Multi-Processor Systems

Pirouz Bazargan Sabet

Sorbonne Université - LIP6

Pirouz.Bazargan-Sabet@lip6.fr

Outline

- The Problem of Shared Resources
- Atomic Operations

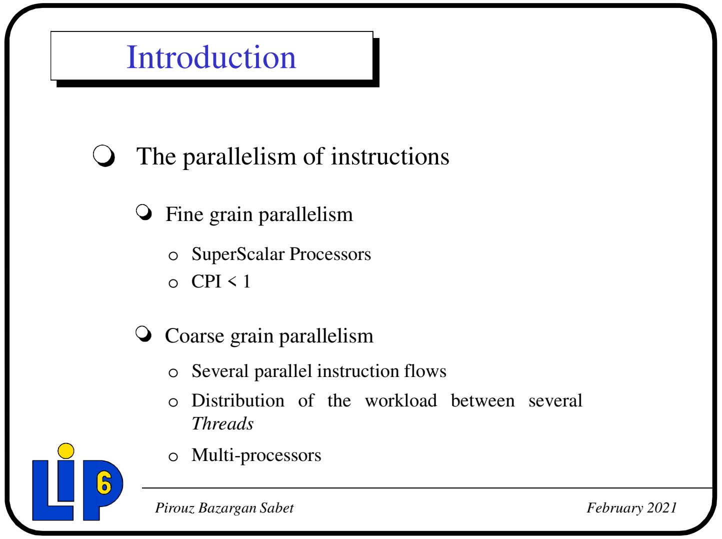
Introduction

Performance

- To enhance the performance of a system (execution time) we can play with 2 parameters
  - The processor's frequency
    - Pipeline
  - The number of cycles required to execute an operation

Introduction

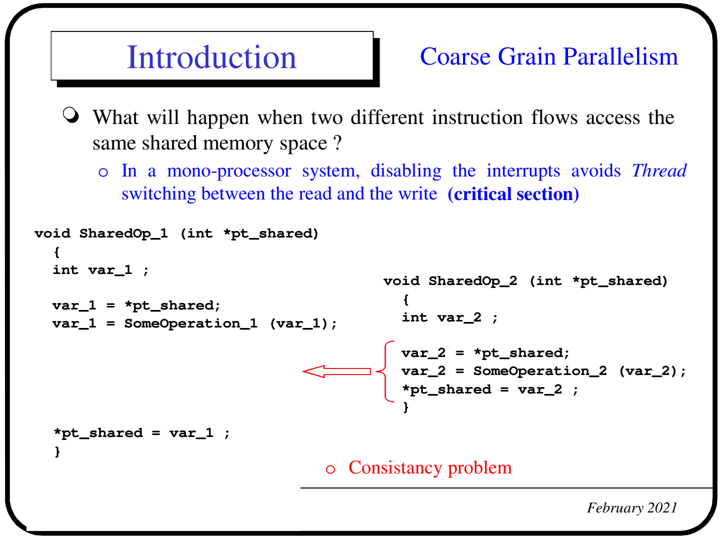
- To reduce the number of cycles necessary to execute an operation, we should increase the *parallelism*
  - The parallelism on processing data
    - 4 bits (1970's)
    - 8 bits (1970's)
    - 16 bits (1980's)
    - 32 bits (1990's)
    - 64 bits (2000's)
    - 128 bits (?)
  - The parallelism of instructions



Introduction

- The parallelism of instructions
  - Fine grain parallelism
    - SuperScalar Processors
    - CPI < 1
  - Coarse grain parallelism
    - Several parallel instruction flows
    - Distribution of the workload between several *Threads*
    - Multi-processors

Pirouz Bazargan Sabet *February 2021*



Introduction

Coarse Grain Parallelism

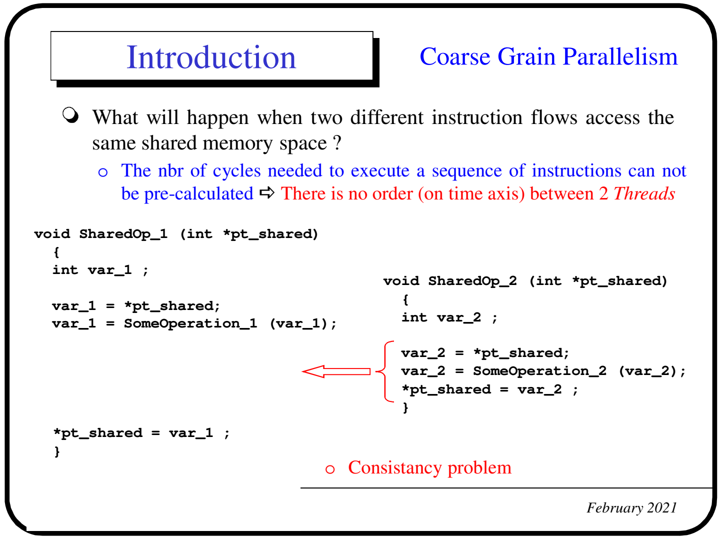
- What will happen when two different instruction flows access the same shared memory space ?
  - In a mono-processor system, disabling the interrupts avoids *Thread* switching between the read and the write **(critical section)**

```
void SharedOp_1 (int *pt_shared)
  {
  int var_1 ;

  var_1 = *pt_shared;
  var_1 = SomeOperation_1 (var_1);




  *pt_shared = var_1 ;
  }
```

```
void SharedOp_2 (int *pt_shared)
  {
  int var_2 ;

  var_2 = *pt_shared;
  var_2 = SomeOperation_2 (var_2);
  *pt_shared = var_2 ;
  }
```

