire.

Afin de déterminer si un registre R_j est dépendant de la valeur d'un registre R_i, on utilise la commande permettant de tracer le chemin entre la sortie d'un registre source et l'entrée d'un registre destination *'get_timing_paths -through R_i/Q -to R_j/D'*. Si celle-ci indique qu'un chemin existe, alors l'élément d'indice (i, j) prend la valeur 1. L'opération est répétée pour chaque combinaison i, j.

Au terme de cette procédure, une première forme de réseau de registres est obtenue. Nous l'appelerons $M_{RN_{STD}}$ dans la suite.

Avec cette matrice d'interconnexion, nous pouvons également utiliser un autre élément de la *Tcllib* : la structure *graph* qui va nous permettre de parcourir celui-ci en utilisant des algorithmes connus. La traduction de $M_{RN_{STD}}$ en graphe dirigé est assez directe :

— Chaque registre R_i est représenté par un nœuds, N_i

— Pour chaque combinaison (i, j) possible, si $M_{RN_{STD}}[i, j]$ est égal à 1, alors on ajoute un arc entre N_i et N_j

Le graphe dirigé obtenu sera appelé dans la suite RN_{STD}.

Détection des boucles de dépendances de données

Avant de créer l'*Asynchronous Network*, il est nécessaire d'identifier toutes les boucles présentes dans le circuit afin de pouvoir déterminer la position optimale pour l'ajout des transitions observables dans l'AN.

Utilisant le modèle de graphe dirigé, RN_{STD}, nous pouvons nous appuyer sur de nombreux travaux portant sur la recherche de cycles dans ces graphes : [24, 28, 58, 60].

Les travaux de Hawick et James utilisent un certain nombre de travaux précédents sur la problématique de recherche des cycles dans un graphe dirigé et propose un algorithme qui permet d'obtenir de manière exhaustive tous les cycles d'un graphe dirigé, qu'il soient élémentaires ou non.

Nous n'allons pas entrer dans les détails de l'algorithme étant donné que ce n'est pas le sujet des travaux de cette thèse. Nous considérerons que l'application de la méthode renvoie l'ensemble des boucles présentes dans le RN sous forme de liste, en conservant l'ordre de passage dans chaque nœud de chaque boucle. Nous appellerons cette liste $loops$.

Chaque boucle dans la liste obtenue est considérée comme unique, c'est à dire qu'une liste d'élément d'une boucle représente toutes les permutations circulaires possibles des éléments de cette boucle.

4.1.1.3 Positionnement des registres tampons

Détermination de la position optimale des *buffers*

L'insertion des buffers est une nécessité pour le respect de la Règle 2. Le coût de l'ajout de ces registres tampons peut être significatif, il est donc crucial de faire les choix permettant de limiter leur impact en surface.

Pour appliquer la règle d'intégrité des données, chaque boucle de plus d'un élément doit contenir au moins un contrôleur de *capture* initialisé vide et son registre *buffer* associé.

Afin de différencier les registres, nous avons utilisé une fonction de coût (Équation (4.1)) permettant d'estimer l'impact relatif de l'ajout de ces registres *buffer*.

N.B. *Dans la suite, nous utiliserons le verbe « bufféri**ser** » pour décrire l'action d'ajouter des registres tampons à des registres.*

$$BuffCost(R) = \frac{Area(WCHB) + Area(DFF).Size(Reg)}{OccurencesInLoops(R)} \tag{4.1}$$

La procédure utilisée pour déterminer quels registres sont les plus intéressants à « *bufferiser* » consiste en la suite d'opérations suivante :

— On regroupe chaque registre présent dans une boucle (non-unitaire) dans une liste.

— Pour chaque registre de cette liste, on associe un poids calculé avec la fonction de coût de l'Équation (4.1).
— On trie la liste de registres dans l'ordre croissant des coûts.
— On considère le premier élément de cette liste comme bufférisée et on considère donc toutes les boucles le contenant comme valides.
  Tant que toutes les boucles ne contiennent pas au moins un registre bufférisé, on répète l'opération sur l'élément de la liste de registre suivant.

Au final, nous obtenons une liste de registres sur lesquels il faut appliquer la *bufférisation* de manière à rendre toutes les boucles à plusieurs registess valides.

Il est à noter qu'une solution exhaustive calculant toutes les combinaisons de *bufférisations* possibles afin de trouver la moins « coûteuse » a été envisagée. Néanmoins, les temps de calculs alors mis en jeu nous ont incité à utiliser cette méthode plus simpliste qui donne des résultats concluants pour nos applications.

Insertion des registres *buffers* dans le RN Nous avons maintenant à notre disposition la liste des registres à bufférisers rendant valide notre modélisation et par conséquent le contrôleur asynchrone qui en résultera. Le réseau de registres étant utilisé par la suite pour effectuer les opérations, nous reflétons cette *bufférisation*sur celui-ci.

Dans le RN_{STD}, la procédure appliquée pour chaque registre à bufférisers peut se résumer, pour un registre R_i, et sa place associée N_i, à :

— On récupère toutes les places reliées en aval de N_i, $Loads(N_i)$
— On supprime tous les arcs sortant de N_i
— On crée la place associée au *buffer* $(N_i)_{buff}$
— On crée l'arc $N_i \rightarrow (N_i)_{buff}$
— On crée tous les arcs $(N_i)_{buff} \rightarrow Loads(N_i)$

Un exemple de cette insertion de buffer est présenté sur la Figure 4.3.

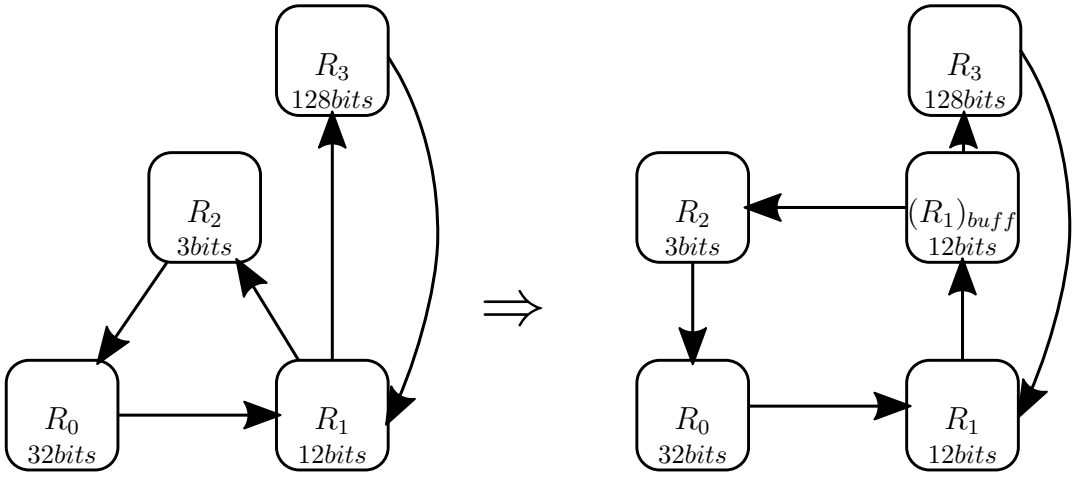


Figure 4.3: Exemple de choix et d'insertion d'un registre *buffer* dans un réseau de registre

Dans cette situation, deux boucles sont présentes : $R_0 \rightarrow R_1 \rightarrow R_2$ et $R_1 \rightarrow R_3$. Pour la première boucle, le bon choix semblerait être de bufférisier R_2 puisque c'est le plus registre de moins grande taille.

Cependant, ne rendrait pas pour autant la boucle $R_1 \rightarrow R_3$ valide. Il serait donc nécessaire d'ajouter un *buffer* dans cette boucle. Le choix évident pour cette dernière boucle est d'appliquer la *bufférisation*à R_1 puisqu'il est beaucoup plus petit.

Par ailleurs, nous pouvons remarquer qu'en bufferisant R_1, les deux boucles deviennent valides. La solution choisie est finalement de bufferiser uniquement R_1.

On insère donc le registre $(R_1)_{buff}$ de même taille que R_1 et les connexions sont reportées comme cela est précédemment décrit.

4.1.2 Génération du contrôleur asynchrone global

Nous avons maintenant à notre disposition la version finale du *Register Network* ainsi que les composants de la bibliothèque asynchrone décrits dans la Section 3.3.

En suivant le flot présenté sur la Figure 4.4 nous pouvons aboutir à un contrôleur asynchrone global qui pourra par la suite être utilisé pour désynchroniser le circuit.

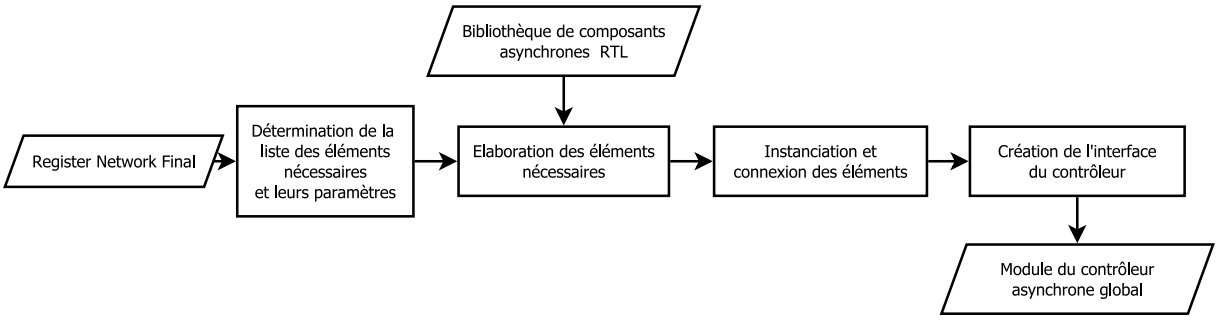


Figure 4.4: Flot de désynchronisation, détail de l'étape 2 : Génération du contrôleur global

Détermination et élaboration des structures asynchrones nécessaires

Comme nous l'avons vu dans le Chapitre 3, les transitions d'une modélisation à une autres sont assez directes et nous travaillerons par la suite directement avec le RN pour déduire les éléments dont nous avons besoin.

La commande précédemment utilisée pour construire le circuit synchrone permet également de construire des modules en prenant en compte des paramètres. En considérant le RN_{STD} final obtenu lors des étapes suivantes, la liste des structures nécessaires peut être déterminée de la manière suivante :

— Si le circuit comprend au moins un registre, on ajoute à la liste un ensemble de deux contrôleurs WCHB l'un de *capture* initialisé plein, et l'autre de *bubble* initialisé vide. Ce dernier pourra également servir si le circuit comprend des *self-loops*.

— Si le circuit contient au moins un registre *buffer*, on ajoute à la liste de structures un contrôleur WCHB de *capture* initialisé vide.
— Pour chaque registre, on s'intéresse maintenant au nombre d'arcs d'entrée :
  — S'il existe un registre sans arc d'entrée, alors cela correspond soit à une requête permanente soit à un registre d'entrée. Si au cours des précédentes étapes une connexion vers le monde synchrone en entrée de ce registre à été détectée, alors on ajoute un registre de type entrée à la liste de structures. Sinon, une structure de requête permanente est ajoutée à la liste.
  — Si plusieurs arcs d'entrée sont détectés, alors une structure *join* correspondant au nombre d'entrées est ajoutée.
— De la même manière, en considérant le nombre d'arcs de sortie :
  — S'il existe un registre sans arc de sortie, alors cela correspond soit à un *token-trap* soit à un registre de sortie. Si une connexion vers le monde synchrone en sortie de ce registre a été relevée dans les étapes précédentes du flot. Sinon, on ajoute un registre de sortie à la liste de structures, un *token trap*.
  — Si plusieurs arcs de sortie sont détectés, alors une structure *fork* correspondant au nombre de sorties est ajoutée.

Lorsque la liste est constituée, chaque élément la composant peut être « élaboré ». Chaque élément est alors associé à la place dans le RN qui lui correspond. Au terme de cette étape, chaque place du RN_{STD} référence au final la structure abstraite nécessaire pour son interface d'entrée, son contrôleur, et son interface de sortie. Nous avons maintenant dans la base de donnée de l'outil tout ce qu'il nous faut pour générer le contrôleur.

Création du contrôleur global

La première étape à effectuer est de créer un nouveau module, avec la commande *'create_design'*, qui sera notre circuit de contrôle global.

Au sein de celui-ci, chaque place du RN_{STD} référençant les modules abstraits qu'elle doit implémenter, on utilise maintenant la commande *'create_cell'* qui instancie les structures d'entrée, de contrôleurs et de sorties de chaque place. Pour les étapes suivantes, on ajoute la référence de la cellule créée aux attributs de la place du RN_{STD}.

Il est à noter que s'il existe plusieurs places instanciant le même module abstrait, 2 places nécessitant un *fork* à 2 sorties par exemple, l'outil ne dupliquera pas cet élément en mémoire. Bien que cela permette d'optimiser le coût mémoire de la synthèse, ce n'est pas souhaitable dans notre cas puisque cela ne nous permet pas de connecter ensemble les différents modules. Afin de pouvoir procéder à cette connexion, il faut appliquer la commande *'uniquify'* qui va différencier chaque instance de module en avec un nom qui lui est propre.

Tous les modules étant créés, il faut maintenant les connecter entre eux suivant le RN_{STD}.

Pour utiliser la modularité des modules asynchrones et leur connexion générique utilisant les canaux, nous avons développé pendant nos travaux une procédure *Tcl* qui permet d'automatiser la connexion, via un canal, d'un module source jusqu'à un module de destination.

Cette fois encore, en se basant sur les informations contenues dans le graphe RN_{STD}, on connecte les différents modules entre eux.

Création de l'interface

La dernière étape à effectuer pour obtenir un contrôleur complet est de créer son interface. Fonctionnellement, l'outil n'en aurait pas besoin puisque l'on pourrait « traverser » les étages de hiérarchie sans avoir les interfaces existantes, elles seraient créées à la volée.

Cependant, les signaux d'interface du contrôleur asynchrone vont majoritairement être les signaux contrôlant les différents registres. Il est donc pertinent de faire en sorte qu'un concepteur humain puisse facilement les identifier ce qui ne serait pas aussi évident avec une interface créée automatiquement par *Design-Compiler*™.

Chaque contrôleur de registre, WCHB de *capture*, délivre un signal qui viendra remplacer la connexion à l'arbre d'hologe de chaque registre. Pour chaque place dans le graphe RN_{STD}, on ajoute donc un port vers l'extérieur qui prendra le nom du registre qu'il est censé contrôler suffixé de $'\_reg\_ctrl'$.

Dans le cas d'utilisation d'interfaces synchrone vers asynchrone, et leurs symétriques, il faut également créer des connexions correctes afin de retrouver les signaux pertinents sans avoir à analyser la hiérarchie générée du contrôleur, qui est difficilement intelligible par le designer.

4.1.2.1 Désynchronisation et synthèse du circuit

Les dernières étapes de la partie synthèse du flot de désynchronisation consistent en la transformation du chemin de données synchrone en un circuit désynchronisé. Les étapes de ce procédé sont décrites sur la Figure 4.5.

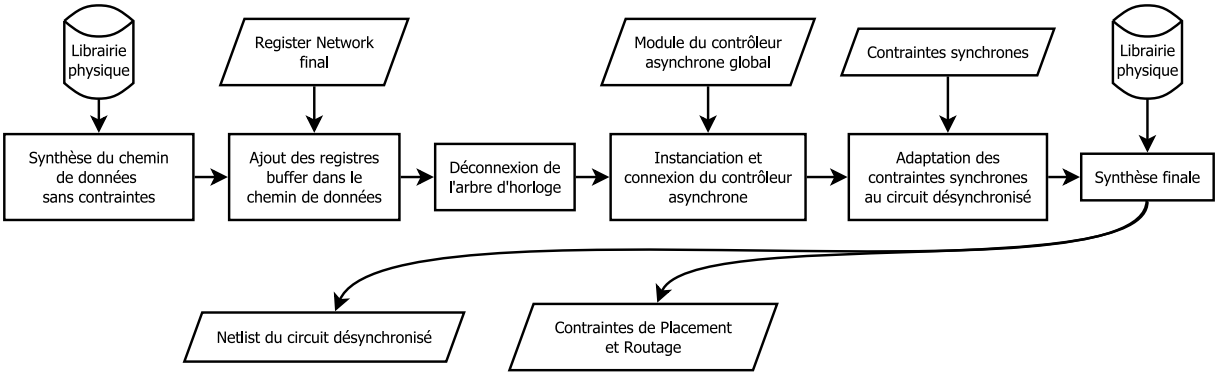


Figure 4.5: Flot de désynchronisation, détail des étapes 3 et 4 : Désynchronisation et synthèse du circuit désynchronisé

Synthèse préliminaire et ajout des registres *buffers* dans le chemin de donnée

Avant de procéder à d'autres opérations, on effectue une synthèse préliminaire sans contrainte de performances sur le chemin de données.

Cette synthèse permet d'utiliser maintenant des cellules ayant une représentation physique, mais permet aussi de dupliquer sans risque les registres qui doivent être bufférisés. En effet, l'ajout des registres *buffers* a été reporté dans le graphe RN_{STD} et par conséquent dans le contrôleur asynchrone global. Il faut cependant insérer physiquement les registres dans le chemin de donnée. Afin de s'assurer d'être le plus transparent possible vis-à-vis du chemin de donnée, il faut que les *flip-flops* ajoutées soient de même type que celles du registre bufférisé. Cependant, cette information n'est accessible qu'après une première synthèse du chemin de données.

Nous appliquons donc la procédure suivante pour chaque registre qui a été bufférisé dans le RN_{STD} :

- On récupère la liste des bascules du registre à bufférisér et leur type de cellule standard.
- Pour chaque bascule de la liste :
  - La cellule standard associée est dupliquée.
  - Les ports de la cellule insérée sont connectés de la même manière que la cellule copiée sauf pour les entrées et sorties de données.
  - Le fil connecté à la *pin* de sortie de données de la cellule de base est déconnecté
  - Ce même fil est reconnecté à la *pin* de sortie de la cellule ajoutée
  - La pin de sortie des données de la bascule à dupliquer est connectée directement à la pin d'entrée de données de la cellule ajoutée

En reprenant le RN exemple de la Figure 4.3, l'application de la *bufférisation*sur le circuit serait comme sur la Figure 4.6.

Dans cette représentation, on remarque que comparé aux autres registres, les bascules de R_1 sont associés à un signal de *reset*. Cette connexion doit donc bien être reportée sur les *flip-flops* du registre $(R_1)_{buff}$. Il est à noter qu'aucun nuage combinatoire n'est représenté, et pour cause, les connexion entre les bascules de R_1 et celles de $(R_1)_{buff}$ sont de simples fils.

Déconnexion de l'arbre d'hologe et connexion du contrôleur asynchrone

Le circuit est maintenant modifié mais pour l'instant toujours synchrone. En utilisant la commande *'disconnect_net'* sur chaque *pin* d'horloge des registres présents dans le RN_{STD} on déconnecte l'arbre d'horloge du circuit.

Nous pouvons maintenant instancier le contrôleur asynchrone précédemment généré. Chaque signal de contrôle doit être connecté au registre qui lui est associé.

Les interfaces depuis et vers l'extérieur du circuit (entrées et sorties) doivent aussi être reportées sur l'interface du module désynchronisé.

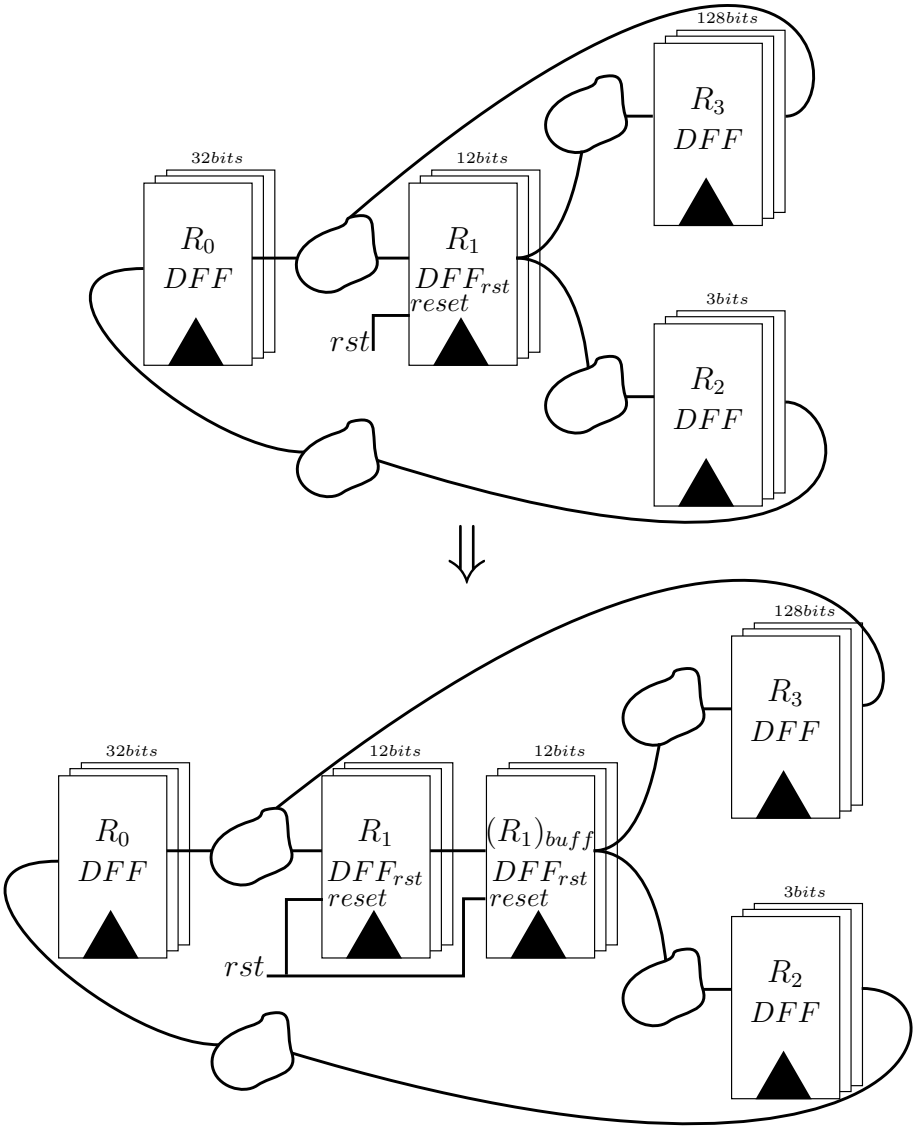


Figure 4.6: Exemple de *bufférisation*d'un registre en se basant sur le RN de la Figure 4.3

Adaptation des contraintes synchrones au circuit désynchronisé et synthèse finale

Nous allons partir de la supposition que la désynchronisation ne porte que sur un seul domaine d'horloge synchrone, l'adaptation pour de multiples domaines d'horloges suivrait quoi qu'il en soit la même logique.

Pour pouvoir comparer les résultats par la suite, nous sommes partis de l'idée que les chemins de données synchrones et désynchronisés doivent avoir la même contrainte. Ce choix permet de facilement faire correspondre les contraintes synchrones au circuit désynchronisé.

Dans un circuit synchrone, la contrainte de performance est déterminée par la période d'horloge. Cette contrainte est appliquée via une horloge définie avec la période visée, nommée P_{clk}.

Pour le circuit désynchronisé, il suffit alors de créer une horloge pour chaque signal contrôlant des registres en leur donnant une période de valeur P_{clk}. Enfin, il faut spécifier qu'entre

chaque domaine horloge, dans notre cas chaque contrôleur de registre, la donnée est contrainte par cette même période P_{clk}.

Une fois le circuit totalement désynchronisé et contraint, une dernière phase de synthèse permet de synthétiser le contrôleur asynchrone, qui n'était pour jusqu'alors que représenté sous une forme abstraite propre à l'outil, et d'assurer les contraintes de *timing* placées avec la création des horloges. L'outil génère également un certain nombre de contraintes qui seront utilisées dans les étapes suivantes du flot : le placement-routage.

4.2 Méthode des *Concurently Activated Registers* (CAR)

Le réseau de registres représente les dépendances entre les différents registres. Nous avons également vu dans la section précédente qu'il permet de déduire, en prenant en compte certaines règles, le contrôleur asynchrone global d'un circuit.

Les circuits synchrones n'étant généralement pas conçus pour être désynchronisés, leur architecture n'est pas toujours linéaire. Or il s'avère que la procédure de désynchronisation peut être très sensible à cette non-linéarité du pipeline.

4.2.1 Estimation de la linéarité d'un circuit

Afin de d'estimer comment la désynchronisation va impacter un circuit, en termes de surface et de complexité, nous avons créé un coefficient de linéarité, C_{lin}, qui est un indicateur du nombre moyen d'interconnexions entre les registres.

La formule déterminant le coefficient C_{lin} est celle de l'Équation (4.2), avec n représentant le nombre de registres dans le circuit.

$$C_{lin} = \frac{\sum^{\forall (i,j) \in ([0,n[)^2} M_{RN}[i,j] + 1}{n} \tag{4.2}$$

Le membre '$+1$' de la formule est seulement ajouté pour obtenir un C_{lin} de 1 dans le cas d'un pipeline totalement linéaire, comme par exemple dans les RN de la Figure 4.7.

Le cas opposé, sur la Figure 4.8, où chaque registre est connecté avec tous les registres y compris lui même, donne un $C_{lin} = n + 1/n$, soit environ n pour un circuit assez grand.

Au vu des règles édictées dans le Chapitre 3, plus une architecture sera linéaire moins elle sera sujette à la présence de boucles et, par conséquent, moins elle nécessitera d'ajout de matériel (contrôleur *bubble* et bufférisation).

Nous verrons dans le Chapitre 6 comment l'architecture de départ du circuit synchrone influence le résultat de la désynchronisation de manière plus concrète. Dans un premier temps,

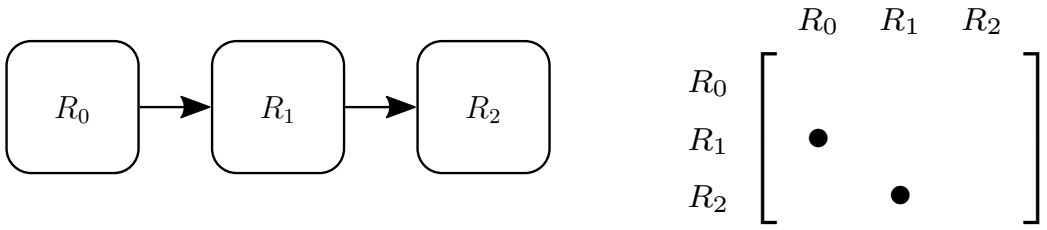


Figure 4.7: Réseaux de registres sous forme de graphe et matricielle d'un pipeline totalement linéaire.

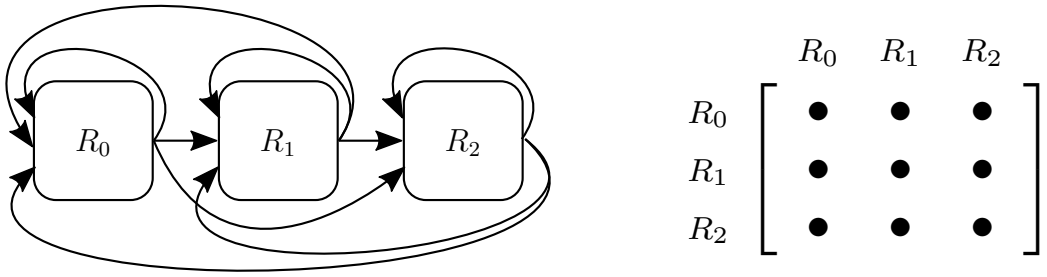


Figure 4.8: Réseaux de registres sous forme de graphe et matricielle d'un pipeline complètement connecté.

nous pouvons considérer qu'une architecture linéaire est préférable pour la désynchronisation sans utilisation de structures de choix.

Il est d'ailleurs à noter que ce coefficient n'a de sens que pour des circuits désynchronisés sans structure de choix.

Afin de simplifier le contrôleur asynchrone global et améliorer l'efficacité de la désynchronisation, nous chercherons à faire tendre ce coefficient de linéarité, C_{lin} vers 1.

Note sur le groupement de registres

La dynamique d'exdcution dans les contrôleurs asynchrones nous permettent d'opérer les modifications dans le RN que l'on souhaite tant que les dépendances existantes sont toujours retranscrites.

En effet, la propagation de requêtes dans le contrôleur est telle que, quelques soient les dépendances ajoutées, le fonctionnement initial est assuré. À chaque fois qu'un registre est activé, alors la donnée produite doit être consommée par tous les registres de destination avant de pouvoir libérer le registre source et donc permettre son actualisation.

Cela implique que n'importe quel regroupement de registres peut être effectué pourvu qu'il ne supprime pas de dépendances de données initialement détectées. Dans la suite de ce manuscrit, nous allons présenter différentes méthodes permettant de choisir quels registres regrouper afin de simplifier le réseau de registres et par conséquent réduire la complexité du contrôleur asynchrone du circuit désynchronisé.

4.2.2 Principe d'activation simultanée

Dans les pipelines non-linéaires sans structures de choix, des mécanismes de synchronisation, ou transitions, sont utilisés pour garantir l'équivalence de flot. Ces dernier peuvent créer des phénomènes d'attente qui font que certains registres seront dans tous les cas sollicités, ou activés, au même instant.

Considérons par exemple le fragment de RN de la Figure 4.9. Dans cette représentation partielle, nous indiquons l'existence d'autres arcs, avec des extrémités en pointillés. Faisons seulement apparaître ceux pertinents pour notre fragment.

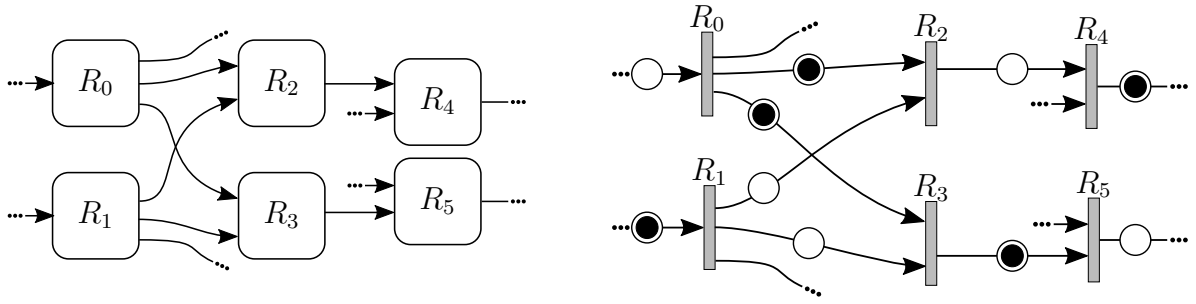


Figure 4.9: Fragment de graphe des réseaux de registres et de réseaux asynchrones où des registre ont des sources communes.

Le premier aspect que l'on peut remarquer sur le réseau de registres est que R_2 et R_3 sont tous deux dépendants de R_0 et R_1. Intéressons-nous à la propagation de données au travers des transitions associées aux registres R_2 et R_3. En suivant la règle d'activation stricte régissant les ANs, il faut remplir les conditions :

— R_0 a été activé, donc des *tokens* se trouvent dans chacun des canaux en aval de sa transition
— R_1 a été activé, donc des *tokens* se trouvent dans chacun des canaux en aval de sa transition
— R_2 est libre, donc la place suivant la transition associée est vide
— R_3 est libre, donc la place suivant la transition associée est vide

Nous avons donc l'activation de R_2 qui est contrainte par la validité des données en entrées de R_3 et la vacuité de la place en sortie de R_3. De la même manière, le transit d'une donnée au travers de R_3 est contraint par R_2.

Du point de vue asynchrone, nous pouvons donc considérer que R_2 et R_3 sont un seul et même registre, noté $R_{2,3}$. En fusionnant ces deux registres au niveau du réseau de registres, on obtient alors les RN et AN sur la Figure 4.10.

Intéressons-nous maintenant à la forme matricielle, (voirFigure 4.11) du réseau de registre initial de la Figure 4.9.

Nous pouvons calculer le coefficient de linéarité du fragment : $C_{lin} = 7/6$. Il est à noter que les connexions potentielles vers d'autres registres du circuit global n'ont pas été prises en

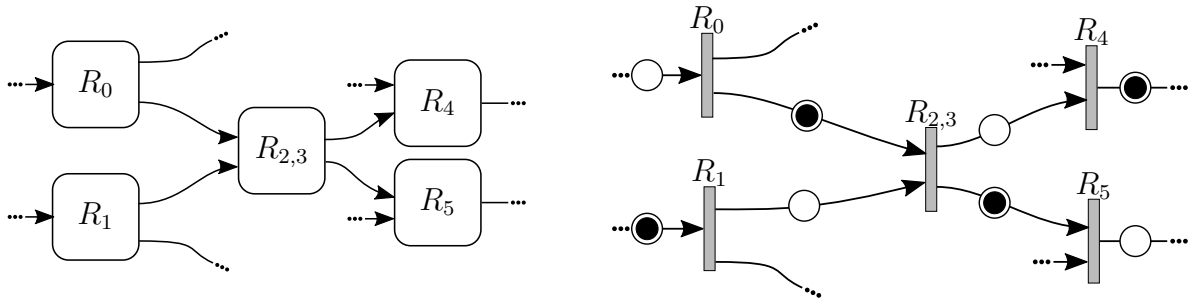


Figure 4.10: Fragment de graphe des réseaux de registres et de réseaux asynchrones d'une situation où des registres ont des sources communes.

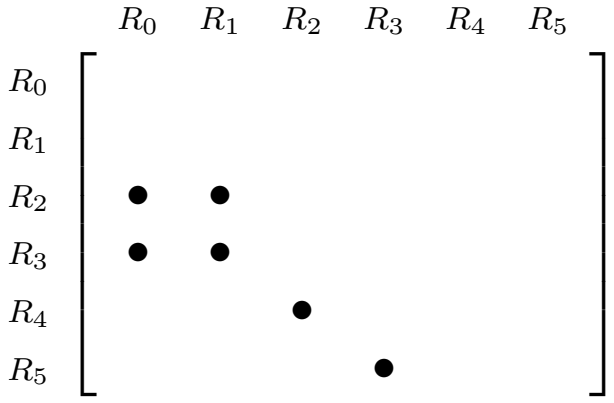


Figure 4.11: Réseau de registre sous forme matricielle associé à la Figure 4.9.

compte.

Nous pouvons remarquer que les lignes associées aux registres R_2 et R_3 sont identiques. En effet, la ligne de la matrice de RN représente la liste des registres sources du registre index.

Nous pouvons également faire le compte des différents éléments nécessaires pour construire le fragment de contrôleur associé :

— 6 contrôleurs de registres, initialisés pleins : $(2 * 17) * 6 = 204\ u.a.$

— 2 *forks* 1 vers 2 : $((2 - 1) * 14) * 2 = 28\ u.a.$

— 2 *joins* 2 vers 1 : $((2 - 1) * 14) * 2 = 28\ u.a.$

Le contrôleur de ce fragment représenterait donc une surface équivalente de $260\ u.a.$

Dans la Figure 4.12, la forme matricielle du réseau de registres a été déduite de la forme de graphe de la Figure 4.10.

Le coefficient de linéarité est maintenant $C_{lin} = 1$. L'opération de groupage nous a finalement bien permis de nous approcher d'une forme plus linéaire du pipeline tout en conservant une dynamique d'exécution identique.

De la même manière que précédemment, le décompte des structures nécessaires pour le contrôleur du fragment est :

— 5 contrôleurs de registres, initialisés pleins : $(2 * 17) * 5 = 170\ u.a.$

— 1 *forks* 1 vers 2 : $((2 - 1) * 14) * 1 = 14\ u.a.$

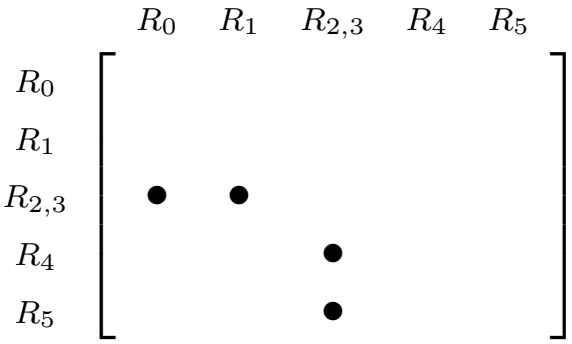


Figure 4.12: Réseau de registre sous forme matricielle associé à la Figure 4.10.

