



Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza



Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza



Tema 12 – Sincronización

Multiprocesadores

3er curso, Grado Ingeniería Informática
Especialidad Ingeniería Computadores

V. Viñals y J. Alastruey
Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas

Guión del tema

- Shared-memory communication, interleaving and synchronization
 - Mutual exclusion
 - Barrier and point-to-point synchronization
- Synchronization
 - Acquire, release and waiting method
 - ISA support
 - T&S y F&A examples
 - Traffic reduction: T&T&S
 - Load-Linked and Store-Conditional
- LL-SC, T&S and F&A implementation
 - F&A *combining* in Multistage Interconnection Networks



Mucho material tomado de:

“Parallel Computer Organization and Design”.

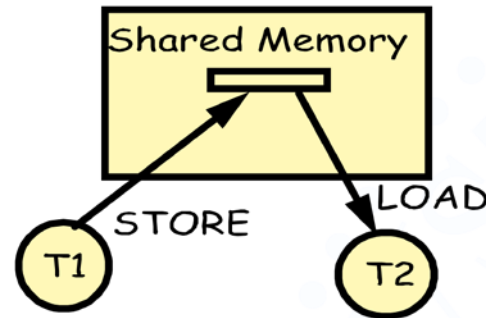
M. Dubois, M. Annavaram and P. Stenström.

Cambridge University press 2012.

Slides, Chapter 7. *Coherence, synchronization and memory consistency.*

Shared-memory communication, interleaving and synchronization

- Processors share memory,
so communication is implicit through **loads** and **stores**



*no assumption
on the relative speed
of threads T1 and T2*

- Programmers of parallel applications need to know:
 - thread managing** see for instance POSIX interface: create-start, exit-join, ...
 - consistency** how loads and stores from different threads to *different* addresses, interleave accesses
 - synchronization**
 - ... already required in old, time-sharing multiprogramming:
 - to acquire a physical resource before using it
 - to access a shared variable in an exclusive way → **mutual exclusion**
 - to signal that computation has reached a given point

Synchronization needs: Mutual Exclusion

- Assume threads, **T1** and **T2** executing the following *statements*

<u>T1</u>	<u>T2</u>
A = A+1	A = A+1

- The programmer's expectation could be that, **A** will be incremented by 2,
... however, program *statements* are not executed in an atomic fashion
- Compiled code on a RISC machine will include several instructions
 - a possible interleaving is:

<u>T1</u>	<u>T2</u>
r1 = A	
	r1 = A
r1 = r1 + 1	
	r1 = r1 + 1
A = r1	
	A = r1

we must have a way to make
program statements
"A = A + 1"
appear atomic

- at the end, A increases by 1, NOT 2 !

C11 solution

```
#include <stdatomic.h>

#define ATOMIC_VAR_INIT(value)    /* implementation defined*/

...

_atomic int A = ATOMIC_VAR_INIT(0);
```

T1

A++;

T2

A++;

INCITS/ISO/IEC
9899-2011 [2012]

American National Standard

INCITS/ISO/IEC 9899-2011 [2012]
(ISO/IEC 9899:2011, IDT)

Information technology — Programming
languages — C

Developed by



Where IT all begins





Mutual exclusion

- When several threads execute code intended to access to a shared resource or variable, those codes should never execute at the same time, but in **mutual exclusion**
- We call “**critical sections**” to such codes
 - low-level implementation: place the **critical section** within a **lock-unlock** frame
 - lock-unlock works with a memory location **L** that represents the **availability** of the shared resource or variable

<u>T1</u>	<u>T2</u>	
lock (L)	lock (L)	/ acquires lock L
A = A+1	A = A+1	/ critical section
unlock (L)	unlock (L)	/ releases lock L

- Pre-DEKKER’S algorithm for mutual exclusion → binary flags **f1** and **f2** are both **0** initially

<u>T1</u>	<u>T2</u>	
f1 = 1	f2 = 1	/ acquire
while(f2==1)  ;	while(f1==1)  ;	
<critical section>	<critical section>	
f1 = 0	f2 = 0	/ release

But

- deadlock may arise → See Annex at the end of lecture
- not scalable: more complex to solve deadlock and synchronize with more threads
- only works with ISAs offering **sequential consistency**

➡ use ISA instructions to synchronize !