- Consistancy problem

February 2021



Introduction

Coarse Grain Parallelism

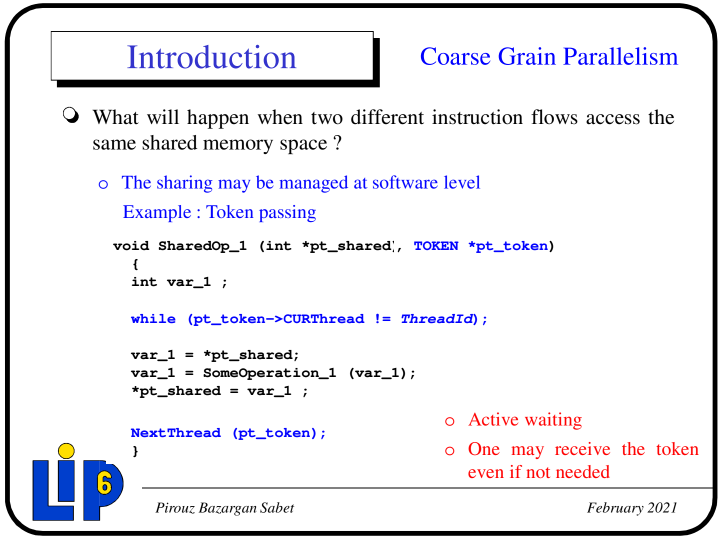
- What will happen when two different instruction flows access the same shared memory space ?
  - The nbr of cycles needed to execute a sequence of instructions can not be pre-calculated ⇨ There is no order (on time axis) between 2 *Threads*

```
void SharedOp_1 (int *pt_shared)
  {
  int var_1 ;

  var_1 = *pt_shared;
  var_1 = SomeOperation_1 (var_1);




  *pt_shared = var_1 ;
  }
```

```
void SharedOp_2 (int *pt_shared)
  {
  int var_2 ;

  var_2 = *pt_shared;
  var_2 = SomeOperation_2 (var_2);
  *pt_shared = var_2 ;
  }
```

- Consistancy problem

February 2021



Introduction

Coarse Grain Parallelism

- What will happen when two different instruction flows access the same shared memory space ?
  - The sharing may be managed at software level

    Example : Token passing

```
void SharedOp_1 (int *pt_shared), TOKEN *pt_token)
  {
  int var_1 ;

  while (pt_token->CURThread != ThreadId);

  var_1 = *pt_shared;
  var_1 = SomeOperation_1 (var_1);
  *pt_shared = var_1 ;

  NextThread (pt_token);
  }
```

- Active waiting
- One may receive the token even if not needed

Pirouz Bazargan Sabet *February 2021*

Introduction

Coarse Grain Parallelism

- What will happen when two different instruction flows access the same shared memory space ?
  - Hardware support : Atomic Operations
    - In a mono-processor system :

      A sequence of *Read-Modify-Write* instructions at an address A_0 is said *Atomic* if it cannot be interrupted

    - In a multi-processor system :

      During the execution of a sequence of *Read-Modify-Write* by a processor P_i at an address A_0 the address A_0 should not receive other writes from another processor $P_{j\ (j \neq i)}$

Introduction

Coarse Grain Parallelism

- What will happen when two different instruction flows access the same shared memory space ?
  - Hardware support : Atomic Operations
    - Swap
    - Compare and Swap
    - Linked Load and Store Conditional

Atomic Operations

Swap : exchanges the contents of a register and a word

- The *Swap* instruction allows to set up a *lock* that protects the accesses to a shared memory space

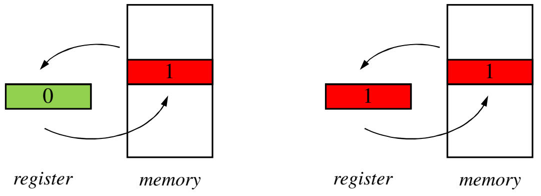
Atomic Operations

Swap : exchanges the contents of a register and a word

- The word in the memory may contain 0 or 1
  - 0 : Access allowed : the shared memory space in not being used
  - 1 : Access denied : The shared memory space is in use
- The register contains always 1

Atomic Operations

Swap : exchanges the contents of a register and a word



getting the *lock :* Successful

getting the *lock* : Failed

Atomic Operations

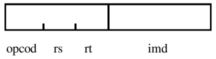
Swap : exchanges the contents of a register and a word

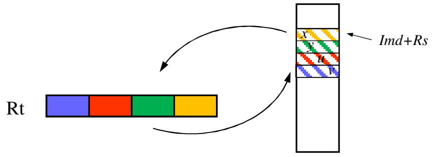
```
void Lock (int *pt_lock)
  {
  register int r = 1;

  while ((r = Swap (r, pt_lock)) != 0)
    {
    r = 1 ;
    Wait ... ;
    }
  }
```

The *Lock* is usually *uncachable* but not necessarily

Atomic Operations

Swap

Assembly language	Action	Format
Swap Rt, Imd (Rs) Swap words	4 bytes from memory at address *Imd+Rs* is exchanged with Rt	 I format



