Réseaux d'interconnexions pour systèmes multiprocesseurs fortement couplés : évolution, challenges

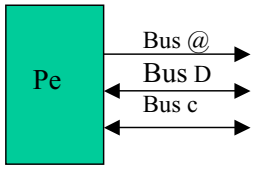
Daniel Litaize
IRIT

PLAN

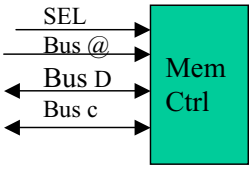
- Définition de l'espace de réalisation des multiprocesseurs fortement couplés
- Les problèmes incontournables : horloge, arbitrage
- Comment améliorer la performance
- En présence de mémoires caches
- Alternatives

Multiprocesseur: les unités fonctionnelles à interconnecter

Maître: Processeur, sans cache(s), sans MMU, sans ressources (mémoire,E/S)

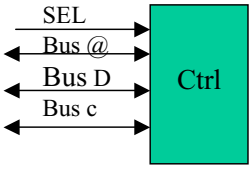


Esclave : mémoire, contrôleur d'E/S



Maître et esclave:

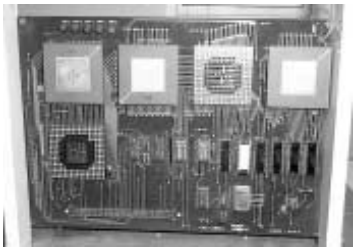
DMA

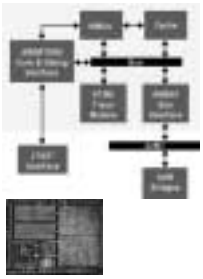


ASSEMBLAGE

2 dimensions ou 3 dimensions:

U.F. : Circuit intégré bloc IP carte





Carte puce chassis

Chassis/carte/puce multiprocesseurs

Chassis « ouvert » => standard de bus : VME, PCI : grande souplesse d'utilisation mais performance limitée et évolution lente du fait de la compatibilité ascendante.
« verrues » pour une augmentation locale de la performance : connecteurs-sabots inter-cartes, bus inter-cartes,..
Domaine d'application : industrie temps réel.

Chassis « fermé » => machine constructeur : Enterprise 10000, .. : pas évolutif, meilleure performance, durée de vie limitée, car pas de compatibilité ascendante.
Domaine d'application : Multiprocesseurs dédiés calcul scientifique, serveurs.

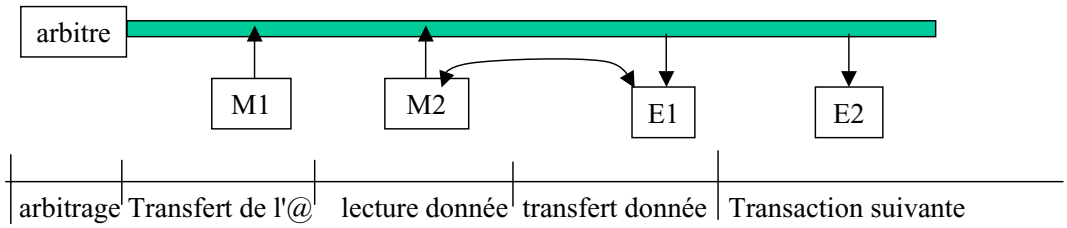
Carte : possibilités limitées par la taille de la carte, non évolutive. Meilleur rapport performance/prix
Domaine d'application : Serveurs bas coûts, PC.

Puce : versatile, mais pour marché de masse.
Domaine d'application : essentiellement traitement du signal.

Points communs : unité fonctionnelles reliées par un réseau d'interconnexion.
Plus simple : bus partagé
Plus performant :crossbar.

Réseau d'interconnexion conceptuellement de même nature dans les trois cas

Analyse d'une transaction mémoire sur un bus partagé



Cas le plus élémentaire : un seul bus "multiplexé", c'est à dire utilisé pour les adresses et les données, sans aucune mémoire intermédiaire : ni cache côté processeur, ni tampon côté mémoire. La largeur du bus est à priori dictée par la taille de l'adresse, égale souvent à la taille de la donnée "standard": 32 bits.

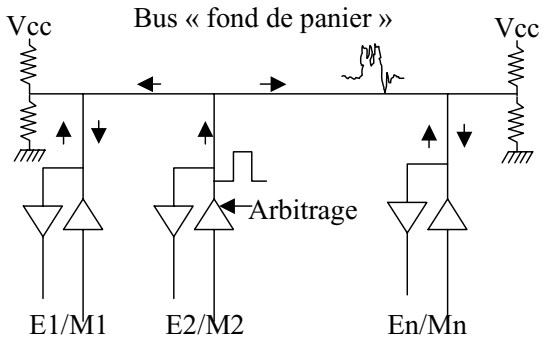
Problèmes génériques posés :
-Technique d'interconnexion : ou câblé, multiplexage des sources sur un bus unique destination,
-Technique de fonctionnement utilisée : synchrone, asynchrone, asynchrone synchronisé,
-Technique d'arbitrage d'accès à la ressource partagée, le bus : parallèle ou série,
-Technique de sélection de l'unité adressée : centralisée, distribuée.

Fonctionnement :
Lecture "mot" : arbitrage, transfert de l'adresse mot demandé, attente lecture mémoire, transfert mot lu.
Ecriture "mot" : arbitrage, transfert de l'adresse mot demandé, transfert mot à écrire, puis écriture (la phase d'écriture peut en partie se recouvrir avec la phase de transfert du mot).

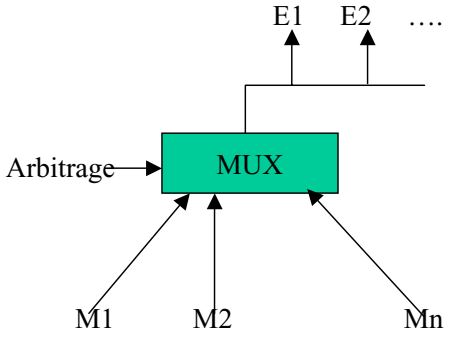
Evolutions :
Comment améliorer la performance intrinsèque du bus par duplication et spécialisation et/ou adapter le bus aux mécanismes d'amélioration locale de la performance : pipe-line, recouvrement, anticipation,….

Interconnexion 2D/3D, version bus partagé

Version 3D : « OU câblé »
A priori pour systèmes "ouverts"



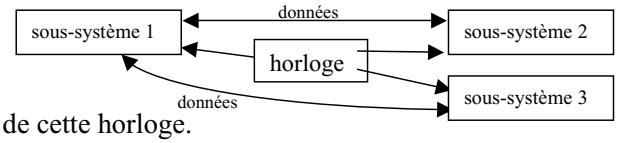
Version 2D : OU câblé ou multiplexage
A priori pour systèmes "fermés"



E=Esclave, M=Maître, / désigne ET OU

Nécessité d'un « settling time », diminué par:
-drivers avec signaux trapézoïdaux,
-excursion tension plus faible
- Low Voltage differential Signal,..

Synchrone, asynchrone, asynchrone synchronisé



Synchrone : qu'est ce qui est synchrone ?
Une seule horloge distribuée
à tout le système =>
toutes les communications sont synchrones de cette horloge.
Avec l'augmentation de la fréquence se pose le problème de la mise en phase de l'horloge.

Certaines parties ont un fonctionnement synchrone :

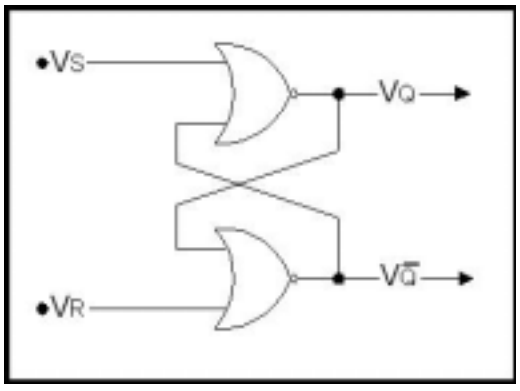
- le bus partagé : les requêtes se font relativement à une horloge globale associée au bus. Revient à dire que l'arbitrage est synchronisé.
- les requêtes d'accès au bus sont asynchrones, elles peuvent être resynchronisées au niveau de l'arbitre, mais l'arbitre peut aussi être totalement asynchrone.

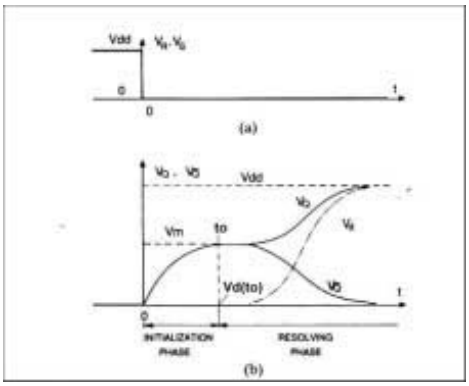
Asynchronisme total via la logique asynchrone ?

Solutions qui se dessinent :

Sous ensembles fonctionnels en logique synchrone d'une horloge locale ou en logique asynchrone, interconnectés par des protocoles asynchrones de type handshake ou des protocoles synchrones d'une horloge globale. Toute resynchronisation entraîne un risque de métastabilité

Métastabilité (1)





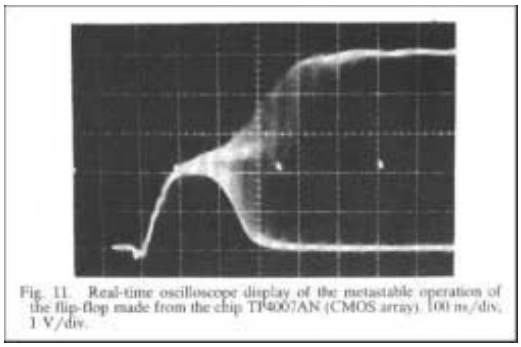


Fig. 11. Real-time oscilloscope display of the metastable operation of the flip-flop made from the chip TP4007AN (CMOS array). 100 ns/div, 1 V/div.

