Approche Système de la Conception

ENSEA

ETIS / ENSEA
Mel : Bertrand.Granado@ensea.fr

Printemps 2012

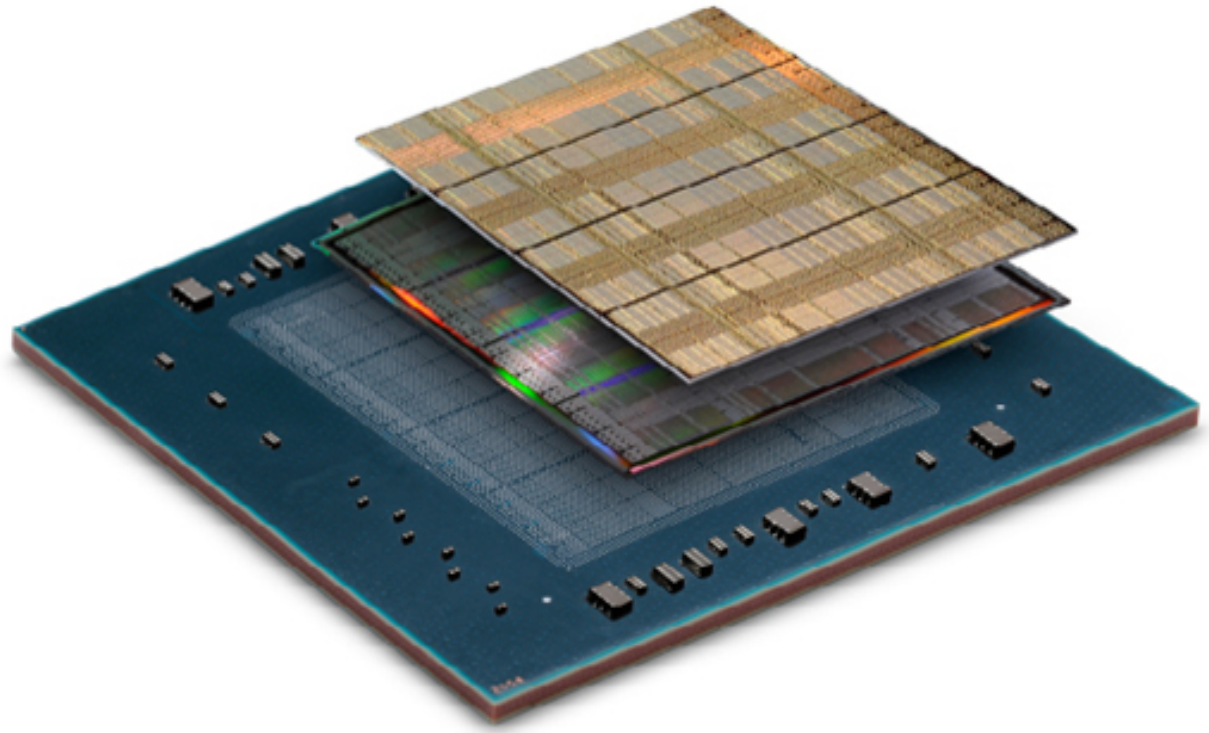
Sommaire

Introduction

Comment le réaliser ?

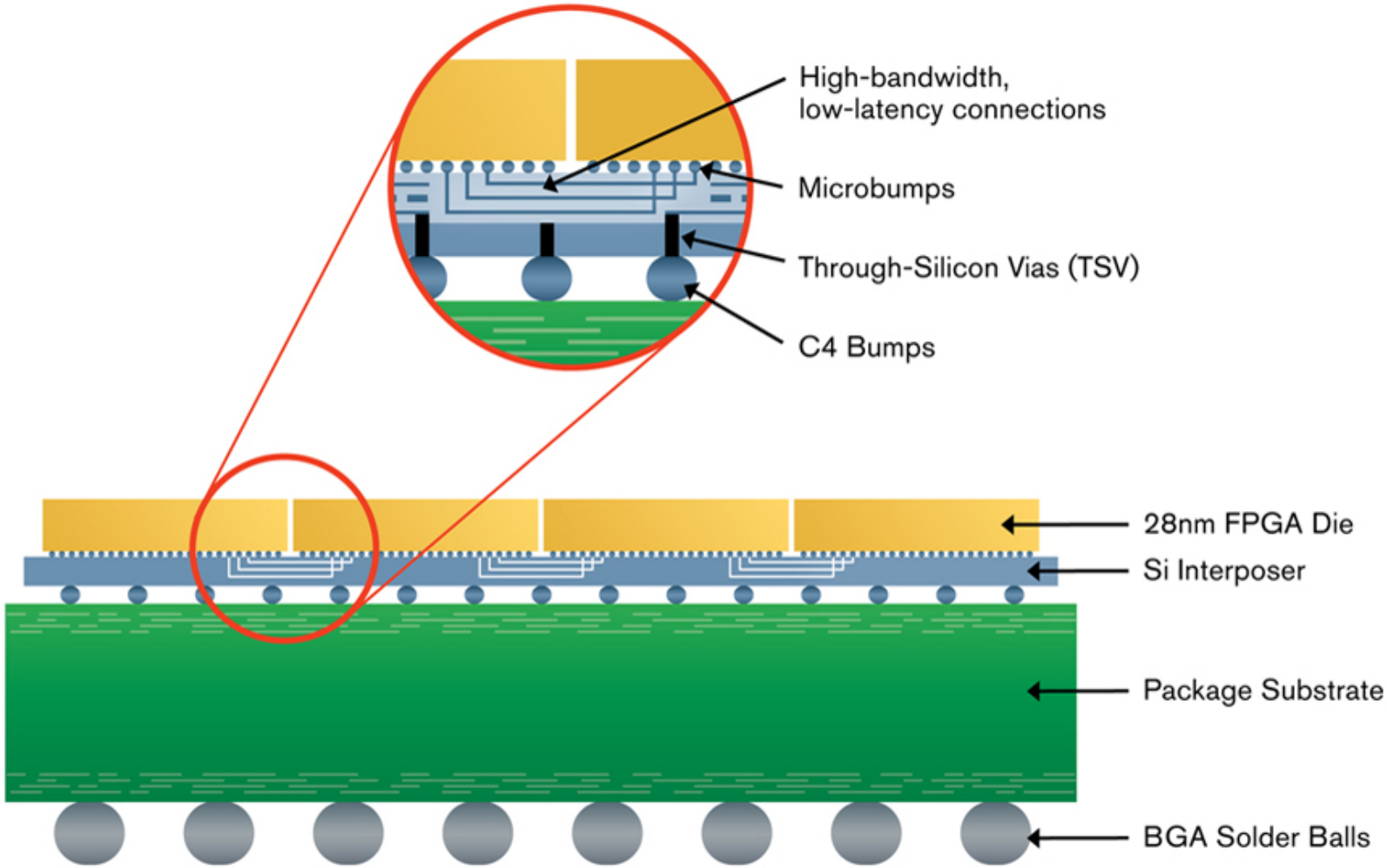
Des idées ?

Systèmes Sur Puce (SoC)

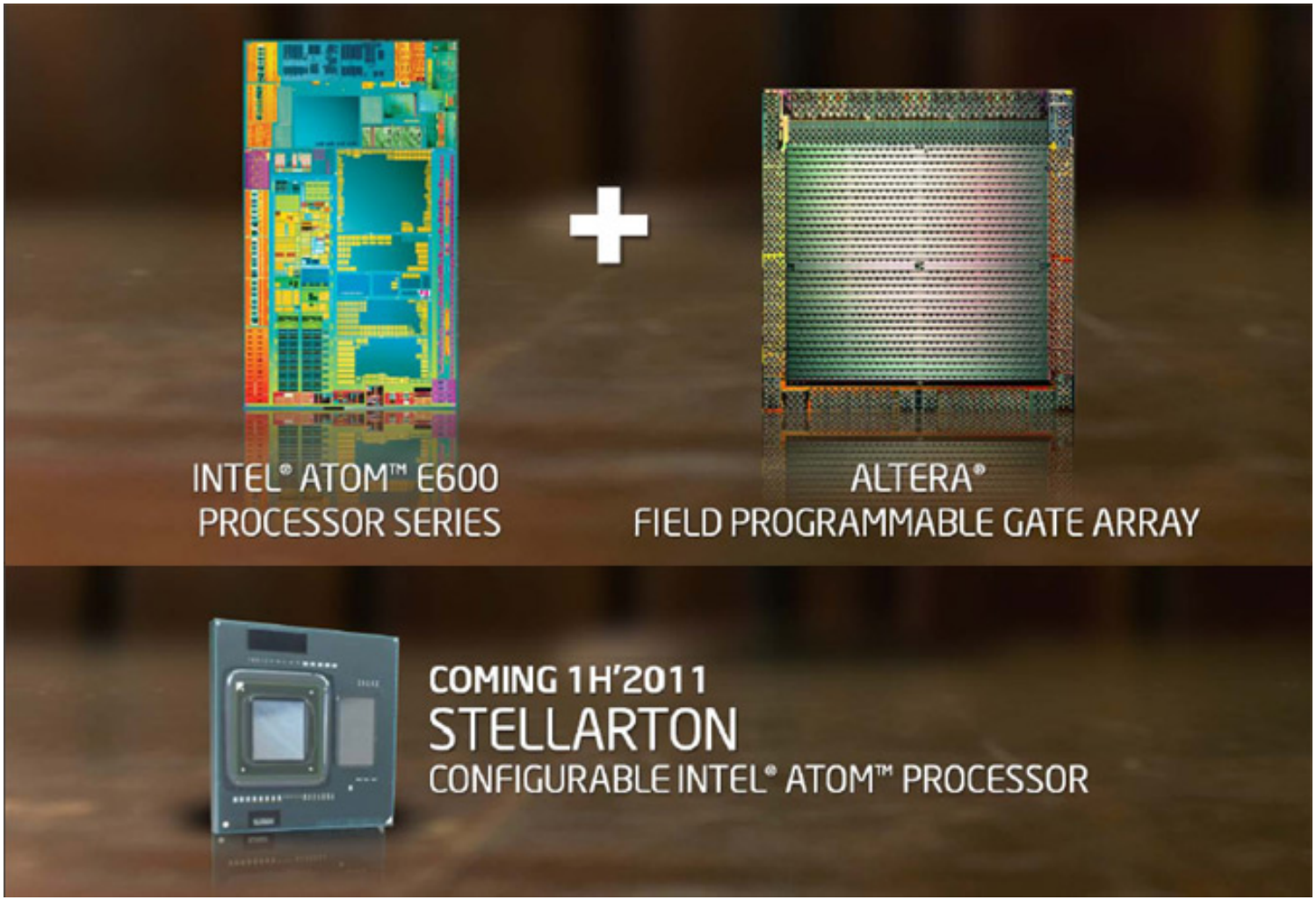


Virtex-7 2000T FPGA Utilizing
Stacked Silicon Interconnect Technology

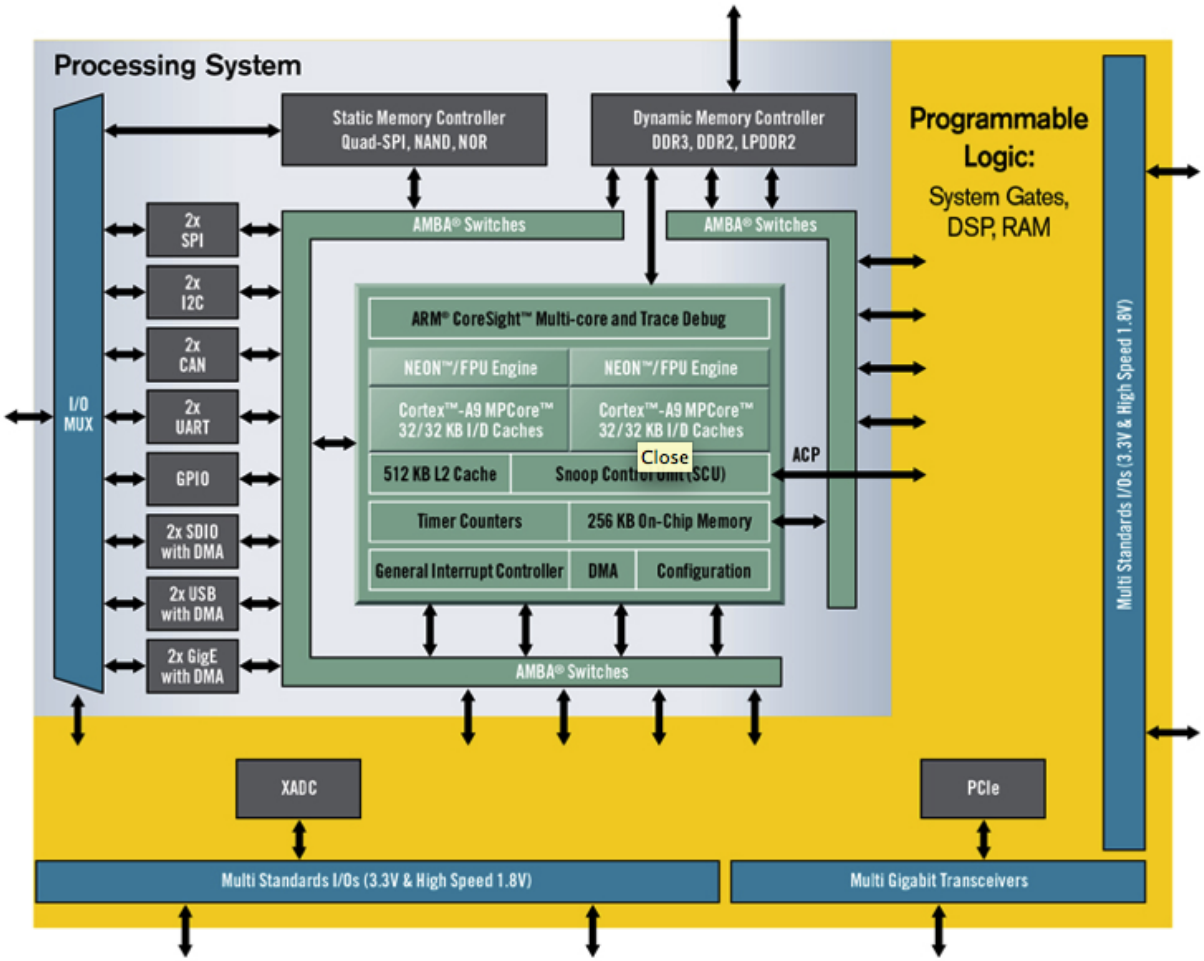
Systèmes Sur Puce (SoC)



Systèmes Sur Puce (SoC)

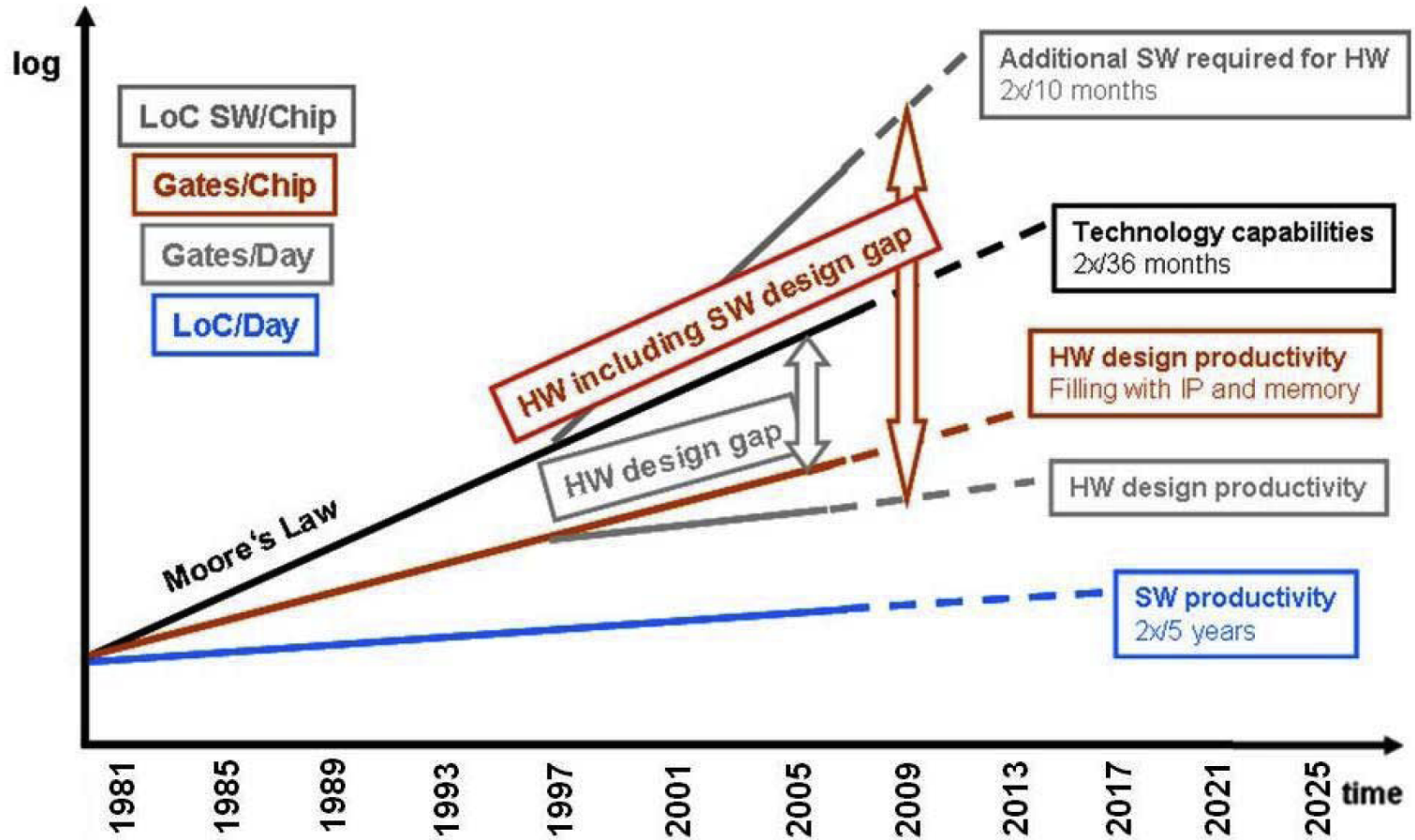


Systèmes Sur Puce (SoC)



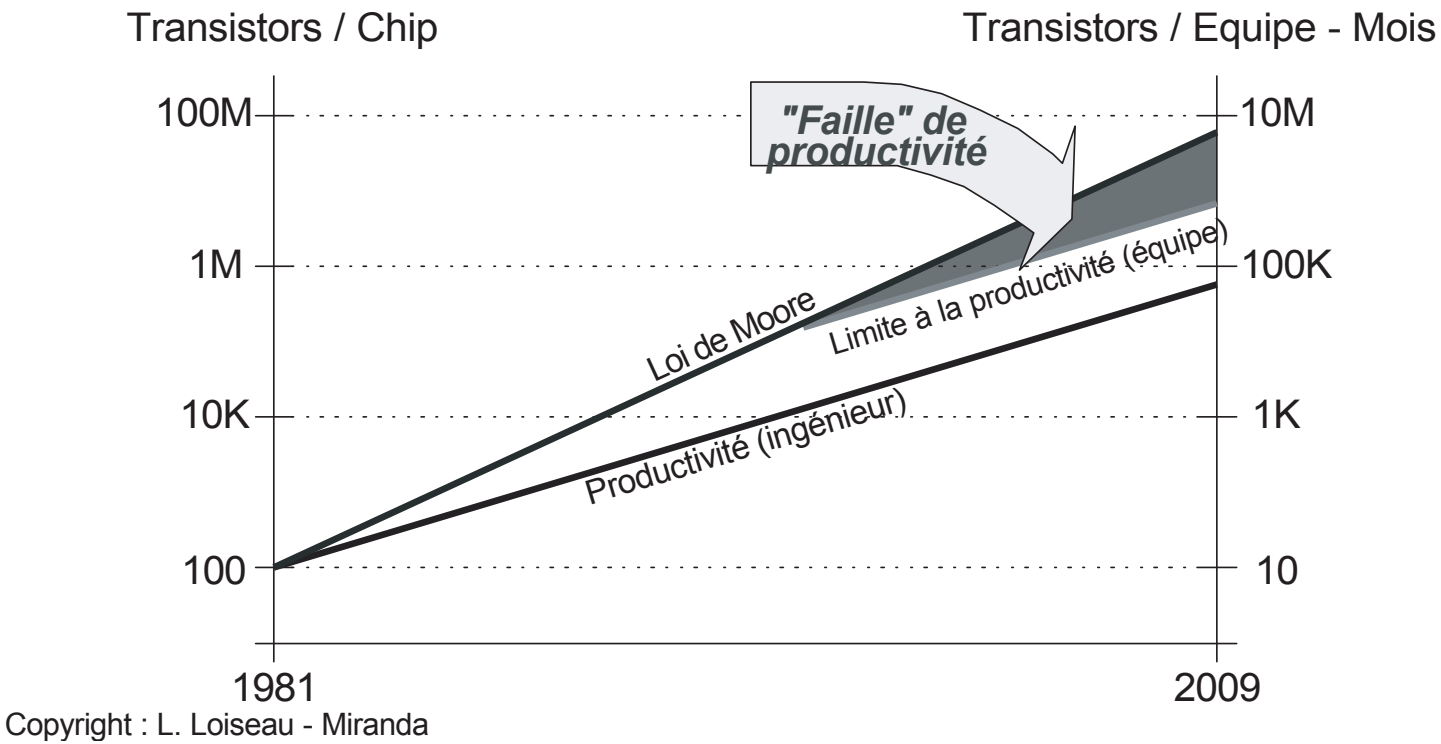
- Dual ARM Cortex™-A9 MPCore
  - Up to 800MHz
  - Enhanced with NEON Extension and Single & Double Precision Floating point unit
  - 32kB Instruction & 32kB Data L1 Cache
- Unified 512kB L2 Cache
- 256kB on-chip Memory
- DDR3, DDR2 and LPDDR2 Dynamic Memory Controller
- 2x QSPI, NAND Flash and NOR Flash Memory Controller
- 2x USB2.0 (OTG), 2x GbE, 2x CAN2,0B 2x SD/SDIO, 2x UART, 2x SPI, 2x I2C, 4x 32b GPIO
- AES & SHA 256b encryption engine for secure boot and secure configuration
- Dual 12bit 1Msps Analog-to-Digital converter
  - Up to 17 Differential Inputs
- Advanced Low Power 28nm Programmable Logic:
  - 28k to 235k Logic Cells (approximately 430k to 3.5M of equivalent ASIC Gates)
  - 240kB to 1.86MB of Extensible Block RAM
  - 80 to 760 18x25 DSP Slices (58 to 912 GMACS peak DSP performance)
- PCI Express® Gen2x8 (in largest devices)
- 154 to 404 User IOs (Multiplexed + SelectIO™)
- 4 to 12 12.5Gbps Transceivers (in largest devices)

Systèmes Sur Puce (SoC)



Systèmes Sur Puce (SoC)

Gain de productivité



Sommaire

UML

- Un cadre de travail

UML

- Un cadre de travail
- Support d'une méthode de conception dirigée par les modèles

Bertrand
LE GAL

Maître de conférences

ENSEIRB
bertrand.legal@enseirb.fr
http://www.enseirb.fr/~legal/

Laboratoire IMS
bertrand.legal@ims-bordeaux.fr
Université de Bordeaux 1
351, cours de la Libération
33405 Talence - France

Filière Electronique
2ème année
2009 / 2010

"Introduction au langage UML"

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

1

“ Retour sur le flot de Conception ”

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Les causes courantes des échecs de projets

- *Mauvaise compréhension* des besoins des utilisateurs finaux.
- Incapacité à gérer les *modifications des exigences* des clients au cours du développement.
- Les modules composant le projet ne fonctionnent pas ensemble (*mauvaises interfaces*)
- *Code source* du logiciel *difficile à maintenir* ou à *faire évoluer*
- Logiciel de *mauvaise qualité* (beaucoup d'anomalies de conception / exécution)

L'idéal pour la gestion d'un projet !

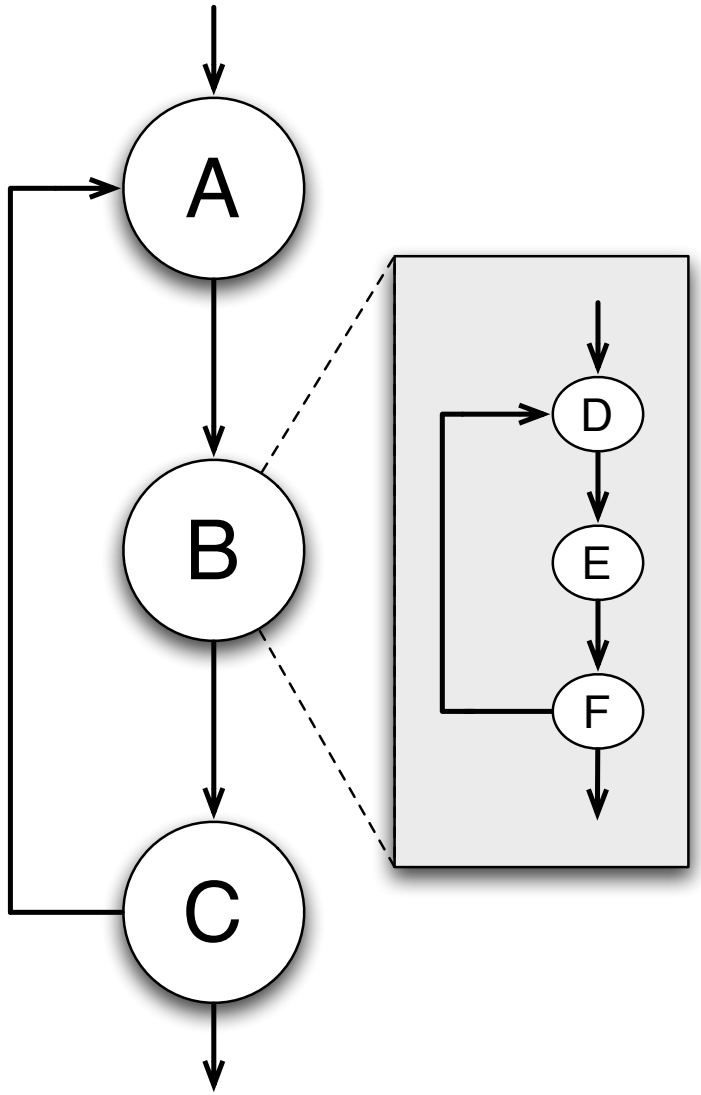
1. *Bien comprendre les besoins*, les demandes et les exigences des utilisateurs finaux.
2. *Bonne communication avec le client* pour valider certains choix et vérification de l'étape (1).
3. *Tester et valider chaque phase* de la conception pour ne pas découvrir des problèmes plus tard.
4. *Traiter au plus tôt les problèmes*.
5. *Maîtriser la complexité* et augmenter la réutilisation.
6. Définir une *architecture robuste*.
7. Faciliter le *travail en équipe*.

Les 7 bonnes pratiques

1. Développement *itératif*,
2. Développement à base de *composants* centré sur l'architecture,
3. Pilotage du *développement par les risques*,
4. Gestion des *exigences*,
5. Maîtrise des modifications,
6. *Evaluation continue de la qualité,*
7. Modélisation visuelles.

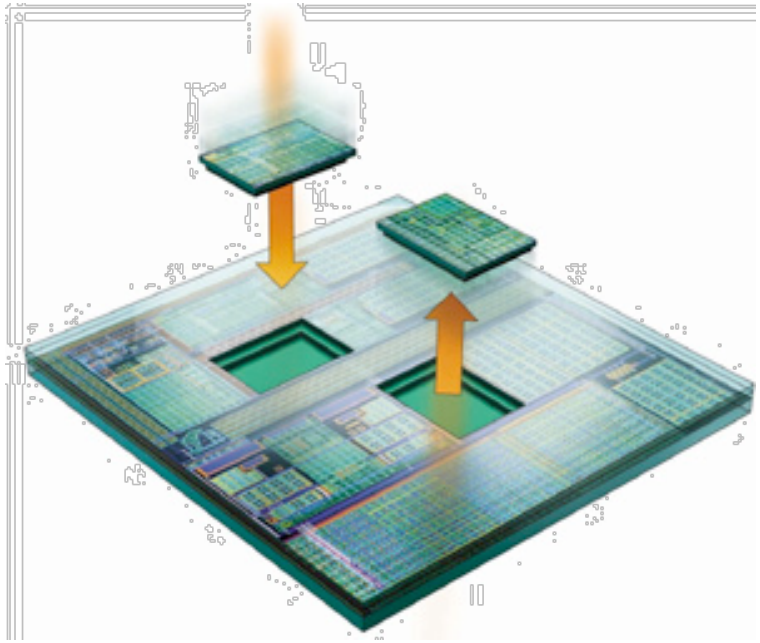
1 - Le développement itératif

- Cette méthode est basée sur de petites étapes (élémentaires)
- Chaque étape produit des retours et des adaptations si nécessaire
- Autres dénominations
  - Développement en spiral
  - Développement évolutif



2. Développement basé sur les composants

- Validation de l'architecture logicielle lors des premières itérations :
  - Développement basé sur des IP du commerce,
  - Réutilisation préférée au redéveloppement,
- Tester l'architecture au plus tôt même si elle n'est pas fonctionnellement complète.



Méthodologie valable dans le cadre du développement de systèmes logiciels et matériels

3. Pilotage par les risques

- Analyser les risques potentiels au plus tôt (début de la conception) :
  - Les regrouper et les hiérarchiser,
  - Travailler en priorité sur les risques critiques,
- Cela concerne les risques techniques, ainsi que :
  - Les risques liés aux clients,
  - Les risques liés au domaine applicatif,
  - Les risques liés à l'organisation du projet,
- Sémantique
  - *Un risque est un événement redouté dont l'occurrence est plus ou moins prévisible et qui aura des répercussions sur le projet.*
  - *Un problème est un risque qui s'est révélé.*

4. Les exigences du client

- Sémantique
  - *Une exigence est une condition à laquelle le système doit satisfaire ou une capacité dont il doit faire preuve.*
- Les exigences peuvent être :
  - Fonctionnelles
    - Elles décrivent ce que le système doit savoir faire,
  - Non fonctionnelles
    - Qualité des services,
    - Temps de réponse, temps de traitement,
    - Sécurité au fonctionnement,
    - IHM adaptée aux utilisateurs.

5. Les demandes de changement

- Sémantique
  - *Une demande de changement est une requête visant à modifier un artefact ou un processus.*
- Il faut dans ce cas indiquer dans la documentation, la nature du changement, *sa cause* et *ses impacts* sur la solution proposée.
- Il existe 2 types de changements :
  1. La demande d'*évolution*
     - ➡ Nouvelle caractéristique du système ou modifications d'une existante
  2. Le rapport d'*anomalie*

6. Evaluation constante de la qualité

- Il a été prouvé que l'identification des erreurs de conception au plus tôt permet de "gagner" de l'argent (ou d'en perdre moins)
  - Ratio 10 à 100 !
- Les problèmes de qualité sont souvent les causes principales des dépassements de délais,
- Les phases de test, de vérification et de correction consomment environ 70% du temps total d'un projet !
- 60% des problèmes existent déjà lors de la conception de l'application.

Pourquoi utiliser la modélisation visuelle ?

- On doit pouvoir mémoriser nos pensées et les communiquer en utilisant des langages visuels et schématiques.
  - Réseaux de Pétri, Grafcet,
  - Schémas blocs (décomposition fonctionnelle),
  - UML, etc.
- Avantages de ces approches
  - Les *langages visuels* sont naturels et *faciles à comprendre* (non relatifs à un langage de communication),
  - On estime que 50% du cerveau est impliqué dans le processus visuel.

2

“ Introduction à UML ”

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Introduction à UML

- UML est un langage standard conçu pour permettre aux concepteurs d'élaborer les plans des logiciels qu'ils doivent développer.

- On se sert du langage UML pour:
  - *Visualiser* => *Schématiser le travail à réaliser*,
  - *Spécifier* => *Exprimer formellement les contraintes*,
  - *Construire* => *Commencer à développer une solution*,
  - *Documenter* => *Expliquer les choix réalisés (voir les problèmes)*,
  - *Communiquer* (travailler en équipe),

UML est un langage

- Ce langage possède un *vocabulaire et des règles* qui permettent de *combiner les mots* afin de communiquer.
- La modélisation permet de comprendre le système à concevoir de manière formelle,
  - Tous les modèles sont liés les uns avec les autres afin de représenter le système et les interactions entre les blocs élémentaires.
- Le vocabulaire et les règles d'écriture régissent la manière de construire et de lire les modèles correctement mis en forme.
  - *Aucune décision d'implémentation n'est prise lors de la phase de spécification et d'analyse du cahier des charges.*

UML, un langage pour visualiser

- Pour certains développeurs, il y existe un lien direct entre une idée et son code d'implémentation.
  - Utilisation d'un *modèle de représentation mental*,
- Problème dans le partage de ces modèles avec d'autres personnes,
  - *Risques d'incompréhension*, d'incertitudes...
  - Chaque organisation et/ou personne possède son propre langage (difficulté d'intégration),
  - Risque d'oubli des décisions prises avec le temps !
- *UML permet de résoudre les problématiques du partage et de mémorisation de l'information de manière visuelle.*

UML, un langage pour visualiser

- UML est bien plus qu'un assemblage de symboles graphiques !
  - Chaque symbole graphique possède sa sémantique propre (signification universelle),
  - Un modèle peut être écrit par un concepteur et compris pas un autre sans ambiguïté à la lecture,
  - Une automatisation par outil peut ainsi être mise en place pour traiter certaines informations => Génération de code source,
- UML est un *méta-langage de modélisation* permettant d'unifier les modèles utilisés dans les méthodes de développement (raffinement).

UML, un langage pour spécifier

- Dans le cadre d'UML, *spécifier signifie construire sans ambiguïté*,
- UML offre la possibilité de spécifier le comportement de l'application de manière formelle,
  - Réflexion sur des *modèles formels*,
  - *Approche indépendante du langage* objet d'implémentation (C++, Java, etc.),
  - *Raffinement progressif des modèles* (on construit une solution de manière itérative en plusieurs passes successives),
- UML n'est pas une méthode mais une notation qui laisse la liberté de conception (choix dans les solutions à mettre en oeuvre).

UML, un langage pour documenter

- Lors du développement d'un produit, on réalise des choix lors de l'élaboration de la solution :
  - Architecture d'implémentation,
  - Besoins identifiés,
  - Code source du projet,
  - Planning de développement,
  - Les plans et les jeux de test, etc.
- UML permet de réaliser la définition et la documentation de ces taches en fonction des contraintes.

Les domaines d'utilisation de UML

- UML est employé dans l'ensemble des secteurs du développement informatique et électronique :
  - Systèmes d'information,
  - Télécommunication, défense,
  - Transport, aéronautique, aérospatial,
  - Domaines scientifiques,
- Mais pas seulement : on peut modéliser des comportement mécaniques, humain, etc.
  - Structure d'un système de santé / judiciaire, etc.

Les briques de base d'UML

- Dans le langage UML, il existe 3 types de briques de base pour modéliser tous systèmes :

1. Les *éléments*

   ✓ Ce sont les abstractions essentielles au modèle.

2. Les *relations*

   ✓ Les relations expriment les liens existants entre les différents éléments.

3. Les *diagrammes*

   ✓ Les diagrammes comprennent des ensembles d'éléments dignes d'intérêt.

Il existe 4 types d'éléments dans UML

- Ce sont les éléments de base qui vont permettre la définition par la suite des modèles,
- Nous allons avoir différents types de modèles pour représenter les différentes caractéristiques du système,
  1. Les éléments *structurels,*
  2. Les éléments *comportementaux,*
  3. Les éléments de *regroupement,*
  4. Les éléments *d'annotation.*

Les éléments structurels

- Les éléments structurels sont les éléments les plus statiques d'UML, il en existe 7 types distincts :
  1. Les classes,
  2. Les interfaces,
  3. Les collaborations,
  4. Les cas d'utilisation,
  5. Les classes actives,
  6. Les composants,
  7. Les noeuds,
- A partir de là, on peut exprimer visuellement les relations structurelles de tous types de systèmes .

Les éléments comportementaux

- Les *éléments comportementaux* représentent les *parties dynamiques* des modèles,
- Ils permettent de définir le comportement du modèle dans le temps et l'espace,
- Il en existe 2 types fondamentaux de diagrammes pour exprimer les relations comportementales :
  1. Les interactions,
  2. Les automates à états finis.

Les éléments de regroupement

- Les *éléments de regroupement* représentent les *parties organisationnelles* des modèles UML,
- Ce sont les boites dans lesquelles un modèle peut être décomposé,
- Il existe un seul type d'élément de regroupement : "le package"

Les éléments d'annotation

- Les *modèles d'annotation* représentent les *parties explicatives* des modèles UML.
- Ce sont les commentaires qui vont être ajoutés aux modèles afin :
  - D'expliquer un choix de conception,
  - Décrire le fonctionnement du système ou d'une sous-partie,
  - Faire une remarque sur un élément du système,
  - Détailler une hypothèse d'utilisation,
  - Spécifier une contrainte d'implémentation (temps d'exécution maximum, consommation en énergie, etc.).

Les relations dans UML

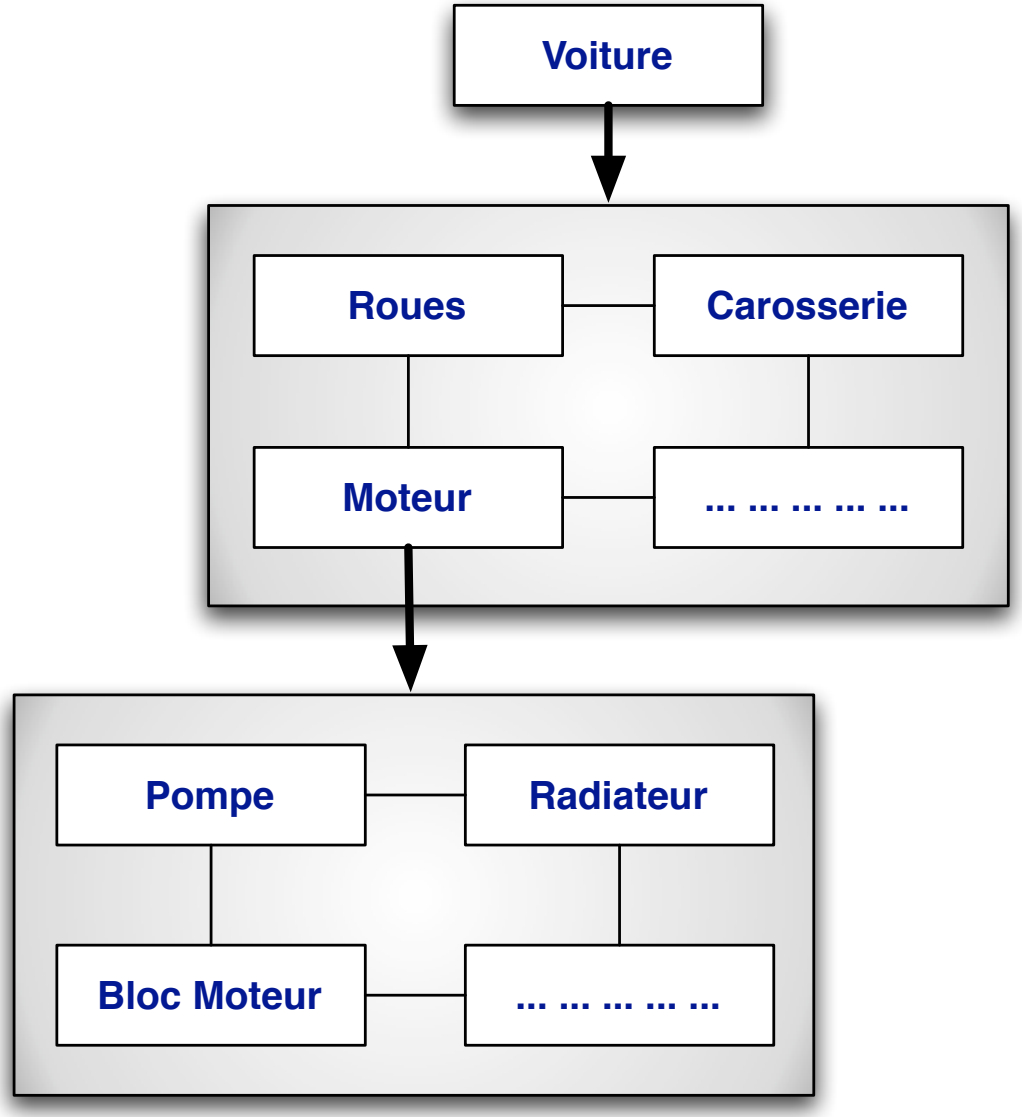
- Afin d'exprimer les liens interconnectant les différents éléments des modèles, *4 relations de base* ont été définies :

1. La *dépendance*,
2. La *généralisation*,
3. L'*association*,
4. La *réalisation*,

Les diagrammes en UML

- Un *système est un ensemble de sous-systèmes* organisés pour atteindre un objectif et décrit à l'aide de modèles.
- Un *sous-système est un regroupement d'éléments* , dont certains spécifient le comportement proposé par les autres éléments contenus.
- Un *modèle* est une *abstraction complète et cohérente d'un système* permettant d'en faciliter la compréhension.
  - Modélisation des *parties structurelles* (statique)
  - Modélisation du *comportement* (dynamique)

Exemple de décomposition hiérarchique



Les diagrammes avec UML

- Un *diagramme* est une *représentation visuelle* de l'ensemble des éléments qui constituent le système.
  - Ils servent à visualiser un système sous différents angles (utilisateur, administrateur par ex.)
- Dans les systèmes complexes, *un diagramme* ne fourni qu'une *vue partielle* du système.
  - L'ensemble des diagrammes aboutés permet d'obtenir une vue globale du système à concevoir.
  - Chaque diagramme va permettre de modéliser ou spécifier une vue (spécificité) du système à concevoir.

Les diagrammes dans UML : 9 types

- Les diagrammes *"statiques"*
  1. Diagramme de classes,
  2. Diagramme d'objets,
  3. Diagramme de composants,
  4. Diagramme de déploiement,
- Les diagrammes *"dynamiques"*
  5. Diagramme de cas d'utilisation,
  6. Diagramme de séquences,
  7. Diagramme de collaboration,
  8. Diagramme d'états-transitions,
  9. Diagramme d'activités.

Les diagrammes structurels

- Les *diagrammes de classes*
  - Ils représentent l'ensemble des classes, des interfaces et des collaborations ainsi que leurs relations.
  - Les diagrammes les plus utilisés dans une approche de développement objets.
- Les *diagrammes d'objets*
  - Ils représentent un ensemble d'objets (instances de classes) avec leurs relations,
  - Similaire aux diagrammes de classes, mais avec des détails réels et des prototypes.

Les diagrammes structurels

- Les *diagrammes de composants*
  - Ils sont apparentés aux diagrammes de classes.
  - Un composant correspond généralement à une ou plusieurs classe, interfaces ou collaboration.
- Les *diagrammes de déploiements*
  - Ils sont apparentés aux diagrammes de composants.
  - Un noeud englobe généralement un ou plusieurs composants.

Les diagrammes comportementaux

- Les *diagrammes de cas d'utilisation*
  - Organisent les comportements du système.
  - Représentation des acteurs et de leurs relations avec l'application.
- Les *diagrammes de séquences*
  - Centrés sur l'ordre chronologique des messages.
  - Il modélisent chronologiquement les messages échangés par les objets du système (sous-système).

✓ ***Ce sont les 2 modèles les plus employés (modélisation dynamique)***

Les diagrammes comportementaux

- Les *diagrammes de collaboration*
  - Centrés sur l'organisation structurelle des objets qui communiquent (envoie & réception des messages),
  - Mise en évidence des liens existants entre les objets,
- Les *diagrammes états-transitions*
  - Centré sur le changement d'état du système en fonction des événements,
  - Modélisation sous forme d'un automate à états-finis (état, action, transition),
- Les *diagrammes d'activités*
  - Centrés sur le flot de contrôle d'une activité sur une autre.

3

“ Diagramme de cas d’utilisation ”

ENSEIRB
ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Introduction aux cas d'utilisation

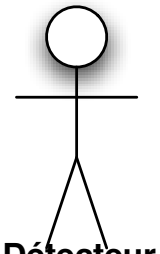
- Les cas d'utilisation servent à modéliser le *comportement dynamique* d'un système (sous système).
- Les comportements différents d'un système sont représenté sous forme de cas d'utilisation,
  - On ne se préoccupe pas de l'implémentation du système, juste des services qu'il doit rendre aux utilisateurs,
  - Données directement extraites du cahier des charges,
- Un cas d'utilisation décrit un *ensemble de séquences* dans lequel chaque séquence représente les interactions entre l'extérieur du système (acteurs) et le système lui même.

Introduction

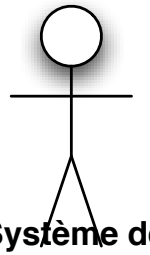
- L'expression des cas d'utilisation permet de déterminer les points suivants :
  - Quels sont les services que doit implémenter le système ?
  - A qui ces services doivent être rendus ?
- Les cas d'utilisation se composent :
  - *D'acteurs externes* au système (personnes, capteurs, autres applications ...),
  - *Du système* en lui même et des services qu'il doit rendre,
  - *Des relations* qui relient les acteurs aux services qui leurs sont rendus,

Cas d'utilisation, les acteurs

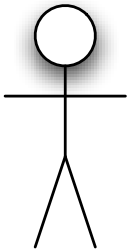
" Un acteur est une personne ou une chose qui va rentrer en interaction avec le système à concevoir ! "



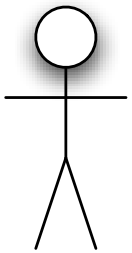
Détecteur de fumé



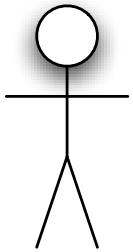
Système de gestion des badges



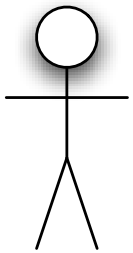
Secrétaire



Enseignant

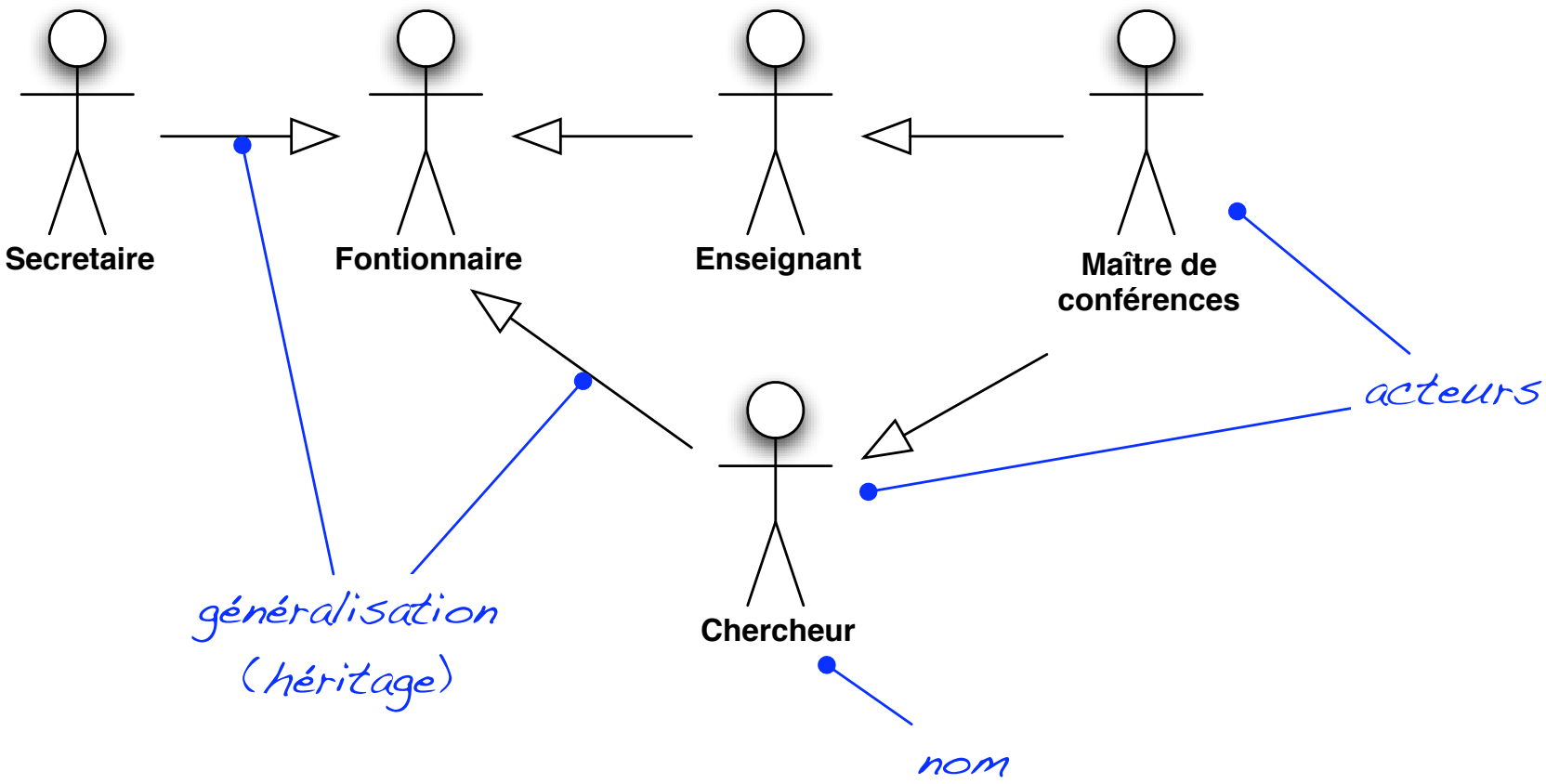


Etudiant



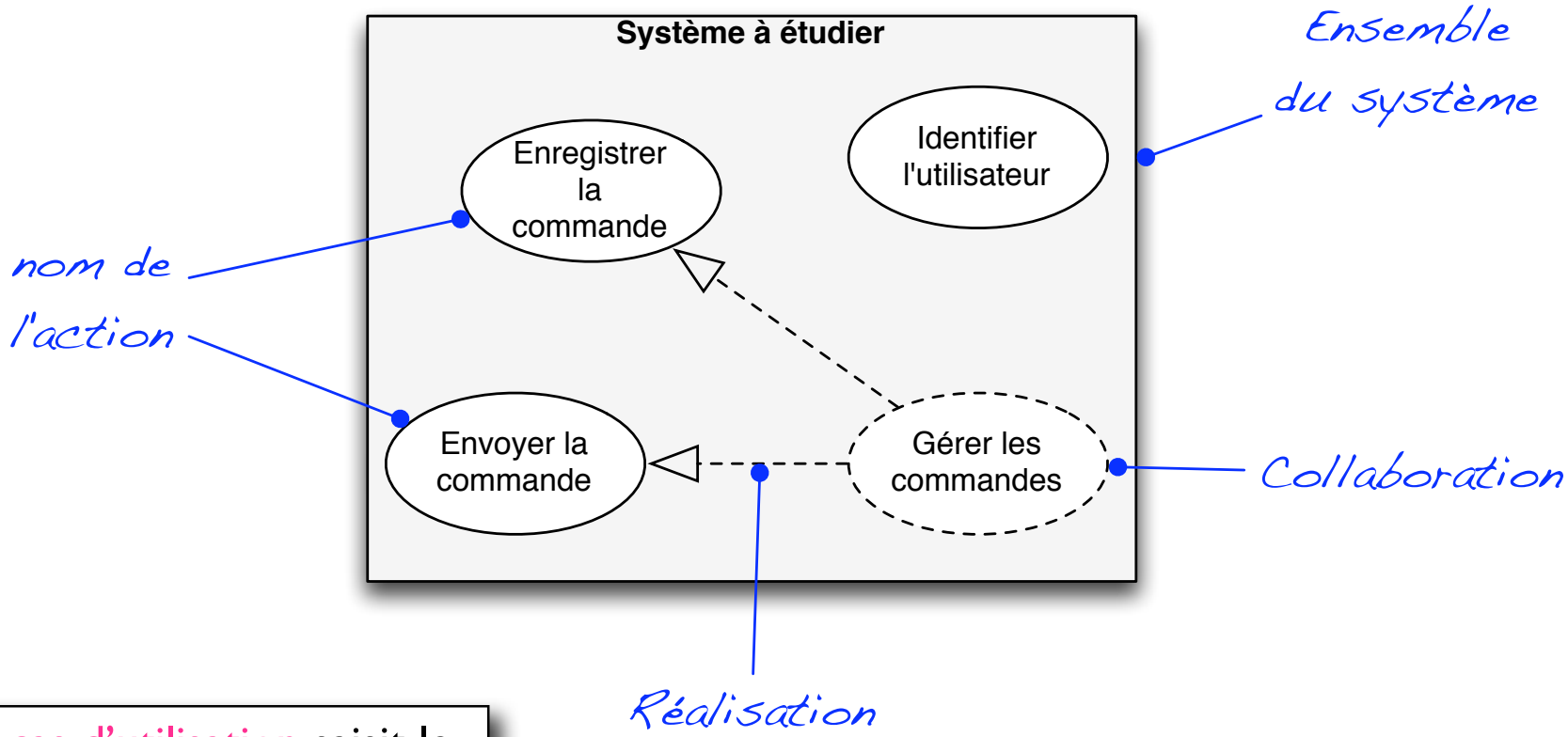
Administrateur

Cas d'utilisation, relations entre acteurs



Une secrétaire est une fonctionnaire, tout comme les enseignants et les chercheurs. Un maître de conférence est un fonctionnaire qui fait un travail d'enseignant et de chercheur à la fois.