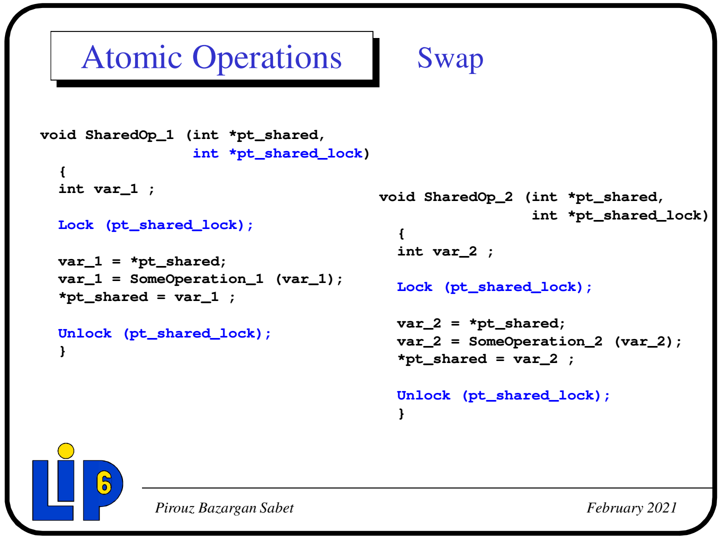
Atomic Operations

Swap : exchanges the contents of a register and a word

```
void Lock (int *pt_lock)
  {
  register int r = 1;

  while ((r = Swap (r, pt_lock)) != 0)
    {
    r = 1 ;
    while (*pt_lock != 0) Wait ... ;
    }
  }
```

A *cachable Lock* avoids overloading the bus

On the other hand the *Swap* is necessarily *uncachable*



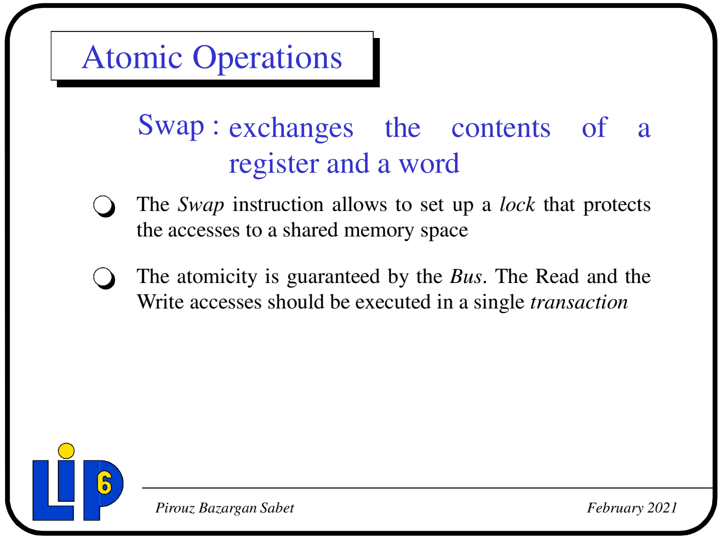
Atomic Operations

Swap

```
void SharedOp_1 (int *pt_shared,
                 int *pt_shared_lock)
  {
  int var_1 ;

  Lock (pt_shared_lock);

  var_1 = *pt_shared;
  var_1 = SomeOperation_1 (var_1);
  *pt_shared = var_1 ;

  Unlock (pt_shared_lock);
  }
```

```
void SharedOp_2 (int *pt_shared,
                 int *pt_shared_lock)
  {
  int var_2 ;

  Lock (pt_shared_lock);

  var_2 = *pt_shared;
  var_2 = SomeOperation_2 (var_2);
  *pt_shared = var_2 ;

  Unlock (pt_shared_lock);
  }
```