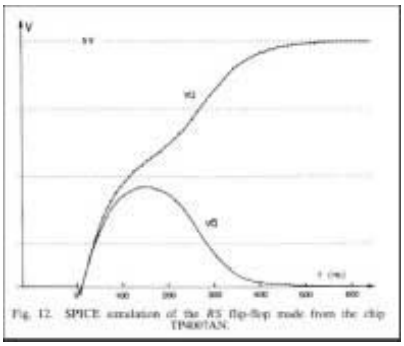
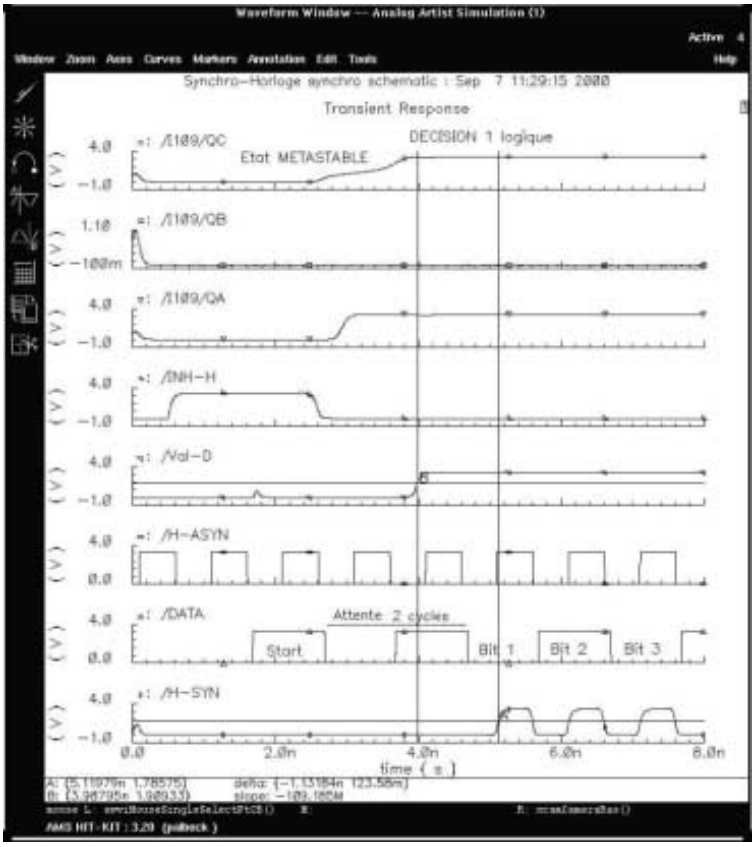


Fig. 12. SPICE simulation of the *RS* flip-flop made from the chip TP4007AN.

T. KACPRZAK and A. ALBICKI « Analysis of Metastable Operation in *RS* CMOS Flip-Flops *IEEE Journal of Solid-State Circuits, Feb. 1987, pp. 57-64.*

Métastabilité (2)



Le fonctionnement correct dépend du temps alloué à la bascule pour résoudre son état métastable. En augmentant ce temps, on minimise la probabilité de non résolution de l'état métastable mais cette probabilité ne peut être nulle.

Afin d'évaluer la fiabilité du circuit, il faut calculer le temps moyen entre deux fautes MTBF, qui dépend des caractéristiques physiques de la bascule et du temps alloué pour la résolution de la métastabilité.

Constat : la probabilité de métastabilité diminue avec l'augmentation de la fréquence de fonctionnement de la technologie.

Techniques d'arbitrage de bus

Trois techniques de base:

1. Arbitrage parallèle. Arbitre centralisé avec deux lignes de liaison par unité requérant le bus : Bus ReQest, Bus GranT. Autorise diverses politiques d'accès : priorité fixe, tournante, préemption,…Bus avec signaux spécialisés. Synchrone ou asynchrone.
2. Arbitrage série ("daisy-chain"). Politique d'accès fixe. Bus avec signaux banalisés. Synchrone ou asynchrone
3. Arbitrage par "éliminations successives". Arbitrage distribué. Bus avec signaux banalisés. Asynchrone

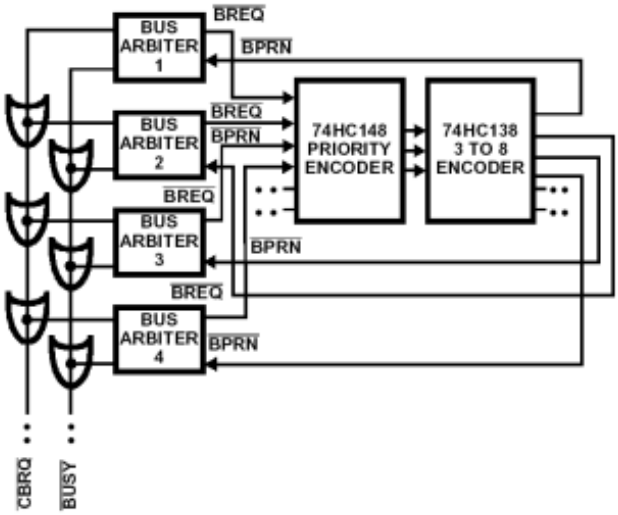
La notion de synchrone/asynchrone doit être précisée : qu'est ce qui est synchrone/asynchrone ?

- peut-on soumettre sa requête quand on veut ou doit-on attendre un signal de l'arbitre: est-on alors asynchrone ou synchrone par rapport à l'arbitre ?
- L'arbitre peut être synchrone ou asynchrone : il échantillonne les demandes ou fonctionne "en continu"

Le moment de l'arbitrage doit-être précisé :

- L'arbitrage est-il réalisé en temps masqué pendant l'occupation du bus, ou au moment de la libération du bus ?

Arbitrage parallèle synchrone d'une horloge commune



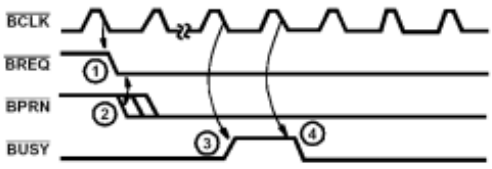


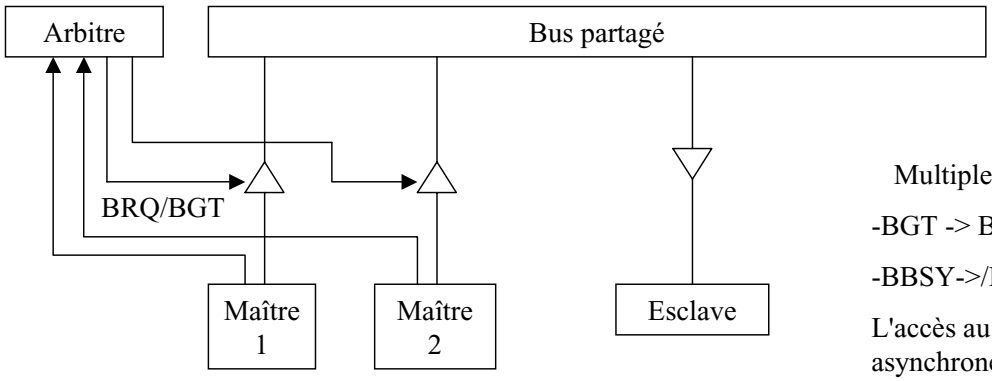
FIGURE 2. HIGHER PRIORITY ARBITER OBTAINING THE BUS FROM A LOWER PRIORITY ARBITER

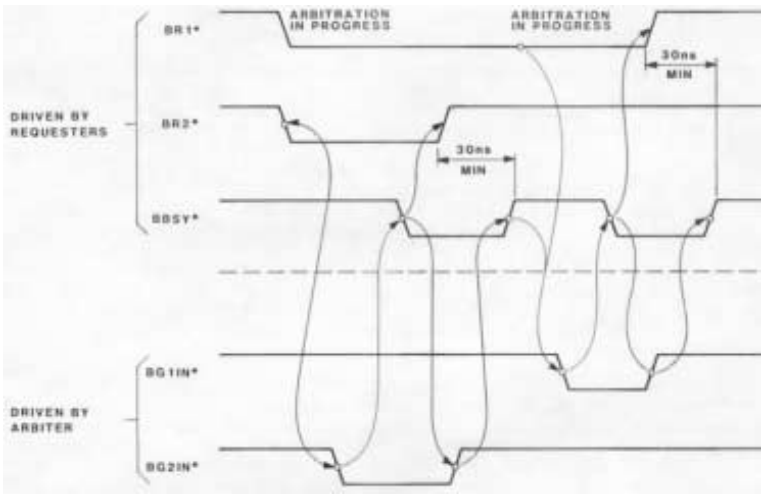
NOTES:

1. Higher priority bus arbiter requests the Multi-Master system bus.
2. Attains priority.
3. Lower priority bus arbiter releases $\overline{\text{BUSY}}$.
4. Higher priority bus arbiter then acquires the bus and pulls $\overline{\text{BUSY}}$ down.

L'arbitre est dit distribué : chaque processeur a son arbitre.Ce circuit gère en réalité les demandes du processeur. Le véritable arbitre est le couple encodeur/décodeur.
Les opérations sont synchrone d'une horloge disponible sur le bus.
Un processeur qui prend le contrôle du bus peut le garder aussi longtemps qu'il le souhaite.

Arbitrage parallèle centralisé





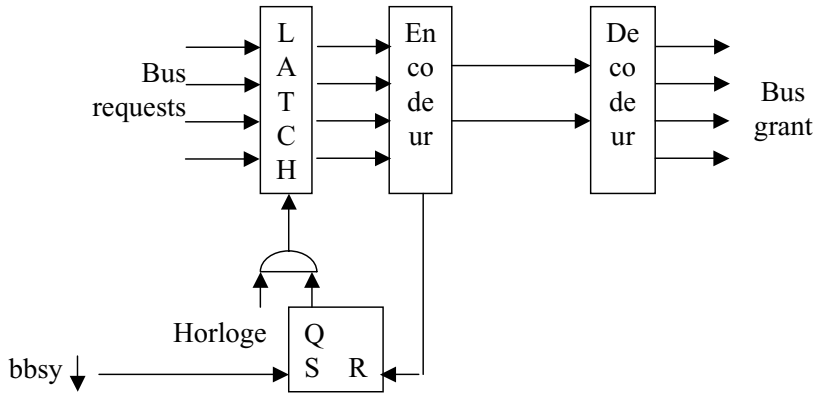
Multiples handshakes:

-BGT -> BBSY

-BBSY->/BRQT et/BGT

L'accès au bus est asynchrone, mais rien n'est précisé pour la structure de l'arbitre.