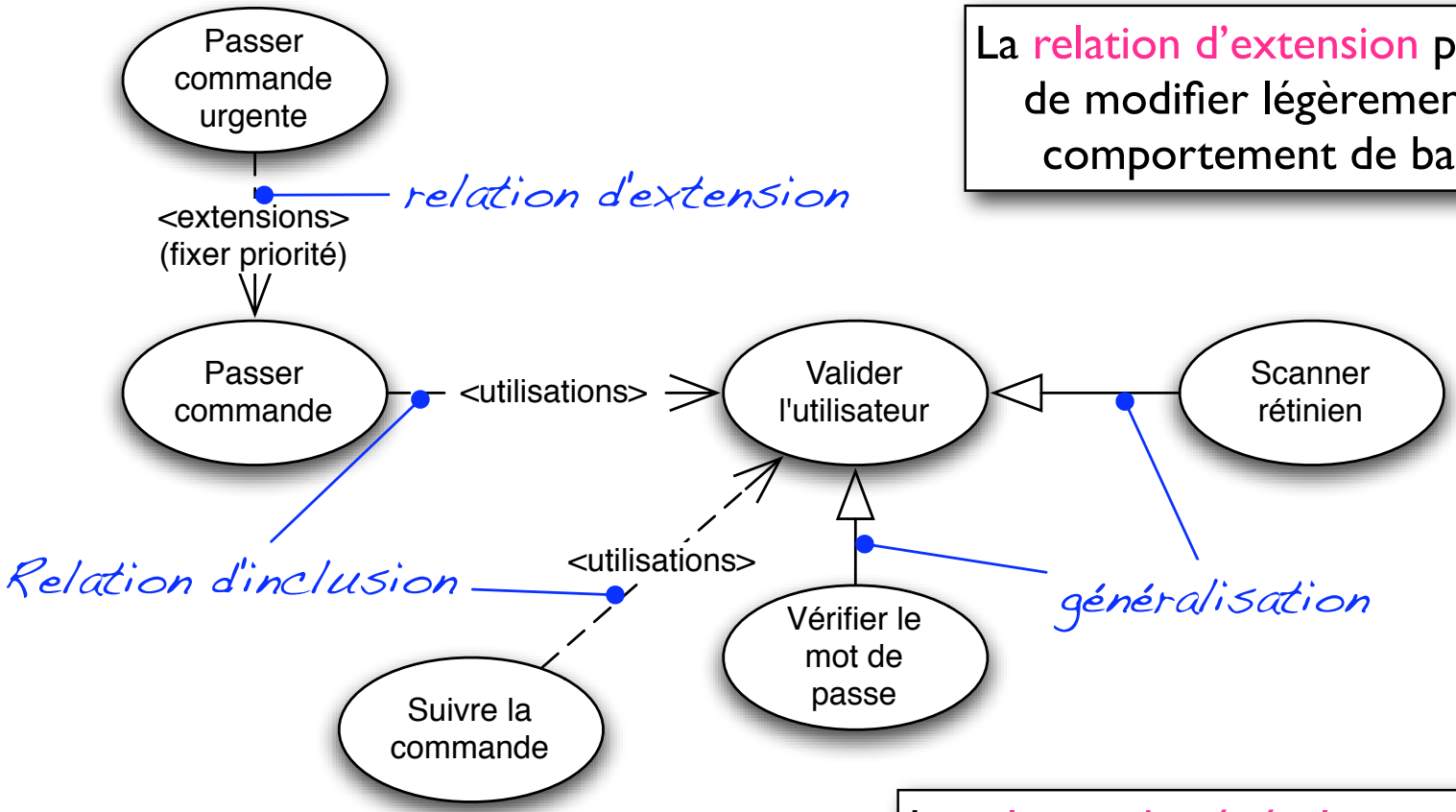
Cas d'utilisation, les actions



Un cas d'utilisation saisit le comportement attendu du système sans que l'on précise comment cela est réalisé.

La collaboration permet de raffiner la vue implémentation du système en factorisant les points communs.

Cas d'utilisation, Relations entre actions



La relation d'extension permet de modifier légèrement le comportement de base.

La relation d'inclusion est faite pour éviter la répétition des actions en factorisant les services rendus. Cela revient à déléguer une partie des actions.

La relation de généralisation modélise un héritage du comportement de base afin de l'affiner.

Afin de dessiner un cas d'utilisation...

- Identifier l'ensemble des acteurs possibles qui vont interagir avec l'application,
  - Identifier tous les acteurs,
  - Les factoriser sous forme de groupes,
- Prendre en considération les besoins de tous les utilisateurs
  - Identifier tous les services que doit rendre le système,
  - Le regrouper les besoins communs à l'aide des relations d'inclusion, de généralisation et d'extension,

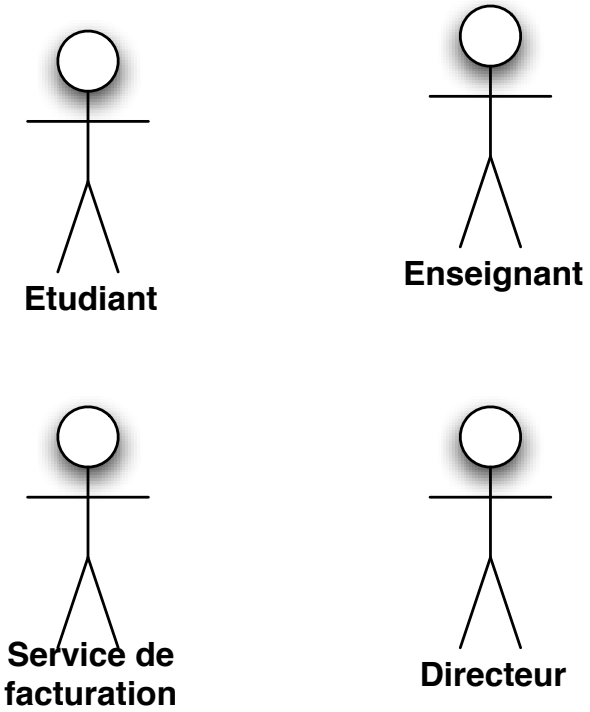
Exemple pédagogique : l'ENSEIRB

- L'ENSEIRB désire automatiser le processus d'inscription des étudiants, en voici le cahier des charges :
  - Le *responsable des filières* établit au début de chaque semestre le *programme des cours*
  - *Chaque étudiant* doit sélectionner *4 cours obligatoires* et *2 cours optionnels* par semestre.
  - Lorsqu'un étudiant s'inscrit, le *système de facturation* est averti et génère immédiatement la facture.
  - *Les enseignants* utilisent ces informations pour consulter leur service.
  - *Les étudiants* peuvent consulter *leur emploi* du temps en fonction des cours qu'ils ont choisis.

Exemple, Identification des acteurs

- 4 acteurs existent dans notre cahier des charges :
  1. L'étudiant,
  2. L'enseignant,
  3. Le système de facturation,
  4. Le directeur des filières,

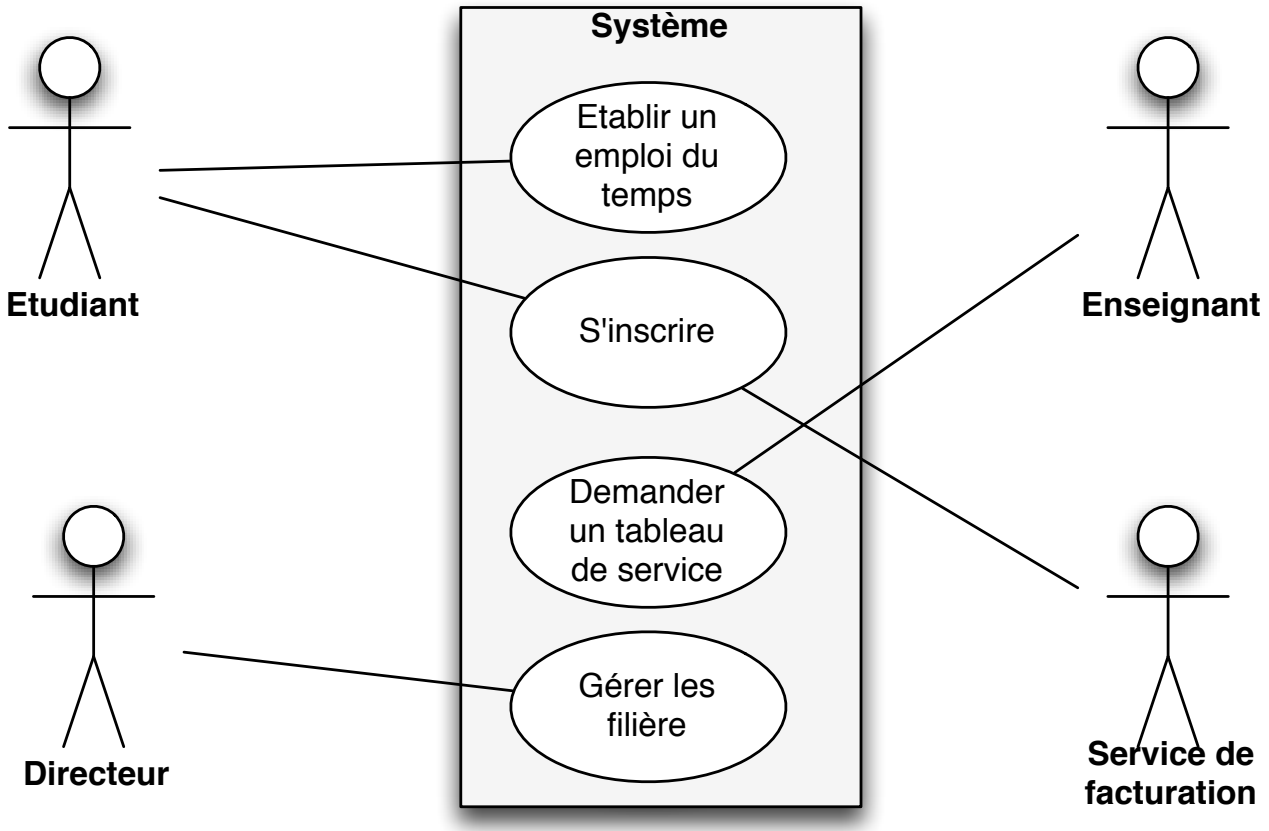
- Ce sont les seules personnes qui interagissent avec le système (selon le cahier des charges).



Exemple, Identification des fonctions

- Un cas d'utilisation est un motif de comportement intrinsèque au système,
  - Chaque cas d'utilisation est une séquence de transactions connectées, effectuées par un dialogue entre un acteur et le système.
- Identification des besoins propres au acteurs
  - *Chef de service* - maintenir le programme des études.
  - *Enseignant* - demander un tableau de service.
  - *Etudiant* - établir un emploi du temps.
  - *Système de facturation* - recevoir les informations de facturation du système d'inscription.

Les cas d'utilisations, solution



Bien identifier les cas d'utilisation

- Définition d'un cas d'utilisation
  - Ensemble d'actions produisant un résultat observable pour un acteur particulier,
- Vérifier que chaque cas d'utilisation candidat,
  - Fournit une valeur ajoutée à la description,
  - Qu'il existe un événement externe le produisant,
  - On peut le décrire succinctement pour l'expliquer,
- Attention au niveau de granularité du modèle !
  - Ne pas descendre trop bas,
  - Limiter le nombre de niveau (<20)

Documenter les cas d'utilisation

- Pour chaque cas d'utilisation, il faut :
  - Exprimer ce que fourni l'utilisateur au système,
  - Ce que l'utilisateur reçoit en retour après exécution,
  - Il faut aussi penser à détailler les hypothèses réalisées lors de la description du cas d'utilisation (pré-requis et les post-conditions)
- L'expression des transactions entre le système et l'utilisateur sera détaillé par d'autres modèles plus adaptés :
  - Diagrammes de séquences,
  - Diagrammes d'activité, etc.

Scénario d'un cas d'utilisation

- Demande d'un tableau de service (enseignant)

1. L'enseignant se présente devant un terminal,
2. Le système affiche un message d'accueil,
3. L'enseignant demande un relevé d'heures,
4. Le système lui demande de s'authentifier,
5. L'enseignant s'authentifie (login et mot de passe),
6. Le système affiche les informations à l'écran,
7. L'enseignant précise qu'il a terminé sa consultation et qu'il désire se déconnecter,
8. Le système effectue la requête et retourne à l'écran d'accueil.

Description simplifiée

Exemple de scénario d'un cas d'utilisation

- Demande d'un tableau de service (enseignant)
  - ✦ Pré-conditions
    (avant que l'opération ne puisse avoir lieu)
    - ➡ L'enseignant doit être inscrit à l'université,
    - ➡ L'enseignant doit connaître ses identifiants,
  - ✦ Post-condition
    (si l'opération s'est bien déroulée)
    - ➡ L'enseignant a pris connaissance des heures de cours qu'il a données.

4

“ Diagrammes de classes ”

ENSEIRB
ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Les diagrammes de classes

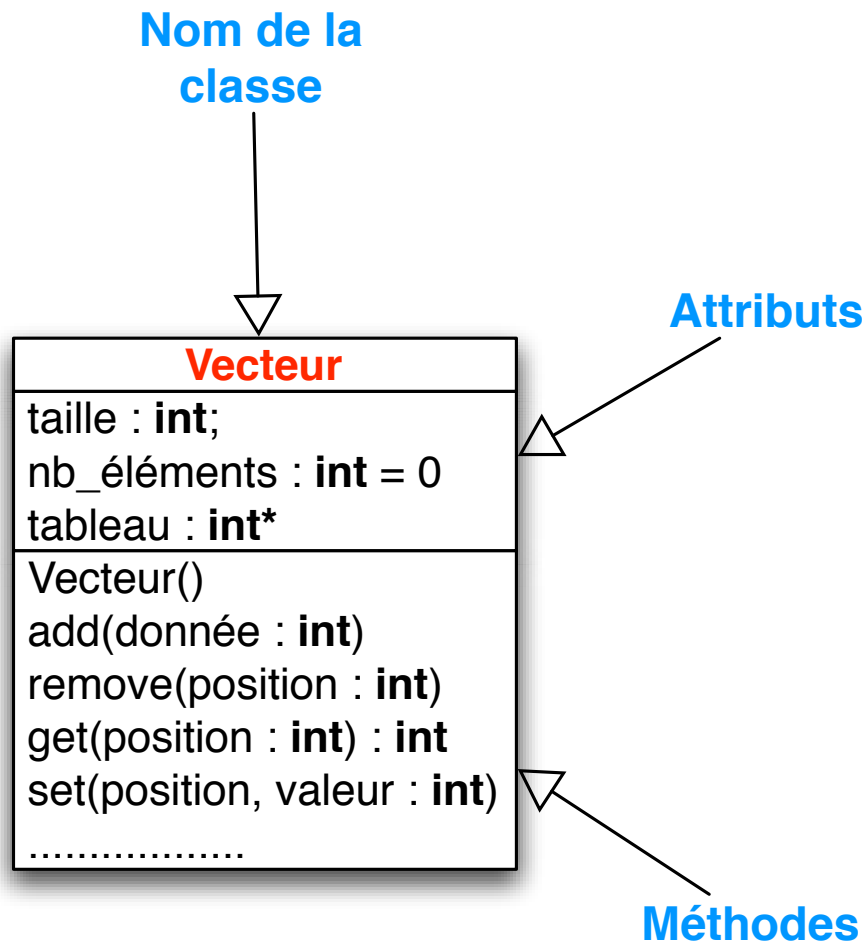
- Un diagramme de classe se compose des éléments suivants,
  - Des classes,
  - Des interfaces (classes virtuelles),
  - Des collaborations,
  - Des relations (dépendances, associations, généralisation),
- Le diagramme de classes peut être défini à différents niveaux d'abstraction
  - Point de *vue conceptuel* (entités fournissant des services),
  - Point de *vue des spécifications* (raffinement fonctionnel des entités),
  - Point de *vue implémentation* (les classes que l'on doit mettre en oeuvre).

Définition d'une classe

- Une classe est la description élémentaire d'un ensemble d'objets qui partagent des attributs et des méthodes en communs.

- La définition des attributs et des méthodes n'est *pas obligatoire* :
  - Les attributs et les méthodes sont trop nombreux pour être représentés dans la majorité des cas...
  - On indique alors les responsabilités de la classe sous forme de texte (ce qu'elle doit implémenter comme service).
  - On peut dès la définition d'une classe préciser les droits d'accès aux méthodes et aux attributs (+, -, #)

Exemple de modélisation d'une classe



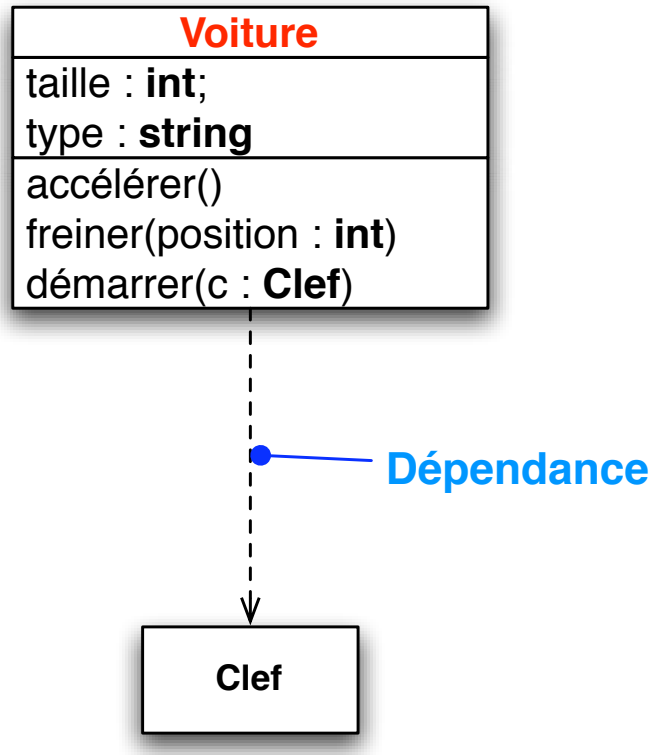
Introduction aux relations

- En UML, la manière dont les éléments sont interconnectés entre eux (logiquement ou structurellement) est modélisé sous forme de relation,
  - Les roues, le moteur, les portes, la carrosserie font partie du vocabulaire de construction d'une voiture mais ils n'existent pas de manière indépendantes.
- Il existe différents types de relation entre les éléments :
  - La dépendance,
  - L'utilisation,
  - La généralisation,
  - L'association,
  - L'agrégation.

Les relations : la dépendance

- La *dépendance* établit une *relation d'utilisation* entre 2 entités d'un même diagramme.
- La plus part du temps il s'agit d'une *dépendance d'utilisation*,
  - Argument d'une méthode par exemple,
  - On parle alors de *"relation d'utilisation"*,
- Cela permet d'identifier implications possibles des modifications à apporter dans une entité
  - Changement du comportement d'une des classe par exemple.

Les relations : la dépendance

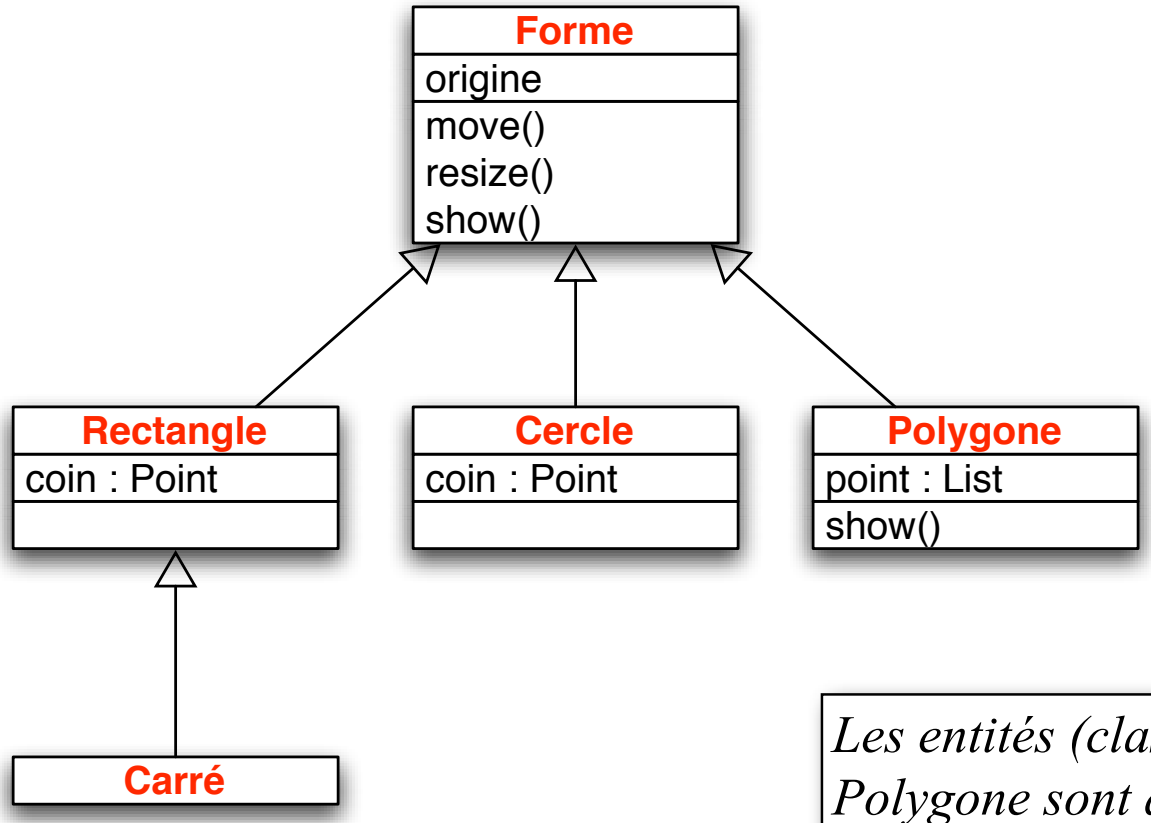


La classe "Voiture" dépend pour son utilisation de la classe "Clef"

Les relations : la généralisation

- La *généralisation* modélise la *notion d'héritage* qui existe dans les langages objets
  - La généralisation correspond à la notion *"est une sorte de"*,
  - Modélisation des relations parents / enfants,
- Cela implique que les entités issues d'une généralisation sont utilisables partout ou leur classe mère peut l'être (mais pas l'inverse)
  - Généralisation (classe, classe) ou (classe, interface),
- Cette relation est modélisée par une *flèche pointant sur la classe mère*,

Les relations : la généralisation

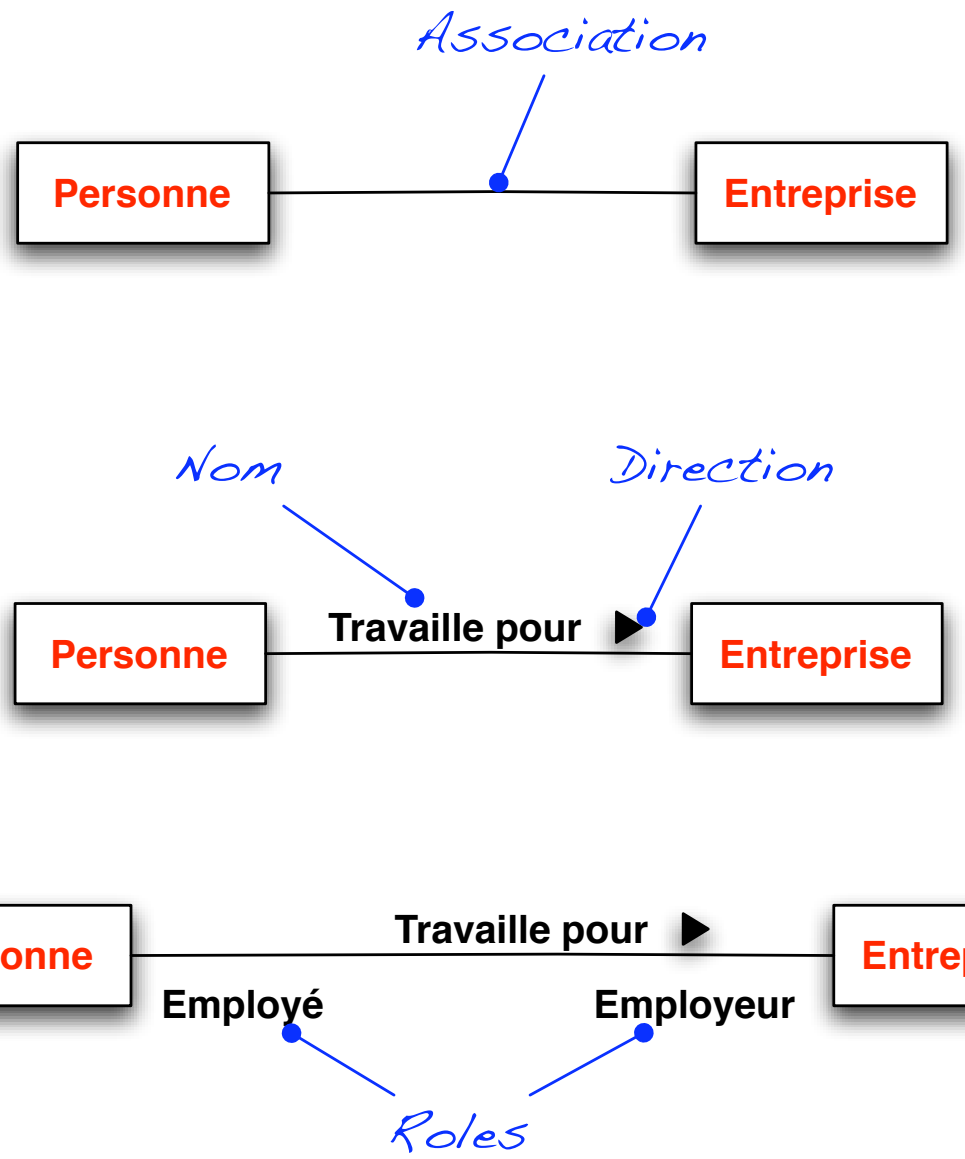


Les entités (classes) Rectangle, Cercle et Polygone sont des classe filles qui héritent de la classe mère Forme. De la même manière, l'entité Carré hérite de l'entité Rectangle.

Les relations : l'association

- Les *relations d'association* permettent de modéliser un *lien structurel* entre 2 éléments du modèle.
  - Si le lien joint 2 classes alors cela signifie qu'il existe une relation de navigabilité entre ces 2 classes,
- 4 décorations permettent de spécifier plus finement le lien entre les objets :
  - *Le nom* : décrire la nature de la relation entre les objets,
  - *La direction* : lever l'ambiguïté sur le nom,
  - *Les rôles* : indiquer le rôle spécifique de chacune des classes dans l'association,
  - *La cardinalité* : spécifie le nombre d'éléments affectés.

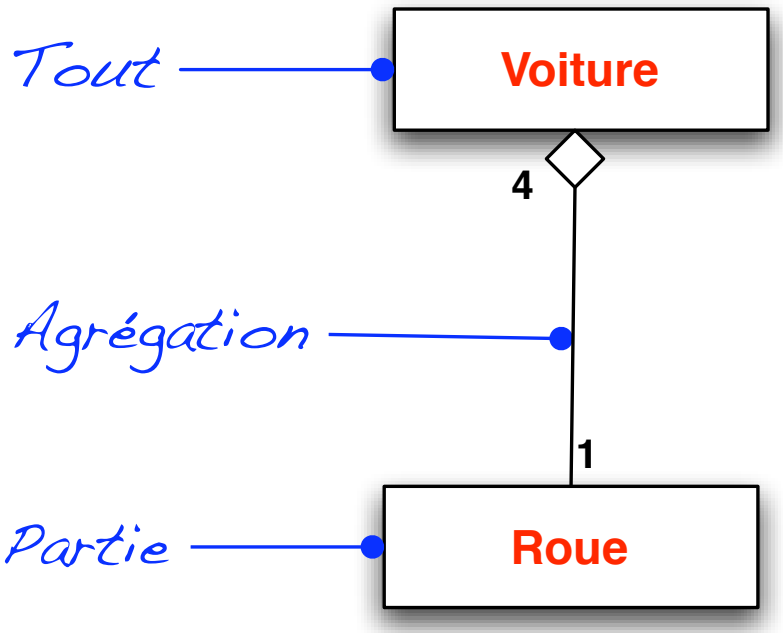
Les relations : l'association



Les relations : l'agrégation

- Dans l'association, les entités se trouvent au même niveau hiérarchique.
- Dans l'agrégation, il existe une relation hiérarchique entre les entités,
- *L'agrégation* répond à la question *"se compose de"* et modélise une *notion de possession,*
  - Notion de *"tout"* et de *"parties"*,
- Se note comme une association avec *un losange du coté du tout*.

Les relations : l'agrégation



Conclusion sur les relations

- Les relations doivent être spécifiées pour toutes les classes qui interagissent entre elles,
- Les relations sont des *liens statiques* entre les différentes entités (classes),
  - Les relations apparaissent quasi-exclusivement dans les diagrammes de classes,
- Il existe 2 autres types de relations utilisés dans la modélisation dynamique :
  - Les *"liens"* : flux permettant le transfert de messages entre objets.
  - Les *"transitions"* : utilisés dans la modélisation des machines à états finis.

Spécification des commentaires (Notes)

- Lors des phases de modélisation et de spécification, il est nécessaire de *documenter* ses modèles :
  - Contraintes *matérielles*,
  - Contraintes de *performance*,
  - *Choix techniques* réalisés,
  - *Références* à d'autres documents,
  - Explications techniques, etc.
- Dans le langage UML, il existe une sémantique pour les commentaires.

Modélisation des commentaires

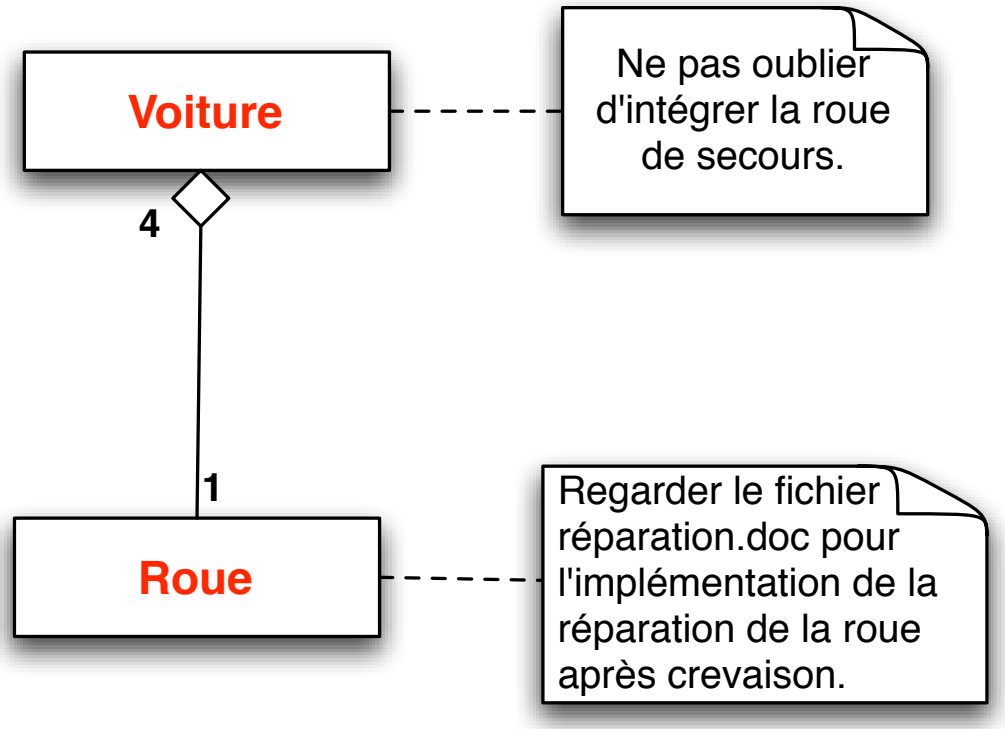


Diagramme de classes, exemple

- Nous allons maintenant essayer de modéliser une entreprise à l'aide d'un diagramme de classes :
  - Une *entreprise* est *composée* de *plusieurs services*, dans ces *services officie* un *certain nombre de personnes*.
  - Dans les locaux de l'entreprise se *trouvent* des *bureaux utilisés* par les *employés*, chaque bureau *porte* un *numéro* et *possède* un *numéro de téléphone*.
  - Les *employés sont reconnus* à l'aide de leur *nom* et de leur *numéro personnel*. Chaque *employé possède* un *statut* dans l'entreprise.

Diagramme de classes - Exemple (1)

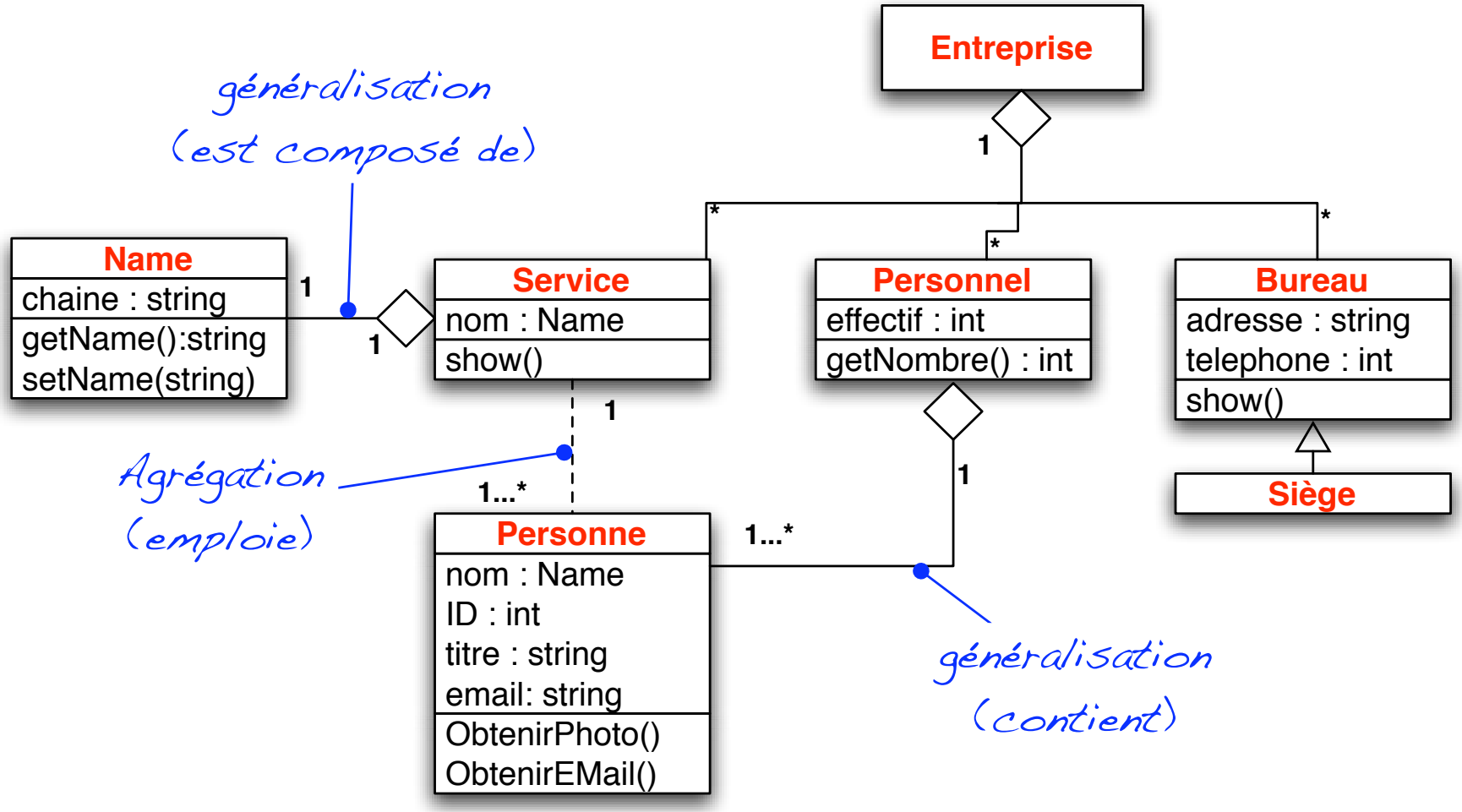


Diagramme de classes - Exemple (2)

- Nous allons maintenant nous attacher à essayer de modéliser une université :
  - L'*université* est *composée* d'*étudiants* qui *suivent* des *cours* dans les *filières* de leur choix.
  - Ces *cours* sont *dispensés* par des *enseignants* qui peuvent être *responsable* des filières.
  - Les *cours* sont *dispensés* dans *au moins* une *filière* et peuvent être *mutualisés* avec d'autres.
- A partir de là, réalisez le diagrammes de classes modélisant l'université.

Correction de l'exemple (1)

Université

Filière

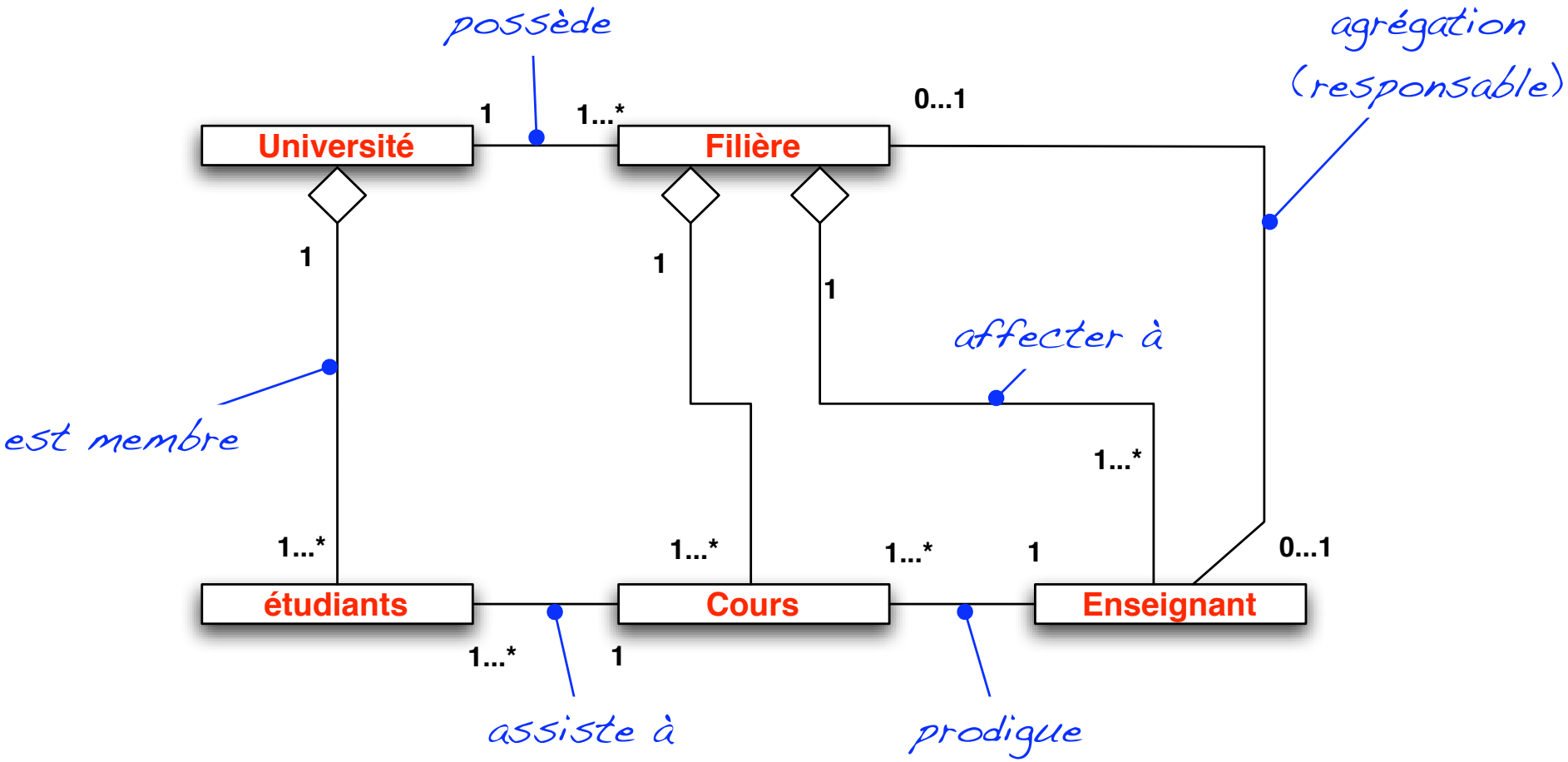
étudiants

Cours

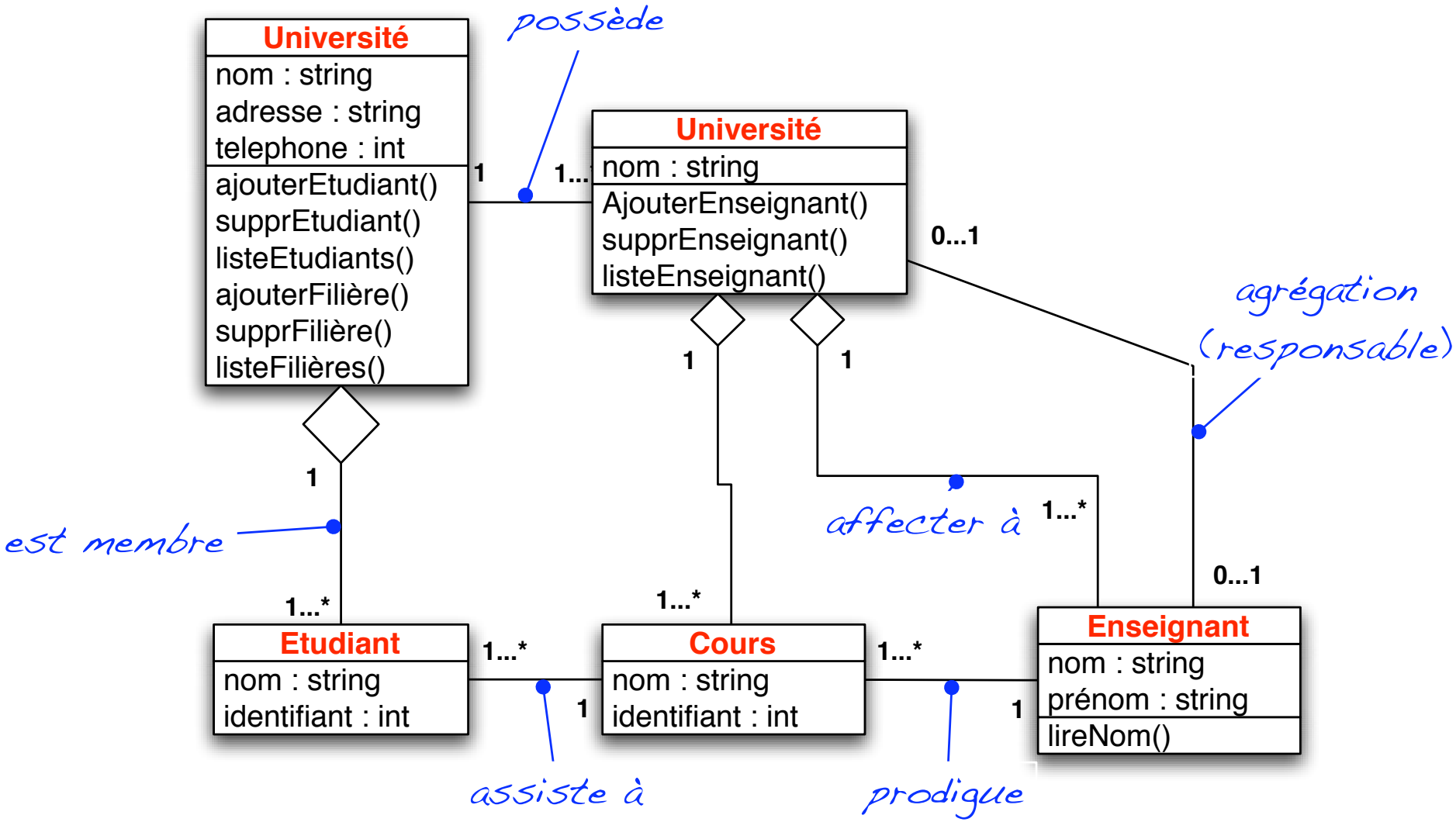
Enseignant

Voici les 5 entités de base qui constituent l'université que nous devons modéliser

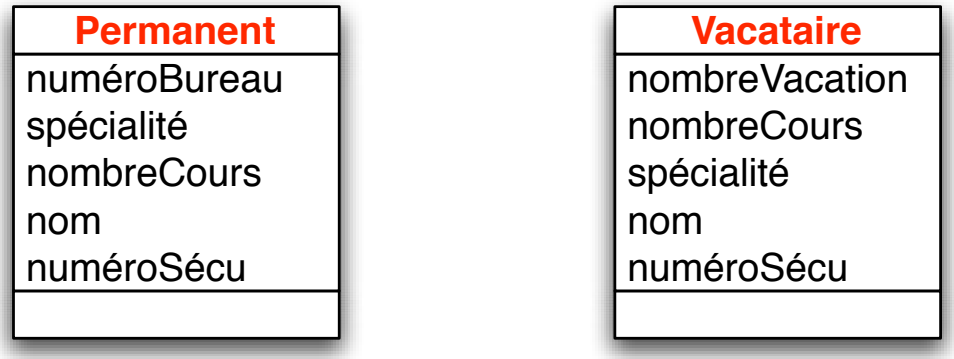
Correction de l'exemple (2)



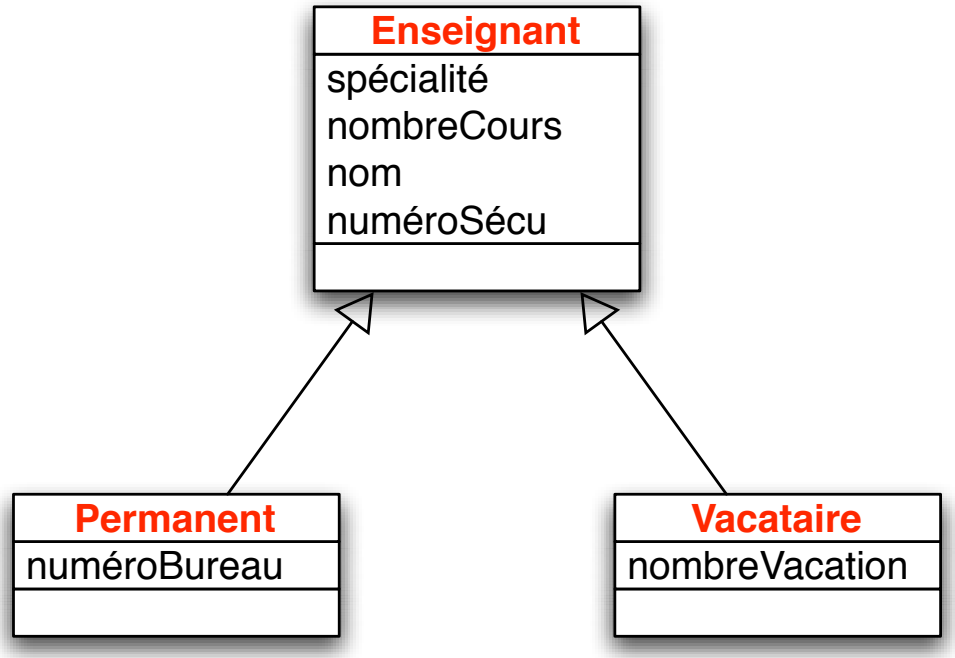
Correction de l'exemple (3)



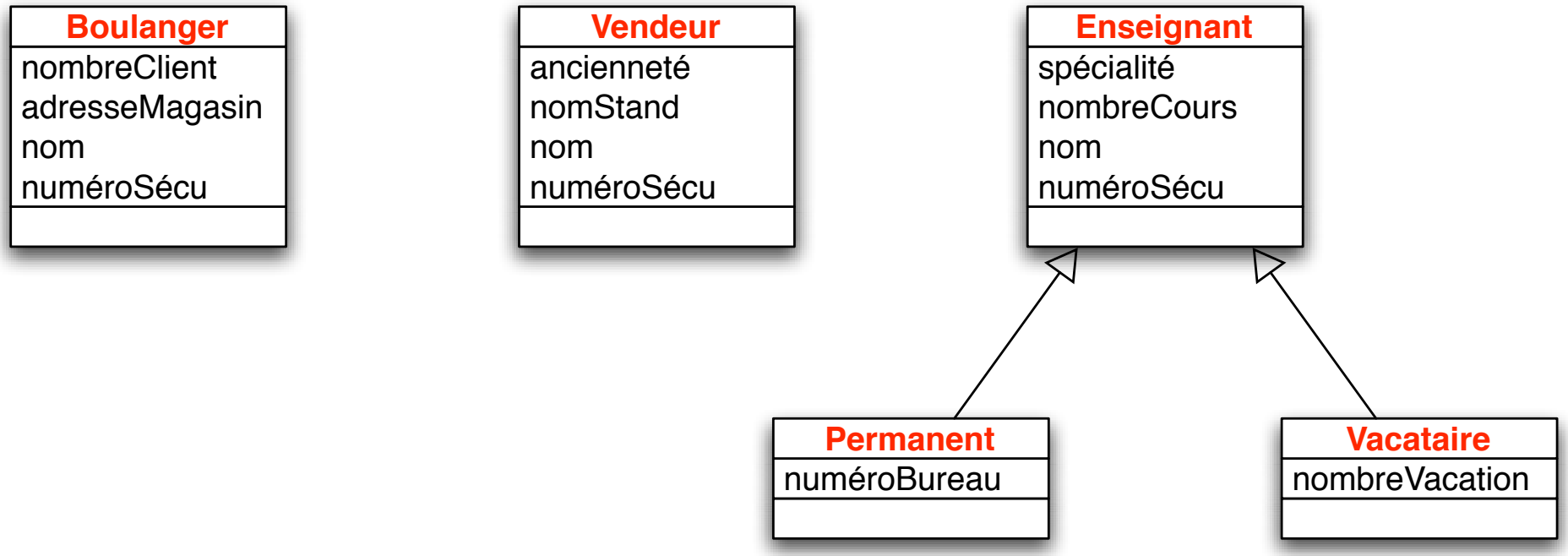
Exemple de travail itératif sur le modèle (1)



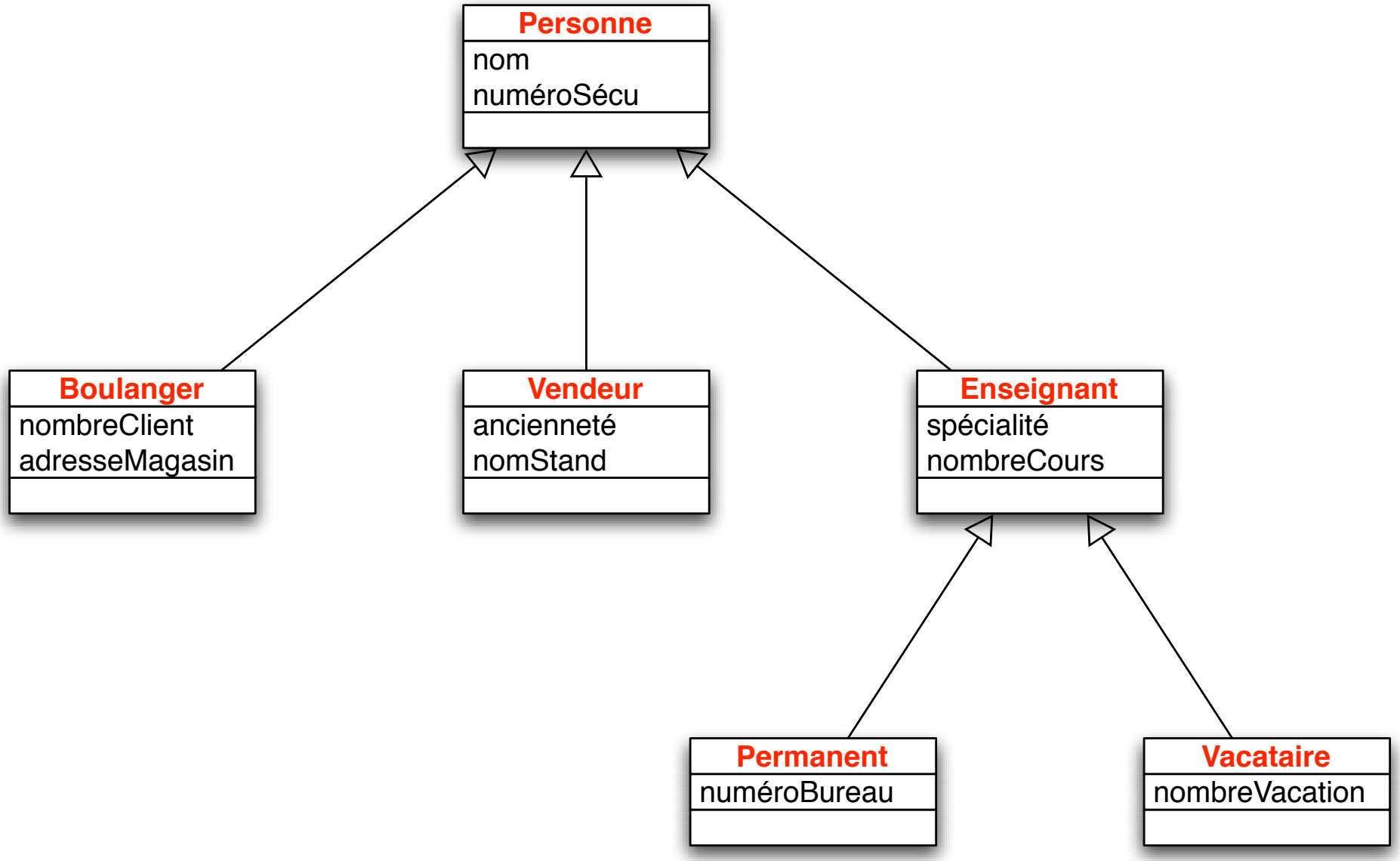
Exemple de travail itératif sur le modèle (2)



Exemple de travail itératif sur le modèle (3)



Exemple de travail itératif sur le modèle (4)



5

“ Implémentation des diagrammes de classes ”

ENSEIRB
ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

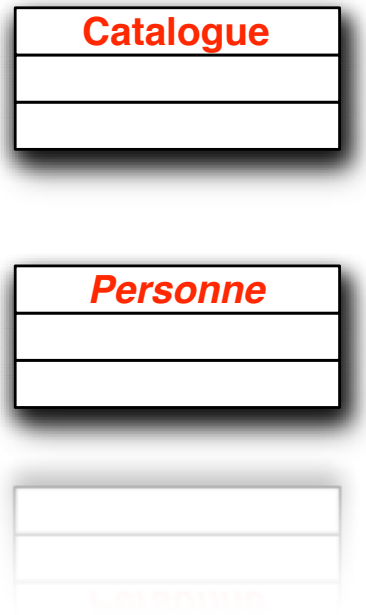
Et après les modèles que fait'on ?

