Ludovic Loiseau 21 Juin 2001

METHODOLOGIE DESIGN REUSE

Ecole Polytechnique de Montréal
Département Génie Electrique

Miranda Technologies Inc.

TABLE DES MATIERES

PHILOSOPHIE DU DOCUMENT

Les principe et les règles qui sont présentées dans ce document servent d'une part à produire des modules à haut niveau de performance et de confiance, et permettent également d'aboutir à une certaine homogénéité (de codage, mais aussi de validation et de documentation) permettant de favoriser les échanges et la compréhension à l'intérieur d'un groupe ou d'une compagnie.

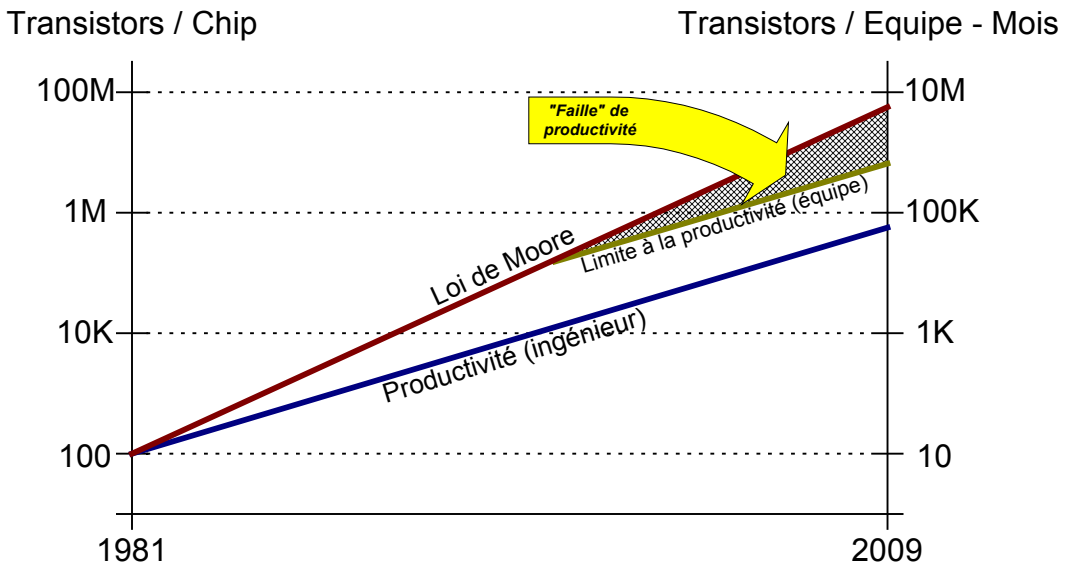
Cet ensemble de règles peuvent paraître contraignantes. Toutefois, une fois assimilées, l'application de la plupart d'entre elles n'entraînent pas ou très peu de travail supplémentaire. Par contre, des efforts particuliers concernant notamment la validation et la documentation sont effectivement inévitables.

Cette méthode sera à raffiner et devra évoluer en fonction des résultats rencontrés et du feed-back généré.

POURQUOI REUTILISER ?

L'évolution rapide de la micro-électronique met à disposition des concepteurs de FPGA comme d'ASIC des puces de plus en plus grandes (10 millions de portes dans le Virtex de Xilinx fin 2001), permettant d'intégrer des systèmes de plus en plus complexes sur une même puce.

En gardant un processus de conception identique à ce qui se fait à l'heure actuelle, il faudrait augmenter la taille des équipes de développement, malgré l'évolution des outils et des méthodes. Toutefois, il est démontré qu'au delà d'une dizaine d'ingénieurs travaillant sur une même puce, l'efficacité n'évolue plus de manière proportionnelle avec l'augmentation de la taille de l'équipe. Ainsi, comme le montre les courbes suivantes, il apparaît une faille de productivité.



Sans évolution de la méthode de conception, on ne pourra tout simplement pas (en un temps de design raisonnable) exploiter les ressources disponibles sur une puce.

Une méthodologie possible pour exploiter cette faille est le *design-reuse*, permettant de réutiliser dans un autre contexte des modules déjà conçus. Comme un module sera donc amené à être réutilisé, sa conception devra respecter un certain nombre de règles permettant justement de le rendre facile à réutiliser, mais aussi fiable et bien documenté. Ainsi, moyennant un effort plus ou moins important lors de la conception du bloc, chaque utilisation future ne nécessitera qu'un minimum de temps et d'efforts, dépendant de la qualité du bloc réutilisable.

On peut noter que la plupart du temps, la réutilisation est naturelle pour les concepteurs. On recherche souvent ce qu'on a fait de semblable auparavant. Egalement, on réutilise ses connaissances. Parfois même, on regarde ce qui a déjà été conçu de semblable dans l'équipe ou dans la compagnie. Toutefois, tout ceci ne sert en général que d'inspiration au travail actuel, pour contrer la peur naturelle de partir d'une feuille blanche. C'est un processus beaucoup moins efficace que de réutiliser entièrement et en pleine confiance. On remarque également qu'en général, on sera souvent beaucoup plus critique envers un tiers qu'envers soi même (typiquement, lorsqu'une personne va intégrer un sous-module étranger et que l'entité globale ne fonctionne pas, le module étranger va être accusé en premier). Il y a donc nécessité de développer une méthode structurée afin de répondre à ces objections naturelles.

Plusieurs approches de réutilisation sont envisageables :

- Approche "client" : on utilise dans ses projets des modules faits par d'autres compagnies (ex : Xilinx CoreGen, Altera LPM, ou d'autres compagnies spécialisées)

  Cette approche a l'avantage de demander très peu d'investissement initial. Par contre, on est obligé de se satisfaire des blocs disponibles, aux fonctionnalités souvent limités ou mal adaptées aux besoins. De même, l'intégration n'est pas toujours très pratique (le flot de conception normal est brisé, puisque généralement, le module à

intégrer n'est pas décrit en VHDL, mais plutôt sous la forme d'une net-list imposant alors l'utilisation d'un modèle pour la simulation).

- Approche "mixte" : on conçoit ses propres modules réutilisables, mais leur diffusion reste limitée, généralement à l'échelle de la compagnie.

  Cette approche est très avantageuse, puisqu'elle vise à se créer des modules parfaitement adaptés à ses besoins. Comme la diffusion reste interne, il n'y a pas à craindre de dévoiler son savoir-faire. Ainsi, le flot de conception d'un système reste identique (pas de différence entre intégrer un bloc réutilisable et intégrer un bloc spécifiquement conçu)

- Approche "fournisseur" : on conçoit des modules réutilisables pour les vendre (modules IP).

  Même si cette activité peut être très rentable, elle demande de développer d'autres compétences qui ne sont pas forcément naturelles dans une activité classique de conception (protection intellectuelle, ou encore support spécifique à la clientèle). De plus, on peut être amené à "offrir" ses compétences et sa technologie à son principal concurrent. Enfin, il faut absolument que la méthode soit parfaitement rodée en interne avant d'envisager de mettre des blocs sur le marché, pour ne pas compromettre la réputation d'une compagnie.

VUE D'ENSEMBLE

Avant de vouloir réutiliser un module, il faut avant tout pouvoir l'UTILISER. C'est à dire que toutes les règles qui font un bon design doivent être appliquées :

- Bien coder
- Bien commenter
- Bien vérifier
- Bien documenter

En plus, pour être réutilisable, un module doit être :

- **CONFIGURABLE**
  Conçu pour résoudre un problème général
- **PORTABLE**
  Conçu pour différentes technologies (i.e. différents fournisseurs de composants), et pour différents outils
- **DÉBOGUABLE**
  Vérifié avec un haut niveau de confiance
- **LISIBLE**
  Documenté clairement au niveau des applications et restrictions

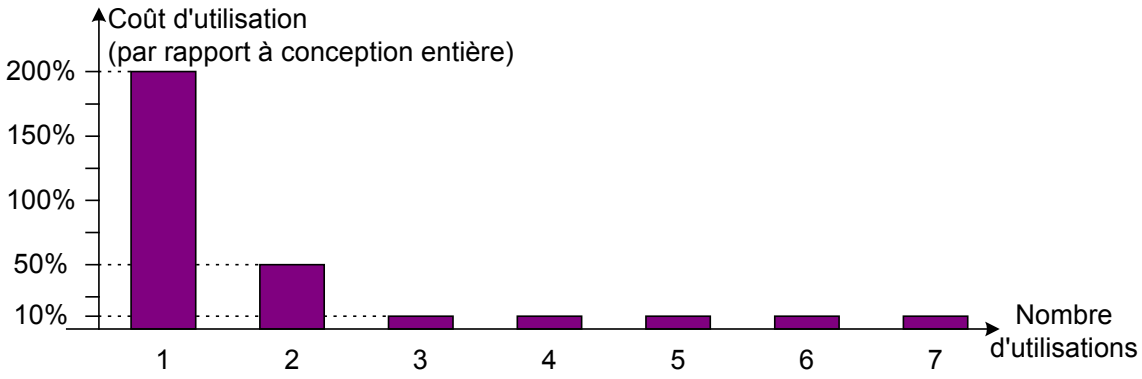
Ainsi, on voit qu'un effort supplémentaire est nécessaire pour concevoir un module réutilisable. Les aspects de vérification et de documentation sont probablement les plus

chronovores dans le projet. De plus, il faudra souvent faire quelques concessions quant au matériel nécessaire pour implémenter la fonction. Le code sera en effet souvent moins optimisé qu'un code n'implémentant que la fonction de base (pour intégrer d'autres fonctionnalités, l'architecture sera probablement moins optimale)

Plusieurs stratégies peuvent être envisagées lorsqu'on envisage de concevoir un module réutilisable.

