



Synthèse de moniteurs asynchrones à partir d'assertions temporelles pour la surveillance robuste de circuits synchrones

Alexandre Porcher

► To cite this version:

Alexandre Porcher. Synthèse de moniteurs asynchrones à partir d'assertions temporelles pour la surveillance robuste de circuits synchrones. Autre. Université de Grenoble, 2012. Français. NNT : 2012GRENT004 . tel-00986690

HAL Id: tel-00986690

<https://theses.hal.science/tel-00986690>

Submitted on 4 May 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nanoélectronique et Nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

M. Alexandre PORCHER

Thèse dirigée par **M. Laurent FESQUET**
et codirigée par **Mme. Katell MORIN-ALLORY**

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

Synthèse de moniteurs asynchrones à partir d'assertions temporelles pour la surveillance robuste de circuits synchrones

Thèse soutenue publiquement le **3 Mai 2012**,
devant le jury composé de :

M. Ian O'CONNOR

Professeur à l'École Centrale Lyon, Président

M. Ashraf SALEM

Professeur à l'Université Ain Shams, Egypte, Rapporteur

M. Habib MEHREZ

Professeur à l'Université Pierre et Marie Curie, Rapporteur

M. Gilles DEPEYROT

Directeur de la ligne de produits MEDALTM, Dolphin Intergration, Examineur

M. Laurent FESQUET

Maître de Conférence à l'Université de Grenoble, Directeur de thèse

Mme. Katell MORIN-ALLORY

Maître de Conférence à l'Université de Grenoble, Co-Directrice de thèse



Je sers la science et c'est ma joie...
Disciple

Remerciements

Je tiens tout d'abord à remercier les membres de mon jury qui ont permis cette soutenance :

- Ian O'Connor qui préside ce jury
- Habib Merhez et Ashraf Salem qui ont accepté de rapporter ma thèse malgré leurs charges de travail
- Gilles Depeyrot qui m'a fait l'honneur de participer à mon jury

Je tiens à remercier plus particulièrement Laurent et Katell, mes directeurs de thèse, pour m'avoir fait confiance lorsqu'ils ont accepté d'encadrer ma thèse. Merci pour leurs soutiens lorsque je doutais, pour leurs encouragements et leurs conseils. Merci aussi pour les discussions que nous avons pu avoir et qui ont largement dépassées le cadre du travail. Un merci spécial à Katell pour la verveine, les discussions de travail chez elle ; plus que mon encadrante tu es devenue une amie à part entière et c'est toujours un plaisir de vous voir toi, Olivier et les deux monstres !

Je tiens à exprimer ma gratitude à Dominique Borrione, directrice du laboratoire, qui a fait germer en moi l'envie de faire une thèse lors de mon stage de master. Je la remercie aussi pour son soutien, sa proximité et son expertise dans de nombreux domaines. Je ne me serais toutefois pas engagé en thèse sans les encouragements et la bonne humeur des membres du groupe VDS qui m'ont accueillis lors de mon premier stage à TIMA. Je pense à Yann le pro de PSL et du Select, Luca l'expert du SystemC, Amr le spécialiste de la tchatch, Eric l'amoureux du tacos, et Laurence, la directrice du groupe pour son franc parler ! Grâce à eux, mon arrivée à TIMA ressemblait à une réunion de famille ! Je ne suis pas arrivé seul à TIMA et je remercie Renaud pour sa capacité à réexpliquer tout un manuel d'utilisation lorsqu'on lui demande un simple conseil et aussi car j'ai trouvé plus gros mangeur que moi ! Je ne remercierai jamais assez Florent, mon co-bureau, de m'avoir supporté, moi et ma musique. Même si j'ai eu du mal au début à supporter le métal, je le remercie car en travaillant tout deux sur l'asynchrone et ayant les mêmes encadrants, nous avons pu parler de nos moments de doute et se soutenir mutuellement. Je remercie aussi les autres membres du groupe qui sont arrivés ensuite : Laila et Zeineb les nouvelles thésardes, avenir de la recherche. Je remercie enfin les stagiaires qui arrivaient chaque année avec le printemps. Enfin je remercie Jérôme, pour les parties de console/pizzas entre midi et deux.

Je tiens aussi à remercier tous les membres du groupe CIS où j'avais établi mes quartiers durant ces quatre années. Les anciens qui sont partis : Greg le pro des magic, Livier pour la bonne humeur, Rodrigo pour son inimitable sourire, Eslam pour les longues discussions, Hatem l'homme le plus discret que je connaisse, Khaled le poète, David le pro du poker, Cedric car je n'étais pas le seul en casquette et en tongues, Oussama l'expert des rings asynchrones et aussi pour sa gentillesse et sa bonne humeur, Hakim l'as du pixel, Matt pour les apéros dans ton jardin et Jérémie pour les pauses au soleil ! Je n'oublie pas les membres actuels grâce à qui l'ambiance CIS persiste, Harwaa, Hassan, Ahmed la relève du pixel ! Franck merci pour ton soutien et ta culture, à mon tour de te souhaiter bon courage pour l'écriture du manuscrit ! Merci à Karim, qui bien que loin de Grenoble, fait partie de l'équipe et qui a été la première personne à qui j'ai parlé en arrivant à l'ENSERG. Je n'oublierais jamais nos TPs ! Merci à Tugdual qui reprend le flambeau de l'asynchrone dans l'équipe, bon courage et merci pour tes conseils musicaux ! Merci à Gilles, jamais je n'aurais pensé qu'un chef puisse être aussi accessible et agréable. Merci à Taha, et

j'attends que tu deviennes maître du monde pour venir tout te piquer ! Encore un dernier merci aux stagiaires : Noémie ma tit' fillotte, Laure, Laurent avec qui j'ai eu plaisir à travaillé, Mounir et Julien et tous les autres qui ont amené un peu d'animation dans les couloirs de CIS. Enfin merci à Alice, qui bien que pièce rapportée de CIS, a complètement assimilé l'esprit du groupe.

Un gros merci à l'équipe ITA du laboratoire TIMA, pour leur disponibilité, leur bonne humeur et leur accueil. Merci à Laurence pour les Spams, Claire et Amandine pour leur sympathie, Youness sans qui j'aurais passé une nuit au poste. Sophie pour les missions, Anne-Laure pour les contrats et surtout parce que souvent quand je lui écrivais un mail c'était pour des congés. Merci à Marie-Christine pour sa gentillesse et à Lucie pour toutes les fois où je suis allé l'embêter pour une feuille ou un stylo ! Je remercie aussi tous le service info, Ahmed, Fred et Nicolas pour leur disponibilité quand je les appelais pour des conseils ou des installations.

D'une manière plus générale, je remercie vraiment tous les membres du labo, certains que j'ai eu comme enseignants à l'ENSERG et que j'ai recroisés dans les couloirs du labo, et tous les autres car j'ai toujours trouvé en face de moi des personnes sympathiques et ouvertes à la discussion.

Je ne peux pas écrire de remerciement sans penser aux membres de CMP avec qui j'ai partagé de nombreuses pauses café et de nombreux repas. Plus particulièrement Clothilde, Laurianne et Virginie car il est plaisant dans une journée de travail de passer des moments où l'on ne parle pas travail ! Merci à Isabelle qui m'a permis d'entrer dans mon bureau quand j'oubliais mes clefs. Mais surtout un gros merci à Patou et Sophie pour les pauses au fond du couloir CIS et les gros délires sur Dark Vador. Sophie merci pour ton soutien, ton écoute et ton amitié. J'espère de tout cœur que le nouveau départ que tu viens de prendre te mènera au bonheur. Merci enfin à Bernard d'avoir supporter les squatteurs de CIS devant la machine à café CMP !

Je souhaite remercier aussi les personnes avec qui j'ai pu travailler hors du labo. Tout le personnel de minatec et plus particulièrement Lorraine et Débora, Robin pour les conseils FPGA. Un immense merci à Alejandro car je n'ai jamais rencontré de personne plus polie et sympathique que toi et j'ai découvert avec toi qu'une personne pouvait nous remercier quand on lui demande un service par mail ! Je ne veux surtout pas oublier de remercier toute l'équipe de l'ESISAR qui m'a accueilli et a permis mes premiers pas dans l'enseignement. Merci à Bernard et Vincent pour leurs encadrements, merci à Yvan pour la clarté de ses cours et plus généralement merci à tous pour les bons moments passés là bas où j'ai une fois de plus eu l'impression d'arriver dans une grande famille, entièrement dévouée à ses élèves. J'en profite pour remercier les élèves que j'ai rencontrés et avec qui j'ai eu plaisir à enseigner. Un merci particulier à Keal pour la soirée passée chez lui ! Un gros merci à Richard Monvoisin pour m'avoir initié à la zététique et surtout merci à la team zétébière, Rem's, Poual et Cyrdu pour les grosses réunions zététique au O'Callaghan, car l'esprit critique ne s'use que si on ne s'en sert pas !!

Je ne peux pas non plus écrire mes remerciements sans penser à tous les amis que je me suis fait pendant les études sans qui je ne serais jamais arrivé au bout de cette thèse. Je pense d'abord aux amis qui ont animés les journées de cours au lycée et avec qui j'ai eu la chance de garder contact : Fanny, Céline, tout le groupe de la mafia Valencinoise et ceux qui gravitent autour ! Nico le sportif, l'homme le plus sain de corps et d'esprit que je connaisse. Martial le musicien et épicurien comme moi ! Ben le grimpeur et l'hyper sportif avec qui j'ai eu de nombreux fou rires. Merci à Yves le penseur et le sage, Romain le speed,

moteur de nombreuses motivations. Merci à Tonio le bosseur et l'amoureux de la bière, Djé pour nos longues discussions sur le bonheur, Steve l'homme de la sûreté nucléaire, Jeff qui a été mon colloc pendant un an, Maud sans qui je ne serais jamais rester à la Martinière et sans qui je ne serais jamais devenu docteur aujourd'hui. Un gros merci à Jérémy pour les sessions films le dimanche soir à point d'heure, Anne-Laure parce qu'elle parle plus vite que moi, Sébastien pour le subtil mélange de sagesse et de folie, Polo pour le trip à Amsterdam, Rosa pour la sortie luge, Ricky pour le style, Guillaume pour sa connaissance de la musique, Mata qui apporte un peu d'exotisme (et de la super nourriture) dans le groupe, Julie et Donni qui même s'ils sont loin, occupent une place particulière dans le groupe et dans mes pensées, Clément et Yannick pour leur côté rêveur qui fait du bien, Karine pour son message de répondeur et ses fous rires, David le postier fou pour les soirées magic, Troll qui comprend mon côté geek, Ti Jo pour les soirées tranquilles à Grenoble, Lyon ou Heyrieux, Florie que je considère comme ma sœur et Rémi qui est un peu mon grand frère dans mes souvenirs de gosse! Merci à Aline et Mouche pour sa présence incroyable qui emporte nos soirées vers d'autre monde! Merci à Caro pour sa tolérance et son ouverture d'esprit, à Daphnée d'amener un peu de noblesse dans ce groupe mais surtout pour son naturel. Merci à Gui, mon poussin, pour son déguisement du réveillon. Merci à Sly l'orfèvre du groupe, sauf après l'apéro. Je ne compte plus les soirées exceptionnelles en votre compagnie à tous et le nombre de fois où autour d'un décontractant, nous avons refait le monde. Une pensée aussi pour les parents de tous ces amis qui nous ont souvent accueillis à leur table. Grâce à vous tous j'ai pu grandir et devenir la personne que je suis.

Je voudrais aussi remercier les « voyageurs » en camion, Yannick, Kiwi et Rosen qui débarquaient à l'appart de temps en temps pour un petit repas entre amis. Merci à eux de m'avoir permis de ne jamais oublier que l'on ne cherche pas tous le bonheur dans la même direction. Merci à Yannou, Guiz gâteau et tous les autres qui nous ont toujours accueillis les dimanches après midi au soleil et à la campagne. Je tiens à remercier aussi les amis que j'ai rencontré en prépa, Xavier, Pingoo, Tom, Mick et Pierre sans qui je n'aurais jamais passé les vacances de Toussain en première année. J'ai eu la chance de rencontrer des gens fantastiques durant mes deux ans de prépa. Merci à Valère pour les pauses du soir après les révisions, à Tom pour sa motivation quand il faut organiser une soirée et pour les films souvenirs que je garde précieusement, Manu et Gégé pour le trip en Europe de l'Est, Pingoo pour nos discussions sur l'avenir et pour tes pecs, Mick pour ce style inimitable, Pierre qui ne laissera pas mourir le communisme, Candy pour sa voix, merci aux bisous pour la soirée mémorable de votre mariage, Fred et Mariel pour les bons petits repas chez vous, David Ko pour les délires et pour la reprise récente du badminton! Merci à toutes les filles du groupe qui ont organisées plusieurs réveillons que je n'oublierai pas, à Louise pour le printemps de Bourges et les sessions musiques dans la voiture. Merci à Camille (ou Leeloo?!), Mathilde et Caro pour avoir été LA colloc de filles, à Celia pour les délires de fin d'année en cours, Cécé, Aurélie et Marjo les trois comparses, merci à Carole pour son BDR! Merci à tous les autres que je ne cite pas mais avec qui j'ai au moins un souvenir marquant qui m'aura fait avancé, et vous êtes nombreux. Je remercie aussi les professeurs qui m'ont accompagnés pendant ces deux ans et plus particulièrement Mr Vincent pour sa pédagogie de l'enseignement et Romain avec qui j'ai passé des vacances surréalistes à Barcelone. Grâce à vous tous, la prépa restera un excellent souvenir!

Je tiens aussi à exprimer ma gratitude à tous mes amis de l'ENSERG. Tout d'abord tous les membres du BDE et BDS avec qui j'ai gardé de très bon contact et qui ont

transformé mes trois ans d'école en trois ans de folie ! Merci à Jojo, Mish et Bouze pour les sessions PES, révisions, pizza et films, à Sven pour l'initiation à la botanique, à Nieu, Befu et Soksy pour leurs poèmes, à Pince pour les TP de 3A, Zine pour la zik au sono, Sid pour la poutre, à la Bombe pour être aussi déjantée, à Lou Bart pour ces monologues génialissimes, à Amélie pour son calme qui était bien nécessaire dans certaines réunions, à CamCam ma binôme de projet, Ramon pour les sessions pêche, Ha lan pour sa motivation (extrême ? !) pour les jeux, à Laure pour les soirées films et les longues discussions. Merci aussi à Olivia qui a été mon meilleur soutien en arrivant à l'ENSERG après ma chute. Merci à Repi pour les champis, Nanou car elle nous supporte en soirée et que c'est une bonne perdante, Audrey pour le côté roots, Fab pour ces principes et pour le squash, Annick pour sa gentillesse et sa sensibilité sans borne, à Edith qui me prête Jojo entre deux répètes ! Merci à Brigitte et Hervé pour leur bonne humeur lors des balades natures (champi, pêche, méchoui et vin de noix !). Merci à Sylvain pour les discussions autour d'une mousse à l'AMAP et plus généralement je remercie les connaissances nouées à l'AMAP avec qui j'ai plusieurs fois refait le monde. Merci à Bertrand pour les pauses à Minatec et à Francesca pour son incomparable sourire ! Un merci particulier à François Cayre qui m'a confirmé qu'on pouvait être très bon scientifique, avoir une culture générale énorme et être un grand fêtard. Merci aussi Dédé, Sandrine, Chti, HP et tous le groupe de fous qui nous précédait d'un an à L'ENSERG. J'oublie sûrement beaucoup de monde mais je n'oublie pas les moments que j'ai passé dans cette école avec eux.

Un gros merci à tous le groupe de potes Grenoblois rencontré par l'intermédiaire de Zitoune. Merci infiniment à Zitoune pour m'avoir recueilli de nombreuses fois après des journées de travail moralement épuisantes, pour m'avoir écouté (très longtemps !) et m'avoir permis de penser à autre chose en refaisant le monde ou en faisant une petite coinche ou un PES. Tu es pour beaucoup dans l'accomplissement de cette thèse. Merci à Caillon pour les soirées coinches chez toi, pour le tour en parapente et pour tes alcools atypiques mais tellement bons. Merci à Ako, Anaïs (et bibi !) pour toutes les soirées passées chez vous autour d'un bon repas et d'un match de foot ou un massive music quizz. Merci à Ju pour les sessions consoles et apéro/jeux. Merci à Aurélie et Marie qui amène une touche féminine et beaucoup de bonne humeur au groupe. Merci à Ju Couchet pour les top spins, à Moin's et Celine pour la facilité qu'ils ont à transformer un apéro en soirée et les soirées en grosses fêtes. Merci à tous les autres avec qui j'ai passé au moins une soirée coinche ou console, peut-être mes deux loisirs favoris durant cette thèse ! Grâce à vous j'ai pu extérioriser toute la frustration qui m'envahissait quand ma thèse stagnait et reprendre le travail le lendemain avec une nouvelle détermination et parfois mal aux cheveux !

Un merci particulier à Christel qui a eu l'immense courage de relire cette thèse et de corriger les innombrables fautes !!! Une pensée pour Christine, Stéphanie et Nathalie, les Viennoises qui m'auront fait progresser en anglais et merci aussi pour leur accueil lors de ma conférence à Vienne.

Je remercie ma famille sans qui je ne serais pas la personne que je suis aujourd'hui. J'ai eu l'immense chance d'avoir une famille de personnes sincères, chaleureuses et aimantes. Je pense tous particulièrement à mon grand-père Jean et ma grand-mère Mercédès qui chaque année m'apprenaient tant de bonnes choses à Nyons. Vous me manquez. Merci à mes oncles et tantes : Daniel pour son côté révolutionnaire et pour ses talents de cuisinier que j'envie, Anne-Marie pour sa gentillesse, Linda pour sa douceur et Bernard pour son naturel. Merci à eux pour les souvenirs géniaux que j'ai gardé de Givors et de la rivière, ou du Cap d'Agde. Merci à mes cousins avec qui j'ai eu de supers discussions que ce soit

Morgane, Hugo, Manu et maintenant Barbara. Merci aux membres plus éloignés de la famille, Emmanuelle, Antoine et Mélanie pour votre soutien et pour quelques moments passés ensemble que je n'oublierai jamais. Merci à Jean-Manuel pour m'avoir confirmé qu'on peut vivre et allier voyage et travail, et merci aussi pour les paellas en famille. Merci à Marie-Jo, Odette et Raymond que je considère comme mes grands parents de cœurs. Ils m'ont transmis à leur façon une certaine notion du bonheur que je n'oublie pas. Merci à ma grand-mère a toujours été fière de moi et n'a jamais douté de mes capacités. Je tiens aussi à remercier ma belle famille, plus particulièrement Mireille, Sandrine, Alain, Astrie, Michel et Sandrine qui m'ont accueilli à bras ouverts et qui m'ont eux aussi soutenu pendant ces quatre années de thèse.

Tous les mots et les mercis ne suffiraient pas à exprimer la gratitude que j'éprouve envers mes parents et ma sœur. Merci papa d'avoir été mon modèle pendant ces années, j'admire tant ta capacité à assumer dans les moments difficiles. Grâce à toi je n'ai jamais manqué de rien et surtout pas d'amour. Merci à toi maman pour les millions de discussions que nous avons eu et pour ta compréhension, ton amour et ton soutien de tous les instants. Vous m'avez appris le respect de soi et des autres, le partage et l'ouverture d'esprit. Encore une fois, je ne vous remercierai jamais assez. Merci à toi ma 'tite soeurette, ma counette! Merci d'être si différente de moi et pourtant si proche. Tu sais à quel point je tiens à toi et comme je suis fier de toi et de la femme que tu es.

Un dernier merci pour la femme qui partage ma vie, Isabelle, qui m'a soutenu durant ces années de thèse, a supporté mes incessantes soirées chez les potes et mes coups de gueules ou mes pétages de plombs quand je rentrais le soir. Merci d'avoir été là dans les moments où je m'écroulais moralement, et que tu étais la seule à voir. Merci pour tes conseils. Merci à toi de m'apporter ton univers, tes rêves et tes espoirs. Tu sais à quel point cela compte pour moi, scientifique cartésien que je suis! Sans toi, cette thèse n'aurait pas été possible. J'attends avec impatience notre voyage et notre avenir.

Une petite pensée aussi pour ma nourrice et son mari, mes voisins d'Heyrieux qui eux aussi, m'ont permis de grandir et de devenir ce que je suis. Enfin à tous ceux que j'ai oubliés, je suis désolé et je vous remercie.

Je reste persuadé que l'on ne devient pas adulte en grandissant, en passant des diplômes et en s'assurant. On devient adulte un peu plus chaque jour, grâce aux rencontres, aux discussions et aux échanges. J'ai eu la chance de rencontrer énormément de personnes qui m'ont fait grandir un peu et qui, à leur manière, ont toutes participées à l'accomplissement de cette thèse.

À tous un grand merci !

Table des matières

Glossaire	xxi
Introduction	1
1 La vérification de circuit	5
1.1 Introduction	6
1.2 Les méthodes de vérifications	9
1.2.1 Simulation et prototypage	9
1.2.2 Les méthodes formelles	9
1.3 Vérification à base d'assertions	11
1.4 Les langages de spécification de propriétés	13
1.4.1 Le langage PSL	14
1.4.1.1 Naissance et évolution de PSL	14
1.4.1.2 Présentation du langage : syntaxe et sémantique	14
1.4.1.3 La couche Temporelle	15
1.4.2 Présentation des opérateurs FL de PSL	17
1.4.2.1 Sémantique des principaux opérateurs FL	17
1.4.2.2 Les opérateurs FL additionnels	20
1.4.2.3 Classification des opérateurs PSL : retard et ré-entrance	24
1.4.2.4 Le sous-ensemble simple de PSL	25
1.4.3 Les autres langages	25
1.5 Vérification à l'aide de moniteurs synthétisables	26
1.5.1 Approches existantes	26
1.5.2 La plateforme et la méthode Horus	27
1.5.2.1 Moniteurs primitifs	28
1.5.2.2 Les moniteurs complexes	29
1.6 Conclusion sur la vérification de circuits	32
2 État de l'art de la technologie asynchrone	33
2.1 Introduction	34
2.1.1 Principe de fonctionnement et inconvénient des circuits synchrones	34
2.1.2 Principe de fonctionnement des circuits asynchrones	35
2.2 Les communications asynchrones	36
2.2.1 Le canal de communication	36
2.2.2 Représentation de l'information	36

2.2.2.1	Le codage données groupées	36
2.2.2.2	Le codage "insensible aux délais"	36
2.2.3	Les protocoles de communication	37
2.2.3.1	Protocole deux phases	38
2.2.3.2	Protocole quatre phases	38
2.2.4	Implémentation du protocole : la porte de Müller	39
2.3	Modèles de délai et modes de fonctionnement	39
2.3.1	Les modèles de délai	40
2.3.2	Les modes de fonctionnement	40
2.3.2.1	Mode fondamental	40
2.3.2.2	Mode entrée/sortie	40
2.4	Classification des circuits asynchrones	40
2.4.1	Les circuits insensibles aux délais	41
2.4.2	Les circuits quasi-insensibles aux délais	41
2.4.3	Les circuits indépendants de la vitesse	42
2.4.4	Les circuits micro pipelines	42
2.4.5	Les circuits de Huffman	44
2.5	Les aléas	44
2.5.1	Les aléas combinatoires fonctionnels	45
2.5.2	Les aléas combinatoires logiques	45
2.5.3	Les aléas séquentiels	45
2.5.4	Conclusion sur les aléas	45
2.6	Méthodes et outils de synthèse	46
2.6.1	Modélisation et spécification d'un circuit asynchrone	46
2.6.2	Méthodologies de synthèse	47
2.6.3	Les outils d'aide à la conception	49
2.7	Éléments utiles aux implémentations QDI	50
2.8	Conclusion sur la technologie asynchrone	52
3	Les moniteurs asynchrones	53
3.1	Synthèse d'un moniteur asynchrone	54
3.1.1	Interface des moniteurs primitifs asynchrones	54
3.1.2	Interconnexion des moniteurs primitifs asynchrones	54
3.2	Bibliothèque de moniteurs primitifs asynchrones	56
3.2.1	Différences entre moniteurs synchrones et asynchrones	56
3.2.2	Nécessité d'un synchroniseur	58
3.2.3	Lien entre sémantique et implémentation matérielle	60
3.2.4	Les opérateurs sans cycle de retard, sans ré-entrance	60
3.2.4.1	Architecture générique	60
3.2.4.2	Implémentation de la partie "Logique QDI"	61
3.2.5	Les opérateurs sans cycle de retard, avec ré-entrance	62
3.2.5.1	Le phénomène de ré-entrance : l'exemple du <i>until</i> !	62
3.2.5.2	Signal <i>Restart</i> et architecture générique	64
3.2.6	Les opérateurs introduisant des cycles de retard	65
3.2.6.1	Les opérateurs <i>next</i> et <i>next!</i>	65
3.2.6.2	Opérateurs introduisant un ou plusieurs cycles de retard sans ré-entrance	68

3.2.6.3	Opérateurs introduisant un ou plusieurs cycles de retard avec ré-entrance	70
3.2.7	Autres moniteurs primitifs	71
3.3	Validation par simulation numérique	72
3.3.1	Simulation et validation de la bibliothèque de moniteurs primitifs	72
3.3.2	Simulation et validation de la méthode d'interconnexion	75
3.4	Conclusion	75
4	Le synchroniseur	77
4.1	Architecture du synchroniseur	78
4.2	Validation par simulation du <i>Synchro@clk</i>	81
4.2.1	Le <i>Synchro@clk</i> seul	81
4.2.1.1	Le <i>Synchro@clk</i> dans un moniteur	82
4.3	Preuve formelle du synchroniseur	82
4.3.1	L'outil RAT	83
4.3.2	Présentation de la méthode de preuve	83
4.3.3	Description d'une porte	84
4.3.4	Description de l'environnement	84
4.3.5	Ensemble des preuves réalisées	85
4.3.6	Résultats	86
4.4	Validation et caractérisation du synchroniseur	87
4.4.1	Caractérisation en Fréquence du <i>Synchro@clk</i>	87
4.4.2	Caractérisation en aire du <i>Synchro@clk</i>	89
4.5	Conclusion sur le <i>Synchro@clk</i>	90
5	Étude de la robustesse	91
5.1	Caractérisation en surface des moniteurs asynchrones	92
5.1.1	Caractérisation en surface des moniteurs primitifs	92
5.1.2	Croissance linéaire de l'aire en fonction de la complexité de la propriété PSL	94
5.2	Comparaison de la robustesse des solutions synchrones et asynchrones	95
5.2.1	Environnement de simulation et résultats	95
5.2.2	Explication des résultats	98
5.3	Implémentation d'une solution : le <i>Clk stretching</i>	101
5.3.1	Principe	101
5.3.2	Définition d'un nouveau synchroniseur	103
5.3.2.1	Architecture et fonctionnement du <i>Synchro_qdi</i>	103
5.3.2.2	<i>Synchro_qdi_f</i>	105
5.3.3	Contrôle de l'horloge : le <i>Clk_manager</i>	107
5.4	Validation de l'implémentation du "clock stretching"	109
5.4.1	Validation numérique	109
5.4.2	Preuve formelle	112
5.4.3	Validation analogique	113
5.4.4	Caractérisation de la solution	115
5.5	Conclusion sur la robustesse des moniteurs asynchrones	118

6 Conclusion	119
6.1 Bilan	120
6.1.1 Implémentation de moniteurs asynchrones	120
6.1.2 Le "clock stretching"	120
6.2 Perspectives	121
6.2.1 Amélioration de la robustesse d'un circuit synchrone grâce à l'utilisation d'un moniteur asynchrone	121
6.2.2 Moniteurs asynchrones vérifiant des circuits asynchrones	121
6.2.3 Synchronisation entre un domaine synchrone et un domaine asynchrone	122
A Bibliothèque de moniteurs primitifs asynchrones	123
A.1 Les moniteurs asynchrones sans cycles de retard et sans ré-entrance	123
A.1.1 Le moniteur <code>M_imply</code>	123
A.1.2 Le moniteur <code>M_signal</code>	125
A.2 Les opérateurs sans cycles de retard, avec ré-entrance	126
A.2.1 Les moniteurs primitifs <code>M_until</code>	126
A.2.1.1 Les moniteurs primitifs <code>M_until!</code> et <code>M_until</code>	126
A.2.1.2 Les moniteurs primitifs <code>M_until!</code> et <code>M_until_</code>	128
A.2.2 Le moniteur primitif <code>M_always</code>	130
A.2.3 Le moniteur primitif <code>M_never</code>	131
A.2.4 Le moniteur primitif <code>M_eventually!</code>	132
A.2.5 Les moniteurs primitifs <code>M_before</code>	133
A.2.5.1 Les moniteurs <code>M_before!</code> et <code>M_before</code>	133
A.2.5.2 Les moniteurs <code>M_before!</code> et <code>M_before_</code>	134
A.2.5.3 Les moniteurs primitifs <code>M_next_event(b)</code> et <code>M_next_event!(b)</code>	136
A.3 Les moniteurs asynchrones introduisant un ou des cycles de retard et sans ré-entrance	138
A.3.1 Les moniteurs <code>M_next</code> , <code>M_next!</code> , <code>M_next[K]</code> et <code>M_next![K]</code>	138
A.3.1.1 Les moniteurs <code>M_next</code> et <code>M_next!</code>	138
A.3.1.2 Les moniteurs <code>next[K]</code> et <code>next![K]</code>	138
A.3.2 Les moniteurs <code>M_next_a![i..j]</code> et <code>M_next_a[i..j]</code>	140
A.3.3 Les moniteurs <code>M_next_e[i..j]</code> et <code>M_next_e![i..j]</code>	141
A.4 Les moniteurs asynchrones introduisant un ou des cycles de retard et avec ré-entrance	144
A.4.1 Les moniteurs primitifs <code>M_next_event_a!</code> et <code>M_next_event_a</code>	144
A.4.2 Les moniteurs primitifs <code>M_next_event_e!</code> et <code>M_next_event_e</code>	146
A.5 Les moniteurs asynchrones particuliers	149
A.5.1 Le moniteur primitif asynchrone <code>M_imply_b</code>	149
A.5.2 Le moniteur primitif asynchrone <code>M_and</code>	150
A.5.3 Le moniteur primitif asynchrone <code>M_or</code>	152
A.6 Équivalence entre opérateurs PSL et moniteurs primitifs	152
B Le <i>Synchro@clk</i>	155
B.1 Code VHDL comportemental du <i>Synchro@clk</i>	155
B.2 Description PSL des portes utiles à la preuve RAT des synchroniseurs	156
B.2.1 La porte M2RB	156
B.2.2 La porte M3R	156

B.2.3	La porte NOR2A	157
B.2.4	La porte NOR3A	157
B.2.5	La porte NOR3	158
B.2.6	La porte NOR2	158
B.2.7	La porte AND2	159
B.2.8	La porte NAND2	159
B.2.9	La porte INV2	160
B.2.10	La Flip-Flop	160
B.2.11	Le SYNC	161
Bibliographie		163
Publications de l'Auteur		171

Table des figures

1.1	Flot de conception simplifié	6
1.2	Flot de conception complexe des circuits type SoCs	8
1.3	Exemple illustratif : la barrière de parking	12
1.4	Structure en couche du langage PSL	14
1.5	Exemple d'unité de vérification <i>vunit</i>	15
1.6	Illustration des caractères fort et faible en PSL	16
1.7	Trace satisfaisant la propriété $SERE_1$	16
1.8	Chronogramme satisfaisant P_2	18
1.9	Chronogramme satisfaisant P_3	18
1.10	Chronogramme satisfaisant P_4	19
1.11	Chronogramme satisfaisant P_5	20
1.12	Chronogramme satisfaisant P_6	21
1.13	Chronogramme satisfaisant P_7	21
1.14	Chronogramme satisfaisant P_8	22
1.15	Chronogramme satisfaisant P_9	22
1.16	Chronogramme satisfaisant P_{10}	23
1.17	Chronogramme satisfaisant P_{11}	23
1.18	Chronogramme satisfaisant P_{12}	23
1.19	Chronogramme satisfaisant P_{13}	24
1.20	Interface des moniteurs primitifs synchrones	28
1.21	Architecture d'un moniteur primitif synchrone	29
1.22	Arbre syntaxique de la propriété A_1	29
1.23	Moniteur synchrone complexe de A_1	30
1.24	Architecture des moniteurs primitifs synchrones $\rightarrow$ et <i>mnt_Signal</i>	30
1.25	Trace satisfaisant la propriété A_1	31
2.1	Communication de type "poignée de main"	35
	a) "données groupées"	36
	b) 4 états	36
	c) 3 états	36
2.2	Codage de la validité	36
2.3	Chronogramme du protocole 2 phases	38
2.4	Chronogramme du protocole 4 phases WCHB	39
	a) Porte de Müller	39
	b) Porte de Müller dissymétrique	39

2.5	Terminologie des différentes classes de circuits asynchrones	41
2.6	Equivalence entre les modèles QDI et SI	42
2.7	Structure de base des circuits micropipelines	43
2.8	Structure micropipeline avec traitement et registre	43
2.9	Exemple d'aléa statique à '1' pour une porte logique ET	44
2.10	Aléas dynamiques	44
2.11	Aléas de type SIC	45
2.12	Implémentation DIMS de la fonction S	49
2.13	<i>Half Buffer</i> : élément de mémorisation pour le codage double rails	51
2.14	Le SYNC, un élément de synchronisation	51
2.15	Générateur de jetons	51
2.16	Implémentation de la fourche : le bloc <i>Fork</i>	52
3.1	Interface des moniteurs primitifs asynchrones	55
3.2	Implémentation du "OU" double rails asynchrone	55
3.3	Moniteur asynchrone de A_1	56
3.4	Transposition naïve du moniteur primitif <i>M_imply</i> en asynchrone	57
3.5	Codage double rails non respecté	57
3.6	Deux évaluations en un seul cycle	58
	a) Interface du <i>Synchro@clk</i>	58
	b) Chronogramme du <i>Synchro@clk</i>	58
3.7	Spécification du <i>Synchro@clk</i>	58
3.8	Propagation des jetons dans le cas synchrone	59
3.9	Propagation des jetons dans le cas asynchrone	60
3.10	Moniteur primitif asynchrone sans ré-entrance et sans cycle de retard	61
3.11	Implémentation de la partie Logique QDI pour le moniteur primitif $\rightarrow$	62
3.12	Chronogramme satisfaisant P_2	63
3.13	Moniteur asynchrone de P_2	63
3.14	Moniteur primitif avec ré-entrance et sans cycle de retard	64
3.15	Architecture du moniteur primitif <i>M_next</i>	65
3.16	Introduction d'un cycle de retard dans la transition d'un jeton	67
3.17	Moniteur primitif asynchrone <i>M_next</i> [K]	68
3.18	Architecture du moniteur primitif <i>M_next</i> !	68
3.19	Bloc matériel <i>M_next_a</i>	69
3.20	Moniteur primitif asynchrone <i>M_next_a</i> [i..j]	69
3.21	Bloc asynchrone <i>M_next_event</i> K	70
3.22	Moniteur primitif <i>M_next_event</i> (b)[K]	71
3.23	Moniteur primitif <i>G_init</i>	72
3.24	Résultat de simulation pour la propriété P_{15}	74
3.25	Résultat de simulation pour la propriété A_1	76
4.1	Architecture du <i>Synchro@clk</i>	78
4.2	Mise en évidence d'un aléa dans le bloc "codage"	79
4.3	Architecture du <i>Synchro@clk</i>	80
4.4	Chronogramme du <i>Synchro@clk</i>	80
4.5	Structure du banc de test	81
4.6	Résultat de simulation du <i>Synchro@clk</i>	81
4.7	Résultat de simulation de la propriété P_{16}	82

4.8	Ensemble " <i>Synchro@clk</i> + environnement"	85
4.9	Banc de caractérisation du <i>Synchro@clk</i>	88
4.10	Fréquence maximale du <i>Synchro@clk</i> en fonction de Vdd	89
4.11	Le <i>Synchro@clk</i> placé et routé	89
5.1	Croissance linéaire de l'aire avec la complexité des propriétés implémentées	95
5.2	Environnement de simulation	96
5.3	Simulation analogique des moniteurs de A_1	97
5.4	Mise en évidence d'un défaut de robustesse	99
5.5	Problème d'acquiescement des sorties du <i>Synchro@clk</i>	100
5.6	Nouveau système "circuit monitoré + moniteur asynchrone"	102
5.7	Architecture du <i>Synchro_qdi</i>	103
5.8	Fonctionnement du <i>Synchro_qdi</i>	104
5.9	Architecture du <i>Synchro_qdi_f</i>	105
5.10	Fonctionnement du <i>Synchro_qdi_f</i>	106
5.11	Architecture du <i>Clk_manager</i>	107
5.12	Fonctionnement du <i>Clk_manager</i>	108
5.13	Banc de test pour la simulation numérique de A_1	109
5.14	Résultat de la simulation numérique de A_1	110
5.15	Résultat de la simulation numérique de A_1 en fonctionnement dégradé	111
5.16	Environnement de preuve	112
5.17	Banc de simulation analogique	113
5.18	Résultat de simulation analogique en fonctionnement normal	114
5.19	Résultat de simulation analogique en fonctionnement dégradé	114
5.20	Résultat d'évaluation de A_1 en fonctionnement dégradé	116
5.21	Caractérisation en fréquence du moniteur A_1	117
A.1	Moniteur primitif asynchrone sans ré-entrance et sans cycle de retard	123
A.2	Implémentation de la partie "Logique QDI" du <i>M_imply</i>	124
A.3	Implémentation de la partie "Logique QDI" du <i>M_signal</i>	125
A.4	Moniteur primitif asynchrone avec ré-entrance et sans cycle de retard	126
A.5	Implémentation de la partie "Logique QDI" du <i>M_until!</i>	127
A.6	Implémentation de la partie "Logique QDI" du <i>M_until!</i>	128
A.7	Implémentation de la partie "Logique QDI" du <i>M_always</i>	130
A.8	Implémentation de la partie "Logique QDI" du <i>M_never</i>	131
A.9	Implémentation de la partie "Logique QDI" du <i>M_eventually!</i>	132
A.10	Implémentation de la partie "Logique QDI" du <i>M_before!</i>	133
A.11	Implémentation de la partie "Logique QDI" du <i>M_before!</i>	134
A.12	Implémentation de la partie "Logique QDI" du <i>M_next_event!(b)</i>	136
A.13	Architecture du moniteur primitif <i>M_next</i>	138
A.14	Architecture du moniteur primitif <i>M_next!</i>	139
A.15	Architecture du moniteur <i>M_next![K]</i>	139
A.16	Architecture du moniteur primitif <i>M_next_a!</i>	140
A.17	Architecture du moniteur primitif <i>M_next_e!</i>	141
A.18	Implémentation de la partie "Logique QDI" du <i>M_next_e!</i>	142
A.19	Implémentation complète du moniteur <i>M_next_e![i..j]</i>	143
A.20	Implémentation du bloc matériel <i>M_next_event_a!</i>	145
A.21	Implémentation complète du <i>M_next_event_a!(b)[K..L]</i>	145

A.22	Implémentation du bloc <code>M_next_event_e!</code>	146
A.23	Implémentation complète du <code>M_next_event_e!(b)[K..L]</code>	148
A.24	Architecture du moniteur primitif <code>M_imply_b</code>	149
A.25	Architecture du moniteur primitif <code>M_and</code>	150
A.26	Architecture du bloc <code>Or_pending</code>	150
A.27	Architecture d'un moniteur asynchrone du type <i>Expr and Cond</i>	151
A.28	Implémentation de <i>Expr or Cond</i>	152
B.1	La porte M2RB	156
B.2	La porte M3R	156
B.3	La porte NOR2A	157
B.4	La porte NOR3A	158
B.5	La porte NOR3	158
B.6	La porte NOR2	159
B.7	La porte AND2	159
B.8	La porte NAND2	160

Liste des tableaux

1.1	Restrictions du sous-ensemble simple PSL_{ss}	25
1.2	Type pour chaque moniteur primitif de PSL	28
	Moniteurs primitifs asynchrones	93
	Moniteurs primitifs synchrones	93
5.1	Caractérisation en aire des moniteurs primitifs	93
5.2	Ensemble des propriétés PSL placées routées	94
5.3	Éléments utiles au "clock stretching" placés routés	115
A.1	Correspondance matériel et opérateur PSL	153

Glossaire

A

ABV *Assertion Based Verification*

B

BDD *Binary-Decision Diagram*

BSV *BlueSpec System Verilog*

C

CTL *Computational Tree Logic*

CSP *Communicating Sequential Process*

CHP *Communicating Hardwar Process*

CRC *Contrôle de Redondance Cyclique ou Cyclic Redundancy Check*

D

DI *Delay Insensitive*

DIMS *Delay Insensitive Min-term Synthesis*

DES *Data Encryption Standard*

F

FPGA *Field Programmable Gate Array*

FSM *Finite State Machine*

FL *Foundation Language*

FDFB *Fully Decoupled Full Buffer*

G

GaLs *Globally Asynchronous Locally Synchronous*

H

HDL *Hardware Description Language*

HOL *Higher Order Logic*

I

IEEE *Institute of Electrical and Electronics Engineers*

L

LTL *Linear Temporal Logic*

M

MIC *Multiple Input Change*

N

NoC *Network on Chip*

O

OBE *Optionnal Branching Extension*

OVA *Open Vera Assertion Language*

OVL *Open Verification Library*

P

PSL *Property Specification Language*

PSL_{ss} *Sous ensemble simple de PSL*

PVS *Prototype Verification System*

PCHB *Pre-Charged Half Buffer*

PCFB *Pre-Charged Full Buffer*

Q

QDI *Quasi Delay Insensitive*

R

RE *Reset Expression*

RAT *Requieurement Analysis Tool*

S

SERE *Sequential Extended Regular Expression*

SoC *System on Chip*

SVA *SystemVerilog Assertions*

SAT *boolean SATisfiability problem*

SI *Speed Independant*

SIC *Single Input Change*

STG *Signal Transition Graph*

SFINCS *Semi-Formal INstrumentation for Circuits and Systems*

T

TAL *Tima Asynchronous Library*

TLM *Transaction Level Modeling*

V

VHDL *Very high speed integrated circuit Hardware Description Language*

W

WCHB *Weak-Conditionned Half Buffer*

Introduction

Motivations et contexte du travail L'étude et la conception des circuits intégrés numériques a évolué de façon extrêmement rapide aux cours des dernières décennies et il est désormais possible d'intégrer un ou plusieurs systèmes complexes sur une seule puce. Cette évolution n'a été possible que grâce au développement des méthodes de conception des circuits synchrones. En effet, les principes de fonctionnement des circuits synchrones sont simples (synchronisation globale avec un signal d'horloge, délais bornés, ...) et ont permis leur large diffusion. Afin d'obtenir des systèmes toujours plus complexes, les flots de conception se sont fortement complexifiés permettant la mise sur le marché de produit de consommation courante capable de décoder des fichiers vidéo, de décoder des fichiers son, de communiquer par WIFI ou bluetooth, de téléphoner...

Toutefois la complexification des circuits produits a fortement fait augmenter les coûts des phases de test et de validation des circuits. De plus, les outils permettant la conception de circuits synchrones ont évolué bien plus rapidement que les outils permettant la vérification de ces derniers, creusant un peu plus le fossé entre les outils de conception et les outils de vérification. Afin de diminuer le temps nécessaire à la vérification et au test, les étapes de vérification sont couplées aux différentes étapes de conception et distribuées tout au long du flot de conception afin de détecter les erreurs au plus près de leurs sources. Parmi les méthodes de vérification développées, la vérification à l'aide d'assertion (Assertion Based Verification ou ABV) dispose de nombreux avantages. Ces méthodes formelles ou semi formelles permettent de spécifier le comportement d'un circuit ou de son environnement à l'aide de propriétés écrites dans un langage formellement défini. Les propriétés sont ensuite vérifiées statiquement ou dynamiquement sur le circuit lors des différentes étapes de sa conception. Plusieurs travaux menés lors des dix dernières années ont démontré la possibilité et l'intérêt de générer des petits circuits matériels, vérifiant lors du fonctionnement d'un circuit, les propriétés spécifiées au cours de sa conception.

La présence de circuits électroniques dans des applications mettant en jeux la vie de l'utilisateur est une autre conséquence de la diffusion à très large échelle des circuits électroniques. En effet, dans une voiture produite en 2012, il est impensable de n'avoir aucun dispositif électronique. On y retrouve des processeurs pour contrôler l'assistance au freinage, la vitesse du véhicule, l'allumage des feux, la pression des pneus... Si l'un de ces dispositifs électroniques critiques présente un comportement fautif, un accident peut survenir et l'utilisateur être blessé. L'utilisation de l'ABV apparaît être une solution pour améliorer la sécurité des usagers. Par exemple, le fait d'embarquer un circuit matériel couplé à l'application critique pour vérifier des propriétés permet au circuit vérifiant les propriétés de détecter une erreur lorsqu'elle survient dans le circuit critique. Ainsi, il sera

possible de déclencher une contre mesure afin d'éviter que l'erreur détectée dans le circuit n'ait de grave conséquence sur la santé de l'utilisateur.

Jusqu'alors, les travaux permettant de générer les circuits embarqués pour la vérification de propriété ne proposent que des solutions utilisant les principes de conception des circuits synchrones. Or, si le circuit de vérification et le circuit à vérifier utilisent la même technologie (synchrone) et sont embarqués sur la même puce, ils sont susceptibles de présenter les mêmes comportements fautifs (violation de chemin critique...). Dans ce cas, le résultat fourni par le circuit de vérification peut être faux et l'utilisateur mis en danger. Il est donc nécessaire d'élever le niveau de confiance que l'on accorde au circuit effectuant la vérification des propriétés. Une manière d'y parvenir est l'utilisation de circuits reconnus pour leur robustesse, les circuits asynchrones et plus particulièrement les circuits quasi-insensibles aux délais.

Contribution et plan du manuscrit Le travail présenté dans ce manuscrit s'inscrit dans le projet ANR SFINCS (Thales, Dolphin Integration, TIMA) et doit permettre de répondre aux problèmes de robustesse des circuits de vérification. Ce travail s'appuie sur la méthode de génération de circuits de vérification synchrones développée au laboratoire TIMA et implémentée sur la plateforme logicielle **Horus**.

Le premier chapitre de ce manuscrit présente les concepts et méthodes permettant la vérification et plus particulièrement l'ABV. Ce chapitre présente aussi la sémantique du langage Property Specification Language (PSL) à l'aide duquel sont écrites les propriétés spécifiant le comportement du circuit à vérifier. Enfin, la méthode implémentée dans **Horus** et permettant la synthèse de circuits de vérification synchrones est présentée plus en détails et comparée aux solutions existantes.

Le second chapitre présente les bases de la conception de circuits implémentés en technologie asynchrone. En effet, les circuits asynchrones n'étant pas synchronisés globalement à l'aide d'un signal d'horloge, ils doivent déployer d'autres méthodes de synchronisation. Ces mécanismes de synchronisation locale permettent de rendre le système indépendant du temps de calcul dans chaque bloc. Des hypothèses sont faites sur les modèles de délais dans les portes et les connexions permettant de simplifier la conception des circuits asynchrones mais aussi de classer ces derniers. Ce chapitre donne une vision résumée de la notion de canal communiquant, des différents protocoles permettant la synchronisation locale, et plus généralement des différents paramètres qui permettent d'établir un classement des diverses implémentations asynchrones possibles. Enfin, les différentes méthodes permettant de spécifier et de synthétiser un circuit asynchrone ainsi que certains des outils implémentant ces méthodes sont présentés dans cette introduction à la technologie asynchrone.

Après étude des différentes méthodes permettant l'ABV, le choix de la méthode permettant la synthèse de circuit de vérification s'est porté sur celle implémentée dans **Horus**. Cette méthode basée sur une bibliothèque de circuits élémentaires et sur une méthode d'interconnexion permet de tirer parti de la modularité et de la facilité d'interconnexion des circuits asynchrones. Le troisième chapitre présente les modifications réalisées pour permettre la génération de circuits de vérification asynchrone. Il a fallu tout d'abord redéfinir l'interconnexion des blocs élémentaires et plus précisément les signaux d'entrée et de sortie des blocs élémentaires. La bibliothèque de circuits élémentaires a ensuite dû être entièrement redéfinie pour permettre des implémentations asynchrones.

Le travail réalisé dans le troisième chapitre montre aussi la nécessité de synchroniser

le circuit synchrone à vérifier avec le circuit asynchrone effectuant la vérification. Le quatrième chapitre présente l'implémentation de ce synchroniseur et le bon fonctionnement de ce dernier. Afin d'obtenir une implémentation asynchrone robuste, la preuve formelle du fonctionnement du synchroniseur est réalisée à l'aide de l'outil RAT. Cette preuve démontre que le bon fonctionnement d'un circuit asynchrone QDI synchronisé avec un circuit synchrone peut être fortement impacté par les hypothèses temporelles formulées sur les circuits synchrones.

Le cinquième chapitre illustre l'impact des hypothèses temporelles héritées des circuits synchrones sur le fonctionnement d'un circuit de vérification asynchrone. Afin de garantir la robustesse des circuits asynchrones, il faut relâcher les contraintes temporelles à l'interface des domaines synchrones et asynchrones. L'objet de ce chapitre est l'implémentation d'une solution, le "clock stretching", permettant d'étirer l'horloge du circuit synchrone lorsque cela est indispensable au bon fonctionnement du circuit asynchrone. Cette solution est ensuite validée par simulation dans un premier temps, puis grâce à la preuve réalisée avec RAT.

Enfin, le dernier chapitre de ce manuscrit conclut sur ces travaux et présente quelques pistes de recherche s'appuyant sur des éléments et des réflexions issues du travail sur les circuits de vérification asynchrones.

La vérification de circuit

Sommaire

1.1 Introduction	6
1.2 Les méthodes de vérifications	9
1.2.1 Simulation et prototypage	9
1.2.2 Les méthodes formelles	9
1.3 Vérification à base d'assertions	11
1.4 Les langages de spécification de propriétés	13
1.4.1 Le langage PSL	14
1.4.2 Présentation des opérateurs FL de PSL	17
1.4.3 Les autres langages	25
1.5 Vérification à l'aide de moniteurs synthétisables	26
1.5.1 Approches existantes	26
1.5.2 La plateforme et la méthode Horus	27
1.6 Conclusion sur la vérification de circuits	32

1.1 Introduction

Depuis l'invention du circuit intégré par Jack Kilby en 1958, le nombre de transistors par puce n'a cessé d'augmenter jusqu'en 2001 en respectant les prévisions de la loi de Moore. Doubler le nombre de transistors par puce a d'abord été possible grâce aux évolutions technologiques telle que l'amélioration de la finesse de gravure des transistors sur le silicium. Les transistors de plus en plus petits ont permis une augmentation des performances des circuits intégrés : augmentation de la fréquence, diminution de la consommation... Cependant, à partir de 2004, cette croissance exponentielle a été ralentie à cause des problèmes de dissipation thermique. Afin de continuer à développer des circuits toujours plus complexes, les industriels ont commencé à produire des systèmes plus complexes, les systèmes sur puce (SoC), où plusieurs composants (processeur, GPU, mémoires...) sont intégrés sur une même puce et communiquent grâce à des bus ou des NoCs.

Toutes ces évolutions dans la conception des circuits intégrés permettent la production d'un grand nombre de circuits aux fonctionnalités diverses et pour des coûts de plus en plus faibles. Grâce à cela l'électronique a largement été diffusée au niveau du grand public, à un point tel que son usage est devenue indispensable. Aujourd'hui, on trouve plusieurs dizaines de processeurs dans un produit aussi courant que l'automobile. Ces processeurs contrôlent l'aide au freinage, le régulateur de vitesse, la direction... L'utilisateur est donc dépendant de ces processeurs et de leur bon fonctionnement. Afin d'assurer au mieux la sécurité de l'utilisateur, il est nécessaire de garantir la fonctionnalité correcte des systèmes embarqués, surtout lorsque ces derniers sont utilisés dans des applications mettant en jeu des vies humaines (aéronautique, automobile, santé). Des phases de vérifications sont donc nécessaires lors des différentes étapes du flot de conception d'un circuit intégré. La figure 1.1 illustre le principe du flot de conception de circuit.

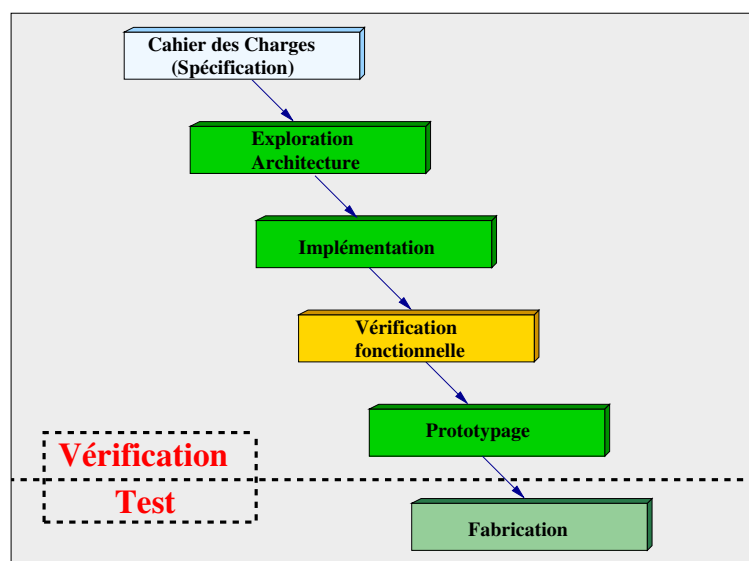


FIGURE 1.1 – Flot de conception simplifié

Tout d'abord, une exploration d'architecture est réalisée pour analyser de nombreux paramètres tels que :

- partitionnement logiciel/matériel

- consommation
- fréquence de fonctionnement
- température
- coût du système

Une fois cette étape terminée, l'implémentation des parties logicielles et une description des parties matérielles sont effectuées. A cette étape du flot de conception, les descriptions sont réalisées au niveau d'abstraction numérique. Une phase de vérification est alors nécessaire pour s'assurer que la fonctionnalité du système respecte la spécification initiale. Si c'est effectivement le cas, une étape de prototypage est réalisée pour tester le circuit en condition de fonctionnement réel afin de prendre en compte les effets analogiques. Cela permet de valider la fonctionnalité du circuit lorsque que l'on passe d'un niveau d'abstraction à un autre plus faible. Enfin, après des étapes d'optimisations et de vérifications supplémentaires, la liste de portes est envoyée au fondeur pour produire le circuit en série.

L'amélioration des outils de conception et une montée dans les niveaux d'abstraction sont deux facteurs permettant de maîtriser la complexité des circuits. Cependant, les outils de vérification n'ont pas suivi cette tendance et le fossé entre outils de conception et outils de vérification ne cesse de s'accroître. Pour pallier cette tendance, les flots de conception se sont fortement complexifiés. La figure 1.2 illustre le flot de conception toujours plus complexe des SoCs. Ces flots s'appuient sur deux principes :

- Conception modulaire : chaque composant d'un système sur puce est développé indépendamment des autres. Cela permet de réutiliser des composants déjà disponibles sur le marché mais aussi une vérification simplifiée car on vérifie un composant ayant une complexité plus faible que le système complet.
- Couplage de la vérification et de la conception : les étapes de vérification sont distribuées tout au long du flot.

Une différenciation doit être faite entre la vérification et le test. Le test regroupe un ensemble de techniques appliquées après fabrication pour s'assurer que chaque exemplaire produit est fonctionnellement correct. Ces techniques permettent de détecter un court-circuit, un transistor défectueux, *etc.* La vérification s'effectue en amont de l'étape de fabrication du circuit et permet de s'assurer qu'à chaque étape du flot de conception, la fonctionnalité du circuit est préservée et respecte la spécification initiale. Nous allons maintenant présenter les diverses méthodes de vérification.

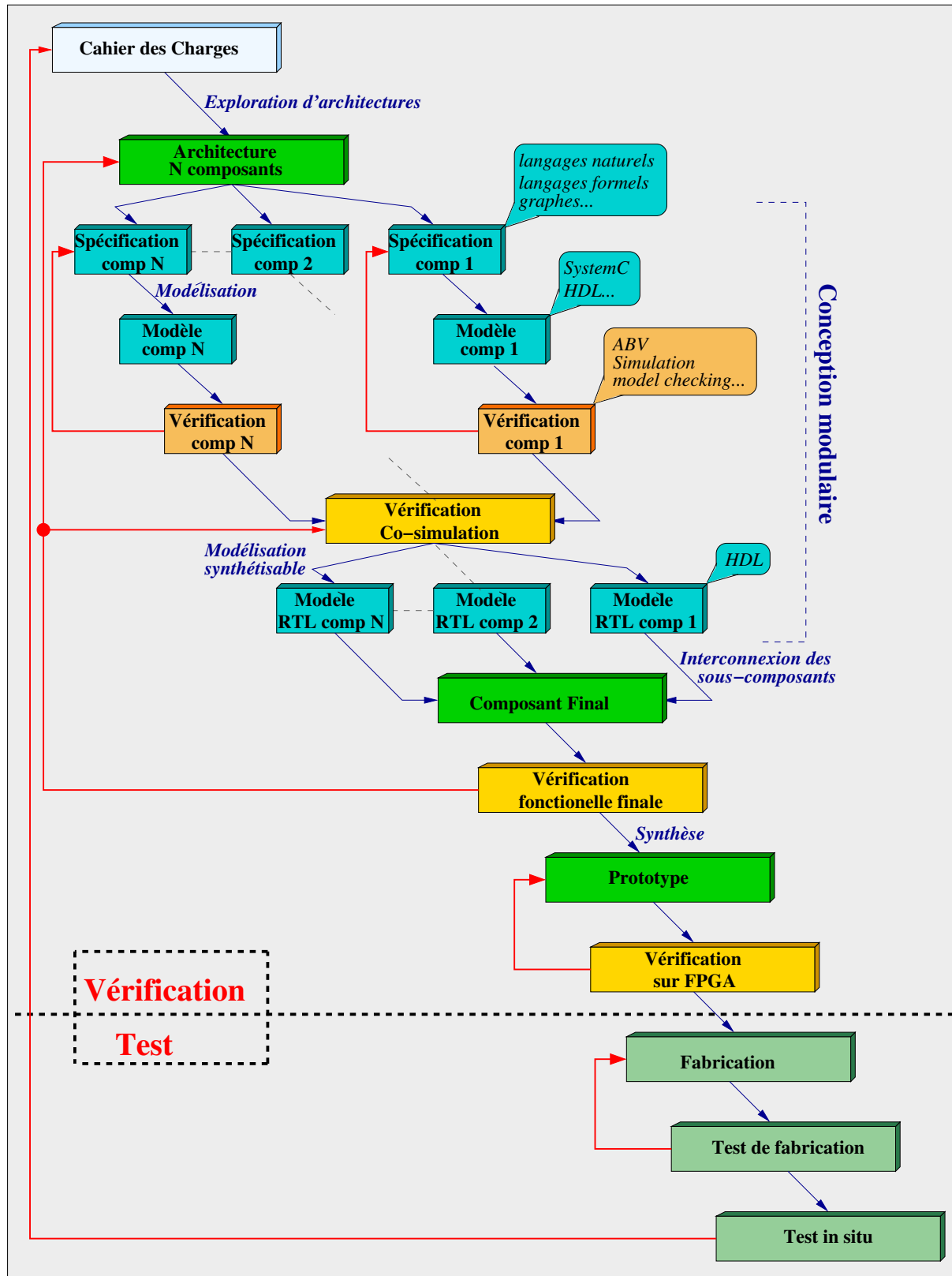


FIGURE 1.2 – Flot de conception complexe des circuits type SoCs

1.2 Les méthodes de vérifications

De nombreuses méthodes sont disponibles pour la vérification des circuits. Elles sont classées en deux groupes : les méthodes formelles et les méthodes plus classiques (simulation et prototypage).

1.2.1 Simulation et prototypage

La simulation est la technique de vérification la plus répandue depuis le développement des langages de description de matériel (HDL). Facilement applicable à toutes les étapes du flot de conception, elle permet une vérification efficace et simple à mettre en œuvre. Toutefois la complexité grandissante des systèmes à simuler est telle que le temps de simulation explose et que la puissance de calcul nécessaire devient considérable, ce temps de simulation pouvant facilement dépasser plusieurs jours. Le coût de la simulation devient donc un paramètre critique dans le développement des circuits sur puce.

Une alternative a été apportée grâce aux circuits programmables de type FPGA (Field Programmable Gate Array). Ils permettent de tester sur un prototype la description HDL d'un système. Cette approche augmente considérablement la rapidité de la vérification comparée à une simulation classique. De plus, le développement d'outils dédiés aux prototypage FPGA a permis à cette approche de s'intégrer dans de nombreux flots de conception depuis plusieurs années. Aujourd'hui, les industriels utilisent des approches hybrides mixant simulation, émulation et/ou prototypage. Par exemple, les parties critiques du système à vérifier sont implémentées sur un FPGA et les composants non critiques pour l'application ou dont le comportement est déjà garanti sont simulés directement.

Le principal inconvénient de ces méthodes de vérification dites "classiques" est qu'il est impossible de tester un système de manière exhaustive. La complexité de la vérification augmente exponentiellement avec la complexité des systèmes. Pour vérifier une simple RAM de 2Mb de manière exhaustive, il faudrait vérifier les 2^{2048} états possibles. Or même si la vérification d'un état ne dure qu'une nanoseconde, il faudrait plusieurs millions d'années. Les méthodes de vérification classique sont donc incapables de garantir la fonctionnalité correcte d'un système à 100% mais reste une approche fournissant une évaluation rapide et facile de la fonctionnalité du composant. Pour les systèmes critiques, d'autres méthodes de vérification doivent être envisagées.

1.2.2 Les méthodes formelles

Ces méthodes de vérifications s'appuient sur des théories mathématiques. Elles permettent une vérification exhaustive d'un circuit sous réserve de disposer d'une puissance de calcul suffisante car les opérations de vérifications formelles explosent avec l'augmentation de la complexité du circuit. De plus, ces techniques ne sont généralement pas automatiques. Depuis une dizaine d'années, de nombreuses équipes de recherche concentrent leurs efforts pour améliorer l'efficacité et la facilité d'utilisation de telles méthodes. Les techniques de vérifications formelles sont divisées en quatre groupes principaux.

Vérification d'équivalence (Equivalence-checking) Cette méthode utilise un modèle de référence ("golden model"). Les sorties du circuit à vérifier sont ensuite comparées aux sorties du modèle de référence avec des portes XOR par exemple. Pour que la vérification soit réussie, il faut que le circuit à vérifier et le modèle de référence aient les mêmes

sorties finales lorsqu'ils possèdent les mêmes signaux d'entrées, ce qui implique que les sorties des portes XOR doivent toujours être inactives. On dit alors que le circuit est équivalent au modèle de référence. Cette approche peut-être effectuée à l'aide de techniques formelles basées sur les diagrammes de décision binaire (BDD) ou des approches de type SAT (*boolean SATisfiability problem*) [KPKG02, PK00]. On extrait les BDDs du modèle de référence et du circuit à vérifier puis ils sont comparés afin de s'assurer que les deux circuits sont équivalents. Pratiquement, la vérification d'équivalence est utilisée à divers niveaux d'abstraction et peut servir à vérifier la fonctionnalité d'un circuit lors du passage d'une étape du flot de conception à une autre, par exemple pour des conversions FPGA-ASIC (niveau porte/niveau porte) ou pour des migrations de langages (VHDL/Verilog)...

Simulation symbolique La simulation utilisée dans cette technique ne s'effectue pas sur des valeurs explicites de données mais sur des ensembles de valeurs abstraites [Ber06]. Si on veut simuler de manière exhaustive un additionneur simple avec les entrées A , B et une sortie C , il faut considérer toutes les combinaisons possibles des entrées ce qui est impossible en pratique. En simulation symbolique, les entrées sont spécifiées de manière ensembliste.

Vérification de modèle (Model-checking) Le principe de la vérification de modèle puise ces fondements dans les travaux de Edmund M. Clarke et E. Allen Emerson menés sur le Model-checking temporel en 1981 ainsi que dans les travaux de Jean-Pierre Queille et Joseph Sifakis en 1982 [CES86]. Dans la vérification par Model-checking, les spécifications du circuit à vérifier sont exprimées dans des logiques temporelles appelées CTL (Computational Tree Logic) ou LTL (Linear Temporal Logic). Une propriété temporelle est une propriété se déroulant dans le temps. La propriété *toute requête d'accès à une ressource est satisfaite au maximum 10 cycles d'horloges après son émission* est un exemple de propriété temporelle. Le circuit pour lequel on souhaite vérifier que la propriété est vraie est modélisé à l'aide d'un automate à états finis. L'outil de Model-checking parcourt ensuite toutes les exécutions possibles du système en traversant tous les chemins de l'automate à partir d'un état initial et vérifie que les propriétés temporelles sont respectées pour tous les états atteignables. On parle de preuve de propriétés.

La logique CTL est utilisée pour exprimer des propriétés sur plusieurs, voire toutes, les exécutions possibles du circuit à vérifier. Cette logique ne peut être utilisée qu'avec des outils de vérification statique, c'est-à-dire qui parcourt le modèle du circuit sans exécuter explicitement celui-ci.

La logique LTL est linéaire et permet d'exprimer des propriétés relatives à une seule exécution du circuit, l'exécution courante. Il est donc possible, en plus de la vérification statique, de vérifier des propriétés en même temps que l'exécution du circuit : on parle de vérification dynamique. L'historique complet des logiques CTL et LTL est donné dans [Var06].

Plus le système à vérifier est complexe, plus l'automate le modélisant comporte d'états et le parcours de tous ces états provoque une explosion combinatoire. Pour les systèmes industriels il est donc impossible de vérifier de manière exhaustive le système. De nombreuses techniques dérivant du Model-checking classique ont vu le jour pour pallier cet inconvénient majeur :

- Le Model-checking borné permet de vérifier de manière exhaustive une partie de l'automate en restreignant le nombre de cycles analysés [BCC⁺03].

- Le Model-checking orienté permet de sélectionner les chemins à parcourir et permet ainsi d'orienter l'exécution à vérifier.
- Le Model-checking symbolique utilise les concepts de la simulation symbolique. Les états de l'automate initial modélisant le circuit sont représentés par un ensemble abstrait d'états. Cela permet d'obtenir une abstraction du modèle initial sur lequel on effectue le Model-checking [McM93]. De nombreuses méthodes de représentation d'ensembles d'états ont vu le jour. La plus connue utilise les BDDs.

Preuve de théorèmes Contrairement à la preuve de propriétés, la preuve de théorèmes n'est pas totalement automatique et nécessite plus ou moins d'interventions de la part de l'utilisateur. Cette vérification utilise un ensemble d'axiomes et un modèle du circuit afin d'effectuer des preuves sur une modélisation du circuit à vérifier. L'un des principaux avantages de cette méthode est la capacité à formaliser le comportement du circuit à vérifier à différents niveaux d'abstractions (pas de limitation au niveau booléen, possibilité de prendre en compte des circuits paramétrés). Cela permet de réaliser des vérifications sur différents niveaux d'abstractions du modèle de circuit. De plus, cette approche permet des raisonnements logiques permettant de prouver formellement des comportements du circuit qui peuvent être impossibles à prouver avec d'autres techniques. Par exemple, cette méthode peut traiter de gros systèmes ayant des espaces d'états infinis et peut permettre la vérification totale de systèmes complexes comme le processeur Intel Core i7 [KGN⁺09] ce qui est impossible avec les méthodes de Model-checking.

De nombreux prouveurs de théorèmes existent, chacun ayant ces propres caractéristiques :

- **Applicative Common Lisp2 (ACL2)** : cet outil est un démonstrateur inductif basé sur une logique du premier ordre avec induction. Ce démonstrateur automatique utilise un moteur d'enchaînement de règles de déduction et des stratégies pour les appliquer dans un ordre donné. ACL2 est non typé et basé sur Lisp ce qui rend sa logique exécutable.
- **Higher Order Logic (HOL)** : cet outil utilise une logique typée d'ordre supérieure qui n'est pas exécutable. Comme pour ACL2, les règles qui ont été prouvées peuvent ensuite être réutilisées comme règles de base. L'utilisateur doit aussi guider manuellement le démonstrateur inductif de HOL en choisissant les stratégies de démonstrations durant le processus de vérification.
- **Prototype Verification System (PVS)** : cet outil reprend les bases conceptuelles de HOL et ACL2 et utilise une logique fortement typée et qui n'est pas exécutable. Ce démonstrateur inclut de plus un vérificateur de propriétés pour des automates à états finis.

Cette technique, considérée comme la plus rigoureuse, demande en contrepartie un effort considérable de la part du concepteur. Un ingénieur souhaitant utiliser les prouveurs de théorèmes doit d'abord suivre de longs mois de formation. En effet, les outils permettant la preuve de théorèmes doivent être guidés au moment de la vérification et la description formelle du circuit n'est pas toujours triviale à obtenir.

1.3 Vérification à base d'assertions

Lors de la conception d'un circuit intégré, plus une erreur est découverte tard dans le flot, plus le coût de cette dernière sera important. Il est donc indispensable de découvrir les

erreurs de conception le plus tôt possible dans le flot de conception. Ces dernières années, une méthode à la popularité grandissante a fait son apparition dans les milieux industriels et académiques : la **Vérification à Base d'Assertions** (Assertion Based Verification ou ABV) [FKL03]. Une assertion est une description concise d'un comportement complexe que le circuit doit respecter.