- Les diagrammes permettent de spécifier de manière claire, les classes et les relations existantes entre elles.
  - Génération de code source à partir du modèle,
  - Les modèles ne sont pas spécifiques à un langage (choix à la génération => 1 modèle plusieurs implémentations possibles suivant les contraintes),
- A partir de tout diagramme UML, il est possible de remonter vers le cahier des charges
  - Les éléments du modèle ont une sémantique propre et unique.

Et après les modèles que fait'on ?

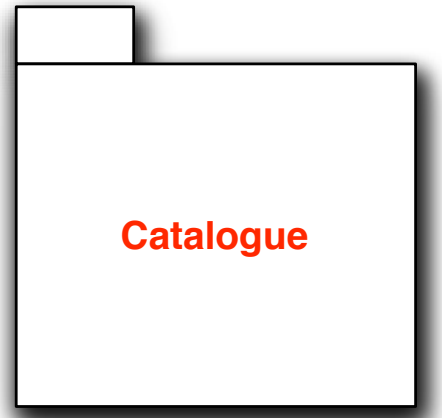
- Une fois que l'on a réalisé l'ensemble des diagrammes de classe, on va traduire les modèles dans un langage objet :
  - Indépendance du modèle vis-à-vis du langage utilisé pour l'implémentation.
    - ➡ Nous utiliserons ici du C++
- Chaque type d'entité et de relation possède une implémentation particulière,
  - Pas d'ambiguïté possible.

Transformation du modèle au code source



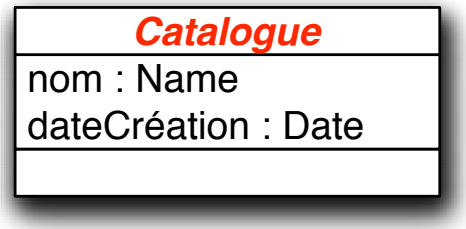

```
class Catalogue{
      // ... ... ...
};


class Personne {
      // ... ... ...
};
```

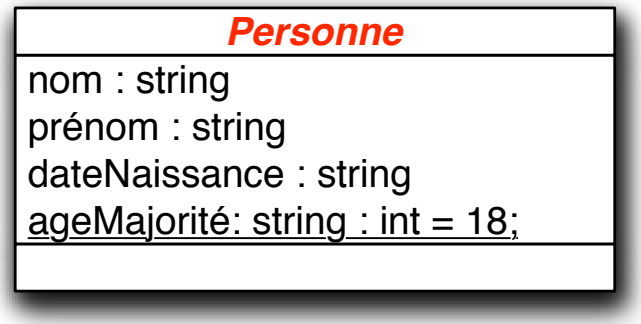
Transformation du modèle au code source




```
namespace Catalogue
{
    // ... ... ...
};
```

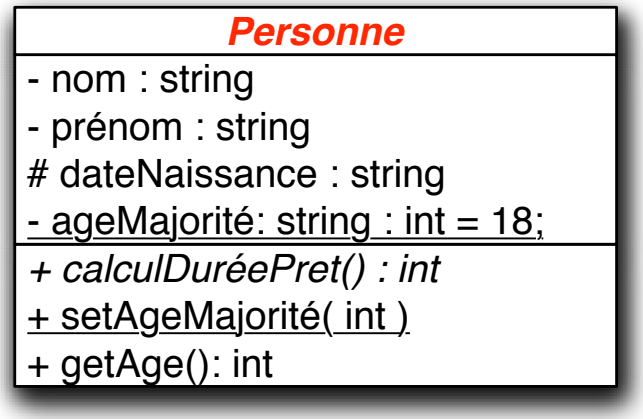
Transformation du modèle au code source





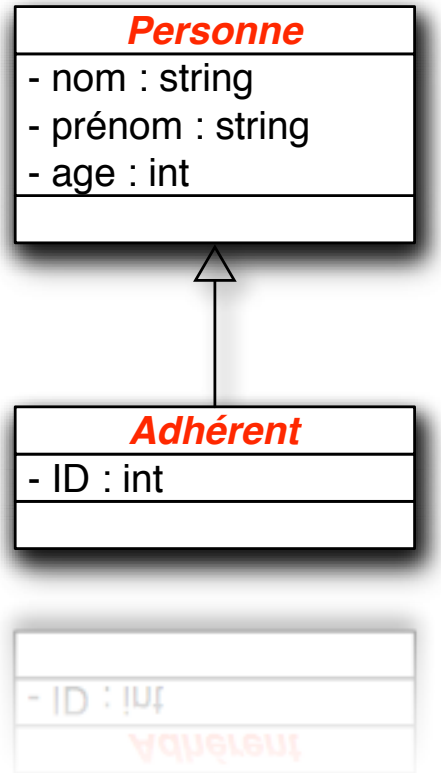

```
class Catalogue {
private:
     string nom;
      DateTime dateCreation;
      // ... ... ...
}

class Personne {
private:
   string nom;
   string prénom;
protected:
      DateTime dateNaissance;
private:
     static int ageMajorite = 18;
}
```

Transformation du modèle au code source



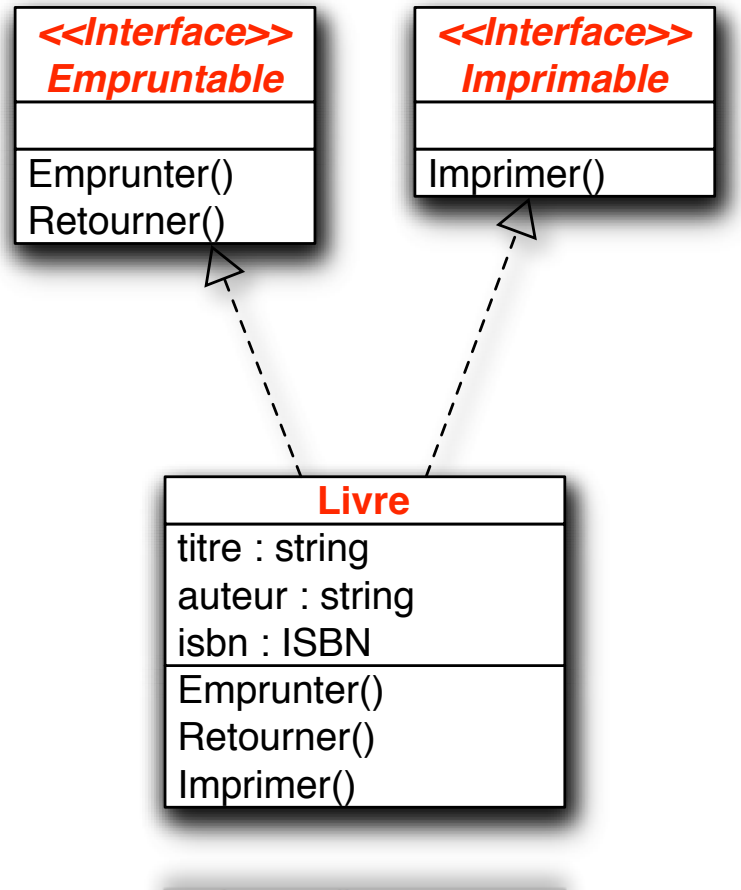

```
class Personne {
private:
     string nom;
     string prénom;
     static int ageMajorite = 18;
protected:
    DateTime dateNaissance;
public:
    int CalculerDureePret();
    static void SetAgeMajorite(int
                                aMaj) {
         // ... ... ...
    }
    public int GetAge() {
         // ... ... ...
    }
}
```

Implémentation d'une généralisation (héritage)



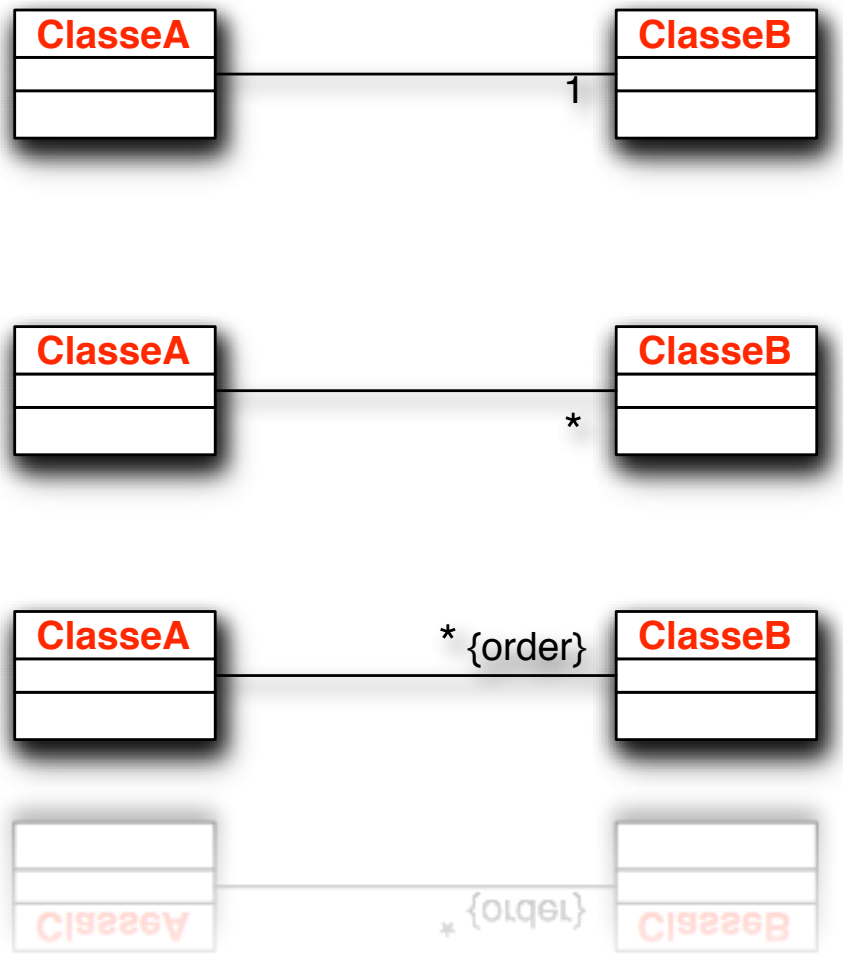

```
class Personne {
   // ... ... ...
}

class Adhèrent : Personne {
    private int iD;
}
```

Implémentation d'un héritage multiple



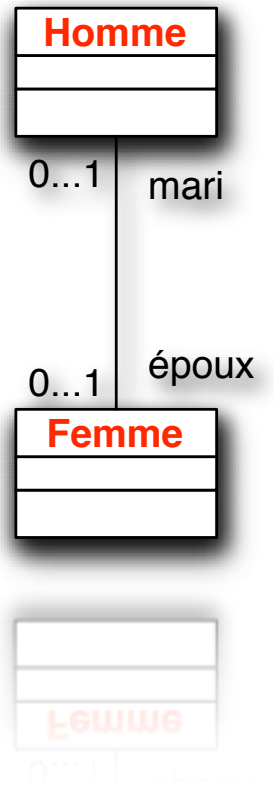

```
class Livre : public Imprimable :
              public Empruntable {
private:
     string titre;
    string auteur;
    ISBN isbn;
public:
     void Imprimer(){
          // ... ... ...
     }
    void Emprunter(){
          // ... ... ...
     }
    void Retourner(){
          // ... ... ...
     }
}
```

Implémentation de la composition (agrégation)



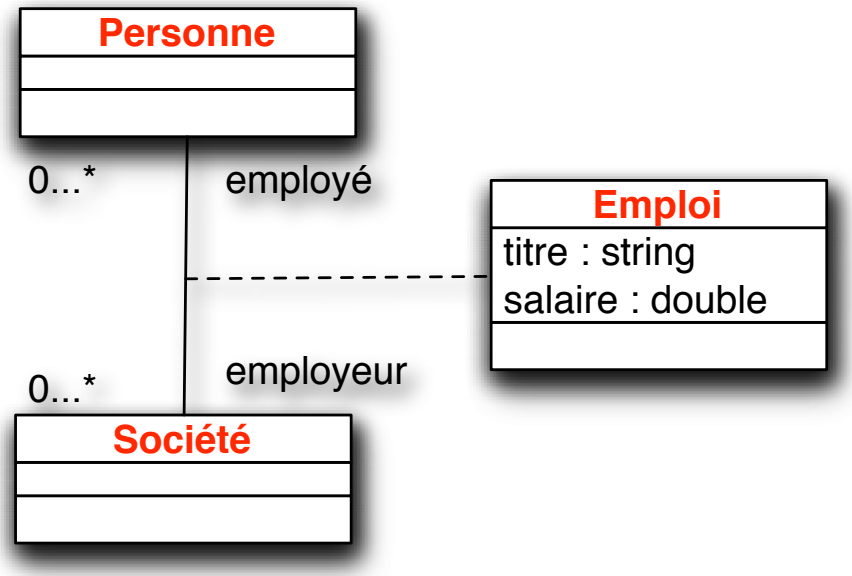

```
public class A1 {
private:
   ClasseB leB;
   // ... ... ...
}

public class A2 {
private:
   ClasseB[] lesB;
   // ... ... ...
}

public class A3 {
private:
   ArrayList lesB;
     // ... ... ...
   lesB = = new ArrayList();
}
```

Implémentation de la composition (agrégation)




```
class Homme {
private:
      Femme épouse;
      // ... ... ...
}

class Femme {
private:
      Homme mari;
      // ... ... ...
}
```

Implémentation d'une association entre classes




```
class Emploi {
private:
    string titre;
    double salaire;
    Personne employé;
    Société employeur ;
     // ... ... ...
}
```

6

“ Les diagrammes de séquence et de collaboration ”

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Les diagrammes de séquence

- Mise en évidence de l'*aspect temporel* des traitements (ordre chronologique de réalisation),
- Mise en évidence des *objets et des messages échangés* par les entités interagissant avec et/ou dans le système,
- La modélisation est basée sur l'affichage des objets participant à la séquence et à l'ordre des messages et des actions associées.

Définition de la sémantique du diagramme

- Les *diagrammes de séquence* permettent de modéliser les interactions entre les objets :
  - Communications *synchrones* (flèches pleines),
  - Communications *asynchrones* (flèches pointillés),
  - Les données transmises entre les objets (annotation des flèches),
- Ils permettent aussi de connaître pour chaque objet :
  - *Ses dates* de création et de destruction,
  - Sa *durée de vie*, si la création est externe au diagramme (ligne verticale en pointillés).

Diagramme de séquence - Ex. de l'ascenseur

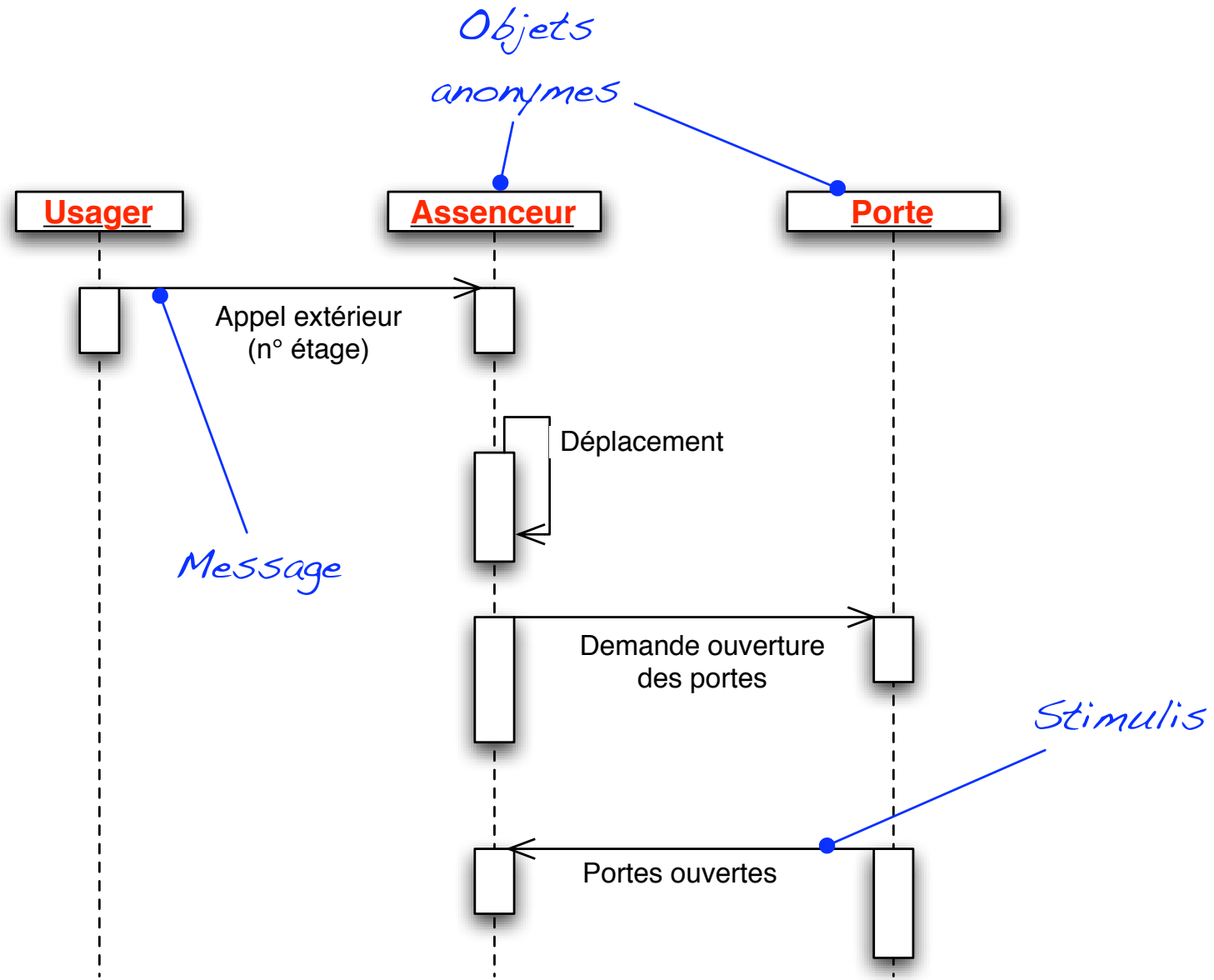
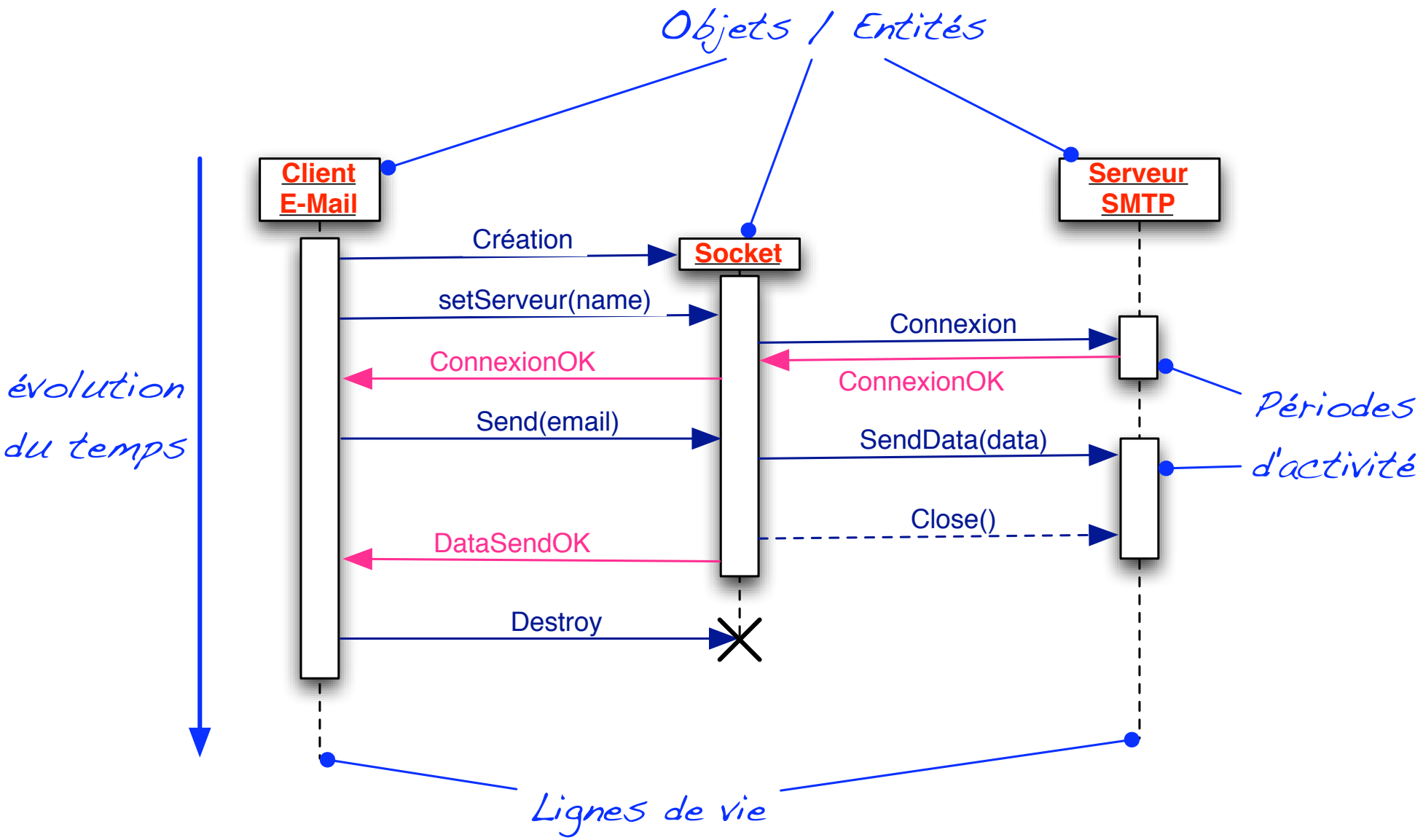


Diagramme de séquence - Ex. de client e-mail



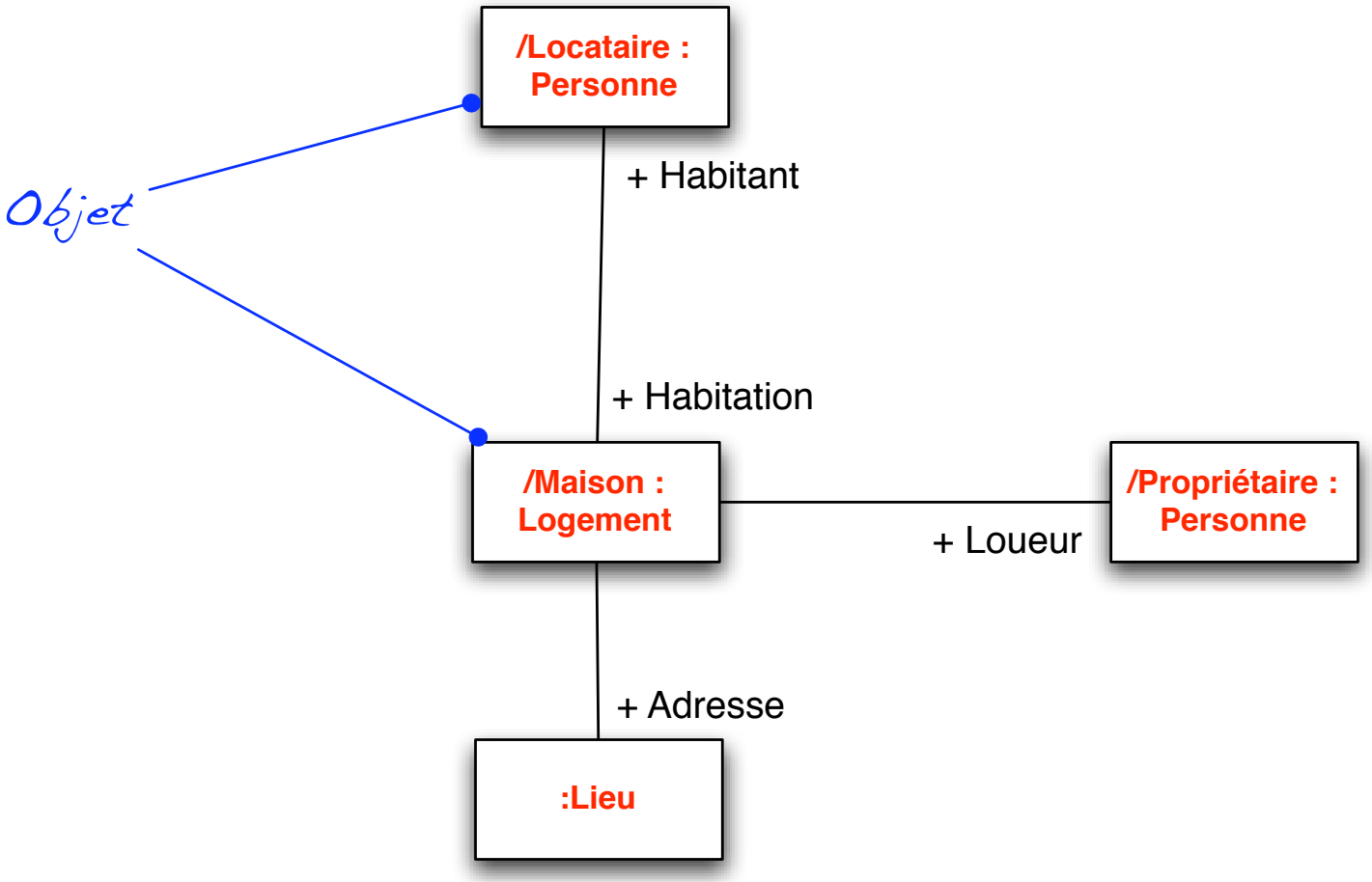
Le diagramme de collaboration

- Mise en évidence de *l'organisation des objets qui vont collaborer* pour effectuer une interaction.
  - On représente les objets qui vont intervenir (les sommets),
  - On modélise les liens qui vont modéliser les communications entre les objets (les arcs),
  - On annote les liens à l'aide des informations qui vont être échangées entre les entités,
- Visualisation claire du flot de contrôle dans le contexte de l'organisation structurelle.

Le diagramme de collaboration

- 2 niveaux de représentation (d'abstraction)
  - Le niveau spécification
    - Définition des classes et de leurs rôles,
  - Le niveau instance
    - Définition des objets et des messages échangés,

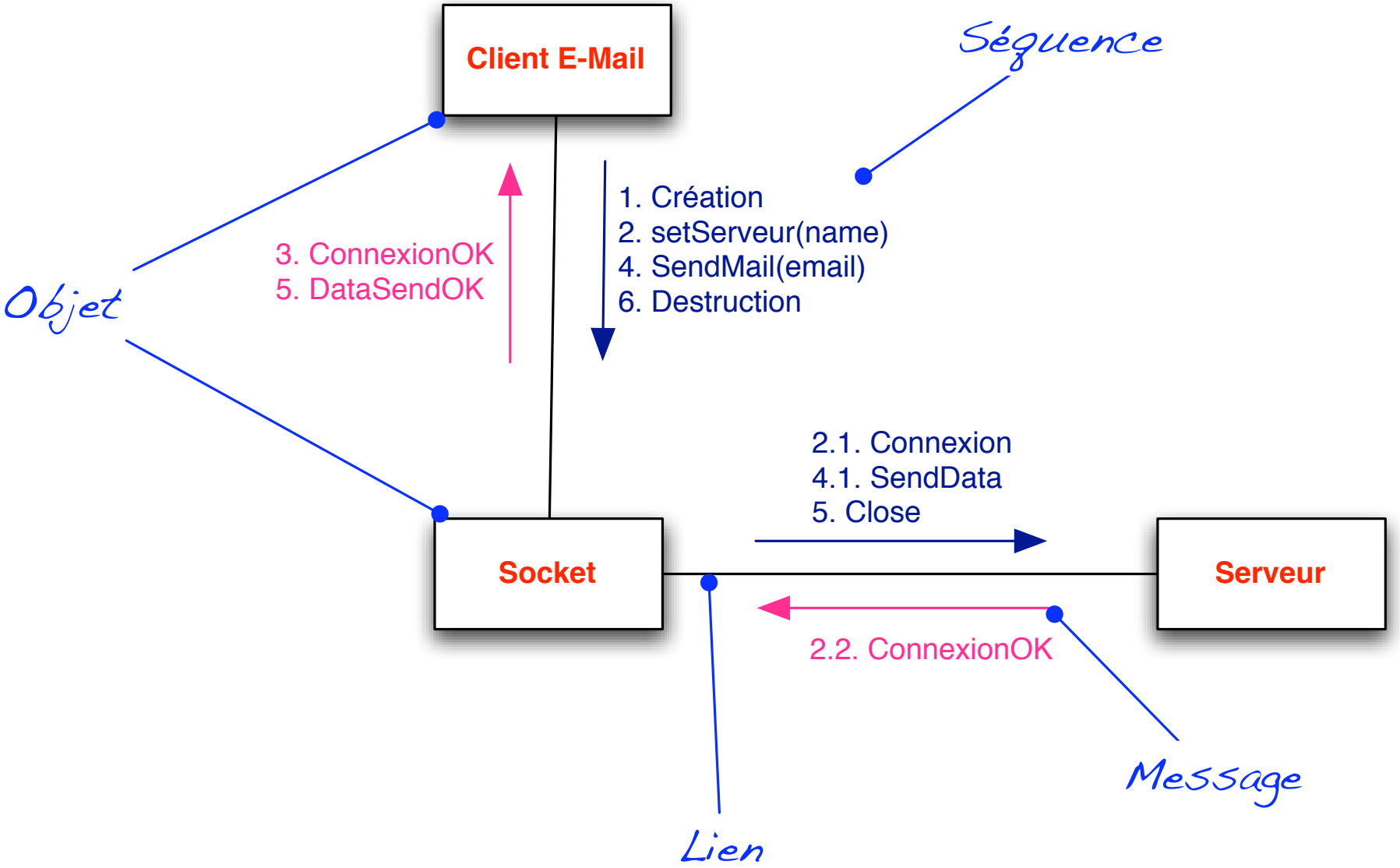
Modélisation d'un graphe de rôle



Le diagramme de collaboration

- La *modélisation temporelle* est en partie masquée,
  - Utilisation de la numérotation des séquences pour ordonner les traitements réalisés,
- Contrairement au diagramme de séquence, certaines informations ne sont pas représentées :
  - On ne modélise pas le ligne de vie des objets,
  - On ne modélise pas les temps d'activité des objets,
- Par contre on modélise explicitement les liens qui existent entre les différents objets.

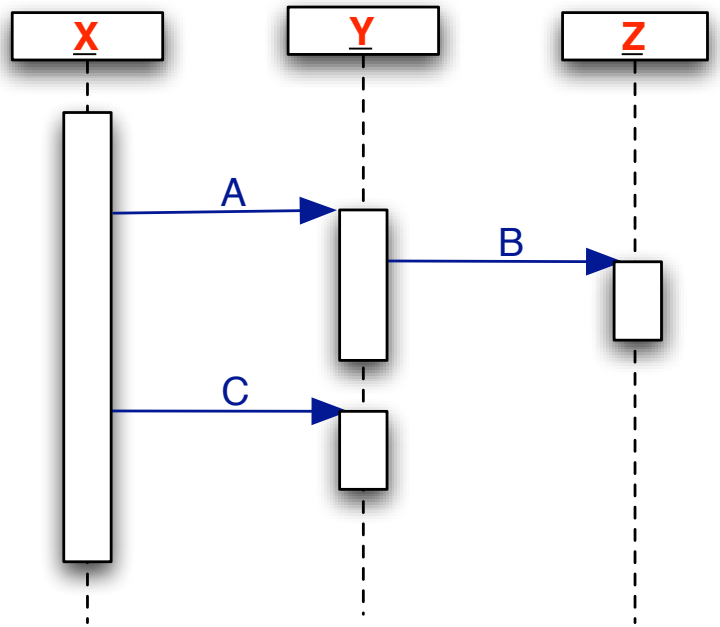
Exemple de diagramme de collaboration

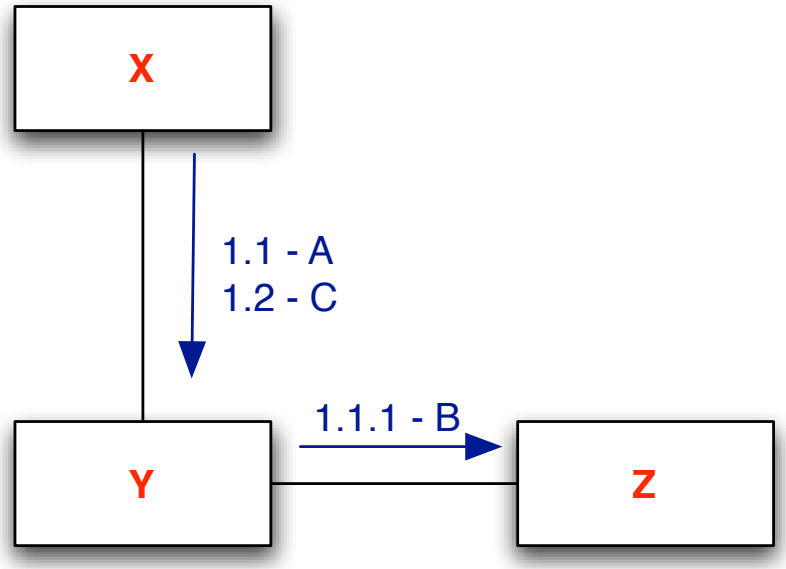


Utilisation de ces diagrammes

- Ces diagrammes sont utilisés pour modéliser les *comportements dynamiques* d'un système,
  - Modéliser le fonctionnement d'un sous système,
  - Possibilité de joindre ces diagrammes aux cas d'utilisation afin de spécifier un scénario.
- Il est possible de *rajouter des contraintes* sur les diagrammes :
  - Contraintes *temporelles* (durée d'exécution),
  - Formaliser le flot de contrôle : ajouter des *pré/post-conditions*.

Equivalence entre les diagrammes





La transition d'un modèle de représentation à l'autre est une tache aisée pour le concepteur. Ces 2 modèles expriment des informations différentes !

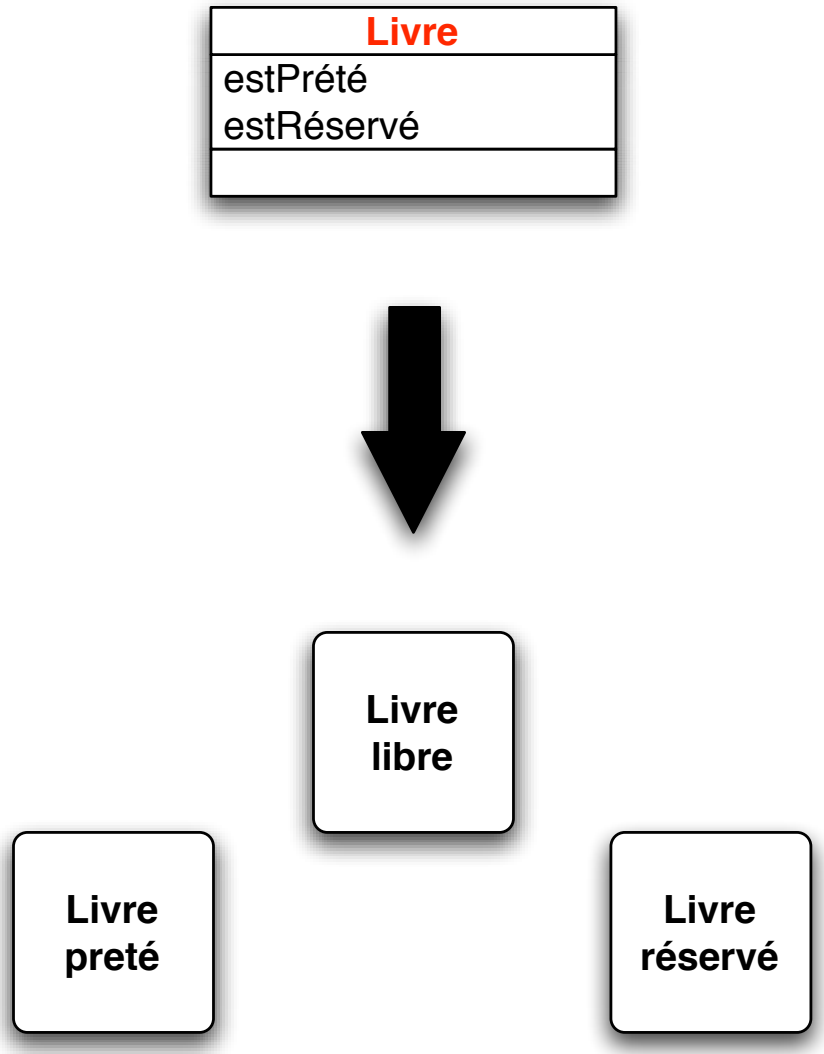
7

“ Diagrammes d’états-transitions ”

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

La recherche des états

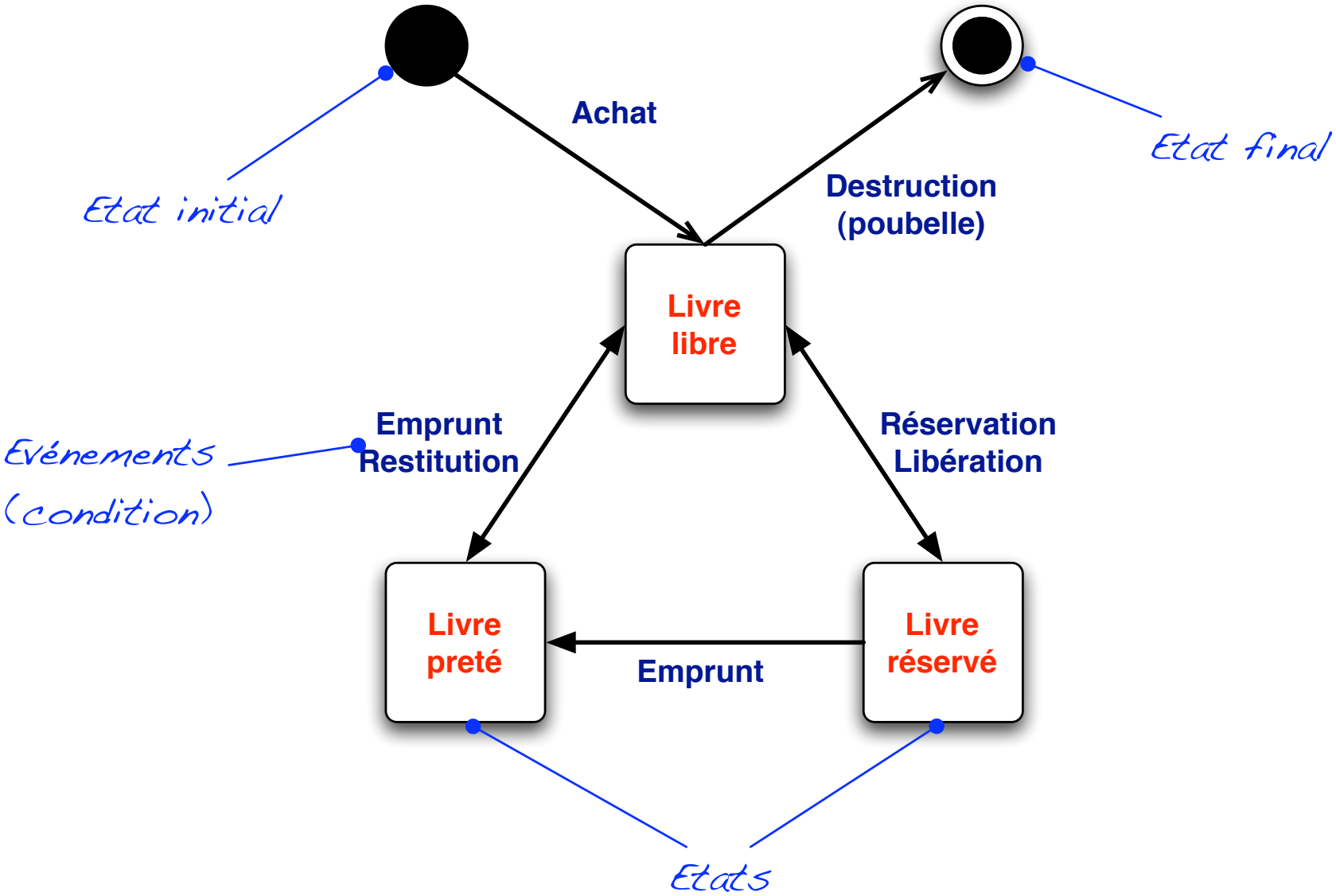
- L'état d'un objet est lié aux valeurs de ses attributs et de ses interactions avec les autres objets,
- Il est nécessaire de conserver les états significatifs,
- La réponse d'un événement à un stimuli dépend :
  - Du stimuli reçu par l'objet,
  - De l'état dans lequel se trouve l'objet.



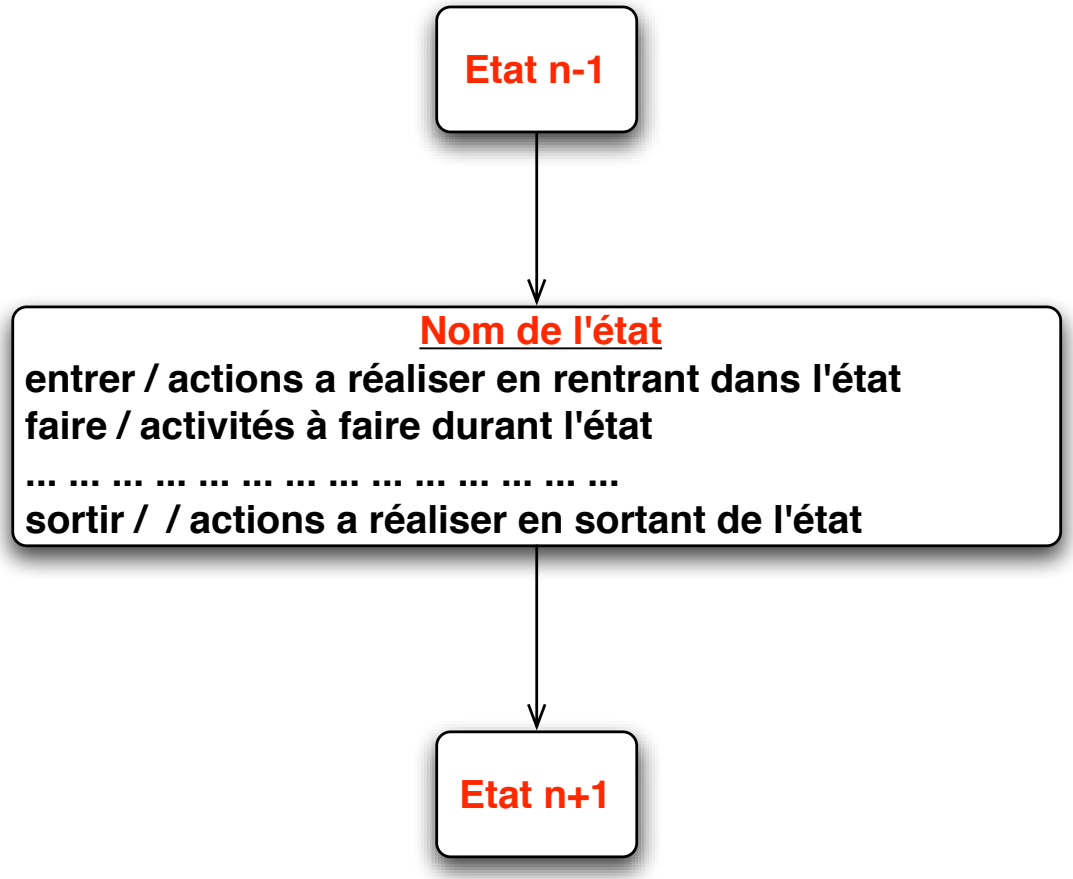
Règles liées au diagramme d'états

- Chaque transition (passage d'un état à un autre) est provoqué par un événement,
- Une transition n'a pas de durée (utilisation possible d'états de temporisation),
- La durée d'un état est l'intervalle qui s'écoule entre l'événement d'entrée et celui de départ.

Diagramme d'états d'un livre



Sémantique liée à la représentation



8

“ Et la conception dans tout cela … ”

ENSEIRB

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Et la conception du système ...

- La phase de conception a pour objectif de chercher comment est implémenté le système,
  - L'analyse vise quant à elle à identifier ce qui doit être fait (et non comment).
- L'approche est hiérarchique s'attachant d'abord au grandes parties du système et à leurs interactions, puis raffinement progressif,
- Prise en compte des contraintes intrinsèques et explicites exprimées à la conception (langage, BDD, temps d'exécution, processeurs, périphériques, etc.).
- C'est une étape ou on va faire intervenir des spécialistes de chaque discipline en fonction des technologies mises en oeuvre.

9

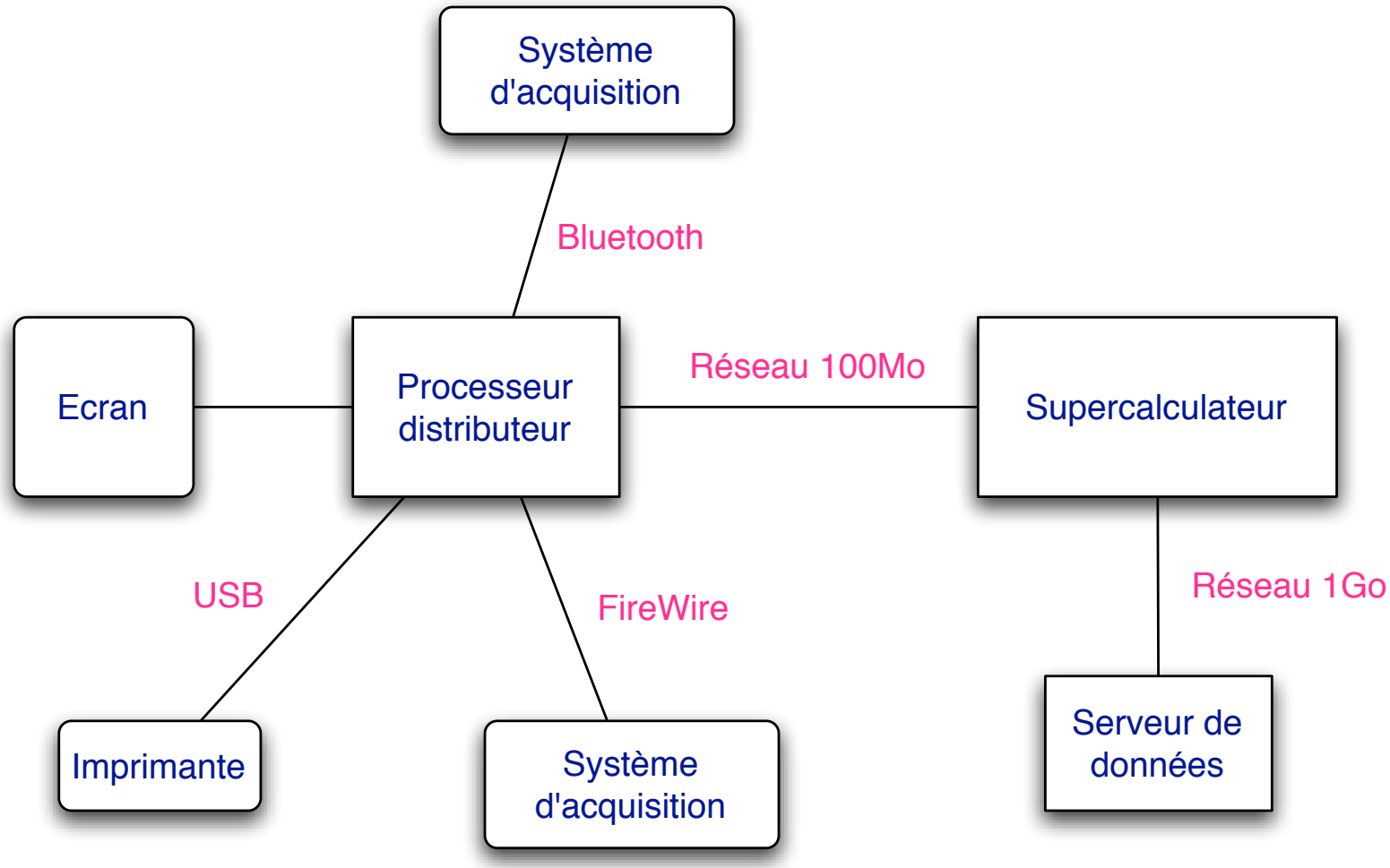
“ Diagrammes de déploiement ”

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Le diagramme de déploiement

- Ce modèle définit le diagramme de l'architecture matérielle du système ou de l'application,
- Il représente les différentes ressources matérielles à disposition du concepteur pur : processeurs et périphériques et la répartition de l'application sur le système hétérogène,
- Il précise aussi les liens de communication existants entre ces différentes ressources avec leurs caractéristiques.

Le diagramme de déploiement



10

“ Et la conception dans tout cela … ”

ENSEIRB

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

Conclusions sur UML

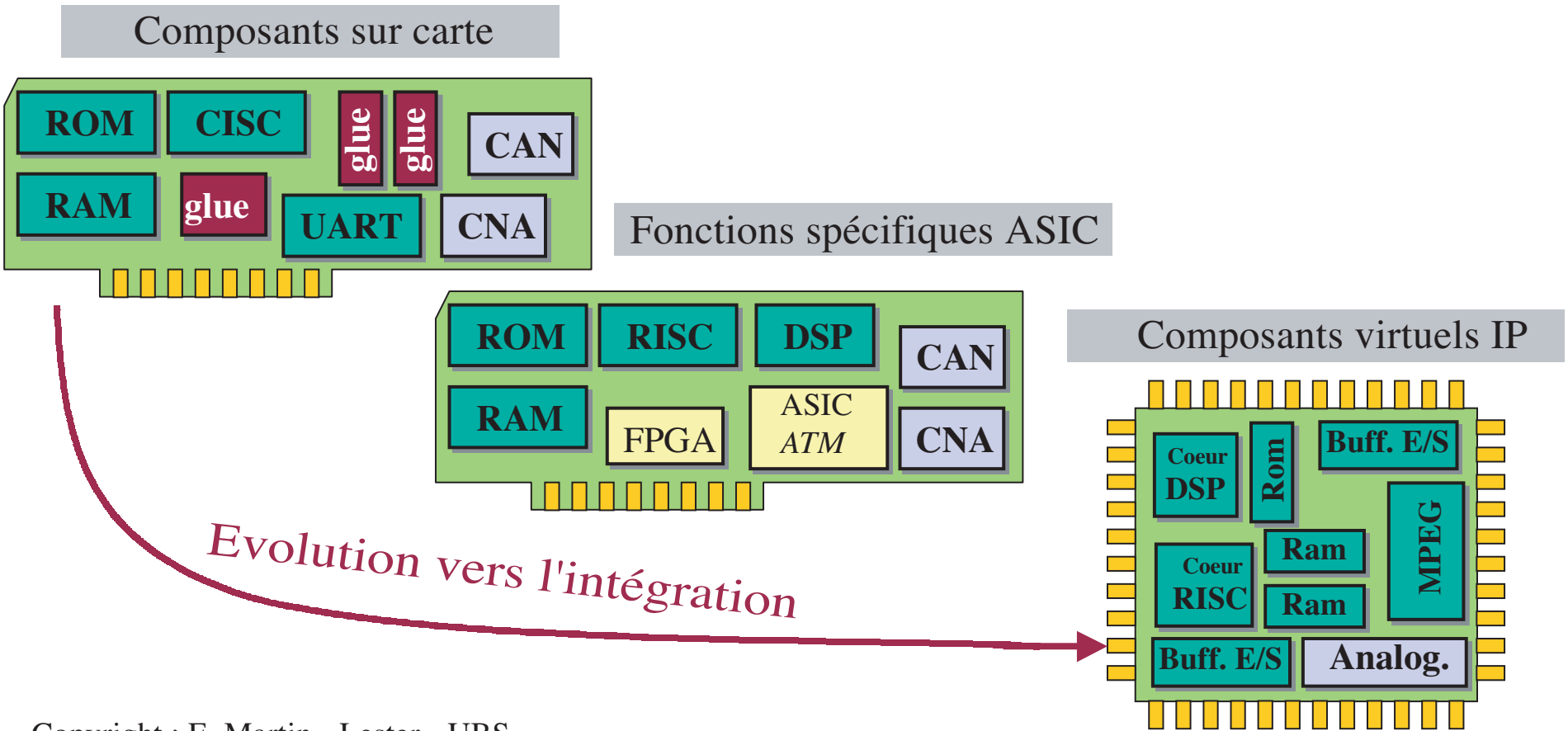
- Langage permettant d'appréhender la réalisation d'un système informatique complexe,
- Diagrammes qui permettent d'aider la réflexion, de permettre la discussion sur des bases normalisées et formalisées (clients, développeurs),
- Pas de solution unique mais un ensemble de solutions plus ou moins acceptables,
  - Contraintes clients (performances et fonctionnalités),
  - Architectures logicielles et matérielles à disposition,
- Des itérations successives du flot permettent d'aboutir à une solution.