STRATEGIE PLANIFIEE

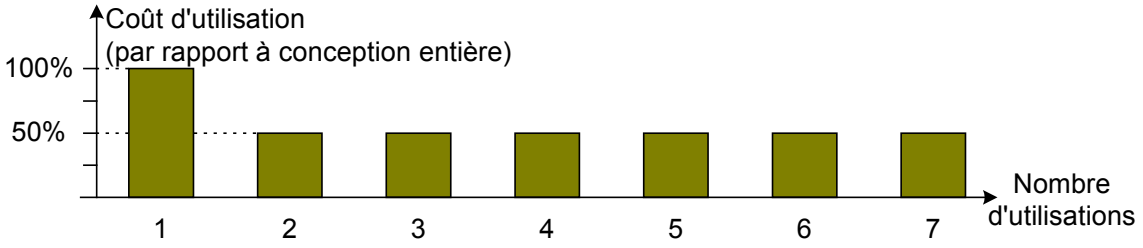
Pour une stratégie planifiée, on présume que le bloc sera réutilisé très souvent. Ainsi, dès la conception initiale, on va faire beaucoup d'efforts pour obtenir un bloc très performant. Il va donc être tout de suite très configurable, parfaitement vérifié, pour toutes les configurations possibles, et aussi complètement documenté. Cette conception initiale demandera donc beaucoup de temps et de ressources.



On estime qu'il faudra passer environ 2 fois plus de temps lors de la conception initiale (par rapport à une conception sans objectif de réutilisation). Lors de la seconde utilisation, on estime également qu'un peu de travail serait à faire. Il peut s'agir de caractéristiques "oubliées" lors de la conception initiale, ou encore d'une configuration négligée lors du test. Par contre, les utilisations futures seront très économiques, se résumant idéalement à une lecture de documentation suivie de l'intégration du module.

STRATEGIE AD-HOC

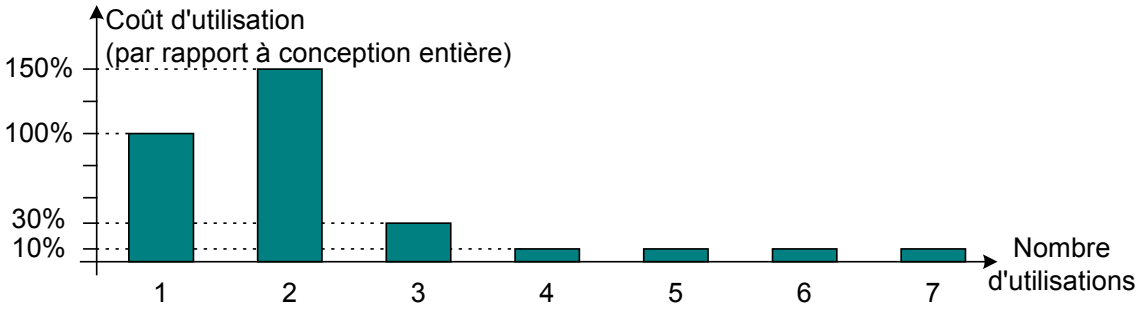
Pour cette stratégie, on ne sait pas encore si le module en question sera réutilisé ou non. Ainsi, on ne fera pas d'effort particulier pour le rendre réutilisable dès la conception initiale. De même, lors de chaque réutilisation, on va juste adapter ou modifier le code existant à son nouveau contexte d'utilisation.



La conception initiale n'est donc pas plus coûteuse qu'une conception sans objectif *reuse* (elle est d'ailleurs en tout point identique). Par contre, chaque utilisation future restera encore assez coûteuse (50% est une moyenne, dépendant du degré de nouveauté du contexte courant d'utilisation). Même si le module sera réutilisé, il ne sera jamais réutilisable en tant que tel.

STRATÉGIE *REDESIGN FOR REUSE*

Pour cette stratégie également, on n'est pas certain que le module devra être réutilisé souvent. Ainsi, on ne fait pas plus d'effort particulier lors de la conception initiale. Par contre, lors de la seconde utilisation et si on est désormais convaincu du potentiel *reuse* du bloc, on va faire des efforts de conception afin de le rendre parfaitement réutilisable. Le module devrait devenir aussi performant que le même module conçu avec une stratégie planifiée.



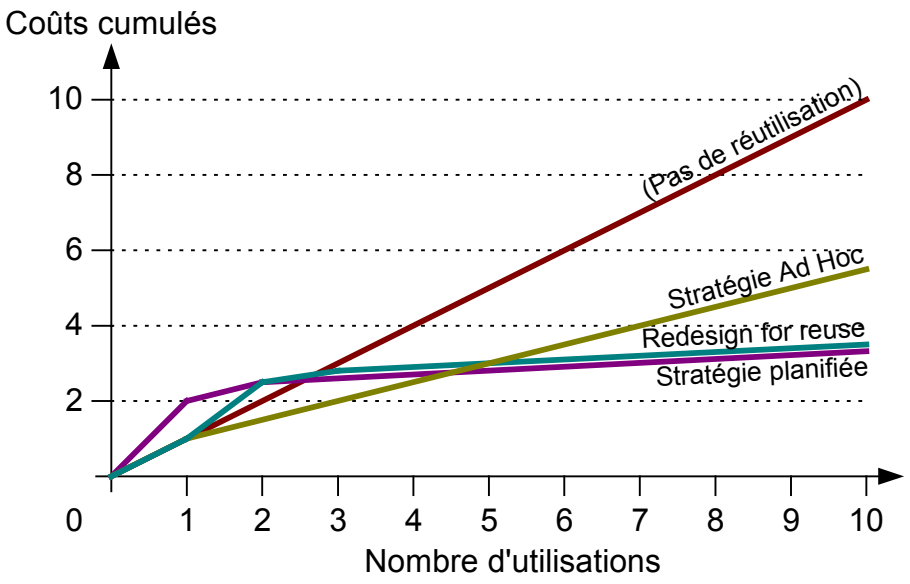
L'effort fourni lors de la seconde utilisation sera probablement moins soutenu que la conception initiale avec une stratégie planifiée. En effet, comme c'est une reconception, on aura une meilleure connaissance du bloc, et donc on saura mieux comment le rendre bien configurable, et la validation sera également plus facile.

On note toutefois que la conception initiale ne se fait pas sans aucun objectif *reuse*. C'est à dire que la méthode de conception est légèrement différente d'une conception sans objectif de réutilisation. Cependant, les règles envisagées lors de cette conception initiale ne devront pas être très contraignantes. Après assimilation, leur application ne devrait pas ou très peu entraîner d'effort particulier par rapport à un bon design sans objectif particulier de réutilisation.

COMPARAISON DES STRATEGIES

Les coûts cumulés de chaque stratégie en fonction du nombre d'utilisations montrent

l'avantage d'une stratégie planifiée ou du *redesign for reuse* par rapport à une stratégie ad-hoc. Par contre, on voit que la stratégie planifiée est très coûteuse à la conception initiale.



La stratégie ad-hoc est presque toujours inintéressantes, sauf peut-être dans le cas de blocs dont on est certain de très peu les réutiliser (typiquement moins de 4 à 5 utilisations)

Les avantages d'une stratégie *redesign for reuse* sont très intéressants :

- Moins de risques de se tromper sur le potentiel *reuse* d'un bloc : on attend un besoin de réutiliser avant de faire des efforts sur un bloc. Il serait très gênant d'avoir fait un bloc réutilisable alors que l'on en a finalement pas besoin.

- Après la conception initiale, on a plus de connaissances sur le bloc, et on peut donc mieux savoir comment le rendre bien configurable et adapté à un usage général. Ici également, il serait très gênant d'avoir fait un bloc réutilisable et avoir "oublié" une ou plusieurs caractéristiques importantes qui va le rendre beaucoup moins intéressant à réutiliser.

- L'effort de rendre le bloc réutilisable peut se faire quand on le juge nécessaire ou quand on a les ressources nécessaires (généralement à la seconde utilisation, voire plus tard dans certains cas). Le processus de *reuse* pourra donc commencer même sur un module se trouvant sur un chemin critique.

Pour ces raisons, c'est la stratégie qui est conseillée dans un premier temps pour mettre en place la méthodologie *reuse* à Miranda. Par contre, une fois bien assimilé, une stratégie planifiée pourra être envisagé, lorsqu'on saura comment faire un bon bloc réutilisable.

QUAND ABORDER LE *REUSE* DANS UN PROJET ?

Les aspects de réutilisation doivent être pris en compte tout au long de la phase de conception du projet.

- Un travail préliminaire doit être effectué par le concepteur du bloc présumé réutilisable. Il devra d'une part s'assurer que le bloc a effectivement un potentiel *reuse*. Il devra également réfléchir aux caractéristiques intéressantes que le bloc pourrait avoir pour le rendre adapté à un usage général. Pour cela, le concepteur est amené à consulter les personnes compétentes à juger des caractéristiques futures nécessaires. Ces personnes peuvent être les concepteurs algorithmiques, les chefs de groupes de conception, les directeurs des sections R&D ou encore les responsables développement de produit. Ce travail doit être fait lors de la conception initiale, et éventuellement lors de la reconception réutilisable, pour savoir quelles caractéristiques ajouter.