Le contrôleur asynchrone aurait donc une surface équivalente de $184\ u.a.$ soit une réduction de près de 30% par rapport au contrôleur initialement estimé.

Le groupage effectué a été nommé groupage des registres simultanément activés, ou CAR pour *Concurrently Activated Registers*.

4.2.3 Types de *Concurrently Activated Registers* et impacts sur le réseau de registres

Dans la section précédente, nous avons observé le comportement de registres simultanément activés ayant le même ensemble de registres sources. Nous appelons ces registres UCAR pour *Upstream* CAR. La situation symétrique, des registres ayant le même ensemble de registres de destination, peut aussi se produire est appelé DCAR pour *Downstream* CAR.

Nous avons pu nous rendre compte sur un exemple simple que ce regroupement permet de s'approcher d'une structure de *pipeline* linéaire et par la même occasion d'économiser de la surface au niveau du contrôleur.

Certains désavantages peuvent néanmoins se manifester en groupant des registres :

— La création de dépendances de données : cet effet se présente dans l'exemple que nous avons étudié auparavant. En effet, la dépendance du registre R_5 envers le registre R_2 n'existait pas dans la situation initiale. De fait, le chemin d'acquittement étant commun, le flot de donnée à travers de R_5 est maintenant dépendant de la propagation des données dans le réseau après R_4, et vice-versa. Cela n'est pas critique vis-à-vis du flot de donnée puisque les dépendances précédemment existantes, et donc fonctionnelles, sont conservées, mais cela peut créer des mécanismes d'attente qui n'auraient pas existé sans le groupage.

— La mise en commun des chemins critiques : Si on se base sur le fragment de la Figure 4.9, la mise en commun de R_2 et R_3 implique que la contrainte concernant les chemins vers R_2 était séparée de celle vers R_3. Une fois le regroupement effectué, la contrainte étant commune, il faut maintenant assurer la contrainte vis-à-vis du plus long chemin arrivant

à R_2 ou R_3. D'un côté, cela peut potentiellement ralentir le circuit, mais d'un autre, cela économise de la surface en insertion de délais et par conséquent de l'activité dans le contrôleur.

Il est à noter que ces effets néfastes se cantonnent à des problématiques de performances du circuit désynchronisé, comme bien souvent la conception d'un circuit, il est souvent affaire de compromis.

Il existe néanmoins des situations où l'effet de création de dépendances de données ne se produit pas : lorsque des registres sont à la fois DCAR et UCAR, donc avec le même ensemble de sources et de destinations dans le RN, ou les mêmes lignes et mêmes colonnes dans sa forme matricielle. Dans ce cas, les registres sont dénommés CCAR pour *Completely Concurrently Activated Registers*.

Grouper des registres en suivant la logique des registres simultanément activés est ce que l'on appelle dans nos travaux la méthode CAR. Les travaux initiaux décrivant cette méthode ont été présentés dans le cadre de cette thèse dans l'article [5].

4.2.4 Détection, force de regroupement et regroupement des registres

La représentation matricielle permet une formalisation, et par conséquent une automatisation aisée, des règles de détection de registres activés simultanément. Les trois types de CAR sont décrits dans les Définitions 2 à 4.

Définition 2 (Registres UCAR)**.** *Deux registres sont considérés comme UCAR s'ils partagent le même ensemble de registres de sources. Soit R_i les registres indexés par $i \in [0, n[$ au RN composé de n registres distincts. Soient deux registres R_k et R_l appartenant au même RN :*

$$\forall i \in [0, n[, M_{RN}[R_i, R_k] = M_{RN}[R_i, R_l] \Rightarrow \exists R_{k,l} = UCAR(R_k, R_l)$$

Définition 3 (Registres DCAR)**.** *Deux registres sont considérés comme DCAR s'ils partagent le même ensemble de registres de destination. Soit R_i les registres indexés par $i \in [0, n[$ au RN composé de n registres distincts. Soient deux registres R_k et R_l appartenant au même RN :*

$$\forall i \in [0, n[, M_{RN}[R_k, R_i] = M_{RN}[R_l, R_i] \Rightarrow \exists R_{k,l} = DCAR(R_k, R_l)$$

Définition 4 (Registres CCAR)**.** *Deux registres sont considérés comme DCAR s'ils partagent le même ensemble de registres source et destination. Soient deux registres R_k et R_l appartenant*

au RN composé de n registres distincts :

$$\left\{ \begin{array}{l} \exists UCAR(R_k, R_l) \\ \exists DCAR(R_k, R_l) \end{array} \right\} \Rightarrow \exists R_{k,l} = CCAR(R_k, R_l)$$

Les circuits synchrones sont en général conçus de telle façon qu'il est fréquent de rencontrer des configurations de *self-loops* dans le RN. Celles-ci peuvent rendre des registres similaires dans le RN incompatibles avec les définitions précédemment édictées. Malgré tout, ces registres bouclés sur eux-mêmes peuvent être intéressant à regrouper. Afin d'assurer la règle de vivacité (Règle 1) il faut ajouter un contrôleur WCHB *bubble* par *self loop* dans le contrôleur asynchrone ce qui peut avoir un impact significatif sur sa surface.

Dans nos travaux, nous avons exploré 2 niveaux de « force » de regroupement de registres qui se différencient suivant la prise en compte des différents index croisés dans la représentation matricielle du RN. Les deux niveaux définis sont :

– *soft* : La définition stricte de détermination des CAR est respectée.

–*greedy* : En considérant deux registres R_k et R_l, alors la définition est respectée pour tous les indices $i \in [0, n[\backslash k, l$. Les index croisés $\{k, l\}$, $\{l, k\}$, $\{l, l\}$ et $\{k, k\}$ sont ignorés pour la correspondance.

Avec ces « forces » de regroupements, on obtient donc 6 types de groupage de registres :

— *Upstream Concurrently Activated Registers Soft : UCAR_soft*
— *Upstream Concurrently Activated Registers Greedy : UCAR_greedy*
— *Downstram Concurrently Activated Registers Soft : DCAR_soft*
— *Downstream Concurrently Activated Registers Greedy : DCAR_greedy*
— *Completely Concurrently Activated Registers Soft : CCAR_soft*
— *Completely Concurrently Activated Registers Greedy : CCAR_greedy*

Une fois les différents regroupements à effectuer identifiés, il faut les reporter sur le réseau de registres. La Règle 3 présente la manière dont le groupage doit être effectué.

Règle 3 (Groupage de registres dans la matrice RN)**.** *Soient R_k et R_l deux registres d'un RN à regrouper. Un nouveau registre $R_{k,l}$ est ajouté au RN maintenant de taille $n + 1$. Avec $\vee$ représentant l'opérateur « ou », les valeurs de ses indices dans la matrice sont définies de la manière suivante :*

$$\forall i \in [0, n[\backslash \{k, l\} \quad \left\{ \begin{array}{l} M_{RN}[R_{kl}, R_i] = M_{RN}[R_k, R_i] \vee M_{RN}[R_l, R_i] \\ M_{RN}[R_i, R_{kl}] = M_{RN}[R_i, R_k] \vee M_{RN}[R_i, R_l] \end{array} \right\}$$

$$\begin{aligned} M_{RN}[R_{kl}, R_{kl}] &= M_{RN}[R_l, \quad R_k] \\ &\vee \quad M_{RN}[R_l, \quad R_l] \\ &\vee \quad M_{RN}[R_k, \quad R_k] \\ &\vee \quad M_{RN}[R_k, \quad R_l] \end{aligned}$$

Après la création et le remplissage de la ligne et de la colonne indexées par $R_{k,l}$, les lignes et les colonnes indéxées par R_k et R_l sont supprimées de M_{RN}, ce qui donne un nouveau RN à $n-1$ élements

4.2.5 Utilisation de la méthode CAR dans le flot de synthèse

La méthode CAR agissant sur la forme matricielle du RN, elle s'effectue dès lors que celle-ci est disponible. Sur la Figure 4.13 sont représentées les étapes ajoutées par rapport à celles existantes dans le flot de synthèse.

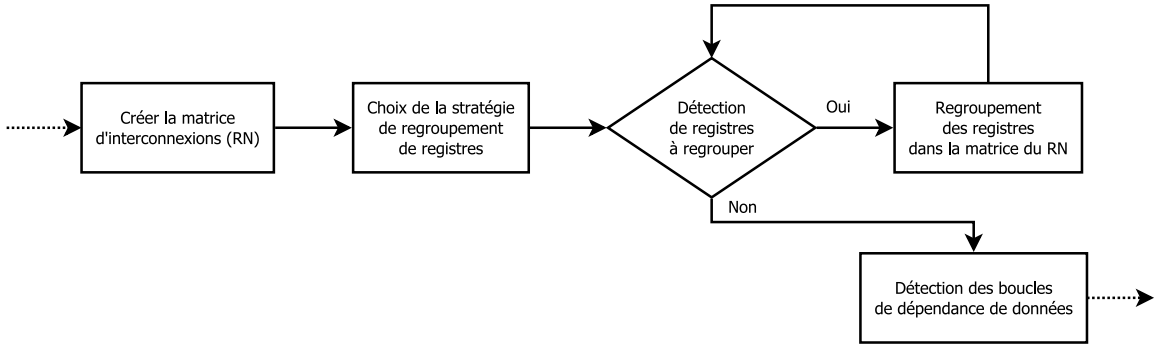


Figure 4.13: Étapes ajoutées dans le flot de synthèse pour l'utilisation du groupage de registres

Choix de la stratégie de regroupement de registres Dans nos travaux, nous avons choisi de n'appliquer qu'un seul type de simplification CAR, soit l'une des 6 stratégies suivantes :

- *UCAR_soft*
- *UCAR_greedy*
- *DCAR_soft*
- *DCAR_greedy*
- *CCAR_soft*
- *CCAR_greedy*

Il est à noter qu'il est envisageable d'utiliser une méthode mixte utilisant toutes les possibilités de simplifications. Elle consiste en l'étude complète du circuit afin de trouver toutes les simplifications possibles pour un RN puis de faire le choix de quelle simplification effectuer en premier lieu, par exemple la simplification réduisant le plus le coefficient C_{lin}. Cependant, les essais d'application de cette méthode se sont trouvés trop gourmands en temps de calcul

pour pouvoir être exploitées. De plus les résultats de l'application d'une stratégie unique se sont avérés satisfaisant.

Détection et regroupement des registres

Une fois la stratégie choisie, on étudie le réseau de registres afin de déterminer la présence ou non de registres à regrouper.

Si des registres ont été déterminés comme CAR, alors on effectue le regroupement dans la matrice et on cherche à nouveau la présence de registres à regrouper.

Lorsqu'il n'y a plus de registres à regrouper suivant la stratégie choisie, alors le flot reprend son cours comme nous l'avons décrit précédemment.

Ce chapitre présente une méthode automatique permettant de désynchroniser des circuits synchrones à partir de spécification synchrone en se basant sur la représentation par réseaux de registres. En prenant en compte les dépendances entre les données, nous pouvons en effet déduire un contrôleur asynchrone permettant de remplacer l'arbre d'horloge tout en conservant une exécution équivalente.

Certaines dépendances de données peuvent cependant être redondantes les unes avec les autres, c'est pourquoi, nous proposons également une méthode d'identification de registres pertinent à regrouper afin de simplifier le contrôleur asynchrone global.

Pour assurer que ce dernier fonctionne comme il le doit, nous allons maintenant passer à la vérification des contraintes temporelles dans les contrôleurs.

Chapitre 5

Analyse de *timing* dans les circuits désynchronisés

Nous avons vu au travers des chapitres précédents comment modéliser les systèmes à désynchroniser et les différentes façons de les désynchroniser.

Le protocole utilisé, 4-phases *bundled-data*, retranscrit les transferts de données de contrôleur à contrôleur, ou de registre à registre, mais ne traduit pas naturellement la propagation « réelle » des données dans le chemin combinatoire.

Ainsi, afin d'assurer le bon fonctionnement des circuits désynchronisés, des vérifications de *timing* doivent être effectuées. L'ensemble de ces vérifications est appelé STA pour *Static Timing Analysis*.

Dans ce chapitre, nous allons tout d'abord nous intéresser aux différentes contraintes de *timing* dans les circuits désynchronisés.

Ensuite, nous porterons notre attention sur comment l'adaptation des outils, conçus initialement pour la vérification des circuits synchrones, pour réaliser une STA automatique sur des circuits désynchronisés.

Enfin, nous nous pencherons sur différentes méthodes envisagées pour résoudre les potentielles violations de *timing*.

5.1 Contraintes de *timing* dans les circuits désynchronisés

En conception synchrone, la STA permet de vérifier automatiquement un large panel de violations de *timing*. Les outils standards n'étant pas conçus pour faire de même sur des circuits désynchronisés, un travail est à fournir pour adapter les contraintes synchrones à un circuit désynchronisé.

Afin de comprendre comment interpréter ces contraintes, nous allons nous intéresser à chacunes d'elles, d'abord dans le cas synchrone, puis dans le contexte de circuits désynchronisés.

5.1.1 Contrainte de *setup*

La contrainte de *setup* cherche à assurer qu'entre deux registres, la donnée obtenue après un front d'horloge en sortie de la source ait suffisamment de temps pour s'établir à l'entrée de la destination avant le prochain front montant d'horloge.

Dans la Figure 5.1 est représenté le chemin entre deux registres R_{source} et R_{dest}. Les différents *timing* mis en jeu sont décomposés.

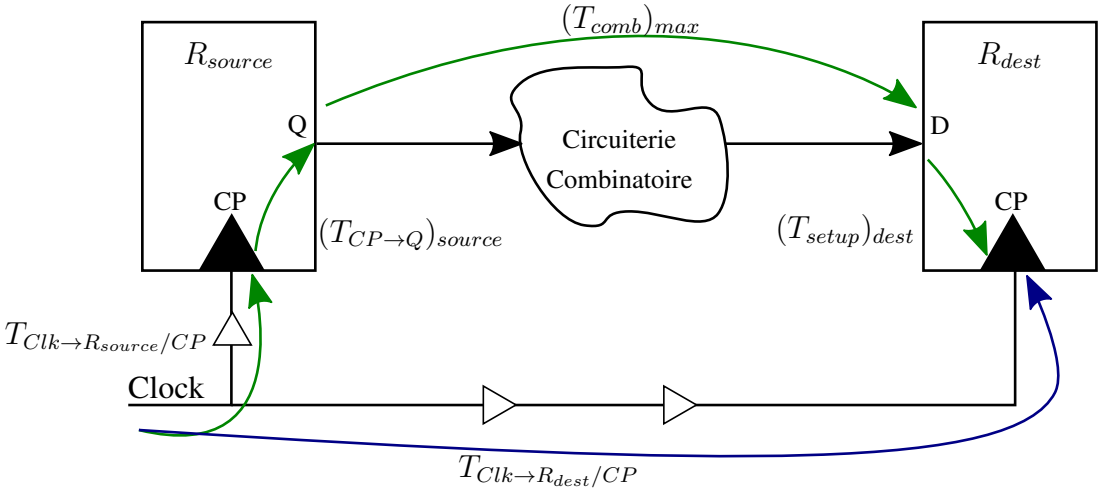


Figure 5.1: Décomposition de la contrainte de *setup* dans un circuit synchrone

La Figure 5.2 représente ces différents *timings* dans un chronogramme.

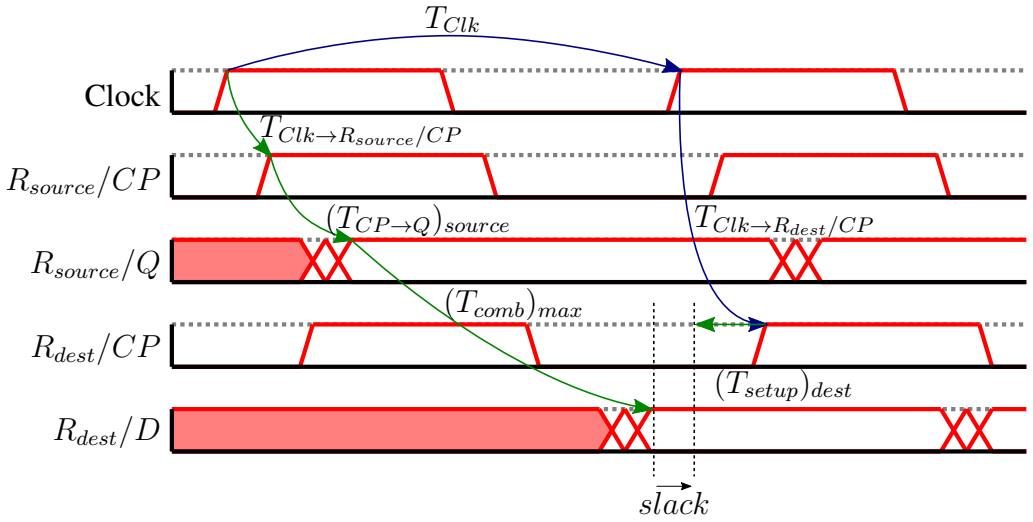


Figure 5.2: Chronogramme du transfert de données entre deux registres R_{source} et R_{dest} dans un circuit synchrone

Les différents *timings* présentés dans ces schémas sont :

— $T_{Clk\to R_{source}/CP}$ ($T_{Clk\to R_{dest}/C}$) est le temps que met le signal d'horloge depuis sa source jusqu'à l'entrée d'horloge du registre R_{source} (respectivement R_{dest}).

— $(T_{CP\to Q})_{source}$ représente le temps de l'établissement de la sortie après l'arrivée d'un front montant sur la *pin* d'horloge du registre source.

— $(T_{comb})_{max}$ est le plus grand temps de propagation de la donnée dans le nuage combinatoire.
— $(T_{setup})_{dest}$ est le temps avant le prochain front d'horloge pendant lequel la donnée doit être stable.

La grandeur appelée $slack$ dans le chronogramme n'est pas un temps réel, mais le budget de temps disponible pour respecter la contrainte de *setup*. Lorsque le $slack$, aussi appelé marge, est positif, alors la contrainte de setup est respectée.

En utilisant les notations de ce dernier schéma, avec la période d'horloge T_{clk}, alors les différents timings peuvent se mettre sous la forme de l'Équation (5.1).

$$\begin{aligned} &T_{clk} + T_{Clk \rightarrow R_{dest}/CP} = \\ &T_{Clk \rightarrow R_{source}/CP} + (T_{CP \rightarrow Q})_{source} + (T_{comb})_{max} + (T_{setup})_{dest} + slack \end{aligned} \tag{5.1}$$

Pour respecter la contrainte de *setup*, il faut que la marge soit positive pour tous les chemins entre registres partageant la même horloge. La contrainte globale peut être exprimée comme dans l'Équation (5.2).

$$slack > 0 \Rightarrow T_{clk} > max_{circuit} \left(\begin{array}{l} T_{Clk \rightarrow CP_{source}} - T_{Clk \rightarrow CP_{dest}} \\ +(T_{CP \rightarrow Q})_{source} \\ +T_{comb} \\ +(T_{setup})_{dest} \end{array} \right) \tag{5.2}$$

Dans un circuit synchrone, la marge de manœuvre est limitée pour assurer la contrainte de *setup*. En effet, il est difficilement imaginable de modifier fortement les *timings* $T_{Clk \rightarrow CP_{source}}$ et $T_{Clk \rightarrow CP_{dest}}$. Supposons que l'on augmente fortement $T_{Clk \rightarrow CP_{dest}}$, alors sur un autre chemin, où cette fois le registre de destination prend le rôle d'une source, la contrainte de *setup* serait dégradée. En général, un équilibrage est recherché entre ces deux valeurs, c'est d'ailleurs ce qui est fait par le mécanisme du *Clock Tree Synthesis* ou CTS.

Les facteurs $(T_{CP \rightarrow Q})_{source}$ et $(T_{setup})_{dest}$ dépendent principalement de la technologie utilisée. En cela, ils sont liés au couple résistance/capacité parasite des fils connectés et au temps de transition des signaux d'entrée. Ce ne sont cependant généralement pas des grandeurs qu'il est souhaitable d'utiliser pour résoudre des violations de *timing* puisque, comme dans les cas des temps de traversée d'arbres d'horloge, cela revient à déplacer le problème.

Il reste donc T_{clk} et T_{comb} sur lesquels il est possible d'influer. Concernant le temps de propagation à travers le chemin combinatoire, les outils sont capables de les optimiser jusqu'à un certain point avec néanmoins un coût en surface et en consommation qui peut être significatif. Il ne reste ensuite plus que la période d'horloge sur laquelle le concepteur à toute latitude dans la limite des spécifications du circuit.

Intéressons-nous maintenant à la contrainte de *setup* dans les circuits désynchronisés. La Figure 5.3 présente de la même manière que pour le cas synchrone, les différents timings mis en jeu en prenant en compte la spécificité des circuits désynchronisés : le contrôleur asynchrone.

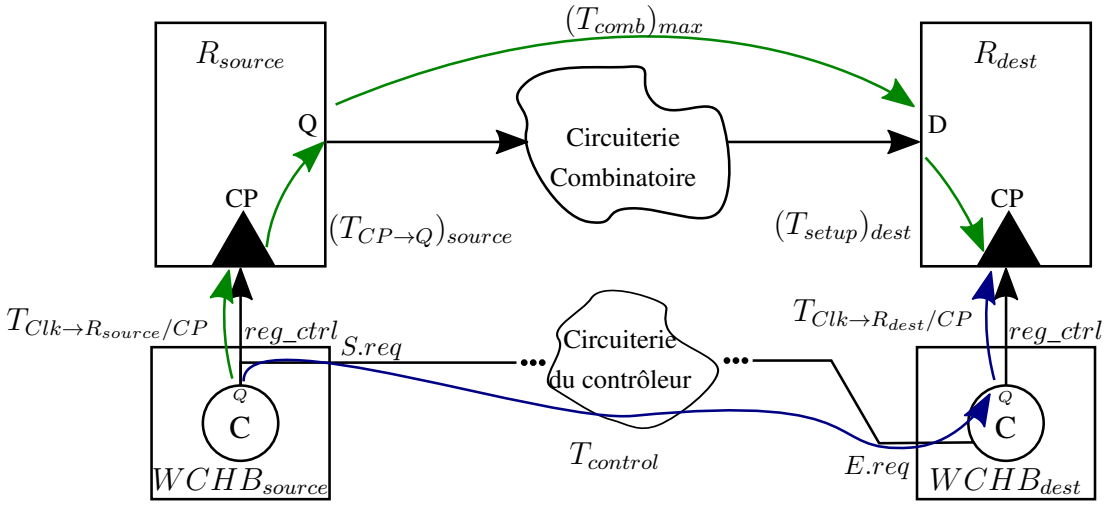


Figure 5.3: Décomposition de la contrainte de *setup* dans un circuit désynchronisé

Il est à noter qu'entre les deux contrôleurs de registres peut se trouver une certaine quantité de circuiterie due aux transitions et contrôleurs *bubble* traversés. Cependant, par construction pour un chemin de registres à registres, seulement un chemin dans le contrôleur existe. Par conséquent, la circuiterie entre les deux contrôleurs de capture est seulement considérée comme du délai faisant partie de $(T_{control})_{source\rightarrow dest}$.

Le chronogramme de la Figure 5.4 présente les *timings* mis en jeu pour cette fois la contrainte de *setup* asynchrone.

En prenant les notations présentées dans ces schémas, nous pouvons déduire l'Équation (5.3).

$$\begin{aligned}&(T_{control})_{source\rightarrow dest} + (T_{reg\_ctrl\rightarrow CP})_{dest} = \\ &(T_{reg\_ctrl\rightarrow CP})_{source} + (T_{CP\rightarrow Q})_{source} + (T_{comb})_{max} + (T_{setup})_{dest} + slack\end{aligned} \tag{5.3}$$

Afin d'assurer la contrainte de *setup*, on doit donc vérifier l'Équation (5.4).

$$slack > 0 \Rightarrow (T_{control})_{source\rightarrow dest} > max_{source\rightarrow dest} \begin{pmatrix} (T_{reg\_ctrl\rightarrow CP})_{source} \\ -(T_{reg\_ctrl\rightarrow CP})_{dest} \\ +(T_{CP\rightarrow Q})_{source} \\ +(T_{comb})_{max} \\ +(T_{setup})_{dest} \end{pmatrix} \tag{5.4}$$

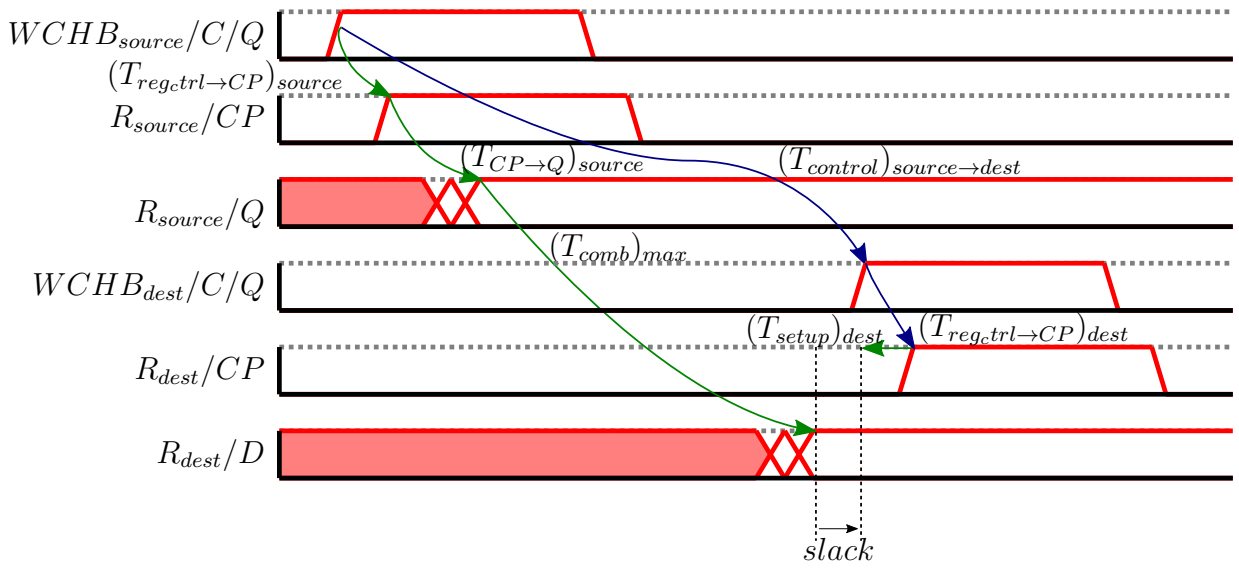


Figure 5.4: Chronogramme du transfert de données entre deux registres R_{source} et R_{dest} dans un circuit désynchronisé

L'équation est proche de son équivalent synchrone. La majeure différence est que le terme T_{clk} global dans la version synchrone est remplacé par $(T_{control})_{source \to dest}$ qui est local puisqu'il dépend de la source et de la destination considérées. En outre, le terme $(T_{comb})_{max}$ est lui aussi local.

En cas de violation de la contrainte de setup, il suffit donc d'ajouter localement du délai pour augmenter le *timing* de $(T_{control})_{source \to dest}$. Nous verrrons dans la Section 5.3 les méthodes explorées pour la génération de ces délais.

5.1.2 Contrainte de *hold*

L'entrée d'une bascule doit rester stable assez longtemps après le front d'horloge pour assurer une capture correcte. C'est ce que l'on appelle la contrainte de *hold*. Bien que les violations de *hold* soient généralement moins fréquentes que les violations de *setup*, elle peuvent être assez problématiques dans les circuits synchrones.

Les *timings* mis en jeu pour la contrainte de *hold* étant similaires à ceux utilisés pour le *setup*, nous allons considérer les mêmes notations que précédemment en remplaçant les *timings* $(T_{setup})_{dest}$ par $(T_{hold})_{dest}$ et $(T_{comb})_{max}$ par $(T_{comb})_{min}$.

Le chronogramme de la Figure 5.5 présente les différents timings mis en jeu.

Nous pouvons en déduire l'Équation (5.5).

$$\begin{aligned} &T_{Clk \to R_{dest}/CP} + (T_{hold})_{dest} + slack = \\ &T_{Clk \to R_{source}/CP} + (T_{CP \to Q})_{source} + (T_{comb})_{min} \end{aligned} \tag{5.5}$$

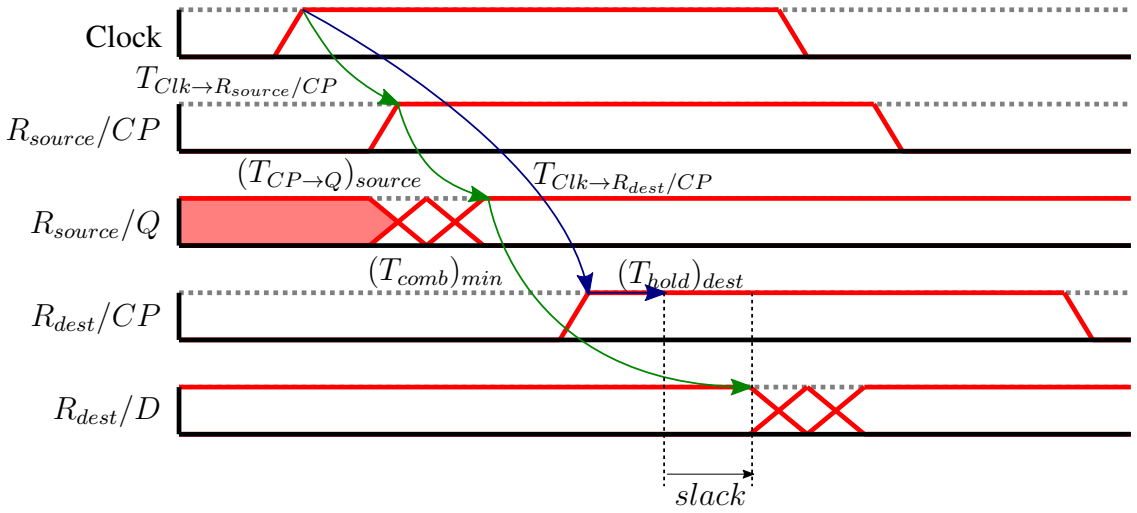


Figure 5.5: Chronogramme de *hold* entre deux registres R_{source} et R_{dest} dans un circuit synchrone

Afin de vérifier la contrainte de *hold*, il faut assurer comme dans le cas du *setup* que le $slack$ soit positif. Dans le cas du *hold*, cela se traduit par la vérification de l'Équation (5.6).

$$slack > 0 \Rightarrow (T_{hold})_{dest} < min \left(\begin{array}{l} T_{Clk \rightarrow R_{source}/CP} - T_{Clk \rightarrow R_{dest}/CP} \\ +(T_{CP \rightarrow Q})_{source} \\ +(T_{comb})_{min} \end{array} \right) \tag{5.6}$$

Pour assurer le *hold*, il faut donc assurer un temps minimum. Cette condition est en général relativement simple à gérer pour les outils. En effet, étant donné que cela concerne les chemins « rapides », ils ne font pas partie des chemins critiques. Le temps T_{comb} peut donc être augmenté grâce à l'ajout de *buffer* ou cellules de délai afin de vérifier la contrainte de *hold*.

Pour ce qui est des circuits désynchronisés, nous devons étudier les chemins d'acquittement afin de déterminer si une violation de *hold* peut survenir. En effet, les signaux d'acquittement sont ceux qui vont « libérer » le contrôleur source et donc permettre une nouvelle capture de données qui pourrait causer des problèmes de *hold*.

La Figure 5.6 présente les *timings* mis en jeu dans l'étude de cette problématique.

En considérant le chronogramme de la Figure 5.7 nous pouvons déduire les Équations (5.7) et (5.8).

$$\begin{array}{l} (T_{reg\_ctrl \rightarrow CP})_{dest} + (T_{hold})_{dest} + slack = \\ (T_{release})_{dest \rightarrow source} + (T_{reg\_ctrl \rightarrow CP})_{source} + (T_{CP \rightarrow Q})_{source} + T_{comb} \end{array} \tag{5.7}$$

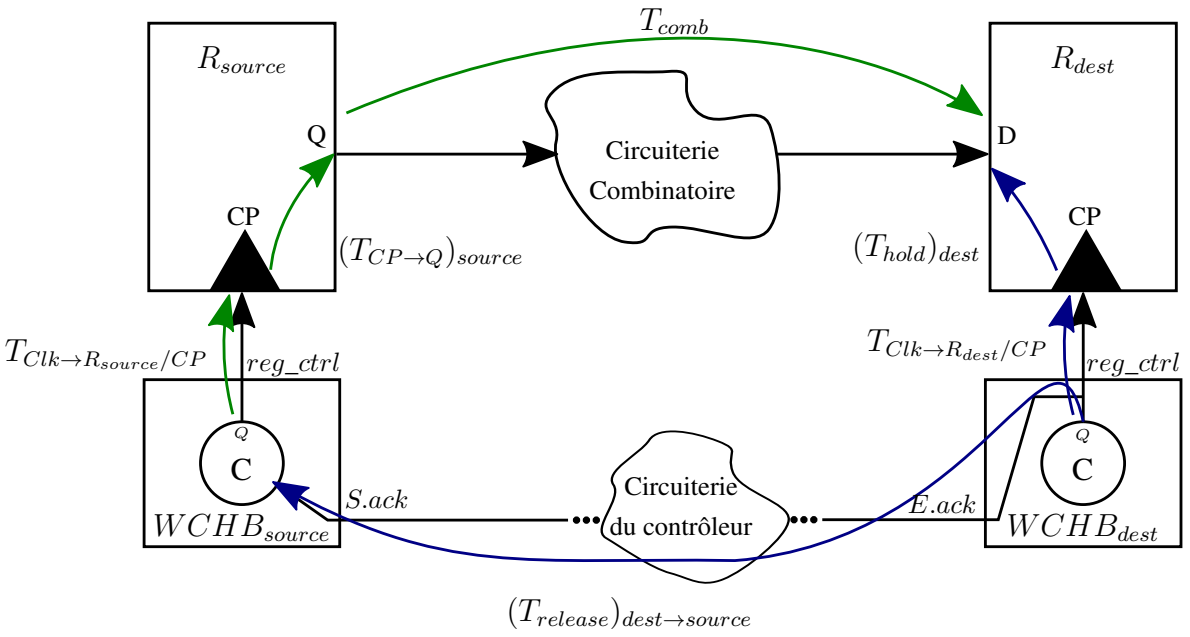


Figure 5.6: Décomposition de la contrainte de *hold* dans un circuit désynchronisé

$$slack > 0 \Rightarrow (T_{hold})_{dest} < min \left(\begin{array}{l} (T_{reg\_ctrl\rightarrow CP})_{source} - (T_{reg\_ctrl\rightarrow CP})_{dest} \\ +(T_{CP\rightarrow Q})_{source} \\ +(T_{release})_{dest\rightarrow source} + (T_{comb})_{min} \end{array} \right) \tag{5.8}$$

Il est à noter que, dans les circuits désynchronisés considérés, la contrainte de *hold* est très rarement problématique. En effet, le temps de traversée du contrôleur, au travers du chemin d'acquittement, suffit généralement à assurer cette contrainte.

5.1.3 Contrainte de *width*

Cette contrainte vise à vérifier que le signal d'horloge reste au niveau haut un certain temps avant de repasser au niveau bas, et vice-versa. On parle alors de largeur, ou *width*, du *pulse* d'horloge.

Dans les circuits synchrones, cette contrainte est assez simple à vérifier puisqu'à première vue, il suffit d'assurer que la période d'horloge et la forme du signal d'horloge soient corrects. Cependant, cette contrainte peut se trouver mise à mal dans le cas de grands arbres d'horloge dans lesquels on peut observer une différence importante entre temps de montée et temps de descente du signal. C'est l'une des raison pour lesquelles l'équilibrage du *clock-tree* est nécessaire.

Dans un circuit désynchronisé la forme d'onde des signaux de contrôle dépend du flot de requêtes dans le circuit de contrôle.

