Atomic CISC Instructions

- **atomic read-modify-write (RMW) instructions**
  - test a certain condition σ referring to some memory addresses
  - if σ holds, modify (some of) the memory addresses
  - particular instances
- particular instances
  - atomic test-and-set instructions
  - atomic fetch-and-increment
  - atomic fetch-and-decrement

$\rightsquigarrow$ old fashioned, not in spirit of modern load/store architectures

Load/Store RMW Primitives

- load/store architectures offer other RMW primitives to guarantee exclusive access to a memory address
- instruction pairs $\mathcal{I}_1$, $\mathcal{I}_2$ are used
  - $\mathcal{I}_1$ starts a critical code region by noting some information about the memory address that has to be protected
  - critical code instructions between $\mathcal{I}_1$ and $\mathcal{I}_2$
  - $\mathcal{I}_2$ checks by consulting information written by $\mathcal{I}_1$ whether the execution from $\mathcal{I}_1$ to $\mathcal{I}_2$ had exclusive access to the considered memory address
  - if so, proceed execution
  - otherwise do something else (usually retry)

Load/Store RMW Primitives

- how can instructions $\mathcal{I}_1$ and $\mathcal{I}_2$ be implemented?
- consider MIPS instruction set
- $\mathcal{I}_1 :\equiv$ LL rt,c(rs) is called load linked
  - load Mem[c + Reg[rs]] to Reg[rt]
  - store address c + Reg[rs] in a special register L
- $\mathcal{I}_2 \equiv$ SC rt,c(rs) is called store conditional
  - if $L \neq 0$, store Reg[rt] to memory address Mem[c + Reg[rs]] and set Reg[rt] := 1
  - if $L = 0$, set Reg[rt] := 0

Semantics of Load/Store RMW Primitives

- $\mathcal{I}_1 :\equiv$ LL rt,c(rs) does the following

  next(Reg[rt]) := Mem[c + Reg[rs]];
  next(L) := c + Reg[rs];

- cache controller executes the following in each cycle

  if ($\mathit{WriteOnBus}$&($L = \mathit{AdrOnBus}$)) **next**(L) := 0;

- finally, $\mathcal{I}_2 \equiv$ SC rt,c(rs) does the following

```
if (L ≠ 0) {
    next(Mem[c + Reg[rs]]) = Reg[rt];
    next(Reg[rt]) := 1;
}else next(Reg[rt]) := 0;
```

$\rightsquigarrow$ the cache controller helps $\mathcal{I}_2$ to detect if there has been a write to Mem[L] between the execution of $\mathcal{I}_1$ and $\mathcal{I}_2$

Implementing Other Atomic Primitives

- example: exchange contents of Reg[4] and Mem[Reg[1]]

```
try:  OR R3,R4,R0     ; Reg[3] := Reg[4]
      LL R2,0(R1)     ; load and protect Mem[Reg[1]]
      SC R3,0(R1)     ; try Mem[Reg[1]] := Reg[3]
      BEQZ R3,try     ; if not atomically executed, retry
      ADD R4,R2,R0    ; Reg[4] := Reg[2]
```

Implementing Other Atomic Primitives

- example: fetch-and-increment memory location

```
try:  LL R2,0(R1)       ; load and protect Mem[Reg[1]]
      ADDIU R3,R2,#1    ; increment Reg[2]
      SC R3,0(R1)       ; try Mem[Reg[1]] := Reg[3]
      BEQZ R3,try       ; if not atomically executed, retry
```

Implementing Other Atomic Primitives

- example: protect critical region by a lock variable Mem[Reg[1]]

```
try:  ADDI R3,R0,#1   ; Reg[3] := 1
      LL R2,0(R1)     ; load and protect Mem[Reg[1]]
      SC R3,0(R1)     ; try Mem[Reg[1]] := Reg[3]
      BEQZ R3,try     ; if not atomically executed, retry
      BNEZ R2,try     ; if region is locked, retry
```

- process can proceed if lock Mem[Reg[1]] is zero
- after critical code is executed, lock Mem[Reg[1]] must be reset
- note serialization of load/stores due to bus arbitration

Barrier Synchronization

```
lock(counterlock);
if counter=0 then
  release:=0
end
counter := counter+1;
unlock(counterlock);
if counter=N then
  counter := 0;
  release := 1
else
  await(release=1)
end
```

- left hand side is the code of one process
- variable *counterlock* protects variable *counter*, so that it can be atomically incremented
- *count* is the number of processes that have reached the barrier
- *release* = 1 allows the processes to pass the barrier
- *release* is set to 1 iff all N processes have reached the barrier