- Lors de la révision de concept du projet dans lequel s'intègre le module, le concepteur présente le bloc et ses caractéristiques envisagées. Le concepteur présente également quand est prévue l'intégration de chaque caractéristique (à la conception initiale ou lors de la seconde utilisation). Une discussion devrait également être amorcée, afin d'identifier des caractéristiques non encore envisagées et qui pourraient être intéressantes à avoir. Faire cette discussion lors d'une révision conceptuelle est très intéressant, puisque les personnes présentes sont généralement des futurs utilisateurs potentiels du bloc, et donc donneront un feed-back extrêmement valable et enrichissant. Cette discussion s'apparenterait à une relation client - fournisseur.

  Par contre, lors d'une révision de concept correspondant à la seconde utilisation du module, on peut se contenter d'aborder les nouvelles caractéristiques qui seront intégrées lors du *redesign* du bloc, ainsi que discuter sur celles qui devraient être l'être.

- La révision détaillée n'est pas forcément nécessaire pour un bloc réutilisable. On peut la limiter seulement aux blocs complexes et à fort contenu algorithmique. L'architecture du bloc devrait être présentée, faisant apparaître quelles caractéristiques seront disponibles lors de la conception initiale.

  La révision détaillée d'un bloc lors de sa deuxième utilisation ne devrait être faite que pour des ajouts ou modifications majeurs de caractéristiques qui pourraient remettre en cause l'architecture de base. Toutefois, ce cas devrait être le plus rare possible, puisque l'architecture initiale devrait être conçue pour pouvoir supporter les principales caractéristiques à venir.

- Un audit final du module devra être fait, une fois que celui-ci est réellement réutilisable, donc probablement après sa seconde utilisation. Il s'agit d'une action de vente du module auprès de ses utilisateurs éventuels (ou des personnes qui pourraient décider de son utilisation). On doit présenter le bloc finalisé, ses caractéristiques principales, ses performances mais aussi ses limitations. On se différencie toutefois légèrement d'un véritable processus de vente, dans le sens où il ne faut absolument pas vouloir cacher les défauts éventuels du bloc. Le but est de

fournir une information claire et précise à l'utilisateur éventuel, afin qu'il sache parfaitement dans quel contexte il pourra utiliser ou non ce bloc. Il ne doit pas faussement croire que le bloc répondrait à son attente, pour finalement se rendre compte qu'il devra entièrement concevoir un module adapté (par exemple parce que l'architecture actuelle ne permettrait pas d'intégrer les caractéristiques nécessaires à la nouvelle utilisation).

MODELE EN COUCHES

L'ébauche de méthodologie proposée (redesign for reuse) ci dessus fait ainsi apparaître 2 étapes lors de la conception d'un module réutilisable : la conception initiale puis le redesign pour obtenir un module pleinement réutilisable. Ainsi, chacune des deux étapes fournit un module qui a un certain niveau de qualité. Cette qualité englobe tous les aspects du bloc : spécifications, codage, validation, documentation, mais aussi le niveau de configurabilité. En résumé, plus le bloc sera de qualité, mieux il sera réutilisable. Le but de la reconception est ainsi d'augmenter le niveau de qualité du bloc en l'enrichissant (par exemple, en le documentant mieux, en le validant plus rigoureusement, en lui ajoutant des caractéristiques…).

On peut donc essayer de classifier un bloc par son niveau de qualité :

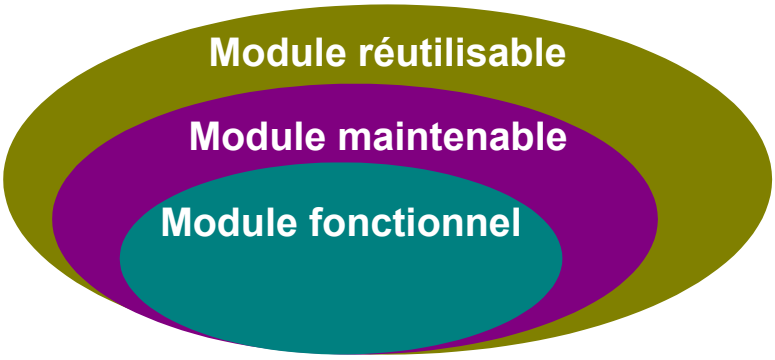
- **Fonctionnel** : C'est le niveau minimum qu'un bloc doit atteindre pour pouvoir être utilisé. Le module doit être vérifié par rapport à l'application visée, et les timings respectés. Toutefois, le code n'est pas optimisé, la validation est très sommaire et rien n'est documenté. Ainsi, ce niveau de qualité ne devrait pas être acceptable, même si le module peut sembler fonctionnel. Son intégration posera probablement des problèmes, et la moindre modification sera très coûteuse.

- **Maintenable** : Par rapport au niveau fonctionnel, un bloc maintenable serait tout aussi fonctionnel, mais également bien documenté (schémas, codage propre et clair, commentaires de code, spécifications sommaires…). Par contre, il n'est pas nécessairement optimisé. Ce niveau de performance est théoriquement le minimum acceptable pour un bloc spécifique, permettant de faciliter son intégration. Généralement, un tel bloc est un minimum configurable, mais ne permet pas un grand nombre de configurations. On peut donner comme exemple la taille des données, qui devrait être déterminée par une constante (générique ou non), permettant d'offrir une flexibilité à peu de frais. Par contre, la validation se limitera généralement au fonctionnement prévu dans le cadre de l'utilisation pour laquelle le bloc est conçu initialement. L'intérêt d'un tel module est qu'il pourra être modifié à moindre coût.

- **Réutilisable** : Un module réutilisable serait un enrichissement d'un module

maintenable. Dans ce cas, le module serait beaucoup plus configurable et adapté à un usage général. De plus, il serait parfaitement validé pour toutes les configurations valides, et portable vers différents outils et pour différentes technologies. Le code devrait être optimisé. De même, le module est accompagné d'une documentation complète.

On voit ici que chaque niveau de qualité est un enrichissement par rapport à un niveau inférieur, laissant apparaître un modèle en couche, comme le montre la figure suivante. On peut ainsi passer d'un niveau de qualité à un autre en faisant un effort supplémentaire de conception.



On peut ramener cette description en couches en rapport à la stratégie de design-reuse envisagée :

- **Planifiée :** Le module sera réutilisable dès la conception initiale
- **Ad-hoc :** Le module devra être maintenable à la première conception, mais, même si il sera enrichi au fur et à mesure des réutilisations, il ne sera jamais pleinement réutilisable (notamment pas complètement documenté, ni validé avec une confiance absolue).
- **Redesign for reuse :** Le module devra être maintenable à la conception initiale, puis deviendra réutilisable au moment du redesign.

REGLES DE DESIGN

Les numéros des règles reprennent la notation du livre "Reuse Methodology Manual", pour pouvoir aisément s'y reporter.

REGLES DE BASE

Les règles de base ont pour but d'une part de permettre un codage clair et compréhensible et d'autre part indiquent également les principes pour faire un bon design (sans nécessairement parler de réutilisation). Ces règles devraient être appliquées quel que

soit l'objectif du module.

5.2.1.1 – Définir une convention pour nommer les signaux, variable, boucles, process…, et la suivre tout au long du design.

* 5.2.1.2 – Utiliser des minuscules pour nommer les signaux, les variables et les ports.

* 5.2.1.3 – Utiliser des majuscules pour nommer les constantes et les sous-types dérivés de types standards

* 5.2.1.4 – Utiliser des noms descriptifs clairs pour les signaux, ports, fonctions et paramètres

* 5.2.1.6 – Utiliser le nom clk pour l'horloge, ou le préfixe clk_ suivi d'un court descriptif (pour donner sa fonction ou sa fréquence par exemple).

* 5.2.1.8 – Les signaux actifs bas doivent être indiqués clairement par le suffixe _n

* 5.2.1.9 – Utiliser le nom rst pour un signal de reset, ou le préfixe rst_ suivi d'un court descriptif (pour donner sa fonction). Si il est actif bas, le préciser par rst_n.

* 5.2.1.10 – Déclarer les ports avec MSB à gauche et LSB à droite (7 downto 0 par exemple).

* 5.2.3.1 – Utiliser la notation suivant pour nommer les architectures :
  - **rtl** pour l'architecture synthétisable d'un module, en description RTL ou mixte (RTL + structurelle).
  - **struct** pour une description complètement structurelle (instanciation de sous-blocs) d'un module
  - **behav** pour une architecture comportementale non-synthétisable (utilisée pour la simulation)
  - **tb** pour le test-bench

* 5.2.4.1 – Inclure une entête standard dans toutes les fichiers sources. Inclure les informations essentielles, dont l'utilisateur pourrait avoir besoin.

```
--------------------------------------------------------------------------------
-- TITLE :       Consolidation (basic, not by block, not conditionnal)
-- DESCRIPTION : Configurable consolidation module
--               Size of the window, threshold and type of consolidation are generic
--               Only odd values for window size are valid.
--               The processing delay (due to the structure) is 3 clk cycles, and you
--               must add the time to reach central pixel.
--               Ex : for 3*3 : 1 line delay + 4 clk cycles.
--               Processes only demultiplexed data.
-- FILE :        consolidation.vhd
--------------------------------------------------------------------------------
-- CREATION
-- DATE         AUTHOR            PROJECT   REVISION
-- 2001/02/13   Ludovic Loiseau   DT4101    V1.0
--------------------------------------------------------------------------------
-- MODIFICATION HISTORY
-- DATE        AUTHOR             PROJECT   REVISION    COMMENTS
--------------------------------------------------------------------------------
```

** 5.2.5.1 – Commenter généreusement le code pour expliquer tous les process, fonctions et déclarations de types et sous-types. Utiliser des commentaires intelligents (expliquer pourquoi, plutôt que de retraduire littéralement le code)

** 5.2.5.2 – Tous les ports, signaux et variables, ainsi que les groupes de signaux et de variables doivent être commentés.

