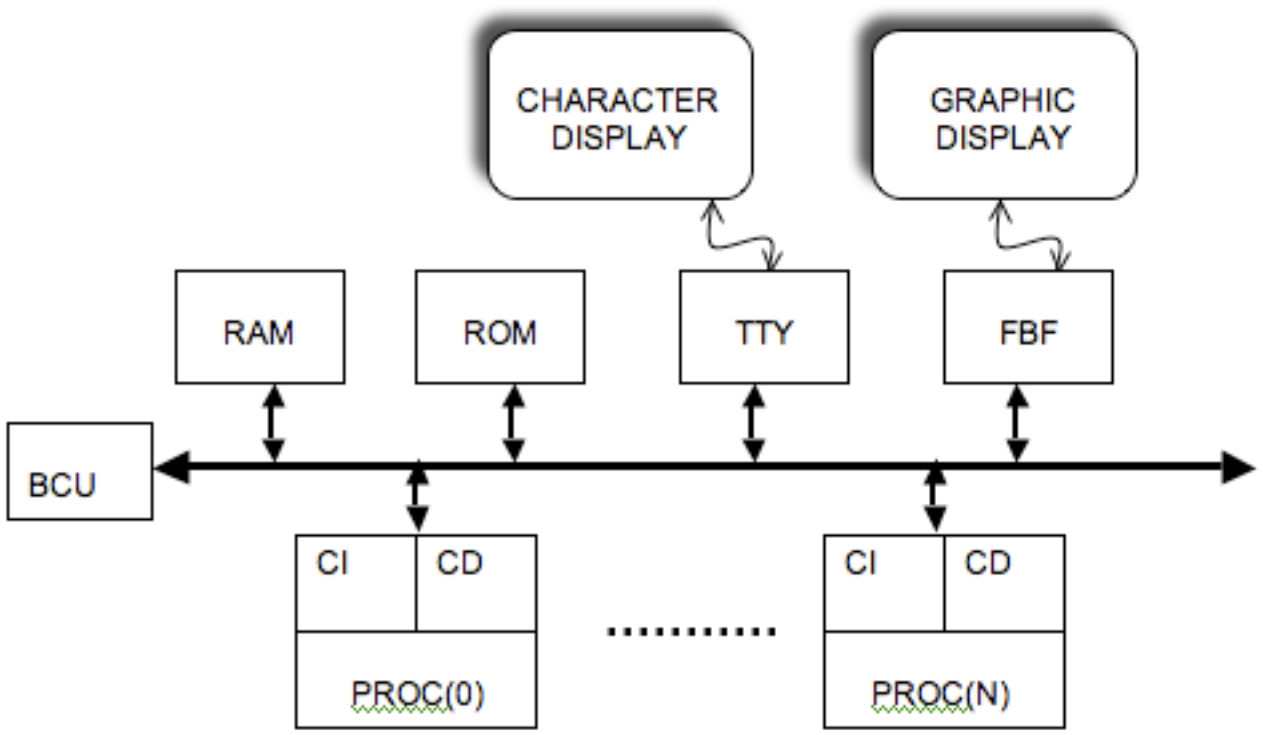
TP5 : Partage du bus dans les architectures multi-processeurs

Songlin Li : 3770906
Hongbo Jiang: 3602103

A) Objectifs

On s'intéresse dans ce TP aux problèmes de performances posés par le partage du bus entre les processeurs. Quand on augmente le nombre de processeurs connectés sur le bus, celui-ci devient un goulot d'étranglement, car la « bande passante » du bus (nombre maximum d'octets transférés par unité de temps) est bornée. Le temps d'attente pour accéder au bus peut devenir très long lorsque le nombre de processeurs devient grand : La conséquence est que chaque processeur passe plus de temps bloqué en attente du bus que de temps à exécuter des instructions.

B) Architecture matérielle



Question B1: En consultant l'en-tête du fichier **pibus_frame_buffer.h**, précisez la signification des arguments du constructeur du composant PibusFrameBuffer.
This component has 7 constructor parameters

- sc_module_name name : instance name
- unsigned int index : target index
- pibusSegmentTable segmap : segment table
- unsigned int latency : number of wait cycles
- unsigned int width : number of pixels per line frame width
- unsigned int height : number of lines frame height
- unsigned int subsampling : pixel format default = 420

Question B2: quelle est la longueur du segment associé à ce composant ?

