La mémoire virtuelle

Gérer la mémoire d'un processus

- Un processus a besoin de mémoire pour
  - Le code exécutable,
  - La pile,
  - Eventuellement d'autres données de grande taille.

- La quantité de mémoire nécessaire à un processus n'est pas toujours connue *a priori* et peut varier au cours de l'exécution du processus.

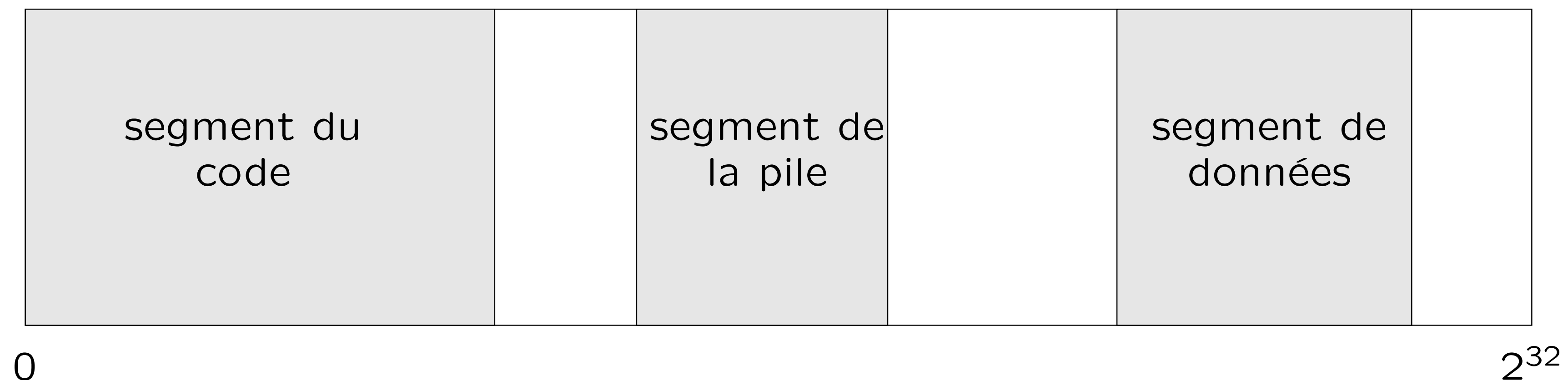
- Réserver un espace fixe pour un processus est excessivement contraignant du point de vue de processus (contrainte sur les adresses) et du point de vue du système (manque de flexibilité dans la gestion des ressources).

Une vue virtuelle de la mémoire

- Chaque processus voit un espace mémoire virtuel qui lui est propre et qui a la taille de l'espace adressable de la machine utilisée (2^{32} octets pour β).

- Dans cet espace mémoire virtuel le processus demande l'allocation de *segments* de mémoire (espaces contigus) qu'il peut alors utiliser.

- Les adresses mémoire virtuelles se trouvant dans un segment alloué sont traduites en adresses physiques ; un accès aux autres adresses donne lieu à une erreur.