Bibliographie

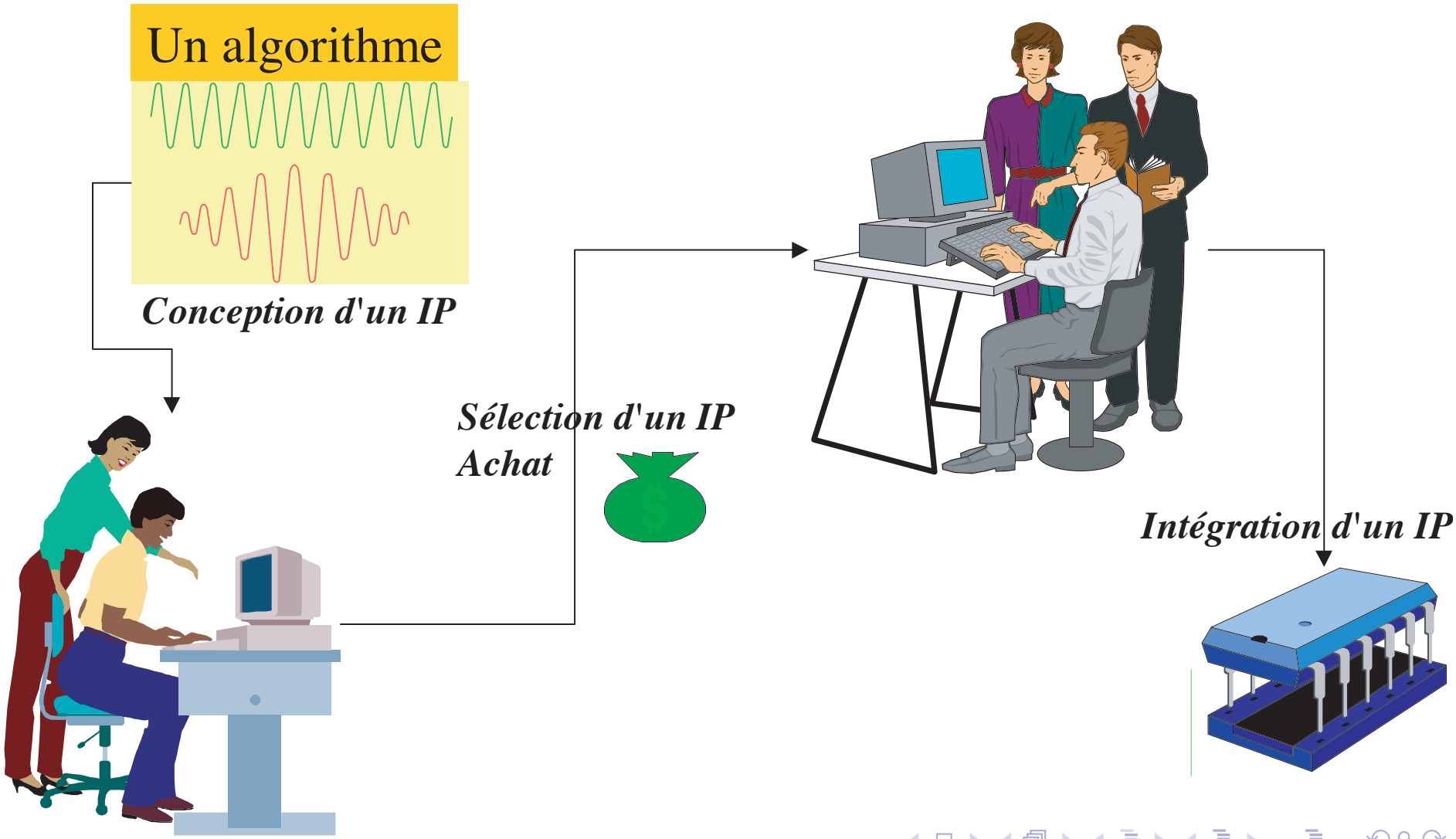
- **Modélisation avec UML**, *Robert Ogor*, ENST Bretagne, Mai 2003.
- **UML : Unified Modeling Language**, *Mireille Blay-Fornarino*, ESSI, Sophia Antipolis
- **UML**, *Martin Fowler*, Collection “Le tout en poche”, CampusPress
- **Le Guide de l’utilisateur UML**, *Grady Booch and al.*, Edition Eyrolles.

Sommaire

Notion d'IP : intellectual property



Notion d'IP : intellectual property



Notion d'IP : intellectual property

	Flot de conception	Représentation	Librairies	Technologie	Portabilité
Logiciel (soft) Non prédictible Très flexible	Conception système Conception RTL	Comporte-mental RTL	-	Indépendant technologie	Illimitée
Firm Flexible Prédictible	Synthèse de plan de masse Placement	RTL & blocs Netlist	De référence • Temps • Dessin	Technologie générique	Portable sur librairies
MatÈriel (hard) Pas flexible Très prédictible	Routage Vérification	PolygÙnes Données	Spécifique process Règles de dessin	Technologie fixe	Dépendant du process

Notion d'IP : intellectual property

- IP = Réutisabilité

Notion d'IP : intellectual property

- IP = Réutisabilité
- Définir un composant virtuel pouvant être utilisé dans plusieurs systèmes électroniques

Notion d'IP : intellectual property

- IP = Réutisabilité
- Définir un composant virtuel pouvant être utilisé dans plusieurs systèmes électroniques
- Exemples de ce type de composants : registres, Compteur, Additionneurs, etc ...

Notion d'IP : intellectual property

- Nécessité de règles permettant

Notion d'IP : intellectual property

- Nécessité de règles permettant
  - La réutilisation

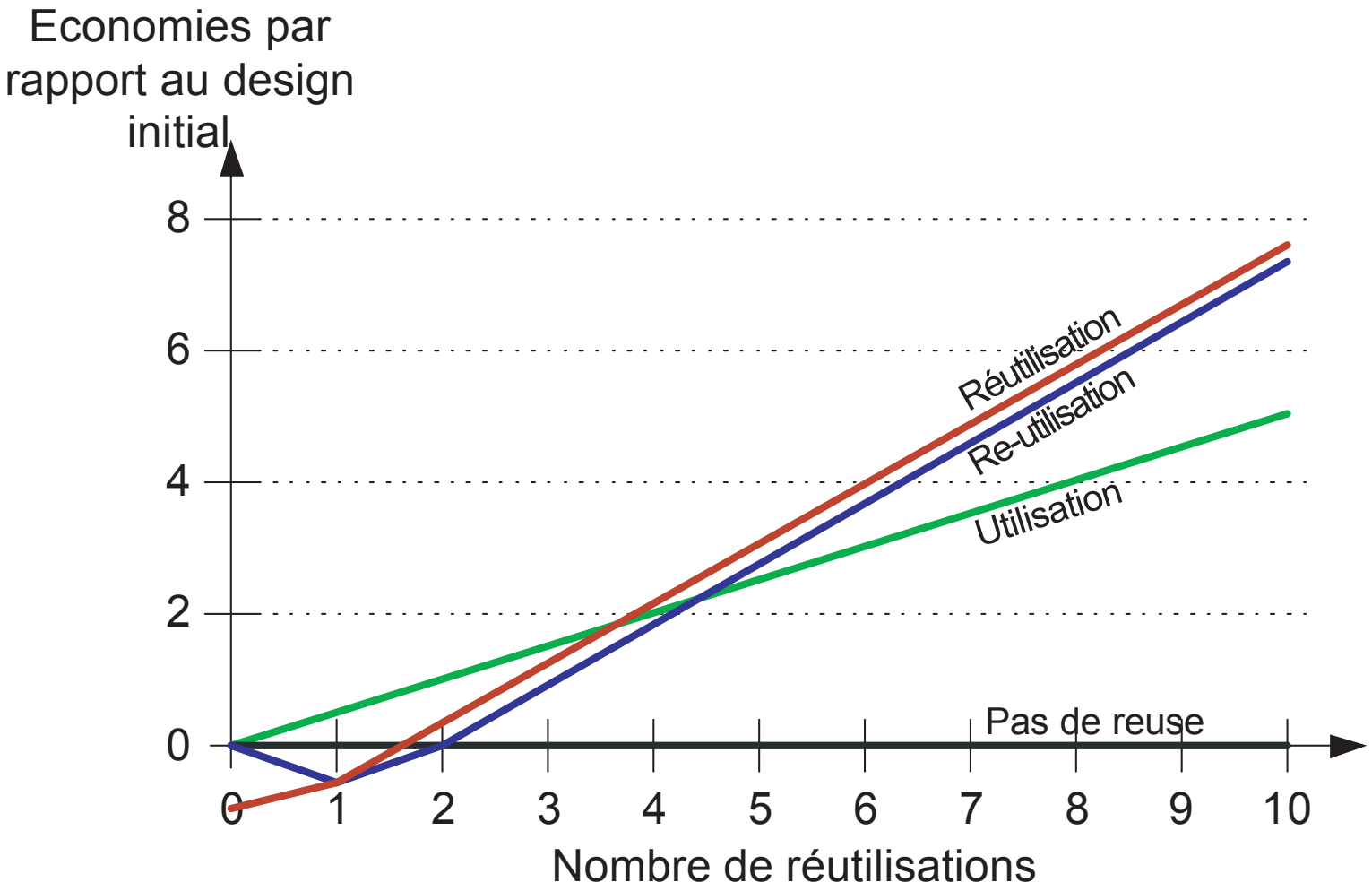
Notion d'IP : intellectual property

- Nécessité de règles permettant
  - La réutilisation
  - La fiabilité

Notion d'IP : intellectual property

- Nécessité de règles permettant
  - La réutilisation
  - La fiabilité
- Nécessité d'une bonne documentation

Notion d'IP : intellectual property



Sommaire

Bibliographie

- The designer's guide to VHDL - Peter Ashenden, Morgan Kaufman (http://www.ashenden.com.au/designers-guide/DG.html)
- Alain Vachoux : Digital System Modeling - http://lsmwww.epfl.ch/design_languages/
- Cours Master SdI - Patrick Garda (patrick.garda@upmc.fr)
- Ressources
  - Hamburg VHDL archive :
    - http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.htm
  - http://www.geocities.com/SiliconValley/Heights/8831/websrc.html

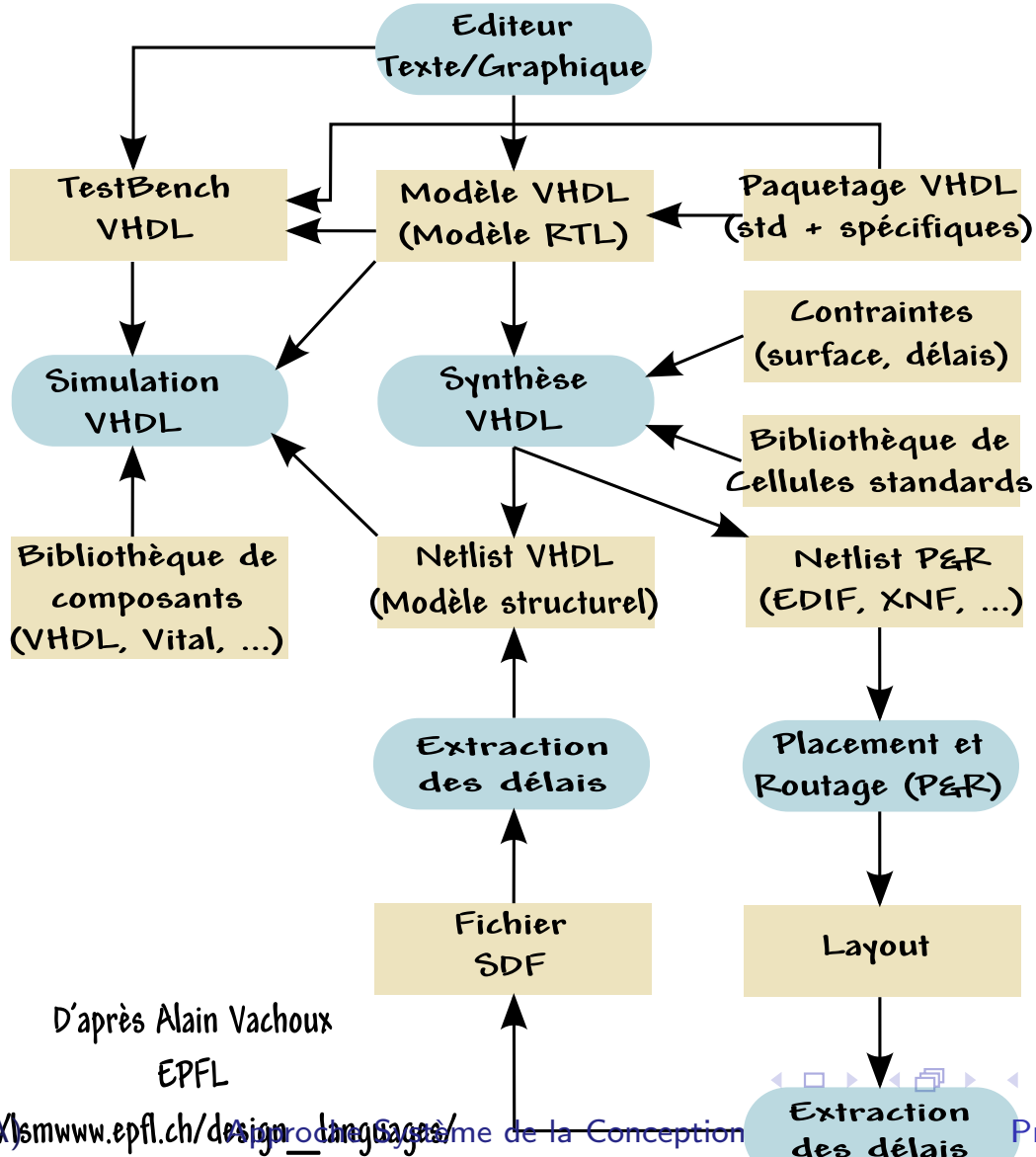
Outils

- Free Model Foundation - http://www.freemodelfoundry.com/
- Opencores - http://www.opencores.org
- XILINX : ISE - http://www.xilinx.com/
- ALTERA : Quartus - http://www.altera.com/
- Mentor Graphics : HDL Designer, ModelSim, Precision - http://www.mentor.com/

Historique

- VHDL : VHSIC Hardware Description Language Historique
- Langage introduit dans le cadre du projet DARPA VHSIC : Very High Speed Integrated Circuits
- IBM, Texas Instruments et Intermetrics ont obtenu le contrat en 1983 et produit VHDL "7.2" en 1985
- Proposé à IEEE pour normalisation en 1986
- Norme en 1987 : IEEE Std 1076-1987, dit VHDL-87
- Norme révisée en 1992 : IEEE Std 1076-1993, dit VHDL-93
- Normalisation simultanée de IEEE 1164-1993, dit std_logic_1164
- Norme révisée en 2002 IEEE Std 1076-2002, VHDL - 2002

Flot de conception VHDL



D'après Alain Vachoux
EPFL
http://lsmwww.epfl.ch/design_language/

VHDL - RTL

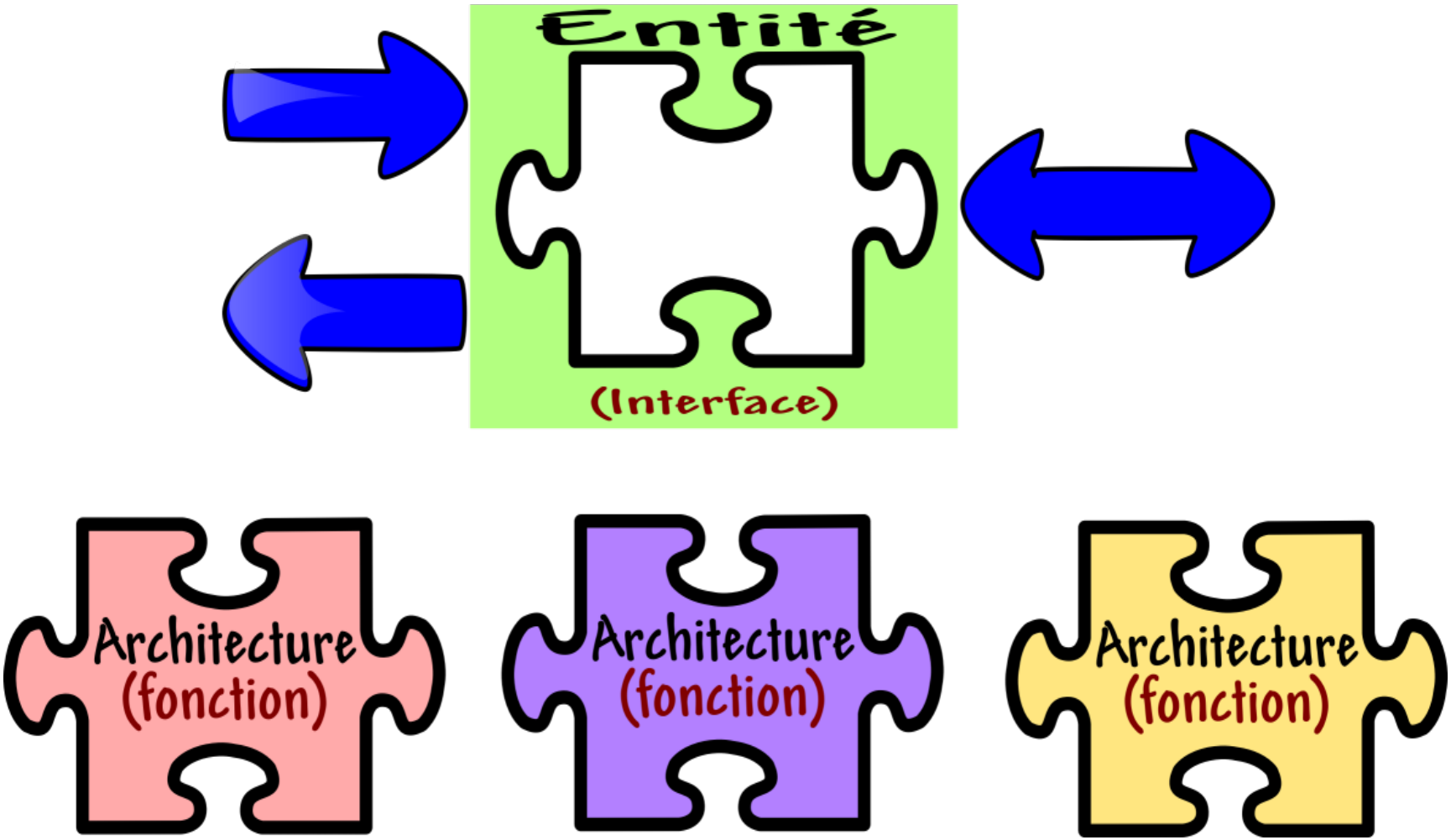
- RTL : Register Transfert Level
- Description Synthétisable
- Utilisable pour configurer un circuit logique programmable
- Sous ensemble de constructions VHDL

VHDL - Base

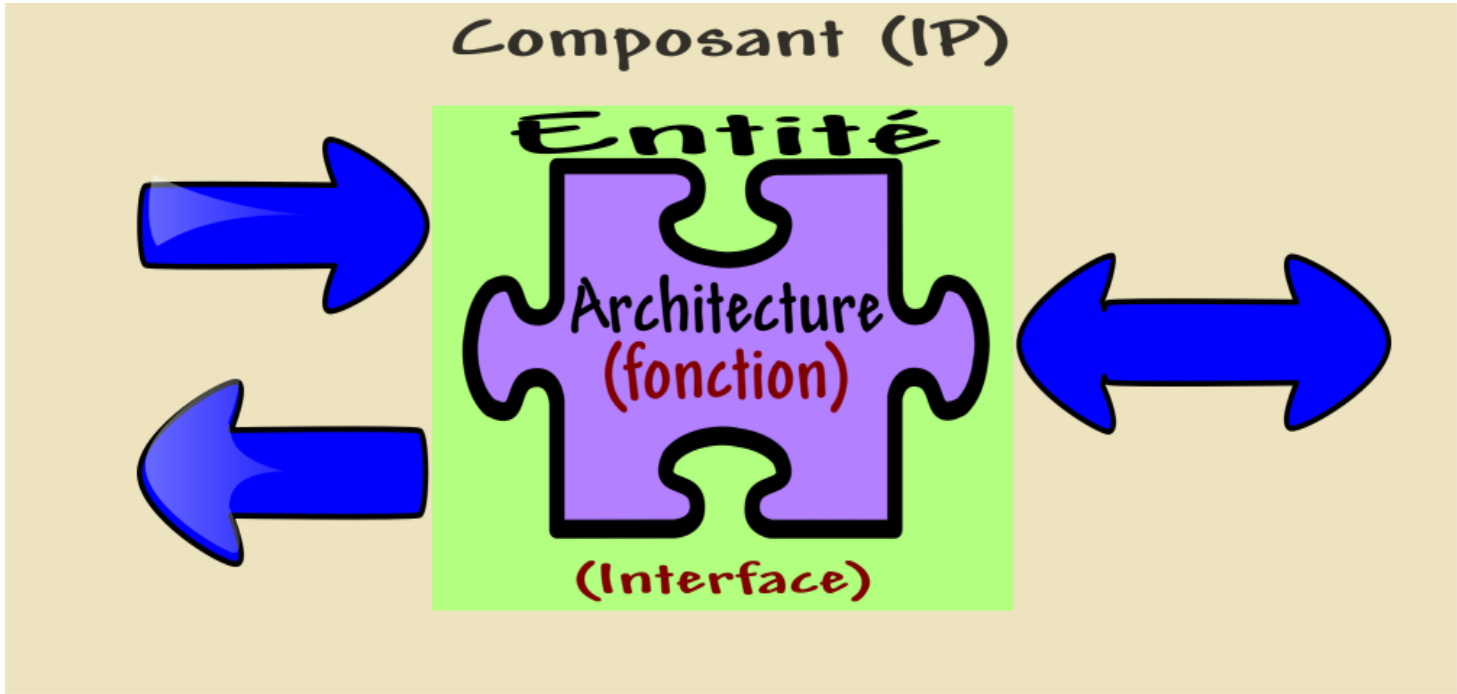
Blocs de base

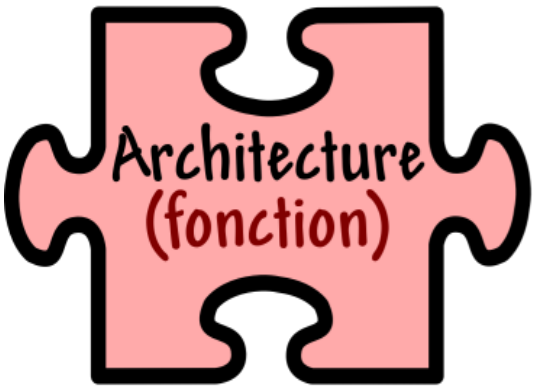
1. Les bibliothèques (ou librairies)
2. L'entité : Décrit l'interfaçage du composant
3. L'architecture : Décrit le fonctionnement du composant

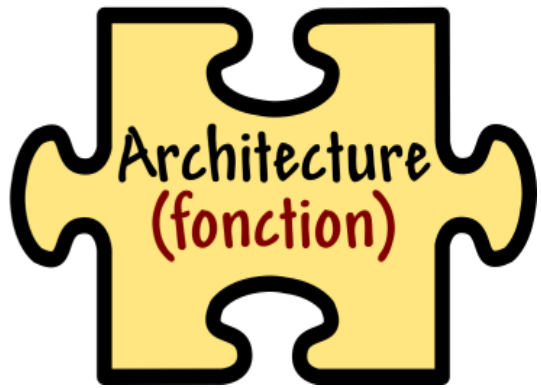
VHDL - Base


Entité
(Interface)
Architecture
(fonction)
Architecture
(fonction)
Architecture
(fonction)

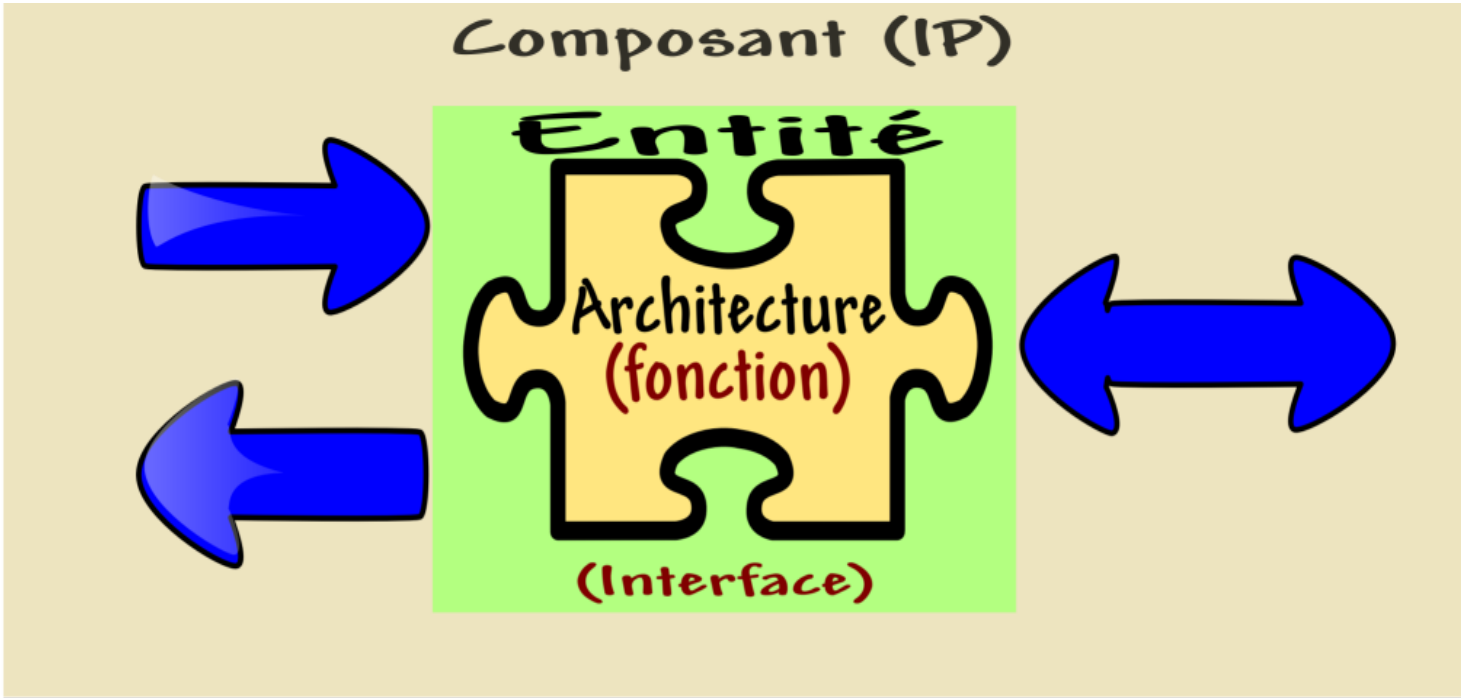
VHDL - Base

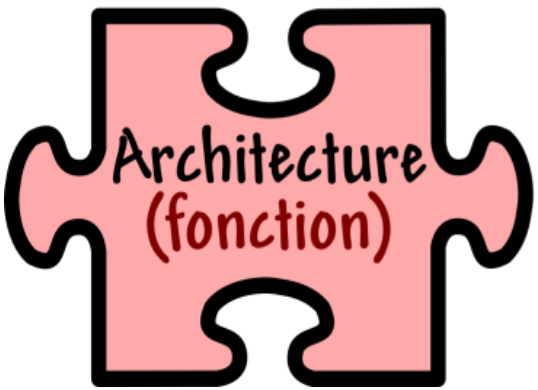

Composant (IP)
Entité
Architecture
(fonction)
(Interface)

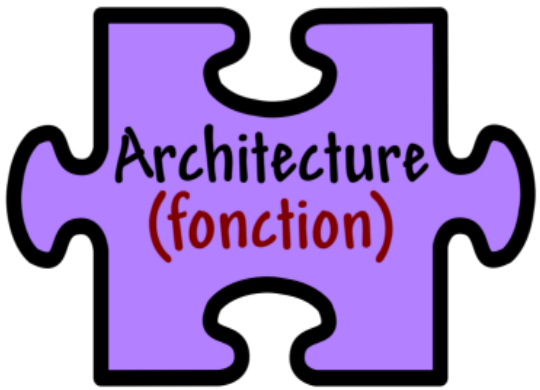

Architecture
(fonction)


Architecture
(fonction)

VHDL - Base


Composant (IP)
Entité
Architecture
(fonction)
(Interface)


Architecture
(fonction)


Architecture
(fonction)

VHDL - Bibliothèque

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

VHDL - Entité

context-clause

entity entity-name is

[generic (parameter-list)]

[port (port-list) ;]

[local-declarations]

type, subtype, constant, signal, subprogram

[begin

passive-concurrent-statement]

concurrent procedure call, assertion, passive process

end [entity] [entity-name] ;

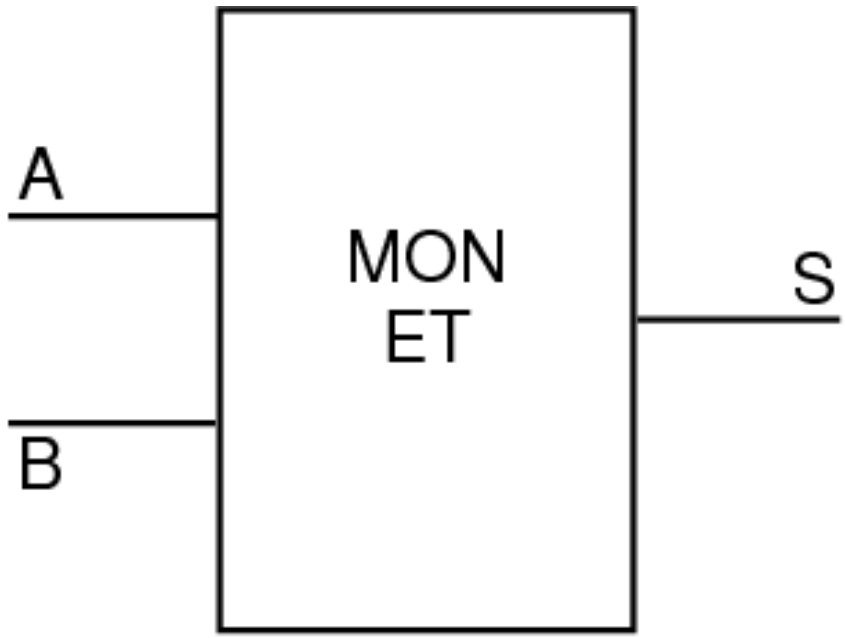
VHDL - Entité

```
generic(
parametre-name ,...: parametre-type [:=default-value];
...
parametre-name ,...: parametre-type [:=default-value]);
```

VHDL - Entité

```
port(
[signal] signal-name ,... : mode signal-type;
...
[signal] signal-name ,... : mode signal-type);
```

VHDL - Entité




```
entity MON-ET is
generic (tp: time := 2ns);
port( A : in std_logic;
      B : in std_logic;
      S : out std_logic);
end entity MON-ET;
```

VHDL - Architecture

architecture archi-name of entity-name is

[local-declaration]

begin

concurrent-statement

concurrent procedure call, assertion, passive process

end [architecture] [archi-name] ;

VHDL - L'architecture

$$S = A \text{ et } B$$

```
architecture FLOT of MON-ET is
begin
    s <= a and b after tp;
end architecture FLOT;
```

VHDL - L'architecture

- Une architecture décrit le fonctionnement d'une entité de conception
- C'est-à-dire la détermination des sorties en fonction des entrées
- Une architecture peut être décrite de différentes manières :
  - Flot de données
  - Structurelle
  - Comportementale
  - RTL : Register Transfer Level

Architecture Flot de données

- Une architecture "flot de données" (dataflow) décrit un circuit au niveau des portes logiques
- Elle représente le flot des informations des entrées (binaires) vers les sorties à l'aide d'opérateurs combinatoires :
  - not, and, nand, or, nor, xor, xnor
- Les signaux se propagent de façon asynchrone
- La détermination des valeurs est faite par l'instruction d'affectation des signaux $<=$
- Elle permet de modéliser le temps de propagation des portes grâce à la clause `after`

Architecture Flot de données

```
architecture flot of addc is
begin
sum <= a xor b xor cin after 2 ns ; -- somme
cout <= (a and b) or (b and cin) or (cin and a) after 1 ns ;
 -- retenue
end architecture flot;
```

Architecture Structurelle

- Utilise des composants déjà définis
- L'architecture structurelle définit alors la structure comme un assemblage d'instances de composants reliées par des signaux
- Elle correspond à la description textuelle d'un schéma utilisant des cellules de bibliothèque

Architecture Structurelle

```
 entity add4 is port (
a, b : in std_logic_vector (3 downto 0) ;
s : out  std_logic_vector (4 downto 0) )
end entity  add4 ;
```

Architecture Structurelle

```
architecture struct of add4 is
signal c : std_logic_vector (3 downto 1) ;
begin
addc_i0 : entity work.addc(flot)
port map ( a(0),  b(0),  '0',  s(0),  c(1)) ;
addc_i1 : entity work.addc(flot)
port map ( a(1),  b(1),  c(1),  s(1),  c(2)) ;
addc_i2 : entity work.addc(flo)t
port map ( a(2),  b(2),  c(2),  s(2),  c(3)) ;
addc_i3 : entity work.addc(flot)
port map ( a(3),  b(3),  c(3),  s(3),  s(4)) ;
end architecture struct ;
```

Architecture Structurelle

```
architecture struct of add4 is
signal c : std_logic_vector (3 downto 1) ;
begin
addc_i0 : entity work.addc(flot)
port map (a => a(0), b => b(0), cin => '0', sum => s(0), cout => c(1)) ;
addc_i1 : entity work.addc(flot)
port map (a => a(1), b => b(1), cin => c(1), sum => s(1), cout => c(2)) ;
addc_i2 : entity work.addc(flo)t
port map (a => a(2), b => b(2), cin => c(2), sum => s(2), cout => c(3)) ;
addc_i3 : entity work.addc(flot)
port map (a => a(3), b => b(3), cin => c(3), sum => s(3), cout => s(4)) ;
end architecture struct ;
```

L'instantiation de composants PORT MAP

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity demiadd is
port( a, b : in std_logic;
      c, s : out std_logic);
end entity demiadd;

architecture flot of demiadd is
begin
s <= a xor b;
c <= a and b;
end architecture flot;
```

L'instantiation de composants PORT MAP

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity add is
port( a, b, cin : in std_logic;
      cout, s : out std_logic);
end entity add;

architecture struct of add is
signal stmp, ctmp1,ctmp2 : std_logic;
begin
     demiadd1 : entity work.demiadd(flot)
                port map(a,b,stmp,ctmp1);
     demiadd2 : entity work.demiadd(flot)
                port map(cin,stmp,s,ctmp2);
     cout <= ctmp1 or ctmp2;
end architecture struct;
```

Architecture Comportementale

- L'architecture est représentée par un ensemble de processus séquentiels qui s'exécutent simultanément
- Chaque processus est en attente jusqu'à ce que l'une de ses entrées change de valeur
- Lorsque cela se produit le code du processus est exécuté séquentiellement
- Lorsqu'on arrive à la fin d'un processus on recommence son exécution au début
- La réalisation électronique de l'architecture n'est pas décrite

Le Process

- Permet de réaliser des parties séquentielles
- Introduit un niveau d'abstraction proche de l'informatique

Le Process

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity reg is
port( d : in std_logic_vector(7 downto 0);
      hor : in std_logic;
      q : out std_logic_vector(7 downto 0);
end entity reg;

architecture comport of reg is
begin
clocked: process(d,hor) is
      begin
          if (hor'event and hor = '1') then
               q<=d;
          end if;
      end process clocked;
end architecture comport;
```

Les variables

- Niveau d'abstraction niveau algorithmique
- Pas assimilable à un fil

L'affectation de variables

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity reg is
port( d : in std_logic_vector(7 downto 0);
      hor : in std_logic;
      q : out std_logic_vector(7 downto 0);
end entity reg;

architecture comport of reg is
begin
clocked: process(d,hor) is
      variable q_int : std_logic_vector(7 downto 0);
      begin
          if (hor'event and hor = '1') then
               q_int :=d;
          end if;
          q <= q_int;
      end process clocked;
end architecture comport;
```

Les structures de contrÙle

- Les structures Conditionnelles
  - La structure IF
  - La structure CASE
- Les structure de Répétitions LOOP
  - La structure FOR

Conditionnelle IF

if *condition1* then
instructions séquentielles 1
elsif *condition2* then
instructions séquentielles 2
elsif *condition3* then
instructions séquentielles 3
else
instructions séquentielles n
end if ;

Conditionnelle IF

```
library ieee;
use ieee.std_logic_1164.all;

entity decodeur is
port ( choix : in std_logic_vector(1 downto 0);
       decode : out std_logic_vector(3 downto 0));
end entity decodeur;

architecture comport of  decodeur
decodage : process(choix) is
           begin
             IF (choix= "00") THEN decode <="0001";
             ELSIF (choix="01") THEN decode <="0010";
             ELSIF (choix="10") THEN decode <="0100";
             ELSE  decode <="1000";
             END IF;
           end process decodage;
end architecture comport;
```

Conditionnelle IF

```
architecture behaviour of addc is
begin
 calc_sum : process (a, b, cin) is
 begin
   if ( ((a + b + cin) rem 2) = 0) then sum <= '0' after 2 ns ;
   else sum <= '1 ' after 2 ns ;
   end if ;
 end process calc_sum ;

 calc_cout :  process (a, b, cin) is
 begin
  if (a + b + cin >= 2) then cout <= '1' after 1 ns ;
  else cout <= '0' after 1 ns ;
  end if ;
 end process calc_cout ;
end architecture behaviour ;
```

Conditionnelle CASE

case *expression* is
    when *VALUE-1* =>
        instructions séquentielles 1
    when *VALUE-2* — *VALUE-3* =>
        instructions séquentielles 2
    when *VALUE-M to VALUE-N* =>
        instructions séquentielles 3
    when others =>
        instructions séquentielles n
end case ;

Conditionnelle CASE

```
library ieee;
use ieee.std_logic_1164.all;

entity decodeur is
port ( choix : in std_logic_vector(1 downto 0);
       decode : out std_logic_vector(3 downto 0));
end entity decodeur;

architecture comport of  decodeur
decodage : process(choix) is
           begin
             CASE choix
                 WHEN "00" => decode <="0001";
                 WHEN "01" => decode <="0010";
                 WHEN "10" => decode <="0100";
                 WHEN "11" => decode <="1000";
                 WHEN OTHERS => NULL;
             END CASE;
           end process decodage;
end architecture comport;
```

Boucle FOR

[loop_label]
for *identifier* in *discrete_range* loop
instructions du corps de boucle
end loop [loop_label] ;

- La variable *identifier* est
  - auto-déclarée
  - elle n'est visible que dans le corps de la boucle

Boucle FOR

```
entity additionneur is
    generic (N : natural := 8);
    port (a, b : in std_logic_vector(N-1 downto 0);
          s : out std_logic_vector(N-1 downto 0));

end additionneur;


architecture flot of additionneur is

begin  -- flot

    add : process(a,b)
        variable sum,btemp : std_logic_vector(N-1 downto 0);
        variable retenue : std_logic_vector(N downto 0);
    begin  -- process add
        btemps := b;
        retenue(0):='0';
        calcul : for i in 0 to N-1 loop
            sum(i):= a(i) xor btemp(i) xor retenue(i);
            retenue(i+1) := (a(i) and btemp(i)) or (a(i) and retenue(i)) or (btemp(i) and retenue(i));
        end loop calcul;
    end process add;
end flot;
```

Les différents Type VHDL

- Les types originaux
  - `Bit is ('0', '1');`
  - `Bit_vector is array (Natural range <>) of Bit;`
  - `Boolean is (false, true);`
  - `Character`
  - `Integer`
  - `Natural is Integer range 0 to Integer'high`
  - `Positive is Integer range 1 to Integer'high`
  - `Real`
  - `String is array (Positive range <>) of Character`
  - `Time`

```
units
fs; -- femtoseconde
ps = 1000 fs; -- picoseconde
ns = 1000 ps; -- nanoseconde
us = 1000 ns; -- microseconde
ms = 1000 us; -- milliseconde
sec = 1000 ms; -- seconde
min = 60 sec; -- minute
hr = 60 min; -- heure
end units;
```

Les différents Type VHDL

- Les types ajoutés par ieee (paquetage 1164)
  - std_ulogic is (
    'U', -- Non Initialisé - Non synthétisé
    'X', -- Forçage Indéterminé - Non synthétisé
    '0', -- Forçage à 0 - Synthèse = 0
    '1', -- Forçage à 1 - Synthèse = 1
    'Z', -- Haute Impedance - Synthèse = Z
    'W', -- Faible Indéterminé - Non synthétisé
    'L', -- Faible à 0 - Synthèse = 0
    'H', -- Faible à 1 - Synthèse = 1
    '-' -- Peu Importe - Non synthétisé
  - std_ulogic_vector is array (natural range <>) of std_ulogic;

Std_logic type résolu de std_ulogic

	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Les variables

- Les variables sont utilisées dans les instructions séquentielles
- Elles ne correspondent à aucune réalité électronique
- Elles jouent le rôle des variables dans un langage de programmation
- Elles ont une portée limitée au processus dans lequel elles sont déclarées
- Elles sont typées

Les variables

- Affectation
  - Symbole d'affectation de variables :=
  - L'affectation d'une variable est instantanée
- Utilisation
  - Affectation d'un signal à une variable
  - Exécution d'un algorithme séquentiel
  - Affectation d'une variable à un signal
  - Autre utilisation : identificateur de boucle

Les signaux

- Les signaux représentent les données physiques échangées entre les modules (anglais data signal)
- Chaque signal sera matérialisé dans le circuit final par une équipotentielle
- Exemples :
  - ports d'entrées et de sorties d'une entité
  - signaux internes à une architecture

Les signaux

- Instruction d'affectation de signaux :
  - s <= d after delay ;
  - s est le signal
  - d est le driver
  - delay est le délai

Les signaux

- Chaque signal a un type, comme dans un langage structuré
- Le type définit l'ensemble des valeurs que peut prendre le signal.
- Les signaux de type synthétisable sont synthétisables.
- VHDL est fortement typé, pas de conversion automatique
  - Nécessité d'avoir recours à des fonctions de conversion

Les signaux

- Types énumérés
  - type boolean is (false, true) ;
  - type character is (liste_des_caractères)
    - Tous les caractères ISO 8 bits
    - Une constante caractère est notée 'A'
  - type bit is ('0', '1') ;
  - type severity_level is (note, warning, error, failure) ;

Les signaux

- Entiers :
  - type integer is range -2147483 to 2147482 ;
  - subtype positive is integer range 1 to integer'high ;
  - subtype natural is integer range 0 to integer'high ;
- Remarques sur entiers :
  - Valeurs négatives codées en complément à 2
  - Entiers synthétisés comme bus 32 bits par défaut
- Intervalles :
  - Subtype byte is integer range -128 to 127 – codé 8 bits C2
- Réels :
  - type real is range $- to $+
  - Réels pas synthétisables

Les signaux

```
library ieee;
use ieee.std_logic_1164.all;

entity aff is
port( a : in std_logic;
      d : out std_logic_vector(3 downto 0));
end entity aff;

architecture flot of aff is
signal aint : std_logic;
begin
  aint <= '1';
  a<=aint; -- Erreur On ne peut pas ecrire sur une entree
  d<="0001";
end architecture flot;
```

Les opérateurs Logiques

- Les opérateurs logiques :

and	et bit à bit
or	ou bit à bit
xor	ou-exclusif bit à bit
not	non bit à bit

- Les opérateurs arithmétiques

+	Addition	
-	Soustraction	
*	Multiplication	
/	Division	Peu Synthétisable
mod	Modulo	Pas Synthétisable
exp	Exponentitation	Pas synthétisable

Les Tableaux

- Déclaration
  - type nom is array (intervalle) of type_de_base
- Utilisation
  - Modélisation de bus.
  - Modélisation de mémoires.
- Synthèse
  - Tableaux à 1 dimension
  - Indices entiers ou sous-type entiers
  - Eléments du tableau synthétisables

Les Tableaux

- Types tableaux prédéfinis
  - type bit_vector is array (natural range <>) of bit ;
  - type string is array (positive range <>) of character ;
  - type std_ulogic_vector is array (natural range <>) of std_ulogic ;
  - type std_logic_vector is array (natural range <>) of std_logic ;
- Exemples de modélisation de bus
  - Subtype byte is std_logic_vector (7 downto 0) ;
  - Subtype word is std_logic_vector (15 downto 0) ;

Generic

- Les paramètres génériques sont définis dans l'entité, avant les ports d'entrées-sorties
- Ils peuvent être utilisés :
  - Dans l'entité après leur déclaration
  - Dans le corps de toute architecture associée à l'entité

```
entity MON-ET is
generic (tp: time := 2ns);
port( A : in std_logic;
      B : in std_logic;
      S : out std_logic);
end entity MON-ET;
```

Generic

- La valeur du paramètre générique est précisée lors de l'instanciation de l'entité
- Des instances différentes peuvent utiliser des valeurs différentes

```
entity doubleor is
 port (in1, in2 : in std_logic;
       out2 : out std_logic);
end entity doubleor;

architecture struct of doubleor is
signal out1 : std_logic;
begin
Gate1 : entity work.or2(behaviour)
        Generic map (2 ns) ;
        Port map (in1, in2, out1) ;
Gate2 : entity work.or2(behaviour)
        Generic map (T_pd => 3 ns) ;
        Port map (a => out1, b => in2, y => out2) ;
end architecture struct;
```

Generic

- Une valeur par défaut du paramètre générique peut être indiquée lors de sa déclaration
- Lors de l'instanciation, cette valeur peut être :
- Utilisée telle quelle
- Remplacée par une autre valeur

```
entity dff is
Generic (
 T_pd, T_su : time := 2 ns ; T_h : time := 0 ns) ;
Port (clock, d : in std_logic ;
      q : out std_logic)
end entity dff;

architecture struct of reg is
.......
Request_dff : entity work.dff (behaviour)
              Generic map (4 ns, 3 ns, open) ;
              Port map (system_clock, request, prending_request)
.......
end architecture struct;
```

Generate

- L'instruction concurrente generate permet de dupliquer des instructions concurrentes de manière itérative ou conditionnelle.

Generate

```
entity shiftreg is
  generic (nbits: positive := 8);
  port (clk, rst, d: in std_logic;
        q: out std_logic);
end entity shiftreg;

architecture structure of shiftreg is
 component dff is
      port (clk, rst, d: in std_logic;
            q: out std_logic);
 end component dff;

signal qint: std_logic _vector(1 to nbits-1);

Begin

  cell_array: for i in 1 to nbits generate
    first_cell: if i = 1 generate
     dff1: dff port map (clk, rst, d, qint(1));
    end generate first_cell;

    int_cell: if i > 1 and i < nbits generate
     dffi: dff port map (clk, rst, qint(i-1), qint(i));
    end generate int_cell;

    last_cell: if i = nbits generate
     dffn: dff port map (clk, rst, qint(nbits-1), q);
    end generate last_cell;
  end generate cell_array;
end architecture structure;
```

Les machines à états en VHDL

- Moore ou Mealy
- Utilisation d'au moins 2 process
  - Un process de transition de l'état futur à l'état présent
  - Un process de détermination des sorties et de l'état futur
- Utilisation de type énuméré

Machines de Moore - 2 process - Entité

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY mae IS
   PORT(
      a   : IN     std_logic;
      hor : IN     std_logic;
      raz : IN     std_logic;
      b   : OUT    std_logic
   );
END mae ;
```

Machine de Moore - 2 process - Architecture

```
ARCHITECTURE diagram OF mae IS

   TYPE STATE_TYPE IS (Etat0,Etat1,Etat2);

   SIGNAL EtatPresent : STATE_TYPE ;
   SIGNAL EtatFutur : STATE_TYPE ;

BEGIN
   clocked : PROCESS(hor,raz)

   BEGIN
      ...
   END PROCESS clocked;

   nextstate : PROCESS (EtatPresent,a)
   BEGIN
      ...
   END PROCESS nextstate;
END diagram;
```

Machine de Moore - 2 process - Architecture

```
clocked : PROCESS(hor,raz)
BEGIN
   IF (raz = '0') THEN
      EtatPresent <= Etat0;
   ELSIF (hor'EVENT AND hor = '1') THEN
      EtatPresent <= EtatFutur;
   END IF;
END PROCESS clocked;
```

Machine de Moore - 2 process - Architecture

```
nextstate : PROCESS (EtatPresent,a)
BEGIN
   CASE EtatPresent IS
   WHEN Etat0 =>
      b <= '1';
      EtatFutur <= Etat1;
   WHEN Etat1 =>
      b <= '0';
      IF (a = '1') THEN
         EtatFutur <= Etat2;
      ELSIF (a = '0') THEN
         EtatFutur <= Etat1;
      ELSE
         EtatFutur <= Etat1;
      END IF;
   WHEN Etat2 =>
      b <= '0';
      EtatFutur <= Etat0;
   WHEN OTHERS =>
      EtatFutur <= Etat0;
   END CASE;
END PROCESS nextstate;
```

Machine de Mealy - 2 process - Entité

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY mae IS
   PORT(
      a   : IN     std_logic;
      hor : IN     std_logic;
      raz : IN     std_logic;
      b   : OUT    std_logic
   );
END mae ;
```

Machine de Mealy - 2 process - Architecture

```
ARCHITECTURE diagram OF mae IS
   TYPE STATE_TYPE IS (Etat0,Etat1,Etat2);

   SIGNAL EtatPresent : STATE_TYPE ;
   SIGNAL EtatFutur : STATE_TYPE ;

BEGIN

   clocked : PROCESS(hor,raz)
   BEGIN
      ...
   END PROCESS clocked;

   nextstate : PROCESS (EtatPresent,a)
   BEGIN
      ...
   END PROCESS nextstate;
END diagram;
```

Machine de Mealy - 2 process - Architecture

```
clocked : PROCESS(hor,raz)

 BEGIN
    IF (raz = '0') THEN
       EtatPresent <= Etat0;
       -- Reset Values
    ELSIF (hor'EVENT AND hor = '1') THEN
       EtatPresent <= EtatFutur;
       -- Default Assignment To Internals

    END IF;

 END PROCESS clocked;
```

Machine de Mealy - 2 process - Architecture

```
nextstate : PROCESS (EtatPresent,a)
BEGIN
   CASE EtatPresent IS
   WHEN Etat0 =>
      b <= '1';
      EtatFutur <= Etat1;
   WHEN Etat1 =>
      IF (a = '1') THEN
         b <= '0';
         EtatFutur <= Etat2;
      ELSIF (a = '0') THEN
         b <= '1' ;
         EtatFutur <= Etat1;
      ELSE
         EtatFutur <= Etat1;
      END IF;
   WHEN Etat2 =>
      b <= '0';
      EtatFutur <= Etat0;
   WHEN OTHERS =>
      EtatFutur <= Etat0;
   END CASE;
END PROCESS nextstate;
```

La clause Wait

- `wait on liste_de_signaux`
- un évènement sur l'un des signaux de la `liste_de_signaux` provoque l'exécution du processus

```
half_adder : process is
begin
 sum <= a xor b  after T_pd ;
 carry <= a and b after T_pd ;
 wait on a, b ;
end process half_adder ;
```

```
half_adder : process(a, b) is
 begin
  sum <= a xor b  after T_pd ;
  carry <= a and b after T_pd ;
 end process half_adder ;
```

La clause Wait

- `wait until condition`
- condition est une condition booléenne
- Ce qui se passe :
  - Le processus est suspendu lorsqu'il arrive à l'instruction wait
  - Le processus est relancé lorsque condition est testée et vraie
- Liste de sensibilité de wait until
  - condition est testée lorsqu'un évènement se produit sur l'un des signaux qui y apparaissent
  - la liste de sensibilité est la liste des signaux qui apparaissent dans condition
- wait until condition
  - équivaut à : `wait on liste_des_signaux_de_condition until condition`

La clause Wait

- `wait for duree`
- Ce qui se passe :
  - Le processus est suspendu lorsqu'il arrive à l'instruction wait
  - Le processus est relancé après une temporisation duree

```
clock_gen : process  is
begin
  clock <= '1' after T_pw, '0' after 2* T_pw ;
  wait for 2* T_pw ;
end process clock_gen ;
```

  - Pas synthétisable

La clause Wait

- `Wait`
- Ce qui se passe :
  - Le processus est suspendu lorsqu'il arrive à l'instruction wait
  - Il reste suspendu jusqu'à la fin de la simulation
- Exemple d'application : test bench
- VHDL permet de décrire dans le même langage :
  - Le circuit à tester
  - La génération des signaux d'entrée.
  - La vérification des signaux de sortie.