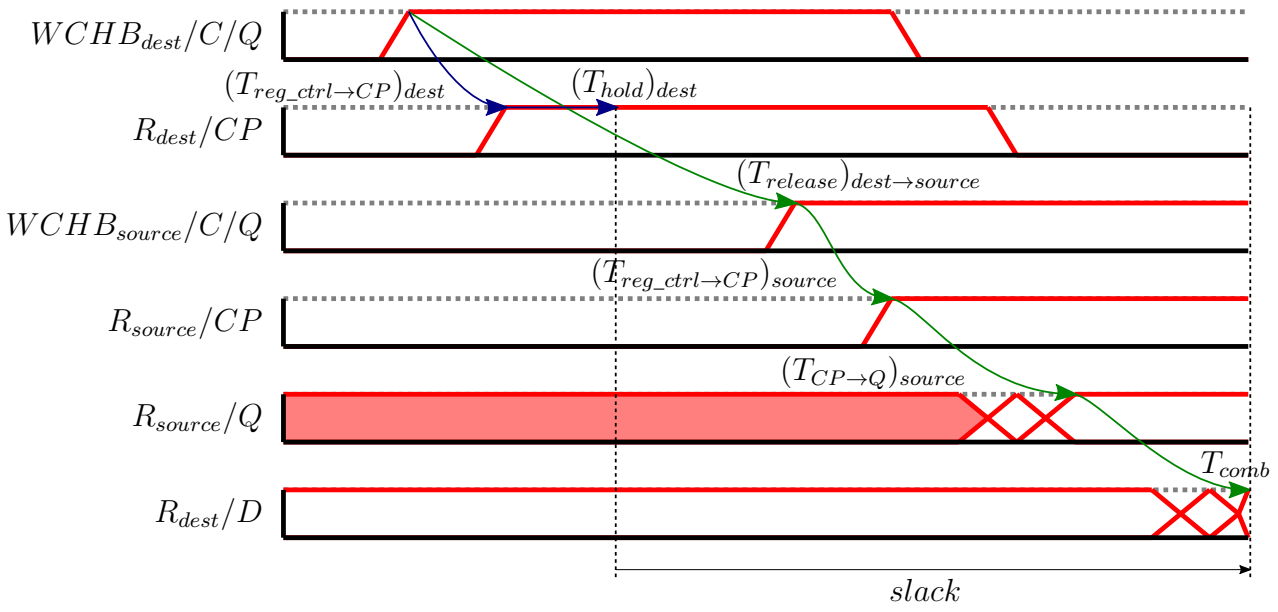


Figure 5.7: Chronogramme de *hold* entre deux registres R_{source} et R_{dest} dans un circuit désynchronisé

Afin d'effectuer les vérifications adéquates, nous considérons donc le pire cas possible. Sur la Figure 5.8 est représenté un fragment de contrôleur avec les arcs de *timing* pertinents pour la contrainte de *width*.

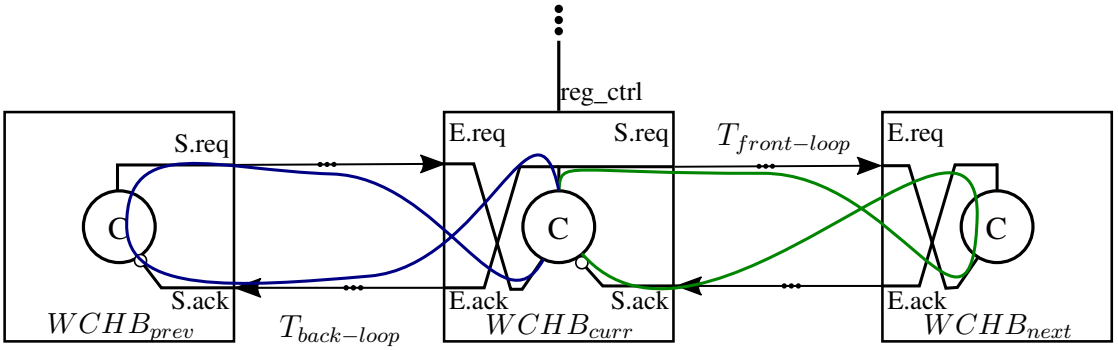


Figure 5.8: Chemins de boucles de requêtes et d'acquittement dans un circuit de contrôle

Supposons que $WCHB_{curr}/reg\_ctrl$ passe à un niveau haut, *i.e.* $WCHB_{curr}/E.req$ est à 1 et $WCHB_{curr}/S.ack$ est à 0. Pour qu'il repasse à un niveau bas, il faut remplir deux conditions :

— La requête d'entrée doit retomber, donc il faut que l'acquittement de l'entrée se propage jusqu'à $WCHB_{prev}$ et que la retombée de la requête se propage jusqu'à $WCHB_{curr}$: $T_{loop-back}$

— L'acquittement de sortie doit être au niveau haut, donc il faut que la requête de sortie de $WCHB_{curr}$ se propage jusqu'à $WCHB_{next}$, que la requête soit traitée et enfin que l'acquittement revienne au premier contrôleur : $T_{loop-front}$

Comme spécifié dans l'Équation (5.9), la contrainte de *width* est donc dépendante du maxi-

mum des deux temps de boucle.

$$T_{width} < max(T_{loop-back}, T_{loop-front}) \tag{5.9}$$

Il est à noter que le raisonnement a été effectué pour assurer une largeur minimale, T_{width}, à un niveau haut du signal de contrôle mais qu'il faut valider de façon similaire le niveau bas.

5.1.4 Contrainte de sélection

La dernière contrainte sur laquelle nous allons nous pencher est une contrainte spécifique aux transitions de sélection. Bien que celles-ci ne soient pas utilisées de manière automatique, afin de fournir le plus de possibilités au concepteur, nous allons nous intéresser aux contraintes à garantir pour ces structures.

Ces transitions utilisent des bits de données ainsi que la requête associée pour diriger les requêtes d'entrée ou de sortie. Pour assurer que le bon canal soit sélectionné, il faut vérifier que la donnée de sélection est prise en compte avant que la requête associée ne survienne. Cette contrainte peut s'apparenter à un contrainte de *setup* de sélection.

La Figure 5.9 présente les *timing* mis en jeu dans un fragment de sélection du canal de sortie (donc pour une transition *split*, *c.f.* Figure 3.31) pour cette contrainte de sélection.

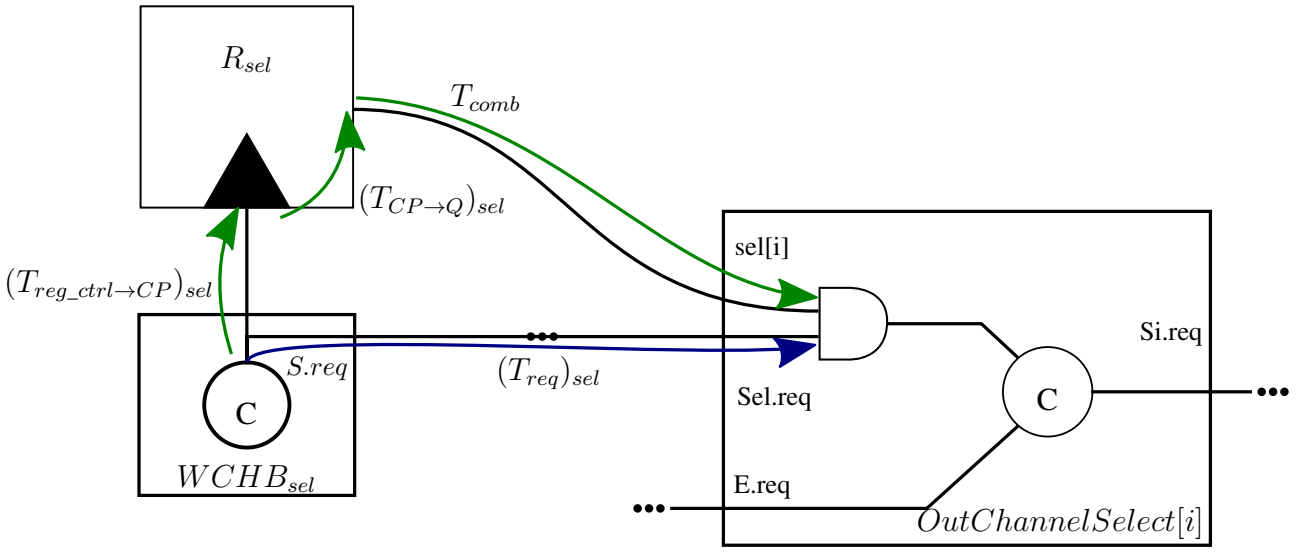


Figure 5.9: Chemins de la contrainte de sélection dans un élément $OutChannelSelect[i]$

Dans ce cas, il faut donc respecter l'Équation (5.10).

$$(T_{req})sel > (T_{reg\_ctrl\to CP})_{sel} + (T_{CP\to Q})_{sel} + T_{comb} \tag{5.10}$$

Il est à noter que, le point de rencontre entre la donnée et la requête de sélection étant le même, le raisonnement reste identique pour une transition *merge*.

5.2 Implémentation de la STA dans les circuits désynchronisés

Nous savons maintenant quelles sont les contraintes à vérifier dans les circuits désynchronisés.

La prochaine étape est de transformer ces différentes équations en contraintes que les outils pourront comprendre, vérifier et, idéalement, traiter automatiquement en apportant les corrections nécessaires.

5.2.1 Vérification des contraintes de *setup* et *hold*

Les circuits désynchronisés nécessitent une adaptation de l'utilisation des outils prévus pour le synchrone. En effet, la présence de nombreuses boucles combinatoires, dues au contrôleur asynchrone global ou aux boucles architecturales existantes dans le circuit, les amènent à « couper » les chemins aux endroits qui leurs semblent opportuns.

Néanmoins, ces points de coupure s'avèrent rarement pertinent pour le concepteur et l'interprétation des *timings* dans le circuit. Une façon de s'assurer que les outils cassent ces boucles combinatoires aux bons endroits est de choisir spécifiquement les points de coupure. Cependant, l'utilisation d'exceptions locales limite les optimisations pouvant être opérées en une seule passe par les outils. Par exemple, certains chemins utiles pour le *hold* ont besoin d'être coupés pour vérifier le *setup*. Ainsi, plusieurs passes sont généralement nécessaires pour que la caractérisation soit complète.

Une approche plus flexible a été publiée pendant la réalisation de cette thèse : les travaux de Grégoire Gimenez *et. al.* [21]. C'est cette méthode qui a finalement été retenue pour la STA de nos circuits désynchronisés.

Outre le fait que les travaux présentés dans cette publication apportent une méthode utilisant les outils de conception standard, les auteurs proposent une méthode utilisant ce qu'ils appellent des *Relative Timing Constraint*, ou RTC.

Les approches classiques utilisent en général des contraintes statiques basées sur les commandes *'set_min_delay/set_max_delay'*. Celles-ci demandent cependant de caractériser les conditions de fabrication, appelées *corner* PVT (*Process Voltage Temperature*), afin d'adapter les contraintes exercées.

L'approche proposée dans [21] modélise la propagation d'événement dans le contrôleur et ainsi s'applique de la même manière dans chaque *corner* PVT.

Pour illustrer comment celle-ci fonctionne, appliquons-la, pour vérifier la contrainte de *setup*, entre R_{source} et R_{dest} au fragment de circuit désynchronisé de la Figure 5.10.

Les étapes à suivre sont :

1. La création des différentes horloges :

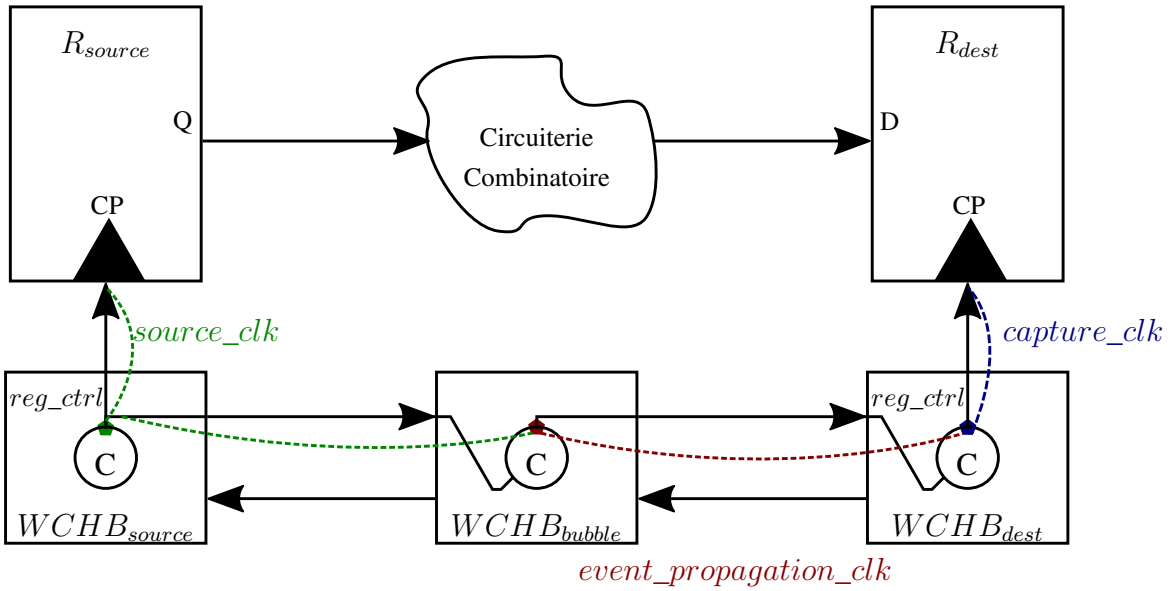


Figure 5.10: Fragment de circuit pour appliquer la méthode LCS

(a) Horloge de départ de l'événement sur la sortie du C-element de $WCHB_{source}$ (point vert) : $source\_clk$.

(b) Horloge de propagation de l'événement sur la sortie du C-element de $WCHB_{bubble}$ (point rouge) avec comme horloge de base $source\_clk$: $event\_propagation\_clk$.

(c) Horloge de capture de l'événement sur la sortie du C-element de $WCHB_{dest}$ (point bleu) avec comme horloge de base $event\_propagation\_clk$: $capture\_clk$.

2. La suppression des chemins non pertinents :
   (a) Les chemins de *hold* se terminant par $capture\_clk$. En effet, cette horloge est seulement utile pour le setup.
   (b) Les chemins de *setup* partant de $capture\_clk$, seuls les chemins avec comme départ $source\_clk$ ont un sens puisque c'est le point de départ de l'événement.
3. La transformation des horloges en événement : on applique la commande *'set_multicycle_path'* avec le multiplicateur 0 pour le *setup* depuis $source\_clk$ vers $capture\_clk$.
4. La propagation des horloges, qui sont dans notre cas des événements, dans le réseau de contrôle.

Avec cette dernière étape, le signal $WCHB_{dest}/reg\_ctrl$ est considéré comme issu de la source de $source\_clk$. Cela contraint ensemble les registres de R_{source} avec ceux de R_{dest} tout en prenant en compte le temps de propagation dans le réseau asynchrone, pour atteindre la destination, comme une propagation dans l'arbre d'horloge.

Pour appliquer les vérifications de *hold*, la méthode est similaire mais les événements sont cette fois propagés au travers des chemins d'acquittement.

5.2.2 Vérification de la contrainte de *width*

Bien qu'elle ne permette pas à notre connaissance de vérifier les contraintes de *width* d'horloge automatiquement, la méthode LCS pose un cadre propice pour mesurer les *timings* dans les contrôleurs asynchrones.

En effet, celle-ci définit une horloge sur chaque contrôleur de registre, de capture, mais également sur chaque contrôleur intermédiaire, WCHB *bubble*. Les sources d'horloges sont considérées par les outils comme des points d'arrêt pour le *timing*. Par conséquent, les points utilisés comme source pour les horloges sont des points de coupure, comme si une exception *'set_disable_timing'* avait été appliquée. La différence cependant est que cette dernière exception s'applique sur une cellule physique, entre une entrée et une sortie, et désactive totalement le lien entre les deux points. L'information de transition dans la cellule est donc ignorée. En outre, en définissant une source d'horloge, cette information n'est pas perdue.

Les horloges, utilisées pour la méthode LCS, permettent de casser toutes les boucles combinatoires pouvant exister dans le contrôleur, et offrent des points aisément identifiables pour mesurer les différents *timings*.

La Figure 5.11 présente la situation vue précédemment pour exprimer la contrainte de *width* en prenant en compte les différents points où les horloges de la méthode LCS sont définies.

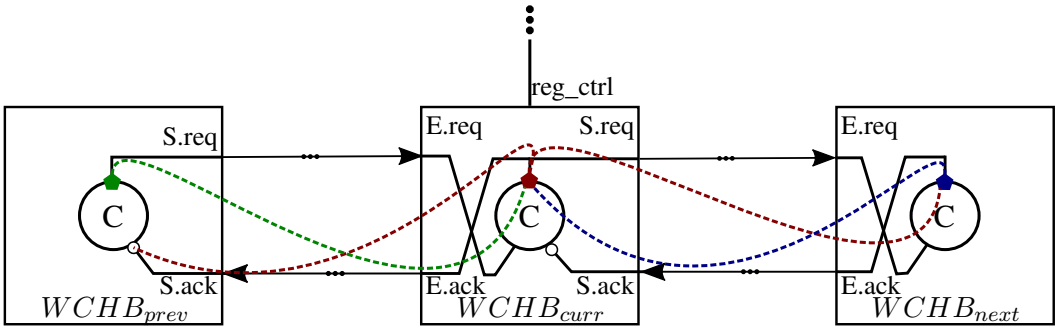


Figure 5.11: Fragment de circuit pour procéder aux mesures de *width* des signaux de contrôle

En utilisant la commande *'get_timing_paths'* entre chacun des chemins nous pouvons donc obtenir toutes les mesures pertinentes pour la contrainte de *width*, et les faire correspondre avec l'équation à respecter dans ce cas.

Ne se reposant pas sur des mécanismes de vérification internes à l'outil, la contrainte de *width*, si elle n'est pas vérifiée, ne sera pas reportée par l'outil comme une violation. Pour déterminer si la contrainte de *width* est vérifiée, le concepteur devra procéder aux comparaisons pertinentes.

5.2.3 Vérification de la contrainte de sélection

La vérification peut se faire de manière manuelle, comme pour la contrainte de *width*, c'est-à-dire effectuer les deux mesures pertinentes et les comparer.

Elle peut également se faire, en utilisant des exceptions point à point de type *'set_min_delay'*. Avec cette dernière solution, il faut tout d'abord mesurer le temps pris par la donnée pour arriver à la cellule procédant à la sélection, T_{data_sel}. Ensuite, il faut créer un point d'arrêt au niveau du point de rencontre entre la donnée de sélection et la requête associée en utilisant une exception *'set_disable_timing'*. Il est à noter que cette dernière exception « casse » également le chemin de requête. Cela implique que la vérification des contraintes de sélection doit se faire lors d'une passe spécifique.

Enfin, une contrainte utilisant la commande *'set_min_delay'*, avec la valeur T_{data_sel}, est placée entre la source de la requête et le point de rencontre entre la requête et la donnée.

Contrairement à la vérification manuelle utilisée pour la contrainte de *width*, cette solution utilise les mécanismes de vérifications interne des l'outils. Si une violation se présente, elle sera donc indiquée par l'outil en tant que violation de *timing*.

5.3 Méthodes de résolution des violations de *timing*

Nous avons à notre disposition les éléments nécessaires pour procéder à toutes les vérifications de *timing* permettant d'assurer un circuit désynchronisé fonctionnel.

Cependant, les contraintes telles que spécifiées, bien qu'interprétées correctement par les différents outils utilisés, ne sont pas toutes utilisables pour générer les délais nécessaires.

Si nous revenons sur la méthode LCS par exemple, celle-ci permet effectivement de vérifier automatiquement les *timings* pour le *setup* et le *hold* dans les circuits désynchronisés.

La piste la plus naturelle qui consisterait à remplir les différentes contraintes en l'ajoutant des délais sur les chemins pertinents n'est pas intégrée dans les algorithmes des outils.

Nous nous heurtons ici à la principale limitation des outils synchrones pour leur utilisation dans le cas de circuits désynchronisés car ils ne sont pas conçus pour générer des délais.

Diverses pistes ont été explorées pendant cette thèse pour faire synthétiser ces délais par les outils synchrones.

Nous allons voir dans la suite les différentes méthodes explorées et les pistes envisagées pour améliorer la *Quality of Result* ou QoR des circuits désynchronisés.

N.B. : *Sauf spécification contraire, l'outil utilisé dans cette section est IC-Compiler. Nous utiliserons donc certaines commandes spécifiques à cet outil. En particulier, nous utiliserons fréquemment le concept de scénario utilisé par l'outil pour avoir plusieurs conditions d'opération, ou configuration d'exceptions, dans une même instance de l'outil.*

Il est à noter que les contraintes des *hold* et *width* n'ayant jamais eu besoin d'opérations spécifiques dans nos travaux, les méthodes de génération de délai utilisées par la suite se sont concentrées sur les problématiques de *setup* et de sélection.

5.3.1 Méthode manuelle

En ayant une pleine connaissance des rapports de STA, il est tout à fait possible d'ajouter les cellules de délai « à la main ». Il est possible de les ajouter directement dans la *netlist* de manière textuelle mais cela serait très certainement long et fastidieux, et par conséquent source d'erreurs.

Une méthode plus robuste envisageable est d'utiliser une commandes fournie par l'outil de placement-routage : *'insert_buffer'*. Cette commande insère une cellule sur le fil spécifié et gère automatiquement les connexions.

Il faut néanmoins être vigilant afin d'éviter que l'outil n'enlève pas la cellule ajoutée en appliquant l'exception *'set_dont_touch'* sur celle-ci. En effet, la difficulté avec les outils (non prévus pour notre usage) fait que les cellules ajoutées seront vues comme « inutiles » et qu'elles seront donc susceptibles d'être enlevées lors d'une phase d'optimisation.

Évidemment, cette méthode est difficilement utilisable sur de grands circuits et demande un certain nombre de passes afin d'obtenir le circuit final. Elle peut cependant être utilisée pour corriger les dernières violations qui resteraient après une approche plus automatisée.

5.3.2 Méthodes statiques

Nous avons déjà exprimé certaines réserves à utiliser des contraintes statiques pour la STA, et pour cause, elles demandent de faire des études de timing dans chaque *corner* PVT pour appliquer les contraintes adaptées à chacun d'eux.

Cependant, le fait est que ces méthodes restent nécessaires la résolution de certaines violations de *timing*.

5.3.2.1 Contraintes locales de délai minimum, identification des conflits

Dans un premier temps, nous avons utilisé des contraintes directes permettant de spécifier les contraintes de chemin à chemin.

De la même manière que pour la STA, la première étape est de récupérer le *timing* de référence (de la donnée par exemple), puis de placer la contrainte correspondante dans le contrôleur.

Il faut ensuite s'assurer que l'outil, *IC-Compiler*™ en l'occurrence, soit bien configuré pour corriger les violations qui pourraient survenir :

— Dans le cas d'utilisation de contraintes en délai maximum, il faut alors que le scénario utilisé dans l'outil soit paramétré pour optimiser le *setup*

— Dans le cas d'utilisation de contraintes en délai minimum, il faut alors que le scénario utilisé dans l'outil soit paramétré pour optimiser le *hold* et appliquer la commande *'set_fix_hold'*

Sans ces paramétrages, les contraintes seront considérées par la STA mais aucune modification ne sera apportée au circuit pour corriger les violations.

Cette approche directe permet en théorie d'assurer toutes les contraintes assez simplement. Cependant, plusieurs points peuvent s'avérer bloquant en utilisant ce type de contraintes :

— L'application de contraintes, ou exceptions, locales peut nécessiter de couper le chemin à un point de référence, comme par exemple dans le cas des contraintes de sélection. Cependant, cette coupure peut couper d'autres chemins sur lesquels il serait nécessaire d'appliquer une contrainte. Sur la Figure 5.12, une contrainte de sélection coupe le chemin de requête pour le *setup*. Cela implique que pour appliquer les deux contraintes, il

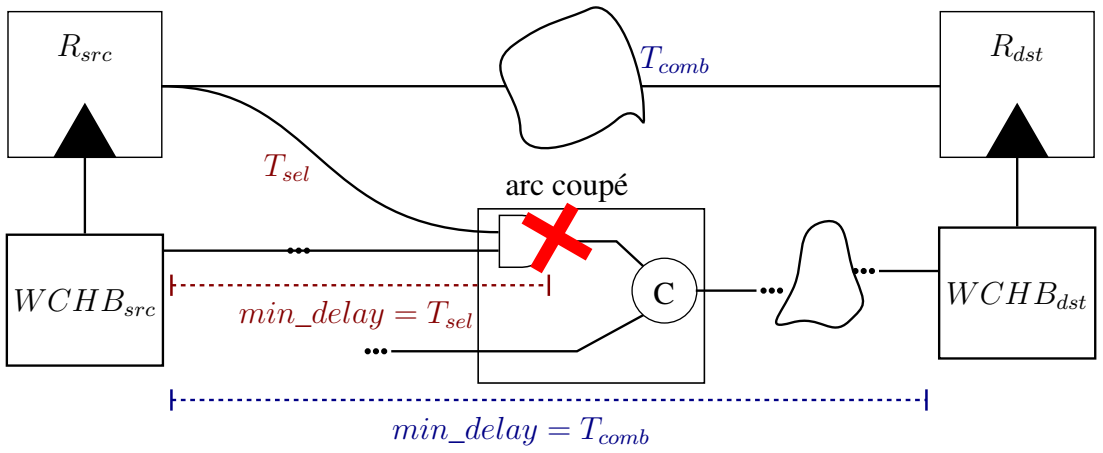


Figure 5.12: Conflit entre une contrainte de sélection et une contrainte de *setup* sur un même chemin

est nécessaire de disposer deux scénarios.

— Chaque contrainte est vue par l'outil comme une exception. La prise en compte de ces multiples exceptions peut engendrer des temps de calcul relativement longs même pour des circuits de taille modeste.

— L'outil *IC-Compiler*™ n'étant pas un outil de STA, les mesures de temps qu'il renvoie sont moins précises que celles d'un outil spécialisé comme *Primetime*™ par exemple.

Afin de régler les deux derniers points, il est possible de procéder à une étude préliminaire avec un outil de STA pour extraire directement les contraintes à appliquer sous un format commun, nommé SDC pour *Synopsys Design Constraints*.

Enfin, la topologie du contrôleur asynchrone global est telle qu'elle peut mener à des « conflits » de contraintes. En effet, pour chaque contrôleur de registre, en fonction de son *fanin* et de son *fanout*, de multiples contraintes peuvent être à vérifier.

Prenons par exemple le cas de la Figure 5.13 où est représenté le chemin de requête entre 4 contrôleurs WCHB ainsi que les contraintes associées.

Dans ce chemin de requête, toutes les portions du chemin ont deux contraintes différentes appliquées. Considérons d'abord les chemins à destination de $dst1$. Les deux contraintes de *setup* à appliquer sont $10ns$ depuis $src1$ et $15ns$ depuis $src2$. La manière la plus économe en délai de résoudre cette situation est d'avoir un délai de $10ns$ sur la portion de chemin commune à toutes les requêtes : entre le *join* et le *fork*. Il reste alors $5ns$ à placer pour la contrainte depuis

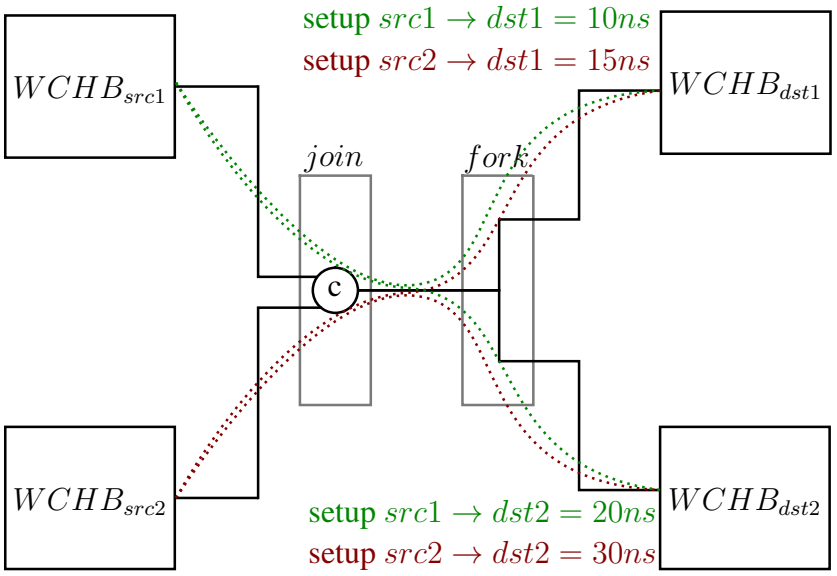


Figure 5.13: Croisement de contraintes de *setup* pouvant mener à un conflit

$src2$, on peut donc les placer au plus proche de $src2$, avant le *join*. Passons maintenant au cas des chemins à destination de $dst2$. Pour ne pas déteriorer les performances sur les chemins à destination de $src1$, le seul chemin restant sur lequel nous pouvons influer est le chemin au plus proche de $dst2$, après le *fork*. Or on à ici 2 délais manquants : $10ns$ depuis $src1$ et $15ns$ depuis $src2$. Cette situation est un cas ou l'outil estime qu'il y à un conflit.

Pour cette raison, nous avons généré nos contrôleurs de circuits désynchronisés d'une autre manière. La Figure 5.14 présente comment cette situation source de conflit serait générée par la méthode présentée dans Chapitre 4.

En utilisant cette méthode de construction, la surface du contrôleur global est augmentée, chaque chemin dispose en contrepartie d'au moins une portion où une seule contrainte s'applique : les situations de conflit son résolues.

À l'utilisation, il s'avère que l'utilisation de ces contraintes point-à-point donne une QoR variables. L'outil n'étant pas explicite vis-à-vis de ses choix d'optimisations, il est difficile de déterminer ce qui l'empêche de parvenir à un résultat plus satisfaisant lors de la génération des délais.

Nous supposons cependant que la stratégie synchrone des outils empêche la bonne résolution des contraintes appliquées. En effet, à chaque étape d'insertion de délai, l'outil utilise certainement une fonction de coût qui détermine si l'opération effectuée est pertinente ou non. Nous pensons que dans ce cas, la violation de contrainte de délai minimum est considéré moins importante que l'insertion de cellules supplémentaires et que donc l'insertion de délais se stoppe à partir d'un certain point. Il existe dans les outils des commandes afin de modifier la priorité de certaines contraintes. La priorisation des contraintes de délai minimum n'a cependant eu qu'un effet marginal sur le QoR.

En conclusion, l'utilisation de ces contraintes locales est conseillé soit pour des designs

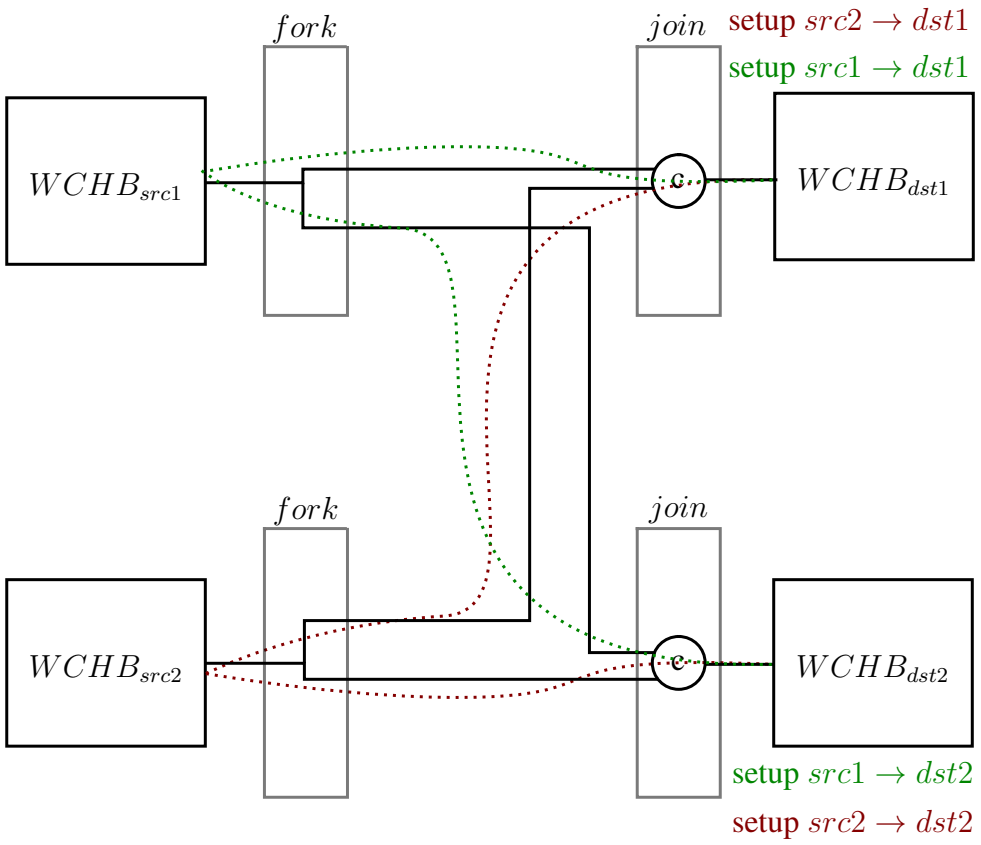


Figure 5.14: Configuration de transitions évitant les conflits de contraintes

simples soit pour des contraintes ponctuelles comme les signaux de sélection ou l'interfaçage avec un protocole synchrone.

5.3.2.2 Génération de délais en utilisant la synthèse d'arbre d'horloge

Une des fonctionnalités de l'outil de placement routage qui n'a en général aucun mal à générer des délais est le moteur de synthèse d'arbres d'horloge. En effet, celui-ci doit équilibrer des arbres parfois de grande taille et sur de grandes distances. À cette fin, il utilise des *buffers* ou des inverseurs afin d'obtenir des temps d'arrivée les plus proches les uns des autres possibles, c'est-à-dire avec un *skew* minimal.

L'outil possède une procédure permettant d'appliquer des exceptions à des arbres d'horloges : *'set_clock_tree_exception'*. Plusieurs exceptions vont nous servir par la suite :

— *stop_pin* : spécifie une entrée de cellule standard comme point d'arrêt de l'arbre d'horloge

— *float_pin* : même fonctionnalité que la *stop_pin*. Prend en plus en paramètre une valeur de temps qui va venir s'ajouter à la latence totale jusqu'au point d'arrêt. La *pin* considérée sera équilibrée en prenant en compte ce temps.

— *exclude_pin* : spécifie une entrée de cellule standard à ignorer pendant l'équilibrage.

La méthode appliquée pendant cette thèse est la suivante :

— Étant toujours dans une approche statique, il faut auparavant collecter tous les temps à respecter.
— Chaque signal de contrôle de registre est défini comme source pour une horloge.
— Pour chaque contrôleur de registre :
  — Sur chaque contrôleur dans le *fanout* du contrôleur courant, on place une exception de type *float_pin* avec le *timing* de la donnée associée à compenser.
  — On exclut les *pins* non pertinentes dans l'arbre d'horloge (*pins* servant aux acquittements par exemple).
— On synthétise les arbres d'horloge

La Figure 5.15 illustre la manière dont se placent les différents points de synchronisation pour l'horloge définie au point source.

