Variante des MSI-Protokolls

Im Zustand M wird BusRd gesehen.
Welche Transition ist durchzuführen?

- hängt vom Zugriffsmuster der Anwendung ab:
  - Übergang zu **S** erwartet eher ein nachfolgendes *Read* und nicht unbedingt ein *Write* eines anderen Prozessors
    - gut wenn hauptsächlich Daten nur gelesen werden
    - aber was ist mit „wandernden" Daten?
      - *Read* und *Write* von P_1, dann *Read* und *Write* von P_2) dann *Read* und *Write* von P_x, ...
      - ggf. besser in Zustand I wechseln und Cache-Line wird frühzeitig frei
  - Rechnersystem *Synapse* wechselt von Zustand M in Zustand **I**
  - Rechnersysteme *Sequent Symmetry* und *MIT Alewife* benutzen **adaptives Protokolle**
- Wahl des Protokolls hat durchaus Einfluss auf die Performance des Speichersystems

Invalidierungsprotokoll mit 4 Zuständen

Problem mit MSI

- *Read* & *Modify* von Daten erfordert zwei Bus-Transaktionen
  - BusRd (I→ S) gefolgt von BusRdX oder BusUpgr (S→M)
- Zwei Bus-Transaktionen notwendig, auch wenn der Block nicht geteilt wird

Lösung:

- Erweiterung um *Exclusive* Zustand
  - damit lokales *Write* ohne Bus-Transaktion durchführbar ist
- Benötigte Zustände
  - *Invalid* und *Modified* (wie bisher)
  - *Exclusive* bzw. *Exclusive-Clean* (nur in diesem Cache, nicht modifiziert)
  - *Shared* (zwei oder mehrere Caches können Kopien haben)

MESI Invalidierungsprotokoll

- Zustände
  - *Modified, Exclusive, Shared, Invalid*
- Zusätzliches Signal auf dem Bus
  - „Shared-Line“ gesetzt durch Cache-Controller
  - Controller setzt Signal als Reaktion auf BusRd, falls der Cache den angefragten Block als *Valid* vorliegen hat

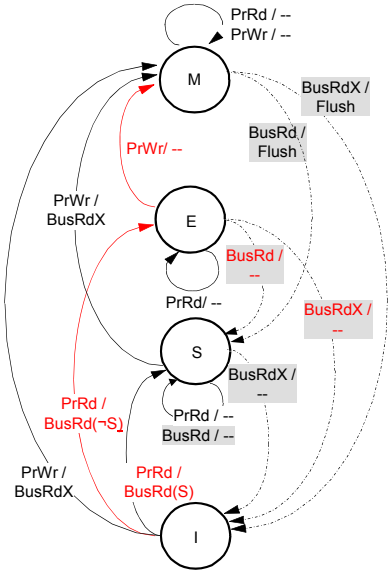
- Neuer Zustandsübergang I→ E
  - PrRd mit BusRd und kein anderer Cache hat eine Kopie
  - bedeutet, Shared-Line auf dem Bus ist nicht aktiv
- Zustandsübergang I→ S
  - PrRd mit BusRd und mindestens ein anderer Cache hat eine Kopie
  - bedeutet, Shared-Signal ist gesetzt

MESI Varianten eingesetzt in Intel x86-32, PowerPC, MIPS, …

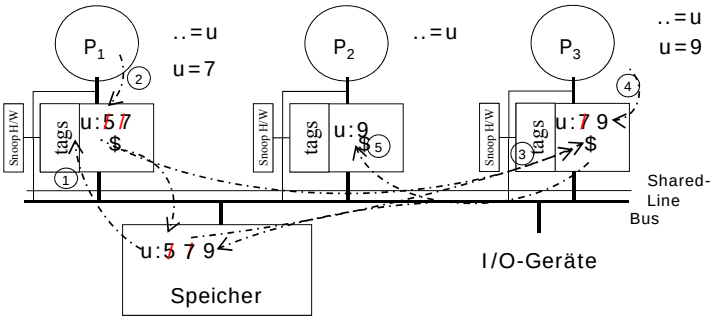
Zustandsübergangsdiagramm: MESI

Ohne Cache-to-Cache Sharing

- BusRd(S): Shared-Line wurde während BusRd-Transaktion gesetzt
- BusRd(¬S): Shared-Line wurde nicht gesetzt
- Cache-Controler setzt S-Signal, falls BusRd / BusRdX auf Block im M,E oder S Zustand gesehen wird
- Hauptspeicher
  - antwortet mit Daten, falls BusRd / BusRdX ohne Flush gesehen wird
  - beim Flush wird der Block auch im Speicher abgelegt, damit ist BusWB nur beim Verdrängen im Zustand M notwendig