Test Bench

- Génération de Stimuli
- Analyse des résultats
- Utilisation de clause `Assert`

Assert

- ASSERT condition
  REPORT string SEVERITY severity_level;

```
check_setup: PROCESS (clk, d)
BEGIN
      IF (clk'EVENT AND clk='1') THEN -- test si front m
            ASSERT d'STABLE(setup_time) -- regarde si d
                  REPORT "Setup Violation..." -- affiche
                  SEVERITY WARNING;
      END IF;
END PROCESS check_setup;
```

Assert

- Niveaux de sévérité :
  - Note : utilisé pour information seulement
    "Note : Chargement de données d'un fichier"
  - Warning : utilisé pour fournir une information sur une erreur en instance
    "Warning : Détection d'un pic"
  - Error : utilisé pour information seulement
    "Error : Violation du temps d'initialisation"
  - Failure : raporte une grosse erreur
    "Failure : Ligne RESET instable"

Simulation Evénementielle

Initialisation

1. Assign initial values to signals and variables.
2. $T_c = 0ns$, $\delta = 0$
3. Execute all processes until they suspend.
4. Determine next time T_n according to 4.

Cycle

1. Tc = Tn.
2. Update signals.
3. Execute all processes sensitive to updated signals.
4. Determine next time Tn:
   - if pending transactions at current time: $\delta = \delta + 1$ then 2
   - if no more pending transactions or Tn = time'high then STOP
   - else T_n = time of next earliest pending transaction, $\delta = 0$
5. Execute postponed processes.

Paquetage

```
package proc_pkg is
   subtype data is integer range 0 to 3;
   type darray is array (1 to 3) of data;
end package proc_pkg;

use work.proc_pkg.all;
entity procstmt is
 port (
       inar : in darray;
       outar: out darray);
end entity procstmt;
```

Procédure

```
architecture a of procstmt is
begin
  process (inar)
    procedure swap (d: inout darray; l, h: in postitive) is
       variable tmp: data;
       begin
         if d(l) > d(h) then
               tmp := d(l);
               d(l) := d(h);
               d(h) := tmp;
         end if;
       end swap;
    variable tmpar: darray;
    begin
     tmpar := inar;
      swap(tmpar,1,2);
      swap(tmpar,2,3);
      swap(tmpar,1,2);
      outar <= tmpar;
    end process;
 end architecture a;
```

Fonction

```
entity parity_check is
 generic (NBITS: positive := 8);
 port (
       data: in bit_vector(NBITS-1 downto 0);
       prty: out bit);
end entity parity_check;

architecture func of parity_check is
 begin
    process (data)
      function parity (bv: bit_vector) return bit is
        variable result: bit;
        begin
          result := '0';
          for i in bv'range loop
              result := result xor bv(i); -- odd parity
          end loop;
          return result;
        end function parity;
     begin
        prty <= parity(data);
     end process;
  end architecture func;
```

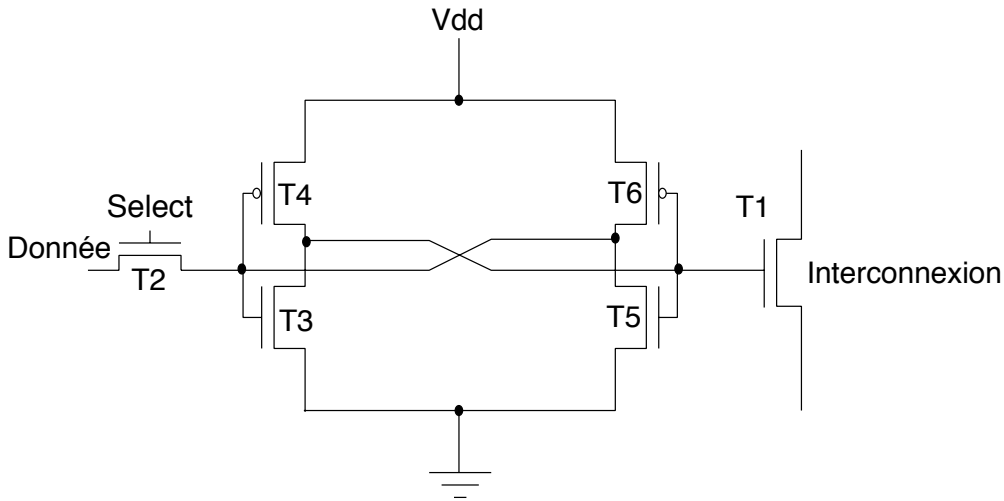
Fonction de conversion

		numeric_std	std_logic_arith
Type Conversion			
std_logic_vector	-> unsigned	unsigned(*arg*)	unsigned(*arg*)
std_logic_vector	-> signed	signed(*arg*)	signed(*arg*)
unsigned	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
signed	-> std_logic_vector	std_logic_vector(*arg*)	std_logic_vector(*arg*)
integer	-> unsigned	to_unsigned(*arg*, *size*)	conv_unsigned(*arg*, *size*)
integer	-> signed	to_signed(*arg*, *size*)	conv_signed(*arg*, *size*)
unsigned	-> integer	to_integer(*arg*)	conv_integer(*arg*)
signed	-> integer	to_integer(*arg*)	conv_integer(*arg*)
integer	-> std_logic_vector	integer -> unsigned/signed ->std_logic_vector	
std_logic_vector	-> integer	std_logic_vector -> unsigned/signed ->integer	
unsigned + unsigned	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
signed + signed	-> std_logic_vector	std_logic_vector(*arg1* + *arg2*)	*arg1* + *arg2*
Resizing			
unsigned		resize(*arg*, *size*)	conv_unsigned(*arg*, *size*)
signed		resize(*arg*, *size*)	conv_signed(*arg*, *size*)

Sommaire

F.P.G.A et SRAM

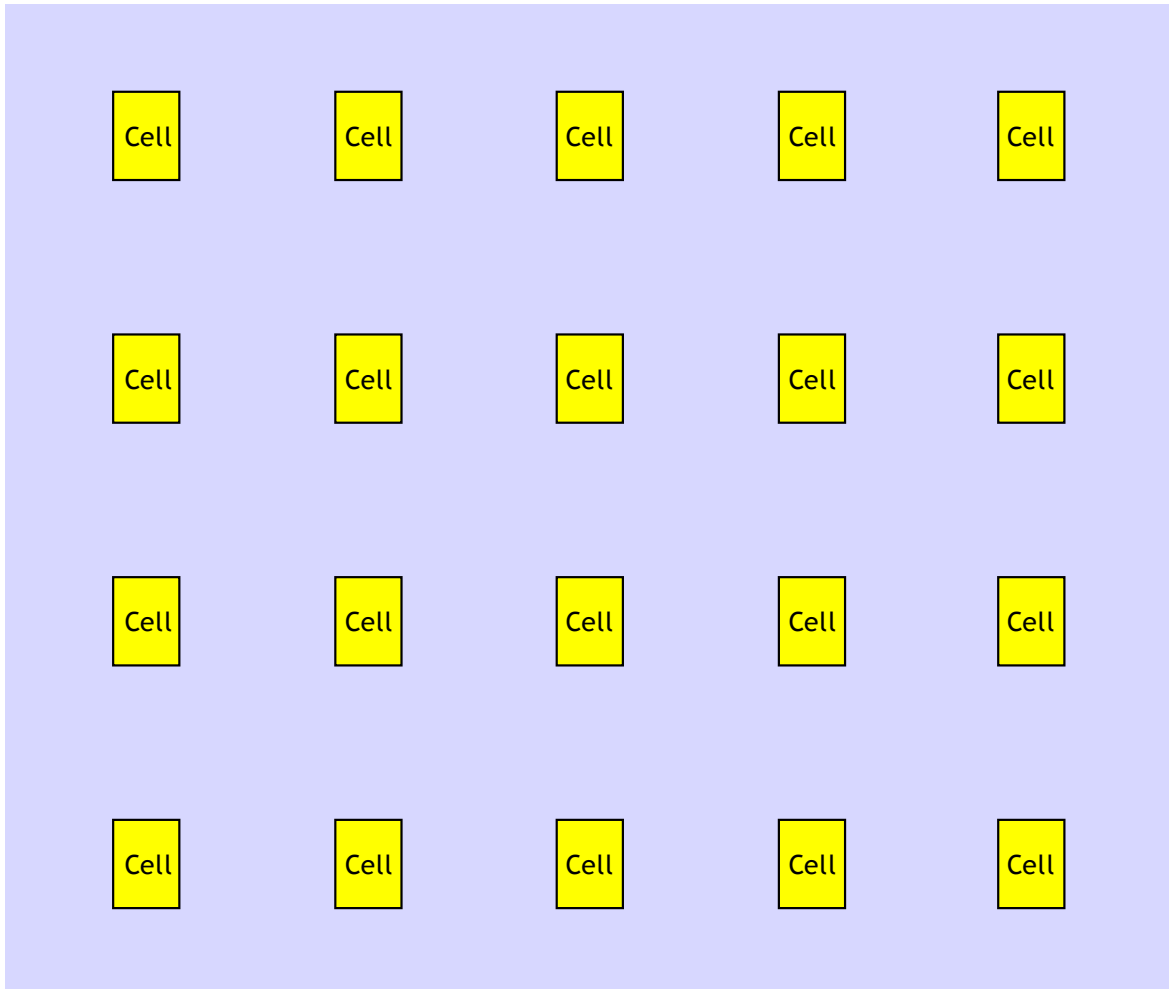
- Utilisation de cellule de mémoire statique



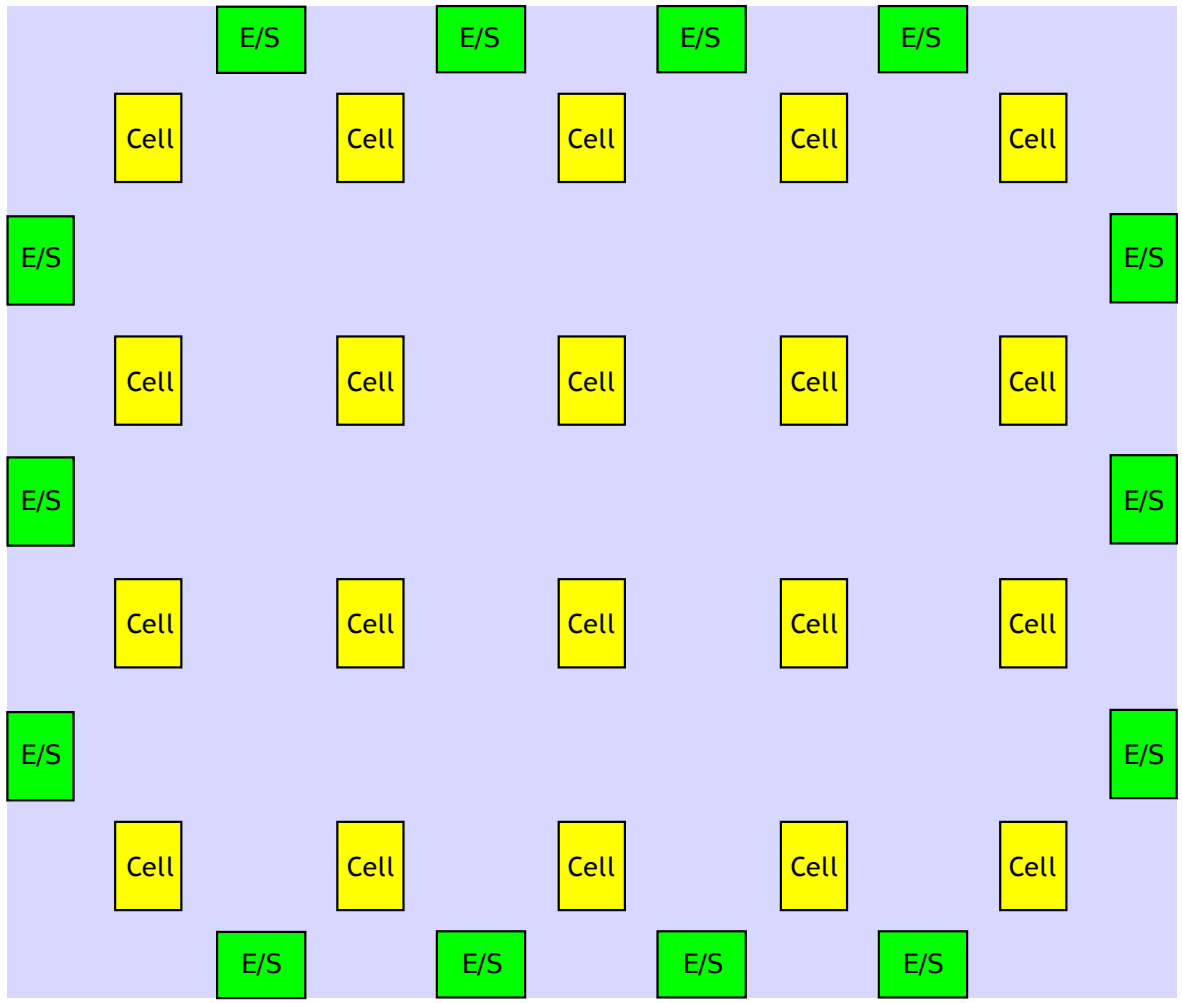
- Volatile
- Reprogrammable

Structure d'un F.P.G.A

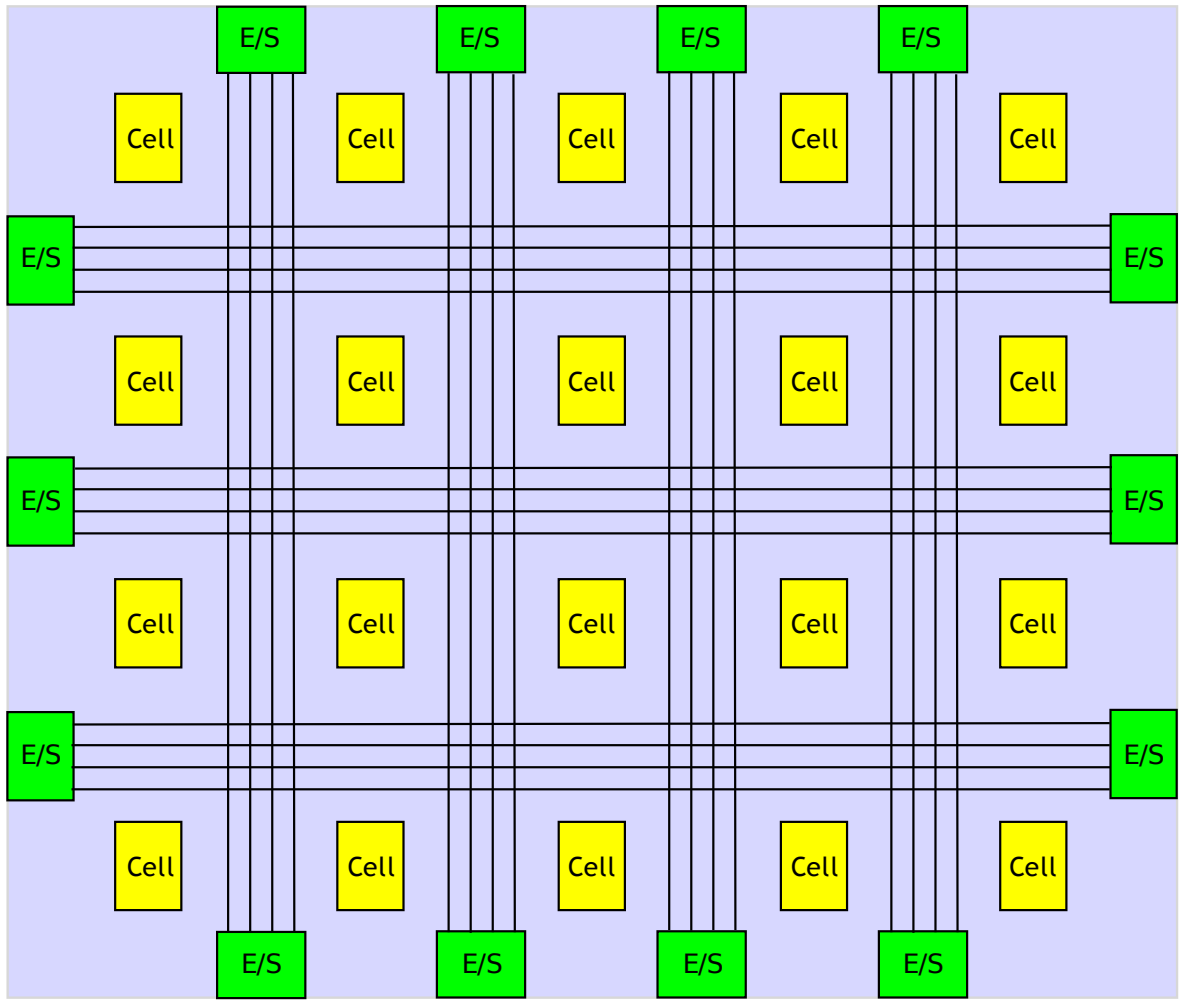
Structure d'un F.P.G.A



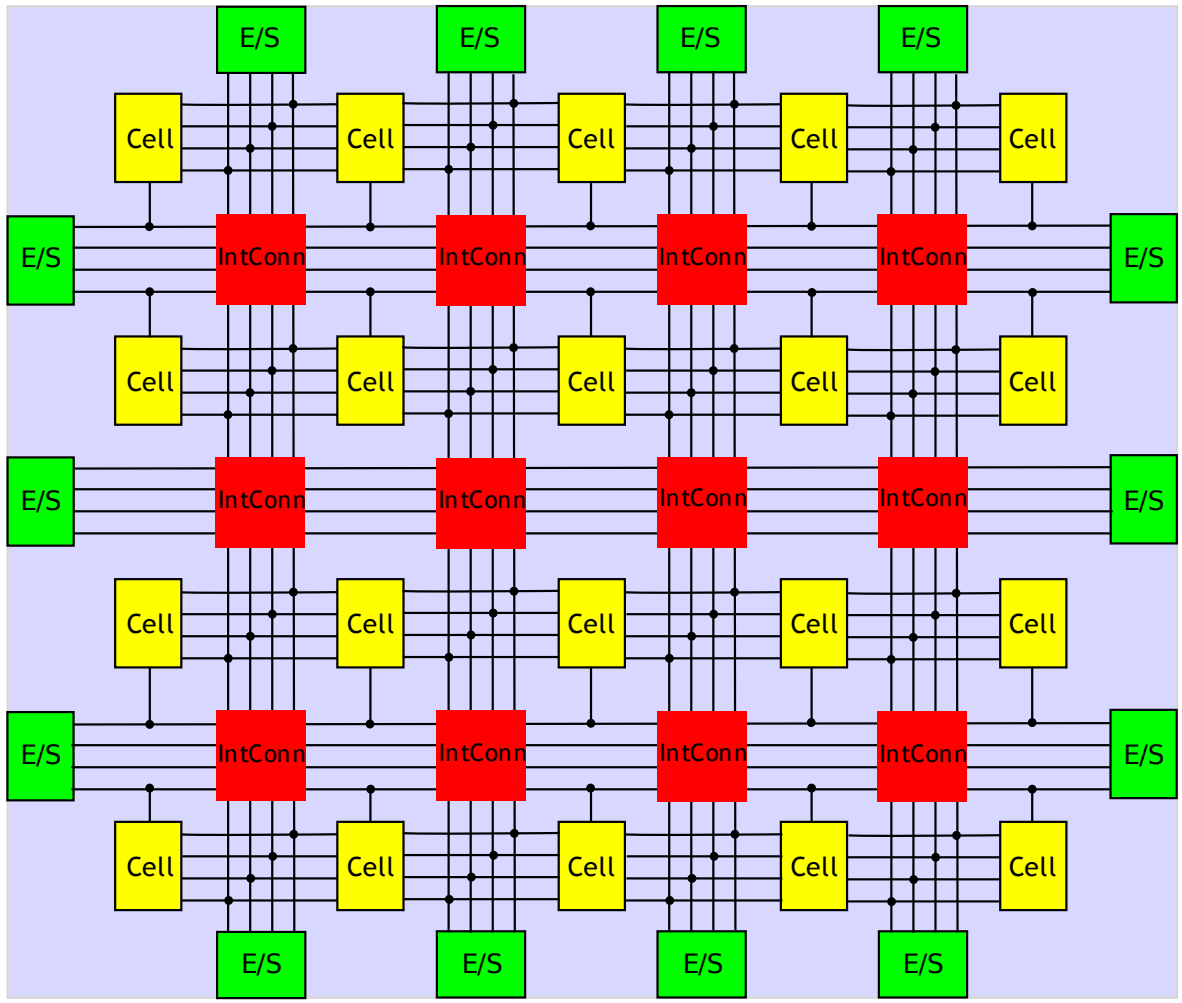
Structure d'un F.P.G.A



Structure d'un F.P.G.A



Structure d'un F.P.G.A



Qu'y a t'il dans un F.P.G.A ?

- Des Cellules de Base

Qu'y a t'il dans un F.P.G.A ?

- Des Cellules de Base
  - LUT (LookUp Table)- *Codage des fonctions* - **Combinatoire**

Qu'y a t'il dans un F.P.G.A ?

- Des Cellules de Base
  - LUT (LookUp Table)- *Codage des fonctions* - **Combinatoire**
  - Bascules - *Synchronisation* - **Séquentiel**

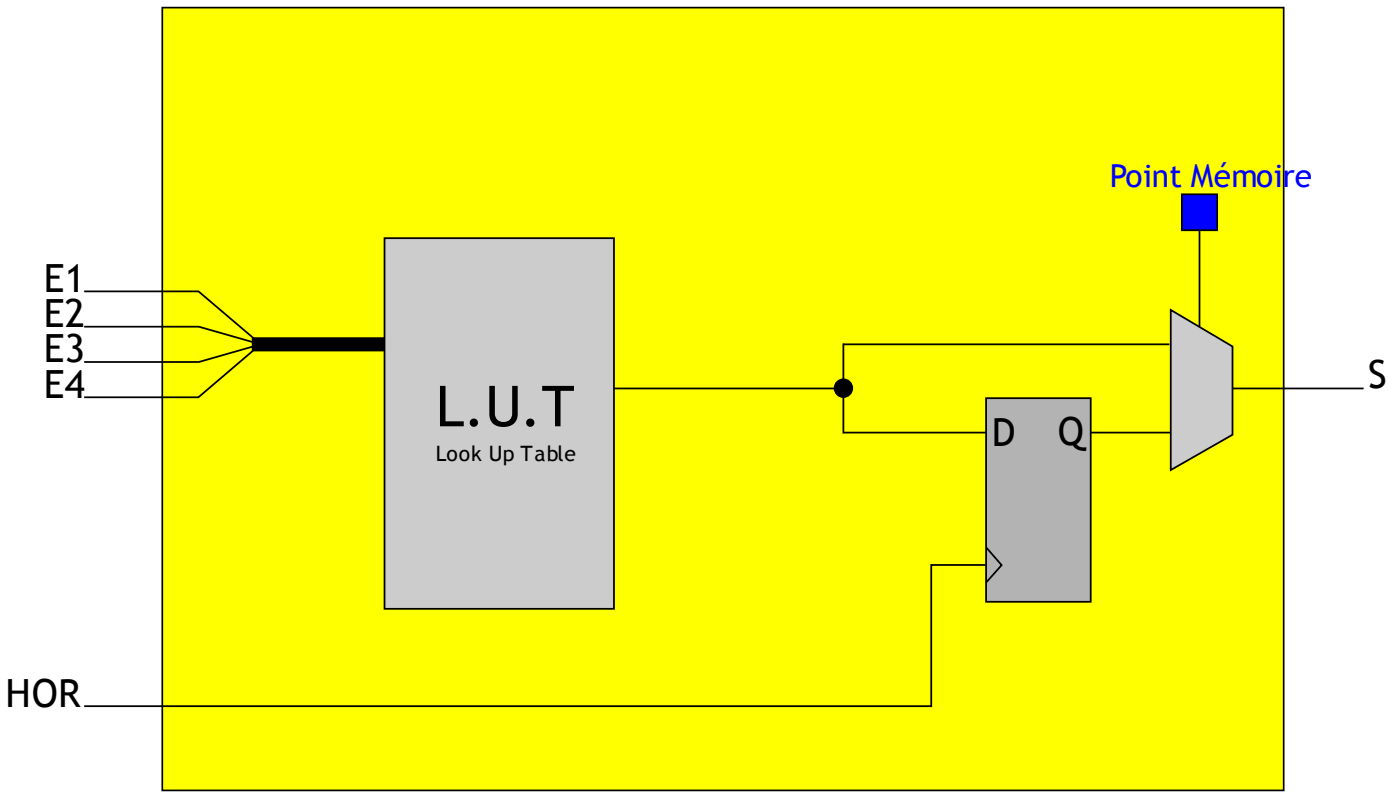
Qu'y a t'il dans un F.P.G.A ?

- Des Cellules de Base
  - LUT (LookUp Table)- *Codage des fonctions* - **Combinatoire**
  - Bascules - *Synchronisation* - **Séquentiel**
- Des Entrées-Sorties

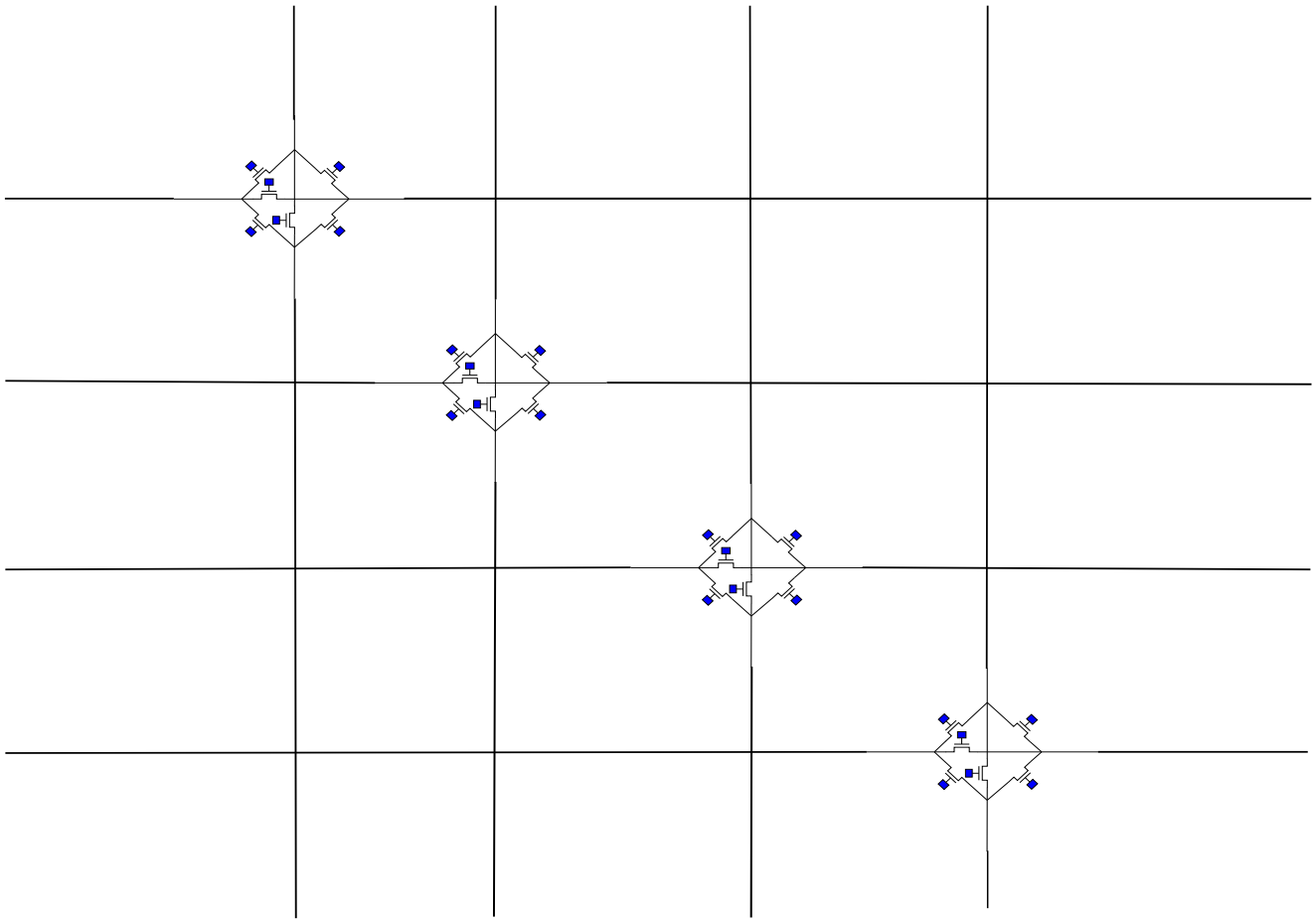
Qu'y a t'il dans un F.P.G.A ?

- Des Cellules de Base
  - LUT (LookUp Table)- *Codage des fonctions* - **Combinatoire**
  - Bascules - *Synchronisation* - **Séquentiel**
- Des Entrées-Sorties
- De la logique de routage

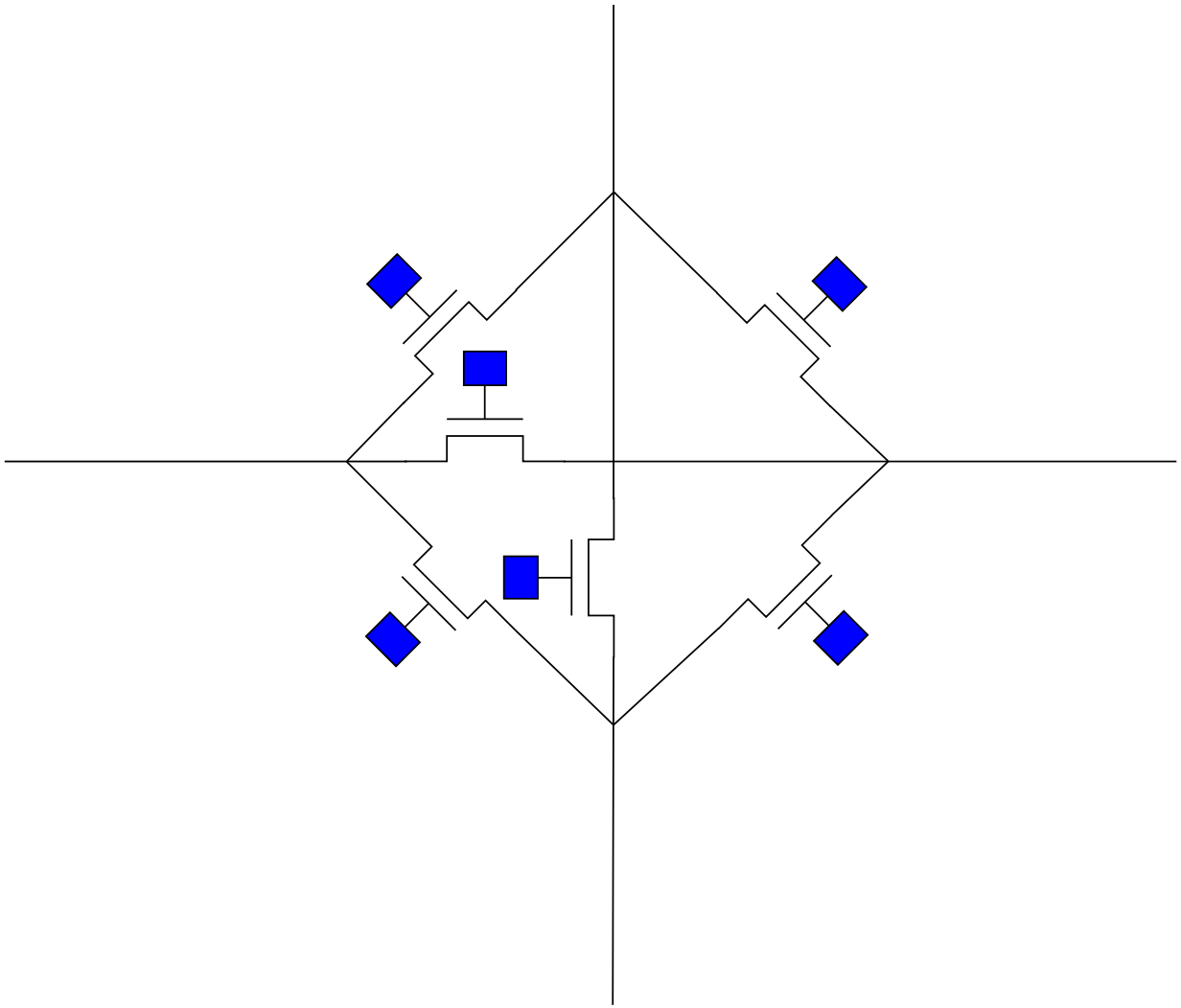
Structure d'une Cellule de Base



Structure de l'interconnexion



Structure de l'interconnexion



Exemple de F.P.G.A : Le Stratix d'Altera

- jusqu'à 114 140 Cellules

Exemple de F.P.G.A : Le Stratix d'Altera

- jusqu'à 114 140 Cellules
- jusqu'à 1 Moctet de mémoire embarquée

Exemple de F.P.G.A : Le Stratix d'Altera

- jusqu'à 114 140 Cellules
- jusqu'à 1 Moctet de mémoire embarquée
- mise en oeuvre de multiplieurs cadencé jusqu'à 250 MHz

Exemple de F.P.G.A : Le Stratix d'Altera

- jusqu'à 114 140 Cellules
- jusqu'à 1 Moctet de mémoire embarquée
- mise en oeuvre de multiplieurs cadencé jusqu'à 250 MHz
- 16 horloges globales différentes

Exemple de F.P.G.A : Le Stratix d'Altera

- jusqu'à 114 140 Cellules
- jusqu'à 1 Moctet de mémoire embarquée
- mise en oeuvre de multiplieurs cadencé jusqu'à 250 MHz
- 16 horloges globales différentes
- 12 PLL

Exemple de F.P.G.A : Le Stratix d'Altera - Mémoire

- 3 différents Blocs

Exemple de F.P.G.A : Le Stratix d'Altera - Mémoire

- 3 différents Blocs
- M512 : 512 bits organisés en 32x18 bits

Exemple de F.P.G.A : Le Stratix d'Altera - Mémoire

- 3 différents Blocs
- M512 : 512 bits organisés en 32x18 bits
- M4K : 4Kbits organisés en 128X36 bits

Exemple de F.P.G.A : Le Stratix d'Altera - Mémoire

- 3 différents Blocs
- M512 : 512 bits organisés en 32x18 bits
- M4K : 4Kbits organisés en 128X36 bits
- M-RAM : 576Kbits organisée en 4KX144 bits

Exemple de F.P.G.A : Le Stratix d'Altera - Multiplieurs

- 1 bloc DSP : 3 modes

Exemple de F.P.G.A : Le Stratix d'Altera - Multiplieurs

- 1 bloc DSP : 3 modes
- 8 Multiplieurs 9x9

Exemple de F.P.G.A : Le Stratix d'Altera - Multiplieurs

- 1 bloc DSP : 3 modes
- 8 Multiplieurs 9x9
- 4 Multiplieurs 18x18

Exemple de F.P.G.A : Le Stratix d'Altera - Multiplieurs

- 1 bloc DSP : 3 modes
- 8 Multiplieurs 9x9
- 4 Multiplieurs 18x18
- 1 Multiplieur 36x36

Exemple de F.P.G.A : Le Stratix d'Altera

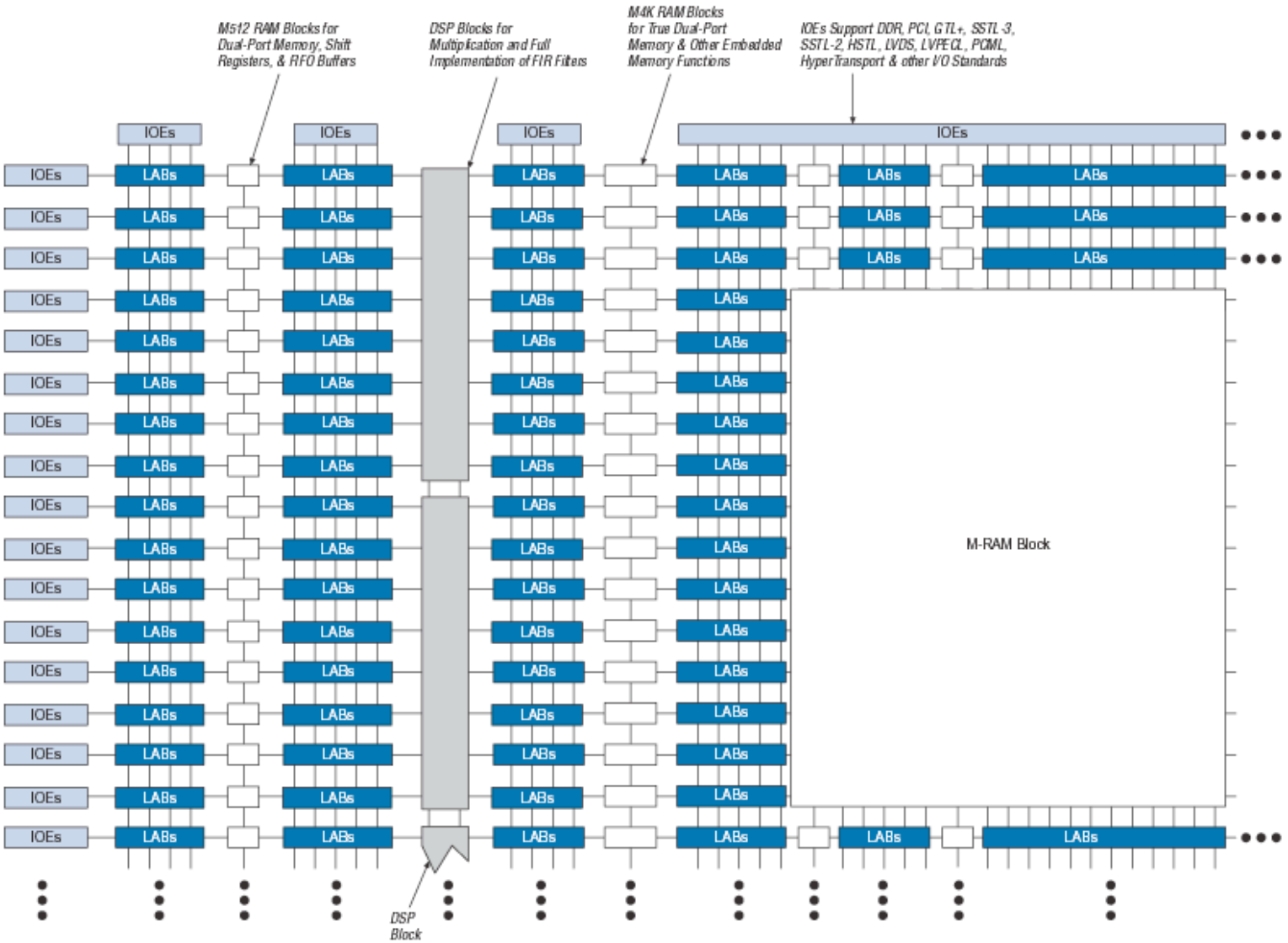
Table 1. Stratix Device Features (Part 1 of 2)

Feature	EP1S10	EP1S20	EP1S25	EP1S30
LEs	10,570	18,460	25,660	32,470
M512 RAM blocks (32 × 18 bits)	94	194	224	295
M4K RAM blocks (128 × 36 bits)	60	82	138	171
M-RAM blocks (4K × 144 bits)	1	2	2	4
Total RAM bits	920,448	1,669,248	1,944,576	3,317,184
DSP blocks	6	10	10	12
Embedded multipliers *(1)*	48	80	80	96
PLLs	6	6	6	10
Maximum user I/O pins	426	586	706	726

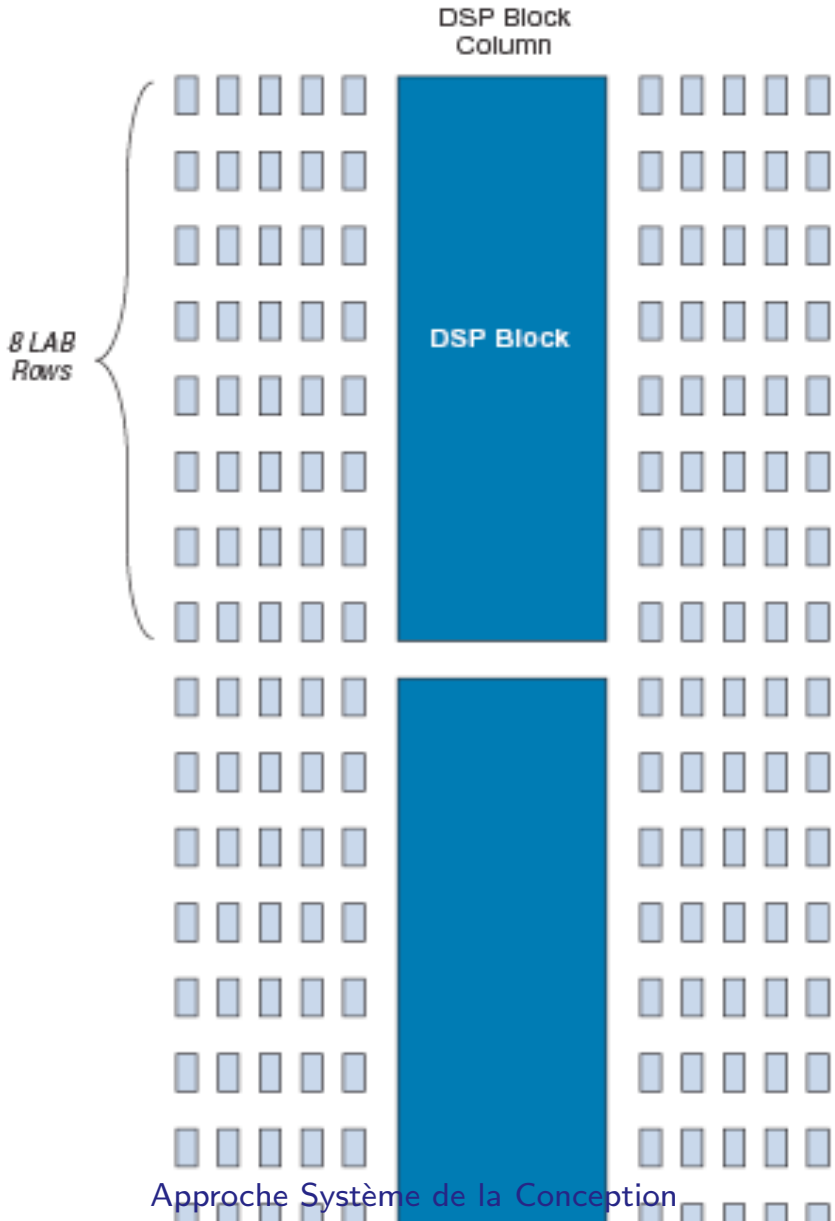
Stratix Device Features (Part 2 of 2)

Feature	EP1S40	EP1S60	EP1S80	EP1S120
LEs	41,250	57,120	79,040	114,140
M512 RAM blocks (32 × 18 bits)	384	574	767	1,118
M4K RAM blocks (128 × 36 bits)	183	292	364	520
M-RAM blocks (4K × 144 bits)	4	6	9	12
Total RAM bits	3,423,744	5,215,104	7,427,520	10,118,016
DSP blocks	14	18	22	28
Embedded multipliers *(1)*	112	144	176	224
PLLs	12	12	12	12
Maximum user I/O pins	822	1,022	1,238	1,334

Exemple de F.P.G.A : Le Stratix d'Altera



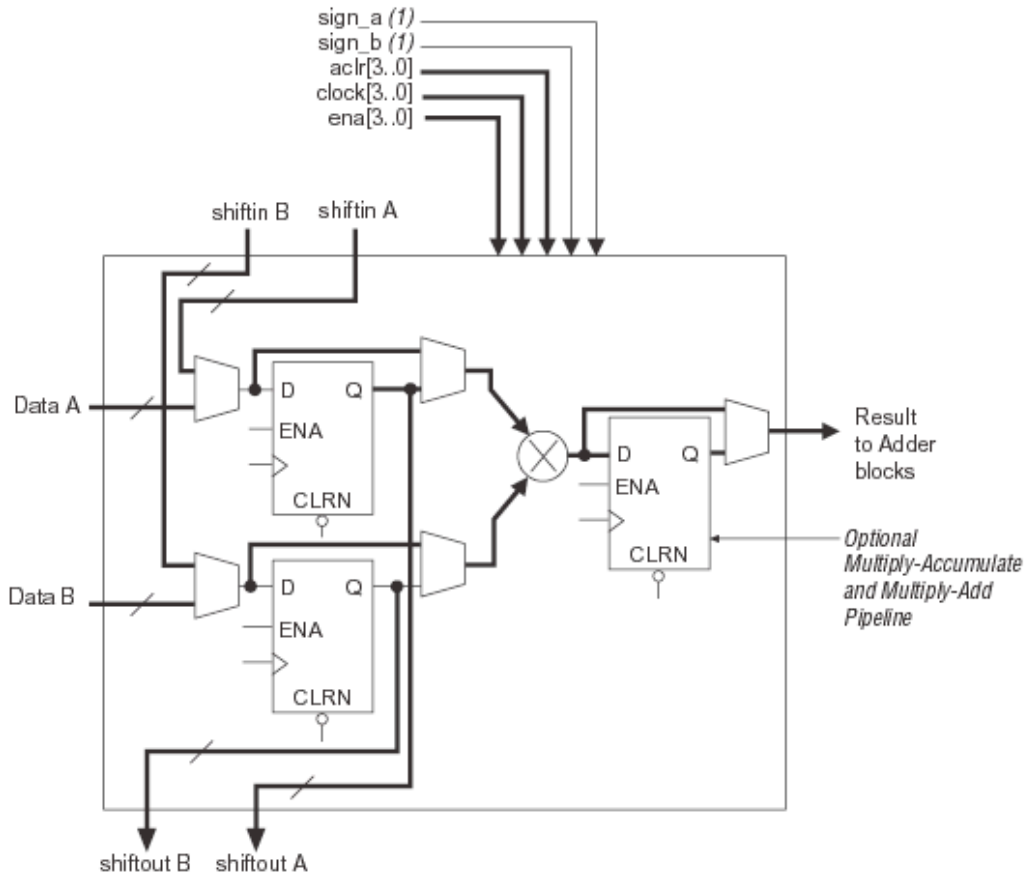
Exemple de F.P.G.A : Le Stratix d'Altera



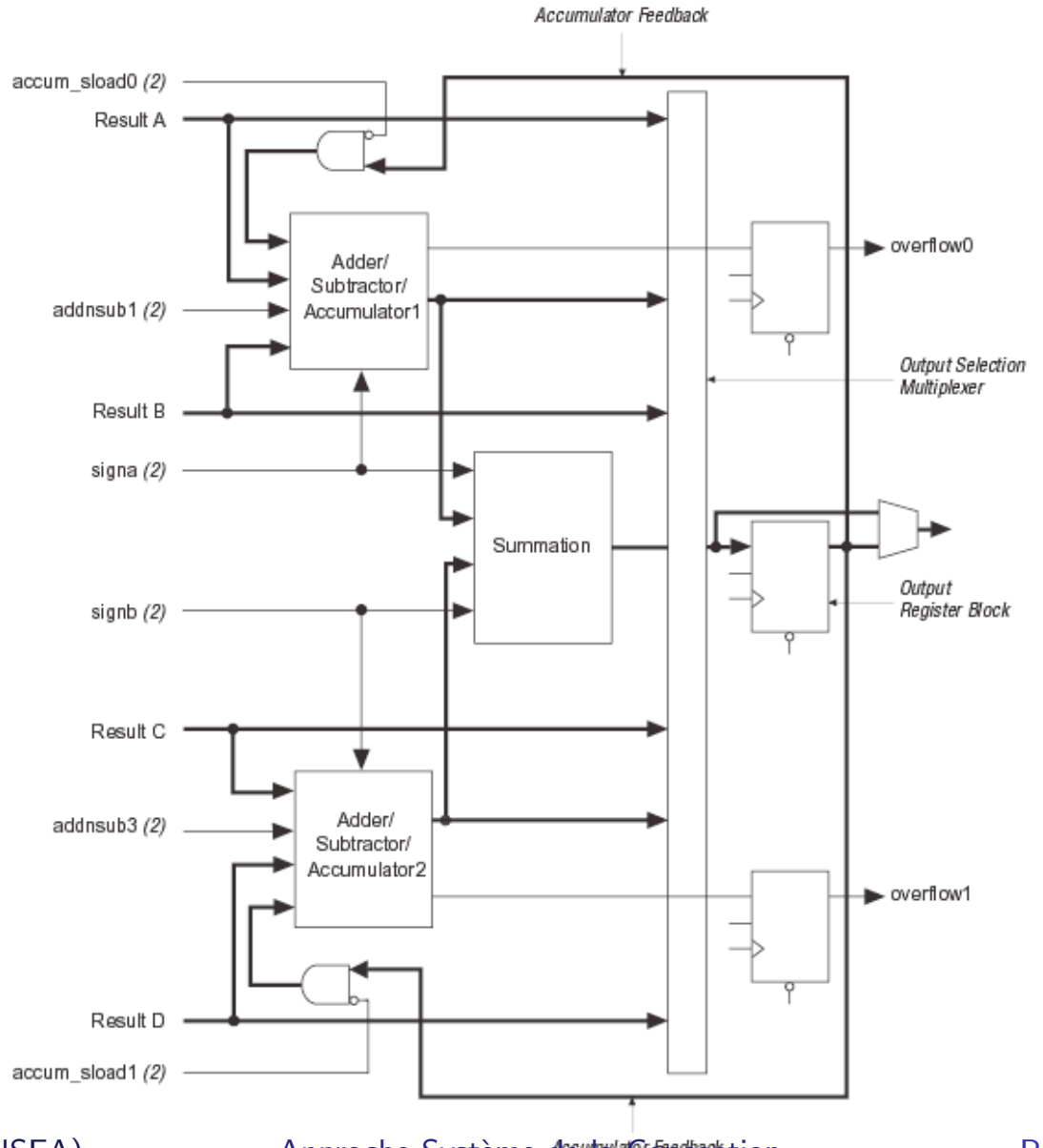
Exemple de F.P.G.A : Le Stratix d'Altera

Table 18. DSP Blocks in Stratix Devices *Note (1)*				
Device	**DSP Blocks**	**Total 9 × 9 Multipliers**	**Total 18 × 18 Multipliers**	**Total 36 × 36 Multipliers**
EP1S10	6	48	24	6
EP1S20	10	80	40	10
EP1S25	10	80	40	10
EP1S30	12	96	48	12
EP1S40	14	112	56	14
EP1S60	18	144	72	18
EP1S80	22	176	88	22
EP1S120	28	224	112	28

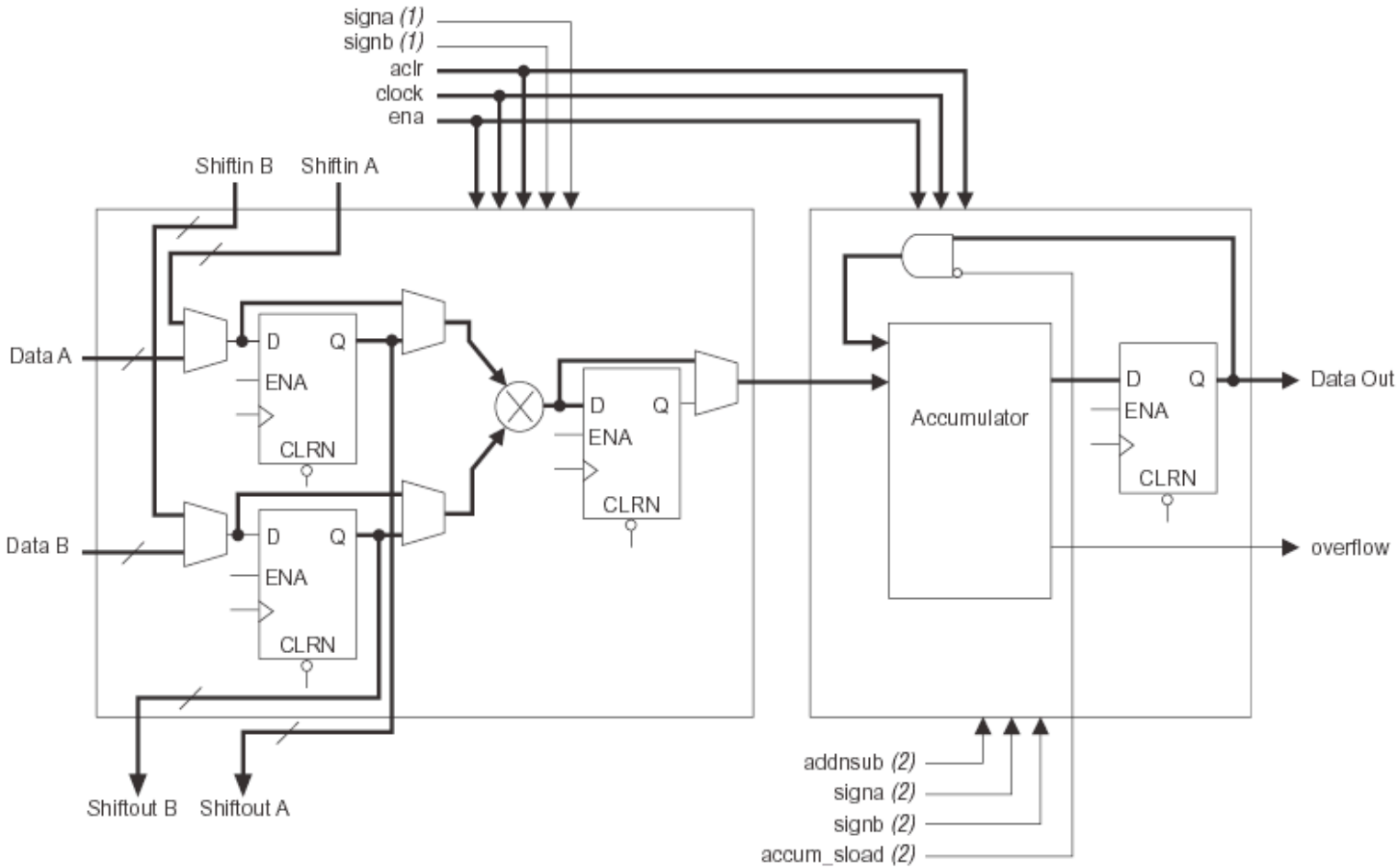
Exemple de F.P.G.A : Le Stratix d'Altera