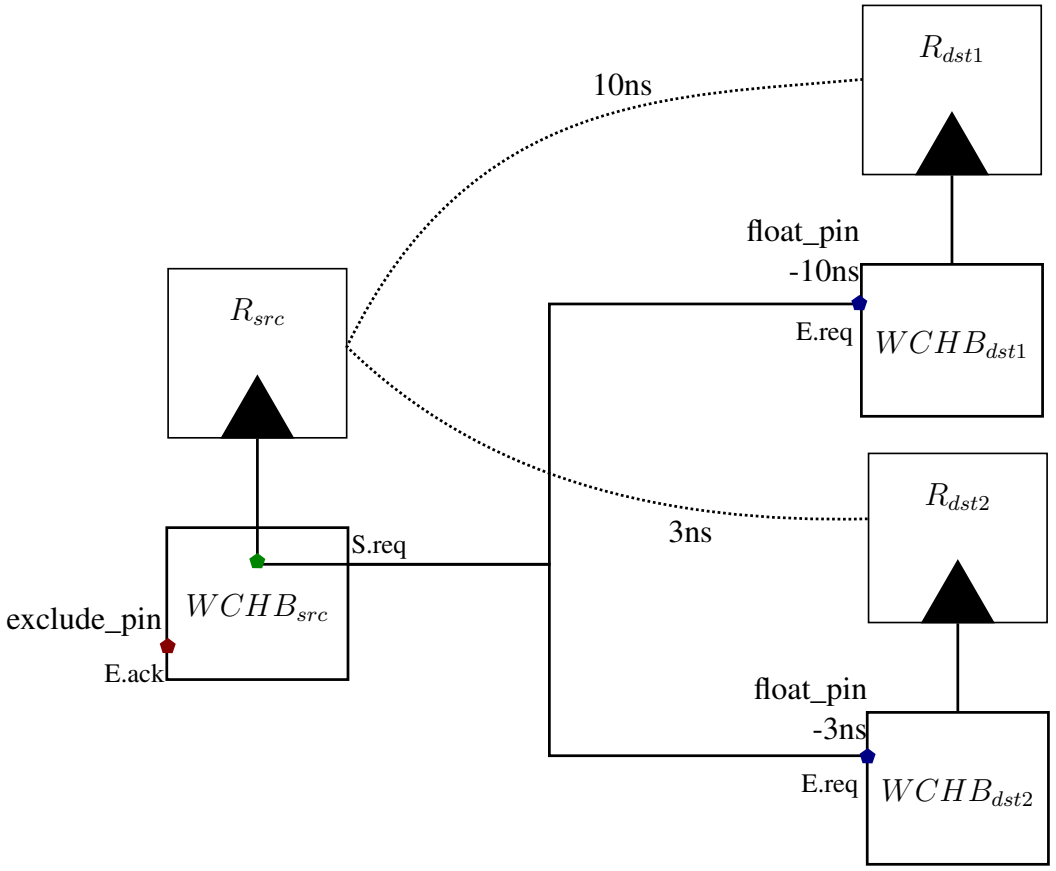


Figure 5.15: Positionnement des exceptions d'arbres d'horloge pour la génération de délai, méthode statique

Dans l'exemple, on définit l'horloge dans $WCHB_{src}$. Par défaut, le moteur de CTS (*Clock Tree Synthesis*) ne prend en compte que les *pins* étant naturellement considérées comme des points d'arrêt : les *pins* CP des registres contrôlés. Pour ne pas risquer d'équilibrer des points non pertinents, il est plus prudent de placer une exception *exclude_pin* sur l'acquittement sortant du contrôleur. Ensuite, en connaissant les temps de traversée des chemins combinatoires, des *float_pins* sont placées avec les *timings* correspondant à chaque chemin.

Il est à noter que le comportement des *floats pins* est tel qu'en plaçant des *timings* négatifs, en paramètre des *float_pins*, alors le moteur de CTS cherchera à ajouter du délai pour compenser ces *timings*.

Comme l'outil qui est conçu pour générer des arbres d'horloges, il est plus aisé de spécifier et de *debugger* les différentes contraintes appliquées sur les arbres d'horloge. En effet, une fois les arbres d'horloges et leurs exceptions appliquées, le concepteur peut exécuter la commande *'check_clock_tree'* qui permet de vérifier que la spécification ne comporte pas de conflit et qui, lorsque cela est possible, propose des solutions pour gérer ces problèmes.

Toutefois, en appliquant les contraintes sur un circuit complet, des situations considérées comme des conflits peuvent subsister, bien qu'elles n'aient pas forcément de sens au niveau circuit. Pour éviter ces conflits, nous avons décidé de diviser les contraintes par groupes ne générant pas de conflit.

5.3.3 Méthode dynamique : équilibrage *launch/capture*

Le mécanisme de synthèse d'arbre d'horloge cherche à équilibrer les points finaux de l'arbre entre eux. En partant de ce principe, nous avons cherché à équilibrer le chemin de donnée, ou chemin de *launch*, avec le chemin de la requête dans le contrôleur, ou chemin de *capture*.

La Figure 5.16 présente les différents points d'intérêt de l'arbre d'horloge avec cette méthode.

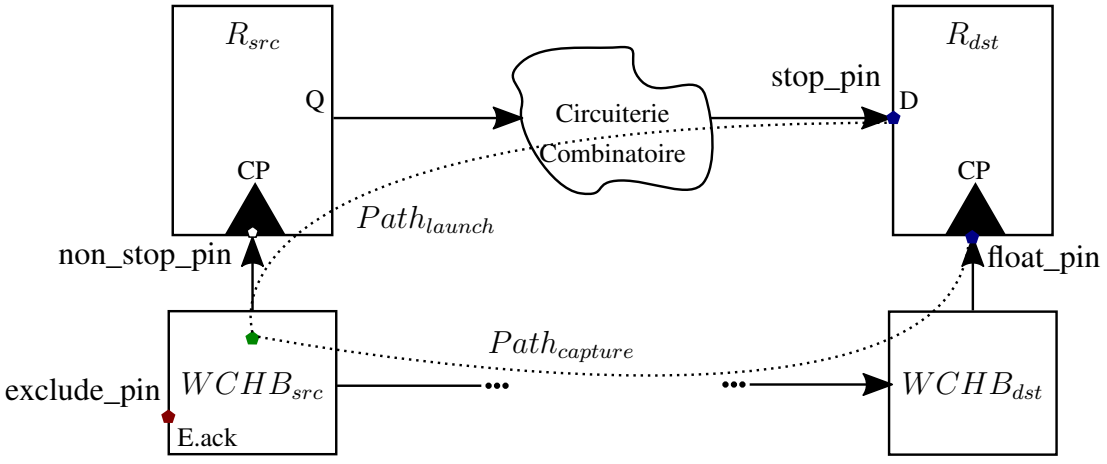


Figure 5.16: Positionnement des exceptions d'arbres d'horloge pour la génération de délai, méthode dynamique

En prenant l'exemple comme référence, les *pins* CP des registres contrôlés par $WCHB_{src}$ sont normalement considérés comme des points d'arrêt. Pour que l'outil prenne en compte le chemin $R_{src}/CP \rightarrow R_{dst}/D$, une exception *non_stop_pin* est placée sur l'entrée d'horloge du registre source. Ensuite, une exception *stop_pin* est ajoutée à l'entrée de donnée du registre de destination, R_{dst}/D, pour indiquer à la génération d'horloge un point d'arrêt de l'arbre.

Du côté du chemin de *capture*, une *float_pin* doit être placée pour prendre en compte le temps de *setup* de la bascule d'arrivée qui ne sera pas considéré par l'équilibrage. Le temps de compensation de la *float_pin* peut également servir pour ajouter une marge indépendante dans ce cas du chemin de donnée.

Dans l'idéal, l'équilibrage entre ces deux points permet d'assurer la contrainte de *setup* et surtout de reporter automatiquement des changements dans le chemin de données sur l'équilibrage de ces points.

Quelques précautions sont cependant à prendre en compte pour assurer une qualité de résultat correcte :

— Il faut protéger les chemins combinatoires pendant l'équilibrage d'horloge avec une exception *dont_touch_subtree*. En effet, ce n'est pas au mécanisme de CTS d'apporter des optimisations sur ces chemins, il ne doit être considéré que comme une référence.

— Isoler uniquement le chemin critique avec des exceptions *exclude_pin* sur les chemins non pertinents. Entre le point de départ (R_{src}/Q) et le point d'arrivée (R_{dst}/C), le nuage combinatoire peut contenir plusieurs chemins, donc avec des temps de traversée potentiellement très différents comme par exemple sur la Figure 5.17.

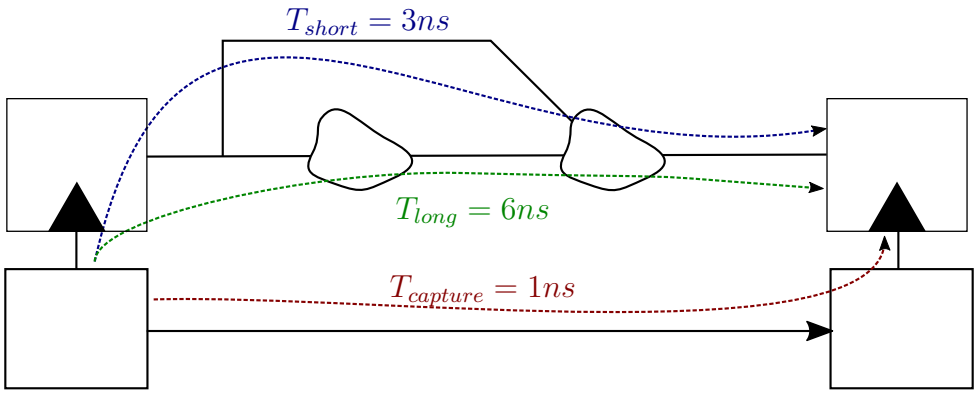


Figure 5.17: Multiples chemins combinatoires entre un même *startpoint* et un même *endpoint*.

En utilisant cet exemple, dans l'état initial, l'outil calcule un *skew* de $T_{long} - T_{capture} = 5ns$. En insérant des délais sur le chemin de contrôle, on augmente $T_{capture}$ jusqu'à atteindre $T_{capture} > T_{short}$. Après cette optimisation, le *skew* calculé par l'outil est de $T_{long} - T_{short} = 3ns$. Cependant, comme l'insertion de délai dans le contrôleur n'améliore plus le *skew*, l'outil va naturellement stopper l'insertion de délais pour ne pas augmenter la surface « inutilement ». Dans ce cas, on aurait donc $T_{capture} \approx T_{short}$, ce qui ne satisfait pas la contrainte de *setup*, alors qu'il faudrait $T_{capture} \approx T_{long}$.

— Une fois le chemin critique isolé, il est aussi nécessaire de contraindre le chemin de données à avoir le même comportement que le pire cas, *i.e.* le plus long temps de traversée. En effet, les cellules combinatoires n'ont pas les mêmes temps de traversée si leur sortie passe à un niveau haut que si elle passe à un niveau bas. Cette variation est en général assez faible mais peut devenir contraignante sur un chemin comprenant beaucoup de

cellules. Pour forcer le chemin dans la configuration pertinente, il faut utiliser des exceptions *'set_case_analysis'* pour fixer le sens de la transition de chaque cellule sur le chemin critique.

Une fois ces préparations effectuées, on peut procéder à la synthèse des différents arbres d'horloge. Le principal avantage de cette méthode est que des modifications du chemin de données seront automatiquement prises en compte et que l'équilibrage sera ajusté pour correspondre à la contrainte.

Cette méthode étant dynamique, elle peut être exercée simultanément sur de multiples *corners* PVT. Cependant, les temps de propagation dans les différents *corners* peuvent varier grandement, et surtout de manière différente de cellule à cellule.

Or, le mécanisme de synthèse de délai de l'arbre d'horloge n'utilise que deux types de cellules, des *buffers* et des inverseurs. Toutefois, la variation du temps de propagation dans les chemins combinatoires est peut être très différente de la variation du temps de propagation de la requête dans le contrôleur du fait de l'utilisation de ces cellules qui se comportent différemment avec les variations de tensions et de température que les cellules logiques à plusieurs entrées

5.3.4 Pistes d'amélioration de la génération des délais

Une part significative du travail de thèse a porté sur la génération des délais. Nous avons conservé une approche utilisant les fonctionnalités de l'outil mais nous pensons avoir atteint les limites de ce que l'outil peut nous offrir comme solution dans ce cadre.

Néanmoins, nous avons exploré plusieurs pistes pour améliorer le QoR de cette génération de délais.

5.3.4.1 Variabilité des temps de traversée de cellules en fonction des conditions d'opération

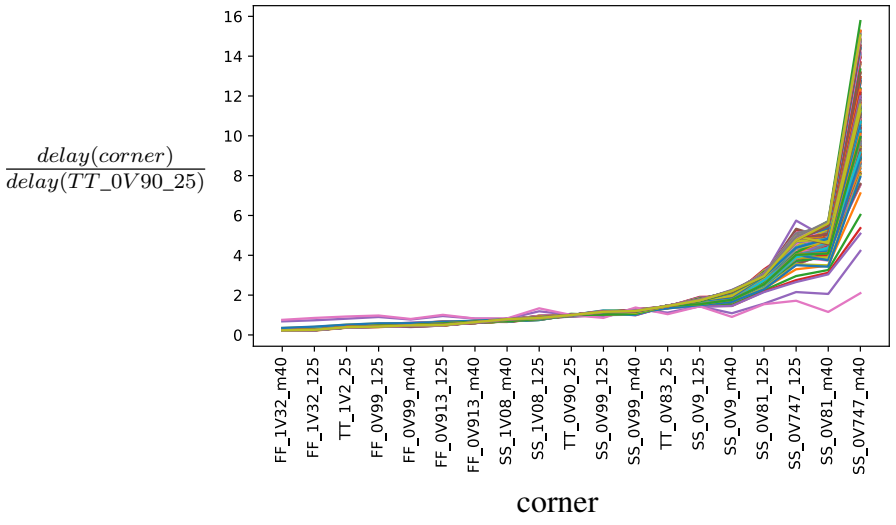
L'un des principaux avantages des circuits désynchronisé est leur robustesse. Celle-ci est néanmoins dépendante du fait que le délai utilisé dans le contrôleur varie de la même manière que le délai dans le chemin critique associé en fonction du *corner* PVT.

Cependant, les méthodes décrites précédemment n'utilisent que des cellules à une entrée et une sortie : principalement des inverseurs ou des *buffers*.

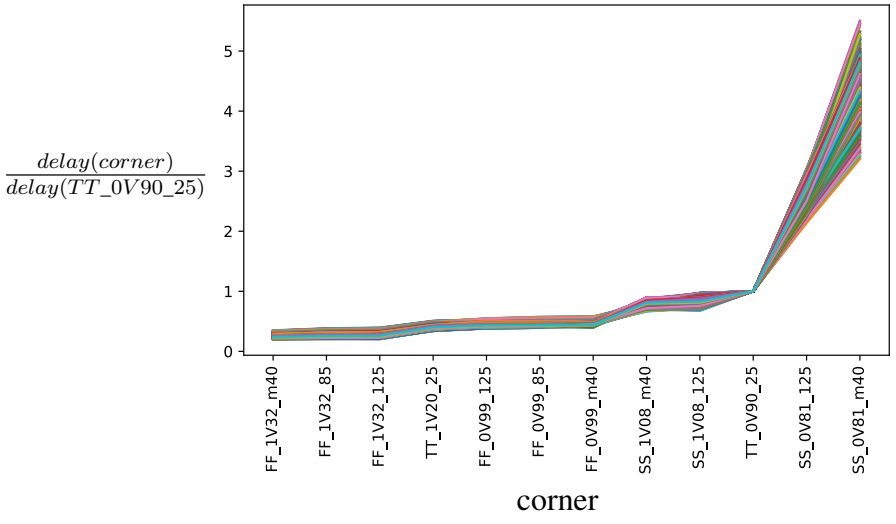
Cette limitation peut détériorer la robustesse des circuits obtenus puisque les cellules utilisées pour créer du délai n'ont pas nécessairement le même comportement que le chemin combinatoire dans les différentes conditions d'opération.

En effet, la dispersion des temps de traversée des cellules est très dépendante de la technologie utilisée mais aussi des fournisseurs de cellules standards. Sur la Figure 5.18 est représentée la variation du temps de traversée de toutes les cellules de deux librairies dans la même technologie. Les résultats sont normalisés pour chaque cellule par rapport à leur temps de traversée

dans le *corner* typique : TT_0V90_25 (procédé typique, tension d'alimentation à 0.9 Volts, et température à 25°C).



(a) Fournisseur A



(b) Fournisseur B

Figure 5.18: Évolution des temps de traversée (normalisés par rapport à TT_OV9_25) à travers les cellules standard de librairies de même technologie mais de fournisseurs différents. *Corner :* (Procédé)_(Tension d'alimentation)_(Température)

Bien que les points de caractérisations ne soient pas exactement les mêmes, certains points sont communs et permettent la comparaison. Nous pouvons ainsi noter des comportements très différents en fonction du fournisseur considéré. En effet, le cône de dispersion pour la librairie du fournisseur B parait assez régulière et resserrée, l'amplitude de la variation entre le cas typique et le cas extrême reste contenu entre des facteurs 3 et 5.

Dans le cas de la librairie du fournisseur B, si le comportement dans les *corners* rapides semble être relativement régulier, la dispersion dans les *corner* lents l'est moins et on remarque

un certain nombre de croisements entre les courbes. L'amplitude des variations reste cependant similaire à celle observée pour le fournisseur B en prenant les points de comparaison communs (et en excluant les extrêmes minimums).

Ces variations de comportement impliquent que les chaînes de délais générées peuvent avoir des variations très différentes aux conditions d'opérations, en fonction des cellules utilisées mais aussi de variables fonction des fournisseurs choisis.

5.3.4.2 Utilisation de cellules combinatoires pour créer des délais

Dans [40], Moreno et Cortadella ont fait une étude sur la génération de lignes de délais utilisant uniquement des cellules logiques.

Toute cellule combinatoire peut en effet être utilisée comme une cellule de délai. Pour transformer une cellule combinatoire en cellule de délai, à une entrée et à une sortie, il faut fixer les entrées non utilisées à une valeur permettant de sensibiliser la sortie à une variation de l'entrée conservée.

Par exemple, comme présenté dans la Figure 5.19, pour une porte *AND* à deux entrées, il faut fixer une entrée à 1, ici A2. Ainsi, un changement de valeur sur l'entrée I(A1) sera propagé à la sortie Z.

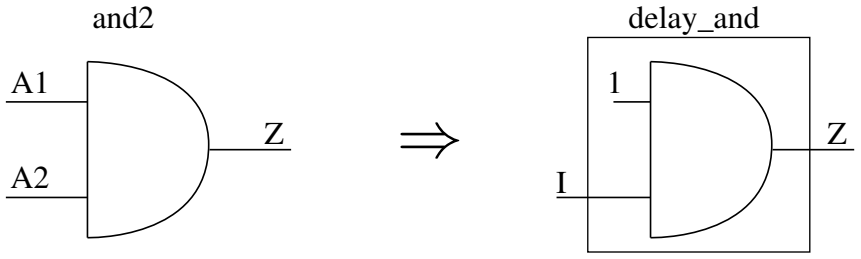


Figure 5.19: Utilisation d'une cellule *AND* comme cellule de délai

Les travaux de Moreno se basent sur ce principe pour créer un bibliothèque étendue de cellules de délai. Avec cette dernière, ils proposent de construire la chaine de délais de la manière suivante :

— On part d'une chaine de délais vide
— On essaie chaque cellule disponible comme cellule de délai
— On conserve les n chaines de délais les plus proches de la cible
— Pour chaque chaine de délais conservée, on ajoute chaque cellule de délai disponible
— On conserve les n chaines obtenues les plus proches de la cible
— ...
— Lorsque la cible est atteinte, on obtient une chaine de délais finale.

D'après les auteurs cette méthode permet donc d'obtenir des chaines de délais très proches de la cible, dans les résultats présentés, la déviation à la cible est toujours de moins de 5%.

Cette méthode a été implémentée pendant nos travaux et s'avère en effet donner de bons résultats sur un *corner*. En outre, bien que moins marquée qu'en utilisant uniquement des *buffers* et inverseurs, les variations de temps de transition entre les différentes conditions d'utilisation sont restées significatives.

Le temps de calcul supplémentaire nécessaire à l'application de cette méthode est aussi relativement long. Du fait de sa complexités et des faibles bénéfices apportés, cette méthode n'a finalement pas été retenue dans ces travaux pour la génération de délais. Elle reste cependant une alternative intéressante pour la génération ponctuelle de chaine de délais.

5.3.4.3 Méthode de duplication du chemin critique

Le dernier point auquel nous allons nous intéresser est un concept qui a été envisagé pendant mes travaux de thèse. Nous n'avons malheureusement pas eu le temps de tester cette méthode pour tirer des conclusions définitives.

La limitation principale de la génération de délais, outre les limitations liées aux outils, a été de pouvoir la rendre robuste aux changements de conditions d'opération. La cause étant la disparité des variations de temps de traversée des différentes cellules.

Cependant, la simple identification du chemin critique donne également la chaîne de délais parfaitement taillée pour lui correspondre : le chemin critique lui-même.

L'idée est donc, comme dans la Figure 5.20, de récupérer le chemin combinatoire du chemin critique, de fixer à la valeur permettant de sensibiliser la sortie du chemin à une variation de l'entrée, et de l'utiliser comme chaîne de délai.

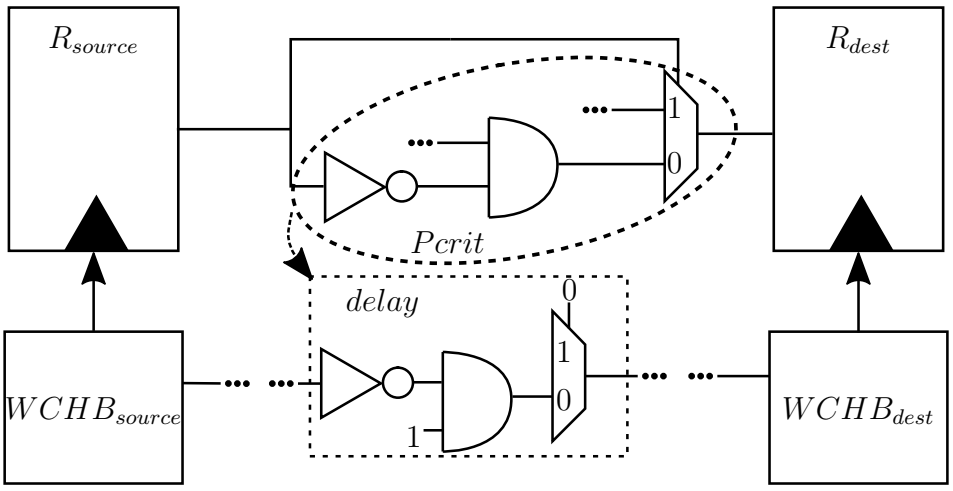


Figure 5.20: Duplication du chemin critique pour utilisation en tant que chaîne de délai

Il est à noter qu'au temps de traversée de cette chaîne de délais, celui des éléments du contrôleur asynchrone (WCHB, transitions) qui peuvent ainsi couvrir les différents temps internes (T_{setup}, temps de $CP \rightarrow Q$) ne sont pas pris en compte dans le chemin combinatoire. Dans le pire des cas, le délai à générer devrait rester minime comparé au temps total nécessaire.

Le principal risque identifié de cette méthode est la possibilité que le chemin critique soit différent dans chaque condition d'opération, ce qui est généralement le cas. Deux solutions nous paraissent alors envisageables :

— Comme présenté dans la Figure 5.21, on peut dupliquer le chemin critique de chaque *corner* et faire se rencontrer la traversée des chemins avec une porte 'AND' en bout des chaînes de délai. Il est évident qu'avec cette méthode le délai peut atteindre une surface conséquente. Nous pouvons par contre pressentir que le délai ainsi généré doit être très robuste. De plus, la performance serait dans ce cas conservée puisque chaque *corner* serait contraint par le chemin critique qui lui correspond.

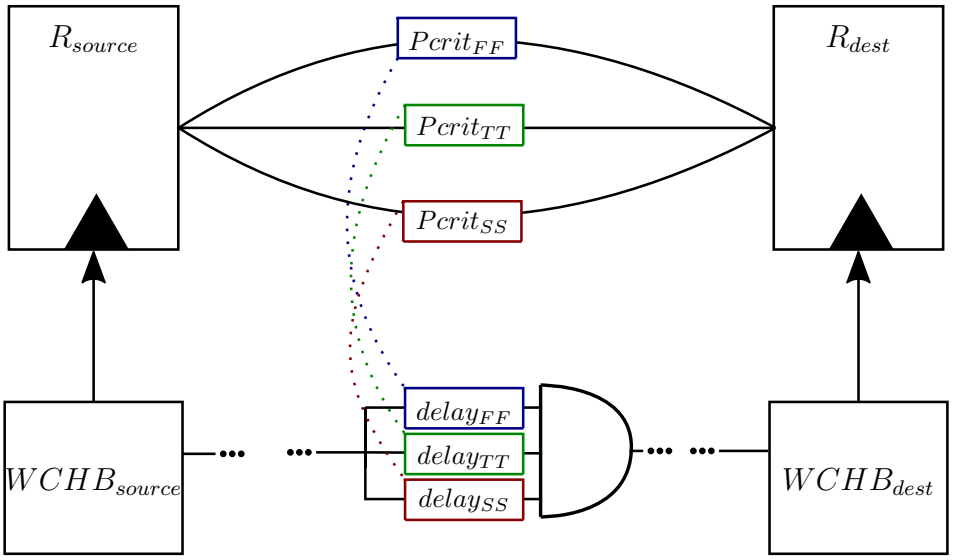


Figure 5.21: Solution pour assurer la robustesse dans plusieurs conditions d'opération : Duplication de chemin par *corner*

— Le cas typique reste le cas le plus courant d'utilisation. Nous pouvons donc imaginer, comme dans la Figure 5.22, dupliquer le chemin critique du *corner* typique et de générer le délai manquant avec l'une des méthodes vues auparavant. Cependant, si un fort écart de comportement entre deux *corners* se présente, alors la valeur de délai à générer risque d'être conséquente. De ce fait, en plus de la surface nécessaire supplémentaire, les performances peuvent être détériorées avec le *corner* « typique ».

Au terme de ce chapitre, nous avons donc à disposition tous les outils nécessaires pour assurer le bon fonctionnement et la robustesse des circuits désynchronisés. Dans le prochain chapitre, nous allons valider le flot complet de désynchronisation et revenir sur les différents résultats obtenus qu'ils soient issus de mesures physiques ou de simulations.

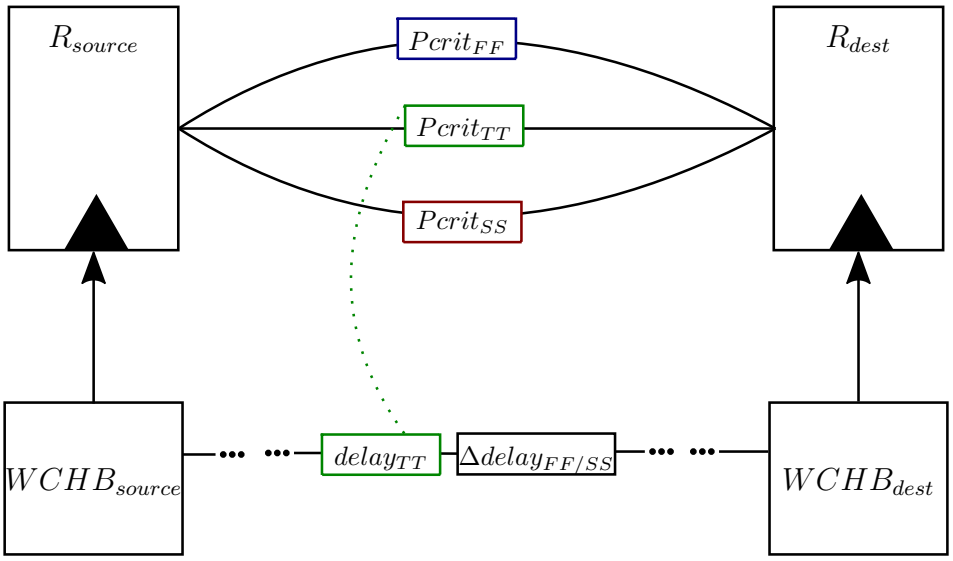


Figure 5.22: Solution pour assurer la robustesse dans plusieurs conditions d'opération : Duplication du chemin en *corner* typique avec compensation pour les cas extrêmes

Chapitre 6

Application à des circuits de chiffrement

L'entreprise encadrant la thèse étant spécialisée dans le développement de circuits sécurisés, les premières cibles pour la désynchronisation ont naturellement été des circuits de chiffrement qui sont connus pour être gourmands en énergétie.

Nous avons en particulier étudié le crypto-processeur *Advanced Encryption Standard* (AES) qui utilise l'algorithme de Rijndael [16]. Dans un premier temps, nous avons testé la désynchronisation sur un circuit, appelé *AES_SC*, existant dans l'entreprise. L'architecture de ce circuit a été désynchronisée suivant plusieurs méthodes :

— Méthode standard
— Méthode CAR
— Méthode CAR dite déroulée, ou *unrolled*
— Méthode dite raffinée, ou *refined*, utilisant des transitions sélectives
— Méthode ROC, utilisant un oscillateur en anneaux ou *Ring oscillator* (ROC)

Ces 5 implémentations différentes ont été embarqués, ainsi qu'une version synchrone de l'AES de base, dans une puce de test et ont pu être caractérisés physiquement en laboratoire.

Pour élargir le panel de comparaisons, nous avons également effectué la désynchronisation sur une architecture différente d'AES qui a été trouvée sur la base de données OpenCores [25]. Cette dernière implémentation, appelée *tinyAES*, a été implémentée seulement en utilisant la méthodologie CAR.

N.B. : *les noms des registres de l'AES_SC ont été changés pour des raisons de confidentialité. En ce qui concerne le tinyAES, les noms ont aussi été changés mais dans un souci de lisibilité.*

6.1 Désynchronisation des circuits

6.1.1 Désynchronisation standard d'*AES_SC*

La première étape pour désynchroniser un circuit suivant notre méthode est d'obtenir le réseau de registres. Pour l'*AES_SC*, la forme matricielle de ce réseau est la Figure 6.1.

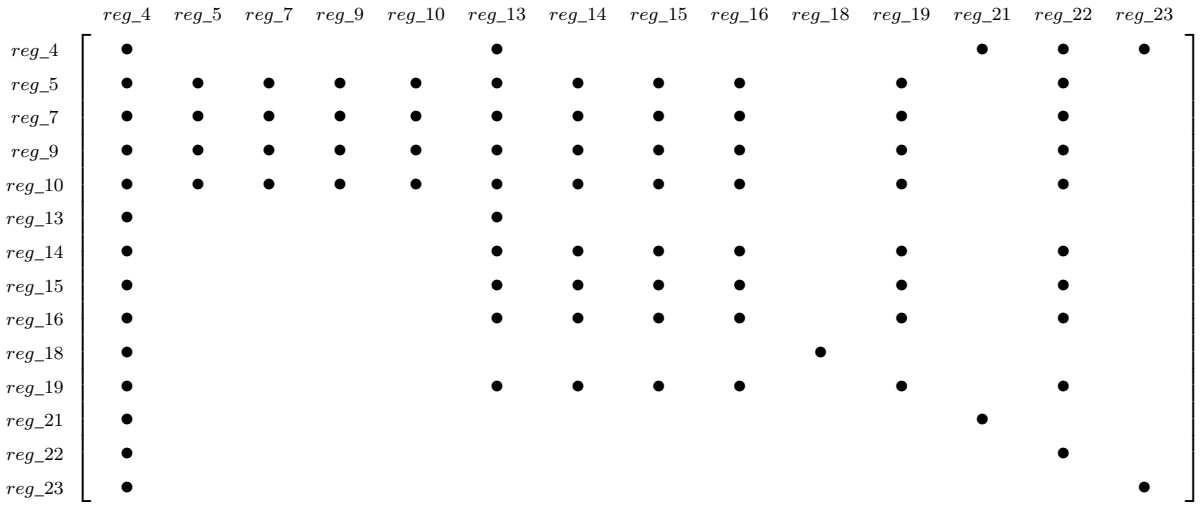


Figure 6.1: Réseau de registres matriciel de l'*AES_SC*

Depuis ce réseau de registres initial, en insérant les *buffers* nécessaires, nous pouvons déduire le réseau de registres qui donne un réseau asynchrone, et donc le contrôleur associé, vivace. Sur la Figure 6.2 est présenté ce RN. Étant généré automatiquement par l'outil, les notations sont différentes de celles définies dans le Chapitre 3. Nous utiliserons ici et dans la suite :

— Des rectangles à double bords pour les registres, dont le contrôleur doit être initialisé.
— Des rectangles à simples bords pour les registres *buffer*.
— Des arcs dirigés noirs pour les connexions entre les registres.
— Des arcs dirigés rouges pour les connexions depuis les ports d'entrée.
— Des arcs dirigés verts pour les connexions vers les ports de sortie.

Avec le contrôleur de registres *bufferisé*, nous pouvons calculer les grandeurs suivantes :

— Le nombre de registres : $N_{reg}(\textbf{\textit{AES\_SC}}_{STD}) = 14 + 7 = 21$
— Le coefficient de linéarité : $C_{lin}(\textbf{\textit{AES\_SC}}_{STD}) = 5.6$
— Coût de *bufferisation* : $Cost_{buff}(\textbf{\textit{AES\_SC}}_{STD}) = (6 * 32 + 3) * 31 \approx 6000\ u.a.$
— Coût du contrôleur : $Cost_{ctrl}(\textbf{\textit{AES\_SC}}_{STD}) \approx 2400\ u.a$

Le nombre de bascules présentes dans le circuit *AES_SC* est $Nb_{dff}(AES\_SC) = 280 \approx 8700\ u.a.$. On peut donc estimer le coût de la désynchronisation en termes de surface, comprenant le coût de la *bufferisation* et celui du contrôleur, à une surface équivalente à la totalité des registres dans le circuit. Nous pouvons donc supposer que la désynchronisation standard à un fort impact sur la surface du circuit.

Il est à noter que cette estimation ne prend pas en compte la surface de la circuiterie combinatoire, ni l'existence d'un arbre d'horloge complexe. Il faudra donc relativiser la valeur de cette estimation grossière dans la suite.