Réaliser la mémoire virtuelle : la pagination

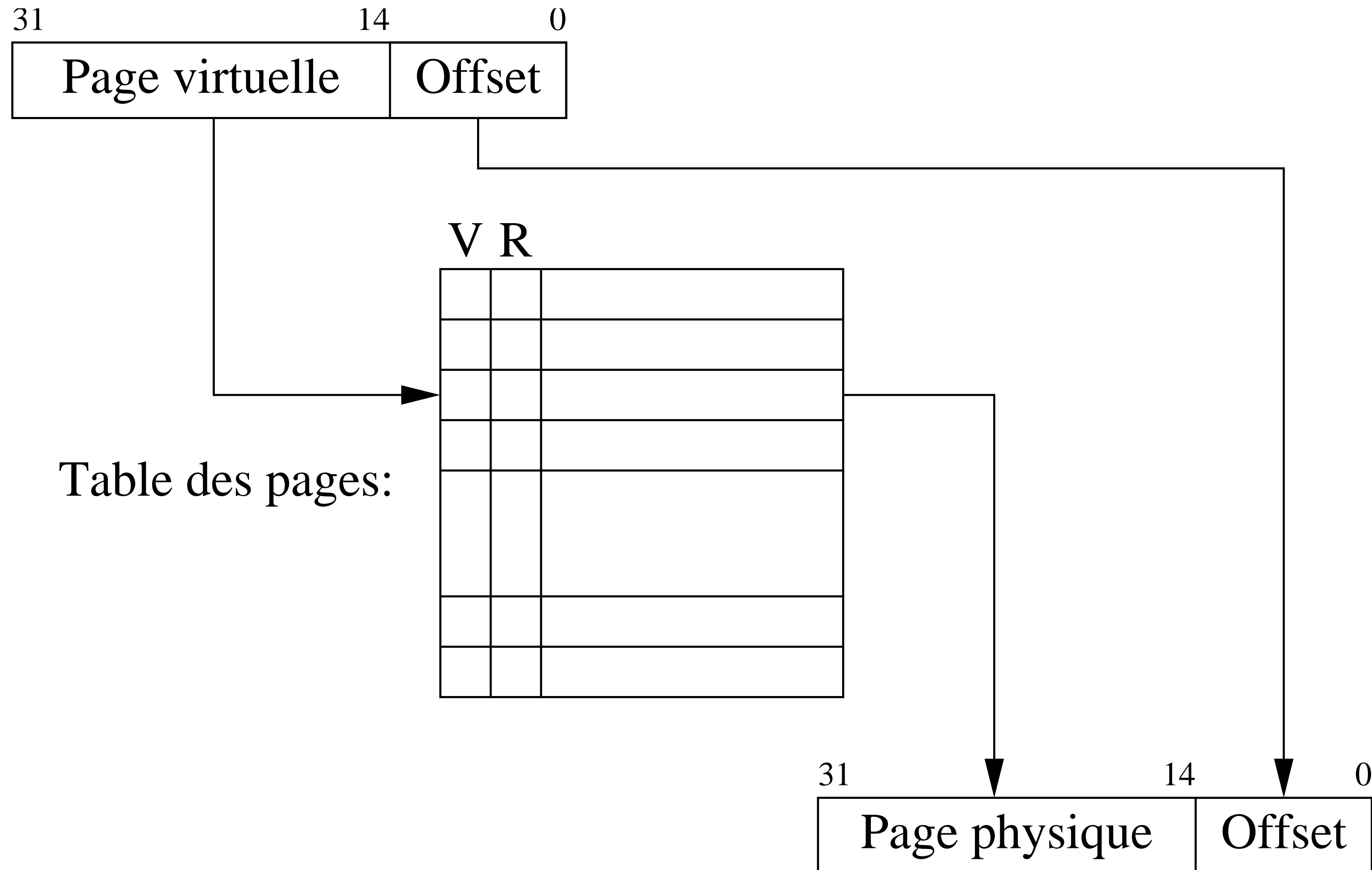
- Pour réaliser la mémoire virtuelle, on divise l'espace adressable en *pages*, par exemple 2^{18} pages de 4K mots pour la version ULg de la machine β.

- Seules certaines pages de l'espace virtuel d'un processus sont en mémoire physique.

- La correspondance entre page virtuelle et page physique est réalisée par une *table des pages*.

- Pour étendre la mémoire disponible, on peut sauver certaines pages sur un disque dur.

La traduction d'adresse du virtuel au physique



Outre l'adresse physique, la table des pages comporte de l'information sur la validité des pages. Le bit `V(alid)` indique si la page est allouée ou non, le bit `R(esident)` si la page est résidente en mémoire ou sauvée sur disque.

La table des pages : le problème de la vitesse

- Chaque fois que l'on accède à la mémoire, il faut consulter la table de pages qui elle même est aussi . . . en mémoire.

- Donc l'accès à la mémoire virtuelle nécessite un accès à la mémoire virtuelle. Un moyen simple de rompre cette circularité est de conserver la table des pages dans la mémoire du noyau qui est gérée directement sous la forme de mémoire physique.

- Même sans circularité, multiplier les accès à la mémoire ralentit très fort le fonctionnement d'une machine.

- Pour éviter cela, on maintient en mémoire rapide une partie de la table des pages. C'est ce qu'on appelle une *cache* de la table des pages, ou encore un *Translation Lookaside Buffer* (TLB).

