

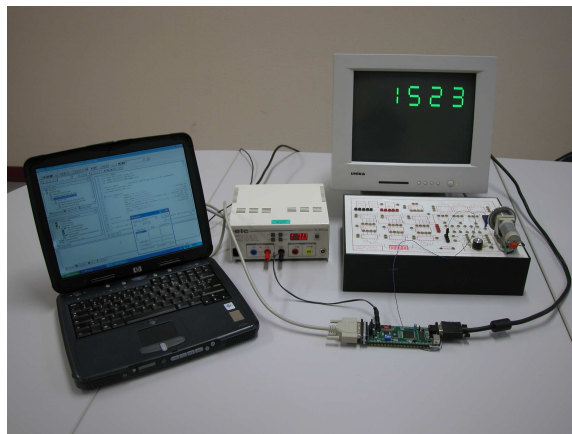
TP ARS 2

Implantation d'un système automatisé complet sur circuit logique programmable : le Tachymètre¹

Objectif du TP Ce TP, qui s'étale sur 7 séances de 1h30, a pour but de développer un système automatisé complet. Le projet concerne le développement d'un tachymètre mesurant la vitesse de rotation d'un moteur à courant continu. Au cours du TP, le travail consistera à développer la partie opérative et la partie commande de ce tachymètre. L'implémentation matérielle de cette application se fera sur une carte de développement à base de FPGA de Xilinx en exploitant le langage VHDL pour la décrire.

Introduction Ce TP a donc pour objectif la description en VHDL d'un tachymètre mesurant la vitesse de rotation d'un moteur à courant continu. La figure ci-dessous présente une photo des matériels mis en oeuvre. On y retrouve :

- un PC de développement sous Windows 98 avec le logiciel de développement de Xilinx (WebPack 4.2) et le logiciel de gestion du téléchargement de l'application vers la carte de développement (XessTools) ;
- une carte cible (XSA-50) développée par la société XESS intégrant un FPGA Xilinx (Spartan2 XC2S50tq144-5) avec un câble de téléchargement sur port parallèle ;
- une maquette intégrant un moteur à courant continu. Cette maquette comporte un potentiomètre de réglage manuel de la vitesse de rotation, permettant une variation de la vitesse dans une *plage de 0 à 1100 tours/minute* environ. En outre, la maquette reçoit un dispositif électronique, basé sur une fourche optique et un disque perforé, fournissant des impulsions électriques calibrées à l'image de la vitesse de rotation du moteur (fréquence des impulsions proportionnelle à la vitesse de rotation);
- une alimentation pour la carte de développement ;
- un écran VGA de PC qui servira à afficher la vitesse mesurée en tours par minute.



Matériel pour un poste de développement autonome

L'ensemble de ces éléments forme donc tout l'environnement matériel et logiciel qui permettra le développement complet de l'application. Pendant le TP, chaque binôme ne disposera pas du poste de développement complet de la figure ci-dessus. Les étudiants disposeront d'un PC de développement avec tout l'environnement logiciel nécessaire. Un ou deux postes de téléchargement de l'application avec tout l'environnement matériel seront disponibles au fond de la salle de TP pour tester physiquement l'application décrite.

Cahier des charges L'objectif, ici, est de présenter succinctement les quelques points importants du projet, en termes de contraintes et méthodes envisagées, pour développer l'application. Les détails seront fournis plus tard au gré des besoins.

Vitesse L'objectif est de mesurer, et afficher sur écran VGA, la vitesse de rotation du moteur. La plage de vitesse couverte par le moteur s'étalant de 0 à 1100 tours/minute, il faudra afficher la vitesse sous la forme d'un nombre à 4 digits.

¹Document réalisé avec Lyx1.3.5 et Xfig3.2.4 sous FreeBSD5.4

<i>signal</i>	<i>broche</i>	<i>signal</i>	<i>broche</i>	<i>signal</i>	<i>broche</i>	<i>signal</i>	<i>broche</i>	<i>signal</i>	<i>broche</i>
clk25	88	initgraf	54	rgb<3>	20	x1	49	x5	62
hsynch	23	rgb<0>	12	rgb<4>	21	x2	46	x6	57
vsynch	26	rgb<1>	13	rgb<5>	22	x3	39	bt	44
i	80	rgb<2>	19	led	60	x4	67		

Table 1: Correspondance entre les signaux et les broches du FPGA pour l'application complète.

Impulsions Un disque, régulièrement percé de 60 trous sur toute sa circonférence, est fixé solidairement à l'axe du moteur. Le disque perforé passe entre les deux branches de la fourche optique de manière à ce que la rotation de celui-ci entraîne le passage/rupture du faisceau infrarouge depuis la LED jusqu'au photo-transistor. Ainsi, ce dispositif fournit un signal logique de rapport cyclique 1/2 et dont la fréquence est à l'image de la vitesse de rotation du moteur.

Affichage Les 4 digits seront générés sur l'écran VGA au format de 4 afficheurs 7 segments. Naturellement, le FPGA devra au préalable générer les signaux de synchronisation permettant à l'écran VGA de se synchroniser. Le FPGA pourra alors générer en parallèle les signaux d'information de la vidéo pour afficher les digits (signal RGB mélangeant les 3 couleurs primaires Rouge - Vert - Bleu pour synthétiser au cours du temps la couleur de chaque pixel de l'écran) .

Base de temps La mesure de la vitesse se fera par comptage des impulsions sur une durée d'une seconde. Dans la mesure où le moteur ne subit pas de variations brutales de sa vitesse, la vitesse moyenne mesurée sur 1 seconde sera alors considérée comme une vitesse instantanée. Une conversion du nombre d'impulsions par seconde en tours/minute sera à étudier. Le comptage se fera en cascasant 4 compteurs BCD, ce qui facilitera la gestion de l'affichage.

Interconnexions et réglages initiaux En toute rigueur, un certain nombre de réglages et de connexions sont à réaliser pour mettre en place le système complet. Dans la pratique, ces réglages seront déjà fait lorsque les étudiants prendront possession du poste de téléchargement. On rappelle cependant les divers points qu'il faudrait aborder pour mettre en oeuvre le système en partant de rien, d'autant plus qu'en cas de non respect de ces réglages, il peut y avoir disfonctionnement, voire détérioration, du matériel. Le contrôle de ces réglages pourra le cas échéant, constituer la première étape d'un débogage. Le schéma synoptique de la figure 1 présente les interconnexions entre les divers matériels du poste de mesure.

1. On prendra bien soin de régler l'alimentation de la carte XSA-50 à 7V DC en vérifiant la polarité des branchements sous peine de détérioration du matériel.
2. Les connexions entre le FPGA et les divers éléments sont imposés soit par la conception de la carte XSA-50, soit par le pré-câblage réalisé pour faciliter et accélérer les tests. Par conséquent, il sera nécessaire de respecter ces contraintes de câblage dans la phase de développement de l'application, et tout particulièrement dans la phase d'attribution des broches du circuit FPGA aux signaux entrants/sortants. Le tableau 1 récapitule les connexions utilisées pour l'ensemble de l'application. Les étudiants adopteront les noms de signaux choisis dans l'énoncé pour faciliter le suivi du développement des projets.
3. Au cours du développement de l'application, il sera utile de réaliser des tests intermédiaires avec des connexions légèrement adaptées aux différents exercices abordés. Il faudra alors prendre les libertés nécessaires pour utiliser d'autres entrées/sorties et s'appuyer sur la documentation de la carte XSA-50. On y trouvera les connexions existantes sur la carte, entre le FPGA et les divers interrupteurs, led ... pour s'en servir à bon escient.
4. La carte XSA-50 dispose d'un oscillateur programmable (DS 1075) qui fournit au FPGA un signal d'horloge de référence. Cette horloge programmable oscille initialement à 100MHz. Il est possible de programmer ce circuit avec une division de cette fréquence de base par des valeurs entières comprises entre 1 et 2052. Cela permet de disposer d'une horloge dont la fréquence peut s'étaler entre 100MHz et 48,7 KHz environ. **Pour les besoins de l'application, elle devra être divisée par 4**, c'est à dire 25 MHz. Cette programmation s'effectue avec les outils spécifiques de la société XESS : XSTOOLS ; et en particulier le programme GXSSSETCLK.

Encore une fois, tous ces réglages devraient être faits par défaut mais il pourra être utile de se les rappeler en cas de non fonctionnement de l'application.

Organisation du travail Comme cela a déjà été présenté ci-dessus, le travail s'organise en 7 séances de 1h30. Il va de soi que l'on ne peut aborder l'ensemble de l'application d'un bloc (description du comportement des sorties à partir du comportement des entrées). Le travail sera donc segmenté en exercices, dont on capitalisera les résultats, pour aboutir à l'application complète. Par ailleurs, cela reste la démarche classique employée dans l'industrie pour développer une application tant du point de vue de la gestion de la difficulté de l'application, que

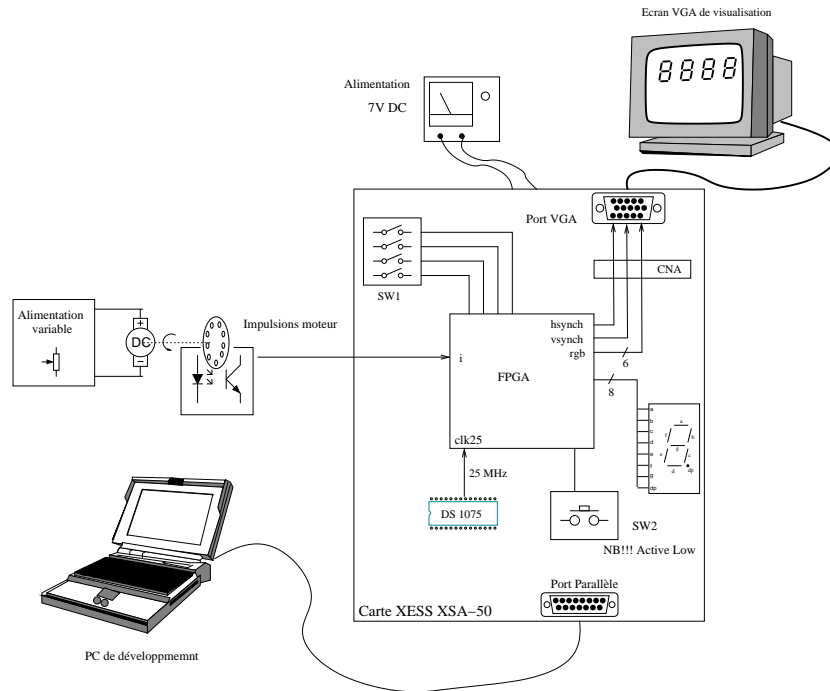


Figure 1: Schéma synoptique des éléments matériels constituant le poste de mesure

de la gestion du partage des tâches dans le cadre d'un travail d'équipe : développer des modules en les validant un par un, puis assembler ces modules pour concevoir l'application complète. Du point de vue de l'enseignement, cette démarche est tout à fait adaptée, dans la mesure où elle permet d'aborder :

- les divers styles de programmation et concepts de la description VHDL (description structurale et comportementale, notion de process, component, librairie, vecteur, ...) ;
- les concepts de câblage des algorithmes (logique combinatoire, séquentielle, synchrone, asynchrone ...) ;
- et bien sûr, l'implantation, ici dans un FPGA, d'un système automatisé complet constitué par une partie commande (décrite par un grafset) et une partie opérative avec les contraintes que cela occasionne et les règles à respecter qui découlent de ces contraintes.