#define SEG_FBF_BASE 0x96000000
#define SEG_FBF_SIZE FB_NPIXEL*FB_NLINE (256*256)
64 Koctets. Il y a 2 tampons de 64Koctets, l'un contient les valeurs de luminance; l'autre contient les valeurs de chrominance (pas utilisé pour image en niveau gris).

C) Compilation de l'application logicielle

Question C1 : Pourquoi faut-il utiliser un appel système pour accéder (en lecture comme en écriture) au contrôleur de frame-buffer?
Que se passe-t-il si un programme utilisateur essaie de lire ou d'écrire directement dans le Frame Buffer, sans passer par un appel système?

- le frame buffer appartient à un periphérique qui est non-cachable. Les périphiriques sont controlés par le kernal. Son adresse sont dans l'espace protégée.
  "Permission denied" pour accéder au periphérique sans "system call".

Question C2 : Pourquoi préfère-t-on construire une ligne complête dans un tableau intermédiaire de 256 pixels plutôt que d'écrire directement l'image - pixel par pixel - dans le frame buffer?

- Pour écrire au périphérique, il faut faire l'appel système: fb_sync_write(), cela prend aussi beaucoup de temps. Ecrire ligne par ligne permet d'économiser le temps d'appel système.

Question C3 : Consultez le fichier **stdio.c**, et expliquez la signification des trois arguments de l'appel système *fb_sync_write()*. Complétez le fichier **main.c** en conséquence.

fb_sync_write()
This blocking function use a memory copy strategy to transfer data
from a user buffer to the frame buffer device in kernel space,

- offset : offset (in bytes) in the frame buffer
- buffer : base address of the memory buffer
- length : number of bytes to be transfered
- It returns 0 when the transfer is completed.

```
int fb_sync_write(size_t offset, void* buffer, size_t length)
{
   return sys_call(SYSCALL_FB_SYNC_WRITE,
           offset,
           (int)buffer,
           length,
           0);
}
```

```
if ( fb_sync_write( line*256 ,buf, 256  ) )
```

Vérifiez que les adresses de base des différents segments définies dans le fichier **seg_ld** sont identiques à celles définies dans le fichier**tp5_top.cpp** décrivant l'architecture matérielle, et lancez la compilation du logiciel dans le répertoire **soft**, en utilisant le makefile qui vous est fourni.

D) Caractérisation de l'application logicielle

Architecture Logicielle et Matérielle
L'architecture proposée, et le protocole du PIBUS définissent 4 types de transactions sur le bus :

1. IMISS : Lecture d'une ligne de cache suite à un MISS sur lecture instruction
2. DMISS : Lecture d'une ligne de cache suite à un MISS sur lecture de donnée
3. UNC : Lecture d'un mot appartenant à un segment non-cachable.

4. WRITE : Ecriture d'un mot (vers la RAM, le contrôleur TTY ou le Frame Buffer)

On va commencer par caractériser l'application logicielle en l'exécutant sur une architecture monoprocesseur.

Pour cela, on va relever, lorsque l'application se termine. les informations accumulées dans différents compteurs d'instrumentation.

Commencez par lancer la simulation sans limitation du nombre de cycles pour déterminer combien de cycles sont nécessaires pour afficher l'image. On considèrera que l'application est terminée quand on exécute l'appel système exit().

Relancez la simulation, en imposant le nombre de cycles à simuler, et en activant l'affichage des statistiques avec la commande (-STATS) sur la ligne de commande.

Question D1 : Quel est le nombre de cycles nécessaires pour afficher l'image avec un seul processeur. Combien d'instructions ont été exécutées pour afficher l'image? Quel est le nombre moyen de cycles par instruction (CPI) ?