Beispiel: MESI



(1) P_1 liest *u*
Cache Miss

(2) P_1 schreibt *u*

(3) P_3 liest *u*
Cache Miss

(4) P_3 schreibt *u*

(5) P_2 liest *u*
Cache Miss

Aktion	*Pr-Aktion*	*Bus-Aktion*	*Signal*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1a	PrRd (1)	BusRd	?		-	-	-
1b			-	Memory	E	-	-
2	PrWr (1)	-	-	-	M	-	-
3a	PrRd (3)	BusRd	?				
3b		Flush	S	Cache P_1	S	-	S
4	PrWr (3)	BusRdX	-	Memory	I	-	M
5a	PrRd (2)	BusRd	?				
5b		Flush	S	Cache P_3	I	S	S

MESI Protokoll-Varianten

Falls nicht im *Modified* Zustand, wer liefert beim Miss die Daten: Speicher oder Cache?

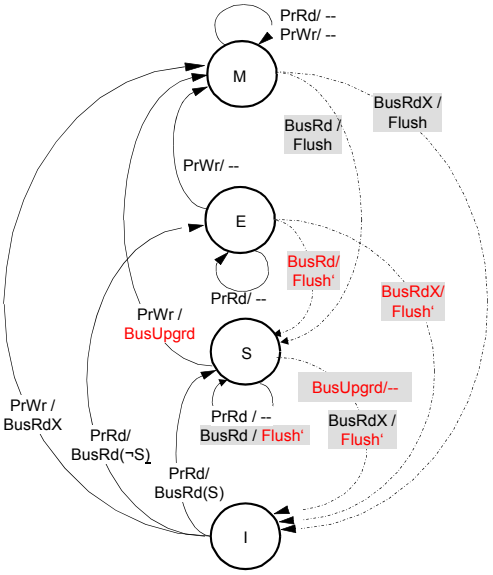
- Original Illionois MESI (1984): Cache, der gegenüber Speicher als schneller angenommen wird
  - Cache-to-Cache Sharing
- für manche Systeme ist diese Annahme nicht unbedingt gültig
  - Stören anderer Caches ist ggf. teurer als ein Speicherzugriff

- Cache-to-Cache Sharing ist
  - komplexer
    - woher weiß Speicher, dass Daten ausgeliefert werden müssen
      - muss Speicher erst auf Caches warten? Time-outs?
    - Auswahlverfahren, wenn mehrere Caches validen Wert haben
  - aber gut in Cache-kohärenten System mit verteiltem Speicher
    - kann günstiger sein, wenn Daten im nahen Cache und nicht im entfernten Speicher liegen
    - besonders dann, wenn SMP-Knoten eingesetzt werden (Stanford DASH)

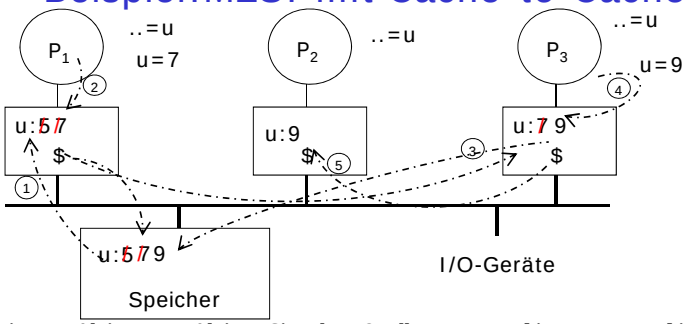
Zustandsübergangsdiagramm: Extended MESI