Les exercices seront donc présentés dans un ordre respectant à la fois :

- la progression dans la difficulté de programmation en VHDL ;
- le souci d'un test facile des diverses descriptions VHDL.

De ce fait, l'ordre dans lequel seront présentés les exercices par rapport à l'ensemble des modules nécessaires à l'application pourrait apparaître comme un peu aléatoire. Les exercices seront envisagés pour permettre une description structurale de l'application où l'on viendra connecter des modules les uns aux autres pour constituer l'application complète. Par ailleurs, cette méthode de description permettra le développement de fonctionnalités supplémentaires améliorant l'application, le cas échéant (détection de survitesse avec alarme, ...).

Le schéma synoptique de la figure 2 présente l'ensemble des modules de l'application interconnectés. A la vue du temps imparti, on ne pourra envisager la description de tous les modules. Certains seront donc fournis prêts à l'emploi. Leur utilisation, qui nécessitera la compréhension du code VHDL écrit pour régler les valeurs de certains paramètres, sera en soi un bon exercice. Dans la mesure où les exercices seront capitalisés pour aboutir à l'application finale, il est indispensable de les traiter tous, et dans l'ordre proposé. D'un point de vue pratique, il faudra générer un projet par exercice. De plus, il faudra gérer l'ensemble sur les comptes utilisateur du serveur Samba du Département GEII, pour conserver la trace des travaux déjà menés avec succès et les réutiliser.

Ressources documentaires Pour aborder ce projet, il sera nécessaire de consulter un certain nombre de documents. Ils couvriront des aspects :

- logiciels : langage VHDL et logiciels de développement ;
- matériels : documentation de la carte de développement XSA-50 ;
- théoriques : Cours, TD et TP d'ENSL1 et d'ARS2 couvrant la logique et le grafset notamment.

Ces documents pourront donc provenir de diverses sources telles que :

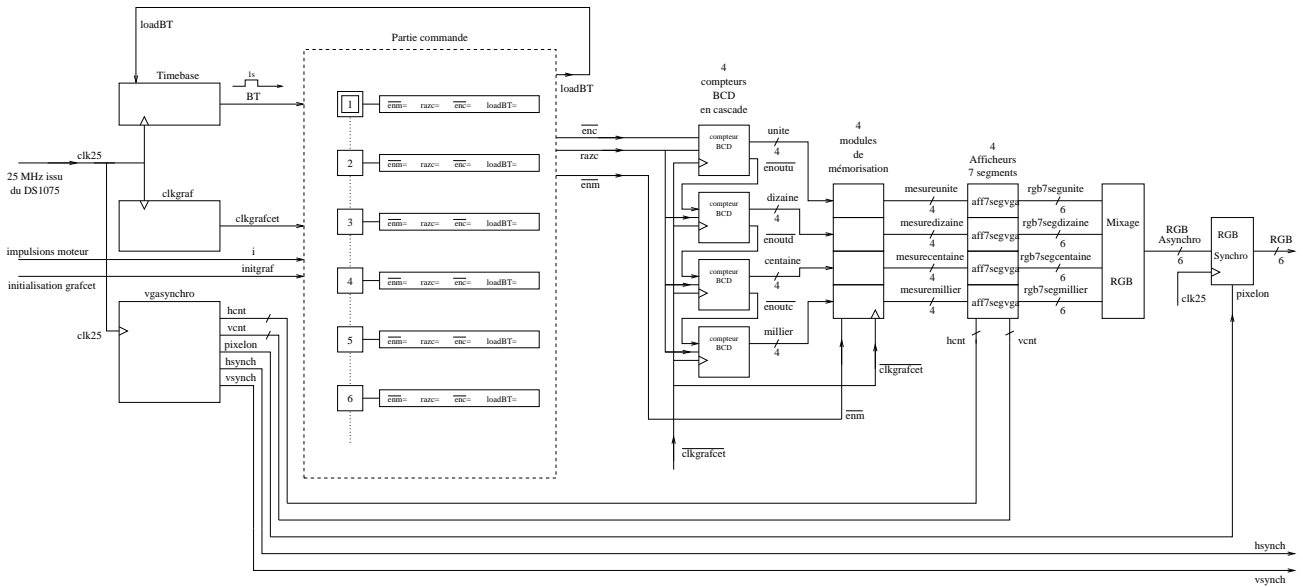


Figure 2: Schéma synoptique de l'application complète

- les enseignants, par le biais des polycopiés distribués dans les divers cours ;
- des fichiers .pdf de description de l'application ;
- les sites internet des sociétés Xilinx et XESS à savoir <http://www.xilinx.com> et <http://www.xess.com>.

Etape n°1 : Transcodeur BCD - 7 segments

Objectif Cette étape a pour but d'aborder essentiellement la *description de fonctions logiques combinatoires*, la notion de *component* et de *description structurelle* associée aux notions de *package* et de *library*. Bien sûr, cette première étape sera aussi l'occasion de la *première prise en main* du logiciel et de la carte de développement XSA-50.

Transcodeur première version

Préparation La solution retenue pour le comptage des impulsions (images de la vitesse de rotation) repose sur la gestion de 4 compteurs BCD placés en cascade pour fournir 4 nombres codés en BCD. L'affichage de la vitesse se fait, alors, sous la forme de 4 chiffres (unité, dizaine, centaine, millier) dessinés sur l'écran VGA sous la forme de 4 afficheurs 7 segments. Il est donc nécessaire de transformer le nombre délivré par le compteur BCD en une information permettant de piloter chaque segment de l'afficheur 7 segments. C'est donc une fonction de transcodage d'un code BCD en un code indiquant l'état logique de chaque segment de l'afficheur qui doit être mise en place.

1. Déterminer le nombre d'entrées et de sorties impliquées dans cette fonction de transcodage.
2. Dresser la table de vérité décrivant les valeurs des sorties en fonction des valeurs des entrées. **NB !!!** On décide d'allumer le segment g seul en cas d'erreur sur les entrées.
3. Etudier la documentation de la carte XSA-50 pour trouver les différents moyens matériels disponibles (interrupteurs, boutons poussoirs, afficheurs ...) permettant de tester cette fonction. La documentation de la carte XSA-50 est disponible à l'adresse suivante : <http://www.xess.com/manuals.html> sous la forme d'un fichier .pdf à *télécharger librement* ou bien directement sur le disque dur de la machine dans un répertoire que l'enseignant vous indiquera (ex c:\documentation). Proposer alors un schéma synoptique présentant le FPGA connectés avec les divers éléments matériels, utiles au test.
4. Retrouver dans l'annexe A de la documentation de la carte XSA-50 les connexions du FPGA avec les divers éléments de la carte. Préparer alors un tableau d'affectation des entrées et sorties de votre application aux broches du FPGA.

Travail à faire en TP On pourra se référer, partiellement au moins, au document placé en annexe A de cet énoncé. Il décrit la manière de créer un projet ainsi que les différentes étapes pour obtenir un *fichier .bit* nécessaire à la programmation du FPGA.

1. Prise en main du logiciel avec création d'un projet. Voir l'annexe A du polycopié de TP.
2. Décrire en VHDL l'*entity* de ce module transcodeur BCD-7 segments en vous appuyant sur la question 1. de la préparation.
3. Décrire ensuite l'architecture de ce transcodeur en vous appuyant sur la question 2. de la préparation.

4. Préparer un fichier de contraintes pour l'affectation des signaux aux broches du FPGA. C'est la même démarche que pour les *attribute pin_numbers* utilisés en ENSL1 sauf qu'il s'agit de générer un fichier texte séparé avec l'extension .ucf. La syntaxe est la suivante :


```
# un commentaire est précédé du signe #
# pour un signal simple sur 1 seul bit :
net "din" loc = "p35"; # le signal din sera rattaché à la broche 35 du FPGA
# pour une composante d'un vecteur :
net "s<2>" loc="p50"; # le rang du bit concerné est placé entre les signes < et >
# ici c'est le bit de rang 2 du vecteur s qui sera associé à la broche 50
```
5. Rassembler le code VHDL des questions 2. et 3. dans un même fichier .vhd si ce n'est pas le cas. Associer le fichier .ucf au projet comme source de contraintes pour l'*entity* du transcodeur (la seule pour l'instant quoi qu'il en soit). Compiler le programme en corrigeant d'éventuelles fautes de syntaxe ou de conception. Récupérer le *fichier .bit* issu de la compilation sur une disquette et programmer le FPGA en téléchargeant ce fichier .bit à l'aide du logiciel du package XSTOOLS : *GXSload*, sur la carte du poste de mesure au fond de la salle.
6. Vérifier le bon fonctionnement du transcodeur et faire valider par l'enseignant.

