Table des matières

Chapitre 2: Conception d'un MPSoC temps réel et estimation de performance

Chapitr 3: Expérimentations et validations

Liste des figures

Liste des tableaux

Introduction générale

Avec le progrès de la capacité d'intégration de centaines de millions de transistors sur une seule puce et l'avancement au niveau de la conception des cœurs de processeurs embarqués deux tendances architecturales peuvent être distinguées.

La première tendance consiste à développer une architecture monoprocesseur extensible par un ensemble de coprocesseurs ou/et accélérateurs matériels génériques ou dédiés [1,3]. Comme exemple d'architecture monoprocesseur on peut citer : Power PC, Intel Pentium 4, ST100, et beaucoup d'autres processeurs de type *VLIW*[1] ou super-scalaire [3]. Dans de telles architectures, la communication est basée sur le principe maître/esclaves : le *CPU*[2] est le maître mais les périphériques sont les esclaves. Les interfaces des co-processeurs sont généralement faites de registres transposés dans la mémoire du CPU et peuvent produire des interruptions au CPU. Ces communications se font généralement via un bus partagé (le bus mémoire du CPU). En terme de performance, de telles architectures centrées autour d'un seul CPU ont un inconvénient : la dégradation de performance engendrée par le fait que le processeur effectue la communication aussi bien que le calcul [3].

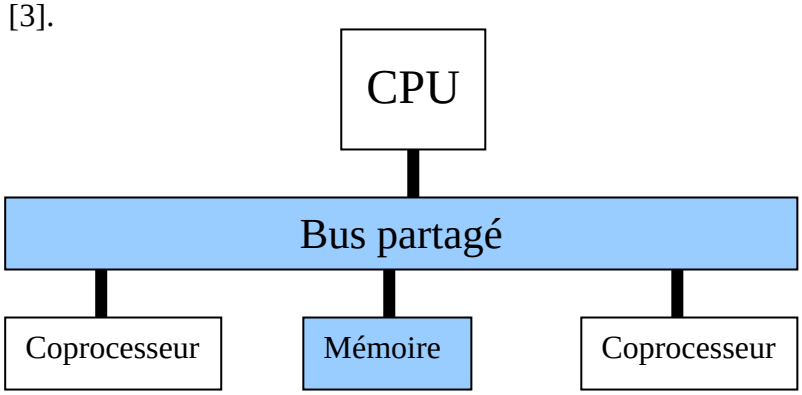

Figure 1: architecture multiprocesseurs

La deuxième tendance adresse des architectures multiprocesseurs. En effet, l'implémentation multiprocesseurs était réservée aux stations de calculs scientifiques [9, 10] et actuellement les systèmes embarqués sont adhérés. Ainsi, le fait de viser une implémentation multiprocesseurs de systèmes embarqués permet d'améliorer la réponse du système aux contraintes de performance et de faible consommation.

De très nombreux systèmes embarqués font appel à un ou plusieurs systèmes d'exploitation pour faciliter la gestion des événements et gérer la réactivité de ces systèmes.

[1] *Verly Long Instruction World*
[2] *Central Processing Unit*

En plus, et du fait de la complexité croissante de ces systèmes (exemple : Système sur puce), de la présence de fortes contraintes temps réel, de la limitation des ressources disponibles, tant en mémoire qu'en énergie disponible et donc en puissance de calcul, mais également de la pression exercée par le marché sur ces produits, l'usage de systèmes d'exploitation temps réel (*RTOS*[3]) est devenu nécessaire dans les systèmes embarqués. Des premiers travaux ont été élaborés dans notre équipe [1] afin de proposer un modèle de communication interprocesseur. Ce modèle a été étudié avec des primitives logicielles.

Parmi les objectifs de ce projet on cite l'implémentation d'un modèle sur une plateforme mutliporcesseur d'une part et l'expérimentation d'un environnement de prototypage des systèmes réactifs/multiprocesseurs (Processeurs et RTOS embarqués), sur des architectures reconfigurables dans le cadre des Systèmes sur puce (*SoC*[4]) d'autre part. Cette plateforme a été utilisée afin d'évaluer les performances des systèmes sur puce temps réel en second lieu.

Ce rapport est organisé de la façon suivante :

Dans le premier chapitre, on présentera d'abord, une étude des différentes architectures multiprocesseurs qui existent, Puis on mettra en évidence les concepts de base des systèmes d'exploitation temps réel ainsi que quelques exemples de RTOSs embarqués les plus connus et on terminera ce chapitre par une étude de quelques outils d'estimation de performance dans le cadre des systèmes sur puce.

Dans le deuxième chapitre, on commencera d'abord par la présentation de notre plateforme de travail. Ensuite, on passera à la conception d'une architecture multiprocesseurs en utilisant des modules de communication inter processeur implémentés en hardware offert par Altera. On terminera ce chapitre par la génération d'un modèle d'estimation de performance pour les applications temps réel.

Dans le dernier chapitre, nous envisagerons, dans la première section, la description de l'application de traitements d'images 3D et la validation du modèle d'estimation proposé à travers cette application, et dans la deuxième section, la conception de SoC multiprocesseurs sur des architectures reconfigurables. Il s'agit en fait d'étudier les points clés liés à la conception d'un tel système multiprocesseurs, dans le cadre des SoCs. En effet, et vu les limites des RTOS embarqués, un modèle de communication inter processeur doit se mettre en place. Afin de valider ce modèle, un exemple de système multiprocesseurs a été réalisé à base de la plate-forme ALTERA autour du cœur de processeur « *NIOS* » et le bus on chip « *AVALON* ».

[3] *Real Time Operating System*
[4] *System on Chip*

Chapitre 1

Système sur puce multiprocesseur temps réel: Etat de l'art

1- Introduction

Les nouvelles technologies s'orientent vers l'intégration sur une même puce de plusieurs processeurs, *DSP*[5], *IP*[6] matériels et logiciels, mémoires, bus partagés, etc. Nous parlons ainsi de systèmes multiprocesseurs mono puce (*MPSoC*[7]). En fait, les systèmes multiprocesseurs sont l'une des solutions pour répondre à la complexité croissante des systèmes intégrés utilisés pour des applications telles que les applications multimédia.

Pour ce faire, depuis plusieurs années, des systèmes d'exploitation temps réel sont utilisés dans les architectures multiprocesseurs sur puce vu que la présence des RTOS permet de structurer et de simplifier la programmation de la partie logicielle d'un tel SoC.

Ce chapitre est organisé comme suit :

La première partie est consacrée pour la présentation des différentes topologies possibles pour la réalisation d'une plateforme multiprocesseurs.

Dans la deuxième partie de ce chapitre, nous décrirons les concepts de base des systèmes d'exploitation temps réel, puis nous présentons les caractéristiques de quelques RTOS utilisés dans le cadre des systèmes sur puce.

La troisième partie présentera un état de l'art sur les différents outils d'estimation du temps d'exécution existant ainsi que notre contribution.

2- Taxonomie d'architectures multiprocesseurs

2.1- Les systèmes à mémoire partagée (Shared-memory systems)

Les machines du premier groupe que nous appelons les architectures à mémoire partagée centralisée Figure 2, ont au maximum quelques douzaines de processeurs au milieu des années 90. Les multiprocesseurs avec un faible nombre de processeurs peuvent se partager une mémoire centralisée unique et un bus pour interconnecter les processeurs et la mémoire [1]. Avec de gros caches, le bus et la mémoire unique peuvent satisfaire les besoins mémoire d'un petit nombre de processeurs. Puisqu'il y a une seule mémoire principale avec un temps d'accès uniforme pour chaque processeur, ces machines sont parfois appelées *UMA*[8] pour Accès Mémoire Uniforme. Ces systèmes offrent un modèle de programmation général et

[5] *Digital Signal Processor*
[6] *Intellectual Properties*
[7] *MultiProcessor System on Chip*
[8] *Uniform Memory Access*

« *commode* » permettant un partage simple des données, à travers un mécanisme uniforme de lecture et d'écriture des structures partagées dans la mémoire globale.

La facilité et la portabilité de la programmation sur de tels systèmes réduisent considérablement le coût de développement des applications parallèles. Par contre, ces systèmes souffrent d'un handicap qui est la grande latente pour l'accès à la mémoire, ce qui rend la flexibilité (l'extensibilité de l'architecture pour d'autres applications) assez limitée. Ce type d'architecture à mémoire partagée centralisée reste de loin l'organisation la plus populaire actuellement dans les multi-ordinateurs distribués sur réseau.

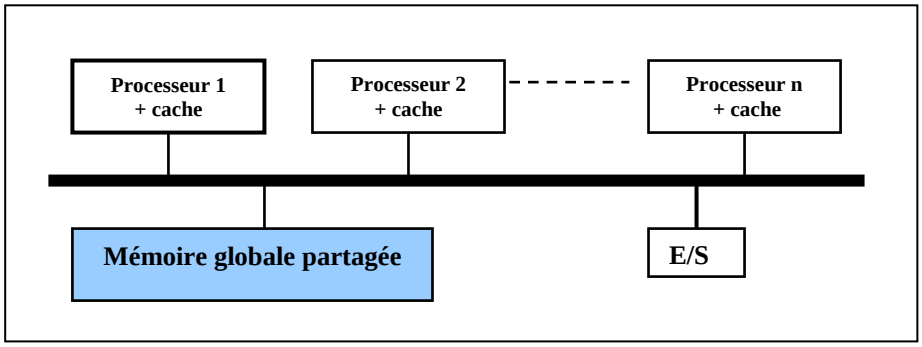

Figure 2: architecture à mémoire partagée centralisée

2.2- Les systèmes à mémoire distribuée (distributed memory system)

Ces systèmes sont souvent appelés « *les multi–ordinateurs* ». Ils sont constitués de plusieurs nœuds indépendants. Chaque nœud consiste en un ou plusieurs processeurs et de la mémoire centrale. Les nœuds sont connectés entre eux en utilisant des technologies d'interconnexion extensibles (scalable) Figure 3. Ces systèmes sont dits aussi machines à architecture de type *NUMA*[9], car en pratique, dans un réseau de stations de travail, l'accès à la mémoire locale de la station est nettement plus rapide que celui à la mémoire d'une station distante via le réseau [1].

La nature flexible de tels systèmes, les rend d'une très grande capacité de calcul. Mais, la communication entre des processus résidents dans des nœuds différents nécessite l'utilisation de modèles de communication par passage impliquant un usage explicite de primitives du type Send/Recieve[8]. En optant pour ce type de systèmes, le concepteur doit particulièrement faire attention à la distribution des données ainsi qu'à la gestion des communications (le transfert des processus pose un important problème à cause des différents espaces d'adressage, c'est-à-dire deux variables distinctes peuvent avoir la même adresse

[9] *Non Uniform Memory Access*

logique et deux adresses physiques différentes). Ainsi, les problèmes logiciels, contrairement aux problèmes matériels sont relativement complexes dans les systèmes à mémoire distribuée.

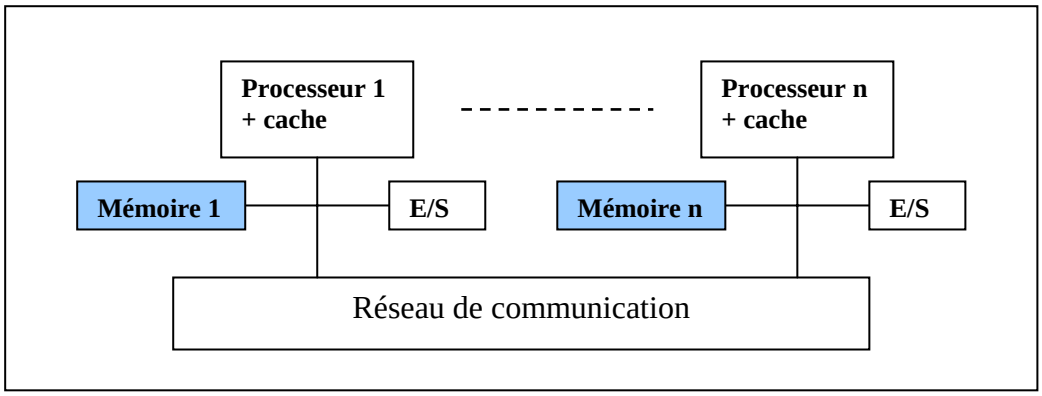

Figure 3: architecture à mémoire distribuée

2.3- Comparaison entre un système à mémoire partagée et celui à mémoire distribuée

Le tableau suivant récapitule les différences essentielles entre les architectures à mémoire partagée et celles à mémoire distribuée. Nous constatons que la communication entre les processeurs est plus simple dans les systèmes à mémoire distribuée. En effet, ce type de systèmes est utilisé pour un nombre élevé de processeurs contrairement aux systèmes à mémoire partagée.

Architecture à mémoire partagée	**Architecture à mémoire distribuée**
Temps d'accès à la mémoire uniforme pour tous les processeurs (UMA)	Temps d'accès dépendant de la position du mot de donnée en mémoire
Petit nombre de processeurs	Grand nombre de processeurs
Communication des données entre processeurs assez complexe	Communication facile entre processeurs
Les processeurs disposent généralement de plusieurs niveaux de cache (ou gros cache)	Processeurs avec des caches ordinaires
Architectures d'une flexibilité limitée	Architectures flexibles
Processeurs interconnectés par bus	Processeurs interconnectés par réseau d'interconnexion
Grande mémoire physiquement centralisée	Petites mémoires physiquement distribuées

Tableau 1: Comparaison entre une architecture à mémoire partagée et celle distribuée

2.4- Les systèmes à mémoire distribuée partagée

Un concept relativement nouveau, qui est la mémoire distribuée partagée, combine les avantages des deux approches. Un système *DSM*[10] implante (logiquement) un système à mémoire partagée sur une mémoire physiquement distribuée. Ces systèmes préservent la facilité de programmation et la portabilité des applications sur des systèmes à mémoire distribuée, sans imposer pour autant la gestion des communications par le concepteur. Les systèmes DSM, permettent une modification relativement simple et une exécution efficace des applications déjà existantes sur des systèmes à mémoire partagée, tout en héritant de la flexibilité des systèmes à mémoire distribuée [1].

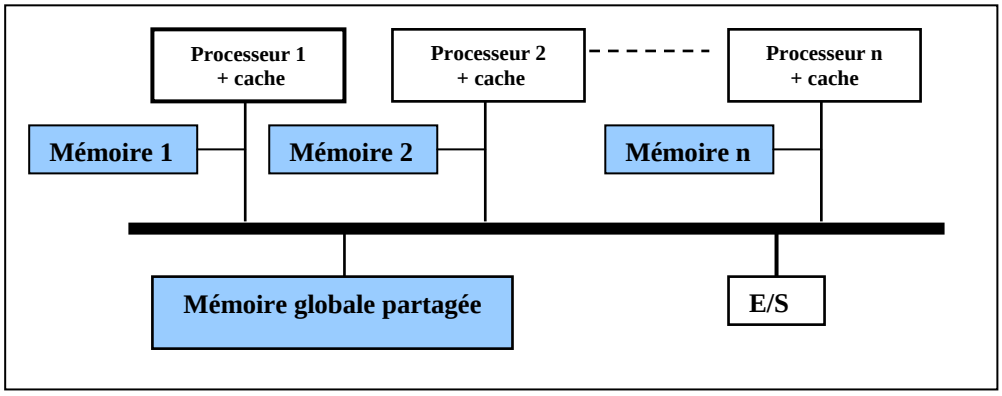

Figure 4: architecture à mémoire distribuée partagée

Un système multiprocesseurs avec mémoire distribuée partagée est généralement constitué d'un ensemble de nœuds (clusters), connectés par un réseau d'interconnexion Figure4. Un nœud peut être soit un simple processeur ou une hiérarchie qui cache une autre architecture multiprocesseurs, souvent organisée autour d'un bus partagé. Les caches privés aux processeurs sont d'une grande importance afin de réduire la latente. Chaque nœud possède un module de mémoire local (physiquement), faisant partie du système DSM global, ainsi qu'une interface le connectant au système.

3- Système d'exploitation temps réel

3.1- RTOS et contrainte temps réel

Un système est dit temps réel lorsqu'il est soumis à des contraintes de temps et qu'il y répond dans un intervalle acceptable. Il n'est pas nécessairement rapide, tout dépend des contraintes imposées par l'application [1, 2].

[10] *Distributed shared Memory*

Parmi l'ensemble des systèmes temps réels existants, nous distinguons deux grandes familles: les systèmes à contraintes souples et les systèmes à contraintes dures [36].

3.1.1- Systèmes à contraintes souples

Ils acceptent des variations de durée quant aux traitements des données. Un système de visioconférence, par exemple, est de ce type. Avec ce type de système, il est souhaitable que les images soient affichées à une cadence de 30 images par seconde. Cependant, si l'ensemble du système subit une forte charge de travail, certaines images pourront être supprimées afin de garder une cohérence avec le son.

3.1.2- Systèmes à contraintes dures

Ils n'acceptent aucun compromis sur la durée de traitement des données. Ce type de système se trouve, par exemple, dans les centrales nucléaires. Si un réacteur vient d'avoir un disfonctionnement, le système devra être capable de déclencher un processus de sécurité dans un délai extrêmement court.

En outre, pour le système de gestion des airbags dans une voiture, il possède des contraintes temps réel très importantes. En fait, quand un capteur détecte une déformation de la carrosserie, il envoie un signal au contrôleur qui doit avoir gonflé les airbags dans les 10 ms sous peine d'arriver trop tard, ce qui aurait des conséquences désastreuses pour les occupants de la voiture. Même si un capteur tombe en panne et qu'il n'envoie plus d'informations, le système doit continuer à fonctionner et fournir les meilleures réponses possibles.

3.2- Caractéristiques d'un RTOS

Un système d'exploitation temps réel doit s'affranchir des incertitudes sur le temps. Si une tâche ne peut être effectuée immédiatement, elle devra l'être au bout d'un temps « *t* » connu.

Un tel système possède donc la caractéristique d'être déterministe. Il apporte aussi les services suivants :

- Communication;
- Synchronisation;
- Gestion et ordonnancement des tâches;
- Gestion de la mémoire et du temps;
- Gestion des interruptions et des entrées/sorties physiques.

3.3- Concepts de base d'un RTOS

Dans ce paragraphe, nous rappelons quelques concepts clés concernant les systèmes d'exploitation temps réel [38]:

3.3.1- Section critique

Une section critique de code (appelée aussi une région critique) doit être traitée continuellement. Une fois, la section de code commence l'exécution, elle ne devra pas être interrompue. Afin d'assurer cela, les interruptions sont généralement désactivées avant que le code critique ne soit exécuté mais on les activera de nouveau quand le code sera achevé

3.3.2- Ressource partagée et exclusion mutuelle

Une ressource partagée est un objet qui peut être utilisé par plusieurs parties du programme. Cette ressource peut être un registre, une variable, une structure de données, ou quelque chose physique comme un LCD, ou un beeper.

Si deux parties séparées ont besoin de la même ressource, on devra gérer cela par l'exclusion mutuelle. A chaque fois, qu'une partie du programme veut utiliser une ressource partagée, il faudrait obtenir un accès exclusif à cette ressource afin d'éviter le conflit.

3.3.3- Tâche

3.3.3.1- Multi-tâches

Le multitâche maximise l'utilisation du CPU, et fournit une construction modulaire des applications. En plus, il permet au programmeur de l'application de gérer la complexité inhérente dans les applications temps réel. Les programmes d'application sont typiquement plus faciles à concevoir et à maintenir si le multitâche est mis en œuvre.

Un système temps réel multitâche doit avoir les caractéristiques suivantes [3] :

- Plusieurs tâches doivent être exécutées périodiquement et à des intervalles différents;
- Une tâche peut s'exécuter avec une faible priorité de façon à garantir les contraintes de temps des autres tâches.
- Une tâche peut communiquer de l'information à une autre.

3.3.3.2- Ordonnancement

3.3.3.2.1- <u>Ordonnancement préemptif</u>

La préemption se produit quand une tâche jugée plus prioritaire que la tâche courante apparaît et devient éligible pour s'exécuter. Tant que la préemption peut se produire à tout moment, elle exige donc l'utilisation des interruptions et la gestion de la pile pour garantir l'exactitude du changement de contexte. Par une neutralisation temporaire de la préemption, les programmeurs peuvent empêcher les ruptures non désirées dans leurs programmes pendant les sections critiques du code [5].

L'ordonnancement préemptif est très « *stack-intensive* ». En fait, l'ordonnanceur maintient une pile séparée pour chaque tâche. Ainsi, quand une tâche suspendue reprend l'exécution après le changement de contexte, toutes les valeurs de la pile, uniques et propres pour cette tâche, sont remises en place. Elles sont généralement les adresses de retour des appels des sous-routines, les paramètres et les variables locales.

3.3.3.2.2- Ordonnancement coopératif

L'ordonnanceur coopératif est susceptible d'être plus simple que celui préemptif. Comme les tâches devraient toutes se coopérer pour que le changement de contexte se produise, l'ordonnanceur est alors moins dépendant des interruptions, et il peut être plus petit et potentiellement plus rapide. En plus, les programmeurs connaissent exactement quand les changements de contexte vont se produire, et peuvent donc protéger les régions critiques du code [5].

En le comparant avec l'ordonnanceur préemptif, l'ordonnanceur coopératif possède certains avantages tels que sa simplicité relative, son contrôle total au changement de contexte et un temps de réponse de l'interruption plus court.

3.3.3.3- Changement de contexte

Quand un noyau décide d'exécuter une autre tâche, il sauvegarde simplement le contexte de la tâche courante (registres du CPU) dans sa propre pile. Une fois cette opération est effectuée, le contexte de la nouvelle tâche reconstituée de sa zone de stockage reprend alors l'exécution de son code. Ce processus est appelé un changement de contexte (*Context Switch*). Ce contexte représente l'état du processeur à un moment donné [4]:

- Registre: La tâche suspendue devra pouvoir continuer son exécution sans être affectée. La première opération à effectuer est la sauvegarde de l'état du processeur au moment de la suspension. Le noyau possède pour chaque tâche non dormante un espace mémoire réservé à cet effet.

- Pile: Les données temporaires utilisées par la tâche suspendue doivent être préservées lors des opérations de la nouvelle tâche active. Ces données sont organisées sous la forme d'une pile contenant le contexte (adresse de retour, valeur des registres), des appels de sous-routines et les variables temporaires de ces sous-routines.

3.3.3.4- Structure d'une tâche

Généralement, une tâche est une opération qui a besoin de se produire à plusieurs reprises dans l'application. La structure est réellement très simple. Elle se compose d'une initialisation optionnelle et d'une boucle infinie qui se répète inconditionnellement.

Avec un ordonnanceur préemptif, une tâche peut se voir comme suit [5]:

```
Initialize() ;
For ( ;;)
 {
    ……..
  }
```

Structure d'une tâche pour un multitâche préemptif

Quant à l'utilisation d'un ordonnanceur coopératif, une tâche peut se voir comme suit:

```
Initialize() ;
For ( ;;)
{…..
 TaskSwitch( ) ;
 ……}
```

Structure d'une tâche pour un multitâche coopératif

La seule différence entre les deux versions est le besoin d'exiger explicitement le changement de contexte dans la version coopérative. Dans le multitâche coopératif, chaque tâche décide elle-même d'abandonner le contrôle du processeur pour une autre tâche.

Mais, dans le multitâche préemptif, l'ordonnanceur procède à un changement de contexte juste à l'apparition d'une tâche de priorité plus élevée éligible pour s'exécuter. Nous pouvons noter aussi que les changements de contextes peuvent se produire à des temps multiples à l'intérieur d'une tâche, dans les deux types de systèmes coopératifs et préemptifs.

3.3.3.5- Etat d'une tâche

Un RTOS maintient chaque tâche dans un état bien défini. La figure 5 illustre les différents états qu'une tâche peut être dedans, et les transitions permises entre les différents états [18].

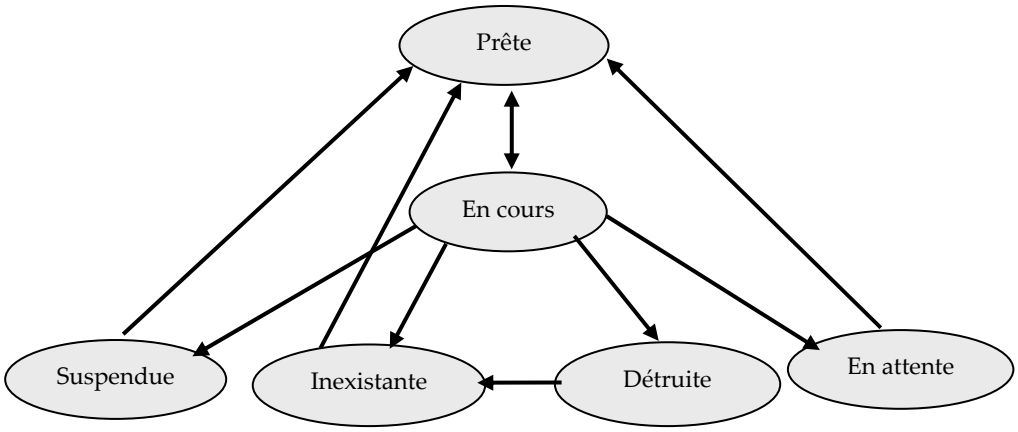

Figure 5: Etats d'une tâche

▪ Avant qu'une tâche ne soit crée, elle est dans un état « *Inexistante* ». Elle retournera à cet état quand elle sera détruite.