Dans les approches classiques de test, lorsqu'une erreur est détectée il faut ensuite en isoler la cause à l'aide d'une analyse longue et coûteuse. L'ABV permet de considérablement diminuer le temps nécessaire à la phase de test d'un circuit car les éléments de vérification sont directement couplés aux éléments de conception. L'utilisation d'assertions permet donc de capturer les erreurs au plus près de leur source. Ainsi le gain de temps au niveau du debug est très important.

Afin d'illustrer nos propos nous considérerons par la suite un système simple : la barrière de parking (c.f. figure 1.3).

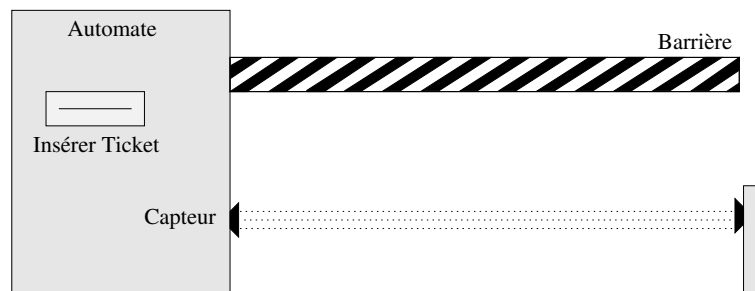


FIGURE 1.3 – Exemple illustratif : la barrière de parking

Ce système possède le comportement suivant :

- Le paiement du parking est effectué sur une autre borne et permet d'obtenir un ticket valide pour sortir.
- Lorsque qu'un ticket valide (signal *ticket*='1' et signal *valide*='1') est introduit la barrière doit s'ouvrir (signal *open*='1').
- Lorsqu'un ticket non valide est introduit (signal *ticket*='1' et *valide*='0'), la barrière doit rester fermée (signal *open*='0') et le ticket doit être rendu à l'utilisateur (signal *ticket_out*='1').
- Lorsque la barrière est ouverte, si une voiture passe devant le capteur (signal *capteur*='1'), la barrière doit se fermer dès que la voiture est passée (signal *fin_passage*='1').

La modélisation d'un circuit à l'aide d'assertions permet au concepteur d'avoir une première réflexion sur ce qu'il est en passe d'implémenter en plus de permettre l'amélioration de la phase de test. L'ABV utilise deux types de propriétés distinctes pour modéliser le circuit à tester :

- **Les Hypothèses** (ou Assumption) : elles décrivent le fonctionnement de l'environnement du circuit à tester. Plus précisément, les hypothèses expriment les contraintes sur les entrées du circuit à vérifier. Dans le cas de la barrière de parking, l'hypothèse suivante peut être faite :

Hypothèse **H1** : *Un ticket ne peut pas être inséré alors qu'une voiture est en train de passer.*

- **Les Assertions** : elles décrivent les comportements que le composant doit assurer. Ces propriétés portent sur les signaux internes ou les sorties du circuit à vérifier. Dans le cas de la barrière de parking, l'assertion suivante peut être effectuée :

Assertion **A1** : *Quand un ticket valide est inséré, la commande d'ouverture de la barrière doit être activée un cycle plus tard et rester ouverte jusqu'à ce que le véhicule soit passé.*

Un ensemble complet d'hypothèses et d'assertions permet de définir un environnement complet de vérification où les comportements du circuit à vérifier sont analysés lorsque l'environnement respecte les protocoles de communication avec celui-ci.

En plus des avantages précédemment énoncés, l'ABV définit un cadre de travail appelé **Assum-Garantee Paradigm** qui permet une vérification hiérarchique d'un système complexe. Le principe est de vérifier au mieux un composant à l'aide de propriétés, puis lorsque le composant est validé, de l'inclure dans un système plus complexe. La vérification du système complexe est alors réduite à la vérification des protocoles de communication entre l'environnement et le composant pré-vérifié.

1.4 Les langages de spécification de propriétés

Lors de la conception d'un circuit, un document de départ fiable est nécessaire pour décrire les spécifications que le circuit doit respecter. La spécification d'un circuit est désormais quasi-impossible à l'aide d'un langage naturel. Trop ambigu et pas assez précis, le langage naturel laisse les ingénieurs de conception interpréter les spécifications et souvent le circuit obtenu est non conforme aux exigences.

Parallèlement, bien avant le développement des techniques d'ABV, le couplage entre les éléments formels de vérification et de conception a été introduit dans des outils de conceptions standards. Depuis 1987, le langage VHDL permet d'embarquer directement dans le code source des assertions simples (réduites à des propriétés booléennes) supportant la vérification du circuit. Si l'on reprend l'exemple de la barrière de parking et que l'on considère qu'il existe un signal *barriere* à '1' quand la barrière doit être levée et un signal *ticket_valide* à '1' quand un ticket valide est inséré dans l'automate, alors on peut écrire :

```
ASSERT (ticket_valide = 1 IMPLIES barriere = 1)
REPORT "problème de barrière" SEVERITY ERROR;
```

Actuellement, tous les simulateurs VHDL supportent ces assertions. L'outil de simulation vérifie continuellement leur validité et renvoie un message d'erreur en cas de violation d'une ou plusieurs assertions. Ces assertions restent limitées à des propriétés booléennes, n'exprimant que des combinaisons de signaux valides ou non. Cela est parfaitement adapté aux parties combinatoires du circuit mais ne permet pas de décrire correctement les parties séquentielles. Dans ce domaine, les relations de causalité dans le temps sont primordiales pour le bon fonctionnement du circuit. Dans le cas de la barrière de parking, il est essentiel de vérifier que le signal *barriere* reste à '1' tant que le signal *capteur* provenant du capteur de passage de voiture n'a pas indiqué que la voiture a fini de passer. Sans quoi la barrière pourrait se refermer sur l'automobile et blesser le conducteur. Ces propriétés où le temps entre en jeu sont nommées "propriétés logico-temporelles". L'évaluation de booléens dans le temps est apparue grâce aux langages de spécification de propriétés temporelles comme CTL, LTL, etc. Par la suite, reposant sur ces deux langages, d'autres langages ayant une syntaxe plus facile à lire et à écrire ont fait leur apparition.

La suite présente quelques uns de ces langages et plus particulièrement le langage Property Specification Language (PSL).

1.4.1 Le langage PSL

1.4.1.1 Naissance et évolution de PSL

Le langage PSL est une évolution du langage Sugar d'IBM créé au début des années 1995¹. PSL a été normalisé par Accelera en 2003 puis par l'IEEE en 2005 [FWMG05]. La dernière révision de la norme date de 2010 [20110]. PSL permet d'écrire des propriétés logico-temporelles en adressant trois domaines liés à la vérification de circuit [EF06] :

- spécification : fournit un langage structuré. La sémantique, définie formellement, permet d'éviter toute ambiguïté liée aux langages naturels.
- vérification formelle : permet le model-checking.
- vérification dynamique : analyse des propriétés en parallèle de la simulation fonctionnelle HDL.

1.4.1.2 Présentation du langage : syntaxe et sémantique

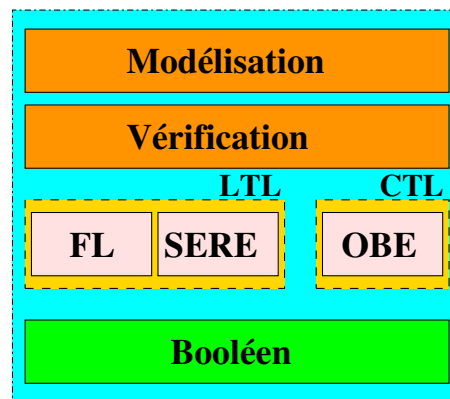


FIGURE 1.4 – Structure en couche du langage PSL

Le langage PSL supporte la syntaxe de cinq langages dont SystemC, VHDL, SystemVerilog et Verilog. PSL est structuré en 4 couches (*c.f.* figure 1.4) :

- **Booléenne** : regroupe les expressions booléennes dont les valeurs sont réduites à vrai ('1') et faux ('0'). Cette couche est construite sur les opérateurs booléens classiques : {and, not, or $\rightarrow$ }. Par exemple dans la propriété H1_barriere de la figure 1.5, `not(ticket and capteur)` appartient à la couche booléenne.
- **Temporelle** : cette couche représentant le cœur de PSL, est formée des trois ensembles : **FL** (Foundation Language), **SERE** (Sequential Extended Regular Expressions) et **OBE** (Optionnal Branching Extension). Les deux premiers ensembles sont basés sur la logique LTL alors que **OBE** utilise la logique CTL pour la vérification statique. La couche temporelle définit les relations entre les expressions booléennes au cours du temps. Dans la propriété A1_barriere de la figure 1.5, `always`, `next!` et `until!` sont des opérateurs appartenant à la couche **FL**. La propriété A1_barriere est elle-même une **FL**. Un exemple détaillé de **SERE** est donné dans la suite (*c.f.* figure 1.7).
- **Vérification** : fournit à l'aide de mots clés les directives précisant comment utiliser les propriétés lors de la vérification. On peut citer les mots clés **assert** (la propriété

1. Documentation : http://www.pslsugar.org/technical_paper.html

doit être vérifiée), **assume** (définit le comportement que les entrées doivent vérifier), **cover** (pour les calculs de couverture) ou **restrict_garantee**. Le rôle des mots clés **assert** et **assume** est illustré dans la figure 1.5.

- **Modélisation** : définit le modèle d’environnement de vérification (contraintes sur les entrées, affectation de variables auxiliaires...). L’environnement et les propriétés sont généralement encapsulés dans une structure appelée **vunit** (c.f. figure 1.5).

```
vunit {Barrière_spec}(Barrière){
  default clock is rising_edge(Clk);
  property A1_barriere is assert always((ticket and valide) → (next! (open un-
  til!_ fin_passage)));
  property H1_barriere is assume always (not(ticket and capteur));
}
```

FIGURE 1.5 – Exemple d’unité de vérification **vunit**

1.4.1.3 La couche Temporelle

Le sous ensemble FL cet ensemble contient notamment les opérateurs temporels suivants : **always**, **eventually!**, **before**, **before_**, **until**, **until_**, **next**, **next[k]**, **next_a[k :l]**, **next_e[k :l]**, **next_event**, **next_event[k]**, **next_event_a[k :l]**, **next_event_e[k :l]**. La signification et l’utilisation de ces opérateurs sont présentées dans [20110] et des détails sur ces opérateurs sont fournis dans la suite. De plus, la signification de certains de ces opérateurs sera abordé plus tard dans ce manuscrit.

Certains opérateurs comme l’opérateur **next** sont dit bornés car ils ont une condition de terminaison future connue et fixe. Les autres opérateurs comme l’opérateur **eventually!** sont dits non borné car il est a priori impossible de connaître la date d’occurrence de la condition de terminaison.

PSL fournit deux déclinaisons pour certains opérateurs temporels : **forts** et **faibles**. On précise le caractère fort d’un opérateur en rajoutant **!** à la fin de l’opérateur. La relation entre la force d’un opérateur et la satisfaction d’une propriété est ainsi définie dans la norme :

- Soit P une propriété et P_s la même propriété où tous les opérateurs sont dans leur version forte.
- La propriété P "Holds Strongly" sur une trace finie si et seulement si P_s "Holds" sur cette trace.
- Si la condition de terminaison de P_s n’est pas rencontrée et que la trace est terminée, alors P_s "Failed".

Pour toutes propriétés PSL, on peut définir les états suivants :

- **Holds Strongly** : la propriété est satisfaite fortement et toute extension de la trace conserve cette caractéristique.
- **Holds** : la propriété est satisfaite faiblement, il peut exister une extension de la trace violant la propriété.
- **Pending** : La propriété est en cours de vérification. Cet état n’a de sens que pour les propriétés comprenant un ou des opérateurs forts.
- **Failed** : la propriété a été violée et toute extension de la trace vérifiera cette caractéristique, ou la propriété est forte et la vérification n’est pas terminée.

On reprend l'exemple de la barrière de parking. On suppose que quand un ticket valide est inséré, le signal de commande de la barrière est mis à '1' au cycle suivant. Cela peut s'écrire :

```
property P1 is assert always((ticket and valide) → next open);
property P1S is assert always((ticket and valide) → next! open); (version forte)
```

Les différences de satisfaction entre P_1 et P_{1S} sont explicitées grâce aux chronogrammes de la figure 1.6. Au cycle #0, P_{1S} et P_1 sont dans l'état **Holds** car la partie gauche de

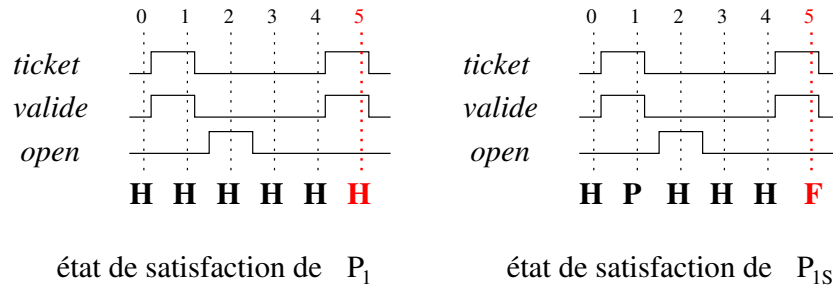


FIGURE 1.6 – Illustration des caractères fort et faible en PSL

l'implication est fausse. Les propriétés sont vérifiées faiblement car il peut exister un prolongement de la trace les violant. Au cycle #1, la partie gauche de l'implication est vraie, la propriété n'est pas violée mais toutes les conditions de terminaisons n'ont pas été rencontrées. P_1 est donc **Holds** et P_{1S} est **Pending**. Au cycle #2 la partie droite de l'implication est vraie et les propriétés sont de nouveaux dans l'état **Holds**. Au cycle #5, la partie gauche est vraie de nouveau. P_1 et P_{1S} sont donc en cours d'évaluation et il faut vérifier $open='1'$ au prochain cycle. Si la simulation est stoppée au cycle #5 alors que les propriétés sont en cours d'évaluation. La version forte de la propriété n'a pas rencontré de condition de terminaison, elle est donc dans l'état **Failed** alors que la version faible est dans un état **Holds**.

Le sous-ensemble SERE ce sous ensemble, formé d'expressions régulières étendues, permet d'exprimer des propriétés sur des séquences de signaux. Il est composé des expressions régulières classiques dont certaines sont explicitées dans l'exemple à venir :

- Séquentialité : $\{ ; , : \}$
- Répétitions consécutives : $\{ [*] , [* i] \}$
- Opérateurs complexes de répétitions non consécutives : $\{ [- > i] , [= i] \}$

La propriété $SERE_1$ est un exemple de SERE :

```
property SERE1 is assert { { a ; b } [ 3 * ] | → { c [ → 2 ] : d } } ;
```

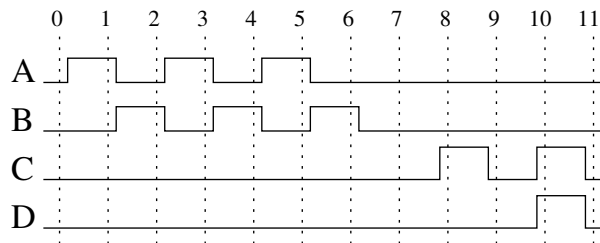


FIGURE 1.7 – Trace satisfaisant la propriété $SERE_1$

La partie droite de l'implication est vraie si la séquence $\{a;b\}$ (a suivit de b) se répète trois fois consécutives. Si ce n'est pas le cas, la propriété est vraie par vacuité, sinon il faut vérifier la partie gauche de l'implication. Dans le second cas, la propriété est vraie à partir du cycle où la partie droite de l'implication, $\{c[\rightarrow 2] : d\}$, est vérifiée. La sous séquence de la partie droite de l'implication est vérifiée s'il y a deux répétitions non forcément consécutives de c et que lors de la dernière répétition de c , d est vrai. Le chronogramme de la figure 1.7 illustre une trace satisfaisant la **SERE**₁.

Le sous-ensemble OBE : ce sous-ensemble supporte les opérateurs A^2 et E^3 de la logique CTL. Le sous ensemble OBE n'est utilisé que pour la vérification statique.

1.4.2 Présentation des opérateurs FL de PSL

Le langage PSL fournit une sémantique formelle pour les principaux opérateurs FL, les autres opérateurs étant obtenus grâce à des règles de ré-écriture. La sémantique de PSL est définie par rapport à un ensemble non vide P de propositions atomiques p . Afin de présenter la sémantique PSL, on introduit les notations suivantes :

- $\doteq$: ré-écriture.
- $\equiv$: équivalence logique entre deux formulations.
- une lettre l est un ensemble de propositions atomiques p
- un mot v est un ensemble de lettre, par exemple $v = abcdef$ où a, b, c, d, e et f sont des lettres.
- $v^{i..}$: est le suffixe du mot v à partir de la i -ème lettre du mot. Si on a $v = abcdef$, alors $v^{3..} = def$.
- $|v|$: représente la longueur du mot v . Si $v = abcd$, alors $|v| = 4$.
- $v \models \varphi$: signifie que le mot v satisfait la propriété φ . En d'autre termes, cela signifie que la trace v vérifie la formule temporelle φ .

En pratique, une proposition atomique p correspond à un signal booléen du circuit sous vérification. Une lettre l correspond à un ensemble de propositions atomiques, c'est-à-dire aux valeurs à un instant donnée d'un ensemble de signaux du circuit sous vérification. Enfin un mot v étant un ensemble de lettres, il peut être vu comme les valeurs successives d'un ensemble de signaux du circuit sous vérification.

La suite présente brièvement la sémantique des principaux opérateurs FL ainsi que les règles de ré-écritures permettant d'obtenir tous les opérateurs FL. On considère par ailleurs que toutes les propriétés utilisées dans la suite sont synchronisées sur Clk , c'est-à-dire que les instants d'évaluation sont les fronts montants d'horloges.

1.4.2.1 Sémantique des principaux opérateurs FL

Soient φ et ψ deux formules FL, la norme PSL fournit les définitions suivantes :

L'opérateur until!

$$v \models [\varphi \text{ until! } \psi] \equiv \exists k, k < |v| \text{ tel que } v^{k..} \models \psi \wedge \forall j, j < k, v^{j..} \models \varphi$$

2. $A(P)$: tous les chemins d'exécution doivent vérifier la propriété P .

3. $E(P)$: il existe au moins un chemin d'exécution vérifiant la propriété P .

Ceci signifie qu'il existe un cycle k où ψ sera vérifiée et que jusqu'à ce cycle (non inclus), la trace commençant à chaque position j doit vérifier φ . Cela nécessite de vérifier la validité de ψ à chaque cycle précédant le cycle k . L'évaluation peut donc avoir lieu sur plusieurs cycles. On qualifie **until!** d'opérateur avec **ré-entrance**. On notera que pour tous les instants précédents k , aucune contrainte n'est donnée sur ψ . La trace présentée figure 1.8 illustre la sémantique de l'opérateur **until!** sur une trace satisfaisant :

property P_2 is **assert** a until! b ;

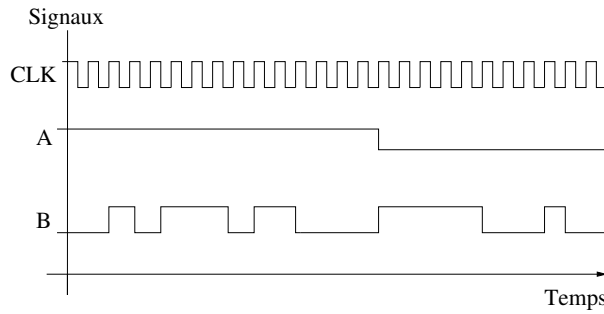


FIGURE 1.8 – Chronogramme satisfaisant P_2

L'opérateur eventually!

$$\text{eventually! } \varphi \doteq [\text{true until! } \varphi]$$

En utilisant la définition de **until!** on obtient après simplification :

$$v \models \text{eventually! } \varphi \equiv \exists k, k < |v| \text{ tel que } v^{k..} \models \varphi$$

Ceci signifie que la trace décrite par le mot v contient une position k à partir de laquelle la formule φ sera satisfaite. Cette position k doit arriver obligatoirement comme l'illustre la trace de la figure 1.9 satisfaisant :

property P_3 is **assert** eventually! c ;

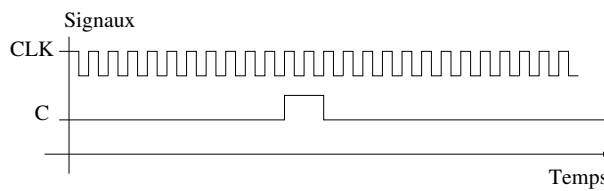


FIGURE 1.9 – Chronogramme satisfaisant P_3

L'opérateur always

$$\text{always } \varphi \doteq \neg \text{eventually! } \neg \varphi$$

En utilisant la définition de **eventually!** on obtient après simplification :

$$\text{always } \varphi \equiv \forall k, k < |v| \Rightarrow v^{k..} \models \varphi$$

Ceci signifie que la trace représentée par le mot v et commençant à chaque instant k doit vérifier φ . Ainsi, la trace débutant au cycle 0 doit vérifier φ , ainsi que celles débutant aux cycles 1, 2, ..., $|v|$. Des exemples de traces satisfaisant cet opérateur sont fournis plus loin dans le manuscrit.

L'opérateur until

$$[\varphi \text{ until } \psi] \doteq [\varphi \text{ until! } \psi] \vee \text{always } \varphi$$

En utilisant la définition de **until!** et de **always** on obtient après simplification :

$$\varphi \text{ until } \psi \equiv (\exists k, k < |v| \text{ tel que } v^{k..} \models \psi \wedge \forall j, j < k, v^{j..} \models \varphi) \vee (\forall k, k < |v| \Rightarrow v^{k..} \models \varphi)$$

L'opérateur **until** est la version faible de l'opérateur **until!**, la modélisation du comportement faible est contenue dans le OU. En effet celui-ci permet deux alternatives possibles : soit la trace de v débutant à un cycle k vérifiera ψ et alors pour tous les cycles antérieurs la trace v aura vérifiée φ ; soit aucun cycle à partir duquel la trace v pourra vérifier ψ ne se présente, et alors la trace devra vérifier φ pour tous les cycles. La première alternative modélise un **until!** et la seconde un **always** sur φ .

Les opérateurs next! et next

$$\text{next! } \varphi \equiv |v| > 1 \wedge v^{1..} \models \varphi$$

Ceci signifie que la trace de v dure un minimum de deux cycles **et** que la trace débutant à la position l'instant suivant vérifie φ . Autrement dit, cet opérateur introduit un cycle de retard dans l'évaluation de φ . On qualifie **next!** d'opérateur avec **retard**.

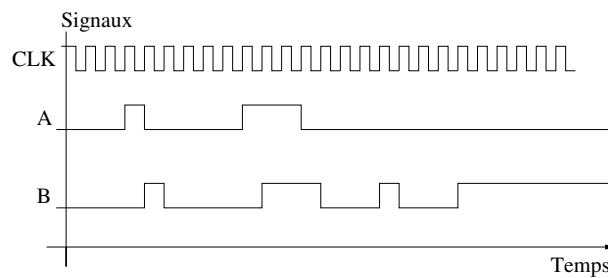


FIGURE 1.10 – Chronogramme satisfaisant P_4

La trace de la figure 1.10 donne un exemple de trace satisfaisant :

property P_4 is **assert** **always** $a \rightarrow \text{next! } b$;

On voit que pour chaque cycle d'évaluation, quand a est vrai alors b l'est aussi un cycle plus tard.

La version faible de l'opérateur est obtenue par ré-écriture de l'opérateur **next!** :

$$\text{next } \varphi \doteq \neg \text{next! } \neg \varphi$$

On obtient donc après développement :

$$\text{next } \varphi \equiv |v| \leq 1 \vee v^{1..} \models \varphi$$

Contrairement à l'opérateur **next!**, celui-ci est faible et il n'impose pas que la trace doive durer deux cycles. Ainsi deux cas se présentent : la trace ne dure pas deux cycles **ou** alors elle dure au moins deux cycles et la trace débutant au cycle suivant vérifie la formule φ . Le choix entre ces deux alternatives est modélisé par l'opérateur de disjonction $\vee$.

1.4.2.2 Les opérateurs FL additionnels

L'opérateur never Cet opérateur est obtenu grâce à la règle de ré-écriture suivante :

$$\text{never } \varphi \doteq \text{always } \neg \varphi$$

On obtient alors après développement et simplification :

$$\text{never } \varphi \equiv \forall k, k < |v| \Rightarrow \neg(v^{k..} \models \varphi)$$

La sémantique signifie que pour une trace correspondant à un mot v , il n'existe aucun suffixe de la trace satisfaisant φ . Soit une trace v , alors à l'instant 0 la trace ne vérifie pas φ , ni à l'instant 1, 2, ..., $|v|$. La figure 1.11 donne une trace satisfaisant :

property P_5 is **assert never** (a and next! b) ;

Dans cette trace on a $\varphi \equiv a$ and next! b et l'on voit que " a vrai et b vrai un cycle plus tard" n'est jamais rencontré. Donc la trace satisfait **never** φ .

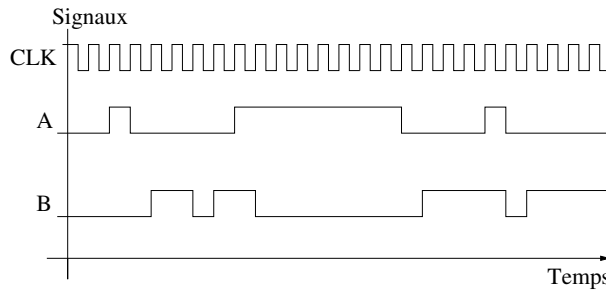


FIGURE 1.11 – Chronogramme satisfaisant P_5

Les opérateurs until* La notation **until*** regroupe l'ensemble des opérateurs { **until!**, **until**, **until!**, **until** }. Les opérateurs **until!** et **until** ont été traités précédemment. Les opérateurs **until!** et **until** sont les versions recouvrantes des opérateurs **until!** et **until** et sont obtenus grâce aux règles suivantes :

$$\begin{aligned} \varphi \text{ until! } \psi &\doteq [\varphi \text{ until! } \varphi \wedge \psi] \\ \varphi \text{ until } \psi &\doteq [\varphi \text{ until } \varphi \wedge \psi] \end{aligned}$$

La seule différence entre les opérateurs recouvrants ou non de la famille **until*** intervient au cycle où l'on doit vérifier ψ : l'opérateur recouvrant est plus contraignant car il nécessite que l'on vérifie $\psi \wedge \varphi$ et pas simplement φ .

Les opérateurs before* Ils sont obtenus grâce aux règles suivantes :

$$\begin{aligned} \varphi \text{ before! } \psi &\doteq [\neg \psi \text{ until! } \varphi \wedge \neg \psi] \\ \varphi \text{ before! } \psi &\doteq [\neg \psi \text{ until! } \varphi] \\ \varphi \text{ before } \psi &\doteq [\neg \psi \text{ until } \varphi \wedge \neg \psi] \\ \varphi \text{ before } \psi &\doteq [\neg \psi \text{ until } \varphi] \end{aligned}$$

Ces opérateurs servent donc à vérifier que φ est vraie avant que ψ ne le soit. Contrairement au cas des opérateurs **until***, le **before** recouvrant est moins contraignant que le **before** non recouvrant. En effet Le **before** non recouvrant impose que lorsque φ vraie, ψ **ne doit pas**

être vraie au même cycle. Dans le cas du **before** recouvrant, la valeur de ψ au cycle où φ est vraie, importe peu. On ne détaillera pas plus la sémantique de ces opérateurs et on illustre le comportement du **before!** grâce à la trace de la figure 1.12 qui satisfait :

property P_6 is **assert** a **before!** b ;

Dans cette trace, on a bien a vrai avant que b ne le soit et a n'est pas actif lors du cycle où b est actif pour la première fois.

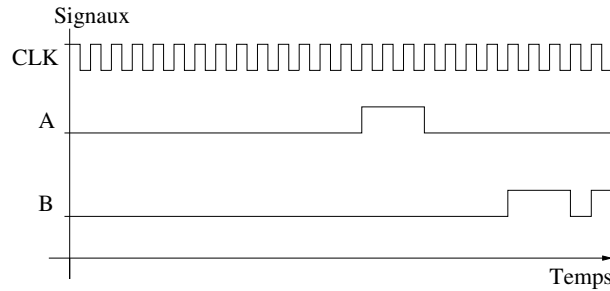


FIGURE 1.12 – Chronogramme satisfaisant P_6

Les opérateurs next* Les opérateurs **next** et **next!** ont déjà été présentés. Ils permettent tous deux d'introduire un cycle de retard dans l'évaluation d'une opérande φ . Les opérateurs **next[i]** et **next![i]** permettent d'introduire i cycles de retard. Les opérateurs **next[i]** et **next![i]** sont obtenus par ré-écriture de la façon suivante :

$\text{next!}[i] \varphi \doteq \text{next!} \text{ next!} \text{ next!} \text{ next!} \varphi$ avec **next!** répété i fois
 $\text{next}[i] \varphi \doteq \text{next} \text{ next} \text{ next} \text{ next} \varphi$ avec **next** répété i fois

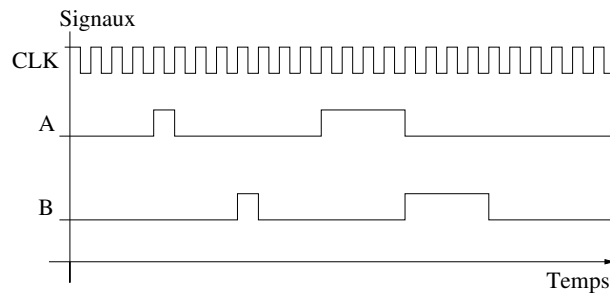


FIGURE 1.13 – Chronogramme satisfaisant P_7

La trace de la figure 1.13 illustre l'opérateur **next[4]** sur une trace satisfaisant :

property P_7 is **assert** **always** $(a \rightarrow \text{next}[4] b)$;

En effet, on remarque que chaque fois que a est actif, b l'est aussi 3 cycles plus tard.

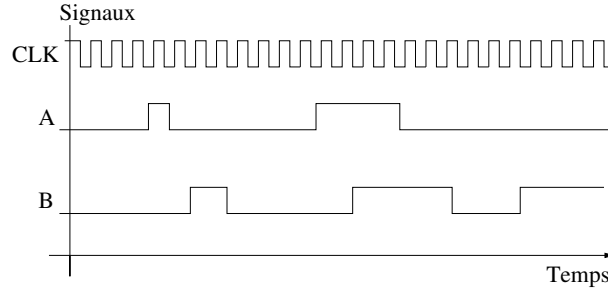
Les opérateurs next_a* Les règles de ré-écriture sont les suivantes :

$\text{next_a!}[i..j] \varphi \doteq (\text{next!}[i] \varphi \wedge \dots \wedge \text{next!}[j] \varphi)$
 $\text{next_a}[i..j] \varphi \doteq (\text{next}[i] \varphi \wedge \dots \wedge \text{next}[j] \varphi)$

Cela signifie que l'on doit vérifier que φ est vraie aux instants $i, i+1, \dots, j-1$ et j . La figure 1.14 donne un exemple de trace satisfaisant :

property P_8 is **assert** **always** $a \rightarrow \text{next_a!}[2..3] b$;

La trace précédente satisfait car chaque fois que l'on rencontre a actif, deux cycles plus tard b est vrai et le reste encore un cycle.

FIGURE 1.14 – Chronogramme satisfaisant P_8

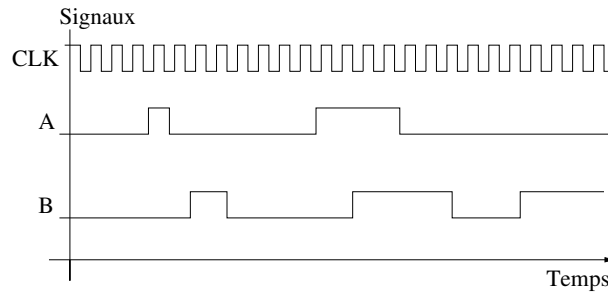
Les opérateurs next_e^* Les règles de ré-écriture sont les suivantes :

$$\begin{aligned}\text{next_e!}[i..j]\varphi &\doteq (\text{next!}[i]\varphi \vee \dots \vee \text{next!}[j]\varphi) \\ \text{next_e}[i..j]\varphi &\doteq (\text{next}[i]\varphi \vee \dots \vee \text{next}[j]\varphi)\end{aligned}$$

Ces opérateurs signifient que l'on doit vérifier φ au moins une fois entre les instants d'évaluation i et j comme l'illustre la trace de la figure 1.16 satisfaisant :

property P_9 is assert $\text{next_e!}[8..17] a$;

Cette trace satisfait la propriété car a est vrai au cycle #3 et des cycles #11 à #14.

FIGURE 1.15 – Chronogramme satisfaisant P_9

Les opérateurs $\text{next_event}(b)$ et $\text{next_event!}(b)$ Les règles de ré-écritures sont les suivantes :

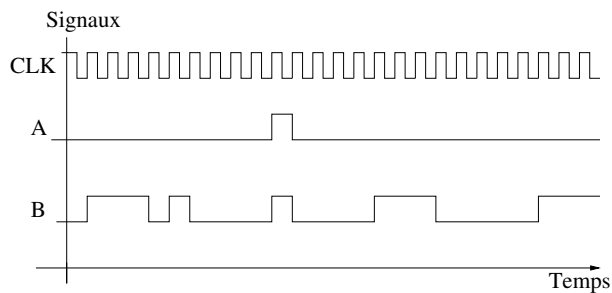
$$\begin{aligned}\text{next_event!}(b)(\varphi) &\doteq [\neg b \text{ until! } b \wedge \varphi] \\ \text{next_event}(b)(\varphi) &\doteq [\neg b \text{ until } b \wedge \varphi]\end{aligned}$$

Ces opérateurs signifient que la trace doit satisfaire φ à la prochaine occurrence de b . La trace de la figure 1.16 illustre ce comportement et satisfait :

property P_{10} is assert $\text{next_event!}(a) b$;

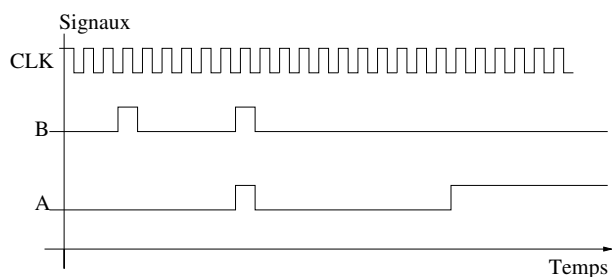
Les opérateurs $\text{next_event}(b)[K]^*$ Les règles de ré-écriture donnent les équivalences suivantes :

$$\begin{aligned}\text{next_event!}(b)[K](\varphi) &\doteq \text{next_event!}(b) \overbrace{(\text{next! } \text{next_event!}(b) \dots (\text{next! } \text{next_event!}(b)(\varphi)) \dots)}^{K-1 \text{ fois}} \\ \text{next_event}(b)[K](\varphi) &\doteq \text{next_event}(b) \overbrace{(\text{next } \text{next_event}(b) \dots (\text{next } \text{next_event}(b)(\varphi)) \dots)}^{K-1 \text{ fois}}\end{aligned}$$

FIGURE 1.16 – Chronogramme satisfaisant P_{10}

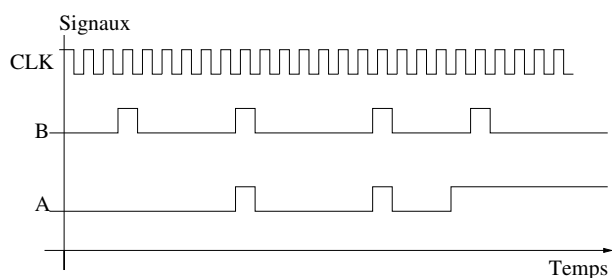
Ces opérateurs signifient que l'on doit vérifier φ à la K ième occurrence de b . La trace de la figure 1.17 illustre la signification de l'opérateur **next_event!(b)[2]** sur la trace satisfaisant :

property P_{11} is **assert** next_event!(b)[2] a ;

FIGURE 1.17 – Chronogramme satisfaisant P_{11}

Les opérateurs next_event_a(b)[K..L]* On a les équivalences suivantes :

$$\begin{aligned} \text{next_event_a!}(b)[K..L](\varphi) &\doteq \text{next_event!}(b)[K](\varphi) \wedge \dots \wedge \text{next_event!}(b)[L](\varphi) \\ \text{next_event_a}(b)[K..L](\varphi) &\doteq \text{next_event}(b)[K](\varphi) \wedge \dots \wedge \text{next_event}(b)[L](\varphi) \end{aligned}$$

FIGURE 1.18 – Chronogramme satisfaisant P_{12}

Ces opérateurs signifient que la trace doit satisfaire φ de la K ième à la L ième occurrence de b . La figure 1.18 fournit un exemple de trace satisfaisant :

property P_{12} is **assert** next_event_a!(b)[2..4] a ;

P_{12} signifie que l'on doit vérifier a vrai lors des 2ième, 3ième et 4ième occurrences de b .

Les opérateurs $\text{next_event_e}(b)[K..L]^*$ On a les équivalences suivantes :

$$\begin{aligned}\text{next_event_e!}(b)[K..L](\varphi) &\doteq \text{next_event!}(b)[K](\varphi) \vee \dots \vee \text{next_event!}(b)[L](\varphi) \\ \text{next_event_e}(b)[K..L](\varphi) &\doteq \text{next_event}(b)[K](\varphi) \vee \dots \vee \text{next_event}(b)[L](\varphi)\end{aligned}$$

Ces opérateurs signifient que la trace doit satisfaire φ au moins une fois entre la K ème à la L ème occurrence de b . La figure 1.19 fournit un exemple de trace satisfaisant :

P_{13} is **assert** $\text{next_event_e!}(b)[3..4] a$;

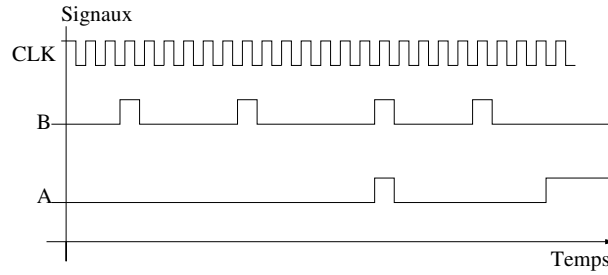


FIGURE 1.19 – Chronogramme satisfaisant P_{13}

1.4.2.3 Classification des opérateurs PSL : retard et ré-entrance

On peut classer les différents opérateurs PSL en fonction des règles de ré-écriture qui permettent de les obtenir. En effet, lorsque l'opérateur est purement booléen, l'évaluation des opérandes est faite en un cycle. On parlera dans ce cas d'opérateurs **sans ré-entrance et sans cycle de retard**. On a vu précédemment que l'opérateur **until!** présentait un phénomène de **ré-entrance**. Autrement dit, il doit évaluer la même opérande sur plusieurs cycles. Ainsi, tous les opérateurs qui sont obtenus avec des règles de ré-écriture utilisant l'opérateur **until!** seront considérés comme étant des opérateurs avec **ré-entrance**. De la même façon, on avait mis en évidence que l'opérateur **next!** introduisait un cycle de décalage dans l'évaluation d'une opérande. L'opérateur **next!** avait été qualifié d'opérateur avec **retard**. Tous les opérateurs utilisant l'opérateur **next!** dans les règles de ré-écriture permettant leur obtention seront considérés comme des opérateurs avec **retard**. On obtient donc quatre classes d'opérateurs :

1. Les opérateurs sans retard et sans ré-entrance : exprimés sans l'opérateur **until!** et sans l'opérateur **next!**. C'est le cas de l'opérateur $\rightarrow$ et de l'évaluation d'un booléen.
2. Les opérateurs sans retard et avec ré-entrance : exprimés avec l'aide de l'opérateur **until!** mais sans l'opérateur **next!**. C'est le cas des opérateurs **until***, **always**, **never**, **eventually!**, **next_event!(b)** et **next_event(b)**.
3. Les opérateurs avec retard et sans ré-entrance : exprimés à l'aide de l'opérateur **next!** et sans l'opérateur **until!**. C'est le cas des opérateurs **next***, **next_e*** et **next_a***.
4. Les opérateurs avec retard et avec ré-entrance : exprimés à l'aide de l'opérateur **until!** et de l'opérateur **next!**. C'est le cas des opérateurs **next_event_e(b)***, **next_event_a(b)*** et **next_event(b)[K]***.

1.4.2.4 Le sous-ensemble simple de PSL

Les travaux adressés par ce manuscrit portent sur la vérification de propriété à l'aide de moniteurs matériels et se trouvent donc dans le domaine de la vérification dynamique. PSL possède un sous-ensemble pour la vérification dynamique : le sous-ensemble simple de PSL noté PSL_{ss} . La restriction de PSL à ce sous-ensemble permet de s'assurer que la validité d'une propriété ne dépend que des événements passés. On a donc un écoulement du temps de gauche à droite lors de la lecture de la propriété permettant une vérification à la volée.

Opérateur PSL	Restriction
Not	opérande booléen
never	opérande booléen ou séquence
eventually!	opérande booléen ou séquence
Or	au moins un opérande booléen
$\rightarrow$	opérande gauche booléen
$\leftrightarrow$	deux opérandes booléens
until until !	opérande droit booléen
until_ until_ !	deux opérandes booléens
before*	deux opérandes booléens
next_e	opérande booléen
next_event_e	opérande droit booléen

Table 1.1 – Restrictions du sous-ensemble simple PSL_{ss}

Le tableau 1.1 donne les restrictions nécessaires sur les opérateurs PSL permettant de se placer dans PSL_{ss} tel que le présente la Section 4.4.4 dans [FWMG05].

1.4.3 Les autres langages

PSL n'est pas l'unique langage permettant l'écriture de propriété, il existe d'autres langages de spécification de propriétés parmi lesquels on peut citer OVA, OVL et SVA.

SVA Le langage SVA (System Verilog Assertions) permet l'écriture de propriétés logico-temporelles. Il est basé sur la syntaxe C et a été normalisé en 2005 [IEE05] avec la norme de System Verilog. La couche "assertion" peut être facilement combinée au sein même du code System Verilog grâce au fort couplage de SVA et de System Verilog. On distingue les deux types d'assertions, concurrentes ou séquentielles. Les premières sont de simples expressions booléennes, placées n'importe où dans le code et exécutées comme des instructions classiques. Les secondes sont exécutées en continue parallèlement à la simulation et peuvent être booléennes ou temporelles. Ce langage, comme PSL, est structuré en 4 couches : booléen, séquence, propriété et assertion. SVA ne possède pas d'équivalent à la couche OBE de PSL et est donc plus orienté pour la vérification dynamique.

OVA OVA (OpenVera Assertion) [OVA] est basé sur ForSpec et la version 2.0 a été mise sur le marché par Synopsys en 2002. Ce langage polyvalent (création de bench, vérification etc...) semble offrir des caractéristiques proches de SVA. Sa syntaxe est basée

sur SystemVerilog. OVA est structuré en 4 couches : booléenne, événement et assertion, implémentation, interface. Les deux premières permettent d'écrire les propriétés, la troisième permet de créer et connecter le module de vérification et la dernière couche offre des options permettant de mieux définir les modules créés à l'aide de la troisième couche.

OVL Dès 1990, Hewlett Packard a développé une bibliothèque de composants paramétriques pour la vérification dynamique appelée Open Verification Library (OVL) [FLT06]. L'utilisation d'une bibliothèque d'assertions pré-conçues, normalisée en 2001 par Accelera, permet aux concepteurs d'écrire plus facilement des propriétés logico-temporelles (sur de la logique 4 états) sans avoir à apprendre un langage de spécification de propriétés. En contrepartie, OVL possède un pouvoir d'expression plus faible que PSL ou SVA.

1.5 Vérification à l'aide de moniteurs synthétisables

L'utilisation de propriétés permet des vérifications statiques (model-checking...) aussi bien que dynamiques (simulation, prototypage...). Dans le second cas, un concept fondamental utilisé par la majorité des outils de vérification est la synthèse de propriétés.

Les assertions sont synthétisées en blocs matériels ou logiciels qui analysent le comportement du circuit à vérifier et reportent une erreur si le circuit présente un comportement contraire à celui décrit par les propriétés : ce sont les **moniteurs**. Un moniteur logiciel est exécuté en parallèle de la simulation du circuit à vérifier tandis qu'un moniteur matériel est connecté physiquement à ce même circuit.

De façon similaire, on peut créer des blocs matériels ou logiciels produisant des séquences de signaux correspondant à une hypothèse : ce sont les **générateurs**. Ainsi la synthèse de l'ensemble des hypothèses permet de fournir un modèle d'environnement pour vérifier le circuit en fonctionnement normal (à l'aide de moniteur par exemple).

Le but des travaux exposés dans la suite de ce manuscrit portant uniquement sur la génération de moniteurs matériels, seules les approches existantes pour la synthèse d'assertions en moniteurs matériels seront présentées dans la suite.

1.5.1 Approches existantes

Apparue en 2005, la vérification à l'aide de moniteurs s'est répandue en grande partie grâce à la commercialisation d'un outil d'IBM : FoCs. Cet outil permet la synthèse de propriétés PSL en moniteurs SystemC, VHDL ou Verilog à l'aide d'une approche basée sur les automates. Si à l'origine, les moniteurs HDL n'étaient pas synthétisables, il est désormais possible de les utiliser en simulation et en émulation. Afin de minimiser l'impact des moniteurs sur le circuit à vérifier, d'autres approches ont vu le jour afin de diminuer le temps de simulation du aux moniteurs ou encore la taille de ces derniers.

BlueSpec SystemVerilog (BVS) [PLBN05] permet la synthèse d'assertions SVA en module BlueSpec synthétisable n'adressant cependant qu'un sous ensemble de SystemVerilog. Le traitement d'opérateurs temporels est interdit car la méthode duplique les FSMs pour traiter la réentrance. Cela conduit à une explosion de la taille des moniteurs.

L'outil MBAC [BZ05, BCZ06, BZ06] développé au Canada, à l'Université McGill, permet la synthèse de moniteurs matériels grâce à une approche basée sur les automates. Pour une propriété PSL donnée, l'automate permettant de reconnaître les violations de

la propriété est construit, puis une description HDL synthétisable est extraite de cet automate.

Pour cela, les opérateurs PSL sont séparés en deux sous-ensembles. Le premier sous-ensemble contient essentiellement les opérateurs booléens et SERE, chacun de ces opérateurs possédant un automate pré-défini. Les opérateurs du second sous-ensemble sont d'abord exprimés à l'aide d'opérateurs du premier sous-ensemble en utilisant des règles de ré-écritures prouvées correctes [MABBZ08]. Puis l'automate des opérateurs du second sous-ensemble est obtenu en combinant des automates correspondants aux opérateurs du premier sous ensemble.

Pour obtenir l'automate d'une propriété complexe, le principe est le même que pour obtenir les automates des opérateurs du second sous-ensemble. On ré-écrit la propriété en l'exprimant à l'aide d'opérateurs du premier sous-ensemble puis on extrait l'automate de la propriété en combinant les automates prédéfinis de ces opérateurs.

L'intérêt de cette solution est de pouvoir simplifier l'automate et ainsi obtenir un moniteur matériel optimisé contrairement à une approche modulaire où aucune amélioration n'est apportée puisque ces méthodes consistent à connecter des blocs matériels pré-existants. Ces optimisations sont rendues possibles grâce à une astuce fondamentale : les parties droite et gauche d'une implication sont traitées différemment. Pour notifier les violations d'une propriété exprimée sous forme d'une implication, il faut détecter tous les instants où la partie gauche est vraie et il suffit ensuite de détecter à ces instants les violations de la partie droite. On distingue alors deux modes d'interprétation d'une propriété :

- **Condition** : reconnaissance d'une séquence, expression booléenne vraie
- **Obligation** : violation de séquence, expression booléenne fausse

Cette méthode est sans doute la plus efficace avec la méthode **Horus** présentée dans la suite.

1.5.2 La plateforme et la méthode Horus

Les travaux présentés dans la suite de ce manuscrit sont très largement inspirés de la génération de moniteurs matériels synchrones grâce à la plateforme **Horus** [OMAB08a, OMAB08b]. Cette plateforme fournit un ensemble d'outils pour l'ABV : synthèse automatique de moniteurs synchrones synthétisables pour la vérification en ligne de propriétés [Odd09], synthèse automatique de moniteurs asynchrones [MAFRB07], construction de moniteur au niveau SystemC TLM [Fer11, PF08], synthèse de générateurs synchrones permettant de générer des stimuli définis à l'aide de propriétés temporelles [Odd09, OMAB06] et enfin synthèse de circuits corrects par construction à partir de spécifications temporelles exprimées en PSL.

La première partie d'**Horus** à avoir vu le jour est celle concernant la construction des moniteurs synchrones [BLOF06]. La synthèse est dirigée par la syntaxe et permet une approche totalement modulaire. Un composant matériel décrit au niveau HDL, appelé *moniteur primitif*, est construit pour chaque opérateur élémentaire de PSL (ou SVA). Les moniteurs primitifs sont ensuite interconnectés grâce à l'arbre syntaxique de la propriété considérée.

1.5.2.1 Moniteurs primitifs

Les moniteurs primitifs sont divisés en deux types : les **observateurs** et les **connecteurs**. Un connecteur est utilisé pour l'activation d'une sous-partie de la propriété (opérandes de type FL) tandis qu'un observateur détecte les violations de propriété (opérandes de type booléen). Un observateur particulier est le *mnt_Signal* qui correspond à l'observation d'un simple signal booléen. Le tableau 1.2 donne le type de chaque moniteur primitif de PSL.

Observateur	<code>mnt_Signal</code> , <code>iff</code> , <code>eventually!</code> , <code>never</code> , <code>next_e</code> , <code>next_event_e</code> , <code>before</code>
Connecteur	<code>→</code> , <code>and</code> , <code>or</code> , <code>always</code> , <code>next!</code> , <code>next_a</code> , <code>next_event</code> , <code>next_event_a</code> , <code>until</code>

Table 1.2 – Type pour chaque moniteur primitif de PSL

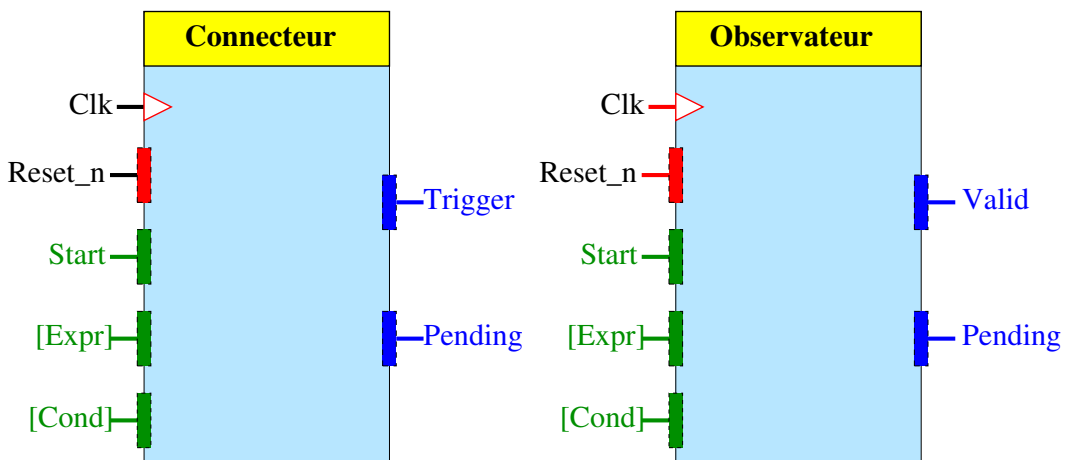


FIGURE 1.20 – Interface des moniteurs primitifs synchrones

Interface des moniteurs primitifs Tous les connecteurs ont la même interface générique illustrée par la figure 1.20. Ils prennent en entrée les signaux *Reset_n* et *Clk* afin d'assurer la synchronisation entre les moniteurs primitifs et le circuit à tester. Le signal *Start* permet d'activer le moniteur primitif. Une ou deux entrées, *Expr* et *Cond*, sont les opérandes à observer. Les connecteurs possèdent deux signaux de sortie *Trigger* et *Pending*. Le premier permet de déclencher la vérification de la sous propriété de l'opérande courant alors que le second renseigne sur l'état de vérification de l'opérande.

Les observateurs possèdent une interface générique avec les mêmes entrées que les connecteurs. La seule différence concerne le signal de sortie *Trigger* qui n'existe plus et fait place au signal *Valid*. Ce signal renseigne, tout comme *Pending*, sur l'état de la propriété.

Architecture des moniteurs primitifs Les moniteurs primitifs possèdent tous une architecture basée sur un même modèle. Cette structure, illustrée figure 1.21, se compose d'un bloc SEM implémentant la sémantique de l'opérateur PSL et d'un bloc CTRL permettant de contrôler le moniteur (initialisation, activation ou non du bloc SEM). Les paramètres des opérateurs PSL sont traités directement au niveau HDL à l'aide de paramètre générique comme *OP_TYPE* pour définir le caractère fort ou faible, recouvrant ou non de l'opérateur.

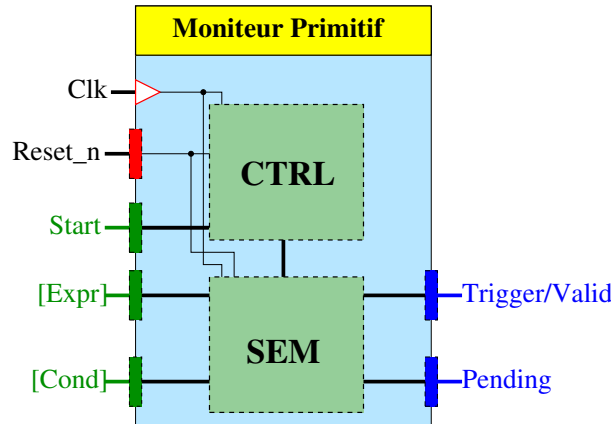


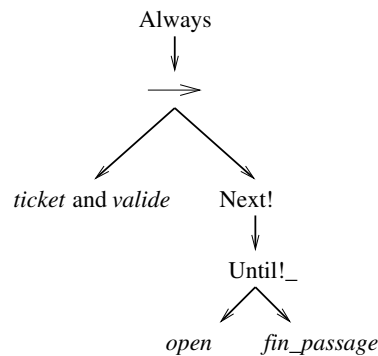
FIGURE 1.21 – Architecture d'un moniteur primitif synchrone

1.5.2.2 Les moniteurs complexes

Afin d'illustrer la construction d'un moniteur complexe on reprend l'exemple de la barrière de parking introduit dans 1.3. Lorsqu'un ticket valide est introduit, la barrière doit s'ouvrir au cycle suivant et le signal *open* passe à '1'. La barrière doit rester ouverte jusqu'au passage à '1' du signal *fin_passage*, indiquant que la voiture a fini de passer la barrière. On rappelle la propriété A_1 :

$\text{property } A_1 \text{ is assert always}((\text{ticket and valide}) \rightarrow \text{next! open until!_ fin_passage});$

Construction d'un moniteur complexe Le moniteur complexe de A_1 est obtenu grâce à la plateforme **Horus**. Le programme construit tout d'abord l'arbre syntaxique de la propriété (c.f. figure 1.22). A chaque nœud de l'arbre correspond un moniteur primitif. Les fils d'un nœud n représentent les opérandes de l'opérateur du nœud n . Le nœud le plus en bas à droite de l'arbre représente l'opérateur qui sera activé en dernier dans la propriété. Chaque moniteur complexe possède ainsi un seul observateur qui correspond à ce nœud spécifique. Les nœuds restants sont de types connecteurs.

FIGURE 1.22 – Arbre syntaxique de la propriété A_1

Dans le cas où le fil droit, respectivement gauche, d'un nœud n est un booléen, le signal correspondant à ce booléen est connecté au port *Expr*, respectivement *Cond*, du moniteur primitif correspondant au nœud n . Si le fils du nœud n est une FL, le moniteur correspondant à cette FL recevra sur son entrée *Start* le signal *Trigger* provenant de la sortie du moniteur primitif du nœud n .

Un moniteur primitif particulier, le *G_init*, fournit le signal *Start* au moniteur primitif correspondant au nœud le plus en haut dans l'arbre syntaxique. Les signaux *Pending* de chaque moniteur primitifs sont connectés à une porte OR, la sortie de la porte OR donne le signal *Pending* pour le moniteur complexe. On obtient ainsi le moniteur complexe complet illustré par la figure 1.23.

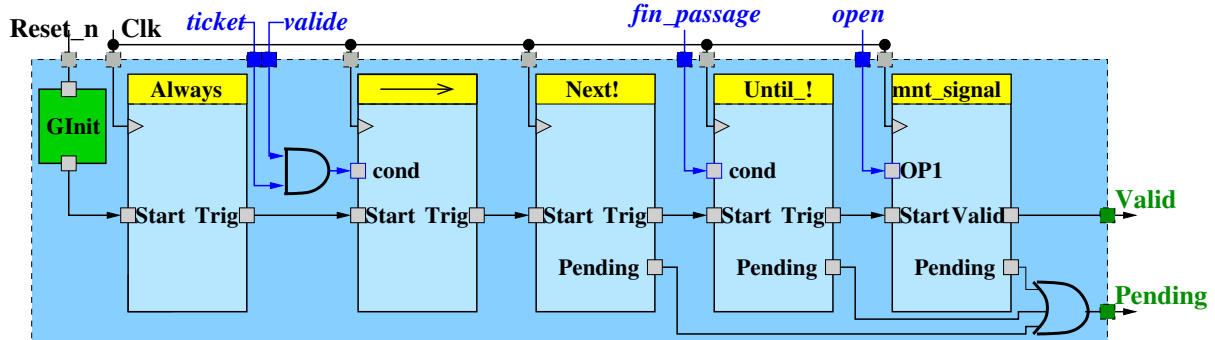


FIGURE 1.23 – Moniteur synchrone complexe de A_1

Les moniteurs complexes présentent tous une interface similaire : deux signaux *Reset_n* et *Clk* assurent la synchronisation avec le circuit sous vérification, un ensemble de signaux d'entrée provenant du circuit sous vérification représentant les opérandes de la propriété à vérifier, enfin deux sorties *Valid* et *Pending* fournissant l'état de la propriété.

Principe de fonctionnement Une bonne compréhension du fonctionnement de cette solution est indispensable afin de comprendre les travaux exposés dans la suite de ce manuscrit. Les fils de connexion entre les moniteurs primitifs sont des bits. Lorsqu'un moniteur primitif est activé (port *Start* à '1'), on dit qu'il possède un jeton. Ce jeton est transmis entre moniteurs primitifs *via* les ports *Trigger* et *Start*. Comme plusieurs moniteurs primitifs peuvent être actifs à un instant t , plusieurs jetons peuvent être présents dans le circuit. La valeur d'un jeton est '1' (moniteur primitif actif) ou '0' (moniteur primitif inactif). Pour comprendre comment les moniteurs primitifs propagent les jetons, on donne l'architecture des moniteurs $\rightarrow$ et *mnt_Signal* figure 1.24.

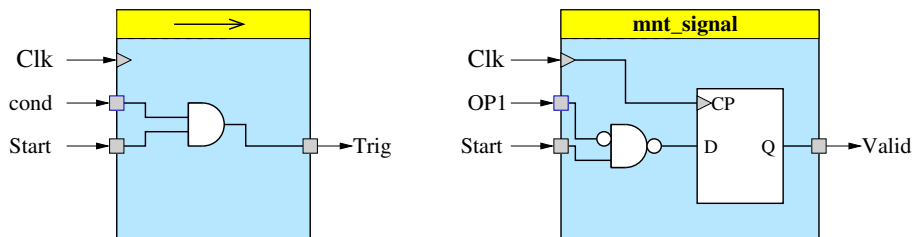


FIGURE 1.24 – Architecture des moniteurs primitifs synchrones $\rightarrow$ et *mnt_Signal*

Nous illustrons le comportement du moniteur complexe A_1 sur la trace de la figure 1.25. Au cycle #0, le circuit vient d'être initialisé. Le moniteur *always* reçoit un jeton du *gen_init* et le transmet directement au moniteur suivant ($\rightarrow$). Le moniteur primitif *always* reçoit un jeton dès que son port *Start* est activé, c'est-à-dire dès que la sortie *Trigger* du *gen_init*

est active. On note `gen_init.trigger`, le signal *Trigger* du moniteur `gen_init`. Le signal `always.trigger` est ensuite stable à '1' permettant l'évaluation de la sous-propriété " $((ticket \text{ and } valide) \rightarrow next! \text{ open until!}__ fin_passage)$ " à chaque cycle.

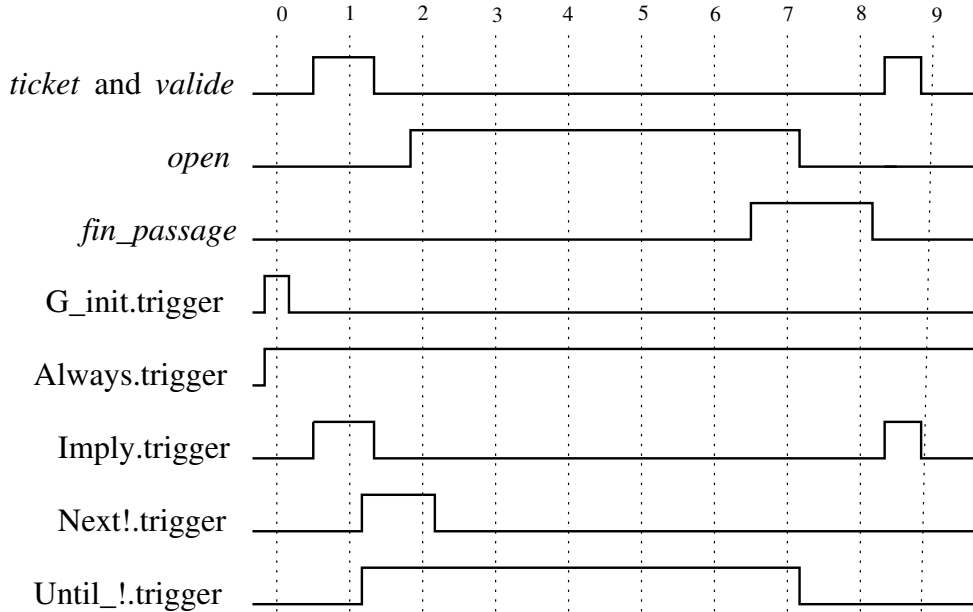


FIGURE 1.25 – Trace satisfaisant la propriété A_1

Peu avant le front montant d'horloge correspondant au cycle #1, les signaux *ticket* et *valide* passent à '1'. La partie gauche de l'implication est vérifiée, donc la partie droite doit être vérifiée. Le moniteur $\rightarrow$ avait reçu un jeton du moniteur `always` (`always.trigger`='1') et son entrée *Cond* ($\leq$ *ticket and valide*) est à '1', sa sortie *Trigger* passe donc à '1' (c.f. signal `ImPLY.trigger` de la figure 1.24).

Cela permet d'activer le moniteur primitif `next!` qui va retarder la transmission du jeton d'un cycle. Le moniteur primitif `next!` avait reçu un jeton au cycle #1, le signal `next!.trigger` passe donc à '1' juste après le front d'horloge correspondant au cycle #1. De cette manière, le moniteur primitif `until!_` ne pourra utiliser ce jeton que pour le cycle d'évaluation #2. Une fois activé, à chaque cycle, le moniteur primitif `until!_` transmet un jeton (`until!_.trigger`='1') au moniteur *mnt_Signal* tant que *fin_passage* n'apparaît pas.

Ainsi le *mnt_Signal* peut vérifier à chaque cycle que *open* est bien actif. En effet, le port *Start* du *mnt_Signal* est à '1' et si *open* passait à '0', la sortie de la porte NAND (c.f. figure 1.24) passerait à '0' elle aussi. Lors du prochain front montant de *Clk* on aurait donc signal de sortie *Valid*='0'.

Peu avant le cycle #7, le signal *fin_passage* passe à '1'. Le moniteur `until!_` envoie un dernier jeton au *mnt_Signal* afin de vérifier que *open* est vrai au cycle où *fin_passage* est vrai car c'est un opérateur recouvrant.

Au cycle #8, le moniteur `always` envoie un jeton au moniteur primitif $\rightarrow$ comme il l'a fait à chaque cycle précédant. Quand *ticket* et *valide* passe à '1', la sortie *Trigger* du moniteur primitif $\rightarrow$ passe à '1'. Cependant, *ticket* et *valide* repasse à '0' avant que le front montant de *Clk* correspondant au cycle #9 n'ait lieu. La sortie *Trigger* repasse donc à '0' et lors du cycle #9 le moniteur primitif `next!` ne sera pas activé.

1.6 Conclusion sur la vérification de circuits

La vérification fonctionnelle d'un circuit électronique est devenue la clé du succès dans la conception d'un nouveau produit électronique. En effet, la vérification fonctionnelle nécessite près de la moitié du temps et du coût de conception et l'ABV représente une alternative prometteuse pour améliorer ces phases de vérification. Les assertions et les hypothèses sont vérifiées tout au long du flot de conception à des niveaux d'abstractions différents, permettant ainsi la mise en évidence de comportement fautif à toutes les étapes et au plus près de leurs sources. De plus, l'utilisation d'un langage avec une sémantique et une syntaxe formellement définie incite le concepteur à approfondir sa réflexion sur la fonctionnalité du circuit à implémenter. Plus récemment, des outils issus de la recherche universitaire ont permis la synthèse de propriété en circuits matériels, les moniteurs, qui connectés physiquement au circuit sous vérification, renseignent "on-line" sur la validité d'une propriété. Parmi ces outils, les plus efficaces sont MBac et Horus qui utilisent tous deux une approche modulaire basée sur la syntaxe de la propriété à synthétiser.

Ces moniteurs matériels offrent une sécurité supplémentaire lorsqu'on les utilise pour vérifier le fonctionnement de circuits mettant en jeu des vies humaines. En effet, ces moniteurs peuvent détecter la violation d'une propriété et déclencher une routine de correction (remise à zéro du système...). Toutefois, lorsque le moniteur et le circuit en vérification sont tous deux conçus dans la même technologie, si un moniteur reporte une violation de propriété, comment savoir alors si le comportement fautif provient du circuit sous vérification ou du moniteur lui-même ? La solution consiste donc à implémenter des moniteurs plus robustes que les circuits qu'ils doivent vérifier en utilisant la technologie asynchrone. Le chapitre suivant présente les circuits asynchrones et comment ces circuits permettent une plus grande robustesse qui pourrait être utilisée pour synthétiser des moniteurs.

État de l'art de la technologie asynchrone

Sommaire

2.1	Introduction	34
2.1.1	Principe de fonctionnement et inconvénient des circuits synchrones	34
2.1.2	Principe de fonctionnement des circuits asynchrones	35
2.2	Les communications asynchrones	36
2.2.1	Le canal de communication	36
2.2.2	Représentation de l'information	36
2.2.3	Les protocoles de communication	37
2.2.4	Implémentation du protocole : la porte de Müller	39
2.3	Modèles de délai et modes de fonctionnement	39
2.3.1	Les modèles de délai	40
2.3.2	Les modes de fonctionnement	40
2.4	Classification des circuits asynchrones	40
2.4.1	Les circuits insensibles aux délais	41
2.4.2	Les circuits quasi-insensibles aux délais	41
2.4.3	Les circuits indépendants de la vitesse	42
2.4.4	Les circuits micro pipelines	42
2.4.5	Les circuits de Huffman	44
2.5	Les aléas	44
2.5.1	Les aléas combinatoires fonctionnels	45
2.5.2	Les aléas combinatoires logiques	45
2.5.3	Les aléas séquentiels	45
2.5.4	Conclusion sur les aléas	45
2.6	Méthodes et outils de synthèse	46
2.6.1	Modélisation et spécification d'un circuit asynchrone	46
2.6.2	Méthodologies de synthèse	47
2.6.3	Les outils d'aide à la conception	49
2.7	Éléments utiles aux implémentations QDI	50
2.8	Conclusion sur la technologie asynchrone	52

2.1 Introduction

La conception de circuits synchrones numériques est aujourd'hui bien maîtrisée grâce à de nombreux outils d'aide à la conception. Ces outils s'appuient sur deux concepts fondamentaux : les signaux manipulés sont codés en binaires et le temps est discrétisé grâce à un signal d'horloge. La binarisation des signaux permet une implémentation électrique simple reposant sur l'algèbre de Boole. La discrétisation du temps permet de s'affranchir des aléas électriques transitoires.

Les circuits asynchrones n'utilisent pas un signal de contrôle global du circuit comme l'horloge. Ils conservent un codage des données discrets mais ne font pas l'hypothèse que le temps est discrétisé. Le contrôle des données est assuré par des moyens alternatifs à l'horloge.

2.1.1 Principe de fonctionnement et inconvénient des circuits synchrones

La plupart des circuits numériques modernes sont synchrones. L'activité du système est cadencée par un signal d'horloge. Le contrôle de la mémorisation des données du système est assuré par un événement sur le signal d'horloge (front montant, niveau haut...). C'est cette mémorisation qui permet la mise à jour du système. Le temps entre deux occurrences de l'événement sur l'horloge, c'est à dire entre deux mémorisations des données, introduit une contrainte temporelle forte : le temps de calcul de n'importe quel élément du circuit doit respecter un temps d'exécution maximum fixé par la fréquence des occurrences des événements sur l'horloge. La simplicité du contrôle global du circuit synchrone a permis à ces circuits de dominer l'industrie microélectronique depuis de nombreuses années. Cependant ce système de contrôle présente aussi des inconvénients de plus en plus contraignants avec la diminution de la taille des transistors.