Arbitrage parallèle centralisé synchrone (principe)



Arbitrage parallèle centralisé asynchrone

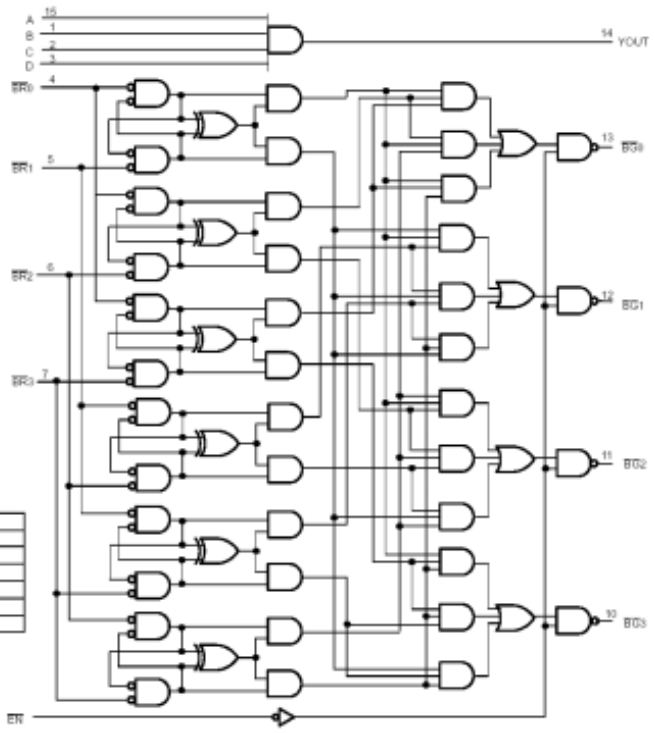
74F786 FUNCTIONAL DESCRIPTION

The BRn inputs have no inherent priority. The arbiter assigns priority to the incoming requests as they are received, therefore, the first BR asserted will have the highest priority. When a bus request is received its corresponding bus grant becomes active, provided that EN is low. If additional bus requests are made during this time they are queued. When the first request is removed, the arbiter services the bus request with the next highest priority. Removing a request while a previous request is being serviced can cause a grant to be changed when arbitrating between three or four requests. For that reason, the user should not remove ungranted requests when arbitrating between three or four requests. This does not apply to arbitration between two requests.

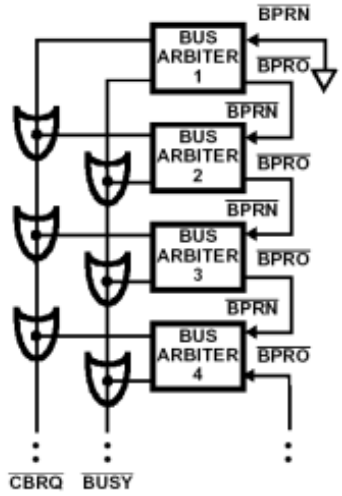
If two or more BRn inputs are asserted at precisely the same time, one of them will be selected at random, and all BGn outputs will be held in the high state until the selection is made. This guarantees that an erroneous BGn will not be generated even though a metastable condition may occur internal to the device. When the EN is in the high state the BGn outputs are forced high.

ARBITER FUNCTION TABLE

INPUTS					OUTPUTS			
EN	$\overline{BR}_0$	$\overline{BR}_1$	$\overline{BR}_2$	$\overline{BR}_3$	$\overline{BG}_0$	$\overline{BG}_1$	$\overline{BG}_2$	$\overline{BG}_3$
L	↑	X	X	X	L	H	H	H
L	X	↑	X	X	H	L	H	H
L	X	X	↑	X	H	H	L	H
L	X	X	X	↑	H	H	H	L
H	X	X	X	X	H	H	H	H



Arbitrage série ou "daisy-chain" synchrone

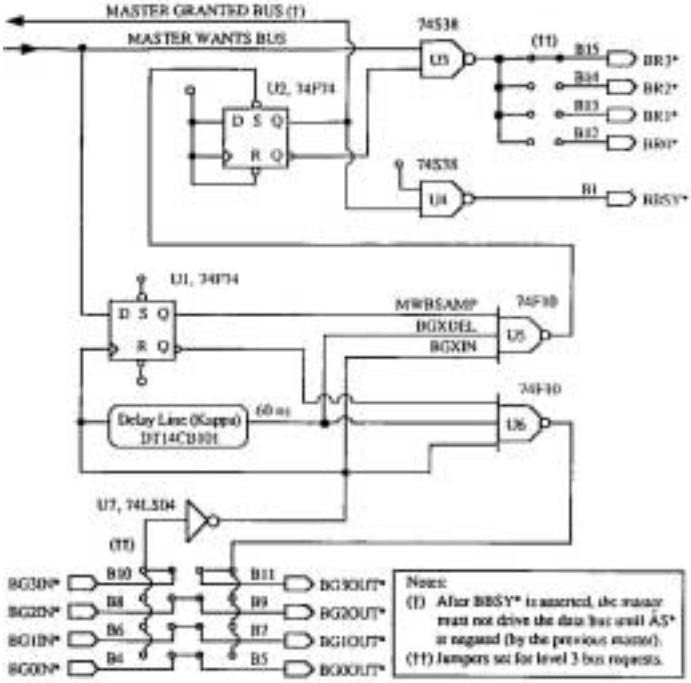


NOTE: The number of arbiters that may be daisy-chained together

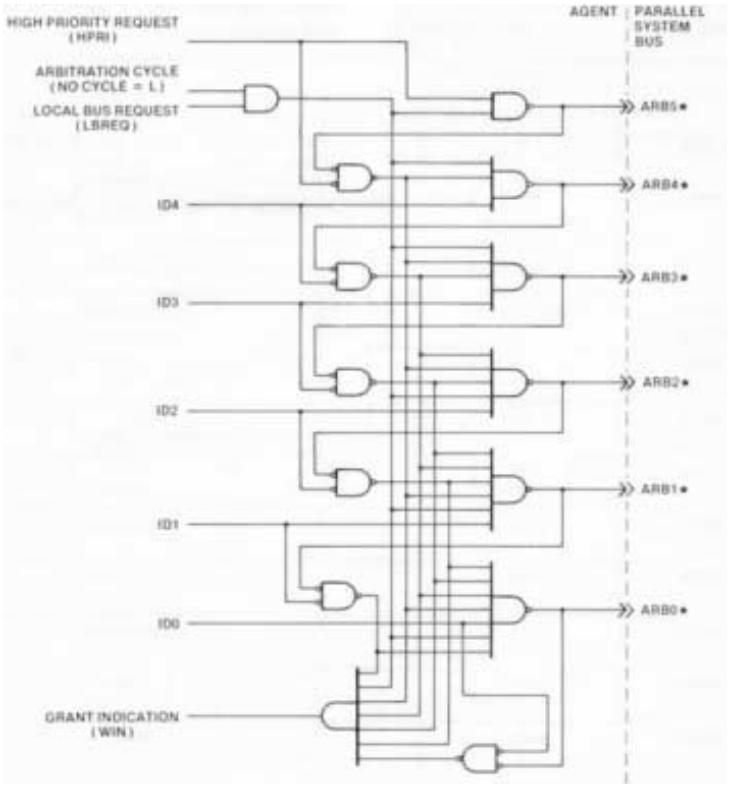
in the serial priority resolving scheme is a function of BCLK and the

propagation delay from arbiter to arbiter. Normally, at 10MHz only 3arbiters may be daisychained.

Daisy-Chain asynchrone



Arbitrage parallèle distribué "à éliminations successives"



Ci-contre schéma du Multibus II d'Intel:
Il faut un signal "arbitration cycle" pour lancer un cycle d'arbitrage, géré par un contrôleur d'ensemble.
Intérêt de cette technique :
Chacun "voit" qui prend le contrôle du bus

Techniques d'adressage : Centralisé ou distribué

Sur un bus "fond de panier" ouvert :

Sol 1 : Le maître qui a le bus place son adresse sur le bus adresse, tous les esclaves comparent cette adresse à la portion d'espace d'adressage (ou aux portions…) qui leur est allouée et un seul répond.
Si une portion a une taille multiple d'une puissance de 2 et est située à une adresse multiple de sa taille, la comparaison porte sur les bits d'adresse qui restent.

Sol2 : Un décodeur placé sur la carte maître génère autant de signaux qu'il y a d'emplacements esclaves et c'est la position dans le châssis qui définit l'adresse et sa taille.

Un seul décodeur est nécessaire s'il est placé sur le fond de panier lui-même monté en parallèle sur le bus adresse.

Dans une solution 2D, le multiplexage du bus adresse est généralement préféré, et un décodeur unique est placé en parallèle sur le bus adresse derrière ce multiplexeur.

Coreconnect arbiter

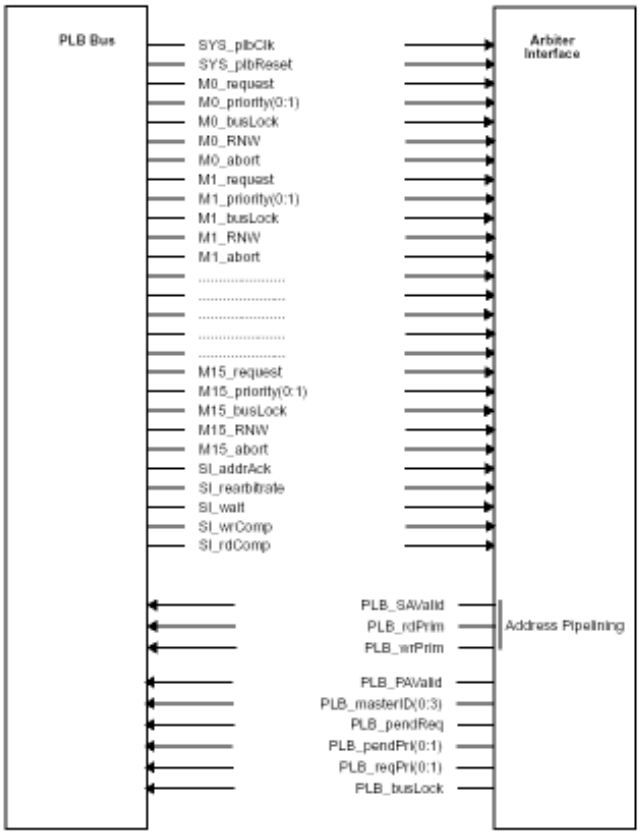


Figure 7. PLB Arbiter Interface

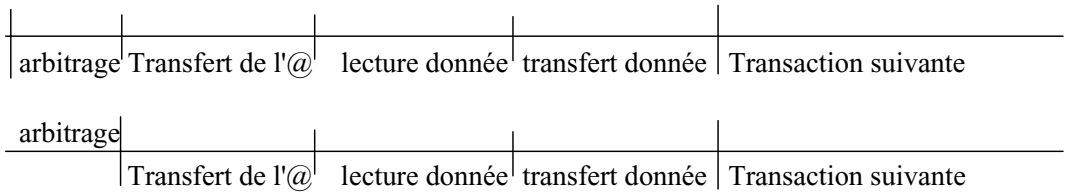
Buslock : Verrous logiciels

La section critique.
Le logiciel a besoin de temps en temps d'exécuter une suite d'instructions ou d'accéder à des données sans devoir être interrompu par qui que ce soit. Ceci est réalisé à l'aide d'un drapeau, qui lui même doit être testé et levé sans être interrompu : rôle de l'instruction TAS.
Mais ceci suppose un accès au bus atomique pour cette opération de lecture_modification_écriture, possible du fait que le maître du bus peut, en théorie, garder le bus aussi longtemps qu'il le désire (signal BBSY sur VME, LOCK sur Amba,..)