▪ Une tâche « *Prête* » est à l'état prêt pour s'exécuter, mais elle ne le peut pas car elle n'est pas la tâche la plus prioritaire. Elle devra donc rester dans cet état jusqu'à ce que l'ordonnanceur détermine quelle est la plus prioritaire et la bascule à l'état « *En cours* ».

▪ Une tâche « *En cours* », c'est à dire en cours d'exécution, devra retourner à l'état éligible après un simple changement de contexte. Mais, elle peut transiter vers un autre état si la tâche appelle un service de RTOS qui détruit, stoppe, retarde, ou fait attendre la tâche.

▪ Une tâche « *Suspendue* » est une tâche qui était précédemment en cours d'exécution, mais maintenant elle est suspendue et en attente de l'expiration du « *delay timer* ». Une fois, le temporisateur associé est expiré, le système d'exploitation bascule la tâche à l'état « *Prêt* ». Nous pouvons ainsi noter que les tâches périodiques sont susceptibles d'être « *Suspendue* » à tout instant particulier.

▪ Une tâche « *Inexistante* » est précédemment en cours d'exécution, et elle sera par la suite suspendue indéfiniment. Elle ne sera recommencée que via un appel d'un service RTOS permettant de provoquer sa regénèrissance.

▪ Une tâche « *Bloquée* » est suspendue ; elle restera en attente jusqu'à la production de l'événement désiré.

3.3.4- Communication entre tâches

Un RTOS fournit diverses méthodes pour faire communiquer les tâches entre elles. Dans le multitâches à base d'événements, et pour qu'une tâche réagisse à un événement, ce dernier doit déclencher une sorte de communication avec la tâche.

Les tâches peuvent aussi coopérer l'une avec l'autre par divers outils de communication comme les sémaphores, les messages, et les queues de messages. Ainsi nous distinguons deux principales actions [18]:

- Signalisation; appelée aussi envoi (*Posting*).
- Attente; appelée aussi réception (*Pending*).

3.3.4.1- Sémaphores

Il y a deux types de sémaphores : les sémaphores binaires (*Binary Semaphore*) et les sémaphores compteurs (*Counting Semaphore*). Un sémaphore binaire peut prendre uniquement deux valeurs, 0 ou 1. Un sémaphore compteur peut prendre une gamme de valeurs basées sur sa taille. Par exemple, la valeur d'un sémaphore compteur à 8 bits peut s'étendre de 0 à 255. Il peut aussi être à 16 ou 32 bits. Les sémaphores et leurs valeurs se présentent généralement de la façon suivante figure 6 :

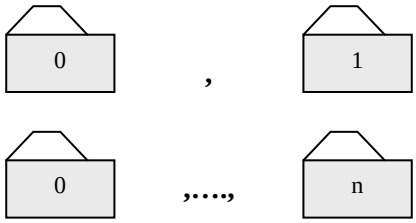

Figure 6: Sémaphores binaires et compteur

3.3.4.2- Messages

Les messages fournissent un moyen arbitraire d'envoi d'informations à une tâche. L'information peut être un nombre, une chaîne, un tableau, une fonction, un pointeur, ou toute autre chose. Comme avec les sémaphores, le système d'exploitation fournit le moyen pour créer, signaler, et attendre les messages.

Actuellement, le contenu du message n'est pas l'information elle-même, mais plutôt un pointeur indiquant l'emplacement de l'information. Quand un message est initialisé d'être vide, il contient donc un pointeur nul, qui ne pointe à aucune chose.

3.3.4.3- Files de communication (message queue)

Les files de messages sont une extension de messages. Une queue de messages peut contenir des messages multiples (jusqu'à un nombre prédéterminé) à tout moment. L'envoi des messages peut continuer jusqu'à ce que la boîte de messages soit pleine. La réception aussi peut continuer jusqu'à ce qu'elle soit vide.

Un RTOS aura besoin d'allouer une certaine *RAM*[11] additionnelle pour gérer chaque queue de messages. Cette RAM sera utilisée pour garder la trace du nombre de messages dans la queue, et l'ordre dans lequel les messages sont envoyés.

3.4- Conflits

Une variété de conflits peut se produire dans un environnement multitâche. Les plus connus sont les suivants: l'interblocage et l'inversion des priorités [5].

3.4.1- Interblocage (Deadlock)

Le « *Deadlock* » se produit avec deux ou plusieurs tâches, quand chaque tâche est en attente sur une ressource contrôlée par une autre. Cette ressource resterait non disponible indéfiniment. Les tâches en attente seront donc abouties à une impasse. La solution, pour toutes les tâches désirant acquérir des ressources, est de :

- Acquérir souvent des ressources dans un ordre prédéterminé.
- Acquérir toutes les ressources avant de continuer.
- Libérer les ressources dans un ordre opposé.

En utilisant un « *timeout* », nous pouvons éviter le « *Deadlock* ». En fait, en essayant d'obtenir la ressource, une période de temps optionnelle peut être spécifiée. Si la ressource n'est pas acquise dans une telle période de temps, la tâche continue mais avec un code d'erreur indiquant que le temps d'attente de la ressource a expiré.

3.4.2- Inversions des priorités

Les inversions de priorité se produisent quand une tâche de haute priorité est en attente sur une ressource commandée par une autre de basse priorité. La tâche la plus prioritaire devra attendre jusqu'à ce que la tâche de basse priorité libère la ressource, sur laquelle elle peut continuer. Pour ce, la priorité de la tâche la plus prioritaire doit être réduite à celle de la tâche de basse priorité.

Il y a une variété de méthodes pour éviter ce problème (exemple, transmission prioritaire). La méthode la plus pratique consiste à changer dynamiquement la priorité de la tâche qui commande une ressource. Cette méthode est donc basée sur les priorités des tâches désirant acquérir cette ressource [18].

[11] Random Access Memory

3.5- Systèmes d'exploitation dans les systèmes embarqués

L'usage de systèmes d'exploitation est nécessaire dans les systèmes embarqués, du fait de la complexité croissante de ces systèmes (exemple : Systèmes sur puce), de la présence de fortes contraintes temps réel, de la limitation des ressources disponibles, tant en mémoire qu'en énergie disponible et donc en puissance de calcul, mais également de la pression exercée par le marché sur ces produits. En effet, le temps de développement doit être raisonnable, afin de limiter le temps de mise sur le marché (*time to market*), et permet ainsi d'assurer le succès du produit.

Les systèmes d'exploitation à micronoyaux sont mieux adaptés aux systèmes embarqués [9]: la tolérance aux fautes doit être forte, car les conditions environnementales ne sont pas toujours optimales (exemple: systèmes pour l'automobile ou l'industrie). Ceci est permis par la possibilité de redondance, ainsi que le confinement des erreurs, qui est bien meilleur dans un système d'exploitation à micronoyau que dans un système d'exploitation plus classique. Par ailleurs, il est également possible de charger dynamiquement les modules à exécuter, ainsi que de les distribuer, ce sont des besoins forts pour ce type de système. La figure 7 montre l'architecture d'un tel système d'exploitation.

Quelques exemples d'implémentation de Systèmes d'Exploitation pour Systèmes Embarqués [9]:

- *Inferno* (Lucent) : avec sécurité intégrée,
- *VxWorks*, version générique,
- VxWorks, pour l'automobile,
- VxWorks, pour l'électronique grand public,
- VxWorks, pour l'industrie,
- *pSOS* Systems,
- Lynx Real-Time Systems (*LynxOS*).

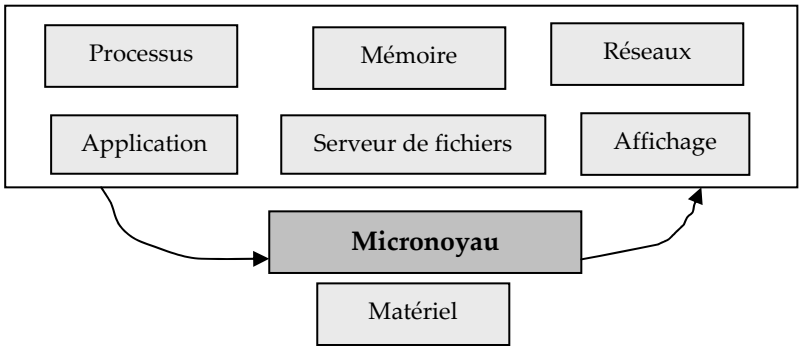

Figure 7: Architecture d'un SE à micronoyau

Parmi les Systèmes d'Exploitation pour les Systèmes Embarqués, nous distinguons les systèmes d'exploitation temps réel (RTOS) qui se caractérisent par la présence de contraintes temps réel.

3.5.1- Caractéristiques des RTOS pour les systèmes embarqués

Les caractéristiques essentielles d'un RTOS pour les systèmes embarqués sont:

- La première caractéristique des RTOS est son temps de réponse prévisible à un stimulus externe [13]. Un tel système possède donc la caractéristique d'être déterministe.
- Un RTOS doit s'affranchir des incertitudes sur le temps. Si une tâche ne peut être effectuée immédiatement, elle le devra au bout d'un certain temps « *t* » connu [14].
- Si un périphérique génère une interruption, le RTOS doit répondre et démarrer le service à l'intérieur d'une période de temps connu, et ce, peu importe la charge du processeur sur lequel s'exécute le RTOS [13].
- Dans l'industrie, on s'entend dire de façon générale qu'un OS est RTOS lorsque le changement de contexte et le temps de réponse à une interruption sont garantis à l'intérieur d'une période de 10 µs [13].
- Finalement, un bon RTOS doit aussi supporter les mécanismes d'un OS distribué (permet ainsi l'exploitation des systèmes temps réel distribués), mécanismes d'ordonnancement [15] (préemptif, non préemptif ou coopératif), précision de l'horloge et des minuteries, les outils de visualisation, et la compatibilité POSIX.

3.5.2- Importance des RTOS pour les systèmes embarqués

Depuis plusieurs années des systèmes d'exploitation temps réel (RTOS) sont introduits dans les architectures embarquées [13] monoprocesseurs et multiprocesseurs, telles que les architectures réactives embarquées afin de gérer la réactivité du système. En fait, la présence d'un RTOS permet de:

- Structurer et simplifier la programmation de la partie logicielle du SoC. En effet, le RTOS gère lui-même le matériel et propose aux applications logicielles des fonctions d'accès de haut niveau. Ainsi, le travail du programmeur d'application est soulagé de la programmation des accès au matériel.
- Utiliser des spécificités des processeurs. En effet, Les systèmes d'exploitation spécialement programmés pour le processeur sur lequel ils vont s'exécuter, peuvent tirer parties de ses spécificités en ce qui concerne le mécanisme d'interruption, les instructions de réduction de consommation et de gestion de cache [16].

- Et le réutiliser comme un IP logiciel dans les systèmes embarqués.

Grâce au RTOS, la réutilisation du point de vue logiciel peut se faire à plusieurs niveaux [13]:

- Gestion des tâches dans un système temps réel multitâches (notion de priorité, ordonnancement, changement de contexte, etc.);
- gestion des interruptions et des entrées/sorties physiques;
- gestion du temps (minuteries);
- gestion des communications entre tâches;
- gestion de la mémoire, etc.

3.5.3- Limites des RTOS dans les systèmes embarqués

Il est courant d'utiliser un système d'exploitation temps réel dans un tel système embarqué comme étant une structure logicielle permettant de gérer l'exécution complète de plusieurs tâches concurrentes sur le même système. Cependant, et bien que cette méthode soit couramment employée dans les systèmes embarqués spécifiques, celle-ci peut entraîner certains inconvénients en terme de coût, consommation et performances. Parmi ces limites, nous citons [16] :

- Les systèmes d'exploitation consomment de la mémoire. En plus, ils sont spécifiques au processeur sur lequel ils s'exécutent.

- Les impératifs de performances empêchent souvent l'utilisation d'interfaces génériques abstraites, et la multitude des systèmes d'exploitation et des architectures sont des freins à l'uniformité des interfaces.

- La généralité du système d'exploitation vis à vis de l'application faite est qu'il soit souvent plus volumineux que nécessaire. C'est un défaut important dans le monde des systèmes embarqués où la mémoire est limitée.

- La vitesse du système d'exploitation est aussi limitée par l'ordonnancement dynamique des tâches qui demande du temps aussi bien pour la décision que pour le passage d'une tâche à l'autre.

- Les systèmes d'exploitation peuvent être non déterministes : En fait, il est souvent impossible de savoir, avant utilisation, si une application basée sur un système d'exploitation respectera des délais ou non.

4- Quelques exemples de noyaux embarqués

Nous présentons dans la suite de cette section deux systèmes d'exploitation embarqués pour chaque catégorie. Nous rappelons ainsi les caractéristiques, et les services de chacun de ces systèmes d'exploitation. Cette étude est inspirée de [1].

4.1- RTEMS

RTEMS[12] est un noyau temps réel qui fournit de hautes performances pour les applications embarquées militaires. En plus, de point de vue code, il est très complet et assez lourd par rapport à certains autres RTOS. En fait, le taux d'occupation de cet RTOS est généralement très grand (200Mo). Les architectures supportant RTEMS sont les suivantes : M68k, Coldfire, Hitachi SH, Intel i386, Intel i960, MIPS, PowerPC, SPARC, AMD A29k et HP PA-RISC.

4.1.1- Architecture interne de RTEMS

Comme indiqué dans la figure 8, RTEMS peut être envisagé comme un ensemble de composants superposés en couches qui travaillent en harmonie pour produire un ensemble de services pour des applications temps réel.

L'interface exécutive de l'application est formée par des groupes de directives constituant *le manager de ressources*. Les fonctions utilisées par les différents managers tels que l'ordonnanceur (*scheduler*), le partitionneur (*dispatcher*), et la gestion de l'objet sont intégrées dans le noyau exécutif. Ce dernier est en relation avec le « *code dépendant du CPU*».

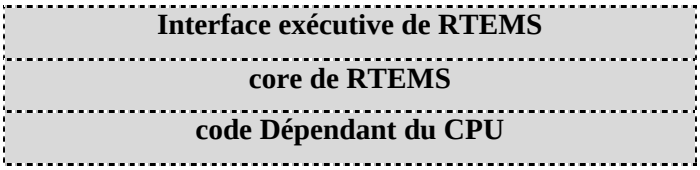

Figure 8: Organisation de RTEMS

4.1.2- Architecture d'une application RTEMS

RTEMS se sert d'un Bridge (pont) entre les deux couches critiques : celle de l'architecture matérielle cible et celle du code de l'application dépendante. La plupart des

[12] *Real Time Executive for Multiprocessors Systems*

opérations matérielles dépendantes de l'application temps réel peuvent être localisées dans la couche la plus basse du gestionnaire des périphériques.

Le manager de l'interface I/O de RTEMS présente un outil efficace pour l'incorporation des dépendances matérielles dans le système en fournissant un mécanisme général au code de l'application qui y accède. Un système temps réel bien conçu peut bénéficier de cette architecture en construisant une bibliothèque riche en composants pouvant être utilisée dans d'autres applications ou projets temps réel.

4.1.3- Services de RTEMS

- Les services primaires : Erreur Fatale, Initialisation, Tâche, horloge et temporisateur, Interruption ;
- Les services de Communication et de Synchronisation : Signal, Sémaphore, Message et Evénement ;
- Le multiprocessing ;
- I/O;
- Rate Monotonic;
- Les services de mémoire : Partition, Région, Dual-ported memory et I/O ;

4.1.4- Ordonnancement

L'ordonnanceur de RTEMS utilise l'un des algorithmes suivants :

- A base de priorité (*priority-based*),
- Préemptif,
- Méthode de tourniquet (*round-robin*),
- Par partage de temps (*Time slicing*).

Il est à noter que l'ordonnanceur de RTEMS choisit toujours la tâche prête de plus haute priorité.

4.1.5- Caractéristiques et capacités

RTEMS inclut les aspects suivants :

- Le multitâche ;
- Des systèmes supportant des processeurs homogènes ou hétérogènes ;
- Pilotage par évènement (*Event-driven*) et en fonction de la priorité (*Priority-based*) ;
- Ordonnancement préemptif (*Preemptive scheduling*);
- Ordonnancement Rate Monotonic (*RMS*);

- Communication et synchronisation entre tâches;
- L'héritage de priorité ;
- Gestion des interruptions ;
- Allocation dynamique de la mémoire.

4.1.6- Aspect multiprocesseurs

RTEMS supporte le multiprocesseurs. La communication inter processeur est gérée par la couche MPCI non complète (*MPCI*[13]).

4.2- Ecos

Ecos est un noyau à code source ouvert, développé par RedHat. L'objectif principal d'Ecos est de fournir aux développeurs embarqués une infrastructure logicielle commune pour délivrer une gamme diverse de produits embarqués.

La nature de configuration d'Ecos permet à ce système d'exploitation d'être personnalisé aux exigences de l'application, en délivrant de meilleures performances en temps d'exécution (*run time*) et une occupation de ressource matérielle optimisée. Ecos est visé aux applications à fort débit dans l'électronique grand public, les télécommunications, les véhicules à moteur, et les applications profondément embarquées. Dans la version 1.3.1 d'Ecos par exemple, le OS occupe 3K de ROM et 1K de RAM [21].

Les plus adoptés sont: Motorola PowerPC, Intel strong ARM, Advanced RISC Machines ARM7, NEC VR4300, MB8683 * series, Hitachi SH3, Toshiba TX39, Matsushita MN10300, Fujitsu SPARClite, etc.

4.2.1- Caractéristiques et capacités

- Conception modulaire pour la configuration au niveau source.
- Un ensemble riche en primitives de synchronisation.
- Choix des algorithmes d'ordonnancement.
- Choix de la stratégie d'allocation de la mémoire.
- Des horloges et compteurs.
- L'acceptation des interruptions et des communications.
- Traitement des exceptions.
- Une bibliothèque C conforme à la norme ISO.
- Une bibliothèque mathématique.

[13] *Multiprocessor Communication Interface*

- La portabilité : Ecos et ses composants sont liés à HLA. Ainsi, ils s'exécutent sur n'importe quelle cible, une fois le HAL et ses pilotes sont portés sur l'architecture cible.

4.2.2- HAL

Ecos inclut une couche d'abstraction du matériel (***HAL***[14]), qui cache les traits spécifiques du CPU et des différents dispositifs « *On Chip* » de la plate-forme supportée. Ainsi le noyau et les autres composants peuvent être implémentés de façon portative. Cette abstraction concerne exactement : le changement de contexte, la réaction suite à une interruption matérielle et l'accès aux registres.

Il y a trois couches sur lesquelles fonctionne le HAL :

- *L'architecture :* est le premier sous module de HAL. Chaque famille de processeurs supportée par eCos est considérée comme une architecture différente. Chaque sous module d'architecture contient :
  - le code nécessaire pour le démarrage du CPU,
  - la livraison des interruptions,
  - le changement de contexte,
  - et autres fonctionnalités spécifiques à l'architecture.
- *La variante :* est le deuxième sous module de HAL. Elle représente un processeur spécifique dans la famille de processeurs décrite par l'architecture.
- *La plate –forme :* est le troisième sous module de HAL. Une plate-forme est une partie spécifique du matériel qui inclut l'architecture à base de processeur choisi, et même la variante. Typiquement, ce module inclut le code pour le démarrage de la plate-forme, la configuration sélectionnée de chip, les contrôleurs des interruptions, et les dispositifs de temporisateur.

4.2.3- Détails du noyau

Ce noyau présenté dans la figure ci-dessous est composé des éléments suivants : L'application mise en jeu.

- Des librairies écrites en C, utilisées par le noyau lors de la compilation.
- Les pilotes et HLA qui coopèrent ensemble afin d'assurer l'abstraction du matériel et le portage.
- Le noyau qui gère les tâches, la communication, la synchronisation, les compteurs et les interruptions.

[14] *Hardware Abstraction Layer*

- Le debug : c'est un support débogueur du multitâche.
- IRON : c'est une couche qui assure l'utilisation efficace des hétérogénéités présentes dans les systèmes embarqués.

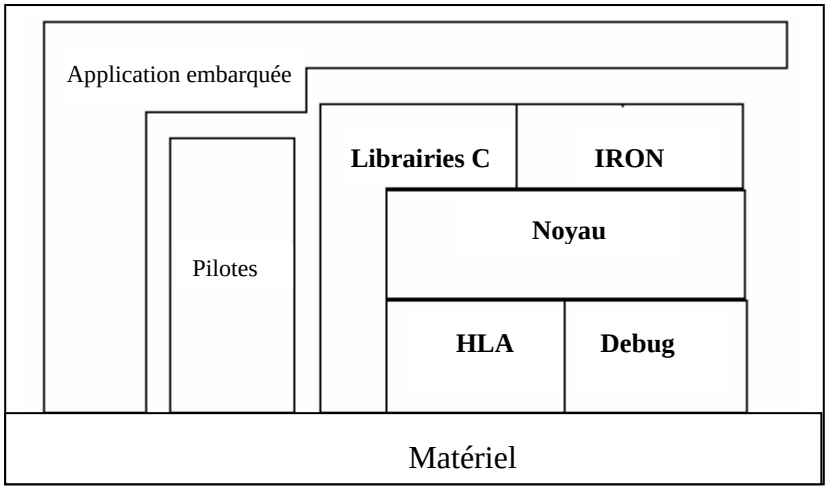

Figure 9: Les éléments d'Ecos

4.2.4- Ordonnancement

Ecos soutient deux différents ordonnanceurs qui mettent en application des politiques distinctes. Les deux ordonnanceurs sont les suivants:

- *Bitmap Scheduler:* il permet l'exécution des threads à des niveaux de priorité multiples. Cependant, un seul et simple thread peut exister à chaque niveau de priorité. Ceci facilite l'algorithme d'ordonnancement, et rend l'ordonnanceur bitmap très efficace. Le nombre de niveaux de priorité est au maximum 32 : 0 correspond à la priorité la plus élevée et 31 la plus basse. Chaque niveau de priorité est représenté par un bit.
- *Multi-level Queue Scheduler:* Le nombre de niveaux de priorité est au maximum 32 : 0 correspond à la priorité la plus élevée et 31 la plus basse. Un même niveau de priorité peut être attribué à plusieurs tâches en même temps. Cet ordonnanceur permet :
  - La préemption entre les différents niveaux de priorité.
  - Le soutien du *SMP*[15].
  - Le partage du temps sur un même niveau de priorité.

4.2.5- Avantages d'Ecos

- *La configurabilité :* Ecos est désigné comme une architecture composante et configurable formée par plusieurs composants logiciels principaux. La nature de configuration d' Ecos nous permet de sélectionner les différentes options nécessaires dans un

[15] *Symmetric Multi-Processing*

composant logiciel, ou/et de supprimer des composants tout à fait inutilisables afin de créer un système spécialement conçu pour répondre aux exigences de notre application. Par conséquent, la taille de l'application devient compacte et plus rapide comprenant seulement les composants à utiliser.

- *La portabilité :* Ecos est désigné être portatif sur une large gamme d'architectures cibles en incluant les architectures à 16, 32 et 64 bits, les microcontrôleurs et les DSPs. La raison permettant à Ecos d'être fortement portatif est l'implémentation de HAL.

4.3- QNX

QNX est un système temps réel de type *UNIX* développé par la société canadienne « *QNX Software* ». Il est conforme à *POSIX*, il permet de développer directement des applications sur la plate-forme cible, et il intègre l'environnement graphique « *Photon* », proche du système Xwindows.

Le point fort de QNX réside dans son architecture à micronoyau préemptif. Comme son nom peut déjà le laisser présumer, QNX a pour domaine de prédilection les applications temps réel dans lesquelles un nombre d'événements doit être géré dans un laps de temps déterminé et garanti. QNX se voit donc doté d'une architecture dans laquelle le noyau du système est réduit à sa plus simple structure. L'unique tâche de ce dernier est de gérer un ensemble de processus de même priorité et inter communiquant entre eux. Graphiquement, cette architecture se résume de la façon suivante figure 10:

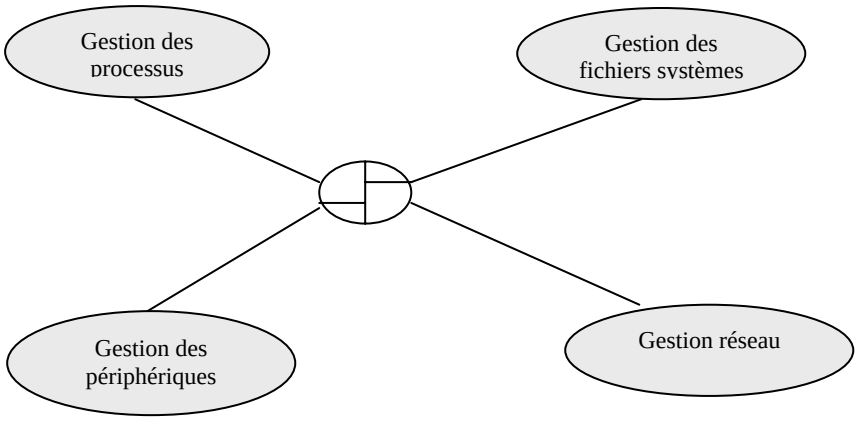

Figure 10: Gestion de processus par Micro-Kernel

Le micronoyau est au centre de cette structure. Il joue le rôle de passerelle entre les différents processus en cours. Il n'occupe que *10 Ko* de mémoire. Ainsi, il peut être tenu dans de nombreuses applications embarquées.