- Si l'adresse de la page recherchée ne se trouve pas dans la cache on parle de *cache miss*

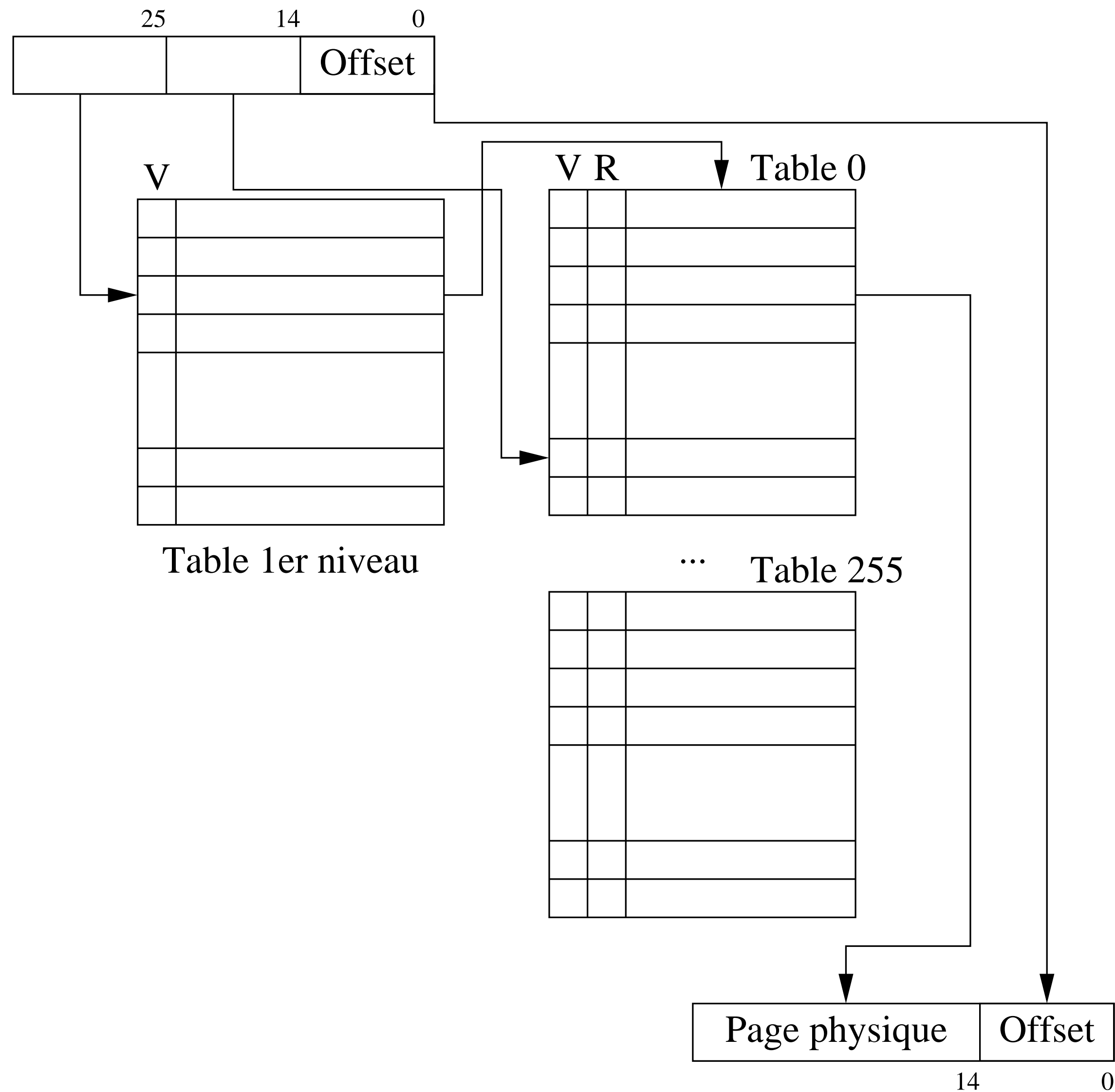
La table des pages : le problème de la taille

- Une table des pages pour un espace adressable de 2^{32} octets et des pages de 4K mots comporte 2^{18} entrées et a une taille de l'ordre de 1 MBytes. Toutefois seule une petite partie de cette table sera effectivement utilisée.

- Pour la tables des pages, on peut donc utiliser une structure de données adaptée (table hash, arbre) qui permet d'en limiter la taille tout en maintenant un accès rapide).

- Une autre solution courante est d'avoir une table à deux niveaux.

Une table des pages à deux niveaux



Seules les tables de 2ième niveau effectivement utilisées sont allouées.

La pagination : pourquoi fonctionne-t-elle?

- Si les accès à la mémoire étaient totalement aléatoires, une cache de table des pages ne donnerait pas de bon résulats et l'utilisation de la mémoire virtuelle serait compromis.

- Ce qui permet un fonctionnement correct est que les accès à la mémoire satisfont au principe de localité : des accès voisins en temps le sont souvent aussi en adresse.

