PROJET ASTRE

Modélisation et Vérification de systèmes concurrents

3 janvier 2022

Inas KACI - Aymeric PLOTON

Master SAR - Sorbonne Université

Étude du protocole et des accès aux données partagées

Analyse de signaux sur mono-processeur

SCIENCES SORBONNE UNIVERSITÉ

Hypothèses

- La mémoire est initialisé à 0
- Le signal *req_i* est maintenu jusqu'à une réponse **VALID** de la mémoire (cf Figure 3 page 3).

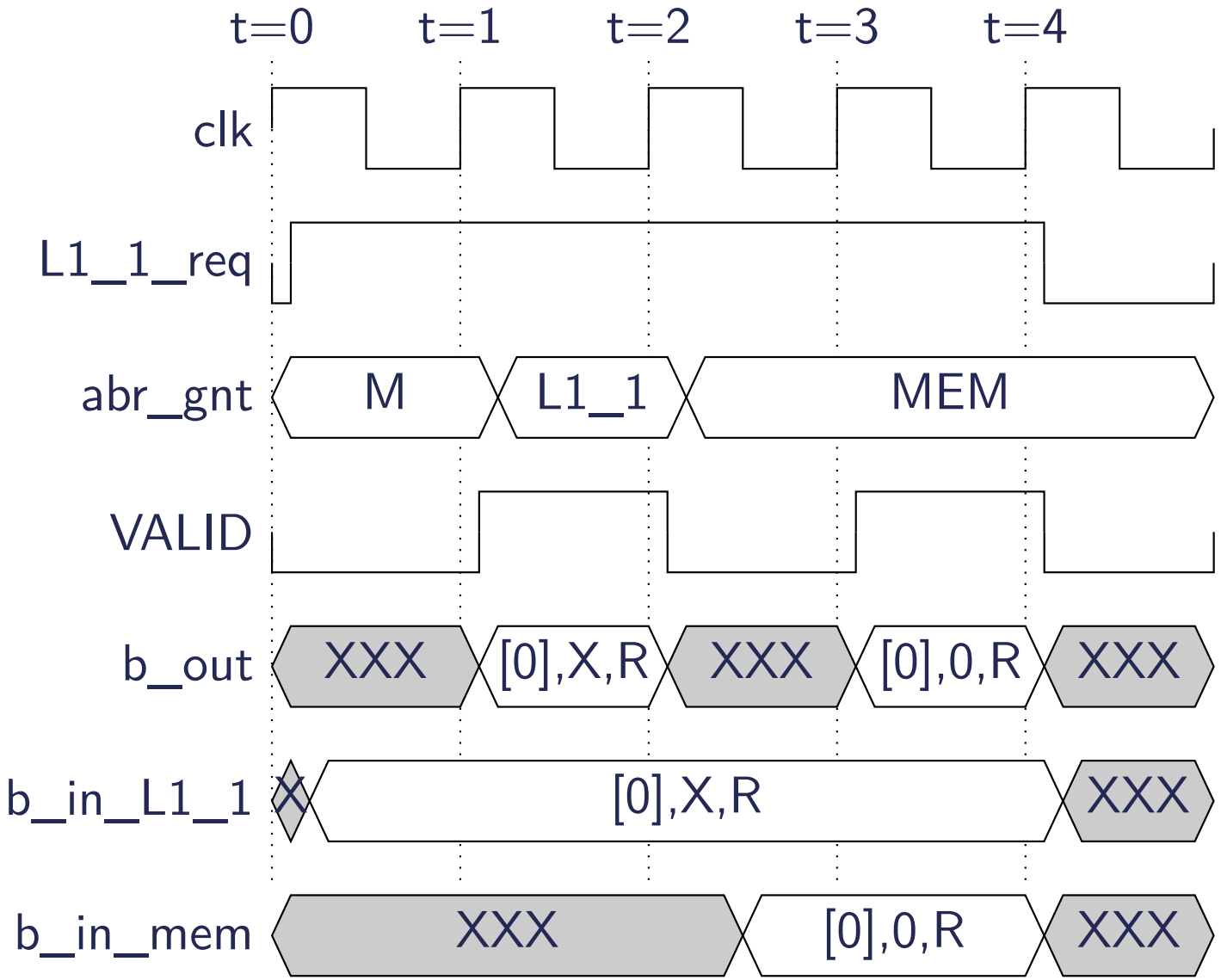
Programme P1a (mono-processeur)

SCIENCES SORBONNE UNIVERSITÉ

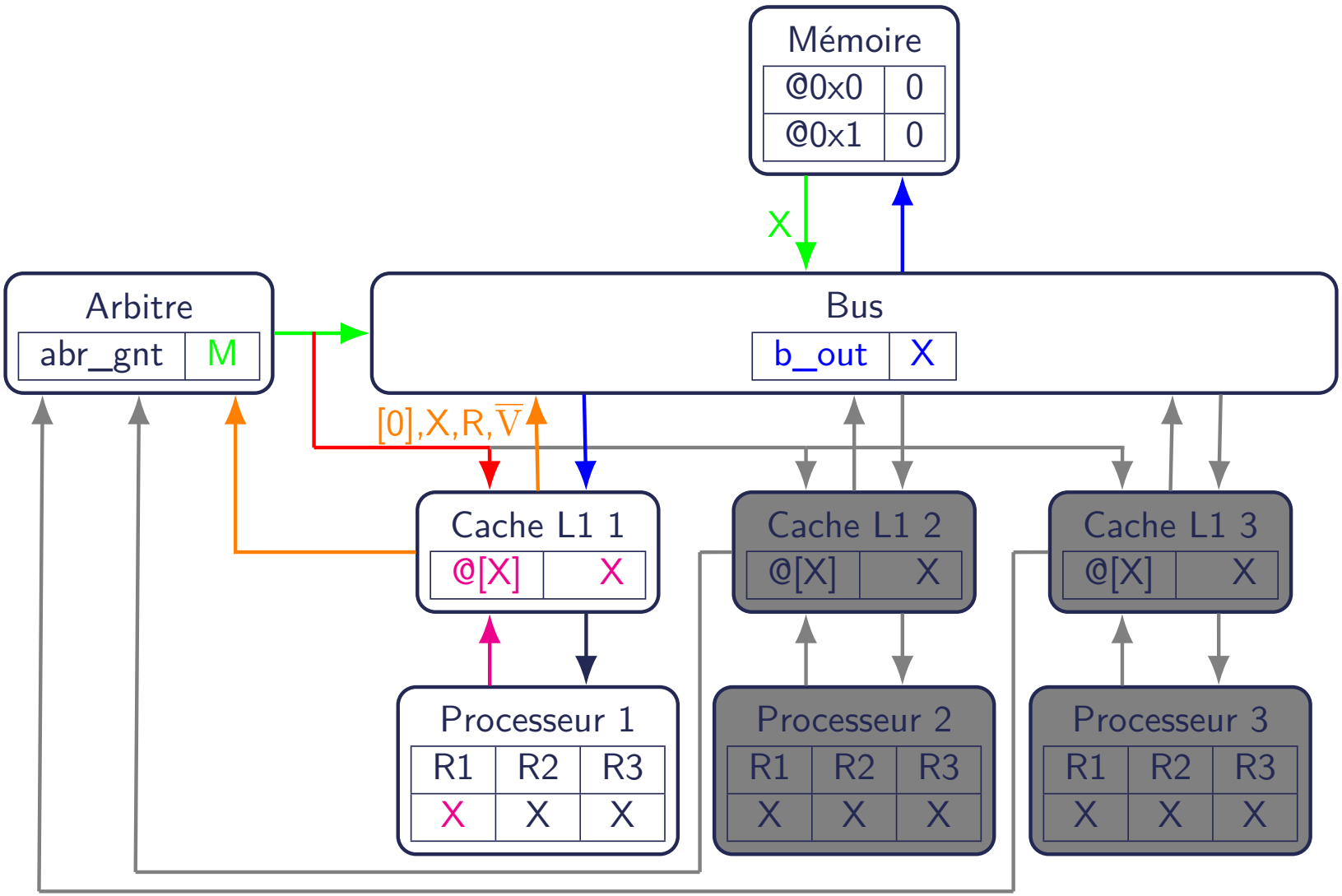
P1a.asm

```
1   ld  R1, [0]
2   add R1, R1, 1
3   st  R1, [1]
4   ld  R2, [0]
5   add R2, R2, 1
6   st  R2, [0]
```

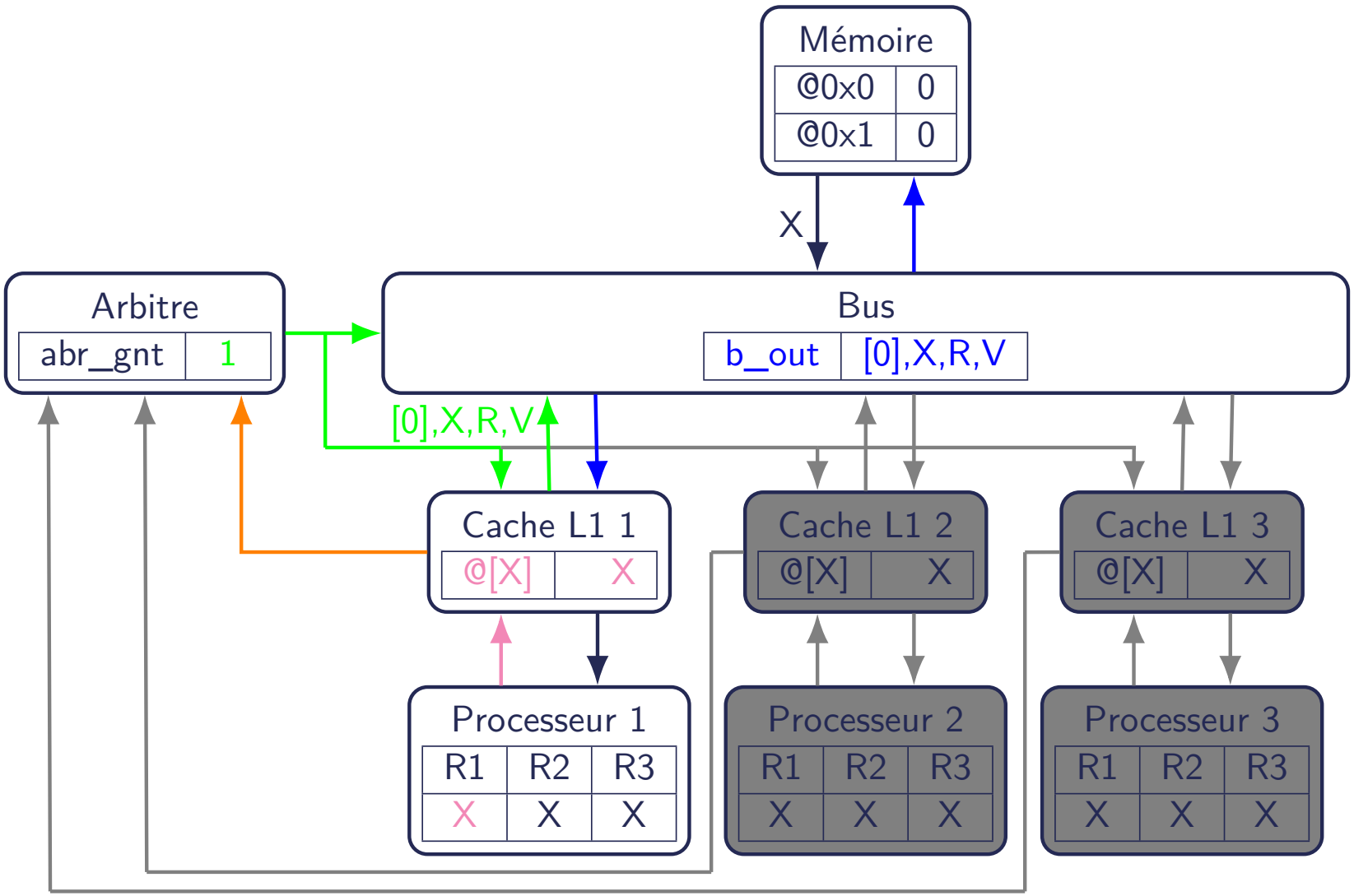
Chronogramme P1a (mono-processeur) : ld R1, [0]



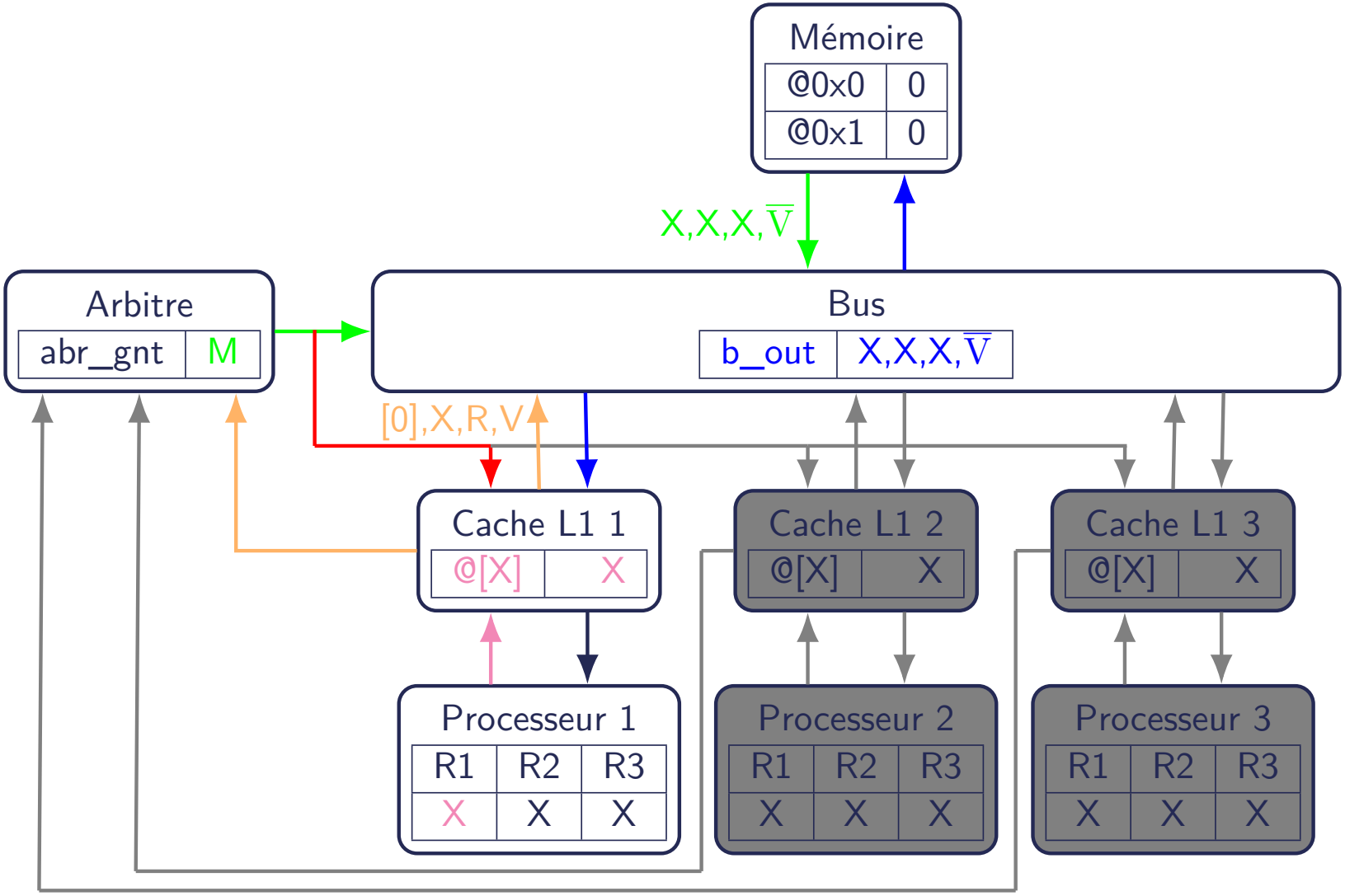
Schema P1a (mono-processeur) : ld R1, [0] (cycle 0)