* 5.2.6.1 – Une seule instruction par ligne de code.

5.2.7.1 – Limiter la longueur des lignes de codes à 100 caractères pour améliorer la portabilité entre éditeurs de texte.

* 5.2.8.1 – Indenter le code pour en améliorer la lisibilité et repérer rapidement la structure des boucles ou des actions conditionnelles.

* 5.2.8.2 – Indenter par des espaces (et non des tabulations), de largeur 1 à 4. Rester consistent pour tout le codage. 3 semble être la meilleure valeur.

* 5.2.9.1 – Ne pas utiliser de mots réservés au VHDL ou au Verilog pour nommer les éléments du code (voir la liste des mots réservés en annexe).

5.2.10.1 – Déclarer les ports dans un ordre logique, par fonctionnalité plutôt que par type.

* 5.2.10.2 – Ne déclarer qu'un seul port par ligne, si possible avec le commentaire associé sur la même ligne.

* 5.2.10.3 – Pour chaque groupe logique de ports, respecter l'ordre suivant :
  - clocks,
  - resets,
  - enables,
  - autre signaux de contrôle,
  - données.

5.2.10.4 – Commenter chaque groupe de ports.

* 5.2.11.1 – Utiliser toujours un mapping explicite pour les ports et les paramètres génériques. Préciser toujours la valeur d'un générique au generic map (ne pas garder la valeur par défaut de l'entité).

5.2.12.1 – Un seul fichier VHDL doit contenir l'entité, l'architecture et les configurations (si applicable).

5.2.13.1 – Utiliser des fonctions plutôt que répéter une même section de code. Ne pas oublier de commenter la fonction.

5.2.14.1 – Utiliser des boucles et des tableaux pour améliorer la clarté du code.

** 5.2.14.2 – Utiliser des opérations vectorielles plutôt que des boucles lorsque c'est possible (ex : mapper directement 2 colonnes d'un tableau plutôt qu'élément par élément).

5.2.15.2 – Nommer un process en indiquant son but (ou le signal qu'il génère), précédé du préfixe do_.

* 5.2.15.3 – Le nom d'une instance d'un composant doit être U_component_name, ou Ui_component_name pour plusieurs instances d'un même composant dans une même entité.

* 5.2.15.5 – Ne pas utiliser de noms des signaux, variables ou entités comme étiquette de process.

Autres règles importantes :

• – Ne pas mélanger majuscules et minuscules dans les noms des signaux, sauf exception précises (par exemple LD pour Line Delay). Utiliser le caractère _ pour séparer des notions dans un même nom

* • – L'entité devrait porter le même nom que le fichier dans lequel elle est décrite.

• – Ne générer qu'un seul signal par process séquentiel, ou alors un groupe de signaux de même niveau logique (par exemple, dans le cas d'une structure pipelinée, on peut assigner dans un même process tous les signaux du même niveau de pipeline).

* • – Effacer toute ligne de code mise en commentaire.

* • – N'utiliser que le type *std_logic* ou *std_logic_vector* aux port des blocs et sous-blocs. Pour des opérations arithmétiques, convertir les signaux au besoin en interne en type *signed* ou *unsigned*.

* • – Ne pas donner de valeur par défaut à un signal. La plupart du temps, cette valeur est ignorée à la synthèse, donc peut amener des mismatchs par rapport à la simulation.

• – Tous les noms de signaux, et tous les commentaires doivent être en anglais, afin d'avoir une certaine homogénéité par rapport à la syntaxe du langage.

Le document "*VHDL Work Method* ", disponible en annexe, présente une proposition de convention pour nommer les signaux. Ce document a été fortement inspiré par les règles données dans l'ouvrage "*Reuse Methodology Manual*", qui ont ensuite été développées pour en faire une convention.

REGLES AU NIVEAU SYSTEME

Ces règles s'appliquent lorsque le module réutilisable peut être considéré comme un système complet, qui pourrait se suffire à lui-même.

3.2.3.1 – Le nombre de domaines d'horloge ainsi que leurs fréquences sont bien documentés. Les timings externes à respecter (setup time, hold time…) ainsi que les PLL/DLL utilisées (ou à utiliser) sont bien définis. Les horloges au niveau système ne doivent utiliser que des arbres de distribution dédiés.

3.2.3.2 – Si 2 domaines d'horloge asynchrones doivent être utilisés, s'assurer qu'ils soient synchronisés dans un module simple qui devrait juste être composé de registres permettant de rendre les 2 domaines d'horloge synchrones.

3.2.4.1 – La stratégie de reset doit être documentée au niveau système, en particulier pour les points suivants :

- synchrone ou asynchrone ?
- reset à la mise sous-tension interne ou externe ?
- chaque bloc doit-il être pouvoir être remis à zéro individuellement (pour déboguage) ?

3.4.1 – La stratégie de vérification au niveau système doit être développée et documentée avant de commencer la sélection (si le système est un assemblage de modules réutilisables) ou le codage des sous-blocs

3.4.2 – La stratégie de vérification au niveau des sous-blocs doit être développée et documentée avant d'en commencer la conception.

3.5.5.2 – Soigner la contrôlabilité et l'observabilité pour faciliter le déboguage au niveau système. La contrôlabilité est facilitée en donnant la possibilité d'éteindre individuellement chaque sous-module, ou de le mettre en mode déboguage (fonctionnement simplifié). L'observabilité est améliorée en ajoutant des bus permettant de contrôler l'état du système.

CODAGE POUR PORTABILITE

* 5.3.1.1 – N'utiliser que des types définis dans les librairies IEEE (ou des sous-types dérivés de librairies IEEE).

* 5.3.1.2 – Utiliser les types *std_logic* et *std_logic_vector* plutôt que *std_ulogic* et *std_ulogic_vector*, moins bien supporté par les outils.

5.3.1.3 – Limiter le nombre de sous-types dérivés.

* 5.3.1.4 – Ne pas utiliser les types *bit* et *bit_vector*.

5.3.2.1 – Ne pas utiliser de valeurs numériques hard-codées autres que 0, 1 ou 2.

5.3.3.1 – Tous les paramètres ou fonctions communes à un design doivent être regroupées dans un package séparé nommé *design_name_pack.vhd*.

5.3.6.1 – Utiliser autant que possible des librairies indépendantes de la technologie utilisée.

** 5.3.6.2 – Ne pas instancier de portes ou de composants spécifiques directement dans un

design. Les encapsuler dans des sous-modules pour rendre le top-level indépendant de la technologie.

REGLES POUR CLOCK ET RESET

* 5.4.4.1 – Les registres ne doivent être sensibles qu'à un seul front d'horloge (généralement montant). Exception possible pour les mémoires DDR.

* 5.4.3.1 – Pas de logique sur l'horloge (gated clock).

* 5.4.4.1 – Pas d'horloge générées en interne (ou donner un moyen d'utiliser une horloge externe en phase de déboguage).

* 5.4.5.1 – Pas de reset générés en interne ou de reset conditionnels (ou donner un moyen d'utiliser un reset externe en phase de déboguage).

Autres règles importantes :

• – Un reset asynchrone est conseillé (plus facile à tester, et également moins de risques de problèmes à la mise sous tension).

CODAGE POUR LA SYNTHESE

5.5.5.1 –Utiliser des registres indépendants de la technologie lors du codage style RTL pour la logique séquentielle.

5.5.2.1 – Le module doit être synchrone et basé sur des bascules D. Ne pas instancier de latches lors du codage RTL, particulièrement des bascules S-R.

5.5.2.2 – Lors d'une assignation conditionnelle, utiliser l'une des règles suivantes pour ne pas inférer de latches :
- Assigner une valeur par défaut en début d'un process,
- Assigner une valeur de sortie pour chaque combinaison des entrées,
- Terminer l'assignation conditionnelle par la condition *else* ou *when others*.

5.5.4.1 – Pas de retour combinatoire, i.e. de boucle sur un process combinatoire.

5.5.5.1 – Compléter toutes les listes de sensibilité des process :
- L'horloge et les signaux asynchrones pour un process séquentiel,
- Tous les signaux d'entrée d'un process combinatoire.

5.5.5.2 – N'indiquer que les signaux nécessaires la liste de sensibilité.

5.5.7.1 – Utiliser des signaux plutôt que des variables. Exception possible pour les indices de tableau ainsi que les valeurs intermédiaires accumulées lors de boucles.

* 5.5.8.1 – Utiliser des condition *case* plutôt que des conditions *if-then-else*, quand approprié.

* 5.5.9.1 – La description d'une machine à états doit se faire en utilisant 2 process, un pour la

logique séquentielle, un pour la logique combinatoire.

5.5.9.2 – Créer un type énuméré pour décrire le vecteur d'état, permettant à l'outil de synthèse d'en optimiser le codage.

* 5.5.9.4 – Pour une machine à état, assigner un état par défaut.

Autres règles importantes :

• – Utiliser un pipelinage massif, permettant d'atteindre des bonnes performances.