Cette compacité extrême du noyau ainsi que cette structure permet à QNX une certaine adaptabilité. En effet, il est envisageable de faire fonctionner le système sans avoir besoin de certains de ces gestionnaires comme par exemple dans le cas des applications embarquées où l'interface graphique est tout bonnement enlevée du système.

Selon les besoins, les développeurs n'ont alors plus qu'à supprimer ou rajouter des modules au micronoyau pour former le système le plus adapté à ce que nous souhaitons réaliser; c'est la grande force de QNX.

4.3.1- Caractéristiques techniques

- Système de type UNIX respectant les spécifications POSIX.
- QNX supporte les architectures MIPS, PowerPC, SH4, StrongArm et x86.
- Compatible POSIX.
- Supporte le multiprocesseurs et le multitâche.
- Reconnaît les partitions FAT 16 et FAT 32, ext2, UNIX, OS/2 HPFS, NFS, ainsi que les partitions issues des versions 1.x à 4.x de QNX.
- Interpréteur de commandes standard dans le monde UNIX

4.3.2- Caractéristiques de spécification de QNXv6.1

Le tableau 2 ci-dessous présente les principales caractéristiques de spécification concernant le système d'exploitation QNX v6.1 [22].

Critères de classification	**QNX v6.1**
ROM-RAM Footprint	12 Ko
Caractéristiques	Le RTOS QNX supporte le multiprocesseurs, le traitement distribué transparent, la gestion de réseau avec tolérance aux défaillances.
Modèle	Threads et Processus.
Priorité	*64* niveaux
Nombre maximum de tâches	*4095* processus. Chaque processus peut avoir *32 767* threads.

Politique d'ordonnancement	- FIFO prioritaire - Ordonnancement Round-Robin - Adaptatif - Sporadique
Soutien MMU[16]	Oui
Taille de la page physique	Dépend de l'architecture supportée.
Mémoire virtuelle	Chaque processus s'exécute dans son propre espace mémoire virtuelle.
Modèles de protection de la mémoire	Protection de la mémoire virtuelle.
Contexte	L'ISR s'exécute dans le contexte du thread qui lui est attaché.
Pile	L'ISR a sa propre pile.

Tableau 2: Caractéristiques de QNXv6.1

4.4- Windows CE

Windows CE prend en charge une multitude de fonctionnalités et peut être compilé sur une grande variété de processeurs. Il supporte un environnement multitâche et inclut, en option, une interface utilisateur graphique (GUI). Windows CE reprend l'architecture de la famille Windows, de sorte que tout programmeur de Windows peut facilement passer à la programmation de Windows CE.

Windows CE 3.0 est un système d'exploitation embarqué modulaire et temps réel pour la configuration 32bits légère. Il combine la compatibilité Windows et les services d'applications avancées. Il supporte aussi de multiples architectures à base de processeurs, ainsi que des options de communication et de réseaux. Il permet donc de construire un système adaptable pour développer une large gamme d'équipements.

Cet OS soutient des terminaux Web, des contrôleurs industriels spécialisés, des équipements d'acquisition de données portables et des appareils communicants embarqués. Cette plate-forme particulièrement modulaire permet aux développeurs de concevoir des configurations 32-bits légères compatibles Windows et Internet.

[16] *Memory Management Unit*

Le tableau 3 ci-dessous présente les principales caractéristiques de spécification concernant le système d'exploitation Windows CE [22] :

Critères de classification	**Windows CE**
ROM-RAM Footprint	*400 Ko* de RAM au grosso modo *200 Ko* de ROM
PVM[17]	Fiabilité par la protection des services d'application critiques.
Processus et Threads	Le nombre maximum de threads dans un processus est seulement limité par la quantité disponible de mémoire. Ce processus peut exécuter simultanément au maximum *32* processus.
Priorité	*256* niveaux
SIM[18]	Réponses très rapides face aux événements avec les ISRs.
Advanced Power Management	Batterie longue durée et dissipation de chaleur réduite.
Support pour le débogage "On-Chip"	Autorise le débogage de l'OAL avant que le noyau de l'OS ne se mette en route.
Politique d'ordonnancement	Round Robin en adoptant un quantum (time slice). Quand ce quantum prend la valeur 0, le thread s'exécute alors jusqu'à son achèvement.
Soutien MMU	Oui

[17] *Protected Virtual Memory*
[18] *Sophisticated Interrupt Management*

Taille de la page physique	Dépend de l'architecture supportée.
Mémoire virtuelle	Oui
Modèles de protection de la mémoire	Protection de la mémoire virtuelle.
Contexte	L'ISR s'exécute dans un contexte spécial, et utilise des adresses virtuelles statiquement tracées par l'OEM[19]. L'IST est un thread normal d'application, et il a son propre contexte.
Pile	L'IST est un thread normal d'application, et a son propre contexte.

Tableau 3: Caractéristiques de Windows CE

5- Estimation de performance dans les systèmes embarqués temps réel

La complexité des Systèmes sur Puce rend la place des RTOS de plus en plus importante. De plus, leur prix, qui reste relativement élevé, rend indispensable de bonnes performances et une bonne fiabilité, pour que ces systèmes soient compétitifs.

Parmi ces systèmes sur puce, nous citons les systèmes réactifs. En effet, Les systèmes réactifs embarqués tels que définis par Harel et Pnueli [10] sont des systèmes qui maintiennent une relation permanante avec leur environnement physique, à une vitesse déterminée par cet environnement.

Etant très contraints au niveau des ressources matérielles, ils doivent aussi réagir à des sollicitations de leur environnement en un temps fini et spécifié (contraintes temporelles) [11]. D'un point de vue logiciel, de très nombreux systèmes réactifs embarqués font appel à un ou plusieurs systèmes d'exploitation temps réel pour faciliter la gestion d'événements. Un système d'exploitation apporte une souplesse dans l'organisation du contrôle de l'application mais se traduit aussi par un surcoût en mémoire, en ressources de calcul, en consommation et en temps d'exécution. En fait, Le concepteur d'un SoC est donc confronté à de multiples choix pour architecturer son système tout en optimisant une fonction multicritères : performances, coûts, consommation, temps d'exécution et durée de conception (***Time to Market***) [12].

[19] *Original Equipment Manufacture*

L'objectif est, donc, de trouver une méthode d'estimation de performance des systèmes sur puce temps réel qui soit rapide et précise et qui s'intègre facilement dans un flot de codesign pour assister le concepteur dans son choix architectural.

5.1- Approches d'estimation du temps d'exécution :

Les méthodes d'estimations que l'on trouve dans la littérature peuvent être classées dans trois catégories : statiques, dynamiques et mixtes

Dynamique : les mesures de performance d'une solution est le résultat d'une analyse statique d'une spécification, (exemple : simulation).

Statique : l'estimation de performance d'une solution est le résultat d'une analyse statique d'une spécification, (exemple : analyse de chemins dans une spécification de flots de contrôle).

Mixte dynamique/statique : C'est l'utilisation de quelques éléments des deux approches précédentes pour l'analyse de performance d'une solution.

Les approches dynamiques sont en général très précises. Leur inconvénient majeur est le temps nécessaire pour l'obtention du modèle à simuler (synthèse, génération, compilation…), ainsi que le temps de la simulation. Ce qui les rend, en pratique, inutilisable dans le contexte particulier de l'exploration où le nombre de modèles à analyser est énorme. D'un autre coté, les approches statiques sont certes très rapides (pas de génération de modèles à simuler, ni de simulation), mais les tâches de modélisation et d'estimation sont complexes à cause de la distance qui sépare les concepts de spécification de l'implémentation.

5.1.1- Travaux visant des architectures cibles monoprocesseur :

Dans cette catégorie, on peut citer PMOSS [30], COSYMA [31] and LYCOS [32]. L'architecture cible est monoprocesseur (une seule unité de contrôle). Il n'y a donc pas de difficultés liées au parallélisme par rapport aux architectures multiprocesseurs. Cependant, les analyses de performance des parties logicielles et matérielles sont réalisées conjointement.

PMOSS se contente de calculer l'accélération due au coprocesseur (partie matérielle), sur la performance globale du système. Pour cela, il utilise, pour le logiciel, des analyses statiques (calcul du temps d'exécution basé sur le code assembleur) et dynamiques (profilage). Pour le matériel, il utilise des analyses statiques (calcul du temps d'exécution basé sur la description de la machine de contrôle). Et pour les communications, des analyses dynamiques (profilage), sont utilisées.

COSYMA calcule des métriques séparées pour le logiciel, le matériel et la communication. Ensuite, ces métriques sont combinées dans des équations particulières pour procéder à une partition basée sur une méthode de recuit simulé (simulated annealing). Des mesures de temps dans le pire cas sont calculées pour les implémentations logicielles en utilisant plusieurs variantes de techniques d'analyse de chemins. Le temps de communication est estimé pour son modèle particulier (mémoire partagée).

LYCOS procède à des estimations de performance en utilisant des techniques de profilage et d'estimations de temps d'exécution à bas niveau pour le matériel, le logiciel et la communication.

Malgré leur performance, ces méthodes ne permettent pas de traiter des architectures complexes pouvant contenir plus qu'un seul processeur.

5.1.2- Travaux visant des architectures cibles multiprocesseurs :

Dans cette catégorie nous trouvons SpecSyn [33], POLIS [34] et la méthode créée par Yen et al [35]. L'architecture cible est multiprocesseurs complexe.

SpecSyn admet des architectures avec un nombre quelconque de microprocesseurs et de coprocesseurs. L'approche utilisée pour l'estimation de performance est mixte statique/dynamique. Elle est faite en deux étapes :

- ***Pre-estimation*** : elle est réalisée avant la phase d'exploration d'architectures. Un profilage de la description du système est réalisé pour obtenir des temps d'exécution pour différents niveaux (processus, bloc de base, communication).
- ***Estimation en ligne*** : elle est faite durant la phase d'exploration d'architectures. Les résultats obtenus durant la phase de pre-estimation sont utilisés dans des expressions complexes pour le calcul de la performance globale du système.

Le problème d'une telle approche est son incapacité à capturer les changements dynamiques du comportement temporel durant la phase d'exploration d'architectures. Car durant cette phase, des méthodes statiques sont utilisées (le temps global est la somme des temps d'exécution partiels des différentes ressources d'exécution). Par exemple, cette méthode n'est pas capable d'estimer le temps d'attente d'un processus pour qu'un autre finisse son exécution. Le passage sur un tel comportement dynamique peut introduire une grande imprécision sur les résultats de l'estimation.

POLIS est capable de surmonter le problème mentionné ci-dessus (capture du comportement dynamique), en combinant une simulation de haut niveau avec des estimations de bas niveau (approche statique/dynamique).

Yen et al, attaquent le problème d'un point de vue générique. Ils analysent, au niveau système, l'interaction entre les différents processus en donnant le meilleur et le pire délai pour chacun d'entre eux. Ensuite, en partant d'un graphe acyclique représentant les dépendances de données entre les processus, et à l'aide d'informations sur le partitionnement (la distribution sur les unités de traitement), ils calculent le temps d'exécution, dans le pire cas, pour le système entier. Cette méthode est précise et capable de prendre en compte les délais de communications [37]. Malheureusement, elle est limitée à des applications pour lesquelles il est suffisant de connaître les délais dans le pire cas. De plus, les processus doivent être représentables par graphes acycliques.

6- Contribution

De nos jours l'usage des systèmes d'exploitation dans les systèmes sur puce est devenu indispensable et vu que ces systèmes ont subit des évolutions telle que l'assistance à l'intégration de plusieurs processeurs dans une seule puce, il a fallu trouver des méthodes pour adapter les systèmes d'exploitation existant pour les nouvelles architectures. Des études ont été faites dans notre équipe pour étendre un RTOS monoprocesseur par une couche de communication implémentée en logiciel afin de l'adapter aux architectures multiprocesseurs. Les travaux entrepris dans notre mastère consistent à remplacer cette couche implémentée en software par l'utilisation des modules d'un RTOS implémenté en hardware.

Suite à l'étude faite sur les outils d'estimation existant, on constate que la plupart des outils d'estimation ne tiennent pas en compte l'utilisation des systèmes d'exploitation temps réel. Alors afin d'exploiter et d'utiliser les outils d'estimation existant pour les systèmes sur puce nous proposons un modèle qui s'intégrera dans ces outils pour qu'ils puissent estimer le temps d'exécution des applications temps réel.

Notre modèle d'estimation consiste à construire une bibliothèque qui contient tous les services de l'RTOS ainsi que leurs temps d'exécution obtenu par l'exécution dans un environnement de prototypage, et, à chaque fois qu'on utilise un service dans l'application, on ajoute le temps approprié au temps global de l'application.

7- Conclusion

Dans ce chapitre, nous avons rappelé les différentes topologies possibles pour la réalisation d'une plateforme multiprocesseurs ainsi que les avantages et inconvénients de chacune d'elles. Nous avons détaillé aussi les caractéristiques et les principaux concepts temps réel des RTOS, ainsi que quelques exemples de RTOS embarqués. Nous avons terminé

ce chapitre par un état de l'art sur les travaux qui concernent les outils d'estimation de performances des systèmes sur puce.

Dans le chapitre suivant, nous nous intéressons à la présentation de notre environnement de travail et à la réalisation d'une plateforme de prototypage des systèmes sur puce multiprocesseurs temps réel, ainsi que la génération d'un modèle d'estimation de performance des applications temps réel.

Chapitre 2

Conception d'un MPSoC temps réel et estimation de performance

1-Introduction

Les architectures reconfigurables ont la possibilité d'offrir une performance comparable à celle d'un matériel dédié et une flexibilité comparable à celle d'un processeur à jeux d'instructions ce qui les rend très efficaces pour le prototypage des systèmes sur puce.

Dans ce chapitre, on envisage la conception d'un environnement de prototypage des systèmes multiprocesseurs temps réel reconfigurables à travers l'environnement d'altera et la génération d'un modèle d'estimation de performance pour les applications temps réel.
Ce chapitre est structuré de la façon suivante :

- en premier lieu, on présentera notre plateforme de travail,
- en deuxième lieu, on décrira l'architecture multiprocesseurs adoptée,
- en troisième lieu, on précisera la génération d'un modèle d'estimation de performance des applications temps réel.

2- Plate-forme de conception

Dans ce projet, on a utilisé le kit EXCALIBUR d'ALTERA qui est composé de :

- un cœur de processeur NIOS-II,
- un système d'exploitation temps réel embarqué *MicroC/OS-II*[20] qui était choisi pour être le support logiciel dans la conception du système réactif embarqué,
- une carte de développement à base du circuit FPGA de la famille *STRATIX-II*,
- et d'un environnement de développement *Quartus-II* .

Le choix de cette plate forme est justifié par la flexibilité donnée par l'environnement de conception et de prototypage des systèmes sur puce proposé par Altera « *QUARTUS II* », qui permet d'accélérer le processus de développement de l'application et par la présence de cet environnement dans notre équipe.

2.1- Etude du système d'exploitation temps réel : MicroC/OS-II

MicroC/OS-II, conçu et mis à point par Jean J. Labrosse, est un noyau temps réel permettant d'effectuer une exécution de plusieurs tâches sur un microprocesseur ou un microcontrôleur [23].

Ce noyau temps réel est maintenant disponible sur un grand nombre de processeurs, et il peut intégrer des protocoles standard comme TCP/IP (µC/IP) pour assurer une connectivité IP sur une liaison série par PPP. Les différentes versions de MicroC/OS-II sont portées sur des

[20] *Micro-Controller Operating System Version II*

systèmes différents : Motorola famille 680x0, 68HC11/16, Power PC 860, Intel 80x86, Philips XA, etc.

2.1.1- Capacités et caractéristiques

Les caractéristiques essentielles de ce noyau sont les suivantes :

- Ouvert, code source disponible [24],
- Portable, ROMable donc Encapsulable dans un produit,
- Fiable et robuste,
- Aux fonctionnalités ajustables,
- Multitâches et préemptif (l'Ordonnanceur de ce noyau contient seulement quatre lignes simples de code C [26]),
- Interruptible : traitement des interruptions Par les *ISR*[21],
- 63 tâches où chaque degré de priorité correspond à une seule tâche, c'est-à-dire deux tâches ne peuvent pas avoir le même degré de priorité,
- Changement de priorité des tâches (inversion et héritage de priorités),
- Fonction d'attente de tâche,
- Occupation optimale dans la mémoire: 2 Koctets taille du code [25],
- Création et gestion des sémaphores, des mutex, des mails box, des queues de messages et des drapeaux d'événements,
- Le temps d'exécution pour la plupart des services fournis par µC/OS-II est constant et « *déterministe* ».

2.1.2- Structure de MicroC/OS-II

Le système MicroC/OS-II peut être vu comme une bibliothèque de fonctions réparties sur des couches logicielles. Cette bibliothèque est liée avec l'application à développer. Ainsi, les services de MicroC/OS-II sont appelés depuis l'application comme de simples fonctions. Et comme le montre la figure 11, le code source de ce noyau est divisé en deux sections : la première est indépendante du processeur et la seconde en est dépendante.

[21] Interrupt Service Request

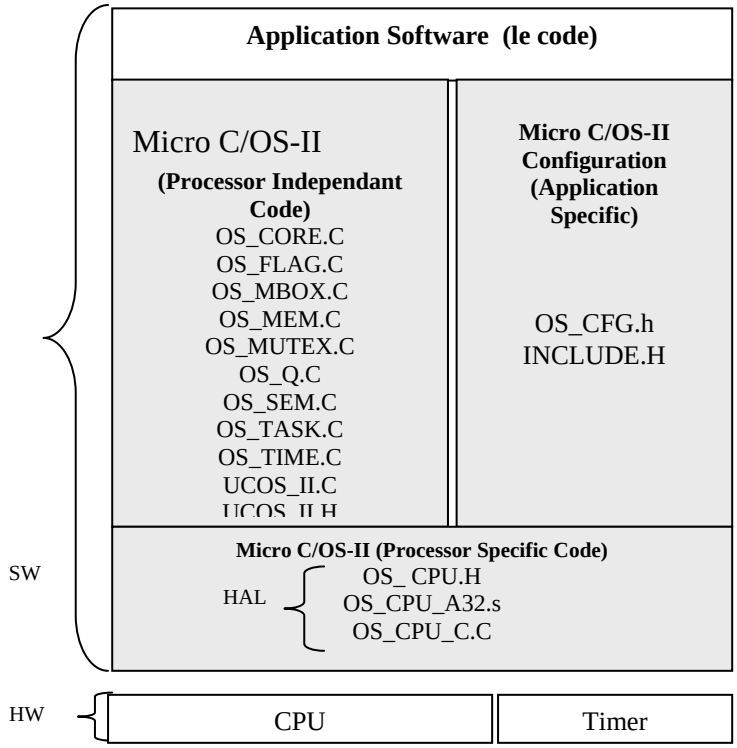

Figure 11: Structure de MicroC/OS-II

2.1.3. Fonctionnement de MicroC/OS-II

Lors de l'initialisation du programme MicroC/OS-II, les différents programmes de l'utilisateur sont considérés comme des tâches qui sont toutes créées pendant cette période d'initialisation. Le programmeur doit alors spécifier le point d'entrée de la tâche, l'emplacement des données pour cette tâche, l'adresse de la tête de la pile de la tâche et le degré de sa priorité. Ainsi, la tâche du plus haut degré de priorité est prête à l'exécution. Les tâches peuvent communiquer avec d'autres grâce aux sémaphores, boîtes aux lettres, files d'attentes et aux drapeaux d'événements, ou bien avec des périphériques grâce aux ISRs.

2.1.3.1. Création d'une tâche

Une telle tâche de l'application est constituée par une zone d'initialisation (une zone permettant d'initialiser les variables du programme utilisateur), une zone où l'utilisateur place le code de son programme et une instruction OSTimeDly(n) permettant de céder « *n* » coups d'horloge aux autres tâches. La création de la tâche se fait en appelant la routine suivante :

*OSTaskCreate(AppTask1, (void *)0, (void *)&AppTask1Stk[255], 3);*

- AppTask1: point d'entrée du programme utilisateur (nom de l'étiquette).
- (void *) 0 : adresse des données.
- (void *)&AppTask1Stk [255] : adresse de la tête de la pile de la tâche.
- 3 : degré de priorité de la tâche.

2.1.3.2. Fonctions de base

Les principales routines de MicroC/OS-II sont [27] :

- Initialisation de µCOSII : OSInit()
- Démarrage du multitâche : OSStart(),
- Gestion des tâches: OSTaskCreate, OSTaskCreateExt, OSTaskQuery, OSTaskDel, OSTaskDelReq, OSTaskChangePrio, OSTaskSuspend et OSTaskResume.
- Gestion d'interruption : OSIntEnter et OSIntExit.
- Gestion du temps: OSTimeDly, OSTimeDlyHMSM, OSTimeDlyResume, OSTimeSet, OSTimeGet et OSTimeTick.
- Gestion des sémaphores : OSSemCreate, OSSemAccept, OSSemPost, OSSemPend, OSSemDel et OSSemQuery.
- Gestion des mails box : OSMboxCreate, OSMboxAccept, OSMboxPost, OSMboxPend, OSMboxDel et OSMboxQuery.
- Gestion des files de communication : OSQCreate, OSQAccept, OSQPost, OSQPend, OSQQuery et OSQDel.
- Gestion des drapeaux d'événements : OSFlagCreate, OSFlagPost, OSFlagPend, OSFlagDel, OSFlagAccept et OSFlagQuery.

2.1.4- Communication inter tâches

Deux mécanismes élémentaires sont adoptés :

2.1.4.1- Partage de variable

Dans le cadre d'un partage de variable, le plus souvent, une tâche produit des données qui sont utilisées par une (ou plusieurs) autre(s) tâche(s). La coopération des tâches de l'application entre elles s'effectue à travers les messages, et les queues de messages. Alors que le sémaphore est employé pour gérer l'accès exclusif à la ressource partagée du système (mémoire vidéo). Le commun entre toute coopération est la présence de deux actions :

- Signalisation ; appelée aussi envoi (*Posting*).
- Attente ; appelée aussi réception (*Pending*).

Avec MicroC/OS, lors de la création d'un tel outil de communication, un ECB (*Event Control Block*) est créé pour maintenir l'état courant de cet outil. En fait, un ECB est une structure de données désignée pour décrire le type de l'événement en cours, ainsi que la liste des tâches en attente sur cet événement, avec d'autres informations nécessaires pour sa gestion.

Les actions de synchronisation mises au point sont les suivantes :

- Lors d'un PEND sur un sémaphore, un mutex, un message de la queue de messages ou un message d'un mail box, la fonction OS_EventTaskWait() est appelée pour retirer la tâche courante de la liste OSRdyGrp, et la mettre à l'état bloqué dans la liste OSEventGrp.
- Lors d'un POST sur l'un de ces outils, la fonction OS_EventTaskRdy() est appelée pour déterminer la prochaine tâche en attente qui aura la section critique. Et donc, celle-ci sera retirée de la liste OSEventGrp et mise à nouveau, active dans la liste OSRdyGrp.
- Lors d'un retour au PEND sur timeout, la fonction OS_EventTo() va retirer la tâche de la liste OSEventGrp, mais sans la mettre à nouveau active car elle l'est déjà active. En effet, c'est OS_TickTime() qui a la responsabilité de mettre OSTCBDly à jour et puis de rendre la tâche active lorsque ce dernier arrive à 0.

2.1.4.2- Synchronisation par événements

Dans ce cadre, les tâches sont synchronisées via les événements. En fait, si deux tâches ont besoin de se synchroniser avec l'apparition de multiples événements, typiquement, la seconde, afin de poursuivre son exécution, devra attendre que la première parvienne à un point donné. La synchronisation est maintenue à travers les drapeaux d'événements (Event Flag).

Les drapeaux d'évènements de µCOS-II sont constitués de deux éléments : une série de bits (8 ou 16 ou 32 bits) utilisés pour maintenir l'état courant des événements dans le groupe, et une liste de toutes les tâches en attente de la combinaison de ces bits (0 et 1) selon l'ordre désiré.

La gestion d'un événement se fait généralement au moyen des actions suivantes:

- Lors d'un PEND, la fonction OS_FlagBlock() est appelée pour maintenir le blocage de la tâche en attente sur l'apparition de l'événement. En fait, si les bits désirés dans le groupe de drapeaux d'événements (*Event Flag Group*) ne sont pas encore obtenus, cette tâche restera en attente indéfiniment jusqu'à la production de l'événement, ou bien l'expiration du timeout. Dans le cas de notre application, nous attribuons la valeur 0 au champ « *timeout* », étant donné que les tâches en attente sur un événement ne consomment aucune capacité de traitement, donc elles restent indéfiniment en attente jusqu'à ce que l'événement se produise.
- Lors d'un POST, la fonction OS_FlagTaskRdy() est appelée pour retirer la tâche bloquée de la liste d'attente (*Waiting List of the Event Flag Group*), et la remettre à nouveau à l'état prêt pour s'exécuter. Pour garantir qu'à tout moment le système puisse répondre aussi rapide que possible à un événement, cette tâche devrait commencer son exécution juste après

la terminaison de la tâche produite. Pour ce faire, si la priorité associée à la tâche produite est « *i* », alors la priorité de la tâche consommatrice sera « *i+1* » sachant que la valeur la plus petite correspond à la priorité la plus élevée.