C11 Solution

```
#include <threads.h>
```

```
...
```

```
int A;
```

```
...
```

T1

```
mtx_lock(&L);
```

```
A++;
```

```
mtx_unlock(&L);
```

T2

```
...
```

```
mtx_lock(&L);
```



```
A++;
```

```
mtx_unlock(&L);
```


Barrier and point-to-point synchronization

Barrier sync. = global synchronization among multiple threads:


ALL threads must reach the barrier before **ANY** thread is allowed to execute beyond the barrier


<u>T1</u>	<u>T2</u>	
...	...	
BAR= BAR + 1;	BAR= BAR + 1;	/ requires a counter
while (BAR<2)  ;	while (BAR<2)  ;	

- need a critical section to increment BAR
 - no critical section to read BAR in the while statement
 - In practice, more complex because barrier count must be reset for the next use

Point-to-point synchronization:

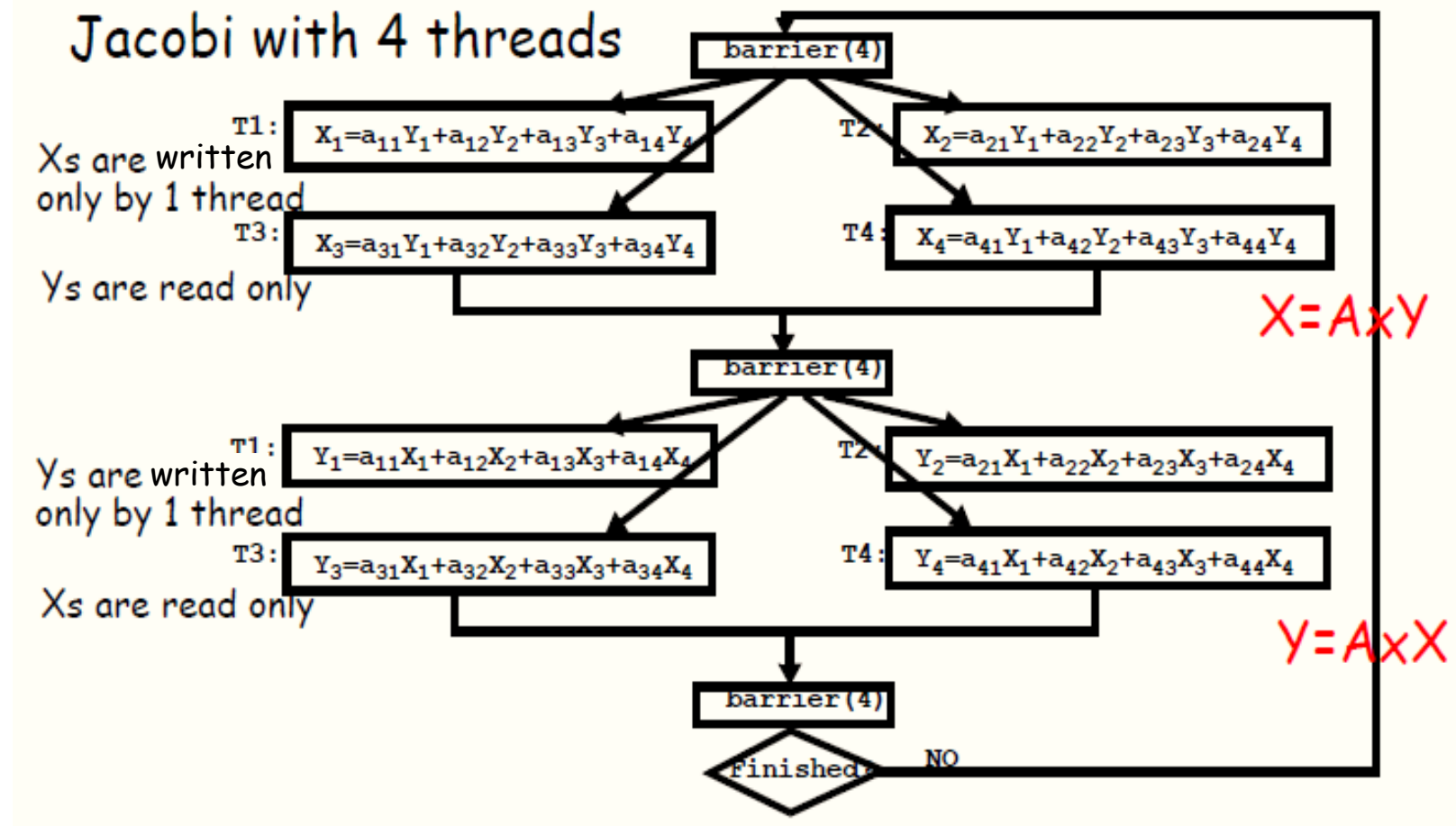
to signal the computation has reached some point