Mit Cache-to-Cache Sharing

- BusUpgrd (S → M): ohne Datentransfer
- Flush' (S → S oder S → I): ggf. mehr als ein Cache versucht Daten auf den Bus zu schreiben
- BusRd / BusRdX, Hauptspeicher antwortet nur mit Daten, falls kein S-Signal gesetzt
- Flush im M-Zustand, dann Block auch im Speicher ablegen
- Write-Back: BusWB nur im M-Zustand notwendig



Beispiel: MESI mit Cache-to-Cache Sharing



(1) P_1 liest *u* **Cache Miss**

(2) P_1 schreibt *u*

(3) P_3 liest *u* **Cache Miss**

(4) P_3 schreibt *u*

(5) P_2 liest *u* **Cache Miss**

Aktion	*Pr-Aktion*	*Bus-Aktion*	*Signal*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1a	PrRd (1)	BusRd	?		-	-	-
1b			-	Memory	E	-	-
2	PrWr (1)	-	-	-	M	-	-
3a	PrRd (3)	BusRd	?				
3b		Flush	S	Cache P_1	S	-	S
4	PrWr (3)	BusUpgrd	-	-	I	-	M
5a	PrRd (2)	BusRd	?				
5b		Flush	S	Cache P_3	I	S	S

MESI State Table Cache Controller (1)

State	Event	Action	Set Bus Signal	Next State
Invalid*	Read miss (if S-Sig is set)	Broadcast Rd Get cache line	-	S
	Read miss (if S-Sig is not set)	Broadcast Rd Get cache line	-	E
	Write miss	Broadcast RdX Get and modify cache line	-	M
Shared	Read hit	-	-	S
	Write hit	Broadcast Upgrade	-	M
	Snoop hit on Rd	Reply with cache line (Flush')	S-Sig	S
	Snoop hit on RdX	Reply with cache line (Flush')	S-Sig	I
	Snoop hit on Upgrade	Invalidate cache line	S-Sig	I

* Implicit Invalid state with replacement of a cache line

MESI State Table Cache Controller (2)

State	Event	Action	Set Bus Signal	Next State
Exclusive	Read hit	-	-	E
	Write hit	-	-	M
	Snoop hit on Rd	-	S-Sig	S
	Snoop hit on RdX	Invalidate cache line	-	I
Modified	Read hit	-	-	M
	Write hit	-	-	M
	Snoop hit on Rd	Flush cache line	S-Sig	S
	Snoop hit on RdX	Flush cache line	-	I
	LRU Replacement	Write back	-	-

MESI Memory Controller Actions

Memory Controller:

Bus Command	Bus Signal	Action
Rd	-	Reply with block
Rd	S-Sig	-
RdX	-	Reply with block
RdX	S-Sig	-
Upgrade		-
Write back		Store block
Flush		Store block
Flush‘		-

Weitere Verbesserungen

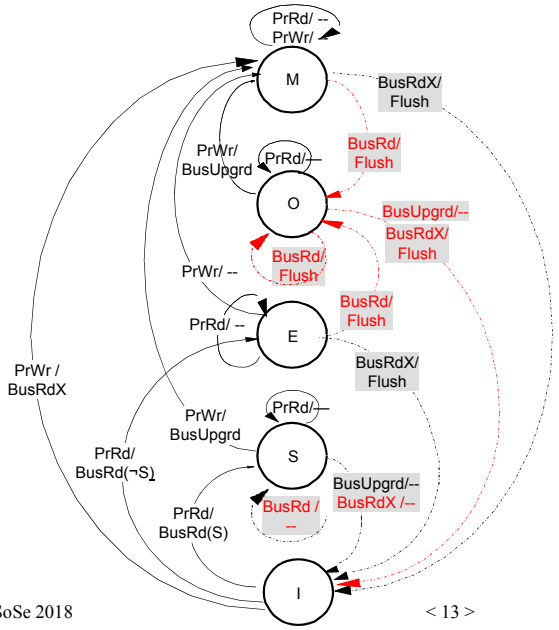
Welcher Cache liefert aktuellen Block falls mehrere Caches Block im Zustand S halten?