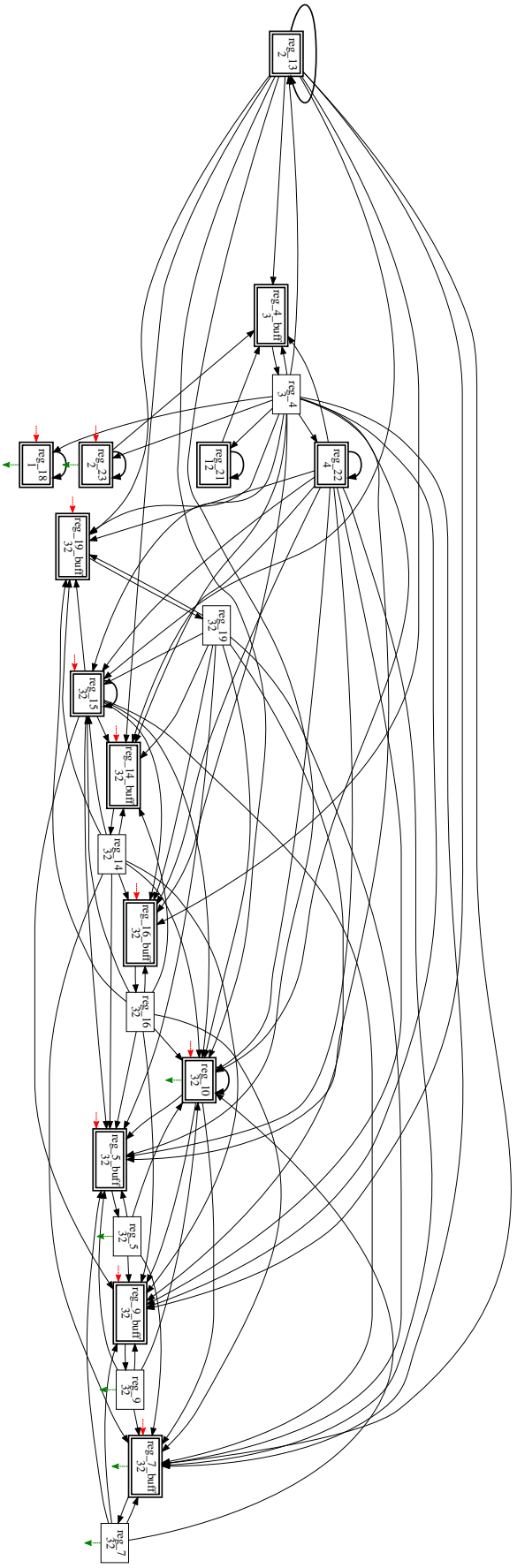


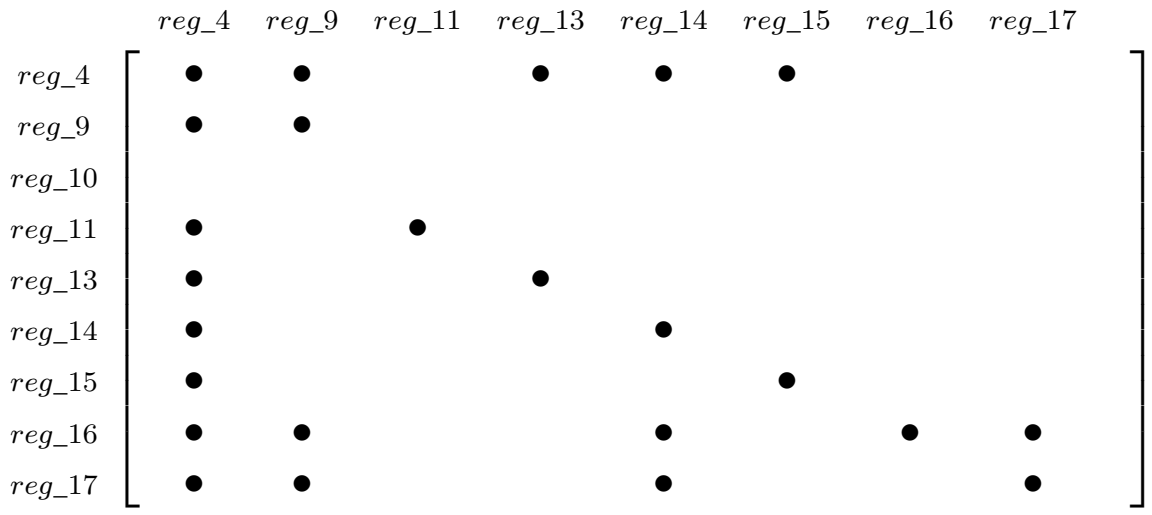
Figure 6.2: Réseau de registres du *AES_SC* initial, avec *bufferisation*

6.1.2 Désynchronisation CAR d'*AES_SC*

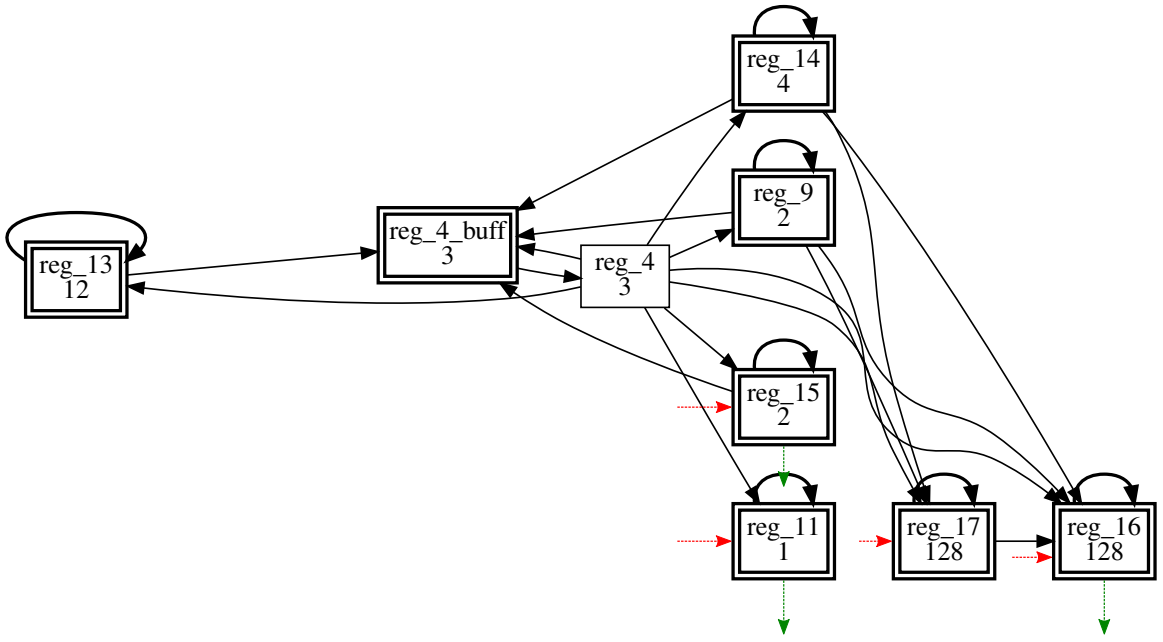
Pour la première application, il est intéressant d'appliquer chacune des stratégies afin d'étudier leur effet sur un circuit réel.

6.1.2.1 Stratégie *Upstream* CAR

Les résultats des deux variantes de cette stratégie sont présentés sur les Figures 6.3 et 6.4.



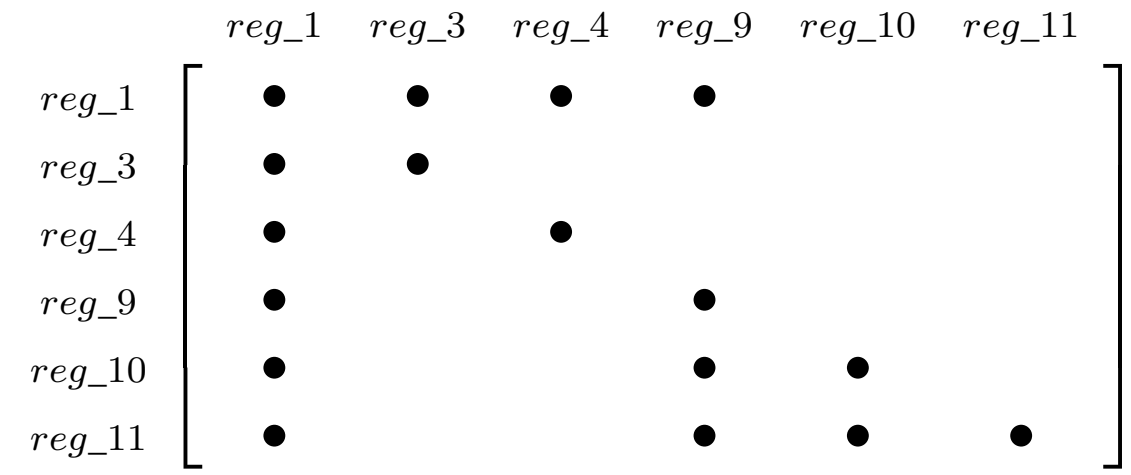
(a) Forme matricielle initiale



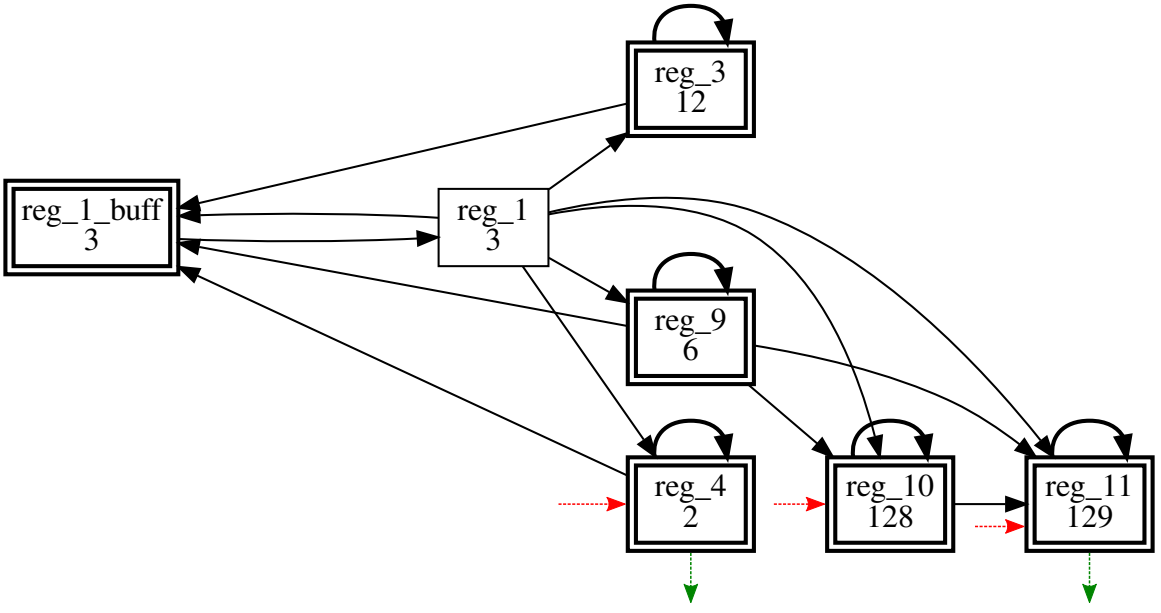
(b) Forme graphique vivace

Figure 6.3: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie UCAR *soft*

Intéressons-nous maintenant aux effets de ces simplifications sur les grandeurs étudiées précédemment :



(a) Forme matricielle initiale



(b) Forme graphique vivace

Figure 6.4: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie UCAR *greedy*

— UCAR_soft :
  — Le nombre de registres : N_{reg}(*AES_SC*$_{UCAR\_soft}$) $= 8 + 1 = 9$
  — Le coefficient de linéarité : C_{lin}(*AES_SC*$_{UCAR\_soft}$) $= 3.3$
  — Coût de *bufferisation* $Cost_{buff}$(*AES_SC*$_{UCAR\_soft}$) $= 3 * 31 \approx 90\ u.a.$
  — Coût du contrôleur $Cost_{ctrl}$(*AES_SC*$_{UCAR\_soft}$) $\approx 750\ u.a$
— UCAR_greedy :
  — Le nombre de registres : N_{reg}(*AES_SC*$_{UCAR\_greedy}$) $= 6 + 1 = 7$
  — Le coefficient de linéarité : C_{lin}(*AES_SC*$_{UCAR\_greedy}$) $= 3$
  — Coût de *bufferisation* $Cost_{buff}$(*AES_SC*$_{UCAR\_greedy}$) $= 3 * 31 \approx 90\ u.a.$
  — Coût du contrôleur $Cost_{ctrl}$(*AES_SC*$_{UCAR\_greedy}$) $\approx 530\ u.a$

Nous pouvons remarquer que la stratégie UCAR, de la méthode CAR, réduit grandement le nombre de registre à *bufferiser*. En effet, la mise en commun de certains registres permet de simplifier des boucles qui s'avèrent coûteuses à contrôler dans le cas standard.

Il est à noter que la réduction de la taille du contrôleur est également significative. Cet effet est attendu puisqu'en regroupant des registres, le nombre total de registres, ainsi que le nombre d'interconnexions entre registres, est réduit. Par conséquent les structures de synchronisation, ou transitions dans le réseau asynchrone, sont moins nombreuses mais également moins volumineuses.

Ayant tendance à faciliter le regroupement de registres, on observe cet effet de façon encore plus marqué avec la stratégie plus agressive, dite *greedy*.

6.1.2.2 Stratégie *Downstream* CAR

Les résultats des deux variantes de cette stratégie sont présentés sur les Figures 6.5 et 6.6.

Nous pouvons remarquer que les groupages utilisant les stratégies $UCAR\_soft$ et $DCAR\_soft$ donnent le même résultat. Nous ne nous attarderons donc pas sur cette dernière variante.

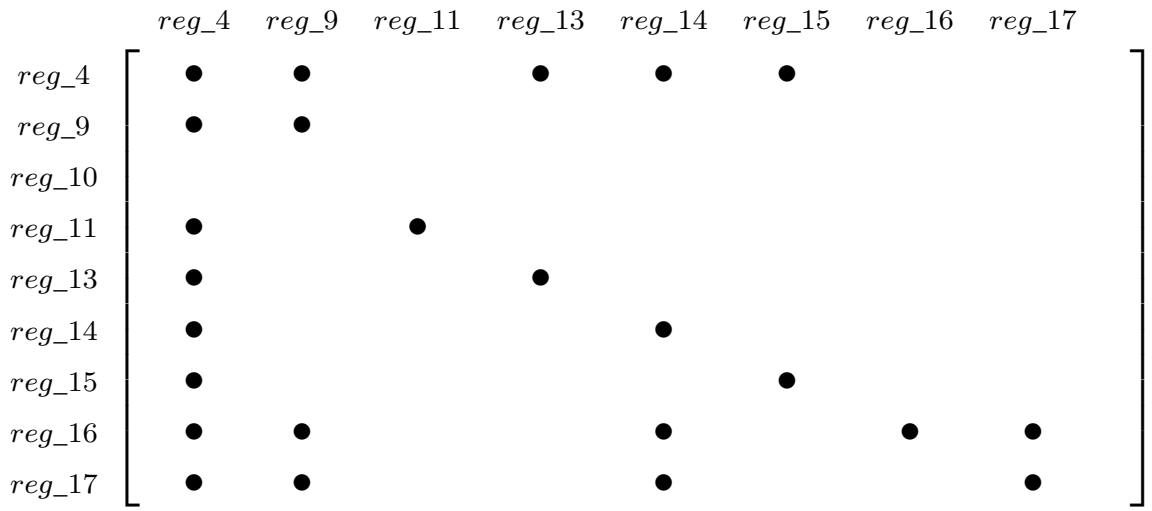
Intéressons-nous à l'impact de la stratégie DCAR_greedy :

— Le nombre de registres : N_{reg}(*AES_SC*$_{DCAR\_greedy}$) $= 4 + 1 = 5$
— Le coefficient de linéarité : C_{lin}(*AES_SC*$_{DCAR\_greedy}$) $= 2.8$
— Coût de *bufferisation* $Cost_{buff}$(*AES_SC*$_{DCAR\_greedy}$) $= 3 * 31 \approx 90\ u.a.$
— Coût du contrôleur $Cost_{ctrl}$(*AES_SC*$_{DCAR\_greedy}$) $\approx 310\ u.a$

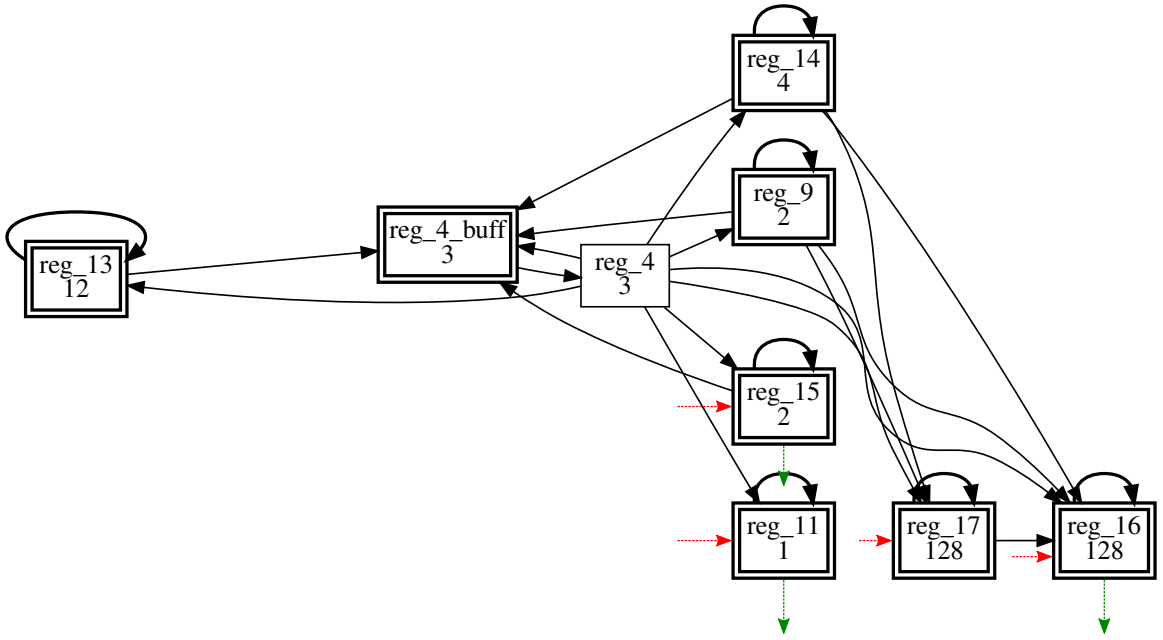
Nous pouvons noter que l'effet du groupage DCAR est plus fort que pour les stratégies UCAR. Cette différence de résultats suivant les stratégies de groupage provient directement de l'architecture considérée.

6.1.2.3 Stratégie *Completely* CAR

Intéressons-nous maintenant à la dernière stratégie proposée dans nos travaux. Les Figures 6.7 et 6.8 présentent les formes matricielles et graphiques pour les RN obtenus.

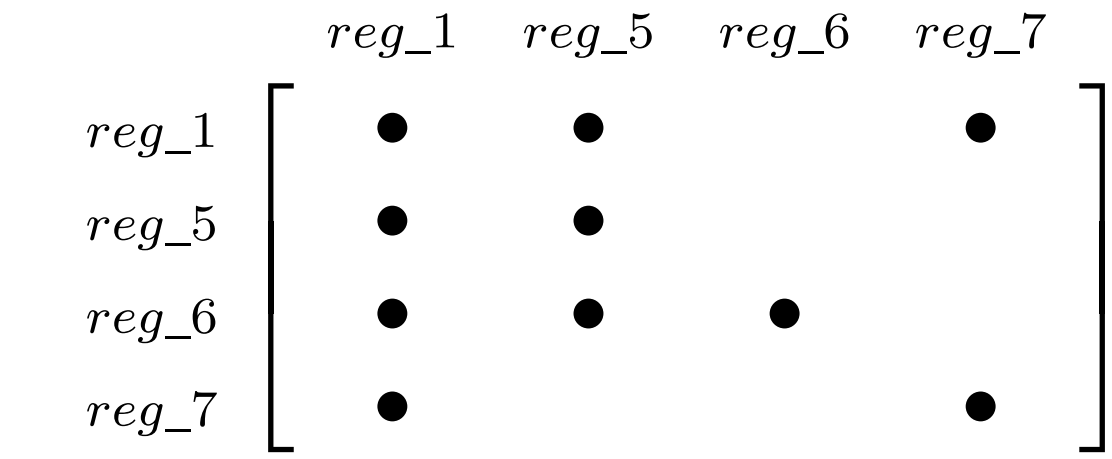


(a) Forme matricielle initiale

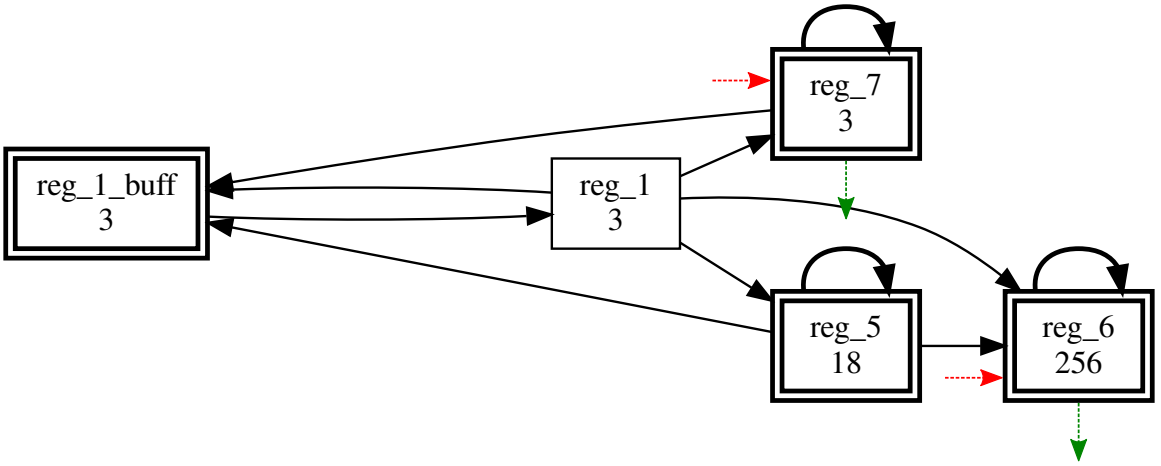


(b) Forme graphique vivace

Figure 6.5: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie DCAR *soft*

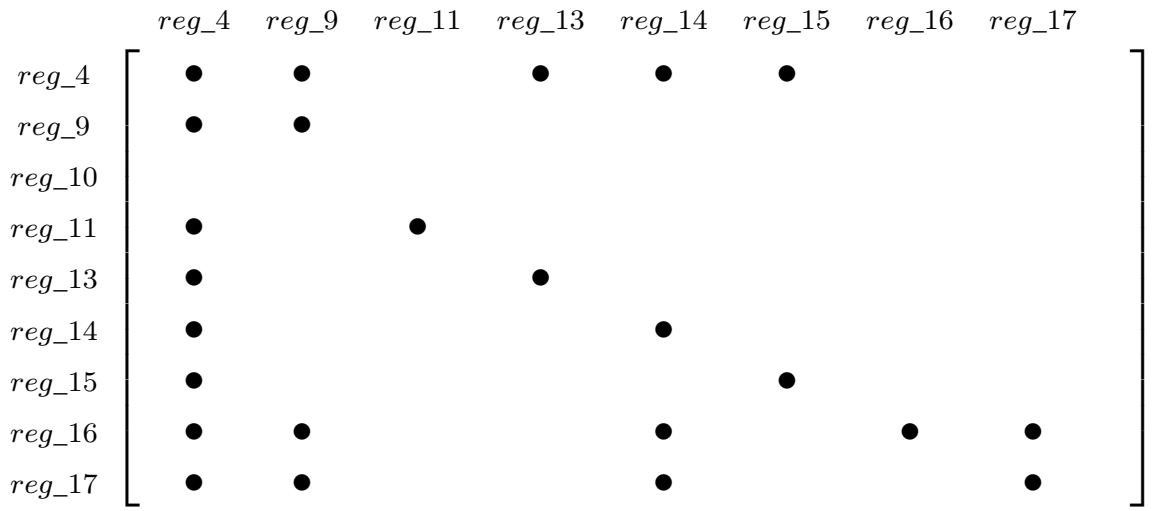


(a) Forme matricielle initiale

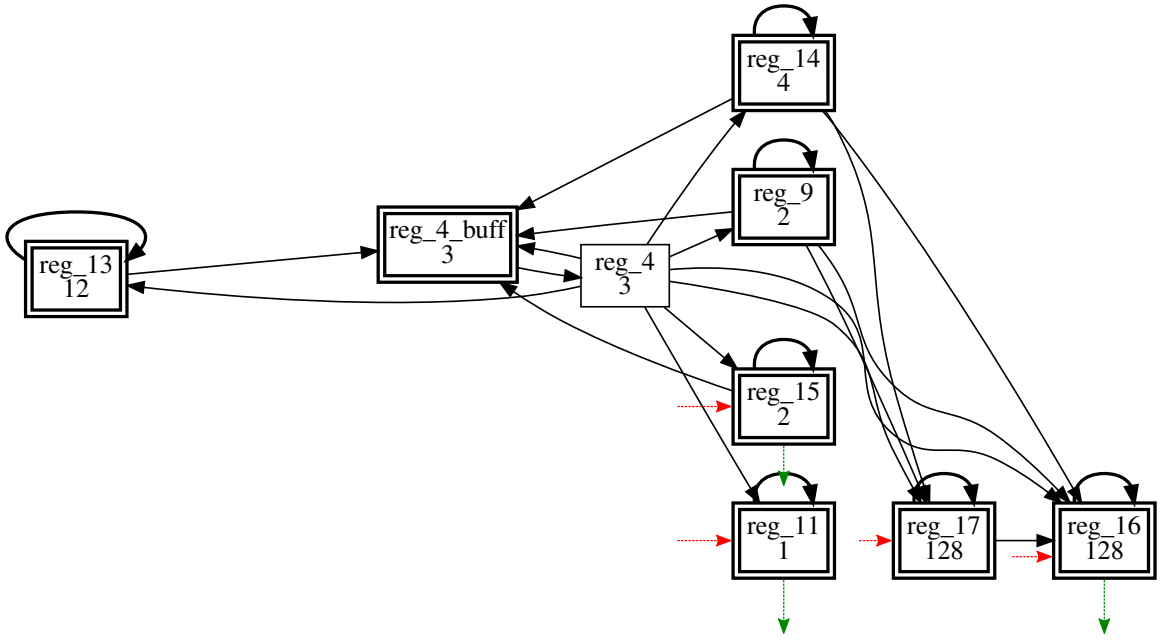


(b) Forme graphique vivace

Figure 6.6: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie DCAR *greedy*

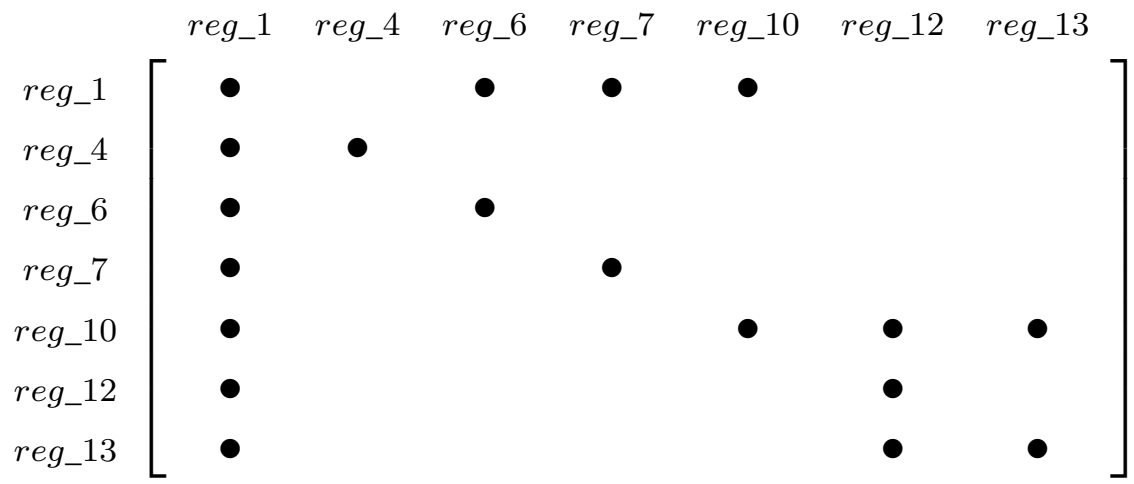


(a) Forme matricielle initiale

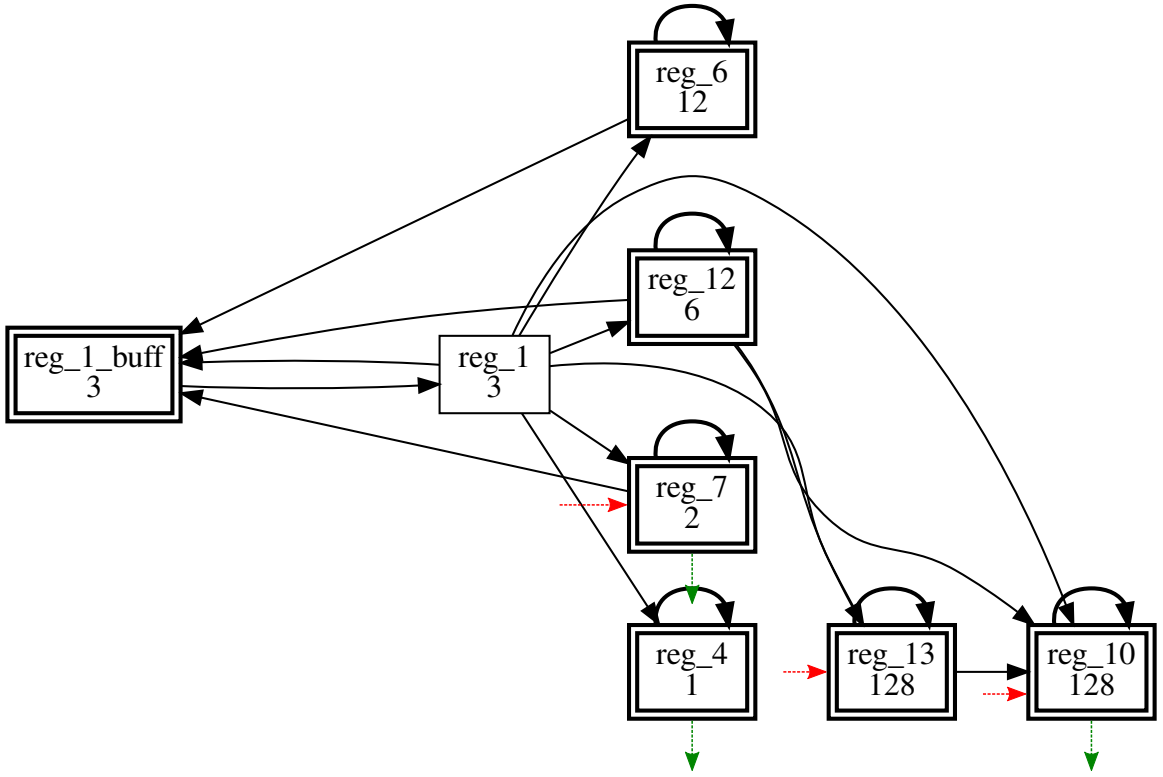


(b) Forme graphique vivace

Figure 6.7: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie CCAR *soft*



(a) Forme matricielle initiale



(b) Forme graphique vivace

Figure 6.8: Réseaux de registres de l'*AES_SC* simplifiés avec la stratégie CCAR *greedy*

Il est à noter que nous obtenons le même résultat en utilisant les stratégies UCAR_soft et DCAR_soft. Par définition, deux registres sont CCAR s'ils sont à la fois CCAR et DCAR. Par conséquent, on obtient un résultat de groupage identique pour la stratégie CCAR_soft et pour les deux stratégies UCAR_soft et DCAR_soft.

Pour ce qui est de la stratégie CCAR_greedy, on calcule les grandeurs suivantes :

— Le nombre de registres : $N_{reg}(\textit{AES\_SC}_{DCAR\_greedy}) = 7 + 1 = 8$

— Le coefficient de linéarité : $C_{lin}(\textit{AES\_SC}_{DCAR\_greedy}) = 2.9$

— Coût de *bufferisation* $Cost_{buff}(\textit{AES\_SC}_{DCAR\_greedy}) = 3 * 31 \approx 90\ u.a.$

— Coût du contrôleur $Cost_{ctrl}(\textit{AES\_SC}_{DCAR\_greedy}) \approx 591\ u.a$

La stratégie CCAR a pour avantage de créer moins de nouvelles dépendances de données dans le RN. Dans ce cas, l'effet de groupage est moins fort que dans les cas précedents.

6.1.2.4 Comparaison des différentes stratégies de groupage de registres

Dans le Tableau 6.1 sont référencées les différentes valeurs calculées pour les différentes stratégies de groupage.

		soft	**greedy**		
	STD	**(U/D/C)CAR**	**UCAR**	**DCAR**	**CCAR**
N_{reg}	14	9	7	5	8
C_{lin}	5.6	3.1	3	2.8	2.9
$Cost_{buff}$	6000	90	90	90	90
$Cost_{ctrl}$	2400	750	530	310	590
$Cost_{tot.}$	**8400**	**840**	**620**	**400**	**680**

Tableau 6.1: Récapitulatif des grandeurs caractéristiques des RNs en fonction des différentes stratégies de groupage

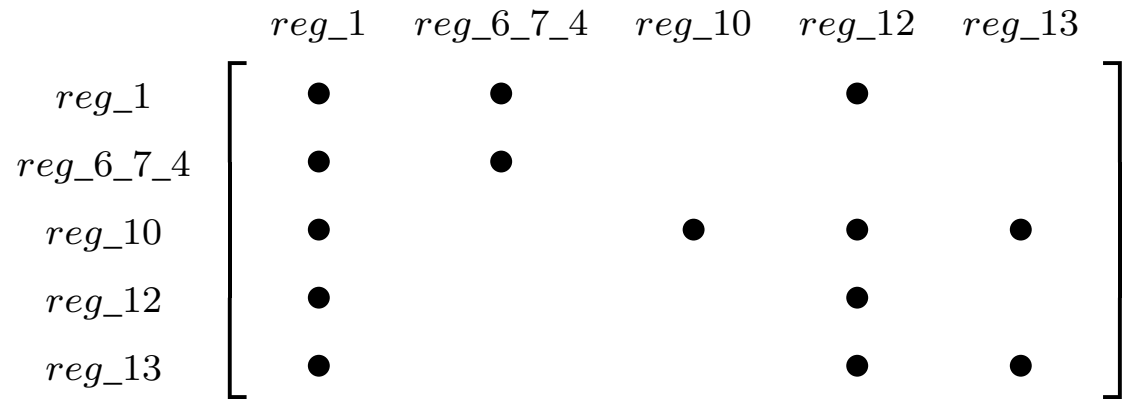
Le premier point remarquable est que, peu importe la stratégie de groupage de registres choisie, le coût de la *bufferisation* est très largement limité par rapport à une approche standard (-98.5%). En effet, le circuit de base comportant une grande quantité de boucles, le groupage de registres est particulièrement efficace pour éviter l'ajout de nombreux registres buffers.

Ensuite, nous pouvons remarquer que l'on réduit grandement le coefficient C_{lin} en utilisant n'importe laquelle des stratégies des groupages. Cette réduction de la complexité se retrouve dans le coût du contrôleur qui réduit également de manière significative par rapport à l'approche standard (entre -75% et -90%).

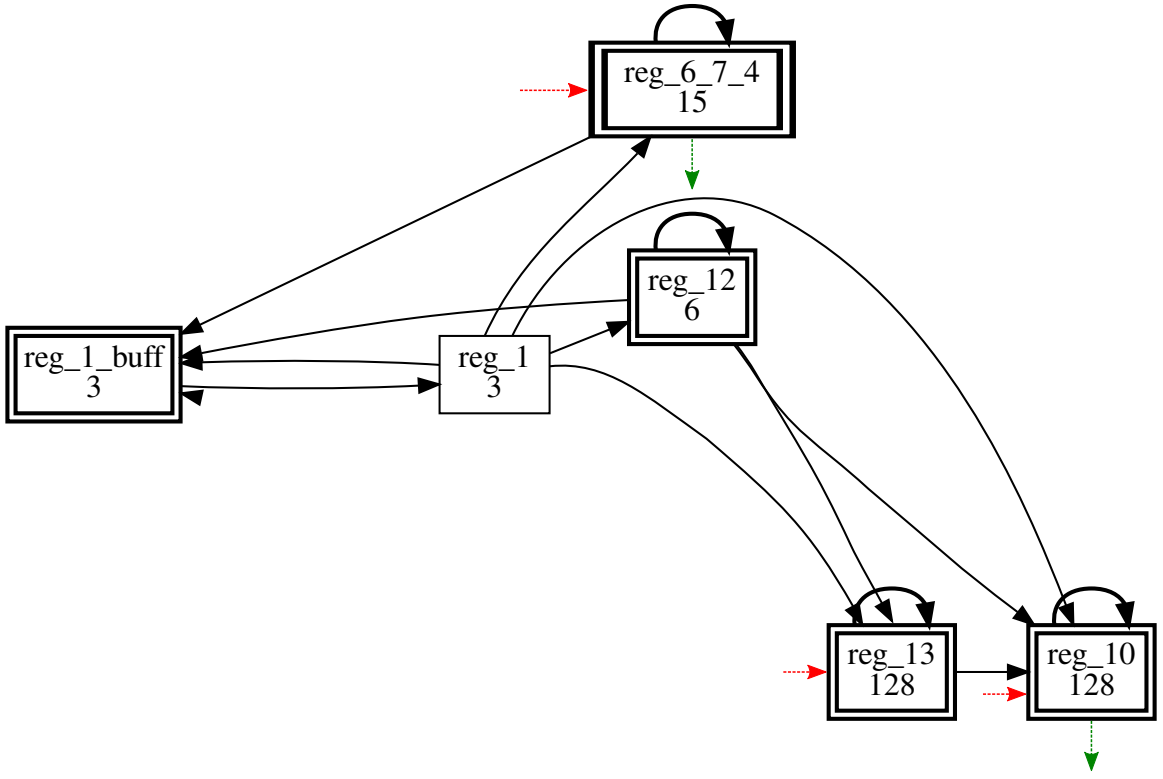
6.1.2.5 Approche CAR implémentée dans le circuit de test

Les différentes variantes offrant des solutions proches les unes des autres pour notre AES, le choix dépend ensuite du compromis que le concepteur décide de faire. Dans notre cas, lors de l'élaboration du circuit de test, la procédure n'était pas encore totalement automatisée. Les

regroupements choisis se sont alors plutôt basés sur la connaissance du circuit et des relations entre les données. Pour rappel, nous avons précisé, dans le Chapitre 4, que nous pouvons procéder à n'importe quel groupage du moment que les dépendances existantes sont conservées. Le groupage finalement choisi est équivalent à celui de la stratégie CCAR_greedy où nous avons regroupé en plus les registres reg_6, reg_7 et reg_4. Le RN associé à ce circuit est celui de la Figure 6.9.



