vers la France. De même, si deux requêtes arrivent sur un *Arbitre* à quelques picosecondes d'écart, le circuit peut faire une pause dans son état métastable avant d'atteindre un de ses états stables et de résoudre le dilemme.

Les concepteurs débutants d'*Arbitre* cherchent souvent à éviter ces délais prolongés occasionnels en développant des circuits complexes. Ils commettent fréquemment l'erreur d'introduire un circuit qui pousse l'*Arbitre* vers une décision particulière si jamais le circuit se trouve dans l'état métastable. Cela revient à dresser l'âne de Buridan à aller vers la gauche (ou vers la droite) lorsqu'il est confronté à un choix difficile. Cependant un tel dressage a pour seul effet de présenter à l'âne trois choix plutôt que deux : aller à gauche, aller à droite, ou s'apercevoir qu'il y a un choix difficile et aller à gauche. Même un âne dressé mourra de faim s'il est incapable de trancher entre les deux dernières possibilités. En d'autres termes, pour reprendre la métaphore géographique, vous pouvez déplacer la ligne de crêtes des Pyrénées, mais vous ne pouvez pas vous en débarrasser. Bien qu'il n'existe aucun moyen d'éliminer la métastabilité, des circuits *Arbitre* simples et bien conçus peuvent garantir des pauses très courtes dans tous les cas. Typiquement, les *Arbitre* actuels ont des pauses normales de 100 picosecondes, et, exceptionnellement (une fois sur dix heures de fonctionnement), de 400 picosecondes. La probabilité qu'une pause survienne diminue exponentiellement avec la durée de la pause : une pause de 800 picosecondes se produit moins d'une fois pour un milliard d'années de fonctionnement.

Une nécessaire rapidité

Notre équipe étudie surtout le développement de systèmes asynchrones rapides. Nous avons observé que la simplicité est souvent garante de rapidité. Nous voulions construire un pipeline à double sens de circulation, comportant deux courants de données opposés (comme deux chaînes de passeurs de seaux parallèles, mais acheminant l'eau en sens contraire). Nous voulions que les données des deux flux interagissent à chaque étape ; la principale difficulté était de faire en sorte que chaque élément de données allant dans un sens interagisse avec chaque élément de données allant dans l'autre sens. L'arbitrage s'est révélé un facteur essentiel. À chaque jonction entre étapes successives, un circuit *Arbitre* autorisait le passage d'un seul élément à la fois. Nous avons beaucoup appris grâce à ce projet à propos de la coordination et de l'arbitrage. Nous avons ensuite construit des puces expérimentales pour vérifier la validité et la fiabilité de nos circuits *Arbitre*.

Plus récemment, nous nous sommes intéressés à une structure nommée FLEET. Ce nom fait référence à la rapidité de la structure, mais aussi à l'ensemble des composants qui la constituent, autant de «navires» de la flotte (*fleet*, en anglais) voguant à leur propre allure. Chaque navire effectue sa tâche en même temps que les autres ; le système FLEET les contrôle en faisant circuler des données parmi eux par l'intermédiaire d'un réseau de commutateurs asynchrones. Les recherches menées pour le projet FLEET nous ont conduits à de nombreuses découvertes. Nous voulions de la vitesse, et c'est pourquoi nous avons créé les circuits GasP de base. Nous voulions aiguiller des données d'un pipeline à un autre, comme des voitures aux jonctions autoroutières, et cela nous a amenés à concevoir une famille de circuits GasP plus vaste, agissant comme un système dense de voies rapides. Ces circuits peuvent transporter les données environ deux fois plus vite que les systèmes cadencés. Pour évaluer la vitesse de nos réseaux de commutateurs, nous construisons souvent sur nos puces expérimentales des circuits en anneau où des données tournent. Nous mesurons le temps qu'il faut à une donnée pour effectuer un tour complet, ce qui est bien plus simple que de mesurer le temps – notablement plus court – nécessaire aux données pour avancer d'une étape.

Fonctionnement d'un module GasP

Les modules GasP servent également d'éléments *Rendez-vous* pour un pipeline asynchrone de données. Au centre de chaque module se trouve un circuit NAND, qui produit une sortie FAUX si les entrées sont toutes deux VRAI. Sinon le circuit NAND produit une sortie VRAI. Au départ, tous les signaux dans les fils de connexion des modules sont VRAI *(en bleu, a)*. L'arrivée de données dans le pipeline *(b)* change le signal d'arrivée en FAUX *(en rouge)*. Un inverseur bascule le signal sur VRAI et l'envoie au circuit NAND, qui change sa sortie en FAUX. Ceci entraîne l'ouverture de la porte, permettant aux données d'avancer dans le pipeline. La sortie FAUX ramène également le fil d'entrée à son état VRAI (c'est-à-dire vide) par l'intermédiaire d'un transistor, et met le fil de sortie dans l'état FAUX (c'est-à-dire plein) par le biais d'un inverseur et d'un transistor. La sortie de la porte NAND redevient alors VRAI *(c)*, ce qui ferme la porte. Pendant ce temps, le signal FAUX du fil de sortie déclenche le même processus dans le module GasP suivant.