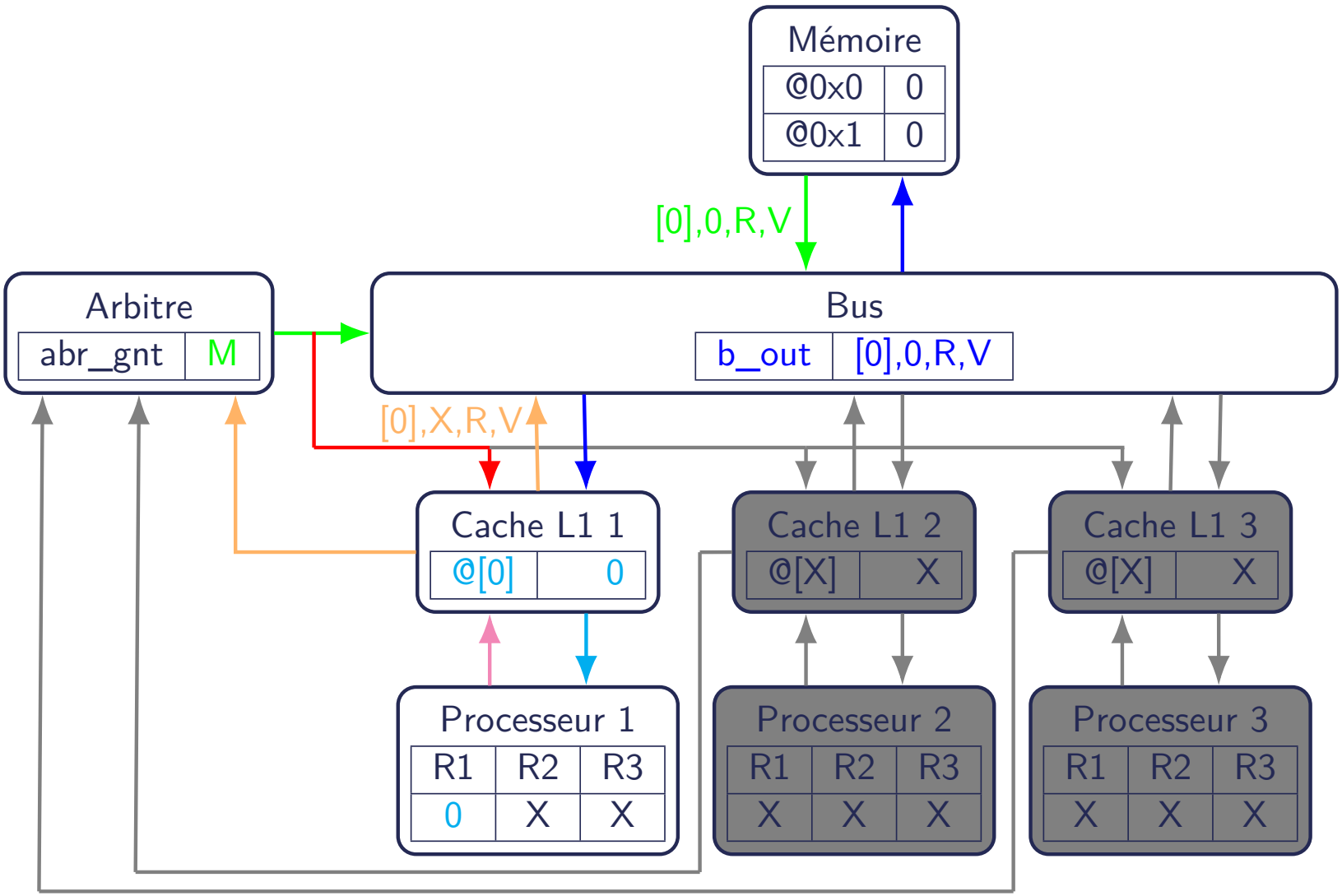
Schema P1a (mono-processeur) : ld R1, [0] (cycle 1)



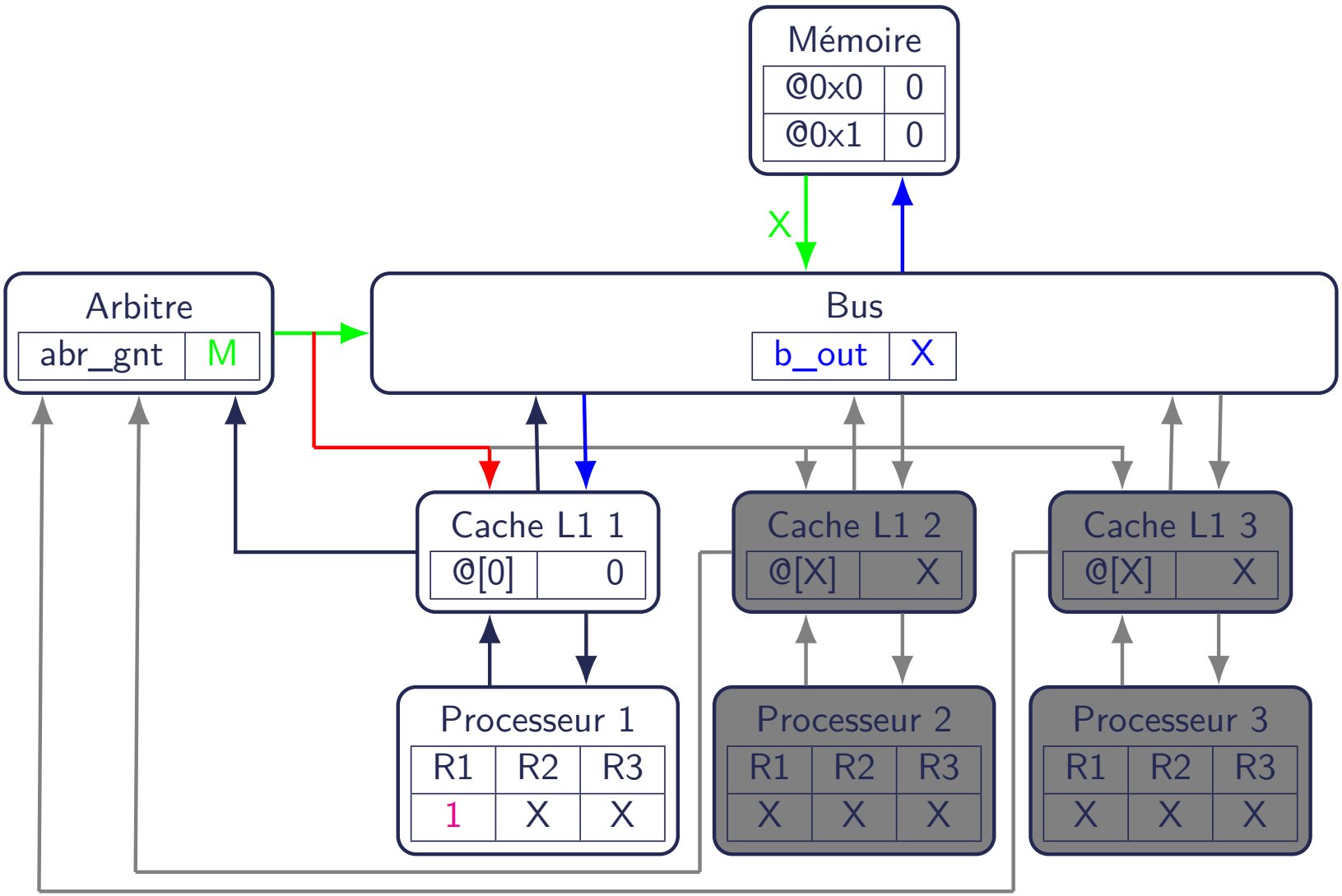
Schema P1a (mono-processeur) : ld R1, [0] (cycle 2 ~(n))



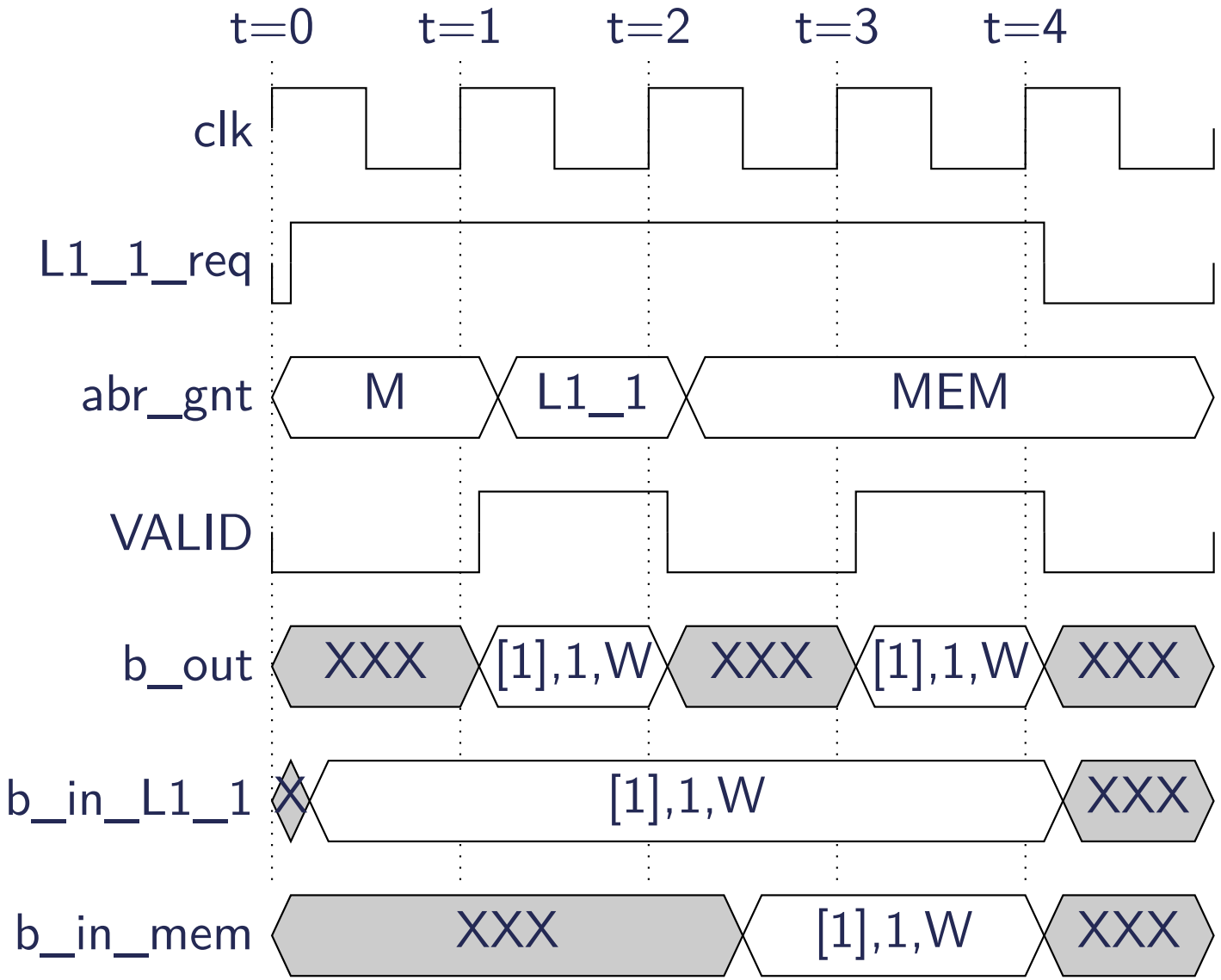
Schema P1a (mono-processeur) : ld R1, [0] (cycle 3 [~(n+1)])



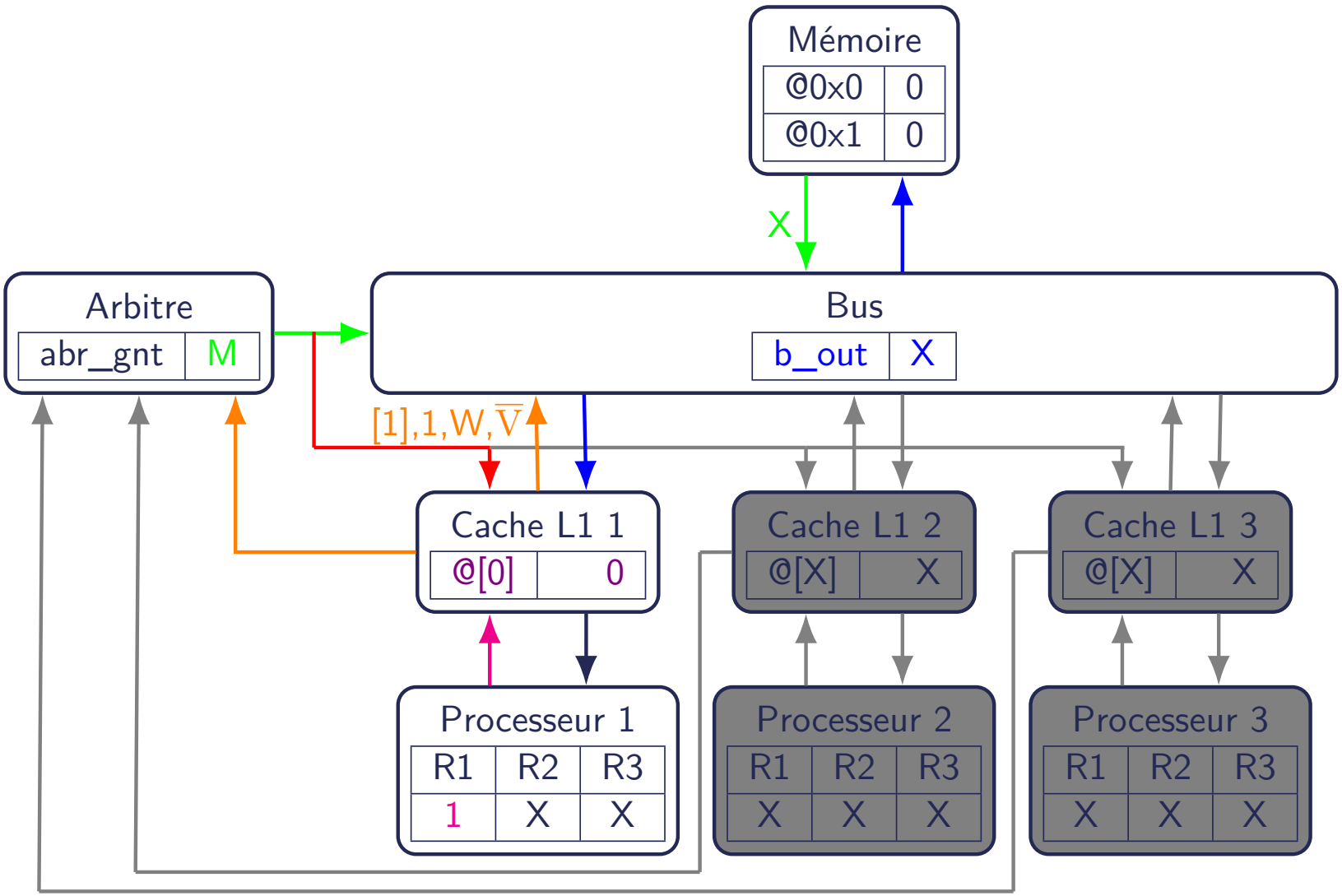
Schema P1a (mono-processeur) : add R1, R1, 1 (cycle 4)