(a) Forme matricielle initiale



(b) Forme graphique vivace

Figure 6.9: Réseaux de registres de l'*AES_SC* embarqué dans le circuit de test

Ses différentes caractéristiques sont :

— Le nombre de registres : $N_{reg}(AES\_SC_{testchip\_car}) = 5 + 1 = 6$

— Le coefficient de linéarité : $C_{lin}(AES\_SC_{testchip}) = 3$

— Coût de *bufferisation* $Cost_{buff}(\textit{AES\_SC}_{testchip\_car}) = 3 * 31 \approx 90\ u.a.$
— Coût du contrôleur $Cost_{ctrl}(\textit{AES\_SC}_{testchip\_car}) \approx 430\ u.a$

Il est à noter que dans cette approche, nous avons fait le choix de ne pas grouper les deux registres les plus grands. En effet, regrouper un trop grand nombre de bascules ensemble aurait pour effet d'être très proche d'un circuit synchrone et l'intérêt d'un tel circuit nous paraissait alors discutable.

Le RN finalement choisi, le contrôleur asynchrone global peut être généré et ensuite remplacer l'arbre d'horloge dans le circuit synchrone initial, de la même manière que celle décrite dans le Chapitre 4.

6.1.3 Désynchronisation CAR unrolled d'*AES_SC*

Un point à remarquer sur le réseau de registres du circuit étudié est que tous les registres sont dans une *self-loop*, *i.e.* une boucle ne sur eux-même. En effet, si on regarde la diagonale du réseau de registres STD, qui correspond à la présence d'une *self-loops*, alors on remarque que tous les registres sont concernés.

Dans les circuits synchrones, il est en effet fréquent de considérer la valeur courante pour le calcul de la valeur suivante, comme par exemple dans un compteur ou une machine d'états finis. Or ces dépendances sont sources d'ajout de matériel puisque chaque *self-loop* nécessite d'ajouter aux connexions existantes :

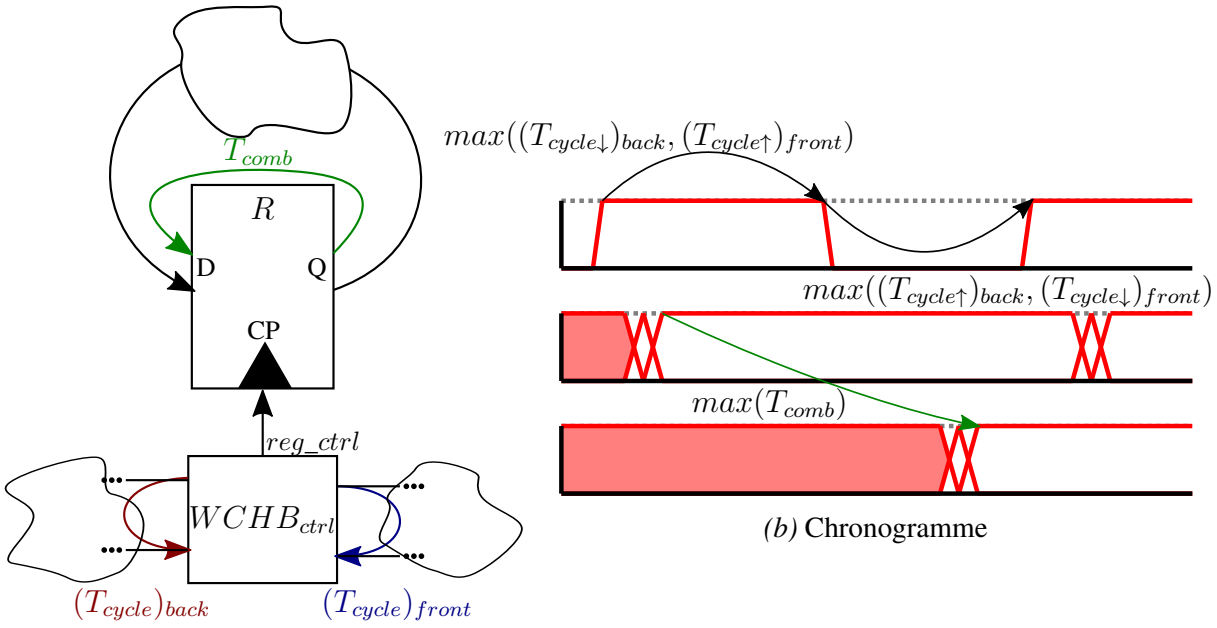
— Un canal de sortie, ajout d'un canal dans la fourche (*fork*) de sortie.
— Un contrôleur de registre *bubble*, pour respecter la Règle 1 (Règle de capacité minimale des boucles).
— Un canal d'entrée, ajout d'un canal dans la jonction (*join*) d'entrée.

On peut donc estimer le surcoût en surface par *self-loop* à $14 + 17 + 14 = 45\ u.a.$. Il est à noter que cette estimation est surévaluée de $14\ u.a.$ dans le cas où le registre est sa seule source, ou sa seule destination.

L'implémentation CAR *Unrolled*, revient à supprimer ces *self-loops*. Le RN considéré pour cette simplification est le même que celui choisi précédemment.

Afin d'assurer un bon fonctionnement du circuit, il faut néanmoins vérifier une contrainte temporelle supplémentaire. Cette dernière vise à vérifier la contrainte de *setup* d'un registre sur lui-même. En effet, la dépendance n'étant plus assurée par un canal direct, il est nécessaire de l'assurer en étudiant le contrôleur et la propagation des signaux au travers de celui-ci.

Sur la Figure 6.10, est présentée une situation de *self-loop* « déroulée ». Dans cette dernière, $\uparrow$ et $\downarrow$ représentent le sens de variation du signal d'arrivée. En reprenant la démarche adoptée



(a) Fragment de circuit

(b) Chronogramme

Figure 6.10: Situation de transfert de donnée dans un fragment de circuit désynchronisé *unrolled*

dans le Chapitre 5, on obtient l'Équation (6.1).

$$max(T_{comb}) < \begin{pmatrix} & max((T_{cycle\downarrow})_{back}, (T_{cycle\uparrow})_{front}) \\ + & max((T_{cycle\uparrow})_{back}, (T_{cycle\downarrow})_{front}) \end{pmatrix} \tag{6.1}$$

Dans le Chapitre 5, nous avons étudié la contrainte de *width* et les manières d'assurer cette contrainte. Le cas du *setup* pour les circuits désynchronisés *unrolled* est très similaire.

Maintenant que nous nous sommes intéressés à ces circuits et comment assurer leur fonctionnement, penchons-nous sur le circuit que nous avons embarqué dans le circuit de test.

Le RN de base est celui obtenu précédemment avec la méthode CAR utilisée pour le circuit de test (Figure 6.9). Les *self-loops* ont été enlevées et on obtient le RN de la Figure 6.11.

La suppression des boucles unitaires, permet d'obtenir :

— $Cost_{ctrl}(AES\_SC_{testchip\_car\_unr}) \approx 300\ u.a.$.

En plus de la réduction de taille du contrôleur, elle supprime la génération des chaînes de délais associées au *self-loops*.

6.1.4 Désynchronisation *refined* d'*AES_SC*

Pour avoir un point de comparaison avec une approche utilisant des structures de sélection, une version désynchronisée, de l'*AES_SC*, les utilisant à été développée et intégrée au *testchip*.

L'intérêt de ces structures est de diriger les données dans le circuit pour n'activer que les

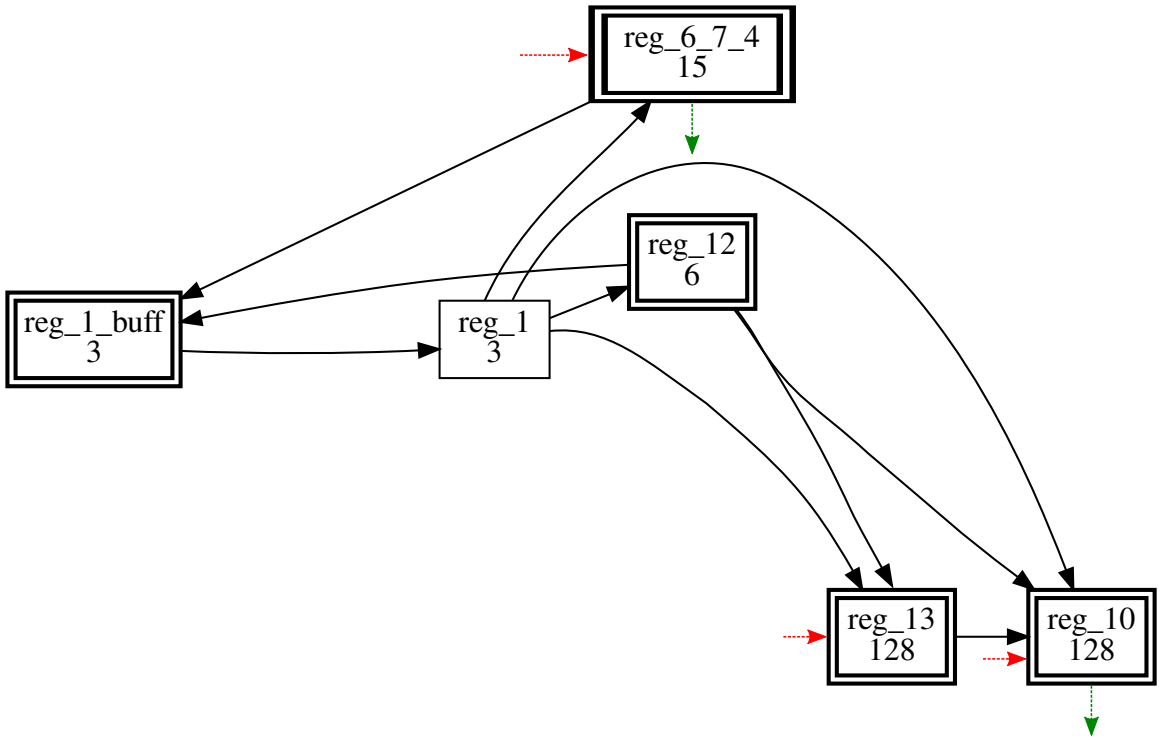


Figure 6.11: Réseau de registres de l'*AES_SC unrolled* utilisé pour le circuit de test

parties du circuit qui ont un traitement à effectuer.

N'ayant pas abouti à une méthode générale permettant de déduire les contrôleurs pour ces circuits, nous n'allons pas entrer dans les détails de la conception de ces derniers.

Cependant, intéressons-nous à ses grandeurs caractéristiques :

— Le nombre de registres : $N_{reg}(\textit{AES\_SC}_{refined}) = 14 + 7 = 21$

— Coût de *bufferisation* $Cost_{buff}(\textit{AES\_SC}_{refined}) = 3 * 31 \approx 6600\ u.a.$

— Coût du contrôleur $Cost_{ctrl}(\textit{AES\_SC}_{refined}) \approx 5000\ u.a$

Il est à noter que cette approche s'avère encore plus conséquente en termes de surface. En effet, les registres à *bufferiser* sont similaires à ceux de l'approche standard donc le coût de *bufferisation* est également proche. Le coût du contrôleur est lui bien supérieur à celui de l'approche standard, ce qui est dû à la taille des structures de choix. Néanmoins, le surcoût en surface devrait être compensé, au niveau de la consommation, par la réduction d'activité apportée par ces mêmes structures.

6.1.5 *AES_SC* contrôlé par un *Ring Oscillator*

La dernière implémentation embarquée dans le circuit de test, basée sur l'*AES_SC*, est une implémentation contrôlée par un oscillateur en anneau, ou *Ring Oscillator* (ROC) comme celui présenté dans la Figure 6.12.

Le circuit de base est l'*AES_SC* synchrone sans *clock-gating* dans lequel nous avons remplacé l'entrée d'horloge externe par un circuit oscillant. L'horloge, dans ce cas interne, sera va-

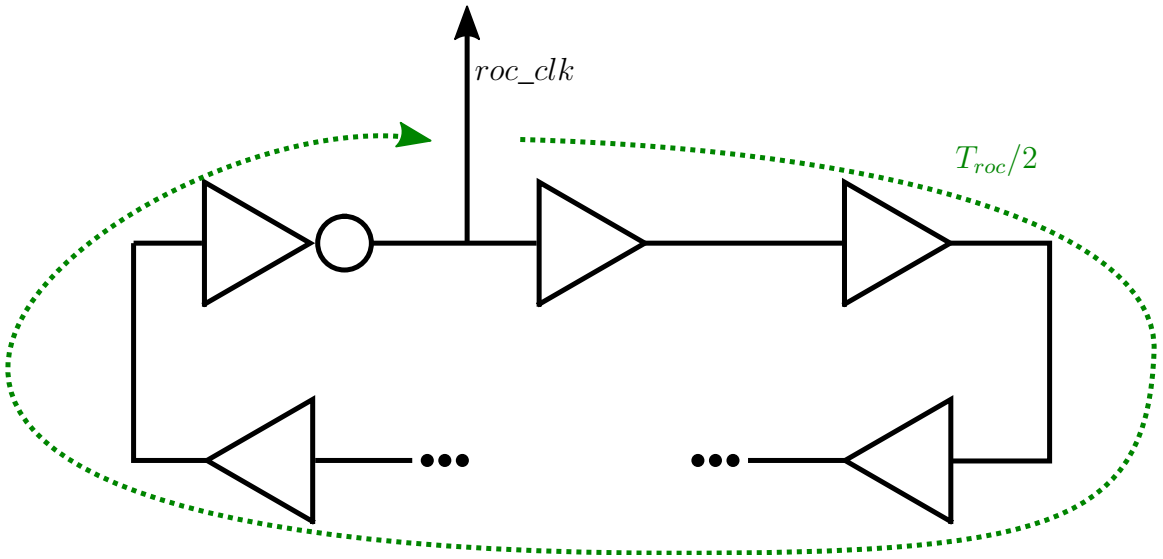


Figure 6.12: Oscillateur en anneau utilisé dans le circuit de test

riable en fonction des conditions d'opération et permettra de simplifier les comparaisons entre un circuit désynchronisé et un circuit synchrone.

Pour ce circuit, la contrainte à respecter est la même que dans un circuit synchrone, *i.e.* la période doit être plus grande que le chemin critique. Afin de s'approcher des conditions observables dans un circuit synchrone, nous avons choisi de faire en sorte que $(T_{clk})_{synchrone} = (T_{clk})_{roc}$ en générant les délais nécessaires dans le circuit.

6.2 Obtention des circuits désynchronisés

6.2.1 Synthèse des circuits désynchronisés

L'étape du flot suivant les choix d'architectures est la synthèse des circuits. Le Tableau 6.2 présente l'impact de la désynchronisation pour les différentes implémentations d'*AES_SC* dans le circuit de test après synthèse en nombre de portes.

Le premier point remarquable sur ces résultats est le fait que l'approche standard engendre un *overhead* en surface conséquent. De la même manière, pour l'implémentation *Refined* on trouve également une surface significativement plus grande. Nous avions pu pressentir ces effets plus tôt dans le flot en estimant les différents coûts.

Un autre point est la réduction de la surface, concernant les cellules séquentielles, dans les implémentations CAR, CAR *Unrolled* et ROC. Cela est dû à l'ajout de cellules de *clock-gating*, dans l'implémentation synchrone.

Concernant la surface totale de ces 3 dernières implémentations, on remarque une augmentation de l'ordre de 10% par rapport à une implémentation synchrone. En considérant les estimations effectuées auparavant, l'augmentation de taille du circuit devrait être plus restreinte

Nombre de portes

	Sync.	**STD**	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Sequentiel	2660	4840	2580	2550	4360	2491
Combinatoire	8720	11140	9990	9970	11780	9840
Contrôleur	N/A	2160	540	410	2620	10
Total	11380	16810	12750	12638	17090	12328

Relativement à l'implémentation synchrone

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Sequentiel ($\Delta\%$)	+82%	-3%	-4%	+64%	-6%
Combinatoire ($\Delta\%$)	+28%	+15%	+14%	+35%	+13%
Contrôleur($\%$ du Total)	19%	5%	4%	23%	0%
Total ($\Delta\%$)	+48%	+12%	+11%	+50%	+8%

Tableau 6.2: Résultat de synthèse en nombre de portes, et relativement à l'implémentation synchrone, des différentes implémentations d'*AES_SC* dans le circuit de test

que ce que nous observons. Néanmoins, nous n'avons pas tenu compte de la taille des interfaces de communication entre les mondes synchrones et asynchrones dans nos évaluations. Notre implémentation d'AES étant compacte, l'impact des mécanismes de synchronisation est ici significatif sur la surface.

Il est à noter qu'à cette étape, aucune contrainte de temps n'a été appliquée aux circuits désynchronisés. L'outil a alors tendance à augmenter la taille des circuits pour optimiser d'autres aspects comme la consommation.

6.2.2 Synthèse des délais

Dans le Chapitre 5, nous avons vu comment vérifier les circuits désynchronisés et générer les délais nécessaires au bon fonctionnement de ces derniers. Il s'est avéré compliqué d'exercer le flot de génération de délais pour tous les *corners* PVT habituellement pris en compte pour un produit. Pour ce *testchip*, nous avons exercé ce flot uniquement dans le *corner* typique. La STA a, quant à elle, été effectuée dans tous les *corners*.

La méthode employée fut la méthode statique utilisant le mécanisme de génération d'arbres d'horloge (Section 5.3.2.2).

Le Tableau 6.3 référence les surfaces des différentes implémentations d'*AES_SC* en sortie de flot de placement routage. Il s'agit donc des circuits fonctionnels tels qu'ils ont été implémentés sur silicium.

Nous pouvons remarquer que les étapes de placement routage influent significativement sur la surface des contrôleurs. En effet, la génération de délais demande d'ajouter potentiellement un grand nombre de cellules.

Pour les désynchronisations standard et *refined*, l'impact de la génération de délais est de

Impact du placement routage sur la surface obtenue en synthèse

	Sync.	**STD**	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Contrôleur ($\Delta\%$)	N/A	+60%	+37%	+50%	+21%	+230%
Total ($\Delta\%$)	+2%	+8%	+4%	+2%	+7%	+0%

Impact final de la désynchronisation relativement à l'implémentation synchrone

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Contrôleur($\%$ du Total)	27%	6%	5%	30%	0%
Total ($\Delta\%$)	+57%	+14%	+11%	+58%	+6%

Tableau 6.3: Impact du placement routage sur les surfaces obtenues en synthèse des différentes implémentations d'*AES_SC* dans le circuit de test

presque 10% sur des circuits déjà de tailles plus conséquentes que les autres. La complexité des contrôleurs de ces circuits demande en effet la génération de beaucoup de délais. De plus, comme nous l'avions évoqué dans le Chapitre 5, la génération de délais n'étant pas optimale, nous nous sommes assurés d'avoir une marge suffisante pour assurer le bon fonctionnement des circuits après fabrication.

Pour les contrôleurs plus compacts, pour les implémentations CAR et CAR *Unrolled*, l'augmentation de la taille du contrôleur est aussi significative par rapport au circuit obtenu à l'issue de la synthèse. Cependant, ayant des contrôleurs de taille restreinte à la base permet de conserver un circuit à l'issue des étapes de placement routage à un niveau proche du cas synchrone.

Enfin, concernant le circuit à base d'oscillateur en anneau, la taille de cet oscillateur augmente beaucoup pendant les phases de PnR mais reste très faible par rapport à la taille du circuit.

Globalement, l'impact en surface de la désynchronisation, sur ce circuit, est plus restreint en utilisant la méthode CAR qu'en utilisant une désynchronisation standard ou *Refined*. Toutefois, l'impact de la désynchronisation, en termes de surface, n'est pas marginal dans les meilleurs cas et représente au final un plus grand impact que l'utilisation d'un arbre d'horloge.

Il est à noter que le circuit synchrone considéré comme base de travail est initialement compact. Par conséquent, bien que l'on cherche à limiter l'impact de la désynchronisation, même avec des contrôleurs de taille restreinte, leur taille n'est pas négligeable par rapport au circuit de base. De plus, l'arbre d'horloge généré pour la faible quantité de bascules présentes dans le circuit synchrone est simple. L'arbre d'horloge est donc de petite taille et l'impact de la désynchronisation s'en fait d'autant plus ressentir.

6.2.3 STA des circuits désynchronisés

L'impact sur la surface de l'insertion des chaînes de délais s'explique également par la marge prise pendant la génération de ceux-ci. En effet, ne contrôlant pas totalement comment l'outil

se comporte, nous avons préféré assurer le bon fonctionnement des circuits après fabrication.

N.B. : *Nous prendrons en référence, dans les résultats présentés dans la suite, les contraintes de setup appliquées à chacun des chemins, i.e. les temps de traversée des chemins critiques locaux. Un écart positif, entre la contrainte et le délai effectivement mesuré, indique alors une marge par rapport à la contrainte à remplir. Un écart négatif indique lui une violation de la contrainte de setup.*

Le Tableau 6.4 présente les grandeurs statistiques sur les délais obtenus dans le *corner* PVT typique.

Écarts à la contrainte en nanosecondes

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Minimum (ns)	0.3	0.7	0.7	0.6	13
Moyenne (ns)	6	7.1	8.2	8.9	13
Médiane (ns)	4.6	8.6	7.6	9.9	13
Maximum (ns)	20	11	20	19	13

Écarts à la contrainte en pourcentage

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Minimum ($\Delta\%$)	+11%	+13%	+27%	+15%	+77%
Moyenne ($\Delta\%$)	+32%	+41%	+44%	+52%	+77%
Médiane ($\Delta\%$)	+26%	+41%	+40%	+55%	+77%
Maximum ($\Delta\%$)	+81%	+74%	+82%	+84%	+77%

Tableau 6.4: Statistique sur les marges de délais obtenues après placement routage en *corner* typique

Le premier point remarquable dans cette table est que est les contraintes minimales atteintes semblent, en valeur absolue, assez proches de la cible. Cela indique que les délais ont été générés pour ces chemins de manière correcte. Cependant, en valeur relative à la contrainte, nous pouvons remarquer que les écarts à la contrainte sont significatifs même dans le minimum des cas ($\approx 10\%$).

Ensuite, concernant les écarts maximaux aux contraintes, on remarque cette fois de forts écarts existants. En effet, un écart de $20ns$ est significatif étant donné que son homologue synchrone a lui comme contrainte une période d'horloge de $33ns$.

Comme nous l'avons vu dans le Chapitre 5, chaque chemin entre contrôleur de registre offre la possibilité d'arranger les délais de telle manière que chaque chemin dispose exactement du délai qu'il lui faut. Cependant, les outils ne sont, pour l'heure, pas capables d'effectuer d'automatiquement ces opérations de *retiming* [44].

Enfin, concernant l'oscillateur en anneau, le délai généré est cohérent avec la contrainte synchrone équivalente (environ une demi période d'horloge). Il est à noter que la contrainte synchrone est cependant largement supérieure aux possibilités du circuit dans le cas typique. En effet, une marge de presque 80% sur le ROC signifie que le circuit synchrone, qui lui est très

similaire, devrait pouvoir fonctionner à une fréquence beaucoup plus haute que celle attendue. Cela est dû au fait qu'habituellement, la fréquence limite de fonctionnement est fixée pour un *corner* PVT lent.

Pour compléter l'étude, intéressons-nous maintenant au la Tableau 6.6 dans laquelle sont reportées les mêmes grandeurs que précédemment mais dans un *corner* de PVT dit rapide, *i.e.* avec des cellules rapides, une tension d'alimentation haute et une température basse.

Écarts à la contrainte en nanosecondes

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Minimum (ns)	0.1	0.3	0.5	0.2	11
Moyenne (ns)	4	5	6	6	11
Médiane (ns)	4	6	7	6	11
Maximum (ns)	12	8	12	11	11

Écarts à la contrainte en pourcentage

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Minimum ($\Delta\%$)	+10%	+15%	+35%	+12%	+131%
Moyenne ($\Delta\%$)	+42%	+49%	+58%	+58%	+131%
Médiane ($\Delta\%$)	+38%	+54%	+56%	+64%	+131%
Maximum ($\Delta\%$)	+86%	+79%	+86%	+87%	+131%

Tableau 6.5: Statistique sur les marges de délais obtenues après placement routage en *corner* rapide

On remarque en comparant avec les valeurs du *corner* typique que les écarts à la contrainte sont, en valeur temporelle, plus faibles tout en conservant un écart relatif proche. Nous pouvons donc considérer qu'entre les *corners* typiques et rapides, les cellules standards ont un comportement relativement linéaire.

Pour le *corner* PVT lent (cellules lentes, tension d'alimentation basse et température haute) en revanche, on obtient de nombreuses violations de *timing* pour toutes les implémentations. Le Tableau 6.6 présente les violations et les écarts à la contraintes observées dans nos différents circuits.

	STD	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Nombre violations	76	8	8	20	0
Pire violation (ns)	-32	-37	-29	-17.6	N/A
Pire violation ($\Delta\%$)	-62%	-19%	-53%	-36.7%	N/A
Plus grande marge (ns)	+41	+22	+27	+41	+3
Plus grande marge ($\Delta\%$)	+64%	+52%	+59%	+72%	+3%

Tableau 6.6: STA dans le *corner* lent

Nous pouvons remarquer de très forts écarts à la contrainte, aussi bien en violation qu'en marge, dans ce *corner*. Dans cette situation, la génération de délai en utilisant uniquement des

buffers et des inverseurs joue fortement en la défaveur des circuits désynchronisés. En effet, la tension d'alimentation s'approchant de la tension de seuil, les écarts de comportements entre différentes cellules standards sont exacerbés et engendre des violations fortes.

Dans un circuit synchrone, la synthèse et les étapes suivantes sont généralement effectuées dans ces *corners* lents afin de couvrir le pire cas possible. Pour le cas asynchrone, effectuer la génération de délais dans ces *corners* nuirait fortement aux performances. En effet, certains délais nécessaires dans ces *corners* sont très élevés et seraient finalement très supérieur à ce qui est nécessaire dans le cas typique. En outre, même en s'assurant de supprimer toutes les violations de timing dans un *corner* lent, les écarts de comportement entre les délais dans les *corners* lent et typiques pourraient tout de même engendrer des violations de *timing* dans le *corner* typique.

Une solution pour s'affranchir de cette problématique serait de prendre en compte tous les *corners* pendant la génération des délais. Cela aurait cependant un coût certain en temps de calcul, surface et performances.

6.3 Évaluation des performances et de la consommation en simulation

Avant de passer aux mesures physiques, un certain nombre de grandeurs ont été évaluées en simulation après placement routage, donc en tenant compte des différents effets parasites.

6.3.1 Temps d'opération et fréquences équivalentes

L'AES considéré fonctionne suivant la séquence suivante :

— La donnée à chiffrer est écrite à l'adresse de la donnée
— La clé à utiliser pour le chiffrement est écrite à l'adresse de la clé
— Le lancement du chiffrement est effectué par l'écriture dans le registre de contrôle
— Tant que l'opération est en cours, un signal *'aes_is_running'* est haut
— Une fois que l'opération est terminée, la donnée de sortie peut être lue

Pour vérifier que l'*AES_SC* fonctionne, une opération est lancée avec une donnée et une clé connue, puis le résultat lu est comparé avec celui attendu.

Sur la Figure 6.13 sont présentés les temps d'opération relevés pour les différentes implémentations de l'*AES_SC* dans le *testchip*.

N.B. : *les corners considérés en simulation sont des corners uniquement en procédé typique et avec une température de 25°C.*

Il est à noter que les temps relevés pour l'implémentation synchrone sont les temps obtenus pour une fréquence maximale d'opération. Pour les circuits désynchronisés, la fréquence

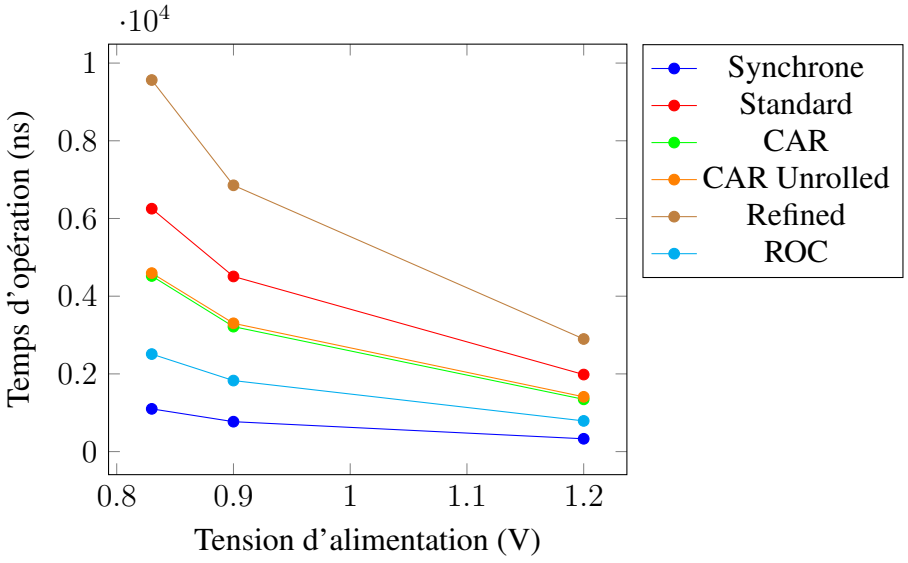


Figure 6.13: Temps d'opération des implémentations d'*AES_SC* en fonction de la tension d'alimentation en simulation

n'ayant pas d'influence sur leur comportement, le temps d'opération est toujours le plus rapide atteignable pour les conditions d'opération considérées.

Comme nous pouvions nous y attendre, après les résultats relevés en *Static Timing Analysis*, les temps d'opération pour les circuits désynchronisés sont plus élevés que pour leur homologue synchrone, et ce pour toutes les tensions d'alimentation observées.

L'interprétation des temps d'opération n'est pas aisée car leur évolution n'est pas linéaire par rapport à la tension d'alimentation. En outre, les performances synchrones sont généralement linéaires avec la tension d'alimentation. Les circuits désynchronisés devraient donc également voir leurs fréquences équivalentes varier linéairement avec la tension d'alimentation.

Avec T_{op} le temps d'une opération, et sachant qu'une opération d'AES nécessite 55 cycles d'horloges, pour l'*AES_SC*, la fréquence équivalente d'un circuit désynchronisé se déduit de la manière suivante :

$$F_{eq}(AES\_SC) = \frac{55}{T_{op}(AES\_SC)} \tag{6.2}$$

Sur la Figure 6.14 sont présentées les fréquences équivalentes pour chaque tension d'alimentation considérée ainsi que la cible de fréquence synchrone située à 33MHz.

De la même manière qu'en prenant en compte le temps d'opération, on observe une forte chute de performance pour les circuits désynchronisés. Nous pouvons cependant noter que la simplification des contrôleurs avec la méthode CAR a permis de limiter cette perte de performances par rapport à une approche standard.

Nous avions évoqué dans le Chapitre 4 la possibilité de perdre en performances avec le groupage de registres, en raison de la mise en commun des chemins critiques des registres groupés. Il s'avère, dans le cas de ce circuit, que cette perte de performance est compensée par la simplification du réseau de contrôle asynchrone qui permet la réduction du nombre de délais

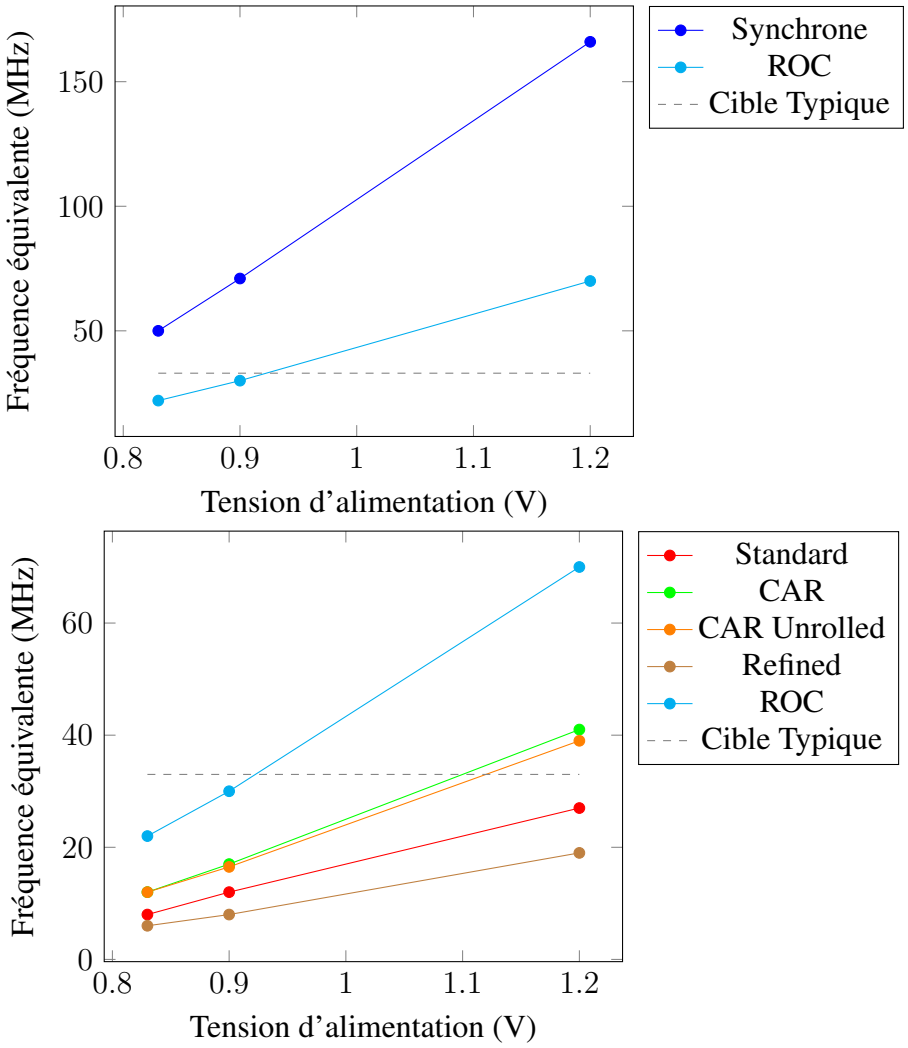


Figure 6.14: Fréquence équivalente des implémentations d'*AES_SC* en fonction de la tension d'alimentation en simulation

à générer et donc une meilleure performance.

Il est à noter que l'impact du « déroulage » des boucles, dans l'implémentation CAR *Unrolled*, a très peu d'impact sur les performances. Cela indique bien que, dans le cas de ce circuit, les boucles unitaires ne sont pas critiques et peuvent donc être supprimées pour économiser de la surface et de la complexité lors de la génération des délais.

Enfin, un autre point pénalise les circuits désynchronisés, la contrainte imposée sur les chemins de données est la même que ce soit pour le circuit synchrone ou le circuit désynchronisé. Cependant, les optimisations apportées par les outils ont tendance à augmenter le temps de traversée des chemins non-critiques pour améliorer d'autres métriques (consommation, surface par exemple). Si cela a du sens pour un circuit synchrone, pour les circuits désynchronisés ce gain potentiel en puissance pourra être en partie perdu par le délai supplémentaire nécessaire à

générer.

6.3.2 Consommation et efficacité énergétique

La dernière grandeur que nous pouvons comparer en simulation est la consommation des circuits.

La Figure 6.15 présente les différentes consommations, des modules seuls, pendant une opération en fonction de la tension d'alimentation.

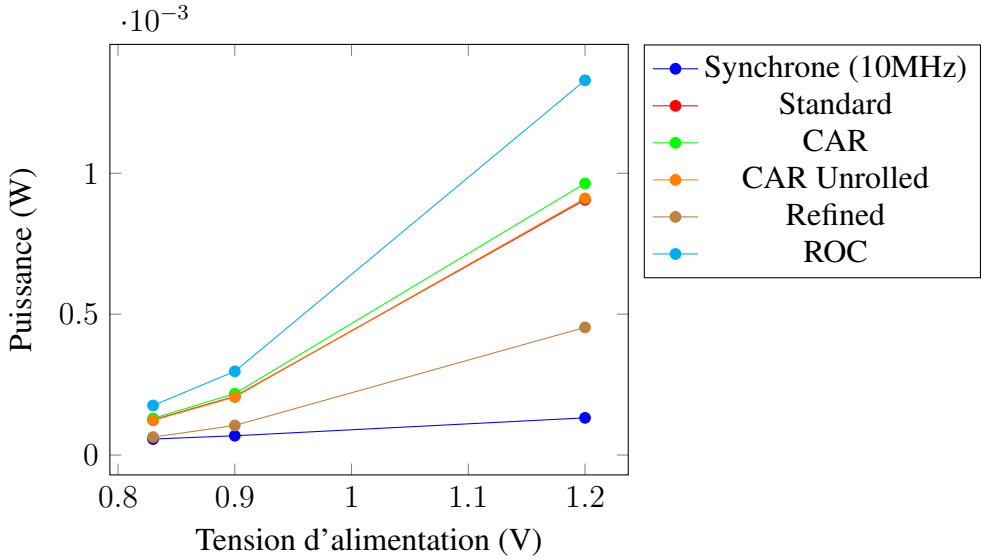


Figure 6.15: Consommation de chaque circuit pendant une opération AES

Il est à noter que dans ce graphe, la consommation du circuit synchrone est fixée pour une fréquence d'opération à 10MHz. Sachant que la puissance consommée par un circuit synchrone est directement proportionnelle à sa fréquence, l'augmentation de la consommation pour les hautes tensions d'alimentation est due aux plus forts courants de fuite.

