# TP1 : Description de l'application MJPEG avec DSX et exécution sur station de travail



# 0. Objectif

L'objectif de ce premier TP est de vous familiariser avec le langage de description **DSX/L** (pour *Design Space Explorer/Language*). Ce langage permet à un concepteur de déployer une application logicielle multi-tâches (écrite en C) sur une architecture matérielle multiprocesseur (*MP-SoC*), modélisée avec les composants matériels fournis par la bibliothèque **SoCLib**.

Le langage de description DSX/L est une API (interface de programmation) implémentée à l'aide du langage Python, qui permet à un concepteur la réalisation des 3 tâches suivantes :

1. Définir la structure de l'application logicielle multi-tâches, c'est à dire le *Graphe des Tâches et des Communications* (aussi appelé *TCG*, pour *Tasks and Communications Graph*). Par cette structure, on suppose que le parallélisme "gros grain" de l'application ainsi que le schéma des communication entre les tâches peuvent être statiquement définis par le concepteur et n'évoluent pas en cours d'exécution.
2. Définir l'architecture matérielle, c'est-à-dire définir le nombre de processeurs, le nombre de bancs mémoires, la taille des caches processeurs, le type de réseau d'interconnexion utilisé, etc.
3. Contrôler le déploiement de l'application logicielle sur la plate-forme matérielle, c'est-à-dire le placement des tâches sur les processeurs et le placement des canaux de communication dans les bancs mémoire.

L'exécution de cette description DSX/L permet générer trois éléments :

1. Une version de l'application logicielle multi-tâches compatible POSIX, pouvant donc être compilée et exécutée sur n'importe quelle station de travail supportant l'API des threads POSIX. Cette première version permet de valider fonctionnellement l'application logicielle, indépendamment de toute architecture MP-SoC.
2. Un modèle SystemC complet de l'architecture matérielle du MP-SoC, configuré pour respecter l'organisation de l'espace adressable définie par le concepteur. Ce modèle SystemC est compilé pour générer un simulateur de cette architecture.
3. Le code binaire de l'application logicielle embarquée, obtenu en cross-compilant cette application pour le ou les processeur(s) présent(s) sur le MPSoC, et en effectuant l'édition de liens avec le système d'exploitation embarqué. Ce code binaire exécutable peut être directement chargé dans la mémoire du MP-SoC juste avant le lancement de la simulation.

Dans ce 1er TP, on se limitera à décrire - en langage DSX/L - la structure de l'application logicielle MJPEG, à écrire quelques unes des tâches de l'application MJPEG, et à valider cette application en l'exécutant sur une station de travail Linux.

Vous fournirez un rapport rédigé, au format PDF, ainsi que certains fichiers de code source. Tous les détails sont expliqués à la fin de cette page, dans la section #Compte-Rendu. Votre rapport devra contenir les réponses aux questions posées dans le sujet, visuellement signalées en ***gras oblique*** et préfixées de ?.

# 1. Prise en main

## 1.1. Exécuter l'application SplitMsg

Pour prendre en main l'outil DSX, on s'intéresse à une application parallèle extrêmement simple comportant deux tâches et un seul canal de communication MWMR. Cette application s'appelle SplitMsg.

- Importez l'environnement nécessaire dans le contexte de votre *shell*

```
$ source /users/outil/dsx/dsx_env.sh
```

- Créez un répertoire `SplitMsg` dans lequel vous mettrez les fichiers à recopier.
  - ♦ Pour les fichiers, voir la page SplitMsg
- Si ce n'est pas déjà fait, rendez la description DSX exécutable :

```
$ chmod +x fichier_de_description
```

- Exécutez la description DSX/L qui réalise la compilation l'application logicielle générée par DSX :

```
$ ./fichier_de_description
```

- ? Q1 : ***Quels fichiers étaient présents avant cette commande ? Quels fichiers ou répertoires ont été créés ?***

- Lancez le programme multi-tâches POSIX généré qui porte le nom `'exe.posix'` (vous pourrez interrompre l'exécution à tout moment en pressant `Ctrl-c`) :

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

- ? Q2 : ***Comment interpréter ce que vous observez lors de l'exécution de cette application ?***

## 1.2. Anatomie de la description DSX/L

DSX fait une distinction entre un *modèle de tâche* et une *instance de tâche* : un même modèle de tâche peut être en effet instancié plusieurs fois dans une application.

