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

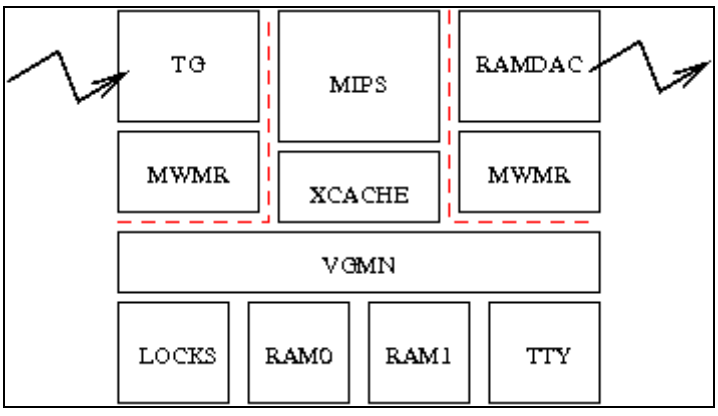
La première partie de ce TP vise à montrer comment décrire - avec DSX - une architecture matérielle de système intégré sur puce, en utilisant les composants matériels de la bibliothèque SoCLib. On rappelle que la bibliothèque SoCLib contient un ensemble modèles de simulation de composants matériels (IP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

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 de déployer l'application MJPEG du 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 sur 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 MIPS R3000. Cette architecture matérielle est appelée 'VgmnNoirqMono'.



- Elle est organisée autour d'un micro-réseau générique à interface VCI (composant VGMN). Ce composant est générique en ce sens qu'il accepte un nombre quelconque d'inititiateurs VCI, et un nombre quelconque de cibles VCI, ainsi qu'un paramètre définissant la latence du réseau: 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: les deux coprocesseurs matériels d'entrée/sortie TG et RAMDAC, ne doivent pas être décrits dans l'architecture VgmnNoirqMono. 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. Ils seront automatiquement générés dans la phase de "déploiement" de l'application MJPEG, lorsqu'on précisera que les tâches `tg` et `ramdac` du TCG sont implémentés comme des composants matériels.

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`. VgmnNoirqMono contient une descrition incomplète que vous devez compléter, en consultant 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 netlist SystemC décrivant la *top-cell*. Créez un fichier contenant:

```
#!/usr/bin/env python

import soclib
from vgmn_noirq_mono import VgmnNoirqMono

archi = VgmnNoirqMono()

archi.generate(soclib.PfDriver('hard'))
```

- Rendez ce nouveau fichier de description exécutable, lancez-le.

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

2. Déploiement de l'application SplitMsg

On va commencer par déployer l'application SplitMsg, qui ne comporte que deux tâches et un canal sur notre architecture de SoC monoprocesseur.

- Créez dans le répertoire TP2 un sous-répertoire 'splitmsg'
- Recopiez dans ce répertoire la description DSX de l'application SplitMsg du TP1.
- Recopiez dans ce répertoire la description DSX de vgmn_noirq_mono
- Modifiez cette description DSX 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 soclib import *

from vgmn_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 accédés à travers mapper.tcg['nom']. Il va falloir placer toutes les tâches, tous les canaux de communication, tous les objets logiciels associés aux processeurs et enfin les objets globaux du système.

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

mapper = Mapper(archi, tcg)

# mapping the MWMR channel

mapper.map( mapper.tcg["fifo"],
  buffer  = "cram1",
  status  = "cram1",
  desc    = "cram1")

# mapping the "prod0" and "cons0" tasks

mapper.map(mapper.tcg["prod0"],
   run = "processor",
   stack   = "cram0",
   desc    = "cram0",
   status  = "uram0")

mapper.map(mapper.tcg["cons0"],
   run = "processor",
   stack   = "cram0",
   desc    = "cram0",
   status  = "uram0")

# mapping the software objects associated to a processor

mapper.map( mapper.platform["processor"],
  desc    = "cram0",
  priv    = "cram0",
  status  = "uram0")

# mapping the software objects used by the embedded OS

mapper.map(mapper.tcg,
  desc    = "cram1",
  shared  = "uram1",
  code    = "cram1")
```

- La dernière étape consiste à générer le code

```
#########################################################
```

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

# Embedded software linked with the Mutek/S OS

mapper.generate( MutekS() )

# The software application for a POSX workstation can still be generated

tcg.generate( Posix() )
```

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

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

```
$ ./SplitMsg.py
```

- Executez 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
```

Q2: ***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 (en analysant 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 video 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. Nous allons donc modifier les déclarations des modèles de tâches en conséquence.

Retournez dans le répertoire mjpeg du TP1, et exécutez la commande `make mrproper` 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.

Dans la description DSX de l'application MJPEG, 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 `from soclib import *` doit être présente avant la description des tâches dans votre fichier de description.

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

```
from soclib.hwtask import HwTask, MwmrCoproc
```

```
hw_fifo_tg = MwmrCoproc(
        module = 'caba:fifo_reader',
        from_coproc = [ 'output:fifo' ],
        to_coproc = [],
        config = [],
        status = [],
        binary = 'bash',
        argv = [ 'bash', '-c', 'while cat %(FILE_NAME)s ; do true ; done' ],
        word_t = 'uint32_t' )

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

- De même, pour la tâche `ramdac`:

```
from soclib.hwtask import HwTask, MwmrCoproc

hw_fifo_ramdac = MwmrCoproc(
        module = 'caba:fifo_writer',
        from_coproc = [],
        to_coproc = [ 'input:fifo' ],
        config = [],
        status = [],
        binary = 'soclib-pipe2fb',
        argv = [ 'soclib-pipe2fb', '%(WIDTH)s', '%(HEIGHT)s' ],
        word_t = 'uint32_t' )

TaskModel(
        'ramdac',
        ports = {'input':MwmrInput(48*8)},
        impl = [ SwTask( 'ramdac',
                                        bootstrap = 'bootstrap',
                                        stack_size = 4096,
                                        sources = [ 'ramdac.c', 'ramdac_x11.c' ],
                                        defines = [ 'WIDTH', 'HEIGHT' ] ),
                        HwTask( hw_fifo_ramdac )
                        ] )
```

- En vous inspirant de ce qui a été fait pour déployer SplitMsg, déployez l'application MJPEG sur la plateforme.
    - 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 par un contrôleur MWMR à un interconnect VCI, et assignés à une adresse. Un déploiement valide pour `tg` est par exemple:

      ```
      mapper.map( 'tg0',
                  vci = 'vgmn0',
                  addr = 0x94000000 )
      ```

    - De même déployez `ramdac` sur la plateforme.
- Relancez la description, recompilez, lancez la simulation.

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

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

Q6: ***Combien faut-il de cycles, approximativement, pour décompresser 25 images ?***

Q7: ***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" , très performant, mais 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).

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

```
mapper.generate( 'muteks' )
mapper.generate( 'mutekh' )
tcg.generate('posix')
```

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

```
$ ./description
$ ./exe.mutekh_hard
```

Q8: ***Combien de cyles faut-il pour décompresser 25 images avec MUTEK/H? Comment expliquer ce résultat?***

Q9: ***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.***

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

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/D.
- Le répertoire `mjpeg/src` contiendra uniquement les implémentation 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, nommé `tp2/rapport.pdf`.

Vous livrerez cette archive avant mardi 26 fevrier 2008, 18h00 à [MailAsim:nipo Nicolas Pouillon].

Suite

TP Suivant: MjpegCourse/Multipro