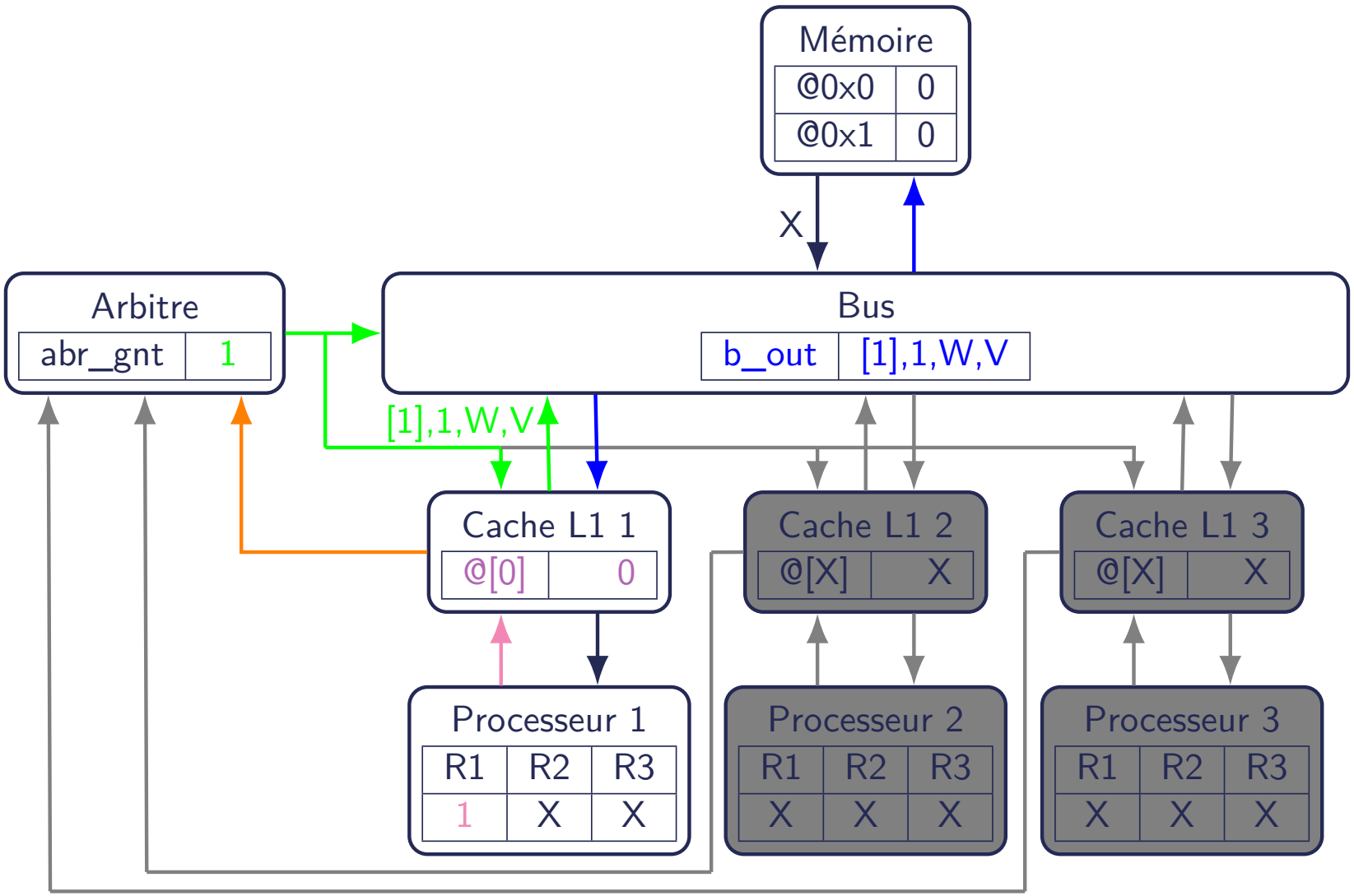
Chronogramme P1a (mono-processeur) : st R1, [1]



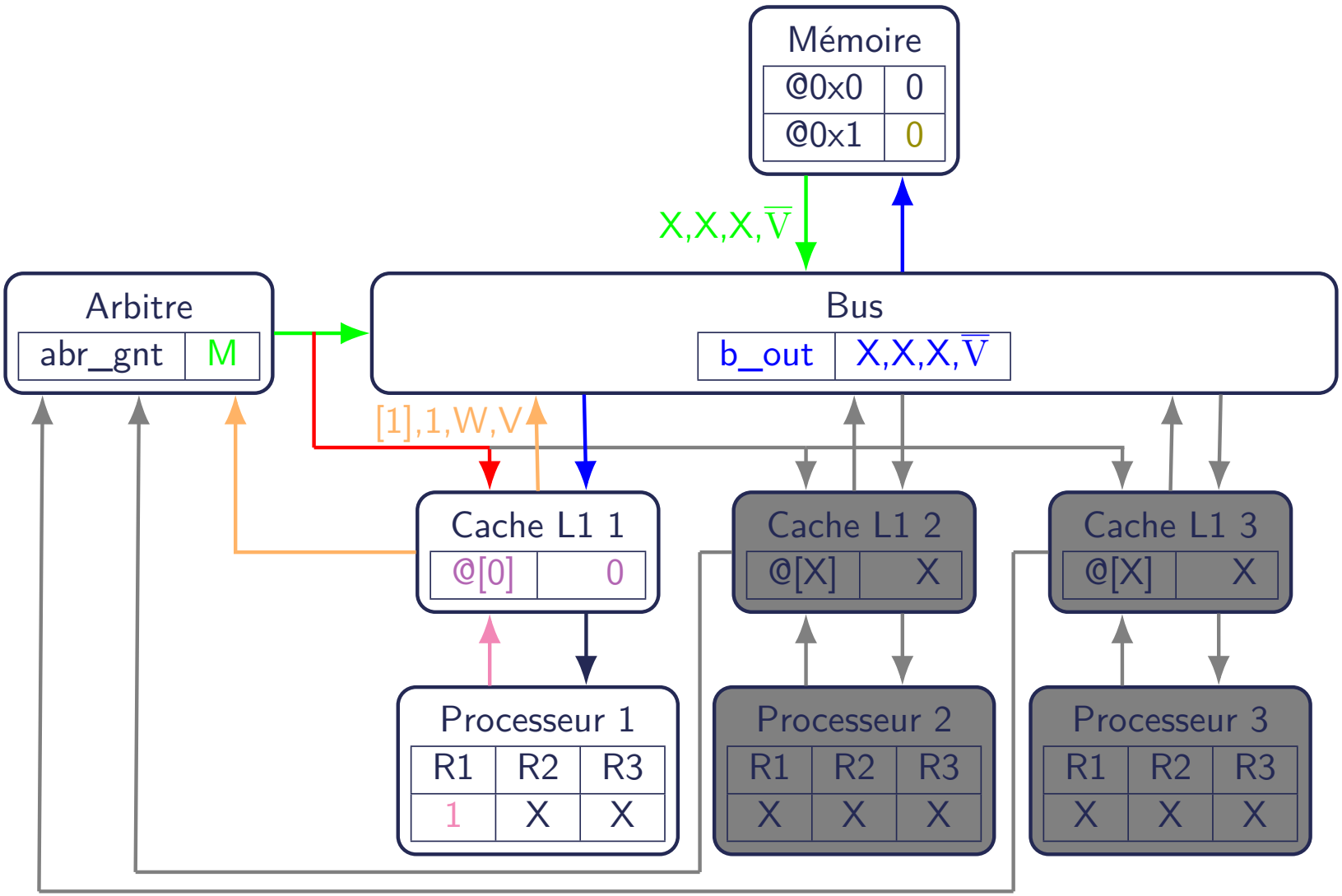
Schema P1a (mono-processeur) : st R1, [1] (cycle 0)



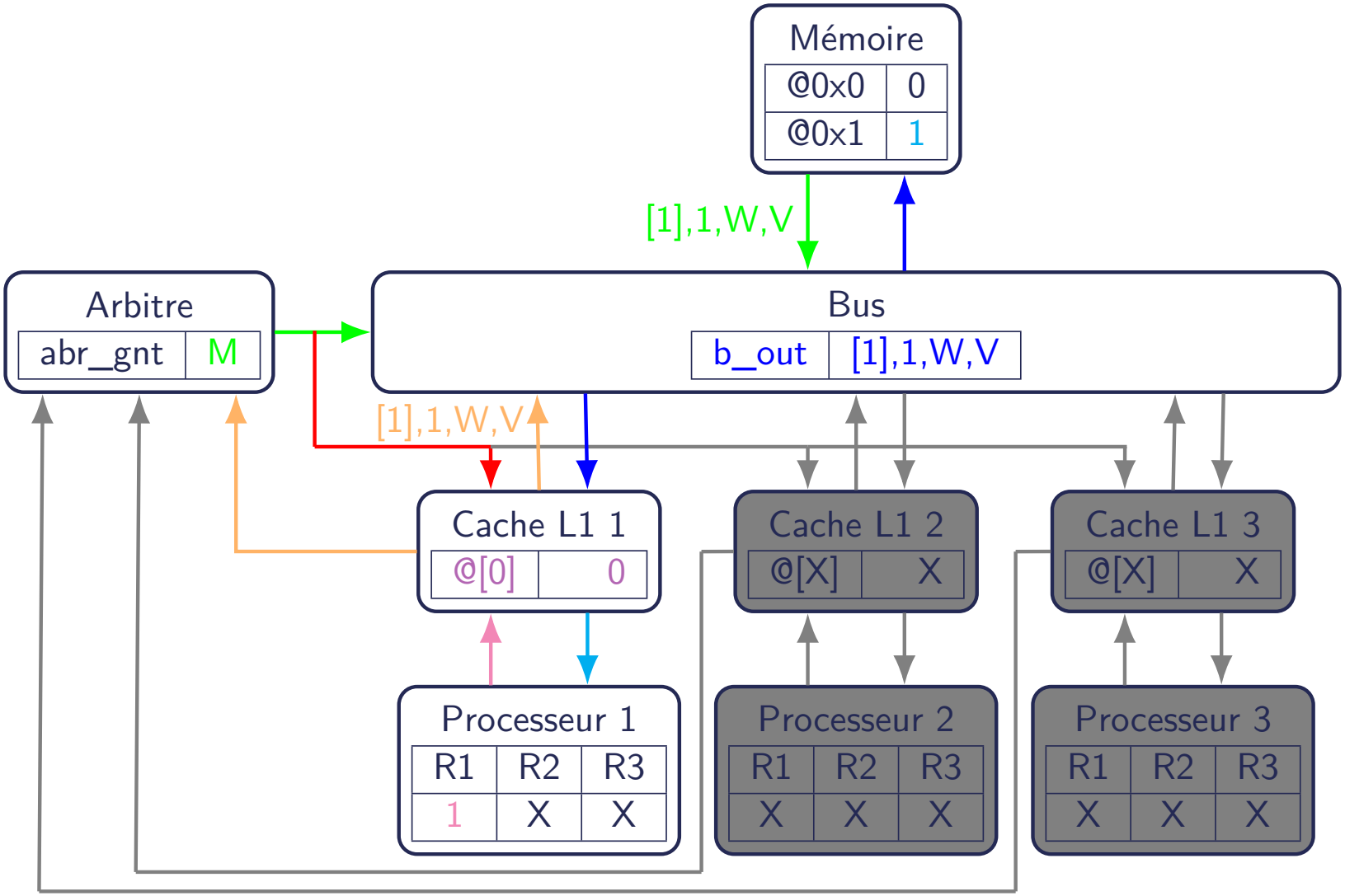
Schema P1a (mono-processeur) : st R1, [1] (cycle 1)



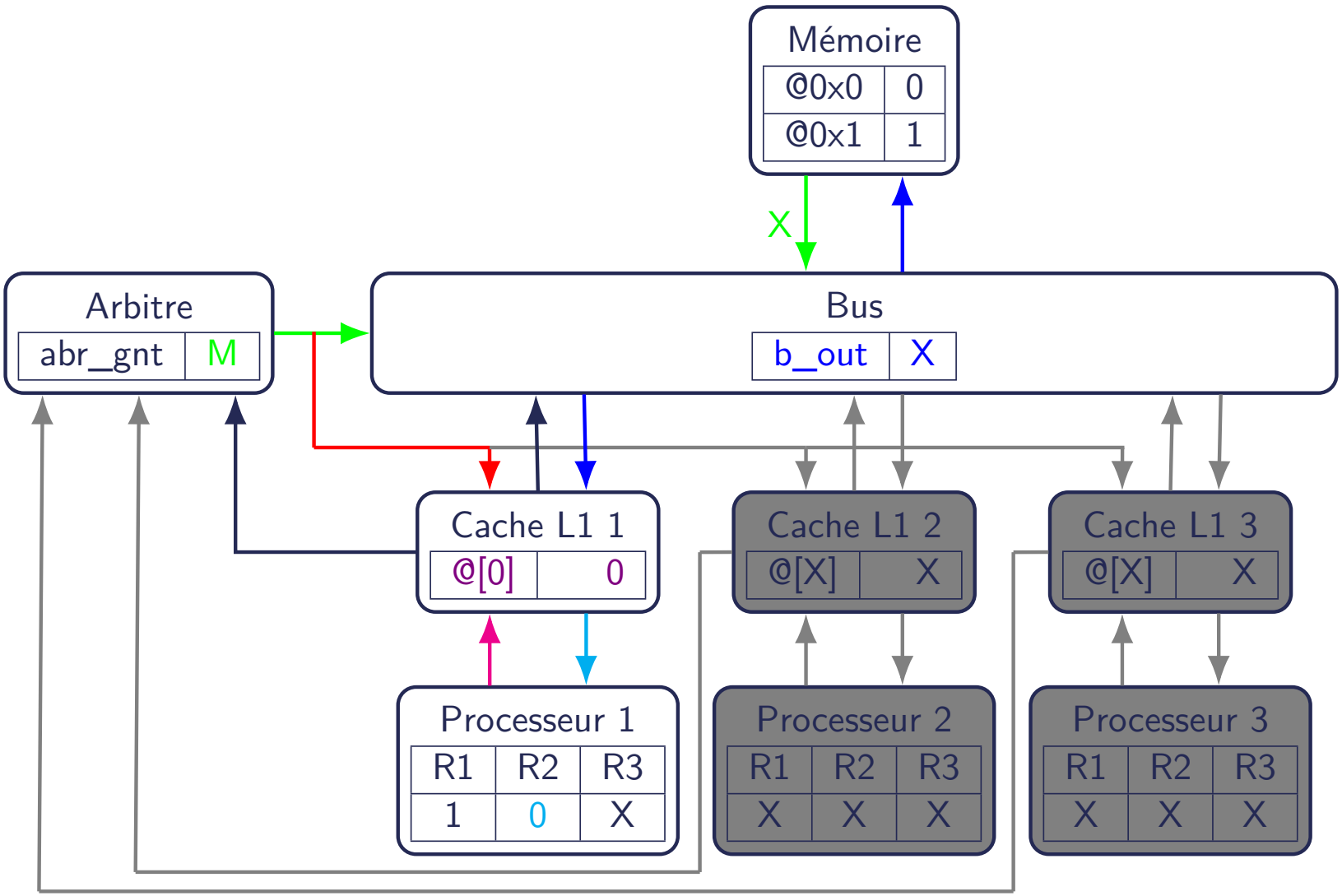
Schema P1a (mono-processeur) : st R1, [1] (cycle 2 ~(n))