<u>T1</u> (cons)	<u>T2</u> (prod)	A = 0 initially
...	A = 1;	/ requires a binary flag
...	FLAG = 1;	/ signal
while (FLAG==0)  ;		/ wait
print A		



- no critical sections to update and read FLAG, but T1 receiving T2 stores in \leq_p is required ([sequential consistency](#))

Sync. Barrier example



- in 1st phase accesses to X_i 's are mutually exclusive
- in 2nd phase, multiple accesses to X_i 's (read-only)
- opposite is true for Y_i 's

Guión del tema

- Shared-memory communication, interleaving and synchronization
 - Mutual exclusion
 - Barrier and point-to-point synchronization
- Synchronization
 - Acquire, release and waiting method
 - ISA support
 - T&S y F&A examples
 - Traffic reduction: T&T&S
 - Load-Linked and Store-Conditional
- LL-SC, T&S and F&A implementation
 - F&A combining in Multistage Interconnection Networks



Lock synchronization overview

- **ACQUIRE method:** gets the right to access the shared data
 - issues the critical section
- **RELEASE method:** enables other threads to acquire that right
- **WAITING algorithm in Acquire:** using the processor or not
 - **busy-waiting**
 - waiting processors repeatedly **tests** a location **L**: **L == busy?** (L == locked?)
 - the releasing thread **sets** the location **L**: **L = free** (L = open)
 - low latency but holds the processor
 - may have a high memory/network traffic
 - **blocking**
 - waiting threads are de-scheduled at runtime
 - high latency
 - allows processors to work on something else
 - **busy-waiting** is better than **blocking** when
 - scheduling overhead is larger than expected waiting time, or
 - no other task to run, or
 - executing an OS kernel thread

ISA support to synchronization (1)

- Since the beginning, general purpose processors have some form of **atomic Read-Compare-Modify-Write** instructions
- For instance, Motorola 680xx, CISC, 32 bits
 - 1979-1994, xx = 00-10-20-30-40-60
 - 1995-2015 Freescale, 2016 - NXP Semiconductors
- test & set: **TAS mem_{8bits}** **atomic RW**
mem_{8bits} – 0; set condition codes; mem_{8bits} = 1
- compare & swap: **CAS rc, rs, mem** (8, 16, 32 bits) **atomic RCW**
mem – rc; set condition codes; if CC.Z then mem = rs else rc = mem

The “Motorola M68000 Family Programmer’s Reference Manual” says:

... **CAS** provides a means of updating system counters, history information, and globally shared pointers. The instruction uses an indivisible **[read-compare-write]** cycle. After **CAS** reads the memory location, no other instruction can change that location before CAS has written the new value. This provides security in single-processor systems, in multitasking environments, and in multiprocessor environments. In a single-processor system, the operation is protected from instructions of an interrupt routine. In a multitasking environment, no other task can interfere with writing the new value of a system variable. In a multiprocessor environment, the other processors must wait until the CAS instruction completes before accessing a global pointer.

Recall: incorrect mutual exclusion

```
L:      .DW      0                / 0=free
lock:   ...      ↖
        LW       R1,L
        BNZ      lock,R1
        SW       L,#1            / 1=busy
        RET
unlock: SW       L,#0
        RET
```

- PROBLEM: the **lock access** is not atomic – two threads can gain the lock at once
- SOLUTION:
 - atomically read L and set it to another value
 - return either success or failure
- The simplest one: **T&S L,R1**
 - reads L and moves to R1 (*test*), and writes 1 into L (*set*)
 - **success** value in R1 is 0 → RET
 - **failure** value is 1 → try again !

must be atomic !