Component pilot7seg

L'objectif de ce deuxième exercice est de mettre en place immédiatement une méthode de développement bien structurée. En effet, le transcodeur précédent sera utilisé dans la suite du projet. Il convient donc de le placer dans un fichier .vhd spécifique pour que cette fonction de transcodage devienne une ressource au sein d'une librairie. Nous allons donc mettre en place une librairie. Le principe est de décrire une "boîte noire" pour y placer une fonctionnalité. En VHDL, on utilise le terme *component* pour désigner cette boîte noire. On trouvera, en annexe B du photocopié de TP, une explication accompagnée d'un exemple pour créer un fichier correspondant à une librairie avec la description d'un component et son utilisation.

Préparation L'idée est de mettre en place une description structurée, très simple ici puisqu'elle n'utilisera qu'une instanciation d'un seul component : le transcodeur BCD-7 segments.

1. Etudier l'exemple de *library* et de *component* de l'annexe B du photocopié de TP.
2. Reprendre le schéma synoptique de l'application précédente en intégrant dans le FPGA, le transcodeur BCD-7 segments sous la forme d'un *component* appelé (on dit aussi *instancié*). On prendra soin de bien présenter les connexions du *component* avec les entrées/sorties du FPGA, le tout au sein de l'application. On appellera ce *component pilot7seg*.

Travail à faire en TP

1. En vous appuyant sur l'annexe B, décrire en VHDL le transcodeur BCD - 7 segments sous la forme d'un *component* dans une *library*. Le package s'appellera *mylib* .
2. Refaire un projet en utilisant cette *library mylib* et en instanciant le *component pilot7seg* dans l'application globale selon le schéma synoptique de la préparation.
3. Configurer le FPGA, tester le fonctionnement et faire valider le programme VHDL et l'application par l'enseignant.

Etape n°2 : Horloge Grafcet et Base de temps

Objectif Cette étape a pour but d'aborder la description de la logique séquentielle synchrone en VHDL par l'utilisation de la notion de *process*.

Horloge de séquençement du grafcet Ce premier exercice permettra de se familiariser avec la description VHDL de la logique synchrone. Il s'agit de réaliser une fonction logique dont le fonctionnement est déclenché par un signal d'horloge, le tout au sein d'une structure de programmation qui se nomme *process* . Ici, nous allons décrire un diviseur de fréquence qui séquencera la partie commande de l'application, à savoir un grafcet. A partir du signal d'horloge à 25 MHz issu du DS1075, il faut générer un signal d'horloge de rapport cyclique 1/2 à 25 KHz . Le principe consiste à compter le temps qui s'écoule sur une demi-période du signal à générer, grâce à l'horloge de base à 25 MHz, et à inverser un signal à chaque fin de demi-période. Ce signal inversé régulièrement aura les caractéristiques souhaitées. La figure 3 présente le principe de ce diviseur de fréquence sous la forme d'un chronogramme partiel.

Préparation

1. Déterminer le nombre de coups d'horloge à 25MHz nécessaires pour obtenir la demi-période du signal d'horloge à 25 KHz souhaité.
2. Le diviseur de fréquence devra être programmé en tant que *component* et placé en *library mylib* car il sera utilisé par la suite dans l'application complète du tachymètre. Etudier ce diviseur de fréquence en termes de signaux d'entrée, de sortie et éventuellement de signaux internes au *component*. On appellera ce signal d'horloge à 25KHz *clkgrafcet*. On pourra appelé le *component* qui le génère *clkgraf*. proposer un schéma synoptique de *clkgraf*.

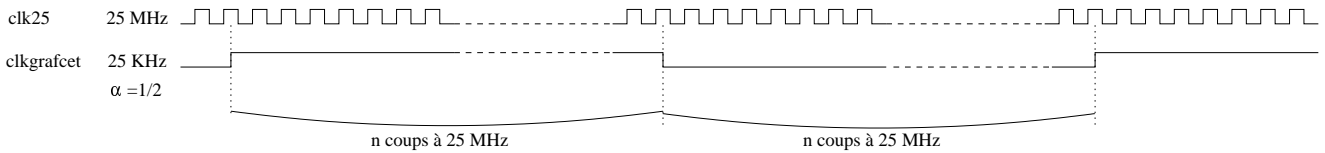


Figure 3: Chronogramme du diviseur de fréquence

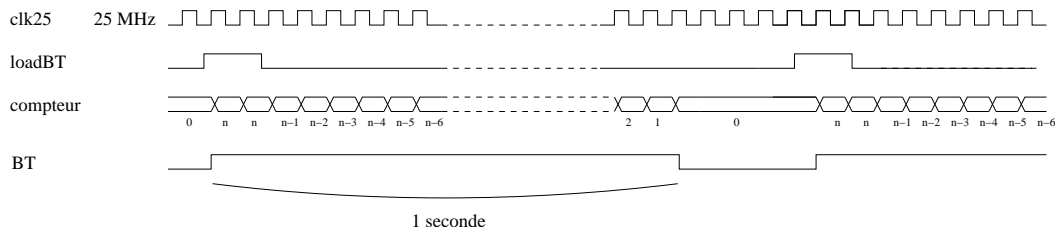


Figure 4: Chronogramme pour le signal **BT**

Travail à faire en TP

1. Ecrire le programme VHDL réalisant ce diviseur de fréquence par 1000 en l'intégrant à votre librairie. Il est à noter que ce diviseur de fréquence peut être décrit selon plusieurs styles de programmation VHDL.
2. Tester le fonctionnement de ce diviseur de fréquence en l'implantant sur la carte XSA-50. Pour cela, on affectera le signal d'horloge généré sur la broche 80 du FPGA et on mesurera la fréquence obtenue sur un fréquencemètre.
3. Faire valider le fonctionnement par l'enseignant.
4. Comment pouvez-vous expliquer la valeur mesurée au fréquencemètre ?

Signal de base de temps Il s'agit ici de générer un signal actif à l'état haut pendant 1 seconde à partir d'un signal de top départ. En d'autres termes, on doit réaliser un compte à rebours d'une seconde, ou encore en termes plus techniques, un monostable. Ce signal sera utilisé comme base de temps pour définir la durée du comptage des impulsions issues du moteur ; ce comptage fournissant la vitesse de rotation du moteur. Le principe de ce module consistera, à partir d'un signal (*loadBT*) à charger, de manière synchrone, une valeur dans un décompteur et à laisser ce décompteur évoluer jusqu'à sa valeur 0, où il devra se bloquer. La gestion du signal de base de temps en sortie (*BT*) consistera à placer ce signal à 1 tant que la valeur du décompteur sera différente de 0. L'objectif, ici encore, est de programmer cette fonction monostable sous la forme d'un *component*. La figure 4 présente le chronogramme partiel du fonctionnement décrit ci-dessus.

Préparation

1. Proposer un schéma bloc pour ce *component timebase* précisant les entrées et les sorties mises en jeu.
2. Déterminer la valeur à charger dans le décompteur si on utilise l'horloge à 25MHz comme horloge pour le décomptage.
3. Proposer un schéma synoptique du FPGA, avec ce *component* instancié et les éléments matériel extérieurs au FPGA présents sur la carte XSA-50, pour tester cette fonctionnalité base de temps.
4. Expliquer en quoi le choix de l'horloge à 25 KHz comme horloge pour le décomptage serait plus intéressant pour implanter cette fonction base de temps. On rappelle que, quoi qu'il arrive, l'horloge à 25 KHz existe puisqu'elle est créée pour séquencer le grafcet.

Travail à faire en TP

1. Ecrire le programme VHDL de ce *component* en le plaçant dans votre librairie. Faire valider le principe du programme par l'enseignant.
2. Ecrire le programme VHDL permettant le test de cette fonctionnalité. Faire valider le principe du programme par l'enseignant.
3. Tester le fonctionnement et faire valider par l'enseignant.

Etape n° 3 : Synchronisation d'un écran VGA

Objectif Cette étape poursuit la programmation de la logique séquentielle synchrone sous forme de *process* et y ajoute la programmation de la logique combinatoire sous la forme de *process asynchrone*.

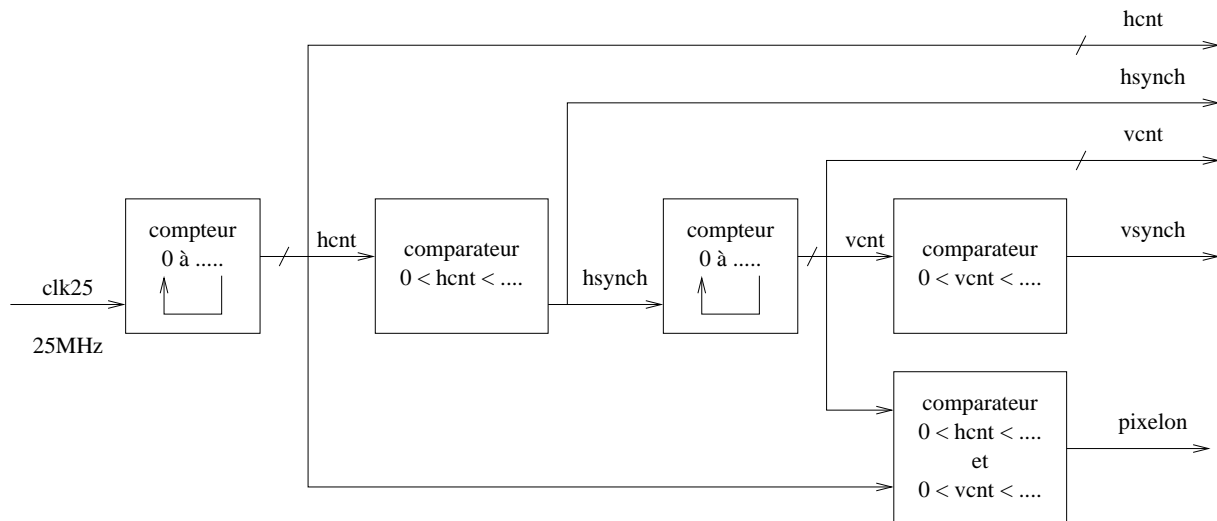


Figure 5: Schéma synoptique de la synchronisation VGA

VGA_Synchro Cette étape a pour but de générer les signaux nécessaires à la synchronisation d'un écran VGA. Il s'agit de permettre le balayage de l'écran par un spot, à la bonne fréquence, pour que l'écran s'illumine. L'écran sera piloté pour une résolution de 640x480 pixels avec un rafraîchissement de 60 images/seconde. De plus, cette étape s'attachera à générer des signaux supplémentaires pour piloter ensuite les signaux d'informations pour l'écran, c'est à dire la couleur que chaque pixel de l'écran doit prendre. Pour cela, l'annexe C présente le principe de fonctionnement d'un écran VGA et les caractéristiques de ses signaux de synchronisation. Ces caractéristiques restent les mêmes dans leur principe, quelle que soit la résolution choisie, mais les valeurs numériques des paramètres, elles, changeront. On développera donc un composant *vgasynchro*.