On verra que la présence de caches et d'une logique d'espionnage permet de s'affranchir de cette opération lecture_modification_écriture : il suffit de mettre sous surveillance le drapeau et on peu ainsi savoir si entre deux accès quelqu'un autre y a écrit une valeur.

Améliorer la performance

Première amélioration : réaliser l'arbitrage en temps masqué par recouvrement avec le cycle bus précédent. Peut se faire "en continu" en cas de technique d'arbitrage "premier arrivé premier servi", ou au cours de la dernière étape du cycle si arbitrage avec priorité fixe ou tournante. Ce dernier cas suppose que l'on puisse identifier la dernière étape du cycle.

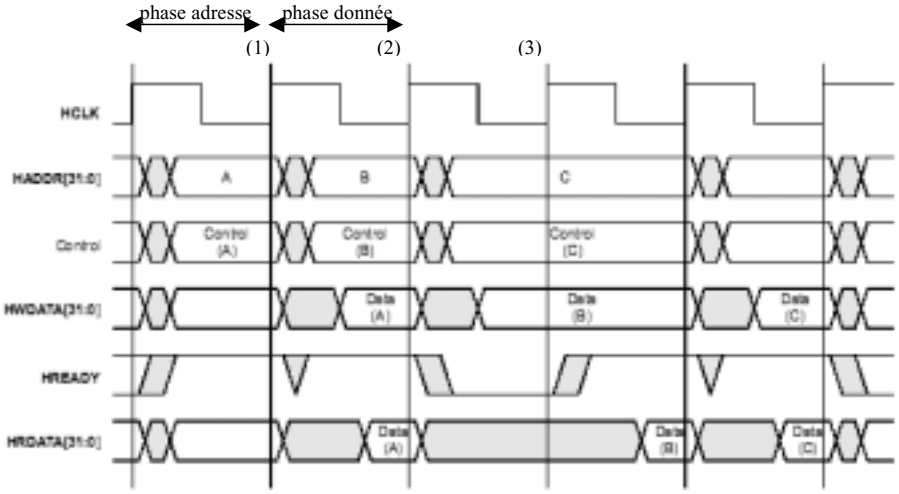


Seconde amélioration : masquer par recouvrement le temps d'attente lié au temps d'accès de la mémoire.
Solution 1 : mode pipe-line.
On profite du temps mort d'attente pour donner la main à un autre maître et passer une seconde adresse.
Ce schéma ne peut se généraliser que si on utilise deux bus, un pour l'adresse et un pour les données.
Solution 2 : mode éclaté '"split-bus" :

1. Le transfert commence normalement par la phase adresse.
2. Si l'esclave peut répondre de suite, il le fait. Sinon il répond par SPLIT . Il note le nom du maître demandeur.
3. Le bus est libéré et peut être alloué à un autre maître.
4. Dès que l'esclave est prêt pour le transfert, il signale ce fait à l'arbitre en activant un signal particulier SPLITx. Ce mécanisme donne une priorité supérieure au maître en attente (mais l'arbitre fait ce qu'il veut…)
5. le transfert s'effectue après prise de contrôle du bus par la maître adéquat.

Mode pipe-line

Les chronogrammes relatifs à un transfert de données sont les suivants (les signaux d'arbitrage ne sont pas indiqués: on suppose que les maîtres A,B et C ont successivement accès au bus) :



Un transfert se fait en deux phases :
une phase adresse et contrôle qui est <u>toujours</u> faite en 1 cycle d'horloge par le maître,
une phase donnée qui peut prendre 1 ou plusieurs cycles d'horloge, phase rallongée par le signal READY, piloté par l'esclave, et qui permet d'insérer des cycles d'attente.
Les transferts A et C sont réalisés sans cycle d'attente, le transfert B comprend 1 cycle d'attente

Le maître qui a le contrôle du bus place adresse et contrôle sur un front montant de l'horloge (1). Ces signaux sont échantillonnés par l'esclave concerné sur le front montant suivant (2) de l'horloge, qui place alors les données demandées sur le bus RDATA afin que le maître échantillonne ces données sur le front montant suivant (3) de l'horloge, à condition que READY soit à 1.
Ici, dans le cycle qui suit la phase adresse/contrôle du maître B, le signal READY n'est pas actif au prochain front montant de l'horloge et la donnée ne sera disponible qu'au cycle suivant (READY actif).

Améliorer la performance

Mode éclaté peut se compliquer si on ajoute des possibilités :
Bloquer une seconde requête faite au même banc mémoire ou l'accepter => il faut mémoriser la demande.
Dans ce cas deux bancs peuvent avoir une donnée prête en même temps (ce cas se produit aussi si les mémoires sont de type DRAM avec temps d'accès variable) et même les données peuvent être prêtes dans un ordre différent de l'ordre des requêtes => faut-il avoir plusieurs bus de données ?

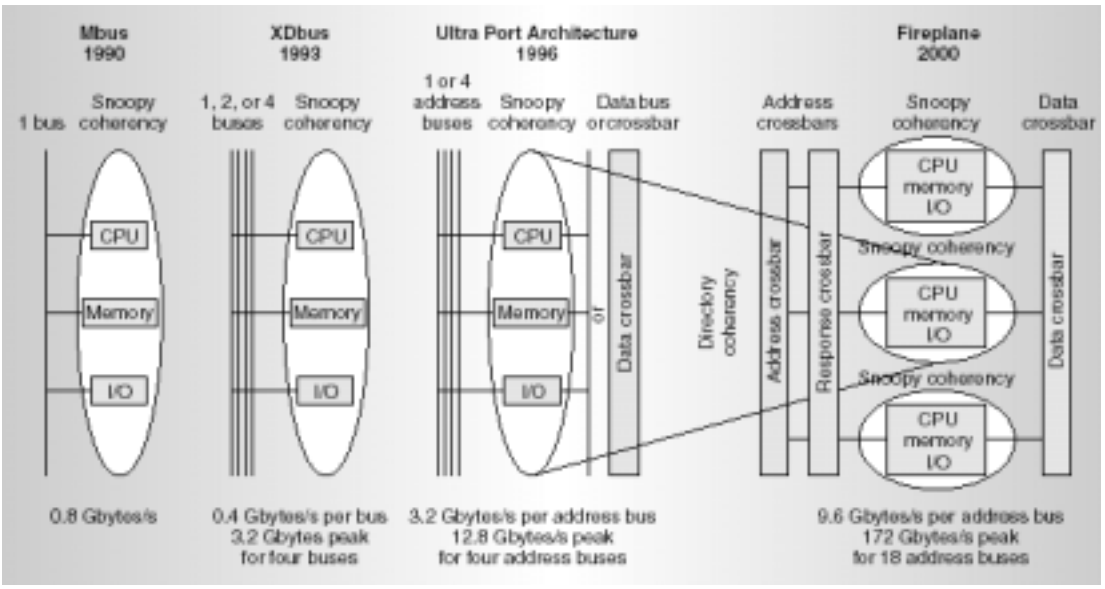
Si on augmente le degré du multiprocesseur, ne faut-il pas alors prévoir d'autres architectures : deux ou plusieurs bus d'adresse, de multiples bus de données ou mieux un crossbar de données ?
Tout est possible et a probablement déjà été envisagé, voir réalisé, mais la première façon d'améliorer la performance est de commencer par augmenter la fréquence de l'horloge du processeur, ce qui oblige rapidement à mettre en place une mémoire locale :
De type cache pour les instructions, avec des instructions de gestion du cache (permet de "geler" le contenu par exemple)
Pour les données,
- soit utiliser une mémoire gérée par le programme avec DMAs sophistiqués, intéressant pour les applications de type multimédia,
Soit utiliser une mémoire cache, intéressant pour les applications de type calcul scientifique, mais qui pose le problème de la cohérence des données partagées, qui peuvent être dupliquées dans les caches.

Les générations de chez Sun

System interconnect generation	1. MBus [7]	2. XDBus [7]	3. Ultra Port Architecture (UPA) [8]	4. Sun Fireplane
First mid-size system shipments	1991	1993	1996	2001
Processor	Cypress SPARC	SuperSPARC	UltraSPARC-I/II	UltraSPARC-III
Maximum processors in a system	4	64	64	>64
Processor clock	40 MHz	40–60 MHz	167–400 MHz	≥750 MHz
System clock	40 MHz	50–55 MHz	80–100 MHz	150 MHz
Cache-coherency mechanism	Broadcast			Broadcast + point-to-point
Packet protocol	Circuit switched	Packet switched		
Address and data	Multiplexed on same wires		Separate wires	
Cache coherency line size	32 bytes	64 bytes		
System clocks per snoop	16	11	2	1
Max snoop rate per address bus	2.5 million/sec	4.5–5 million/sec	40–50 million/sec	150 million/sec
Max data bandwidth per address bus	0.08 GBps	0.29–0.32 GBps	2.5–3.2 GBps	9.6 GBps
Max number of address buses	1	4	4	18
Max address-limited data bandwidth	0.08 GBps	1.28 GBps	12.8 GBps	172 GBps
Datapath width	8 bytes		16 bytes	32 bytes
Interconnect implementation	Bus	Buses	Mid-range: Buses High-end: Switches	Switches
Note: 1 GBps (gigabyte per second) = 10^9 bytes per second				

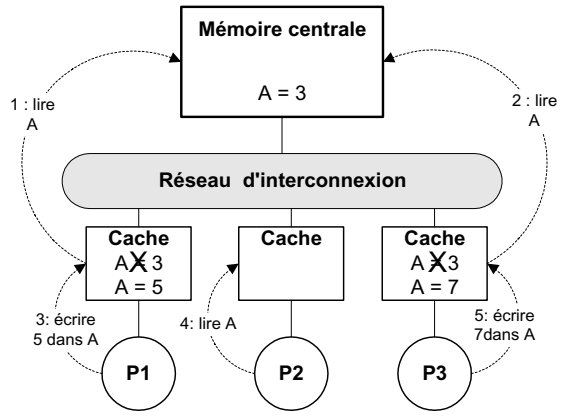
Source : SC2001 voir aussi IEEE MICRO Jan Fev 2002

Les générations de chez SUN



Evolution des architectures de bus

Problème de cohérence des données dupliquées dans les caches

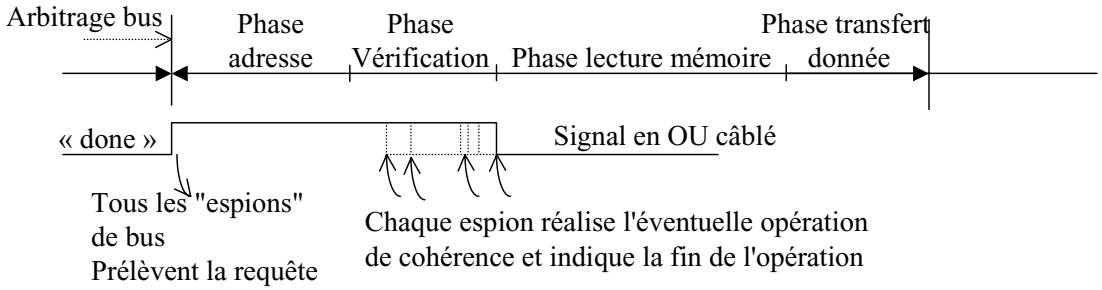


En suivant l'ordre des opérations : 1, 2,.. on observe que, sans précaution, la variable A n'a pas la même valeur partout..