• – Mettre un registre en sortie de chaque sous-bloc (si le design est complètement synchrone, ce registre existe déjà)

• – Mettre un registre en entrée de chaque bloc (à partir d'une certaine complexité seulement, pas nécessaires pour les sous-blocs). Ceci permet de mieux prévoir les timings quelle que soit la complexité et la taille du système dans lequel est intégré le bloc.

PARTITIONNEMENT POUR FACILITER LA SYNTHESE

5.6.1.1 – Pour chaque sous-bloc d'un design hiérarchique, chaque signal de sortie doit être enregistré en sortie.

5.6.3.1 – Isoler le chemin critique des chemins non-critiques, pour permettre aux outils d'optimiser différemment (vitesse pour le chemin critique, surface pour le reste).

5.6.4.1 – Eviter d'utiliser de la logique asynchrone, ou si absolument nécessaire, la séparer dans un module spécifique.

5.6.5.1 – Partitionner les calculs arithmétiques dans le même module, pour permettre de profiter du partage des ressources (resource-sharing).

5.6.8.1 – Pas de glue-logic au top-level.

CONFIGURABILITE

La configurabilité est un des éléments-clés de la réutilisation. Plus un bloc sera configurable, plus il sera adapté à un usage général, plus il sera compatible avec des applications différentes et ainsi, plus il sera réutilisé.

Le but de cette étude n'est pas d'être exhaustif, mais plutôt de montrer les 2 principales manières de concevoir un bloc réutilisable.

1. Ajouter des caractéristiques supplémentaires au module.

   Exemple : Caractéristiques possibles pour un filtre LP :
   - coefficients génériques,
   - taille générique,

- reset asynchrone, facultatif (en mettant une valeur par défaut pour les cas où l'on n'a pas besoin de remise à zéro),
- traitement possible de luminance seule, de chrominance seule, ou des 2 composantes multiplexées (au format 4:2:2).

2. Faire un design modulaire, en facilitant l'intégration de modules aux caractéristiques différentes. Il peut parfois être complexe d'intégrer un grand nombre de caractéristiques dans un même bloc. Créer plusieurs blocs aux caractéristiques différentes peut être plus facile. Dans ce cas, la configurabilité est obtenue en facilitant l'intégration des blocs et en permettant de remplacer un bloc par un autre qui aurait des caractéristiques différentes. Le système ainsi généré est configurable (à condition d'avoir à disposition les sous-blocs adéquats).

   Exemple : Pour un filtre LP, il peut être difficile de concevoir un bloc aux coefficients et à la taille génériques qui soit optimal au niveau performances et complexité. Ainsi, on peut être amené à concevoir un certain nombre de filtres de aux coefficients et tailles différentes. Lorsque ces blocs sont intégrés dans un système, il est intéressant de faire une intégration générique pour faciliter le remplacement d'un filtre par un autre. Dans cette exemple, la généricité se ferait au niveau des lignes à délai, en permettant très facilement l'instanciation d'un nombre variable de lignes.

VERIFICATION

La vérification est un des points-clés de la réutilisabilité, puisque seule une vérification rigoureuse permettra de donner à l'utilisateur la confiance nécessaire dans le bloc pour accepter d'intégrer le bloc dans son design. On estime en moyenne que le temps à consacrer à la validation d'un module ou d'un système devrait se situer autour de 70% du temps total de conception du bloc. Ceci est d'autant plus vrai lorsque le module doit être réutilisable.

Un module réutilisable devrait avoir un fonctionnement garanti à un niveau très proche de 100% pour toutes les configurations valides de celui-ci. Ainsi, la validation peut s'avérer extrêmement laborieuse si le module est fortement configurable.

Si on se ramène à notre stratégie de conception de modules réutilisable, en 2 étapes, on suggérera pour chacune des étapes :

- Conception initiale : On doit tester le module de manière rigoureuse, mais seulement pour l'application envisagée pour cette première utilisation. On ne testera donc pas toutes les caractéristiques possibles du bloc (dans le cas où elles existent déjà mais ne sont pas utilisées pour cette première utilisation). Le temps de validation n'est ainsi pas plus long que tester un module conçu seulement spécifiquement pour l'application.
- Reconception : Il faut refaire un effort de validation pour valider vraiment toutes les caractéristiques que l'on veut garantir pour le module (ces caractéristiques peuvent être celles intégrées lors de la conception initiale mais pas utilisées la première fois,

donc non encore validées, ou des caractéristiques ajoutées lors de la reconception).

Le but de la validation est de prouver que le module va fonctionner dans l'environnement dans lequel il sera intégré. Pour cela, il faut simuler cet environnement, ce qui est l'objectif du test-bench. Voici ce que doit faire un test-bench :

- Instancier le bloc à tester,
- Générer un stimulus d'entrée, aussi proche que possible du stimuli réel auquel le bloc devra faire face,
- Générer des vecteurs de sortie de référence ("*golden vectors*"),

ou

- Instancier ou générer un modèle de simulation,
- Comparer de manière automatique le résultat théorique et le résultat obtenu.

Les stimuli peuvent être :

- Des vecteurs ad-hoc (généralement pour les signaux de contrôle),
- Des vecteurs géométriques simples (rampes…),
- Des vecteurs aléatoires (à distribution connue),
- Des images (en passant par un langage de haut-niveau).

Pour un test-bench basique, seule l'instanciation du bloc et la génération des stimuli d'entrée sont nécessaires. Toutefois, ceci impliquerait une vérification manuelle des résultats, donc une validation incomplète (puisqu'une vérification manuelle ne pourra pas couvrir assez de combinaisons d'entrée et est de plus sujette à erreur).

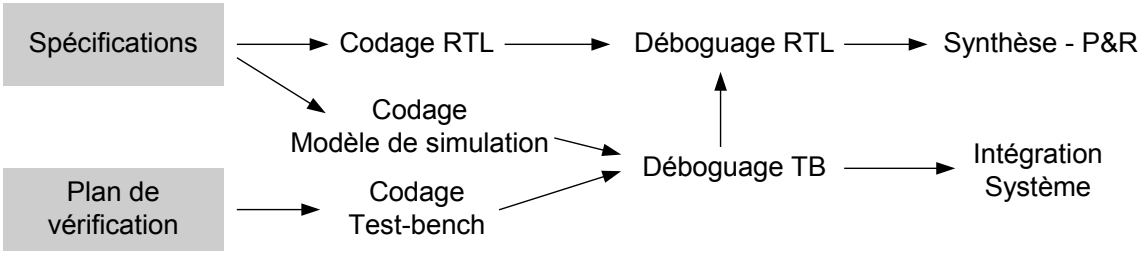
Pour cette raison, une vérification fiable ne peut se faire qu'en faisant une comparaison automatique des résultats de sortie théoriques et pratiques. On peut ainsi faire une validation sur une très grande combinaison d'entrée (et même possiblement toutes les combinaisons), tout en évitant les erreurs humaines. Cette comparaison suppose que l'on dispose de vecteurs de sortie dont on est sûr de la validité.

Pour ces vecteurs, la première solution est d'utiliser un outil de haut niveau tel que le C++, permettant notamment en vidéo de vérifier visuellement le résultat d'un traitement, et donc la validité des vecteurs ainsi générés. Par contre, ceci oblige à jongler entre 2 outils (C et VHDL), obligeant donc à passer par des fichiers comme interface.

La seconde solution est d'utiliser directement le VHDL comme outil de modélisation. En effet, lorsqu'on code en VHDL synthétisable, on n'utilise qu'une restriction des commandes du VHDL. On peut donc utiliser le VHDL pour générer une description "haut-niveau" d'une fonctionnalité. La description devrait être rapide, puisque toutes les notions hardware ne doivent pas apparaître. Les avantages sont les suivants :

- Les spécifications sont abordées tôt dans le processus de conception, donc si un problème apparaît à ce niveau, celles-ci seront "déboguées" au moment de l'écriture de la description synthétisable du module (donc on ne perd pas de temps à revenir aux spécifications lors du codage),

- Comme le montre le schéma suivant, on peut se servir du modèle de simulation pour déboguer le test-bench. On aura donc un test-bench fonctionnel disponible dès que la description synthétisable sera finie,



- Dès que le modèle de simulation est terminé, on peut commencer l'intégration au niveau système sans attendre d'avoir terminé la description synthétisable (en intégrant le modèle plutôt que le module synthétisable),

- La génération du test-bench est facilitée, puisqu'il se limite à une démonstration d'équivalence.

- Les simulations sont accélérées (si l'équivalence est démontrée, on utilise le modèle, plus rapide à simuler, dans les simulations des top-levels).

On voit ici que les avantages sont multiples, et permettent surtout de paralléliser au maximum les chemins de conception, afin d'aboutir à un temps global de conception plus intéressant.

Par contre, un tel modèle de simulation peut paraître complexe à faire efficacement. En effet, la plupart du temps, le VHDL est considéré comme un langage de description de modules hardware, donc on a tendance à oublier (ou à n'avoir jamais connu) les commandes non-synthétisables. Egalement, on a de la difficulté à vraiment s'éloigner des notions matérielles. En VHDL non-synthétisable, on ne doit pas avoir de notions d'horloge ou de FSM, puisque ce ne sont que des concepts utiles pour faire des modules hardware fonctionnels et performants. Un modèle comportemental ne devrait pas être fait après la conception du module synthétisable par le concepteur lui-même. En effet, son codage comportemental pourrait être biaisé par le codage synthétisable déjà fait, et il paraîtra difficile de se détacher suffisamment du hardware. Normalement, un codage du modèle de simulation devrait se faire avant le codage synthétisable (dans le cas d'un même concepteur), ou idéalement en même temps si on veut paralléliser les tâches au maximum. En plus, si 2 personnes différentes codent les 2 descriptions, on évitera les erreurs possibles d'interprétation des spécifications.