Préparation Le premier travail consiste à générer les signaux de synchronisation horizontale (*hsynch*) et verticale (*vsynch*). Pour cela, il faut, au cours du temps, générer un signal actif à l'état bas calibré selon les informations de l'annexe C. Le principe est de mettre en place un compteur qui va compter le temps (c'est à dire un nombre de coups d'horloge pixel d'après l'annexe C) correspondant à la durée exacte d'une ligne. Dès l'arrivée à l'instant de la fin de la ligne, le compteur rebouclera sur lui-même systématiquement pour enchaîner les lignes les unes après les autres. Ainsi en calculant les instants de début et de fin, du top de synchronisation horizontale, on pourra comparer ces instants avec la valeur du compteur précédent pour générer le top de synchronisation horizontale, signal que l'on nommera *hsynch*. Finalement, la gestion du signal de synchronisation horizontale nécessite seulement un compteur et un double comparateur.

De la même manière, le signal de synchronisation vertical sera généré en mettant en place un compteur. Cette fois, celui-ci comptera les lignes dans l'image pour définir la durée (exprimée en nombre de lignes) d'une image, conformément à l'information présentée en annexe C. Notons bien que pour compter les lignes, il faudra utiliser le signal de synchronisation horizontale (*hsynch*) comme signal d'horloge pour ce compteur. En comparant la valeur de ce compteur de lignes avec 2 valeurs qu'il faudra définir, on sera en mesure de générer le top de synchronisation verticale. La figure 5 présente le schéma synoptique de cette gestion de la synchronisation VGA.

1. En reprenant la description ci-dessus et en considérant les informations fournies dans l'annexe C, déterminer numériquement la valeur maximale du compteur pour le signal de synchronisation horizontale. En déduire la taille du compteur en nombre de bits pour supporter cette valeur maximale. On nommera le signal supportant la valeur de ce compteur : *hcnt*. Reprendre le schéma synoptique de la figure 5 et le compléter avec les valeurs numériques déterminées.
2. Nous considérerons que l'instant 0 d'une ligne se situe au début de la plage utile, c'est à dire que lorsque le compteur évoluera entre 0 et 639, il faudra gérer effectivement les pixels à allumer sur l'écran. Dans ces conditions, calculer les 2 valeurs numériques délimitant le top de synchronisation horizontale. Compléter le schéma synoptique.
3. Comme au 1., calculer la valeur maximale du compteur pour le signal de synchronisation verticale. On nommera le signal supportant la valeur de ce compteur : *vcnt*. Compléter le schéma synoptique.
4. Nous considérerons de nouveau que l'instant 0 d'une image se situe au début de la première ligne utile de l'image, c'est à dire que, lorsque le compteur évoluera entre les valeurs 0 et 479, il faudra gérer effectivement les lignes. Dans ces conditions, calculer les valeurs numériques délimitant le top

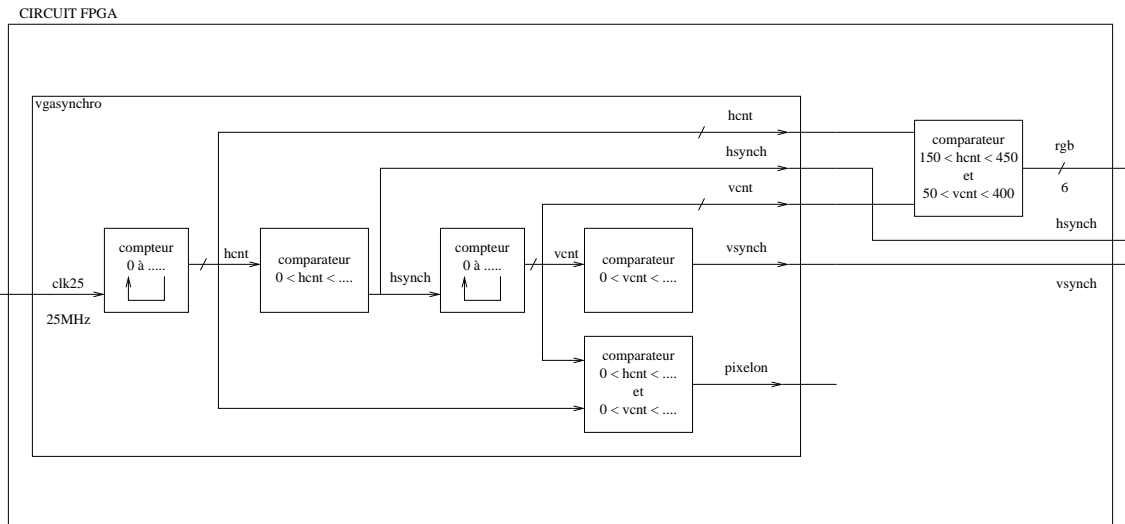


Figure 6: Schéma synoptique du test de la synchronisation VGA.

de synchronisation verticale et compléter le schéma synoptique.

- On souhaite ajouter à cet ensemble de fonctions la gestion d'un signal booléen (*pixelon*), indiquant à tout instant si un pixel doit être géré. En d'autres termes, il s'agit d'indiquer si les valeurs des compteurs (de pixels dans la ligne et de lignes dans l'image) correspondent à un pixel qui sera effectivement affiché à l'écran. Dans le cas contraire, cela signifie que l'on se trouve à un instant où le spot ne doit pas illuminer l'écran. Déterminer les valeurs du comparateur permettant de générer le signal *pixelon* et compléter le schéma synoptique.

On souhaite aussi délivrer les valeurs *hcnt* et *vcnt* en sortie de ce *composant vgasynchro*. En effet, à tout instant, ces 2 valeurs correspondent exactement aux coordonnées ($x \Leftrightarrow hcnt$ et $y \Leftrightarrow vcnt$) du pixel qui doit être géré pour peu que ces coordonnées soient dans la plage de valeurs (0, 0) (639, 479). On appellera ces coordonnées le *pixel courant*. Ceci est dû au choix judicieux de la position de la valeur 0 dans la ligne pour le compteur *hcnt* (instant du début de la zone active dans une ligne). Le principe est le même pour le 0 du compteur *vcnt*.

NB !!! Notez que ces coordonnées (*hcnt*, *vcnt*) du pixel courant seront très importantes dans la suite du projet car elles donnent, rappelons-le, la position du pixel courant à tout instant ce qui permet de gérer l'information vidéo (signal *rgb*) à envoyer à l'écran VGA.

Travail à faire en TP

- Créer le *composant vgasynchro* dans votre *library mylib* et décrire, sous la forme de 2 *process synchrone*, le fonctionnement des 2 compteurs (signaux *hcnt* et *vcnt*) dont le cycle correspond à la durée d'une ligne et à la durée d'une image. On utilisera les valeurs déterminées au 1. et au 3. Faire valider le principe du programme par l'enseignant.
- Ajouter dans ce composant les comparateurs gérant les signaux *hsynch* et *vsynch*. Faire valider le principe du programme par l'enseignant.
- Ajouter ensuite le comparateur générant le signal *pixelon*. Faire valider le principe du programme par l'enseignant.
- Le composant *vgasynchro* prêt, générer un projet conforme au schéma synoptique de la figure 6 en terme d'entrées/sorties et placer le composant *vgasynchro*. Ajouter ensuite un process asynchrone pilotant le signal *rgb* à "001100" quand $150 < hcnt < 450$ et $50 < vcnt < 400$. Sinon, le signal *rgb* vaudra "000000". Faire valider le principe du programme par l'enseignant.
- En vous appuyant sur le tableau 1, créer le fichier .ucf d'attribution des signaux aux broches du FPGA et compiler le programme.
- Tester le fonctionnement de l'application et faire valider par l'enseignant.

Etape n° 4 :

Affichage d'un nombre sur 4 digits sur écran VGA

Objectif Il s'agit ici de mettre en place l'affichage complet d'un nombre sur 4 digits sur l'écran VGA. L'exercice se limitera bien sûr à une valeur qui sera fixe pour le test. Cela nécessitera l'utilisation correcte de la description de plusieurs *composant* d'une librairie fournie à savoir *tachylib*. Cette étape conduira les étudiants à découvrir, entre autres, les *generic* et les *generic map*, les *integer* ... Le travail se "limitera" donc à l'instanciation correcte de *composant* existants. Il faudra cependant en comprendre globalement la description pour en faire une bonne utilisation.

Préparation La librairie *tachylib* fournie décrit un certain nombre de *component* pour gérer efficacement l’affichage d’un afficheur 7 segments sur un écran VGA. **Le travail de préparation consistera tout d’abord à analyser aussi finement que possible la manière dont sont décrits les différents component constituant cette librairie.** Elle décrit en particulier :

- le component *rec* qui permet de dessiner à l’écran un rectangle, en définissant comme paramètres d’entrée, sa position et sa taille respectivement *x,y et dx,dy*, Ses couleurs (*color* et *fond*) qui seront utilisées pour s’afficher en fonction de son état (allumé ou éteint) par le biais d’un bit *onoff* . Il fournit en sortie le signal *rgb* directement utilisable sur le port VGA de la carte. Ce *component* a besoin des coordonnées du pixel courant à tout instant, d’où les signaux *hcnt* et *vcnt* en entrée. Une instantiation de ce component *rec* correspondra au dessin d’un segment, ce qui justifie donc le fait que le rectangle puisse être allumé ou éteint par le bit *onoff* , avec, pour chaque état, une couleur.
- le component *pilot7seg* correspond à l’exercice de l’étape n°1. Il fournit les informations concernant l’allumage (ou non) des segments. En d’autres termes, les 7 sorties de ce component sont à relier directement aux 7 entrées *onoff* des rectangles qui seront instanciés pour dessiner un afficheur 7 segments.
- le component *aff7segvga* dessine un afficheur 7 segments complet sur l’écran. Il utilise donc 7 instantiations du component *rec* et une instantiation du component *pilot7seg* . Du point de vue de ces entrées, on notera que l’on place l’afficheur 7 segments sur l’écran en définissant sa position et sa taille (*x,y,dx et dy*). Dans ces conditions, la position et la taille de chaque rectangle (segment) est calculée automatiquement à partir de ces valeurs. Les autres entrées sont la couleur, la valeur du nombre BCD à afficher et les coordonnées du pixel courant. La sortie se résume simplement au signal *rgb* à transmettre à l’écran. C’est ce *component* qu’il faudra instancier pour obtenir un afficheur 7 segments à l’écran.