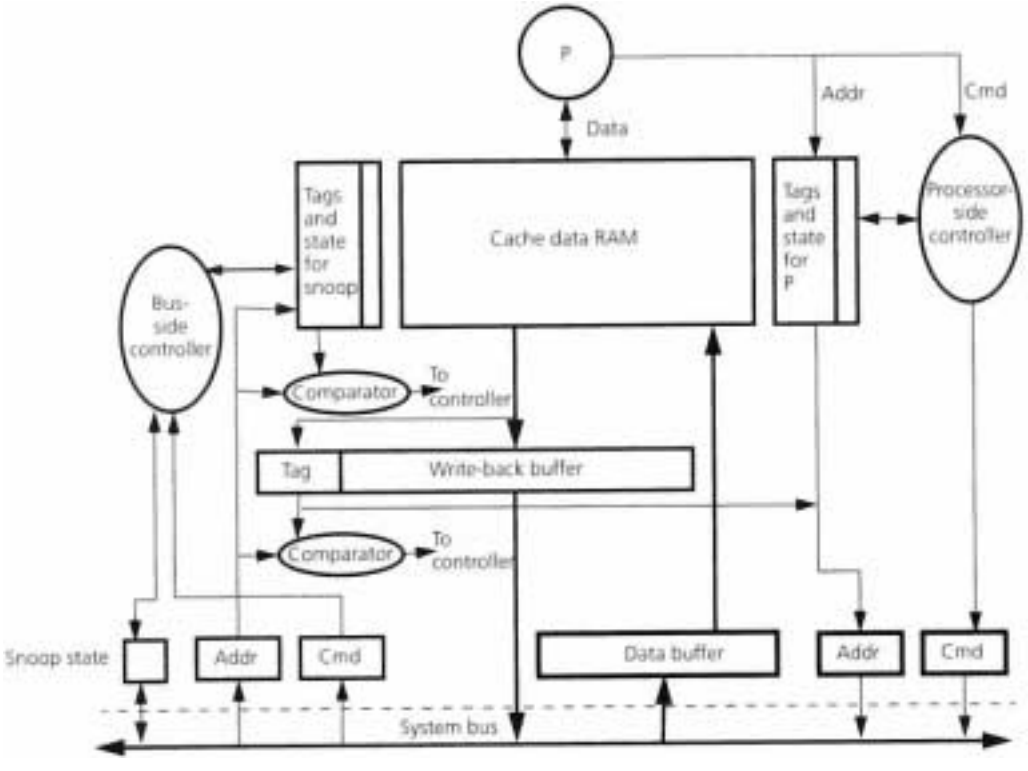
Cohérence des caches : bus atomique

Il existe deux stratégies de mise à jour des données dupliquées dans les caches:
Mise à jour immédiate : toute modification d'une donnée dans un cache est faite simultanément dans tous les autres caches (qui possèdent cette donnée).
Mise à jour différée : toute modification d'une donnée dans un cache est faite lorsque toutes les autres copies de cette donnée présentes dans les autres caches ont été invalidées.
Cette opération doit être réalisée de façon "atomique" : phase adresse, phase espionnage, phase lecture mémoire, phase transfert donnée doivent être considérées comme un tout ininterruptible.

Algorithmes de maintien de la cohérence maintenant bien connus, stabilisés depuis le début des années 90 :
Censier et Feautrier (1978) : technique du "vecteur de présence" pour tout type d'architecture,
Jim Goodman (1983) : algorithme "write once" pour bus partagé,
Jean Loup Baer (1986) : présentation homogène des algorithmes sous forme de graphe d'états.
Principe en bus partagé :



Cache avec espion de bus

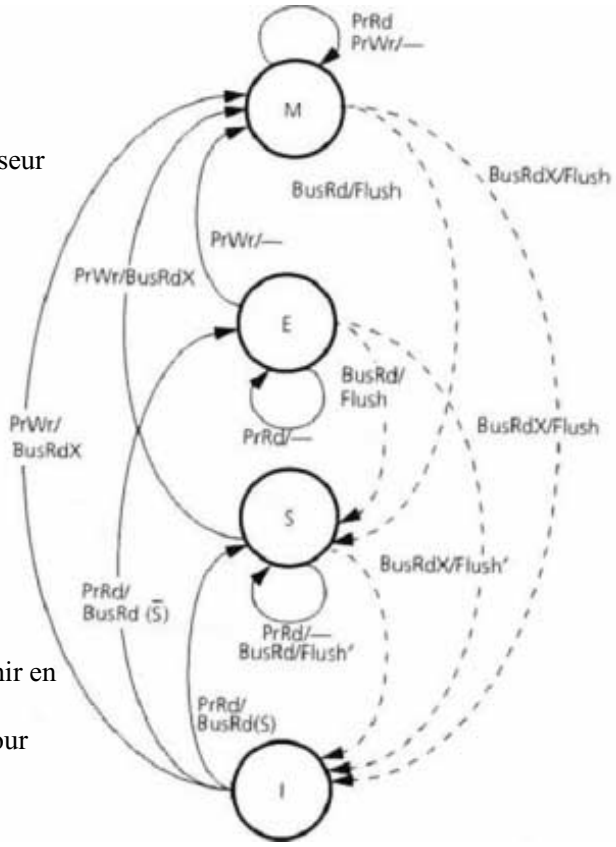


Source : Culler "Parallel Computer Architecture" Morgan Kaufmann

Machine d'états

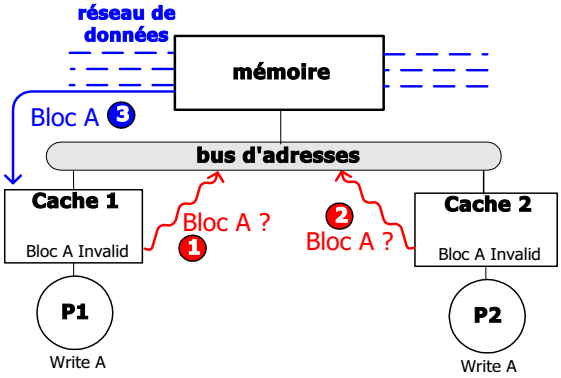
Protocole le plus utilisé : MESI
Nécessite une ligne matérielle "shared" notée S
Trait plein :transition directe (réalisée par le processeur qui fait la demande.
Trait pointillé : transition indirecte (réalisé par les autre processeurs, au vu de la demande observée sur le bus partagé)
Action processeur/ action bus
PrRd ou PrWr/ - (rien) ou BusRd ou BusRdX
Action bus / action espion
BusRd ou BusRdX / Flush

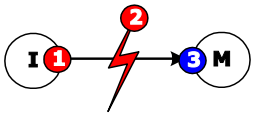
Pas de transfert cache à cache.
Flush implique un abort de la requête, une purge, puis une nouvelle demande.
Le transfert cache à cache implique un protocole à 5 états. Problèmes posés :
s'il existe plusieurs copies du cache, qui doit la fournir en cas de défaut ?
Qui doit mettre à jour la mémoire en cas de purge pour défaut de capacité ?



Cache et bus éclaté

La transaction n'est plus atomique puisque l'on intercale une ou plusieurs phases adresse autres en plein milieu de la transaction en cours.
Un conflit de cohérence se produit à chaque fois qu'il y deux requêtes consécutives sur un même bloc et que l'une des deux au moins est un défaut en écriture. Ces conflits violent l'hypothèse d'atomicité des transitions d'états du protocole car le passage effectif d'un état à un autre n'est jamais instantané. Par exemple, pour passer de l'état invalide à l'état modifié, le bloc doit être demandé et reçu avant d'atteindre la stabilité de l'état modifié. Ce passage non instantané ne pose pas de problème sur le bus atomique car les transitions entre états sont ininterrptibles, ceci dans la mesure où aucune nouvelle demande de bloc ne vient altérer la transition. Ce dernier point n'est plus vérifié sur un bus éclaté comme le montre l'exemple ci-dessous.