DOCUMENTATION

Le but de la documentation est double :

- Donner à l'utilisateur toutes les informations nécessaires à l'intégration du bloc.
- Décrire clairement comment fonctionne le bloc, afin de permettre d'éventuelles modifications ultérieures.

Pour cela, la documentation est divisée en rubriques, auxquelles l'utilisateur ou le re-concepteur accédera en fonction du but visé. Ce document doit être parfaitement clair et devrait, dans la mesure du possible, rester assez concis.

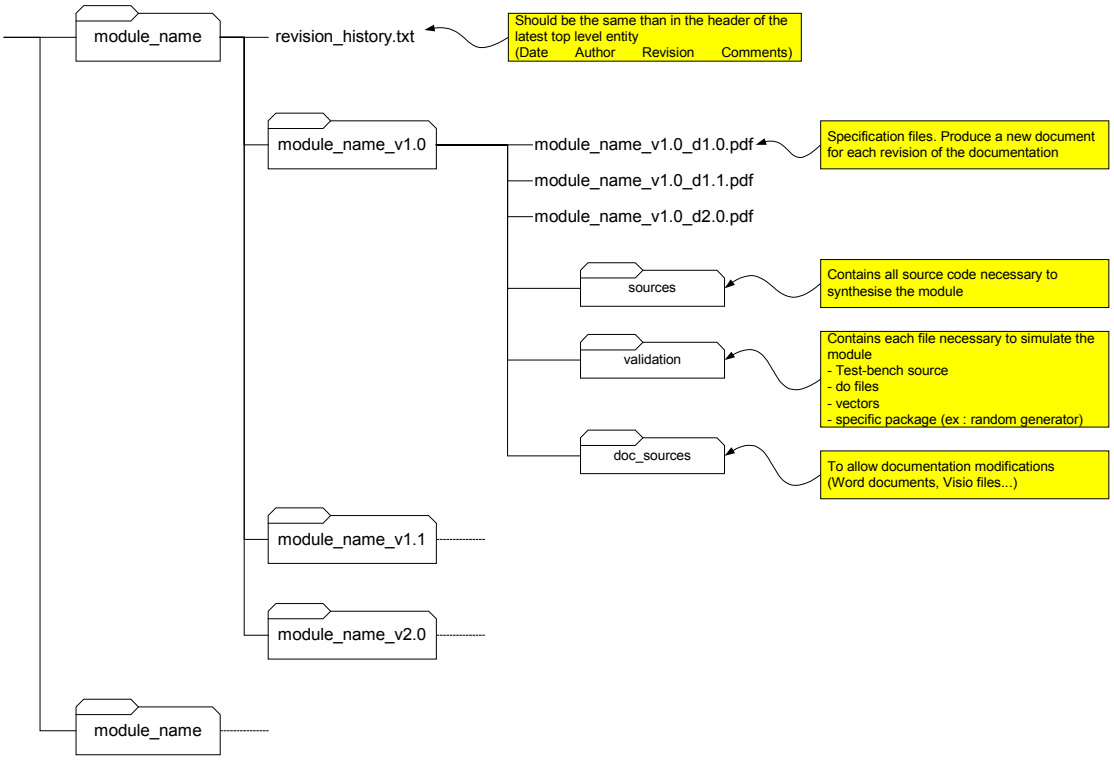
Pour cette tâche également, si on travaille sur une conception en 2 étapes, on peut imaginer que la documentation complète ne soit générée que lors de la phase de reconception, après avoir généré une version allégé (permettant juste de comprendre le fonctionnement du bloc), lors de la conception initiale. Ceci permettrait d'alléger le travail, puisque l'aspect documentation peut s'avérer fastidieux. De même, si on veut une documentation complète, il est intéressant d'avoir un certain nombre d'exemples de performances du module, dans un certain panel de condition (outils utilisés, device choisi…) et pour plusieurs configurations possibles. Ce travail peut être long, donc si nécessaire, on peut imaginer de mettre à jour la documentation pour chaque intégration du module dans d'autres conditions, et ainsi enrichir celle-ci de manière incrémentale.

Un document disponible en annexe, "*Documentation Guideline for Reusable Modules*", décrit la documentation à produire pour accompagner un module réutilisable. Comme un tel module est un peu comme un composant virtuel, cette documentation a l'apparence d'une spécification de produit d'un composant électronique.

Une fois le module complètement terminé, il sera généralement mis sur un serveur accessible à une équipe ou une compagnie entière. Pour cette raison, il convient de se fixer une structure commune de répertoire, permettant aux utilisateurs des modules de trouver instantanément ce qu'ils recherchent. La structure proposée est décrite dans le diagramme page suivante.

Le fichier revision_history.txt est très important, puisqu'il permet à un ancien utilisateur, de voir instantanément les évolutions du bloc qu'il a déjà utilisé, et aussi d'éventuels problèmes découverts et fixés ou non.

Concernant la gestion des révisions, on dissocie les révisions de module et les révision de documentation. Il est n'est pas concevable qu'une révision du module ne s'accompagne pas de révision de documentation, mais par contre, il est possible qu'une révision de documentation soit menée même si le code ne change pas (au moment d'enrichir la documentation par exemple). Pour cela, plusieurs versions de documentation peuvent accompagner un même module.



"TRUCS" ET EXEMPLES

INSTANCIATION D'UN DELAI GENERIQUE

Ceci est très utile lors pour générer un délai variable lors de l'alignement de chemins parallèles. Si le délai de traitement change, il est très facile de modifier la longueur du délai (une constante à changer, au lieu de un ou plusieurs signaux à déclarer ou supprimer et une ou plusieurs assignations à ajouter ou supprimer).

```
...

constant NUMB_OF_DELAY := 8;
type T_DELAYED_SUM_COLUMN is array (0 to NUMB_OF_DELAY) of unsigned (WIDTH-1 downto 0);
signal delayed_sum_column : T_DELAYED_SUM_COLUMN;

...

do_delay_sum_column : process(clk,sum_column)
begin
   delayed_sum_column(0) <= sum_column;    -- First value is not delayed
                                           -- Most delayed value at the greater index
   if rising_edge(clk) then
      for i in (1 to DEPTH) loop            -- Delay of the depth of the window
         delayed_sum_column(i) <= delayed_sum_column(i-1);
      end loop;
   end if;
end process;

...
```

INSTANCIATIONS GENERIQUES DE LIGNES A DELAI :

Lors de l'intégration de sous-blocs nécessitant des lignes à délai, une instanciation générique comme suit permet de limiter les changements au codage à une valeur de constante.

```
...

constant NUMB_OF_LD : natural :=4;
type T_ARRAY_OF_LD is array (0 to NUMB_OF_LD) of unsigned (DATA_WIDTH-1 downto 0);
signal successive_LD : T_ARRAY_OF_LD;

...

successive_LD(0) <= input_of_the_first_LD;

...

gen_line_delay : for i in 0 to NUMB_OF_LD - 1 generate
   U_line_delay : line_delay
      generic map( WIDTH  => DATA_WIDTH,
                   LENGTH => LENGTH)
      port map   ( clk    => clk,
                   in_LD  => successive_LD(i),
                   out_LD => successive_LD(i+1));
   end generate gen_line_delay;

...
```

ADDITIONNEUR PIPELINE

On a ici 2 étages de pipeline, donc on utilise un process par étage.

```
...

do_first_stage_sum : process (clr_a,clk)
begin
   if clr_a='1' then
      sum_1_st1 <= (others => '0');
      sum_2_st1 <= (others => '0');
   elsif rising_edge(clk) then
      sum_1_st1 <= first_input + second_input;
      sum_2_st1 <= third_input + fourth_input;
   end if;
end process first_stage_sum;

do_second_stage_sum : process (clr_a,clk)
begin
   if clr_a='1' then
      sum_ st2 <= (others => '0');
   elsif rising_edge(clk) then
      sum_ st2 <= sum_1_st2 + sum_2_st1;
   end if;
   end process do_second_stage_sum;

...
```

MODULE AVEC NOMBRE DE PORTS GENERIQUE

Il peut souvent être utile d'avoir un nombre de port d'un module qui soit dépendant d'un paramètre générique, permettant d'avoir un degré de configuration supplémentaire.