Exemple de F.P.G.A : Le Stratix d'Altera



Exemple de F.P.G.A : Le Stratix d'Altera



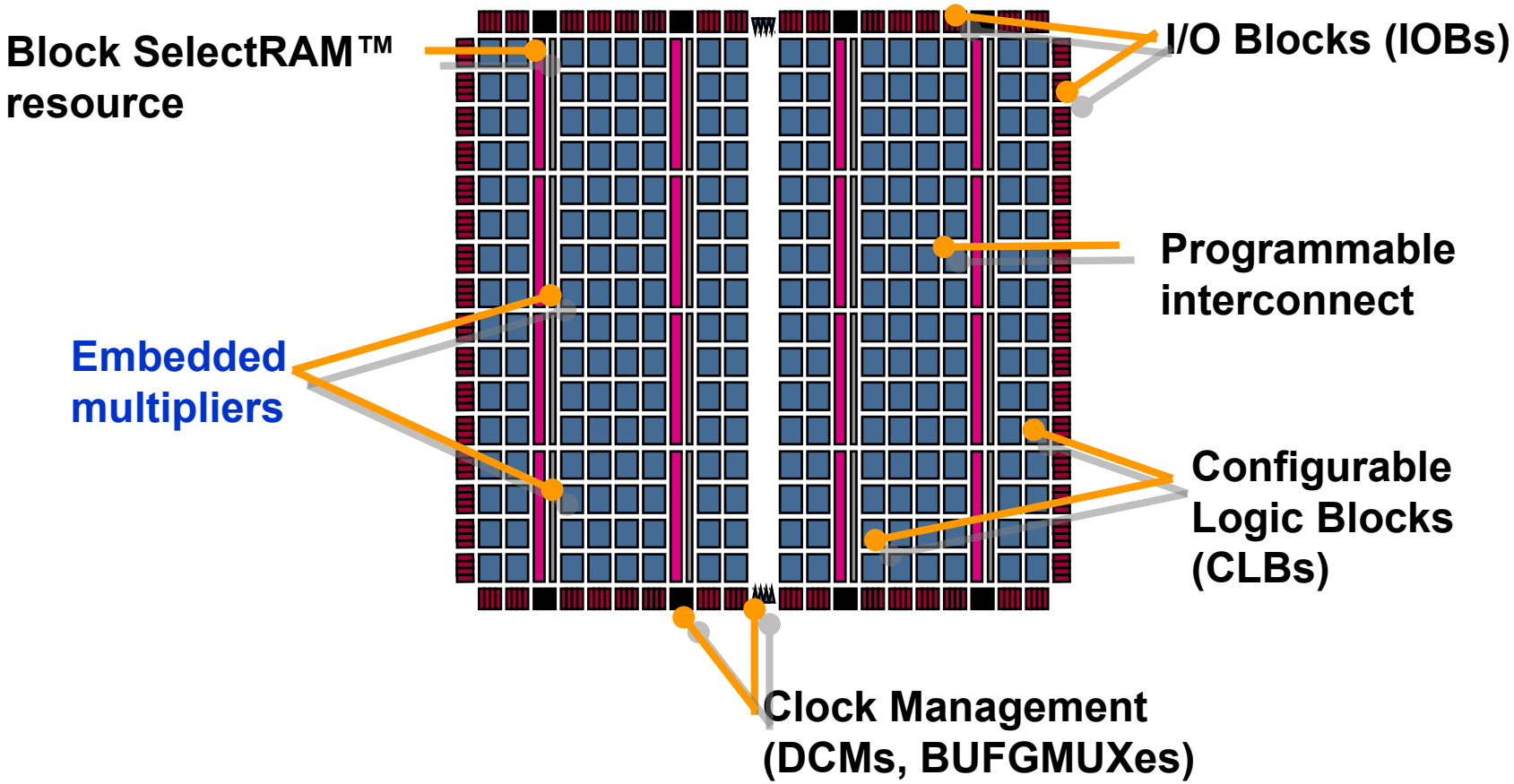
FPGA

Overview

- All Xilinx FPGAs contain the same basic resources
  - Slices grouped into Configurable Logic Blocks (CLBs)
    - Contain combinatorial logic and register resources
  - IOBs
    - Interface between the FPGA and the outside world
  - Programmable interconnect
  - Other resources
    - Memory
    - Multipliers
    - Global clock buffers
    - Boundary scan logic

Virtex-II Architecture

First family with Embedded Multipliers to enable high-performance DSP

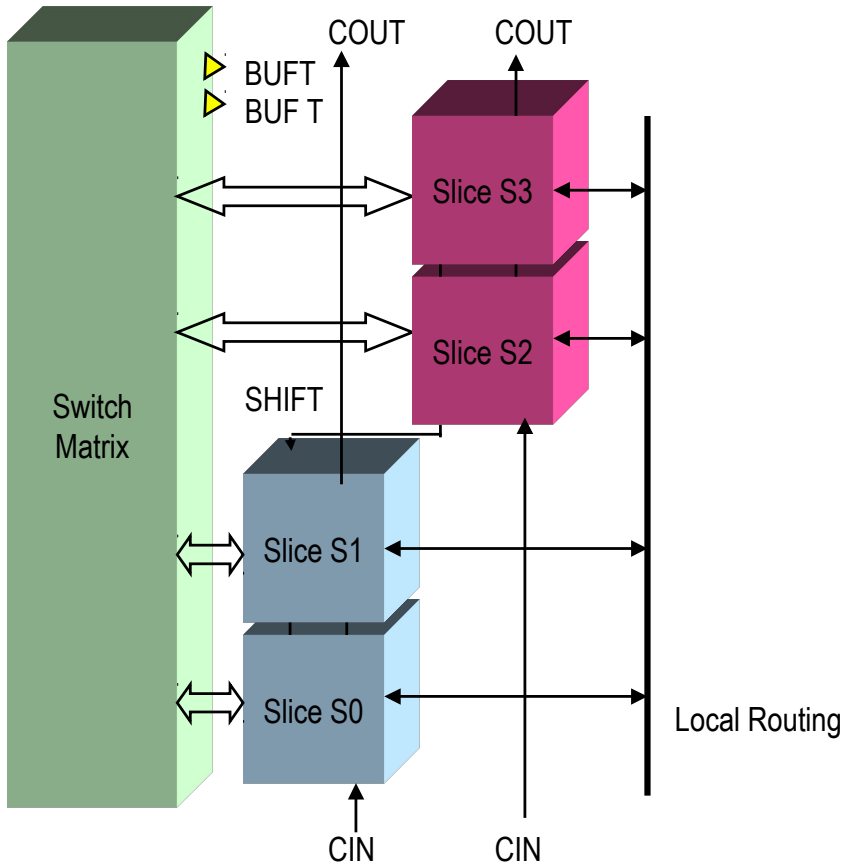


Refer to device data sheet at xilinx.com for detailed technical information

CLBs and Slices

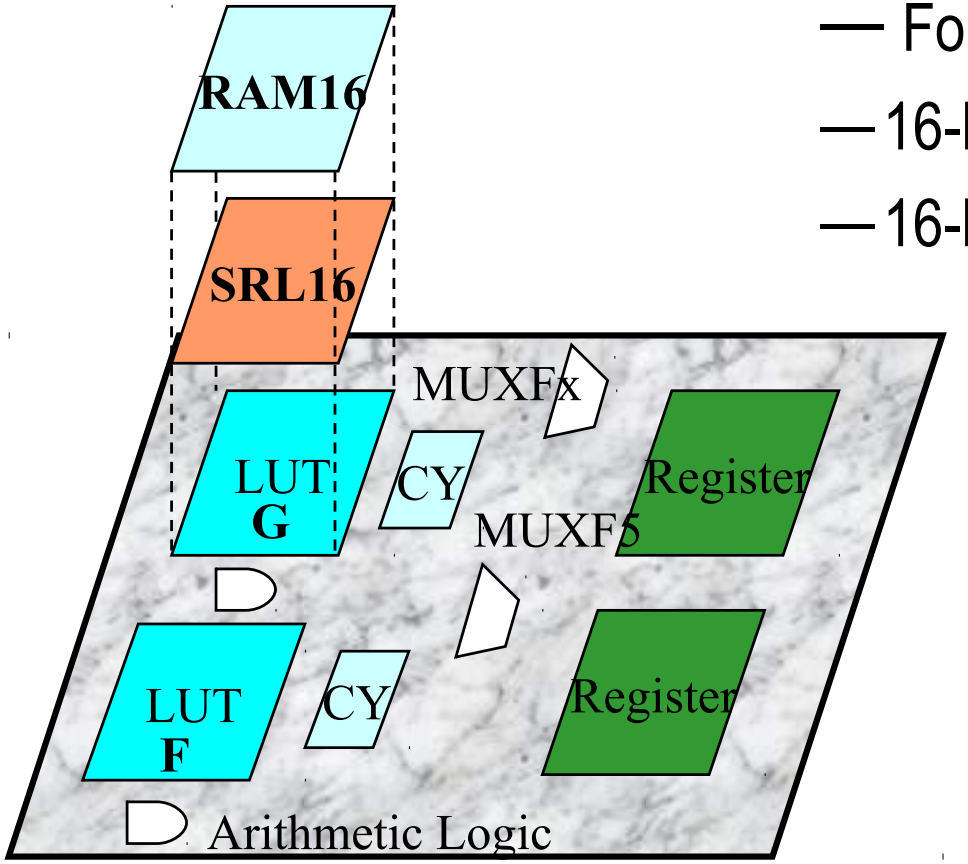
Combinatorial and sequential logic implemented here

- Each Virtex™-II CLB contains four slices
  - Local routing provides feedback between slices in the same CLB, and it provides routing to neighboring CLBs
  - A switch matrix provides access to general routing resources



Slice Resources

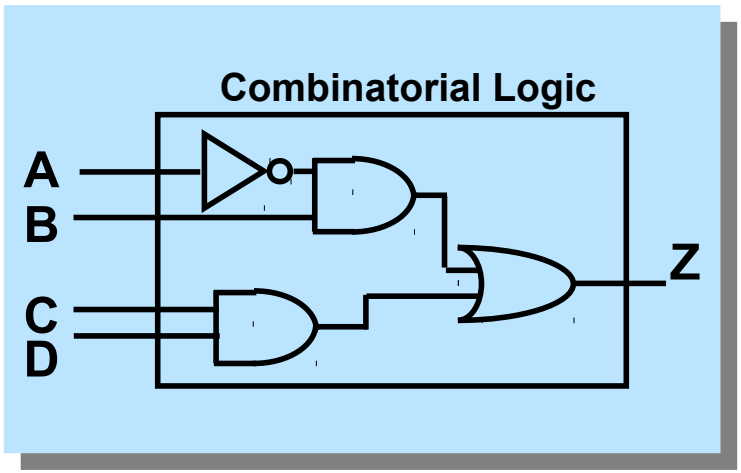
- **Each slice contains two:**
  - Four inputs lookup tables
  - 16-bit distributed SelectRAM
  - 16-bit shift register



- **Each register:**
  - D flip-flop
  - Latch
- **Dedicated logic:**
  - Muxes
  - Arithmetic logic
    - MULT_AND
    - Carry Chain

Look-Up Tables

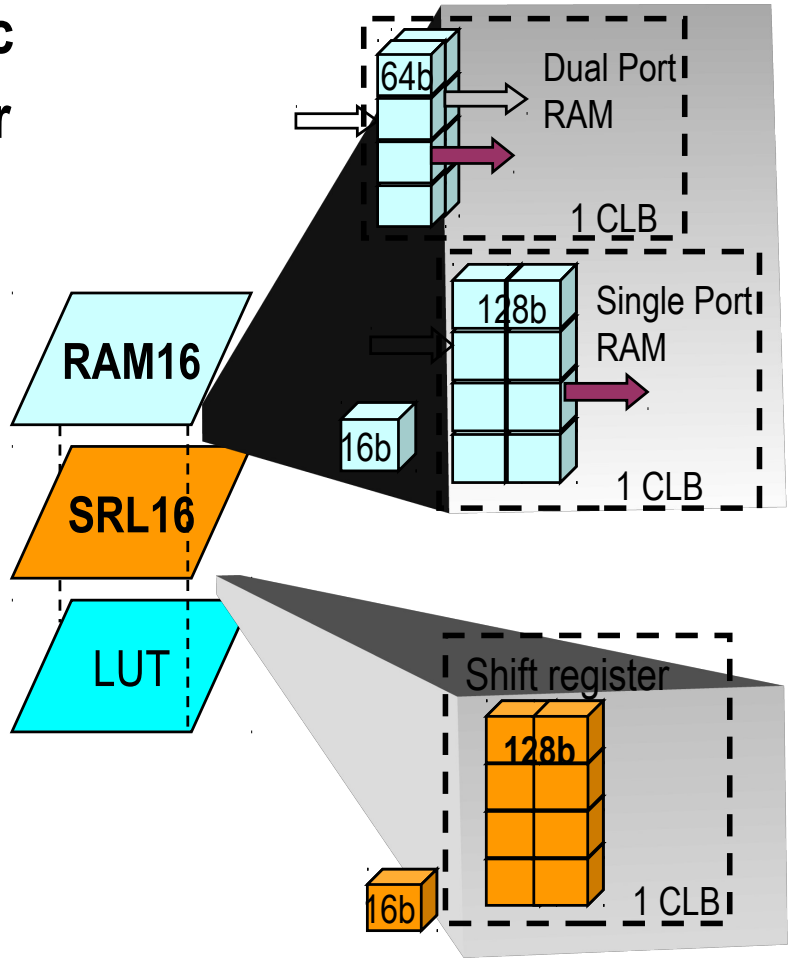
- Combinatorial logic is stored in Look-Up Tables (LUTs)
  - Also called Function Generators (FGs)
  - Capacity is limited by the number of inputs, not by the complexity
- Delay through the LUT is constant



A	B	C	D	Z
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
	.	.	.	
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

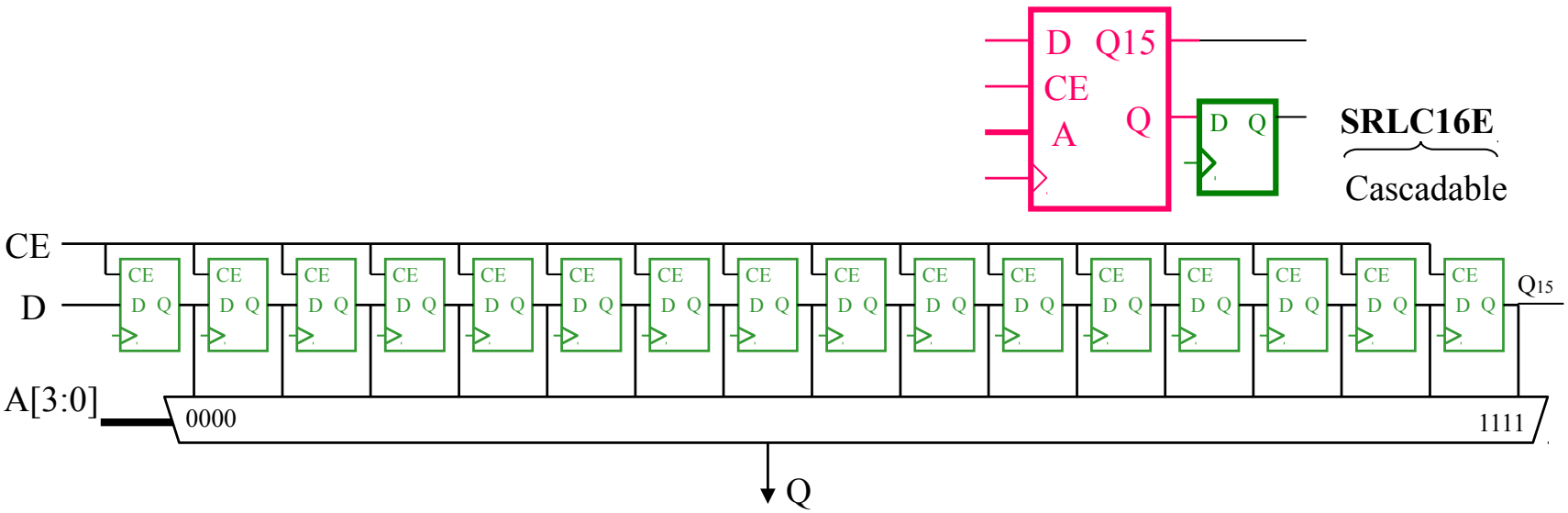
Distributed RAM

- LUTs used as memory inside the fabric
- Flexible, can be used as RAM, ROM, or shift register
- Distributed memory with fast access time
- Cascadable with built-in CLB routing
- Applications
  - Linear feedback shift register
  - Distributed arithmetic
  - Time-shared registers
  - Small FIFO
  - Digital delay lines (Z^{-1})



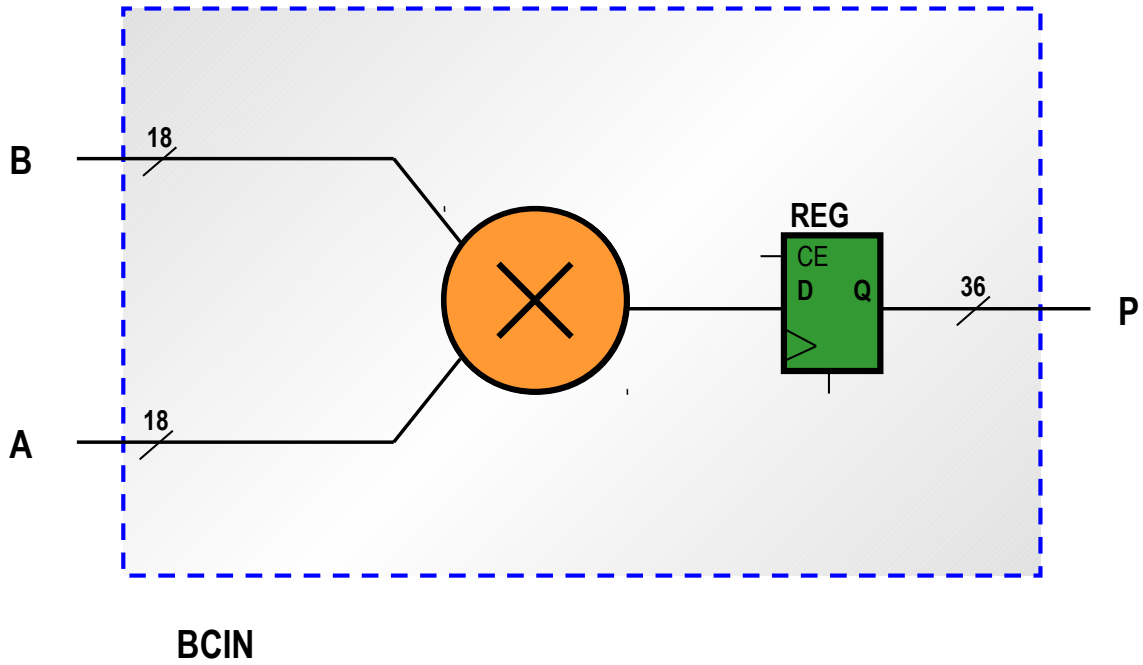
The SRL16E

- **The 16 SRAM cells have been organized into a shift register**
  - The ‘CE’ is used, in conjunction with the clock, to write data into the first flip-flop and for all other data to move right by one position
  - Because this is a predictable operation, no address is required for writing
- **The SRL16E is excellent in implementing efficient DSP Functions**
  - A very efficient way to delay data samples
  - Shifting samples and scanning at faster rate



Enabling high-performance DSP

Virtex-II introduced the embedded 18x18 multiplier



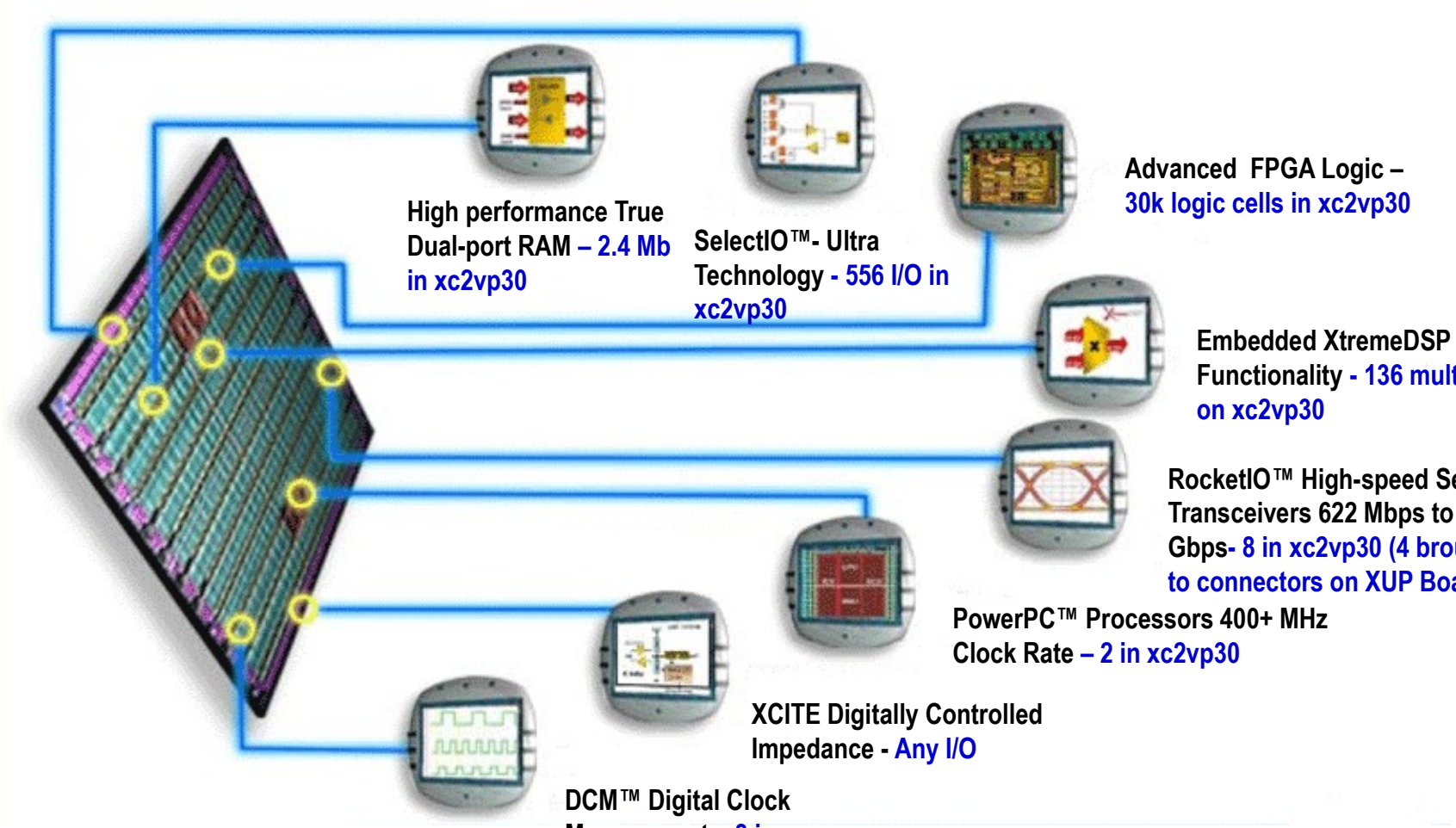
- Situated between the Block RAMs and CLB array to enable high-performance multiply-accumulate operations
- This dramatically increased multiplier speed and density compared to LUT based multipliers and enabled FPGA based DSP

Outline

- Power of Parallelism
- Basic FPGA Architecture
- **Virtex™ -II Pro**
- Virtex-4
- Virtex-5
- Spartan™-3 Family
- Latest Families
  - Virtex-6 Family
  - Spartan-6 Family
- Why should I use FPGAs for DSP?
- The DSP48 Slice Advantage

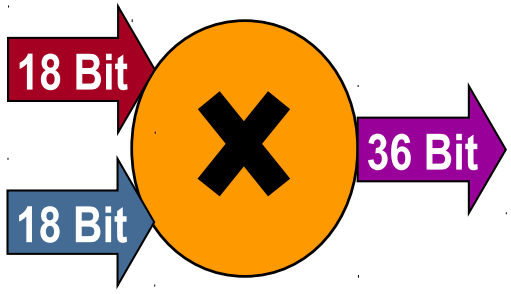
Virtex-II Pro FPGAs

Refer to device data sheet on web for detailed technical information



Multiplier Unit

- Embedded 18-bit x 18-bit multiplier
- The XUP Virtex-II Pro includes a Virtex-II Pro xc2vp30 device with 136 Multipliers
- 2s complement signed operation
- 4- to 18-bit operands
- Combinational & pipelined options
- Operates with block RAM and fabric to implement MAC function

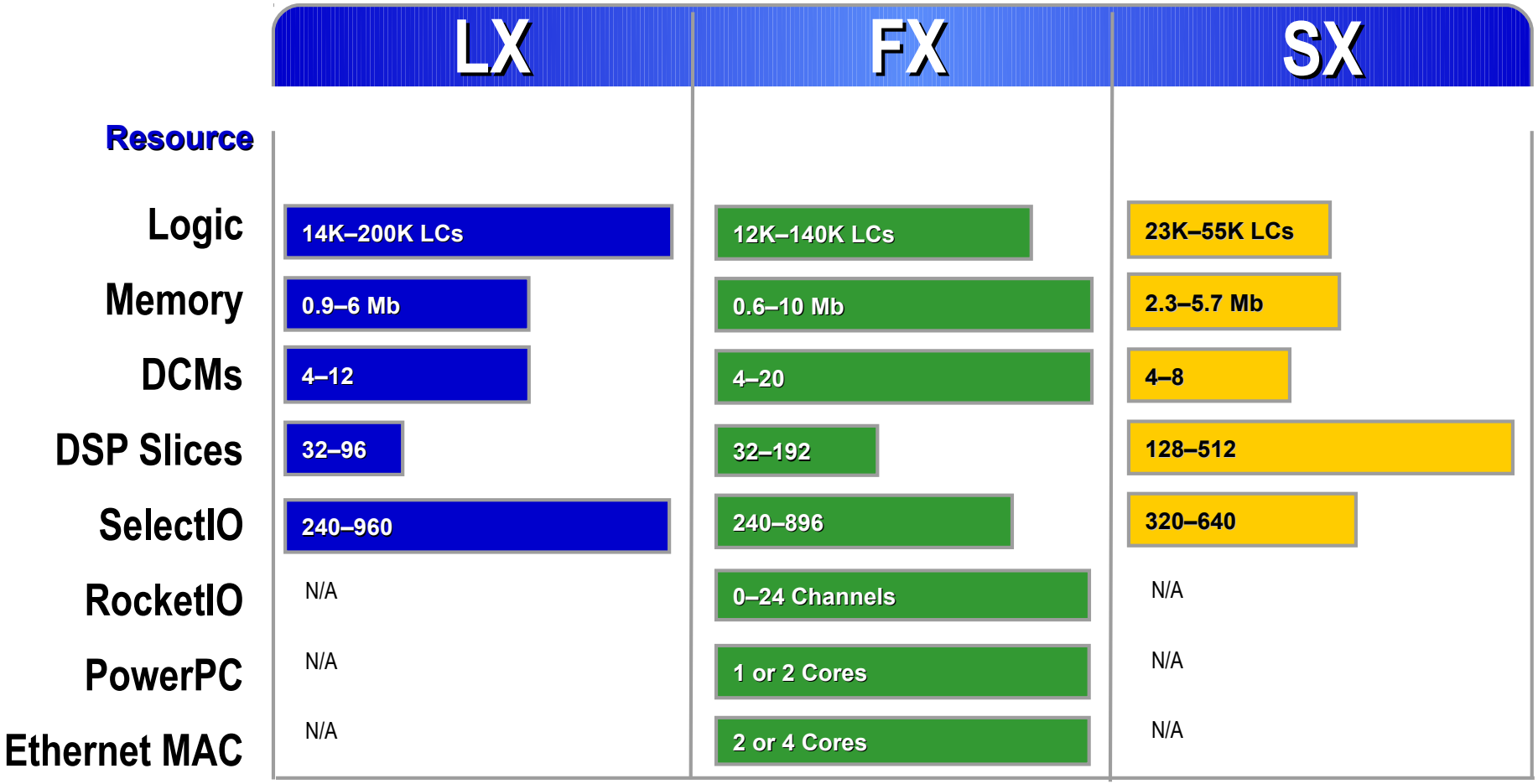


300 MHz Performance in Virtex-II Pro

Pipelined multiplier with registered inputs and outputs

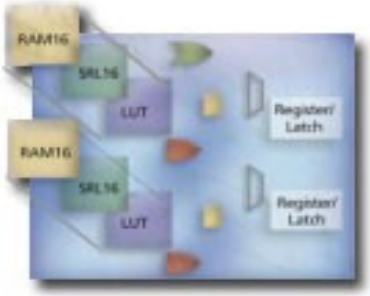
Virtex-4 Family

Advanced Silicon Modular BLock (ASMBL) Architecture
Optimized for logic, Embedded, and Signal Processing



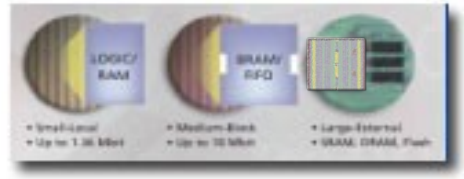
Virtex-4 Features

Refer to device data sheet on web for detailed technical information



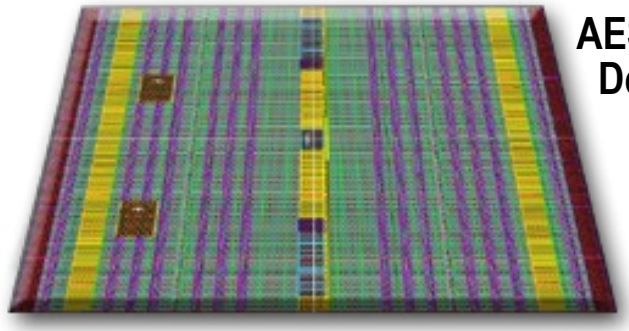
200,000
Logic Cells

500 MHz Xesium™
Differential Clocking



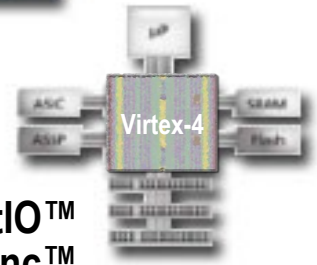
BRAM with FIFO
& ECC

0.6-6.5 Gbps
RocketIO™
Transceivers

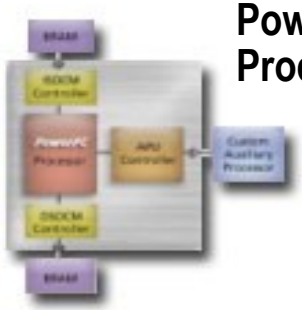


Virtex-4

AES Secure Chip
Design Security



1 Gbps SelectIO™
with ChipSync™

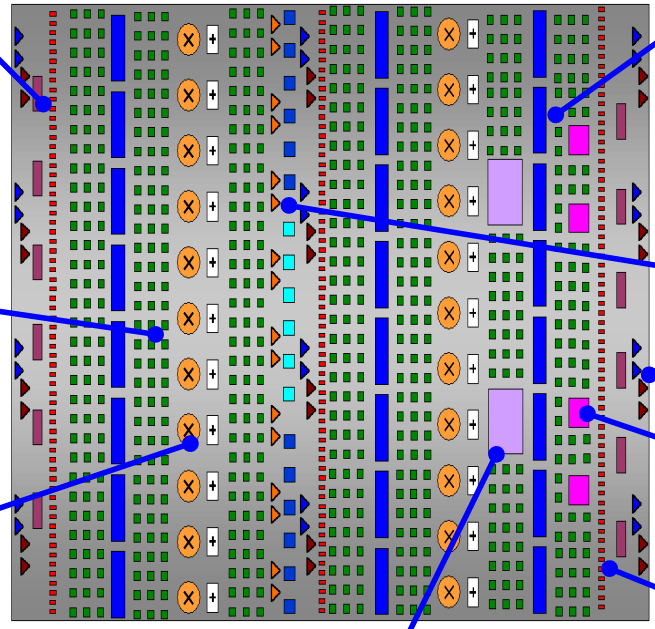


PowerPC®
Processor with APU

10/100/1000
Ethernet MAC

XtremeDSP™
Slice

Virtex-4 Architecture



RocketIO™ Multi-Gigabit Transceivers
622 Mbps–10.3 Gbps

Advanced CLBs
200K Logic Cells

XtremeDSP™ Technology Slices
256 18x18 GMACs

PowerPC™ 405 with APU Interface
450 MHz, 680 DMIPS

Smart RAM
New block RAM/FIFO

Xesium Clocking Technology
500 MHz

Tri-Mode Ethernet MAC
10/100/1000 Mbps

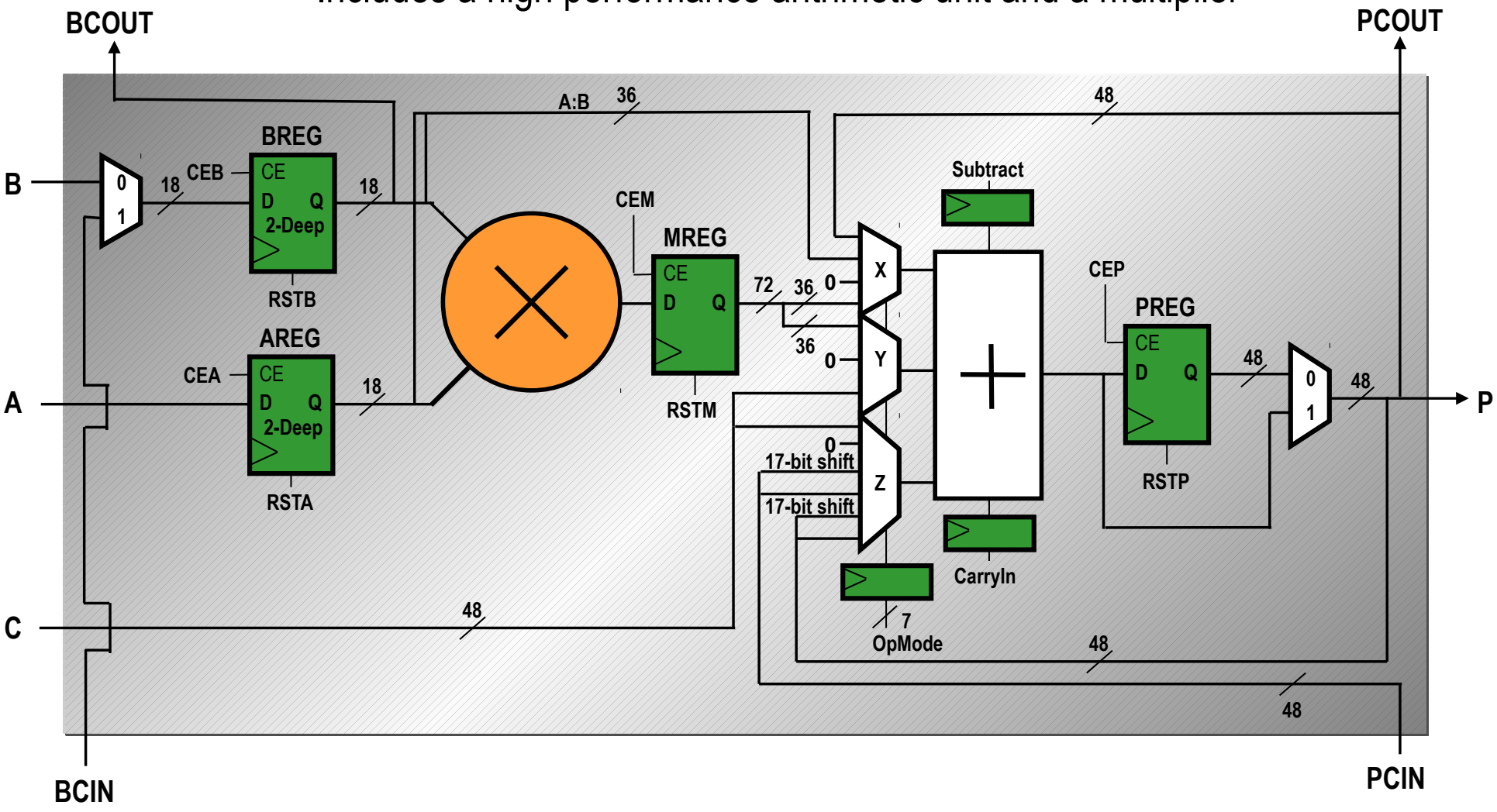
1 Gbps SelectIO™
ChipSync™ Source synch, XCITE Active Termination

The Virtex-4 SX platform

- Virtex-4 introduced a new DSP block that had both multiply and accumulate functionality
- For the first time a true “MAC” unit was offered in a Xilinx FPGA. This block was called the DSP48 due to it’s 48-bit output precision
- Additional modes of the adder allowed subtract and shift functions to support scaling of results
- Integral registers guarantee high-speed pipelined data-paths for maximum clock frequency

DSP48 Block

Includes a high performance arithmetic unit and a multiplier



DSP48 Block

Dynamically Programmable DSP Op Modes

OpMode	Z			Y		X		Output
	6	5	4	3	2	1	0	
Zero	0	0	0	0	0	0	0	+/- Cin
Hold P	0	0	0	0	0	1	0	+/- (P + Cin)
A:B Select	0	0	0	0	0	1	1	+/- (A:B + Cin)
Multiply	0	0	0	0	1	0	1	+/- (A * B + Cin)
C Select	0	0	0	1	1	0	0	+/- (C + Cin)
Feedback Add	0	0	0	1	1	1	0	+/- (C + P + Cin)
36-Bit Adder	0	0	0	1	1	1	1	+/- (A:B + C + Cin)
P Cascade Select	0	0	1	0	0	0	0	PCIN +/- Cin
P Cascade Feedback Add	0	0	1	0	0	1	0	PCIN +/- (P + Cin)
P Cascade Add	0	0	1	0	0	1	1	PCIN +/- (A:B + Cin)
P Cascade Multiply Add	0	0	1	0	1	0	1	PCIN +/- (A * B + Cin)
P Cascade Add	0	0	1	1	1	0	0	PCIN +/- (C + Cin)
P Cascade Feedback Add Ad	0	0	1	1	1	1	0	PCIN +/- (C + P + Cin)
P Cascade Add Add	0	0	1	1	1	1	1	PCIN +/- (A:B + C + Cin)
Hold P	0	1	0	0	0	0	0	P +/- Cin
Double Feedback Add	0	1	0	0	0	1	0	P +/- (P + Cin)
Feedback Add	0	1	0	0	0	1	1	P +/- (A:B + Cin)
Multiply-Accumulate	0	1	0	0	1	0	1	P +/- (A * B + Cin)
Feedback Add	0	1	0	1	1	0	0	P +/- (C + Cin)
Double Feedback Add	0	1	0	1	1	1	0	P +/- (C + P + Cin)
Feedback Add Add	0	1	0	1	1	1	1	P +/- (A:B + C + Cin)
C Select	0	1	1	0	0	0	0	C +/- Cin
Feedback Add	0	1	1	0	0	1	0	C +/- (P + Cin)
36-Bit Adder	0	1	1	0	0	1	1	C +/- (A:B + Cin)
Multiply-Add	0	1	1	0	1	0	1	C +/- (A * B + Cin)
Double	0	1	1	1	1	0	0	C +/- (C + Cin)
Double Add Feedback Add	0	1	1	1	1	1	0	C +/- (C + P + Cin)
Double Add	0	1	1	1	1	1	1	C +/- (A:B + C + Cin)

- Enables time-division multiplexing for DSP
- Over 40 different modes
- Each XtremeDSP Slice individually controllable
- Change operation in a single clock cycle
- Control functionality from logic, memory or processor

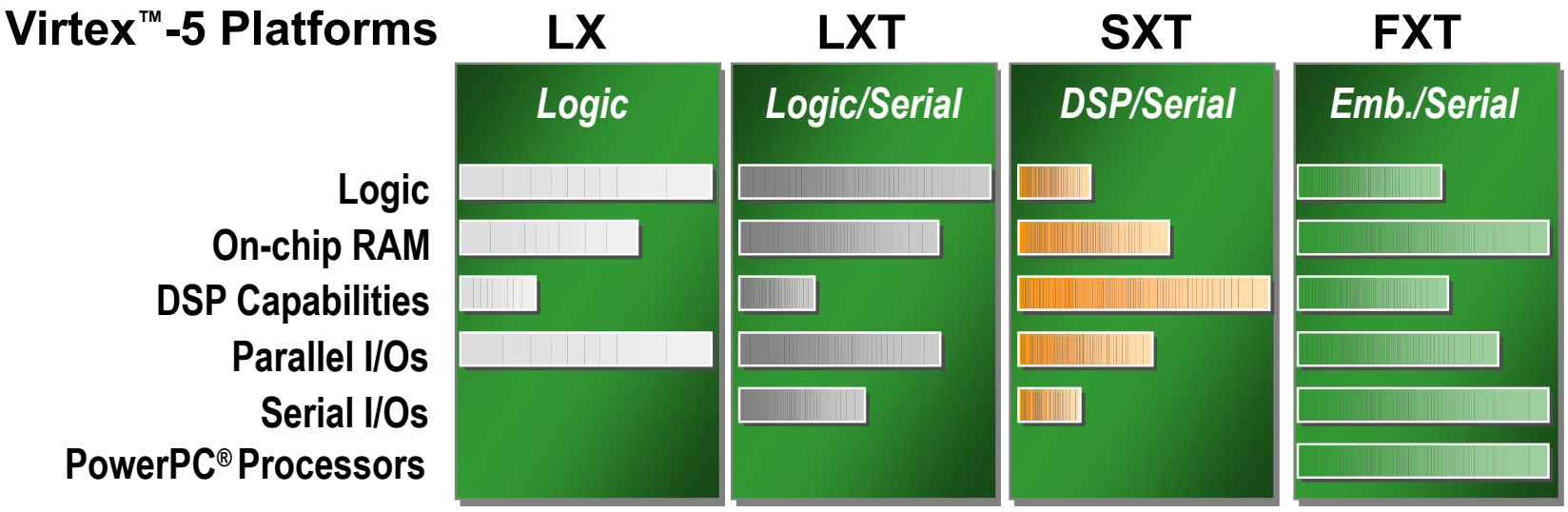
DSP48 Block

Useful For More Than DSP

- 6:1 high-speed, 36-bit Multiplexer
  - Use four XtremeDSP Slice and op-modes
  - 500 MHz performance using no programmable logic
    - Save 1584 LCs to build equivalent function in logic
- Dynamic 18-bit Barrel Shifter
  - Use two XtremeDSP slices
  - Use dedicated cascade routing and integrated 17-bit shift
    - Save 1449 LCs to build equivalent function in logic
- 36-bit Loadable Counter
  - Use a single XtremeDSP slice, achieve 500 MHz performance
    - Save 540 LCs to build equivalent function in logic

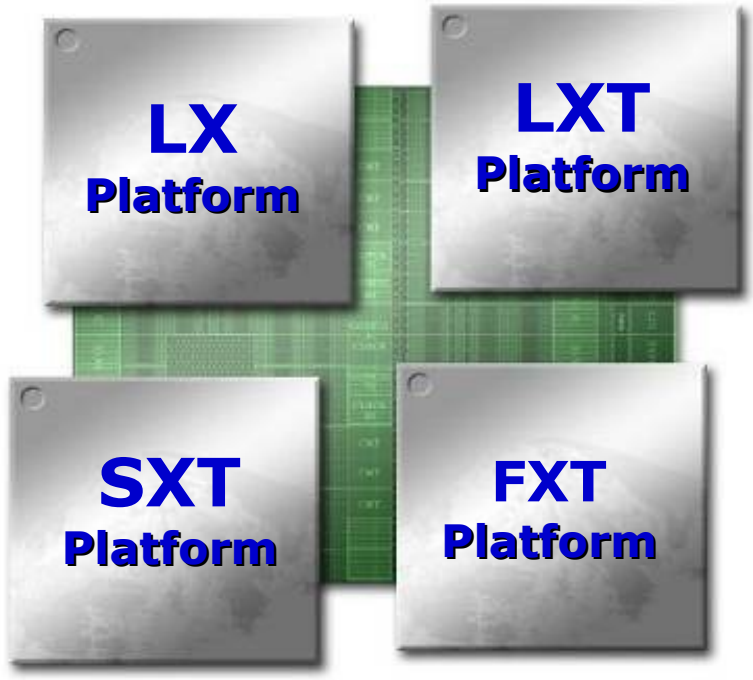
Virtex-5 Family

Optimized for logic, Embedded, Signal Processing, and High-Speed Connectivity



Multiple Platforms

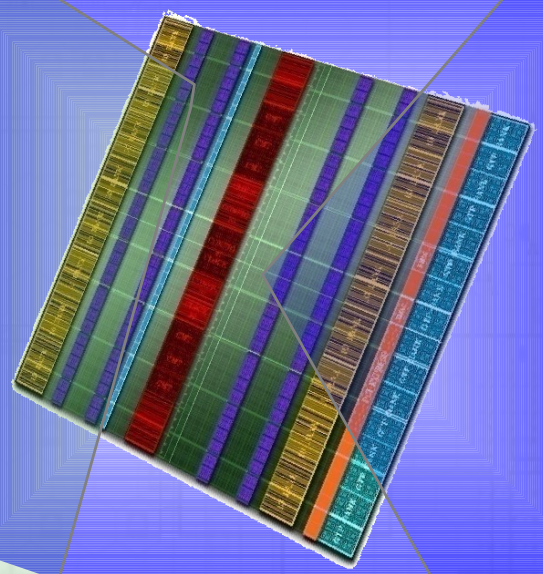
- Easy to create sub-families
  - LX : High-performance logic and parallel IO
  - LXT: High-performance logic with serial connectivity
  - SXT: Extensive signal processing with serial connectivity
  - FXT: Extensive processor oriented
    - Embedded-oriented with Highest Performance Serial Capabilities
- Users can choose the best mix of resources to optimize cost and performance



Virtex-5 Architecture

Enhanced

- 36Kbit Dual-Port Block RAM / FIFO with Integrated ECC
- 550 MHz Clock Management Tile with DCM and PLL
- SelectIO with ChipSync Technology and XCITE DCI
- Advanced Configuration Options
- 25x18 DSP Slice with Integrated ALU
- Tri-Mode 10/100/1000 Mbps Ethernet MACs

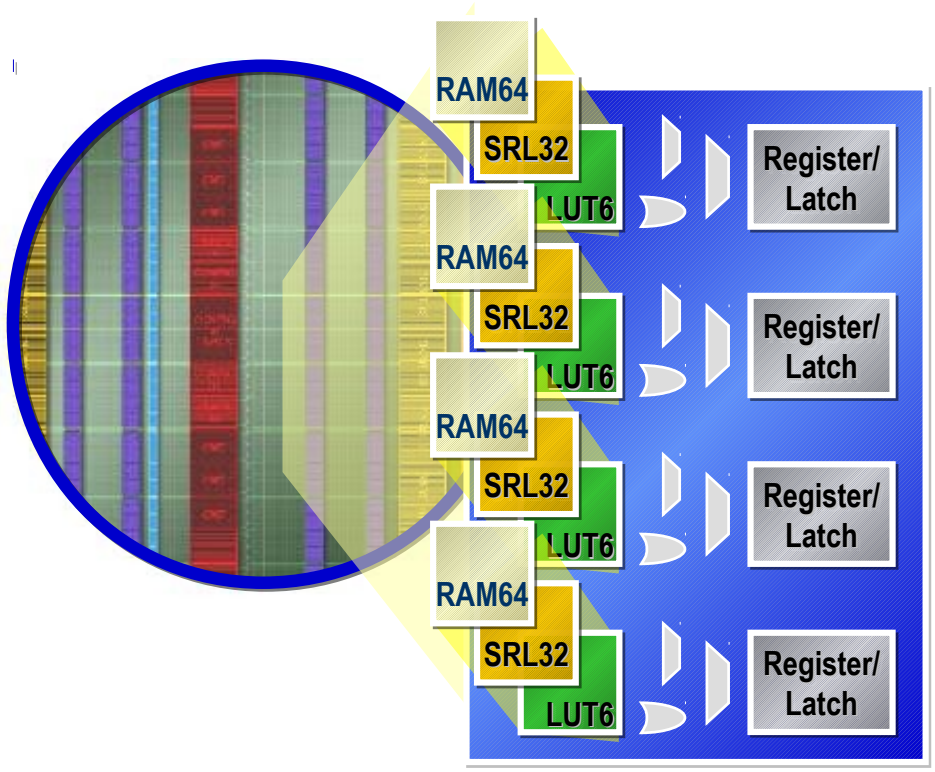


New

- Most Advanced High-Performance Real 6LUT Logic Fabric
- PCI Express® Endpoint Block
- System Monitor Function with Built-in ADC
- Next Generation PowerPC® Embedded Processor
- RocketIO™ Transceiver Options
  - Low-Power GTP: Up to 3.75 Gbps
  - High-Performance GTX: Up to 6.5 Gbps

Advanced Logic Structure

- True 6-input LUTs
- Exclusive 64-bit distributed RAM option per LUT
- Exclusive 32-bit or 16-bit x 2 shift register

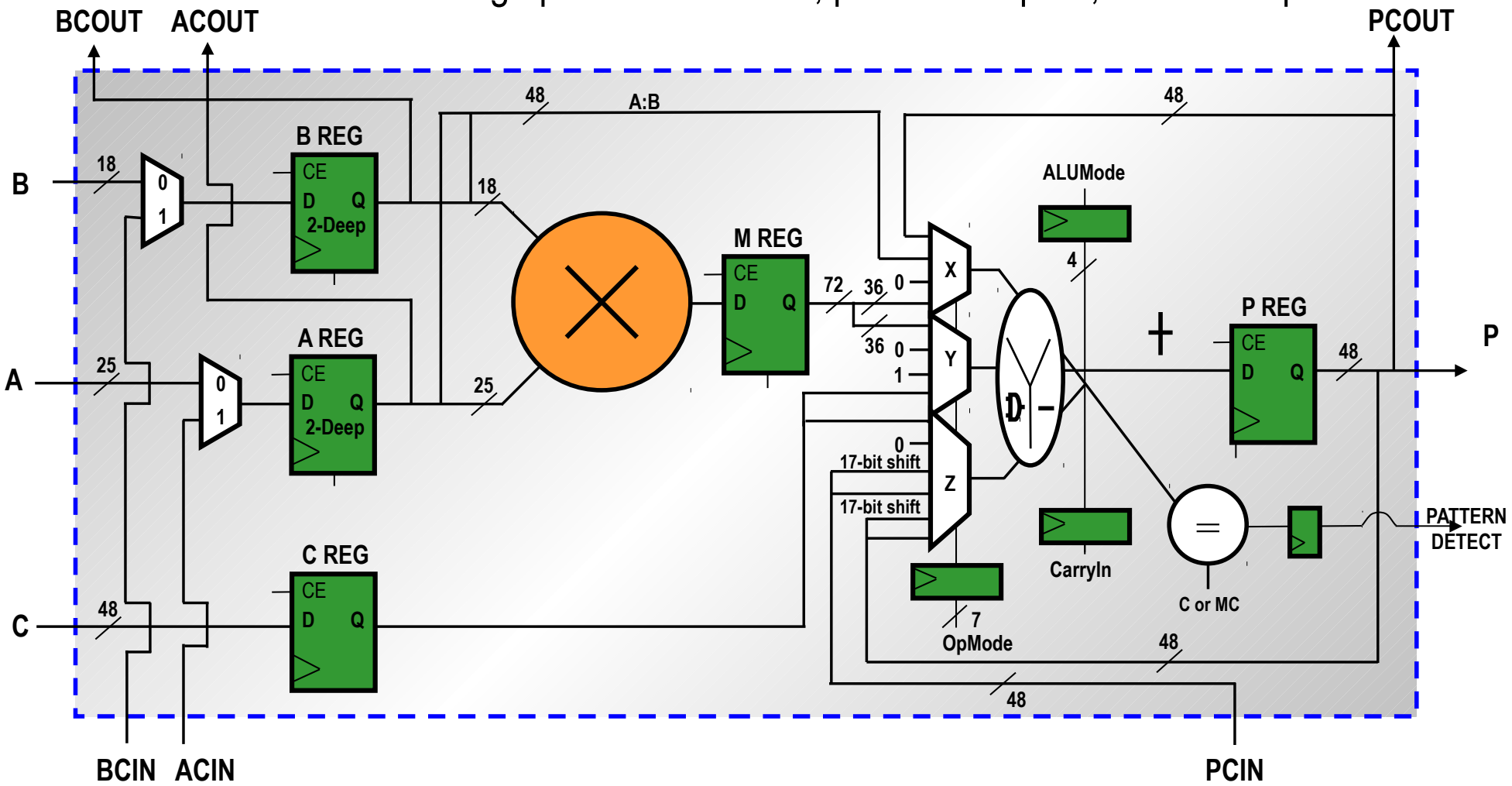


DSP48E

- Virtex-5SX introduced a few new improvements in the DSP48E “enhanced” DSP block
- The adder block was modified to become a mulifunctional ALU. A pattern compare was added to support the detection of saturation, overflow and underflow conditions
- A 48-bit carry chain supports the propagation of partial sum and product carry’s so multiple DSP48E blocks can be chained to give higher bit precision
- ALU opcodes are dynamically controlled allowing functional changes on a clock cycle basis