2.2- Carte de développement : STRATIX-II

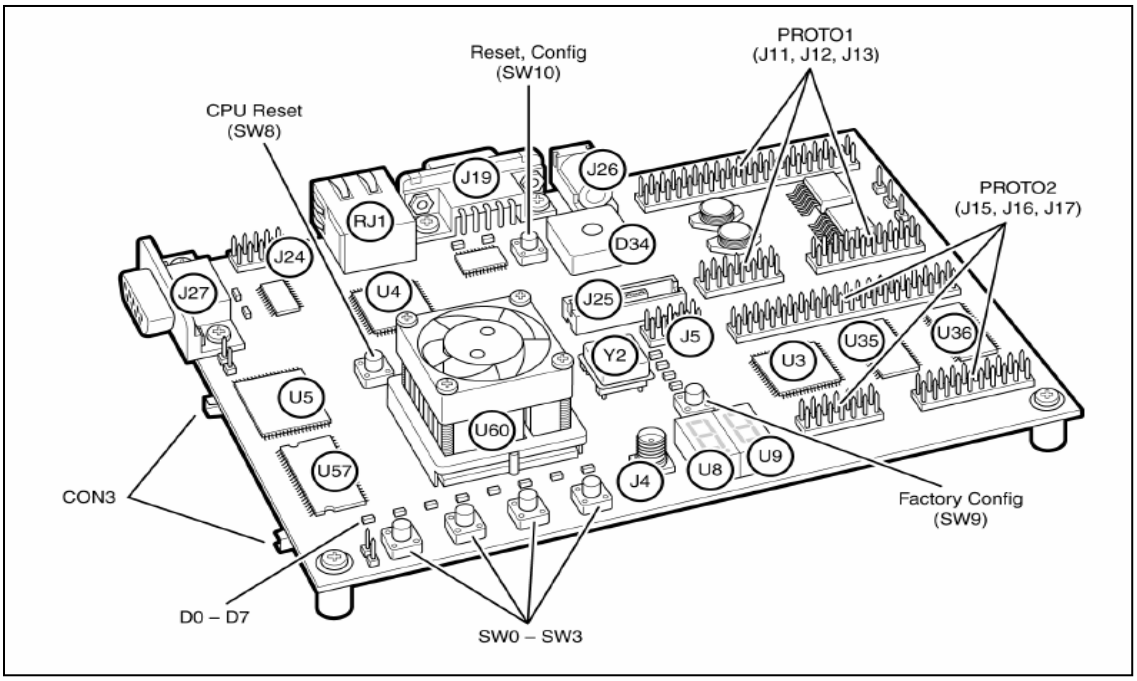

Figure 12: Carte de développement STRATIX II

2.2.1. Description

Cette carte de développement STRATIX-II d'Altera comprend les éléments suivants (figure 12) :

- Un circuit STRATIX II EP2S60 Device (U60),
- Deux SRAM d'1Mbits (512Ko * 16) (U35-U36),
- Une mémoire flash de 16 Mb (U5),
- Deux connecteurs de port série (J19, J27),
- Deux expansions de connecteurs de prototype (PROTO1, PROTO2),
- Connecteur de mémoire flash (CON3),
- Connecteur Mictor (J25),
- Contrôleur de configuration de la carte (U3),
- Deux connecteurs JTAG pour les fonctions de configuration (J24, J5),
- Quatre boutons poussoirs définissables par l'utilisateur (SW0-SW3),
- mémoire SDRAM (U57),
- interface Ethernet MAC/PHY (U4),

- Deux boutons affectés en priorité :
  - Le « *Reset* » : recharge la configuration de la carte selon le contrôleur de configuration,
  - Le « *Clear* » : effectue le Reset du CPU.
- Huit diodes electro-luminéscentes (D0-D7),
- Deux afficheurs 7 segments (U8 et U9),
- Un circuit de contrôle de l'alimentation,
- Un oscillateur générateur d'une horloge.

2.3- Environnement de développement

2.3.1. Environnement Quartus

Cet environnement permet la création, la compilation, la simulation et le prototypage sur la carte Excalibur d'applications pour les circuits Altera.

2.3.2. SOPC Builder

L'environnement Quartus permet la création des systèmes complexes comportant des processeurs, des périphériques, des mémoires, des bus, des arbitres, et des noyaux d'IPs. Alors que le SOPC Builder produit automatiquement la logique nécessaire pour intégrer tous ces composants sur la même carte.

Ce système inclut automatiquement un bus pour l'interconnexion logique entre ce bus Avalon et les ports de tous les périphériques du système NIOS-II. La bibliothèque du SOPC Builder contient des composants sous forme de blocs :

- Simple de logique fixée,
- Complexe paramétrable,
- Sous systèmes dynamiquement générés.

Le SOPC Builder permet de générer le système par la génération de fichiers pour la synthèse et la simulation. Il est composé de :

- Une interface graphique pour spécifier et placer les composants constituant notre système. Chaque composant peut être configuré selon les besoins à travers une petite interface graphique spécifique pour lui. Une fois les composants sont bien listés et arrangés, une description du système dans un fichier (*.PTF) se crée.

- Un programme générateur pour convertir la description du système (*.PTF) vers une implémentation matérielle. Il permet ainsi de créer une description HDL du système pour une cible sélectionnée.

3- Méthodologie de développement logiciel et matériel

Dans le cadre du prototypage d'un modèle de système réactif embarqué sur une plate-forme à base d'un cœur de processeur RISC et d'un RTOS, nous adoptons une méthodologie de conception et de développement logiciel/matériel.

Avec cette méthodologie, nous devons bien évidemment, d'un coté, concevoir tout le système matériel devant être intégré au FPGA, et parallèlement, développer le logiciel et prévoir son intégration.

La figure 13 présente les étapes nécessaires pour le prototypage d'un SoC en utilisant le kit de développement Excalibur d'Altera.

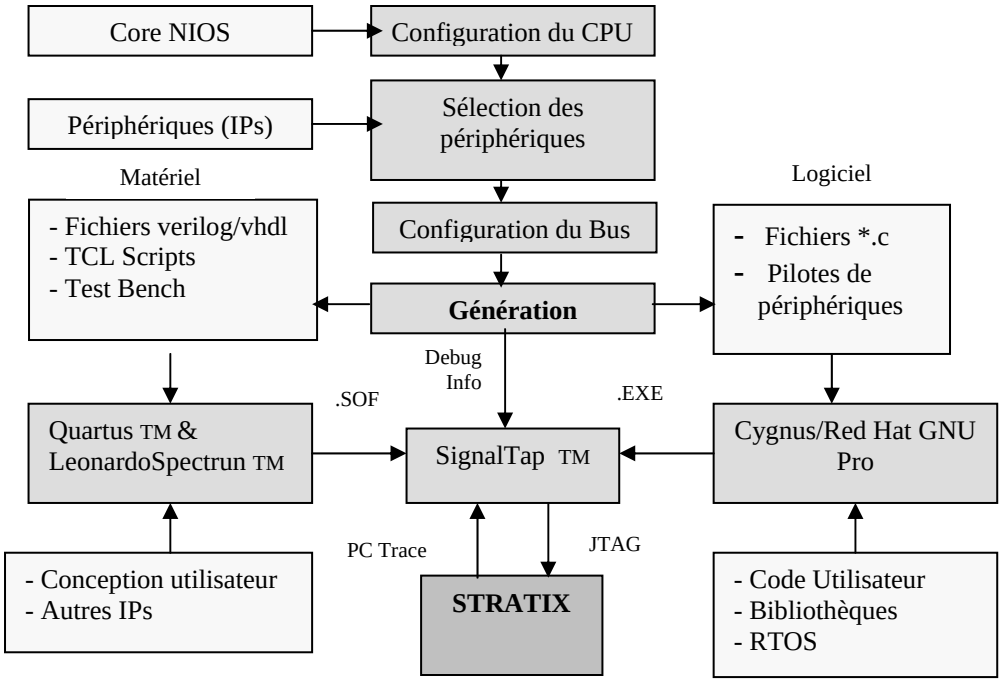

Figure 13: Flot de conception logiciel et matériel

4- Conception d'un système réactif embarqué monoprocesseur

4.1- Réalisation de la plate-forme matérielle à base de NIOS II

La conception des systèmes sur puce par l'environnement d'Altera est rendue très simple grâce à l'entrepreneur SOPC Builder qui permet de concevoir le système voulu par

l'assemblage d'un ensemble d'IP fourni dans une bibliothèque avec cet environnement. Le système monoprocesseur qu'on a proposé contient les composants suivants:

- Un processeur NIOS II 32 bits,
- Deux UART « *interface série* » l'une pour le Jtag utilisé lors de la configuration et l'autre pour l'affichage sur l'écran de l'ordinateur,
- Une interface avec la SRAM externe,
- Une mémoire flash,
- Une mémoire interne,
- Une ROM de boot,
- Un timer utilisé pour donner le contrôle au code du RTOS adopté,
- Un autre timer employé pour compter le nombre de cycles d'horloges réservés pour chaque tâche de l'application.

Il est à noter que le timer peut être configuré suivant les besoins de l'utilisateur. Notre système fonctionne avec une fréquence de 50 Mhz, la durée d'un cycle d'horloge est donc égale à T=1/f.

Une fois le choix des différents composants du système est fait, il est nécessaire de faire l'interconnexion entre eux afin de réaliser la fonction globale du système ; ce qui se fait d'une manière graphique dans le SOPC Builder. A chaque fois qu'on veut faire interconnecter deux composants, il faut cocher le point initialement blanc qui les relie. Ce point devenu noir, indique qu'ils sont connectés. Après la phase d'interconnexion, il reste l'assignement de la base d'adresse de tous les composants constituant le système. Cette opération peut se faire de deux manières : soit automatiquement ou manuellement (dans le cas ou le concepteur veut fixer des adresses bien précises). Dans le deuxième cas, il faut prêter l'attention à ne pas faire de conflits d'adresses qui peuvent causer un disfonctionnement du système.

La figure suivante illustre la phase de conception de notre système par le SOPC Builder :

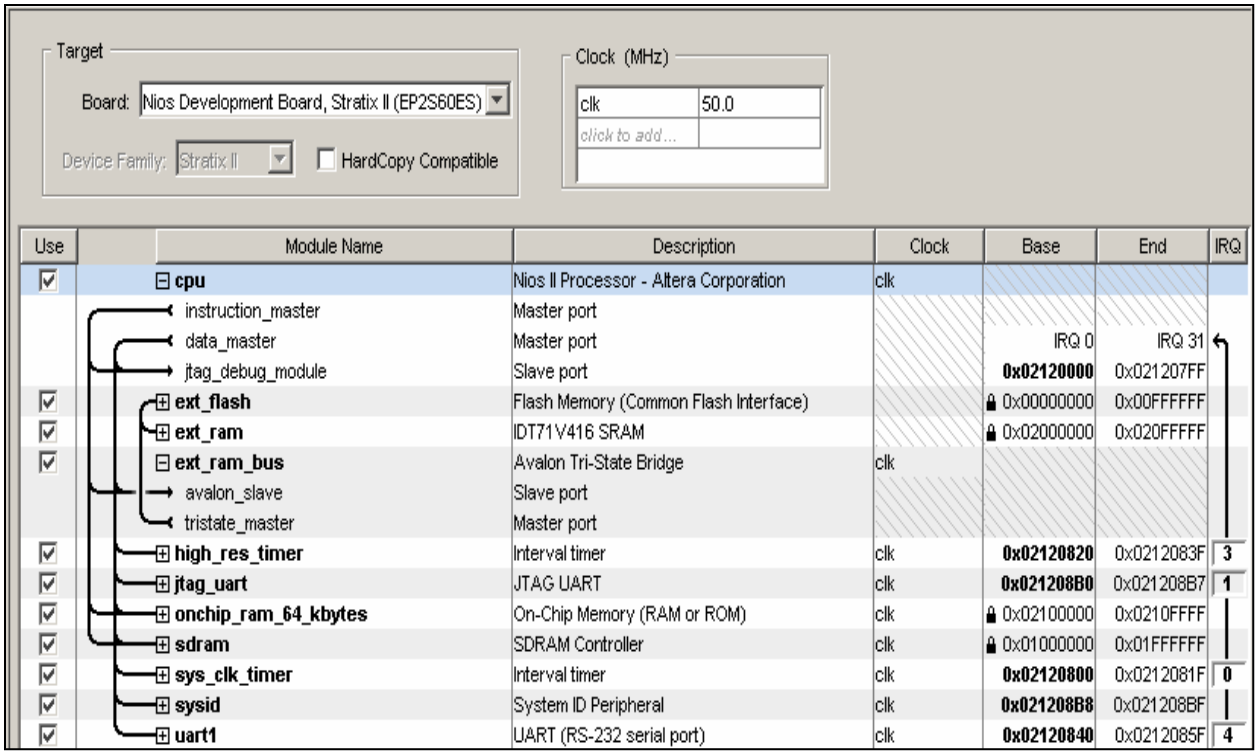

Figure 14: Architecture monoprocesseur

On termine la phase de conception de la partie matérielle par le SOPC Builder par une étape de génération du système qui permet de générer le système global sous forme de boîte noire (on ne peut voir que les entrées et les sorties du système) dans le logiciel QUARTUS.

Si l'étape de génération est terminée avec succès, il restera à interconnecter les différents composants externes (mémoire externe, accélérateur,…) au système déjà conçu. Une fois cette tâche accomplie, on procède à la compilation du système global pour vérifier s'il comporte des erreurs de conception ou de dépassement de la capacité de notre carte de prototypage. Si la compilation est terminée avec succès, on pourrait conclure que notre partie matérielle est prête, et qu'il ne reste plus d'a que le développer la partie logicielle.

Pour le développement et l'exécution de la partie logicielle, la nouvelle version de QUARTUS II comporte un nouveau outil appelé NIOS-II IDE « *Nios II Integrated Development Environment*» qui peut supporter le langage C/C++ et permet la compilation des projets réalisés suivant les spécifications de la plateforme d'Altera. Cet outil permet aussi la configuration de l'FPGA avec le système déjà conçu et l'exécution du résultat de la compilation, directement sur la carte de prototypage.

4.2- Portage du MicroC/OS-II sur le processeur NIOS

Avec les versions antérieures de notre environnement de conception, il a été nécessaire de configurer le port du MicroC/OS-II suivant les spécificités du processeur NIOS, pour

pouvoir exécuter des applications écrites avec l'utilisation des routines de notre RTOS. Alors qu'avec la version actuelle, le code et le port du MicroC/OS-II étaient fournis avec l'environnement. Ainsi, pour bénéficier des routines de ce RTOS et pouvoir exécuter les applications temps réel sur le processeur NIOS II, il suffit de choisir, lors de la création du projet de travail, celui qui utilise le MicroC/OS-II.

4.3- Configuration des services de MicroC/OS-II

Le nouvel outil NIOS II IDE fournit une interface pour la configuration des services du MicroC/OS-II. A travers cette étape, on peut : fixer le nombre maximum de tâches que l'on peut utiliser dans notre application, sélectionner les services que l'on va utiliser tel que les sémaphores, les drapeaux d'événements, et fixer la valeur la moins prioritaire.

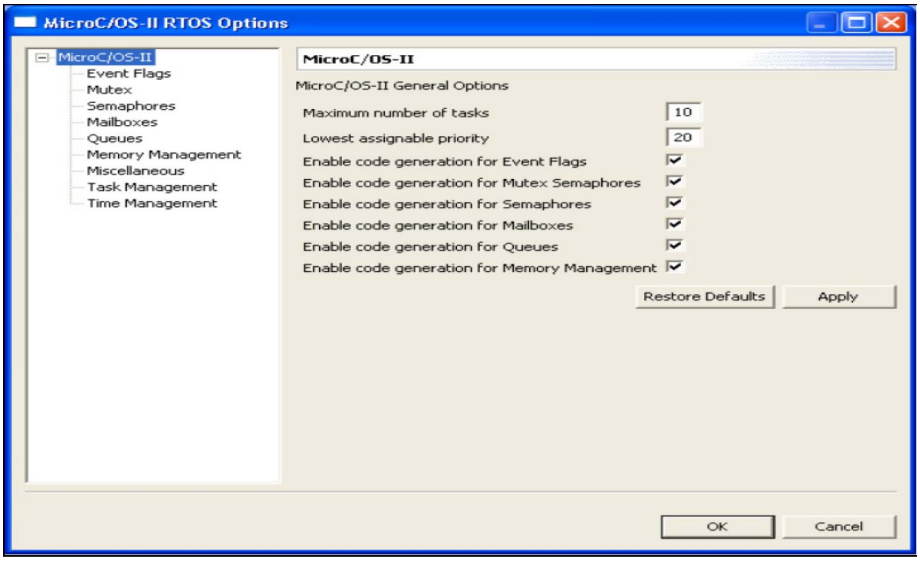

Figure 15: Configuration de l'RTOS

5- Conception d'une architecture multiprocesseurs

5.1- Bus Avalon

C'est le bus utilisé par le processeur NIOS-II. Il peut être vu comme un ensemble de signaux prédéfinis, permettant de connecter un ou plusieurs blocks IP. En plus, il est généré automatiquement par le NIOS-II Builder. Le bus Avalon a comme caractéristiques principales Figure 16 :

- Plusieurs circuits maîtres simultanés [28]. Et en cas d'une ressource partagée, un arbitrage nécessaire pour le partage de cette ressource par les circuits maîtres est automatiquement inclus.
- Dimensionnement dynamique des interfaces [28]. Ceci permet d'utiliser de la mémoire avec une taille de données inférieure à celle du bus NIOS-II. Par exemple, un

système configuré avec un bus de données de 32 bits, pourra intégrer facilement une mémoire flash 8-bits. Le NIOS-II Builder aura automatiquement généré la logique nécessaire à cette opération.

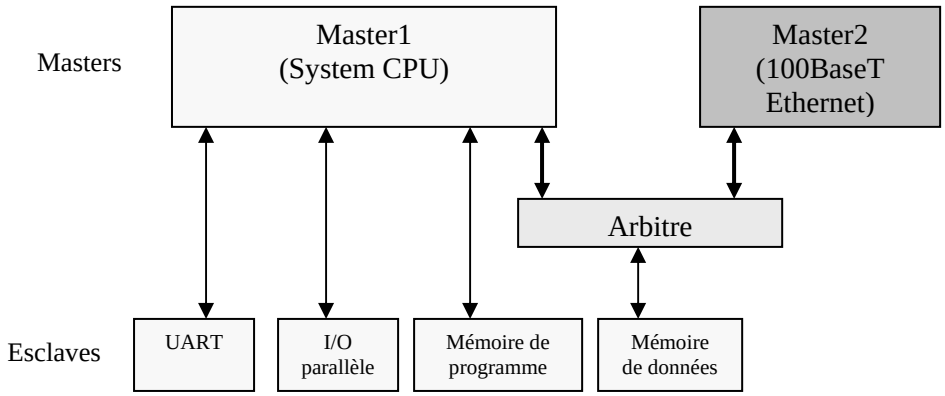

Figure 16: Système multi masters

5.2- Mutex fourni par l'interface Avalon

Le mutex est un composant Hardware fourni comme un IP avec l'environnement d'ALTERA. Il est généralement utilisé par les environnements multiprocesseurs afin de coordonner les accès à une ressource partagée. Le mutex fournit un protocole pour assurer la propriété mutuelle exclusive d'une ressource partagée.

Le mutex fournit une opération « *test-and-set* » à base de matériel, permettant au logiciel dans un environnement multiprocesseurs de déterminer le processeur qui possède l'accès à une ressource partagée. Le mutex est utilisé dans la conjonction avec la mémoire partagée pour mettre en oeuvre des dispositifs de coordination d'inter processeurs complémentaires, comme les boîtes aux lettres et le logiciel mutex. Il est conçu pour l'utilisation dans des systèmes de processeurs, mais il faut configurer cet IP suivant la plateforme utilisée. Pour notre plateforme de travail, Altera fournit un driver pour le processeur de Nios-II afin de permettre l'utilisation du matériel mutex.

5.2.1- Comportement de base

Le mutex a une interface d'esclave d'Avalon simple qui fournit l'accès à deux registres 32 bits. Le tableau 4 expose ces registres.

Offset	Nom du Registre	L/E	Description du Bit		
			31..16	15..1	0
0	mutex	L/E	Propriétaire	VALEUR	
1	reset	L/E	-	-	RESET

Tableau 4: Registre de mutex

Le mutex a le comportement de base suivant. Cette description suppose qu'il y a des processeurs multiples ayant accès à un mutex simple et que chaque processeur a un identificateur unique (ID).

- Quand le champ de VALEUR est 0x0000, le mutex est disponible (i.e, ouvert). Autrement, le mutex est indisponible (c'est-à-dire, fermé).
- Le registre de mutex est toujours lisible. Un processeur (ou n'importe quel périphérique de maître Avalon) peut lire le registre de mutex pour déterminer son état actuel.
- Le registre de mutex est échangeable seulement dans des conditions spécifiques. Une opération d'écriture qui vise à changer le contenu du registre du mutex ne se fait que si une ou deux des conditions suivantes soient vraie :
  - ♦ Le champ de VALEUR du registre de mutex est le zéro.
  - ♦ Le champ de PROPRIÉTAIRE du registre de mutex correspond au champ de PROPRIÉTAIRE dans les données à être écrit.
- Un processeur essaie d'acquérir le mutex en écrivant son ID au champ de PROPRIÉTAIRE et écrit une valeur différente de zéro dans le champ VALEUR. Le processeur vérifie alors si l'acquisition est succédée en examinant le champ de PROPRIÉTAIRE.
- Après la remise à zéro du système, le bit RESET dans le registre « *Reset* » est au niveau haut. L'écriture de 1 dans ce bit l'efface.

5.2.2- Configuration du mutex dans SOPC Builder

Les concepteurs du matériel utilisent la configuration du Constructeur SOPC Builder pour spécifier les fonctions du matériel. La fenêtre de configuration du mutex fournit les fixations des paramètres suivants :

- **Initial Value**--Le contenu initial du champ de VALEUR après « Reset ». Si le champ **Initial Value** est différent de zéro, vous devez aussi spécifier le **Propriétaire** Initial «*Initial Owner* ».

- **Initial Owner**--Le contenu initial du champ de PROPRIÉTAIRE après « Reset ». Quand le Propriétaire Initial est spécifié, ce propriétaire doit sortir le mutex avant qu'il ne soit acquis par un autre propriétaire**.**

5.2.3- Modèle de programmation logicielle

Les sections suivantes décrivent le logiciel programmant le modèle pour le mutex, comme le logiciel construit est utilisé pour avoir accès au matériel. Pour des utilisateurs de processeur de Nios-II, Altera fournit des routines pour avoir accès au matériel fondamental mutex. Ces fonctions sont spécifiques au mutex et manipulent directement le matériel à bas niveau. Le mutex ne peut pas être en accès via l'API de HAL ou l'ANSI C la bibliothèque standard. Dans des systèmes de processeurs de Nios-II, un processeur ferme le mutex en écrivant la valeur de son registre de contrôle de CPU_ID au champ de PROPRIÉTAIRE du registre de mutex.

Le fichier altera_avalon_mutex.h déclare une structure alt_mutex_dev qui représente un cas d'un mutex. Il déclare aussi des fonctions pour avoir accès à la structure de matériel mutex, inscrite dans le Tableau suivant.

Nom de la fonction	Description
Altera_avalon_mutex_open()	Revendique à une poignée à un mutex, permettant à toutes les fonctions d'avoir accès au coeur mutex.
Altera_avalon_mutex_trylock()	Essaie de fermer le mutex. Envoie des retours immédiats s'il échoue à fermer le mutex.
Altera_avalon_mutex_lock()	Ferme le mutex. Ne retourne pas jusqu'à ce qu'il aie avec succès fermé le mutex.
Altera_avalon_mutex_unlock()	Ouvre le mutex.
Altera_avalon_mutex_is_mine()	Détermine si ce CPU possède le mutex.
Altera_avalon_mutex_first_lock()	Teste si le mutex a été sorti depuis un « Reset »

Tableau 5: Fonction du mutex

5.3- Boîte aux lettres « Mailbox» fournie par altéra

La boîte aux lettres doit contenir deux mutexes : L'un pour s'assurer qu'un seul processeur possède l'accès à la mémoire partagée à la fois et l'autre, afin de garantir qu'il y a un unique accès en lecture de la mémoire partagée. La boîte aux lettres est utilisée dans la conjonction avec une mémoire séparée dans le système qui est partagé parmi des processeurs multiples.

Le coeur de la boîte aux lettres soutient toutes les familles de dispositifs Altera soutenues par le Constructeur SOPC et fournit pour la couche d'abstraction du matériel (HAL) de Nios-II la bibliothèque système.

5.3.1- Comment utiliser le cœur de la boîte aux lettres dans SOPC Builder

1- Décider quels processeurs doivent partager la boîte aux lettres.

2- Dans l'étiquette du SOPC Builder **System Contents**, instantier un composant de mémoire pour servir de buffet de boîte aux lettres. N'importe quelle RAM peut être utilisée comme buffet de boîte aux lettres qui peut partager l'espace dans une mémoire existante, comme la mémoire de programme; il n'exige pas de mémoire consacrée.

3- Dans le SOPC Builder System Contents étiquette, instantier le composant de boîte aux lettres. La fenêtre de configuration de boîte aux lettres ne présente aucune fixation configurable.

4- Faire les connections nécessaires dans l'étiquette du SOPC Builder **System Contents.**

a- connecter chaque port du bus de donnée maître du processeur au port esclave de la boîte aux lettres.

b- connecter chaque port du bus de donnée maître du processeur à la mémoire partagée de la boîte aux lettres.

