M1 SESI 2017-2018
Architecture Multi-Processeurs
TP6 : Contrôleur DMA

Kevin Mambu

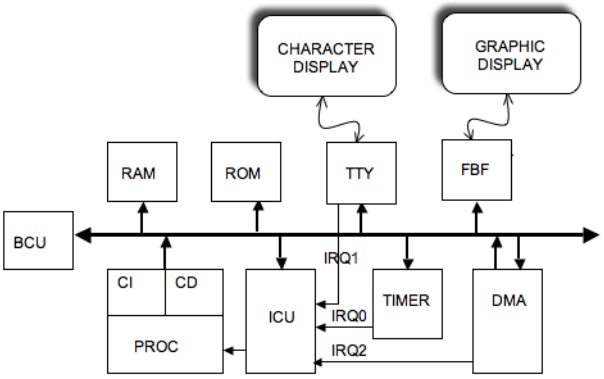
April 6, 2018

A) Objectifs

Le but de ce TP est d'analyser le fonctionnement d'un périphérique plus complexe que ceux analysés dans le TP6. Un périphérique possédant une capacité DMA (Direct Memory Access) se comporte à la fois comme un maître capable de lire ou d'écrire directement en mémoire, et comme une cible capable - comme n'importe quel périphérique - de recevoir des commandes provenant du système d'exploitation.

On utilise la même architecture que dans le TP5 et le TP6, mais on instanciera un seul processeur, et on activera le contrôleur DMA. On utilisera des caches possédant une capacité de 2 Koctets (lignes de 16 octets, 4 niveaux d'associativité, 32 sets).

B) Contrôleur DMA



Question B1

Registres adressables du contrôleur DMA		
`SOURCE`	0x00 (RW)	Adresse de base du tampon source
`DEST`	0x04 (RW)	Adresse de base du tampon destinataire
`NWORDS/STATUS`	0x08 (RW)	• si écriture → démarre un transfert DMA pour le nombre de mots spécifié. • si lecture → contient le statut de retour du transfert DMA : – DMA_IDLE – DMA_SUCCESS – DMA_READ_ERROR – DMA_WRITE_ERROR
`RESET`	0x0C (R)	Reset logiciel, écrire dans ce registre pour acquitter l'interruption du contrôleur DMA.
`IRQ_DISABLED`	0x10 (RW)	IRQ désactivée si $\neq 0$

- L'adresse de base du segment associé au contrôleur DMA en tant que cible doit être alignée sur une frontière de bloc de 32 octets par souci d'alignement (segment de 32 octets).

Question B2

`burst` est l'argument permettant de spécifier le nombre maximal de mots qui peuvent être transferés lors d'une rafale (*Burst*).

Question B3

Le coprocesseur DMA requiert deux automates MASTER_FSM et TARGET_FSM afin de pouvoir :

- Recevoir des commandes d'autres maîtres (en qualité de cible).
- Initier des transactions mémoires (en qualité de maître).

Ces deux tâches doivent pouvoir se faire simultanément.

Question B4

La bascule `r_stop` est une bascule commandée par l'automate TARGET_FSM. Elle est en popsition basse lors d'un transfert DMA (e.g. lorsque MASTER_FSM est à l'état DMA_IDLE) et est positionnée à true à la fin d'un transfert DMA par le bais du registre RESET.

Question B5

Transitions de MASTER_FSM	
A'	$\overline{R\_STOP}$
A	R_STOP
B'	$\overline{GNT}$
B	GNT
C	$true$
D'	$WAIT$
D	$\overline{WAIT}$
E'[1]	$\overline{ERROR} \bullet READY \bullet R\_STOP$
E	$\overline{ERROR} \bullet READY \bullet \overline{R\_STOP}$
F	$ERROR$
G	$\overline{READY} \bullet ERROR$
H'	$\overline{READ}$
H	$READ$
I	$true$
J'	$\overline{READY}$
J	$READY$
K'[1]	$\overline{ERROR} \bullet READY \bullet R\_STOP$
K	$\overline{ERROR} \bullet READY \bullet \overline{R\_STOP} \bullet R\_COUNT$
L	$ERROR$
M	$\overline{ERROR} \bullet READY \bullet \overline{R\_STOP} \bullet \overline{R\_COUNT}$
N	$\overline{READY} \bullet \overline{ERROR}$
O', P', Q'	$\overline{R\_STOP}$
O, P, Q	R_STOP