- Cela est clairement vrai pour les adresses du code, mais l'est souvent aussi pour les adresses des données.

Pagination et processus

- Pour séparer la mémoire de processus partageant une machine, il suffit de prévoir une tables des pages par processus.

- Pour passer d'un processus à un autre, il suffit donc de restaurer les registres et de changer de table de pages.

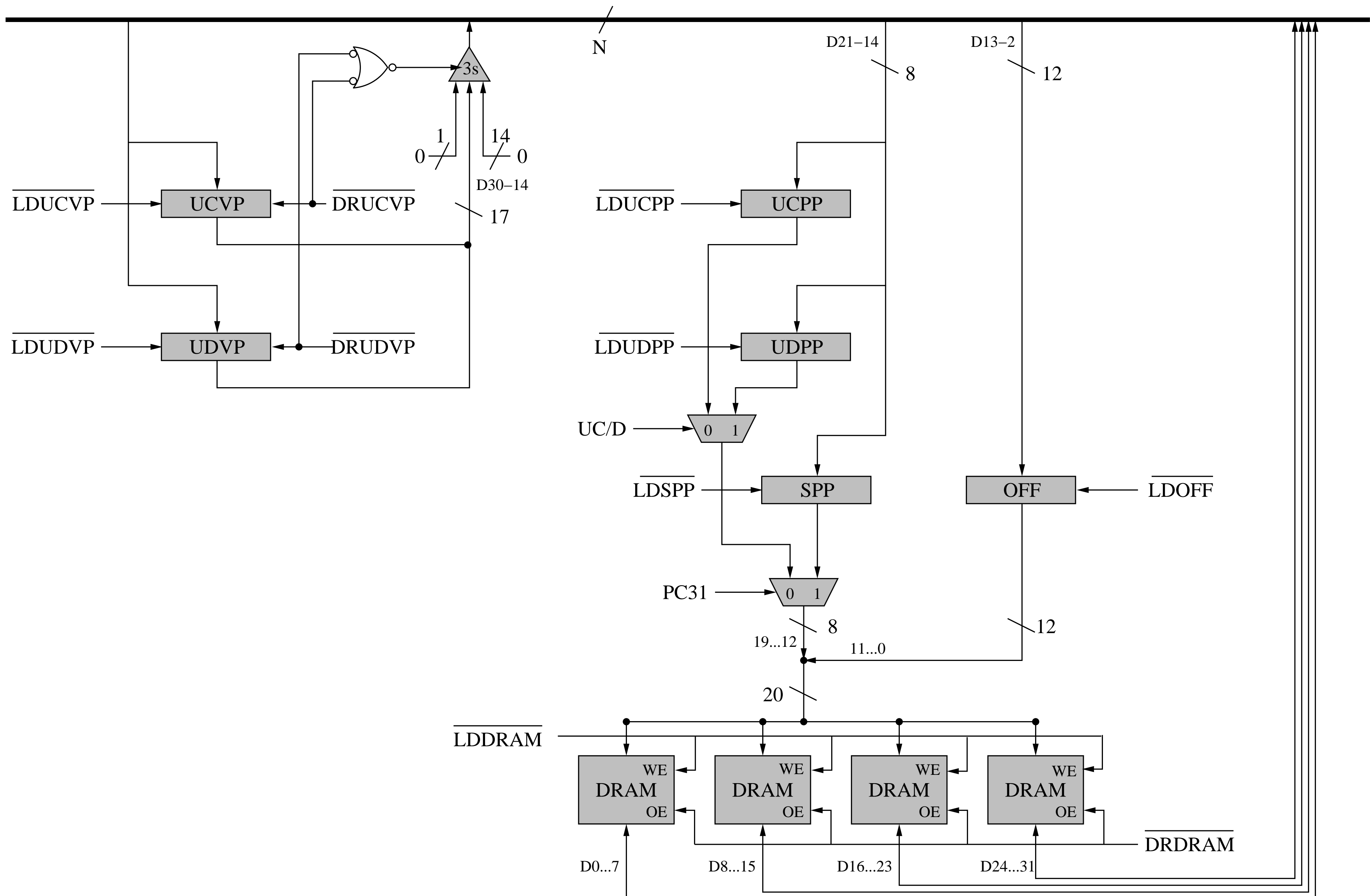
- Le système gérant les tables de pages et l'allocation de pages aux processus, il est impossible à un processus d'avoir accès à la mémoire d'un autre. Les processus travaillent donc dans des *contextes* bien distincts et isolés.