T&S mutex and other R/M/W operations

```

L:      .DW      0          / 0=free
        ...
lock:   T&S      R1,L       / spin
        BNZ      lock,R1   / loop
        RET
unlock: SW      L,#0
        RET

```

- Other atomic operations

- Swap **swap** R1, mem RW
exchange contents between R1 and mem
- Fetch&op **F&ADD** R1, mem, small_const RMW
fetch mem in R1, then add small_const to mem
 - proposed first in MIN multis: NYU Ultracomputer (1983), IBM RP3 (1983)
 - present in MIPS, Intel-x86, Berkeley RISC-V

One of the F&A original purposes

- Self-scheduling, aka dynamic scheduling
 - ex1: threads add and retrieve ready tasks in a FIFO queue
 - **ex2: threads get the next unexecuted iteration index in a parallel loop**
 - same code on different data, and execution time of each iteration different and unknown

```
do i=1, N
    body(i)          / parallel
enddo
```

```
count:      .DW      1          / initially, first iteration
...
par_body:   F&A      R1,count,1
            while R1 ≤ N
                body(R1)
                F&A      R1,count,1
            end while
```


ISA support to synchronization (2)

Atomic Instruction	cmp&swap	XCHG r, m (8, 16, 32 bits)	bit test&set	LOCK prefix any instruction can carry it	load-linked, store- conditional	fetch&op
Read Compare Write Modify	RCW	RW	RW		R ... CW	RMW
IBM 370	✓					
IBM zSeries	✓					
IBM Power	✓				✓	
x86 (Intel, AMD)	✓	✓	✓	✓		✓
Intel IA-64 (Itanium)					✓	
Alpha (DEC, Compaq)					✓	
MIPS					✓	✓ not cacheable
SPARC (Sun-Oracle, Fujitsu)	✓					
ARM					✓	
Berkeley RISC-V					✓	✓

- Very recently, ISA support to **Transactional Memory**

Guión del tema

- Shared-memory communication, interleaving and synchronization
 - Mutual exclusion
 - Barrier and point-to-point synchronization
- Synchronization
 - acquire, release and waiting method
 - ISA support
 - T&S y F&A examples
 - Traffic reduction: T&T&S
 - Load-Linked and Store-Conditional
- LL-SC, T&S and F&A implementation
 - F&A combining in Multistage Interconnection Networks



Reduce frequency of issuing T&S

- T&S with *backoff* (*retirada*)
 - Insert a delay after every failure in the spin loop
 - Increase that delay between successive trials
 - e.g. exponential backoff for thread number k
 - back off by $k \cdot c^i$ at the i th trial
- TEST and TEST&SET lock (T&T&S)
 - first, Test with ordinary load
 - when value changes to 0, try to acquire lock with T&S

```
lock:      LW      R1, L
           BNEZ    R1, lock
           T&S     R1, L
           BNEZ    R1, lock
           RET

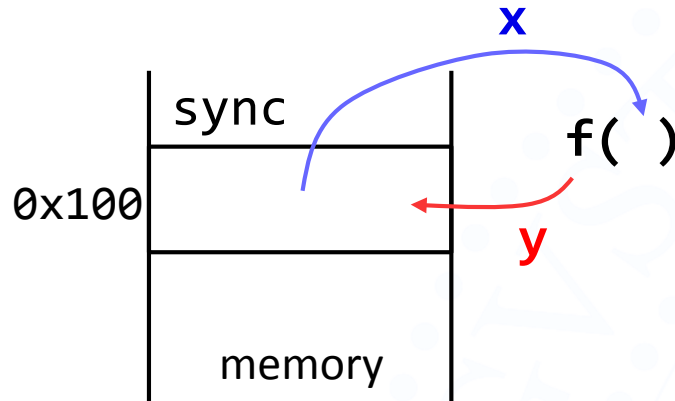
unlock:    SW      L, R0
           RET
```

- specially suited for decreasing coherence traffic with private caches:
failed attempts generate few cache invalidations ♦♦

Load Linked and Store Conditional (1)

Puntero a la variable **sync**

p = 0x100



We want atomic RMW !