Le problème posé par ce type de configuration est que le nombre de ports d'une entité est fixe. La seule manière est de jouer sur la taille des ports. Ainsi, la solution est de concaténer les ports dans un seul vecteur, comme dans l'exemple suivant. Cette solution peut paraître ambiguë, mais à moins de passer par un outil plus puissant (C++) pour générer automatiquement du code VHDL à partir d'un jeu de paramètre, il semble que ce soit la seule solution.

```
...

entity filter is
   generic (DATA_WIDTH    : natural := 8;
            NUMB_OF_PORTS : natural := 3)
   port    (clk           : in  std_logic;
            in_filter     : in  std_logic_vector ((DATA_WIDTH*NUMB_OF_PORTS)-1 downto 0);
            out_filter    : out std_logic);
end consolidation;

...
```

Instanciation du module :

```
...

U_filter : filter
   generic map(DATA_WIDTH    => 8,
               NUMB_OF_PORTS => 3)
   port map   (clk           => clk,
               in_filter     => in_LD0 & in_LD1 & in_LD2,
               out_filter    => out_filter);


...
```

FONCTION LOG2

Cette fonction est nécessaire lorsque l'on doit calculer la largeur d'un bus en fonction de la plage de valeurs à coder. Cette fonction est généralement à inclure dans le package général du design, ou au début de l'architecture du module s'il elle n'est pas utile aux autres modules du système.

```
...

--================================================================================================
-- Function : Determine the log base 2 of a natural constant
--            The valid range of the input is 1 to 2**15
--            The result is rounded down
--            Used to determine the width of a bus
--================================================================================================

function log2(n: natural) return natural is
variable tmp : natural :=0;
begin
   tmp := 0;
   for i in 0 to 15 loop         -- i raises until the end of the loop
      if (n >= 2**(i+1)) then
         tmp := tmp + 1;         -- tmp variable raises until n >= 2**(i+1),
                                 -- meaning (i+1) >= log2(n)
                                 -- corresponding to i = round down(log2(n))
      end if;                    -- tmp remain unchanged after
   end loop;
   return tmp;                   -- tmp corresponds to the value we want to calculate
end log2;

...

WIDTH = log2(MAX_VALUE) ;

...
```

FONCTIONS DE CONVERSION

Les fonctions de conversion peuvent faire apparaître des lignes de code assez large et difficilement lisibles, lorsque l'on doit passer par un type intermédiaire sans vouloir créer de signal intermédiaire (dans ce cas, on voit mal à quel fonction se rapporte un certain paramètre). L'idée ici est de ne mettre qu'une seule commande par ligne et d'indenter le code pour qu'un paramètre se trouve aligné avec la fonction de conversion à laquelle il est associé. On identifie également tout de suite le nom du signal converti, qui se trouve dans la partie la plus indentée.

```
...

term_2_low    <=   conv_unsigned (
                          conv_integer (
                             in_term_2 (WIDTH_TERM_2_LOW-1 downto 0)
                          ),
                   WIDTH_TERM_2_LOW);

...
```

ASSERTION POUR NON-VALIDITE DE PARAMETRE

Ce code permet de générer une erreur à la simulation lorsqu'un paramètre outrepasse la plage de valeurs admissibles. Par contre, la synthèse ne donnera pas de message d'erreur.

```
...

constant MIN_THRESH : natural := 1 ;
constant MAX_THRESH : natural := 31;

...

do_check_thresh_validity : process
begin
   assert (THRESH >= MIN_THRESH)
      report "Minimum value for THRESH is " & natural'image(MIN_THRESH)
           & " Current value is " & natural'image(THRESH)
      severity failure;
   assert (THRESH <= MAX_THRESH)
      report "Maximum value for THRESH is " & natural'image(MAX_THRESH)
           & " Current value is " & natural'image(THRESH)
      severity failure;
   wait;
end process do_check_thresh_validity;

...
```

BIBLIOGRAPHIE

- M. Keating, P. Bricaud, *Reuse Methodology Manual For System-On-A-Chip*, KAP 2000
- J. Bergeron, *Writing Testbenches, Functional Verification Of HDL Models*, KAP 2000
- Qualis & Xilinx, *FPGA Reuse Field Guide*
- Xilinx, *Xilinx Design Reuse Methodology for ASIC and FPGA Designers*

ANNEXES

VHDL WORK METHOD

INTRODUCTION

The goal of this document is to improve the readability of the written code. This amelioration, will in turn improve communications between different individuals having different coding styles.

PHILOSOPHY

The VHDL blocks should be stored in separate directories on the server.

All files (VHDL source, VISIO, doc, test-bench, simulation vectors, C code for generating vectors,...) regarding a block should be in the same directory (no hierarchy, since each block should be relatively small).

Only one test case should be place in this directory unless it is necessary to have many test cases .

It would be preferable if all TAB characters be replaced by spaces (UltraEdit can easily do this) so that the code looks the same regardless of the editor. In the same way, limit the length of the lines to 80 characters

DIRECTORY CONTENT FOR A VHDL BLOCK

README

A file named "README" should placed in the directory. This file should explain the function of all other files in the directory. For video processing, this file should specify the form of the input/output (demultiplexed (each component separated) or multiplexed (Cb, Y, Cr, Y, Cb, …), or otherwise,…). You can also add a set of performances of the module (required material and maximum clock frequency AFTER P&R).

Concept

Some files should be included to explain the concept of the block. Typically, a visio file should be sufficient. In this case, signals names used in the visio schematics should match the signal names used in VHDL files.

Files and directories

As much as possible put only one entity/architecture pair per file.

As much as possible the name of the file should be the same as the name of the entity declared in the file

Packages should be put in a file having the suffix "_pkg".

Test bench files (and stuff that is typically not synthesizable) should have their names

suffixed with "_tb".

As much as possible, big designs should have their files in a hierarchy of directories. One of those directory should be dedicated to building blocks shared among different modules. This should also help cooperative work.

VHDL source code

The top level of the block should specify the form of the input or output. This is to allow the same function to be used in the same design with different I/O format.

Example :

```
FIR_lowpass_io_demuxed.vhd
FIR_lowpass_in_YCbCr_out_demuxed.vhd
FIR_lowpass_in_YCbCr_out_YCbCr.vhd
```

This convention creates long names but should make the use of the README file less important.

Test-bench source code and validation files

The VHDL test-bench file used to validate the block, should have the prefix "_tb" to stress the fact that they are not part of the design.

IDENTIFIER'S NAME CONVENTION

Long descriptive identifiers should be used. Each word (except the first one) of the descriptive identifier should be preceded by an underscore, don't use a mix of upper and lower case. The descriptive identifier is usually more useful than a comment for 2 reasons: 1) after some time, the comment does not match the actual code… 2) a descriptive identifier is a "lighter" way (for the writer AND the reader) to document code.

Example :

```
signal  the_current_data_format : integer;
```

Two main factors affect the identifier naming convention: Type and Context. The effect of these factors will be reflected by the addition of a prefix and/or postfix to the main part of the identifier. In the rest of this document, the main part of the identifier (descriptive part) will be referred to as "mainName".

CONSTANTS AND GENERICS

Use all upper cases.

Example :

```
constant MASTER_CLK    : time := 9.25 ns;
generic( WIDTH  : natural := 8,
         LENGTH : natural := 1);
```

The use of constants is highly recommended : the only hard-coded constants should be 0, 1 and 2.

LOCAL VARIABLES AND SIGNALS

When judged necessary, the prefix "the" or "a" could be used for variables or signals. This is usually used to with variables to specify that its scope is local.

Example :

```
variable  the_count : integer;
```

BOOLEAN TYPE

The identifier of a boolean variable/signal should formulate a question to which the answer specifies the polarity of the logic (negative or positive logic). The prefix "is" or "are" followed by a meaningful description can often serve that purpose.

Example :

```
variable is_good : boolean;
```

PROCESS, PROCEDURE AND FUNCTIONS

Use the prefix "do" followed by a meaningful description of the parameter.

Example :

```
procedure do_read_file(…)…
do_logic : process …
```

Nota: The label name of the process should never have the same name as an internal signal, this will cause an error in the ModelSim environment simulator.

GENERATE STATEMENTS

Use a label beginning by the prefix "gen", followed by a meaningful description of the component generated.

Example :

```
gen_line_delay : for i in (0 to NUMB_OF_LD-1) generate …
```

TYPE AND SUBTYPE

Use the prefix "T" followed by a meaningful description of the parameter starting with a lower case letter. Try to minimize the use of subtypes. Too many subtypes is not helpful for the readability of the code.

Example :

```
subtype T_VIDEO is natural range 0 to ((2**WIDTH)-1);
```

SUB-PROGRAM PARAMETERS AND ENTITY PORTS

Input

Use the prefix "in" followed by a meaningful description of the parameter.

Example :

```
port(in_data : in std_ulogic) …
or
procedure do_open_file(constant IN_FILENAME : in string) …
```

Output

Use the prefix "out" followed by a meaningful description of the parameter.

Example :

```
port(inData    : in  std_ulogic,
     outResult : out std_logic) …
or
procedure do_something(variable out_result : out natural) …
```

Input/Output

It is strongly recommended to avoid the "inout" qualifiers for signals. If an output signal has to be read, it is preferable to use an internal signal for reading and assigning the latter to an output port.

Example :

```
the_count <= the_count + 1;
out_count <= the_count;
```

USEFUL PREFIX AND SUFFIX FOR IDENTIFIER'S NAME

Clocks

Use the prefix "clk" followed by a meaningful description for a clock signal.

Example :

```
clk, clk_master, clk_135, …
```

Delayed signals

To specify that a signal is a delayed version of another signal, use the prefix "**d***X*", where *X* refers to the number of delays. If *X* is omitted, it is assumed to be "1".

Example :

```
d1_value, d2_value, …
```

Counters

Use the prefix "cnt" followed by a meaningful description for a signal at the output of a counter (the count itself).

Example :

```
cnt, cnt_modulo4, cnt_H, …
```

Addition

Use the prefix "sum" followed by a meaningful description for a signal resulting from an addition.

Example :

```
sum_values, sum_st1, …
```

Subtraction

Use the prefix "diff" followed by a meaningful description for a signal resulting from a subtraction.

Example :

```
diff_values, diff_state2, …
```

Control/Status register (to be read or written by a micro-controller)

Use the prefix "reg" followed by a meaningful description for a signal at the output of a register.