D'un autre côté, un circuit désynchronisé augmente sa vitesse de calcul avec la tension d'alimentation, donc sa fréquence équivalente. La forte augmentation de la consommation pour les circuits désynchronisés était donc attendue. En outre, les différents circuits ont des performances très différentes, il est donc difficile de les comparer en l'état.

Afin d'avoir des résultats plus représentatifs pour les circuits désynchronisés, nous allons plutôt nous baser sur ce que nous appelons un coefficient de coût énergétique d'un AES, $Cost_{en}(AES)$, suivant l'Équation (6.3).

$$Cost_{en}(AES) = \frac{P(AES)}{F_{eq}(AES)} \tag{6.3}$$

La dimension de ce coefficient est une énergie (J) mais sera plutôt considérée comme un facteur de consommation par performance (W/MHz) dans la suite.

Les valeurs de ce coefficient sont présentées sur la Figure 6.16.

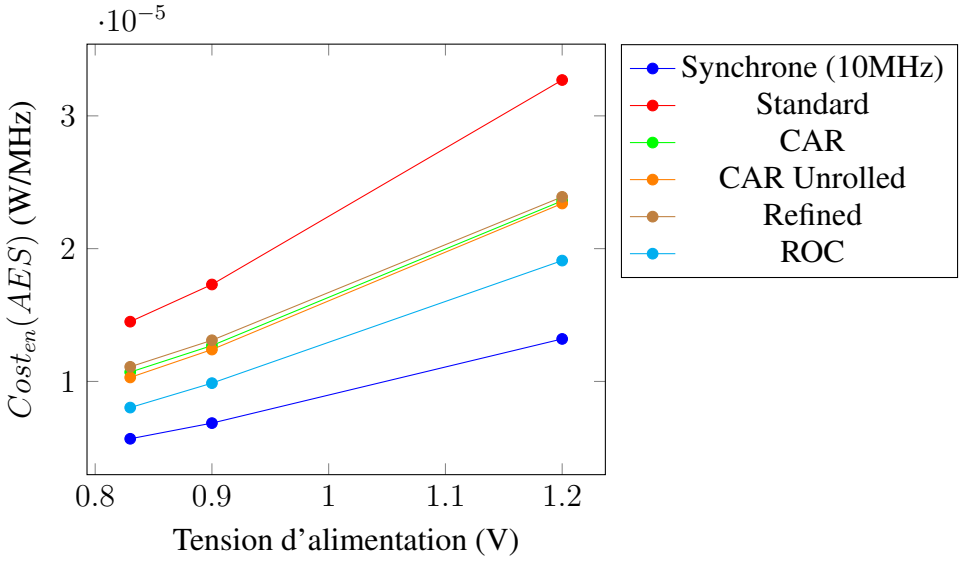


Figure 6.16: Coefficient de coût énergétique pour chaque implémentation d'AES en fonction de la tension d'alimentation

En premier lieu, il est pertinent de remarquer que le coefficient est biaisé pour l'implémentation synchrone. En effet, la fréquence pour ce dernier est fixée à 10MHz, or, pour avoir une comparaison équitable avec les circuits désynchronisés, il faudrait avoir les points pour la fréquence maximale atteignable à chaque tension d'alimentation.

La première différence remarquable est celle entre l'implémentation synchrone et l'implémentation ROC. Ce dernier a en effet un coefficient plus haut que son homologue synchrone. On retrouve très certainement ici l'effet de l'utilisation du *clock-gating* dans le circuit synchrone.

Ensuite, trois implémentations ont des coûts énergétiques très proches : CAR, CAR *Unrolled* et *Refined*. On en déduit que l'utilisation de structures de choix n'est pas pertinente, pour ce circuit, puisque avec les implémentations CAR et CAR *Unrolled*, qui ont de plus faibles impacts en surface et performances, on obtient un même résultat énergétique.

Enfin, la désynchronisation standard est la plus coûteuse énergétiquement mais aussi une des plus coûteuses en surface et en performances, c'est ici le pire cas possible pour la désynchronisation.

Un dernier point à comparer est la part de la consommation due au circuit de contrôle. Ces proportions sont présentées dans le Tableau 6.7.

En regard des valeurs de coût énergétique, on remarque que les surconsommations dues aux contrôleurs sont répercutées directement sur les implémentations sans structure de choix (Standard, CAR, CAR *Unrolled*) par rapport à l'approche ROC. Au contraire, la forte part de consommation due au contrôleur du circuit utilisant l'approche raffinée n'est pas totalement répercutée sur le coût énergétique. Cela confirme bien que les structures de choix permettent

V_{dd}	**Sync.**	**STD**	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
0.81V	<1%	25%	8%	6%	23%	1%
0.90V	<1%	25%	8%	6%	22%	1%
1.20V	<1%	23%	8%	6%	22%	1%

Tableau 6.7: Proportions de la consommation de la partie contrôle de chaque implémentation

d'économiser de l'activité dans le chemin de données.

Il est à noter que la partie « contrôle » du circuit synchrone, *i.e.* l'arbre d'horloge, consomme très peu. Cela nous indique que le circuit considéré a un nombre de bascules très réduit. Par conséquent, la génération de l'arbre d'horloge impacte très peu la surface et la puissance consommée par ce circuit synchrone.

6.4 Caractérisation des circuits fabriqués

Les différentes simulations nous ont déjà permis de tirer un certain nombre de conclusions. Cependant, les points de caractérisations disponibles en simulation sont peu nombreux. La caractérisation physique, en plus d'assurer le fait d'avoir obtenu des circuits fonctionnels, permet de pousser les circuits dans leurs retranchements et donc de mieux les cerner.

Dans le cadre du projet, nous avons pu fabriquer 60 circuits de test. Une caractérisation rapide a pu être effectuée sur la totalité de ces puces et nous a permis de valider le bon fonctionnement de 59 puces au total.

Sur 25 de celles-ci, une caractérisation complète a été effectuée, à l'aide d'un testeur industriel, pour 3 températures : -40°C, 25°C et 125°C. Les résultats présentés dans la suite porteront sur ces 25 pièces car leur environnement de testa été contrôlé.

Enfin, afin de pouvoir mesurer les performances à très basse tension d'alimentation (<0.7V), nous avons dû ajouter un plot de *microprobe* par procédure FIB, pour *Focused Ion Beam*, sur deux pièces.

6.4.1 Caractérisation de robustesse

Température typique

Les tests de robustesse en tension d'alimentation ont été effectués pour toutes les implémentations. On appelle la tension à laquelle un circuit arrête de fonctionner la tension de coupure ($(V_{dd})_{cut}$) et les valeurs sont présentées dans le Tableau 6.8.

Nous pouvons remarquer que les tensions d'alimentation de coupures sont très proches les unes des autres. Pour l'implémentation synchrone, cela n'est pas étonnant étant donné que la fréquence imposée au circuit a été fortement baissée.

Ces tensions de coupure basses sont cependant notables pour les implémentations asyn-

$(V_{dd})_{cut}$	**Sync.**	**STD**	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
Minimum	0.24	0.26	0.22	0.24	0.26	0.24
Moyenne	0.34	0.34	0.36	0.31	0.32	0.30
Maximum	0.48	0.56	0.60	0.48	0.46	0.48

Tableau 6.8: Grandeurs statistiques des tensions d'alimentation minimales garantissant un bon fonctionnement du circuit à 25°C

chrones puisque des réserves fortes avaient été émises vis-à-vis de la génération de délais n'utilisant que des *buffers* et inverseurs. Les résultats sont très satisfaisants étant donné qu'il sont proches de la limite de fonctionnement des cellules standards. Il faut cependant garder à l'esprit que les marges obtenues après la génération de ces délais sont conséquentes et jouent donc certainement en faveur d'une forte robustesse des circuits désynchronisés.

Influence de la température

La Tableau 6.9 présente les valeurs moyenne de $(V_{dd})_{cut}$ obtenues pour les autres températures de caractérisation.

Température	**Sync.**	**STD**	**CAR**	**CAR *Unr.***	**Refined**	**ROC**
-40°C	0.36	0.36	0.38	0.34	0.34	0.33
125°C	0.33	0.35	0.37	0.33	0.33	0.32

Tableau 6.9: Tensions de coupures moyennes relevées pour deux valeurs de température

Par rapport à la température typique, les variations de la valeur moyenne de la tension de coupure restent faibles. On remarque par contre que les basses températures ont tendance à plus dégrader la robustesse, ceci est certainement dû aux ralentissements des chemins de données qui doivent être plus marqués que les ralentissements des chaînes de délais.

6.4.2 Mesures de performances

Température typique

Sur les 25 pièces considérées pour la caractérisation, nous avons également pu mesurer les performances. Dans la Figure 6.17 sont présentées les fréquences équivalentes moyennes obtenues pour les différentes tensions d'alimentation supérieures à 0.7V.

Pour le circuit synchrone, la valeur plafond de 50MHz représente la limite de nos appareils de mesure.

Les résultats observés ont le même comportement global que ceux obtenus en simulation. Sur la valeur moyenne de ces 25 pièces, nous pouvons cependant remarquer que la simulation est optimiste, d'environ 10%, quant aux performances annoncées.

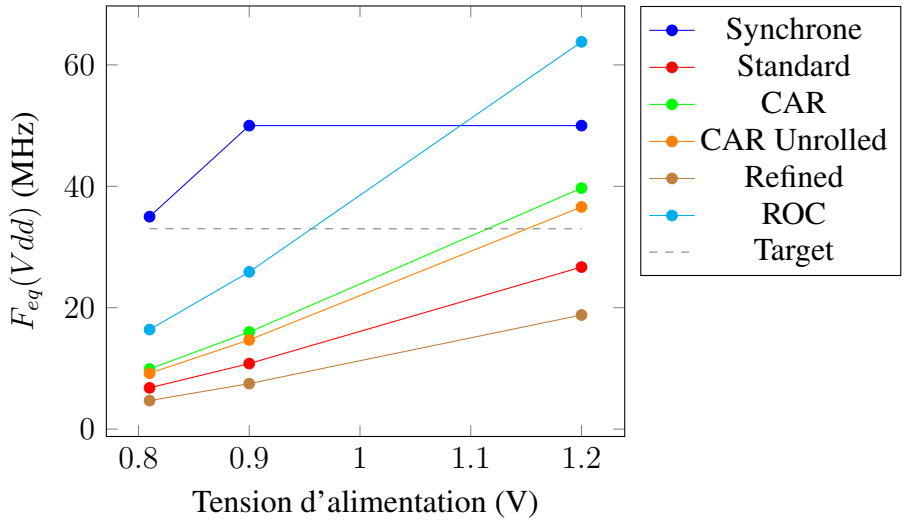


Figure 6.17: Fréquences équivalentes mesurées sur les circuits de tests en fonction de la tension d'alimentation à 25°C

Nous nous sommes également intéressés aux performances en limite de fonctionnement des circuits, ou à faible tension d'alimentation. Les valeurs relevées sur une pièce sont présentées sur la Figure 6.18.

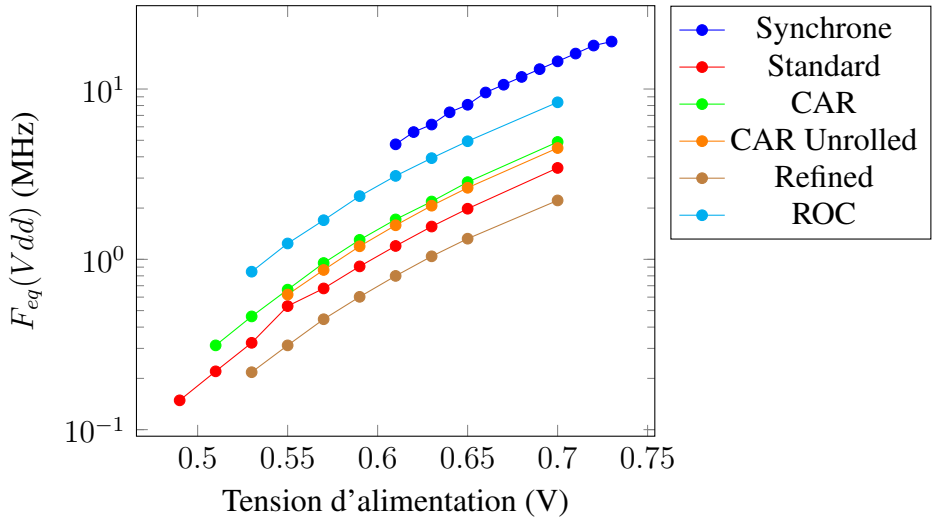


Figure 6.18: Fréquence équivalente à très basse tension d'alimentation à 25°C

Nous avions précédemment un comportement linéaire entre la performance et la fréquence équivalente, ce qui est ce qui est généralement attendu pour les circuits synchrones. À très basse tension d'alimentation cependant, le comportement de la fréquence équivalente se rapproche plus d'une tendance logarithmique. Cela signifie que l'on entre dans les zones de fonctionnement non-linéaires des transistors.

Nous pouvons cependant remarquer que l'ordre des performances est conservé, et ce même pour ces valeurs très faibles de tension d'alimentation et de performances.

Il est néanmoins à noter que la comparaison effectuée n'est pas à l'avantage des implémentations asynchrone. Nous considérons en effet que le synchrone peut réduire sa fréquence de manière transparente. Cependant, modifier la fréquence de fonctionnement d'un circuit est en pratique plus compliqué qu'il ne parait sur les courbes observées.

Influence de la température

La température influant de la même manière sur toutes les implémentations le l'*AES_SC*, nous ne nous intéressons, dans la Figure 6.19, qu'à un seul des circuits désynchronisés : l'implémentation CAR.

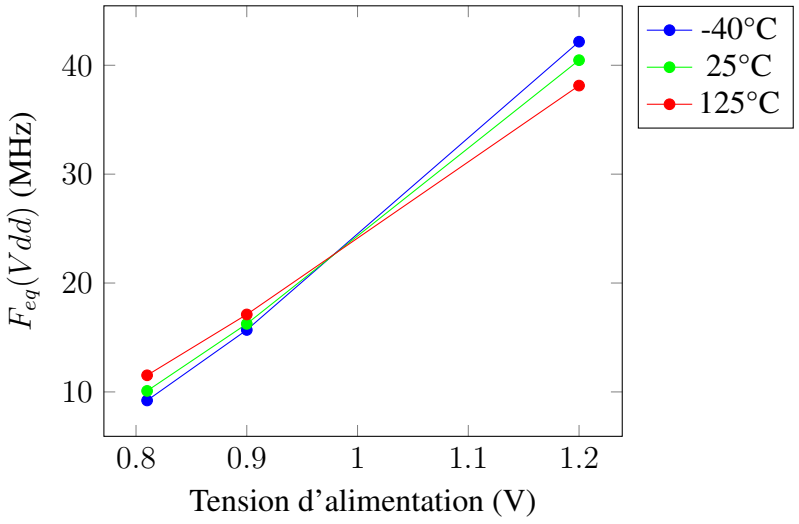


Figure 6.19: Fréquence équivalente mesurée pour chaque température du circuit CAR en fonction de la tension d'alimentation

On observe une inversion de comportement entre les points chauds et froid. Le fait que le circuit auquel on applique une tension d'alimentation élevée avec une haute température soit moins rapide, qu'avec une faible température, vient certainement de l'agitation thermique qui, ajoutée à la forte vitesse des charges dans le circuit, engendre des phénomènes de congestion et donc ralentit la vitesse de commutation des transistors.

Pour les faibles tensions d'alimentation, n'ayant pas d'explication simple, nous renvoyons le lecteur vers les travaux de Mauricio ALTIERI SCARPATO [2] qui portent sur cette problématique.

6.4.3 Mesures de consommation

Température typique

En ce qui concerne la consommation des circuits, la mesure n'est pas aussi évidente qu'en simulation. En effet, en simulation, nous pouvons aisément isoler le circuit pertinent. Cependant, étant donné que rien n'a été prévu, pendant la conception, pour isoler chaque circuit nous mesurons la totalité du circuit.

Néanmoins, pour les circuits désynchronisés, un mécanisme à été mis en place afin de les arrêter lorsqu'ils ne sont pas sollicités : ils ne consomment de l'énergie que lorsqu'ils sont actifs (sauf pour ce qui est de la consommation de fuite). Ne pouvant se baser sur la consommation seule de chacun des circuits, nous allons donc nous intéresser, sur la Figure 6.20, à la surconsommation engendrée par une opération pour chaque implémentation de l'*AES_SC*.

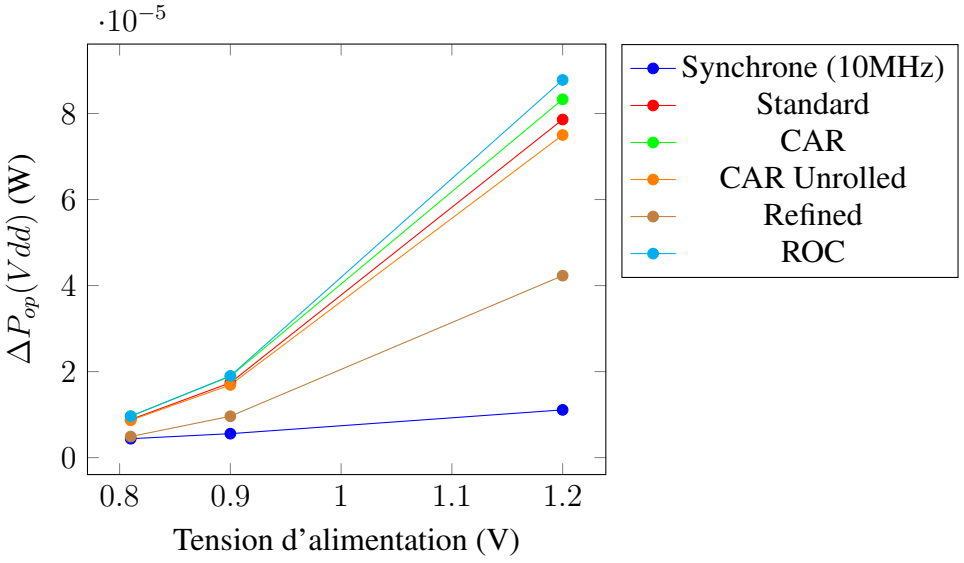


Figure 6.20: Surconsommation moyenne engendrée par une opération pour chaque implémentation de l'*AES_SC* à 25°C

Nous pouvons noter que la tendance est la même que celle observée en simulation. Cependant, ces mesures sont difficilement interprétables vu que les performances des différentes implémentations sont très différentes les unes des autres.

La Figure 6.21, présente ces valeurs normalisées par rapport aux performances moyennes associées (le coefficient $Cost_{en}(AES)$).

Il est à noter que le comportement relatif de circuit à circuit est bien retranscrit.

Les détails ne seront pas présentés dans ce manuscrit, mais en corrélant les différentes simulations et mesures physiques effectuées, les écarts observés pour la consommation sont généralement autour de 10% et ont tendance à se réduire pour les fortes tensions d'alimentation.

Influence de la température

La Figure 6.22 présente l'influence la température sur la consommation pendant une opération.

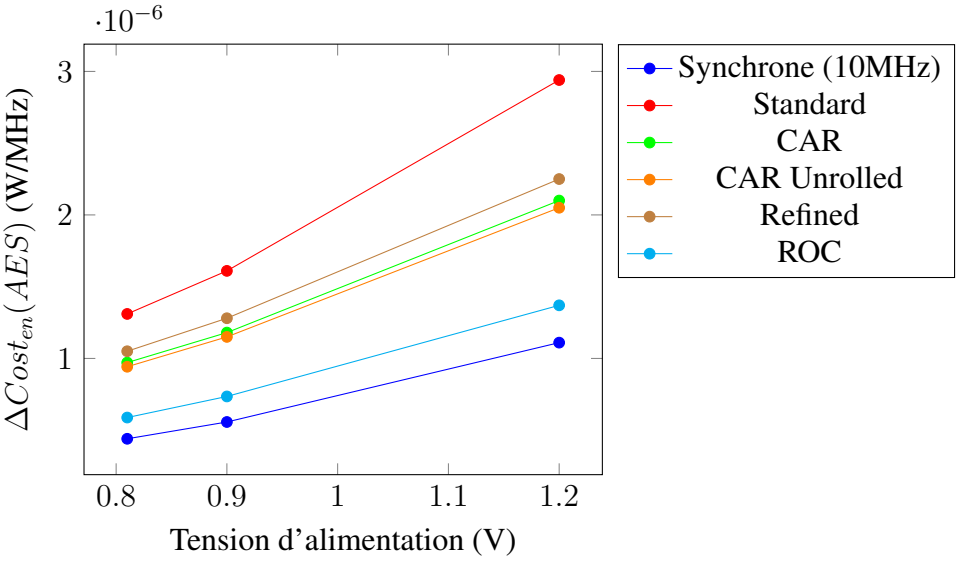


Figure 6.21: Surconsommation normalisée moyenne engendrée par une opération pour chaque implémentation de l'*AES_SC* à 25°C

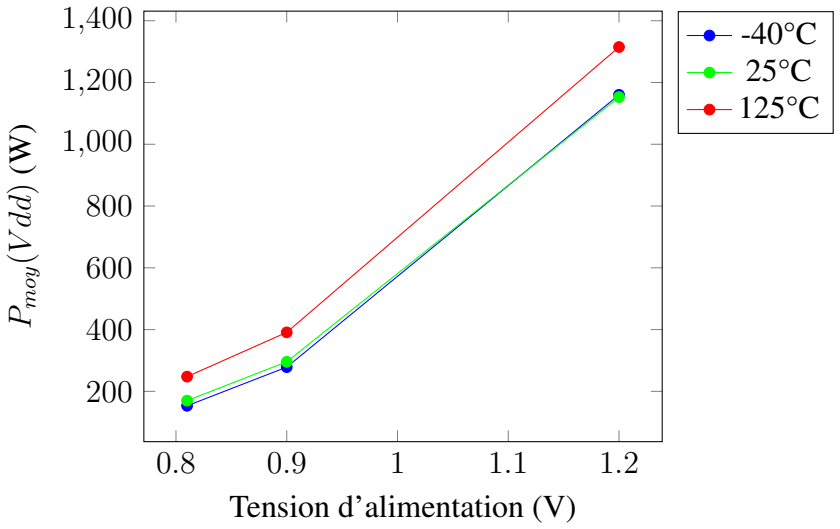


Figure 6.22: Puissance moyenne totale du circuit de test pendant une opération AES à différentes températures

On remarque qu'entre la température typique et la température basse, l'écart est très faible pour les tensions d'alimentation inférieures à 1V et quasiment nul à 1.2V. Les courants de fuite sont en effet minoritaires pour une température typique et encore plus pour une température basse. Les écarts sont donc très faibles.

Cependant, pour une haute température, la consommation de *leakage* est généralement plus élevée. On observe bien cette consommation plus élevée dans tous les points de mesure.

Au terme de ces différentes mesures et simulations, nous pouvons conclure que la simu-

lation offre des résultats cohérents et ce même pour les circuits désynchronisés. Cela permet donc d'aisément tester de nouvelles architectures à désynchroniser et de rapidement estimer les résultats finaux que le concepteur peut espérer.

Concernant les performances brutes obtenues dans notre application, nous ne pouvons nier qu'elles ne sont pas excellentes. La principale raison de cela est, à notre avis, due au choix de l'architecture désynchronisée. En effet, la multitude de dépendances bouclées et la petite taille du circuit synchrone initial engendre un fort surcoût pour la désynchronisation et ce même en utilisant des contrôleurs de très petite taille. En outre, la maîtrise de la génération des délais n'est pas encore au point, mais aussi limitée par les capacités des outils synchrones. Cela nous permet certes d'obtenir de bons résultats de robustesse, mais accompagné d'une forte chute de performances et d'une augmentation de la consommation notable par rapport à une implémentation synchrone. Néanmoins, nous ne considérons pas dans nos mesures l'environnement nécessaire pour faire fonctionner le circuit synchrone (oscillateur à quartz, *Phase-Locked Loop* ou PLL, ...), là où les circuits asynchrones sont autonomes et n'ont pas besoin de cette circuiterie supplémentaire.

6.5 Application de la méthode de désynchronisation sur le circuit *tinyAES*

Pour soutenir notre point affirmant que l'architecture du circuit désynchronisé est la cause principale des performances médiocres obtenues pour la désynchronisation, nous avons à nouveau appliqué cette procédure sur un circuit de cryptage AES différent : *tinyAES* [25].

6.5.1 Réseau de registres standard

Le circuit synchrone considéré est une implémentation de l'algorithme d'AES complètement pipelinée, *i.e.* chaque étape est stockée dans un étage de registres. Avec cette implémentation, chaque cycle fourni une donnée cryptée en sortie, et consomme une donnée en entrée.

Le réseau de registres standard est celui de la Figure 6.23.

Ce réseau de registres est beaucoup plus volumineux que ce que nous avons rencontré jusqu'à présent. Cependant, il est à noter qu'il présente une certaine linéarité.

Intéressons-nous à quelques grandeurs caractéristiques de ce circuit :

— Nombre de bascules $Nb_{dff}(tinyAES) = 8000 \approx 248k\ u.a.$

— Nombre de registres $Nb_{reg}(tinyAES\_STD) = 405$

— Coefficient de linéarité $C_{lin}(tinyAES\_STD) = 1.15$

— Coût du contrôleur $Cost_{ctrl}(tinyAES\_STD) = 40k\ u.a. \approx 15\%\ Nb_{dff}(tinyAES)$

— Coût de la *bufferisation* $Cost_{buff}(tinyAES\_STD) = 0\ u.a.$

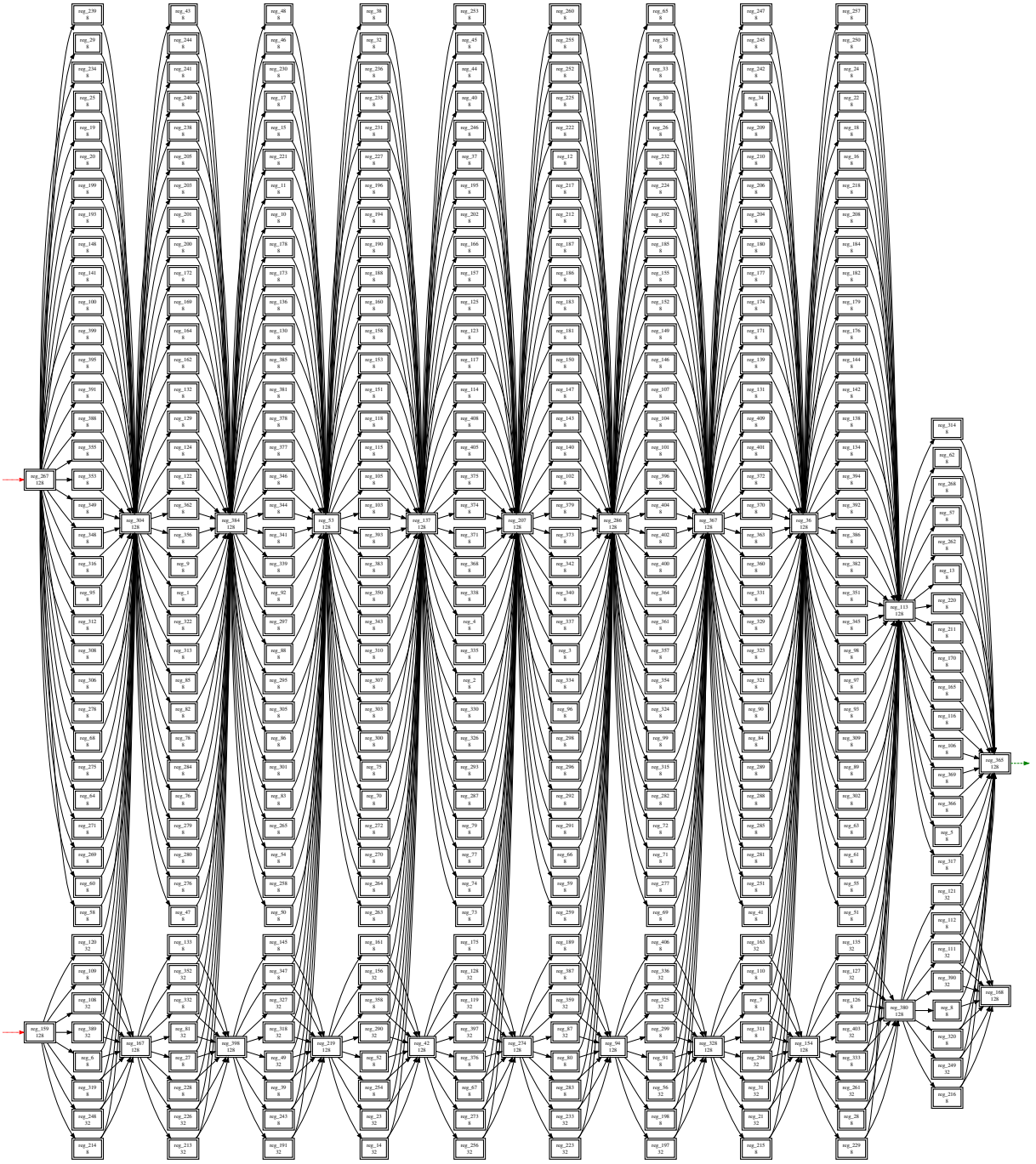


Figure 6.23: Forme graphique du réseau de registres du *tinyAES* standard

Le coefficient de linéarité, proche de 1, confirme la première impression. On remarque également que le contrôleur asynchrone est volumineux, 405 contrôleurs de registres, mais comparé au nombre de bascules dans le circuit, on atteint une proportion équivalente à ce que l'on obtenait pour les implémentations CAR de l'*AES_SC*. Enfin, la linéarité du circuit implique qu'il n'y a pas de boucles. Par conséquent, aucun registre *buffer* n'est nécessaire.

Néanmoins, implémenter un tel contrôleur n'est pas souhaitable, le nombre de chaînes de délais à générer par la suite serait trop élevé pour être réalisable par les outils dans un temps raisonnable.

6.5.2 Application de la méthode CAR

Les 6 stratégies, de regroupement de registres ont été explorées. Il en résulte 2 réseaux de registres différents présentés dans la Figure 6.24. Avec les stratégies CCAR et UCAR, on obtient deux *pipelines* parallèles, un *pipe* de donnée et un *pipe* de clé. Bien que cette implémentation pourrait être intéressante à étudier, nous avons préféré nous concentrer sur celle utilisant la stratégie DCAR pour avoir un *pipeline* le plus linéaire possible. Les caractéristiques du contrôleur de registre DCAR sont :

— Nombre de registres $Nb_{reg}(tinyAES\_DCAR) = 21$
— Coefficient de linéarité $C_{lin}(tinyAES\_DCAR) = 1$
— Coût du contrôleur $Cost_{ctrl}(tinyAES\_DCAR) = 710\ u.a. \approx 0.3\%\ Nb_{dff}$
— Coût de la *bufferisation* $Cost_{buff}(tinyAES\_DCAR) = 0\ u.a.$

Le réseau de registres étant totalement linéaire, le coût en surface du contrôleur associé est très faible.

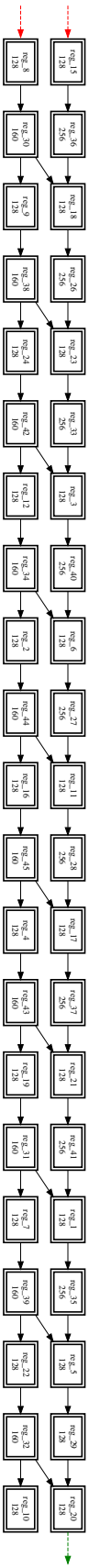
6.5.3 Placement Routage et analyse de timing du circuit désynchronisé

La linéarité du contrôleur choisie à un avantage indéniable comparé au circuit *AES_SC* : aucun conflit de contraintes n'est présent.

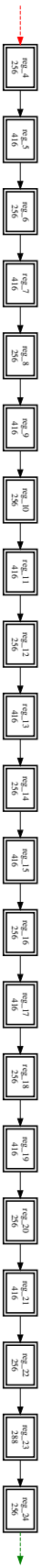
En procédant au flot complet de placement routage, pour l'implémentation synchrone et la version désynchronisée, nous avons observé le comportement de la surface à chaque étape de ce flot (Tableau 6.10).

À l'état initial, les implémentations synchrone et désynchronisée sont très proches, et ce jusqu'à l'étape de génération des délais. Pendant cette dernière, l'implémentation désynchronisé augmente sa surface de près de 5%, en raison des délais nécessaires à la contrainte de *setup*.

Il est à noter que cette étape prend également un temps considérable, $\approx 25h$. Bien que cela reste raisonnable dans un cadre industriel, il est à noter que l'architecture choisie est la plus simple possible. Une architecture plus complexe pourrait donc faire augmenter le temps pris, par cette étape de génération de délais, jusqu'à le rendre rédhibitoire.



(a) CCAR / UCAR



(b) DCAR

Figure 6.24: Forme graphique du réseau de registres du *tinyAES* après groupages

	Synchrone			DCAR			
	Surface		Temps	Surface			Temps
Étape	**u.a.**	**Δ**	**de run(h)**	**u.a.**	**Δ**	**vs. Synchr.**	**de run(h)**
Initialisation	253622		0.02	252782		-0.3%	0.03
Placement	254338	0.3%	0.36	253773	0.4%	-0.2%	0.42
CTS	254859	0.5%	0.47	254593	0.7%	-0.1%	0.7
Géné. Délais	254872	0.5%	0.65	265490	5.0%	4.2%	25.52
Routage	255042	0.6%	1.04	265835	5.2%	4.2%	26.12
Final	255043	0.6%	1.68	268982	6.4%	5.5%	27.62

Tableau 6.10: Évolution de la surface pendant le flot de placement routage pour l'implémentation synchrone et l'implémentation désynchronisée de *tinyAES*

Le temps de cette opération est dû au fait que pour cette expérimentation, nous avons procédé au flot de placement routage dans trois *corners* PVT : lent, rapide et typique. Cela est certes plus réaliste pour un flot synchrone mais, au vu des points soulevés sur la variabilité du temps de traversée des cellules en fonction des *corners*, qui s'avère compliqué à gérer pour la génération des délais par les outils synchrones.

6.5.4 Performances des circuits en simulation

Après avoir obtenu des circuits, synchrone et désynchronisé, nous avons procédé à des simulations afin d'évaluer les performances.

N.B. : *toutes les mesures présentées par la suite ont été effectuées dans le corner typique.*

La cible de performances synchrones imposée, pour tous les *corners*, est une fréquence d'horloge de 25MHz.

Cette implémentation d'AES traite une donnée dès qu'une donnée est disponible en entrée. Dans un contexte synchrone, cela demande de considérer à chaque cycle une nouvelle donnée en entrée, et d'en fournir une en sortie. Dans le cas d'un circuit désynchronisé, étant donné qu'un canal d'entrée à été ajouté, lorsqu'une nouvelle donnée arrive en entrée du *pipeline*, alors une requête lui est associée. La donnée est alors disponible en sortie lorsque cette même requête arrive en sortie.

Pour le premier scénario d'exécution, une donnée seule arrivant en entrée du *pipeline*, on relève :

— Pour le circuit synchrone, une latence entre l'entrée et la sortie de $T_{op}(tinyAES\_sync) = 165.5ns$. Cette valeur à été obtenue pour la fréquence maximale atteignable par l'implémentation synchrone : $F_{max}(tinyAES\_sync) = 125MHz$

— Dans le cas du circuit désynchronisé, entre l'arrivée et la sortie de la requête on obtient $T_{op}(tinyAES\_DCAR) = 165.2ns$ soit une fréquence équivalente égale à $F_{max}(tinyAES\_sync)$.

Dans ce cas de pipeline linéaire, la performance en latence obtenue est équivalente à celle du système synchrone homologue.