- Distribution d'horloge : le signal d'horloge doit être uniformément distribué. L'événement du signal d'horloge contrôlant le séquençement du circuit synchrone doit arriver à tous les éléments du circuit en même temps. Dans le cas contraire, l'horloge arrivant à des instants différents ("skew"), le circuit peut ne pas fonctionner correctement. Le problème peut toutefois être résolu en diminuant la fréquence au prix d'une perte de performance, ou en équilibrant le chemin d'horloge au prix d'une augmentation de surface.
- Ordre d'occurrence des entrées : les entrées pouvant arrivées à des instants arbitraires, les éléments de mémorisations peuvent entrer dans un état indéfini : c'est l'état métastable [CM73]. Il existe des contremesures mais le prix à payer est une baisse des performances du circuit.
- Le chemin critique : même si certains composants du circuit peuvent traiter les données rapidement, ils seront contraints par la fréquence d'horloge elle-même définie par le chemin de donnée le plus lent : le chemin critique.
- Consommation : les éléments mémorisants du circuit synchrone étant tous synchronisés avec le signal d'horloge, lors de l'occurrence d'un événement d'activation, tous les éléments mémorisants vont consommer lors de leurs mises à jours, même lorsque la mise à jour n'est pas nécessaire. Cette consommation augmente lorsque la fréquence d'horloge augmente.
- Émissions électromagnétiques : la commutation des signaux étant simultanée, car

régié par l'horloge, les appels de courants aux instants de synchronisation sont très importants et provoquent de forts pics d'émission aux harmoniques de la fréquence d'horloge.

- Modularité : dans un circuit synchrone, il est très difficile de changer une sous partie du système sans impacter le fonctionnement global du circuit en particulier si le nouveau sous-système a une fréquence de fonctionnement plus basse que le système initial. Il est donc difficile de réutiliser facilement des modules synchrones fonctionnant à des fréquences différentes.

Ces différents points prennent une importance considérable quand le circuit synchrone est soumis à un environnement hostile ou à un défaut de fabrication. Par exemple, une modification de la tension d'alimentation va entraîner une augmentation des délais dans les parties combinatoires et ceci peut mener à une violation du chemin critique et donc à un dysfonctionnement du circuit synchrone. Les circuits asynchrones présentés dans ce chapitre ne souffrent pas des inconvénients des circuits synchrones et sont donc beaucoup plus robustes aux environnements hostiles.

2.1.2 Principe de fonctionnement des circuits asynchrones

Par opposition aux circuits synchrones, les circuits asynchrones n'utilisent pas de signal d'horloge. La gestion des communications se fait de manière locale par une synchronisation entre blocs fonctionnels. Un des concepts fondamentaux dans les circuits asynchrones est le contrôle distribué. Cela signifie qu'une unité d'un circuit asynchrone ne se synchronisera qu'avec le ou les blocs auxquels elle doit fournir ou recevoir des données. Cette synchronisation entre unités ne se fait que lorsqu'elle est nécessaire, indépendamment du reste du système. Il est donc nécessaire que chaque bloc possède des signaux de synchronisation explicites permettant l'implémentation et l'exécution de protocole de type requête/acquittement ou "poignée de main" avec les blocs voisins (*c.f.* figure 2.1).

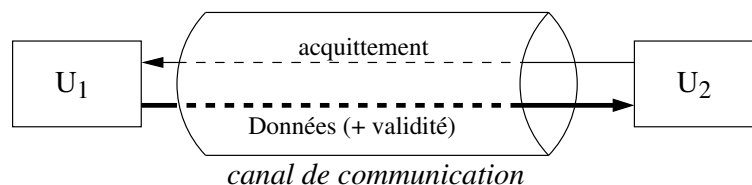


FIGURE 2.1 – Communication de type “poignée de main”

Le dialogue entre les deux unités U_1 et U_2 peut être explicité de la façon suivante :

- U_2 : je suis prêt pour recevoir des données.
- U_1 : j'émet des données.
- Un laps de temps s'écoule pendant que U_2 utilise les données de U_1 .
- U_2 : j'ai fini d'utiliser les données.
- U_1 : d'accord je les enlève.

Grâce à ce mécanisme, le bon fonctionnement du système devient indépendant du temps de calcul de chaque unité composant le circuit. Le circuit ne dépend plus des retards dans les portes et les connexions. Cependant, la synchronisation entre blocs dépend de certaines hypothèses simplificatrices d'un point de vue fonctionnel. Ce sont ces hypothèses qui traduisent les différents modes de fonctionnement des circuits asynchrones. Le choix des hypothèses peut simplifier la conception des circuits et permet de classer les circuits asynchrones.

2.2 Les communications asynchrones

2.2.1 Le canal de communication

Les blocs fonctionnels d'un circuit asynchrone échangent des données (ou jetons) via des canaux de communication. Un canal se compose d'un paquet de fils et d'un protocole de communication. Le paquet de fils est composé des fils transportant les données et des fils de contrôle de protocole. Le protocole permet de gouverner la communication afin de maintenir le bon fonctionnement du système asynchrone. La figure 2.1 illustre un canal de communication.

2.2.2 Représentation de l'information

Les circuits asynchrones réagissent à la présence de données valides sur leurs entrées. Afin d'assurer correctement la synchronisation entre les différents blocs du système, l'information de validité d'une donnée est associée à chaque bus de données. Ceci implique nécessairement un nouveau codage des données. On distingue principalement deux façons de coder la validité d'une donnée [Jer02, Ren00].

2.2.2.1 Le codage données groupées

La façon la plus simple de procéder est de mettre les données sur un bus et d'ajouter un fil au bus qui servira uniquement à spécifier la validité des données sur le bus (*c.f.* figure 2.2.a). Dans ce cas, on a généralement un fil par bit de donnée et le fil spécifiant la validité des données, ou fil de requête, doit être implémenté avec le retard adéquat. Ce retard est conçu supérieur ou égal au temps de calcul dans le pire cas.

Si cette solution est très efficace en terme de surface, elle présente l'inconvénient de fixer un délai au pire cas et donc le fonctionnement ne dépend pas de la propagation réelle des données dans le circuit.

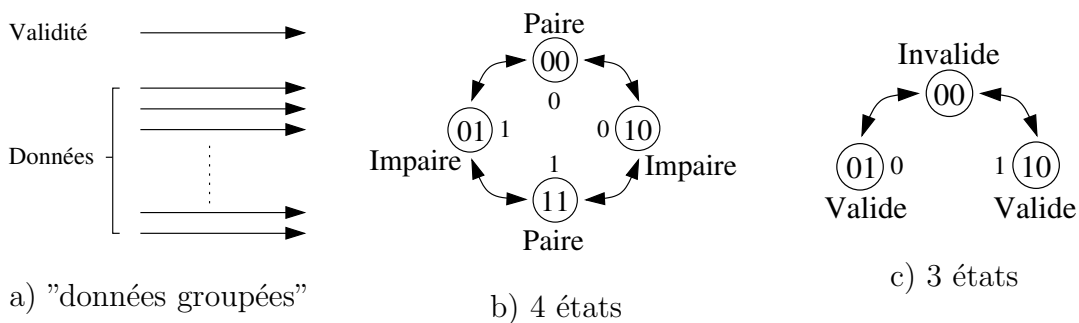


FIGURE 2.2 – Codage de la validité

2.2.2.2 Le codage "insensible aux délais"

Contrairement au codage présenté précédemment, une approche plus astucieuse consiste à intégrer l'information de validité au sein même de la donnée. L'avantage de cette approche est le suivant : les données sont détectées à leur arrivée sans aucune hypothèse temporelle. Ceci permet une plus grande robustesse, une meilleure portabilité et une plus grande facilité lors de la conception (pas de délai à dimensionner).

Le codage 4 états Pour ce codage (*c.f.* figure 2.2.b), chaque bit de donnée utilise deux fils. Ils y donc quatre états possibles pour ces deux fils, la moitié codant la valeur '0', l'autre moitié la valeur '1'. La présence d'une nouvelle donnée valide se traduit par le changement de valeur d'un des fils : celui de gauche pour exprimer la même valeur que précédemment, celui de droite pour exprimer une valeur opposée. La validité d'une donnée est donc exprimée par le changement de parité du couple de fils.

Le codage 3 états Le codage 3 états, tout comme le codage 4 états, utilise deux fils pour un bit de donnée. Cependant, les valeurs représentées ne sont pas dupliquées. Uniquement trois des quatre états possibles pour le couple de fils sont utilisés : un état représente la valeur valide '0', un état représente la valeur valide '1' et le dernier représente l'absence de donnée valide, ou état invalide (*c.f.* figure 2.2.c). Cette figure montre que pour passer d'une donnée à une autre, le couple de fils doit impérativement repasser par l'état invalide.

Ce codage est couramment appelé codage *dual-rail* et est le plus utilisé pour des raisons d'implémentation et de sécurité. Comparé aux autres codages, il y a peu de logique additionnelle : le codage 4 états implique de la logique supplémentaire pour tester la parité du couple de fils et dans le codage "données groupée", il faut re-combiner le signal de validité et les données.

Le codage 1 parmi N Le codage *dual-rail* est un cas particulier du codage "one-hot" utilisant deux fils par bit de données. Pour étendre cette représentation à une donnée comportant N valeurs, on utilise N fils. Chaque fil représente une valeur et l'état invalide correspond à la remise à '0' de tous les fils. Les combinaisons où plusieurs fils sont à '1' sont interdites.

Le codage *mono-rail* Lorsque l'information à transporter est uniquement une information de validité/invalidité, un seul fil est utilisé. Le fil à '1' représente la validité et le fil à '0' représente l'invalidité. On parle de codage *mono-rail*.

2.2.3 Les protocoles de communication

La description du comportement d'un bloc asynchrone dépend du choix du protocole utilisé pour communiquer avec les blocs voisins. Le protocole "poignée de main" est un ensemble de règles gouvernant la communication dans un canal de communication entre deux blocs asynchrones.

Le protocole de communication est caractérisé par :

- La séquence de transfert des informations développée selon un cycle élémentaire (transfert d'un jeton). Avec cette convention, on distingue principalement deux types de protocoles : le protocole 4 phases et le protocole 2 phases.
- Le codage adopté pour les données et l'information de validité.
- L'activité des ports. Un port est considéré comme actif s'il initie la communication, passif si ce n'est pas le cas. Dans la suite du manuscrit, on considèrera le port du bloc asynchrone émettant comme étant toujours le port actif.

2.2.3.1 Protocole deux phases

Ce protocole, aussi appelé NRZ pour *Non Retour à Zéro*, est représenté figure 2.3. Il constitue la séquence minimale d'événements nécessaires à l'établissement d'une communication.

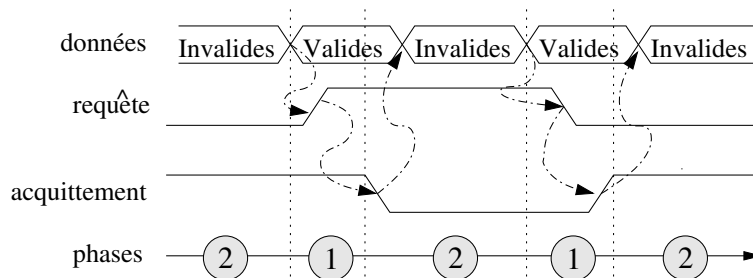


FIGURE 2.3 – Chronogramme du protocole 2 phases

La validité des données et les acquittements sont représentés par les fronts (montants et descendants). Les deux phases peuvent être explicitées de la façon suivante :

- Phase 1 : phase active du récepteur qui détecte une nouvelle donnée valide (front montant ou descendant sur le signal de requête). Le récepteur effectue le traitement de la donnée puis en informe l'émetteur en générant un front (montant ou descendant) sur le signal d'acquittement.
- Phase 2 : phase active de l'émetteur qui détecte le front sur le signal d'acquittement. L'émetteur enlève ces données qui vont être momentanément invalides puis émet une nouvelle donnée valide.

Ce protocole est très efficace car il ne nécessite que deux phases. Cependant, l'asymétrie entre deux transferts successifs (fonctionnement sur front montant ou front descendant) impose une implémentation complexe et se révèle être un obstacle majeur à l'utilisation de ce protocole.

2.2.3.2 Protocole quatre phases

Ce protocole, aussi appelé RZ pour *Retour à Zéro*, possède quatre phases que l'on peut expliciter de la façon suivante :

- Phase 1 (*donnée valide et acquittement inactif*) : le récepteur détecte une donnée valide, effectue le traitement et génère le signal d'acquittement.
- Phase 2 (*donnée valide et acquittement actif*) : l'émetteur détecte le signal d'acquittement et invalide les données (*requête* = '0').
- Phase 3 (*donnée invalide et acquittement actif*) : le récepteur détecte l'invalidité des données et désactive l'acquittement.
- Phase 4 (*donnée invalide et acquittement inactif*) : l'émetteur détecte l'invalidité de l'acquittement et émet une nouvelle donnée si elle est disponible.

Ce protocole est le plus utilisé car, en raison de sa symétrie, il est facilement implémentable. Il existe plusieurs protocoles quatre phases dont les différences se basent sur la durée de la validité des données dans un cycle de communication. On citera par exemple les protocoles quatre phases WCHB [Lin98], PCHB, PCFB ou FDFB [Yah09].

Dans la suite de ce manuscrit, on utilisera le protocole WCHB (Weak Conditionned Half Buffer). Il présente le fonctionnement suivant (*c.f.* figure 2.4) : une donnée valide

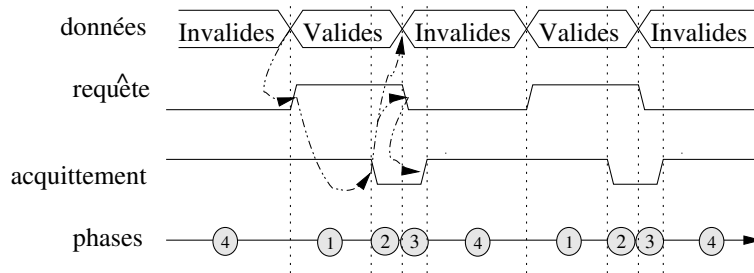
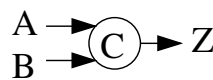


FIGURE 2.4 – Chronogramme du protocole 4 phases WCHB

se présente sur le canal accompagnée de son signal de requête, l'émetteur attend que le récepteur ait consommé la donnée et remît l'acquittement à '0', l'émetteur remet à '0' la requête invalidant du même coup les données et finalement le récepteur n'ayant plus de signal de requête actif remet l'acquittement à '1'. La validité d'une donnée commence sur le front montant du signal de requête et termine sur le front descendant de l'acquittement si l'acquittement est actif au niveau bas.

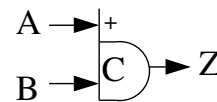
2.2.4 Implémentation du protocole : la porte de Müller

Afin d'implémenter de tels protocoles, les portes logiques élémentaires ne suffisent plus et il est indispensable d'utiliser des portes de Müller (ou *C élément*) [MB59]. Ces portes implémentent un "rendez-vous" entre des signaux. La porte de Müller de base à deux entrées est illustrée figure 2.2.4.a. Si les deux entrées sont égales, la sortie reçoit la valeur des entrées, sinon la sortie mémorise sa valeur précédente.



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

a) Porte de Müller



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

b) Porte de Müller dissymétrique

Dans certains cas, toutes les entrées de la porte n'interviennent pas dans le passage à '0' ou à '1' de la sortie. La porte est alors appelée porte de Müller dissymétrique. La porte de Müller dissymétrique est illustrée figure 2.2.4.b. Ici, les entrées A et B font commuter la sortie à '1' quand elles valent toutes deux '1' mais l'entrée B à '0' permet à elle seule le passage de la sortie à '0'.

2.3 Modèles de délai et modes de fonctionnement

La comparaison des modèles de délai mais aussi de la façon dont le circuit interagit avec son environnement constituent des éléments qui permettent de caractériser les différents circuits asynchrones.

2.3.1 Les modèles de délai

Il existe deux modèles courants de délai : les délais "purs" ou "inertiels" [Ung71, Ung83]. Dans le premier cas, la forme d'onde du signal n'est pas modifiée, ce qui n'est pas réaliste. En effet, cela implique que les portes et les pistes du circuit ont une bande passante infinie. Le second modèle, plus réaliste, supprime les impulsions courtes (certains aléas) en plus de décaler la forme d'onde du signal retardé.

Les délais peuvent aussi être caractérisés vis-à-vis du temps de trois façons différentes :

- Le délai "non borné" possède une valeur arbitraire potentiellement infinie.
- Le délai "borné" possède une valeur arbitraire comprise dans un intervalle connu.
- Le délai "fixe" possède une valeur finie constante.

2.3.2 Les modes de fonctionnement

En plus du modèle de délai, il est nécessaire de caractériser le comportement du circuit dans son environnement.

2.3.2.1 Mode fondamental

Dans le mode fondamental, avant que l'environnement ne soit autorisé à changer une entrée et une seule, le circuit doit être dans un état stable, c'est-à-dire que toutes les entrées, tous les signaux internes et toutes les sorties du circuit asynchrone doivent être stables. Comme l'environnement ne peut pas connaître l'état stable du circuit, il doit respecter le délai le plus long nécessaire pour stabiliser le circuit. Cela implique de borner les délais dans le circuit.

Ce mode de fonctionnement est basé sur les travaux de Huffman [Huf64] et Unger et les circuits correspondants à ce type de fonctionnement sont les circuits de Huffman qui datent des années 50.

Le mode "Burst" ou "Rafale" est une extension du mode fondamental où le circuit accepte un ou plusieurs changements sur ces entrées quand il n'en acceptait qu'un en mode fondamental.

2.3.2.2 Mode entrée/sortie

Dans ce cas, l'environnement répond au circuit sans contrainte de délai. Les seules restrictions que l'on applique sur l'environnement sont des relations de causalité entre les transitions sur les entrées et les sorties.

Pour ces raisons, ces circuits sont souvent spécifiés à base de méthodes reposant sur l'utilisation de graphes. Ainsi le concepteur spécifie toutes les séquences possibles des signaux d'entrée et de sortie pouvant être observés à l'interface du circuit et de son environnement. Ces circuits sont dits insensibles aux délais.

2.4 Classification des circuits asynchrones

Les circuits asynchrones sont communément classifiés suivant le mode de fonctionnement et le modèle de délai adoptés. La figure 2.5 présente la terminologie habituellement utilisée pour qualifier les circuits asynchrones.

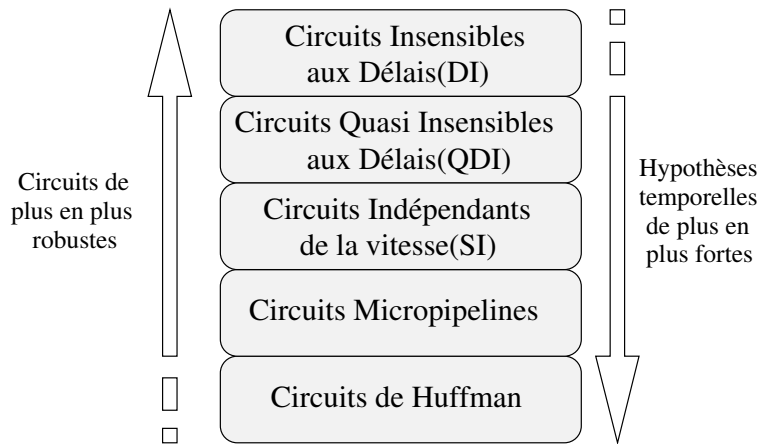


FIGURE 2.5 – Terminologie des différentes classes de circuits asynchrones

Plus le circuit respecte la notion d’asynchronisme dans les communications, plus il sera robuste et complexe. Dans la suite, nous présentons brièvement ces catégories de circuits en commençant par les circuits avec le moins d’hypothèses temporelles.

2.4.1 Les circuits insensibles aux délais

Cette classe de circuits utilise un mode de fonctionnement purement asynchrone. Aucune hypothèse temporelle n’est introduite. D’un point de vue théorique, cela signifie qu’ils sont basés sur un modèle de délai ”non bornés”.

Basés sur les travaux de [Cla67] et re-formalisés par [Udd86], les circuits ”Delay Insensitive” (DI) sont supposés répondre toujours correctement à une sollicitation de l’environnement pourvu qu’ils aient assez de temps pour calculer. Cela impose que le circuit récepteur doit détecter la réception d’une donnée et que le circuit émetteur doit attendre un compte rendu avant d’émettre une nouvelle donnée. L’utilisation de portes logiques standards n’ayant qu’une seule sortie est donc proscrite. En effet, la sortie de la plupart des portes logiques standards peut changer si une seule des entrées change. Cela n’est pas compatible avec le fait que chaque composant d’un circuit DI doit s’assurer du changement de ses entrées avant de mettre à jour sa sortie. Si la sortie d’une porte change alors qu’une seule de ses entrées change, seule l’entrée active sera acquittée sans pouvoir tester l’activité des autres entrées. La seule porte respectant ce comportement est la porte de Müller. Malheureusement, les fonctions réalisables uniquement avec des portes de Müller sont très limitées [Mar90a].

2.4.2 Les circuits quasi-insensibles aux délais

Cette classe de circuits adopte les mêmes modèles de délais ”non bornés” pour les connexions et les composants élémentaires du circuit mais introduit en plus l’hypothèse temporelle de *fourche isochrone* (”isochronic fork”) [Mar90a, vB92].

Une fourche est une connexion reliant un émetteur à deux (ou plusieurs) récepteurs. Elle sera qualifiée d’isochrone si le délai entre l’émetteur et chacun des récepteurs est le même. Cette hypothèse permet de résoudre le problème des portes logiques à une seule sortie que l’on rencontrait dans les circuits DI. En effet, avec une fourche isochrone, il est possible de tester l’acquiescement provenant d’un seul récepteur en considérant que

le signal s'est propagé de la même façon vers les autres récepteurs de la fourche. Alain Martin a montré dans [Mar90b] que l'hypothèse de fourche isochrone est la plus faible hypothèse temporelle à ajouter aux circuits DI pour les rendre réalisables avec des portes logiques standards à plusieurs entrées et une seule sortie. Les circuits "Quasi Delay Insensitive" (QDI) sont donc implémentables à l'aide de cellules standards tels qu'on les utilise dans la conception des circuits synchrones.

Dans les circuits QDI, les signaux peuvent mettre un temps arbitraire à se propager dans les portes et les connexions sans que cela remette en cause la fonctionnalité du circuit. Ces circuits présentent un degré de robustesse quasi-parfait. De plus, l'hypothèse de fourche isochrone est assez faible en pratique et cette condition est facilement remplie en apportant une attention particulière lors du placement routage et de l'étude des seuils de commutation. Il suffit, finalement, que la dispersion des temps de propagation jusqu'aux extrémités de la fourche soit inférieure aux délais des opérateurs qui lui sont connectés (au minimum une porte et un fil dont la sortie interagit avec la fourche [Mar90a, vB92]).

2.4.3 Les circuits indépendants de la vitesse

Les circuits "Speed Independant" (SI) ont été initialement décrit par [Mil65]. Dans ces circuits, les délais dans les portes conservent un modèle "non borné" mais l'hypothèse est faite que les délais dans les connexions sont négligeables. Ce modèle équivaut en pratique à réaliser un circuit QDI où toutes les fourches sont considérées comme isochrones. Il existe aujourd'hui un consensus pour considérer les circuits SI et QDI quasi-équivalents. Hauck montre dans [Hau95] comment une fourche isochrone peut être représenté par un circuit SI (*c.f.* figure 2.6).

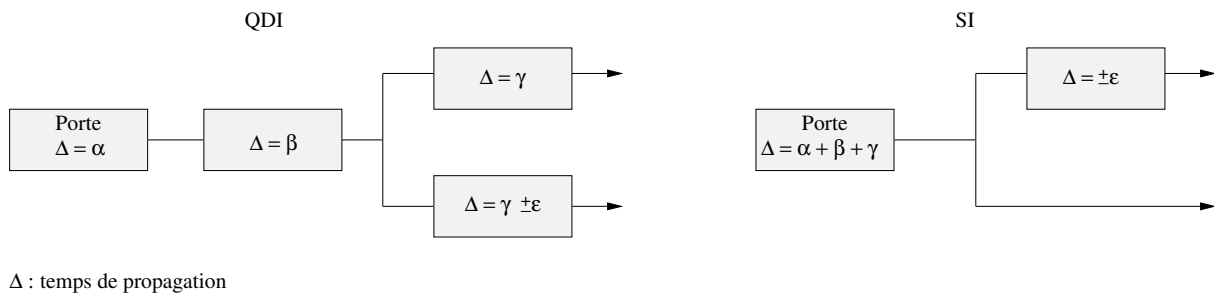


FIGURE 2.6 – Equivalence entre les modèles QDI et SI

Bien que les circuits SI et QDI puissent être considérés comme équivalents, il apparaît plus pertinent de choisir le modèle QDI qui identifie clairement les fourches isochrones de celles qui ne le sont pas.

2.4.4 Les circuits micro pipelines

La technique micropipeline a été initialement introduite par Ivan Sutherland [Sut89]. La technique de conception des circuits micropipelines est plus une classe d'architecture qu'un modèle de délai de circuit asynchrone. Les parties contrôlant le chemin de données sont insensibles aux délais. Le chemin de donnée utilise un modèle de délai "borné". La figure 2.7 représente le contrôle d'une file (FIFO) qui est la structure la plus simple des circuits micropipelines.

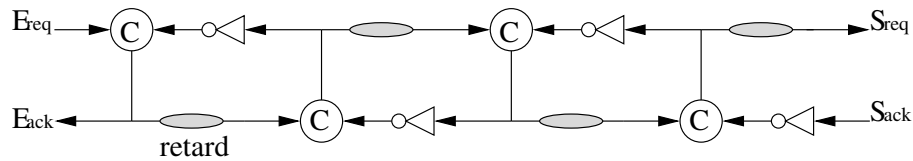


FIGURE 2.7 – Structure de base des circuits micropipelines

Le circuit réagit à des transitions et non pas aux états des signaux (protocole 2 phases). Une transition positive sur E_{req} provoque une transition positive sur E_{ack} qui se propage à l'étage suivant. Le deuxième étage provoque alors une transition positive qui d'une part se propage à l'étage suivant et d'autre part revient vers le premier étage, l'autorisant à traiter une transition négative cette fois. Les signaux se propagent donc dans la structure tant qu'ils ne rencontrent pas un étage déjà "occupé". C'est bien un fonctionnement de type FIFO.

Cette structure peut être enrichie d'opérateurs de mémorisation ou combinatoire. La figure 2.8 illustre un micropipeline avec traitement.

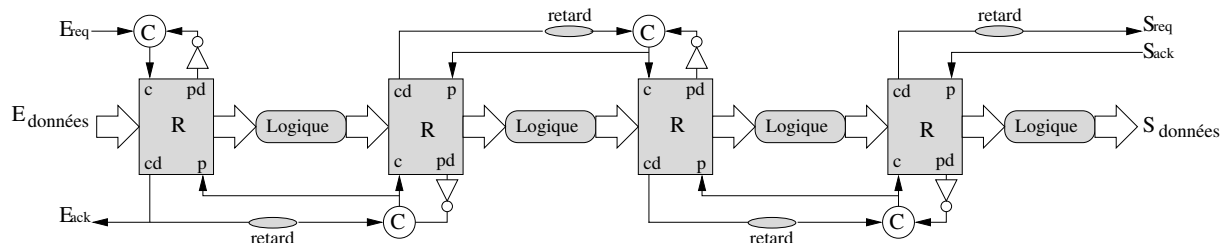


FIGURE 2.8 – Structure micropipeline avec traitement et registre

Dans cette structure, les blocs dénotés "R" sont des registres qui capturent la donnée entrante sur les transitions du signal C . Ils produisent un signal C_D lorsque la donnée est effectivement mémorisée. En pratique, C_D correspond au signal C retardé. Pendant cette phase, la sortie du registre est maintenue et est utilisée par l'opérateur de traitement logique pour générer les sorties. Il faut donc correctement dimensionner le retard pour que les données en entrée du prochain registre soient stables quand le signal C du prochain registre changera de valeur et permettra la mémorisation de la donnée. Lorsque le signal P est actif, le registre est sensibilisé et laisse passer la donnée d'entrée directement à la sortie. Le signal P_D indique que le registre est effectivement sensibilisé.

L'hypothèse temporelle de ces circuits consiste à s'assurer que la mémorisation d'un registre a lieu quand ses données en entrée sont stables. Cela passe par un dimensionnement correct des délais dans les chemins de contrôle des données. On a un modèle de codage "données groupées" associé à un protocole "deux phases". Les signaux de contrôle des données sont en pratique équivalents à des horloges locales avec un dimensionnement des délais fixes qui ne permettent pas de tirer parti de la variation dynamique du temps de traitement des opérateurs.

De nombreuses méthodes de conception s'attachent à concevoir des circuits micropipelines avec des modèles de délai différents. L'idée générale de ces approches consiste à séparer le chemin de données et le contrôle de ce dernier afin de les implémenter séparément dans des styles différents. En particulier, les contrôleurs des registres peuvent être obtenus avec diverses méthodes et divers modèles de délais.

2.4.5 Les circuits de Huffman

Ces circuits, précédemment évoqués dans la partie 2.3.2.1, reposent sur des principes de conception très proches des circuits synchrones. Les délais sont "bornés", voire ont des valeurs fixes et connues. Une étude précise des délais est donc nécessaire pour concevoir ces circuits de façon à dimensionner correctement les temps d'occurrence des signaux de contrôles locaux. Ces circuits sont difficiles à concevoir car une erreur de dimensionnement des délais les rend totalement non fonctionnels.

2.5 Les aléas

La conception des circuits asynchrones présente la contrainte majeure de concevoir un circuit sans aléa. En effet, dans un circuit synchrone, la discrétisation du temps permet d'ignorer les phénomènes transitoires pouvant survenir dans les parties logiques combinatoires ("glitches") entre deux instants de mémorisation. La logique asynchrone est une logique événementielle et un aléa peut être interprété comme un événement porteur d'information. Une attention toute particulière doit donc être portée sur la conception des parties logiques et séquentielles pour obtenir un circuit sans aléas ou "hazard free" [CKK⁺02]. On peut distinguer plusieurs types d'aléas dont les circuits asynchrones doivent se prémunir :

- Les aléas combinatoires fonctionnels.
- Les aléas combinatoires logiques.
- Les aléas séquentiels.

De plus pour les différents types d'aléas, on peut distinguer différentes formes d'aléas (statique ou dynamique) et différentes causes d'aléas (*Single Input Change* ou *Multiple Input Change*).

Aléas statiques ou dynamiques Un aléa est dit statique lorsqu'un signal doit rester constant et qu'il change de niveau deux fois successivement. On appelle plus communément cet aléa un "spike". Un exemple d'aléa statique à '1' pour une porte "ET" est donné figure 2.9.

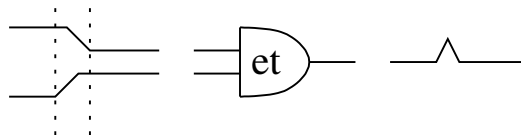


FIGURE 2.9 – Exemple d'aléa statique à '1' pour une porte logique ET

Un aléa est dit dynamique lorsqu'un signal doit changer de valeur une seule fois en théorie et qu'il change plusieurs fois de niveau en pratique. La figure 2.10 illustre un phénomène d'aléas dynamiques.

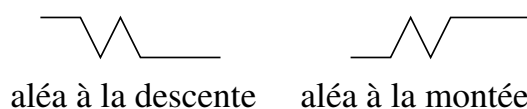


FIGURE 2.10 – Aléas dynamiques

Aléas Single Input Change ou Multiple Input Change Les aléas Single Input Change (SIC), comme indiqué par leur nom, sont la conséquence d'une transition sur une seule des entrées. A contrario, les aléas Multiple Input Change (MIC) se produisent à la suite d'un changement de plusieurs entrées. La figure 2.11 est un exemple d'aléa SIC et la figure 2.9, un exemple d'aléa MIC.

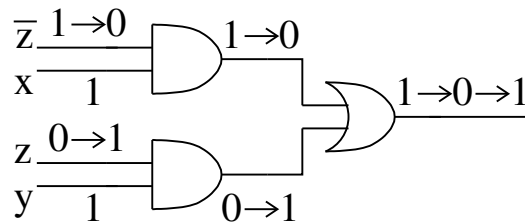


FIGURE 2.11 – Aléas de type SIC

2.5.1 Les aléas combinatoires fonctionnels

Les aléas combinatoires fonctionnels sont propres à une fonction logique et dépendent de la façon dont est décrite la fonction à haut niveau. Ils ne dépendent ni de l'implémentation matérielle de la fonction, ni de la technologie employée. Ainsi, ces aléas étant la propriété de la fonction logique et non de la manière dont cette dernière est implémentée, le seul moyen pour supprimer ces aléas est la modification à haut niveau de la spécification logique de la fonction. Ce type d'aléa peut se produire lorsque plusieurs variables d'entrée changent simultanément, ce qui correspond à des transitions non adjacentes dans le tableau de Karnaugh. Ce type d'aléas est présenté plus en détail dans [BH72].

2.5.2 Les aléas combinatoires logiques

Généralement dus à des courses dans deux chemins combinatoires concurrents qui se rejoignent sur une même porte, ces aléas dépendent de la distribution des délais dans la logique et de l'implémentation des fonctions logiques. Comme pour les aléas combinatoires fonctionnels, ils peuvent être de type statique ou dynamique. Les aléas combinatoires statiques apparaissent quand une transition n'est pas complètement couverte par une porte. Une attention particulière doit être apportée lors de l'implémentation de la fonction, par exemple lors de l'étape de "technology mapping".

2.5.3 Les aléas séquentiels

Ces aléas trouvent leur origine dans les signaux bouclés d'un système séquentiel. Ils proviennent en général de la course sur le signal de sortie lorsque ce dernier est bouclé sur les entrées. Un exemple d'aléa séquentiel est l'aléa essentiel d'Unger qui peut être mis en évidence lorsque l'on change la même entrée une fois puis trois fois et que l'état final n'est pas le même dans les deux cas.

2.5.4 Conclusion sur les aléas

De nombreux travaux portent sur la détection et la suppression des aléas [BH70, BH72, Eic65, Wak94, Ung71, Ung83]. En résumé, les aléas sont générés pour un certain modèle de

délais. Les aléas combinatoires fonctionnels sont difficiles à détecter et à corriger puisqu'ils dépendent uniquement de la spécification du circuit. En revanche, les aléas combinatoires logiques sont liés à la façon d'implémenter une fonction logique. Ces derniers peuvent être généralement supprimés grâce à une couverture des transitions dans le tableau de Karnaugh lorsque l'implémentation utilise des portes logiques.

2.6 Méthodes et outils de synthèse

La conception des circuits synchrones est bien maîtrisée et dispose de nombreux outils d'aide à la conception. Malheureusement, ces outils ne sont pas adaptés à la conception des circuits asynchrones car ils ne prennent pas en compte certaines contraintes comme la nécessité de supprimer les aléas lors de la conception et de l'implémentation. La démocratisation de la conception asynchrone implique le développement d'outils d'aide à la conception mais ces derniers sont longs et coûteux à mettre en place. En conséquence, malgré les nombreux atouts des circuits asynchrones (modularité, robustesse, faible consommation...), les outils permettant leur synthèse ne sont pas si nombreux. La suite permet de différencier les façons dont un circuit asynchrone est spécifié et synthétisé et ainsi de classer les différents outils de conception existants.

2.6.1 Modélisation et spécification d'un circuit asynchrone

Il existe principalement deux façons de spécifier et de représenter le comportement d'un circuit asynchrone :

- Les méthodes basées sur un langage de description de haut niveau tel que System Verilog, VHDL, Communicating Sequential Processes (CSP), Communicating Hardware Process (CHP), Balsa, Tangram...
- Les méthodes basées sur une description comportementale sous forme de graphes telles que les graphes d'états, les Signal Transition Graph (STG) [Chu87] ou les réseaux de Petri [Pet62].

Les méthodes basées sur les langages de description Ces méthodes permettent une continuité sémantique entre les niveaux de description d'un circuit asynchrone. En effet, le même langage peut être utilisé à différents niveaux pour décrire le même circuit depuis la spécification jusqu'à la description structurelle. Ils constituent donc des outils puissants puisqu'ils permettent de s'affranchir d'aspects liés à l'implémentation (choix du protocole, codage des données...) et d'effectuer une exploration architecturale du circuit à synthétiser. De plus, l'utilisation de langage de description matériel permet de décrire des systèmes complexes avec une structure modulaire et hiérarchisée. Toutefois, les circuits synthétisés depuis ces langages ne sont pas optimaux. Il existe deux principaux groupes de langages de description haut niveau : les langages de description matérielle (VHDL, Verilog) et les langages dérivés de CSP. Généralement, dans le premier cas, les langages fournissent un haut niveau d'abstraction mais ne permettent pas au concepteur de définir correctement la notion de canal communicant (exception faite du langage System Verilog). Dans le cas des langages dérivés de CSP, la notion de communication entre processus concurrents à l'aide d'un canal est bien définie mais il n'existe pas de réel consensus dans la communauté asynchrone sur l'utilisation de l'un ou l'autre de ces langages dérivés.

Les méthodes basées sur les graphes Ces méthodes spécifient le fonctionnement d'un circuit asynchrone avec un niveau d'abstraction faible, fréquemment au niveau signal. L'utilisation de graphes s'avère souvent ardue et pénible, surtout pour les systèmes complexes. Cependant, comme la spécification est faite à bas niveau, et si le circuit à synthétiser n'est pas trop complexe, la synthèse est rapide et l'implémentation obtenue est souvent plus optimisée que la solution obtenue avec un langage de description de haut niveau. Ces méthodes de description sont donc le plus souvent utilisées pour spécifier le comportement de contrôleurs ou d'arbitres asynchrones.

2.6.2 Méthodologies de synthèse

Afin d'automatiser la synthèse des circuits asynchrones, plusieurs méthodes ont été développées. D'une manière générale, on peut distinguer les méthodes de synthèse utilisant une représentation du circuit sous forme de graphe et les méthodes de synthèse utilisant une représentation du circuit à l'aide d'un langage de description de haut niveau.

Synthèse depuis un langage Lorsque le circuit à synthétiser est spécifié à l'aide d'un langage de description de haut niveau, plusieurs stratégies sont utilisées pour obtenir la netlist du circuit. Une première méthode consiste à reconnaître des structures bien particulières dans le langage afin de les "traduire" directement en un bloc matériel fixe (prédéfini dans une bibliothèque). Dans ce cas, on parle de **synthèse dirigée par la syntaxe**. Les outils de synthèse utilisant ce principe sont simples puisque que le langage est directement "traduit" en bloc matériel. Cela permet d'une part, d'éviter le problème d'explosion d'états et, d'autre part, d'avoir une approche très modulaire et hiérarchisée. De plus, l'approche modulaire est très intéressante lorsque l'on souhaite réaliser des preuves formelles sur le circuit obtenu. Le principal inconvénient de ces méthodes est le manque d'optimisations globales, les seules optimisations ayant lieu à un niveau primaire en changeant les blocs prédéfinis en d'autres blocs plus efficaces. Le circuit obtenu est donc souvent redondant et bien que la synthèse dirigée par la syntaxe préserve la correction par construction, elle ne suffit pas à assurer la fonctionnalité et le circuit peut présenter un blocage ("deadlock").

Une autre façon de procéder lorsque la spécification du circuit à synthétiser utilise un langage de description de haut niveau consiste à transformer le langage en netlist à l'aide de plusieurs transformation successives, on parle alors de **synthèse par compilation**. À haut niveau, le circuit est représenté comme un ensemble de processus communicants : le protocole est implicite et seules les actions de lectures/écritures sont explicitées. Des transformations successives du code sont réalisées par l'outil, le concepteur devant souvent intervenir manuellement pour guider l'outil et choisir certains paramètres comme le choix du protocole ou du codage des données. Au terme de toutes les étapes de transformations, on obtient une description explicite des signaux. Sophistiquées et complexes, les étapes de synthèse préservent la sémantique du langage mais sont souvent réalisées manuellement. En contrepartie d'un effort plus important du concepteur, ces méthodes permettent des optimisations à plusieurs niveaux d'abstraction et les circuits obtenus sont généralement plus optimisés qu'avec une synthèse dirigée par la syntaxe.

Synthèse depuis un graphe Lorsque le circuit à synthétiser est décrit au niveau signal, on utilise le plus souvent une description du circuit sous forme de graphes (STG,

réseau de Petri...). Pour obtenir le circuit à partir du graphe, une première façon de procéder consiste à "mapper" directement le graphe sur du matériel ("**direct mapping**"). Dans ce cas, les nœuds du graphe correspondent à des éléments du circuit et les arcs du graphe correspondent aux interconnexions, c'est-à-dire aux canaux de communication. Cette façon de procéder ne permet pas l'obtention de circuit très optimisés puisque chaque nœud du graphe est traduit par un élément matériel mais, en contrepartie, il n'est pas nécessaire de calculer l'espace des états atteignables. Des spécifications assez complexes peuvent être synthétisées et la complexité du circuit obtenu croît de manière linéaire avec la complexité de la spécification.

Toutefois, la plupart des outils utilisant un graphe comme spécification du circuit à synthétiser n'utilise pas la technique du "direct mapping". L'outil calcule d'abord l'espace des états atteignables, puis extrait les fonctions logiques permettant de calculer les fonctions générant les signaux importants (sorties, état suivant du système...). Enfin, les fonctions ainsi extraites sont mappées sur le matériel. Lors de ces différentes étapes, il est indispensable de vérifier certaines propriétés sur le graphe telle que la complétude du codage des états (CSC) pour les STGs. Ces méthodes sont limitées à la synthèse de petits circuits car la description d'un système complexe, un microprocesseur par exemple, n'est pas envisageable sous forme de graphe où l'on traite des transitions sur les signaux. De plus, l'exploration de l'ensemble des états atteignables contraint au calcul de l'ensemble des états qui explose rapidement avec la complexité de la spécification.

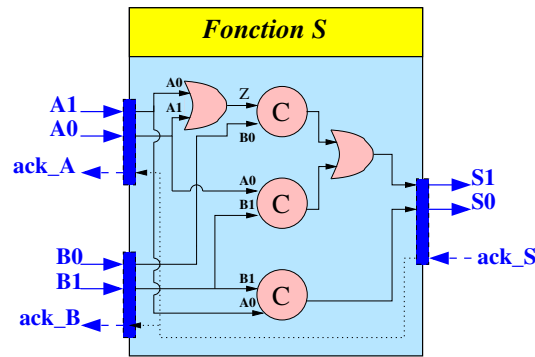
Un exemple de synthèse : la synthèse DIMS La synthèse DIMS (Delay Insensitive Min-Term Synthesis) [SDMD93, DGY92] permet une synthèse facile et DI des fonctions présentes sur un chemin de donnée asynchrone. Lorsqu'une fonction combinatoire est présente sur un chemin de donnée asynchrone, si l'on peut exprimer cette fonction sous la forme d'une somme de produits, alors on applique le principe suivant :

- Tous les mintermes¹ sont implémentés avec des portes de Müller. Ainsi, l'étage d'entrée est composé de portes de Müller implémentant le calcul de tous les mintermes.
- La fonction et le codage sont tels que un et un seul minterme est évalué et produit un '1' en sortie.
- Un deuxième étage formé de portes "OU" propage l'unique valeur à '1' jusqu'à la sortie.

Lorsque toutes les entrées sont invalides, tous les minterms produisent un '0' et les sorties de toutes les portes sont à '0'. On illustre le principe de la logique DIMS sur une fonction S de deux opérandes codées en double rails quatre phases poignée de main tel que $S_1 = A_1.B_0 + A_0.B_0 + A_1.B_1$ (c.f. figure 2.12).

On remarque aussi que lorsque l'on a une fonction du type $S = (A_1 + A_0).B_0$, on peut réaliser le "OU" logique entre A_0 et A_1 avec une porte "OR" puis connecter la sortie Z de cette porte à l'entrée de la porte de Müller qui va implémenter la fonction & entre Z et B_0 . La gestion du protocole de communication est réalisée par les organes de mémorisation placés de part et d'autre de la fonction combinatoire.

1. On rappelle qu'un minterme est un produit où toutes les variables d'entrée de la fonction apparaissent sous leur forme directe ou complémentée

FIGURE 2.12 – Implémentation DIMS de la fonction S

2.6.3 Les outils d'aide à la conception

Il existe de nombreux outils de synthèse pour les circuits asynchrones dont beaucoup sont des productions universitaires. Nous présentons ici les plus connus et ce qui les différencie dans le but de donner un aperçu des principales méthodes existantes. Une liste plus complète de ces outils peut être trouvée dans [Jun04].

Exemples d'approches basées sur un langage de description :

Handshake Technology Cet outil a été initialement développé par Philips [phi], puis vendu par la "spin-off" Handshake Solution jusqu'en 2010. Si aujourd'hui Handshake Solution n'existe plus, NXP continue de produire des circuits asynchrones grâce à cet outil. La spécification est réalisée dans le langage de description haut niveau Haste² utilisant les principes de CSP avec une syntaxe proche des langages de programmation classiques, tels que C ou Pascal. Ce langage propriétaire est traduit en un format intermédiaire reposant sur les circuits "poignée de main" [vBKR⁺91, Ber93]. La compilation dirigée par la syntaxe permet ensuite d'obtenir directement une structure de composant "poignée de main". Ces circuits communiquent via des canaux de communication à l'aide d'un protocole quatre phases données groupées.

CAST Cet outil développé à l'Université Caltech par le groupe de A. Martin repose sur une spécification à l'aide du langage de description de haut niveau CHP. Grâce à de multiples transformations du langage plus ou moins automatisées, l'outil génère des circuits QDI avec un protocole de communication quatre phases. Ces transformations permettent de garantir l'insensibilité aux délais depuis le plus haut niveau jusqu'à l'implémentation physique [Mar90b]. Cet outil a permis l'implémentation de circuits complexes optimisés dont le MIPS R3000 [MLM⁺97].

Balsa Développé à l'Université de Manchester [Bal], cet outil repose sur le langage Balsa [BE00a]. La méthode est similaire à celle de Handshake Technology et repose sur la traduction directe d'un morceau de code Balsa en un circuit "poignée de main" afin d'obtenir le réseau complet de circuits "poignée de main". Dans le cadre du projet Amulet, le microprocesseur Amuleti3 [BE00b] a permis d'illustrer cette méthode de conception.

2. Haste était aussi appelé Tangram avant la création de Handshake Solution

Exemple d'une approche basée sur les graphes :

Petrify Présenté dans [CKK⁺97, CKK⁺02], cet outil est basé sur une spécification STG du circuit à synthétiser. Le STG en entrée de l'outil Petrify peut être graphique ou textuel. Cependant, la synthèse à partir de STG requiert une exploration des états atteignables et limite donc la complexité des circuits traités. Cet outil produit des circuits SI et des circuits utilisant le mode burst (*c.f.* page 40).

Exemples d'approches mixtes :

TAST Cet outil est issu du laboratoire TIMA. Il propose une méthode de synthèse basée sur une spécification à l'aide d'un langage de description haut niveau [DD03, SRSR04]. Le langage utilisé, appelé CHP étendu, combine CHP [Mar90b] et VHDL. A l'aide d'une série de transformations, la spécification est transformée en un format interne basé sur les réseaux de Petri et les graphes de flots de données (Data Flow Graph). Cet outil permet de tirer partie d'une description haut niveau permettant la spécification de circuits complexes mais aussi d'une synthèse efficace produisant de petits circuits grâce à un format intermédiaire approprié. Le concepteur obtient une description VHDL pour la simulation ou une netlist pour une cible ASIC ou FPGA [RIG02]. Les circuits ainsi synthétisés peuvent être Mircopipeline ou QDI en fonction du choix du concepteur.

ACC Asynchronous Circuit Compiler (ACC) est un outil développé par Tiempo [tiea] qui reprend les principes de TAST, ACC étant l'évolution industrielle de TAST. Cet outil prend en entrée une spécification écrite en System Verilog au niveau Transaction-Level Modeling (TLM). L'outil génère des netlists de portes au format Verilog. Les circuits ainsi synthétisés peuvent être routés sur une bibliothèque de cellules standards ou de cellules asynchrones permettant de meilleures performances en consommation/vitesse/surface. L'avantage de cet outil est l'utilisation des outils de conception standards pour la simulation et le placement routage. Des démonstrations de cet outil ont été réalisées à plusieurs reprises [tieb, LPF] et ACC a permis la production de nombreuses IPs.

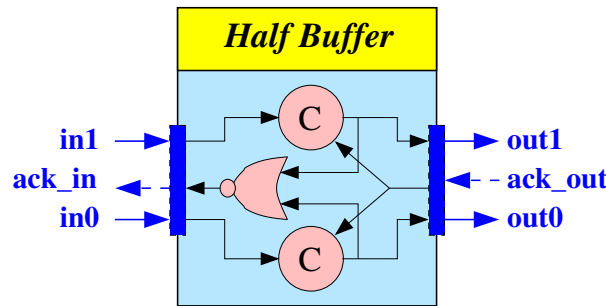
2.7 Éléments utiles aux implémentations QDI

Dans cette partie, on présente brièvement des éléments qui seront ré-utilisés dans la suite du manuscrit. Ces éléments sont utiles pour réaliser des adaptations de protocole, des mémorisations, des synchronisations ou même charger une donnée à l'initialisation.

Le Half Buffer Le bloc *Half Buffer* permet de mémoriser un signal double rails respectant le protocole poignée de main. Son architecture est donnée par la figure 2.13.

Le *Half Buffer* présenté ici permet plus précisément d'implémenter le protocole quatre phases WCHB.

Le Half Buffer Load Ce bloc est un bloc permettant la mémorisation d'une donnée codée en double rails et respectant le protocole quatre phases poignée de main. Le *Half Buffer Load* et le *Half Buffer* présentent la même interface et la même fonction de mémorisation, la différence entre ces éléments se fait lors de l'initialisation. En effet le *Half Buffer Load* permet de charger une donnée lors de l'initialisation.

FIGURE 2.13 – *Half Buffer* : élément de mémorisation pour le codage double rails

Le SYNC Ce composant est implémenté dans la bibliothèque TAL (Tima Asynchronous Library) [MRB⁺03]. Il permet d'échantillonner une donnée entrante *Expr* sur les fronts montants du signal *CP*. On obtient *D1*='1' si *Expr*='1' sur le front montant de *CP*, *D0*='1' sinon. Les sorties *D0* et *D1* restent à '1' tant que *CP*='1'. Dès que *CP* repasse à '0', *D0* et *D1* repassent à '0'. L'interface de ce composant ainsi que son fonctionnement sont explicités par la figure 2.14.

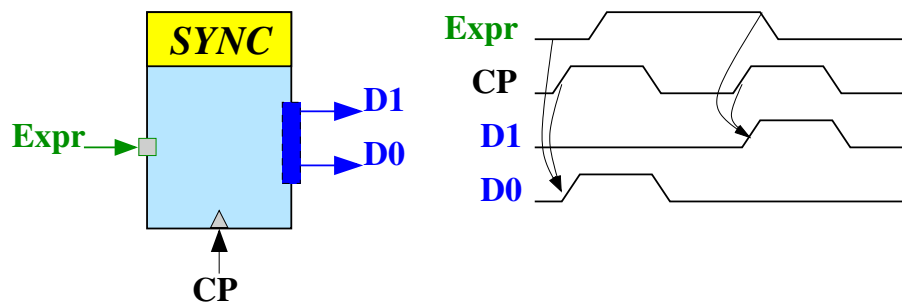


FIGURE 2.14 – Le SYNC, un élément de synchronisation

Le générateur de jetons Ce composant introduit dans [CBC⁺10] permet de transformer un front montant sur l'entrée *data* en un jeton asynchrone. On appelle jeton

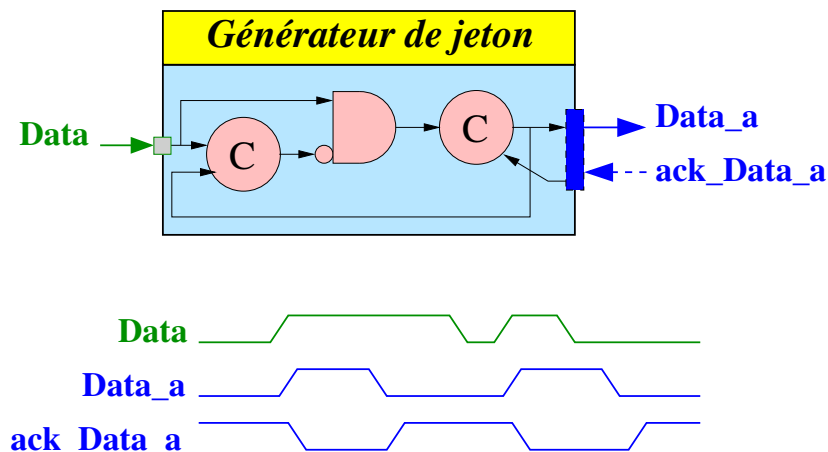


FIGURE 2.15 – Générateur de jetons

asynchrone une donnée valide dans un circuit asynchrone. Par exemple, l'implémentation

de la fonction S (c.f. figure 2.12) prend deux jetons en entrée et les consomme/détruit pour créer un jeton sur la sortie. Le signal de sortie $data_a$ respecte le protocole quatre phases poignée de main et est acquitté grâce au signal ack_data_a . L'architecture et le comportement de ce composant sont résumés dans la figure 2.15.

La fourche Lorsqu'un canal de communication se scinde en deux canaux, la donnée transportée initialement doit être dupliquée et envoyée dans les deux autres canaux. Cette fonction est réalisée par le bloc matériel *Fork*. Lorsque les canaux de communication utilisent un protocole quatre phases WCHB et un codage des données double rails on obtient l'architecture proposée Figure 2.16

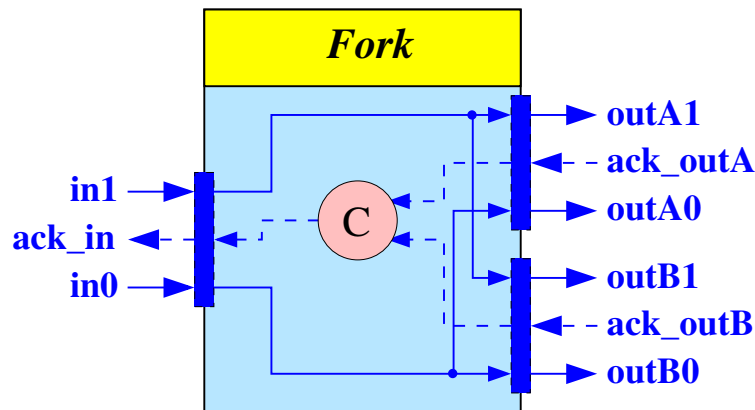


FIGURE 2.16 – Implémentation de la fourche : le bloc *Fork*

2.8 Conclusion sur la technologie asynchrone

L'utilisation de la technologie asynchrone, et plus particulièrement QDI, permet d'obtenir des circuits présentant des avantages importants par rapport aux circuits synchrones comme la robustesse, la faible consommation, la conception modulaire sans signal de synchronisation global... En contrepartie, l'implémentation des protocoles et des codages donne des circuits plus gros que leurs équivalents synchrones.

Le but de ce manuscrit étant la conception de moniteurs robustes, l'utilisation de la technologie asynchrone semble particulièrement adaptée. Le surplus de surface induit par l'utilisation de la technologie asynchrone n'est pas un gros inconvénient car il doit permettre une robustesse bien plus grande du moniteur asynchrone comparé à son équivalent synchrone. De plus, la méthode de synthèse des moniteurs utilisée dans *Horus* dont ce travail s'inspire présente une synthèse basée sur la syntaxe et donc sur l'interconnexion de blocs élémentaires. Ce type d'approche est particulièrement adapté à la conception asynchrone.

Le chapitre suivant présente comment la méthode utilisée pour la synthèse des moniteurs synchrones est modifiée pour permettre la synthèse de moniteurs asynchrones. La suite présente aussi comment est implémenté chaque moniteur primitif de la bibliothèque.

Les moniteurs asynchrones

Sommaire

3.1 Synthèse d'un moniteur asynchrone	54
3.1.1 Interface des moniteurs primitifs asynchrones	54
3.1.2 Interconnexion des moniteurs primitifs asynchrones	54
3.2 Bibliothèque de moniteurs primitifs asynchrones	56
3.2.1 Différences entre moniteurs synchrones et asynchrones	56
3.2.2 Nécessité d'un synchroniseur	58
3.2.3 Lien entre sémantique et implémentation matérielle	60
3.2.4 Les opérateurs sans cycle de retard, sans ré-entrance	60
3.2.5 Les opérateurs sans cycle de retard, avec ré-entrance	62
3.2.6 Les opérateurs introduisant des cycles de retard	65
3.2.7 Autres moniteurs primitifs	71
3.3 Validation par simulation numérique	72
3.3.1 Simulation et validation de la bibliothèque de moniteurs primitifs	72
3.3.2 Simulation et validation de la méthode d'interconnexion	75
3.4 Conclusion	75

L'idée fondamentale de ce travail est de concevoir des moniteurs en technologie asynchrone à partir du langage PSL afin de tirer parti des avantages de l'ABV et de la robustesse des circuits QDI. La construction des moniteurs asynchrones repose sur la méthode utilisée par la plateforme **Horus** : une bibliothèque de moniteurs primitifs asynchrones et une méthode d'interconnexion basée sur l'arbre syntaxique des propriétés PSL à implémenter. Ce chapitre présente les modifications nécessaires pour modifier la méthode d'interconnexion synchrone afin de l'adapter à une implémentation asynchrone. Ces modifications, portant sur l'interface des moniteurs primitifs mais aussi sur le fonctionnement global du moniteur, sont détaillées et expliquées dans ce chapitre. On montrera que l'utilisation de la technologie asynchrone nécessite l'ajout d'un synchroniseur. Les spécifications de ce bloc sont elles aussi explicitées dans ce chapitre. La construction et la validation du synchroniseur seront développées dans le chapitre 4.

3.1 Synthèse d'un moniteur asynchrone

La sémantique PSL est bien maîtrisée lorsque les propriétés sont synchronisées sur l'horloge du circuit à vérifier. De plus, la solution synchrone ayant été prouvée, elle offre un bon modèle de référence pour comparer et valider le fonctionnement de la solution asynchrone. Pour ces raisons, les moniteurs asynchrones ont été conçus pour vérifier des propriétés sur des circuits synchrones. On va donc avoir d'un côté le circuit synchrone à vérifier et de l'autre les moniteurs asynchrones assurant la vérification.

3.1.1 Interface des moniteurs primitifs asynchrones

Les moniteurs primitifs asynchrones présentent des ports d'entrée/sortie similaires à ceux des moniteurs primitifs synchrones utilisés dans **Horus**. La distinction entre moniteurs primitifs observateurs et connecteurs faite dans la solution synchrone s'applique aussi pour la solution asynchrone. On rappelle que connecteurs et observateurs présentent en entrée les signaux *Start*, *Expr* (et/ou *Cond*), *Clk*, *Reset_n* et le signal de sortie *Trigger* pour les connecteurs, respectivement *Valid* si c'est un observateur. Une sortie *Pending* est aussi présente pour les opérateurs forts. Toutefois, afin d'obtenir une implémentation QDI (et donc robuste) des moniteurs asynchrones, les signaux *Trigger*, *Start*, *Valid* et *Pending* évoqués précédemment sont codés en double rails comme l'illustre la figure 3.1. De plus, pour implémenter le protocole quatre phases poignée de main, il est nécessaire d'ajouter un fil d'acquittement en plus des deux fils utilisés pour le codage double rails.

Les signaux *Clk* et *Reset_n* proviennent du circuit à vérifier et permettent de synchroniser les moniteurs asynchrones avec ce dernier pour vérifier les propriétés sur les fronts montants de l'horloge. Les opérandes booléens de la propriété PSL, *Expr* et/ou *Cond*, proviennent eux aussi du circuit à vérifier. On obtient ainsi une nouvelle interface très proche de celle des moniteurs primitifs synchrones. La différence entre interfaces synchrones et asynchrones réside dans la nécessité, pour la solution asynchrone, d'implémenter un protocole poignée de main.

3.1.2 Interconnexion des moniteurs primitifs asynchrones

La méthode d'interconnexion des moniteurs primitifs asynchrones est la même que pour la solution synchrone. On extrait d'abord l'arbre syntaxique de la propriété PSL. Tout

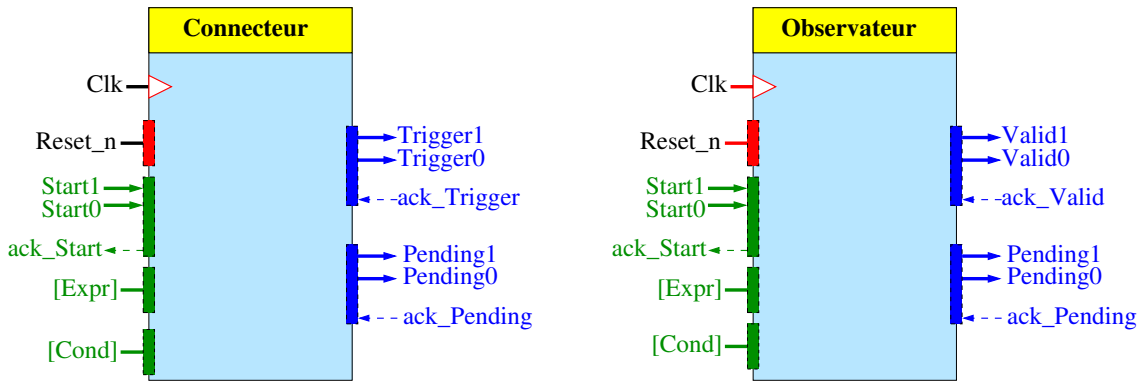


FIGURE 3.1 – Interface des moniteurs primitifs asynchrones

comme en synchrone, le moniteur primitif asynchrone correspondant au nœud le plus en bas à droite est un observateur. Un moniteur particulier, le *G_init* est utilisé pour démarrer l'évaluation de la propriété. Enfin, le port *Trigger* d'un moniteur primitif asynchrone est connecté au port *Start* du moniteur primitif suivant et les signaux provenant du circuit à vérifier représentant les opérandes booléens de la propriété sont connectés aux entrées *Cond* et *Expr* des moniteurs primitifs.

Le signal *Pending* d'un moniteur asynchrone est obtenu en réalisant le "OU" logique entre les différents signaux *Pending* provenant des moniteurs primitifs asynchrones. Les moniteurs primitifs asynchrones utilisent des signaux double rails respectant le protocole quatre phases poignée de main. Il faut donc implémenter une fonction "OU" pour le double rails. On utilise une synthèse DIMS (*c.f.* partie 2.6.2, page 48) et l'on obtient l'implémentation illustrée par la figure 3.2. L'acquiescement *ack_Out* du signal de sortie est propagé vers les sorties *ack_InA* et *ack_InB*. Le protocole quatre phases n'est donc pas directement contrôlé dans l'implémentation du "OU" double rails mais dans les blocs asynchrones de mémorisation le précédant et le suivant.

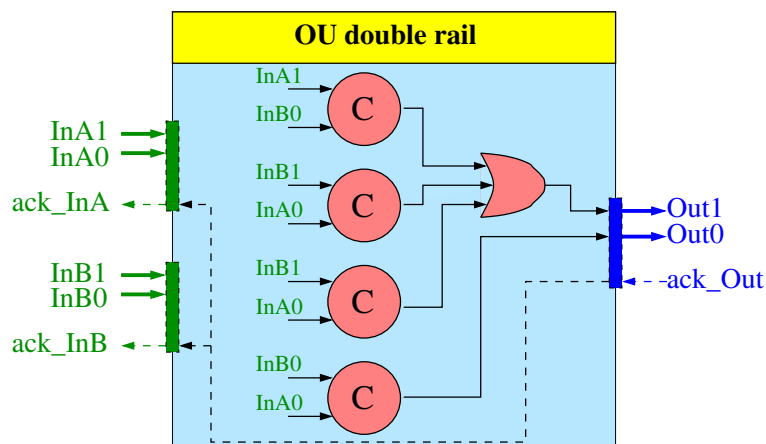
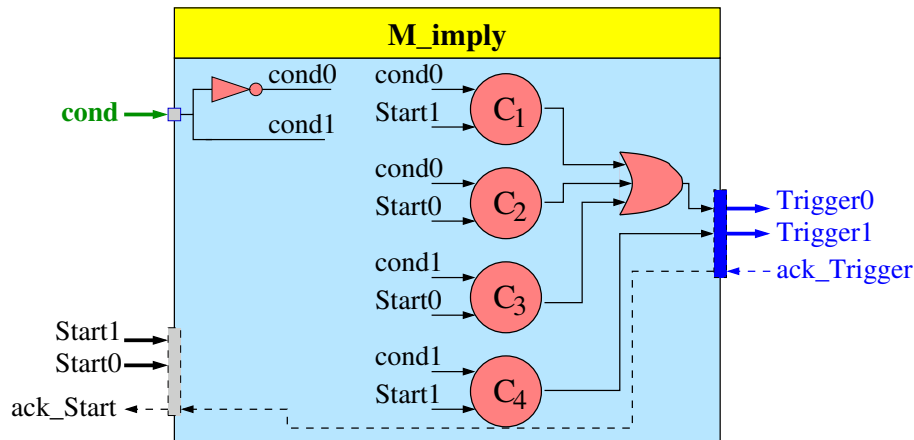


FIGURE 3.2 – Implémentation du "OU" double rails asynchrone

On illustre l'interconnexion des moniteurs primitifs asynchrones grâce à la figure 3.3 représentant le moniteur asynchrone de la propriété A_1 :

property A_1 is assert always(*ticket* and *valide*) $\rightarrow$ next! *open* until!_ *fin_passage* ;

FIGURE 3.4 – Transposition naïve du moniteur primitif M_imply en asynchrone

vérifier, il ne respecte pas le protocole quatre phases "poignée de main" et est donc susceptible de changer de valeur. Du point de vue de la fonction asynchrone implémentée un changement sur *Cond* alors que les 4 phases du protocole ne sont pas terminées, sera perçu comme un aléa. Or, la conception asynchrone doit être exempte d'aléas et l'on peut voir se réaliser les scénarios suivant :

- **Codage double rails non respecté** : si *Start1* (ou *Start0*) est actif et que *Cond*='1', la porte C_4 (ou C_3) est activée. Si *Cond* change de valeur avant que la donnée produite par C_4 (ou C_3) n'ait été acquittée et que *Start1* (ou *Start0*) ne soit repassé à '0', alors la porte C_1 (ou C_2) sera activée à son tour et on obtient en sortie du moniteur primitif les deux rails de la sortie *Trigger* activés. La valeur (1,1) pour le couple de fils *Trigger* est interdite et le fonctionnement explicité ci-avant n'est donc pas correct.

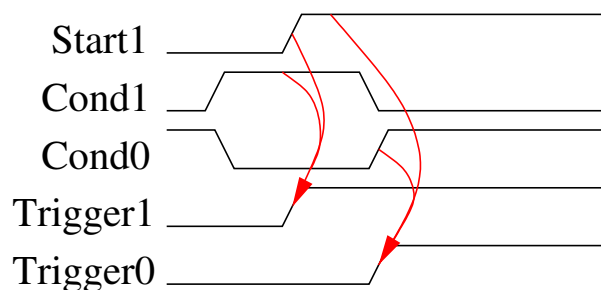


FIGURE 3.5 – Codage double rails non respecté

- **Deux résultats d'évaluation pour un seul cycle** : si *Cond*='1' reste stable pendant tout le cycle d'horloge et que l'entrée *Start1* est active, on obtient une sortie *Trigger1* active. Mais si une nouvelle donnée *Start* se présente en entrée du moniteur primitif, un nouveau résultat est produit pour ce moniteur primitif et ce cycle d'évaluation. Cela correspond aussi à un comportement incorrect du moniteur primitif asynchrone car il ne peut y avoir qu'une évaluation de la propriété PSL par cycle d'horloge.