Pirouz Bazargan Sabet *February 2021*

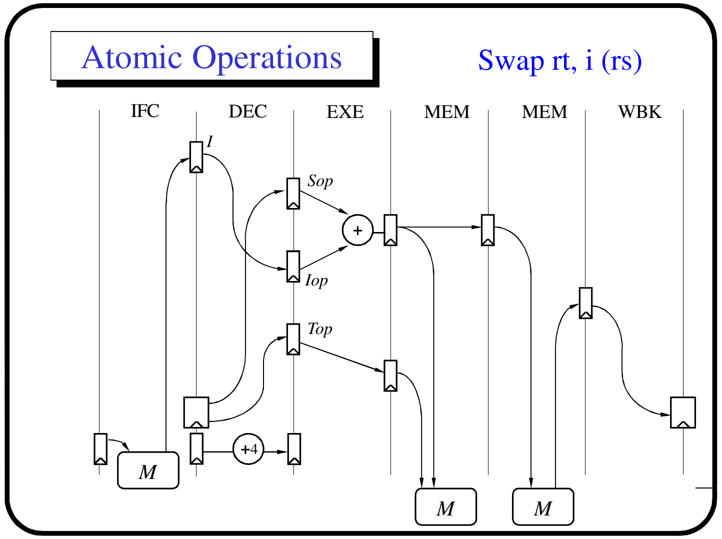


Atomic Operations

Swap : exchanges the contents of a register and a word

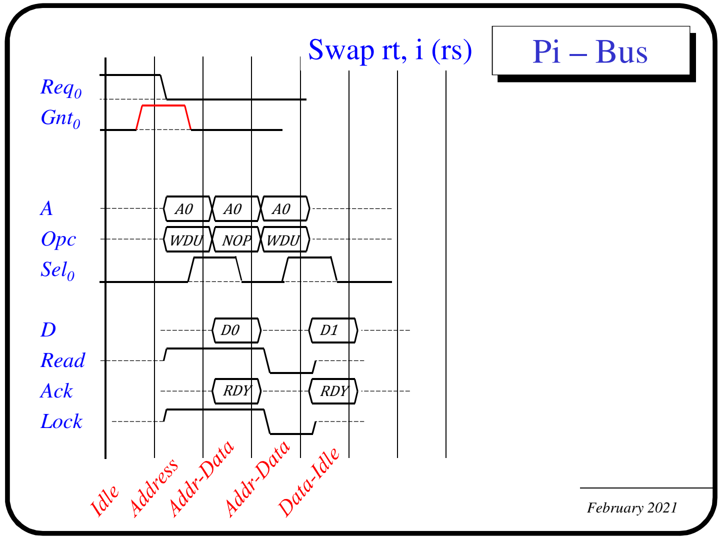
- The *Swap* instruction allows to set up a *lock* that protects the accesses to a shared memory space
- The atomicity is guaranteed by the *Bus*. The Read and the Write accesses should be executed in a single *transaction*

Pirouz Bazargan Sabet *February 2021*



Atomic Operations

Swap rt, i (rs)



Pi – Bus

Swap rt, i (rs)

February 2021

Atomic Operations

Swap *Deadlock !*

```
void SharedOp_1 (int *pt_a      ,
                 int *pt_b      ,
                 int *pt_a_lock,
                 int *pt_b_lock)
  {
  int var_1 ;
  int var_2 ;

  Lock  (pt_a_lock);
  Lock  (pt_b_lock);

  var_a = SomeOp_a1  (pt_a, pt_b);
  var_b = SomeOp_b1  (pt_a, pt_b);

  *pt_a = var_a ;
  *pt_b = var_b ;

  Unlock  (pt_a_lock);
  Unlock  (pt_b_lock);
  }
```

```
void SharedOp_2 (int *pt_a      ,
                 int *pt_b      ,
                 int *pt_a_lock,
                 int *pt_b_lock)
  {
  int var_1 ;
  int var_2 ;

  Lock  (pt_b_lock);
  Lock  (pt_a_lock);

  var_a = SomeOp_a2  (pt_a, pt_b);
  var_b = SomeOp_b2  (pt_a, pt_b);

  *pt_a = var_a ;
  *pt_b = var_b ;

  Unlock  (pt_a_lock);
  Unlock  (pt_b_lock);
  }
```

Atomic Operations

Swap : exchanges the contents of a register and a word

- The *Swap* instruction allows to set up a *lock* that protects the accesses to a shared memory space
- The atomicity is guaranteed by the *Bus*. The Read and the Write accesses should be executed in a single *transaction*
- The protection of each shared memory space requires a proper *Lock*
- *Swap* is not powerful enough
  Is it possible to create a direct *exclusive* access to shared memory word ?

Atomic Operations

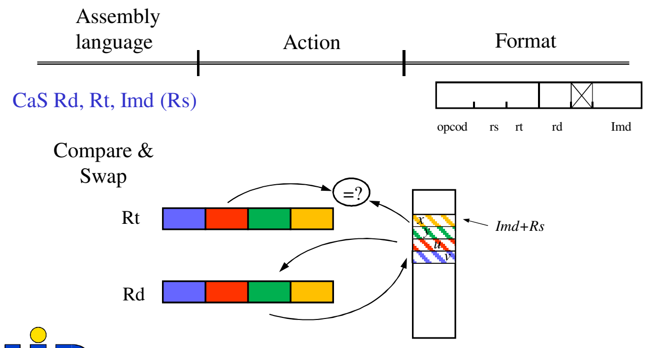
CaS : *Compare and Swap*

Compares the contents of a register with a memory word. If identical, exchanges the contents of a register with the contents of the word

- The *CaS* instruction allows an exclusive access (*Read-Modify-Write*) to a memory word
- If an other *Thread* modifies the word between the Read and the Write, the write operation is rejected

Atomic Operations

CaS



Assembly language | Action | Format

CaS Rd, Rt, Imd (Rs)

Compare & Swap

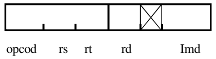
Successful CaS

Atomic Operations

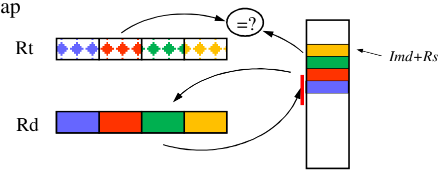
CaS

Assembly language	Action	Format

CaS Rd, Rt, Imd (Rs)



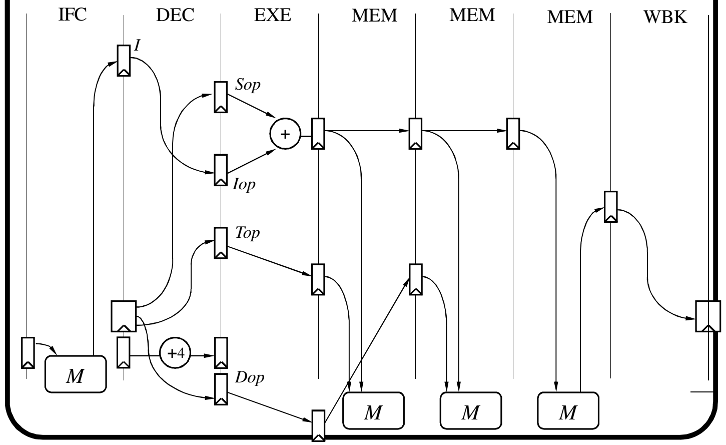
Compare & Swap



Failing *CaS*

Atomic Operations

CaS rd, rt, i (rs)