Remarque : Notons enfin que chaque *component rec* fournit un signal *rgb* à lui seul. Il est donc nécessaire de "mixer" tous ces signaux *rgb* puisque l’écran VGA n’en reçoit qu’un seul. Ce mélange des signaux *rgb* se fait par un simple *ou logique* réalisé entre tous les bits de même rang générés par les divers *component rec* . Il y a donc 6 *ou logique* à mettre en place pour mixer toutes les composantes des signaux *rgb* existants. Un *process* utilisant un style de programmation VHDL efficace permet une description concise de cette fonction de mixage.

1. Faire le schéma synoptique de l’application permettant de tester l’affichage d’un nombre de 4 digits sur l’écran VGA. On considérera que les valeurs de 3 digits sont fixées égales à 9. Le dernier digit verra sa valeur pilotée par les 4 switch disponibles sur la carte XSA-50. Le travail de l’étape n°3 pourra être repris en grande partie et modifié pour correspondre à la nouvelle application. On pourra aussi s’appuyer sur le schéma synoptique de l’application complète de la figure 2
2. Préparer un tableau de correspondance entre les signaux et les broches du FPGA (fichier .ucf).

Travail à faire en TP

1. Démarrer un nouveau projet pour tester cette application d’affichage de nombres sur écran VGA. Rappelons que le travail de l’étape n°3 pourra être repris en grande partie. Instancier, dans un premier temps, un seul afficheur 7 segments. On le placera aux coordonnées $x=160$ $y=60$ avec les dimensions $dx=80$ $dy=120$. On choisira une couleur quelconque mais visible sans ambiguïté ! Tester le fonctionnement et faire valider par l’enseignant.
2. Reprendre l’application en l’étendant à 4 digits. On pourra prendre les coordonnées et les dimensions suivantes :
 - pour le digit des milliers : $x=160$ $y=60$ $dx=80$ $dy=120$
 - pour le digit des centaines : $x=280$ $y=60$ $dx=80$ $dy = 120$
 - pour le digit des dizaines : $x=400$ $y=60$ $dx=80$ $dy = 120$
 - pour le digit des unités : $x=520$ $y=60$ $dx=80$ $dy=120$

Dans cette deuxième application, on ne pourra oublier de mélanger correctement les signaux *rgb* de chacun des 4 digits en sortie.

Tester et faire valider par l’enseignant.

3. Quelle(s) remarque(s) pouvez-vous faire sur la qualité de l’image sur l’écran VGA ? Proposer une(des) explication(s).
4. Proposer une(des) solution(s) pour remédier au(x) défaut(s) constaté(s). Faire valider vos idées par l’enseignant.

Etape n° 5 :

Compteurs BCD cascadables

Objectif Cette étape a pour but d'aborder de manière un peu plus poussée les *process*. On s'attache alors à décrire un *composant* de type compteur mais cette fois avec des caractéristiques précises et plus contraignantes qu'à l'habitude. Il s'agit de faire la description d'un compteur BCD cascadable, proche de ce qui existe en circuits logiques discrets sous la forme du compteur/décompteur 74191. Nous ne nous attarderons pas sur ce compteur 74191 en particulier dans la mesure où notre compteur restera légèrement différent. On pourra là encore s'appuyer sur le schéma synoptique de l'application complète de la figure 2

Préparation Les caractéristiques de notre *composant compteur* sont les suivantes :

- *clk* : entrée d'horloge pour faire évoluer le compteur ;
 - *raz* (remise à zéro) : entrée *active à l'état haut*, de remise à 0 *synchrone* du compteur quels que soient les états des autres entrées (*raz* est donc prioritaire).
 - *ctenbar* (counter enable bar) : entrée *active à l'état bas* autorisant le compteur/décompteur à évoluer à chaque front montant de l'horloge *clk*. Si *ctenbar=1* le compteur reste *bloqué* à sa valeur, si *ctenbar=0* le compteur *évolue* en fonction des autres entrées. Cela suppose que le signal *raz* soit à 0 puisqu'il est prioritaire.
 - *dbaru* (down bar / up) : entrée de sélection de mode de fonctionnement du compteur. Si *dbaru=0*, le compteur *décompte* en rebouclant sur lui-même. Si *dbaru=1*, le compteur *compte* en rebouclant sur lui-même ;
 - *max* : valeur en entrée indiquant la valeur maximale atteinte par le compteur/décompteur. Exemple : si *max="111"*, le compteur évoluera de 0 à 7 avant de retourner à 0 en mode compteur et décomptera de 7 à 0 avant de retourner à 7 en mode décompteur. *max* sera un vecteur dont la dimension dépendra de sa valeur elle-même.
 - *q* : vecteur en sortie supportant la valeur courante du compteur. Il sera dimensionné en fonction de la valeur maximale autorisée (*max*) donc de la même dimension que *max*.
 - *enoutbar* (enable out bar) : sortie *asynchrone active à l'état bas* permettant de piloter un compteur placé en cascade. Pour cela, il faudra notamment connecter l'entrée *ctenbar* du compteur placé en cascade à cette sortie et fournir le même signal d'horloge au 2 compteurs. Si *enoutbar=0*, le *compteur placé en cascade sera autorisé à évoluer* sur front d'horloge sinon, il restera bloqué.
1. Pour un compteur BCD, donner la valeur de *max*. Dimensionner alors les vecteurs *max* et *q*.
 2. Donner le schéma synoptique du composant *compteur*.
 3. Le signal *enoutbar* doit être activé dans 2 situations. Donner sous la forme d'une table "si-alors" le fonctionnement complet de cette sortie.
 4. En reprenant le travail de l'étape précédente, proposer un schéma synoptique intégrant un *composant compteur* pour tester son fonctionnement. Proposer notamment la manière de piloter les entrées de ce *composant* qui permettront un test à un rythme maîtrisé.

Travail à faire en TP

1. En vous appuyant sur un *process* auquel vous associez une structure de programmation VHDL *if then else* (éventuellement plusieurs structures *if then else* imbriquées), décrire le fonctionnement de ce composant *compteur* en l'intégrant dans votre librairie *mylib*. Faire valider le principe de la description par l'enseignant.
2. Refaire un projet en récupérant le fichier VHDL de l'étape précédente. Modifier alors ce fichier source pour y intégrer un *compteur*.
3. Tester le fonctionnement et faire valider par l'enseignant.
4. Modifier le programme pour instancier 4 *compteur* en cascade. Pour le test, on choisira un signal d'horloge externe dont la fréquence sera adaptée au test (les impulsions issues du moteur à courant continu pourraient convenir dès lors que le moteur serait réglé à une vitesse assez faible.
5. Tester le fonctionnement et faire valider par l'enseignant.

Conclusion à ce niveau de progression du projet A ce niveau d'avancement du TP, la *partie opérative de l'application tachymètre* est quasiment complète. Il ne reste en réalité qu'à intercaler un *module de mémorisation* de l'information issue des 4 *compteur* conformément au schéma synoptique présenté à la figure 2. Ce module de mémorisation est décrit dans la librairie *tachylib* fournie. Il sera nécessaire de l'ajouter à la description de la partie opérative lors de la description de l'application complète.

Etape n° 6 : Application complète

Partie commande

Objectif Cette étape permettra d'aborder la programmation de grafset et des aspects plus théoriques sur la conception électronique de système automatisé complet. En particulier, pour implanter la partie

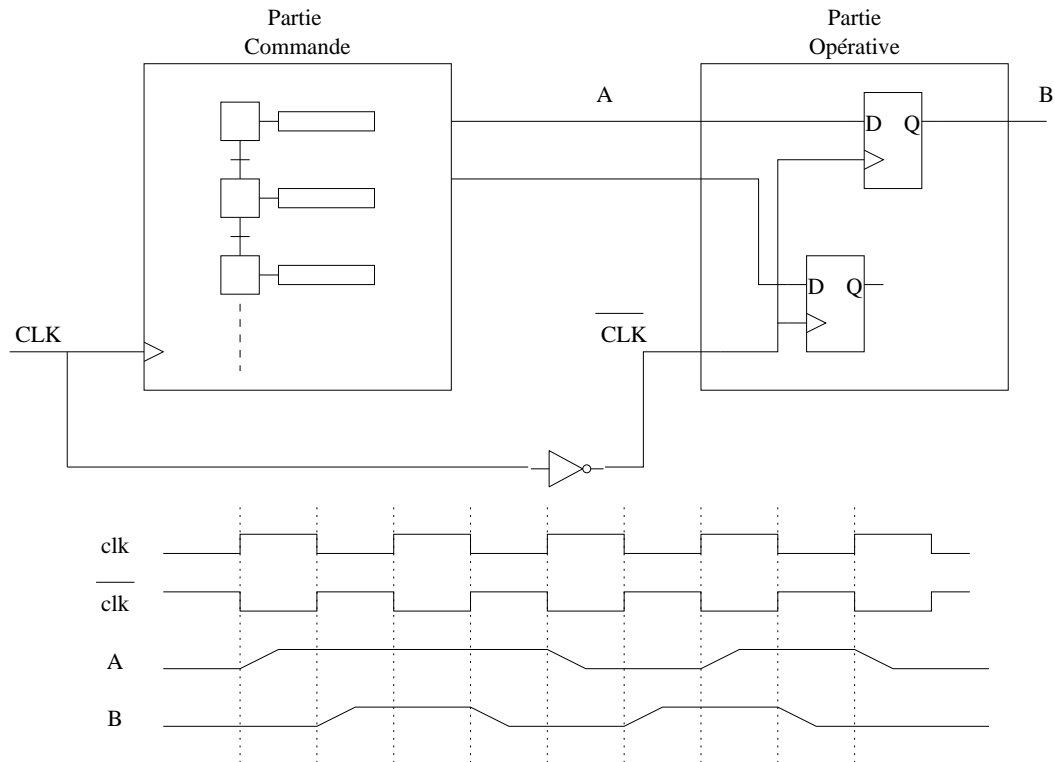


Figure 7: Illustration de la synchronisation des parties commande et opérative d'un système.