- Il est possible de prévoir plusieurs caches des tables de pages et donc de disposer de plusieurs contextes accessibles rapidement.

Une implémentation de la mémoire virtuelle dans l'architecture β : ULg03

- Les processus utilisateurs travaillent en mémoire virtuelle, chacun dans son contexte ; le noyau travaille directement en mémoire physique.
- Il y a deux contextes disponibles en hardware : un contexte utilisateur et le contexte du noyau (mémoire physique).
- La cache de la table des pages est divisée en deux parties comportant chacune une adresse de page : une partie utilisée pour l'accès aux données (`LD` et `ST`) ; une partie utilisée pour l'accès aux instructions (instruction suivante, `JMP` et `BR`).
- Il y a des instructions spéciales, uniquement accessibles en mode superviseur, pour accéder à la cache de la table des pages.
- Les tables de page sont conservées dans la mémoire du noyau. Un accès à une page dont l'adresse ne se trouve pas dans la cache génère une exception qui est gérée par une routine du noyau.

Le module RAM dynamique de ULg03



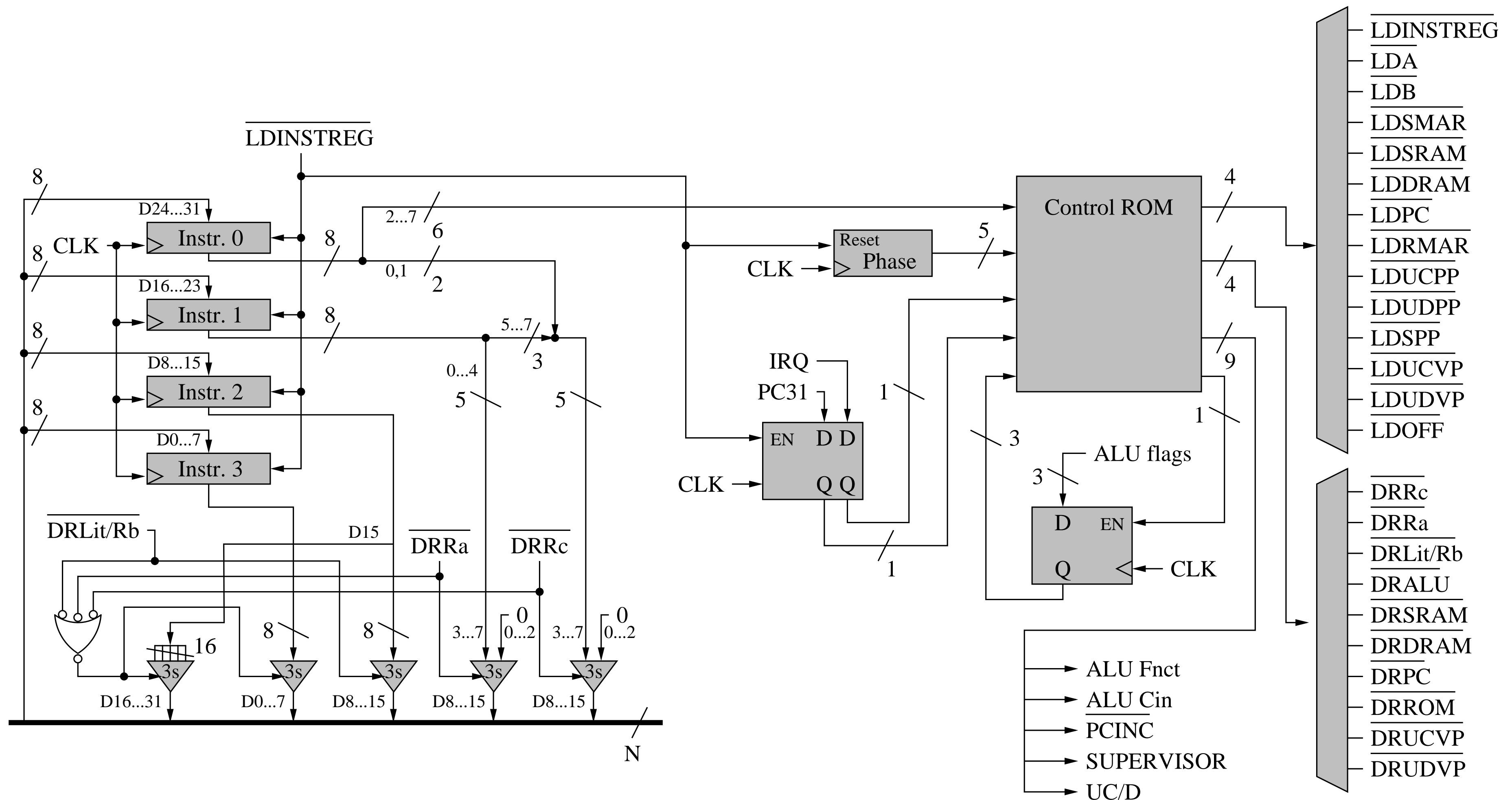
Le module RAM dynamique de ULg03 (suite)