Atomic Operations

CaS : Compare and Swap

```
void ExclusiveR_M_W (int *pt_shared)
  {
  register int var ;
  register int sav ;

  do
    {
    sav = *pt_shared ;
    var = SomeOp (sav ...);
    }
  while ((var = Cas (var, sav, pt_shared)) != sav);
  }
```

Atomic Operations

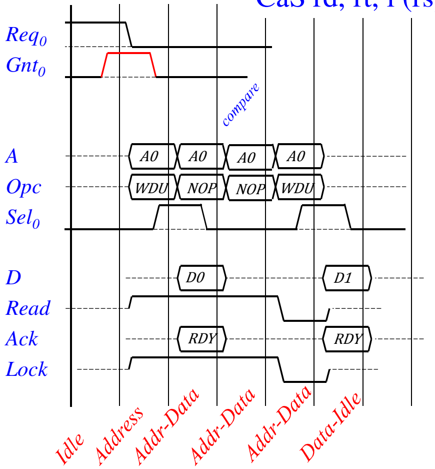
CaS : *Compare and Swap*

Compares the contents of a register with a memory word. If identical, exchanges the contents of a register with the contents of the word

- The *CaS* instruction allows an exclusive access (*Read-Modify-Write*) to a memory word
- If an other *Thread* modifies the word between the Read and the Write, the write operation is rejected
- The comparison is done in the cache
- The atomicity is guaranteed by the *Bus* (a single *transaction*)

CaS rd, rt, i (rs)

Pi – Bus



Successful comparison

Atomic Operations

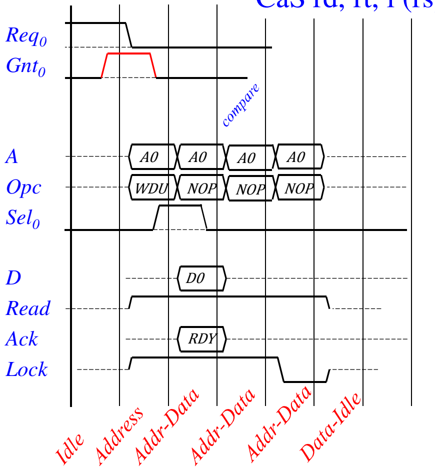
CaS : *Compare and Swap*

Compares the contents of a register with a memory word. If identical, exchanges the contents of a register with the contents of the word

- The *CaS* instruction allows an exclusive access (*Read-Modify-Write*) to a memory word
- If an other *Thread* modifies the word between the Read and the Write, the write operation is rejected
- The comparison is done in the cache
- The atomicity is guaranteed by the *Bus* (a single *transaction*)
- The *ABA* problem

CaS rd, rt, i (rs)

Pi – Bus

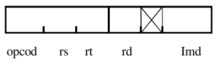


Failing comparison

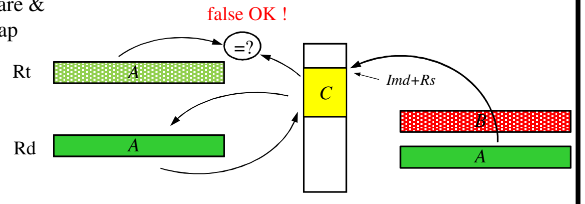
Atomic Operations

CaS : ABA

Assembly language	Action	Format
Cas Rd, Rt, Imd (Rs)		



Compare & Swap



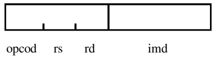
Atomic Operations

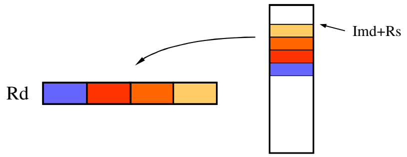
LL : *Linked Load*
SC : *Store Conditional*

LL reads a word in the memory and *reserves* the location
SC writes the word if no other writes have been seen

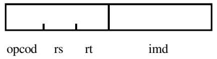
- The pair of instructions *LL, SC* guarantees the atomicity of a *Read-Modify-Write* operation on a word
- If another *Thread* modifies the word in the memory between the *Read* and the *Write*, the *Write* operation is rejected
- An *SC* non preceded by an *LL* is rejected

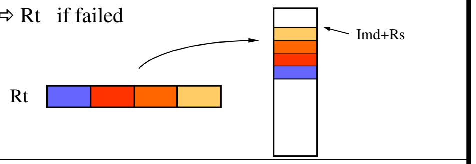
Atomic Operations

Assembly language	Action	Format
Ll Rd, Imd (Rs) Linked Load	4 bytes from memory at address *Imd+Rs* ⇨ Rd the @ is reserved	 I format