DSP48E Block

Includes a high performance ALU, pattern compare, and a multiplier

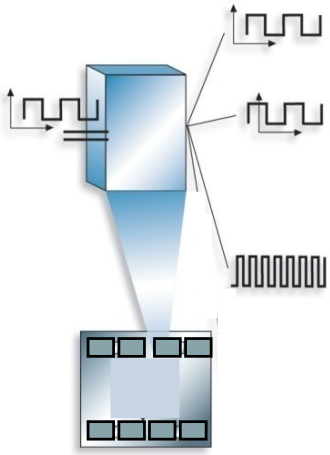


450 MHz operation in the slowest speed grade

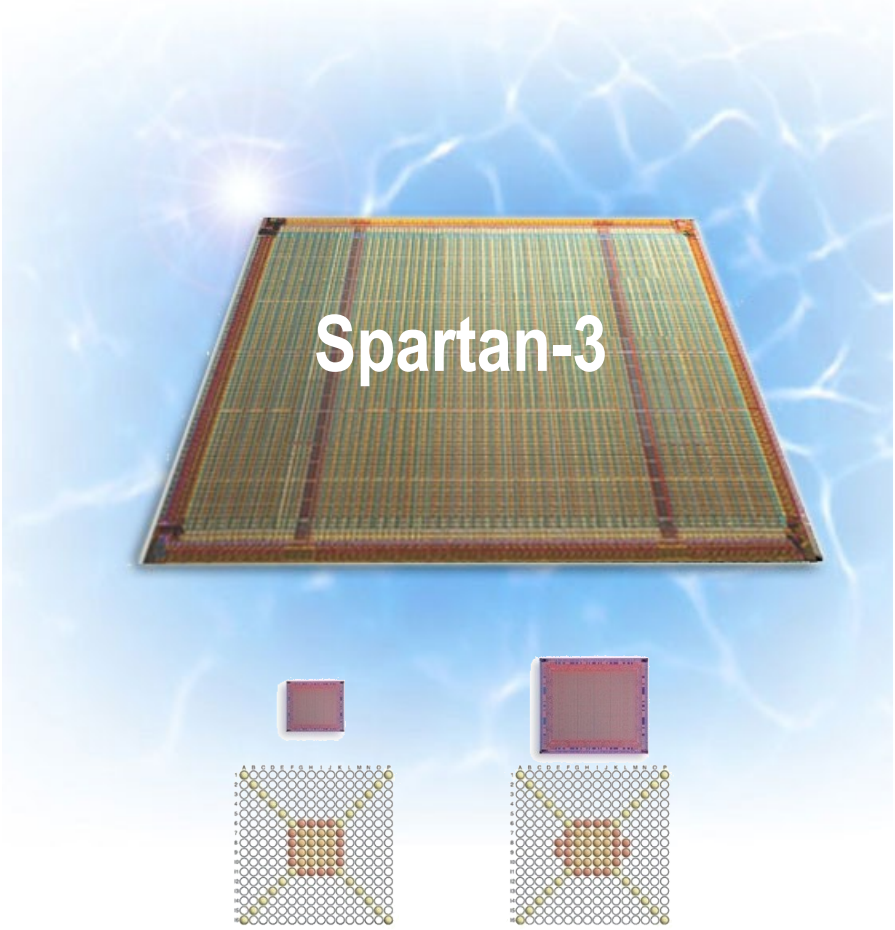
Spartan-3

Designed for low-cost, high-volume applications

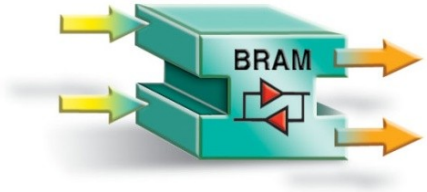
18x18 bit Embedded Pipelined Multipliers for efficient DSP



Up to eight on-chip Digital Clock Managers to support multiple system clocks



Guaranteed Density Migration
Numerous parts in the same package



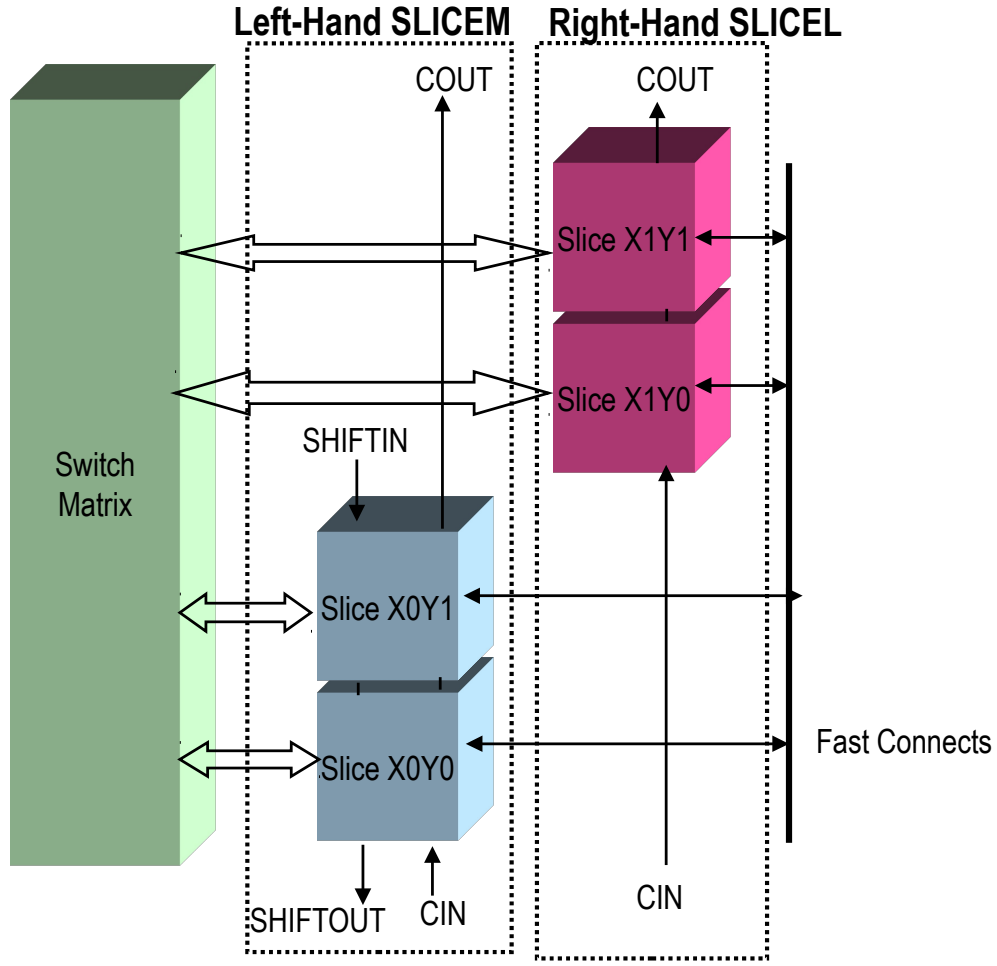
4 I/O Banks, Support for all I/O Standards including PCI™, DDR333, RSDS, mini-LVDS

PCI, PCIe, PCI-X and PCI EXPRESS are registered trademarks and/or service marks of PCI-SIG.

Modified Slices

SLICEM and SLICEL

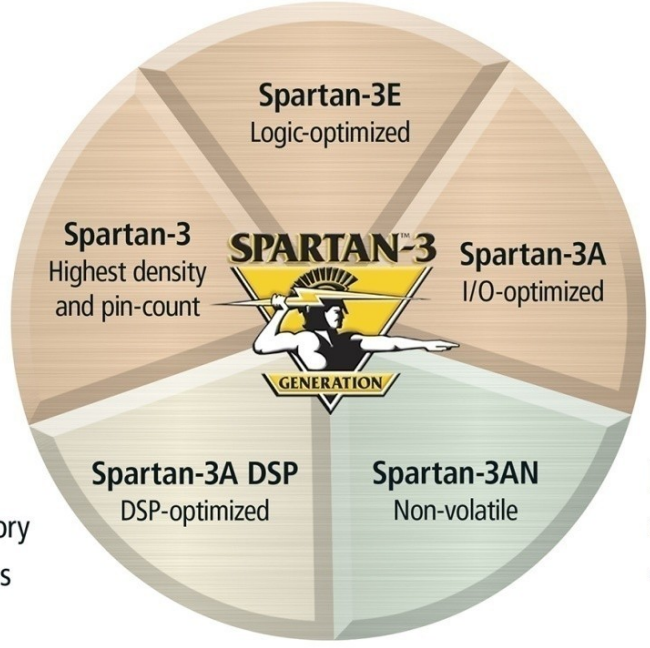
- Each Spartan™-3 CLB contains four slices
  - Similar to Virtex™-II device
- Slices are grouped in pairs
  - Left-hand SLICEM (Memory)
    - LUTs can be configured as memory or SRL16
  - Right-hand SLICEL (Logic)
    - LUT can be used as logic only



Multiple Domain-optimized Platforms

Mainstream

- Broad range of densities, general functionality and targeted specific application solutions
- Lower total system cost while increasing functionality



DSP

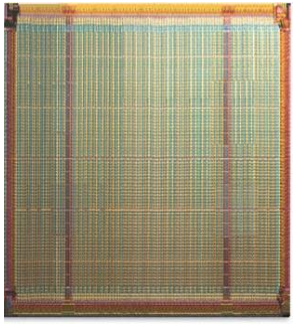
- Integrated DSP MACs and expanded memory
- Optimized for signal processing applications

Non-Volatile

- Combines leading-edge technology FPGAs & Flash technologies
- New evolution in security, protection and functionality

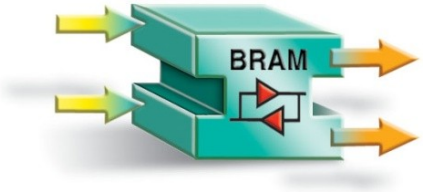
Spartan-3A

Spartan-3A DSP is a superset of Spartan-3A

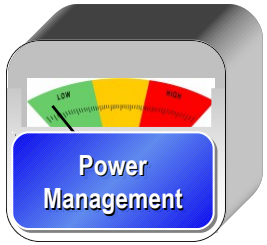


True 3.3v
PPDS_25
PPDS_33
TMDS_33
DDR 2
Hot Swapping
Minimized Power Rails

- Power Management
  - Hibernate and Suspend modes
- Minimized power rails
- New I/O Standards
- BRAM with Byte write enable
- SPI/BPI Flash Interface
- Hot swapping



18K Block RAMs
Byte Write Enable
Improved Bus Access



Hibernate Mode
Suspend Mode

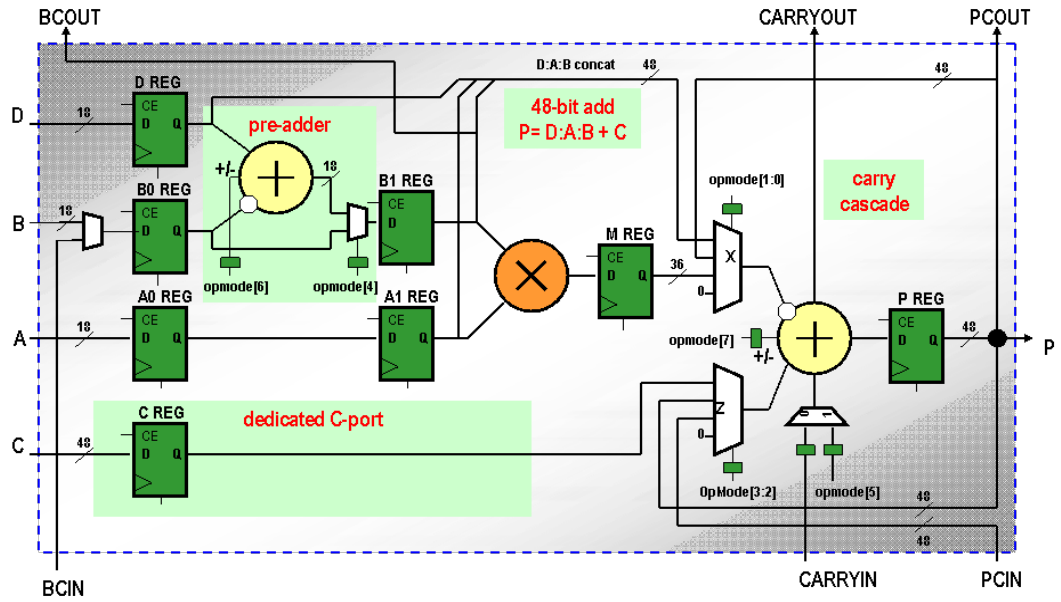
Spartan-3A DSP

- Incorporates the primary features from earlier Virtex family DSP48 blocks
- The DSP48A block supports full MAC support with a pre-adder stage, multiplier, and add/accumulate state
- Dedicated DSP block offer the lowest cost/MAC in a FPGA

DSP48A Block

Incorporates primary features from V4 DSP48 and includes a pre-adder stage

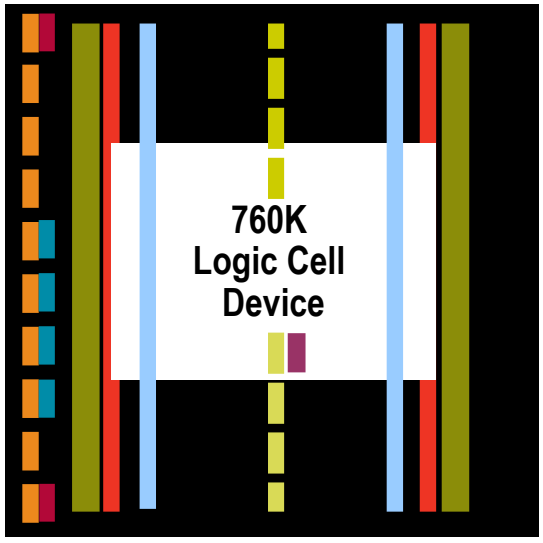
- Integrated XtremeDSP Slice
  - Application optimized capacity
    - 3400A – 126 DSP48As
    - 1800A – 84 DSP48As
  - Integrated pre-adder optimized for filters
  - 250 MHz operation, standard speed grade
  - Compatible with Virtex-DSP



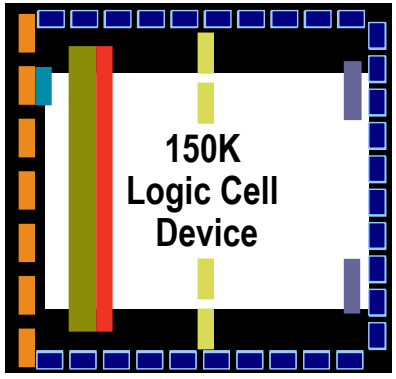
- Increased memory capacity and performance
  - Also important for embedded processing, complex IP, etc

Architecture Alignment

Virtex-6 FPGAs



Spartan-6 FPGAs



Common Resources

- LUT-6 CLB
- BlockRAM
- DSP Slices
- High-performance Clocking
- Parallel I/O
- HSS Transceivers*
- PCIe® Interface

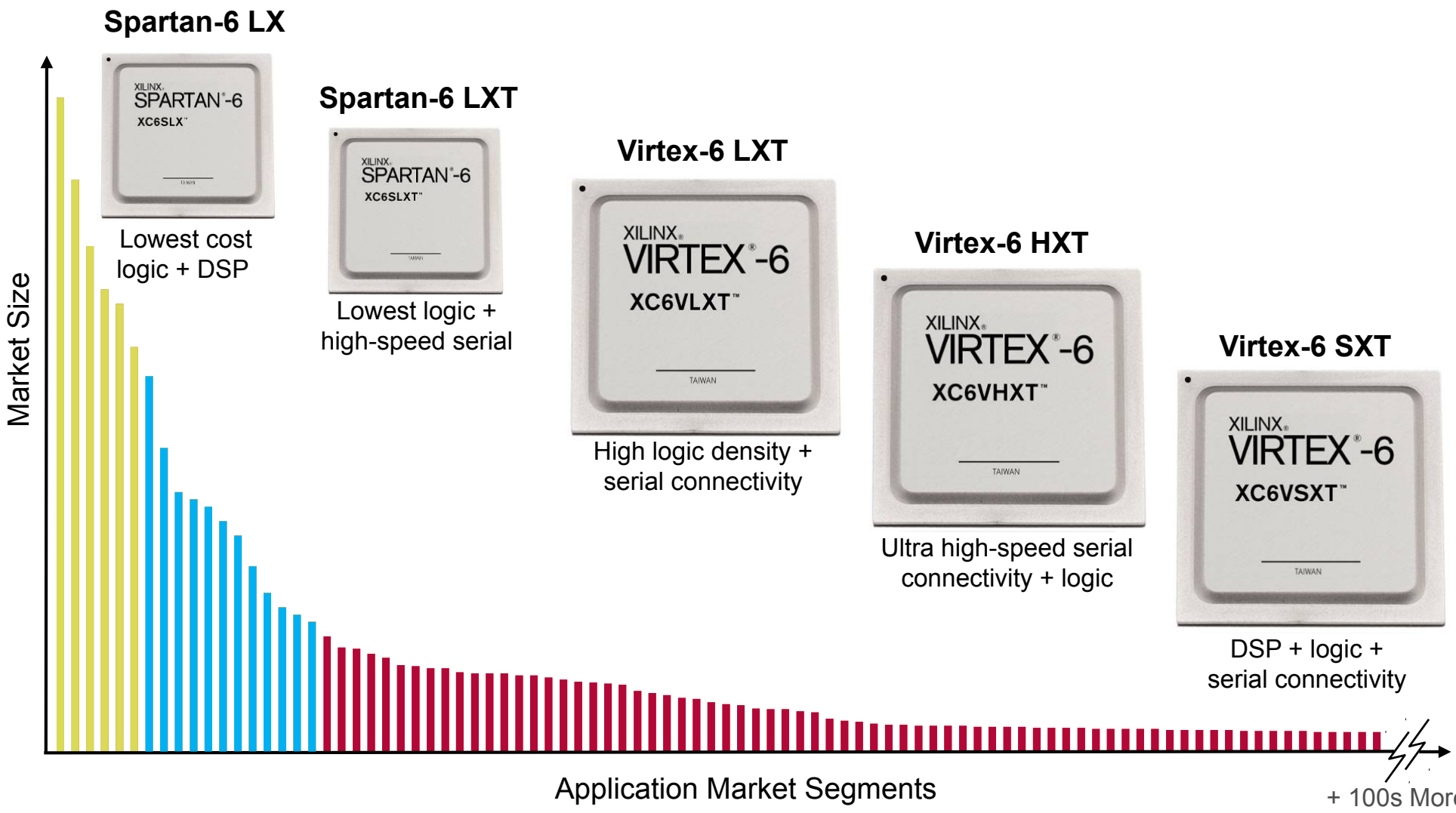
- FIFO Logic
- Tri-mode EMAC
- System Monitor

- Hardened Memory Controllers
- 3.3 Volt compatible I/O

*Optimized for target application in each family

Enables IP Portability, Protects Design Investments

Addressing the Broad Range of Technical Requirements



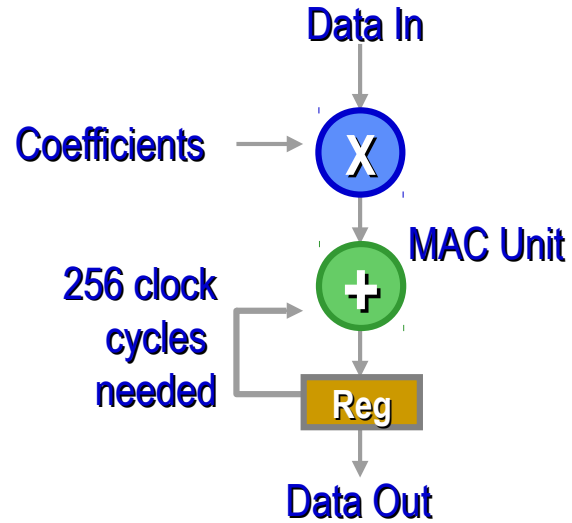
Higher DSP Performance

- Most advanced DSP architecture
  - New optional pre-adder for symmetric filters
  - 25x18 multiplier
    - High resolution filters
    - Efficient floating point support
  - ALU-like second stage enables mapping of advanced operations
    - Programmable op-code
    - SIMD support
    - Addition / Subtraction / Logic functions
  - Pattern detector
- Lowest power consumption
- Highest DSP slice capacity
  - Up to 2K DSP Slices

Reason 1: FPGAs handle high computational workloads

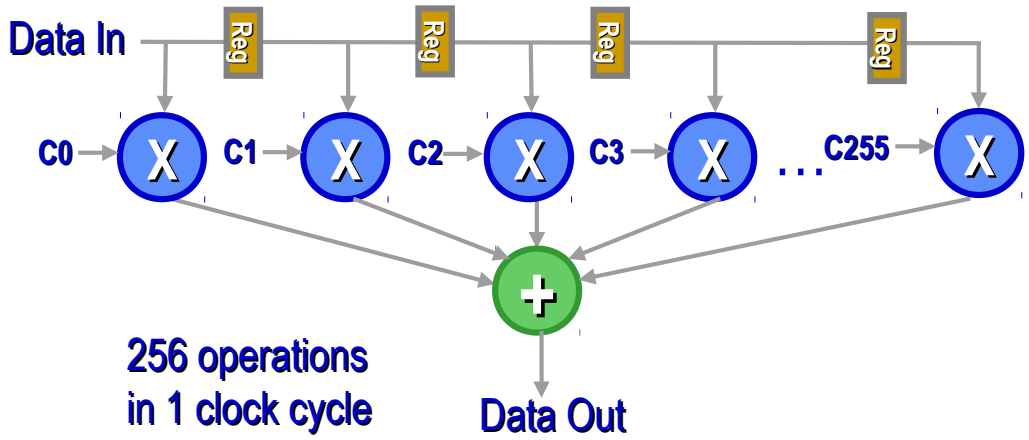
Speed up FIR Filters by implementing with parallel architecture

Programmable DSP - Sequential



$$\frac{1\ \text{GHz}}{256\ \text{clock cycles}} = 4\ \text{MSPS}$$

FPGA - Fully Parallel Implementation

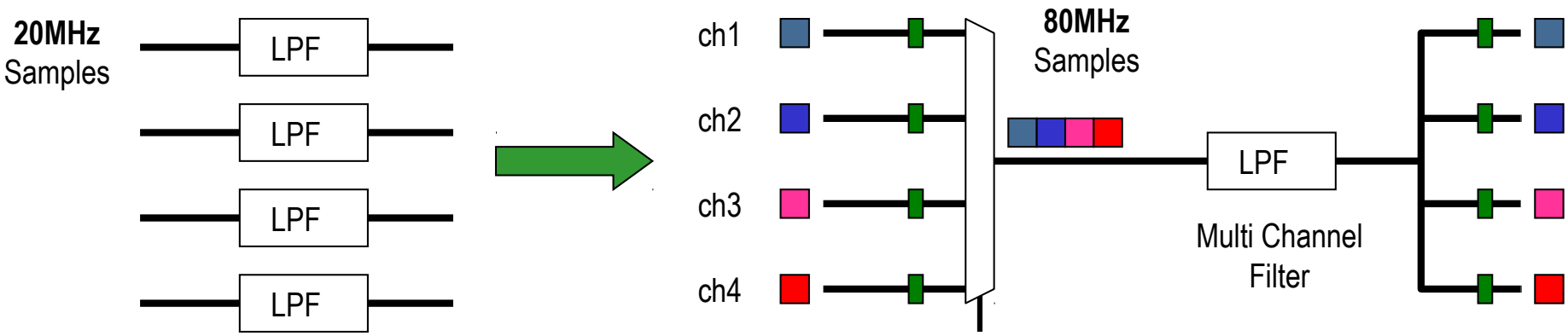


$$\frac{500\ \text{MHz}}{1\ \text{clock cycle}} = 500\ \text{MSPS}$$

Example 256 TAP Filter Implementation

Reason 2: FPGAs are ideal for multi-channel DSP Designs

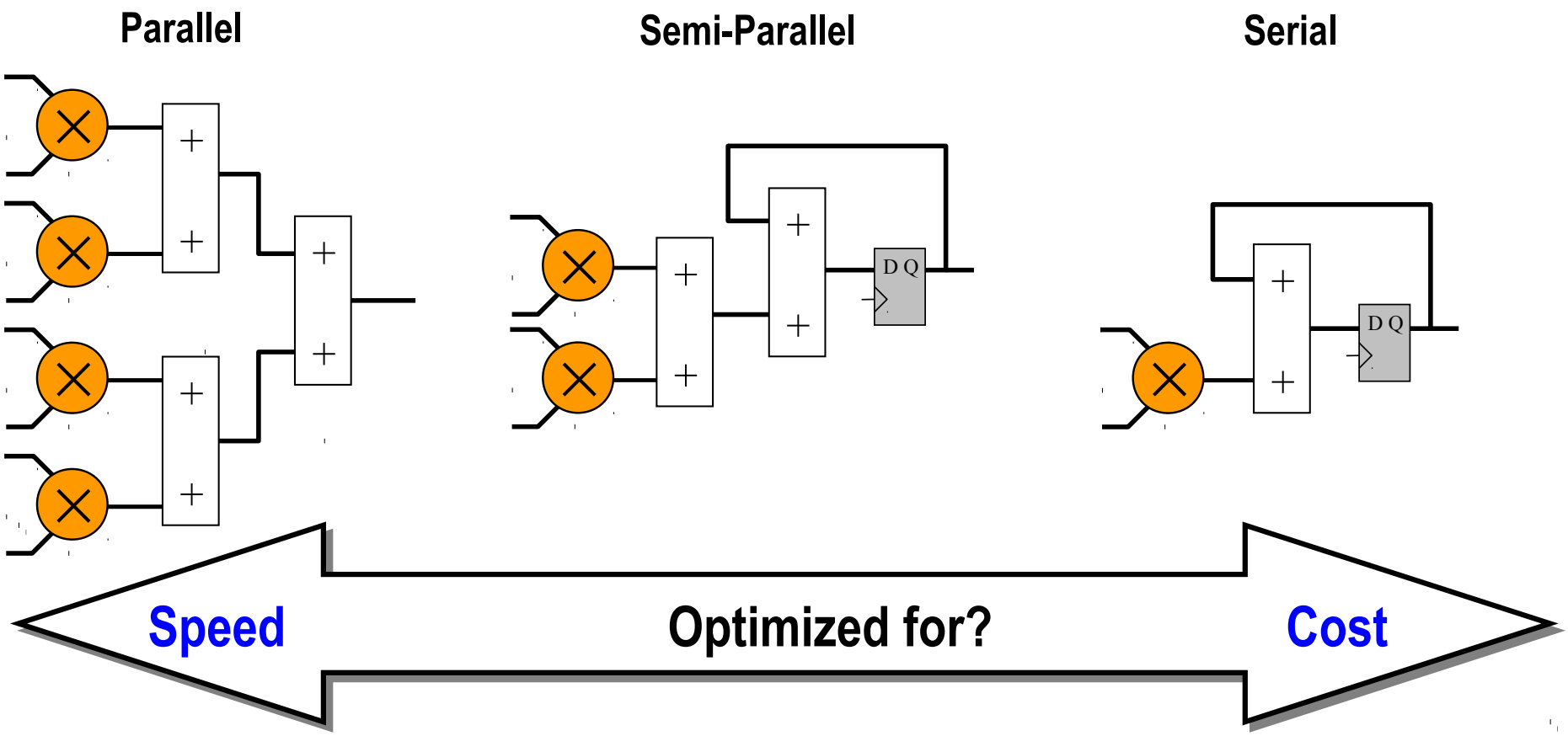
Can implement multiple channels running in parallel or time multiplex channels into one filter



- Many low sample rate channels can be multiplexed (e.g. TDM) and processed in the FPGA, at a high rate
- Interpolation (using zeros) can also drive sample rates higher

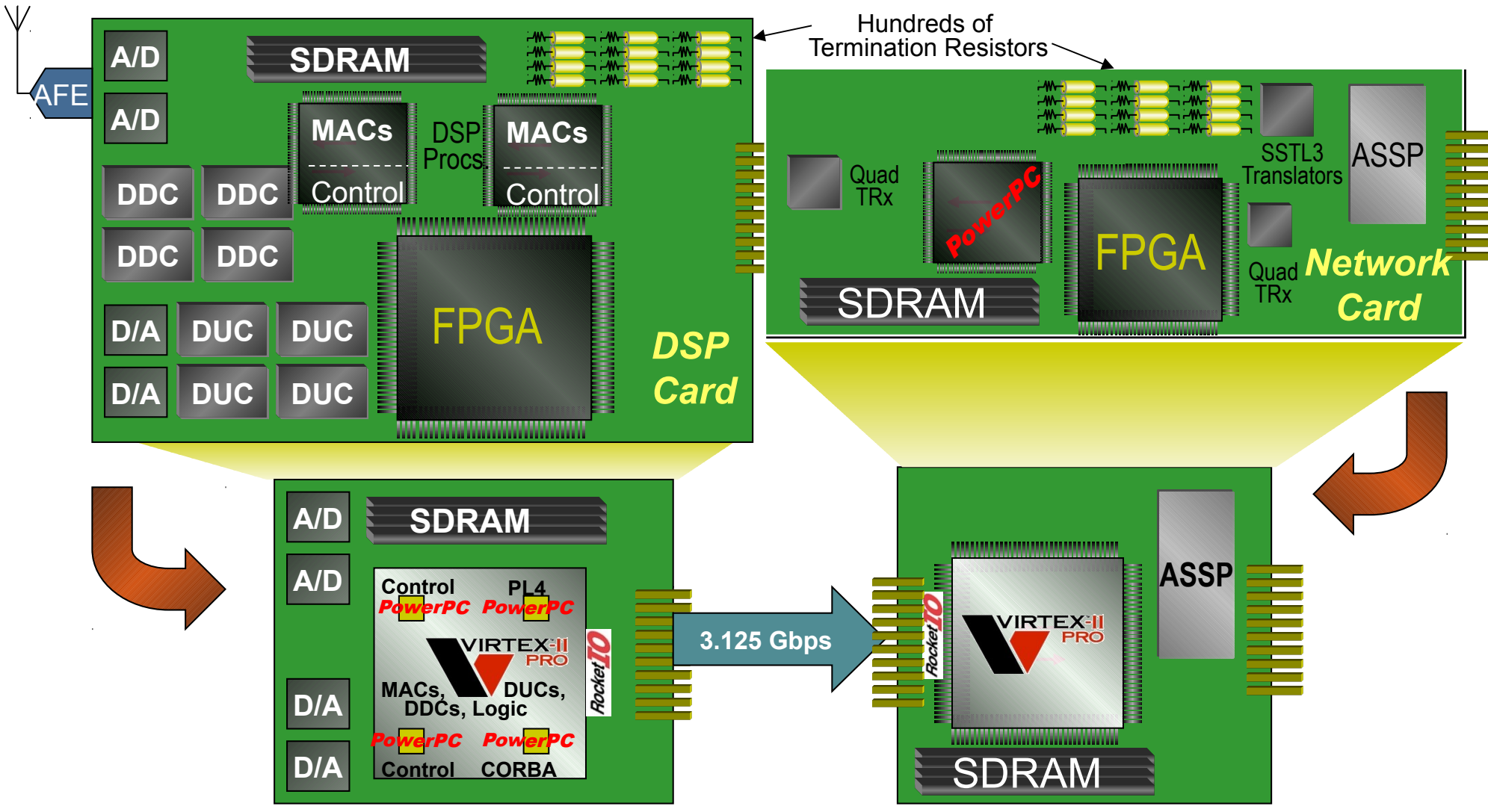
Reason 3: Customize Architectures to Suit your Goals

FPGAs allow Cost/Performance tradeoffs



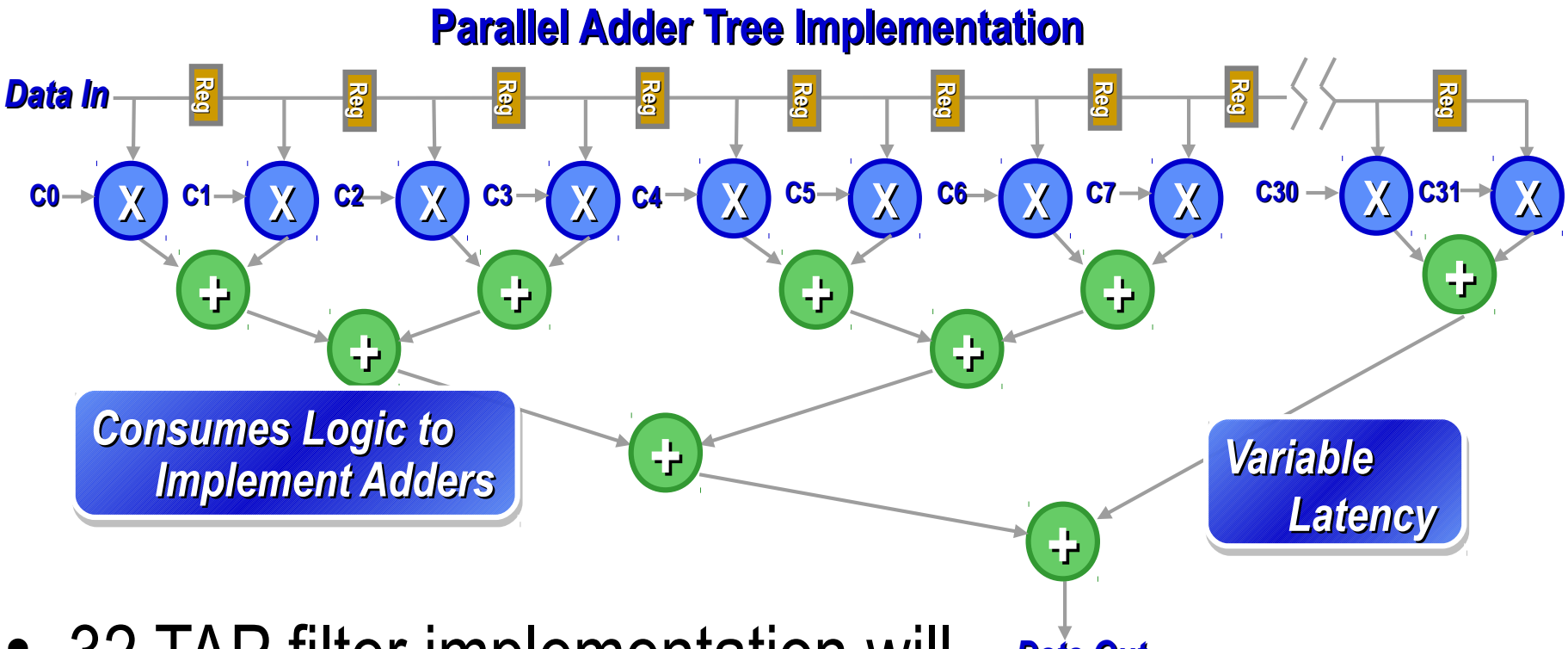
Reason 4: Lower System Cost through Integration

Implement Interface Logic within FPGA to connect DSP functions to I/O and Memory Devices



The XtremeDSP Slice Advantage

Without XtremeDSP Slice, Parallel Adder Tree Consumes Logic Resources

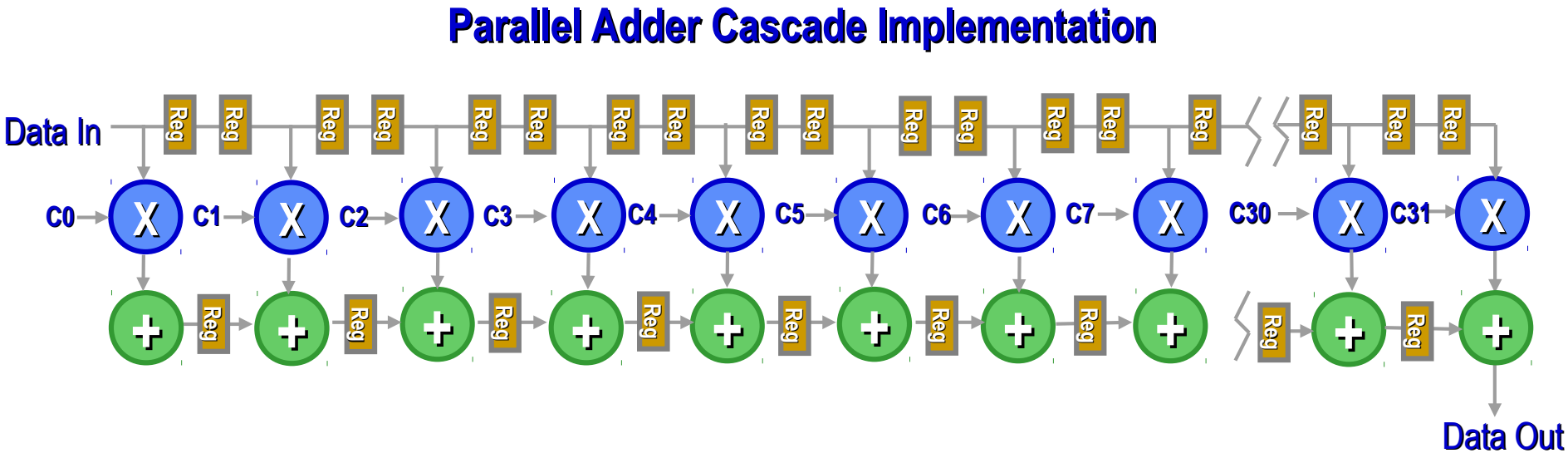


- 32 TAP filter implementation will consume 1,461 logic cells to implement adders in fabric

Fabric and Routing May Reduce Performance

The XtremeDSP Slice Advantage

With XtremeDSP Slice, Parallel Adder Tree Consumes Zero Logic Resources



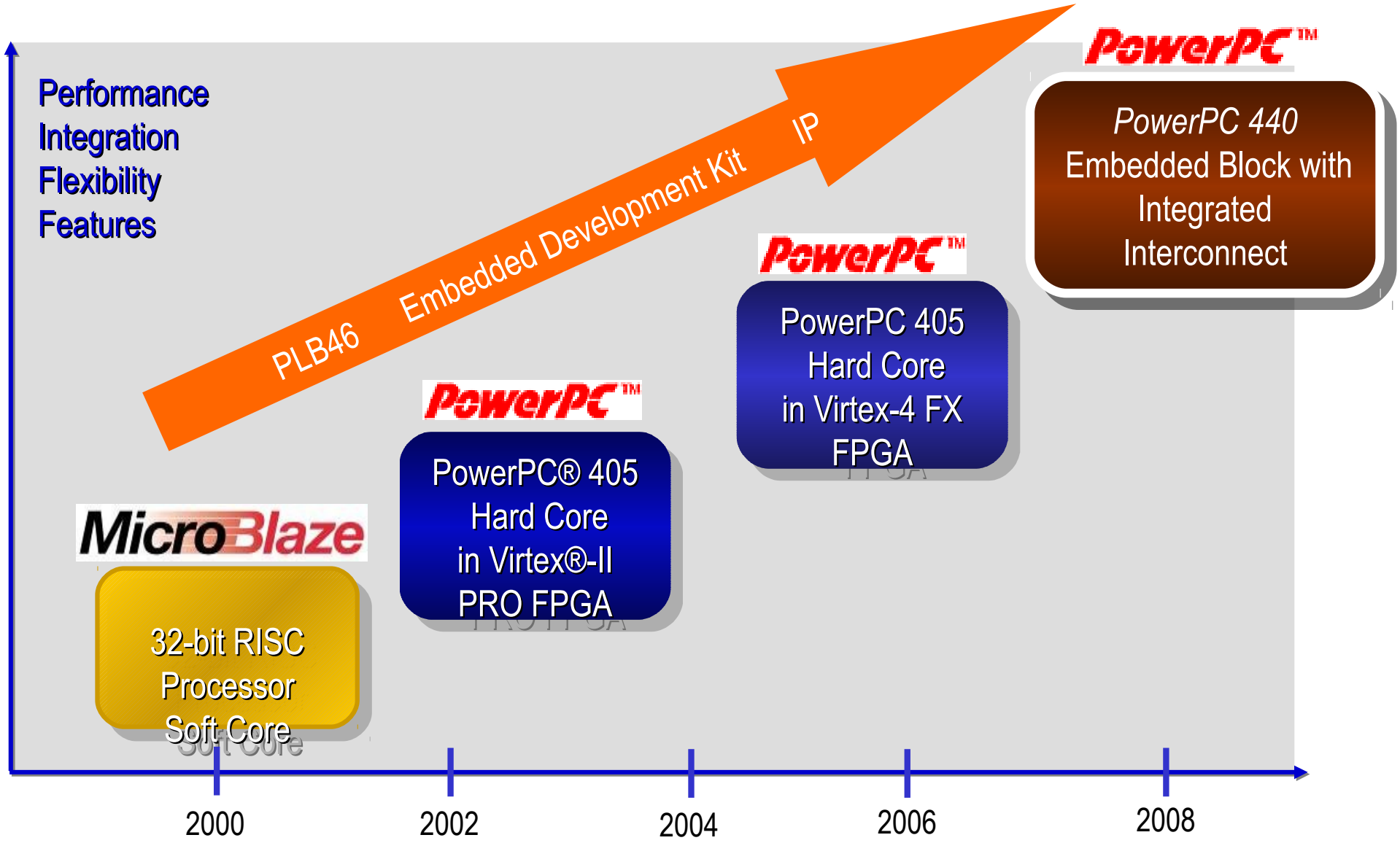
32 TAP filter implementation implemented entirely with XtremeDSP Slices

Sommaire

Embedded Design with the MicroBlaze Soft Processor Core

XILINX.

Xilinx Embedded Processor Innovation



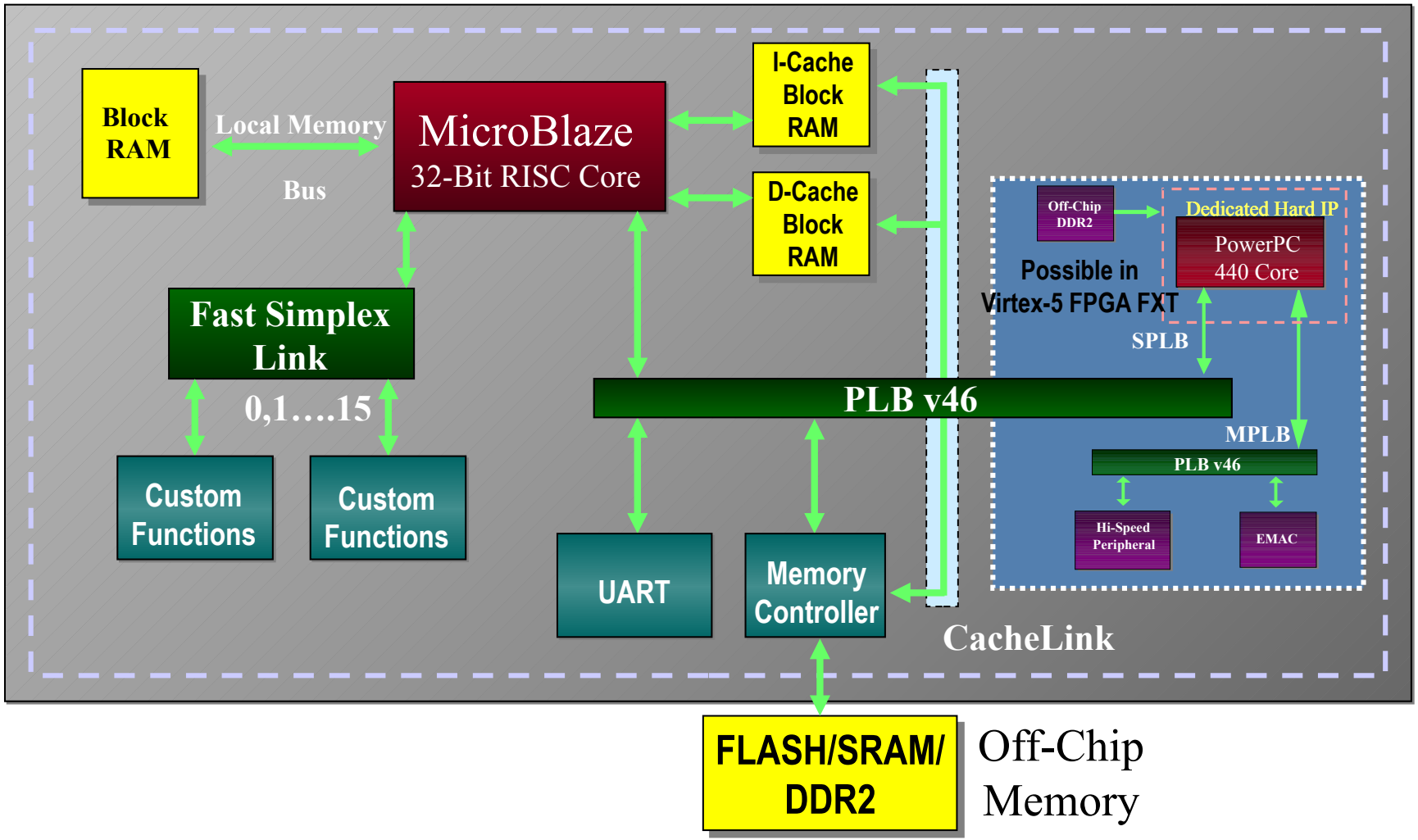
Supported FPGAs

- FPGA families
  - Spartan-3/3A/3AN/3A DSP/3E FPGA (MicroBlaze processor)
  - Spartan-6 (MicroBlaze Processor)
  - Virtex-4 FX (MicroBlaze and PowerPC 405 processors) and LX/SX FPGA (MicroBlaze processor)
  - Virtex-5 FXT (MicroBlaze and PowerPC 440 processor) LX/LXT FPGA (MicroBlaze)
  - Virtex-6 (MicroBlaze processor)

Embedded Design in an FPGA

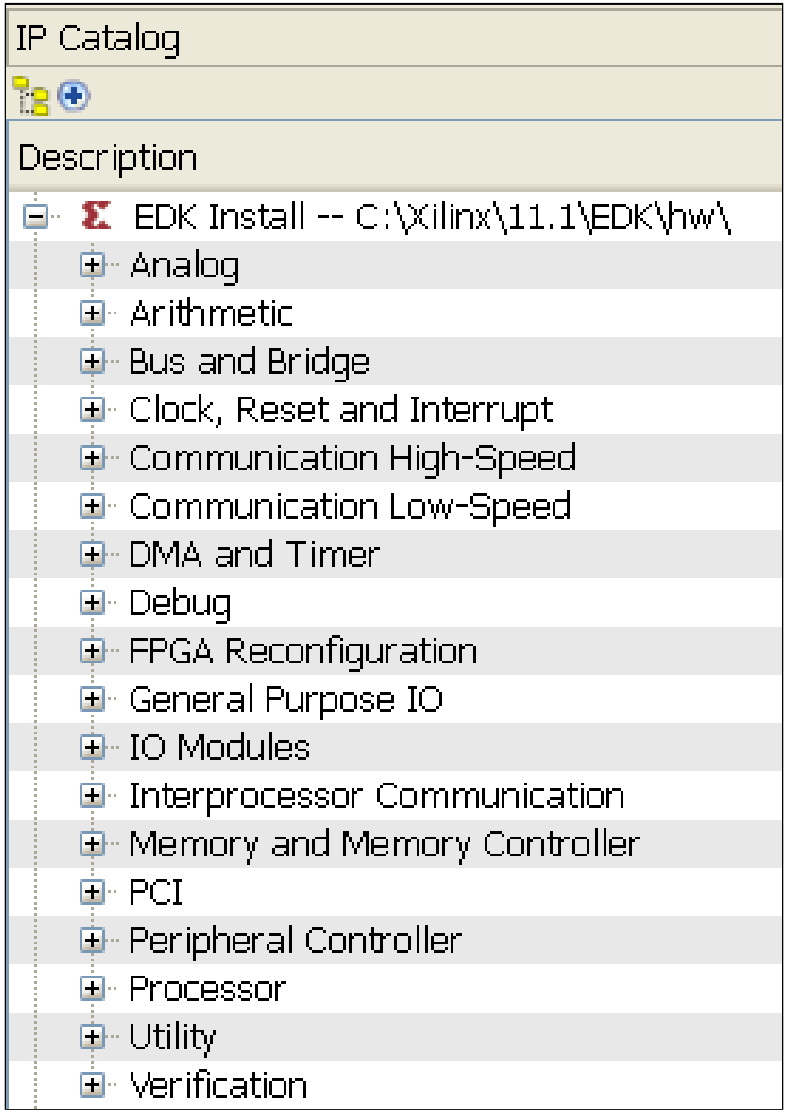
- Embedded design in an FPGA can consist of the following
  - FPGA hardware design
    - Processor system
      - MicroBlaze processor (soft core)
      - PowerPC processor (PPC440 hard core)
      - PLB or PLB v46 bus
      - PLB bus components
    - Other FPGA hardware
      - Peripherals can either be custom made by the user with a Xilinx bus interface or a library of pre-optimized peripherals are available

MicroBlaze Processor-Based Embedded Design

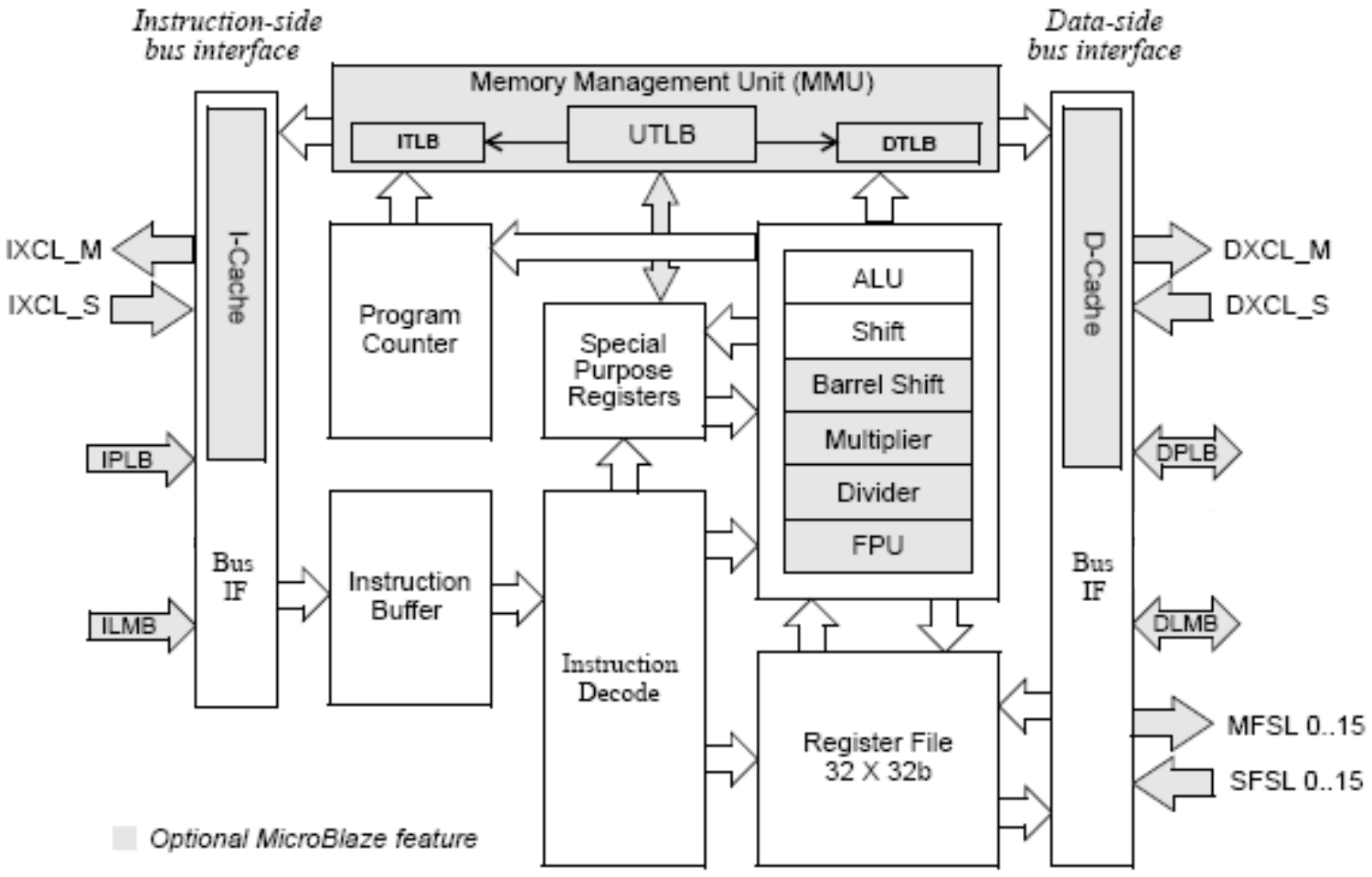


IP Peripherals

- All are included FREE!
- Bus infrastructure and bridge cores
- Memory and memory controller cores
- Debug
- Peripherals
- Arithmetic
- Timers
- Inter-processor communication
- External peripheral controller
- DMA controller
- PCI
- User core template
- …and Other cores