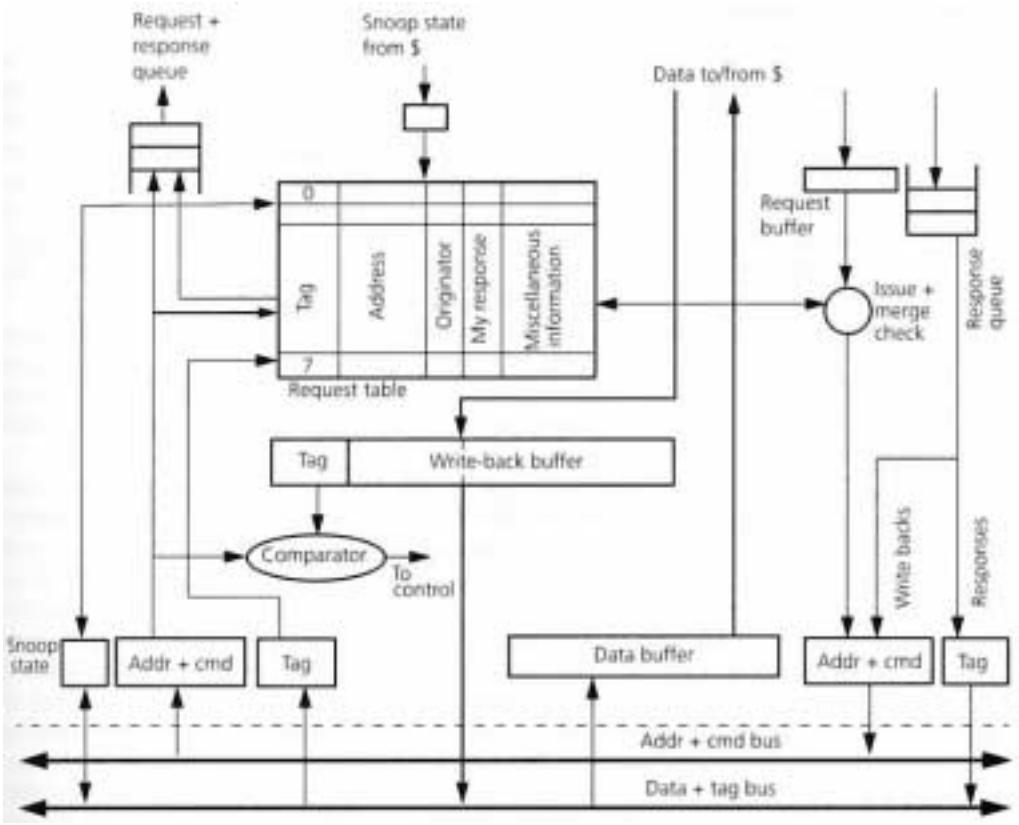
les deux processeurs P1 et P2 veulent écrire dans un bloc invalide, et donc les deux écritures se transforment en requête de défaut d'écriture. Le processeur P1 lance sa requête en premier et attend le retour du bloc, pendant ce temps le processeur P2 lance sa requête. La requête de P2 est observée sur le bus par P1, qui d'après le protocole MESI, doit prendre cette requête en charge puisque c'est lui le dernier écrivain du bloc. Mais P1 n'est pas encore en mesure de répondre car il n'a pas encore reçu le bloc qu'il va modifier. Le processeur P1 ne peut pas traiter la requête de P2, et en même temps il ne peut pas l'ignorer. La situation devient plus complexe si maintenant un processeur P3 demande le bloc en lecture puis un autre P4 le demande en écriture.

Cas1 : bus de données partagé

L'atomicité garantit que l'opération réalisée par un processeur concernant un bloc de cache ne peut être interrompue entre le moment ou il a eu le bus pour passer l'adresse et le moment ou il récupère la donnée. Ceci garantit que rien ne peut affecter l'état de ce bloc pendant la transaction mémoire.
Si le bus de donnée est aussi partagé, chaque processeur peut observer toute phase adresse et la phase donnée qui lui correspondra. Si chaque processeur s'interdit de faire une quelconque opération sur un bloc de cache en cours de transaction, on retrouve la propriété d'atomicité.
On a en fait reconstitué l'atomicité de la transaction.

Exemple : Pi veut invalider les copies d'un bloc afin d'y faire une écriture (Pi fait un PrWr, l'état du bloc est S dans son cache. Normalement Pi demande le bus et lorsqu'il l'obtient réalise un BusRdX, ce qui a pour effet d'invalider les copies existantes : les espions des autres caches qui ont une copie réalisent la transition S->I et Pi fait l'écriture avec la transition S->M).
Si un processeur Pj a demandé ce bloc avant cette invalidation et qu'il n'est pas encore arrivé, il ne peut l'invalider.
Pi doit donc attendre que Pj ait reçu son bloc avant de lancer l'invalidation.
Pour mettre en place ce mécanisme, il faut doter chaque espion d'une petite table associative qui contient les transactions en cours : toute nouvelle demande sur le bus adresse est notée, sa conclusion observée sur le bus donnée l'élimine. Un processeur qui veut faire une opération sur le bus ne peut le faire que si aucune action sur ce bloc n'est en cours. Sinon attente.

Split-bus snooping cache

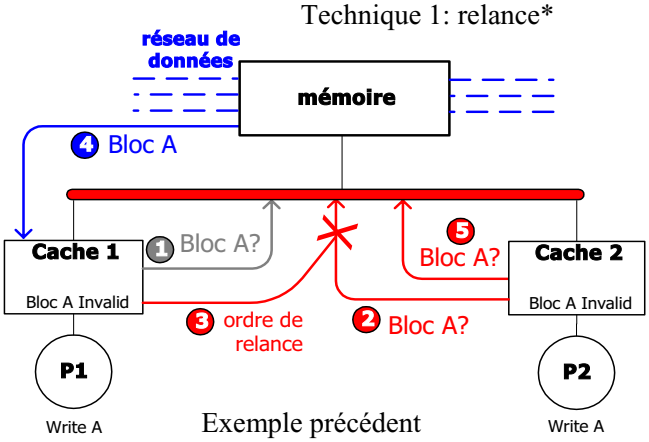


Cas 2: bus de donnée non partagé (1)

les données sont transférées sur un réseau, en général de type crossbar. Les processeurs ne peuvent plus observer les fins d'opérations.

Pour identifier toutes les situations conflictuelles possibles, nous allons **supposer** au départ qu'il y a conflit chaque fois qu'une requête de défaut en lecture/écriture est lancée sur le bus et que le bloc est dans un état que nous qualifierons "d'instable modifié" dans un autre cache. Par définition un bloc est instable modifié s'il y a eu récemment une opération lancée par le cache qui détient le bloc dans cet état, que cette opération n'est pas encore terminée, et que l'état (de départ ou final) du bloc sera modifié (sans tenir compte de la requête présente sur le bus).

Technique 1: relance*



Exemple précédent

Lorsqu'un contrôleur détecte sur le bus une requête conflictuelle (deuxième requête qui entre en conflit avec une autre déjà lancée sur le bus et non encore terminée), il demande à l'émetteur de la relancer ultérieurement (par activation d'un signal de type NACK par exemple). L'émetteur fera un nouvel essai en espérant que le conflit ne se présentera plus. Il peut être amené à faire plusieurs tentatives avant de réussir

* : cet algorithme peut aussi être utilisé pour le bus éclaté

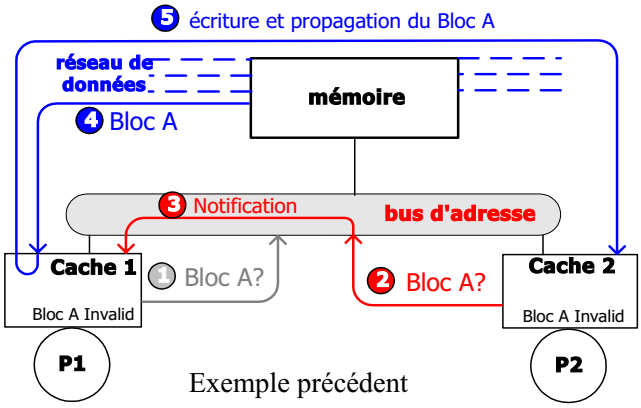
Cas 2: bus de donnée non partagé (2)

Exemple 1 plus complexe :

P1,P2,P3,P4 lancent des défauts en écriture sur le bus. Le processeur P1 voit la requête de P2 et la note dans la table des requêtes en cours. Il note aussi que le bloc doit être invalidé après avoir été reçu, modifié puis propagé au processeur P2. La requête du processeur P3 est ensuite observée par les deux processeurs P1 et P2, mais seul le processeur P2 la note. En effet, le processeur P1 détecte, après avoir consulté sa table, qu'il a déjà enregistré la requête de P2, et c'est donc à P2 de noter la nouvelle requête. Et ainsi de suite, chaque processeur qui a un défaut en écriture en cours enregistre une nouvelle requête, mais uniquement s'il n'a pas déjà noté une précédente demande. L'écriture du bloc à propager peut être faite à la volée dans un tampon, car le bloc sera invalidé dans le cache, et il n'a donc pas besoin d'être effectivement chargé dans le cache pour être modifié.

Les demandes concernant un même bloc sont chaînées

Technique de notification*



Exemple précédent

Avec cette technique une requête est lancée avec succès même si elle est conflictuelle, sachant qu'une réponse cohérente est toujours garantie pour cette requête. Ceci veut dire que chaque requête doit recevoir la copie du bloc correspondante à la dernière écriture à l'origine d'une opération sur le bus. A cet effet chaque requête en cours note juste la requête conflictuelle suivante qui la concerne émise sur le bus. La copie correspondante du bloc sera aussitôt retransmise après réception puis modification.

* : cet algorithme peut aussi être utilisé pour le bus éclaté

Cas 2: bus de donnée non partagé (2bis)

Exemple 2 : P1,P2,P3,P4 avec DE,DL,DL,DE

Des défauts en lecture viennent s'intercaler entre des défauts en écriture. La différence ici apparaît au niveau des défaut en lecture. Un processeur qui a un défaut en lecture ne doit enregistrer une requête que si le bloc demandé n'est pas à jour dans la mémoire. Il doit donc attendre le résultat de l'espionnage des autres processeurs pour savoir si une copie modifiée du bloc réside dans un cache. Si c'est le cas, la requête est enregistrée, à condition que le processeur n'ait pas déjà noté une précédente requête sur ce bloc. Si on suppose que P1 reçoit le bloc et le propage (avec une mise à jour de la mémoire) après la requête de P3 mais avant celle de P4, alors la demande de P4 ne sera pas rajoutée à la chaîne DL(P2)->DL(P3) de notification, elle est servie depuis la mémoire (P3 ignore la requête de P4 car il voit qu'aucun processeur n'a signalé un état modifié du bloc qui vient d'être mis à jour dans la mémoire). Dans cette chaîne, le processeur P2 invalide le bloc après l'avoir propagé puisqu'il a vu la demande en écriture de P4. Notons qu'il est possible d'avoir à un moment donné une chaîne de propagation de réponses alors que la mémoire est déjà à jour, ce qui est le cas dans cet exemple au moment où la requête de P4 est lancée.

Plusieurs bus adresse..