5- Configurer le cœur de la boîte aux lettres dans l'étiquette **More**<nom mailbox>**Settings**. Cette étiquette se trouve dans l'interface graphique utilisateur du SOPC Builder chaque fois qu'un mailbox existe dans le système.

L'étiquette **More**<nom mailbox>**Settings** fournit les options suivantes :

- **Memory module** spécifie quelle mémoire on va utiliser pour le mailbox. Si la liste apparue ne contient pas la mémoire désirée c'est que cette dernière n'est pas connectée au système correctement.
- **Shared Mailbox Memory Offset** spécifie un offset dans la mémoire à partir duquel commence le mailbox

- **Mailbox Size (bytes)** spécifie le nombre d'octets à utiliser pour le mailbox message buffet. Le logiciel driver de Nios-II fourni par Altera utilise huit octets d'au-dessus pour mettre en oeuvre la fonctionnalité de la boîte aux lettres. Pour une boîte aux lettres capable de passer seulement un message à la fois, la Taille de Boîte aux lettres doit être au moins 12 octets puisque la taille d'un message qui peut être stocké dans une boîte aux lettres doit être 4 octets.

5.3.2- Caractéristiques du mailbox

Le logiciel de boîte aux lettres programmant le modèle a les caractéristiques suivantes et suppose qu'il y a des processeurs multiples ayant accès à un coeur de boîte aux lettres simple et une mémoire partagée.

- Chaque message de boîte aux lettres est un mot 32 bits.
- Il y a une gamme d'adresses prédéterminées dans la mémoire partagée consacrée au stockage de messages. La taille de cette gamme d'adresses impose une limite maximale au nombre de messages stockés.
- Le logiciel de boîte aux lettres met en oeuvre un message *FIFO*[22] (premier entré premier sorti) entre des processeurs. Un seul processeur peut écrire à la boîte aux lettres par fois et un seul processeur peut lire de la boîte aux lettres par fois, assurant l'intégrité de message.
- Le logiciel tend sur l'envoi que les processeurs de réception doivent convenir d'un protocole pour interpréter des messages de boîte aux lettres. Typiquement les processeurs traitent le message comme un pointeur sur une structure dans la mémoire partagée.
- Le processeur d'envoi peut poser des messages dans la succession, jusqu'à la limite imposée par la taille de la gamme d'adresse de messages.
- Quand les messages existent dans la boîte aux lettres, le processeur de réception peut les lire . Le processeur de réception peut être bloqué jusqu'à ce qu'un message apparaisse, ou il peut voter la boîte aux lettres pour de nouveaux messages.
    - La lecture du message enlève le message de la boîte aux lettres.

5.3.3- Programmation du cœur de la boite aux lettres

Cette section décrit le logiciel construit pour manipuler le matériel de la boîte aux lettres.

[22] *First In First Out*

Le fichier altera_avalon_mailbox.h déclare une structure alt_mailbox_dev qui représente un cas d'un dispositif de la boîte aux lettres. Il déclare aussi des fonctions pour avoir accès à la structure de matériel de la boîte aux lettres, inscrite dans le tableau suivant.

Nom de la fonction	Description
altera_avalon_mailbox_close()	Ferme la poignée à une boîte aux lettres.
altera_avalon_mailbox_get()	Rend un message s'il en a présent, mais ne se bloque pas en attente d'un message.
altera_avalon_mailbox_open()	Revendique à une poignée à une boîte aux lettres, permettant à toutes les autres fonctions d'avoir accès à la boîte aux lettres.
altera_avalon_mailbox_pend()	Se bloqué en attendant un message pour être dans la boîte aux lettres
altera_avalon_mailbox_post()	Poste un message à la boîte aux lettres.

Tableau 6: Fonction du mailbox

5.4-Topologie proposée pour une architecture multiprocesseurs :

Suite à l'étude faite sur les différentes architectures existantes pour un système multiprocesseurs dans le chapitre1, le bus avalon d'Aletra qui est un bus simultané multi maître et les modules fournis par Aletra « *mutex et mailbox* » pour la communication entre les processeurs, on a proposé la topologie suivante figure 17 pour le prototypage des systèmes réactifs multiprocesseurs sur des architectures reconfigurables.

Cette architecture est composée essentiellement de :

-un ensemble de sous systèmes qui contient :

- processeur,
- accélérateur,
- coprocesseur,
- IP,
- Mémoire,
- …

- une mémoire partagée par l'ensemble des processeurs pour assurer la communication entre eux. L'accès à cette mémoire est contrôlé par un mutex ou une boite aux lettres. Au début on

a utilisé un seul module de communication entre processeurs (mutex ou mailbox)Mais on s'est rendu compte que cette méthode présente quelques problèmes On va donc ralentir un peu le système puisqu'à chaque fois que le processeur va prendre un message de la mémoire, il doit consulter tous les messages pour trouver celui qui lui est destiné. En outre, dans le cas où on utilise un mailbox, et puisque ce module est de type FIFO, le processeur doit remettre les messages qui ne lui sont pas destinés dans le mailbox, vu que la consultation du message l'enlève automatiquement du mailbox. En conséquence, le temps de communication va hausser surtout en augmentant le nombre de processeurs et de messages échangés entre eux.

Pour remédier à ce problème, on a proposé comme solution, l'utilisation d'un module de communication «*mutex ou mailbox*» pour chaque processeur. Ce dernier peut écrire dans n'importe quel module associé à d'autres processeurs et ne peut prendre les messages que de son propre module de communication. De cette façon, on est certain que chaque processeur ne consulte que les messages qui lui sont destinés.

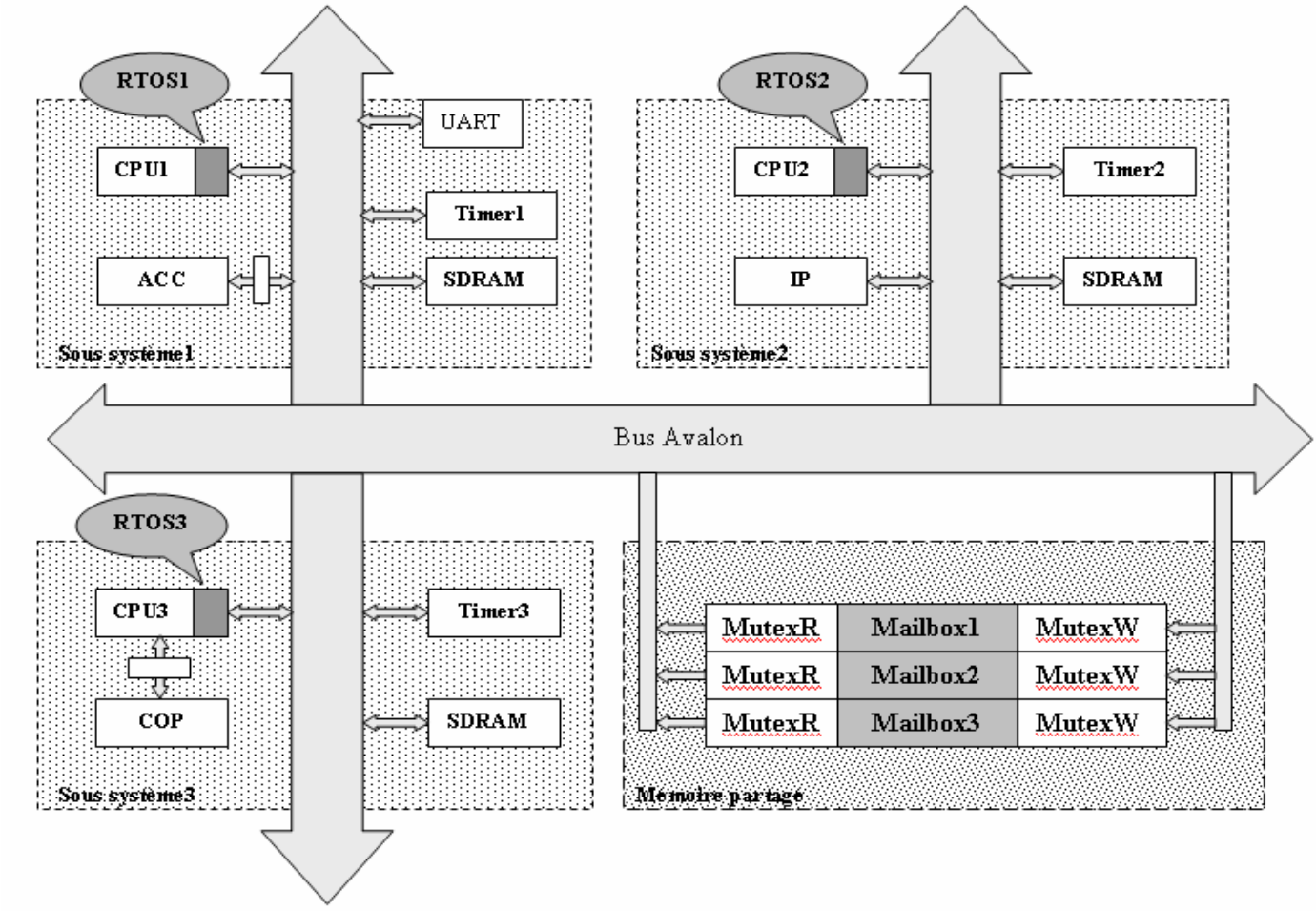

Figure 17: Architecture multiprocesseurs proposée

6- Estimation de performance des systèmes sur puce temps réel

Il est courant d'utiliser un système d'exploitation temps réel dans un tel système embarqué comme étant une structure logicielle permettant de gérer l'exécution complète de plusieurs tâches concurrentes sur le même système. Cependant, et bien que cette méthode soit couramment employée dans les systèmes embarqués spécifiques, celle-ci peut entraîner certains inconvénients en terme de coût, consommation et performances [16].

Dans le cadre de notre projet, on s'intéresse à l'estimation du temps d'exécution d'une application temps réel écrite avec les services d'un RTOS. Une étude des différents outils d'estimation du temps d'exécution a été faite au premier chapitre. Après cette étude on s'est rendu compte que ces outils ne tiennent pas compte des systèmes d'exploitation temps réel. Dans la suite de notre projet, on va donc essayer de générer un modèle qui s' intègrerait dans ses outils pour qu'il puisse estimer le temps d'exécution des applications écrites avec les routines d'un RTOS.

6.1- Principe

Notre idée de départ se base sur le fait que le temps d'exécution d'une application temps réel (tâches + services RTOS) est égal au temps d'exécution de la même application non temps réel auquel on ajoute une certaine valeur due a l'effet de l'utilisation de l'RTOS et le temps pris par chaque service utilisé. Notre modèle consiste en premier lieu, à trouver une méthode qui puisse déterminer l'effet de l'utilisation d'un système d'exploitation sur le temps d'exécution de l'application. En second lieu, on construira une base de données qui contiendrait tous les services qu'on peut utiliser d'un RTOS , ainsi que leurs temps d'exécution. A chaque fois qu'on emploie un service de cet RTOS dans notre application, on ajoute le temps approprié au temps déjà calculé.

6.2- Remarque

- Le modèle généré peut être intégré dans n'importe quel outil d'estimation, puisqu'il utilise les temps d'exécution des différents services du RTOS obtenus par exécution directe sur la plateforme de travail.
- Toutes les mesures sont faites autour de l'environnement d'Altera et le système temps réel MicroC/OS-II. Si on change l'environnement de conception ou le RTOS, on procèdera à la démarche présentée ci-dessous.

6.3- Evaluation de l'effet du MicroC/OS-II sur le temps d'exécution d'une fonction

Afin de pouvoir générer une méthode qui puisse déterminer l'effet de l'utilisation d'un RTOS sur le temps d'exécution d'une application, on a procédé aux étapes suivantes :

1- On a écrit une fonction qui fait un traitement quelconque. Cette fonction a été exécutée sur une plateforme monoprocesseur et on a pris son temps d'exécution sans utiliser un ROTS.

2- On a pris le même code de la fonction et on a mesuré son temps d'exécution sur la même plateforme, mais dans une application temps réel. Cette dernière ce compose uniquement de la tâche qui contient le code de la fonction sans utiliser des routines offertes par notre RTOS (dans cette étape, on ne mesure pas le temps de création de la tâche et d'activation des services de l'RTOS mais plutôt le temps d'exécution de la portion du code qui exécute la même fonction déjà mesurée à l'étape 1).

3- On a refait les étapes 1 et 2 pour des fonctions qui prennent des temps d'exécution différents. Le tableau 7 illustre les résultats trouvés lors de l'exécution des différentes fonctions sur notre plateforme qui se compose essentiellement du processeur NIOS II et du système d'exploitation MicroC/OS-II.

Temps sans RTOS	Temps avec RTOS
90294	91106
449559	451645
897117	901089
4477789	4495777
8956136	8990554
44779001	44950710
89556865	89897197
447781676	449484289
895561972	898968551

Tableau 7: Mesure du temps d'exécution avec et sans RTOS

4- A partir des mesures déjà effectuées dans les étapes précédentes, on a construit un graphe figure 18. On constate que l'ensemble des points forme une droite linaire, et, cela est dû au fait que chaque RTOS possède des tâches système qui interviennent d'une façon périodique dans l'exécution de l'application.

Les tâches systèmes permettent de:

- configurer le système.
- afficher des messages de système sans appeler des routines I/O exécutées dans le contexte de la tâche courante.
- exécuter de différentes fonctions spécifiques aux tâches, à une priorité supérieure.
- exécuter des fonctions de réseau.
- obtenir des statistiques dynamiques (ex.: le degré d'utilisation du CPU par l'application, en pourcentage).

Donc, en utilisant ce graphe, on peut déterminer le temps d'exécution de n'importe quelle application temps réel, tout en sachant son temps d'exécution sans RTOS, bien évidemment sans utiliser les services offerts par l'RTOS.

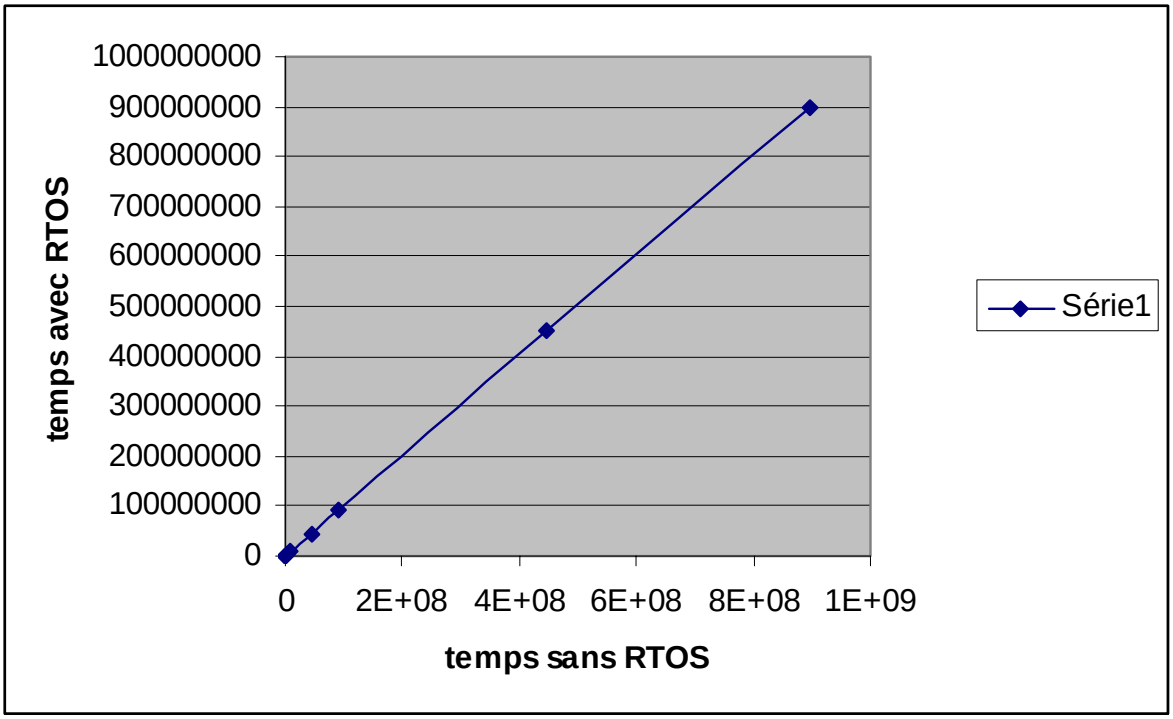

Figure 18: Courbe d'estimation

Pour déterminer le temps avec RTOS, il suffit d'appliquer la formule suivante : Y=1.0038X qui représente l'équation de la droite linaire figure ci-dessus.

X étant le temps de l'application sans RTOS.

Y est le temps de la même application exécutée dans une seule tâche

6.4- Mesure du temps pris par les services du RTOS

Une application écrite en utilisant les routines d'un système temps réel se compose essentiellement d'un ensemble de tâches. Ces tâches utilisent les différents services offerts par l'RTOS, pour gérer la communication et la synchronisation entre elles afin de réaliser la fonction globale de l'application.

On peut diviser les services d'un RTOS en deux groupes :

- Des services qui permettent d'une part la création des différentes tâches, mécanismes de synchronisation et de communication, et l'initialisation de l'RTOS ; et d'autre part, le démarrage de l'application temps réel. Généralement ces services n'entraînent pas de changement de contexte.
- Des services de communication et de synchronisation. Généralement appelés dans le code des tâches à des moments bien déterminés pour réaliser la fonction globale du système. L'appel de ses services peut causer parfois des changements de contexte.

Pour le premier groupe, on constate que le temps pris par n'importe quel service est indépendant du contexte là où il est appelé, puisqu'ils n'entraînent pas de changements de contexte. Par conséquent, ce temps restera le même dans n'importe quel contexte et moment il est utilisé. Alors que les services du deuxième groupe sont plus complexes puisqu'ils exigent un ré-ordonnancement du système et peuvent entraîner des changements de contexte. Donc, il faut mesurer le temps pris par ces services dans les deux cas :

- Appel du services mais pas de changement de contexte : dans ce cas, on mesure le temps pris par l'appel du service et l'exécution de l'instruction qui le suit.
- Appel du service avec un changement de contexte : dans ce cas, on mesure le temps pris par l'appel du service approprié et l'exécution de la première instruction de la nouvelle tâche qui va être exécutée.

Vu que les services du MicroC/OS-II sont très nombreux, on va mesurer le temps pris par ceux que l'on va utiliser dans notre application. (Bien entendu les plus utilisés, lors du développement de n'importe quelle application)

Le tableau suivant présente les mesures de quelques services offerts par MicroC/OS-II :

	Services du MicroC/OS-II	Nombre de tics d'horloge
Groupe1	Création d'une tâche OSTaskCreateExt	9756
	Fonction OS_Start	827
	Création d'un mailbox OSMboxCreate	679
	Création d'un message queue OSQCreate	1565
	Création d'un drapeau d'évènement OSEventFlagCreate	419
Groupe2	OSEventFlag sans changement de contexte	876
	OSEventFlag avec changement de contexte	4464
	OSMboxPost sans changement de contexte	854
	OSMboxPost avec changement de contexte	3418
	OSQPost sans changement de contexte	912
	OSQPost avec changement de contexte	3966

Tableau 8: Temps pris par les services du MicroC/OS-II

6.5- Formalisation du modèle

Etant donné une application qui consomme « *n* » tics d'horloge lors de son exécution sur la plateforme d'Altera sans utiliser les routines du MicroC/OS-II.

Cette application sera décomposée en un ensemble de tâches pour réaliser la fonction globale du système figure 19. A partir du graphe déjà construit, on doit extraire l'une des solutions possibles pour construire le diagramme de séquences qui décrit la succession des différentes tâches ainsi que les routines de l'RTOS, utilisées pour assurer la synchronisation et la communication entre elles (figure 20).

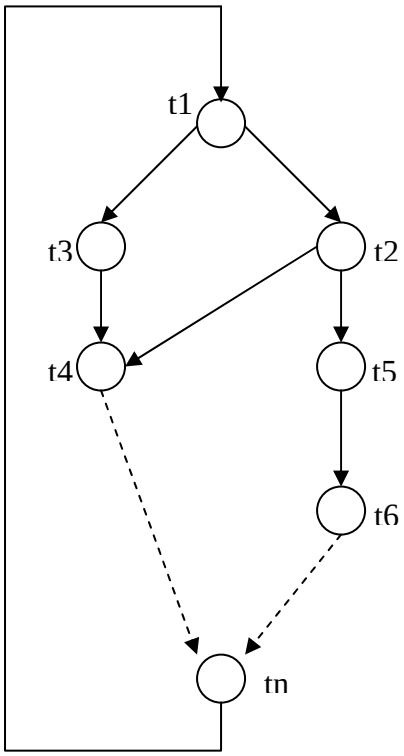

Figure 19: Graphe de tâches

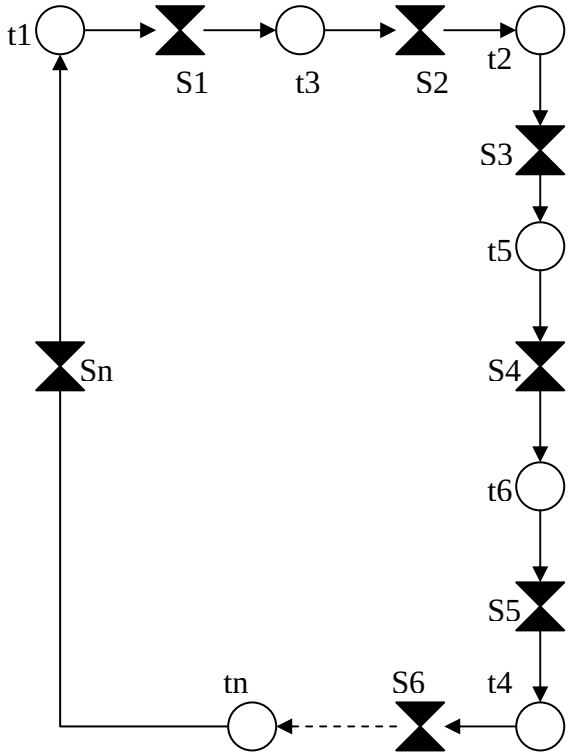

Figure 20: Graphe de séquences

 : représentent les tâches constituants notre application

 : représentent les services utilisés d'un RTOS (sémaphore, mailbox, event_flag …)

Pour calculer le temps global de l'application écrite avec les routines de l'RTOS, il faut :

- Déterminer la nouvelle valeur du temps d'exécution de l'application en utilisant le modèle.
- En utilisant le graphe de l'application, ajouter, à chaque fois qu'on utilise un service de l'RTOS, le temps approprié, à partir du tableau déjà mesuré.

6.6- Mise en équation

Le modèle proposé peut se récapituler dans l'équation suivante :

$$\text{Ntr} = N \times 1.0038 + \sum_{i=n}^{i=0} T_i(S_i)$$

Ntr : nombre de tics de l'application temps réel.

N : nombre de tics de l'application sans RTOS.

$T_i(S_i)$: nombre de tics du service i déterminé à partir du tableau construit.

n : nombre de services utilisés.

6.6- Limites de la méthode d'estimation

- Les applications réactives sont caractérisées par leur interaction avec l'environnement extérieur qui peut influer, sur l'ordre d'exécution des tâches et le comportement du système, dans des temps non prédéterminés. Dans notre modèle, on ne tient pas compte de ce type d'application. On doit connaître, dés le début, l'ordre d'exécution des différentes tâches et les instants de production des interruptions qui peuvent influer sur le fonctionnement de l'application.
- Cette méthode n'est applicable que pour l'application totale. On ne peut pas, par exemple, appliquer le même modèle à une portion de code d'une application temps réel.
- Il faut disposer d'un outil qui fait l'estimation du temps d'exécution sans l'utilisation d'un RTOS.

7. Conclusion

Dans ce chapitre, on a présenté l'environnement d'Altera, notre plateforme de travail, puis on a décrit l'architecture multiprocesseurs proposée et on a terminé par la génération d'un modèle d'estimation de performance utilisé dans le cadre des systèmes sur puce temps réel.

On s'intéressera dans le chapitre suivant, à la validation du modèle d'estimation généré à travers l'application de traitement d'images 3D, et la présentation des différentes étapes de la conception d'un système sur puce multiprocesseurs.

Chapitre 3

Expérimentations et Validations

1- Introduction

Généralement, un système sur puce est formé par un ou plusieurs processeurs, des accélérateurs, des contrôleurs de périphériques, des IP, de la mémoire et une structure de bus ou réseau, etc… Et vu que les applications sont de plus en plus complexes, l'usage des systèmes d'exploitation dans de tel système devient indispensable afin de contrôler le système et de gérer son interactivité avec l'environnement extérieur.

Les objectifs des travaux présentés dans ce chapitre consistent à évaluer l'effet de l'utilisation des systèmes d'exploitation temps réel dans les systèmes sur puce et à expérimenter la conception d'un système multiprocesseurs temps réel.

Ce chapitre est organisé comme suit :

La première section est consacrée à la présentation des différentes étapes du pipeline graphique utilisé dans l'application de traitement d'images 3D et la validation du modèle d'estimation de performance des applications temps réel, à travers cette application. A la deuxième section, nous focaliserons les travaux sur la réalisation de la plateforme multiprocesseurs. On terminera par la validation de l'application de traitement d'images 3D sur l'architecture proposée.