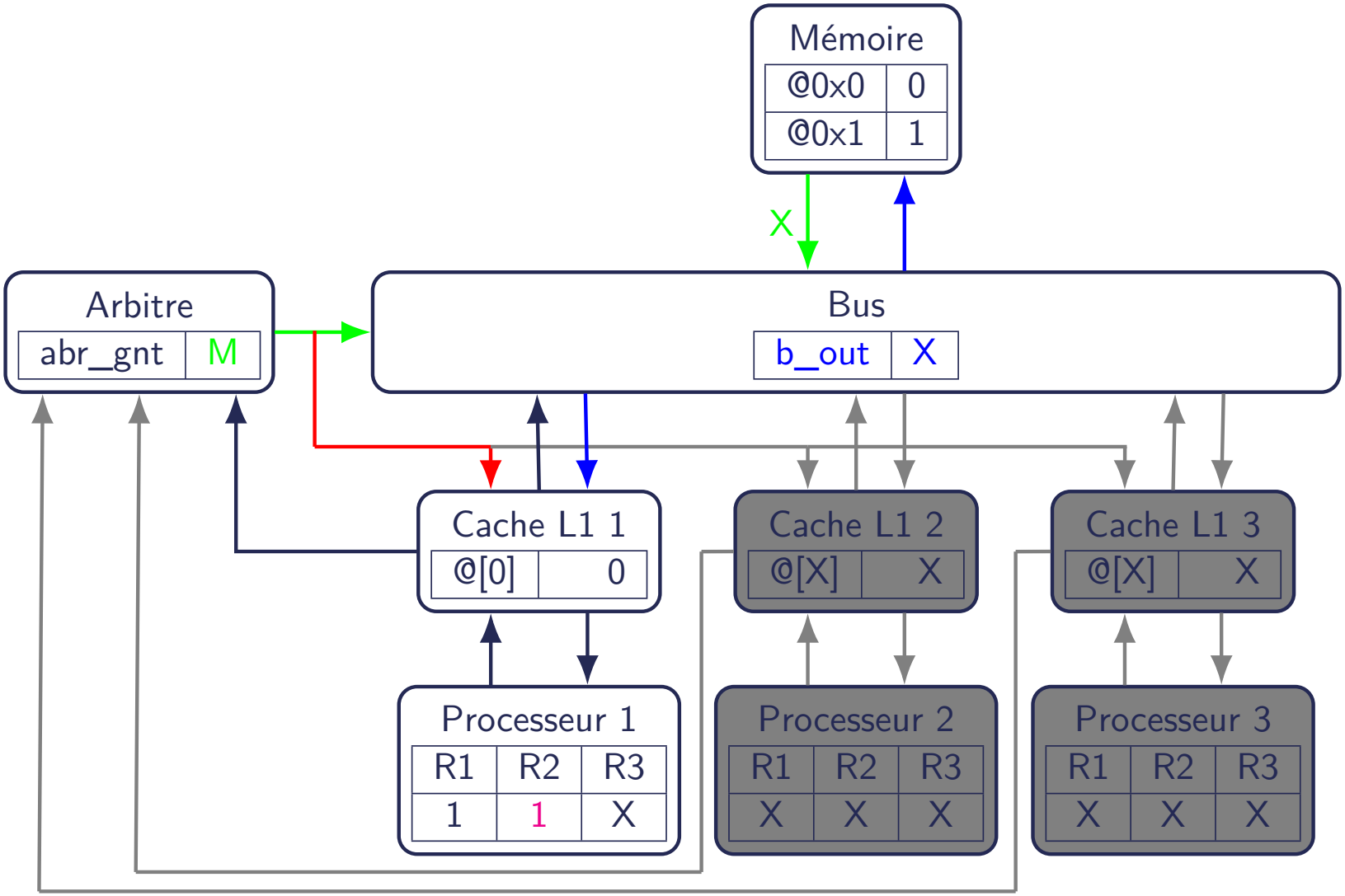
Schema P1a (mono-processeur) : st R1, [1] (cycle 3 ~(n+1))



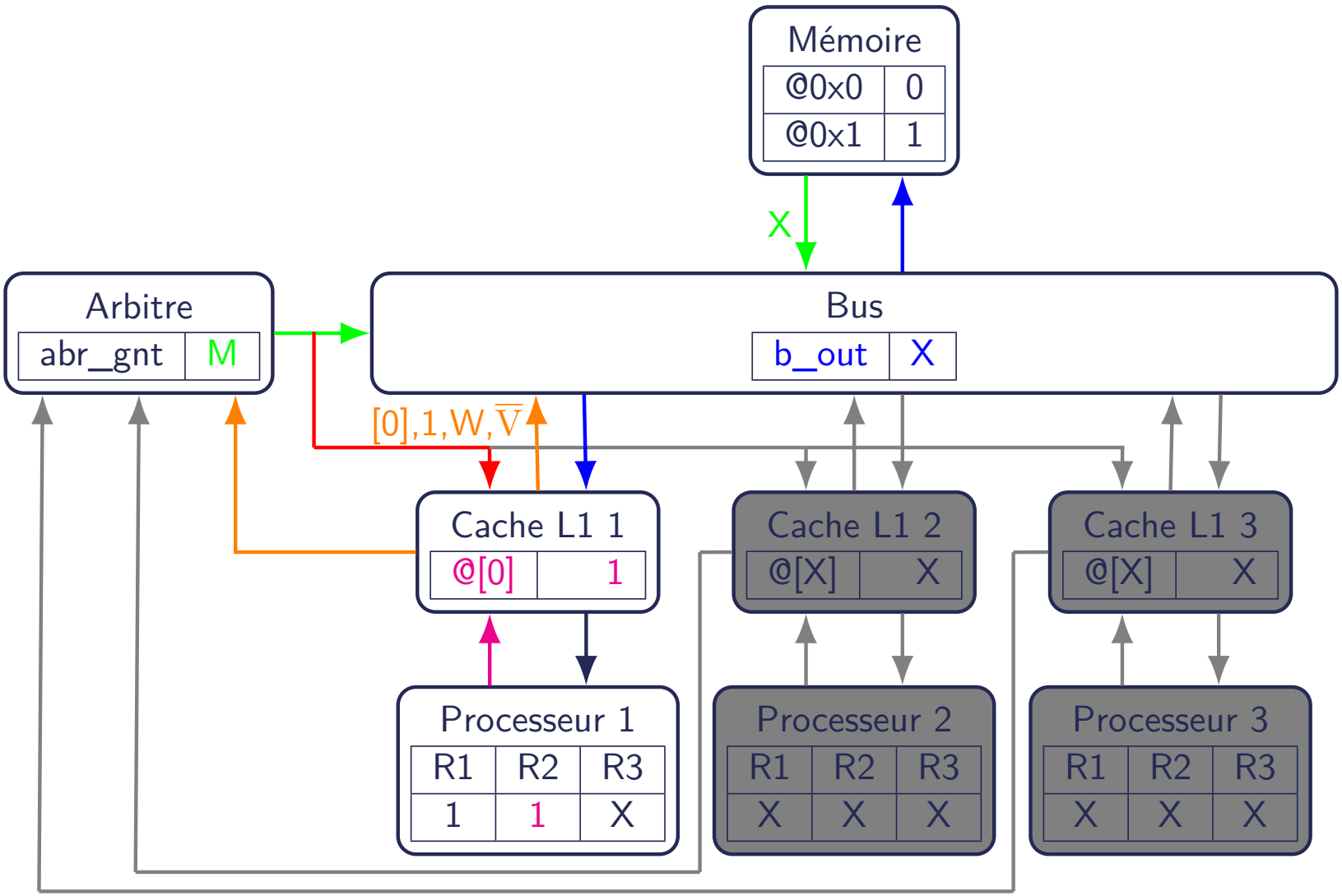
Schema P1a (mono-processeur) : ld R2, [0]



Schema P1a (mono-processeur) : add R2, R2, 1



Schema P1a (mono-processeur) : st R2, [0] (cycle 0)



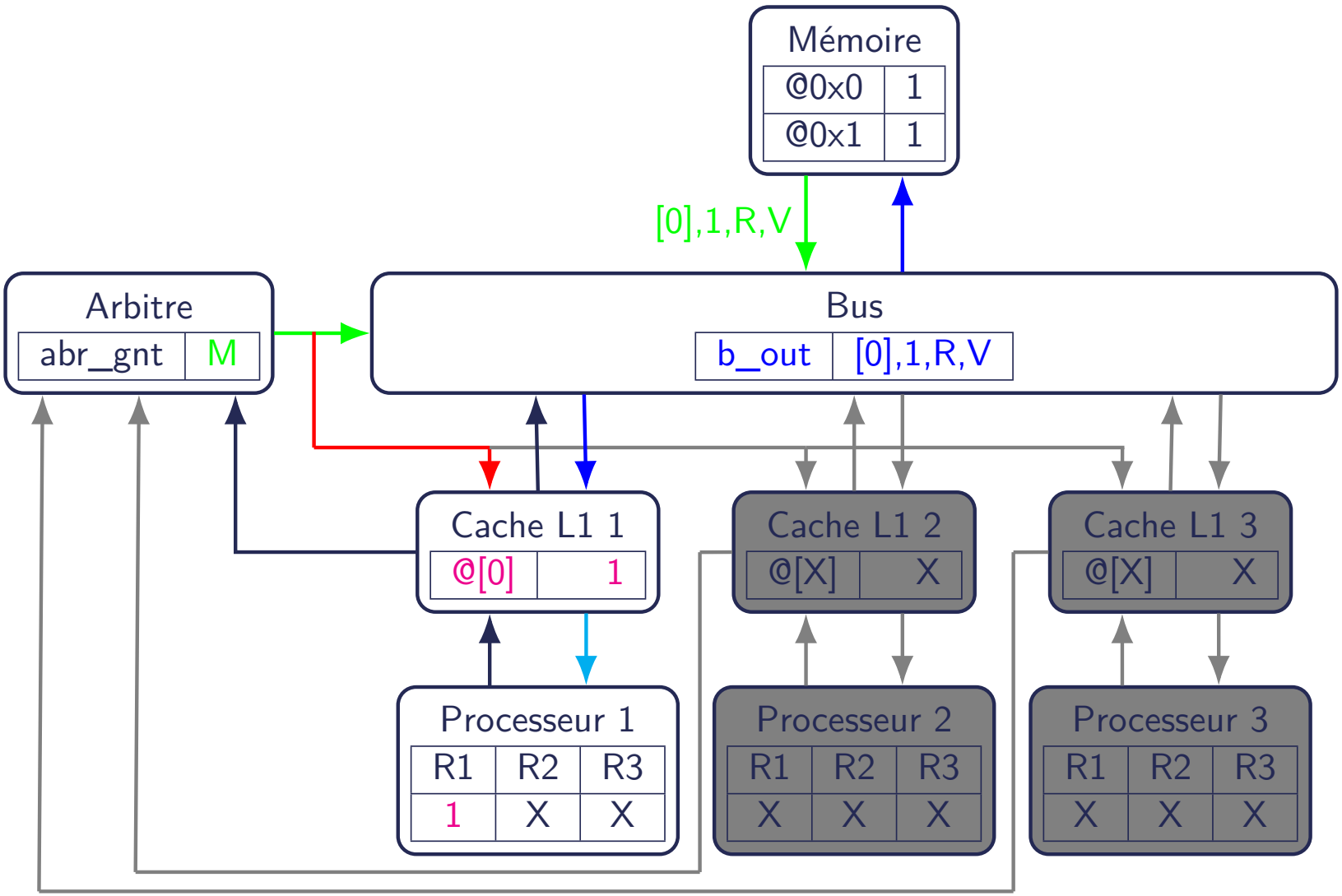
Programme P1b (mono-processeur)

SCIENCES SORBONNE UNIVERSITÉ

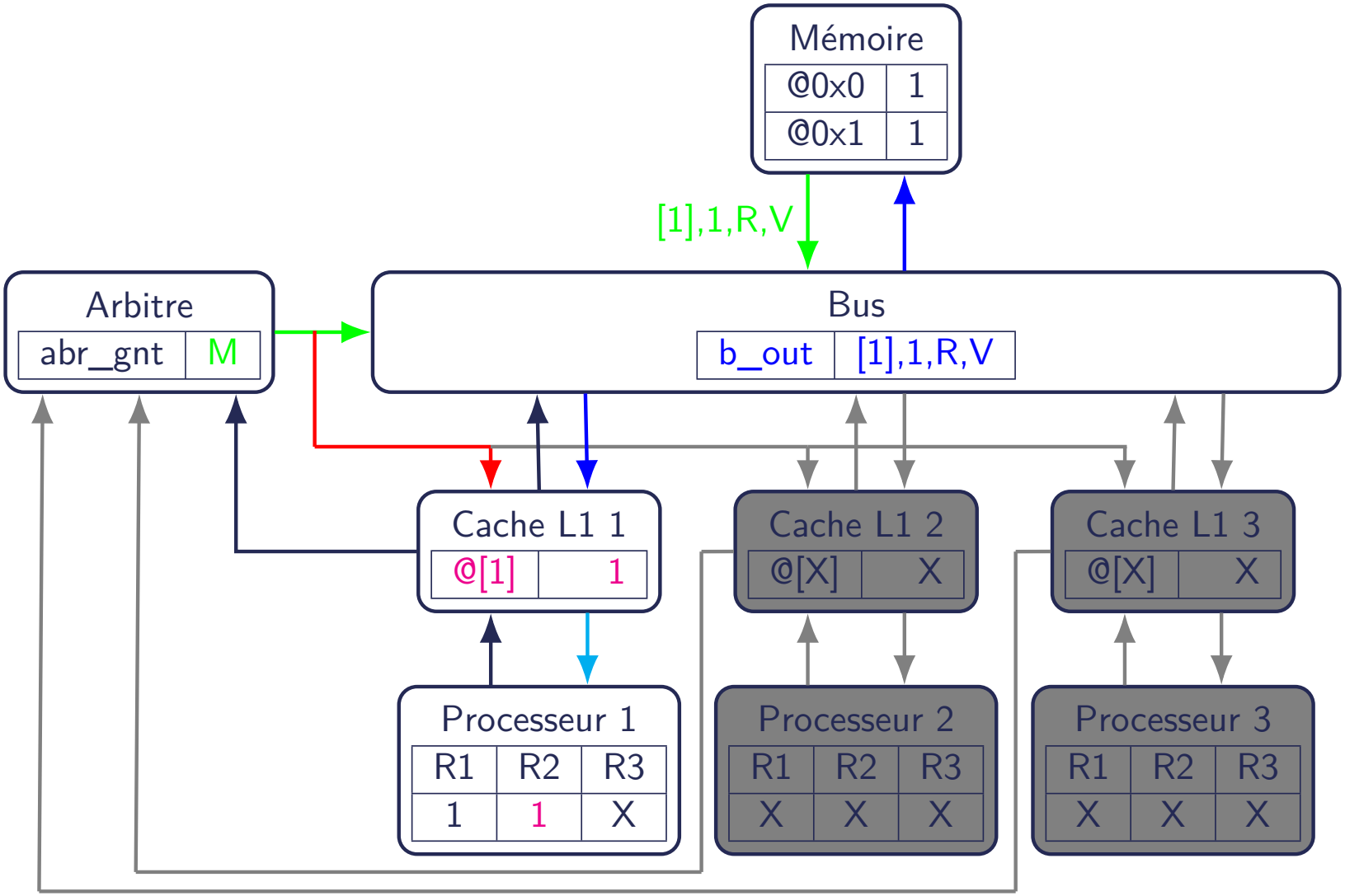
P1b.asm

```
1   ld  R1, [0]
2   ld  R2, [1]
3   add R3, R1, R2
4   st  R3, [0]
```