opérative et la partie commande d'une application, toutes deux sensibles à une horloge, on s'attachera à gérer le fonctionnement de l'une sur front montant et l'autre sur front descendant de la même horloge. Ainsi, la partie commande positionnera des signaux à un instant donné sur le front de l'horloge et la partie opérative exploitera le niveau de ces signaux une demi-période plus tard lorsque les signaux seront *effectivement positionnés et surtout stabilisés* (l'établissement d'un signal à un niveau n'est pas instantané du simple fait du temps de transfert de l'information au travers des couches logiques de la fonction qui le gère). Le chronogramme de la figure 7 présente graphiquement ce principe. Ainsi, on rythmera le fonctionnement du grafcet avec le front montant de l'horloge *clkgrafcet* générée par le diviseur de fréquence développé à l'étape n°2, c'est à dire le *composant clkgraf*. Et on gèrera donc tous les modules de la partie opérative avec le front descendant de cette même horloge *clkgrafcet*. Cela correspond au schéma synoptique de l'application complète présenté à la figure 2.

Préparation Il s'agit ici de préparer l'application complète en s'intéressant tout d'abord au grafcet qui constituera la partie commande.

1. En premier lieu, calculer de manière littérale la vitesse en tours/minute correspondant à une mesure de n impulsions pour une durée de mesure d'une seconde. On rappelle que le disque est percé de 60 trous.
2. Lister l'ensemble des signaux concernés par la partie commande, notamment en les repérant dans la figure 2
3. Envisager et proposer un grafcet (6 étapes) pilotant la mesure de la vitesse. On rappelle qu'il faudra dans le cycle de mesure :
 - mémoriser la valeur des compteurs BCD (la mesure) pour afficher une valeur stable de la mesure sur l'écran VGA ;
 - remettre à zéro les compteurs BCD pour préparer la prochaine campagne de mesure de 1s ;
 - mesurer la vitesse : c'est à dire incrémenter le compteur à chaque fois qu'une impulsion se présente. Cela nécessite la détection du passage à 1 pour autoriser l'incrément, puis la détection du passage à 0 pour, ensuite, attendre le passage à 1 suivant. Cela sera de plus conditionné par un niveau logique 1 sur le signal *BT* généré par le module développé à l'étape n°2.
4. Proposer un schéma synoptique permettant de tester le fonctionnement du grafcet seul en sortant, sur les leds de l'afficheur 7 segments de la carte XSA-50, le niveau logique des étapes du grafcet.

Travail à faire en TP

1. Programmer le grafcet "seul" avec le style de programmation *équations de récurrence*. Il faudra en réalité intégrer les modules développés notamment dans l'étape n°2 pour disposer des signaux

BT et *clkgrafcet* .

2. Tester le fonctionnement et faire valider par l'enseignant.

Chaîne de mesure complète

Objectif Il s'agit ici d'intégrer tous les modules (*component*) développés au cours des 5 étapes initiales de manière à disposer de la chaîne de mesure complète avec le pilotage de la partie opérative avec la partie commande développée précédemment. Cela doit permettre l'affichage de la vitesse en tours/minute sur l'écran VGA avec un rafraîchissement à peu près à chaque seconde.

Préparation

1. Reprendre le schéma synoptique de la figure 2 et vérifier que tous les modules sont disponibles en librairie (*tachylib* et *mylib*).

Travail à faire en TP

1. Développer le *component mémo* correspondant à la mémorisation de la valeur mesurée, s'il n'a pas encore été développé. Faire valider le principe du programme par l'enseignant sans le tester matériellement.
2. Développer le *component rgbsynchro* permettant de synchroniser le signal *rgb* par le signal d'horloge *clkvideo* (question 4. du travail à faire en TP dans l'étape n° 4). Faire valider le principe du programme par l'enseignant sans le tester matériellement.
3. Développer l'application complète de la chaîne de mesure conformément au schéma synoptique de la figure 2.
4. Tester le fonctionnement et faire valider par l'enseignant.

HOWTO : logiciel ISE WebPack avec carte FPGA XSA-50

- Objectifs :
 - aider à utiliser le logiciel de développement des FPGA de Xilinx
 - aider à l'implantation de l'application sur la carte de démonstration : XSA50 board
- 1. Projet :
 - Démarrer le logiciel avec l'icône sur le bureau (ou bien `c : \xilinx_webpack\bin\nt\webpack.exe`)
 - créer un nouveau projet : **File -> new -> project**
 - **project location** : choisir un répertoire cible (ex : `c : \titi`)
 - **project name** : choisir le nom du projet (ex : `toto`) NB!!! ceci va créer un répertoire `toto` dans `c : \titi`
 - **device family** : Spartan 2 (circuit de la carte XSA-50)
 - **device** : XC2S50-5tq144 (le modèle de Spartan 2 présent sur la carte XSA-50)
 - **design flow** : XST VHDL (langage de développement) (NB!!! a priori, dans le cas où on veut faire du schéma, faire tout de même XST VHDL puis il faudra générer des sources de type schéma qui seront, de toute manière, transformées en VHDL. a priori car non testé!!!)
 - faire OK
 - le projet est créé et on doit voir dans une fenêtre en haut à gauche le nom du projet et le circuit ciblé
 - ajouter des sources :
 - si les fichiers VHDL sont déjà prêts. Il faut les placer dans le répertoire `c : \titi\toto` puis faire **project -> add source** et ajouter les fichiers. A priori, le fichier "top level" , c'est à dire celui qui décrit le circuit réalisant l'application globale est à sélectionner comme fichier VHDL MODULE. Pour tous les autres fichiers de description de composant ce sont des fichiers VHDL PACKAGE. Dans le cas d'une librairie, cela reste valide, on ajoute le fichier en tant que VHDLPACKAGE
 - si les fichiers VHDL ne sont pas prêts. Il faut faire **file -> new** et cela ouvre dans une fenêtre en haut à droite un fichier texte pour générer un fichier VHDL. Faire immédiatement **file -> save as** et enregistrer le fichier avec l'extension `.vhd` dans le repertoire du nom de projet (`c : \titi\toto`) pour bénéficier de la coloration syntaxique. Mais il ne sera pas ajouté automatiquement comme source du projet. Donc il faudra revenir à l'alinéa ci-dessus pour ajouter les sources vhd dans le projet.
- 2. Compilation
 - sélectionner par simple clic le fichier VHDL "top level" dans la fenêtre du projet
 - dans la fenêtre **process view** (à gauche au milieu) une arborescence propose 4 étapes : soit on les regarde de près pour bien comprendre ce que l'on fait, soit on va directement à la compilation pour faire un double clic sur **generate programming file** .
 - **Design Entry utilities** : faire **User constraints -> Edit Implementation Constraints File (NB!!! ne pas faire Edit Implementation Constraints File (Constraints Editor))**
=> génère un fichier `.ucf` dont le nom est celui de l'entité du fichier "top level" . Ce fichier `.ucf` (User Constraint File) permet d'attribuer des broches particulières aux divers signaux qui entrent ou sortent du FPGA lorsque le câblage de la maquette est déjà réalisé (ce qui est le cas le plus souvent). Le fichier est généré avec beaucoup de commentaires indiquant toutes les contraintes qu'il est possible de donner et leur syntaxe associée. La syntaxe pour cette contrainte est :
NET <nom du signal> LOC=<numéro de la broche> ;
(ex : `NET "clk" LOC=P88 ;`) NB!!! le numéro de la broche a bien souvent une notation particulière comme le montre l'exemple proposé ci-avant, ce n'est pas simplement 88 mais P88. En fonction du circuit ciblé, et plus particulièrement le modèle du boîtier (PGA,BGA, PQFP, PLCC ...), cette appellation des broches peut changer. Dans le cas des signaux rassemblés au sein d'un vecteur, il faut considérer chaque bit du vecteur un par un. La syntaxe pour définir un signal d'un vecteur dans un fichier `.ucf` est : `NET "din<3>" loc = "p25"` ;
Lorsque le fichier `.ucf` est mis à jour, l'enregistrer de préférence dans le répertoire du projet (ex : `c : \titi\toto`). Ainsi, on le retrouvera automatiquement au moment de le prendre en compte pour les phases de compilation/placement/routage.
 - **Synthesize** : pointer à la souris sur ce mot-clé, sélectionner par simple clic gauche puis cliquer bouton droit de souris => menu déroulant où on va chercher **properties**.
Cela fournit une fenêtre de réglages des options de compilation des sources vhd disponibles via plusieurs onglets. A priori, laisser toutes les options déjà sélectionnées par défaut. Aller tout de même observer ce qui peut être réglé à ce niveau par curiosité (ex : stratégie de codage des états des Finite State Machine...).
 - **Implement design** : pointer à la souris sur ce mot-clé, sélectionner par simple clic gauche puis cliquer bouton droit de souris => menu déroulant où on va chercher **properties**.
Cela fournit une fenêtre de réglages des options de placement/routage de l'application sur le circuit via plusieurs onglets.
 - onglet **Translate properties -> Implement User Constraint File** : choisir le fichier `.ucf` généré précédemment.