Le bus adresse devient rapidement un goulet d'étranglement. Le bus de données peut être remplacé par d'autres types de réseaux, ce qui n'est pas le cas pour le bus adresse si on veut pouvoir maintenir la cohérence par des algorithmes de type espion.
Mais on peut dédoubler ce bus, l'un utilisé pour les adresses paires et l'autre utilisé pour les adresses impaires. Les mémoires sont aussi indépendantes et contiennent respectivement les adresses paires et impaires. En entrelaçant les adresses de façon ad hoc, on peut espérer répartir la charge de façon assez équilibrée. Es ce réellement le cas ?
Ci dessous quelques résultats

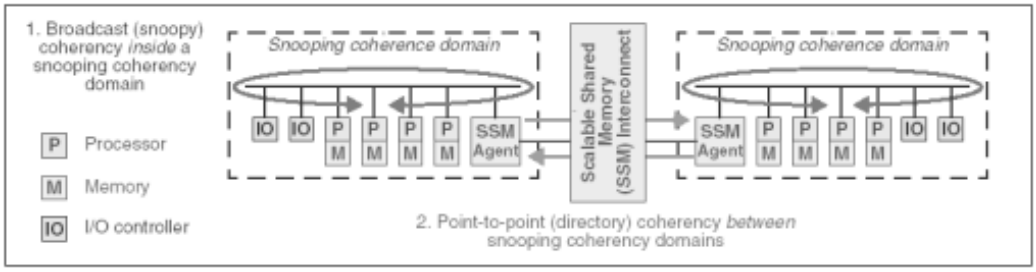
Avec 2 bus :

Application	Rendement	Sollicitation BUS0	Sollicitation BUS1
LU	99,0%	49,89%	50,11%
FFT	96,5%	50,32%	49,68%
OCEAN	95,4%	49,95%	50,05%
RADIX	96,9%	49,89%	50,11%

Avec 4 bus (entrelacement des adresses modulo 4) :

Application	Rendement	Sollicitation BUS0	Sollicitation BUS1	Sollicitation BUS2	Sollicitation BUS3
LU	88,5%	24,31%	24,84%	25,24%	25,61%
FFT	94,7%	25,73%	24,90%	24,82%	24,55%
OCEAN	86,4%	25,16%	24,26%	25,05%	25,53%
RADIX	94,7%	24,93%	25,02%	24,89%	25,15%

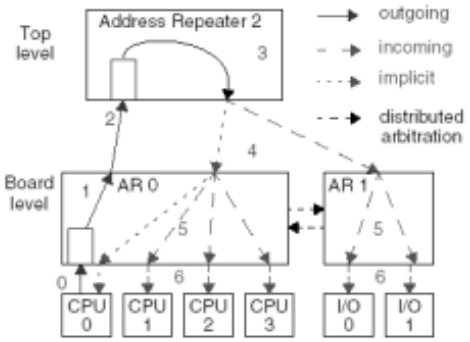
Bus partagé virtuel : solution bus Fireplane de Sun



Address bus implementation

The address bus is implemented using a two-level bidirectional tree structure of Address Repeater ASICs. CPU 0 issues an outgoing request (indicated by the solid red arrows) in cycle 0 to its board Address Repeater (AR0). Assuming that the bus to the top-level Address Repeater is available, the outgoing request is sent up in cycle 2. In cycle 3, the top-level Address Repeater arbitrates between requests, and selects this request to send back down to the board-level Address Repeaters.

In this example, the request is physically transmitted (indicated by a dashed green arrow) only to AR1, since AR0 sent the original request and kept a copy of it, so it only needs to be told when to use the copy. Thus, in cycle 4 the AR0 to AR2 bus is not physically used (indicated by the dotted green arrow). In cycle 6 all the system devices see the same transaction. CPU 0 sent the original request and therefore does not need a physical re-transmission of the address (indicated by the dotted green arrow). The incoming path length (green dashed or dotted) is a fixed length for all devices on the bus. The outgoing path (red) may hold queued transactions.



Source: The Sun Fireplane System Interconnect, Alan Charlesworth, Sun Microsystems, Inc SuperComputing 2001

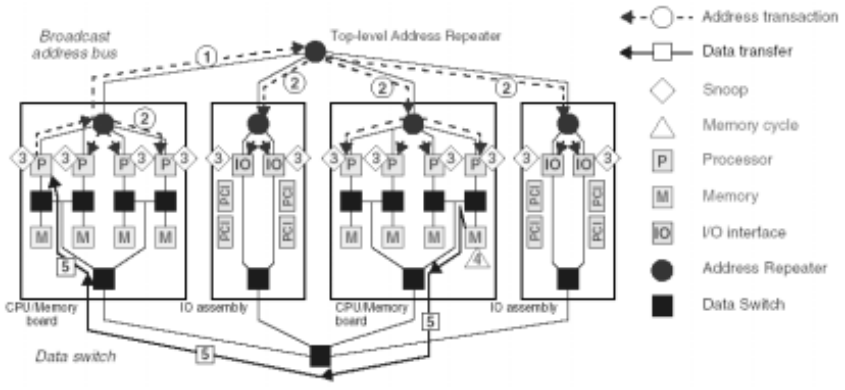
Chronologie d'un accès mémoire sur le bus Fireplane

Example Fireplane interconnect operation

1 Address request (cycle 0–2). The requesting CPU makes a *ReadToShare* request to its board Address Repeater, which sends the request up to the top-level Address Repeater.

2 Broadcast address (cycle 3–6). The top-level Address Repeater chooses this request to broadcast throughout the snooping coherence domain. It sends the request back down to the board-level Address Repeaters. Every system device gets the address on cycle 6.

3 Snoop (cycle 7–15). Each system device examines its coherency tags to determine the state of the cache line. There are three snoop-state signals: *shared*, *owned*, or *mapped*. Each device sends its snoop-out result to its Board Data Switch on cycle 11. Each Board Data Switch ORs its local snoop results, and sends them to every other Board Data Switch.



Each Board Data Switch computes the global snoop result, and sends the snoopin result to its system devices on cycle 15. If the snoop result had been a hit, then a cache-to-cache transfer would have been initiated on cycle 16 by the owning device. In that case, the memory cycle would have been ignored.

4 Read from memory (cycle 7–22). The target memory controller (which is inside a CPU) recognizes that the request is in its address range, and initiates a read cycle in its memory unit. In cycle 22, the data block is sent from the DIMMs to the local Dual CPU Data Switch.

5 Transfer data (cycle 23–36). The data block is moved through the Board Data Switch, the System Data Switch, the requester's Board Data Switch, and the Dual CPU Data Switch. The four 18-byte portions of the data block arrive on the wires into the requesting CPU in cycles 33-36. The address interconnect takes 15 system cycles (150 ns) to obtain the global snoop result and send it to the system devices. The DRAM access is started partway through the snoop process, and takes an additional 7 cycles (47 ns) after the snooping is finished. These two latencies have a fixed minimum time. The data transfer takes a variable number of clocks, depending upon how many data switch levels are between data source and destination.

It takes 14 cycles (93 ns) when they are on different boards, 9 cycles (60 ns) when they are on the same board, and 5 cycles (33 ns) when they are on the same CPU.

Cache-to-cache transfers This is the case when data is owned (modified) in a cache.

Inside a snooping coherence domain. The owning system device asserts a snoop result of *owned*, and sends the data directly

to the requester. The memory cycle is ignored. The pin-to-pin latency is increased by 11 system clocks (73 ns) over the time required to do a load from memory.

Between snooping coherence domains. A three-way transfer is done. The coherency directory cache entry of the home SSM

agent tells it which snooping coherency domain currently owns the data. The home SSM agent sends the owning SSM agent a *Read- ToShareMtag*. The owning SSM agent runs the transaction on its local address bus, and supplies the data directly to the requesting data agent. The latency is 21 system clocks (140 ns) more than for an unowned SSM transfer, and 24 system clocks (160 ns) more than for a cache-to-cache transfer in a non-SSM system.

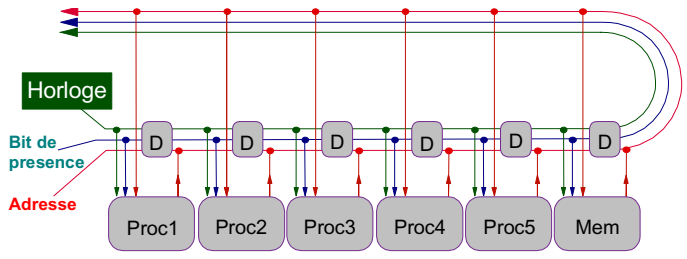
Conclusion sur les multiprocesseurs à bus partagé

Structure d'interconnexion simple mais qui devient vite un goulet d'étranglement par manque de bande passante. La solution cache règle ce problème de façon élégante mais se prête mal à des applications de type temps réel critique, ou seules des mémoires locales doivent être envisagées avec des réseaux de communication plus performants.

La solution bus partagé est la seule qui autorise l'utilisation des algorithmes de type espion de bus, qui s'accommode d'ailleurs de nombreuses améliorations : réseaux d'interconnections pour les données variés avec transfert de cache à cache possible, multiplication des bus d'adresse avec faible effet de bord.

Le répertoire du cache semble devenir un autre goulet d'étranglement, puisqu'il est consulté à chaque accès mémoire du processeur et à chaque accès de l'espion de bus.
Dupliquer le répertoire est une solution usuelle.
Paradoxalement, une hiérarchie de caches à 2 niveaux règle le problème, sous certaines conditions bien identifiées, ou le niveau 1 (le plus près du processeur) est un sous-ensemble strict du niveau 2. Le processeur travaille sur le répertoire du niveau 1, l'espion sur celui du niveau 2, et tous deux se synchronisent uniquement lors des modifications d'états, heureusement peu fréquentes.

support d'adresse fibre optique



+ Plusieurs processeurs envoient une adresse en même temps

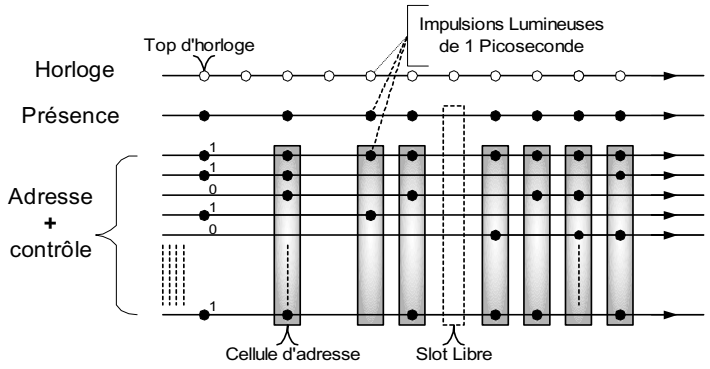
+ Pas d'arbitrage (détection simple des conflits)