$x = *p;$ $y = f(x);$ $*p = y$ *conditionally*
R M W, or not

*failed attempts **do not** generate
cache invalidations ♦♦*

Q: How many interleavings are possible among $R_1M_1W_1$ from thread 1 and $R_2M_2W_2$ from thread 2 ?

A: $20 = 6! / (3! \times 3!)$

Load Linked and Store Conditional (2)

$x = *p;$ $y = f(x);$ $*p = y$ *conditionally*

- **Load Linked**
(locked, reserved, exclusive)
 - reads mem and starts tracking whether any other thread writes to address p
- **Store Conditional** (exclusive)
 - Writes mem **only if** the tracked address p has not been written since tracking started
- implementation dependent
 - 1 track per CPU, per thread per cache line, ...
- Code in $f(x)$ has to be designed carefully
 - large code = higher chance of conflict
 - writes in $f(x)$ are unconditional
 - only the final store $*p = y$ is conditional
- Forward Progress ?
 - not at the HW level
 - requires algorithm support



Load Linked and Store Conditional (3)

Thread 1 and Thread 2:

```
_Atomic(int) *p;
...
(*p)++;
```

```
do
{  int x = LL(p);
  int y = x + 1;}

while(SC(p,y));
```

SC p, R1, R2

memory
address

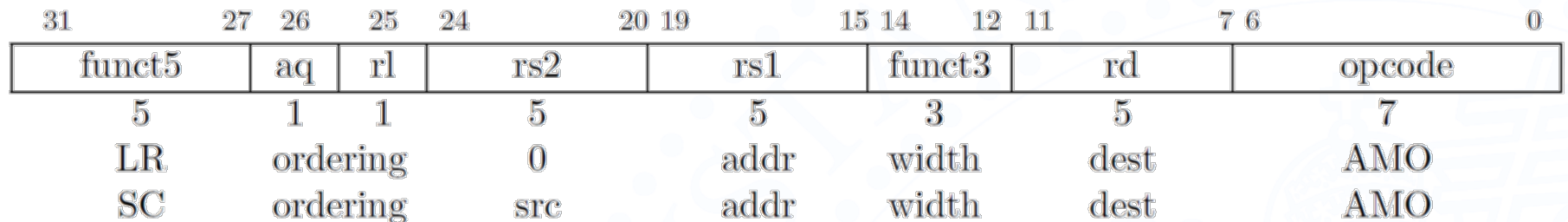
value
to write

condition
register:
0 → success
≠ 0 → fail

Thread 1:	Thread 2:	*p
LL R1,p		0
add R1,1		0
	LL R1,p	0
SC p,R1,R2 <i>escribe</i>		1
bnz R2 <i>not taken</i>		1
	add R1, 1	1
	SC p,R1,R2 <i>no escribe</i>	1
LL R1,p		1
add R1,1		1
SC p,R1,R2 <i>escribe</i>	bnz R2 <i>taken</i>	2
bnz R2 <i>not taken</i>	LL R1,p	2
	add R1,1	2
	SC p,R1,R2 <i>escribe</i>	3
	bnz R2 <i>not taken</i>	3

Load Linked and Store Conditional: RISC-V (4)

5.2 Load-Reserved/Store-Conditional Instructions



- Complex atomic memory operations on a single memory word are performed with the load-reserved (LR) and store-conditional (SC) instructions.
- LR loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a reservation on the memory word.
- SC writes a word in *rs2* to the address in *rs1*, provided a valid reservation still exists on that address. SC writes zero to *rd* on success or a nonzero code on failure, e.g. a conflicting memory access occurred or there was an intervening context switch.