- Un modèle de tâche est défini par la directive `TaskModel`. Il spécifie pour une tâche donnée ses ressources utilisées (canaux de communication, etc.) ainsi que ses implémentations existantes. Pour l'instant, on ne s'intéressera qu'aux implémentations logicielles déclarées par `SwTask`. Voir DsxTaskModel.
- Une instance de tâche est définie par la directive `Task`. Elle fait partie d'un TCG. Elle est connectée aux autres tâches par les ressources. Voir DsxTasks.

Dans les applications décrites dans ce TP, chaque modèle de tâche ne sera utilisé qu'une fois.

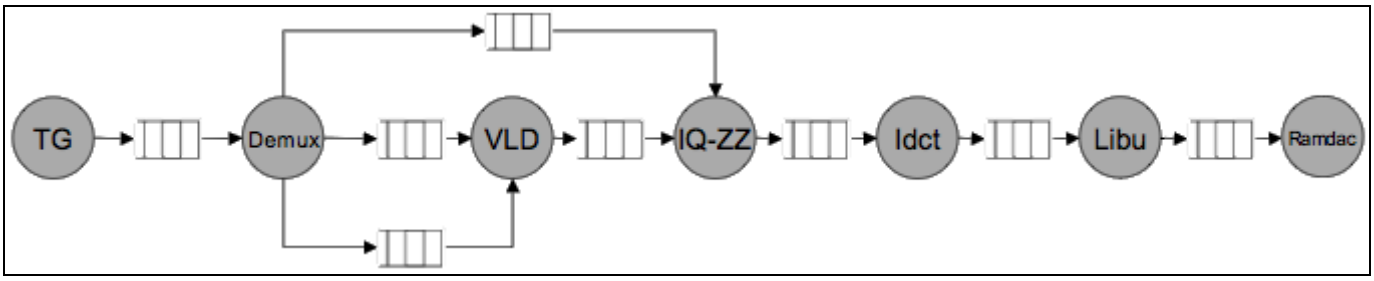
La description DSX/L de l'application SplitMsg comprends deux parties.

? Q3 : ***Rappelez à quoi sert chacune des 2 parties ?***

Q4 : ***Qu'est-ce qui n'est pas présent dans la description DSX/L, mais qui est quand même indispensable à la construction et à l'exécution de l'application ?***

# 2. Application MJPEG

Dans le reste du TP, on s'intéresse à l'application MJPEG telle que décrite en cours. On rappelle la structure du Graphe des Tâches et des Communications :




Chaque tâche effectue un traitement élémentaire dans la décompression d'une image. Dans ce TCG, on représente les tâches par des ronds et les canaux de communication par des rectangles : il s'agit donc d'un graphe bipartie.

Notre animation MJPEG sera composée d'une séquence d'images ayant toutes la même taille. Puisqu'une compression JPEG découpe l'image en blocs de 8x8 pixels, la hauteur et la largeur de chaque image sont multiples de 8.

Afin que l'application soit générique (c'est-à-dire supporte différentes tailles d'images), elle utilise les constantes suivantes :

- `WIDTH` : largeur de l'image en pixels
- `HEIGHT` : hauteur de l'image en pixels

À partir de ces deux constantes, d'autres constantes sont définies :

- `BLOCKS_W` : nombre de blocs en largeur
- `BLOCKS_H` : nombre de blocs en hauteur
- `NBLOCKS` : nombre de blocs par image (= BLOCKS_W*BLOCKS_H)

## 2.1. Spécifier le TCG

Téléchargez le fichier attachment:mjpeg_tp1.tar.bz2 chez vous et décompressez-le :

```
$ tar xjvf mjpeg_tp1.tar.bz2
```

La description DSX de l'application qui vous est fournie est incomplète, c'est à vous de remplir le code manquant.

Commencez par compléter les modèles de tâches incomplets, qui se trouvent dans les fichiers `'src/*/*.task'`, en spécifiant les largeurs des canaux de communications manquantes. Pour trouver ces largeurs, vous devez consulter le code des tâches. **Note**: même si le code source des tâches `'iqzz'` et `'libu'` n'est pas fourni, vous pouvez connaître les largeurs des canaux en vous référant au code des autres tâches.