Example :

```
reg, reg_status, reg_control, …
```

Address

Use the prefix "addr" followed by a meaningful description for a signal representing an address.

Example :

```
addr_ROM, addr_RAM, …
```

Line delays

Use the prefix "**LD***x*" followed by a meaningful description for a signal at the output of a line delay. Here, "*x*" is the number of line delays between this signal and the original. If *x* is omitted, it is assumed to be "1".

Example :

```
LD_input, LD2_input, LD3_input, …
```

Enables

Use the prefix "ena" followed by a meaningful description for a signal used to enable a component.

Example :

```
ena, ena_cnt, …
```

Multiplexer selector signal

Use the prefix "sel" followed by a meaningful description for a signal used to select an input of a multiplexer.

Example :

```
sel, sel_mode, …
```

Instance of a component

Use the prefix "**U***x*" followed by the name of the entity for an instantiation, where *x* is an instance number. If *x* is omitted, it is assumed to be "1".

Example :

```
U1_fast_counter, U2_fast_counter, …
```

Active-low signal

Use the suffix "n" for an active-low signal.

Example :

```
write_n, ce_n, …
```

Asynchronous signal

Use the suffix "a" for an asynchronous signal (should be limited to an asynchronous reset, since nearly every signal should be synchronous).

Example :

```
rst_a, …
```

Registered signal

Use the suffix "r" for a signal that you want to register for the purpose of improving speed performances (typically in I/O cells).

Example :

```
signal_r  <=  signal;
```

Pipelined structures

You can use the suffix "st*i*" to precise the stage of a pipelined structure

Example :

```
Sum_st1, sum_st2 …
```

Data type

You can use a suffix to identify the type of a signal (it can be useful for type conversion).

- _slv for std_logic_vector
- _uns for unsigned
- _sgn for signed
- _int for integer
- _nat for natural

Example :

```
data_slv  <=  conv_std_logic_vector(in_data);
```

FILE HEADERS

Example :

```
--------------------------------------------------------------------------------------------
-- TITLE        : Half-band filter with 7 taps.
-- DESCRIPTION  : [ -5, 0, 37, 64, 37, 0, -5 ]
-- FILE         : halfband7taps.vhd
--------------------------------------------------------------------------------------------
-- CREATION
--      DATE         AUTHOR              PROJECT             REVISION    COMMENTS
--      2000/10/25   Robert Noory        ADVP                2.0
--------------------------------------------------------------------------------------------
-- MODIFICATION HISTORY
--      DATE         AUTHOR              PROJECT             REVISION    COMMENTS
--------------------------------------------------------------------------------------------
-- Copyright (c) 2000 Miranda Technologies inc.
--------------------------------------------------------------------------------------------
```

DOCUMENTATION GUIDELINE FOR REUSABLE MODULES

PRELIMINARY PROPOSITION

This document describes the documentation to be delivered with a reusable module.

The documentation should be only one document, with different subsections, easily identifiable, depending on what kind of information the user wants to get (from basic integration information to advanced information about the detailed function of the module)

The document should be written in English, and delivered in *pdf* format. A *Microsoft Word* template will be available.

As the module is a reusable module, the documentation should look like a component data-sheet, or a product specification.

Concerning the revision history, a separate file should indicate which minor and major changes have been made. A new component specification has to be delivered for any change. In the specification, just precise which version is described.

CONCEPT (OPTIONAL)

If the module has an algorithm content, a high-level description is to be made in this section. Explain the principle of the algorithm. Diagrams or schematic views are well adapted, especially for video concepts.

FEATURES

An overview of the module, with a brief description of the main features of the module. Limit the description to 1 or 2 lines for each.

For example :

- Basic functionality
- Range of the input
- What is configurable
- Pipeline level

- …

LIMITS AND RESTRICTIONS

This section should clearly give the known restrictions of the module. For example :

- Performances limitations
- Devices or tools not supported
- …

PINOUT

Schematic view of the module(external view), with I/Os, sizes…

Write the VHDL code of the declaration of the entity. You can also add an example of the mapping (generic map and port map) of the component.

A table should be included, describing each I/O and its purpose

Port name	Direction	Type	Size / range	Description
in_pixel	input	std_logic_vector	(DATA_WIDTH downto 0)	…

GENERIC PARAMETERS

Follow the example table to give a detailed description of each user-defined parameter. When required, write a paragraph if the description in the table is too short.

Parameter name	Type	Values		Description
		Valid range	Already tested	
DATA_WIDTH	integer	2 to 32	8 – 10 – 12	…

It is very important for the user to know the range of valid values. Clearly specify which values have been successfully validated, synthesized and implemented, and which should be valid but not validated yet. Thus, the user knows very quickly if the module is already fully validated for the required parameters, or if he will need more time to validate the module.

LATENCY

Specify the latency of the block, in number of clock cycle (precise which clock is concerned, for multi-clock systems).

If the latency depends on a configuration or of an input's size, give the relation used to determine it, and give a few examples for representative configurations. You can provide non-synthesizable VHDL to calculate the latency, if it depends on generic parameters. Ex : `LATENCY := DATA_WIDTH + 2;`

Also give the number of cycles per result (usually 1 clock cycle produces 1 result, but in some cases, we can have less than 1 result/cycle)

FUNCTIONAL DESCRIPTION

This section quickly describes the functionality of the module, but should not be very detailed. Try to give an external view of the system, or eventually a global block-diagram of the architecture.

PERFORMANCES

The goal of this section is to indicate the performances that the user can expect from this module. The expected working frequency and the required resources to implement the function should be indicated. The chosen configurations should be representative of the most popular use of the function.

The following table is required.

Configuration		Working frequency		Resources (after P&R)		
DATA_WIDTH	…	After synthesis	After P&R	Flip-Flops	LUTs	RAM Blocks
8	…	117 MHz	109 MHz	112	97	3
10	…	103 MHz	95 MHz	147	131	4

Information to clearly indicate :

- the tools (including the version) used to get these results
- the options used for the tools (resource sharing, P&R effort level…)
- the constraints used, for synthesis and P&R
- the targeted device, with its speed grade

The same table can be made for another device (one for a Xilinx device, and one for an Altera device, for example)

For Virtex devices, it is important to indicate the necessary resources in flip-flops and LUTs (and, if applicable, other resources such as RAM Blocks, DLL, hardware multipliers…), rather than slices or CLBs, because P&R tools usually optimise for speed, when the device is not full.

You must be aware that the indicated results are given with no warranties, corresponding to this ideal case where the module is alone on the FPGA. Thus, the performances can

be considered as best-case. However, if the module is on a critical path, you can put strong constraints on it and get results not far from this ideal case.

TOOLS AND LIBRARIES

You have to clearly indicate which tools (synthesis, simulation and P&R) support the module and which libraries are necessary (IEEE, UNISIM, SIMPRIM, VITAL…), and the versions used.

If a core is used, join its source and documentation, and describe how it was generated (which tool and version used)

This list can be updated if new tools are available and support the module.

If a tool is not supported, describe the problem if you identified it.

FILES

Give the list of the files needed to implement the function, in the order in which it should be compiled. For a hierarchical design, give a short description of each sub-module function.

DETAILED DESCRIPTION

This section describes precisely how the module works, and presents the detailed architecture.

A theoretical part can be added, when required, if not done in the functional description.

You have to present a view of the architecture, and decompose it in simple blocks (each often corresponds to a single file).

Describe precisely the function of the sub-blocks, from external view, I/Os description to detailed architecture of each one. If the sub-blocks are already fully documented modules, you can refer to its documentation.

You can draw timing diagrams if necessary.

STATUS OF THE MODULE

This section describes the status of the module regarding its validation status (only simulation, or hardware functional), and the project which have (successfully or not) integrated the module. It could be good to precise which configuration of the block was

used in each project.

REVISION HISTORY

The revision history should shortly describe both module revisions and documentation revisions.

HDL RESERVED WORD

VHDL

abs	configuration	impure	Null	rem	type
access	constant	in	Of	report	unaffected
after	disconnect	inertial	On	return	units
alias	downto	inout	Open	rol	until
all	else	is	or	ror	use
and	elsif	label	others	select	variable
architecture	end	library	out	severity	wait
array	entity	linkage	package	signal	when
assert	exit	literal	port	shared	while
attribute	file	loop	postponed	sla	with
begin	for	map	procedure	sll	xnor
block	function	mod	process	sra	xor
body	generate	nand	pure	srl	
buffer	generic	new	range	subtype	
bus	group	next	record	then	
case	guarded	nor	register	to	
component	if	not	reject	transport	

VERILOG

always	end	ifnone	or	rpmos	tranif1
and	endcase	initial	output	rtran	tri
assign	endmodule	inout	parameter	rtranif0	tri0
begin	endfunction	input	pmos	rtranif1	tri1
buf	endprimitive	integer	posedge	scalared	triand
bufif0	endspecify	join	primitive	small	trior
bufif1	endtable	large	pull0	specify	trireg
case	endtask	macromodule	pull1	specparam	vectored
casex	event	medium	pullup	strong0	wait
casez	for	module	pulldown	strong1	wand
cmos	force	nand	rcmos	supply0	weak0
deassign	forever	negedge	real	supply1	weak1
default	for	nmos	realtime	table	while
defparam	function	nor	reg	task	wire
disable	highz0	not	release	time	wor
edge	highz1	notif0	repeat	tran	xnor
else	if	notif1	rnmos	tranif0	xor