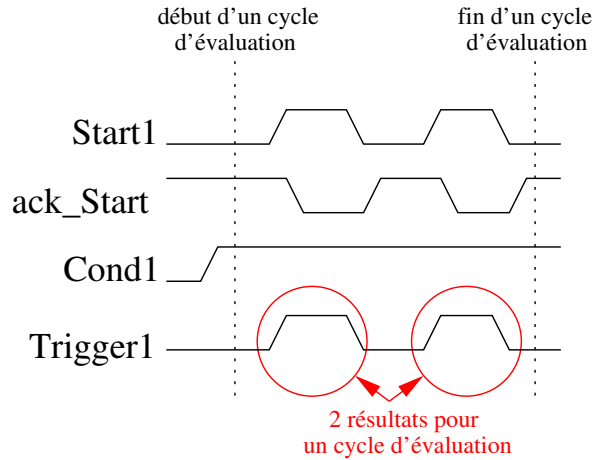
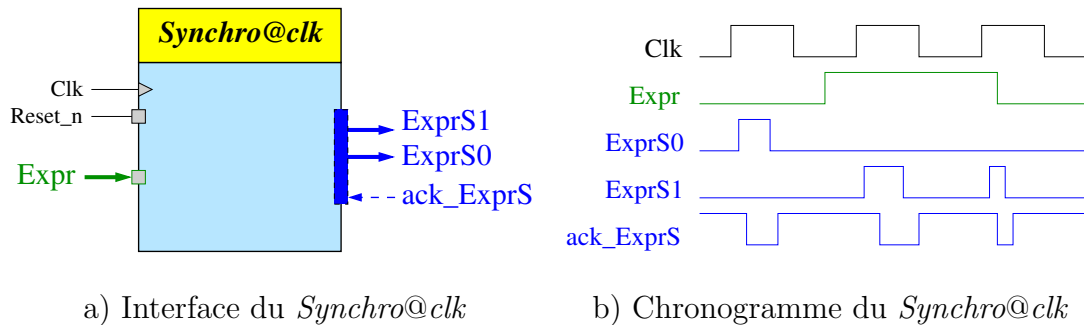


FIGURE 3.6 – Deux évaluations en un seul cycle

FIGURE 3.7 – Spécification du *Synchro@clk*

3.2.2 Nécessité d'un synchroniseur

Afin de résoudre les comportements non souhaités, il est nécessaire de réaliser une adaptation de protocole pour les entrées *Cond* et *Expr* d'un moniteur primitif asynchrone. Cette adaptation de protocole permet de transformer un signal booléen en une donnée double rails respectant le protocole quatre phases "poignée de mains". Il faut aussi déterminer à quels instants l'adaptation de protocole est réalisée. Le but des moniteurs asynchrones étant de vérifier des propriétés PSL sur un circuit synchrone, les opérandes d'une propriété PSL seront observés sur les fronts montant de l'horloge du circuit à vérifier puis la donnée échantillonnée est adaptée au protocole quatre phases. Pour réaliser ces deux fonctions, adaptation de protocole et synchronisation sur l'horloge, on implémente un synchroniseur appelé *Synchro@clk* dont le fonctionnement et l'interface sont explicités respectivement par la figure 3.7.a et 3.7.b. À chaque front montant d'horloge, le signal d'entrée *Expr* est échantillonné puis transformé en un signal double rails respectant le protocole quatre phases poignée de main. L'architecture et la conception du *Synchro@clk* seront développées dans le chapitre 4.

Le *Synchro@clk* introduit une différence notable dans le fonctionnement global des moniteurs asynchrones par rapport à leurs équivalents synchrones. En effet, dans le cas des moniteurs synchrones, le signal *Start* permet d'activer un moniteur primitif lui permettant de *pré-calculer* son résultat. Le résultat sera mémorisé au prochain front montant d'horloge. Dans le cas des moniteurs asynchrones, le *Synchro@clk* échantillonne *Expr* à chaque front montant d'horloge et fournit donc à chaque cycle une donnée respectant le protocole

asynchrone, ou jeton. Le signal *Start* sert désormais à valider ou à invalider l'évaluation de *Expr* faite au front précédent et donc à consommer le jeton *Expr* correspondant. On voit que le signal *Start* est utilisé pour *post-calculer* les sorties du moniteur asynchrone. On pourrait qualifier le signal *Start* de **pré-Start** pour les moniteurs synchrones et de **post-Start** pour les moniteurs asynchrones. Cependant, dans le reste du manuscrit on gardera la notation *Start* pour les moniteurs synchrones et asynchrones.

On va illustrer le fonctionnement du signal *Start* pour les moniteurs asynchrones grâce à la propriété A_1 :

property A_1 is **assert** always(*ticket and valide*) $\rightarrow$ **next!** *open until!_ fin_passage*);

Afin de comparer les solutions synchrones et asynchrones, la trace illustrant le comportement du moniteur synchrone est rappelée figure 3.8 et la trace illustrant le comportement du moniteur asynchrone est donnée figure 3.9. Dans le cas des moniteurs synchrones, la vérification d'une propriété débute au cycle suivant le *Reset_n* synchrone, c'est-à-dire que si *Reset_n* est actif au cycle $\#n$, le cycle où débute la vérification est le cycle $\#n + 1$. Afin de comparer nos moniteurs avec leurs équivalents synchrones, ces derniers seront initialisés de la même façon, avec un signal *Reset_n* synchrone.

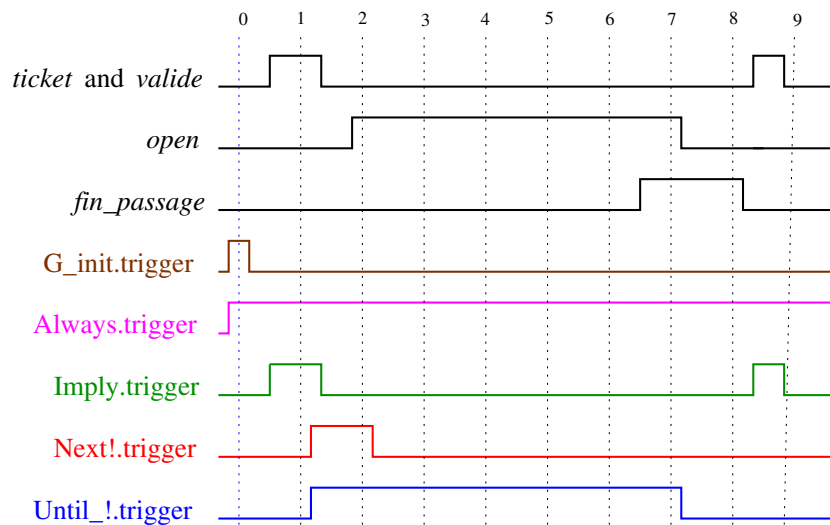


FIGURE 3.8 – Propagation des jetons dans le cas synchrone

On suppose que le moniteur asynchrone a été correctement initialisé au cycle $\#0$ (signal *Reset_n*='0' lors du front montant de *Clk*). Le moniteur primitif *G_init* génère un jeton *Start1*. Lors du front montant d'horloge suivant (cycle $\#1$), chaque moniteur primitif évalue ses opérandes à l'aide des *Synchro@clk*. Le moniteur primitif *M_always* a un jeton *Start1* présent sur son port d'entrée *Start*, il génère donc un jeton *Trigger1* pour le moniteur primitif suivant (*M_imply*). Comme *ticket_valide*='1' au cycle $\#1$, le *Synchro@clk* du *M_imply* a généré une sortie *ExprS1*, la partie droite de l'implication doit être prise en compte. Un rendez vous est réalisé entre le signal *Start1* et *ExprS1* et permet de générer un jeton *Trigger1* pour le moniteur primitif suivant (*M_next!*).

Lors du front montant d'horloge, chaque *Synchro@clk* de chaque moniteur primitif évalue ses opérandes et génère une sortie *ExprS1* ou *ExprS0*. Il reçoit ensuite un jeton *Start1* s'il doit prendre en compte l'évaluation de *Expr* qui vient d'être réalisée, *Start0* sinon. Ce comportement est différent de celui des moniteurs synchrones où le signal *Start* activant l'évaluation des opérandes d'un moniteur primitif se propage avant le front montant de

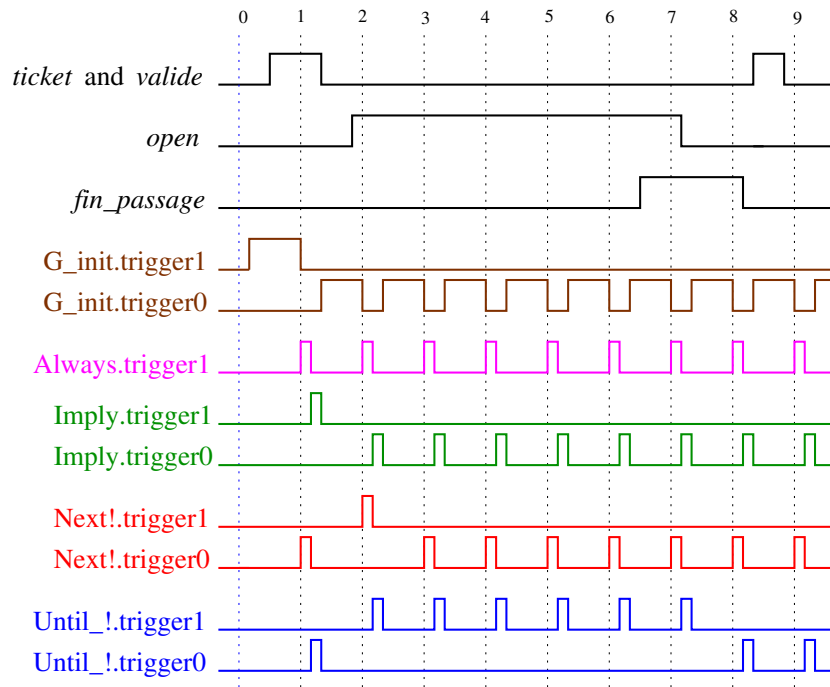


FIGURE 3.9 – Propagation des jetons dans le cas asynchrone

l'horloge.

3.2.3 Lien entre sémantique et implémentation matérielle

Les opérateurs PSL ont été classifiés en quatre groupes (*c.f.* partie 1.4.2.3, page 24) :

1. sans retard et sans ré-entrance
2. avec retard et sans ré-entrance
3. sans retard et avec ré-entrance
4. avec retard et avec ré-entrance

L'appartenance d'un opérateur PSL à l'un ou l'autre de ces groupes a une influence directe sur son implémentation matérielle. La suite du manuscrit montre comment sont implémentés les moniteurs asynchrones et comment la classification des opérateurs PSL en fonction de leur sémantique permet l'obtention d'architecture générique à plusieurs opérateurs d'un même groupe.

3.2.4 Les opérateurs sans cycle de retard, sans ré-entrance

C'est le cas des moniteurs primitifs `M_imply` et `M_signal`. Le moniteur `M_imply` implémente l'opérateur $\rightarrow$ dont l'opérande droit est une FL et le moniteur `M_signal` permet de vérifier la valeur d'un signal booléen.

3.2.4.1 Architecture générique

Les moniteurs primitifs asynchrones n'introduisant ni cycle de retard, ni ré-entrance, sont implémentés à l'aide de l'architecture générique de la figure 3.10.

Cette architecture est composée de trois parties distinctes :

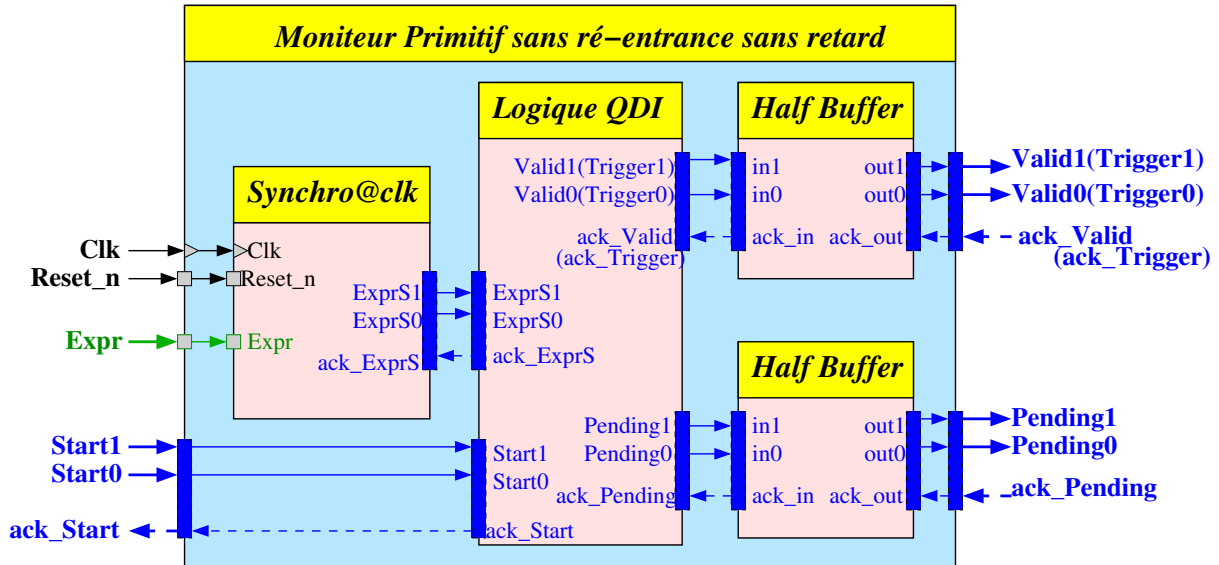


FIGURE 3.10 – Moniteur primitif asynchrone sans ré-entrance et sans cycle de retard

1. La synchronisation : implémente la synchronisation et l'adaptation de protocole et génère une sortie *ExprS1* ou *ExprS0* grâce au bloc *Synchro@clk*
2. La partie logique QDI : implémente la sémantique de l'opérateur et fournit les sorties voulues en fonction de la valeurs des signaux *Start* et *Expr*,
3. La partie mémorisation : permet d'implémenter le protocole quatre phases WCHB grâce aux blocs *Half Buffer* (c.f. partie 2.7, page 50).

Dans cette architecture, le moniteur primitif n'a qu'un opérande. Pour les moniteurs primitifs évaluant deux opérandes, il suffit d'ajouter un port d'entrée *Cond* à l'interface et de connecter ce signal à un nouveau synchroniseur. Le synchroniseur fournit les sorties *CondS1* et *CondS0* qui sont connectées à la partie "Logique QDI".

On remarque aussi que le moniteur primitif présenté correspond à un opérateur fort car il y a une sortie *Pending*. Pour les versions faibles, il suffit que la partie "Logique QDI" ne génère pas de signal *Pending* et que l'on enlève le *Half Buffer* correspondant à cette sortie.

3.2.4.2 Implémentation de la partie "Logique QDI"

On détaille ici l'implémentation de la partie "Logique QDI" du moniteur primitif *M_imply*. C'est un opérateur faible, il n'y a donc pas de sortie *Pending*. De plus ce moniteur primitif ne possède qu'un opérande *Cond*. On a donc les possibilités suivantes pour les combinaisons entre *Start* et *CondS*¹ :

- *Start0.CondS0* + *Start0.CondS1* : le moniteur n'a pas à tenir compte de l'évaluation de *Cond* faite au front montant d'horloge précédent. La partie droite de l'implication n'a pas besoin d'être vérifiée. On génère donc une sortie *Trigger0*.
- *Start1.CondS0* : le moniteur doit tenir compte de l'évaluation de *Cond*. Comme *CondS0* est actif, cela signifie que *Cond* était à '0' au front montant d'horloge pré-

1. On rappelle que *Cond* et *Start* sont codés en double rails, ainsi *CondS1*= '1' signifie que *Cond* valait '1' au front d'horloge précédent

cédent. La partie droite de l'implication n'a pas à être vérifiée et le moniteur génère une sortie *Trigger0*.

- *Start1.CondS1* : le moniteur doit tenir compte de l'évaluation de *Cond*. Comme *CondS1* est actif, cela signifie que *Cond* était à '1' au front montant d'horloge précédent. La partie droite de l'implication doit être vérifiée et le moniteur génère une sortie *Trigger1*.

Les fonctions que doit réaliser la partie "Logique QDI" peuvent être exprimées sous la forme de sommes de produits :

$$\begin{aligned} \text{Trigger1} &= \text{Start1.CondS1} \\ \text{Trigger0} &= \text{Start1.CondS0} + \text{Start0.CondS1} + \text{Start0.CondS0} \end{aligned}$$

On réalise une synthèse de type DIMS (c.f. partie 2.6.2, page 48). On obtient ainsi l'implémentation présentée figure 3.11 pour la partie "Logique QDI" du moniteur primitif asynchrone *M_imply*.

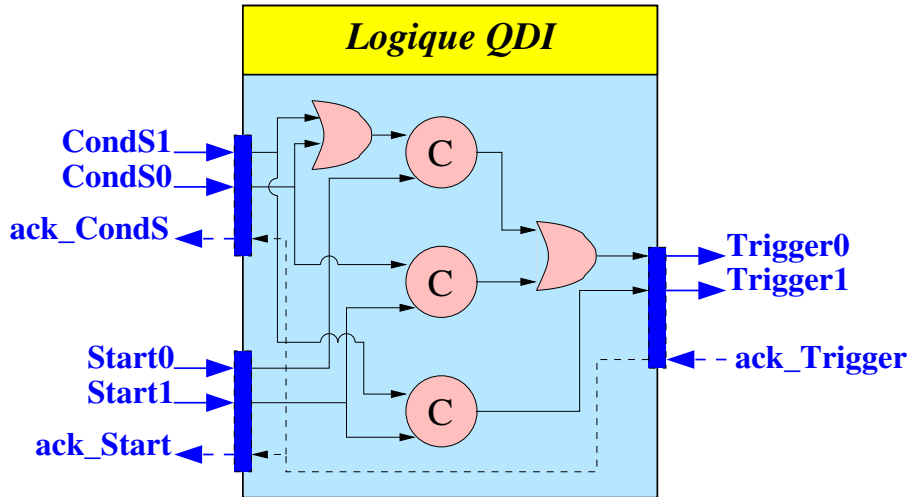


FIGURE 3.11 – Implémentation de la partie Logique QDI pour le moniteur primitif →

3.2.5 Les opérateurs sans cycle de retard, avec ré-entrance

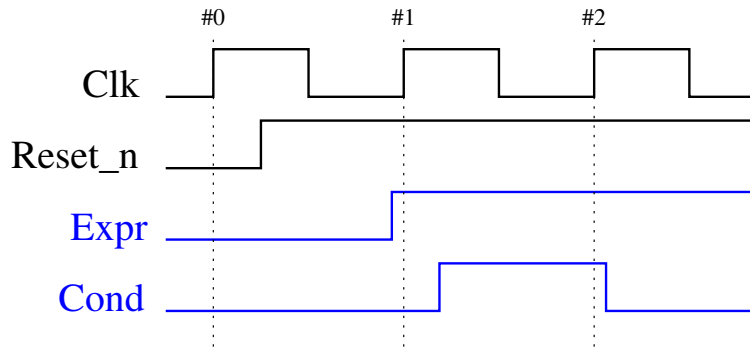
Dans cette partie, on présente l'architecture et la synthèse des moniteurs primitifs sans cycle de retard et avec ré-entrance. C'est le cas des opérateurs suivants : *until**, *before**, *always*, *never* et *eventually!*. On notera *M_until* le moniteur primitif asynchrone correspondant à l'opérateur PSL *until*.

3.2.5.1 Le phénomène de ré-entrance : l'exemple du *until!*

Afin d'illustrer le phénomène de ré-entrance, on prend l'exemple suivant :

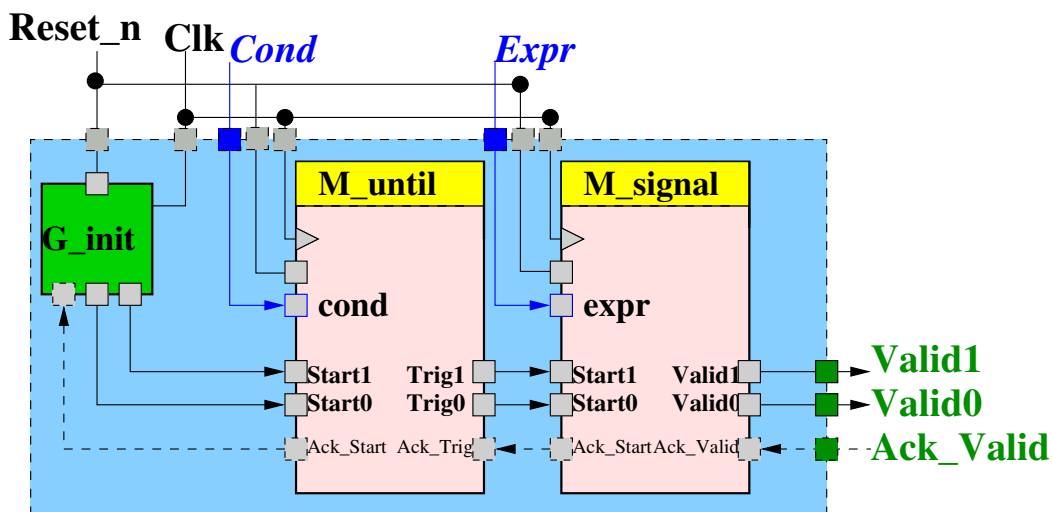
property P_2 is assert *Expr until! Cond*;

La trace de la figure 3.12 satisfait P_2 . L'initialisation du moniteur asynchrone évaluant P_2 (c.f. figure 3.13) est réalisée au cycle #0, donc l'évaluation de P_2 commence au cycle #1.

FIGURE 3.12 – Chronogramme satisfaisant P_2

Au cycle #1, le `M_until` reçoit un jeton `G_init.trigger1`² lui indiquant qu'il doit tenir compte de l'évaluation de `Cond`. Comme `Cond='0'` à ce cycle, la validité de P_2 dépend donc de la validité de `Expr`. Or, au cycle #1, `Expr='1'`. L'évaluation de P_2 n'est donc ni "Failed", ni "Holds Strongly" et doit se poursuivre au cycle #2. C'est ce phénomène que l'on appelle **ré-entrance**.

Au cycle #2, le `M_until!` reçoit un jeton `G_init.trigger0` lui indiquant qu'il ne doit pas tenir compte de l'évaluation de son opérande faite à ce cycle. Or, l'évaluation de P_2 est toujours en cours et le `M_until` aurait dû tenir compte de l'évaluation de `Cond`. Afin de pallier cette situation, on rajoute un signal interne `Restart` au `M_until!`. Le signal `Restart`, tout comme `Start` est codé en double rails. On a `Restart1` actif quand le `M_until!` doit tenir compte de son évaluation au prochain cycle, `Restart0` sinon. Le `M_until!` doit donc tenir compte de son évaluation lorsque au moins un des deux signaux `Start1` ou `Restart1` est actif. Cela revient à faire le "OU double rails" entre `Start` et `Restart`.

FIGURE 3.13 – Moniteur asynchrone de P_2

2. On rappelle que `G_init.trigger1` correspond à la sortie `Trigger1` du `G_init` active

3.2.5.2 Signal *Restart* et architecture générique

On reprend l'exemple précédant avec un signal *Restart* dans le *M_until!*. Lors de l'initialisation (cycle #0), le *M_until!* crée un jeton *Restart0*. Au cycle #1, on a le signal *Start1* actif provenant du *G_init* et le signal *Restart0* actif provenant de l'initialisation du *M_until!*. C'est le rail codant '1' qui est actif en sortie du "OU double rails" indiquant au *M_until!* qu'il doit tenir compte de son évaluation et devra en plus ré-évaluer *Cond* au prochain cycle : un jeton *Restart1* est généré. Enfin au cycle #2, on refait le "OU double rails" entre *Start* et *Restart*. Le *M_until!* doit tenir compte de son évaluation. Ainsi le phénomène de ré-entrance est pris en compte car *Cond* est bien ré-évalué au cycle #2. On obtient l'architecture présentée figure 3.14 pour les moniteurs primitifs sans cycle de retard et avec ré-entrance. Le signal *Restart* est généré en interne du *M_until!* en ajoutant un signal de sortie *Restart* au bloc "Logique QDI".

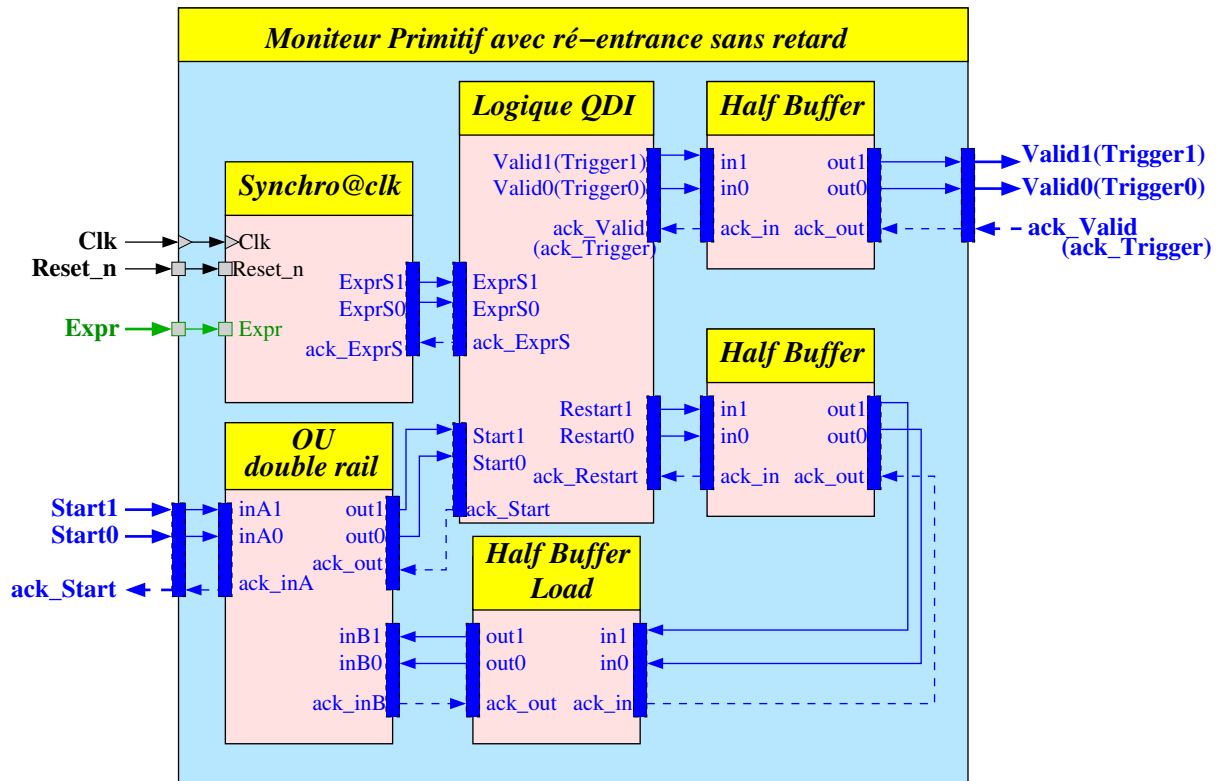


FIGURE 3.14 – Moniteur primitif avec ré-entrance et sans cycle de retard

On remarque l'ajout d'un bloc "Half Buffer Load" sur le chemin du signal *Restart* entre les blocs "Logique QDI" et "OU double rails". C'est ce bloc qui permet de créer le jeton *Restart0* dans le *M_until!* lors de l'initialisation.

L'implémentation de la partie "Logique QDI" à l'aide de la logique DIMS est réalisée comme cela a été précédemment expliqué (*c.f.* partie 3.2.4.2). On remarque aussi que la version présentée ici est une version faible car il n'y a pas de sortie *Pending*. L'ajout de la sortie *Pending* se fait en ajoutant une sortie *Pending* au bloc "Logique QDI" et un "Half Buffer". L'architecture générique pour la version forte des opérateurs sans cycle de retard et sans ré-entrance est présentée dans l'annexe A.2, figure A.4.

3.2.6 Les opérateurs introduisant des cycles de retard

3.2.6.1 Les opérateurs next et next!

Le moniteur primitif asynchrone correspondant à l'opérateur `next`, le `M_next` est un connecteur. Le principe de ce moniteur primitif est d'introduire un cycle de retard entre la réception d'un jeton entrant *Start* et l'émission des sorties *Trigger*. Lorsqu'un jeton *Start* apparaît sur l'entrée du `M_next` au cycle N , on veut qu'il soit transmis sur la sortie *Trigger* au cycle $N + 1$. De cette façon, lors de l'évaluation de `next  $\varphi$` , l'évaluation de φ commencera un cycle après l'activation du `M_next`.

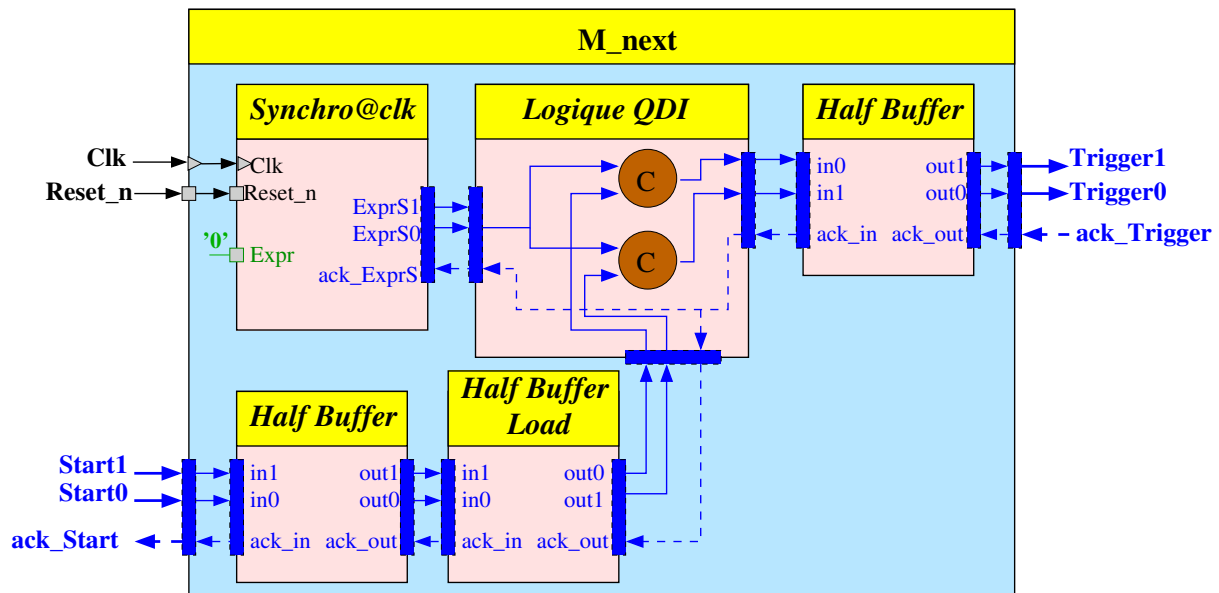


FIGURE 3.15 – Architecture du moniteur primitif `M_next`

Architecture du `M_next` L'architecture complète du `M_next` est présentée figure 3.15. Dans les moniteurs primitifs présentés précédemment, lorsqu'un jeton *Start* est présent sur les entrées, il est consommé par le bloc "Logique QDI" qui réalise un "rendez-vous" entre le jeton *Start* et les sorties du `Synchro@clk`. Le bloc "Logique QDI" transmet ensuite son résultat au moniteur primitif suivant. Toutes ces opérations ont lieu au cours du même cycle d'évaluation.

Afin d'introduire un cycle de retard entre la réception du jeton *Start* et la transmission des sorties, on rajoute un "Half Buffer Load" entre l'entrée *Start* et la partie "Logique QDI". De cette manière, le jeton présent sur l'entrée *Start* ne sera consommé qu'après le jeton présent dans le "Half Buffer Load". Comme le `Synchro@clk` n'évalue son opérande qu'une fois par cycle, il n'y a qu'un jeton *Expr* disponible par cycle. Le jeton *Start* ne sera consommé qu'au prochain cycle et donc les sorties *Trigger* correspondant à ce jeton *Start* ne seront disponibles qu'un cycle plus tard.

On remarque que l'opérateur `next` ne possédant pas d'opérande booléen, l'entrée *Expr* du `Synchro@clk` est reliée à la masse et la sortie du `Synchro@clk` sera donc *ExprS0* à tous les cycles d'évaluation. On n'utilise donc que *ExprS0* pour les rendez vous. Le "Half Buffer"

présent entre l'entrée *Start* du moniteur et le "Half Buffer Load" est indispensable pour éviter les deadlocks.

Propagation d'un jeton *Start* dans le *M_next* La transmission d'un jeton de l'entrée *Start* vers la sortie *Trigger* du moniteur primitif *M_next* est illustrée par la figure 3.16 où l'on a supprimé le nom des signaux et les "Half Buffer" pour plus de visibilité.

Lors de l'étape 1, le moniteur *M_next* reçoit un jeton *Start* résultat de l'évaluation du cycle *N* des moniteurs primitifs le précédant. Le *Synchro@clk* a fourni les sorties correspondant à l'évaluation du cycle *N*. Le jeton *Start* du cycle précédant $N - 1$ est chargé dans le "Half Buffer Load".

Un rendez-vous est effectué entre le jeton *Start* du cycle $N - 1$ et les sorties du *Synchro@clk* du cycle *N* (étape 2) pour générer la sortie du cycle d'évaluation *N* (étape 3). A cette étape, le jeton *Start* du cycle *N* peut se propager jusqu'au "Half Buffer Load" désormais vide.

Au cycle d'évaluation suivant ($N + 1$), un nouveau jeton *Start* se présente sur les entrées du moniteur primitif et de nouvelles sorties sont générées par le *Synchro@clk* (étape 4). Le jeton *Start* du cycle *N* peut être consommé grâce au rendez vous effectué avec les sorties du *Synchro@clk* (étape 5). On obtient ainsi un signal de sortie au cycle $N + 1$ (étape 6) à l'aide du jeton *Start* du cycle *N*. On a bien introduit un cycle de retard entre l'arrivée d'un jeton *Start* et la sortie d'un jeton *Trigger* correspondant.

On utilise un "Half Buffer Load" et non un "Half Buffer" pour des problèmes dus à l'initialisation. On se place au cycle #1 après que l'initialisation ait été correctement réalisée (cycle #0) et l'on utilise un simple "Half Buffer". Dans ce cas il n'y a pas de jeton #0 dans le "Half Buffer" et le jeton *Start* du cycle #1 peut directement être envoyé vers le bloc "Logique QDI" car il n'y aura dans le moniteur que des jetons numérotés #1. L'utilisation d'un "Half Buffer Load" permet de charger un jeton #0 à l'initialisation et ainsi la sortie du cycle #1 correspondra au rendez vous entre le jeton #0 du "Half Buffer Load" et les sorties #1 du *Synchro@clk*. Le rendez vous utilisant le jeton *Start* #1 ne se fera qu'au cycle d'évaluation #2.

On remarque aussi que pour générer le moniteur primitif asynchrone *M_next*[*K*] correspondant à l'opérateur *next*[*K*], il suffit de connecter à la suite *K* moniteurs primitifs *M_next* comme illustré par la figure 3.17.

Architecture du *M_next!* Contrairement aux architectures présentées précédemment où il suffisait d'ajouter une sortie *Pending* au bloc "Logique QDI" pour obtenir l'architecture correspondant à la version forte d'un opérateur, il faut raisonner différemment dans le cas du moniteur *M_next!* correspondant à l'opérateur *next!*. En effet, quand un signal *Start1* est présent en entrée du moniteur primitif *M_next!*, ce dernier est actif et doit fournir une sortie *Pending1*. Or le jeton *Start1* du *M_next!* ne sera consommé par la partie "Logique QDI" qu'un cycle plus tard. Afin d'avoir immédiatement un signal *Pending1* en sortie du moniteur primitif *M_next!* lorsqu'un jeton *Start1* est présent sur ces entrées, il faut dupliquer le jeton *Start1* à l'aide d'une fourche et envoyer directement l'une des sortie de la fourche sur la sortie *Pending* du moniteur primitif.

On voit sur la figure 3.18 que l'on a juste rajouté une fourche sur le signal *Start1*, les sorties de la fourche étant utilisées comme signal *Pending* pour le moniteur primitif *M_next!* et comme signal *Start* pour le moniteur primitif *M_next*.

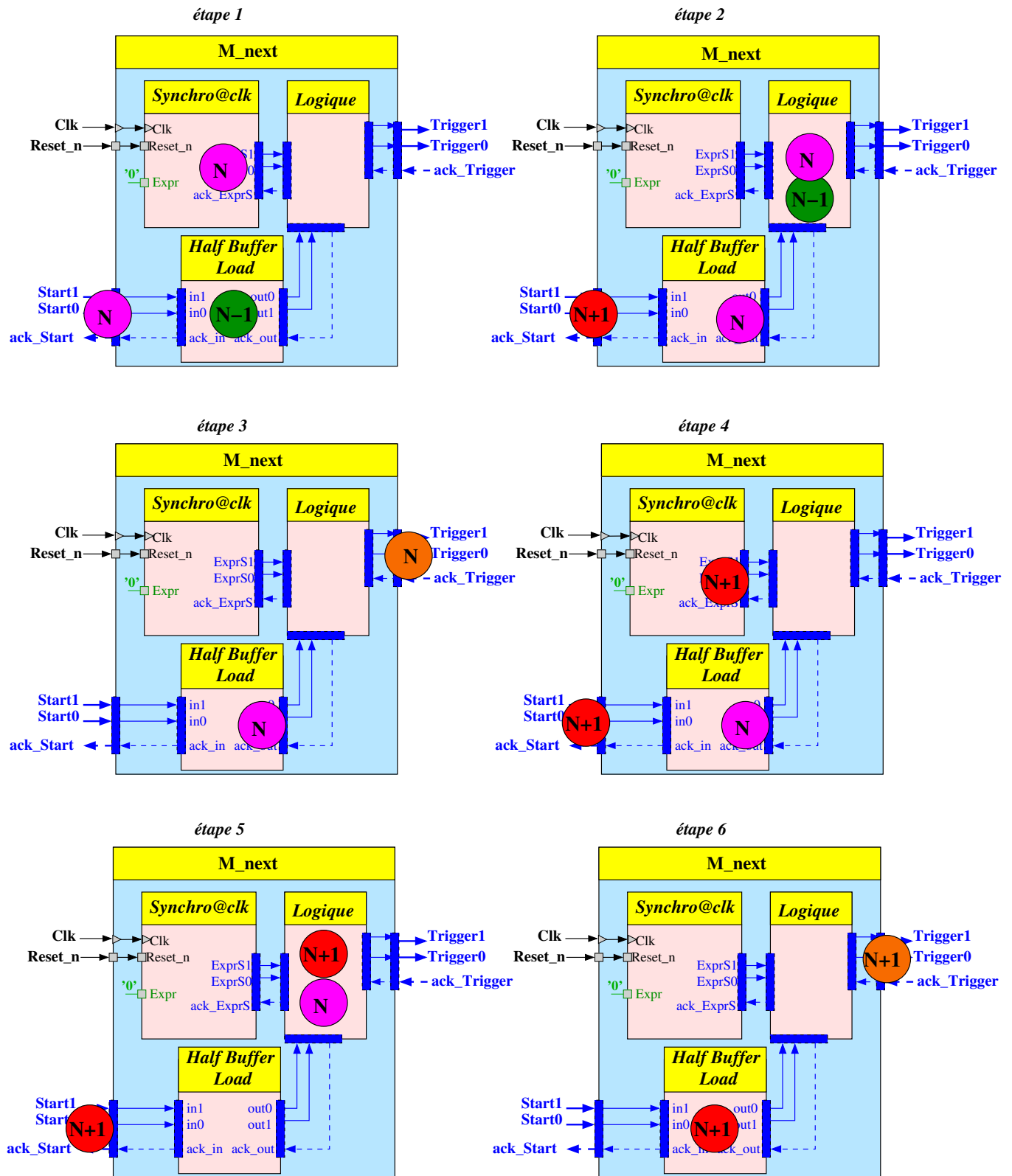


FIGURE 3.16 – Introduction d'un cycle de retard dans la transition d'un jeton

au niveau matériel par l'émission d'un jeton *Trigger1* en sortie du i^{ime} étage aux cycles i et $i + 1$.

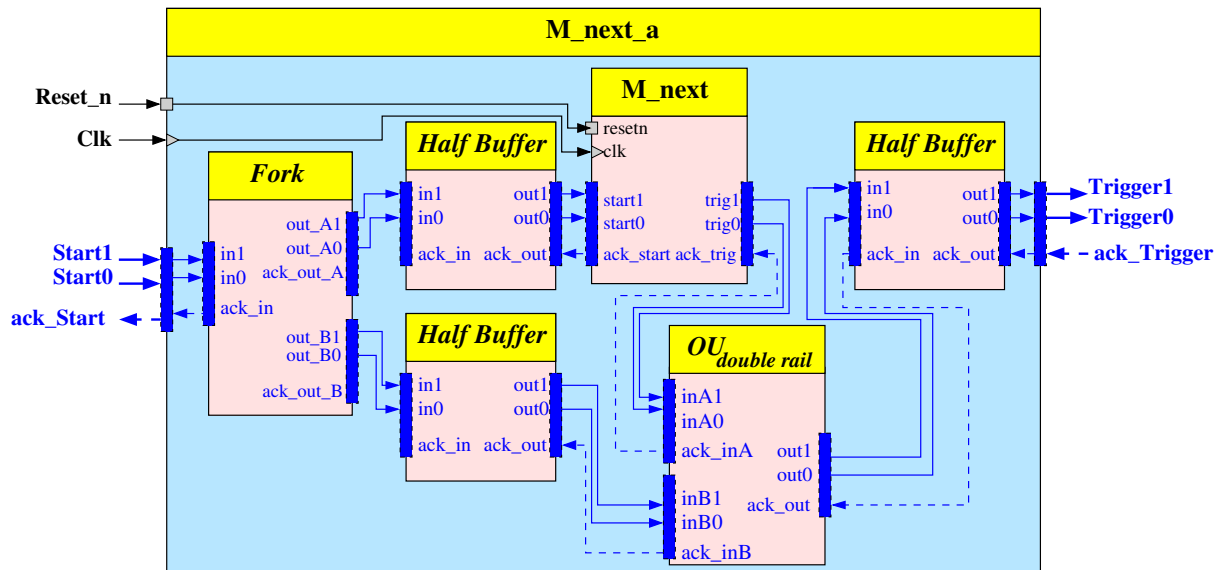


FIGURE 3.19 – Bloc matériel *M_next_a*

Il suffit pour cela de dupliquer le jeton *Start1* de l'étage i grâce au bloc *Fork* (c.f. partie 2.7, page 50) et de l'envoyer directement sur le port de sortie *Trigger1* de ce même étage. Cela permet d'avoir un jeton *Trigger1* en sortie du i^{ime} étage au cycle i . L'autre jeton *Start1* issu de la duplication de l'entrée *Start* est envoyé vers un moniteur primitif *M_next* permettant ainsi la présence d'un jeton *Trigger1* en sortie de l'étage i au cycle $i + 1$. Le bloc matériel permettant le comportement précédent appelé *M_next_a* est illustré par la figure 3.19.

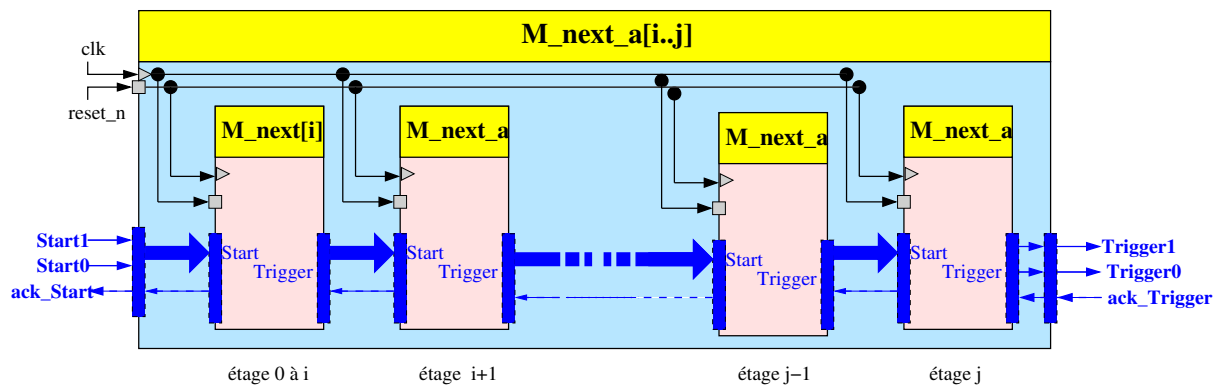


FIGURE 3.20 – Moniteur primitif asynchrone *M_next_a[i..j]*

Pour créer un moniteur primitif complet *M_next_a[i..j]*, il suffit de d'utiliser tout d'abord un *M_next[i-1]* puis des blocs *M_next_a* pour les étages i à j . Cette structure est illustrée par la figure 3.20.

On remarque que l'on n'a pas d'architecture générique pour les moniteurs primitifs implémentant des opérateurs introduisant des cycles de retard sans ré-entrance mais l'on

réutilise le moniteur primitif M_next pour introduire un cycle de retard dans l'évaluation. On procède de la même manière pour obtenir le moniteur primitif $M_next_e[i..j]$ (*c.f.* appendice A.3.3).

3.2.6.3 Opérateurs introduisant un ou plusieurs cycles de retard avec ré-entrance

C'est le cas des moniteurs primitifs $M_next_event_e$, $M_next_event_a$ et M_next_event correspondant respectivement aux opérateurs PSL $next_event_e$, $next_event_a$ et $next_event$. On va expliciter la construction du $next_event(b)[K]$ dont la règle de ré-écriture est rappelée : opérateur est définie de la sorte :

$$next_event!(b)[K](\varphi) \doteq next_event!(b) \overbrace{(next! next_event!(b) \dots (next! next_event!(b)(\varphi)) \dots)}^{K-1 \text{ fois}}$$

On remarque immédiatement dans la règle de ré-écriture du $next_event(b)[K] \varphi$ que l'on a besoin de réaliser deux "opérations" distinctes : $next_event(b)$ et $next next_event(b)$. La première de ces opérations est implémentée par le moniteur primitif M_next_event qui présente l'architecture d'un moniteur primitif sans cycle de retard avec ré-entrance (*c.f.* partie 3.2.5). Il est donc aisé d'implémenter ce moniteur primitif (*c.f.* annexe A.2.5.3). La seconde opération, $next next_event(b)$, est implémentée facilement en connectant tout d'abord un moniteur primitif M_next suivi d'un moniteur primitif $M_next_event(b)$. On appelle ce bloc le M_next_eventK (*c.f.* figure 3.21).

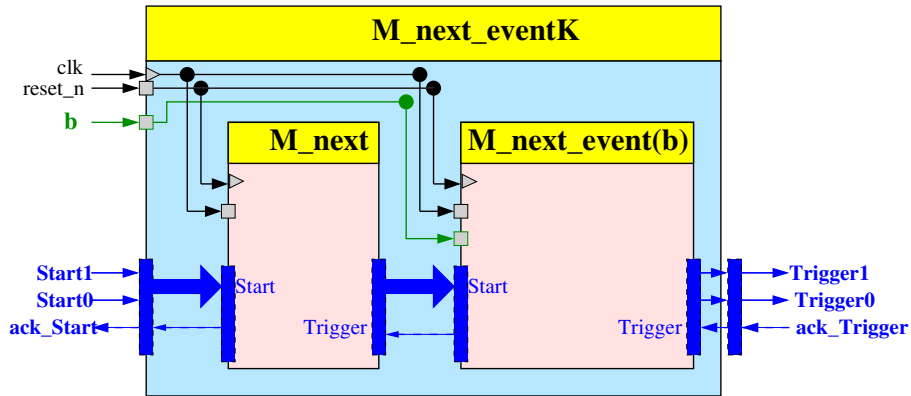
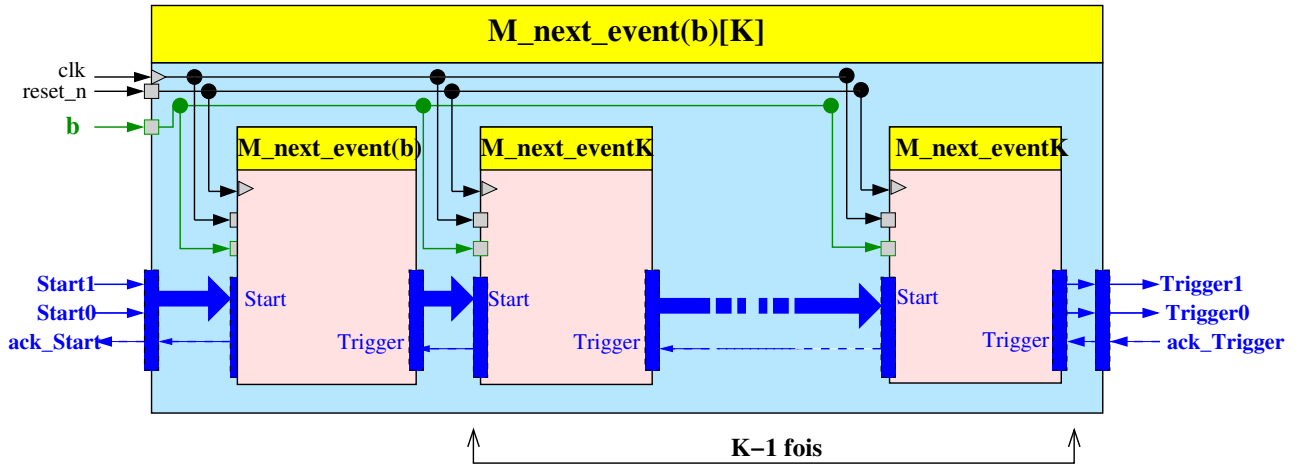


FIGURE 3.21 – Bloc asynchrone M_next_eventK

Pour obtenir le moniteur primitif complet du $M_next_event(b)[K] \varphi$ il suffit enfin de connecter tout d'abord un bloc M_next_event suivi de $k - 1$ blocs M_next_eventK . On illustre la structure du moniteur primitif complet figure 3.22.

Dans ce cas aussi on n'a pas d'architecture générique pour les moniteurs primitifs implémentant des opérateurs introduisant des cycles de retard avec ré-entrance mais l'on réutilise le moniteur primitif M_next pour introduire un cycle de retard dans l'évaluation. La ré-entrance est prise en compte dans le $M_next_event(b)$. On procède de la même manière pour obtenir les moniteurs primitifs $M_next_event_e$ et $M_next_event_a$ (*c.f.* appendice A.4.1 et A.4.2).

FIGURE 3.22 – Moniteur primitif $M_next_event(b)[K]$

3.2.7 Autres moniteurs primitifs

Ces autres moniteurs primitifs correspondent à des opérateurs dont l'évaluation est réalisée en un cycle. C'est le cas notamment des opérateurs purement booléens et des opérateurs FL `and` et `or`. On présente aussi l'architecture du `G_init`.

Les opérateurs booléens Lorsque les deux opérandes de l'opérateur $\rightarrow$ sont booléens, il est judicieux de ne pas utiliser le moniteur primitif asynchrone `M_imply` mais plutôt des portes logiques implémentant la fonction $\neg OP1 \vee OP2$ où $OP1$ et $OP2$ sont des booléens. Le résultat de cette fonction est ensuite simplement connectée à un `M_signal`.

De la même façon, lorsque leurs deux opérandes sont booléens, les opérateurs `or`, `and` et `iff` seront implémentés en utilisant de simples portes standards dont la sortie est connectée à un `M_signal`. L'exemple de l'opérateur $\rightarrow$ avec deux opérandes booléens est détaillé en annexe A.5.1.

Opérateur `and` et `or` Dans le sous-ensemble simple de PSL, la propriété φ `or` ψ possède nécessairement un opérande booléen. Or, dans le sous-ensemble simple de PSL on a $\varphi \rightarrow \psi$ avec φ booléen et $\varphi \rightarrow \psi \stackrel{def}{=} \neg \varphi \vee \psi$. Il suffit donc de connecter l'opérande booléen de l'opérateur `or` à un inverseur puis de connecter la sortie de l'inverseur à l'entrée `Cond` d'un moniteur primitif `M_imply`. La sortie du moniteur primitif `M_imply` est ensuite connectée à l'entrée des moniteurs primitifs suivants évaluant ψ .

L'opérateur `and` permet de réaliser l'évaluation d'une propriété PSL de type φ `and` ψ . Il suffit d'implémenter le moniteur asynchrone correspondant à l'évaluation de φ et celui correspondant à l'évaluation de ψ . Les sorties `Valid` des deux moniteurs sont ensuite connectées grâce à un "AND" double rails. La sortie du "AND" donne la validité de φ `and` ψ .

Plus de détails sur l'utilisation et la connection de ces blocs sont donnés dans l'annexe A.5.2 et A.5.3.

Le cas particulier du `G_init` Lorsqu'il est initialisé avec un signal `Reset_n`, le bloc matériel `G_init` doit d'abord fournir un signal `Trigger1` pour confirmer l'évaluation de la

propriété faite par les autres moniteurs primitifs. Pour tous les cycles suivants, le `G_init`

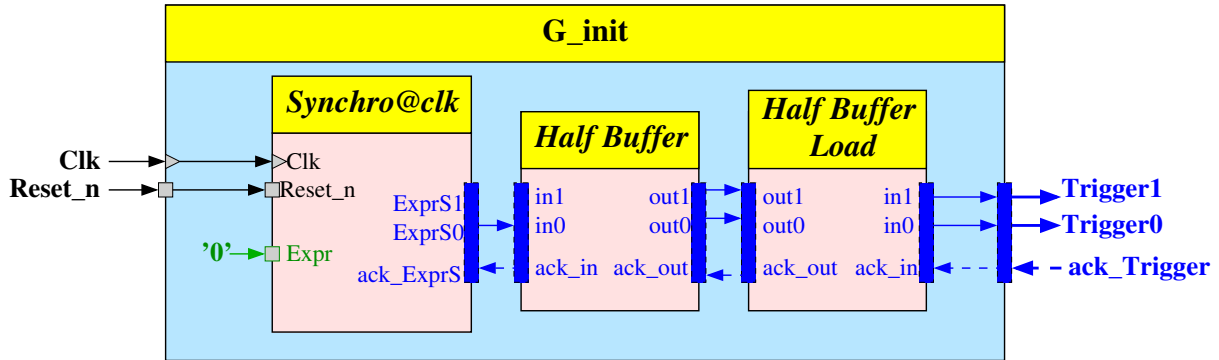


FIGURE 3.23 – Moniteur primitif `G_init`

doit fournir un signal *Trigger0*. Le signal *Trigger1* est généré grâce à un "Half Buffer Load". La génération des autres jetons peut être facilement réalisée par un *Synchro@clk* dont l'entrée *Expr* est connectée à la masse et dont la sortie *ExprS0* est connectée à l'entrée '0' du "Half Buffer". Le "Half Buffer" est nécessaire avant le "Half Buffer Load" pour éviter tous "deadlocks". On obtient la structure présentée figure 3.23 pour le `G_init`.

3.3 Validation par simulation numérique

La construction des moniteurs primitifs asynchrones terminée, il est indispensable de vérifier leur comportement. La simulation représente une première vérification simple à mettre en œuvre pour valider notre méthode d'interconnexion et la bibliothèque de moniteurs primitifs. Les simulations sont réalisées à l'aide de l'outil MODELSIM de la société MENTOR GRAPHICS.

Pour réaliser ces simulations, il est indispensable d'avoir un ou plusieurs synchroniseurs dans chacun des moniteurs primitifs utilisés dans le moniteur asynchrone. À ce stade, une simple description comportementale de notre synchroniseur est requise. Le code correspondant au *Synchro@clk* comportemental est présenté dans l'annexe B.1.

Le code de la plateforme **Horus** a été modifié par N.GLEITZ et J.SESTER afin de permettre la connexion de moniteurs primitifs asynchrones. La méthode d'interconnexion est toujours aussi efficace qu'en synchrone puisqu'il suffit d'un clic pour obtenir quasi-instantanément le code VHDL ou VERILOG de la propriété. On compile grâce à cette plateforme l'ensemble des propriétés présentées dans l'introduction à la sémantique de PSL(*c.f.* page 17). Les moniteurs asynchrones obtenus ont été simulés en parallèle de leurs équivalents synchrones qui ont été prouvés dans [MAB06]. La comparaison des résultats de simulation entre synchrones et asynchrones nous permet de vérifier la fonctionnalité des moniteurs asynchrones.

3.3.1 Simulation et validation de la bibliothèque de moniteurs primitifs

On simule tout d'abord des propriétés très simples permettant de vérifier le bon fonctionnement d'un moniteur primitif. On commence tout d'abord par vérifier le fonctionne-

ment du *M_signal* grâce à la propriété suivante :

P₁₄ is **assert** *a* ;

Le moniteur asynchrone implémentant *P₁₄* est composé d'un *G_init* et d'un *M_signal*. De plus, l'évaluation de *P₁₄* ne dure qu'un cycle impliquant qu'il y a peu de trace possible à vérifier. On initialise les moniteurs synchrones et asynchrones grâce à un signal *Reset_n* actif au cycle *n*, puis au cycle *n + 1*. Il y a deux cas possibles :

1. Le signal *a* est actif et le résultat de l'évaluation est *Valid1*.
2. Le signal *a* est inactif et le résultat de l'évaluation est *Valid0*.

On peut désormais vérifier d'autres moniteurs primitifs comme le *M_before!*. On utilise la propriété *P₁₅*

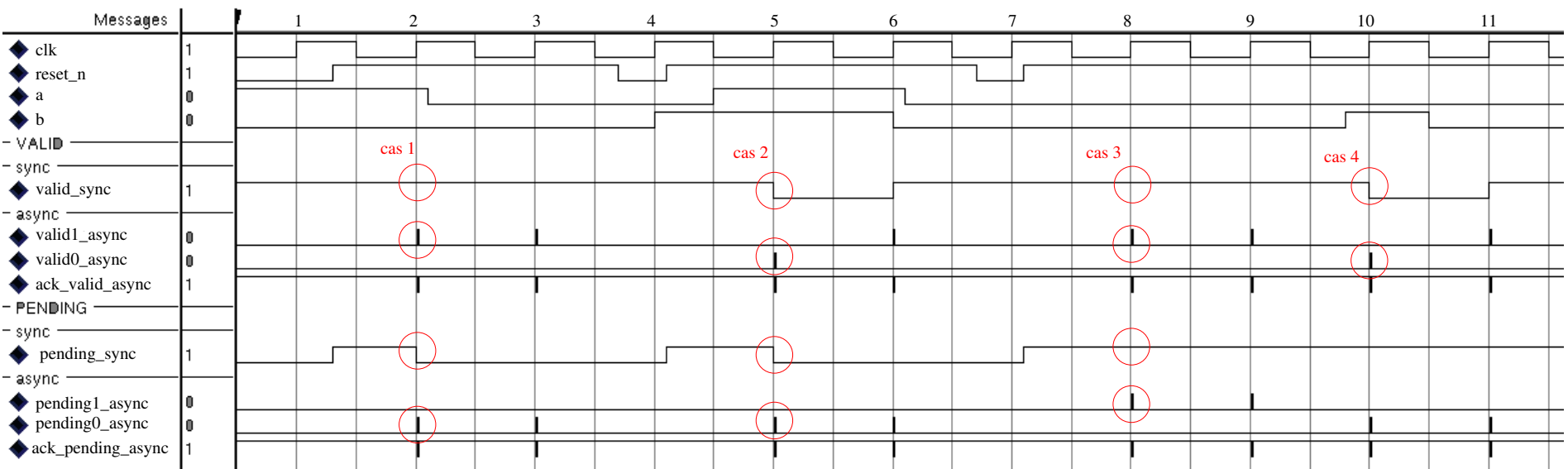
P₁₅ is **assert** *a before! b* ;

Le moniteur asynchrone implémentant *P₁₅* est composé d'un *G_init* et d'un *M_before!*. Le résultat de la simulation des moniteurs synchrones et asynchrones évaluant *P₁₅* est donné figure 3.24.

Lors du premier front montant d'horloge, le signal *Reset_n* est dans l'état bas et l'initialisation du moniteur asynchrone est correctement effectuée. En effet, aucun résultat n'est produit pour ce premier cycle. On vérifie ensuite *a before! b* dans les cas suivants :

1. lorsque *a* et *b* sont vrais au même cycle. Dans ce cas, l'évaluation est valide et terminée. On constate que ce cas se produit lors du deuxième front montant de *Clk* et que l'on obtient effectivement les signaux *Valid_sync*, *Valid1_async* actifs ainsi que *Pending_sync*=0 et *Pending0_async* actifs.
2. lorsque *b* est inactif au cycle où *a* est actif pour la première fois. Cette évaluation correspond au cinquième front montant de *Clk*, juste après que les moniteurs n'aient été ré-initialisés par le signal *Reset_n* qui est au niveau bas lors du quatrième front montant de *Clk*.
3. lorsque *a* et *b* sont faux au même cycle. Dans ce cas, l'évaluation n'est pas terminée et doit se poursuivre au prochain cycle. On doit obtenir les signaux *Pending_sync* et *Valid_sync* actifs ainsi que les signaux *Pending1_async* et *Valid1_async*. Cette évaluation correspond au huitième front montant de *Clk* qui intervient après la ré-initialisation (septième front de *Clk*).
4. lorsque *b* est actif à un cycle où *a* est inactif et que *a* n'a jamais été actif avant. Cette évaluation correspond au dixième front montant de *Clk*. On obtient bien l'information que la propriété est violée grâce aux signaux *Valid_sync*=0 et *Valid0_async* actifs.

On procède de la même manière pour tous les moniteurs primitifs. Dans tous les cas on obtient des résultats identiques à la solution synchrone excepté pour les cycles suivant un cycle où la propriété est violée. Dans ce cas on peut observer des différences sur le signal *Pending* comme par exemple lors du cas 4 sur la trace de la figure 3.24. Ces différences ne sont pas importantes car la propriété ayant été violée, la suite de l'évaluation n'a plus aucun sens. L'ensemble des simulations permet de valider la bibliothèque de moniteurs primitifs asynchrones. Les simulations précédentes n'ayant permis de valider l'interconnexion des moniteurs primitifs que sur de petits exemples composés de deux ou trois moniteurs primitifs, il est nécessaire de réaliser des simulations sur des exemples de propriétés plus complexes afin de valider la méthode d'interconnexion.

FIGURE 3.24 – Résultat de simulation pour la propriété P₁₅

3.3.2 Simulation et validation de la méthode d'interconnexion

Lorsque la propriété PSL à vérifier est complexe, elle nécessite plus de moniteurs primitifs et donc les problèmes de blocages liés à la méthode d'interconnexion seront plus faciles à détecter. De nombreuses simulations ont été réalisées, seul le résultat de la simulation de la figure 3.25 correspondant à l'évaluation de A_1 est présentée :

`property  $A_1$  is assert always((ticket and valide) → next! open until!_ fin_passage);`

On voit très rapidement sur cette trace que lorsque les sorties des moniteurs synchrones et asynchrones sont "identiques", c'est-à-dire que l'on a *Valid1_async* si *Valid_sync* est actif, sinon on a *Valid0_async*. On a la même correspondance pour le signal *Pending*, on obtient *Pending1_async* si *Pending_sync* est actif, sinon on obtient *Pending0_async*. L'évaluation et le sens de la propriété A_1 ayant été présentés à plusieurs reprises dans le manuscrit, on ne commente pas les résultats d'évaluation cycle par cycle.

3.4 Conclusion

On dispose d'une méthode d'interconnexion et d'une bibliothèque de moniteurs primitifs asynchrones nous permettant de générer facilement et très rapidement des moniteurs asynchrones implémentant des propriétés PSL complexes. Les premiers tests réalisés en simulation numérique semblent valider la méthode de construction des moniteurs primitifs asynchrones et leur interconnexion. Toutefois, ces simulations sont réalisées à l'aide d'une version comportementale du synchroniseur embarqué dans les moniteurs primitifs et la validation finale ne peut se faire qu'en utilisant une version matérielle du *Synchro@clk*. Le chapitre 4 présente donc l'implémentation et la validation de la version matérielle du *Synchro@clk*.

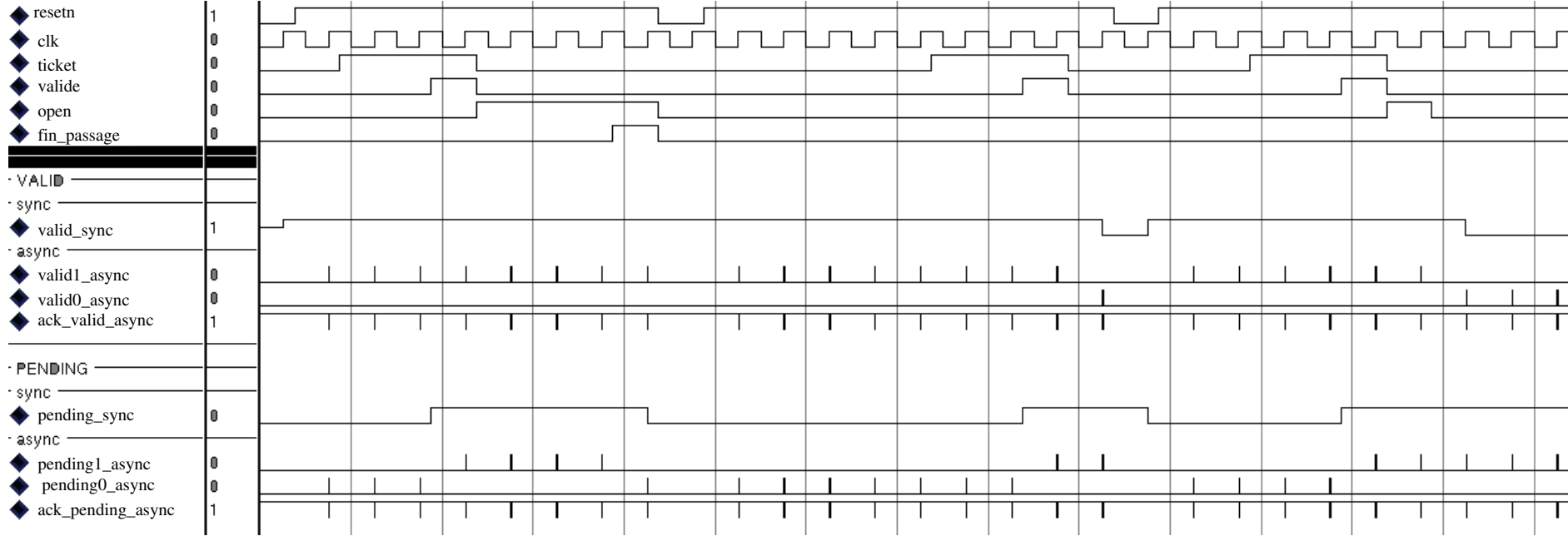


FIGURE 3.25 – Résultat de simulation pour la propriété A₁

Chapitre 4

Le synchroniseur

Sommaire

4.1	Architecture du synchroniseur	78
4.2	Validation par simulation du <i>Synchro@clk</i>	81
4.2.1	Le <i>Synchro@clk</i> seul	81
4.3	Preuve formelle du synchroniseur	82
4.3.1	L'outil RAT	83
4.3.2	Présentation de la méthode de preuve	83
4.3.3	Description d'une porte	84
4.3.4	Description de l'environnement	84
4.3.5	Ensemble des preuves réalisées	85
4.3.6	Résultats	86
4.4	Validation et caractérisation du synchroniseur	87
4.4.1	Caractérisation en Fréquence du <i>Synchro@clk</i>	87
4.4.2	Caractérisation en aire du <i>Synchro@clk</i>	89
4.5	Conclusion sur le <i>Synchro@clk</i>	90

Les moniteurs asynchrones ne peuvent être fonctionnels que s'ils embarquent un bloc de synchronisation respectant les spécifications énoncées dans le chapitre 3. La suite présente le circuit *Synchro@clk* qui permet, d'une part d'échantillonner une valeur sur front montant d'horloge et d'autre part, de transformer cette valeur échantillonnée en une donnée codée en double rails et respectant le protocole quatre phases. On présente aussi les différentes étapes de validation du synchroniseur et quelles sont les conditions à respecter pour que le *Synchro@clk*, et donc les moniteurs, soient fonctionnels.

4.1 Architecture du synchroniseur

Le *Synchro@clk* est constitué de 3 blocs différents (c.f. figure 4.1) :

1. Un bloc "d'échantillonnage", dans notre cas une simple flip-flop, réalise la synchronisation et l'échantillonnage de *Expr* sur front montant d'horloge. La donnée échantillonnée, *Expr@Clk*, est connectée à un bloc "encodage".
2. Le bloc "encodage" assure le passage d'un codage simple rail vers un codage double rails. Chaque rail en sortie du bloc "encodage" est connectée à un "générateur de jeton".
3. Les deux blocs "générateurs de jetons" (c.f. partie 2.7, p 51) assurent que la donnée de sortie respecte le protocole 4 phase poignée de main.

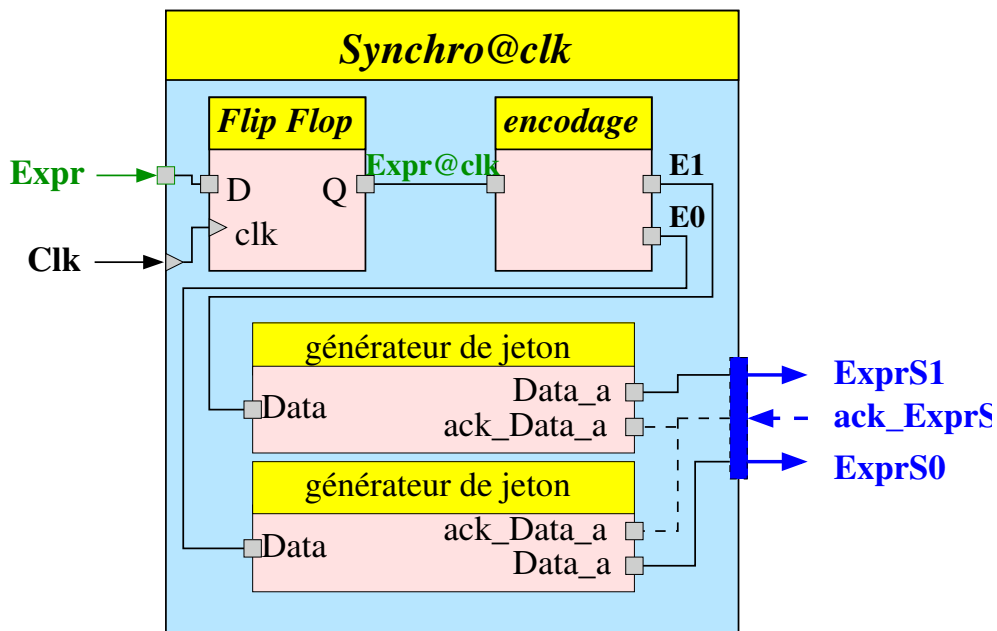


FIGURE 4.1 – Architecture du *Synchro@clk*

Le bloc "échantillonnage" L'échantillonnage sur front montant de *Clk* peut être assuré par une simple bascule de type Flip-Flop. Ainsi l'échantillonnage de *Expr* se fait au moment du front montant de *Clk* et la donnée *Expr@Clk* sera stable tout un cycle d'horloge.