Lösung: Einführung eines weiteren Zustands O (Owner)

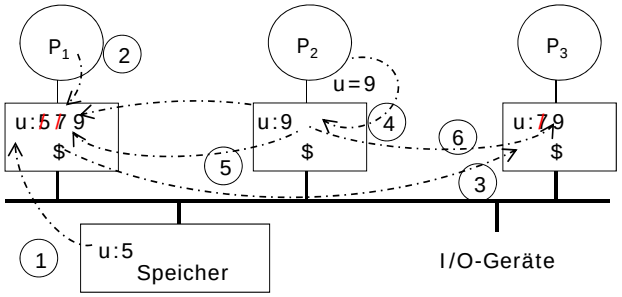
- Zustand O besagt:
  - Eigentümer des Blocks hat besondere Aufgaben
  - Block ggf. verändert, aber noch nicht im Speicher geschrieben
  - Valide Kopien des Blocks in anderen Caches möglich
  - Alle anderen Kopien des Blocks sind im Zustand S

Zustandsübergangsdiagramm MOESI

- Cache-to-Cache Sharing
  Flush im Zustand M, O oder E,
  Flush ohne Wirkung auf den Hauptspeicher
- Write-Back
  im Zustand M und O mit BusWB,
  im Zustand O zusätzlich mit Inval aller Kopien.



Beispiel: MOESI



(1) P_1 liest *u*
Cache Miss

(2) P_1 schreibt *u*

(3) P_3 liest *u*
Cache Miss

(4) P_2 schreibt *u*
Cache Miss

(5) P_1 liest *u*
Cache Miss

(6) P_3 liest *u*
Cache Miss

Aktion	*Pr-Aktion*	*Bus-Aktion*	*Signal*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1					E	-	-
2	PrWr (1)	-	-	-	M	-	-
3a	PrRd (3)	BusRd	?				
3b		Flush (1)	S	Cache P_1	O	-	S
4a	PrWr (2)	BusRdX					
4b		Flush (1)	-	Cache P_1	I	M	I
5a	PrRd (1)	BusRd	?				
5b		Flush (2)	S	Cache P_2	S	O	I
6a	PrRd (3)	BusRd	?				
6b		Flush (2)	S	Cache P_2	S	O	S

ccNUMA Unterstützung

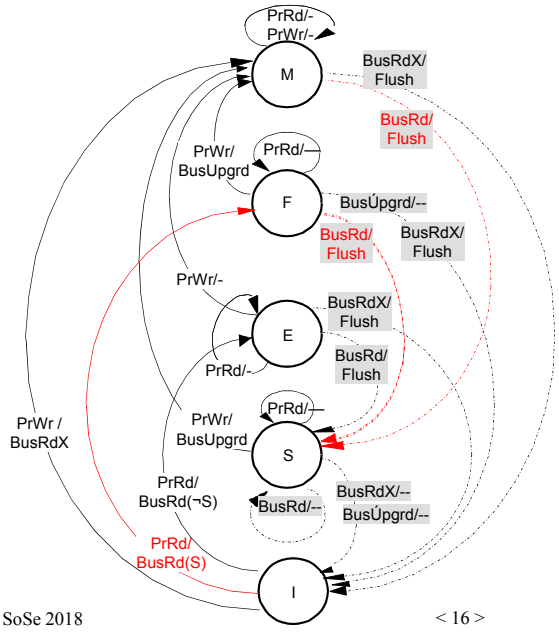
- Einführung eines neuen Zustandes „Forward (F)“
- Cache mit Block im F-Zustand antwortet auf BusRd, Caches mit Block im S-Zustand müssen nichts machen
- Nach einem BusRd mit Teilhaber wandert der Zustand F zum Cache dessen Prozessor das PrRd initiiert hat
  - Cache-Line im alten F-Zustand geht in Zustand S
  - Cache-Line mit neuer Kopie geht in Zustand F

- Vorteile:
  - Verdrängung einer Cache-Line im Zustand F wird weniger wahrscheinlich (wegen zeitlicher Lokalität)
  - bessere Verteilung der Rolle der antwortenden Caches über alle Knoten (weniger Hot-Spots)