- onglet **Map properties** :
 - sélectionner l'option **Trim unconnected Signals** (permet de détruire la logique en amont d'un signal qui n'est, au final, pas utilisé)
 - sélectionner l'option **Pack I/O registers into IOBS** (permet d'exploiter les bascules D des blocs d'entrée/sortie pour synchroniser les sigaux en entrée ou en sortie, et ainsi, économiser les bascules D dans les CLB du FPGA)
 - onglet **Place and route** : laisser les options sélectionnées par défaut
 - onglet **Post place and route Static Timing Report** : laisser les options sélectionnées par défaut
 - onglet **Simulation model properties** : laisser les options sélectionnées par défaut
 - onglet **Post map static timing report properties** : laisser les options sélectionnées par défaut
- **Generate programming file** : dans l'onglet **general options**, sélectionner les options **Design rule checker** et **create bit file** a priori, laisser les autres onglets avec les options sélectionnées par défaut.
- Synthèse et compilation : dans la fenêtre **process view**
 - soit dans l'ordre et un par un :
 - double clic sur **Synthesize** puis laisser faire...
 - double clic sur **Implement Design** puis laisser faire....
 - double clic sur **Generate Programming File** puis laisser faire...
 - soit directement :
 - double clic sur **Generate Programming File** puis laisser faire...

Deux possibilités :

- si tout s'est bien passé, chaque étape se termine par **successfull**. On dispose alors d'un **fichier .bit dont le nom est celui de l'entité du fichier "top level"**, c'est à dire l'entité correspondant au circuit complet. On pourra éventuellement retrouver dans les branches de l'arborescence de la fenêtre **process view** les fichiers de rapport des différentes étapes qui ont été exécutées pour en extraire des informations (location effective des signaux sur les broches, pourcentage d'utilisation du circuit, et beaucoup d'autres choses)
- si tout ne s'est pas bien passé, le logiciel refusera d'exécuter les étapes suivant celle qui a échoué. On peut alors trouver, dans la fenêtre tout en bas de l'écran, l'historique des messages fournis par le logiciel et notamment :
 - les **warnings** signifiés par un **smiley jaune**
 - les **erreurs** signifiées par un **smiley rouge**
 Et là commencent les difficultés du débogage!!!!

3. Carte XSA-50 et son environnement

- carte XSA50 : voir la documentation fournie avec le logiciel. Elle est disponible en ligne sur le PC via le fichierpdf

A noter, l'utilisation de l'oscillateur programmable DS1075 pour obtenir une horloge générale de synchronisation pour faire fonctionner le FPGA. Cette horloge est de fréquence 100MHz à la base et peut être divisée par un nombre entier compris entre 1 et 2052 soit une fréquence délivrée variant entre 100 MHz et environ 47KHz.
- environnement : celui de l'application en termes de connexions avec des signaux ou des périphériques extérieurs, à savoir :
 - port parallèle du PC de développement connecté à la carte XSA50 pour le téléchargement de l'application ;
 - alimentation de la carte XSA par une fiche jack (alim. 9V DC maximum) ;
 - connexions par fils wrappés sur les broches de la carte XSA50 donnant accès à certaines broches du FPGA. connexions permettant d'entrer ou sortir des informations
- Téléchargement :

Lorsque le fichier .bit est prêt :

 - Eventuellement, programmer l'oscillateur programmable pour avoir l'horloge à 100MHz divisée par une valeur correcte en fonction des besoins de l'application. NB!!! En principe, cela est déjà fait sur la maquette câblée!!! Sinon, il faut démarrer GXSETCLK, sélectionner **XSA-50** et **lpt1** puis choisir le nombre diviseur (entre 1 et 2052) . Lancer et suivre les instructions présentées dans les fenêtres de message.
 - télécharger l'application : lancer **GXSLOAD**, sélectionner **XSA-50** et **lpt1**. puis faire un drag and drop (glisser et lâcher) depuis l'explorateur windows, le **fichier .bit** dans la fenêtre intitulée **FPGA/CPLD**. Faire **LOAD** . En principe tout se passe bien en moins de 5 secondes et quand tout est fini, faire **EXIT**. **Le FPGA est configuré avec succès et fonctionne dès que la configuration est terminée.** Faire éventuellement une manoeuvre d'initialisation de l'application (bouton init, si cela a été prévu dans le développement de l'application) et faire fonctionner l'application

Annexe B

Création d'une librairie et de component

Le principe de la librairie est, classiquement, de rassembler au sein d'un fichier l'ensemble des fonctionnalités dont on peut avoir besoin régulièrement. Le code source qui les décrit est alors écrit une seule fois et on réutilise le fichier à chaque application. Ce principe s'applique bien sûr au cas de projet en VHDL. L'idée sera alors de décrire des composants dans la librairie et d'utiliser ces composants dans l'application. En pratique, il y aura donc 2 fichiers VHDL source dans nos applications qui restent des projets de petite taille.

L'écriture d'une librairie s'organise de la manière suivante :

- Appel des librairies classiques de l'IEEE pour nos applications ;
- Mise en place d'un package (éventuellement plusieurs) où l'on déclare l'ensemble des composants qui seront décrits par la suite. Il s'agit de déclarer le component d'un point de vue extérieur, c'est à dire en termes d'entrées et de sorties principalement. On reproduit donc à l'identique le port de l'entity dans cette déclaration du component.
- A la suite du (des) package(s), on place la description proprement dite de chaque component, c'est à dire une entity qui doit donc être la même que le port proposé dans le package et une architecture qui décrit le comportement du component.

Un exemple est donné ci-dessous pour bien comprendre cette organisation et préciser en même temps quelques détails de la syntaxe.

```
-- appel des librairies classiques de l'IEEE pour nos applications
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
package portes is -- début de notre package portes
-- porte ou a 2 entrées
component ou2 is port (
    e1: in std_logic;
    e2: in std_logic;
    s: out std_logic
);
end component;
-- porte et-non a 2 entrées
component etnon2 is port (
    e1 : in std_logic;
    e2: in std_logic;
    s: out std_logic
);
end component;
end portes; -- fin de notre package portes
-- début de la description des component de notre package portes
-- description des composants (entity et architecture) dans n'importe quel ordre
-- il est conseillé de rappeler les librairies classiques de l'IEEE pour chaque component
-- l'aspect obligatoire de cet appel systématique serait à vérifier !!!
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- l'entity du component ou2
entity ou2 is port (
    e1: in std_logic;
    e2: in std_logic;
    s: out std_logic
);
end ou2;
-- puis l'architecture du component ou2
architecture behavior of ou2 is
begin
```

```

    s <= e1 or e2;
end behavior;
-- etc etc pour les autres component
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity etnon2 is port (
    e1 : in std_logic;
    e2: in std_logic;
    s: out std_logic
);
end etnon2;
architecture behavior of etnon2 is
begin
    s <= not ( e1 and e2 );
end behavior;

```

Lorsque la librairie est décrite, on souhaite bien sûr l'utiliser. Il suffit :

- d'intégrer le fichier source de cette librairie dans le projet : **project -> add sources** puis sélectionner le fichier.vhd . Le logiciel propose alors d'intégrer le fichier comme **module**, **package**, ... On sélectionnera alors **package** . On termine la procédure d'intégration et le fichier doit apparaître dans la fenêtre de gestion du projet (en haut à gauche).
- de faire appel à cette librairie : ajouter simplement un appel de librairie classique au début du fichier VHDL "top level"
- puis utiliser le component directement dans l'architecture.

Un exemple de code source d'un fichier "top level" utilisant des **component** est donné ci-dessous pour faciliter la compréhension. On prendra un exemple où l'on place les components décrits ci-avant dans l'exemple du **package portes**.

```

library IEEE; -- l'appel classique
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library work; --l'appel de notre librairie
use work.portes.all; -- appel du package portes de notre librairie, et avec .all,
-- on autorise tout ce qui a été décrit dans le package portes
entity exemple_lib is port (
    a1,a2,a3 : in std_logic;
    s : out std_logic_vector(1 downto 0)
);
end entity;
architecture behavior of exemple_lib is
    signal alora2 : std_logic; -- signal intermédiaire, interne au FPGA,
    -- qui reçoit une sortie d'un component pour être réutilisé ensuite
begin
    first_gate: ou2 port map (e1 => a1, e2 => a3, s => alora2 );
    -- le component peut piloter un signal interne ...
    second_gate: ou2 port map (e1 => a2, e2 => a3, s=>s(1));
    -- ou bien une sortie du FPGA ...
    third_gate: etnon2 port map ( e1 => a3, e2 => alora2, s=>s(0));
    -- ici on réutilise le signal interne comme entrée d'un autre component
end behavior;

```

Conclusion : La figure présente le schéma synoptique de la description **exemple_lib**. Les intérêts de cette méthode de description, à base de component dans une librairie, sont les suivants :

- cela rend le code source du fichier top level relativement court même lorsqu'il s'agit d'une application conséquente.

- cela permet de développer l'application avec une approche *modules à connecter entre eux*. Cela facilite :
 - un travail d'équipe puisque l'on peut répartir le développement des modules entre différents programmeurs (structuration, planification... du travail) ;
 - le test (simulation ou test réel) des modules un par un, pour les valider avant de les utiliser dans l'application globale ;
 - le développement de l'application, car cela contraint le(s) programmeur(s) à structurer leur travail de développement de l'application, en les obligeant à réfléchir sur le découpage de l'application en fonctionnalités .

Ce style de description VHDL porte le nom de *description structurelle*. Cela correspond bien à une structuration assez forte de l'application pour décrire des composants que l'on viendra connecter entre eux par la suite.