- cycles = 5697953 Instructions = 4010542 CPI = 1.42074

*** proc[0] at cycle 5458818

- INSTRUCTIONS = 4365774
- CPI = 1.25037
- CACHED READ RATE = 0.264193
- UNCACHED READ RATE = 0.00219136
- WRITE RATE = 0.139718
- IMISS RATE = 1.99277e-05
- DMISS RATE = 2.25419e-05
- IMISS COST = 16.0575
- DMISS COST = 15.0769
- UNC COST = 6
- WRITE COST = 0

  bcu : Statistics

  master 0 : n_req = 619655 , n_wait_cycles = 619655 , access time = 1

  master 1 : n_req = 0 , n_wait_cycles = 0 , access time = -nan

  master 2 : n_req = 0 , n_wait_cycles = 0 , access time = -nan

cycle 5012203

*** proc[0] at cycle 5012203

- INSTRUCTIONS = 4046331
- CPI = 1.2387
- CACHED READ RATE = 0.261666
- UNCACHED READ RATE = 0.00130143
- WRITE RATE = 0.141599
- IMISS RATE = 2.00181e-05
- DMISS RATE = 2.45564e-05
- IMISS COST = 16
- DMISS COST = 15.4231
- UNC COST = 6
- WRITE COST = 0

  bcu : Statistics

  master 0 : n_req = 578329 , n_wait_cycles = 578329 , access time = 1

  master 1 : n_req = 0 , n_wait_cycles = 0 , access time = -nan

  master 2 : n_req = 0 , n_wait_cycles = 0 , access time = -nan

Question D2 : Quel est le pourcentage d'écritures sur l'ensemble des instructions exécutées ?

- WRITE RATE = 0.142863

Quel est le pourcentage de lectures de données ?

- **CACHED READ RATE + UNCACHED READ RATE**
- CACHED READ RATE = 0.266426
- UNCACHED READ RATE = 0.00131304

Quels sont les taux de miss sur le cache instruction et sur le cache de données ?

- IMISS RATE = 0.00894243
- DMISS RATE = 0.00912391

Quel est le coüt d'un miss sur le cache instructions ?

- IMISS COST = 16.0686

Quel est le coût d'un miss sur le cache de données ?

- DMISS COST = 14.3438

Quel est le coût d'une écriture pour le processeur ?

- WRITE COST = 0

On rappelle que les couts se mesurent en nombre moyen de cycles de gel du processeur. Comment expliquez-vous que ces coûts ont des valeurs non entières?

- valeur moyenne non-fixed. Ces coûts ont des valeurs non entières parce que pour chaque Miss, le Nb de cycle de gel n'est pas constant. Peut-être un autre Mâitre possède le bus, ce cas ne se passe pas dans cette question; l'autre cas c'est qu'il y a peut-être une écriture qui occupe le bus, l'écriture possède toujour en proprité le bus.
  IMISS COST = c_imiss_frz / c_imiss_count
  DMISS COST = c_dmiss_frz / c_dmiss_count

```
run\_cycles = c\_total\_cycles - c\_frz_cycles
INSTRUCTIONS = run_cycles
CPI =c\_total\_cycles / run_cycles
CACHED READ RATE = (c\_dread\_count – c\_dunc\_count) / run_cycles
UNCACHED READ RATE =c\_dunc\_count / run_cycles
WRITE RATE = c\_write\_count / run_cycles
IMISS RATE = c\_imiss\_count / run_cycles
DMISS RATE = c\_dmiss\_count / (c\_dread\_count - c\_dunc\_count)
IMISS COST = c\_imiss\_frz / c\_imiss\_count
DMISS COST = c\_dmiss\_frz / c\_dmiss\_count
UNC COST =c\_dunc\_frz / c\_dunc\_count
WRITE COST = c\_write\_frz / c\_write\_count
```

Question D3 : Evaluez le nombre de transactions de chaque type pour cette application. Que remarquez-vous ?

- WRITE: c_total_inst * WRITE RATE = 4010542 * 0.142863 = 572958
- IMISS: c_total_inst * IMISS RATE = 4010542 * 0.0089 = 35694
- DMISS: (dread_count-dunc_count) * DMISS RATE = 4010542 * 0.266 * 0.009 = 9601
- UNC: c_total_inst * UNCACHED READ RATE = 4010542 * 0.0013 = 5214

Le pourcentage de transaction WRITE est le plus grand.