- Les noms des registres sont les suivants :
  `UCVP`, `UCPP` : User Code Virtual/Physical Page.
  `UDVP`, `UDPP` : User Data Virtual/Physical Page.
  `SPP` : System Physical Page

- En mode superviseur (`PC31` à 1), les adresses utilisées sont directement des adresses physiques.

- En mode utilisateur (`PC31` à 0), suivant que l'on accède à une page de code ou de données (signal `UC/D`), on utilise l'adresse de page `UCPP` ou `UDPP`, pour autant que le numéro de page virtuelle coïncide avec le contenu de `UCVP` (`UDVP`), ce qui est vérifié par le microcode. Si non, il y a une exception.

L'unité de contrôle de ULg03



- On passe à 32 phases (microcode plus long).
- Nouveau signaux `LD` et `DR` pour contrôler les registres d'adresse de la DRAM ; nouveau signal `UC/D`.

La ROM de constantes de ULg03

Adresse	Contenu
0x00	⋮
	⋮
0xF9	10000000 00000000 00100000 00010000 *(Adresse du handler "Illegal Operation" — 0x2016)*
0xFA	10000000 00000000 00100000 00001100 *(Adresse du handler "IRQ" — 0x2012)*
0xFB	10000000 00000000 00100000 00001000 *(Adresse du handler "Supervisor Call" — 0x2008)*
0xFC	10000000 00000000 00100000 00000100 *(Adresse du handler "Cache Miss Code" — 0x2004)*
0xFD	10000000 00000000 00100000 00000000 *(Adresse du handler "Cache Miss Data" — 0x2000)*
0xFE	00000000 00000000 11110000 00000000 *(Adresse du registre XP)*
0xFF	00000000 00000000 00111111 11111111 *(Masque d'offset)*

La ROM de constantes de ULg03 (suite)

- On a ajouté un *masque d'offset* pour pouvoir séparer numéro de page et offset.

- Il y a deux nouvelles adresses de handlers (*Cache Miss Data* et *Cache Miss Code*).

- Les adresses des handlers ont été modifiées pour qu'elles soient groupées (les handlers débutent par un "branch").

Le microcode de ULg03

- Chaque instruction devient différente en mode superviseur et en mode utilisateur (il y a toujours un accès à la mémoire pour passer à l'instruction suivante).
- En mode superviseur, les adresses sont des adresses physiques et sont donc utilisées directement.
- En mode utilisateur, l'adresse de page est comparée à celle contenue dans `UC(D)VP`. S'il y a coïncidence, l'adresse de page physique contenue dans `UC(D)PP` est utilisée. Si non, une exception a lieu.
- Lors d'un *cache miss* sur code on revient à l'instruction suivante (ou à la destination du saut) ; lors d'un *cache miss* sur données, on réexécute l'instruction qui en est la cause.
- L'instruction `JMP` est un cas intéressant car elle permet de passer d'une adresse physique (mode superviseur) à une adresse virtuelle (mode utilisateur).

Le microcode de LD

LD(Ra, Literal, Rc) (mode utilisateur)

Opcode = 011000   IRQ = 0   PC31 = 0

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F, $\overline{C_{in}}$, M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0010	0100	0	0	B <- SRAM
00010	*	1	0	000000	0001	0010	0	0	A <- Lit
00011	*	1	0	100110	0001	0011	0	0	A <- A+B
00100	*	1	0	000000	0010	1001	0	0	B <- UDVP
00101	*	1	0	111111	1101	0011	0	0	OFF <- A
00110	*	1	0	111111	1100	0011	0	0	UDVP <- A
00111	*	1	0	100101	0010	0011	0	0	B <- A /XOR B
01000	*	1	0	110011	0111	0011	0	0	RMAR <- 0xFFFFFFFF
01001	*	1	0	000000	0001	0111	0	0	A <- ROM
01010	*	0	0	111001	0001	0011	0	0	A <- A OR B; Latch

A contient uniquement des 1 si et seulement si le numéro de page virtuelle coïncide avec celui contenu dans la cache.