Ejercicio 12.1

- Funciones Lock(L) y Unlock(L) en RISC-V para proteger una sección crítica mediante consulta continua (*spinning*) en la variable L

Comparison

LL - SC

- MIPS, Intel IA-64, IBM Power, ARM, RISC-V, ...
- Easiest to implement in hardware
 - **load** is a single instruction, **store** is a single instruction
- Sync failures does not trigger cache invalidations

CAS

- x86, Itanium
- Load and Store in one instruction
 - Problem for simple pipelined machines

Fetch & Op

- MIPS, RISC-V
- Allows request combining
 - IBM Blue Gene
- Load and Store in one instruction
 - Problem for simple pipelined machines

Both LL-SC and compare-and-swap (CAS) can be used to build *lock-free data structures*

www.drdobbs.com/lock-free-data-structures/184401865

Guión del tema

- Shared-memory communication, interleaving and synchronization
 - Mutual exclusion
 - Barrier and point-to-point synchronization
- Synchronization
 - acquire, release and waiting method
 - ISA support
 - T&S y F&A examples
 - Traffic reduction: T&T&S
 - Load-Linked and Store-Conditional
- LL-SC, T&S and F&A implementation
 - F&A combining in Multistage Interconnection Networks



Implementing LL-SC

- LL-bit and LL-register per core
 - LL-bit *set* means exclusive access
 - LL-bit *reset* means a write to the LL address appeared on the bus i.e. a “conflicting write”
 - LL-register keeps the L address
- LL executes: set LL-bit and record L address in LL-register
- SC executes:
 - if the LL-bit is still set,
perform the write transaction and reset LL-bit
 - otherwise the SC fails and the write is not performed
- between LL and SC,
the LL-register address is matched against write addresses on the bus
if hit, reset LL-bit
- context switches also reset LL-bit
- works the same for circuit or packet switching

Implementing T&S and F&A (1)

- Circuit switching

- maintain BR during all the required accesses:
“lock” the bus, crossbar path, etc.
- T&S: BR high - read, write - BR low
- F&A: BR high - read, add [in the CPU], write - BR low