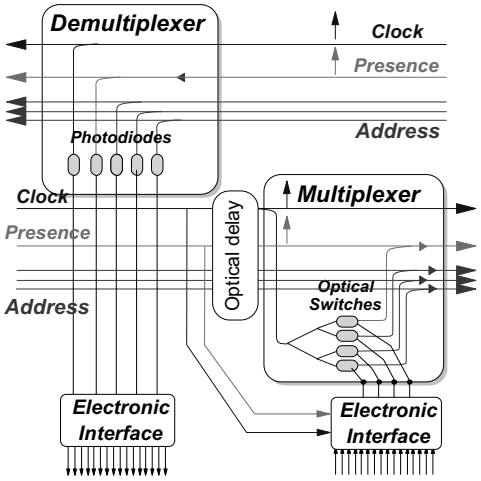
+ Propagation libre des cellules (pas de retransmission)

- Perte de l'atomicité d'envoi : complexifie la gestion de la cohérence

Le bus optique est un ruban unidirectionnel qui connecte les processeurs à la manière d'un anneau, c'est à dire que plusieurs requêtes peuvent le traverser en même temps. Le bus est logiquement divisé en deux régions : la partie inférieure qui sert pour l'envoi des requêtes et la partie supérieure qui délivre les requêtes. La mémoire est située entre ces deux parties. L'extrémité supérieure est laissée ouverte pour annihiler les cellules optiques émises. Le ruban est divisé en trois bandes : l'horloge constituée d'une seule fibre optique, le bit de présence également constitué d'une fibre optique unique et enfin le lien adresse qui est en fait un ensemble de n+p fibres optiques permettant d'envoyer en parallèle les n bits d'une adresse ainsi que p bits d'identification de la requête. Les "bits optiques" sont matérialisés par des impulsions lumineuses dont la durée est de l'ordre de la picoseconde. L'horloge synchronise les transferts, et à chaque impulsion lumineuse de l'horloge correspond ou non une impulsion lumineuse pour le bit de présence qui indique si le lien d'adresse qui achemine l'adresse est libre ou non.

Détails de fonctionnement du bus optique



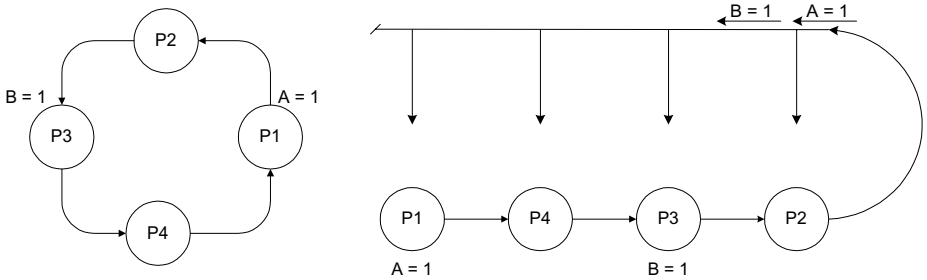


Le fonctionnement du bus optique part de deux principes :

- Tous les processeurs peuvent envoyer des requêtes en même temps, aucun arbitrage n'est nécessaire ;

- Les requêtes se propagent librement sur le bus, elles ne sont ni stockées ni re-émises à chaque passage d'un nœud comme c'est le cas dans l'anneau électronique.

Propriétés logiques de l'Ubus comparées à celles de l'anneau

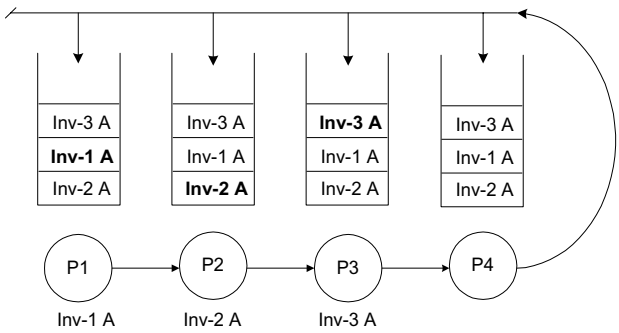


D'un point de vue logique, l'Ubus et l'anneau se ressemblent car tous deux suppriment la contrainte d'arbitrage que l'on trouve sur le bus électronique. Les requêtes se propagent librement dans un sens unique et sont prélevées après un tour complet sur le support. Cependant, une différence primordiale existe entre ces deux structures. Sur l'Ubus, les requêtes sont reçues dans un ordre identique par tous les processeurs, ce qui n'est pas le cas de l'anneau sur lequel la position d'un processeur détermine l'ordre de réception.

Sur la figure ci-dessus les deux processeurs P1 et P3 lancent une requête d'invalidation sur les emplacements mémoire A et B respectivement, et en même temps. Dans le cas de l'anneau, si on suppose que le sens de propagation est celui inverse des aiguilles d'un montre, le processeur P2 observe la requête pour A suivie de celle pour B alors que le processeur P4 observe la requête pour B suivie de celle pour A. Sur l'Ubus, cette situation n'est pas possible, car tous les processeurs observent les requêtes à partir du même endroit, et donc tous les processeurs verront la requête de B suivie de celle de A.

L'ordre de propagation unique observé sur l'Ubus est une propriété clé pour la cohérence et la consistance. Elle facilite la mise en œuvre de la sérialisation et de l'atomicité des accès mémoires. C'est un avantage fondamental que présente une architecture Ubus par rapport à celle d'un anneau.

Gestion de la cohérence : adaptation du protocole MESI



Difficultés recensées

1 - gestion des écriture multiples

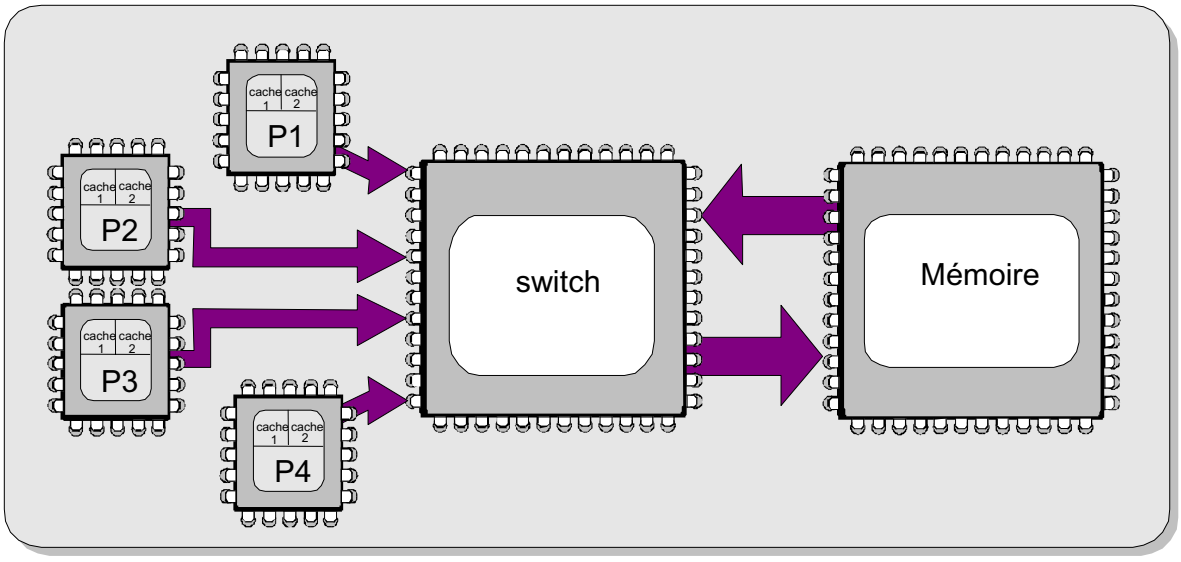
2 - décision de notification

3 - gestion des purges

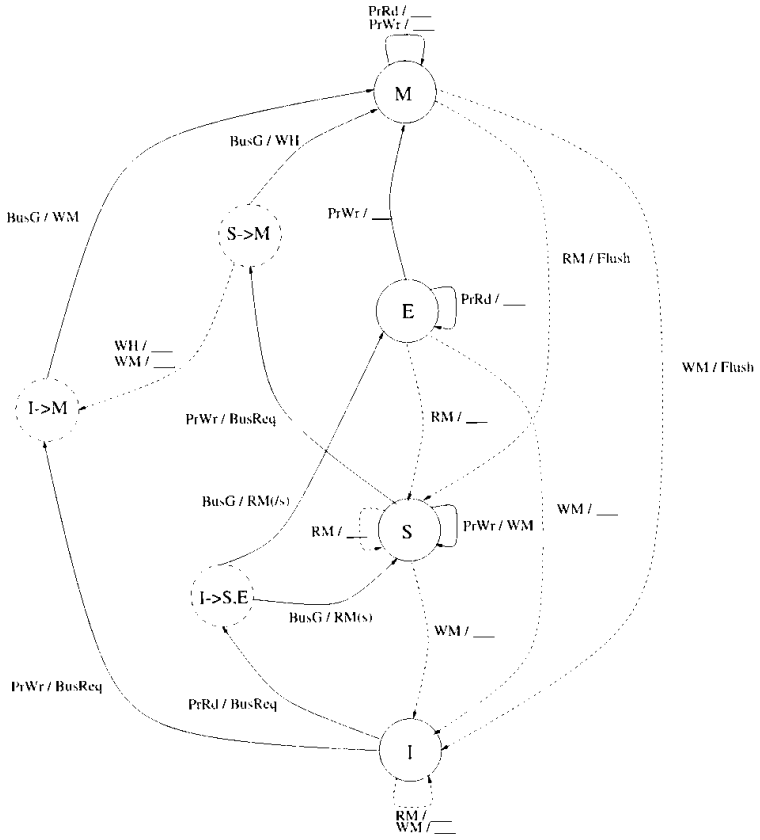
Exemple de situation de conflit : Plusieurs processeurs tentent d'écrire un bloc en même temps. Si le bloc est dans l'état partagé alors les écritures vont générer une suite d'invalidations sur l'Ubus. Il faut alors disposer d'un mécanisme qui permette de choisir et d'accepter une seule invalidation dans le lot, et de signaler aux autres processeurs ce choix.

le processeur qui voit son invalidation arriver en tête considère automatiquement qu'elle a été acceptée. Aucun accusé d'opération n'est alors nécessaire car chaque processeur prend localement la même décision. P2 effectue son écriture dès qu'il voit son invalidation arriver en tête de la file de réception. Les processeurs P1 et P3 abandonnent leurs invalidations et considèrent qu'elles sont automatiquement vues comme des défauts en écriture: le processeur P2 qui voit derrière son invalidation une invalidation du processeur P1 considère que cette dernière est un défaut en écriture, et de même pour le processeur P1 qui considère que l'invalidation du processeur P3 est un défaut en écriture.

Conclusion ?



Cohérence et états intermédiaires sur bus éclaté



Cohérence et états intermédiaires sur Ubus