Le bloc "encodage" Ce bloc doit permettre deux fonctionnalités. Son rôle premier est la transformation de la donnée $Expr@Clk$ en une donnée codée sur deux rails, $E0$ et $E1$. De plus, ce bloc doit permettre une remise à '0' des deux rails à chaque cycle. En effet, si on souhaite obtenir un jeton $ExprS$ en sortie du $Synchro@clk$ après chaque front montant de Clk , il est indispensable qu'il y ait un front montant sur $E1$ ou $E0$ à chaque cycle car le générateur de jeton ne génère de sortie que suite à un front montant sur la donnée d'entrée $data$.

Une façon très simple de procéder consiste à implémenter les deux fonctions suivantes :

$$- E1 = Expr@Clk.Clk$$

$$- E0 = \overline{Expr@Clk}.Clk$$

De cette façon, dès que $Clk=0$, alors $E1=E0=0$ et l'on observe un retour à '0' des deux rails pour chaque cycle. Cependant, l'implémentation de cette fonction n'est pas exempte d'aléas. L'aléa provient du temps de mise à jour des sorties de la bascule comme l'illustre le chronogramme de la figure 4.2. Au cycle #2, lorsque Clk passe à '1', $Expr@Clk$ est encore à '1', la sortie $E1$ passe donc à '1'. Lorsque la sortie de la bascule est mise à jour, $Expr@Clk$ passe à '0' entraînant le passage à '1' de $E0$ et le passage à '0' de $E1$. Dans ce cas on produit donc un front montant sur $E1$ et un front montant sur $E0$ ce qui entraîne l'activation des deux sorties $ExprS1$ et $ExprS0$. Dans ce cas le codage double rails n'est pas respecté.

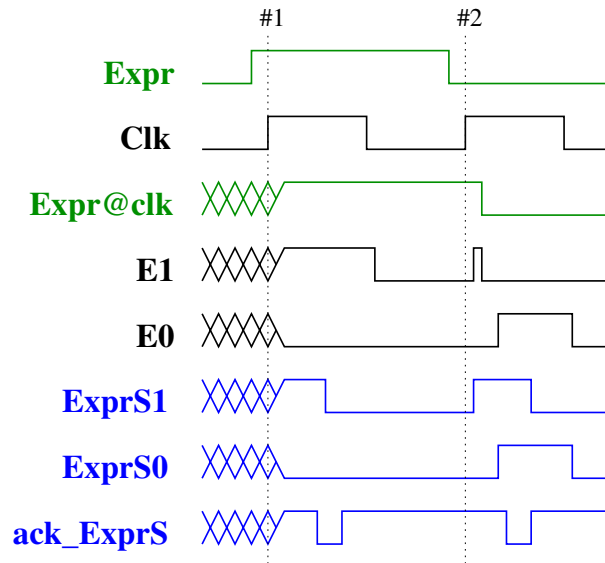


FIGURE 4.2 – Mise en évidence d'un aléa dans le bloc "codage"

On supprime les aléas en forçant $E1=E0=0$ quand $Clk=1$, et en autorisant la mise à '1' de $E1$ (respectivement $E0$) uniquement si $Clk=0$, $Expr@Clk=1$ (respectivement '0') et que $E0=0$ (respectivement $E1$). On doit donc implémenter les fonctions suivantes :

$$- E1 = Expr@Clk \& \overline{Clk} \& \overline{E0} = \overline{Expr@Clk} + Clk + E0$$

$$- E0 = \overline{Expr@Clk} \& \overline{Clk} \& \overline{E1} = \overline{Expr@Clk} + Clk + E1$$

On obtient donc l'architecture présentée figure 4.3 pour le bloc "encodage". Le chronogramme du fonctionnement du $Synchro@clk$ est présenté figure 4.4. Comme les signaux $E1$ et $E0$ ne peuvent être à '1' que lorsque Clk est à '0', une demi période de retard est

introduite entre le front montant de Clk et la disponibilité des sorties du $Synchro@clk$, au mieux après le front descendant de Clk .

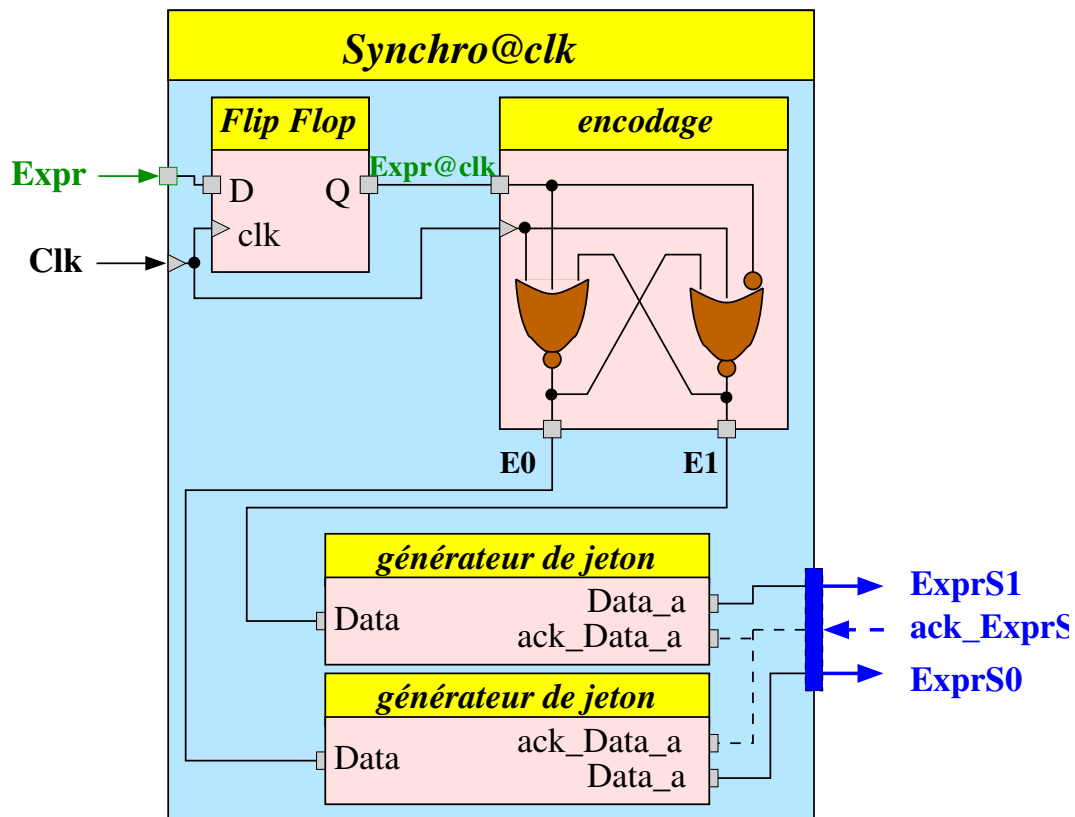


FIGURE 4.3 – Architecture du $Synchro@clk$

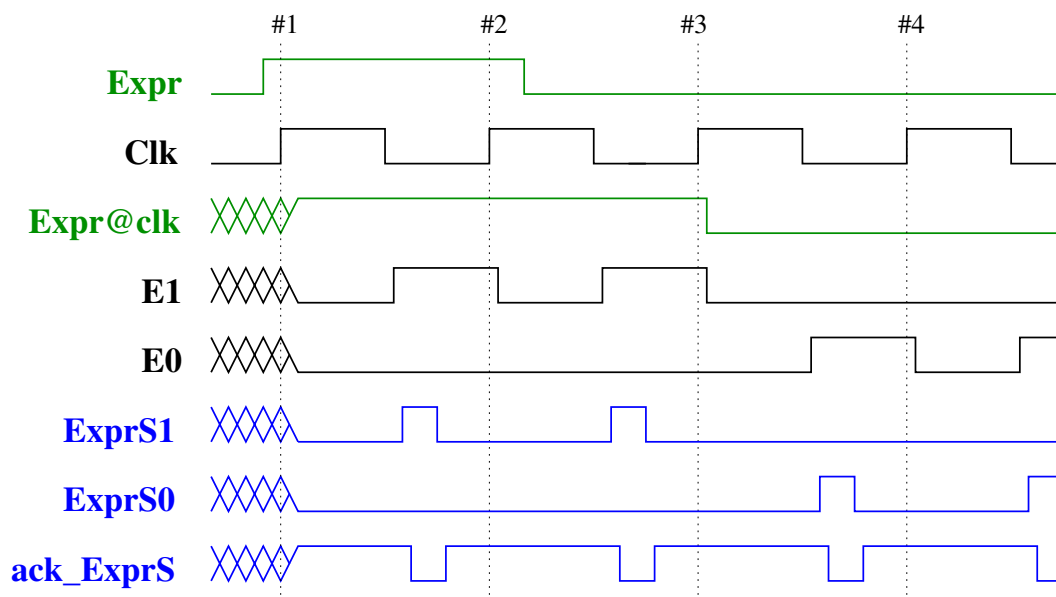


FIGURE 4.4 – Chronogramme du $Synchro@clk$

4.2 Validation par simulation du *Synchro@clk*

La validation numérique du *Synchro@clk* est réalisée grâce à l'outil MODELSIM de la société MENTOR GRAPHIC.

4.2.1 Le *Synchro@clk* seul

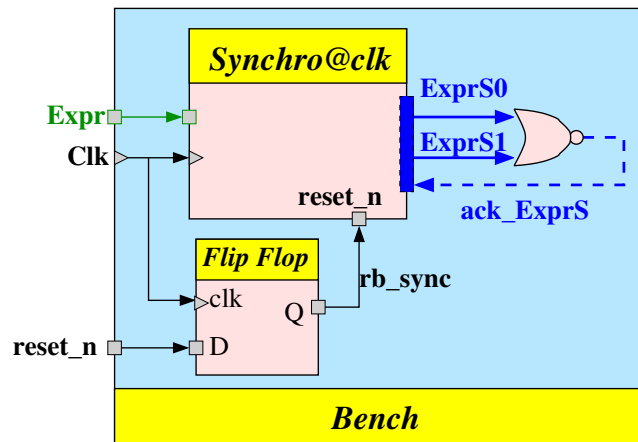


FIGURE 4.5 – Structure du banc de test

Afin de vérifier le bon fonctionnement du *Synchro@clk* il est nécessaire d'avoir un circuit capable d'assurer le respect du protocole quatre phases poignée de mains en sortie du synchroniseur. Cette fonction est réalisée par une simple porte NOR, on appelle ce circuit le "circuit bouchon" (*c.f.* figure 4.5).

On remarque la présence d'une bascule pour le signal *Reset_n* pour générer le signal de reset synchrone *rb_sync*. L'initialisation synchrone du *Synchro@clk* est indispensable pour obtenir une initialisation synchrone des moniteurs asynchrones afin de comparer les solutions synchrones et asynchrones (*c.f.* partie 3.2.2, page 58). L'initialisation synchrone du synchroniseur présente un autre avantage majeur. Cela permet de supprimer les éventuels aléas sur le signal *Reset_n* qui pourraient réinitialiser des parties du *Synchro@clk* de manière anarchique lorsque ce dernier est en fonctionnement. En effet, dans le *Synchro@clk* on utilise des portes de Müller dont l'initialisation est réalisée de façon complètement asynchrone et l'utilisation d'un signal de reset synchrone permet de l'éviter. Le résultat de la simulation du *Synchro@clk* associé au circuit bouchon est donné figure 4.6.

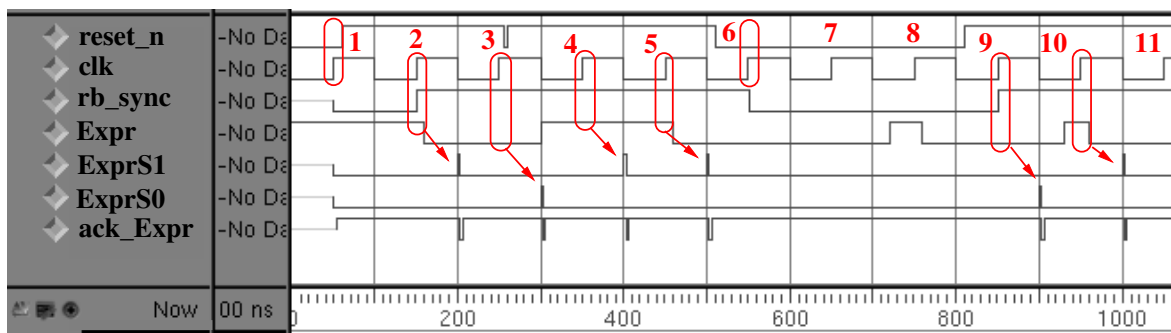


FIGURE 4.6 – Résultat de simulation du *Synchro@clk*

Le cycle correspondant au front montant #1 est un cycle d'initialisation puisque *Reset_n* est actif. Aucune sortie du *Synchro@clk* ne doit donc être activée. On voit que c'est le cas puisque le premier résultat d'évaluation (*ExprS1*) n'est présent qu'après le deuxième front montant de *Clk* et avec un demi cycle d'horloge de retard comme expliqué précédemment, page ??.

L'évaluation se passe ensuite normalement pour les cycles #3, 4, 5. On remarque que l'état bas du signal *Reset_n* entre les cycles #3 et #4 n'est pas pris en compte car il n'y a pas de front montant pour le valider.

On a une nouvelle initialisation du *Synchro@clk* lors du cycle #6 qui dure jusqu'au cycle #8 inclus durant lesquels aucune sortie n'est produite conformément à nos attentes. Ensuite l'évaluation reprend correctement pour les cycles #9 et 10.

Le fonctionnement du *Synchro@clk* est donc conforme aux spécifications décrites dans la section 3.2.1, figure 3.7.

4.2.1.1 Le *Synchro@clk* dans un moniteur

Afin de vérifier que le *Synchro@clk* s'intègre correctement dans les moniteurs asynchrones, on crée un bench pour la propriété P_{16} :

Property P_{16} is assert always(*A*) ;

Cette propriété, très simple, nous permet de vérifier qu'il n'y a pas de problème lorsque l'on connecte le *Synchro@clk* dans les moniteurs primitifs. Ici encore, les signaux *Valid1* et *Valid0* permettent de générer le signal *Ack_Valid* grâce à un circuit bouchon.

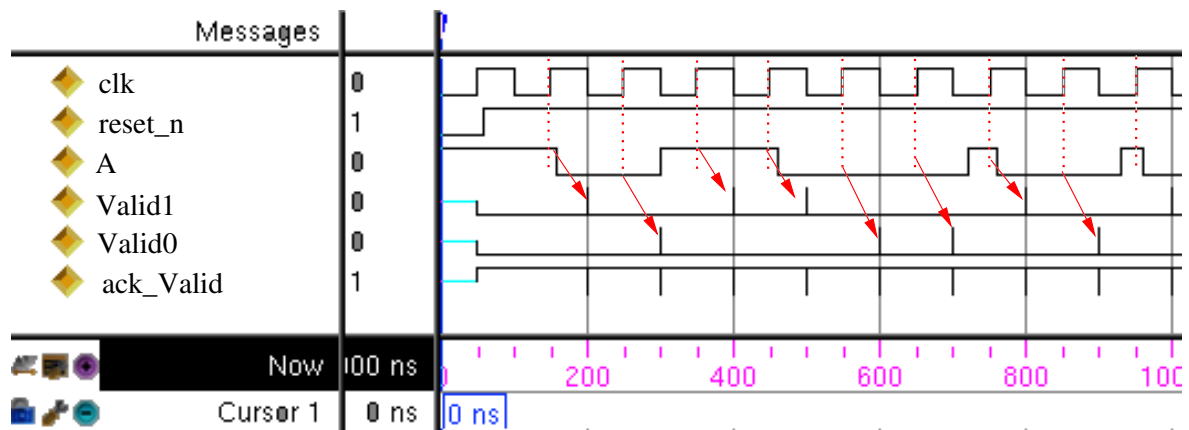


FIGURE 4.7 – Résultat de simulation de la propriété P_{16}

À l'aide de MODELSIM on obtient les résultats de la figure 4.7. On constate qu'à chaque front montant où *A* est vrai, on obtient le résultat *Valid1* un demi cycle d'horloge plus tard. On a une évaluation correcte de P_{16} indiquant que les *Synchro@clk* dans les moniteurs primitifs de ce moniteur asynchrone jouent parfaitement leur rôle.

4.3 Preuve formelle du synchroniseur

Afin de s'assurer du bon fonctionnement du *Synchro@clk*, on utilise l'outil **RAT** [BCP⁺07] développé à l'Université de Graz associé à la méthode de preuve formelle de circuits asynchrones développée par K. ALSAYEG dans [AMAF09].

4.3.1 L'outil RAT

Initialement, l'outil RAT (Requierevements Analysis Tool) permet de détecter des erreurs dans un ensemble d'obligations spécifiant le comportement d'un système grâce à la méthodologie suivante :

- Soit un ensemble R d'obligations écrites en PSL décrivant le comportement d'un système.
- Soit un ensemble A d'assertions écrites en PSL que le système doit respecter.
- Soit un ensemble P de comportements possibles décrits en PSL que le système peut avoir.
- L'outil vérifie ensuite que l'ensemble R des obligations implique qu'aucune assertion de l'ensemble A n'est violée.
- L'outil vérifie aussi que l'ensemble R n'est pas trop stricte et n'exclut pas les comportements décrits dans P .

Cet outil permet de manipuler des propriétés PSL que l'on peut classer en deux groupes : le groupe (**R**) contenant les propriétés qui décrivent un comportement imposé, par l'environnement par exemple, et le groupe (**A**) regroupant les propriétés décrivant des comportements attendus du circuit. Cet outil permet ensuite de vérifier que ces deux ensembles sont cohérents.

4.3.2 Présentation de la méthode de preuve

La méthode de preuve s'appuie sur la méthode développée au laboratoire TIMA. À l'origine, la méthode présentée était utilisée pour prouver la correction de petits contrôleurs de protocole asynchrones. Cette méthode de preuve des circuits asynchrones peut être résumée de la manière suivante :

- L'ensemble des signaux utiles à la description du système est défini.
- Chaque porte du circuit est décrite grâce à un ensemble de propriété PSL.
- L'ensemble de propriétés décrivant les portes du circuit à prouver est inclus dans l'ensemble R .
- Le comportement de l'environnement est décrit en PSL et inclus dans le même ensemble R .
- L'état initial du système est décrit grâce à une propriété PSL inclus dans R .
- L'ensemble des comportements à prouver du circuit (respect du protocole quatre phases, pas de deadlock, pas d'aléa...) est décrit à l'aide de propriété PSL inclus dans l'ensemble A .
- On vérifie la cohérence entre A et R .
- On vérifie que l'outil ne trouve pas de contre exemple à une des propriétés de l'ensemble A .

La preuve est effectuée par Model Checking Symbolique Borné utilisant un solveur SAT. Quand l'outil trouve une trace d'exécution violant une des propriétés de A , la trace est affichée permettant une analyse du mauvais comportement du circuit. Cette méthode est particulièrement adaptée aux circuits asynchrones car les propriétés PSL permettent de décrire efficacement les protocoles asynchrones en raisonnant sur des événements et non sur des temps. Cela permet de bien capturer la possibilité de délais très longs dans les portes logiques correspondant au fonctionnement des circuits quasi insensibles aux délais. Les connexions sont considérées sans délai et toutes les fourches sont considérées comme isochrones.

4.3.3 Description d'une porte

Un travail préliminaire à la preuve est la description des portes grâce à des propriétés PSL. De nombreuses portes ont déjà été décrites par K. ALSAYEG et nous allons illustrer la méthode sur une porte de Müller à deux entrées (A et B) et une sortie Z.

On appelle Set Expression (SE) la combinaison d'entrées provoquant le passage à '1' de la sortie et Reset Expression (RE) la combinaison d'entrée provoquant le passage à '0' de la sortie. Pour une porte de Müller on a un basculement de la sortie à '1' si les deux entrées sont à '1', et le basculement de la sortie à '0' a lieu lorsque les deux entrées sont à '0'. On a donc $\mathbf{SE} = A.B$ et $\mathbf{RE} = !A. !B$.

Tout d'abord une propriété *prop0* est utilisée pour décrire le maintien à '0' de la sortie, respectivement une propriété *prop1* permet de décrire le maintien à '1' de la sortie. Ces propriétés de maintien de la sortie permettent de modéliser le fait que les portes utilisées dans un circuit QDI ne créent pas d'aléas sur leurs sorties.

Property *prop0* is **assert** always($!Z \rightarrow !Z$ until_ SE);
la sortie Z de la porte vaut '0' et reste à '0' tant que les entrées de la porte ne permettent pas le basculement à '1' de la sortie
Property *prop1* is **assert** always($Z \rightarrow Z$ until_ RE);
la sortie Z de la porte vaut '1' et reste à '1' tant que les entrées de la porte ne permettent pas le basculement à '0' de la sortie

Deux propriétés supplémentaires, *prop1to0* et *prop0to1* permettent de décrire quand la sortie doit basculer de '1' à '0' respectivement de '0' à '1'.

Property *prop1to0* is **assert** always($(RE.Z) \rightarrow$ eventually! $!Z$);
la sortie Z de la porte vaut '1' et les entrées de la porte sont telles que le prochain évènement sur la sortie sera le passage à '0' de cette dernière
Property *prop0to1* is **assert** always($(SE. !Z) \rightarrow$ eventually! Z);
la sortie Z de la porte vaut '0' et les entrées de la porte sont telles que le prochain évènement sur la sortie sera le passage à '1' de cette dernière

En appliquant ces méthodes de descriptions des portes à l'ensemble des portes dont on a besoin, on est en mesure de décrire complètement le *Synchro@clk* à l'aide de propriétés PSL en suivant la méthode présentée précédemment. La modélisation complète du circuit en PSL et plus particulièrement la modélisation de la bascule en PSL sont données dans l'annexe B.2.

Il faut maintenant spécifier les contraintes sur les signaux, notamment l'initialisation de ces derniers, et les contraintes sur l'environnement. On pourra ainsi vérifier que la description PSL du synchroniseur est cohérente avec les contraintes de l'environnement.

4.3.4 Description de l'environnement

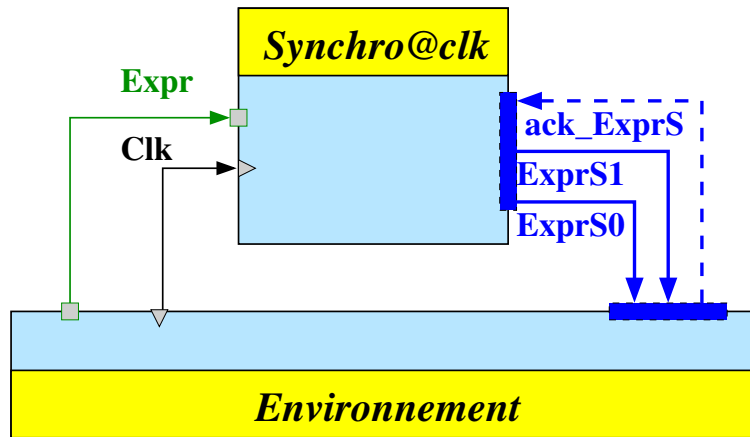
On a le système *Synchro@clk* + *environnement* présenté figure 4.8. La façon dont le signal *ack_ExprS* est généré par l'environnement doit respecter le protocole quatre phases poignée de mains. C'est-à-dire qu'à chaque fois qu'une donnée *ExprS1* ou *ExprS0* est émise, le signal *Ack_ExprS* passe à '0' afin d'invalider la donnée et doit rester dans l'état bas jusqu'à ce que la donnée soit effectivement invalidée. Puis *Ack_ExprS* doit repasser à '1' indiquant qu'une nouvelle donnée est attendue. Il doit rester dans cet état jusqu'à l'émission d'une nouvelle donnée. Cela est exprimé grâce à la propriété suivante :

Property *env1* is assert

```

always( (ExprS1+ExprS0).Ack_ExprS → next_event!(Ack_ExprS)(Ack_ExprS until_
    (!ExprS1.!ExprS0))
    &&
    always((!ExprS1.!ExprS0).!Ack_ExprS → next_event!(Ack_ExprS)(Ack_ExprS
    until_(ExprS1+ExprS0))
    &&
    always( Ack_ExprS → Ack_ExprS until_ (ExprS1+ExprS0))
    &&
    always( !Ack_ExprS → !Ack_ExprS until_ (!ExprS1.!ExprS0) );

```

FIGURE 4.8 – Ensemble ”*Synchro@clk* + environnement”

L’horloge est un signal provenant de l’environnement du *Synchro@clk*. Il est indispensable de contraindre l’horloge car si cette dernière reste bloquée dans un état il est impossible de vérifier le bon fonctionnement du *Synchro@clk*. Les propriétés précédentes imposent des changements d’état sur *Clk* :

Property *env2* is assert

```

always(!clk → eventually! clk)
    &&
    always( clk → eventually! !clk);

```

4.3.5 Ensemble des preuves réalisées

L’ensemble des preuves réalisées porte essentiellement sur les sorties du *Synchro@clk* et plus particulièrement sur les points suivants :

Correction du protocole quatre phases poignée de main en sortie Vérifier la correction du protocole de sortie revient à vérifier que lorsqu’une sortie (*ExprS1* ou *ExprS0*) est active, elle reste active au moins jusqu’à ce que le signal d’acquiescement *Ack_ExprS* ne passe à ’0’. Pour finir, lorsque les deux sorties sont repassées à ’0’, il reste à vérifier qu’elles ne seront pas actives de nouveau avant que *Ack_ExprS* ne soit repassé à ’1’. On contrôle que le circuit mette ses sorties correctement à jour en fonction de la valeur de *Ack_ExprS*. La mise à jour de *Ack_ExprS* en fonction de la valeur des sorties *ExprS1* et

$ExprS0$ est contraint par la propriété $env1$. La vérification à réaliser peut s'exprimer grâce à une propriété PSL de la manière suivante :

Property proof1 is assert

$$\begin{aligned} & \text{always}((ExprS1 + ExprS0) \rightarrow (ExprS1 + ExprS0) \text{ until_ } (!Ack_ExprS)) \\ & \quad \& \\ & \text{always}((!ExprS1. !ExprS0) \rightarrow (!ExprS1. !ExprS0) \text{ until_ } (Ack_ExprS)) \end{aligned}$$

Pas de deadlock en sortie On veut s'assurer que lorsque les moniteurs fonctionnent correctement, les sorties du *Synchro@clk* ne restent pas bloquées dans un état. Le bon fonctionnement des moniteurs primitifs est exprimé ici encore par la propriété $env1$. Vérifier l'absence de "deadlock" sur la sortie revient à prouver que lorsqu'une sortie $ExprS1$ ou $ExprS0$ est active, si l'acquiescement $Ack_ExprS = '0'$, alors la sortie $ExprS1$ ou $ExprS0$ sera mise à '0'. De façon symétrique, lorsque les sorties sont inactives et que l'acquiescement est actif, nécessairement l'une des sorties $ExprS1$ ou $ExprS0$ sera activée. L'absence de "deadlock" s'exprime donc en PSL de la façon suivante :

Property proof2 is assert

$$\begin{aligned} & \text{always}((ExprS1 + ExprS0). !Ack_ExprS \rightarrow \text{eventually}(!ExprS1. !ExprS0)) \\ & \quad \& \\ & \text{always}((!ExprS1. !ExprS0). Ack_ExprS \rightarrow \text{eventually}!(ExprS1 + ExprS0)); \end{aligned}$$

Codage double rails en sortie Le principal problème rencontré lors de l'élaboration du *Synchro@clk* était la possibilité d'avoir pour un même cycle d'évaluation les sorties $ExprS1$ et $ExprS0$ actives. Si le cas se présentait, le codage double rails des sorties ne serait plus respecté. Il faut donc vérifier que $ExprS1$ et $ExprS0$ ne sont jamais actives en même temps :

Property proof3 is assert

$$\text{never}(ExprS1. ExprS0);$$

4.3.6 Résultats

La preuve des propriétés précédentes (absence de "deadlock", respect du codage, respect du protocole) n'est possible que si l'on respecte la condition suivante : *lorsque l'on a un instant d'échantillonnage (front montant de Clk), il faut qu'une des sorties du Synchro@clk ait été active puis soit repassée à '0' grâce au signal d'acquiescement avant que l'on ait un nouvel instant d'échantillonnage*. Autrement exprimé, il faut que les quatre phases du protocole soient résolues avant un nouvel échantillonnage. On peut l'exprimer en PSL de la manière suivante :

Property env3 is assert

$$\begin{aligned} & \text{always}((Clk.\text{prev}(!Clk)) \rightarrow \text{next}!((!ExprS1. !ExprS0. \text{prev}(ExprS1 + ExprS0)) \text{ before!} \\ & \quad (Clk.\text{prev}(!Clk))))); \end{aligned}$$

On a donc une contrainte temporelle forte conditionnant le fonctionnement du *Synchro@clk*. Cette contrainte provient du domaine synchrone et est propagée à la réunion des domaines synchrones et asynchrones représentée en matériel par le *Synchro@clk*.

En simulation numérique on avait un fonctionnement du *Synchro@clk* qui paraissait conforme aux spécifications énoncées dans le chapitre 3, page 58. La preuve permet de mettre en évidence que le fonctionnement du *Synchro@clk* n'est correcte que si une hypothèse temporelle *env3* sur l'acquiescement des sorties du synchroniseur est respectée. On va maintenant étudier dans quelle plage de fonctionnement le *Synchro@clk* respecte cette hypothèse.

4.4 Validation et caractérisation du synchroniseur

Le *Synchro@clk* est fonctionnel uniquement si l'environnement du synchroniseur, c'est-à-dire le moniteur primitif, respecte effectivement le protocole quatre phases poignée de main. Des simulations analogiques du synchroniseur seul puis intégré dans un moniteur primitif sont donc importantes pour valider le fonctionnement du *Synchro@clk*. De plus, afin d'avoir une première estimation de la fréquence maximale admissible pour les moniteurs asynchrones, une caractérisation en fréquence du *Synchro@clk* est réalisée.

4.4.1 Caractérisation en Fréquence du *Synchro@clk*

L'utilisation correcte de la technologie QDI pour implémenter les moniteurs asynchrones doit amener une plus grande robustesse grâce notamment à l'insensibilité aux délais. Cependant, avant de bénéficier des avantages d'une telle implémentation, il faut s'assurer que le *Synchro@clk* réalise correctement son rôle. Le *Synchro@clk* a besoin de temps pour échantillonner sur front montant de *Clk* et pour réaliser l'adaptation de protocole. Le temps nécessaire pour générer une sortie et compléter les quatre phases du protocole lorsque la sortie est acquiescée immédiatement donne une indication sur la fréquence d'horloge maximale admissible par le *Synchro@clk*.

Environnement de simulation Le but des moniteurs asynchrones est la robustesse et la correction du résultat d'évaluation lorsque le moniteur est placé dans un environnement hostile, notamment lorsque la tension d'alimentation du circuit baisse. On doit donc caractériser le *Synchro@clk* en fréquence et en tension, c'est-à-dire connaître la fréquence maximale admissible en fonction de la tension d'alimentation *vdd*. Cependant, le synchroniseur étant un système mixe synchrone et asynchrone, les outils de conception synchrone permettant de faire de l'analyse de timing vont être inefficaces à moins d'ouvrir toutes boucles combinatoires présentes dans le *Synchro@clk*, boucles provenant des chemins d'acquiescements. Afin de s'affranchir de cette étape, on préfère réaliser cette mesure en utilisant l'outil de simulation analogique de la société CADENCE. On considérera que pour une tension donnée, la fréquence maximale $F_{MAX}(VDD)$ du *Synchro@clk* est obtenue si pour toutes fréquences supérieures à $F_{MAX}(VDD)$, il existe au moins un cycle d'évaluation où le temps d'acquiescement des sorties *ExprS1* et *ExprS0* est suffisamment long pour qu'une nouvelle évaluation ait lieu avant la résolution des quatre phases du protocole poignée de main. Dans ce cas, la propriété *env3* est violée et le bon fonctionnement du *Synchro@clk* n'est plus assuré (c.f. partie 4.3.6).

L'importation du *Synchro@clk* dans l'environnement de simulation CADENCE nécessite la netlist du synchroniseur. Cette dernière est obtenue "manuellement" en deux étapes : d'abord en effectuant traduction de la description structurelle VHDL vers Verilog puis en remplaçant le nom de chacune des portes par celui de la porte correspondante dans les

bibliothèques technologiques ciblées¹. La vue ainsi importée est une vue de type "schematic". Cela signifie que les résultats suivants sont un peu surévalués. En effet dans une vue schématique, les fils ne présentent pas un comportement physique réel. Pour avoir une meilleure caractérisation du *Synchro@clk*, une simulation post-placement routage aurait été requise. Toutefois la bibliothèque TAL présente quelques erreurs qui empêchent de réaliser une simulation post placement routage.

Le banc de caractérisation du *Synchro@clk* sous Cadence Analog Environment est donné figure 4.9. Afin d'assurer le respect du protocole de sortie du *Synchro@clk*, un circuit bouchon est ajouté (porte NOR). Les trois signaux *Reset_n*, *Clk* et *Expr* sont générés par des sources de tension. Ces sources de tension présentent des fronts de montée/descente identiques quelque soit la valeur de la tension d'alimentation. Des buffers sont ajoutés entre les sources de tension et le *Synchro@clk* pour avoir une déformation des signaux analogiques plus proche de la réalité. Une Flip-Flop permet de générer le signal *rb_sync* (c.f. page 81).

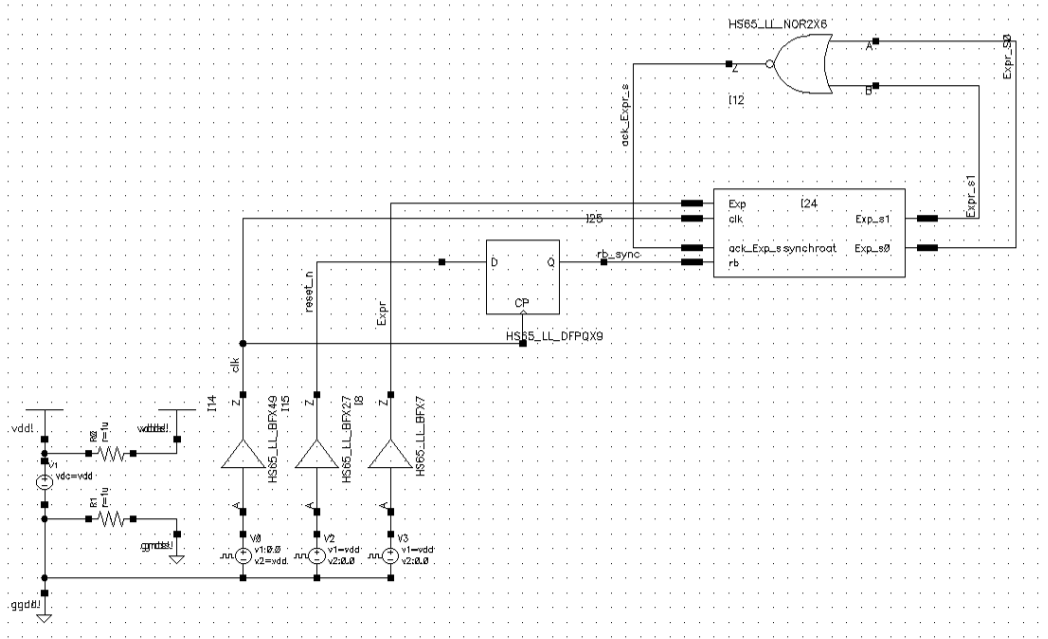
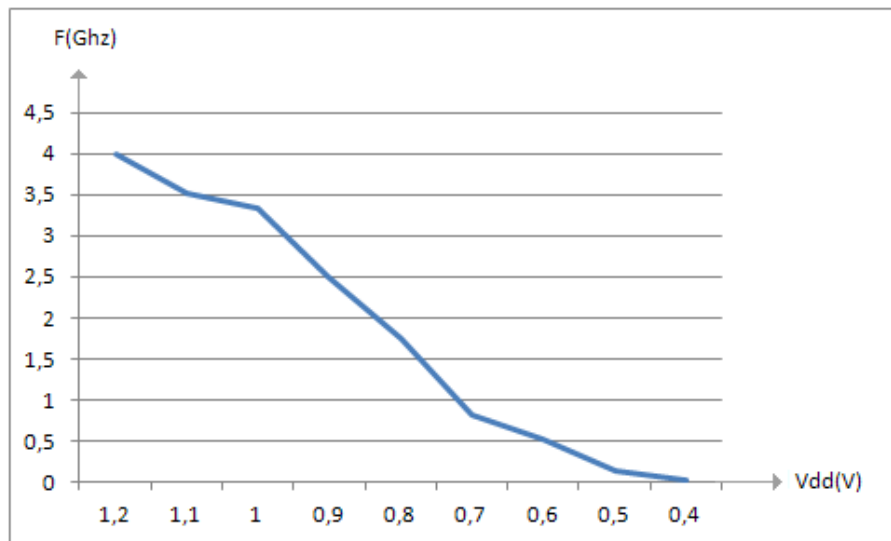


FIGURE 4.9 – Banc de caractérisation du *Synchro@clk*

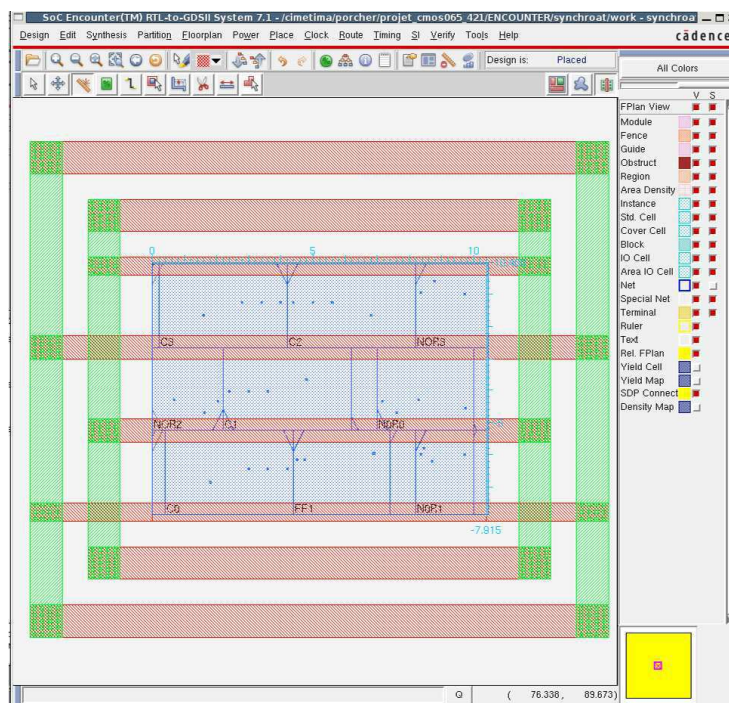
Caractérisation de la Fréquence maximale en fonction de la tension d'alimentation On réalise plusieurs simulations en faisant varier la tension d'alimentation (Vdd) par pas de 0,5V. Pour chaque tension d'alimentation, on cherche la fréquence maximale du *Synchro@clk*. On obtient ainsi la courbe de la figure 4.10. On voit qu'à tension nominale, le *Synchro@clk* est capable d'échantillonner à une fréquence importante mais que cette fréquence diminue ensuite quasi linéairement avec la tension d'alimentation.

1. CORE 65nm Low Power Low VT de ST Microelectronics et Tima Asynchronous Library (TAL) 65nm

FIGURE 4.10 – Fréquence maximale du *Synchro@clk* en fonction de Vdd

4.4.2 Caractérisation en aire du *Synchro@clk*

Afin de finir de caractériser le *Synchro@clk*, une estimation de la surface de ce dernier est requise. Pour réaliser l'opération de placement routage, on utilise l'outil SOC ENCOUNTER et on obtient le circuit de la figure 4.11 composé de 9 portes standard et de 6 Fillers.

FIGURE 4.11 – Le *Synchro@clk* placé et routé

Dans notre cas, le but est simplement de positionner correctement les portes standards et les "Fillers" (cellules de remplissage) afin d'obtenir une approximation de la surface. La connexion à des plots d'entrées/sorties n'est pas utile car on ne s'intéresse qu'à la surface du *Synchro@clk* proprement dite. L'aire du *Synchro@clk* obtenue après placement routage est de $79,8 \mu m^2$, hors anneaux d'alimentation. L'aire de ce composant est primordiale et doit être aussi faible que possible car le *Synchro@clk* est présent en un ou deux exemplaires dans tous les moniteurs primitifs asynchrones. Dans notre cas, l'aire de la bascule représente uniquement un dixième de l'aire totale du *Synchro@clk*. Il y a un facteur 10 entre l'aire d'une bascule assurant la fonction de synchronisation dans les moniteurs synchrones et le *Synchro@clk* qui assure cette même fonctionnalité dans les moniteurs asynchrones. En conséquence, nos moniteurs asynchrones nécessiteront plus de surface que leurs équivalents synchrones.

4.5 Conclusion sur le *Synchro@clk*

À partir des contraintes liées à l'utilisation de la technologie asynchrone et plus précisément l'utilisation des implémentations QDI, des spécifications ont été exprimées dans les chapitres précédents pour le bloc de synchronisation des moniteurs asynchrones avec le circuit synchrone en cours de vérification. On a construit le *Synchro@clk* en respectant ces spécifications avec un détail supplémentaire, l'introduction d'un demi-cycle d'horloge de retard entre l'instant d'évaluation (front montant d'horloge) et le moment où le résultat apparaît en sortie du moniteur.

Les simulations numériques ont permis de valider la bonne intégration du *Synchro@clk* dans les moniteurs élémentaires asynchrones et plus généralement dans les moniteurs asynchrones. La preuve réalisée sur le *Synchro@clk* permet d'assurer le bon fonctionnement du synchroniseur sous réserve du respect d'une hypothèse temporelle : les sorties du *Synchro@clk* doivent être acquittées avant qu'une nouvelle évaluation ne soit demandée. Dans le cas contraire, il est impossible d'assurer le bon fonctionnement de nos moniteurs car si un moniteur élémentaire manque un cycle d'évaluation, alors l'évaluation de la propriété PSL en cours n'a plus de sens.

Des simulations analogiques ont permis de déterminer une fréquence maximale de fonctionnement pour le *Synchro@clk* en fonction de la tension d'alimentation. Les fréquences obtenues étant assez élevées, on peut espérer que nos moniteurs possèdent eux aussi des fréquences de fonctionnement élevées. Cependant, le respect de l'hypothèse temporelle précédemment évoquée représente une contrainte très forte pour les moniteurs asynchrones. En effet, pour assurer l'aspect QDI d'un circuit asynchrone il est indispensable de ne pas avoir d'hypothèse temporelle sur les délais. On voit que l'hypothèse temporelle du domaine synchrone contraint le fonctionnement du *Synchro@clk*, c'est-à-dire l'intersection des domaines synchrones et asynchrones.

Un moniteur asynchrone sera effectivement plus robuste que son équivalent synchrone, à une tension et une fréquence données, si l'hypothèse temporelle contraignant le fonctionnement du *Synchro@clk* n'est pas violée et que le moniteur synchrone présente un comportement fautif. Afin de vérifier le dernier point, les études au niveau numérique réalisées sur les moniteurs asynchrones doivent être étendues à des niveaux d'abstraction plus faible. Le prochain chapitre porte donc sur l'étude de la robustesse des moniteurs asynchrones et plus précisément sur le comportement des moniteurs asynchrones au niveau analogique.

Étude de la robustesse

Sommaire

5.1	Caractérisation en surface des moniteurs asynchrones	92
5.1.1	Caractérisation en surface des moniteurs primitifs	92
5.1.2	Croissance linéaire de l'aire en fonction de la complexité de la propriété PSL	94
5.2	Comparaison de la robustesse des solutions synchrones et asynchrones	95
5.2.1	Environnement de simulation et résultats	95
5.2.2	Explication des résultats	98
5.3	Implémentation d'une solution : le <i>Clk</i> stretching	101
5.3.1	Principe	101
5.3.2	Définition d'un nouveau synchroniseur	103
5.3.3	Contrôle de l'horloge : le <i>Clk_manager</i>	107
5.4	Validation de l'implémentation du "clock stretching"	109
5.4.1	Validation numérique	109
5.4.2	Preuve formelle	112
5.4.3	Validation analogique	113
5.4.4	Caractérisation de la solution	115
5.5	Conclusion sur la robustesse des moniteurs asynchrones . . .	118

Les précédents résultats ont permis de valider le comportement de nos moniteurs asynchrones au niveau numérique. À ce niveau de description, ils sont aussi fonctionnels que les moniteurs synchrones. Cependant, le but de ces travaux étant la conception de moniteurs robustes, une comparaison entre les moniteurs synchrones et asynchrones est requise au niveau analogique. Les bibliothèques utilisées ont été réalisées en technologie CMOS 65nm Low Power Low VT.

5.1 Caractérisation en surface des moniteurs asynchrones

Une première étape dans la caractérisation des moniteurs asynchrones est l'étude de l'aire de ces derniers. En effet, l'utilisation d'une méthode de synthèse des moniteurs basée sur la syntaxe de la propriété PSL doit permettre d'obtenir une croissance de l'aire linéaire avec la complexité de la propriété. Cet avantage est hérité de la méthode de génération des moniteurs synchrones implémentée dans la plateforme *Horus* et permet l'obtention de circuit de petite taille pour nos moniteurs. C'est très important car les moniteurs sont prévus pour être embarqués sur la même puce que le circuit monitoré. De plus, cela nous permettra de comparer l'aire des moniteurs asynchrones et synchrones. Afin de réaliser cette étude, on utilise SOC ENCOUNTER comme on l'a fait pour caractériser l'aire du *Synchro@clk*.

5.1.1 Caractérisation en surface des moniteurs primitifs

La plupart des moniteurs primitifs et des blocs élémentaires utilisés dans l'implémentation de la bibliothèque de moniteurs primitifs ont été placés routés. Le nom de ces blocs ainsi que ce qu'ils implémentent est résumé dans le tableau A.1 de l'annexe A. Le tableau 5.1 donne les résultats de placement routage pour les moniteurs primitifs synchrones et asynchrones. Pour les opérateurs `next_event*`, la comparaison de l'aire d'un étage n'a pas de réel sens car l'aire des moniteurs asynchrones de ces opérateurs est directement proportionnel à l'aire de l'étage de base alors qu'en synchrone il y a l'aire de l'étage de base puis l'aire d'un registre à décalage. Par exemple pour l'opérateur `next_event_e[2..3]`, en synchrone on a une aire proportionnelle à $X + 3Y$ où X est l'aire de l'étage de base et Y celle d'un étage du registre à décalage, en asynchrone on a une proportionnelle à $3.X$. Dans le tableau, pour le cas asynchrone on donne aussi l'aire de certains blocs élémentaires utilisés pour construire des moniteurs primitifs. Ce tableau fournit aussi le nombre de portes utilisées pour implémenter chaque moniteur primitif. Enfin, dans le cas des moniteurs synchrones il est parfois difficile d'isoler un étage pour les moniteurs implémentant des opérateurs dont la synthèse est conditionnée par un paramètre, par exemple K pour le `next_event!(b)[K]`. C'est pour cette raison que certains moniteurs primitifs synchrones n'ont pas été placés routés.

On remarque immédiatement que le nombre de portes utilisées pour les moniteurs asynchrones est bien plus important que le nombre de portes utilisées en synchrone. Ceci est justifié car pour le moniteur synchrone de l'opérateur $\rightarrow$, une simple porte suffit alors qu'en asynchrone il est nécessaire de mettre un synchroniseur et d'implémenter le protocole quatre phases poignée de main. À titre de comparaison, il faut un *Half Buffer*, 1 *Synchro@clk*, le bloc logique QDI) et une porte pour le rendez-vous des signaux d'acquitte-

Bloc matériel	Aire (μm^2)	nombre de portes
<i>Synchro@clk</i>	79,8	9
G_init	116,8	17
M_signal	127,7	19
M_always	220,8	32
M_never	242,7	35
M_eventually!	271	39
M_imply	127,7	17
M_until	239,2	34
M_until!	267	38
M_until_	242,7	35
M_until!_	270,2	39
M_before	345	48
M_before_	345	48
M_before!	372,7	50
M_before!_	372,7	50
M_next	156	22
M_next!	206,3	30
M_next_a	268,9	38
M_next_a!	318,4	46
M_next_e	285,5	40
M_next_e!	313,2	44
M_next_event(b)	242,7	35
M_next_event!(b)	270,4	39
M_next_eventK	296,7	57
M_next_event!K	538,5	77
M_next_event_a	487,6	63
M_next_event_a!	618,2	79
M_next_event_e	528,1	75
M_next_event_e!	757,5	107

Moniteurs primitifs asynchrones

Bloc matériel	Aire (μm^2)	nombre de portes
G_init	11,5	3
M_signal	12,8	2
M_always	18,6	3
M_never	27,7	4
M_eventually!	21,4	4
M_imply	3	1
M_until	23,7	5
M_until!	38,8	7
M_until_	21,4	4
M_until!_	36,4	6
M_before	78,6	18
M_before_	80,9	19
M_before!	95,9	19
M_before!_	96,5	21
M_next	11,5	2
M_next!	15,1	3
M_next_a	16,8	4
M_next_a!	16,8	4
M_next_e!	53,2	11

Moniteurs primitifs synchrones

Table 5.1 – Caractérisation en aire des moniteurs primitifs

ment soit $3 + 9 + 4 + 1 = 17$ portes pour la version asynchrone du $\rightarrow$. On a globalement un facteur 10 entre l'aire d'un moniteur primitif synchrone et de son équivalent asynchrone. Ce facteur est raisonnable et le surplus d'aire entre synchrone et asynchrone est lié à l'implémentation des protocoles asynchrones.

Le but des moniteurs asynchrones étant d'avoir une solution plus robuste que la solution synchrone, on peut considérer que le surplus d'aire n'est pas un problème si l'on atteint effectivement l'objectif de robustesse.

5.1.2 Croissance linéaire de l'aire en fonction de la complexité de la propriété PSL

Afin de vérifier que l'aire de nos moniteurs n'explose pas avec la complexité des propriétés PSL à implémenter, on réalise l'opération de placement-routage sur un ensemble de moniteurs asynchrones correspondant aux propriétés du tableau 5.2 dans lequel toutes les propriétés sont de type "assert".

Propriété PSL version faible	Propriété PSL version forte	Nombre d'étages
A		2
always A before B	always A before! B	3
always A $\rightarrow$ B before C	always A $\rightarrow$ B before! C	4
always A $\rightarrow$ next B before C	always A $\rightarrow$ next! B before! C	5
always A $\rightarrow$ next[5] B before C	always A $\rightarrow$ next![5] B before! C	9
always A $\rightarrow$ next[10] B before C	always A $\rightarrow$ next![10] B before! C	14
always A $\rightarrow$ next[20] B before C	always A $\rightarrow$ next![20] B before! C	24
always A $\rightarrow$ next[50] B before C	always A $\rightarrow$ next![50] B before! C	54
always A $\rightarrow$ next[100] B before C	always A $\rightarrow$ next![100] B before! C	104
always A $\rightarrow$ next[150] B before C	always A $\rightarrow$ next![150] B before! C	154
always A $\rightarrow$ next[200] B before C	always A $\rightarrow$ next![200] B before! C	204

Table 5.2 – Ensemble des propriétés PSL placées routées

Lorsque cela est possible, on réalise aussi l'opération de placement-routage sur la version forte de la propriété. On pourra ainsi voir le surplus d'aire uniquement due à la génération du signal *Pending*. Enfin, on indique aussi le nombre d'étages qui vont être nécessaires pour implémenter le moniteur asynchrone correspondant à la propriété PSL car le nombre d'étages dans un moniteur et la complexité de la propriété PSL implémentée sont fortement corrélés. Par exemple, pour implémenter la propriété :

assert always A $\rightarrow$ next[5] B before C

il faut un bloc `G_init` suivi d'un moniteur primitif `M_always` dont les sorties sont connectées à un moniteur primitif `M_imply`. On connecte ensuite un `M_next[5]` composé de 5 étages (5 `M_next`) et, enfin, un moniteur primitif `M_before`. Le moniteur asynchrone correspondant à cette propriété sera donc composé de 9 étages.

L'aire de chacun des moniteurs asynchrones correspondant aux propriétés du tableau 5.2 est présentée dans le graphique de la figure 5.1.

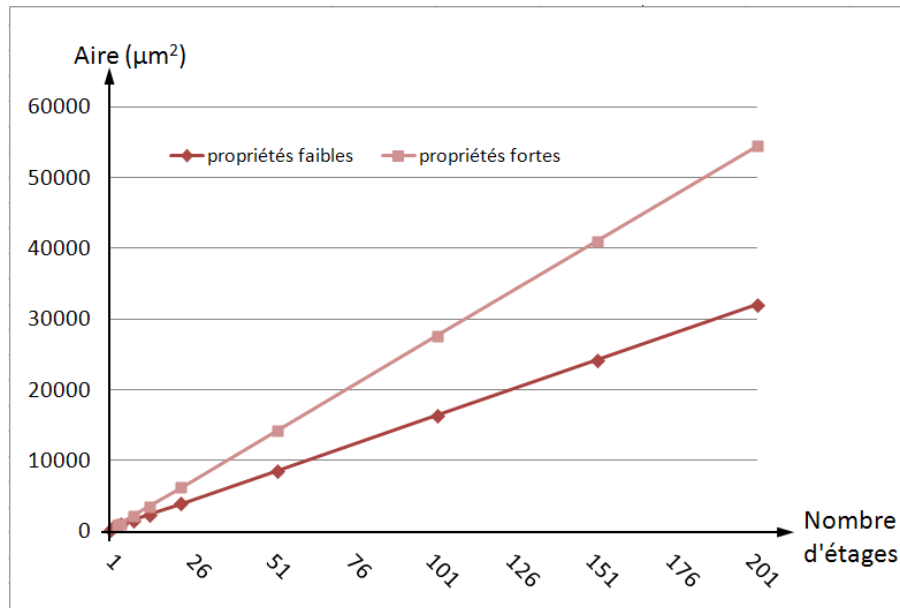


FIGURE 5.1 – Croissance linéaire de l'aire avec la complexité des propriétés implémentées

On voit immédiatement que les moniteurs correspondant à l'implémentation d'une propriété forte nécessitent plus de surface que leurs équivalents faibles mais surtout cette courbe permet de montrer que la croissance de l'aire des moniteurs asynchrones est bien linéaire avec l'augmentation de la complexité des propriétés PSL à implémenter. On a donc bien réussi à garder l'avantage de la méthode d'interconnexion empruntée à la solution synchrone et cela nous permet de produire rapidement des moniteurs complexes dont la taille reste acceptable pour une utilisation embarquée.

Il reste à vérifier si les moniteurs asynchrones ainsi construits apportent davantage de robustesse que les moniteurs synchrones.

5.2 Comparaison de la robustesse des solutions synchrones et asynchrones

Afin de comparer les deux solutions, on implémente les moniteurs synchrone et asynchrone de la propriété A_1 :

property A_1 is assert always(*ticket and valide*) $\rightarrow$ next! *open until!_ fin_passage*);

5.2.1 Environnement de simulation et résultats

Les moniteurs synchrones et asynchrones produits par Horus sont importés en vue "schematic" dans Cadence Analog Environment. Les *stimuli* correspondant aux signaux à évaluer sont générés à l'aide de sources de tension. Comme pour la simulation du *Synchro@clk*, on ajoute des "buffers" entre les sources de tension et l'entrée des moniteurs afin d'avoir des signaux présentant des fronts de montée ou de descente plus proches d'un comportement physique réel. Enfin, il est indispensable d'avoir une bascule pour resynchroniser le signal *Reset_n* avec l'horloge (*c.f.* page 81). Cet environnement de simulation

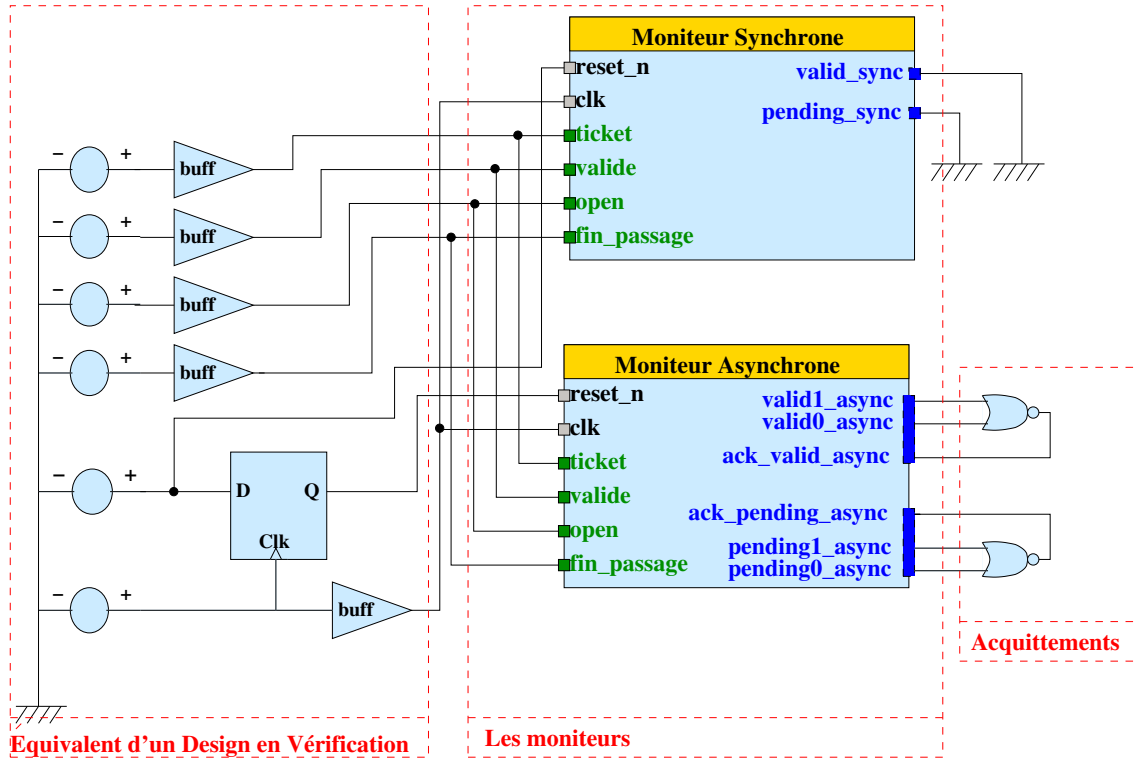


FIGURE 5.2 – Environnement de simulation

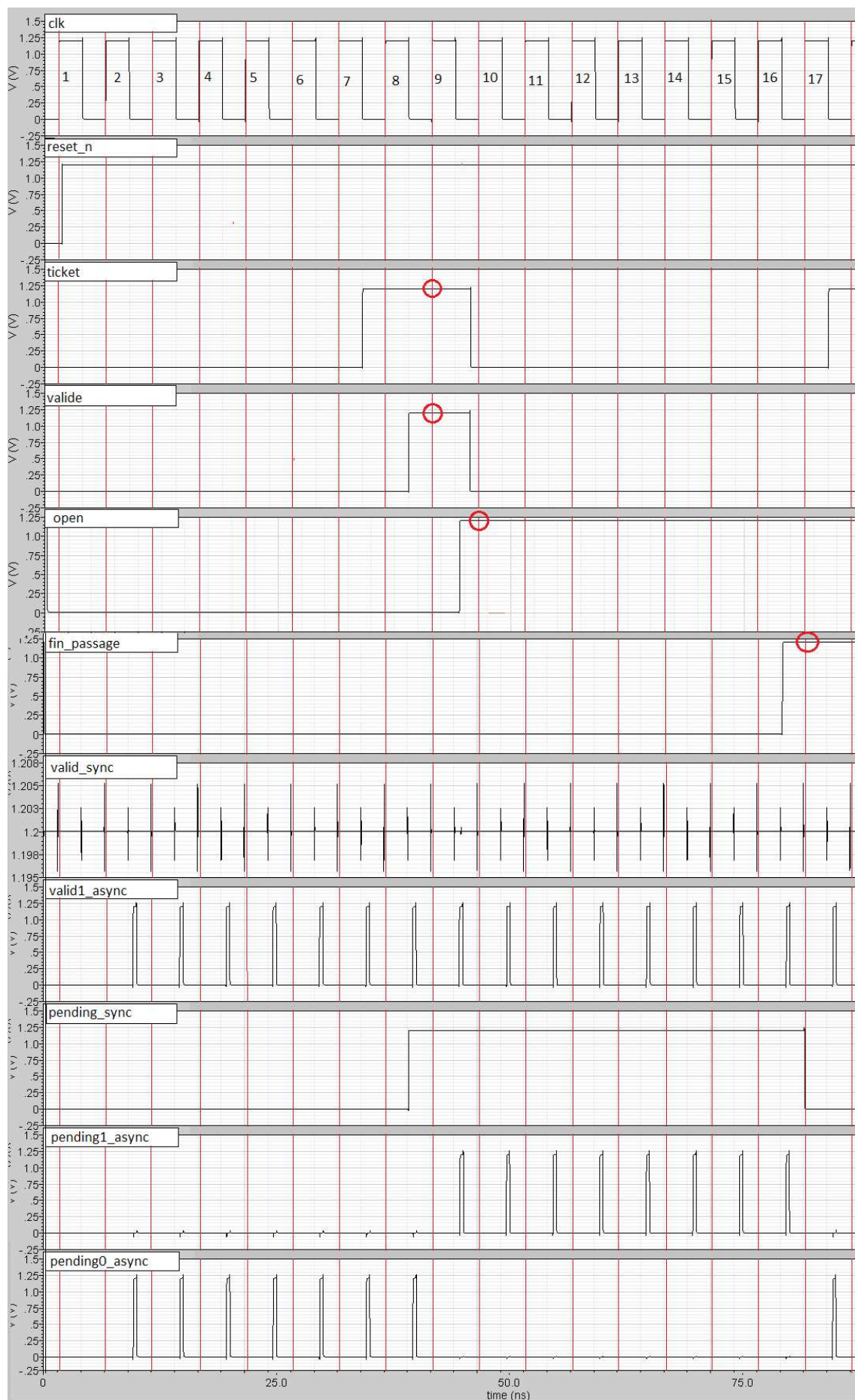
est présenté figure 5.2.

Simulation en conditions nominales On réalise une première simulation avec la tension d'alimentation nominale $V_{dd} = 1,2V$ et une période de 5 nanosecondes. On obtient le chronogramme de la figure 5.3. Lors du premier front montant de Clk , le signal $Reset_n$ est actif. La première évaluation de la propriété aura donc lieu au second front montant d'horloge.

Au neuvième front montant de Clk on a la partie droite de l'implication (*ticket* and *valide*) vraie, l'évaluation de la propriété A_1 est donc *Pending*. On constate que la sortie *Pending1_async* est bien active après le neuvième front montant de Clk . Pour tous les cycles antérieurs la propriété est vraie car la partie droite de l'implication est fausse. On a donc à chaque fois $Valid_sync='1'$ et $Pending_sync='0'$ ainsi que $Pending0_async$ et $Valid1_async$ actifs.

Au dixième front montant de Clk on a bien $open='1'$ mais $fin_passage='0'$. La propriété est toujours en cours d'évaluation et la condition de terminaison du *until!* n'a pas été rencontrée. On obtient en sortie des moniteurs les signaux $Valid_sync='1'$ et $Pending_sync='1'$ ainsi que $Pending1_async$ et $Valid1_async$ actifs. La propriété reste dans l'état *Pending* ($Pending_sync='1'$, $Valid_sync='1'$, $Pending1_async$ et $Valid1_async$ actifs) jusqu'à ce qu'on rencontre la condition de terminaison du *until!* ou jusqu'à ce que $open$ retombe à '0'.

Au dix-septième front montant de Clk , $fin_passage='1'$, on a la condition de terminaison du *until!*, et pour les cycles 10 à 17, le signal $open$ était actif. L'évaluation commencée au neuvième cycle est donc terminée et vraie. De plus pour les cycles 10 à 17, la partie droite de l'implication était fausse et donc, A_1 est vraie par vacuité. On obtient alors $Va-$

FIGURE 5.3 – Simulation analogique des moniteurs de A_1

lid_sync='1' et *Pending_sync*='0' ainsi que *Pending0_async* et *Valid1_async* actifs comme résultats d'évaluation pour le dix-septième cycle.

On constate avec cette évaluation que les signaux de sortie des moniteurs asynchrones et synchrones indiquent le même état pour l'évaluation de la propriété à un détail près. Lorsque *ticket and valide*='1', le signal *Pending_sync* passe immédiatement à '1' alors que le signal *Pending1_async* n'est disponible qu'après le front montant de *Clk*. Cela provient tout simplement du fait que le signal *Pending_sync* est en grande partie combinatoire alors que le signal *Pending1_async* ne peut être calculé qu'après un échantillonnage de *ticket and valide*='1'. Pour avoir exactement les mêmes résultats il suffirait de mettre une bascule sur la sortie *Pending_sync*.

Simulation en conditions agressives : $V_{dd} = 0,6V$ Afin de vérifier la robustesse de nos moniteurs asynchrones par rapport à leurs équivalents synchrones, on simule de nouveau les moniteurs de la propriété A_1 mais en utilisant une tension d'alimentation bien plus faible. Cette tension d'alimentation faible simule des conditions de fonctionnement dégradées.

On simule avec les mêmes signaux d'entrée pour *valide*, *ticket*, *open* et *fin_passage*, on doit donc obtenir une sortie *Valid1_async* après chaque front montant de *Clk*. Lors du premier front montant de *Clk*, *Reset_n* est au niveau bas et donc l'évaluation de A_1 commence lors du deuxième front montant de *Clk*. Il n'est même pas nécessaire de faire la simulation complète, on voit très facilement sur le chronogramme de la figure 5.4 que l'on rate des cycles d'évaluation. En effet, il y a six fronts montants de *Clk* après celui où le *Reset_n* est actif et l'on obtient que trois sorties.

5.2.2 Explication des résultats

Afin de comprendre pourquoi nos moniteurs asynchrones ratent un cycle alors que les moniteurs synchrones continuent à être fonctionnels, on reprend les conditions de simulation précédentes et on observe les sorties des *Synchro@clk* présents dans les moniteurs primitifs asynchrones. Le chronogramme de la figure 5.5 montre les entrées/sorties du synchroniseur présent dans le moniteur primitif asynchrone *M_signal*. Ce moniteur est le dernier moniteur primitif utilisé pour synthétiser le moniteur asynchrone de A_1 . Il est utilisé pour évaluer le signal *open* et calculer le résultat de l'évaluation de A_1 à l'aide de ses signaux d'entrées *Start1* et *Start0*. Le signal double rails *Start* provient des évaluations faites par les moniteurs primitifs précédents qui évaluent *valide*, *ticket* et *fin_passage*.

On constate le comportement suivant en sortie de notre synchroniseur :

- **A** : lors du premier front montant de *Clk*, *Reset_n* est actif, l'évaluation de A_1 commence au cycle suivant.
- **B** : la sortie *ExprS0* correspond à l'évaluation de *open*='0' lors du deuxième front montant de *Clk* n'est disponible qu'après le front descendant de *Clk*.
- **C** : le résultat de l'évaluation de *valide*, *ticket* et *fin_passage* réalisée par les moniteurs primitifs précédents (*Start0*) parvient au *M_signal*.
- **D** : le *M_signal* calcule sa sortie et génère un signal *Valid1*.
- **E** : la création du jeton *Valid1* permet l'acquiescement des signaux *Start* et *ExprS*.

