

# TP2 : Déploiement de l'application MJPEG sur une architecture SoC monoprocesseur

1. 0. Objectif
2. 1. Description de l'architecture matérielle
3. 2. Déploiement de l'application SplitMsg
4. 3. Déploiement de l'application MJPEG
5. 4. Influence du système d'exploitation embarqué
6. 5. Compte-rendu
7. Suite

TP Précédent : [MjpegCourse/Station](#)

## 0. Objectif

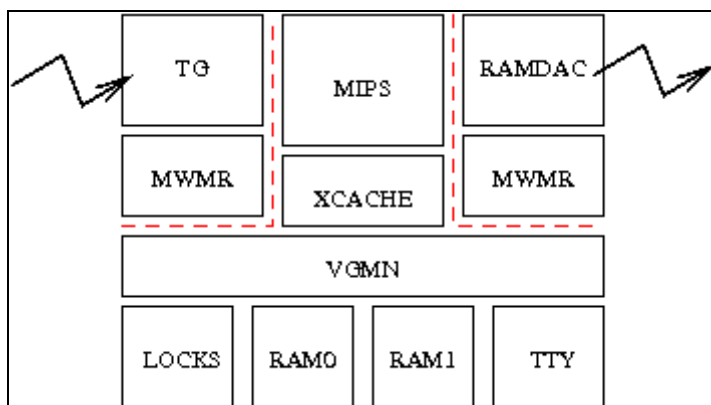
La première partie de ce TP vise à montrer comment décrire - en utilisant le langage DSX/L - une architecture matérielle de système intégré sur puce, en exploitant les composants matériels de la bibliothèque *SoCLib*. On rappelle que la bibliothèque SoCLib contient un ensemble modèles de simulation de composants matériels (IP cores), décrits en langage SystemC. L'intérêt d'utiliser DSX (plutôt que de décrire directement l'architecture en langage SystemC), réside dans la facilité de décrire des architectures génériques (par exemple, à nombre variable de processeurs ou bancs mémoire).

La seconde partie du TP vous permettra d'utiliser DSX pour décrire et contrôler précisément le déploiement de l'application logicielle SplitMsg, sur l'architecture matérielle décrite dans la première partie. On validera ce déploiement en simulant l'exécution du code binaire de l'application logicielle sur le modèle SystemC de l'architecture matérielle.

La troisième partie du TP vous permettra d'atteindre notre véritable but, qui est de déployer l'application MJPEG, vue au TP1, sur l'architecture de SoC monoprocesseur décrite dans la première partie, en contrôlant précisément le placement des tâches sur les processeurs ou coprocesseurs, et le placement des tampons de communication dans les bancs mémoire embarqués.

## 1. Description de l'architecture matérielle

On se limitera dans ce TP à une architecture ne contenant qu'un seul processeur programmable de type MIPS32. Cette architecture matérielle est appelée 'VgmnNoirqMono' (voir schéma de l'architecture ci-contre).



- Elle est organisée autour d'un micro-réseau générique à interface VCI (composant VGMN). Le VGMN est générique dans le sens où il accepte un nombre quelconque d'initiateurs VCI, et un nombre quelconque de cibles VCI, ainsi qu'un paramètre définissant la latence du réseau : c'est-à-dire le nombre minimal de cycles pour *une* traversée du réseau ("one-way").
- Elle comporte un processeur et son cache, deux contrôleurs mémoire RAM0 et RAM1, et un contrôleur de terminal TTY.

**Attention:** pour l'application MJPEG, les deux coprocesseurs matériels d'entrée/sortie TG et RAMDAC, doivent être décrits dans l'architecture VgmnNoirqMono. Cependant, ces deux coprocesseurs, ainsi que les deux contrôleurs MWMR leur permettant d'accéder aux canaux MWMR, ne sont utilisés que par l'application MJPEG, et pas par l'application SplitMsg. Vous pouvez donc les laisser commenter pour le moment.

Commencez par créer un répertoire de travail 'TP2'. Pour faciliter la réutilisation, l'architecture matérielle est généralement décrite dans un fichier séparé. Créez, dans le répertoire TP2, le fichier `vgmn_noirq_mono.py` et saisissez-y le contenu de [VgmnNoirqMono](#). La description que vous venez de saisir est incomplète : il vous faut donc compléter le code à trou fourni (sachant que les trous sont signalés par le texte '`*** remplir ***`') en vous aidant de la documentation [SoCLibComponents](#) qui définit les paramètres des différents composants de la bibliothèque SoCLib.

**?** Q1 : *Quelle est la syntaxe utilisée par DSX pour exprimer que le port  $P_0$  du composant matériel  $C_0$  est connecté au port  $P_1$  du composant matériel  $C_1$ ?*

Une fois la description de la plateforme complète, nous pouvons la tester en générant une description SystemC décrivant la *top-cell*.

- Rendez le fichier de description exécutable et exécutez-le.

Si tout se passe bien, vous devriez avoir un nouveau répertoire nommé `hard` dans le répertoire courant. La description SystemC de la *top-cell* est contenue dans le fichier `hard/topcell_hard/topcell_hard.cpp`.