Atomic Operations

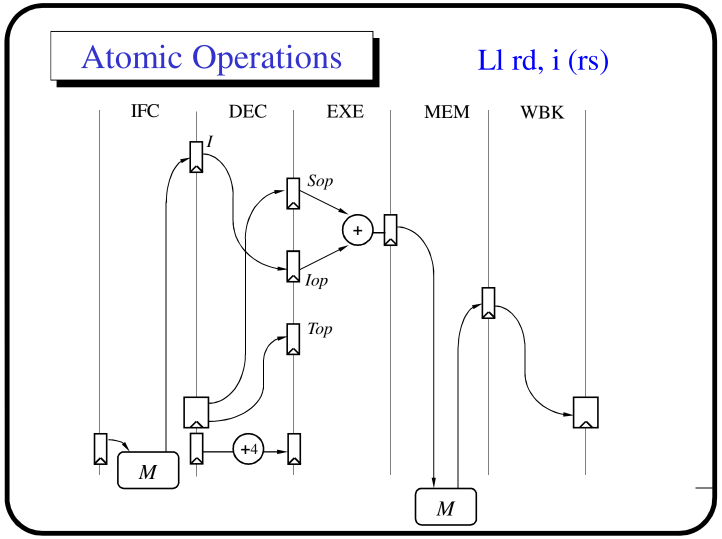
Assembly language	Action	Format
Sc Rt, Imd (Rs) Store Conditional	4 bytes from Rt is written into memory at address *Imd+Rs* only if the reservation exists 1 ⇨ Rt if succeeded 0 ⇨ Rt if failed	 I format



Atomic Operations

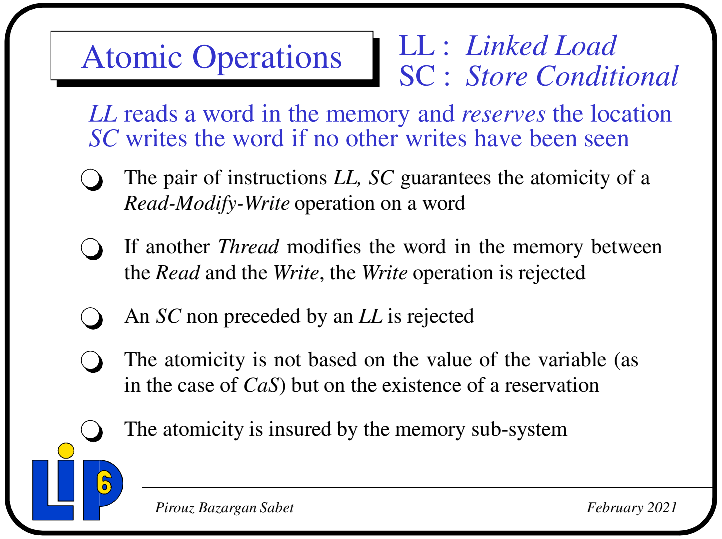
Linked Load / Store Conditional

```
void ExclusiveR_M_W (int *pt_shared)
  {
  register int var ;

  do
    {
    var = LinkedLoad (pt_shared);
    var = SomeOp (...);
    var = StoreConditional (var, pt_shared);
    }
  while (var == 0);
  }
```



Atomic Operations

Ll rd, i (rs)



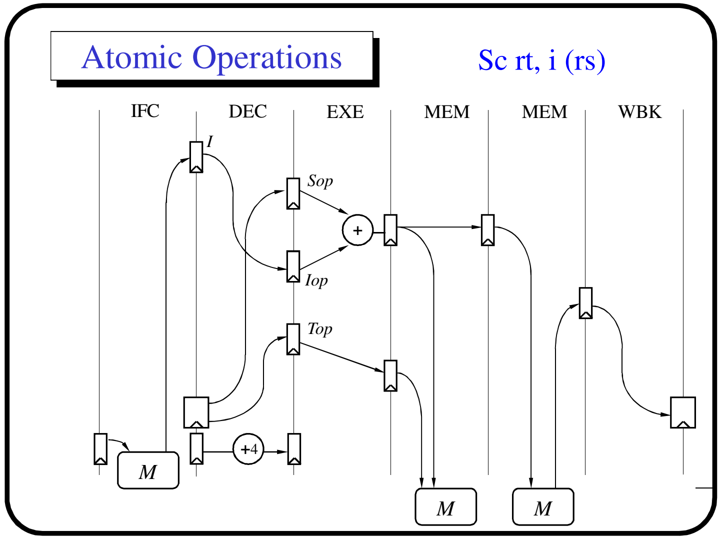
Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

LL reads a word in the memory and *reserves* the location
SC writes the word if no other writes have been seen

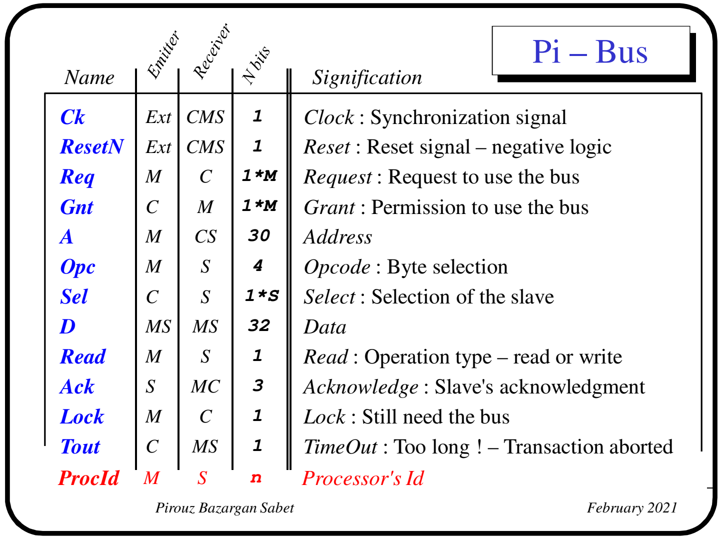
- The pair of instructions *LL, SC* guarantees the atomicity of a *Read-Modify-Write* operation on a word
- If another *Thread* modifies the word in the memory between the *Read* and the *Write*, the *Write* operation is rejected
- An *SC* non preceded by an *LL* is rejected
- The atomicity is not based on the value of the variable (as in the case of *CaS*) but on the existence of a reservation
- The atomicity is insured by the memory sub-system