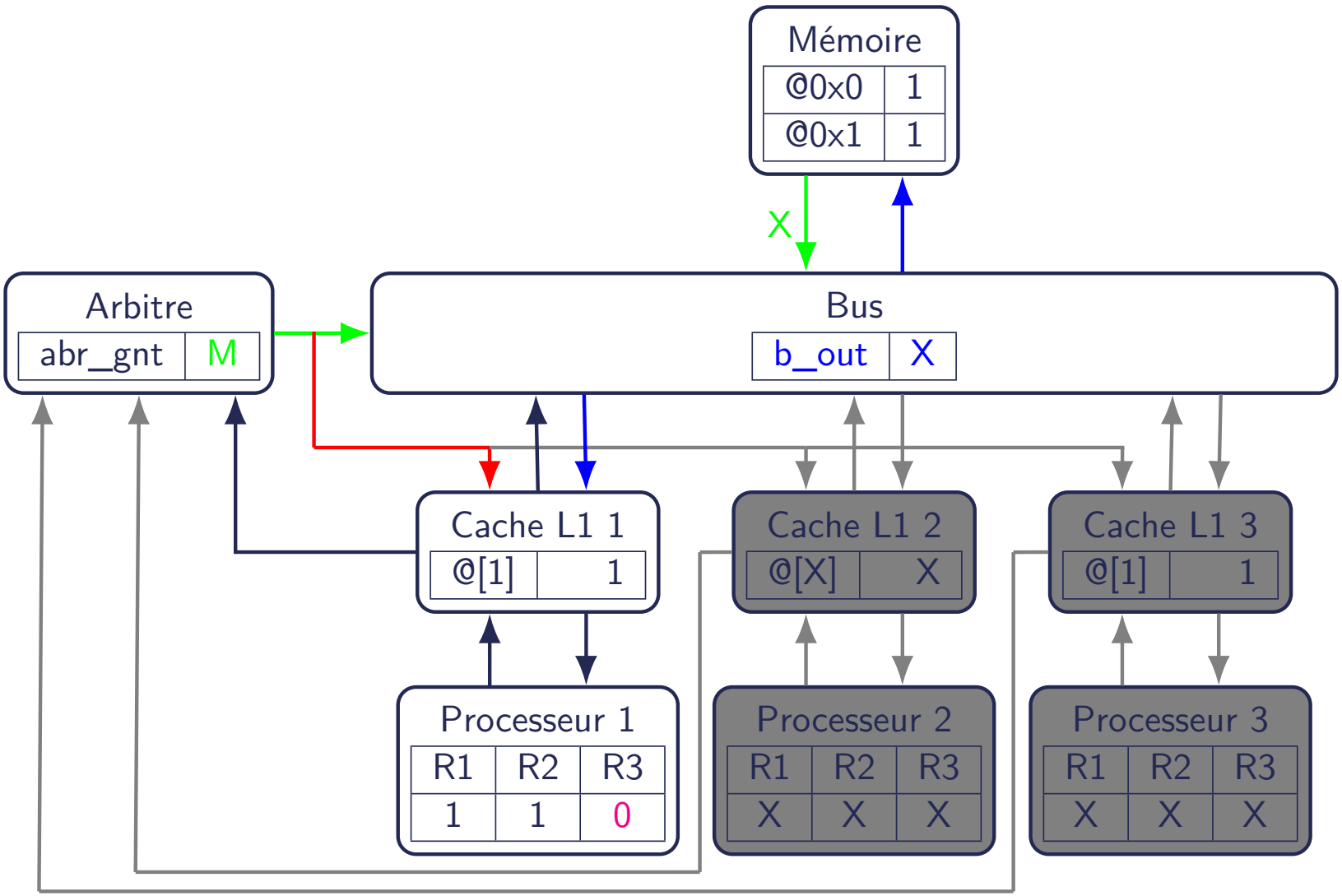
Schema P1b (mono-processeur) : ld R1, [0]



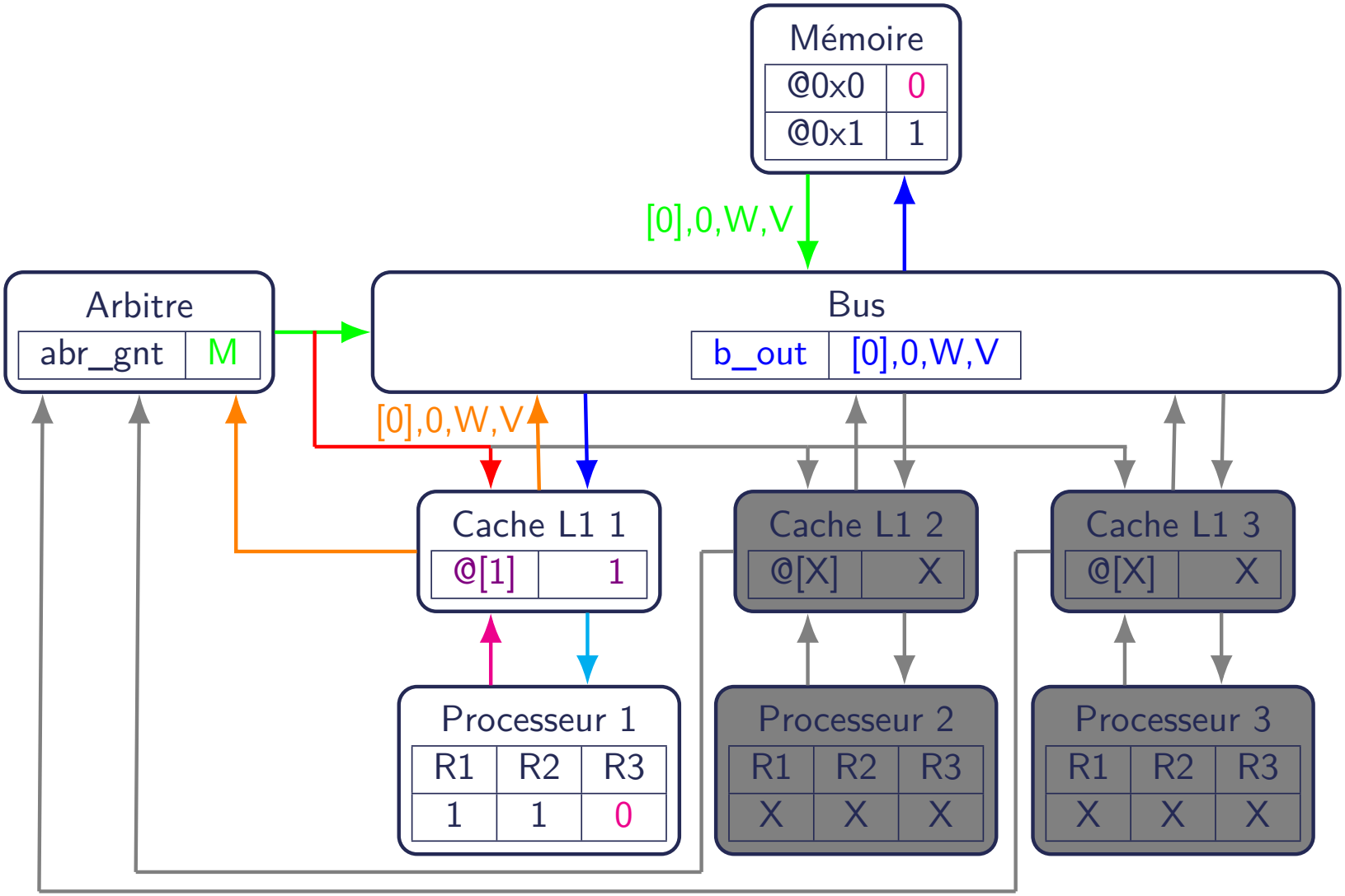
Schema P1b (mono-processeur) : ld R2, [1]



Schema P1b (mono-processeur) : add R3, R1, R2



Schema P1a (mono-processeur) : st R3, [0]



Programme PXa (multi-processeur)

SCIENCES SORBONNE UNIVERSITÉ

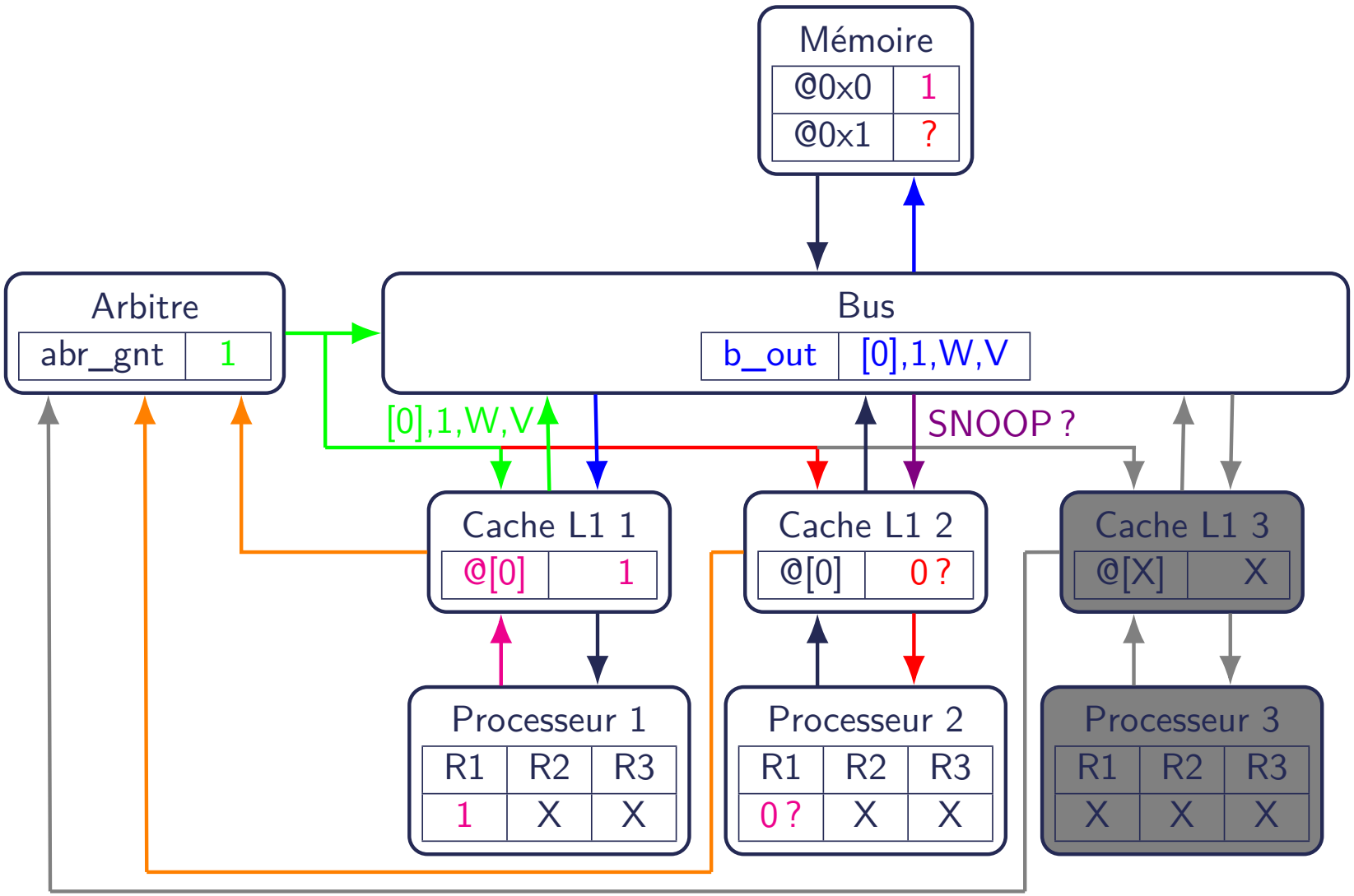
P1a.asm

```
1   ld  R1, [0]
2   add R1, R1, 1
3   st  R1, [0]
4   [...]
5   [...]
```

P2a.asm

```
1   ld  R1, [0]
2   [...]
3   [...]
4   add R2, R1, 1
5   st  R2, [1]
```

Schema PXa (multi-processeur) : le snoop



Programme PXb (multi-processeur)

SCIENCES SORBONNE UNIVERSITÉ

P1b.asm / P2b.asm

```
1  dbt: ld R1, [0]
2  cmp R1,0
3  beq dbt
4  dec R1
5  st R1, [0]
6  <sc>
```

P3b.asm

```
1  [...]
2  ld R1, [1]
3  [...]
```

Programme PXb (multi-processeur)

La limite du SNOOP

Si les deux processus P1b et P2b ont vue la nouvelle valeur de P3b et fait leur comparaison avant de l'écrire ils entreront tout DEUX en section critique en "même temps"

Une solution : Uncachable ?

Même combat...

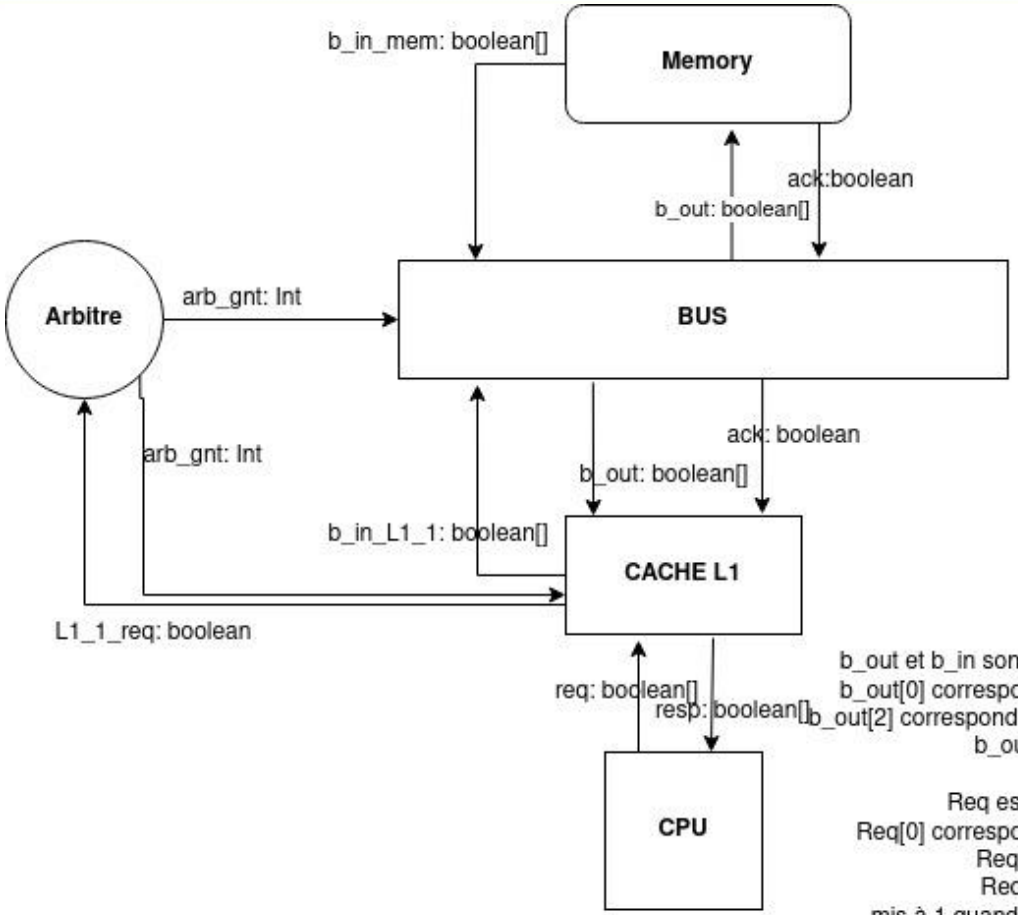
Solution : combiner avec des instructions spécifiques

Qui garantisse l'atomicité :

- Compare And Swap (ne détecte pas la restauration d'ancienne valeur)
- Test and test-and-set
- Load-Link/Store-Conditional, ...