2- Application de traitement d'images 3D

Afin d'expertiser les services du MicroC/OS-II et d'évaluer les performances de notre architecture multiprocesseurs, nous avons considéré une étude de cas sur une application de traitement d'images 3D, ciblée vers une architecture embarquée, basée sur une plate-forme NIOS-II et le système d'exploitation temps réel µC/OS-II.

2.1- Introduction à la création d'objet 3D

L'écran d'ordinateur est seulement capable de représenter des coordonnées en deux dimensions. Comme les écrans de sortie tridimensionnelle n'existent pas encore, on est amené à transformer les coordonnées 3D en coordonnées 2D. Pour ce faire, on utilise la projection par perspective, qui permet de représenter correctement la « *profondeur* » d'un objet en donnant l'impression de volume. Mais précisons dés à présent que toutes les méthodes de création et de visualisation d'un contenu 3D ne permettent de donner à l'utilisateur que l'illusion qu'il évolue dans un « *monde 3D* ».

2.2- Pipeline 3D

Le pipeline 3D est l'ensemble des étapes nécessaires pour la création et la visualisation d'une image 3D. Cette chaîne est décomposée en un ensemble d'opérations nécessaires pour afficher un objet 3D observé à partir d'une position et avec une orientation donnée. Une mise en forme est montrée dans la figure 21 [39].

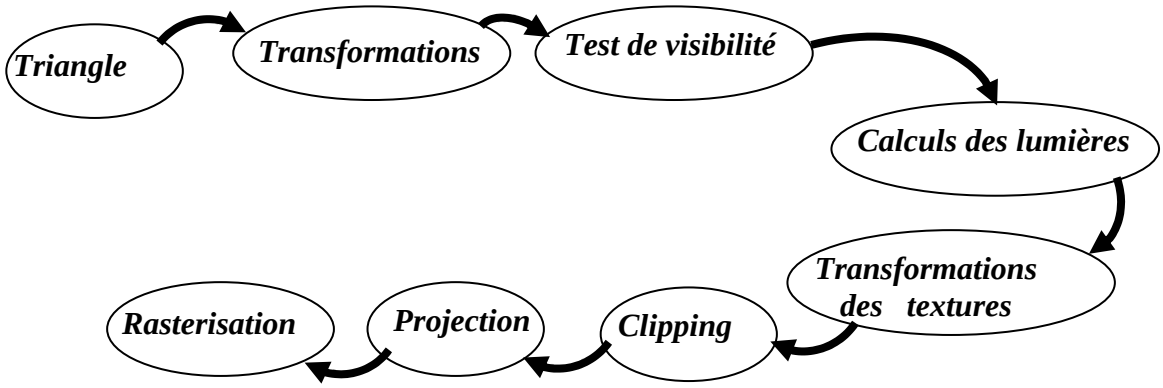

Figure 21: Diagramme de conception de pipeline

2.3- Maillage Triangulaire

Les méthodes de maillage de surface dans l'espace tridimensionnel sont aujourd'hui en plein essor en raison du nombre croissant d'applications dans de nombreux domaines.

Différents types de maillage sont possibles.

2.4- Transformation géométrique

La notation homogène est de grande importance dans les transformations géométriques. En effet, elle permet de concaténer plusieurs transformations Elle représente un outil géométrique très puissant, s'appuyant sur le concept d'ajout d'une troisième coordonnée w. Ainsi, un point 3D devient un vecteur à quatre coordonnées (x, y, z, w).

2.4.1- Translation

La modification est simple dans ce cas. Elle est donnée par :

$$\begin{cases} x' = x + w\,t_x \\ y' = y + w\,t_y \\ z' = z + w\,t_z \\ w' = w \end{cases}$$

En notation vectorielle, la translation est une somme vectorielle donnée par :

$$P' = T_{(t_x,t_y,t_z)}\ P \Rightarrow \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

2.4.2- Changement d'échelles

Les coordonnées sont multipliées par le facteur de changement d'échelles :

$$\begin{cases} x' = S_x\ x \\ y' = S_y\ y \\ z' = S_z\ z \\ w' = w \end{cases}$$

En notation vectorielle, on écrit :

$$P' \ .\ P \Rightarrow \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}.$$

2.4.3- Rotation

La matrice de rotation, dépend de l'axe et de l'angle. A titre d'exemple, on montrera :

- La matrice Rr de la rotation d'angle θ_x par rapport à l'axe Ox.
- La matrice Ry de la rotation d'angle θ_y par rapport à l'axe Oy.
- La matrice Rz de la rotation d'angle θ_z par rapport à l'axe Oz.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2.4.4- Composition de transformation

Pour composer plusieurs transformations, il suffit de multiplier les matrices. Par exemple, la composition d'une rotation par rapport à l'axe x et d'une translation est donnée par :

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = M \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Avec,

M = $T_{(t_x,t_y,t_z)}$. R_x : produit matriciel de la matrice de rotation R_x et de la matrice de translation $T_{(t_x,t_y,t_z)}$.

2.5- Test de visibilité

Le test de visibilité d'un triangle est basé sur l'hypothèse suivante : $V_{1,2,3} \,.\, N \leq 0$

Si l'angle formé entre le vecteur normal $\vec{N}$ et le vecteur de vision $\vec{V}$ est aiguë alors la face sera visible. Sinon elle sera invisible. En d'autres termes, la face est visible si le produit scalaire de $\vec{N}$ et $\vec{V}$ est positif. Ce qui est illustré par la figure 22 [40].

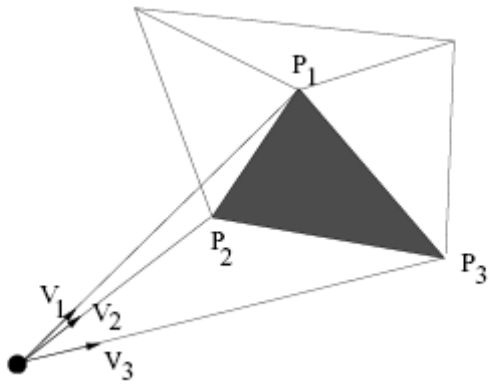

Figure 22: Test de visibilité d'une facette triangulaire

Soit : $V_1 \,(P_1P_2 \wedge P_1P_3) \leq 0$

Et : $V_1 \,((V_2 - V_1) \wedge (V_3 - V_1)) \leq 0$

D'où : $V_1(V_2 \wedge V_3) \leq 0$

Le critère de visibilité consiste donc à déterminer le signe du déterminant correspondant au produit mixte de l'expression précédente :

$$\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{vmatrix} \leq 0$$

2.6- Calculs des lumiéres

Dans cette partie, on va voir des modèles d'illumination constitués principalement de trois composantes : ambiante, diffuse et spéculaire.

On désire calculer particulièrement la quantité de lumière par unité de surface.

2.6.1- Lumière ambiante

Le modèle d'éclairement le plus simple est celui de la lumière ambiante. On considère qu'il existe une source lumineuse uniformément répartie, qui éclaire toutes les directions. Cette lumière représente le niveau minimum d'éclairage qui sera appliqué sur les objets.

On définit l'intensité de cette lumière sur une surface, en particulier une surface triangulaire, par l'équation E(1).

$$\mathbf{I_p} = \delta_a * \mathbf{I_a} \quad \text{E(1)}$$

Cette intensité lumineuse est constante sur toute la surface.

- $\mathbf{I_a}$ désigne l'intensité de la lumière,
- δ_a est le coefficient de réflexion de la lumière ambiante par la surface ($0 \leq \delta_a \leq 1$).
- $\mathbf{I_p}$ correspond à l'intensité de la lumière résultant de la réflexion sur la surface.

2.6.2- Lumière due à une réflexion diffuse

2.6.2.1- principe de la réflexion diffuse

On considère comme hypothèse que la source de lumière est ponctuelle et qu'elle émet de manière constante dans toutes les directions de l'espace.

Dans le modèle de réflexion diffuse, l'intensité en un point d'une surface dépend de l'angle formé entre le rayon de lumière qui touche le point de la surface et la normale à la surface. Plus l'angle formé entre le rayon de lumière et la normale au plan est faible, plus l'intensité lumineuse , réfléchie et visible par l'observateur est forte. Ce principe est illustré par la figure 23

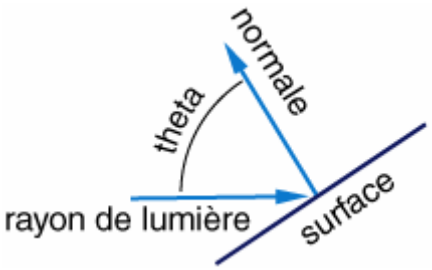

Figure 23: Principe de la réflexion diffuse

2.6.2.2- Calcul des lumières

La lumière émise en direction de l'observateur dépend de l'intensité de la source lumineuse Ib, de l'angle theta formé par le rayon de lumière et la normale au plan et du coefficient de réflexion δ_d de la lumière diffuse par la surface ($0 \leq \delta_d \leq 1$). On obtient la formule E(2) :

$I_p = \delta_d * I_b * \cos(theta)$ E(2)

Remarque : Si *theta* est supérieur à $\frac{\pi}{2}$ alors la face n'est pas du tout éclairée par la source lumineuse. Dans ce cas, l'intensité lumineuse est 0.

2.6.3- Lumière due à une réflexion spéculaire

2.6.3.1- Principe de la réflexion spéculaire

On appelle réflexion spéculaire le phénomène de réflexion de la lumière dans un cône plus ou moins ouvert autour d'une direction privilégiée. A l'extrême, ce cône peut être totalement fermé, la réflexion spéculaire est alors parfaite et ne s'effectue que selon l'axe privilégié (effet miroir parfait).

- visualisation des reflets des sources lumineuses,
- aspect laqué.

La quantité de lumière réfléchie séculairement est en fonction de la distance angulaire entre la direction privilégiée de réflexion et l'axe de vision de l'observateur. Plus cette distance angulaire est grande, moins il y a de lumière spéculaire. Si l'observateur change de position, les taches de lumière spéculaire changent de position. Ce principe est expliqué par la figure24.

2.6.3.2- Calcul de la lumière

Il faut calculer d'abordle rayon réfléchi sur la face. Ensuite, l'intensité de la lumière observée ; dépendra de theta qui correspond à l'angle entre le rayon réfléchi et le point d'observation, de l'intensité Il de la source de lumière et du coefficient de réflexion de la lumière spéculaire par la surface (0 , 1). Le calcul de l'intensité des rayons lumineux résultants de la réflexion spéculaire peut être décrit par le modèle d'illumination donné par l'équation E(3).

Is = * **Il** * cos (**theta**)^**n** (E3)

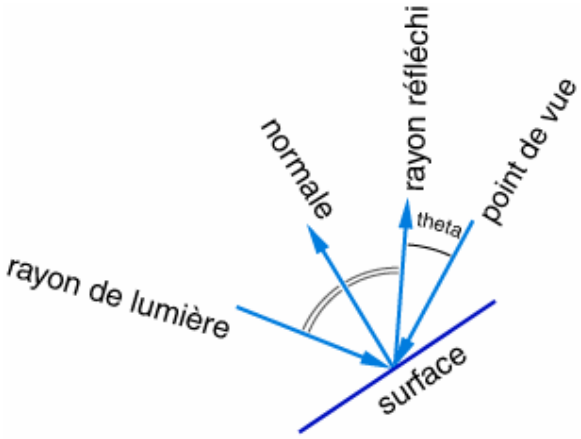

Figure 24: Principe de la réflexion spéculaire

2.7- Transformations des textures

Cette étape permet de transformer les textures avant qu'elles ne soient appliquées au triangle dans l'étape de Rastérisation dans le cas de la chaîne pipeline 3D. Ce sont des transformations 2D sur les images qui sont un cas simplifié des transformations 3D précédentes. Si aucune texture ne doit être appliquée au triangle, cette étape sera sautée.

2.8- Clipping (fenêtrage)

Dans cette étape on élimine les triangles qui ne font pas partie du volume de vue et on découpe ceux en partie visible selon leurs intersections avec le volume de vue. Le Clipping consiste à limiter le tracé d'une figure à une région déterminée. C'est-à-dire rechercher l'intersection entre des figures géométriques simples (formées de triangles élémentaires) et des zones de Clipping graphiques (formées par des rectangles ou des, polygones convexes). Il s'agit donc de déterminer si le triangle considéré est derrière l'observateur, trop loin sur l' un de ses côtés, au-dessus ou en dessous de l'écran. Si le triangle se positionne dans l'un de ces cas, il ne sera pas pris en compte pour le reste du pipeline. Si une partie du triangle n'est pas

visible, le triangle sera "clippé" (découpé). Ceci est remarqué à la figure 25 et sa partie visible suit le traitement du pipeline

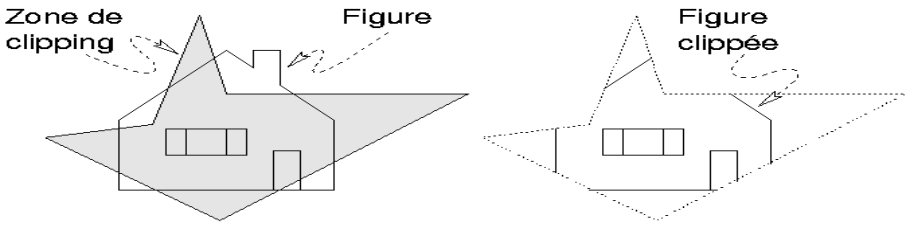

Figure 25: Clipping d'une figure

2.9- Projection

La projection est la transformation qui permet de donner la position du point image sur le plan à partir d'un point dans l'espace.

Le principe de la projection est représenté à la figure 26.

Les nouvelles coordonnées de l'objet sont présentées sous le bon angle, devant la caméra qui est placée sur l'origine du repère grâce aux translations effectuées. On se propose alors de calculer les intersections entre l'écran de l'ordinateur et les droites définies par notre regard d'une part et chaque sommet du triangle "objet" d'autre part. Ainsi, les coordonnées X et Y obtenues sont les coordonnées à partir desquelles il faudra tracer le triangle sur l'écran.

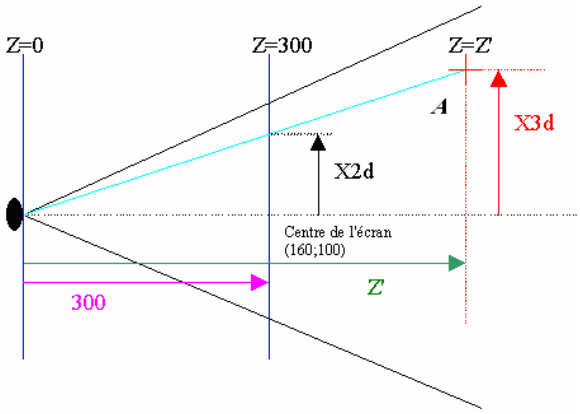

Figure 26: Principe de la projection

Les coordonnées du point sont X', Y' et Z'. On trouve dans une configuration de Thalès, que 300 / Z' = X2d / X3d .

D'où: (300 * X3d) / Z' = X2d où la constante 300 représente la distance supposée entre notre œil et la surface de l'écran.

2.10- Rastérisation

La rastérisation est l'étape transformant les formes géométriques 3D en des pixels sur l'écran tout en donnant un aspect réel à l'objet 3D en question.

La solution la plus simple pour effectuer le rendu d'une surface consiste à calculer l'illumination en chaque point visible de la surface. Cette méthode est très coûteuse en temps de calcul. Dans cette partie, nous allons voir les différentes méthodes permettant de diminuer le coût en temps en n'effectuant le calcul d'illumination qu'en un nombre limité de points.

2.10.1- Ombrage plat

La méthode d'ombrage la plus simple pour les facettes polygonales est l'ombrage plat. Elle consiste à calculer l'intensité de couleurs pour un seul point de la surface que l'on veut représenter. Ensuite, on applique la même intensité pour toute la surface. La figure 27 montre une application d'ombrage plat sur une sphère.

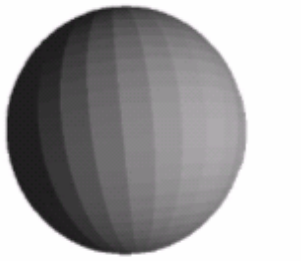

Figure 27: Application d'ombrage plat sur une sphère

2.11- Graphe de tâches de l'application 3D

Dans le processus de conception d'une application, la première étape consiste à décrire le comportement souhaité. C'est la phase de spécification. Cette tâche, essentielle, peut s'avérer extrêmement difficile dans le cas de systèmes embarqués assez complexes.

On a décomposé l'application de traitement d'images 3D en 11 tâches. A la figure 28, nous proposons un graphe de tâches modélisant cette application :

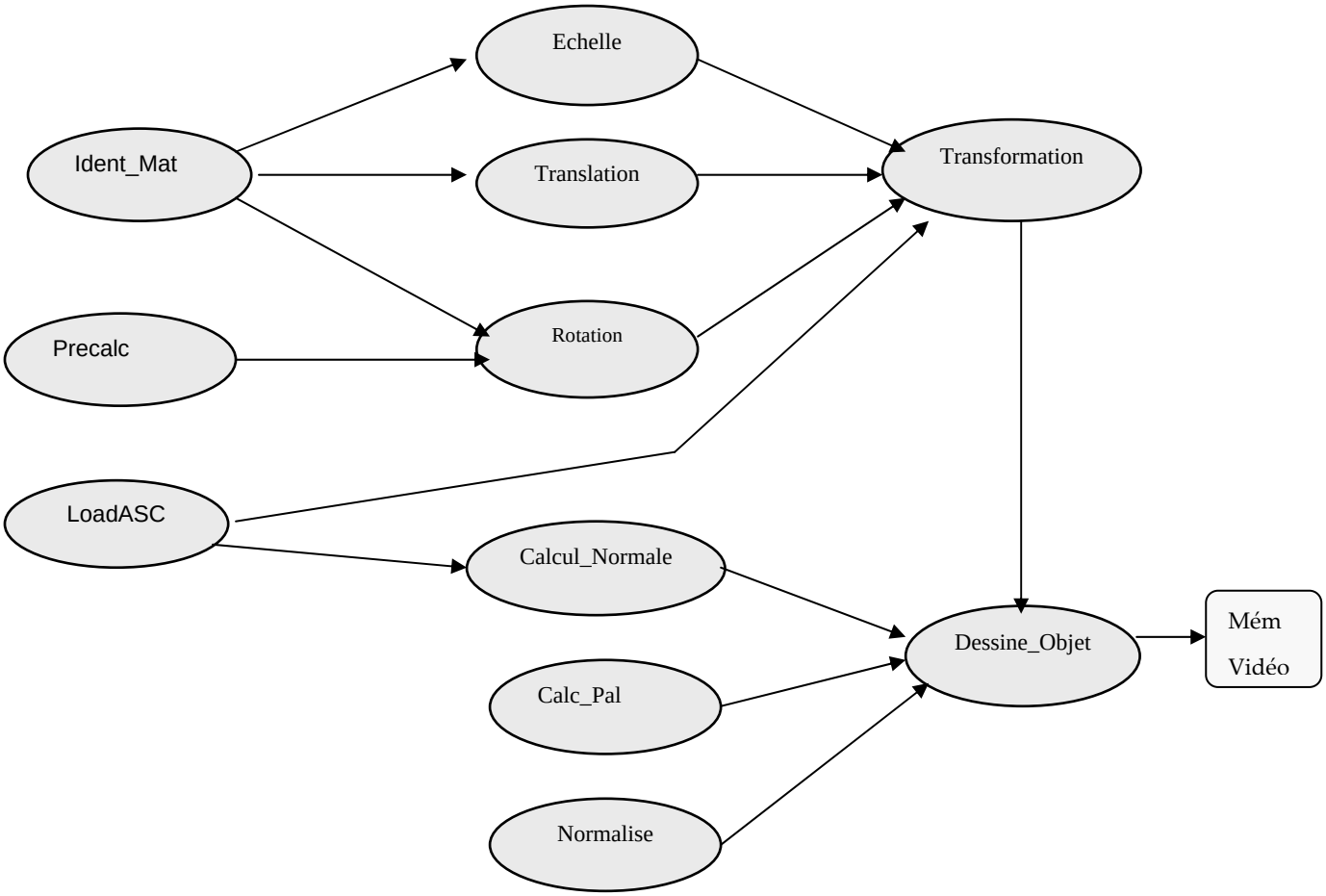

Figure 28: graphe de taches de l'application 3D

3- Validation du modèle d'estimation proposé

Par manque de disposition d'un outil d'estimation du temps d'exécution pour notre environnement de travail, on a déterminé le temps d'exécution de l'application de traitement d'images 3D, sans utiliser des services d'un RTOS par l'exécution directe sur la carte, mais en utilisant un module hardware appelé Timer qui permet de calculer le nombre de tics nécessaires à l'exécution de l'application complète.

Après l'exécution, on a obtenu le résultat suivant : N=**1066889330** tics.

On a procédé, ensuite, à l'extraction d'un diagramme de séquences à partir du graphe de tâches déjà construit tout en décrivant les mécanismes adoptés pour assurer la communication et la synchronisation des différentes tâches, afin de réaliser la fonction globale du système.

Le graphe de séquences suivant est le modèle qui sera adopté pour réaliser tous les tests.

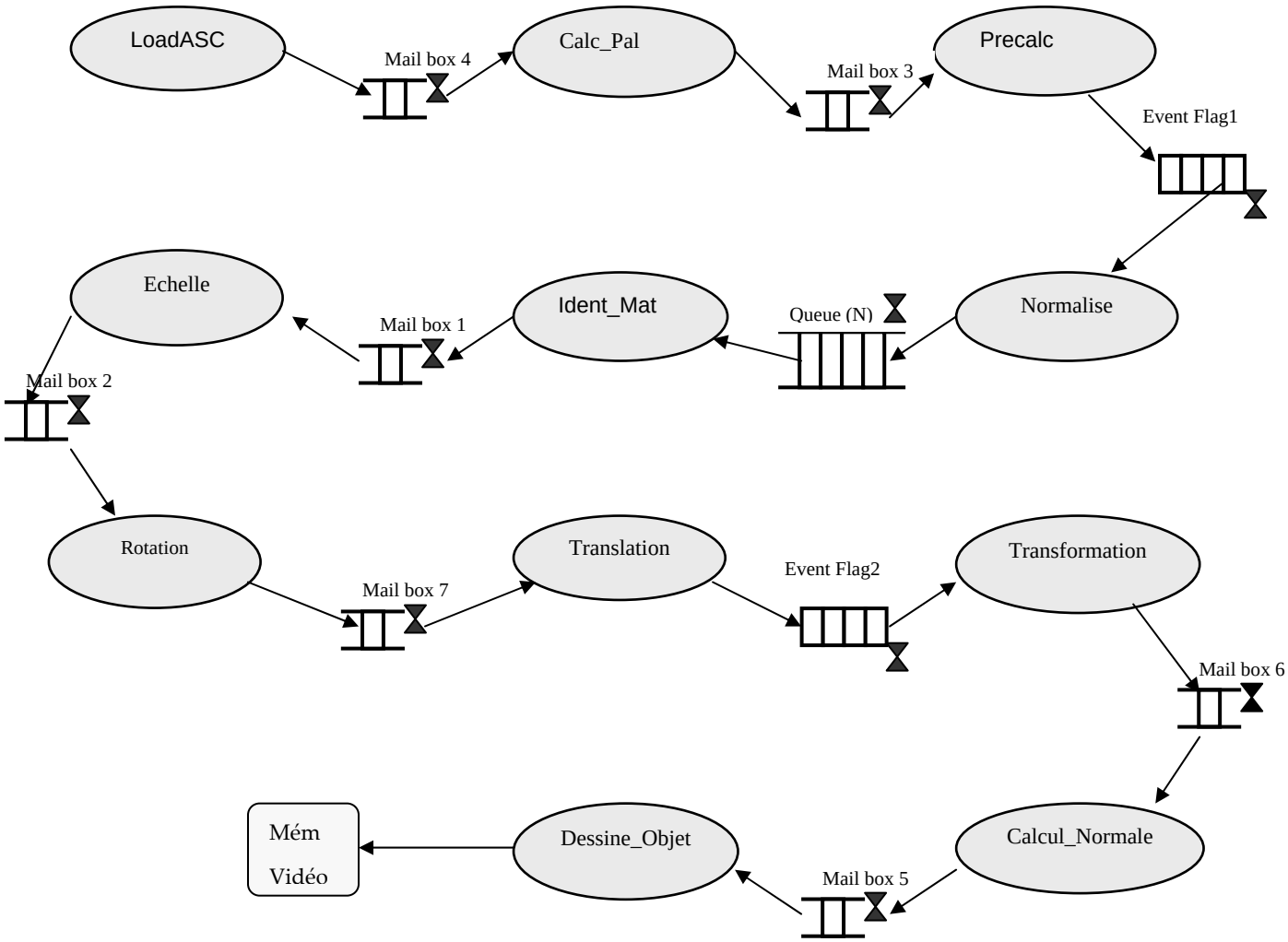

Figure 29: Diagramme de séquences de l'application 3D

Afin d'évaluer le modèle d'estimation proposé, on va essayer de comparer le temps d'exécution de l'application obtenu par exécution directe sur notre plateforme de travail et celui trouvé en appliquant le modèle proposé.