MicroBlaze Processor Block Diagram



MicroBlaze Core Block Diagram

MicroBlaze Processor Basic Architecture

- Embedded soft RISC processor
  - 32-bit address and data buses
  - 32-bit instruction word (three operands and two addressing modes)
  - 32 registers (32-bit wide)
  - Three or five pipeline stages (three stages if area optimization is selected)
  - Big-endian format
- Buses
  - Full Harvard architecture
  - PLB v46 (CoreConnect bus architecture standard), instruction, and data (user selectable)
  - LMB for connecting to local block RAM (faster), instruction, and data (user selectable)
  - Fast Simplex Links: dedicated, unidirectional point-to-point data streaming interfaces; support for up to 16 FSLs
  - Dedicated CacheLink ports for instruction and data caching with four-word cache line size and critical word-first access capability

MicroBlaze Processor Features

- ALU
  - Hardware multipliers/DSP48
  - Barrel shifter
- Floating Point Unit (FPU)
  - Implements IEEE 754 single-precision, floating-point standards
  - Supports addition, subtraction, multiplication, division, and comparison
- Program counter
- Instruction decode
- Instruction cache
  - Direct mapped
  - Configurable caching with CacheLink
  - Configurable size: 2 kB, 4 kB, 8 kB, 16 kB, 32 kB, 64 kB

MicroBlaze Processor Performance

- All instructions take one clock cycle, except the following
  - Load and store (two clock cycles)
  - Multiply (two clock cycles)
  - Branches (three clock cycles, can be one clock cycle)
- Operating frequency – fast speed grade, 5-stage pipeline
  - 307 MHz on Virtex-6 FPGA (-3)
  - 245 MHz on Virtex-5 FPGA (-3)
  - 154 MHz on Spartan®-6 FPGA (-3)
  - 119 MHz on Spartan-3 FPGA (-5)
- Performance of 1.15 DMIPS/MHz
- Fabric utilization – in LUT's, size optimized/speed optimized
  - 779/1,134 LUTs in Virtex-6 FPGA
  - 240/330 LUTs in Virtex-5 FPGA
  - 770/1,154 LUTs in Spartan-6 FPGA
  - 1,258/1,821 LUTs in Spartan-3 FPGA

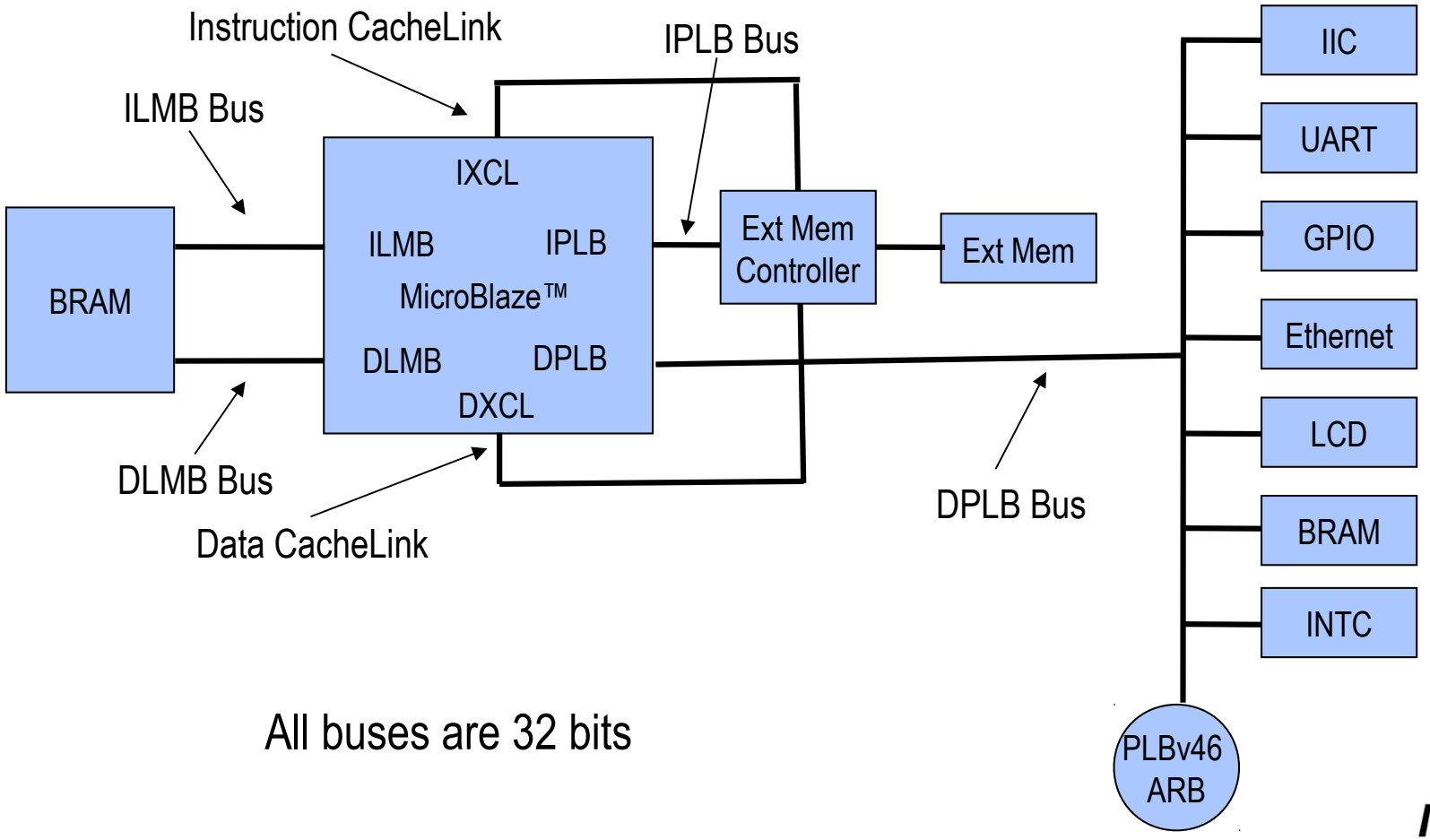
New MicroBlaze Processor v7 Features

- New features and improvements
  - High-performance PLB v46 interface and PLB v46 peripherals
  - Memory Management Unit (MMU) implements virtual memory management
    - Virtual memory management provides greater control over memory protection, which is especially useful with applications that can use an RTOS
    - Note…the MicroBlaze processor MMU is compatible but does not have the same functionality as the PPC405 processor MMU
  - Processing improvements
    - New float-integer conversion and float-square root instructions
    - Speeds up
      - FP > Int conversion
      - Int > FP conversion
      - FP square root

Buses 101

- Bus masters have the ability to initiate a bus transaction
- Bus slaves can only respond to a request
- Bus arbitration is a three-step process
  - A device requesting to become a bus master asserts a bus request signal
  - The arbiter continuously monitors the request and outputs an individual grant signal to each master according to the master's priority scheme and the state of the other master requests at that time
  - The requesting master samples its grant line until granted access. When the current bus master releases the bus, the master then drives the address and control lines to initiate a data transaction to a slave bus agent.
- Arbitration mechanisms
  - Fixed priority, round-robin, or hybrid

MicroBlaze Processor Bus Example



All buses are 32 bits

MicroBlaze

Local Memory Bus (LMB)

- The Local Memory Bus (LMB) provides single-cycle access to on-chip dual-port block RAM for MicroBlaze processors
- The LMB provides a simple synchronous protocol for efficient block RAM transfers
- The LMB provides a maximum guaranteed performance of 307 MHz in Virtex-6 FPGAs, for the local memory subsystem
- Harvard processor architecture
  - DLMB: data interface, local memory bus (block RAM only)
  - ILMB: instruction interface, local memory bus (block RAM only)

MicroBlaze

LMB Timing

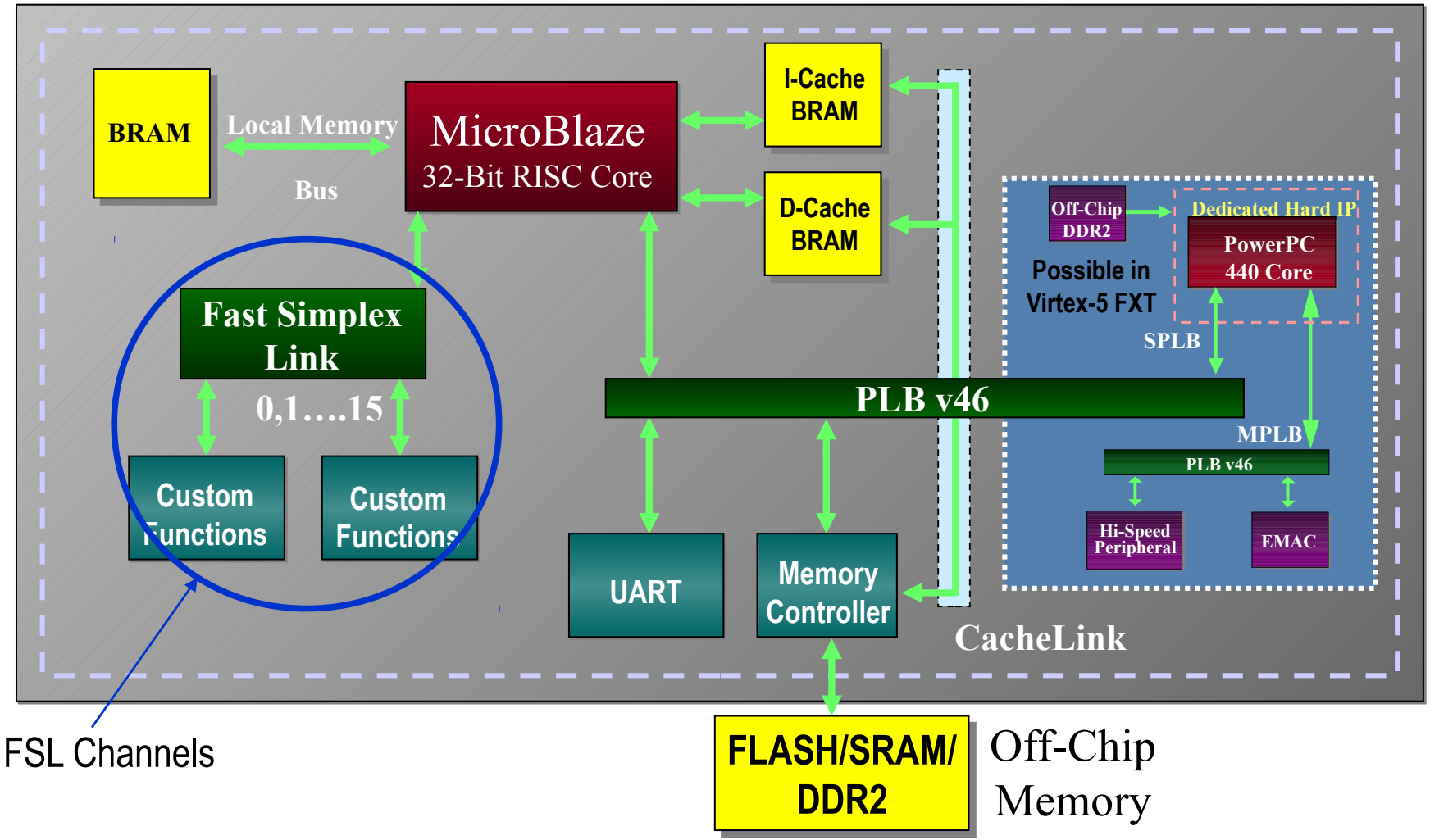
- Rules for generating an LMB clock
  - The MB, LMB, PLBv46 clock must be the same clock
- Use a timing period constraint on the processor clock line to insure place and route timing closure
  - This is done for you in Base System Builder
  - If the period constraint is placed on an external FPGA clock pin it will be *pushed through* any DCMs and global buffers that drive the MicroBlaze processor clock line

MicroBlaze

Bus Summary

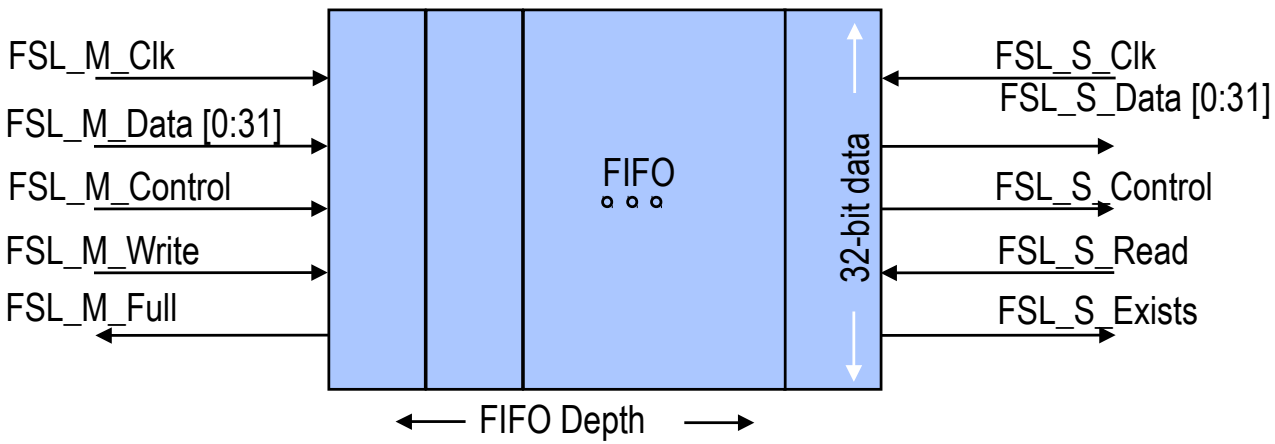
	CoreConnect Buses		Other Buses	
Feature	**PLB v46**	**DCR**	**OCM**	**LMB**
Processor family	PPC440, MicroBlaze	PPC405	PPC405	MicroBlaze
Data bus width	32, 64, or 128	32	32	32
Address bus width	32	10	32	32
Clock rate, MHz (max)[1]	100	125	375	125
Masters (max/typical)	16/2-8	1/1	1/1	1/1
Slaves (max/typical)	8/2-8	1/1	1/1	1/1
Data rate (peak)[2]	1600 MB/s	500 MB/s	500 MB/s	500 MB/s
Data rate (typical)[3]	533 MB/s[4]	100 MB/s[5]	333 MB/s[6,7]	333 MB/s
Concurrent read/write	Yes	No	No	No
Address pipelining	Yes	No	No	No
Bus locking	Yes	No	No	No
Retry	Yes	No	No	No
Timeout	Yes	No	No	No
Fixed/Variable Burst	Yes	No	No	No
Cache Fill	Yes	No	No	No
Target Word First	Yes	No	No	No
FPGA Resource Usage	High	Low	Low	Low

MicroBlaze Processor-Based Embedded Design



Fast Simplex Links (FSL)

- Unidirectional point-to-point FIFO-based communication
- Dedicated (unshared) and non-arbitrated architecture
- Dedicated MicroBlaze processor C and ASM instructions for easy access
- High-speed access in as little as two clocks on the processor side; 600 MHz at the hardware interface
- Available in Xilinx Platform Studio (XPS) as a bus interface library core from **Hardware** > **Create or Import Peripheral**



FSL Advantages

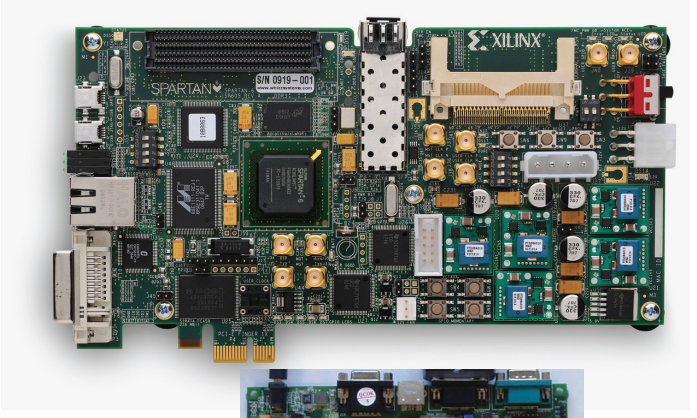
- Simple, fast, and easy to use
- Clock speed is not slowed down by new hardware
- FSL is faster than a bus interface
  - Saves clock cycles
- No arbitration/address decode/acknowledge cycles
- Decoupled data clock from CPU allows for asynchronous operation
- Minimal FPGA fabric overhead
- MicroBlaze processor v7 allows for up to 16 parallel FSL channels

Starting a Processor Design

- Many vendors support evaluation and demo boards with Xilinx FPGAs
  - Xilinx
  - Avnet
  - NuHorizons
  - Digilent
- Base System Builder (BSB) is a wizard to facilitate a fast processor-based system design by high abstraction, level-specification entry

Virtex®-5 FPGA ML507

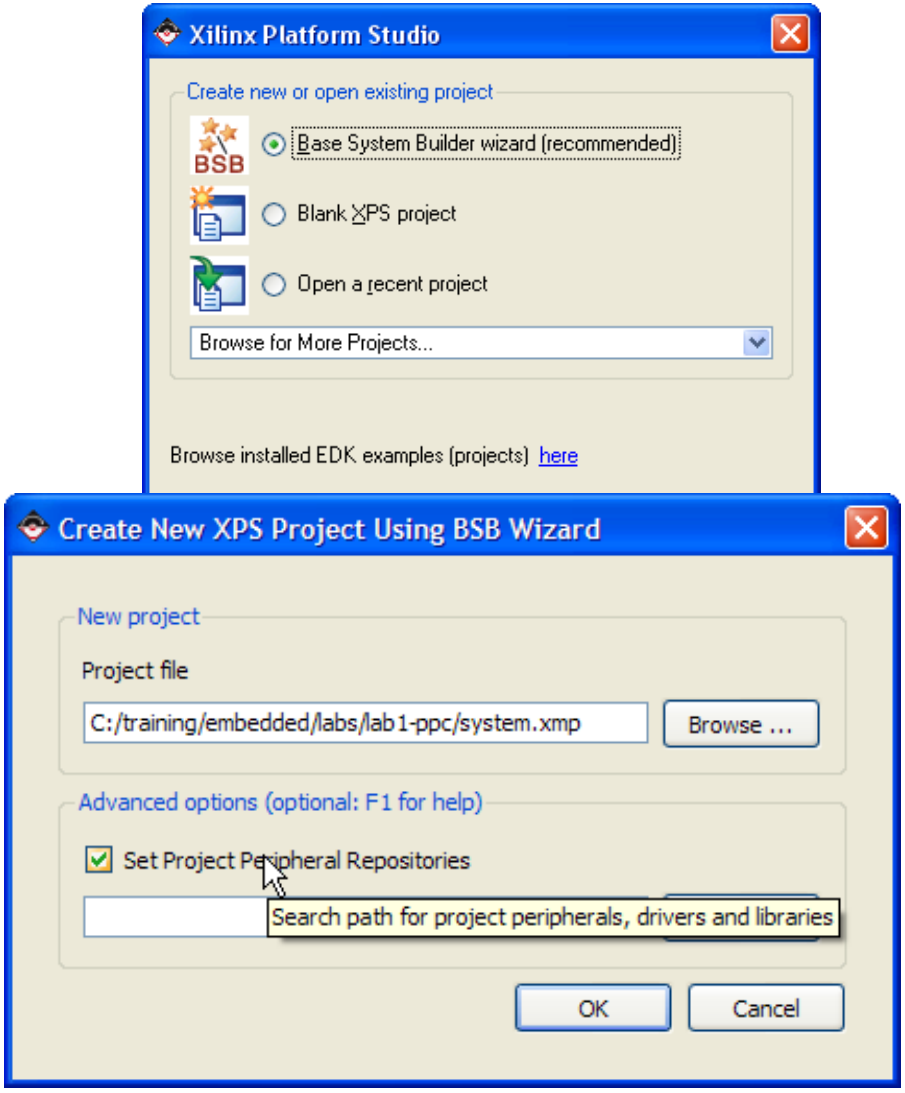
Spartan®-6
SP605 FPGA



Spartan-3E FPGA 1600E

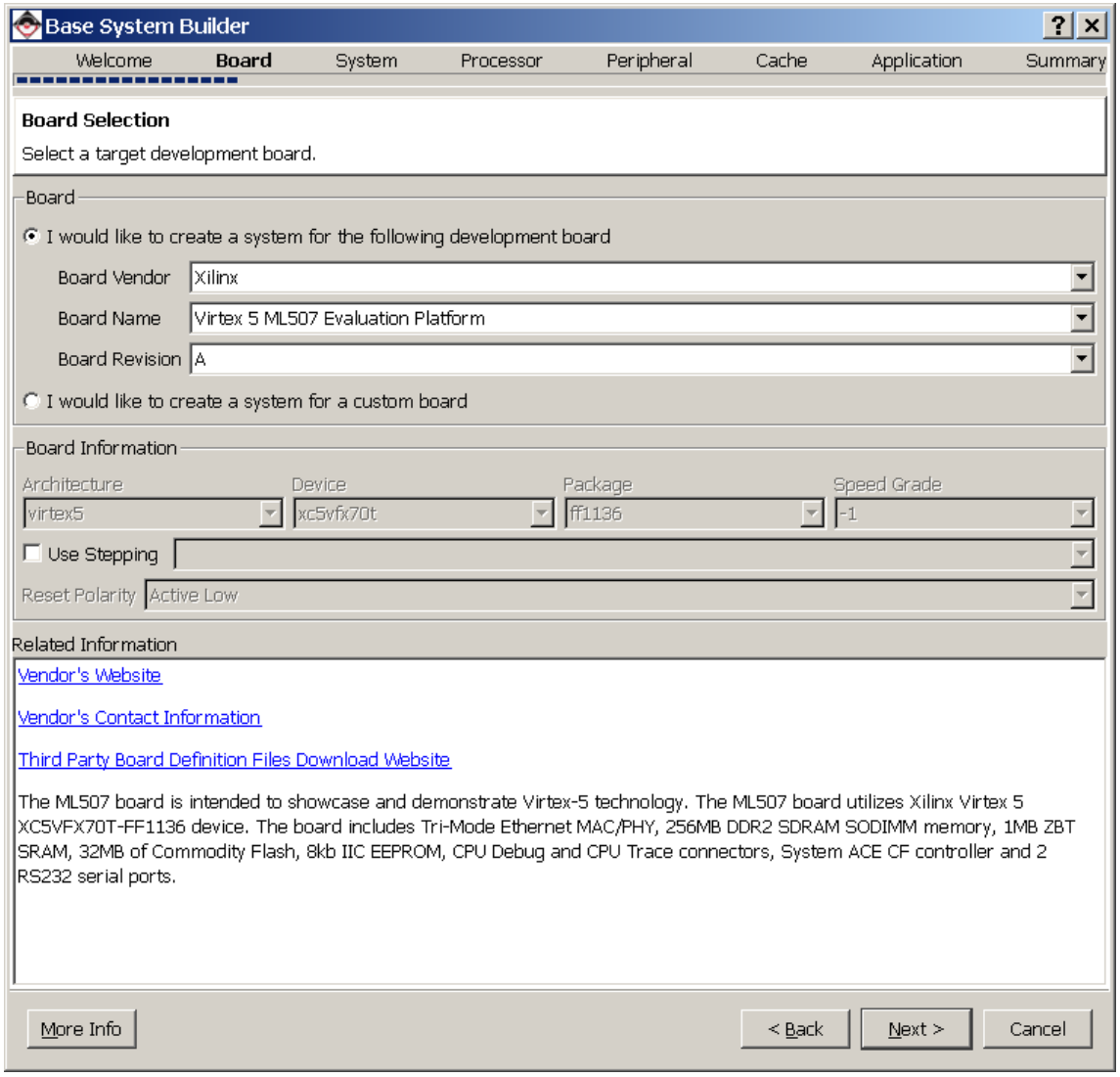
Create a New Project Using the BSB

- BSB enables fast design construction
  - Creates a completed platform and test application that is ready to download
  - Creates this system faster than you could by editing the MHS directly
  - Automatically matches the pinout of the design to the board
- The Set Project Peripheral Repository option is used for storing custom peripherals and drivers in a reserved location



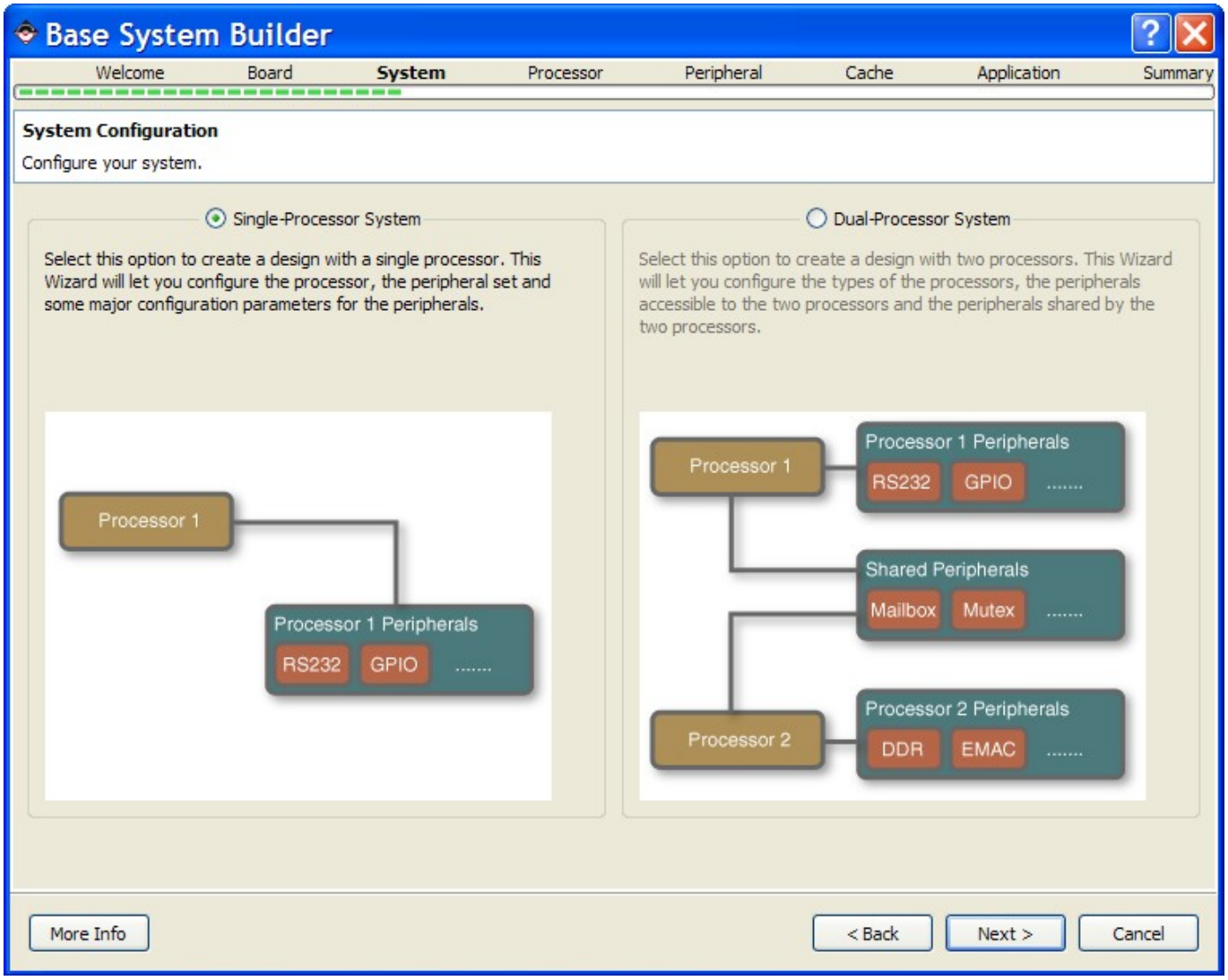
Selecting a Board

- Xilinx and its distribution partners sell demo boards with a wide range of added components
  - This dialog box allows you to quickly learn more about all available demo boards
  - It also allows you to install the necessary BSB files if you want to target a demo from another vendor
  - Note that you can also create your own BSB file for a custom board



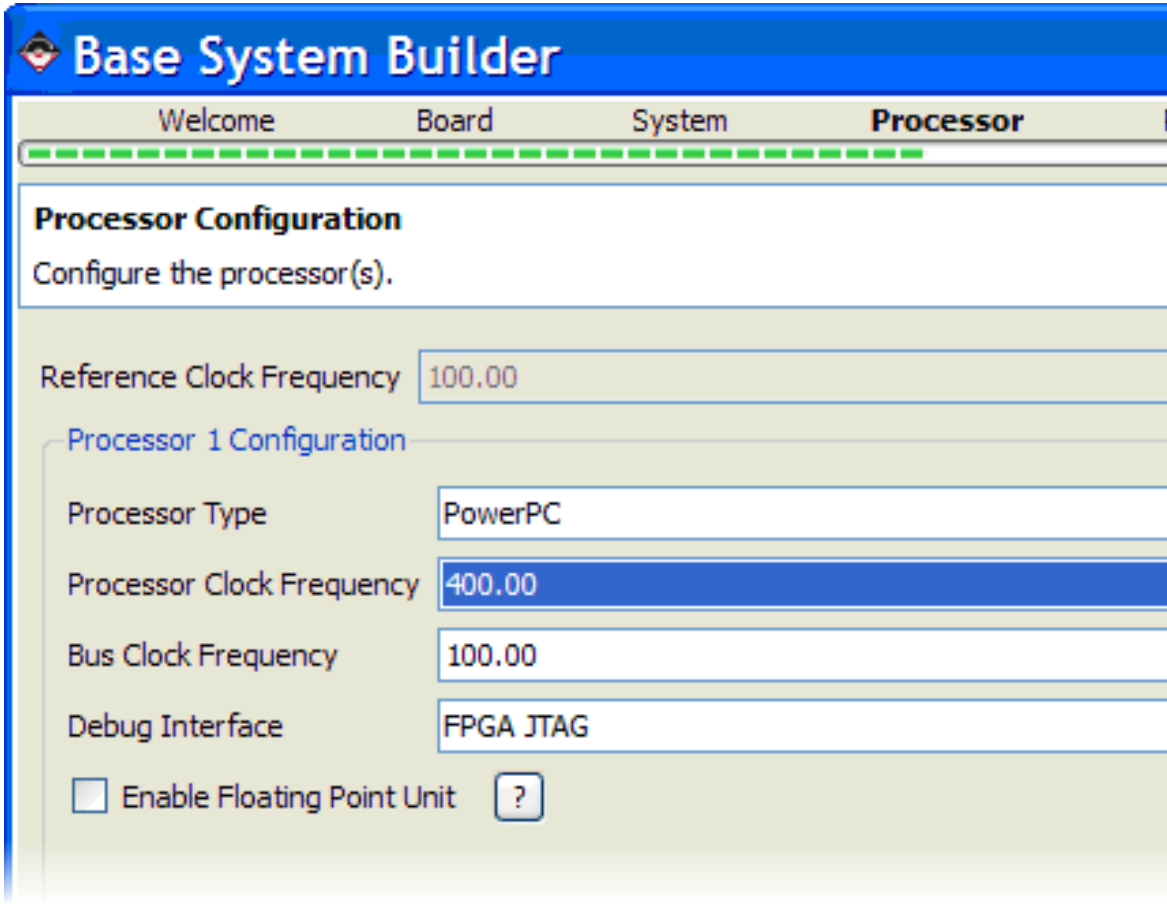
Selecting a Processor

- The Base System Builder (BSB) supports the construction of single- and dual-processor systems



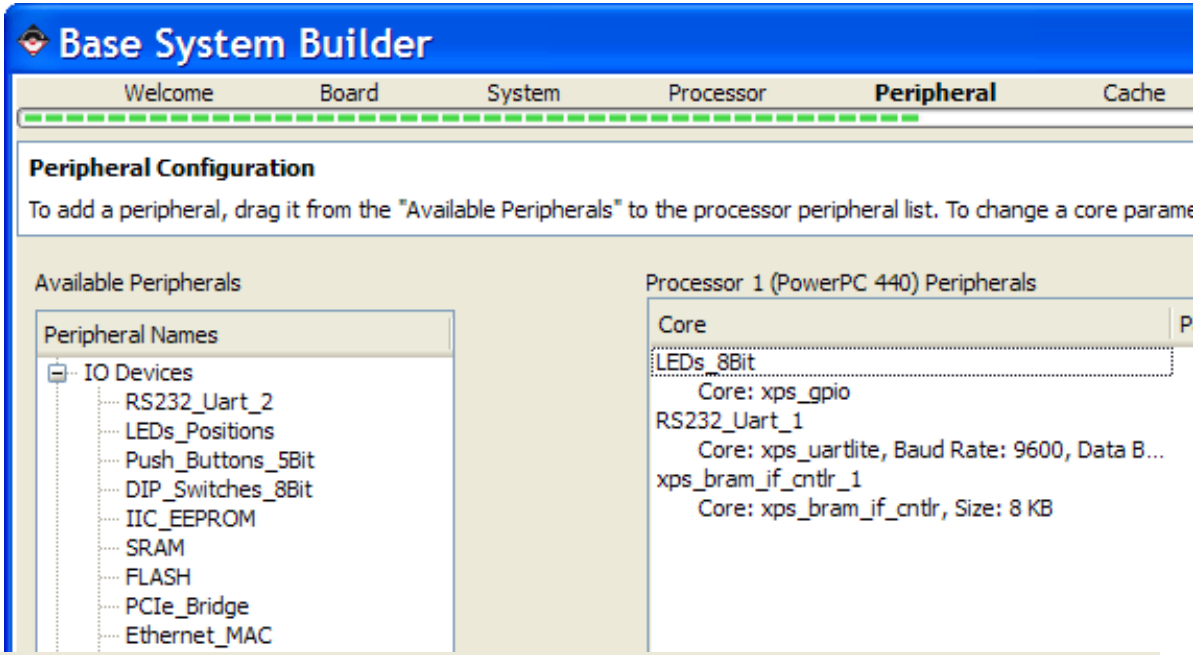
Configuring the Processor

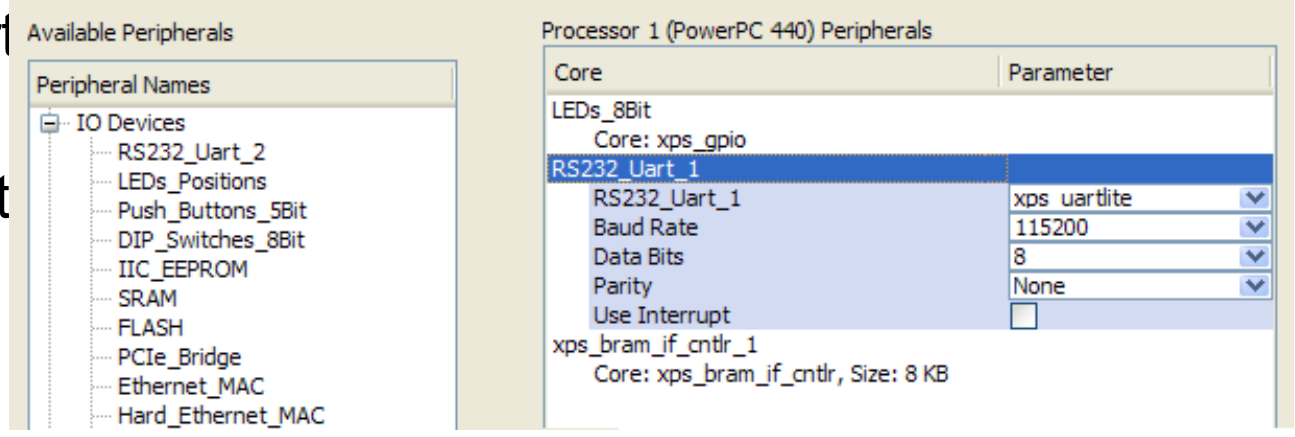
- Processor clock frequency is the clock rate connected directly to the processor
- Bus clock frequency is the clock rate of all bus peripherals in the system
- These selections will automatically customize the clock generator module
- The appropriate debug interface is added automatically



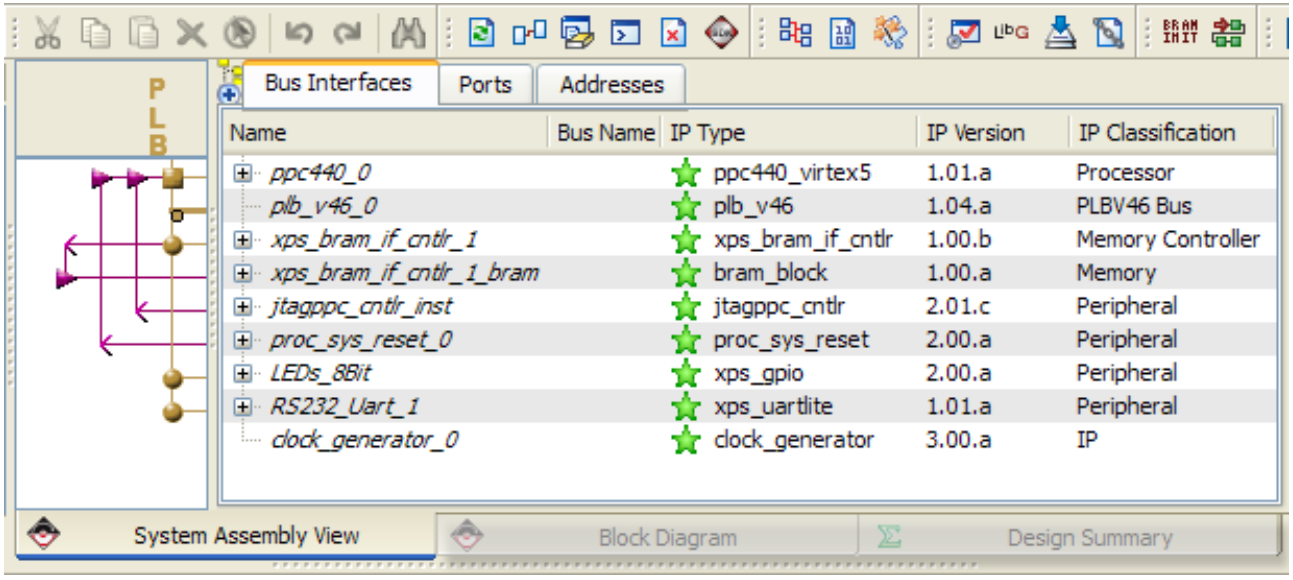
Configuring the I/O Interfaces

- Choose the peripherals you need from those available for your demo board
- Peripherals can be added or removed
- Most peripherals are customizable via drop-down lists when selected
- Most peripherals support the use of interrupts
- Internal peripherals exist for all board hardware configurations



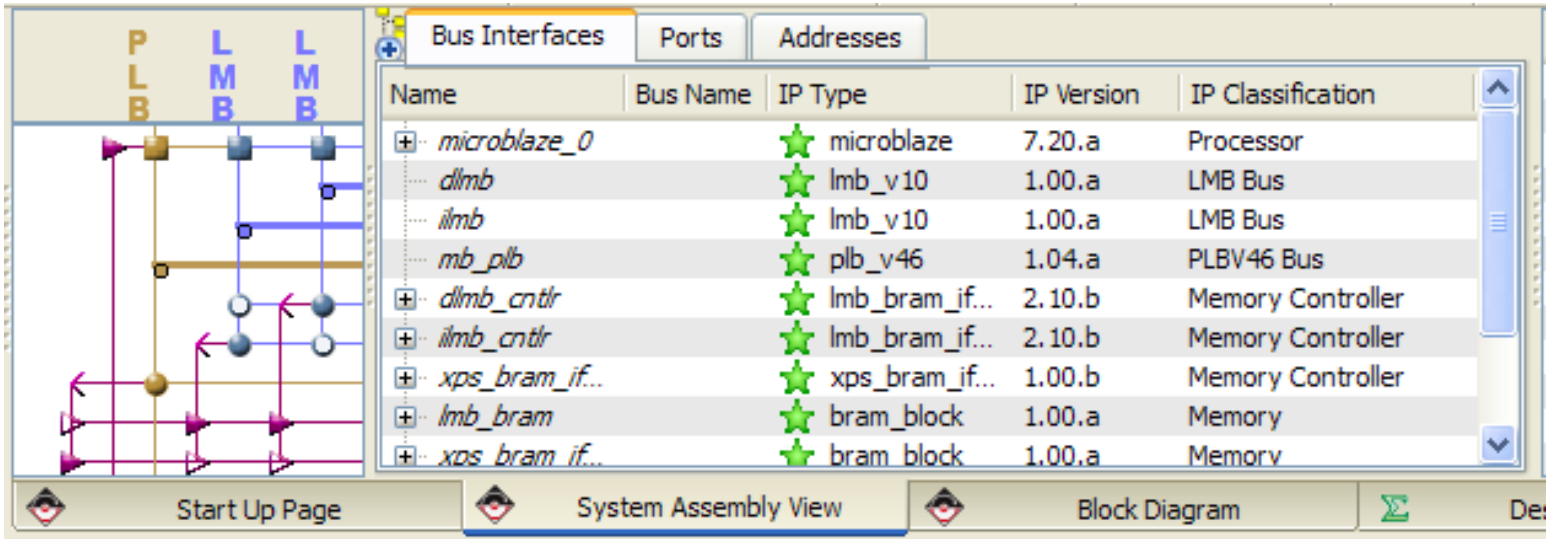


A Good Start on a Processor Design



Basic PowerPC processor system

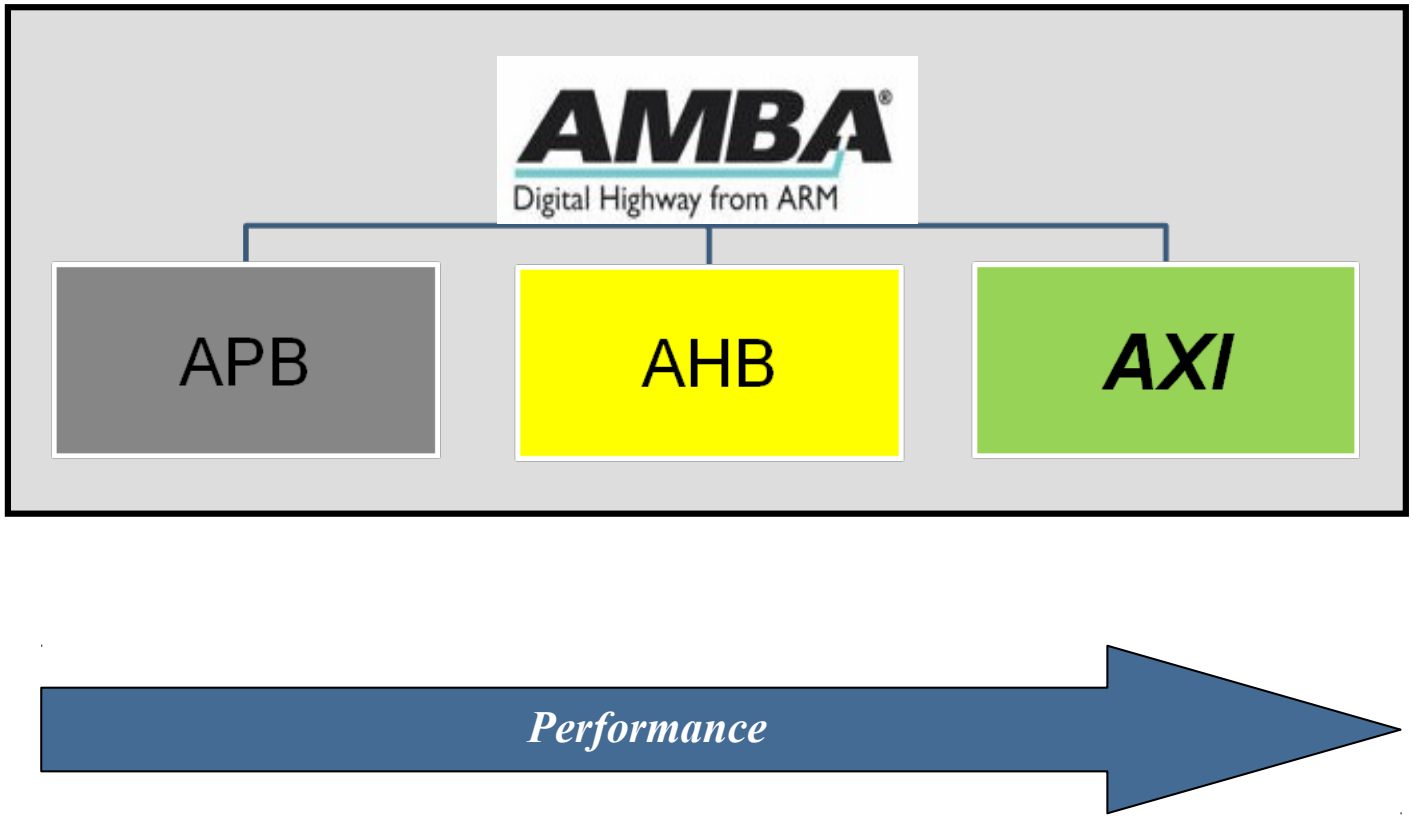
Basic MicroBlaze processor system



How to Convert a PLB-based Embedded System to an AXI-based System

AXI is Part of AMBA

- Advanced Microcontroller Bus Architecture



The ARM AXI

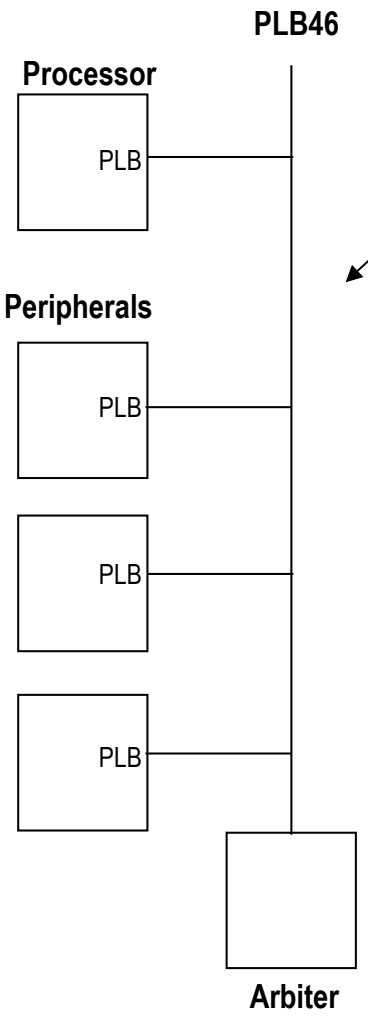
- Is an interface and protocol definition and is not a bus standard
  - Preproduction in the EDK 12.3 release
  - Currently, only targets the MicroBlaze soft processor core
    - Only for Spartan-6 and Virtex-6 devices
    - Most EDK peripherals will support AXI with the EDK 13.1 release

- Who should migrate their design?
  - Anyone that believes the features are beneficial (AXI advantages are coming up)
  - Anyone building an embedded design that will target the next generation of product families (after Spartan-6 and Virtex-6)

Pre-Production AXI

- ISE Design Suite 12.3 is a pre-production release for designs that use AXI IP
  - The AXI IP peripherals in this release have not yet completed qualification for use in production designs
  - Some wizard functionality in Xilinx Platform Studio does not yet fully support AXI-based designs
- For ISE Design Suite 12.3, pre-production status applies only to designs making use of AXI IP peripherals
  - If you design does not use these peripherals then your ok

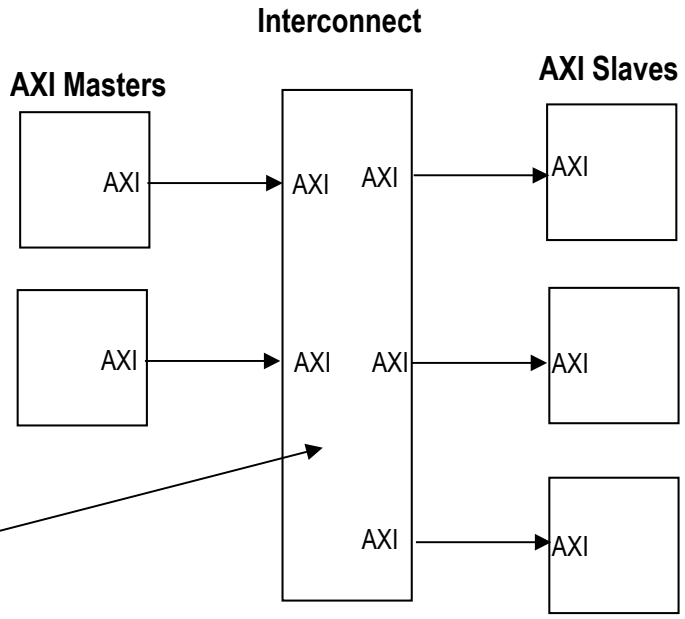
AXI is an Interface Specification



"Shared Access" Bus

AXI Interconnect IP

- Implementation is not described in the spec
- Several companies build and sell "AXI interconnect IP"
- Xilinx is building its own



Arrows indicate master/slave relationship, not direction of dataflow

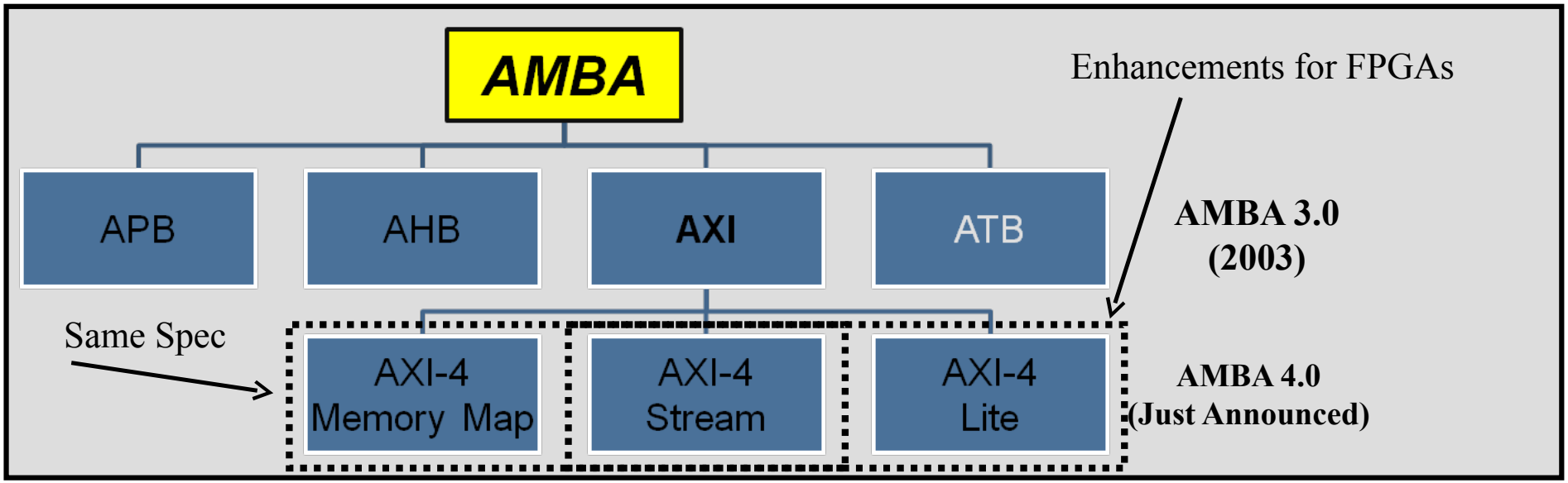
Master → Slave

AXI is an interface specification, not a bus specification

Why use AXI?

- Higher performance
  - AXI allows systems to be optimized for highest Fmax, maximum throughput, lower latency or some combination of those attributes. This flexibility enables you to build the most optimized products for your markets
- Easier to use
  - By consolidating a broad array of interfaces into AXI, you only need to know one family of interfaces, regardless of whether they are embedded, DSP or logic users. This makes it easier to integrate IP from different domains, as well as developing your own IP
- Enable ecosystem
  - Partners are embracing the move to AXI: an open, widely adopted interface standard. Many of them are already creating IP targeting AXI and other AMBA® interfaces. This gives you a greater catalog of IP, ultimately leading to faster time to market

AXI is Part of AMBA



Interface	Features	Similar to
Memory Map / Full	Traditional Address/Data Burst (single address, multiple data)	PLBv46, PCI
Streaming	Data-Only, Burst	Local Link / DSP Interfaces / FIFO / FSL
Lite	Traditional Address/Data—No Burst (single address, single data)	PLBv46-single OPB

Design Conversion

- Re-building the Embedded System is usually best
  - This is NOT difficult
    - Adding processors, busses, and IP is literally “drag-and-drop”
    - Custom IP…has some challenges
- If the IP was built with the IP Wizard, the IP can be migrated using templates provided in the following solution record
  - http://www.xilinx.com/support/answers/37425.htm
- If the IP cannot be altered to AXI, just add the AXI-to-PLB bridge component
  - This is described in the Xilinx AXI Reference Guide
- If the IP was built from scratch, refer to the “Memory Mapped IP Feature Adoption and Support,” in the Xilinx AXI Reference Guide