Modélisation des composants

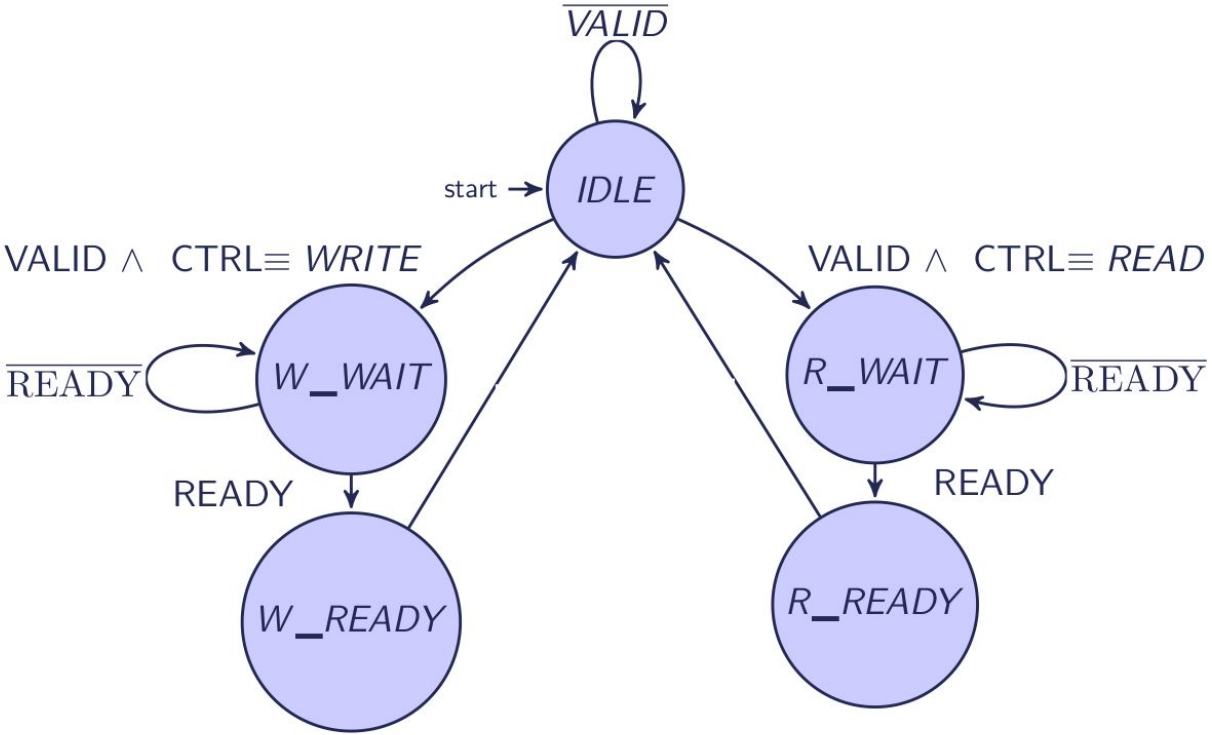
Étape 1: signaux, nature des données



b_out et b_in sont des tableaux de quatres éléments,
b_out[0] correspond à AD, b_out[1] correspond à DT,
b_out[2] correspond à CTRL et b_out[3] correspond à Valid
b_out[4] correspond à ACK

Req est un tableau de 3 valeurs
Req[0] correspond à AD, Req[1] correspond à DT,
Req[2] correspond à CTRL
Req[3] correspond à Valid
mis à 1 quand une requête est faite par le CPU

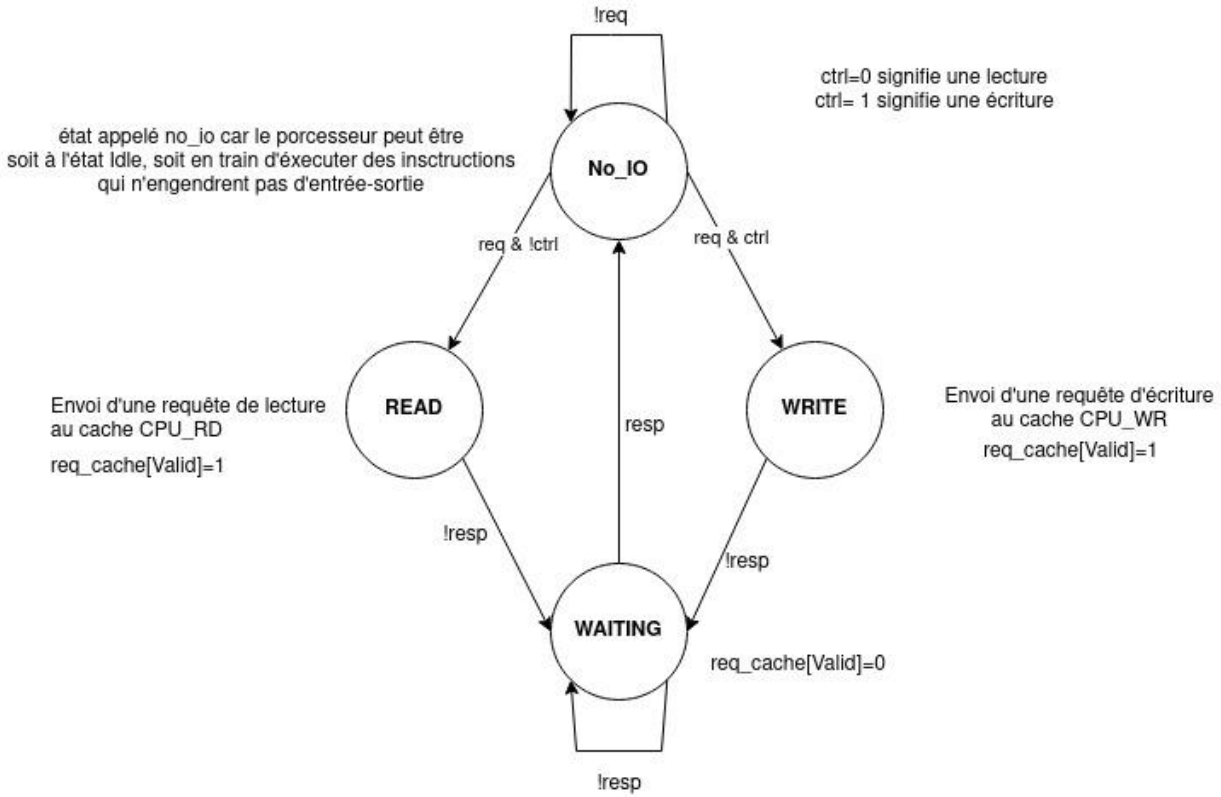
Mémoire


VALID
start
IDLE
VALID ∧ CTRL≡ WRITE
VALID ∧ CTRL≡ READ
READY
W_WAIT
R_WAIT
READY
READY
W_READY
R_READY

```
MODULE memory-device(AD, DT, CTRL, VALID, ACK,b_out)
VAR
   state: { idle, read_wait, write_wait, read_ready, write_ready };
   ready: boolean;
   b_in_mem: array 0..4 of boolean;
   cases: array 0..1 of boolean;

SPEC
-- Vivacité
-- si la mémoire est à l'état read-wait alors il aura un résultat à un moment donné
   AG (memory.state= read_wait -> AF memory.state=read_ready)
-- de même pour write-wait
SPEC
   AG (memory.state= write_wait  -> AF memory.state=write_ready)
```

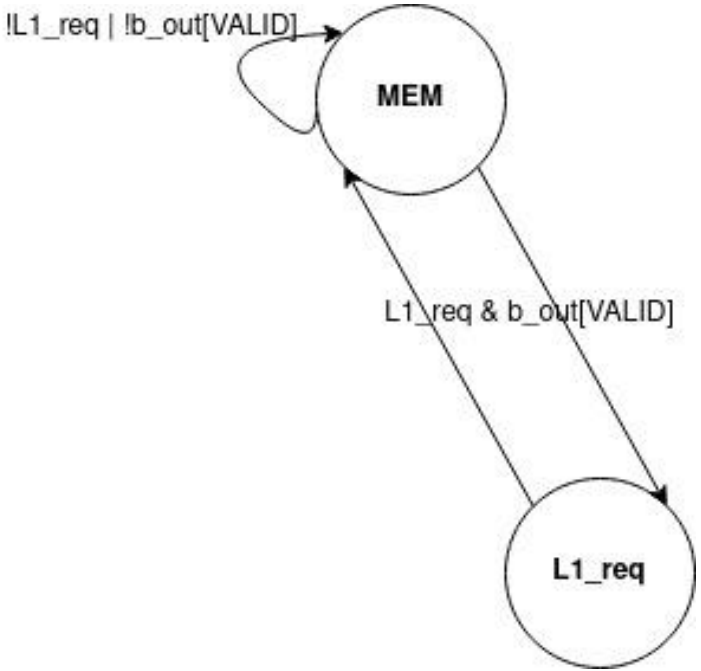
Processeur


!req
ctrl=0 signifie une lecture
ctrl= 1 signifie une écriture
état appelé no_io car le porcesseur peut être
soit à l'état Idle, soit en train d'éxecuter des insctructions
qui n'engendrent pas d'entrée-sortie
No_IO
req & !ctrl
req & ctrl
READ
WRITE
Envoi d'une requête de lecture
au cache CPU_RD
req_cache[Valid]=1
resp
Envoi d'une requête d'écriture
au cache CPU_WR
req_cache[Valid]=1
!resp
!resp
WAITING
req_cache[Valid]=0
!resp

```
MODULE processor-device(AD, DT, CTRL, VALID,req, resp)
-- req is provided by the program
-- resp is provided by the cache
VAR
   state: { NO_IO, WR_RD, WAITING};
   req_cache: array 0..3 of boolean;


-- Vivacité: le processeur finit par avoir une réponse
SPEC
   AG( processor.state=WR_RD -> AF processor.state=NO_IO)
```

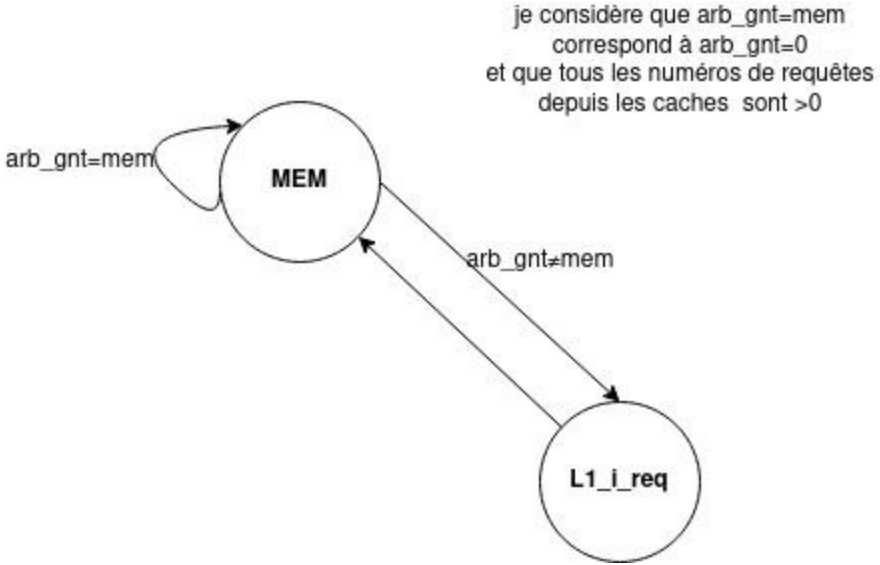
Arbitre



```
MODULE arbitre-device(VALID,L1_req,b_out )
VAR
   arb_gnt: {0,1};
-- Vivacité
-- Quelque chose de bon finira par arriver
-- Si la requête L1_req est positionné, elle finira par avoir l'arbitre
SPEC
   AG ( L1_req -> AF arbitre.arb_gnt=1)

-- Sureté-- On ne peut pas avoir l'arbitre attribué à deux composants au même temps
SPEC
   AG !( arbitre.arb_gnt=1 & arbitre.arb_gnt=0)
-- Equité
SPEC
   -- il y a une exécution tel que le bus est attribué à la mémoire infirniment souvent
   EG EF(arbitre.arb_gnt =0)
-- de même pour le cache
SPEC
   EG EF(arbitre.arb_gnt =0)
```

Bus


je considère que arb_gnt=mem
correspond à arb_gnt=0
et que tous les numéros de requêtes
depuis les caches sont >0
arb_gnt=mem
MEM
arb_gnt≠mem
L1_i_req

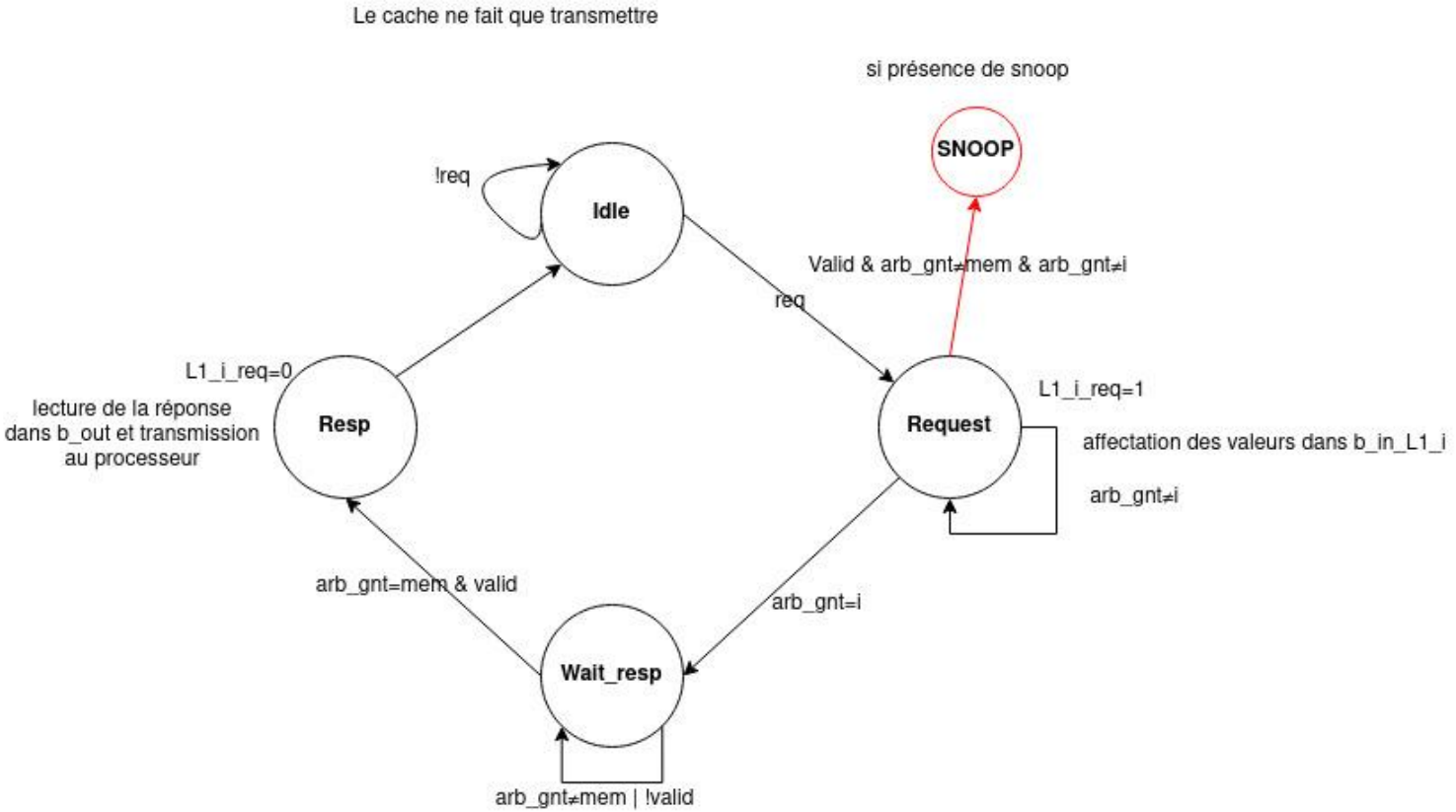
```
MODULE bus-device(AD,DT, CTRL, VALID, ACK,ARB_GNT, b_in_L1_1, b_in_mem)
VAR
   state: { MEM, CACHE};
   b_out: array 0..4 of boolean;
-- quand il n'y a pas de requête posée, il est toujours attribué à la mémoire
SPEC
   AG ((ARB_GNT =0) -> AX(bus.state=MEM))

-- s'il y a une requête, le cache fini par obtenir le bus
SPEC
   AG (ARB_GNT=1 & ! (bus.state=CACHE) -> AX(bus.state)=CACHE)

-- Après l'obtention du bus par une requête, la mémoire prend la main
SPEC
   AG (bus.state=CACHE -> AX(bus.state=MEM))

-- Le bus n'est pas attribué à deux composants au même temps
SPEC
   AG !( ARB_GNT= 0 & ARB_GNT=1)
```