Il y a *cache miss* pour les données ; il faut sauter au handler correspondant qui doit revenir en `XP -4`, ce qui signifie que l'instruction sera réexécutée.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01011	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01100	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
01101	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
01110	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
01111	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10000	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10001	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
10010	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10011	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10100	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Noter que, à partir de la phase `10010`, `PC31` vaut 1 et donc que l'on travaille avec des adresses physiques.

Il n'y a pas de *cache miss* données ; il faut charger l'instruction suivante en vérifiant qu'il n'y a pas de *cache miss* code.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01011	E=1	1	0	000000	0011	0000	0	0	SMAR <- Rc
01100	E=1	1	1	000000	0100	0101	0	0	SRAM <- DRAM; UC/D
01101	E=1	1	0	000000	0010	1000	0	0	B <- UCVP
01110	E=1	1	0	000000	0001	0110	0	0	A <- PC
01111	E=1	1	0	111111	1011	0011	0	0	UCVP <- A
10000	E=1	1	0	100101	0010	0011	0	0	B <- A /XOR B
10001	E=1	1	0	110011	0111	0011	0	0	RMAR <- 0xFFFFFFFF
10010	E=1	1	0	000000	0001	0111	0	0	A <- ROM
10011	E=1	1	0	111111	0001	0011	0	0	A <- A
10100	E=1	0	0	111001	0001	0011	0	0	A <- A OR B; Latch

Il y a *cache miss* pour le code ; il faut sauter au handler correspondant qui doit revenir en `XP`, ce qui permettra d'exécuter l'instruction suivante.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
10101	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
10110	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10111	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
11000	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
11001	E=0	1	0	111110	0001	0011	0	0	A <- A-1
11010	E=0	1	0	111110	0001	0011	0	0	A <- A-1
11011	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
11100	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
11101	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
11110	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
11111	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Noter que, à partir de la phase 29, `PC31` vaut 1.

Il n'y a pas de *cache miss* relatif au code ; on passe à l'instruction suivante.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
10101	E=1	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10110	E=1	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Le micro code de JMP

JMP(Ra, Rc) (mode utilisateur)

Opcode = 011011   IRQ = 0   PC31 = 0

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F, $\overline{C_{in}}$, M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00011	*	1	0	000000	0100	0110	0	0	SRAM <- PC
00100	*	1	0	000000	0010	1000	0	0	B <- UCVP
00101	*	1	0	111111	1011	0010	0	0	UCVP <- A
00110	*	1	0	111111	1101	0010	0	0	OFF <- A
00111	*	1	0	111111	0110	0001	0	0	PC <- A
01000	*	1	0	100101	0010	0011	0	0	B <- A /XOR B
01001	*	1	0	110011	0111	0011	0	0	RMAR <- 0xFFFFFFFF
01010	*	1	0	000000	0001	0111	0	0	A <- ROM
01011	*	0	0	111001	0001	0011	0	0	A <- A OR B; Latch

Pas de *cache miss* sur le code, charger la destination du JMP

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F,\overline{C_{in}},M$	LD SEL	DR SEL	PC+	SVR	
01100	E=1	1	0	000000	0000	0101	1	0	INSTREG <- DRAM; PC+

Cache miss de code, sauter au handler correspondant.

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU $F,\overline{C_{in}},M$	LD SEL	DR SEL	PC+	SVR	
01100	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01101	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
01110	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
01111	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
10000	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10001	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10010	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10011	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
10100	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10101	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10110	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

JMP(Ra, Rc) (mode superviseur)

Opcode = 011011   IRQ = *   PC31 = 1

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00011	*	1	0	000000	0100	0110	0	0	SRAM <- PC
00100	*	0	0	111111	0011	0010	0	0	A <- A; Latch

Saut vers une adresse dont le bit de poids fort est 1 : on reste en mode superviseur et on gère une adresse physique.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00101	N=1	1	0	111111	0110	0010	0	1	PC <- A; SVR
00110	N=1	1	0	000000	1010	0110	0	0	SPP <- PC
00111	N=1	1	0	000000	1101	0110	0	0	OFF <- PC; PC+
01000	N=1	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Saut vers une adresse dont le bit de poids fort est 0 : on passe en mode utilisateur et on gère une adresse virtuelle qu'il faut traduire si nécessaire. A partir de la phase 9, PC31 vaut 0.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00101	N=0	1	0	000000	0010	1000	0	0	B <- UCVP
00110	N=0	1	0	111111	1011	0010	0	0	UCVP <- A
00111	N=0	1	0	111111	1101	0010	0	0	OFF <- A
01000	N=0	1	0	111111	0110	0001	0	0	PC <- A
01001	N=0	1	0	100101	0010	0011	0	0	B <- A /XOR B
01010	N=0	1	0	110011	0111	0011	0	0	RMAR <- 0xFFFFFFFF
01011	N=0	1	0	000000	0001	0111	0	0	A <- ROM
01100	N=0	0	0	111001	0001	0011	0	0	A <- A OR B; Latch

Le reste se passe comme pour un JMP en mode utilisateur.