Un signal *Start* est nécessaire pour consommer les sorties du *Synchro@clk* et l'on constate dans la précédente simulation que ce dernier arrive après le troisième front montant de *Clk*. En effet, suite à la baisse de V_{dd} , l'augmentation des délais dans la partie

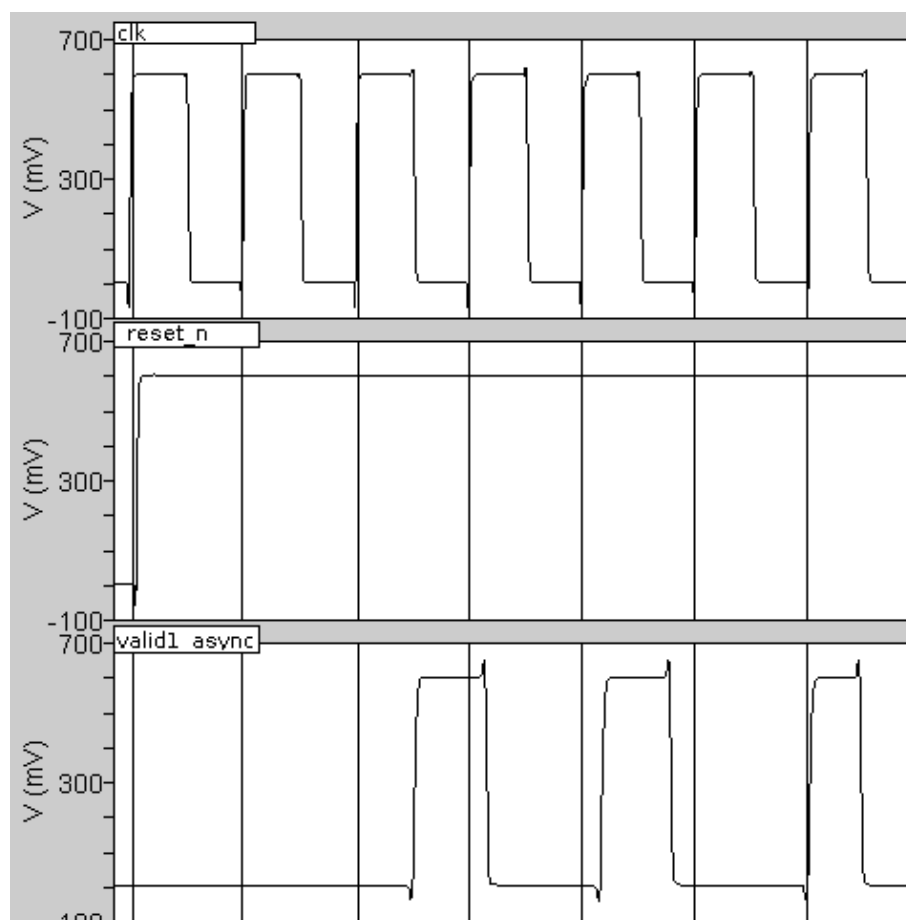
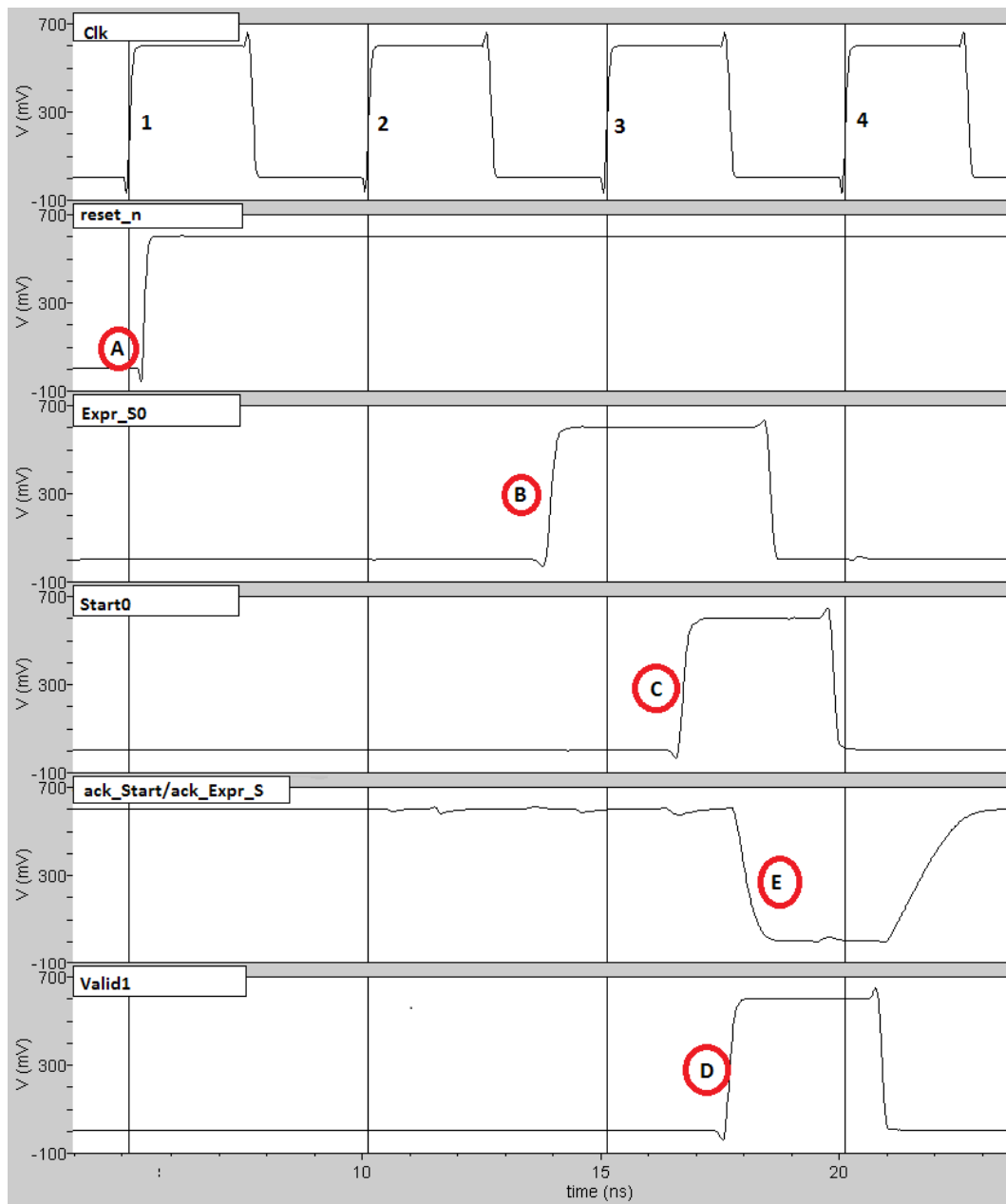


FIGURE 5.4 – Mise en évidence d'un défaut de robustesse

FIGURE 5.5 – Problème d’acquittement des sorties du *Synchro@clk*

QDI des moniteurs provoque un retard trop important sur la propagation des jetons *Start* et *Trigger* entre les moniteurs primitifs. Le temps de cycle de nos moniteurs augmente et devient plus important que la période d'horloge du circuit synchrone et en conséquence, la sortie *ExprS0* du *Synchro@clk* n'est acquittée que bien après le troisième front montant d'horloge. Le synchroniseur ayant encore une donnée présente sur ses sorties, il ne peut pas fournir le résultat de l'évaluation de *open* ayant lieu au troisième cycle.

Le comportement précédent présente une violation de la propriété *env3* mise en évidence lors de la preuve du *Synchro@clk* (c.f. partie 4.3.6, page 86). Cette propriété impose une condition temporelle sur l'acquiescement des sorties d'un *Synchro@clk* qui doit avoir lieu avant un nouvel échantillonnage. Lorsqu'elle n'est pas respectée, bien que les moniteurs asynchrones et les *Synchro@clk* continuent à fonctionner correctement, une évaluation n'est pas correctement effectuée par un des *Synchro@clk* et le résultat fourni par les moniteurs asynchrones n'a plus de sens. On voit bien comment la synchronisation d'un moniteur asynchrone QDI, et plus généralement d'un circuit asynchrone QDI, avec un circuit synchrone propage les hypothèses temporelles du domaine synchrone dans le domaine asynchrone.

La solution synchrone reste fonctionnelle car dans les moniteurs synchrones, le nombre de portes que le signal *Start* doit traverser avant d'atteindre le *M_signal* est bien plus faible. Ainsi, la fréquence maximale de fonctionnement des moniteurs synchrones est plus élevée que le temps de cycle des moniteurs asynchrones. La marge, ou "slack", disponible sur le chemin critique synchrone est telle que même une baisse de tension permet un fonctionnement correct.

5.3 Implémentation d'une solution : le *Clk stretching*

Si on souhaite tirer partie de la robustesse de la technologie asynchrone, il faut que les *Synchro@clk* embarqués dans chaque moniteur primitif QDI respectent la propriété *env3*. C'est-à-dire, il faut que les sorties de chaque synchroniseur du moniteur asynchrone soient acquittées avant un nouvel instant d'échantillonnage. Dans le cas contraire, le moniteur asynchrone rate un cycle d'évaluation de la propriété qu'il vérifie et le résultat fourni par le moniteur n'a plus aucun sens. La contrainte formulée par la propriété *env3* est purement temporelle, on décide donc d'agir sur les hypothèses temporelles du circuit synchrone, et plus particulièrement la plus contraignante d'entre elles, l'horloge. Si l'horloge est contrainte pour assurer le respect de la propriété *env3*, le résultat de l'évaluation fournie par le moniteur asynchrone sera correcte et le circuit synchrone à vérifier n'aura subi aucune modification de son architecture ou de son fonctionnement.

5.3.1 Principe

Le "*Clk stretching*" doit permettre d'étirer l'horloge si le temps nécessaire à l'acquiescement des sorties d'un *Synchro@clk* entraîne une violation de la propriété *env3*. Ainsi, on a deux fonctionnements possibles pour le système "circuit synchrone observé + moniteur asynchrone". En fonctionnement normal, le système fonctionne à la fréquence initialement prévue pour le circuit synchrone. En fonctionnement dégradé, si le temps nécessaire à la résolution des quatre phases du protocole de sortie du *Synchro@clk* est trop important, pour éviter une violation de la propriété *env3*, on étire l'horloge, c'est-à-dire on allonge ponctuellement la durée d'une période. Afin d'éviter des problèmes avec les temps de hold

2. Le Clk_manager a reçu le signal *Ack_Clk_mgt* à l'état bas, il doit permettre de laisser passer un front descendant de *Clk* vers *Clk_mgt*.
3. Les sorties des synchroniseurs sont acquittées pendant que *Clk_mgt* est dans l'état bas. Lorsque cela est terminé, on a fini les quatre phases du protocole poignée de main.
4. Le Clk_manager est averti de la fin des quatre phases du protocole grâce au signal *Ack_Clk_mgt* qui repasse dans l'état haut. Le Clk_manager peut laisser passer un nouveau front de *Clk* vers *Clk_mgt*.

En résumé, on veut que le bloc Clk_manager laisse, d'une part, passer les fronts montants de *Clk* vers *Clk_mgt* lorsque le signal *Ack_Clk_mgt* est dans l'état haut et, d'autre part, laisse passer les fronts descendants de *Clk* vers *Clk_mgt* quand le signal *Ack_Clk_mgt* est dans l'état bas. Dans les autres cas, *Clk_mgt* est maintenu dans son état, bas ou haut.

5.3.2 Définition d'un nouveau synchroniseur

Pour avoir le fonctionnement souhaité, il est nécessaire de créer un nouveau synchroniseur. En effet, dans le *Synchro@clk*, les sorties sont disponibles après le front descendant du signal d'horloge et le fonctionnement explicité précédemment montre que les sorties doivent être disponibles après le front montant de l'horloge. Il est donc indispensable de créer un nouveau synchroniseur : le *Synchro_qdi*.

5.3.2.1 Architecture et fonctionnement du *Synchro_qdi*

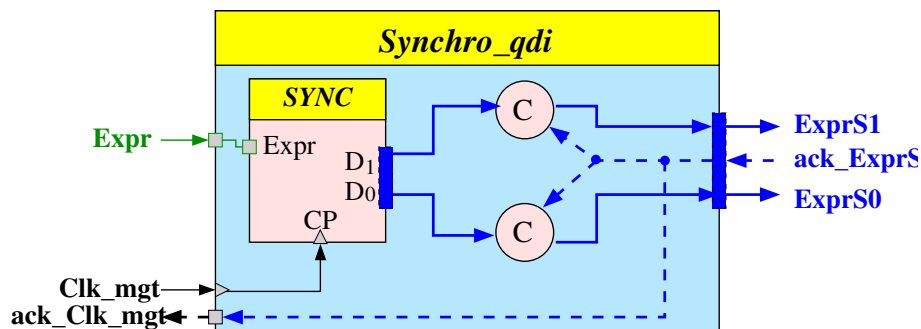
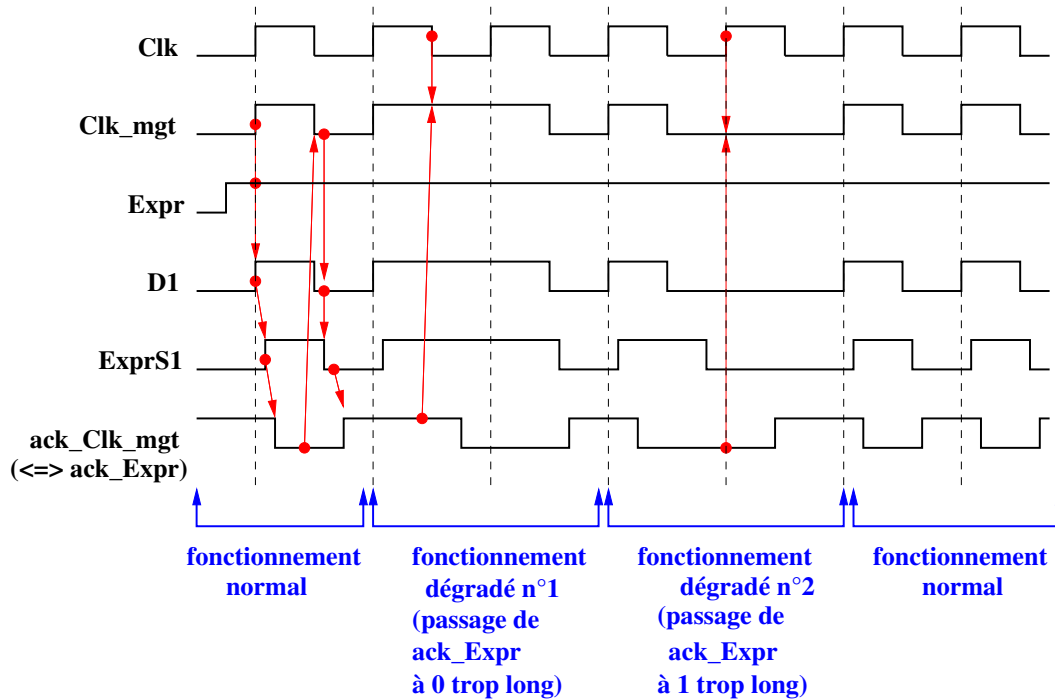


FIGURE 5.7 – Architecture du *Synchro_qdi*

L'architecture de ce nouveau synchroniseur est donnée figure 5.7. Pour ce synchroniseur, l'échantillonnage n'est plus assuré par une Flip-Flop mais par le composant SYNC de la bibliothèque TAL (*c.f.* partie 2.7, page 50). L'adaptation de protocole n'est plus assuré par des "générateurs de jetons" mais par de simples portes de Müller. Le chronogramme de fonctionnement du *Synchro_qdi* est disponible figure 5.8. Pour des raisons de lisibilité, on suppose que *Expr* est toujours à '1', ainsi il n'est pas nécessaire de représenter les chronogrammes de *D0* et *ExprS0*. De même, les problèmes d'initialisation seront abordés dans la suite, on suppose pour l'instant que l'initialisation est correctement réalisée.

Fonctionnement normal En fonctionnement normal, on a les quatre phases suivantes :

1. Les sorties *ExprS* du *Synchro_qdi* sont inactives et l'acquiescement *Ack_ExprS* est au niveau haut et donc *Ack_Clk_mgt* aussi.

FIGURE 5.8 – Fonctionnement du *Synchro_qdi*

2. Si un front montant de *Clk* est présent en entrée du *Clk_manager*, il peut être transmis sur sa sortie *Clk_mgt*. Le *Synchro_qdi* fournit une sortie *ExprS*. L'acquittement *Ack_ExprS* passe à '0' et donc le *Clk_manager* reçoit le signal *Ack_Clk_mgt* à '0'.
3. Le bloc *Clk_manager* peut recopier le prochain front descendant de *Clk* sur *Clk_mgt*. Les sorties du SYNC repassent donc à '0' et comme le signal d'acquittement est au niveau bas, les sorties du *Synchro_qdi* sont désactivées.
4. Les sorties *ExprS* ne sont plus actives et donc les signaux *Ack_ExprS* et *Ack_Clk_mgt* repassent dans l'état haut. Les quatre phases du protocole sont terminées et le bloc *Clk_manager* pourra transmettre le prochain front montant de *Clk* vers *Clk_mgt*.

Fonctionnement dégradé 1 Ce fonctionnement correspond au cas où le signal d'acquittement des sorties du *Synchro_qdi* met du temps à passer dans l'état bas, c'est-à-dire que les sorties du *Synchro_qdi* n'ont pas encore été acceptées par le bloc QDI suivant. On a les quatre phases suivantes :

1. Les sorties du *Synchro_qdi* sont inactives et le signal *Ack_ExprS* est dans l'état haut. Le signal *Ack_Clk_mgt* est dans l'état haut.
2. Le bloc *Clk_manager* laisse passer un front montant de *Clk* vers *Clk_mgt*. Des sorties sont produites par le *Synchro_qdi*.
3. Lors du front descendant de *Clk*, l'acquittement de *ExprS* n'est pas encore passé dans l'état bas. Cela signifie que la donnée n'a pas été reçue par le bloc suivant le *Synchro_qdi*. Le signal *Ack_Clk_mgt* est dans l'état bas et le bloc *Clk_manager* empêche la transmission d'un front descendant de *Clk* vers *Clk_mgt*, bloquant *Clk_mgt* dans l'état haut, jusqu'à ce que l'acquittement soit repassé dans l'état bas et que l'on puisse reprendre un fonctionnement normal. Une fois le signal d'acquittement

Synchro_qdi_f, est donnée figure 5.9. On présente aussi le chronogramme de fonctionnement du *Synchro_qdi_f* figure 5.10.

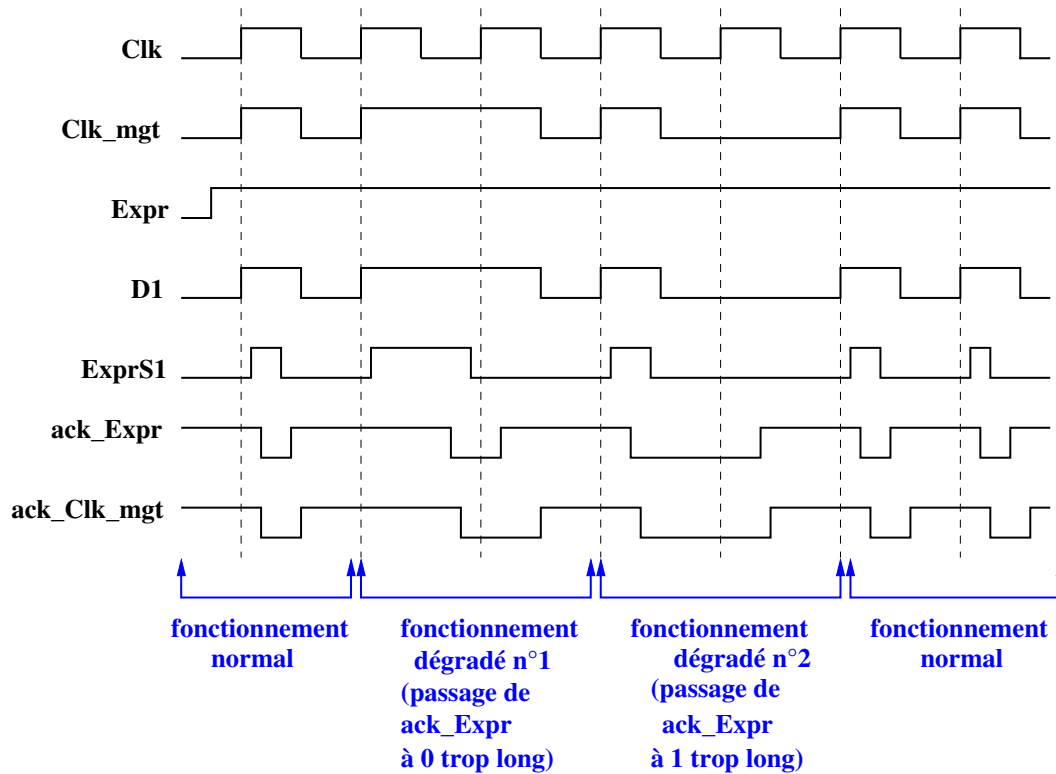


FIGURE 5.10 – Fonctionnement du *Synchro_qdi_f*

Fonctionnement normal En fonctionnement normal, on a les quatre phases suivantes :

1. Les sorties *ExprS* sont inactives, le signal *Ack_ExprS* est dans l'état haut et les sorties du SYNC sont à '0' car *Clk_mgt* est dans l'état bas. On a *Ack_Clk_mgt* dans l'état haut.
2. Le *Clk_manager* recopie un front de *Clk* sur *Clk_mgt*. L'une des sorties du SYNC devient active et comme *Ack_ExprS* est à '1', le "générateur de jetons" ayant un front sur son entrée *Data* va générer une sortie *ExprS*. Le signal d'acquittement *Ack_ExprS* passe à '0' et donc le signal *Ack_Clk_mgt* aussi.
3. Le *Clk_manager* a reçu l'acquittement *Ack_Clk_mgt* à '0', il peut donc recopier le prochain front descendant de *Clk* sur *Clk_mgt*. Cela entraîne l'inactivation des sorties du SYNC. Dans le même temps, la sortie *ExprS* repasse à '0' car la sortie de la porte de Müller responsable de l'acquittement de *ExprS* a basculé à '0'.
4. En conséquence, *Ack_ExprS* repasse dans l'état haut. À cet instant, le signal *Ack_Clk_mgt* repasse dans l'état haut. On retrouve dans l'état de la première phase et le *Clk_manager* peut accepter un nouveau front montant sur *Clk*.

Fonctionnements dégradés 1 et 2 Dans le cas du fonctionnement dégradé 1, c'est-à-dire lorsque que l'acquittement de *ExprS* met trop de temps à passer dans l'état bas, la sortie de la porte de Müller fournissant le signal *Ack_Clk_mgt* ne peut pas basculer dans

l'état bas. Le *Clk_manager* n'ayant pas reçu de signal *Ack_Clk_mgt* à '0', il ne laisse pas passer les fronts descendants de *Clk* vers *Clk_mgt*. Ainsi, *Clk_mgt* reste bloqué dans l'état haut jusqu'à ce que *Ack_ExprS* soit inactif et que *Ack_Clk_mgt* puisse basculer à '0' et que l'on repasse en fonctionnement normal.

Dans le cas du fonctionnement dégradé 2, c'est-à-dire lorsque que le signal d'acquittement *Ack_ExprS* reste trop longtemps dans l'état bas, le signal *Ack_Clk_mgt* reste aussi bloqué dans l'état bas. Le *Clk_manager* laisse donc passer un front descendant de *Clk* vers *Clk_mgt* mais ne laissera pas passer de front montant tant que *Ack_Clk_mgt*, et donc *Ack_ExprS*, ne soit repassé à '1'.

5.3.3 Contrôle de l'horloge : le Clk_manager

On a réalisé deux synchroniseurs permettant le "clock stretching" (c.f. partie 5.3.1, page 101) et il faut désormais réaliser le bloc *Clk_manager*. Ce dernier doit permettre le passage d'un front montant de *Clk* vers *Clk_mgt* lorsque *Ack_Clk_mgt* est dans l'état haut et il doit permettre le passage d'un front descendant de *Clk* vers *Clk_mgt* quand *Ack_Clk_mgt* est dans l'état bas. Dans les autres cas, le *Clk_manager* doit maintenir sa sortie *Clk_mgt* dans l'état où elle se trouve. Ces autres cas correspondent à un fonctionnement où *Ack_Clk_mgt* reste bloquée dans l'état haut ou bas. On obtient l'implémentation de la figure 5.11 où la première porte de Müller (*C1*) permet le passage de l'état bas de *Clk* vers la sortie *Clk_mgt* quand la deuxième porte de Müller (*C2*) permet le passage de l'état bas de *Clk* vers *Clk_mgt*. Le maintien dans un état de l'horloge de sortie, *Clk_mgt*, est dû au signal d'acquittement *Ack_Clk_mgt*. En effet, étirer l'horloge n'est nécessaire que lorsque les sorties d'un synchroniseur ne sont pas acquittées assez vite, c'est-à-dire quand *Ack_Clk_mgt* reste bloqué dans un état. Comme *Ack_Clk_mgt* est une entrée de *C1* et de *C2* et que les portes de Müller possèdent une fonction mémorisante, les sorties de *C1* (*C1o*) et de *C2* (*C2o*) restent bloquées dans leur état.

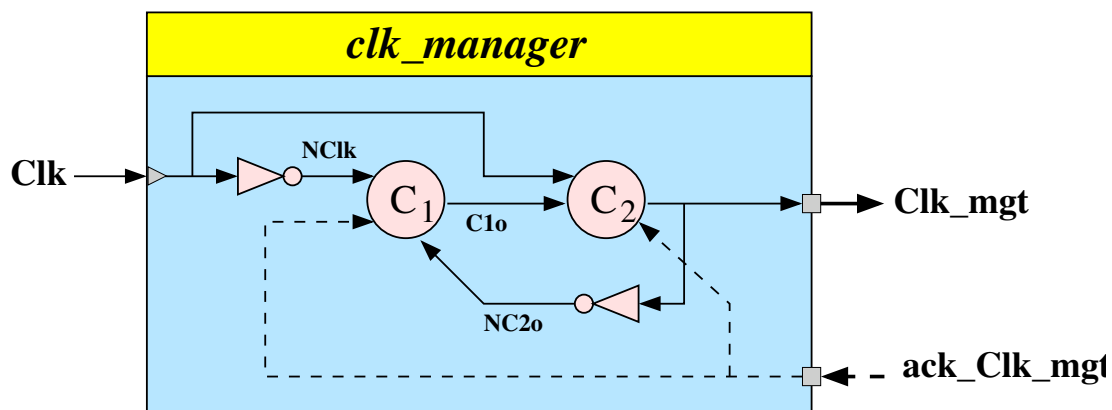


FIGURE 5.11 – Architecture du *Clk_manager*

Les deux synchroniseurs, *Synchro_qdi* et *Synchro_qdi_f*, ne peuvent fournir de sorties que s'ils reçoivent un front montant sur l'entrée *Clk_mgt*. Afin de ne pas avoir de problème d'initialisation, il suffit de s'assurer que l'initialisation se passe correctement dans le *Clk_manager*. On utilise donc une bascule prenant en entrée le signal *Reset_n* et qui met à jour sa sortie *Reset_n_sync* sur les fronts montants de *Clk*. On obtient ainsi le chronogramme de fonctionnement du *Clk_manager* présenté sur la figure 5.12.

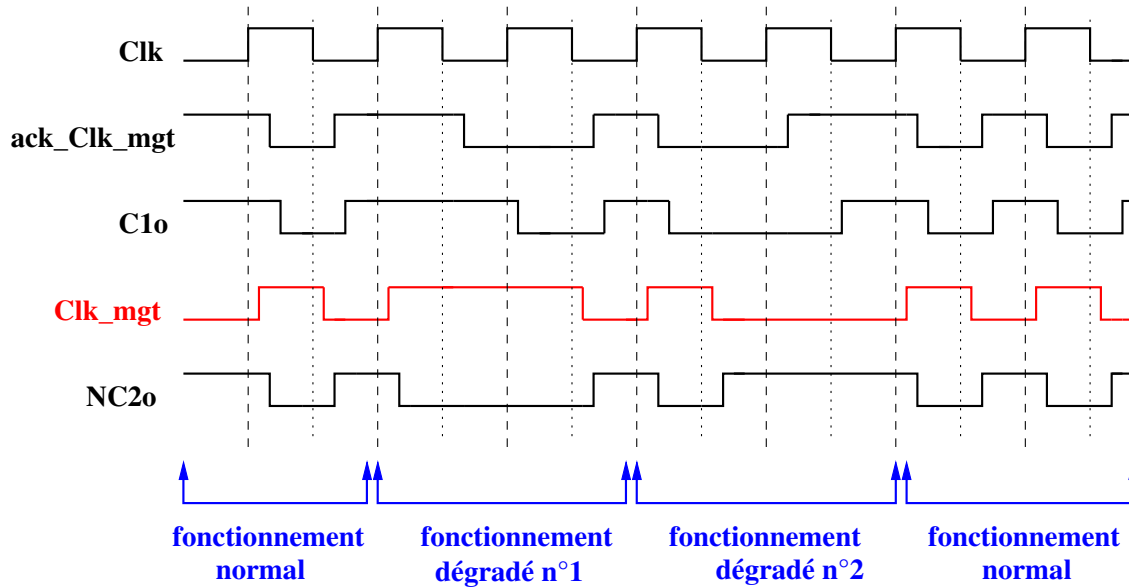


FIGURE 5.12 – Fonctionnement du Clk_manager

En fonctionnement normal, on a les quatre étapes suivantes :

1. Avant le front montant de Clk , les signaux Ack_Clk_mgt , $NC2o$ (signal $C2o$ inversé) et $Nclk$ (signal Clk inversé) sont à '1' provoquant le basculement de $C1o$ à '1'. Lors du front montant de Clk , la sortie de la porte $C2$ peut basculer à '1', on a un front montant sur Clk_mgt .
2. Après un moment mais avant que Clk ne repasse à '0', le signal Ack_Clk_mgt passe à '0'. Comme $Nclk='0'$ et $NC2o='0'$, le signal $C1o$ passe à '0'.
3. Clk passe à '0', comme $C1o$ et Ack_Clk_mgt sont dans l'état bas, le signal Clk_mgt passe dans l'état bas. Ainsi $NC2o$ passe dans l'état haut.
4. Ack_Clk_mgt repasse dans l'état haut avant que Clk ne présente un nouveau front montant. Le dispositif est prêt pour accepter un nouveau front montant de Clk (phase 1).

En fonctionnement dégradé 1, on a réalisé la première phase du dispositif. On a donc $Clk_mgt='1'$, $Clk='1'$, $C1o='1'$ et $NC2o='0'$. Si le signal Ack_Clk_mgt ne passe pas à '0' et qu'un front descendant a lieu sur Clk , alors $C1o$ n'a pas eu la possibilité de basculer à '0'. Comme $C1o$ est une entrée pour la porte $C2$, cette dernière ne peut faire basculer sa sortie Clk_mgt qui reste dans l'état haut jusqu'à la reprise d'un fonctionnement normal.

En fonctionnement dégradé 2, on a réalisé les phases 1,2 et 3 du dispositif. On a donc $Clk_mgt='0'$, $Clk='0'$, $C1o='0'$ et $NC2o='1'$. Si le signal Ack_Clk_mgt ne passe pas à '1' avant qu'un front montant ait lieu sur Clk , alors $C1o$ n'a pas eu la possibilité de basculer à '1'. Comme $C1o$ est une entrée pour la porte $C2$, cette dernière ne peut faire basculer sa sortie Clk_mgt qui reste dans l'état bas jusqu'à la reprise d'un fonctionnement normal.

5.4 Validation de l'implémentation du "clock stretching"

Afin de valider les précédents raisonnements sur l'implémentation de la solution "clock stretching", on réalise tout d'abord une série de simulations numériques. Puis de la même manière que l'on avait réalisé la preuve du *Synchro@clk*, on fera la preuve du bon fonctionnement de cette solution et enfin on illustrera l'intérêt d'une telle solution sur des simulations analogiques. Enfin, la solution sera brièvement caractérisée en aire et en fréquence.

5.4.1 Validation numérique

Afin de disposer de moniteurs asynchrones permettant l'implémentation du "clock stretching", on implémente une deuxième bibliothèque de moniteurs primitifs. Ces moniteurs primitifs ont quasiment la même interface que les moniteurs de la première bibliothèque présentés dans la partie 3.1.1, page 54. La seule différence est la présence d'une sortie de plus, le signal *Ack_Clk_mgt*. De plus, pour se synchroniser avec le circuit synchrone à observer, ces moniteurs primitifs n'utilisent plus un *Synchro@clk* mais un *Synchro_qdi* ou un *Synchro_qdi_f*.

Pour s'assurer du bon fonctionnement de la solution de "clock stretching", on réalise de nouveau la simulation de la propriété A_1 dans l'environnement MODELSIM en ajoutant le bloc *Clk_manager* pour générer le signal *Clk_mgt*. Le signal *Ack_Clk_mgt* est obtenu en réalisant des rendez-vous successifs entre les signaux *Ack_Clk_mgt* de chaque moniteur primitif. Ces rendez-vous sont réalisés à l'aide de portes de Müller comme l'illustre la figure 5.13 qui présente le banc de test.

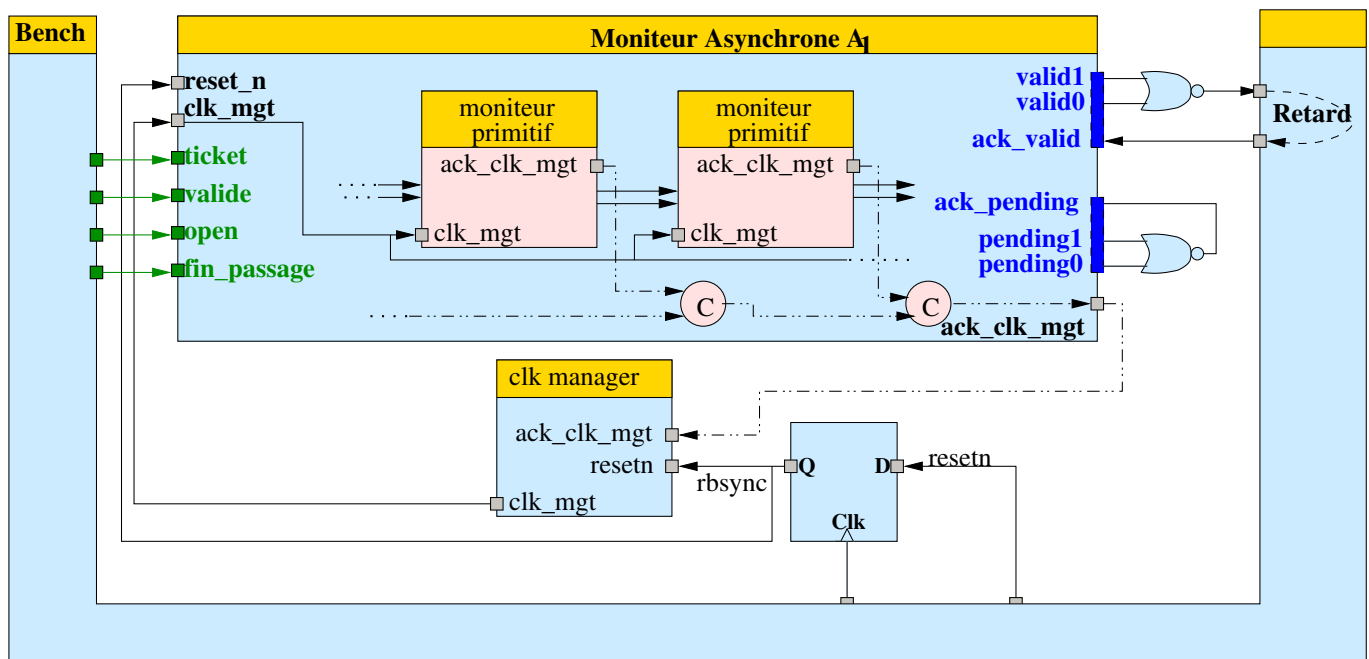


FIGURE 5.13 – Banc de test pour la simulation numérique de A_1

On remarque que le signal d'initialisation *rb_sync* est encore généré par une bascule afin d'éviter que les aléas sur le signal *Reset_n* ne provoquent des remises à zéro intempestives du *Clk_manager* ou du moniteur asynchrone. Cette bascule est cadencée par le signal *Clk* qui correspond à l'horloge initiale. De plus, afin de provoquer des comportements dégradés, c'est-à-dire des temps d'acquittement longs, on ajoute un retard sur le chemin du signal *ack_valid* pour les besoins de la validation.

Simulation en fonctionnement normal On réalise une première simulation avec un retard nul afin de vérifier qu'en condition de fonctionnement normal, le signal *Clk_mgt* à la même période que le signal d'horloge initiale *Clk*. On obtient le chronogramme présenté figure 5.14.

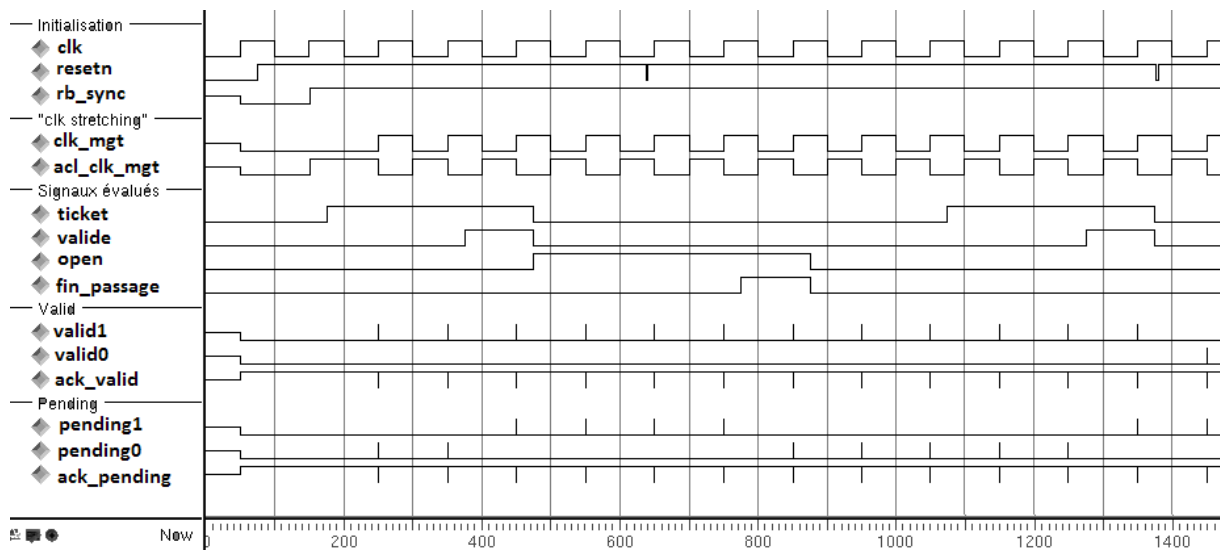
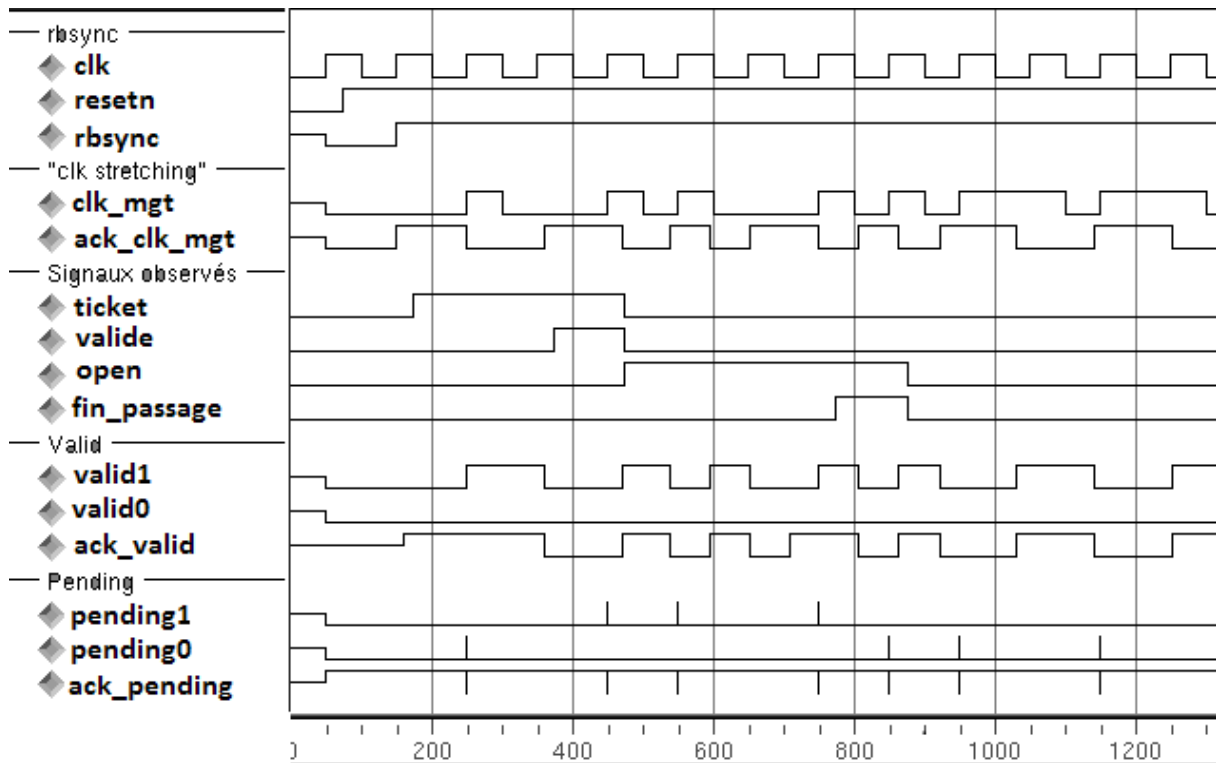


FIGURE 5.14 – Résultat de la simulation numérique de A_1

Sur ce chronogramme, on voit tout d'abord que les aléas sur le signal *Reset_n* (à 650ns et 1370ns) ne provoquent pas de problème dans le fonctionnement de notre système car ils sont filtrés par la bascule. On remarque aussi immédiatement que *Clk_mgt* a la même période que l'horloge initiale, c'est-à-dire *Clk*. Enfin, on constate que l'évaluation de A_1 par le moniteur asynchrone est correctement réalisé car on voit l'état "pending" de la propriété lorsque la partie droite de l'implication (*ticket and valide*) est vraie et on détecte la violation de propriété à $t = 1450ns$.

Simulation en fonctionnement dégradé On constate que l'implémentation de la solution de "clock stretching" permet d'avoir la même horloge que celle initialement prévue pour le signal synchrone si les acquittements des sorties des synchroniseurs des moniteurs primitifs sont acquittés suffisamment vite, c'est-à-dire si la propriété *env3* est vérifiée. On veut maintenant valider la capacité du système à étirer l'horloge lorsque les temps d'acquittement sont trop grands. En pratique, si on retarde l'acquittement des sorties du moniteur synchrone, on va aussi retarder l'acquittement sur les sorties des synchroniseurs. On réalise donc une simulation de A_1 en introduisant ponctuellement des retards sur l'acquittement du signal de sortie *Valid* du moniteur. On obtient le chronogramme de la figure 5.15.

FIGURE 5.15 – Résultat de la simulation numérique de A_1 en fonctionnement dégradé

On voit que les temps d'acquiescement de la sortie *Valid* du moniteur sont effectivement augmentés puisque *Ack_Valid* reste bloqué dans l'état haut ou l'état bas. Il résulte de ces retards d'acquiescement un étirement de l'horloge *Clk_mgt*.

À $t = 350ns$, le signal *Valid1* n'a pas encore été acquitté car *Ack_Valid* est encore à '1'. Au niveau des synchroniseurs cela se traduit par un signal d'acquiescement bloqué dans l'état bas. Afin de ne pas violer la propriété *env3* et d'assurer ainsi la robustesse liée à l'utilisation de la technologie QDI, l'horloge *Clk_mgt* est maintenue dans l'état bas. Cela correspond au fonctionnement dégradé 1.

À $t = 450ns$ et jusqu'à $t = 600ns$, les sorties *Valid* sont acquittées suffisamment vite et on peut avoir un fonctionnement normal où *Clk_mgt* a la même forme que l'horloge initiale *Clk*.

À $t = 1000ns$, le signal *Valid1* a été acquitté mais le signal *Ack_Valid* n'est pas encore repassé à '1' et donc on ne peut pas encore accepter une nouvelle donnée sur la sortie *Valid* du moniteur. Au niveau des synchroniseurs, cela se traduit par un signal d'acquiescement bloqué dans l'état haut. On voit que dans ce cas, qui correspond au fonctionnement dégradé 2, l'horloge *Clk_mgt* est maintenue dans l'état haut.

On constate qu'en plus de correctement étirer l'horloge, notre système permet une évaluation correcte de la propriété A_1 sur les fronts montants de *Clk_mgt*. En effet, lors du premier front montant de *Clk_mgt*, la partie droite de l'implication étant fautive, la propriété est valide par vacuité : on obtient le couple de sortie *Pending0* et *Valid1*. Lors du deuxième front montant de *Clk_mgt*, la partie droite de l'implication est vraie, l'évaluation de la propriété est donc en cours et on obtient le couple de sortie *Pending1* et *Valid0*. Pour les deux fronts montants suivants, *open* est actif mais pas *fin_passage*, la propriété n'est

pas violée et est toujours en cours d'évaluation, on obtient deux fois le couple de sortie *Valid1* et *Pending1*. Enfin, au front montant de *Clk_mgt* suivant, *open* et *fin_passage* sont actifs, l'évaluation commencée au deuxième front montant de *Clk_mgt* est terminée et *A₁* est valide. On obtient le couple de sortie *Valid1* et *Pending0*.

Bien que les simulations numériques soient convaincantes quant au bon fonctionnement du "clock stretching", on veut être sûr de la robustesse de cette dernière. Dans ce but, on va réaliser la preuve du système "synchroniseur + Clk_manager".

5.4.2 Preuve formelle

Lors de l'étape de preuve du précédent synchroniseur, le *Synchro@clk*, une limitation temporelle forte provenant du domaine synchrone était propagée au synchroniseur, empêchant d'assurer le bon respect du protocole de communication des sorties avec les parties QDI des moniteurs asynchrones. La solution de "clock stretching" doit permettre de réaliser la preuve sans avoir recours à la propriété *env3* (c.f. page 86). Si on peut réaliser la preuve de la correction du protocole de sortie du *Synchro_qdi_f* (ou *Synchro_qdi*) connecté au Clk_manager sans cette contrainte temporelle, cela signifie que notre solution sera QDI.

Afin de réaliser la preuve, on ré-utilise la méthode présentée plus tôt dans ce manuscrit (c.f. page 83 sur le système présenté figure 5.16). Le *Half Buffer* représente le reste du moniteur primitif dans lequel se trouve le *Synchro_qdi_f*. Les propriétés explicitant le comportement de l'environnement sont *env1* et *env2* (c.f. page 84), c'est-à-dire le respect par l'environnement du protocole quatre phases poignée de main et la présence de front sur *Clk*. Les blocs matériels sont décrits en utilisant les propriétés PSL modélisant le comportement des divers composants comme la Müller à deux entrées. Ces descriptions sont fournies en annexe B.2.

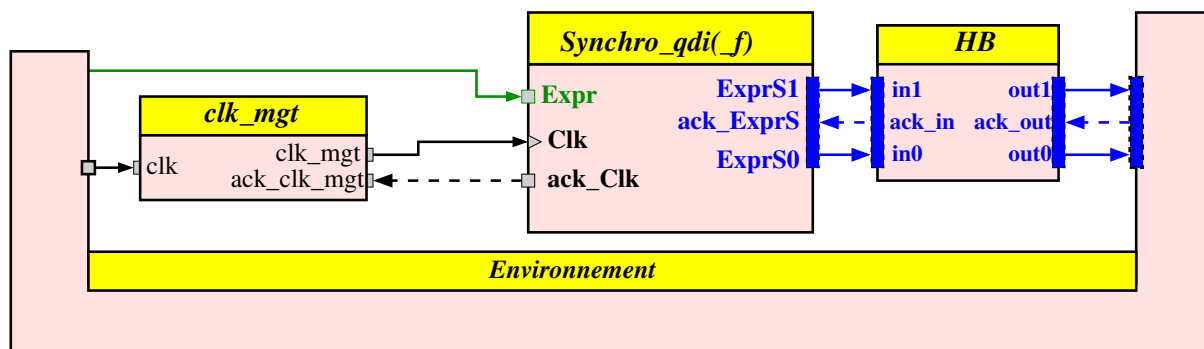


FIGURE 5.16 – Environnement de preuve

De la même manière que l'on avait prouvé le respect des propriétés *proof1*, *proof2* et *proof3* pour le *Synchro@clk* (c.f. partie 4.3, page 82), on prouve le respect de ces propriétés pour les sorties des synchroniseurs *Synchro_qdi* et *Synchro_qdi_f*. Cela assure le respect du protocole quatre phases pour les sorties *Expr* des synchroniseurs. La preuve étant réalisée sans l'hypothèse supplémentaire *env3*, il n'y a plus de limitation temporelle et le synchroniseur, ainsi que les moniteurs primitifs, fonctionneront quelque soit le temps d'acquiescement de leurs sorties. On retrouve un fonctionnement purement QDI de nos moniteurs et donc la robustesse héritée de l'utilisation de ces circuits.

5.4.3 Validation analogique

Pour vérifier le bon fonctionnement de notre solution en tenant compte des effets analogiques, on simule de nouveau le moniteur de la propriété A_1 dans ANALOG ENVIRONMENT de l'outil Cadence. Cette fois, les synchroniseurs présents dans les moniteurs primitifs sont des *Synchro_qdi_f*. Le banc de simulation est illustré figure 5.17. On note la présence d'une bascule pour générer le signal *rb_sync*, ainsi que la présence de "buffers" entre les sources de tensions générant les signaux observés et les entrées du moniteur asynchrone.

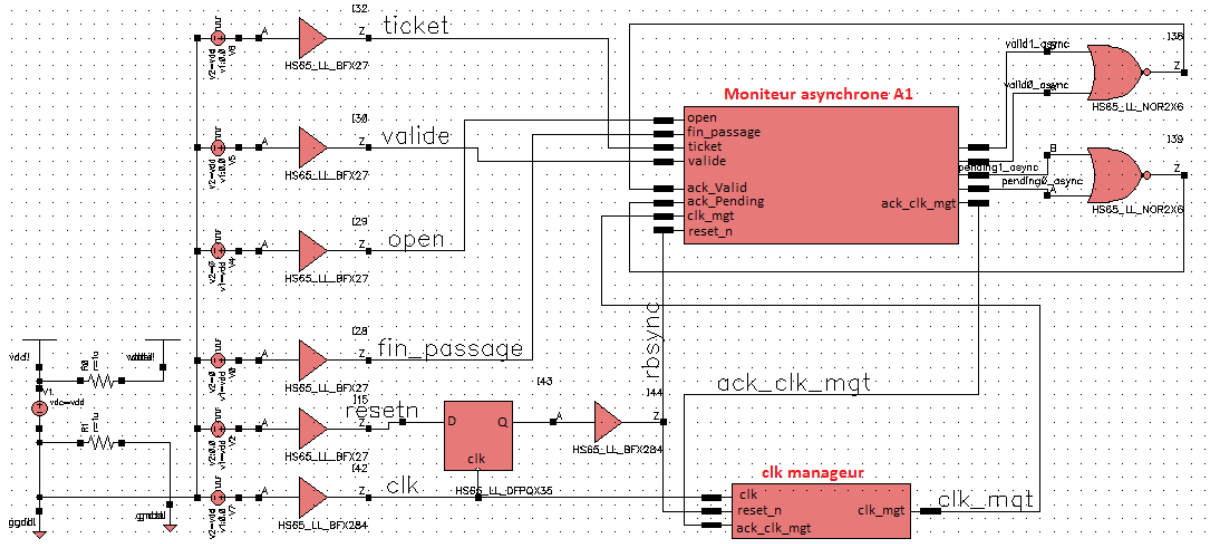


FIGURE 5.17 – Banc de simulation analogique

En fonctionnement normal On réalise une première simulation en fonctionnement normal. Pour cela on reprend les conditions de simulation utilisées dans la validation analogique des moniteurs (*c.f.* page 96). On obtient le chronogramme de la figure 5.18 qui présente uniquement les signaux *Clk*, *Clk_mgt*, *Ack_Clk_mgt* et la sortie *Valid* du moniteur A_1 implémenté à l'aide de *Synchro_qdi_f* et du *Clk_manager*.

Sur ce chronogramme on voit immédiatement que *Clk_mgt* a la même fréquence que *Clk* et que pour chaque front montant de *Clk_mgt*, le moniteur A_1 fournit une sortie *Valid* et une sortie *Pending*¹.

On réalise la même simulation en introduisant des chutes de la tension d'alimentation afin de vérifier que la solution de "clock stretching" fonctionne correctement. Le résultat de cette simulation est présenté sur la figure 5.19.

On voit que lorsque $vdd=1,2V$ pour les intervalles $t \in [0 : 20]$, $[47 : 68]$, $[103 : 117]$ et $[144 : 150]$, l'horloge calculée par le *Clk_manager* a la même période que l'horloge *Clk* initialement utilisée par le circuit synchrone.

Dans l'intervalle $t \in [20 : 46]$, on a $vdd=0,7V$ provoquant une augmentation des délais dans le circuit. L'horloge s'étire afin d'assurer la résolution des quatre phases du protocole de sortie des différents *Synchro_qdi_f*. Le temps durant lequel *Clk_mgt* est dans l'état haut,

1. Le détail n'est pas présenté ici mais les valeurs de sorties obtenues en simulation correspondent aux valeurs attendues.

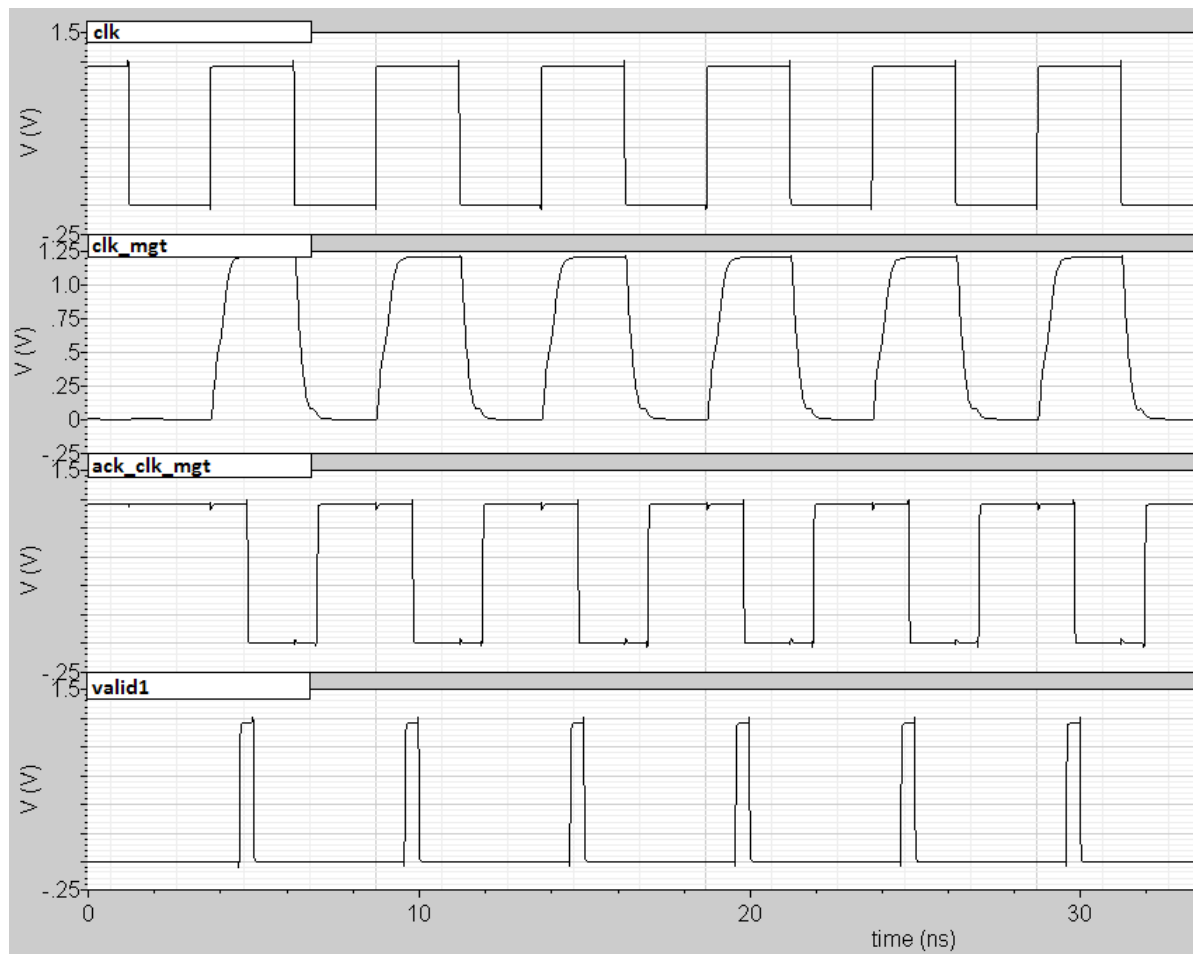


FIGURE 5.18 – Résultat de simulation analogique en fonctionnement normal

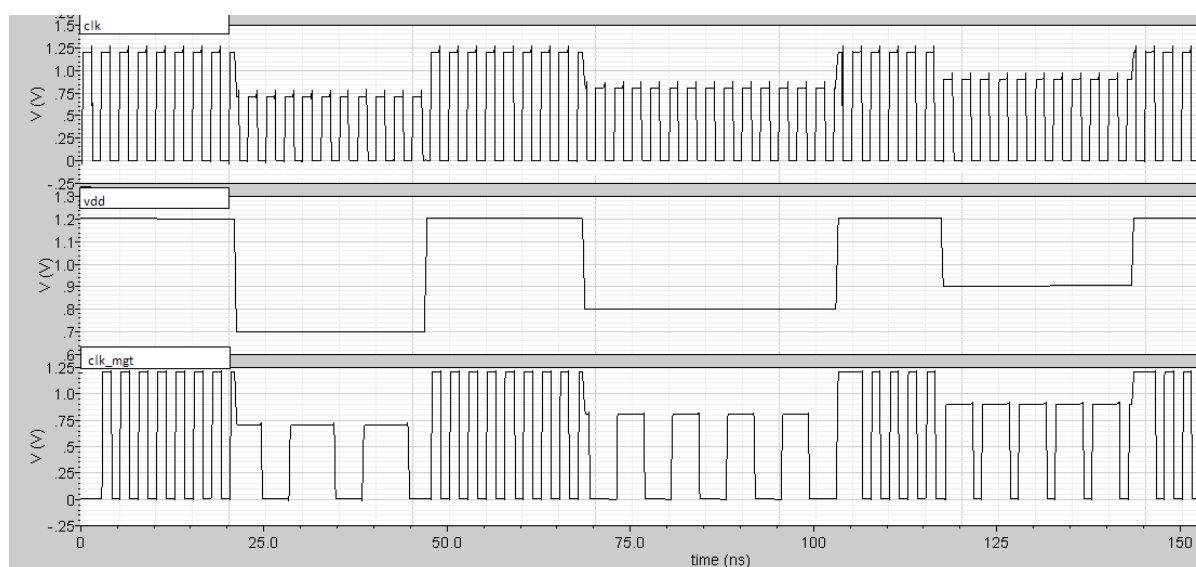


FIGURE 5.19 – Résultat de simulation analogique en fonctionnement dégradé

respectivement bas, est multiplié par quatre, respectivement par trois, comparé à la demi période de *Clk*.

Dans l'intervalle $t \in [69 : 102]$, on a $vdd=0,8V$. Cette fois encore le système mis en place étire l'horloge. Comme la baisse de tension est moins prononcée que précédemment, l'augmentation des délais est moins importante et le temps durant lequel *Clk_mgt* est dans l'état haut n'est plus multiplié que par trois comparé à la demi période de *Clk*.

Dans le dernier intervalle, $t \in [118 : 143]$, $vdd=0,9V$. Dans ce dernier cas on peut remarquer que les temps de maintien haut et bas du signal *Clk_mgt* sont encore réduits, en accord avec une augmentation des délais moins importante.

On constate donc que cette solution est capable de s'adapter à la volée aux conditions de fonctionnement du système "circuit monitoré + moniteur asynchrone". L'absence de cycles de latence lors de la modification de la forme d'onde du signal *Clk_mgt* permet aux moniteurs asynchrones de ne rater aucun cycle d'évaluation. On montre sur la figure 5.20 le résultat de l'évaluation de la propriété A_1 par le moniteur asynchrone.

Ce chronogramme montre que chaque fois qu'il y a un front montant sur le signal *Clk_mgt*, il y a une sortie *Pending* et une sortie *Valid* calculée. Grâce à cela, à chaque cycle, la propriété A_1 est évaluée par le moniteur asynchrone et le résultat de l'évaluation est correcte. Lors du septième front montant de *Clk_mgt*, *ticket and valide* est actif, l'évaluation devient donc pending mais reste valide. On obtient alors le couple de sortie *Pending1* et *Valid1*. Lors des huitième et neuvième fronts montants de *Clk_mgt*, *open* est actif mais pas le signal *fin_passage*. La propriété est toujours en cours d'évaluation et l'on obtient encore le couple de sortie *Pending1* et *Valid1*. Enfin, lors du dixième front montant de *Clk_mgt*, le signal *open* est encore actif et *fin_passage* est actif aussi. L'évaluation de A_1 commencée au septième cycle est terminée et la propriété n'est pas violée. On obtient donc le couple de sortie *Pending0* et *Valid1*. Les résultats fournis par les moniteurs asynchrones sont donc corrects.

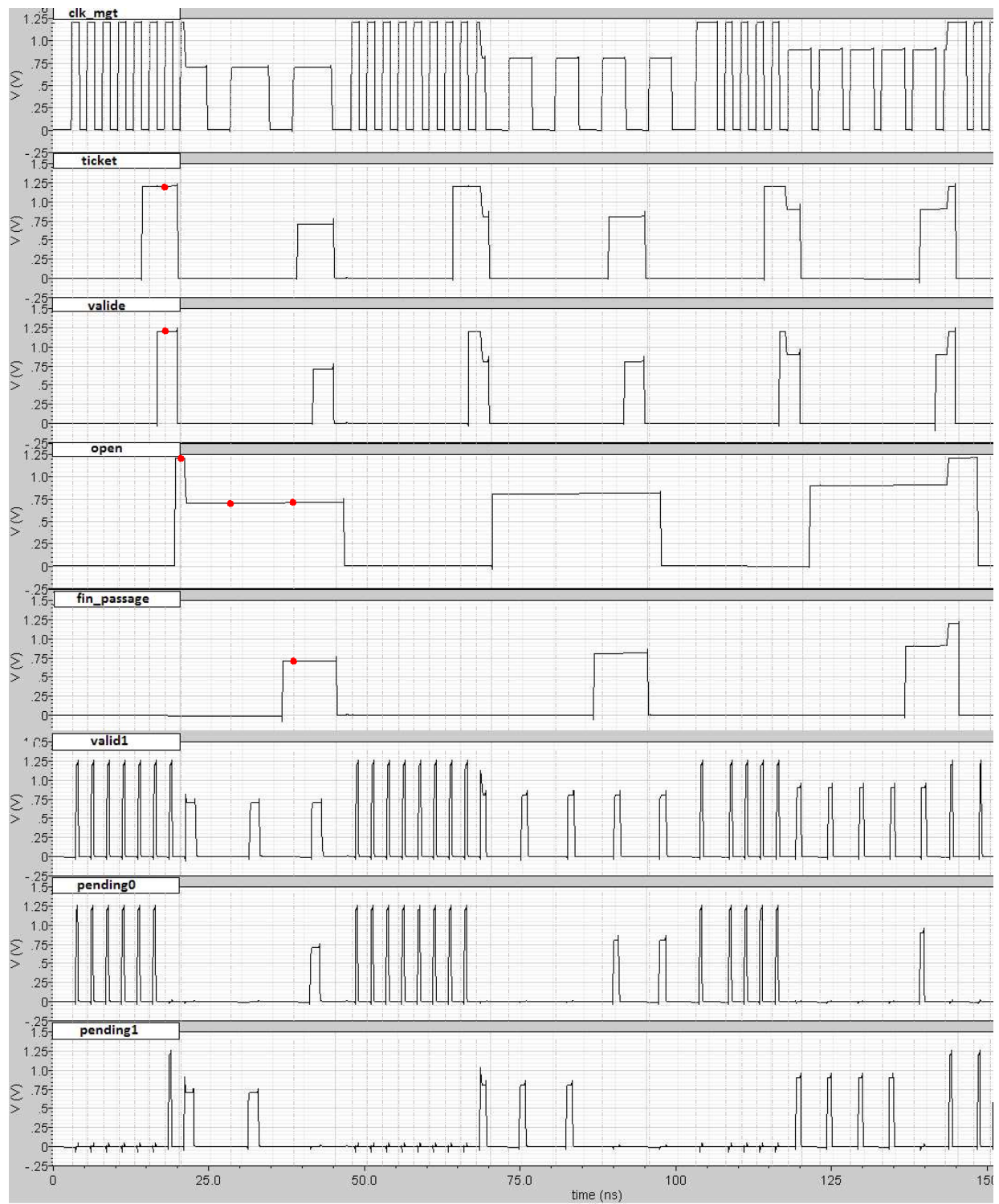
5.4.4 Caractérisation de la solution

Placement routage Afin de caractériser cette nouvelle solution on réalise l'étape de placement routage sur les deux synchroniseurs et sur le *Clk_manager*. Le placement routage est effectué à l'aide de SOC ENCOUNTER comme précédemment pour le *Synchro@clk* et la technologie utilisée est toujours la CMOS 65nm Low Power, Low VT. Le tableau 5.3 présente la surface des différents éléments utilisés pour l'implémentation de la solution de "clock stretching".

Élément	Aire(μm^2)
SYNC	45,2
<i>Synchro_qdi_f</i>	116,3
<i>Synchro_qdi</i>	96,4
<i>Clk_manager</i>	31,8

Table 5.3 – Éléments utiles au "clock stretching" placés routés

On constate que même si le composant SYNC est un composant d'une bibliothèque, il utilise beaucoup plus de surface que la flip flop initialement utilisée pour le *Synchro@clk*. En conséquence, bien que l'on ait moins de portes dans le *Synchro_qdi* et le *Synchro_qdi_f*

FIGURE 5.20 – Résultat d'évaluation de A_1 en fonctionnement dégradé

que dans le *Synchro@clk*, ces derniers restent toutefois plus importants en taille. Ce surplus d'aire consommée est nécessaire pour implémenter une version QDI et robuste des moniteurs asynchrones sans modifier trop la bibliothèque de moniteurs élémentaires et l'interconnexion de ces derniers.

Fréquence de fonctionnement Afin de connaître le fonctionnement de notre solution et notamment la fréquence de fonctionnement de celle-ci, on utilise le moniteur A_1 implémenté avec des *Synchro_qdi*. On va chercher à connaître en fonction de la tension d'alimentation V_{dd} , à partir de quelle fréquence le système étire effectivement l'horloge et donc, ralentit le fonctionnement du circuit synchrone observé. On réalise la même caractérisation pour le moniteur A_1 implémenté à l'aide de *Synchro_qdi_f*. On obtient les courbes de la figure 5.21.

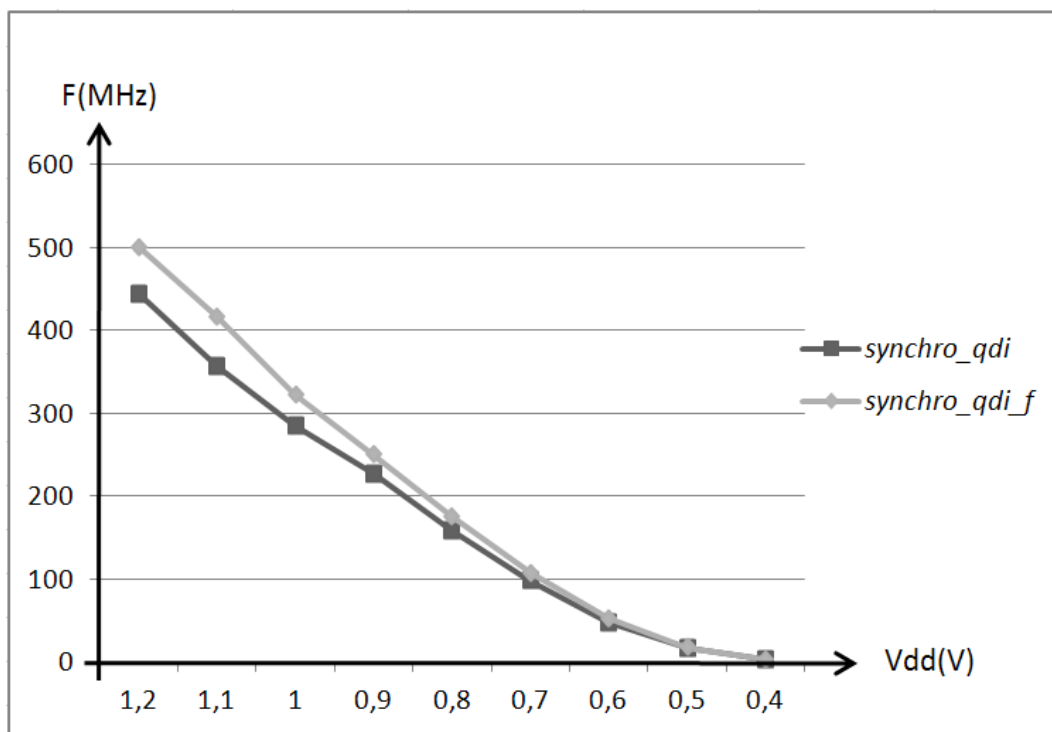


FIGURE 5.21 – Caractérisation en fréquence du moniteur A_1

On voit que le *Synchro_qdi_f* permet grâce au générateur de jetons de gagner un peu de temps sur la résolution des quatre phases du protocole poignée de main. En effet, pour des tensions d'alimentation supérieures à 0,7V le *Synchro_qdi_f* permet de découpler l'état des sorties du synchroniseur de l'état des sorties du SYNC. Comme l'état des sorties du SYNC dépend de *Clk_mgt*, cela permet de découpler l'état des sorties du synchroniseur de l'état de *Clk_mgt*. En conséquence, l'acquittement des sorties des synchroniseurs peut avoir lieu avant que *Clk_mgt* ne soit passé dans l'état bas et le temps de nécessaire à la résolution du protocole quatre phases est diminué. Cette diminution de temps de cycle permet d'augmenter la fréquence à partir de laquelle l'horloge est étirée. Pour des tensions inférieures, cela n'est plus intéressant car les délais dans les connexions et les portes sont si importants que le temps nécessaire à la résolution des quatre phases du protocole n'est plus influencé par le couplage de l'état des sorties du synchroniseur avec *Clk_mgt*.