Cache


Le cache ne fait que transmettre
si présence de snoop
SNOOP
!req
Idle
Valid & arb_gnt≠mem & arb_gnt≠i
req
L1_i_req=0
lecture de la réponse
dans b_out et transmission
au processeur
Resp
Request
L1_i_req=1
affectation des valeurs dans b_in_L1_i
arb_gnt≠i
arb_gnt=mem & valid
arb_gnt=i
Wait_resp
arb_gnt≠mem | !valid

```
MODULE cache-device(AD, DT, CTRL, VALID, req, ARB_GNT, b_out)
VAR
   state: { idle, request, waiting, response};
   L1_req: boolean;
   ID: {1};
   b_in_L1_1: array 0..4 of boolean;
resp: array 0..3 of boolean

SPEC
   AG ( cache.ID=1)
-- Vivacité
SPEC
  -- si une requête est posé par un cache, alors il aura forcément une réponse
 AG ( cache.L1_req -> AF (cache.state=response))
-- Sureté
SPEC
   -- impossible d'avoir le signal L1_req levé et le cache à l'état idle
   AG !( cache.L1_req & cache.state=idle )
SPEC
   -- quand une requête a reçu sa réponse, elle positionne le signal à 0à condition que l
   -- arb_gnt=0 correspond à attribution du bus à la mémoire
   AG ( arb_gnt=0 & b_out[VALID] & cache.L1_req -> AF !cache.L1_req)
```

2.3 plateforme mono-processeur

```
MODULE main
VAR
        -- intern variable activated when an
executing program
    -- encounters a read or write instruction
    req:  array 0..3 of boolean;
    -- activated when the cache has an answer for
the processor
    processor: processor-device(AD, DT, CTRL,
VALID,req, cache.resp);
    cache: cache-device(AD, DT, CTRL, VALID, ACK,
req, arbitre.arb_gnt, bus.b_out);
    bus: bus-device(AD,DT, CTRL, VALID,
ACK,arbitre.arb_gnt, cache.b_in_L1_1,
memory.b_in_mem);
    memory: memory-device(AD, DT, CTRL, VALID,
ACK,bus.b_out);
    arbitre:
arbitre-device(VALID,cache.L1_req,bus.b_out );
```

Spécifications:

- toutes les propriétés précédentes
- propriétés de sûreté
- garantie qu'une requête d'écriture ou lecture du processeur est validée par la mémoire

Points d'amélioration

Automate de la mémoire

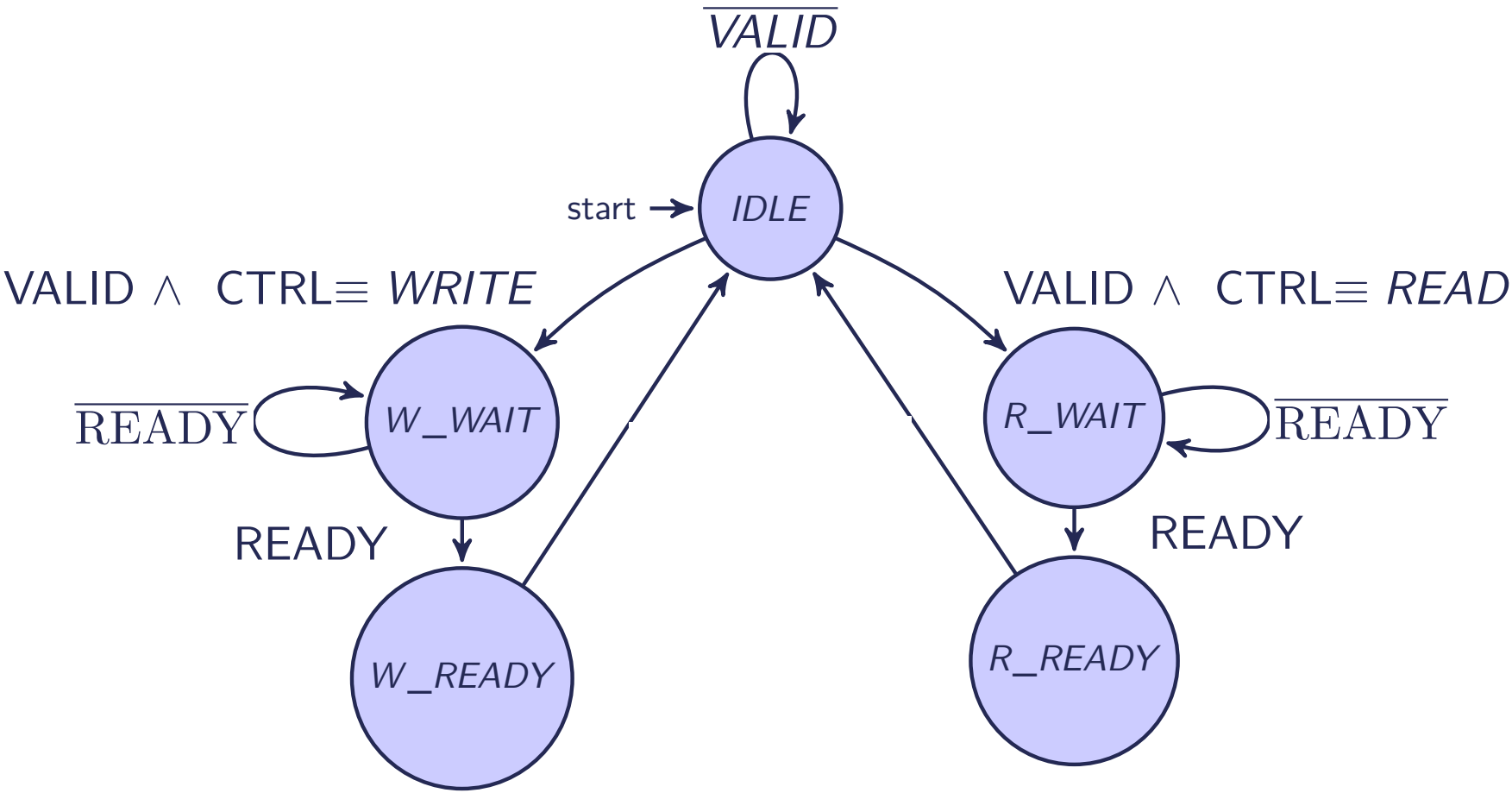
Interface :

- 1 signal d'entrée : b_out avec (AD, DT, CTRL, VALID)
- 1 signal interne : READY
- 1 signal de sortie : b_in_mem avec (AD, DT, CTRL, VALID)

Etats :

- IDLE : attente de la réception d'une commande de lecture ou d'écriture
- W_WAIT :
- R_WAIT :

Automate de la mémoire



Mémoire : écriture

	AD_in	DT_in	CTRL_in	VALID_in
IDLE	X	X	X	0
W_WAIT	X	X	X	0
W_READY	AD_out	DT_out	CTRL_out	VALID

Mémoire : lecture

	AD_in	DT_in	CTRL_in	VALID_in
IDLE	X	X	X	0
R_WAIT	X	X	X	0
R_READY	AD_out	DT[AD]	CTRL_out	VALID

Automate de la mémoire

Particularités

- pas de gestion des erreurs
- Quand l'automate repasse à l'état IDLE, il doit set son signal VALID sur b_in_mem à 0
- On pourrait rajouter des états intermédiaires sur après W_WAIT et R_WAIT, respectivement W_READY et R_READY, qui seraient atteint après un lapse de temps via un signal interne READY pour simuler la latence de la mémoire.
- Par conséquent plus besoin d'attendre VALID mais sur READY, sachant que la réponse doit se faire en un cycle.

Programme memory.smv

memory.smv

```
1  Insérer le code
```

Automate du processeur

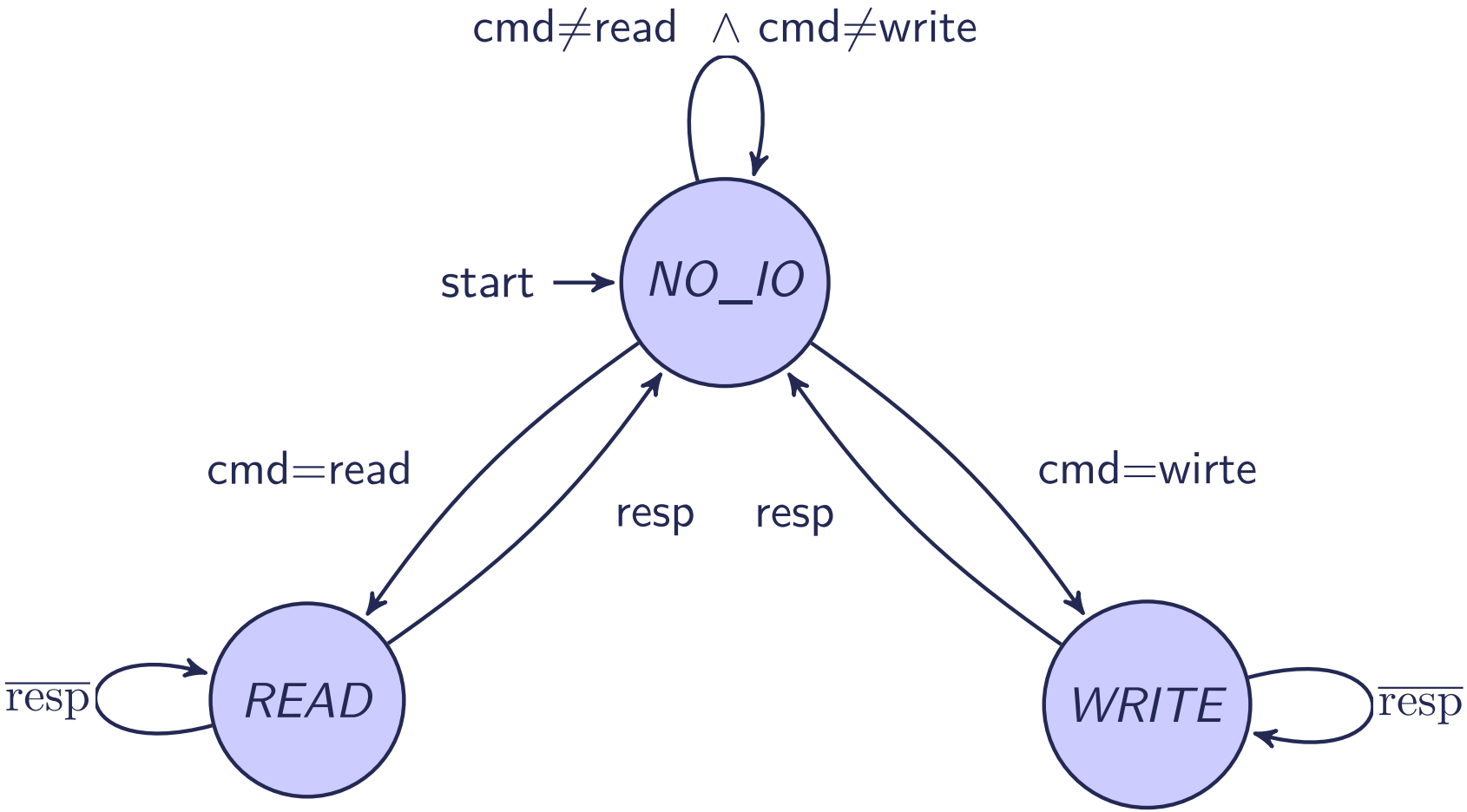
Interface :

- 1 signal d'entrée : resp
- 3 signaux internes : add, data, cmd
- 3 signaux de sortie : AD, DT, CTRL

Etats :

- NO_IO : pas de requête d'écriture ou de lecture
- READ : requête de lecture et attente de la réponse du cache
- WRITE : requête d'écriture et attente de la réponse du cache

Automate du processeur



processeur : écriture et lecture

	AD	DT	CTRL
NO_IO	X	X	cmd≠read $\wedge$ cmd≠write
WRITE	add	data	cmd

	AD	DT	CTRL
NO_IO	X	X	cmd≠read $\wedge$ cmd≠write
READ	add	X	cmd

Programme bus.smv

bus.smv

```
1  Insérer le code
```

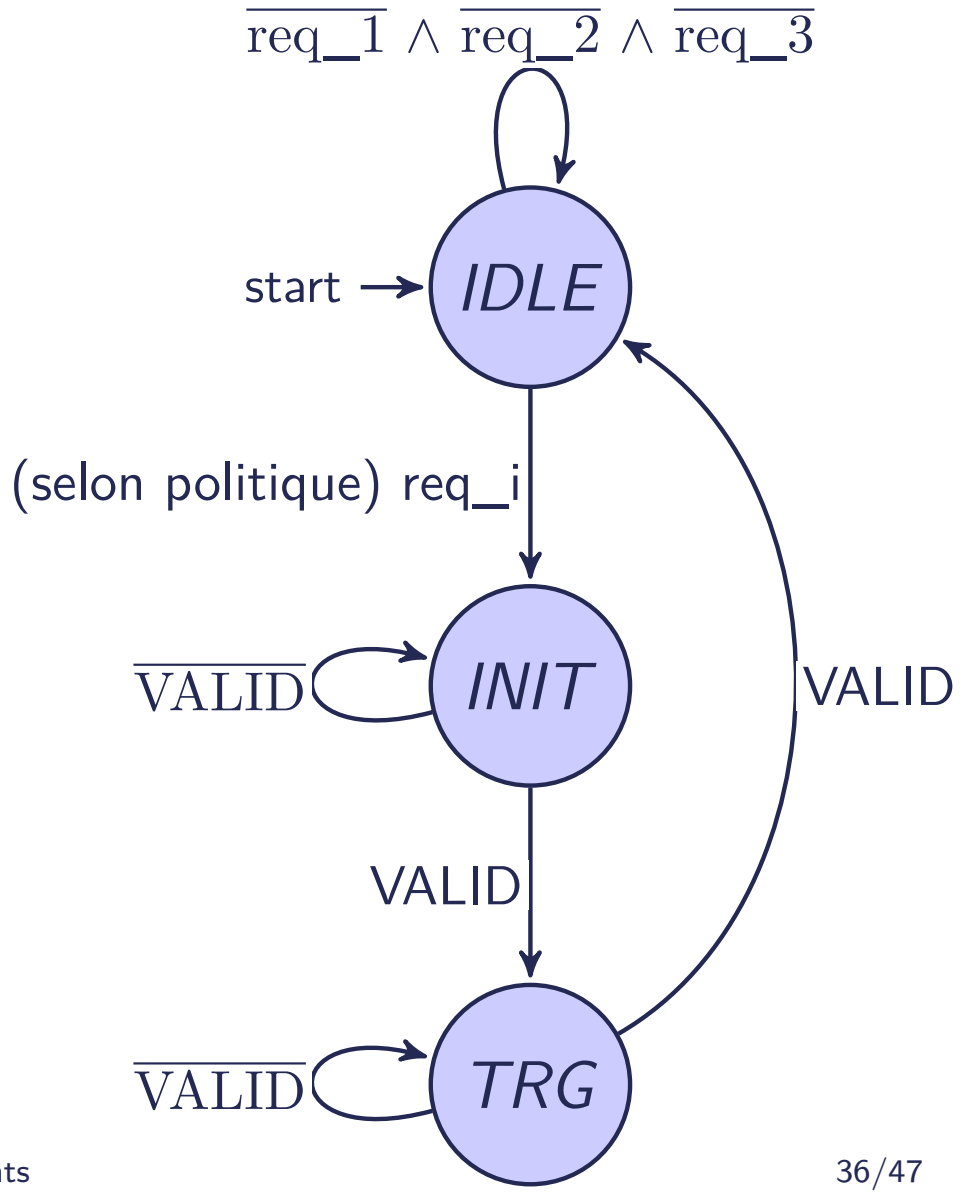
Automate de l'arbitre

Interface :