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
01101	E=1	1	0	000000	0000	0101	1	0	INSTREG <- DRAM; PC+
01101	E=0	1	0	110011	0001	0011	0	0	A <- 0xFFFFFFFF
01110	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
01111	E=0	1	0	000000	0011	0111	0	0	SMAR <- ROM
10000	E=0	1	0	000000	0100	0110	0	0	SRAM <- PC
10001	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10010	E=0	1	0	111110	0001	0011	0	0	A <- A-1
10011	E=0	1	0	111110	0111	0011	0	0	RMAR <- A-1
10100	E=0	1	0	000000	0110	0111	0	1	PC <- ROM; SVR
10101	E=0	1	0	000000	1010	0110	0	0	SPP <- PC
10110	E=0	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
10111	E=0	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Les instructions spéciales du mode superviseur

On introduit quatre nouvelles instructions spéciales, uniquement disponibles en mode superviseur, pour implémenter les handler de *cache miss*

Opcode	nom	définition
001000	RDUCVP(Rc)	Rc <- UCVP
001001	RDUDVP(Rc)	Rc <- UDVP
001010	WRUCPP(Rc)	UCPP <- Reg[Rc]
001011	WRUDPP(Rc)	UDPP <- Reg[Rc]
001100	WRUCVP(Rc)	UCVP <- Reg[Rc]
001101	WRUDVP(Rc)	UDVP <- Reg[Rc]

Les macros correspondantes sont les suivantes.

```
.macro RDUCVP(Rc)          ENC_NOLIT(0b001000,0,0,Rc)
.macro RDUDVP(Rc)          ENC_NOLIT(0b001001,0,0,Rc)
.macro WRUCPP(Rc)          ENC_NOLIT(0b001010,0,0,Rc)
.macro WRUCPP(Rc)          ENC_NOLIT(0b001011,0,0,Rc)
.macro WRUCVP(Rc)          ENC_NOLIT(0b001100,0,0,Rc)
.macro WRUCVP(Rc)          ENC_NOLIT(0b001101,0,0,Rc)
```

Le microcode de RDUCVP

RDUCVP(Rc) (mode superviseur) : Rc ¡- UCVP

Opcode = 001000   IRQ = *   PC31 = 1

Phase	Fl.	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00001	*	1	0	000000	0100	1000	0	0	SRAM <- UCVP
00010	*	1	0	000000	1010	0110	0	0	SPP <- PC
00011	*	1	0	000000	1101	0110	1	0	OFF <- PC; PC+
00100	*	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

On travaille en mode superviseur et donc avec des adresses physiques.

Un handler pour les exceptions “cache miss” data

Le handler débute par le stub suivant.

```
h_stub:  SUBC(XP, 4, XP)        | prévoir de  reéxécuter l'instruction
                                | l'instruction suspendue
         ST(r0, User, r31)      | sauver
         ST(r1, User+4, r31)
              . . .
         ST(r30, User+30*4)
         CMOVE(KStack, SP)      | Charger le SP du système
         RDUDVP(r1)             | l'adresse de page virtuelle
                                | à traduire
         PUSH(r1)               | la transmettre en argument

         BR(CMHandler,LP)       | Appel du Handler
         DEALLOCATE(1)          | enlever l'argument de la pile
         WRUDPP(r0)             | installer la valeur reçue
         LD(r31, User, r0)      | restaurer
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)                | retour à l'application
```

La partie C du handler est la suivante.

```
struct PTentry {short valid, resid ; int PhysPage} PageMap[262144]:
      /* La table des page du processus courant, 2^18 entrées*/

CMHandler(VpageNo)
{ if (PageMap[VpageNo].valid != 0 && PageMap[VpageNo].resid != 0)
       return PageMap[VpageNo].PhysPage ;
     else Pagerror(VpageNo);
}
```

L'erreur peut être une page non allouée au processus (faute de segment), ou une page sauvée qu'il faut recharger en mémoire (faute de page).