Intéressons-nous maintenant au cas d'un flot continu de données afin de pouvoir calculer le débit du *pipeline* désynchronisé. Pour le circuit synchrone, cette différence de flot de données n'a aucun impact. Cependant, pour les circuits désynchronisés, de la congestion dans le réseau du contrôleur peut survenir. Dans la Figure 6.25 est présentée l'évolution du temps d'opération en fonction du nombre de données fournies, de manière soutenue, en entrée du *pipeline* désynchronisé.

À partir de la 6ème opération, on observe une augmentation forte de la latence de calcul à travers le circuit désynchronisé jusqu'à atteindre un palier à partir de la 9ème donnée. Le temps d'opération subit une augmentation d'environ 30%.

De plus, si on s'intéresse à la cadence à laquelle le circuit désynchronisé peut fournir des données en sortie, on trouve alors un temps de cycle de $21ns$ ce qui représente une fréquence

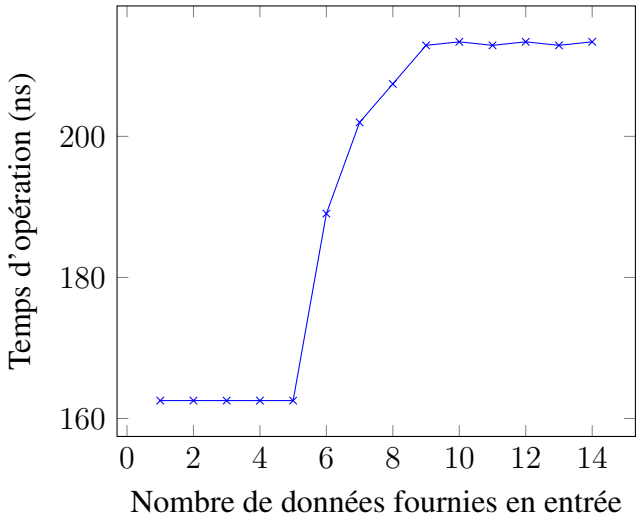


Figure 6.25: Évolution du temps d'opération dans le *tinyAES_DCAR* en fonction du nombre de données fournies en entrées

équivalente de $F_{eq}(tinyAES\_DCAR) = 47MHz$, là où le synchrone reste avec une fréquence maximale atteignable constante de $125MHz$.

Note sur la congestion et le placement des délais

Cette chute de performance pour le circuit peut néanmoins s'expliquer par le placement des délais dans le contrôleur. En effet, considérons le fragment de contrôleur de la Figure 6.26.

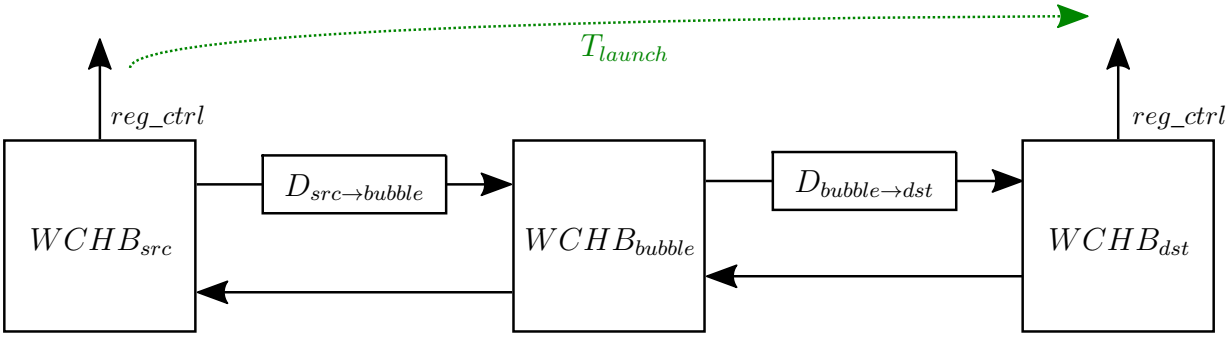


Figure 6.26: Placement des délais dans un contrôleur asynchrone

Pour respecter la contrainte de *setup*, avec $D_{a \to b}$ les valeurs des délais entre les contrôleurs indexés a et b, nous pouvons approximer :

$$D_{src \to bubble} + D_{bubble \to dst} > T_{launch}$$

Pour ce qui est de la latence, si on suppose le fragment et la suite du contrôleur vide, lorsque qu'une requête traverse ce fragment, la latence entre $WCHB_{src}/reg\_ctrl$ et $WCHB_{dst}/reg\_ctrl$ est simplement la somme des deux délais séparant les deux contrôleurs.

$$T_{latence} = D_{src \to bubble} + D_{bubble \to dst}$$

Intéressons-nous maintenant au temps minimal pour que deux requêtes consécutives puissent être acceptées en entrée. Supposons qu'une requête arrive en entrée et que le *pipeline* peut l'accepter, la requête passe une première fois par le délai $D_{src \to bubble}$ pour le niveau haut, l'acquittement revient jusqu'au $WCHB_{src}$ et fait retomber la requête. Pour libérer le premier contrôleur, la requête doit passer une seconde fois par $D_{src \to bubble}$, pour propager le niveau bas, et revenir par le chemin d'acquittement.

Avec un raisonnement similaire, on obtient également que le temps minimal entre deux données en sortie du *pipeline* sera régi par la traversée de deux fois $D_{bubble \to dst}$.

La fonction de rendez-vous du contrôleur implique que le temps de cycle est contraint par le maximum de ces deux temps de traversée, on à donc :

$$T_{cycle} = 2 \times max(D_{src \to bubble}, D_{bubble \to dst})$$

En considérant les deux cas extrêmes de répartition des délais :

— Le délai est totalement concentré sur une des deux chaînes de délais :

$$T_{cycle} = 2 \times max(T_{launch}, 0) = 2 \times T_{launch}$$

— Le délai est également réparti entre les deux chaînes de délais :

$$T_{cycle} = 2 \times max(T_{launch}/2, T_{launch}/2) = T_{launch}$$

De plus, avec un *pipeline* linéaire le raisonnement est applicable sur la totalité du pipeline. Avec $WCHBs$ l'ensemble des contrôleurs, et $D_{a \to b}$ la valeur du délai entre deux contrôleurs, indexés par a et b, s'il existe, 0 sinon, on en déduit les deux temps caractéristiques suivants :

$$T_{latence}(pipeline) = \sum^{\forall (a,b) \in WCHBs} D_{a \to b}$$
$$T_{cycle}(pipeline) = 2 \times max(D_{a \to b}),\ \forall (a,b) \in WCHBs$$

En prenant en compte les délais à assurer pour la contrainte de *setup*, on déduit les temps caractéristiques suivants :

$$T_{latence}(pipeline) \approx \sum^{circuit} T_{launch}$$
$$max_{pipeline}(T_{launch}) < T_{cycle}(pipeline) < 2 \times max_{pipeline}(T_{launch})$$

Pour la latence, on obtient, sans prendre en compte la congestion, le résultat attendu pour des circuits désynchronisés : une performance en temps moyen.

Pour ce qui est du temps de cycle, qui est lié au débit du *pipeline*, les circuits désynchronisés (considéré dans nos travaux) sont limités, comme les circuits synchrones, par le plus long chemin de donnée existant entre deux registres dans le circuit. Cependant, dans le cas où les délais sont générés de manière déséquilibrés, cette performance peut être jusqu'à réduite de moitié.

Deux axes d'améliorations ont été envisagés pour pallier cette problématique :

— Contrôler finement la génération de délais pour pouvoir les répartir de manière adéquate dans le contrôleur.

— Utiliser d'autres types de contrôleurs permettant un meilleur découplage entre les requêtes, ceux présentés dans [51] par exemple

6.5.5 Consommation

Nous avions vu dans l'application sur *AES_SC* que la consommation énergétique n'était pas améliorée par la désynchronisation. La principale cause à cela est que de l'activité était toujours présente dans le contrôleur, mais aussi l'utilisation de *clock-gating* dans le circuit synchrone.

Cependant, dans le cas d'une structure linéaire comme celle du *tinyAES*, de l'activité dans le circuit désynchronisé n'est créée que lorsqu'une donnée transite au travers du *pipeline*. De plus, l'utilisation de *clock-gating* automatique pour le circuit synchrone n'est pas aussi naturel que dans un circuit à base de machine d'état.

Il est à noter que pour implémenter un *clock-gating* efficace, il faudrait propager un signal de validité de la donnée en même temps que la donnée elle-même, ce qui ressemble fortement à un protocole *bundled-data*.

Dans la Figure 6.27 sont présentés les profils de consommation lorsqu'une seule donnée est fournie en entrée du circuit synchrone et du circuit désynchronisé.

Comme attendu, de l'activité dans le circuit désynchronisé n'est présente que lorsqu'une donnée traverse le *pipeline* contrairement au cas synchrone qui présente toujours des pics de consommation même sans calcul « utile ».

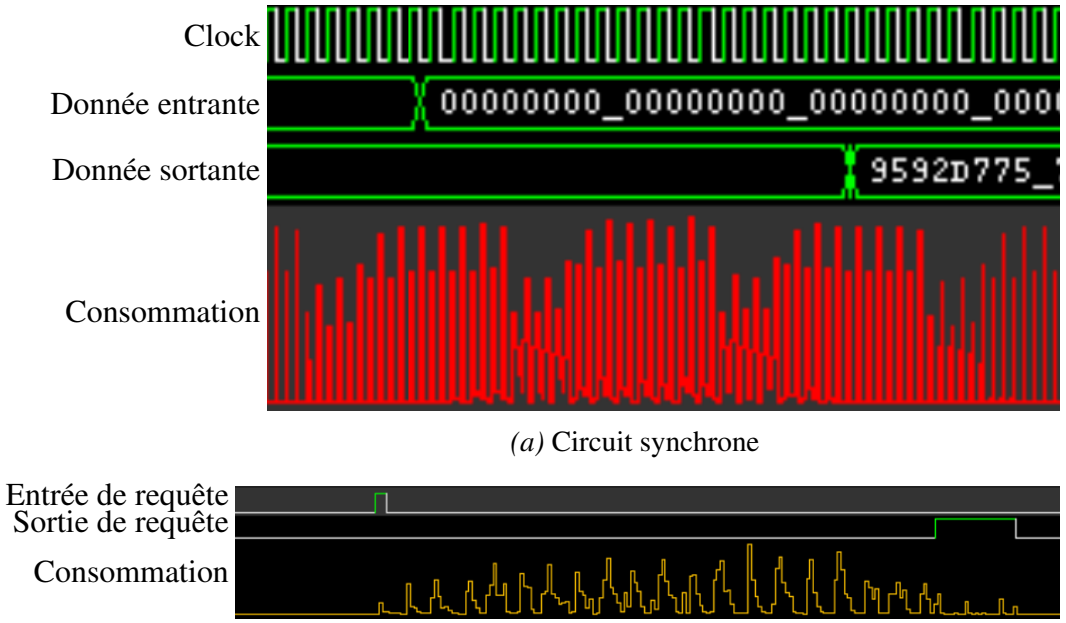
La Figure 6.28 présente les profils de consommation pour les situations où une donnée est fournie à chaque cycle, où une requête est fournie dès que possible.

Dans le cas de calculs soutenus, on remarque bien l'augmentation progressive de consommation dans les deux circuits. Pour des performances équivalentes, *i.e.* $F(tinyAES\_synchrone) = F_eq(tinyAES\_DCAR) = 47MHz$, on relève les consommations moyennes :

— $P_{moy}(tinyAES\_synchrone) = 11.2mW$

— $P_{moy}(tinyAES\_DCAR) = 7.9mW$

La désynchronisation permet donc dans ce cas de substantiellement économiser de l'énergie à performances équivalentes. Le gain en consommation est ici certainement dû à l'absence de pics de consommation à chaque front montant d'horloge.



(a) Circuit synchrone

(b) Circuit désynchronisé

Figure 6.27: Profil de consommation du circuit pendant la traversée d'une donnée dans le *pipeline*

Il est à noter que, dans le cas synchrone, ces pics de consommation sont bien identifiables alors que, dans le circuit désynchronisé, la consommation semble être plus proche d'un niveau de bruit.

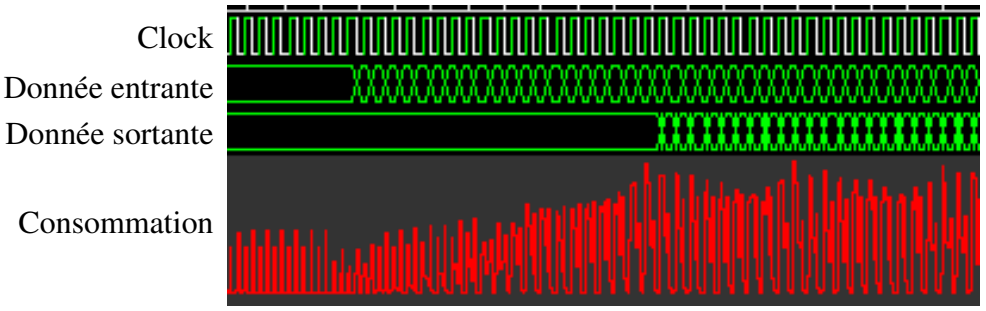
Ce point peut être un avantage en termes de sécurité puisque certaines attaques de circuits sécurisées sont basées sur les profils de consommation [32].

Pour vérifier ce point, la Figure 6.29 présente les transformées de Fourrier des deux circuits considérés en opération soutenue.

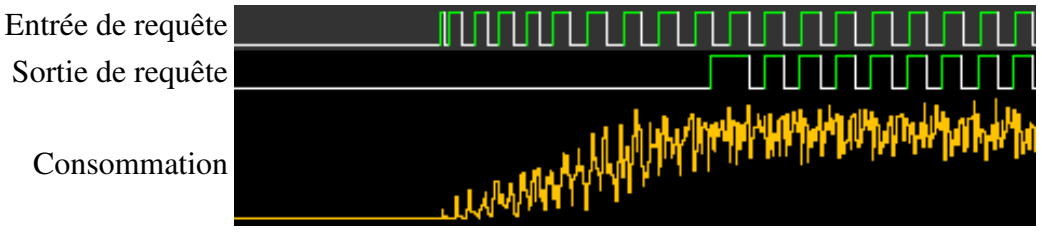
On remarque que, bien qu'ils soient d'amplitude moindre, les harmoniques présents dans la transformée de Fourrier synchrone se retrouvent également dans celle du circuit désynchronisé. Sur un *pipeline* linéaire, la propagation des données dans les circuits synchrones et désynchronisés sont finalement très similaires. Par conséquent, l'utilisation des circuits désynchronisés, tels que nous les avons conçus,ne permet pas d'assurer une meilleure résistance aux attaques classiques.

Dans ce chapitre, nous avons pu étudier l'impact réel de la désynchronisation de circuits. Nous avons pu remarquer que sur la première application effectuée, sur *AES_SC*, les résultats de performances et de consommation, en mesure et en simulation, ne sont pas en général excellents bien que la robustesse soit correcte.

Les méthodologies de désynchronisation et de groupage CAR proposées permettent cependant d'obtenir une efficacité bien meilleure que celle obtenue avec une approche standard.



(a) Circuit synchrone



(b) Circuit désynchronisé

Figure 6.28: Profil de consommation du circuit avec une séquence de données soutenue à l'entrée du *pipeline*

Néanmoins, les comparaison effectuées avec le circuit synchrone initial sont très désavantageuses pour les circuits désynchronisés. En effet, pendant nos mesures, nous n'avons pas tenu compte de la circuiterie supplémentaire nécessaire pour faire fonctionner un circuit synchrone (oscillateur à quartz, PLL, ...).

Les résultats obtenus permettent cependant de valider la fiabilité des mesures en simulation. Cela nous a donc permis d'utiliser notre méthode sur une autre architecture d'AES plus adaptée à la désynchronisation.

Dans ce dernier cas, on obtient alors des résultats plus satisfaisant pour les circuits désynchronisés, mais nous nous somme également heurtés à des problématiques de congestion dans le circuit qui peuvent significativement réduire les performances.

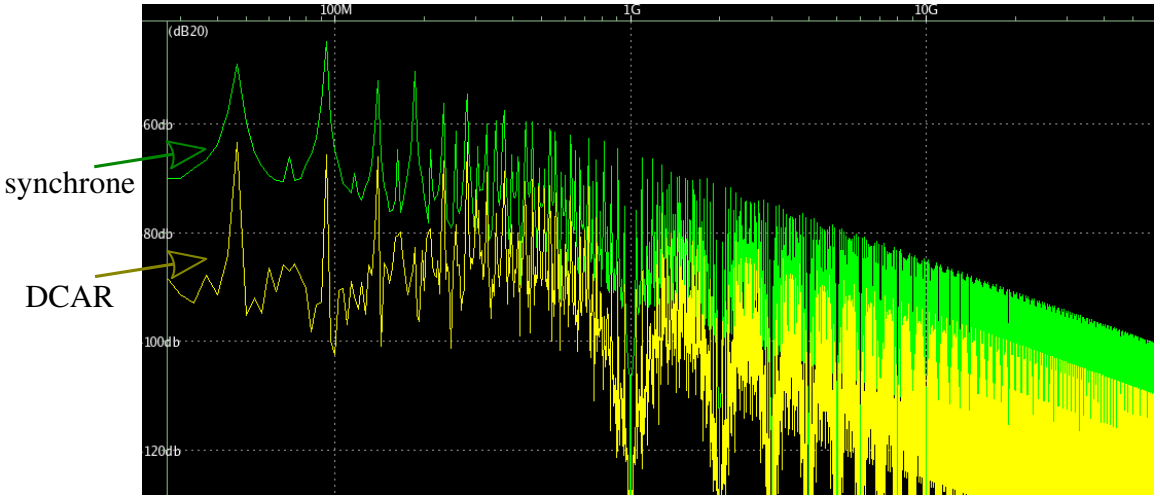


Figure 6.29: Transformées de Fourrier de la consommation des circuits *tinyAES* synchrone, et désynchronisés, à performance équivalente

Chapitre 7

Conclusion

L'idée porteuse de mes travaux de thèse était de rendre la conception des circuits asynchrones plus simple et plus abordable pour les concepteurs de circuits synchrones. Dans cette optique, nous avons décidé de proposer une méthode systématique permettant de transformer un circuit synchrone existant en circuit asynchrone : la désynchronisation.

Pour y parvenir, nous avons choisi un paradigme asynchrone proche des circuits synchrones : les circuits *micropipelines*. En effet, en utilisant cette classe de circuit avec des canaux à donner groupées (*bundled-data*), implémentant un protocole 4 phases, la désynchronisation peut se résumer au remplacement de l'arbre d'horloge par un contrôleur asynchrone gérant les communications entre les registres. La méthode de désynchronisation proposée doit donc permettre de créer ce contrôleur asynchrone et de s'assurer que pour tout circuit synchrone, sa version désynchronisée soit équivalente.

Cette méthode se base sur l'extraction des dépendances de données que nous représentons sous forme de réseaux de registres ou RN pour *Register Network*. Ce dernier, qui peut se représenter sous deux formes, graphique ou matricielle, permet de déduire le modèle formel du contrôleur associé aux chemins de données que nous appelons réseau asynchrone ou AN pour *Asynchronous Network*. Enfin, en utilisant une bibliothèque de cellules asynchrones, adaptées aux choix de canal et de protocole, nous obtenons le contrôleur asynchrone ou CN pour *Controller Network*.

La traduction directe du réseau de registres en contrôleurs asynchrones est appelée méthode directe. Bien que cette dernière permette d'obtenir une solution fonctionnelle, nous avons remarqué que la taille des contrôleurs générés est conséquente. Pour cette raison, nous avons proposé dans ce manuscrit des pistes afin de réduire la taille des réseaux de registres et, par conséquent, des contrôleurs asynchrones. La méthode proposée de regroupement des registres se base sur le principe des registres activés simultanément ou CAR pour *Concurrently Activated Registers*. En utilisant cette méthode de regroupement de registres, on réduit de manière substantielle le coût en surface du contrôleur asynchrone et des chemins de données (*bufferisation* limitée). Par ailleurs, les analyses de timing s'en trouvent simplifiées.

Une fois le circuit de contrôle obtenu, il faut alors assurer que toutes les contraintes de *timing* soient bien respectées en générant les délais nécessaires. Durant cette thèse, plusieurs méthodes ont été évaluées. Même si aucune méthode ne s'avère parfaite, la fonctionnalité des circuits désynchronisés a toujours été garantie. Les outils utilisés, étant prévus pour la conception synchrone, ont en effet généré des délais en utilisant des cellules simples (*buffers* et inverseurs), mais ils ne sont malheureusement pas idéaux pour exploiter au mieux la robustesse de nos circuits désynchronisés. Enfin, une dernière méthode a été envisagée pour concevoir les délais. Elle se base sur la duplication du chemin critique et nous est apparu comme une méthode prometteuse à explorer.

Afin de valider les méthodes présentées, nous les avons testés sur un circuit de chiffrement AES existant dans l'entreprise. Plusieurs variantes désynchronisées de ce circuit ont été générées afin d'effectuer des comparaisons. Les différentes versions ont été implémentées sur silicium et des mesures physiques ont été réalisées et mise en regard des résultats obtenus en simulation. Le circuit obtenu nous permis d'identifier des points cruciaux suivants :

— La désynchronisation est premièrement très fortement dépendante de l'architecture du circuit. En effet, l'implémentation synchrone de l'AES, conçue dans une optique synchrone d'économie de surface et d'énergie, est très compacte et son architecture comporte de nombreuses boucles de dépendances de données. La désynchronisation directe, méthode standard, de ce circuit résulte en un circuit beaucoup plus grand que son équivalent synchrone à cause des dispositions à prendre pour gérer les boucles (Règle 1, Capacité minimale des boucles et Règle 2, Intégrité des données).
— La méthode CAR permet de grandement réduire l'impact en surface et en performances de la désynchronisation. Les regroupements proposés permettent d'améliorer également l'efficacité énergétique du circuit désynchronisé.
— Les résultats entre simulation et mesures physiques étant cohérents, nous pouvons considérer que la simulation est un moyen fiable de comparaison. Elle pourra être utilisée afin d'évaluer au préalable l'impact d'une désynchronisation.
— Les performances obtenues avec les circuits désynchronisés se sont avérées moins bonnes que prévues initialement. En effet, la génération des délais n'est pas actuellement satisfaisante car elle est le principal contributeur de la perte de performances.
— Un fonctionnement à très basse tension d'alimentation a été validé. Cependant, étant donné les marges de délais prises, il est difficile de tirer des conclusions la robustesse des circuits désynchronisés.

Grâce à notre étude, nous avons compris que l'architecture du circuit synchrone est la principale source des problèmes rencontrés durant la désynchronisation. Nous avons donc décidé de désynchroniser une architecture différente de l'AES. Le résultat de cette désynchronisation est, à première vue, bien plus efficace que dans le premier cas. En effet, avec les mêmes difficultés de génération des délais, nous obtenons des performances équivalentes en termes de latence

entre le circuit synchrone et sa version désynchronisée. Cependant, le débit atteignable avec le circuit désynchronisé a été détérioré cette fois par le placement des délais dans le contrôleur.

En conclusion, les travaux menés pendant cette thèse ont permis d'obtenir de manière systématique des circuits désynchronisés depuis une spécification synchrone et en utilisant uniquement des outils industriels standards. Les circuits obtenus bien que fonctionnels présentent des performances très variables en fonction de l'architecture synchrone initiale. La génération des délais et leur placement dans le contrôleur restent cependant un point majeur à investiguer pour obtenir des résultats vraiment satisfaisants. Pour améliorer d'avantage l'efficacité énergétique de la désynchronisation, des travaux sont à mener pour identifier automatiquement les chemins de données où des structures de choix pourraient être utilisées. En effet, si l'utilisation de multiplexeurs (*merge*) paraît assez naturelle, il n'en est pas de même de leur symétrique, les démultiplexeurs (*split*). Ces structures sont plus difficiles à identifier car elles ne sont pas utilisées dans les circuits synchrones. Au-delà d'une méthode de désynchronisation, les travaux proposés se veulent une invitation à l'utilisation de technologies asynchrones dans le monde industriel.

Bibliographie

[1] ABDALLA, Y. S. Reduction of short circuit current in static cmos inverters using novel smart delay generator circuits. *Elektrotechnik und Informationstechnik 129* (2012), 83–87.

[2] ALTIERI SCARPATO, M. *Digital circuit performance estimation under PVT and aging effects*. Theses, Université Grenoble Alpes, Dec. 2017.

[3] ANGHEL, L., SAVULIMEDU VEERAVALLI, V., ALEXANDRESCU, D., STEININGER, A., SCHNEIDER-HORNSTIEN, K., AND COSTENARO, E. Single event effects in muller c-elements and asynchronous circuits over a wide energy spectrum.

[4] ANIS, M., AREIBI, S., MAHMOUD, M., AND ELMASRY, M. Dynamic and leakage power reduction in mtcmos circuits using an automated efficient gate clustering technique. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)* (June 2002), pp. 480–485.

[5] BERTRAND, F., CHERKAOUI, A., SIMATIC, J., MAURE, A., AND FESQUET, L. Car : On the highway towards de-synchronization. In *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (Dec 2017), pp. 339–343.

[6] BLOOM, B., ISTRAIL, S., AND MEYER, A. R. Bisimulation can't be traced. In *15th ACM Symposium on Principles of Programming Languages (SIGPLAN-SIGACT)* (New York, NY, USA, 1988), POPL '88, ACM, pp. 229–239.

[7] BLUNNO, I., CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., LWIN, K., AND SOTIRIOU, C. Handshake protocols for de-synchronization. In *10th International Symposium on Asynchronous Circuits and Systems (ASYNC)* (2004), pp. 149–158.

[8] BLUNNO, I., AND LAVAGNO, L. Automated synthesis of micro-pipelines from behavioral verilog hdl. In *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)* (April 2000), pp. 84–92.

[9] BRAMS, G. *Réseaux de Petri : théorie et pratique*, vol. 2. Masson, 1983.

[10] BUTZEN, P., AND RIBAS, R. Leakage current in sub-micrometer cmos gates.

[11] CHINNERY, D., AND KEUTZER, K. *Closing the Gap Between ASIC & Custom : Tools and Techniques for High-Performance ASIC Design*. Springer, 2007.

[12] CHU, T.-A. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[13] CORTADELLA, J., KISHINEVSKY, M., KONDRATYEV, A., LAVAGNO, L., PASTOR, E., AND YAKOVLEV, A. Petrify : a tool for synthesis of petri nets and asynchronous circuits. Software, `http://www.cs.upc.edu/~jordicf/petrify/` [accessed 2016-04-11].

[14] CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., AND SOTIRIOU, C. Desynchronization : Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25*, 10 (Oct 2006), 1904–1921.

[15] CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., AND SOTIRIOU, C. P. Desynchronisation : Synthesis of asynchronous circuits from synchronous specifications.

[16] DAEMEN, J., AND RIJMEN, V. The design of rijndael : Aes - the advanced encryption standard.

[17] DAVIES, M., SRINIVASA, N., LIN, T., CHINYA, G., CAO, Y., CHODAY, S. H., DIMOU, G., JOSHI, P., IMAM, N., JAIN, S., LIAO, Y., LIN, C., LINES, A., LIU, R., MATHAIKUTTY, D., MCCOY, S., PAUL, A., TSE, J., VENKATARAMANAN, G., WENG, Y., WILD, A., YANG, Y., AND WANG, H. Loihi : A neuromorphic manycore processor with on-chip learning. *IEEE Micro 38*, 1 (January 2018), 82–99.

[18] DINH DUC, A. V. *Automatic synthesis of QDI asynchronous circuits*. Theses, Institut National Polytechnique de Grenoble - INPG, Mar. 2003. ISBN 2-84813-010-5.

[19] FURBER, S., AND DAY, P. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 4*, 2 (1996), 247–253.

[20] FURBER, S. B., DAY, P., GARSIDE, J. D., PAVER, N. C., AND WOODS, J. V. Amulet1 : a micropipelined arm. In *Proceedings of COMPCON '94* (Feb 1994), pp. 476–485.

[21] GIMENEZ, G., CHERKAOUI, A., COGNARD, G., AND FESQUET, L. Static Timing Analysis of Asynchronous Bundled-data Circuits. In *24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC 2018)* (Vienna, Austria, May 2018). Best Paper Award.

[22] GUYOT, A., AND ABOU-SAMRA, S. Power consumption in digital circuits.

[23] HAN, Z., AND LEE, G. B. Reduction method for reachability analysis of petri nets. *Tsinghua Science and Technology 8*, 2 (2003), 231–235.

[24] HAWICK, K. A., AND JAMES, H. A. Enumerating circuits and loops in graphs with self-arcs and multiple-arcs. In *FCS* (2008), pp. 14–20.

[25] HSING, H. Tiny aes. *https ://opencores.org/projects/tiny_aes* (2012).

[26] HUFFMAN, D. A. The synthesis of sequential switching circuits.

[27] JAYANTHI, D., AND RAJARAM, M. The design of high performance asynchronous pipelines with quasi delay-insensitive. *International Journal of Computer Applications 52* (08 2012), 35–41.

[28] JOHNSON, D. B. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing 4*, 1 (1975), 77–84.

[29] KESSELS, J., PEETERS, A., WIELAGE, P., AND KIM, S.-J. Clock synchronization through handshake signalling. In *Proceedings Eighth International Symposium on Asynchronous Circuits and Systems* (April 2002), pp. 59–68.

[30] KLEEMAN, AND CANTONI. Can redundancy and masking improve the performance of synchronizers ? *IEEE Transactions on Computers C-35*, 7 (July 1986), 643–646.

[31] KLEEMAN, L., AND CANTONI, A. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers C-36*, 1 (Jan 1987), 109–112.

[32] KOCHER, P. C., JAFFE, J. M., AND JUN, B. Introduction to differential power analysis and related attacks.

[33] LAVINGTON, S. H. *Early British computers*. Univ. Pr., 1980.

[34] LIGTHART, M., FANT, K., SMITH, R., TAUBIN, A., AND KONDRATYEV, A. Asynchronous design using commercial HDL synthesis tools. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on* (2000), pp. 114–125.

[35] LINDER, D. H., AND HARDEN, J. H. Phased logic : supporting the synchronous design paradigm with delay-insensitive circuitry. *IEEE Transactions on Computers 45*, 9 (Sep. 1996), 1031–1044.

[36] LINES, A. M. Pipelined asynchronous circuits. Tech. rep., California Institute of Technology, 1998.

[37] MARTIN, A., LINES, A., MANOHAR, R., NYSTROEM, M., PENZES, P., SOUTHWORTH, R., CUMMINGS, U., AND KWAN LEE, T. The design of an asynchronous mips r3000 microprocessor.

[38] METERELLIYOZ, M., MAHMOODI, H., AND ROY, K. A leakage control system for thermal stability during burn-in test. In *IEEE International Conference on Test, 2005.* (Nov 2005), pp. 10 pp.–991.

[39] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (April 1965).

[40] MORENO-CONDE, A., AND CORTADELLA, J. Synthesis of all-digital delay lines. *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)* (2017), 75–82.

[41] MULLER, D., AND BARTKY, W. *A Theory of Asynchronous Circuits*. No. vol. 1 in Report : Digital Computer Laboratory. Univ., 1956.

[42] MURATA, T. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (1989), 541–580.

[43] MYERS, C. J. Asynchronous circuit design.

[44] PARHI, K. K. Vlsi digital signal processing systems : Design and implementation.

[45] PHAM-QUOC, C., AND DINH-DUC, A.-V. Hazard-free muller gates for implementing asynchronous circuits on xilinx fpga. pp. 289–292.

[46] REZZAG, A. *Syntèse Logique de Circuits Asynchrones Micropipeline*. PhD thesis, Laboratoire TIMA, CNRS/Grenoble INP/UJF, 2004.

[47] RUPP, K. Karl rupp. 42 years of microprocessor trend data. *https ://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/* (2018).

[48] SHARMA, K., AND KHATRI, S. A robust c-element design with enhanced metastability performance. pp. 95–100.

[49] SHARMA, S. Power and delay analysis of various c-element topologies : a comparative study. pp. 1–5.

[50] SIMATIC, J. *Design flow for ultra-low power : non-uniform sampling and asynchronous circuits*. Theses, Université Grenoble Alpes, Dec. 2017.

[51] SIMATIC, J., CHERKAOUI, A., BASTOS, R. P., AND FESQUET, L. New asynchronous protocols for enhancing area and throughput in bundled-data pipelines. In *29th Symposium on Integrated Circuits and Systems Design (SBCCI)* (Aug 2016), pp. 1–6.

[52] SIMATIC, J., CHERKAOUI, A., BERTRAND, F., BASTOS, R., AND FESQUET, L. A practical framework for specification, verification, and design of self-timed pipelines. .

[53] SIMATIC, J., CHERKAOUI, A., BERTRAND, F., BASTOS, R. P., AND FESQUET, L. A practical framework for specification, verification, and design of self-timed pipelines. *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)* (2017), 65–72.

[54] SPARS, J., AND FURBER, S. *Principles of Asynchronous Circuit Design : A Systems Perspective*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[55] ST-MICROELECTRONICS. Fdsoi, https ://www.st.com/content/st_com/en/about/innovation—technology/fd-soi.html.

[56] SUNIL DEEP MAHESHWARI, N. S. . R. R. Aspects of ic power dissipation. *https ://www.edn.com/design/integrated-circuit-design/4440402/Aspects-of-IC-power-dissipation* (2015).

[57] SUTHERLAND, I. E. Micropipelines. *Communications of the ACM 32*, 6 (1989), 720–738.

[58] TARJAN, R. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing 2*, 3 (1973), 211–216.

[59] TCL/TK. Tcllib 1.19 documentation. https ://core.tcl.tk/tcllib/doc/tcllib-1-19/embedded/www/toc.html, 17 Février 2018.

[60] TIERNAN, J. C. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM 13*, 12 (1970), 722–726.

[61] UMC. Design support. *http ://www.umc.com/english/pdf/design_support_Brochure.pdf*.

[62] VERHOEFF, T. Delay-insensitive codes - an overview. *Distributed Computing 3* (03 1988), 1–8.

[63] YEAP, G. K. *Practical Low Power Digital VLSI Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

Résumé

Mots clés : *Asynchrone, micropipeline, désynchronisation, flot de conception,outils industriels*

Dans le monde des *smartcards*, le besoin de communications *via* des dispositifs radio fréquences est de plus en plus courant. Le coût étant un point critique pour ces dispositifs, l'antenne, et la quantité d'énergie récupérée par celle-ci, à tendance à être réduite. Pour suivre cette contrainte, de nombreuses solutions techniques améliorant l'efficacité énergétique des circuits numériques synchrones existent. Néanmoins, les circuits asynchrones, s'adaptant naturellement aux conditions d'opération qui lui sont appliquées, fournissent une solution particulièrement adaptée aux problématiques liées à télé-alimentation.

Cependant, le manque de formation des ingénieurs, et d'outils qualifié, pour la conception de circuits asynchrones rend marginale leur utilisation dans un cadre industriel. Pour remédier à ces difficultés, une méthode systématique permettant de traduire une spécification synchrone en un circuit asynchrone *micropipeline* a été développée. Exploitant des modèles formels, un flot de désynchronisation a été mis en place et exercé sur un module cryptographique pour aboutir à une implémentation sur silicium. La caractérisation de ce circuit nous a montré la pertinence de la désynchronisation mais aussi ses limites. L'efficacité de la désynchronisation dépendant fortement de l'architecture du circuit synchrone initialement considéré, le flot proposé permet de rapidement estimer les performances du circuit désynchronisé, mais aussi d'obtenir un circuit physique fonctionnel.

Abstract

Keywords : *Asynchronous, micropipeline, desynchronization, design flow, industrial tools*

In the smart-cards market, contactless communication becomes more and more common. In order to reduce the cost of the systems, the power budget available reduces along with the size reduction of the antenna. Many design techniques are available to improve the efficiency of digital design. However, asynchronous designs, because of their adaptability to variable operating conditions, offer a natural solution for the contactless power supply problematics.

Nevertheless, the lack of engineers trained, and proven tools, to design asynchronous circuits brakes the adoption of asynchronous designs in the industry. To address theses issues, a systematic method allowing to turn already existing synchronous designs into asynchronous micropipelines had been developed. Using a formal model, a desynchronization process was established and used on a cryptographic module which was implemented on silicon. The characterization allowed us to evaluate the relevance of the desynchronization flow but also its limitations. The efficiency of desynchronized systems depending largely on the architecture of the initial synchronous circuit, the proposed method allows a quick estimation of the performances of the desynchronized circuit, and results in functional circuits.