[1] : Ces transitions n'apparaissent pas sur le graphe du sujet, mais servent de point de retour en cas d'erreur : si le maître responsable de la transaction se fait tué avant la fin de cette dernière, le contrôleur DMA avorte alors son transfert puis retourne dans l'état IDLE. **Uniquement à la fin d'un transfert sur le PIBUS (XXX_DT), sinon ce dernier se retrouverait bloqué!**

C) Architecture matérielle

Question C1

- Par défault, la longeur d'une rafale de mots 32 bits est de 16 mots.
- Utiliser de grosses rafales permet de réduire le nombre de réquisitions du Pibus, plusieurs mots tranferés pour une réquisition du bus.
- Dans la description *pibus_dma.cpp*, on peut voir que le `burst` est borné entre 0 et 128. Augmenter la longeur maximale d'une rafale augmenterait la taille des tampons internes du périphérique DMA.

Question C2

- L'adresse de base du segment associé au périphérique DMA est `0x93000000`.
- L'index de cible vis-à-vis du BCU est `DMA_INDEX`, égal à 6.

- Son numéro de maître est à `nprocs + 1`. Sur notre, architecture mono-processeur, son index de maître est alors à 2.
- La ligne d'interruption IRQ contrôlée par le périphérique DMA est relié au port d'entrée IRQ_IN[0] du composant ICU.

D) Application logicielle

Question D1

- Lors de la fonction fb_sync_write, le processeur exécutant cette fonction est à la charge de l'écriture du tampon dans le segment réservé au FrameBuffer, par l'intermédiaire du appel système.
- Cet appel est alors naturellement bloquant car, rappelons le, parmi les requêtes d'accès mémoire, les écritures en Tampon d'Écriture Postées sont les plus prioritaires et la donnée à écrire l'est dans un segment non-cachable.

Question D2

	Damier 5	Damier 4	Damier 3	Damier 2	Damier 1	Moyenne
Display (cy)	117148	117161	117161	117175	117353	117200
Build (cy)	717883	713813	717827	717822	N.C.	716837

Question D3

La fonction fb_sync_write() force une attente active jusqu'à la fin de l'écriture vers le FrameBuffer tandis que fb_write() déclenche l'écriture de manière non-bloquante, via le coprocesseur DMA. La vérification de la fin du transfert se fait grâce à la fonction fb_completed().

Question D4

	Damier 5	Damier 4	Damier 3	Damier 2	Damier 1	Moyenne
Display (cy)	4199	4199	4199	4260	4036	4179

Question D5

Sur le bord gauche de l'image affichée, on peut observer une portion de cases appartenant au damier de l'itération suivante. Cela est dû au fait que faute de synchronisation sur la bonne écriture du buffer, les premières cases du buffer ont le temps d'être écrasées avant d'être envoyées au FrameBuffer.

Question D6

-
  - Dans _fb_completed(), tant que la variable _dma_busy est différente de 0, le processeur est bloqué.
  - Dans _fb_write(), tant que _dma_busy est différente de 0, le processeur attend une période pseudo-aléatoire, une fois que _dma_busy est égale à 0, le processeur la remet à 1.
- La routine d'interruption _isr_dma est la fonction qui remet à 0 le flag _dma_busy.
- Le vecteur de flags `_dma_busy` est stocké dans le segment seg_unckdata.

E) Pipe-line logiciel

Question E1

Pour pouvoir passer de la période (n) à la période (n+1), il faut que le transfert du DMA au Framebuffer soit terminé à la période (n). Nous utiliserons la fonction _fb_completed.

Question E2

	Damier 5	Damier 4	Damier 3	Damier 2	Damier 1	Moyenne
Display (cy)	3979	4086	3984	4240	3690	3996
Build (cy)	716739	716789	716901	716871	N.C.	716825

- Le gain apporté par le parallelisme pipeline par rapport à une exécution séquentielle est de 183 cycles, ce qui est bien maîgre gain comparé au coût de l'implémentation (ajout d'un if, ajout d'un tampon).

F) Traitement des erreurs

Question F1

fb_read() et fb_write() sont des fonctions qui font des accès mémoire, en ce sens il ne faut pas que le tampon source ou destination soit de la zone mémoire privilégiée :

- fb_read() a la possibilité de permettre de l'injection de données dans le système d'exploitation.
- fb_write() a la possibilité de dévoiler des portions de l'espace système auquel l'utilisateur ne devrait pas avoir accès.

Une fois la requête envoyée au contrôleur DMA, plus aucun contrôle de droits ou de sécurité n'est effectué. Cette vérification doit être fait a priori.