Ensuite, vous devez compléter le fichier de description DSX `'mjpeg'` en plusieurs étapes :

Tout d'abord, notez qu'il faut nommer chaque élément du TCG :

- Les noms des tâches ont été définis en cours et sont donc imposés.
- Vous pouvez par contre choisir librement les noms des canaux de communication. Chaque canal de communication est attaché à au moins deux tâches par des *ports*. On distingue le nom des canaux de communication et le nom des ports des tâches connectées à ces canaux. Pour déterminer les noms des ports de chaque tâche, vous pouvez consulter le code des tâches ou bien le code des modèles de tâches.

Puis,

- Pour chaque canal de communication Mwmr :
  - Choisissez un nom et instanciez le canal. Il y a 7 canaux à définir, pour lesquels vous pouvez vous aider de l'API décrite dans DsxResource (`tg_demux` est fournie en exemple).
  - Dimensionnez (largeur et profondeur) les canaux en fonction des contraintes imposées par le code des tâches.
- Créez un Tcg :
  - Instanciez une tâche de chaque modèle. Voir DsxTcg.
  - Connecter les canaux aux ports des tâches, en les désignant par leurs noms.

## 2.2. Exécution de l'application

- Exécutez le fichier de description, qui réalise la compilation :

```
$ ./mjpeg
```

- Lancez l'exécution de l'application :

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

- ? Q5 : ***Décrivez brièvement ce que vous observez***

## 2.3. Écriture en C de la tâche IQZZ

IQZZ est une tâche faisant un double traitement, appliqué successivement à chaque bloc de 8x8 pixels de l'image.

IQZZ nécessite un tableau de quantisation inverse *T*, venant de la tâche `Demux` par un canal de communication dédié.

Cette table doit être lue **une fois par image**, et elle sert au traitement de **tous** les blocs d'une image. Le nombre de blocs dans l'image est donné par la constante `NBLOCKS`, définie dans le fichier `'jpeg.h'`.

Un bloc entrant dans IQZZ est composé de 8x8=64 facteurs.

$F_0$ $F_1$ $F_2$ $F_3$ $F_4$ $F_5$ $F_6$ $F_7$
$F_8$ $F_9$ $F_{10}$ $F_{11}$ $F_{12}$ $F_{13}$ $F_{14}$ $F_{15}$
$F_{16}$ $F_{17}$ $F_{18}$ $F_{19}$ $F_{20}$ $F_{21}$ $F_{22}$ $F_{23}$
$F_{24}$ $F_{25}$ $F_{26}$ $F_{27}$ $F_{28}$ $F_{29}$ $F_{30}$ $F_{31}$
$F_{32}$ $F_{33}$ $F_{34}$ $F_{35}$ $F_{36}$ $F_{37}$ $F_{38}$ $F_{39}$
$F_{40}$ $F_{41}$ $F_{42}$ $F_{43}$ $F_{44}$ $F_{45}$ $F_{46}$ $F_{47}$
$F_{48}$ $F_{49}$ $F_{50}$ $F_{51}$ $F_{52}$ $F_{53}$ $F_{54}$ $F_{55}$
$F_{56}$ $F_{57}$ $F_{58}$ $F_{59}$ $F_{60}$ $F_{61}$ $F_{62}$ $F_{63}$

On applique sur ce bloc deux traitement successifs :

- La quantisation inverse (IQ) est la multiplication de chaque élément d'entrée par un facteur de la table de 64 coefficients de quantisation inverse $T_n$, globale pour l'image.