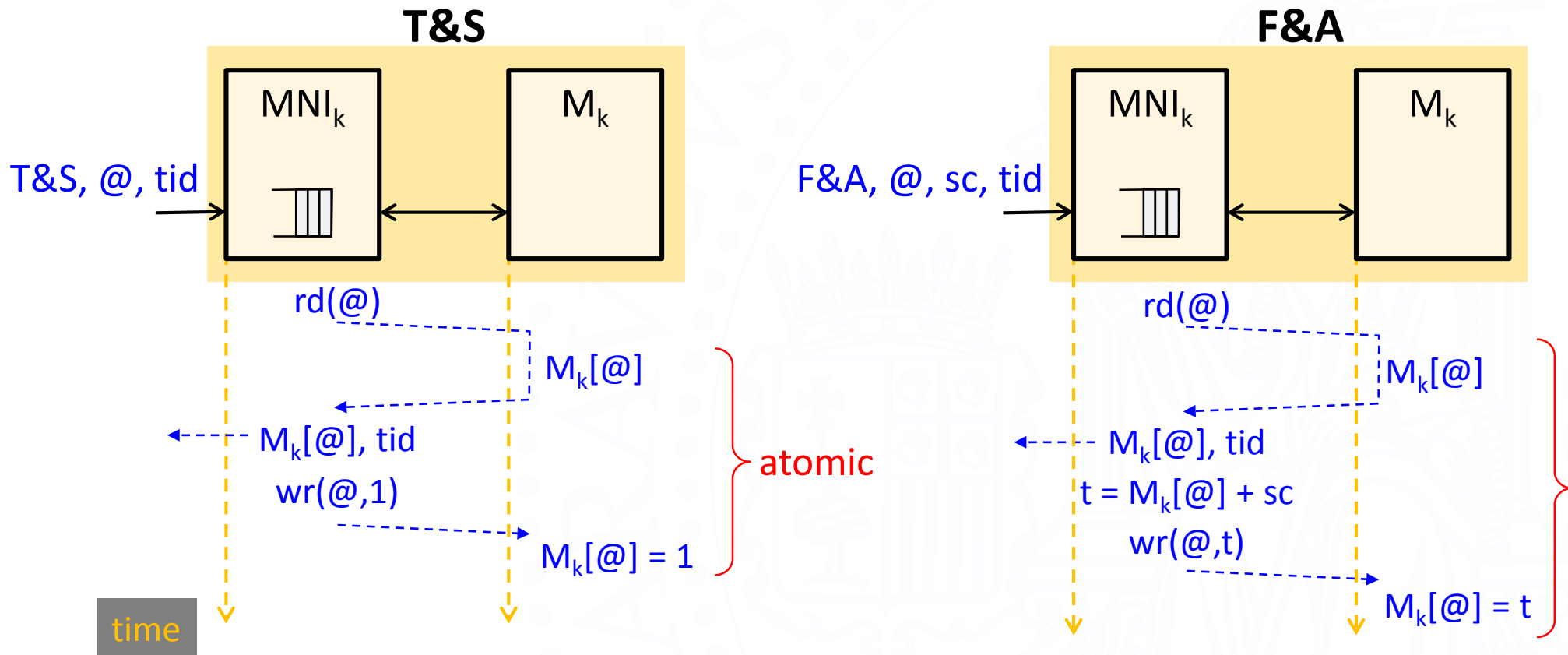
- Packet switching

- we don't want to lock paths or memory banks ...
- forget for the moment the existence of private caches
- Key idea: we increase the functionality of memory banks
they not only accept load and store requests
but also synch. primitives

processors issue synch. primitives without breaking them
into individual loads and stores → → →

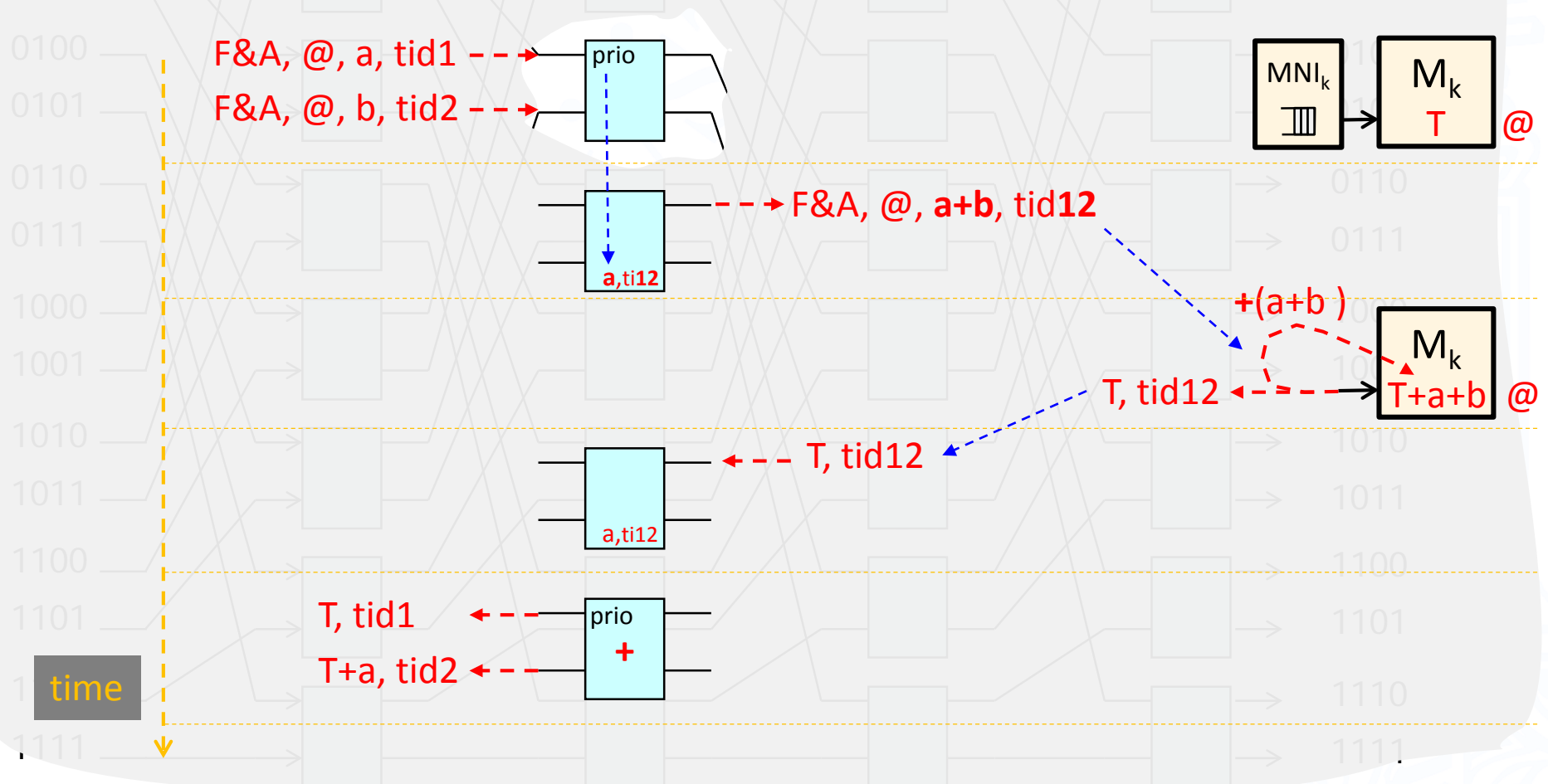
Implementing T&S and F&A (2)