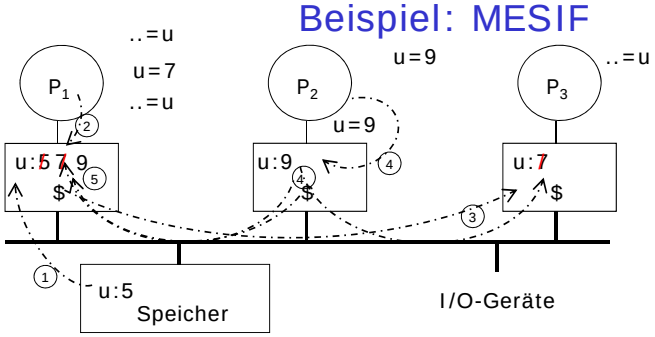
Zustandsübergangsdiagramm MESIF

- Cache-to-Cache Sharing
  Flush im Zustand M, E und F,
  Flush ohne Wirkung auf den Hauptspeicher.

- Write-Back
  BusWB im Zustand M und F,
  Im Zustand F zusätzlich *Inval* aller Teilhaber.



Beispiel: MESIF



(1) P_1 liest u
Cache Miss

(2) P_1 schreibt u

(3) P_3 liest u
Cache Miss

(4) P_2 schreibt u
Cache Miss

(5) P_1 liest u
Cache Miss

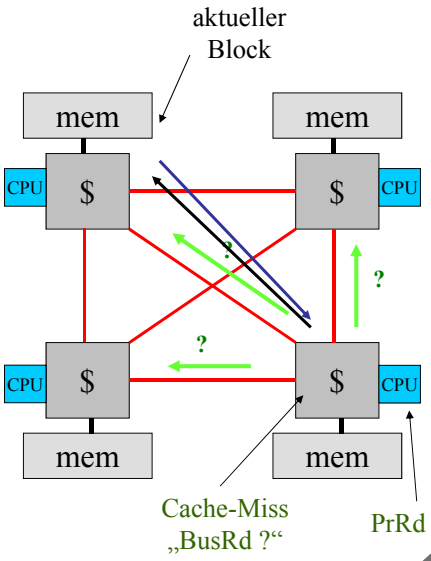
Aktion	*Pr-Aktion*	*Bus-Aktion*	*Signal*	*Quelle*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1	PrRd (1)	BusRd	?		-	-	-
1b			-	Memory	E	-	-
2	PrWr (1)	-	-	-	M	-	-
3a	PrRd (3)	BusRd					
3b		Flush(1)	S	Cache P_1	S	-	F
4a	PrWr (2)	BusRdX					
4b		Flush(3)	-	Cache P_3	I	M	I
5a	PrRd (1)	BusRd					
5b		Flush(2)	S	Cache P_2	F	S	I

Source Snooping

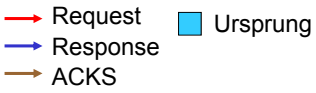
Geht das auch ohne Bus?

Beispiel:
System mit vier Prozessoren

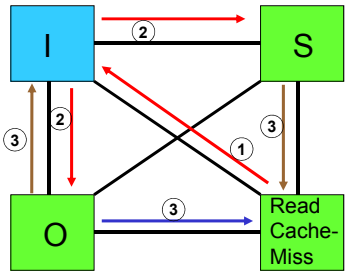
- HyperTransport (AMD HT) oder Quick Path Interconnect (Intel QPI) als Verbindungsstruktur
- Jeder Prozessor mit lokalem Speicher
- Globaler Adressraum verteilt über lokale Speicher



Protokolle ohne Snooping-BUS

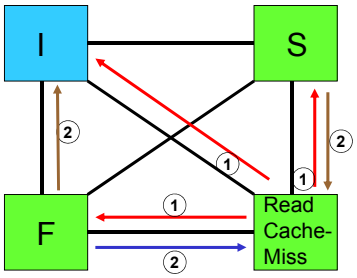


MOESI (z.B. AMD Opteron)



3 Hop Protokoll

MESIF (z.B. Intel Core i7)



2 Hop Protokoll

Update-basierte Protokolle

Invalidierung-basierte Protokolle verschicken *Inval*-Nachrichten

Kann dabei nicht auch der aktuelle Wert bekannt gegeben werden?