Pirouz Bazargan Sabet *February 2021*



Atomic Operations

Sc rt, i (rs)



Pi – Bus

Name	*Emitter*	*Receiver*	*N bits*	*Signification*
Ck	*Ext*	*CMS*	1	*Clock* : Synchronization signal
ResetN	*Ext*	*CMS*	1	*Reset* : Reset signal – negative logic
Req	*M*	*C*	1*M	*Request* : Request to use the bus
Gnt	*C*	*M*	1*M	*Grant* : Permission to use the bus
A	*M*	*CS*	30	*Address*
Opc	*M*	*S*	4	*Opcode* : Byte selection
Sel	*C*	*S*	1*S	*Select* : Selection of the slave
D	*MS*	*MS*	32	*Data*
Read	*M*	*S*	1	*Read* : Operation type – read or write
Ack	*S*	*MC*	3	*Acknowledge* : Slave's acknowledgment
Lock	*M*	*C*	1	*Lock* : Still need the bus
Tout	*C*	*MS*	1	*TimeOut* : Too long ! – Transaction aborted
ProcId	*M*	*S*	n	*Processor's Id*

Pirouz Bazargan Sabet *February 2021*

Pi – Bus

Opc	*M*	*S*	*4*	*Opcode* : Byte selection

WDU	0010	*Word Undefined*
HW0	1000	*Half Word 0*
HW1	1010	*Half Word 1*
BY0	1100	*Byte 0*
BY1	1101	*Byte 1*
BY2	1110	*Byte 2*
BY3	1111	*Byte 3*
NOP	0000	*No Operation*
INV		*Invalidate*
LL		*Linked Load*

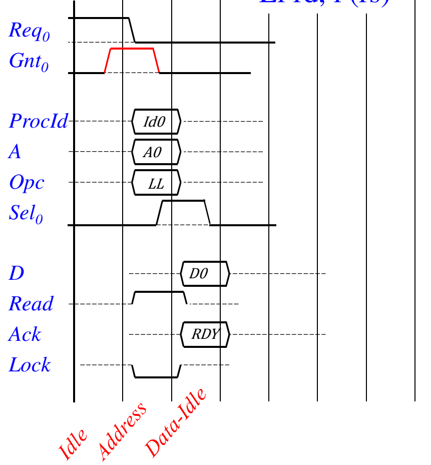
Pi – Bus

Opc	*M*	*S*	*4*	*Opcode* : Byte selection

WDU	0010	*Word Undefined*
HW0	1000	*Half Word 0*
HW1	1010	*Half Word 1*
BY0	1100	*Byte 0*
BY1	1101	*Byte 1*
BY2	1110	*Byte 2*
BY3	1111	*Byte 3*
NOP	0000	*No Operation*
INV		*Invalidate*
LL		*Linked Load*
SC		*Store Conditional*

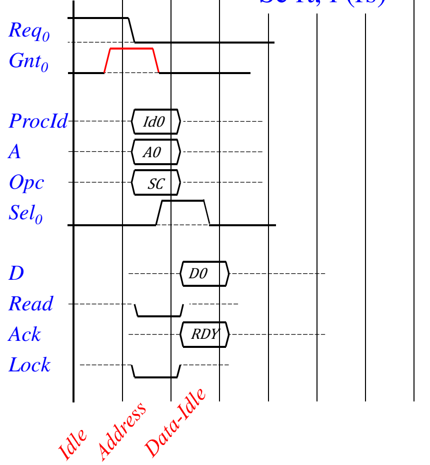
Ll rd, i (rs)

Pi – Bus



Sc rt, i (rs)

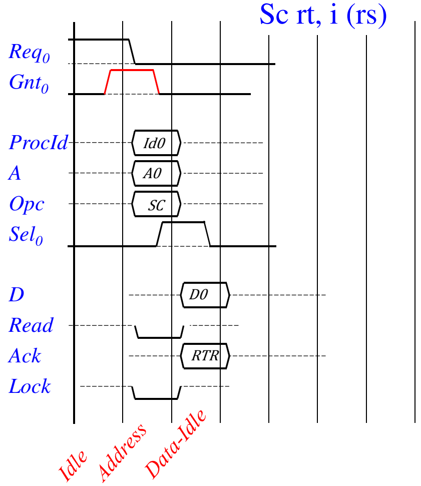
Pi – Bus



Successful *SC*

Pi – Bus

Sc rt, i (rs)