- 2 signaux d'entrée : req, VALID
- 1 signal de sortie : abr_gnt

Etats :

- IDLE : aucune transaction, et l'arbitre attend au moins la demande d'un initiateur
- INIT (INITIATOR) : le bus a été alloué à un initiateur, et envoie de la commande
- TRG (TARGET) : le bus a été alloué à une cible, et attend la réponse VALID de la commande



Arbitre, exemple avec politique de priorité sur CPU_X

SCIENCES SORBONNE UNIVERSITÉ

Politique par priorité

Le bus est alloué selon la priorité des initiateur demandant, ici le numéro de CPU le plus faible, ici req_2, req_3 sont levés puis req_1 pendant la première transaction

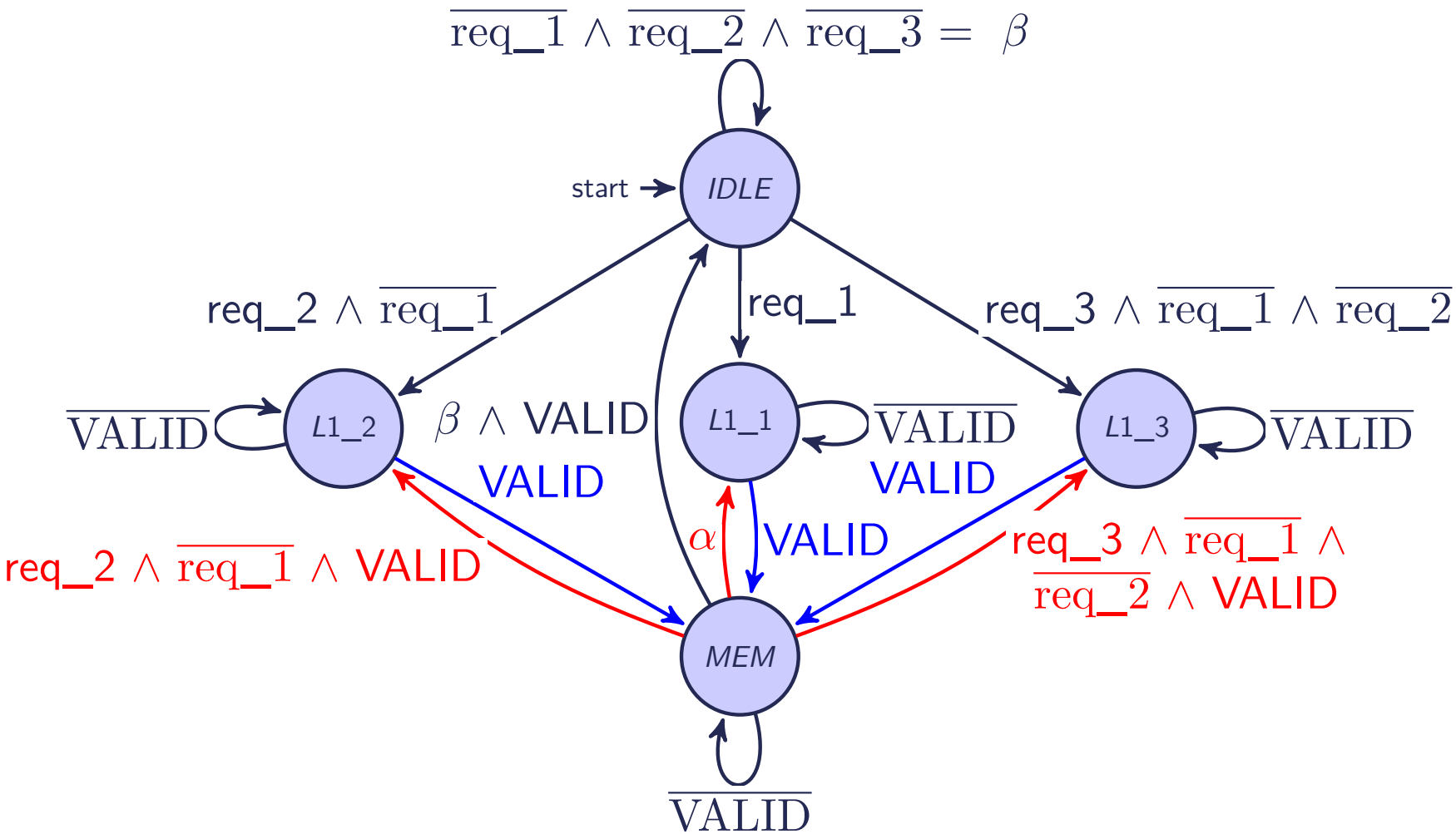
	abr_gnt
IDLE	M
INIT	req_2
TRG	M

	abr_gnt
IDLE	M
INIT	req_1
TRG	M

Avantage et Inconvénient

unfair, risque de famine, gestion des priorités sur processeur piné

Automate de l'arbitre avec priorité



$\alpha = \text{req\_1} \wedge \text{VALID}$

Arbitre, exemple avec round robin

Politique par round robin

Le bus est alloué selon un tour cyclique des initiateur demandant si plusieurs en conflit, ici req_1 vient d'avoir le bus, req_2, req_3 sont levés puis req_1 pendant la première transaction

	abr_gnt
IDLE	M
INIT	req_2
TRG	M

	abr_gnt
IDLE	M
INIT	req_3
TRG	M

Avantage et Inconvénient

plutôt équitable, moins de risque de famine, pas de priorité propre

D'autre politique

pseudoLRU, LFU, RR, ..., préallocation processeur, ...

Programme arbiter.smv

arbitrer.smv

```
1  Insérer le code
```

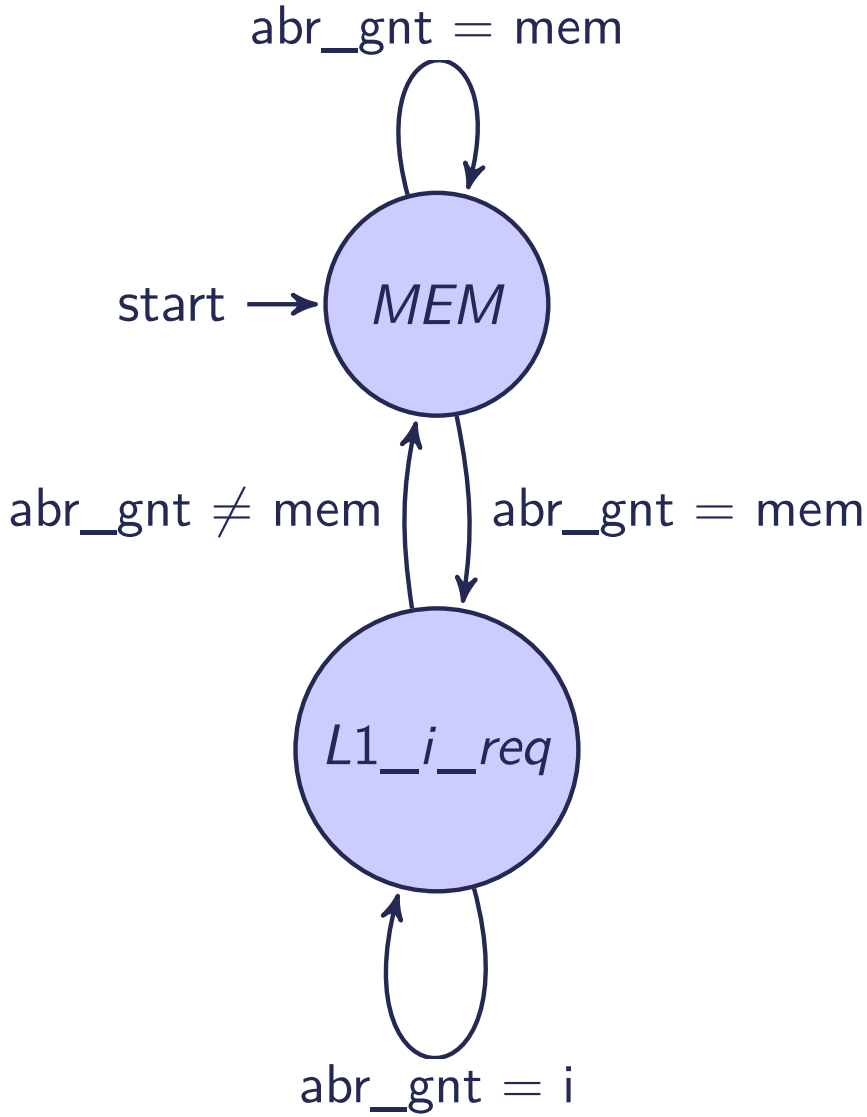
Automate du bus

Interface :

- 5 signaux d'entrée : abr_gnt, b_in_1, b_in_2, b_in_3, b_in_mem
- 1 signal de sortie : b_out

Etats :

- MEM : soit le bus est en attente d'une commande, soit il est positionné pour la réponse de la mémoire
- L1_i_req (INITIATOR) : le bus a été alloué à un initiateur L1_i_req, et envoie de la commande



Bus : écriture

	b_out
MEM	b_in_mem
L1_i_req	b_in_i

Programme bus.smv

bus.smv

```
1  Insérer le code
```

Automate du Cache L1

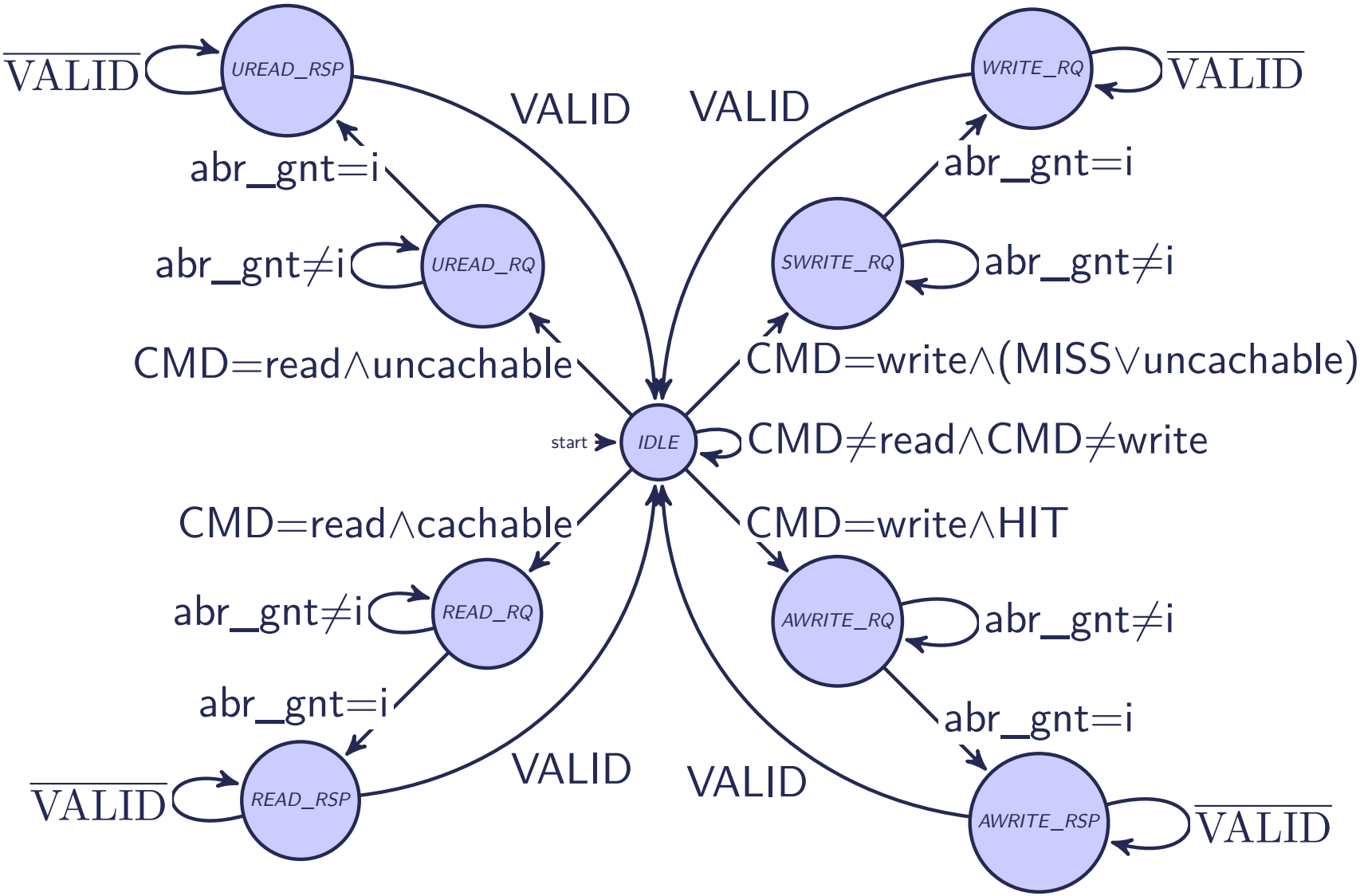
Interface :

- 3 signaux d'entrée : ADD, DATA, CMD, abr_gnt, b_out (AD, DT, CTRL, VALID)
- 3 signaux de sortie : resp, req_i, b_in_L1_i (AD, DT, CTRL, VALID)

Etats :

- IDLE : pas de requête d'écriture ou de lecture de la part du processeur
- READ_RQ : requête de lecture et attente d'accès du bus
- UREAD_RQ : requête de lecture d'une donnée non cachable et attente d'accès du bus
- AWRITE_RQ : requête d'écriture asynchrone (HIT) et attente de la réponse du cache
- SWRITE_RQ : requête d'écriture synchrone (MISS) et attente de la réponse du cache

Automate du Cache L1



Programme arbitre.smv

arbitre.smv

```
1  Insérer le code
```

Conclusion

- Conclusion
- Merci de votre attention
- Des questions ?