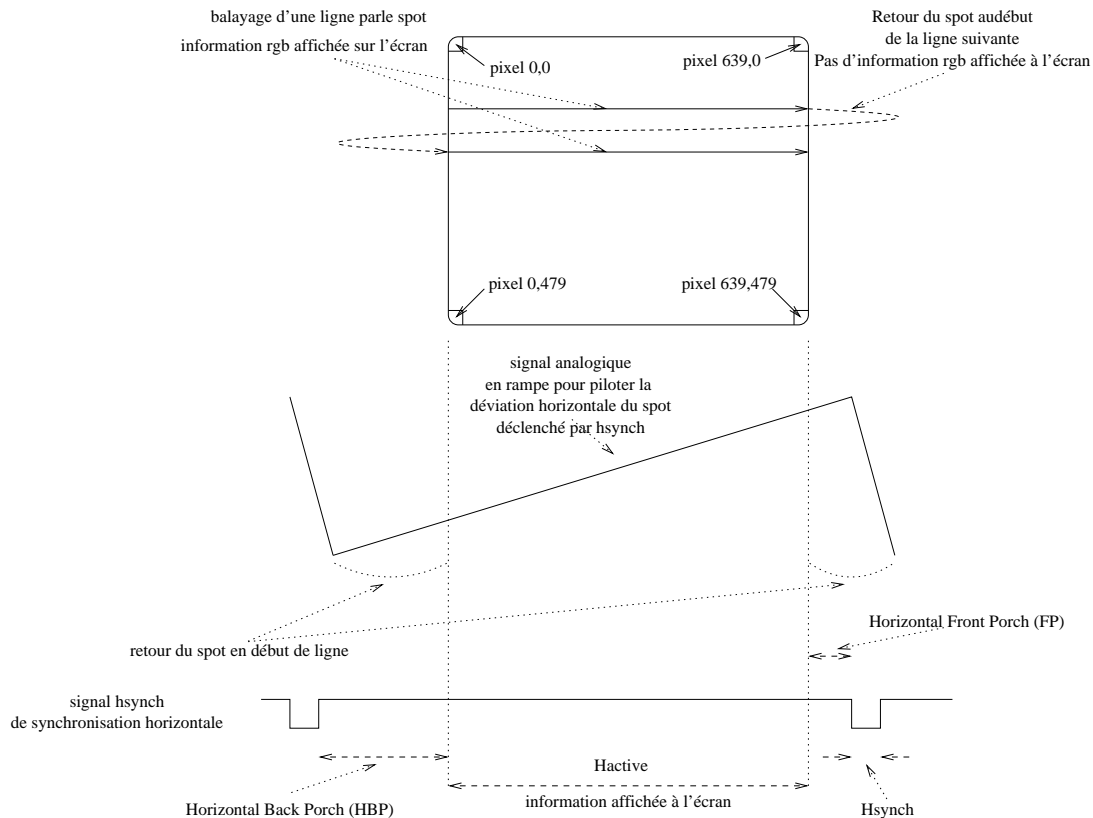


Figure 8: Illustration du balayage d'un écran VGA

Annexe C

Caractéristiques des signaux de synchronisation pour un écran VGA

Avant-propos

Ce paragraphe est largement inspiré de plusieurs documents. Il s'agit du *Spartan-3 Starter Kit Board User Guide* et du *Xilinx University Program Virtex-II Pro development System Hardware Reference Manual*. Ces documents sont disponibles pour un téléchargement libre au format .pdf aux adresses respectives :

- <http://www.digilentinc.com/Products/Detail.cfm?Nav1=Products&Nav2=Programmable&Prod=S3BOARD> et cliquer sur le lien **Reference Manual** dans le cadre **Links** pour obtenir le fichier .pdf correspondant ;
- <http://www.digilentinc.com/Products/Detail.cfm?Nav1=Products&Nav2=Programmable&Prod=XUPV2P> et cliquer sur le lien **Reference Manual** dans le cadre **Links**.

En particulier, les pages 21 à 25 du premier document expliquent le fonctionnement de l'écran, ce qui justifie parfaitement les caractéristiques des signaux de synchronisation au moins pour le principe. Le second document (pages 34 à 39) reprend essentiellement les durées des différentes parties des signaux de synchronisation et cela pour de nombreuses résolutions d'un écran VGA. L'ensemble de ces informations est résumé au mieux ci-dessous. En cas de besoin, les informations complètes pourront être retrouvées dans les références données.

Le principe d'un signal vidéo repose sur le balayage de toute la surface de l'écran par un spot, qui illumine le point (pixel) de l'écran où il se trouve. La rémanence de l'écran associée à la persistance rétinienne donne alors l'impression que tout l'écran est allumé en permanence.

Le balayage de l'écran par le spot se fait ligne par ligne. Cela implique qu'à chaque fin de ligne le spot doit retourner au début de la ligne suivante. Ce retour à la ligne se traduit dans le signal vidéo par 2 phases :

- l'une où le spot illumine effectivement l'écran ;
- et une autre pendant laquelle le spot revient en début de ligne et par conséquent n'illumine pas l'écran.

La figure 8 présente ce principe.

Pour gérer un écran, il faut donc lui fournir un signal de synchronisation lui permettant de déclencher le retour à la ligne. Ce signal doit être calibré en fonction de la résolution de l'écran (nombre de pixels par ligne). On l'appelle signal de synchronisation horizontale (*hsynch*).

De la même manière, le spot, lorsqu'il arrive en bas de l'écran doit remonter en haut pour afficher l'image suivante et ainsi maintenir l'écran allumé en permanence. Le principe est alors exactement le même que pour le signal de

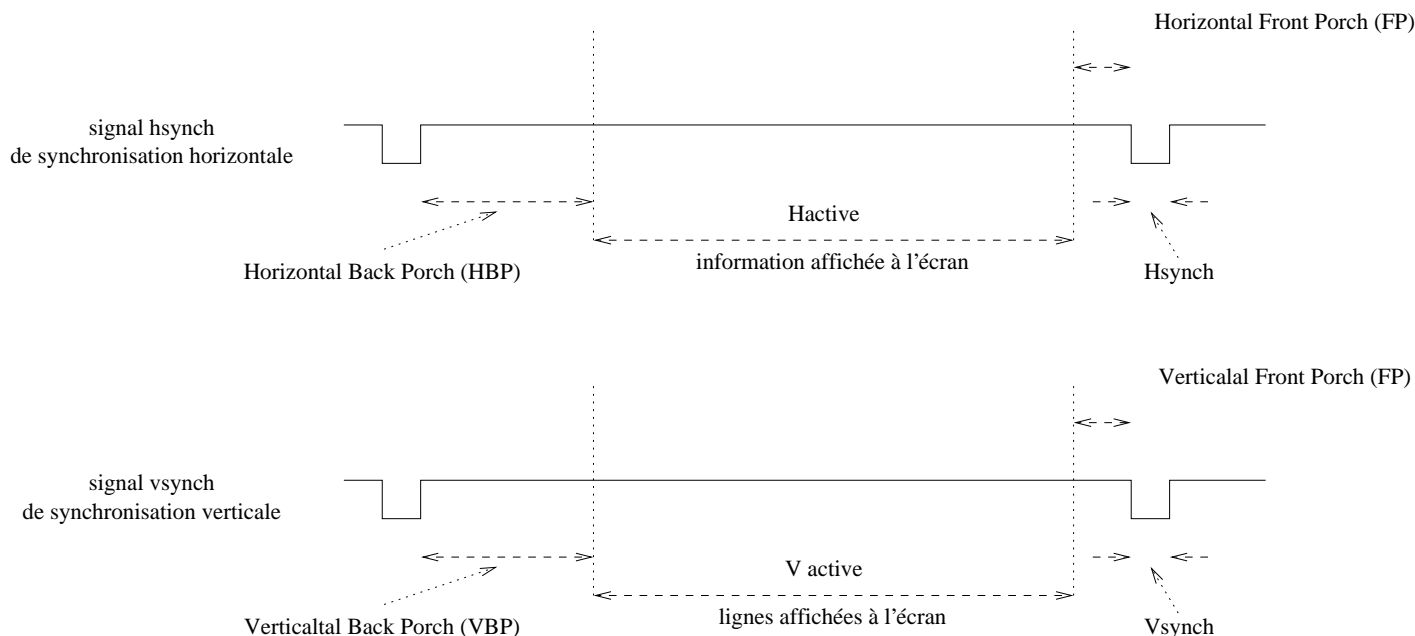


Figure 9: Signaux de synchronisation pour un écran VGA

Output format	pixel clock MHz	Horizontal Timing Parameters				
		Hactive pixels	HFP pixels	HSYNCH pixels	HBP pixels	Htotal pixels
640x480@60Hz	25	640	16	96	48	800
800x600@75Hz	50	800	16	80	168	1064

Output format	Hsynchro clock MHz	Vertical Timing Parameters				
		Vactive lignes	VFP lignes	VSYNCH lignes	VBP lignes	Vtotal lignes
640x480@60Hz	25/800	480	9	2	29	520
800x600@75Hz	50/1064	600	1	2	23	626

Table 2: Tableaux de valeurs pour les signaux de synchronisation d'un écran VGA à 2 résolutions

synchronisation horizontale. On a alors un signal de synchronisation verticale (*vsynch*) qui se caractérise par une durée permettant le balayage de l'écran du haut en bas et une deuxième phase où le spot remonte du bas vers le haut. Ce second signal aura exactement la même allure que le signal *hsynch*, mais avec des durées différentes.

Les signaux de synchronisation horizontale et verticale ont alors des caractéristiques temporelles adaptées, en fonction de la résolution de l'écran et du nombre d'images affichées par seconde. La figure présente l'allure des signaux de synchronisation et le tableau 2 les valeurs des différents paramètres pour deux résolutions.

Il faut noter que les durées ne sont pas données en temps mais en nombre de coups d'horloge. En effet, il faut disposer d'une horloge de base pour décompter le temps et générer les signaux avec le bon timing. En fonction de la résolution choisie, cette horloge de base a une fréquence adaptée. Dans ces conditions, il est tout aussi simple de donner des durées exprimées sous la forme d'un nombre de coups d'horloge à une fréquence donnée plutôt que la durée en ms ou μs . L'horloge est appelée "horloge pixel" (clkpixel ou clkvideo). En effet, sa fréquence est celle à laquelle il faut délivrer les pixels utiles à l'affichage.

Lorsque les signaux de synchronisation sont générés, l'écran doit pouvoir se synchroniser. Il reste alors à lui fournir l'information que l'on veut afficher pour qu'une image stable soit présente à l'écran. L'information doit être délivrée pendant toute la durée utile de la ligne (H active), c'est à dire lorsque le spot illumine effectivement l'écran. On se sert alors de l'horloge pixel pour rythmer l'apport des informations à l'écran : à chaque coup d'horloge, un pixel doit être fourni.