Question F2

- À l'échelle du composant Pibus_DMA, en cas d'erreur le composant passe dans l'état DMA_XXXXX_ERROR, puis lève une interruption.
- À l'échelle de la routine _isr_dma, la routine sauvegarde dans le registre _dma_status la cause de la levée d'interruption.
- À l'échelle de _fb_completed(), si le _dma_status est différent de 0, la fonction retourne 1 (cas d'erreur).

Question F3

- fb_sync_write
  - sys_call(FB_SYNC_WRITE...
  - syscall
  - _fb_write(...
  - C'est dans la fonction de driver que la vérification des arguments se fait :

```
1  if (((unsigned int) buffer >= 0x80000000)
     || (((unsigned int)buffer + length) >= 0x80000000) )
3    return 1;
```

```
1
#include "stdio.h"
3
#define NPIXEL 128
5#define NLINE  128

7////////////////////////////////////////////////////////////////////////////
//  main function
9////////////////////////////////////////////////////////////////////////////
__attribute__ ((constructor)) void main()
11{
  unsigned char   BUF1[NPIXEL*NLINE];
13  unsigned char   BUF2[NPIXEL*NLINE];
  unsigned int  line;
15  unsigned int  pixel;
  unsigned int  step;
17
  /***** Prologue *****/
19  tty_printf("\n*** damier 1 ***\n\n");
  for(pixel = 0 ; pixel < NPIXEL ; pixel++)
21  {
    for(line = 0 ; line < NLINE ; line++)
23    {
      if( ( (pixel>>step & 0x1) && !(line>>step & 0x1)) || (!(pixel>>step & 0x1) &&  (line>>step & 0x1)) )
25      BUF2[NPIXEL*line + pixel] = 0xFF;
      else
27      BUF2[NPIXEL*line + pixel] = 0x0;
    }
29  }
  tty_printf(" - build   OK at cycle %d\n", proctime());
31
  /***** Software pipelining *****/
33  for(step = 2 ; step < 6 ; step++)
  {
35    if(((step%2) == 0) && (step < 5)) {
      if(fb_write(0, BUF2, NLINE*NPIXEL) != 0)
37      {
        tty_printf("\n!!! error in fb_syn_write syscall !!!\n");
39      exit();
      }
41      tty_printf(" - display OK at cycle %d\n", proctime());
      tty_printf("\n*** damier %d ***\n\n",step);
43      for(pixel = 0 ; pixel < NPIXEL ; pixel++)
      {
45      for(line = 0 ; line < NLINE ; line++)
        {
47        if( ( (pixel>>step & 0x1) && !(line>>step & 0x1)) || (!(pixel>>step & 0x1) &&  (line>>step & 0x1
    )) )
            BUF1[NPIXEL*line + pixel] = 0xFF;
49        else
            BUF1[NPIXEL*line + pixel] = 0x0;
51      }
      }
53      tty_printf(" - build   OK at cycle %d\n", proctime());
    }
55    else {
      if(fb_write(0, BUF1, NLINE*NPIXEL) != 0)
57      {
        tty_printf("\n!!! error in fb_syn_write syscall !!!\n");
59      exit();
      }
61      tty_printf(" - display OK at cycle %d\n", proctime());
      tty_printf("\n*** damier %d ***\n\n",step);
63
      for(pixel = 0 ; pixel < NPIXEL ; pixel++)
65      {
        for(line = 0 ; line < NLINE ; line++)
67        {
          if( ( (pixel>>step & 0x1) && !(line>>step & 0x1)) || (!(pixel>>step & 0x1) &&  (line>>step & 0x1
    )) )
69          BUF2[NPIXEL*line + pixel] = 0xFF;
          else
71          BUF2[NPIXEL*line + pixel] = 0x0;
        }
73      }
```

```
75        tty_printf(" - build   OK at cycle %d\n", proctime());
      }
77      if(fb_completed() != 0)
      {
79        tty_printf("\n!!! error in fb_completed syscall !!!\n");
          exit();
81      }
    }
83  /***** Epilogue *****/
    if(fb_write(0, BUF1, NLINE*NPIXEL) != 0)
85  {
      tty_printf("\n!!! error in fb_syn_write syscall !!!\n");
87    exit();
    }
89  tty_printf(" - display OK at cycle %d\n", proctime());
    tty_printf("\nFin du programme au cycle = %d\n\n", proctime());
91  exit();
} // end main
```

./main_pipe.c