5.5 Conclusion sur la robustesse des moniteurs asynchrones

On vient de montrer qu'il était possible de garantir l'insensibilité aux délais, et donc la robustesse, de nos moniteurs asynchrones. Toutefois, afin de garantir la quasi-insensibilité aux délais, il est indispensable de supprimer la contrainte temporelle issue du domaine synchrone. Cette contrainte dépend de l'horloge et il est donc nécessaire de contrôler l'horloge. La solution de "clock stretching" permet ce contrôle de l'horloge uniquement dans le cas où le calcul du moniteur n'est pas terminé, dans les autres cas l'horloge n'est pas modifiée. Cette solution présente l'avantage d'agir uniquement sur l'horloge du circuit à vérifier et non pas sur le circuit lui même.

Chapitre 6

Conclusion

Sommaire

6.1	Bilan	120
6.1.1	Implémentation de moniteurs asynchrones	120
6.1.2	Le "clock stretching"	120
6.2	Perspectives	121
6.2.1	Amélioration de la robustesse d'un circuit synchrone grâce à l'utilisation d'un moniteur asynchrone	121
6.2.2	Moniteurs asynchrones vérifiant des circuits asynchrones	121
6.2.3	Synchronisation entre un domaine synchrone et un domaine asynchrone	122

6.1 Bilan

L'ABV se démocratise et est aujourd'hui supportée par de plus en plus de simulateurs industriels. Une partie des recherches touchant au domaine de l'ABV se concentre sur la possibilité de vérifier des propriétés sur un circuit en fonctionnement, embarqué dans un FPGA ou sur Silicium. Des travaux ont montré qu'il était possible d'obtenir de petits circuits matériels, les moniteurs, qui assurent cette tâche. Les moniteurs présentent un intérêt dans la surveillance de circuits en fonctionnement et utiliser ces moniteurs pour la vérification de circuits critiques embarqués semble être une idée à développer. Toutefois, si les moniteurs utilisent la même technologie que le circuit critique à vérifier, il est impossible de garantir la robustesse de ces derniers et donc la bonne vérification du circuit critique.

6.1.1 Implémentation de moniteurs asynchrones

Basée sur la méthode de génération des moniteurs synchrones de la plateforme **Horus** (une bibliothèque de moniteurs primitifs et une méthode d'interconnexion), l'implémentation des moniteurs asynchrones permet de vérifier des propriétés PSL sur des circuits synchrones en fonctionnement. La plateforme **Horus** permet de transformer instantanément une propriété PSL en un circuit asynchrone grâce à la méthode d'interconnexion des moniteurs primitifs asynchrones.

Afin d'adapter la méthode initiale aux moniteurs asynchrones, il a fallu ré-implémenter en asynchrone tous les moniteurs primitifs et modifier les signaux d'entrée sortie de ces derniers. On dispose désormais d'une bibliothèque de moniteurs asynchrones primitifs. Cette bibliothèque est construite sur des modèles génériques de moniteurs primitifs asynchrones. La sémantique de l'opérateur PSL permet de choisir quel modèle générique de moniteur primitif asynchrone doit être utilisé pour obtenir le moniteur correspondant à l'opérateur.

Les moniteurs asynchrones ont été caractérisés par rapport à leurs équivalents synchrones et, bien qu'ils soient plus importants en taille à cause de la nécessité d'implémenter le protocole quatre phases, leur croissance reste linéaire avec la complexité de la propriété et leur synthèse est rapide et facile. De plus, à notre connaissance, aucun travail extérieur au laboratoire ne présente de solution permettant la vérification de propriétés PSL sur un circuit synchrone en fonctionnement à l'aide d'un moniteur matériel asynchrone.

Pour observer les signaux du circuit à vérifier aux bons instants, une synchronisation des domaines synchrone et asynchrone est nécessaire. Elle est réalisée par un synchroniseur. Le principal obstacle à la réalisation de moniteurs asynchrones robustes a pu être mis en évidence grâce à une étape de preuve de ce synchroniseur. Comme l'on souhaite vérifier à l'aide des moniteurs asynchrones la validité de propriété PSL à chaque front montant d'horloge du circuit à vérifier, il faut que le moniteur calcule son résultat en moins d'une période d'horloge sans quoi on ne peut pas évaluer les opérandes au cycle suivant.

Dans le but de supprimer la propagation des hypothèses temporelles du domaine synchrone vers le domaine asynchrone, une technique a été développée : le "clock stretching".

6.1.2 Le "clock stretching"

Le "clock stretching" étire l'horloge lorsqu'un moniteur asynchrone nécessite plus qu'un cycle d'horloge pour calculer le résultat de l'évaluation d'une propriété PSL. On ne modifie pas le circuit à vérifier mais uniquement son horloge. La mise en place de cette solution

a donné lieu à la création d'un bloc, le `Clk_manager` qui peut être utilisé pour résoudre d'autres problèmes de synchronisation entre domaine synchrone et asynchrone.

La solution a été validée grâce, d'une part, à des simulations numériques et analogiques, mais surtout grâce à une étape de preuve formelle. Cette solution permet de garantir les propriétés QDI de nos moniteurs lorsque qu'ils sont utilisés pour vérifier un circuit synchrone. De cette façon on garantit la robustesse de nos moniteurs.

Un dépôt de brevet est à l'étude pour le `Clk_manager` et des études plus poussées sur l'intérêt du "clock stretching" dans l'amélioration de la surveillance et la robustesse des circuits critiques synchrones sont à envisager.

6.2 Perspectives

6.2.1 Amélioration de la robustesse d'un circuit synchrone grâce à l'utilisation d'un moniteur asynchrone

Lors de l'implémentation du "clock stretching", on étire l'horloge lorsque les délais dans les moniteurs asynchrones sont trop importants. Les délais peuvent augmenter par exemple lors d'une baisse de la tension d'alimentation. Dans ce cas, les délais dans le circuit synchrone à vérifier vont eux aussi augmenter et mener jusqu'à une violation de chemin critique, c'est à dire à une erreur dans le circuit synchrone. Si l'horloge a été étirée suite à une augmentation des délais dans le moniteur asynchrone, cela signifie que le temps disponible pour parcourir le chemin critique dans le circuit synchrone a aussi augmenté. Dans ce cas, il est possible que cet étirement de l'horloge permette au circuit synchrone de rester fonctionnel. Le "clock stretching" permet donc à un circuit synchrone couplé à un moniteur asynchrone de potentiellement augmenter sa robustesse vis-à-vis de variations en tension, en température ou aux procédés de fabrication. Ce point constitue à coup sûr une voie de recherche à étudier.

6.2.2 Moniteurs asynchrones vérifiant des circuits asynchrones

L'un des principaux problèmes rencontrés avec les moniteurs asynchrones connectés à un circuit synchrone à vérifier provient de la propagation de l'hypothèse temporelle du synchrone vers le domaine asynchrone (*c.f.* partie 4.3.6, page 86). On a montré à l'aide de RAT qu'il était indispensable que les sorties de chacun des synchroniseurs d'un moniteur asynchrone aient résolu les quatre phases du protocole poignée de mains avant un nouvel instant d'échantillonnage pour assurer le bon fonctionnement du moniteur asynchrone. Avant de réaliser la solution de "clock stretching", une étude sur l'intérêt de synthétiser des moniteurs asynchrones vérifiant des propriétés PSL sur un circuit asynchrone a été réalisée. En effet, si l'on travaille uniquement des circuits asynchrones, on espère voir disparaître l'hypothèse temporelle et ainsi conserver la robustesse des moniteurs asynchrones.

Ces études ont été réalisées grâce à des exemples de circuits asynchrones, un DES asynchrone [MRL⁺05] entièrement décrit en VHDL structurel et un CRC de 8bits [LPF] asynchrone décrit et synthétisé à l'aide de l'outil ACC de Tiempo SAS. La connexion de ces circuits avec les moniteurs asynchrones implémentés dans cette thèse n'a pas été aussi simple que ce qui était prévu initialement et de nombreuses questions ont été soulevées : doit-on implémenter des propriétés portant sur des données transitant dans le circuit asynchrone ou bien sur les signaux régissant le protocole associé à ces données ? Quand

doit-on observer une donnée dans un circuit asynchrone ? Comment assurer l'insensibilité aux délais du système "circuit observé asynchrone + moniteur asynchrone" ? Comment faire lorsque plusieurs protocoles sont présents dans un circuit asynchrone ? Répondre à ces questions est un prérequis pour observer des circuits asynchrones avec des moniteurs asynchrones.

6.2.3 Synchronisation entre un domaine synchrone et un domaine asynchrone

Enfin un autre résultat intéressant connexe aux travaux de ce manuscrit concerne la synchronisation entre deux domaines : l'un synchrone et l'autre asynchrone. En effet, l'un des principaux obstacle à la synthèse des moniteurs asynchrones a été la réalisation d'un synchroniseur efficace pour re-synchroniser nos moniteurs asynchrones avec l'horloge du circuit synchrone à vérifier.

Le travail réalisé dans le chapitre 5 concernait essentiellement des problèmes de contrôle d'une horloge d'un circuit synchrone communiquant avec un circuit asynchrone. L'implémentation de la solution de "clock stretching" a permis de contrôler cette synchronisation entre domaines et ainsi de garantir la correction fonctionnelle du système (moniteur + circuit observé). Lors du développement de cette solution, le bloc `Clk_manager` a été créé.

Les problèmes de synchronisation entre domaines synchrones et asynchrones sont présents dans de nombreux champs d'applications laissant entrevoir des possibilités de réutilisation du `Clk_manager`. Par exemple, ce bloc pourrait intervenir dans les problèmes de synchronisation dans des GALS embarquant des circuits synchrones et un réseau de communication asynchrone par exemple. Dans ce cas le réseau de communication asynchrone n'activerait que l'horloge des blocs synchrones avec lesquels une communication est en cours. L'utilisation du `Clk_manager` peut aussi être envisagée pour la synchronisation dans les SoCs possédant plusieurs domaines d'horloge.

Bibliothèque de moniteurs primitifs asynchrones

Cette annexe a pour but de présenter plus en détail l'architecture de chaque moniteur asynchrone.

A.1 Les moniteurs asynchrones sans cycles de retard et sans ré-entrance

On rappelle l'architecture de ce type de moniteurs primitifs figure A.1

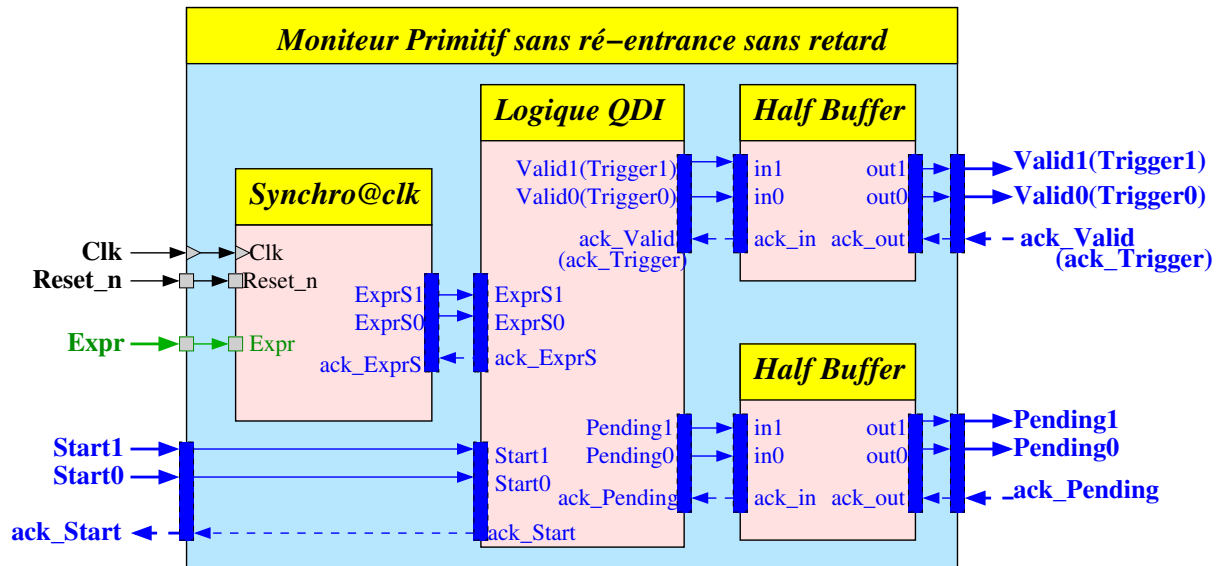


FIGURE A.1 – Moniteur primitif asynchrone sans ré-entrance et sans cycle de retard

A.1.1 Le moniteur M_imply

La partie "Logique QDI" du moniteur primitif M_imply doit implémenter les rendez-vous suivants entre les signaux *Start* et *Cond* :

- $Start0 \& CondS0$ ou $Start0 \& CondS1$: le moniteur n'a pas à tenir compte de l'évaluation de $Cond$ faite au front montant d'horloge précédent. La partie droite de l'implication n'a pas besoin d'être vérifiée. On génère donc une sortie $Trigger0$.
- $Start1 \& CondS0$: le moniteur doit tenir compte de l'évaluation de $Cond$. Comme on a $CondS0$, cela signifie que $Cond$ était à '0' au front montant d'horloge précédent. La partie droite de l'implication n'a pas à être vérifiée et le moniteur génère une sortie $Trigger0$.
- $Start1 \& CondS1$: le moniteur doit tenir compte de l'évaluation de $Cond$. Comme on a $CondS1$, cela signifie que $Cond$ était à '1' au front montant d'horloge précédent. La partie droite de l'implication doit être vérifiée et le moniteur génère une sortie $Trigger1$.

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.2

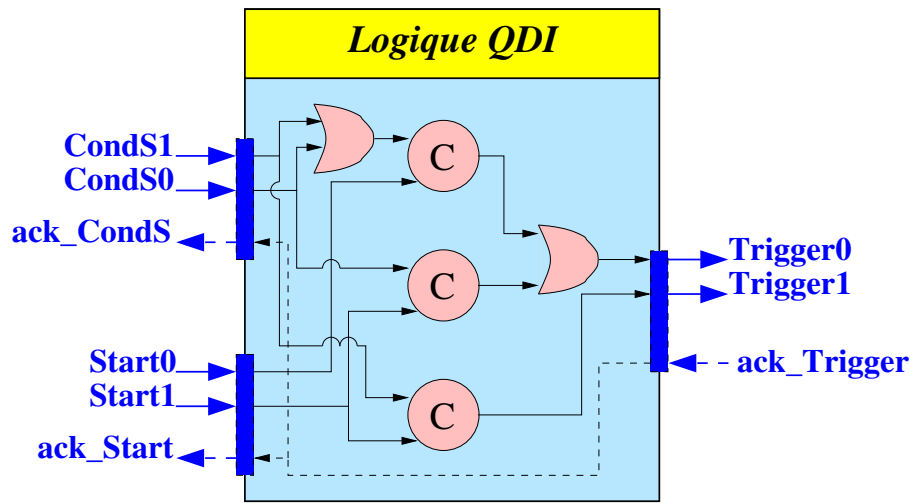


FIGURE A.2 – Implémentation de la partie "Logique QDI" du M_imply

A.1.2 Le moniteur M_signal

Le M_signal vérifie si l'opérande *Expr* est vraie ou fausse. C'est un observateur et son opérande ne peut être que booléen. La partie "Logique QDI" du moniteur primitif M_signal doit implémenter les rendez-vous suivants entre les signaux *Start* et *Expr* :

- *Start0* & *ExprS0* ou *Start0* & *ExprS1* : le moniteur n'a pas à tenir compte de l'évaluation de *Expr* faite au front montant d'horloge précédent. La propriété est considérée comme *Valid* donc le moniteur génère une sortie *Valid1*.
- *Start1* & *ExprS0* : le moniteur doit tenir compte de l'évaluation de *Expr*. Comme on a *ExprS0*, cela signifie que *Expr* était à '0' au front montant d'horloge précédent et le moniteur génère une sortie *Valid0*.
- *Start1* & *ExprS1* : le moniteur doit tenir compte de l'évaluation de *Expr*. Comme on a *ExprS1*, cela signifie que *Expr* était à '1' au front montant d'horloge précédent et le moniteur génère une sortie *Valid1*.

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.3

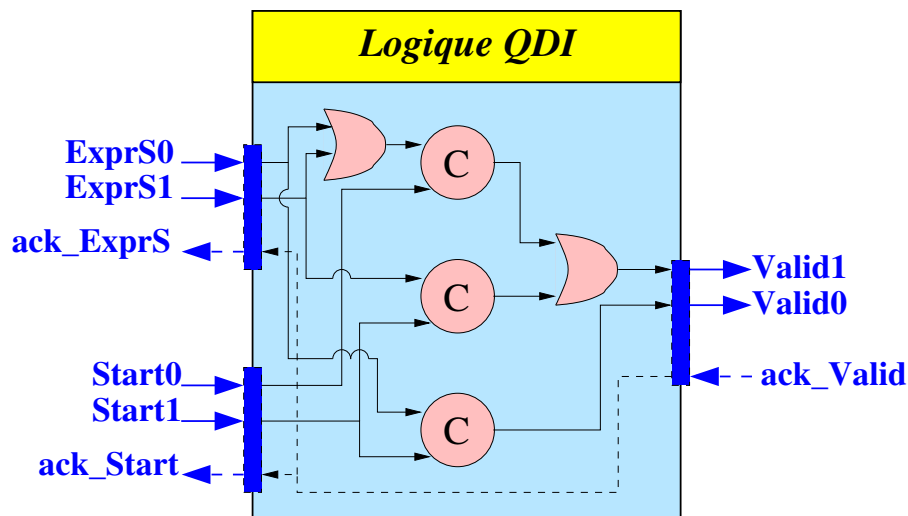


FIGURE A.3 – Implémentation de la partie "Logique QDI" du M_signal

A.2 Les opérateurs sans cycles de retard, avec ré-entrance

On rappelle l'architecture de ce type de moniteurs primitifs figure A.4. L'architecture présentée correspond aux opérateurs forts. L'architecture pour les opérateurs faibles est donnée dans la partie 3.2.5 figure 3.14.

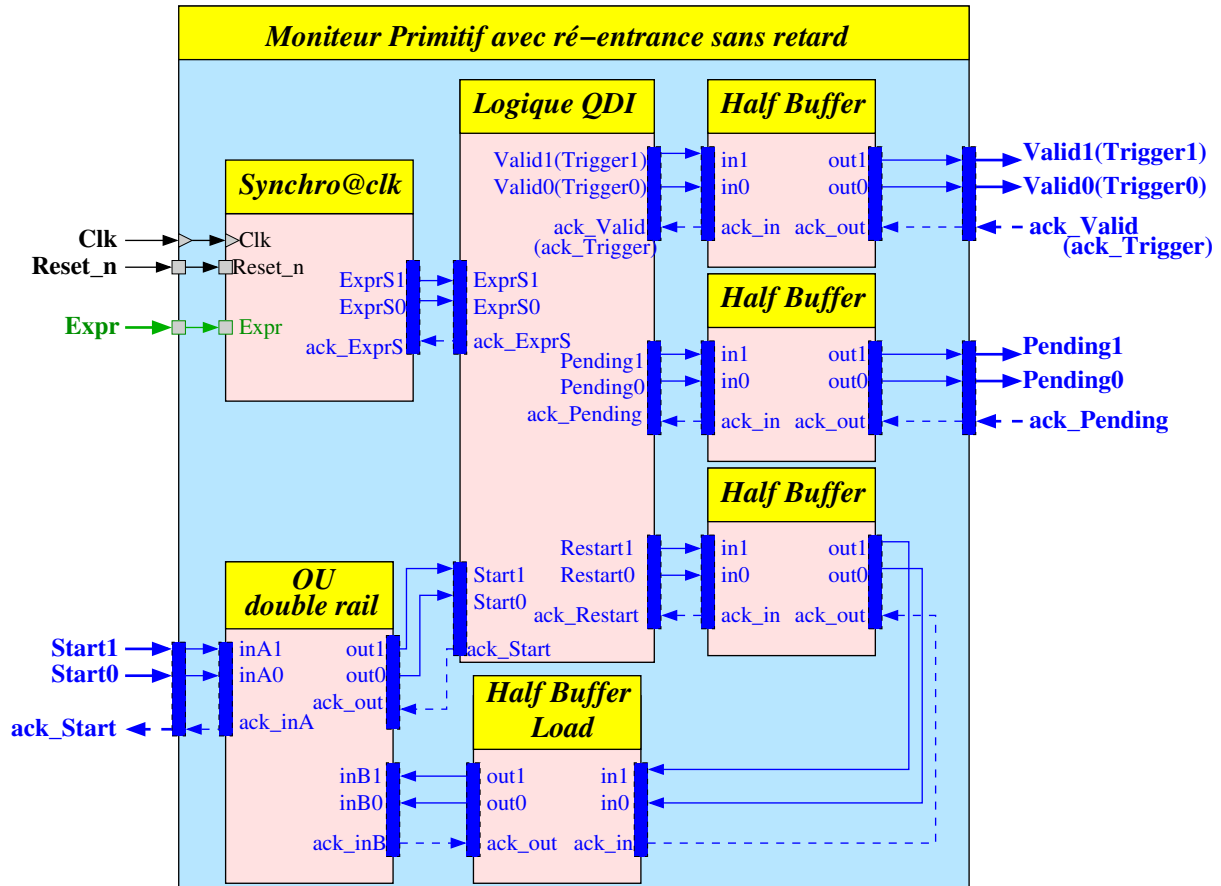


FIGURE A.4 – Moniteur primitif asynchrone avec ré-entrance et sans cycle de retard

A.2.1 Les moniteurs primitifs M_until

On présente les implémentations des parties "Logique QDI" pour chacune des versions de l'opérateur *until* (faible, fort, recouvrant, non recouvrant). Les propriétés vérifiées par cet opérateur sont du type *Expr until Cond*. Les moniteurs primitifs de la famille *until* évaluent *Cond* et en fonction de la valeur de *Cond*, valide ou invalide l'évaluation de la sous-propriété *Expr*. Dans PSL_{ss} , *Cond* est booléen et *Expr* peut être une FL ou un booléen. Les moniteurs primitifs de la famille *until* sont des connecteurs.

A.2.1.1 Les moniteurs primitifs M_until! et M_until

On appellera *M_until* et *M_until!* les moniteurs primitifs correspondant respectivement aux opérateurs *until* et *until!*. Ces opérateurs sont non recouvrants. Cela signifie que lorsque

que *Cond* est vraie, on ne se soucie pas de savoir si *Expr* est vraie ou non au même cycle. La partie "Logique QDI" du moniteur primitif *M_until!* doit implémenter les rendez-vous suivants entre les signaux *Start* et *Cond* :

- *Start0* & *CondS0* ou *Start0* & *CondS1* : le moniteur n'a pas à tenir compte de l'évaluation de *Cond* faite au front montant d'horloge précédent. La sous-propriété *Expr* n'a pas besoin d'être vérifiée. On génère donc une sortie *Trigger0*. De plus le moniteur *M_until!* n'étant pas actif car il a reçu une entrée *Start0*, le moniteur génère une sortie *Pending0*. Il n'y a pas besoin de relancer l'évaluation au prochain cycle donc le moniteur génère une sortie *Restart0*.
- *Start1* & *CondS0* : le moniteur doit tenir compte de l'évaluation de *Cond*. Comme on a *CondS0*, cela signifie que *Cond* était à '0' au front montant d'horloge précédent. La validité de *Expr* until! *Cond* dépend de la validité de *Expr* qui doit donc être vérifiée. Le moniteur génère une sortie *Trigger1* et comme il est en cours d'évaluation il génère aussi une sortie *Pending1*. Si *Expr* est effectivement vrai, il faudra ré-évaluer *Cond* au cycle suivant. Le moniteur génère donc un signal *Restart1*.
- *Start1* & *CondS1* : le moniteur doit tenir compte de l'évaluation de *Cond*. Comme on a *CondS1*, cela signifie que *Cond* était à '1' au front montant d'horloge précédent et que *Expr* until! *Cond* est vrai quelque soit la valeur de *Cond*. Le moniteur génère une sortie *Trigger0* et comme l'évaluation de *Expr* until! *Cond* est terminée, le moniteur génère aussi une sortie *Pending0*. Il n'y a pas besoin de réactiver le moniteur pour le prochain cycle, une sortie *Restart0* est générée.

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.5

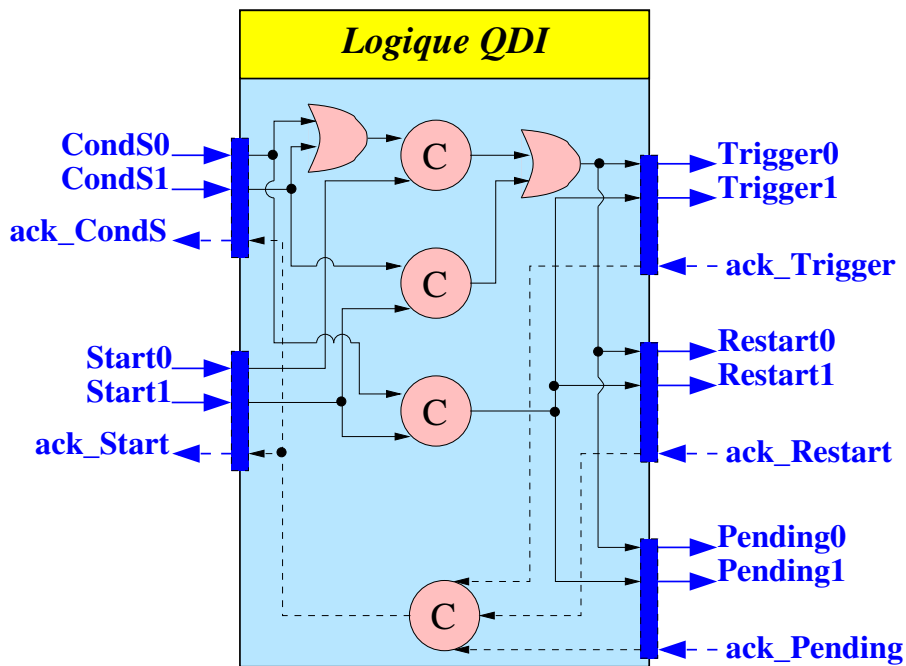


FIGURE A.5 – Implémentation de la partie "Logique QDI" du *M_until!*

L'implémentation de la version faible de cet opérateur, le *M_until*, est obtenue en supprimant le *Half Buffer* et les portes utilisées pour générer le signal *Pending* dans la partie "Logique QDI" qui est ensuite utilisée dans l'architecture générique présentée dans la partie 3.2.5 figure 3.14.

A.2.1.2 Les moniteurs primitifs $M_until!$ et M_until

On appellera M_until et $M_until!$ les moniteurs primitifs correspondant respectivement aux opérateurs $until$ et $until!$. Ces opérateurs sont recouvrants, cela signifie que lorsque $Cond$ est vrai, $Expr$ doit être vrai au même cycle pour que $Expr\ until!\ Cond$ soit vrai. On a donc les rendez vous suivants entre les signaux $Start$ et $Cond$ pour la partie "Logique QDI" du moniteur primitif $M_until!$:

- $Start0$ & $CondS0$ ou $Start0$ & $CondS1$: le moniteur n'a pas à tenir compte de l'évaluation de $Cond$ faite au front montant d'horloge précédent. La sous-propriété $Expr$ n'a pas besoin d'être vérifiée. On génère donc une sortie $Trigger0$. De plus le moniteur $M_until!$ n'étant pas actif car il a reçu une entrée $Start0$, le moniteur génère une sortie $Pending0$. Il n'y a pas besoin de relancer l'évaluation au prochain cycle donc le moniteur génère une sortie $Restart0$.
- $Start1$ & $CondS0$: le moniteur doit tenir compte de l'évaluation de $Cond$. Comme on a $CondS0$, cela signifie que $Cond$ était à '0' au front montant d'horloge précédent. La validité de $Expr\ until!\ Cond$ dépend de la validité de $Expr$ qui doit donc être vérifiée. Le moniteur génère une sortie $Trigger1$ et comme il est en cours d'évaluation il génère aussi une sortie $Pending1$. Si $Expr$ est effectivement vrai, il faudra ré-évaluer $Cond$ au cycle suivant. Le moniteur génère donc un signal $Restart1$.
- $Start1$ & $CondS1$: le moniteur doit tenir compte de l'évaluation de $Cond$. Comme on a $CondS1$, cela signifie que $Cond$ était à '1' au front montant d'horloge précédent et que $Expr\ until!\ Cond$ est vrai si $Cond$ est vrai à ce cycle. Le moniteur génère une sortie $Trigger1$ et comme l'évaluation de $Expr\ until!\ Cond$ est terminée, le moniteur génère aussi une sortie $Pending0$. Il n'y a pas besoin de réactiver le moniteur pour le prochain cycle, une sortie $Restart0$ est générée.

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.6

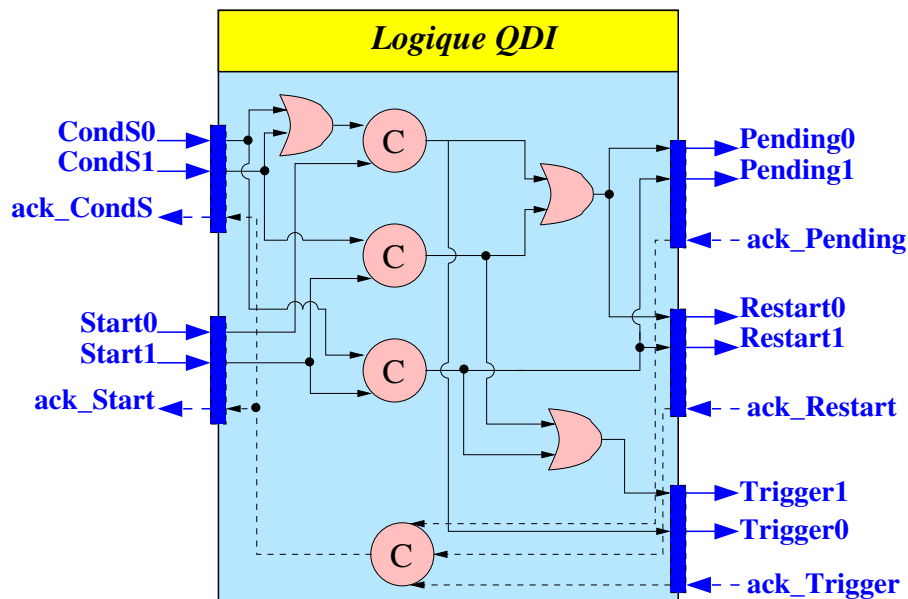


FIGURE A.6 – Implémentation de la partie "Logique QDI" du $M_until!$

L'implémentation de la version faible de cet opérateur, le M_until , est obtenue en supprimant les portes utilisées pour générer le signal $Pending$ dans la partie "Logique

QDI'' qui est ensuite utilisée dans l'architecture générique présentée dans la partie 3.2.5 figure 3.14.

A.2.2 Le moniteur primitif M_always

Ce moniteur primitif correspond à l'opérateur **always**. Dès l'instant où il est activé par une entrée *Start1*, il doit fournir un signal *Trigger1* à chaque cycle d'évaluation. Le **M_always** est un connecteur qui valide ou invalide l'évaluation de la sous-propriété qui peut être FL ou booléenne. Afin d'assurer la synchronisation, on utilise un *Synchro@clk* dont l'entrée *Expr* est connectée à la masse ou à l'alimentation. On a donc les rendez-vous suivants entre les signaux *Start* et *Expr* pour la partie "Logique QDI" du moniteur primitif **M_always** :

- *Start0* & *ExprS0* ou *Start0* & *ExprS1* : le moniteur n'est pas activé. L'évaluation de la sous-propriété ne doit pas être prise en compte, une sortie *Trigger0* est générée. De plus, comme ce moniteur n'a pas encore été activé, il n'y a pas besoin de ré-évaluer la sous-propriété au cycle suivant et donc le moniteur génère une sortie *Restart0*.
- *Start1* & *ExprS0* ou *Start1* & *ExprS1* : le moniteur est activé, L'évaluation de la sous-propriété devra être prise en compte à ce cycle (sortie *Trigger1*) et à tous les cycles suivants (sortie *Restart1*).

On obtient pour la partie "Logique QDI" du **M_always** l'implémentation présentée figure A.7

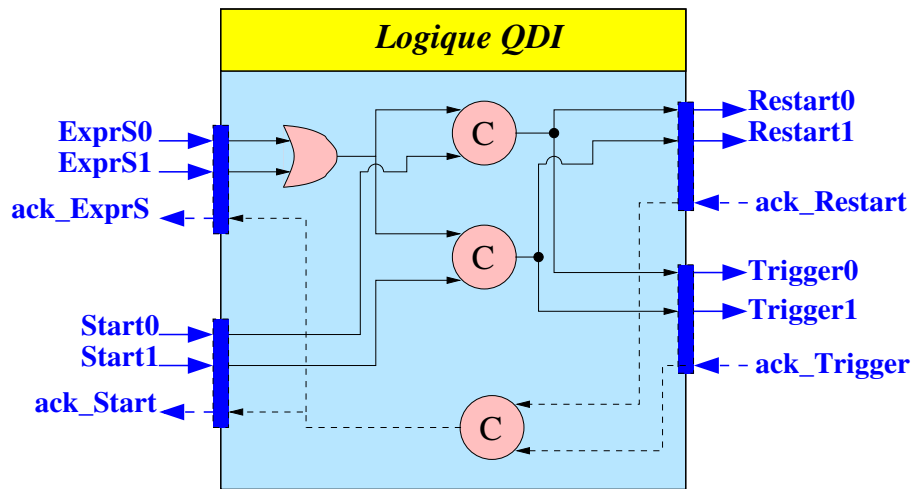


FIGURE A.7 – Implémentation de la partie "Logique QDI" du **M_always**

A.2.3 Le moniteur primitif M_never

Dans PSL_{ss}, l'opérateur **never** ne prend pour opérande qu'un booléen ou une séquence. Le travail de ce manuscrit ne portant pas sur les séquences, le moniteur primitif **M_never** correspondant ne pourra prendre comme opérande qu'un booléen et sera un observateur. Une fois qu'il a été activé, ce moniteur doit vérifier à chaque cycle que $Expr=0$. On a donc les rendez-vous suivants pour les signaux *Start* et *Expr* :

- *Start0* & *ExprS0* ou *Start0* & *ExprS1* : le moniteur n'est pas activé. L'évaluation de *Expr* ne doit pas être prise en compte et **never Expr** est considérée vraie, une sortie *Valid1* est générée. De plus, comme ce moniteur n'a pas encore été activé, il n'y a pas besoin de ré-évaluer la sous-propriété au cycle suivant et donc le moniteur génère une sortie *Restart0*.
- *Start1* & *ExprS1* : l'évaluation de *Expr* doit être prise en compte. Comme *Expr* était à '1' lors du précédent front montant d'horloge, **never Expr** est violée. Le moniteur génère une sortie *Valid0* et on n'a plus besoin de savoir ce qu'il se passe lors des prochains cycles (sortie *Restart0*).
- *Start1* & *ExprS0* : l'évaluation de *Expr* doit être prise en compte. Comme *Expr* était à '0' lors du précédent front montant d'horloge, **never Expr** est vraie pour ce cycle. Le moniteur génère une sortie *Valid1*. Il faut re-vérifier *Expr* au prochain cycle, le moniteur génère donc une sortie *Restart1*.

On obtient pour la partie "Logique QDI" du **M_never** l'implémentation présentée figure A.8

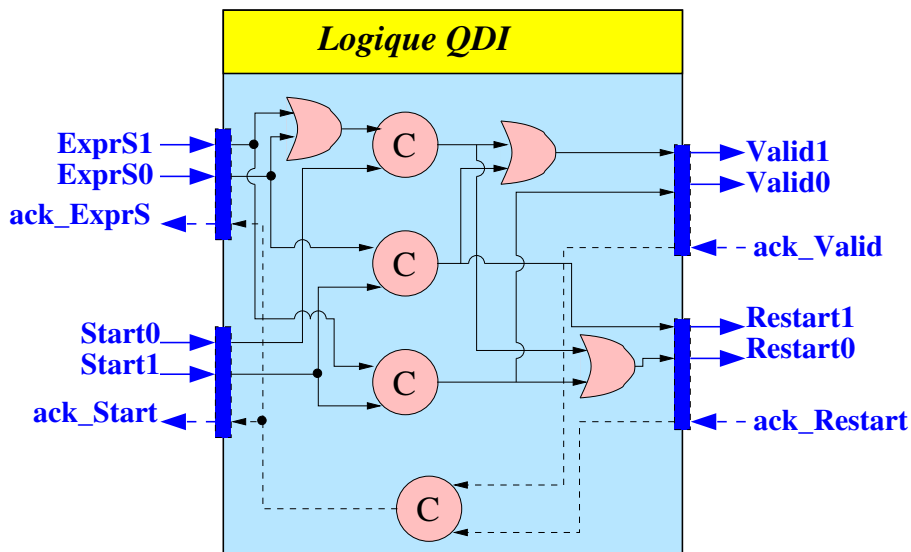


FIGURE A.8 – Implémentation de la partie "Logique QDI" du **M_never**

A.2.4 Le moniteur primitif M_eventually!

Dans PSL_{ss} , l'opérateur `eventually!` ne prend pour opérande qu'un booléen ou une séquence. Le travail de ce manuscrit ne portant pas sur les séquences, le moniteur primitif `M_eventually!` ne pourra prendre comme opérande qu'un booléen et sera un observateur. Une fois qu'il a été activé, ce moniteur doit vérifier *Expr* à chaque cycle. Si lors de la trace on rencontre *Expr*=`'1'`, `eventually! Expr` est vraie et reste vraie quelque soit la suite de la trace. On a donc les rendez vous suivants pour les signaux *Start* et *Expr* :

- *Start0* & *ExprS0* ou *Start0* & *ExprS1* : le moniteur n'est pas activé. L'évaluation de *Expr* ne doit pas être prise en compte et `eventually! Expr` est considérée vraie, une sortie *Valid1* est générée. De plus, comme ce moniteur n'a pas encore été activé, il n'y a pas besoin de ré-évaluer la sous-propriété au cycle suivant et donc le moniteur génère une sortie *Restart0*. La propriété n'est pas en cours d'évaluation, un signal *Pending0* est généré par le moniteur.
- *Start1* & *ExprS1* : l'évaluation de *Expr* doit être prise en compte. Comme *Expr* était à `'1'` lors du précédent front montant d'horloge, `eventually! Expr` est vraie. Le moniteur génère une sortie *Valid1* et on n'a plus besoin de savoir ce qu'il se passe lors des prochains cycles (sortie *Restart0*). Comme l'évaluation est terminée, le moniteur génère une sortie *Pending0*.
- *Start1* & *ExprS0* : l'évaluation de *Expr* doit être prise en compte. Comme *Expr* était à `'0'` lors du précédent front montant d'horloge, `eventually! Expr` est vraie pour ce cycle (sortie *Valid1*) et il faut vérifier *Expr* au front suivant (sortie *Restart1*). De plus l'évaluation est toujours en cours. Il y a aussi la sortie *Pending1* active.

On obtient pour la partie "Logique QDI" du `M_eventually!` l'implémentation présentée figure A.9

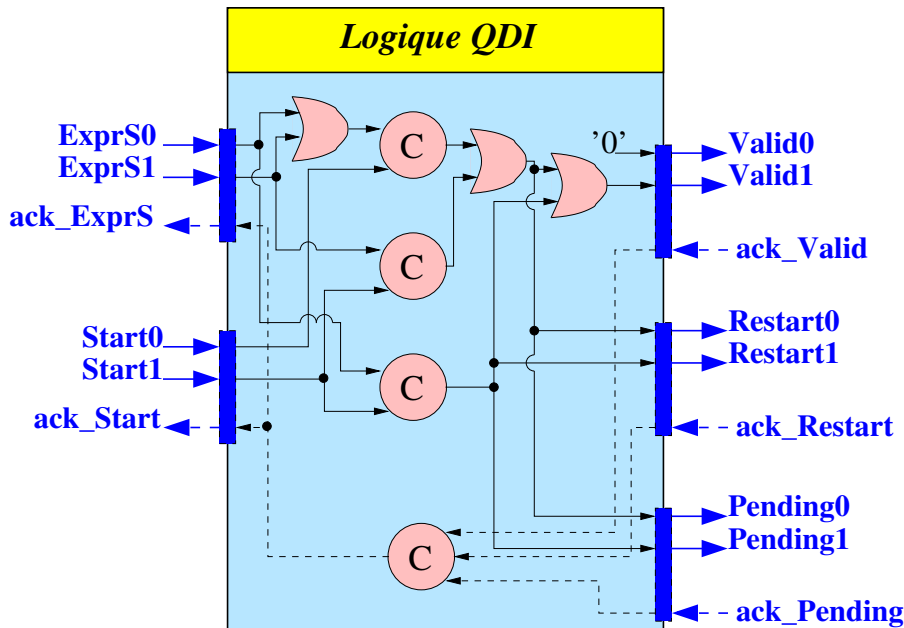


FIGURE A.9 – Implémentation de la partie "Logique QDI" du `M_eventually!`

Pending1). Il faudra re-vérifier *Expr* et *Cond* au prochain cycle, le moniteur génère donc une sortie *Restart1*.

- $Start1 \ \& \ CondS1 \ \& \ (ExprS1 + ExprS0)$: le moniteur doit tenir compte de l'évaluation de *Cond*. Comme on a *CondS1*, cela signifie que l'on a rencontré *Cond*='1' avant ou en même temps que *Expr*='1'. La propriété est donc violée (sortie *Valid0*) et l'évaluation terminée (sorties *Restart0* et *Pending0*).
- $Start1 \ \& \ CondS0 \ \& \ ExprS1$: on a bien *Expr* vrai strictement avant *Cond*. L'évaluation est donc terminée (sortie *Pending0* et *Restart0*) et la propriété est valide (sortie *Valid1*).

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.10. L'implémentation de la version faible de cet opérateur, le *M_before*, est obtenue en supprimant les portes utilisées pour générer le signal *Pending* dans la partie "Logique QDI". La partie "Logique QDI" est ensuite utilisée dans l'architecture générique présentée dans la partie 3.2.5 figure 3.14.

A.2.5.2 Les moniteurs *M_before!* et *M_before*

On appellera *M_before* et *M_before!* les moniteurs primitifs correspondant respectivement aux opérateurs *before* et *before!*. Ces opérateurs sont recouvrants. Cela signifie que lors du cycle où *Expr* est vraie, *Cond* peut être vrai au faux au même cycle. La partie "Logique QDI" du moniteur primitif *M_before!* doit implémenter les rendez-vous suivants entre les signaux *Start*, *Expr* et *Cond* :

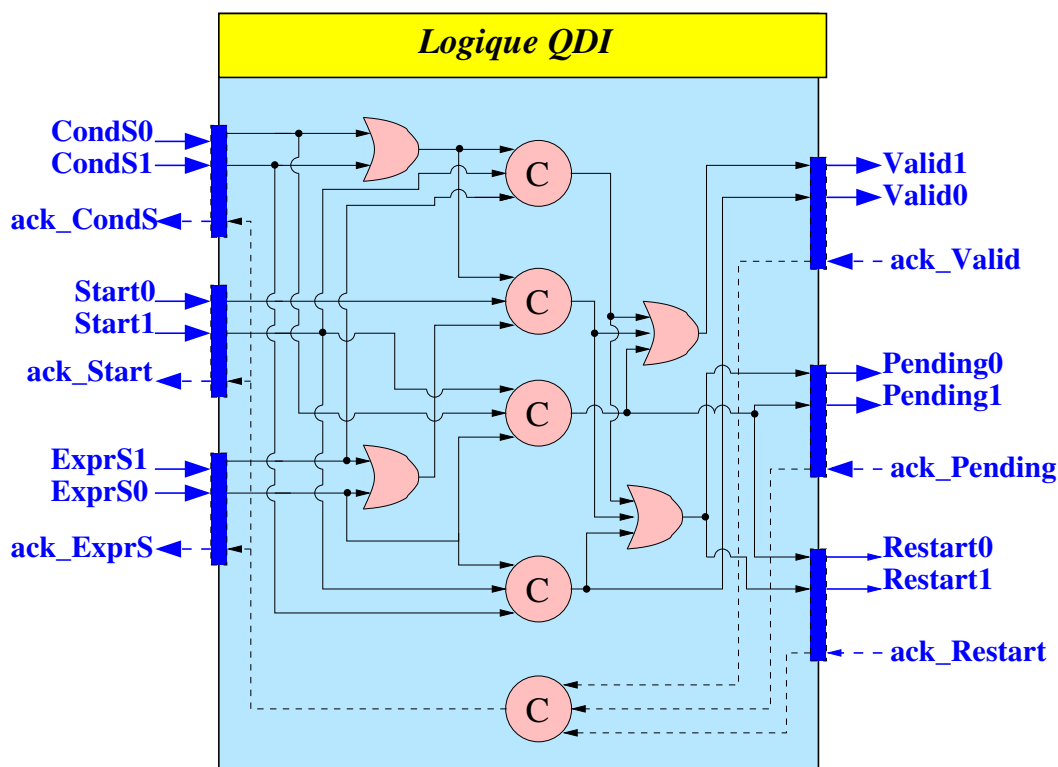


FIGURE A.11 – Implémentation de la partie "Logique QDI" du *M_before!*

- $Start0 \ \& \ (CondS0 + CondS1) \ \& \ (ExprS1 + ExprS0)$: le moniteur n'a pas à tenir compte des évaluations de *Cond* et *Expr* faites au front montant d'horloge précédent.

Le moniteur n'étant pas actif car il a reçu une entrée *Start0*, une sortie *Pending0* est générée. Il n'y a pas besoin de relancer l'évaluation au prochain cycle donc le moniteur génère une sortie *Restart0*. Enfin par défaut, comme il n'y a pas eu d'évaluation, la propriété est *Valid* et le moniteur fournit un signal *Valid1*.

- *Start1 & CondS0 & ExprS0* : le moniteur doit tenir compte de l'évaluation de *Cond* et *Expr*. Comme les deux opérandes étaient à '0' au précédent front d'horloge, la propriété n'est pas violée (sortie *Valid1*) et l'évaluation est toujours en cours (sortie *Pending1*). Il faudra re-vérifier *Expr* et *Cond* au prochain cycle, le moniteur génère donc une sortie *Restart1*.
- *Start1 & CondS1 & ExprS0* : le moniteur doit tenir compte de l'évaluation de *Cond*. Comme on a *CondS1*, cela signifie que l'on a rencontré *Cond*='1' avant *Expr*='1'. La propriété est donc violée (sortie *Valid0*) et l'évaluation terminée (sorties *Restart0* et *Pending0*).
- *Start1 & (CondS0 + CondS1) & ExprS1* : On a bien *Expr* vrai avant ou au même cycle que *Cond*. L'évaluation est donc terminée (sortie *Pending0* et *Restart0*) et la propriété est valide (sortie *Valid1*).

On obtient pour la partie "Logique QDI" l'implémentation présentée figure A.11. L'implémentation de la version faible de cet opérateur, le *M_before_*, est obtenue en supprimant les portes utilisées pour générer le signal *Pending* dans la partie "Logique QDI" qui est ensuite utilisée dans l'architecture générique présentée dans la partie 3.2.5 figure 3.14.

A.2.5.3 Les moniteurs primitifs $M_next_event(b)$ et $M_next_event!(b)$

On appelle $M_next_event(b)$ et $M_next_event!(b)$ les moniteurs primitifs correspondant respectivement aux opérateurs $next_event(b)$ et $next_event!(b)$. Une propriété PSL de type $next_event!(b)$ φ doit permettre de vérifier que lors de la première occurrence de b , φ est vrai aussi. Autrement dit pour un cycle d'évaluation donné, on vérifie si b est vrai. Si c'est le cas on devra demander aux moniteurs primitifs suivants de vérifier que φ est vrai aussi. Si b est faux, on doit demander au moniteur primitif de ré-évaluer b au prochain cycle. On a donc un signal *Restart* et un phénomène de ré-entrance. Ce moniteur primitif est un connecteur et le fonctionnement pour la partie "Logique QDI" est le suivant :

- *Start0* & (*b_S0* + *b_S1*) : le moniteur n'a pas à tenir compte de l'évaluation de b faite au front montant d'horloge précédent. Le moniteur $M_next_event!(b)$ n'étant pas actif car il a reçu une entrée *Start0*, une sortie *Pending0* est générée. Il n'y a pas besoin de relancer l'évaluation au prochain cycle donc le moniteur génère une sortie *Restart0*. Enfin il n'est pas nécessaire de tenir compte de l'évaluation de φ et le moniteur génère une sortie *Trigger0*.
- *Start1* & *b_S0* : le moniteur doit tenir compte de l'évaluation de b . Comme b était à '0' au précédent front d'horloge, il n'est pas nécessaire de tenir compte de l'évaluation de φ (sortie *Trigger0*). De plus on devra vérifier si b est vrai au prochain cycle. Il faut donc réactiver le moniteur à l'aide d'un signal *Restart1*. Enfin comme b est en cours d'évaluation, le moniteur est actif et le moniteur fournit une sortie *Pending1*.
- *Start1* & *b_S1* : le moniteur doit tenir compte de l'évaluation de b . Comme b était à '1' au précédent front d'horloge, il faut tenir compte de l'évaluation de φ (sortie *Trigger1*). De plus on a rencontré $b=1$, il n'est donc plus nécessaire de vérifier si b est vrai au prochain cycle (sortie *Restart0*). Le moniteur a terminé son évaluation et n'est plus actif (sortie *Pending0*).

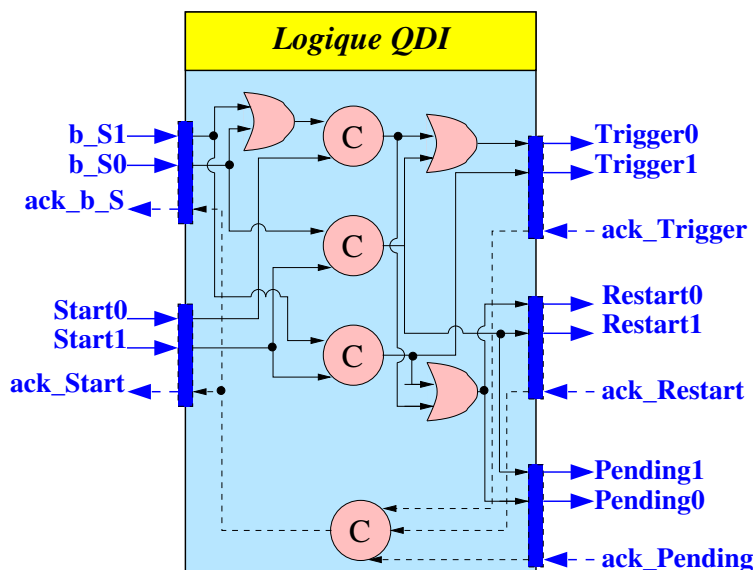


FIGURE A.12 – Implémentation de la partie "Logique QDI" du $M_next_event!(b)$

On donne figure A.12 l'implémentation détaillée de la partie "Logique QDI" qui est ensuite intégré dans l'architecture générique présentée figure A.4 pour la version forte de l'opérateur. L'obtention de la version faible du moniteur, le $M_next_event(b)$, se fait en

supprimant tout le matériel lié au signal de sortie *Pending* dans la partie "Logique QDI" qui est ensuite intégrée dans l'architecture présentée figure A.4.

A.3 Les moniteurs asynchrones introduisant un ou des cycles de retard et sans ré-entrance

Plusieurs opérateurs introduisent des cycles de retard dans l'évaluation d'une opérande ou d'une sous-propriété. Cependant seuls les opérateurs `next`, `next!`, `next_e![i..j]`, `next_e[i..j]`, `next_a![i..j]` et `next_a[i..j]` ne présentent pas de ré-entrance.

A.3.1 Les moniteurs `M_next`, `M_next!`, `M_next[K]` et `M_next![K]`

A.3.1.1 Les moniteurs `M_next` et `M_next!`

On appelle `M_next` et `M_next!` les moniteurs primitifs correspondant respectivement aux opérateurs `next` et `next!`. Ces moniteurs seront utiles dès que l'on voudra implémenter un opérateur introduisant un cycle de retard. La méthode d'implémentation de ces deux opérateurs est détaillée dans la partie 3.2.6.1 de ce manuscrit et n'est pas rappelée dans cette annexe. Les architectures de ces opérateurs sont rappelées figure A.13 pour le `M_next` et figure A.14 pour le `M_next!`.

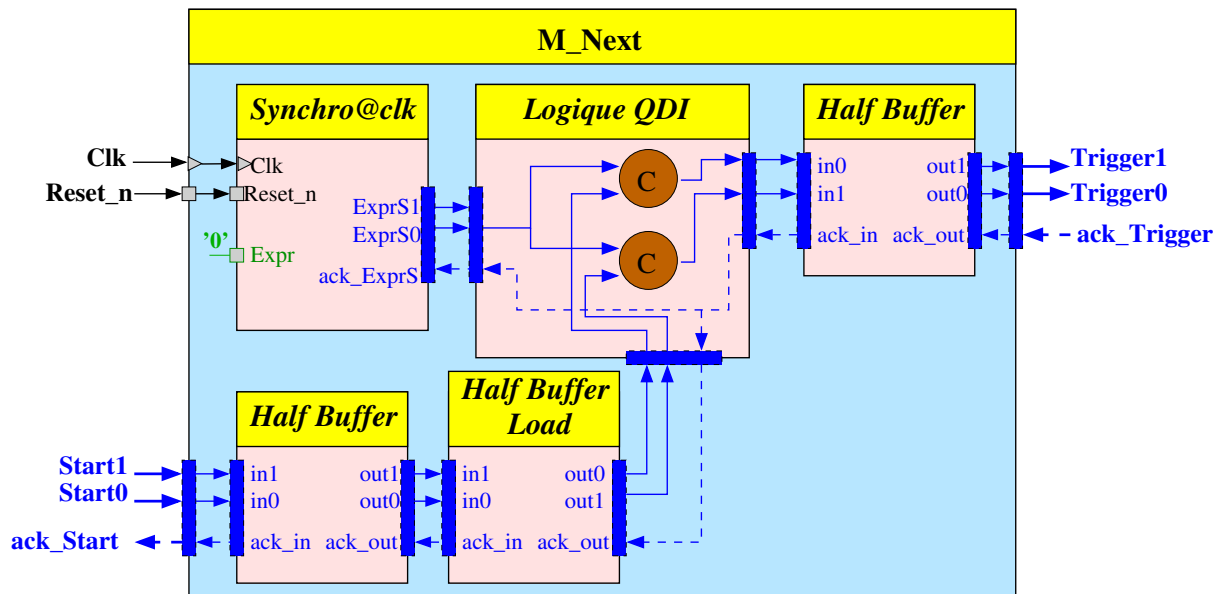
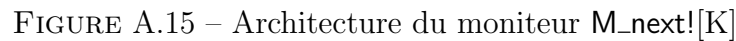


FIGURE A.13 – Architecture du moniteur primitif `M_next`

A.3.1.2 Les moniteurs `next[K]` et `next![K]`

Une fois les moniteurs primitifs `M_next` et `M_next!` construits, il est possible d'implémenter les moniteurs `M_next[K]` et `M_next![K]`. Pour implémenter le `M_next![K]`, il suffit de mettre `K` `M_next!` à la suite en connectant leurs sorties *Pending* grâce à des "OU" double rails comme le montre la figure A.15.

Il est très important de mettre des *Half Buffer*, noté HB sur la figure, après chaque opérateur double rail. Dans notre cas on fait un "OU" double rails et afin qu'il n'y ait pas de blocages, un *Half Buffer* est placé après chacun des blocs "OU". De manière plus générale, chaque fois que plusieurs moniteurs primitifs ayant des sorties *Pending* seront



Pour obtenir le moniteur `M_next[K]`, il suffit de connecter des moniteurs primitifs `M_next` à la place des moniteurs `M_next!`. La structure du moniteur `M_next[K]` est détaillée dans la partie 3.2.6.1, figure 3.17 de ce manuscrit.

A.3.2 Les moniteurs $M_next_a![i..j]$ et $M_next_a[i..j]$

La méthode de construction du moniteur $M_next_a[i..j]$, correspondant à l'opérateur $next_a[i..j]$, est détaillée dans la partie 3.2.6.2. On présente ici comment obtenir la version forte du moniteur, le $M_next_a![i..j]$. Le $M_next_a![i..j]$ doit fournir un signal *Pending1* dès qu'un de ces étages est activé, c'est-à-dire dès qu'un signal *Start1* est présent en entrée d'un des étages. Or dans le bloc M_next_a , on utilise un bloc M_next , l'utilisation d'un bloc $M_next!$ à la place permet d'obtenir un signal *Pending1* dès qu'un signal *Start1* est présent en entrée d'un étage $M_next_a!$. Le signal *Pending* est propagé sur le port de sortie *Pending* du $M_next_a!$ comme le montre la figure A.16.

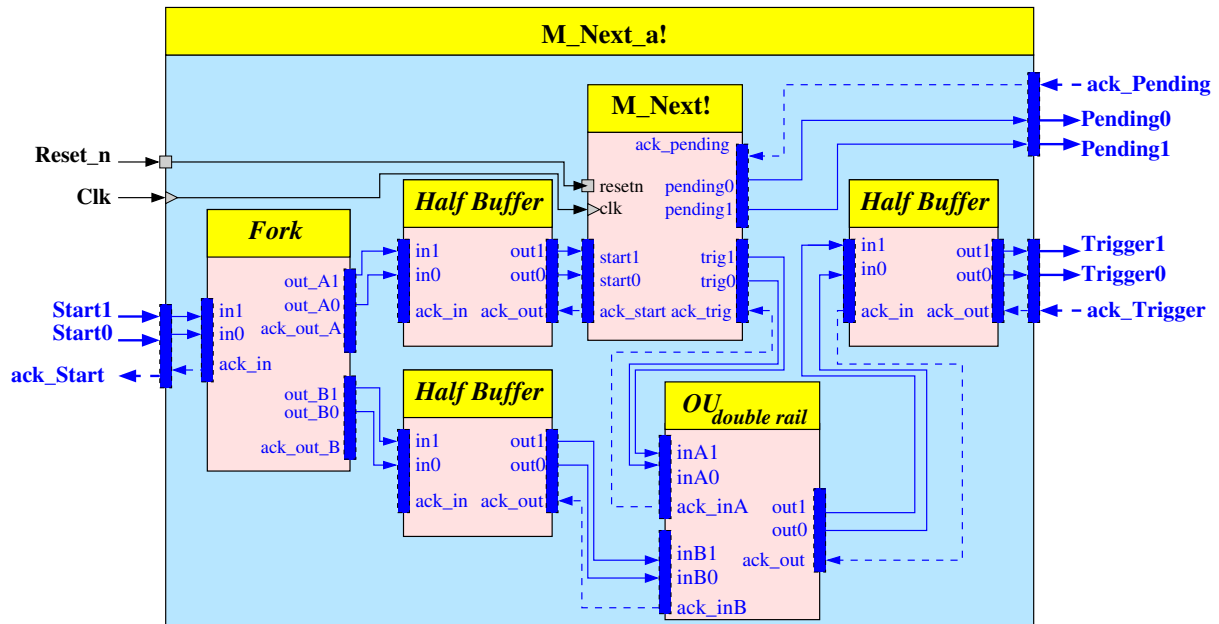


FIGURE A.16 – Architecture du moniteur primitif $M_next_a!$

Dans la structure du $M_next_a[i..j]$ présentée dans la partie 3.2.6.2, figure 3.20, on utilise un $M_next[i]$. Il faut désormais utiliser un $M_next![i]$ à la place du $M_next[i]$. Enfin il faut connecter les signaux *Pending* de chaque étage à l'aide de "OU" double rail et de *Half Buffer* comme cela est présenté figure A.15.

A.3.3 Les moniteurs $M_next_e[i..j]$ et $M_next_e! [i..j]$

Dans PSL_{ss} les opérateurs $next_e! [i..j]$ et $next_e [i..j]$ ne peuvent prendre pour opérande qu'un booléen. Les moniteurs matériels correspondants, respectivement $M_next_e! [i..j]$ et $M_next_e [i..j]$, seront donc des observateurs.

Pour que $next_e! [i..j] Expr$ soit vrai, il faut que $Expr$ soit vrai au moins une fois entre les cycles d'évaluation i et j inclus. Lors des cycles 0 à $i - 1$, il n'est nullement nécessaire de vérifier $Expr$. On utilise un moniteur $M_next! [i]$ pour se ramener au cycle d'évaluation i .

Lors du cycle i , l'étage actif du $next_e! [i..j]$ reçoit le signal $Start1$ ou $Start0$ en provenance du $next! [i]$. Si l'étage reçoit un signal $Start1$, il doit vérifier $Expr$ et introduire un cycle de retard pour que l'évaluation de $Expr$ par l'étage suivant ait lieu au cycle $i + 1$. Si l'étage reçoit un signal $Start0$, il n'a pas à vérifier $Expr$ et doit juste introduire un cycle de retard pour que l'évaluation de $Expr$ par l'étage suivant ait lieu au cycle $i + 1$. Ce bloc possède la même architecture que celle présentée figure A.1 où l'on rajoute un moniteur primitif M_next afin de retarder les signaux $Trigger1$ et $Trigger0$ d'un cycle entre chaque étage comme le montre la figure A.17. On appellera $M_next_e!$ le bloc matériel de la figure A.17 correspondant à un étage.

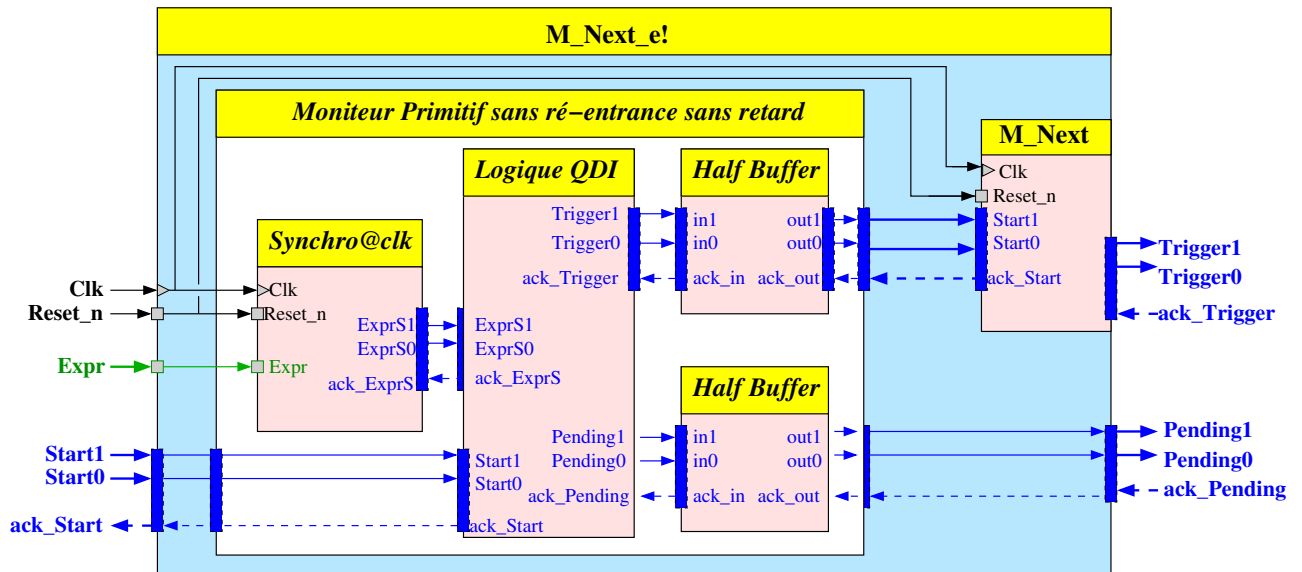


FIGURE A.17 – Architecture du moniteur primitif $M_next_e!$

On a le fonctionnement suivant pour la partie "Logique QDI" :

- $Start0 \& ExprS0$ ou $Start0 \& ExprS1$: l'étage i n'est pas activé. L'évaluation de $Expr$ ne doit pas être prise en compte et pour l'instant l'évaluation de $next_e! [i..j] Expr$ est considérée vraie. La propriété n'est pas en cours d'évaluation, un signal $Pending0$ est généré par le moniteur. On n'a pas non plus besoin d'activer l'étage suivant, une sortie $Trigger0$ est générée.
- $Start1 \& ExprS1$: l'évaluation de $Expr$ doit être prise en compte. Comme $Expr$ était à '1' lors du précédent front montant d'horloge, l'évaluation de $next_e! [i..j] Expr$ est vraie et terminée. On n'a plus besoin de savoir ce qu'il se passe lors des prochains cycles (sortie $Trigger0$). Comme l'évaluation est terminée, le moniteur génère une sortie $Pending0$.

- $Start1 \ \& \ ExprS0$: l'évaluation de $Expr$ doit être prise en compte. Comme $Expr$ était à '0' lors du précédent front montant d'horloge, il faut activée l'étage suivant afin de vérifier $Expr$ au front suivant (sortie $Trigger1$). De plus l'évaluation est toujours en cours. Il y a aussi la sortie $Pending1$ active.

L'implémentation de la partie "Logique QDI" du $M_next_e!$, correspondant à un étage du $M_next_e![i..j]$, est donnée figure A.18.

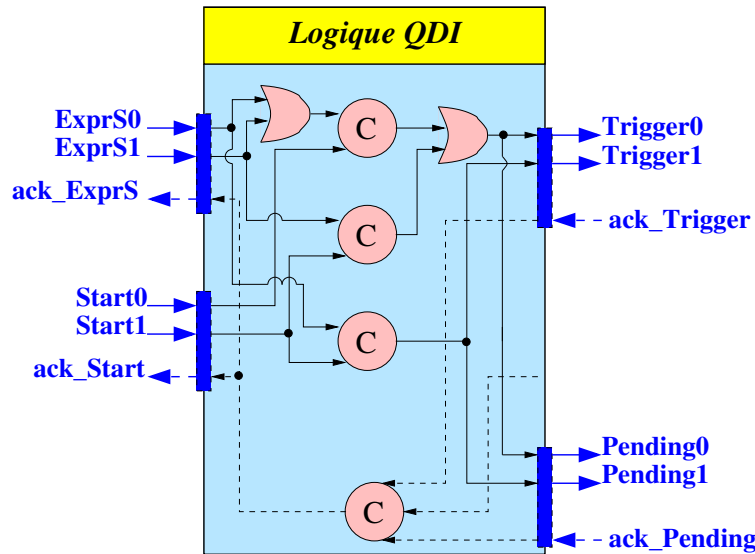


FIGURE A.18 – Implémentation de la partie "Logique QDI" du $M_next_e!$

Afin que le moniteur soit effectivement un observateur, il doit générer une sortie *Valid*. Pour assurer cette fonctionnalité, le dernier étage (étage j) du $M_next_e![i..j]$ est différent des étages i à $j - 1$. De plus comme cet étage est le dernier, il ne possède pas de sortie $Trigger$ et donc pas de moniteur primitif M_next pour retarder d'un cycle les sorties $Trigger$. On a le fonctionnement suivant pour la partie "Logique QDI" :

- $Start0 \ \& \ ExprS0$ ou $Start0 \ \& \ ExprS1$: l'étage i n'est pas activé. L'évaluation de $Expr$ ne doit pas être prise en compte, l'évaluation de $next_e![i..j] \ Expr$ est considérée vraie.
- $Start1 \ \& \ ExprS1$: l'évaluation de $Expr$ doit être prise en compte. Comme $Expr$ était à '1' lors du précédent front montant d'horloge, l'évaluation de $next_e![i..j] \ Expr$ est vraie et terminée.
- $Start1 \ \& \ ExprS0$: l'évaluation de $Expr$ doit être prise en compte. $Expr$ était à '0' lors du précédent front montant d'horloge et la présence d'un signal $Start1$ en entrée de cet étage signifie que $Expr='1'$ n'a jamais été rencontrée lors des cycles i à $j - 1$. L'évaluation de $next_e![i..j] \ Expr$ est donc terminée et fausse (sortie $Valid0$).