- Quand cette application temps réel a été implémentée et testée sur notre plateforme de travail (NIOS II + MicroC/OS-II) nous avons obtenu le résultat suivant : **1074048767** tics
- Calcul du temps d'exécution en utilisant le modèle trouvé :

Ntr= N * 1.0038

+ 11 temps de création de tâches

+ 7 temps de création de mailbox

+ 2 temps de création de eventFlag

+ 1 temps de création d'un message queue

+ 1 temps de la fonction OSStart()

+ 7 OSMboxPost avec changement de contexte

+ 2 OSEventFlag avec changement de contexte

+ OSQPost avec changement de contexte

Ntr = 1066889330 * 1.0038 + 11 * 9756 + 7 * (679 + 3418) + 2 * (419 + 4464) + 1565 + 3966 + 827 = **1071095628** tics

Calcul du pourcentage d'erreurs : (1074048767-1071095628) / 1074048767 = **0.27%**

Comme on le constate, le taux d'erreurs est très faible ; ce qui valide le bon fonctionnement de notre modèle.

4- Conception de coprocesseurs

Pour la conception des coprocesseurs, il faut disposer du code VHDL d'opérations à implémenter. Pour notre application, nous avons préféré l'utilisation de quatre coprocesseurs : l'addition, la soustraction, la multiplication et la division.

Pour ajouter des coprocesseurs à notre système, nous utilisons le SOPC Builder et, à l'intérieur de la fenêtre du paramétrage du CPU dans le menu « *Custum Instruction* », nous ajoutons les codes VHDL des coprocesseurs, puis nous faisons la lecture de leurs ports afin de pouvoir les ajouter, par la suite, au système, tout en appuyant sur le bouton « *add to System* »

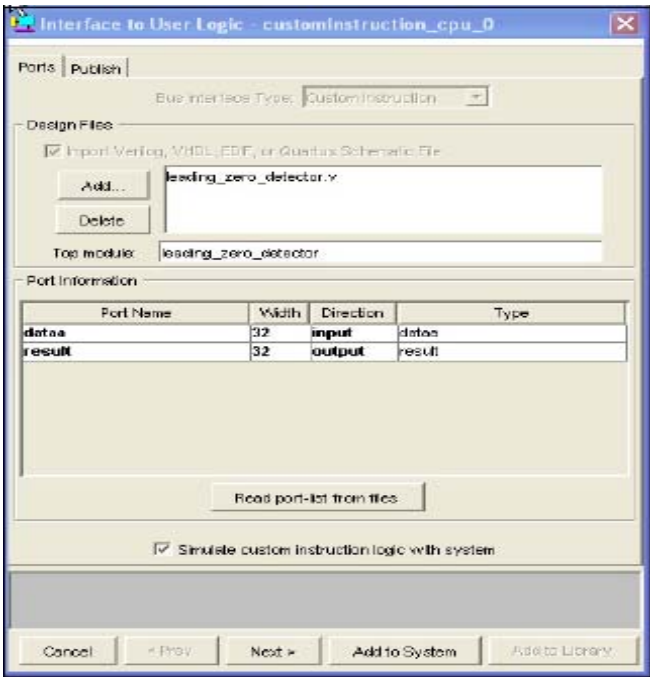

Figure 30: Ajout des coprocesseurs

Tout le code de l'application a été modifié pour remplacer les opérateurs arithmétiques utilisés par l'utilisation des coprocesseurs. Les résultats obtenus sont les suivants :

- Sans utilisation de RTOS : **716576082** tics
- Avec un RTOS : **720223958** tics

5- Conception d'accélérateurs pour le traitement d'images 3D

5.1- Détermination de la normale à une face

Pour calculer la normale à une face triangulaire, on prend les trois vecteurs délimiteurs du triangle, dans le sens des aiguilles d'une montre. On soustrait celui du milieu des 2 autres, et on obtient 2 vecteurs dont le produit vectoriel est la normale de la face.

Soit les trois vecteurs $\vec{V_1}:\begin{pmatrix} x_a \\ y_a \\ z_a \end{pmatrix}$, $\vec{V_2}:\begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix}$ et $\vec{V_3}:\begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix}$, on détermine $\vec{V}_{n1}$ et $\vec{V}_{n2}$ tel que :

$$\vec{V}_{n1}:\begin{cases} V_{n1.x} = x_a - x_b \\ V_{n1.y} = y_a - y_b \\ V_{n1.z} = z_a - z_b \end{cases} \qquad \vec{V}_{n2}:\begin{cases} V_{n2.x} = x_c - x_b \\ V_{n2.y} = y_c - y_b \\ V_{n2.z} = z_c - z_b \end{cases}$$

La normale de la face est $\vec{N}\begin{pmatrix} Nx \\ N_y \\ Nz \end{pmatrix}$ tel que : $\begin{cases} N_x = V_{n1.y} \times V_{n2.z} - V_{n2.y} \times V_{n1.z} \\ N_y = V_{n1.z} \times V_{n2.x} - V_{n2.z} \times V_{n1.x} \\ N_z = V_{n1.x} \times V_{n2.y} - V_{n2.x} \times V_{n1.y} \end{cases}$ (E1)

Pour la réalisation du circuit de calcul de la normale, on propose le schéma suivant:

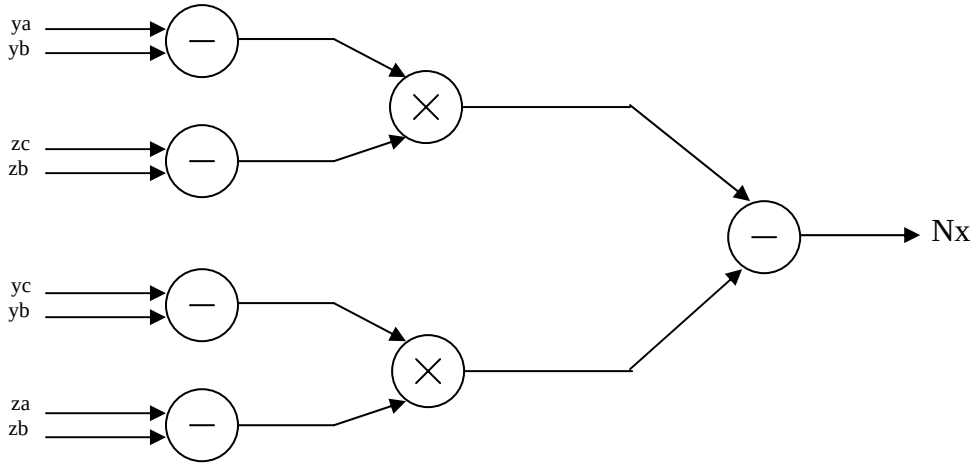

Figure 31: Schéma bloc du module de calcul normal

5.2- Projection

La projection permet le passage du coordonné du monde au cordonné d'écran. Le code effectuant cette opération est le suivant

*ecran = monde * DISTANCE / monde + MX*

On propose le circuit suivant pour la réalisation de cette fonction sous forme d'accélérateur matériel :

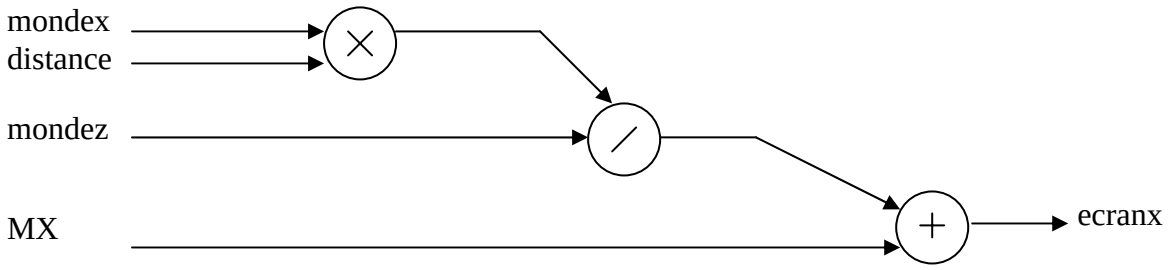

Figure 32: Schéma bloc du module de Projection

5.3- Produit vectoriel

Pour le calcul du produit vectoriel de deux vecteurs v1(x, y, z) et v2(x, y, z) on a recourt au traitement suivant :

vx = (v1y * v2z) - (v2y * v1z)

vy = (v1z * v2x) - (v2z * v1x)

vz = (v1x * v2y) - (v2x * v1y)

On propose le circuit suivant pour la réalisation de cette fonction sous forme d'accélérateur matériel :

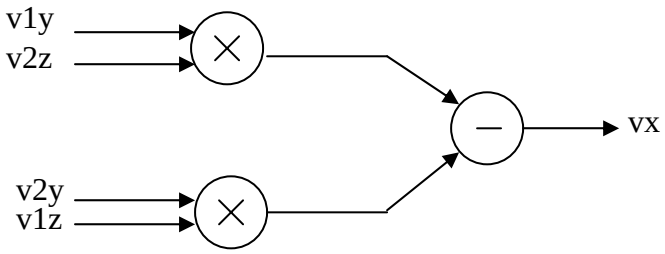

Figure 33: Schéma bloc du module du produit vectoriel

5.4- Transformation

La fonction de transformation permet le passage des coordonnées de tous les sommets des coordonnées locales aux coordonnées du monde, et l'application de la matrice de transformation globale à toutes ces coordonnées.

On propose le circuit suivant pour la réalisation de cette fonction sous forme d'accélérateur matériel :

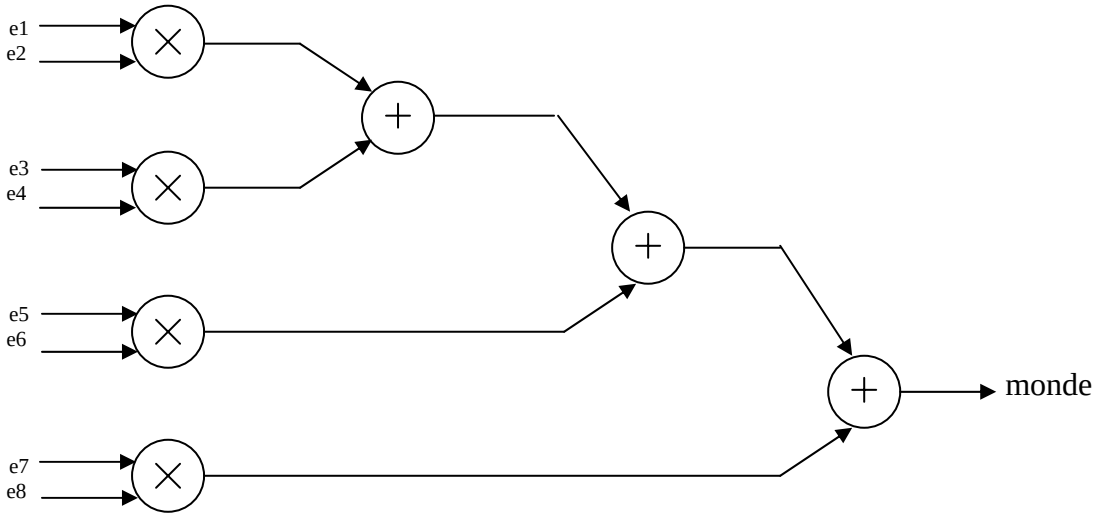

Figure 34: Schéma bloc du module de transformation

5.5- Interconnexion processeur accélérateur

5.5.1- Différentes méthodes d'interconnexion

Le SOPC fournit trois méthodes [9] pour l'ajout des modules propres à l'utilisateur :

- PIO (Parallel Input/Output)

  Les PIOs jouent le rôle d'interfaces entre le logiciel et les modules utilisateurs.

  Une PIO a deux fonctions distinctes :

  - Fournir une interface entre la partie logicielle et la logique utilisateur au sein d'un même circuit.
  - Fournir une interface entre le logiciel et la logique utilisateur, définie à l'extérieur du circuit du NIOS.

- Interface avec les modules utilisateurs *« user defined interface »*

Cette fonctionnalité permet de définir l'interface entre le périphérique utilisateur et le bus AVALON du processeur NIOS. C'est une interface que l'utilisateur peut adapter selon ses besoins, pour pouvoir établir des communications entre le NIOS et les modules qu'il a au préalable définis.

Cette interface est automatiquement générée par l'outil SOPC Builder.

- Ajout d'un composant à la bibliothèque : cette fonction permet de définir un module qu'on peut ajouter à la bibliothèque du QUARTUS II par le développement d'un fichier (d'extension **.ptf)** qui comporte tous les composants utilisés, ainsi que leur liaison et fonctionnement. Par la suite, on peut ajouter ce module au NIOS comme si l'on ajoute un composant prédéfini dans l'environnement SOPC.

Dans notre projet, nous avons préféré de travailler avec les PIO.

5.5.2- Interconnexion à travers les PIO

La procédure de l'interconnexion à travers les PIO est la suivante :

- On ajoute le nombre nécessaire de PIO pour relier les composants au processeur tout en définissant la taille du bus de données et les types d'interruptions matérielles pour les entrées afin de pouvoir récupérer le résultat.
- On génère l'ensemble (processeur, PIO) dans le but d'ajouter les broches PIO dans le schéma bloc du Standard.
- On termine par l'interconnexion des entrées/sorties de l'accélérateur avec le CPU NIOS II par l'intermédiaire des PIO. La figure suivante représente le schéma du système et les quatre modules d'accélération matérielle déjà conçus .

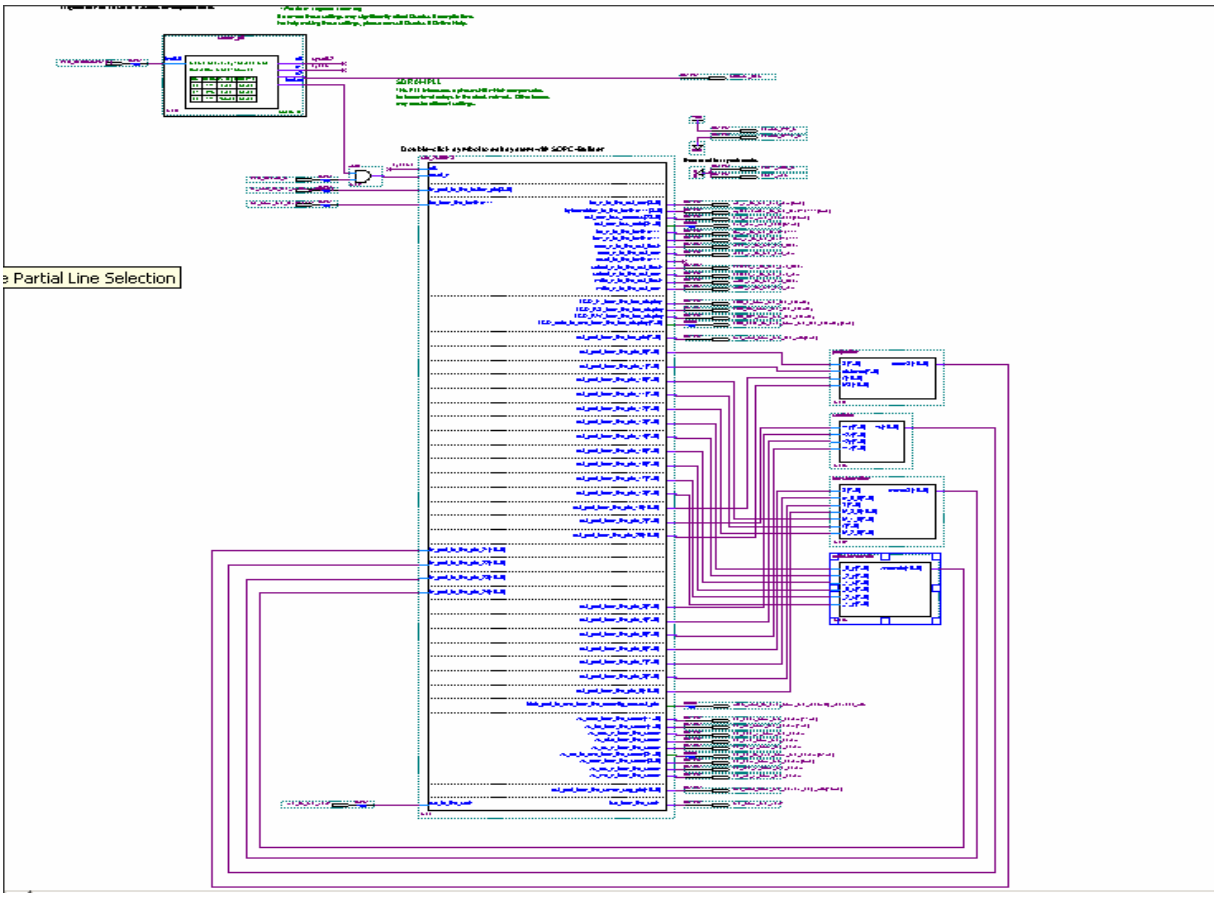

Figure 35: Schéma du système et des accélérateurs

5.6- Compilation

Une fois toutes les connexions établies, une étape de compilation est nécessaire pour s'assurer si le schéma bloc ne contient pas d'erreurs (manque de liaison, tailles du bus de données de deux composants connectés non égales…). Une fois la compilation terminée avec succès, le fichier de configuration (extension .sof) se crée et sera, par la suite, envoyé vers la carte STRATIX II pour le prototypage.

5.7- Mesure accélérateurs

Les accélérateurs sont considérés comme des boîtes noires par le système. Donc, pour utiliser un accélérateur, le CPU envoie les données nécessaires à ce dernier et attend le résultat. Deux types d'instructions sont utilisés dans le développement de la partie logicielle : l'une pour l'envoi des données et l'autre pour la réception Pour se faire, on utilise des pointeurs sur les adresses des ports des accélérateurs.

Les mesures effectuées dans cette partie nous ont conduit aux résultats suivants :

Le temps global de l'application de traitement d'images 3D en utilisant les quatre modules d'accélération déjà présentés est :

- sans utiliser un RTOS : **707680249** tics
- avec un RTOS : **707872710** tics

6- Conception d'un système réactif embarqué multiprocesseurs

6.1- Création du système hardware

La conception des systèmes sur puce multiprocesseurs est rendue plus simple grâce aux deux services de communication implémentés en hardware fournis comme IP avec l'environnement d'Altera le mutex et le mailbox.

6.1.1- Démarche à suivre pour la réalisation d'une plateforme multiprocesseurs

Dans cette partie, on commencera la conception de notre plateforme multiprocesseurs, soit en partant du système monoprocesseur déjà réalisé, ou en utilisant l'exemple de système monoprocesseur « *standard* » fourni avec l'environnement d'Altera et l'étendre pour le rendre multiprocesseurs. Pour ce, on doit procéder à la démarche suivante :

- Prendre une copie du projet monoprocesseur existante
- Ouvrir le projet monoprocesseur par le logiciel Quartus et lancer le SOPC Builder
- Pour ajouter un deuxième processeur NIOS II à notre système, faire un double clic sur « *Nios II Processor – Altera Corporation* » qui se trouve parmi la liste des composants offerts par notre outil de conception. Ainsi, une fenêtre apparaît à l'écran dans le menu « *NIOS II core* ». Il faut choisir NiosII/s, et, dans le menu « *JTAG Debug Module* » sélectionner Level 1, puis appuyer sur le bouton « *Finish* » (Des messages d'erreurs vont apparaître dans la fenêtre des messages du SOPC. Ceci est dû au fait que le nouveau processeur n'est pas encore connecté aux autres composants du système. Il faut donc laisser ces erreurs en instance pour y revenir dans d'autres étapes).
- Ajouter un autre Timer pour le nouveau CPU ; donc, doubles clics sur le composant « *Timer Interval* » qui se trouve dans la liste des composants et, une fenêtre de paramétrages apparaît à l'écran. On accepte les paramètres par défaut et on appuie sur le bouton « *Finish* »
- Connecter le nouveau Timer au bus « *data master* » du deuxième CPU et déconnecter toutes les autres connexions aux autres processeurs. Si les connexions n'apparaissent pas dans le menu du SOPC, il faudra choisir « *Show Connections* »
- Fixer la priorité du nouveau Timer à 0 pour qu'il soit le hardware le plus prioritaire.

- Faire un double clic sur le composant Mutex pour l'ajouter au système. On accepterait les paramètres par défaut du mutex
- Connecter le Mutex au « *Data Master* » de tous les processeurs du système.
- Ajouter la mémoire qui sera partagée entre les CPU et protégée par le Mutex,faire un double clic sur le composant « *On-Chip Memory* », fixer la taille de la mémoire que l'on veut utiliser (on a proposé dans notre topologie d'utiliser pour chaque processeur un Mutex). Donc, cette étape et l'étape précédente vont être dupliquées.
- Ajouter une autre mémoire au système qui va contenir les mailbox.
- Ajouter le nombre voulu de mailbox au système (cette étape est bien détaillée dans la section 4.3.1 du deuxième chapitre.)
- Connecter le SDRAM et le Ext_ram_bus au « *Data Master* » et « *Instruction Master* » de chaque processeur à travers la matrice de connexion.
- Connecter la mémoire protégée par le Mutex au « *Data Master* » de tous les processeurs et enlever la connexion avec l' « *Instruction Master* » du CPU 1.
- Dans le menu System, choisir Auto-Assign Base Adresse pour donner à chaque périphérique une adresse unique.

La figure suivante représente le système obtenu, au cas où on a suivi les étapes ci-dessus :

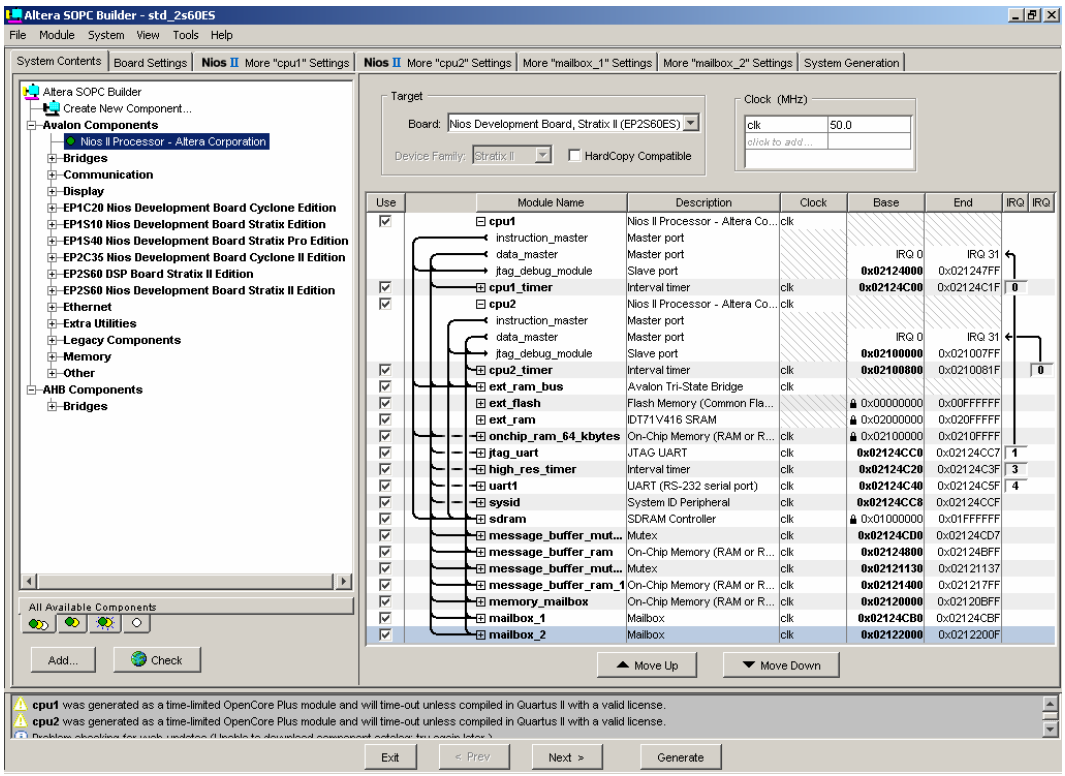

Figure 36: Conception du système par le SOPC Builder

- Spécifier l'adresse de Reset et des Exceptions pour les deux CPU dans le menu « *Nios II More 'cpu' Setting* ».
- Générer le système en appuyant sur le bouton « *Generate* ».Si la génération est achevée avec succès, on quittera le SOPC Builder,et un message apparaîtra directement sur l'écran pour inviter le concepteur à mettre à jour, son système, afin d'afficher les modifications faites.
- Une compilation est nécessaire à cette étape pour vérifier le système complet. Si la compilation se termine avec succès, cela veut dire que notre projet est prêt pour être téléchargé dans le FPGA.

A ce stade, il ne reste qu'à développer la partie logicielle pour tester notre architecture multiprocesseurs.

6.2- Développement de la partie software

Pour le développement de la partie software qui servira à tester notre plateforme multiprocesseurs, on procèdera de la même manière que celle pour le monoprocesseur, puisque chaque CPU exécutera son propre code. Il faut tenir compte qu'un seul processeur peut afficher des messages sur l'écran. Donc, pour que les autres processeurs puissent afficher des messages, ils doivent les envoyer à ce processeur pour qu'il les affiche.

On utilise Nios II IDE pour le développement de la partie software pour une plateforme multiprocesseurs.

A présent, comment agir pour exécuter simultanément les codes développés sur les processeurs spécifiques ? Pour surmonter ce problème, il faut procéder à la démarche suivante :

- Lors de l'étape de création du projet, on doit spécifier le processeur sur lequel il va être exécuté (figure 37).