$F_n' = F_n * T_n$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $F_0'$ | $F_1'$ | $F_2'$ | $F_3'$ | $F_4'$ | $F_5'$ | $F_6'$ | $F_7'$ |
| $F_8'$ | $F_9'$ | $F_{10}'$ | $F_{11}'$ | $F_{12}'$ | $F_{13}'$ | $F_{14}'$ | $F_{15}'$ |
| $F_{16}'$ | $F_{17}'$ | $F_{18}'$ | $F_{19}'$ | $F_{20}'$ | $F_{21}'$ | $F_{22}'$ | $F_{23}'$ |
| $F_{24}'$ | $F_{25}'$ | $F_{26}'$ | $F_{27}'$ | $F_{28}'$ | $F_{29}'$ | $F_{30}'$ | $F_{31}'$ |
| $F_{32}'$ | $F_{33}'$ | $F_{34}'$ | $F_{35}'$ | $F_{36}'$ | $F_{37}'$ | $F_{38}'$ | $F_{39}'$ |
| $F_{40}'$ | $F_{41}'$ | $F_{42}'$ | $F_{43}'$ | $F_{44}'$ | $F_{45}'$ | $F_{46}'$ | $F_{47}'$ |
| $F_{48}'$ | $F_{49}'$ | $F_{50}'$ | $F_{51}'$ | $F_{52}'$ | $F_{53}'$ | $F_{54}'$ | $F_{55}'$ |
| $F_{56}'$ | $F_{57}'$ | $F_{58}'$ | $F_{59}'$ | $F_{60}'$ | $F_{61}'$ | $F_{62}'$ | $F_{63}'$ |

- Le ZigZag (ZZ) est un réordonnancement des pixels d'un bloc en diagonale. Il permet d'améliorer la compression.

Après le réordonnancement, l'ordre des facteurs en sortie doit être:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $F_0'$ | $F_1'$ | $F_5'$ | $F_6'$ | $F_{14}'$ | $F_{15}'$ | $F_{27}'$ | $F_{28}'$ |
| $F_2'$ | $F_4'$ | $F_7'$ | $F_{13}'$ | $F_{16}'$ | $F_{26}'$ | $F_{29}'$ | $F_{42}'$ |
| $F_3'$ | $F_8'$ | $F_{12}'$ | $F_{17}'$ | $F_{25}'$ | $F_{30}'$ | $F_{41}'$ | $F_{43}'$ |
| $F_9'$ | $F_{11}'$ | $F_{18}'$ | $F_{24}'$ | $F_{31}'$ | $F_{40}'$ | $F_{44}'$ | $F_{53}'$ |
| $F_{10}'$ | $F_{19}'$ | $F_{23}'$ | $F_{32}'$ | $F_{39}'$ | $F_{45}'$ | $F_{52}'$ | $F_{54}'$ |
| $F_{20}'$ | $F_{22}'$ | $F_{33}'$ | $F_{38}'$ | $F_{46}'$ | $F_{51}'$ | $F_{55}'$ | $F_{60}'$ |
| $F_{21}'$ | $F_{34}'$ | $F_{37}'$ | $F_{47}'$ | $F_{50}'$ | $F_{56}'$ | $F_{59}'$ | $F_{61}'$ |
| $F_{35}'$ | $F_{36}'$ | $F_{48}'$ | $F_{49}'$ | $F_{57}'$ | $F_{58}'$ | $F_{62}'$ | $F_{63}'$ |

Notes d'implémentation:

- Pour implémenter ZZ, un tableau statique commençant par les valeurs ZZ[0]=0, ZZ[1]=1, ZZ[2]=8, ZZ[3]=16, ZZ[4]=9, etc. vous sera probablement utile.
- Les transformations IQ et ZZ doivent être implémentées dans la même boucle de code.
- Les types des données sont :
  - T : Table de quantisation inverse (IQ) : entiers non signés 8 bits
  - $F_n$ : Blocs en entrée : entiers 16 bits signés
  - $F_n'$ : Blocs en sortie : entiers 32 bits signés (car 8bits*16bits nécessite au plus 24 bits...)
- Votre code **doit** être portable quelle que soit l'endianness du processeur sous-jacent (si vous ne faites pas de transtypages hasardeux sur les pointeurs, ça devrait bien se passer).
- Votre code **doit** gérer toutes les tailles d'images (tant qu'elles sont multiples de 8x8). Cela signifie que toutes les boucles doivent utiliser les tailles issues des constantes (`WIDTH`, `HEIGHT`, `BLOCKS_W`, `BLOCKS_H`).