Problem: Barrier Synchronization in a Loop

- the previous code must not be used in loop bodies
- the following could happen:
  - if all processes reached the barrier, *release* is set to 1
  - process P_1 goes ahead and iterates the loop body, while the other processes do not proceed their execution
  - in the worst case, P_1 reaches the barrier again, before the last process of the previous iteration has passed the barrier

⇝ a fast process can trap a slow process in the barrier by resetting *release*

Sense-Reversing Barrier Synchronization

```
evenodd := ¬ evenodd;
lock(counterlock);
counter := counter+1;
if counter=N then
  counter := 0;
  release := evenodd
end;
unlock(counterlock);
await(release=evenodd)
```

- sense-reversing barrier synchronization solves the problem
- alternatingly, the processes wait either until *release* = 0 or *release* = 1 holds
- previous and new barrier synchronization due to a loop are no longer mixed up
- still possible that one process reaches barrier again before another has even left it
- however, this time the slow one can still leave the barrier

Bus Arbitration

- several processors may wish to access the bus

⇝ arbitration required to manage access to the shared bus

- arbiters are implemented as hardware circuits (bus controller)
- simplest form: arbitration with static priorities
  - processors $P_1, \ldots, P_n$ have priorities $1, \ldots, n$
  - processor with highest priority yields the bus
  - problem: unfair arbitration P_n may always use the bus

Bus Arbitration

- fairness can be obtained by dynamic priorities
- simplest form: token ring
  - a token is sent from P_i to $P_{(i \bmod n)+1}$
  - processor P_i with token yields the bus, if P_i wants to do so
  - problem: if only one P_i wants access to the bus, it may have to wait n cycles to receive the access, although the bus is available

⇝ combination of static and dynamic priorities recommended

Bus Arbitration

- combination of static and dynamic priorities
  - again, send a token from P_i to $P_{(i \bmod n)+1}$
  - assume P_i currently has the token
  - P_j is granted access to the bus iff
    - either $i \leq j$ and no P_k with $i \leq k < j$ requests the bus
    - or neither a P_k with $i \leq k \leq n$ nor a P_k with $1 \leq k < j$ requests the bus
  - ⇝ in principle, priorities are changed by token moves
- ⇝ fair and efficient arbitration
- in the worst case, still $n - 1$ cycles have to be awaited for access

Memory Consistency Models

- initially, $x_1 = x_2 = 0$ are both in the caches of P_1 and P_2

processor P_1	processor P_2
$y_1 := 0;$ **do** $y_1 := y_1 + 1;$ $x_1 := 1$ **while** $(x_2 \neq 0)$	$y_2 := 0;$ **do** $y_2 := y_2 + 1;$ $x_2 := 1;$ **while** $(x_1 \neq 0)$

- the messages of the writes $x_i := 1$ are sent by the directories
- assume each P_i proceeds execution (with outdated value x_i)
- ⇝ $(y_1, y_2) \in \{(1,1),(1,2),(2,1),(2,2)\}$ possible!

Memory Consistency Models

- memory consistency models determine the semantics of parallel execution
- several consistency models are used by multiprocessors
  - sequential consistency
  - weak consistency
  - release consistency
- depending on the consistency model, updates from one processor may not be immediately visible to all processors
- instead, other processors may notice the update only after some time or after some explicit synchronization operations

Sequential Consistency

- **sequential consistency**
  - load/stores of different processes may be arbitrarily interleaved
  - however, execution of a load/store can only be started if other load/store executions terminated and all updates are acknowledged
- this holds trivially for snooping based protocols
- however, directory based protocols require acknowledge messages to implement sequential consistency in order to know that a previous load/store is done
- using sequential consistency, the previous program can not end with $y_1 = y_2 = 2$

Weak Consistency

- **sequential consistency is often too inefficient**
  - since it requires to send a lot of acknowledge messages
- ⇝ weaker consistency model have been considered
  - several load/stores may be pending in messages of the network
  - explicit synchronization mechanisms guarantee that all load/stores are done
  - memory consistency is only given after such synchronization steps
- further references [2, 49, 63]

Steinke-Nutt Hierarchy of Memory Models

- Steinke and Nutt [96] present a categorization of weak memory models
- to this end, a fixed number of processes $\mathcal{P} = \{p_1, \ldots, p_m\}$ working on shared variables $\mathcal{V} = \{x_1, \ldots, x_n\}$ are considered
- each process $p \in \mathcal{P}$ performed a set of read/write actions: $\alpha_1 \preceq_p \ldots \preceq_p \alpha_{n_p}$
- note: each $\preceq_p$ is transitive, but we only draw a few lines
- different memory consistency models are defined on this basis
- also, [96] defined a lattice of memory consistency models by orthogonal consistency properties in a systematic way