TP1: Description d'application avec DSX, et exécution sur station de travail

1. 0. Objectif
2. 1. Prise en main
    1. 1.1. Exécuter l'application SplitMsg
    2. 1.2. Anatomie de la description DSX
3. 2. Application MJPEG
    1. 2.1. Spécifier le TCG
    2. 2.2. Exécution de l'application
    3. 2.3. Écriture en C de la tâche IQZZ
    4. 2.4. Écriture en C de la tâche LIBU
4. 3. Compte-Rendu
5. Suite

0. Objectif

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

Le langage de description DSX est une API implémentée à l'aide du langage Python, et il permet au concepteur de faire 3 choses:

- Définir la structure de l'application logicielle multi-tâches, c'est à dire le Graphe des Tâches et des Communications. (aussi appelé TCG: Tasks&Communication Graph). On suppose que le parallélisme "gros grain" de l'application et le shéma des communication entre les tâches peuvent être statiquement définis par le concepteur, et n'évoluent pas en cours d'exécution.
- Définir l'architecture matérielle, c'est à dire définir le nombre de processeurs, le nombre de bancs mémoires, la taille des caches, le type d'inerconnect utilisé, etc...
- 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 sur les bancs mémoire.

L'exécution de cette description DSX permet générer trois choses:

- Une version de l'application logicielle multi-tâches compatible POSIX, qui peut 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.
- Un ensemble de fichiers de directives permettant de compiler l'application logicielle pour le(s) processeur(s) embarqué(s) sur le MP-SoC, d'effectuer l'édition de liens avec le système d'exploitation embarqué, et de générer le code binaire exécutable.
- Un modèle SystemC complet de l'architecture matérielle, correctement configuré pour respecter l'organisation de l'espace adressable défini par le concepteur, permettant de générer un simulateur complet de cette architecture, capable d'exécuter en simulation le code embarqué.

Dans ce 1er TP, on se limitera à décrire - en langage DSX - 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 GNU/Linux.

Vous fournirez un rapport rédigé, en format Adobe Acrobat (PDF), ainsi que certaines sources. Tous les détails sont à la fin dans la section 'Compte-Rendu'. Les points du présent sujet devant faire l'objet d'un écho dans votre rapport sont en ***gras oblique***, préfixés 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
```

- ? Q1: ***Quels fichiers ou répertoires ont été créés?***

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