Instructions:

- Écrivez en C le code de la tâche IQZZ à l'aide de l'API logicielle définie dans SrlApi (inspirez vous également du code des autres tâches).
- Changez la définition du modèle de la tâche IQZZ dans la description DSX pour prendre en compte cette nouvelle implémentation :

```
# On avait dans iqzz.task:
sources = [ 'iqzz.bc' ],

# On peut alors déclarer iqzz comme une tâche logicielle en C.
sources = [ 'iqzz.c' ],
```

En fonction de la définition d'`iqzz` que vous utilisez (celle en `.bc` ou la vôtre en `.c`), et en recompilant, vous pouvez observer les différences entre l'implémentation de référence et la vôtre.

- Affinez votre fonction. Si besoin, lancez l'application `exe.posix` dans un débugger (`'gdb'`) : dans ce cas, notez que la fonction implémentant `iqzz` portera probablement le nom `iqzz_func_iqzz`.

## 2.4. Écriture en C de la tâche LIBU

Un *Ramdac* est une RAM couplée à un DAC (Digital to Analog Converter). Le contenu de la ram est converti en signal analogique pour être envoyé sur un écran. Le Ramdac que nous utilisons ici possède un accès particulier : il se comporte comme une FIFO. Il faut écrire les pixels dans l'ordre où ils vont être affichés : tous les pixels d'une ligne, puis toutes les lignes d'une image.

Il se trouve que les blocs issus de la décompression JPEG ont une taille de 8x8 pixels. Ils n'occupent donc pas la largeur de l'image, et il faut alors construire des lignes d'image à partir des blocs issus de la décompression. C'est le but de la tâche *Libu* (Line Builder).

Libu récupère `BLOCKS_W` blocs de 8x8 pixels et s'en sert pour construire 8 lignes de `WIDTH` pixels de large (rappel: BLOCKS_W*8 = WIDTH). Libu peut alors envoyer successivement ces lignes au Ramdac.

En pseudo-code, le traitement exécuté par Libu est :

```
bloc : 8x8 pixels
buffer : WIDTH*8 pixels

Pour chaque 0 .. BLOCKS_H:
    Pour chaque 0 .. BLOCKS_W:
        Lire un bloc
        Pour chaque ligne du bloc
            Copier les 8 pixels en les mettant à leur place dans buffer
    Pour chacune des 8 lignes du buffer:
        Envoyer la ligne
```

- Implémentez cette tâche en C à l'aide de l'API logicielle définie dans SrlApi.
- Modifiez la description de l'application DSX pour prendre en compte votre source.
- Testez l'application (après recompilation).

# 3. Compte-Rendu

Vous devrez créer une archive `.tar.gz`, contenant un seul répertoire nommé `tp1`. Dans ce répertoire vous devrez mettre (uniquement) :

- le fichier `mjpeg`, contenant la description de l'application.
- Les répertoires des tâches `iqzz` et `libu`, contenant uniquement les fichier `.task` et `.c` (pas le `.bc`). Bien entendu, ces deux descriptions de tâches `.task` doivent faire référence au `.c`, pas au `.bc`.
- Votre rapport (*une page maximum*) qui répond aux questions posées dans le sujet de TP, doit être au format PDF (et aucun autre) dans un fichier nommé **exactement** `rapport.pdf`.

Le nom de fichier de l'archive doit contenir les nom des deux auteurs, séparés par un *underscore* (_) (par exemple: `dupond_dupont.tar.gz`).

Faites particulièrement attention à cette archive : elle fera l'objet d'une correction automatique pour la validation des sources, d'où le format strict.

Pour être sûr de vous, l'exécution de la commande suivante sur votre archive doit vous rendre exactement la sortie correspondante :

```
$ tar tzf nombinome0_nombinome1.tar.gz
tp1/
tp1/src/mjpeg
tp1/src/iqzz/iqzz.c
tp1/src/iqzz/iqzz.task
tp1/src/libu/libu.c
tp1/src/libu/libu.task
tp1/rapport.pdf
$
```

Envoyez **impérativement** cette archive avant le mercredi 8/12/2010, 18h00 (heure de Paris) à [MailAsim:joel.porquet Joël Porquet]. Tout retard sera refusé, donc n'attendez pas la dernière minute.

# Suite

TP Suivant: MjpegCourse/Monopro