## 2. Déploiement de l'application SplitMsg

Vous allez commencer par déployer l'application SplitMsg, qui ne comporte que deux tâches et un canal sur votre architecture de SoC monoprocesseur.

- Créez dans le répertoire TP2 un sous-répertoire '`splitmsg`'.
- Recopiez dans ce répertoire la description DSX `vgmn_noirq_mono.py` que vous venez d'écrire.
- Recopiez dans ce répertoire la description DSX de l'application SplitMsg que vous aviez écrite dans le TP1, ainsi que les sources de l'application (répertoire `src`).
- Modifiez la description `SplitMsg` en ajoutant, après la description du TGG, l'instanciation de l'architecture matérielle `VgmnNoirqMono`.

```
#####
# Section B : Hardware architecture
#
# The file containing the architecture definition
# must be included, and the path to the directory
# containing this file must be defined
#####

from vgm_noirq_mono import VgmnNoirqMono

archi = VgmnNoirqMono()
```

- Définissez le mapping de l'application SplitMsg sur l'architecture VgmnNoirqMono. Vous devez consulter la page [DsxMapping](#) pour plus d'informations.

Dans cette section, un objet Mapper doit être créé. Supposons qu'on crée une variable mapper, les objets logiciels doivent être identifiés par leur nom. Il va falloir placer tous les canaux de communication, toutes les tâches, tous les objets logiciels associés aux processeurs et enfin les objets globaux du système (*dans cet ordre*).

```
#####
# Section C : Mapping
#
#####

mapper = dsx.Mapper(archi, tcg)

# mapping the MWMR channel

mapper.map( "fifo0",
  buffer = "cram1",
  status = "cram1",
  desc   = "cram1")

# mapping the "prod0" and "cons0" tasks

mapper.map("prod0",
  run = "cpu0",
  stack = "cram0",
  desc = "cram0",
  status = "uram0")

mapper.map("cons0",
  run = "cpu0",
  stack = "cram0",
  desc = "cram0",
  status = "uram0")

# mapping the software objects associated to a processor

mapper.map( 'cpu0',
  private = "cram0",
  shared = "cram0")

# mapping the software objects used by the embedded OS

mapper.map(tcg,
  private = "cram1",
  shared = "uram1",
  code = "cram1",

  # These lines are for getting output messages:
  tty = "tty0",
  tty_no = 0)
```

- La dernière étape consiste à générer le code

```
#####
# Section D : Code generation
#####

# Embedded software linked with the Mutek/S OS

mapper.generate( dsx.MutekS() )

# The software application for a POSIX workstation can still be generated
```

```
tcg.generate( dsx.Posix() )
```

**?** Q2 : *Quels objets logiciels doit-on placer dans l'espace adressable pour une tâche ? pour un canal mwmr ? pour un processeur ?*

- Relancez l'exécution de la description DSX de votre application :

```
$ ./splitmsg.py
```

- Exécutez l'application logicielle sur la station de travail :

```
$ ./exe.posix
```

- Simulez l'exécution de l'application logicielle sur le modèle SystemC du SoC :

```
$ ./exe.muteks_hard
```

**?** Q3 : *Qu'observez-vous ? En quoi est-ce différent de ce qui se passe dans la version pour station de travail ?*

## 3. Déploiement de l'application MJPEG

L'application MJPEG est différente de l'application SplitMsg car elle utilise deux périphériques d'entrée/sortie spécialisés :

- le coprocesseur Tg : un composant matériel qui récupère le flux binaire MJPEG (fournit par un signal radio-fréquence, par exemple), effectue la conversion analogique/numérique et écrit le résultat dans un canal MWMR
- le coprocesseur Ramdac : un composant matériel qui lit une image décompressée dans un canal MWMR et génère le signal vidéo pour affichage sur l'écran.

Pour pouvoir déployer ces deux tâches sous forme de coprocesseurs matériels, il faut prévenir DSX qu'il existe des coprocesseurs implémentant ces tâches. Vous allez donc modifier les déclarations des modèles de tâches en conséquence.

Retournez dans le répertoire `mjpeg` du TP1, et exécutez la commande `./mjpeg -m clean` qui détruit tous les fichiers générés par les différentes compilations effectuées. Ce ménage est indispensable pour vous éviter de dépasser votre quota d'espace disque. Recopiez ce répertoire `mjpeg` nettoyé dans votre répertoire TP2.

Modifiez la définition des modèles de tâches `tg` et `ramdac` pour introduire leurs implémentations matérielles. Comme ces deux implémentations sont définies dans `soclib`, la directive `import soclib` doit être présente avant la description des tâches dans votre fichier de description de tâche (`.task`).