=> Update-basiertes Protokoll

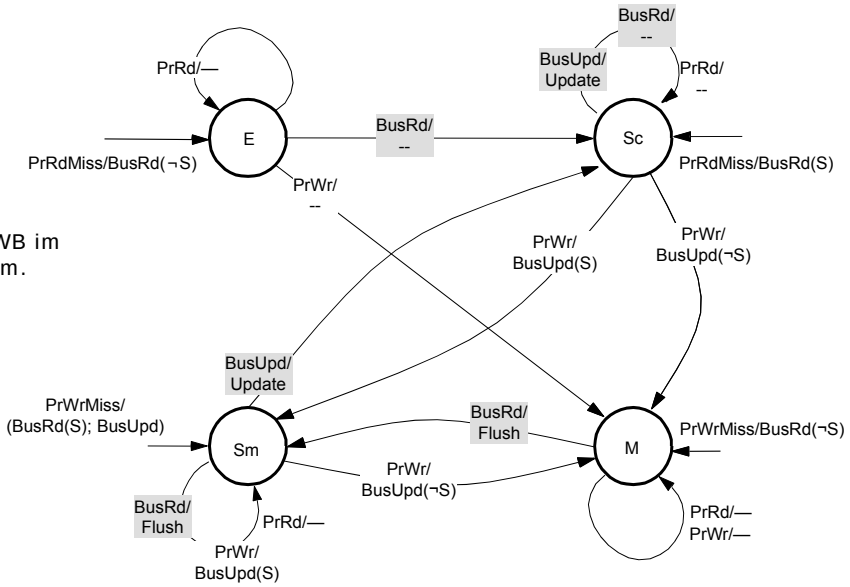
=> Damit kein expliziter Zustand I notwendig!

DRAGON: Update-basiertes Protokoll für Write-Back Caches

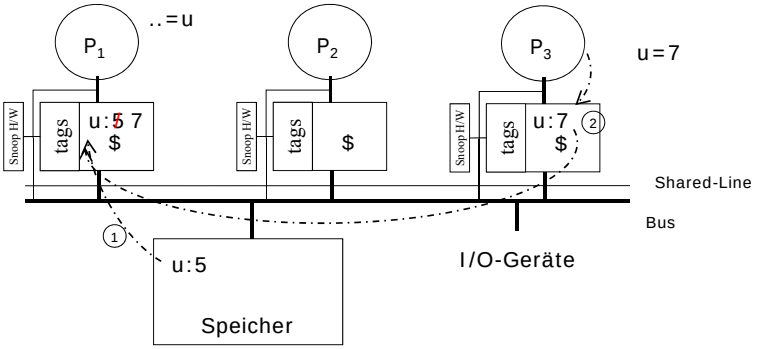
- 4 Zustände
  - *Exclusive-clean (E)*: Man selber und der Hauptspeicher haben aktuellen Wert
  - *Shared-clean (Sc):* Man selber und andere(r) Cache(s) haben den Wert; Hauptspeicher hat aktuellen Wert oder anderer Cache ist Eigentümer
  - *Shared-modified (Sm):* Man selber und andere haben den Wert, aber nicht der Speicher; **man selber ist Eigentümer**
    - *Sm* and *Sc* können gleichzeitig in unterschiedlichen Caches sein (!), aber nur einer mit *Sm*
  - *Modified oder Dirty (M):* Nur man selber hat den Wert
- Bemerke: Es gibt kein *Invalid* Zustand
  - im Cache ⇒ nicht *Invalid* (Protokoll hält Blöcke immer *Valid*)
  - nicht im Cache ⇒ kann als *Invalid* angesehen werden (Bootstrap des Protokolls)
- neue Prozessor-Ereignisse: PrRdMiss, PrWrMiss
  - falls Block nicht im Cache enthalten; Initialisierung des Zustands
- neue Bus-Transaktion (Bus-Update): BusUpd
  - *Broadcast* eines einzelnen Worts über Bus; dadurch *Update* anderer Caches

Dragon Zustandsübergangsdiagramm

- Write-Back: BusWB im Zustand M und Sm.