```
$ ./fichier_de_description
```

- Lancez le programme multitâche généré qui porte le nom "exe.posix". Vous pourrez interrompre l'exécution à tout moment en pressant Ctrl-c.

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

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

1.2. Anatomie de la description DSX

Dans DSX, on fait une distinction entre un modèle de tâche et une instance de tâche, car un même modèle de tâche peut être instancié plusieurs fois dans une application.

- Un modèle de tâche est défini par la directive `TaskModel`. Il spécifie pour une tâche ses ressources utilisées (canaux de communication, ...) 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.

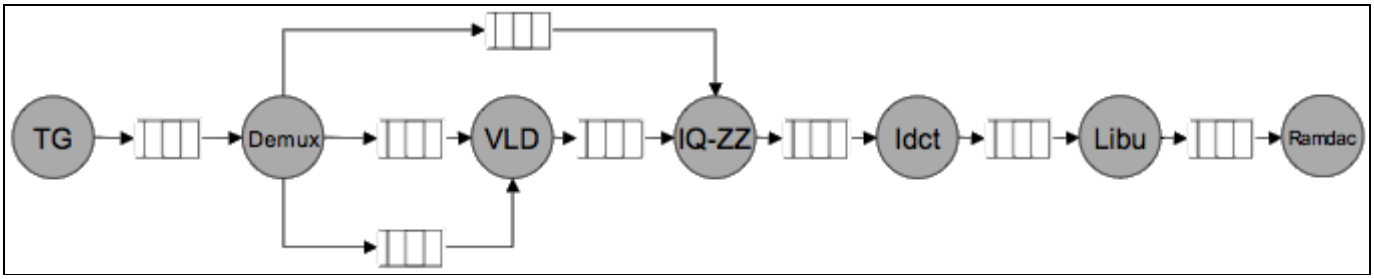
La description DSX de l'application SplitMsg est en deux parties.

? Q3: ***A quoi sert chacune des parties ?***

? Q4: ***Qu'est-ce qui n'est pas dans la description de l'application, mais qui est quand même indispensable à sa réalisation ?***

2. Application MJPEG

Dans tout le reste du TP, on s'intéressera à l'application MJpeg telle que décrite en cours. On en rappelle le graphe de tâches:



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'images faisant toutes la même taille. Comme une compression JPEG découpe l'image en blocs de 8x8 pixels, chacune des dimensions de l'image sera multiple de 8 (on ne gère qu'un nombre entier de blocs).

Nous allons utiliser les constantes suivantes:

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

A partir de ces deux constantes, d'autres constantes sont définies dans le fichier `jpeg.h`:

- 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

Il faut nommer chaque élément du TCG:

- Les noms des tâches ont été définis en cours et sont imposés.
- Vous pouvez 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 des tâches, il est impératif de consulter le code des tâches fourni dans le fichier attachment:mjpeg_tp1.tar.bz2, recopiez ce fichier chez vous et décompressez-le.

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

Note: Même si le code sources des tâches `iqzz` et `libu` n'est pas fourni, vous pouvez connaître les largeurs de tous les canaux en vous référant au code des autres tâches, et le nom des ports est fourni dans la description DSX, dans le fichier `mjpeg.py` (à compléter).

Dans le fichier de description DSX `mjpeg.py`,

- Pour chacun des modèles de tâches:

    - Reportez des noms pour chacun des ports d'entrée/sortie (cf DsxTasks).
    - Complétez la description des modèles de tâches (voir dans `src/*/*.task`) Iqzz, et Libu ont une déclaration particulière à ne pas prendre en compte pour l'instant, car ces modèles tâches sont fournis sans les sources: vous les écrirez aux prochaines questions
- Pour chaque canal de communication:
    - Choisissez un nom et instanciez le canal (Nous avons 7 canaux Mwmr, utiliser l'API décrite dans DsxResource, `tg_demux` est fournie en exemple)
    - Dimensionnez (profondeur et largeur) les canaux en fonction des contraintes imposées par le code des tâches.
- Créez un Tcg
    - en instanciant une tâche de chaque modèle, voir DsxTcg
    - en connectant les canaux aux ports des tâches, en les désignant par leurs noms

2.2. Exécution de l'application

- Exécutez la description, qui fait 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**, 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 "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, ... vous sera probablement utile.
- Les transformations IQ et ZZ doivent être implémentées dans la même boucle.
- 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). 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
- Réécrivez la définition de la tâche IQZZ dans la description DSX

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

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

Inspirez-vous des autres déclarations, n'oubliez pas les `defines` si vous voulez un code portable.

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

- Affinez votre fonction. Si besoin, lancez l'application `exe.posix` dans un débugger. 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. Notre Ramdac a un accès particulier: Il a un comportement 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 font 8x8 pixels. Ils ne font pas la largeur de l'image, il faut donc construire des lignes d'image à partir des blocs issus de la décompression. C'est le but de la tâche Libu (Line Builder).

Libu prend BLOCKS_W blocs de 8x8 pixels et en construit 8 lignes de WIDTH pixels de large (rappel: BLOCKS_W*8 = WIDTH). Il peut alors envoyer successivement ces lignes au Ramdac.

En pseudo-code, le traitement de 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 nouvellement compilée

3. Compte-Rendu

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

- Les répertoires des tâches iqzz et libu, contenant uniquement les fichier `.task` et `.c` (pas le `.bc`) Bien entendu, ces deux descriptions de tâches doivent faire référence au .c, pas au .bc !)
- Votre rapport (une page maximum) en format PDF (et aucun autre) dans `tp1/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 surs de vous, le listing du contenu de l'archive doit donner cette liste avec ces noms, et rien de plus (l'ordre des fichiers n'importe pas):

```
$ tar tzf nombinome0_nombinome1.tar.gz
tp1/
tp1/src/
```

```
tp1/src/iqzz/iqzz.c
tp1/src/iqzz/iqzz.task
tp1/src/libu/libu.c
tp1/src/libu/libu.task
tp1/rapport.pdf
$
```

Envoyez cette archive avant le 19/02/2007, 18h00 à [MailAsim:nipo Nicolas Pouillon].

Suite

TP Suivant: MjpegCourse/Monopro