- Pour la tâche `tg`, modifiez la déclaration de la tâche pour ajouter l'implémentation matérielle virtuelle (`SyntheticTask`):

```
from soclib import SyntheticTask

TaskModel(
    'tg',
    ports = {'output':MwmrOutput(32)},
    impls = [ SwTask( 'tg',
                                                              bootstrap = 'bootstrap',
                                                              stack_size = 4096,
                                                              sources = [ 'tg.c' ],
                                                              defines = [ 'FILE_NAME' ] ),
```

```
SyntheticTask( defines = {"FILE_NAME":"plan.jpg"} )
] )
```

- De même, pour la tâche `ramdac`.
- Recopiez dans le répertoire `mjpeg` la description DSX `vgnn_noirq_mono.py` que vous avez écrite dans la première partie, puis décommentez la partie de la description qui instancie les composants `tg` et `ramdac`.
- En vous inspirant de ce qui a été fait pour déployer `SplitMsg`, déployez l'application MJPEG sur la plateforme en complétant le fichier de description `mjpeg`.
  - ◆ Les coprocesseurs `tg` et `ramdac` sont spécifiques, ils doivent faire l'objet d'un déploiement non pas en tant que tâches logicielles sur un processeur, mais en tant que coprocesseurs rattachés à un contrôleur MWMR. Un déploiement valide pour `tg` est par exemple :

```
mapper.map('tg',
           coprocessor = 'tg0',
           controller = 'tg0_ctrl1'
           )
```

- Relancez la description, recompilez et lancez la simulation.

```
$ ./mjpeg
$ ./exe.muteks_hard
```

Pour avoir des statistiques, vous pouvez positionner la variable d'environnement `STATS_EVERY` à un nombre de cycles (500000 semble une bonne valeur):

```
$ STATS_EVERY=500000 ./exe.muteks_hard
```

L'avion fait le "tour" en 25 images. Vous avez un compteur de cycles sur le terminal qui contient le simulateur.



Q4 : *Combien faut-il de cycles, approximativement, pour décompresser 25 images ?*



Q5 : *Supposant un SoC cadencé à 200MHz, combien d'images sont affichées en une seconde ?*

## 4. Influence du système d'exploitation embarqué

L'environnement DSX permet actuellement d'utiliser deux systèmes d'exploitation embarqués.

- Mutek/S, un noyau "statique", ne fournissant pas la compatibilité POSIX. En particulier, il ne permet pas la création dynamique de tâches au moment de l'exécution.
- Mutek/H, un noyau fournissant aux applications l'API des threads POSIX (y compris la création dynamique de tâches), permettant d'héberger sur le même système des applications tierces.

Modifiez la description DSX de l'application MJPEG, pour utiliser l'OS Mutek/H. La section D devient:

```
mapper.generate( dsx.MutekS() )
mapper.generate( dsx.MutekH() )
```

- Relancez la description, recompilez et lancez la simulation du SoC avec Mutek/H (attention, il y a deux simulateurs avec des noms différents)

```
$ ./mjpeg
$ ./exe.mutekh_hard
```

**?** Q6 : *Combien de cycles faut-il pour décompresser 25 images avec Mutek/H ? Comment expliquer ce résultat ?*

**?** Q7 : *En consultant les headers de l'objet binaire généré, déterminez la capacité mémoire des deux bancs mémoire RAM0 et RAM1 suivant qu'on utilise Mutek/S ou Mutek/H.*

Pour analyser les headers d'un fichier binaire, servez vous des commandes suivantes :

```
$ mipsel-unknown-elf-objdump -h muteks/soft/bin.soft
$ mipsel-unknown-elf-objdump -h mutekh/soft/bin.soft
```

## 5. Compte-rendu

Vous rendrez une archive dans le même format que la semaine précédente, nommée `binome0_binome1.tar.gz`, contenant exactement les fichiers:

```
tp2/
tp2/rapport.pdf
tp2/vgmn_noirq_mono.py
tp2/splitmsg/
tp2/splitmsg/producer.c
tp2/splitmsg/producer.task
tp2/splitmsg/consumer.c
tp2/splitmsg/consumer.task
tp2/splitmsg/splitmsg.py
tp2/mjpeg/mjpeg.py
tp2/mjpeg/src/
tp2/mjpeg/src/iqzz/
tp2/mjpeg/src/iqzz/iqzz.c
tp2/mjpeg/src/iqzz/iqzz.task
tp2/mjpeg/src/libu/
tp2/mjpeg/src/libu/libu.c
tp2/mjpeg/src/libu/libu.task
```

- Les fichiers `splitmsg.py` et `mjpeg.py` seront complets, avec vos descriptions de TCG et le mapping. Pour `mjpeg`, il y aura les directives de génération de code pour Mutek/S et Mutek/H.
- Le répertoire `mjpeg/src` contiendra uniquement les implémentations de vos deux tâches `libu` et `iqzz` (éventuellement mises à jour par rapport à la semaine dernière) vous ayant servi à exécuter les tests de ce TP.
- Le rapport sera court (une table des matières pour dire que tout est sur la même page est superflue), répondant aux questions posées dans le texte, et nommé exactement `tp2/rapport.pdf`.

Vous enverrez cette archive avant le jeudi 16/12/2010, 18h00 (heure de Paris) à [MailAsim:joel.porquet Joël Porquet].

## Suite

TP Suivant : [MjpegCourse/Multipro](#)