- Packet switching
processors issue sync. primitives without breaking them
- The new functionality goes to a new front-end module called Memory Network Interface (MNI)



F&A combining in MIN nets. (NYU Ultracomputer, Gottlieb 83)

- Self-scheduling (parallel loops, OS scheduler) often produces burst of F&As directed to the same shared counter
- Having a multistage IN we can add functionality to the switches and reduce contention and traffic a lot by **combining F&A requests**



Advanced material: transactional memory (see Dubois et al. Book)

```
1 struct Shared{
2     int item1;
3     int item2;
4 } myShared;
6 Producer(){
7     lock(1);
8     if(cond1)
9         myShared.item1++;
10    else
11        myShared.item2++;
12    unlock(1);
13 }
14 Consumer(){
15     lock(1);
16     if((cond2)&&(myShared.item1>0))
17         myShared.item1--;
18     if((!cond2)&&(myShared.item2>0))
19         myShared.item2--;
20    unlock(1);
21 }
```

Using Locks

```
1 struct Shared{
2     int item1;
3     int item2;
4 } myShared;
6 Producer(){
7     Transaction_Begin;
8     if(cond1)
9         myShared.item1++;
10    else
11        myShared.item2++;
12    Transaction_End;
13 }
14 Consumer(){
15     Transaction_Begin;
16     if((cond2)&&(myShared.item1>0))
17         myShared.item1-- ;
18     if((!cond2)&&(myShared.item2>0))
19         myShared.item2-- ;
20    Transaction_End;
21 }
```

Using Transactions

Annex: Dekker's algorithm

- First attempt for mutual exclusion
 - binary flags f1 and f2 are both 0 initially (not busy)

T1

f1 = 1;

while (f2==1)

<critical section>

f1 = 0;

T2

f2 = 1: / acquire

while (f1==1)

<critical section>

f2 = 0; / release

- deadlock may arise

Dekker's algorithm, second attempt

- to avoid *deadlock*, let a thread give up the reservation (i.e. T1 sets **f1** to 0) while waiting

T1

```
f1 = 1;
```

```
while ( f2==1 )
```

```
{f1 = 0; f1 = 1};
```

```
<critical section>
```

```
f1 = 0;
```

T2

```
f2 = 1;
```

```
while ( f1==1 )
```

```
{f2 = 0; f2 = 1};
```

```
<critical section>
```

```
f2 = 0;
```

```
/ acquire
```

```
/ release
```

- Deadlock is not possible,
but with a low probability a *livelock* may occur
- An unlucky thread may never get to enter the critical section
⇒ *starvation*

A Protocol for Mutual Exclusion, T. Dekker, 1966

- Proved correct by *E. W. Dijkstra* in 1968
- A protocol based on 3 shared variables **f1**, **f2** and turn

T1

f1 = 1

turn = 1

while

(**f2**==1 & turn ==1);

<critical section>

f1 = 0

T2

f2 = 1

turn = 2

while

(**f1**==1 & turn ==2);

<critical section>

f2 = 0

/ release

- turn = *i* ensures that only process *i* can wait ,
the last entering the while
- flags **f1** and **f2** ensure *mutual exclusion*.
Solution for *n* threads was given by Dijkstra and is quite tricky!

Notes

- Indeed Dekker's algorithm don't require special ISA support, but may need *fences* if the multiprocessor does not follow *sequential consistency*
- Don't scale
- Complex design
- Complex prove of correctness