On remarque qu'il n'y a pas de signal $Pending$ pour le dernier étage. En effet, quelle que soit la valeur de $Expr$ au cycle j , correspondant en matériel à l'étage j , l'évaluation de $next_e![i..j] \ Expr$ est terminée. L'état *Pending* ou non de la propriété est donnée par le "OU" logique des sorties $Pending$ de chaque étage $M_next_e!$ et de la sortie du $M_next![K]$. L'interconnexion des différents étages est donnée figure A.19. De plus on remarque que le comportement spécifié pour l'étage j n'est ni plus ni moins que celui du M_signal . On peut donc réutiliser ce dernier.

A.4 Les moniteurs asynchrones introduisant un ou des cycles de retard et avec ré-entrance

Tous comme les moniteurs correspondants aux opérateurs PSL `next`, `next!`, `next_e![i..j]`, `next_e[i..j]`, `next_a![i..j]` et `next_a[i..j]`, les moniteurs primitifs des opérateurs PSL `next_event_e(b)[K]`, `next_event_a(b)[K]` et `next_event(b)[K]` introduisent un ou plusieurs cycles de retard dans l'évaluation. Cependant, ces opérateurs introduisent de surcroît une ré-entrance. Concernant les moniteurs `M_next_event(b)[K]` et `M_next_event!(b)[K]`, leurs implémentations sont présentées dans la partie 3.2.6.3 de ce manuscrit.

A.4.1 Les moniteurs primitifs `M_next_event_a!` et `M_next_event_a`

Évaluer une propriété du type `next_event_a![K..L]  $\varphi$` signifie que l'on doit vérifier φ de la Kième à la Lième occurrence de `b`. Cet opérateur est proche de l'opérateur `next_a![K..L]` (c.f. partie A.3.2) à une différence près : dans l'évaluation de `next_a![K..L]  $\varphi$` , on regarde si φ est vrai du Kième au Lième cycle d'évaluation et non pas lors des occurrences de `b`. On s'intéresse aux évaluations ayant lieu à partir de la Kième occurrence de `b`. Pour arriver à ce cycle d'évaluation, on utilise le moniteur matériel `M_next_event!(b)[K]`. On est donc à la Kième occurrence de `b`, l'étage actif doit informer la partie du moniteur correspondant à l'évaluation de φ que l'on doit prendre en compte l'évaluation de φ . Pour cela on reprend la structure du `M_next_a!` (c.f. figure A.16) qui permet de réaliser la duplication du signal *Start* et de le transmettre aux moniteurs primitifs suivants pour prendre en compte l'évaluation de φ . Cependant, dans le `M_next_a![K..L]`, le prochain étage est activé un cycle d'évaluation plus tard grâce à un `M_next!`. Dans le cas du `M_next_event_a![K..L]`, on veut que l'étage suivant soit activé à la prochaine occurrence de `b`. Il suffit donc de remplacer le bloc `M_next!` par un `M_next_event!` pour obtenir le comportement voulu. On obtient ainsi le bloc `M_next_event_a!` de la figure A.20. La ré-entrance de cet opérateur est incluse dans le bloc `M_next_event!`.

Pour obtenir l'implémentation complète du `M_next_event_a![K..L]` il faut réaliser les étapes suivantes :

- On génère un moniteur `M_next_event!(b)[K]` pour se ramener à la Kième occurrence de `b`.
- Si $K=L$, on a terminé.
- Sinon, pour les étages $K+1$ à L on génère des blocs `M_next_event_a!`.
- Enfin, on connecte les signaux *Pending* de chaque étage `M_next_event_a!` et du `M_next_event!(b)[K]` à l'aide de "OU" double rails et de *Half Buffer*.

On obtient ainsi l'implémentation présentée figure A.21. Si on souhaite implémenter la version faible de l'opérateur, on utilisera le bloc `M_next_eventK` à la place du `M_next_event!K` dans le `M_next_event_a!` afin d'obtenir le `M_next_event_a`, ainsi il n'y a plus de sortie *Pending*.

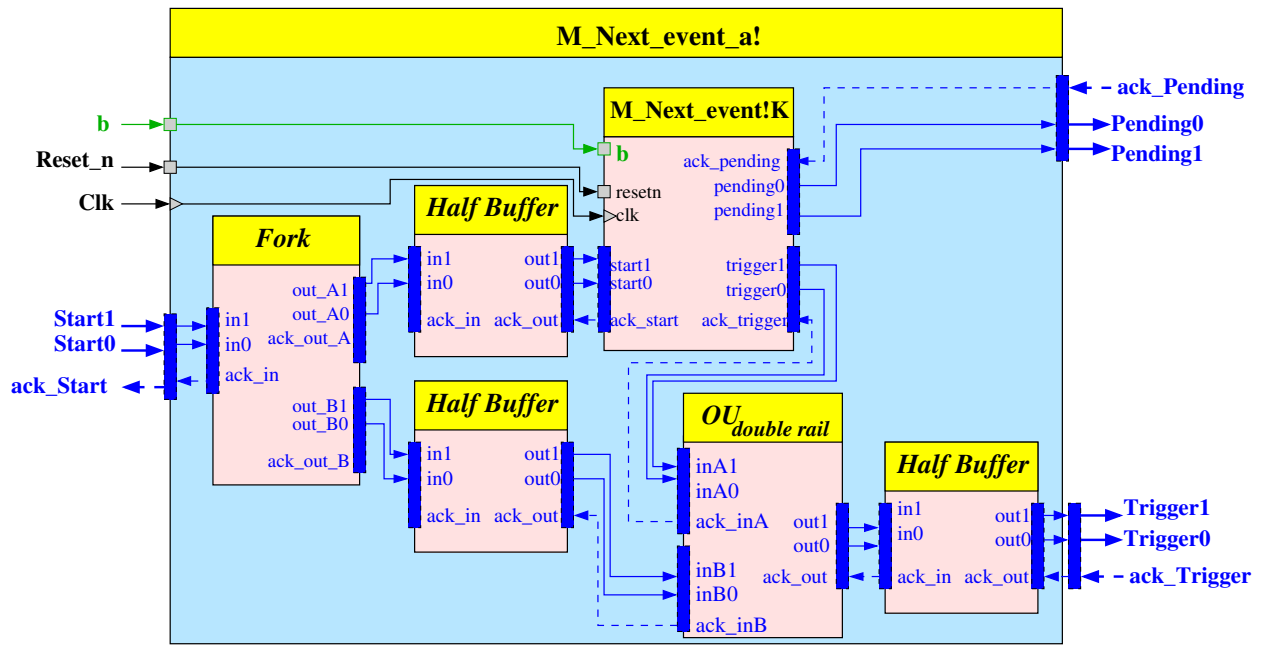


FIGURE A.20 – Implémentation du bloc matériel **M_next_event_a!**

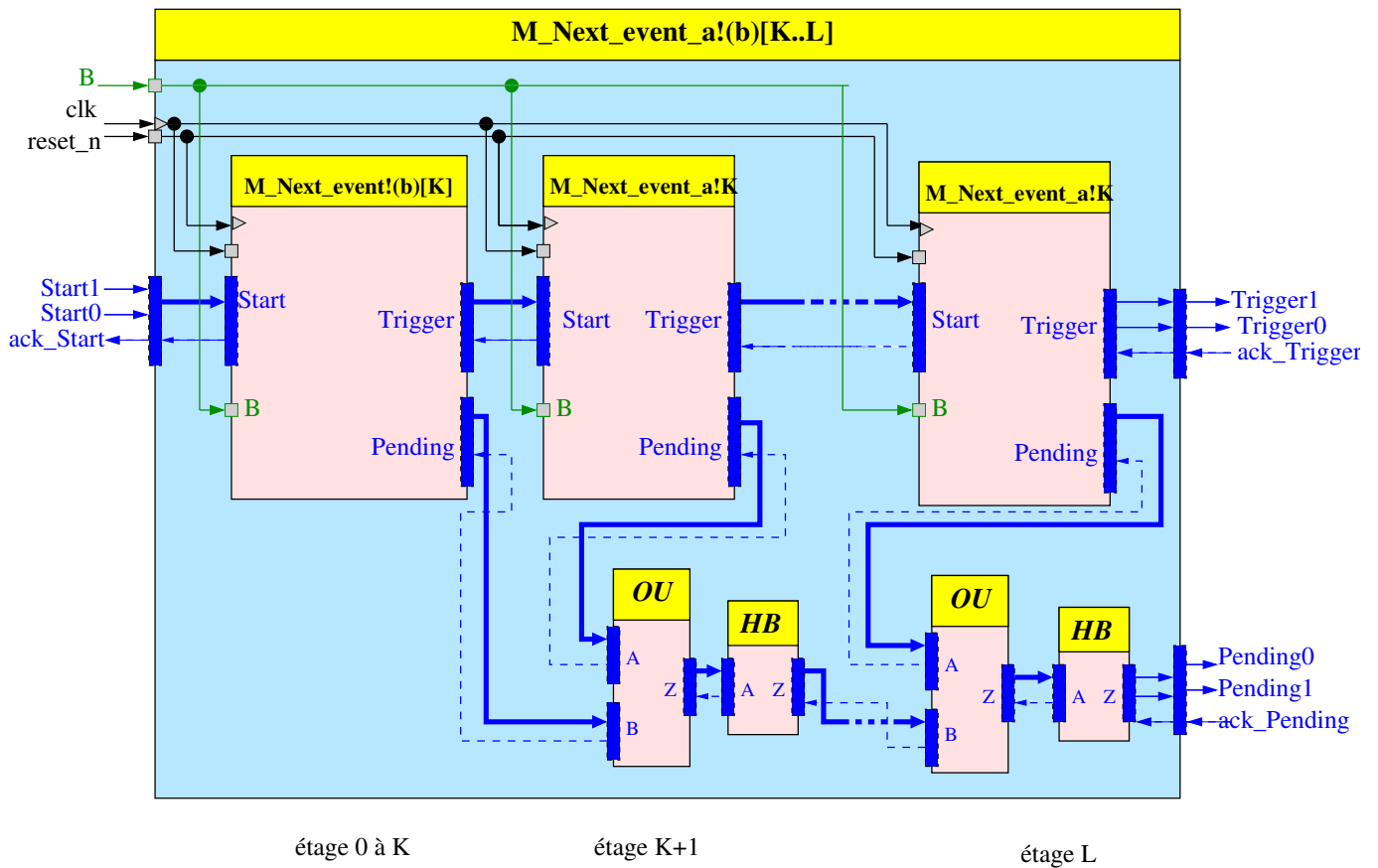


FIGURE A.21 – Implémentation complète du **M_next_event_a!(b)[K..L]**

A.4.2 Les moniteurs primitifs $M_next_event_e!$ et $M_next_event_e$

Dans PSL_{ss} , l'opérande d'un opérateur $next_event_e!(b)[K..L]$ est un booléen. Le moniteur $M_next_event_e(b)[K..L]$ correspondant est donc un observateur. Pour construire le moniteur primitif correspondant à l'opérateur $PSL\ next_event_a!(b)[K..L]$ on a réutilisé la structure du $M_next_a![K..L]$, on va procéder de la même façon pour le moniteur $M_next_event_e!(b)[K..L]$ en s'inspirant de la structure du $M_next_e![K..L]$ (c.f. figures A.17 et A.19, partie A.3.3).

Une propriété du type $next_event_e!(b)[K..L]\ \varphi$ est vraie si on rencontre au moins une fois φ vrai de la K ième à la L ième occurrence de b . Pour se ramener au cycle d'évaluation correspondant à la K ième occurrence de b , on utilise un moniteur primitif $M_next_event!(b)[K]$. Lorsque l'on est à la K ième occurrence de b , le bloc $M_next_event!(b)[K]$ va générer une sortie *Trigger1* pour l'étage suivant. L'étage suivant a donc un signal d'entrée *Start1* et doit vérifier φ . Si φ est faux, il faut aller à la prochaine occurrence de b pour vérifier à nouveau φ . Or il existe déjà un bloc permettant de passer d'une occurrence de b à la prochaine, c'est le bloc $M_next_event!K$ présenté dans la partie 3.2.6.3, figure 3.21 de ce manuscrit. La ré-entrance est incluse dans le bloc matériel $M_next_event!K$.

On a donc la même architecture pour un étage du moniteur $M_next_e![I..J]$ et pour un étage du moniteur $M_next_event_e!(b)[K..L]$. La différence réside dans l'utilisation d'un bloc $M_next_event!K$ dans le second cas plutôt qu'un simple bloc M_next . Comme on avait nommé $M_next_e!$ un étage du $next_e![I..J]$, on appelle $M_next_event_e!$ le bloc correspondant à un étage du $M_next_event_e!(b)[K..L]$ présenté figure A.22. Pour générer le signal *Pending* du $M_next_event_e!K$, il faut faire le "OU" logique entre les signaux *Pending* provenant du $M_next_event!K$ et ceux provenant de la partie "Logique QDI".

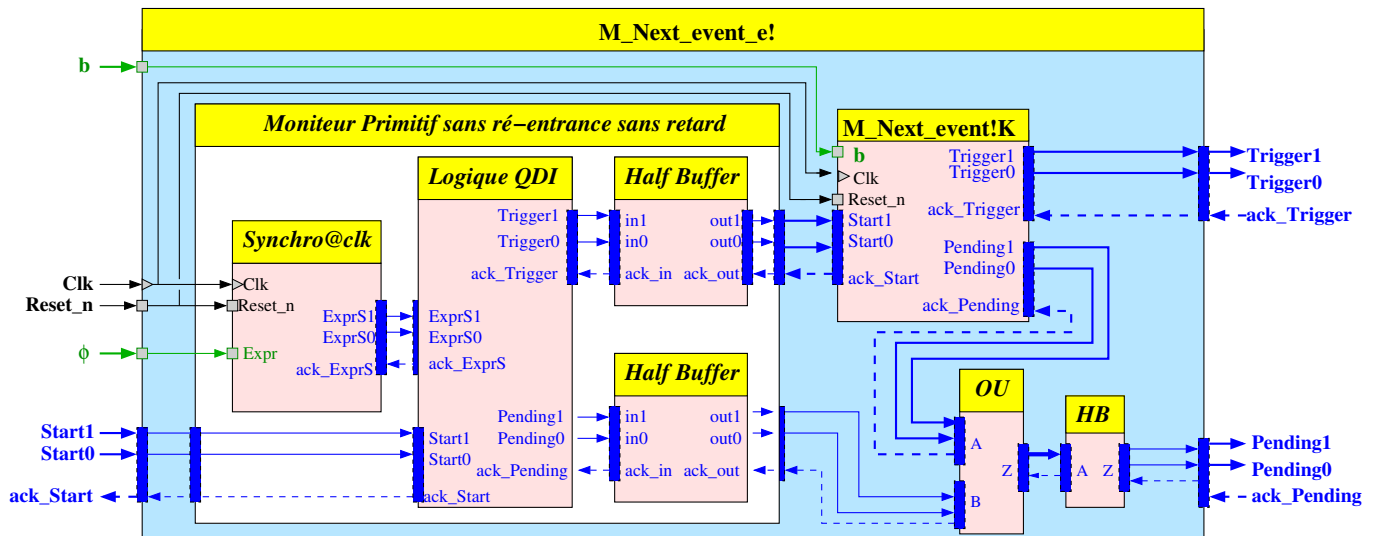


FIGURE A.22 – Implémentation du bloc $M_next_event_e!$

L'implémentation de partie "logique QDI" du $M_next_event_e!$ doit permettre le comportement suivant :

- $Start0 \ \& \ (\varphi_1 + \varphi_0)$: on a une entrée *Start0*, il n'est pas nécessaire de prendre en compte l'évaluation de φ et il n'est pas nécessaire de passer à la prochaine occurrence de b . Les sorties *Pending0* et *Trigger0* sont donc générées.

- $Start1 \ \& \ \varphi0$: on n'a pas rencontré φ vrai pour la K ième occurrence de b , on doit donc aller à la $K+1$ ième occurrence de b en activant le bloc $M_next_event!K$. Les sorties générées sont donc *Pending1* et *Trigger1*.
- $Start1 \ \& \ \varphi1$: on a φ vrai lors de la K ième occurrence de b , il n'est donc pas nécessaire de vérifier la valeur de φ pour les occurrences $K+1$ à L de b . L'évaluation de $next_event_e!(b)[K..L] \ \varphi$ est terminée. Les sorties générées sont donc *Pending0* et *Trigger0*.

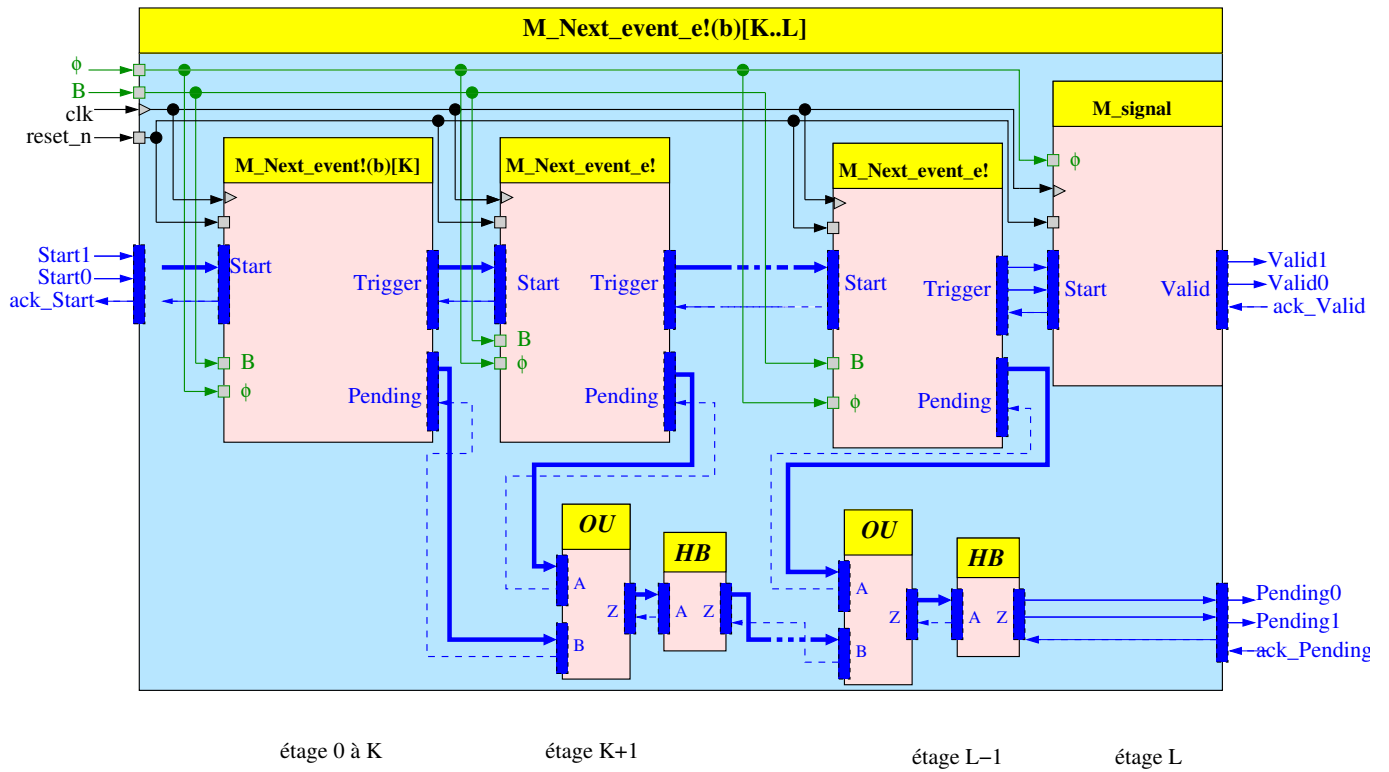
On remarque que le comportement de la partie "Logique QDI" du bloc *moniteur primitif sans ré-entrance sans retard* du $M_next_event_e!$ (c.f. figure A.22) est exactement le même que celui spécifié pour la partie "Logique QDI" du bloc *moniteur primitif sans ré-entrance sans retard* du $M_next_e!$. On peut donc réutiliser directement le bloc $M_next_e!$ dans l'implémentation du $M_next_event_e!$.

Comme pour le $M_next_e![I..J]$, le dernier étage doit permettre de générer les sorties *Valid1* et *Valid0*. Ce dernier étage reçoit un signal *Start1* uniquement quand les étages précédents correspondant aux occurrences K à $L-1$ de b n'ont jamais rencontrés φ vrai. Dans tous les autres cas, il reçoit un signal *Start0*. De plus le signal *Start1* provenant du $next_event!K$ de l'étage précédent, cela signifie que l'on est actuellement au cycle d'évaluation correspondant à la L ième occurrence de b . Dans le cas où le dernier étage a reçu un signal *Start1* et que φ est faux, cela signifie que l'évaluation de $next_event_e!(b)[K..L] \ \varphi$ est terminée et fausse car on n'a jamais rencontré φ vrai lors d'une occurrence de b , on a alors une sortie *Valid0*. Dans tous les autres cas, la sortie sera *Valid1*. On remarque que ce comportement est celui d'un M_signal évaluant φ .

On a donc l'architecture complète d'un moniteur primitif $M_next_event_e!(b)[K..L]$ (c.f. figure A.23) en respectant les étapes suivantes :

- On génère un $M_next_event!(b)[K]$
- Si $K=L$ on connecte ensuite un M_signal prenant φ comme opérande
- Sinon on génère des $M_next_event_e!$ pour les étages $K+1$ à L et enfin un M_signal pour le dernier étage.

Pour obtenir la version faible de cet opérateur, il suffit d'utiliser une version faible du bloc $M_next_event!K$, enlever la sortie *Pending* sur la partie "logique QDI". Cela permet d'obtenir le moniteur $M_next_event_e(b)[K..L]$ correspondant à l'opérateur faible $next_event_e(b)[K..L]$.

FIGURE A.23 – Implémentation complète du $M_next_event_e!(b)[K..L]$

A.5 Les moniteurs asynchrones particuliers

Certaines implémentations de moniteurs peuvent être simplifiées suivant que les opérandes soient FLs ou booléennes. De plus certains blocs peuvent être très utiles dans l'étape de connexion automatique des moniteurs primitifs réalisée par HORUS. Enfin certains moniteurs sont construits en réutilisant des moniteurs primitifs existants et des règles de ré-écriture. Cette partie présente ces divers blocs un peu particuliers.

A.5.1 Le moniteur primitif asynchrone M_imply_b

Le moniteur `M_imply` présenté dans la partie 3.2.4.2 de ce manuscrit est utilisé pour les propriétés PSL de type $Cond \rightarrow Expr$ où $Expr$ est une FL. Dans les cas où les deux opérandes sont booléens, il existe une façon plus simple et moins coûteuse en matériel. En effet, on sait que :

$$\begin{aligned} Cond \rightarrow Expr \\ \Leftrightarrow \\ \neg Cond \vee Expr \end{aligned}$$

Il suffit donc d'implémenter l'expression booléenne $\neg Cond \vee Expr$ suivie d'un `M_signal` comme l'illustre la figure A.24.

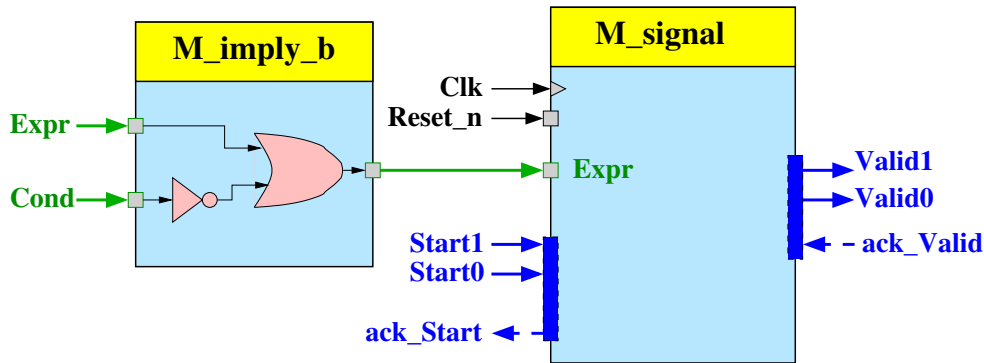


FIGURE A.24 – Architecture du moniteur primitif `M_imply_b`

A.5.2 Le moniteur primitif asynchrone M_and

Lorsque l'on souhaite implémenter une propriété de type *Expr* and *Cond*, deux façons de faire sont envisageables suivant que *Cond* et *Expr* soient deux booléens ou non. Si les deux opérandes sont booléens, on utilise une simple porte "ET" dont la sortie est connectée à un *M_signal*. Si au moins un des opérandes est une FL, il suffit d'implémenter le moniteur asynchrone correspondant à l'évaluation de *Expr* et celui correspondant à l'évaluation de *Cond* puis de connecter les sorties *Valid* des deux moniteurs à l'aide du bloc *M_and* de la figure A.25.

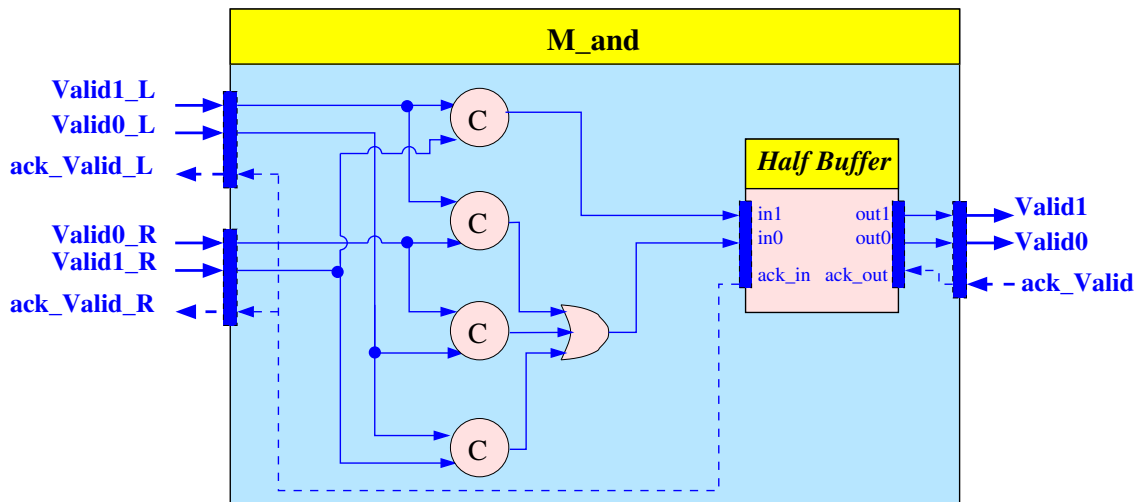


FIGURE A.25 – Architecture du moniteur primitif M_and

Dans le cas où les moniteurs de *Expr* et/ou *Cond* ont des sorties *Pending*, les signaux *Pending* des deux moniteurs sont connectés à l'aide d'un *Or_pending* (c.f. figure A.26).

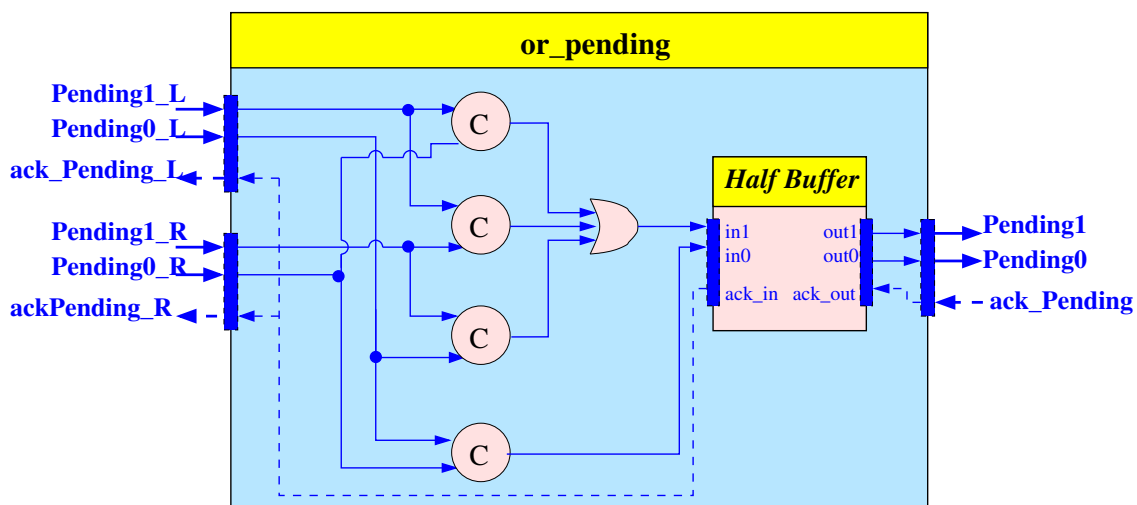


FIGURE A.26 – Architecture du bloc Or_pending

Les blocs *M_and* et *Or_pending* ne sont pas des moniteurs à proprement parler, ce sont tout simplement des blocs implémentant les fonctions "AND" et "OU" pour des signaux double rails avec de surcroît un *Half Buffer* afin d'empêcher tout deadlock. La connexion

complète de *Expr* and *Cond* est illustrée par la figure A.27. Le bloc *Or_pending* est surtout utile pour simplifier la connexion automatique des moniteurs dans HORUS.

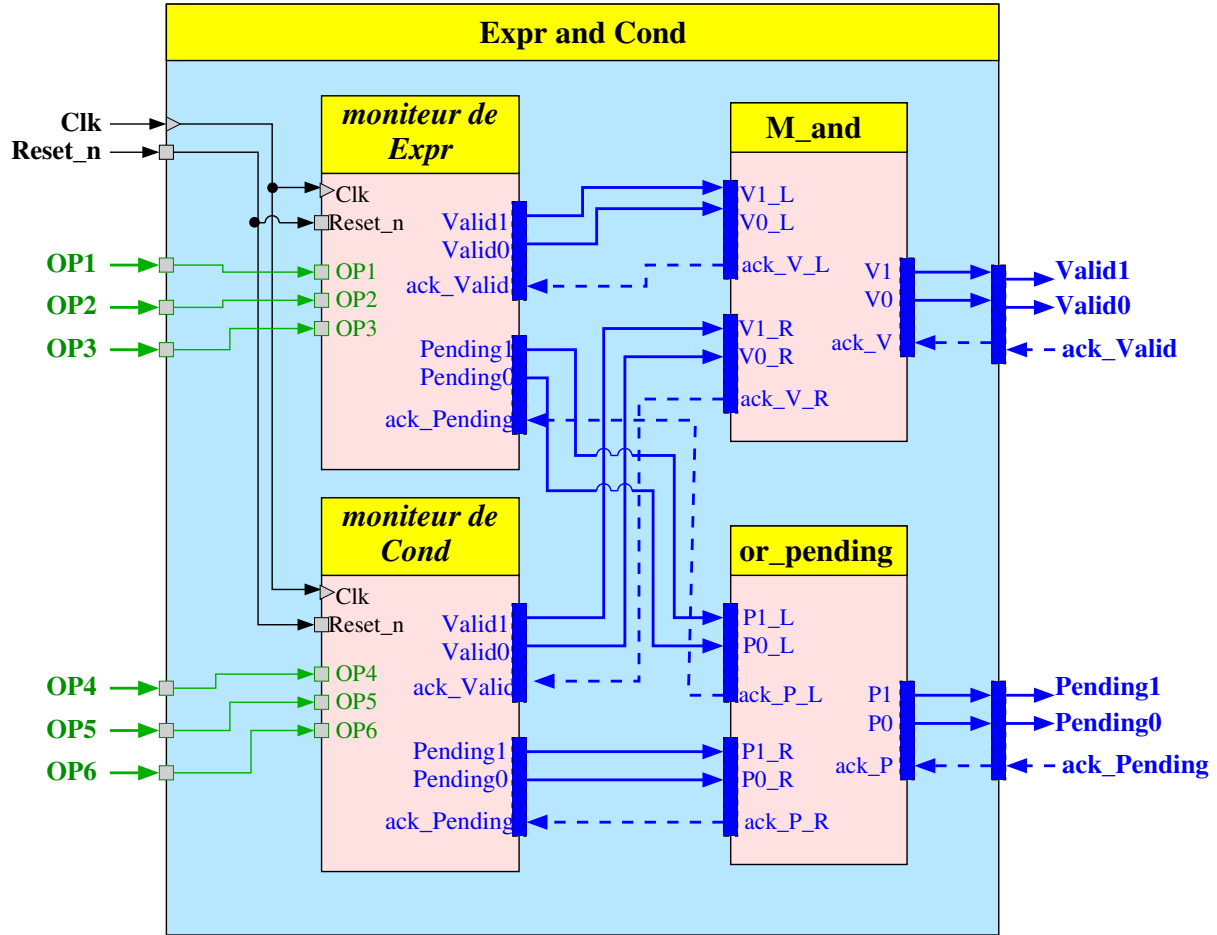


FIGURE A.27 – Architecture d'un moniteur asynchrone du type *Expr* and *Cond*

A.5.3 Le moniteur primitif asynchrone M_or

Dans PSL_{ss} , pour les propriétés du type $Expr$ or $Cond$, $Expr$ est nécessairement un opérande booléen. De plus on sait :

$$\begin{aligned} Cond \vee Expr &\leftrightarrow \\ \neg Cond \rightarrow Expr \end{aligned}$$

On peut donc ré-utiliser le moniteur primitif M_imply comme dans la figure A.28 afin d'implémenter $Expr$ or $Cond$.

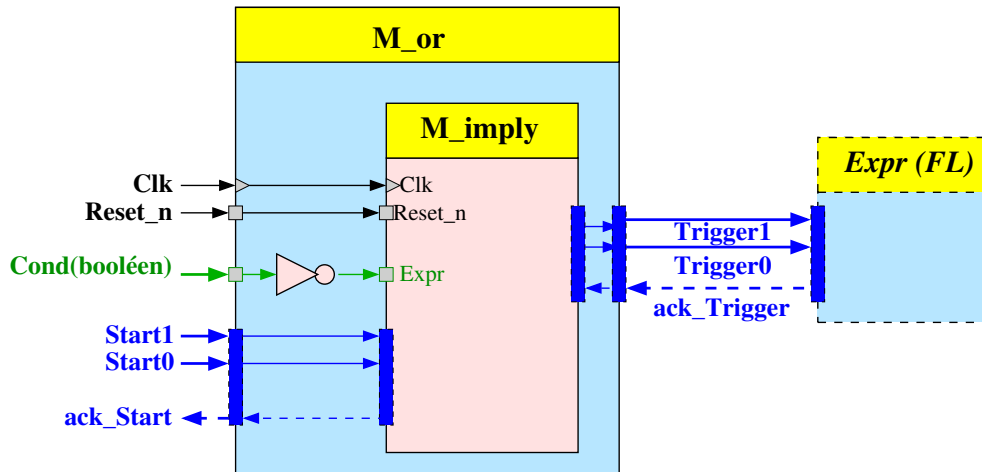


FIGURE A.28 – Implémentation de $Expr$ or $Cond$

Lorsque les deux opérandes sont booléens, il suffit de les connecter à une porte "OU" dont la sortie sera l'entrée d'un M_signal .

A.6 Équivalence entre opérateurs PSL et moniteurs primitifs

Le tableau A.1 fait correspondre le nom de chaque bloc précédemment décrit dans cette annexe à l'opérateur PSL, ou partie de l'opérateur PSL, qu'il implémente.

Bloc matériel	Opérateur PSL ou fonction
G_init	Démarre l'évaluation d'une propriété
M_signal	Vérifie la valeur d'un booléen
M_always	always
M_never	never
M_eventually!	eventually!
M_imply	→ avec opérande droit FL
M_imply_b	→ avec opérande droit booléen
M_and	and avec un ou deux opérandes FL
Or_pending	Utile à la connexion des signaux <i>Pending</i>
M_or	or avec un opérande FL
M_until, M_until!, M_until_, M_until!_	until, until!, until_, until!_
M_before, M_before!, M_before_, M_before!_	before, before!, before_, before!_
M_next, M_next!	next, next! ou un étage du M_next[K] ou du M_next![K]
M_next[K], M_next![K]	next[K], next![K]
M_next_a, M_next_a!	correspond à un des étages i à j du M_next_a[I..J] ou du M_next_a![I..J]
M_next_a[I..J], M_next_a![I..J]	next_a![I..J], next_a[I..J]
M_next_e, M_next_e!	correspond à un des étages i à j du M_next_e[I..J] ou du M_next_e![I..J]
M_next_e[I..J], M_next_e![I..J]	next_e![I..J], next_e[I..J]
M_next_event(b), M_next_event!(b)	next_event(b), next_event!(b)
M_next_eventK, M_next_event!K	correspond à next(next_event(b)), respectivement next!(next_event!(b))
M_next_event(b)[K], M_next_event!(b)[K]	next_event(b)[K], next_event!(b)[K]
M_next_event_a, M_next_event_a!	correspond à un des étages i à j du M_next_event_a(b)[I..J] ou M_next_event_a!(b)[I..J]
M_next_event_e, M_next_event_e!	correspond à un des étages i à j du M_next_event_e[I..J] ou M_next_event_e![I..J]

Table A.1 – Correspondance matériel et opérateur PSL

Annexe **B**

Le *Synchro@clk*

B.1 Code VHDL comportmental du *Synchro@clk*

```
entity synchroat is
    port( Exp, clk ,ack_Exp_S, rb: in  std_logic;
          Exp_S1, Exp_S0:out std_logic);
end synchroat;

architecture behav of synchroat is
    signal E1, E0:std_logic;
begin

PIQ : Process (clk , ack_Exp_S , rb)
begin
    If rising_edge(clk) and rb = '0' then
        E0<='0';
        E1<='0';
    elsif rising_edge(clk) and rb = '1' and Exp='1' and ack_Exp_S='1' then
        E0<='0';
        E1<='1';
    elsif rising_edge(clk) and rb='1' and Exp='0' and ack_Exp_S='1' then
        E0<='1';
        E1<='0';
    elsif ack_Exp_S = '0' then
        E1<='0';
        E0<='0';
    else
        E1<=E1;
        E0<=E0;
    end if;
end Process;
Exp_S1 <= E1 after 1 ns;
Exp_S0 <= E0 after 1 ns;
end behav;
```

B.2 Description PSL des portes utiles à la preuve RAT des synchroniseurs

On donne ici les propriétés PSL permettant de décrire les portes qui ont été utilisées pour la preuve du fonctionnement du *Synchro@clk*, du *Synchro_qdi*, du *Synchro_qdi_f* et du *Clk_manager* dans RAT. On rappelle pour chaque porte la **Set Expression (SE)** et la **Reset Expression (RE)**. La **SE** est la combinaison des entrées faisant basculer la sortie de la porte à '1' et la **RE** est la combinaison des entrées faisant basculer la sortie de la porte à '0'.

B.2.1 La porte M2RB

On rappelle la porte de Müller à deux entrées sur la figure B.1. Pour la porte M2RB,

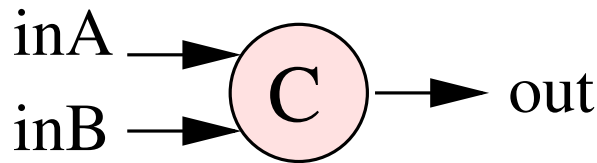


FIGURE B.1 – La porte M2RB

on a :

SE=inA.inB

RE= !inA. !inB

On obtient donc :

prop0to1 is assert

always((inA.inB. !out) -> eventually! out)

prop1to0 is assert

always((!inA. !inB.out) -> eventually! !out)

prop1 is assert

always(out -> (out until_ (!inA. !inB)))

prop0 is assert

always(!out -> (!out until_ (inA.inB)))

B.2.2 La porte M3R

On rappelle la porte de Müller à trois entrées sur la figure B.2. Pour la porte M3R, on

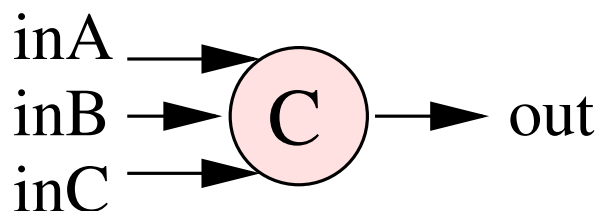


FIGURE B.2 – La porte M3R

a :

```

SE=inA.inB.inC
RE=!inA.!inB.!inC
On obtient donc :
prop1to0 is assert
always((inA.inB.inC.!out) -> eventually! out)
prop0to1 is assert
always((!inA.!inB.!inC.out) -> eventually! !out)
prop1 is assert
always(out -> (out until_ (!inA.!inB.!inC)))
prop0 is assert
always(!out -> (!out until_ (inA.inB.inC)))

```

B.2.3 La porte NOR2A

On rappelle la porte NOR assymétrique à deux entrées sur la figure B.3. Pour la porte

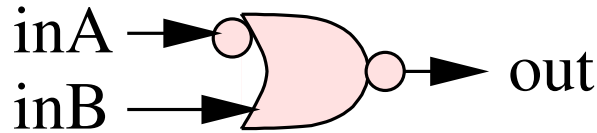


FIGURE B.3 – La porte NOR2A

NOR2A, on a :

```

SE=!inA+inB
RE=inA.!inB
On obtient donc :
prop1to0 is assert
always(((!inA+inB).out)-> eventually! !out)
prop0to1 is assert
always((inA.!inB.!out)-> eventually! out)
prop1 is assert
always((inA.!inB.out) -> (out until_ (!inA+inB)))
prop0 is assert
always(((!inA+inB).!out)->(!out until_ (inA.!inB)))

```

B.2.4 La porte NOR3A

On rappelle la porte NOR assymétrique à trois entrées sur la figure B.4. Pour la porte NOR3A, on a :

```

SE=!inA+inB+inC
RE=inA.!inB.!inC
On obtient donc :
prop1to0 is assert
always(((!inA+inB+inC).out)-> eventually! !out)

```

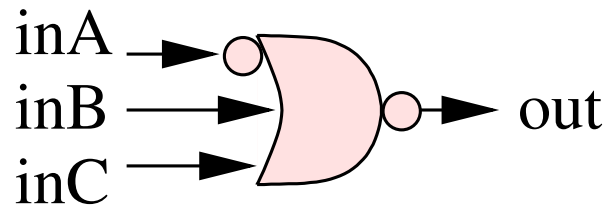


FIGURE B.4 – La porte NOR3A

prop0to1 is assert

`always((inA. !inB. !inC. !out)-> eventually! out)`

prop1 is assert

`always((inA. !inB. !inC.out) -> (out until_ (!inA+inB+inC)))`

prop0 is assert

`always((( !inA+inB+inC). !out)->( !out until_ (inA. !inB. !inC)))`

B.2.5 La porte NOR3

On rappelle la porte NOR à trois entrées sur la figure B.5. Pour la porte NOR3, on a :

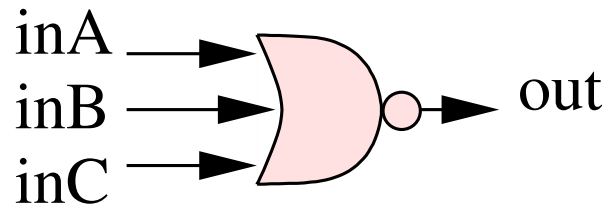


FIGURE B.5 – La porte NOR3

SE=inA+inB+inC

RE= !inA. !inB. !inC

On obtient donc :

prop1to0 is assert

`always(((inA+inB+inC).out)-> eventually! !out)`

prop0to1 is assert

`always(( !inA. !inB. !inC. !out)->eventually! out)`

prop1 is assert

`always(( !inA. !inB. !inC.out) -> (out until_ (inA+inB+inC)))`

prop0 is assert

`always(((inA+inB+inC). !out)->( !out until_ ( !inA. !inB. !inC)))`

B.2.6 La porte NOR2

On rappelle la porte NOR à deux entrées sur la figure B.6. Pour la porte NOR2, on

a :

SE=inA+inB

RE= !inA. !inB

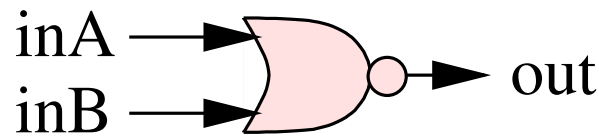


FIGURE B.6 – La porte NOR2

On obtient donc :

prop1to0 is assert

`always(((inA+inB).out)-> eventually! !out)`

prop0to1 is assert

`always((!inA. !inB. !out)-> eventually! out)`

prop1 is assert

`always((!inA. !inB.out) -> (out until_ (inA+inB)))`

prop0 is assert

`always(((inA+inB). !out)->(!out until_ (!inA. !inB)))`

B.2.7 La porte AND2

On rappelle la porte AND à deux entrées sur la figure B.7. Pour la porte AND2, on

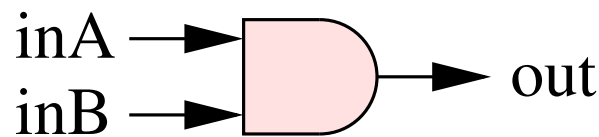


FIGURE B.7 – La porte AND2

a :

SE=inA.inB

RE= !inA+ !inB

On obtient donc :

prop1to0 is assert

`always(((!inA+ !inB).out)-> eventually! !out)`

prop0to1 is assert

`always((inA.inB. !out)-> eventually! out)`

prop1 is assert

`always((inA.inB.out) -> (out until_ (!inA+ !inB)))`

prop0 is assert

`always(((!inA+ !inB). !out)->(!out until_ (inA.inB)))`

B.2.8 La porte NAND2

On rappelle la porte NAND à deux entrées sur la figure B.8. Pour la porte NAND2, on a :

SE= !inA+ !inB

RE=inA.inB

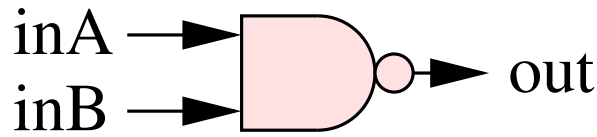


FIGURE B.8 – La porte NAND2

On obtient donc :

```

prop1to0 is assert
always(((inA.inB).out)-> eventually! !out)
prop0to1 is assert
always((!inA+!inB.!out)-> eventually! out)
prop1 is assert
always((!inA+!inB.out) -> (out until_ (inA.inB)))
prop0 is assert
always(((inA.inB).!out)->(!out until_ (!inA+!inB)))

```

B.2.9 La porte INV2

On appelle *in* l'entrée de la porte inverseuse et *out* sa sortie. Pour la porte INV2, on a :

SE=!in

RE=in

On obtient donc :

```

prop1to0 is assert
always(in.out)-> eventually! !out)
prop0to1 is assert
always(!in.!out)-> eventually! out)
prop1 is assert
always(!in.out) -> (out until_ in))
prop0 is assert
always(in.!out)->(!out until_ !in))

```

B.2.10 La Flip-Flop

Pour la Flip-Flop la mise à jour de la sortie *Q* ne se fait que lors des fronts montants de *Clk*, on définit un front montant de *Clk* de la manière suivante :

```

Proprising_clk_edge is assert
always(rising_clk_edge<=>( prev(!Clk) and Clk))

```

On appelle *D* l'entrée de la Flip-Flop, on a donc **SE**=*rising_clk_edge.D* et **RE**=*rising_clk_edge.!D* ce qui nous donne :

```

prop0to1 is assert
always((rising_clk_edge.D.!Q)-> eventually! Q)
prop1to0 is assert

```

```
always((rising_clk_edge.!D.Q)-> eventually! !Q)
```

Pour les conditions de maintien de la sortie Q dans l'état haut, si on a un front montant de Clk et que D est actif, alors la sortie Q reste dans l'état haut. De plus tant qu'il n'y a pas de front montants de Clk , la sortie reste dans son état précédent, c'est à dire l'état haut. On obtient donc :

prop1 is assert

```
always((!rising_clk_edge.Q)+ (rising_clk_edge.D.Q)) -> ( Q until_ rising_clk_edge.!D ))
```

prop0 is assert

```
always((!rising_clk_edge.!Q)+ (rising_clk_edge.!D.!Q)) -> (!Q until_ rising_clk_edge.D ))
```

Ainsi on obtient les quatre propriétés qui permettent de décrire en PSL le fonctionnement de la Flip-Flop.

B.2.11 Le SYNC

On rappelle que le SYNC échantillonne la valeur de son entrée $Expr$ sur les front montant de son signal de synchronisation CP . La sortie $Z1$ sera à '1' si $Expr='1'$ au moment du front montant de CP et le reste jusqu'à front descendant de CP . La sortie $Z0$ sera à '1' si $Expr='0'$ au moment du front montant de CP et le reste jusqu'à front descendant de CP . Lorsque $CP='0'$, les deux sorties $Z1$ et $Z0$ sont nulles. On a donc une **SE** et une **RE** pour la sortie $Z1$. On a aussi une **SE** et une **RE** pour la sortie $Z0$. Enfin il est important de noter que les sorties $Z1$ et $Z0$ du SYNC ne peuvent être actives en même temps. On a donc :

prop0to1_{Z1} is assert

```
always((rising_CP_edge.Expr.!Z1.!Z0)-> eventually!(Z1.!Z0))
```

prop0to1_{Z0} is assert

```
always((rising_CP_edge.!Expr.!Z1.!Z0)-> eventually!(!Z1.Z0))
```

prop1to0_{Z1} is assert

```
always((!CP.Z1.!Z0)-> eventually!(!Z1.!Z0))
```

prop1to0_{Z0} is assert

```
always((!CP.!Z1.Z0)-> eventually!(!Z1.!Z0))
```

prop_{ME} is assert

```
always(!Z1 + !Z0)
```

prop1_{Z1} is assert

```
always((CP.Z1.!Z0)-> ((Z1.!Z0) until_ !CP ))
```

prop1_{Z0} is assert

```
always((CP.!Z1.Z0)-> ((!Z1.Z0) until_ !CP ))
```

prop0_{Z1} is assert

```
always((!Expr.rising_CP_edge.!Z1.!Z0)-> ((!Z1) until_ Z0 ))
```

prop0_{Z0} is assert

```
always((Expr.rising_CP_edge.!Z1.!Z0)-> ((!Z0) until_ Z1 ))
```

prop0bis_{Z1} is assert

```
always((!CP.!Z1.Z0)-> ((!Z1) until_ !Z0 ))
```

prop0bis_{Z0} is assert

```
always((!CP.Z1.!Z0)-> ((!Z0) until_ !Z1 ))
```

On explique la description ci-dessus : les propriétés **prop0to1** permettent de modéliser le passage d'une des sorties ($Z1$ ou $Z0$) à '1' lors d'un front montant de CP (*rising_CP_edge*), les propriétés **prop1to0** permettent de modéliser le passage des deux sorties de l'état haut vers l'état bas lorsque CP passe à '0', **prop_{ME}** permet de modéliser le fait que les sorties $Z1$ et $Z0$ du SYNC ne sont jamais actives en même temps, les propriétés **prop1** permettent de modéliser l'absence d'aléas à '0' sur une sortie qui se trouve dans l'état haut, enfin les propriétés **prop0** et **prop0bis** permettent de modéliser l'absence d'aléas à '1' sur les sorties $Z1$ ou $Z0$ lorsque ces dernières sont dans l'état bas.

Bibliographie

- [20110] *IEEE standard for property specification language PSL*. pub-IEEE-STD, 2010.
- [AMAF09] K. Alsayeg, K. Morin-Allory, and L. Fesquet. Rat-based formal verification of qdi asynchronous controllers. In *Forum on specification & Design Languages (FDL'09)*, Sophia-Antipolis (France), September 2009.
- [Bal] <http://apt.cs.man.ac.uk/apt/projects/tools/balsa/>.
- [BCC⁺03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
- [BCP⁺07] Roderick Bloem, Roberto Cavada, Ingo Pill, Marco Roveri, and Andrei Tchaltsev. Rat: A tool for the formal analysis of requirements. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 263–267. Springer Berlin / Heidelberg, 2007.
- [BCZ06] M. Boulé, J-S. Chenard, and Z. Zilic. Adding debug enhancements to assertion checkers for hardware emulation and silicon debug. In *Proceedings of the 24th IEEE International Conference on Computer Design: ICCD'06*, Oct 2006.
- [BE00a] A Bardsley and Da Edwards. *The Balsa asynchronous circuit synthesis system*. 2000.
- [BE00b] Andrew Bardsley and Doug A. Edwards. Synthesising an asynchronous dma controller with balsa. *Journal of Systems Architecture*, 46(14):1309–1319, 2000.
- [Ber93] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.
- [Ber06] V. Bertacco. *Scalable Hardware Verification with Symbolic Simulation*. Springer Science + Business Media, Jan. 2006.
- [BH70] Jon G. Bredeson and Paul T. Hulina. Elimination of static and dynamic hazards in combinational switching circuits. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory (swat 1970)*, pages 104–108, Washington, DC, USA, 1970. IEEE Computer Society.
- [BH72] Jon G. Bredeson and Paul T. Hulina. Elimination of static and dynamic hazards for multiple input changes in combinatorial switching circuits. *Information and Control*, 20(2):114–124, 1972.

- [BLOF06] D. Borriore, M. Liu, P. Ostier, and L. Fesquet. *Applications of Specification and Design Languages for SoCs Selected papers from FDL 2005*, chapter PSL-based online monitoring of digital systems, pages 5–22. Springer Netherlands, Sep 2006.
- [BZ05] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proceedings of the 23rd IEEE International Conference on Computer Design: ICCD'05*, pages 221–228. IEEE Computer Society, 2005.
- [BZ06] M. Boulé and Z. Zilic. Efficient automata-based assertion-checker synthesis of psl properties. In *IEEE High Level Design Validation and Test Workshop (HLDVT'06)*, Nov 2006.
- [CBC⁺10] J. F. Christmann, E. Beigne, C. Condemine, N. Leblond, P. Vivet, G. Waltisperger, and J. Willemin. Bringing robustness and power efficiency to autonomous energy harvesting microsystems. *Asynchronous Circuits and Systems, International Symposium on*, 0:62–71, 2010.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
- [Chu87] T. Chu. Synthesis of self-timed vlsi circuits from graph-theoric specifications. Technical report, Cambridge, MA, USA, 1987.
- [CKK⁺97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, March 1997.
- [CKK⁺02] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.
- [Cla67] Wesley A. Clark. Macromodular computer systems. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 335–336, New York, NY, USA, 1967. ACM.
- [CM73] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.*, 22:421–422, April 1973.
- [DD03] A.V. Dinh-Duc. *Synthèse automatique de circuits asynchrones QDI*. 2003.
- [DGY92] Ilana David, Ran Ginosar, and Michael Yoeli. An efficient implementation of boolean functions as self-timed circuits. *IEEE Trans. Comput.*, 41:2–11, January 1992.
- [EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc. ISBN=0387353135, Secaucus, NJ, USA, 2006.
- [Eic65] E. B. Eichelberger. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Dev.*, 9:90–99, March 1965.
- [Fer11] Lucas Ferro. *Vérification de propriétés logico-temporelles de spécifications SystemC TLM*. PhD thesis, Grenoble Université, Juillet 2011.

- [FKL03] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, ISBN=1402074980, Jun. 2003.
- [FLT06] H. Foster, K. Larsen, and M. Turpin. Introduction to the new accellera open verification library. In *Proceedings of the Conference on Design and Automation and Test in Europe : DATE'06*, 2006.
- [FWMG05] H. Foster, Y. Wolfshal, E. Marschner, and IEEE 1850 Work Group. *IEEE standard for property specification language PSL*. pub-IEEE-STD, pub-IEEE-STD:adr, Oct 2005.
- [Hau95] Scott Hauck. Asynchronous design methodologies: An overview. In *PROCEEDINGS OF THE IEEE*, pages 69–93, 1995.
- [Huf64] D. A. Huffman. *The synthesis of sequential switching circuits*. E.F.Moore, Sequential Machines : Selected Papers. Addison-Wesley, 1964.
- [IEE05] *IEEE Std. 1800-2005, IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language*. 2005.
- [Jer02] A.A. Jerraya. *Conception logique et physique des systèmes monopuces (Traité EGEM Série électronique et micro-électronique)*. Hermes, 2002. ISBN: 2-7462-0434-7.
- [Jun04] Design, automation and test for asynchronous circuits and systems, 3rd edition. Technical report, ACiD-WG, June 2004.
- [KGN⁺09] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whitemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel Core i7 processor execution engine validation. In *Proceedings of the 21st International Conference on Computer Aided Verification: CAV'09*, pages 414–429, Jul. 2009.
- [KPKG02] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 21, pages 1377–1394, Dec 2002.
- [Lin98] Andrew M Lines. Pipelined asynchronous circuits. Technical report, Pasadena, CA, USA, 1998.
- [LPF] N. Leblond, A. Porcher, and L. Fesquet. Tiempo asynchronous design flow tutorial - modeling and debug. In *Design Automation Conference (DAC11)*.
- [MAB06] K. Morin-Allory and D. Borriane. Proven correct monitors from psl specifications. In *Proceedings of the Conference on Design, Automation and Test in Europe: DATE'06*, pages 1246–1251, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
- [MABBZ08] K. Morin-Allory, M. Boulé, D. Borriane, and Z. Zilic. Proving and disproving assertion rewrite rules by automated theorem proving. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop: HLDVT'08*, Nov. 2008.
- [MAFRB07] K. Morin-Allory, L. Fesquet, B. Roustan, and D. Borriane. Asynchronous online-monitoring of logical and temporal assertions. In *Proceedings of the 10th Forum on Specification and Design Languages: FDL'07*, pages 286–290. ECSI, 2007.

- [Mar90a] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.
- [Mar90b] Alain J. Martin. *Programming in VLSI: from communicating processes to delay-insensitive circuits*, pages 1–64. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
- [MB59] D. E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of the International Symposium on the Theory of Switching*, 1959.
- [McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, ISBN: 0-7923-9380-5, 1993.
- [Mil65] R.E. Miller. *Sequential Circuits and Machines, vol 2 of Switching Theory*, John Wiley & Sons. 1965.
- [MLM⁺97] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous mips r3000 microprocessor. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, ARVLSI '97, pages 164–, Washington, DC, USA, 1997. IEEE Computer Society.
- [MRB⁺03] P. Maurine, J. Rigaud, F. Bouesse, G. Sicard, and M. Renaudin. Static implementation of qdi asynchronous primitives. In Jorge Chico and Enrico Macii, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 2799 of *Lecture Notes in Computer Science*, pages 181–191. Springer Berlin / Heidelberg, 2003.
- [MRL⁺05] Y. Monnet, M. Renaudin, R. Leveugle, S. Dumont, and G.F. Bouesse. An Asynchronous DES Crypto-Processor Secured against Fault Attacks. In *15th IFIP Int. Conf. on Very Large Scale Integration Systems, VLSI-SoC'05, October 17-19*, Perth, Australie, 2005. ISBN: 07298-0610-3.
- [Odd09] Yann Oddos. *Vérification semi-formelle et synthèse automatique de PSL vers HDL*. PhD thesis, Université de Grenoble, Novembre 2009.
- [OMAB06] Y. Oddos, K. Morin-Allory, and D. Borriane. On-line test vector generation from temporal constraints written in PSL. In *Proceedings of the 14th IFIP/IEEE International Conference on Very Large Scale Integration System on Chip: VLSI SoC'06*, October 2006.
- [OMAB08a] Y. Oddos, K. Morin-Allory, and D. Borriane. Assertion-based design with horus. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign: MEMOCODE'2008*, Jun. 2008.
- [OMAB08b] Y. Oddos, K. Morin-Allory, and D. Borriane. Assertion-based verification and on-line testing in HORUS. In *Proceedings of the 3rd International Design and Test Workshop: IDT'08*, Monastir, Dec. 2008.
- [OVA] <http://www.open-vera.com/home.html>.
- [Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.
- [PF08] L. Pierre and L. Ferro. A tractable and fast method for monitoring systemC TLM specifications. In *IEEE Transactions on Computers*, volume 57, pages 1346–1356, 2008.

- [phi] <http://www.eetimes.com/electronics-news/4164955/philips-gambit-self-timing-s-time-is-here>.
- [PK00] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Proceedings of the International Conference on Computer Design: ICCD'00*, pages 459–464, 2000.
- [PLBN05] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proceedings of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign: MEMOCODE'05*, pages 15–24, Jul. 2005.
- [Ren00] M. Renaudin. Asynchronous circuits and systems : a promising design alternative. *Microelectronic Engineering*, Volume 54, Issues 1-2 , December:133–149, 2000.
- [RIG02] J. B. RIGAUD. *Spécification de bibliothèques pour la synthèse de circuits asynchrones*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.
- [SDMD93] Polly Siegel, Giovanni De Micheli, and David Dill. Automatic technology mapping for generalized fundamental-mode asynchronous designs. In *Proceedings of the 30th international Design Automation Conference, DAC '93*, pages 61–67, New York, NY, USA, 1993. ACM.
- [SRSR04] K. Slimani, Y. Remond, G. Sicard, and M. Renaudin. TAST profiler and low energy asynchronous design methodology. In *Integrated-Circuit-and-System-Design.-Power-and-Timing-Modeling,-Optimization-and-Simulation.-14th-International-Workshop,-PATMOS-2004.-Proceedings-, Lecture-Notes-in-Comput.-Sci.-Vol.3254*, pages 268–77, Santorini, Grèce, 2004. Springer-Verlag. ISBN: 3-540-23095-5.
- [Sut89] I. E. Sutherland. Micropipelines. *Commun. ACM*, 32:720–738, June 1989.
- [tiea] <http://www.tiempo-ic.com/products/acc.html>.
- [tieb] <http://www.tiempo-ic.com/uploads/press/tiempo>
- [Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1(4):197–204, 1986.
- [Ung71] S. H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Trans. Comput.*, 20:1437–1444, December 1971.
- [Ung83] Stephen H. Unger. *Asynchronous Sequential Switching Circuit*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1983.
- [Var06] M-Y. Vardi. From church and prior to PSL. *Proceedings of the Workshop on 25 Years of Model Checking, Federated Logic Conference: FLOC'06*, Aug. 2006.
- [vB92] Kees van Berkel. Beware the isochronic fork. *Integr. VLSI J.*, 13:103–128, June 1992.
- [vBKR⁺91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The vlsi-programming language tangram and its translation into handshake circuits. In *Proceedings of the conference on European design automation, EURO-DAC '91*, pages 384–389, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

- [Wak94] John F. Wakerly. *Digital design: principles and practices (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
- [Yah09] Eslam Yahya. *Performance Modeling, Analysis and Optimizations of Multi-Protocol Asynchronous Circuits*. PhD thesis, Institut Polytechnique de Grenoble, December 2009.

Index

Clk_manager, 102–110, 112, 113, 115, 121, 122, 156
Clk_mgt, 102–113, 115, 117
Fork, 52, 69
G_init, 30, 55, 59, 63, 64, 71–73, 93, 94, 153
Half Buffer, 50, 51, 61, 92, 112, 127, 138, 140, 144, 150
Half Buffer Load, 50
Horus, 2, 27, 29, 32, 52, 54, 72, 92, 95, 120

M_always, 59, 93, 94, 130, 153
M_and, 150, 153
M_before, 73, 93, 94, 133, 134, 153
 _, 93, 134, 153
M_before_, 93, 134, 135, 153
M_eventually, 93, 132, 153
M_imply, 56, 57, 59–62, 71, 93, 94, 123, 124, 149, 152, 153
M_imply_b, 149, 153
M_never, 93, 131, 153
M_next, 59, 65, 66, 68–70, 93, 94, 138–144, 146, 153
M_next_a, 68, 69, 93, 140, 144, 146, 153
M_next_e, 68, 70, 93, 141–143, 146, 147, 153
M_next_event, 70, 71, 93, 136, 144, 146, 147, 153
M_next_event_a, 70, 93, 144, 145, 153
M_next_event_e, 70, 93, 146–148, 153
M_or, 152, 153
M_signal, 60, 71, 73, 93, 98, 101, 125, 142, 147, 149, 150, 152, 153
M_until, 62–64, 93, 126, 127, 153
 _, 93, 128, 153
M_until_, 93, 128, 153

Or_pending, 150, 151, 153
PSL_{ss}, xxii, 25, 126, 131–133, 141, 146, 152
Synchro@clk, 58, 59, 61, 65, 66, 72, 75, 78–82, 84–90, 92, 93, 95, 98, 100–103, 109, 112, 115, 117, 130, 155, 156
Synchro_qdi, 103–105, 107, 109, 112, 115, 117, 156
Synchro_qdi_f, 105–107, 109, 112, 113, 115, 117, 156

Publications de l'Auteur

Conférences internationales avec comité de lecture, Sélection sur le papier complet

- A. Porcher, K. Morin-Allory, L. Fesquet, *Synthesis of Asynchronous Monitors for Critical Electronic Systems*, Proceeding of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS'10, Vienna - Austria, April 2010.
- A. Porcher, K. Morin-Allory, L. Fesquet, *Synthesis of Quasi-Delay Insensitive Monitors*, Proceeding of the 7th Conference on Ph.D. Research in Microelectronics and Electronics PRIME'11, Madonna Di Campaglio - Italia, July 2011.
- A. Porcher, K. Morin-Allory, L. Fesquet, A. Chagoya, *Does Asynchronous Technology Bring Robustness in Synchronous Circuit Monitoring?*, Proceeding of 2011 Forum on specification and Design Language FDL'11, Oldenburg - Germany, September 2011.

Présentation de Poster

- A. Porcher, L. Fesquet, K. Morin-Allory, *Synthesis of asynchronous monitors for critical systems*, Journée EMSoc Recherche, Villard de Lans, Octobre 2008.
- A. Porcher, L. Fesquet, K. Morin-Allory, *Les circuits asynchrones absorbent les hypothèses temporelles des circuits synchrones!*, GDR System on Chips-System in Package SoC-SiP, CPE Lyon, Juin 2011.

Tutorial

- N. Leblond, A. Porcher, L. Fesquet, *Tiempo Asynchronous Design Flow Tutorial - Modeling and Debug*, Design Automation Conference DAC'11, San Diego - USA, June 2011.

Mots clés

Quasi Insensibilité aux Délais, Vérification basée sur les assertions, Moniteur asynchrone, Property Specification Language, Circuits asynchrones, Vérification semi-formelle, Synchronisation, Robustesse

Key words

Quasi Delay Insensitive, Assertion Based Verification, Asynchronous Monitor, Property Specification Language, Asynchronous Circuit, Semi-Formal Verification, Synchronization, Robustness

Résumé

Avec l'avènement des systèmes intégrés complexes, la vérification par assertion (Assertion Based Verification ou ABV) s'est imposée comme une solution pour la vérification semi-formelle des circuits. L'ABV permet de valider qu'un circuit satisfait ou non une propriété (ou assertion). Des travaux antérieurs ont montré qu'il était possible de synthétiser ces propriétés sous la forme de moniteurs matériels. Ces derniers peuvent ainsi être embarqués à demeure sur un circuit afin qu'ils assurent une tâche de monitoring. Avec un objectif de surveillance et de sûreté, l'utilisation de tels moniteurs est un plus. Néanmoins, ces derniers sont aussi sensibles que les circuits surveillés à une dégradation environnementale (tension, température, vieillissement. . .). Afin de réduire le risque de dysfonctionnement des moniteurs, initialement conçus comme des circuits synchrones, une variante asynchrone (quasi-insensible aux délais) est proposée dans cette thèse. Ces travaux s'inscrivent dans le cadre du projet ANR SFINCS (Thalès, Dolphin Integration, TIMA) et ont mené à la définition d'une méthode de synthèse de moniteurs asynchrones matériels tirant parti de la robustesse et de la modularité des implémentations asynchrones. Les études menées se focalisent en premier lieu sur la conception d'une bibliothèque de moniteurs élémentaires asynchrones et sur une méthode d'interconnexion ad hoc permettant de constituer des moniteurs complexes. Afin de garantir les bonnes propriétés de robustesse de ces moniteurs, une étude a été menée à l'aide de l'outil de vérification formelle RAT. Il a notamment été prouvé que la connexion d'un moniteur asynchrone avec un circuit synchrone (à surveiller) était un point particulièrement délicat car les hypothèses du circuit synchrone contraignent le moniteur asynchrone. Il a donc été proposé d'introduire un dispositif de contrôle de l'horloge du circuit synchrone, appelé « clock stretching », afin de relaxer les hypothèses temporelles synchrones qui sont appliquées à la partie asynchrone.

Abstract

With the advent of complex integrated systems, the assertion based verification (ABV) has emerged as a solution for the semi-formal circuits verification. The ABV is used to validate that a circuit satisfies a property (or assertion). Previous works have shown that it is possible to synthesize these properties in the form of hardware monitors. These can then be embedded permanently on a circuit so that they provide monitoring task. With a goal of security and surveillance, the use of such monitors is a plus. Nevertheless, they are as sensitive as the monitored circuits to environmental degradation (voltage, temperature, age...). To reduce the risk of failure in monitors, originally designed as synchronous circuits, an asynchronous variant (quasi-delay insensitive) is proposed in this thesis. This work is part of the ANR project SFINCS (Thales, Dolphin Integration, TIMA) and led to the definition of a method for synthesizing asynchronous hardware monitors leveraging the robustness and modularity of asynchronous implementations. The studies focus primarily on the design of a library of basic asynchronous monitors and an ad hoc method of interconnection to build complex monitors. To ensure the robustness of these monitors, a study was conducted using formal verification tool RAT. In particular it was proved that the connection of an asynchronous monitor with a synchronous circuit (to watch) was particularly tricky because the timing assumptions of synchronous circuit impact the asynchronous monitor. It was therefore proposed to introduce a device, called "clock stretching", for controlling the clock of the synchronous circuit and relax synchronous timing assumptions that are applied to the asynchronous monitor.