E) Exécution sur architecture multi-processeurs

Le numéro de processeur est une valeur *cablée* au moment de la fabrication dans le registre protégé $15. Cette valeur n'est donc pas modifiable par le logiciel, mais peut être lue, lorsque le processeur est en mode KERNEL, en utilisant l'instruction *mfc0*. En assembleur l'instruction suivante copie le registre protégé S15 dans le registre non protégé $27:
***mfc0* (move from coprocessor 0)**
***mtc0* (move to coprocessor 0)**

```
    mfc0  $27,  $15,  1
    addi  $27,  $27,  27 /* $27 <= proc_id */
```

Dans un programme utilisateur en C, cette information peut être obtenue en utilisant l'appel système *procid()*.

Question E1 : Modifiez la boucle principale dans le fichier **main.c** pour partager le travail entre les différents processeurs de l'architecture.

```
for(line = 0 ; line < NLINE ; line++) {
        if (( line  % nprocs  == n)){
                for(pixel = 0 ; pixel < NPIXEL ; pixel++){
                        buf[pixel] = build(pixel, line, 5);
                }
                if ( fb_sync_write( line*256 ,buf, 256  ) )//int
fb_sync_write(size_t offset, void* buffer, size_t length);
                        tty_printf(" !!! wrong transfer to frame buffer
for line %d\n", line);
                else
                        tty_printf(" - building line %d at cycle %d\n",
line, proctime());
        }
    }
```

Il faut également modifier le code de boot, car on a maintenant plusieurs tâches qui s'exécutent en parallèle, et chaque tâche doit disposer de sa propre pile d'exécution. Ici aussi, tous les processeurs exécutent le même code de boot, mais l'exécution dépend du numéro de processeur.

Question E2 : Pourquoi les piles d'exécution des N programmes s'exécutant sur les N processeurs doivent-elles être strictement disjointes?

- Parce que chaque processeur s'éxecute les intruction indépendendamment. Si ils partagent un pile, les nouvelles données vont effacer les anciennes.
- initializes stack pointer SP <= stack_base + nb_proc*size + size

```
# initializes stack pointer     SP <= stack_base + nb_proc*size + size
        mfc0    $27,    $15,    1
        andi    $27,    $27,    7    # 8 processors maximium
        la      $29,    seg_stack_base
        sll     $27,    $27,    16   # $27 <= proc_id * 64K
        add     $29,    $29,   0x10000
        add     $29,    $29,   $27

# version 2 la même chose:)
# initializes stack pointer
        la      $29,    seg_stack_base
        addiu   $29,    $29,    0x4000          # stack size = 16 Kbytes
#64K marche pas
        ori             $7,             $0,             0x4000
        mfc0    $27,    $15,    1
        andi    $27,    $27,    0x7
        multu   $27,    $7
        mflo    $5
        addu    $29,    $29,    $5
```

Question E3 : Donnez deux raisons pour lesquelles le code binaire doit être recompilé chaque fois qu'on change le nombre de processeurs.

- raison 1: le programme va distribuer à nouveau les tâches pour chaque processeur.
- raison 2: l'adresse du haut du pile doit être changée.

Question E4 Remplissez le tableau ci-dessous, et représentez graphiquement speedup(N) en fonction de N.

	1 proc	2 proc	4 proc	6 proc	8 proc
cycles	6241680	3183584	1869993	1796799	1793194

speedup	1	1.96	3.34	3.47	3.48

Comment expliquez-vous que le speedup ne soit pas linéaire?

- Le speedup n'est pas linéaire parce que: le temps d'accéder au bus pour chaque processeur est prolongé parce qu'il faut attendre les autres processeurs.

F) Evaluation des temps d'accès au bus

pibus_seg_bcu.cpp :

- La méthode *printStatistics()* associée au processeur permet de mesurer le coût moyen des différents types de transaction. Ce coût est obtenu en divisant le nombre total de cycles de gel du processeur par le nombre d'événements de chaque type.
- La méthode *printStatistics()* associée au BCU permet de mesurer, pour chaque maître utilisant le bus, le temps moyen d'accès au bus, défini comme le nombre de cycles d'attente entre l'activation du signal REQ[i] (demande d'accès au bus par le maître (i)) et l'activation du signal GNT[i] (allocation du bus au maître (i)). On divise pour cela le nombre total de cycles d'attente du maître (i) par le nombre total de requêtes effectuées par le maître (i).

question F1 Pourquoi faut-il absolument effectuer la mesure au moment précis où l'application se termine?