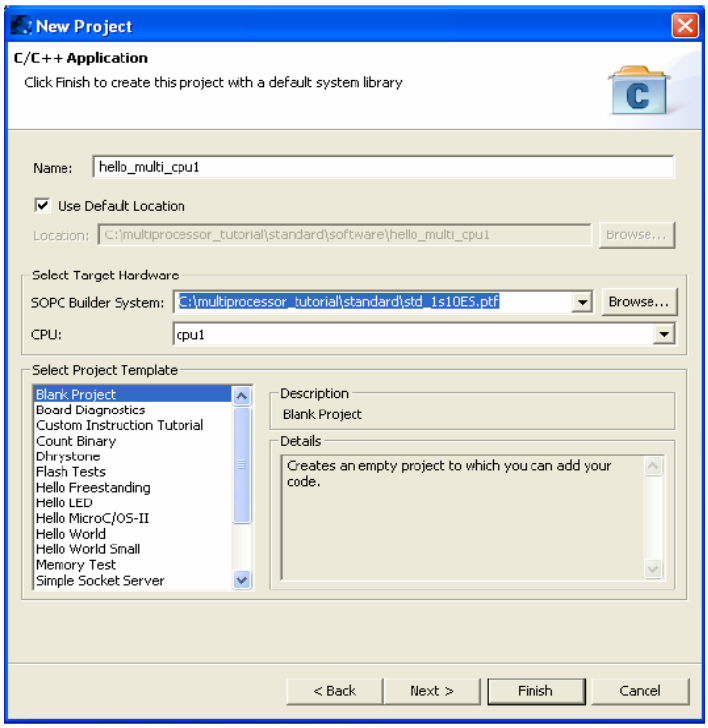

Figure 37: Choix du processeur

- Dans la fenêtre « *system Library* », il faut selectionner le Timer utilisé pour chaque système dans « *System Clock Timer* ».

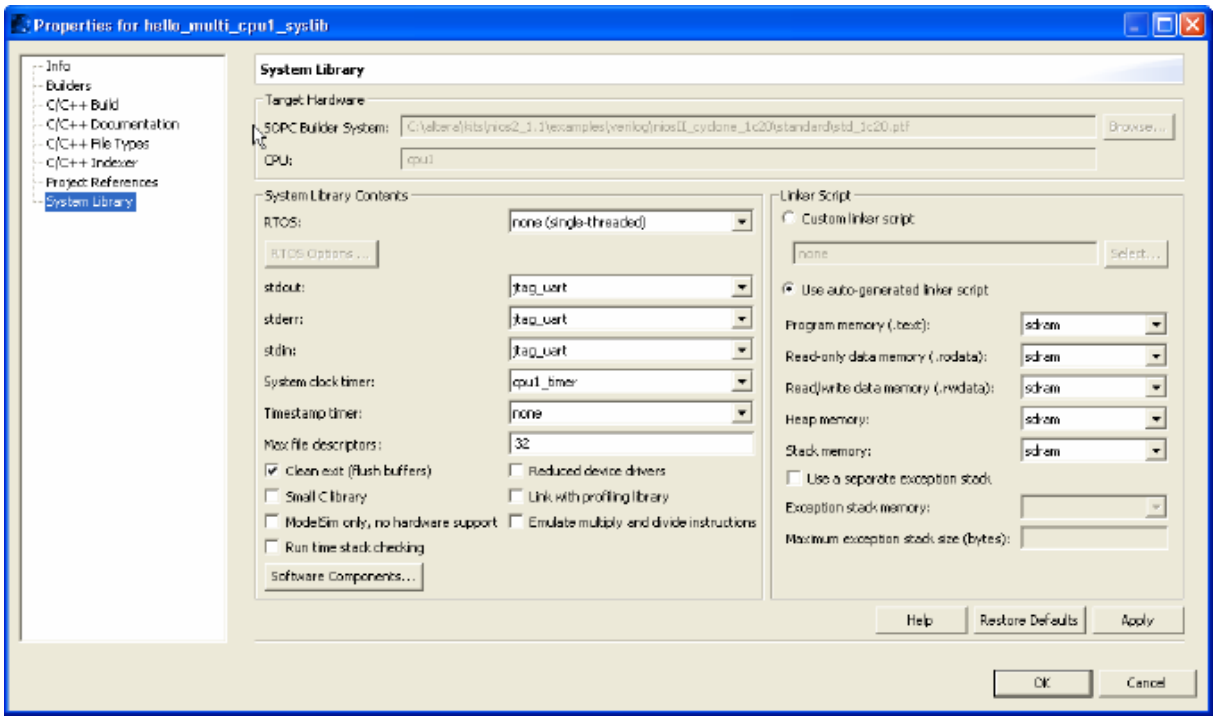

Figure 38: Choix du timer

- Builder les projets, l'un après l'autre, en accédant au menu « *project* » puis « *Build Project* »
- Une fois, cette tâche terminée avec succès, on passe à la création de la configuration du Debug pour chaque processeur. On sélectionne le projet et dans le menu Run, on choisit « *Run…* » . Une fenêtre apparaît. Alors, on clique sur le bouton « *new* » puis « *Apply* » et « *Close* ». On refait de même pour tous les projets.
- Pour exécuter le tout sur notre plateforme multiprocesseurs dans le menu « *Run* », on choisit « *Run…* » et on sélectionne dans la fenêtre qui apparaît « *Nios II Multiprocessors Collection* ». Puis, on appuie sur le bouton « *New* » et on coche les projets qu'on veut exécuter. Enfin on clique sur « *Run* » (figure 39). Ainsi, la partie sera exécutée sur la plateforme réalisée.

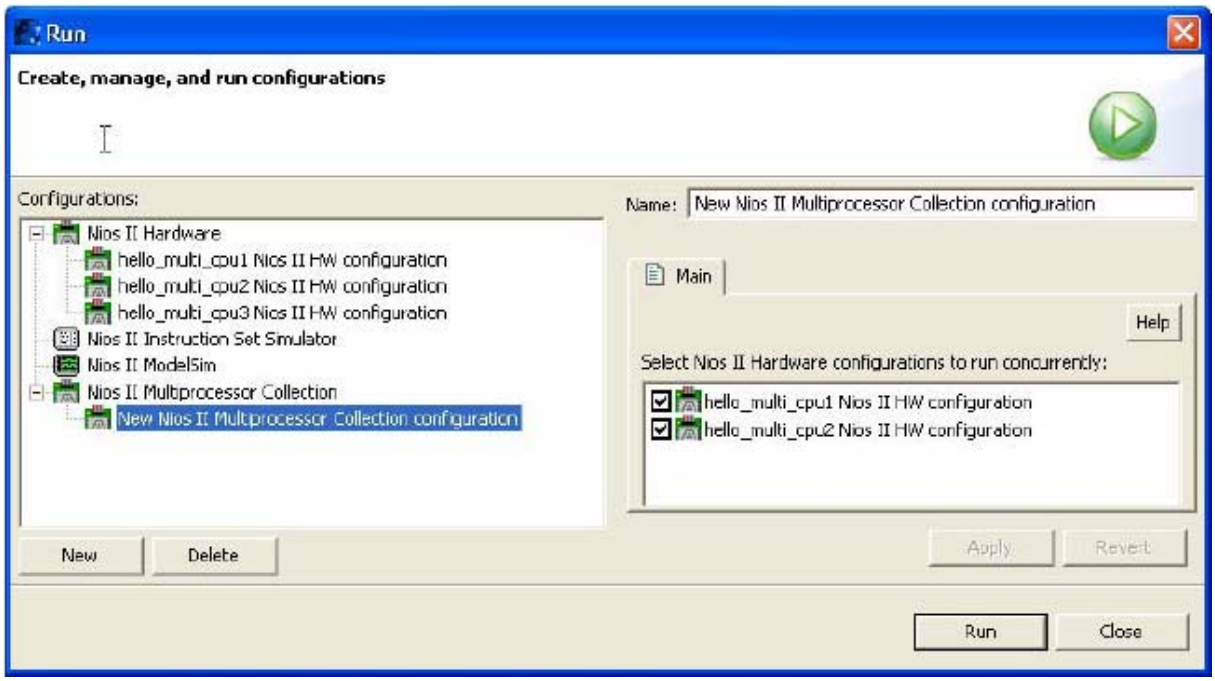

Figure 39: Exécution sur une architecture multiprocesseurs

6.3- Exécution de l'application de traitement d'images 3D sur la plateforme multiprocesseurs

Pour développer n'importe quelle application sur une plateforme multiprocesseurs, il faut essayer de minimiser au maximum, la communication entre les processeurs qui peuvent ralentir le temps d'exécution de notre application. Pour pouvoir répartir une application sur

plusieurs processeurs, il faut décomposer l'application en un ensemble de fonctions et on détermine ceux qui peuvent s'exécuter en parallèle. Afin de faire un bon choix, il faut qu'on précise, dès le début, les informations échangées entre les différentes fonctions.

Le graphe suivant illustre la décomposition de l'application de traitement d'images 3D en un ensemble de fonctions ainsi que les données échangées entre elles :

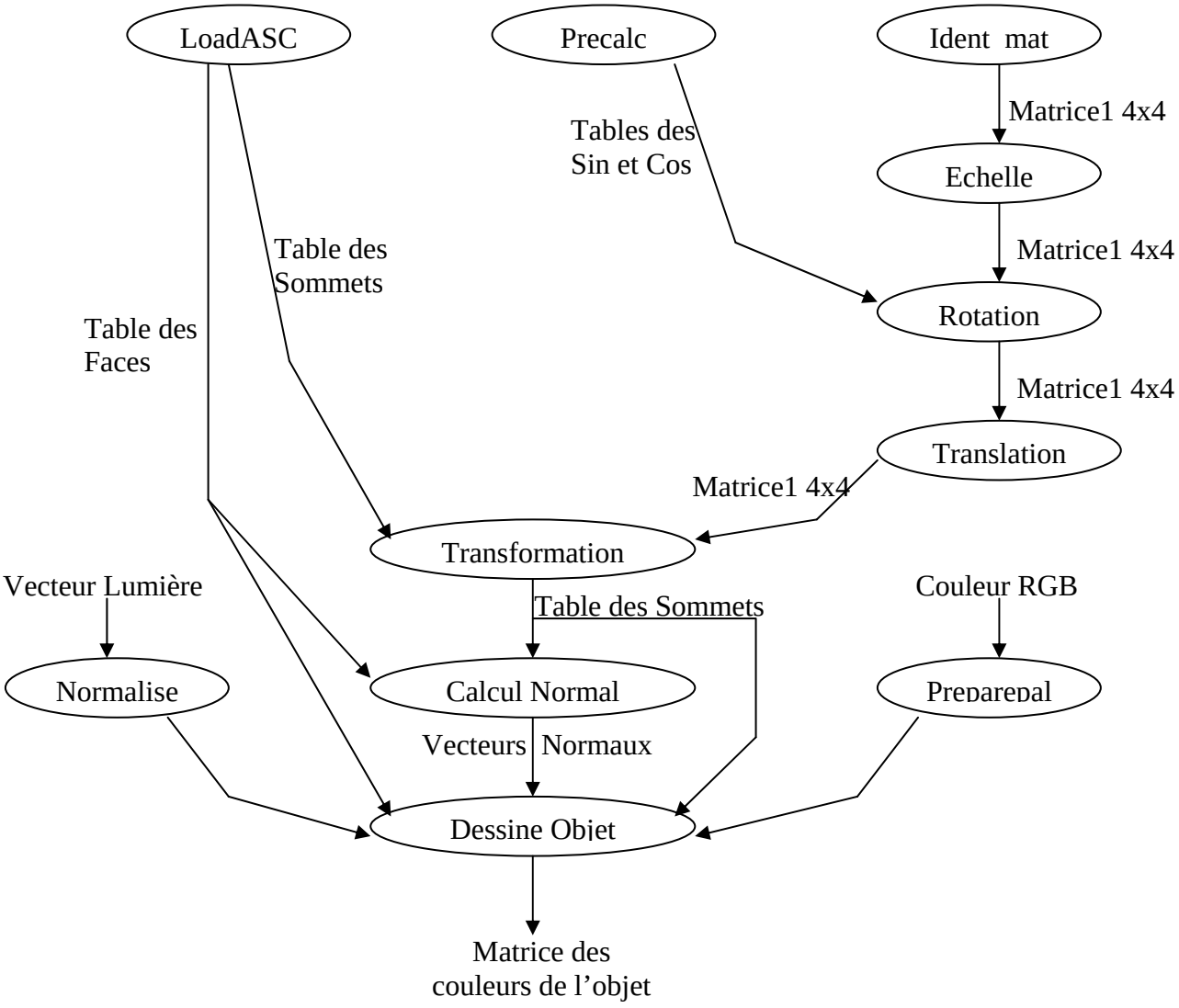

Figure 40: Graphe de dépendance de données de l'application 3D

Comme on le constate dans la figure ci-dessus, la plupart des fonctions de l'application 3D se font d'une manière séquentielle. Donc, pour exécuter cette application sur notre plateforme multiprocesseurs qui contient deux processeurs CPU1 et CPU2, on a proposé de:

- exécuter la fonction « *precalc* » qui permet de remplir deux tableaux sin et cos, sur le cpu2 qui va envoyer le résultat de cette fonction au processeur CPU1.
- exécuter les fonctions « *ident_mat* » et « *echelle* » sur le CPU1.
- utiliser le résultat de la fonction « *precalc* » fourni par le CPU2 pour exécuter la fonction « *rotation* » sur le CPU1.
- exécuter la fonction « *translation* » sur CPU1 pour que notre matrice de transformation globale soit prête en vue de l'appliquer à tous les sommets des polygones qui construisent notre objet 3D
- envoyer la matrice de transformation globale au CPU2.
- exécuter la fonction « *loadASC* » sur les deux processeurs. Le CPU1 va télécharger la première moitié des sommets et des faces qui construisent l'objet, alors que le deuxième processeur téléchargera la deuxième moitié.
- terminer l'exécution du reste de l'application pour la moitié du nombre de polygones sur chacun des processeurs (transformation, Normalise, calcul normal, preparepal et dessine objet).
- lorsque le deuxième processeur termine ses calculs, il envoie un message le signaler au CPU1.

Les valeurs suivantes représentent les mesures du temps total d'exécution de l'application 3D sur notre plateforme multiprocesseurs (2cpu) :

3D avec deux processeurs sans rtos : **712248263** tics

3D avec deux processeurs avec rtos : **717162554** tics

Ces mesures sont faites avec l'emploi des quatre coprocesseurs sur chaque processeur :

3D avec deux processeurs avec coprocesseurs sans rtos : **558288052** tics

3D avec deux processeurs avec coprocesseurs avec rtos : **570376431** tics

7-Interprétation

	1 cpu	1cpu+coproc	1cpu+acc	2 cpu	2cpu + coproc
Total ALUTS	3505 (7%)	4676 (9%)	4254 (8%)	9020 (18%)	9153 (18%)
Total memory bits	571136 (22%)	571136 (22%)	571136 (22%)	657920 (25%)	657920 (25%)
Occupation mémoire sans RTOS	204 Ko	197 Ko	199 Ko	Cpu1:193 Ko Cpu2:161 Ko	Cpu1:190 Ko Cpu2:168 Ko
Occupation mémoire avec RTOS	339 Ko	332 Ko	336 Ko	Cpu1:328 Ko Cpu2:263 Ko	Cpu1:322Ko Cpu2:254 Ko
Temps d'exécution sans RTOS	1066889330	716576082 **(-32 ,83%)**	707680249 **(-33,66%)**	712248263 **(-33,24%)**	558288052 **(-47,67%)**
Temps d'exécution avec RTOS	1071095628	720223958	707872710	717162554	570376431
Temps d'exécution avec modèle	1074048767	719451189	710521551	715106924	560561644
Erreur du modèle	**0.27%**	**0.1%**	**0.37%**	**0.28%**	**1.72%**

Tableau 9: Performances des architectures proposées

On constate bien, d'après le tableau ci-dessus, qu'en utilisant deux processeurs, on aura le même gain obtenu qu'avec des accélérateurs ou des coprocesseurs, en terme de temps d'exécution, mais, en contre partie, on trouvera qu'on a perdu, en terme de surface du circuit, vue l'augmentation du nombre de ressources utilisées.

Peut-on dire que l'utilisation des systèmes multiprocesseurs est inutile en comparant ces performances avec celles obtenues en utilisant des coprocesseurs ou des accélérateurs ?

D'après moi, la réponse est « non » pour plusieurs raisons. En premier lieu, on trouve que les accélérateurs et les coprocesseurs sont spécifiques à des traitements bien précis : donc ce type d'architecture est non flexible. En second lieu, et d'après le tableau 9, on constate qu'on a atteint 50% de gain sur le temps d'exécution en utilisant une architecture qui comporte deux processeurs et quatre coprocesseurs sur chacun d'eux. Donc, si on a besoin de la rapidité du traitement, il sera souhaitable de combiner les solutions pour profiter de l'avantage de chacune d'elles.

Pour conclure, le choix de l'architecture dépend des contraintes imposées par l'application tel que la rapidité, la surface, la consommation, la flexibilité etc.

8- Conclusion

Dans ce chapitre, on a présenté les différentes étapes de conception d'un système monoprocesseur temps réel. Ensuite, on a généré un modèle d'estimation de performances des applications temps réel. Ce modèle a été validé à travers l'application de traitement d'images 3D. On a terminé par la conception d'une plateforme de prototypage des systèmes réactifs/multiprocesseurs reconfigurables dont la validation a été faite par la même application.

Conclusions et perspectives

Les systèmes embarqués sont de plus en plus complexes et l'architecture multiprocesseurs ne va pas tarder à être un choix crucial lorsqu'il s'agit d'un système puissant et temps réel.

En plus, et du fait de la présence de fortes contraintes temps réel, de la limitation des ressources disponibles, tant en mémoire qu'en énergie disponible et donc en puissance de calcul, mais également de la pression exercée par le marché sur ces produits, l'usage de systèmes d'exploitation temps réel est devenu nécessaire dans les systèmes embarqués.

Ces évolutions ont rendu indispensable l'adaptation des systèmes d'exploitation temps réel afin de les rendre compatibles avec les nouvelles architectures des Systèmes Embarqués d'exploitation pour faciliter la gestion des événements et gérer la réactivité de ces systèmes.

Le travail entrepris dans ce stage de mastère a permis d'étudier de prêt les contraintes et les problèmes engendrés par le Prototypage des systèmes réactifs/multiprocesseurs sur des architectures reconfigurables. En premier lieu, des études bibliographiques sur les architectures des systèmes multiprocesseurs, les systèmes d'exploitation temps réel utilisés dans le cadre des systèmes sur puce et les différentes méthodes d'estimation de performance ont été faites pour explorer le domaine et avoir une idée sur leurs caractéristiques.

En deuxième lieu, une étude sur l'environnement EXCALIBUR d'ALTERA a été faite pour explorer et bénéficier des services offerts par cet environnement. On a pu proposer après ces études une topologie pour une architecture multiprocesseurs et on a généré un modèle d'estimation de performance dans le cadre des systèmes sur une puce temps réel.

La partie pratique du travail a visé le test et la validation du modèle d'estimation de performance proposé et de la plateforme mono/multiprocesseurs à travers l'application de traitement d'images 3D.

L'ensemble des étapes de ce travail nous a permis, entre autre, de maîtriser quasiment toutes les étapes du flot du Codesign et d'expérimenter l'implantation d'un système embarqué mono/multiprocesseurs temps réel sur une cible de prototypage et cela nous a incités à dégager plusieurs réflexions sur de futurs travaux :

- *La génération automatique du code des applications temps réel.*
- *La conception et la mise en place d'un superviseur sur les différents maîtres de l'architecture multiprocesseurs élaborée. En fait, le rôle de ce superviseur est d'affecter les tâches sur les maîtres du système de façon transparente, automatique et dynamique, etc.*

- *Le développement d'un outil d'exploration d'architecture qui peut aider le concepteur à prendre la décision pour le choix de l'architecture du système à partir du code de l'application.*

Bibliographie

[1] H.Krichene « **Conception des RTOS dans les SoS : Etude de cas** », 2004.

[2] T. Pop, P. Eles et Z. Peng, « **Holistic Scheduling and Analysis of Mixed Time/EventTriggered Distributed Embedded Systems** », Conférence internationale: « on Computer Aided Design », 2001.

[3] S. Nasri, Note de cours « **Introduction aux systèmes temps réel** », Avril2001, Version 1,ENIM.

[4] F.Skivee, « **Les systèmes temps réel** », cours en juin 2002.
http://www.montefiore.ulg.ac.be/~skivee/download/rapport/rapportpdf.html

[5] « **Documentation du RTOS : RTX 5.1** », copyright CKeil Software.

[6] Samy MEFTALY : « **Exploration d'architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs monopuces** »

[7] Amer BAGHDADI: « **Exploration et conception systématique d'architectures multiprocesseurs monopuces dédiées à des applications spécifiques** ».

[8] CM, « **The CM-5 Connection Machine : A Scalable Supercomputer** », W. Daniel Hillia and Lewis W.Tucker. Communication of the ACM, November 1993, Vol. 36, No. 11.

[9] P. Parrend, « **RTOS et gestion matérielle de la mémoire** », INSA Lyon, Janvier 2005.

[10] D. Harel et A. Pnueli, « **On the Development of Reactive Systems** », Proceeding: «NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems », Springer-Verlag, pages 477-498, 1985.

[11] V. Bertin, « **Validation d'applications temps réel par analyse de programmes synchrones temporisés** », le 20 Octobre 2000.

[12] M.Auguin, « **Les systèmes sur puce : la problématique de la conception descendante, les méthodes et outils associés** », rapport interne, I3S, Université de Nice Sophia Antipolis, CNRS, 2000, France.

[13] INF4600, « **systèmes temps réel** », chapitre1, l'école polytechnique de Montréal.

[14] W. Blachier, **« Le Temps Réel** », Institut national polytechnique à Grenoble, 26 Juin 2000.

[15] D. Donsez, « **Systèmes d'exploitation pour l'embarqué** », université Joseph Fourier.

[16] Lauvic Gauthier, « **Génération de système d'exploitation pour le ciblage de logiciel multitâches sur des architectures multiprocesseurs hétérogènes dans le cadre de systèmes embarqués** », thèse PhD, décembre 2001, TIMA France.

[18] J.J. Labrosse, "**Micro C/OS-II, the Real-Time Kernel**", Second Edition.

[19] Scott S. and Thorson G., "**The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus**", HOT Interconnects IV, Stanford University, August 1996.

[20] Damien LYONNARD: « **Approche d'assemblage systématique d'éléments d'interface pour la génération d'architecture multiprocesseurs** ».

[21] Site du noyau Ecos : http://sources.redhat.com/ecos, 2004.

[22] « **Comparison between QNX RTOS v6.1, VXWorks AE 1.1 and Windows CE.NET** », Copyright Dedicated Systems Experts NV.
http://www.dedicated-systems.com

[23] Y. Benaben, « **Le noyau temps réel uC/OS - uC/OS II** », Stage IUT GEII, 1999.
http:// www.enseirb.fr/~kadionik/68hc11/carte_enserb/ucos-ii.html

[24] P. Mabilleau , «**Systèmes en temps réel** », GEI 455, janvier 2001
http://www.gel.usherb.ca/mab/gei455/index.html.

[25] F. Ghaffari, « **Etude d'un RTOS: µC/OS-II et son portage sur le processeur NIOS** », Rapport de stage DEA 2002, ENIS, page 6-8.

[26] Site du noyau uCO/S : http://www.ucos-ii.com/.2004.

[27] Y. Benaben, « **Le noyau temps réel uC/OS - uC/OS II** » Stage IUT GEII, 1999, http://www.enseirb.fr/~kadionik/68hc11/carte_enserb/ucos-ii.html.

[28] « **Excalibur NIOS** », P.N, ENSEIRB.

[29] IBM, "**The CoreConnect Bus Architecture**" available at:
http://www.chips.ibm.com/products/coreconnect/,2002.

[30] H.J. Eikerling, W. Hardt, J. Gerlach, W.Rosenstiel. " **A Methodology for Rapid Analysis and Optimization of Embedded System**" International IEEE Symposium and workshop on ECBS, pp. 252-259, march 1996.

[31] J. Henkel and R.Emst. "**A Path-Based Estimating Hardware Runtime in HW/SW-Cosynthesis**" IEEE International Symposium on system level synthesis pp. 116-121, 1995.

[32] J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen and A. Haxthausen. «**LYCOS: the Lyngby Co-Synthesis System** » International Conference on Computer design 1995.

[33] D. Gajski, F. Vahid, S. Narayan and J. Gong. " **SpecSyn: An Environment supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design**"

[34] J. Liu, M. Lajolo and A. Sangiovanni-Vincentelli. " **LYCOS: the Lyngby Co-Synthesis System**"

[35] T-Y, Yen and W. Wolf. « **Performance Estimation for Real-Time Distributed Embedded Systems** » International Symposium on System synthesis, 1995.

[36] Projet SOLIDOR : « **Construction de systèmes et d'applications distribués** », Rapport d'activité 2003, INRIA, Rennes.

[37] T-Y. Yen and W. Wolf. « **Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems** » International Symposium on System synthesis, 1995.

[38] F. Skivee : **« Concepts des systèmes d'exploitation temps réel** », cours le 04 Juin 2002.

[39] K. Loukil et H. ben chikha : « **Conception d'accélérateurs pour le traitement d'images 3D** » projet de fin d'étude Juin 2003

[40] A.Topol, « **VRML étude, mise en oeuvre et applications** », Diplôme d'Ingénieur C.N.A.M en Informatique, CNAM Paris, 2001.