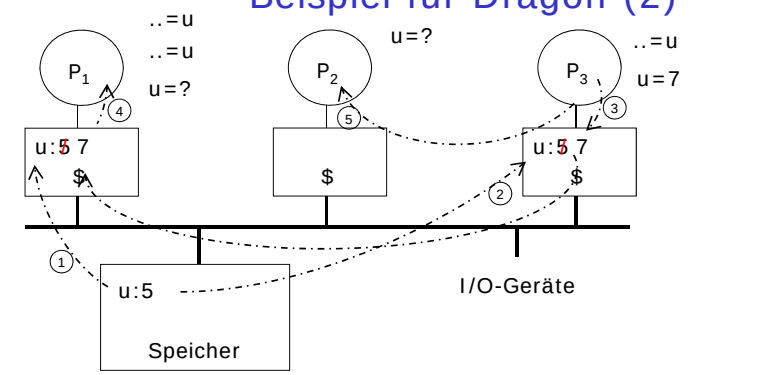
Beispiel für Dragon (1)



(1) P_1 liest *u* aus Hauptspeicher
(2) P_3 beschreibt *u*

	Pr-Aktion	*Bus-Aktion*	*Signal*	*Source*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1a	PrRdMiss (1)	BusRd	?		-	-	-
1b			-	Memory	E	-	-
2a	PrWrMiss (3)	BusRd	?				
2b			S	Memory	Sc	-	Sm
2c		BusUpd		**u** aus Cache P_3	Sc	-	Sm

Beispiel für Dragon (2)



(1) P_1 liest *u* aus Hauptspeicher
(2) P_3 liest *u* aus Hauptspeicher
(3) P_3 beschreibt *u* in Caches
(4) P_1 liest *u* aus **eigenem Cache**
(5) P_2 liest *u* aus P_3 Cache

	Pr-Aktion	*Bus-Aktion*	*Signal*	*Source*	*Zustand in P_1*	*Zustand in P_2*	*Zustand in P_3*
1	PrRdMiss (1)	BusRd	-	Memory	E	-	-
2	PrRdMiss (3)	BusRd	S	Memory	Sc	-	Sc
3	PrWr (3)	BusUpd	S	**u** aus Cache P_3	Sc	-	Sm
4	PrRd (1)	-		-	Sc	-	Sm
5	PrRdMiss (2)	BusRd	S	Cache P_3	Sc	Sc	Sm

Design und Implementierung von

„Snoop-based" Multiprozessorsystemen

Im Folgenden geht es um:

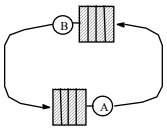
- *nicht-atomare Zustandsübergänge*
- *Multilevel-Caches*
- *Busse mit „Split-Transactions"*

Design Goals

- Leistung und Kosten einer Rechnerarchitektur hängen nicht so sehr von der Wahl des Cache-Kohärenzprotokolls ab, sondern vielmehr von dessen Design und Implementierung.
- Voneinander abhängige Ziele gelten:
  - **Korrektheit** (z.B. wegen nicht-atomaren Aktionen nicht leicht beweisbar)
  - **Hohe Leistung** (z.B. mehrere ausstehende *low-level* Ereignisse ⇒ mehr potenzielle *Bugs*)
  - **Minimale Hardware** (reduzierte Komplexität führt zu kürzerer *time-to-market*)

Korrektheitsfragen

- Erfüllung der Bedingungen für Kohärenz und Konsistenz
  - *CC: Write-Propagierung*, Serialisierung
  - SC: Vollständigkeit, Atomarität
- *Deadlock*: Das System führt keine Aktivität durch
  - Kreis aus Ressource-Abhängigkeiten



- *Livelock*: kein Prozessor schreitet in der Ausführung fort, Transaktionen werden aber ausgeführt
  - Z.B. simultane Schreiboperationen im „invalidation-based“ Protokoll
    - jeder fragt nach *Ownership*, invalidiert andere, verliert dieses bevor Arbitrierungsrecht auf Bus zugeteilt wurde
- *Starvation*: Ein oder mehrere Prozessoren schreiten in ihrer Ausführung nicht fort, während aber andere weiterkommen.
  - z.B. *interleaved Memory* System mit *NACK* bei *Bank-Busy*
  - oftmals nicht vollständig auszuschließen (sehr geringe Wahrscheinlichkeit, damit nicht katastrophal)

Einfache Cache-Kohärenz-Designs

Im Folgenden wird vorausgesetzt:

- „Single-level write-back” Cache
- Invalidierung-basierte Protokoll
- Max. eine ausstehende Speicheranfrage pro Prozessor
- Atomare Speicherbus-Transaktionen
  - BusRd, BusRdX: Keine Unterbrechen der Bus-Transaktionen während Adressübertragung und Datenempfang
  - BusWB: Gleichzeitige Bearbeitung der Adresse / Daten vom Speichersystem bevor eine neue Busanfrage bearbeitet wird
- Atomare Operationen im Prozessor
  - Eine Operation endet bevor nächste in Programmfolge startet

Einige Fragestellungen zum Design

- Design des Cache-Controllers und der Tags?
  - sowohl Prozessor- als auch bus-seitig Zugriff auf die Tags und Cache-Lines
- Wie und wann werden Snoop-Resultate auf den Bus gelegt?
- Wie wird Write-Back behandelt?
- Ggf. nicht alle Aktionen der Speicheroperationen sind atomar - kann Race-Conditions zur Folge haben
- Neue Probleme mit Deadlock, Livelock, Starvation, Serialisierung, usw.
- Implementierung atomarer Operationen
  - Z.B. Read-Modify-Write

Betrachten wir nun eins nach dem anderen ...

Cache-Controller und Tags

- Cache-Controller arrangiert Teile der auszuführenden Operation
  - ist selber endlicher Automat (aber nicht identisch mit dem Automat des Protokolls)
- Im Einprozessor-System (während eines *Cache-Misses*)
  1. Erbitten um Bus-Zugang
  2. Warten auf Bus-Zuteilung
  3. Steuerung von Adressen und Kommando auf dem Bus
  4. Warten auf Annahme des Kommandos der zuständigen Einheiten
  5. Transfer der Daten
- Im *Snooping*-basierten Multiprozessorsystem muss ein Cache-Controller:
  - Bus und Prozessor überwachen
    - kann als zwei Controller angesehen werden; bus- und prozessor-seitig
    - *Single-Level*-Cache: doppelte *Tags* (nicht Daten) oder *dual-ported Tag RAM*
      - muss beim „*Update*“ vereinigt werden, aber meiste Zugriffe *Look-Ups*
  - falls notwendig auf Bus-Transaktionen antworten

Übermittlung der Snoop-Resultate: Wie?

Ein gemeinschaftliches Antworten der Caches muss über den Bus organisiert werden

Beispiel: im MESI-Protokoll muss festgestellt werden, ob Block

- *Dirty (Modified)*; muss ggf. der Speicher antworten?
- *Shared*; Übergang zum E oder S Zustand beim *Read-Miss*?

Lösung: Drei verdrahtete ODER-Signale

1. Draht: ***Shared*** - gesetzt, falls irgendein Cache eine Kopie hat
2. Draht: ***Dirty*** - gesetzt, falls ein Cache eine „dirty"-Kopie hat
3. Draht: ***Snoop-valid*** - gesetzt, wenn beide anderen Signale gültig

- Illinois MESI erfordert ein priorisiertes Schema für Cache-to-Cache Transfers
  - Welcher Cache soll im *Shared*-Zustand die Daten liefern?
    - einige Systeme erlauben dem Speicher die Daten zu liefern
    - ... oder gleich MOESI bzw. MESIF

Übermittlung der Snoop-Resultate: Wann?

Der Speicher muss wissen was ggf. getan werden muss

- Adresse muss bestimmte Taktanzahl auf dem Bus liegen
  - *Dual-Tags* erforderlich, um potentielle Konflikte mit Prozessor zu vermeiden
  - muss konservativ sein (*Update* beider *Tags* beim *Write*: E→M)
- variable Taktanzahl
  - Speicher nimmt solange an, dass Cache Daten liefert, bis alle Cache-Controller „*Sorry*" sagen
  - weniger konservativ und flexibler, aber komplexer
  - Speicher kann Daten präventiv holen
- Direkte Methode: zusätzliches Status-*Bit pro Block* im Speicher
  - Valid/Invalid-Bit kann ggf. in den Parity/Fehlerkorrektur-Bits des Speichers kodiert werden
  - erhöht Hardware-Komplexität des Speicher-Interfaces