- Parce que le temps moyen d'accès au bus et le temps coût moyen de transaction ne sont pas stables, ils changent au mesure du temps.

Question F2 Remplissez le tableau ci-dessous, en exécutant successivement le programme sur différentes architectures:

NPROCS	1	2	4	6	8
IMISS COST	15.9	18.1	31.0	47.6	65.7
DMISS COST	14.5 17.3	26.3	41.4	53.4	
WRITE COST	0	0	0.32653	3.70303	7.7488

ACCESS_TIME	1	1.5	4.8	11.7	17.0
CPI	1.515	1.543	1.809	2.584	3.457

Il faut tenir compte de la question précédente, car la durée d'exécution de l'application dépend du nombre de processeurs.

Question F3 : Interprétez ces résultats. La dégradation de la valeur du CPI quand on augmente le nombre de processeurs est-elle principalement dûe à l'augmentation du coût des MIS, ou à l'augmentation du coût des écritures?

- Quand le Nb de processeur est très petit, 1 ou 2, le CPI est grand et il depend de la vitesse de processeur;
- après le Nb de processeur augmente, le CPI diminue; mais quand le Nb de processeur est trop grand, le CPI augmente principalement dû à l'augmentation du coût des MISS et des écritures.

G) Modélisation du comportement du bus

Question G1 : En exploitant votre compréhension du protocole PIBUS, calculez le nombre de cycles d'occupation du bus pour chacun des 4 types de transaction. Attention : il ne faut pas compter le temps d'attente pour accéder au bus, car la phase REQ->GNT n'utilise pas le bus et a lieu pendant la fin de la transaction précédente. Par contre, il faut compter un cycle supplémentaire correspondant au "cycle mort" entre deux transactions.
TEMPS_IMISS : IMISS COST - ACCESS_TIME + 1 = 15.9
TEMPS_DMISS : DMISS COST - ACCESS_TIME + 1 = 14.5
TEMPS_UNC : UNC COST - ACCESS_TIME + 1 = 6
TEMPS_WRITE : WRITE COST + 1 = 1
Question G2 : En exploitant les résultats de la partie D (pourcentage de lectures cachées, pourcentage d' écritures, taux de MISS sur les caches, valeur du CPI), calculez la fréquence de chacun des 4 types de transaction et complétez le tableau ci-dessous. Puisque notre unité de temps est le cycle, ces fréquences seront exprimées en *nombre d'événements par cycle*.

	Temps_Occupation	**Fréquence**
IMISS	573553	0.0062

DMISS	137715	0.0017
UNC	31284	0.0009
WRITE	0	0.1007

On rappelle qur pour n'importe quel type d'événement, la fréquence peut être calculée comme:

> *(nombre d'événement / cycle) = (nombre d'événement / instruction) * (nombre d'instruction / cycle)*

Question G3 : En utilisant les résultats des deux questions précédentes, calculez le pourcentage de la bande passante du bus utilisé par un seul processeur. En déduire le nombre de processeurs au delà duquel le bus commence à saturer.

- la « bande passante » du bus (nombre maximum d'octets transférés par unité de temps) est bornée.
- le pourcentage de la bande passante du bus P

> *P = TEMPS_IMISS * F(IMISS) + TEMPS_DMISS * F(DMISS) + TEMPS_UNC * F(UNC) + TEMPS_WRITE * F(IMISS) = 0.23*

Nb de proc maximum : 1 / 0.23 = 4
Par conséquence, au delà de 4 processeurs, le bus commence à saturer.

LL/SC excercise

```
void automic_increment( int * ptr , int inc ){
        while (*ptr != (ptr + inc) ) {ptr++;}
        return;
}
```

Ecrit en MIPS32 cette fonction ne returne pas quand elle a réussi à incrémenter automatiquement la case mémoire pointer par ptr (*ptr = *ptr + inc)