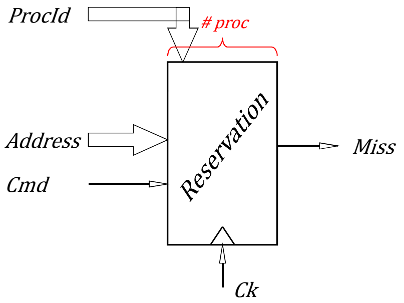


Failing *SC*

Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

Reservation Associative Array



Cmd : Write , Check–Remove , Invalidate

Atomic Operations

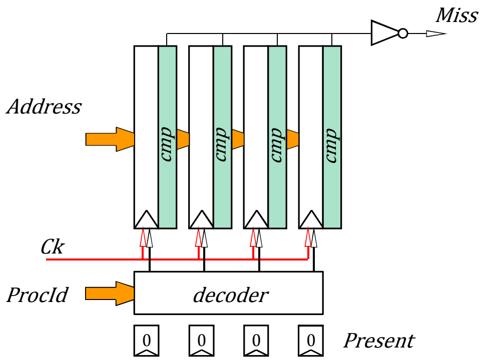
LL : *Linked Load*
SC : *Store Conditional*

A *Reservation Table* should be implemented in the memory

- A Table should be implemented in the memory sub-system to hold the reservations (*ProcId, Address*)
- A *Linked Load* creates a new reservation in the Table
- A single record per processor in the Table
  Table size = Number of processors
- An *Store Conditional* succeeds if a reservation is present in the Table
- Any write to an address invalidates all the reservations for that address

Atomic Operations

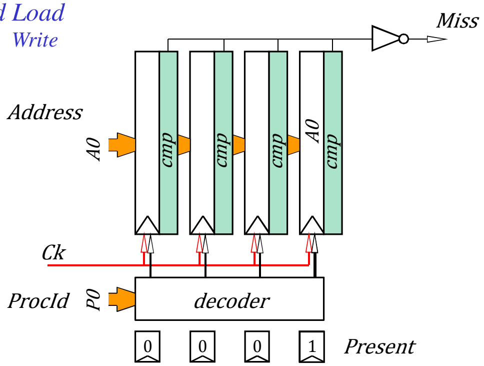
LL : *Linked Load*
SC : *Store Conditional*



Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

Linked Load
Write

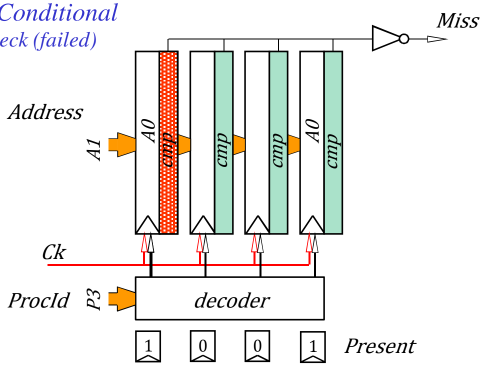


Pirouz Bazargan Sabet *February 2021*

Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

Store Conditional
Check (failed)

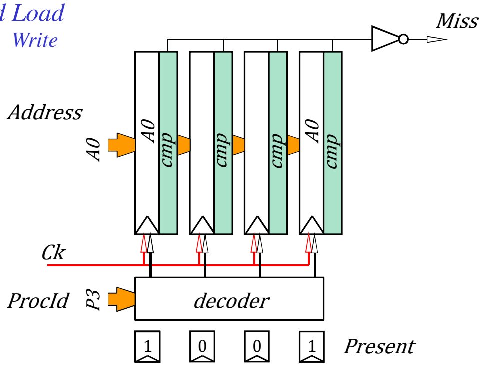


Pirouz Bazargan Sabet *February 2021*

Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

Linked Load
Write

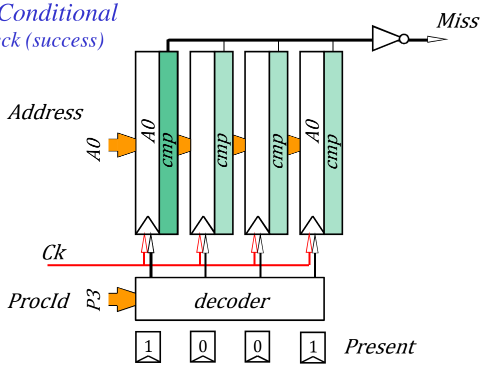


Pirouz Bazargan Sabet *February 2021*

Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

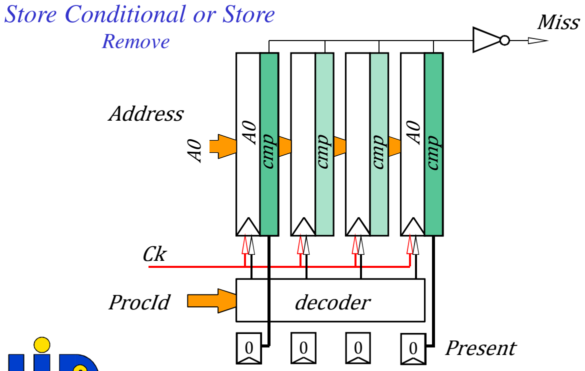
Store Conditional
Check (success)



Pirouz Bazargan Sabet *February 2021*

Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*



Atomic Operations

LL : *Linked Load*
SC : *Store Conditional*

LL reads a word in the memory and *reserves* the location
SC writes the word if no other writes have been seen

- The pair of instructions *LL*, *SC* guarantees the atomicity of a *Read-Modify-Write* operation on a word
- If another *Thread* modifies the word in the memory between the *Read* and the *Write*, the *Write* operation is rejected
- An *SC* non preceded by an *LL* is rejected
- The atomicity is not based on the value of the variable (as in the case of *CaS*) but on the existence of a reservation
- The atomicity is insured by the memory sub-system
- The implementation becomes more difficult as the number of processors increases